Visual Identification of State-Space for ‘Street Fighter II’

Bryson Howell, Adrian Ruvalcaba, Sandesh Jain

Fall 2021 ECE 4554/5554 Computer Vision: Course Project

Virginia Tech

Abstract

Reinforcement learning has been used to train agents to play Street Fighter II and similar fighting games in the past [1,2], but these approaches often rely upon directly interfacing with the games’ code to extract state information. Since most video games are proprietary software and access to their code is limited, state information must be extracted visually from video footage of the game. The objective of this project is to apply computer vision techniques to extract a few key state features within the video game Street Fighter II for the Sega Genesis. This technique is accurate for a few features under ideal conditions, but could be made more robust to failure in future work.

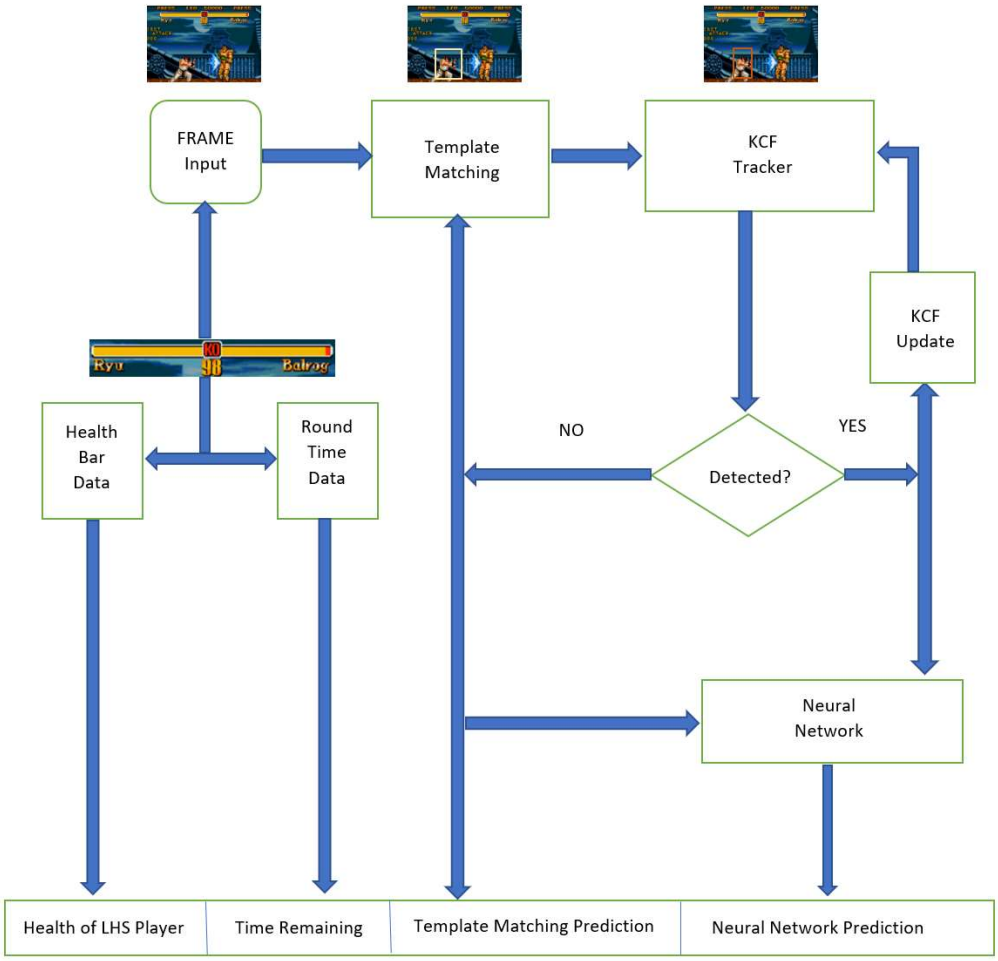
Teaser figures

Introduction

Visual perception is by far one of the most significant areas for exploration due to the sheer volume of associative and independent attributes that they represent for any specific entity given a background. This field has seen enormous growth in real-world applications such as Autonomous driving, robotics, and other visual data technologies. Now, consider an autonomous agent that has three paradigms of processing, Sense, Plan, and Act to perform a task in the real world. Within this framework of the agent's world, we can see that the 'Plan' state can only perform according to what the 'Sense' state perceives. As further developments were made, it was clear that adding all the computational burden on the Plan paradigm is not feasible, this gave rise to a host of high-level perception algorithms, Visual feature extraction being one of them. In this project we demonstrate how using the appropriate computer vision algorithms can help extract all the relevant features for a potential autonomous agent within the Street Fighter II game environment in close to real-time. Instead of using sophisticated deep learning architectures like CNNs, we use a set of computer vision algorithms and a relatively simple artificial neural network to extract a target set of features representing the state-space of the agent (this terminology is usually used within the reinforcement learning community). There are several state features within the Street Fighter II environment, each of which must be identified visually with a suitable technique. There are three major pieces of information related to the game environment that must be recognized: the time remaining in a match, the current health of each character, and the number of rounds won by each character. This information is presented in a static user interface, and each piece of information can be found efficiently through different means. The number of rounds each character has won are indicated by icons in the corners of the game screen, and can be identified by a simple similarity score. One of the most important state features within the Street Fighter II environment is the current status of each character. The status of a character is decided by actions that a character has taken or actions that the opponent has taken against the character, these actions are identified by the 'Character Action Identification' module. For example, if character A lands a "close standing jab" against character B, character A will be in the "close standing jab" status while character B will be in the "damaged" status. A character's status largely determines the actions available to that character. Using the previous example, character A will be able to follow up their attack with another while character B will be unable to act at all until they no longer have the damaged status.

