Slippify: Parsing Super Smash Bros. Melee Frames

William Hu
Stanford University
willhu@stanford.edu

Samuel Do
Stanford University
samdo@stanford.edu

Matthew George Lee
Stanford University
mattglee@stanford.edu

Abstract

We use convolutional neural networks (CNNs) and deep spatial autoencoders to convert visual gameplay from Super Smash Bros. Melee to a state representation including observations and player actions. We will consider some of the key characteristics and challenges to parsing Melee frames, evaluate data augmentation techniques, and extend our existing model architecture to temporal data. Our experiments demonstrate that larger CNN architectures do better than simpler models and highlight challenges in capturing spatial information with spatial autoencoders.

1. Introduction

Games have proven to be a good means to evaluate the capabilities of models, motivating a number of research efforts from highly specific models that optimize for mastering a single game [6] to ones that promise generality [4]. To achieve generality, an input format that is uniform across a wide range of games is needed, and vision promises to fulfill this role. We want to make visual data more accessible to training models that play games.

In Super Smash Bros. Melee, SLP is a standardized data format for encapsulating the entire game state throughout the course of a game, including information such as player positions, actions, and hit points. These SLP files have been the basis of a number of projects that aim to leverage this data to train a bot with human-like performance [3]. However, collecting this data often requires players to not only play through the slippi emulator, but also enable SLP recording. Melee games that were played on the console or without the SLP recording feature enabled have previously been out of reach in a SLP-based machine learning system. Our project aims to remove this limitation by exploring a number of techniques (CNNs, data augmentation, and AutoEncoders) to convert a Melee replay (in the form of a sequence of frame-by-frame images) into a compact state representation of (observation, action) pairs. Specifically, we do the following:

1. Introducing a baseline CNN model for predicting state representations from gameplay frames.
2. Evaluating color jitter as a data augmentation method.
3. Investigating larger and deeper CNN architectures (ResNet-like structures).
4. Exploring unsupervised learning via deep spatial autoencoders.
5. Extending the framework to encode actions i.e. player controller input.

1.1. Why Melee?

Our choice of Melee was a bit arbitrary, but a few factors definitely made the choice easier. Most significantly, there is an existing emulator (Slippi) and a gameplay replay data format (SLP) that is supported by the emulator. This allows us to, with a SLP file of a game, reconstruct gameplay frame-by-frame, lending itself naturally to an abundant source of data pairs such a system can be trained on.

2. Related Works

2.1. Deep Residual Learning for Image Recognition

He et al. introduced residual connections, an important architectural component that has enabled the creation of deeper neural networks [5]. Through these skip connections, larger models have the ability to represent the identity function and do at least as well as their smaller counterparts. We base the model used in one of our ablation studies on the ResNet architecture presented in the paper.

2.2. Deep Spatial Autoencoders for Visuomotor Learning

The task of learning good state representations exists outside of game-play agents (i.e. robotics). Finn et al. [1] propose the use of deep spatial autoencoders, an autoencoder architecture that uses a spatial softmax after the convolution layers to reinforce learning spatial information (i.e. *where* instead of *what*). Rather than a flat vector of activations, this architecture applies a softmax over the spatial

dimension (W * H) for each channel. It then computes the expected 2D position of each softmax distribution and uses it as the low-dimension state representation. We apply a similar architecture to learning the Cartesian coordinates of Players 1 and 2 given a Melee frame in an unsupervised manner and contrast this approach to others.

2.3. Other work in playing Melee

Firoiu et al. [2] tackled playing Melee in the pre-slippi era and, as such, approached it using raw pixel inputs. They developed environment wrappers around the Slippi emulator, collected data via expert replays, and trained standard convolutional neural networks for policies using direct RL algorithms (such as PPO). Their work demonstrated the feasibility of learning robust visual-based policies for Melee but highlighted significant challenges in sample efficiency and frame prediction accuracy. Their work highlights the difficulties of applying RL and CV to Melee. In the post-slipppi era where we now have a concise representation of game state in the form of a SLP file, a mature emulator in the form of Slippi, and a rich dataset of Melee replays, we are able to decouple the problem into two: (1) drawing observations from a Melee frame and (2) acting on those observations to emulate human-level performance in playing Melee. The focus of this project is in understanding the characteristics of and make an attempt at solving (1).

3. Dataset

3.1. Manually Generated Data

When Melee is played on a certain emulator known as Slippi Dolphin, completed matches are automatically saved as `.slp` (Slippi) files for personal replay and review. These files contain game state data, frame-by-frame player inputs, etc. without storing the video of the match. This is an efficient representation of data for both storage and machine learning, but our goal is not to learn from the raw game state data.

