

# Advancing Numerical and Spatial Consistency for Generative Games – Appendix –

001

002

Anonymous ICCV submission

003

Paper ID 913

## Contents

004

A Introduction of the Attached Zip File

2

005

B Detailed Architecture of LogicNet

2

006

C Details of CNN in the Spatial Module

3

007

D Examine the Effectiveness of Map Construction for Traveler

3

008

E Constructing 2D Map for Pac-Man

3

009

F Details of Using LLMs for Creating Games

3

010

G Details of Models Provided by PGG

4

011

H Detailed Architecture of Valid Action Model and Valid Numerical Model

4

012

I. Visualizations of Failure Cases

8

013

## 014 A. Introduction of the Attached Zip File

015 The attached zip file includes training code, inference code, model, and dataset creation code. We also provide samples for  
016 training and inference on three different games.

## 017 B. Detailed Architecture of LogicNet

018 The following code shows the architecture of LogicNet, a lightweight neural network design is proposed. Using the Traveler  
019 game as an example, the model processes images with an input resolution of 96x96, resulting in a hidden state spatial  
020 dimension of 24x24. For games such as Pong and Pac-Man, which output resolutions of 128x128, modifications to the input  
021 dimensions and corresponding network architecture are necessary to accommodate these differences.

022 Furthermore, considering Pong as a two-player game, the model should be adapted to receive inputs from both players.  
023 This can be achieved by introducing an additional action embedding layer and appropriately adjusting the input dimension  
024 of the linear layers to integrate dual player inputs effectively. This adjustment ensures that the network architecture remains  
025 adaptable and responsive to the specific requirements of different game dynamics.

```
026 1 import torch
027 2 import torch.nn as nn
028 3
029 4 class LogicNet(nn.Module):
030 5     def __init__(self, num_embeddings, embedding_dim=64):
031 6         super(LogicNet, self).__init__()
032 7
033 8         # Image processing: input [batch, 32, 24, 24]
034 9         self.conv_layers = nn.Sequential(
035 10             nn.Conv2d(32, 32, kernel_size=3, padding=1),
036 11             nn.BatchNorm2d(32), # Add BatchNorm to improve stability
037 12             nn.ReLU(),
038 13             nn.MaxPool2d(2), # -> [batch, 32, 12, 12]
039 14
040 15             nn.Conv2d(32, 64, kernel_size=3, padding=1), # Increase number of channels
041 16             nn.BatchNorm2d(64),
042 17             nn.ReLU(),
043 18             nn.MaxPool2d(2), # -> [batch, 64, 6, 6]
044 19         )
045 20
046 21         # Integer embedding layer
047 22         self.embedding = nn.Embedding(num_embeddings, embedding_dim)
048 23
049 24         # Calculate flattened feature size: 64 * 6 * 6 = 2304
050 25         conv_output_size = 64 * 6 * 6
051 26
052 27         self.fusion_layer = nn.Sequential(
053 28             nn.Linear(conv_output_size + embedding_dim, 512),
054 29             nn.ReLU(),
055 30             nn.Dropout(0.5)
056 31         )
057 32
058 33         # Final classification layer
059 34         self.classifier = nn.Linear(512, 1) # Binary classification
060 35
061 36     def forward(self, image, action):
062 37         # Process image
063 38         img_features = self.conv_layers(image)
064 39         img_features = img_features.flatten(1) # [batch, 64 * 6 * 6]
065 40
066 41         # Process integer
067 42         action_features = self.embedding(action)
068 43
069 44         # Feature fusion
070 45         combined = torch.cat([img_features, action_features], dim=1)
071 46         fused_features = self.fusion_layer(combined)
072 47
073 48         # Classification
074 49         predictions = self.classifier(fused_features)
```

```

50         return predictions
51
52
53
54 if __name__ == "__main__":
55     # Create model with vocabulary size of 10
56     logicnet = LogicNet(10)
57
58     # Create random image tensor with correct shape [batch, channels, height, width]
59     image = torch.randn(1, 32, 24, 24)
60
61     # Create integer tensor with a valid index (between 0 and num_embeddings-1)
62     action = torch.tensor([5]).long() # Example using index 5
63
64     # Run forward pass
65     output = logicnet(image, action)
66     print("Output shape:", output.shape)
67     print("Output value:", output)

