PySuperTuxKart Optimization

By: Chase Maivald, Nafis Abeer, and Rajiv Ramroop

It is worth noting that for this project, we were able to utilize a Shared Computing Cluster which cut down our training times by a large amount. For this reason, our first approach is to try out a variety of training conditions and test if any of them yield a greater performance for little Mario in PySuperTuxKart. These conditions include first testing performance from the professor's controller vs. our own controller and selecting a "control" controller.py. Next, we will find out what is the best dataset to train our model on. We will then try training with a variety of layers by tuning planner.py and see if that has an impact on performance. Our next portion of experimentation includes coming up with a better controller algorithm from conclusions we draw in the previous approach, since we will have an idea of which dataset and model is optimal for testing. Finally we will try an approach that involves Deep-Q-Learning. The performance times from each trial will be reported, and any conclusions drawn from the trials will be listed under the results.

From our experimentation in homework 5 we had determined that our own controller had returned faster track times compared to the HW4 solution controller. For this reason, we are going with our controller and planner from homework 5 to test for an optimal dataset. We will call the aggregated images and testing on the source of images the "optimizing dataset" section. This section involves training on 10000 images vs. 15000 images vs. 25000 images and training on just the 3 tracks with the longest times to completion. The results of these experiments are listed below. For each track test we did two runs and took the best time of the two.

Optimizing Dataset

10000 images, 20000 steps per track and homework 5 planner.py, solution controller

Fig 1.1	Zengarden	Lighthouse	Hacienda	Snow tux peak	Cornfield_ crossing	Scotland
Time to Finish	45.4 seconds	46.3 seconds	51.4 seconds	57.0 seconds	76.1 seconds	63.2 seconds

Final Loss: 0.085 Average times: 56.5 seconds

10000 images, 20000 steps per track and homework 5 planner.py, our own controller - retrained (control group)

Fig 1.2	Zengarden	Lighthouse	Hacienda	Snow tux peak	Cornfield_ crossing	Scotland
Time to Finish	43.8 seconds	43.6 seconds	54.5 seconds	62.2 seconds	70.6 seconds	52.7 seconds

Final Loss: 0.086 Average times: 54.56 seconds

15000 images, 30000 steps per track and homework 5 planner.py, our own controller

Fig 1.3	Zengarden	Lighthouse	Hacienda	Snow tux peak	Cornfield_ crossing	Scotland
Time to Finish	40.1 seconds	45.9 seconds	50.8 seconds	72.7 seconds	70.9 seconds	53.5 seconds

Final Loss: 0.080 Average times: 55.65 seconds

15000 images, 20000 steps per track and homework 5 planner.py, our own controller

Fig 1.4	Zengarden	Lighthouse	Hacienda	Snow tux peak	Cornfield_ crossing	Scotland
Time to Finish	47.8 seconds	44.1 seconds	57.1 seconds	58.8 seconds	67.8 seconds	57.4 seconds

Final Loss: 0.095 Average times: 55.5 seconds

From this initial experimentation, before trying 25000 images, a few conclusions can be drawn.

1. I (Nafis) retrained my own solution to assignment 4 and the solution controller and my assignment 4 controller was chosen as the control group for these experiments. My controller seems to perform a bit worse than Chase's due to its use of the (abs) function, which might add minor delays, so for the next portion when we test different planners, we will use Chase's controller.py solution to assignment 4.
2. No matter which controller is used, if the number of images and number of steps per track are kept the same, the loss seems to remain steady, as the planner has had no

changes made. We can further test this with Rajiv's controller since he was in a different group and had developed his own algorithm

3. Initial increase in number of images and number of steps per track only improved performance for two tracks (Zengarden and Hacienda) even though the loss seemed to have decreased. This decrease in loss was reflected by cornfield_crossing where previously the controller had performed badly in the descending section. It did seem that the kart was overdoing some turns.
4. Increasing the number of images but keeping the number of steps per track at 20000 increased the loss. The performance was worse than the control group for four of the tracks, which had the same number of steps but less images. This suggests that the number of steps should be increased proportionally to the number of images. Interestingly enough, the kart seemed to accelerate a lot faster in this scenario.
5. Seeing how there hasn't been a great deal of improvement in performance from 15000 images, we will also try reducing the number of images, along with the number of steps per track as an added experiment. The results will be reported below.

