Using Deep Learning to Play Temple Run

github.com/dungwoong

Abstract

In this project, I will use deep learning to play the popular game Temple Run 2, created by Imangi studios.

I used the Shufflenet_v2 architecture and did transfer learning on a pre-trained model to classify screenshots of the game into seven action categories. My AI will then perform the resulting action in my Android game emulator, Bluestacks, creating an autonomous player.

I found that transfer learning is very useful for training a network on limited data. Also, the shufflenet_v2 architecture is fast enough to make real-time predictions on GPU. While training, I encountered many problems such as overfitting and generalizability issues and made decisions to deal with them. The main limitations of the AI is the generalizability of the collected data to in-game situations, the static nature of the dataset, and imbalanced classes within the dataset.

Introduction

Temple run is an "infinite runner" game in which players must navigate a road of obstacles, taking the correct action at each obstacle in order to survive. There are two objectives of the game: survival time, and a score that is dependent on collected coins and items during gameplay. This investigation will emphasize survival time over score.

Just like the game, there are many obstacles that must be navigated by a researcher aiming to create an effective AI for this game. Since the game is played in real-time, I took both accuracy and prediction speed into account. I also considered the best way to create a generalizable dataset, especially since the game changes speed over time and the model can only be fed static snapshots of the game. I believe this research topic is a great way to apply general deep learning knowledge.

I was initially inspired to pursue this project when I stumbled across a repository for an AI that was able to play racing games, predicting the x-coordinate of the computer cursor(which controls steering) based on in-game screenshots. It inspired me to use convolutional neural networks to handle a problem that would normally prompt the use of reinforcement learning algorithms.

Currently there is only one other project tackling this problem, by Arun Kavishwar and Hunter Karas, two students at Brown University. It has a similar methodology to my idea, using convolutional neural networks to mimic the actions of a real player. However, the team encountered issues reducing inference time, resulting in inconsistencies in their predictions when running the model in real time.

Methods

Basic methodology

The main idea for my project is to train a convolutional neural network to classify screenshots of the game into 7 categories: neutral, up(jump), down(duck), left turn, right turn, left lean and right lean(A and D keys). I planned to run the model alongside my game, having it make real-time predictions and execute the corresponding actions.

Speed and accuracy

For the model, I needed a lightweight model that could make fast predictions, with an architecture complex enough to capture the intricacies of the data. After reading the literature, I found the research paper “Shufflenet V2: Practical Guidelines for Efficient CNN Architecture Design” by Ma et al. The authors prioritize direct metrics such as network speed over indirect metrics such as the number of floating point operations, or FLOPs, proposing a novel architecture that is fast and performs well on CNN benchmarks. Thus, I decided to use the Shufflenet V2 architecture for my project.

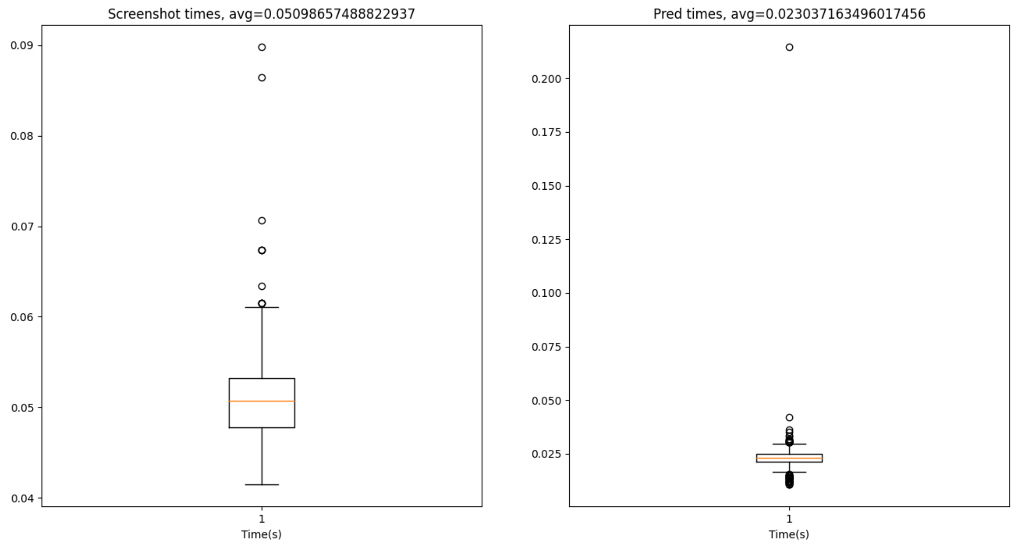


Figure 1: The average time to take a screenshot is 0.05 seconds, and the average prediction time of a ShuffleNet V2 x1.0 model is 0.02 seconds, suggesting I can make real-time predictions.

