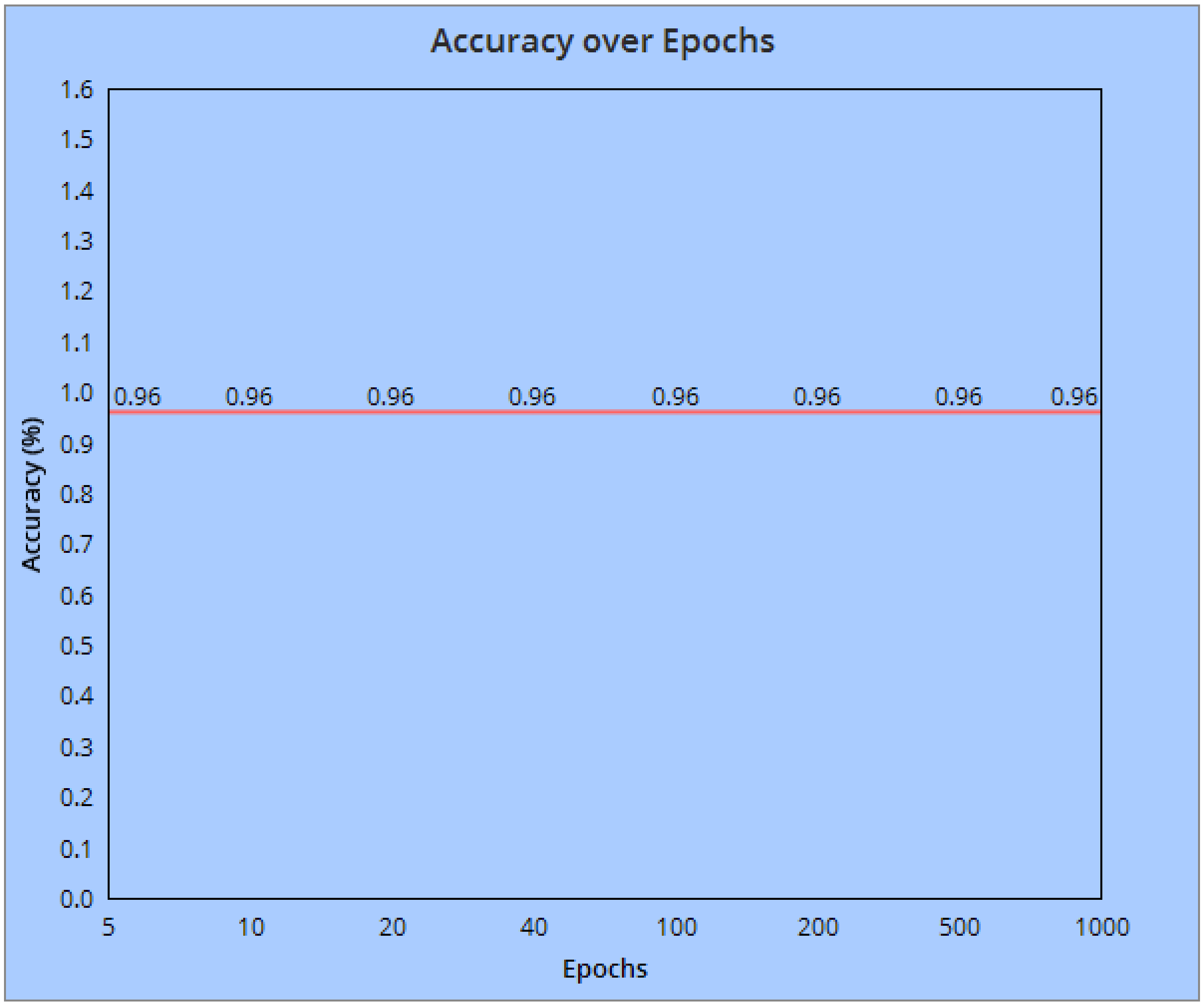
COS 429: Assignment 4

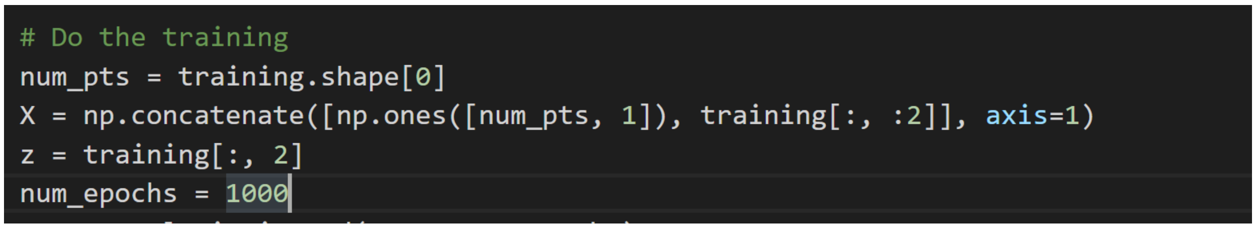
Austin Mejia and Matthew Hetrick

Part 1

Graph plotting accuracies over epochs:



Modified code snippet: we changed the # of epochs to get our data, this snippet has 1000:

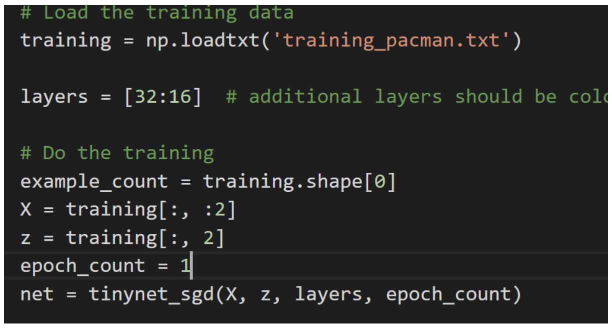


```
# Do the training
num_pts = training.shape[0]
X = np.concatenate([np.ones([num_pts, 1]), training[:, :2]], axis=1)
z = training[:, 2]
num_epochs = 1000
```

Part 2

- Do additional layers always help?
    - Additional layers help up to a certain point. After about two layers, you start overfitting the data and are fitting noise within the input to the second+ layers. This also takes longer to converge and sometimes won't reach convergence.
- What about more neurons in each layer?
    - More neurons help to fit more parameters. So for one hidden layer, more neurons can help up to a certain point to ensure all input parameters are dealt with. But too many neurons will again result in overfitting the data. This also takes longer to converge and sometimes won't reach convergence.
- How does the number of epochs and learning rate affect the network?
    - Based on the trial, the number of epochs only slightly increased the performance of the network (an average of 96% accuracy to an average of 96.6% accuracy for 3 trials for 10 epochs vs 2). 2 epochs run faster though.
    - Based on the trial, larger learning rates begins to slightly decrease the accuracy of the network, and at around a rate of 5/epochs, it failed to converge. A smaller learning rate, .01, performed ever so slightly better but took longer to reach convergence. Therefore, too small of a learning rate would imply a failure to reach convergence. Based on the tests, an optimal learning rate would be somewhere between .001/epochs and .1/epochs depending on the user's preference for faster or more accurate convergence.
- Try a network with a constant number of neurons (i.e. 6,6,6) versus one with an hourglass shape (i.e. 9,6,3). Does one exhibit better accuracy or convergence properties than the other?
    - The hourglass network seems to have better average accuracy and more consistent convergence than the constant, as, over a set of 5 trials, the hourglass network averaged 97.6% accuracy with great consistency, and the constant averaged a 93.4% accuracy with significant fluctuations (97% to 86%).

