Justine John "JJ" A. Serdoncillo
October 16, 2023

EE 5561: Image Processing

ResNet9 using PyTorch: HW 4

Problem #1: Special Convolution Block

For this problem, I actually had some troubles because I didn't seem to understand at first what the "add" block meant in the homework guidelines. After figuring out that I needed to add the residual to the current value, I was getting some errors that my tensor sizes weren't matching up.

```
Residual in layer 1 has size: torch.Size([100, 64, 12, 12])
After Residual in layer 1 has size: torch.Size([100, 64, 8, 8])

RuntimeError: The size of tensor a (8) must match the size of tensor
    b (12) at non-singleton dimension 3
```

I was able to resolve this by adding a stride and padding to the convolution if it is inside the special residual block. Overall my final code for the conv_block is as shown below

```python
def conv_block(in_channels, out_channels, pool=False, res=False, pool_size=2):
    layers = []
    if res:
        layers.append(nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding
    =1))
    else:
        layers.append(nn.Conv2d(in_channels, out_channels, kernel_size=3))
    layers.append(nn.BatchNorm2d(out_channels))
    layers.append(nn.ReLU())
    if pool:
        layers.append(nn.MaxPool2d(pool_size))
    return nn.Sequential(*layers)
```

Problem #2: ResNet9 Implementation

For this next problem, my main problem was I wasn't sure what the size of the flattened layer was. I was able to remedy this by having a print function to get the shape after applying the flattened layer. From this I started my fully connected layer with a size of 1024.

```
After Flatten in classifier layer has size: torch.Size([100, 1024])
```

For the fully connected layer as well, I wanted to switch things up by adding multiple linear and relu blocks which might improve its performance. I will also compare the results if I just used one linear layer. I also implemented the addition of the res block in the forward propagation and the overall ResNet9 class can be seen below.

```python
class ResNet9(nn.Module):
    def __init__(self, in_channels, num_classes):
        super().__init__()
        # Preparation Layer
        self.prep = conv_block(in_channels, 32)
        # Layer 1 w/ Res Blocks
        self.layer1_conv = conv_block(32, 64, pool=True)
        self.layer1_res  = nn.Sequential(conv_block(64, 64, res=True),
                                         conv_block(64, 64, res=True))
        # Layer 2
        self.layer2 = conv_block(64, 128)
        # Layer 1 w/ Res Blocks
        self.layer3_conv = conv_block(128, 256, pool=True)
        self.layer3_res  = nn.Sequential(conv_block(256, 256, res=True),
                                         conv_block(256, 256, res=True))
        # Classifier
        self.classifier = nn.Sequential(nn.MaxPool2d(2),
                                        nn.Flatten(),
                                        nn.Linear(1024,10))
                                        #nn.Linear(1024,256),
                                        #nn.ReLU(),
                                        #nn.Linear(256,64),
                                        #nn.ReLU(),
                                        #nn.Linear(64,10),
                                        #nn.ReLU())
        self.shape_tester = nn.Sequential(nn.MaxPool2d(2),
                                        nn.Flatten())
        self.fc = nn.Linear(1024,10)

        # Params
        self.num_classes = num_classes

    def class3(self, size1d, num_classes):
        return nn.Linear(size1d, num_classes)

    def forward(self, x):

        # Preparation Layer
        y = self.prep(x)

        # Layer 1 w/ Res Blocks
        y1 = self.layer1_conv(y)
        #print(f"Residual in layer 1 has size: {y1.shape}") # has 12,12
        #y = self.layer1_res(y1) + y1
        y2 = self.layer1_res(y1)
        #print(f"After Residual in layer 1 has size: {y2.shape}") # has 8,8
        y = y1 + y2

        # Layer 2
        y = self.layer2(y)

        # Layer 3 w/ Res Blocks
        y1 = self.layer3_conv(y)
        y = self.layer3_res(y1) + y1
```

```python

        # Classifier
        y = self.shape_tester(y)
        #print(f"After Flatten in classifier layer has size: {y.shape}")
        y = self.fc(y)

        return y
```

Problem #3: ResNet9 Training and Testing

After the creation of ResNet9, it was then trained with the Cifar MNIST dataset which meant that it would take in 1 channel and have 10 outputs, one for each of the digits. It was trained for 10 epochs with a learning rate and optimizer same as the one shown in class.

After 10 epochs composed of a total of 3000 iterations, the table below shows the Validation and Test loss and accuracy at the end of each epoch. IT can be seen that there is a difference at the first few epochs between the two ResNet9s but in the end they have a difference of only around 0.2%. Overall, this ResNet9 architecture is crazy good knowing that it can get to this level of accuracy with such a small amount of training epochs.

