Inferring Structured World Models from Videos

by

Shreyas Kapur

S.B., Computer Science and Engineering, Massachusetts Institute of Technology (2020)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2022

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2022

Certified by .....
Joshua B. Tenenbaum
Professor
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Inferring Structured World Models from Videos

by

Shreyas Kapur

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Advances in reinforcement learning have allowed agents to learn a variety of board games and video games at superhuman levels. Unlike humans - which can generalize to a wide range of tasks with very little experience - these algorithms typically need vast number of experience replays to perform at the same level. In this thesis, we propose a model-based reinforcement learning approach that represents the environment using an explicit symbolic model in the form of a domain-specific language (DSL) that represents the world as a set of discrete objects with underlying latent properties that govern their dynamical interactions. We present a novel, neurally guided, on-line inference technique to recover the structured world representation from raw video observations, with the intent to be used for downstream model-based planning. We qualitatively evaluate our inference performance on classical Atari games, as well as on physics-based mobile games.

Thesis Supervisor: Joshua B. Tenenbaum
Title: Professor

Acknowledgments

Thank you to Josh Tenenbaum for his invaluable guidance, technical feedback, inspiration, and kind humanity throughout this project. Thank you to Ferran Alet for being an exceptional mentor, and technical advisor, and for his loving guidance and support. Thank you to Pedro Tsidivis for inspiring me to work on this project, his research laid the groundwork for this thesis. Thank you to Elizabeth DeTienne for supporting me with compassion and love throughout my program. Thank you to Arnav Kapur for his support and writing the physics engine used in this thesis, and to my mom for her unconditional love. Finally, I want to thank Nick Watters and Tan Zhi Xuan for deep technical discussions that have influenced this work.

Contents

1 Introduction **9**

1.1 Scope 11

1.2 Related Work 12

2 Generative Model **15**

2.1 Domain-Specific Language 15

2.2 Game Engine 19

2.2.1 Representation 20

2.2.2 Simulation 21

2.3 Probabilistic Model 22

3 Perception **25**

3.1 Game Manual Assumption 25

3.2 Object Detection 26

3.3 Object Tracking 28

3.4 Perception Module 30

4 Inference **33**

4.1 Problem 33

4.1.1 Likelihood 34

4.2 Classical Inference 35

4.2.1 Monte-Carlo Markov-Chain 35

4.2.2 Sequential Monte Carlo 37

4.3 Online Neural Amortized Inference 38
4.3.1 Independence and Windowing 39
4.3.2 Neural Conditional Density Estimation 39
4.3.3 Priors and Lonely Spaces 42
4.3.4 Contrastive Learning of Likelihood 43
4.3.5 Importance Sampling 43
4.4 Qualitative Results 44
4.4.1 Synthetic Examples 44
4.4.2 Tools Challenge 44
4.4.3 Games 46

5 Conclusion **47**
5.1 Future Work 47
5.1.1 Rigorous Evaluation on a Diverse Set of Tasks 47
5.1.2 Planning 48
5.1.3 Loosing The Game Manual Assumption 48
5.1.4 Differentiable Game Description Language 48

Chapter 1

Introduction

Reinforcement Learning (RL) is an area of Machine Learning that trains agents to take actions in some environments to maximize some notion of reward. Video games, and games in general, provide a convenient test-bed, since games are reproducible, deterministic, and provide rich, challenging complexities for RL algorithms. Recent advances in reinforcement learning have allowed these agents to learn a wide variety of board games and video games at superhuman levels [1]. However, unlike humans, these require an enormous number of experience trajectories to learn, typically on the order of millions of replays and do not generalize to unseen tasks without significant retraining [2].

One approach for tackling sample efficiency is to construct a model of a world that can predict future states given some simulated actions. We can then use this model to form a policy for an agent. While the trade-offs between *model-based* and *model-free* approaches are an active area of research [3], recent work has shown that model representations can have a significant impact on agent performance and sample efficiency [4–8].

In this work, we propose an explicit symbolic model for the environment in the form of a domain-specific language (DSL). We posit that game environments can be factorized into discreet objects with underlying latent properties that govern their dynamic interactions. The goal of the project is to design a language that can express these latent properties and interactions in an explicit symbolic represen-

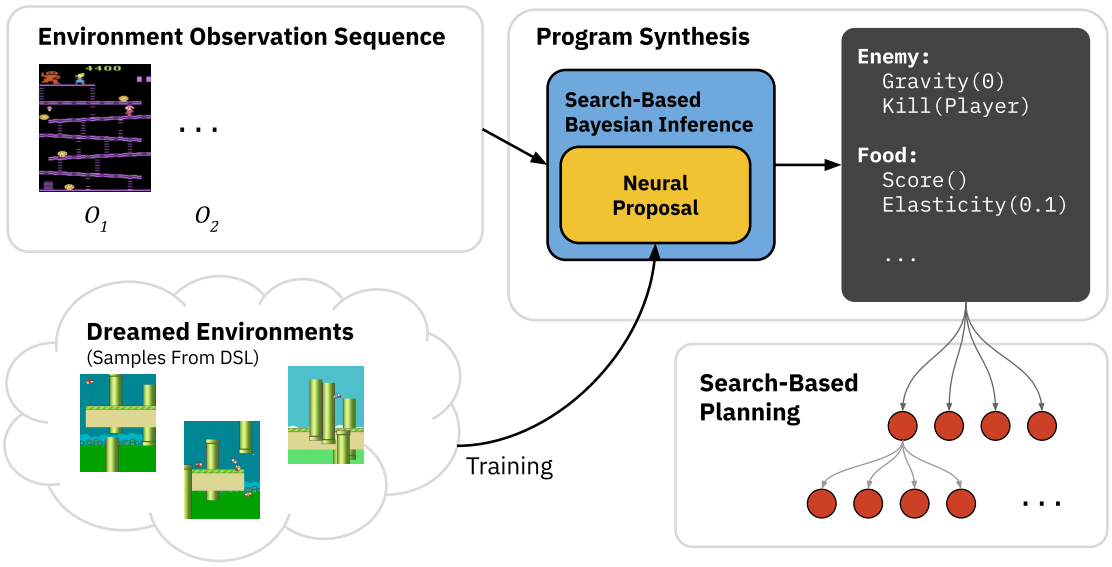


Figure 1-1: Overview of the algorithm proposed in this thesis. We first observe a sequence of observations from our current task environment. A search-based Bayesian inference algorithm makes use of a neural proposal to generate a distribution over plausible programs that can potentially explain the input observation sequence. The program is then used to predict future states and rewards for a tree-search to decide the best action for the agent.

tation that is rich enough to model video-games and to work on approaches to automatically synthesize this language from a series of raw pixel observations in a video-game.

We hope that we can learn these symbolic models faster than neural models while being rich enough to achieve comparable performance on downstream planning and policy tasks. Since we have access to a deterministic game description language, we hope that we can leverage the ability to sample a large number of example games to train a meta-inference engine that that can generalize to a range of tasks. We also hope that the learned models will be more interpretable by humans.

1.1 Scope

While learning symbolic models opens the door to fine-tuning planning algorithms and heuristics, we are going to limit the scope of the project to focus on just the inference of the world models. Here, we list a set of potential candidates:

- The Atari game suite is a standard benchmark in reinforcement learning and has a canonical implementation through the Arcade learning environment [9]. The task set consists of a range of complexities, from simple control problems (like *Pong*) to tasks that require significant long-term planning (like *Frostbite*). Since we are limiting the scope to just representation learning, these tasks provide a challenging benchmark for learning to model complex interactions between game objects.

- The tools challenge [10] provides a suite of physical intuition tasks where the agent must use shaped tools to solve physical puzzles. The tools challenge is an appealing candidate since, in this work, the authors showed that performing trial and error simulations in a noisy physics model can outperform deep policies. Our approach could potentially build on top of this work, by first learning the physical model from game-play.

- Mobile games is another area we would like to explore. In particular, we would like to explore simple physics bases games. We have published a few games in this genre that could be useful to collect data on human learning in these games. These could provide insightful inspirations and comparisons for our approaches.

For this thesis, we limit the scope of the work to just the inference, and do not evaluate sample efficiency for planning using such models.

1.2 Related Work

Model-Based Reinforcement Learning by Simulating Video. Prior work in this category typically learn neural models that can synthesize future image frames. The CDNA model [11] uses a convolutional LSTM to directly predict RGB frames. Lee et al. [12] combine the success of variational auto-encoders at modeling stochasticity and the success of generative-adversarial networks at producing naturalistic images into a unified video prediction model. Kaiser et al. [4] introduced SimPLe, which uses a video prediction model for reinforcement learning.

These approaches represent the scene with a single vector, which doesn't allow meaningful disentanglement in the object space. From a cognitive science perspective, these representations are susceptible to the binding problem [13] due to the central limiting effect of the permutation invariance of objects. Ideally, we would find components in these vectors that would represent each object separately but object-level disentanglement can be hard to achieve in practice without additional inductive biases [14].

Object-Centric Representation Learning. Learning disentangled object representations from pixels has been an active area of research. Approaches like MONet [15] can perform unsupervised scene decomposition by using an attention-like mechanism before a variational auto-encoder. IODINE [16] improves on this by

introducing a probabilistic generative model that assumes separate latent vectors for each object. The final image is modeled as a mixture of these individual objects, and the posterior is estimated via iterated variational inference. Recently, approaches like slot attention [17] show how a transformer-like attention model for each object slot can be used for object-based scene decomposition. In all these approaches, the latent representations are discovered in an unsupervised manner. Since we impose a DSL, we also impose a strict structure to the latent variables, making them disentangled by construction.