25000 images, 50000 steps per track and homework 5 planner.py, our own controller

Fig 1.5	Zengarden	Lighthouse	Hacienda	Snow tux peak	Cornfield_ crossing	Scotland
Time to Finish	42.0 seconds	47.9 seconds	53.2 seconds	47.8 seconds	69.2 seconds	53.0 seconds

Final Loss: 0.081 Average times: 52.1 seconds

Although some tracks have a bit slower time, the consistency in maintaining center of track seems to have tremendously improved, indicating that training on a larger set of images with more steps per image has its benefits in performance. The kart also handled increases in elevation just fine as indicated by its performance in Cornfield_crossing and Hacienda. Snowtuxpeak was especially impressive as the kart seamlessly handled sharp corners, which indicated that the extra images helped with turns. The kart did not seem to overdo any turns. It is worth noting that gathering images and training for this experiment took a significant amount of time. For the future we will need to find a middle ground between #images and training time.

5000 images, 10000 steps per track and homework 5 planner.py, our own controller

Fig 1.6	Zengarden	Lighthouse	Hacienda	Snow tux peak	Cornfield_ crossing	Scotland
Time to Finish	43.7 seconds	43.5 seconds	DNF	53.5 seconds	68.9 seconds	53.1 seconds

Final Loss: 0.093 Average times: 52.54 seconds*

Training on this dataset does not handle sudden changes in elevation well (Hacienda has a sudden increase in elevation). It handles soft turns well, but sharp turns not so much. The performance in Cornfield_crossing should be viewed as an outlier due to the fact that all the traps were somehow avoided. The first run in Cornfield_crossing was in 84 seconds, indicating that reducing images makes the model perform worse. From these sets of tests, we may assume that increasing the number has its benefits, but not enough to increase all the way to 25,000 images since training takes too long, and the benefits are minimal in comparison.

10000 images, 20000 steps per track and homework 5 planner.py, our own controller - only trained on images from Hacienda, Snowtuxpeak and Cornfield_Crossing

Fig 1.7	Zengarden	Lighthouse	Hacienda	Snow tux peak	Cornfield_ crossing	Scotland
Time to Finish	DNF	DNF	50.2 seconds	52.9 seconds	70.3 seconds	DNF

Final Loss: 0.079 Average times: 57.8 seconds*

Loss and performance are unrelated! If it was not trained on a track, it will not know what it is doing on said track, currently. Did well on Hacienda, Snowtuxpeak and Cornfield_crossing though. Sure it still does not handle sharp turns too well, we need to train the planner to anticipate those. It also still does not do too well with rise in elevation in those tracks, probably because we never incorporated aim_pointer[1]. But seeing how the 2 tracks that finished faster had double the images and steps collected compared to the control group, and based on our findings from aggregated images, we conclude this section of experimentation by choosing to train on all the tracks with 20000 images and 40000 steps per track to go with those images.

Improving Planner.py

Our expectation is that once the neural network is optimized for learning, the most feature dense parts of the track will be the track itself, which should make the aim_pointer on it easier to follow for little Mario. Training should be sensitive to changes in the aim_pointer, but we must avoid overtraining the model to avoid being too sensitive to changes in the aim_pointer.

In order for planner.py to work, utils.py converts an image to a tensor such that a pytorch deep learning network in planner.py can use the datatype tensor for the input and output channels. This utils.py generates labels for our training data drive_data with the pystk state instance of aim_point being fed into every kart instance, to constantly modify the steering aim point of the kart. The first input channel of planner.py is size 3 because of the three colors in the RGB color wheel. The images of the race track are in color.

Our first neural network has a similar architecture to a deep Q network from the Pytorch documentation.[1] It includes the same number of convolutional layers, batch normalization layers, and nonlinearity layers, except our model also contains three pooling layers. The main purpose of the pooling layers is to reduce the number of parameters to reduce overfitting, extract representative features from the input tensor, and reduce computation to aid efficiency. In a convolutional neural network, output feature maps are sensitive to the location of features in the input. So, small movements in the position of a feature in an input image will result in a different feature map. To address this sensitivity problem, the down sampling ability of pooling layers creates lower resolution of the input which still contains the most important structural features of the original image without the fine details that may not be useful for the task of learning. Each convolution is a small filter slid over a whole image of the race track. It's a dot product over all pixels, extracting our features from each image. The network is able to learn optimal filters given the input constraints. Filters are initialized at random and are location invariant, meaning they can locate features anywhere in the image. Research has shown that smaller filters generally perform better, so the kernel size and size of input/output channels are not too large. Notice the preference for an odd numbered n x n kernel size. For an odd sized filter, all the previous layer pixels are symmetrically around the output pixel. Without this symmetry, it is necessary to account for distortions across the layers when using an even sized kernel. Therefore, odd numbered kernel sizes are preferred for their implementation simplicity to interpolate a center

