Architecture 1

We developed this simple architecture by investigating other solutions and noticing a common pattern; Most models consisted of multiple convolutional layers (often with reparations of layers with the same structure, e.g multiple conv2 in this architecture), followed by pooling, followed by being passed to a few fully connected layer and finally the output layer. The neurons on the first fully connected layer correspond to one pixel (or more generally datapoint) of one channel of the output layer, and therefore the size of the input of the first connected layer is o * w * h, where o is the number of output channels in the previous (convolutional layer), w is the width of the image and h is its height. In this case you can see that conv4 (named incorrectly by the way, it is not a convolutional layer!) has 8192 input neurons which is equal to 128 * 8 * 8.

Code:

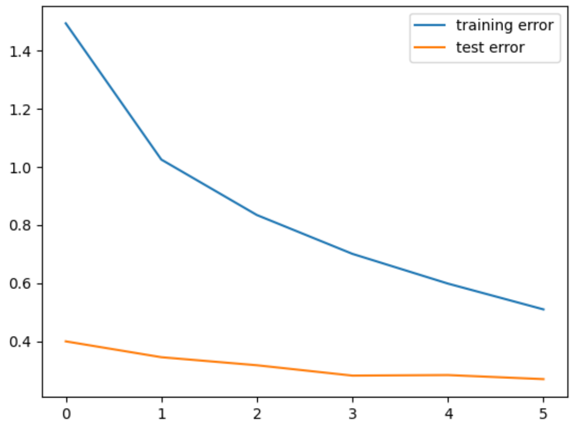
```
self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=(3, 3), padding=1)
self.conv2 = nn.Conv2d(in_channels=32, out_channels=32, kernel_size=(3, 3), padding=1)
self.pool = nn.MaxPool2d(2, 2)
self.conv3 = nn.Conv2d(in_channels=32, out_channels=128, kernel_size=(3, 3), padding=1)
self.conv4 = nn.Linear(8192, 128)
self.conv5 = nn.Linear(128, 64)
self.conv6 = nn.Linear(64, 10)
```

```
# Input: 32 x 32, 3 channels
x = self.conv1(x) # image unchanged, more channels; 32 x 32, 32 channels
x = F.relu(x)
x = self.conv2(x) # same as above, 32 x 32, 32 channels
x = F.relu(x)
x = self.conv2(x) # same as above, 32 x 32, 32 channels
x = F.relu(x)
x = self.pool(x) # half the image size, channels the same. 16 x 16, 32 channels
x = self.conv3(x) # same image size, more channels 16 x 16, 128 channels
x = F.relu(x)
x = self.pool(x) # half the image size, same channels. 8 x 8, 128 channels
x = torch.flatten(x, 1)  # flatten. number of neurons is 8 * 8 * 128 = 8192
x = self.conv4(x) # 8192 -> 128 layer, now 128 neurons.
x = F.relu(x)
x = self.conv5(x) # 128 -> 64 layer, now 64 neurons.
x = F.relu(x)
x = self.conv6(x) # 64 -> 10 layer, now 10 neurons. this is output
return x
```

Hyperparameters:
6 epochs, lr = 0.001, Adam

Tests:

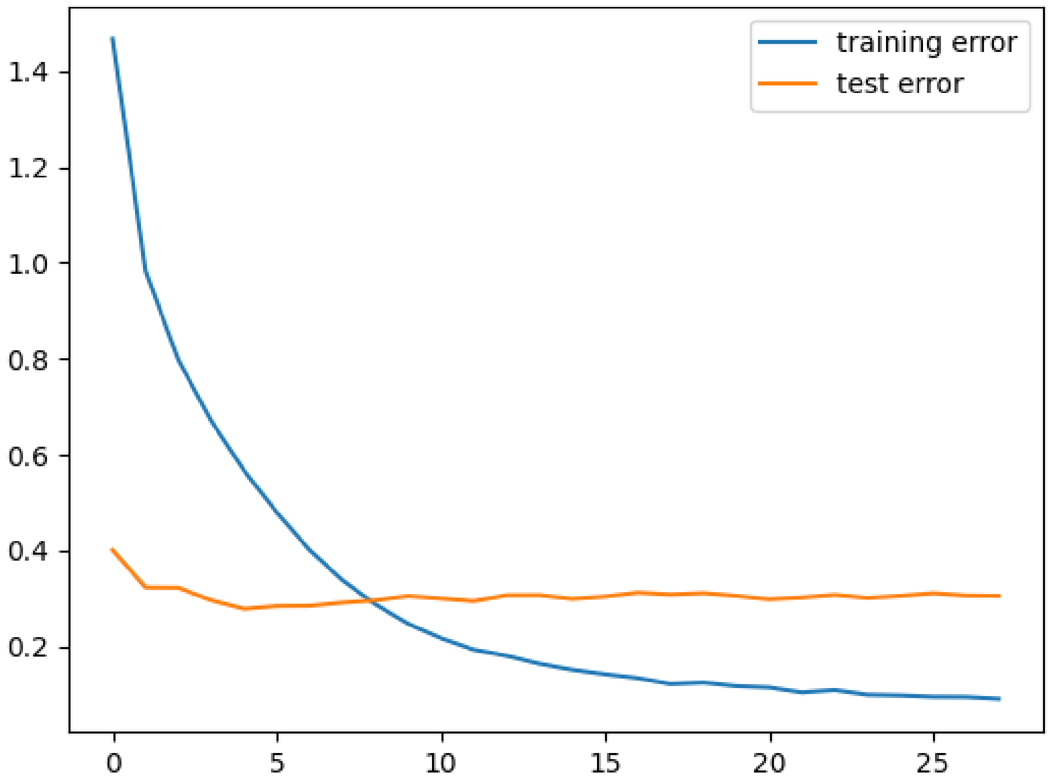
Test 1




```
<============================================================>
50000 images processed with training loss: 0.509
Network accuracy on 10000 test images: 73 %
test loss for epoch:0.269
epoch  6 3124  complete.
<============================================================>
```

Comments:
Strong algorithm that achieves low test error quickly, still possibly some room for lower test error without overfitting so will run again on more epochs

Test 2

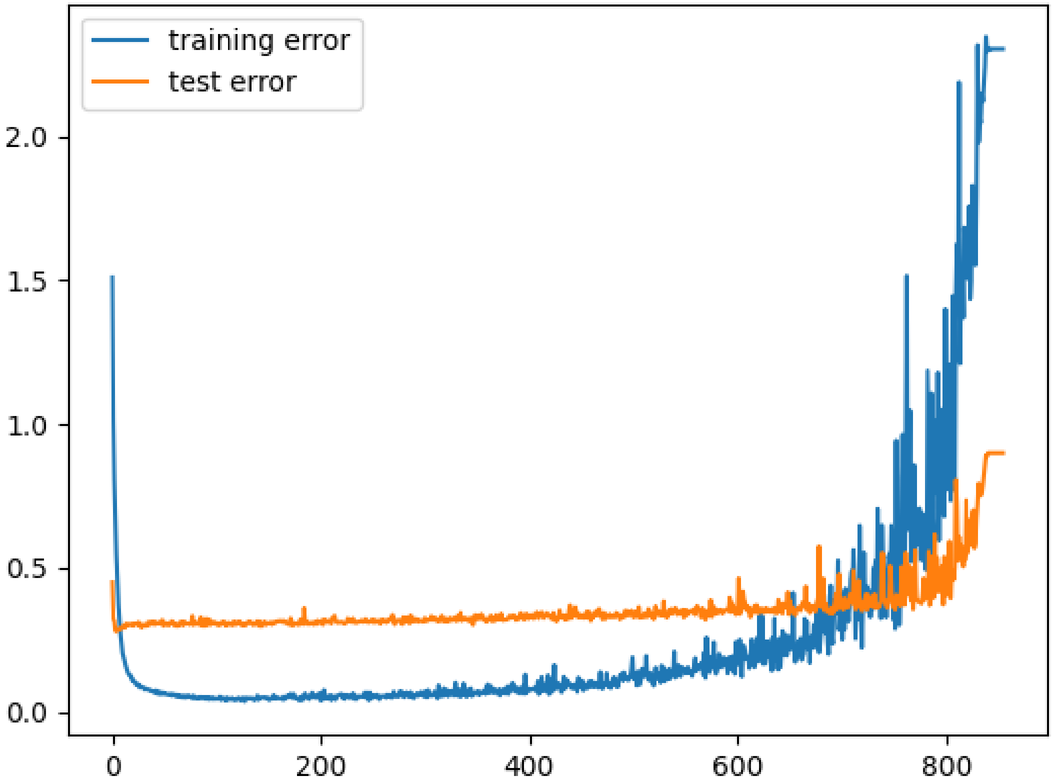



```
50000 images processed with training loss: 0.090
Network accuracy on 10000 test images: 69 %
test loss for epoch:0.305
epoch  28 3124  complete.
```

Comments:
Best success rate was epoch 5 with 72% accuracy on test images. It seems the model learnt *something* very quickly, but then started overfitting after around epoch 7 or so. This suggests we have probably reached the limits of the model, however we will try one more test with SGD with a very long runtime to be absolutely certain.

Test 3

```
<------------------------------------------------------------>
Finished Training
Best parameters were at epoch 5, With test error rate 0.2802. Saving these parameters to model1
```



Comments:

Oh dear, not sure what happened at the end there. The best test error was at 5, with an accuracy of 72%. This is not bad but clearly we need to change the model to reduce overfitting. The model very quickly converges to 0 and then starts overfitting, and eventually does… whatever on earth is going on at the end

3 I think you are approximate with this small learning rate so slowly to the local minimum that the point where the loss value slightly increases again (because you exceed the minimum) requires too many iterations. This increase in loss value is due to Adam, the moment the local minimum is exceeded and a certain number of iterations, a small number is divided by an even smaller number and the loss value explodes. – Freundlicher Jan 26, 2018 at 22:38

Possibly due to this; seems to be a limitation with adam. Worth commenting on in the report.
{}

Architecture 2

We renamed some layer names to be more consistent. The main addition here is the dropout which should in theory reduce the [extent to which our model overfits.]
Code:

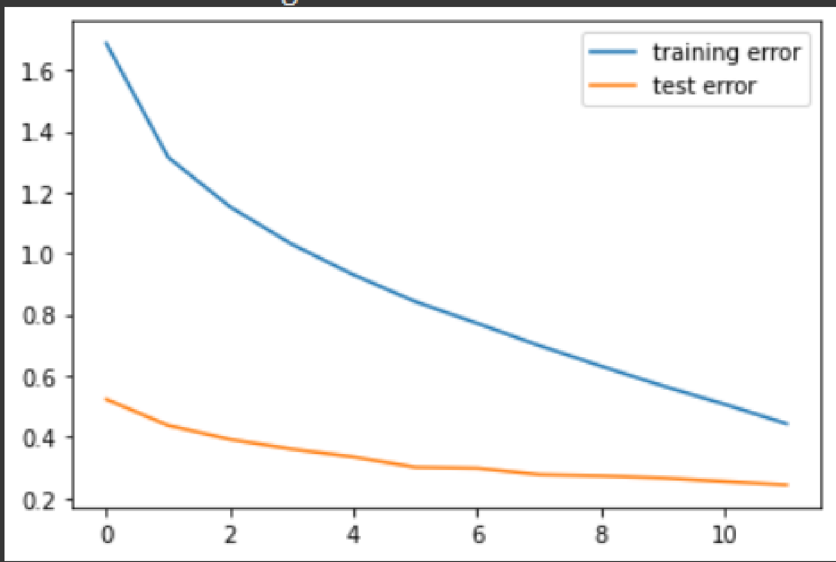
```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.layer2 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.layer3 = nn.Conv2d(in_channels=128,out_channels=256,kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.layer4 = nn.Conv2d(in_channels=256,out_channels=512,kernel_size=3,padding=1)
        self.layer5 = nn.Linear(512 * 8 * 8, 128)
        self.layer6 = nn.Linear(128, 64)
        self.layer7 = nn.Linear(64, 10)
        self.dropout = nn.Dropout(0.25)

    def forward(self, x):
        x = self.layer1(x)
        x = F.relu(x)
        x = self.pool(x)
        x = self.layer2(x)
        x = F.relu(x)
        x = self.pool(x)
        x = self.layer3(x)
        x = F.relu(x)
        x = self.layer4(x)
        x = F.relu(x)
        x = torch.flatten(x, 1)  # flatten all dimensions except batch
        x = self.layer5(x)
        x = F.relu(x)
        x = self.layer6(x)
        x = F.relu(x)
        x = self.dropout(x)
        x = self.layer7(x)
        return x
```

Hyperparameters:
lr = 0.0001, Adam

Tests:

Test 1

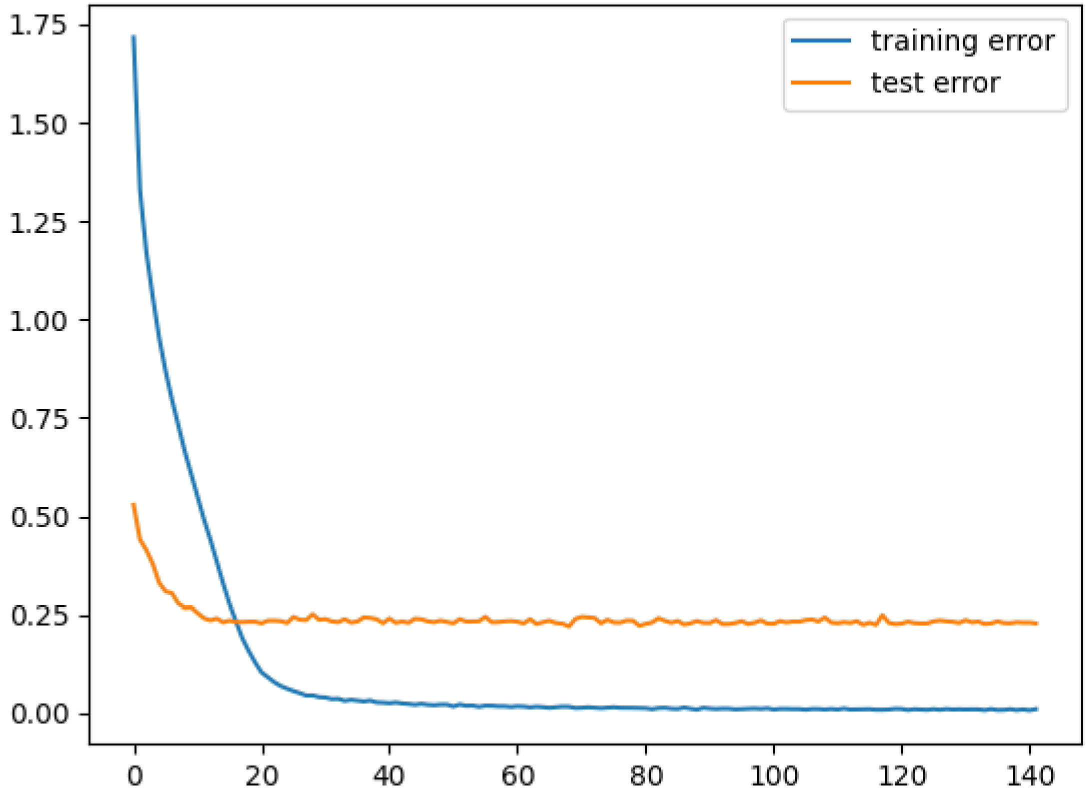
```
<=============================================================>
 3125 images processed with training loss: 0.443
Network accuracy on 10000 test images: 75 %
test loss for epoch:0.242
epoch  12 3124  complete.
<=============================================================>
Finished Training
```




```
Network accuracy on 10000 test images: 75 %
```

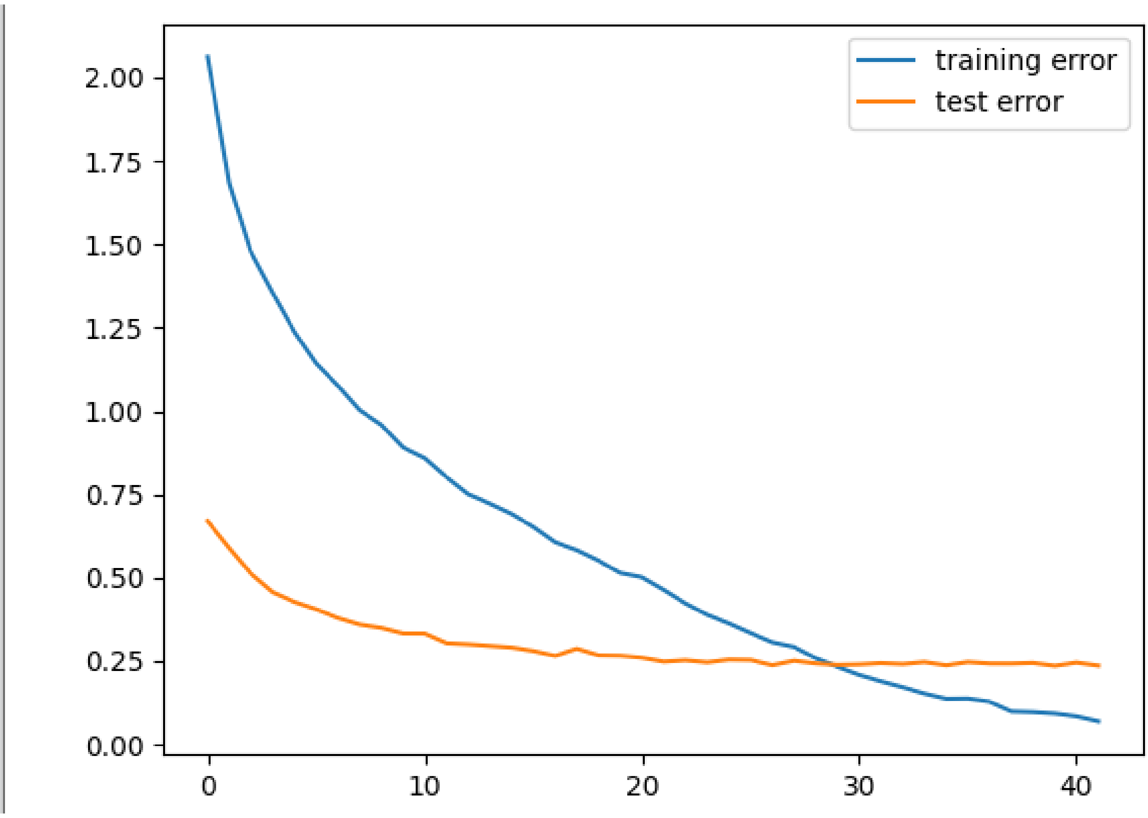
Comments: Using SGD as an optimizer breaks the model for a reason that is yet to be determined. 75% is now the current benchmark accuracy that we have. Worth testing further as it looks like there might be more room for lower test loss without overfitting. Will run again with more epochs.

