In [1]:
```
# -*- coding: utf-8 -*-
import random
import torch
from torch.autograd import Variable


class DynamicNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        """
        In the constructor we construct three nn.Linear instances that we
will use
        in the forward pass.
        """
        super(DynamicNet, self).__init__()
        self.input_linear = torch.nn.Linear(D_in, H)
        self.middle_linear = torch.nn.Linear(H, H)
        self.output_linear = torch.nn.Linear(H, D_out)

    def forward(self, x):
        """
        For the forward pass of the model, we randomly choose either 0, 1,
 2, or 3
        and reuse the middle_linear Module that many times to compute hidd
en layer
        representations.

        Since each forward pass builds a dynamic computation graph, we can
 use normal
        Python control-flow operators like loops or conditional statements
 when
        defining the forward pass of the model.

        Here we also see that it is perfectly safe to reuse the same Modul
e many
        times when defining a computational graph. This is a big improveme
nt from Lua
        Torch, where each Module could be used only once.
        """
        h_relu = self.input_linear(x).clamp(min=0)
        for _ in range(random.randint(0, 3)):
            h_relu = self.middle_linear(h_relu).clamp(min=0)
        y_pred = self.output_linear(h_relu)
        return y_pred


# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs, and wrap them in Varia
bles
x = Variable(torch.randn(N, D_in))
y = Variable(torch.randn(N, D_out), requires_grad=False)
```

```
# Construct our model by instantiating the class defined above
model = DynamicNet(D_in, H, D_out)

# Construct our loss function and an Optimizer. Training this strange mode
l with
# vanilla stochastic gradient descent is tough, so we use momentum
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4, momentum=0.9)
for t in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    y_pred = model(x)

    # Compute and print loss
    loss = criterion(y_pred, y)
    print(t, loss.data[0])

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

```
0 617.4964599609375
1 586.7207641601562
2 594.462890625
3 584.2737426757812
4 582.7604370117188
5 448.11956787109375
6 581.6450805664062
7 560.4078369140625
8 550.8650512695312
9 326.7133483886719
10 526.0846557617188
11 271.10186767578125
12 497.96246337890625
13 480.2035827636719
14 574.2371826171875
15 171.4656982421875
16 562.2449340820312
17 129.00836181640625
18 567.35595703125
19 91.0340347290039
20 560.6006469726562
21 64.0407485961914
22 520.9205322265625
23 545.0962524414062
24 494.73541259765625
25 474.9066467285156
26 449.0616760253906
27 258.177490234375
28 471.19976806640625
29 351.294189453125
30 410.5562744140625
31 369.94610595703125
32 247.9542694091797
33 232.91111755371094
34 220.1400146484375
35 180.5821533203125
```

```
36 195.04095458984375
37 170.70648193359375
38 147.42210388183594
39 130.0604705810547
40 129.25
41 175.88394165039062
42 159.11351013183594
43 110.16151428222656
44 90.27572631835938
45 85.23445129394531
46 140.77098083496094
47 80.3070068359375
48 299.40576171875
49 175.52284240722656
50 160.46807861328125
51 112.88336181640625
52 160.72097778320312
53 88.89665985107422
54 84.09154510498047
55 95.96849060058594
56 105.40737915039062
57 161.2249298095703
58 59.2724494934082
59 150.36978149414062
60 32.99752426147461
61 63.87287139892578
62 77.5291976928711
63 28.106475830078125
64 56.13401794433594
65 45.39598846435547
66 69.26172637939453
67 33.775001525878906
68 50.69169616699219
69 46.71824264526367
70 29.331144332885742
71 17.41274070739746
72 23.169130325317383
73 60.33283233642578
74 43.14490509033203
75 21.277198791503906
76 20.213865280151367
77 38.46455383300781
78 37.77091598510742
79 13.811922073364258
80 11.173827171325684
81 22.871294021606445
82 35.649566650390625
83 14.941598892211914
84 14.705291748046875
85 14.09249496459961
86 12.003541946411133
87 15.535683631896973
88 11.029898643493652
89 13.333113670349121
90 7.82374382019043
91 10.962665557861328
92 7.802312850952148
```

```
93 18.658042907714844
94 6.890983581542969
95 7.18881893157959
96 6.156620979309082
97 7.817047595977783
98 4.329520225524902
99 6.894481182098389
100 12.170260429382324
101 6.716583728790283
102 5.814269542694092
103 17.671649932861328
104 4.881534099578857
105 14.49454116821289
106 10.76485538482666
107 7.746699333190918
108 8.56892204284668
109 5.658292770385742
110 5.030376434326172
111 5.626359462738037
112 8.0494966506958
113 4.980729103088379
114 4.507913589477539
115 6.668847560882568
116 3.400421380996704
117 5.216052055358887
118 2.8582923412323
119 5.035233497619629
120 4.2797088623046875
121 2.0455820560455322
122 3.380805253982544
123 2.1711719036102295
124 3.088573932647705
125 4.6662211418151855
126 3.303863286972046
127 2.1570217609405518
128 4.15886116027832
129 2.5761682987213135
130 2.398016929626465
131 3.5235066413879395
132 4.511153221130371
133 2.107137441635132
134 1.5901541709899902
135 3.369044780731201
136 3.3211681842803955
137 3.6903843879699707
138 2.796448230743408
139 1.6557893753051758
140 2.0419225692749023
141 1.5717331171035767
142 2.225872278213501
143 1.8292890787124634
144 4.478525161743164
145 1.6952733993530273
146 1.859634280204773
147 4.831842422485352
148 0.9982556700706482
149 2.1715002059936523
```

```
150 1.9586626291275024
151 2.1136515140533447
152 1.3249876499176025
153 2.5335443019866943
154 0.9659452438354492
155 1.6678433418273926
156 1.3219949007034302
157 4.030327796936035
158 0.9548989534378052
159 3.8564045429229736
160 0.6781408786773682
161 2.7413361072540283
162 2.3992137908935547
163 1.5485725402832031
164 1.6324256658554077
165 1.5675405263900757
166 2.9987711906433105
167 1.915053367614746
168 1.0318866968154907
169 2.04632306098938
170 5.08028507232666
171 1.0896153450012207
172 1.2423239946365356
173 14.853652954101562
174 1.6805720329284668
175 2.0311498641967773
176 4.8007025718688965
177 16.79224395751953
178 0.9883328676223755
179 2.6131718158721924
180 4.563420295715332
181 8.250673294067383
182 2.6278228759765625
183 3.5495855808258057
184 1.1998058557510376
185 1.2639880180358887
186 1.2520971298217773
187 2.652158737182617
188 1.2062733173370361
189 2.8759870529174805
190 1.5796542167663574
191 3.877002477645874
192 2.7653510570526123
193 8.453703880310059
194 6.103426933288574
195 1.9580942392349243
196 2.4415130615234375
197 5.864106178283691
198 11.303057670593262
199 0.9691779613494873
200 22.905794143676758
201 7.470695972442627
202 1.1197893619537354
203 15.502533912658691
204 3.773719549179077
205 1.4057092666625977
206 2.066617965698242
```

```
207 10.620091438293457
208 0.6033884286880493
209 1.7042262554168701
210 2.14969801902771
211 1.1560842990875244
212 2.2531049251556396
213 0.6178454160690308
214 2.419261932373047
215 3.7376160621643066
216 3.54567289352417
217 6.498558521270752
218 3.392005443572998
219 6.798036098480225
220 0.2540091872215271
221 1.5067037343978882
222 6.246980667114258
223 3.2084572315216064
224 3.7739193439483643
225 1.4191114902496338
226 0.6949692964553833
227 16.48684310913086
228 8.658540725708008
229 2.4220564365386963
230 4.220971584320068
231 7.836079120635986
232 20.76011085510254
233 0.4346191883087158
234 11.331007957458496
235 5.6215314865112305
236 4.780257701873779
237 3.506639003753662
238 1.2792421579360962
239 7.938282012939453
240 6.497415065765381
241 2.939133644104004
242 3.32114315032959
243 2.800471544265747
244 5.428300380706787
245 0.6228451132774353
246 0.3326306641101837
247 7.4255571365356445
248 1.00912606716156
249 2.436945915222168
250 1.5155794620513916
251 1.9352447986602783
252 2.458794355392456
253 6.955153465270996
254 3.1197938919067383
255 2.2026214599609375
256 0.3747260272502899
257 2.607583999633789
258 0.5932360887527466
259 7.845485210418701
260 2.0091772079467773
261 8.53829574584961
262 2.1867425441741943
263 2.263881206512451
```

```
264 1.0491591691970825
265 0.48314815759658813
266 0.9175407290458679
267 1.1300252676010132
268 24.631303787231445
269 1.4745445251464844
270 6.909215450286865
271 12.641260147094727
272 57.27519226074219
273 0.5653321743011475
274 4.688138008117676
275 9.574002265930176
276 95.72614288330078
277 24.40271759033203
278 3.313650369644165
279 13.107523918151855
280 20.444656372070312
281 25.748130798339844
282 21.845626831054688
283 6.494312286376953
284 2.6667439937591553
285 11.61217975616455
286 12.990994453430176
287 7.207010746002197
288 9.37540054321289
289 6.768373489379883
290 3.8008391857147217
291 2.7420644760131836
292 3.5836541652679443
293 36.96145248413086
294 23.076881408691406
295 7.872142314910889
296 7.252026081085205
297 61.25562286376953
298 4.7811665534973145
299 4.283871650695801
300 6.03846549987793
301 16.74391746520996
302 14.682907104492188
303 12.277552604675293
304 8.576719284057617
305 8.677407264709473
306 10.543718338012695
307 6.608772277832031
308 5.564152717590332
309 3.348787546157837
310 7.60455846786499
311 7.37207555770874
312 3.251549482345581
313 13.723657608032227
314 3.4809303283691406
315 6.241969108581543
316 2.031262159347534
317 0.9793280363082886
318 3.077669382095337
319 5.346007347106934
320 5.276621341705322
```

```
321 27.445606231689453
322 28.65699005126953
323 26.991525650024414
324 38.68727111816406
325 55.18824768066406
326 12.477653503417969
327 4.929076671600342
328 13.613712310791016
329 14.591026306152344
330 15.918105125427246
331 14.886709213256836
332 5.447444915771484
333 5.6419878005981445
334 25.929611206054688
335 9.424022674560547
336 6.24612283706665
337 4.999917030334473
338 4.090578079223633
339 8.248003005981445
340 17.481401443481445
341 1.399496078491211
342 2.0626049041748047
343 6.712996482849121
344 6.91818380355835
345 4.869098663330078
346 2.8310725688934326
347 5.74797248840332
348 2.603666305541992
349 7.299453258514404
350 1.7987719774246216
351 1.1637463569641113
352 2.222529888153076
353 7.670896530151367
354 1.3952560424804688
355 1.0159368515014648
356 1.2963148355484009
357 2.6970505714416504
358 1.865067958831787
359 0.8906550407409668
360 1.1378997564315796
361 1.3592840433120728
362 1.199123740196228
363 3.1202688217163086
364 2.5934956073760986
365 0.7691701650619507
366 2.9970085620880127
367 0.9741449952125549
368 0.4719432294368744
369 0.9354196190834045
370 1.8485444784164429
371 0.470345675945282
372 1.4169516563415527
373 1.1111966371536255
374 0.45924073457717896
375 1.3810803890228271
376 0.38162529468536377
377 1.035822868347168
```

```
378 1.039865255355835
379 1.013710379600525
380 0.24925124645233154
381 0.9570943713188171
382 0.2069171965122223
383 1.1371022462844849
384 0.781284749507904
385 0.5542237162590027
386 0.22487430274486542
387 0.7474405765533447
388 0.6208946108818054
389 0.19531965255737305
390 0.5539401769638062
391 0.226301908493042
392 0.40292954444885254
393 1.694028615951538
394 1.4330745935440063
395 0.14397381246089935
396 0.5758671760559082
397 0.14175888895988464
398 0.12704035639762878
399 0.10257995873689651
400 2.051517963409424
401 1.0105328559875488
402 1.0537744760513306
403 1.2365745306015015
404 1.4436901807785034
405 0.42905786633491516
406 0.8277648091316223
407 0.1363527923822403
408 0.9801706671714783
409 0.9845675230026245
410 0.17707951366901398
411 0.6263932585716248
412 0.10676196217536926
413 1.6918052434921265
414 0.08687221258878708
415 0.9948474168777466
416 0.8633851408958435
417 0.8391584753990173
418 0.9019174575805664
419 0.7176405191421509
420 0.09812083840370178
421 0.8930495381355286
422 0.91794753074646
423 0.6222383379936218
424 0.5952858924865723
425 0.5641317367553711
426 1.110972285270691
427 0.12041153013706207
428 0.13540488481521606
429 0.15196429193019867
430 1.2471449375152588
431 0.0910307914018631
432 0.3556502163410187
433 0.35611391067504883
434 0.0994490534067154
```

```
435 0.11399756371974945
436 1.6917039155960083
437 0.046628717333078384
438 0.6359814405441284
439 1.1207636594772339
440 0.7801993489265442
441 0.46574991941452026
442 0.04961245507001877
443 0.05201117694377899
444 1.5348111391067505
445 0.06092946603894234
446 0.5069867372512817
447 0.06881222873926163
448 0.9531847238540649
449 0.19873511791229248
450 0.8506504893302917
451 0.2495613694190979
452 0.15855902433395386
453 0.5126480460166931
454 0.048380088061094284
455 0.5856323838233948
456 0.6962810754776001
457 0.2926711142063141
458 0.49724799394607544
459 0.25228527188301086
460 1.1223406791687012
461 0.6005898118019104
462 0.6434618830680847
463 0.8075220584869385
464 0.6333624720573425
465 0.4601361155509949
466 0.4571605324745178
467 0.6399435997009277
468 0.4992857277393341
469 0.4062232971191406
470 0.17993730306625366
471 0.8852510452270508
472 0.516291081905365
473 0.3104937672615051
474 0.5995038747787476
475 0.7114810943603516
476 0.07153534889221191
477 0.2936292290687561
478 0.8982550501823425
479 0.6369932293891907
480 0.47761425375938416
481 0.08314242959022522
482 0.8414137363433838
483 0.7177855372428894
484 0.34585222601890564
485 0.2613247334957123
486 0.45507505536079407
487 0.22859375178813934
488 0.6225958466529846
489 0.06258411705493927
490 0.3562828600406647
491 0.376590758562088
```

```
492 0.5920135378837585
493 0.25771933794021606
494 0.662420392036438
495 0.5979882478713989
496 0.4690578579902649
497 0.45877304673194885
498 0.42165857553482056
499 0.3244536221027374
```

In [3]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import torch
from torch import autograd
import torch.nn.functional as F

images = np.load("D:/work/JHUschoolStuff/machinelearning/project1/cs475_pr
oject_data/images.npy")
labels = np.load("D:/work/JHUschoolStuff/machinelearning/project1/cs475_pr
oject_data/labels.npy")
test = np.load("D:/work/JHUschoolStuff/machinelearning/project1/cs475_proj
ect_data/test_images.npy")
height = images.shape[1]
width = images.shape[2]
size = height * width
images = (images - images.mean()) / images.std()
data = images.reshape(images.shape[0],size)
test_data = test.reshape(test.shape[0], size)
test_data = (test_data - test_data.mean()) / test_data.std()
batch_size = 1
NUM_OPT_STEPS = 5000
train_seqs = data[0:45000,:]
train_labels = labels[0:45000]
val_seqs = data[45000:,:]
val_labels = labels[45000:]
```

In [12]:
```
class_1 = images[labels == 0]
class_2 = images[labels == 1]
class_3 = images[labels == 2]
class_4 = images[labels == 3]
class_5 = images[labels == 4]

print(len(class_1))
print(len(class_2))
print(len(class_3))
print(len(class_4))
print(len(class_5))
```

```
10000
10000
10000
10000
10000
```

Using a random classifier would give you an accuracy of 0.2 because you have a 1/5 chance of getting a prediction correct. A majority vote classifier would get also an accuracy of 0.2 because there are 10000 images of each kind. Therefore we would only get 10000/50000 predictions correct which is 0.2

In [2]:
```
class LinearModel(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = torch.nn.Linear(height * width, 5)
    def forward(self, x):
        x = self.linear(x)
        return x
```

From the documentation, torch.nn.Linear creates "in features *out features" number of weights and "out features" number of biases(which are booleans). In our case our number of in features is 26*26 = 676 and our number of out features is 5. Therefore the number of weights we have is 676*5 and the number of biases will be 5. This makes sense because our input is a vector of 676 and we have 5 different classes to predict and therefore we would need at least 676*5 weights to compute our prediction, and a bias associated with each prediction.

In [3]:
```
model = LinearModel()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-6)
```

In our previous homeworks we implemented were torch.nn.SGD and torch.nn.Adam, which are the stochastic gradient descent and Adam optimizations. The two most important arguments these optimizers need us to provide are the model parameters themselves and the rate of update. These two allow us to update the weights of our model with a provided rate.

In [4]:
```
def train(batch_size):
    # model.train() puts our model in train mode, which can require differ
ent
    # behavior than eval mode (for example in the case of dropout).
    model.train()
    # i is is a 1-D array with shape [batch_size]
    i = np.random.choice(train_seqs.shape[0], size=batch_size, replace=Fal
se)
    x = autograd.Variable(torch.from_numpy(train_seqs[i].astype(np.float32
)))
    y = autograd.Variable(torch.from_numpy(train_labels[i].astype(np.int))
).long()
    optimizer.zero_grad()
    y_hat_ = model(x)
    loss = F.cross_entropy(y_hat_, y)
    loss.backward()
    optimizer.step()
    return loss.data[0]
```

In [5]:
```
def approx_train_accuracy(model):
    i = np.random.choice(train_seqs.shape[0], size=1000, replace=False)
    x = autograd.Variable(torch.from_numpy(train_seqs[i].astype(np.float32
)))
    y = autograd.Variable(torch.from_numpy(train_labels[i].astype(np.int))
)
    y_hat_ = model(x)
    y_hat = np.zeros(1000)
    for i in range(1000):
        y_hat[i] = torch.max(y_hat_[i,:].data, 0)[1][0]
    return accuracy(y_hat, y.data.numpy())
```

In [6]:
```
def val_accuracy(model):
    x = autograd.Variable(torch.from_numpy(val_seqs.astype(np.float32)))
    y = autograd.Variable(torch.from_numpy(val_labels.astype(np.int)))
    y_hat_ = model(x)
    y_hat = np.zeros(5000)
    for i in range(5000):
        y_hat[i] = torch.max(y_hat_[i,:].data, 0)[1][0]
    return accuracy(y_hat, y.data.numpy())
```

In [7]:
```
def accuracy(y, y_hat):
    return (y == y_hat).astype(np.float).mean()
```

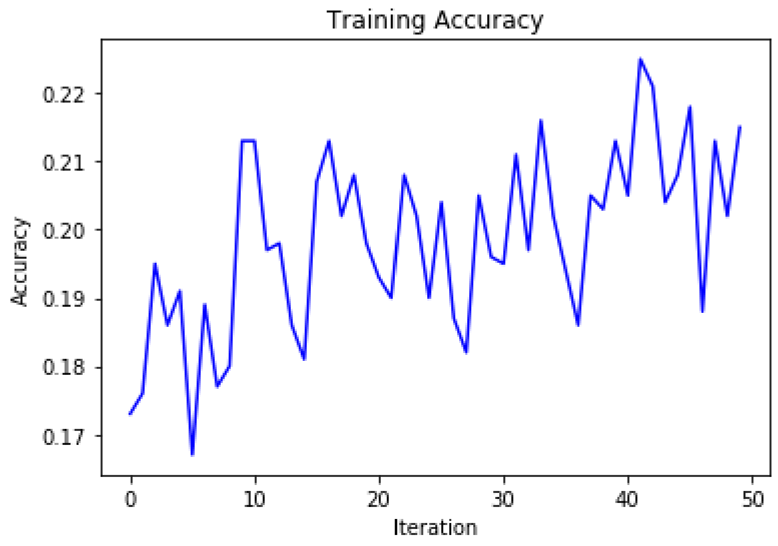
In [8]:
```
def plot(train_accs, val_accs):
    plt.figure(200)
    plt.title('Training Accuracy')
    plt.xlabel('Iteration')
    plt.ylabel('Accuracy')
    plt.plot(train_accs, 'b')
    plt.show()
    plt.figure(300)
    plt.title('Validation Accuracy')
    plt.xlabel('Iteration')
    plt.ylabel('Accuracy')
    plt.plot(val_accs, 'b')
    plt.show()
```

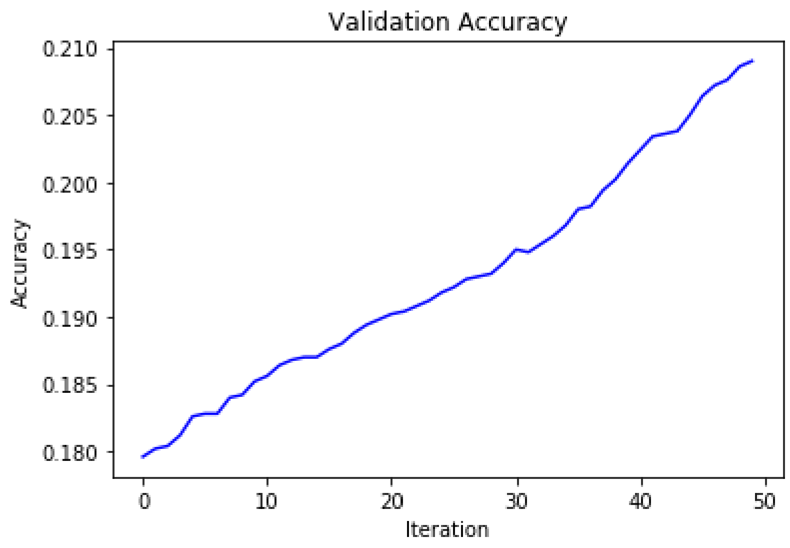
In [9]:
```
def runModel(model, batch_size):
    train_accs, val_accs = [], []
    for i in range(NUM_OPT_STEPS):
        train(batch_size)
        if i % 100 == 0:
            train_accs.append(approx_train_accuracy(model))
            val_accs.append(val_accuracy(model))
            print("%6d %5.2f %5.2f" % (i, train_accs[-1], val_accs[-1]))
    plot(train_accs, val_accs)
```

In [10]:
```
runModel(model, 1)
```

```
     0  0.17  0.18
   100  0.18  0.18
   200  0.20  0.18
   300  0.19  0.18
   400  0.19  0.18
   500  0.17  0.18
   600  0.19  0.18
   700  0.18  0.18
   800  0.18  0.18
   900  0.21  0.19
  1000  0.21  0.19
  1100  0.20  0.19
  1200  0.20  0.19
  1300  0.19  0.19
  1400  0.18  0.19
  1500  0.21  0.19
  1600  0.21  0.19
  1700  0.20  0.19
```

```
1800  0.21  0.19
1900  0.20  0.19
2000  0.19  0.19
2100  0.19  0.19
2200  0.21  0.19
2300  0.20  0.19
2400  0.19  0.19
2500  0.20  0.19
2600  0.19  0.19
2700  0.18  0.19
2800  0.20  0.19
2900  0.20  0.19
3000  0.20  0.20
3100  0.21  0.19
3200  0.20  0.20
3300  0.22  0.20
3400  0.20  0.20
3500  0.19  0.20
3600  0.19  0.20
3700  0.20  0.20
3800  0.20  0.20
3900  0.21  0.20
4000  0.20  0.20
4100  0.23  0.20
4200  0.22  0.20
4300  0.20  0.20
4400  0.21  0.20
4500  0.22  0.21
4600  0.19  0.21
4700  0.21  0.21
4800  0.20  0.21
4900  0.21  0.21
```





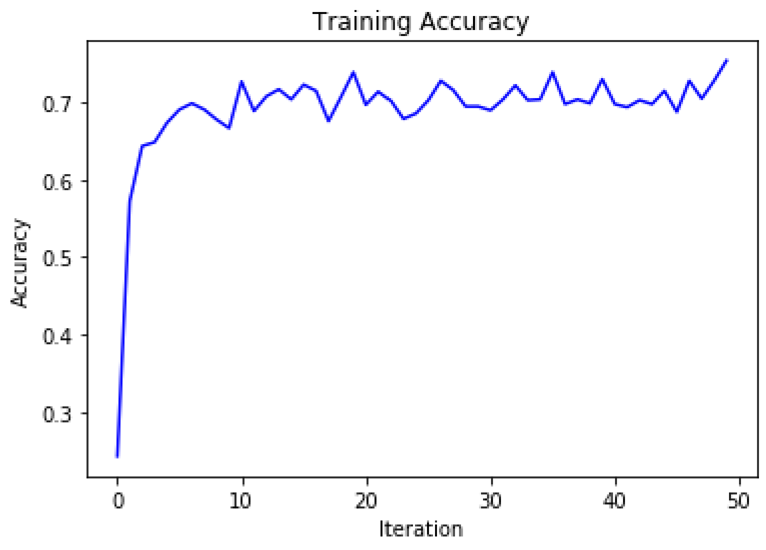
The top train and validation accuracies we reached were 0.23 for training and 0.21 for validation. The problem here is that we may not have the best hyperparameters for our model. Our learning rate could have been too small which means that our model was not making enough progress toward an optima within our 5k optimization steps.