[1] https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

pixel. 1x1 filters are ignored due to being fine grained and local, and for having no information from the neighboring pixels. Therefore, a 1x1 kernel would not possibly achieve an ideal amount of feature extraction. Each convolutional layer and max pooling layer is 0-padded, which allows us to also capture the edges of the image when applying a convolutional filter. This is especially useful when a sharp turn is approaching but is barely in the viewable screen, in any direction. Each of the four 2D convolutional layers is followed by 2D batch normalization, then a ReLU activation function to ensure all the tensors are positive, and it scales data in the benefit of speeding up time to train each layer. Batch normalization is used to standardize the inputs of the model, each normalization layer is after each convolutional layer and before each nonlinearity. It allows the model to converge faster in training and therefore use much higher learning rates because the model can adjust all of the weights at once instead of adjusting one at a time. Chase (I) designed this small kernel instead of a fully connected network to contrast the benefits of weight sharing and reduction in computational costs with fully connected networks. Designing the same kernel for different sets of pixels in an image causes the same weights to be shared across the pixels as they are convolved on. Having less weights than with fully connected layers means less weights to backpropagate on, which reduces training time, and seems to overtrain less than with fully connected layers are at the very end of the network. The following networks seek to best test how kernel size, different types of 2D pooling layers, having fully connected layers, or changing the network architecture affects performance and time to train.

Fig 2.1

Layers	**Occurrences (number of layers, ordered)**
2D Convolution	4x -- (input, output) (3,16), kernel size=3, stride=1,padding=0 , (16,32),kernel size=5 ,stride=1,padding=0 , (32,32),kernel size=7, stride=2,padding=0 , (32,16),kernel size=3, stride=1,padding=0
2D Batch normalization	4x -- (16x16) , (32x32) , (32x32), (16x16)
Nonlinearity (ReLU activation)	4x

2D Max Pooling	3x -- kernel size=3, stride=1, padding=0

Fig 2.2	Zengarden	Lighthouse	Hacienda	Snow tux peak	Cornfield _Crossing	Scotland	Cocoa temple
Time to Finish	44.6 seconds	48.0 seconds	51.1 seconds	61.7 seconds	76.1 seconds	53.3 seconds	84.0 seconds

Final loss: 0.078 Average time: 59.8 s

I thought steering was erratic due to overtraining, but it seems the opposite is actually true. The green aim pointer (prediction) does not always follow the red aim pointer (truth), which can lead to missing sharp turns. During straights, it is overly responsive though, it keeps turning side to side when the truth value is barely moving. The train.py using L1 regularization proactively addresses any overfitting concerns with a deep learning model. It is still useful to test whether other methods of pooling or if changing network architecture can produce better results. Firstly, we double check that max pooling is the best method of pooling by preserving the network architecture, I only changed the max pooling layers to average pooling layers.

Fig 2.3

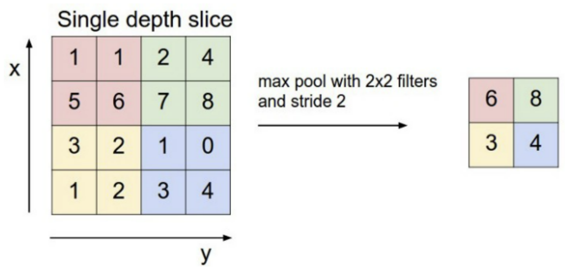
Layers	**Occurrences (number of layers ordered)**
2D Convolution	4x -- (input, output) (3,16),kernel size=3, stride=1,padding=0 , (16,32),kernel size=5, stride=1,padding=0 , (32,32),kernel size=7, stride=2,padding=0 , (32,16),kernel size=3, stride=1,padding=0
2D Batch normalization	4x -- (16x16) , (32x32) , (32x32), (16x16)
Nonlinearity (ReLU activation)	4x

2D Average Pooling	3x -- kernel size=3, stride=1, padding=0

Fig 2.4	Zengarden	Lighthouse	Hacienda	Snow tux peak	Cornfield_ Crossing	Scotland	Cocoa temple
Time to Finish	39.3 seconds	48.3 seconds	63.9 seconds	97.9 seconds	74.8 seconds	62.0 seconds	DNF

Final loss: 0.073

Average pooling follows the same bit reduction technique as max pooling (see Fig 2.3), except averaging all input pixels in its filter rather than taking the maximum pixel is noisier with the given dataset. In the future, it should be chosen early in the network because it preserves more information than max pooling. Average pooling occasionally has a faster time per track than max pooling, but average pooling generally performs about 20% worse than max pooling in terms of the average time to finish. The predicted aim pointer (green) is less responsive to following changes in the truth (red) aim pointer with average pooling instead of max pooling with the same number of max pooling or average pooling layers for each, and otherwise keeping the same model architecture intact. This demonstrates the ability of max pooling to not only yield a better time on average, but it also generalizes to finish cocoa temple blindly without utils.py creating any images of this track for planner.py to train on. Based on weight decay, the model is only flexible enough to generalize to finish the unseen track, cocoa temple, with 2D max pooling layers because it is a more optimal dimensionality reduction method.



