SuperTuxSmart

EC418 Final Project

Hilario Gonzalez, Josh Barwick, Noah Hathout

{hilario,jbardwic,nhathout}@bu.edu

Figure 1. Screenshots from the PyTuxKart emulated environment showcasing autonomous driving.

Abstract

This project explores improving autonomous driving in an emulated version of SuperTuxKart using reinforcement learning and machine learning methods. Starting from a baseline planner that reliably completes all available tracks, we implement and train convolutional neural networks (CNNs) to estimate aim points from raw, labelled, images. We experiment with different architectures, including the baseline CNN, deeper networks, and a ResNet combined with an autoencoder for feature preprocessing. We train models for 50, 100, 400, and 1000 epochs and evaluate them on all tracks. Our results show that the original planner baseline achieves 100% completion across tracks, whereas our trained CNN planners struggle to reach similar performance, often completing significantly lower percentages of each track. Additionally, we attempt a reinforcement learning approach using a Deep Q-Network for action selection. Without advanced RL techniques such as replay buffers and target networks, the RL approach fails to improve completion rates. While initial results are modest, our extensive exploration of architectures, training regimes, and RL methods provides insights for future improvements and refinements.

1. Introduction

The goal of this project is to optimize completion time of tracks in the driving simulator pySuperTuxKart. We use machine learning and reinforcement learning methods to estimate aim points and controller input for steering, acceleration, and braking. We began the project with a controller and planner that worked well, completing every course in a reasonable time. Our goal was to improve on the baseline planner by exploring various convolutional neural network (CNN) architectures. We were able to use a dataset with labelled optimal aim points to train our CNNs. We wanted to improve our controller with reinforcement learning and explored various techniques using deep neural networks to make this work. For both approaches, we experimented with different training strategies. At some points, it seemed that we were overtraining and memorizing data and finding the optimal training time was another problem we had to optimize. In conclusion, we started with a baseline planner and controller and aimed to improve upon these with machine learning and reinforcement learning. We verify our results with completion times across all available tracks.

2. Background

PySuperTuxKart is set up with two opportunities for lap time improvement. The first is the planner, which receives an image of the track and returns a coordinate pair that the kart should aim towards. The planner is a CNN that is pretrained on labelled images of SuperTuxKart tracks. This coordinate pair is passed to the controller, which returns the PySTK action class. This class has steering, which ranges from -1 to 1, and acceleration which ranges from 0 to 1. It also includes five boolean fields, brake, nitro, drift, rescue, and fire. As fire and rescue are environmental variables, we did not adjust these. Our base controller used if-then statements to make decisions about the action output, but our goal was to make our controller a neural network that we trained with reinforcement learning, specifically Deep Q-Learning. The update rule for Deep Q-Learning is outlined below.

$$\theta_{+1} = \theta + \alpha (r + \gamma \max_{a'} Q\theta(s', a') - Q\theta(s, a)) \nabla\theta Q\theta(s, a)$$

3. Experimental Setup

Setting up the environment to run PySuperTuxKart was quite challenging because the pystk library was not integrating smoothly with Apple's M1 and M2 chips, and would clash against different Python versions. After much trial and error, the solution was to create a new user on our Apple laptops and install and create a Miniconda3 environment. We found that Python 3.10.15 allowed us to install and import pystk. Despite the installation of mesa-libegl, we could not get a functioning graphical window for PySuperTuxKart and relied on completion times for results.

The CNN training was done using a Linux-based environment running under WSL2 on a Windows 11 system with an NVIDIA RTX 4080 GPU. Because training was much faster on the GPU, we trained on the Windows environment and sent the pre-trained weights to our M1 and M2 Apple laptops for visualization and testing. To train the CNN planner, we first generated a dataset of images annotated with optimal aim points as shown in Figure 1 at the front of the report. Given to us was a utilities file (utils.py), which, when running "python3 -m utils [track_names]", will collect all the necessary data for each track. The script stores the images and their corresponding aim point labels in the /drive_data directory. This dataset then serves as the supervised training set for our CNN-based aim point estimators.

With CNN training complete, we created our reinforcement learning training implementation. Every time that an aim point was calculated, e.g. for every frame, we took that aim point and decided a Q-Value for each action. The Q-Value update was also performed every frame of the simulation. The weights of the controller neural network were written to a .pth so we could update across simulations. Additionally frames, completion times, epoch, and track name were written to a csv so that we could perform analytics on them after the training was completed. We then ran each version of the reward system for at least 1000 epochs.

4. Initial Approaches: Planner

In order to begin improving the planner, we needed to lay a foundation to test our future iterations against. We accomplished this by running all the race courses with the default original planner that comes with the Pystk modules as well as the controller we developed in coding assignment three. This controller would be used across all future planner tests to ensure consistency in our data. Seeing that this default planner completed all the courses and averaged 530.3 frames per track, we took its times as the baseline. See Appendix B for a comparison of performance across all tracks.

With this baseline of times, we knew what we were aiming to outperform. However, we didn't know anything about the architecture of the default planner, how long it was trained, not even what its loss looked like, so we would have to come up with our approach entirely from scratch.

We started with our theoretical approach. Our task is to develop a convolutional network that can predict an aim point coordinate when provided with a frame from the SuperTuxKart simulation. In our understanding this could be boiled down to an object detection and localization task. A lot of research has been done on creating better object detection using databases like CIFAR-100, so there is a lot of conventional wisdom to learn from. Additionally, there are a few key features of our data that make it a little easier to deal with. First, the SuperTuxKart frames are very low resolution. This makes computation done on these images less expensive. Second, the aim points we are detecting follow somewhat of a pattern. Object detection works by first extracting low level features from an image like edges, corners, and textures. Then, subsequent layers combine these features into higher level representations. Network architecture depends on how many features you want your network to be able to extract, distinguish, and classify. Therefore, we are fortunate that the aim points we are trying to detect don't have a super high level representation. They are essentially always the intersecting point of the track with the horizon, and there aren't many high level features to be extracted that will give a better idea of where this could be. Furthermore, the aim points don't really scale much. A classic problem in object detection was dealing with scale. If you're applying the same kernels across the whole image, how are you supposed to distinguish between a small person far away and a large one in the foreground? Many object detection architectures include a Feature Pyramid Network to address this problem, and a lot of the architecture's complexity can be

attributed to the complexity of the data it is trying to represent. Since our data is not very complex, and we are only looking for one thing, we can afford to start with a simpler architecture.

In the code provided with coding assignment three, the planner.py file already had a small Convolutional Network written in. This was made up of one convolutional layer that applied a 5x5 kernel with stride 2 across 16 channels. We used this network as our architecture baseline. Our plan was to begin experimenting with different kernel sizes and combinations, as well as the number of epochs, before implementing more advanced architectures.

The first thing we did was add a weighted combination layer, which allowed us to train multiple architectures in parallel, and since the weights were themselves learnable parameters, the model can decide for itself how much value is placed on each network. We trained the 5x5 single layer network alone first, and then trained it against a similar shallow network that used 3x3 kernels. These two networks performed similarly, with the 5x5 kernel network performing slightly better. The training loss after 50 epochs was around 0.15 for these shallow networks, and when tested on-track, they couldn't complete a single course. Next, we added a layer to the 3x3 kernel network, because two 3x3 kernels can capture a similar amount of data as one 5x5 kernel. This time, the two layered 3x3 kernel network performed better, reducing the loss to 0.147. From this we learned that a deeper architecture was better than a shallow one that holds the same amount of feature representation, and that a larger kernel is better than a smaller kernel at holding the spatial context required for identifying an aim point given the resolution of our data. More importantly, we learned that there is a lot of room for improvement.

Another thing we experimented with when figuring out our initial approach was changing the number of epochs trained. We noticed that even with the shallow networks, if we kept increasing the number of epochs, the loss would go down. We thought to make a simple multi-layered network, and then train it for a long time and it would get a low enough loss to be good. We started with increasing to four convolutional layers and used an increasing kernel size to capture features of varying complexity. We also increased the channel number across these layers to increase the feature space and balanced it with average pooling layers that would decrease the dimensional space and preserve computational efficiency. This model yielded a training loss of 0.08 after fifty epochs, which was a significant improvement over the previous shallow models. We then trained this model as well as the CNN baseline [shallow 5x5 kernel

network] for 400 and 1000 epochs each. When we tested them with all the tracks none of them performed well. The shallow networks performed equally bad regardless of how much they were trained, and the deeper network performed worse after 1000 epochs than it did after 400. We observed that in their loss functions, while the average slope approached zero, they would reach a point after a few hundred epochs where they would begin on a linear trajectory towards the x axis, and begin experiencing much higher volatility. This indicated to us that they were becoming overtrained.

