Object Oriented Prediction for ViZDoom

6.882: Embodied Intelligence
Final Project

Nick Stathas
nistath@mit.edu

I. Introduction

In 6.882 we covered many different ways to represent an environment's state and learn its dynamics lying broad along the dimension of supervision. Hand-crafted models such as chess engines have access to the simplest and most compressed state representation available, without the need to learn a transformation. On the other end of the spectrum, approaches such as video prediction are capable of deriving a joint representation of the environment statics and dynamics without explicit supervision on the learned representation.

Many real-world tasks are visual in nature, providing the agent with a frame buffer and expecting an action in return. In the world model [1] approach we see the use of a generative model to compress that frame-buffer into a lower-dimensional latent space, making the observation space easier to process. In the case of a VAE, this technique encodes a simple structural bias: no change to the environment is possible within the same frame. The object factorization method we see in O2P2 [2] provides even more biases. The model views each frame through a set of stateful objects whose state transitions over time. Additionally, it is possible that these object state transitions are not individual, but affected by interactions with the other objects. The key idea is the object-factorization bias in the world model. The object representation can be learned with a generative model, an autoencoder in O2P2, or provided by the environment, if available.

In this paper, we will explore modifications to the O2P2 approach that allow it to be deployed on more commonly used visual environments. The core difference is in the purpose of the prediction module. In O2P2, the prediction module is trained to predict the *steady-state* of the observed environment from the input configuration. The exact example is going from a potentially unstable stack of toy blocks to the result after it falls and settles. Common visual environments, including ViZDoom, take more than several frames to reach steady-state and may not always have a steady-state. Thus, we will use the prediction module to regress to the state representation of the next frame instead.

II. Environment

A. Features and Usage

The ViZDoom environment [3] is built on Doom, a first-person shooter supporting maps that pose various levels of difficulty. ViZDoom allows the developer to tap into the internals of the game's state. As such, the level of state observability can be tuned to the specifications of the designed task. In this paper we will use the frame buffer as well as its instance segmentation map for the objects visible in the scene. The segmentation map is one of the easier ways to obtain an object factorization from the ViZDoom frame.

In principle, the instance segmentation can be learned in a supervised manner by a CNN, should we want to deploy this in a model without access to the game-engine's segmentation. The model was only provided the segmented frame buffers and not the masks used to generate them.

B. Collecting Experience

There were several parameters in the decision of what types of experience to collect. The simple environment `basic.cfg` was chosen for its low number of object classes present and useful constraints imposed on the fundamental state of each object. The environment is a square room with only three walls visible. The player may only move left/right and always faces in a direction perpendicular to the wall behind. A cacodemon spawns on the other side of the room and may only move left/right as well. Both the player and cacodemon are allowed to shoot.

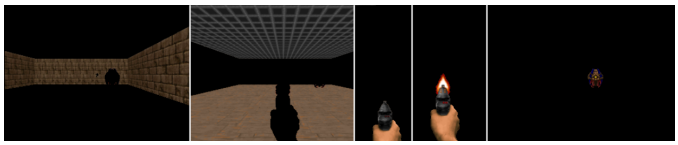


Fig. 1: Most common segmented sprites in `basic.cfg`

Because of the small set of possibilities in this environment, a random player agent was sufficient to capture many environment states and behaviors such as killing a demon or getting hit. Future work could use a pre-trained agent capable of exploring and forming a curriculum for more complicated environments.

We designed a custom serialization format using NumPy structured arrays to store the full game state. Each frame represents a row in this array and there is an array for every episode. Frame buffers and the corresponding instance segmentation maps were separately stored as png files. We collected 20 episodes of a random agent dataset on the aforementioned environment. Most episodes survived for 300 frames.

III. Autoencoder

A. Autoencoder Design

The development of the model started out as a simple autoencoder. The motivation for focusing on it first is the shifted focus from O2P2's visually simple environment to ViZDoom's complex and detailed object sprites. Small visual differences in the appearance of a sprite can reveal a lot about the object's states. For example, a gun about to fire first slightly translates up for one frame and shows its muzzle flash another frame later.

The images were collected at $(480, 640)$ resolution and were downsampled to $(240, 320)$ for training. This size was sufficient for a human player to succeed in the game and provided a simple initial dimensionality reduction. Care was taken to use nearest neighbor interpolation for downsampling so as to not affect the instance segmentation mask values.

1) Baseline: A baseline deep fully-connected autoencoder trained with mean squared error loss was only able to classify the floor/ceiling and gun sprites, producing zero images for the rest. The gun sprite was too faint, because the empty shadow on the floor sprite directly competed with the gun's body. A similar architecture with a convolutional encoder and a fully connected decoder yielded the same results, suggesting end-to-end convolutions are necessary.