Object-Centric Model-Based Reinforcement Learning. These approaches are the closest prior work to the proposed thesis project. The Transporter algorithm [18] learns to disambiguate key-points from an observation image to predict future frames. COBRA [5] uses an autoregressive neural network to factorize the scene into objects. It has an adversarial exploration phase where the agent freely learns the objects and dynamics of the scene which it can then use for model-based RL. C-SWM [19] and OP3 [8] follows [16] by using an object-based probabilistic generative model, explicitly modeling discreet entities, their symmetric dynamics, and observations.

Program Synthesis and Reinforcement Learning. Program synthesis has seen success with neurally-guided search [20–22]. Reinforcement learning has also been used in optimizing program traces [23]. Close to the proposed work is prior work learning RL policies as symbolic programs [24]. We do not know any publicly available approaches that model the environment in a domain-specific language.

Inferring Symbolic Models The closest work to this thesis, and what we build on is the EMPA agent described by Tsividis et al. [25]. Here, the authors posited a domain-specific language and performed Bayesian inference to learn the game rules, successfully achieving human-efficient performance on a diverse set of tasks. In this work, however, the authors used hand designed heuristics to drive the in-

ference and applied the technique to grid games. Ullman et al. [26] similarly developed novel techniques to infer physical properties from dynamical scenes but on a hypothesis space that is computationally enumerable.

Chapter 2

Generative Model

2.1 Domain-Specific Language

We first introduce our custom domain-specific language (DSL) that can describe 2D video games. The design goals for this language are,

1. We want to be able to represent a variable number of multiple objects in a scene, alongside their *fixed* dynamical properties and *time-varying* state.

2. We want to be able to flatten this representation into a single fixed-size vector such that it could be the output of a neural network.

3. We would like the language to have inherent performance features that would allow us to perform a large number of game simulations in parallel.

To this end, we opted for an Entity-Component-System (ECS) like design pattern, inspired by its wide usage in the video game development community. Härkönen [27] provides a deeper review of the trade-offs for this pattern.

In our language, Entities represent an abstract container for each game object and can be indexed using the object's entity index. Figure 2-1 shows an example of the *Donkey Kong* game, which we will use as a running example to illustrate the DSL. Here, all game objects are entities. The player's sprite, each obstacle, the goal (E), each ladder (C), etc.

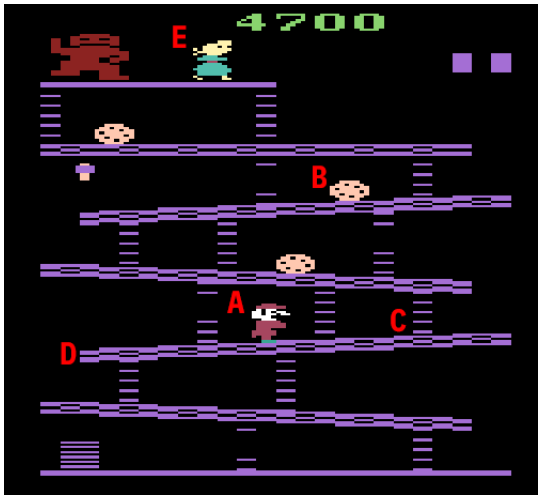


Figure 2-1: A screenshot from the Atari 2600 version of *Donkey Kong*. We will use this screenshot as a running example to describe our DSL. The goal of the game is for the player (A) to reach (E) while avoiding the obstacles (B). The player is allowed to climb ladders (C). The obstacles are spawned at random instances.

Entities themselves do not hold any state or rules, except for their entity index. Instead, entities hold a collection of Components. A component is a parameterized rule or state. For instance, a `Position(x: float32, y: float32)` component holds data about the position of the entity it is attached to. A `Sticky()` component enables an entity to allow other entities to not be influenced by gravity (for instance, ladders would be sticky). In our language, we pre-define a set of primitive attachable components alongside their behaviors. A full list of them can be found in Table 2.1. Components that change their data values over time are marked as *dynamic*.

Component Signature	Behaviour	Dynamic	Generic

`Position(`
`  x: float32,`
`  y: float32,`
`)`
	Stores the current position of the game object.	✓	

`Velocity(`
`  vx: float32,`
`  vy: float32,`
`)`
	Stores the current velocity of the game object.	✓

`Gravity(`
`  gx: float32,`
`  gy: float32,`
`)`
	Acceleration due to gravity.	

`CollidingBody(`
`  size_x: float32,`
`  size_y: float32,`
`  body_type:`
`  {`
`  STATIC,`
`  DYNAMIC,`
`  SENSOR`
`  },`
`  shape: {RECT, ELLIPSE},`
`)`
	Dynamical body of the object. Includes the size and shape of the object. If the body is static, it is not allowed to move. A sensor body is allowed to move but does not respond to physical collisions.	

`ActionSetVelocity<K>(`
`  vx: float32,`
`  vy: float32,`
`  is_zero: bool,`
`)`
	Set the velocity of the object to (vx, vy) for action K.	✓

`Perishable()`	Object is destroyed if contacted by a `Killer`.	

`Killer()`	Object destroys a `Perishable`.	
`Sticky()`	Temporarily sets gravity to 0 for a contacting object.	
`Scoreable()`	Enables scoring for object.	
`GameOver()`	Stops the game with negative reward if `Scoreable` contacts.	
`Score()`	Earns positive reward on contact with `Scoreable`.	
`OOBGameOver<E>()`	Trigger game over if object goes out of bounds at edge $E \in \{$ `TOP,LEFT,BOTTOM,RIGHT`$\}$	✓

Table 2.1: Complete list of primitive components available to our DSL.

In a traditional ECS design, a system performs the actual computation to simulate the rules and states described by the components. For instance, a physics system would process the physical dynamics, an action system would process keyboard/mouse inputs, etc. For our DSL, we have a single system we call the *simulator*, which performs the necessary computations to step the scene one time-step.

The language is compositional, as opposed to hierarchical, which is traditional in object-oriented design patterns. This is a salient feature that makes our language particularly expressive for games, allowing us to layer complex behavior for entities by simply attaching or removing specific components.

The reader might notice that we do not have any components that describe the visual properties of game objects. This is by design and is discussed extensively

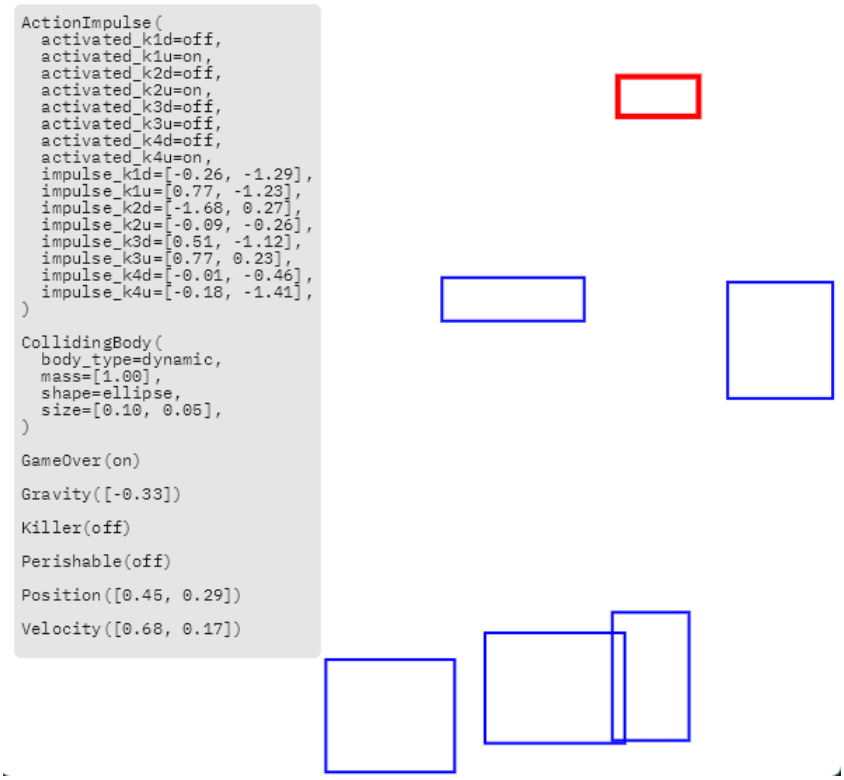


Figure 2-2: A visualization of the Entity-Component DSL. Each box is an entity in our game. The left hand side shows all the components that are attached to the selected entity (in red).

in the next chapter on perception. For simplicity, our generative model produces bounding box observations for each game object. Additionally, we have not yet specified any assumptions about object types that are critical in describing games. We sidestep this issue by lifting the object types to the perception stack as well, as discussed in the next chapter.

2.2 Game Engine

The game engine is responsible for performing the computation needed to advance the game state, execute the rules, and process the physics.