Data Collection

I attempted multiple data collection methods, settling on two options.

The first option is to collect real-time data, saving every frame of my gameplay and labeling it based on the action I performed in the game. This creates a lot of related data, and may affect the model’s ability to classify obstacles. For example, a player will usually tap the up arrow to jump. Thus, only a few frames will be labeled “jump,” with the rest labeled as “neutral.” However, most of the frames will look very visually similar, making it difficult for the convolutional neural network to tell the images apart.

The second option is to collect data based on “triggers.” Every time I press a key while playing the game, my data collection program would save the corresponding screenshot to a labeled folder. When I lean, I would record the frame when I initially press the A or D key to lean, and proceed to record a frame each second that I continued holding down the lean button. I would also take some manual screenshots of the “neutral” player state.
This makes it easier for the model to clearly differentiate between different classes, and to not be overwhelmed by related data points. In addition, I have more control over the points that are collected, allowing me to fine-tune my dataset to be more generalizable to real in-game situations.

I decided to use the latter method to train my models.

Thus, I would collect data by playing the game while the program captured my actions, and then train my model.

Main problems in model-building

First I will describe some problems I encountered that had definitive solutions that improved performance.

Imbalanced action classes

The player performs certain actions much more than others. For example, there are more obstacles to jump over, than to duck under. However, this is representative of the actual game, and I want the model to learn the different frequencies of each class.

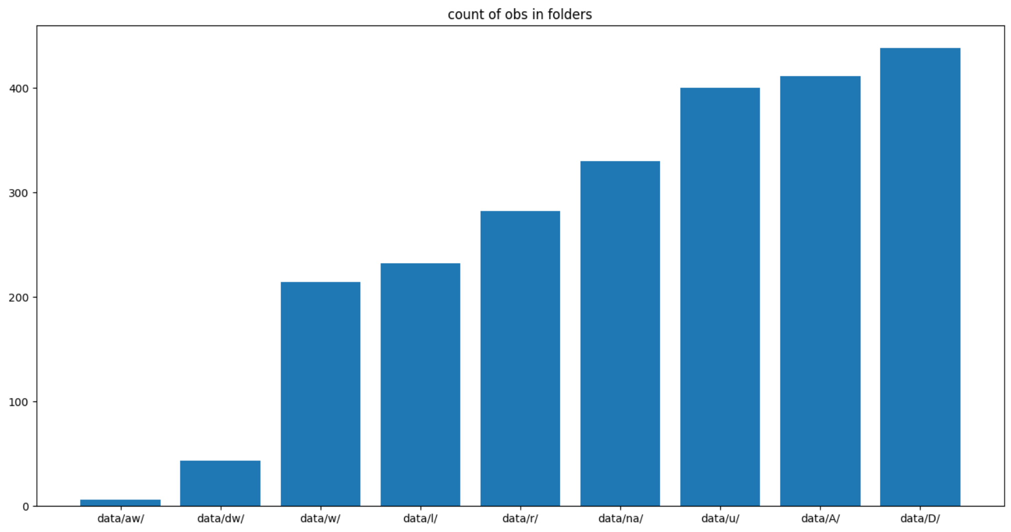


Figure 2: Count of observations in folders. There are imbalanced classes. Ignore the leftmost two folders.

I tried using weighted sampling to even the classes during training, but the model performed worse on average, compared to models trained on the unbalanced classes.

Imbalanced obstacle classes

Within each action(eg. jump) there are many different obstacles where this action applies(eg. Walls, bridges, rivers/waterfalls). However, these obstacles do not have an even probability of appearing during gameplay.

In this case, imbalanced obstacle classes caused the model to be disproportionately worse at identifying uncommon obstacles. In-game, models would struggle to take action when faced with these specific obstacles.

My solution to this problem was to manually collect more data for these specific obstacles. I did this by both duplicating existing images, and by altering my data collection system to only collect data for these specific obstacles, in hopes of balancing the obstacle classes. I found that creating even obstacle classes tended to increase model performance.

Trade-offs in model building

I will now describe some problems in model building that involve trade-offs, where decisions must be made.

Variation in training data

The game speeds up as the player continues playing, causes jump/duck timings to change slightly over time. Since the model can only see one image at a time, it cannot gauge the speed of the game, and I must consider this when collecting data.