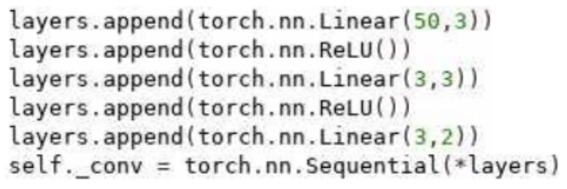

```
layers.append(torch.nn.Linear(50,3))
layers.append(torch.nn.ReLU())
layers.append(torch.nn.Linear(3,3))
layers.append(torch.nn.ReLU())
layers.append(torch.nn.Linear(3,2))
self._conv = torch.nn.Sequential(*layers)
```

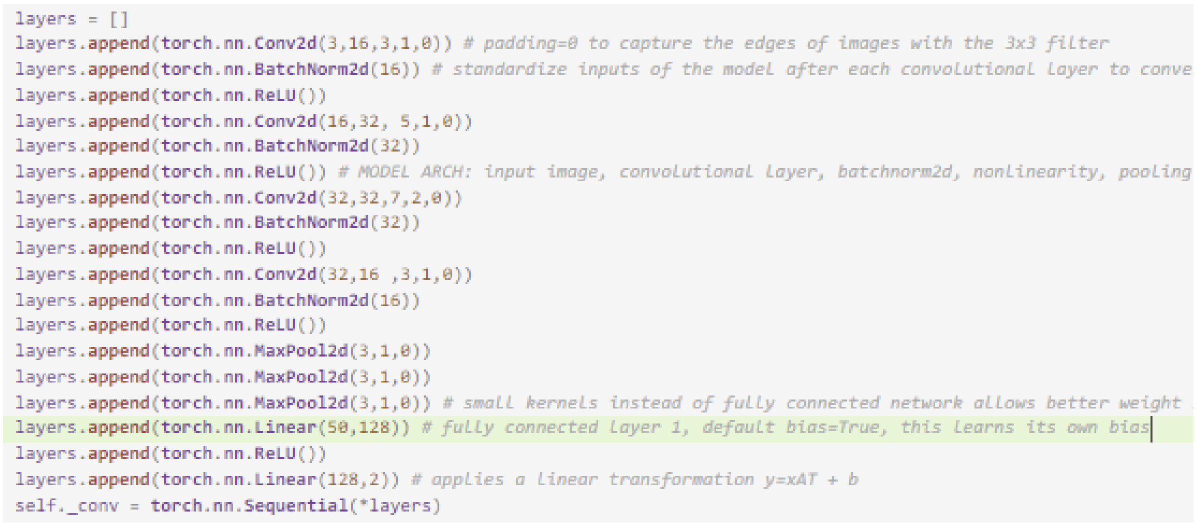
Fig 2.5

Layers	**Occurrences (number of layers ordered)**
2D Convolution	4x -- (input, output) (3,16),kernel size=3, stride=1,padding=0 , (16,32),kernel size=5, stride=1,padding=0 , (32,32),kernel size=7, stride=2,padding=0 , (32,16),kernel size=3, stride=1,padding=0
2D Batch normalization	4x -- (16x16) , (32x32) , (32x32), (16x16)
Nonlinearity (ReLU activation)	4x
2D Max Pooling	3x -- kernel size=3, stride=1, padding=0
2D Linear	3x -- (50x3) (3x3) (3x2)

Fig 2.6	Zengarden	Lighthouse	Hacienda	Snow tux peak	Cornfield_ Crossing	Scotland	Cocoa temple
Time to Finish	46.9 seconds	40.9 seconds	DNF seconds	95.4 seconds	71.2 seconds	61.8 seconds	DNF seconds

Final loss: 0.077

Even with the loss being about the same as the original max pooling network in Fig 2.1, the performance is about 25% worse on average with the addition of just fully connected layers to decode our features and map them to their respective classes. Increasing the size of the channels in the fully connected layers with memory > 2^7 creates a kernel that takes more than an hour to train, a size greater than 2^14 is too large for the memory constraints of the shared computing cluster. The two fully connected linear layers were tested with size up to 2^7, and performance generally improved as we scaled 50x128 128x128 128x2 to 50x3 3x3 3x2. Anyway, it is shown through our research that a smaller kernel size performs better, this network with the kernel sizes 3,5,7,3,3,3,3 (^2) has better performance than with kernel sizes > 7. To improve our model architecture further, we evaluated current research in architectures such as in ResNet for design principles such as an encoder being the first ‘x’ many layers which increase feature size; a decoder being the last several linear layers which maps features to their respective classes, and general strategies to balance network depth and performance while keeping parameters as low as possible.[2]