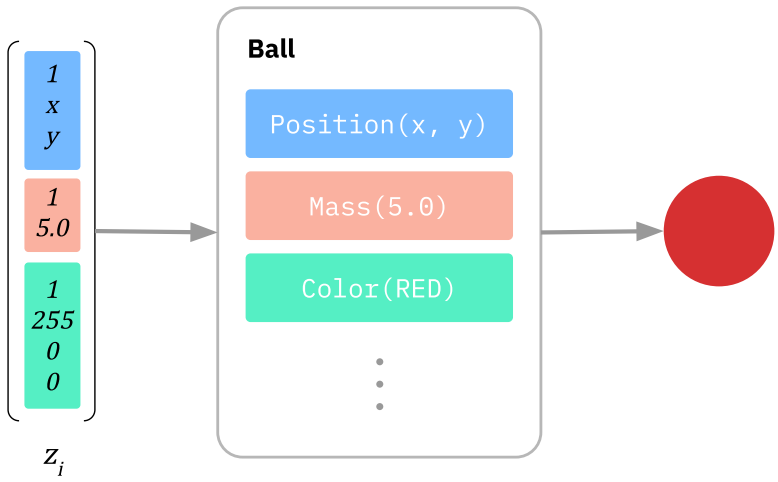


Figure 2-3: An example of how a latent representation for a single entity gets translated into its respective components. The vector z_i consists of the component's presence bit followed by the component's parameters. Since we establish a fixed library of components, the size, L of each z_i is fixed across all entities.

2.2.1 Representation

Our DSL is embedded inside Python for ease of developer use. Embedding our DSL also allows us to skip expensive parser and lexer steps. Since our DSL is entirely compositional, we do not need to maintain a complex abstract-syntax tree. Instead, we can flatten the entire state of the game at any point into $z \in \mathbb{R}^{M \times L}$ tensor, where M is the maximum number of object slots for the engine. Each of those M slots consists of the ith object vector, z_i. Figure 2-3 shows how we interpret each z_i. The number of components in our primitive library is fixed, allowing us to fix the indexes where we would find a particular component's parameters.

Leveraging this property allows us to have a direct mapping of states to memory, skipping any expensive parsing. For a better developer experience, we implement a `schema` object that provides us with the relevant slices for specific properties in the state tensor, mapping human-readable component names to their relevant indexes. In Python, the schema is used as,

```
# Initialize a schema.
schema = Schema(...)
```

```
# Some game state.
state = np.array([...])

# Interpret the tensor in a human-friendly proxy object.
view = schema.interpret(state)
view.CollidingBody.size += 1  # Mutates `state` directly.

# Bulk access properties.
state[:, schema.Gravity.gravity.slice] = 5
```

A side effect of this approach is that whenever a particular property is accessed, the neighboring properties of the same entity are brought into the cache, allowing for fast property reads by the simulation engine.

2.2.2 Simulation

The simulator is a function $s : \mathbb{R}^{M \times L} \to \mathbb{R}^{M \times L}$ which simulates our world state by a single time-step, dt,

$$z_{t+1}, r_t = s(z_t; a_t)$$

where a_t is the action at time t and r_t is the reward.

Practically, we also need the simulator to be able to do batched simulations, i.e., forwarding a batch of game states to the next. We implement our physics engine using Numba [28], which is a high-performance compiler for numerical Python code. The Numba compiler directly provides us with a high-performance compiled artifact for batched s. For simulating the physics, our engine can interface with Chipmunk [29], Box2D [30], Bullet [31] rigid body physics libraries, or our our own physics engine written in Numba. Our game engine can simulate our DSL at $\sim 10^4$ time-steps per second in batched mode, taking full advantage of multiple

threads, CPU vectorization, and caching.

2.3 Probabilistic Model

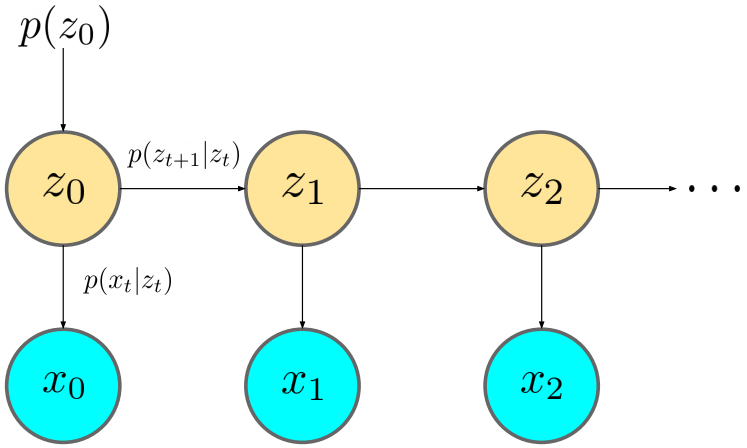


Figure 2-4: The probabilistic graphical version for our generative model. The actions are omitted for clarity.

For some time t, let z_t represent the latent variables (as described in previous sections), and $x_{i,t}$ represent the observations for the ith object. Let δ be the Dirac delta function, and $\texttt{BoundingBox}(z_{i,t})$ extract an axis-aligned bounding box of the ith object's latent state. Then, our probabilistic generative model is,

$$
\begin{aligned}
z_0 &\sim U\{\mathbb{R}^{M\times L}\} \\
z_{t+1} &\sim \delta(s(z_t; a_t)) \\
B_i &\sim \text{Bernoulli}[1-m] \\
x_{i,t} &\sim B_i \cdot \mathcal{N}(\text{BoundingBox}(z_{i,t}), \sigma_o^2)
\end{aligned}
$$

If A represents the set of all actions, and p represents the probability of pressing a key, we sample actions as,

$$
\begin{aligned}
K_t &\sim \text{Bernoulli}[p] \\
a_t &\sim K_t \cdot U\{A\}
\end{aligned}
$$

The above two describe a Hidden-Markov Model (HMM), where the observation is simply a noisy bounding box of the true underlying state, with observation noise as σ_o^2. Sometimes, we might not be able to detect an object in the game, and therefore include a probability of missing an observation as $m \in (0, 1]$. The HMM formulation yields a factorization for the joint distribution for a total of T time-steps,

$$p(z_{0:T}, x_{0:T}; a_{0:T}) = p(z_0) \prod_t p(x_t|z_t) \prod_t p(z_{t+1}|z_t; a_t)$$

For the limiting case when T is very large, our joint product is dominated by the transition and emission distributions. To that end, we assume a uniform prior over any possible valid instantiation of z_0. Figure 2-4 shows the graphical version of our probabilistic model.

Chapter 3

Perception

In the previous chapter, we discussed a generative model that can take some latent game state and simulate it. Additionally, this model can produce noisy bounding box observations for each game object. However, for real game environments, the observations are in the form of raw pixel image sequences. In this chapter we discuss how we use generative modeling techniques to infer bounding boxes from raw pixel observations, to be consumed by an inference pipeline downstream.

3.1 Game Manual Assumption

Inferring game rules and game states directly from video is challenging since we do not have access to the visual language of the game. This means that a generative model would not be able to produce image observations that are in-distribution of a real game. We do not claim that this is an impossible proposition, and existing work has used generative assumptions [32] or motion assumptions [18] to extract object-level information from visual videos. In this thesis, we focus on the inference of dynamical rules and states, and to that end make certain simplifying assumptions about the visual language of the game.

In particular, we assume access to the visual language of the game in the form of sprite sheets. Figure 3-1 shows an example sheet we assume access to. In addition to the sprites used for the game, we also assume semantic labeling of

object classes. Under this assumption, all game objects that have the same object class will share the same non-dynamic game components. As an example, a game object that is an instance of the `Ladder` class, will always have a `Sticky()` component attached to it.

We refer to these assumptions as the *Game Manual Assumption*, which loosely alludes to how these assumptions are similar to if our system had access to the game manual, which usually contains descriptions and sprites for the games. To make in-game data-efficiency comparisons fair against baselines, we will provide baseline methods with the semantically segmented output from our perception engine.

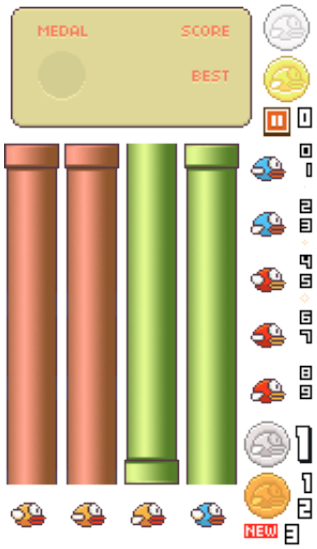


Figure 3-1: An example of a sprite-sheet used to train the perception model. This is only a part of the complete sprite-sheet.

3.2 Object Detection

We use the Faster R-CNN approach [33] to train a neural object detector. We use the sprite sheet to generate random images with known object bounding boxes to train the detector.

Algorithm 1 describes the process in which we sample random images pro-

Algorithm 1 Probabilistic generative model to train the object detector for perception.

procedure SampleImage
  canvas $\leftarrow$ EmptyCanvas()
  $n \sim U\{1, M\}$ ▷ Sample number of objects

  background $\sim U\{$backgrounds$\}$
  draw background on canvas

  boxes $\leftarrow \phi$

  for $i \leftarrow 1, n$ **do**
    anti_alias $\sim$ Bernoulli(0.5)
    sprite $\sim U\{$sprites$\}$
    $sx \sim U(s_{\min}, s_{\max})$ ▷ Sample scale
    $sy \sim U(s_{\min}, s_{\max})$

    axis_aligned $\sim$ Bernoulli(0.5)
    if axis_aligned **then**
      $\theta \sim U\left\{0, \frac{\pi}{2}, \pi, \frac{3}{2}\pi, 2\pi\right\}$
    else
      $\theta \sim U(0, 2\pi)$
    end if

    $x \sim U(0, 1)$
    $y \sim U(0, 1)$

    tile $\sim$ Bernoulli(0.1)

    set draw tile and anti_alias
    draw sprite on canvas with transform (x, y, sx, sy, θ)

    boxes $\leftarrow$ boxes $+ (x, y, sx, sy)$
  end for

  return canvas and boxes
end procedure

vided by the sprite sheet. We first sample a background image and draw it on an empty canvas. Then we sample the number of objects in the scene uniformly between 0 and M, which is the max number of object slots. For each object slot, we sample a random sprite image and sample a random transform. We also allow the rotation of the sprite to be axis-aligned since this is a prevalent case in real games. We also randomly apply anti-aliasing or not while drawing since this behavior is also game dependent. For instance, games that are heavy in pixel art like to use nearest-neighbor resampling to keep the pixel art aesthetic. In such games, anti-aliasing would blur the game sprites, making such training examples out of distribution. We also allow sprites to randomly be tiled across the screen in the direction chosen by θ. This allows us to represent tileable sprites for objects like walls, ground, etc.

