The Game Imitation: A Portable Deep Learning Model for Modern Gaming AI

Zhao Chen
Department of Physics
Stanford University, Stanford, CA 94305
zchen89[at]Stanford[dot]EDU

Darvin Yi
Department of Biomedical Informatics
Stanford University, Stanford, CA 94305
darvinyi[at]Stanford[dot]EDU

Abstract

We present a purely convolutional model for gaming AI which works on modern games that are too complex/unconstrained to easily fit a reinforcement learning framwork. We train a late-integration parallelized AlexNet on a large dataset of Super Smash Brothers (N64) gameplay, which consists of both gameplay footage and player inputs recorded for each frame. Our model learns to mimic human behavior by predicting player input given this visual gameplay data, and thus operates free of any biased information on game objectives. This allows our AI framework to be ported directly to a variety of game titles with only minor alterations. After training, we report a top 1,2, and 3 (of 30) classification accuracy of 80%, 92%, and 95%, respectively. We also demonstrate that for Super Smash Brothers, our AI can run in real-time on moderate hardware and, with minimal test-time tweaking, is competitive with the most difficult CPU AI available within the game.

We also discuss results of applying our model to Mario Tennis, a different game with vastly different objectives from those of Super Smash Brothers. We finish with a discussion of model advantages and disadvantages and also consider methods of building more complexity into our model for increased robustness.

1. Introduction and Related Work

In 2014, Google DeepMind successfully used convolutional neural networks with deep reinforcement learning to teach a computer to play classic score-based Atari games[1]. We propose to study a complementary problem - that of teaching a computer to play modern games that are not score-based through pure mimicry of human actions viewed through gameplay screenshots. Primarily, we propose training a convolutional network system on Super Smash Brothers on the Nintendo 64 using both gameplay frames ($\sim$128x128px images) and key input generated from a human player (who will be one of the researchers). Given a large set of data, we hope to show that a computer will be able to successfully play in real time a complex game at the level of a competent human player. For comparison purposes, we will also train the same model on a Mario Tennis dataset to demonstrate the portability of our model to different titles.

Reinforcement learning requires some high bias metric for reward at every stage of gameplay, i.e. a score, while many of today's games are too inherently complex to easily assign such values to every state of the game. Not only that, but pure enumeration of well-defined game states would be an intractable problem for most modern games; DeepMind's architecture, which takes the raw pixel values as states, would be difficult in games with 3D graphics, dynamic camera angles, and sporadic camera zooms. Our aim is to create an algorithm that would extend previous AI gameplay capabilities to games that are too complex to easily solve via a predefined notion of value, and would also imbue computers with more human-like behaviors that make for more realistic AIs in game applications. To circumvent the complexity issues that arise with deep reinforcement learning techniques, our method will solely rely on learning how to mimic human gameplay from correlating on-screen visuals with their corresponding human actions. We will show that we achieve reasonable results with our framework despite having much fewer data points and much less training time compared to the deep reinforcement learning setting. Our model would also be more easily portable across game titles, as the algorithm operates free of any assumptions of same objectives and a new system can be trained for a different game as soon as more gameplay is captured.

Also related to our topic is work on video classification. Our algorithm is fundamentally a frame classification algorithm, i.e. what button combination (out of 30 possible classes) should be pressed at any given frame for some point in time? However, it is natural that information in temporally earlier frames should affect our decision for what to do next. Looking into previous experiments done on large-scale video classification[2], we can see that there are several different ways to deal with input data with temporal

information: the three primary methods are single frame, early integration, and late integration. We discuss these methods and their implementations on our data set in section 4.

2. Problem Description

As mentioned in Section 1, we propose a supervised learning, classification problem as an alternative to deep Q learning for gaming AI. More precisely, our eventual final classification pipeline takes as input a concatenation of four temporally sequential gameplay images, each of dimension $(128, 128, 3)$, making the input a total size $(4, 128, 128, 3)$. We then use a convolutional neural network to output 30 class scores, each corresponding to a different possible input command on the Nintendo 64 console. We not only want to see that our model can correctly predict button presses on a held-out test dataset, but also want to evaluate the efficacy of our neural network in actually controlling a game character, and so will feed these scores into a test-time module which will then send input commands to a game character during a live game against a CPU opponent.

