Panda Trainer

Emulating Human Behavior in Games using Computer Vision

Logan Hollmer
Computer Science
California State University - Sacramento
Sacramento, California, U.S.A
lhollmer@csus.edu

Quinn Roemer
Computer Science
California State University - Sacramento
Sacramento, California, U.S.A
quinnroemer@csus.edu

Abstract

Unlike humans, computers often lack the ability to interpret visual data and react to it. However, with modern techniques, we can demonstrate that a computer is able to analyze and react to similar image-based stimuli.

In this research project, we will use computer vision to enable a computer to play a game as if it was a human through the use of images.

In order to achieve this goal, we will create deep learning convolutional neural networks, both with and without transfer learning. These models will be tuned to achieve the best results through a genetic algorithm. The models will be trained on a dataset created from a custom-built game. This input is the result of conventional human players. After training the models, their performance will be measured both in and out of the game.

The results achieved suggest that computer vision is capable of performing complex tasks, such as playing a game, with only images as input. The models were able to more consistently complete a level in the game than their counterpart human players. In addition, they showed a small ability to analyze and react to untrained situations.

1 Introduction

This project (Panda Trainer) will address the problem of computer vision, specifically using computer vision to perform human-like tasks. In this project, a CNN network is used to play a custom-built 2D game called *Panda Runner*. This is an important problem as there are many applications for computer-vision to perform human-like tasks. This project will advance the understanding of how machine learning can be used for these types of applications. The contributions for this project include:

- Creating a custom 2D platformer/runner game.
- Creating a custom dataset.
- Creating and training both custom and transfer learning CNN models.
- Using a genetic algorithm to fine-tune hyperparameters.
- Using computer vision to solve human-like tasks such as playing a video game.

This paper will go over the problem formulation, system/algorithm design, experimental evaluation, related work, and conclusions. Lastly, it will discuss the work division between team members and what was learned.

2 Problem Formulation

The end goal of our project is to be able to emulate human behavior using similar input. For example, when a person sees an image on-screen they are able to almost instantly formulate a response or action to take. In light of this, we will focus on processing similar input. We will be using the entire image displayed on-screen on a frame-by-frame basis. This will be directly sent to our computer vision models after preprocessing. Once done, our models will predict the correct action based on the image as a whole. In this particular problem set, there are only 3 possible actions that our models can take. Those being, no action, jump, or slide. The probability of each action was then examined and the action with the highest probability of being correct used as our output.

3 System/Algorithm Design

This section will discuss the system design and the steps that were taken over the course of this project.

3.1 System Architecture

The first step taken in our project was to identify a game to use as our input for our models. However, after examining several open-source options we quickly decided that our project would be best served with a custom-built game to satisfy our project requirements. In light of this, the game used (Panda Runner) was developed over the course of 3 weeks. It was programmed in Python using the open-source PyGame library. Panda Runner is essentially a 2D runner/platformer game that involves jumping over and sliding under obstacles while collecting coins. The goal being, to collect the most coins while also completing the level. There are 4 types of objects in the game. Boxes that the player must jump over, vines that must be slid under, coins that should be collected, and lastly saw blades that result in the game starting over if the player collides with them. The total score the player earns is the weighted sum of the distance traveled and the number of coins collected.

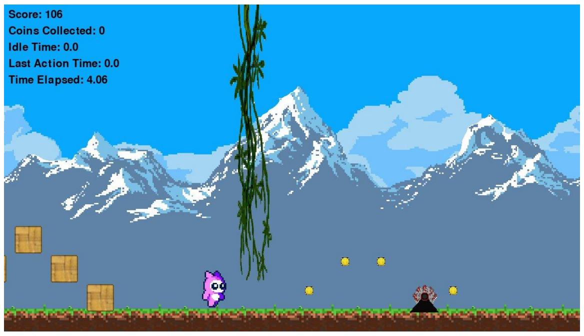

Figure 1: Image from the game Panda Runner. The image contains all 4 objects as described in the text above.

