Deep Learning Final: implementing a FCN for puck-detection in a SuperTuxKart ice-hockey player*

Dustin Hayden **Akshay Patel**

Neil Kamdar **Mustafa Soby**

Abstract

The field of deep learning has blossomed in the past decade. New research has made deep neural nets larger, faster, and more able to tackle complicated problems. Herein we utilize some of these advancements to create a fully convolutional network (FCN) tasked with identifying the existence and screen location of a puck in a ice-hockey video game. We find our FCN performs very well at both detecting the presence and location of a puck. We then use this FCN as well as a hand-designed logic controller to compete in a tournament against other AI agents.

1 Introduction

This report is part of a final project for UT Austin's Deep Learning course:

`http://www.philkr.net/dl_class/`

1.1 Overview

Deep learning, a non-linear function estimation technique, has been applied to many machine learning tasks in recent decades. A major focus of deep learning research involves image content estimation. The network might be designed to detect an object's existence, location, bounding region, pose, or various other features of an image. Herein, we utilize recent advances in deep learning architecture to predict whether an object is present in an image and, if so, where that object is located.

For this task, we utilize *SuperTuxKart*, a 3D open-source arcade racer similar to many cartoon racing video games (Gagnon, 2007). Notably, it has a reduced Python implementation that strips down many gameplay elements, allowing ease of use for computer vision and reinforcement learning instruction (Krähenbühl, 2021). A side-game within this system is a 2 vs 2 ice hockey battle wherein teams must use their go-karts to manipulate a large puck into the opposing team's goal. We sought to create and train a fully convolutional network that could reliably detect the position of the puck. Combining this information with provided state-space data and a custom-built controller would allow two go-karts to automatically attempt to score goals against an opposing team.

We acquired training data, built, and then trained a fully convolutional network (FCN) to detect the presence and location of a puck. On the test set we achieved $\sim 90\%$ accuracy with puck detection. Of the correctly-detected puck-containing images, the predicted location was within 10 pixels ($2-4\%$ of the image size) in $\sim 99\%$ of the images.

Using a hand-tuned controller and this FCN, we were able to score 13 goals across 16 total games against various state-based AI agents, only allowing 4 within the same span. This resulted in a 75% when scored within the context of the Deep Learning course.

*formatted report created with NeurIPS 2020 style and LaTeX file

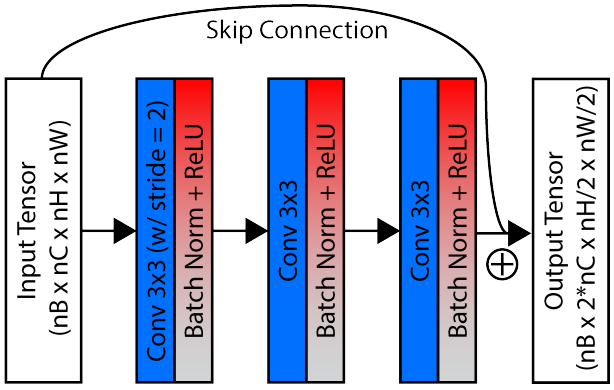

Figure 1: Schematic of a skip block connection utilized for the down-convolution blocks. The input to the block is added into the output of the block before being returned. This allows the network to skip the block.

1.2 Building a Fully Convolutional Network

Deep learning is not a new field. One of the pioneers of deep learning was Frank Rosenblatt, whose 1962 book *Principles of Neurodynamics* proposed using hidden layers within perceptrons (Rosenblatt, 1962). These were unfortunately difficult or impossible to train, requiring hand-tuning of individual weights within the hidden layers. It took two more decades until backpropagation could reliably be used to train network's hidden layers (Rumelhart et al., 1986). However, the lack of computational power and large datasets prevented widespread advancement and utilization of the techniques. Almost three decades later, thanks to improvement in both dataset size and computational power, a deep neural network won an image classification challenge and the deep learning renaissance was born (Krizhevsky et al., 2012).

Our design for this task was a fully-convolutional network. Our design was based on both the U-Net architecture (Ronneberger et al., 2015) and the residual connection from ResNet (He et al., 2016). The architecture uses skip connections to add features seen early on to the later layers of the model. In addition, throughout the architecture, we use batch normalization to control for vanishing and exploding gradients.