We now had a solid foundation of understanding to move forwards with. We knew that deeper architectures with more representation space performed better, and that our data was low resolution and our predictions were relatively simple, so too much representation space would have diminishing returns for the task at hand. We also knew that deeper networks would be more sensitive to overtraining, and that the ideal training range we found was between fifty and one hundred epochs. With all of this in mind we could move forward with experimenting with more complex architectures.

5. Improved Architectures:

To further improve the performance of the aim point detector network, we explored a variety of convolutional network paradigms that could enhance feature extraction, representation, and robustness.

In the previous iteration of our CNN, we reduced the spatial dimensions with strided convolutions. To add depth, we experimented with multiple convolutional layers of stride 1 followed by a pooling layer. Pooling layers serve to reduce the spatial dimensions of feature maps, enabling the network to focus on the most relevant features while maintaining computational efficiency. MaxPooling selects the maximum value from a window of pixels, emphasizing the most prominent features in the input, which was particularly useful for retaining sharp edges and high-contrast regions critical for identifying the aim point along the horizon. Average Pooling, on the other hand, calculates the mean of pixel values in a window, providing a more generalized abstraction of the input, which helps smooth noise and improve the robustness of feature extraction. By experimenting with these pooling techniques, we balanced the trade-off between computational efficiency and feature preservation, incorporating pooling layers to strategically downsample intermediate feature maps.

As we experimented with more convolutional and pooling layers to make our network deeper, we realized it would be a good idea to address the vanishing gradient problem by encapsulating blocks of layers with skip connections. These skip connections allowed feature maps from earlier layers to bypass intermediate layers and be directly combined with later outputs. This improved gradient flow during backpropagation, enabling the network to learn more complex patterns without degrading performance. Incorporating residual connections allowed us to experiment with deeper architectures, which were capable of capturing hierarchical features at varying levels of abstraction, with the goal of improving the network's ability to localize the aim point.

Increasing the depth of the network and using residual connections to prevent vanishing gradients proved to be a good approach to extracting a good heatmap from an image, but we thought we could use our knowledge of the task at hand to improve it even more. We knew that with a human eye one could identify an aimpoint very easily, even in low resolution. This is because as humans we are able to prioritize the more relevant features. In other words a human can figure out pretty quickly where to aim by focusing on things like where the road is, how it

curves, and where the horizon is, and we don't get distracted by different colors in the road or shapes in the background. Up until now, we have increased the depth of the network, giving it the space to learn complex representations and features, so now with a lot of feature-space to work with we considered implementing a squeeze-excitation attention mechanism.

Squeeze-and-Excitation (SE) blocks enhance the network's channel-wise feature recalibration. These blocks allowed the network to adaptively weight the importance of each feature map, focusing more on features critical to detecting the aim point. By "squeezing" global spatial information into a channel descriptor and "exciting" the important channels, the SE blocks dynamically adjusted the network's feature representation. This mechanism improved the network's ability to handle variations in track textures and lighting conditions, making it more robust across different tracks.

The Residual network with squeeze excitation proved to be even better performing than our previous CNN with increasing kernel sizes, although it was still pretty bad, only completing a few tracks. The next step was to combine the strengths of both of these approaches. The Resnet had depth, and an attention mechanism, and the increasing kernel size network made use of different kernel sizes to extract features of varying complexity. First, we combined the two by connecting the kernel-CNN straight into the Resnet. This Resnet-Kernel CNN proved to be slightly better, so we modified it again by adding more residual blocks. Previously, the model used three residual blocks. Each block would halve the spatial dimensions of the image and double the channel number across two convolutions. Now, there would be another residual block that preserves the dimensions across two convolutions. This block was added after each block that changed dimensions, making six residual blocks in total.

Up until now, every residual block used 3x3 filters only. After seeing how increasing the kernel size across layers proved to be beneficial I decided to define new residual blocks that performed convolutions with 5x5 and with 7x7 kernels. This way I could start experimenting with increasing the kernel size within the residual network as opposed to feeding the Increasing Kernel CNN straight into a simple Resnet. This approach proved the most successful so far, and having modularized my various types of blocks it became easy to experiment with different combinations of kernel sizes.

Ultimately, the most successful network I produced this way featured a first block that performed a 5x5 kernel strided convolution into a max pooling layer which downsampled by two

again. This was then fed into six residual blocks which alternated between 3x3 and 5x5 kernel size while every pair of residual blocks doubled the channel number and halved the spatial dimensions. The output of the residual layers was multiplied by the squeeze-excitation block, and then sent through one final layer that reduced the channel numbers down to 1 to produce the heatmap and finally the spatial argmax, which produced the aim target coordinates. This Resnet with alternating kernels, trained at 50 epochs, was one of the few that completed every track, and it did it with the most consistently good times compared to the other networks.

Something that we considered but ultimately left out were autoencoders. Autoencoders use convolutions followed by inverted convolutions to compress and reconstruct input images, effectively learning a compact representation of features. Our thinking was that this would enable the network to focus on reconstructing the most salient features related to the aim point while discarding irrelevant background details. However this proved not to be very ideal. Our thinking was that since the aimpoint can be identified with low level features, then we can use an autoencoder to learn to reconstruct a more general idea of the image that makes identifying the aimpoint simpler. However in practice the autoencoder simply lost too much information and context from the original image to be able to generate a meaningful prediction for the aimpoint. We tested various architectures that incorporated the autoencoder, as well as testing the autoencoder on its own. It proved to contribute little to nothing to the success of the kart, so this idea was scrapped.

6. Initial Attempt at Reinforcement Learning

As we examined the existing code, we noticed that we were using a neural network for aim points. We wanted to see if we could implement some sort of actor-critic-esque update to our controller so we created a neural network for the controller. The neural network for the controller was very simple, it was an input layer with aim point, current velocity, and target velocity. We took that to a linear layer with 64 nodes, ran it through ReLU, then brought it down to seven nodes, one for each member of the PySTK action member. We used sigmoid to get a boolean for boolean members, and clipped steering and acceleration appropriately.

We then updated the weights of this neural network following the deep Q-Learning method. This did not work at all because we were not predicting scalars for a set of action. The update rule did not push us towards conversion at all and our loss never stopped oscillating. Our calculation subtracted previous action member values from current action member values which did not make any sense at all for something like a boolean. Overall this approach was a misunderstanding of the Q-Learning application and the structure of the PySTK action member. This effort did help us understand reward shaping and the general structure of the code, especially when using a neural network for the controller instead of a function and so was not a complete waste of time. We really wanted to implement Q-Learning so we decided to go for a classic implementation as outlined in the next section.

7. Reinforcement Learning: Deep Q-Learning

In this section, we describe our attempts to use Deep Q-Learning to train our controller. We really wanted to see this work rather than a PID because we are already doing some control theory in a different class. Because we had already attempted to get a version of Q-Learning to work we decided to go with a direct implementation of it. Our first step was to discretize the action options. Below is a table of the eight actions we created:

Action	Steer	Acceleration	Brake	Nitro	Drift
Turn Left	-1.0	0.0	False	False	False
Turn Right	1.0	0.0	False	False	False
Accelerate	0.0	1.0	False	False	False
Brake	0.0	0.0	True	False	False
Accelerate with Nitro	0.0	1.0	False	True	False
Drift	0.0	0.0	False	False	True
Accelerate Left	-1.0	1.0	False	False	True
Accelerate Right	1.0	1.0	False	True	True

Once this was done, we created the neural network for our controller. This is a pretty simple neural network with three linear layers connected by ReLU activation functions. Our network is visualized in Figure Fifteen. Our update rule follows Deep Q-Learning. First we calculate the reward which began as a simple distance_covered/track_length. We then used our previous input tensor and current input tensor to get predicted Q-Values for both states. We then did the update we did in class which is the derivative of MSE loss between the current Q-Value and target Q-Value. We also implemented an epsilon greedy selection function that allowed us to explore. We decreased epsilon by 0.95 between each run to make sure we stopped using it eventually. All of these runs are using the baseline CNN provided to us.

Attempt One:

Our reward in this run was calculated as kart.overall_distance / track_state.length. In this run, we found that the average completion percentage was 8.17%. When excluding cases where the track had crossed the starting line backwards, that completion rose to 12.41%. Overall, this was the most successful run. This reward function encouraged the least amount of negative completion percentages, as demonstrated by Figure Sixteen. In this run, there were 177 runs within 99% completion, more than three times either of the following tests. There were also 6 runs that were fully completed before the 999 frames ran out.