Once the game was developed, image data was collected for training. A method was created to take a screenshot every N seconds where N was a float set between 0.2 and 1. A screenshot was also taken whenever a player jumped or slid. All of this data was saved with the current action being taken at the time of the screenshot. Possible actions were None (No action), Jump (player had pressed the jump key), or Slide (player had pressed the slide key). A total of 5095 images were collected during this process from two different human players.

Once the data was collected it was preprocessed and saved as pickled Numpy files for easy future access. The data was preprocessed by resizing the images from (1280, 720, 3) to (224, 224, 3). This was done to help reduce the complexity of the images, allowing our models to more quickly process images. In addition, this resolution was the same resolution that our base transfer learning models were trained on. During this process, the labels were also extracted and saved alongside the images. Lastly, the data was normalized and converted into the *float32* type. However, the labels were treated differently. As they are categorical data, we performed One-Hot-Encoding on them to produce binary columns in order to perform supervised learning. Before training our models a train/test split was performed on our dataset. In total, 75% of the dataset was kept as the training data while the remainder served as our test data.

From this point, two slightly different approaches were pursued. Logan focused on creating and using transfer learning models to solve the problem presented, while Quinn focused on creating custom models to do the same.

3.2 Transfer Learning

For transfer learning two established models were tested, these were the VGG16 model and the MobileNetV2 model. A genetic algorithm was used to help tune the hyper-parameters for both of these models. Each individual was represented by a list of 6 integers that range from [0,9]. We will refer to this as the model's chromosome. The first number was the optimizer, this was modded by 2 to limit the choices between *adam* and *sgd*. The second value was the activation function. This was modded by 3 to limit the choices to *relu*, *tanh*, and *sigmoid*. The third value was the number of layers that would be trainable on the transfer learning model. A value of 0 meant that no layers in the transfer model would be trainable, a value of 9 meant that the last 9 layers would be trainable. The fourth value is the dropout percentage for the dense layers at the end of the model. This value would be divided by 13, so a value of 0 would result in every dense layer having a 0% dropout chance, and a value of 9 would result in each dense layer having a 69% dropout chance. The fifth value is the number of dense layers at the end of the model. A value of 0 means that there is only a flatten layer that goes directly into the output layer, while a value of 9 means that there will be 9 dense layers between the flatten layer and the output layer. Note that these layers are ordered by decreasing powers of two, with the last hidden layer always having 4 neurons. So a model with two hidden layers would have a single flatten layer, a hidden layer with 8 neurons, a dropout layer, another hidden layer with 4 neurons, a dropout layer, and finally the output layer. The final value in the chromosome determines which base transfer learning model to use. This value is modded by 2 to limit the choice between VGG16 and MobileNetV2.

3.3 Genetic Algorithms with DEAP

The genetic algorithm was implemented using the DEAP Python library. This allowed us easy control over the many parameters that affected the genetic algorithm. The fitness function for our genetic algorithm returns the F1 score of a particular model. Said function was set up as follows: if the individual had been previously seen, the function would return the F1 score of that individual. These had been stored in a dictionary for later use. If the individual had not been seen before, the model defined by the chromosome would be created and trained with its respective F1 score being returned after saving that model's chromosome in the aforementioned dictionary. If the model had an F1 score over 0.7, the model structure and weights would be saved locally for later examination. Our individuals were "mated" using the two-point crossover method provided in DEAP. Mutation occurred by performing uniform integer mutation with a probability of 0.1 drawn between the upper and lower limits of the values contained in our chromosomes. Lastly, for parent selection, we originally used the ranked selection method. However, this was swapped out for K-tournament selection with a tournament size of 3. Training was performed via the genetic algorithm with a varying population size for 10-30 generations. This function was run 3 times, resulting in over 30 hours of training time. The best model was based on VGG16 with an F1 score of 0.82667.