```

## C. Details of CNN in the Spatial Module

The CNN within the spatial module primarily comprises five convolutional layers and four downsampling max-pooling layers, which function to compress the spatial dimensions into map tokens for generation. The code is shown as follows:

```

1 self.cnn = nn.Sequential(
2     nn.Conv2d(3, 32, kernel_size=3, padding=1),
3     nn.ReLU(),
4     nn.MaxPool2d(2),
5     nn.Conv2d(32, 32, kernel_size=3, padding=1),
6     nn.ReLU(),
7     nn.MaxPool2d(2),
8     nn.Conv2d(32, 32, kernel_size=3, padding=1),
9     nn.ReLU(),
10    nn.MaxPool2d(2),
11    nn.Conv2d(32, 32, kernel_size=3, padding=1),
12    nn.ReLU(),
13    nn.MaxPool2d(2),
14    nn.Conv2d(32, 384, kernel_size=3, padding=1),
15 )

```

## D. Examine the Effectiveness of Map Construction for Traveler

We showcase some samples in Figure 1, where each pair represents the map of an episode in the dataset. For each pair, the upper map denotes the ground truth, which can be directly obtained from the Pygame program, while the bottom map is constructed using the sliding window algorithm. Our experiments on the evaluation dataset demonstrate the effectiveness of this method, with an average PSNR (Peak Signal-to-Noise Ratio) of 38.77. Additionally, it is worth noting that during the matching process, the black square are also included in the matching process, even though they have actually shifted to other locations. It is observed that our matching algorithm can tolerate the errors introduced by this displacement.

## E. Constructing 2D Map for Pac-Man

Due to the relatively complex visuals in Pacman, we simplify the process by extracting the blue components (within a specified range) from the images to serve as features for sliding window matching. This is the result after extracting the blue components, as shown in Figure ?? . We present the final constructed maps of two episodes in the Figure 3. Note that since the score area in the game may overlap with the blue regions, this presents some challenges during the matching process. We leave the development of a more effective map construction algorithm for future work.

## F. Details of Using LLMs for Creating Games

The data collection process guided by LLMs is shown as follows:

- **Basic Demo:** Start by describing the game we want to create using a prompt. This will help generate a simple Pygame demo of the basic game.