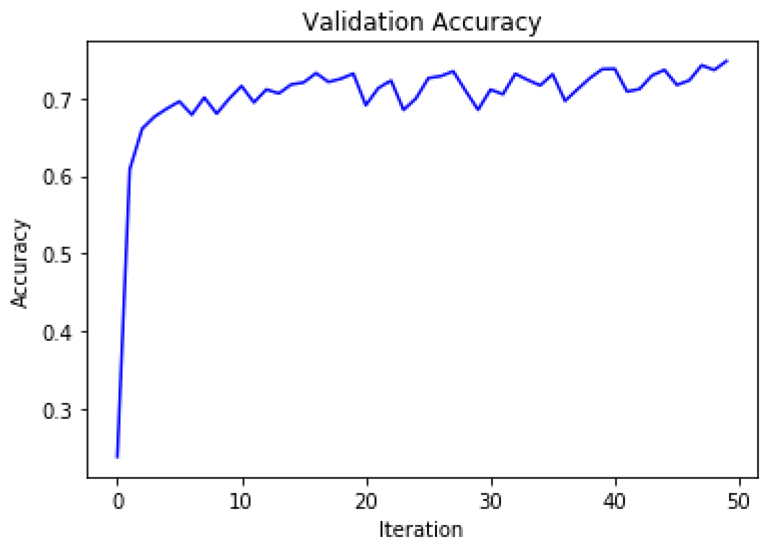
In [11]:
```
model = LinearModel()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

In [12]:
```
runModel(model, 1)
```

```
   0  0.24  0.24
 100  0.57  0.61
 200  0.64  0.66
 300  0.65  0.68
 400  0.67  0.69
 500  0.69  0.70
 600  0.70  0.68
 700  0.69  0.70
 800  0.68  0.68
 900  0.67  0.70
1000  0.73  0.72
1100  0.69  0.69
1200  0.71  0.71
1300  0.72  0.71
1400  0.70  0.72
1500  0.72  0.72
1600  0.71  0.73
1700  0.68  0.72
1800  0.71  0.73
1900  0.74  0.73
2000  0.70  0.69
2100  0.71  0.71
2200  0.70  0.72
2300  0.68  0.69
2400  0.69  0.70
2500  0.70  0.73
2600  0.73  0.73
2700  0.71  0.73
```

```
2800  0.69  0.71
2900  0.69  0.69
3000  0.69  0.71
3100  0.70  0.71
3200  0.72  0.73
3300  0.70  0.72
3400  0.70  0.72
3500  0.74  0.73
3600  0.70  0.70
3700  0.70  0.71
3800  0.70  0.73
3900  0.73  0.74
4000  0.70  0.74
4100  0.69  0.71
4200  0.70  0.71
4300  0.70  0.73
4400  0.71  0.74
4500  0.69  0.72
4600  0.73  0.72
4700  0.70  0.74
4800  0.73  0.74
4900  0.75  0.75
```





The final optimizer used was the Adam optimizer with a learning rate of 0.001. This lead to an accuracy in the mid 70s. The best validation accuracy achieved was 76.

In [18]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import torch
from torch import autograd
import torch.nn.functional as F
import time

images = np.load("D:/work/JHUschoolStuff/machinelearning/project1/cs475_pr
oject_data/images.npy")
labels = np.load("D:/work/JHUschoolStuff/machinelearning/project1/cs475_pr
oject_data/labels.npy")
test = np.load("D:/work/JHUschoolStuff/machinelearning/project1/cs475_proj
ect_data/test_images.npy")
height = images.shape[1]
width = images.shape[2]
size = height * width
images = (images - images.mean()) / images.std()
data = images.reshape(images.shape[0],size)
test_data = test.reshape(test.shape[0], size)
test_data = (test_data - test_data.mean()) / test_data.std()
NUM_OPT_STEPS = 5000
train_seqs = data[0:45000,:]
train_labels = labels[0:45000]
val_seqs = data[45000:,:]
val_labels = labels[45000:]
```

In [19]:
```
class TwoLayerNN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.layer_1 = torch.nn.Linear(height * width, 100)
        self.layer_2 = torch.nn.Linear(100, 5)
    def forward(self, x):
        x = self.layer_1(x)
        y = F.relu(x)
        z = self.layer_2(y)
        return z
```

In [20]:
```
def train(batch_size):
    # model.train() puts our model in train mode, which can require differ
ent
    # behavior than eval mode (for example in the case of dropout).
    model.train()
    # i is is a 1-D array with shape [batch_size]
    i = np.random.choice(train_seqs.shape[0], size=batch_size, replace=Fal
se)
    x = autograd.Variable(torch.from_numpy(train_seqs[i].astype(np.float32
)))
    y = autograd.Variable(torch.from_numpy(train_labels[i].astype(np.int))
).long()
    optimizer.zero_grad()
    y_hat_ = model(x)
    loss = F.cross_entropy(y_hat_, y)
    loss.backward()
    optimizer.step()
    return loss.data[0]
```

In [21]:
```
def approx_train_accuracy(model):
    i = np.random.choice(train_seqs.shape[0], size=1000, replace=False)
    x = autograd.Variable(torch.from_numpy(train_seqs[i].astype(np.float32
)))
    y = autograd.Variable(torch.from_numpy(train_labels[i].astype(np.int))
)
    y_hat_ = model(x)
    y_hat = np.zeros(1000)
    for i in range(1000):
        y_hat[i] = torch.max(y_hat_[i,:].data, 0)[1][0]
    return accuracy(y_hat, y.data.numpy())
```

In [22]:
```
def val_accuracy(model):
    x = autograd.Variable(torch.from_numpy(val_seqs.astype(np.float32)))
    y = autograd.Variable(torch.from_numpy(val_labels.astype(np.int)))
    y_hat_ = model(x)
    y_hat = np.zeros(5000)
    for i in range(5000):
        y_hat[i] = torch.max(y_hat_[i,:].data, 0)[1][0]
    return accuracy(y_hat, y.data.numpy())
```

In [23]:
```
def accuracy(y, y_hat):
    return (y == y_hat).astype(np.float).mean()
```

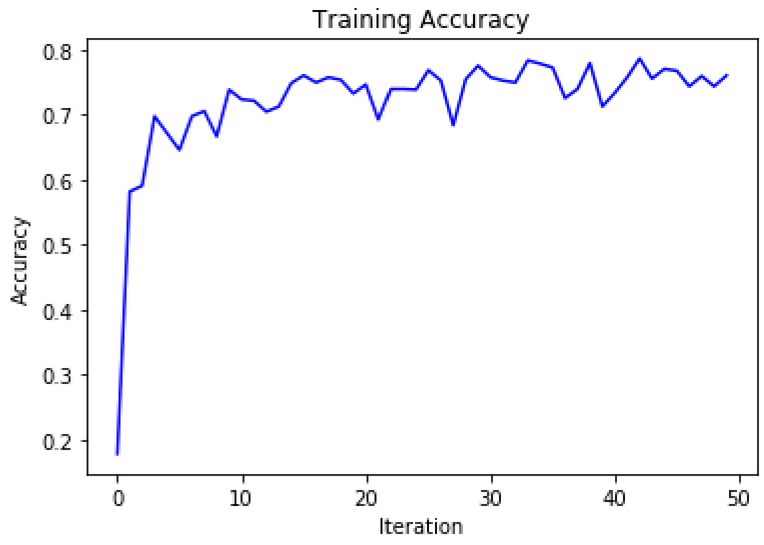
In [24]:
```
def plot(train_accs, val_accs):
    plt.figure(200)
    plt.title('Training Accuracy')
    plt.xlabel('Iteration')
    plt.ylabel('Accuracy')
    plt.plot(train_accs, 'b')
    plt.show()
    plt.figure(300)
    plt.title('Validation Accuracy')
    plt.xlabel('Iteration')
    plt.ylabel('Accuracy')
    plt.plot(val_accs, 'b')
    plt.show()
```

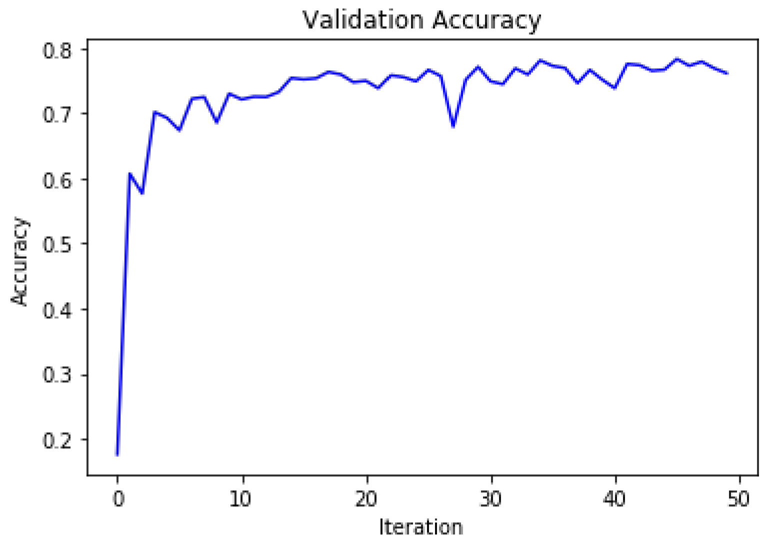
In [25]:
```
def runModel(model, batch_size):
    train_accs, val_accs = [], []
    for i in range(NUM_OPT_STEPS):
        train(batch_size)
        if i % 100 == 0:
            train_accs.append(approx_train_accuracy(model))
            val_accs.append(val_accuracy(model))
            print("%6d %5.2f %5.2f" % (i, train_accs[-1], val_accs[-1]))
    plot(train_accs, val_accs)
```

In [26]:
```
model = TwoLayerNN()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

```
runModel(model, 1)
```

```
   0  0.18  0.18
 100  0.58  0.61
 200  0.59  0.58
 300  0.70  0.70
 400  0.67  0.69
 500  0.65  0.67
 600  0.70  0.72
 700  0.71  0.72
 800  0.67  0.69
 900  0.74  0.73
1000  0.72  0.72
1100  0.72  0.73
1200  0.70  0.72
1300  0.71  0.73
1400  0.75  0.75
1500  0.76  0.75
1600  0.75  0.75
1700  0.76  0.76
1800  0.75  0.76
1900  0.73  0.75
2000  0.75  0.75
2100  0.69  0.74
2200  0.74  0.76
2300  0.74  0.76
2400  0.74  0.75
2500  0.77  0.77
2600  0.75  0.76
2700  0.68  0.68
2800  0.76  0.75
2900  0.78  0.77
3000  0.76  0.75
3100  0.75  0.74
3200  0.75  0.77
3300  0.78  0.76
3400  0.78  0.78
3500  0.77  0.77
3600  0.73  0.77
3700  0.74  0.75
3800  0.78  0.77
3900  0.71  0.75
4000  0.73  0.74
4100  0.76  0.78
4200  0.79  0.77
4300  0.76  0.77
4400  0.77  0.77
4500  0.77  0.78
4600  0.74  0.77
4700  0.76  0.78
4800  0.74  0.77
4900  0.76  0.76
```



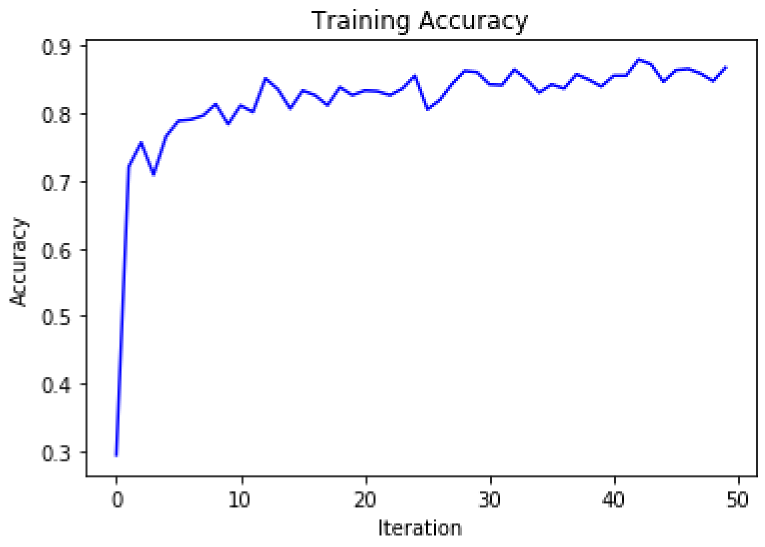


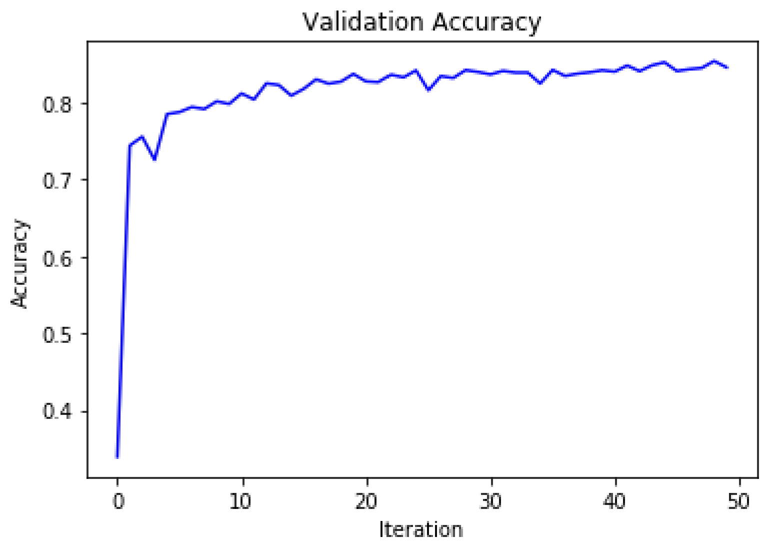
In [27]:
```
for m in model.children():
    m.reset_parameters()
```

In [28]:
```
model = TwoLayerNN()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
runModel(model, 10)
```

```
   0  0.29  0.34
 100  0.72  0.74
 200  0.76  0.76
 300  0.71  0.73
 400  0.77  0.79
 500  0.79  0.79
 600  0.79  0.79
 700  0.80  0.79
 800  0.81  0.80
 900  0.78  0.80
1000  0.81  0.81
1100  0.80  0.80
1200  0.85  0.83
1300  0.84  0.82
1400  0.81  0.81
```

```
1500  0.83  0.82
1600  0.83  0.83
1700  0.81  0.82
1800  0.84  0.83
1900  0.83  0.84
2000  0.83  0.83
2100  0.83  0.83
2200  0.83  0.84
2300  0.84  0.83
2400  0.86  0.84
2500  0.81  0.82
2600  0.82  0.83
2700  0.84  0.83
2800  0.86  0.84
2900  0.86  0.84
3000  0.84  0.84
3100  0.84  0.84
3200  0.86  0.84
3300  0.85  0.84
3400  0.83  0.82
3500  0.84  0.84
3600  0.84  0.83
3700  0.86  0.84
3800  0.85  0.84
3900  0.84  0.84
4000  0.86  0.84
4100  0.86  0.85
4200  0.88  0.84
4300  0.87  0.85
4400  0.85  0.85
4500  0.86  0.84
4600  0.87  0.84
4700  0.86  0.85
4800  0.85  0.85
4900  0.87  0.85
```

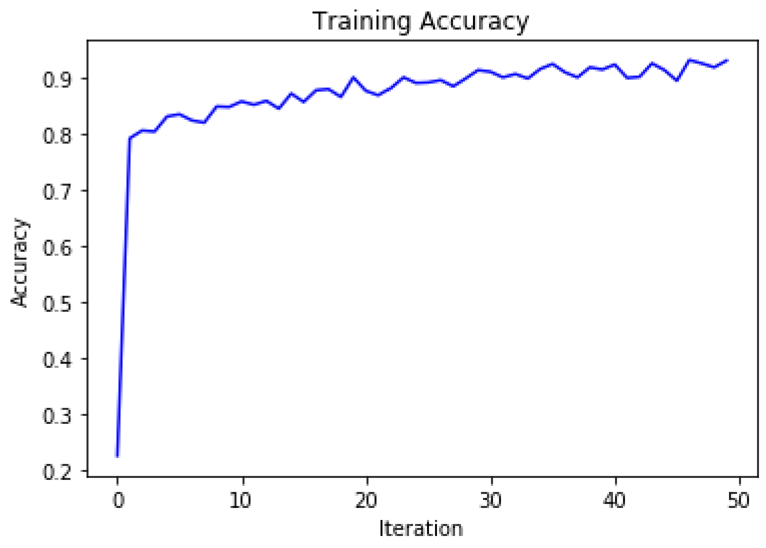


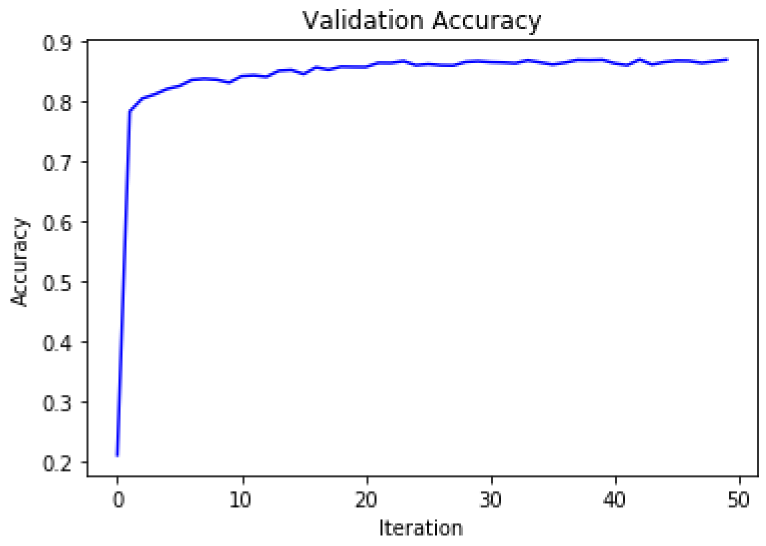


In [29]:
```
for m in model.children():
    m.reset_parameters()
```

In [30]:
```
start = time.time()
runModel(model, 64)
end = time.time()
print(end - start)
```

```
       0   0.23   0.21
     100   0.79   0.78
     200   0.81   0.80
     300   0.81   0.81
     400   0.83   0.82
     500   0.84   0.83
     600   0.82   0.84
     700   0.82   0.84
     800   0.85   0.84
     900   0.85   0.83
    1000   0.86   0.84
    1100   0.85   0.84
    1200   0.86   0.84
    1300   0.85   0.85
    1400   0.87   0.85
    1500   0.86   0.84
    1600   0.88   0.86
    1700   0.88   0.85
    1800   0.87   0.86
    1900   0.90   0.86
    2000   0.88   0.86
    2100   0.87   0.86
    2200   0.88   0.86
    2300   0.90   0.87
    2400   0.89   0.86
    2500   0.89   0.86
    2600   0.90   0.86
    2700   0.89   0.86
    2800   0.90   0.87
    2900   0.92   0.87
    3000   0.91   0.86
    3100  0.90  0.86
    3200  0.91  0.86
    3300  0.90  0.87
    3400  0.92  0.86
    3500  0.93  0.86
    3600  0.91  0.86
    3700  0.90  0.87
    3800  0.92  0.87
    3900  0.92  0.87
    4000  0.93  0.86
    4100  0.90  0.86
    4200  0.90  0.87
    4300  0.93  0.86
    4400  0.92  0.87
    4500  0.90  0.87
    4600  0.93  0.87
    4700  0.93  0.86
    4800  0.92  0.87
    4900  0.93  0.87
```





```
18.778249740600586
```

The best validation accuracy I achieved was 88. The batch size used was 64 and the learning rate was 0.001. I used 5000 optimization steps to reach this accuracy. Initially I had tried 10k however it seemed to not improve after about 5k steps. Training only took ~19 seconds.

In [17]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import torch
from torch import autograd
import torch.nn.functional as F
import time

images = np.load("D:/work/JHUschoolStuff/machinelearning/project1/cs475_pr
oject_data/images.npy")
labels = np.load("D:/work/JHUschoolStuff/machinelearning/project1/cs475_pr
oject_data/labels.npy")
test = np.load("D:/work/JHUschoolStuff/machinelearning/project1/cs475_proj
ect_data/test_images.npy")
height = images.shape[1]
width = images.shape[2]
size = height * width
images = (images - images.mean()) / images.std()
data = images.reshape(images.shape[0],size)
test_data = test.reshape(test.shape[0], size)
test_data = (test_data - test_data.mean()) / test_data.std()
NUM_OPT_STEPS = 2000
NUM_CLASSES = 5
train_seqs = data[0:45000,:]
train_labels = labels[0:45000]
val_seqs = data[45000:,:]
val_labels = labels[45000:]
```

In [18]:
```
class TooSimpleConvNN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        # 3x3 convolution that takes in an image with one channel
        # and outputs an image with 8 channels.
        self.conv1 = torch.nn.Conv2d(1, 8, kernel_size=3, stride=2)
        # 3x3 convolution that takes in an image with 8 channels
        # and outputs an image with 16 channels. The output image
        # has approximately half the height and half the width
        # because of the stride of 2.
        self.conv2 = torch.nn.Conv2d(8, 16, kernel_size=3, stride=2)
        # 1x1 convolution that takes in an image with 16 channels and
        # produces an image with 5 channels. Here, the 5 channels
        # will correspond to class scores.
        self.final_conv = torch.nn.Conv2d(16, 5, kernel_size=1)
    def forward(self, x):
        # Convolutions work with images of shape
        # [batch_size, num_channels, height, width]
        x = x.view(-1, height, width).unsqueeze(1)
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        n, c, h, w = x.size()
        x = F.avg_pool2d(x, kernel_size=[h, w])
        x = self.final_conv(x).view(-1, NUM_CLASSES)
        return x
```

A fully connected neural network has all units connected to each other where as Convolutional neural nets only have some close by units connected to each other. This makes convolutional neural nets less expensive.