Epoch	Normal	Special
1	96.2	97.4
2	98.3	99.0
3	98.4	98.5
4	98.7	98.7
5	99.0	98.7
6	99.0	99.0
7	99.1	99.2
8	98.9	99.2
9	99.1	99.3
10	99.2	99.3

Table 1: Validation Accuracy % for both ResNet9s

Normal	Special
99.1	99.3

Table 2: Final Test Accuracy % for both ResNet9s

At the same time, the loss from each image test and its mean are plotted below. It can be seen that as the training moves forward, there is a decrease of the loss until it somewhat stagnates at near 0.

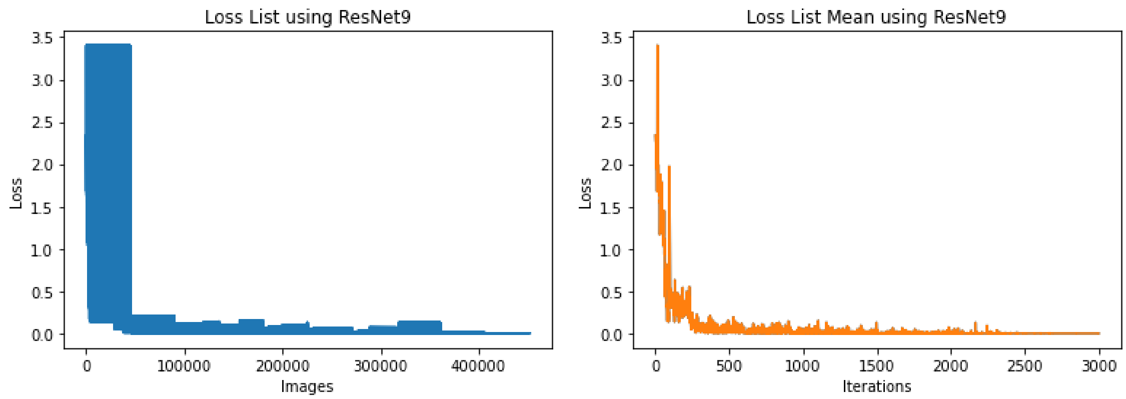

Figure 1: Loss List and Loss List Mean using ResNet9

I was also testing it on various example images and the predicted labels as seen in the figure below where it was succesful in predicting all of the images.

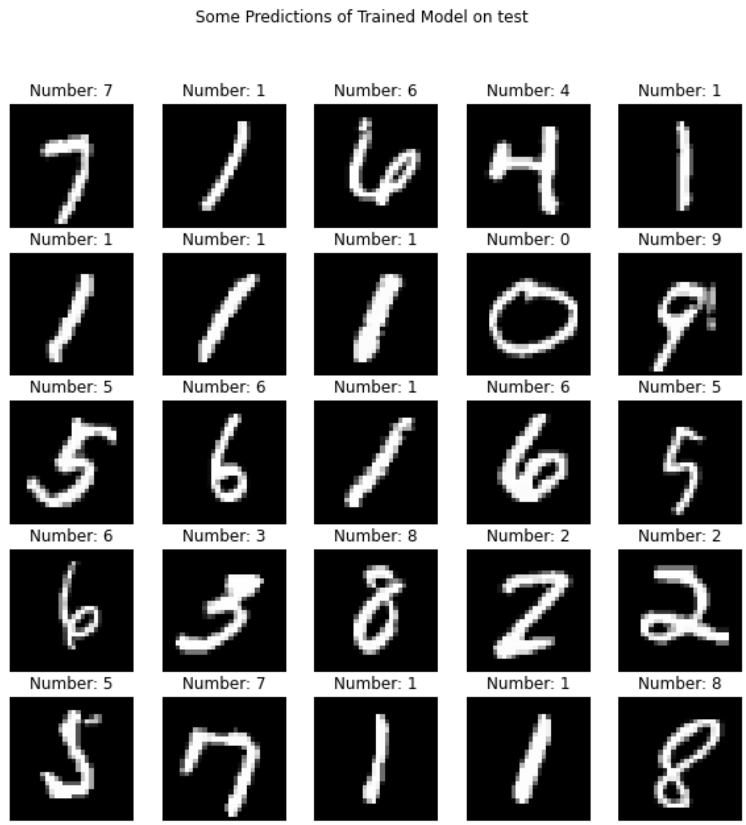

Figure 2: Example Photos Model testing

Overall, I would say the implementation was a success and using this ResNet9 was pretty cool in learning and applying different architectures in the realm of machine learning. I hope that I am able to use this more in the future and maybe in our final project!