3. Data Collection and Preprocessing

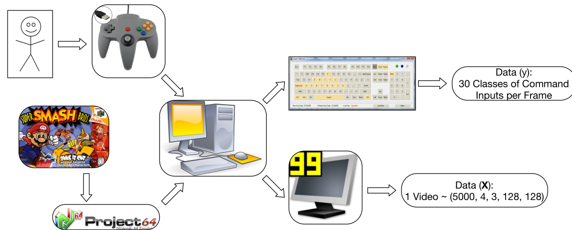
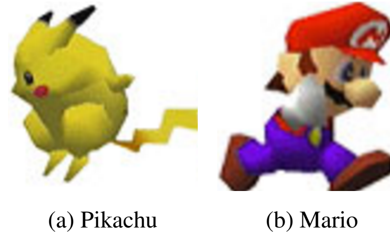


Figure 1: **Data Collection Pipeline.**

The data was collected by the authors via a combination of Nintendo 64 emulation and screen capture tools. The Nintendo 64 and games (Super Mario Bros and Mario Tennis) were emulated using Project 64 v2.1[3][1]. Keyboard Test Utility, which visually registers key pushes on a windows machine, was used in parallel to keep track of the button presses made during the gameplay. The full gameplay was captured with the screen capture program Fraps v3.5.99[4] at 30 frames per second. Figure 2 shows a frame of the screen capture data. We then extract the game window frames as our inputs and the keyboard window from which we can extract our ground truth. 60 5-minute games were played of Super Smash Bros, giving approximately 600,000 frames of gameplay with corresponding key presses. 25 games (each game the length of a normal game in tennis) were played of Mario Tennis, giving approximately 125,000 frames of data. To create a more structured data set, the game parameters were kept constant. The player always played as Pikachu against Mario in Super Smash Bros., (see figure 3), and the game stage was also always Hyrule Castle. In Mario Tennis, the player always played Yoshi against Birdo on a clay court. A schematic of the data collection pipeline is shown in Figure 1.

For our model to truly fit within an end-to-end context, preprocessing was done only to a minimal extent. Each frame of the video was downsampled from an original size of $(334, 246)$ (in pixels) to $(128, 128)$. The downsampling was *not* done by bicubic interpolation, which is a pretty standard image resizing metric, but rather by a nearest neighbor interpolation. Thus, every pixel in our final downsampled frame will be a pixel from the original image that was close in position to its final downsampled location. We made this choice because we realized that as these game sprites are somewhat small (they occupy about 1-3% of the screen as can be seen in figure 2), bicubic downsampling muted a lot of the colors of these sprites. By averaging over a larger area, specific textures and features could be lost.

Figure 2: **Screenshot.** Our screen capture setup. We capture the visual game data together with the keys pressed for each frame of gameplay. The keyboard in the bottom right shows currently pushed keys (orange), previously pushed keys (yellow), and inactive keys (white).



(a) Pikachu (b) Mario

Figure 3: **Character Sprites**

[1]An official copy of Super Smash Bros. and Mario Tennis along with a Nintendo 64 machine are owned by the authors ensuring fair use, but an emulator software was used for easier interfacing with computer software.

Thus, even though nearest neighbor sampling is known to be a much noisier downsampling metric, we chose to try to preserve the stronger colors of our images. The only other pre-processing we did before inputting our data into the models described in section 4 is subtracting the mean image. Once we split up our data into training, validation, and testing sets, we took a pixel-wise mean over all frames in our training set, and all frames had that mean image subtracted before being input through our neural net.

4. Model and Methods

Three main convolutional neural network architectures were considered as possible solutions to our "frame classification" problem: single frame classification, CNN with early integration of video frames, and CNN with late integration of video frames. The foundational CNN architecture for all three of our models is an AlexNet. We will minimize Softmax Loss

$$\mathcal{L}_i = -f_{y_i} + \log \sum_j e^{f_j} \tag{1}$$

with respect to our backpropagation. As we discussed in lecture, the Softmax Classifier minimizes the cross-entropy between the "true" distribution of our class probabilities (which should have $p = 0$ everywhere except a $p = 1$ at our correct class) and our estimated distribution. This is useful for us as it will allow us to treat the final scores as probabilities from a sampling distribution (see section 4.5).

Below, we describe our three main approaches in detail. We will then go into detail about how we trained those models and how we tweaked the model at test time to optimize performance during live gameplay.