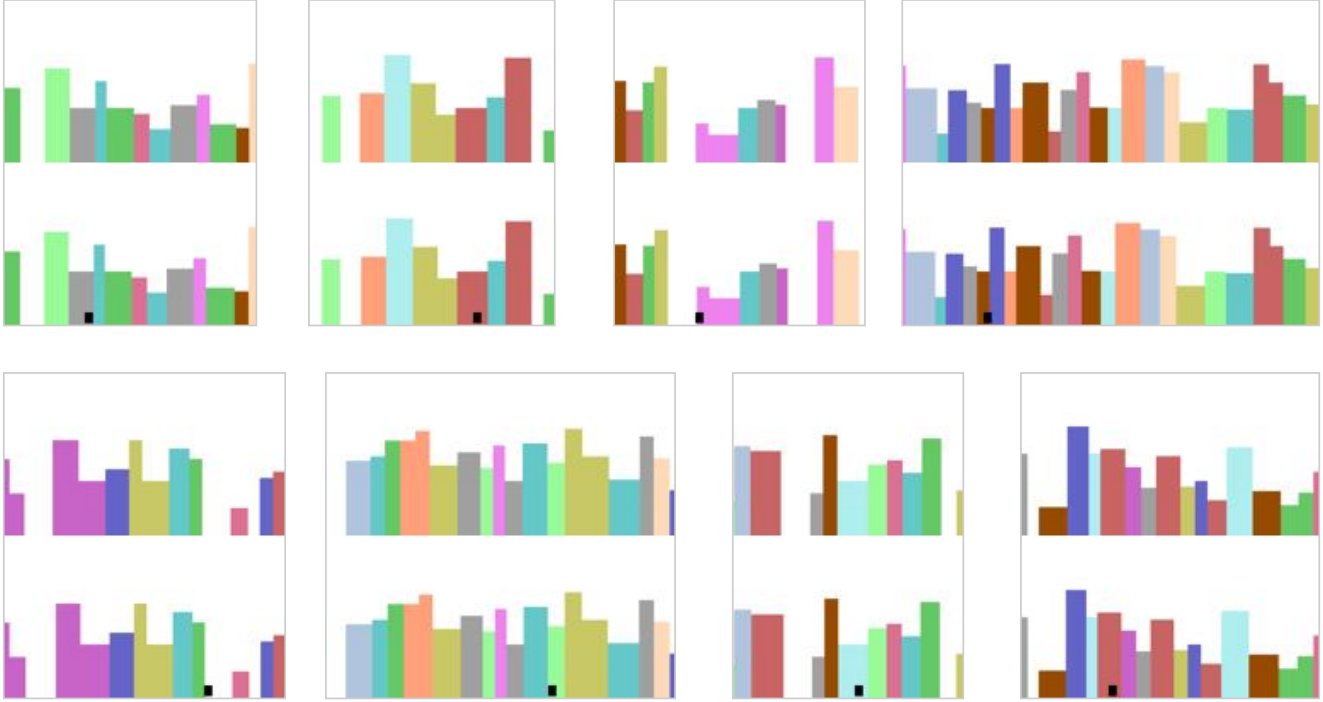


Figure 1. The upper map in each pair represents the ground truth, which is directly extracted from the Pygame program, whereas the bottom map is generated using the sliding window algorithm.

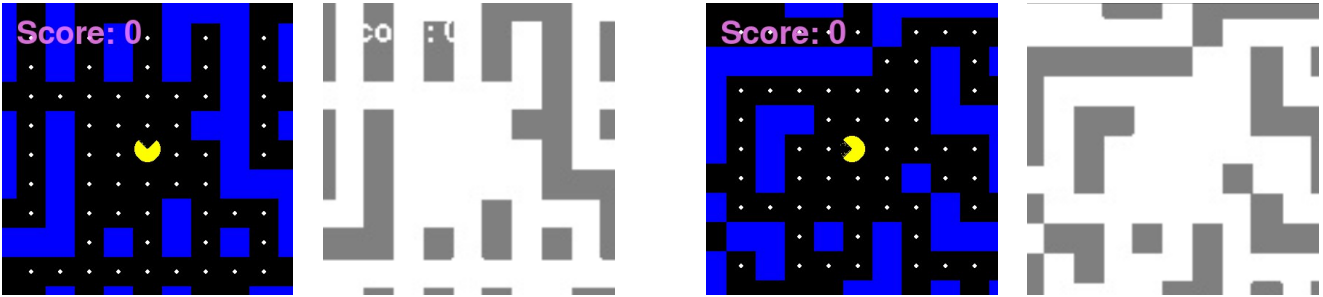


Figure 2. Pre-process to extract the blue areas in the game Pac-Man for matching.

- **Refinement:** Adjust and improve the demo based on specific needs. This involves deciding where to place numbers on the game screen. Make sure the numbers are big enough so they're clear and can be easily used by the VAE for encoding and decoding.
- **Automated Sequences:** Use prompts to design ways for the AI to play the game, either by creating intelligent exploration strategies or by setting up a rule-based action sequence. The goal is to make sure the AI explores every possible game state.
- **Multiprocess Sampling:** Run the game in a headless mode using 96 processes to gather data fast. Typically, we can collect data from 100K game episodes in less than an hour.

## G. Details of Models Provided by PGG

We inherit the models provided by PGG [2] and implement minor modifications, as outlined in Table 1 and 2.

## H. Detailed Architecture of Valid Action Model and Valid Numerical Model

The architecture of the Valid Action Model is presented below. Once trained, this model processes continuous inference on two observed frames to determine whether the predicted results are consistent with the actual actions input by the player. We