In [19]:
```
model = TooSimpleConvNN()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

In [20]:
```
def train(batch_size):
    # model.train() puts our model in train mode, which can require differ
ent
    # behavior than eval mode (for example in the case of dropout).
    model.train()
    # i is is a 1-D array with shape [batch_size]
    i = np.random.choice(train_seqs.shape[0], size=batch_size, replace=Fal
se)
    x = autograd.Variable(torch.from_numpy(train_seqs[i].astype(np.float32
)))
    y = autograd.Variable(torch.from_numpy(train_labels[i].astype(np.int))
).long()
    optimizer.zero_grad()
    y_hat_ = model(x)
    loss = F.cross_entropy(y_hat_, y)
    loss.backward()
    optimizer.step()
    return loss.data[0]
```

In [21]:
```
def approx_train_accuracy(model):
    i = np.random.choice(train_seqs.shape[0], size=1000, replace=False)
    x = autograd.Variable(torch.from_numpy(train_seqs[i].astype(np.float32
)))
    y = autograd.Variable(torch.from_numpy(train_labels[i].astype(np.int))
)
    y_hat_ = model(x)
    y_hat = np.zeros(1000)
    for i in range(1000):
        y_hat[i] = torch.max(y_hat_[i,:].data, 0)[1][0]
    return accuracy(y_hat, y.data.numpy())
```

In [22]:
```
def val_accuracy(model):
    x = autograd.Variable(torch.from_numpy(val_seqs.astype(np.float32)))
    y = autograd.Variable(torch.from_numpy(val_labels.astype(np.int)))
    y_hat_ = model(x)
    y_hat = np.zeros(5000)
    for i in range(5000):
        y_hat[i] = torch.max(y_hat_[i,:].data, 0)[1][0]
    return accuracy(y_hat, y.data.numpy())
```

In [23]:
```
def accuracy(y, y_hat):
    return (y == y_hat).astype(np.float).mean()
```

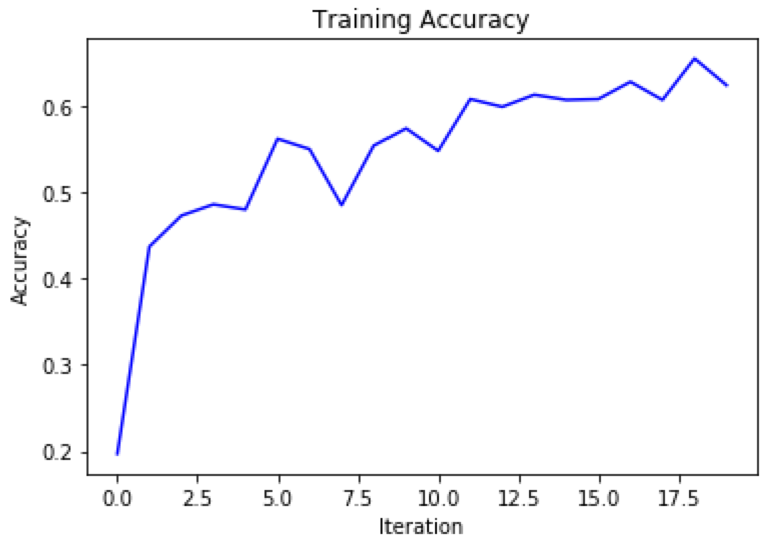
In [24]:
```
def plot(train_accs, val_accs):
    plt.figure(200)
    plt.title('Training Accuracy')
    plt.xlabel('Iteration')
    plt.ylabel('Accuracy')
    plt.plot(train_accs, 'b')
    plt.show()
    plt.figure(300)
    plt.title('Validation Accuracy')
    plt.xlabel('Iteration')
    plt.ylabel('Accuracy')
    plt.plot(val_accs, 'b')
    plt.show()
```

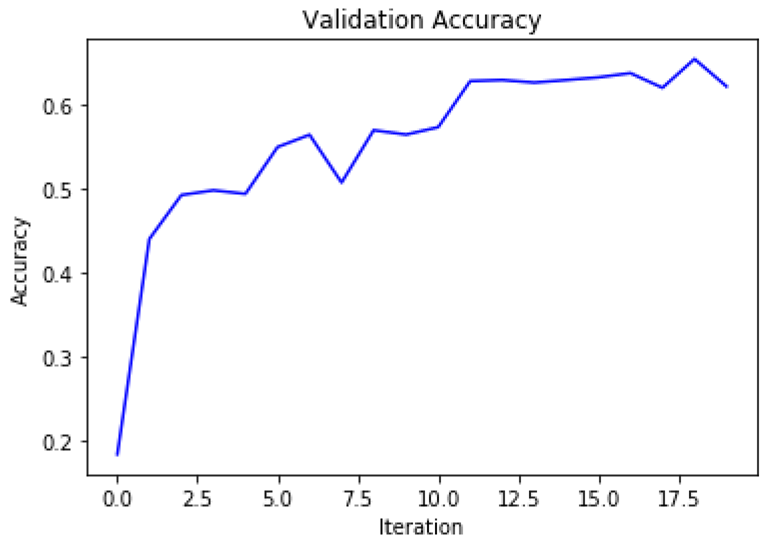
In [25]:
```
def runModel(model, batch_size):
    train_accs, val_accs = [], []
    for i in range(NUM_OPT_STEPS):
        train(batch_size)
        if i % 100 == 0:
            train_accs.append(approx_train_accuracy(model))
            val_accs.append(val_accuracy(model))
            print("%6d %5.2f %5.2f" % (i, train_accs[-1], val_accs[-1]))
    plot(train_accs, val_accs)
```

In [26]:
```
def reset(model):
    for m in model.children():
        m.reset_parameters()
```

In [27]:
```
runModel(model, 10) #2000 steps
```

```
     0  0.20  0.18
   100  0.44  0.44
   200  0.47  0.49
   300  0.49  0.50
   400  0.48  0.49
   500  0.56  0.55
   600  0.55  0.56
   700  0.48  0.51
   800  0.55  0.57
   900  0.57  0.56
  1000  0.55  0.57
  1100  0.61  0.63
  1200  0.60  0.63
  1300  0.61  0.63
  1400  0.61  0.63
  1500  0.61  0.63
  1600  0.63  0.64
  1700  0.61  0.62
  1800  0.66  0.65
  1900  0.62  0.62
```



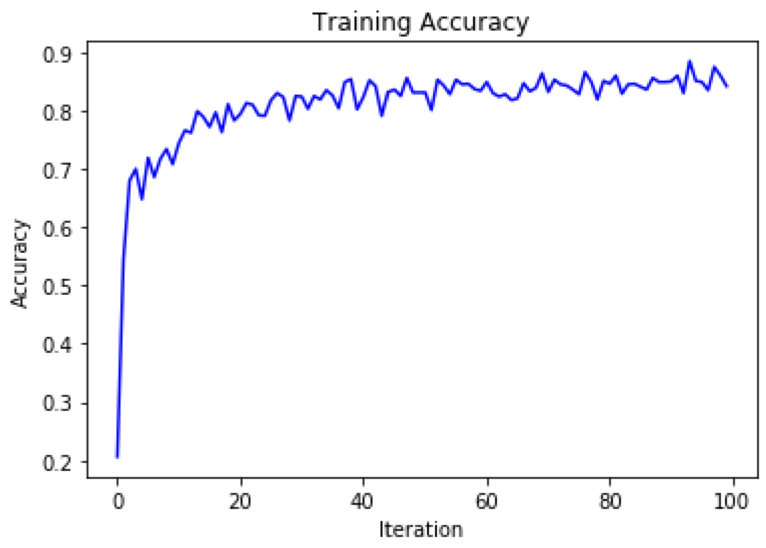


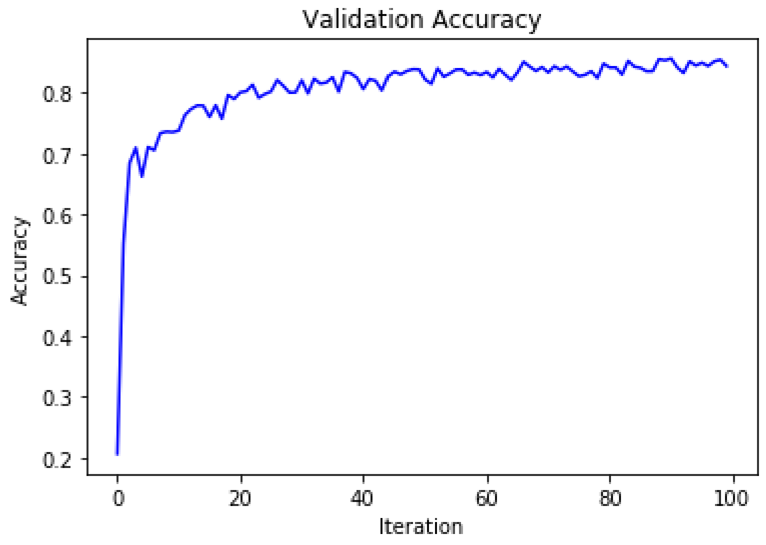
In [14]:
```
reset(model)
start = time.time()
runModel(model, 60)#10k steps
end = time.time()
print(end - start)
```

```
   0  0.21  0.21
 100  0.55  0.55
 200  0.68  0.68
 300  0.70  0.71
 400  0.65  0.66
 500  0.72  0.71
 600  0.69  0.71
 700  0.72  0.73
 800  0.73  0.74
 900  0.71  0.73
1000  0.74  0.74
1100  0.77  0.76
1200  0.76  0.77
1300  0.80  0.78
1400  0.79  0.78
1500  0.77  0.76
1600  0.80  0.78
```

```
1700  0.76  0.76
1800  0.81  0.80
1900  0.78  0.79
2000  0.79  0.80
2100  0.81  0.80
2200  0.81  0.81
2300  0.79  0.79
2400  0.79  0.80
2500  0.82  0.80
2600  0.83  0.82
2700  0.82  0.81
2800  0.78  0.80
2900  0.82  0.80
3000  0.82  0.82
3100  0.80  0.80
3200  0.82  0.82
3300  0.82  0.81
3400  0.83  0.82
3500  0.83  0.83
3600  0.80  0.80
3700  0.85  0.83
3800  0.85  0.83
3900  0.80  0.82
4000  0.82  0.81
4100  0.85  0.82
4200  0.84  0.82
4300  0.79  0.80
4400  0.83  0.83
4500  0.84  0.83
4600  0.82  0.83
4700  0.86  0.84
4800  0.83  0.84
4900  0.83  0.84
5000  0.83  0.82
5100  0.80  0.81
5200  0.85  0.84
5300  0.84  0.83
5400  0.83  0.83
5500  0.85  0.84
5600  0.84  0.84
5700  0.85  0.83
5800  0.84  0.83
5900  0.83  0.83
6000  0.85  0.83
6100  0.83  0.82
6200  0.82  0.84
6300  0.83  0.83
6400  0.82  0.82
6500  0.82  0.83
6600  0.85  0.85
6700  0.83  0.84
6800  0.84  0.84
6900  0.86  0.84
7000  0.83  0.83
7100  0.85  0.84
7200  0.84  0.84
7300  0.84  0.84
```

```
7400  0.84  0.83
7500  0.83  0.83
7600  0.87  0.83
7700  0.85  0.83
7800  0.82  0.82
7900  0.85  0.85
8000  0.85  0.84
8100  0.86  0.84
8200  0.83  0.83
8300  0.84  0.85
8400  0.85  0.84
8500  0.84  0.84
8600  0.84  0.83
8700  0.86  0.84
8800  0.85  0.85
8900  0.85  0.85
9000  0.85  0.86
9100  0.86  0.84
9200  0.83  0.83
9300  0.89  0.85
9400  0.85  0.84
9500  0.85  0.85
9600  0.83  0.84
9700  0.88  0.85
9800  0.86  0.85
9900  0.84  0.84
```





```
213.13990354537964
```

The best validation accuracy I obtained was 86. The configuration I used was 60 batch size, 10k optimization steps, 0.01 learning rate. It took a total of 212 seconds to run.

In [2]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import torch
from torch import autograd
import torch.nn.functional as F
import time

images = np.load("D:/work/JHUschoolStuff/machinelearning/project1/cs475_pr
oject_data/images.npy")
labels = np.load("D:/work/JHUschoolStuff/machinelearning/project1/cs475_pr
oject_data/labels.npy")
test = np.load("D:/work/JHUschoolStuff/machinelearning/project1/cs475_proj
ect_data/test_images.npy")
height = images.shape[1]
width = images.shape[2]
size = height * width
images = (images - images.mean()) / images.std()
data = images.reshape(images.shape[0],size)
data = torch.from_numpy(data).float()
labels = torch.from_numpy(labels).float()
test_data = test.reshape(test.shape[0], size)
test_data = (test_data - test_data.mean()) / test_data.std()
train_seqs = data[0:45000,:]
train_labels = labels[0:45000]
val_seqs = data[45000:,:]
val_labels = labels[45000:]
```

In [3]:
```
class TwoLayerNN(torch.nn.Module):
    def __init__(self, layer_1):
        super().__init__()
        self.layer_1 = torch.nn.Linear(height * width, layer_1)
        self.layer_2 = torch.nn.Linear(layer_1, 5)
        self.drop = torch.nn.Dropout(p = 0.3)
    def forward(self, x):
        x = self.layer_1(x)
        y = F.relu(x)
        y = self.drop(y)
        z = self.layer_2(y)
        return z
```

In [14]:
```
class ThreeLayerNN(torch.nn.Module):
    def __init__(self, layer_1, layer_2):
        super().__init__()
        self.layer_1 = torch.nn.Linear(height * width, layer_1)
        self.layer_2 = torch.nn.Linear(layer_1, layer_2)
        self.layer_3 = torch.nn.Linear(layer_2, 5)
    def forward(self, x):
        x = self.layer_1(x)
        y = F.relu(x)
        z = self.layer_2(y)
        a = F.relu(z)
        b = self.layer_3(a)
        return b
```

In [15]:
```
class FourLayerNN(torch.nn.Module):
    def __init__(self, layer_1, layer_2, layer_3):
        super().__init__()
        self.layer_1 = torch.nn.Linear(height * width, layer_1)
        self.layer_2 = torch.nn.Linear(layer_1, layer_2)
        self.layer_3 = torch.nn.Linear(layer_2, layer_3)
        self.layer_4 = torch.nn.Linear(layer_3, 5)
    def forward(self, x):
        x = self.layer_1(x)
        y = F.relu(x)
        z = self.layer_2(y)
        a = F.relu(z)
        b = self.layer_3(a)
        c = F.relu(b)
        d = self.layer_4(c)
        return d
```

In [4]:
```
def train(model, optimizer, batch_size):
    # model.train() puts our model in train mode, which can require differ
ent
    # behavior than eval mode (for example in the case of dropout).
    model.train()

    # i is is a 1-D array with shape [batch_size]
    i = np.random.choice(train_seqs.shape[0], size=batch_size, replace=Fal
se)
    i = torch.from_numpy(i).long()
    x = autograd.Variable(train_seqs[i, :])
    y = autograd.Variable(train_labels[i]).long()
    optimizer.zero_grad()
    y_hat_ = model(x)
    loss = F.multi_margin_loss(y_hat_, y) #using multi_margin_loss for las
t one
    #loss = F.cross_entropy(y_hat_, y)
    loss.backward()
    optimizer.step()
    return loss.data[0]
```

In [5]:
```
def approx_train_accuracy(model):
    model.eval()
    i = np.random.choice(train_seqs.shape[0], size=1000, replace=False)
    i = torch.from_numpy(i).long()
    x = autograd.Variable(train_seqs[i, :])
    y = autograd.Variable(train_labels[i]).long()
    y_hat_ = model(x)
    y_hat = np.zeros(1000)
    for i in range(1000):
        y_hat[i] = torch.max(y_hat_[i,:].data, 0)[1][0]
    return accuracy(y_hat, y.data.numpy())
```

In [6]:
```
def val_accuracy(model):
    model.eval()
    x = autograd.Variable(val_seqs)
    y = autograd.Variable(val_labels)
    y_hat_ = model(x)
    y_hat = np.zeros(5000)
    for i in range(5000):
        y_hat[i] = torch.max(y_hat_[i,:].data, 0)[1][0]
    return accuracy(y_hat, y.data.numpy())
```

In [7]:
```
def accuracy(y, y_hat):
    return (y == y_hat).astype(np.float).mean()
```

In [8]:
```
def plot(train_accs, val_accs):
    plt.figure(200)
    plt.title('Training Accuracy')
    plt.xlabel('Iteration')
    plt.ylabel('Accuracy')
    plt.plot(train_accs, 'b')
    plt.show()
    plt.figure(300)
    plt.title('Validation Accuracy')
    plt.xlabel('Iteration')
    plt.ylabel('Accuracy')
    plt.plot(val_accs, 'b')
    plt.show()
```

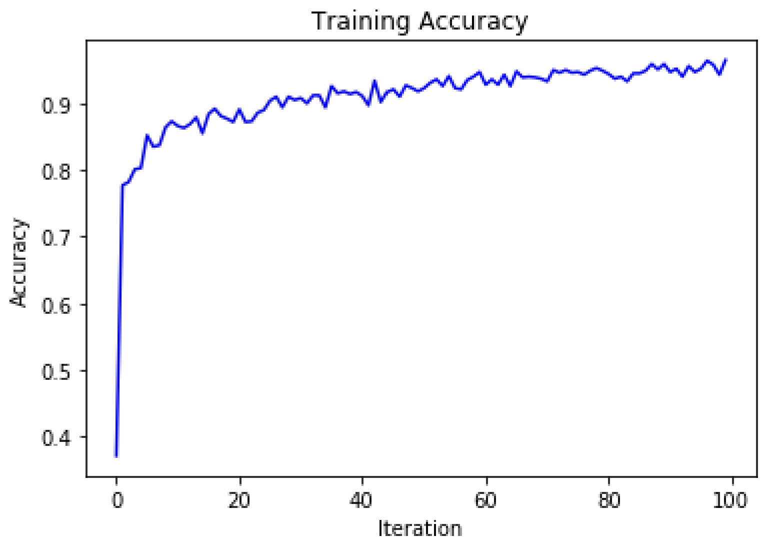
In [9]:
```
def runModel(model, batch_size, NUM_OPT_STEPS, optimizer):
    train_accs, val_accs = [], []
    for i in range(NUM_OPT_STEPS):
        train(model, optimizer, batch_size)
        if i % 100 == 0:
            train_accs.append(approx_train_accuracy(model))
            val_accs.append(val_accuracy(model))
            print("%6d %5.2f %5.2f" % (i, train_accs[-1], val_accs[-1]))
    plot(train_accs, val_accs)
```

