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')
```

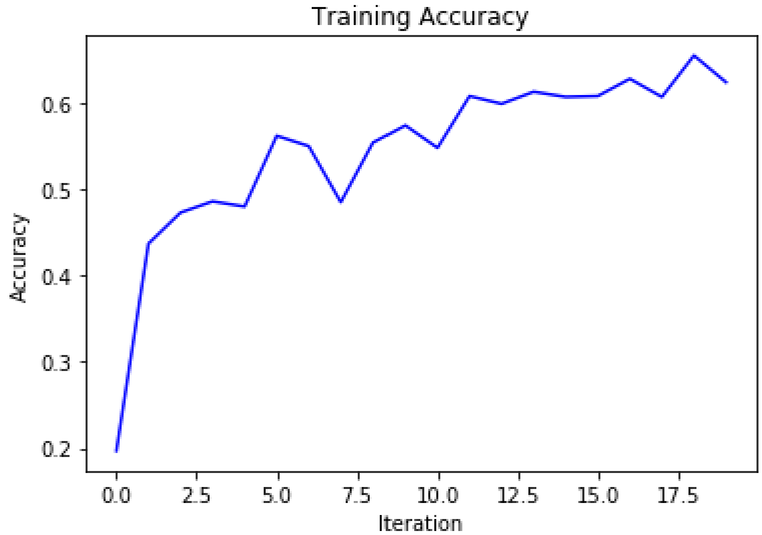
```
    plt.xlabel('Iteration')
    plt.ylabel('Accuracy')
    plt.plot(train_accs, 'b')
    plt.show()
    plt.figure(300)
    plt.title('Validation Accuracy')
    plt.xlabel('Iteration')
    plt.ylabel('Accuracy')
    plt.plot(val_accs, 'b')
    plt.show()
```

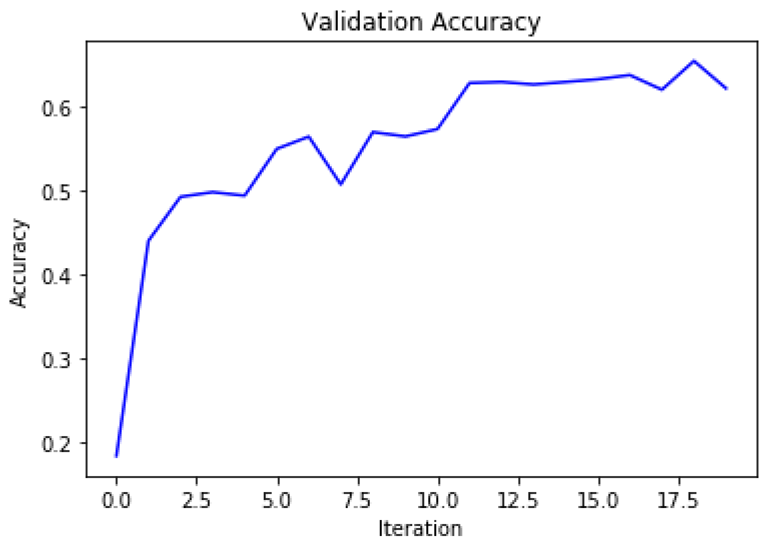
In [25]:
```
def runModel(model, batch_size):
    train_accs, val_accs = [], []
    for i in range(NUM_OPT_STEPS):
        train(batch_size)
        if i % 100 == 0:
            train_accs.append(approx_train_accuracy(model))
            val_accs.append(val_accuracy(model))
            print("%6d %5.2f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





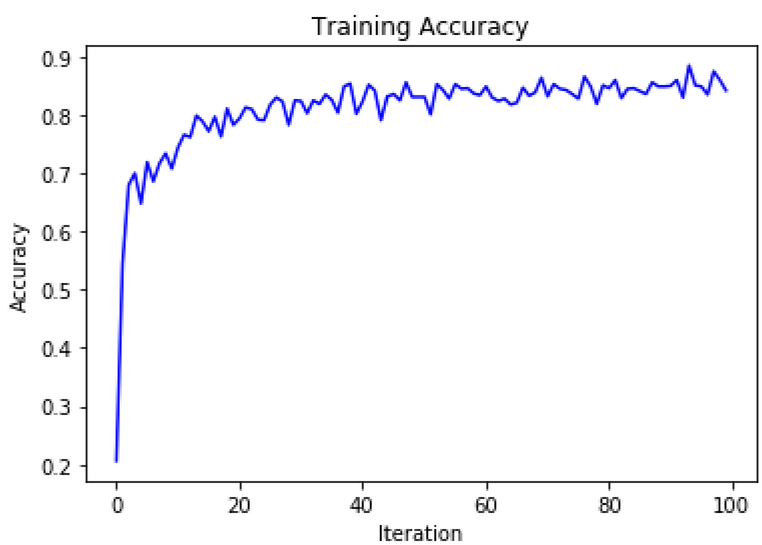
In [14]:

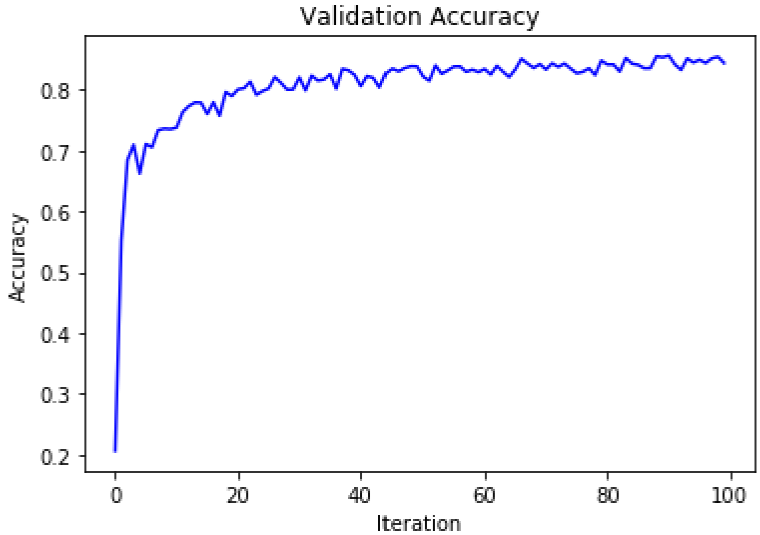
```
reset(model)
start = time.time()
runModel(model, 60)#10k steps
end = time.time()
print(end - start)
```

```
   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.