Supplementary Material: Graph Networks as Learnable Physics Engines for Inference and Control

Alvaro Sanchez-Gonzalez [1] **Nicolas Heess** [1] **Jost Tobias Springenberg** [1] **Josh Merel** [1] **Martin Riedmiller** [1] **Raia Hadsell** [1] **Peter Battaglia** [1]

A. Summary of prediction and control videos

Table A.1. Representative trajectory prediction videos. Each shows several rollouts from different initial states for a single model trained on random control inputs. The labels encode the videos' contents: [**P**rediction/**C**ontrol].[**F**ixed/**P**arameterized/System **ID**].[(System abbreviation)]

	Fixed	Parametrized	System ID
Pendulum	link-P.F.Pe	link-P.P.Pe	link-P.I.Pe
Cartpole	link-P.F.Ca	link-P.P.Ca	link-P.I.Ca
Acrobot	link-P.F.Ac	-	-
Swimmer6	link-P.F.S6	-	-
(eval. DDPG)	link-P.F.S6(D)	-	-
SwimmerN	link-P.F.SN	link-P.P.S6	link-P.I.S6
(zero-shot)	link-P.F.SN(Z)	-	-
Cheetah	link-P.F.Ch	link-P.P.Ch	link-P.I.Ch
Walker2d	link-P.F.Wa	-	-
JACO	link-P.F.JA	link-P.P.JA	link-P.I.JA
Multiple systems	link-P.F.MS	link-P.P.MS	-
(with cheetah)	link-P.F.MC	-	-
Real JACO	link-P.F.JR	-	-

Table A.2. Representative control trajectory videos. Each shows several MPC trajectories from different initial states for a single trained model. The labels encode the videos' contents: [**P**rediction/**C**ontrol].[**F**ixed/**P**arameterized/System **ID**].[(System abbreviation)]

	Fixed	Parametrized	System ID
Pendulum (balance)	link-C.F.Pe	link-C.P.Pe	link-C.I.Pe
Cartpole (balance)	link-C.F.Ca	link-C.P.Ca	link-C.I.Ca
Acrobot (swing up)	link-C.F.Ac	-	-
Swimmer6 (reach)	link-C.F.S6	-	-
SwimmerN (reach)	link-C.F.SN	link-C.P.SN	link-C.I.SN
" baseline	link-C.F.SN(b)	-	-
Cheetah (move)	link-C.F.Ch(m)	link-C.P.Ch	link-C.I.Ch
Cheetah (k rewards)	link-C.F.Ch(k)	-	-
Walker2d (k rewards)	link-C.F.Wa(k)	-	-
JACO (imitate pose)	link-C.F.JA(o)	link-C.P.JA(o)	link-C.I.JA(o)
JACO (imitate palm)	link-C.F.JA(a)	link-C.P.JA(a)	link-C.I.JA(a)
Multiple systems	link-C.F.MS	link-C.P.MS	-

B. Description of the simulated environments

Name (Timestep)	Number of bodies (inc. world)	Generalized coordinates	Actions	Random parametrization[a] (relative range of variation, uniformly sampled)
Pendulum (20 ms)	2	Total: 1 1: angle of pendulum	1: rotation torque at axis	Length (0.2-1) Mass (0.5-3)
Cartpole (10 ms)	3	Total: 2 1: horizontal position of cart 1: angle of pole	1: horizontal force to cart	Mass of cart (0.2-2) Length of pole (0.3-1) Thickness (0.4-2.2) of pole
Acrobot (10 ms)	3	Total: 2 2: angle of each of the links angle of pole	1: rotation force between the links	N/A
SwimmerN (20 ms)	N+1	Total: N+2 2: 2-d position of head 1: angle of head N-1: angle of rest of links	N-1: rotation force between the links	Number of links (3 to 9 links) Individual lengths of links (0.3-2) Thickness (0.5-5)
Cheetah (10 ms)	8	Total: 9 2: 2-d position of torso 1: angle of torso 6: thighs, shins and feet angles	6: rotation force at thighs, shins and feet	Base angles (-0.1 to 0.1 rad) Individual lengths of bodies (0.5-2 approx.) Thickness (0.5-2)
Walker2d (2.5 ms)	8	Total: 9 2: 2-d position of torso 1: angle of torso 6: thighs, leg and feet angles	6: rotation at hips, knees and ankles	N/A
Jaco (100 ms)	10	Total: 9 3: angles of coarse joints 3: angles of fine joints 3: angles of fingers	9: velocity target at each joint	Individual body masses (0.5-1.5) Individual motor gears (0.5-1.5).