We want to use computer vision to learn from the frames (images) of the match, so we convert the `.slp` file to a `.mp4` file using Slippipedia, which uses Playback Dolphin to re-emulate the match as a video. The command-line utility `ffmpeg` then allows us to slice the `.mp4` file into individual frames which can be fed into our computer vision model.

We mostly rely on an existing dataset that is mentioned in the next section, but having this data pipeline allows us to convert the dataset into the representation we require. Additionally, if we need specific training examples, we can handcraft precise situations using the manual pipeline - though this is likely not that practical or useful.

3.2. Existing Dataset

Luckily, there exists a very active online community surrounding machine learning for Melee. Because of this, there is a large curated dataset[7] of `.slp` files that we can use out-of-the-box with the following properties:

- "95,102 SLP files."
- "Unzips to 200GB."
- "All tournament sets, with varying skill levels."
- "Pruned to remove handwarmers, doubles, less than 30 second matches."
- "CC0 Licensed, so use it however you want"

This dataset helps with the bulk of the model learning. Unfortunately, the existing infrastructure to dump frames from `.slp` files requires simulating the entire match in real-time, so we only retrieved the frames from a couple replays. In each replay, there are over 10 thousand frames, so this was sufficient for our purposes.

The Google Drive link to the dataset is here.

3.3. Data Preprocessing

The SLP files are their corresponding Melee frames need to be preprocessed accordingly depending on the downstream task and methodology used. Firstly, observation and action data need to be normalized to have zero mean and unit norm. This is done by taking the global mean and norm of the entire dataset and subtracting each point by the corresponding global mean and dividing it by the global norm with some small ϵ to prevent division by zero. This is important for ensuring that all targets are weighted equally during training.

The frames themselves are resized to the expected input shape each method requires (i.e. the deep spatial autoencoder expects 240x240 images whereas the CNN expects 64x64 images). The pixel channels are then normalized between [-1, 1].

An individual frame is paired up with its corresponding entry in the SLP file and consecutive chunks of them are shuffled in the data loader for training. In the context of learning the observation state (i.e. Player 1 X, Player 1 Y, etc), these chunks are used to black out portions of the data so that frames neighboring test data are not included inside the training data (we pick a chunk size of 5 by qualitatively inspecting the replays). In the context of learning the action state, this same logic is used to specify a window of frames from which the model learns the action taken from the middle frame. We believe this context is essential to inferring actions, and the window size is later used as a hyperparameter we sweep over.

4. Methodology

4.1. Input/Output

To formalize the problem, we want the model to take, as input, a frame of game-play and output an appropriate state representation in the format of (Player 1 X, Player 1 Y, Player 1 Percent, Player 1 Facing, Player 1 Action, Player 2 X, ...). Notably, we break down the task into two subtasks:

- **Predicting observations**: this task is invariant to the ordering of frames in a replay. Indeed, one would expect the model to be able to accurately predict the positions, percentages, and other similar information from a single frame without looking at the surrounding frames. All the information needed is contained within a frame.

- **Predicting actions**: this task is contingent on being able to understand an entire window of frames. The actions that players take at any given frame should be inferred from the game-state before and after that frame. As a result, the action-prediction model expects, as input, a window of frames from which it predicts the action taken at the middle frame.

4.2. Supervised Observation Prediction

To our knowledge, there are no existing baselines or projects for converting Melee game-play to SLP. Due to how the view of any particular frame changes relative to player positions, there is no easy way to apply object detection methods that rely on a fixed camera view. As a result, we trained a simple model based on the following methodology and architecture to target Melee with the hypothesis that the model may be able to learn absolute positions of players relative to distinctive features of the stage or background.

- An **Encoder** that takes a RGB frame o as input and outputs a vector $\tilde{z}$ with information that encapsulates the game-state at that frame. This is a 4-layer CNN with layers consisting of a convolutional layer followed by a ReLU activation and finally a dropout layer. The output of the last convolutional layer is fed through a fully-connected linear layer to produce the final state representation $\tilde{z}$.

- The **Loss** is computed as the MSE loss between the predicted state representation $\tilde{z}$ and the target state representation z that represents that frame in the SLP file.

- The **Optimizer** used is Adam and learning proceeds using stochastic gradient descent over a configurable batch size and learning rate.

4.3. Data Augmentation

Data augmentation techniques such as color jitter and random crop are commonly used as a form of regularization by presenting the model with a greater diversity of data during training. However, due to the importance of spatial information in our context, we do not use data augmentation techniques such as horizontal flip or random crop since it 1) changes the absolute position of players and 2) may crop out either player entirely. We instead opt to simply apply random color jitter using the `torchvision.transforms` library and specify the jitter with `brightness` $= 0.2$, `contrast = 0.2`, `saturation = 0.2`, and `hue` $= 0.05$. This data augmentation is only applied during training. The validation and test datasets are not affected. The transformed data is added on top of the original training dataset and, as a result, we reduce the number of epochs during training by half, ensuring that all models train for the same amount of time.

4.4. A Larger Model

In addition to the base 4-layer CNN model, we experiment training with a larger model. It mimics the ResNet architecture, using residual connections between layers whose input and output shapes match [5]. This stabilizes training in deeper architectures that employ a larger number of layers.

4.5. Unsupervised Learning using Deep Spatial Autoencoders

Having established some measure of a baseline on the task itself, we wanted to explore how we might tackle the task in a context where labeled data might not be readily available. To do so, we apply the deep spatial autoencoder architecture introduced by Finn et al. [1]. We would like to note that, in using a deep spatial autoencoder, we are limiting the task of constructing a state representation of the frame to only the relative player positions with respect to the current frame (rather than including other information such as player percentages, orientations, and actions). This limitation is a characteristic of this specific methodology, and we leave a more general autoencoder for encoding Melee frames as future work.

- An **Encoder** that takes a RGB frame o as input and outputs a vector $\tilde{c}$ with expected 2D positions. This is accomplished by applying a spatial softmax over each of the output channels of the last convolutional layer to attain a per-channel probability distribution. This probability distribution is then used to compute the expected Cartesian coordinate for each channel. The final output is a vector of shape (B, 2C) where B represents the batch size and C the number of output channels of the last convolutional layer.

- A **Decoder** that takes the above vector of coordinates as input and reconstructs the image. This is a simple fully-connected linear layer (which Finn et al. argues is sufficient to produce a suitable feature representation) [1].
- The **Loss** is computed as the reconstruction loss between a down-sampled and grayscaled version of the input image $I_{downsample_{k,t}}$ and the output of the Decoder $dec(f_{k,t})$ where k refers to the replay and t refers to the image in that replay.

  $L = \frac{1}{t*k} \sum_{t,k} ||I_{downsample_{k,t}} - dec(f_{k,t})||_2^2$

4.6. Supervised Action Prediction

We extended our base formalization by supporting another novel encoding task: player action prediction. In the original formalization in Section 4.2, we encoded each frame of game-play into a game state representation known as **observations**. It contains information about what is visible in the frame, such as character positions, character damage, etc.

Predicting player's controller input, or **actions**, cannot be derived from a single frame because the effects of actions cross frame boundaries. For example, consider a player using the controller to walk right. From one frame, we can see that the player may be facing right, but it is impossible to know a player is moving right unless we have at least two frames.

Thus, predicting actions presents a new challenge in learning over the temporal axis. Instead of learning a frame to observation encoding, we learn a frame window to action encoding, where the frame window has a length of W frames. This is a hyperparameter that we can tune, and we will show the performance of different window sizes in Section 5.

Model architecture. We use the encoder architecture described in Section 4.2 to test the application of CNNs to the temporal domain. We do not use autoregressive models such as RNNs, because we make the assumption that actions are temporally local and do not require a long context window into the past.

To feed input into this architecture, we concatenate all frames in the window along their width and use the following two architecture ablations.

1. **Allow model growth.** We avoid adding layers, and the 4-layer CNN is not changed. This means that the shape of the last CNN layer output is kept at (C, 4, $4 \cdot W$). Because the linear layer is the last layer left, its input dimension must be multiplied by W since the input of the model is W concatenated frames. This dramatically increases the number of parameters; each additional frame in the window would cause the linear layer to increase its input channels by $C \cdot 4 \cdot 4$.

   One intuition for this architecture is that as the frame window increases in size, the model size should also greatly increase to capture longer patterns across the window. One argument against it is that although the input size is larger, we are at more risk of overfitting because the dataset size does not change. We feed overlapping windows of the original dataset as different samples for data efficiency.

2. **Minimize model growth.** We add one more layer to the CNN, so that the output of the CNN has shape (C, 1, W). This means that the model size would not grow as quickly for a larger frame window as in the last variation. We cannot entirely remove growth in the temporal dimension, because the order of frames matter. We assume that any model the applies pooling over the temporal dimension would not perform well; a confirmation of this would be future work.

