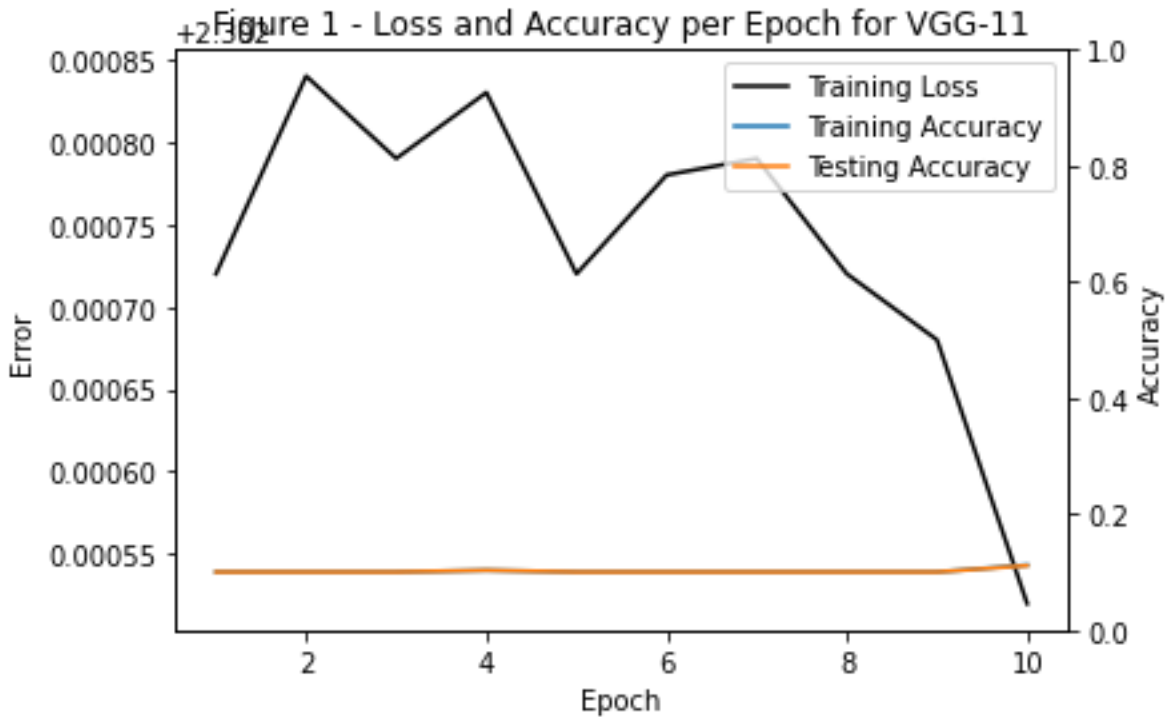
https://github.com/Christian-Martens-UNCC/ECGR-4106/tree/main/Homework_3-Modern_CNNs

I had a lot of issues with this homework. Not with understanding the material or writing the code out, but in getting models to train. I spent well over 20 hours and 100 Google Colab GPU credits trying to tweak learning rates, batch normalization, dropout, weight decay, anything at all to try and get outputs that I could discuss for this homework, but nothing I did solved my problems. I know the code works (I've used the same code for training on each of the other homework in addition to the fact that it successfully trained the ResNet model). I am at a loss as for why my VGG and GoogLeNet models weren't learning, but rest assured it wasn't due to a lack of trying (my hundreds of lines of code will attest to that).

