Super Smash Ultimate Match Analyzer

Jay Yoo, Enya Liu, Bryce Caro

1. Introduction

Every great competitor knows that analysis and study are extremely important parts of learning and developing skills and knowledge; being able to go back and study past footage, to see what and when mistakes are made, and learning how to minimize error and learn the habits of opponents plays a huge part in modern sports and competition. In today's modern competitive landscape, this extends not only to traditional competitions but also competitions in the virtual space, namely Smash Brothers.

Smash, while on the surface seems simple and cartoonish, actually has a surprising amount of depth and skill in serious levels of play. Any competitive Smash fan will tell you about the sheer speed and intensity of the game, anywhere from professional players battling for international titles or friends going head-to-head at local tournaments and gatherings. Either way, Smash is very fast, and sometimes, it can be very easy to miss things when going back and reviewing footage. That's why we wanted to create a tool to help analyze Smash footage, and to allow players to get an entirely new perspective on Smash Bros. Ultimate.

While some applications that analyze Smash footage already exist (such as Project Slippi for Smash Bros Melee), none that rely solely on video data exist - that is, there is no application that can analyze and extract data without the use of external systems or hardware, such as input capture devices or separate real-time capture cards. We wanted to create a system that could return similar analysis, but without the need for these external devices. We wanted to allow players to see these statistics and analyses while only requiring the video footage of matches.

While both learning models and competitive gaming have been around for decades, we still have yet to see a successful and watershed combination of the two - and while our analyzer is relatively small in scope and may not be as refined and accurate as it could be, it can prove to be a stepping stone of potential for another huge role for machine learning.

2. Contributions

The project's main goal was to act as an analysis tool which provided information on what each character in a match was doing at a specific time. Due to the unexpected scope of the problem, we were only able to complete the first two parts of the system: an FRCNN for finding bounding boxes and character labels, and the binary classifier, which identified whether or not a character was currently performing a move or not.

In regards to algorithms, we utilized pre-existing network architectures, Keras FRCNN and Alexnet, and already existing training weights. The code for the Keras FRCNN was borrowed and augmented to fit our system implementation. For Alexnet, we used Pytorch's built-in implementation and restructured it to fit our pipeline's needs. The general structure of this system, using an FRCNN and 2 different Alexnet models, was an original idea. The video data was both taken from already existing Youtube clips and generated by our group. Because of the novelty of this projects goal, there did not exist any prelabeled data for Smash Ultimate matches - therefore, our group was required to label and annotate frames from various Smash Ultimate matches ourselves.

2.1. Idea and Algorithms

We decided to implement our first network using Keras FRCNN by another person's training of the network to detect Fox from Super Smash Brothers Melee: https://adamspannbauer.github.io/2017/12/28/super-smash-cv/. Seeing that he only needed to annotate about 300 frames made us hopeful that we wouldn't need too much data to get decent results. Unfortunately, because Melee and Ultimate are visually very different games, as well as the scope of our project being much larger, training this first model proved to be more difficult than anticipated.

2.2. Code

The project trained a Keras FRCNN to detect and draw bounding boxes around 8 separate characters. We took an already implemented Keras FRCNN from https://github.com/kbardool/keras-frcnn. We set up a Colab Notebook to run the python files. Unfortunately, it seemed this repository hadn't been maintained recently, and we ran into errors with function names changing, which we had to fix.

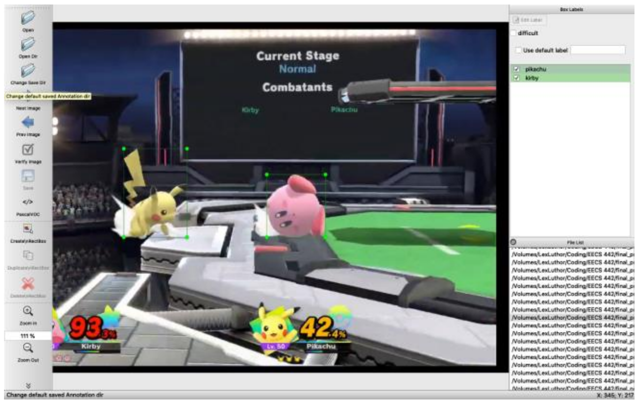
We also made changes to the test python file to better suit our needs. In order to get the test_frcnn.py file to work in Colab and be more flexible in general, we added an extra argument to specify in what directory the labeled frames would be saved to. We also added another argument and fixed the implementation so we could extract the actual prediction data and save it to a specified directory. This data was stored in a file and formatted in the same way that the train_frcnn.py file expected input. This was purposefully done to make the data more simple to parse.

Although we used the pre-trained Alexnet model for our binary classifier, we had to set up our own DatatSet class and handle training and testing. We took training and testing functions from homework 5.

To help simplify our pipeline, we wrote python scripts for parsing our annotations into a single text file for easy input. Because we ended up using two different programs for annotation, one compatible with Linux systems and one for Windows, the script parsed the data as both PascalVOC XML and JSON. This script then went through each of the annotation folders, split the data into training and test sets on separate CSV files, and then grabbed all the test images and put them into a separate directory.

