CSCI 599 Applied Machine Learning in Hovenstar

Members:
Zhewen Miao, 7336060923
Michael Yuen, 1039985005
Yifan Meng, 2111119233
Ye Xiao, 3517078008
Mengchen Tan, 4423672224
Chengxiang Duan, 3288443478

Abstract:

Applied Machine Learning in Hovenstar is a two-fold Machine Learning project that deals with Reinforcement Learning in a turn-based game with a large variety of different problem states and single-player content creation with Generative Adversarial Networks on different tiles in Hovenstar. The Reinforcement Learning strategy revolves around Proximal Policy Optimization to create a more durable and versatile artificial intelligence to play against in the game. The Generative Adversarial Networks are made to uniquely generate tiles and environments for the player to experience more variety in playable scenarios. The overall reasoning behind applying Machine Learning algorithms to Hovenstar is to improve the game quality of a single-player game and to speed up the single-player content creation pipeline in developing scenarios.

Introduction:

Hovenstar is a single-player tactical turn-based game. The game holds a campaign of different scenarios that are scripted. Each scenario allows the player to use different units to accomplish different objectives, battle against enemy artificial intelligence (AI) controlled units, and fight in different environments. The player can individually move each unit and attack enemy units. The enemy can also do so turn by turn, swapping turns with the player. The basic gameplay loop requires the player to make trades that are beneficial in health over multiple turns until the player can take over the board and eliminate all enemy units.

With this form of game, a robust AI is needed to produce the best possible single-player experience. Initially, the game had an AI revolving around a complex state machine and behavior trees per scenario of the game. This can only hold for so long; different scenarios and units enforce a different playstyle per scenario and the player can adapt against the AI quickly, finding abusable points in the behavior trees. This calls for a more robust AI algorithm, Reinforcement

Learning (RL). This algorithm relies on the AI learning on the agent's own to figure out the best patterns and strategies to utilize against the player. The AI relies on finding the optimal strategy through rewards, punishments, and observations given. The AI then saves the best-trained rounds of play that have been given the most reward and the least punishment. With this algorithm, the AI will be more robust in playing with different units and scenarios.

Another issue in the game is how static the scenarios are; once a player plays through all of the given scenarios, the player has no way to advance. Single-player content is difficult to create, due to the amount of time and effort needed to make new and innovative single-player content. To tackle this issue, Generative Adversarial Networks (GANs) can generate new images for Hovenstar to grant more tiles for different boards and make new units for players to play with. GANs use a Generator Neural Network (Generator NN) to generate fake images and a Discriminator Neural Network (Discriminator NN) to classify images as fake or real, based on a dataset of real images. Once the images created by the Generator are able to disguise the images well enough to be classified as real by the Discriminator, the images are strong enough to be used as new tiles and units for Hovenstar. This simplifies the process of creating new single-player content by removing the need for a dedicated artist, allowing for less time spent in creating single-player scenarios.

Reinforcement Learning

Related Work:

RL has been a part of AI since AI's beginnings; trial-and-error learning has existed for eternity, with systems like MENACE, which played tic-tac-toe and gained or lost beads upon winning or losing matches of tic-tac-toe. The machine learned from a "teacher" through these basic rewards and punishments through randomly doing moves in a game of tic-tac-toe. Further development pushed into Q-Learning, which utilized this basic Markov Decision Process (MDP) structure with observations, actions, rewards, and punishments. The parameters are passed into a graph model to gather values with backpropagation and a policy function that can adjust rewards and punishments using a discount function. Dynamic Programming (DP) was also researched to bolster the storage of rewards and punishments, further making decisions in systems like MENACE easier to compute. Q-Learning uses DP via the Q-Table to store previous results, known as Q-Values, and to find an optimal path in the table. Q-Values load in both the action taken and the long term discounted reward in order to determine the best action to take within an environment. Fast forward to Deep Q-Learning: Google, Open AI, and many other companies are developing Deep RL AIs with high computational power to increase training speed and potential. This is utilized in Google's DeepMind project in applying Deep Q-Learning to Atari

games, DeepMind's AlphaGo that beat professional players in the game of Go, and Open AI's Dota 2 AI that beat professional players in Dota 2. In particular, the AlphaGo AI doesn't rely on computer vision tactics and Convolutional Neural Networks (CNN) but rather pushes for Monte Carlo tree searching with Q-Value optimization in the Neural Network (NN), which is similar to the method used in our RL strategy. Deep RL is the future to game AI, and in a single-player game like Hovenstar, this can be a game-changer in how the game typically plays out.

Technical Approach:

RL focuses on trial and error learning, different from Supervised Learning, which holds a clear output or goal that needs to be reached. Supervised Learning also holds a clear dataset to train on, which as indicated above, Hovenstar doesn't have an extensive dataset to train off of. An agent is given observations about an environment. This agent does actions in this environment and is given rewards and punishments based on how successful these actions are towards the agent's goals, without previous knowledge on how other AIs or players played the game.

In Hovenstar, the game state observations include the turn of the player, with 0 showing the current player as unable to move and 1 showing the current player as able to move. The observations also include both the player's units and the enemy's units, each with their individual positions, health, move range, attack range, and attack value. Lastly, the observations include map size and tile type. These observations allow the agent to optimize a strategy to do one of the following actions for each unit under control: idle, move, attack without moving, or attack with moving. If the agent loses all units under the agent's control, the agent is punished with -1 to the reward, and if the agent wins by killing all enemy units, the agent is rewarded with +1 to the reward.

Further rewards were implemented to experiment with a more offensive AI and a more defensive AI. The reward will be used within a policy function to optimize the reward gained. To foster an offensive strategy, points were given to attacking and killing enemy units, along with negative rewards upon being idle during turns. To foster a defensive strategy, points were given to staying alive, not taking damage, and removing points for how long a match takes in order to prevent not moving during turns.

Proximal Policy Optimization (PPO) is a strategy that increases the speed of computation to be faster than Q-Learning. Using KL Divergence to determine the ratio between new and old policies, PPO uses Trust Region Policy Optimization to maximize the long term reward and penalize the agent for a large difference in policy. If an action is greater than 0 in reward, the action is encouraged and the ratio of policies is increased, and if the action is under 0, the action

is disregarded and the ratio of policies is decreased. This analysis of ratios in policies is fed into the objective function to decide upon the most rewarding action to take. PPO allows for optimized training with our agents and scenarios, running more episodes of our game in less time.

Software Written and Datasets Used:

Hovenstar is a previously built game in the Advanced Game Projects course. Unity's ml-agents game asset is also used, which holds the PPO algorithm. Software that the team wrote includes connecting Python to Hovenstar's Unity and C# codebase, adding rewards to actions, incorporating the Hovenstar environments to fit with the ml-agents game asset, and a player brain that allows users to train agents based on the player's movements.

Since the algorithms are reliant on trial-and-error methods and rewards to function, there are no datasets involved in training the models. There is data coming directly from the player-brain and stored previously trained models, but since the game is an unreleased game, there is no public player base's data and inputs to train with.

Experiments, Evaluation, and Difficulty:

To evaluate our models, the team chose to simply check for which models and rewards assigned prompted the best gameplay loops as the team played against the AI and pitted each model against each other. The team created two different scenarios to test the AI on, a 1 unit versus 1 unit scenario and a 3 unit versus 2 unit scenario. After training for 600k steps with PPO, the variance of the reward is high, leading from 0 reward to -16 reward. The average value is around -4, which means that the reward structure is not optimal or the agent is not training to an optimal value. This means that there isn't a clear victor in reward strategies and training is not stable.

Due to the difficulty in detecting a numerical difference in the algorithms or reward strategies, along with no visual differences, RL may not be the perfect choice to dealing with a constantly changing turn-based game with countless scenarios. The algorithm would have to train over significant amounts of time for each scenario, only to not be viable in another scenario. Both ml-agents and Hovenstar's APIs are also in flux; the implementations of both of these programs and the connections between Unity and Python are not perfect, leading to constant necessary debugging. The pipeline to adding a new scenario to work with RL takes far too long for the limited computational power that the team has, along with limited results that are gained in comparison to the state machine AI.

Generative Adversarial Networks

Related Work:

GANs are relatively new in deep learning. GANs became a study point in art and generating images when a Stanford student created an open-source project to generate artwork images and had an artwork generated from the network sold in a museum. GANs were then further pursued with Google's style transfer project known as Deep Dream, which showed NNs generating images in different styles that copy other images, and Deepfakes, which are fake images and videos of different humans doing different activities that the humans did not do. In other video games such as Final Fantasy and Resident Evil, GANs are used to make images and videos of older games much sharper in resolution or to generate art and cutscenes that can be used within the game's world. GANs are a hot topic for research among different existing applications that require artwork to generate: Hovenstar's single-player experience is a place that would benefit from GANs. Single-player content requires huge staff commitment to art, music, writing, game design, and programming, leading to a large need of staff and communication between the people working on the content to have the same vision and ideas. Most games that are released within the past few years are multiplayer heavy, especially those produced by large AAA companies, as the need for artists, musicians, and designers are significantly lowered for the amount of content made. However, single-player content is something that has an audience, yet high quality and lengthy single-player content is not easy to create. With GANs, this issue can be solvable, as shown within experimental research in level generation of games like Mario and DOOM.

Technical Approach:

Hovenstar's limited datasets will prove to be an issue, but with data cleansing and data gathering from different sources like Unity's game assets and RPGmaker's game assets, there should be enough data to train the models. The dataset consists of 482 tiles expanded into 48200 images using data augmentation, which include rotation, addition, multiplication, blurring, blending, translation, sharpening, adding noise, erosion, and dilation. The original images are all expanded by these augments to make enough data for the GAN. These tiles are split into 10 classes, including brick, city, desert, forest, lava, space, stone, water, and wood. The GAN is trained on some of these 10 classes and with the dataset altogether.

GANs rely on a Generator NN and a Discriminator NN. The Generator NN generates images by randomly generating noise and building slowly into images that correlate to the dataset. The Discriminator NN classifies images as real or fake based on the same dataset. The

ultimate goal of these two NNs are to have the Generator create images that the Discriminator cannot decipher as fake from real images in the dataset. The Discriminator NN is a CNN that tackles the issue of classification and the Generator NN is a Deconvolutional NN that aims to fool the Discriminator's classification. Both of these NNs have backpropagation added to them in order to make both networks better at challenging one another.

The Generator NN is filled with transposed convolutions to do deconvolutional operations or the reverse of a standard convolutional layer. The transposed convolution operation does the same as a standard convolutional layer, only the output is a larger output; the spatial transformation is reversed. The operation upscales the image, reconstructing the image through the reverse convolutions. The Generator is relatively complex, with 4 repeated layers of transposed convolutions, cropping, batch normalization, and ReLU activation layers, and a final transposed convolution, cropping, and a Tanh activation layer. This model uses a kernel size of 4 and divides the total number of filters by 2 starting from 1024 filters until reaching 3 filters, outputting an image of size 96 x 96 x 3. The Generator NN requires an input of some randomized noise in order to change this noise into an image.

The Discriminator NN is a standard CNN; instead of transposed convolutions, normal convolutions are used, which create smaller outputs in order to finally generate a classification value to indicate which class the Discriminator chose. Leaky ReLUs are used as the choice of activation function over ReLUs, since the gradients need to have access to negative numbers in order to make gradients flow stronger into the Generator NN. With 4 repeated layers of Leaky ReLUs, padding, batch normalization, convolutions, and a final flatten operation, the Discriminator is complex in the layers used. The model uses a kernel size of 4, a batch size of 64 images, 64 filters multiplied by 2 until the layers are flattened, and expects inputs of 96 x 96 x 3 images.