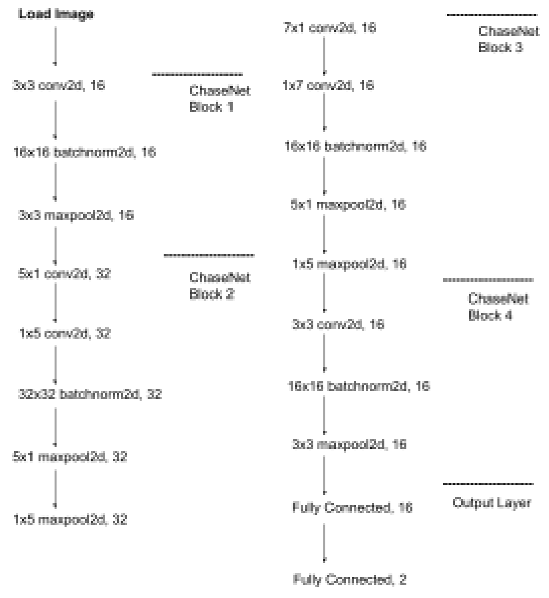


```
layers = []
layers.append(torch.nn.Conv2d(3,16,3,1,0)) # padding=0 to capture the edges of images with the 3x3 filter
layers.append(torch.nn.BatchNorm2d(16)) # standardize inputs of the model after each convolutional layer to conve
layers.append(torch.nn.ReLU())
layers.append(torch.nn.Conv2d(16,32, 5,1,0))
layers.append(torch.nn.BatchNorm2d(32))
layers.append(torch.nn.ReLU()) # MODEL ARCH: input image, convolutional layer, batchnorm2d, nonlinearity, pooling
layers.append(torch.nn.Conv2d(32,32,7,2,0))
layers.append(torch.nn.BatchNorm2d(32))
layers.append(torch.nn.ReLU())
layers.append(torch.nn.Conv2d(32,16 ,3,1,0))
layers.append(torch.nn.BatchNorm2d(16))
layers.append(torch.nn.ReLU())
layers.append(torch.nn.MaxPool2d(3,1,0))
layers.append(torch.nn.MaxPool2d(3,1,0))
layers.append(torch.nn.MaxPool2d(3,1,0)) # small kernels instead of fully connected network allows better weight
layers.append(torch.nn.Linear(50,128)) # fully connected layer 1, default bias=True, this learns its own bias
layers.append(torch.nn.ReLU())
layers.append(torch.nn.Linear(128,2)) # applies a linear transformation y=xAT + b
self._conv = torch.nn.Sequential(*layers)
```

Fig 2.7

[2] https://arxiv.org/pdf/1512.00567v1.pdf

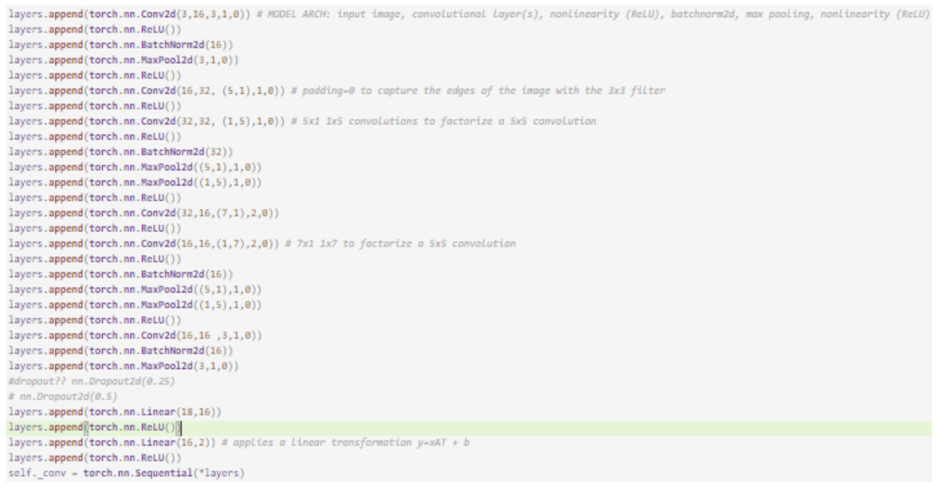
Fig 2.8



25000 images, 50000 steps per track, 500 epochs, and homework 4 solution controller

Fig 2.9	Zengarden	Lighthouse	Hacienda	Snow tux peak	Cornfield _Crossing	Ssncotla nd	Cocoa temple
Time to Finish	34.0 seconds	38.8 seconds	51.0 seconds	49.4 seconds	62.8 seconds	56.3 seconds	60.4 seconds

