# Understanding and Improving Computer Vision and Spatiotemporal Deep Neural Networks

by

Michael A. Alcorn

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
August 7, 2021

Keywords: Multi-agent spatiotemporal modeling, Transformers, Deep learning, Representation learning, Distribution estimation, Adversarial machine learning

Copyright 2021 by Michael A. Alcorn

Approved by

Anh Nguyen (Chair), Assistant Professor of Computer Science and Software Engineering
Jan Van Haaren, Research Fellow at Katholieke Universiteit Leuven
Hans-Werner Van Wyk, Assistant Professor of Mathematics and Statistics
Levent Yilmaz, Professor of Computer Science and Software Engineering

## Abstract

Deep neural networks are increasingly being used to solve a wide range of machine learning tasks. In this dissertation, I discuss several research projects I have worked on at Auburn with the goal of understanding and improving deep learning architectures for image classification, multi-agent spatiotemporal modeling, and likelihood-based generative modeling.

**(1)** Despite excellent performance on stationary test sets, deep neural networks (DNNs) can fail to generalize to out-of-distribution (OoD) inputs, including natural, non-adversarial ones, which are common in real-world settings. In the second chapter of this dissertation, we present a framework for discovering DNN failures that harnesses 3D renderers and 3D models. That is, we estimate the parameters of a 3D renderer that cause a target DNN to misbehave in response to the rendered image. Using our framework and a self-assembled dataset of 3D objects, we investigate the vulnerability of DNNs to OoD poses of well-known objects in ImageNet. For objects that are readily recognized by DNNs in their canonical poses, DNNs incorrectly classify 97% of their pose space. In addition, DNNs are highly sensitive to slight pose perturbations. Importantly, adversarial poses transfer across models and datasets. We find that 99.9% and 99.4% of the poses misclassified by Inception-v3 also transfer to the AlexNet and ResNet-50 image classifiers trained on the same ImageNet dataset, respectively, and 75.5% transfer to the YOLOv3 object detector trained on MS COCO. **(2)** In the third chapter of this dissertation, we show that the covariance matrix adaptation evolutionary strategy algorithm is much more effective than finite differences for discovering adversarial poses of objects, outperforming finite differences in 94% of trials.

**(3)** Multi-agent spatiotemporal modeling is a challenging task from both an algorithmic design and computational complexity perspective. Recent work has explored the efficacy of

traditional deep sequential models in this domain, but these architectures are slow and cumbersome to train, particularly as model size increases. Further, prior attempts to model interactions between agents across time have limitations, such as imposing an order on the agents, or making assumptions about their relationships. In the fourth chapter of this dissertation, we introduce `baller2vec`[1], a multi-entity generalization of the standard Transformer that can, with minimal assumptions, *simultaneously and efficiently* integrate information across entities and time. We test the effectiveness of `baller2vec` for multi-agent spatiotemporal modeling by training it to perform two different basketball-related tasks: (1) simultaneously forecasting the trajectories of all players on the court and (2) forecasting the trajectory of the ball. Not only does `baller2vec` learn to perform these tasks well (outperforming a graph recurrent neural network with a similar number of parameters by a wide margin), it also appears to "understand" the game of basketball, encoding idiosyncratic qualities of players in its embeddings, and performing basketball-relevant functions with its attention heads.

**(4)** In many multi-agent spatiotemporal systems, the agents are under the influence of shared, *unobserved* variables (e.g., the play a team is executing in a game of basketball). As a result, the trajectories of the agents are often statistically *dependent* at any given time step; however, almost universally, multi-agent models implicitly assume the agents' trajectories are statistically *independent* at each time step. In the fifth chapter of this dissertation, we introduce `baller2vec++`[2], a multi-entity Transformer that can effectively model coordinated agents. Specifically, `baller2vec++` applies a specially designed self-attention mask to a *mixture* of location *and* "look-ahead" trajectory sequences to learn the distributions of statistically dependent agent trajectories. We show that, unlike `baller2vec`, `baller2vec++` can learn to emulate the behavior of perfectly coordinated agents in a simulated toy dataset. Additionally, when modeling the trajectories of professional basketball players, `baller2vec++` outperforms `baller2vec` by a wide margin.

[1] All data and code for the chapter are available at: `https://github.com/airalcorn2/baller2vec`.

[2] All data and code for the chapter are available at: `https://github.com/airalcorn2/baller2vecplusplus`.

**(5)** Order-agnostic autoregressive distribution (density) estimation (OADE), i.e., autoregressive distribution estimation where the features can occur in an arbitrary order, is a challenging problem in generative machine learning. Prior work on OADE has encoded feature identity by assigning each feature to a distinct fixed position in an input vector. As a result, architectures built for these inputs must strategically mask either the input or model weights to learn the various conditional distributions necessary for inferring the full joint distribution of the dataset in an order-agnostic way. In the sixth chapter of this dissertation, we propose an alternative approach for encoding feature identities, where each feature's identity is included *alongside* its value in the input. This feature identity encoding strategy allows neural architectures designed for sequential data to be applied to the OADE task without modification. As a proof of concept, we show that a Transformer trained on this input (which we refer to as "the DEformer"[3], i.e., the distribution estimating Transformer) can effectively model binarized-MNIST, approaching the performance of fixed-order autoregressive distribution estimating algorithms while still being entirely order-agnostic. Additionally, we find that the DEformer surpasses the performance of recent flow-based architectures when modeling a tabular dataset.

[3]All data and code for the chapter are available at: `https://github.com/airalcorn2/deformer`.

# Table of Contents


Abstract . . . . . ii

List of Figures . . . . . ix

List of Tables . . . . . xx

1 Introduction . . . . . 1

2 Strike (with) a Pose: Neural Networks Are Easily Fooled by Strange Poses of Familiar Objects . . . . . 5

2.1 Introduction . . . . . 5

2.2 Framework . . . . . 7

2.2.1 Problem formulation . . . . . 7

2.2.2 Classification networks . . . . . 9

2.2.3 3D renderers . . . . . 9

2.2.4 3D object dataset . . . . . 9

2.2.5 Background images . . . . . 11

2.3 Methods . . . . . 11

2.3.1 Pose transformations . . . . . 11

2.3.2 Random search . . . . . 12

2.3.3 $z_\delta$-constrained random search . . . . . 12

2.3.4 Gradient descent with finite-difference . . . . . 13

2.4 Experiments and results . . . . . 14

2.4.1 Neural networks are easily confused by object rotations and translations 14

2.4.2 Common object classifications are shared across different lighting settings 15

2.4.3 Correct classifications are highly localized in the rotation and translation landscape . . . . . 17

2.4.4 Optimization methods can effectively generate targeted adversarial poses 18
2.4.5 Adversarial poses transfer to different image classifiers and object detectors . . . 19
2.4.6 Adversarial training . . . 20
2.5 Related work . . . 21
2.6 Discussion and conclusion . . . 22
3 Using Covariance Matrix Adaptation Evolution Strategy to Discover Adversarial Poses for Artificial Neural Networks . . . 24
3.1 Introduction . . . 24
3.2 Covariance matrix adaptation evolution strategy . . . 25
3.2.1 Updating $m$ . . . 26
3.2.2 Updating $C$ . . . 27
3.2.3 Updating $\sigma$ . . . 30
3.3 Experiments and results . . . 32
4 `baller2vec`: A Multi-Entity Transformer For Multi-Agent Spatiotemporal Modeling . . . 34
4.1 Introduction . . . 34
4.2 Methods . . . 36
4.2.1 Multi-entity sequences . . . 36
4.2.2 Multi-agent spatiotemporal modeling . . . 37
4.2.3 The multi-entity Transformer . . . 39
4.3 Experiments . . . 40
4.3.1 Dataset . . . 40
4.3.2 Model . . . 41
4.3.3 Baselines . . . 42
4.3.4 Ablation studies . . . 43
4.4 Results . . . 44

4.4.1 `baller2vec` is an effective learning algorithm for multi-agent spatiotemporal modeling. . . . . 44
4.4.2 `baller2vec` uses information about all players on the court through time, in addition to player identity, to model spatiotemporal dynamics. 45
4.4.3 `baller2vec`'s learned player embeddings encode individual attributes. 46
4.4.4 `baller2vec`'s trajectory forecast distributions depend on both the historical and current context. . . . . 47
4.4.5 Attention heads in `baller2vec` appear to perform basketball-relevant functions. . . . . 48
4.5 Related Work . . . . . 50
4.5.1 Trajectory modeling in sports . . . . . 50
4.5.2 Transformers for multi-agent spatiotemporal modeling . . . . . 51
4.6 Limitations . . . . . 51
4.7 Conclusion . . . . . 53
5 `baller2vec++`: A Look-Ahead Multi-Entity Transformer For Modeling Coordinated Agents . . . . . 54
5.1 Introduction and Related Work . . . . . 54
5.2 Background . . . . . 56
5.2.1 Multi-agent trajectory modelling . . . . . 56
5.2.2 `baller2vec` is a (conditional) generative model. . . . . 57
5.3 Model Architecture . . . . . 61
5.4 Experiments . . . . . 64
5.4.1 Toy dataset . . . . . 65
5.4.2 Basketball dataset . . . . . 65
5.4.3 Model . . . . . 65
5.5 Results . . . . . 66

5.5.1 `baller2vec++` can effectively model coordinated agents in both simulated and real settings . . . . . 66
5.5.2 `baller2vec++` makes better forecasts when conditioned on concurrent trajectory information from other agents . . . . . 67
5.5.3 The joint probability assigned to a sequence by `baller2vec++` is approximately permutation invariant with respect to the order of the agents . . . . . 68
5.6 Conclusion and Future Work . . . . . 69
6 The DEformer: An Order-Agnostic Distribution Estimating Transformer . . . . 71
6.1 Introduction . . . . . 71
6.2 Architecture . . . . . 72
6.3 Experiments . . . . . 75
6.4 Results . . . . . 77
6.5 Related Work . . . . . 79
6.5.1 Interleaved input Transformers . . . . . 79
6.5.2 DeepNADE . . . . . 80
6.5.3 Spatial inputs as feature identities . . . . . 81
6.6 Conclusion . . . . . 82
Bibliography . . . . . 83

# List of Figures

1.1 A typical convolutional neural network displaying both the hierarchical design common among deep learning architectures and the inductive biases encoded by convolutional layers. Applying a convolution filter (here, a gray-outlined square overlaid on a portion of the image or feature map) to an input is equivalent to multiplying the *entire* (flattened) input by a single weight matrix, where: (1) the weight matrix is highly sparse and follows a highly regimented sparsity pattern, and (2) non-zero parameters are *shared* across rows of the weight matrix [32]. In practice, the inductive biases of convolutional neural networks make them highly effective at both extracting low-level image features that are information rich, and combining these extracted features to represent increasingly complex concepts [67, 74]. Image source: `https://commons.wikimedia.org/wiki/File:Typical_cnn.png`. . . . . 2

1.2 **Left**: a parse tree of a sentence based on constituency grammars. **Right**: a simple recursive neural network depicting how feature vectors corresponding to nodes of a tree-like data structure are combined and transformed to generate a feature vector representing a parent node. The basic operation in a recursive neural network is $p_{1,2} = \sigma(W[c_1; c_2] + b)$ where $c_1$ and $c_2$ are $d$-dimensional feature vectors for the input (child) nodes, $W$ is a $d \times 2d$ weight matrix, $b$ is a $d$-dimensional bias vector, $\sigma$ is a nonlinearity, and $p_{1,2}$ is the $d$-dimensional output representation for the parent node of the input nodes. Image sources: `https://commons.wikimedia.org/wiki/File:Parse_tree_1.jpg` and `https://commons.wikimedia.org/wiki/File:Simple_recursive_neural_network.svg`. 3

2.1 The Google Inception-v3 classifier [117] correctly labels the canonical poses of objects (a), but fails to recognize out-of-distribution images of objects in unusual poses (b–d), including real photographs retrieved from the Internet (d). The left $3 \times 3$ images (a–c) are found by our framework and rendered via a 3D renderer. Below each image are its top-1 predicted label and confidence score. . . . . 6

2.2 To test a target DNN, we build a 3D scene (a) that consists of 3D objects (here, a school bus and a pedestrian), lighting, a background scene, and camera parameters. Our 3D renderer renders the scene into a 2D image, which the image classifier labels `school bus`. We can estimate the pose changes of the school bus that cause the classifier to misclassify by (1) approximating gradients via finite differences; or (2) backpropagating (red dashed line) through a differentiable renderer. . . . . 8

2.3 The distributions of individual pose parameters for (a) high-confidence ($p \geq 0.7$) incorrect classifications and (b) correct classifications obtained from the random sampling procedure described in Sec. 2.3.2. $x_\delta$ and $y_\delta$ have been normalized w.r.t. their corresponding $s$ from Eq. (2.4). . . . . 14

2.4 Inception-v3's ability to correctly classify images is highly localized in the rotation and translation parameter space. (a) The classification landscape for 15 vehicle objects when altering $\theta_r$ and $\theta_p$ and holding $(x_\delta, y_\delta, z_\delta, \theta_y)$ at $(0, 0, -3, \frac{\pi}{4})$. Lighter regions correspond to poses with a greater number of correctly classified vehicle objects. Green and red circles indicate correct and incorrect classifications, respectively, corresponding to the fire truck object poses found in (b). . . 16

3.1 The Google Inception-v3 neural network [118] correctly recognizes a 3D object of a school bus in a "canonical pose" (a) but is confused by the same object in an adversarial pose (b). . . . . 25

3.2 Pseudocode for the CMA-ES algorithm (image source: `https://en.wikipedia.org/wiki/CMA-ES`). . . . . 26

3.3 The distribution of candidate solutions when using the covariance matrix update from (3.4) (top row) or (3.6) (bottom row). Notice how (3.6) *decreases* the variance of candidate solutions in the direction that improves the cost function. Image source: [51]. . . . . 28

3.4 Example evolution paths of six steps (image source: [51]). . . . . 31

3.5 Target probabilities for the CMA-ES and FD algorithms. Each point represents a different trial. . . . . 33

4.1 After solely being trained to predict the trajectory of the ball (●) given the locations of the players and the ball on the court through time, a self-attention (SA) head in `baller2vec` learned to anticipate passes. When the ball handler (■●) is driving towards the basket at $t = 4$, SA assigns near-zero weights (black) to all players, suggesting no passes will be made. Indeed, the ball handler did not pass and dribbled into the lane ($t = 8$). SA then assigns a high weight (white) to a teammate (□), which correctly identifies the recipient of the pass at $t = 13$. 34

4.2 An example of a binned trajectory. The agent's starting position is at the center of the grid, and the cell containing the agent's ending position is used as the label (of which there are $n^2$ possibilities). . . . . 37

4.3 An overview of our multi-entity Transformer, `baller2vec`. Each time step $t$ consists of an *unordered* set $Z_t$ of entity feature vectors (colored circles) as the input, with either (**left**) a corresponding set $V_t$ of entity labels (colored diamonds) or (**right**) a single label $w_t$ (gray triangle) as the target. Matching colored circles/diamonds across time steps correspond to the same entity. In our experiments, each entity feature vector $z_{t,k}$ is produced by an MLP $g$ that takes a player's identity embedding $e(p_k)$, raw court coordinates $(x_{t,k}, y_{t,k})$, and a binary variable indicating the player's frontcourt $h_{t,k}$ as input. Further, each entity label $v_{t,k}$ is an integer indexing the trajectory bin derived from the player's raw trajectory, while each $w_t$ is an integer indexing the ball's trajectory bin. . . . . 38

4.4 **Left**: the standard self-attention mask matrix $M$. The element at $M_{t_1,t_2}$ indicates whether or not the model can "look" at time step $t_2$ when processing time step $t_1$. **Right**: the matrix form of our multi-entity self-attention mask tensor. In tensor form, element $M_{t_1,k_1,t_2,k_2}$ indicates whether or not the model can "look" at agent $p_{k_2}$ at time step $t_2$ when processing agent $p_{k_1}$ at time step $t_1$. In matrix form, this corresponds to element $M_{t_1K+k_1,t_2K+k_2}$ when using zero-based indexing. The $M$ shown here is for a static, fully connected graph, but other, potentially evolving network structures can be encoded in the attention mask tensor. . . . . 39

4.5 A log-normalized heat map of the trajectory bins in the full NBA dataset. The center bin corresponds to the "stationary" trajectory (i.e., the player did not move further than 0.5 ft in either the $x$ or $y$ direction). The elongated shape of the distribution reflects the rectangular shape of the court. . . . . 41

4.6 As can be seen in this 2D UMAP of the player embeddings, by exclusively learning to predict the trajectory of the ball, `baller2vec` was able to infer idiosyncratic player attributes. The left-hand side of the plot contains tall post players (⊿, □), e.g., Serge Ibaka, while the right-hand side of the plot contains shorter shooting guards (☆) and point guards (+), e.g., Stephen Curry. The connecting transition region contains forwards (□, ○) and other "hybrid" players, i.e., individuals possessing both guard and post skills, e.g., LeBron James. Further, players with similar defensive abilities, measured here by the cube root of the players' blocks per minute in the 2015-2016 season [12], cluster together. . . . . 46

4.7 Nearest neighbors in `baller2vec`'s embedding space are plausible doppelgängers, such as the explosive point guards Russell Westbrook and Derrick Rose, and seven-foot tall brothers Pau and Marc Gasol. . . . . 47

4.8 `baller2vec`'s trajectory forecast distributions are affected by both the historical and current context. At $t = 1$, `baller2vec` is fairly uncertain about the target player's (▲; $k = 8$) trajectory (left grid and dotted red line; the blue-bordered center cell is the "stationary" trajectory), with most of the probability mass divided between trajectories moving towards the ball handler's sideline (right grid; black = 1.0; white = 0.0). After observing a portion of the sequence ($t = 6$), `baller2vec` becomes very certain about the target player's trajectory ($f_{6,8}$), but when the player reaches a decision point ($t = 13$), `baller2vec` becomes split between trajectories (staying still or moving towards the top of the key). ● = ball, □ = offense, ▲ = defense, and $f_{t,k} = f(\mathcal{Z})_{t,k}$. . . . . 48

4.9 With less historical information (solid lines), `baller2vec`'s generated trajectories (dotted lines) tend to look less natural. Each player is a different color, and the colored lines are their corresponding trajectories. The shapes are in the *starting* positions for the trajectories, and the ground truth location of the ball is used as input at each time step (i.e., the ball trajectories are not generated). △ = offense, □ = defense, and ● = ball. . . . . 49

4.10 The attention outputs from `baller2vec` suggest it learned basketball-relevant functions. Players are shaded according to the *sum* of the attention scores assigned to the players *through time* with reference to the ball in the current frame (recall that each player occurs multiple times in the input). Higher attention scores are lighter. For both of these attention heads, the sum of the attention scores assigned to the ball through time was small (0.01 for both the left and middle frames where the maximum is 1.00). . . . . 49

5.1 **Left**: most multi-agent systems implicitly assume the trajectories of the agents at each time step ($\Delta x_{t,k}$) are conditionally independent given the agents' previous positions ($x_{1:t,k}$). **Right**: however, the various decompositions of the joint probability of the trajectories, e.g., $p(\Delta x_{t,1})p(\Delta x_{t,2}|\Delta x_{t,1})p(\Delta x_{t,3}|\Delta x_{t,1}\Delta x_{t,2})$ (note, we omit the conditional $x_{1:t,k}$ terms for brevity), suggest more complex statistical dependencies between the agents' trajectories can exist (i.e., the independence assumption is an extremely strong one). Indeed, there are often shared unobserved variables influencing the spatiotemporal behaviors of agents—such as the play that the players on a basketball team are executing, or events occurring in a pedestrian environment—which suggests statistical dependencies between the agents' trajectories are likely. . . . . 55

5.2 At inference time, a model is not *required* to *simultaneously* generate the trajectories for all of the agents at a specific time step. An alternative strategy is to allow the model to generate the agents' trajectories one at a time, and let the model use the previously generated trajectories to inform the trajectories it generates for the remaining agents. . . . . 58

5.3 `baller2vec` can be viewed as a conditional generative model that assigns a probability to a sequence of trajectories given the initial positions of the agents. Here, we show a graphical model depiction of a `baller2vec` model that generates a sequence of one-dimensional trajectories for a single agent. Given the initial position of the agent (the circle containing $x_1$), one of $n$ different uniform distributions (the square containing $i_1$) is sampled using the mixture proportions ($\pi_i$) output by `baller2vec` ($f$). The agent's trajectory (the diamond containing $\Delta x_1$) is then sampled from the selected uniform distribution, which has bounds $-\infty < a_i < b_i < \infty$. At the start of the next time step, the agent's position is $x_2 = x_1 + \Delta x_1$. Maximizing the likelihood of `baller2vec` as a classifier over the binned trajectories is thus equivalent to maximizing its likelihood when assuming the trajectories are generated from a mixture of uniform distributions that partition the Euclidean trajectory space (see Section 5.2.2 for details). . . . . 59

5.4 A naive strategy for learning to predict statistically dependent agent trajectories is to adapt the `baller2vec` self-attention mask so that `baller2vec` can "look ahead" at future positions of agents whose trajectories are generated *prior* to the agent being processed in the current time step. However, this look-ahead self-attention mask cannot be used with multi-layer Transformers because doing so necessitates "seeing the future". For example, after the model attends to the blue agent's position at time step $t+1$ when processing the yellow agent at time step $t$, the yellow agent's resultant feature vector contains information about the blue agent's future position. As a result, when the model attends to the yellow agent while processing the blue agent at the next level, the model is seeing the future. . . . . 62

5.5 To learn statistically dependent agent trajectories, `baller2vec++` uses a specially designed self-attention mask to simultaneously process three different sets of features vectors in a single Transformer. The three sets of feature vectors consist of location feature vectors like those found in `baller2vec` ($z_{t,k}$), look-ahead trajectory feature vectors ($u_{t,k}$), and starting location feature vectors ($r_k$; not shown). As can be seen in these partial depictions of `baller2vec++` and the `baller2vec++` self-attention mask, this design allows the model to integrate information about *concurrent* agent trajectories through *multiple* Transformer layers without seeing the future. . . . . 63

5.6 When trained on a dataset of perfectly coordinated agent trajectories (a), the trajectories generated by `baller2vec` are completely *uncoordinated* (b) while the trajectories generated by `baller2vec++` are perfectly coordinated (c). Animated versions can be found in the code repository. . . . . 67

5.7 While `baller2vec` occasionally generates realistic trajectories for the red defender (b), it also makes egregious errors (c). In contrast, the trajectories generated by `baller2vec++` often seem plausible (d and e). For both the `baller2vec` and `baller2vec++` generated trajectories, the ground truth trajectories (a) for all of the non-red players were used as input at each time step. The red player was placed *last* in the player order when generating his trajectory with `baller2vec++`. Animated versions can be found in the code repository. . . . . 68

5.8 **Left**: when modeling the trajectories of professional basketball players, the performance gap between `baller2vec++` and `baller2vec` is largest at the beginning of the sequence, before shared unobserved variables can be inferred by `baller2vec`. Each bar indicates a model's average NLL over the entire test set for that particular time step. For full sequences, `baller2vec++` outperforms `baller2vec` by 8.9%. **Right**: the joint probability assigned to a sequence by `baller2vec++` is approximately permutation invariant with respect to the order of the agents. For each point, its $x$ value indicates `baller2vec++`'s average NLL for a test set sequence using the *original* order of the agents in the sequence, while its $y$ value indicates `baller2vec++`'s average NLL *for the same sequence* with the order of the agents *shuffled*. The shuffled average NLLs are highly correlated with their corresponding unshuffled average NLLs. . . . . 69

6.1 By including each feature's identity *alongside* its value in the input, sequential models can be used to perform order-agnostic autoregressive distribution estimation. The DEformer is a Transformer that uses an interleaved input design (partially depicted here with the self-attention mask) for this task. The two sets of interleaved feature vectors consist of identity feature vectors ($z_k$) and identity/value feature vectors ($u_k$), and $g_z$ and $g_u$ are their respective multilayer perceptrons. For the binarized-MNIST dataset, each feature identity $i_k$ is a tuple ($r_k$, $c_k$) where $r_k$ and $c_k$ are the row and column for the pixel indexed by $k$ in the permuted sequence, respectively, and $v_k$ is the value of the pixel (which is zero or one for binary images). For tabular data, each $i_k$ corresponds to a column, and $v_k$ is the value of the indexed column in the row. Lastly, $e$ is an identity encoding function, which is simply the identify function in the case of binarized-MNIST, and is an embedding layer for tabular data. . . . . 73

6.2 **Top**: A sample of 50 images from the test set of binarized-MNIST organized by their average NLLs according to the DEformer (starting with the lowest average NLL, 42.1, at the top left, and ending with the highest average NLL, 119.6, at the bottom right). **Bottom**: 50 generated images organized by their average NLLs according to the DEformer (starting with the lowest average NLL, 42.0, at the top left, and ending with the highest average NLL, 131.1, at the bottom right). The average NLLs for both sets of images are calculated over 10 random orderings. . . . . 78

6.3 Because the DEformer is order-agnostic, it can easily "fill in" images where pixels are missing in a variety of patterns by placing the missing pixels at the end of the input sequence. Here, each row corresponds to a different ground truth image from the test set (depicted in the first column). The remaining pairs of columns show 100 removed pixels (red) from the ground truth image and the corresponding filled in image. . . . . 79

6.4 The distribution of the DEformer average NLLs for the binarized-MNIST test set (left) and a subset of 10,000 images from the binarized-notMNIST dataset (right) diverge considerably (notice the difference in scales of the $x$-axes), i.e., the DEformer consistently assigns lower probabilities to out-of-distribution samples. 80

# List of Tables


2.1 The median of the median failure rates and the median of the median minimum disturbances (Min. $\Delta$) for the single parameter sensitivity tests described in Section 2.4.3. Int. $\Delta$ converts the values in Min. $\Delta$ to more interpretable units. For $x_\delta$ and $y_\delta$, the interpretable units are pixels. For $z_\delta$, the interpretable unit is the percent change in the area of the bounding box containing the object. . . . 18

2.2 The median percent of target hits and the median of the median target probabilities for random search (ZRS), gradient descent with finite difference gradients (FD-G), and DR gradients (DR-G). All attacks are targeted and initialized with $z_\delta$-constrained random search. †DR-G is not directly comparable to FD-G and ZRS. . . . . 19

2.3 The median percent of misclassifications (Error) and high-confidence (i.e., $p > 0.7$) misclassifications by the pre-trained AlexNet (PT) and our AlexNet trained with adversarial examples (AT) on random poses of training-set objects (T) and held-out objects (H). . . . . 21

4.1 The perplexity per trajectory on the test set when using `baller2vec` vs. the marginal distribution of the trajectory bins in the training set ("Train") for all predictions. `baller2vec` considerably reduces the uncertainty over the trajectory bins. . . . . 44

4.2 The average NLL (lower is better) on the **Task P** test set and seconds per training epoch (SPE) for `baller2vec` (`b2v`) and our GRNN. `baller2vec` trains ∼3.8 times faster per epoch compared to our GRNN, and `baller2vec` outperformed our GRNN by 10.5% when given the same amount of training time. Even when only allowed to train for half ("0.5x") and a quarter ("0.25x") as long as our GRNN, `baller2vec` outperformed our GRNN by 9.1% and 1.5%, respectively.. 44

4.3 The average NLL on the test set for each of the models in our ablation experiments (lower is better). For **Task P**, using all 10 players improved model performance by 18.0%, while using player identity improved model performance by an additional 4.4%. For **Task B**, using player identity improved model performance by 2.7%. 1/10 indicates whether one or 10 players were used as input, respectively, while I/NI indicates whether or not player identity was used, respectively. . . . 45

6.1 The average NLL on the binarized-MNIST test set for different models. Despite being entirely order agnostic ("OA"), the DEformer is competitive with PixelRNN and TraDE, which use a single fixed order ("FO"). For MADE, the model was trained on 32 different orders [125]. The average NLLs for both DeepNADE and the DEformer are calculated over 10 random orderings. . . . . 77

6.2 The average NLL on the POWER test set for different models. The DEformer surpasses the performance of recent flow-based architectures while still retaining order-agnostic and autoregressive properties. The average NLL for the DEformer is calculated over 10 random orderings. . . . . 77

# Chapter 1

# Introduction

Deep neural networks (also referred to as "deep learning models") are a family of machine learning algorithms that use hierarchical compositions of functions to map inputs to outputs [41]. The compositional design of deep neural networks allows them to exploit the underlying structure of various high-dimensional inputs, such as images or natural language, in ways that other machine learning algorithms cannot. For example, convolutional neural networks (Figure 1.1) are designed to take advantage of two properties of images: (1) the fact that pixels tend to be highly spatially correlated, i.e., knowing the color of one pixel tells you a lot about the potential colors of its neighbors, and (2) the fact that visual concepts are typically comprised of sub-concepts, e.g., a `human face` consists of two `eye`s, a `nose`, and a `mouth`. In contrast, recursive neural networks (Figure 1.2) are designed around the hypothesis that words and phrases in sentences have tree-like semantic relationships [112, 111], with groups of words in the sentence forming noun phrases and verb phrases, for example. Interestingly, the inductive biases introduced by various architectural design choices often transfer to tasks other than the ones they were designed for—e.g., the attention mechanism [43, 44, 131, 11], which was motivated by sequence learning tasks, is also useful for image processing [92, 142, 31].