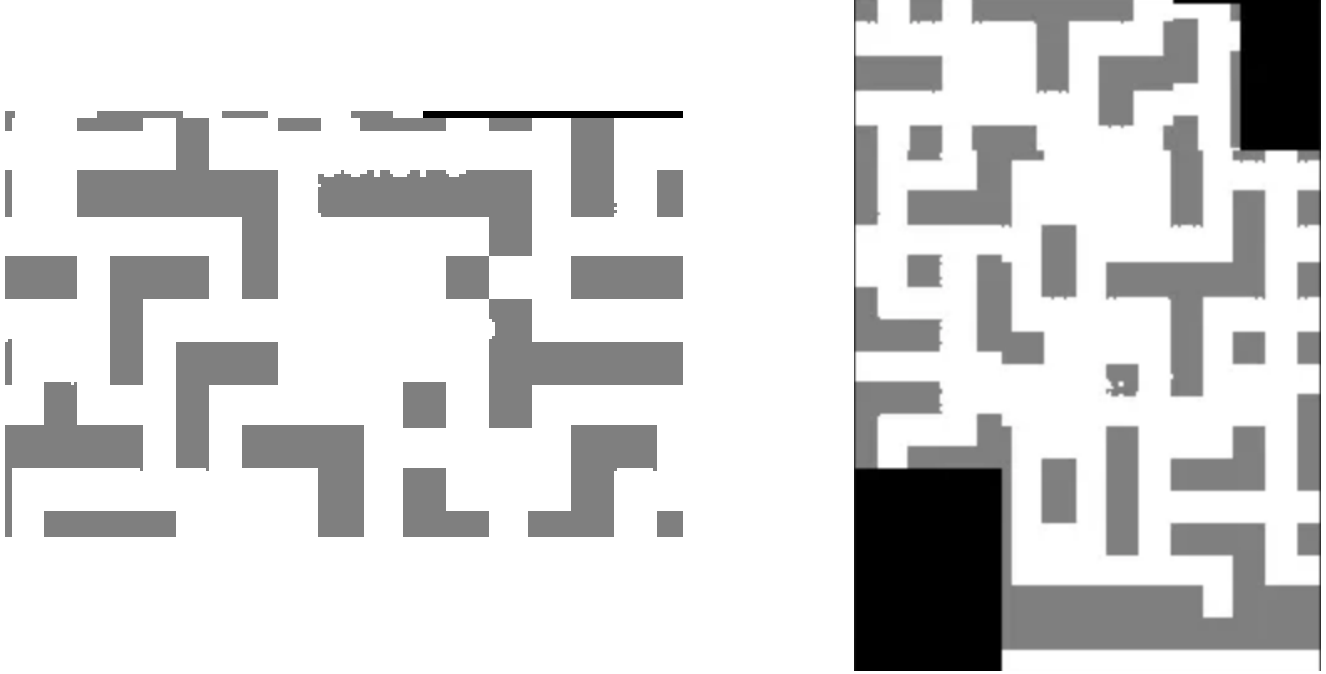


Figure 3. The constructed map of two episodes in the dataset.

Table 1. The configuration of VAE.

Item	Value
Batch Size	512
Channels	128
Channel Multiplier	[1, 2, 4]
ResNet Block Number	2
KL Loss Weight	1e-6
Perceptual Loss Weight	1
Learning Rate	4.5e-6

Table 2. The configuration of DiT.

Item	Value
Batch Size	512
Diffusion Steps	1000
Noise Schedule	linear
Sequence Length	32
DiT Depth	12
DiT Hidden Size	384
DiT Patch Size	2
DiT Num Heads	6
Time Embedding Dimension	192
Action Embedding Dimension	192
Learning Rate	3e-4

use the game Traveler as an example, with an input resolution of 96x96. For other games that require higher resolutions, the input dimensions can be appropriately adjusted. Accuracy is employed as the metric for evaluation.

140  
141

```

142 1 import torch
143 2 import torch.nn as nn
144 3
145 4 class ValidActionModel(nn.Module):
146 5     def __init__(self):
147 6         super(ValidActionModel, self).__init__()
148 7
149 8         # CNN encoder - these weights are shared between two images
150 9         self.encoder = nn.Sequential(
151 10             nn.Conv2d(3, 32, kernel_size=3, stride=2, padding=1), # 48x48x32
152 11             nn.ReLU(),
153 12             nn.BatchNorm2d(32),
154 13
155 14             nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1), # 24x24x64
156 15             nn.ReLU(),
157 16             nn.BatchNorm2d(64),
158 17
159 18             nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1), # 12x12x128
160 19             nn.ReLU(),
161 20             nn.BatchNorm2d(128),
162 21
163 22             nn.Conv2d(128, 256, kernel_size=3, stride=2, padding=1), # 6x6x256
164 23             nn.ReLU(),
165 24             nn.BatchNorm2d(256),
166 25
167 26             nn.Flatten() # 6*6*256 = 9216
168 27         )
169 28
170 29         # MLP classifier
171 30         self.classifier = nn.Sequential(
172 31             nn.Linear(9216 * 2, 1024), # *2 because we have features from two images
173 32             nn.ReLU(),
174 33             nn.Dropout(0.5),
175 34
176 35             nn.Linear(1024, 256),
177 36             nn.ReLU(),
178 37             nn.Dropout(0.3),
179 38
180 39             nn.Linear(256, 3) # Output 3 classes
181 40         )
182 41
183 42     def forward(self, img1, img2):
184 43         # Encode both images
185 44         feat1 = self.encoder(img1) # [batch_size, 9216]
186 45         feat2 = self.encoder(img2) # [batch_size, 9216]
187 46
188 47         # Concatenate features
189 48         combined = torch.cat([feat1, feat2], dim=1) # [batch_size, 9216*2]
190 49
191 50         # Classification
192 51         output = self.classifier(combined)
193 52
194 53         return output
195 54
196 55
197 56 if __name__ == '__main__':
198 57     model = ValidActionModel()
199 58     img1 = torch.randn(1, 3, 96, 96)
200 59     img2 = torch.randn(1, 3, 96, 96)
201 60     output = model(img1, img2)
202 61     print(output)

