

## 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

```
1 def conv_block(in_channels, out_channels, pool=False, res=False, pool_size=2):
2     layers = []
3     if res:
4         layers.append(nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding
5                                 =1))
6     else:
7         layers.append(nn.Conv2d(in_channels, out_channels, kernel_size=3))
8         layers.append(nn.BatchNorm2d(out_channels))
9         layers.append(nn.ReLU())
10    if pool:
11        layers.append(nn.MaxPool2d(pool_size))
12    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.

```
1 class ResNet9(nn.Module):
2     def __init__(self, in_channels, num_classes):
3         super().__init__()
4         # Preparation Layer
5         self.prep = conv_block(in_channels, 32)
6         # Layer 1 w/ Res Blocks
7         self.layer1_conv = conv_block(32, 64, pool=True)
8         self.layer1_res = nn.Sequential(conv_block(64, 64, res=True),
9                                         conv_block(64, 64, res=True))
10
11        # Layer 2
12        self.layer2 = conv_block(64, 128)
13        # Layer 1 w/ Res Blocks
14        self.layer3_conv = conv_block(128, 256, pool=True)
15        self.layer3_res = nn.Sequential(conv_block(256, 256, res=True),
16                                        conv_block(256, 256, res=True))
17
18        # Classifier
19        self.classifier = nn.Sequential(nn.MaxPool2d(2),
20                                        nn.Flatten(),
21                                        nn.Linear(1024,10)),
22                                        #nn.Linear(1024,256),
23                                        #nn.ReLU(),
24                                        #nn.Linear(256,64),
25                                        #nn.ReLU(),
26                                        #nn.Linear(64,10),
27                                        #nn.ReLU())
28
29        self.shape_tester = nn.Sequential(nn.MaxPool2d(2),
30                                        nn.Flatten())
31
32        self.fc = nn.Linear(1024,10)
33
34        # Params
35        self.num_classes = num_classes
36
37    def class3(self, size1d, num_classes):
38        return nn.Linear(size1d, num_classes)
39
40    def forward(self, x):
41
42        # Preparation Layer
43        y = self.prep(x)
44
45        # Layer 1 w/ Res Blocks
46        y1 = self.layer1_conv(y)
47        #print(f"Residual in layer 1 has size: {y1.shape}") # has 12,12
48        #y = self.layer1_res(y1) + y1
49        y2 = self.layer1_res(y1)
50        #print(f"After Residual in layer 1 has size: {y2.shape}") # has 8,8
51        y = y1 + y2
52
53        # Layer 2
54        y = self.layer2(y)
55
56        # Layer 3 w/ Res Blocks
57        y1 = self.layer3_conv(y)
58        y = self.layer3_res(y1) + y1
```

```
55
56     # Classifier
57     y = self.shape_tester(y)
58     #print(f"After Flatten in classifier layer has size: {y.shape}")
59     y = self.fc(y)
60
61     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.

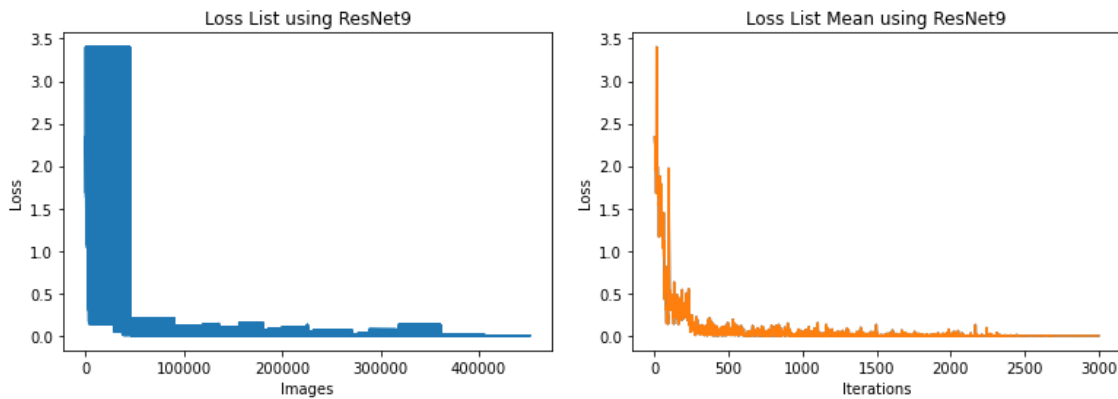


Figure 1: Loss List and Loss List Mean using ResNet9

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.