This adaptability of neural networks, combined with: (1) recent advances in computing power and (2) access to much larger amounts of data than was previously available, has led to rapid progress in the state-of-the-art for a wide variety of high-dimensional machine learning problems since the early 2010s [41]. However, with this adaptability comes challenges, namely:

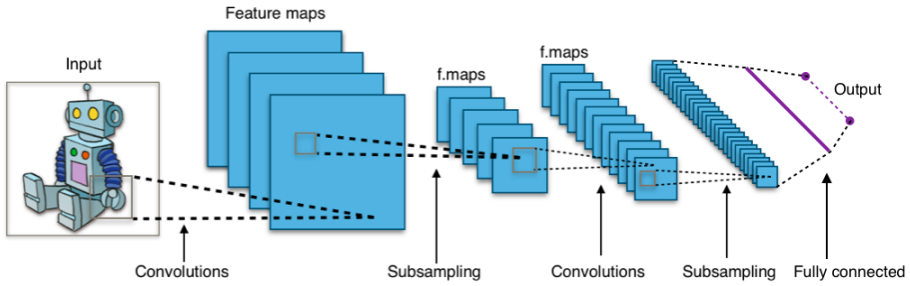



Figure 1.1: A typical convolutional neural network displaying both the hierarchical design common among deep learning architectures and the inductive biases encoded by convolutional layers. Applying a convolution filter (here, a gray-outlined square overlaid on a portion of the image or feature map) to an input is equivalent to multiplying the *entire* (flattened) input by a single weight matrix, where: (1) the weight matrix is highly sparse and follows a highly regimented sparsity pattern, and (2) non-zero parameters are *shared* across rows of the weight matrix [32]. In practice, the inductive biases of convolutional neural networks make them highly effective at both extracting low-level image features that are information rich, and combining these extracted features to represent increasingly complex concepts [67, 74]. Image source: `https://commons.wikimedia.org/wiki/File:Typical_cnn.png`.

1. The same quality that allows deep neural networks to learn highly complicated functions can also make them susceptible to unpredictable behavior when exposed to inputs that are statistically divergent from the training data (but appear innocuous to the human eye).

2. Developing new, *useful* architectures, and recognizing when inductive biases transfer between tasks, is non-trivial and typically requires a mix of intuition and trial-and-error.

Chapters 2 and 3 of this dissertation focus on the first challenge. Specifically, these chapters describe methods for discovering a surprising type of failure in state-of-the-art image classifiers—misclassifications of objects in a variety of poses. Our system uses a 3D graphics pipeline together with a variety of search strategies—including random sampling, finite differences, and covariance matrix adaptation evolution strategy (a derivative-free approach)—to describe the pose misclassification space of image classifiers. Strikingly, we find

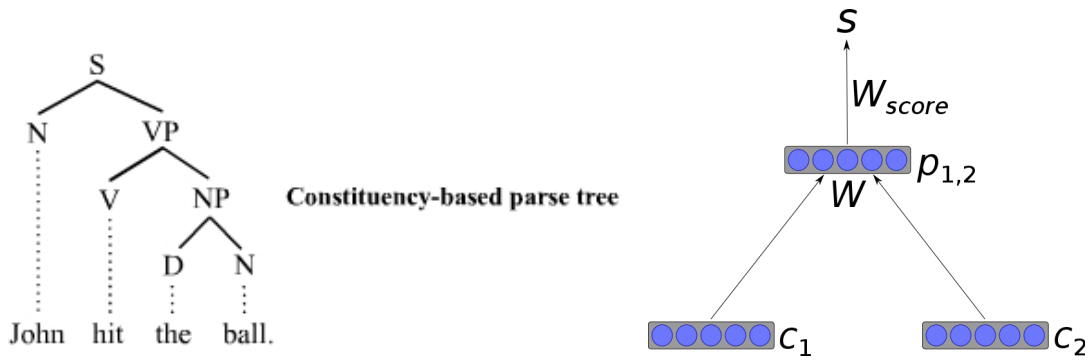



Figure 1.2: **Left**: a parse tree of a sentence based on constituency grammars. **Right**: a simple recursive neural network depicting how feature vectors corresponding to nodes of a tree-like data structure are combined and transformed to generate a feature vector representing a parent node. The basic operation in a recursive neural network is $p_{1,2} = \sigma(W[c_1; c_2] + b)$ where $c_1$ and $c_2$ are $d$-dimensional feature vectors for the input (child) nodes, $W$ is a $d \times 2d$ weight matrix, $b$ is a $d$-dimensional bias vector, $\sigma$ is a nonlinearity, and $p_{1,2}$ is the $d$-dimensional output representation for the parent node of the input nodes. Image sources: `https://commons.wikimedia.org/wiki/File:Parse_tree_1.jpg` and `https://commons.wikimedia.org/wiki/File:Simple_recursive_neural_network.svg`.

that a number of image classifiers are highly sensitive to pose, misclassifying the vast majority of the pose space for a variety of objects, and switching from a correct classification to an incorrect one following minute pose transformations.

Chapters 4 and 5 focus on the second challenge, specifically in the context of multi-agent spatiotemporal modeling. In these chapters, we describe two novel Transformer[1] [127] variants that are highly effective for modeling multi-agent systems. Specifically, our first model, `baller2vec`, extends the standard Transformer so that it can operate on sequences *of sets* (the underlying data structure of multi-agent systems), while our second model, `baller2vec++`, uses a novel interleaved input design to model the behaviors of *coordinated* agents. Using a dataset of sequences from National Basketball Association games, we show that these architectures are both much faster and much more accurate than graph recurrent neural networks when modeling multi-agent systems.

Lastly, Chapter 6 describes an example of the transferability of neural network inductive biases. Specifically, we show that the interleaved input design of `baller2vec++` can also be

[1]Transformers are a type of deep neural network (originally designed for modeling natural language) that rely heavily on the attention mechanism to process inputs.

used to perform order-agnostic distribution estimation. As a proof of concept, we use our DEformer ("distribution estimating Transformer") to model the binarized-MNIST dataset. Not only is the DEformer competitive with fixed-order approaches at modeling binarized-MNIST, but it can also effortlessly fill in pixels of images that are missing in a variety of patterns (which fixed-order models cannot do).

# Chapter 2

# Strike (with) a Pose: Neural Networks Are Easily Fooled by Strange Poses of Familiar Objects

**This chapter was published as "Strike (with) a Pose: Neural Networks Are Easily Fooled by Strange Poses of Familiar Objects" (Alcorn et al., 2019) at the 2019 Conference on Computer Vision and Pattern Recognition.**

## 2.1 Introduction

For real-world technologies, such as self-driving cars [22], autonomous drones [38], and search-and-rescue robots [106], the test distribution may be non-stationary, and new observations will often be out-of-distribution (OoD), i.e., not from the training distribution [115]. However, machine learning (ML) models frequently assign wrong labels with high confidence to OoD examples, such as adversarial examples [120, 86]—inputs specially crafted by an adversary to cause a target model to misbehave. But ML models are also vulnerable to *natural* OoD examples [69, 2, 123, 3]. For example, when a Tesla autopilot car failed to recognize a white truck against a bright-lit sky—an unusual view that might be OoD—it crashed into the truck, killing the driver [3].

Previous research has successfully used 3D graphics as a diagnostic tool for computer vision systems [15, 89, 122, 93, 96]. To understand natural Type II classification errors in DNNs, we searched for misclassified 6D poses (i.e., 3D translations and 3D rotations) of 3D objects. Our results reveal that state-of-the-art image classifiers and object detectors trained on large-scale image datasets [101, 75] misclassify most poses for many familiar training-set objects. For example, DNNs predict the front view of a school bus—an object in the

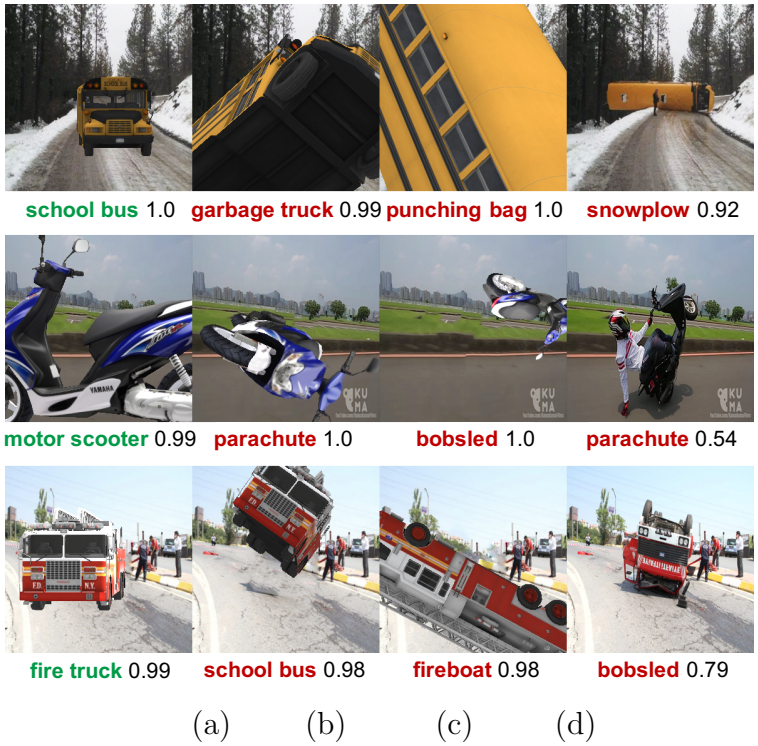



Figure 2.1: The Google Inception-v3 classifier [117] correctly labels the canonical poses of objects (a), but fails to recognize out-of-distribution images of objects in unusual poses (b–d), including real photographs retrieved from the Internet (d). The left $3 \times 3$ images (a–c) are found by our framework and rendered via a 3D renderer. Below each image are its top-1 predicted label and confidence score.

ImageNet dataset [101]—extremely well (Fig. 4.1a) but fail to recognize the same object when it is too close or flipped over, i.e., in poses that are OoD yet exist in the real world (Fig. 4.1d). However, a self-driving car needs to correctly estimate at least some attributes of an incoming, unknown object (instead of simply rejecting it [53, 108]) to handle the situation gracefully and minimize damage. Because road environments are highly variable [3, 2], addressing this type of OoD error is a non-trivial challenge.

In this chapter, we propose a framework for finding OoD errors in computer vision models in which iterative optimization in the parameter space of a 3D renderer is used to estimate changes (e.g., in object geometry and appearance, lighting, background, or camera settings) that cause a target DNN to misbehave (Fig. 2.2). With our framework, we generated unrestricted 6D poses of 3D objects and studied how DNNs respond to 3D translations

and 3D rotations of objects. For our study, we built a dataset of 3D objects corresponding to 30 ImageNet classes relevant to the self-driving car application. The code for our framework is available at `https://github.com/airalcorn2/strike-with-a-pose`. In addition, we built a simple GUI tool that allows users to generate their own adversarial renders of an object. Our main findings are:

- ImageNet classifiers only correctly label 3.09% of the entire 6D pose space of a 3D object, and misclassify many generated adversarial examples (AXs) that are human-recognizable (Fig. 4.1b–c). A misclassification can be found via a change as small as 10.31°, 8.02°, and 9.17° to the yaw, pitch, and roll, respectively.
- 99.9% and 99.4% of AXs generated against Inception-v3 transfer to the AlexNet and ResNet-50 image classifiers, respectively, and 75.5% transfer to the YOLOv3 object detector.
- Training on adversarial poses generated by the 30 objects (in addition to the original ImageNet data) did not help DNNs generalize well to held-out objects in the same class.

In sum, our work shows that state-of-the-art DNNs perform *image classification* well but are still far from true *object recognition*. While it might be possible to improve DNN robustness through adversarial training with many more 3D objects, we hypothesize that future ML models capable of visual reasoning may instead benefit from better incorporation of 3D information.

## 2.2 Framework

### 2.2.1 Problem formulation

Let $f$ be an image classifier that maps an image $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ onto a softmax probability distribution over 1,000 output classes [117]. Let $R$ be a 3D renderer that takes as input a

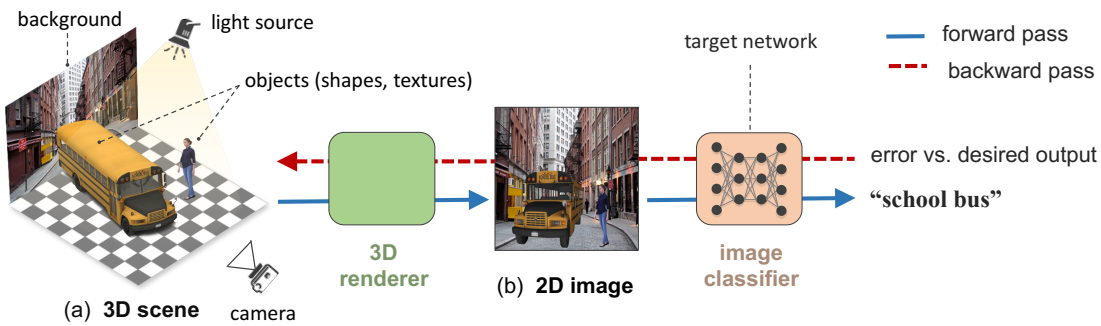



Figure 2.2: To test a target DNN, we build a 3D scene (a) that consists of 3D objects (here, a school bus and a pedestrian), lighting, a background scene, and camera parameters. Our 3D renderer renders the scene into a 2D image, which the image classifier labels `school bus`. We can estimate the pose changes of the school bus that cause the classifier to misclassify by (1) approximating gradients via finite differences; or (2) backpropagating (red dashed line) through a differentiable renderer.

set of parameters $\phi$ and outputs a render, i.e., a 2D image $R(\phi) \in \mathbb{R}^{H \times W \times C}$ (see Fig. 2.2). Typically, $\phi$ is factored into mesh vertices $V$, texture images $T$, a background image $B$, camera parameters $C$, and lighting parameters $L$, i.e., $\phi = \{V, T, B, C, L\}$ [59]. To change the 6D pose of a given 3D object, we apply a 3D rotation and 3D translation, parameterized by $\mathbf{w} \in \mathbb{R}^6$, to the original vertices $V$ yielding a new set of vertices $V^*$.

Here, we wish to estimate only the pose transformation parameters $\mathbf{w}$ (while keeping all parameters in $\phi$ fixed) such that the rendered image $R(\mathbf{w}; \phi)$ causes the classifier $f$ to assign the highest probability (among all outputs) to an incorrect target output at index $t$. Formally, we attempt to solve the below optimization problem:

$$\mathbf{w}^* = \arg\max_{\mathbf{w}} (f_t(R(\mathbf{w}; \phi))) \tag{2.1}$$

In practice, we minimize the cross-entropy loss $\mathcal{L}$ for the target class. Eq. (2.1) may be solved efficiently via backpropagation if both $f$ and $R$ are differentiable, i.e., we are able to compute $\partial \mathcal{L} / \partial \mathbf{w}$. However, standard 3D renderers, e.g., OpenGL [132], typically include many non-differentiable operations and cannot be inverted [81]. Therefore, we attempted two

approaches: (1) harnessing a recently proposed differentiable renderer and performing gradient descent using its analytical gradients; and (2) harnessing a non-differentiable renderer and approximating the gradient via finite differences.

We will next describe the target classifier (Sec. 2.2.2), the renderers (Sec. 2.2.3), and our dataset of 3D objects (Sec. 2.2.4) before discussing the optimization methods (Sec. 2.3).

### 2.2.2 Classification networks

We chose the well-known, pre-trained Google Inception-v3 [119] DNN from the PyTorch model zoo [94] as the main image classifier for our study (the default DNN if not otherwise stated). The DNN has a 77.45% top-1 accuracy on the ImageNet ILSVRC 2012 dataset [101] of 1.2 million images corresponding to 1,000 categories.

### 2.2.3 3D renderers

**Non-differentiable renderer.** We chose ModernGL [1] as our non-differentiable renderer. ModernGL is a simple Python interface for the widely used OpenGL graphics engine. ModernGL supports fast, GPU-accelerated rendering.

**Differentiable renderer.** To enable backpropagation through the non-differentiable rasterization process, Kato et al. [59] replaced the discrete pixel color sampling step with a linear interpolation sampling scheme that admits non-zero gradients. While the approximation enables gradients to flow from the output image back to the renderer parameters $\phi$, the render quality is lower than that of our non-differentiable renderer. Hereafter, we refer to the two renderers as NR and DR.

### 2.2.4 3D object dataset

**Construction.** Our main dataset consists of 30 unique 3D object models (purchased from many 3D model marketplaces) corresponding to 30 ImageNet classes relevant to a traffic

environment. The 30 classes include 20 vehicles (e.g., `school bus` and `cab`) and 10 street-related items (e.g., `traffic light`).

Each 3D object is represented as a mesh, i.e., a list of triangular faces, each defined by three vertices [81]. The 30 meshes have on average 9,908 triangles. To maximize the realism of the rendered images, we used only 3D models that have high-quality 2D image textures. We did not choose 3D models from public datasets, e.g., ObjectNet3D [133], because most of them do not have high-quality image textures. That is, the renders of such models may be correctly classified by DNNs but still have poor realism.

**Evaluation.** We recognize that a reality gap will often exist between a render and a real photo. Therefore, we rigorously evaluated our renders to make sure the reality gap was acceptable for our study. From $\sim$100 initially-purchased 3D models, we selected the 30 highest-quality models using the evaluation method below.

First, we quantitatively evaluated DNN predictions on the renders. For each object, we sampled 36 unique views (common in ImageNet) evenly divided into three sets. For each set, we set the object at the origin, the up direction to $(0, 1, 0)$, and the camera position to $(0, 0, -z)$ where $z = \{4, 6, 8\}$. We sampled 12 views per set by starting the object at a $10^\circ$ yaw and generating a render at every $30^\circ$ yaw-rotation. Across all objects and all renders, the Inception-v3 top-1 accuracy was 83.23% (compared to 77.45% on ImageNet images [117]) with a mean top-1 confidence score of 0.78.

Second, we qualitatively evaluated the renders by comparing them to real photos. We produced 116 (real photo, render) pairs via three steps: (1) we retrieved real photos of an object (e.g., a car) from the Internet; (2) we replaced the object with matching background content in Adobe Photoshop; and (3) we manually rendered the 3D object on the background such that its pose closely matched that in the reference photo. While discrepancies can be spotted in our side-by-side comparisons, we found that most of the renders passed our human visual Turing test if presented alone.

### 2.2.5 Background images

Previous studies have shown that image classifiers may be able to correctly label an image when foreground objects are removed (i.e., based on only the background content) [146]. Because the purpose of our study was to understand how DNNs recognize an object itself, a non-empty background would have hindered our interpretation of the results. Therefore, we rendered all images against a plain background with RGB values of $(0.485, 0.456, 0.406)$, i.e., the mean pixel of ImageNet images. Note that the presence of a non-empty background should not alter our main qualitative findings in this chapter—adversarial poses can be easily found against real background photos (Fig. 4.1).

## 2.3 Methods

We will describe the common pose transformations (Sec. 2.3.1) used in the main experiments. We were able to experiment with non-gradient methods because: (1) the pose transformation space $\mathbb{R}^6$ that we optimize in is fairly low-dimensional; and (2) although the NR is non-differentiable, its rendering speed is several orders of magnitude faster than that of DR. In addition, our preliminary results showed that the objective function considered in Eq. (2.1) is highly non-convex (see Fig. 2.4), therefore, it is interesting to compare (1) random search vs. (2) gradient descent using finite-difference (FD) approximated gradients vs. (3) gradient descent using the DR gradients.

### 2.3.1 Pose transformations

We used standard computer graphics transformation matrices to change the pose of 3D objects [81]. Specifically, to rotate an object with geometry defined by a set of vertices $V = \{v_i\}$, we applied the linear transformations in Eq. (2.2) to each vertex $v_i \in \mathbb{R}^3$:

$$v_i^R = R_y R_p R_r v_i \tag{2.2}$$

where $R_y$, $R_p$, and $R_r$ are the $3 \times 3$ rotation matrices for yaw, pitch, and roll, respectively. We then translated the rotated object by adding a vector $T = \begin{bmatrix} x_\delta & y_\delta & z_\delta \end{bmatrix}^\top$ to each vertex:

$$v_i^{R,T} = T + v_i^R \tag{2.3}$$

In all experiments, the center $c \in \mathbb{R}^3$ of the object was constrained to be inside a sub-volume of the camera viewing frustum. That is, the $x$-, $y$-, and $z$-coordinates of $c$ were within $[-s, s]$, $[-s, s]$, and $[-28, 0]$, respectively, with $s$ being the maximum value that would keep $c$ within the camera frame. Specifically, $s$ is defined as:

$$s = d \cdot \tan(\theta_v) \tag{2.4}$$

where $\theta_v$ is one half the camera's angle of view (i.e., 8.213° in our experiments) and $d$ is the absolute value of the difference between the camera's $z$-coordinate and $z_\delta$.

### 2.3.2 Random search

In reinforcement learning problems, random search (RS) can be surprisingly effective compared to more sophisticated methods [114]. For our RS procedure, instead of iteratively following some approximated gradient to solve the optimization problem in Eq. (2.1), we simply randomly selected a new pose in each iteration. The rotation angles for the matrices in Eq. (2.2) were uniformly sampled from $(0, 2\pi)$. $x_\delta$, $y_\delta$, and $z_\delta$ were also uniformly sampled from the ranges defined in Sec. 2.3.1.

### 2.3.3 $z_\delta$-constrained random search

Our preliminary RS results suggest the value of $z_\delta$ (which is a proxy for the object's size in the rendered image) has a large influence on a DNN's predictions. Based on this observation, we used a $z_\delta$-constrained random search (ZRS) procedure both as an initializer for our gradient-based methods and as a naive performance baseline (for comparisons in Sec.

2.4.4). The ZRS procedure consisted of generating 10 random samples of $(x_\delta, y_\delta, \theta_y, \theta_p, \theta_r)$ at each of 30 evenly spaced $z_\delta$ from $-28$ to 0.

When using ZRS for initialization, the parameter set with the maximum target probability was selected as the starting point. When using the procedure as an attack method, we first gathered the maximum target probabilities for each $z_\delta$, and then selected the best two $z_\delta$ to serve as the new range for RS.

### 2.3.4 Gradient descent with finite-difference

We calculated the first-order derivatives via finite central differences and performed vanilla gradient descent to iteratively minimize the cross-entropy loss $\mathcal{L}$ for a target class. That is, for each parameter $\mathbf{w}_i$, the partial derivative is approximated by:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_i} = \frac{\mathcal{L}(\mathbf{w}_i + \frac{h}{2}) - \mathcal{L}(\mathbf{w}_i - \frac{h}{2})}{h} \tag{2.5}$$

Although we used an $h$ of 0.001 for all parameters, a different step size can be used per parameter. Because radians have a circular topology (i.e., a rotation of 0 radians is the same as a rotation of $2\pi$ radians, $4\pi$ radians, etc.), we parameterized each rotation angle $\theta_i$ as $(\cos(\theta_i), \sin(\theta_i))$—a technique commonly used for pose estimation [88] and inverse kinematics [25]—which maps the Cartesian plane to angles via the $atan2$ function. Therefore, we optimized in a space of $3 + 2 \times 3 = 9$ parameters.

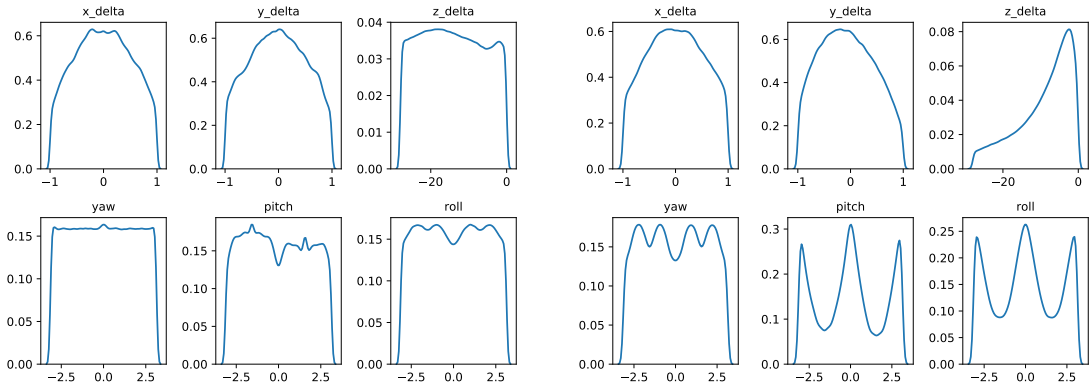
The approximate gradient $\nabla\mathcal{L}$ obtained from Eq. (2.5) served as the gradient in our gradient descent. We used the vanilla gradient descent update rule:

$$\mathbf{w} := \mathbf{w} - \gamma \nabla \mathcal{L}(\mathbf{w}) \tag{2.6}$$

with a learning rate $\gamma$ of 0.001 for all parameters and optimized for 100 steps (no other stopping criteria).

## 2.4 Experiments and results

### 2.4.1 Neural networks are easily confused by object rotations and translations




(a) Incorrect classifications (b) Correct classifications

Figure 2.3: The distributions of individual pose parameters for (a) high-confidence ($p \geq 0.7$) incorrect classifications and (b) correct classifications obtained from the random sampling procedure described in Sec. 2.3.2. $x_\delta$ and $y_\delta$ have been normalized w.r.t. their corresponding $s$ from Eq. (2.4).