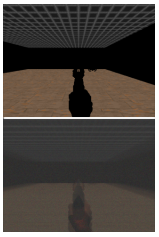


Fig. 2: Fully Connected Baseline Train Set Results

A baseline convolutional autoencoder trained with mean squared error loss was only capable of rendering static images of the three largest sprites in the dataset. Shadows of objects in front of the walls or muzzle flashes were not succesfully reproduced. Essentially a naive convolutional autoencoder was only able to classify three of the easiest objects, but not capture their state.

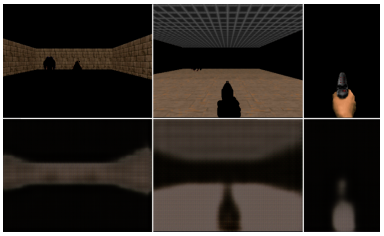
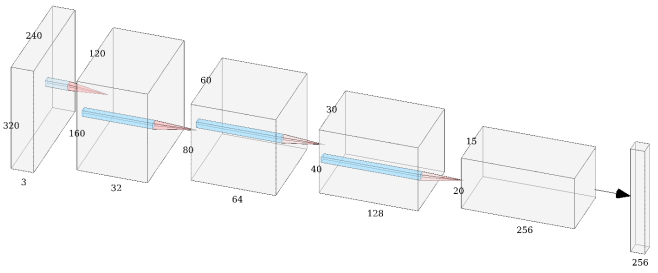
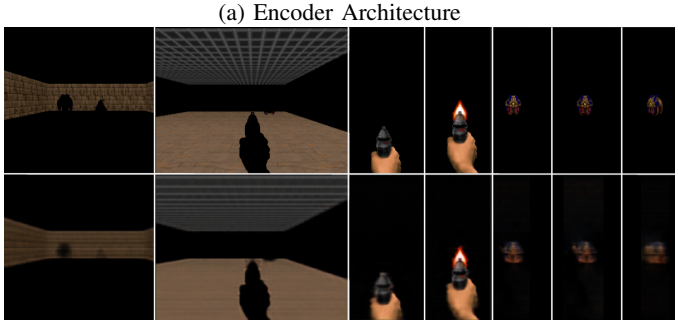


Fig. 3: Naive CNN Baseline Train Set Results

2) High-capacity CNN: An encoder following specifications similar to O2P2 [2] was designed, using filters that double up to to 256. A linear layer transforms the output block of the final convolutional layer down to the desired 256 dimension object-space. A symmetric decoder built with `ConvTranspose2d` operations and a slightly larger kernel size was used.



(a) Encoder Architecture



(b) Test Results — Top: Target, Bottom: Reconstruction

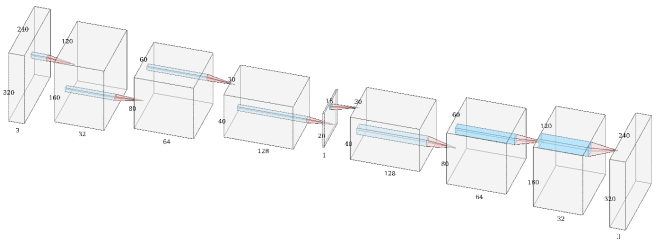
Fig. 4: High-capacity CNN Autoencoder

Through a qualitative analysis of test set performance, the high-capacity CNN was always observed to correctly classify the shown sprite. The biggest non-negligible downside was the robustness of reconstructing shadows of objects on the wall. This might be fixable by further tuning the loss function.

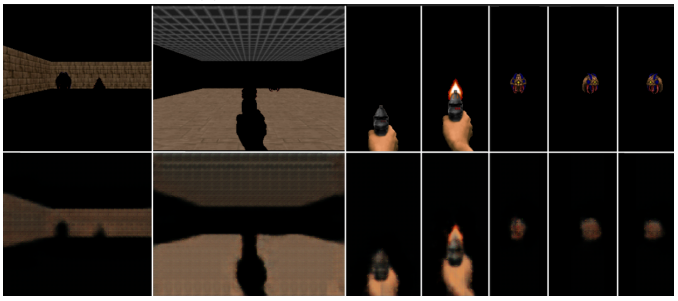
Although a small failure rate for reproducing shadows on the wall might not itself be important for downstream planning purposes, it hints at two problems. First, it is possible that the linear transformation from CNN feature space to object space and backwards reduced the robustness of shadow reconstruction. Thus it could be deepened to allow it to develop more margins around different shadow representations, but this would hurt generalization for unseen shadow positions. Second, failure to reconstruct shadows implies failure to reconstruct the silhouette of a large sprite. In a more complicated environment with large enemy sprites, their silhouette might lead to significantly different implications about their state.