Final loss: 0.069 Average time: 48.72 s



```
layers.append(torch.nn.Conv2d(3,16,3,1,0)) # MODEL ARCH: input image, convolutional layer(s), nonlinearity (ReLU), batchnorm2d, max pooling, nonlinearity (ReLU)
layers.append(torch.nn.ReLU())
layers.append(torch.nn.BatchNorm2d(16))
layers.append(torch.nn.MaxPool2d(3,1,0))
layers.append(torch.nn.ReLU())
layers.append(torch.nn.Conv2d(16,32, (5,1),1,0)) # padding=0 to capture the edges of the image with the 3x3 filter
layers.append(torch.nn.ReLU())
layers.append(torch.nn.Conv2d(32,32, (1,5),1,0)) # 5x1 1x5 convolutions to factorize a 5x5 convolution
layers.append(torch.nn.ReLU())
layers.append(torch.nn.BatchNorm2d(32))
layers.append(torch.nn.MaxPool2d((5,1),1,0))
layers.append(torch.nn.MaxPool2d((1,5),1,0))
layers.append(torch.nn.ReLU())
layers.append(torch.nn.Conv2d(32,16,(7,1),2,0))
layers.append(torch.nn.ReLU())
layers.append(torch.nn.Conv2d(16,16,(1,7),2,0)) # 7x1 1x7 to factorize a 5x5 convolution
layers.append(torch.nn.ReLU())
layers.append(torch.nn.BatchNorm2d(16))
layers.append(torch.nn.MaxPool2d((5,1),1,0))
layers.append(torch.nn.MaxPool2d((1,5),1,0))
layers.append(torch.nn.ReLU())
layers.append(torch.nn.Conv2d(16,16 ,3,1,0))
layers.append(torch.nn.BatchNorm2d(16))
layers.append(torch.nn.MaxPool2d(3,1,0))
#dropout?? nn.Dropout2d(0.25)
# nn.Dropout2d(0.5)
layers.append(torch.nn.Linear(18,16))
layers.append(torch.nn.ReLU())
layers.append(torch.nn.Linear(16,2)) # applies a linear transformation y=xAT + b
layers.append(torch.nn.ReLU())
self._conv = torch.nn.Sequential(*layers)
```

Fig 2.10: ChaseNet

This final model architecture uses a block structure which, at first, has increasing channel sizes and increasing kernel sizes like an Inception architecture but without any 1x1 convolutions or asymmetric, even kernel sizes. Each block is as follows: convolutional layer(s), nonlinearity (ReLU), batchnorm2d, max pooling, nonlinearity (ReLU). Each batch normalization layer immediately follows convolution(s) because convolving shares weights across channels, while batch normalization only normalizes one channel at a time (see Fig 2.11). We used batch normalization after each convolutional layer like ResNet. Both AlexNet and LeNet use pooling after their convolutions, we chose to use it after convolutions but also after batch normalization. We also chose to make the model have more layers like AlexNet, but due to initially poor performance with fully connected layers, we chose 2 fully connected layers instead of 3 like in AlexNet. To prune our earlier models, we assume all convolutions with a kernel size exceeding 3x3 are generally not useful because larger filters can be concatenated into a sequence of 3x3 convolutional layers. By this same principle, concatenating larger filters into 2x2 convolutions saves 11% of computational time, but asymmetric convolutions e.g. n x 1 followed by 1 x n is even better, it's 33% cheaper for the same number of output filters. This method is also less likely to overfit the model as we go deeper into training. For all convolutional and max pooling layers with a kernel size greater than 3x3, I factorized them into n x 1 then a 1 x n convolution. Factorizing convolutions smaller than 3x3 may hinder network performance. After increasing the kernel size above 3x3 to 7x7 and factorizing, I then chose to progressively lower the kernel size as the network approached its two fully connected layers at its end.