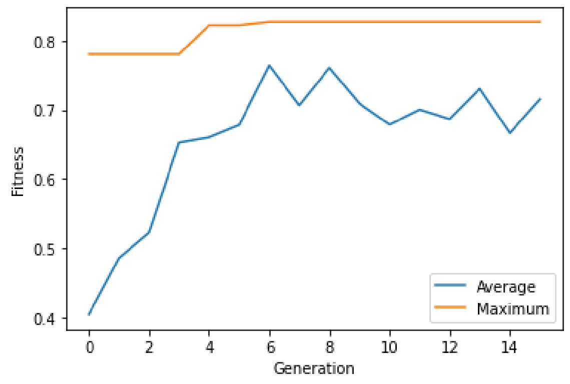

Figure 2: The average and maximum values of the genetic algorithm on transfer learning models over 15 generations.

3.4 Custom CNN Models

In addition to creating deep learning models using transfer learning we also defined several custom models. These models incorporated no inherited features and were built from the ground up. In total 3 custom models were defined, built, and tested. Certain strategies were tested in each model. For example, custom CNN type 1 had 3 blocks of convolution paired with max-pooling. CNN type 2 added batch normalization layers with dropout layers and CNN type 3 used stacked convolution layers with a global average pooling layer. As in the transfer learning models, the hyper-parameters for these models were defined in chromosomes. However, the chromosomes for our custom CNN models differed slightly, losing the parameter defining trainable layers. An example chromosome follows: *[4, 3, 5, 1, 1].* The first parameter defines the model type. This being the type of custom CNN to be created. This was modded by 3 to produce 3 possible values. In this example, CNN type 2 is chosen. The second parameter defines the activation type in the same manner. In this case, the activation function chosen is relu. The third parameter defines the optimizer. This value is modded by 2. In said example, the optimizer chosen is sgd. The last two values define the number of hidden layers and the dropout chance. In the above chromosome, they are translated as 3, and 0.07692 in our code. As before, the chromosomes were used to allow the genetic algorithm to perform hyper-parameter tuning on the created models. The strategies used in the genetic algorithm and the evaluation function are virtually identical to the transfer learning ones, with only minimal cases of disparity. One being, that all models (regardless of F1 score) were saved locally.

The training for our custom CNN models was performed with varying population sizes ranging from 10-30 trained for 10-20 generations. In all, over 15 hours of training was performed. The best model created was of CNN type 1 and had an F1 score of 0.8212.

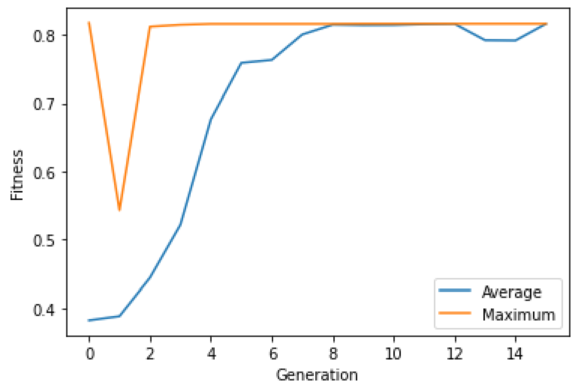

Figure 3: The average and maximum values of the genetic algorithm on custom CNN models over 15 generations.

4 Experimental Evaluation

In this section, you will find information on the methodology used, and the results produced.

4.1 Methodology

The data used to train our models was custom-tailored to our problem. It was manually created by playing the game multiple times with multiple people. This data was then collated into a single dataset.

The model training took place on Google Colab. Both Quinn & Logan followed the same preprocessing techniques and used the same strategies in their genetic algorithms. Once the models were trained, the top model in each F1 bracket was tested in Panda Runner. These models played the game at a simulated 60 frames per second. This was done to put each model on the same playing field, as the time required to process a frame and produce a prediction varied across models.

To compare our models we used two techniques. During training, the F1 score was the main metric and served as the evaluation parameter for our genetic algorithms. Outside of training and while playing the game, the weighted sum of the distance traveled and the number of coins collected was used. In addition to this, the raw number of coins collected was also examined.

Both custom CNN models and models created from transfer learning were created and compared. Their performance during training and while playing the game was examined.