**Experiment.** To test DNN robustness to object rotations and translations, we used RS to generate samples for every 3D object in our dataset. In addition, to explore the impact of lighting on DNN performance, we considered three different lighting settings: bright, medium, and dark. In all three settings, both the directional light and the ambient light were white in color, i.e., had RGB values of $(1.0, 1.0, 1.0)$, and the directional light was oriented at $(0, -1, 0)$ (i.e., pointing straight down). The directional light intensities and ambient light intensities were $(1.2, 1.6)$, $(0.4, 1.0)$, and $(0.2, 0.5)$ for the bright, medium, and darksettings, respectively. All other experiments used the mediumlighting setting.

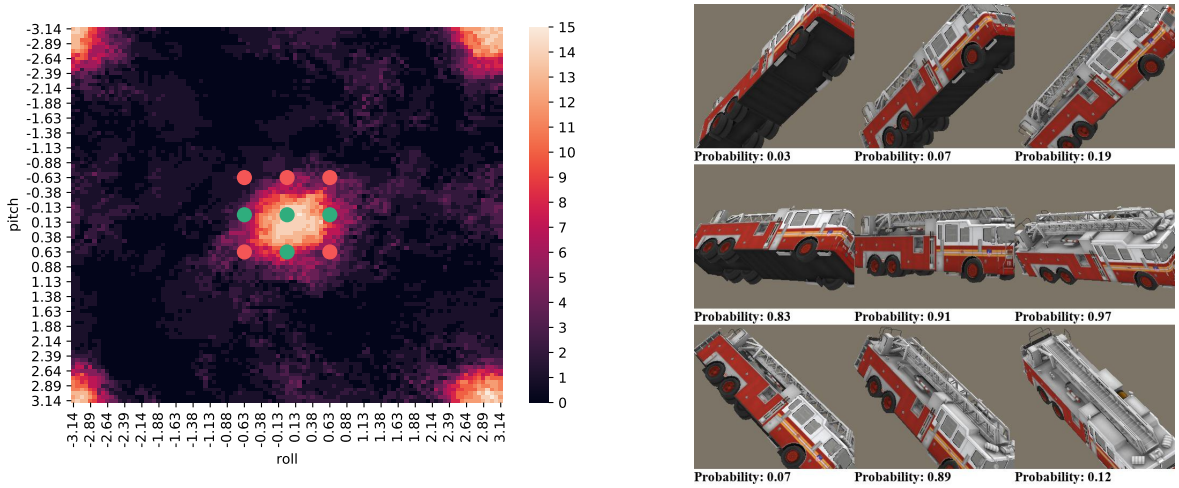
**Misclassifications uniformly cover the pose space.** For each object, we calculated the DNN accuracy (i.e., percent of correctly classified samples) across all three lighting settings. The DNN was wrong for the vast majority of samples, i.e., the median percent of correct classifications for all 30 objects was only 3.09%. We verified the discovered adversarial poses transfer to the real world by using the 3D objects to reproduce natural, misclassified poses

found on the Internet. High-confidence misclassifications ($p \geq 0.7$) are largely uniformly distributed across every pose parameter (Fig. 2.3a), i.e., AXs can be found throughout the parameter landscape for examples). In contrast, correctly classified examples are highly multimodal w.r.t. the rotation axis angles and heavily biased towards $z_\delta$ values that are closer to the camera (Fig. 2.3b). Intriguingly, for ball-like objects (not included in our main traffic dataset), the DNN was far more accurate across the pose space.

**An object can be misclassified as many different labels.** Previous research has shown that it is relatively easy to produce AXs corresponding to many different classes when optimizing input images [120] or 3D object textures [8], which are very high-dimensional. When finding adversarial poses, one might expect—because all renderer parameters, including the original object geometry and textures, are held constant—the success rate to depend largely on the similarities between a given 3D object and examples of the target in ImageNet. Interestingly, across our 30 objects, RS discovered 990/1000 different ImageNet classes (132 of which were shared between all objects). When only considering high-confidence ($p \geq 0.7$) misclassifications, our 30 objects were still misclassified into 797 different classes with a median number of 240 incorrect labels found per object. Across all adversarial poses and objects, DNNs tend to be more confident when correct than when wrong (the median of median probabilities were 0.41 vs. 0.21, respectively).

### 2.4.2 Common object classifications are shared across different lighting settings

Here, we analyze how our results generalize across different lighting conditions. From the data produced in Sec. 2.4.1, for each object, we calculated the DNN accuracy under each lighting setting. Then, for each object, we took the absolute difference of the accuracies for all three lighting combinations (i.e., bright vs. medium, bright vs. dark, and medium vs. dark) and recorded the maximum of those values. The median "maximum absolute difference" of accuracies for all objects was 2.29% (compared to the median accuracy of 3.09% across all




(a) (b)

Figure 2.4: Inception-v3's ability to correctly classify images is highly localized in the rotation and translation parameter space. (a) The classification landscape for 15 vehicle objects when altering $\theta_r$ and $\theta_p$ and holding $(x_\delta, y_\delta, z_\delta, \theta_y)$ at $(0, 0, -3, \frac{\pi}{4})$. Lighter regions correspond to poses with a greater number of correctly classified vehicle objects. Green and red circles indicate correct and incorrect classifications, respectively, corresponding to the fire truck object poses found in (b).

lighting settings). That is, DNN accuracy is consistently low across all lighting conditions. Lighting changes would not alter the fact that DNNs are vulnerable to adversarial poses.

We also recorded the 50 most frequent classes for each object under the different lighting settings ($S_b$, $S_m$, and $S_d$). Then, for each object, we computed the intersection over union score $o_S$ for these sets:

$$o_S = 100 \cdot \frac{|S_b \cap S_m \cap S_d|}{|S_b \cup S_m \cup S_d|} \tag{2.7}$$

The median $o_S$ for all objects was 47.10%. That is, for 15 out of 30 objects, 47.10% of the 50 most frequent classes were shared across lighting settings. While lighting does have an impact on DNN misclassifications (as expected), the large number of shared labels across lighting settings suggests ImageNet classes are strongly associated with certain adversarial poses regardless of lighting.

### 2.4.3 Correct classifications are highly localized in the rotation and translation landscape

To gain some intuition for how Inception-v3 responds to rotations and translations of an object, we plotted the probability and classification landscapes for paired parameters (e.g., Fig. 2.4; pitch vs. roll) while holding the other parameters constant. We qualitatively observed that the DNN's ability to recognize an object (e.g., a fire truck) in an image varies radically as the object is rotated in the world (Fig. 2.4). Further, adversarial poses often generalize across similar objects (e.g., 83% of the sampled poses were misclassified for *all* 15 four-wheeled vehicle objects).

**Experiment.** To quantitatively evaluate the DNN's sensitivity to rotations and translations, we tested how it responded to single parameter disturbances. For each object, we randomly selected 100 distinct starting poses that the DNN had correctly classified in our random sampling runs. Then, for each parameter (e.g., yaw rotation angle), we randomly sampled 100 new values[1] while holding the others constant. For each sample, we recorded whether or not the object remained correctly classified, and then computed the failure (i.e., misclassification) rate for a given (object, parameter) pair.

Additionally, for each parameter, we calculated the median of the median failure rates. That is, for each parameter, we first calculated the median failure rate for all objects, and then calculated the median of those medians for each parameter. Further, for each (object, starting pose, parameter) triple, we recorded the magnitude of the smallest parameter change that resulted in a misclassification. Then, for each (object, parameter) pair, we recorded the median of these minimum values. Finally, we again calculated the median of these medians across objects (Table 2.1).

**Results.** As can be seen in Table 2.1, the DNN is highly sensitive to all single parameter disturbances, but it is especially sensitive to disturbances along the depth ($z_\delta$), pitch ($\theta_p$), and roll ($\theta_r$). To aid in the interpretation of these results, we converted the raw disturbance

[1] using the random sampling procedure described in Sec. 2.3.2

values in Table 2.1 to image units. For $x_\delta$ and $y_\delta$, the interpretable units are the number of pixels the object shifted in the $x$ or $y$ directions *of the image* (however, note that 3D translations are *not* equivalent to 2D translations due to the perspective projection).

We found that a change in rotation as small as 8.02° can cause an object to be misclassified (Table 2.1). Along the spatial dimensions, a translation resulting in the object moving as few as 2 px horizontally or 4.5 px vertically also caused the DNN to misclassify.[2] Lastly, along the $z$-axis, a change in "size" (i.e., the area of the object's bounding box) of only 5.4% can cause an object to be misclassified.

| Parameter | Fail Rate (%) | Min. $\Delta$ | Int. $\Delta$ |
|---|---|---|---|
| $x_\delta$ | 42 | 0.09 | 2.0 px |
| $y_\delta$ | 49 | 0.10 | 4.5 px |
| $z_\delta$ | 81 | 0.77 | 5.4% |
| $\theta_y$ | 69 | 0.18 | 10.31° |
| $\theta_p$ | 83 | 0.14 | 8.02° |
| $\theta_r$ | 81 | 0.16 | 9.17° |

Table 2.1: The median of the median failure rates and the median of the median minimum disturbances (Min. $\Delta$) for the single parameter sensitivity tests described in Section 2.4.3. Int. $\Delta$ converts the values in Min. $\Delta$ to more interpretable units. For $x_\delta$ and $y_\delta$, the interpretable units are pixels. For $z_\delta$, the interpretable unit is the percent change in the area of the bounding box containing the object.

### 2.4.4 Optimization methods can effectively generate targeted adversarial poses

Given a challenging, highly non-convex objective landscape (Fig. 2.4), we wish to evaluate the effectiveness of two different types of approximate gradients at targeted attacks, i.e., finding adversarial examples misclassified as a target class [120]. Here, we compare (1) random search; (2) gradient descent with finite-difference gradients (FD-G); and (3) gradient descent with analytical, approximate gradients provided by a differentiable renderer (DR-G) [59].

[2]Note that the sensitivity of classifiers and object detectors to *2D* translations has been observed in concurrent work [98, 34, 143, 9].

**Experiment.** Because our adversarial pose attacks are inherently constrained by the fixed geometry and appearances of a given 3D object (see Sec. 2.4.1), we defined the targets to be the 50 most frequent incorrect classes found by our RS procedure for each object. For each (object, target) pair, we ran 50 optimization trials using ZRS, FD-G, and DR-G. All treatments were initialized with a pose found by the ZRS procedure and then allowed to optimize for 100 iterations.

**Results.** For each of the 50 optimization trials, we recorded both whether or not the target was hit and the maximum target probability obtained during the run. For each (object, target) pair, we calculated the percent of target hits and the median maximum confidence score of the target labels (see Table 2.2). As shown in Table 2.2, FD-G is substantially more effective than ZRS at generating targeted adversarial poses, having both higher median hit rates and confidence scores. In addition, we found the approximate gradients from DR to be surprisingly noisy, and DR-G largely underperformed even non-gradient methods (ZRS).

| | | Hit Rate (%) | Target Prob. |
|---|---|---|---|
| ZRS | random search | 78 | 0.29 |
| FD-G | gradient-based | **92** | **0.41** |
| DR-G$^\dagger$ | gradient-based | 32 | 0.22 |

Table 2.2: The median percent of target hits and the median of the median target probabilities for random search (ZRS), gradient descent with finite difference gradients (FD-G), and DR gradients (DR-G). All attacks are targeted and initialized with $z_\delta$-constrained random search. $^\dagger$DR-G is not directly comparable to FD-G and ZRS.

### 2.4.5 Adversarial poses transfer to different image classifiers and object detectors

The most important property of previously documented AXs is that they transfer across ML models, enabling black-box attacks [138]. Here, we investigate the transferability of our adversarial poses to (a) two different image classifiers, AlexNet [67] and ResNet-50 [52],

trained on the same ImageNet dataset; and (b) an object detector YOLOv3 [97] trained on the MS COCO dataset [75].

For each object, we randomly selected 1,350 AXs that were misclassified by Inception-v3 with high confidence ($p \geq 0.9$) from our untargeted RS experiments in Sec. 2.4.1. We exposed the AXs to AlexNet and ResNet-50 and calculated their misclassification rates. We found that almost all AXs transfer with median misclassification rates of 99.9% and 99.4% for AlexNet and ResNet-50, respectively. In addition, 10.1% of AlexNet misclassifications and 27.7% of ResNet-50 misclassifications were identical to the Inception-v3 predicted labels.

There are two orthogonal hypotheses for this result. First, the ImageNet training-set images themselves may contain a strong bias towards common poses, omitting uncommon poses. Second, the models themselves may not be robust to even slight disturbances of the known, in-distribution poses.

**Object detectors.** Previous research has shown that object detectors can be more robust to adversarial attacks than image classifiers [79]. Here, we investigate how well our AXs transfer to a state-of-the-art object detector—YOLOv3. YOLOv3 was trained on MS COCO, a dataset of bounding boxes corresponding to 80 different object classes. We only considered the 13 objects that belong to classes present in both the ImageNet and MS COCO datasets. We found that 75.5% of adversarial poses generated for Inception-v3 are also misclassified by YOLOv3. These results suggest the adversarial pose problem transfers across datasets, models, and tasks.

### 2.4.6 Adversarial training

One of the most effective methods for defending against OoD examples has been adversarial training [42], i.e. augmenting the training set with AXs—also a common approach in anomaly detection [19]. We tested whether adversarial training can improve DNN robustness to new poses generated for (1) our 30 training-set 3D objects; and (2) seven held-out 3D objects. Following adversarial training, the accuracy of the DNN substantially increased for

*known* objects (Table 2.3; 99.67% vs. 6.7%). However, the model (AT) still misclassified the adversarial poses of held-out objects at an 89.2% error rate.

| | PT | AT |
|---|---|---|
| Error (T) | 99.67 | 6.7 |
| Error (H) | 99.81 | 89.2 |
| High-confidence Error (T) | 87.8 | 1.9 |
| High-confidence Error (H) | 48.2 | 33.3 |

Table 2.3: The median percent of misclassifications (Error) and high-confidence (i.e., $p > 0.7$) misclassifications by the pre-trained AlexNet (PT) and our AlexNet trained with adversarial examples (AT) on random poses of training-set objects (T) and held-out objects (H).

## 2.5 Related work

**Out-of-distribution detection.** OoD classes, i.e., classes not found in the training set, present a significant challenge for computer vision technologies in real-world settings [108]. Here, we study an orthogonal problem—correctly classifying OoD poses of objects from *known* classes. While rejecting to classify is a common approach for handling OoD examples [53, 108], the OoD poses in our work come from known classes and thus *should be* assigned correct labels.

**2D adversarial examples.** Numerous techniques for crafting AXs that fool image classifiers have been discovered [138]. However, previous work has typically optimized in the 2D input space [138], e.g., by synthesizing an entire image [86], a small patch [58, 35], a few pixels [18], or only a single pixel [113]. But pixel-wise changes are uncorrelated [85], so pixel-based attacks may not transfer well to the real world [78, 80] because there is an infinitesimal chance that such specifically crafted, uncorrelated pixels will be encountered in the vast physical space of camera, lighting, traffic, and weather configurations. [134] generated spatially transformed adversarial examples that are perceptually realistic and more difficult to defend against, but the technique still directly operates on pixels.

**3D adversarial examples.** Athalye et al. [8] used a 3D renderer to synthesize textures for a 3D object such that, under a wide range of camera views, the object was still rendered into an effective AX. We also used 3D renderers, but instead of optimizing textures, we optimized the poses of known objects to cause DNNs to misclassify (i.e., we kept the textures, lighting, camera settings, and background image constant).

**Concurrent work.** We describe below two concurrent attempts that are closely related to ours. First, Liu et al. [76] proposed a differentiable 3D renderer and used it to perturb both an object's geometry and the scene's lighting to cause a DNN to misbehave. However, their geometry perturbations were constrained to be infinitesimal so that the visibility of the vertices would not change. Therefore, their result of minutely perturbing the geometry is effectively similar to that of perturbing textures [8]. In contrast, we performed 3D rotations and 3D translations to move an object inside a 3D space (i.e., the viewing frustum of the camera).

Second, Engstrom et al. [34] showed how simple 2D image rotations and translations can cause DNNs to misclassify. However, these 2D transformations still do not reveal the type of adversarial poses discovered by rotating 3D objects (e.g., a flipped-over school bus; Fig. 4.1d).

To the best of our knowledge, our work is the first attempt to harness 3D objects to study the OoD poses of well-known training-set objects that cause state-of-the-art ImageNet classifiers and MS COCO detectors to misclassify.

## 2.6 Discussion and conclusion

In this chapter, we revealed how DNNs' understanding of objects like "school bus" and "fire truck" is quite naive—they can correctly label only a small subset of the entire pose space for 3D objects. Note that we can also find real-world OoD poses by simply taking photos of real objects. We believe classifying an arbitrary pose into one of the object classes is an ill-posed task, and that the adversarial pose problem might be alleviated via multiple

orthogonal approaches. The first is addressing biased data [124]. Because ImageNet and MS COCO datasets are constructed from photographs taken by people, the datasets reflect the aesthetic tendencies of their captors. Such biases can be somewhat alleviated through data augmentation, specifically, by harnessing images generated from 3D renderers [109, 7]. From the modeling view, we believe DNNs would benefit from the incorporation of 3D information, e.g., [7].

Finally, our work introduced a new promising method (Fig. 2.2) for testing computer vision DNNs by harnessing 3D renderers and 3D models. While we only optimize a single object here, the framework could be extended to jointly optimize lighting, background image, and multiple objects, all in one "adversarial world". Not only does our framework enable us to enumerate test cases for DNNs, but it also serves as an interpretability tool for extracting useful insights about these black-box models' inner functions.

# Chapter 3

# Using Covariance Matrix Adaptation Evolution Strategy to Discover Adversarial Poses for Artificial Neural Networks

**This chapter features contributions from Chinedu Eleh and Qi Li.**

## 3.1 Introduction

A flurry of studies over the past several years have revealed the myriad of ways artificial neural networks are vulnerable to "adversarial examples" [121], i.e., examples that appear benign to a human but cause a neural network to fail. Typically, adversarial examples are generated by using gradient information from a target neural network to update the pixels of an input image. Essentially, this procedure adds human-imperceptible "noise" to an image in a way that causes a neural network to make a specific, incorrect prediction. In Chapter 2, we showed that neural networks are also easily confused by "natural adversarial" examples, i.e., unmanipulated images of the real world that are easily recognized by a human but cause a neural network to fail (e.g., a flipped over school bus; see Fig. 3.1).

Because the rendering process is non-differentiable, black box optimization techniques are necessary for discovering adversarial poses of 3D objects. In Chapter 2, we used finite differences (FD) to approximate the partial derivatives of a neural network's loss with respect to 3D pose parameters, and showed that gradient descent using these approximated partial derivatives could more effectively generate targeted adversarial poses than an informed random search approach. However, the pose loss landscape appears to be highly nonconvex and may possess many plateaus and saddle points (as is often the case with neural networks

(a) (b)

Figure 3.1: The Google Inception-v3 neural network [118] correctly recognizes a 3D object of a school bus in a "canonical pose" (a) but is confused by the same object in an adversarial pose (b).

[27]), so derivative-free techniques may be better suited for this task. Evolutionary algorithms are a class of derivative-free techniques inspired by evolutionary biology that have been successfully used in a number of applications. Broadly, evolutionary algorithms consist of "populations" of candidate solutions, a "mutation" mechanism (i.e., a mechanism for randomly changing parameter values), a notion of "fitness" (typically defined in relation to a cost function), and a "natural selection" process (i.e., a way for "fitter" candidate solutions to reproduce and "weaker" candidate solutions to be removed). In this report, we compare the effectiveness of covariance matrix adaptation evolution strategy (CMA-ES) [51, 48], an evolutionary algorithm that has shown promise in reinforcement learning settings [49, 104], to FD for adversarial pose discovery.

### 3.2 Covariance matrix adaptation evolution strategy

In CMA-ES, candidate solutions are sampled from a multivariate normal distribution where the mean vector $m$, covariance matrix $C$, and a scalar $\sigma$ (which scales $C$ and is referred

to as the "step size" or "path length control") are updated through an evolutionary process (see pseudocode in Fig. 3.2). CMA-ES also maintains two vectors, $p_c$, referred to as the "anisotropic evolution path", and $p_\sigma$, referred to as the "isotropic evolution path", which contain information on the history of updates. In the following sections, we provide detailed descriptions of how $m$, $C$, and $\sigma$ are updated and further explain the roles of $p_c$ and $p_\sigma$.

```
set λ  // number of samples per iteration, at least two, generally > 4
initialize m, σ, C = I, p_σ = 0, p_c = 0  // initialize state variables
while not terminate  // iterate
   for i in {1...λ}  // sample λ new solutions and evaluate them
      x_i = sample_multivariate_normal(mean=m, covariance_matrix=σ^2 C)
      f_i = fitness(x_i)
   x_{1...λ} ← x_{s(1)...s(λ)} with s(i) = argsort(f_{1...λ}, i)  // sort solutions
   m' = m  // we need later m − m' and x_i − m'
   m ← update_m(x_1, ..., x_λ)  // move mean to better solutions
   p_σ ← update_ps(p_σ, σ^{-1} C^{-1/2} (m − m'))  // update isotropic evolution path
   p_c ← update_pc(p_c, σ^{-1} (m − m'), ||p_σ||)  // update anisotropic evolution path
   C ← update_C(C, p_c, (x_1 − m')/σ, ..., (x_λ − m')/σ)  // update covariance matrix
   σ ← update_sigma(σ, ||p_σ||)  // update step-size using isotropic path length
return m or x_1
```

Figure 3.2: Pseudocode for the CMA-ES algorithm (image source: `https://en.wikipedia.org/wiki/CMA-ES`).

### 3.2.1 Updating $m$

Like all evolutionary algorithms, the first step of CMA-ES is to generate a population of individuals. To do so, CMA-ES samples $\lambda$ candidate solutions (also referred to as "children") from the multivariate normal distribution specified by the values of $m$, $\sigma$, and $C$ for the current generation $g$, i.e.:

$$x \sim \mathcal{N}(m^{(g)}, (\sigma^{(g)})^2 C^{(g)})$$

Each $x$[1] is then assigned a fitness score[2]. To update the population mean vector $m$, CMA-ES simply takes a weighted average of the top $\mu$ individuals (also referred to as "parents") as determined by the fitness function:

$$m^{(g+1)} = \sum_{i=1}^{\mu} w_i \, x_{i:\lambda}^{(g)} \tag{3.1}$$

which is equivalent to:

$$m^{(g+1)} = m^{(g)} + \sum_{i=1}^{\mu} w_i y_{i:\lambda}^{(g)} \tag{3.2}$$

where $w_1 \geq w_2 \geq \cdots \geq w_\mu > 0$ sum to one, $y_{i:\lambda}^{(g)} = x_{i:\lambda}^{(g)} - m^{(g)}$ (i.e., a step direction), and $x_{i:\lambda}^{(g)}$ is the $i$th ranked candidate solution.

### 3.2.2 Updating $C$

Suppose at generation $g = 0, 1, 2, \cdots$ we obtain the vectors $x_1^{(g)}$, $x_2^{(g)}, \cdots$, $x_\lambda^{(g)}$. The empirical covariance matrix $C^{(g)}$ can be defined in terms of rank-one matrices:

$$C^{(g)} = \frac{1}{\lambda - 1} \sum_{i=1}^{\lambda} \left( x_i^{(g)} - \frac{1}{\lambda} \sum_{j=1}^{\lambda} x_j^{(g)} \right) \left( x_i^{(g)} - \frac{1}{\lambda} \sum_{j=1}^{\lambda} x_j^{(g)} \right)^T \tag{3.3}$$

where $\frac{1}{\lambda} \sum_{j=1}^{\lambda} x_j^{(g)} = m^{(g)}$ is the mean of the random vectors. By using only the rank-one matrices from the fittest individuals, e.g.:

$$C_\mu^{(g+1)} = \sum_{i=1}^{\mu} w_i \left( x_{i:\lambda}^{(g+1)} - m^{(g)} \right) \left( x_{i:\lambda}^{(g+1)} - m^{(g)} \right)^T \tag{3.4}$$

we may be able to produce a covariance matrix that generates better candidate solutions in the subsequent generation.

[1] In our case, $x$ contains the pose parameters.
[2] In our case, the fitness function is derived from the neural network loss.

Notably, substituting the point $m^{(g)}$ in (3.4) with a point $m^{(g+1)}$ from the convex hull of $m^{(g)}$ and the mean of the $\mu$ "fittest" individuals produces the Estimation of Multivariate Normal Algorithm (EMNA) covariance matrix:

$$C_{\text{EMNA}}^{(g+1)} = \sum_{i=1}^{\mu} w_i \left(x_{i:\lambda}^{(g+1)} - m^{(g+1)}\right)\left(x_{i:\lambda}^{(g+1)} - m^{(g+1)}\right)^T \tag{3.5}$$

For the simple case where $w_i = \frac{1}{\mu}$, (3.5) reduces to

$$C_{\text{EMNA}}^{(g+1)} = \frac{1}{\mu}\sum_{i=1}^{\mu} \left(x_{i:\lambda}^{(g+1)} - m^{(g+1)}\right)\left(x_{i:\lambda}^{(g+1)} - m^{(g+1)}\right)^T \tag{3.6}$$

Further, multiplying (3.6) by $\frac{\mu}{\mu-1}$ gives the unbiased estimator of the covariance matrix for the parent vectors. Algorithms using $\frac{\mu}{\mu-1}C_{\text{EMNA}}^{(g+1)}$ can converge very fast; however, in cases where the initial sampling does not bracket the optimum (which is likely for a small $\mu$), convergence is often premature [50], which is why CMA-ES uses $m^{(g)}$ instead (see Fig. 3.3).

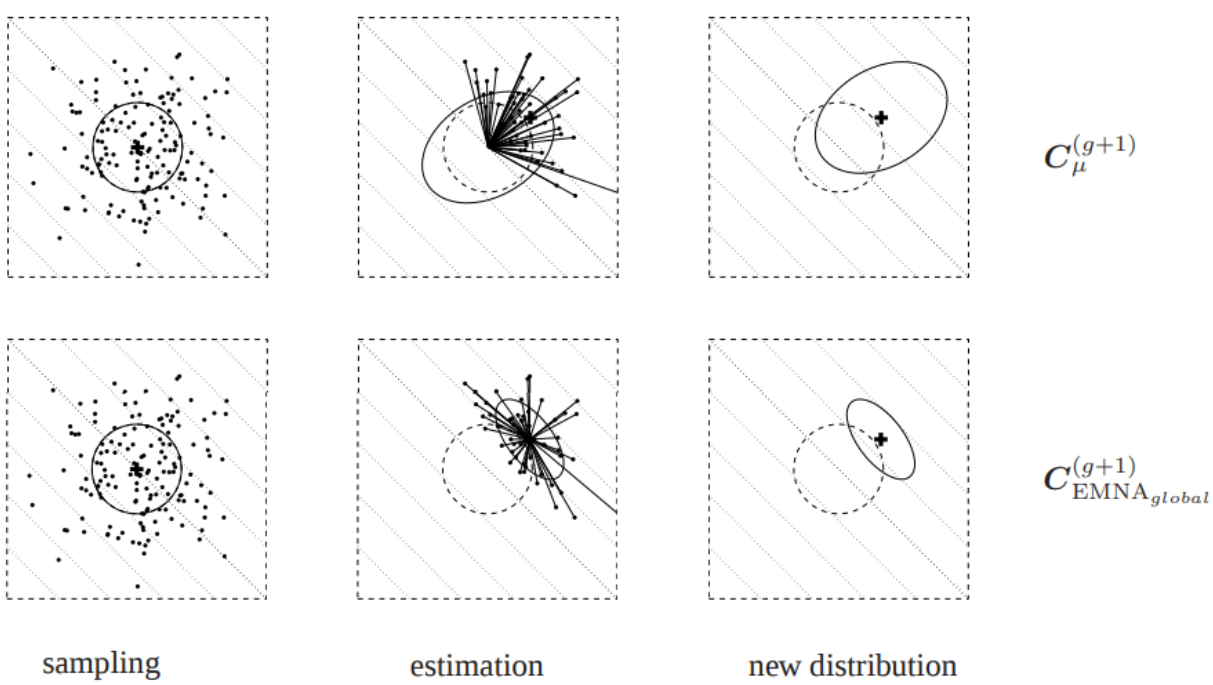



Figure 3.3: The distribution of candidate solutions when using the covariance matrix update from (3.4) (top row) or (3.6) (bottom row). Notice how (3.6) *decreases* the variance of candidate solutions in the direction that improves the cost function. Image source: [51].