The architecture uses three down-convolution blocks of gradually reducing size. Each block first halves the input spatial dimension via a strided and padded 3x3 convolution. This is batch-normalized, rectified to introduce non-linearities, and fed into a simple padded 3x3 convolution (with no striding) to retain the spatial dimensions. The process is repeated once more (again without striding) before being added to the untouched (but still down-sampled via striding) input, allowing the network information to skip the entire block (Figure 1). We use 32, 64, and 128 layers for the down-convolution blocks, respectively. Thus, each tensor that goes in will double the number of channels while halving the spatial dimension.

After this has reduced the spatial dimension by a factor of three, we feed the activations through three up-convolution blocks. These up-convolution blocks double the spatial dimension of an activation (effectively "blurring" the image) while halving the number of channels, reversing the effects of the down-convolution blocks. To help sharpen edges, these up-convolution blocks are also provided the original input to the down-convolution block, allowing for a clearer determination of object edges. A final 1x1 convolution creates an image-sized heatmap (Figure 2).

This heatmap, along with the true object location heatmap, is used as the input to a binary cross-entropy loss function containing a numerically-stable sigmoid implementation.

$$\ell(x, y) = \{\ell_1, ..., \ell_N\}^\top, \ell_n = -w_n \left[p \cdot y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n))\right]$$

In the above equation, x is our predicted heatmap and y is the true heatmap (a matrix of zeros with a single 1 with the central puck location). Each tensor is $[width \times height \times N]$, where N is the batch size. We used positive weight p to weight positive examples more heavily.

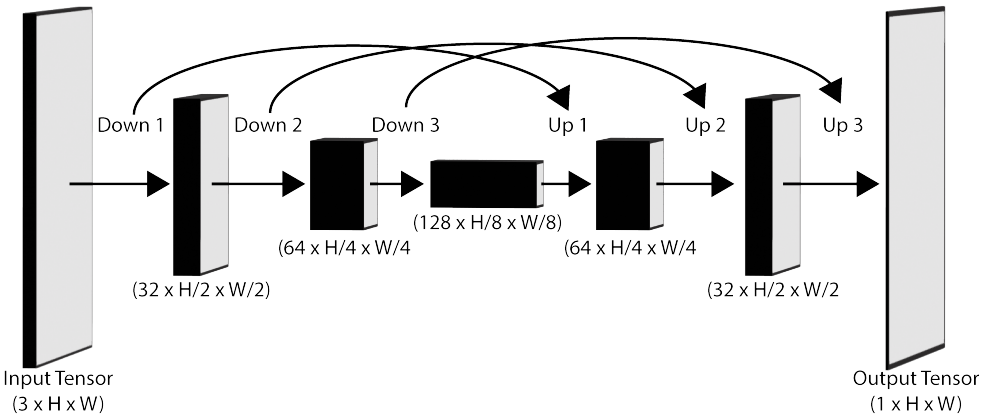

Figure 2: Schematic of our FCN architecture. Both 3×3 down-convolution and up-convolution blocks are used. At each stage, the number of channels are doubled while the spatial dimensions are halved. Each down-convolution block uses an internal skip connection to bypass the block. The tensors that are input to down-convolution blocks are also fed into later up-convolution blocks to sharpen pixel fidelity of the estimated location.

2 Methods

With an architecture and overall design in mind, we had to first create and label images, then train then network, then hand-design controllers to compete in a tournament with our AI agents.

2.1 Obtaining the Dataset

For training, we needed both an image from a game of ice hockey within SuperTuxKart and the puck location (should the puck be in view). A Python toolbox is available for the game allowing for both to be acquired and saved (Krähenbühl, 2021). We simulated matches against all combinations of teams using the provided artificial intelligence implementations. The starting puck position was placed in 1 of 6 positions around the hockey rink and the game was started. The game finished when 1200 frames had passed or one team had scored 3 goals. The remaining 5 positions were then tested. This yielded 60 unique simulations with respect to team and puck starting location. Each simulation had a video recorded from all 4 players perspectives as well as a file containing state information that included, among other things, the puck location.

We post-processed the videos to split them into single images with a sibling file containing the on-screen location of the puck (or NaN if the puck was not present). To accomplish this, we utilized a utility function `to_image` that uses the world-coordinate of the puck as well as a player's camera information to calculate the on-screen coordinate of the puck. However, this function often caused spurious puck locations when the puck wasn't visible, showing the puck as being far off on the background environment. It also didn't distinguish far-away pucks (that might only have a few pixels of visibility) from nearby pucks. Both of these could cause issues with training the deep network due to faulty or inadequate ground-truth labels. To curtail this, we only allowed images to be classified as pucks if the puck was in front of the kart and was within a certain distance of the kart. Conversely, we only allowed images to be classified as not-pucks if the puck was very far away or if it was behind the kart and not nearby (to prevent the camera picking up the puck from the third-person perspective of the camera).