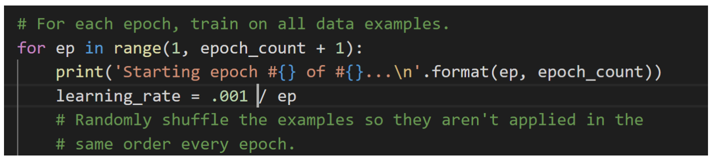
Modified code snippets: Left for layers and epochs, right for learning rate



```
# Load the training data
training = np.loadtxt('training_pacman.txt')

layers = [32:16]  # additional layers should be col

# Do the training
example_count = training.shape[0]
X = training[:, :2]
z = training[:, 2]
epoch_count = 1
net = tinynet_sgd(X, z, layers, epoch_count)
```



```
# For each epoch, train on all data examples.
for ep in range(1, epoch_count + 1):
    print('Starting epoch #{} of #{}...\n'.format(ep, epoch_count))
    learning_rate = .001 / ep
    # Randomly shuffle the examples so they aren't applied in the
    # same order every epoch.
```

Part 3

Q1: The first dimension of x is 3 because the image has three channels (RGB).
Q2:

- Y will have dimensions K x U_1 - F_1 + 1 x U_2 - F_2 + 1. This is because the filter is zero-padding with a stride of 1, causing the overall resolution to shrink.
- u + v indicates the location of a pixel on a certain layer. You can add them because, practically, an array can be conceptualized as one large list and the indexes can be summed to access values. C and K represent channels and layers respectively. As they are separate from each other, we cannot simply sum them to access a specific channel on a specific layer. Also, note how *x* does not have a "k" as it only applies to the filters.

Q3: The responses are strong despite a random w because both the image and the Gaussian filters are three dimensions "deep" (ie, both have an RGB layer). Each images is a composite of 3 different convolutions (an RGB channel and the matching RBG channel in the Gaussian). This convolution response is strong enough that random Gaussians can be applied.
Q4:

- The filter implemented is a discrete approximation of the Laplacian Filter, used for edge detection. It highlights the edges because the sum of the kernel is zero.
- The RGB channels are processed as three separate intensity maps with the same filter. As each RGB channel has it's own intensities for each pixel, these intensities can be convoluted with the filter separate from other channels.
- The filter detects the edges of the elements, as well as significant changes in features within an element. These features are seen best with color differences.

Q5: Nonlinear activation functions allow for non-linearity in a network, and can model variables that have non-linear behavior. Without non-linearity, we can only map inputs to outputs linearly, which is neither robust nor effective. Moreover, non-linearity is needed in activation functions because we want a nonlinear decision boundary, accomplished by having non-linear combos of inputs and weight vectors.
Q6: As described at the start of part 1.3, the image is the result of a max pooling function. A max pooling will identify a region of pixels, take the highest pixel of each region, and compose a new image comprised of just these highest pixels. The resulting image of the max_pool2d function is the reduced image comprised of the maximal pixels. The three images represent the three different channels of the image (RGB).
Q7: The output derivatives describe the gradient of the parameters in the network. Given this, they must have the same size as described by the derivations in the chain rule.
Q8:

- By implementation, delta is a random sampling of values. That said, within the code it serves as a hypothetical change in x between two steps of z(x).

- dzdx_numerical is a mathematical approach to finding the derivative of z with respect to x. This is computed through backpropagation and is an iterative method to reduce error and results in a more exact answer.
- dzdx_analytical could be seen as a Taylor Series approximation, representing the difference between two steps of dz/dx. This analytical approach is faster but sometimes less accurate. This is similar to why we use a numerical gradient calculation in gradient descent so that we catch analytical errors.

Q9: dzdx3 represents the derivative of the output of the CNN. Each “x” denotes an input/output of a layer, with x3 being the final output. It should be noted that this derivative is computed after a pooling layer is applied to the output of a convolutional layer.
Q10:

- If λ=0, the lose from regularization is the only element that’s affected. If $E(\mathbf{w},b)$=0 as well, we know that all positive pixels have a score of at least 1 and all negative pixels have a score of at most zero. This would be our intended effect
- The margin is (λ/2) ‖$\mathbf{w}$‖^2

Q11:

- Based on the equation, a momentum rate greater than one would mean that there is no momentum at all, and in fact result in negative momentum. Any negative changes should be reflected in the gradient, not the momentum, otherwise it would flip the direction. A momentum rate close to 1 would accelerate the learning rate.
- A large learning rate will adjust for the error too much and thus never converge. An inappropriately large learning rate will overshoot convergence and infinitely grow instead.