Attempt Two:

In an effort to improve track times, we punished rewards that took longer by calculating reward as (kart.overall_distance / track_state.length) / (sqrt(t)). We found that performance for this reward function was much worse. Overall there were many more 0% completion across all tracks. This can be seen especially in cornfield_crossing in Figure Seventeen. There was only track completion in this entire run on lighthouse and 47 runs that got more than 99%. Viewing Figure Eighteen, the epoch vs completion percentage for all tracks demonstrates that our second attempt at reward shaping was unsuccessful. The average completion percentage with negative values from the cart crossing over the start line was 2%. Without negative values, the average completion percentage was 11%.

Attempt Three:

We changed our reward again by calculating it as (kart.overall_distance / track_state.length) / (log (t + 0.01)). The small 0.01 is for when t = 0 to prevent undefined behavior. Again, there was only one run that hit the completion, but 48 runs that got above 99%. From this, we can see that the runs are pretty similar at the top end. The total average completion was -15% if we include runs that went back over the finish line and returned -1 for completion. Without those negative values, we found that the average was 8.9% completion. Based on these numbers as well as Figure Nineteen, which shows the average completion vs epoch, we can see that this attempt at reward shaping performed the worst overall.

8. Discussion

We evaluated many architectures for the planner CNN. We started with shallow models where we experimented with varying kernel sizes and downsampling steps, and then expanded into more sophisticated architectures that incorporated conventional ideas about object detection and CNNs. Our initial CNN remained far from achieving the same level of success as the default planner that comes with pystk. It demonstrates moderate performance gains with more epochs, varying kernel sizes, and more depth, but still falling short of the baseline's score. The ResNet combined with an autoencoder, despite its theoretical advantages, failed to outperform the simpler CNN architecture under the given conditions, suggesting that the task of aim point detection might not benefit significantly from the added complexity. The simplicity of aim point detection, where the target is often defined by low-level features like track boundaries and the horizon, may have rendered complex models prone to overfitting or hindered their generalization capabilities. This was further corroborated by observing the effect of training duration. Increasing the number of epochs from 400 to 1000 did not consistently enhance performance; instead, some architectures showed signs of overfitting or instability, with losses becoming volatile after prolonged training. Ultimately, our findings highlight the importance of matching architectural complexity to the specific demands of a task. While deeper networks with residual connections and advanced attention mechanisms showed promise, the project revealed diminishing returns beyond a certain depth and sophistication. By focusing on balanced training durations and leveraging simpler but robust models, we established a foundation for further experimentation and refinement

Our Q-Learning implementation was not successful. Clearly some part of the reward function was suggesting that the kart turn around and cross the starting line, rather than do a lap. We tried to encourage the model to take less time by dividing the reward by some function of t. The simulation is structured such that frames continue to run if the starting line is crossed rather than a completed lap, which ends the frame. By discouraging rewards that took longer to get, we hoped to encourage the kart to drive forward. This did not work but the sqrt(t) did perform better than log(t) suggesting that there may have been some merit to this approach. In the future we need to isolate the part of the code that is returning a full track length when the kart has just

turned around. It is clear from our completion graph that there is a good amount of oscillation between going backwards and going forwards. In which case did the kart go forward? Once that instance is isolated we can force the simulation to follow that initial behavior. It seems that our implementation of Deep Q-Learning itself may have had some issues but there seemed to be a more critical error in the way that we were counting a “track completion.” Once we solve the problem of the car backtracking, we believe that our implementation would have much more reasonable behavior. Overall, our first iteration without punishing for extended time trials performed the best, demonstrating that adjusting the reward function absolutely changes the behavior of our controller.

9. Future Improvements

Some of the initial room for improving our Deep Q-Learning is in running many more epochs and implementing actor-critic. Beyond these changes, we could improve compute performance with a replay buffer, and tweak the structure of our controller neural network. It does seem that the greatest issue was rewards. Somehow the kart was being told it was optimal to drive backwards, which was obviously not our desired behavior. Perhaps a fix would be dynamic distance calculations that are constantly checked and corrected per time step as opposed to at the end of an episode. A different problem occurs: this kind of reward would focus too much on the current step and gear the controller to make greedy choices that might cause later time steps to crash or fail.

Additionally, there is some ambiguity in the actions that we selected. Refining our set of controls and setting clear bounds will allow smoother steering, acceleration, and enable the kart to make tighter turns and drive better "racing lines". We only offered our controller full turning input and full acceleration input. This may have reduced the simulator's ability to succeed. The possibility of implementing a continuous action space would certainly reduce the challenges presented by trying to discretize all possible actions.

Beyond the RL component, there are several improvements to consider in regards to the planner's CNN. Higher variety of tracks and more edge-case data would allow the model to generalize its learning better to any type of track, since it evident, for example, that on average Table 4 (played on the map SnowTuxPeak) had a significantly worse completed percentage of the track compared to the other tables in Appendix B. Other improvements include extensive hyperparameters tuning and automated search strategies such as AutoML would also help us identity better CNN architectures, possibly incorporating attention layers, or predicting the waypoint more probabilistically rather than deterministic (layer a Gaussian heatmap on top of the images pointing at the most likely direction the kart should take). Finally, by combining development in both the planner and controller, we hope to further improve our PySuperTuxKart.

10. Conclusion

The original planner's baseline was strong. Simple and deep CNN planners struggled alike to match that performance. Despite serious time spent with various CNN structures, complex architectures like ResNet + autoencoder did not outperform simpler models under current settings. Our reinforcement learning approach in its current naive form did not yield improvements, but there's potential if stabilized methods are applied. The real trick here is to reward shape properly. Despite our attempts to use time spent as a punishment we still found that the kart would rather sit at the end for an entire run than get reward for finishing the track early.

Appendix A. Code Repository

https://github.com/nhathout/EC418-Final-Project

main branch has CNNs and controller branch has q-learning

Appendix B. Additional Figures

Table 1. Testing performance data by track: “Zengarden”

Model	Epochs Trained (#)	Frames Until End of Episode	Percentage of Track Completed (%)
Original Planner Baseline	NA	417	100
Our CNN Baseline	400	999	84.274
	1000	999	96.322
Autoencoder into ResNet	1000	999	19.768
Autoencoder (alone)	400	999	89.090
ResNet into Kernel CNN	50	447	99.835
	100	522	100.052
	400	703	99.848
Multi-Layer Modified Resnet Kernel CNN	50	472	99.811
	100	654	99.908
Simple ResNet	50	999	63.798
	100	999	67.895
Modified ResNet	50	430	99.963
	100	542	99.830
ResNet with Shifting Kernels	50	477	99.839
	100	526	99.933

Table 2. Testing performance data by track: “Lighthouse”

Model	Epochs Trained (#)	Frames Until End of Episode	Percentage of Track Completed (%)
Original Planner Baseline	NA	440	100
Our CNN Baseline	400	999	46.813
	1000	999	54.884
Autoencoder into ResNet	1000	999	17.652
Autoencoder (alone)	400	497	100
ResNet into Kernel CNN	50	461	99.941
	100	654	99.844
	400	561	99.969
Multi-Layer Modified Resnet Kernel CNN	50	482	99.849
	100	441	99.992
Simple ResNet	50	842	100
	100	999	66.89
Modified ResNet	50	476	99.868
	100	452	100
ResNet with Shifting Kernels	50	442	99.828
	100	451	99.942

Table 3. Testing performance data by track: “Hacienda”

Model	Epochs Trained (#)	Frames Until End of Episode	Percentage of Track Completed (%)
Original Planner Baseline	NA	580	100
Our CNN Baseline	400	999	35.616
	1000	999	33.917
Autoencoder into ResNet	1000	999	24.909
Autoencoder (alone)	400	570	99.828
ResNet into Kernel CNN	50	541	99.878
	100	999	16.527
	400	567	99.856
Multi-Layer Modified Resnet Kernel CNN	50	613	99.952
	100	551	99.878
Simple ResNet	50	917	99.864
	100	999	79.307
Modified ResNet	50	745	99.843
	100	553	99.920

ResNet with Shifting Kernels	50	560	99.801
	100	549	99.893

Table 4. Testing performance data by track: “SnowTuxPeak”

Model	**Epochs Trained (#)**	**Frames Until End of Episode**	**Percentage of Track Completed (%)**
Original Planner Baseline	NA	610	100
Our CNN Baseline	400	999	5.6097
	1000	999	3.962
Autoencoder into ResNet	1000	999	3.962
Autoencoder (alone)	400	999	4.293
ResNet into Kernel CNN	50	632	99.973
	100	999	20.621
	400	999	12.208
Multi-Layer Modified Resnet Kernel CNN	50	719	100
	100	999	3.362
Simple ResNet	50	999	5.016
	100	999	3.962

