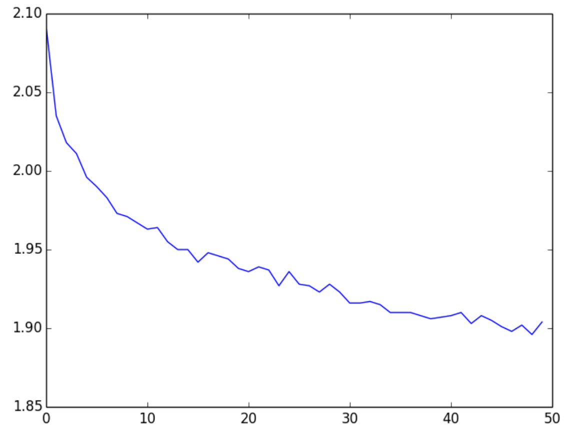
CSE 455 HW 5 Report

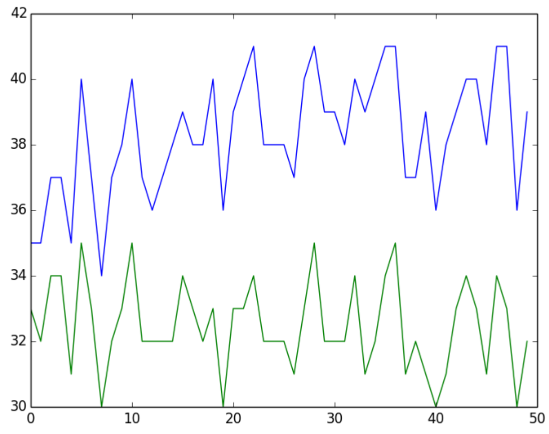
Akshat Shrivastava
CSE 455
HW 5

2.1: Lazy Net

Layers:

```
self.layer = nn.Linear(32 * 32 * 3, 10)
```





Analyze the behavior of your model (how well does it work?, how fast does it train?, why do you think it's working/not working?) and report the plots in your report file.

We can see the result of running in the graphs above, even though the loss was steadily decreasing, the accuracy on the train and test set was all over the place, so it seems like the classifier was not working properly. I suspect it is not working due to the simplicity of the network.

Running for 50 epochs on a GPU, the classifier took around 10 minutes to run.

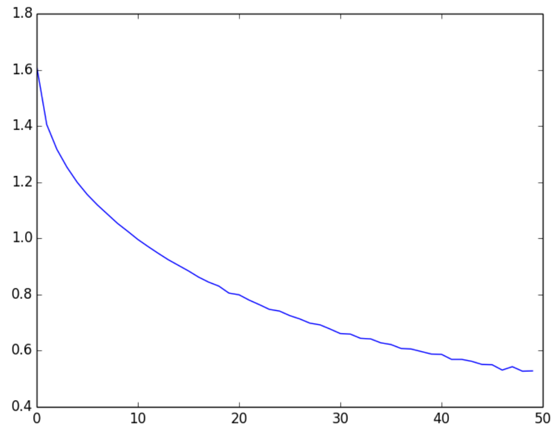
2.2: Boring Net

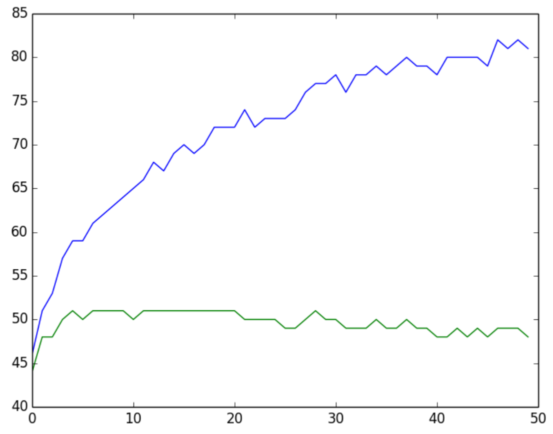
Layers:

```
self.layers = nn.Sequential (
    nn.Linear(32 * 32 * 3, 120),
    nn.ReLU(),
    nn.Linear(120, 84),
    nn.ReLU(),
    nn.Linear(84, 10)
)
```