An alternative way to update the covariance matrix is to average all of the covariance matrices given in (3.4) from previous generations, i.e.:

$$C^{(g+1)} = \frac{1}{g+1} \sum_{i=0}^{g} \frac{1}{(\sigma^{(i)})^2} C_{\mu}^{(i+1)}$$

where $(\sigma^{(i)})^2$ is incorporated to make $C_{\mu}^{(i+1)}$ comparable for each $i$. However, this weighting scheme may place too much emphasis on covariance matrices from the distant past. To address this potential shortcoming, CMA-ES uses an exponential moving average update:

$$C^{(g+1)} = (1 - c_{\mu}) C^{(g)} + c_{\mu} \frac{1}{(\sigma^{(i)})^2} C_{\mu}^{(g+1)}. \tag{3.7}$$

where $c_{\mu} \in [0, 1]$ is a learning rate specifying how much information is retained versus learned between generations[3]. Intuitively, (3.7), which is referred to as the "rank-$\mu$ update", increases the likelihood of successful steps by following the *natural* gradient of the expected fitness [51, 50].

Still another way to update $C$ is by omitting explicit covariance matrix estimates altogether and instead relying on information contained in the history of changes to $m$, i.e., its "evolution path". The evolution path $p_c$ is also updated as an exponential moving average:

$$p_c^{(g+1)} = (1 - c_c) p_c^{(g)} + \sqrt{c_c(2 - c_c)\mu_{\text{eff}}} \, \frac{m^{(g+1)} - m^{(g)}}{\sigma^{(g)}} \tag{3.8}$$

where $c_c$ is a learning rate and $\sqrt{c_c(2 - c_c)\mu_{\text{eff}}}$ is a carefully chosen normalization constant that ensures $p_c^{(g+1)} \sim \mathcal{N}(0, C^{(g)})$ under random selection.[4] $C^{(g)}$ can then be updated with:

$$C^{(g+1)} = (1 - c_1) C^{(g)} + c_1 p_c^{(g+1)} p_c^{(g+1)^T} \tag{3.9}$$

where $c_1$ is a learning rate. (3.9) is referred to as the "rank-one update" and contains information on correlations between generations, which makes it useful when dealing with

[3]For this reason, $\frac{1}{c_{\mu}}$ is known as the "backward time horizon".

[4]See [51] for a detailed explanation.

small populations [50]. CMA-ES integrates both the population-level information contained in the rank-$\mu$ update (3.7) and the intergenerational information contained in the rank-one update (3.9) into a single update for $C$:

$$C^{(g+1)} = (1-c_1-c_\mu)C^{(g)}+c_1 p_c^{(g+1)} p_c^{(g+1)^T} + c_\mu \sum_{i=1}^{\mu} w_i \left(x_{i:\lambda}^{(g+1)} - m^{(g)}\right)\left(x_{i:\lambda}^{(g+1)} - m^{(g)}\right)^T \quad (3.10)$$

### 3.2.3 Updating $\sigma$

While the covariance matrix update controls the distribution of future step *directions*, it does not allow for explicit control of future step *sizes*. [51] provides two motivations for including step-size adaptation in the CMA-ES algorithm:

1. The covariance matrix update cannot approximate the optimal step size well.
2. The best learning rate for updating the covariance matrix is too slow for changing the overall step size effectively.

To update $\sigma$, CMA-ES uses what is referred to as "cumulative step length adaption" (CSA). Intuitively, if the distance of an evolution path from some starting point is short, CMA-ES must be backtracking (see Fig. 3.4), so the step size should be decreased. Alternatively, if the evolution path is long, CMA-ES must be consistently moving in the same direction, so the step size should be increased.[5] Lastly, if the evolution path is neither short nor long (i.e., when steps tend to be perpendicular), then CMA-ES is moving as fast as it can, so the step size should remain the same.

To determine whether the length of an evolution path is "short" or "long", we can compare the path's length with its expected length under random selection. Given a vector $p_\sigma^{(g+1)}$ summarizing an evolution path, CMA-ES uses the following update rule for $\sigma$:

[5] The conceptual similarities between CSA and methods like momentum should be apparent [50].

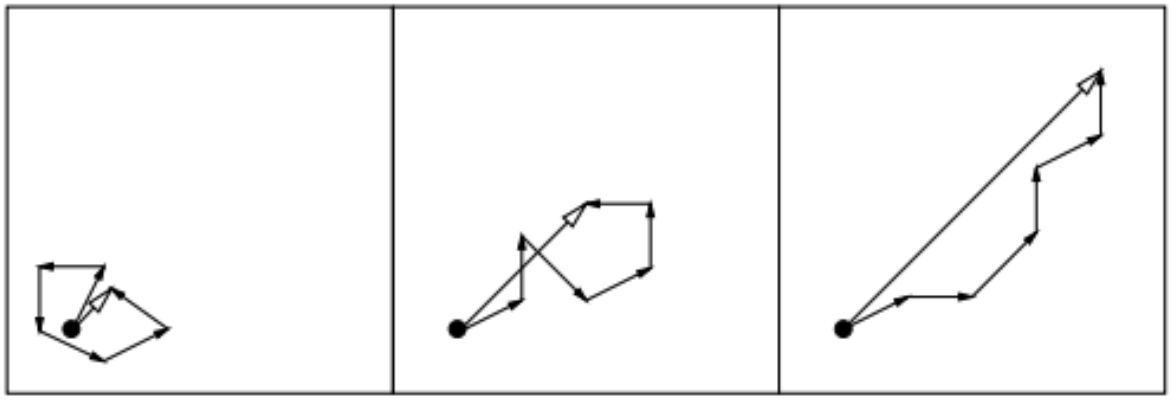


Figure 3.4: Example evolution paths of six steps (image source: [51]).

$$\sigma^{(g+1)} = \sigma^{(g)} \times \exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{\|p_\sigma^{(g+1)}\|}{\mathbb{E}\|\mathcal{N}(0,I)\|} - 1\right)\right) \tag{3.11}$$

where $c_\sigma^{-1}$ is the backward time horizon for $\sigma$, $d_\sigma$ is a damping parameter, and $\mathbb{E}\|\mathcal{N}(0,I)\|$ is the average Euclidean norm for $\mathcal{N}(0,I)$ distributed vectors. When $\|p_\sigma^{(g+1)}\| \approx \mathbb{E}\|\mathcal{N}(0,I)\|$, the exponent in (3.11) is approximately zero and the step size remains approximately the same. Similarly, longer evolution paths will increase the step size while shorter evolution paths will decrease the step size.

Importantly, the length of $p_\sigma$ is direction agnostic (which is not the case for $p_c$), which is why it is referred to as the "*conjugate* evolution path" or "*isotropic* evolution path". The update rule for $p_\sigma$:

$$p_\sigma^{(g+1)} = (1 - c_\sigma)\, p_\sigma^{(g)} + \sqrt{c_\sigma\,(2 - c_\sigma)\,\mu_{\text{eff}}}(C^{(g)})^{-\frac{1}{2}}\,\frac{m^{(g+1)} - m^{(g)}}{\sigma^{(g)}} \tag{3.12}$$

is nearly identical to (3.8) except for the inclusion of $(C^{(g)})^{-\frac{1}{2}}$, the square root of the inverse of $C^{(g)}$. $(C^{(g)})^{-\frac{1}{2}}$ makes the expected length of $p_\sigma^{(g+1)}$ independent of its direction and ensures $p_\sigma^{(g+1)} \sim \mathcal{N}(0,I)$ under random selection.[6]

[6]See [51] for a detailed explanation.

### 3.3 Experiments and results

For our CMA-ES experiments, we used the same optimization framework described in Chapter 2, and we used the FD approach described in Chapter 2 as our baseline. We compared the effectiveness of CMA-ES to FD for adversarial pose discovery by generating targeted adversarial poses for a 3D school bus object. We used the `pycma`[7] Python package for our CMA-ES implementation. The code for our experiments can be found at: `https://github.com/airalcorn2/strike-with-a-pose/blob/master/paper_code/optimizer_example.py`. Because the FD approach performed 18 forward passes through the neural network per iteration[8], we used $\lambda = 18$ for our CMA-ES experiments to make the computational demands comparable between the two algorithms (the vast majority of the computation for each iteration occurs in the forward pass).

For each optimization method, we ran 20 trials targeting each of the 20 most common classifications for the school bus object[9]. For each (target, trial) pair, we generated a starting point using the $z_\delta$-constrained random search procedure described in Chapter 2, and we used this *same* starting point to initialize both FD and CMA-ES runs. For each (method, target, trial) triple, we recorded both the best target probability $p_{o,t,l}$[10] discovered by the method and whether the target misclassification occurred. Each experiment was allowed to run for 100 iterations.

In 94% of the trials, CMA-ES discovered a higher target probability than FD (see Fig. 3.5). Following Chapter 2, we present "median of medians" statistics where we first take the median of quantities of interest for a specific object, and then take the median of these median values across objects. For each $(o, t)$ pair, we calculated the median $p_{o,t,l}$ across all trials and denote it $p_{o,t}$. We then calculated the difference of these median values between the optimization methods $p_t = p_{e,t} - p_{f,t}$ where $e$ indicates CMA-ES and $f$ indicates FD.

[7] `https://github.com/CMA-ES/pycma`

[8] Two forward passes for each of the nine pose parameters.

[9] For a total of $2 \times 20 \times 20 = 800$ experiments.

[10] $o$ denotes the optimization method, $t$ denotes the target, and $l$ denotes the trial number.

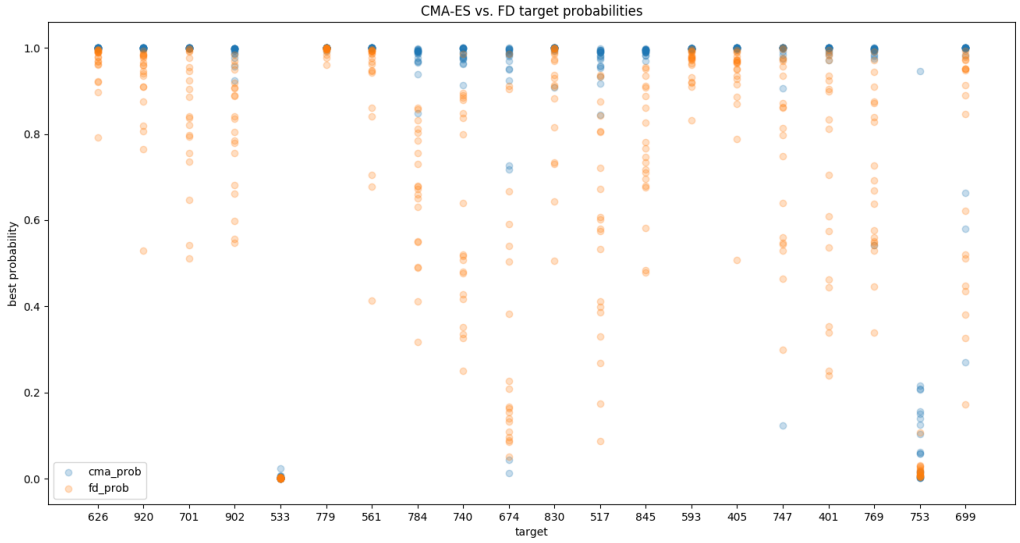



Figure 3.5: Target probabilities for the CMA-ES and FD algorithms. Each point represents a different trial.

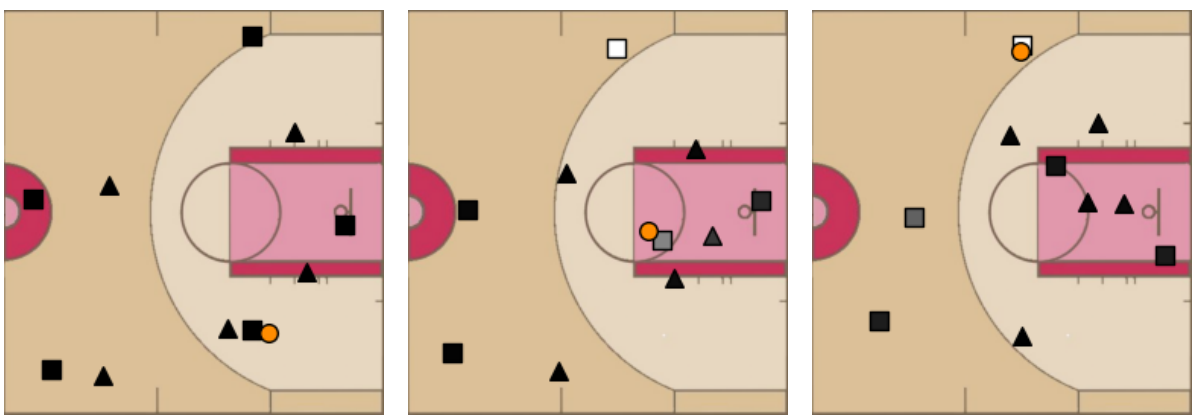
The median $p_t$ was 0.12, i.e., the median $p_{e,t}$ was often at least 0.12 higher than the median $p_{f,t}$. In fact, for all 20 targets, the median $p_{e,t}$ was greater than the median $p_{f,t}$. For 18 of the 20 target classes, there was no difference in the number of trials leading to a "hit" between the two methods. However, for two targets, CMA-ES had two and 11 more hits than FD. Combined, these results provide strong evidence that CMA-ES is much better suited for the adversarial pose discovery task than FD.

# Chapter 4

# `baller2vec`: A Multi-Entity Transformer For Multi-Agent Spatiotemporal Modeling

## 4.1 Introduction

Whether it be a defender anticipating where the point guard will make a pass in a game of basketball, a marketing professional guessing the next trending topic on a social media platform, or a theme park manager forecasting the flow of visitor traffic, humans frequently attempt to predict phenomena arising from processes involving multiple entities interacting through time. When designing learning algorithms to perform such tasks, researchers face two main challenges:



(a) $t = 4$ (b) $t = 8$ (c) $t = 13$

Figure 4.1: After solely being trained to predict the trajectory of the ball (●) given the locations of the players and the ball on the court through time, a self-attention (SA) head in `baller2vec` learned to anticipate passes. When the ball handler (■●) is driving towards the basket at $t = 4$, SA assigns near-zero weights (black) to all players, suggesting no passes will be made. Indeed, the ball handler did not pass and dribbled into the lane ($t = 8$). SA then assigns a high weight (white) to a teammate (□), which correctly identifies the recipient of the pass at $t = 13$.

1. Given that entities lack a natural ordering, how do you effectively model interactions between entities across time?
2. How do you *efficiently* learn from the large, high-dimensional inputs inherent to such sequential data?

Prior work in athlete trajectory forecasting, a widely studied application of multi-agent spatiotemporal modeling (MASM; where entities are agents moving through space), has attempted to model player interactions through "role-alignment" preprocessing steps (i.e., imposing an order on the players) [37, 141] or graph neural networks [136], but these approaches may destroy identity information in the former case (see Section 4.4.2) or limit personalization in the latter case (see Section 4.5.1). Recently, researchers have experimented with variational recurrent neural networks (VRNNs) [26] to model the temporal aspects of player trajectory data [136, 141], but the inherently sequential design of this architecture limits the size of models that can feasibly be trained in experiments.

Transformers [127] were designed to circumvent the computational constraints imposed by other sequential models, and they have achieved state-of-the-art results in a wide variety of sequence learning tasks, both in natural language processing (NLP), e.g., GPT-3 [16], and computer vision, e.g., Vision Transformers [31]. While Transformers have successfully been applied to *static* multi-entity data, e.g., graphs [128], the only published work we are aware of that attempts to model multi-entity *sequential* data with Transformers uses four different Transformers to *separately* process information temporally and spatially before merging the sub-Transformer outputs [137].

In this chapter, we introduce a *multi-entity* Transformer that, with minimal assumptions, is capable of *simultaneously* integrating information across agents and time, which gives it powerful representational capabilities. We adapt the original Transformer architecture to suit multi-entity sequential data by converting the standard self-attention mask matrix used in NLP tasks into a novel self-attention mask *tensor*. To test the effectiveness

of our multi-entity Transformer for MASM, we train it to perform two different basketball-related tasks (hence the name `baller2vec`): (1) simultaneously forecasting the trajectories of all players on the court (**Task P**) and (2) forecasting the trajectory of the ball (**Task B**). Further, we convert these tasks into classification problems by binning the Euclidean trajectory space, which allows `baller2vec` to learn complex, multimodal trajectory distributions via strictly maximizing the likelihood of the data (in contrast to variational approaches, which maximize the evidence lower bound and thus require priors over the latent variables). We find that:

1. `baller2vec` is an effective learning algorithm for MASM, obtaining a perplexity of 1.64 on **Task P** (compared to 15.72 when simply using the label distribution from the training set) and 13.44 on **Task B** (vs. 316.05) (Section 4.4.1). Further, compared to a graph recurrent neural network (GRNN) with similar capacity, `baller2vec` is $\sim$3.8 times faster and achieves an 10.5% lower average negative log-likelihood (NLL) on **Task P** (Section 4.4.1).
2. `baller2vec` demonstrably integrates information across *both* agents and time to achieve these results, as evidenced by ablation experiments (Section 4.4.2).
3. The identity embeddings learned by `baller2vec` capture idiosyncratic qualities of players, indicative of the model's deep personalization capabilities (Section 4.4.3).
4. `baller2vec`'s trajectory forecast distributions depend on both the historical and current context (Section 4.4.4), and several attention heads appear to perform different basketball-relevant functions (Figure 4.1; Section 4.4.5), which suggests the model learned to "understand" the sport.

## 4.2 Methods

### 4.2.1 Multi-entity sequences

Let $A = \{1, 2, \ldots, B\}$ be a set indexing $B$ entities and $P = \{p_1, p_2, \ldots, p_K\} \subset A$ be the $K$ entities involved in a particular sequence. Further, let $Z_t = \{z_{t,1}, z_{t,2}, \ldots, z_{t,K}\}$ be an

*unordered* set of $K$ feature vectors such that $z_{t,k}$ is the feature vector at time step $t$ for entity $p_k$. $\mathcal{Z} = (Z_1, Z_2, \ldots, Z_T)$ is thus an *ordered* sequence of sets of feature vectors over $T$ time steps. When $K = 1$, $\mathcal{Z}$ is a sequence of individual feature vectors, which is the underlying data structure for many NLP problems.

We now consider two different tasks: (1) sequential entity labeling, where each entity has its own label at each time step (which is conceptually similar to word-level language modeling), and (2) sequential labeling, where each time step has a single label (see Figure 4.3). For (1), let $\mathcal{V} = (V_1, V_2, \ldots, V_T)$ be a sequence of sets of labels corresponding to $\mathcal{Z}$ such that $V_t = \{v_{t,1}, v_{t,2}, \ldots, v_{t,K}\}$ and $v_{t,k}$ is the label at time step $t$ for the entity indexed by $k$. For (2), let $W = (w_1, w_2, \ldots, w_T)$ be a sequence of labels corresponding to $\mathcal{Z}$ where $w_t$ is the label at time step $t$. The goal is then to learn a function $f$ that maps a set of entities and their time-dependent feature vectors $\mathcal{Z}$ to a probability distribution over either (1) the entities' time-dependent labels $\mathcal{V}$ or (2) the sequence of labels $W$.

### 4.2.2 Multi-agent spatiotemporal modeling

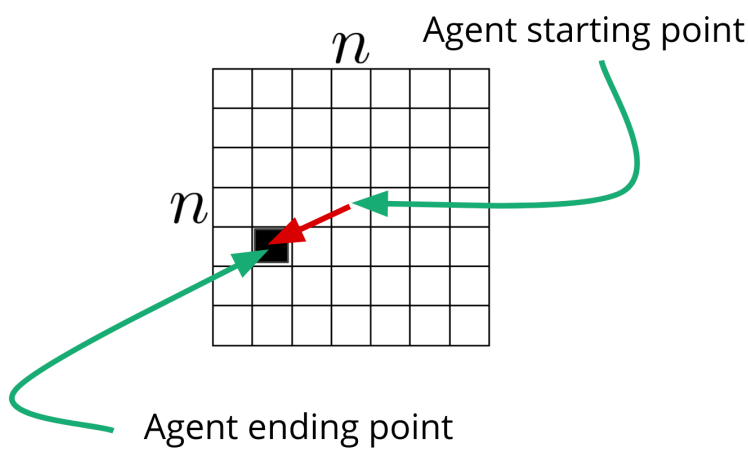



Figure 4.2: An example of a binned trajectory. The agent's starting position is at the center of the grid, and the cell containing the agent's ending position is used as the label (of which there are $n^2$ possibilities).

In the MASM setting, $P$ is a set of $K$ different agents and $C_t = \{(x_{t,1}, y_{t,1}), (x_{t,2}, y_{t,2}), \ldots, (x_{t,K}, y_{t,K})\}$ is an unordered set of $K$ coordinate pairs such that $(x_{t,k}, y_{t,k})$ are the coordinates for agent

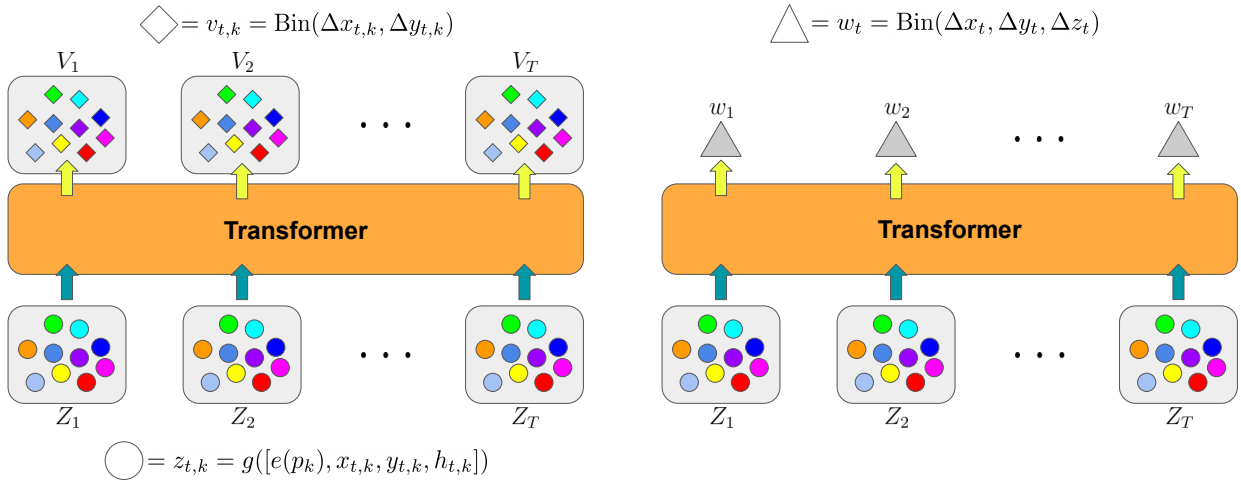



Figure 4.3: An overview of our multi-entity Transformer, `baller2vec`. Each time step $t$ consists of an *unordered* set $Z_t$ of entity feature vectors (colored circles) as the input, with either (**left**) a corresponding set $V_t$ of entity labels (colored diamonds) or (**right**) a single label $w_t$ (gray triangle) as the target. Matching colored circles/diamonds across time steps correspond to the same entity. In our experiments, each entity feature vector $z_{t,k}$ is produced by an MLP $g$ that takes a player's identity embedding $e(p_k)$, raw court coordinates $(x_{t,k}, y_{t,k})$, and a binary variable indicating the player's frontcourt $h_{t,k}$ as input. Further, each entity label $v_{t,k}$ is an integer indexing the trajectory bin derived from the player's raw trajectory, while each $w_t$ is an integer indexing the ball's trajectory bin.

$p_k$ at time step $t$. The ordered sequence of sets of coordinates $\mathcal{C} = (C_1, C_2, \ldots, C_T)$, together with $P$, thus defines the trajectories for the $K$ agents over $T$ time steps. We then define $z_{t,k}$ as:

$$z_{t,k} = g([e(p_k), x_{t,k}, y_{t,k}, h_{t,k}]) \tag{4.1}$$

where $g$ is a multilayer perceptron (MLP), $e$ is an agent embedding layer, and $h_{t,k}$ is a vector of optional contextual features for agent $p_k$ at time step $t$. The trajectory for agent $p_k$ at time step $t$ is defined as $(x_{t+1,k} - x_{t,k}, y_{t+1,k} - y_{t,k})$. Similar to [145], to fully capture the multimodal nature of the trajectory distributions, we binned the 2D Euclidean space into an $n \times n$ grid (Figure 4.2) and treated the problem as a classification task. Therefore, $\mathcal{Z}$ has a corresponding sequence of sets of trajectory labels (i.e., $v_{t,k} = \text{Bin}(\Delta x_{t,k}, \Delta y_{t,k})$ is an integer from one to $n^2$), and the loss for each sample in **Task P** is:

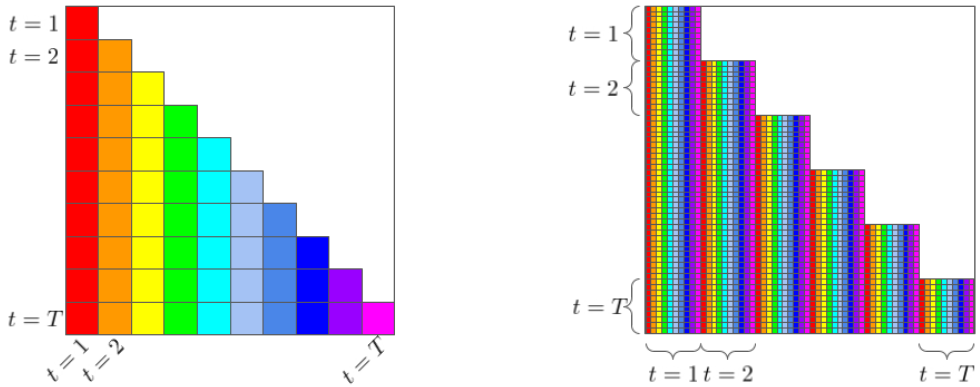



Figure 4.4: **Left**: the standard self-attention mask matrix $M$. The element at $M_{t_1,t_2}$ indicates whether or not the model can "look" at time step $t_2$ when processing time step $t_1$. **Right**: the matrix form of our multi-entity self-attention mask tensor. In tensor form, element $M_{t_1,k_1,t_2,k_2}$ indicates whether or not the model can "look" at agent $p_{k_2}$ at time step $t_2$ when processing agent $p_{k_1}$ at time step $t_1$. In matrix form, this corresponds to element $M_{t_1K+k_1,t_2K+k_2}$ when using zero-based indexing. The $M$ shown here is for a static, fully connected graph, but other, potentially evolving network structures can be encoded in the attention mask tensor.

$$\mathcal{L} = \sum_{t=1}^{T} \sum_{k=1}^{K} -\ln(f(\mathcal{Z})_{t,k}[v_{t,k}]) \tag{4.2}$$

where $f(\mathcal{Z})_{t,k}[v_{t,k}]$ is the probability assigned to the trajectory label for agent $p_k$ at time step $t$ by $f$; i.e., Equation (4.2) is the NLL of the data according to the model.