Approach

The high-level roadmap for this project revolves around the below mentioned block diagram. To set the necessary background knowledge to understand this roadmap it is only fitting that we describe the functional overview of the game. Inside the arcade game SF-II we focus on four major attributes that could be significant to an intelligent planning system. The first is the location of the player character, this is one of the most critical operations, failing which will immediately invalidate the second attribute. Due to these reasons we consider two region-of-interest seeking algorithms namely Template Matching and the Kernelized Correlation Filter-based Tracker [5]. Here, the requirement for KCFT to run correctly is to have an initial ROI fed to it. The Template Matching Algorithm (TMA) suffices this need, for the TMA to detect such a character ROI, we utilize a separate template set for specific action characteristics (e.g., Block, Crouch, Idle, etc.) The initial state of the character when the game is about to begin will always be in Idle mode, hence, we reduce the TMA's template set to just one action characteristic i.e., Idle, resulting in an appropriate location for the character.



The next step now is to keep tracking the player position, now, since the TMA utilizes a looping structure to first apply all the templates for search and then return the template name with the highest confidence of match along with the character location. To skip this computational onus, we switch to KCFT [5] which by our own approximations runs at ~20 times faster than TMA. Details of this algorithm are mentioned in later sections. While the KCFT keeps updating and reporting the latest ROIs we move towards the 2nd significant feature of the game i.e., the action characteristic. For this feature we utilize an ANN that provides a limited set of characteristics from a collection of them provided by the RL agent internally. The network is trained using a customised training dataset consisting of the ROI subsections of the entire image and resulting in a final output that indicates the most likely action by the argument maximizing output off the 7 total output nodes. One key issue we deal with here is that as soon as the KCFT is unable to identify the ROI, the ANN cannot function which breaks not one but two of the four features in our State Space. To avoid this fiasco, we use the TMA to re-identify the character position and allocate it back to the KCFT. This ensures double safety of retrieving the features at all times but adds to the time complexity of our system. As the algorithm runs through the aforementioned pipeline, we perform 2 additional feature extraction tasks in parallel. The processing for all the features takes place 1 frame at a time, thus, the frame rate can be viewed as the clock rate for our system. The additional features extracted are the player's current health and the time remaining in the round. The player's health is estimated by determining the current x position of the edge of the player's health bar, while the time remaining in the round is extracted through a template match. We accumulate the ground truth sets using the DeepQ RL Agent, furthermore, there exists a discrepancy in the number of action characteristics that exist internally for SF-II which is 27 and the number of characteristics we report, we have a total of 7 characteristic predictions that can be made by the ANN. We keep this number lower to ensure only significantly different actions are predicted e.g., we have grouped all kinds of punches into a single action 'punch' and so on.

Dataset Creation

DeepQ RL Agent:

Given how our approach aimed to visually identify the state-space of an environment, we needed to gather multiple different videos of Street Fighter II gameplay. More importantly, we needed to gather videos with accompanying actions being taken during each frame. Without the pre-recorded actions, we would have no way of verifying that all of our predictions were correct aside from manually checking the individual frames, which is unfeasible for a large-scale project. We solved this problem by utilizing an existing Street Fighter II AI framework built off of OpenAI's Gym-Retro platform [3]. This AI framework uses a deep neural network to train an agent to play the game using certain values from the game's memory as input. We were able to extend this framework to report certain values from Street Fighter II's memory as well as the actions taken by the agent along with the images of the frames corresponding to each memory state. This approach allowed us to rapidly generate labelled datasets for evaluating the performance of our visual state space extraction techniques. We recorded datasets containing a full match between the AI agent and each of the available opponents in Street Fighter II: Blanka, Chun-Li, Dhalsim, E. Honda, Guile, Ken, Ryu, and Zangief. Each match lasts for a maximum of ~200 seconds and frames are sampled at a rate of ~40 frames per second. In total, we produced 39,367 labelled samples for training and evaluation.