Some Predictions of Trained Model on test

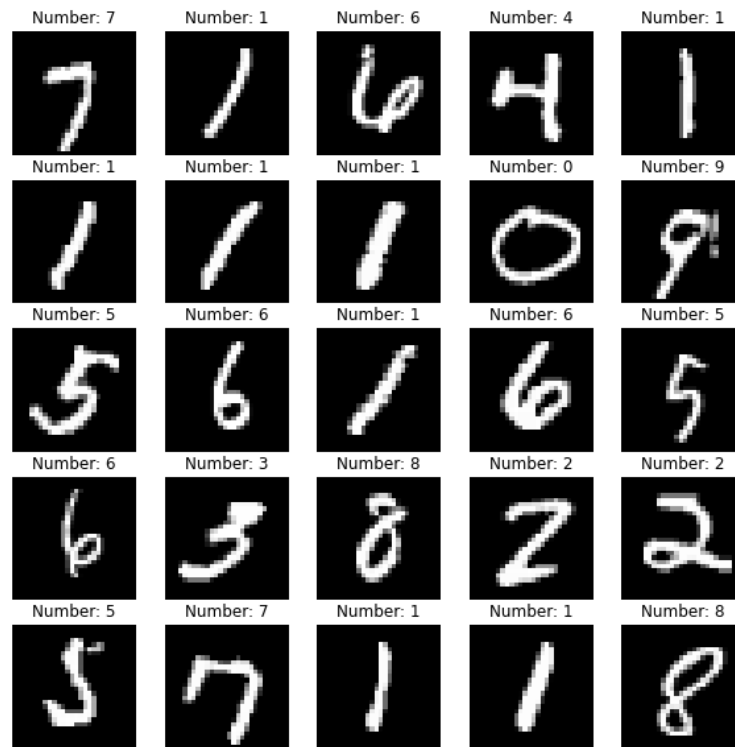


Figure 2: Example Photos Model testing

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

```
1 '''
2 Justine Serdoncillo
3 EE 5561 - Image Processing
4 Problem Set 4
5 December 6, 2023
6 '''
7
8 # %% Problem Statement
9 """
10 In this exercise, you will implement a ResNet structure and will
11 use it for MNIST database classification. You can use the given PyTorch tutorial for
12 training
13 and replace the model with ResNet by using the following ResNet9.
14 """
15
16 # %% import libraries
17 import torch
18 from torchvision import datasets
19 from torchvision.transforms import ToTensor
20 import matplotlib.pyplot as plt
21 import torch.nn as nn
22 import numpy as np
23
24 # %% LOAD THE MNIST DATASET
25 data_full = datasets.MNIST(root = 'data',
26                             train = True,
27                             transform = ToTensor(),
28                             download = True)
29
30 # visualize the data
31 plt.imshow(data_full.data[0])
32 plt.imshow(data_full.data[0], cmap='gray')
33 plt.imshow(data_full.data[61], cmap='gray')
34
35 # access the data
36 data1 = data_full.data[0]
37 plt.imshow(data1, cmap='gray')
38
39 # visualize multiple images
40 figure = plt.figure(figsize=(10,10))
41 cols, rows = 5,5
42
43 for i in range(1, cols*rows+1):
44
45     idx = torch.randint(len(data_full), size=(1,)).item()
46
47     img, label = data_full[idx]
48
49     figure.add_subplot(rows,cols,i)
50     plt.title('Number: ' + str(label))
51     plt.axis('off')
52     plt.imshow(img.squeeze(), cmap='gray')
53
54 plt.show()
55
56
57 # %% SPLIT THE DATASET
58 # train, test, validation seperation
59 train_data, test_data, valid_data = torch.utils.data.random_split(data_full, [30000, 10000,
60     20000])
61
62 batch_size = 100
63
64 loaders = {'train': torch.utils.data.DataLoader(train_data,
65     batch_size = batch_size,
66     shuffle=True),
```

```

66         'test': torch.utils.data.DataLoader(test_data,
67                                             batch_size = batch_size,
68                                             shuffle=True),
69         'valid': torch.utils.data.DataLoader(valid_data,
70                                             batch_size = batch_size,
71                                             shuffle=True)}
72 """
73 #visualize the dictionary
74 train_part = loaders.get('train')
75 data2 = train_part.dataset
76 element1 = data2[0][0].squeeze()
77 plt.imshow(element1, cmap='gray')
78 """
79
80
81 # %%
82 #####
83 ## EE 5561 Assignment 4 ##
84 ## JJ Personal Code ##
85 #####
86 def conv_block(in_channels, out_channels, pool=False, res=False, pool_size=2):
87     layers = []
88     if res:
89         layers.append(nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding
90                                =1))
91     else:
92         layers.append(nn.Conv2d(in_channels, out_channels, kernel_size=3))
93     layers.append(nn.BatchNorm2d(out_channels))
94     layers.append(nn.ReLU())
95     if pool:
96         layers.append(nn.MaxPool2d(pool_size))
97     return nn.Sequential(*layers)
98
99 class ResNet9(nn.Module):
100     def __init__(self, in_channels, num_classes):
101         super().__init__()
102         # Preparation Layer
103         self.prep = conv_block(in_channels, 32)
104         # Layer 1 w/ Res Blocks
105         self.layer1_conv = conv_block(32, 64, pool=True)
106         self.layer1_res = nn.Sequential(conv_block(64, 64, res=True),
107                                         conv_block(64, 64, res=True))
108         # Layer 2
109         self.layer2 = conv_block(64, 128)
110         # Layer 1 w/ Res Blocks
111         self.layer3_conv = conv_block(128, 256, pool=True)
112         self.layer3_res = nn.Sequential(conv_block(256, 256, res=True),
113                                         conv_block(256, 256, res=True))
114         # Classifier
115         self.classifier = nn.Sequential(nn.MaxPool2d(2),
116                                         nn.Flatten(),
117                                         nn.Linear(1024,10))
118         #nn.Linear(1024,256),
119         #nn.ReLU(),
120         #nn.Linear(256,64),
121         #nn.ReLU(),
122         #nn.Linear(64,10),
123         #nn.ReLU()
124         self.shape_tester = nn.Sequential(nn.MaxPool2d(2),
125                                         nn.Flatten())
126         self.fc = nn.Linear(1024,10)
127
128         # Params
129         self.num_classes = num_classes
130
131     def class3(self, size1d, num_classes):
132         return nn.Linear(size1d, num_classes)
133
134     def forward(self, x):
135
136         # Preparation Layer