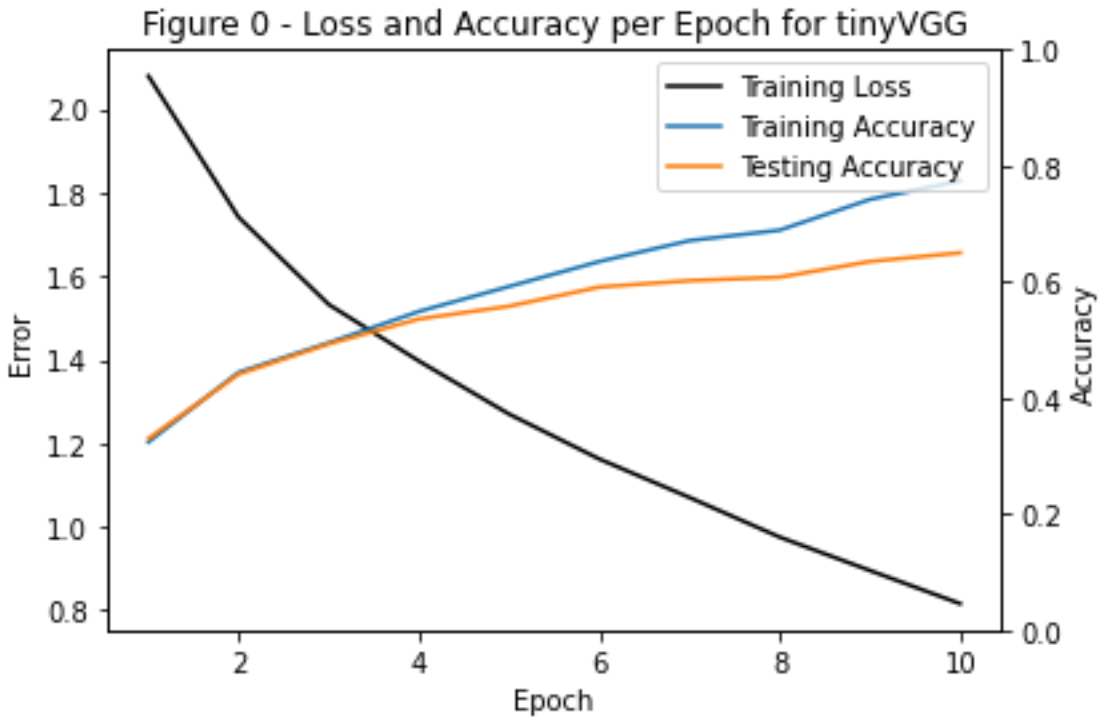
Problem 1:

A) I could not get a valid VGG-11 model to work, although I was able to produce a model which was much simpler to work. Again, this leads me to believe that the models were too complicated for the task at hand. As can be seen from the graph, the training loss, training accuracy, and testing accuracy are completely stagnant (the training loss scale is very small). Below the VGG-11 model figure is the "tinyVGG" model I produced using the same structure and code as VGG, albeit simpler.

```
class VGG_11(nn.Module):
    def __init__(self, arch, num_classes=10):
        super(VGG_11, self).__init__()
        conv_blks = []
        for (num_convs, out_channels) in arch:
            conv_blks.append(vgg_block(num_convs, out_channels))
        self.conv_blks = nn.Sequential(*conv_blks, nn.Flatten())
        self.fc1 = nn.LazyLinear(4096)
        self.fc2 = nn.LazyLinear(4096)
        self.fc3 = nn.LazyLinear(num_classes)
        self.fc_drop = nn.Dropout(p=0.5)
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.conv_blks(x)
        out = self.fc_drop(self.relu(self.fc1(out)))
        out = self.fc_drop(self.relu(self.fc2(out)))
        out = self.fc3(out)
        return out

model_1 = VGG_11(arch=((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))).to(device=try_gpu())
optimizer_1 = optim.SGD(model_1.parameters(), lr=0.1)
model_1.eval()
```




```python
class tinyVGG(nn.Module):
    def __init__(self, arch, num_classes=10):
        super(tinyVGG, self).__init__()
        conv_blks = []
        for (num_convs, out_channels) in arch:
            conv_blks.append(vgg_block(num_convs, out_channels))
        self.conv_blks = nn.Sequential(*conv_blks, nn.Flatten())
        self.fc1 = nn.LazyLinear(128)
        self.fc2 = nn.LazyLinear(64)
        self.fc3 = nn.LazyLinear(num_classes)
        self.fc_drop = nn.Dropout(p=0.5)
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.conv_blks(x)
        out = self.fc_drop(self.relu(self.fc1(out)))
        out = self.fc_drop(self.relu(self.fc2(out)))
        out = self.fc3(out)
        return out

model_0 = tinyVGG(arch=((1, 64), (1, 128))).to(device=try_gpu())
optimizer_0 = optim.SGD(model_0.parameters(), lr=0.08)
model_0.eval()
```




```
Epoch 10:
        Duration = 53.07 seconds
        Training Loss: 0.8161
        Training Accuracy: 0.774
        Validation Accuracy: 0.65
tensor([[752,  15,  40,  19,  24,   8,  29,  22,  64,  27],
        [ 46, 738,  10,  16,  10,   7,  14,  10,  41, 108],
        [ 80,   5, 446,  66, 177,  59,  97,  53,  10,   7],
        [ 18,   5,  58, 484,  96, 134, 130,  47,  14,  14],
        [ 35,   2,  55,  55, 652,  21,  91,  82,   6,   1],
        [ 13,   2,  46, 202,  87, 500,  74,  66,   5,   5],
        [  6,   5,  27,  55, 105,  16, 761,  15,   6,   4],
        [ 20,   2,  26,  37,  77,  53,  24, 746,   4,  11],
        [102,  41,  13,  22,  14,  11,  23,   5, 742,  27],
        [ 59, 116,  12,  14,   8,   6,  25,  33,  46, 681]])
```