```

The architecture of the Valid Numerical Model is illustrated below. Once trained, the model performs inference by receiving a single frame input and an action to determine whether a specific event, defined as the ground truth, will occur. For our enhanced architecture, we can utilize an external score record to verify if an event has been triggered. For the baseline model, we employ a checkpoint from the TrOCR [1] printed text recognition system to obtain the current score. It has

been observed that TrOCR can almost accurately recognize text within game observations. We use the game Traveler as an example, with an input resolution of 96x96. For other games that require higher resolutions, the input dimensions can be appropriately adjusted.

```

1 import torch
2 import torch.nn as nn
3
4 class ValidNumericalModel(nn.Module):
5     def __init__(self):
6         super(ValidNumericalModel, self).__init__()
7
8         # CNN encoder - single image
9         self.encoder = nn.Sequential(
10             nn.Conv2d(3, 32, kernel_size=3, stride=2, padding=1), # 48x48x32
11             nn.ReLU(),
12             nn.BatchNorm2d(32),
13
14             nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1), # 24x24x64
15             nn.ReLU(),
16             nn.BatchNorm2d(64),
17
18             nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1), # 12x12x128
19             nn.ReLU(),
20             nn.BatchNorm2d(128),
21
22             nn.Conv2d(128, 256, kernel_size=3, stride=2, padding=1), # 6x6x256
23             nn.ReLU(),
24             nn.BatchNorm2d(256),
25
26             nn.Flatten() # 6*6*256 = 9216
27         )
28
29         # Action embedding layer
30         self.action_embedding = nn.Embedding(3, 128)
31
32         # MLP classifier
33         self.classifier = nn.Sequential(
34             nn.Linear(9216 + 128, 1024), # Image features + action embedding
35             nn.ReLU(),
36             nn.Dropout(0.5),
37
38             nn.Linear(1024, 256),
39             nn.ReLU(),
40             nn.Dropout(0.3),
41
42             nn.Linear(256, 3)
43         )
44
45     def forward(self, img, action):
46         # Encode image
47         img_feat = self.encoder(img) # [batch_size, 9216]
48
49         # Encode action
50         action_feat = self.action_embedding(action.long()) # [batch_size, 128]
51
52         # Concatenate features
53         combined = torch.cat([img_feat, action_feat], dim=1) # [batch_size, 9216+128]
54
55         # Classification
56         output = self.classifier(combined)
57
58         return output
59
60
61 if __name__ == '__main__':
62     model = ValidNumericalModel()
63     img = torch.randn(1, 3, 96, 96)

```

```

273 64 action = torch.randn(1, 3)
274 65 output = model(img, action)
275 66 print(output)

```

## 276 I. Visualizations of Failure Cases

277 As demonstrated in Figure 4, our spatial module will fail when the background comprises solely a single-colored building.  
 278 Regardless of the movements made by the black square, methods based on sliding window matching fail to extend the  
 279 map because the PSNR values remain high even when the window’s center is kept at its original position. A promising  
 280 improvement is to replace algorithm-based matching with neural network predictions. This approach requires taking into  
 281 account the actions for prediction and necessitates appropriate data training to handle these corner cases effectively.