4.1. Single Frame CNN

For our single frame CNN, we use a vanilla AlexNet structure. We design our AlexNet to have the following layers:

1. INPUT: $128 \times 128 \times 3$
2. CONV7: 7×7 size, 96 filters, 2 stride
3. ReLU: $\max(x_i, 0)$
4. NORM: $x_i = \frac{x_i}{(k + (\alpha \sum_j x_j^2))^\beta}$
5. POOL: 3×3 size, 3 stride
6. CONV5: 5×5 size, 256 filters, 1 stride
7. ReLU: $\max(x_i, 0)$
8. POOL: 2×2 size, 2 stride
9. CONV3: 3×3 size, 512 filters, 1 stride
10. ReLU: $\max(x_i, 0)$
11. CONV3: 3×3 size, 512 filters, 1 stride
12. ReLU: $\max(x_i, 0)$
13. CONV3: 3×3 size, 512 filters, 1 stride
14. ReLU: $\max(x_i, 0)$
15. POOL: 3×3 size, 3 stride
16. FC: 4096 Hidden Neurons
17. DROPOUT: $p = 0.5$
18. FC: 4096 Hidden Neurons
19. DROPOUT: $p = 0.5$
20. FC: 30 Output Classes

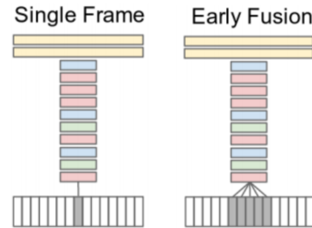


Figure 4: **Single Frame and Early Integration.** This figure shows two of the three convolutional neural network architectures that we considered. On the left is the single frame CNN, which is just a vanilla image classification CNN. On the right is our early fusion model as described in section 4.2. This figure was taken from Karpathy's "Large-scale Video Classification with Convolutional Neural Networks."[2]

We can see that our single frame CNN turns our supervised learning problem into a simple single image classification problem. Given any frame, we will learn what would be the most probable button press class for that frame. We can see the validation accuracy through train-time in figure 7.

However, it does not make sense to ignore the temporal information of our data set in our model. For example, we can think of one possible frame where Pikachu is next to Mario. In such a situation, depending on whether they are currently moving together or moving apart, we might want to choose a different action at that given state. This brings in the idea of using previous frames to better understand features such as velocity, acceleration, and higher-order temporal derivatives.

4.2. Early Integration CNN

The most naive incorporation of temporal information would be to treat the time dimension as another channel in the depth dimension of our input data. That is, if we consider four time-points of image data, each image already having 3 RGB channels, we would concatenate these frames for a final input of dimension of $128 \times 128 \times 12$. This is exactly the idea behind an early integration CNN. We will use the same layers 2-20 proposed in section 4.1, but our first layer will be

1. INPUT: $128 \times 128 \times 12$

as we will consider four time-points of image data.

Our frames will be $t = \frac{1}{6}$s apart. Thus, a single input piece of data for us covers a timespan of half a second. This separation value was chosen through intuition, as $\frac{1}{2}$s seems

to be a reasonable timescale for gameplay memory, and due to time constraints was not tuned as a hyperparameter.

The main problem with this method lies within the fact that in an early integration CNN, only the first convolutional layer is directly coupled with time. Once the first layer does its convolutions, all temporal information has been collapsed into the first layer activations. Thus, all subsequent layers do not have direct access to the time information in a separated format, diminishing their ability to learn temporal features. Our desire for effective learning of temporal features leads us to our final model: the late integration CNN.

4.3. Late Integration CNN

Similar to section 4.2, we will still consider four time points of image data, each consecutive points separated by $\frac{1}{6}$s. However, rather than combining all of our input data at once in the beginning as in early integration, we will let each frame go through a fully convolutional network first. We apply layers 2-15 from the architecture presented in section 4.1 on each of our four frames of size $(128, 128, 3)$.

We can then merge the final activations from these four independent fully convolutional networks and connect them to a fully connected neural network. The fully connected neural network will have the same design as layers 16-20 from the architecture presented in section 4.2. Our final model architecture is schematically shown in figure 5. We will see in section 5 that our choice of the late integration model is supported by improved validation accuracy.

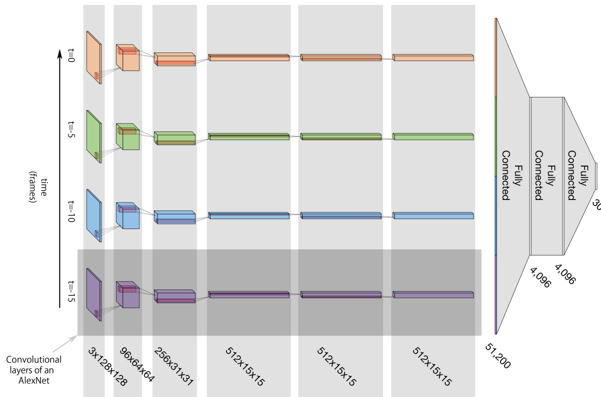


Figure 5: **Late Integration.** This figure shows our implementation of a late integration convolutional neural network as described in section .