Appendix

Complete ResNet9 Creation, Training and Testing

```python
'''
Justine Serdoncillo
EE 5561 - Image Processing
Problem Set 4
December 6, 2023
'''

# %% Problem Statement
"""
In this exercise, you will implement a ResNet structure and will
use it for MNIST database classification. You can use the given PyTorch tutorial for training
and replace the model with ResNet by using the following ResNet9.
"""


# %% import libraries
import torch
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt
import torch.nn as nn
import numpy as np

# %% LOAD THE MNIST DATASET
data_full = datasets.MNIST(root = 'data',
                           train = True,
                           transform = ToTensor(),
                           download = True)

# visualize the data
plt.imshow(data_full.data[0])
plt.imshow(data_full.data[0], cmap='gray')
plt.imshow(data_full.data[61], cmap='gray')

# access the data
data1 = data_full.data[0]
plt.imshow(data1, cmap='gray')

# visualize multiple images
figure = plt.figure(figsize=(10,10))
cols, rows = 5,5

for i in range(1, cols*rows+1):

    idx = torch.randint(len(data_full), size=(1,)).item()

    img, label = data_full[idx]

    figure.add_subplot(rows,cols,i)
    plt.title('Number: ' + str(label))
    plt.axis('off')
    plt.imshow(img.squeeze(), cmap='gray')

plt.show()


# %% SPLIT THE DATASET
# train, test, validation seperation
train_data, test_data, valid_data = torch.utils.data.random_split(data_full, [30000, 10000, 20000])

batch_size = 100

loaders = {'train': torch.utils.data.DataLoader(train_data,
                                                batch_size = batch_size,
                                                shuffle=True),
```

```
            'test': torch.utils.data.DataLoader(test_data,
                                                 batch_size = batch_size,
                                                 shuffle=True),
            'valid': torch.utils.data.DataLoader(valid_data,
                                                 batch_size = batch_size,
                                                 shuffle=True)}
"""
#visualize the dictionary
train_part = loaders.get('train')
data2 = train_part.dataset
element1 = data2[0][0].squeeze()
plt.imshow(element1, cmap='gray')
"""


# %%
##########################
## EE 5561 Assignment 4 ##
## JJ Personal Code     ##
##########################
def conv_block(in_channels, out_channels, pool=False, res=False, pool_size=2):
    layers = []
    if res:
        layers.append(nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding
    =1))
    else:
        layers.append(nn.Conv2d(in_channels, out_channels, kernel_size=3))
    layers.append(nn.BatchNorm2d(out_channels))
    layers.append(nn.ReLU())
    if pool:
        layers.append(nn.MaxPool2d(pool_size))
    return nn.Sequential(*layers)


class ResNet9(nn.Module):
    def __init__(self, in_channels, num_classes):
        super().__init__()
        # Preparation Layer
        self.prep = conv_block(in_channels, 32)
        # Layer 1 w/ Res Blocks
        self.layer1_conv = conv_block(32, 64, pool=True)
        self.layer1_res  = nn.Sequential(conv_block(64, 64, res=True),
                                          conv_block(64, 64, res=True))
        # Layer 2
        self.layer2 = conv_block(64, 128)
        # Layer 1 w/ Res Blocks
        self.layer3_conv = conv_block(128, 256, pool=True)
        self.layer3_res  = nn.Sequential(conv_block(256, 256, res=True),
                                          conv_block(256, 256, res=True))
        # Classifier
        self.classifier = nn.Sequential(nn.MaxPool2d(2),
                                        nn.Flatten(),
                                        nn.Linear(1024,10))
                                        #nn.Linear(1024,256),
                                        #nn.ReLU(),
                                        #nn.Linear(256,64),
                                        #nn.ReLU(),
                                        #nn.Linear(64,10),
                                        #nn.ReLU())
        self.shape_tester = nn.Sequential(nn.MaxPool2d(2),
                                        nn.Flatten())
        self.fc = nn.Linear(1024,10)

        # Params
        self.num_classes = num_classes

    def class3(self, size1d, num_classes):
        return nn.Linear(size1d, num_classes)

    def forward(self, x):

        # Preparation Layer
```

```
        y = self.prep(x)

        # Layer 1 w/ Res Blocks
        y1 = self.layer1_conv(y)
        #print(f"Residual in layer 1 has size: {y1.shape}") # has 12,12