Q12: Although the image itself is larger than 32x32, the actual elements that we are attempting to identify are characters, which can be appropriately captured with a 32x32 patch. This allows us to train a model much more quickly and apply it to larger inputs.

<u>Part 4</u>

Task 0) Ran the code to train the data set

Task 1)
Best class accuracy was frog, avg was ship, worst was cat. Attached are screenshots of a correct and incorrect classification for each of the three aforementioned classes.

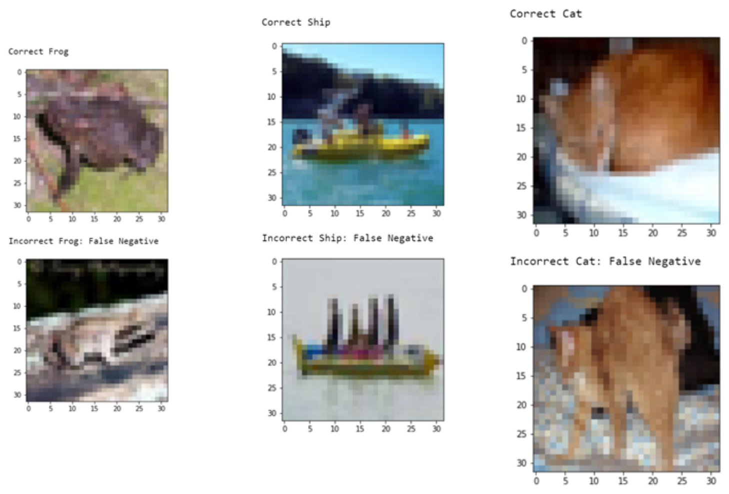

Correct Frog
Correct Ship
Correct Cat
Incorrect Frog: False Negative
Incorrect Ship: False Negative
Incorrect Cat: False Negative

Figure 2: Class Accuracy Examples

The best-classified example according to the softmax score was of a horse with a score of .994; the worst classification with a score of .187 was of a bird. An average example had a score of .64, which was a cat. The highest score for an incorrectly classified image was .84 of a plane that was classified as a ship. Below are these examples:

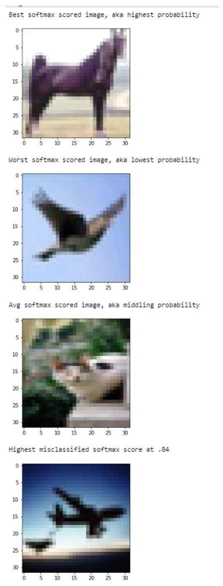
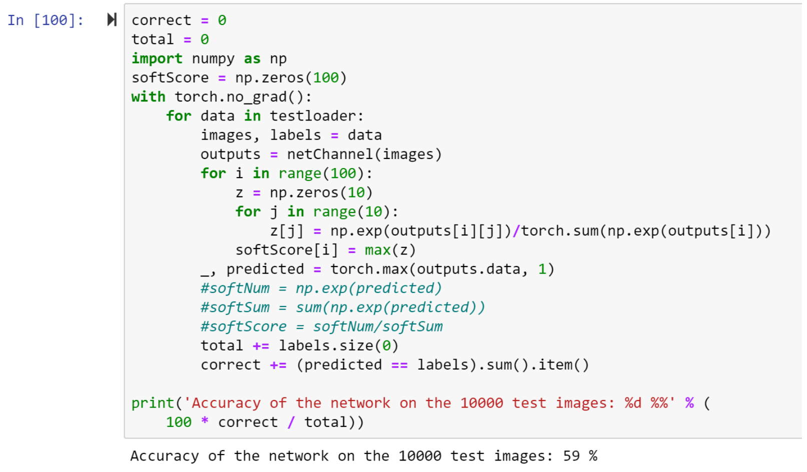