Modified ResNet	50	566	99.807
	100	999	11.267
ResNet with Shifting Kernels	50	721	99.91
	100	999	16.246

Table 5. Testing performance data by track: “Cornfield”

Model	**Epochs Trained (#)**	**Frames Until End of Episode**	**Percentage of Track Completed (%)**
Original Planner Baseline	NA	716	100
Our CNN Baseline	400	999	55.074
	1000	999	62.846
Autoencoder into ResNet	1000	999	25.428
Autoencoder (alone)	400	662	99.914
ResNet into Kernel CNN	50	655	99.833
	100	791	99.904
	400	751	99.811
Multi-Layer Modified Resnet Kernel CNN	50	693	99.800
	100	660	99.943
Simple ResNet	50	999	87.366
	100	789	99.920
Modified ResNet	50	617	99.836
	100	692	99.877

ResNet with Shifting Kernels	50	692	99.921
	100	663	99.854

Table 6. Testing performance data by track: “Scotland”

Model	Epochs Trained (#)	Frames Until End of Episode	Percentage of Track Completed (%)
Original Planner Baseline	NA	628	100
Our CNN Baseline	400	999	29.2957
	1000	999	36.9
Autoencoder into ResNet	1000	999	8.136
Autoencoder (alone)	400	999	1.393
ResNet into Kernel CNN	50	641	99.904
	100	999	5.11
	400	999	0.0
Multi-Layer Modified Resnet Kernel CNN	50	622	99.870
	100	999	0.649
Simple ResNet	50	999	2.22
	100	999	2.22
Modified ResNet	50	621	99.851
	100	999	24.0322
ResNet with Shifting Kernels	50	618	99.934
	100	999	10.874

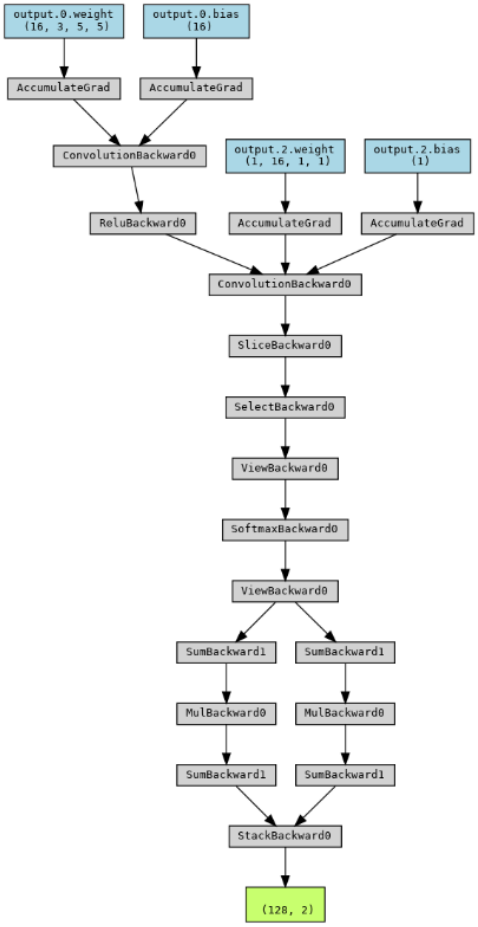

Figure One. Model architecture of baseline architecture.

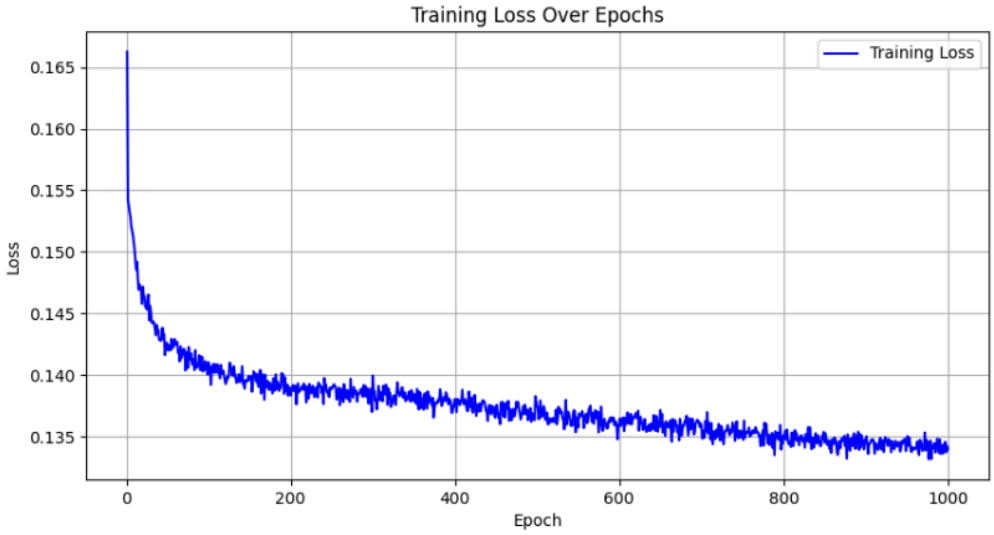

Figure Two. Training loss of baseline architecture over 1000 epochs.

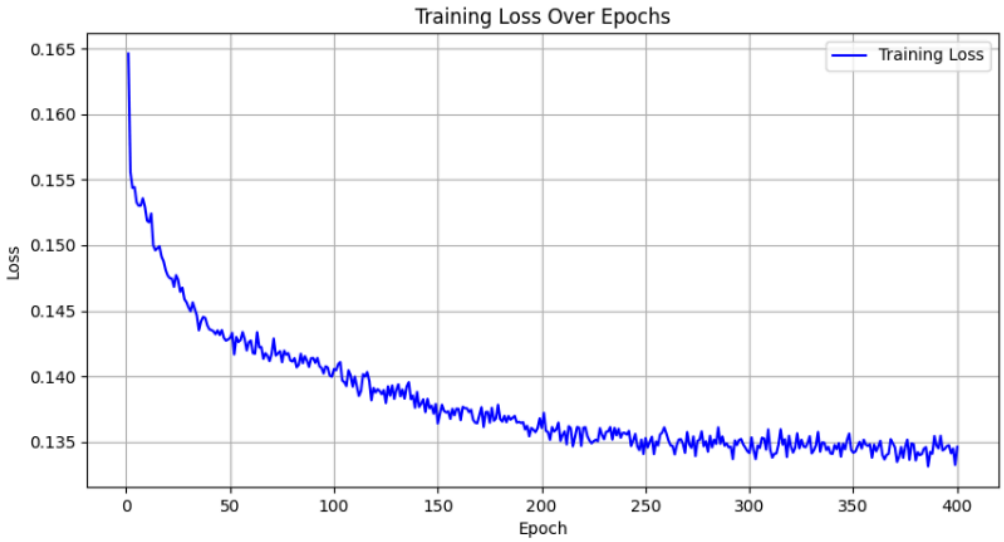

Figure Three. Training loss of baseline architecture over 400 epochs.

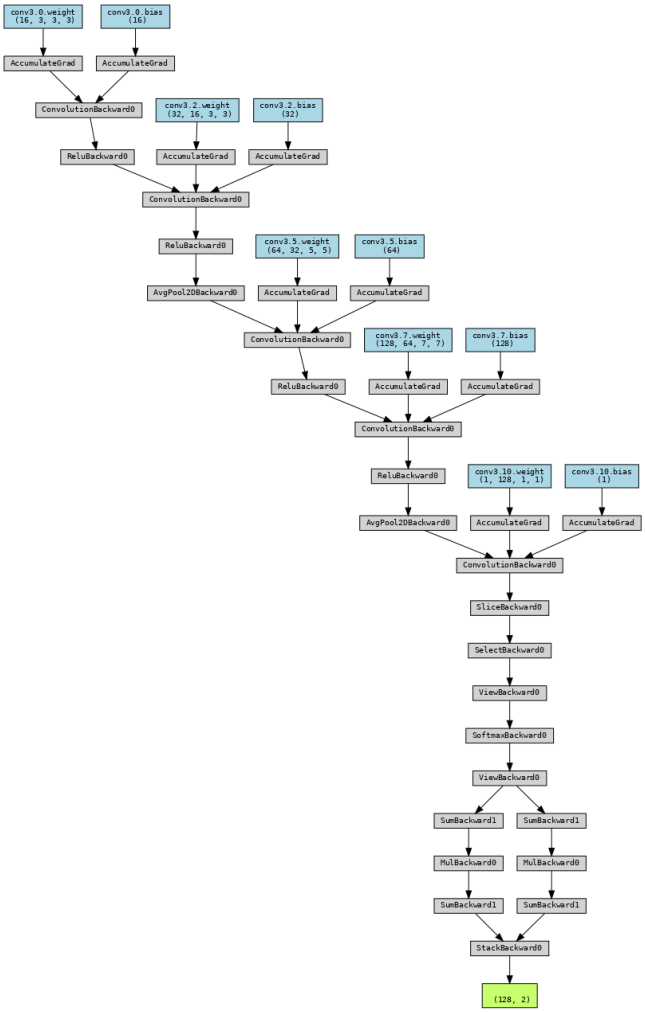

Figure Four. Model architecture of Multi Layer Increasing Kernels.