In order to deal with computational problems, difficulties such as one Neural Network being too strong, and a lack of evaluation, the team will use Google Colab's Tesla K80 GPU to train the network, running multiple parameters at various learning rates and network architectures. This will be an easier way for us to evaluate the results, as the images are simpler to determine visually whether the images are good enough for usage in Hovenstar.

Software Written and Datasets Used:

Datasets used include an anime character dataset, which is simply a testing dataset in order to start developing the GANs and testing different NN architectures. For the Hovenstar dataset that was gathered over time, the team used data augmentation on the original 482 gathered images to expand the dataset to 48200 images, split into 10 classes. The data

augmentation process relies on different algorithms provided by OpenCV and Sci-kit Image to generate a more versatile dataset.

Software written by the team include the Generator and Discriminator NNs. The NNs rely on Keras and Tensorflow as the frontend and backend respectively; the backpropagation, loss functions, optimizers, layers of convolution, activation functions, and data loading are all handled by Keras and Tensorflow. Putting together the framework into two usable NNs is a major part of the software written. Also, jupyter notebooks and Google Colab were used to train the model and optimize the training speed.

Experiments and Evaluation:

Using the simple Generator and Discriminator on the anime character dataset, after 1000 iterations, the images gained colors in the right places (generating hair color and skin color). After 5000 iterations, the images started to resemble anime character faces, and after 20000 iterations, the images consistently looked like full-fledged anime character faces. The overfitting limit was around 70000 iterations, in which excessive noise was shown on each image. There are still images in the 20000th iteration that are not perfect, but the NNs seem to do well on this dataset.

With the Hovenstar tile dataset, the results were very similar, in which overfitting reached 60000 iterations and training results looked best at around 15000 iterations to 20000 iterations. The entire dataset's results were much more varied and had a large difference in color and formation, while specified class' results were very specific in form, occasionally had bias to a certain image if that class didn't have enough data, and had bias to augmentation forms that were used, making the results too bright or dark. The Hovenstar tile dataset still worked very well since the dataset isn't that large and some classes were very small in a variety of tiles.

The NNs are optimized to give the best results, as many iterations of different NN architectures and optimizers were used. The optimizers used include RMSProp and Adam, and the loss function is the mean between real images classified by the Discriminator and fake images classified by the Discriminator upon generation by the Generator. Over many iterations, the lowest loss between the Discriminator and Generator is at 20000 iterations for the Hovenstar dataset on all tiles, having 0.871582 loss, in comparison to 7.104878 loss at 1000 iterations. The RMSProp optimizer seemed to be better visually than the Adam optimizer, creating clearer images with less noise and clearer form.

Training the GAN took extensive amounts of computing power; without Google's Tesla K80, the GAN wouldn't even be finished training now. Luckily, Google's GPU proved to be

relatively quick at training, only requiring around 10 hours of training per specified class to reach 20000 iterations. Still, 10 hours is too long to train such a model, clearly demonstrating the need to optimize and refactor the code to simplify memory needs and reducing the computational complexity. The GAN needs to be simplified in layers without reducing accuracy, bigger batches of images need to be loaded in, and a better loss function needs to be used, such as Sparse Categorical Cross Entropy or Binary Cross Entropy. These optimizations would greatly hasten the GAN in training.

Conclusion:

Hovenstar's core gameplay loop heavily benefits from a robust AI in the game's unique scenarios and units. Despite that, RL with PPO does not seem to work well without significant computation and specific scenarios to emphasize learning on. This algorithm relies on trial and error learning, but the algorithm must learn and train upon every scenario individually. Single player content creation takes a lengthy process to create new single player scenarios in Hovenstar. This problem requires GANs to speed up the process by removing the need for dedicated artists in drawing environments and tiles, in which the GAN built works accurately, creating a variety of tiles in different classes that can be used in Hovenstar.

Future tasks to tackle include trying to gather a dataset of environments that are fit for Hovenstar to run the GANs on. This would allow for more content to be created without an artist. An even more difficult task with the GAN is to create new units in the game that fit within the environment and tiles; the dataset of similar looking units is very limited since Hovenstar's units are distinct. Placing these units, environments, and tiles together into Hovenstar's scenarios would be the ultimate goal to reach, using strategies such as tile stitching to place tiles together in an appealing format. Future tasks for the RL portion of the project include using Curriculum Learning and utilizing Transfer Learning to transfer the RL between scenarios in order to have the agent learn significantly faster. These tasks are all very difficult due to the way Hovenstar was built; with steady updates and bug fixes, Hovenstar can use ML algorithms to add to the quality of life in the player experience.

References:

- **Reinforcement Learning**
  - **History**
  - http://www.incompleteideas.net/book/ebook/node12.html
  - https://en.wikipedia.org/wiki/Reinforcement_learning
  - https://www.geeksforgeeks.org/what-is-reinforcement-learning/#targetText=Reinforcement%20learning%20is%20an%20area,take%20in%20a%20specific%20situation.
  - https://medium.com/@SeoJaeDuk/archived-post-a-history-of-reinforcement-learning-prof-a-g-barto-849477846086
  - http://matt.colorado.edu/teaching/highcog/readings/sb98c1.pdf
  - https://arxiv.org/pdf/1708.05866.pdf
  - **Technical Approach**
  - https://hackernoon.com/reinforcement-learning-and-supervised-learning-a-brief-comparison-1b6d68c45ffa
  - https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287
  - https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-part-ii-trpo-ppo-87f2c5919bb9
  - https://medium.com/@SmartLabAI/reinforcement-learning-algorithms-an-intuitive-overview-904e2dff5bbc
  - https://medium.com/@jonathan_hui/rl-reinforcement-learning-algorithms-quick-overview-6bf69736694d
  - https://www.nature.com/articles/nature16961
- **Generative Adversarial Models**
  - **History**
  - https://skymind.ai/wiki/generative-adversarial-network-gan
  - https://techxplore.com/news/2018-05-adversarial-networks-unleashed-video-games.html
  - **Technical Approach**
  - https://medium.com/ai-society/gans-from-scratch-1-a-deep-introduction-with-code-in-pytorch-and-tensorflow-cb03cdcdba0f
  - https://towardsdatascience.com/image-augmentation-examples-in-python-d552c26f2873
  - https://medium.com/@thimblot/data-augmentation-boost-your-image-dataset-with-few-lines-of-python-155c2dc1baec
  - https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d
  - https://machinelearningmastery.com/tour-of-generative-adversarial-network-models/
  - **Tiles**
  - Hovenstar's Tiles
  - https://www.moddb.com/games/mech-commander-2

- https://assetstore.unity.com/packages/2d/textures-materials/alien-tile-set-116827
- https://assetstore.unity.com/packages/2d/environments/free-wall-tile-set-130514
- https://assetstore.unity.com/packages/2d/textures-materials/stone/hand-painted-textures-stone-tiles-51535
- https://assetstore.unity.com/packages/2d/environments/too-cube-forest-the-free-2d-platformer-game-tile-set-117493
- https://assetstore.unity.com/packages/2d/environments/2d-forest-tileset-pack-toon-style-93499
- https://assetstore.unity.com/packages/2d/textures-materials/floors/five-seamless-tileable-ground-textures-57060
- https://assetstore.unity.com/packages/2d/textures-materials/floors/tileable-hand-painted-ground-texture-pack-2-62018
- https://assetstore.unity.com/packages/2d/textures-materials/dynamic-space-background-lite-104606
- https://assetstore.unity.com/packages/2d/textures-materials/flat-textures-demo-141275
- https://assetstore.unity.com/packages/2d/textures-materials/floors/20-man-made-ground-materials-12835
- https://assetstore.unity.com/packages/2d/textures-materials/10-texture-sets-industrial-02-17624