Neural Network ROI:

In order for the Neural Network architecture to properly identify character states, it needed to be trained on a dataset of labeled character actions. While there are resources online with images of the character models [9], none of the datasets contained individual labeled images. Additionally, the images often had the backgrounds completely removed, which would be a problem for the Neural Network as it could potentially learn features that ultimately lead to inaccurate predictions. By utilizing both Template Matching and the datasets generated by the DeepQ RL Agent, we were able to generate a large training dataset for the Neural Network. This dataset consisted of 12 different labeled folders each containing cropped ROI images of the player performing different actions. These images were incredibly useful because they were not only correctly labeled and sorted, but we had examples from different locations in each level of the game. After removing incorrectly labeled images from the folders, we ended with a dataset of 50,960 different images. While not all of these images were going to be used in the project, collecting the large dataset allows for the possibility of larger complex networks with more accurate predictions in the future, as well as giving us flexibility to select different image groups.

State Space Identification

Template Matching:

Character statuses are conveyed visually to players through unique animations which we as humans can easily identify on the screen. This is not the case for computers, who interpret the image as a matrix of numbers. One way to detect characters is by utilizing Template Matching [8]. This method works by using a small template of the image we want to find and overlaying it on the target image we are looking at. We then calculate the cross correlation between the two images and store the value before slightly shifting the template location and again checking the cross correlation. Once the entire image has been traversed, the location with the highest correlation value gets returned as the best match for the template. While modern approaches to template matching can account for changes in the transformation of the object being recognized [4], these methods are not necessary for the Street Fighter II environment since character animations for specific statuses are only varied by vertical flipping. A template matching algorithm that prioritizes computational speed would be preferred, since the current state of the environment must be recognized in real-time for some reinforcement learning applications We utilized Template Matching in our approach with the main purpose being KCFT initialization [5]. While the KCFT is a robust method of tracking a Region of Interest, it requires this ROI to already contain the character model before starting the tracking. Normally this ROI is manually selected, but our approach aimed at a completely visually-based implementation with no user interaction. We can accomplish this by

taking advantage of a window when the game starts where no inputs are recorded for a few seconds. By Template Matching the first few frames, we can extract the location of the player and calibrate the KCFT before the round begins. We also tested Template Matching as a secondary method for determining the state-space of the environment. While the main algorithm utilized the KCFT and Neural Network architecture to get the player states, the comparison with a pure Template Matching approach would help determine which of the two methods performed better given our implementations.

KCFT:

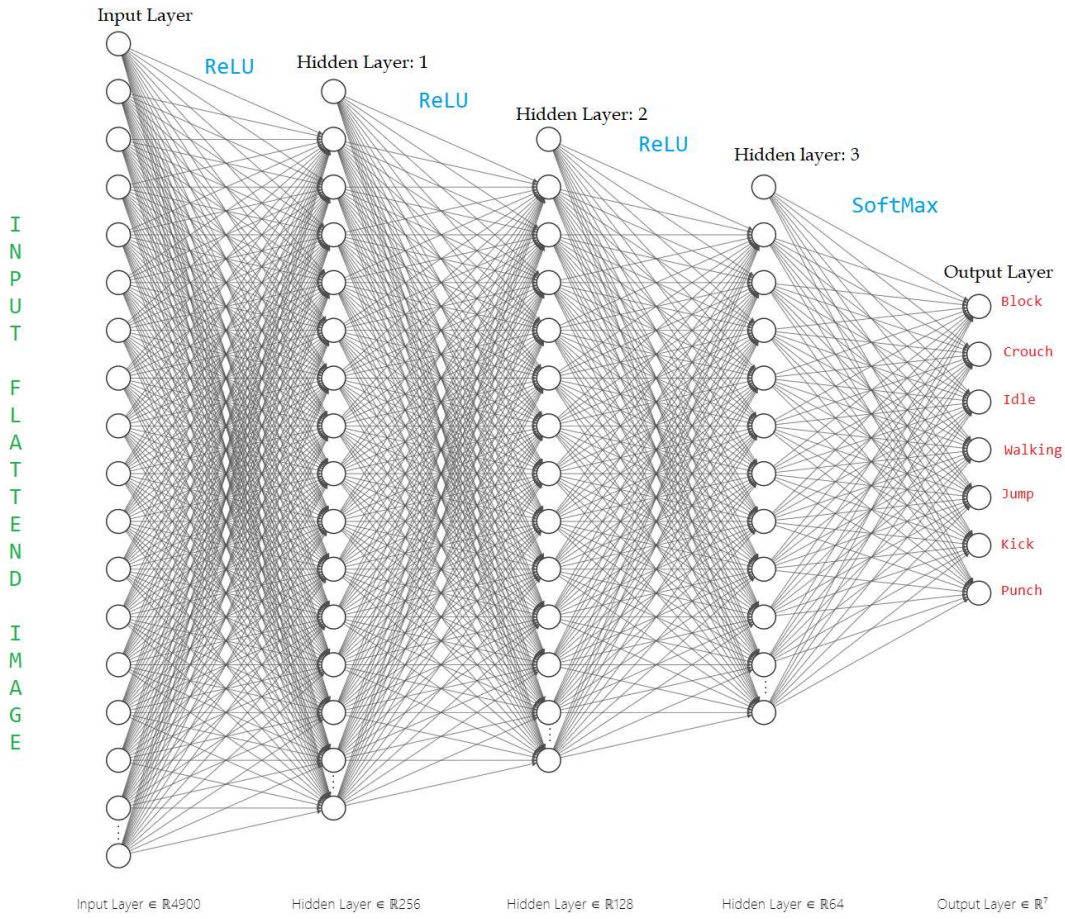
After the initial region of interest has been provided, we switch to an efficient tracker known as “Kernelized Correlation Filter-based Tracking” or KCFT [5]. The objective of this module is to track the game character ROI throughout the video game. The core idea behind KCFT is derived from ridge regression and solved in the Fourier domain with complex values, this is not a huge shift as the transpose of the data matrix with one sample per row gets replaced by the Hermitian Transpose to accommodate complex numbers. Additionally, the kernelization here is similar to the famous kernel trick used to map the input matrix by a non-linear kernel. Specific kernels like the RBF Gaussian kernel work for KCFT. The filter is trained from transitioned data points of the target window. Once the new ROI has been found the target is updated and the iterations continue until the end-of-frame or if the target moves out of the ROI faster than the KCFT output. The edge of the KCF tracker is the computational efficiency. The reason is that the computation can be performed effectively in the Fourier domain. Thus, the tracker runs in real-time, at an impressive frame rate. While KCFT [5] is handling the ROI extraction at a higher overall speed, it becomes difficult to obtain continuous results when there are intermediate frames missing. This observation is explained in more detail in a later section. Additionally, the basic functioning of KCFT expects the target entity to be moving, there’s virtually no means for it to distinguish between the Background and the Foreground other than the movements. In this game, we observe that the background also moves relative to the static observer, thus, it can mislead the tracker to instead track the background. A simple resolution to this issue is to control the areawide occupancy of the target entity with respect to the background, i.e., if we set our initial bounding box as close as possible to surround the character, then the disarray caused by background movements can be mitigated to a great extent.