Best softmax scored image, aka highest probability
Worst softmax scored image, aka lowest probability
Avg softmax scored image, aka middling probability
Highest misclassified softmax score at .84

Figure 4: Examples of Best/Worst/Avg Softmax

We got the softmax scores by calculating the softmax score of each class for each example, and then selecting the max of these scores so we can compare which are the ‘best’ and ‘worst’. Note that the commented-out code was just something we were implementing before we figured out how to actually do it.

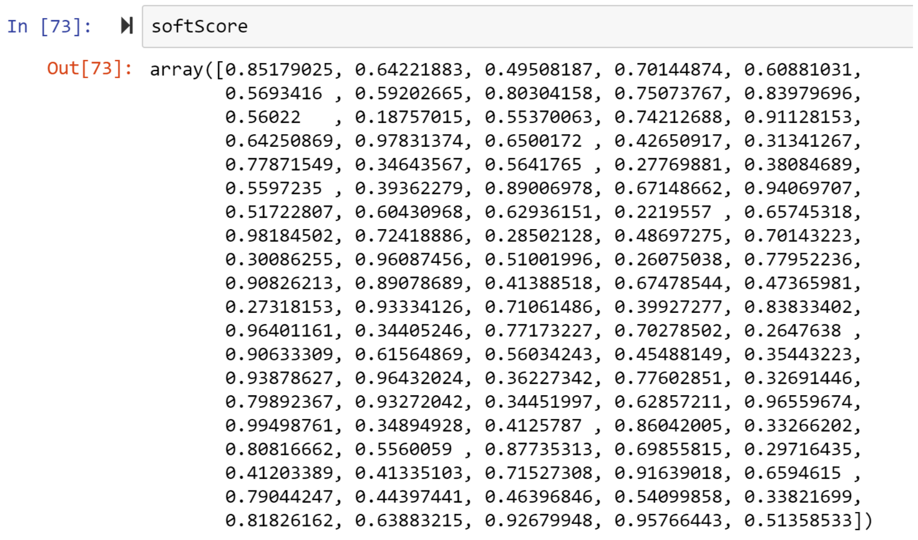


```
In [100]:
correct = 0
total = 0
import numpy as np
softScore = np.zeros(100)
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = netChannel(images)
        for i in range(100):
            z = np.zeros(10)
            for j in range(10):
                z[j] = np.exp(outputs[i][j])/torch.sum(np.exp(outputs[i]))
            softScore[i] = max(z)
        _, predicted = torch.max(outputs.data, 1)
        #softNum = np.exp(predicted)
        #softSum = sum(np.exp(predicted))
        #softScore = softNum/softSum
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))

Accuracy of the network on the 10000 test images: 59 %
```

Figure 5: Calculating softmax scores



```
In [73]:  softScore

Out[73]: array([0.85179025, 0.64221883, 0.49508187, 0.70144874, 0.60881031,
                0.5693416 , 0.59202665, 0.80304158, 0.75073767, 0.83979696,
                0.56022   , 0.18757015, 0.55370063, 0.74212688, 0.91128153,
                0.64250869, 0.97831374, 0.6500172 , 0.42650917, 0.31341267,
                0.77871549, 0.34643567, 0.5641765 , 0.27769881, 0.38084689,
                0.5597235 , 0.39362279, 0.89006978, 0.67148662, 0.94069707,
                0.51722807, 0.60430968, 0.62936151, 0.2219557 , 0.65745318,
                0.98184502, 0.72418886, 0.28502128, 0.48697275, 0.70143223,
                0.30086255, 0.96087456, 0.51001996, 0.26075038, 0.77952236,
                0.90826213, 0.89078689, 0.41388518, 0.67478544, 0.47365981,
                0.27318153, 0.93334126, 0.71061486, 0.39927277, 0.83833402,
                0.96401161, 0.34405246, 0.77173227, 0.70278502, 0.2647638 ,
                0.90633309, 0.61564869, 0.56034243, 0.45488149, 0.35443223,
                0.93878627, 0.96432024, 0.36227342, 0.77602851, 0.32691446,
                0.79892367, 0.93272042, 0.34451997, 0.62857211, 0.96559674,
                0.99498761, 0.34894928, 0.4125787 , 0.86042005, 0.33266202,
                0.80816662, 0.5560059 , 0.87735313, 0.69855815, 0.29716435,
                0.41203389, 0.41335103, 0.71527308, 0.91639018, 0.6594615 ,
                0.79044247, 0.44397441, 0.46396846, 0.54099858, 0.33821699,
                0.81826162, 0.63883215, 0.92679948, 0.95766443, 0.51358533])
```