Our system also included a script to run the first trained network on images to provide input for the second network, including cropping the images by the outputted bounding box.

2.3 Data



Since there was no pre-labeled data of Smash Ultimate footage that included bounding boxes and character annotations, we had to gather all of the data ourselves. Some of the data came from Youtube videos, and others were videos generated by our group recording matches from our Nintendo Switches. We did all the annotations by hand, as well. To do this, first we used ffmpeg to split up the video locally into frames, around one frame per second. After they were broken up into frames, we then labeled the frames with a labeling program. We uploaded these labeled frames into Google Drive so our Colab Notebook could use them. The programs we used to annotate were labelImg, which is shown above in the image, from https://github.com/tzutalin/labelImg, and VGG annotator, a simple browser-based image annotation tool (http://www.robots.ox.ac.uk/~vgg/software/via).

3. Data

Our dataset is limited to 8 characters, with different costume colors and around 5 different stages, played on either the Battlefield (triple-platform) or Omega (flat) forms.

When gathering the data for the bounding boxes, we went frame by frame and drew the boxes around each character shown in the frame. Again, this process had to be done manually. We also labeled the boxes with which character they encompassed. This proved difficult at times because characters often ended up blending in with the background or stage. Characters with complex clothing and weapons also proved difficult to annotate, as determining exactly the boundary of these complex characters was not always clear.

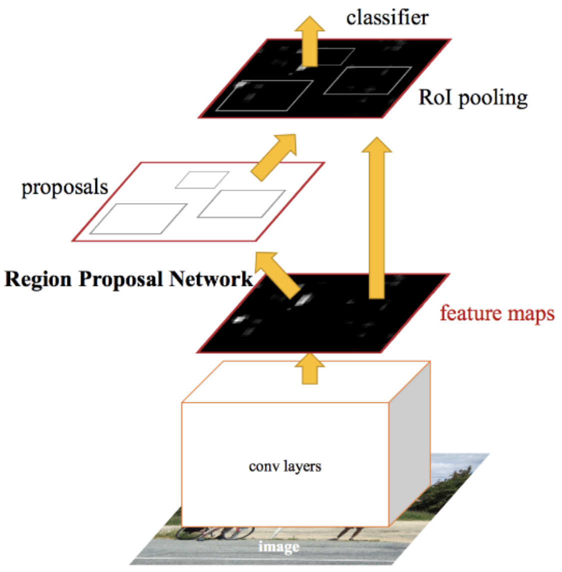
The camera also moves to focus on the characters as they fight, so the backgrounds aren't static. Sometimes, because of the complex visual effects and composition, the characters can also appear to be inside platforms or covered by other animations or effects, especially when the characters are using certain attacks or being hit. Some characters, such as Ganondorf, have capes or, like Lucina, have swords, and we wanted to include those in the bounding box. This sometimes led to much larger bounding boxes for these characters and brought in more background, which we worried could throw the network off. Because of these variations, there was inevitably a decent amount of noise in our dataset. However, we hoped the noisy data could help the network learn the character, instead of picking up on other factors like the default color of the character, giving our model more robustness.

Annotating the data for the idle binary classifier was especially tricky, because we had to differentiate ourselves whether or not the character was using a move. While this may sound simple, because of the complex and sometimes subtle starting and ending animations for each move, this proved to be a difficult task. We were able to work around this difficulty by recording games where a player-controlled character simply didn't use any moves and another game where the player-controlled character used lots of moves. Because of the large amount of idle movement though, this led to an uneven split in our training data, roughly 70% being idle images and 30% being non-idle. We kept our testing data an even 50:50 split between idle and non-idle.

For the character and bounding box detector, the network can't generalize on any new characters it's shown. Ideally, though, our idle binary classifier will be able to generalize on new characters, since the pose of characters walking or running are pretty similar.

4. Method

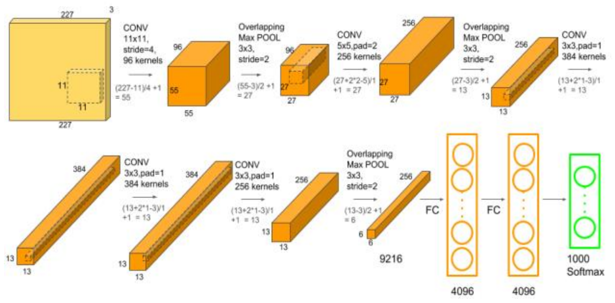
4.1. Character Recognizer and Bounding Box



Our character detector was implemented using a pre-existing model. Specifically we used Keras's Faster R-CNN implementation. The input to the training program for this network was a CSV file consisting of the filepath of an image, bounding box coordinates, and the label for the box. We added data augmentation, including vertical flips, horizontal flips, and 90 degree rotations, to generate more data from the relatively limited frames we annotated.

The optimizer used was Keras's Adam with a learning rate of 1e-5. This was how the training python file was initially set, therefore we felt that changing any of these factors may affect performance. For testing, the input to the test program was a directory for test images.