```

```

137     y = self.prep(x)
138
139     # Layer 1 w/ Res Blocks
140     y1 = self.layer1_conv(y)
141     #print(f"Residual in layer 1 has size: {y1.shape}") # has 12,12
142     #y = self.layer1_res(y1) + y1
143     y2 = self.layer1_res(y1)
144     #print(f"After Residual in layer 1 has size: {y2.shape}") # has 8,8
145     y = y1 + y2
146
147     # Layer 2
148     y = self.layer2(y)
149
150     # Layer 3 w/ Res Blocks
151     y1 = self.layer3_conv(y)
152     y = self.layer3_res(y1) + y1
153
154     # Classifier
155     y = self.shape_tester(y)
156     #print(f"After Flatten in classifier layer has size: {y.shape}")
157     y = self.fc(y)
158
159     return y
160
161 # %% SET UP THE MODEL
162
163 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
164 model = ResNet9(1, 10)
165 model.to(device)
166
167 # define the loss function
168 criterion = nn.CrossEntropyLoss()
169
170 # define the optimizer
171 learning_rate = 0.1
172 optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
173
174 # define epoch number
175 num_epochs = 10
176
177 # initialize the loss
178 loss_list = []
179 loss_list_mean = []
180
181
182 # %% TRAINING STARTS HERE
183
184 iter = 0
185 for epoch in range(num_epochs):
186
187     print('Epoch: {}'.format(epoch))
188
189     loss_buff = []
190
191     for i, (images, labels) in enumerate(loaders['train']):
192
193         # getting the images and labels from the training dataset
194         images = images.requires_grad_().to(device)
195         labels = labels.to(device)
196
197         # clear the gradients
198         optimizer.zero_grad()
199
200         # call the NN
201         outputs = model(images) # torch.Size([100, 1, 28, 28])
202
203         # loss calculation
204         loss = criterion(outputs, labels)
205         loss_buff = np.append(loss_buff, loss.item())
206
207         # back propagation
208         loss.backward()

```



```

209     loss_list = np.append(loss_list, (loss_buff))
210
211     #update parameters
212     optimizer.step()
213
214     iter += 1
215
216     if iter % 10 == 0:
217         print('Iterations: {}'.format(iter))
218
219     ##### VALIDATION PART#####
220
221     if iter % 100 == 0:
222
223         # accuracy
224         correct = 0
225         total = 0
226
227         for i, (images, labels) in enumerate(loaders['valid']):
228
229             # getting the images and labels from the training dataset
230             images = images.to(device)
231             labels = labels.to(device)
232
233             # clear the gradients
234             optimizer.zero_grad()
235
236             # call the NN
237             outputs = model(images)
238
239             # get the predictions
240             _, predicted = torch.max(outputs.data, 1)
241
242             total += labels.size(0)
243
244             correct += (predicted == labels).sum()
245
246         accuracy = 100 * correct / total
247
248         print('Iterations: {} Loss: {}. Validation Accuracy: {}'.
249               format(iter, loss.item(), accuracy))
250
251         loss_list_mean = np.append(loss_list_mean, (loss.item()))
252         #####
253
254     #visualize the loss
255     plt.plot(loss_list)
256     plt.plot(loss_list_mean)
257
258     fig, ax = plt.subplots()
259     ax.plot(loss_list)
260     ax.set_title("Loss List using ResNet9")
261     ax.set_xlabel("Images")
262     ax.set_ylabel("Loss")
263
264     fig, ax = plt.subplots()
265     ax.plot(loss_list_mean)
266     ax.set_title("Loss List Mean using ResNet9")
267     ax.set_xlabel("Iterations")
268     ax.set_ylabel("Loss")
269     ax.plot(loss_list_mean)
270
271     # %% TEST PART#####
272     correct = 0
273     total = 0
274
275     for i, (images, labels) in enumerate(loaders['test']):
276
277         # getting the images and labels from the training dataset
278         images = images.to(device)
279         labels = labels.to(device)