Neural Network:

To Implement the following we use a 3 hidden layer neural network. We implement the ANN using high-level libraries such as keras from tensorflow. The 1st layer known as the Input layer consists of 4900 Neurons. The 1st layer accepts the Input Image into a form of a large row vector and generates the activation values for the hidden layer. It's usually believed that the initial layer of a neural network performs low-level image processing tasks such as edge detection. The hidden layer brings these low-level features together to form parts of a higher-level or associative feature. The dot products between the weights and the input are passed through a function known as the transfer function. The output of the transfer functions act as inputs to the 1st hidden layer which consists of 256 hidden neurons. Similarly, the next two hidden layers perform these computations in a similar manner with possibly different transfer functions. The final layer of our architecture also known as the output layer consists of 7 neurons which can be seen as a probability array wherein each neural output represents the probability of classification belonging to the associated label. The reason we moved ahead with the Rectified Linear Unit (ReLU) transfer function is to allow for the values to have some flexibility in these initial stages to form reasonable high-level features at the end. Adding Softmax functions created a bottleneck for the A Back propagation being the most important segment helps to correct the weight and bias mentioned earlier. The backpropagation uses a loss function to keep a track of its performance and updates the weights throughout the layers using partial derivatives and the chain rule. With appropriate datasets, the model gets trained as the number of epoch cycles pass by. The final layer uses a Softmax transfer function to produce an output compatible with one-hot encoding used in the training dataset. Additionally, mini batches are used to render the learning faster, however, a trade-off between speed and noisy output is inevitable. We present the information regarding each layer below:

Number of Neurons & Activation Functions:

Input Layer (0th Layer): 4,900
Activation function: ReLu [Rectified Linear Unit = max(0, z)]
Hidden Layer (1st Layer): 256
Activation function: ReLu
Hidden Layer (2nd Layer): 128
Activation function: ReLu
Hidden Layer (3rd Layer): 64
Activation function: SoftMax [1/(e-z + 1)]
Output Layer (4th Layer): 7



Optimizer:

We have used Adam Optimizer which is an adaptive learning rate algorithm and computes individual learning rates for different parameters. Adam can be looked at as a combination of RMSprop and Stochastic Gradient Descent with momentum. It uses the squared gradients to scale the learning rate like RMSprop and it takes advantage of momentum by using the moving average of the gradient instead of the gradient itself like SGD with momentum.

Loss function

We use sparse-categorical cross-entropy as a loss function to calculate the loss function for the generated hypothesis function over the true output equation. Using the following loss function, we finally calculate the cost function and try to have the necessary parameter for the minimization of the loss function over the iterations.

Health Bar Data Extraction:

The health of the player character is represented visually by a rectangular gauge in the top left corner of the screen. As the player takes damage, a yellow bar within the gauge begins to shorten which reveals a red background. Therefore, the player's current health can be estimated by determining the end location of the yellow bar relative to the start of the health gauge. The equation used for computing the player's current health percentage is displayed below. Health = (bar_edge / gauge_length) * 100 The position of the health gauge is static across all frames of the game, so the gauge length is treated as a constant value. Since the colors of the health gauge are constant across all states of the game, the edge of the bar is found by iterating through the pixels of the health gauge along the x direction and checking their color value. Once a pixel with the yellow color value is found, the x index of that pixel is used as the location of the

edge of the health bar.