[a]Density of bodies is kept constant for any changes in size.

C. System data

C.1. Random control

Unless otherwise indicated, we applied random control inputs to the system to generate the training data. The control sequences were randomly selected time steps from spline interpolations of randomly generated values (see SM Figure C.1). A video of the resulting random system trajectories is here: Video.

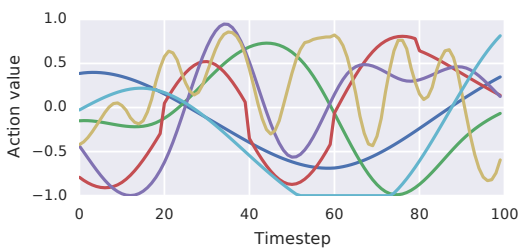

Figure C.1. Sample random sequences obtained from the same distribution than that used to generate random system data to train the models. Sample trajectory video: Video.

C.2. Datasets

For each of the individual fixed systems, we generated 10000 100-step sequences corresponding to about 10^6 supervised training examples. Additionally, we generated 1000 sequences for validation, and 1000 sequences for testing purposes.

In the case of the parametrized environments, we generated 20000 100-step sequences corresponding to about $2 \cdot 10^6$ supervised training examples. Additionally, we generated 5000 sequences for validation, and 5000 sequences for testing purposes.

Models trained on multiple environments made use of the corresponding datasets mixed within each batch in equal proportion.

C.3. Real JACO

The real JACO data was obtained under human control during a stacking task. It consisted of 2000 (train:1800, valid:100, test:100) 100-step (timestep 40 ms) trajectories. The instantaneous state of the system was represented in this case by proprioceptive information consisting of joint angles (cosine and sine) and joint velocities for each connected body in the JACO arm, replacing the 13 variables in the dynamic graph.

As the Real JACO observations correspond to the generalized coordinates of the simulated JACO Mujoco model, we use the simulated JACO to render the Real JACO trajectories throughout the paper.

D. Implementation of the models

D.1. Framework

Algorithms were implemented using TensorFlow and Sonnet. We used custom implementations of the graph networks (GNs) as described in the main text.

D.2. Graph network architectures

Standard sizes and output sizes for the GNs used are:

- Edge MLP: 2 or 3 hidden layers. 256 to 512 hidden cells per layer.
- Node and Global MLP: 2 hidden layers. 128 to 256 hidden cells per layer.
- Updated edge, node and global size: 128
- (Recurrent models) Node, global and edge size for state graph: 20
- (Parameter inference) Node, global and edge size for abstract static graph: 10

All internal MLPs used layer-wise ReLU activation functions, except for output layers.

D.3. Data normalization

The two-layer GN core is wrapped by input and output normalization blocks. The input normalization performs linear transformations to produce a zero-mean, unit-variance distributions for each of the global, node and edge features. It is worth noting that for node/edge features, the same transformation is applied to all nodes/edges in the graph, without having specific normalizer parameters for different bodies/edges in the graph. This allows to reuse the same normalizer parameters regardless of the number and type of nodes/edges in the graph. This input normalization is also applied to the observed dynamic graph in the parameter inference network.

Similarly, inverse normalization is applied to the output nodes of the forward model, to guarantee that the network only needs to output nodes with zero-mean and unit-variance.

No normalization is applied to the inferred static graph (from the system identification model), in the output of the parameter inference network, nor the input forward prediction network, as in this case the static graph is already represented in a latent feature space.

Algorithm D.1 Forward prediction algorithm.