4.4. Model Training

We perform training on a Nvidia GTX 870M GPU and primarily use Lasagne/Theano to build our network.[7, 8, 9] We run a small number (less than 10) of cross-validation tests on *training loss* with randomized learning rate and regularization, running our model for less than 500 iterations to save on compute time. We use training loss because it is much smoother and easier to interpret than validation accuracy especially given the low number of cross-validation iterations. We also check that our final model has validation accuracy which follows training accuracy closely, showing a lack of overfitting and thus that doing cross validation on training loss is valid. We find that the model training efficacy is generally insensitive to regularization but is very dependent on learning rate, and we settle on the following training parameters:

- Learning Rate: 1e-4
- Regularization Constants: 1e-7
- Update Rule: Adam (used as a safe default)
- Annealing: 0.95 every 5000 iterations
- Batch Size: 25 (maximal size given our GPU capacity)
- Training Time: 2 epochs over 2 days.

Note that this is significantly less training on much less data when compared to deep Q learning methods like in [1], which used 10 million frames of gameplay and 100 epochs.

Batches were randomized across the entire training data set and saved in hdf5 format before training began. 15000 batches were generated for Super Smash Bros to correspond to 1 epoch of training, while 5000 were generated for Mario Tennis. We do not use transfer learning, as we are classifying actions and not object type. Our input resolution of $(128, 128)$ is also different from the input resolution many of these other models are expecting - this does not break transfer learning, but may degrade performance. We also do not augment our data; since all our data was generated in an identical context of a game emulator, we expect the data set to be quite robust and for augmentation to have limited benefit.

4.5. Test-Time Tweaking for Live Simulations

Unfortunately, taking the pure maximum softmax class score as the next input for a live simulator does not produce reasonable test-time behavior; the reason for this becomes evident if we look at the histogram in figure 6, which shows the frequency of occurrences of each class in the training data. We can see that classes 0 (None), 26 (Left), and 27 (Right) are by a large margin the most represented in the training set, and thus their scores will tend to be inflated at test-time as well. For reasonable test-time performance, we must compensate for the fact that our training data was quite lopsided.

First, because our top 3 validation score (see Section 5) exceeds 96%, we take the top 3 class scores for each forward pass, renormalize them to sum to one, and then treat those renormalized scores as a probability distribution. We sample from this distribution to arrive at the final chosen class. This allows us to inject some notion of stochasticity

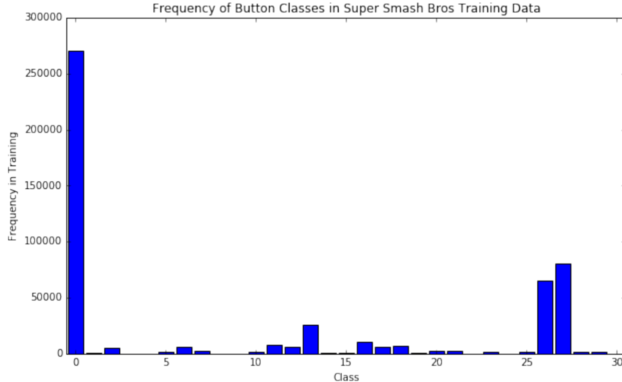


Figure 6: **Training Data Histogram.** Histogram of the frequency of all 30 classes in our full training data set.

into our model, and incorporates within our model the fact that a human player may act differently in similar situations.

This, however, still leaves some classes at a disadvantage because they were highly unrepresented in our data set and do not have a score in the top 3 classes very often. One solution is to weight the loss function at training time so that the loss penalty of misclassifying any particular class is equalized. However, this poses major disadvantages: this will necessarily decrease our validation accuracy and muddy our interpretation of this test metric, and it is also not necessarily true that equal penalties across classes is the correct weighting scheme. Finding optimal class weights would also be very computationally expensive, as we would have to re-tune the network each time. Thus, we keep the network we train unweighted and rely on test-time tweaking to adjust for training set imbalance.