4.2. Idle Binary Classifier



Our idle binary classifier used the existing AlexNet model. The above figure shows the architecture of AlexNet. Since we only had 2 classes, idle and non-idle, we added a final layer to go from 1000 dimensions to 2 dimensions, as opposed to the 1000 classes AlexNet normally outputs.

We also included data augmentation, in the form of random horizontal flips, vertical flips, and 90 degree rotations. We used Cross Entropy as our loss function, and Adam for our optimizer, with a learning rate of 0.0005 and a weight decay of 0.0005. We achieved our best results training on a batch size of 64 and 124 epochs.

4.3. Damage Detection and Character Decider System

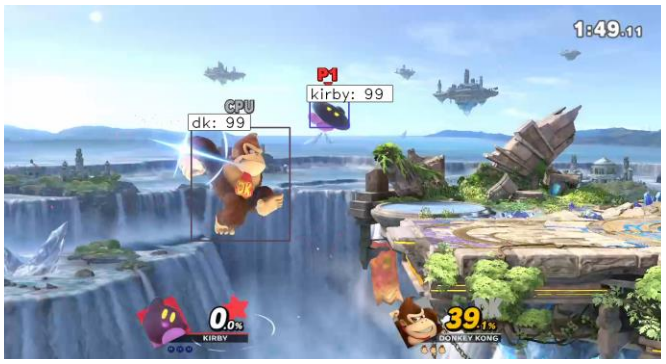
Initially, part of our project was to have a system that would read the percentages displayed at the bottom of the screen and record values, in order to create a time plot of damage over time for each player in the match. Our first plan included using a python OCR package, pytessaract, to detect these percentage values per frame - however, this proved ineffective due to the game's font and intense visual effects, causing pytessaract to get inaccurate or non-existent readings. We attempted to alleviate this issue through various preprocessing on the percentage images, including adaptive thresholding, using the color data, and applying gaussian blur. However, no methods we tried yielded consistent results, and our group ultimately decided that it was much more important to allocate our time towards the multiple models in the pipeline. Therefore, and unfortunately, this part of the project remains to be determined.

We also implemented a character decider that was supposed to go hand-in-hand with the damaged detection, so that we could match the damage data to a character. We gathered images of the 10 characters that we initially planned on using, although 2 of them were later cut out due to limitations on data gathering for the neural networks. We cropped the given image of a frame to extract the 2 character portraits then used ORB from OpenCV to detect features and select matches between an input image with each of the 10 character portraits we had. We calculated the sum of the 10 matches that had the smallest distance and that value was used as a distance metric to determine which character it was.

Our character images were of the characters at their default color. We tested with input images where the characters had different costume colors, and it seemed to work well. However, because our first model was trained to detect characters as well as bounding boxes, we decided that this part of the project was somewhat redundant and unnecessary to continue developing, especially considering the damage detection portion was also postponed.

5. Experiments

5.1. Character Recognizer and Bounding Box



Our best results consisted of a final training accuracy of around 89%, but we know that this is potentially due to overfitting since our dataset was relatively small and greatly limited in size by our group's ability to annotate and generate test data.

Because the performance of this network was heavily dependent on how visibly well the boxes fit around each character and whether the labels were correct, it was very easy to gain an understanding of how accurate our model was predicting correct boxes and labels. We fed it several video frames that were not in the training set for testing and observed each one to see whether or not the predictions were accurate. Our best result produced an estimate of 60% correctly labeled frames, with the other 40% either missing a label, having an incorrect label, or plainly drawing random bounding boxes around background or stage content.

5.2. Idle Binary Classifier

The best test accuracy we achieved for the binary classifier was 93%. Since our training data was a 70:30 split between idle and non-idle frames, if our network were to simply choose to label every input as idle, we would see 50% test accuracy, since our testing data was an even 50:50 split.

If labelling idle or non-idle was left to random chance, the probability of getting a 93% accuracy on the test data of 90 images, which would mean guessing 83 of the images correctly, would be $(0.5)^{83} = 6.36 \times 10^{-26}$. This goes to show that our network performs much better than random chance.

Even if our model guessed idle 70% of the time and non-idle 30% of the time, due to the training data division, the probability of it getting a 93% accuracy, assuming the majority of the images it got correct were idle, would be $(0.7)^{45} + (0.3)^{38} \ = \ 1.07 \times 10^{-7}$, which is still very small. Considering we were consistently able to get testing accuracy results over 90%, this shows that our model has actually learned something from the data and posture of the characters, and isn't just performing a random 50/50 guess.

6. Conclusion

While our end results are far from perfect, and there clearly needs to be much more work and data for our pipeline to function adequately, we believe that the Smash Ultimate match analyzer as it is currently shows the potential for computer vision and machine learning for the competitive gaming landscape. Had our group had more time and resources, and were able to generate or get access to a larger volume of annotated data, we believe that this model could create output similar to other analysis tools, but without the use of external hardware. For what needs to be done next for our analyzer, we believe that generating further labeled data to train the third model in our pipeline would be the next step, along with completing the implementation for the damage detection system such that it can create consistent readings. Hopefully, given more time in the future and more access to training data, the pipelined learning models could be refined to create consistent and accurate results.