I tried to combat this by collecting a larger proportion of data from earlier in the game, compared to later in the game, by stopping my runs at 1000-2000 meters when collecting data.

However, when the model contains more early-game data, it will typically perform worse later on in the game. I found that models trained on general data would often fail early-game but would perform very well if they made it past 500-750 meters. However, models trained on more early-game data would perform better overall but never achieve any high scores(over 2000m).

I repeatedly collected more data based on my model's problems, until I created what I considered to be a "good" dataset with the correct ratio of early and late-game data. I wanted to create a model that could attain high scores of over 2000 meters, but would also consistently make it past the early game(0-750m)

Overfitting, underfitting and generalizability

Model overfitting was a problem, due to the complexity of the network and the small size of the training set (2000-3000 images).

However, I found that slight overfitting of the network was fine, as the training data is very representative of the actual in-game scenarios.

Since I measure overfitting by comparing model performance on the training set and validation set, which is an indirect metric of how it will perform in game, it does not directly correlate with the model's performance. I found that slight overfitting led to better models.
However, severe overfitting would lead to issues when generalizing to in-game situations.

Some methods I implemented, or tried implementing, to combat overfitting were:

- Augmenting the training dataset using random affine transformations
- L2 regularization
- Dropout

I also tried using a shufflenet 0.5x instead of a 1.0x, simplifying the architecture to reduce overfitting.

Model size

Using a shufflenet 0.5x instead of a 1.0x reduces the amount of overfitting at the cost of performance. I found that 1.0x models would typically be able to set high scores(>2000m), whereas 0.5x models would typically be better at generalizing.

I also tried freezing different amounts of layers when transfer learning to combat overfitting.

The final models

Model 1

I downloaded a pre-trained Shufflenet V2 model online, trained on the ImageNet classification task. I then used transfer learning to train on my images, freezing most of the layers.

After training the model on some images, the model was able to play the game, attaining maximum survival distances of over 1000 meters. This validated that my data collection method was appropriate. It also validated that my code for making and executing real-time predictions was correct

I collected more data and created a validation set. This model was able to attain 95% training accuracy, but only had a ~70% validation accuracy.

Model 3

This is the second model with significant changes, denoted as 'model 3' in my github repository. I collected more training data, and then implemented a few measures to combat overfitting.

- I reduced the model architecture down to a ShufflenetV2 0.5x.
- I applied a random affine transformation to the input images, augmenting the original dataset.
- I reduced the initial learning rate.
- After 15 epochs of training, I unfroze a few more layers of the Shufflenet, and continued training at a reduced learning rate, trying to make up for the smaller model architecture by increasing the number of trainable layers.

This seemed to alleviate some of the overfitting issues, and my model was able to attain around 80% validation accuracy.

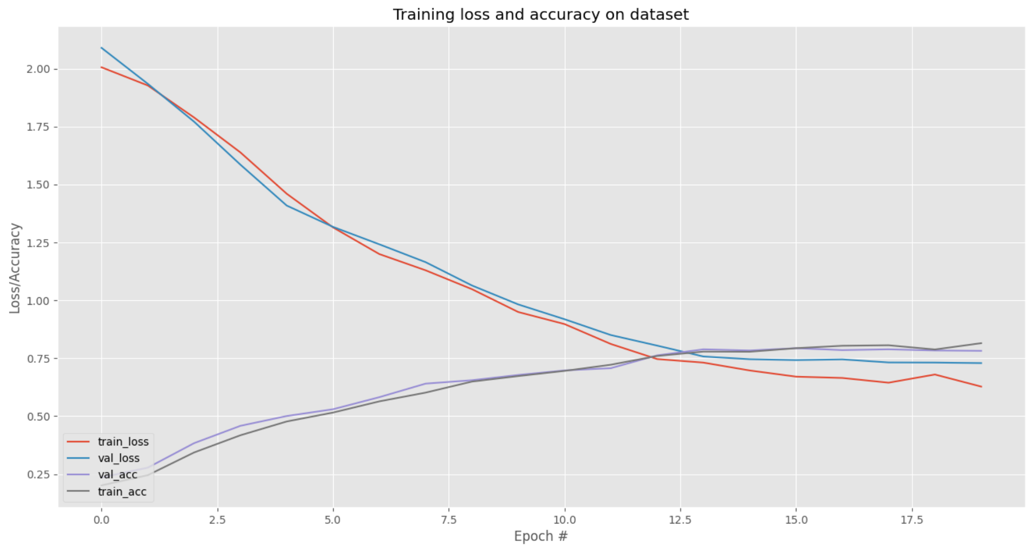


Figure 3: Model 3 shows less signs
of overfitting than the previous model

Model 4

I noticed that model 3 was repeatedly failing to jump over a few specific obstacles early in the game, either jumping too early or not jumping at all.