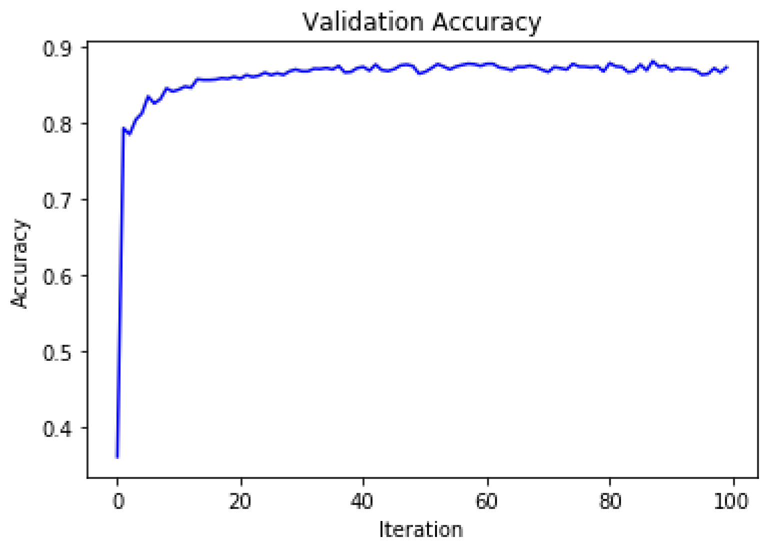
In [22]:
```
three_layer = ThreeLayerNN(200, 100)
optimizer_three_layer = torch.optim.Adam(three_layer.parameters(), lr=0.00
1)
runModel(three_layer, 32, 10000, optimizer_three_layer)
```

```
     0  0.37  0.36
   100  0.78  0.79
   200  0.78  0.79
   300  0.80  0.80
   400  0.80  0.81
   500  0.85  0.83
   600  0.83  0.83
   700  0.84  0.83
   800  0.86  0.85
   900  0.87  0.84
  1000  0.87  0.84
  1100  0.86  0.85
  1200  0.87  0.85
  1300  0.88  0.86
  1400  0.85  0.86
  1500  0.88  0.86
  1600  0.89  0.86
```

```
                    1700  0.88  0.86
                    1800  0.88  0.86
                    1900  0.87  0.86
                    2000  0.89  0.86
                    2100  0.87  0.86
                    2200  0.87  0.86
                    2300  0.89  0.86
                    2400  0.89  0.87
                    2500  0.90  0.86
                    2600  0.91  0.87
                    2700  0.89  0.86
                    2800  0.91  0.87
                    2900  0.91  0.87
                    3000  0.91  0.87
                    3100  0.90  0.87
                    3200  0.91  0.87
                    3300  0.91  0.87
                    3400  0.89  0.87
                    3500  0.93  0.87
                    3600  0.92  0.88
                    3700  0.92  0.87
                    3800  0.91  0.87
                    3900  0.92  0.87
                    4000  0.91  0.87
                    4100  0.90  0.87
                    4200  0.93  0.88
                    4300  0.90  0.87
                    4400  0.92  0.87
                    4500  0.92  0.87
                    4600  0.91  0.88
                    4700  0.93  0.88
                    4800  0.92  0.87
                    4900  0.92  0.86
                    5000  0.92  0.87
                    5100  0.93  0.87
                    5200  0.94  0.88
                    5300  0.93  0.87
                    5400  0.94  0.87
                    5500  0.92  0.87
                    5600  0.92  0.88
                    5700  0.94  0.88
                    5800  0.94  0.88
                    5900  0.95  0.88
                    6000  0.93  0.88
                    6100  0.94  0.88
                    6200  0.93  0.87
                    6300  0.94  0.87
                    6400  0.93  0.87
                    6500  0.95  0.87
                    6600  0.94  0.87
                    6700  0.94  0.88
                    6800  0.94  0.87
                    6900  0.94  0.87
                    7000  0.93  0.87
                    7100  0.95  0.87
                    7200  0.95  0.87
                    7300  0.95  0.87
```

```
7400  0.95  0.88
7500  0.95  0.87
7600  0.94  0.87
7700  0.95  0.87
7800  0.95  0.87
7900  0.95  0.87
8000  0.94  0.88
8100  0.94  0.87
8200  0.94  0.87
8300  0.93  0.87
8400  0.94  0.87
8500  0.94  0.88
8600  0.95  0.87
8700  0.96  0.88
8800  0.95  0.87
8900  0.96  0.88
9000  0.95  0.87
9100  0.95  0.87
9200  0.94  0.87
9300  0.96  0.87
9400  0.95  0.87
9500  0.95  0.86
9600  0.96  0.86
9700  0.96  0.87
9800  0.94  0.87
9900  0.96  0.87
```



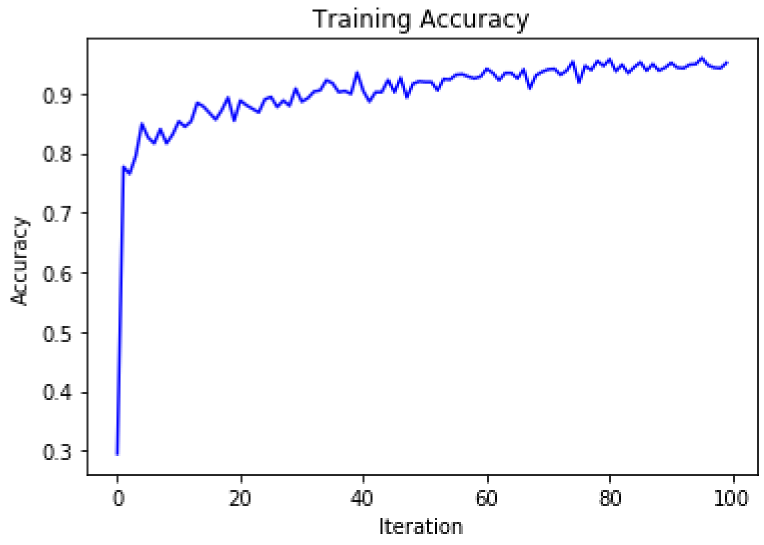


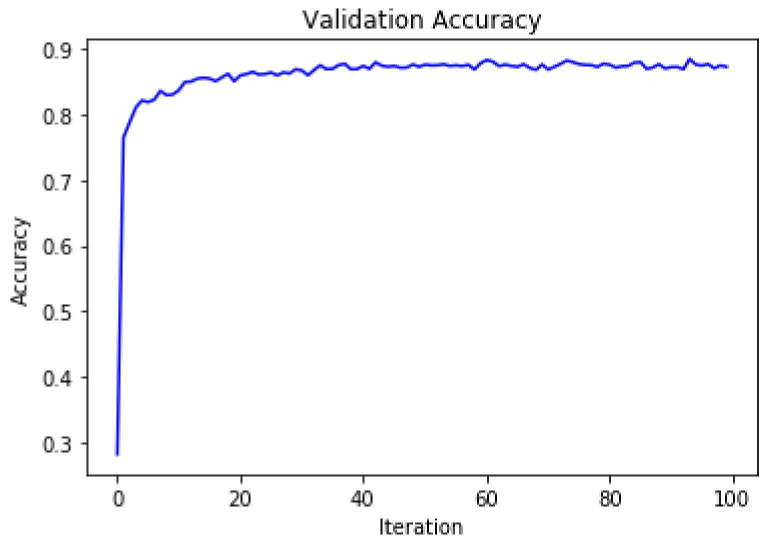
In [23]:
```
four_layer = FourLayerNN(200, 100, 50)
optimizer_four_layer = torch.optim.Adam(four_layer.parameters(), lr=0.001)
runModel(four_layer, 32, 10000, optimizer_four_layer)
```

```
   0  0.29  0.28
 100  0.78  0.76
 200  0.77  0.79
 300  0.80  0.81
 400  0.85  0.82
 500  0.83  0.82
 600  0.82  0.82
 700  0.84  0.84
 800  0.82  0.83
 900  0.83  0.83
1000  0.85  0.84
1100  0.84  0.85
1200  0.85  0.85
1300  0.88  0.85
1400  0.88  0.86
1500  0.87  0.85
1600  0.86  0.85
1700  0.87  0.86
1800  0.89  0.86
1900  0.85  0.85
2000  0.89  0.86
2100  0.88  0.86
2200  0.87  0.86
2300  0.87  0.86
2400  0.89  0.86
2500  0.89  0.86
2600  0.88  0.86
2700  0.89  0.86
2800  0.88  0.86
2900  0.91  0.87
3000  0.89  0.87
3100  0.89  0.86
3200  0.90  0.87
3300  0.91  0.87
3400  0.92  0.87
```

```
3500  0.92  0.87
3600  0.90  0.87
3700  0.90  0.88
3800  0.90  0.87
3900  0.94  0.87
4000  0.90  0.87
4100  0.89  0.87
4200  0.90  0.88
4300  0.90  0.87
4400  0.92  0.87
4500  0.90  0.87
4600  0.93  0.87
4700  0.89  0.87
4800  0.92  0.88
4900  0.92  0.87
5000  0.92  0.88
5100  0.92  0.87
5200  0.91  0.87
5300  0.92  0.88
5400  0.92  0.87
5500  0.93  0.87
5600  0.93  0.87
5700  0.93  0.88
5800  0.93  0.87
5900  0.93  0.88
6000  0.94  0.88
6100  0.93  0.88
6200  0.92  0.87
6300  0.93  0.88
6400  0.93  0.87
6500  0.93  0.87
6600  0.94  0.88
6700  0.91  0.87
6800  0.93  0.87
6900  0.94  0.88
7000  0.94  0.87
7100  0.94  0.87
7200  0.93  0.88
7300  0.94  0.88
7400  0.95  0.88
7500  0.92  0.88
7600  0.95  0.88
7700  0.94  0.87
7800  0.95  0.87
7900  0.95  0.88
8000  0.96  0.88
8100  0.94  0.87
8200  0.95  0.87
8300  0.93  0.87
8400  0.94  0.88
8500  0.95  0.88
8600  0.94  0.87
8700  0.95  0.87
8800  0.94  0.88
8900  0.94  0.87
9000  0.95  0.87
9100  0.94  0.87
```

```
9200  0.94  0.87
9300  0.95  0.88
9400  0.95  0.88
9500  0.96  0.87
9600  0.95  0.88
9700  0.94  0.87
9800  0.94  0.87
9900  0.95  0.87
```



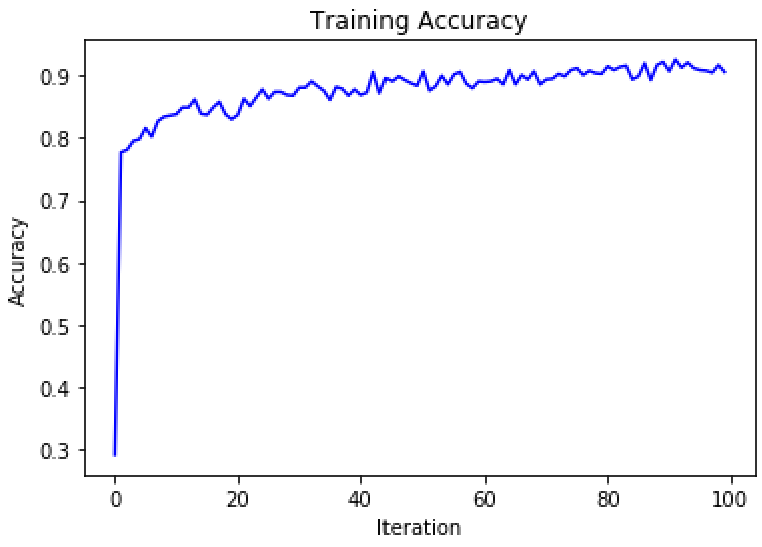


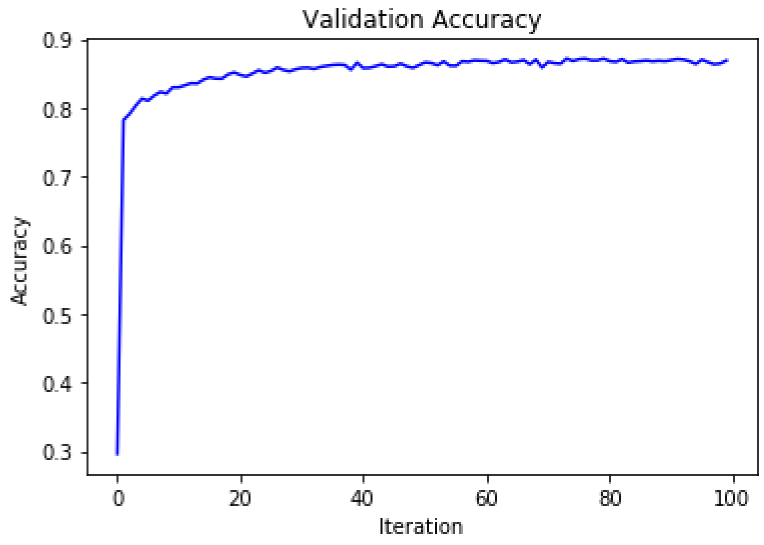
In [24]:
```
two_layer = TwoLayerNN(100) #using dropout at 0.3
optimizer_two_layer = torch.optim.Adam(two_layer.parameters(), lr=0.001)
runModel(two_layer, 32, 10000, optimizer_two_layer)
```

```
   0  0.29  0.30
 100  0.78  0.78
 200  0.78  0.79
 300  0.80  0.80
 400  0.80  0.81
 500  0.82  0.81
 600  0.80  0.82
 700  0.83  0.82
 800  0.83  0.82
 900  0.84  0.83
```

```
1000  0.84  0.83
1100  0.85  0.83
1200  0.85  0.84
1300  0.86  0.84
1400  0.84  0.84
1500  0.84  0.84
1600  0.85  0.84
1700  0.86  0.84
1800  0.84  0.85
1900  0.83  0.85
2000  0.84  0.85
2100  0.86  0.85
2200  0.85  0.85
2300  0.86  0.85
2400  0.88  0.85
2500  0.86  0.85
2600  0.87  0.86
2700  0.87  0.86
2800  0.87  0.85
2900  0.87  0.86
3000  0.88  0.86
3100  0.88  0.86
3200  0.89  0.86
3300  0.88  0.86
3400  0.88  0.86
3500  0.86  0.86
3600  0.88  0.86
3700  0.88  0.86
3800  0.87  0.86
3900  0.88  0.87
4000  0.87  0.86
4100  0.87  0.86
4200  0.91  0.86
4300  0.87  0.86
4400  0.90  0.86
4500  0.89  0.86
4600  0.90  0.86
4700  0.89  0.86
4800  0.89  0.86
4900  0.88  0.86
5000  0.91  0.87
5100  0.88  0.87
5200  0.88  0.86
5300  0.90  0.87
5400  0.89  0.86
5500  0.90  0.86
5600  0.91  0.87
5700  0.89  0.87
5800  0.88  0.87
5900  0.89  0.87
6000  0.89  0.87
6100  0.89  0.86
6200  0.90  0.87
6300  0.89  0.87
6400  0.91  0.87
6500  0.89  0.87
6600  0.90  0.87
```

```
6700  0.89  0.86
6800  0.91  0.87
6900  0.89  0.86
7000  0.89  0.87
7100  0.90  0.86
7200  0.90  0.86
7300  0.90  0.87
7400  0.91  0.87
7500  0.91  0.87
7600  0.90  0.87
7700  0.91  0.87
7800  0.90  0.87
7900  0.90  0.87
8000  0.92  0.87
8100  0.91  0.87
8200  0.91  0.87
8300  0.92  0.87
8400  0.89  0.87
8500  0.90  0.87
8600  0.92  0.87
8700  0.89  0.87
8800  0.92  0.87
8900  0.92  0.87
9000  0.91  0.87
9100  0.93  0.87
9200  0.91  0.87
9300  0.92  0.87
9400  0.91  0.86
9500  0.91  0.87
9600  0.91  0.87
9700  0.91  0.86
9800  0.92  0.86
9900  0.91  0.87
```



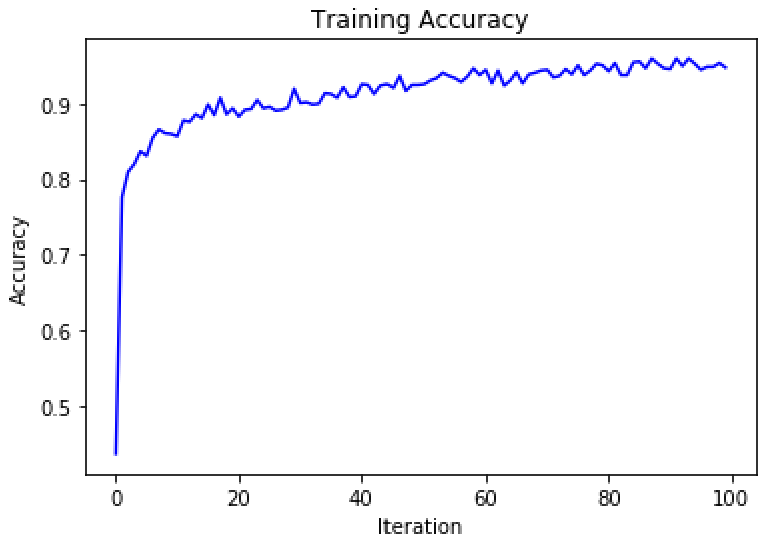


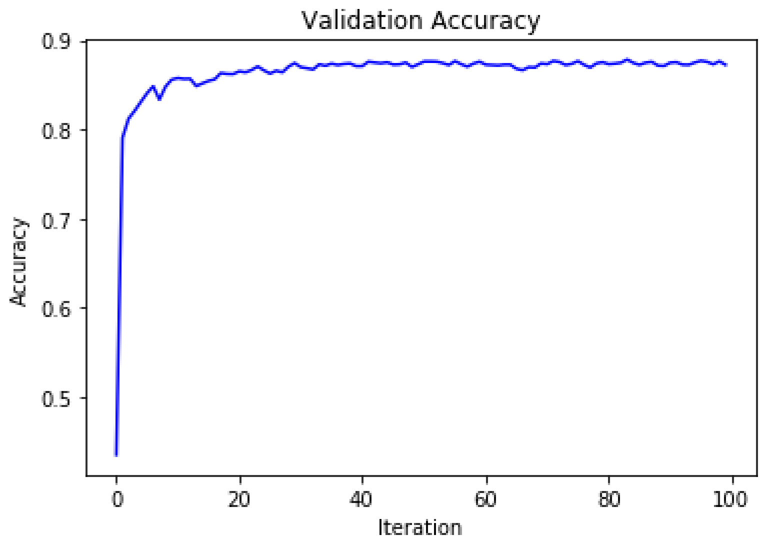
In [10]:
```
start = time.time()
two_layer = TwoLayerNN(200) #using dropout at 0.3
optimizer_two_layer = torch.optim.Adam(two_layer.parameters(), lr=0.001)
runModel(two_layer, 65, 10000, optimizer_two_layer)#using multi hinge loss
end = time.time()
print(end - start)
```

```
    0  0.44  0.43
  100  0.78  0.79
  200  0.81  0.81
  300  0.82  0.82
  400  0.84  0.83
  500  0.83  0.84
  600  0.85  0.85
  700  0.87  0.83
  800  0.86  0.85
  900  0.86  0.86
 1000  0.86  0.86
 1100  0.88  0.86
 1200  0.88  0.86
 1300  0.89  0.85
 1400  0.88  0.85
 1500  0.90  0.85
 1600  0.89  0.86
 1700  0.91  0.86
 1800  0.89  0.86
 1900  0.89  0.86
 2000  0.88  0.87
 2100  0.89  0.86
 2200  0.89  0.87
 2300  0.91  0.87
 2400  0.89  0.87
 2500  0.90  0.86
 2600  0.89  0.87
 2700  0.89  0.86
 2800  0.90  0.87
 2900  0.92  0.87
 3000  0.90  0.87
 3100  0.90  0.87
```

```
                    3200  0.90  0.87
                    3300  0.90  0.87
                    3400  0.91  0.87
                    3500  0.91  0.87
                    3600  0.91  0.87
                    3700  0.92  0.87
                    3800  0.91  0.87
                    3900  0.91  0.87
                    4000  0.93  0.87
                    4100  0.93  0.88
                    4200  0.91  0.88
                    4300  0.92  0.87
                    4400  0.93  0.88
                    4500  0.92  0.87
                    4600  0.94  0.87
                    4700  0.92  0.88
                    4800  0.93  0.87
                    4900  0.93  0.87
                    5000  0.93  0.88
                    5100  0.93  0.88
                    5200  0.93  0.88
                    5300  0.94  0.87
                    5400  0.94  0.87
                    5500  0.93  0.88
                    5600  0.93  0.87
                    5700  0.94  0.87
                    5800  0.95  0.87
                    5900  0.94  0.88
                    6000  0.94  0.87
                    6100  0.93  0.87
                    6200  0.94  0.87
                    6300  0.92  0.87
                    6400  0.93  0.87
                    6500  0.94  0.87
                    6600  0.93  0.87
                    6700  0.94  0.87
                    6800  0.94  0.87
                    6900  0.94  0.87
                    7000  0.94  0.87
                    7100  0.94  0.88
                    7200  0.94  0.88
                    7300  0.95  0.87
                    7400  0.94  0.87
                    7500  0.95  0.88
                    7600  0.94  0.87
                    7700  0.94  0.87
                    7800  0.95  0.87
                    7900  0.95  0.88
                    8000  0.94  0.87
                    8100  0.95  0.87
                    8200  0.94  0.87
                    8300  0.94  0.88
                    8400  0.95  0.87
                    8500  0.96  0.87
                    8600  0.95  0.87
                    8700  0.96  0.88
                    8800  0.95  0.87
```

```
8900  0.95  0.87
9000  0.95  0.87
9100  0.96  0.88
9200  0.95  0.87
9300  0.96  0.87
9400  0.95  0.88
9500  0.94  0.88
9600  0.95  0.88
9700  0.95  0.87
9800  0.95  0.88
9900  0.95  0.87
```





```
80.23156976699829
```

Best validation accuracy achieved was 88. This was using my two layer neural net with 200 units, learning rate of 0.001 using Adam optimizer, 10k optimization steps, and 65 batch size. This was also run with a dropout before the second layer with probability 0.3 and using multi hinge loss instead of cross entropy loss. The total time was about 76 seconds.