3) Fully Convolutional: One approach to address the above problem was to remove the linear layers. This architecture was achieved by reducing the filter size of the encoder's output to 1. Combined with a depth of 4 and a stride of 2 at each level, a $2^4 = 16$ downsampling of the original $(240, 320)$ image produces a $(15, 20)$ image of 300 total dimensions. As a side-effect, the network's capacity was decreased, both by reducing the filter count and by removing the parameter-heavy linear layers. The training performance increased nearly 3 times.

Although significantly more blurry, the encoding is rich even if the decoder is unable to faithfully render it. Qualitative



(a) Autoencoder Architecture



(b) Test Results — Top: Target, Bottom: Reconstruction

Fig. 5: Fully Convolutional Autoencoder

analysis of validation set performance showed no significant information loss. Textures, particularly on the cacodemon were not as faithful due to the reduced model capacity. However, there were visible differences among the reconstructions of cacodemons exhibiting different behaviors. Thus, it is likely that the necessary information about the state is encoded but not rendered. This was tested by increasing the capacity of the decoder, which improved the qualitative fidelity of the results, but not to the same level of figure 4. The capacity of the encoder and/or the presence of linear layers remain important for visual fidelity.

B. Autoencoder Training Details

The above architectures do not work or exhibit degraded performance without the following details, ranked in rough order of importance:

1) Random Mini-Batches and Batch Normalization: Forming batches of random frames first and segmenting them after, produces a training order that is worse, on average, than randomly selecting pre-segmented sprites from the whole dataset. Along the same line, batch normalization [4] significantly helps in training and likely works better with batching by sprites instead of frames.

2) Masked Focus MSE Loss: Different sprites occupy a different amount of pixels. Using a simple MSE loss has two negative effects: it *i*. gives more weight to objects with more pixels and *ii*. incentivizes predicting all-zero in a manner inversely proportional to the sprite's size. Simultaneously, segmented sprites fundamentally exhibit two important overarching features. A sprite's *texture*, defined by the frame buffer pixels bound by the masked region, is important for classifying the object and partially observing its state through the pose it visually exhibits. A good *silhouette*, defined by how sharply the boundary of the autoencoded sprite is drawn, is important for encoding its precise spatial location. A proxy metric for the silhouette is the summed square error from zero for all the pixels outside the sprite's segmentation mask.

A custom loss function to replace MSE was designed to balance these two properties. The squared error for each pixel was weighted by w_{in} if it is inside the masked region and w_{out} if it is in the background. These two weights normalize the relative importance of the masked region and its background by area.

$$\begin{aligned} w_{\text{in}} &= \textsc{Area}(\texttt{background}) \\ w_{\text{out}} &= \textsc{Area}(\texttt{mask}) * \texttt{focus} \end{aligned} \tag{1}$$

The focus parameter is less than 1 when the texture is deemed more important and greater than 1 when the silhouette/background is to be prioritized.

3) Class Stratification: Different classes of sprites appear with different frequencies in the dataset. Walls, the floor, ceiling, and gun are always there. Cacodemons, bullet puffs, blood effects, and other sprites appear conditionally on player actions, fundamentally reducing their frequency.

4) Focus Annealing: Dynamically adjusting the focus as a function of the epoch helps provide sub-targets during training. The focus was varied with the following sequence $\{0.1,\ 0.5,\ 1,\ 2,\ 5,\ 1.5,\ 1,\ 1,\ 0.7\}$. The sequence consists of three horizontally-stacked regimes of three epochs each: prioritization of texture, prioritization of silhouette, and a final balancing region to undo the negative effects caused by either previous region.

5) Perceptual Similarity Loss: Perceptual similarity is helpful in achieving better texture quality. We use SqueezeNet [5] pre-trained as a LPIPS [6]. It aids in reconstructing the texture style, particularly for the cacodemon sprites. Because it was trained on real-world images, this loss was weighted at 10% relative to the masked focus MSE loss, to not detract from the importance of capturing the true texture.