Once these images were created, up to 20 frames per player were selected, each at least 10 frames apart, to create the training dataset. This was selected to prevent biased datasets dominated by any player, video, or a short stretch of time within a video. Each player was only allowed to contribute as many not-puck images (which are more plentiful) as they had puck images, thus ensuring a balanced dataset. This created $\sim 4,000$ images each of puck images and not-puck images (Figure 3). To create the validation training set, which admittedly contains some overlap with the training, we extract every frame that's at least 10 frames apart, not just a balanced amount of each with some maximum. This creates $\sim 18,000$ images ($\sim 7,000$ of which contain a puck).

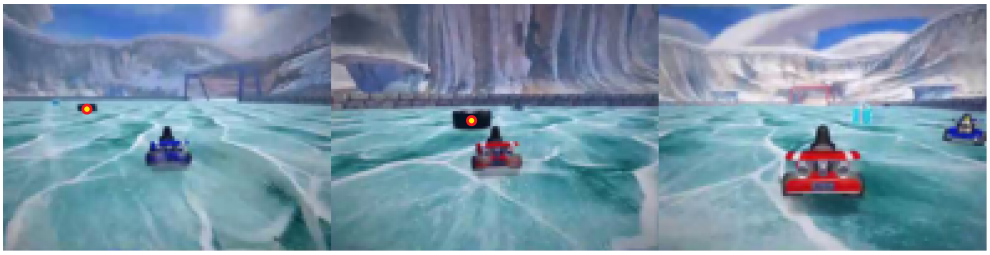

Figure 3: Example correct detections of the puck (left, middle) and correct non-detection of a non-existent puck (right). The yellow dot with a red outer edge indicates the detected location of a puck.

2.2 Training the Fully Convolutional Network

Training a neural network requires the tuning of multiple hyperparameters. These include learning rate, weight decay, the number of epochs, and the positive weight assigned to the puck location pixel.

In our dataset, a learning rate of 0.001 with the Adam optimizer achieved the best results. A larger learning rate prevented convergence and a smaller rate often became stuck in a local optimum. We used a standard weight decay of $1e^{-5}$ for our testing. We ran the training for 12 epochs to avoid overfitting (and the loss appeared to plateau). Additionally, we used multiple transforms on the data (color jittering and random horizontal flips) to artificially inflate our dataset.

Our loss function, as stated above, is the binary cross-entropy between the predicted heatmap and the actual heatmap for the puck location on the screen. The puck location, if it is even present, takes up a single pixel in a $[400 \times 300]$ image. This results in a loss that is heavily weighted towards negative examples. Given this large imbalance between positive and negative pixels, we heavily tuned the positive weight parameter for the loss. Ultimately, we choose a positive weight of 8, giving each positive pixel example 8 times more impact than a negative pixel example.

2.3 Hand-designing a Controller

To begin designing a controller, a perfect FCN was created using the state data of the game (which contained information about all players and the puck location). The provided state data was immediately transformed into a perfect puck-locator, which would always correctly output whether a puck was on screen and, if so, where the screen location was. This allowed the various controller designs to be evaluated before a FCN was trained. Additionally, a dummy opposing team was created for testing strategies. This opposing team did nothing; each act was simply to stay in place. This allowed controller designs to be tested without the nuisance of another (better) AI impeding our attempts to score goals. We tested our designs using the characters Tux, Wilber, and Sara the Racer. Tux is the default kart. Wilber is a lighter kart used for better control, particularly within the hockey mini-game (Wax-stk, 2019). Initial testing found Wilber to be superior to Tux, but a provided AI agent utilizing Sara the Racer routinely out-scored our logic. We thus switched to Sara the Racer and noticed a remarkable improvement in scores.

3 Results

With our trained FCN and hand-designed controller, we quantified how accurate the FCN was at capturing puck information from a still image and how well the hand-designed controllers could score goals with this FCN (or with a simulated perfect-FCN that always knew the exact location of a puck on screen).