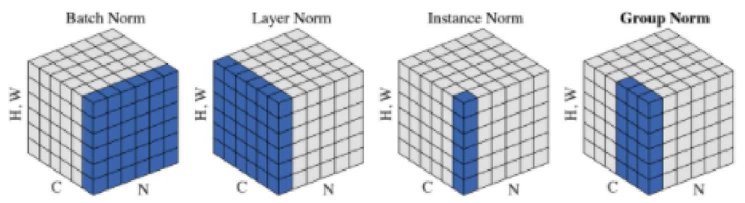
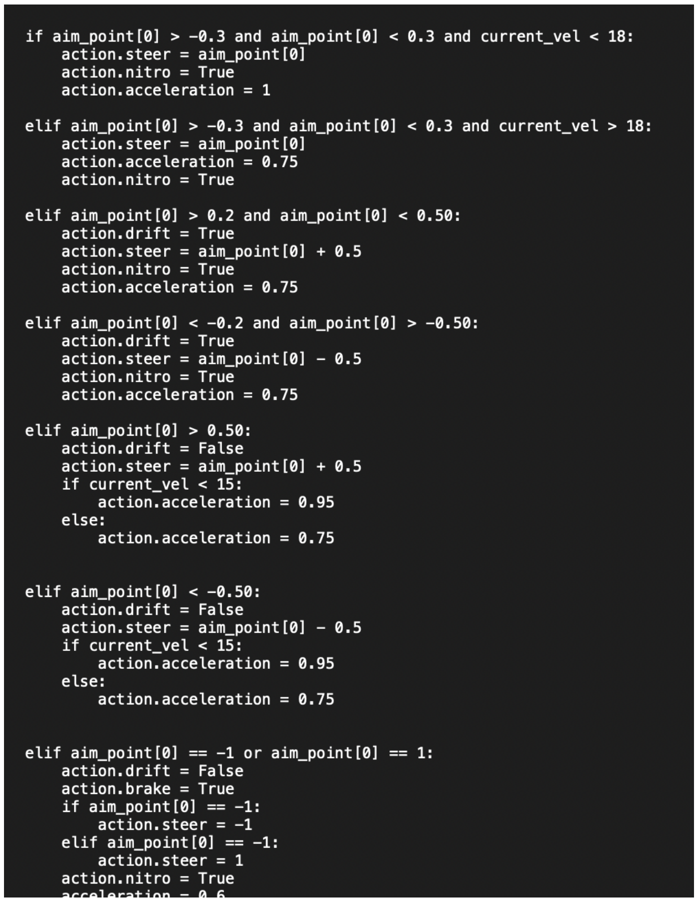


Figure 2. **Normalization methods**. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

Improving Controller.py

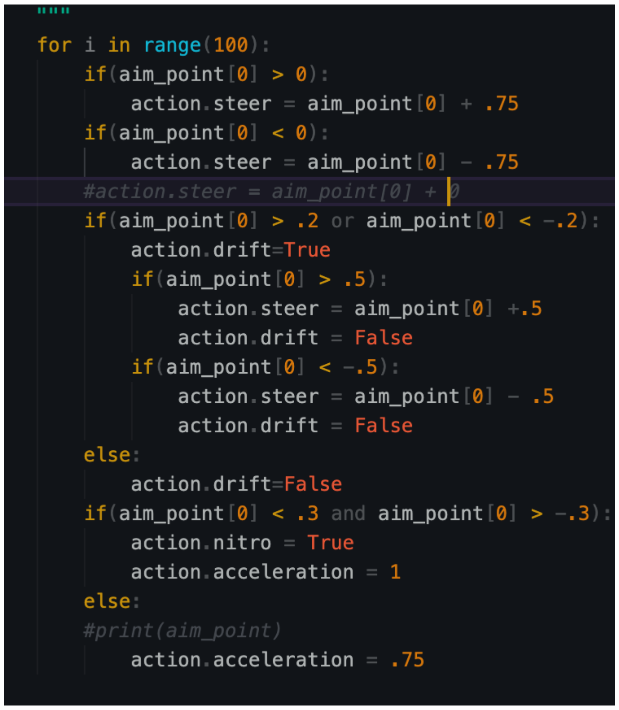
We began improving the controller by coming up with an algorithm to optimize the steer. There are several factors that we have to consider when improving the controller, and these vary from map to map. As seen in the datasets above, the same controller yielded different results across each map, so initially, I (Rajiv) sought to create a new controller specific for each map. We began to cut run times, but instead of keeping several controllers in the same file, we decided to look specifically at how we changed the file, controller.py. In simpler maps such as hacienda and zengarden, adding a lot of if statements to specify specific states of the kart yielded good results, as the track rarely had any complicated twists and turns. However, for maps such as scotland, there were major glitches that seemed to take place, even though we accounted for lots of different scenarios in the code. We realized that what happened was that the if statements began to override each other during the kart's steering, and the kart thus did not know which action to take. Too many if statements increase run time significantly as well, so we knew that too many conditions would be detrimental to completing the course faster. Figure 2.11 is the code that was used under this model.