```

```

281
282     # clear the gradients
283     optimizer.zero_grad()
284
285     # call the NN
286     outputs = model(images)
287
288     # get the predictions
289     _, predicted = torch.max(outputs.data, 1)
290
291     total += labels.size(0)
292
293     correct += (predicted == labels).sum()
294
295 accuracy = 100 * correct / total
296
297 print('Iterations: {} Loss: {}. Test Accuracy: {}'.format(iter, loss.item(), accuracy))
298
299 # %%
300
301 # Save:
302 torch.save(model.state_dict(), "weights_1Linear.h5")
303
304
305 # Load:
306 device = torch.device('cpu')
307 model = ResNet9(1, 10)
308 model.to(device)
309 model.load_state_dict(torch.load("weights_1Linear.h5"))
310 model.eval()
311
312 # visualize multiple images
313 fig, ax = plt.subplots(5,5,figsize=(10,10))
314 fig.suptitle("Some Predictions of Trained Model on test ")
315 cols, rows = 5,5
316
317 for i in range(5):
318     for j in range(5):
319         for test_images, test_labels in loaders['test']:
320             img = test_images[0]
321             img.unsqueeze_(1)
322             img = img.to(device)
323             optimizer.zero_grad()
324             #print(img.shape)
325             outputs = model(img)
326             _, predicted = torch.max(outputs.data, 1)
327
328             ax[i,j].set_title('Number: ' + str(predicted.numpy()[0]))
329             ax[i,j].axis('off')
330             ax[i,j].imshow(img.squeeze().numpy(), cmap='gray')