4.2 Results

The performance between our models using transfer learning and our custom CNN models was very close. Below we will examine the confusion matrices, as well as the model makeup for each of these models.

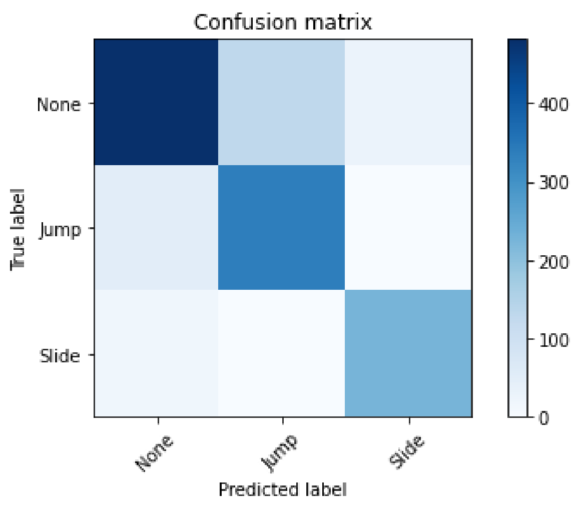

Figure 4: The confusion matrix for the best custom CNN model produced

Above we find the confusion matrix of the best custom CNN model that we trained. This model had an overall F1 score of 0.8212. As you can see, this model was very good at predicting when no action or when sliding was required. Both of these were predicted with an accuracy of 88%. It was slightly worse at determining when to jump. It predicted this accurately only 72% of the time. This model was composed of 3 convolutional blocks, followed by a flattening layer with 3 hidden fully connected layers (not including the output layer). It used the *relu* activation function for all layers and was compiled with the *sgd* optimizer. Its dropout layers had a chance of 0.077.

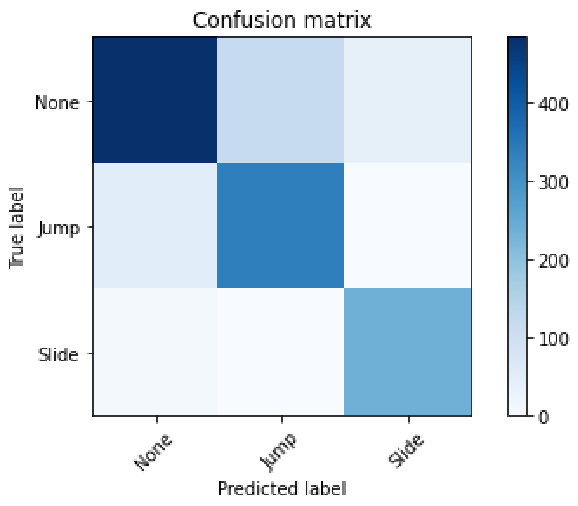

Figure 5: The confusion matrix for the best model created with transfer learning

As can be seen, the model that was created with transfer learning has an almost identical confusion matrix. This is due to the very similar F1 score of 0.82667. Moreover, the model had very similar accuracy rates for each label. It predicted none with 89% accuracy and jump and slide with 74% and 86% accuracy respectively. This particular model used VGG16 as its base and incorporated the *tanh* activation function with the *sgd* optimizer. It had 7 fully connected layers with 7 layers of the transferred model being unfrozen.

In light of these results, it is hard to come to a conclusion on which model type proved superior. Both models have their disadvantages and advantages. Both scores were only achieved after producing several generations of less superior models. We believe the reason that transfer learning did not have a noticeable effect on model performance was due to the highly specialized nature of our dataset. Unfortunately, due to this, it is almost impossible to find a pre-trained model that could easily transfer over to our dataset. This idea is reinforced with the number of layers that were left unfrozen by our genetic algorithm. This tells us that the best models using transfer learning were essentially retraining their networks for our data. It would seem that the viability of transfer learning falls short when your data is vastly different from the data that the model was trained on.