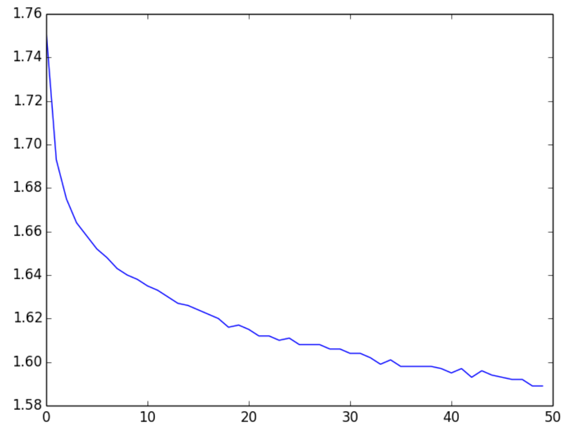
Note: the activation functions were commented out to see the difference between including them and excluding them.

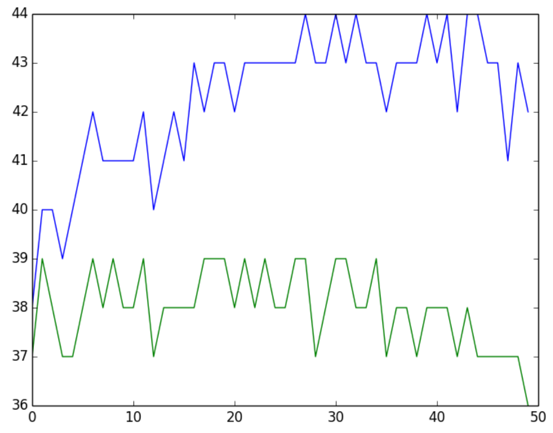
With Activation





Without Activation





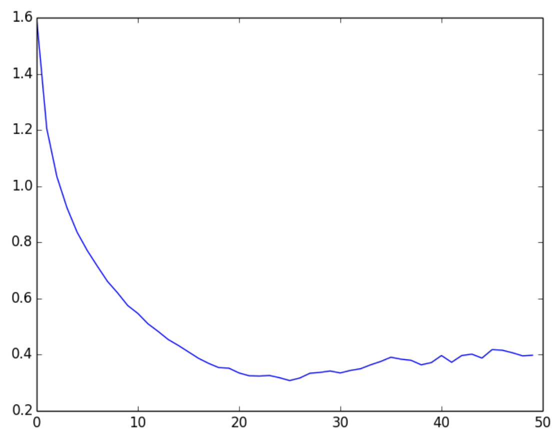
We can see that the activation function resulted in significantly better results than without activation, and training did overfit the data quite a bit. Since the gap between train and test is incredibly large towards the end. I suspect this is the case because using an activation function allows each node in the network to turn "on" and "off" depending on the input, so not all nodes will be active for each input, which allows better generalization.

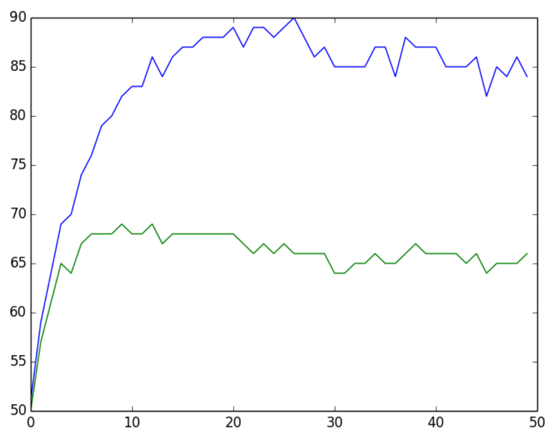
2.3: CoolNet

Layers:

```
self.firstLayer = nn.Sequential(
    nn.Conv2d(3, 32, kernel_size=8, stride=1),
    nn.ReLU(),
    nn.MaxPool2d(2)
)
self.secondLayer = nn.Sequential(
    nn.Conv2d(32, 64, kernel_size=8, stride=1),
    nn.ReLU(),
    nn.MaxPool2d(2, stride=1)
)
self.lin1 = nn.Linear(1024, 100)
self.lin2 = nn.Linear(100, 10)
```

For our ML class I had tuned a CNN and the layers to work for MNIST and found pretty good results with the network above, so I decided to use a similar network for the CIFAR dataset.