        #y = self.layer1_res(y1) + y1
        y2 = self.layer1_res(y1)
        #print(f"After Residual in layer 1 has size: {y2.shape}") # has 8,8
        y = y1 + y2

        # Layer 2
        y = self.layer2(y)

        # Layer 3 w/ Res Blocks
        y1 = self.layer3_conv(y)
        y = self.layer3_res(y1) + y1

        # Classifier
        y = self.shape_tester(y)
        #print(f"After Flatten in classifier layer has size: {y.shape}")
        y = self.fc(y)

        return y

# %% SET UP THE MODEL

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = ResNet9(1, 10)
model.to(device)

# define the loss function
criterion = nn.CrossEntropyLoss()

# define the optimizer
learning_rate = 0.1
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

# define epoch number
num_epochs = 10

# initialize the loss
loss_list = []
loss_list_mean = []


# %% TRANING STARTS HERE

iter = 0
for epoch in range(num_epochs):

    print('Epoch: {}'.format(epoch))

    loss_buff = []

    for i, (images,labels) in enumerate(loaders['train']):

        # getting the images and labels from the training dataset
        images = images.requires_grad_().to(device)
        labels = labels.to(device)

        # clear the gradients
        optimizer.zero_grad()

        # call the NN
        outputs = model(images) # torch.Size([100, 1, 28, 28])

        # loss calculation
        loss = criterion(outputs, labels)
        loss_buff = np.append(loss_buff, loss.item())

        # back propagation
        loss.backward()
```

```

        loss_list = np.append(loss_list, (loss_buff))

        #update parameters
        optimizer.step()

        iter += 1

        if iter % 10 == 0:
            print('Iterations: {}'.format(iter))

####### VALIDATION PART############

        if iter % 100 == 0:

            # accuracy
            correct = 0
            total = 0

            for i, (images,labels) in enumerate(loaders['valid']):

                # getting the images and labels from the training dataset
                images = images.to(device)
                labels = labels.to(device)

                # clear the gradients
                optimizer.zero_grad()

                # call the NN
                outputs = model(images)

                # get the predictions
                _, predicted = torch.max(outputs.data, 1)

                total += labels.size(0)

                correct += (predicted == labels).sum()

            accuracy = 100 * correct / total

            print('Iterations: {} Loss: {}. Validation Accuracy: {}'.
                  format(iter, loss.item(), accuracy))

        loss_list_mean = np.append(loss_list_mean, (loss.item()))
        ################################

#visualize the loss
plt.plot(loss_list)
plt.plot(loss_list_mean)

fig, ax = plt.subplots()
ax.plot(loss_list)
ax.set_title("Loss List using ResNet9")
ax.set_xlabel("Images")
ax.set_ylabel("Loss")

fig, ax = plt.subplots()
ax.plot(loss_list_mean)
ax.set_title("Loss List Mean using ResNet9")
ax.set_xlabel("Iterations")
ax.set_ylabel("Loss")
ax.plot(loss_list_mean)

# %% TEST PART############
correct = 0
total = 0

for i, (images,labels) in enumerate(loaders['test']):

    # getting the images and labels from the training dataset
    images = images.to(device)
    labels = labels.to(device)
```

```

    # clear the gradients
    optimizer.zero_grad()

    # call the NN
    outputs = model(images)

    # get the predictions
    _, predicted = torch.max(outputs.data, 1)

    total += labels.size(0)

    correct += (predicted == labels).sum()

accuracy = 100 * correct / total

print('Iterations: {} Loss: {}. Test Accuracy: {}'.format(iter, loss.item(), accuracy))

# %%

# Save:
torch.save(model.state_dict(), "weights_1Linear.h5")


# Load:
device = torch.device('cpu')
model = ResNet9(1, 10)
model.to(device)
model.load_state_dict(torch.load("weights_1Linear.h5"))
model.eval()
# visualize multiple images
fig, ax = plt.subplots(5,5,figsize=(10,10))
fig.suptitle("Some Predictions of Trained Model on test ")
cols, rows = 5,5

for i in range(5):
    for j in range(5):
        for test_images, test_labels in loaders['test']:
            img = test_images[0]
            img.unsqueeze_(1)
        img = img.to(device)
        optimizer.zero_grad()
        #print(img.shape)
        outputs = model(img)
        _, predicted = torch.max(outputs.data, 1)

        ax[i,j].set_title('Number: ' + str(predicted.numpy()[0]))
        ax[i,j].axis('off')
        ax[i,j].imshow(img.squeeze().numpy(), cmap='gray')
```