Another item that interested us was our inability to create models with a higher F1 score. After much thought and discussion, we came to the conclusion that this was due to how we collected our data. Since the way a game is played differs from human to human it is almost impossible to obtain extremely high accuracies. For example, when playing a game, one human player may decide to prioritize coins at the risk of hitting an obstacle, while another may decide not to do that. In addition, one player may decide to jump even when there is no apparent reason to do so, while another may keep their feet firmly planted on the ground. As a result of these facts, the dataset produced was not balanced and had conflicting actions.

After training our models we plugged the top model from each F1 bracket into our game.

Chromosome	F1 Score	Game Score	Coins Collected
[6, 3, 5, 1, 1]	0.8212	993	52
[3, 3, 5, 1, 1]	0.7986	993	51
[3, 3, 2, 2, 5]	0.6284	17	0
[5, 1, 3, 2, 4]	0.5496	17	0
[1, 6, 4, 3, 3]	0.4318	17	0

Table 1: Lists the performance of top custom CNN models in each F1 score bracket.

In this data, we can see that the models in the 0.8 and 0.7 F1 brackets successfully completed the game. This is shown by a score of greater than 900. These models also did a very good job collecting a great number of coins. However, our models in the lower F1 brackets failed to even get past the first obstacle in the game. This is due to how they always predict that no action should be taken. Interestingly enough, in our dataset, the number of images with that label outnumbers the other labels. So a frequent occurrence in our training would be a model deciding that always predicting no action would achieve a better score.

Chromosome	F1 Score	Game Score	Coins Collected
[7, 1, 7, 1, 5, 7]	0.8267	993	50
[1, 3, 3, 1, 5, 1]	0.7992	993	49
[2, 0, 3, 0, 2, 4]	0.7294	463	16

Table 2: Lists the performance of top transfer learning models in each F1 score bracket. Also includes the worst model created.

In a similar fashion, the top models that incorporated transfer learning also managed to finish the game. However, they seemed slightly less motivated to collect coins. This can be seen in the small disparity between these values. Note that transfer learning models with an F1 score lower than 0.7 were discarded.

These results show that it is possible for a computer vision network to emulate human behavior by predicting the correct action to take solely on the image alone. In addition, once trained on a level, these networks almost never fail to complete that level as they take the exact same action on the same image frame. This shows a consistency that human players lack.

While our goal was to create a model capable of completing a single level, in the interest of experimentation we decided to unleash our trained models on a game level they had never seen before. The top model for each implementation strategy is listed in the below table. Note, the top model is the custom CNN model with the bottom being the model that used transfer learning.

Chromosome	F1 Score	Game Score	Coins Collected
[3, 1, 3, 1, 4]	0.8189	781	44
[5, 0, 2, 1, 5, 5]	0.8197	627	35

Table 3: List the top-performing models on a new level never trained

As expected our models failed to complete the level. Once again a complete run was denoted by a score greater than 900. In the end, the models fell prey to never-before-seen obstacle combinations. Nevertheless, we were surprised by how well they actually did. This means that some of the models trained were capable of determining a solution for untrained situations. This showed some level of generalization.

Of note, is the fact that our top models on the first level, we're not our top models on the second level. This shows that it was common for models with a higher F1 score to fail to generalize on data they had never seen.

5 Related Work

The use of artificial intelligence to play video games has been studied for many years and countless articles have been published on this subject. This section lists and discusses a few articles on this topic and how they differ from the work in this paper.

One paper that addresses a problem in a similar field is *Game-Independent AI Agents for Playing Atari 2600 Console Games* by Yavar Naddaf [1]. This project focused on creating AI agents that would be trained on 4 different Atari 2600 games [1]. It would then attempt to play 50 different randomly selected games using the knowledge it had gained from its previous training [1]. This project used both reinforcement learning with gradient descent and search-based methods [1]. This project is significantly different from Panda Trainer. First of all, Panda Trainer focuses on the depth of skill rather than breadth. The goal with Panda Trainer was to have the agent be extremely skilled at one particular complex task rather than having a more general understanding of many tasks. Panda Trainer also implements a genetic algorithm to help fine-tune the hyperparameters. This method gave a significant advantage by allowing an algorithm to evolve the hyperparameters over a series of generations. By allowing a model's hyper-parameters to change over time, you avoid potential bias. Plus, it allows for many more pattern combinations to be explored without having to perform an exhaustive search.