```

## Output from normal ResNet9

```
1 Epoch: 0
2 Iterations: 10
3 Iterations: 20
4 Iterations: 30
5 Iterations: 40
6 Iterations: 50
7 Iterations: 60
8 Iterations: 70
9 Iterations: 80
10 Iterations: 90
11 Iterations: 100
12 Iterations: 100 Loss: 0.8954293131828308. Validation Accuracy: 73.26499938964844
13 Iterations: 110
14 Iterations: 120
15 Iterations: 130
16 Iterations: 140
17 Iterations: 150
18 Iterations: 160
19 Iterations: 170
20 Iterations: 180
21 Iterations: 190
22 Iterations: 200
23 Iterations: 200 Loss: 0.14886337518692017. Validation Accuracy: 95.35499572753906
24 Iterations: 210
25 Iterations: 220
26 Iterations: 230
27 Iterations: 240
28 Iterations: 250
29 Iterations: 260
30 Iterations: 270
31 Iterations: 280
32 Iterations: 290
33 Iterations: 300
34 Iterations: 300 Loss: 0.1941969096660614. Validation Accuracy: 96.18999481201172
35 Epoch: 1
36 Iterations: 310
37 Iterations: 320
38 Iterations: 330
39 Iterations: 340
40 Iterations: 350
41 Iterations: 360
42 Iterations: 370
43 Iterations: 380
44 Iterations: 390
45 Iterations: 400
46 Iterations: 400 Loss: 0.1246100440621376. Validation Accuracy: 97.66999816894531
47 Iterations: 410
48 Iterations: 420
49 Iterations: 430
50 Iterations: 440
51 Iterations: 450
52 Iterations: 460
53 Iterations: 470
54 Iterations: 480
55 Iterations: 490
56 Iterations: 500
57 Iterations: 500 Loss: 0.08373874425888062. Validation Accuracy: 97.48500061035156
58 Iterations: 510
59 Iterations: 520
60 Iterations: 530
61 Iterations: 540
62 Iterations: 550
63 Iterations: 560
64 Iterations: 570
65 Iterations: 580
66 Iterations: 590
67 Iterations: 600
68 Iterations: 600 Loss: 0.05688638240098953. Validation Accuracy: 98.30999755859375
69 Epoch: 2
70 Iterations: 610
```

```
71 Iterations: 620
72 Iterations: 630
73 Iterations: 640
74 Iterations: 650
75 Iterations: 660
76 Iterations: 670
77 Iterations: 680
78 Iterations: 690
79 Iterations: 700
80 Iterations: 700 Loss: 0.034620415419340134. Validation Accuracy: 98.39999389648438
81 Iterations: 710
82 Iterations: 720
83 Iterations: 730
84 Iterations: 740
85 Iterations: 750
86 Iterations: 760
87 Iterations: 770
88 Iterations: 780
89 Iterations: 790
90 Iterations: 800
91 Iterations: 800 Loss: 0.028844986110925674. Validation Accuracy: 98.48500061035156
92 Iterations: 810
93 Iterations: 820
94 Iterations: 830
95 Iterations: 840
96 Iterations: 850
97 Iterations: 860
98 Iterations: 870
99 Iterations: 880
100 Iterations: 890
101 Iterations: 900
102 Iterations: 900 Loss: 0.02131034806370735. Validation Accuracy: 98.36499786376953
103 Epoch: 3
104 Iterations: 910
105 Iterations: 920
106 Iterations: 930
107 Iterations: 940
108 Iterations: 950
109 Iterations: 960
110 Iterations: 970
111 Iterations: 980
112 Iterations: 990
113 Iterations: 1000
114 Iterations: 1000 Loss: 0.07528997957706451. Validation Accuracy: 98.56999969482422
115 Iterations: 1010
116 Iterations: 1020
117 Iterations: 1030
118 Iterations: 1040
119 Iterations: 1050
120 Iterations: 1060
121 Iterations: 1070
122 Iterations: 1080
123 Iterations: 1090
124 Iterations: 1100
125 Iterations: 1100 Loss: 0.013861780986189842. Validation Accuracy: 98.70499420166016
126 Iterations: 1110
127 Iterations: 1120
128 Iterations: 1130
129 Iterations: 1140
130 Iterations: 1150
131 Iterations: 1160
132 Iterations: 1170
133 Iterations: 1180
134 Iterations: 1190
135 Iterations: 1200
136 Iterations: 1200 Loss: 0.06764683872461319. Validation Accuracy: 98.71499633789062
137 Epoch: 4
138 Iterations: 1210
139 Iterations: 1220
140 Iterations: 1230
141 Iterations: 1240
142 Iterations: 1250
```

```

143 Iterations: 1260
144 Iterations: 1270
145 Iterations: 1280
146 Iterations: 1290
147 Iterations: 1300
148 Iterations: 1300 Loss: 0.015819817781448364. Validation Accuracy: 98.59500122070312
149 Iterations: 1310
150 Iterations: 1320
151 Iterations: 1330
152 Iterations: 1340
153 Iterations: 1350
154 Iterations: 1360
155 Iterations: 1370
156 Iterations: 1380
157 Iterations: 1390
158 Iterations: 1400
159 Iterations: 1400 Loss: 0.008826137520372868. Validation Accuracy: 98.41999816894531
160 Iterations: 1410
161 Iterations: 1420
162 Iterations: 1430
163 Iterations: 1440
164 Iterations: 1450
165 Iterations: 1460
166 Iterations: 1470
167 Iterations: 1480
168 Iterations: 1490
169 Iterations: 1500
170 Iterations: 1500 Loss: 0.004896394908428192. Validation Accuracy: 99.02999877929688
171 Epoch: 5
172 Iterations: 1510
173 Iterations: 1520
174 Iterations: 1530
175 Iterations: 1540
176 Iterations: 1550
177 Iterations: 1560
178 Iterations: 1570
179 Iterations: 1580
180 Iterations: 1590
181 Iterations: 1600
182 Iterations: 1600 Loss: 0.016535509377717972. Validation Accuracy: 99.01499938964844
183 Iterations: 1610
184 Iterations: 1620
185 Iterations: 1630
186 Iterations: 1640
187 Iterations: 1650
188 Iterations: 1660
189 Iterations: 1670
190 Iterations: 1680
191 Iterations: 1690
192 Iterations: 1700
193 Iterations: 1700 Loss: 0.05973479151725769. Validation Accuracy: 98.90499877929688
194 Iterations: 1710
195 Iterations: 1720
196 Iterations: 1730
197 Iterations: 1740
198 Iterations: 1750
199 Iterations: 1760
200 Iterations: 1770
201 Iterations: 1780
202 Iterations: 1790
203 Iterations: 1800
204 Iterations: 1800 Loss: 0.0067049115896224976. Validation Accuracy: 99.08999633789062
205 Epoch: 6
206 Iterations: 1810
207 Iterations: 1820
208 Iterations: 1830
209 Iterations: 1840
210 Iterations: 1850
211 Iterations: 1860
212 Iterations: 1870
213 Iterations: 1880
214 Iterations: 1890

```

```

215 Iterations: 1900
216 Iterations: 1900 Loss: 0.006134942173957825. Validation Accuracy: 99.18499755859375
217 Iterations: 1910
218 Iterations: 1920
219 Iterations: 1930
220 Iterations: 1940
221 Iterations: 1950
222 Iterations: 1960
223 Iterations: 1970
224 Iterations: 1980
225 Iterations: 1990
226 Iterations: 2000
227 Iterations: 2000 Loss: 0.0297554824501276. Validation Accuracy: 98.91999816894531
228 Iterations: 2010
229 Iterations: 2020
230 Iterations: 2030
231 Iterations: 2040
232 Iterations: 2050
233 Iterations: 2060
234 Iterations: 2070
235 Iterations: 2080
236 Iterations: 2090
237 Iterations: 2100
238 Iterations: 2100 Loss: 0.002833949401974678. Validation Accuracy: 99.08499908447266
239 Epoch: 7
240 Iterations: 2110
241 Iterations: 2120
242 Iterations: 2130
243 Iterations: 2140
244 Iterations: 2150
245 Iterations: 2160
246 Iterations: 2170
247 Iterations: 2180
248 Iterations: 2190
249 Iterations: 2200
250 Iterations: 2200 Loss: 0.002344260923564434. Validation Accuracy: 99.15999603271484
251 Iterations: 2210
252 Iterations: 2220
253 Iterations: 2230
254 Iterations: 2240
255 Iterations: 2250
256 Iterations: 2260
257 Iterations: 2270
258 Iterations: 2280
259 Iterations: 2290
260 Iterations: 2300
261 Iterations: 2300 Loss: 0.002676570089533925. Validation Accuracy: 99.15499877929688
262 Iterations: 2310
263 Iterations: 2320
264 Iterations: 2330
265 Iterations: 2340
266 Iterations: 2350
267 Iterations: 2360
268 Iterations: 2370
269 Iterations: 2380
270 Iterations: 2390
271 Iterations: 2400
272 Iterations: 2400 Loss: 0.007678781170397997. Validation Accuracy: 98.91500091552734
273 Epoch: 8
274 Iterations: 2410
275 Iterations: 2420
276 Iterations: 2430
277 Iterations: 2440
278 Iterations: 2450
279 Iterations: 2460
280 Iterations: 2470
281 Iterations: 2480
282 Iterations: 2490
283 Iterations: 2500
284 Iterations: 2500 Loss: 0.0034435675479471684. Validation Accuracy: 99.2249984741211
285 Iterations: 2510
286 Iterations: 2520