In [1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import torch
from torch import autograd
import torch.nn.functional as F
import csv
import time

images = np.load("D:/work/JHUschoolStuff/machinelearning/project1/cs475_pr
oject_data/images.npy")
labels = np.load("D:/work/JHUschoolStuff/machinelearning/project1/cs475_pr
oject_data/labels.npy")
test = np.load("D:/work/JHUschoolStuff/machinelearning/project1/cs475_proj
ect_data/part_2_test_images.npy")
height = images.shape[1]
width = images.shape[2]
size = height * width
images = (images - images.mean()) / images.std()
data = images.reshape(images.shape[0],size)
data = torch.from_numpy(data).float()
labels = torch.from_numpy(labels).float()
test_data = test.reshape(test.shape[0], size)
test_data = (test_data - test_data.mean()) / test_data.std()
test_data = torch.from_numpy(test_data).float()
batch_size = 1
NUM_OPT_STEPS = 5000
train_seqs = data[0:45000,:]
train_labels = labels[0:45000]
val_seqs = data[45000:,:]
val_labels = labels[45000:]
NUM_CLASSES = 5
```

In [2]:
```
class TooSimpleConvNN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        # 3x3 convolution that takes in an image with one channel
        # and outputs an image with 8 channels.
        self.conv1 = torch.nn.Conv2d(1, 16, kernel_size=3, stride = 2)
        # 3x3 convolution that takes in an image with 8 channels
        # and outputs an image with 16 channels. The output image
        # has approximately half the height and half the width
        # because of the stride of 2.
        self.conv2 = torch.nn.Conv2d(16, 32, kernel_size=3, stride = 2)
        # 1x1 convolution that takes in an image with 16 channels and
        # produces an image with 5 channels. Here, the 5 channels
        # will correspond to class scores.
        self.final_conv = torch.nn.Conv2d(32, 5, kernel_size=1)
    def forward(self, x):
        # Convolutions work with images of shape
        # [batch_size, num_channels, height, width]
        x = x.view(-1, height, width).unsqueeze(1)
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        n, c, h, w = x.size()
        x = F.avg_pool2d(x, kernel_size=[h, w])
        x = self.final_conv(x).view(-1, NUM_CLASSES)
        return x
```

In [3]:
```
def train(model, optimizer, batch_size):
    model.train()
    # i is is a 1-D array with shape [batch_size]
    i = np.random.choice(train_seqs.shape[0], size=batch_size, replace=Fal
se)
    i = torch.from_numpy(i).long()
    x = autograd.Variable(train_seqs[i, :])
    y = autograd.Variable(train_labels[i]).long()
    optimizer.zero_grad()
    y_hat_ = model(x)
    loss = F.cross_entropy(y_hat_, y)
    loss.backward()
    optimizer.step()
    return loss.data[0]
```

In [4]:
```
def approx_train_accuracy(model):
    i = np.random.choice(train_seqs.shape[0], size=1000, replace=False)
    i = torch.from_numpy(i).long()
    x = autograd.Variable(train_seqs[i, :])
    y = autograd.Variable(train_labels[i]).long()
    y_hat_ = model(x)
    y_hat = np.zeros(1000)
    for i in range(1000):
        y_hat[i] = torch.max(y_hat_[i,:].data, 0)[1][0]
    return accuracy(y_hat, y.data.numpy())
```

In [5]:
```
def val_accuracy(model):
    x = autograd.Variable(val_seqs)
    y = autograd.Variable(val_labels)
    y_hat_ = model(x)
    y_hat = np.zeros(5000)
    for i in range(5000):
        y_hat[i] = torch.max(y_hat_[i,:].data, 0)[1][0]
    return accuracy(y_hat, y.data.numpy())
```

In [6]:
```
def accuracy(y, y_hat):
    return (y == y_hat).astype(np.float).mean()
```

In [7]:
```
def plot(train_accs, val_accs):
    plt.figure(200)
    plt.title('Training Accuracy')
    plt.xlabel('Iteration')
    plt.ylabel('Accuracy')
    plt.plot(train_accs, 'b')
    plt.show()
    plt.figure(300)
    plt.title('Validation Accuracy')
    plt.xlabel('Iteration')
    plt.ylabel('Accuracy')
    plt.plot(val_accs, 'b')
    plt.show()
```

In [8]:
```
def runModel(model, batch_size, NUM_OPT_STEPS, optimizer):
    train_accs, val_accs = [], []
    for i in range(NUM_OPT_STEPS):
        train(model, optimizer, batch_size)
        if i % 100 == 0:
            train_accs.append(approx_train_accuracy(model))
            val_accs.append(val_accuracy(model))
            print("%6d %5.2f %5.2f" % (i, train_accs[-1], val_accs[-1]))
    plot(train_accs, val_accs)
```

In [20]:
```
model = TooSimpleConvNN()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

In [21]:
```
runModel(model, 32, 30000, optimizer)
```

```
     0  0.21  0.19
   100  0.52  0.50
   200  0.59  0.61
   300  0.65  0.65
   400  0.67  0.69
   500  0.66  0.70
   600  0.70  0.71
   700  0.70  0.70
   800  0.71  0.72
   900  0.71  0.74
  1000  0.73  0.74
  1100  0.70  0.73
  1200  0.74  0.76
  1300  0.73  0.75
  1400  0.75  0.75
  1500  0.73  0.75
  1600  0.77  0.78
  1700  0.75  0.77
  1800  0.75  0.77
  1900  0.76  0.76
  2000  0.77  0.77
  2100  0.75  0.78
  2200  0.78  0.77
  2300  0.77  0.77
  2400  0.75  0.78
  2500  0.77  0.79
  2600  0.77  0.78
  2700  0.78  0.78
  2800  0.79  0.78
  2900  0.77  0.79
  3000  0.77  0.80
  3100  0.79  0.79
  3200  0.77  0.79
  3300  0.80  0.80
  3400  0.81  0.80
  3500  0.79  0.81
  3600  0.80  0.81
  3700  0.82  0.80
```

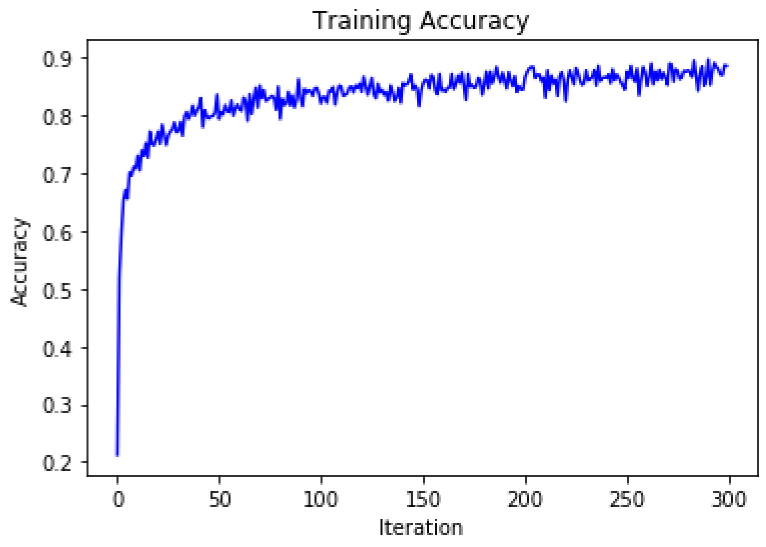
```
3800  0.80  0.81
3900  0.81  0.81
4000  0.81  0.80
4100  0.83  0.81
4200  0.78  0.81
4300  0.81  0.80
4400  0.80  0.80
4500  0.80  0.81
4600  0.80  0.79
4700  0.80  0.81
4800  0.81  0.81
4900  0.84  0.81
5000  0.79  0.82
5100  0.81  0.82
5200  0.80  0.81
5300  0.82  0.81
5400  0.81  0.82
5500  0.81  0.82
5600  0.83  0.82
5700  0.80  0.80
5800  0.81  0.81
5900  0.82  0.82
6000  0.81  0.81
6100  0.81  0.83
6200  0.83  0.83
6300  0.82  0.84
6400  0.79  0.81
6500  0.84  0.82
6600  0.80  0.83
6700  0.83  0.82
6800  0.85  0.83
6900  0.81  0.82
7000  0.85  0.84
7100  0.83  0.83
7200  0.84  0.84
7300  0.82  0.84
7400  0.83  0.83
7500  0.83  0.84
7600  0.83  0.83
7700  0.83  0.83
7800  0.81  0.83
7900  0.85  0.84
8000  0.79  0.82
8100  0.83  0.83
8200  0.82  0.84
8300  0.82  0.84
8400  0.84  0.84
8500  0.82  0.84
8600  0.83  0.83
8700  0.81  0.83
8800  0.83  0.84
8900  0.86  0.85
9000  0.83  0.83
9100  0.82  0.82
9200  0.85  0.84
9300  0.84  0.85
9400  0.84  0.83
```

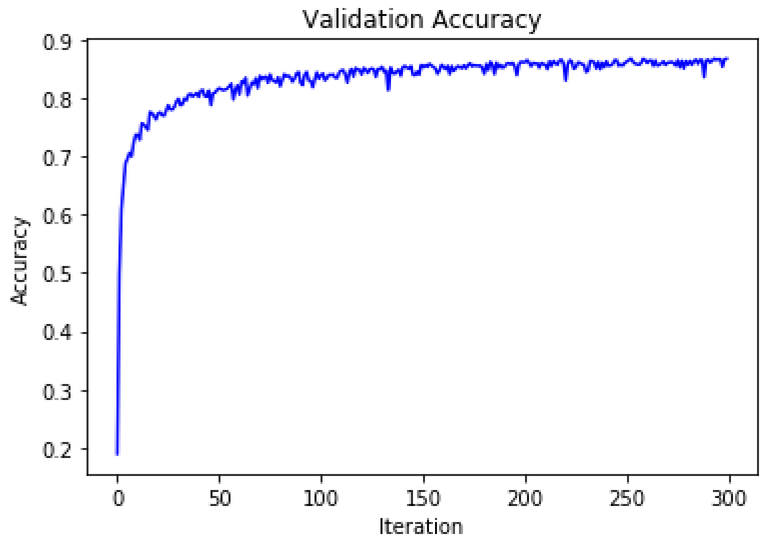
```
 9500  0.84  0.83
 9600  0.83  0.82
 9700  0.85  0.83
 9800  0.85  0.84
 9900  0.84  0.83
10000  0.82  0.84
10100  0.83  0.84
10200  0.83  0.83
10300  0.82  0.83
10400  0.84  0.84
10500  0.84  0.84
10600  0.85  0.84
10700  0.82  0.84
10800  0.84  0.83
10900  0.85  0.84
11000  0.85  0.85
11100  0.83  0.85
11200  0.84  0.84
11300  0.84  0.83
11400  0.85  0.85
11500  0.85  0.85
11600  0.84  0.84
11700  0.85  0.85
11800  0.85  0.85
11900  0.85  0.85
12000  0.84  0.84
12100  0.87  0.85
12200  0.85  0.85
12300  0.84  0.84
12400  0.85  0.85
12500  0.86  0.85
12600  0.84  0.85
12700  0.83  0.84
12800  0.85  0.85
12900  0.84  0.85
13000  0.84  0.85
13100  0.84  0.85
13200  0.84  0.85
13300  0.83  0.81
13400  0.85  0.85
13500  0.84  0.84
13600  0.82  0.84
13700  0.83  0.85
13800  0.85  0.85
13900  0.82  0.84
14000  0.86  0.85
14100  0.85  0.86
14200  0.85  0.85
14300  0.86  0.85
14400  0.87  0.86
14500  0.84  0.84
14600  0.85  0.84
14700  0.85  0.85
14800  0.82  0.84
14900  0.84  0.86
15000  0.86  0.85
15100  0.86  0.86
```

```
15200  0.86  0.85
15300  0.85  0.86
15400  0.87  0.86
15500  0.87  0.85
15600  0.84  0.85
15700  0.84  0.84
15800  0.87  0.85
15900  0.84  0.86
16000  0.84  0.85
16100  0.84  0.85
16200  0.85  0.86
16300  0.85  0.84
16400  0.87  0.86
16500  0.85  0.85
16600  0.85  0.85
16700  0.87  0.86
16800  0.86  0.85
16900  0.87  0.85
17000  0.85  0.86
17100  0.83  0.85
17200  0.87  0.86
17300  0.85  0.86
17400  0.85  0.85
17500  0.85  0.86
17600  0.85  0.86
17700  0.86  0.86
17800  0.86  0.86
17900  0.84  0.85
18000  0.85  0.84
18100  0.88  0.86
18200  0.85  0.85
18300  0.88  0.86
18400  0.86  0.86
18500  0.87  0.84
18600  0.88  0.86
18700  0.87  0.85
18800  0.86  0.86
18900  0.87  0.85
19000  0.86  0.85
19100  0.85  0.86
19200  0.88  0.86
19300  0.86  0.86
19400  0.85  0.86
19500  0.87  0.85
19600  0.84  0.84
19700  0.85  0.86
19800  0.84  0.86
19900  0.84  0.86
20000  0.87  0.86
20100  0.88  0.87
20200  0.88  0.86
20300  0.88  0.85
20400  0.88  0.86
20500  0.86  0.86
20600  0.87  0.86
20700  0.87  0.85
20800  0.86  0.86
```

```
20900  0.87  0.86
21000  0.83  0.86
21100  0.88  0.85
21200  0.84  0.86
21300  0.87  0.86
21400  0.86  0.86
21500  0.86  0.86
21600  0.83  0.86
21700  0.87  0.87
21800  0.88  0.87
21900  0.86  0.86
22000  0.82  0.83
22100  0.86  0.86
22200  0.88  0.87
22300  0.87  0.86
22400  0.85  0.85
22500  0.88  0.85
22600  0.87  0.86
22700  0.86  0.86
22800  0.85  0.86
22900  0.86  0.86
23000  0.88  0.85
23100  0.86  0.85
23200  0.86  0.86
23300  0.87  0.86
23400  0.88  0.86
23500  0.85  0.85
23600  0.89  0.86
23700  0.86  0.85
23800  0.86  0.86
23900  0.87  0.85
24000  0.87  0.86
24100  0.87  0.86
24200  0.86  0.86
24300  0.88  0.86
24400  0.86  0.87
24500  0.87  0.87
24600  0.87  0.86
24700  0.86  0.86
24800  0.85  0.86
24900  0.84  0.86
25000  0.88  0.86
25100  0.86  0.87
25200  0.89  0.87
25300  0.86  0.86
25400  0.86  0.86
25500  0.88  0.86
25600  0.83  0.86
25700  0.86  0.86
25800  0.88  0.87
25900  0.87  0.87
26000  0.85  0.86
26100  0.87  0.86
26200  0.89  0.86
26300  0.85  0.86
26400  0.88  0.86
26500  0.86  0.86
```

```
26600  0.88  0.86
26700  0.86  0.86
26800  0.87  0.86
26900  0.86  0.86
27000  0.85  0.86
27100  0.89  0.86
27200  0.89  0.86
27300  0.86  0.86
27400  0.88  0.86
27500  0.88  0.87
27600  0.86  0.85
27700  0.87  0.86
27800  0.88  0.85
27900  0.88  0.86
28000  0.88  0.86
28100  0.88  0.86
28200  0.87  0.86
28300  0.90  0.86
28400  0.87  0.87
28500  0.84  0.86
28600  0.87  0.87
28700  0.89  0.87
28800  0.85  0.84
28900  0.86  0.86
29000  0.90  0.87
29100  0.85  0.86
29200  0.88  0.86
29300  0.89  0.87
29400  0.88  0.87
29500  0.88  0.87
29600  0.87  0.87
29700  0.87  0.85
29800  0.89  0.87
29900  0.89  0.87
```





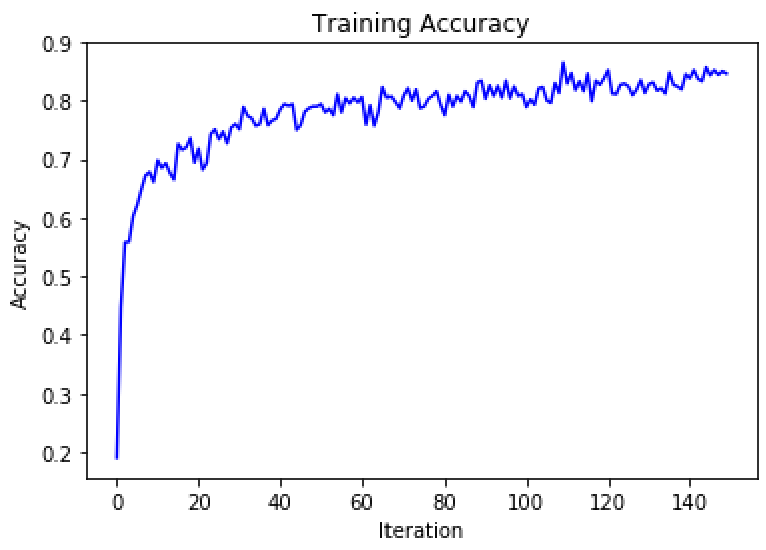
In [9]:
```
model = TooSimpleConvNN()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
runModel(model, 32, 15000, optimizer) #using stride = 1
```

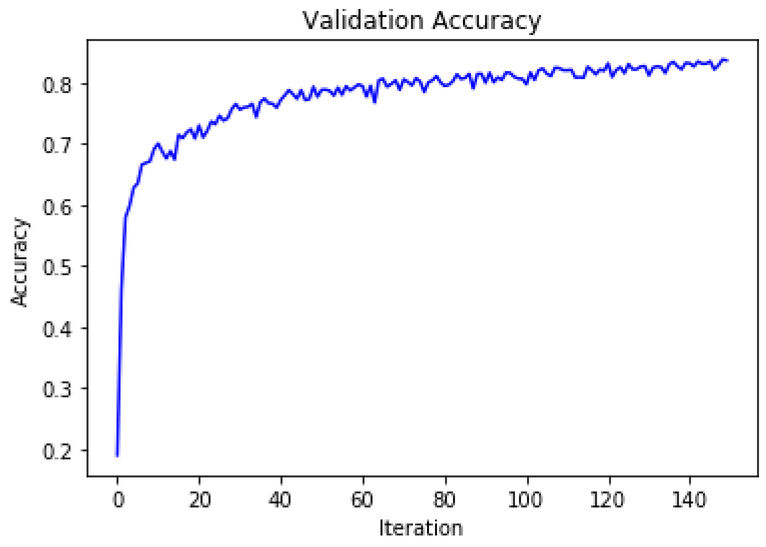
```
    0  0.19  0.19
  100  0.45  0.46
  200  0.56  0.58
  300  0.56  0.60
  400  0.60  0.63
  500  0.62  0.64
  600  0.65  0.67
  700  0.67  0.67
  800  0.68  0.67
  900  0.66  0.69
 1000  0.70  0.70
 1100  0.69  0.69
 1200  0.69  0.68
 1300  0.68  0.69
 1400  0.67  0.67
 1500  0.73  0.71
 1600  0.72  0.71
 1700  0.72  0.72
 1800  0.74  0.72
 1900  0.69  0.71
 2000  0.72  0.73
 2100  0.68  0.71
 2200  0.69  0.72
 2300  0.74  0.74
 2400  0.75  0.73
 2500  0.73  0.75
 2600  0.75  0.74
 2700  0.73  0.74
 2800  0.75  0.76
 2900  0.76  0.77
 3000  0.75  0.76
 3100  0.79  0.76
 3200  0.77  0.76
 3300  0.77  0.77
 3400  0.76  0.74
```

```
3500  0.76  0.77
3600  0.79  0.77
3700  0.76  0.77
3800  0.77  0.77
3900  0.77  0.76
4000  0.79  0.77
4100  0.79  0.78
4200  0.79  0.79
4300  0.79  0.78
4400  0.75  0.77
4500  0.76  0.79
4600  0.78  0.77
4700  0.79  0.77
4800  0.79  0.79
4900  0.79  0.78
5000  0.79  0.79
5100  0.78  0.79
5200  0.79  0.79
5300  0.78  0.78
5400  0.81  0.79
5500  0.78  0.78
5600  0.80  0.79
5700  0.80  0.79
5800  0.81  0.79
5900  0.80  0.80
6000  0.81  0.79
6100  0.76  0.78
6200  0.79  0.79
6300  0.76  0.77
6400  0.78  0.80
6500  0.82  0.81
6600  0.81  0.79
6700  0.81  0.80
6800  0.80  0.80
6900  0.79  0.79
7000  0.81  0.81
7100  0.82  0.80
7200  0.80  0.80
7300  0.82  0.81
7400  0.79  0.80
7500  0.79  0.78
7600  0.80  0.80
7700  0.81  0.80
7800  0.82  0.81
7900  0.79  0.80
8000  0.78  0.79
8100  0.81  0.80
8200  0.79  0.80
8300  0.81  0.81
8400  0.80  0.81
8500  0.82  0.81
8600  0.81  0.81
8700  0.79  0.79
8800  0.83  0.81
8900  0.83  0.81
9000  0.80  0.80
9100  0.83  0.82
```

```
 9200  0.81  0.80
 9300  0.82  0.81
 9400  0.81  0.80
 9500  0.83  0.82
 9600  0.80  0.82
 9700  0.82  0.81
 9800  0.81  0.81
 9900  0.81  0.81
10000  0.79  0.80
10100  0.80  0.82
10200  0.79  0.80
10300  0.82  0.82
10400  0.82  0.82
10500  0.80  0.81
10600  0.80  0.81
10700  0.83  0.82
10800  0.81  0.82
10900  0.86  0.82
11000  0.83  0.82
11100  0.85  0.82
11200  0.82  0.81
11300  0.83  0.81
11400  0.82  0.81
11500  0.85  0.83
11600  0.80  0.82
11700  0.83  0.81
11800  0.83  0.82
11900  0.84  0.82
12000  0.85  0.83
12100  0.81  0.81
12200  0.81  0.82
12300  0.83  0.83
12400  0.83  0.82
12500  0.82  0.83
12600  0.81  0.82
12700  0.82  0.82
12800  0.83  0.83
12900  0.81  0.83
13000  0.83  0.81
13100  0.83  0.82
13200  0.82  0.83
13300  0.82  0.83
13400  0.81  0.82
13500  0.85  0.83
13600  0.83  0.83
13700  0.82  0.83
13800  0.82  0.82
13900  0.84  0.83
14000  0.84  0.83
14100  0.85  0.83
14200  0.84  0.83
14300  0.83  0.83
14400  0.86  0.83
14500  0.84  0.83
14600  0.85  0.82
14700  0.84  0.83
14800  0.85  0.84
```

```
14900  0.85  0.84
```