Our solution is to bias each softmax class score by two weights:

$$s_c \mapsto \frac{b_c s_c}{|\#c|_{train}} \tag{2}$$

where s_c is the score for class c and $|\#c|_{train}$ is the number of occurrences of class c in the training set. Thus, we see that we first linearly penalize class scores by their training set class frequency. Beyond that, we also assign each class a weight bias b_c which accounts for any additional bias necessary to allow under-represented classes to be represented in the final results. For instance, if it is found that Pikachu never uses the Up+B button combination during gameplay, b_c for this class can be adjusted upwards. Currently, this presents an additional 30 hyperparameters to search over manually, but there are ways to learn these biases automatically from simulated gameplay (see Section 6.2).

5. Results and Discussion

Our major results are for Super Smash Bros. We discuss comparisons with Mario Tennis in a later session. We use top N validation accuracy[2] as a metric for preliminary model evaluation; we can also generate a 30 x 30 confusion matrix heat map on a test set to visualize specific problem areas with our final classifier. We also use performance in live games of our neural network player as a crucial evaluation metric.

5.1. Validation Accuracy

For comparison of the three main models we described in section 4, we recorded top-1 validation accuracies over one epoch of training, and the results are shown in figure 7. We can see that the late integration does indeed beat out both early integration and the single frame model. However, this difference is not huge, and we hope to show with some visualizations in section 5.2 that the late integration model can really learn not just spatial information, but the evolution of spatial information over time.

Another metric we can use to check our classification

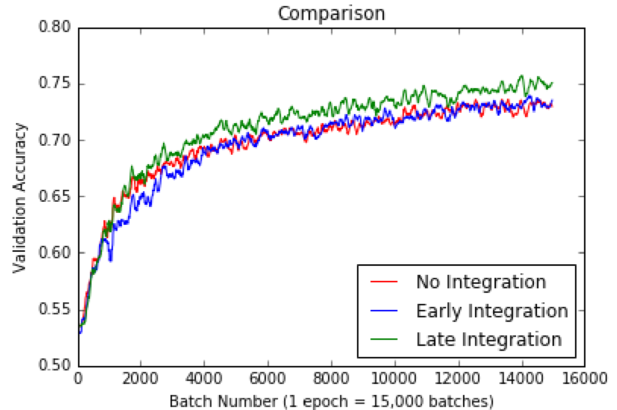
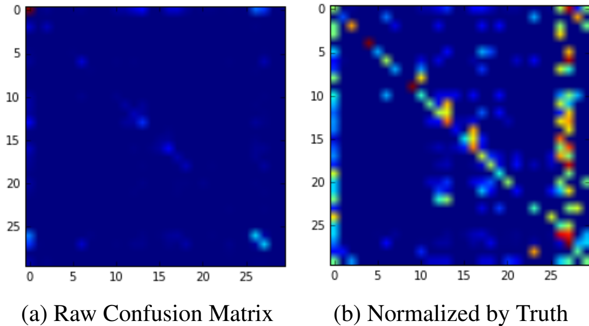


Figure 7: **Validation Accuracy of our Three Models.** This figure shows the validation accuracy of all three time-integration models investigated in section 4. We trained all three models for 1 epoch (15000 batches of 25 frames each).

accuracy is a confusion matrix for all 30 classification. As a 30×30 grid of numbers is a bit difficult to parse, we will be using a heatmap to visualize our predictions. We can see our confusion matrix below in figure 8. The heat map on the left represents the raw confusion matrix counts. Because of the highly skewed distribution of class 0 in our dataset, a better way to visualize the matrix might be to normalize by rows (corresponding to true class values). We see by this heatmap

[2] Note that because we cross-validated on training loss instead of validation accuracy, the validation accuracy is equivalent to a test accuracy.

that the by far the classes that induce the most misclassifications are classes 0, 26, and 27 (their columns in the heat map are significantly more lit up). This is very consistent with the lopsided training data issue discussed in 4.5 and even further demonstrates that the bias offsets discussed in that same section are well-motivated. After 2 epochs of training



(a) Raw Confusion Matrix (b) Normalized by Truth

Figure 8: **Confusion Matrix Heatmap**

each, we arrive at the validation scores in table 1. We can see that the correct button is within the top 3 softmax scores over 96% of the time, which is quite a high accuracy.

Game	Top 1 Error	Top 3 Error	Top 5 Error
Super Smash Bros.	79.9%	96.2%	98.7%
Mario Tennis	75.2%	96.0 %	99.5%

Table 1: Validation Accuracy After 2 Epochs Training

5.2. Validation Set Saliency Maps

We will visualize the saliency map for one particular move: "`DOWN+B`". Figure 9 shows the saliency map, which is visualized with respect to our 2-epoch late integration model.