```

```
287 Iterations: 2530
288 Iterations: 2540
289 Iterations: 2550
290 Iterations: 2560
291 Iterations: 2570
292 Iterations: 2580
293 Iterations: 2590
294 Iterations: 2600
295 Iterations: 2600 Loss: 0.0009345505386590958. Validation Accuracy: 99.14999389648438
296 Iterations: 2610
297 Iterations: 2620
298 Iterations: 2630
299 Iterations: 2640
300 Iterations: 2650
301 Iterations: 2660
302 Iterations: 2670
303 Iterations: 2680
304 Iterations: 2690
305 Iterations: 2700
306 Iterations: 2700 Loss: 0.0014243305195122957. Validation Accuracy: 99.1199951171875
307 Epoch: 9
308 Iterations: 2710
309 Iterations: 2720
310 Iterations: 2730
311 Iterations: 2740
312 Iterations: 2750
313 Iterations: 2760
314 Iterations: 2770
315 Iterations: 2780
316 Iterations: 2790
317 Iterations: 2800
318 Iterations: 2800 Loss: 0.005153914913535118. Validation Accuracy: 99.18499755859375
319 Iterations: 2810
320 Iterations: 2820
321 Iterations: 2830
322 Iterations: 2840
323 Iterations: 2850
324 Iterations: 2860
325 Iterations: 2870
326 Iterations: 2880
327 Iterations: 2890
328 Iterations: 2900
329 Iterations: 2900 Loss: 0.01095188595354557. Validation Accuracy: 99.18999481201172
330 Iterations: 2910
331 Iterations: 2920
332 Iterations: 2930
333 Iterations: 2940
334 Iterations: 2950
335 Iterations: 2960
336 Iterations: 2970
337 Iterations: 2980
338 Iterations: 2990
339 Iterations: 3000
340 Iterations: 3000 Loss: 0.01367961149662733. Validation Accuracy: 99.15499877929688
341 Iterations: 3000 Loss: 0.01367961149662733. Test Accuracy: 99.13999938964844
```