Output from normal ResNet9

```
Epoch: 0
Iterations: 10
Iterations: 20
Iterations: 30
Iterations: 40
Iterations: 50
Iterations: 60
Iterations: 70
Iterations: 80
Iterations: 90
Iterations: 100
Iterations: 100 Loss: 0.8954293131828308. Validation Accuracy: 73.26499938964844
Iterations: 110
Iterations: 120
Iterations: 130
Iterations: 140
Iterations: 150
Iterations: 160
Iterations: 170
Iterations: 180
Iterations: 190
Iterations: 200
Iterations: 200 Loss: 0.14886337518692017. Validation Accuracy: 95.35499572753906
Iterations: 210
Iterations: 220
Iterations: 230
Iterations: 240
Iterations: 250
Iterations: 260
Iterations: 270
Iterations: 280
Iterations: 290
Iterations: 300
Iterations: 300 Loss: 0.1941969096660614. Validation Accuracy: 96.18999481201172
Epoch: 1
Iterations: 310
Iterations: 320
Iterations: 330
Iterations: 340
Iterations: 350
Iterations: 360
Iterations: 370
Iterations: 380
Iterations: 390
Iterations: 400
Iterations: 400 Loss: 0.1246100440621376. Validation Accuracy: 97.66999816894531
Iterations: 410
Iterations: 420
Iterations: 430
Iterations: 440
Iterations: 450
Iterations: 460
Iterations: 470
Iterations: 480
Iterations: 490
Iterations: 500
Iterations: 500 Loss: 0.08373874425888062. Validation Accuracy: 97.48500061035156
Iterations: 510
Iterations: 520
Iterations: 530
Iterations: 540
Iterations: 550
Iterations: 560
Iterations: 570
Iterations: 580
Iterations: 590
Iterations: 600
Iterations: 600 Loss: 0.05688638240098953. Validation Accuracy: 98.30999755859375
Epoch: 2
Iterations: 610
```

```
Iterations: 620
Iterations: 630
Iterations: 640
Iterations: 650
Iterations: 660
Iterations: 670
Iterations: 680
Iterations: 690
Iterations: 700
Iterations: 700 Loss: 0.03462041541934O134. Validation Accuracy: 98.3999389648438
Iterations: 710
Iterations: 720
Iterations: 730
Iterations: 740
Iterations: 750
Iterations: 760
Iterations: 770
Iterations: 780
Iterations: 790
Iterations: 800
Iterations: 800 Loss: 0.028844986110925674. Validation Accuracy: 98.48500061035156
Iterations: 810
Iterations: 820
Iterations: 830
Iterations: 840
Iterations: 850
Iterations: 860
Iterations: 870
Iterations: 880
Iterations: 890
Iterations: 900
Iterations: 900 Loss: 0.02131034806370735. Validation Accuracy: 98.36499786376953
Epoch: 3
Iterations: 910
Iterations: 920
Iterations: 930
Iterations: 940
Iterations: 950
Iterations: 960
Iterations: 970
Iterations: 980
Iterations: 990
Iterations: 1000
Iterations: 1000 Loss: 0.07528997957706451. Validation Accuracy: 98.56999969482422
Iterations: 1010
Iterations: 1020
Iterations: 1030
Iterations: 1040
Iterations: 1050
Iterations: 1060
Iterations: 1070
Iterations: 1080
Iterations: 1090
Iterations: 1100
Iterations: 1100 Loss: 0.013861780986189842. Validation Accuracy: 98.7049942016016
Iterations: 1110
Iterations: 1120
Iterations: 1130
Iterations: 1140
Iterations: 1150
Iterations: 1160
Iterations: 1170
Iterations: 1180
Iterations: 1190
Iterations: 1200
Iterations: 1200 Loss: 0.06764683872461319. Validation Accuracy: 98.71499633789062
Epoch: 4
Iterations: 1210
Iterations: 1220
Iterations: 1230
Iterations: 1240
Iterations: 1250
```

```
Iterations: 1260
Iterations: 1270
Iterations: 1280
Iterations: 1290
Iterations: 1300
Iterations: 1300 Loss: 0.015819817781448364. Validation Accuracy: 98.59500122070312
Iterations: 1310
Iterations: 1320
Iterations: 1330
Iterations: 1340
Iterations: 1350
Iterations: 1360
Iterations: 1370
Iterations: 1380
Iterations: 1390
Iterations: 1400
Iterations: 1400 Loss: 0.008826137520372868. Validation Accuracy: 98.41999816894531
Iterations: 1410
Iterations: 1420
Iterations: 1430
Iterations: 1440
Iterations: 1450
Iterations: 1460
Iterations: 1470
Iterations: 1480
Iterations: 1490
Iterations: 1500
Iterations: 1500 Loss: 0.004896394908428192. Validation Accuracy: 99.02999877929688
Epoch: 5
Iterations: 1510
Iterations: 1520
Iterations: 1530
Iterations: 1540
Iterations: 1550