Input: trained GNs GN_1, GN_2 and normalizers Norm_{in}, Norm_{out}.
Input: dynamic state $\mathbf{x}^{t_0}$ and actions applied $\mathbf{x}^{t_0}$ to a system at the current timestep.
Input: system parameters $\mathbf{p}$
Build static graph G_s using $\mathbf{p}$
Build input dynamic nodes $N_d^{t_0}$ using $\mathbf{x}^{t_0}$
Build input dynamic edges $E_d^{t_0}$ using $\mathbf{a}^{t_0}$
Build input dynamic graph G_d using $N_d^{t_0}$ and $E_d^{t_0}$
Build input graph $G_i = \text{concat}(G_s, G_d)$
Obtain normalized input graph $G_i^n = \text{Norm}_{in}(G_i)$
Obtain graph after the first GN: $G^{'} = \text{GN}_1(G_i^n)$
Obtain normalized predicted delta dynamic graph: $G^* = \text{GN}_2(\text{concat}(G_i^n, G^{'}))$
Obtain normalized predicted delta dynamic nodes: $\Delta N_d^n = G^*$.nodes
Obtain predicted delta dynamic nodes: $\Delta N_d = \text{Norm}_{out}^{-1}(\Delta N_d^n)$
Obtain next dynamic nodes $N_d^{t_0+1}$ by updating $N_d^{t_0}$ with ΔN_d
Extract next dynamic state $\mathbf{x}^{t_0+1}$ from $N_d^{t_0+1}$
Output: next system state $\mathbf{x}^{t_0+1}$

Algorithm D.2 Forward prediction with System ID.

Input: trained parameter inference recurrent GN GN_p.
Input: trained GNs and normalizers from Algorithm D.1.
Input: dynamic state $\mathbf{x}^{t_0}$ and actions applied $\mathbf{x}^{t_0}$ to a parametrized system at the current timestep.
Input: a 20-step sequence of observed dynamic states x^{seq} and actions x^{seq} for same instance of the system.
Build dynamic graph sequence G_d^{seq} using x_i^{seq} and a_i^{seq}
Obtain empty graph hidden state G_h.
for each graph G_d^t in G_d^{seq} **do**
  $\text{G}_o, \text{G}_\text{h} = \text{GN}_p(\text{Norm}_{in}(G_d^t), \text{G}_\text{h})$,
end for
Assign $\text{G}_{ID} = \text{G}_o$
Use G_{ID} instead of G_s in Algorithm D.1 to obtain $\mathbf{x}^{t_0+1}$ from $\mathbf{x}^{t_0}$ and $\mathbf{x}^{t_0}$
Output: next system state $\mathbf{x}^{t_0+1}$

D.4. System invariance

When training individual models for systems with translation invariance (Swimmer, Cheetah and Walker2d), we always re-centered the system around 0 before the prediction, and moved it back to its initial location after the prediction. This procedure was not applied when multiple systems were trained together.

D.5. Prediction of dynamic state change

Instead of using the one-step model to predict the absolute dynamic state, we used it to predict the change in dynamic state, which was then used to update the input dynamic state. For the position, linear velocity, and angular velocity, we updated the input by simply adding their corresponding predicted changes. For orientation, where the output represents the rotation quaternion between the input orientation and the next orientation (forced to have unit norm), we computed the update using the Hamilton product.

D.6. Forward prediction algorithms

D.6.1. One-step prediction

Our forward model takes the system parameters, the system state and a set of actions, to produce the next system state as explained in SM Algorithm D.1.

D.6.2. One-step prediction with System ID

Algorithm D.3 One step of the training algorithm

Before training: initialize weights of GNs GN_1, GN_2 and accumulators of normalizers Norm_{in}, Norm_{out}.
Input: batch of dynamic states of the system $\{\mathbf{x}^{t_0}\}$ and actions applied $\{\mathbf{a}^{t_0}\}$ at the current timestep
Input: batch of dynamic states of the system at the next timestep $\{\mathbf{x}^{t_0+1}\}$
Input: batch of system parameters $\{\mathbf{p}_i\}$
for each example in batch **do**
  Build static graph G_s using $\mathbf{p}_i$
  Build input dynamic nodes $N_d^{t_0}$ using $\mathbf{x}^{t_0}$
  Build input dynamic edges $E_d^{t_0}$ using $\mathbf{a}^{t_0}$
  Build output dynamic nodes $N_d^{t_0+1}$ using $\mathbf{x}^{t_0+1}$
  Add noise to input dynamic nodes $N_d^{t_0}$
  Build input dynamic graph G_d using $N_d^{t_0}$ and $E_d^{t_0}$
  Build input graph $G_i = \text{concat}(G_s, G_d)$
  Obtain target delta dynamic nodes $\Delta N'_d$ from $N_d^{t_0+1}$ and $N_d^{t_0}$
  Update Norm_{in} using G_i
  Update Norm_{out} using ΔN_d
  Obtain normalized input graph $G_i^n = \text{Norm}_{in}(G_i)$
  Obtain normalized target nodes: $\Delta N_d^{n'} = \text{Norm}_{out}(\Delta N'_d)$
  Obtain normalized predicted delta dynamic nodes: $\Delta N_d^n = \text{GN}_2(\text{concat}(G_i^n, \text{GN}_1(G_i^n))).\text{nodes}$
  Calculate dynamics prediction loss between ΔN_d^n and $\Delta N_d^{n'}$.
end for
Update weights of GN_1, GN_2 using Adam optimizer on the total loss with gradient clipping.

For the System ID foward predictions the model takes a system state and a set of actions for a specific instance of a parametrized system, together with a sequence of observed system states and actions for a for the same system instance. The observed sequence is used to identify the system and then produce the next system state as described in Algorithm D.2.

In the case of rollout predictions, the System ID is only performed once, on the provided observed sequence, using the same graph for all of the one-step predictions required to generate the trajectory.

D.7. Training algorithms

D.7.1. One-step

We trained the one-step forward model in a supervised manner using algorithm D.3. Part of the training required finding mean and variance parameters for the input and output normalization, which we did online by accumulating information (count, sum and squared sum) about the distributions of the input edge/node/global features, and the distributions of the change in the dynamic states of the nodes, and using that information to estimate the mean and standard deviation of each of the features.

Due to the fact that our representation of the instantaneous state of the bodies is compatible with configurations where the joint constraints are not satisfied, we need to train our model to always produced outputs within the manifold of configurations allowed by the joints. This was achieved by adding random normal noise (magnitude set as a hyper-parameter) to the nodes of the input dynamic graph during training. As a result, the model not only learns to make dynamic predictions, but to put back together systems that are slightly dislocated, which is key to achieve small rollout errors.

D.7.2. Abstract parameter inference

The training of the parameter inference recurrent GN is performed as described in Algorithm D.4. The recurrent GN and the dynamics GN are trained together end-to-end by sampling a random 20-step sequence for the former, and a random supervised example for the latter from 100-step graph sequences, with a single loss based on the prediction error for the supervised example. This separation between the sequence at the supervised sample, encourages the recurrent GN to truly extract abstract static properties that are independent from the specific 20-step trajectory, but useful for making dynamics predictions under any condition.

Algorithm D.4 End-to-end training algorithm for System ID.

Before training: initialize weights of parameter inference recurrent GN GN_p, as well as weights from Algorithm D.3.
Input: a batch of 100-step sequences with dynamic states $\{x_i^{\text{seq}}\}$ and actions $\{x_i^{\text{seq}}\}$
for each sequence in batch **do**
  Pick a random 20-step subsequence x_i^{subseq} and a_i^{subseq}.
  Build dynamic graph sequence G_d^{subseq} using x_i^{subseq} and a_i^{subseq}
  Obtain empty graph hidden state G_h.
  for each graph G_d^t in G_d^{subseq} **do**
    $\text{G}_o, \text{G}_\text{h} = \text{GN}_p(\text{Norm}_{in}(G_d^t), \text{G}_\text{h})$,
  end for
  Assign $\text{G}_{ID} = \text{G}_o$
  Pick a different random timestep t_0 from $\{x_i^{\text{seq}}\}$, $\{x_i^{\text{seq}}\}$
  Apply Algorithm D.3 to timestep t_0 using final G_{ID} instead G_s to obtain the dynamics prediction loss.
end for
Update weights of GN_p, GN_1, GN_2 using Adam optimizer on the total loss with gradient clipping.