The training accuracy reaches roughly 77% and appears to be overfitting the validation accuracy somewhat. The adjoining confusion matrix is listed as well.

B) Although I could not get viable accuracies, It was plain to see that the model size for the VGG-16 and VGG-19 were much larger than VGG-11. My VGG-11 computed more than twice as fast as the other program, so unless VGG-16 and/or VGG-19 provide a large accuracy boost (from

discussions I've had with other students, it doesn't), VGG-11 is the preferable model in most circumstances.

```
class VGG_16(nn.Module):
    def __init__(self, arch, num_classes=10):
        super(VGG_16, self).__init__()
        conv_blks = []
        for (num_convs, out_channels) in arch:
            conv_blks.append(vgg_block(num_convs, out_channels))
        self.conv_blks = nn.Sequential(*conv_blks, nn.Flatten())
        self.fc1 = nn.LazyLinear(4096)
        self.fc2 = nn.LazyLinear(4096)
        self.fc3 = nn.LazyLinear(num_classes)
        self.fc_drop = nn.Dropout(p=0.5)
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.conv_blks(x)
        out = self.fc_drop(self.relu(self.fc1(out)))
        out = self.fc_drop(self.relu(self.fc2(out)))
        out = self.fc3(out)
        return out

model_2 = VGG_16(arch=((2, 64), (2, 128), (3, 256), (3, 512), (3, 512))).to(device=try_gpu())
optimizer_2 = optim.SGD(model_2.parameters(), lr=0.1)
model_2.eval()
```