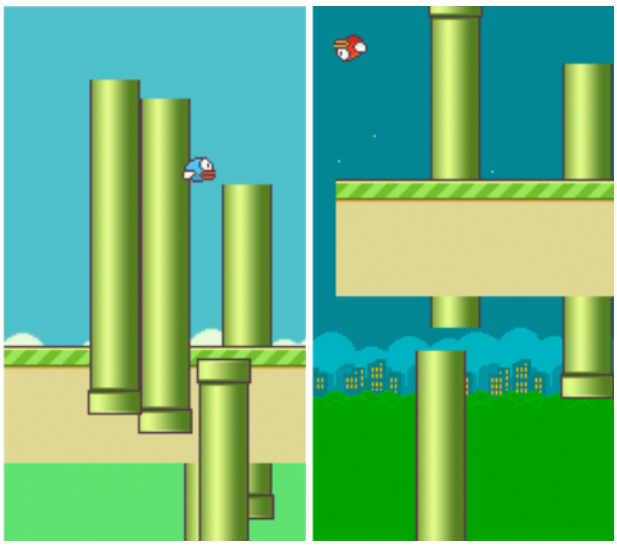


Figure 3-2: Examples of images sampled by the perception generative model.

3.3 Object Tracking

The Object Detection module provides us with bounding box detection of game objects for every video frame. Without any object tracking, the inference engine must consider all permutations of objects for every frame. While straight bound-

ing boxes are sufficient to perform inference for the model described in Section 2.3, we also perform object tracking between time steps to ease the computation burden on the inference engine.

Just like object detection, we treat object tracking through an inference and generative model lens. Since we are no longer in the visual domain, we can leverage our generative simulator, s as a motion model for object tracking. We amortize the inverse inference model by training a graph neural network (GNN) [34] to predict object associations across time steps.

Object tracking is traditionally performed by treating the problem as a graph flow problem [35]. Each node is a bounding box prediction produced by the detection engine, and edges represent potential associations between two bounding boxes across time. Weights on these edges represent a heuristic metric of the probability that the two bounding boxes refer to the same object. If all edges are given unit capacity, and appropriate source and sink nodes are added, the min-cost flow problem on this graph allows us to associate bounding boxes across time.

Instead of solving the resulting flow problem, we choose to learn to directly predict edge labels (whether the edge is active or not) using a graph neural network akin to Brasó and Leal-Taixé [36]. Training data is generated in windows of W time-steps by sampling an initial state $z_0 \sim p(z_0)$ from the prior, and unrolling for W time-steps by repeatedly applying s. Training the GNN based object tracker using data from our game engine makes it familiar with game rules, motion models, and potential actions. Since our generative model simulates observation noise and potentially missing objects, our object tracker robustly handles cases when the object detector fails to detect an object. In the current version, we do not handle the case when there is a spurious detection, or if the detector misclassified an object. Figure 3-3 shows the input graph and the output from our tracker module.

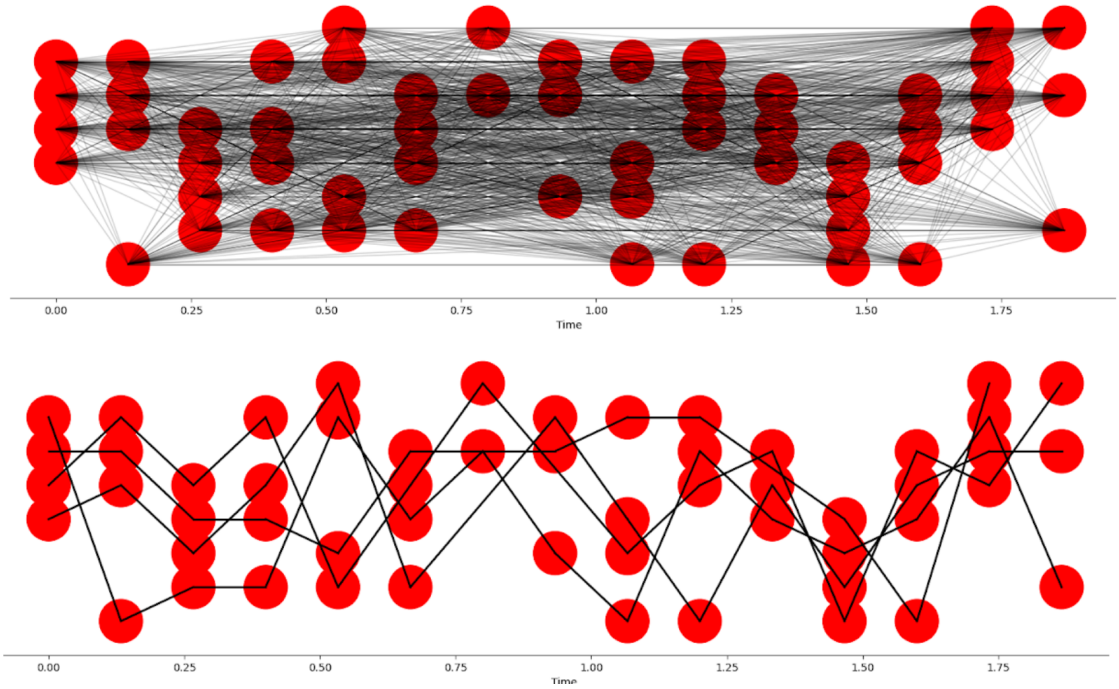


Figure 3-3: Input to the GNN based object tracker (top) and predictions from the GNN (bottom). Each red circle represents a bounding box, with time varying across the x-axis. In the top image, we see *all* possible associations between bounding boxes across time and on the bottom we see a thresholded output from the GNN. Note how we can trace trajectories of objects across time. The object tracker is also robust to missing bounding boxes.

3.4 Perception Module

We refer to the combination of the object detector and object tracker as our *perception module*. A key advantage to this approach is that the tracker neural network only needs to be trained once and works out of the box for novel game environments. The detector neural network must be trained once for every game environment. Figure 3-4 shows the result of the techniques described above on a real game. In all, this module turns video frames into associated bounding box observations to be consumed by a downstream inference engine to infer rules.

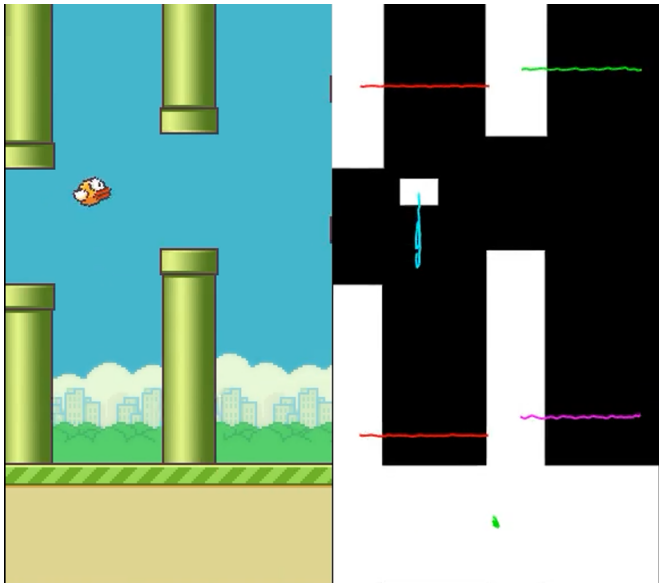


Figure 3-4: Result of the perception module on the game, *Flappy Bird*. On the left we see the raw pixel observation and on the right we see detection and tracking results. Each white box represents a game object detection and each colored line represents an individual object trajectory for the last 30 time-steps.

Chapter 4

Inference

In this chapter, we first formalize the inference problem for the generative model we introduced in Chapter 2. Then, we will motivate some of the challenges of performing inference for our specific problem by reviewing state-of-the-art inference techniques.

4.1 Problem

As defined in Chapter 2, we would like to ingest a sequence of images, $I_0, I_1, \ldots, I_T$ and infer a distribution of games rules and game state encapsulated by the latent variable, z_T. As mentioned in Chapter 3, we are able to translate image observations into tracked bounding box observations over time, denoted by, $x_0, x_1, \ldots, x_T$. From Bayes Law, our posterior of interest is,

$$p(z_{0:T}|x_{0:T}) \propto p(z_{0:T})p(x_{0:T}|z_{0:T}) \tag{4.1}$$

$$= p(z_0) \prod_t p(x_t|z_t) \prod_t p(z_{t+1}|z_t) \tag{4.2}$$

Here, we omit auxiliary variables such as actions (a_t) and rewards (r_t). Actions are treated as constants and rewards are treated as part of the observations. Since our simulator is deterministic, $p(z_{t+1}|z_t)$ is a delta mass, and will always be unity if z_{t+1} did exactly arrive from z_t and 0 otherwise. This yields an intuitive algorithm

for computing the density for a specific query z_0,

1. Unroll the query z_0 deterministically using the simulator s into $z_1, z_2, \ldots, z_T$.

2. Compute the likelihood for each z_t against x_t and take the product.

3. Multiply this product with the prior density of z_0, by evaluating $p(z_0)$.

Our final goal is to be able to efficiently sample from the posterior $p(z_0|x_{0:T})$ (the smoothing distribution) or sample from the posterior $p(z_T|x_{0:T})$ (the filtering distribution) since our simulation is deterministic.