We chose the move "`DOWN+B`" because it is a move that generally takes aforethought. The fact that the move incurs a latency before the attack executes necessitates the player to anticipate when it would be a wise time to use the move. With figure 9, we can show that our model did learn a strong time-dependent relationship when it came to this move.

Starting from $t = -15$ frames, we can see that our network concentrates on Mario's position. However, when compared to the next sequential saliency map frames, we can see that the response is much more diffuse. Indeed, for $t = -10$ and $t = -5$, we see significantly stronger localized responses with respect to both Mario and Pikachu, visualized as tighter clusters of red and yellow. Finally, what is also interesting is that with at least the "`DOWN+B`" class, the $t = 0$ frame seems to be the least important in terms of response. We do see some yellow and blue (the complement of yellow) around Pikachu's location, showing some level of localization for the Pikachu sprite, but when compared

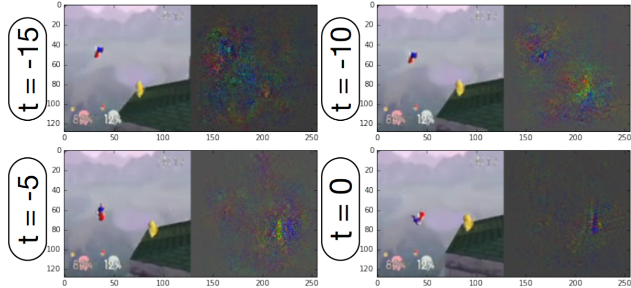


Figure 9: **Saliency Maps.** We show a saliency map for input data to our late integration CNN with particularly high score for the "`DOWN+B`" class. As our data comes from four different time points, we have divided up our saliency map into four images for each of those time points for easier visualization. The units of the time points are in frames, with each frame corresponding to $\frac{1}{30}$s.

to the other three saliency maps, the signal is quite mute. With this visualization, we can see that our late-integration model successfully incorporates previous temporal frames to make its decision.

5.3. Performance on Live Gameplay

We find that our model runs at test-time with a latency of 300ms. This is only slightly below the average human reaction time ($\sim$ 250ms) and so we can reliably run our neural network in real time at 60fps.

To evaluate our model's practical performance, we pit our fully CNN-controlled Pikachu against the pre-packaged Mario game AI. We run 10 games against a level 9 (highest level) Mario AI and 10 games against each a level 3 and level 6 Mario AI and record damage dealt in each game. The results are summarized in Table 2. Margins of error are 95% confidence intervals assuming Gaussian data distributions. We note that the CNN Pikachu handily defeats a level 3 CPU, is just barely outside the margin of error for being better than a level 6 CPU, and is quite competitive with the level 9 CPU (they are statistically tied). One important point

Mario CPU Level	Average Damage Dealt by Pikachu per Game	Average Damage Dealt by Mario per Game
3	$116.6\% \pm 29.3\%$	$66.4\% \pm 12.6\%$
6	$96.4\% \pm 13.8\%$	$80.5\% \pm 18.7\%$
9	$77.2\% \pm 18.3\%$	$71.6\% \pm 24.7\%$

Table 2: CNN Pikachu Performance Against Mario AI

to note is that our training data is generated through matches against a level 9 CPU opponent. Thus, all other factors held constant, we expect our CNN Pikachu to be able to adapt to the style of a Level 9 CPU opponent most readily, which is why our performance against lower-level CPU AI may be

deflated.

We further find in live simulations that our CNN Pikachu successfully learned relative character orientation and tracking. That is, the behavior of our CNN is sensitive to what direction the Mario character model is relative to Pikachu, and Pikachu is inclined to move towards Mario. This is illustrated in Figure 10. We see that Mario's relative position to Pikachu causes strong class predictions when Mario is (a) to Pikachu's left, (b) to Pikachu's right, and (c) above Pikachu, respectively. In case (a), we see a strong class score for left movement and left jump. In case (b), we see a strong class score for right movement. In case (c), we see a strong class score for the down+B attack move, which as described in section 5.2 is an attack that affects the space above Pikachu. All three of these strong class scores are consistent with conventional gameplaying wisdom, and demonstrate our CNN Pikachu can successfully track his opponent's position on the game screen.

This analysis is of course a bit disingenuous, as our algorithm takes as input four frames in temporal sequence, not just one (as we explored in section 5.2). However, even if it is slight oversimplification of the real situation, it illustrates nicely that our filters are very sensitive to relative spatial positioning, drawing a contrast to pure image classification tasks like those in ImageNet which should be relatively insensitive to orientation shifts in its images.