C. Encoding Inspection

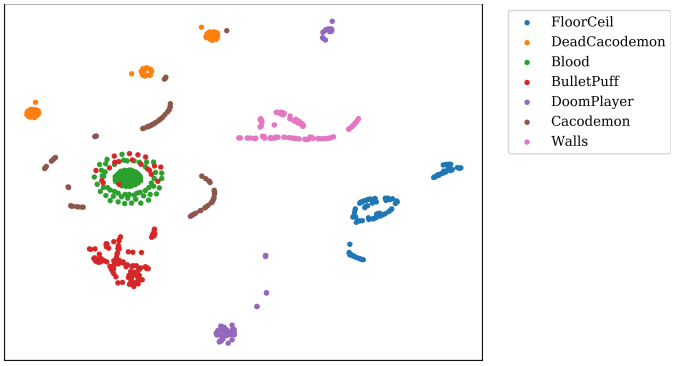
The resultant encoding in figure 6 exhibits sufficient separation among different classes. A dense middle section for the autoencoder further transforms the object encoding space to alleviate some of the discontinuities in the cacodemon clusters.

IV. Prediction

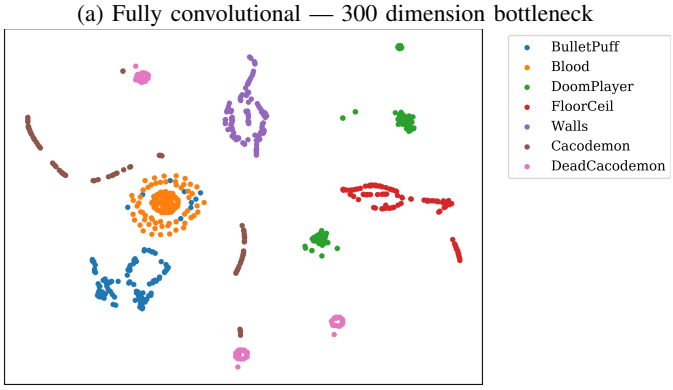
We carried out a limited exploration of prediction in the space of object-factorized encodings. The minimum viable product for prediction results in a simpler architecture than O2P2 [2].

A. Dataset Preparation

Descriptive dataset indices of the form *(episode number, frame number, object id, object information)* were implemented. With those, it becomes possible to know details about the object without loading it. The object information includes sufficient descriptors such that that, given two indices from consecutive frames, it is possible to answer whether they describe the same object.



(a) Fully convolutional — 300 dimension bottleneck



(b) Two linear layers — 128 dimension bottleneck

Fig. 6: t-SNE reduction of encodings labeled by class

For every pair of consecutive frames, correspondences between their indices were calculated using the above method. The result is a predominantly dense mapping from every index in the dataset to the index of the corresponding object in the next frame if such exists or `None` otherwise. This approach allows for the reuse of the class stratification and randomized mini-batch sampling strategy, which operate solely on indices, from the autoencoder.

B. Predictor Architecture

The prediction module receives the object factorized representation of the current frame and is tasked with predicting the representation of the next frame. A hyperbolic tangent activation function was added at the tail of the encoder, to constrain each of its dimensions. This was done to aid the predictor in regressing to valid predicted object encodings. Most importantly, the weights of the trained encoder and decoder were frozen in the interest of time. Preventing fine-tuning of the encoder retains its compatibility with the decoder, allowing us to visualize the results. We believe that a well-trained encoder can learn predictable object encodings. The end-to-end architecture of O2P2 [2] provides an additional structural bias toward noticing visual features that indicate a state transition, if the autoencoder would not manage to learn them alone.

1) State Transition: Interactions among objects are not observed in the `basic.cfg` dataset. Thus, it is sufficient to model the state transition with a feedforward neural network operating in the object encoding space of a single object.

a) Loss: Initially, a MSE loss between the input and target object encoding was described. It was able to get sufficiently close to preserve the class of the object and in many cases predicted a static state. When it did not get close enough, the decoder would produce a blank image. When the next state is ambiguous, the predictor would predict the resting position, the one most frequent in the dataset. Binary cross entropy loss after converting from the hyberbollic tangent representation to sigmoid, yielded quialitatively similar performance.

A visual reconstruction supervisory signal similar to O2P2's rendering module [2] was summed with the object encoding loss. The predicted object encoding was decoded and compared with masked focus MSE loss to the target segmented object sprite. Interestingly, in ambiguous cases the predictor coerced the *frozen* decoder to produce an image interpolating between the possible states.