4.1.1 Likelihood

Step 2 above requires us to repeatedly evaluate $p(x_t|z_t)$, which intuitively refers to the likelihood we would observe bounding boxes specified in x_t if z_t contained the true game state and game rules. In Section 2.3, we modeled bounding boxes as noisy version of the true position and size of the game object. Let $\Phi_{\mu,\sigma^2}(x)$ be the probability density function of a normal, $\mathcal{N}(\mu, \sigma^2)$ evaluated at x. Then,

$$z_{t+1} = s(z_t) \tag{4.3}$$

$$\log p(z_0|x_{0:T}) = \log p(z_0) + \sum_t \log p(x_t|z_t) \tag{4.4}$$

$$\log p(x_t|z_t) = \sum_i \sum_j \log \Phi_{x_{i,t},\sigma_o^2}(\text{BoundingBox}(z_{j,t})) \tag{4.5}$$

For M object slots, the above computation takes $\Theta(M^2)$ time, since we compute the PDF between the bounding boxes and the observations in a pair-wise manner. We optimize this by noting that the contribution of the PDFs for boxes that are far away is going to be negligible. We leverage an efficient data structure such as a k-d tree, to be able to compute the nearest neighbor, and only evaluate PDFs for that.

4.2 Classical Inference

In this section, we will discuss various standard inference techniques in literature, and discuss their respective challenges, trade-offs, and drawbacks for our specific problem. A straw man inference technique would be to enumerate the entire space of z_0 and evaluate the posterior density of the smoothing distribution as in Equation 4.5. However, enumerating the space of all latents is intractable. Additionally, we would like to perform our time-series inference *online*. Since evaluating the density for a single z_0 is $\Theta(T)$ time, every time we receive a new observation x_{t+1}, we will be forced to perform a $\Theta(T)$ computation. Instead, we would like to make $O(1)$ time updates to our posterior.

While offering a promising future direction of work, our generative model and game engine contain a lot of discontinuities and branches and is not differentiable. Inference techniques such as Vocational Inference (VI) [37–39] and Hamiltonian Monte-Carlo (HMC) [40] that utilize the gradients are therefore not applicable here. We, therefore, explore techniques that can effectively leverage the forward model in a black-box manner.

4.2.1 Monte-Carlo Markov-Chain

Instead of enumerating the entire latent space, Monte-Carlo Markov-Chain (MCMC) methods stochastically explore the latent space towards areas with higher probability mass by repeatedly applying a transition kernel to an initial guess. A Metropolis-Hastings accept-reject step decides if a particular transition is useful. A key advantage of MCMC is a guarantee of generating samples from the target posterior density if the number of MCMC iterations goes to ∞. However, in a practical sense, we realize that inferring over physical dynamics and game rules in a high-dimensional latent space tends to be very sensitive to the initial state. Here we used a random-walk kernel of adding normal noise to the latent vector.

Figure 4-1 shows the needle in a haystack behavior of the likelihood surface for a simpler 2 dimensional posterior. A slight change in the initial velocity of an

object can cause a chaotic difference in the final trajectory of an object. Without engineering the kernel, a random-walk MCMC kernel exploring this probability space would aimlessly explore regions of low probability unless it gets lucky to be near the high probability region.

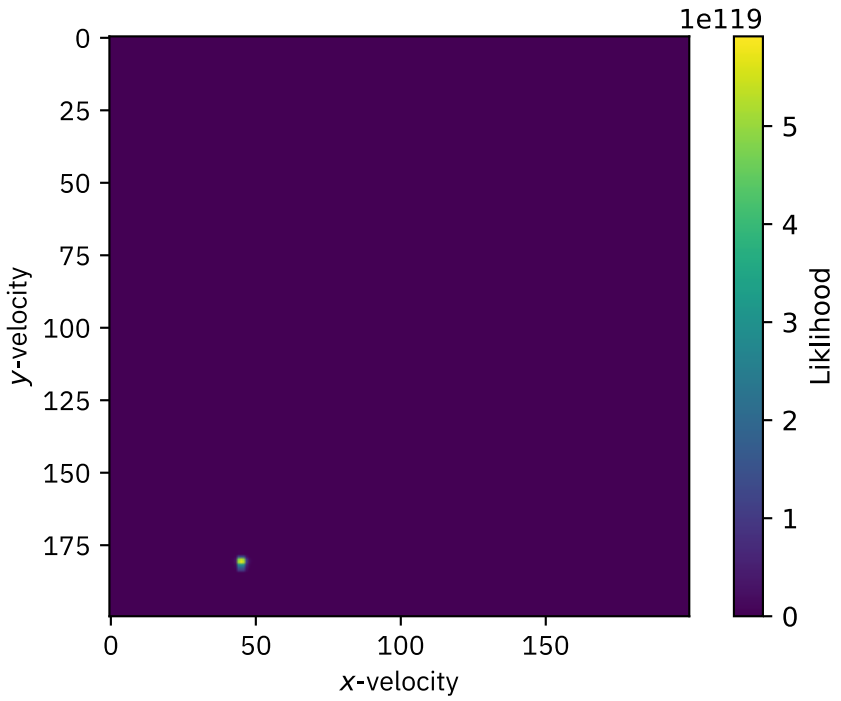


Figure 4-1: Enumerated likelihood surface for a two-dimensional latent space where we are just inferring over the velocity of a single object. We see how the liklihood is very sensitive to the initialization, since a slight change in the initial velocity of an object can result in a very different trajectory.

Even if we solve the scalability issues, this naïve way of using MH-MCMC still does not allow us to perform online inference by repeatedly applying Bayesian updates to a fixed posterior representation. Instead, we have to simulate all T steps to perform a Bayesian update. These issues warrant a method that can zoom into areas of high probability over time and discriminate more higher-level game rules abstractions instead of just state estimation.

4.2.2 Sequential Monte Carlo

Sequential Monte Carlo (SMC) methods (also known as Particle Filters) tend to address these challenges and have been successfully, and widely used in online state estimation tasks [41] where the filtering distribution is desirable. These methods have also been successfully used to infer physical properties in dynamical scenes [42]. However, they do also suffer a dimensionality curse and do not adequately scale to our generative model without proposal engineering. Here, we will briefly review Doucet et al. [41] in the context of our problem.

SMC methods represent the current posterior as a weighted set of K particles, $\{(w_k, p_k)\}_{k=1}^{K}$, which is used to approximate the posterior as,

$$\hat{p}(z|x) = \sum_{k=1}^{K} w_k \delta(p_k) / \sum_{k=1}^{K} w_k$$

We first sample an initial set of K particles from the prior, $p(z_0)$. And scrutinize these particles against the likelihood $p(x_0|z_0)$ to compute their corresponding weights. Then, each particle is deterministically propagated using $p(z_{t+1}|z_t) = \delta(s(z_t))$, and scrutinized again using the likelihood to update the weights. The final collection of particles at time $t = T$ represent the desired posterior samples.

Resampling and Rejuvenation

In this simplistic method of performing SMC, we can see how the initial draw from the prior can affect the performance of the filter. Very similar to the MCMC case, we require a hypothesis in the latent space that is close to the ground truth hypothesis to be sampled in that initial step. Since we have a needle in a haystack problem, most particle weights would degenerate to zero. To counter this, most SMC methods additionally add a resampling step, where the particles with higher weight are duplicated and particles with lower weights are dropped, resetting the weights back to unity. In our experiments, we used the systematic resampling method [43].

For our generative model, the simulation for $p(z_{t+1}|z_t)$ is entirely deterministic. This means that the resampling step does not help us, since it will just duplicate one particle K times in the end. To fix this issue, we added rejuvenation MCMC moves [44]. Here, we simply add a random-walk kernel (in the form of Gaussian noise) and an MH accept-reject step, perturbing all the particles from their deterministic predicted state, encouraging exploration in the hypothesis space. Despite these improvements, we were unable to get SMC to scale beyond a single object.

4.3 Online Neural Amortized Inference

Tsividis et al. [25] and [26] both make use of hand-designed heuristics to perform inference. Intuitively, one can construct heuristics that can estimate game rules and game state without having to perform search as SMC and MCMC methods would do. As an example, one can estimate the velocity of an object by taking local derivatives in the position. One can estimate if an object is `Perishable` by checking if it disappeared after a collision etc. Hand designing heuristics, however, don't respond to changes in the DSL. If we want to add or remove new game rules, or subtly change the behavior of existing rules, we would need to tune these heuristics again.

In this section, we introduce our novel technique for performing time-series inference for our game engine. Our method builds on previous literature [45–48] for using neural density estimators for simulator-based inference, and extends them to the online, time-series domain. We leverage the ability to generate high-quality training examples from our forward model to train a data-driven neural proposal that can serve as high-quality initializations for the classical algorithms we described in the previous section.

4.3.1 Independence and Windowing

Variables inside the latent space are correlated. For instance, if we observe an object disappear, that object must have a `Perishable` component and another object must have the `Killer` component. For game-like environments, we make a simplifying assumption that these correlations would be observed in windows of W time-steps and that across these windows, we can treat each rule as independent random variables. Note that this is not true for our generative model, but allows us to aggregate distributions over time in an online manner by simply adding logits across multiple time windows.

