CS230

Using Fully Connected Networks to Create Penguin Tag AI

Reynaldo Cabansag*
Computer Science Major
Stanford University Class of 2020
cabansag@stanford.edu
Code Repository: https://github.com/jrcabansag/PenguinTagNeuralNetwork

Abstract

This project aimed to use deep learning to create videogame AI for a game called Penguin Tag. Using recorded gameplay, we utilized fully connected networks to create an enemy that emulated our playing style, with the best model achieving 42.3% accuracy when predicting player actions. This was pretty successful given that a player's gameplay is not always consistent or logical in fast-paced games. We were able to play against the AI to evaluate it qualitatively, and it served as a fairly tough opponent, as it learned strategies like proper aiming technique, target prioritization, and diving the opponent.

1 Introduction

AI systems in the videogame industry are currently created through decision trees or reinforcement learning techniques [2]. While a growing field in AI, deep learning, has revolutionized fields such as computer vision, NLP, and speech, it has yet to take root in the videogame industry [1]. For this project, we wanted to explore the capabilities of deep learning when it comes to creating videogame AI. Specifically, we wanted to see if it could create an AI that plays like a human player, by feeding it the gameplay of a specific player. This issue is important, as currently many techniques - like reinforcement learning - reward the AI for wins. Though winning is a generally good metric to strive for, the AI could learn winning strategies/techniques that are deemed unfair by human opponents. This project can introduce a new way to create AI for two-player games that requires less hand-tuning than current industry methods, but can also be controlled to fit a particular gamestyle to allow for more fun gameplay against the AI.

2 Related work

This project was heavily inspired by deep learning research conducted in 2017 by Northwestern students, which aimed to create AI for a fighting game called "Rumblah". In their research, they captured the game state to be used as feature vector and attached the player input as the label. One notable detail is that they took a picture of the game and used that as the feature vector . They then ran the image through a convolutional network, to receive outputs with 60% accuracy and 80% top-2 class accuracy [3]. Since we had access to the game code of Penguin Tag, we chose instead to use raw game data. This has an advantage in minimizing the size of the input - we simply use a (50,1) shaped vector of game data, instead of using a (56,56) shaped vector. Using game data also removes

*To see more projects and games, visit jrcabansag.com

CS230: Deep Learning, Winter 2018, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

the requirement that the neural network has to extrapolate game state features from an image of the game, which makes training it much easier.

3 Penguin Tag

Figure 1. The Penguin Tag starting screen.

Penguin Tag is a two-player game where players play as blue and pink penguins facing off against each other. Players must avoid getting tagged by the enemy penguins, which are the grey penguins wearing a beanie opposite of their color. Players can shoot snowballs at these enemies to change them to their team, and can also shoot snowballs at the enemy player to have a chance of slowing them down. The only inputs needed from the player are the 4 keys for movement (W,A,S,D, or arrow keys), as well as the G key or ALT key to shoot a snowball. The game can be played at: http://jrcabansag.com/penguintag/

4 Dataset and Features

The training data used was two hours of recorded gameplay with two people playing against each other, while test data was 10 consecutive minutes of gameplay. 10 times a second, the game would record the game state as a feature vector, and also record a particular player's key input at the time, to be used as the correct label for that game state. The reasoning for this was to create an AI that given a game state, would perform the same key input as that player. We recorded only one person's key inputs during the entire data collection process, so that the network would only have to worry about matching that person's gameplay.

Each feature vector contained 50 features total: each of the player penguins' x and y coordinates, speed, direction, and life counts, each of enemy penguins' x and y coordinates, directions and team, as well as up to eight snowballs' x and y coordinates and directions. Each label was a number from 0 to 5, to represent each of the 6 different key inputs (no key, up key, right key, down key, left key, and snowball key), and was later converted to a one hot vector during training.

To increase the amount of data acquired, we performed data augmentation on the training set by flipping the game states horizontally, vertically, and both horizontally and vertically along with their correct label. This quadrupled the amount of data collected, and also trains the network to perform the same actions in mirrored game states.

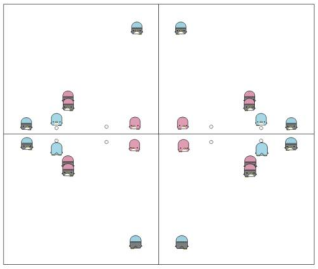

Figure 2. Data augmentation. In each frame, the blue player is running away from the enemy penguins wearing a pink hat. In the meantime, the pink penguin is chasing towards them and throwing a snowball at them.

Mirroring the game state horizontally, vertically, and both horizontally and vertically still preserves the game state and strategies of both penguins, so long as their moves are flipped as well.

After data augmentation, we had a total of 275,000 training examples, as well as 6,500 test examples, which were not augmented.

Figure 3. Graphs visualizing the label distributions throughout the training and test sets. As we can see, the "No Key" label is much more frequent than the other labels, and this imbalanced is addressed when the models are created.