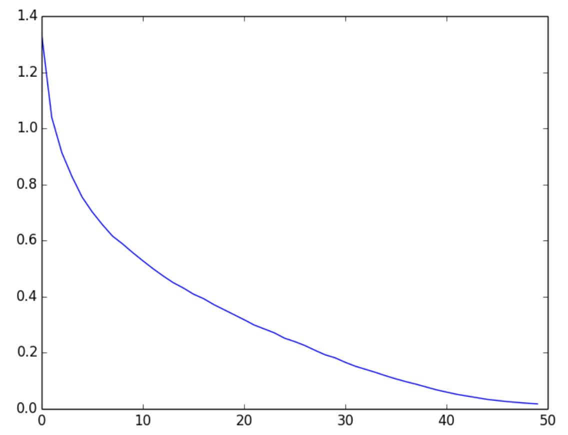
The model does seem to perform significantly better than LazyNet and BoringNet. The accuracy achieved here is higher than both the other models, however it did take a lot longer to train. The network still seems to overfit, but both the test and train accuracies are higher.

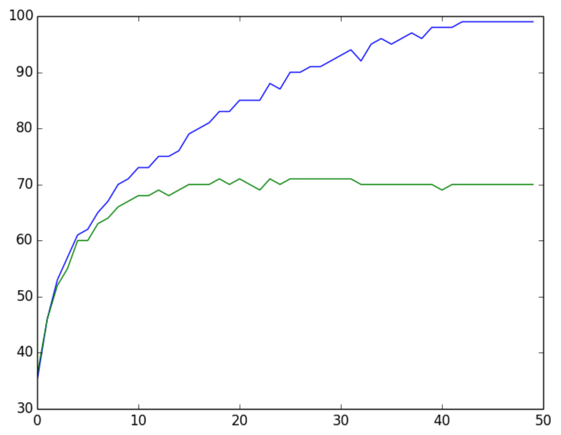
2.3.1 Batch Sizes

To test different batch sizes I tried the following batches: 16, 32, 64, 128, 256

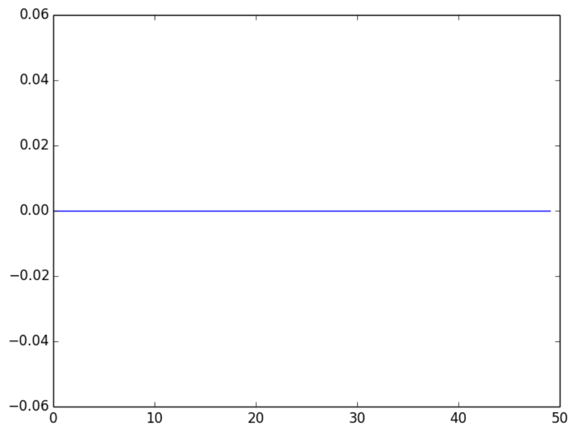
Here are my results:

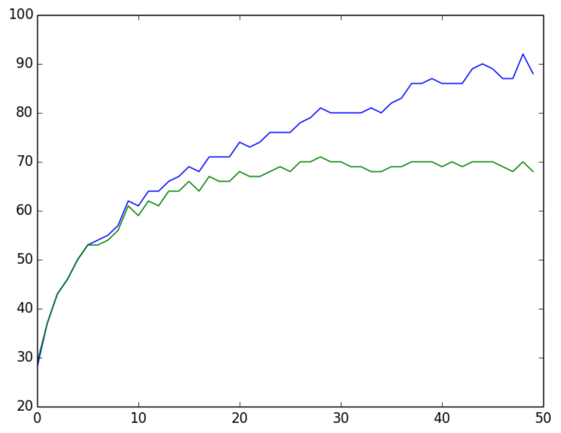
Batch Size: 16



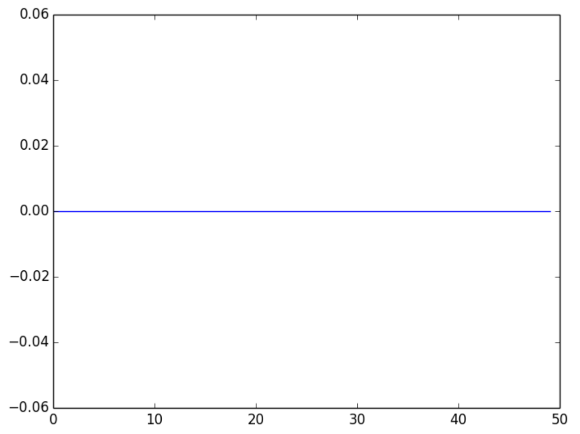


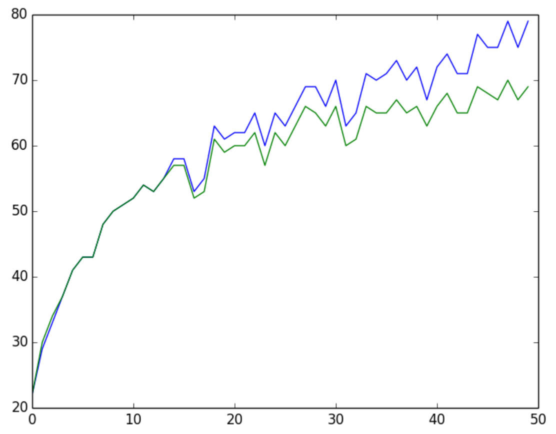
Batch Size: 32



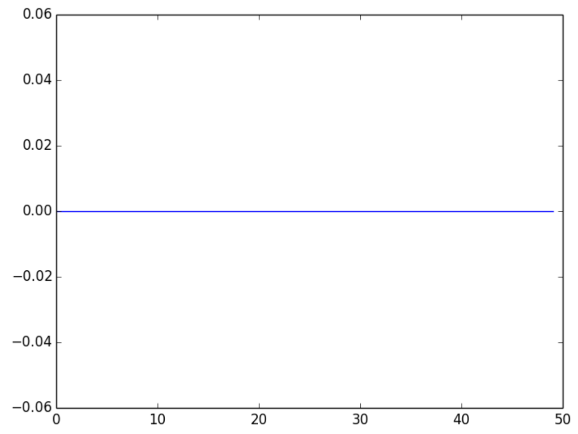


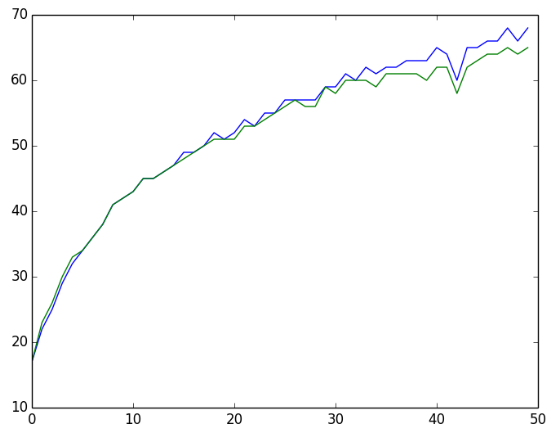
Batch Size: 64



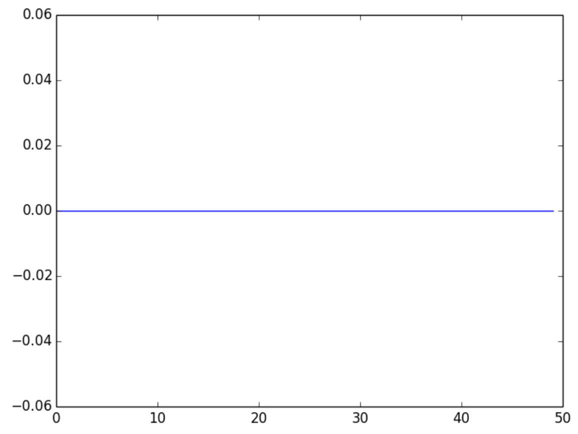


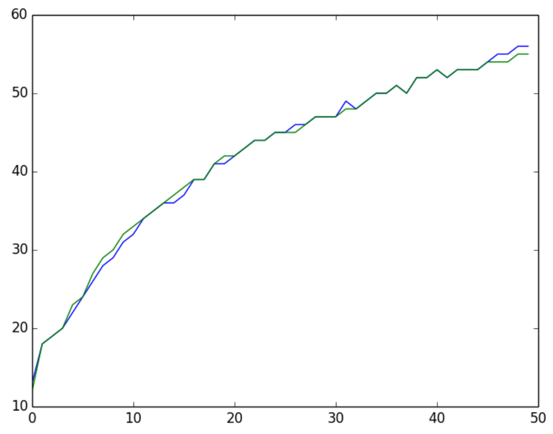
Batch Size: 128





Batch Size: 256





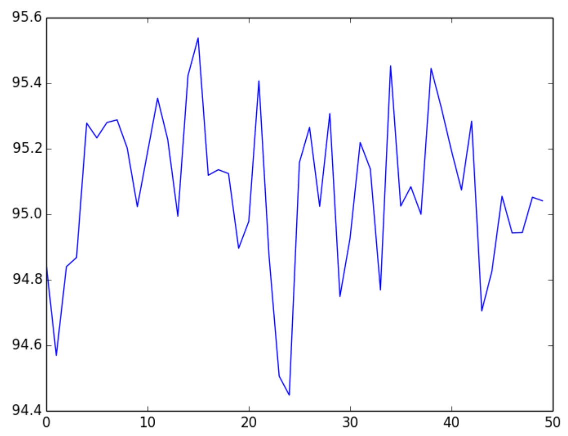