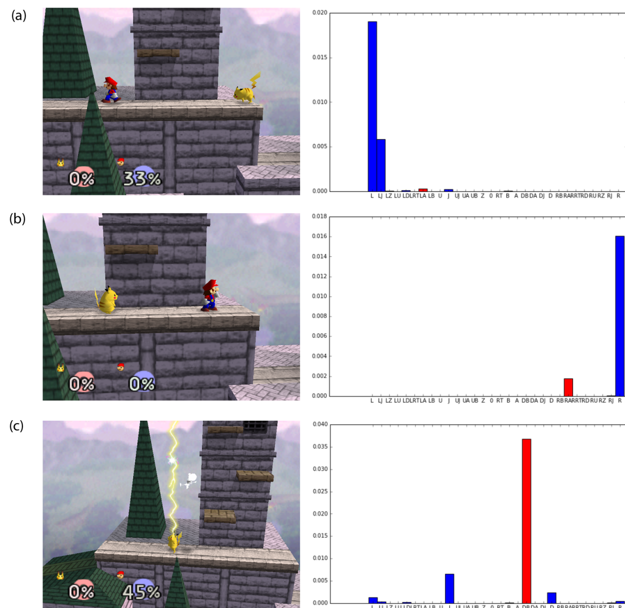


Figure 10: **Class Score Visualization** Game footage (left) is juxtaposed with class scores (right) at that moment in time. These frames were chosen because they exhibit strong class scores in one category, namely (a) move left, (b) move right, (c) down+B attack.

5.4. Performance on Mario Tennis

We also trained the same model on a Mario Tennis dataset generated in the same fashion. The model yielded (as shown in table 1) top-1, top-3, and top-5 validation accuracies of 75.2%, 96.0%, and 99.5%, respectively, numbers comparable to our accuracies for the Super Smash Bros model. The training also proceeded for 2 epochs, but with a dataset four times as small the training was completed within a day. We find that the CNN controlled Yoshi (see figure 11) can track the ball successfully and exhibits behavior that was clearly learned from our gameplay dataset. The live gameplay performance is admittedly not as competent when compared to our CNN Pikachu, but this is also reasonable given that the training data was generated by a novice for this particular game. There are also potential improvements to our model that can benefit the tracking abilities of our CNN, discussed in Section 6.1.

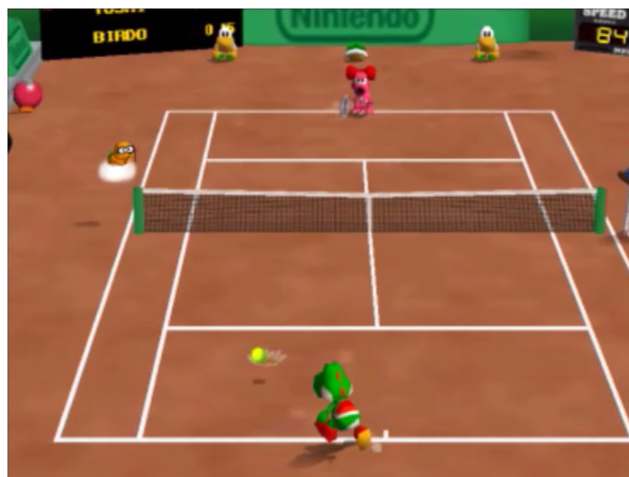


Figure 11: **Automatic Play of Mario Tennis.** Yoshi tracks and runs towards the ball in Mario Tennis live gameplay.

5.5. Model Strength and Weaknesses

We have already shown that our model can play in real time and effectively against difficult CPU AI. It does so without any knowledge of game-objectives, and can work across titles with very different styles of gameplay. And it does so in a supervised learning setting, which allows our models to train faster on fewer data compared to deep-Q learning. However, there are a few limitations to this model:

- **Long-Term Memory** Because we have hard-coded the temporal separation between subsequent frames input into our model (i.e. $\frac{1}{6}$s), the temporal information our model has is limited to $\frac{1}{2}$s into the past. Temporal information beyond that might be important, especially in games like Mario Tennis where consistent tracking of the ball over the entire game is necessary.