Test 2



Comments:
This model clearly achieves a better accuracy but is still very prone to overfitting. Some more work is needed to increase test accuracy

Test 3



Increased batch size to 1024.

```
50176 images processed with training loss: 0.069
Network accuracy on 10000 test images: 76 %
test loss for epoch:0.236
epoch  42 48  complete.
<==========================================================>
Finished Training
Best parameters were at epoch 40, With test error rate 0.23519999999999996. Saving these parameters to model1
```

No real observable gain so we will go back to smaller batch sizes

Architecture 3

```
self.layer1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1)
self.pool = nn.MaxPool2d(2, 2)
self.layer2 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1)
self.pool = nn.MaxPool2d(2, 2)
self.layer3 = nn.Conv2d(in_channels=64,out_channels=128,kernel_size=3, padding=1)
self.pool = nn.MaxPool2d(2, 2)
self.layer4 = nn.Conv2d(in_channels=128,out_channels=128,kernel_size=3,padding=1)
self.layer5 = nn.Linear(8192, 128)
self.layer6 = nn.Linear(128, 64)
self.layer7 = nn.Linear(64, 10)
self.dropout = nn.Dropout(0.25)
```

```
def forward(self, x):
    x = self.layer1(x)
    x = F.relu(x)
    x = self.pool(x)
    x = self.layer2(x)
    x = F.relu(x)
    x = self.layer2(x)
    x = F.relu(x)
    x = self.pool(x)
    x = self.layer3(x)
    x = F.relu(x)
    x = self.layer4(x)
    x = F.relu(x)
    x = self.layer4(x)
    x = F.relu(x)
    x = torch.flatten(x, 1)  # flatten all dimensions except batch
    x = self.layer5(x)
    x = F.relu(x)
    x = self.layer6(x)
    x = F.relu(x)
    x = self.dropout(x)
    x = self.layer7(x)
    return x
```

Test 1

```
50000 images processed with training loss: 0.057
Network accuracy on 10000 test images: 76 %
test loss for epoch:0.239
epoch  56 3124  complete.
<===========================================================>
Time elapsed: 23.415503/40.000000 (minutes)
<===========================================================>
50000 images processed with training loss: 0.056
```

Converged much slower, comparable but slightly higher accuracy. This model is likely a step in the right direction, will try adding more layers.

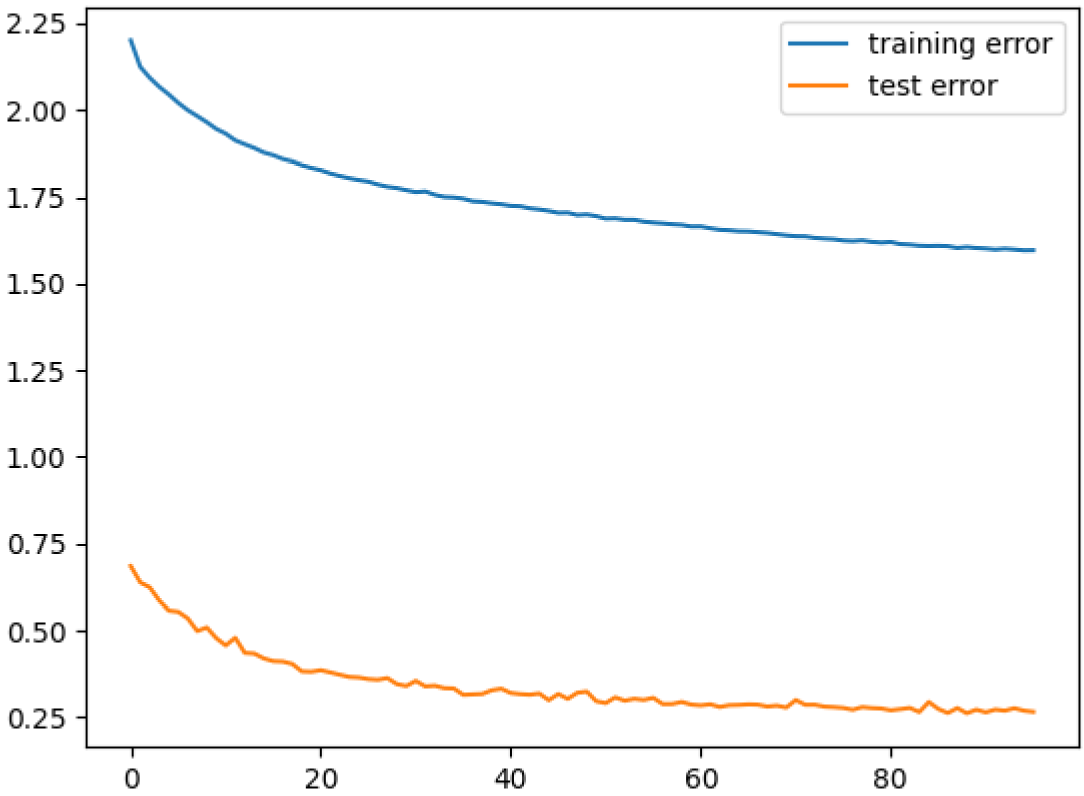
Architecture 4

Changes - increased dropout, added softmax to final layer

```
self.layer1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1)
self.pool = nn.MaxPool2d(2, 2)
self.layer2 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1)
self.pool = nn.MaxPool2d(2, 2)
self.layer3 = nn.Conv2d(in_channels=64,out_channels=128,kernel_size=3, padding=1)
self.pool = nn.MaxPool2d(2, 2)
self.layer4 = nn.Conv2d(in_channels=128,out_channels=128,kernel_size=3,padding=1)
self.layer5 = nn.Linear(8192, 128)
self.layer6 = nn.Linear(128, 64)
self.layer7 = nn.Linear(64, 10)
self.dropout = nn.Dropout(0.5)
self.softmax = nn.Softmax()
```

```
x = self.layer1(x)
x = F.relu(x)
x = self.pool(x)
x = self.layer2(x)
x = F.relu(x)
x = self.layer2(x)
x = F.relu(x)
x = self.pool(x)
x = self.layer3(x)
x = F.relu(x)
x = self.layer4(x)
x = F.relu(x)
x = self.layer4(x)
x = F.relu(x)
x = torch.flatten(x, 1)  # flatten all dimensions except batch
x = self.layer5(x)
x = F.relu(x)
x = self.layer6(x)
x = F.relu(x)
x = self.dropout(x)
x = self.layer7(x)
x = self.softmax(x) # Use softmap to represent output layer as a probability distribution.
return x
```

Test 1




```
50000 images processed with training loss: 1.596
Network accuracy on 10000 test images: 73 %
test loss for epoch:0.266
epoch  96 3124  complete.
<===========================================================>
Finished Training
Best parameters were at epoch 89, With test error rate 0.26160000000000005. Saving these parameters to model1
```

```
.CrossEntropyLoss(
.Adam(lr=0.0001, p
```

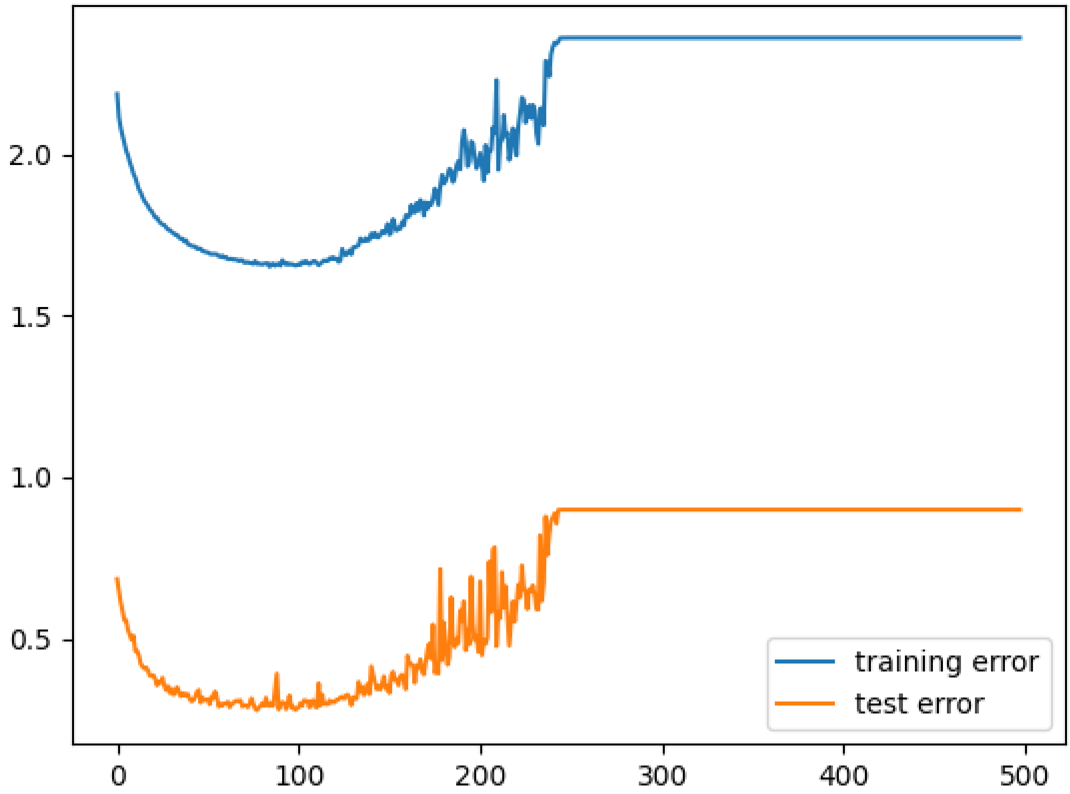
Appears that learning rate is too low; training error is not decreasing further. Very strong model though; achieved 73% accuracy with 1.596 training error. I will try increasing the learning rate.

Test 2

Aborted. Increasing learning rate further seems to result in no training. It is likely that gradients are just jumping around and not reaching any minimum. Next we will try increasing the training rate while keeping learning rate at 0.0001. Ideally we would like a larger learning rate but this seems to not be possible, so we are forced to wait for longer and see what happens.

Test 3

Ran for 5hours, changed batch size to 8




```
Finished Training
Best parameters were at epoch 78, With test error rate 0.2782. Saving these parameters to model1
```

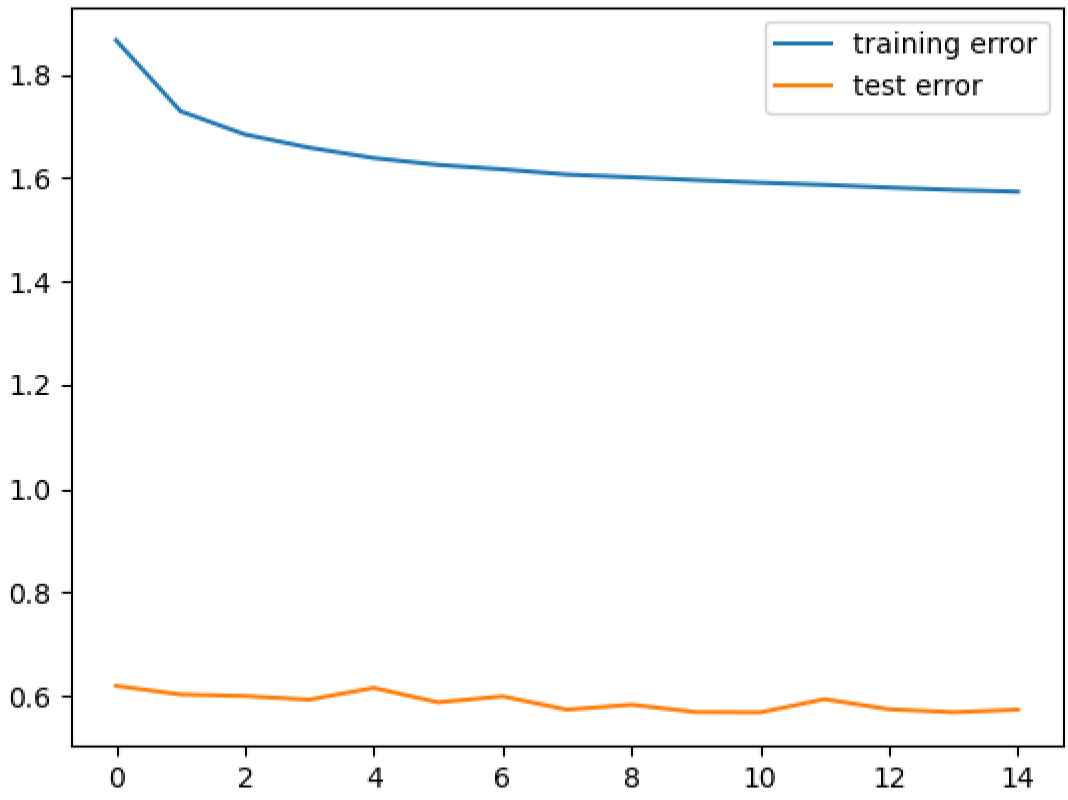
Comments: something is very clearly wrong here :(
I'm going to try going back to basics by setting up a very simple non-convolutional network and building from that.

Architecture 5

```
self.layer5 = nn.Linear(3072, 128)
self.layer6 = nn.Linear(128, 64)
self.layer7 = nn.Linear(64, 10)
```

```python
def forward(self, x):
    x = torch.flatten(x, 1)
    x = self.layer5(x)
    x = F.relu(x)
    x = self.layer6(x)
    x = F.relu(x)
    x = self.layer7(x)
    return x
```

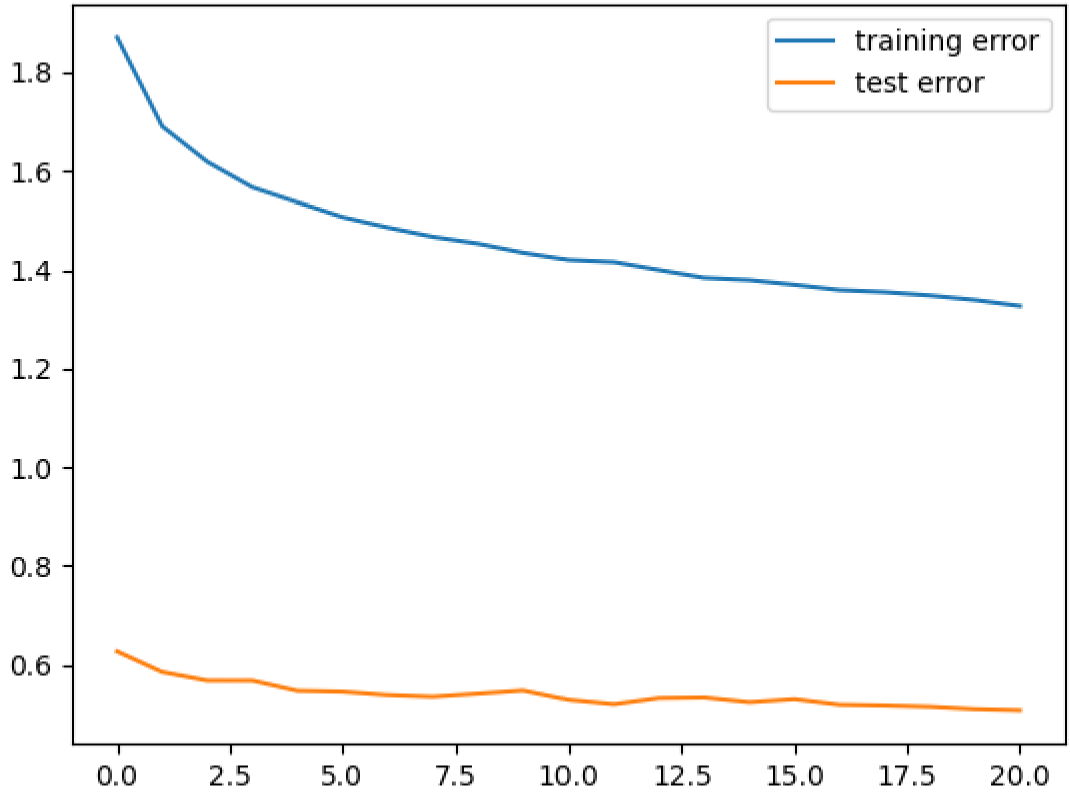
Test 1




```
Finished Training
Best parameters were at epoch 11, With test error rate 0.5686. Saving these parameters to model1
```

Doesn’t seem to have converged yet, I will try increasing learning rate significantly.