   This variation allows us to see whether model growth is truly necessary with a larger frame window.

5. Experiments

5.1. Evaluation

Quantitatively, we evaluate our model based on the loss taken between the predicted observation (or action) $\tilde{z}$ and the ground truth z. As mentioned previously, data is normalized to have unit variance and zero mean so loss is in fact a measure of relative error.

For the deep spatial autoencoder experiment, this comparison breaks down so we qualitatively examine the coordinates output by the encoder as well as the reconstructed image from the decoder.

5.2. Parsing Observations Supervised

`default_small` is our constructed baseline - a small 4-layer CNN trained on the default dataset. `jitter_small` is the result of training the same model over the collected default and jittered data. Runs with the suffix `big` uses the larger CNN architecture with residual connections.

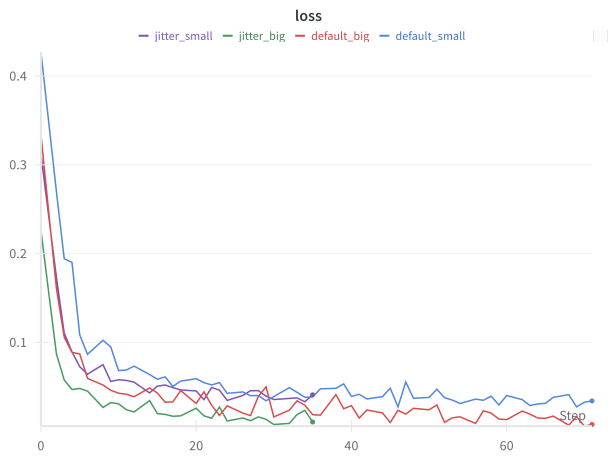

Figure 1. Loss curves for training a small/large CNN on jittered/default data.

Test Loss	Small	Large
Default	0.11592	**0.08699**
Jitter	0.11747	0.10390

Table 1. Comparison of test loss across size and augmentation settings.

As can be seen on Figure 1 and Table 1, the larger model consistently outperforms the smaller model. Furthermore, training on the combined dataset of default and jittered data does not improve test loss over just training on the default data. As mentioned in Section 3, the test and training datasets are derived from the same game replays with windows of frames being blocked out to ensure that an extremely similar frame (i.e. the one right before or after) to test data finds its way into the training dataset. However, this may not be enough to see the benefits of training on jittered data; the test data may need to be drawn from a game with a completely different background or stage. Instead, the larger model is able to properly overfit onto the default dataset and, as a result, perform the best compared to the other configurations.

5.3. Parsing Observations Unsupervised

Training the deep spatial autoencoder for this task proved to be difficult and yielded a few interesting results. At the beginning, the reconstructed image the decoder outputs is just noise.

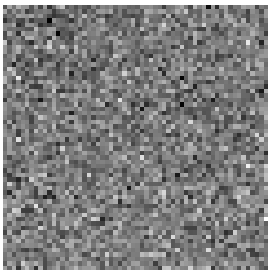

Figure 2. Reconstructed image at Epoch 0

Figure 3. Down-sampled image

Towards the end of training, however, we see some vague features appearing.

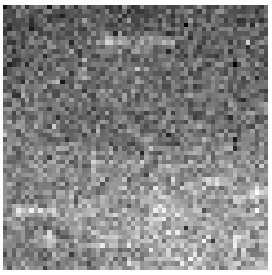

Figure 4. Reconstructed image after training

Figure 5. Down-sampled image

However, instead of being able to capture fine details such as the characters, the model instead focused on stable features across all the input frames such as the game clock (situated at the top of the frame in the middle) and player percentages (situated at the bottom left and right corners). We further visualized the output of the encoder layer by plotting the coordinates on the original image.

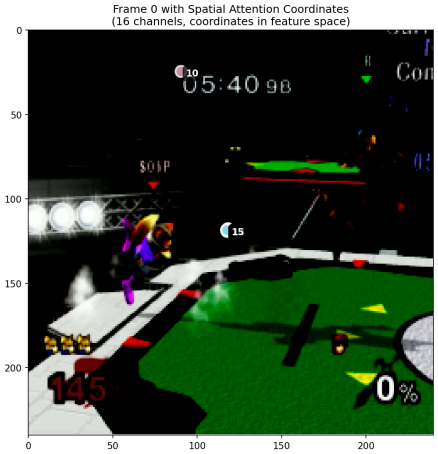

Figure 6. Spatial softmax coordinates plotted over the original image. There are a total of 16 points on this image, but most of them are being blocked by others due to the high overlap in coordinate positions among them. Note the blue and pink points labelled 15 and 10 respectively.