3.1 Puck-classification Accuracy

Our FCN outputs a heatmap whose spatial dimensions are the size of the input image. Each location has a value associated with it; positive values indicate a detection at that location, with higher positive values indicating a more certain detection. Within the tournament, our trained network must first detect whether a puck is visible and, if so, where on the screen it is located. Thus, despite the

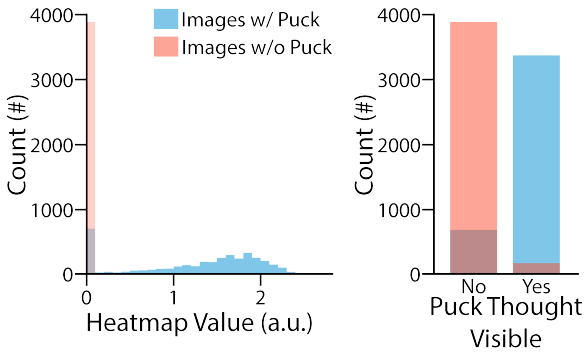

Figure 4: The trained FCN is able to detect the presence of a puck within an image. (Left) A histogram of heatmap values for images with and without a puck indicate a simple threshold can differentiate most images. (Right) Using a threshold heatmap value of 0.05 for classification, the FCN often correctly identifies images with and without pucks.

possibility of multiple peaks within the output heatmap, we created a function that only returns the heatmap value and pixel coordinates of the *single* largest peak.

For each of the $\sim 8,000$ images (both puck and not puck), we extracted the largest heatmap value from our model output. Almost all values for images not containing a puck were 0, whereas the heatmap values for images containing a puck were distributed between 0 and 2.5 (Figure 4). We selected a threshold of 0.05 for a binary classification. If the heatmap value is below this score, a puck is not detected. If above, the puck is detected. This creates a generally accurate prediction, with pucks being correctly identified 83.0% of the time and images without pucks being correctly identified 95.6% of the time (Figure 4). Overall, this leads to an accuracy of 89.3%. Thus, our trained network is able to capture whether or not a puck is on screen most of the time.

However, this could be caused by overfitting to the training set. We took our validation set (which admittedly contains some repeat images from the training set; see 2.1) and performed the same classification accuracy measure. Pucks are correctly identified 83.0% of the time and images without pucks are correctly identified 96.0% of the time, giving an overall accuracy of 89.5%. Thus, our trained network is able to capture whether or not a puck is on screen most of the time.

During gameplay, our FCN must return the peak heatmap value twice (one for each image provided to our team of 2) within 50 milliseconds. Thus, we timed how long a prediction took. For each image, on a NVIDIA GeForce GTX 1080 Ti, the model took a median 4.1 milliseconds to predict a heatmap and extract the maximum value and location, well within our limit.

3.2 Puck-location Accuracy

Our model can accurately determine the presence of a puck. However, can it accurately predict the location of a puck? To determine this, we only took puck images that had correctly been identified as containing pucks. We extracted the screen location of the largest heatmap peak and compared it to the known true position of the puck center. The Euclidean pixel distance between the two locations was calculated (Figure 5). Our model accurately located the puck position to within a few pixels of the true location (median distance = 1.4 pixels). We used a threshold of 10 pixels ($\sim 3\%$ of the image) to determine if the prediction was nearby. Surprisingly, 98.8% of the predicted locations were nearby (Figure 5). Similarly, in our pseudo-validation set (see 2.1), the predicted location is nearby 98.6% of the time. Thus, our model is extremely accurate at predicting the puck location (if a puck was detected).

3.3 Controller Scoring with Perfect and Trained FCN

Obtaining the puck location (if it exists) is only the first step of our AI player. We must use this location, as well as provided knowledge of the kart and goal, to guide the hockey puck into the opposing team's net. To optimize the controller portion of this project we first created a "perfect FCN" that took the provided kart and puck world coordinates and transformed them into what a

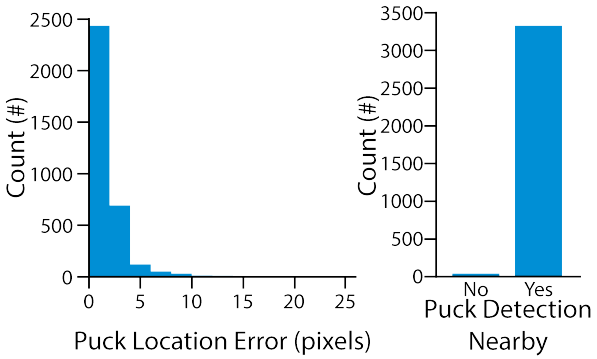

Figure 5: The trained FCN is able to predict the location of a puck within a puck-containing image. (Left) a histogram of the Euclidean pixel distance between the actual and the predicted puck location (for puck-containing images). Most errors are within 10 pixels (of a 400×300 image). (Right) Using a threshold of 10 pixels, most detected locations are near the true location of the puck.

perfect FCN would output. Namely, whether or not the puck was visible and, if so, where was it on the screen?