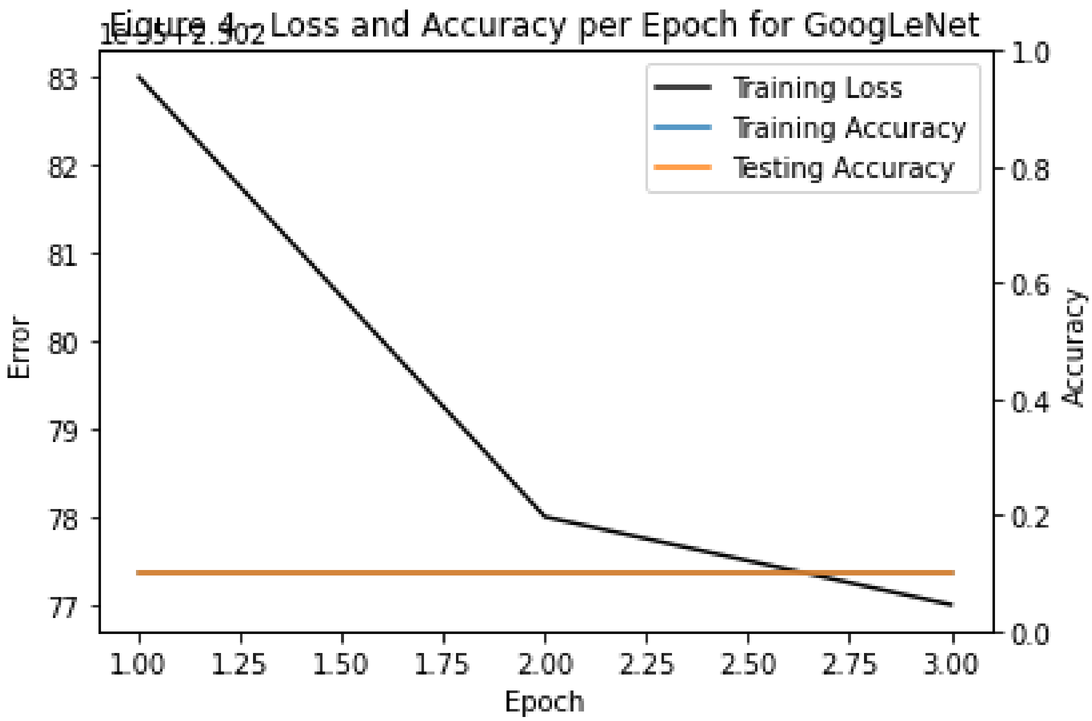
```
class VGG_19(nn.Module):
    def __init__(self, arch, num_classes=10):
        super(VGG_19, self).__init__()
        conv_blks = []
        for (num_convs, out_channels) in arch:
            conv_blks.append(vgg_block(num_convs, out_channels))
        self.conv_blks = nn.Sequential(*conv_blks, nn.Flatten())
        self.fc1 = nn.LazyLinear(4096)
        self.fc2 = nn.LazyLinear(4096)
        self.fc3 = nn.LazyLinear(num_classes)
        self.fc_drop = nn.Dropout(p=0.5)
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.conv_blks(x)
        out = self.fc_drop(self.relu(self.fc1(out)))
        out = self.fc_drop(self.relu(self.fc2(out)))
        out = self.fc3(out)
        return out

model_3 = VGG_19(arch=((2, 64), (2, 128), (4, 256), (4, 512), (4, 512))).to(device=try_gpu())
optimizer_3 = optim.SGD(model_3.parameters(), lr=0.1)
model_3.eval()
```

Problem 2:

A) Similar to the VGG models, I could not find a way to get the GoogLeNet models to successfully train despite having multiple peers look over my code to help me bugfix it. My training loss and accuracies over many attempts was 2.3 and 10%, respectively. Again, I believe this lack of a result to have something to do with the complexity of the model and the few epochs we were supposed to train the model on.

```
class GoogLeNet(nn.Module):
    def __init__(self, num_classes=10):
        super(GoogLeNet, self).__init__()
        self.stem = nn.Sequential(nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                                  nn.ReLU(),
                                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
                                  nn.LazyConv2d(64, kernel_size=1),
                                  nn.ReLU(),
                                  nn.LazyConv2d(192, kernel_size=3, padding=1),
                                  nn.ReLU(),
                                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        self.body1 = nn.Sequential(Inception(64, (96, 128), (16, 32), 32),
                                   Inception(128, (128, 192), (32, 96), 64),
                                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        self.body2 = nn.Sequential(Inception(192, (96, 208), (16, 48), 64),
                                   Inception(160, (112, 124), (24, 64), 64),
                                   Inception(128, (128, 256), (24, 64), 64),
                                   Inception(112, (144, 288), (32, 64), 64),
                                   Inception(256, (160, 320), (32, 128), 128),
                                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        self.body3 = nn.Sequential(Inception(256, (160, 320), (32, 128), 128),
                                   Inception(384, (192, 384), (48, 128), 128),
                                   nn.AdaptiveAvgPool2d((1,1)),
                                   nn.Flatten())
        self.fc = nn.LazyLinear(num_classes)


    def forward(self, x):
        out = self.stem(x)
        out = self.body1(out)
        out = self.body2(out)
        out = self.body3(out)
        out = self.fc(out)
        return out

model_4 = GoogLeNet().to(device=try_gpu())
optimizer_4 = optim.SGD(model_4.parameters(), lr=0.1)
model_4.eval()
```

```python
class AltGoogLeNet(nn.Module):
    def __init__(self, num_classes=10):
        super(GoogLeNet, self).__init__()
        self.stem = nn.Sequential(nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                                  nn.LazyBatchNorm2d(),
                                  nn.ReLU(),
                                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
                                  nn.LazyConv2d(64, kernel_size=1),
                                  nn.LazyBatchNorm2d(),
                                  nn.ReLU(),
                                  nn.LazyConv2d(192, kernel_size=3, padding=1),
                                  nn.LazyBatchNorm2d(),
                                  nn.ReLU(),
                                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        self.body1 = nn.Sequential(Inception(64, (96, 128), (16, 32), 32),
                                  Inception(128, (128, 192), (32, 96), 64),
                                  nn.LazyBatchNorm2d(),
                                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        self.body2 = nn.Sequential(Inception(192, (96, 208), (16, 48), 64),
                                  Inception(160, (112, 124), (24, 64), 64),
                                  Inception(128, (128, 256), (24, 64), 64),
                                  Inception(112, (144, 288), (32, 64), 64),
                                  Inception(256, (160, 320), (32, 128), 128),
                                  nn.LazyBatchNorm2d(),
                                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        self.body3 = nn.Sequential(Inception(256, (160, 320), (32, 128), 128),
                                  Inception(384, (192, 384), (48, 128), 128),
                                  nn.LazyBatchNorm2d(),
                                  nn.AdaptiveAvgPool2d((1,1)),
                                  nn.Flatten())
        self.fc = nn.LazyLinear(num_classes)


    def forward(self, x):
        out = self.stem(x)
        out = self.body1(out)
        out = self.body2(out)
        out = self.body3(out)
        out = self.fc(out)
        return out
```




```
        Training Loss: 2.30277
        Training Accuracy: 0.1
        Validation Accuracy: 0.1
tensor([[   0,    0,    0,    0,    0,    0,    0,    0,    0, 1000],
        [   0,    0,    0,    0,    0,    0,    0,    0,    0, 1000],
        [   0,    0,    0,    0,    0,    0,    0,    0,    0, 1000],
        [   0,    0,    0,    0,    0,    0,    0,    0,    0, 1000],
        [   0,    0,    0,    0,    0,    0,    0,    0,    0, 1000],
        [   0,    0,    0,    0,    0,    0,    0,    0,    0, 1000],
        [   0,    0,    0,    0,    0,    0,    0,    0,    0, 1000],
        [   0,    0,    0,    0,    0,    0,    0,    0,    0, 1000],
        [   0,    0,    0,    0,    0,    0,    0,    0,    0, 1000],
        [   0,    0,    0,    0,    0,    0,    0,    0,    0, 1000]])
```

Many of the confusion matrices for these models that wouldn't train looked like this, although the "guess" the machine would make would change between epochs. My suspicion is that this signifies too high of a learning rate, but upon testing I found that changing the learning rate had no effect on the outcome of the model.