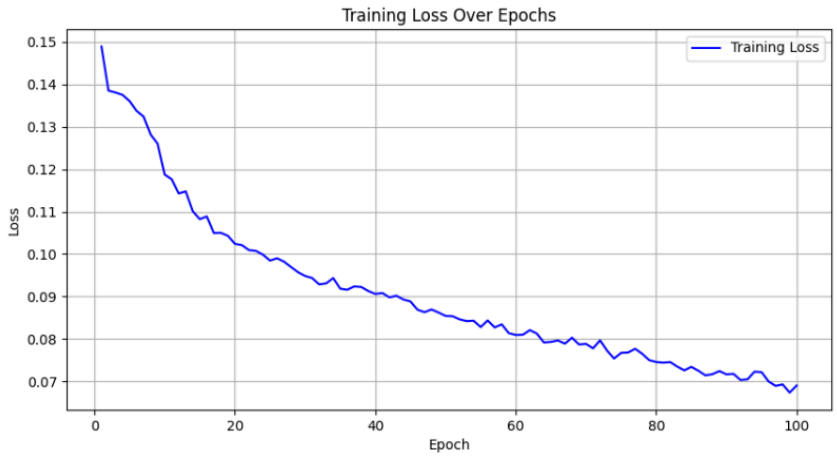

Figure Five. Training loss for Multi Layer Increasing Kernels.

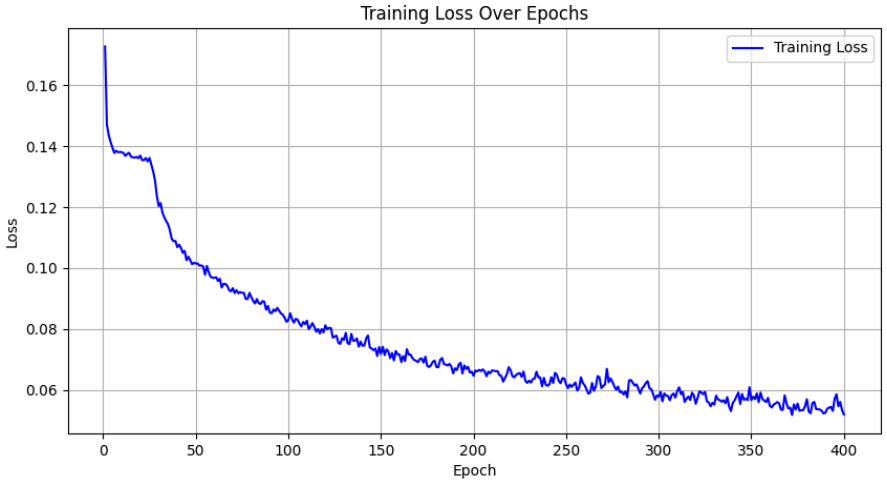

Figure Six. Training loss of autoencoder architecture over 400 epochs.

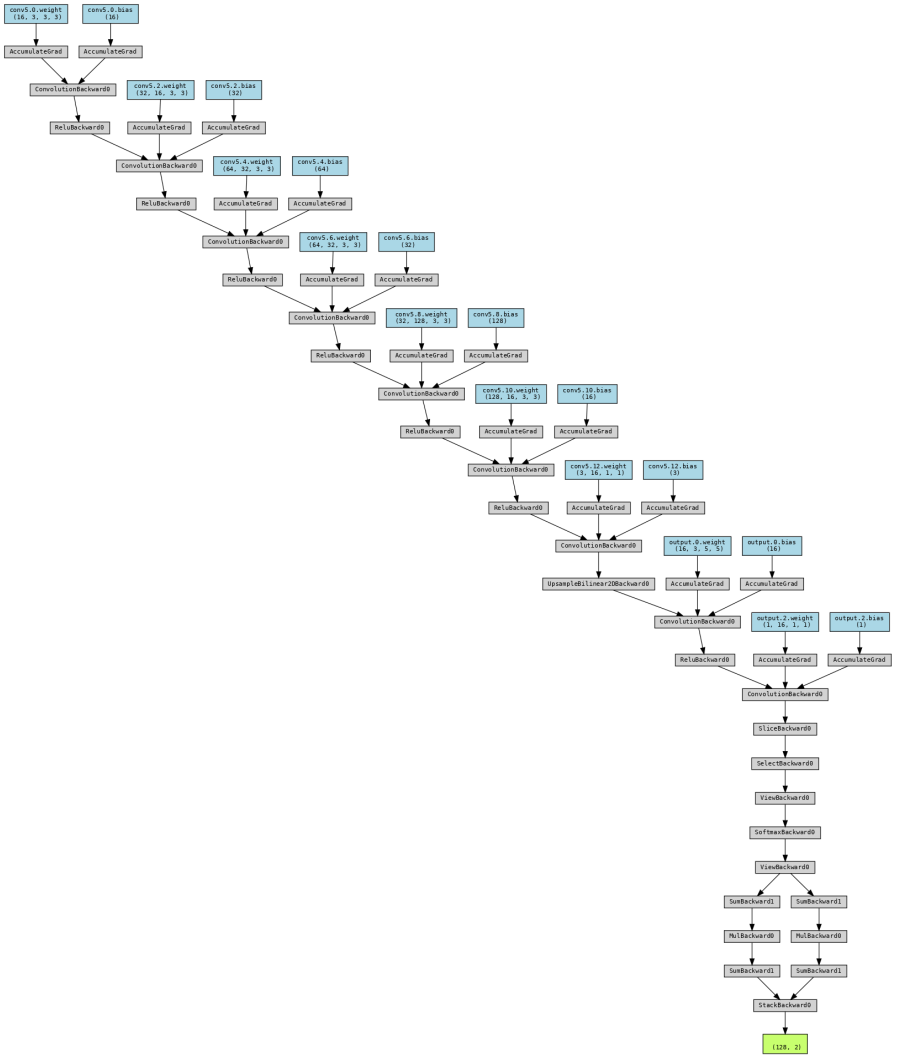

Figure Seven. Model architecture of the autoencoder alone.