- **Locality** Because we only have view of a part of the map at any instant in time, our algorithm has difficulty extrapolating from the local view of the map to a global understanding of character position. In Super Smash Bros, this limitation manifests itself in Pikachu struggling when near an edge of the map; he sometimes prioritizes attacking Mario even if it means falling to his death. This issue is easy to avoid with some basic access to the game backend which provides the global information we need, and so ultimately is not a major limitation of our AI.
- **Data Limitations** Our training data consists of only matches against level 9 CPU. Performance degrades when the CNN encounters situations not well-represented in this training data set. One fix for this issue is to crowd-source game data from a variety of players with different play-styles. We expect such additional data would greatly help the robustness of our model's play.
- **Upper Learning Limit** Because our model only trains on a fixed set of training data, it cannot learn to be better than the players who generated that data. This is, of course, where deep Q learning is much more effective. To allow the algorithm to evolve, we must go to a reinforcement learning framework, but this is not easy to do with such a complex state/action space.

5.6. Video Demonstrations

You can view a full Youtube playlist of some videos relating to this project at:
`https://www.youtube.com/watch?v=c-6XcmM-MSk&list=PLegUCwsQzmnUpPwVv8ygMa19zNnDgJ6OC`

In total, there are 6 videos in the play list:

1. Super Smash Bros. AI Demo 1
2. Super Smash Bros. AI Demo 2
3. Super Smash Bros. AI Demo 3
4. Mario Tennis AI Demo
5. Super Smash Bros. Data Acquisition Time Lapse
6. Super Smash Bros. Single Frame AI Demo

6. Future Directions

6.1. Sequence Modeling with LSTMs

To solve the long-term memory issue discussed in Section 5.5, we can force the algorithm to be sensitive to temporal sequences by attaching an LSTM to the end of our CNN architecture. This would mean removing the 4-input-frame late-integration architecture in favor of a single CNN branch attached in series with an LSTM. We expect this to improve gameplay on Mario Tennis significantly, where we need constant tracking of an object (the tennis ball) and thus a longer memory that is always active.

6.2. Bias Learning with Statistical Methods

Our test-time tweaking (see Section 4.5) gives us 30 more hyperparameters to manually tweak during live simulations, but there are ways to set these automatically. For example, we can record statistics on test-time class scores and adjust the biases b_c so that the expression of each class is equalized. This will adjust class expression automatically to compensate for our lopsided training data set.

7. Conclusions

We have shown that when re-framing an AI reinforcement learning problem in terms of a simpler supervised learning framework, we can achieve good results on complex games with much less training time and training data. For Super Smash Bros in particular, we achieve AI behavior that can defeat the most advanced CPU AI in the game within only two epochs of model training. The low-bias, end-to-end nature of this framework also makes it attractive from a portability perspective. The disadvantage to this approach is that we rely wholly on our training data and our algorithm cannot learn for itself.

There are also many modifications that can improve performance of our model beyond our current model - LSTMs for temporal sequencing and statistical analysis to reduce our hyperparameter search space, for example - so the current model is just a baseline that already is fairly convincing of the fact that supervised learning models can offer a quick and easy alternative to deep-Q learning, especially when the complexity of the games in question continue to skyrocket.

8. References

[1] Mnih, V., *et. al.* (2014). "Playing Atari with Deep Reinforcement Learning." *DeepMind Technologies*

[2] Karpathy, A., *et. al.* (2014) "Large-scale Video Classification with Convolutional Neural Networks." *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

[3] Project 64: Nintendo 64 Emulator. "http://pj64-emu.com/"

[4] Fraps: Real-time Video Capture and Benchmarking. "http://www.fraps.com/download.php"

[5] Simonyan, K., *et. al.* (2014). "Deep Inside Convolutional Networks: Visualizing Image Classification Models and Saliency Maps." *Computer Vision.*

[6] Krizhevsky, A., *et. al.* (2012) "ImageNet Classification with Deep Convolutional Neural Networks." *NIPS.*

[7] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley and Y. Bengio. "Theano: new features and speed improvements". NIPS 2012 deep learning workshop.

[8] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley and Y. Bengio. "Theano: A CPU and GPU Math Expression Compiler". *Proceedings of the Python for Scientific Computing Conference (SciPy) 2010. June 30 - July 3, Austin, TX*

[9] Python Lasagne Wrapper for Theano Deep Learning Package. Lasagne v.0.2. https://github.com/Lasagne/Lasagne

[10] Ji, S., *et. al.* (2010) "3D Convolutional Neural Networks for Human Action Recognition."

[11] Ng, J., *et. al.* (2015) "Beyond Short Snippets: Deep Networks for Video Classification." *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*