Iterations: 1560
Iterations: 1570
Iterations: 1580
Iterations: 1590
Iterations: 1600
Iterations: 1600 Loss: 0.016535509377717972. Validation Accuracy: 99.01499938964844
Iterations: 1610
Iterations: 1620
Iterations: 1630
Iterations: 1640
Iterations: 1650
Iterations: 1660
Iterations: 1670
Iterations: 1680
Iterations: 1690
Iterations: 1700
Iterations: 1700 Loss: 0.05973479151725769. Validation Accuracy: 98.90499877929688
Iterations: 1710
Iterations: 1720
Iterations: 1730
Iterations: 1740
Iterations: 1750
Iterations: 1760
Iterations: 1770
Iterations: 1780
Iterations: 1790
Iterations: 1800
Iterations: 1800 Loss: 0.0067049115896224976. Validation Accuracy: 99.08999633789062
Epoch: 6
Iterations: 1810
Iterations: 1820
Iterations: 1830
Iterations: 1840
Iterations: 1850
Iterations: 1860
Iterations: 1870
Iterations: 1880
Iterations: 1890
```

```
Iterations: 1900
Iterations: 1900 Loss: 0.006134942173957825. Validation Accuracy: 99.18499755859375
Iterations: 1910
Iterations: 1920
Iterations: 1930
Iterations: 1940
Iterations: 1950
Iterations: 1960
Iterations: 1970
Iterations: 1980
Iterations: 1990
Iterations: 2000
Iterations: 2000 Loss: 0.0297554824501276. Validation Accuracy: 98.91999816894531
Iterations: 2010
Iterations: 2020
Iterations: 2030
Iterations: 2040
Iterations: 2050
Iterations: 2060
Iterations: 2070
Iterations: 2080
Iterations: 2090
Iterations: 2100
Iterations: 2100 Loss: 0.002833949401974678. Validation Accuracy: 99.08499908447266
Epoch: 7
Iterations: 2110
Iterations: 2120
Iterations: 2130
Iterations: 2140
Iterations: 2150
Iterations: 2160
Iterations: 2170
Iterations: 2180
Iterations: 2190
Iterations: 2200
Iterations: 2200 Loss: 0.002344260923564434. Validation Accuracy: 99.15999603271484
Iterations: 2210
Iterations: 2220
Iterations: 2230
Iterations: 2240
Iterations: 2250
Iterations: 2260
Iterations: 2270
Iterations: 2280
Iterations: 2290
Iterations: 2300
Iterations: 2300 Loss: 0.002676570089533925. Validation Accuracy: 99.15499877929688
Iterations: 2310
Iterations: 2320
Iterations: 2330
Iterations: 2340
Iterations: 2350
Iterations: 2360
Iterations: 2370
Iterations: 2380
Iterations: 2390
Iterations: 2400
Iterations: 2400 Loss: 0.007678781170397997. Validation Accuracy: 98.91500091552734
Epoch: 8
Iterations: 2410
Iterations: 2420
Iterations: 2430
Iterations: 2440
Iterations: 2450
Iterations: 2460
Iterations: 2470
Iterations: 2480
Iterations: 2490
Iterations: 2500
Iterations: 2500 Loss: 0.0034435675479471684. Validation Accuracy: 99.2249984741211
Iterations: 2510
Iterations: 2520
```

```
Iterations: 2530
Iterations: 2540
Iterations: 2550
Iterations: 2560
Iterations: 2570
Iterations: 2580
Iterations: 2590
Iterations: 2600
Iterations: 2600 Loss: 0.0009345505386590958. Validation Accuracy: 99.14999389648438
Iterations: 2610
Iterations: 2620
Iterations: 2630
Iterations: 2640
Iterations: 2650
Iterations: 2660
Iterations: 2670
Iterations: 2680
Iterations: 2690
Iterations: 2700
Iterations: 2700 Loss: 0.0014243305195122957. Validation Accuracy: 99.1199951171875
Epoch: 9
Iterations: 2710
Iterations: 2720
Iterations: 2730
Iterations: 2740
Iterations: 2750
Iterations: 2760
Iterations: 2770
Iterations: 2780
Iterations: 2790
Iterations: 2800
Iterations: 2800 Loss: 0.005153914913535118. Validation Accuracy: 99.18499755859375
Iterations: 2810
Iterations: 2820
Iterations: 2830
Iterations: 2840
Iterations: 2850
Iterations: 2860
Iterations: 2870
Iterations: 2880
Iterations: 2890
Iterations: 2900
Iterations: 2900 Loss: 0.01095188595354557. Validation Accuracy: 99.18999481201172
Iterations: 2910
Iterations: 2920
Iterations: 2930
Iterations: 2940
Iterations: 2950
Iterations: 2960
Iterations: 2970
Iterations: 2980
Iterations: 2990
Iterations: 3000
Iterations: 3000 Loss: 0.01367961149662733. Validation Accuracy: 99.15499877929688
Iterations: 3000 Loss: 0.01367961149662733. Test Accuracy: 99.13999938964844
```

Output from special ResNet9

```
Epoch: 0
Iterations: 10
Iterations: 20
Iterations: 30
Iterations: 40
Iterations: 50
Iterations: 60
Iterations: 70
Iterations: 80
Iterations: 90
Iterations: 100
Iterations: 100 Loss: 0.8362917900085449. Validation Accuracy: 83.50999450683594
Iterations: 110
Iterations: 120
Iterations: 130
Iterations: 140
Iterations: 150
Iterations: 160
Iterations: 170
Iterations: 180
Iterations: 190
Iterations: 200
Iterations: 200 Loss: 0.3857309818267822. Validation Accuracy: 88.56499481201172
Iterations: 210
Iterations: 220
Iterations: 230
Iterations: 240
Iterations: 250
Iterations: 260
Iterations: 270
Iterations: 280
Iterations: 290
Iterations: 300
Iterations: 300 Loss: 0.09116969257593155. Validation Accuracy: 97.40999603271484
Epoch: 1
Iterations: 310
Iterations: 320
Iterations: 330
Iterations: 340
Iterations: 350
Iterations: 360
Iterations: 370
Iterations: 380
Iterations: 390
Iterations: 400
Iterations: 400 Loss: 0.07021695375442505. Validation Accuracy: 98.67499542236328
Iterations: 410
Iterations: 420
Iterations: 430
Iterations: 440
Iterations: 450
Iterations: 460
Iterations: 470
Iterations: 480
Iterations: 490
Iterations: 500
Iterations: 500 Loss: 0.04268629476428032. Validation Accuracy: 98.40499877929688
Iterations: 510
Iterations: 520
Iterations: 530
Iterations: 540
Iterations: 550
Iterations: 560
Iterations: 570
Iterations: 580
Iterations: 590
Iterations: 600
Iterations: 600 Loss: 0.012973245233297348. Validation Accuracy: 99.0
Epoch: 2
Iterations: 610
```

```
Iterations: 620
Iterations: 630
Iterations: 640
Iterations: 650
Iterations: 660
Iterations: 670
Iterations: 680
Iterations: 690
Iterations: 700
Iterations: 700 Loss: 0.0747428759932518. Validation Accuracy: 98.7249984741211
Iterations: 710
Iterations: 720
Iterations: 730
Iterations: 740
Iterations: 750
Iterations: 760
Iterations: 770
Iterations: 780
Iterations: 790
Iterations: 800
Iterations: 800 Loss: 0.004659766796976328. Validation Accuracy: 98.59500122070312
Iterations: 810
Iterations: 820
Iterations: 830
Iterations: 840
Iterations: 850
Iterations: 860
Iterations: 870
Iterations: 880
Iterations: 890
Iterations: 900
Iterations: 900 Loss: 0.1456929445266723 6. Validation Accuracy: 98.54000091552734
Epoch: 3
Iterations: 910
Iterations: 920
Iterations: 930
Iterations: 940
Iterations: 950
Iterations: 960
Iterations: 970
Iterations: 980
Iterations: 990
Iterations: 1000
Iterations: 1000 Loss: 0.00253286561928689 5. Validation Accuracy: 98.875
Iterations: 1010
Iterations: 1020
Iterations: 1030
Iterations: 1040
Iterations: 1050
Iterations: 1060
Iterations: 1070
Iterations: 1080
Iterations: 1090
Iterations: 1100
Iterations: 1100 Loss: 0.007060263305902481. Validation Accuracy: 98.98500061035156
Iterations: 1110
Iterations: 1120
Iterations: 1130
Iterations: 1140
Iterations: 1150
Iterations: 1160
Iterations: 1170
Iterations: 1180
Iterations: 1190
Iterations: 1200
Iterations: 1200 Loss: 0.07738589495420456. Validation Accuracy: 98.70499420166016
Epoch: 4
Iterations: 1210
Iterations: 1220
Iterations: 1230
Iterations: 1240
Iterations: 1250
```

```
Iterations: 1260
Iterations: 1270
Iterations: 1280
Iterations: 1290
Iterations: 1300
Iterations: 1300 Loss: 0.017915446311235428. Validation Accuracy: 98.94999694824219
Iterations: 1310
Iterations: 1320
Iterations: 1330
Iterations: 1340
Iterations: 1350
Iterations: 1360
Iterations: 1370
Iterations: 1380
Iterations: 1390
Iterations: 1400
Iterations: 1400 Loss: 0.0045769913122057915. Validation Accuracy: 99.02999877929688
Iterations: 1410
Iterations: 1420
Iterations: 1430
Iterations: 1440
Iterations: 1450
Iterations: 1460
Iterations: 1470
Iterations: 1480
Iterations: 1490
Iterations: 1500
Iterations: 1500 Loss: 0.014487535692751408. Validation Accuracy: 98.73500061035156
Epoch: 5
Iterations: 1510
Iterations: 1520
Iterations: 1530
Iterations: 1540
Iterations: 1550
Iterations: 1560