As seen in Figure 6, the coordinates congregate on two points: the game clock and the center of the image. Finn et al. proposed this architecture while targeting images that had fairly uniform backgrounds (i.e. solid color or slightly textured) with only a few objects [1]. As a result, we may need a larger model for complex scenes such as the ones in Melee. Going forward, we would also want to experiment with different data augmentation techniques that occlude the background and any features that might distract from the player positions. Another means to better differentiate the players themselves is increasing the down-sampled image resolution from 60x60.

5.4. Action Prediction

5.4.1 Basic Hyperparameter Sweep

We first ran a hyperparameter sweep over learning rate, batch size, and dropout. We did this first to make some basic hyperparameter choices and fix those in place to avoid a Cartesian explosion when picking values for interesting hyperparameters (i.e. W, the frame window length), due to training constraints. We chose `batch_size=64`, `dropout=0.1`, and `learning_rate=0.001` according to the validation loss graph in Figure 7.

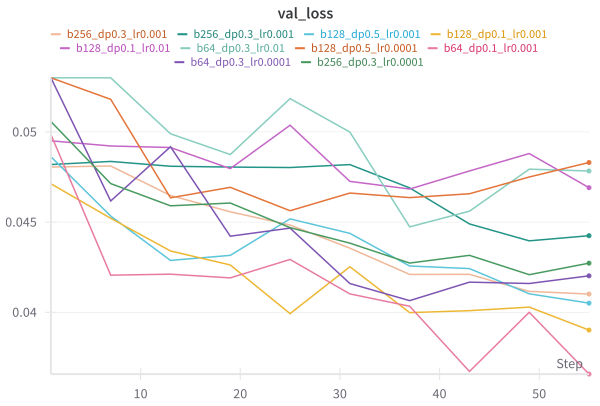

Figure 7. Action encoder basic hyperparameter sweep. Run `bA_dpB_lrC` refers to batch size A, dropout B, and learning rate C.

5.4.2 Frame Window Length Selection

We swept 5 frame window lengths: 1, 3, 5, 7, and 9. These are odd lengths to simplify indexing and have a central frame that we can pick to select the action target. Below is the reasoning for each:

- **1-frame windows.** This is primarily a sanity check of our explanation for why multiple frames are needed to predict actions. We need to see runs with `window_len=1` perform worse than one or more of 3, 5, 7, 9.
- **3,5-frame windows.** We should see an improvement over 1-frame windows due to having the context of multiple frames passed to the model.
- **7,9-frame windows.** We are curious to see if these window length values do better due to more context, or worse due to too much noise from frames that are far away from the actual action.

5.4.3 Model A Results: Allow Model Growth

For the model architecture that allows model growth, we have the train loss over epochs in Figure 8 and final test loss in Table 2.

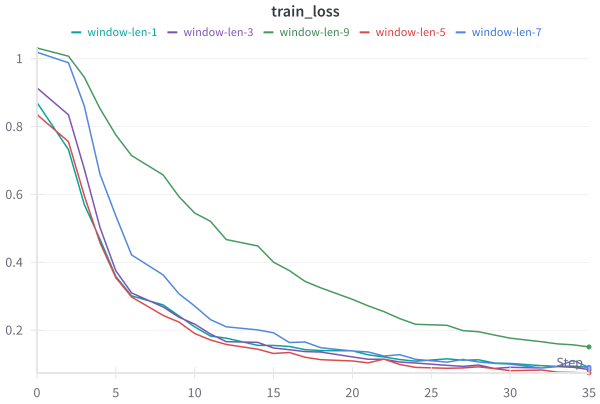

Figure 8. Action encoder with allow model growth: train loss over epochs

5.4.4 Model B Results: Minimize Model Growth

For the model architecture that minimizes model growth, we have the train loss over epochs in Figure 9 and the final test loss in Table 2.