## Output from special ResNet9

```
1 Epoch: 0
2 Iterations: 10
3 Iterations: 20
4 Iterations: 30
5 Iterations: 40
6 Iterations: 50
7 Iterations: 60
8 Iterations: 70
9 Iterations: 80
10 Iterations: 90
11 Iterations: 100
12 Iterations: 100 Loss: 0.8362917900085449. Validation Accuracy: 83.50999450683594
13 Iterations: 110
14 Iterations: 120
15 Iterations: 130
16 Iterations: 140
17 Iterations: 150
18 Iterations: 160
19 Iterations: 170
20 Iterations: 180
21 Iterations: 190
22 Iterations: 200
23 Iterations: 200 Loss: 0.3857309818267822. Validation Accuracy: 88.56499481201172
24 Iterations: 210
25 Iterations: 220
26 Iterations: 230
27 Iterations: 240
28 Iterations: 250
29 Iterations: 260
30 Iterations: 270
31 Iterations: 280
32 Iterations: 290
33 Iterations: 300
34 Iterations: 300 Loss: 0.09116969257593155. Validation Accuracy: 97.40999603271484
35 Epoch: 1
36 Iterations: 310
37 Iterations: 320
38 Iterations: 330
39 Iterations: 340
40 Iterations: 350
41 Iterations: 360
42 Iterations: 370
43 Iterations: 380
44 Iterations: 390
45 Iterations: 400
46 Iterations: 400 Loss: 0.07021695375442505. Validation Accuracy: 98.67499542236328
47 Iterations: 410
48 Iterations: 420
49 Iterations: 430
50 Iterations: 440
51 Iterations: 450
52 Iterations: 460
53 Iterations: 470
54 Iterations: 480
55 Iterations: 490
56 Iterations: 500
57 Iterations: 500 Loss: 0.04268629476428032. Validation Accuracy: 98.40499877929688
58 Iterations: 510
59 Iterations: 520
60 Iterations: 530
61 Iterations: 540
62 Iterations: 550
63 Iterations: 560
64 Iterations: 570
65 Iterations: 580
66 Iterations: 590
67 Iterations: 600
68 Iterations: 600 Loss: 0.012973245233297348. Validation Accuracy: 99.0
69 Epoch: 2
70 Iterations: 610
```



```
71 Iterations: 620
72 Iterations: 630
73 Iterations: 640
74 Iterations: 650
75 Iterations: 660
76 Iterations: 670
77 Iterations: 680
78 Iterations: 690
79 Iterations: 700
80 Iterations: 700 Loss: 0.0747428759932518. Validation Accuracy: 98.7249984741211
81 Iterations: 710
82 Iterations: 720
83 Iterations: 730
84 Iterations: 740
85 Iterations: 750
86 Iterations: 760
87 Iterations: 770
88 Iterations: 780
89 Iterations: 790
90 Iterations: 800
91 Iterations: 800 Loss: 0.004659766796976328. Validation Accuracy: 98.59500122070312
92 Iterations: 810
93 Iterations: 820
94 Iterations: 830
95 Iterations: 840
96 Iterations: 850
97 Iterations: 860
98 Iterations: 870
99 Iterations: 880
100 Iterations: 890
101 Iterations: 900
102 Iterations: 900 Loss: 0.14569294452667236. Validation Accuracy: 98.54000091552734
103 Epoch: 3
104 Iterations: 910
105 Iterations: 920
106 Iterations: 930
107 Iterations: 940
108 Iterations: 950
109 Iterations: 960
110 Iterations: 970
111 Iterations: 980
112 Iterations: 990
113 Iterations: 1000
114 Iterations: 1000 Loss: 0.002532865619286895. Validation Accuracy: 98.875
115 Iterations: 1010
116 Iterations: 1020
117 Iterations: 1030
118 Iterations: 1040
119 Iterations: 1050
120 Iterations: 1060
121 Iterations: 1070
122 Iterations: 1080
123 Iterations: 1090
124 Iterations: 1100
125 Iterations: 1100 Loss: 0.007060263305902481. Validation Accuracy: 98.98500061035156
126 Iterations: 1110
127 Iterations: 1120
128 Iterations: 1130
129 Iterations: 1140
130 Iterations: 1150
131 Iterations: 1160
132 Iterations: 1170
133 Iterations: 1180
134 Iterations: 1190
135 Iterations: 1200
136 Iterations: 1200 Loss: 0.07738589495420456. Validation Accuracy: 98.70499420166016
137 Epoch: 4
138 Iterations: 1210
139 Iterations: 1220
140 Iterations: 1230
141 Iterations: 1240
142 Iterations: 1250
```

```
143 Iterations: 1260
144 Iterations: 1270
145 Iterations: 1280
146 Iterations: 1290
147 Iterations: 1300
148 Iterations: 1300 Loss: 0.017915446311235428. Validation Accuracy: 98.94999694824219
149 Iterations: 1310
150 Iterations: 1320
151 Iterations: 1330
152 Iterations: 1340
153 Iterations: 1350
154 Iterations: 1360
155 Iterations: 1370
156 Iterations: 1380
157 Iterations: 1390
158 Iterations: 1400
159 Iterations: 1400 Loss: 0.0045769913122057915. Validation Accuracy: 99.02999877929688
160 Iterations: 1410
161 Iterations: 1420
162 Iterations: 1430
163 Iterations: 1440
164 Iterations: 1450
165 Iterations: 1460
166 Iterations: 1470
167 Iterations: 1480
168 Iterations: 1490
169 Iterations: 1500
170 Iterations: 1500 Loss: 0.014487535692751408. Validation Accuracy: 98.73500061035156
171 Epoch: 5
172 Iterations: 1510
173 Iterations: 1520
174 Iterations: 1530
175 Iterations: 1540
176 Iterations: 1550
177 Iterations: 1560
178 Iterations: 1570
179 Iterations: 1580
180 Iterations: 1590
181 Iterations: 1600
182 Iterations: 1600 Loss: 9.00411032489501e-05. Validation Accuracy: 99.19499969482422
183 Iterations: 1610
184 Iterations: 1620
185 Iterations: 1630
186 Iterations: 1640
187 Iterations: 1650
188 Iterations: 1660
189 Iterations: 1670
190 Iterations: 1680
191 Iterations: 1690
192 Iterations: 1700
193 Iterations: 1700 Loss: 0.0032250797376036644. Validation Accuracy: 99.08999633789062
194 Iterations: 1710
195 Iterations: 1720
196 Iterations: 1730
197 Iterations: 1740
198 Iterations: 1750
199 Iterations: 1760
200 Iterations: 1770
201 Iterations: 1780
202 Iterations: 1790
203 Iterations: 1800
204 Iterations: 1800 Loss: 0.007128482684493065. Validation Accuracy: 99.04499816894531
205 Epoch: 6
206 Iterations: 1810
207 Iterations: 1820
208 Iterations: 1830
209 Iterations: 1840
210 Iterations: 1850
211 Iterations: 1860
212 Iterations: 1870
213 Iterations: 1880
214 Iterations: 1890
```

```

215 Iterations: 1900
216 Iterations: 1900 Loss: 0.00044600715045817196. Validation Accuracy: 99.04499816894531
217 Iterations: 1910
218 Iterations: 1920
219 Iterations: 1930
220 Iterations: 1940
221 Iterations: 1950
222 Iterations: 1960
223 Iterations: 1970
224 Iterations: 1980
225 Iterations: 1990
226 Iterations: 2000
227 Iterations: 2000 Loss: 0.0005193837569095194. Validation Accuracy: 99.04000091552734
228 Iterations: 2010
229 Iterations: 2020
230 Iterations: 2030
231 Iterations: 2040
232 Iterations: 2050
233 Iterations: 2060
234 Iterations: 2070
235 Iterations: 2080
236 Iterations: 2090
237 Iterations: 2100
238 Iterations: 2100 Loss: 0.002156765665858984. Validation Accuracy: 99.17499542236328
239 Epoch: 7
240 Iterations: 2110
241 Iterations: 2120
242 Iterations: 2130
243 Iterations: 2140
244 Iterations: 2150
245 Iterations: 2160
246 Iterations: 2170
247 Iterations: 2180
248 Iterations: 2190
249 Iterations: 2200
250 Iterations: 2200 Loss: 0.0005823741666972637. Validation Accuracy: 99.18000030517578
251 Iterations: 2210
252 Iterations: 2220
253 Iterations: 2230
254 Iterations: 2240
255 Iterations: 2250
256 Iterations: 2260
257 Iterations: 2270
258 Iterations: 2280
259 Iterations: 2290
260 Iterations: 2300
261 Iterations: 2300 Loss: 0.0023171533830463886. Validation Accuracy: 99.18999481201172
262 Iterations: 2310
263 Iterations: 2320
264 Iterations: 2330
265 Iterations: 2340
266 Iterations: 2350
267 Iterations: 2360
268 Iterations: 2370
269 Iterations: 2380
270 Iterations: 2390
271 Iterations: 2400
272 Iterations: 2400 Loss: 0.0015232330188155174. Validation Accuracy: 99.2449951171875
273 Epoch: 8
274 Iterations: 2410
275 Iterations: 2420
276 Iterations: 2430
277 Iterations: 2440
278 Iterations: 2450
279 Iterations: 2460
280 Iterations: 2470
281 Iterations: 2480
282 Iterations: 2490
283 Iterations: 2500
284 Iterations: 2500 Loss: 0.0015756848733872175. Validation Accuracy: 99.28499603271484
285 Iterations: 2510
286 Iterations: 2520