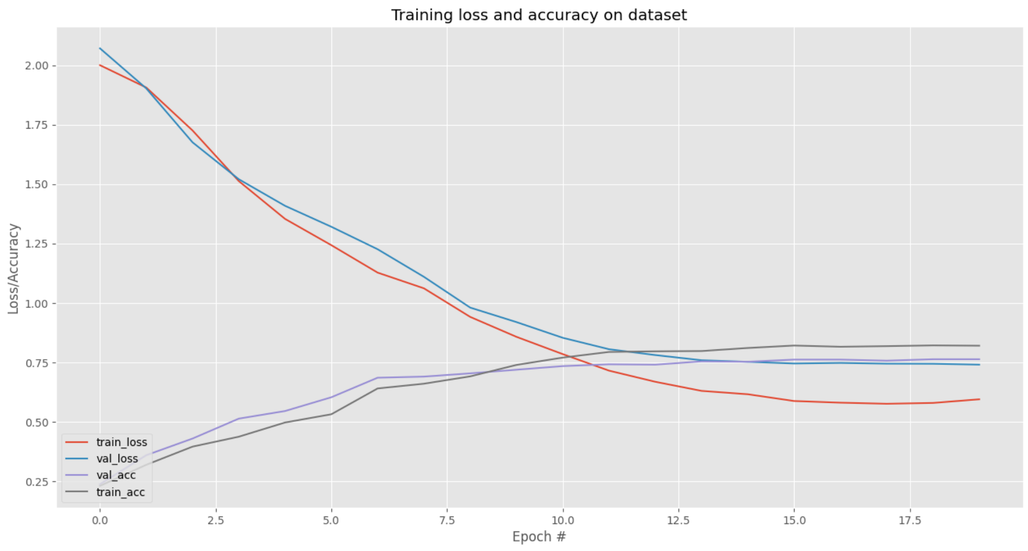
This was due to the imbalanced obstacle classes, and perhaps because the training data contained a lot more late-game data than early-game data. These uncommon obstacles were more likely to occur later on in the game(at faster speeds) than earlier in the game. Almost every time the previous models encountered one of these obstacles(suspension bridge with a gap, or a waterfall) early on, it would fail.

I began collecting more early-game data.

I also collected images of the player a bit before each obstacle, and labeled them as “neutral,” in hopes of the model learning to stop taking actions too early.

Since the model can perform frame-by-frame analysis of the game, I believed that this would not affect late-game outcomes too much.

I then retrained the model on this new dataset.



Model 4 shows more signs of overfitting than model 3.

I believed that this increase in variety to the images affected the loss function of the model, as each class is not as clearly separable from the other classes as before, compared to the older iterations. However, these images are more representative of the real game.

Model 5

Although model 4 performed better on average, it rarely set a score above 1500, and has never been observed to surpass 2000 meters in game.

I tried switching back to a shufflenet x1.0, training for fewer epochs with no additional unfreezing of layers. The goal of this is to combat overfitting in a more complex model.
I believed that since the dataset had increased in variety, a larger model was necessary to capture the differences between the classes accurately. However, I must still avoid overfitting.