Round Time Extraction

The time remaining within the match is displayed at the top of the screen as an integer representing the number of seconds remaining until the round ends. This value starts at ninety-nine and begins to count down to zero after a brief round introduction. If a player is knocked out, the timer will pause for a moment then reset to ninety-nine at the start of the next round. We first attempted to determine the round time through the use of the Python-tesseract library [6], which allows the use of Google's Tesseract optical character recognition engine. First, we processed the region of the screen containing the timer by thresholding the region so that only the pixels of the round timer text were present, then applied a Gaussian blur to the region to make the pixelated text easier to read. However, the text proved to be too difficult for the Python-tesseract library to recognize, likely because the OCR engine was not trained on pixelated block numbers similar to those present in Street Fighter II. Text recognition was implemented manually in Python. First, a binary thresholding algorithm is applied to the region of the screen containing the round timer so that pixels containing the three color values present in the round timer text will be set to black while all other pixels are set to white. After this thresholding, each digit in the timer is compared to a series of reference images containing the thresholded digits zero through nine. If the region containing the thresholded digit is an exact match to one of the reference images, that digit is returned as the text's value.

Experiment and Results

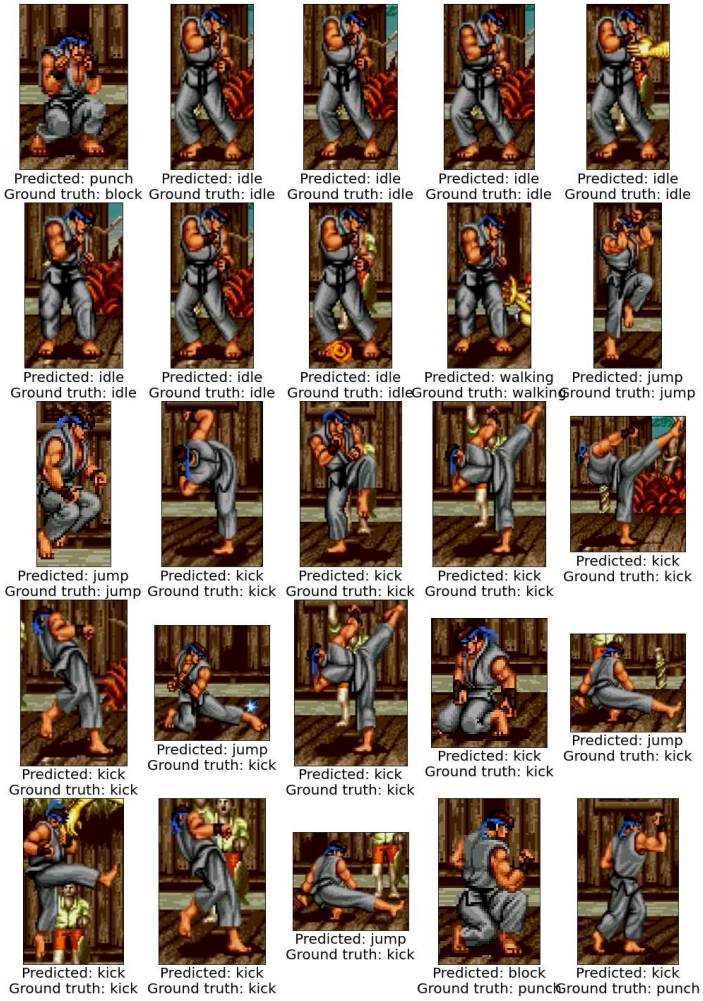
KCFT Evaluation:

To investigate the individual performance of the KCF Tracker [5], we temporarily detach it from the pipeline and observe its performance. With this focussed diagnosis, it is evident that there's a critical role played by the motion flow. In other words, the tracker can only perform well if the immediate reference region-of-interest should be found with very small changes. Thus, the changes in frames impact the performance of KCFT. Within the game when the player falls down, there's a stark difference between the frames. This renders the KCFT [5] unable to detect the matching ROI due to the missing intermediate frames that complete the flow of motion in a smoother manner. We can observe the loss of detection in the below case:

Another interesting observation we came across was the feature of re-detection in KCFT. This happens when KCFT is already in an undetectable state and as soon as the player resets their position, KCFT recaptures the ROI and continues to update the same. These do help the algorithm to recover and continue, however, the time lost in between these tracking failures does result in an accumulated error on the entire dataset. The final relative error of detections produced by the KCFT using the ratio of number of frames with correct detection and the total number of frames, this was found to be 53.20%.