4.3.2 Neural Conditional Density Estimation

We want our proposal neural network must predict distributions over the latents, as opposed to point estimates, conditioned on an observation sequence of W time-steps. For discrete properties, we can simply interpret the output from a softmax as a probability mass function. For continuous latents we follow Papamakarios et al. [46], and make use the Masked Autoregressive Flow (MAF) [49] architecture. MAFs allow us to condition the network, produce sample from the predicted continuous distributions, and evaluate PDFs.

The input to our neural network is a set of bounding box observations for each object slot. We treat this input as a graph, where nodes are $W \times F$ tensors representing the trajectory of that bounding box. Each tensor here also has the entire sequence of actions that were executed in the observation sequence. Therefore, F is the number of dimensions needed to express a bounding box and a one hot encoding of the action. As shown in Figure 4-2, the input graph is fully connected to capture predicting causal dynamics between objects.

The input graph is then fed into a Temporal Graph Convolutional Network (TGCN). We loosely follow the architecture laid out by Zhao et al. [50] from traffic prediction literature, and Yan et al. [51] from the skeleton-based action recognition literature, since the spatio-temporal nature of our problem is very similar. For each

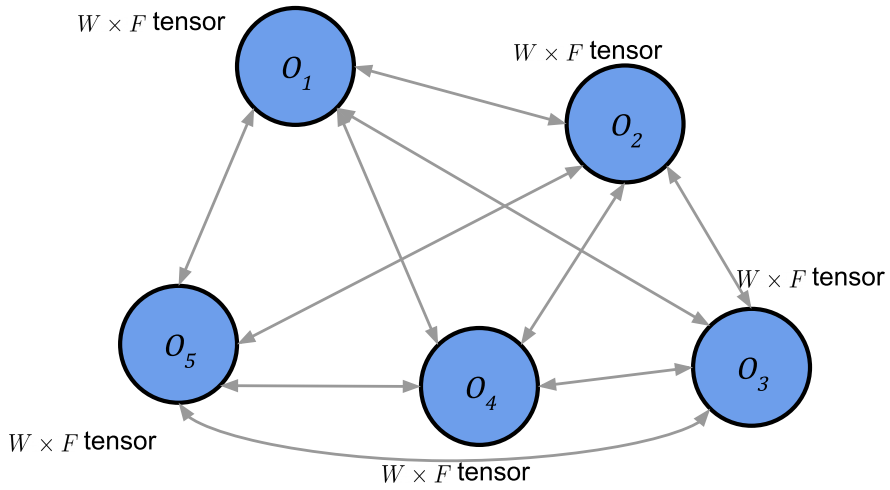


Figure 4-2: Input graph to the neural network. Each node is a bounding box observation. The graph is directed and fully connected.

edge, we have a convolutional edge encoder, $\phi_e(O_i, O_j)$ that produces directed messages. The messages are aggregated by taking their sum to produce node-level edge embeddings, g_e^i. We also pass node features into a node-level encoder, $\phi_n(O_i)$ to produce node-level node embeddings, g_n^i. Finally, the two embeddings are concatenated to produce node-level embeddings for each node, $g^i = [g_e^i, g_n^i]$.

The schema interface as described in Chapter 2 allows us to automatically attach prediction heads on top of these embeddings for each property we want to predict. Figure 4-3 shows the overall neural architecture. For each continuous property, we attach a MAF prediction head and for each discrete property, we attach an MLP with a softmax prediction output layer. We will refer to our neural network as $q_\phi(z_0|x_{0:W})$, since we can concatenate independent samples from each property head to reconstruct a latent vector. Evaluating the log density follows similarly, by adding the log density of each prediction head. Since MAFs can fit arbitrary distribution, we noticed that we were able to easily overfit the data. We add small Gaussian noise to the targets while training to regularize.

In practice, we noticed that some properties were much easier to train than others. For starters, predicting `Position` is equivalent to learning the identity

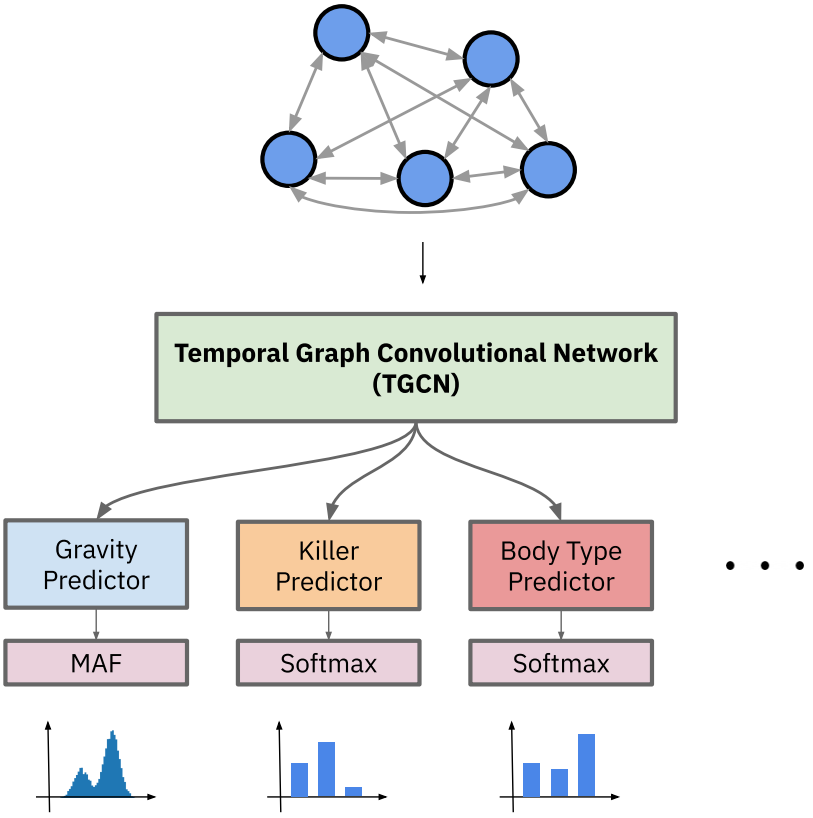


Figure 4-3: Overview of the proposal neural network architecture. The input is a spatio-temporal graph as illustrated by Figure 4-2. The TGCN module produces node-level embeddings after message passing. These embeddings are then used by individual property heads to make density predictions.

function since a noisy version of the position is directly observed. However, predicting something like `Gravity` is much harder since it relies on estimating second-order effects. This problem is well studied in the literature under the multi-task learning name. We attempted to use Gradient Surgery [52] and GradNorm [53], but discovered a significant performance impact from using these techniques. Instead, we manually label properties into difficulty categories, where each category has its own TGCN, and categories don't share weights. For the components described in Table 2.1, we were able to split them up into 4 difficulty categories.

Let $z = z_0$, and $X = x_{0:W}$. To train the proposal neural network, we follow Le

et al. [54] and minimize the Kullback–Leibler divergence, $D_{\mathsf{KL}}\left(p(z|X) \parallel q_\phi(z|X)\right)$,

$$\begin{aligned}\mathcal{L}_\phi &= \mathbb{E}_{p(X)}\left[D_{\mathsf{KL}}\left(p(z|X) \parallel q_\phi(z|X)\right)\right] \\ &= \int_X p(X) \int_z p(z|X) \log \frac{p(z|X)}{q_\phi(z|X)} dz dX \\ &\geq \mathbb{E}_{p(X,z)}\left[-\log q_\phi(z|X)\right]\end{aligned}$$

Therefore, to generate training data, we can sample a z_0 from the prior, unroll it for W time-steps and simulate corresponding observations, $X = [x_0, x_1, \ldots, x_W]$. We can then train the neural network using mini-batches of (X, z) pairs and optimize for the negative log-likelihood.

4.3.3 Priors and Lonely Spaces

Generating training data by sampling initial states, $z_0 \sim p(z)$ from the prior usually leads to cases where no rules can be inferred. If an object is simply moving in free space, we cannot infer any rules about it: we do not know if this object is a `Perishable`, or a `Killer` or any other causal rules. This naïve training data generation leads to us training the proposal neural network with mostly noise, which is unlikely to learn useful data-driven heuristics.

To alleviate this issue we add a notion of *component flags* to our game engine. After a round of simulation, if a particular component's rule was executed, that component's flag is set. At any point, we can reset these flags. Then, while generating training data, we can filter for training examples where some collection of component flags were set. This sort of post-filtering of the training data ensures that there is something learnable going on in each example and that we are not training merely with noise. Of course, we still want the neural network to understand when a property is not inferrable, and include a small percentage of examples where rules cannot be inferred.

4.3.4 Contrastive Learning of Likelihood

Training a density estimator as described above bounds the neural network to have a strong dependence on the prior. While we assume a uniform prior, filtering training examples makes this prior non-uniform. For instance, if the prior probability of an object having `Perishable` component is 0.7, the network will default to outputting 0.7 if the observation sequence is uninformative about `Perishable`. Since we aggregate the posteriors by multiplying each window of observations, this would end up multiplying a biased prior for every time-step.

To fix this issue we use a contrastive based loss as described in Greenberg et al. [47] to train our neural network. We sample a mini-batch of M latents, $Z = \{z^1, z^2, \ldots, z^M\}$, and optimize the objective,

$$\mathcal{L} = \log \frac{q_\phi(z|X)/p(z)}{\sum_{z' \in Z} q_\phi(z'|X)/p(z')}$$

Where $p(z) = p(z')$ is the original intended uniform prior, and can be cancelled,

$$\mathcal{L} = \log \frac{q_\phi(z|X)}{\sum_{z' \in Z} q_\phi(z'|X)} \tag{4.6}$$

Training the network using a contrastive loss as in Equation 4.6 yields us a neural network that can produce samples from the posterior if the prior was uninformative, even if we perform training example filtering.