Iterations: 1570
Iterations: 1580
Iterations: 1590
Iterations: 1600
Iterations: 1600 Loss: 9.00411032489501e-05. Validation Accuracy: 99.19499969482422
Iterations: 1610
Iterations: 1620
Iterations: 1630
Iterations: 1640
Iterations: 1650
Iterations: 1660
Iterations: 1670
Iterations: 1680
Iterations: 1690
Iterations: 1700
Iterations: 1700 Loss: 0.0032250797376036644. Validation Accuracy: 99.08999633789062
Iterations: 1710
Iterations: 1720
Iterations: 1730
Iterations: 1740
Iterations: 1750
Iterations: 1760
Iterations: 1770
Iterations: 1780
Iterations: 1790
Iterations: 1800
Iterations: 1800 Loss: 0.007128482684493065. Validation Accuracy: 99.04499816894531
Epoch: 6
Iterations: 1810
Iterations: 1820
Iterations: 1830
Iterations: 1840
Iterations: 1850
Iterations: 1860
Iterations: 1870
Iterations: 1880
Iterations: 1890
```

```
Iterations: 1900
Iterations: 1900 Loss: 0.00044600715045817196. Validation Accuracy: 99.04499816894531
Iterations: 1910
Iterations: 1920
Iterations: 1930
Iterations: 1940
Iterations: 1950
Iterations: 1960
Iterations: 1970
Iterations: 1980
Iterations: 1990
Iterations: 2000
Iterations: 2000 Loss: 0.0005193837569095194. Validation Accuracy: 99.04000091552734
Iterations: 2010
Iterations: 2020
Iterations: 2030
Iterations: 2040
Iterations: 2050
Iterations: 2060
Iterations: 2070
Iterations: 2080
Iterations: 2090
Iterations: 2100
Iterations: 2100 Loss: 0.00215676565858984. Validation Accuracy: 99.17499542236328
Epoch: 7
Iterations: 2110
Iterations: 2120
Iterations: 2130
Iterations: 2140
Iterations: 2150
Iterations: 2160
Iterations: 2170
Iterations: 2180
Iterations: 2190
Iterations: 2200
Iterations: 2200 Loss: 0.0005823741666972637. Validation Accuracy: 99.18000030517578
Iterations: 2210
Iterations: 2220
Iterations: 2230
Iterations: 2240
Iterations: 2250
Iterations: 2260
Iterations: 2270
Iterations: 2280
Iterations: 2290
Iterations: 2300
Iterations: 2300 Loss: 0.002317153830463886. Validation Accuracy: 99.18999481201172
Iterations: 2310
Iterations: 2320
Iterations: 2330
Iterations: 2340
Iterations: 2350
Iterations: 2360
Iterations: 2370
Iterations: 2380
Iterations: 2390
Iterations: 2400
Iterations: 2400 Loss: 0.0015232330188155174. Validation Accuracy: 99.2449951171875
Epoch: 8
Iterations: 2410
Iterations: 2420
Iterations: 2430
Iterations: 2440
Iterations: 2450
Iterations: 2460
Iterations: 2470
Iterations: 2480
Iterations: 2490
Iterations: 2500
Iterations: 2500 Loss: 0.001575684873387 2175. Validation Accuracy: 99.28499603271484
Iterations: 2510
Iterations: 2520
```

```
Iterations: 2530
Iterations: 2540
Iterations: 2550
Iterations: 2560
Iterations: 2570
Iterations: 2580
Iterations: 2590
Iterations: 2600
Iterations: 2600 Loss: 0.003460629377514124. Validation Accuracy: 99.27499389648438
Iterations: 2610
Iterations: 2620
Iterations: 2630
Iterations: 2640
Iterations: 2650
Iterations: 2660
Iterations: 2670
Iterations: 2680
Iterations: 2690
Iterations: 2700
Iterations: 2700 Loss: 8.577576954849064e-05. Validation Accuracy: 99.26499938964844
Epoch: 9
Iterations: 2710
Iterations: 2720
Iterations: 2730
Iterations: 2740
Iterations: 2750
Iterations: 2760
Iterations: 2770
Iterations: 2780
Iterations: 2790
Iterations: 2800
Iterations: 2800 Loss: 9.463933383813128e-05. Validation Accuracy: 99.31499481201172
Iterations: 2810
Iterations: 2820
Iterations: 2830
Iterations: 2840
Iterations: 2850
Iterations: 2860
Iterations: 2870
Iterations: 2880
Iterations: 2890
Iterations: 2900
Iterations: 2900 Loss: 0.00012100701133022085. Validation Accuracy: 99.2699966430664
Iterations: 2910
Iterations: 2920
Iterations: 2930
Iterations: 2940
Iterations: 2950
Iterations: 2960
Iterations: 2970
Iterations: 2980
Iterations: 2990
Iterations: 3000
Iterations: 3000 Loss: 5.667606728820829e-06. Validation Accuracy: 99.32499694824219
Iterations: 3000 Loss: 5.667606728820829e-06. Test Accuracy: 99.31999969482422
```