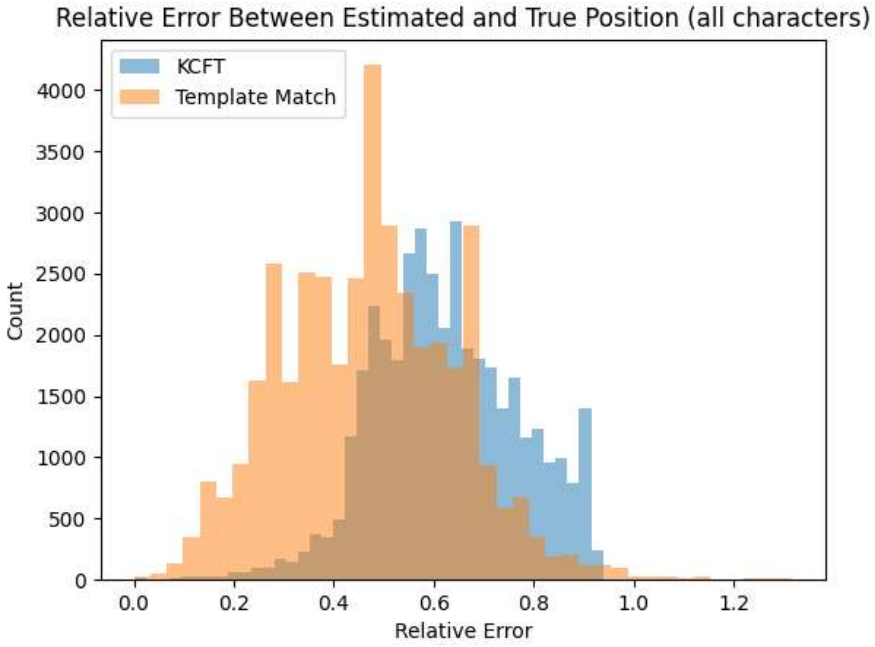
Neural Network Evaluation:

Our project involves two separate evaluations for the accuracy of the neural network. The first evaluation is compared against a separate dataset of randomly chosen regions-of-interest, with proper manual labeling. On this test dataset with 100 random samples of frames from across the game play, the ANN scores an accuracy of 66.67%. The manual dataset generation is aided by the TM algorithm to speed up the process of ROI extraction. The second evaluation is performed against the labels that the RL agent's environment predicts. We observe that there's a stark difference between the true character actions and what the environment reports. E.g When the character is KO'ed the environment takes a while to add that specific 'Falling' pose while the ANN has immediate access to those intermediate poses via the frame images of the game. Due to this discrepancy and inaccuracy at the rendering environment's end, we observe that only 15.84% of 50,960 sample frames from the environment match the ANN's predictions. Given the fact that the ANN performs significantly better when compared to manually verified testing data, we conclude that, in practice, using a separately calibrated and trained network for action recognition is more reliable than blindly trusting the agent's environmental prediction responses.



Template Matching Evaluation:

As with the KCF Tracker, it is important to first analyze the performance of the Template Matching by itself. Our results show that while the Template Matching approach was able to detect a large number of matches in the target images, there were still many incorrect predictions. When creating the ROI dataset for Neural Network training, for example, a total of 80,943 images were taken throughout several different levels. Out of these images, there were 50,960 correct and 33,983 incorrect matches, resulting in 59.99% prediction accuracy. These incorrect predictions were due to the background of the scene containing features that correlated with features in the templates, which is an issue magnified by the large number of different template possibilities. There are more opportunities for the background to match with a false positive, which drastically reduces accuracy. During the testing of state-space extraction, the prediction accuracy of the Template Matching dropped significantly to an average of 18.53%. This is likely largely due to a fault in what we considered “Ground Truth” actions from the RL Agent. While the Template Matching algorithm takes into account all possible animation frames for the character model during predictions, there are several actions that the RL agent can never take. For example, when the player is hit by an opponent an animation is displayed to indicate they are in the ‘hit’ state, after which the Template Matching will detect the animation and return a ‘hit’ prediction. The RL Agent, however, did not perform a ‘hit’ action and will instead return whatever action it was already in the process of taking. When comparing the two states for prediction validation, the Template Matching will not match the original dataset, even though it is the correct prediction. We can verify that the Template Matching is indeed detecting the player by looking at the L2 Norm of the difference between the centroids of the matched template and the validation dataset locations. The average position tracking relative error for the Template Matching rose back to 52.92%, which is near what we had previously calculated with dataset generation.