Another paper discussing a similar topic is *Generating War Game Strategies using a Genetic Algorithm* by Timothy E. Revello and Robert McCartney [2]. This paper discussed the use of genetic algorithms to find optimal solutions for war games that contain uncertainty [2]. In this paper, a genetic algorithm was allowed to optimize 12 groups of ships that were pitted against an opposing force on a simple grid [2]. This model used a genetic algorithm to optimize for a problem, which is similar to what was done in Panda Trainer. However, Panda Trainer uses computer vision to pass information to the model. One advantage of this strategy is the simplicity of the data being passed into the model. It is significantly easier to send a single image rather than numerous different data points.

These are just two of several different papers that have been written on the topic of machine learning for playing games. However, many of these projects differ in significant ways from Panda Trainer. Panda Trainer's use of computer vision and genetic algorithms, with both custom and transfer learning models, makes it stand out from previous lines of research.

6 Conclusion

In this project, we created and trained both custom CNN, and transfer learning models. These models were trained on a custom dataset taken from Panda Runner. Due to the uniqueness of our dataset, the transfer learning models showed no advantage over our custom models.

Once trained, both types of models were able to successfully complete the game level they were trained on. Though the custom CNN models proved a bit better at collecting coins, one of the goals of the game. Only models in the upper 0.7 or in the 0.8 F1 brackets were able to complete the game level.

When unleashed on a level never trained, the previous best models showed shortcomings, often failing to generalize well. Though, many of our slightly lower scoring models progressed further into the new level showing some ability to identify and respond to unknown situations.

Overall, we accomplished our goal of creating a computer vision model capable of emulating human behavior in Panda Runner. This model showed an ability to master a single complex task while also showing a low level of generalization on tasks it was not trained to complete.

7 Work Division

Both team members worked extensively on this project. Both of them worked on coding and debugging the game with Quinn taking the lead on this part. Once the game was created both of them compiled the dataset by playing the game and having the program take screenshots. Quinn wrote the code to preprocess all the data on the Google Colab page. Logan wrote, trained, and tested all the transfer learning models. Quinn wrote, trained, and tested all the custom models. For the report, Logan wrote the

Introduction, System/Algorithm Design, Related Work, Work Division, and Learning Experience sections. Quinn wrote the Problem Formulation, Experimental Evaluation, and the Conclusion sections. Quinn also compiled the final notebook. Overall the work was split evenly with both people participating in all the major steps of this project.

8 Learning Experience

Both members were able to learn many different concepts and valuable lessons during this project. First, both members gained a better understanding and expanded their experience with game design and development. While this was not a goal of the project, it proved to be a valuable experience. Quinn especially was able to further develop his skills with game design This project also expanded the members' understanding and familiarity with genetic algorithms. Both team members became quite familiar with the DEAP library in Python and finished the project feeling comfortable creating and deploying genetic algorithms.

Another item we learned was how datasets with conflicting data make training models difficult and hard to measure. As the data was taken from two different players who played the game differently, there were many cases where there were several possible moves that would be correct. Since each player made different moves, there were times where there was conflicting data. This caused the models to have a lower F1 score when training because there was no way for the model to account for these differences. That being said, it does not seem to have highly affected the actual performance of the model when playing the game. Thus the members learned that having consistent data to feed their models simplifies the training process and results in a higher score.

References

[1] Yavar Naddaf. 2010. Game-Independent AI Agents for Playing Atari 2600 Console Games. Master's thesis. University of Alberta, Edmonton, Alberta. https://doi.org/10.7939/R3134Q

[2] T. E. Revello and R. McCartney, "Generating war game strategies using a genetic algorithm," Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600), 2002, pp. 1086-1091 vol.2, doi: 10.1109/CEC.2002.1004394