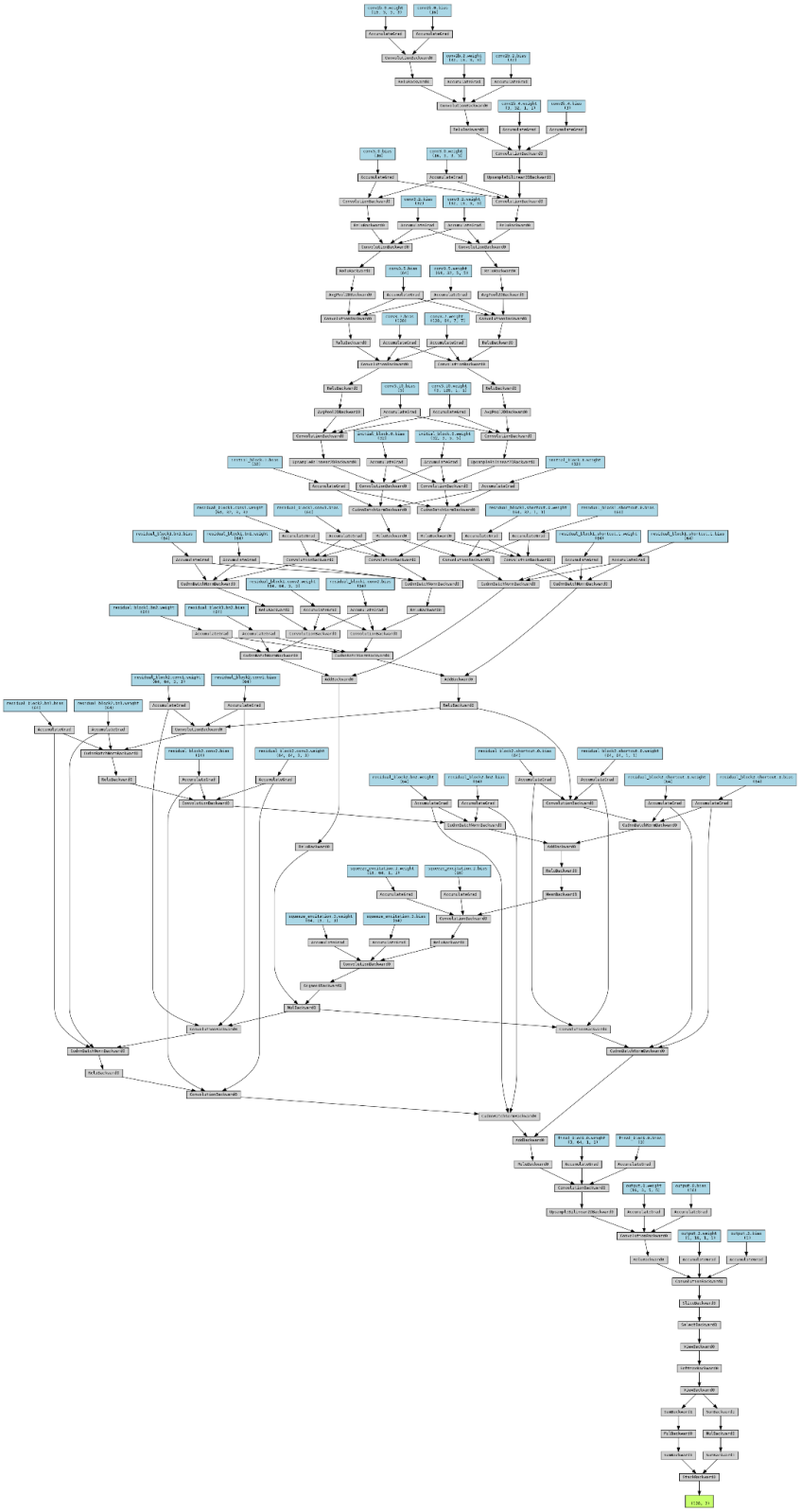

Figure Eight. Model architecture of ResNet + autoencoder.

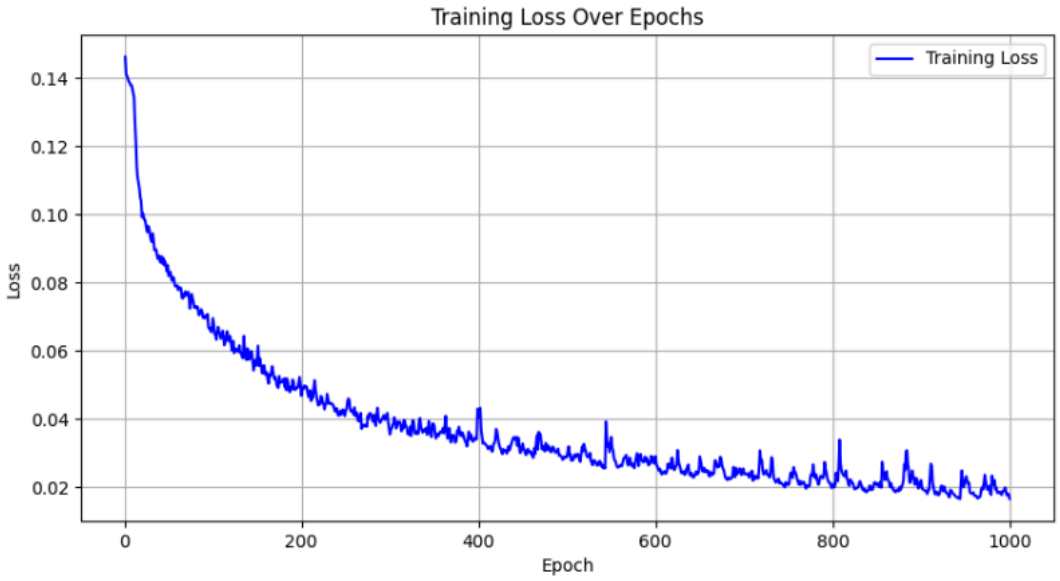

Figure Nine. Training loss of ResNet with an autoencoder over 1000 epochs.

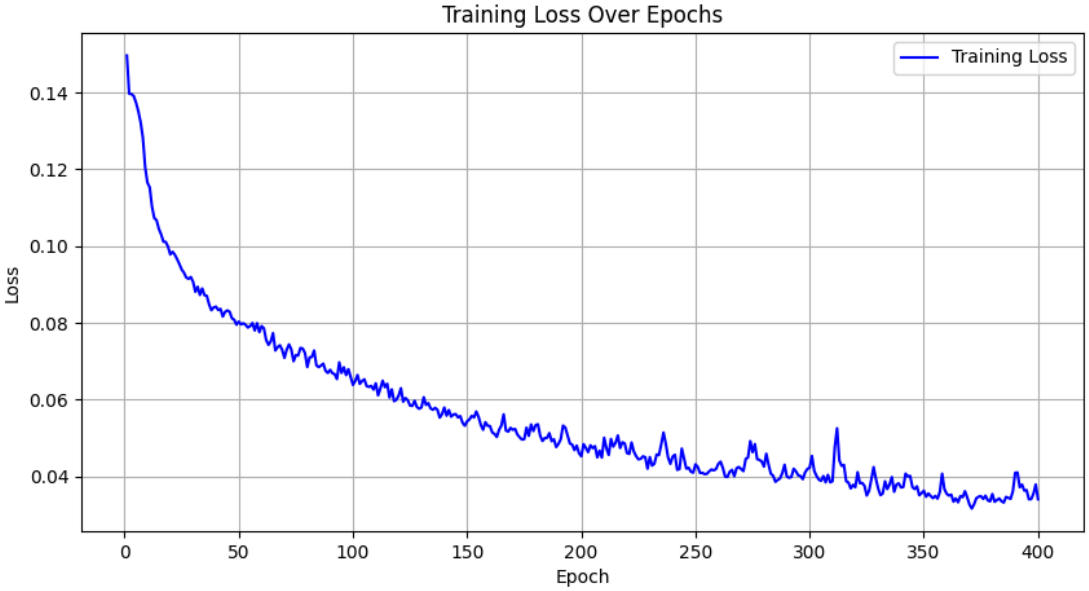

Figure Ten. Training loss of ResNet with an autoencoder over 400 epochs.