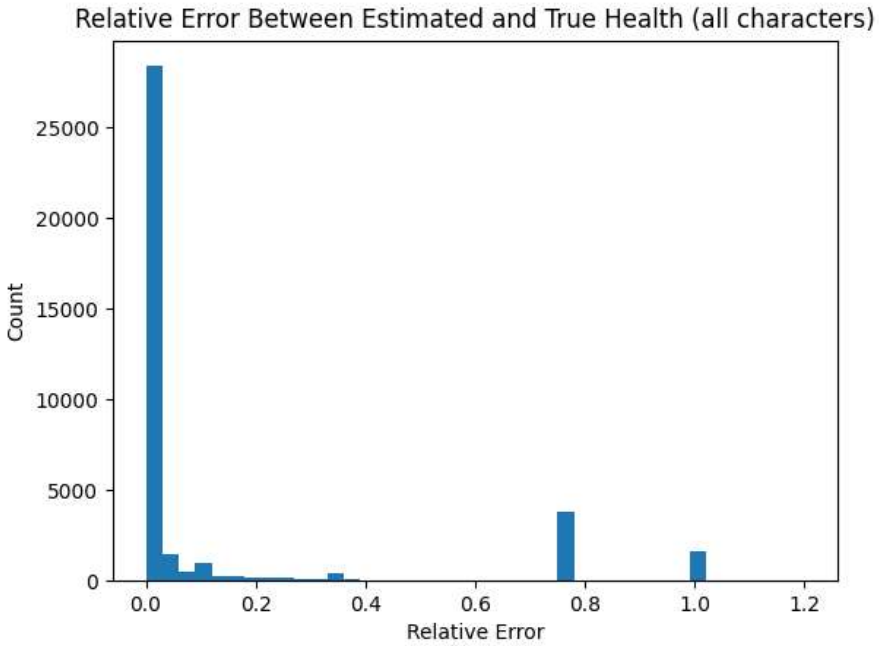


Comparing the probability distributions of the relative errors of the positions predicted by KCFT and Template Matching shows that Template Matching tended to be the more accurate method this is because as soon as the flow of motion is disrupted and the intermediate frame is not found, the KCFT does not update the locations. The automated testing method doesn't take such an event into account and adds to the final errors even when the undetectable flag is true. Template matching on the other hand loops through the entire frame with another outer loop to check for potential matches. This added resource for the TMA in addition to the slow speed of detections helps it gain an edge. It can be clearly seen that this is a classic case of a two way tradeoff between accuracy and speed. While KCFT performs poorly in terms of accuracy, it makes up for the speed. And the loss For reference, in our experimentation, KCFT runs at an astonishing frame rate of over 90 fps, whereas TMA runs at a little over 15 fps. Finally,a summary table of all the aforementioned accuracies compiled across the game and with manually verified dataset is given below:

Attributes	**Algorithms**			
Names:	KCFT	Template Matching (Position)	Template Matching (Classification)	Neural Network
Accuracy	53.20%	52.92%	60%	66%
Processing Time	3.31ms	66.82ms	66.82ms	31.41ms
Percent Match with Environment output	41.28%	48.18%	18.53%	15.84%

Health Estimation Evaluation:

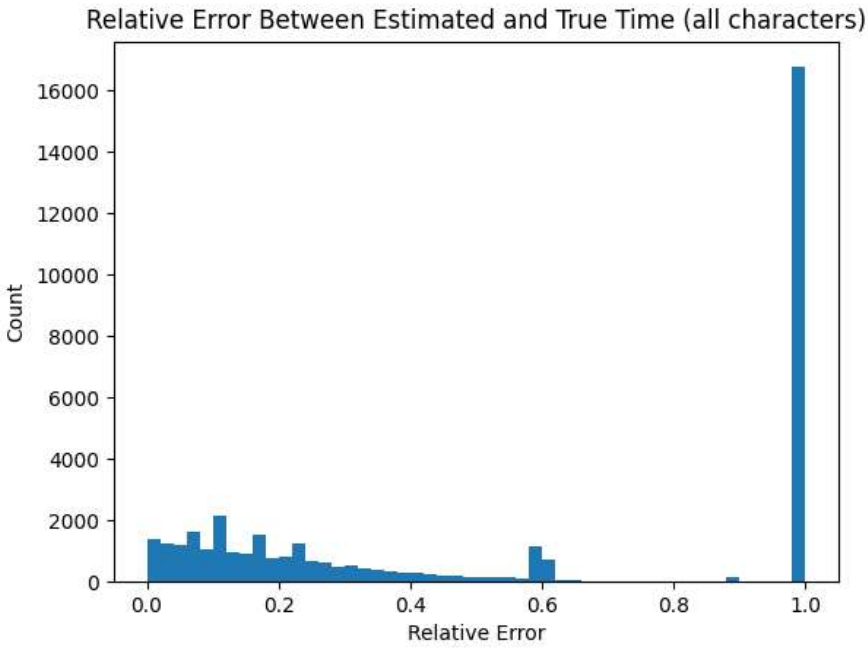
The accuracy of our health estimator was determined experimentally by comparing the predictions of our algorithm to the player health values stored in memory. Our algorithm provides health as a percentage, while the player's health in memory is represented as an absolute value where 176 represents full health while -1 represents a knock-out. We adjusted these memory values to be in the range of 0-100, then computed the relative error between our estimated values and the actual values. The probability distribution of this error for all samples across all datasets collected is displayed in the figure below.