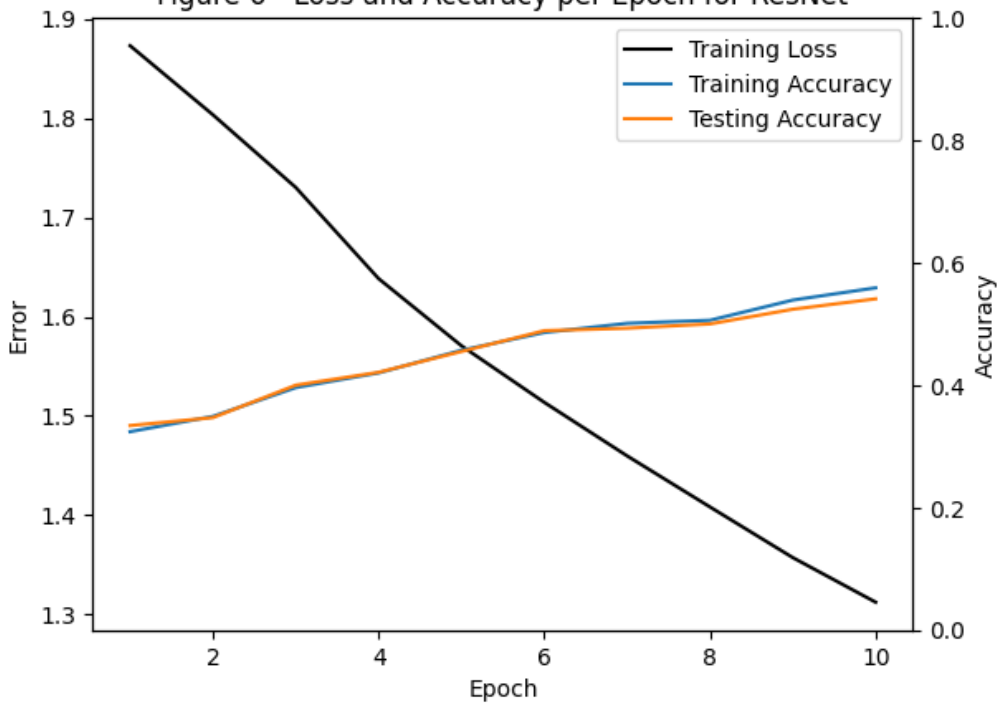
<u>Problem 3:</u>

A) After encountering issue after issue, I was relieved when the ResNet models all trained very easily. The ResNet-18 model had a smaller model size and computation complexity compared to

the GoogLeNet but was larger than the VGG-11 model. ResNet also trained faster than the GoogLeNet and VGG, making it the clear winner of this competition.

```
class ResNet(nn.Module):
    def __init__(self, arch, num_classes=10):
        super(ResNet, self).__init__()
        self.stem = nn.Sequential(nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                                  nn.LazyBatchNorm2d(),
                                  nn.ReLU(),
                                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        blks = []
        for i, b in enumerate(arch):
            blks.append(block(*b, first_block=(i==0)))
        self.blks = nn.Sequential(*blks)
        self.head = nn.Sequential(nn.AdaptiveAvgPool2d((1, 1)),
                                  nn.Flatten(),
                                  nn.LazyLinear(num_classes))

    def forward(self, x):
        out = self.stem(x)
        out = self.blks(out)
        out = self.head(out)
        return out

model_6 = ResNet(arch=((2, 64), (2, 128), (2, 256), (2,512))).to(device=try_gpu())
optimizer_6 = optim.SGD(model_6.parameters(), lr=0.01)
model_6.eval()
```