Figure 2.11



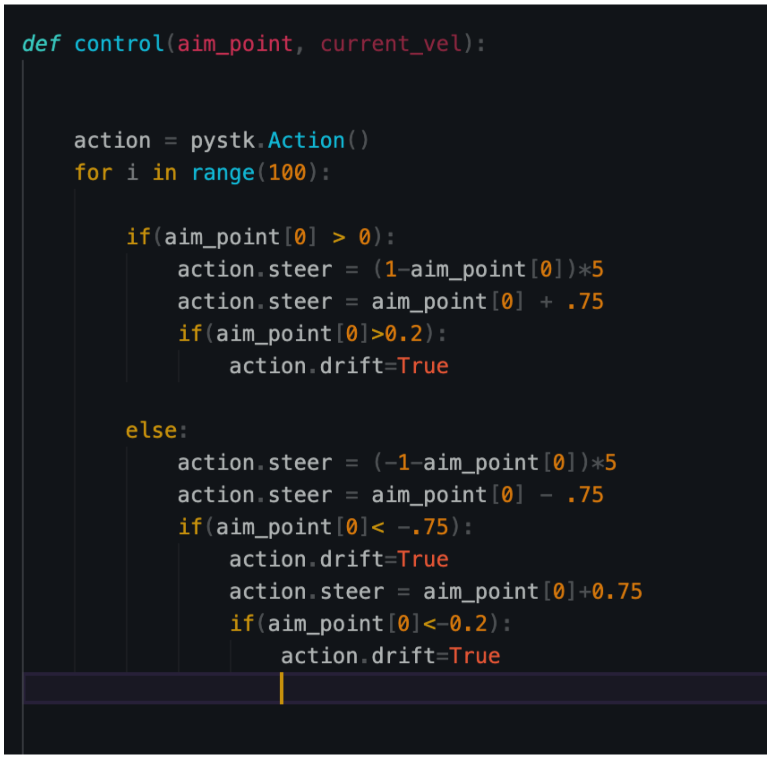
```
if aim_point[0] > -0.3 and aim_point[0] < 0.3 and current_vel < 18:
    action.steer = aim_point[0]
    action.nitro = True
    action.acceleration = 1

elif aim_point[0] > -0.3 and aim_point[0] < 0.3 and current_vel > 18:
    action.steer = aim_point[0]
    action.acceleration = 0.75
    action.nitro = True

elif aim_point[0] > 0.2 and aim_point[0] < 0.50:
    action.drift = True
    action.steer = aim_point[0] + 0.5
    action.nitro = True
    action.acceleration = 0.75

elif aim_point[0] < -0.2 and aim_point[0] > -0.50:
    action.drift = True
    action.steer = aim_point[0] - 0.5
    action.nitro = True
    action.acceleration = 0.75

elif aim_point[0] > 0.50:
    action.drift = False
    action.steer = aim_point[0] + 0.5
    if current_vel < 15:
        action.acceleration = 0.95
    else:
        action.acceleration = 0.75


elif aim_point[0] < -0.50:
    action.drift = False
    action.steer = aim_point[0] - 0.5
    if current_vel < 15:
        action.acceleration = 0.95
    else:
        action.acceleration = 0.75


elif aim_point[0] == -1 or aim_point[0] == 1:
    action.drift = False
    action.brake = True
    if aim_point[0] == -1:
        action.steer = -1
    elif aim_point[0] == -1:
        action.steer = 1
    action.nitro = True
    acceleration = 0.6
```

The controller used in the harder to finish maps included less if statements, with nested if statements to control the priority of actions during a particular state. This made it so that the kart had less instructions to follow simultaneously, and its actions would remain simple, yet responsive, throughout the course. To ensure that the kart was responsive, we made sure that it “wiggled”, meaning that it was constantly adjusting its position by continuously steering left and right. This ensured that the kart was prepared for any turns. Another important thing we discovered was that drifting should not happen if a turn was too sharp, because this would often result in the kart falling off the map. Drifting was thus only done when the turn was not as sharp, so that the kart would remain stable. However, this solution was not able to finish Cocoa Temple (Figure 2.12). There was only one controller that was actually able to finish it (Figure 2.13).



```
"""
for i in range(100):
    if(aim_point[0] > 0):
        action.steer = aim_point[0] + .75
    if(aim_point[0] < 0):
        action.steer = aim_point[0] - .75
    #action.steer = aim_point[0] + 0
    if(aim_point[0] > .2 or aim_point[0] < -.2):
        action.drift=True
        if(aim_point[0] > .5):
            action.steer = aim_point[0] +.5
            action.drift = False
        if(aim_point[0] < -.5):
            action.steer = aim_point[0] - .5
            action.drift = False
    else:
        action.drift=False
    if(aim_point[0] < .3 and aim_point[0] > -.3):
        action.nitro = True
        action.acceleration = 1
    else:
    #print(aim_point)
        action.acceleration = .75
```

Figure 2.12



```python
def control(aim_point, current_vel):


    action = pystk.Action()
    for i in range(100):

        if(aim_point[0] > 0):
            action.steer = (1-aim_point[0])*5
            action.steer = aim_point[0] + .75
            if(aim_point[0]>0.2):
                action.drift=True

        else:
            action.steer = (-1-aim_point[0])*5
            action.steer = aim_point[0] - .75
            if(aim_point[0]< -.75):
                action.drift=True
                action.steer = aim_point[0]+0.75
                if(aim_point[0]<-0.2):
                    action.drift=True

```

Figure 2.13