Overall, our health estimations were extremely close to the values stored in memory. Investigating our results revealed that for frames where the health bar was not moving, our health estimate was nearly identical to the values stored in the game's memory. However, error is introduced while the health bar is decreasing after a player takes damage. The memory value of the player's health is instantly decreased when the player takes damage, but it takes up to 30 frames for this change to be reflected visually in the health bar. Even though our algorithm correctly finds the position of the health bar as it is moving, the resulting health calculation will be incorrect. Additionally, the figure reveals that there are a few outlier samples that produce very high relative error. These occur during the frames in between rounds, as the screen fades to black which makes the health bar impossible to detect. Our algorithm will determine that the player's health is 0 during these frames, while the value in memory is reset to its max value of 100.

Round Timer Recognition Evaluation:

The performance of our round timer estimation was determined through a similar method as the previous test for player health. However, comparing the time provided by our template matching method to the value stored in memory was more difficult since the memory value representing time is initialized to a value around 39205 when a round begins and is decremented every frame of the game. Since the game runs at around 40 frames per second, a simple formula for converting the time stored in memory to the time remaining in the round in seconds is:
Memory time in seconds = 99-((39205-memory_time)%40)
Using this formula to convert the actual time stored in memory into the time remaining in the round, we found the relative error between our visual estimation and the true value for every sample of all datasets. The probability distribution of the relative error is displayed below.



Clearly, the relative error between our measurements and those in the game's memory tended to be extremely high. However, qualitatively investigating the times reported by our template matching algorithm reveals that our method is able to correctly identify the time remaining for the majority of frames. The smaller errors depicted in the figure likely come from inaccuracies in the way we convert the memory time into a time in seconds. If the round doesn't start at exactly the time value we assume it does or the game does not run at a constant 40 frames per second, the time in seconds we calculate will slowly become desynced from the true time and produce an increasing amount of error. Additionally, cases of extreme error are introduced during the frames in between rounds. When the timer disappears, our template matching algorithm will report the time as 0 while the value in memory remains as the time that the round ended.

Conclusion

In conclusion, we believe that specific vision-based algorithms tailored to achieve their respective goals add to the overall visual perception awareness of an intelligent agent. This not only takes the computational burden off the Planning module but also provides immunity against faulty environments. In this project such an approach was considered to counter error-prone player characteristics in the Street Fighter II (Genesis) game environment. Within this framework, we successfully extracted four significant features accompanying the character state. All of these features are derived from a specific logic with low generalization for faster computation or use a set of generalized algorithms to achieve an optimal balance speed and accuracy. The usage of these algorithms have been planned to be interdependent in nature shielding one's weakness to improve overall performance. Such an instance is the functioning of the character location + action recognition algorithm pipeline. Here, the TM algorithm which is slower but has the ability to identify the character location using its template set, supports KCFT which is faster but suffers a loss of accuracy due to its reliance on the motion flow and frame continuation properties on the display. Our methods for determining the player's health and round time were accurate during normal gameplay, but encountered issues during the time between rounds. Finally, we uncovered and resolved the issues with receiving character state space properties by directly using the RL Agent's environment. These values were heavily inaccurate and had update issues discussed in the Experiment and Analysis section. With the use of manual verification of the data sets used by the TM algorithm (template set) and the ones fed to the ANN, we were able to achieve a fair accuracy and determine which properties of the environment were flawed. Throughout the project, as mentioned earlier, we were able to maintain an approximate processing frame rate of ~30 fps without using GPUs.

References

[1] Hasan, M. (2020, May 20). Street Fighter II is hard, so I trained an AI to beat it for me. Towards Data Science. https://towardsdatascience.com/street-fighter-ii-is-hard-so-i-trained-an-ai-to-beat-it-for-me-891dd5fc05be
[2] Oh, I., Rho, S., Moon, S., Son, S., Lee, H., & Chung, J. (2021). Creating pro-level ai for a real-time fighting game using deep reinforcement learning. Ieee Transactions on Games, (2021). https://doi.org/10.1109/TG.2021.3049539
[3] Hustle, C. (2020). StreetFighterAI. https://github.com/corbosiny/StreetFighterAI
[4] Wakahara, T., Yamashita, Y., & 22nd International Conference on Pattern Recognition, ICPR 2014 22 2014 08 24 - 2014 08 28. (2014). Gpt correlation for 2d projection transformation invariant template matching. Proceedings - International Conference on Pattern Recognition, 3810-3815, 3810–3815. https://doi.org/10.1109/ICPR.2014.654
[5] Henriques J. F., Caseirio R., Martins P., Batista J. High-Speed Tracking with Kernelized Correlation Filters. IEEE Trans on PAMI 37(3):583-596. 2015
[6] Hoffstaetter, S. (2021). The Python-tesseract library. Python Software Foundation.
[7] Bradski, G. (2000). The OpenCV Library. Dr. Dobb's Journal of Software Tools.
[8] Thomas, L.S.V., Gehrig, J. Multi-template matching: a versatile tool for object-localization in microscopy images BMC Bioinformatics 21, 44 (2020). https://doi.org/10.1186/s12859-020-3363-7
[9] Zymeth, Lord. "SNES - Super Street Fighter II: The New Challengers - Ryu." The Spriters Resource, https://www.spriters-resource.com/snes/supersf2/sheet/5557/.