Various hand-designed controllers were tested using the perfect FCN. For the final project, the grade was based on goals scored against other AI agents. However, we additionally looked at goals allowed when designing and testing our controllers. Each member worked on their own implementation while sharing overarching ideas. The results of these controllers are summarized in (Table 3.3). Below we explain the design of the winning controller (using the perfect FCN).

The first thing the controller does is obtain the screen location of the puck in normalized coordinates. Then, using the provided player state information, various features are extracted. These are the angles and distances to the player's goal, to the opponent's goal, to the center of the arena, and to the partner player. The angles are bounded by -180 and 180, with a negative angle meaning a left turn. Next, the controller checks whether a new round has begun. It does this via a combination of the current player location (i.e. is it near a starting point) and whether there was a sudden jump in location. If a new round is detected, both players speed towards the origin for a set amount of frames until the puck is seen and the subsequent logic takes over.

The logic for normal game play follows a series of checks. First, if the puck is visible, steer towards it. Second, if it isn't currently visible (but it was recently visible), reverse in the direction opposite where it was last seen. This should theoretically reveal the puck once more in many cases. If that fails, check if your partner sees the puck. If they do, steer towards your partner's known location. Finally, if you cannot see the puck, it wasn't recently seen, and your partner can't see it, return to the origin and make wide loops until you or your partner can see the puck.

While this logic theoretically works well, often the kart can get stuck within a goal or accelerating into a wall. Thus, a check uses the maximum absolute difference in the stored previous 10 location coordinates to detect if the kart hasn't moved beyond a threshold. If it hasn't, the kart enters into a rescue logic that reverses opposite the direction of the origin.

Finally, the current steering value is checked. If it is above a threshold, drifting is engaged.

Of particular note is the fact our winning logic seeks merely to control the puck and not score goals. Indeed, nothing within the winning logic seeks to put the puck into the opponents goal. Instead, we rely on the initial new round start detection and overall time spent with the puck to increase our chances of scoring a goal. Surprisingly, all efforts to steer the puck towards the goal reduced our scores.

With these results in mind, we replaced the perfect FCN with our trained FCN and only ran the best controller. Surprisingly, this performed better than the perfect FCN. Our final submission scored 13 goals over 16 games while only allowing 4 goals, a marked improvement over the perfect FCN results.

Name	Goals Scored	Goals Allowed
Controller 1 (Perfect FCN)	6	19
Controller 2 (Perfect FCN)	13	28
Controller 3 (Perfect FCN)	16	16
Controller 4 (Perfect FCN)	17	17
Controller 4 (Trained FCN)	13*	4*

Table 1: Cumulative scores for 32 games. The final entry (marked with an asterisk) was only for 16 games.

We attribute this improvement to three things. Firstly, luck; the results were consistent when running locally but completely different when run on different machines, indicating some stochastic features to the game. Secondly, our perfect FCN was based on the training image logic, which specifically "didn't see" pucks that were behind or near the edges of the player (to prevent contaminated training images). This could have caused the perfect FCN to trick players into thinking they lost the puck more often than not. Finally, the imperfect predictions of a trained FCN may have caused enough adjustments and randomness to score a few more goals. We found evidence of the highly-sensitive nature of this scoring with our hyperparameter-controller-tuning; minor adjustments in global constant multipliers could drastically change the score results.

4 Discussion

Herein we create an AI to play a 2 vs. 2 game of ice hockey using *SuperTuxKart*, a 3D open-source arcade racer (Gagnon, 2007). We obtained training images using provided code (Krähenbühl, 2021) and our own utility functions. These images were sorted and, if a puck was present, the screen location of the puck was stored. We then designed a fully convolutional network similar to the U-Net architecture (Ronneberger et al., 2015) with residual connections from ResNet (He et al., 2016). This model was able to accurately detect the existence of a puck within an image and, if the image had a puck, where it was located. Finally, we hand-designed controllers that would take the screen image of our players, predict puck presence/puck location, and use provided information about the goal location and player position to navigate the puck into the goal. This was able to score reasonably well against various state-based AI agents, scoring 13 goals over 16 games while allowing only 4 goals.

Numerous improvements could be made to our design. These include: obtaining better/more data, creating a better network, using a better validation set, training more robustly via programmatic hyperparameter search, and a better-designed (or perhaps even reinforcement-learning-created) controller. Each of these will be elucidated upon in turn.