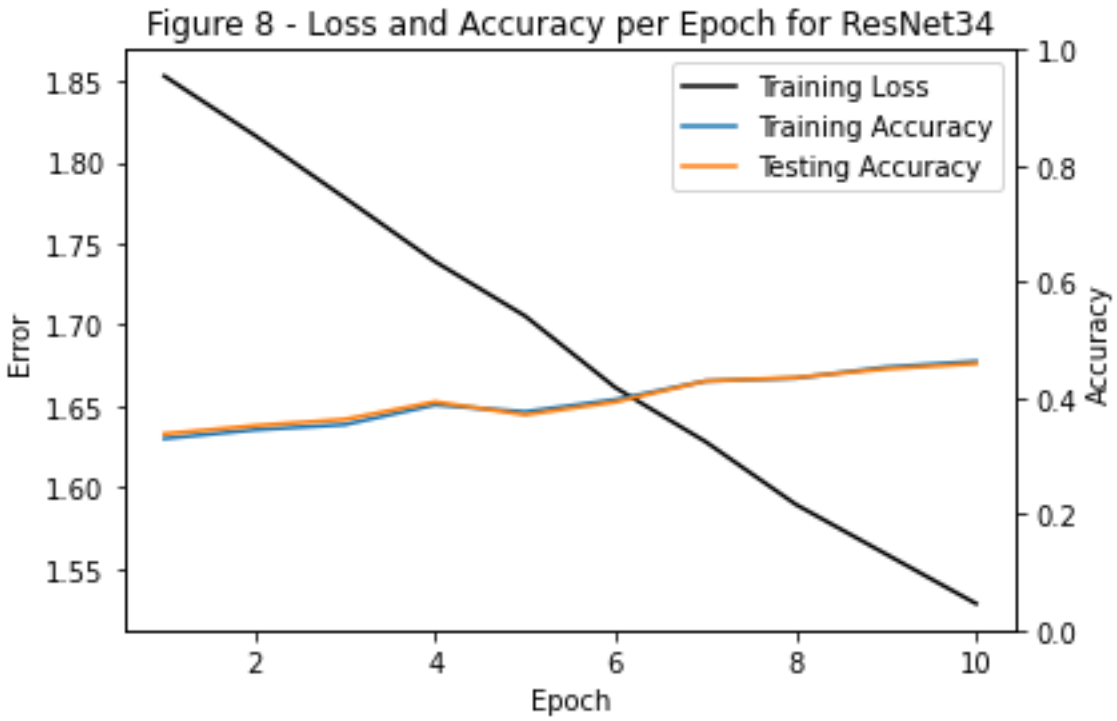
Figure 6 - Loss and Accuracy per Epoch for ResNet



```
Epoch 10:
        Duration = 255.405 seconds
        Training Loss: 1.31234
        Training Accuracy: 0.559
        Validation Accuracy: 0.541
tensor([[659,  40,  44,  11,  31,  15,  12,  19, 114,  55],
        [ 37, 741,   8,   6,   3,  13,   5,   5,  27, 155],
        [ 83,  28, 377,  22, 202, 126,  56,  59,  22,  25],
        [ 28,  23,  74, 172, 102, 336, 129,  71,  22,  43],
        [ 51,  16,  95,  24, 515,  92,  73, 109,  13,  12],
        [ 20,  14,  78,  50,  81, 561,  61,  96,  17,  22],
        [ 18,  18,  56,  42, 160,  67, 578,  31,   8,  22],
        [ 40,  13,  25,  25,  88, 141,  27, 585,  12,  44],
        [190,  85,  24,  15,  10,  19,   6,  13, 589,  49],
        [ 60, 187,  12,   5,   9,  17,  12,  28,  40, 630]])
```

B) I used https://paperswithcode.com/lib/timm/resnet# to find templates for ResNet-34 and ResNet-26. ResNet-34 was similar to ResNet-18, not only in structure but also in accuracy and training time. However, like with most other models which aim to add complexity to simpler models, ResNet-34 did not improve on the ResNet-18 in any way. It was slower to train, less accurate, and more computationally and storage heavy.

