A. Further experimental analysis

A.1. Kuramoto LSTM vs. NRI comparison

From the results in our main paper it became evident that a simple LSTM model excels at predicting the dynamics of a network of phase-coupled oscillators (Kuramoto model) for short periods of time, while predictive performance deteriorates for longer sequences. It is interesting to compare the qualitative predictive behavior of this fully recurrent model with our NRI (learned) model that models the state $\mathbf{x}^{t+1}$ at time $t+1$ solely based on the state $\mathbf{x}^{t}$ at time t and the learned latent interaction graph. In Fig. 9 we provide visualizations of model predictions for the LSTM (joint) and the NRI (learned) model, compared to the ground truth continuation of the simulation.

It can be seen that the LSTM model correctly captures the shape of the sinusoidal waveform but fails to model the phase dynamics that arise due to the interactions between the oscillators. Our NRI (learned) model captures the qualitative behavior of the original coupled model at a very high precision and only in some cases slightly misses the phase dynamics (e.g. in the purple and green curve in the lower right plot). The LSTM model rarely matches the phase of the ground truth trajectory in the last few time steps and often completely goes “out of sync” by up to half a wavelength.

A.2. Spring simulation variants

In addition to the experiments presented in the main paper, we analyze the following two variants of the spring simulation experimental setting: i) we test a trained model on completely non-interacting (free-floating) particles, and ii) we add a third edge type with a lower coupling constant.

To test whether our model can infer an empty graph, we create a test set of 1000 simulations with 5 non-interacting particles and test an unsupervised NRI model which was trained on the spring simulation dataset with 5 particles as before. We find that it achieves an accuracy of 98.4% in identifying ”no interaction” edges (i.e. the empty graph).

The last variant explores a simulation with more than two known edge types. We follow the same procedure for the spring simulation with 5 particles as before with the exception of adding an additional edge type with coupling constant $k_{ij} = 0.5$ (all three edge types are sampled with equal probability). We fit an unsupervised NRI model to this data ($K = 3$ in this case, other settings as before) and find that it achieves an accuracy of 99.2% in discovering the correct edge types.

A.3. Motion capture visualizations

In Fig. 10 we visualize predictions of a trained NRI model with learned latent graph for the motion capture dataset. We show 30 predicted time steps of future movement, conditioned on 49 time steps that are provided as ground truth to the model. It can be seen that the model can capture the overall form of the movement with high precision. Mistakes (e.g. the misplaced toe node in frame 30) are possible due to the accumulation of small errors when predicting over long sequences with little chance of recovery. Curriculum learning schemes where noise is gradually added to training sequences can potentially alleviate this issue.

A.4. NBA visualizations

We show examples of three pick and roll trajectories in Fig. 11. In the left column we show the ground truth, in the middle we show our prediction and in the right we show the edges that where sampled by our encoder. As we can see even when our model does not predict the true future path, which is extremely challenging for this data, it still makes semantically reasonable predictions. For example in the middle row it predicts that the player defending the ball handler passes between him and the screener (going over the screen) which is a reasonable outcome even though in reality the defenders switched players.

B. Simulation data

B.1. Springs model

We simulate $N \in \{5, 10\}$ particles (point masses) in a 2D box with no external forces (besides elastic collisions with the box). We randomly connect, with probability 0.5, each pair of particles with a spring. The particles connected by springs interact via forces given by Hooke’s law $F_{ij} = -k(r_i - r_j)$ where F_{ij} is the force applied to particle v_i by particle v_j, k is the spring constant and r_i is the 2D location vector of particle v_i. The initial location is sampled from a Gaussian $\mathcal{N}(0, 0.5)$, and the initial velocity is a random vector of norm 0.5. Given the initial locations and velocity we can simulate the trajectories by solving Newton’s equations of motion PDE. We do this by leapfrog integration using a step size of 0.001 and then subsample each 100 steps to get our training and testing trajectories.

We note that since the leapfrog integration is differentiable, we are able to use it as a ground-truth decoder and backpropagate through it to train the encoder. We implemented the leapfrog integration in PyTorch, which allows us to compare model performance with a learned decoder versus the ground-truth simulation decoder.

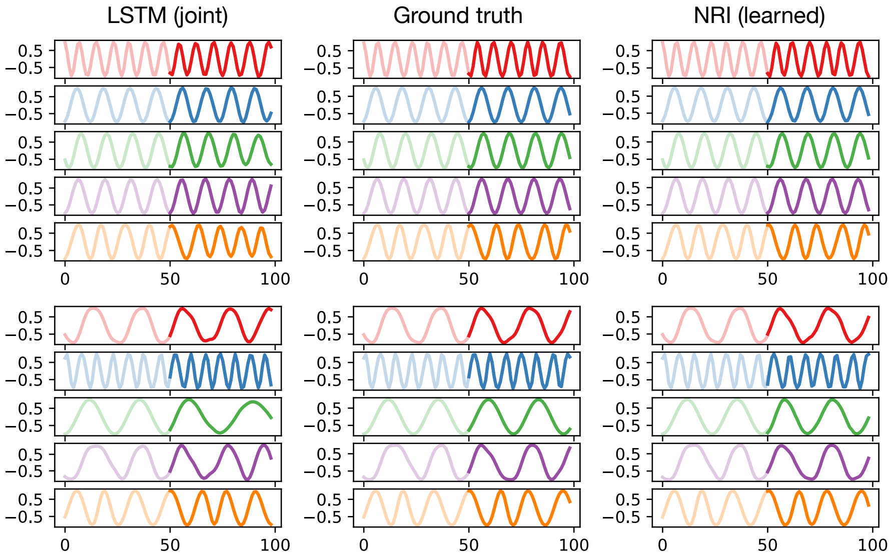


Figure 9. Qualitative comparison of model predictions for the LSTM (joint) model (*left*) and the NRI (learned) model (*right*). The ground truth trajectories (*middle*) are shown for reference.

B.2. Charged particles model

Similar to the springs model, we simulate $N \in \{5, 10\}$ particles in a 2D box, but instead of springs now our particles carry positive or negative charges $q_i \in \{\pm q\}$, sampled with uniform probability, and interact via Coulomb forces: $F_{ij} = C \cdot \text{sign}(q_i \cdot q_j)\frac{r_i - r_j}{||r_i - r_j||^3}$ where C is some constant. Unlike the springs simulations, here every two particles interact, although the interaction might be weak if they stay far apart, but they can either attract or repel each other.

Since the forces diverge when the distance between particles goes to zero, this can cause issues when integrating with a fixed step size. The problem might be solved by using a much smaller step size, but this would slow the generation considerably. To circumvent this problem, we clip the forces to some maximum absolute value. While not being exactly physically accurate, the trajectories are indistinguishable to a human observer and the generation process is now stable.

The force clipping does, however, create a problem for the simulation ground-truth decoder, as gradients become zero when the forces are clipped during the simulation. We attempted to fix this by using "soft" clipping with a $\text{softplus}(x) = \log(1 + e^x)$ function in the differentiable simulation decoder, but this similarly resulted in vanishing gradients once the model gets stuck in an unfavorable regime with large forces.

B.3. Phase-coupled oscillators

The Kuramoto model is a nonlinear system of phase-coupled oscillators that can exhibit a range of complicated dynamics based on the distribution of the oscillators' internal frequencies and their coupling strengths. We use the common form for the Kuramoto model given by the following differential equation:

$$\frac{d\phi_i}{dt} = \omega_i + \sum_{j \neq i} k_{ij} \sin(\phi_i - \phi_j) \qquad (20)$$

with phases ϕ_i, coupling constants k_{ij}, and intrinsic frequencies ω_i. We simulate 1D trajectories by solving Eq. (20) with a fourth-order Runge-Kutta integrator with step size 0.01.

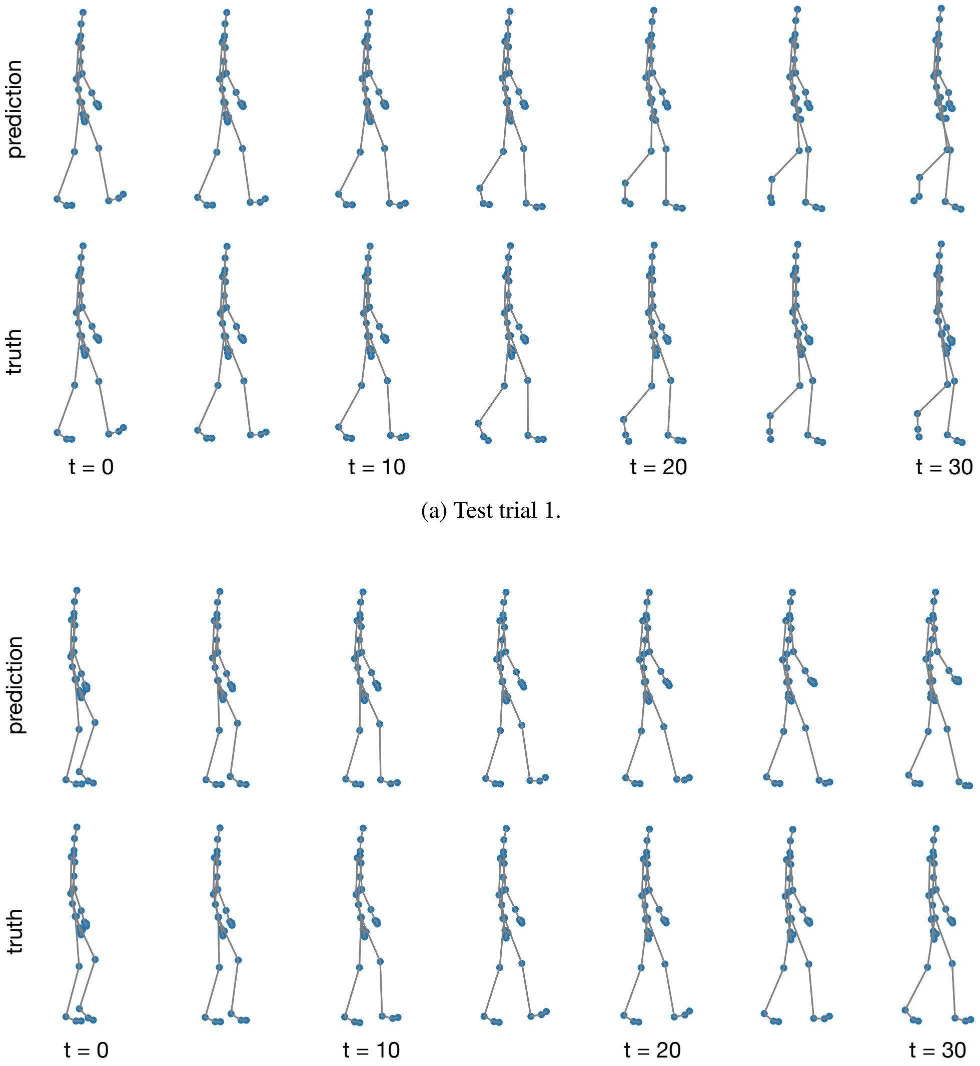
We simulate $N \in \{5, 10\}$ phase-coupled oscillators in 1D with intrinsic frequencies ω_i and initial phases $\phi_i^{t=1}$ sampled uniformly from $[1, 10)$ and $[0, 2\pi)$, respectively. We randomly, with probability of 0.5, connect pairs of oscillators v_i and v_j (undirected) with a coupling constant $k_{ij} = 1$. All other coupling constants are set to 0. We subsample the simulated ϕ_i by a factor of 10 and create trajectories $\mathbf{x}_i$ by concatenating $\frac{d\phi_i}{dt}$, $\sin \phi_i$, and the intrinsic frequencies ω_i (copied for every time step as ω_i are static).

C. Implementation details

We will describe here the details of our encoder and decoder implementations.

C.1. Vectorized implementation

The message passing operations $v{\to}e$ and $v{\to}e$ can be evaluated in parallel for all nodes (or edges) in the graph and allow for an efficient vectorized implementation. More specifi-



(a) Test trial 1.

(b) Test trial 2.

Figure 10. Examples of predicted walking motion of an NRI model with learned latent graph compared to ground truth sequences for two different test set trials.

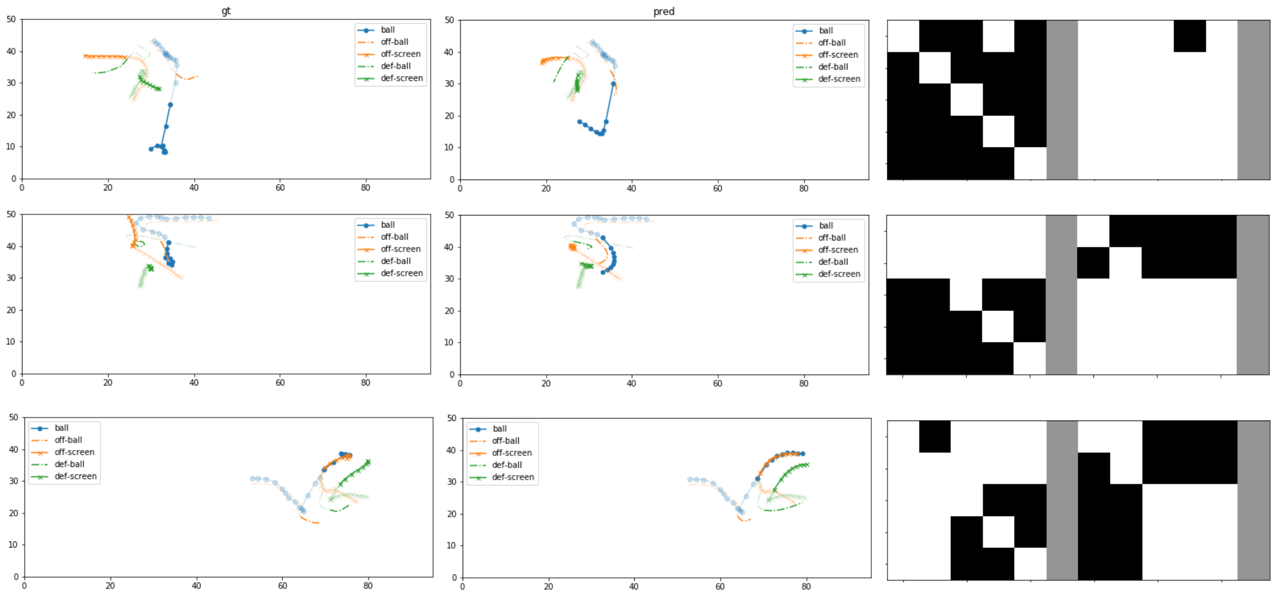


Figure 11. Visualization of NBA trajectories. *Left*: ground truth; *middle*: model prediction; *right*: sampled edges.

cally, the node-to-edge message passing function $f_{v\to e}$ can be vectorized as:

$$\mathbf{H}_e^1 = f_e([\mathbf{M}^{\text{in}}_{v\to e}\mathbf{H}_v^1, \mathbf{M}^{\text{out}}_{v\to e}\mathbf{H}_v^1]) \qquad (21)$$

with $\mathbf{H}_v = [\mathbf{h}_1^\top, \mathbf{h}_2^\top, \ldots, \mathbf{h}_N^\top]^\top \in \mathbb{R}^{N\times F}$ and $\mathbf{H}_e \in \mathbb{R}^{E\times F}$ defined analogously (layer index omitted), where F and E are the total number of features and edges, respectively. $(\cdot)^\top$ denotes transposition. Both message passing matrices $\mathbf{M}_{v\to e} \in \mathbb{R}^{E\times N}$ are dependent on the graph structure and can be computed in advance if the underlying graph is static. $\mathbf{M}^{\text{in}}_{v\to e}$ is a sparse binary matrix with $\mathbf{M}^{\text{in}}_{v\to e,ij} = 1$ when the j-th node is connected to the i-th edge (arbitrary ordering) via an incoming link and 0 otherwise. $\mathbf{M}^{\text{out}}_{v\to e}$ is defined analogously for outgoing edges.

Similarly, we can vectorize the edge-to-node message passing function $f_{e\to v}$ as:

$$\mathbf{H}_v^2 = f_v(\mathbf{M}^{\text{in}}_{e\to v}\mathbf{H}_e^1) \qquad (22)$$

with $\mathbf{M}^{\text{in}}_{e\to v} = (\mathbf{M}^{\text{in}}_{v\to e})^\top$. For large sparse graphs (e.g. by constraining interactions to nearest neighbors), it can be beneficial to make use of sparse-dense matrix multiplications, effectively allowing for an $O(E)$ algorithm.

C.2. MLP Encoder

The basic building block of our MLP encoder is a 2-layer MLP with hidden and output dimension of 256, with batch normalization, dropout, and ELU activations. Given this, the forward model for our encoder is given by the code snippet in Fig. 12. The node2edge module returns for each edge the concatenation of the receiver and sender features. The edge2node module accumulates all incoming edge features via a sum.

```
x = self.mlp1(x) # 2-layer ELU net per node
x = self.node2edge(x)
x = self.mlp2(x)
x_skip = x

x = self.edge2node(x)
x = self.mlp3(x)
x = self.node2edge(x)
x = torch.cat((x, x_skip), dim=2)
x = self.mlp4(x)
return self.fully_connected_out(x)
```

Figure 12. PyTorch code snippet of the MLP encoder forward pass.

C.3. CNN Encoder

The CNN encoder uses another block which performs 1D convolutions with attention. This allows for encoding with changing trajectory size, and is also appropriate for tasks like the charged particle simulations when the interaction can be strong for a small fraction of time. The forward computation of this module is presented in Fig. 13 and the overall decoder in Fig. 14.

C.4. MLP Decoder

In Fig. 15 we present the code for a single time-step prediction using our MLP decoder for Markovian data.

```
# CNN block
# inputs is of shape ExFxT, E: number of edges,
#  T: sequence length, F: num. features
x = F.relu(self.conv1(inputs))
x = self.batch_norm1(x)
x = self.pool(x)
x = F.relu(self.conv2(x))
x = self.batch_norm2(x)
out = self.conv_out(x)
attention = softmax(self.conv_attn(x), axis=2)

out = (out * attention).mean(dim=2)
return out
```

Figure 13. PyTorch code snippet of the CNN block forward pass, used in the CNN encoder.

```
# CNN encoder
x = self.node2edge(x)
x = self.cnn(x) # CNN block from above
x = self.mlp1(x) # 2-layer ELU net per node
x_skip = x

x = self.edge2node(x)
x = self.mlp2(x)
x = self.node2edge(x)
x = torch.cat((x, x_skip), dim=2)
x = self.mlp3(x)
return self.fully_connected_out(x)
```

Figure 14. PyTorch code snippet of the CNN encoder model forward pass.

```
# Single prediction step
pre_msg = self.node2edge(inputs)

# Run separate MLP for every edge type
# For non-edge: start_idx=1, otherwise 0
for i in range(start_idx, num_edges):
  msg = F.relu(self.msg_fc1[i](pre_msg))
  msg = F.relu(self.msg_fc2[i](msg))
  msg = msg * edge_type[:, :, :, i:i + 1]
  all_msgs += msg

# Aggregate all msgs to receiver
agg_msgs = self.edge2node(all_msgs)
hidden = torch.cat([inputs, agg_msgs], dim=-1)

# Output MLP
pred = F.relu(self.out_fc1(hidden)
pred = F.relu(self.out_fc2(pred)
pred = self.out_fc3(pred)

return inputs + pred
```

Figure 15. PyTorch code snippet of a single prediction step in the MLP decoder.

C.5. RNN Decoder

The RNN decoder adds a GRU style update to the single step prediction, the code snippet for the GRU module is presented in Fig. 16 and the overall RNN decoder in Fig. 17.

```
# GRU block
# Takes arguments: inputs, agg_msgs, hidden
r = F.sigmoid(self.input_r(inputs) +
      self.hidden_r(agg_msgs))
i = F.sigmoid(self.input_i(inputs) +
      self.hidden_i(agg_msgs))
n = F.tanh(self.input_n(inputs) +
      r * self.hidden_h(agg_msgs))
hidden = (1 - i) * n + i * hidden
return hidden
```

Figure 16. PyTorch code snippet of a GRU block, used in the RNN decoder.

```
# Single prediction step
pre_msg = self.node2edge(inputs)

# Run separate MLP for every edge type
# For non-edge: start_idx=1, otherwise 0
for i in range(start_idx, num_edges):
  msg = F.relu(self.msg_fc1[i](pre_msg))
  msg = F.relu(self.msg_fc2[i](msg))
  msg = msg * edge_type[:, :, :, i:i + 1]
  # Average over types for stability
  all_msgs += msg/(num_edges-start_idx)

# Aggregate all msgs to receiver
agg_msgs = self.edge2node(all_msgs)

# GRU-style gated aggregation (see GRU block)
hidden = self.gru(inputs, agg_msgs, hidden)

# Output MLP
pred = F.relu(self.out_fc1(hidden))
pred = F.relu(self.out_fc2(pred))
pred = self.out_fc3(pred)

# Predict position/velocity difference
pred = inputs + pred

return pred, hidden
```

Figure 17. PyTorch code snippet of a single prediction step in the RNN decoder.

D. Experiment details

All experiments were run using the Adam optimizer (Kingma & Ba, 2015) with a learning rate of 0.0005, decayed by a factor of 0.5 every 200 epochs. Unless otherwise noted, we train with a batch size of 128. The concrete distribution is used with $\tau = 0.5$. During testing, we replace the concrete distribution with a categorical distribution to obtain discrete latent edge types. Physical simulation and sports tracking experiments were run for 500 training epochs. For motion capture data we used 200 training epochs, as models tended to converge earlier. We saved model checkpoints after every epoch whenever the validation set performance (measured by path prediction MSE) improved and loaded the best performing model for test set evaluation. We observed that using significantly higher learning rates than 0.0005 often produced suboptimal decoders that ignored the latent graph structure.

D.1. Physics simulations experiments

The springs, charged particles and Kuramoto datasets each contain 50k training instances and 10k validation and test instances. Training and validation trajectories where of length 49 while test trajectories continue for another 20 time steps (50 for visualization). We train an MLP encoder for the springs experiment, and CNN encoder for the charged particles and Kuramoto experiments. All experiments used MLP decoders and two edge types. For the Kuramoto model experiments, we explicitly hard-coded the first edge type as a "non-edge", i.e. no messages are passed along edges of this type.

As noted previously, all of our MLPs have hidden and output dimension of 256. The overall input/output dimension of our model is 4 for the springs and charged particles experiments (2D position and velocity) and 3 for the Kuramoto model experiments (phase-difference, amplitude and intrinsic frequency). During training, we use teacher forcing in every 10-th time step (i.e. every 10th time step, the model receives a ground truth input, otherwise it receives its previous prediction as input). As we always have two edge types in these experiments and their ordering is arbitrary (apart from the Kuramoto model where we assign a special role to edge type 1), we choose the ordering for which the accuracy is highest.

D.1.1. Baselines

Edge recovery experiments In edge recovery experiments, we report the following baselines along with the performance of our NRI (learned) model:

- **Corr. (path)**: We calculate a correlation matrix R, where $R_{ij} = \frac{C_{ij}}{\sqrt{C_{ii}C_{jj}}}$ with C_{ij} being the covariance between all trajectories $\mathbf{x}_i$ and $\mathbf{x}_j$ (for objects v_i and v_j) in the training and validation sets. We determine an ideal threshold θ so that $A_{ij} = 1$ if $R_{ij} > \theta$ and $A_{ij} = 0$ otherwise, based on predictive accuracy on the combined training and validation set. A_{ij} denotes the presence of an interaction edge (arbitrary type) between object v_i and v_j. We repeat the same procedure for the absolute value of R_{ij}, i.e. $A_{ij} = 1$ if $|R_{ij}| > \theta'$ and $A_{ij} = 0$ otherwise. Lastly, we pick whichever of the two (θ or θ') produced the best match with the ground truth graph (i.e. highest accuracy score) and report test set accuracy with this setting.

- **Corr. (LSTM)**: Here, we train a two-layer LSTM with shared parameters and 256 hidden units that models each trajectory individually. It is trained to predict the position and velocity for every time step directly and is conditioned on the previous time steps. The input to the model is passed through a two-layer MLP (256 hidden units and ReLU activations) before it is passed to the LSTM, similarly we pass the LSTM output (last time step) through a two-layer MLP (256 hidden units and ReLU activation on the hidden layer). We provide ground truth trajectory information as input at every time step. We train to minimize MSE between model prediction and ground truth path. We train this model for 10 epochs and finally apply the same correlation matrix procedure as in Corr. (path), but this time calculating correlations between the output of the second LSTM layer at the last time step (instead of using the raw trajectory features). The LSTM is only trained on the training set. The optimal correlation threshold is estimated using the combined training and validation set.

- **NRI (sim.)**: In this setting, we replace the decoder of the NRI model with the ground-truth simulator (i.e. the integrator of the Newtonian equations of motion). We implement both the charged particle and the springs simulator in PyTorch which gives us access to gradient information. We train the overall model with the same settings as the original NRI (learned) model by backpropagating directly through the simulator. We find that for the springs simulation, a single leap-frog integration step is sufficient to closely approximate the trajectory of the original simulation, which was generated with 100 leap-frog steps per time step. For the charged particle simulation, 100 leap-frog steps per time step are necessary to match the original trajectory when testing the simulation decoder in isolation. We find, however, that due to the force clipping necessary to stabilize the original charged particle simulation, gradients will often become zero, making model training difficult or infeasible.

- **Supervised**: For this baseline, we train the encoder in isolation and provide ground-truth interaction graphs as labels. We train using a cross-entropy error and monitor the validation accuracy (edge prediction) for model checkpointing. We train with dropout of $p = 0.5$ on the hidden layer representation of every MLP in the encoder model, in order to avoid overfitting.

Path prediction experiments Here, we use the following baselines along with our NRI (learned) model:

- **Static**: This baseline simply copies the previous state vector $\mathbf{x}^{t+1} = \mathbf{x}^t$.
- **LSTM (single)**: Same as the LSTM model in Corr. (LSTM), but trained to predict the state vector difference at every time step (as in the NRI model). Instead of providing ground truth input at every time step, we use the same training protocol as for an NRI model with recurrent decoder (see main paper).
- **LSTM (joint)**: This baseline differs from LSTM (single) in that it concatenates the input representations from all objects after passing them through the input MLP. This concatenated representation is fed into a single LSTM where the hidden unit number is multiplied by the number of objects—otherwise same setting as LSTM (single). The output of the second LSTM layer at the last time step is then divided into vectors of same size, one for each object, and fed through the output MLP to predict the state difference for each object separately. LSTM (joint) is trained with same training protocol as the LSTM (single) model.
- **NRI (full graph)**: For this model, we keep the latent graph fixed (fully-connected on edge type 2; note that edge types are exclusive, i.e. edges of type 1 are not present in this case) and train the decoder in isolation in the otherwise same setting as the NRI (learned) model.
- **NRI (true graph)**: Here, we train the decoder in isolation and provide the ground truth interaction graph as latent graph representation.

D.2. Motion capture data experiments

Our extracted motion capture dataset has a total size of 8,063 frames for 31 tracked points each. We normalize all features (position/velocity) to maximum absolute value of 1. Training and validation set samples are 49 frames long (non-overlapping segments extracted from the respective trials). Test set samples are 99 frames long. In the main paper, we report results on the last 50 frames of this test set data.

We choose the same hyperparameter settings as in the physical simulation experiments, with the exception that we train models for 200 epochs and with a batch size of 8. Our model here uses an MLP encoder and an RNN decoder (as the dynamics are not Markovian). We further take samples from the discrete distribution during the forward pass in training and calculate gradients via the concrete relaxation. The baselines are identical to before (path prediction experiments for physical simulations) with the following exception: For LSTM (joint) we choose a smaller hidden layer size of 128 units and train with a batch size of 1, as the model did otherwise not fit in GPU memory.

D.3. NBA experiments

For the NBA data each example is a 25 step trajectory of a pick and roll (PnR) instance, subsampled from the original 25 frames-per-second SportVU data. Unlike the physical simulation where the dynamics of the interactions do not change over time and the motion capture data where the dynamics are approximately periodic, the dynamics here change considerably over time. The middle of the trajectory is, more or less, the pick and roll itself and the behavior before and after are quite different. This poses a problem for fair comparison, as it is problematic to evaluate on the next time steps, i.e. after the PnR event, since they are quite different from our training data. Therefore in test time we feed in the first 17 time-steps to the encoder and then predict the last 8 steps.

If we train the model normally as an autoencoder, i.e. feeding in the first $N = 17$ or 25 time-steps to the encoder and having the decoder predict the same N, then this creates a large difference between training and testing setting, resulting in poor predictive performance. This is expected, as a model trained with $N = 17$ never sees the post-PnR dynamics and the encoder trained with $N = 25$ has a much easier task than one trained on $N = 17$. Therefore in order for our training to be consistent with our testing, we feed during training the first 17 steps to the encoder and predict all 25 with the decoder.

We used a CNN encoder and RNN decoder with two edge types to have comparable capacity to the full graph model. If we “hard code” one edge type to represent “non-edge” then our model learns the full graph as all players are highly connected. We also experimented with 10 and 20 edge types which did not perform as well on validation data, probably due to over-fitting.