Figure 6: Softmax scores

Task 2:

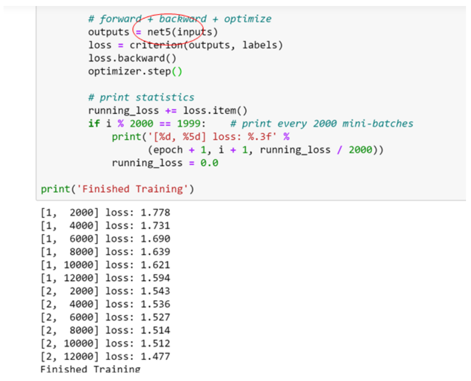
Upon training for 2 more epochs, the loss converged to a value of 1.105. Overall classification accuracy went up to 58%. Poorly classified classes went up in accuracy except for bird, which decreased, whereas well-classified classes decreased in accuracy except for car. Frog, the highest, went from 84% to 62%. Cat and deer, the two lowest, went from 11% and 24% to 44% and 46%, respectively. This shows that in training for more iterations, the network tries to minimize the really small losses, even at the sacrifice of the really high accuracies. Therefore we would predict that with more iterations in training, car would decrease but bird and dog, the new two lowest-accuracy classes, would increase. Additionally, the examples from the previous training

Task 3:

- With a learning rate of .01, the network output a less minimized loss (1.94) which was converted to in the second mini-batch.
- For a learning rate of .1, the network output a loss of 2.36, which was converged to in first mini-batch.
- With a learning rate of .5, it never converges.
- With a learning rate of .0001, the network did not reach convergence until the 10th epoch at a value of around 1.07, lower than the loss of the first learning rate of .001, but it took much longer.
- With a learning rate of .005, the network output a loss of 1.60, which converged to in the 4th mini-batch in the 2nd epoch.

Based on these, a reasonable learning rate is somewhere between .01 and .0001 depending on how important fast convergence is and how minimized you want your error to be.

Attached is are screenshots of the modified optimizer variable for a learning rate of .0001 and the associated output for 4 epochs. We edited the code by making a new net for each learning rate and changing the learning rate in the optimizer variable.

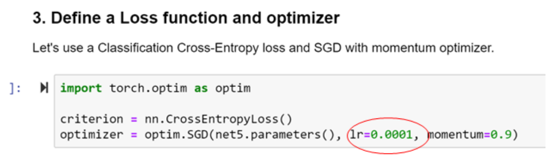


```
        # forward + backward + optimize
        outputs = net5(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