```

```

287 Iterations: 2530
288 Iterations: 2540
289 Iterations: 2550
290 Iterations: 2560
291 Iterations: 2570
292 Iterations: 2580
293 Iterations: 2590
294 Iterations: 2600
295 Iterations: 2600 Loss: 0.003460629377514124. Validation Accuracy: 99.27499389648438
296 Iterations: 2610
297 Iterations: 2620
298 Iterations: 2630
299 Iterations: 2640
300 Iterations: 2650
301 Iterations: 2660
302 Iterations: 2670
303 Iterations: 2680
304 Iterations: 2690
305 Iterations: 2700
306 Iterations: 2700 Loss: 8.577576954849064e-05. Validation Accuracy: 99.26499938964844
307 Epoch: 9
308 Iterations: 2710
309 Iterations: 2720
310 Iterations: 2730
311 Iterations: 2740
312 Iterations: 2750
313 Iterations: 2760
314 Iterations: 2770
315 Iterations: 2780
316 Iterations: 2790
317 Iterations: 2800
318 Iterations: 2800 Loss: 9.463933383813128e-05. Validation Accuracy: 99.31499481201172
319 Iterations: 2810
320 Iterations: 2820
321 Iterations: 2830
322 Iterations: 2840
323 Iterations: 2850
324 Iterations: 2860
325 Iterations: 2870
326 Iterations: 2880
327 Iterations: 2890
328 Iterations: 2900
329 Iterations: 2900 Loss: 0.00012100701133022085. Validation Accuracy: 99.2699966430664
330 Iterations: 2910
331 Iterations: 2920
332 Iterations: 2930
333 Iterations: 2940
334 Iterations: 2950
335 Iterations: 2960
336 Iterations: 2970
337 Iterations: 2980
338 Iterations: 2990
339 Iterations: 3000
340 Iterations: 3000 Loss: 5.667606728820829e-06. Validation Accuracy: 99.32499694824219
341 Iterations: 3000 Loss: 5.667606728820829e-06. Test Accuracy: 99.3199969482422

```