Firstly, our dataset of images contained a few erroneous labels. For instance, a puck may be technically visible (in terms of our calculations), but the image itself had another player or perhaps a pickup item blocking it, meaning a "puck-containing image" actually had no visible puck. We would need to ensure the vector projection from the third-person camera position to the puck was unobstructed by another player/item. However, we would also want some images of the puck only being partially obstructed by obstacles (to help the FCN learn detection in crowded situations). Additionally, there are also false positives within the dataset. For reasons we don't understand the puck would sometimes be "located" in the background scenery. Despite our attempts to programmatically detect and remove these false positives, a few are still within the dataset.

Second, our network could be better designed for the task. Ostensibly we could design a network, or multiple networks, that performed separate functions. One would solely detect the presence of a puck on screen, another would detect the location of puck-containing images, and a final network would detect the size or bounds of the puck to help with distance estimation. We could also train not on puck screen location, but rather the distance and angle in game-coordinates for a more direct applicability to our controller. Better yet, we could train a model that tracks puck location over time such that we could more easily recover position when the puck was lost. Unfortunately, initial attempts to acquire labeled training data containing puck distance, size, bounds, or angle were error-prone, so we settled on the simpler puck-detection and screen-location labeled training set. Given the relative simplicity of such a task, and the minimal size of our network, our inclusion of skip and residual connections is likely unnecessary. However, given we are within the time frame allotted, we didn't try to simplify the network.

Third, our validation dataset and training dataset overlapped. While we theoretically had access to infinite data (due to provided agents and code), testing showed that only the initial puck position and up to the first scored goal provided noticeably different matches. After a goal was scored, the puck was reset to the middle position and the game played similarly. There were also multiple instances of the provided agents getting stuck in a loop and not providing appreciably different data. Our validation dataset was just all good images (see 2.1) containing either a puck or no puck whereas the training dataset was a random subsampling of this. Thus, our model could have easily overfit the training and validation data. To obtain a better validation set it would be preferable to have an entire suite of new agents, acquire data from them, and predict the outcome.

Fourth, we could hyperparamter tune more robustly. Programmatically checking various hyperparameters and storing their metrics would allow a better, more robust model to be created. Additionally, parts of our controller used global constant values as thresholds or multipliers and slight changes in these values caused massive changes in our score results. These values, too, could be tuned to score better.

And finally, our hand-designed controllers could be improved or outright replaced with a reinforcement-learning-created controller. While our logic is sound, it is mostly focus on tracking and keeping possession of the puck. Goals are merely an unintended consequence and likely the result of getting lucky with the initial hit

In summary, we created data for a FCN, trained a FCN to predict the puck screen location, and used the FCN and a hand-designed controller to score goals against opponent AI. Our solution was able to score goals and defend our own goal, but these both were likely unintended consequences of the main logic which was merely to detect and possess the puck. Further work could elucidate a better strategy for detecting when the puck was in our control and how to angle it towards the opponent's goal.

5 References

[1] Gagnon, M. (2007). *SuperTuxKart*. GitHub. Retrieved October 9, 2023, from https://github.com/supertuxkart/stk-code

[2] He, K., Zhang, X., Ren, S., Sun, J. (2016). Deep Residual Learning for Image Recognition. *IEEE Conference on Computer Vision and Pattern Recognition*, 770-778. https://doi.org/10.1109/CVPR.2016.90

[3] Krizhevsky, A., Sutskever, I., Hinton, G. (2012). ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, Vol 60, 84–90 https://doi.org/10.1145/3065386

[4] Krähenbühl, P. (2021). *Python SuperTuxKart*. GitHub. Retrieved October 9, 2023, from https://github.com/philkr/pystk

[5] Ronneberger, O., Fischer, P., Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. *Medical Image Computing and Computer-Assisted Intervention*, 9351, 234–241. https://doi.org/10.1007/978-3-319-24574-4_28

[6] Rosenblatt, F. (1962). Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms. *Spartan Books*. http://catalog.hathitrust.org/Record/000203591

[7] Rumelhart, D., Hinton, G., Williams, R. (1986). Learning representations by back-propagating errors. *Nature*, Vol. 323, 533-536. https://doi.org/10.1038/323533a0

[8] Wax-stk. (2019). *SuperTuxKart Speedrun Guide*. speedrun.com. Retrieved November 13, 2023, from https://www.speedrun.com/stk/guides/ocm26