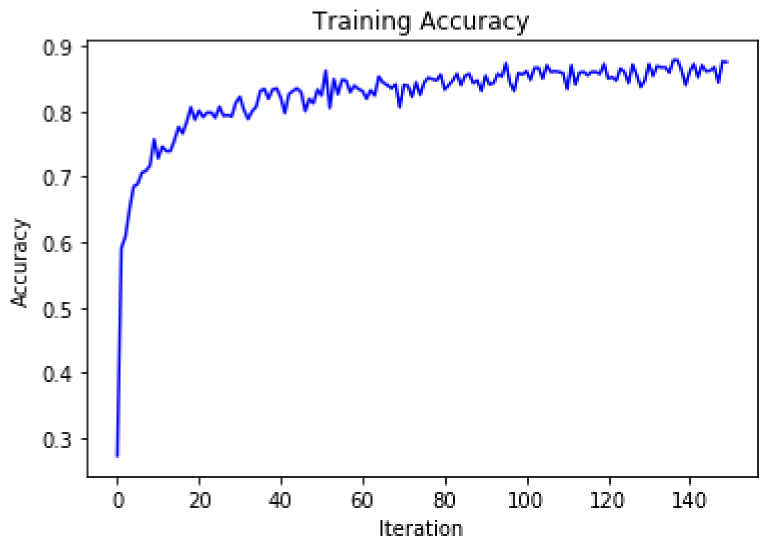
In [23]:
```
model = TooSimpleConvNN()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
runModel(model, 64, 15000, optimizer) #using stride = 2
```

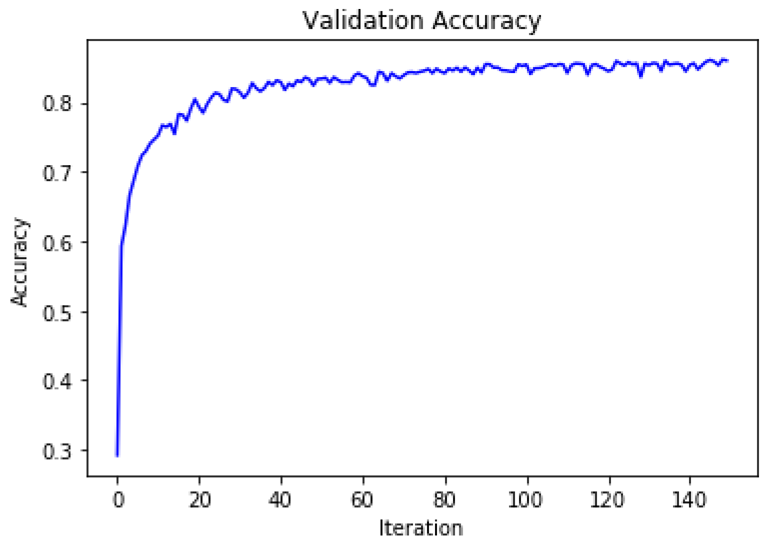
```
   0  0.27  0.29
 100  0.59  0.59
 200  0.61  0.62
 300  0.65  0.67
 400  0.69  0.69
 500  0.69  0.71
 600  0.71  0.72
 700  0.71  0.73
 800  0.72  0.74
 900  0.76  0.75
1000  0.73  0.75
1100  0.75  0.77
1200  0.74  0.76
1300  0.74  0.77
1400  0.76  0.75
1500  0.78  0.78
1600  0.77  0.78
```

```
1700  0.78  0.77
1800  0.81  0.79
1900  0.79  0.80
2000  0.80  0.79
2100  0.79  0.78
2200  0.80  0.80
2300  0.80  0.81
2400  0.79  0.81
2500  0.81  0.81
2600  0.79  0.80
2700  0.79  0.80
2800  0.79  0.82
2900  0.81  0.82
3000  0.82  0.81
3100  0.80  0.81
3200  0.79  0.81
3300  0.80  0.83
3400  0.81  0.82
3500  0.83  0.82
3600  0.83  0.82
3700  0.82  0.83
3800  0.83  0.82
3900  0.83  0.83
4000  0.82  0.83
4100  0.80  0.82
4200  0.83  0.83
4300  0.83  0.82
4400  0.83  0.83
4500  0.83  0.83
4600  0.80  0.84
4700  0.82  0.83
4800  0.81  0.82
4900  0.83  0.83
5000  0.82  0.83
5100  0.86  0.84
5200  0.81  0.83
5300  0.85  0.84
5400  0.83  0.83
5500  0.85  0.83
5600  0.85  0.83
5700  0.83  0.83
5800  0.84  0.84
5900  0.83  0.84
6000  0.83  0.84
6100  0.82  0.84
6200  0.83  0.83
6300  0.82  0.82
6400  0.85  0.84
6500  0.84  0.84
6600  0.84  0.83
6700  0.83  0.84
6800  0.84  0.84
6900  0.81  0.83
7000  0.84  0.84
7100  0.84  0.84
7200  0.82  0.84
7300  0.84  0.84
```

```
 7400  0.82  0.84
 7500  0.84  0.85
 7600  0.85  0.85
 7700  0.85  0.84
 7800  0.85  0.85
 7900  0.86  0.84
 8000  0.83  0.84
 8100  0.84  0.85
 8200  0.85  0.85
 8300  0.86  0.85
 8400  0.84  0.84
 8500  0.85  0.85
 8600  0.86  0.85
 8700  0.84  0.84
 8800  0.85  0.85
 8900  0.83  0.84
 9000  0.85  0.85
 9100  0.84  0.85
 9200  0.84  0.85
 9300  0.86  0.85
 9400  0.85  0.85
 9500  0.87  0.85
 9600  0.84  0.84
 9700  0.83  0.84
 9800  0.86  0.85
 9900  0.86  0.85
10000  0.86  0.85
10100  0.85  0.84
10200  0.87  0.85
10300  0.87  0.85
10400  0.85  0.85
10500  0.87  0.85
10600  0.86  0.85
10700  0.86  0.85
10800  0.86  0.85
10900  0.86  0.85
11000  0.83  0.84
11100  0.87  0.85
11200  0.84  0.86
11300  0.86  0.86
11400  0.86  0.85
11500  0.85  0.84
11600  0.86  0.85
11700  0.86  0.85
11800  0.86  0.85
11900  0.87  0.85
12000  0.85  0.84
12100  0.85  0.85
12200  0.85  0.86
12300  0.86  0.86
12400  0.86  0.85
12500  0.84  0.86
12600  0.87  0.85
12700  0.85  0.86
12800  0.84  0.84
12900  0.85  0.86
13000  0.87  0.85
```

```
13100  0.85  0.86
13200  0.87  0.86
13300  0.87  0.85
13400  0.87  0.86
13500  0.86  0.85
13600  0.88  0.86
13700  0.88  0.86
13800  0.86  0.85
13900  0.84  0.84
14000  0.86  0.85
14100  0.87  0.86
14200  0.85  0.85
14300  0.87  0.85
14400  0.86  0.86
14500  0.86  0.86
14600  0.87  0.86
14700  0.84  0.85
14800  0.88  0.86
14900  0.88  0.86
```





The best validation accuracy I achieved after changing the stride to 2 was 86. I used a batch size of 64, 15000 optimization steps, and Adam as my optimizer with a learning rate of 0.001. My training and validation accuracies were about the same after running them for 15000 steps, however increasing the steps and batch size seems to give me a much higher training accuracy than validation accuracy which suggests that I had begun to overfit my training data. To increase performance further, possibly more convolutional layers may help me detect more complex features and give me a better accuracy. Increasing the channels may also help increase the accuracy of my predictions. I could also add in max pooling between the convolution layers to help with down sampling and reducing computational cost, which in turn will help me reduce overfitting.

In [103]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import torch
from torch import autograd
import torch.nn.functional as F
import csv
import skimage
import skimage.transform

images = np.load("./images.npy")
labels = np.load("./labels.npy")
test = np.load("./part_2_test_images.npy")
height = images.shape[1]
width = images.shape[2]
size = height * width
pre_images = images
images = (images - images.mean()) / images.std()
data = images.reshape(images.shape[0],size)
data = torch.from_numpy(data).float().cuda()
labels = torch.from_numpy(labels).float().cuda()
test_data = test.reshape(test.shape[0], size)
test_data = (test_data - test_data.mean()) / test_data.std()
test_data = torch.from_numpy(test_data).float().cuda()
batch_size = 1
NUM_OPT_STEPS = 5000
train_seqs = data[0:45000,:]
train_labels = labels[0:45000]
val_seqs = data[45000:,:]
val_labels = labels[45000:]
NUM_CLASSES = 5
```

In [11]:
```
class TooSimpleConvNN(torch.nn.Module):
    def __init__(self, chan_1, chan_2, chan_3, chan_4):
        super().__init__()
        # 3x3 convolution that takes in an image with one channel
        # and outputs an image with 8 channels.
        self.conv1 = torch.nn.Conv2d(1, chan_1, kernel_size=3)
        # 3x3 convolution that takes in an image with 8 channels
        # and outputs an image with 16 channels. The output image
        # has approximately half the height and half the width
        # because of the stride of 2.
        self.conv2 = torch.nn.Conv2d(chan_1, chan_2, kernel_size=3, stride
=1)
        self.conv3 = torch.nn.Conv2d(chan_2, chan_3, kernel_size=3, stride
=1)
        self.conv4 = torch.nn.Conv2d(chan_3, chan_4, kernel_size=3, stride
=1)
        # 1x1 convolution that takes in an image with 16 channels and
        # produces an image with 5 channels. Here, the 5 channels
        # will correspond to class scores.
        self.final_conv = torch.nn.Conv2d(chan_4, 5, kernel_size=1)
    def forward(self, x):
        # Convolutions work with images of shape
        # [batch_size, num_channels, height, width]
        x = x.view(-1, height, width).unsqueeze(1)

        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, kernel_size=3, stride=1)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, kernel_size=3, stride=1)
        x = F.relu(self.conv3(x))
        x = F.max_pool2d(x, kernel_size=3, stride=1)
        x = F.relu(self.conv4(x))
        x = F.max_pool2d(x, kernel_size=3, stride=2)
        n, c, h, w = x.size()
        x = F.avg_pool2d(x, kernel_size=[h, w])
        x = self.final_conv(x).view(-1, NUM_CLASSES)
        return x
```

In [12]:
```
def train(model, optimizer, batch_size):
#def train(batch_size):
    # model.train() puts our model in train mode, which can require differ
ent
    # behavior than eval mode (for example in the case of dropout).
    model.train()
    # i is is a 1-D array with shape [batch_size]
    i = np.random.choice(train_seqs.shape[0], size=batch_size, replace=Fal
se)
    i = torch.from_numpy(i).long().cuda()
    x = autograd.Variable(train_seqs[i, :])
    y = autograd.Variable(train_labels[i]).long()
    i.cpu()
    optimizer.zero_grad()
    y_hat_ = model(x)
    loss = F.cross_entropy(y_hat_, y)
    loss.backward()
    optimizer.step()
    return loss.data[0]
```

In [13]:
```
def approx_train_accuracy(model):
    i = np.random.choice(train_seqs.shape[0], size=1000, replace=False)
    i = torch.from_numpy(i).long().cuda()
    x = autograd.Variable(train_seqs[i, :])
    y = autograd.Variable(train_labels[i]).long()
    y_hat_ = model(x)
    y_hat = np.zeros(1000)
    for i in range(1000):
        y_hat[i] = torch.max(y_hat_[i,:].data, 0)[1][0]
    return accuracy(y_hat, y.data.cpu().numpy())
```

In [14]:
```
def val_accuracy(model):
    x = autograd.Variable(val_seqs)
    y = autograd.Variable(val_labels)
    y_hat_ = model(x)
    y_hat = np.zeros(5000)
    for i in range(5000):
        y_hat[i] = torch.max(y_hat_[i,:].data, 0)[1][0]
    return accuracy(y_hat, y.data.cpu().numpy())
```

In [15]:
```
def accuracy(y, y_hat):
    return (y == y_hat).astype(np.float).mean()
```

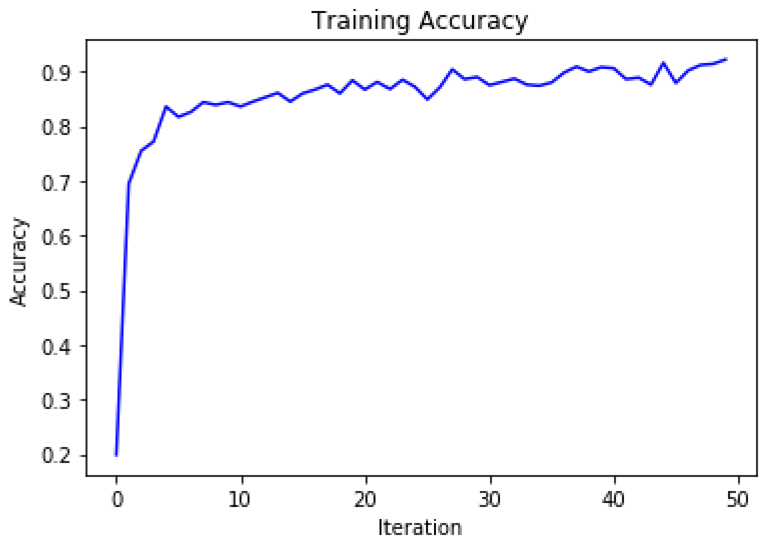
In [16]:
```
def plot(train_accs, val_accs):
    plt.figure(200)
    plt.title('Training Accuracy')
    plt.xlabel('Iteration')
    plt.ylabel('Accuracy')
    plt.plot(train_accs, 'b')
    plt.show()
    plt.figure(300)
    plt.title('Validation Accuracy')
    plt.xlabel('Iteration')
    plt.ylabel('Accuracy')
    plt.plot(val_accs, 'b')
    plt.show()
```

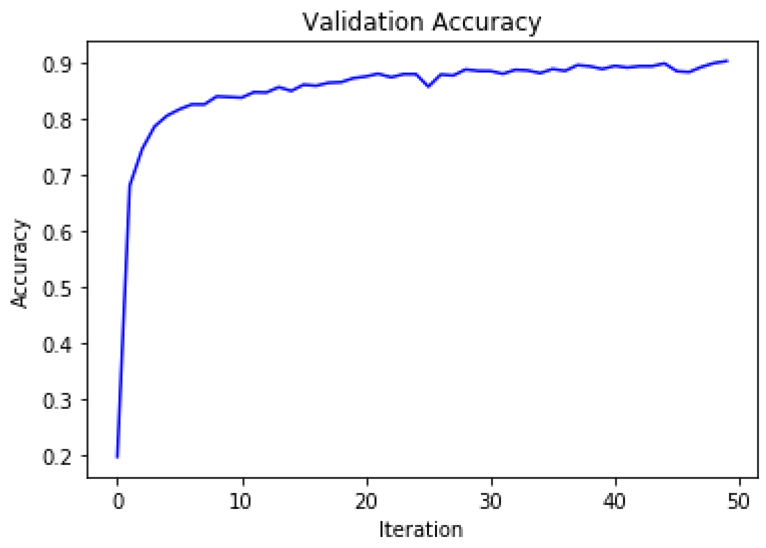
In [17]:
```
def runModel(model, batch_size, NUM_OPT_STEPS, optimizer):
    train_accs, val_accs = [], []
    for i in range(NUM_OPT_STEPS):
        train(model, optimizer, batch_size)
        if i % 100 == 0:
            train_accs.append(approx_train_accuracy(model))
            val_accs.append(val_accuracy(model))
            print("%6d %5.2f %5.2f" % (i, train_accs[-1], val_accs[-1]))
    plot(train_accs, val_accs)
```

In [110]:
```
layer_1 = 8 #average
layer_2 = 16
layer_3 = 24
layer_4 = 32
batch = 64
rate = 0.001
step = 5000
model = TooSimpleConvNN(layer_1, layer_2, layer_3, layer_4)
model.cuda()
optimizer = torch.optim.Adam(model.parameters(), lr=rate)
runModel(model, batch, step, optimizer)
```

```
     0  0.20  0.20
   100  0.69  0.68
   200  0.76  0.75
   300  0.77  0.79
   400  0.84  0.81
   500  0.82  0.82
   600  0.83  0.83
   700  0.84  0.83
   800  0.84  0.84
   900  0.84  0.84
  1000  0.84  0.84
  1100  0.84  0.85
  1200  0.85  0.85
  1300  0.86  0.86
  1400  0.84  0.85
  1500  0.86  0.86
```

```
1600  0.87  0.86
1700  0.88  0.86
1800  0.86  0.87
1900  0.88  0.87
2000  0.87  0.88
2100  0.88  0.88
2200  0.87  0.87
2300  0.89  0.88
2400  0.87  0.88
2500  0.85  0.86
2600  0.87  0.88
2700  0.90  0.88
2800  0.89  0.89
2900  0.89  0.89
3000  0.88  0.89
3100  0.88  0.88
3200  0.89  0.89
3300  0.88  0.89
3400  0.87  0.88
3500  0.88  0.89
3600  0.90  0.89
3700  0.91  0.90
3800  0.90  0.89
3900  0.91  0.89
4000  0.91  0.90
4100  0.89  0.89
4200  0.89  0.89
4300  0.88  0.89
4400  0.92  0.90
4500  0.88  0.89
4600  0.90  0.88
4700  0.91  0.89
4800  0.91  0.90
4900  0.92  0.90
```



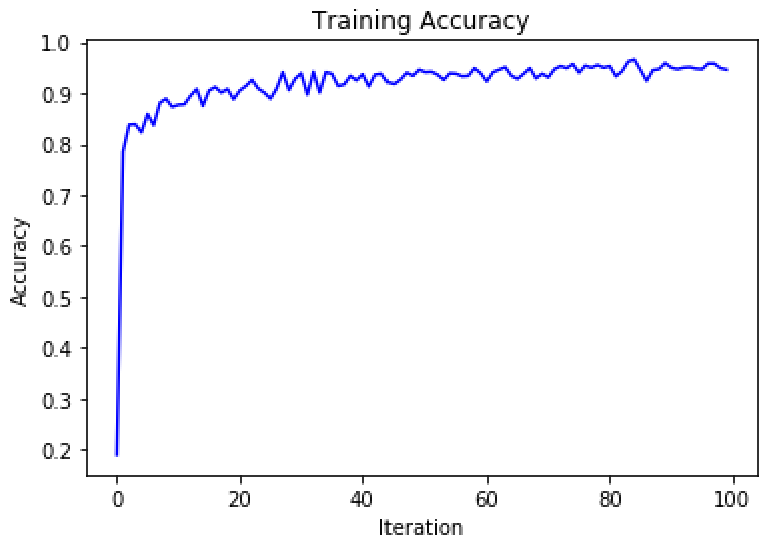


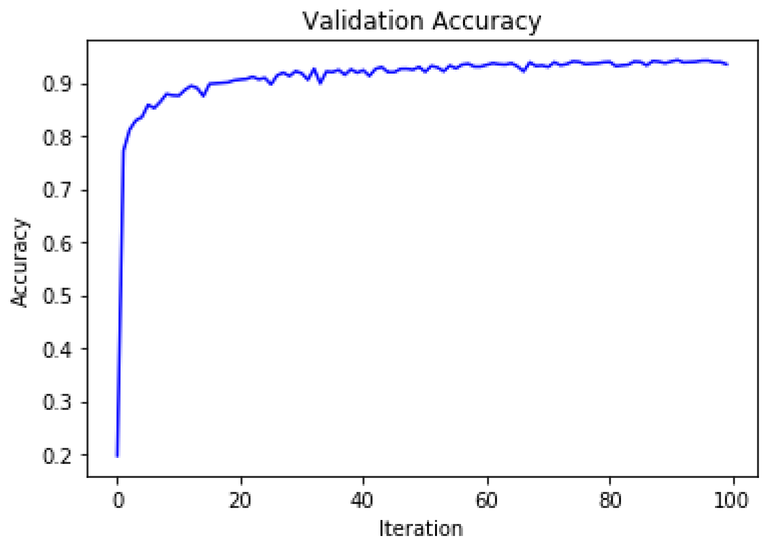
In [9]:
```
layer_1 = 16 #better
layer_2 = 32
layer_3 = 64
layer_4 = 128
batch = 50
rate = 0.001
step = 10000
model = TooSimpleConvNN(layer_1, layer_2, layer_3, layer_4)
model.cuda()
optimizer = torch.optim.Adam(model.parameters(), lr=rate)
runModel(model, batch, step, optimizer)
```

```
     0  0.19  0.20
   100  0.79  0.77
   200  0.84  0.81
   300  0.84  0.83
   400  0.82  0.83
   500  0.86  0.86
   600  0.84  0.85
   700  0.88  0.86
   800  0.89  0.88
   900  0.87  0.88
  1000  0.88  0.88
  1100  0.88  0.89
  1200  0.90  0.89
  1300  0.91  0.89
  1400  0.88  0.87
  1500  0.91  0.90
  1600  0.91  0.90
  1700  0.90  0.90
  1800  0.91  0.90
  1900  0.89  0.90
  2000  0.91  0.91
  2100  0.91  0.91
  2200  0.93  0.91
  2300  0.91  0.91
  2400  0.90  0.91
  2500  0.89  0.90
  2600  0.91  0.91
```

```
                    2700  0.94  0.92
                    2800  0.91  0.91
                    2900  0.93  0.92
                    3000  0.94  0.92
                    3100  0.90  0.91
                    3200  0.94  0.93
                    3300  0.90  0.90
                    3400  0.94  0.92
                    3500  0.94  0.92
                    3600  0.91  0.92
                    3700  0.92  0.91
                    3800  0.93  0.93
                    3900  0.93  0.92
                    4000  0.94  0.92
                    4100  0.91  0.91
                    4200  0.94  0.93
                    4300  0.94  0.93
                    4400  0.92  0.92
                    4500  0.92  0.92
                    4600  0.93  0.93
                    4700  0.94  0.93
                    4800  0.93  0.92
                    4900  0.95  0.93
                    5000  0.94  0.92
                    5100  0.94  0.93
                    5200  0.94  0.93
                    5300  0.93  0.92
                    5400  0.94  0.93
                    5500  0.94  0.93
                    5600  0.93  0.93
                    5700  0.93  0.94
                    5800  0.95  0.93
                    5900  0.94  0.93
                    6000  0.92  0.93
                    6100  0.94  0.94
                    6200  0.95  0.93
                    6300  0.95  0.93
                    6400  0.93  0.94
                    6500  0.93  0.93
                    6600  0.94  0.92
                    6700  0.95  0.94
                    6800  0.93  0.93
                    6900  0.94  0.93
                    7000  0.93  0.93
                    7100  0.95  0.94
                    7200  0.95  0.93
                    7300  0.95  0.94
                    7400  0.96  0.94
                    7500  0.94  0.94
                    7600  0.95  0.94
                    7700  0.95  0.94
                    7800  0.95  0.94
                    7900  0.95  0.94
                    8000  0.95  0.94
                    8100  0.93  0.93
                    8200  0.94  0.93
                    8300  0.96  0.93
```

```
8400  0.97  0.94
8500  0.94  0.94
8600  0.92  0.93
8700  0.94  0.94
8800  0.95  0.94
8900  0.96  0.94
9000  0.95  0.94
9100  0.95  0.94
9200  0.95  0.94
9300  0.95  0.94
9400  0.95  0.94
9500  0.95  0.94
9600  0.96  0.94
9700  0.96  0.94
9800  0.95  0.94
9900  0.95  0.93
```



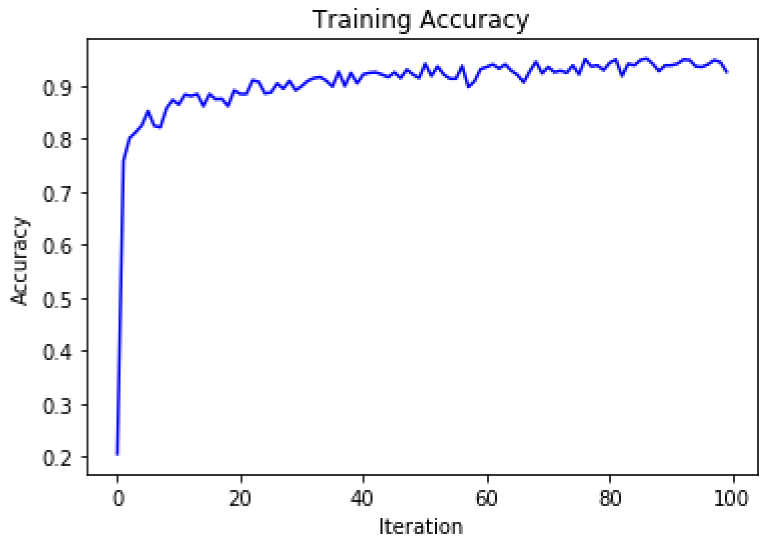


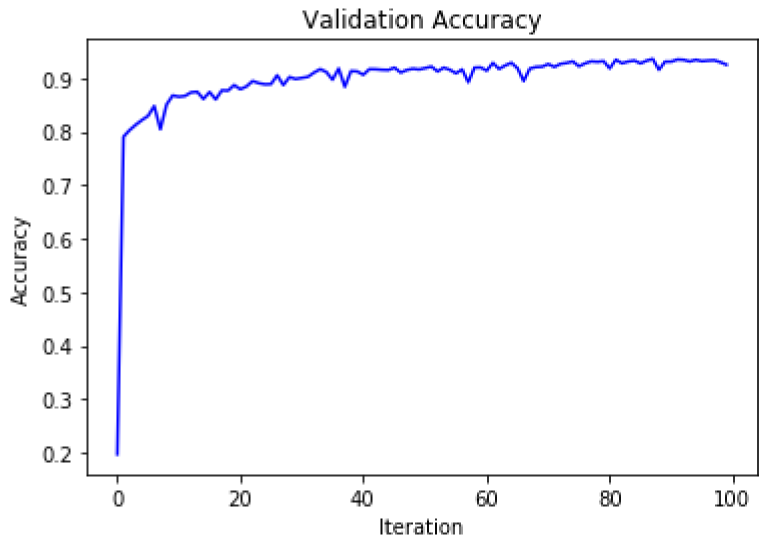
In [9]:
```
layer_1 = 16 #better
layer_2 = 32
layer_3 = 64
layer_4 = 128
batch = 32
rate = 0.001
```

```
step = 10000
model = TooSimpleConvNN(layer_1, layer_2, layer_3, layer_4)
model.cuda()
optimizer = torch.optim.Adam(model.parameters(), lr=rate)
runModel(model, batch, step, optimizer)
```

```
     0  0.20  0.20
   100  0.76  0.79
   200  0.80  0.80
   300  0.81  0.81
   400  0.82  0.82
   500  0.85  0.83
   600  0.82  0.85
   700  0.82  0.80
   800  0.86  0.85
   900  0.87  0.87
  1000  0.86  0.87
  1100  0.88  0.87
  1200  0.88  0.87
  1300  0.88  0.87
  1400  0.86  0.86
  1500  0.88  0.87
  1600  0.87  0.86
  1700  0.88  0.88
  1800  0.86  0.88
  1900  0.89  0.89
  2000  0.88  0.88
  2100  0.88  0.88
  2200  0.91  0.89
  2300  0.91  0.89
  2400  0.89  0.89
  2500  0.89  0.89
  2600  0.90  0.91
  2700  0.89  0.89
  2800  0.91  0.90
  2900  0.89  0.90
  3000  0.90  0.90
  3100  0.91  0.90
  3200  0.91  0.91
  3300  0.92  0.92
  3400  0.91  0.91
  3500  0.90  0.90
  3600  0.93  0.92
  3700  0.90  0.88
  3800  0.92  0.91
  3900  0.90  0.91
  4000  0.92  0.91
  4100  0.92  0.92
  4200  0.93  0.92
  4300  0.92  0.92
  4400  0.92  0.92
  4500  0.93  0.92
  4600  0.91  0.91
  4700  0.93  0.92
  4800  0.92  0.92
  4900  0.91  0.92
  5000  0.94  0.92
```

```
5100  0.92  0.92
5200  0.94  0.91
5300  0.92  0.92
5400  0.91  0.92
5500  0.91  0.91
5600  0.94  0.92
5700  0.90  0.89
5800  0.91  0.92
5900  0.93  0.92
6000  0.94  0.91
6100  0.94  0.93
6200  0.93  0.92
6300  0.94  0.92
6400  0.93  0.93
6500  0.92  0.92
6600  0.91  0.89
6700  0.93  0.92
6800  0.94  0.92
6900  0.92  0.92
7000  0.94  0.93
7100  0.93  0.92
7200  0.93  0.93
7300  0.92  0.93
7400  0.94  0.93
7500  0.92  0.92
7600  0.95  0.93
7700  0.94  0.93
7800  0.94  0.93
7900  0.93  0.93
8000  0.94  0.92
8100  0.95  0.93
8200  0.92  0.93
8300  0.94  0.93
8400  0.94  0.93
8500  0.95  0.93
8600  0.95  0.93
8700  0.94  0.94
8800  0.93  0.92
8900  0.94  0.93
9000  0.94  0.93
9100  0.94  0.93
9200  0.95  0.93
9300  0.95  0.93
9400  0.94  0.93
9500  0.94  0.93
9600  0.94  0.93
9700  0.95  0.93
9800  0.94  0.93
9900  0.93  0.93
```





My starting point was with the basic two layer neural network. I tried optimizing the hyper parameters for it and found that my accuracy was capped around 86.

I started trying more convolutional layers to get better accuracy and was able to raise it to 94-95 accuracy using 4 convolutional layers.

The optimizer I used was the Adam optimizer at a learning rate of 0.001 with a mini batch size of 64. I tried varying the batch size but when I chose something over 100 my training became incredibly slow. My training was also particularly slow when I increased the number of channels at every convolution layer. To circumvent this problem i decided to train on my GPU which allowed for faster training.

Along with that, intially I had a bit of overfitting with my model and to decrease my overfitting I decided to max pool after every convolution layer to help with down sampling.

The most important changes to achieving high accuracy I made were increasing the number of layers, and getting a pyramid like structure with my channels.

What my model does is that it has 4 convolution layers which each output an image with different number of channels. The number of channels I usually set it up with are in a "pyramid" shape, IE 32 64 128 256. When my model makes its prediction it will first take in image data with one channel and then take a 3x3 convolution and then output an image with chan_1 number of channels. I then send that output into my relu activation function and run a max pool in order to get data that is closer bounded and get some down sampling to reduce some overfitting. I then run through the rest of the layers in the same fashion with the only difference being that each layer outputs an image with a different number of channels. I still feed my output of every layer into my relu activation function and run a max pool each time. At the very end I run a 1x1 convolution and output an image with 5 channels that correspond to the class scores. I then take the argmax of those scores and use that as my prediction.

In [20]:
```
with open('jzhan127_part2.csv', 'w', newline='') as csvfile:
    filewriter = csv.writer(csvfile, delimiter=',',quotechar='|', quoting=
csv.QUOTE_MINIMAL)
    filewriter.writerow(['id', 'label'])

    x = autograd.Variable(test_data)
    y_hat_ = model(x)
    for i in range(5000):
        filewriter.writerow([i, torch.max(y_hat_[i,:].data, 0)[1][0]])
```

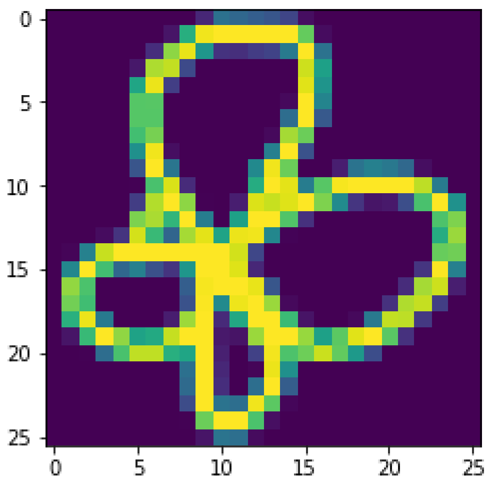
Kaggle Submission: jzhan127_part2.csv

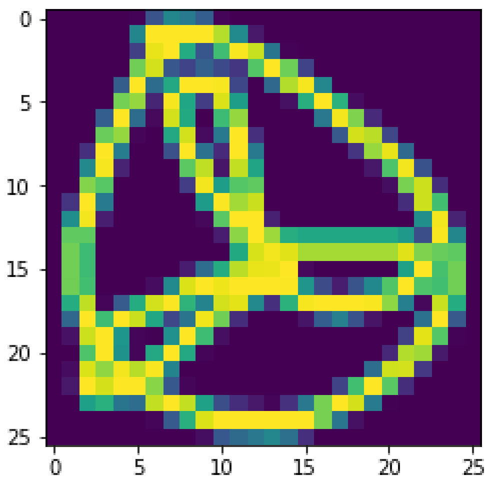
EXPLORING FAILURE MODES

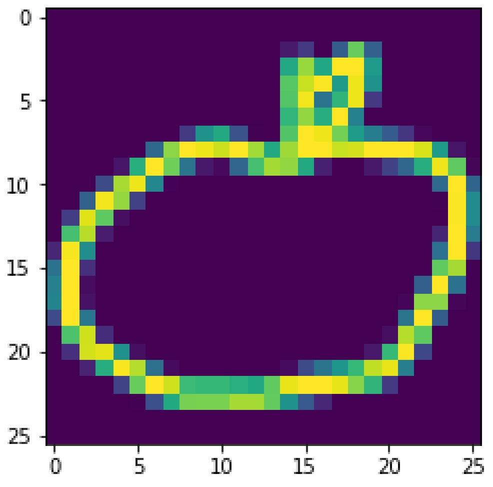
In [18]:
```
with open('test.csv', 'w', newline='') as csvfile:
    filewriter = csv.writer(csvfile, delimiter=',',quotechar='|', quoting=
csv.QUOTE_MINIMAL)
    filewriter.writerow(['id', 'label'])