```
[1,  2000] loss: 1.778
[1,  4000] loss: 1.731
[1,  6000] loss: 1.690
[1,  8000] loss: 1.639
[1, 10000] loss: 1.621
[1, 12000] loss: 1.594
[2,  2000] loss: 1.543
[2,  4000] loss: 1.536
[2,  6000] loss: 1.527
[2,  8000] loss: 1.514
[2, 10000] loss: 1.512
[2, 12000] loss: 1.477
Finished Training
```



3. Define a Loss function and optimizer

Let's use a Classification Cross-Entropy loss and SGD with momentum optimizer.

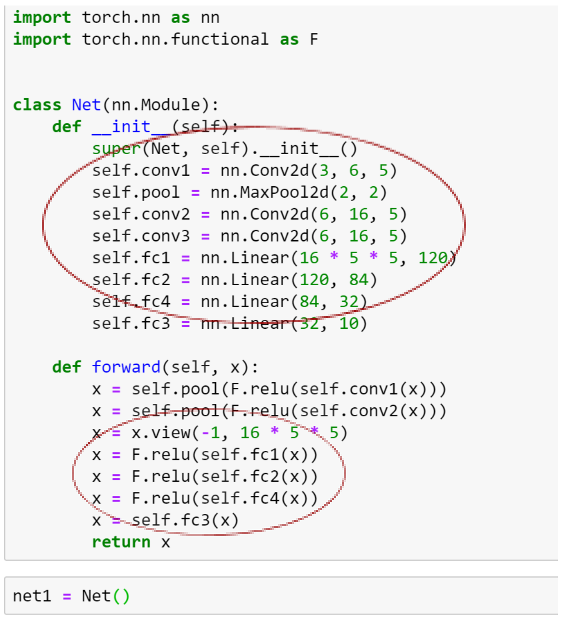
```
]:  import torch.optim as optim

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(net5.parameters(), lr=0.0001, momentum=0.9)
```

Figure 1: Code snippets of learning rate of .0001 and output for 4 epochs

Task 4:

- Adding conv and relu layers with padding and without pooling:
    - We first added one conv+relu layer and it did not increase accuracy, as it decreased to 56% from the 58% we reached with the first couple of tasks.



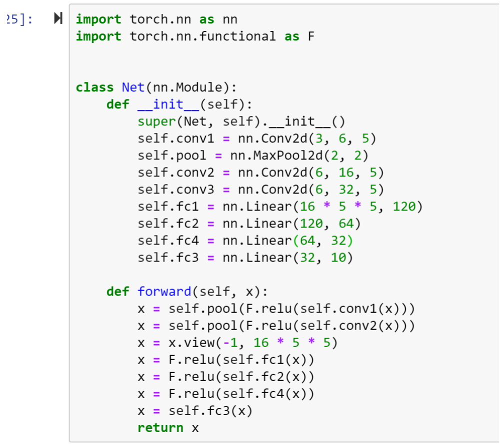
```python
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.conv3 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc4 = nn.Linear(84, 32)
        self.fc3 = nn.Linear(32, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc4(x))
        x = self.fc3(x)
        return x
```

```python
net1 = Net()
```

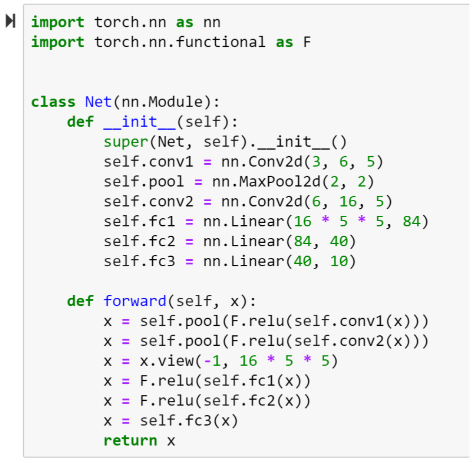
- Changing Channels:
    - Then we kept the added conv+relu layer and played with the channel input/outputs of each layer and eventually deleted the third layer. We've attached some clips below; the left had a 57% accuracy and the right had a 59% accuracy. This shows that more layers are not necessarily better!



```python
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.conv3 = nn.Conv2d(6, 32, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 64)
        self.fc4 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc4(x))
        x = self.fc3(x)
        return x
```



```python
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 84)
        self.fc2 = nn.Linear(84, 40)
        self.fc3 = nn.Linear(40, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```