For **Task B**, the loss for each sample is:

$$\mathcal{L} = \sum_{t=1}^{T} -\ln(f(\mathcal{Z})_{t}[w_{t}])$$

where $f(\mathcal{Z})_{t}[w_{t}]$ is the probability assigned to the trajectory label for the ball at time step $t$ by $f$, and the labels correspond to a binned 3D Euclidean space (i.e., $w_t = \text{Bin}(\Delta x_t, \Delta y_t, \Delta z_t)$ is an integer from one to $n^3$).

### 4.2.3 The multi-entity Transformer

We now describe our multi-entity Transformer, `baller2vec` (Figure 4.3). For NLP tasks, the Transformer self-attention mask $M$ takes the form of a $T \times T$ matrix (Figure 4.4)

where $T$ is the length of the sequence. The element at $M_{t_1,t_2}$ thus indicates whether or not the model can "look" at time step $t_2$ when processing time step $t_1$. Here, we generalize the standard Transformer to the multi-entity setting by employing a $T \times K \times T \times K$ mask *tensor* where element $M_{t_1,k_1,t_2,k_2}$ indicates whether or not the model can "look" at agent $p_{k_2}$ at time step $t_2$ when processing agent $p_{k_1}$ at time step $t_1$. Here, we mask all elements where $t_2 > t_1$ and leave all remaining elements unmasked, i.e., `baller2vec` is a "causal" model.

In practice, to be compatible with Transformer implementations in major deep learning libraries, we reshape $M$ into a $TK \times TK$ matrix (Figure 4.4) and the input to the Transformer is a matrix with shape $TK \times F$ where $F$ is the dimension of each $z_{t,k}$. [56] observed that positional encoding [127] is not only unnecessary, but detrimental for Transformers that use a causal attention mask, so we do not use positional encoding with `baller2vec`. The remaining computations are identical to the standard Transformer (see code).

## 4.3 Experiments

### 4.3.1 Dataset

We trained `baller2vec` on a publicly available dataset of player and ball trajectories recorded from 631 National Basketball Association (NBA) games from the 2015-2016 season.[1] All 30 NBA teams and 450 different players were represented. Because transition sequences are a strategically important part of basketball, unlike prior work, e.g., [37, 136, 141], we did not terminate sequences on a change of possession, nor did we constrain ourselves to a fixed subset of sequences. Instead, each training sample was generated on the fly by first randomly sampling a game, and then randomly sampling a starting time from that game. The following four seconds of data were downsampled to 5 Hz from the original 25 Hz and used as the input.

Because we did not terminate sequences on a change of possession, we could not normalize the direction of the court as was done in prior work [37, 136, 141]. Instead, for each

[1] `https://github.com/linouk23/NBA-Player-Movements`

sampled sequence, we randomly (with a probability of 0.5) rotated the court 180°(because the court's direction is arbitrary), doubling the size of the dataset. We used a training/validation/test split of 569/30/32 games, respectively (i.e., 5% of the games were used for testing, and 5% of the remaining 95% of games were used for validation). As a result, we had access to ∼82 million different (albeit overlapping) training sequences (569 games × 4 periods per game × 12 minutes per period × 60 seconds per minute × 25 Hz × 2 rotations), ∼800x the number of sequences used in prior work. For both the validation and test sets, ∼1,000 different, *non-overlapping* sequences were selected for evaluation by dividing each game into $\lceil \frac{1{,}000}{N} \rceil$ non-overlapping chunks (where $N$ is the number of games), and using the starting four seconds from each chunk as the evaluation sequence.

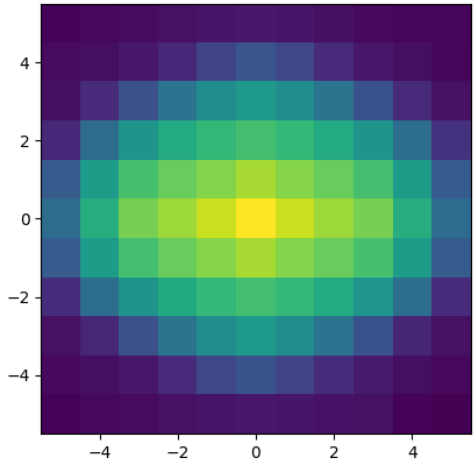



Figure 4.5: A log-normalized heat map of the trajectory bins in the full NBA dataset. The center bin corresponds to the "stationary" trajectory (i.e., the player did not move further than 0.5 ft in either the $x$ or $y$ direction). The elongated shape of the distribution reflects the rectangular shape of the court.

### 4.3.2 Model

We trained separate models for **Task P** and **Task B**. For all experiments, we used a single Transformer architecture that was nearly identical to the original model described in [127], with $d_{\text{model}} = 512$ (the dimension of the input and output of each Transformer layer), eight attention heads, $d_{\text{ff}} = 2048$ (the dimension of the inner feedforward layers), and six

layers, although we did not use dropout. For *both* **Task P** and **Task B**, the players *and* the ball were included in the input, and both the players and the ball were embedded to 20-dimensional vectors. The input features for each player consisted of his identity, his $(x, y)$ coordinates on the court at each time step in the sequence, and a binary variable indicating the side of his frontcourt (i.e., the direction of his team's hoop).[2] The input features for the ball were its $(x, y, z)$ coordinates at each time step.

The input features for the players and the ball were processed by separate, three layer MLPs (Equation (4.1)) before being fed into the Transformer. Each MLP had 128, 256, and 512 nodes in its three layers, respectively, and a ReLU nonlinearity following each of the first two layers. For classification, a single linear layer was applied to the Transformer output followed by a softmax. For players, we binned an 11 ft $\times$ 11 ft 2D Euclidean trajectory space into an $11 \times 11$ grid of 1 ft $\times$ 1 ft squares for a total of 121 player trajectory labels (Figure 4.5 shows the distribution of the player trajectory bins for the entire dataset). Similarly, for the ball, we binned a 19 ft $\times$ 19 ft $\times$ 19 ft 3D Euclidean trajectory space into a $19 \times 19 \times 19$ grid of 1 ft $\times$ 1 ft $\times$ 1 ft cubes for a total of 6,859 ball trajectory labels.

We used the Adam optimizer [61] with an initial learning rate of $10^{-6}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-9}$ to update the model's parameters, of which there were $\sim$19/23 million for **Task P**/**Task B**, respectively. The learning rate was reduced to $10^{-7}$ after 20 consecutive epochs of the validation loss not improving. Models were implemented in PyTorch and trained on a single NVIDIA GTX 1080 Ti GPU for seven days ($\sim$650 epochs) where each epoch consisted of 20,000 training samples, and the validation set was used for early stopping.

### 4.3.3 Baselines

As our naive baseline, we used the marginal distribution of the trajectory bins from the training set for all predictions. For our strong baseline, we implemented a `baller2vec`-like

[2] We did not include team identity as an input variable because teams are collections of players and a coach, and coaches did not vary as we only had access to half of one season of data; however, with additional seasons of data, we would include the coach as an input variable.

graph recurrent neural network (GRNN) and trained it on **Task P** (code is available in the `baller2vec` repository).[3] Specifically, at each time step, the player and ball inputs were first processed using MLPs as in `baller2vec` (Equation (4.1)), and these inputs were then fed into a graph neural network (GNN) as in [136]. The node and edge functions of the GNN were each a Transformer-like feedforward network (TFF):

$$\text{TFF}(x) = \text{LN}(x + W_2\text{ReLU}(W_1 x + b_1) + b_2)$$

where LN is Layer Normalization [10], $W_1$ and $W_2$ are weight matrices, $b_1$ and $b_2$ are bias vectors, and ReLU is the rectifier activation function. For our RNN, we used a gated recurrent unit (GRU) RNN [24] in which we replaced each of the six weight matrices of the GRU with a TFF. Each TFF had the same dimensions as the Transformer layers used in `baller2vec`. Our GRNN had ∼18M parameters, which is comparable to the ∼19M in `baller2vec`. We also trained our GRNN for seven days (∼175 epochs).

### 4.3.4 Ablation studies

To assess the impacts of the multi-entity design and player embeddings of `baller2vec` on model performance, we trained three variations of our **Task P** model using: (1) one player in the input without player identity, (2) all 10 players in the input without player identity, and (3) all 10 players in the input with player identity. In experiments where player identity was not used, a single generic player embedding was used in place of the player identity embeddings. We also trained two variations of our **Task B** model: one with player identity and one without. Lastly, to determine the extent to which `baller2vec` uses historical information in its forecasts, we compared the performance of our best **Task P** model on the full sequence test set with its performance on the test set when *only predicting the trajectories for the first frame* (i.e., we applied the *same* model to a reduced test set).

[3] We chose to implement our own strong baseline because `baller2vec` has far more parameters than models from prior work (e.g., ∼70x [37]).

## 4.4 Results

### 4.4.1 `baller2vec` is an effective learning algorithm for multi-agent spatiotemporal modeling.

Table 4.1: The perplexity per trajectory on the test set when using `baller2vec` vs. the marginal distribution of the trajectory bins in the training set ("Train") for all predictions. `baller2vec` considerably reduces the uncertainty over the trajectory bins.

| | `baller2vec` | Train |
|---|---|---|
| **Task P** | 1.64 | 15.72 |
| **Task B** | 13.44 | 316.05 |

Table 4.2: The average NLL (lower is better) on the **Task P** test set and seconds per training epoch (SPE) for `baller2vec` (`b2v`) and our GRNN. `baller2vec` trains ∼3.8 times faster per epoch compared to our GRNN, and `baller2vec` outperformed our GRNN by 10.5% when given the same amount of training time. Even when only allowed to train for half ("0.5x") and a quarter ("0.25x") as long as our GRNN, `baller2vec` outperformed our GRNN by 9.1% and 1.5%, respectively..

| | `b2v` | `b2v` (0.5x) | `b2v` (0.25x) | GRNN |
|---|---|---|---|---|
| NLL | 0.492 | 0.499 | 0.541 | 0.549 |
| SPE | ∼900 | ∼900 | ∼900 | ∼3,400 |

The average NLL on the test set for our best **Task P** model was 0.492, while the average NLL for our best **Task B** model was 2.598. In NLP, model performance is often expressed in terms of the perplexity per word, which, intuitively, is the number of faces on a fair die that has the same amount of uncertainty as the model per word (i.e., a uniform distribution over $M$ labels has a perplexity of $M$, so a model with a per word perplexity of six has the same average uncertainty as rolling a fair six-sided die). In our case, we consider the perplexity per trajectory, defined as:

$$PP = e^{\frac{1}{NTK}\sum_{n=1}^{N}\sum_{t=1}^{T}\sum_{k=1}^{K} -\ln(p(v_{n,t,k}))}$$

where $N$ is the number of sequences. Our best **Task P** model achieved a perplexity per trajectory of 1.64, i.e., `baller2vec` was, on average, as uncertain as rolling a 1.64-sided fair die (better than a coin flip) when predicting player trajectories (Table 4.1). For comparison, when using the distribution of the player trajectory labels in the training set as the predicted probabilities, the perplexity on the test set was 15.72. Our best **Task B** model achieved a perplexity per trajectory of 13.44 (compared to 316.05 when using the training set distribution).

Compared to our GRNN, `baller2vec` was ∼3.8 times faster and had a 10.5% lower average NLL when given an equal amount of training time (Table 4.2). Even when only given half as much training time as our GRNN, `baller2vec` had a 9.1% lower average NLL.

### 4.4.2 `baller2vec` uses information about all players on the court through time, in addition to player identity, to model spatiotemporal dynamics.

Table 4.3: The average NLL on the test set for each of the models in our ablation experiments (lower is better). For **Task P**, using all 10 players improved model performance by 18.0%, while using player identity improved model performance by an additional 4.4%. For **Task B**, using player identity improved model performance by 2.7%. 1/10 indicates whether one or 10 players were used as input, respectively, while I/NI indicates whether or not player identity was used, respectively.

| Task | 1-NI | 10-NI | 10-I |
|---|---|---|---|
| **Task P** | 0.628 | 0.515 | 0.492 |
| **Task B** | N/A | 2.670 | 2.598 |

Results for our ablation experiments can be seen in Table 4.3. Including all 10 players in the input dramatically improved the performance of our **Task P** model by 18.0% vs. only including a single player. Including player identity improved the model's performance

a further 4.4%. This stands in contrast to [37] where the inclusion of player identity led to slightly *worse* model performance; a counterintuitive result given the range of skills among NBA players, but possibly a side effect of their role-alignment procedure. Additionally, when replacing the players in each test set sequence with random players, the performance of our best **Task P** model deteriorated by 6.2% from 0.492 to 0.522. Interestingly, including player identity only improved our **Task B** model's performance by 2.7%. Lastly, our best **Task P** model's performance on the full sequence test set (0.492) was 70.6% better than its performance on the single frame test set (1.67), i.e., `baller2vec` is clearly using historical information to model the spatiotemporal dynamics of basketball.

#### 4.4.3 `baller2vec`'s learned player embeddings encode individual attributes.

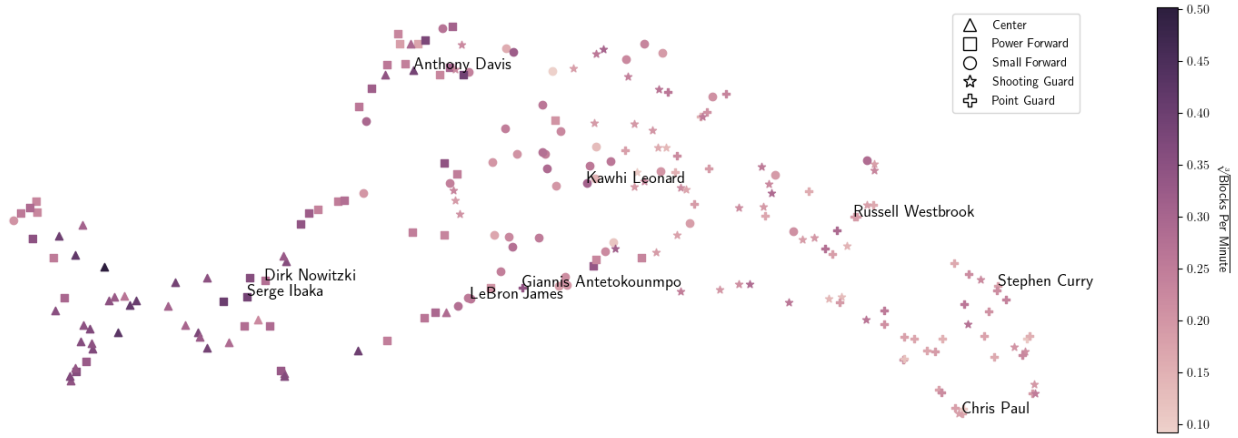



Figure 4.6: As can be seen in this 2D UMAP of the player embeddings, by exclusively learning to predict the trajectory of the ball, `baller2vec` was able to infer idiosyncratic player attributes. The left-hand side of the plot contains tall post players (△, □), e.g., Serge Ibaka, while the right-hand side of the plot contains shorter shooting guards (☆) and point guards (+), e.g., Stephen Curry. The connecting transition region contains forwards (□, ○) and other "hybrid" players, i.e., individuals possessing both guard and post skills, e.g., LeBron James. Further, players with similar defensive abilities, measured here by the cube root of the players' blocks per minute in the 2015-2016 season [12], cluster together.

Neural language models are widely known for their ability to encode semantic relationships between words and phrases as geometric relationships between embeddings—see, e.g., [84, 83, 73, 116]. [5] observed a similar phenomenon in a baseball setting, where batters and

pitchers with similar skills were found next to each other in the embedding space learned by a neural network trained to predict the outcome of an at-bat. A 2D UMAP [82] of the player embeddings learned by `baller2vec` for **Task B** can be seen in Figure 4.6. Like `(batter|pitcher)2vec` [5], `baller2vec` seems to encode skills and physical attributes in its player embeddings.

Figure 4.7: Nearest neighbors in `baller2vec`'s embedding space are plausible doppelgängers, such as the explosive point guards Russell Westbrook and Derrick Rose, and seven-foot tall brothers Pau and Marc Gasol.

Querying the nearest neighbors for individual players reveals further insights about the `baller2vec` embeddings. For example, the nearest neighbor for Russell Westbrook, an extremely athletic 6'3" point guard, is Derrick Rose, a 6'2" point guard also known for his athleticism (Figure 4.7). Amusingly, the nearest neighbor for Pau Gasol, a 7'1" center with a respectable shooting range, is his younger brother Marc Gasol, a 6'11" center, also with a respectable shooting range.

### 4.4.4 `baller2vec`'s trajectory forecast distributions depend on both the historical and current context.

Because `baller2vec` *explicitly* models the distribution of the player trajectories (unlike variational methods), we can easily visualize how its trajectory forecast distributions shift

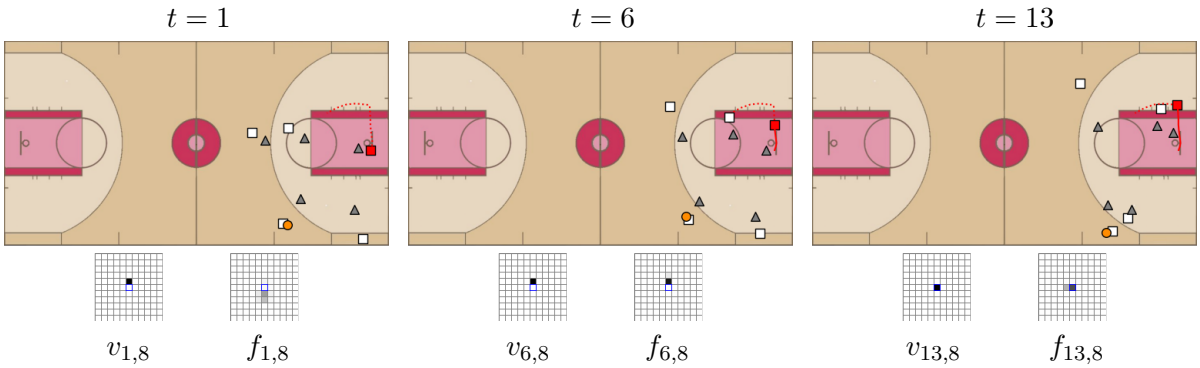
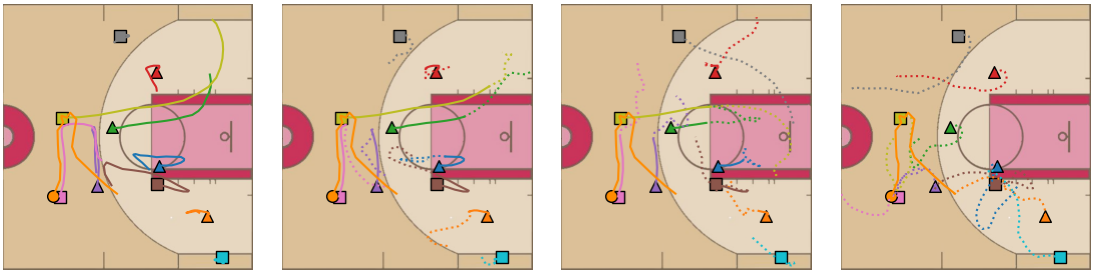



Figure 4.8: `baller2vec`'s trajectory forecast distributions are affected by both the historical and current context. At $t = 1$, `baller2vec` is fairly uncertain about the target player's (▲; $k = 8$) trajectory (left grid and dotted red line; the blue-bordered center cell is the "stationary" trajectory), with most of the probability mass divided between trajectories moving towards the ball handler's sideline (right grid; black = 1.0; white = 0.0). After observing a portion of the sequence ($t = 6$), `baller2vec` becomes very certain about the target player's trajectory ($f_{6,8}$), but when the player reaches a decision point ($t = 13$), `baller2vec` becomes split between trajectories (staying still or moving towards the top of the key). ● = ball, □ = offense, ▲ = defense, and $f_{t,k} = f(\mathcal{Z})_{t,k}$.

in different situations. As can be seen in Figure 4.8, `baller2vec`'s trajectory forecast distributions depend on both the historical and current context. When provided with limited historical information, `baller2vec` tends to be less certain about where the players might go. `baller2vec` also tends to be more certain when forecasting "easy" trajectories (e.g., a player moving into open space) vs. "hard" trajectories (e.g., an offensive player choosing which direction to move around a defender). Similarly, as can be seen in Figure 4.9, player trajectories *generated* by `baller2vec` appear less natural when conditioned on less historical context.

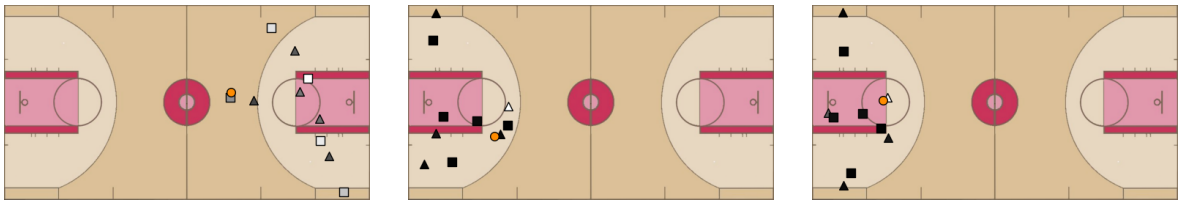
### 4.4.5 Attention heads in `baller2vec` appear to perform basketball-relevant functions.

One intriguing property of the attention mechanism [43, 44, 131, 11] is how, when visualized, the attention scores often seem to reveal how a model is "thinking". For example, [127] discovered examples of attention heads in their Transformer that appear to be performing various language understanding subtasks, such as anaphora resolution. As can be seen



(a) Ground truth. (b) 2 s observed. (c) 1 s observed. (d) 0 s observed.

Figure 4.9: With less historical information (solid lines), `baller2vec`'s generated trajectories (dotted lines) tend to look less natural. Each player is a different color, and the colored lines are their corresponding trajectories. The shapes are in the *starting* positions for the trajectories, and the ground truth location of the ball is used as input at each time step (i.e., the ball trajectories are not generated). △ = offense, □ = defense, and ● = ball.



(a) Attention head 2-7 (layer-head) appears to focus on teammates of the ball handler (■●).

(b) Attention head 6-2 seems to predict who the ball handler will be in future frames (△); see (c).

(c) In this frame, the highlighted player in (b) receives a pass from the previous ball handler.

Figure 4.10: The attention outputs from `baller2vec` suggest it learned basketball-relevant functions. Players are shaded according to the *sum* of the attention scores assigned to the players *through time* with reference to the ball in the current frame (recall that each player occurs multiple times in the input). Higher attention scores are lighter. For both of these attention heads, the sum of the attention scores assigned to the ball through time was small (0.01 for both the left and middle frames where the maximum is 1.00).

in Figure 4.10, some of the attention heads in `baller2vec` seem to be performing basketball understanding subtasks, such as keeping track of the ball handler's teammates, and anticipating who the ball handler will pass to, which, intuitively, help with our task of predicting the ball's trajectory.

## 4.5 Related Work

### 4.5.1 Trajectory modeling in sports

There is a rich literature on MASM, particularly in the context of sports, e.g., [60, 145, 72, 71, 95, 140]. Most relevant to our work is [136], which used a variational recurrent neural network combined with a graph neural network to forecast trajectories in a multi-agent setting. Like their approach, our model is permutation equivariant with regard to the ordering of the agents; however, we use a multi-head attention mechanism to achieve this permutation equivariance while the permutation equivariance in [136] is provided by the graph neural network. Specifically, [136] define:

$$v \rightarrow e : \mathbf{e}_{i,j} = f_e([\mathbf{v}_i, \mathbf{v}_j, \mathbf{t}_{i,j}]) \tag{4.3}$$

$$e \rightarrow v : \mathbf{o}_i = f_v(\sum_{j \in \mathcal{N}_i} [\mathbf{e}_{i,j}, \mathbf{t}_i])$$

where $\mathbf{v}_i$ is the initial state of agent $i$, $\mathbf{t}_{i,j}$ is an embedding for the edge between agents $i$ and $j$, $\mathbf{e}_{i,j}$ is the representation for edge $(i, j)$, $\mathcal{N}_i$ is the neighborhood for agent $i$, $\mathbf{t}_i$ is a node embedding for agent $i$, $\mathbf{o}_i$ is the output state for agent $i$, and $f_e$ and $f_v$ are deep neural networks.

Assuming *each individual player* is a different "type" in Equation (4.3) (i.e., attempting to maximize the level of personalization) would require $450^2 = 202{,}500$ (i.e., $B^2$) different $t_{i,j}$ edge embeddings, many of which would never be used during training and thus inevitably lead to poor out-of-sample performance. Reducing the number of type embeddings requires making assumptions about the nature of the relationships between nodes. By using a multi-head attention mechanism, `baller2vec` learns to integrate information about different agents in a highly flexible manner that is both agent and time-dependent, and can generalize to unseen agent combinations. The attention heads in `baller2vec` are somewhat analogous to edge types, but, importantly, they do not require a priori knowledge about the relationships between the players.

Additionally, unlike recent works that use variational methods to train their generative models [136, 37, 141], we translate the multi-agent trajectory forecasting problem into a classification task, which allows us to train our model by strictly maximizing the likelihood of the data. As a result, we do not make any assumptions about the distributions of the trajectories nor do we need to set any priors over latent variables. [145] also predicted binned trajectories, but they used a recurrent convolutional neural network to predict the trajectory for a single player trajectory at a time at each time step.

### 4.5.2 Transformers for multi-agent spatiotemporal modeling

[40] used a Transformer to forecast the trajectories of *individual* pedestrians, i.e., the model does not consider interactions between individuals. [137] used *separate* temporal and spatial Transformers to forecast the trajectories of multiple, interacting pedestrians. Specifically, the temporal Transformer processes the coordinates of each pedestrian *independently* (i.e., it does not model interactions), while the spatial Transformer, which is inspired by Graph Attention Networks [128], processes the pedestrians *independently at each time step.* [107] used a Transformer to classify on-the-ball events from sequences in soccer games; however, only the coordinates of the $K$-nearest players to the ball were included in the input (along with the ball's coordinates). Further, the *order* of the included players was based on their average distance from the ball for a given temporal window, which can lead to specific players changing position in the input between temporal windows. As far as we are aware, `baller2vec` is the **first** Transformer capable of processing all agents *simultaneously across time* without imposing an arbitrary order on the agents.

## 4.6 Limitations

While `baller2vec` does not have a mechanism for handling unseen players, a number of different solutions exist depending on the data available. For example, similar to what was proposed in [5], a model could be trained to map a vector of (e.g., NCAA) statistics

and physical measurements to `baller2vec` embeddings. Alternatively, if tracking data is available for the other league, a single `baller2vec` model could be jointly trained on all the data.

At least two different factors may explain why including player identity as an input to `baller2vec` only leads to relatively small performance improvements. First, both player and ball trajectories are fairly generic—players tend to move into open space, defenders tend to move towards their man or the ball, point guards tend to pass to their teammates, and so on. Further, the location of a player on the court is often indicative of their position, and players playing the same position tend to have similar skills and physical attributes. As a result, we might expect `baller2vec` to be able to make reasonable guesses about a player's/ball's trajectory just given the location of the players and the ball on the court.