4.3.5 Importance Sampling

We treat the output of the neural network as a heuristic and use importance sampling on the output to refine the prediction using simulation. For every sample from the neural network, we evaluate the likelihood as in Equation 4.3 and 4.5 by performing a simulation of the trajectory and update the importance weight of the sample accordingly. The samples are resampled using the importance weights and each property variable is bucketed to form logits which are then aggregated across multiple time windows in $O(1)$ time per window.

4.4 Qualitative Results

In this section, we evaluate the above approach on multiple real-world and synthetic environments. We note that we were not able to get successful results with the MCMC and SMC approaches, and hence explore the inference results from the perspective of the neural network and importance sampling.

4.4.1 Synthetic Examples

As a sanity check for our system, we apply the algorithm to a two-object system where we only observe the center position of the objects and need to infer their size, position, velocity, and gravity. Figure 4-4 shows the result of our technique on this system. We show the output of using the neural network without importance sampling and with importance sampling. We first note that the neural network largely performed well at estimating key properties. We also note how the importance sampling can discover multi-modality in the posterior that was undiscovered by the neural network alone.

4.4.2 Tools Challenge

Here, we apply our technique to a scene from the Tools Challenge [10]. Here, we have a complex multi-object scene with high dimensional dynamical properties such as velocity, angle, angular velocity, size, and shape.

Figure 4-5 shows qualitative results on one scene of the tools challenge. Here, we draw samples from the inferred posterior colored by their importance weight. We qualitatively note how our technique can capture uncertainty in the inference. The base block and the pillar never really move, and thus over time, our system is very certain about its position, shape, and body type. The other dynamically moving objects have higher tracking uncertainty.

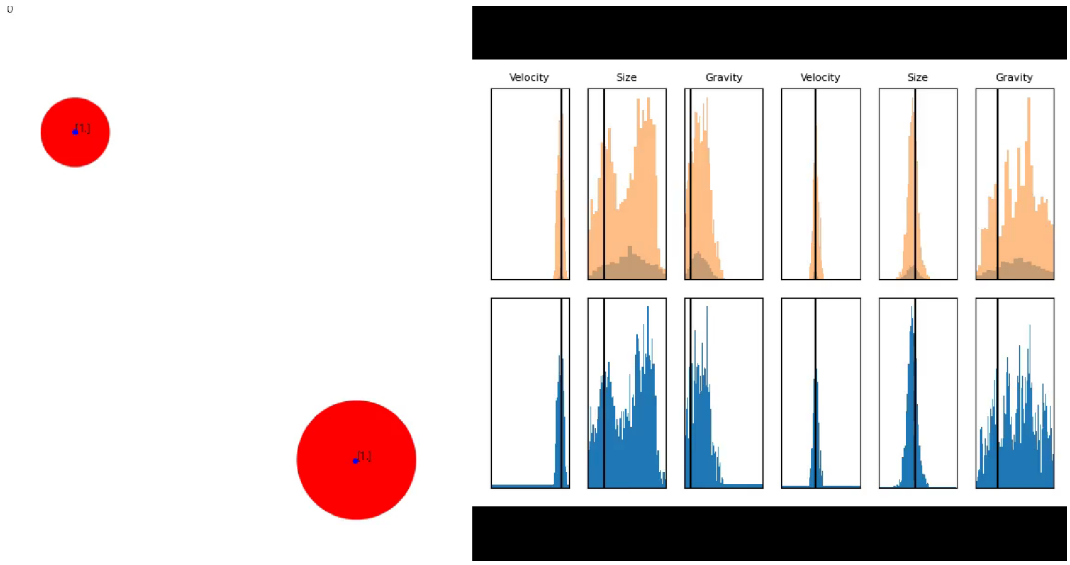


Figure 4-4: Output of multi-object inference using our technique. On the left we see the scene frozen in time. The ground truth objects are drawn in red, the blue dot is the noisy observation. On the right, we see the output of the inference. The top panel shows the output from the neural network (dark orange) and the output from importance sampling *after* the neural network. The bottom shows the posterior for each property aggregated up until the current time. Ground truth is indicated with black vertical lines.

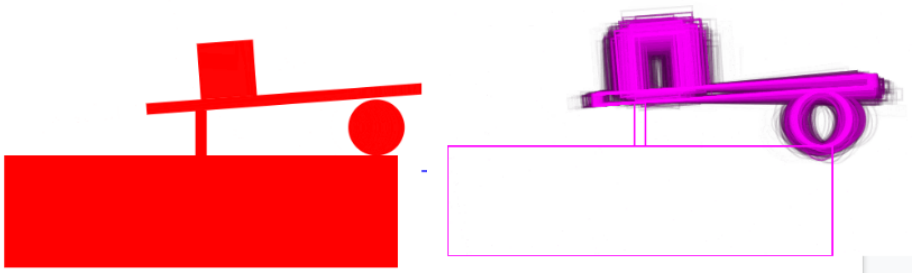
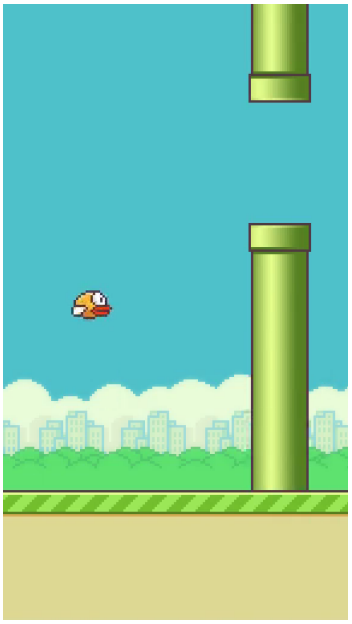


Figure 4-5: Our technique applied to a scene in the tools challenge. Instead of showing the distributions for each property, we show samples from the posterior distribution on the right. The left shows the ground truth scene frozen in time.

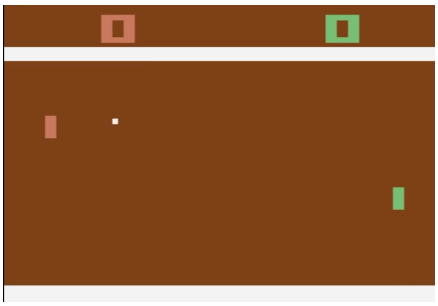


```
Bird:
  ActionSetVelocity<W_UP>(0, -0.11)
  Gravity(0.4)
  Scoreable()
  CollidingBody(..., DYNAMIC, RECT)

Ground:
  GameOver()
  CollidingBody(..., STATIC, RECT)

Pipe:
  GameOver()
  CollidingBody(..., DYNAMIC, RECT)
```

Figure 4-6: MAP output of our inference engine after observation 500 timesteps (16 seconds) of Flappy Bird gameplay.



```
PlayerBat:
  ActionSetVelocity<A_DOWN>(0, -0.10)
  ActionSetVelocity<D_DOWN>(0, 0.12)
  ActionSetVelocity<A_UP>(0, 0)
  ActionSetVelocity<D_UP>(0, 0)
  CollidingBody(..., DYNAMIC, RECT)

Ball:
  GameOverOOB<RIGHT>()
```

Figure 4-7: MAP output of our inference engine after observation 500 timesteps (16 seconds) of Pong gameplay.

4.4.3 Games

We also apply our technique to two real games, *Flappy Bird* and Atari *Pong*. Here we must infer complex rules about the causal relationship of object rules, their dynamical state, and their response to actions. Figures 4-6 and 4-7 show the maximum a posteriori sample from the inferred posterior visualized in our DSL's human-readable syntax. We have rounded numbers and omitted certain objects, parameters, and components for clarity.

Chapter 5

Conclusion

In this thesis, we introduced novel techniques to infer structured world models from pixel observations in the form of video. To our knowledge, this is the first system that can perform online inference and state estimation at the scale of our generative model, inferring rich causal game rules and game state. While this work is very early and has strong assumptions that need further research, we believe that this work lays important groundwork for data-efficient reinforcement learning pipelines since we can quickly build world models that can be leveraged for downstream planning.

5.1 Future Work

5.1.1 Rigorous Evaluation on a Diverse Set of Tasks

In this thesis, we limited our evaluation to qualitative evaluation on simpler game environments. In the future, we would like to perform a much more rigorous evaluation, both quantitatively and qualitatively on a diverse set of tasks and games.

5.1.2 Planning

The motivation for this work is to achieve human-efficient reinforcement learning. To get to this goal, we require to be able to leverage our learned model for deciding which action to take via planning. Our current work implements a very simple breadth-first search-based planner that we did not discuss in detail. Future work should rigorously evaluate if inferred models can be used for planning algorithms to produce better data efficiency.

5.1.3 Loosing The Game Manual Assumption

While the game manual assumption makes our perception task much easier, we would ideally like to not leverage having access to the game spite-sheet. We would like to explore the possibility of automatically extracting sprites through generative modeling akin to Smirnov et al. [32], where object-based inductive biases help discover objects. Being able to couple perception with the game engine is also a future direction that is worth exploring.

5.1.4 Differentiable Game Description Language

A big challenge for inference is that our game engine is treated as a black box, confining us to black-box inference techniques. We would like to explore the possibility of building a differentiable game engine that could allow us to use more scalable inference methods such as Hamiltonian Monte-Carlo and Variational Inference.

Bibliography

[1] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *arXiv preprint arXiv:1911.08265*, 2019.