5 Methods/Experiments

All of the models used in this project were fully connected networks which used RELU activations in each hidden layer, and used a softmax activation for the output layer, which had six neurons to map to the six possible key input possibilities. Each model was trained using the ADAM optimization algorithm, which combines techniques like momentum and RMSprop in order to control the descent of the gradient and speed up training. Models had variations in their numbers of hidden units, numbers of hidden layers, and loss functions, as well as variations in the data that they were trained on.

In this project, "shallow" networks had 3 hidden layers, while "deep" networks had 7 hidden layers. All of the shallow networks had 50 neurons in each hidden layer and the "fat" deep network also had 50 neurons in each hidden layer. The "skinny" deep network had 10 neurons in each hidden layer, and also performed batch norm for each layer.

There was a significant label imbalance in the training and test set - as shown in Figure 3, the frequency of the no key label was more than the frequency of all the other labels combined, for both the training data and the test data. We anticipated that this would be problematic, as the models would be incentivized to output no key due to its much higher label frequency. Thus, we experimented with exposing models to different data, and created three categories: 1) All Data: Trained on all of the data. 2) Actionable Data: Trained only on data where the user pressed a key (up, right, down, left, or snowball labels). 3) Balanced Data: Trained on all the data, with the loss function normalized by class frequency.

Models which trained on all the data as well as the models that trained only on the actionable data used the standard softmax cross entropy loss function:

$$\mathcal{L}(\hat{y}, y) = -\sum_{c}(y_c log(\hat{y_c}) + (1 - y_c) log(1 - \hat{y_c}))$$

However, models marked as training on "balanced data" were different in that they used a modified softmax cross entropy loss function divided by the correct label's frequency in the training data. This was to prevent label frequency from having an impact on the preferred outputs of the model:

$$\mathcal{L}(\hat{y}, y) = -\sum_{c} \frac{y_c log(\hat{y_c}) + (1-y_c) log(1-\hat{y_c})}{count(c)}$$

At the start of training, the models were initially trained with a learning rate of .0005, which was gradually decayed to as low as 0.0000001 whenever the cost of the network showed signs of diverging. All of the models used batch gradient descent, since running through the entire batch was very quick, and because none of the models seemed to have issues getting stuck in saddle points.

6 Results/Discussion

Training Data Accuracies

Model/Trained Data	All Data	No Key	Up	Right	Down	Left	Snowball
Shallow/All	.577	.999	.000	.001	.000	.000	.000
Shallow/Actionable	.189	.000	.174	.423	.164	.419	.705
Shallow/Balanced	.149	.012	.343	.373	.317	.385	.283
Skinny Deep/Balanced	.203	.020	.196	.176	.179	.217	.333
Fat Deep/Balanced	.192	.034	.454	.391	.500	.396	.375

Test Data Accuracies

Model/Trained Data	All Data	No Key	Up	Right	Down	Left	Snowball
Shallow/All	.606	.999	.000	.004	.009	.000	.000
Shallow/Actionable	.175	.000	.150	.486	.219	.313	.743
Shallow/Balanced	.131	.007	.356	.403	.324	.319	.267
Skinny Deep/Balanced	.149	.030	.167	.182	.219	.187	.324
Fat Deep/Balanced	.161	.038	.467	.405	.407	.260	.303

Accuracy was determined by comparing if the max probability label from the softmax output was the same as the true label. Based off of total accuracy, none of the shallow models seem to be overfitting, whereas the deep models are likely slightly overfitting, shown by the drop in total accuracy.

Furthermore, if total data accuracy were the only metric to evaluate the models, the shallow model trained on all of the data would be deemed the best model. However, when analyzing this model's accuracy on each label for the training and test set, we see that it has near perfect accuracy for the no key label, and practically zero accuracy for the other labels. This is likely due to the high frequency of the no key label in the training set, the model was just trained to output no key as the highest possibility every time, and the model failed to learn what game states prompt for actions like moving up, down, etc.

When we removed all of the no key data in the shallow network trained on actionable data, we saw that the other labels finally had accuracies greater than zero (meaning the network was capable of learning when to do each move). However, each of the labels have different accuracies. One conclusion we could draw is that the lower accuracy labels (like up and down) are harder for the network to figure out compared to labels like left, right, and snowball, which had much higher accuracies. However, it's also important to note that the probabilities of the labels roughly correspond to their frequency in the training data. For example, the snowball label had the second highest frequency in the training data, and so when the no key labels were removed, it was likely that the network wanted to output snowball more to accommodate its higher frequency.

This hypothesis was confirmed when analyzing the models trained on balanced data, which rearranged the loss function based on each class' frequency in the training data, thus preventing it from prioritizing any particular label based on its frequency. We see that the balanced models had roughly equal accuracies for all of the actionable label, and very low accuracies for the no key labels, which shows that the network couldn't find a correlation between game state and pressing no-key. This is fairly understandable, as there are many game scenarios where most people would agree which key to press (to run or shoot), but there are few game scenarios where people would agree to not press a key.

Since actionable label accuracies were a better way to show what the model learned, we used a new quantitative metric - the accuracy of the model on the actionable data only (up, right, down, left, and snowball labels).

Actionable Data Accuracies

Model/Trained Data	Training Data	Test Data
Shallow/All	.000	.003
Shallow/Actionable	.377	.382
Shallow/Balanced	.340	.333
Skinny Deep/Balanced	.222	.216
Fat Deep/Balanced	.423	.369

According to this metric, the fat deep network trained on balanced data learned the most with an actionable accuracy of 43.3%, and the shallow network trained on the actionable data learned the second most with an actionable accuracy of 37.7%. On paper, these accuracies may sound fairly low, however, they are reasonable considering a player might not always press the same key in a fast paced game, and also because there are several scenarios where pressing one key is just as logical as pressing another key.

In fact, when we deployed the models and played against them, we found that these two models were pretty difficult opponents, as they learned behaviors like proper aiming technique, target prioritization, and diving towards the opponent, all techniques used by the player they aimed to emulate. In addition, the difficulty of the opponent highly correlated to their actionable data accuracy, further validating that it is a good quantitative metric to evaluate the models. Using this metric, we see that the skinny deep network is deemed the second worst, and when we played against this model, we found that its probabilities were fairly fixed regardless of the different game scenarios. Since the other deep network and other fatter networks were able to learn very well, it's likely that the 10 neurons for each layer in the skinny deep network were not sufficient to extrapolate game scenarios. This makes sense, especially since the first hidden layer had 10 neurons, and trying to squeeze all useful info from the 50 features into 10 neurons might have been impossible.

When deploying all of the models, we found that it was extremely helpful to make the enemy sample from the top X softmax labels instead of simply picking the label with the highest probability. For the models with low actionable data accuracy, they usually got stuck in walls and corners (since there was little data where the players were stuck there), or did only one type of move. Sampling from the top three or four softmax labels allowed for enough noise to prevent those scenarios, and also allowed for more natural-looking gameplay that showed off the probabilities that they learned. On the other hand, the models which had higher actionable data accuracies could sample from the top two moves or not even sample at all (and just use the max move) and still look natural and be a difficult opponent, showing off how much they learned.

Though the top two models were difficult opponents, they were not perfect. For the shallow network trained on actionable data, it sometimes had a hard time regarding an enemy above it as a threat, and often ignored it until it got tagged. This is likely due to the low accuracy the model had for up/down key labels, which was a result of the relatively low frequencies of the up and down key labels in the training data. Furthermore, the best model (the deep fat network trained on balanced data) would confidently charge into enemies with snowballs, but would do so even with clusters of enemies, so while it was able to defeat some, it would eventually run into the remaining enemies in the cluster that weren't hit. It's likely that adding more data would help to perfect both of these models.

It's highly recommended that you play against the models, both to have some fun, as well as to evaluate the different models qualitatively. You can do so at: jrcabansag.com/PenguinTagAI

7 Conclusion/Future Work

In this project, we were able to successfully use fully connected networks in order to create AI that played logically, and emulated the target player's playstyles/techniques. In the future, we aim to collect more training data to see whether it would allow for better actionable accuracies in each of the models, and also see whether it would allow the best models to avoid the scenarios in which they constantly lose lives. We'd also like to try different network architectures, like ResNets, to see whether they would be able to learn how to distinguish game scenarios in which the AI should press no key vs other keys, which fully connect networks struggled on.

Though we believe deep learning is not going to completely replace techniques like decision trees and reinforcement learning for creating videogame AI, we hope that this project showed that deep networks are definitely capable when it comes to creating videogame AI, especially if programmers want to emulate a certain gamestyle in their AI but don't know how to code it. Deep learning can serve as one more tool in the toolbox for game developers, allowing them to create AI that make their game even more engaging and fun.

References

[1] "Epic's Tim Sweeney: Deep Learning A.I. Will Open New Frontiers in Game Design", *Medium*, 2017. [Online]. Available: https://medium.com/@Synced/epics-tim-sweeney-deep-learning-a-i-will-open-new-frontiers-in-game-design-5682ad32454c.

[2] D. Kehoe, "Designing Artificial Intelligence for Games (Part 1)", *Intel*, 2015. [Online]. Available: https://software.intel.com/en-us/articles/designing-artificial-intelligence-for-games-part-1.

[3] E. Chan, "Learning to Fight: Deep Learning Applied to Video Games", *Northwestern MSiA*, 2017. [Online]. Available: http://sites.northwestern.edu/msia/2017/09/19/learning-to-fight-deep-learning-applied-to-video-games/.

[4] M. Abadi, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems", *TensorFlow*, 2015. [Online]. Available: https://www.tensorflow.org/.

[5] D. Smilkov, N. Thorat and C. Nicholson, "deeplearn.js", *Deeplearnjs.org*, 2018. [Online]. Available: https://deeplearnjs.org/.

[6] "RequireJS", *RequireJS.org*, 2018. [Online]. Available: http://requirejs.org/.