Second, `baller2vec` may be able to *infer* the identities of the players directly from the spatiotemporal data. Unlike `(batter|pitcher)2vec` [5], which was trained on several seasons of Major League Baseball data, `baller2vec` only had access to one half of one season's worth of NBA data for training. As a result, player identity may be entangled with season-specific factors (e.g., certain rosters or coaches) that are actually exogenous to the player's intrinsic qualities, i.e., `baller2vec` may be overfitting to the season. To provide an example, the Golden State Warriors ran a very specific kind of offense in the 2015-2016 season—breaking the previous record for most three-pointers made in the regular season by 15.4%—and many basketball fans could probably recognize them from a bird's eye view (i.e., without access to any identifying information). Given additional seasons of data, `baller2vec` would no longer be able to exploit the implicit identifying information contained in static lineups and coaching strategies, so including player identity in the input would likely be more beneficial in that case.

## 4.7 Conclusion

In this chapter, we introduced `baller2vec`, a generalization of the standard Transformer that can model sequential data consisting of multiple, unordered entities at each time step. As an architecture that both is computationally efficient and has powerful representational capabilities, we believe `baller2vec` represents an exciting new direction for MASM. As discussed in Section 4.6, training `baller2vec` on more training data may allow the model to more accurately factor players away from season-specific patterns. With additional data, more contextual information about agents (e.g., a player's age, injury history, or minutes played in the game) and the game (e.g., the time left in the period or the score difference) could be included as input, which might allow `baller2vec` to learn an even more complete model of the game of basketball. Although we only experimented with static, fully connected graphs here, `baller2vec` can easily be applied to more complex inputs—for example, a sequence of graphs with changing nodes and edges—by adapting the self-attention mask tensor as appropriate. Lastly, as a generative model (see Chapter 5 for a full derivation), `baller2vec` could be used for counterfactual simulations (e.g., assessing the impact of different rosters), or combined with a controller to discover optimal play designs through reinforcement learning.

# Chapter 5

# `baller2vec++`: A Look-Ahead Multi-Entity Transformer For Modeling Coordinated Agents

## 5.1 Introduction and Related Work

Whether it is a team executing a play in a game of basketball, a family navigating to an attraction in a theme park, or friends posting about a birthday party on a social media platform, humans frequently coordinate their behavior in response to shared information. When this coordinating information is unobserved (which is often the case in many machine learning datasets), the individuals' observed behaviors become correlated, i.e., the behavior of one individual at a specific time contains information about the behavior of another individual *at the same time*. In the context of forecasting agent trajectories in multi-agent spatiotemporal systems, this property translates to the trajectories being statistically dependent at each time step. However, nearly all multi-agent spatiotemporal models (e.g., [37, 46, 102, 136, 137, 6]) implicitly (through their loss functions) assume the trajectories of the agents at each time step are statistically *independent* given the agents' previous locations (Figure 5.1).

The macro-intents are generated from a separately trained recurrent neural network (RNN) that learns to predict a future, coarse, "stationary" location for each agent at each time step. The macro-intents for all of the agents at a specific time step are concatenated together to form a single, shared, macro-intent variable, which is then provided as input to the trajectories-generating model at that time step. However, similar to the previously mentioned multi-agent trajectory models, the macro-intents model implicitly assumes the macro-intents for the agents at each time step are statistically independent, i.e., the generated

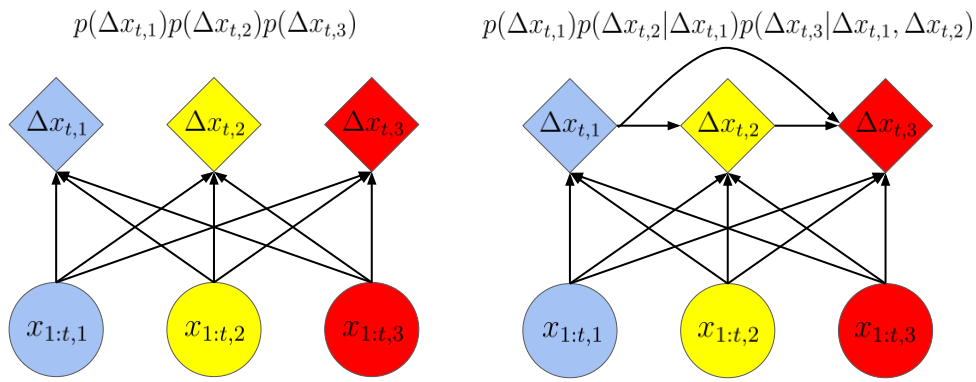



Figure 5.1: **Left**: most multi-agent systems implicitly assume the trajectories of the agents at each time step ($\Delta x_{t,k}$) are conditionally independent given the agents' previous positions ($x_{1:t,k}$). **Right**: however, the various decompositions of the joint probability of the trajectories, e.g., $p(\Delta x_{t,1})p(\Delta x_{t,2}|\Delta x_{t,1})p(\Delta x_{t,3}|\Delta x_{t,1}\Delta x_{t,2})$ (note, we omit the conditional $x_{1:t,k}$ terms for brevity), suggest more complex statistical dependencies between the agents' trajectories can exist (i.e., the independence assumption is an extremely strong one). Indeed, there are often shared unobserved variables influencing the spatiotemporal behaviors of agents—such as the play that the players on a basketball team are executing, or events occurring in a pedestrian environment—which suggests statistical dependencies between the agents' trajectories are likely.

macro-intent for one agent does not depend on the generated macro-intents of the other agents.[1] [141] explicitly focused on modeling coordinated multi-agent trajectories, using "macro-intents" [145] that are shared across agents to do so. Further, the trajectories-generating model still implicitly assumes the trajectories of the agents at each time step are independent, which is only true if the shared macro-intent variable perfectly captures all of the unobserved information that could cause the agents' trajectories to be correlated.

Notably, Social-BiGAT [65] *does* partially account for trajectory correlations through a global adversarial loss. Specifically, the trajectories for each agent are separately passed through an encoder RNN, and the final hidden states for each agent are then processed with a graph attention network (GAT) [129]. The output of the GAT is then used as an input to a discriminator that classifies whether or not the input trajectories are real or generated. This global adversarial loss, however, is only a single component of the full Social-BiGAT

[1]See the authors' implementation here: `https://github.com/ezhan94/multiagent-programmatic-supervision/blob/a1d9152d4c8a287474953cba093c28fef2a05979/models/macro_vrnn.py#L101`.

loss function, and other components of the loss function *do* implicitly make the independence assumption. Further, interestingly, adding the global discriminator to a baseline model only improved the model's performance for one out of six pedestrian datasets.

In this chapter, we describe a novel multi-agent spatiotemporal model that integrates information about *concurrent* actions of agents to predict statistically dependent distributions of trajectories. Specifically, we extend `baller2vec` (described in Chapter 4) by: (1) augmenting its input with a parallel sequence of "look-ahead" agent trajectories and (2) using a specially designed self-attention mask, which allows our model to exploit the chain rule of probability (Section 5.3). We find that:

1. `baller2vec++` is an effective learning algorithm for modeling coordinated agents. Unlike `baller2vec`, `baller2vec++` can learn to emulate perfectly coordinated agents from a simulated toy dataset (Section 5.5.1). Further, `baller2vec++` outperforms `baller2vec` by a wide margin (8.9%) when modeling the trajectories of professional basketball players (Section 5.5.1).

2. `baller2vec++` makes better forecasts when conditioned on concurrent trajectory information from other agents, supporting our proposition that the commonly used independence assumption for agent trajectories is overly strong (Section 5.5.2).

3. Lastly, the joint probability assigned to a sequence by `baller2vec++` is approximately permutation invariant with respect to the order of the agents, i.e., `baller2vec++` respects the properties of the chain rule (Section 5.5.3).

## 5.2 Background

### 5.2.1 Multi-agent trajectory modelling

Our problem setup is identical to the one found in Chapter 4, which we restate here:

> Let $A = \{1, 2, \ldots, B\}$ be a set indexing $B$ agents and $P = \{p_1, p_2, \ldots, p_K\} \subset A$ be the $K$ agents involved in a particular sequence. [Let] $C_t = \{(x_{t,1}, y_{t,1}), (x_{t,2}, y_{t,2}), \ldots, (x_{t,K}, y_{t,K})\}$ [be] an unordered set of $K$ coordinate pairs such that $(x_{t,k}, y_{t,k})$ are the coordinates for agent $p_k$ at time step $t$. The ordered sequence of sets of coordinates $\mathcal{C} = (C_1, C_2, \ldots, C_T)$, together with $P$, thus defines the trajectories for the $K$ agents over $T$ time steps.

In multi-agent trajectory modeling, the goal is to model a joint probability of the form:

$$p(\Delta x_{t,1}, \Delta x_{t,2}, \ldots, \Delta x_{t,K} | x_{1:t,1}, x_{1:t,2}, \ldots, x_{1:t,K})$$

i.e., the joint probability of the $K$ agents' trajectories $\Delta x_{t,k}$ at time step $t$ given the agents' location histories $x_{1:t,k}$. We note here that the common practice of *simultaneously* predicting the trajectories for all of the agents at a specific time step is not required by theory. Using the chain rule of probability, the joint probability of the agents' trajectories can be factorized as, e.g.:

$$p(\Delta x_{t,1}, \Delta x_{t,2}, \ldots, \Delta x_{t,K}) = p(\Delta x_{t,1}) p(\Delta x_{t,2} | \Delta x_{t,1}) \ldots p(\Delta x_{t,K} | \Delta x_{t,1}, \Delta x_{t,2}, \ldots, \Delta x_{t,K-1})$$

where we omit the conditional historical trajectories for brevity. As a result, it is perfectly acceptable to generate trajectories agent-wise, using the previously generated trajectories as additional conditioning information when generating the trajectories for later agents (see Figure 5.2).

### 5.2.2 `baller2vec` is a (conditional) generative model.

Here, we provide a probabilistic interpretation of `baller2vec`, which establishes the theoretical grounds for using the chain rule to generate trajectories agent-wise at each time

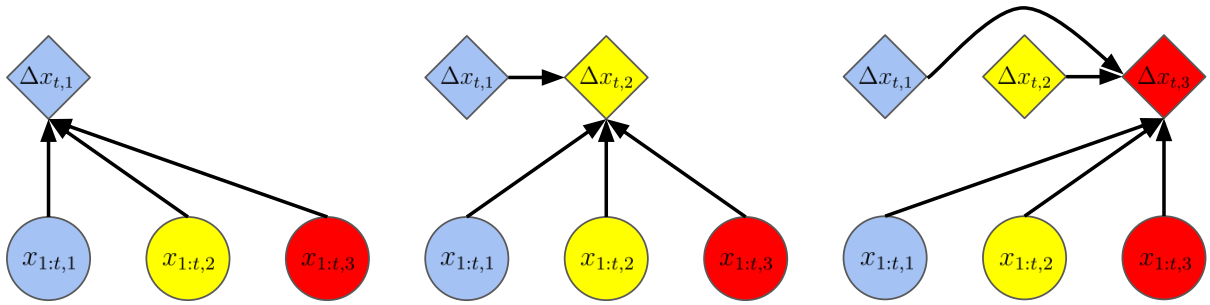



Figure 5.2: At inference time, a model is not *required* to *simultaneously* generate the trajectories for all of the agents at a specific time step. An alternative strategy is to allow the model to generate the agents' trajectories one at a time, and let the model use the previously generated trajectories to inform the trajectories it generates for the remaining agents.

step in `baller2vec++`. Without loss of generality, we only consider one-dimensional trajectories for a single agent here. To briefly summarize, the outputs of the softmax function over the $n$ binned trajectories in `baller2vec` can be interpreted as mixture proportions for a mixture of uniform distributions with predetermined bounds that partition the Euclidean trajectory space. Further, because $x_{t+1} = x_t + \Delta x_t$, `baller2vec` is in fact a conditional generative model that assigns a probability to a sequence of trajectories given the initial position of the agent, i.e., $p(\Delta x_1, \Delta x_2, \ldots, \Delta x_T | x_1)$. Using the chain rule, we decompose the joint probability of the trajectories as:

$$p(\Delta x_1, \Delta x_2, \ldots, \Delta x_T) = p(\Delta x_1) p(\Delta x_2 | \Delta x_1) \ldots p(\Delta x_T | \Delta x_1, \Delta x_2, \ldots, \Delta x_{T-1})$$

to reflect their temporal structure (we omit the conditional initial position term for brevity). Therefore, new trajectories can be generated from `baller2vec` with the following procedure (see Figure 5.3):

1. First, sample one of the $n$ different mixture components using the mixture proportions output from the classifier $f$ (i.e., `baller2vec`) conditioned on the agent's current position, i.e., $i \sim \text{Categorical}(\pi_1, \pi_2, \ldots, \pi_n)$ where $[\pi_1, \pi_2, \ldots, \pi_n] = f(x_t)$.

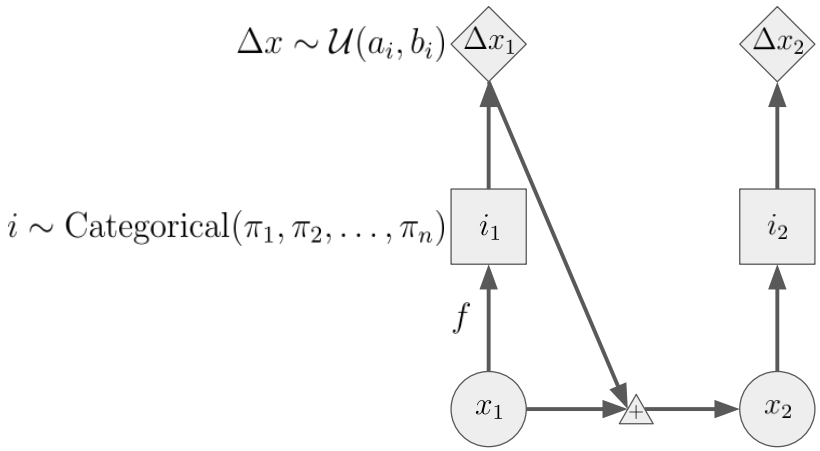



Figure 5.3: `baller2vec` can be viewed as a conditional generative model that assigns a probability to a sequence of trajectories given the initial positions of the agents. Here, we show a graphical model depiction of a `baller2vec` model that generates a sequence of one-dimensional trajectories for a single agent. Given the initial position of the agent (the circle containing $x_1$), one of $n$ different uniform distributions (the square containing $i_1$) is sampled using the mixture proportions ($\pi_i$) output by `baller2vec` ($f$). The agent's trajectory (the diamond containing $\Delta x_1$) is then sampled from the selected uniform distribution, which has bounds $-\infty < a_i < b_i < \infty$. At the start of the next time step, the agent's position is $x_2 = x_1 + \Delta x_1$. Maximizing the likelihood of `baller2vec` as a classifier over the binned trajectories is thus equivalent to maximizing its likelihood when assuming the trajectories are generated from a mixture of uniform distributions that partition the Euclidean trajectory space (see Section 5.2.2 for details).

2. Next, sample a trajectory from the uniform distribution associated with the sampled component, i.e., $\Delta x_t \sim \mathcal{U}(a_i, b_i)$.

3. Finally, add the sampled trajectory to the agent's input position to generate the agent's position at the start of the next time step, i.e., $x_{t+1} = x_t + \Delta x_t$.

Let $[\Delta x_{min}, \Delta x_{max})$ be an interval on the real line such that any trajectory $\Delta x < \Delta x_{min}$ or $\Delta x \geq \Delta x_{max}$ has zero density (i.e., such trajectories are humanly impossible). Let $\{[a_i, b_i)\}_{i=1}^{n}$ be a set of $n$ intervals that partition the interval $[\Delta x_{min}, \Delta x_{max})$ into $n$ bins, i.e., $\cup_{i=1}^{n}[a_i, b_i) = [\Delta x_{min}, \Delta x_{max})$ and $i \neq j \implies [a_i, b_i) \cap [a_j, b_j) = \emptyset$. Recall that the probability density function (PDF) for a uniform distribution with bounds $-\infty < a < b < \infty$ is:

$$p(\Delta x) = \begin{cases} \frac{1}{b-a} & \text{for } \Delta x \in [a, b) \\ 0 & \text{otherwise} \end{cases}$$

Letting $c_i = \frac{1}{b_i - a_i}$, the PDF for a mixture of uniforms with these bounds is thus:

$$p(\Delta x) = \sum_{i=1}^{n} \pi_i \mathcal{U}(\Delta x; a_i, b_i) = \sum_{i=1}^{n} \pi_i c_i \tag{5.1}$$

where $p(\Delta x)$ is the density assigned to $\Delta x$ by the mixture, $\pi_i$ is the mixture proportion for the mixture component indexed by $i$ (i.e., $0 \leq \pi_i \leq 1$ and $\sum \pi_i = 1$), and $\mathcal{U}(\Delta x; a_i, b_i)$ is the density assigned to $\Delta x$ by the uniform distribution with bounds $-\infty < a_i < b_i < \infty$. Because the bounds of the uniform distributions partition $[\Delta x_{min}, \Delta x_{max})$, Equation (5.1) reduces to:

$$p(\Delta x) = \pi_{i'} c_{i'}$$

where $\Delta x \in [a_{i'}, b_{i'})$ (because the other uniform distributions will assign a density of zero to $\Delta x$). The likelihood for data $D$ (with $|D| = N$) is then:

$$\mathcal{L}(D) = \prod_{j=1}^{N} p(\Delta x_j) = \prod_{j=1}^{N} \pi_{j,i'} c_{j,i'}$$

where $\pi_{j,i'}$ is the mixture proportion assigned to the component with $\Delta x_j \in [a_{i'}, b_{i'})$ and $c_{j,i'}$ is the associated density. Taking the negative logarithm of the likelihood gives:

$$-\ln(\mathcal{L}(D)) = -\sum_{j=1}^{N} \ln(\pi_{j,i'}) - \sum_{j=1}^{N} \ln(c_{j,i'}) \tag{5.2}$$

Because the bounds are fixed, the second summation is a constant, and Equation (5.2) becomes:

$$-\ln(\mathcal{L}(D)) = -\sum_{j=1}^{N} \ln(\pi_{j,i'}) + C$$

where $C = -\sum_{j=1}^{N} \ln(c_{j,i'})$. Therefore, minimizing the loss of `baller2vec` as a classifier of binned trajectories is equivalent to minimizing the loss of the model when assuming the trajectories are generated from a mixture of uniform distributions as specified in Equation (5.1).

## 5.3 Model Architecture

We motivate our `baller2vec++` architecture by first highlighting an issue that arises in `baller2vec` when trying to model agent trajectories using the chain rule. The `baller2vec` self-attention mask can be adapted so that `baller2vec` "looks ahead" at the future positions of agents whose trajectories are generated prior to the agent being processed in the current time step (Figure 5.4). However, this look-ahead self-attention mask can only be used with the final layer of the Transformer; otherwise, the model needs to see the future (Figure 5.4). As a result, `baller2vec` is severely limited in the conditional distribution functions it can learn.

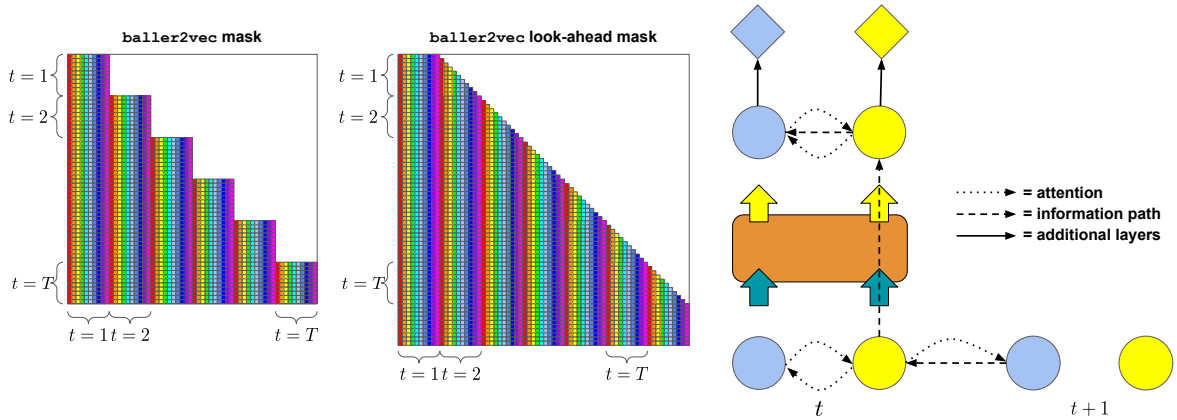



Figure 5.4: A naive strategy for learning to predict statistically dependent agent trajectories is to adapt the `baller2vec` self-attention mask so that `baller2vec` can "look ahead" at future positions of agents whose trajectories are generated *prior* to the agent being processed in the current time step. However, this look-ahead self-attention mask cannot be used with multi-layer Transformers because doing so necessitates "seeing the future". For example, after the model attends to the blue agent's position at time step $t + 1$ when processing the yellow agent at time step $t$, the yellow agent's resultant feature vector contains information about the blue agent's future position. As a result, when the model attends to the yellow agent while processing the blue agent at the next level, the model is seeing the future.

`baller2vec++` (Figure 5.5) overcomes this limitation by: (1) augmenting the `baller2vec` input with two other sets of feature vectors and (2) using a specially designed self-attention mask. The three sets of feature vectors in `baller2vec++` take the following forms:

1. $z_{t,k} = g_z([e(p_k), x_{t,k}, y_{t,k}, h_{t,k}])$ (**current location information**)
2. $u_{t,k} = g_u([e(p_k), x_{t+1,k}, y_{t+1,k}, h_{t,k}, \Delta x_{t,k}, \Delta y_{t,k}])$ (**"look-ahead" information**)
3. $r_k = g_r([e(p_k), x_{1,k}, y_{1,k}, h_{1,k}])$ (**initial location information**)

where $g_z$, $g_u$, and $g_r$ are multilayer perceptrons (MLPs), $e$ is an agent embedding layer, and $h_{t,k}$ is a vector of optional contextual features for agent $p_k$ at time step $t$. $z_{t,k}$ is the same location feature vector used in `baller2vec` and contains information about a specific agent's identity and the agent's location at time step $t$. $u_{t,k}$ is a "look-ahead" trajectory feature vector that contains information about a specific agent's identity, the agent's location at the *next time step* $t + 1$, and the agent's trajectory at time step $t$, i.e.,

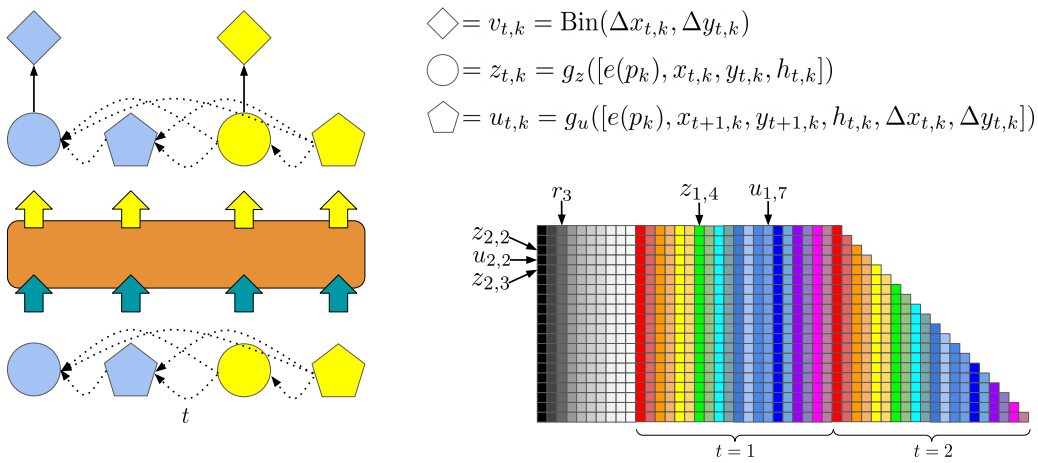



Figure 5.5: To learn statistically dependent agent trajectories, `baller2vec++` uses a specially designed self-attention mask to simultaneously process three different sets of features vectors in a single Transformer. The three sets of feature vectors consist of location feature vectors like those found in `baller2vec` ($z_{t,k}$), look-ahead trajectory feature vectors ($u_{t,k}$), and starting location feature vectors ($r_k$; not shown). As can be seen in these partial depictions of `baller2vec++` and the `baller2vec++` self-attention mask, this design allows the model to integrate information about *concurrent* agent trajectories through *multiple* Transformer layers without seeing the future.

$(x_{t+1,k} - x_{t,k}, y_{t+1,k} - y_{t,k})$. Lastly, $r_k$ is a starting location feature vector that contains information about a specific agent's identity and the agent's location at time step $t = 1$. The $r_k$ feature vectors are necessary so that `baller2vec++` can "see" the initial locations of *all the agents* when processing the agents agent-wise in the first time step.

These three sets of feature vectors are combined to form a $(K + 2TK) \times F$ matrix $Z$ such that the first $K$ rows consist of the $K$ $r_k$ feature vectors, and the remaining $2TK$ rows consist of the $TK$ $z_{t,k}$ and $TK$ $u_{t,k}$ feature vectors interleaved with one another, i.e., each $z_{t,k}$ is followed by its corresponding $u_{t,k}$ in the matrix. This matrix is passed into the Transformer along with the specially designed self-attention mask, which encodes the following relationships (see Figure 5.5):

1. When processing $r_{k_1}$, `baller2vec++` is exclusively allowed to "look" at each $r_{k_2}$ (i.e., `baller2vec++` cannot look at any location or look-ahead feature vectors when processing $r_{k_1}$).

2. When processing $z_{t_2,k_2}$, `baller2vec++` is allowed to "look" at: (i) each $r_{k_1}$, (ii) any $z_{t_1,k_1}$ where (a) $t_1 < t_2$ **or** (b) $t_1 = t_2$ **and** $k_1 \leq k_2$, and (iii) any $u_{t_1,k_1}$ where (a) $t_1 < t_2$ **or** (b) $t_1 = t_2$ **and** $k_1 < k_2$.

3. When processing $u_{t_2,k_2}$, `baller2vec++` is allowed to "look" at: (i) each $r_{k_1}$, (ii) any $z_{t_1,k_1}$ where (a) $t_1 < t_2$ **or** (b) $t_1 = t_2$ **and** $k_1 \leq k_2$, and (iii) any $u_{t_1,k_1}$ where (a) $t_1 < t_2$ **or** (b) $t_1 = t_2$ **and** $k_1 \leq k_2$.

Each processed $z_{t,k}$ feature vector is then passed through a linear layer followed by a softmax to classify the trajectory bin $v_{t,k}$ (where $v_{t,k} = \text{Bin}(\Delta x_{t,k}, \Delta y_{t,k})$ is an integer from one to $n^2$) for agent $p_k$ at time step $t$. As with `baller2vec`, the loss for each sample is:

$$\mathcal{L} = \sum_{t=1}^{T} \sum_{k=1}^{K} -\ln(f(Z)_{t,k}[v_{t,k}]) \tag{5.3}$$

where $f(Z)_{t,k}[v_{t,k}]$ is the probability assigned to the trajectory bin $v_{t,k}$ by $f$, i.e., Equation (5.3) is the negative log-likelihood (NLL) of the data according to the model.

Because any ordering of a chain rule decomposition of a joint probability produces the same value, e.g.:

$$p(\Delta x_{t,1})p(\Delta x_{t,2}|\Delta x_{t,1})p(\Delta x_{t,3}|\Delta x_{t,1}\Delta x_{t,2}) = p(\Delta x_{t,3})p(\Delta x_{t,2}|\Delta x_{t,3})p(\Delta x_{t,1}|\Delta x_{t,3}\Delta x_{t,2})$$

like [135], we shuffled the order of the agents in each training sequence to encourage the model to learn joint probabilities of the agent trajectories that are permutation invariant with respect to the ordering of the agents.

## 5.4 Experiments

We tested `baller2vec++` on two different datasets. To highlight the pathological behavior of models that assume agent trajectories are statistically independent at each time

step, we trained scaled down versions of `baller2vec++` and `baller2vec` on a toy dataset consisting of simulated trajectories for two perfectly coordinated agents. Additionally, to demonstrate the efficacy of `baller2vec++` in real world settings, we trained `baller2vec++` and `baller2vec` on a dataset of trajectories for professional basketball players.

### 5.4.1 Toy dataset

Each training sample was initialized with the agents starting at $(-1, 0)$ and $(1, 0)$ on a grid in random order (i.e., the first agent could be placed to either the left or the right of the origin). At each time step, one of nine actions (corresponding to the $3\times3$ grid surrounding the agent) was sampled from a uniform distribution, and each of the agents was translated along this trajectory. This process was repeated for 20 time steps (see Figure 5.6(a) for a sample).

### 5.4.2 Basketball dataset

We used the same National Basketball Association (NBA) dataset described in Chapter 4.

### 5.4.3 Model

Our `baller2vec++` and `baller2vec` models for the basketball dataset closely followed the architecture described in Chapter 4, and so largely resemble the original Transformer architecture [127]. Specifically, the Transformer settings were: $d_{\text{model}} = 512$ (the dimension of the input and output of each Transformer layer), eight attention heads, $d_{\text{ff}} = 2048$ (the dimension of the inner feedforward layers), six layers, no dropout, and no positional encoding. Each MLP (i.e., $g_z$, $g_u$, and $g_r$) had 128, 256, and 512 nodes in its three layers, respectively, and a ReLU nonlinearity following each of the first two layers. The player embeddings [5] had 20 dimensions, and $h_{t,k}$ was a binary variable indicating the side of the frontcourt for

player $p_k$ (i.e., the direction of his team's hoop) at time step $t$. Lastly, the 11 ft $\times$ 11 ft 2D Euclidean trajectory space was binned into 121 1 ft $\times$ 1 ft squares.

We used the Adam optimizer [61] with an initial learning rate of $10^{-6}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-9}$ to update the model parameters, of which there were $\sim$19 million. The learning rate was reduced to $10^{-7}$ after 20 epochs of the validation loss not improving. Models were implemented in PyTorch and trained on a single NVIDIA GTX 1080 Ti GPU for $\sim$650 epochs (seven days) where each epoch consisted of 20,000 training samples, and the validation set was used for early stopping.

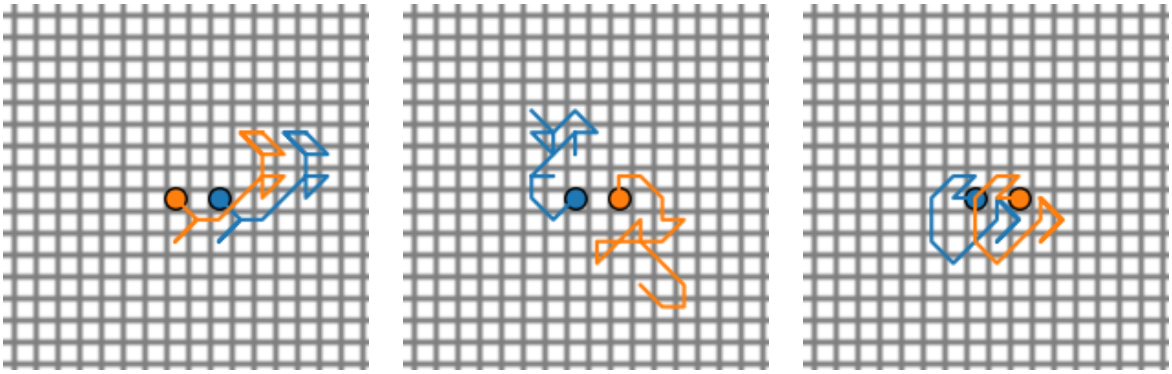
For the toy dataset, we used scaled down versions of the basketball models with $d_{\text{model}} = 128$, four attention heads, $d_{\text{ff}} = 512$, and two layers in the Transformer. Additionally, each MLP had two layers with 64 and 128 nodes, respectively. The models were trained for 50 epochs of 500 samples per epoch ($\sim$10.5 minutes) using a single learning rate of $10^{-5}$.

## 5.5 Results

### 5.5.1 `baller2vec++` can effectively model coordinated agents in both simulated and real settings

For the toy dataset, the training loss for `baller2vec` converged to $\sim 2.2 \approx -\ln(\frac{1}{9})$, i.e., the model was simply independently guessing the trajectories for both agents at every time step. In contrast, the training loss for `baller2vec++` converged to $\sim 1.1 \approx -\ln(\frac{1}{9}) \div 2$, which is the expected loss for a model that perfectly learns the deterministic relationship between the agents' trajectories (because the prediction for the second agent will always contribute $-\ln(1.0) = 0$ to the loss). When generating trajectories with `baller2vec`, the agents are completely uncoordinated, with each agent following an independent random walk around the grid (Figure 5.6(b)). In contrast, trajectories generated by `baller2vec++` display the same coordinated agent behavior as the training data (Figure 5.6(c)).

For the basketball dataset, `baller2vec++` achieved an average NLL of 0.472 on the test set, 8.9% better than the average NLL for `baller2vec` (0.518). As can be seen in Figure 5.7,



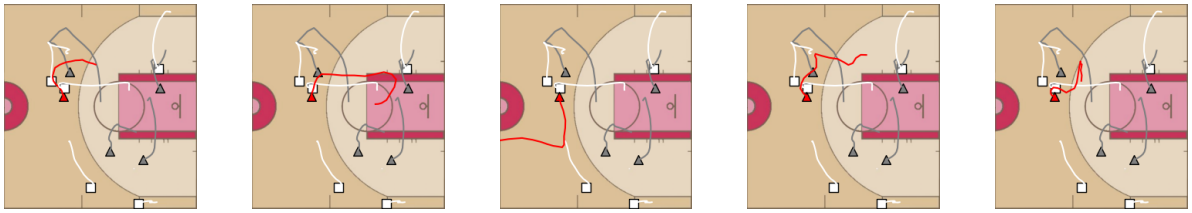
(a) Training sample. (b) `baller2vec`. (c) `baller2vec++`.

Figure 5.6: When trained on a dataset of perfectly coordinated agent trajectories (a), the trajectories generated by `baller2vec` are completely *uncoordinated* (b) while the trajectories generated by `baller2vec++` are perfectly coordinated (c). Animated versions can be found in the code repository.

the trajectories generated by `baller2vec++` are often more realistic than those generated by `baller2vec`. As was observed in Chapter 4, the trajectory forecast distributions for `baller2vec` become much more certain after observing a portion of the sequence (Figure 5.8), which suggests `baller2vec` may be inferring some of the shared hidden variables (e.g., plays) influencing the players. If that hypothesis was true, the performance gap between `baller2vec++` and `baller2vec` should be largest at the beginning of the sequence (before any shared hidden variables can be inferred by `baller2vec`). Indeed, the average NLL for `baller2vec++` in the *first time step* of each test set sequence (1.567) is 16.1% better than the average NLL for `baller2vec` (1.869), while the average NLL for `baller2vec++` in the *last time step* of each test set sequence (0.420) is only 9.7% better than the average NLL for `baller2vec` (0.465) (see Figure 5.8).

### 5.5.2 `baller2vec++` makes better forecasts when conditioned on concurrent trajectory information from other agents

Implicit in much of our discussion has been the intuition that providing a model with additional (relevant) information will improve its performance. To empirically test this conjecture, we compared the performance of `baller2vec++` when predicting the trajectory



(a) Ground truth. (b) `baller2vec`. (c) `baller2vec`. (d) `baller2vec++`. (e) `baller2vec++`.

Figure 5.7: While `baller2vec` occasionally generates realistic trajectories for the red defender (b), it also makes egregious errors (c). In contrast, the trajectories generated by `baller2vec++` often seem plausible (d and e). For both the `baller2vec` and `baller2vec++` generated trajectories, the ground truth trajectories (a) for all of the non-red players were used as input at each time step. The red player was placed *last* in the player order when generating his trajectory with `baller2vec++`. Animated versions can be found in the code repository.

of a specific basketball player placed in the *first position* of the player order (i.e., when $k = 1$) vs. predicting the trajectory for that *same player* placed in the *last position* (i.e., when $k = 10$). Specifically, for each player in each test sequence, we calculated the NLL of the player's trajectory in the first time step[2] with the player in the *first position* of the player order. Next, we moved the player to the *last position* of the player order, and then randomly shuffled the remaining nine players 10 times, calculating the NLL for the player in the last position each time. Finally, we calculated the average percent change in the last position NLLs relative to their corresponding first position NLLs. On average, moving a player from the first to the last position improved the NLL for the player's trajectory by 14.6%.

### 5.5.3 The joint probability assigned to a sequence by `baller2vec++` is approximately permutation invariant with respect to the order of the agents

To determine whether or not `baller2vec++` respects the fact that any ordering of a chain rule decomposition of a joint probability produces the same value, we measured how much the average NLL for each test sequence in the basketball dataset varied when the order of the agents changed. Specifically, for each test set sequence, we shuffled the order

[2]Because, as previously discussed, the benefits of `baller2vec++` were most pronounced in the first time step.

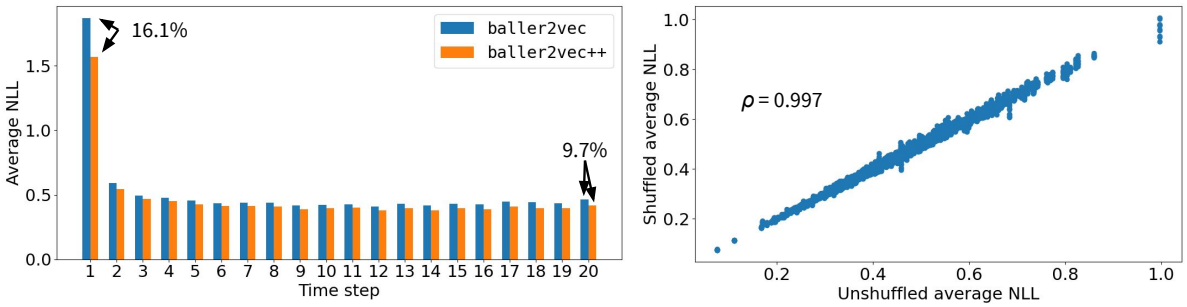



Figure 5.8: **Left**: when modeling the trajectories of professional basketball players, the performance gap between `baller2vec++` and `baller2vec` is largest at the beginning of the sequence, before shared unobserved variables can be inferred by `baller2vec`. Each bar indicates a model's average NLL over the entire test set for that particular time step. For full sequences, `baller2vec++` outperforms `baller2vec` by 8.9%. **Right**: the joint probability assigned to a sequence by `baller2vec++` is approximately permutation invariant with respect to the order of the agents. For each point, its $x$ value indicates `baller2vec++`'s average NLL for a test set sequence using the *original* order of the agents in the sequence, while its $y$ value indicates `baller2vec++`'s average NLL *for the same sequence* with the order of the agents *shuffled*. The shuffled average NLLs are highly correlated with their corresponding unshuffled average NLLs.

of the agents 10 times. Then, for each permuted sequence, we calculated the percent error[3] in the average NLL relative to the original, *unshuffled* sequence. Across all test sequences, the average percent error was only $\pm 1.5\%$. Further, as can be seen in Figure 5.8, the shuffled average NLLs are highly correlated with their corresponding unshuffled average NLLs (Pearson correlation coefficient $= 0.997$), i.e., the joint probability assigned to a sequence by `baller2vec++` is approximately permutation invariant with respect to the order of the agents.

## 5.6 Conclusion and Future Work

In this chapter, we have shown how the commonly used independence assumption of many multi-agent spatiotemporal models can severely limit their ability to learn to emulate coordinated agents. By relaxing this independence assumption in `baller2vec`, `baller2vec++` was able to more accurately model the trajectories of professional basketball

[3]See: `https://en.wikipedia.org/wiki/Approximation_error#Formal_Definition`.

players. Models for other multi-agent spatiotemporal environments, such as pedestrian traffic (see [100] for a survey) and vehicle traffic (e.g., [28, 21, 144, 20]), may also benefit from the look-ahead approach used by `baller2vec++`. Additionally, the interleaved input design of `baller2vec++` could be useful when modeling other systems involving many entities interacting through time, such as social media platforms (e.g., [68, 99]). However, confronting the quadratic complexity of the Transformer attention mechanism as the number of entities grows large in these datasets is an open problem, but recent work in sparse Transformers (e.g., [23, 139, 13, 64, 54, 14]) shows encouraging progress.

# Chapter 6

# The DEformer: An Order-Agnostic Distribution Estimating Transformer

**This chapter was published as "The DEformer: An Order-Agnostic Distribution Estimating Transformer" (Alcorn and Nguyen, 2021) at the 2021 ICML Workshop on Invertible Neural Networks, Normalizing Flows, and Explicit Likelihood Models.**

## 6.1 Introduction

For tasks such as: (a) efficiently imputing arbitrary missing values from an input or (b) preemptive anomaly detection in systems where input features can arrive asynchronously in an arbitrary order (e.g., internet of things applications [4]), order-agnostic autoregressive distribution (density) estimation (OADE) is necessary. However, because there are $D!$ factorizations of the joint probability for a $D$-dimensional input, order-agnosticism adds considerable complexity to the distribution estimation task. As a result, many likelihood-based generative models either: (1) assume a single, fixed order for the input features (e.g., NADE [70], PixelRNN [87], and TraDE [36]), (2) only use a small subset of the possible feature orderings in practice (e.g., MADE [39], IAF [63], MAF [91], and LMConv [57]), or (3) are not autoregressive (e.g., some flows [29, 30, 62, 90]).

In contrast to the previously mentioned approaches, DeepNADE [126, 125] is notable in that it performs full OADE. Specifically, DeepNADE consists of a standard multilayer perceptron (MLP) that takes as input the concatenation of a $D$-dimensional binary mask $\mathbf{m}$ and the masked version of the sample $\mathbf{x}$, $\hat{\mathbf{x}} = \mathbf{m} \odot \mathbf{x}$, i.e., the input $[\hat{\mathbf{x}}, \mathbf{m}]$ is a vector of size

$2D$. The feature identities (e.g., pixel locations) are thus encoded by their positions in the input feature vectors. However, this input design precludes the use of neural architectures that are designed for sequential data (e.g., recurrent neural networks and Transformers [127])—models that are a natural fit for autoregressive problems.

Taking inspiration from the `baller2vec++` model we described in Chapter 5, in this chapter, we propose an alternative approach for encoding feature identities, where each feature's identity is included *alongside* its value in the input. Using this input design, we train an otherwise ordinary Transformer (which we refer to as "the DEformer", i.e., the distribution estimating Transformer) to perform OADE on the binarized-MNIST [103] and POWER [130] datasets. We find that:

1. The DEformer—while being entirely order agnostic and autoregressive—is competitive with fixed-order distribution estimating algorithms when modeling binarized-MNIST and surpasses recent flow-based architectures when modeling the tabular POWER dataset.

2. The DEformer can effortlessly fill in pixels of binarized-MNIST images that are missing in a variety of patterns.

3. The DEformer can easily distinguish between binarized-MNIST and non-binarized-MNIST images.

## 6.2 Architecture

Here, we describe our order-agnostic distribution estimating Transformer, the DEformer (Figure 6.1). The goal in OADE is to model the joint distribution of a $D$-dimensional vector $\mathbf{x}$ by exploiting the chain rule of probability, i.e.:

$$p(\mathbf{x}) = \Pi_{d=1}^{D} p(x_{o_d} | \mathbf{x}_{o_{<d}})$$

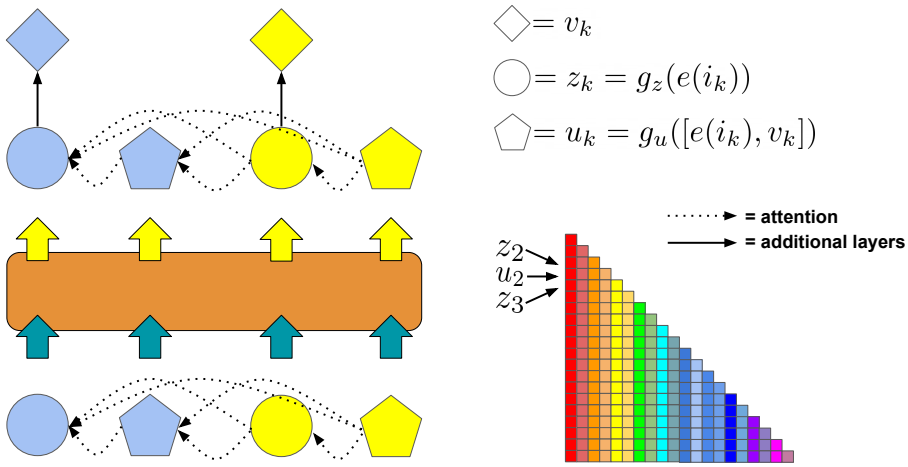



Figure 6.1: By including each feature's identity *alongside* its value in the input, sequential models can be used to perform order-agnostic autoregressive distribution estimation. The DEformer is a Transformer that uses an interleaved input design (partially depicted here with the self-attention mask) for this task. The two sets of interleaved feature vectors consist of identity feature vectors ($z_k$) and identity/value feature vectors ($u_k$), and $g_z$ and $g_u$ are their respective multilayer perceptrons. For the binarized-MNIST dataset, each feature identity $i_k$ is a tuple ($r_k$, $c_k$) where $r_k$ and $c_k$ are the row and column for the pixel indexed by $k$ in the permuted sequence, respectively, and $v_k$ is the value of the pixel (which is zero or one for binary images). For tabular data, each $i_k$ corresponds to a column, and $v_k$ is the value of the indexed column in the row. Lastly, $e$ is an identity encoding function, which is simply the identify function in the case of binarized-MNIST, and is an embedding layer for tabular data.

where, as in [126], $o$ is a $D$-tuple representing a permutation of the elements in $\mathbf{x}$, so $x_{o_d}$ indicates the element of $\mathbf{x}$ indexed by the $d$-th element of $o$, and $\mathbf{x}_{o_{<d}}$ means the elements in $\mathbf{x}$ indexed by the first $d-1$ elements of $o$. We assume each discrete feature can take on one of $C$ labels (which is the case for image datasets), but, in theory, each feature could have a different number of possible labels.

Rather than encoding each feature's identity by confining it to a specific position in the input, here, we propose including the feature's identity as an additional input variable *alongside* its value. Specifically, the input to the DEformer consists of two parallel sequences: one containing only feature identities, and another containing identity/value pairs:

1. $i_1, i_2, ..., i_n$

2. $(i_1, v_1), (i_2, v_2), ..., (i_n, v_n)$

where $i_k$ is the identity of the $k$-th feature in the permuted sequence and $v_k$ is the value of the $k$-th feature. In the case of binarized-MNIST, each $i_k$ is a tuple $(r_k, c_k)$ indicating the row and column of the pixel, respectively, and $v_k$ is the value of the pixel (i.e., zero or one). For tabular data, each $i_k$ indexes a column, and $v_k$ is the value of the indexed column in the row.

The identity inputs are mapped to identity feature vectors using an MLP, i.e., $z_k = g_z(e(i_k))$ where $z_k$ is the identity feature vector, $g_z$ is the identity MLP, and $e$ is an identity encoding function. In the case of binarized-MNIST, $e$ is simply the identity function, i.e., $e(i_k) = [r_k, c_k]$, while for tabular data, $e$ is an embedding layer. The identity/value pairs are similarly mapped to identity/value feature vectors using a separate MLP, i.e., $u_k = g_u([e(i_k), v_k])$ where $u_k$ is the identity/value feature vector and $g_u$ is the identity/value MLP. These two sets of feature vectors are interleaved with one another (i.e., $u_k$ always immediately follows $z_k$ in the input) to form a $2D \times F$ matrix where $F$ is the dimension of the outputs for the MLPs.

This matrix is passed into the Transformer along with a lower triangular self-attention mask, which encodes the following dependencies (see Figure 6.1):

1. When processing $z_{k_2}$, the DEformer is allowed to "look" at: (i) any $z_{k_1}$ where $k_1 \leq k_2$ and (ii) any $u_{k_1}$ where $k_1 < k_2$.

2. When processing $u_{k_2}$, the DEformer is allowed to "look" at: (i) any $z_{k_1}$ where $k_1 \leq k_2$ and (ii) any $u_{k_1}$ where $k_1 \leq k_2$.

Like `baller2vec` and `baller2vec++`, we do not use positional encoding [127] because [56] observed that positional encoding is not only unnecessary, but detrimental for Transformers that use a causal attention mask.

Each processed $z_k$ feature vector is then passed through a final linear layer. When modeling discrete features, the final linear layer is followed by a softmax, which gives a

probability distribution over the labels for the feature indexed by $k$. The loss for each sample is thus:

$$\mathcal{L} = \sum_{k=1}^{K} -\ln(f(Z)_{2k-1}[v_k]) \tag{6.1}$$

where $f(Z)_{2k-1}[v_k]$ is the probability assigned to the label $v_k$ (where $v_k$ is an integer from one to $C$) by $f$, i.e., Equation (6.1) is the NLL of the data according to the model. For continuous features, the output of the final linear layer defines a mixture of Gaussians, so the loss for each sample is:

$$\mathcal{L} = \sum_{k=1}^{K} -\ln(\pi_k \cdot c_k)$$

where $\pi_k = \text{softmax}(f(Z)_{2k-1,1:J})$ is a vector containing the $J$ mixture proportions for feature $k$, and $c_k$ is a vector containing the mixture densities such that:

$$c_k[j] = \frac{1}{\sigma_{k,j}\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{v_k - \mu_{k,j}}{\sigma_{k,j}})^2}$$

where $\sigma_k = f(Z)_{2k-1,J+1:2J}$ and $\mu_k = f(Z)_{2k-1,2J+1:3J}$.

Because any ordering of a chain rule decomposition of a joint probability produces the same value, e.g.:

$$p(x_1)p(x_2|x_1)p(x_3|x_1x_2) = p(x_3)p(x_2|x_3)p(x_1|x_3x_2)$$

like `baller2vec++` and [126, 135], we shuffle the order of the features in each training sample to encourage the DEformer to learn a joint distribution of the dataset that is approximately permutation invariant with respect to the ordering of the features.

## 6.3 Experiments

To test the utility of the DEformer for OADE, we trained a nearly identical architecture to the `baller2vec++` model described in Chapter 5 on the binarized-MNIST [103] and

POWER [130] datasets. The binarized-MNIST dataset consists of 70,000 $28 \times 28$ pixel binary images (i.e., the pixel values are either black or white) of digits (i.e., 0-9) where each digit is represented by the same number of images. We used the standard 60,000/10,000 split for training/testing images, respectively, and used 1,200 of the 60,000 training images (i.e., 2%) for validation. The POWER dataset consists of 2,049,280 power measurements from a single household in a tabular format, where each sample consists of six real values. We used the same preprocessing steps and training/validation/test split described in [91].

The size of the output for the final linear layer was one for the binarized-MNIST dataset and $3 \times 150 = 450$ for the POWER dataset (as in [36]), but all remaining hyperparameters and training details were nearly identical to `baller2vec++`, which itself closely follows the original Transformer [127]. Specifically, the Transformer settings were: $d_{\text{model}} = 512$ (the dimension of the input and output of each Transformer layer), eight attention heads, $d_{\text{ff}} = 2048$ (the dimension of the inner feedforward layers), six layers, dropout probabilities of 0.0 and 0.2 for the binarized-MNIST and POWER datasets, respectively, and no positional encoding. Each MLP (i.e., $g_z$, $g_u$, and $g_r$) had 128, 256, and 512 nodes in its three layers, respectively, and a ReLU nonlinearity following each of the first two layers. Lastly, the identity embedding layer for the POWER dataset mapped column indices to 20-dimensional vectors.

We used the Adam optimizer [61] with an initial learning rate of $10^{-6}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-9}$ to update the model parameters, of which there were $\sim$19 million. The learning rate was reduced to $10^{-7}$ after 5/20 epochs of the validation loss not improving for the binarized-MNIST/POWER datasets, respectively, and we used batch sizes of 1/128 for the binarized-MNIST/POWER datasets, respectively. Models were implemented in PyTorch and trained on a single NVIDIA GTX 1080 Ti GPU for $\sim$50/700 epochs (2.5/6 days) for the binarized-MNIST/POWER datasets, respectively, and the validation set was used for early stopping.

Table 6.1: The average NLL on the binarized-MNIST test set for different models. Despite being entirely order agnostic ("OA"), the DEformer is competitive with PixelRNN and TraDE, which use a single fixed order ("FO"). For MADE, the model was trained on 32 different orders [125]. The average NLLs for both DeepNADE and the DEformer are calculated over 10 random orderings.

| Model | NLL |
|---|---|
| DeepNADE (OA) | 89.17 |
| MADE (32) | 86.64 |
| PixelRNN (FO) | 79.20 |
| TraDE (FO) | 78.92 |
| DEformer (OA) | 80.49 |

Table 6.2: The average NLL on the POWER test set for different models. The DEformer surpasses the performance of recent flow-based architectures while still retaining order-agnostic and autoregressive properties. The average NLL for the DEformer is calculated over 10 random orderings.

| Model | NLL |
|---|---|
| RealNVP | -0.17 |
| MAF | -0.3 |
| NAF | -0.62 |
| NSF | -0.66 |
| TraDE | -0.73 |
| DEformer | -0.68 |

## 6.4 Results

The DEformer achieved an average NLL (taken over 10 orders) of 80.49 on the binarized-MNIST test set. This is a vast improvement over DeepNADE [126] and is competitive with fixed-order distribution estimation algorithms like PixelRNN [87] and TraDE [36] (see Table 6.1). On the POWER dataset, the DEformer achieved an average NLL of -0.68, which surpasses the performance of recent flow-based architectures like NAF [55] and NSF [33]. We suspect the DEformer's performance could be improved with a careful hyperparameter search.

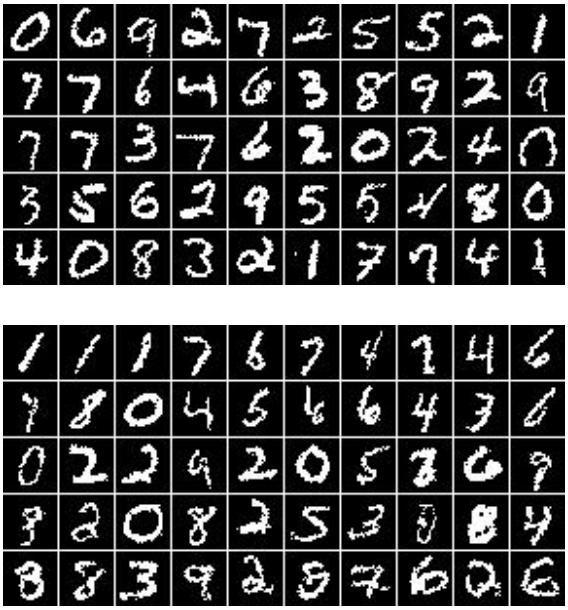


Figure 6.2: **Top**: A sample of 50 images from the test set of binarized-MNIST organized by their average NLLs according to the DEformer (starting with the lowest average NLL, 42.1, at the top left, and ending with the highest average NLL, 119.6, at the bottom right). **Bottom**: 50 generated images organized by their average NLLs according to the DEformer (starting with the lowest average NLL, 42.0, at the top left, and ending with the highest average NLL, 131.1, at the bottom right). The average NLLs for both sets of images are calculated over 10 random orderings.

Following [126], Figure 6.2 shows 50 samples from the test set of binarized-MNIST sorted by their average NLLs (taken over 10 orders) according to the DEformer, along with 50 samples generated by the DEformer, also sorted by their average NLLs. Also following [126], Figure 6.3 shows examples of images with 100 pixels missing in a variety of patterns, which were then "filled in" by the DEformer when conditioned on the remaining 684 pixels. Like DeepNADE, this task is trivial for the DEformer because the pixels can be arranged such that the conditioning pixels are at the beginning of the sequence. Lastly, as can be seen in Figure 6.4, the DEformer can easily distinguish between in-distribution and out-of-distribution (i.e., binarized-notMNIST images [17]) samples.

Figure 6.3: Because the DEformer is order-agnostic, it can easily "fill in" images where pixels are missing in a variety of patterns by placing the missing pixels at the end of the input sequence. Here, each row corresponds to a different ground truth image from the test set (depicted in the first column). The remaining pairs of columns show 100 removed pixels (red) from the ground truth image and the corresponding filled in image.

## 6.5 Related Work

### 6.5.1 Interleaved input Transformers

The DEformer is directly inspired by the `baller2vec++` model described in Chapter 5. Our key contribution is recognizing that this interleaved architecture design can be applied to OADE. The DEformer is architecturally similar to the independently developed XLNet language model [135]. Compared to XLNet, the DEformer:

1. encodes feature identity by including it as an input to the network (instead of using positional embeddings) and

2. uses a full lower triangular attention mask to attend to *both* identity feature vectors and identity/value feature vectors that occur earlier in the shuffled input (instead of only attending to the "content stream").

Notably, XLNet was trained to only predict the final six tokens of the shuffled sentence because the authors observed "slow convergence in preliminary experiments". The DEformer was capable of modeling the values for all 784 pixels in our binarized-MNIST experiments.

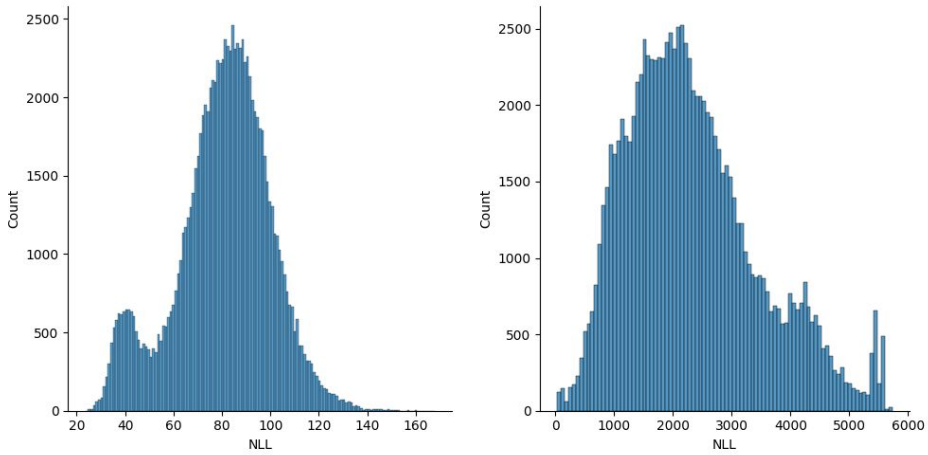



Figure 6.4: The distribution of the DEformer average NLLs for the binarized-MNIST test set (left) and a subset of 10,000 images from the binarized-notMNIST dataset (right) diverge considerably (notice the difference in scales of the $x$-axes), i.e., the DEformer consistently assigns lower probabilities to out-of-distribution samples.

### 6.5.2 DeepNADE

One important way DeepNADE [126, 125] and the DEformer differ is in the size of the outputs for their final classification layers, which are $DC$ and $C$, respectively. While this difference is not particularly important for a relatively simple dataset like binarized-MNIST, for more complex datasets like CIFAR-10 [66], these contrasting designs produce dramatically different parameter counts. Specifically, the size of the output for a CIFAR-10 DeepNADE model would be $32 \times 32 \times 3 \times 256 = 786{,}432$ (because each pixel has three channels, and each channel can take on one of 256 different integer values). Therefore, if the input dimension to the final layer was 500 (as it was in the DeepNADE model for binarized-MNIST), the final layer alone would have $500 \times 786{,}432 + 786{,}432 = 394{,}002{,}432$ parameters. While the number of outputs can be reduced for image datasets by using a discretized logistic mixture likelihood [105], this strategy restricts the complexity of the model, and the discretized logistic mixture likelihood is not applicable to datasets where the labels do not have a clear underlying order.

On the other hand, due to the attention mechanism, the DEformer suffers from the same quadratic complexity problem known to plague Transformers. However, recent work

in sparse Transformers (e.g., [23, 139, 13, 64]) may allow the DEformer to scale to larger inputs.

When training DeepNADE, a mask is randomly generated for each sample by: (1) randomly selecting an integer $c \in \{0, \ldots, D-1\}$ to serve as the number of conditioning variables and (2) randomly assigning a value of one to $c$ locations in the mask and assigning a value of zero to the remaining locations. The loss for each sample is then:

$$\mathcal{L} = \frac{D}{D-c} \sum_{d=1}^{D} (1 - m_d)(-\ln(f(\hat{\mathbf{x}}, \mathbf{m})_{(d-1)C+l_d}))$$

where $l_d$ is the label for the $d$-th feature of $\mathbf{x}$, and $\frac{D}{D-c}$ is a scaling factor ensuring the loss for each sample is an unbiased estimator (which is necessary because the error signal is only computed for $D-c$ features of the sample due to the $1-m_d$ term). In contrast, for the DEformer, there is always an error signal for all of the features of each sample. While MADE [39] also produces an error signal for all of the features of each sample, the authors observed that sampling many different weight masks led to the model underfitting, so it is unclear how well MADE can perform fully OADE.

### 6.5.3 Spatial inputs as feature identities

A number of neural network architectures operate directly on spatial coordinates, which can be interpreted as feature identities in their various contexts (e.g., images [47], point clouds [45], and 3D scenes [110]). Additionally, [77] observed that adding channels to feature maps that contain the spatial coordinates of the pixels greatly improved the performance of convolutional neural networks on certain spatial reasoning tasks. However, none of these models are performing autoregressive distribution estimation, nor do they employ the interleaved input design of the DEformer.

## 6.6 Conclusion

In this chapter, we described an alternative approach to OADE where the identities of features are included *alongside* their values in the input. We believe the performance of the DEformer on the binarized-MNIST and POWER datasets is encouraging, and we are excited to see how this architecture can be applied in different contexts.

# Bibliography


[1] Moderngl — moderngl 5.4.1 documentation. https://moderngl.readthedocs.io/en/stable/index.html. (Accessed on 11/14/2018).

[2] The self-driving uber that killed a pedestrian didn't brake. here's why. https://slate.com/technology/2018/05/uber-car-in-fatal-arizona-crash-perceived-pedestrian-1-3-seconds-before-impact.html. (Accessed on 07/13/2018).

[3] Tesla car on autopilot crashes, killing driver, united states news & top stories - the straits times. https://www.straitstimes.com/world/united-states/tesla-car-on-autopilot-crashes-killing-driver. (Accessed on 06/14/2018).

[4] S. Ahmad, A. Lavin, S. Purdy, and Z. Agha. Unsupervised real-time anomaly detection for streaming data. *Neurocomputing*, 262:134–147, 2017.

[5] M. A. Alcorn. (batter|pitcher)2vec: Statistic-free talent modeling with neural player embeddings. In *MIT Sloan Sports Analytics Conference*, 2018.

[6] M. A. Alcorn and A. Nguyen. baller2vec: A multi-entity transformer for multi-agent spatiotemporal modeling. *arXiv preprint arXiv:2102.03291*, 2021.

[7] H. A. Alhaija, S. K. Mustikovela, A. Geiger, and C. Rother. Geometric image synthesis. *arXiv preprint arXiv:1809.04696*, 2018.

[8] A. Athalye, L. Engstrom, A. Ilyas, and K. Kwok. Synthesizing robust adversarial examples. In *2018 Proceedings of the 35th International Conference on Machine Learning (ICML)*, pages 284–293, 2018.

[9] A. Azulay and Y. Weiss. Why do deep convolutional networks generalize so poorly to small image transformations? *arXiv preprint arXiv:1805.12177*, 2018.

[10] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. In *Advances in Neural Information Processing Systems*, 2016.

[11] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*, 2015.

[12] Basketball-Reference.com. 2015-16 nba player stats: Totals, February 2021.

[13] I. Beltagy, M. E. Peters, and A. Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[14] G. Bertasius, H. Wang, and L. Torresani. Is space-time attention all you need for video understanding? *arXiv preprint arXiv:2102.05095*, 2021.

[15] A. Borji, S. Izadi, and L. Itti. ilab-20m: A large-scale controlled object dataset to investigate deep learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2221–2230, 2016.

[16] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[17] Y. Bulatov, Sep 2011.

[18] N. Carlini and D. Wagner. Towards Evaluating the Robustness of Neural Networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[19] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

[20] R. Chandra, U. Bhattacharya, A. Bera, and D. Manocha. Traphic: Trajectory prediction in dense and heterogeneous traffic using weighted interactions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[21] M.-F. Chang, J. Lambert, P. Sangkloy, J. Singh, S. Bak, A. Hartnett, D. Wang, P. Carr, S. Lucey, D. Ramanan, and J. Hays. Argoverse: 3d tracking and forecasting with rich maps. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[22] C. Chen, A. Seff, A. Kornhauser, and J. Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[23] R. Child, S. Gray, A. Radford, and I. Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[24] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, Oct. 2014. Association for Computational Linguistics.

[25] B. B. Choi and C. Lawrence. Inverse Kinematics Problem in Robotics Using Neural Networks. *NASA Technical Memorandum*, 105869:1–23, 1992.

[26] J. Chung, K. Kastner, L. Dinh, K. Goel, A. Courville, and Y. Bengio. A recurrent latent variable model for sequential data. In *Advances in Neural Information Processing Systems*, 2015.

[27] Y. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *Neural Information Processing Systems*, pages 2933–2941, 2014.

[28] N. Deo and M. M. Trivedi. Convolutional social pooling for vehicle trajectory prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 1468–1476, 2018.

[29] L. Dinh, D. Krueger, and Y. Bengio. NICE: non-linear independent components estimation. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Workshop Track Proceedings*, 2015.

[30] L. Dinh, J. Sohl-Dickstein, and S. Bengio. Density estimation using real nvp. 2017.

[31] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021.

[32] V. Dumoulin and F. Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[33] C. Durkan, A. Bekasov, I. Murray, and G. Papamakarios. Neural spline flows. *Advances in Neural Information Processing Systems*, 32:7511–7522, 2019.

[34] L. Engstrom, B. Tran, D. Tsipras, L. Schmidt, and A. Madry. A rotation and a translation suffice: Fooling CNNs with simple transformations, 2019.

[35] I. Evtimov, K. Eykholt, E. Fernandes, T. Kohno, B. Li, A. Prakash, A. Rahmati, and D. Song. Robust physical-world attacks on machine learning models. *arXiv preprint arXiv:1707.08945*, 2017.

[36] R. Fakoor, P. Chaudhari, J. Mueller, and A. J. Smola. Trade: Transformers for density estimation. *Invertible Neural Networks, Normalizing Flows, and Explicit Likelihood Models Workshop*, 2020.

[37] P. Felsen, P. Lucey, and S. Ganguly. Where will they go? predicting fine-grained adversarial multi-agent motion using conditional variational autoencoders. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 732–747, 2018.

[38] D. Gandhi, L. Pinto, and A. Gupta. Learning to fly by crashing. In *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, pages 3948–3955. IEEE, 2017.

[39] M. Germain, K. Gregor, I. Murray, and H. Larochelle. Made: Masked autoencoder for distribution estimation. In F. Bach and D. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 881–889, Lille, France, 07–09 Jul 2015. PMLR.

[40] F. Giuliari, I. Hasan, M. Cristani, and F. Galasso. Transformer networks for trajectory forecasting. In *International Conference on Pattern Recognition*, 2020.

[41] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[42] I. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations*, 2015.

[43] A. Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[44] A. Graves, G. Wayne, and I. Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[45] Y. Guo, H. Wang, Q. Hu, H. Liu, L. Liu, and M. Bennamoun. Deep learning for 3d point clouds: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–1, 2020.

[46] A. Gupta, J. Johnson, L. Fei-Fei, S. Savarese, and A. Alahi. Social gan: Socially acceptable trajectories with generative adversarial networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2255–2264, 2018.

[47] D. Ha. Generating large images from latent vectors. *blog.otoro.net*, 2016.

[48] D. Ha. A Visual Guide to Evolution Strategies, 2017.

[49] D. Ha and J. Schmidhuber. Recurrent World Models Facilitate Policy Evolution. *Neural Information Processing Systems*, pages 2450–2462, 2018.

[50] N. Hansen. Covariance Matrix Adaptation Evolution Strategy (CMA-ES). Technical report.

[51] N. Hansen. The CMA Evolution Strategy: A Tutorial. 4 2016.

[52] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[53] D. Hendrycks and K. Gimpel. A baseline for detecting misclassified and out-of-distribution examples in neural networks. In *Proceedings of International Conference on Learning Representations*, 2017.

[54] J. Ho, N. Kalchbrenner, D. Weissenborn, and T. Salimans. Axial attention in multidimensional transformers. *arXiv preprint arXiv:1912.12180*, 2019.

[55] C.-W. Huang, D. Krueger, A. Lacoste, and A. Courville. Neural autoregressive flows. In *International Conference on Machine Learning*, pages 2078–2087. PMLR, 2018.

[56] K. Irie, A. Zeyer, R. Schlüter, and H. Ney. Language modeling with deep transformers. In *Proc. Interspeech 2019*, pages 3905–3909, 2019.

[57] A. Jain, P. Abbeel, and D. Pathak. Locally masked convolution for autoregressive models. In J. Peters and D. Sontag, editors, *Proceedings of the 36th Conference on Uncertainty in Artificial Intelligence (UAI)*, volume 124 of *Proceedings of Machine Learning Research*, pages 1358–1367. PMLR, 03–06 Aug 2020.

[58] D. Karmon, D. Zoran, and Y. Goldberg. Lavan: Localized and visible adversarial noise. *arXiv preprint arXiv:1801.02608*, 2018.

[59] H. Kato, Y. Ushiku, and T. Harada. Neural 3D Mesh Renderer. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[60] K. Kim, M. Grundmann, A. Shamir, I. Matthews, J. Hodgins, and I. Essa. Motion fields to predict play evolution in dynamic sport scenes. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 840–847. IEEE, 2010.

[61] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

[62] D. P. Kingma and P. Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[63] D. P. Kingma, T. Salimans, R. Jozefowicz, X. Chen, I. Sutskever, and M. Welling. Improved variational inference with inverse autoregressive flow. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[64] N. Kitaev, L. Kaiser, and A. Levskaya. Reformer: The efficient transformer. In *International Conference on Learning Representations*, 2020.

[65] V. Kosaraju, A. Sadeghian, R. Martín-Martín, I. Reid, H. Rezatofighi, and S. Savarese. Social-bigat: Multimodal trajectory forecasting using bicycle-gan and graph attention networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[66] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[67] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NIPS 2012)*, pages 1097–1105, 2012.

[68] S. Kumar, X. Zhang, and J. Leskovec. Predicting dynamic embedding trajectory in temporal interaction networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1269–1278, 2019.

[69] F. Lambert. Understanding the fatal tesla accident on autopilot and the nhtsa probe. *Electrek, July*, 2016.

[70] H. Larochelle and I. Murray. The neural autoregressive distribution estimator. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 29–37. JMLR Workshop and Conference Proceedings, 2011.

[71] H. M. Le, P. Carr, Y. Yue, and P. Lucey. Data-driven ghosting using deep imitation learning. In *MIT Sloan Sports Analytics Conference*, 2017.

[72] H. M. Le, Y. Yue, P. Carr, and P. Lucey. Coordinated multi-agent imitation learning. In *International Conference on Machine Learning*, volume 70, pages 1995–2003, 2017.

[73] Q. Le and T. Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196. PMLR, 2014.

[74] V. Lempitsky, A. Vedaldi, and D. Ulyanov. Deep image prior. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9446–9454. IEEE, 2018.

[75] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[76] H.-T. D. Liu, M. Tao, C.-L. Li, D. Nowrouzezahrai, and A. Jacobson. Adversarial Geometry and Lighting using a Differentiable Renderer. *arXiv preprint*, 8 2018.

[77] R. Liu, J. Lehman, P. Molino, F. Petroski Such, E. Frank, A. Sergeev, and J. Yosinski. An intriguing failing of convolutional neural networks and the coordconv solution. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[78] J. Lu, H. Sibai, E. Fabry, and D. Forsyth. NO Need to Worry about Adversarial Examples in Object Detection in Autonomous Vehicles. *arXiv preprint*, 7 2017.

[79] J. Lu, H. Sibai, E. Fabry, and D. A. Forsyth. Standard detectors aren't (currently) fooled by physical adversarial stop signs. *CoRR*, abs/1710.03337, 2017.

[80] Y. Luo, X. Boix, G. Roig, T. Poggio, and Q. Zhao. Foveation-based Mechanisms Alleviate Adversarial Examples. *arXiv preprint*, 11 2015.

[81] S. Marschner and P. Shirley. *Fundamentals of computer graphics*. CRC Press, 2015.

[82] L. McInnes, J. Healy, and J. Melville. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*, 2018.

[83] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[84] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, 2013.

[85] A. Nguyen, J. Clune, Y. Bengio, A. Dosovitskiy, and J. Yosinski. Plug & play generative networks: Conditional iterative generation of images in latent space. In *CVPR*, volume 2, page 7, 2017.

[86] A. Nguyen, J. Yosinski, and J. Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 427–436, 2015.

[87] A. V. Oord, N. Kalchbrenner, and K. Kavukcuoglu. Pixel recurrent neural networks. In M. F. Balcan and K. Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1747–1756, New York, New York, USA, 20–22 Jun 2016. PMLR.

[88] M. Osadchy, M. L. Miller, and Y. LeCun. Synergistic Face Detection and Pose Estimation with Energy-Based Models. In *Advances in Neural Information Processing Systems*, pages 1017–1024, 2005.

[89] M. Ozuysal, V. Lepetit, and P. Fua. Pose estimation for category specific multiview object localization. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 778–785. IEEE, 2009.

[90] G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed, and B. Lakshminarayanan. Normalizing flows for probabilistic modeling and inference. *Journal of Machine Learning Research*, 22(57):1–64, 2021.

[91] G. Papamakarios, T. Pavlakou, and I. Murray. Masked autoregressive flow for density estimation. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[92] N. Parmar, A. Vaswani, J. Uszkoreit, L. Kaiser, N. Shazeer, A. Ku, and D. Tran. Image transformer. In *International Conference on Machine Learning*, pages 4055–4064. PMLR, 2018.

[93] N. Pinto, J. J. DiCarlo, and D. D. Cox. How far can you get with a modern face recognition test set using only simple features? In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2591–2598. IEEE, 2009.

[94] PyTorch. torchvision.models — pytorch master documentation. `https://pytorch.org/docs/stable/torchvision/models.html`. (Accessed on 11/14/2018).

[95] M. Qi, J. Qin, Y. Wu, and Y. Yang. Imitative non-autoregressive modeling for trajectory forecasting and imputation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12736–12745, 2020.

[96] W. Qiu, F. Zhong, Y. Zhang, S. Qiao, Z. Xiao, T. S. Kim, Y. Wang, and A. Yuille. Unrealcv: Virtual worlds for computer vision. *ACM Multimedia Open Source Software Competition*, 2017.

[97] J. Redmon and A. Farhadi. YOLOv3: An Incremental Improvement. 2018.

[98] A. Rosenfeld, R. Zemel, and J. K. Tsotsos. The elephant in the room. *arXiv preprint arXiv:1808.03305*, 2018.

[99] E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein. Temporal graph networks for deep learning on dynamic graphs. In *ICML 2020 Workshop on Graph Representation Learning*, 2020.

[100] A. Rudenko, L. Palmieri, M. Herman, K. M. Kitani, D. M. Gavrila, and K. O. Arras. Human motion trajectory prediction: A survey. *The International Journal of Robotics Research*, 39(8):895–935, 2020.

[101] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[102] A. Sadeghian, V. Kosaraju, A. Sadeghian, N. Hirose, H. Rezatofighi, and S. Savarese. Sophie: An attentive gan for predicting paths compliant to social and physical constraints. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1349–1358, 2019.

[103] R. Salakhutdinov and I. Murray. On the quantitative analysis of deep belief networks. In *Proceedings of the 25th international conference on Machine learning*, pages 872–879, 2008.

[104] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever. Evolution Strategies as a Scalable Alternative to Reinforcement Learning. 3 2017.

[105] T. Salimans, A. Karpathy, X. Chen, and D. P. Kingma. Pixelcnn++: A pixelcnn implementation with discretized logistic mixture likelihood and other modifications. In *ICLR*, 2017.

[106] C. Sampedro, A. Rodriguez-Ramos, H. Bavle, A. Carrio, P. de la Puente, and P. Campoy. A fully-autonomous aerial robot for search and rescue applications in indoor environments using learning-based techniques. *Journal of Intelligent & Robotic Systems*, pages 1–27, 2018.

[107] R. Sanford, S. Gorji, L. G. Hafemann, B. Pourbabaee, and M. Javan. Group activity detection from trajectory and video data in soccer. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 898–899, 2020.

[108] W. J. Scheirer, A. de Rezende Rocha, A. Sapkota, and T. E. Boult. Toward open set recognition. *IEEE transactions on pattern analysis and machine intelligence*, 35(7):1757–1772, 2013.

[109] A. Shrivastava, T. Pfister, O. Tuzel, J. Susskind, W. Wang, and R. Webb. Learning from simulated and unsupervised images through adversarial training. In *CVPR*, volume 2, page 5, 2017.

[110] V. Sitzmann, M. Zollhöfer, and G. Wetzstein. Scene representation networks: Continuous 3d-structure-aware neural scene representations. In *Advances in Neural Information Processing Systems*, 2019.

[111] R. Socher, C. C.-Y. Lin, A. Y. Ng, and C. D. Manning. Parsing natural scenes and natural language with recursive neural networks. In *ICML*, 2011.

[112] R. Socher, C. D. Manning, and A. Y. Ng. Learning continuous phrase representations and syntactic parsing with recursive neural networks. In *Proceedings of the NIPS-2010 deep learning and unsupervised feature learning workshop*, volume 2010, pages 1–9, 2010.

[113] J. Su, D. V. Vargas, and S. Kouichi. One Pixel Attack for Fooling Deep Neural Networks. *arXiv preprint*, 2017.

[114] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune. Deep neuroevolution: genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[115] M. Sugiyama, N. D. Lawrence, A. Schwaighofer, et al. *Dataset shift in machine learning*. The MIT Press, 2017.

[116] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, 2014.

[117] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[118] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the Inception Architecture for Computer Vision. *Conference on Computer Vision and Pattern Recognition*, 2016.

[119] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the Inception Architecture for Computer Vision. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826, 12 2016.

[120] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations*, 2014.

[121] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *International Conference on Learning Representations*, 12 2014.

[122] G. R. Taylor, A. J. Chosak, and P. C. Brewer. Ovvv: Using virtual worlds to design and evaluate surveillance systems. In *2007 IEEE conference on computer vision and pattern recognition*, pages 1–8. IEEE, 2007.

[123] Y. Tian, K. Pei, S. Jana, and B. Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*, pages 303–314. ACM, 2018.

[124] A. Torralba and A. A. Efros. Unbiased look at dataset bias. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 1521–1528. IEEE, 2011.

[125] B. Uria, M.-A. Côté, K. Gregor, I. Murray, and H. Larochelle. Neural autoregressive distribution estimation. *The Journal of Machine Learning Research*, 17(1):7184–7220, 2016.

[126] B. Uria, I. Murray, and H. Larochelle. A deep and tractable density estimator. In *International Conference on Machine Learning*, pages 467–475. PMLR, 2014.

[127] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[128] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

[129] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018. accepted as poster.

[130] A. Vergara, S. Vembu, T. Ayhan, M. A. Ryan, M. L. Homer, and R. Huerta. Chemical gas sensor drift compensation using classifier ensembles. *Sensors and Actuators B: Chemical*, 166:320–329, 2012.

[131] J. Weston, S. Chopra, and A. Bordes. Memory networks. In *International Conference on Learning Representations*, 2015.

[132] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[133] Y. Xiang, W. Kim, W. Chen, J. Ji, C. Choy, H. Su, R. Mottaghi, L. Guibas, and S. Savarese. Objectnet3d: A large scale database for 3d object recognition. In *European Conference on Computer Vision*, pages 160–176. Springer, 2016.

[134] C. Xiao, J.-Y. Zhu, B. Li, W. He, M. Liu, and D. Song. Spatially transformed adversarial examples. In *International Conference on Learning Representations*, 2018.

[135] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.

[136] R. A. Yeh, A. G. Schwing, J. Huang, and K. Murphy. Diverse generation for multi-agent sports games. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4610–4619, 2019.

[137] C. Yu, X. Ma, J. Ren, H. Zhao, and S. Yi. Spatio-temporal graph transformer networks for pedestrian trajectory prediction. In *Proceedings of the European Conference on Computer Vision (ECCV)*, August 2020.

[138] X. Yuan, P. He, Q. Zhu, and X. Li. Adversarial Examples: Attacks and Defenses for Deep Learning. *arXiv preprint*, 2017.

[139] M. Zaheer, G. Guruganesh, K. A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, and A. Ahmed. Big bird: Transformers for longer sequences. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 17283–17297. Curran Associates, Inc., 2020.

[140] E. Zhan, A. Tseng, Y. Yue, A. Swaminathan, and M. Hausknecht. Learning calibratable policies using programmatic style-consistency. In *International Conference on Machine Learning*, pages 11001–11011. PMLR, 2020.

[141] E. Zhan, S. Zheng, Y. Yue, L. Sha, and P. Lucey. Generating multi-agent trajectories using programmatic weak supervision. In *International Conference on Learning Representations*, 2019.

[142] H. Zhang, I. Goodfellow, D. Metaxas, and A. Odena. Self-attention generative adversarial networks. In *International conference on machine learning*, pages 7354–7363. PMLR, 2019.

[143] R. Zhang. Making convolutional networks shift-invariant again, 2019.

[144] T. Zhao, Y. Xu, M. Monfort, W. Choi, C. Baker, Y. Zhao, Y. Wang, and Y. N. Wu. Multi-agent tensor fusion for contextual trajectory prediction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[145] S. Zheng, Y. Yue, and J. Hobbs. Generating long-term trajectories using deep hierarchical networks. *Advances in Neural Information Processing Systems*, 29:1543–1551, 2016.

[146] Z. Zhu, L. Xie, and A. L. Yuille. Object recognition with and without objects. *arXiv preprint arXiv:1611.06596*, 2016.