[2] Pedro A Tsividis, Thomas Pouncy, Jacqueline L Xu, Joshua B Tenenbaum, and Samuel J Gershman. Human learning in atari. 2017.

[3] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[4] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, et al. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374*, 2019.

[5] Nicholas Watters, Loic Matthey, Matko Bosnjak, Christopher P Burgess, and Alexander Lerchner. Cobra: Data-efficient model-based rl through unsupervised object discovery and curiosity-driven exploration. *arXiv preprint arXiv:1905.09275*, 2019.

[6] Alex X Lee, Richard Zhang, Frederik Ebert, Pieter Abbeel, Chelsea Finn, and Sergey Levine. Stochastic adversarial video prediction. *arXiv preprint arXiv:1804.01523*, 2018.

[7] Michael Janner, Sergey Levine, William T. Freeman, Joshua B. Tenenbaum, Chelsea Finn, and Jiajun Wu. Reasoning about physical interactions with object-oriented prediction and planning. In *International Conference on Learning Representations*, 2019.

[8] Rishi Veerapaneni, John D Co-Reyes, Michael Chang, Michael Janner, Chelsea Finn, Jiajun Wu, Joshua Tenenbaum, and Sergey Levine. Entity abstraction in visual model-based reinforcement learning. In *Conference on Robot Learning*, pages 1439–1456. PMLR, 2020.

[9] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[10] Kelsey R Allen, Kevin A Smith, and Joshua B Tenenbaum. The tools challenge: Rapid trial-and-error learning in physical problem solving. *arXiv preprint arXiv:1907.09620*, 2019.

[11] Chelsea Finn, Ian Goodfellow, and Sergey Levine. Unsupervised learning for physical interaction through video prediction. *arXiv preprint arXiv:1605.07157*, 2016.

[12] Alex X Lee, Richard Zhang, Frederik Ebert, Pieter Abbeel, Chelsea Finn, and Sergey Levine. Stochastic adversarial video prediction. *arXiv preprint arXiv:1804.01523*, 2018.

[13] Adina L Roskies. The binding problem. *Neuron*, 24(1):7–9, 1999.

[14] Francesco Locatello, Stefan Bauer, Mario Lucic, Gunnar Raetsch, Sylvain Gelly, Bernhard Schölkopf, and Olivier Bachem. Challenging common assumptions in the unsupervised learning of disentangled representations. In *international conference on machine learning*, pages 4114–4124. PMLR, 2019.

[15] Christopher P Burgess, Loic Matthey, Nicholas Watters, Rishabh Kabra, Irina Higgins, Matt Botvinick, and Alexander Lerchner. Monet: Unsupervised scene decomposition and representation. *arXiv preprint arXiv:1901.11390*, 2019.

[16] Klaus Greff, Raphaël Lopez Kaufman, Rishabh Kabra, Nick Watters, Chris Burgess, Daniel Zoran, Loic Matthey, Matthew Botvinick, and Alexander Lerchner. Multi-object representation learning with iterative variational inference. *arXiv preprint arXiv:1903.00450*, 2019.

[17] Francesco Locatello, Dirk Weissenborn, Thomas Unterthiner, Aravindh Mahendran, Georg Heigold, Jakob Uszkoreit, Alexey Dosovitskiy, and Thomas Kipf. Object-centric learning with slot attention. *Advances in Neural Information Processing Systems*, 33, 2020.

[18] Tejas D Kulkarni, Ankush Gupta, Catalin Ionescu, Sebastian Borgeaud, Malcolm Reynolds, Andrew Zisserman, and Volodymyr Mnih. Unsupervised learning of object keypoints for perception and control. *Advances in neural information processing systems*, 32, 2019.

[19] Thomas Kipf, Elise van der Pol, and Max Welling. Contrastive learning of structured world models. *arXiv preprint arXiv:1911.12247*, 2019.

[20] Maxwell I Nye, Armando Solar-Lezama, Joshua B Tenenbaum, and Brenden M Lake. Learning compositional rules via neural program synthesis. *arXiv preprint arXiv:2003.05562*, 2020.

[21] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. *Advances in neural information processing systems*, 31:6059–6068, 2018.

[22] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv preprint arXiv:2006.08381*, 2020.

[23] Yujia Li, Felix Gimeno, Pushmeet Kohli, and Oriol Vinyals. Strong generalization and efficiency in neural programs. *arXiv preprint arXiv:2007.03629*, 2020.

[24] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. *arXiv preprint arXiv:1804.02477*, 2018.

[25] Pedro A Tsividis, Joao Loula, Jake Burga, Nathan Foss, Andres Campero, Thomas Pouncy, Samuel J Gershman, and Joshua B Tenenbaum. Human-level reinforcement learning through theory-based modeling, exploration, and planning. *arXiv preprint arXiv:2107.12544*, 2021.

[26] Tomer Ullman, Andreas Stuhlmuller, Noah Goodman, and Josh Tenenbaum. Learning physical theories from dynamical scenes. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 36, 2014.

[27] Toni Härkönen. Advantages and implementation of entity-component-systems. 2019.

[28] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6, 2015.

[29] Slembcke. Slembcke/chipmunk2d: A fast and lightweight 2d game physics library. URL `https://github.com/slembcke/Chipmunk2D`.

[30] Erin Catto. Box2d: A 2d physics engine for games. 2011. URL `http://www.box2d.org`.

[31] Erwin Coumans. Bullet physics simulation. In *ACM SIGGRAPH 2015 Courses*, page 1. 2015.

[32] Dmitriy Smirnov, Michael Gharbi, Matthew Fisher, Vitor Guizilini, Alexei Efros, and Justin M Solomon. Marionette: Self-supervised sprite learning. *Advances in Neural Information Processing Systems*, 34, 2021.

[33] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28, 2015.

[34] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[35] Li Zhang, Yuan Li, and Ramakant Nevatia. Global data association for multi-object tracking using network flows. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2008.

[36] Guillem Brasó and Laura Leal-Taixé. Learning a neural solver for multiple object tracking. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6247–6257, 2020.

[37] Michael I Jordan, Zoubin Ghahramani, Tommi S Jaakkola, and Lawrence K Saul. An introduction to variational methods for graphical models. *Machine learning*, 37(2):183–233, 1999.

[38] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. Variational inference: A review for statisticians. *Journal of the American statistical Association*, 112 (518):859–877, 2017.

[39] Rajesh Ranganath, Sean Gerrish, and David Blei. Black box variational inference. In *Artificial intelligence and statistics*, pages 814–822. PMLR, 2014.

[40] Michael Betancourt. A conceptual introduction to hamiltonian monte carlo. *arXiv preprint arXiv:1701.02434*, 2017.

[41] Arnaud Doucet, Nando De Freitas, Neil James Gordon, et al. *Sequential Monte Carlo methods in practice*, volume 1. Springer, 2001.

[42] Kevin Smith, Lingjie Mei, Shunyu Yao, Jiajun Wu, Elizabeth Spelke, Josh Tenenbaum, and Tomer Ullman. Modeling expectation violation in intuitive physics with coarse probabilistic object representations. *Advances in neural information processing systems*, 32, 2019.

[43] Jeroen D Hol, Thomas B Schon, and Fredrik Gustafsson. On resampling algorithms for particle filters. In *2006 IEEE nonlinear statistical signal processing workshop*, pages 79–82. IEEE, 2006.

[44] Walter R Gilks and Carlo Berzuini. Following a moving target—monte carlo inference for dynamic bayesian models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(1):127–146, 2001.

[45] George Papamakarios and Iain Murray. Fast ε-free inference of simulation models with bayesian conditional density estimation. *Advances in neural information processing systems*, 29, 2016.

[46] George Papamakarios, David Sterratt, and Iain Murray. Sequential neural likelihood: Fast likelihood-free inference with autoregressive flows. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 837–848. PMLR, 2019.

[47] David Greenberg, Marcel Nonnenmacher, and Jakob Macke. Automatic posterior transformation for likelihood-free inference. In *International Conference on Machine Learning*, pages 2404–2414. PMLR, 2019.

[48] Owen Thomas, Ritabrata Dutta, Jukka Corander, Samuel Kaski, and Michael U Gutmann. Likelihood-free inference by ratio estimation. *Bayesian Analysis*, 17(1):1–31, 2022.

[49] George Papamakarios, Theo Pavlakou, and Iain Murray. Masked autoregressive flow for density estimation. *Advances in neural information processing systems*, 30, 2017.

[50] Ling Zhao, Yujiao Song, Chao Zhang, Yu Liu, Pu Wang, Tao Lin, Min Deng, and Haifeng Li. T-gcn: A temporal graph convolutional network for traffic prediction. *IEEE Transactions on Intelligent Transportation Systems*, 21(9): 3848–3858, 2019.

[51] Sijie Yan, Yuanjun Xiong, and Dahua Lin. Spatial temporal graph convolutional networks for skeleton-based action recognition. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

[52] Tianhe Yu, Saurabh Kumar, Abhishek Gupta, Sergey Levine, Karol Hausman, and Chelsea Finn. Gradient surgery for multi-task learning. *Advances in Neural Information Processing Systems*, 33:5824–5836, 2020.

[53] Zhao Chen, Vijay Badrinarayanan, Chen-Yu Lee, and Andrew Rabinovich. Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks. In *International Conference on Machine Learning*, pages 794–803. PMLR, 2018.

[54] Tuan Anh Le, Atilim Gunes Baydin, and Frank Wood. Inference compilation and universal probabilistic programming. In *Artificial Intelligence and Statistics*, pages 1338–1348. PMLR, 2017.