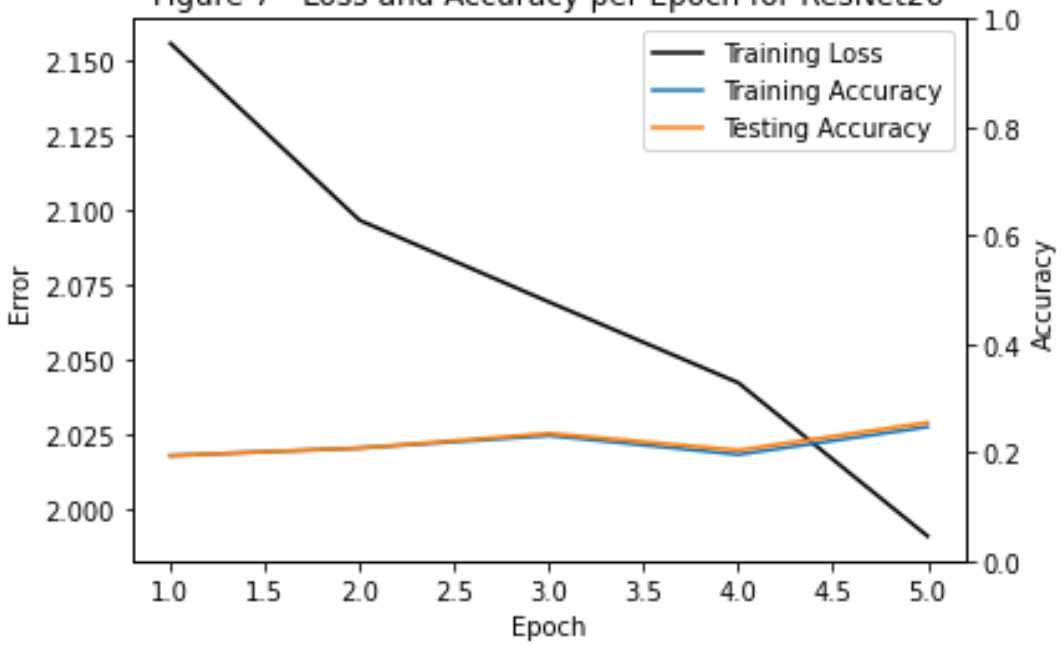
```
class ResNet34(nn.Module):
    def __init__(self, arch, num_classes=10):
        super(ResNet34, self).__init__()
        self.stem = nn.Sequential(nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                                  nn.LazyBatchNorm2d(),
                                  nn.ReLU(),
                                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
        blks = []
        for i, b in enumerate(arch):
            blks.append(block(*b, first_block=(i==0)))
        self.blks = nn.Sequential(*blks)
        self.head = nn.Sequential(nn.AdaptiveAvgPool2d((1, 1)),
                                  nn.Flatten(),
                                  nn.LazyLinear(num_classes))

    def forward(self, x):
        out = self.stem(x)
        out = self.blks(out)
        out = self.head(out)
        return out

model_8 = ResNet34(arch=((3, 64), (4, 128), (6, 256), (3,512))).to(device=try_gpu())
optimizer_8 = optim.SGD(model_8.parameters(), lr=0.01)
model_8.eval()
```




```
        Duration = 67.946 seconds
        Training Loss: 1.52837
        Training Accuracy: 0.463
        Validation Accuracy: 0.459
tensor([[388,  71,  12,  15,  19,  25,  13,  41, 303, 113],
        [ 19, 610,   4,  15,   0,  12,  15,  22,  62, 241],
        [ 61,  40, 141,  68, 171, 147, 126, 147,  59,  40],
        [ 20,  26,  26, 278,  37, 266, 128, 128,  34,  57],
        [ 40,  18,  31,  62, 296, 106, 160, 223,  40,  24],
        [ 10,  24,  31, 128,  42, 442,  70, 183,  37,  33],
        [  2,  17,  17, 123,  83,  51, 563,  90,  10,  44],
        [ 12,  15,   6,  58,  28, 114,  33, 629,  21,  84],
        [ 76,  87,   4,  30,   7,  20,   7,  19, 623, 127],
        [ 17, 183,   5,  18,   0,  17,  16,  49,  73, 622]])
```

ResNet-26 was an entirely different beast altogether. Not only did it not train to become as accurate as 18 or 34, but also the training time was the longest out of any of the models during this homework. As such, I cannot see a reason to ever recommend using ResNet-26 over ResNet-18.

Figure 7 - Loss and Accuracy per Epoch for ResNet26




```
        Duration = 319.756 seconds
        Training Loss: 1.99086
        Training Accuracy: 0.247
        Validation Accuracy: 0.255
tensor([[729,   0,  15,   0,  28,  58,  10,  79,  39,  42],
        [385,   0,  10,   0,   8,  49,  12, 224,  38, 274],
        [218,   0,  46,   0, 456, 134,  32,  96,   6,  12],
        [148,   0,  39,   0, 297, 230,  72, 168,   8,  38],
        [110,   0,  22,   0, 598, 137,  58,  61,   7,   7],
        [126,   0,  33,   0, 330, 282,  71, 123,   5,  30],
        [ 59,   0,  19,   0, 510, 133, 163, 108,   1,   7],
        [144,   0,  47,   0, 242, 183,  46, 290,   2,  46],
        [642,   0,  15,   0,  20,  57,   1,  58, 129,  78],
        [403,   0,  10,   0,  11,  56,  10, 149,  53, 308]])
```