Results

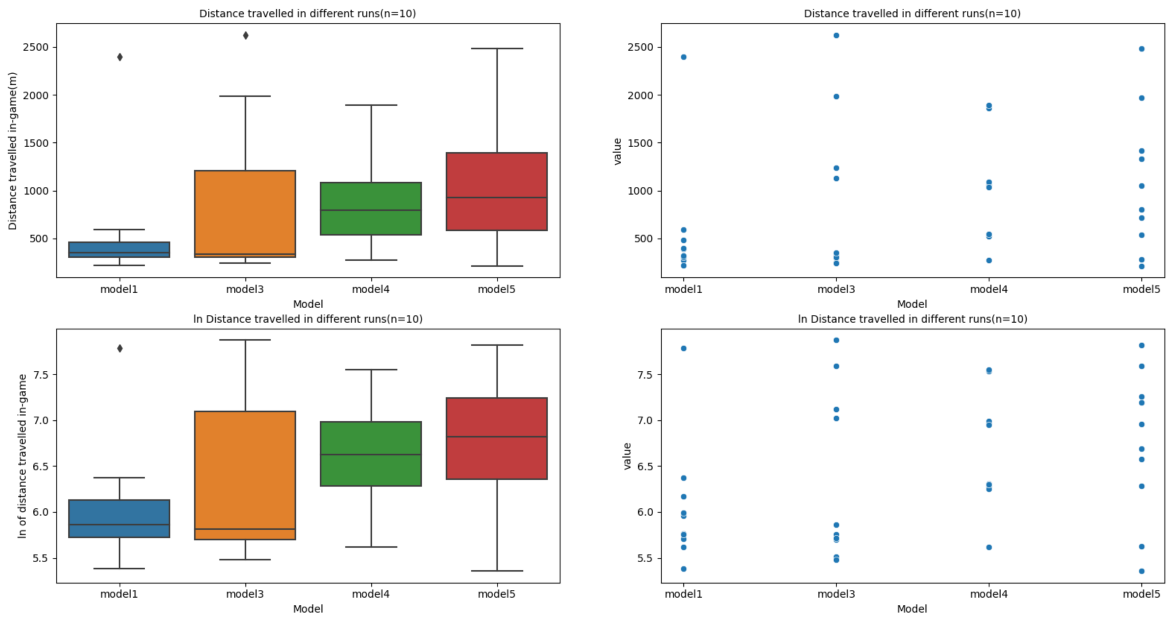


Figure 4: Distance traveled(meters) and
log distance traveled of each model over 10 trials

Log distance is a good measure of performance, as the speed of the game increases with time, so larger distances are covered faster as the game progresses.

These observations are based on the recorded runs, and non-recorded runs outside of the data shown above.

Model 1 typically performs poorly in game, although there are a few outliers in which the model achieves high scores.

Model 3 has more variation, but also typically performs poorly.

Model 4 has less variation and typically has longer runs than models 1 and 3. However, it has never been observed to achieve high scores comparable to that of models 1 and 3. From the log distance chart, it has better average performance, but worse extremes.

Model 5 typically performs better than the previous models, and is able to attain high scores as well.

Temple run 2 includes many different terrain. All models perform poorly on any terrain that is not the basic road terrain, such as on the water or minecart portions of the game.

Discussion

The main obstacles in training an effective model to play Temple Run 2 is the quality of the data, and the generalizability of the training data collected to in-game situations.

I also encountered obstacles normally encountered when training neural networks, such as overfitting, network complexity and different tradeoffs when deciding how to train the network.

With each model, I've outlined the improvements I made and it seems to be reflected in the model performances. However, each model was trained on a progressively larger dataset, which is an alternate explanation for the increase in performance. However, I believe that the dataset size is not large enough to explain the different performances of each model, as the number of observations only increased by ~300 between models.

Limitations and future corrections

Action classes are still imbalanced. In particular, there is a lower proportion of 'neutral' observations, where the player is supposed to do nothing, than the realistic in-game proportion. This is due to our data collection methods, and may explain why the models often make random actions when no obstacles are present.

One idea for improving on this limitation is to fine-tune the model further on real-time game data. I can also consider collecting more data in general.

The changing speed of the game is still an issue. I tried to find a good balance between slow-paced and fast-paced gameplay to optimize model performance across all stages of the game, but were still forced to make a tradeoff.

One improvement that can be made is to train the model on multiple past frames of the game, using 3D convolutions, or starting with deeper 2D convolutions, to allow the model to train on dynamic gameplay rather than static snapshots of the game. I avoided investigating this in my project as it would disallow me from using pre-trained Imagenet models, and would require more training data for models to reach an acceptable level of performance.

Conclusion

It's remarkable that I was able to create an AI agent that could surpass 2000 meters autonomously using less than 3000 training instances. This project taught me a lot about convolutional neural network architectures, and gave me practical experience solving a problem with CNNs.

I enjoyed that the problem included a real-world application for the model, teaching me to heavily consider the generalizability of the training data, and the method by which I collected it. This is something I often do not think about, as I usually only have a small testing dataset that I use to evaluate my models.

References

Kavishwar, Arun. "TempleFlow: The AI for Temple Run." *Medium*, 2 Sept. 2021, medium.com/@arun_kavishwar/templeflow-the-ai-for-temple-run-489d5328315c. Accessed 1 Dec. 2022.

Ma, Ningning, et al. "ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design." *ArXiv:1807.11164 [Cs]*, 30 July 2018, arxiv.org/abs/1807.11164. Accessed 23 Feb. 2021.

Zhang, Xiangyu, et al. "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices." *ArXiv:1707.01083 [Cs]*, 7 Dec. 2017, arxiv.org/abs/1707.01083.