The trend we can see here is that the higher the batch size, the closer the train and test set seem to perform. The accuracies tend to climb up as well, with a peak being around a batch size of 128 or 256.

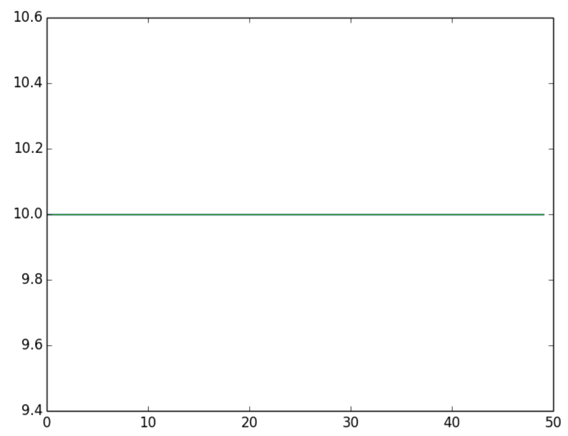
Since I was using a GPU and Cuda, I also realized that a larger batch size tended to increase my performance fairly well. This resulted in a larger batch size (256) being the quickest one.

3 Learning Rate

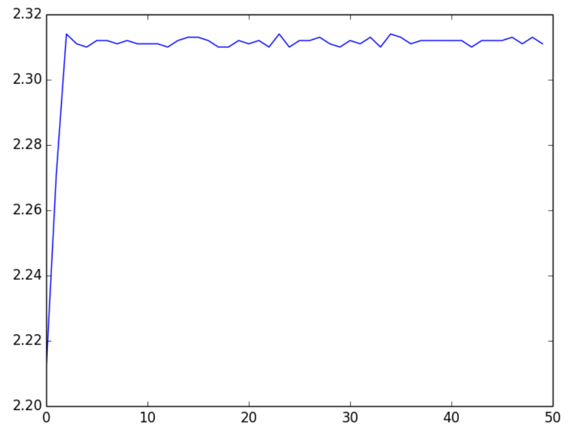
Listed below are the different learning rates I tried out:

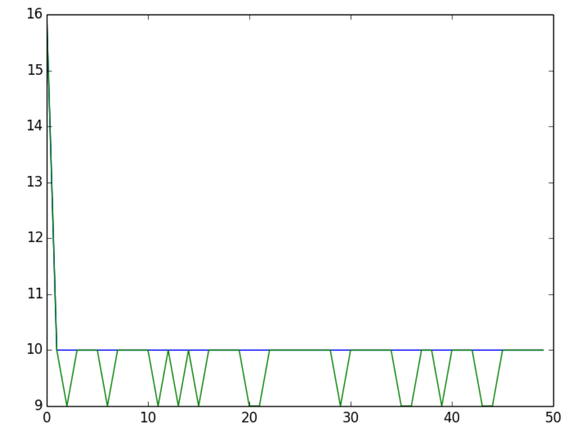
Learning Rate: 10



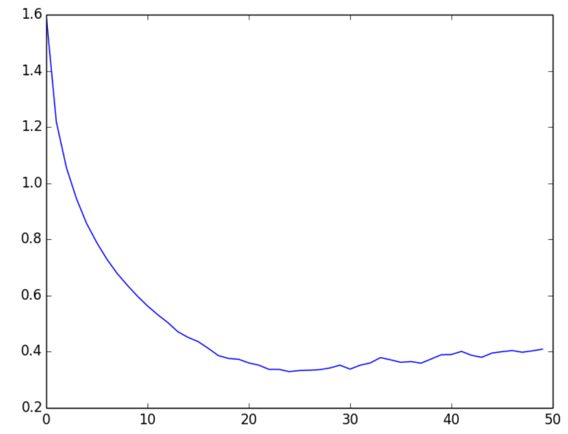


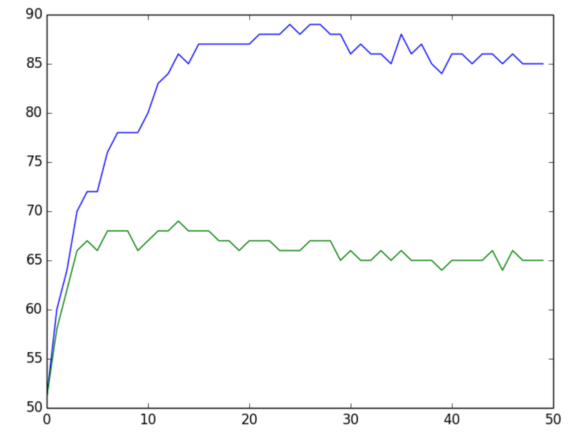
Learning Rate: 0.1



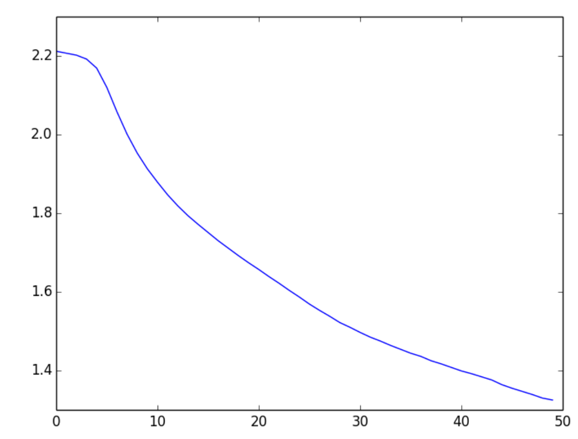


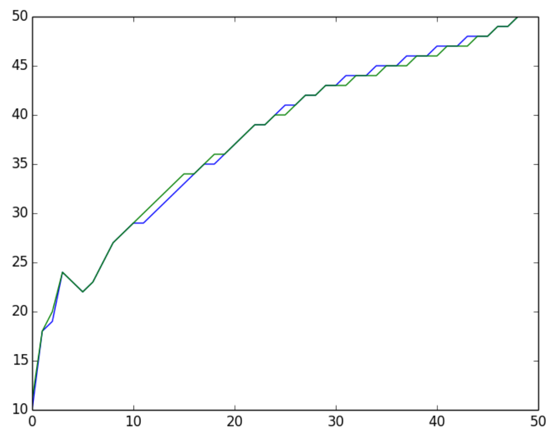
Learning Rate: 0.01





Learning Rate: 0.0001





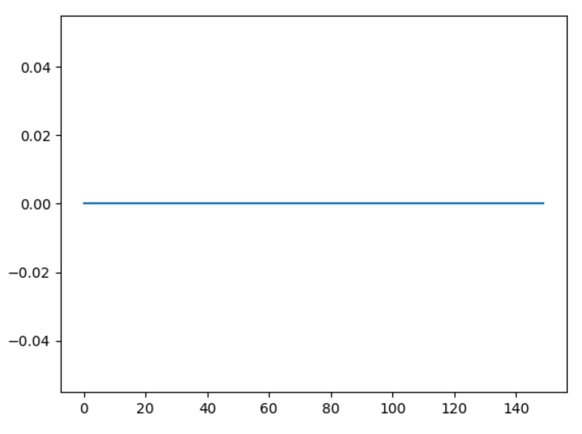
Here we can see that with a learning rate of 10, the rate is too high and the algorithm does not find a local maxima, hence the accuracy is always at 10. With a learning rate of 0.1 we also see that after a little bit the accuracy goes towards 10. Now with 0.1 the accuracy tends to climb and perform much better than 10 and 0.1, however it overfits quite a bit. Lastly, a learning rate of 0.0001 performs very similarly with train and test, however the accuracy is lower than 0.01.