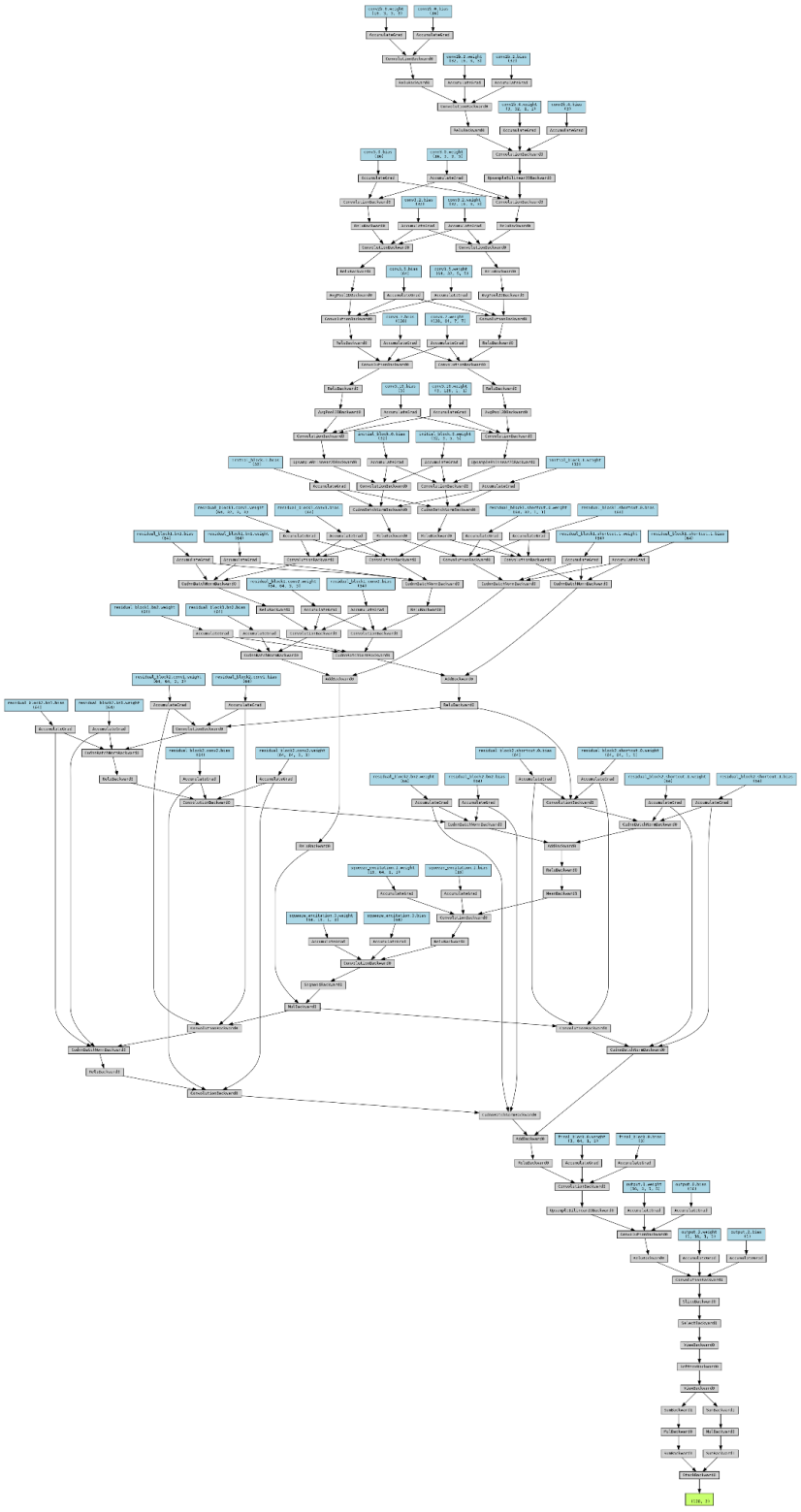

Figure Eleven. Model architecture of ResNet + Kernel CNN

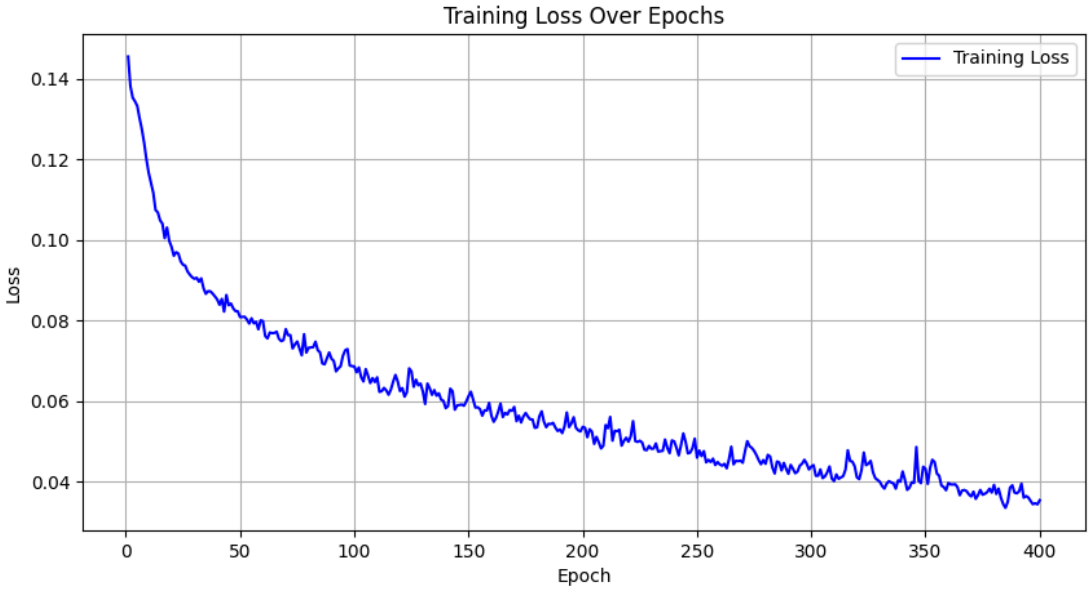

Figure Twelve. Training loss of ResNet with a Kernel CNN over 400 epochs.

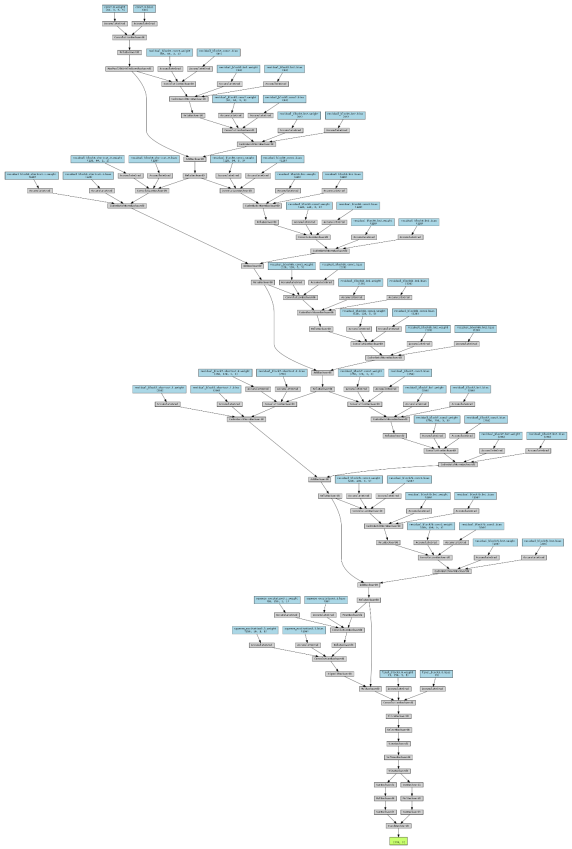

Figure Thirteen. Model Architecture of Resnet with Shifting Kernels

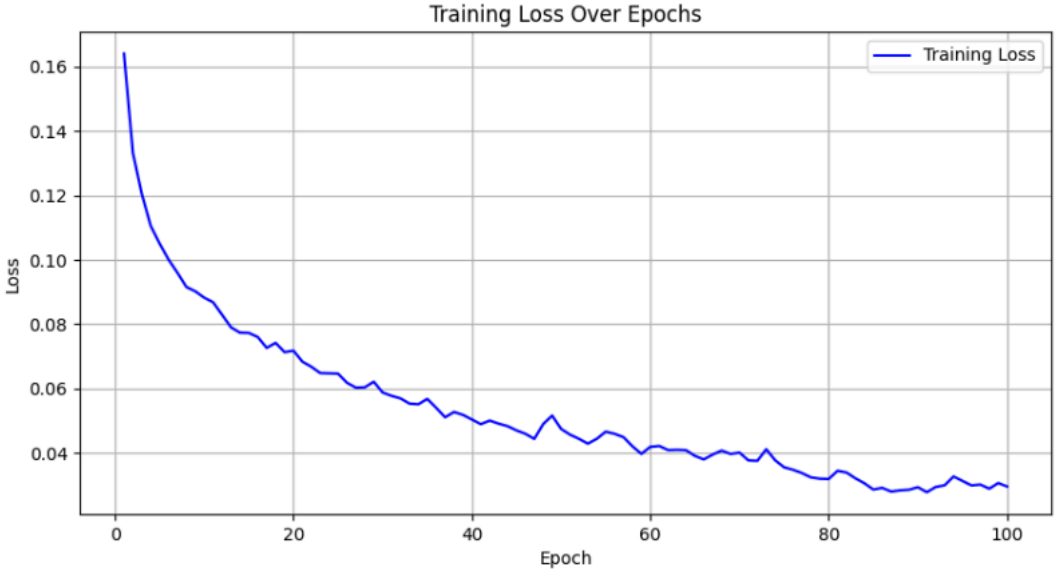

Figure Fourteen. Training loss of ResNet with Shifting kernels over 100 epochs.

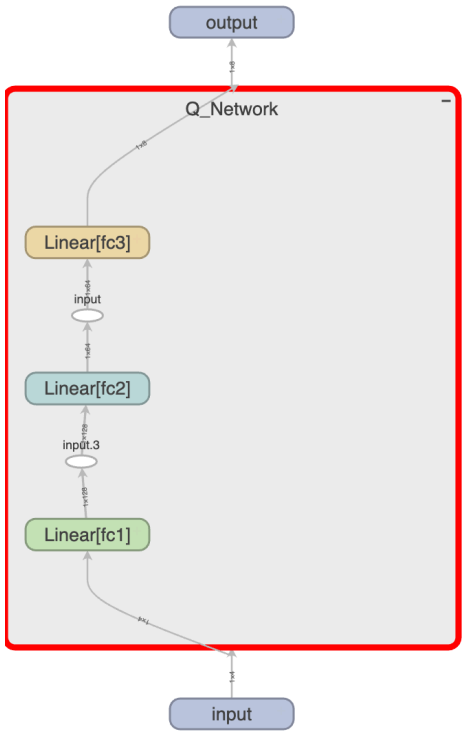

Figure Fifteen: Controller neural network.