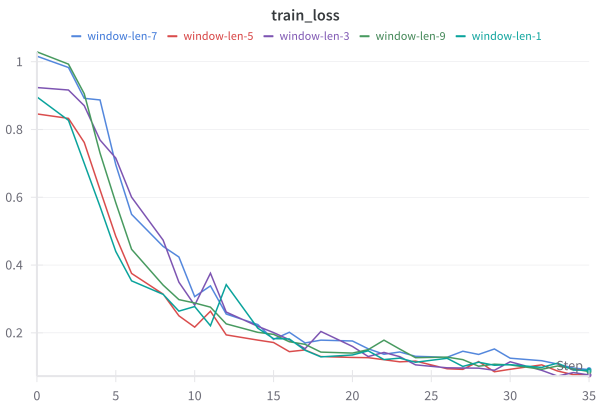

Figure 9. Action encoder with minimum model growth: train loss over epochs

Final Test Loss		
Window Length	**Model A**	**Model B**
1	0.79665	0.77700
3	**0.60027**	**0.56931**
5	0.95704	0.90058
7	0.74687	0.66354
9	0.78414	0.73127

Table 2. Comparison of model growth and minimized growth across different window lengths.

5.4.5 Analysis

We will refer to the architecture that allows for model growth as Model A and the architecture that minimizes model growth as Model B.

All runs train well. The training loss curves between the two architectures have one key difference - larger window lengths in Model A result in larger training losses. This is expected because Model A allows for more parameters with increased window length, so it makes sense that Model A is more difficult to train for a larger window length. We see that the training loss curves for Model B are unremarkable, which is confirmation that the model size is not growing as much with the window length.

The final test loss across both architecture types confirm our strongest hypotheses about window length. Namely, we see that only having one frame (i.e. window length of 1) results in significantly worse performance than having some additional context (window length greater than 1). On the other end of the spectrum, we see that having too much context (window lengths of 5, 6, and 7) performs worse than having only the immediate frame context (window length of 3). Thus, we can be assured that with a window length of 3, the model is using the surrounding context to improve its prediction and using just enough to get better results. Any more context and it seems that the input is too noisy for the model to generalize effectively.

Comparing the two architectures Model A and Model B, we see that the final test loss patterns (i.e. the ranking of window lengths) do not differ significantly between each other. However, the actual losses do differ between Model A and Model B. Consider both architecture types for a window length of 3. Both models trained to similar train losses, but we see that the Model B type does significantly better despite having many less parameters than the Model A type. This is a surprising result: we do not seem to need more parameters for a larger temporal sequence, and it might even be detrimental. However, one alternate idea to consider is that the temporal axis span of the minimum data to predict correctly may be short enough that having more parameters would be cumbersome. In that case, it would be interesting to explore a task that required longer temporal axis span and testing Model A types and Model B types with shorter window lengths inside of that span. Unfortunately, since Melee games happen quickly, the discretized and feasible window lengths are not sufficient to test this theory.

One open question is why a window length of 5 showed much higher loss than even larger context windows such as window lengths of 7 and 9. One possible theory for this result is that a window length of 5 is barely too much data such that the model can memorize the spurious patterns of the training windows, but with window lengths of 7 and 9, the model must actually procure the correct signals for action prediction since there is too much data to memorize.

6. Conclusion

Parsing Melee frames to a condensed state representation continues to be a difficult task. Slippify is the initial effort to characterize this problem and apply a suite of computer vision techniques - supervised and unsupervised - to attempt to solve it. In particular, we note down the temporal nature of extracting action information from consecutive frames. Our experiments ultimately show some promise when regressing to a ground truth in the case of supervised learning and the difficulty of applying unsupervised techniques to cluttered frames.

Moving forward, we would like to gather a more diverse set of game replays to train with and augment frames to emphasize key features such as player positions. Furthermore, future work may include applying a 3D-CNN, where time is the third dimension to represent frame windows.

References

[1] C. Finn, X. Y. Tan, Y. Duan, T. Darrell, S. Levine, and P. Abbeel. Deep spatial autoencoders for visuomotor learning, 2016.

[2] V. Firoiu, W. F. Whitney, and J. B. Tenenbaum. Beating the world's best at super smash bros. with deep reinforcement learning. *CoRR*, abs/1702.06230, 2017.

[3] E. Gu. Hal: Training superhuman ai for super smash bros. melee. `https://github.com/ericyuegu/hal`, May 2025. GitHub repository. Accessed: 2025-05-23.

[4] D. Hafner, T. P. Lillicrap, M. Norouzi, and J. Ba. Mastering atari with discrete world models. *CoRR*, abs/2010.02193, 2020.

[5] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015.

[6] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[7] u/bananaman8367. Help me create a machine learning project with your replay files!, 2021. Reddit post on r/SSBM, accessed May 2025.