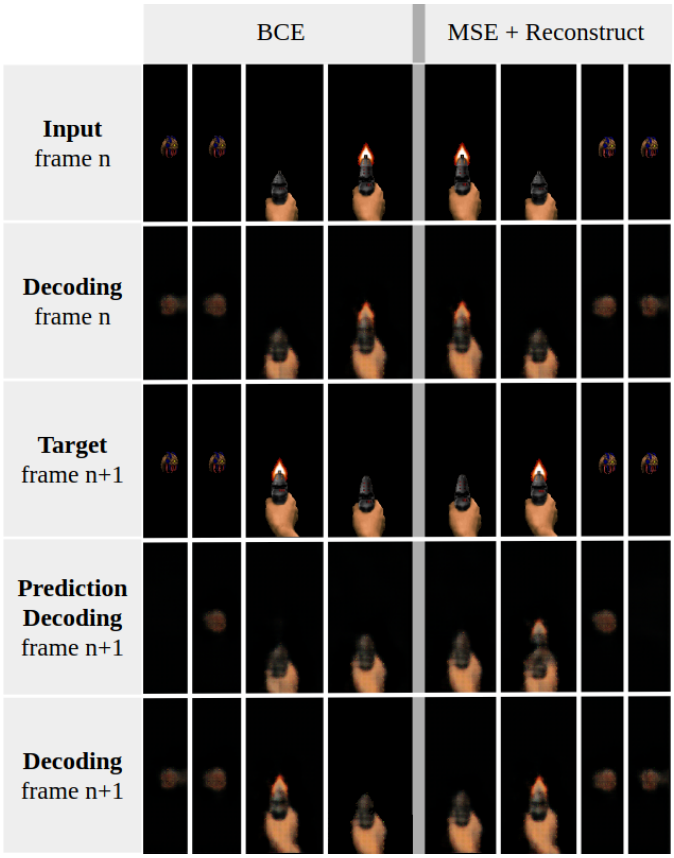


Fig. 7: Prediction Validation – Columns by loss function

b) Future Work: In environments with important interactions, it would be useful to explore the feature and input transformations introduced in PointNet [7]. By treating a frame's state as a point cloud of object encodings, it might be possible to learn a permutation invariant transformation between the point clouds of consecutive frames.

2) Object Permanence: Implementation of section IV-A naturally raises awareness to the possibility that objects may enter and exit the frame. Object entry was not tackled as it is hard to materialize a 300-dimensional object state without a very accurate prior and it is harder to do this for multiple object simultaneously. Object entry is present in the `basic.cfg` dataset as bullet puffs when the gun fires and blood when the player or cacodemon is hit. In its more complicated form, object entry can come by rotating the player's camera into view of a new object or that object itself emerging from behind cover.

Object disappearance is significantly easier to model. A binary classification problem to estimate the probability the object disappears is additionally posed to the predictor network and trained with binary cross entropy loss.

3) Resolving Ambiguities: The aforementioned ambiguous cases are understood. The intent of the prediction module is to learn the dynamics of the environment, analogous to the state transitions of a POMDP. The information presented to the predictor in the above experiments, effectively asks it to jointly learn the state transitions and the *actions* taken. Because the agent is random, that is not a fair construction. To resolve this first ambiguity, future work could provide the action taken before this frame to the predictor.

Another ambiguity arises about the velocity, intended motion, or progress along a fixed animation sequence of possibly predictable objects such as bullet puffs or cacodeomns. Introducing memory could help better solve each object's internal POMDP. A recurrent predictor could use its memory to keep track of several observations for each object. If the predictor is not permutation invariant, it might be fickle based order of presentation of the objects. To avoid that, another approach in case an object tracker is available, a fixed-length history of object encodings could be provided to the predictor. A natural object tracker might be a 1-NN among the object representations of consecutive frames.

V. Implementation

The models were implemented in PyTorch [8]. The project spans 801 source lines of code across 11 Python files.

References

[1] D. Ha and J. Schmidhuber, "Recurrent world models facilitate policy evolution," *CoRR*, vol. abs/1809.01999, 2018. [Online]. Available: http://arxiv.org/abs/1809.01999

[2] M. Janner, S. Levine, W. T. Freeman, J. B. Tenenbaum, C. Finn, and J. Wu, "Reasoning about physical interactions with object-oriented prediction and planning," *CoRR*, vol. abs/1812.10972, 2018. [Online]. Available: http://arxiv.org/abs/1812.10972

[3] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaskowski, "Vizdoom: A doom-based AI research platform for visual reinforcement learning," *CoRR*, vol. abs/1605.02097, 2016. [Online]. Available: http://arxiv.org/abs/1605.02097

[4] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[5] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016. [Online]. Available: http://arxiv.org/abs/1602.07360

[6] R. Zhang, P. Isola, A. A. Efros, E. Shechtman, and O. Wang, "The unreasonable effectiveness of deep features as a perceptual metric," in *CVPR*, 2018.

[7] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," *CoRR*, vol. abs/1612.00593, 2016. [Online]. Available: http://arxiv.org/abs/1612.00593

[8] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf