A. Experimental Details

A.1. Methods Compared

We benchmark our model against Fastron with Forward Kinematics Kernel (Fastron FK), which is known to be state-of-the-art in learning-based collision detection [3, 10]. We note that Fastron and Fastron FK were already benchmarked against the previous state-of-the-art approaches: SSVM [43], ISVM [13], and GJK [4]. In the extensive experiments presented by the authors, Fastron and Fastron FK were shown to be superior to or on-par with all the aforementioned methods, as it relates to the speed-error trade-off (albeit at $\leq$ 7 DoF, with only a few thousand training samples) [3, 10]. Furthermore, they found that Fastron FK is superior to vanilla Fastron, in terms of both speed and time [10]. Due to these reasons, the theoretical analysis described in Section 4, and the apparent lack of open-source code for other learning-based methods, we only benchmark against Fastron FK.

Hyperparameters: In order to fairly compare the methods, we test multiple hyperparameter settings. We made sure to include the hyperparameter settings included by the authors of Fastron, as those were previously determined to be near-optimal [3]. Then, out of the tested settings, we only consider the best hyperparameter settings for each model: it would be unfair to consider other hyperparameter settings, since every model has hyperparameters that could make it useless. See Tables 1 and 2.

Train-Test Split: For almost all of the experiments, we use $30,000$ training points and $5,000$ testing points. All points are uniformly randomly sampled from the configuration space.

There are, however, two exceptions. The first is when we investigate the impact of training dataset size on model performance – here we vary the number of training samples between 10^2 and 10^5, but still use $5,000$ testing samples. The second is when we investigate the impact of sampling strategy on model performance, as uniform sampling may be problematic in the physical world – here we try rejection sampling (as it relates to points inside obstacles), but continue to use $30,000$ training and $5,000$ testing points.

A.2. Environments

In designing the experimental environments, our goals are as follows:

1. To quantify how DeepCollide performs in the speed vs. error trade-off, as compared to the state-of-the-art approach.
2. To validate DeepCollide's ability to express a wide variety of scene geometries.
3. To investigate the scalability of DeepCollide, as it relates to DoF.
4. To investigate the scalability of DeepCollide, as it relates to training dataset size.
5. To validate the architectural design choices for DeepCollide.
6. To understand how sampling strategy affects DeepCollide's performance.

In all environments, we use the Kuka LBR iiwa 7 R800 robot arm, which has seven DoF [55] .

Towards the first goal of quantifying how DeepCollide performs in the speed vs. error trade-off, we place three Kuka robots (21 DoF) in an environment with 25 obstacles (13 cubes, 12 spheres). We set the Kuka robots at random positions, and we set the obstacles at random positions and orientations. We create three such environments using different random seeds. See Figure 6.

Towards the second goal of validating DeepCollide's ability to express a wide variety of scene geometries, we generate environments containing $10, 20, 30, 40, 50$, and 60 randomly placed obstacles. All of these environments have three robots, which are randomly placed. For each number of obstacles, we test arrangements where the robots are far away from each other (to minimize robot-robot collisions) and close to each other (such that robot-robot collisions are frequent). We use three random seeds, making for a total of $6 \times 2 \times 3 = 36$ environments in this experiment. See Figure 7.

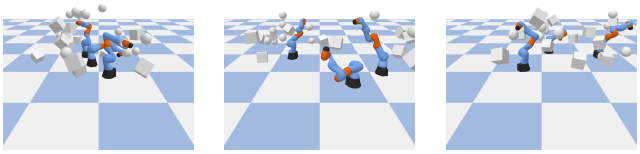
Towards the third goal of investigating DeepCollide's scalability with DoF, we generate environments containing between one (7 DoF) and four (28 DoF) robots, and 25 obstacles. In a sense, this setting is similar to that of Kuffner *et al.* [27] and Okada *et al.* [34], in that there are multiple limbs that must avoid collision with each other in a high-DoF setting. Again, both the obstacles and robots are randomly placed, and we use three random seeds, making for $4 \times 3 = 12$ total environments in this experiment.See Figure 5.

Towards the fourth goal of investigating DeepCollide's scalability with training data, we generate environments as we did in the first experiment, the difference being that we sample more points from these environments.

Towards the fifth goal of validating DeepCollide's architectural choices, we use the same environment as in the speed vs. error experiments, and remove core parts of DeepCollide's architecture (details later).

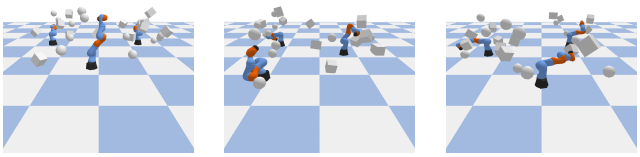
Towards the sixth goal of understanding how sampling strategy affects DeepCollide's performance, we want to see if our method can still perform well if we are not allowed to take samples within obstacles, since that is usually not feasible in a real-world setup. In order to do this, we decided to create environments where approximately half of all con-

The last DoF (the end-effector) actually does not affect collision status, since it just rotates a half-sphere. However, we still include it in our model input, because we think it is important to demonstrate that the model can learn when some DoFs are inconsequential.



(a) 71.3% of the c-space is occupied. (b) 47.6% of the c-space is occupied. (c) 66.3% of the c-space is occupied.

Environments for Speed vs. Error Analysis. Each environment has three randomly placed Kuka LBR iiwa 7 R800 robots (21 DoF), and 25 randomly placed obstacles.



(d) 71.4% of the c-space is occupied. (e) 41.7% of the c-space is occupied. (f) 39.9% of the c-space is occupied.

Environments for Sampling Strategy Analysis. Each environment has three randomly placed Kuka LBR iiwa 7 R800 robots (21 DoF), and 25 randomly placed obstacles.

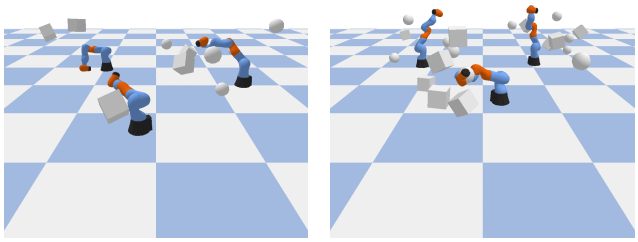
Figure 6. Comparison of environments used in two experimental setups.

figurations result in collision, such that there is enough free space to sample in, but also a proportion of collisions that is high enough to make it non-trivial to avoid sampling within obstacles. So, we generated environments with three robots (21 DoF) far away from each other (to minimize robot-robot collisions) and 25 obstacles (such that there are sufficient objects to collide with). We use three random seeds, making for 3 total environments. See Figure 6.

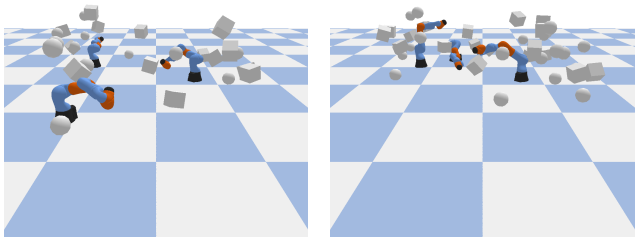
A.3. Ground Truth Collision Detection

We use PyBullet for ground-truth collision detection [31]. Based on our understanding of the source code on GitHub, PyBullet internally uses the standard GJK [4] algorithm for collision detection (along with axis-aligned bounding boxes and EPA [35]).

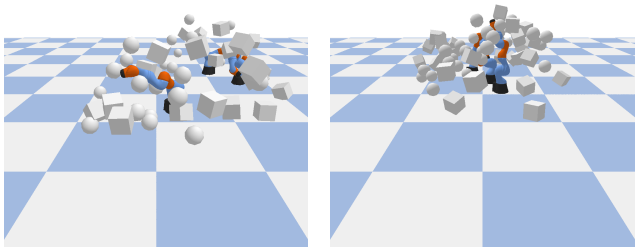
We benchmark our method's speed against PyBullet. However, we note that PyBullet contains some computational overhead time, due to other calculations besides GJK in the physics simulation. While in principle, the overhead could be disentangled from GJK by editing PyBullet's source code, we leave the code as is, because in practice, roboticists would just use the default PyBullet collision detection functions. So, we consider this to be a fair baseline, albeit with the caveat that it could be faster.



(a) 10 obstacles. (b) 20 obstacles.



(c) 30 obstacles. (d) 40 obstacles.



(e) 50 obstacles. (f) 60 obstacles.

Figure 7. **Environments for Determining Impact of Collision Density on Performance.** Each environment has three randomly placed Kuka LBR iiwa 7 R800 robots (21 DoF). Number of obstacles varies from 10 to 60. In the collision density experiments, there are a total of 36 environments like these, generated with a variety of random seeds.

A.4. Metrics

In creating a model for collision detection, we wish to optimize on two fronts: speed and correctness.

As it relates to speed, we measure (following [3]):

- **Training Time:** This is the time it takes to fit our model to the given data on the workspace.
- **Time per Inference:** This measures the average time it takes the model to tell us whether a singular configuration will collide. We measure this separately from training time because the training time is a one-time cost, but after

we have the trained model, we can make as many queries as we want.

We leave out the forward kinematics kernel computation [10] from our speed calculations, because all the methods (*i.e.*, DeepCollide, Fastron FK) we compare use it. So, whether or not we include it, the conclusion of which method is fastest remains the same. Furthermore, since we want to find out which method is the fastest, we believe it is more illuminative to isolate and base our comparison on the parts of the methods that are different.

As it relates to correctness, we measure (again, after [3]):

- **Accuracy:** This is defined as $\frac{TP+TN}{TP+TN+FP+FN}$, where TP is the number of true positives, TN is the number of true negatives, FP is the number of false positives, and FN is the number of false negatives, from the model's predictions. This is the standard metric used in machine learning applications.
- **TPR:** This is defined as $\frac{TP}{TP+FN}$. Intuitively, it quantifies the percent of actual collisions that our model catches. A high value for this metric indicates that our model will be able to show the robot where to avoid obstacles.
- **TNR:** This is defined as $\frac{TN}{TN+FP}$. We can think of this as the percent of the free space that our model correctly identifies. This metric is important because it shows how good the collision detection function will be at showing the robot collision-free paths to its destination.

The TPR and TNR only make sense when reported together (*i.e.*, in a Pareto analysis), because they have an inverse relationship, such that one could artifically set the bias hyperparameter to make one of them always 100% at the cost of the other always being 0%. Thus, we only analyze TPR and TNR in the context of Pareto frontiers, where we can see them side by side. It would not make sense to analyze them in the other experiments, such as DoF vs. performance, since the independent variables are already fixed (*e.g.*, to DoF), making it tricky to analyze both TPR and TNR on a two-dimensional diagram. For these experiments, we only report accuracy.

A.5. Implementation Details

Libraries: We conduct all experiments in Python 3.9.16. For robot simulation and ground truth collision detection, we use PyBullet 3.2.5 [31]. For general computation, we use Numpy 1.24.3 [56]. For our deep learning models, we use PyTorch 1.10.2 [54]. For Fastron, we use the official implementation created by the authors [3].

Hardware: We run most experiments on an AMD Ryzen Threadripper 3960X 24-Core Processor as the CPU. Additionally, we utilize a singular NVIDIA GeForce RTX 3090 to train DeepCollide, because GPUs are widely known to accelerate training of deep learning models.

We do not use a GPU to test Fastron, because the official implementation does not support that. In contrast, we test DeepCollide both on CPU, and on GPU. We conduct CPU testing to create a close-as-possible comparison to Fastron. We also conduct GPU testing because we want to realize the full potential of DeepCollide, and one of the advantages of deep learning is that it is massively parallelizable on GPUs [57].

Hyperparameter	Possible Values
$\mathcal{S}_{max}$ (max # supp. pts)	$3000, 10000, 30000$
$\mathcal{I}_{max}$ (max # updates)	$\mathbf{5000}, 30000$
γ (kernel width)	$1, \mathbf{5}, 10$
β (positive bias)	$1, \mathbf{500}, 1000$

Table 1. **Fastron Hyperparameter Sweep:** We use the recommended hyperparameters (**bolded**) from [3], and also test more to give it a fair chance ($3 \times 2 \times 3 \times 3 = 54$ combinations).

Hyperparameter	Possible Values
$\|L\|$ (# positional freq.)	4, 8, 12
β (positive bias)	1, 2, 5
σ (positional freq. increment)	$\frac{1}{2}$, 1, 2

Table 2. **DeepCollide Hyperparameter Sweep:** The $3 \times 3 \times 3 = 27$ combinations of hyperparameters tried for our method.

A.6. Hyperparameter Recommendations for DeepCollide

A hyperparameter setting for DeepCollide that was on the Pareto-optimal frontier for all charts (*i.e.*, accuracy vs. inference time, accuracy vs. train time, TPR vs. TNR) in Figure 3 was $|L| = 12, \beta = 1, \sigma = 1$. This corresponds to $87.6\% \pm 1.3\%$ accuracy, $(4.5 \pm 0.4) * 10^{-5}$ seconds per inference on CPU ($(0.2 \pm 0.1) * 10^{-5}$ seconds per inference on GPU), 14.1 ± 0.2 seconds to train, $89.9\% \pm 2.0\%$ TPR, and $81.9\% \pm 7.9\%$ TNR, where the mean and standard deviation are calculated across all three environments (*i.e.*, $n = 3$).

B. Impact of Sampling Strategy

B.1. Rationale

In real-world scenarios where workspace geometry is not known, collision data must be gathered by physically setting the robot to multiple configurations and checking if they collide. A result of this setup is that it is often infeasible to obtain datapoints that are embedded deep within obstacles (for instance, we cannot move a robot arm into the center of a solid rock).

Thus, in this experiment, we adopt our sampling strategy to better understand the limits of our method in such a scenario. Particularly, we conduct a rejection sampling strategy, in which we start by sampling points uniformly, but reject them and resample if they fall too far within an obstacle. We allow configurations that collide with an obstacle, provided the collision depth is less than 7.5% of the

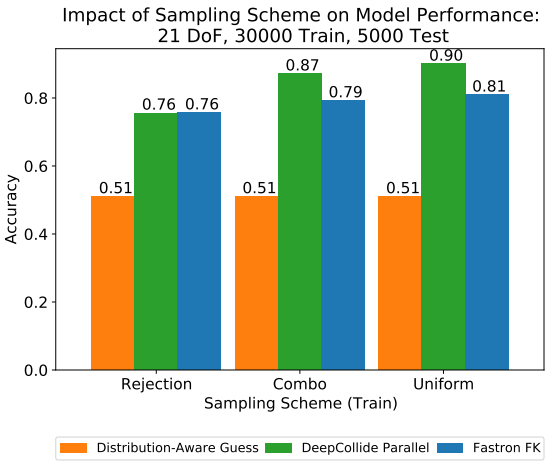


Figure 8. **Impact of Sampling Strategy on Correctness.** The points on the graph show the average accuracy over three environments. We compare DeepCollide on GPU to Fastron FK. The horizontal axis label is the sampling strategy that was used to generate the training data; all testing data was sampled uniformly. "Rejection" indicates that we used rejection sampling to only sample points that were either in free space, or not more than 7.5% indented into an obstacle; "Uniform" indicates that we sampled points uniformly across the configuration space, regardless of whether they were within obstacles or not; "Combo" indicates that half the points came from the "Rejection" strategy and the other half came from the "Uniform" strategy.

obstacle's diameter (in the case of a sphere) or side length (in the case of a cube). This is reasonable because many obstacles in real life are slightly deformable or displacable.

We have three separate training data sampling schemes that we compare: rejection, uniform, and combo. Combo sampling is when we sample half the training points with uniform sampling and half the training points with rejection sampling. We then test on uniformly sampled configuration space points.

In this experiment, we only study the impact on accuracy, because the time it takes to pass data through a machine learning model is independent of the label. We also use the parallelized GPU version of DeepCollide at test time, since the accuracy, in principle, should be the same as the non-parallelized version.

B.2. Outcome

Figure 8 leads us to a few conclusions. Firstly, when we conduct rejection sampling for the train points, the performance is worse than uniformly sampling training points. This shows the importance of sampling a training distribution that reflects the testing distribution for machine learning models. Secondly, machine learning methods are better than the distribution-aware guess baseline, even if there is a disconnect between the training and testing distributions. Finally, the advantage of DeepCollide over Fastron is most apparent when the training and testing distributions were both sampled with the same uniform strategy. The advantage disappears when rejection sampling generates the training distribution, although they are still roughly equal, and both better than the distribution-aware guessing baseline.

C. Impact of Sample Size

In measuring the impact of sample size, we set $\mathcal{I}_{max}$ to $50,000$ and $\mathcal{S}_{max}$ to $50,000$ for Fastron FK, so that it is equipped to process the large number of training points (up to $100,000$).

C.1. Impact on Correctness

Figure 9a shows the impact of train sample size on accuracy. Obviously, model performance gets better as we add more training points for both models. Furthermore, DeepCollide clearly outperforms Fastron FK, regardless of sample size.

We also note that the support point and iteration limits ($\mathcal{S}_{max} = \mathcal{I}_{max} = 50,000$) on Fastron may theoretically diminish the accuracy benefits of more training data. However, as we see in the next subsection, Fastron already has a time scalability problem with respect to training data, and increasing those limits could only serve to exacerbate this.

C.2. Impact on Speed

Figures 9b and 9c show the impact of train sample size on model speed. It is in line with Section 4's theoretical expectations of time complexity. The implication of this is that DeepCollide scales much better than Fastron, with respect to training sample size. Notably, as the number of train samples gets high, Fastron's inference time (Figure 9c) is slower than even the PyBullet baseline (which has considerable overhead besides GJK). In contrast, DeepCollide is consistently faster than PyBullet, and the GPU-parallelized version of DeepCollide is over an order of magnitude better than any of the other methods.

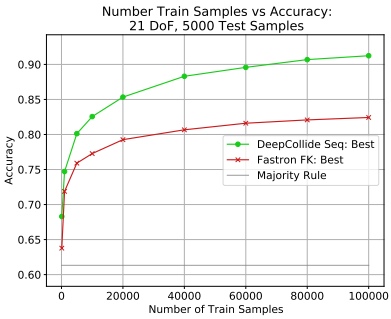
D. Impact of Collision Density

D.1. Impact on Correctness

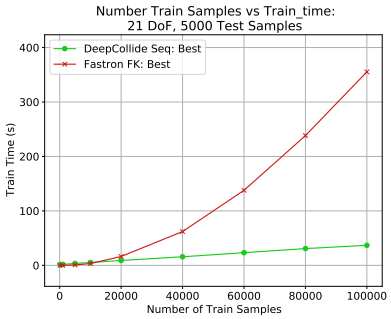
From Figure 10a, we see that DeepCollide consistently outperforms Fastron FK on accuracy. We also see that when the collision density is close to 50%, the benefit of machine learning methods is most clear over distribution-aware guessing, albeit with a slight drop in the performance of both models.

D.2. Impact on Speed

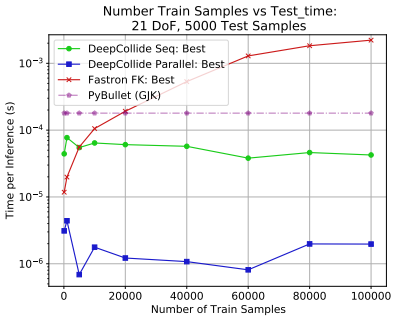
Figures 10b and 10c show the impact of collision density on model speed. Firstly, we notice that overall, collision density does not significantly impact machine learning model



(a) Sample Size vs. Accuracy

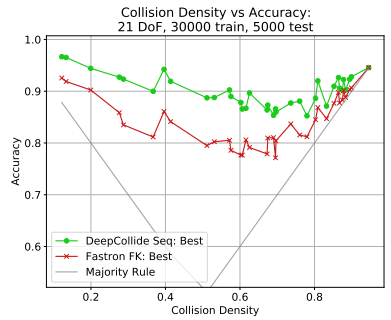


(b) Sample Size vs. Train Time

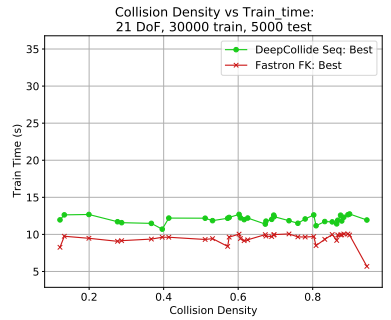


(c) Sample Size vs. Test Time

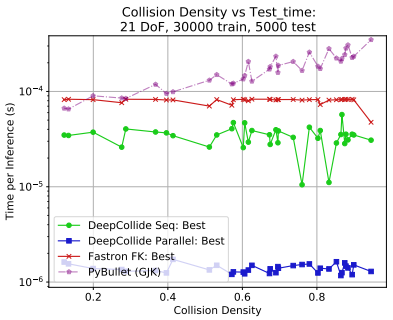
Figure 9. **Impact of Sample Size on correctness.** The points on the graph show the average performance over three environments. Interpretation is the same as Figure 4.



(a) Collision Density vs. Accuracy



(b) Collision Density vs. Train Time



(c) Collision Density vs. Test Time

Figure 10. **Impact of Collision Density on correctness.** Interpretation is the same as Figure 4, except that we are measuring collision density on the x-axis, and these graphs correspond to environments in Figure 7, with only one environment per collision density.

speed in training or testing. The only exception is Fastron's slight speedup at the highest collision density shown. This may be due to Fastron's redundant support point removal mechanism, where it automatically removes points from the support set that would have a positive margin if they were excluded from the support set (more details in [3]). In the high collision density case, fewer points may be needed to construct the support set, since most of the points have the collision label – leading to quicker computations. However, PyBullet's collision checker does slow down as collision density increases, since there are more pairs of objects to check collisions between.

Finally, in line with what we have seen so far, DeepCollide generally has quicker test time but slower train time than Fastron. We also see that at test time (Figure 10c), even the CPU version of DeepCollide is always faster than PyBullet, while the GPU-parallelized version is almost two orders of magnitude faster. This highlights the benefit of GPU-based parallelization in speeding up collision detection.

E. Model Ablation

We test the impact of removing the crucial design components of DeepCollide, particularly: the ending BatchNorm, the skip connections, and the Fourier positional encoding. Figure 11 validates these design choices, as DeepCollide with all of these components has the highest accuracy (averaged over three environments). The positional encoding has the biggest impact on the accuracy, in line with previous work [20]. The ending BatchNorm and skip connections also have an impact, albeit of less than a percentage point. We also tried removing BatchNorm from all layers (rather than just the output node), but we found that this led to diverging NaN loss, and therefore did not include it in Figure 11.

F. Trade-offs

F.1. Speed vs. Correctness Trade-off

We choose inference time vs. error as defined by accuracy for Figure 3a's Pareto frontier, because the goal of this

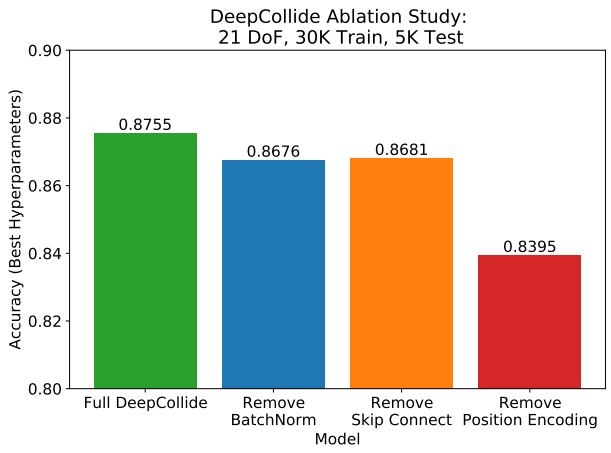


Figure 11. **Ablation Results:** We remove components of DeepCollide, and compare the accuracy of each of these model versions. In line with previous experiments, only the hyperparameter choices yielding the best results are shown.

work is to provide the optimal balance between speed and correctness. As for why we specifically choose inference time as our speed metric in this Pareto frontier, asymptotically, speed can be expressed as a function of inference time alone. We choose accuracy as our correctness metric because because it accounts for all of the categories in the confusion matrix (true positives, true negatives, false positives, false negatives); while other metrics (like TPR) do not account for some of the categories.

We see that DeepCollide (with exception of the one PyBullet point) makes up all of the points on the Pareto-optimal frontier, so we can conclude that DeepCollide outperforms Fastron FK in regards to the speed-correctness trade-off. We can be confident in this conclusion, because we did it over a variety of hyperparameters, including the hyperparameters that the authors of Fastron deemed to be (near-)optimal for their model [3]. Furthermore, we conducted these evaluations over multiple random environments with varying collision densities, so we are fairly confident that the result is not environment-specific. Furthermore, we see that using GPU parallelization increases DeepCollide's speed by about an order of magnitude, greatly outpeforming all the baselines.

Furthermore, while training time does not matter in the asymptotic case (predicting $n \to \infty$, where n is the number of points predicted), it still is important for practical applications. Thus, we also investigate the trade-off between train time and accuracy in Figure 3b. Here, both DeepCollide and Fastron FK have instances on the Pareto-optimal frontier. We observe that Fastron FK has the fastest overall model, at the cost of accuracy, while DeepCollide has the most accurate overall models, at the cost of speed. This is emblematic of the general differences between neural networks and traditional machine learning methods – neural networks are typically more accurate, but often require extra training time to achieve their high accuracy.

A natural question that may arise is how we are able to achieve faster inference times than and comparable train times to Fastron, when our DeepCollide neural network has hundreds of thousands of parameters. Actually, this is in line with our theoretical analysis from Section 4. In high-DoF settings, we need tens of thousands of training samples to get an accurate collision detection function, due to the curse of dimensionality. Given that Fastron's time per inference is linear with respect to both training set size and DoF, while its training time is nearly quadratic with respect to training set size and linear with respect to DoF; this quickly becomes computationally prohibitive, and essentially limits Fastron to low-DoF and low-data cases. In contrast, even though DeepCollide also has (nominally, as will be shown in Section 6) linear time complexity with respect to DoF, its time per inference has no dependency on training dataset size, and its training time is only linear with respect to training dataset size. Thus, in even a moderate-data case like this evaluation, we see DeepCollide's benefits.

Furthermore, the use of GPUs in training DeepCollide (which the official implementation of Fastron does not support) boosts DeepCollide's training speed.

F.2. Error Modality Trade-off

We choose error as defined by TPR vs. error as defined by TNR for Figure 3c's Pareto frontier, because while accuracy is a good "catch-all" correctness metric, it misses the fine-grained detail of what kinds of errors are made. In particular, we care about two kinds of errors: (1) failing to detect a collision (which can be expressed by $1 - TPR$); (2) failing to find a part of the free path (which can be expressed by $1 - TNR$). Thus, this Pareto chart of TPR error vs. TNR error dives deeper into the error modalities of the methods.

Here, the Pareto-optimal frontier is made up of a combination of DeepCollide and Fastron FK instances, but it is still mostly DeepCollide. Thus, we can say that both models have comparable ability in balancing collision and free path detections, although DeepCollide appears to be slightly better.