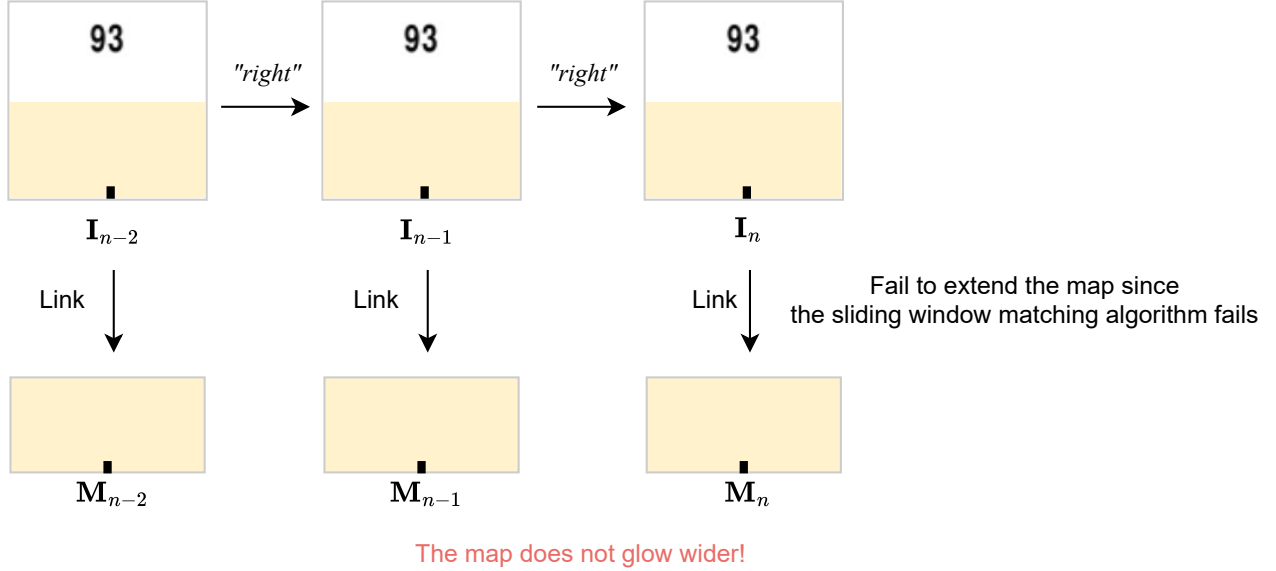


Figure 4. The failure case for the sliding window matching algorithm in the case of single-color background.

282 We observe that without strong supervisory features, such as physical laws, the model exhibits some instability during the  
 283 inference process. As illustrated in Figure 5, the black ball unexpectedly rises while it is supposed to be falling, which can  
 284 influence the players’ judgment and affect the gaming experience. We hypothesize that scaling up the model size or adopting  
 285 a more robust memory mechanism, replacing the current RNN-like diffusion, may mitigate this issue.

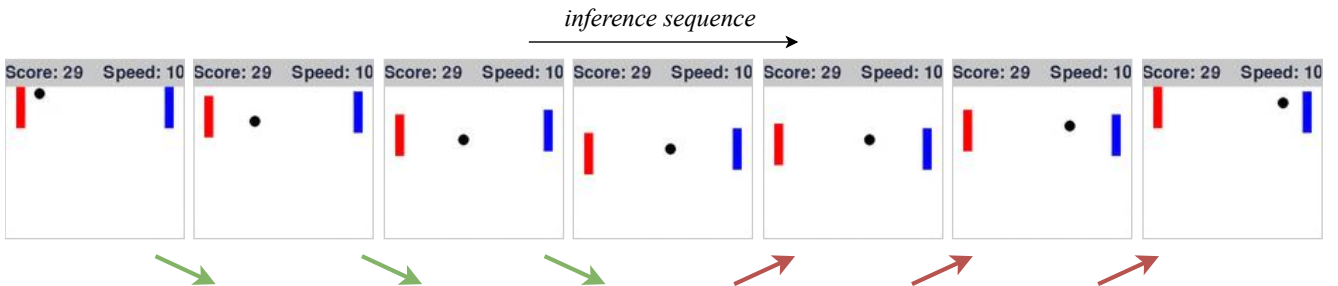


Figure 5. The failure case on the physical laws. During moving down (green arrows), the black ball suddenly move up (red arrows) which is not aligned with its original trajectory.

## References

- |  |                   |
|--|-------------------|
|  | 286               |
| [1] Minghao Li, Tengchao Lv, Jingye Chen, Lei Cui, Yijuan Lu, Dinei Florencio, Cha Zhang, Zhoujun Li, and Furu Wei. Trocr: Transformer-based optical character recognition with pre-trained models. In <i>Proceedings of the AAAI conference on artificial intelligence</i> , pages 13094–13102, 2023. <a href="#">6</a> | 287<br>288<br>289 |
| [2] Mingyu Yang, Junyou Li, Zhongbin Fang, Sheng Chen, Yangbin Yu, Qiang Fu, Wei Yang, and Deheng Ye. Playable game generation. <i>arXiv preprint arXiv:2412.00887</i> , 2024. <a href="#">4</a>   | 290<br>291        |