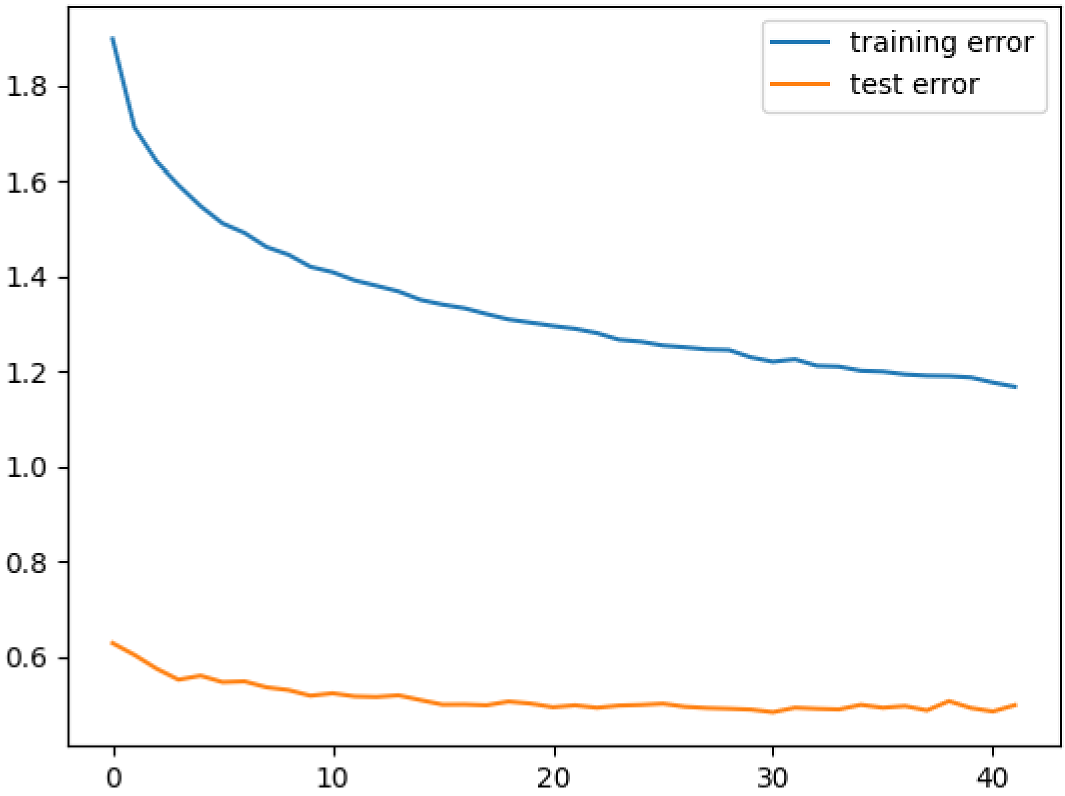
Test 2




```
50048 images processed with training loss: 1.327
Network accuracy on 10000 test images: 49 %
test loss for epoch:0.508
New record for test error.
epoch  21 781  complete.
<============================================================>
Finished Training
Best parameters were at epoch 21, With test error rate 0.5079. Saving these parameters to model1
```

Still some potential it seems. This is at a higher batch size of 64. I'm going to try increasing the batch size again and running for longer.

Test 3




```
<==========================================================>
50048 images processed with training loss: 1.168
Network accuracy on 10000 test images: 50 %
test loss for epoch:0.500
epoch  42 390  complete.
<==========================================================>
Finished Training
Best parameters were at epoch 31, With test error rate 0.4847. Saving these parameters to model1
```

This doesn't seem to be going anywhere. I am going back to architecture 2 and tweaking this to see if we can get better results.

Architecture 6

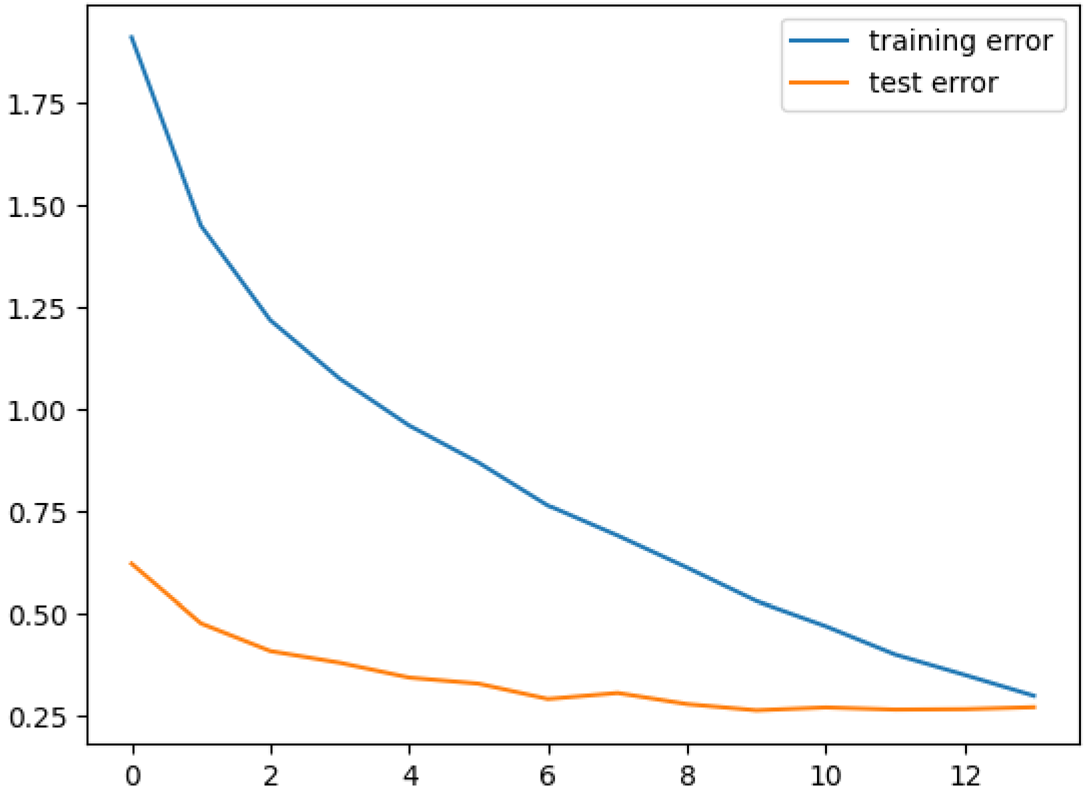
Clone of architecture 2

```
self.layer1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1)
self.pool = nn.MaxPool2d(2, 2)
self.layer2 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
self.layer3 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1)
self.layer4 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, padding=1)
self.layer5 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, padding=1)
self.layer6 = nn.Linear(8192, 128)
self.layer7 = nn.Linear(128, 64)
self.layer8 = nn.Linear(64, 10)
self.dropout = nn.Dropout(0.15)
```

```
x = self.layer1(x)
x = F.relu(x)
x = self.pool(x)
x = self.layer2(x)
x = F.relu(x)
x = self.pool(x)
x = self.layer3(x)
x = F.relu(x)
x = self.layer4(x)
x = F.relu(x)
x = self.layer5(x)
x = F.relu(x)
x = self.layer5(x)
x = F.relu(x)
x = self.layer5(x)
x = F.relu(x)
x = self.pool(x)
x = torch.flatten(x, 1)  # flatten all dimensions except batch
x = self.dropout(x)
x = self.layer6(x)
x = F.relu(x)
x = self.layer7(x)
x = F.relu(x)
x = self.dropout(x)
```

```
x = self.layer8(x)
return x
```

Test 1




```
50048 images processed with training loss: 0.299
Network accuracy on 10000 test images: 72 %
test loss for epoch:0.271
epoch  14 390  complete.
<==========================================================>
Finished Training
Best parameters were at epoch 10, With test error rate 0.26380000000000003. Saving these parameters to model1
```

Pretty good! 72% with seemingly no overfitting.

Architecture 7

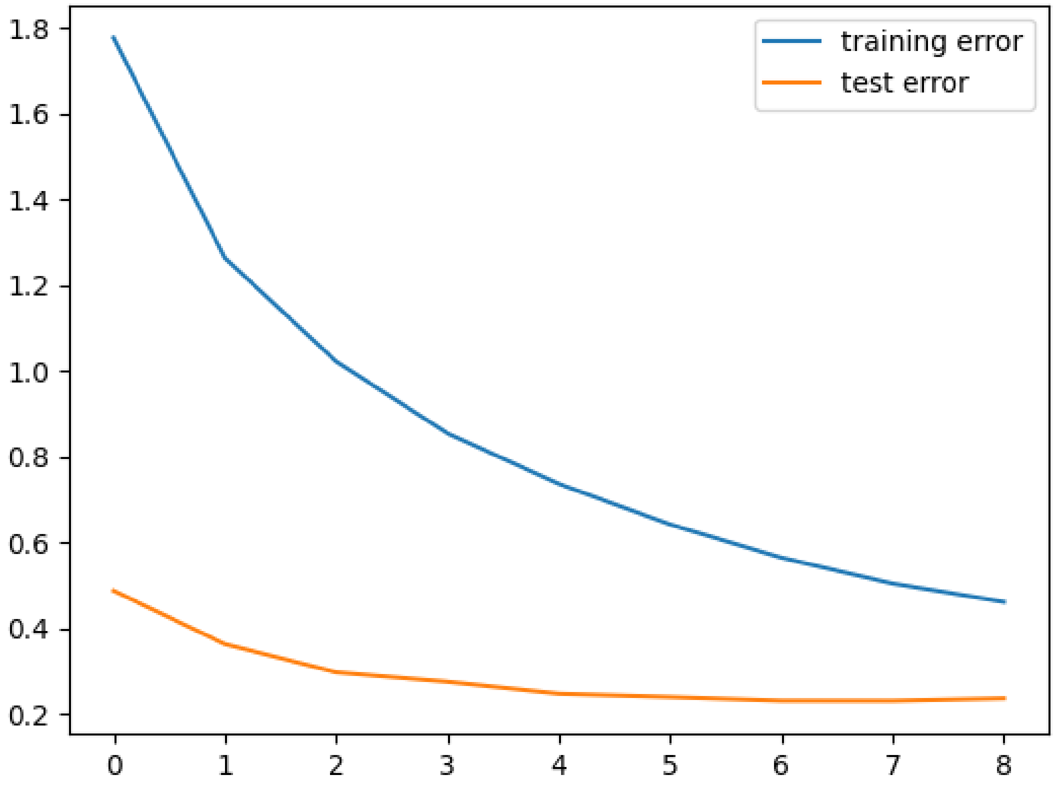
Clone of architecture 6, but now we are using augmentation.

```
augment = transforms.Compose(
    [transforms.AutoAugment(transforms.AutoAugmentPolicy.CIFAR10), transforms.ToTensor(), transforms.Normalize(0, 1)])
```

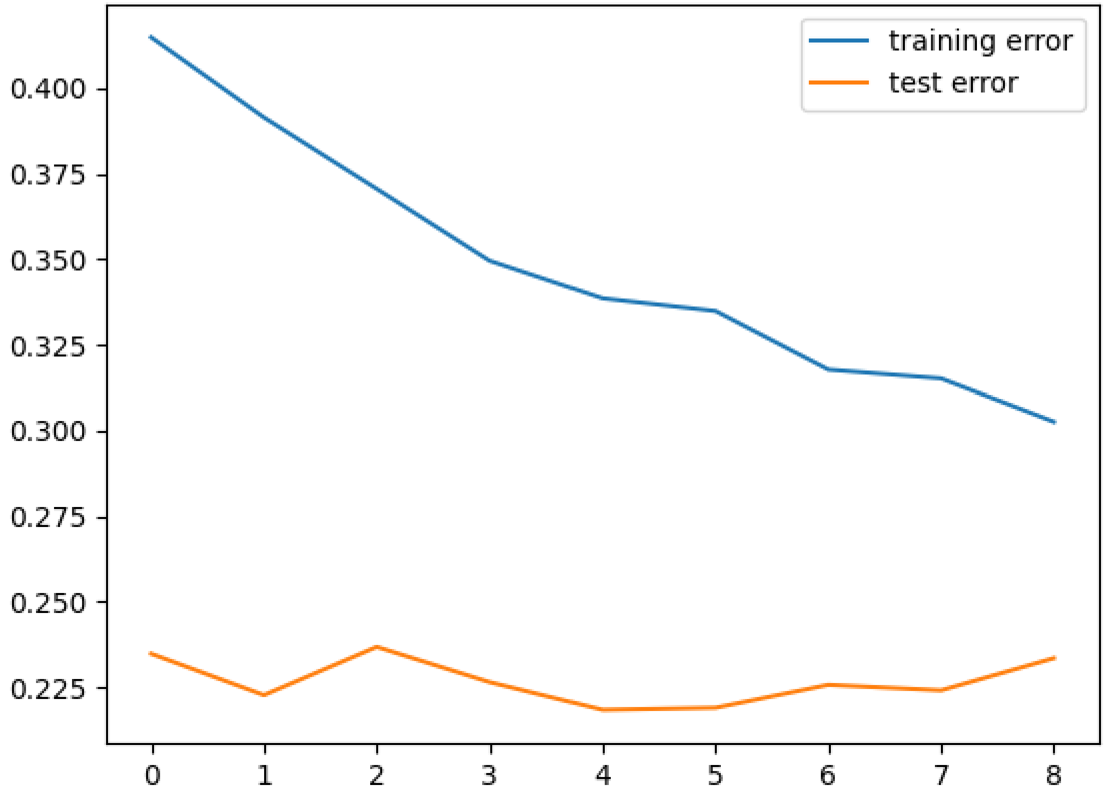
```
training_data = torchvision.datasets.CIFAR10(root='./data',
                                             train=True,
                                             download=not already_downloaded,
                                             transform=transform)


augmented_data = torchvision.datasets.CIFAR10(root='./data',
                                              train=True,
                                              download=not already_downloaded,
                                              transform=augment)


training_data = torch.utils.data.ConcatDataset([training_data, augmented_data])
```

Test 1

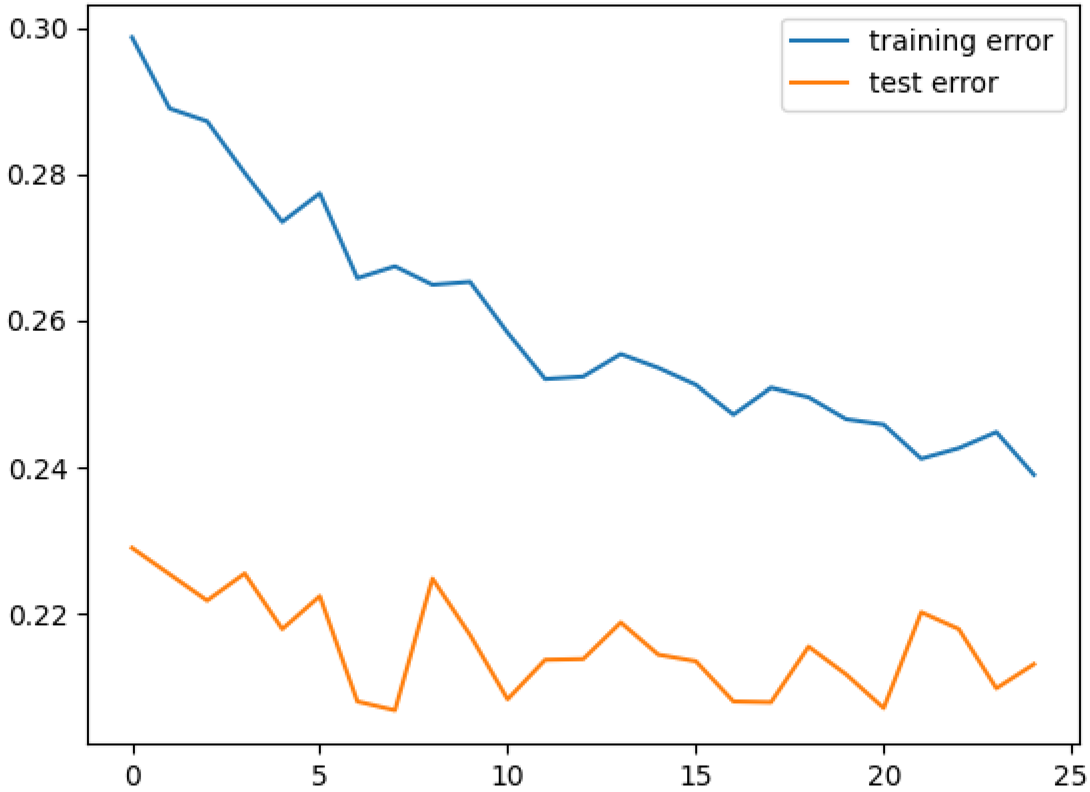
The training was done in steps; two steps of 5 minutes, then one step of 15 minutes. That is why there are 3 graphs




```
100096 images processed with training loss: 0.462
Network accuracy on 10000 test images: 76 %
test loss for epoch:0.238
epoch  9 781  complete.
<==========================================================>
Finished Training
Best parameters were at epoch 8, With test error rate 0.23219999999999996. Saving these parameters to model1
```




```
100096 images processed with training loss: 0.303
Network accuracy on 10000 test images: 76 %
test loss for epoch:0.234
epoch  9 781  complete.
<=========================================================>
Finished Training
Best parameters were at epoch 5, With test error rate 0.21840000000000004. Saving these parameters to model1
```




```
100096 images processed with training loss: 0.239
Network accuracy on 10000 test images: 78 %
test loss for epoch:0.213
epoch  25 781  complete.
<=========================================================>
Finished Training
Best parameters were at epoch 8, With test error rate 0.20679999999999998. Saving these parameters to model1
```

So at the best epoch this model achieved 79%. Not bad!

Test 2 - learning rate tuning

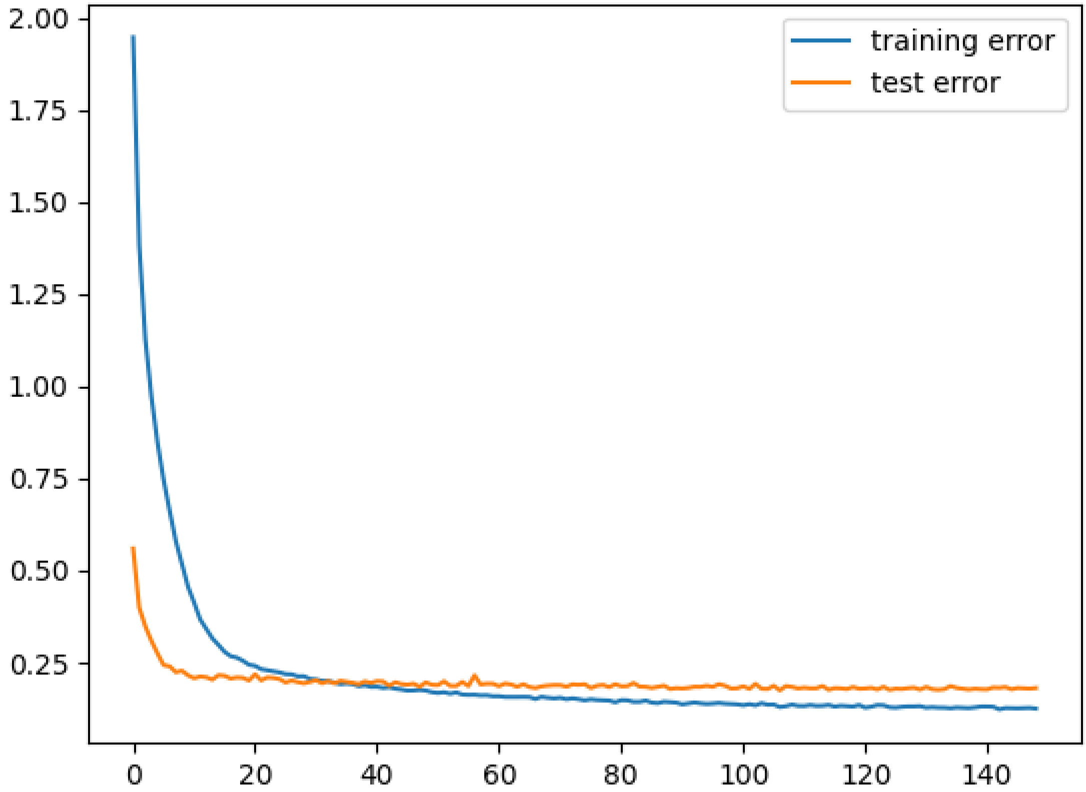
I developed a method that tunes the learning rate for a model by training the model with each learning rate and returning the optimal parameters (by lowest test rate), along with a list of results for each learning rate.

```
Optimization complete. The optimal learning rate was 0.0005, with a test error of 0.17910000000000004, at an optimal epoch of 75
Saving the best model to model2
[0.9, 0.9, 0.3991, 0.22089999999999999, 0.17910000000000004, 0.19879999999999998]
```

So with learning rate tuning we learnt that the best learning rate for this model is 0.0005, with which we managed to eke out another 4% accuracy! Not bad. I will verify this result with another test on just that learning rate and allow the model to run for a bit longer, say 90 minutes.

This test also tells us that high learning rates do not work well with this model for some reason. I wonder if this is potentially a problem with the loss function or optimizer.

Test 3




```
Finished Training
Best parameters were at epoch 107, With test error rate 0.1745.
Saving these parameters to model1
```

This gives us an accuracy of 82%. Seems that our hyperparameter tuning works!
This is a very good result, but it is clear that the model is beginning to over fit after around epoch 20 or so. It is clear we will need to make further improvements to the model

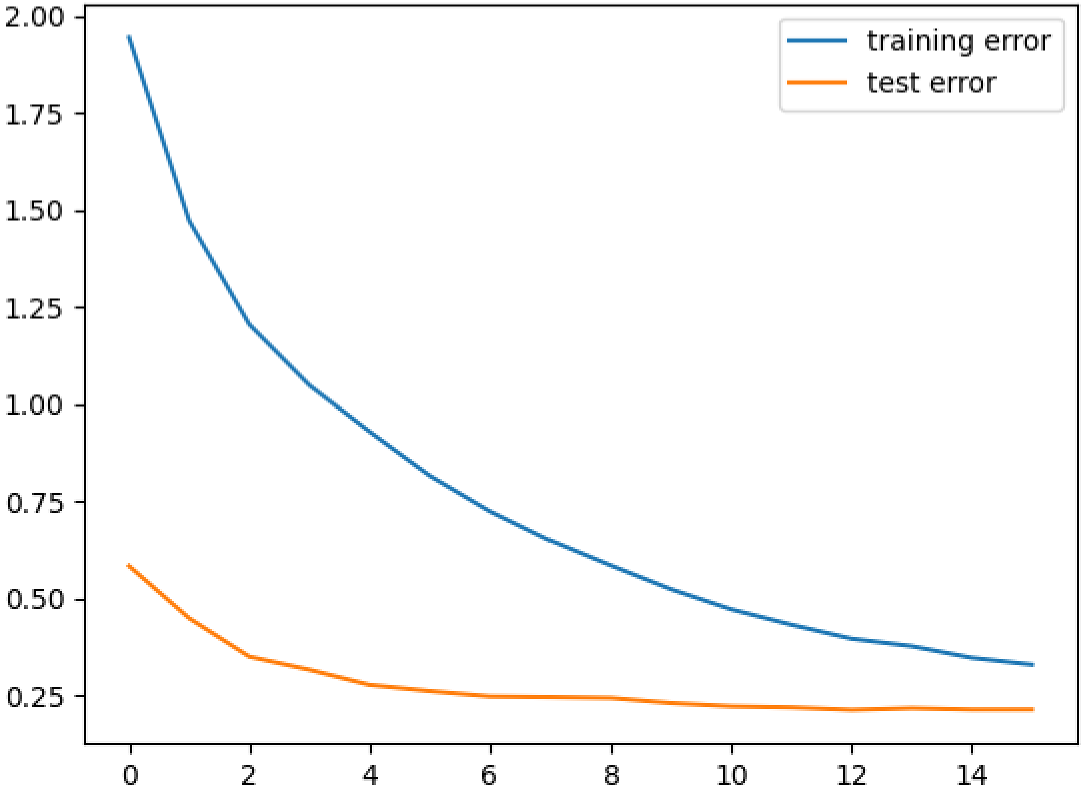
Test 4

Trying increasing dropout to 0.5 before the output layer:

```
self.dropout = nn.Dropout(0.5)
```

The main problem our model has now is that it starts overfitting quickly, after about 20 epochs. I have a suspicion that this might help with the overfitting and thus allow the model to achieve a higher accuracy without overfitting, so I am going to test this idea.

```
<==========================================================>
Finished Training
Best parameters were at epoch 13, With test error rate 0.2138.
Saving these parameters to model1
```



No significant difference. Time to move onto a new architecture

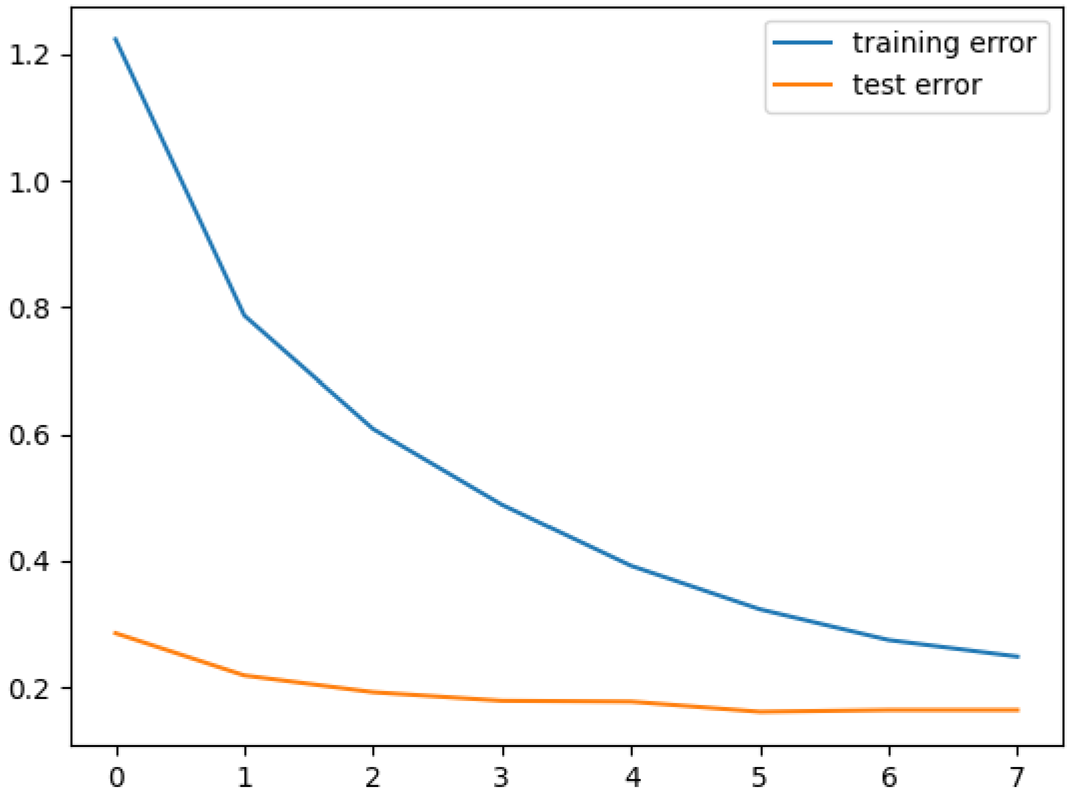
Architecture 8

```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.layer2 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.layer3 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1)
        self.layer4 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, padding=1)
        self.layer5 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, padding=1)
        self.layer6 = nn.Linear(8192, 128)
        self.layer7 = nn.Linear(128, 64)
        self.layer8 = nn.Linear(64, 10)
        self.dropout = nn.Dropout(0.15)
        self.batchnorm1 = nn.BatchNorm2d(64)
        self.batchnorm2 = nn.BatchNorm2d(128)
        self.batchnorm3 = nn.BatchNorm2d(256)
        self.batchnorm4 = nn.BatchNorm2d(512)
        self.batchnorm5 = nn.BatchNorm2d(512)
    def forward(self, x):
        x = self.layer1(x)
        x = F.relu(x)
        x = self.batchnorm1(x)
        x = self.pool(x)
        x = self.dropout(x)
        x = self.layer2(x)
        x = F.relu(x)
        x = self.batchnorm2(x)
        x = self.pool(x)
        x = self.dropout(x)
        x = self.layer3(x)
        x = F.relu(x)
        x = self.batchnorm3(x)
        x = self.layer4(x)
        x = F.relu(x)
        x = self.batchnorm4(x)
        x = self.layer5(x)
        x = F.relu(x)
        x = self.batchnorm5(x)
        x = self.layer5(x)
        x = F.relu(x)
        x = self.batchnorm5(x)
        x = self.layer5(x)
        x = F.relu(x)
        x = self.batchnorm5(x)
        x = self.pool(x)
        x = torch.flatten(x, 1)  # flatten all dimensions except batch
        x = self.layer6(x)
        x = F.relu(x)
        x = self.layer7(x)
        x = F.relu(x)
        x = self.layer8(x)
        return x
```

This model builds on architecture 7 and adds [batch normalisation] in between layers and dropout has been moved towards the beginning of the net. {}

Test 1

```
<===========================================================>
100096 images processed with training loss: 0.249
Network accuracy on 10000 test images: 83.580 %
test loss for epoch:0.164
epoch  8 781  complete.
<===========================================================>
Finished Training
Best parameters were at epoch 6, With test error rate 0.1614.
Saving these parameters to model1
```




```
<===========================================================>
Finished Training
Best parameters were at epoch 6, With test error rate 0.1614.
Saving these parameters to model1
```

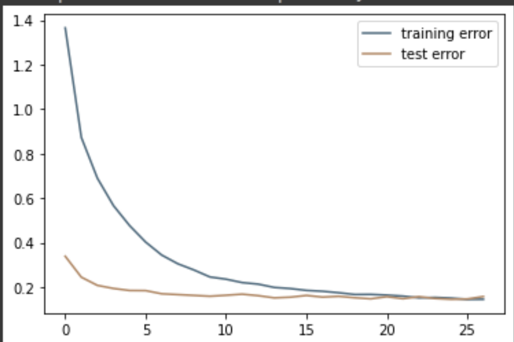

Very impressive! 84% accuracy in just 6 epochs. We will try running the model a bit longer and check the results. Batchnorm really seems to help performance/speed at which the model converges. {write about this in the report}

Test 2

Test 1

Batch size 128
Optimizer: Adam
Learning Rate: 0.001

```
Finished Training
Best parameters were at epoch 25, With test error rate 0.1452. Saving these parameters to model1
```




```
Network accuracy on 10000 test images: 84.76 %
```

```
<===========================================================>
100096 images processed with training loss: 0.151
Network accuracy on 10000 test images: 85.48 %
test loss for epoch:0.145
New record for test error.
epoch  25 781  complete.
<===========================================================>
```

Comments:

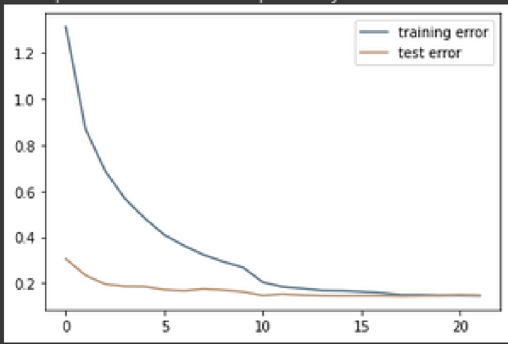
Started to overfit right at the end. New benchmark of 85% nearing 86. Overfitting seems to begin after about epoch 6, so the test error converges very quickly.

I am now going to optimize the learning rate for this architecture and test further.

Test 3

Batch size 64
Optimizer: Adam
Learning Rate:0.001 first 9 epochs, 0.0005 after

```
Finished Training
Best parameters were at epoch 18, with test error rate 0.14429999999999998. Saving these parameters to model1
```




```
Network accuracy on 10000 test images: 85.23 %
```

```
100032 images processed with training loss: 0.150
Network accuracy on 10000 test images: 85.57 %
test loss for epoch:0.144
New record for test error.
epoch  18 1562  complete.
<==================================================>
```

Added a dropout after the 3rd conv layer. A very marginal increase in accuracy and a very slight reduction in overfitting. Another dropout or even an increase in dropout rate may be in consideration for future tests.

Test 4

Same architecture as test 2. Now we are optimising the learning rate to push this model to its limits.

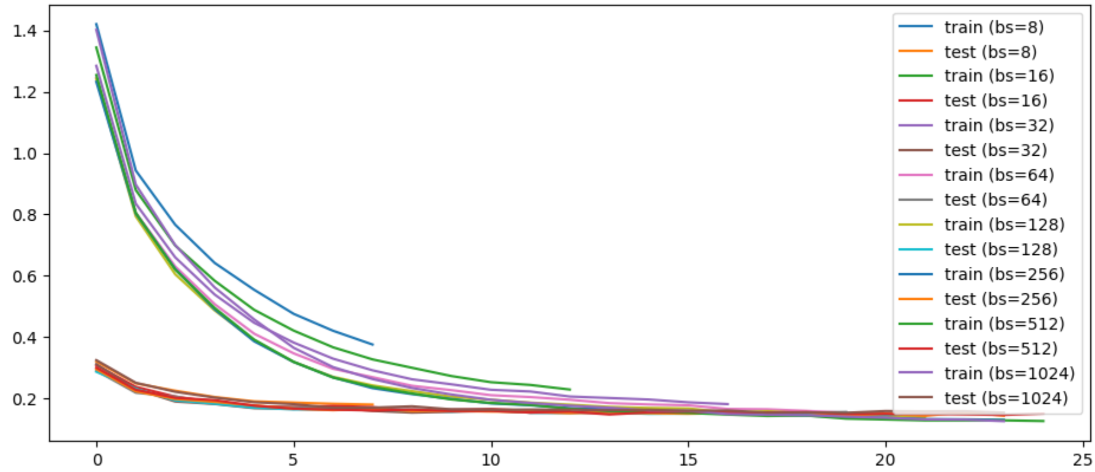
```
Optimization complete. The optimal learning rate was 0.0005, with a test error of 0.13970000000000005, at an optimal epoch of 26
Saving the best model to model2
[0.15469999999999995, 0.14339999999999997, 0.13970000000000005, 0.15759999999999996, 0.17959999999999998, 0.21819999999999995]
```

```
100096 images processed with training loss: 0.130
Network accuracy on 10000 test images: 86.030 %
test loss for epoch:0.140
New record for test error.
epoch  26 781  complete.
<============================================================>
Time elapsed: 17.035562/20.000000 (minutes)
<============================================================>
```

So this confirms that 0.0005 is the optimal learning rate for this model. I am now going to try optimising other hyperparameters to see if we can get any further increases in performance.

Test 5 - Batch optimization

Testing optimizing the batch size. Same method as learning rate tuning. I also added graphs for learning rate tuning and batch tuning, of the form below:

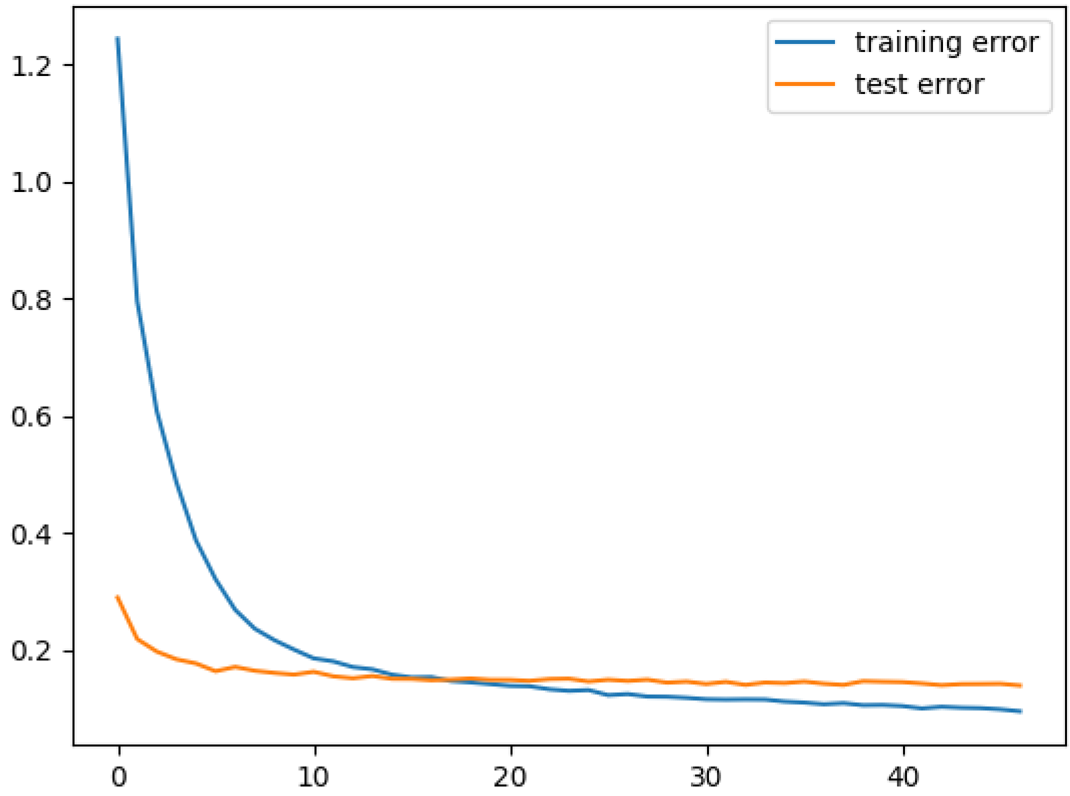



```
----------------------------------
Optimization complete. The optimal batch size was 256, with a test error of 0.1432, at an optimal epoch of 22
Saving the best model to model2
[0.18010000000000004, 0.15739999999999998, 0.15149999999999997, 0.1523, 0.14670000000000005, 0.1432, 0.1462, 0.1533]
```

The best batch size appears to be 256. I will now run this model again to verify this result.

Test 6

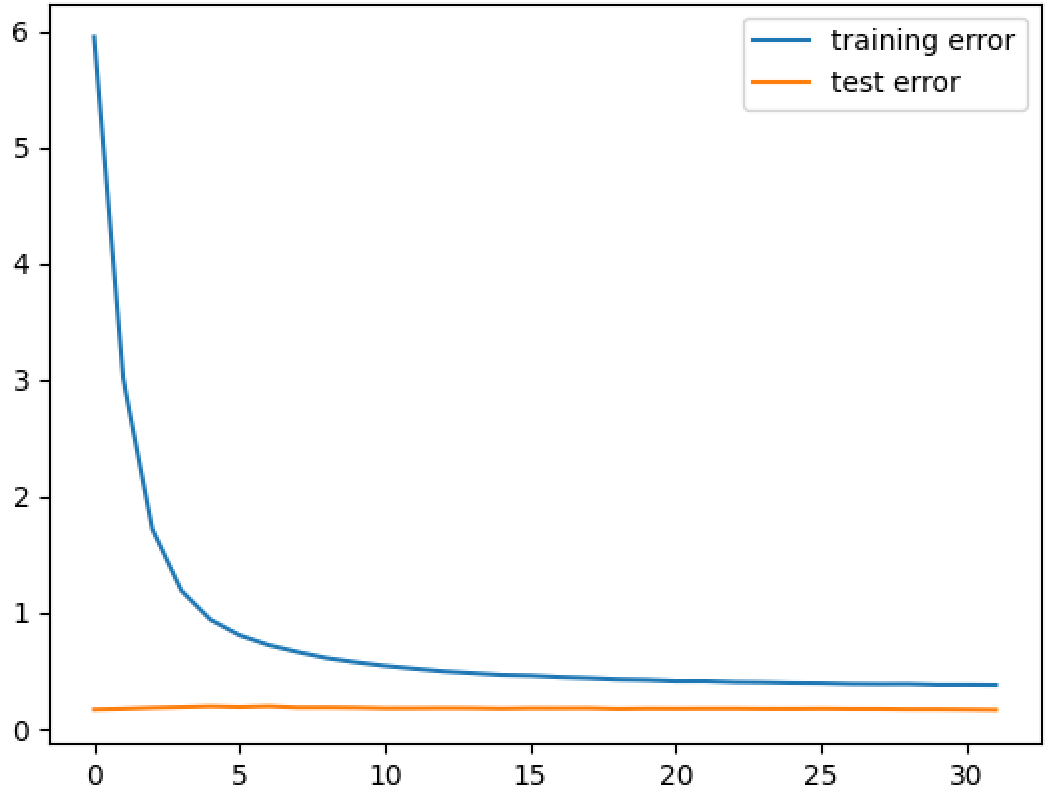
```
<=========================================================>
100096 images processed with training loss: 0.095
Network accuracy on 10000 test images: 86.070 %
test loss for epoch:0.139
New record for test error.
epoch  47 390  complete.
<=========================================================>
Finished Training
Best parameters were at epoch 47, With test error rate 0.13929999999999998.
Saving these parameters to model1
```



Appears to work now.
That seems to be the limit of what we can optimise with this model. Time to make more changes.

Test 7

Went back to this architecture to test SGD.




```
100096 images processed with training loss: 0.379
Network accuracy on 10000 test images: 83.360 %
test loss for epoch:0.166
New record for test error.
epoch  32 390  complete.
<============================================================>
Finished Training
Best parameters were at epoch 32, With test error rate 0.1664.
Saving these parameters to model1
```

No real advantage it seems.

Architecture 9

My plan now is to try a few different small changes, which I will keep if the test error rate is comparable or goes up, and discard those changes if it goes down. I will be training these models on the most optimal parameters for architecture 8, which is a learning rate of 0.0005 and a batch size of 256. The first I will try is adding some repeated 64 -> 64 channel convolution layers after the input layer. This is a common pattern I have noticed in popular image learning algorithms so it might be applicable here. {}

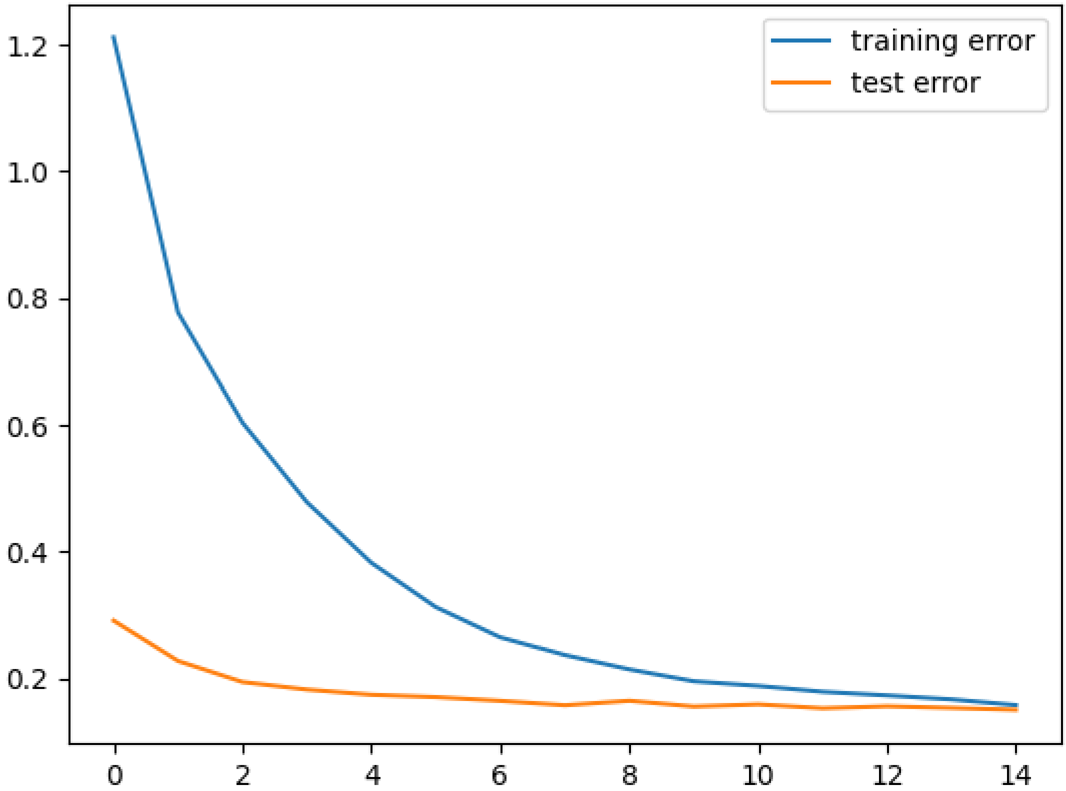
```python
super().__init__()
self.layer1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1)
self.layer1_2 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1)
self.pool = nn.MaxPool2d(2, 2)
self.layer2 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
self.layer3 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1)
self.layer4 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, padding=1)
self.layer5 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, padding=1)
self.layer6 = nn.Linear(8192, 128)
self.layer7 = nn.Linear(128, 64)
self.layer8 = nn.Linear(64, 10)
self.dropout = nn.Dropout(0.15)
self.batchnorm1 = nn.BatchNorm2d(64)
self.batchnorm2 = nn.BatchNorm2d(128)
self.batchnorm3 = nn.BatchNorm2d(256)
self.batchnorm4 = nn.BatchNorm2d(512)
self.batchnorm5 = nn.BatchNorm2d(512)
```

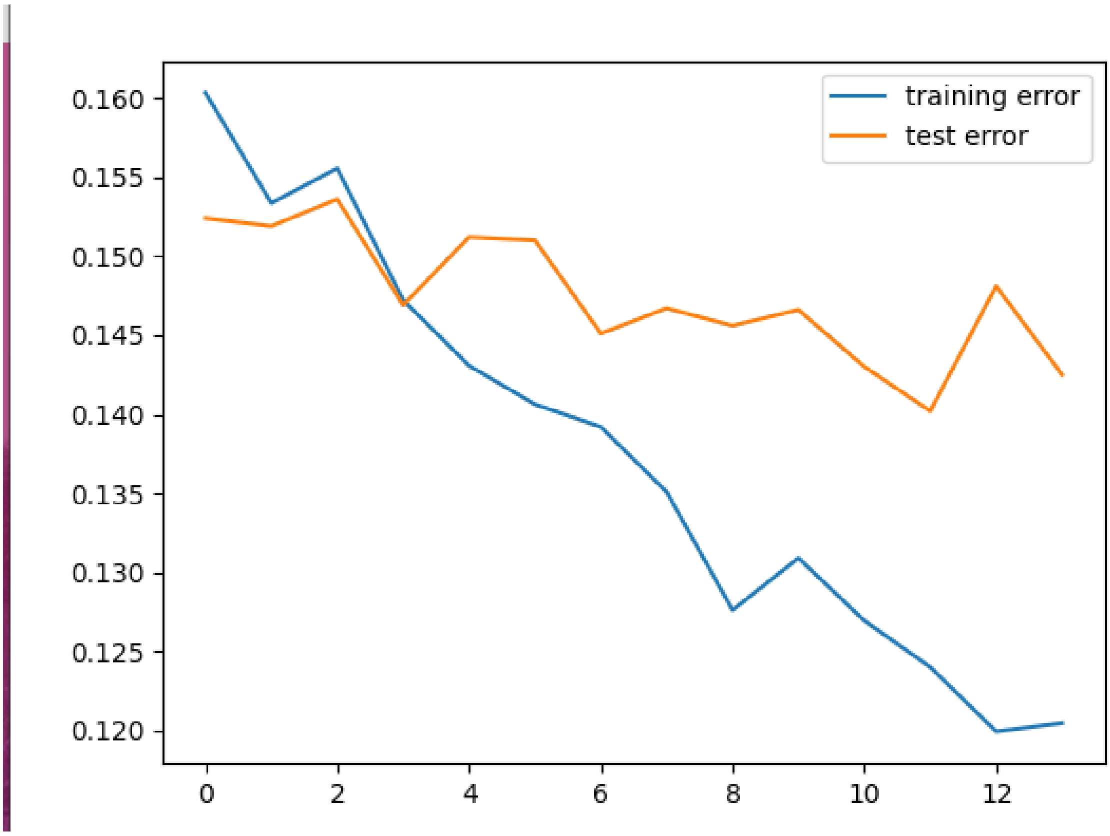
```
x = self.layer1(x)
x = F.relu(x)
self.layer1_2(x)
x = F.relu(x)
self.layer1_2(x)
x = F.relu(x)
self.layer1_2(x)
x = F.relu(x)
x = self.batchnorm1(x)
x = self.pool(x)
x = self.dropout(x)
x = self.layer2(x)
x = F.relu(x)
x = self.batchnorm2(x)
x = self.pool(x)
x = self.dropout(x)
x = self.layer3(x)
x = F.relu(x)
x = self.batchnorm3(x)
x = self.layer4(x)
x = F.relu(x)
x = self.batchnorm4(x)
x = self.layer5(x)
x = F.relu(x)
x = self.batchnorm5(x)
x = self.layer5(x)
x = F.relu(x)
x = self.batchnorm5(x)
```

```
x = self.layer5(x)
x = F.relu(x)
x = self.batchnorm5(x)
x = self.pool(x)
x = torch.flatten(x, 1)  # flatten all dimensions except batch
x = self.layer6(x)
x = F.relu(x)
x = self.layer7(x)
x = F.relu(x)
x = self.layer8(x)
return x
```

Test 1

```
100096 images processed with training loss: 0.158
Network accuracy on 10000 test images: 84.970 %
test loss for epoch:0.150
New record for test error.
epoch  15 390  complete.
<===========================================================>
Finished Training
Best parameters were at epoch 15, With test error rate 0.1503.
```






```
100096 images processed with training loss: 0.120
Network accuracy on 10000 test images: 85.750 %
test loss for epoch:0.142
epoch  14 390  complete.
<============================================================>
Finished Training
Best parameters were at epoch 12, With test error rate 0.1402.
Saving these parameters to model1
```

Seems to be clearly overfitting. No real advantage over architecture 8, so we'll abandon this change.

Architecture 10

Clone of architecture 8 with a softmax at the end. Softmax in theory should make [our model converge more quickly], saving on training time and potentially achieving more accuracy with epochs.

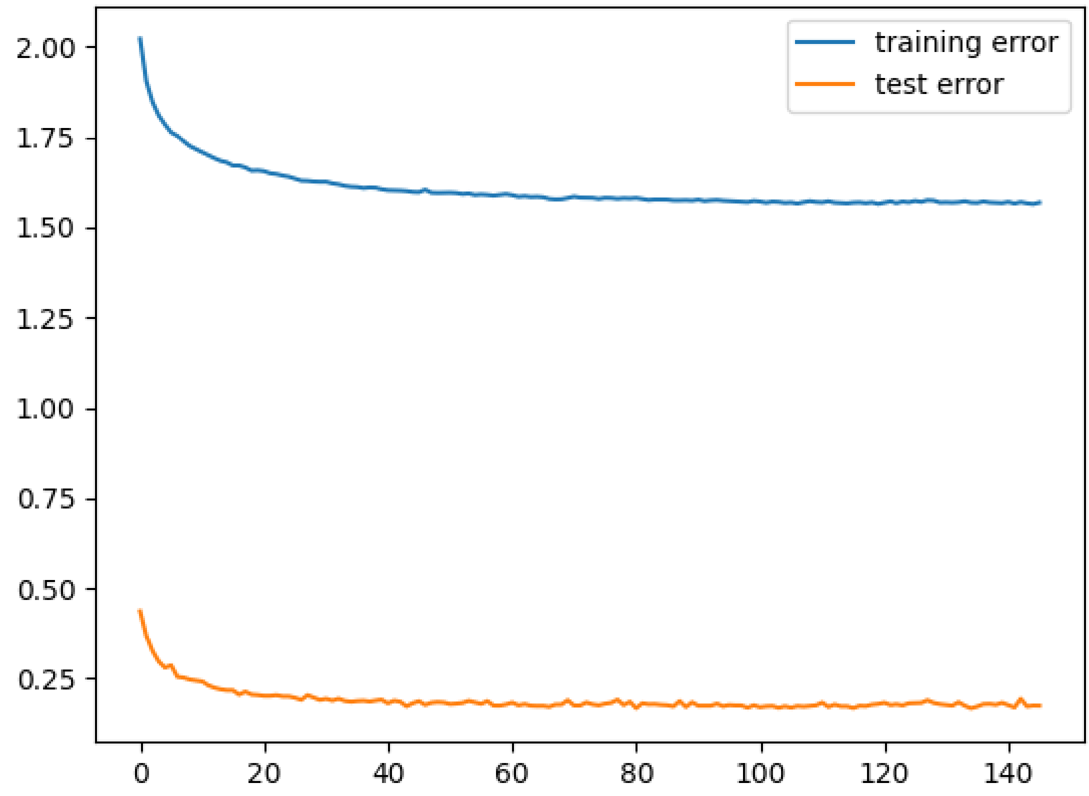
```
self.layer1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1)
self.pool = nn.MaxPool2d(2, 2)
self.layer2 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
self.layer3 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1)
self.layer4 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, padding=1)
self.layer5 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, padding=1)
self.layer6 = nn.Linear(8192, 128)
self.layer7 = nn.Linear(128, 64)
self.layer8 = nn.Linear(64, 10)
self.dropout = nn.Dropout(0.15)
self.batchnorm1 = nn.BatchNorm2d(64)
self.batchnorm2 = nn.BatchNorm2d(128)
self.batchnorm3 = nn.BatchNorm2d(256)
self.batchnorm4 = nn.BatchNorm2d(512)
self.batchnorm5 = nn.BatchNorm2d(512)
self.soft_max = nn.Softmax()
```

```
x = self.layer1(x)
x = F.relu(x)
x = self.batchnorm1(x)
x = self.pool(x)
x = self.dropout(x)
x = self.layer2(x)
x = F.relu(x)
x = self.batchnorm2(x)
x = self.pool(x)
x = self.dropout(x)
x = self.layer3(x)
x = F.relu(x)
x = self.batchnorm3(x)
x = self.layer4(x)
x = F.relu(x)
x = self.batchnorm4(x)
x = self.layer5(x)
x = F.relu(x)
x = self.batchnorm5(x)
x = self.layer5(x)
x = F.relu(x)
x = self.batchnorm5(x)
x = self.layer5(x)
x = F.relu(x)
x = self.batchnorm5(x)
x = self.pool(x)
x = torch.flatten(x, 1)  # flatten all dimensions except batch
x = self.layer6(x)
```

```
x = F.relu(x)
x = self.layer7(x)
x = F.relu(x)
x = self.layer8(x)
x = self.soft_max(x)
return x
```

Test 1

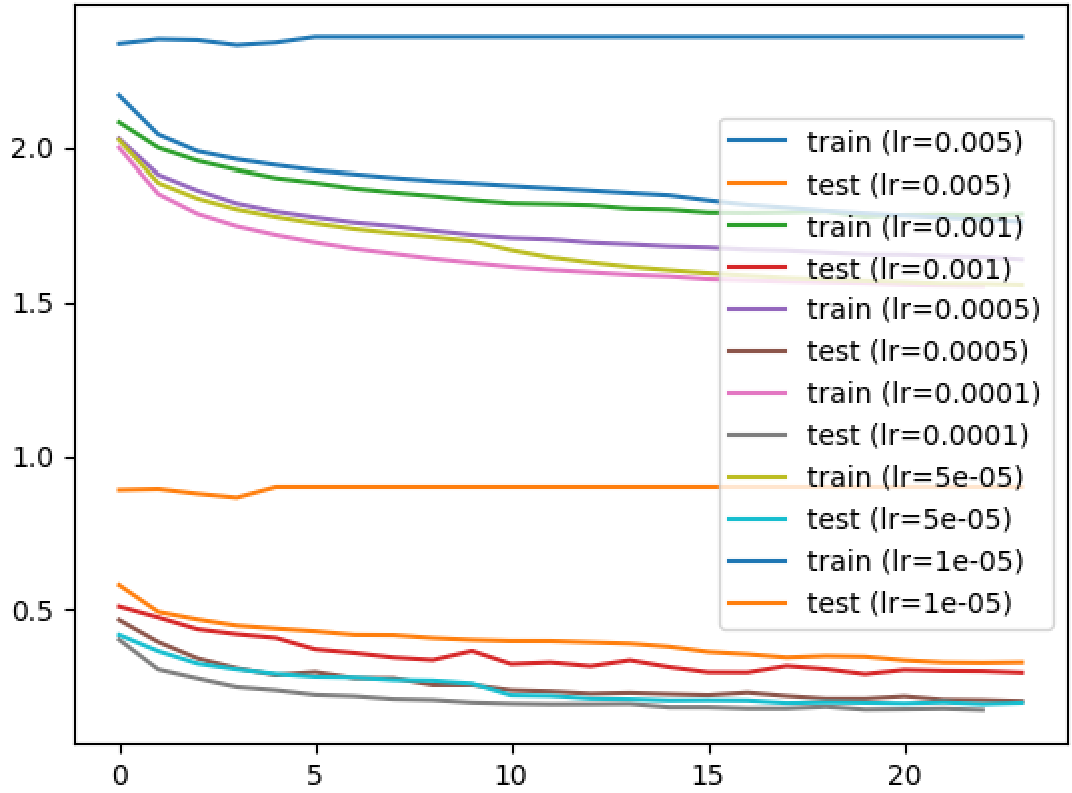
Initial test. Trained on optimal parameters for Architecture 8




```
100096 images processed with training loss: 1.569
Network accuracy on 10000 test images: 82.530 %
test loss for epoch:0.175
epoch  146 390  complete.
<============================================================>
Finished Training
Best parameters were at epoch 135, With test error rate 0.16749999999999998.
Saving these parameters to model1
```

Seemingly considerably less overfitting but has a lower accuracy. Interesting
The training time is also considerably lower as we hoped.

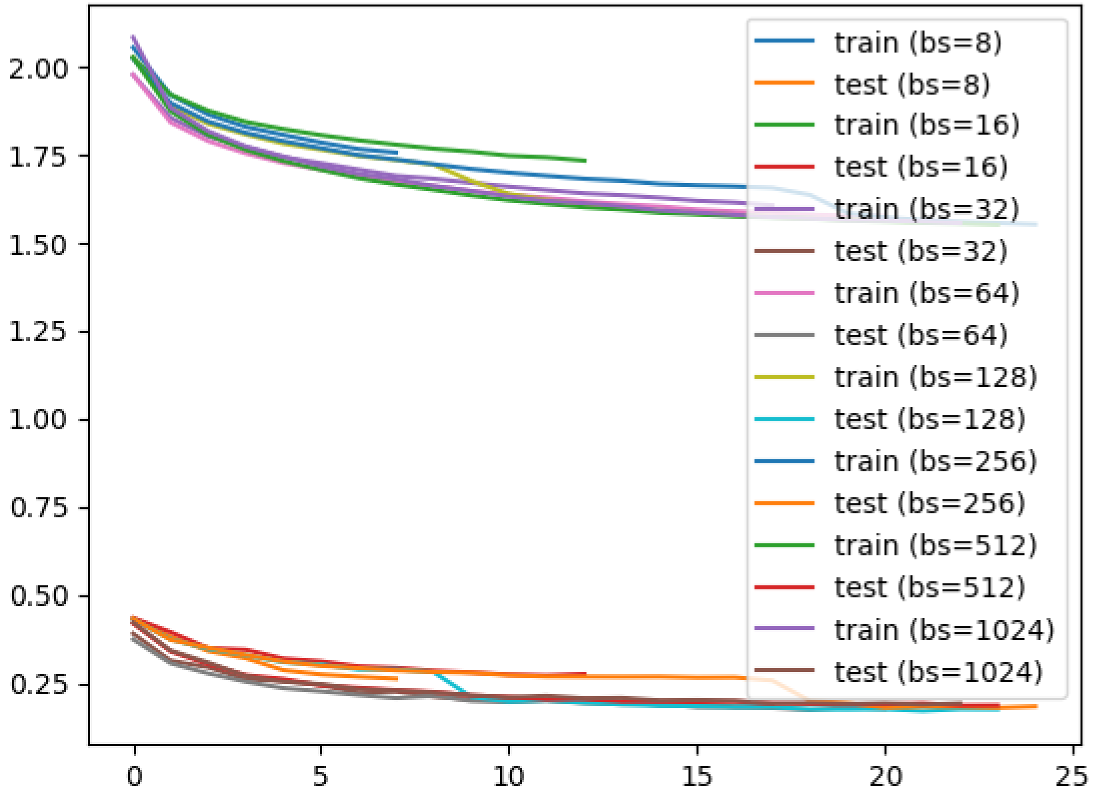
Test 2




```
-----------------------------------
Optimization complete. The optimal learning rate was 0.0001, with a test error of 0.17510000000000003, at an optimal epoch of 23
Saving the best model to model2
[0.8651, 0.2912, 0.20230000000000004, 0.17510000000000003, 0.19320000000000004, 0.32799999999999996]
```

0.0001 seems to be optimal, though there is not much between it and 0.0005. Test 1 was on 0.0005 with a longer training time and had a *slightly* improved accuracy, so this is not conclusive.

Test 3




```
---------------------------------
Optimization complete. The optimal batch size was 128, with a test error of 0.1713, at an optimal epoch of 22
Saving the best model to model2
[0.26280000000000003, 0.2722, 0.18330000000000002, 0.17410000000000003, 0.1713, 0.18030000000000002, 0.18669999999999998, 0.18879999999999997]
```

Optimization for batch size. 128 appears to be optimal.

So on the whole softmax seems to help considerably with overfitting but loses us some accuracy. We will test this again on our final model, but for now we will go back to architecture 8 as we're looking for the highest accuracy possible.

Architecture 11

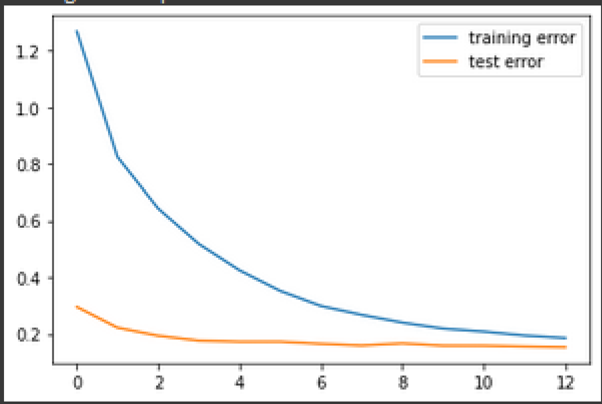
```
def __init__(self):
    super().__init__()
    self.layer1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1)
    self.layer1_2 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1)
    self.pool = nn.MaxPool2d(2, 2)
    self.layer2 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
    self.layer3 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1)
    self.layer4 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, padding=1)
    self.layer5 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, padding=1)
    self.layer6 = nn.Linear(8192, 128)
    self.layer7 = nn.Linear(128, 64)
    self.layer8 = nn.Linear(64, 10)
    self.dropout = nn.Dropout(0.15)
    self.batchnorm1 = nn.BatchNorm2d(64)
    self.batchnorm2 = nn.BatchNorm2d(128)
    self.batchnorm3 = nn.BatchNorm2d(256)
    self.batchnorm4 = nn.BatchNorm2d(512)
    self.batchnorm5 = nn.BatchNorm2d(512)
def forward(self, x):
    x = self.layer1(x)
    x = F.relu(x)
    x = self.batchnorm1(x)
    x = self.pool(x)
    x = self.dropout(x)
    x = self.layer1_2(x)
    x = self.batchnorm1(x)
    x = self.dropout(x)
    x = self.layer2(x)
    x = F.relu(x)
    x = self.batchnorm2(x)
    x = self.pool(x)
    x = self.dropout(x)
    x = self.layer3(x)
    x = F.relu(x)
    x = self.batchnorm3(x)
    x = self.layer4(x)
    x = F.relu(x)
    x = self.batchnorm4(x)
    x = self.layer5(x)
    x = F.relu(x)
    x = self.batchnorm5(x)
    x = self.layer5(x)
    x = F.relu(x)
    x = self.batchnorm5(x)
    x = self.layer5(x)
    x = F.relu(x)
    x = self.batchnorm5(x)
    x = self.pool(x)
    x = torch.flatten(x, 1)  # flatten all dimensions except batch
    x = self.layer6(x)
    x = F.relu(x)
    x = self.layer7(x)
    x = F.relu(x)
    x = self.layer8(x)
    return x
```

Test 1

Batch size = 128
Lr = 0.0005

Optimizer: Adam

```
Finished Training
Best parameters were at epoch 13, With test error rate 0.15259999999999996.
Saving these parameters to model1
```



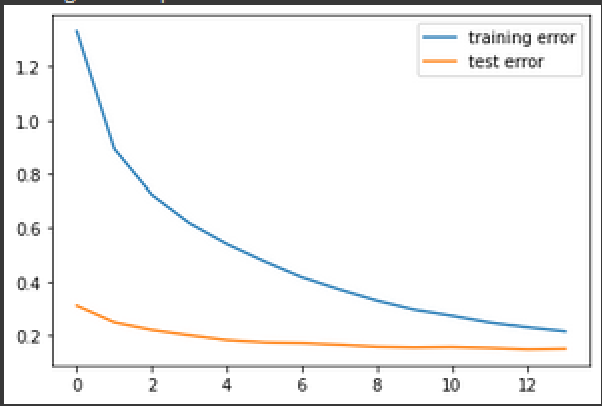

```
<==============================================================>
100096 images processed with training loss: 0.185
Network accuracy on 10000 test images: 84.740 %
test loss for epoch:0.153
New record for test error.
epoch  13 781  complete.
```

This model is built on architecture 8 which had the best result so far. Slightly less accuracy. I will keep adding more layers and testing as I go along to see if I can break the 86% barrier.

Test 2

Batch size = 128
Lr = 0.0005
Optimizer: Adam

```
Finished Training
Best parameters were at epoch 13, With test error rate 0.14790000000000003.
Saving these parameters to model1
```

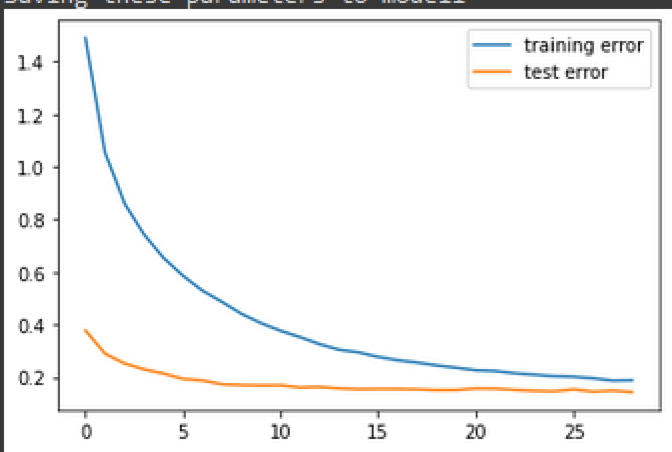



```
<=============================================================>
100096 images processed with training loss: 0.230
Network accuracy on 10000 test images: 85.210 %
test loss for epoch:0.148
New record for test error.
epoch  13 781  complete.
<=============================================================>
```

Adding another layer as mentioned above seems to have helped. There is also less overfitting. I will now add another layer and test again.

Test 3

```
<=============================================================>
Finished Training
Best parameters were at epoch 29, With test error rate 0.14300000000000002.
Saving these parameters to model1
```

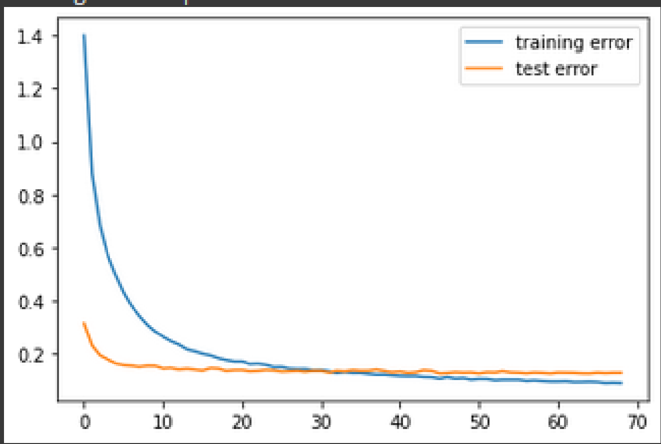


```
<==========================================================>
100096 images processed with training loss: 0.188
Network accuracy on 10000 test images: 85.700 %
test loss for epoch:0.143
New record for test error.
epoch  29 781  complete.
<==========================================================>
```

I decided to add all the remaining layers I planned to add at once. A very slight increase in accuracy but still not enough to beat our best so far.

Test 4

```
Finished Training
Best parameters were at epoch 46, With test error rate 0.12560000000000004.
Saving these parameters to model1
```



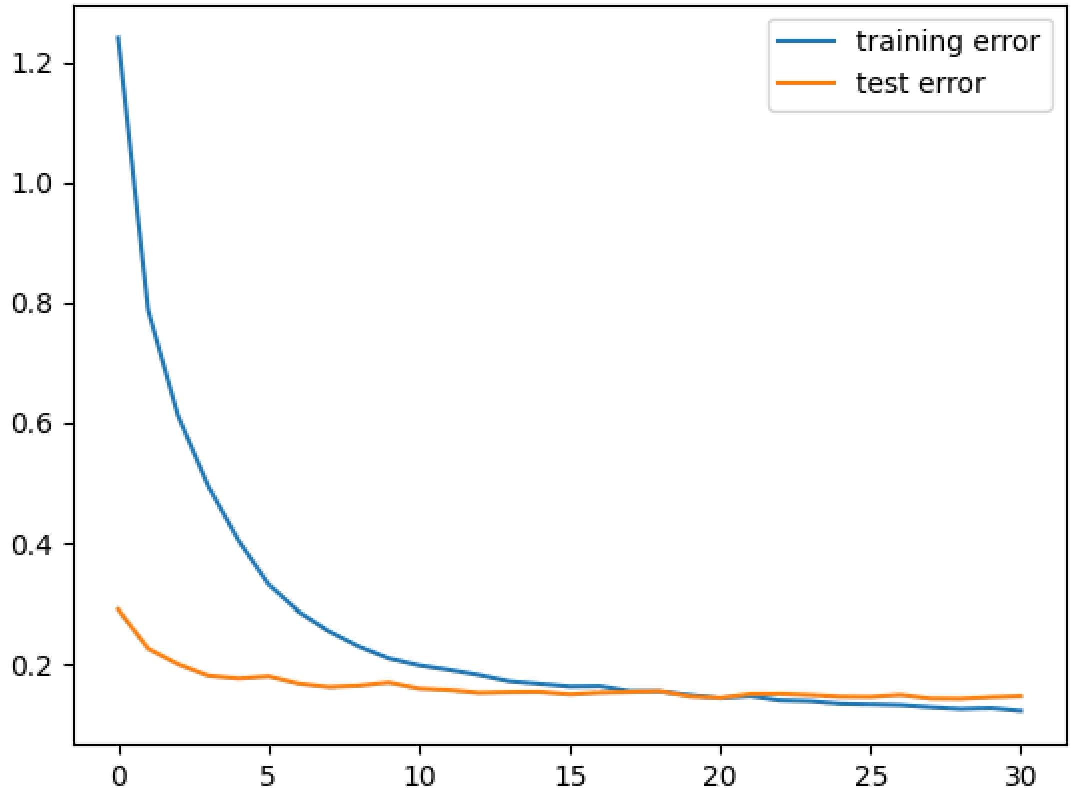
A maxpool has been applied after each layer.
Overfitting after around 30 epochs. The highest accuracy is 87% however this happens too far into the epochs to really count.

Architecture 12

```
x = F.relu(x)
x = self.batchnorm3(x)
x = self.layer4(x)
x = F.relu(x)
x = self.batchnorm4(x)
x = self.dropout(x)
x = self.layer5(x)
x = F.relu(x)
x = self.batchnorm5(x)
x = self.layer5(x)
x = F.relu(x)
x = self.batchnorm5(x)
x = self.layer5(x)
x = F.relu(x)
```

Architecture 8 but with another dropout before layer 5.

Test 2




```
Finished Training
Best parameters were at epoch 29, With test error rate 0.14259999999999995.
Saving these parameters to model1
```

Architecture 13 Best so far

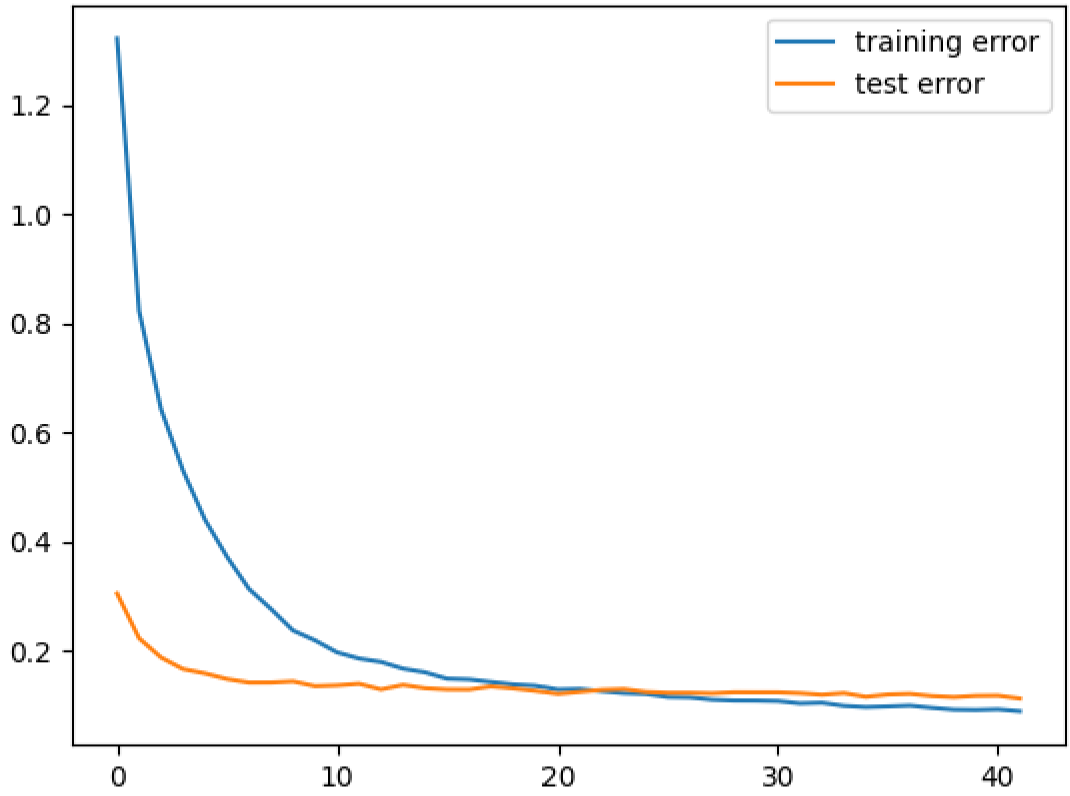
Same as 8, but we moved batchnorm before Relu. This was based on advice we found from a paper.

```
self.layer1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1)
self.pool = nn.MaxPool2d(2, 2)
self.layer2 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
self.layer3 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1)
self.layer4 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, padding=1)
self.layer5 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, padding=1)
self.layer6 = nn.Linear(8192, 128)
self.layer7 = nn.Linear(128, 64)
self.layer8 = nn.Linear(64, 10)
self.dropout = nn.Dropout(0.15)
self.batchnorm1 = nn.BatchNorm2d(64)
self.batchnorm2 = nn.BatchNorm2d(128)
self.batchnorm3 = nn.BatchNorm2d(256)
self.batchnorm4 = nn.BatchNorm2d(512)
self.batchnorm5 = nn.BatchNorm2d(512)
self.soft_max = nn.Softmax()
```

```python
x = self.layer1(x)
x = self.batchnorm1(x)
x = F.relu(x)
x = self.pool(x)
x = self.layer2(x)
x = F.relu(x)
x = self.dropout(x)
x = self.layer3(x)
x = self.batchnorm3(x)
x = F.relu(x)
x = self.dropout(x)
x = self.pool(x)
x = self.layer4(x)
x = self.batchnorm4(x)
x = F.relu(x)
x = self.layer5(x)
x = self.batchnorm5(x)
x = F.relu(x)
x = self.layer5(x)
x = self.batchnorm5(x)
x = F.relu(x)
x = self.layer5(x)
x = self.batchnorm5(x)
x = F.relu(x)
x = self.pool(x)
x = torch.flatten(x, 1)  # flatten all dimensions except batch
```

```python
x = self.layer6(x)
x = F.relu(x)
x = self.layer7(x)
x = F.relu(x)
x = self.layer8(x)
#x = self.soft_max(x)
return x
```

Test 1




```
Finished Training
Best parameters were at epoch 42, With test error rate 0.11339999999999995.
Saving these parameters to model1
```

This test lead to our lowest test error so far! I will extend this test a bit further (by reloading the model) to see if we can eke out a bit more.

<u>Test 2 ****</u>**

Optim : SGD

Lr : 0.01

Batch size: 128

```
<=============================================================>
Finished Training
Best parameters were at epoch 30, With test error rate 0.11699999999999999.
Saving these parameters to model1
```

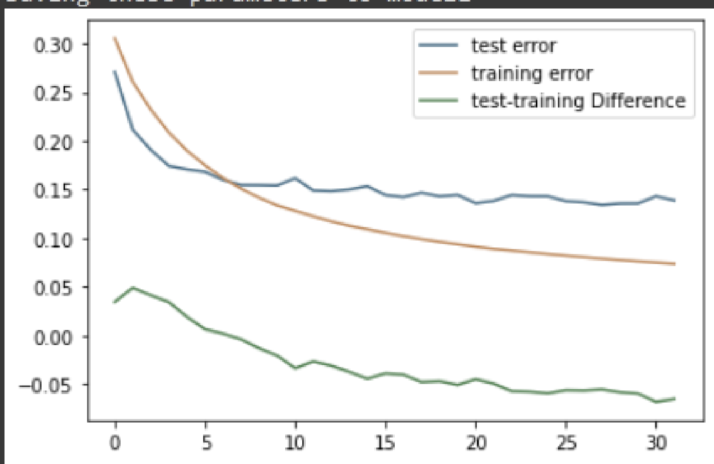
Test 3 *****

Optim : Adam

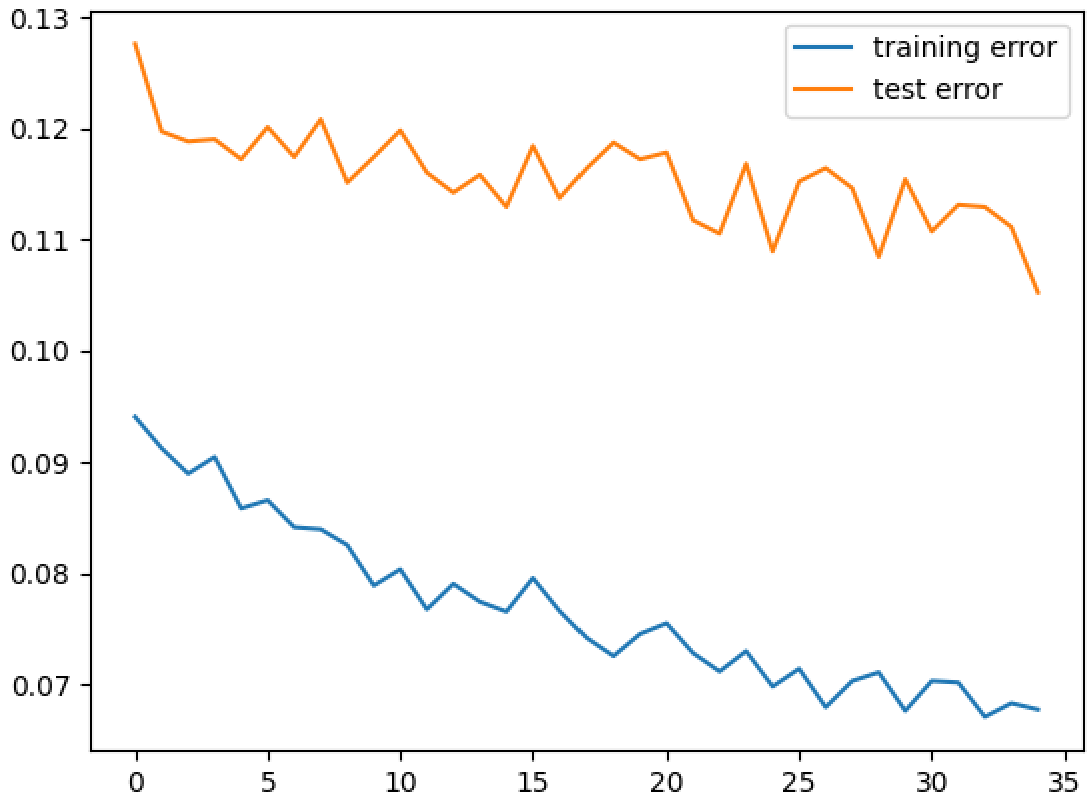
Lr : 0.0001

Batch size: 128

```
<=============================================================>
Finished Training
Best parameters were at epoch 28, With test error rate 0.13419999999999999.
Saving these parameters to model1
```



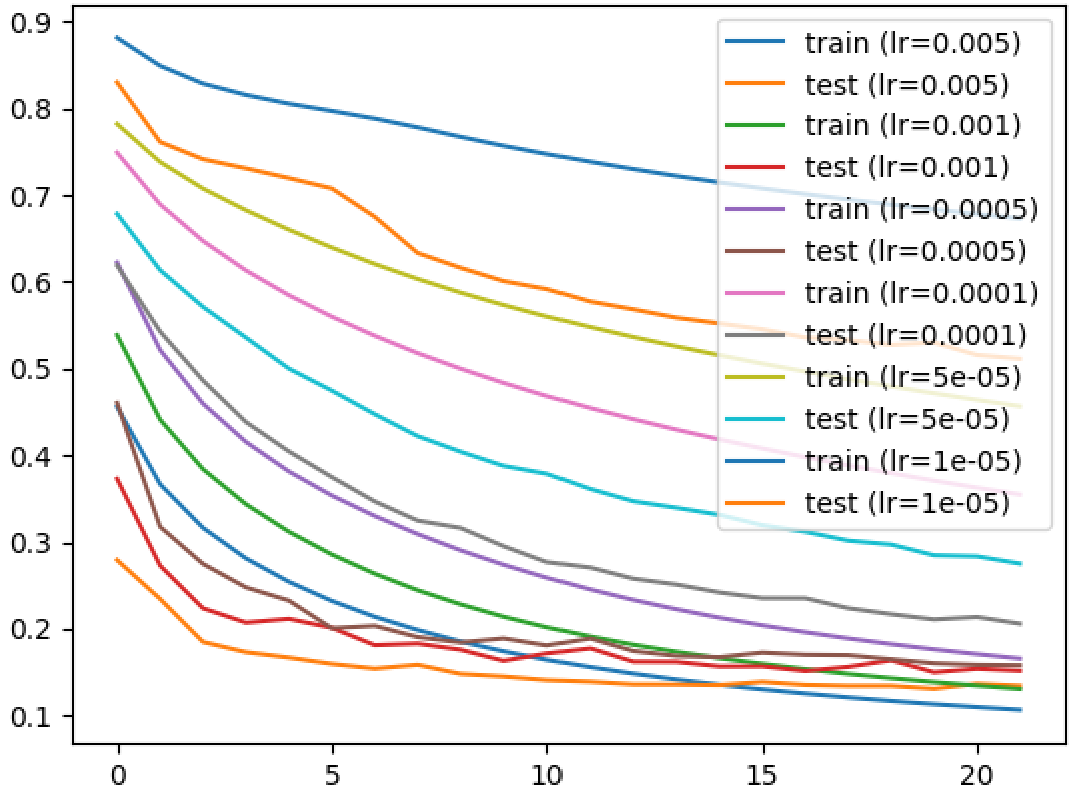
——------------------ #this doesn’t seem to be labeled as a specific test




```
100096 images processed with training loss: 0.068
Network accuracy on 10000 test images: 89.480 %
test loss for epoch:0.105
New record for test error.
epoch  35 390  complete.
<==========================================================>
Finished Training
Best parameters were at epoch 35, With test error rate 0.10519999999999996.
Saving these parameters to model1
```

89.48% accuracy! That’s our best so far. We will now try optimizing the learning rate for this model

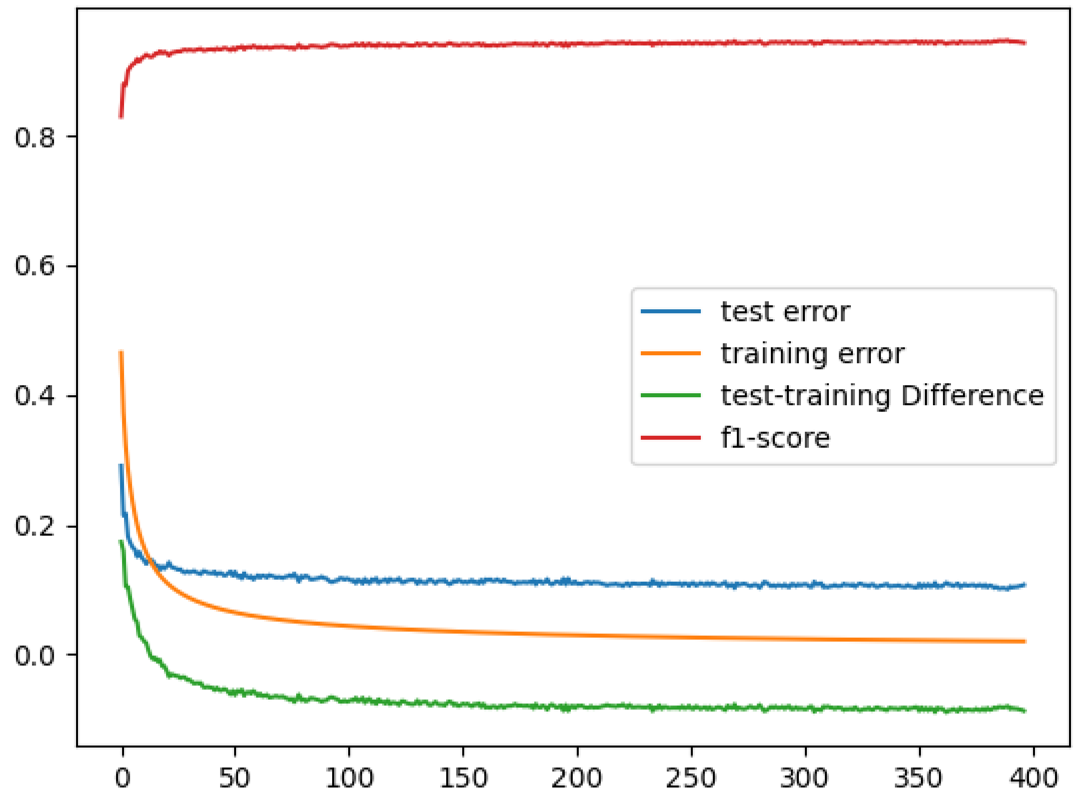
Test 3




```
Finished Training
Best parameters were at epoch 22, With test error rate 0.5114000000000001.
----------------------------------
----------------------------------
Optimization complete. The optimal learning rate was 0.005, with a test error of 0.13080000000000003, at an optimal epoch of 20
Saving the best model to lr_optimized
[0.13080000000000003, 0.14980000000000004, 0.15780000000000005, 0.20579999999999998, 0.2751, 0.5114000000000001]
```

Test 4

Final train with SGD, learning rate of 0.005, momentum 0.9, batch size 128.




```
Finished Training
Best parameters were at epoch 390, With test error rate 0.10040000000000004.
Saving these parameters to model1
```

```
100096  images processed with training loss: 0.023
Network accuracy on 10000 test images: 90.22 %
 F1: 0.949
training accuracy:  99.16831531531531%
test loss for epoch:0.098
New record for test error.
epoch  111  complete.
```

Architecture 14 - ResNet

Residual neural network test

```python
class ResidualBlock(nn.Module):
    '''
    Residual blocks to be used in a residual neural network.
    Implements a 'shortcut' every two blocks that allows the net to learn an identity function as an alternative to those
    two blocks.
    Essentially this means that we can stack as many of these 'blocks' as we like and the outcome will either be:
    a. The additional layers make the performance of the model worse. Therefore the model learns the identity function instead,
    and thus the layers have no effect.
    b. The additional layers improve the accuracy of the model.
    This means that we can keep stacking layers without risking making the model less accurate. The worst-case scenario
    is that there is no change to the model, and ideally the model's accuracy should improve.
    See https://arxiv.org/pdf/1512.03385.pdf for more details.
    For the math see here https://www.youtube.com/watch?v=RYth6EbBUqM
    '''

    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.layer1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding=1)
        self.batch_norm = nn.BatchNorm2d(out_channels)
        self.layer2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1)

    def forward(self, x):
        shortcut = x  # store the value so we can add it later
        x = self.layer1(x)
        x = F.relu(x)
        x = self.layer2(x)
        x = torch.add(x, shortcut)  # add old activation function output element-wise
        return x
```

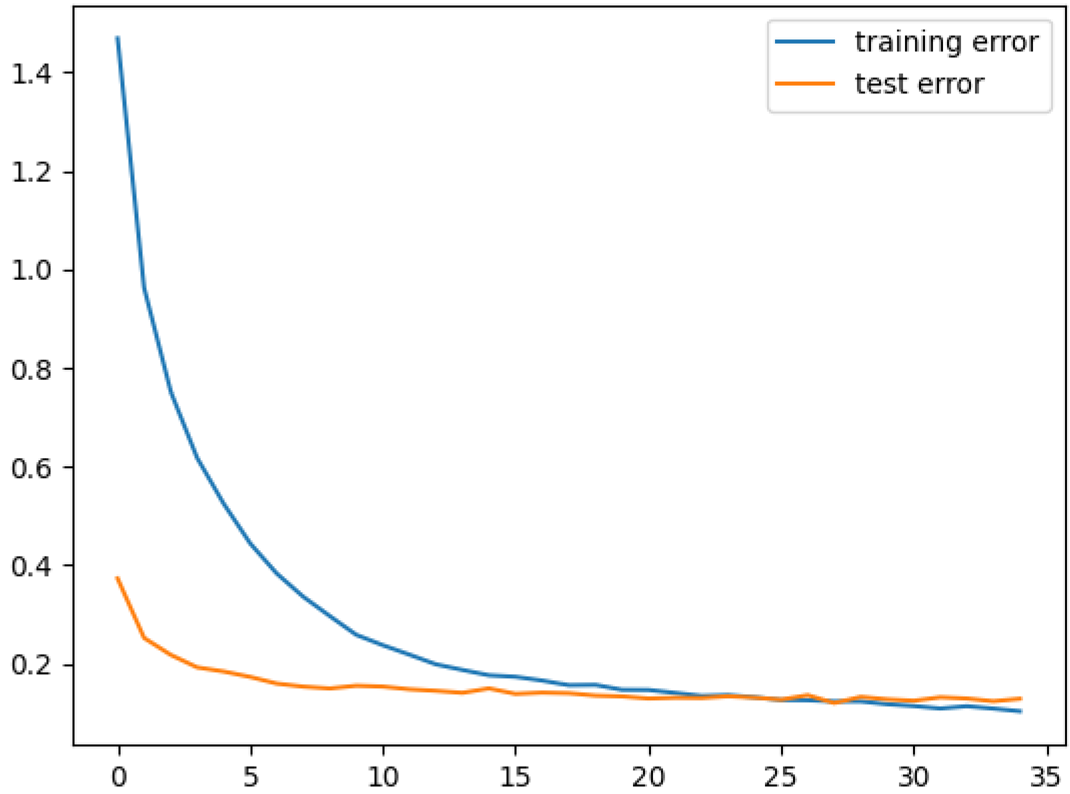
```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.layer2 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.layer3 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1)
        self.layer4 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, padding=1)
        self.layer5 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, padding=1)
        self.layer6 = nn.Linear(8192, 128)
        self.layer7 = nn.Linear(128, 64)
        self.layer8 = nn.Linear(64, 10)
        self.dropout = nn.Dropout(0.15)
        self.residual_block = ResidualBlock(128, 128)
        self.batchnorm1 = nn.BatchNorm2d(64)
        self.batchnorm2 = nn.BatchNorm2d(128)
        self.batchnorm3 = nn.BatchNorm2d(256)
        self.batchnorm4 = nn.BatchNorm2d(512)
        self.batchnorm5 = nn.BatchNorm2d(512)
        self.soft_max = nn.Softmax()
```

```python
    def forward(self, x):
        x = self.layer1(x)
        x = self.batchnorm1(x)
        x = F.relu(x)
        x = self.pool(x)
        x = self.layer2(x)
        x = F.relu(x)
        x = self.residual_block(x)
        x = F.relu(x)
        x = self.residual_block(x)
        x = F.relu(x)
        x = self.dropout(x)
        x = self.layer3(x)
        x = self.batchnorm3(x)
        x = F.relu(x)
        x = self.dropout(x)
        x = self.pool(x)
        x = self.layer4(x)
        x = self.batchnorm4(x)
        x = F.relu(x)
        x = self.layer5(x)
        x = self.batchnorm5(x)
        x = F.relu(x)
        x = self.layer5(x)
        x = self.batchnorm5(x)
        x = F.relu(x)
        x = self.layer5(x)
        x = self.batchnorm5(x)
```

```python
x = F.relu(x)
x = self.pool(x)
x = torch.flatten(x, 1)  # flatten all dimensions except batch
x = self.layer6(x)
x = F.relu(x)
x = self.layer7(x)
x = F.relu(x)
x = self.layer8(x)
# x = self.soft_max(x)
return x
```

Test 1

This was just a test to see if we could run the ResNet without losing any accuracy




```
<==========================================================>
Finished Training
Best parameters were at epoch 28, With test error rate 0.12050000000000005.
Saving these parameters to model1
```

The model seems to work just fine! Time to expand ResNet blocks to more layers.

Architecture 15 - Expanded ResNet

The idea here is to preserve the structure of Architecture 13 - which we know works well - while adding numerous ResNet blocks in between each layer. In theory this should make a model that is at least no worse than Architecture 13, (since in the worst case our model will just learn the identity for these layers) and hopefully much better.

Test 1

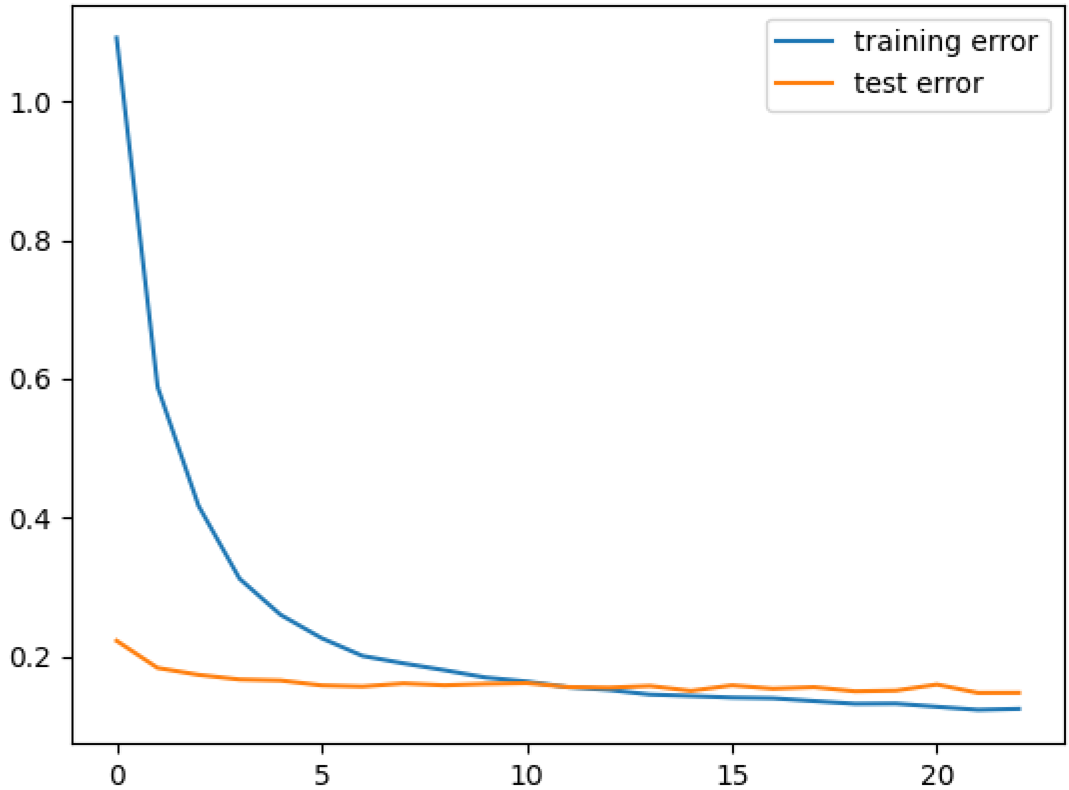
```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1)
        self.residual1 = ResidualBlock(in_channels=64, out_channels=64)
        self.pool = nn.MaxPool2d(2, 2)
        self.layer2 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.residual2 = ResidualBlock(in_channels=128, out_channels=128)
        self.layer3 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1)
        self.residual3 = ResidualBlock(in_channels=256, out_channels=256)
        self.layer4 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, padding=1)
        self.residual4 = ResidualBlock(in_channels=512, out_channels=512)
        self.layer5 = nn.Linear(8192, 128)
        self.layer6 = nn.Linear(128, 64)
        self.layer7 = nn.Linear(64, 10)
        self.dropout = nn.Dropout(0.15)
        self.batchnorm1 = nn.BatchNorm2d(64)
        self.batchnorm2 = nn.BatchNorm2d(128)
        self.batchnorm3 = nn.BatchNorm2d(256)
        self.batchnorm4 = nn.BatchNorm2d(512)
        self.batchnorm5 = nn.BatchNorm2d(512)
        #self.soft_max = nn.Softmax()
```

```python
    def forward(self, x):
        x = self.layer1(x)
        x = F.relu(x)
        x = self.residual1(x)
        x = F.relu(x)
        x = self.residual1(x)
        x = F.relu(x)
        x = self.residual1(x)
        x = F.relu(x)
        x = self.batchnorm1(x)
        x = F.relu(x)
        x = self.pool(x)
        x = self.layer2(x)
        x = F.relu(x)
        self.residual2(x)
        x = F.relu(x)
        self.residual2(x)
        x = F.relu(x)
        self.residual2(x)
        x = self.batchnorm2(x)
        x = F.relu(x)
        x = self.dropout(x)
        x = self.layer3(x)
        x = self.residual3(x)
        x = F.relu(x)
        x = self.residual3(x)
        x = F.relu(x)
        x = self.residual3(x)
```

```
        x = self.batchnorm3(x)
        x = F.relu(x)
        x = self.dropout(x)
        x = self.pool(x)
        x = self.layer4(x)
        x = F.relu(x)
        x = self.residual4(x)
        x = F.relu(x)
        x = self.residual4(x)
        x = F.relu(x)
        x = self.residual4(x)
        x = self.batchnorm4(x)
        x = F.relu(x)
        x = self.layer5(x)
        x = self.batchnorm5(x)
        x = F.relu(x)
        x = self.layer5(x)
        x = self.batchnorm5(x)
        x = F.relu(x)
        x = self.layer5(x)
        x = self.batchnorm5(x)
        x = F.relu(x)
        x = self.pool(x)
        x = torch.flatten(x, 1)  # flatten all dimensions except batch
        x = self.layer6(x)
        x = F.relu(x)
        x = self.layer7(x)
        x = F.relu(x)
```

```
x = self.layer8(x)
# x = self.soft_max(x)
return x
```

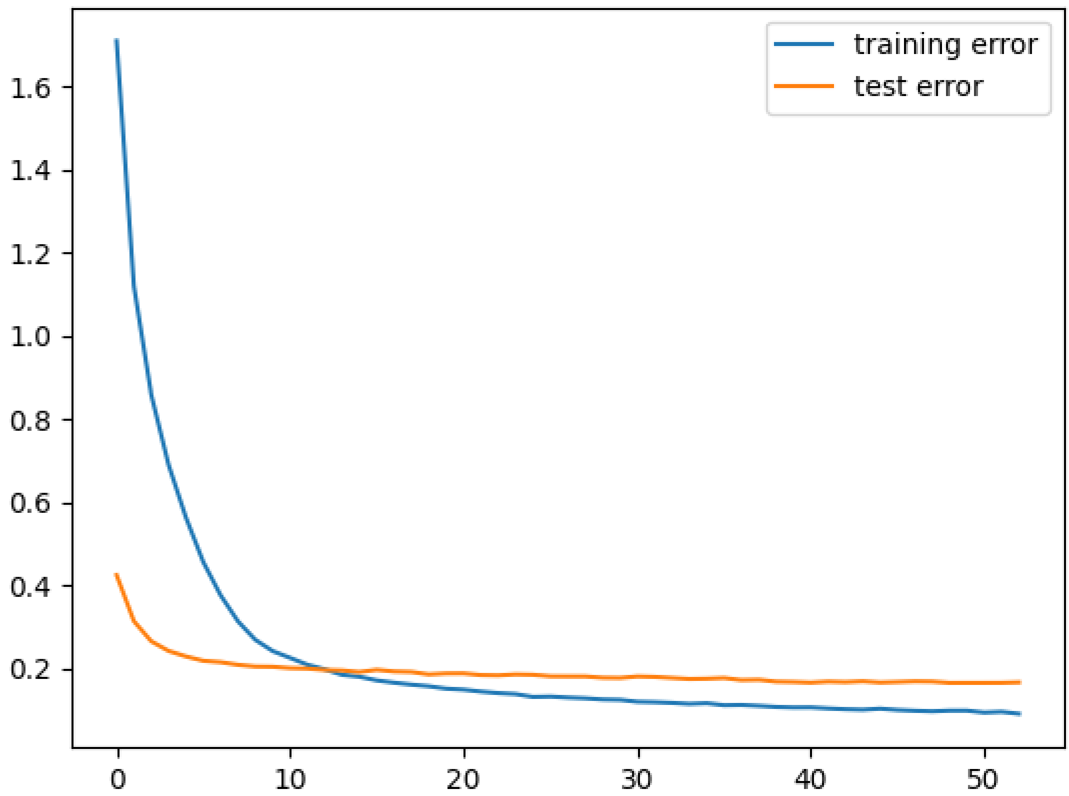
```
<============================================================>
100096 images processed with training loss: 0.124
Network accuracy on 10000 test images: 85.250 %
test loss for epoch:0.147
epoch  23 390  complete.
<============================================================>
Finished Training
Best parameters were at epoch 22, With test error rate 0.14739999999999998.
Saving these parameters to model1
```



Absolutely fine, time to add more layers.

Test 2

This time we will double the number of Res blocks. We will also remove dropouts as the paper recommends not to use them.
(screenshots omitted for brevity)
We will also be moving over to sgd with lr=0.001, momentum =0.9 and weight decay =0.00005. The paper recommends sgd with weight decay and momentum of 0.9.




```
100096 images processed with training loss: 0.091
Network accuracy on 10000 test images: 83.380 %
test loss for epoch:0.166
epoch  53 390  complete.
<============================================================>
Finished Training
Best parameters were at epoch 49, With test error rate 0.16490000000000005.
Saving these parameters to model1
```

Fairly mediocre.