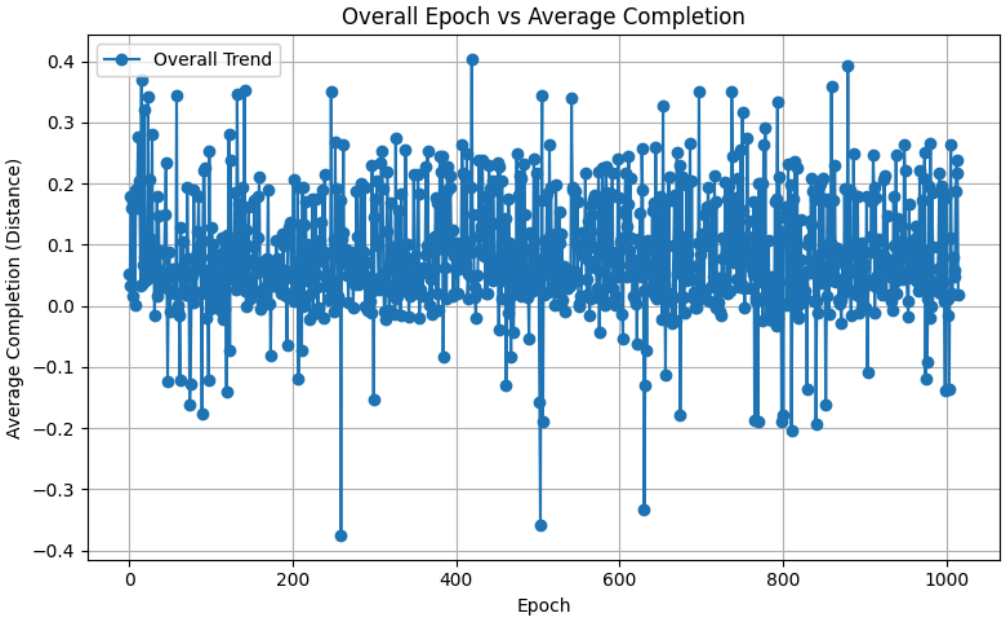

Figure Sixteen: Epoch vs Completion percentage for all tracks with base reward calculation.

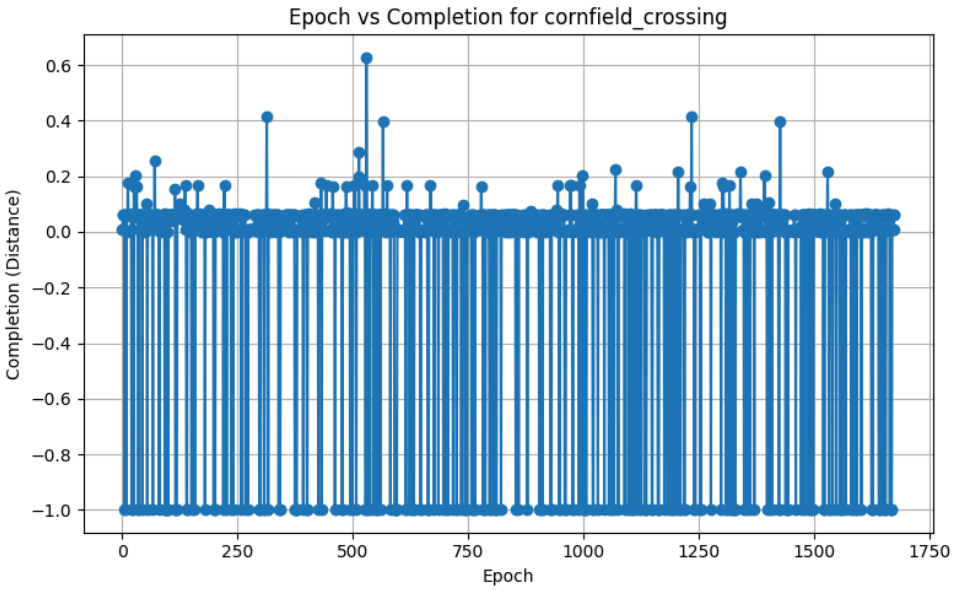

Figure Seventeen: Epoch vs Completion percentage on cornfield_crossing with our second pass at reward generation.

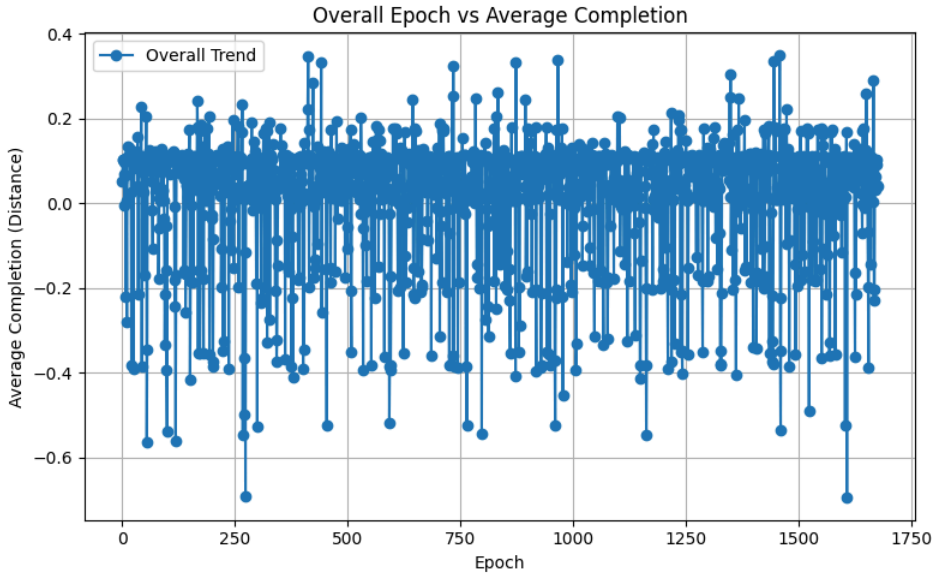

Figure Eighteen: Epoch vs Completion percentage across all tracks with our second pass at reward generation.

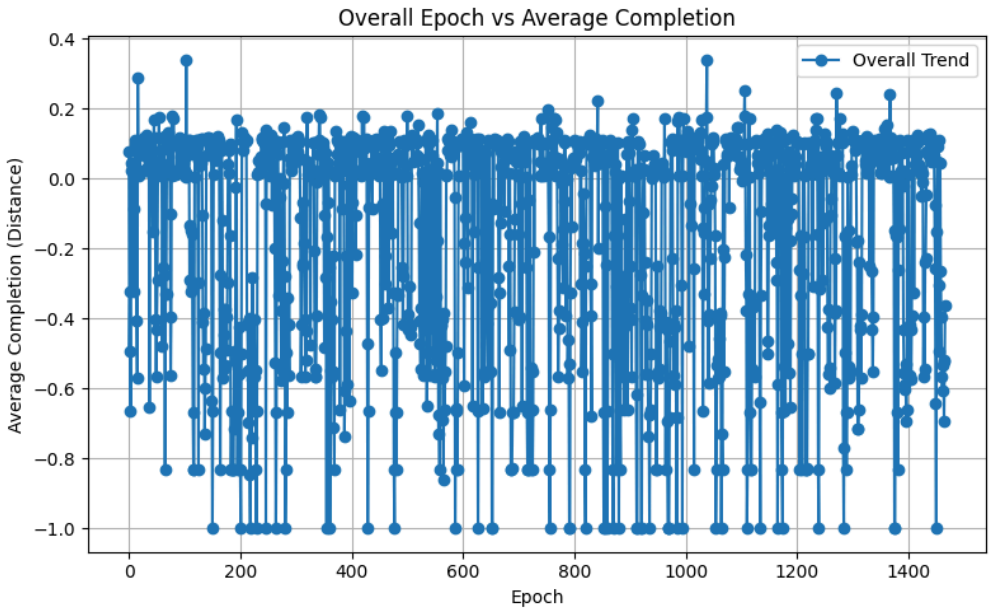

Figure Nineteen: Overall completion percentage vs epoch for log(t) reward shaping.