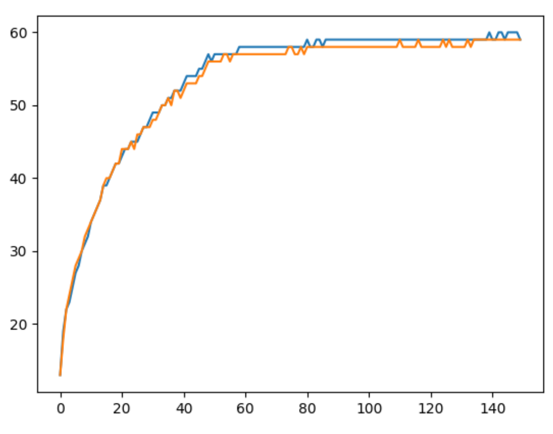
It looks like the best is either 0.01 or 0.0001. I chose 0.01 because even though it overfits, it does have a higher accuracy than the rest.

Adjusting Learning Rate

Now I adjusted the model to decrease the learning rate to 10% of the previous every epoch. Going from 1 → 0.1 → 0.01 and so forth.

I plot the results below





The network now seems to perform better than 0.0001 learning rate in the previous section and does not overfit like the 0.01 learning rate in the previous section did.

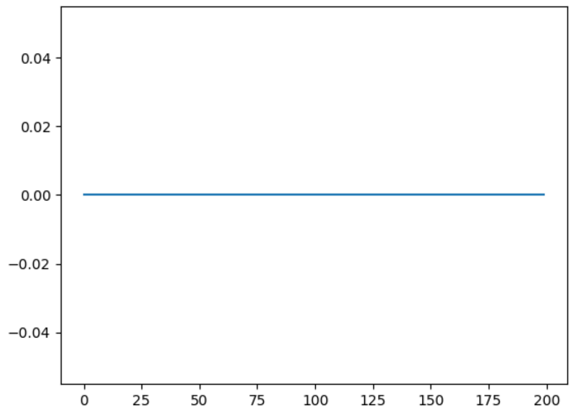
I keep using the adjusting learning rate method for the rest of this report.

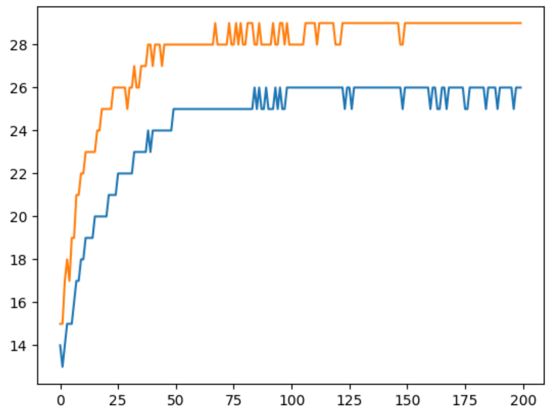
Data Augmentation

I augmented the data with the following transforms:

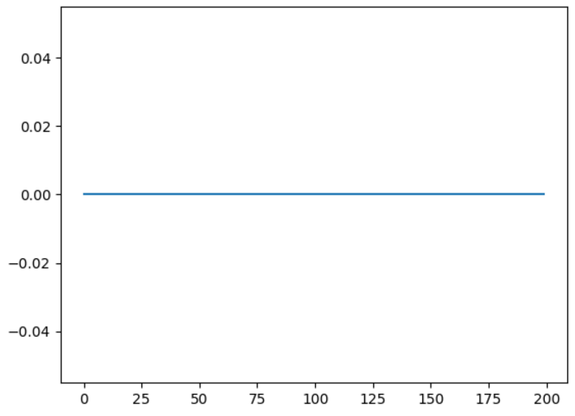
```
transforms.RandomAffine(degrees=90)
transforms.RandomVerticalFlip()
transforms.RandomHorizontalFlip()
transforms.ColorJitter(brightness=0.3, contrast=0.5, saturation=0.5, hue=0.5
)
```

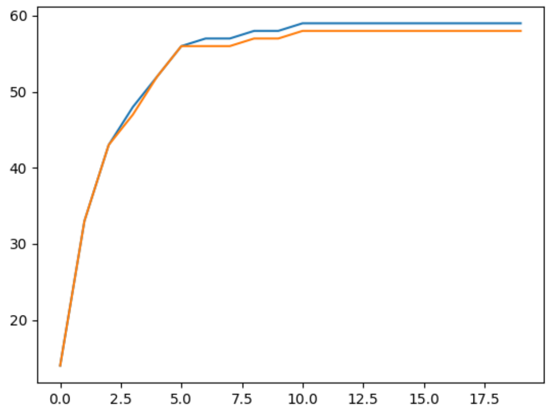
The results with data augmentation:





The results without data augmentation:

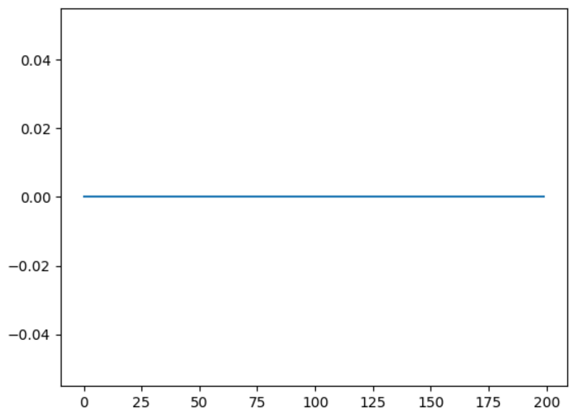


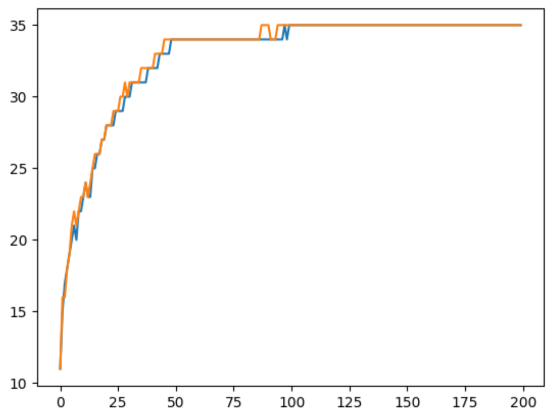


Using a batch size of 256, the results above show that the performance with data augmentation is significantly lower than the performance without data augmentation. This does go against my intuition, and I ran this various different times on different machines to confirm this was the case, but it looks like augmenting the data, maybe with the transformations I made, results in the network learning the wrong kind of classification and maybe relying too much on different features than previous networks, resulting in a lower train and test accuracy.

MSE Loss

In the last section, I adjusted the loss function to be MSE loss instead of Cross Entropy Loss and did not use data augmentation. I ran for 200 epochs, and my results are the following:





We can see that this network does perform much worse than the previous networks in this report. I suspect this is because MSE loss (Mean Squared Error) looks at the difference in the output vectors and tries to minimize it. However, each of the entries in the output vector are not related since the vector does not represent a number, but classes instead for classification. So this would tend to optimize with the wrong bias. Cross Entropy Loss instead is meant for classification and is much better suited for this problem, hence it works better.