    x = autograd.Variable(val_seqs)
    y_hat_ = model(x)
    for i in range(45000, 50000):
        filewriter.writerow([i-45000, torch.max(y_hat_[i - 45000,:].data,
0)[1][0], val_labels[i - 45000]])
```

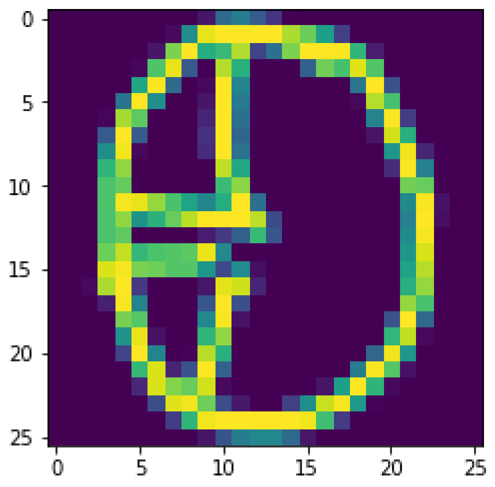
In [79]:
```
wrong = [images[45000], images[45002], images[45004], images[45012], image
s[45013], images[45017], images[45024], images[45038], images[45041], imag
es[45075]]
right = [images[45001], images[45003], images[45005], images[45006], image
s[45007], images[45008], images[45010], images[45014], images[45015], imag
es[45029]]
print("RIGHT IMAGES")
for i in range(10):
    plt.figure(i)
    plt.imshow(right[i])
```

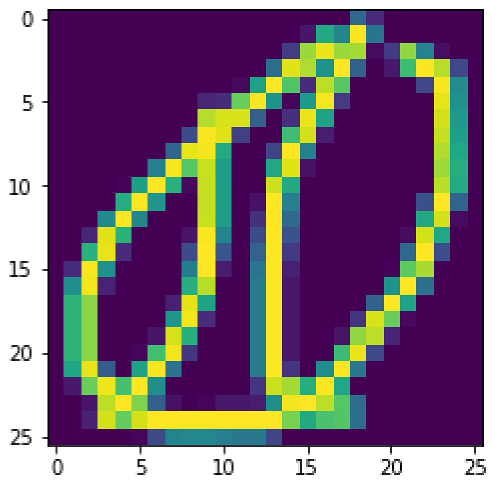
```
RIGHT IMAGES
```

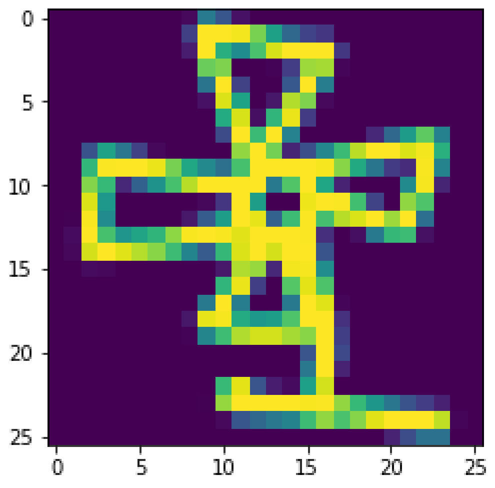

0
5
10
15
20
25
0
5
10
15
20
25

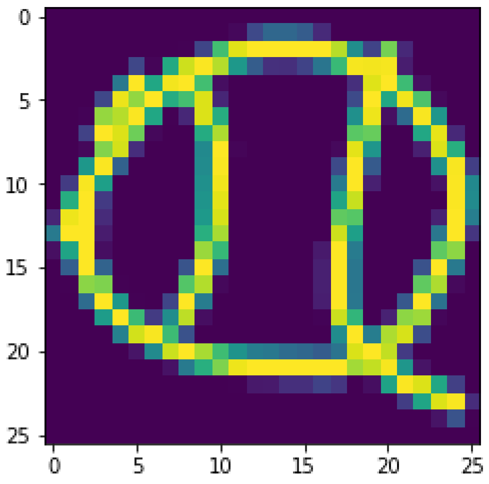

0
5
10
15
20
25
0
5
10
15
20
25

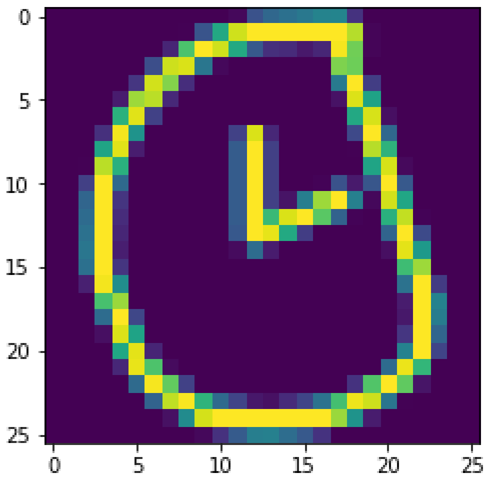

0
5
10
15
20
25
0
5
10
15
20
25

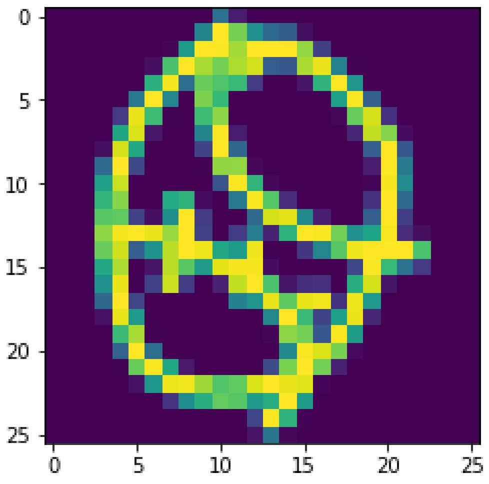

0
5
10
15
20
25
0
5
10
15
20
25

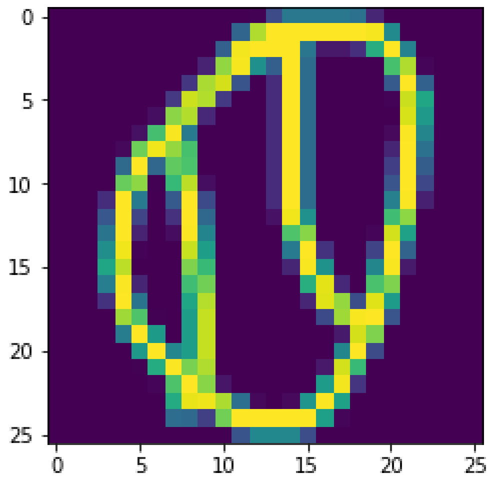

0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25

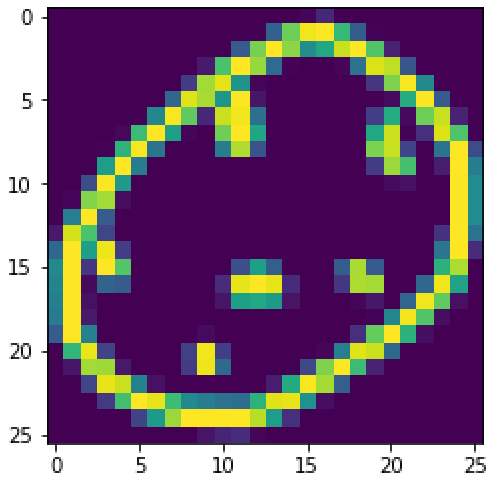

0
5
10
15
20
25
0
5
10
15
20
25

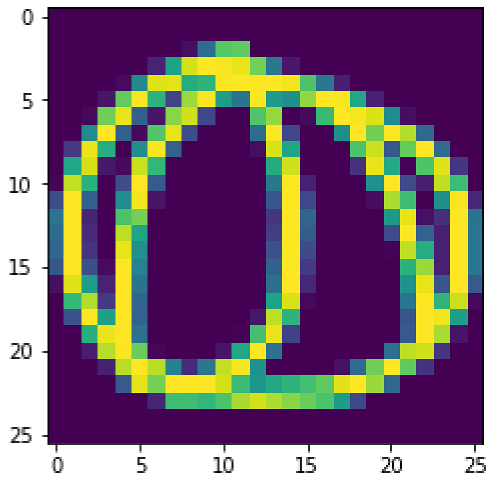

0
5
10
15
20
25
0
5
10
15
20
25

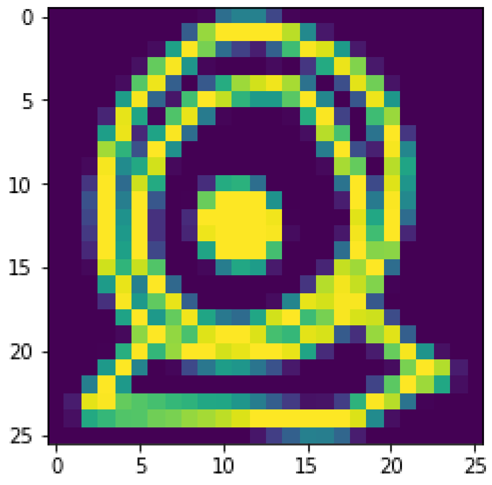


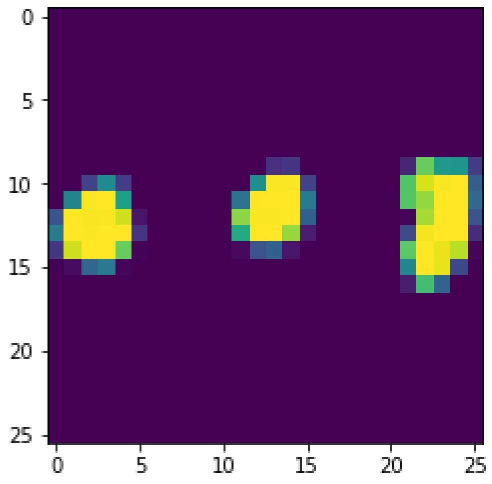
In [78]:
```
print("WRONG IMAGES")
for j in range(10):
    plt.figure(j+10)
    plt.imshow(wrong[j])
```

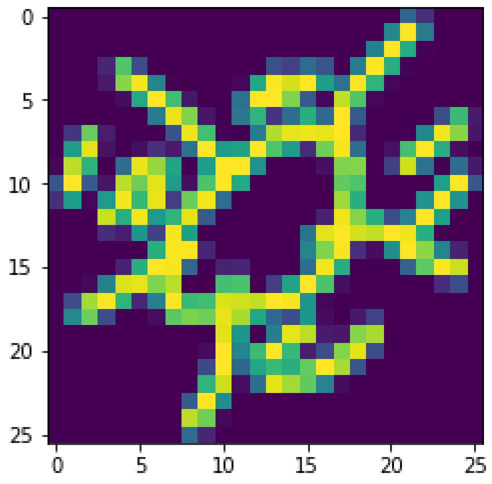
```
WRONG IMAGES
```

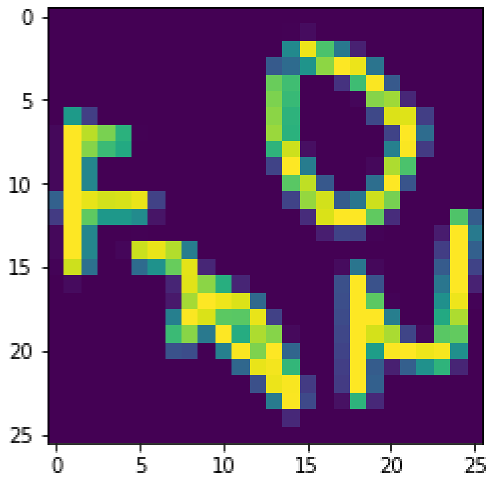


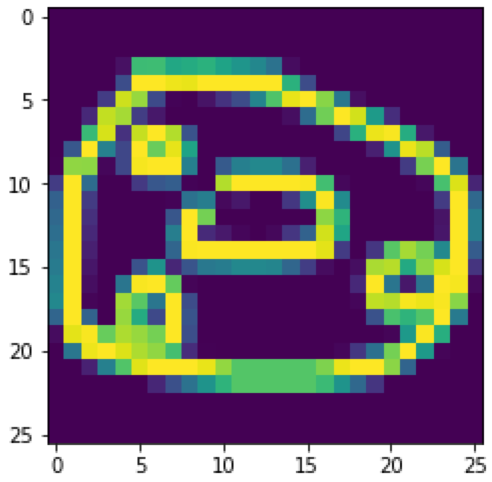


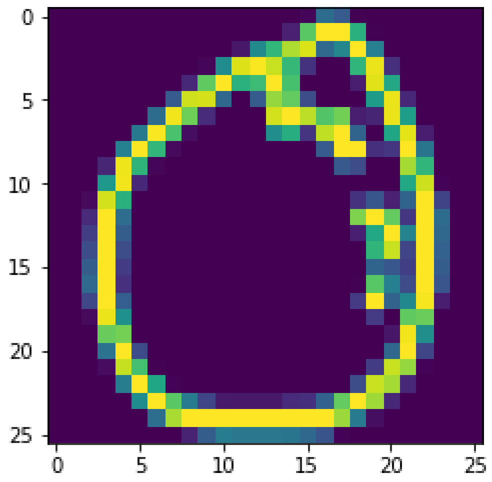

0
5
10
15
20
25
0
5
10
15
20
25

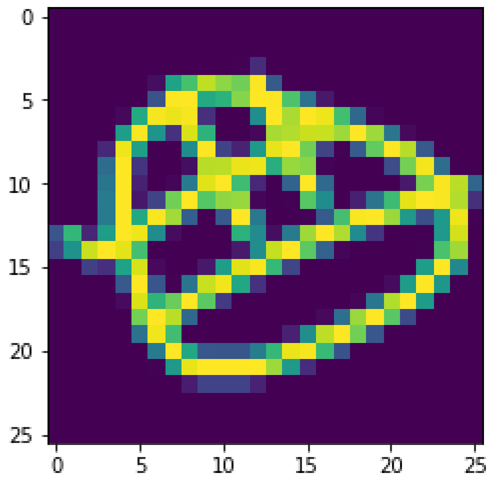

0
5
10
15
20
25
0
5
10
15
20
25

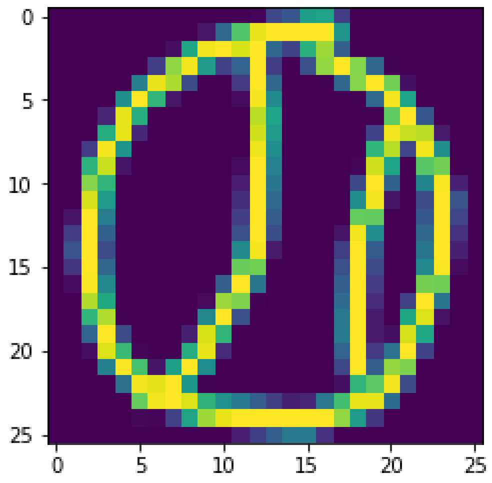

0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25

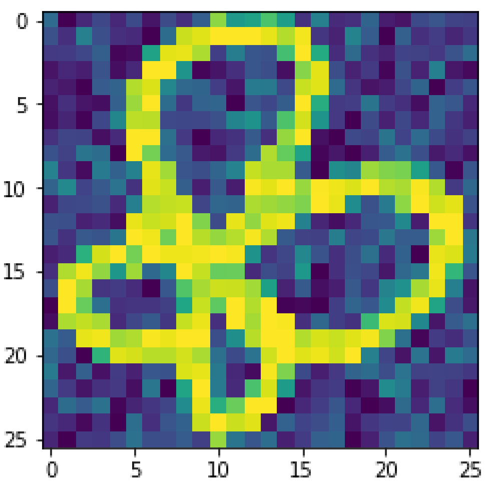

0
5
10
15
20
25
0
5
10
15
20
25

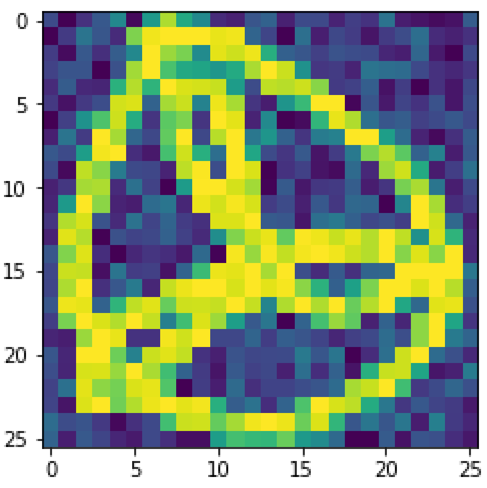


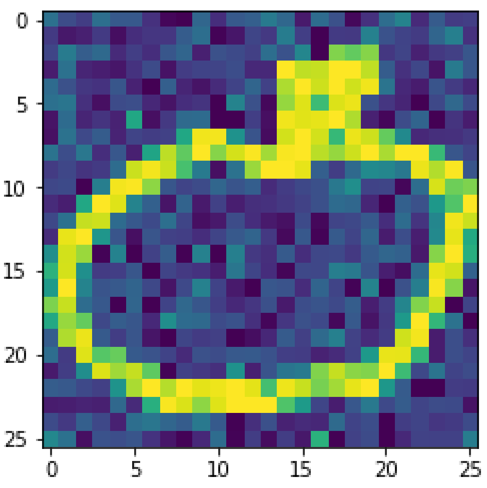


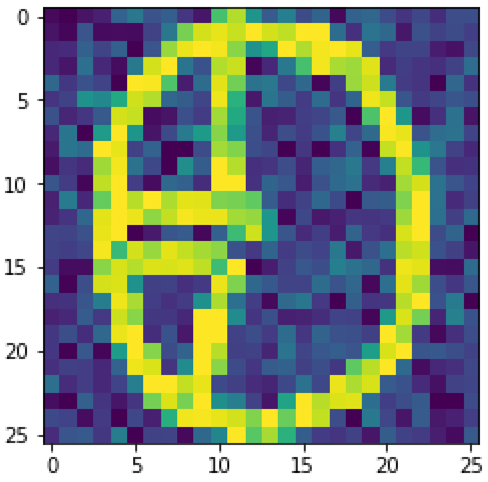
The image quality of both sets are about the same. However in the wrong set the images seem to have slightly different features that make it more difficult to classify. The misclassified examples are slightly more difficult to classify as a human because it is hard to make out what the image is. They're fairly unclear and messy which make it hard to discern what the image is supposed to be.

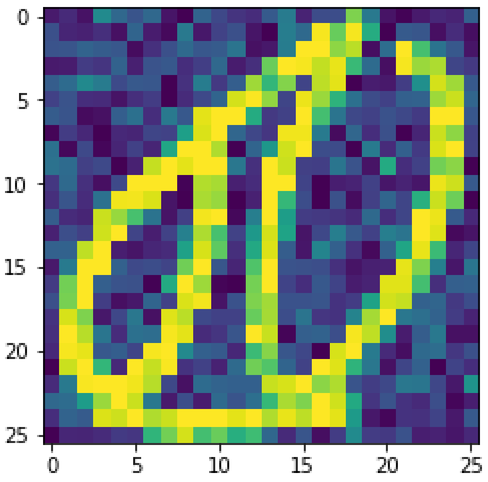
In [88]:
```
noise_add = right
for i in range(10):
    noise_add[i]= skimage.util.random_noise(noise_add[i], mode='gaussian',
 seed=None, clip=True)
    plt.figure(i)
    plt.imshow(noise_add[i])
```

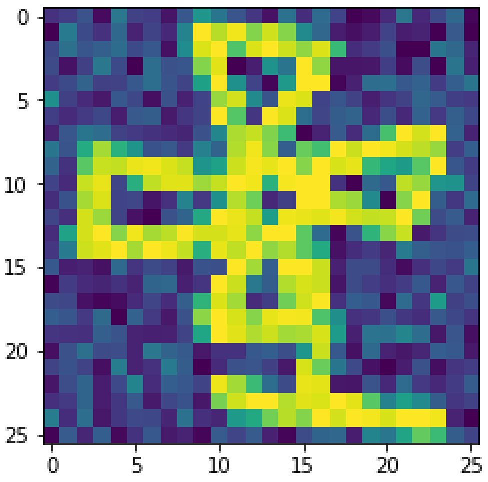

0
5
10
15
20
25
0
5
10
15
20
25

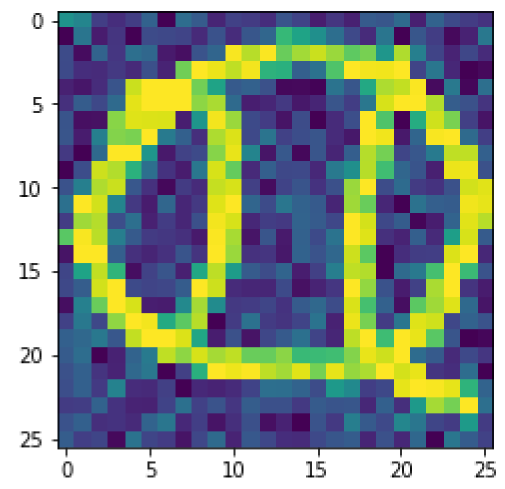

0
5
10
15
20
25
0
5
10
15
20
25

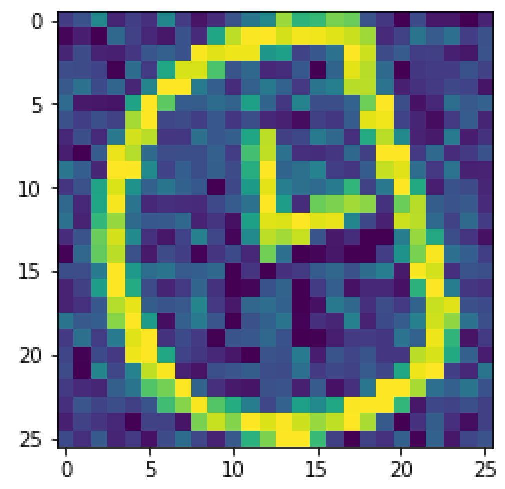

0
5
10
15
20
25
0
5
10
15
20
25

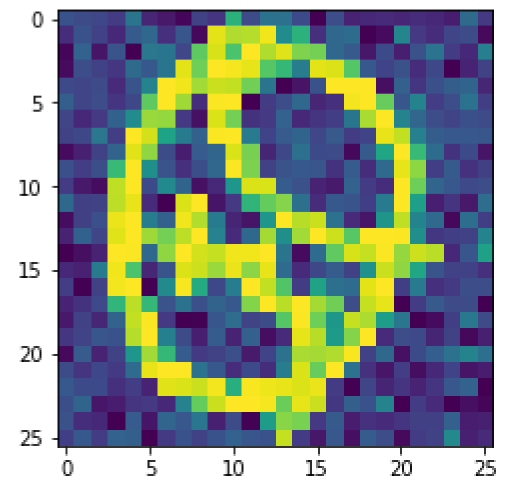

0
5
10
15
20
25
0
5
10
15
20
25

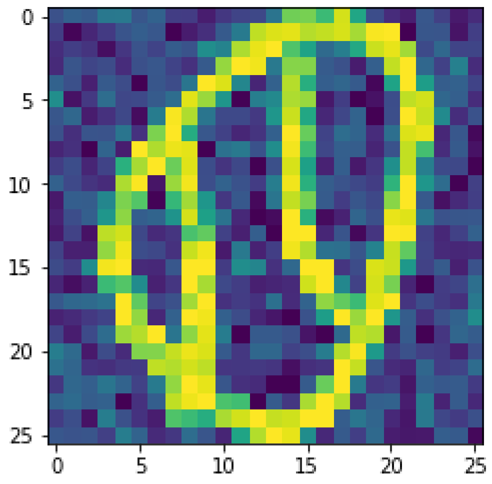

0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25

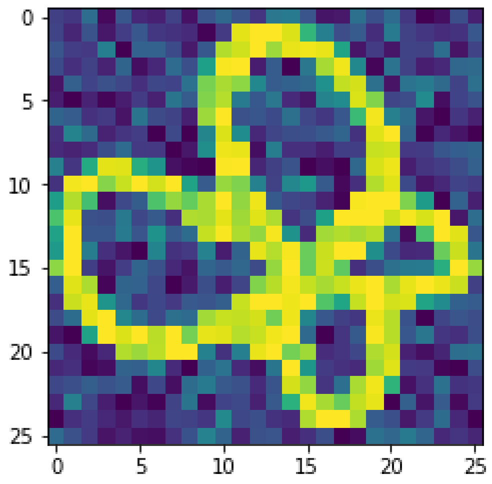


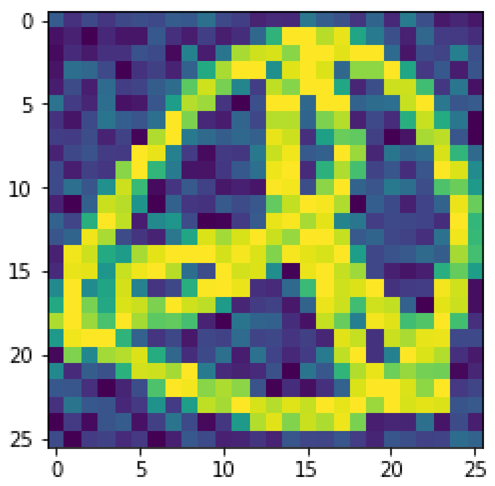
In [97]:
```
right_lab = [labels[45001], labels[45003], labels[45005], labels[45006], l
abels[45007], labels[45008], labels[45010], labels[45014], labels[45015],
labels[45029]]
right_lab = np.array(right_lab)
x = autograd.Variable(torch.from_numpy(np.array(noise_add)).cuda().float()
)
y_hat_ = model(x)
y_hat = np.zeros(10)
for i in range(10):
    y_hat[i] = torch.max(y_hat_[i,:].data, 0)[1][0]
print(accuracy(y_hat, right_lab))
```

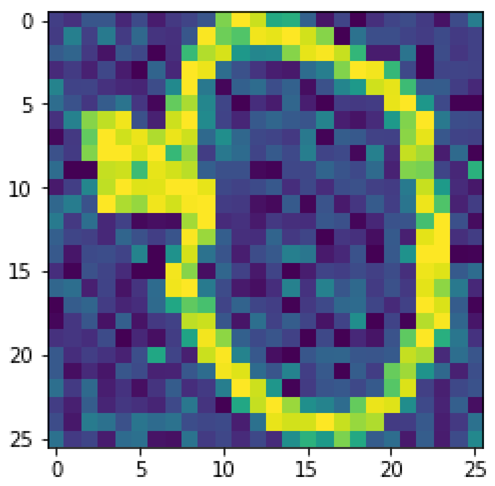
```
0.1
```

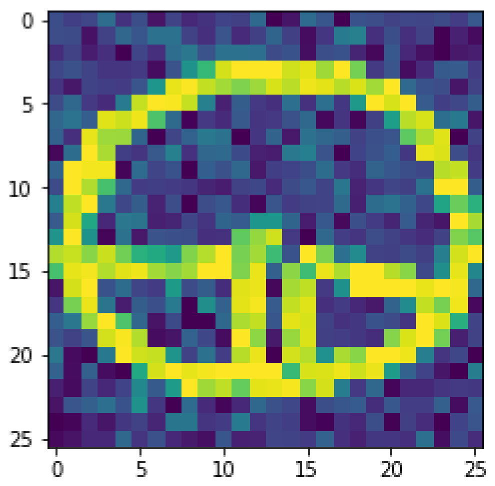
It does not classify the 10 images correctly. In fact it led to an even worse accuracy.

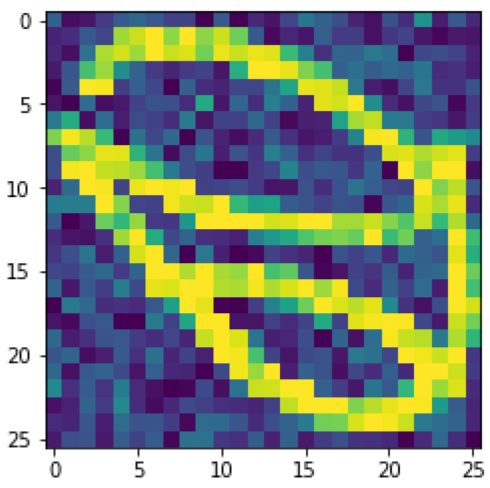
In [106]:
```
to_flip = right
for i in range(10):
    to_flip[i] = skimage.transform.rotate(to_flip[i], 90)
    plt.figure(i)
    plt.imshow(to_flip[i])
```

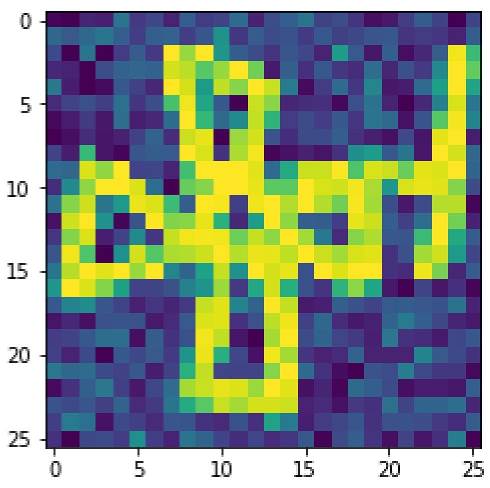


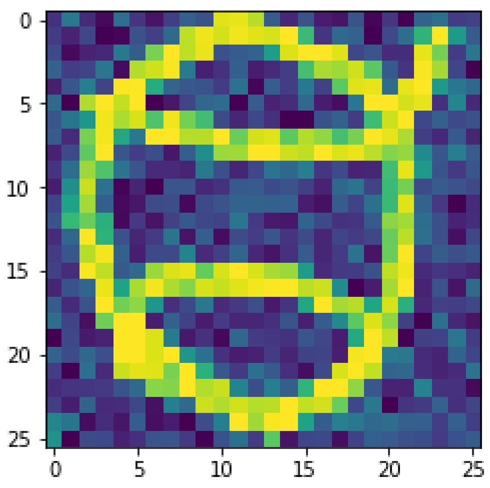


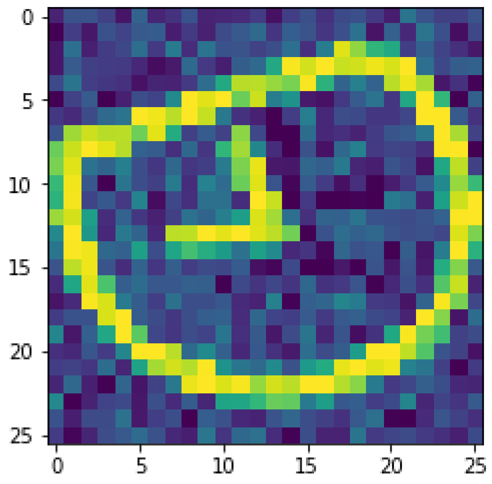


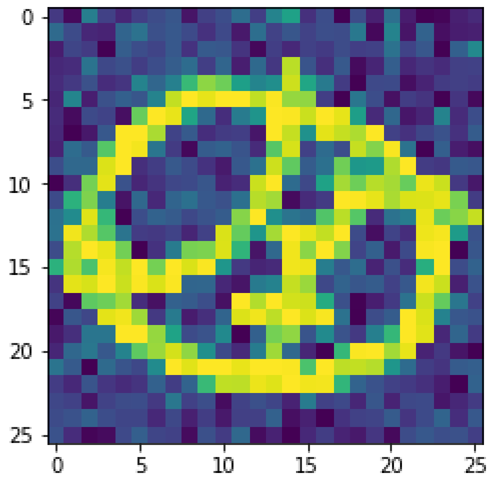


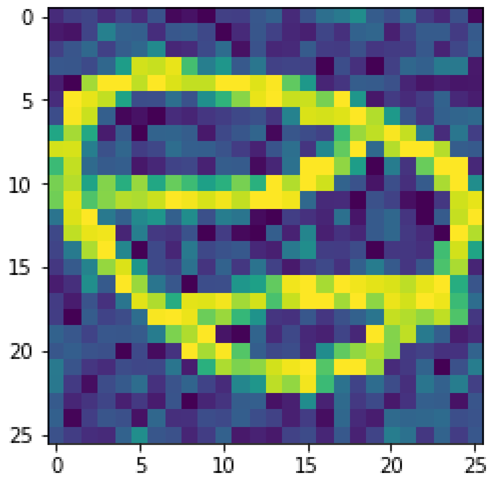

0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25







In [113]:
```
right_lab = [labels[45001], labels[45003], labels[45005], labels[45006], l
abels[45007], labels[45008], labels[45010], labels[45014], labels[45015],
labels[45029]]
right_lab = np.array(right_lab)
x = autograd.Variable(torch.from_numpy(np.array(to_flip)).cuda().float())
y_hat_ = model(x)
y_hat = np.zeros(10)
for i in range(10):
    y_hat[i] = torch.max(y_hat_[i,:].data, 0)[1][0]
print(accuracy(y_hat, right_lab))
```

```
0.0
```

My classifier was not able to classify these 10 images correctly.

Yes there are scenarios when we want to remain invariant to horizontal flipping because there are certain objects that may be sensitive to orientation which would allow us to classify them as those objects. To remain invariant to horizontal flipping we could randomly flip images when we are training and our model will have to train with these flips which will allow our model to be sensitive to flipping and allow our model to remain robust to such transformations.

Kaggle submission: jzhan127_part2.csv