D.7.3. Recurrent one-step predictions

The one-step prediction recurrent model, used for the Real JACO predictions, is trained from 21-step sequences using the *teacher forcing* method. The first 20 graphs in the sequence are used as input graphs, while the last 20 graphs in the sequence are used as target graphs. During training, the recurrent model is used to sequentially process the input graphs, producing at each step a predicted dynamic graph, which is stored, and a graph state, which is fed together with the next input graph in the next iteration. After processing the entire sequence, the sequence of predicted dynamic graphs and the target graphs are used together to calculate the loss.

D.7.4. Loss

We use a standard L2-norm between the normalized expected and predicted delta nodes, for the position, linear velocity, and angular velocity features. We do this for the normalized features to guarantee a balanced relative weighting between the different features. In the case of the orientation, we cannot directly calculate the L2-norm between the predicted rotation quaternion $\mathbf{q}_p$ to the expected rotation quaternion $\mathbf{q}_e$, as a quaternion $\mathbf{q}$ and $-\mathbf{q}$ represent the same orientation. Instead, we minimize the angle distance between $\mathbf{q}_p$ and $\mathbf{q}_e$ by minimizing the loss $1 - \cos^2(\mathbf{q}_e \cdot \mathbf{q}_p)$ after.

D.8. Training details

Models were trained with a batch size of 200 graphs/graph sequences, using an Adam optimizer on a single GPU. Starting learning rates were tuned at 1^{-4}. We used two different exponential decay with factor of 0.975 updated every 50000 (fast training) or 200000 (slow training) steps.

We trained our models using early stopping or asymptotic convergence based the rollout error on 20-step sequences from the validation set. Simple environments (such as individual fixed environments) would typically train using the fast training configuration for a period between less than a day to a few days, depending on the size of the environment and the size of the network. Using slow training in these cases only yields a marginal improvement. On the other hand, more complex models such as those learning multiple environments and multiple parametrized environments benefited from the slow training to achieve optimal behavior for periods of between 1-3 weeks.

E. MLP baseline architectures

For the MLP baselines, we used 5 different models (ReLU activation) spanning a large capacity range:

- 3 hidden layers, 128 hidden cells per layer
- 3 hidden layers, 512 hidden cells per layer
- 9 hidden layers, 128 hidden cells per layer

Algorithm F.1 MPC algorithm

Input: initial system state $\mathbf{x}^0$,
Input: randomly initialized sequence of actions $\{\mathbf{a}^t\}$.
Input: pretrained dynamics model M such
$\mathbf{x}^{t_0+1} = M(\mathbf{x}^{t_0}, \mathbf{a}^{t_0})$
Input: Trajectory cost function L such
$c = C(\{\mathbf{x}^t\}, \{\mathbf{a}^t\})$
for a number of iterations **do**
  $\mathbf{x}_r^0 = \mathbf{x}^0$
  for t in range(0, horizon) **do**
    $\mathbf{x}_r^{t+1} = M(\mathbf{x}_r^t, \mathbf{a}^t)$
  end for
  Calculate trajectory cost $c = C(\{\mathbf{x}_r^t\}, \{\mathbf{a}^t\})$
  Calculate gradients $\{\mathbf{g}_a^t\} = \frac{\partial c}{\partial \{\mathbf{a}^t\}}$
  Apply gradient based update to $\{\mathbf{a}^t\}$
end for
Output: optimized action sequence $\{\mathbf{a}^t\}$

- 9 hidden layers, 512 hidden cells per layer
- 5 hidden layers, 256 hidden cells per layer

The corresponding MLP replaces the 2-layer GN core, with additional layers to flatten the input graphs into feature batches, and to reconstruct the graphs at the output. Both normalization and graph update layers are still applied at graph level, in the same way that for the GN-based model.

Each of the models was trained four times using initial learning rates of 1^{-3} and 1^{-4} and learning rate decays every 50000 and 200000 steps. The model performing best on validation rollouts for each environment, out of the 20 hyperparameter combinations was chosen as the MLP baseline.

F. Control

F.1. Model-based planning algorithms

F.1.1. MPC PLANNER WITH LEARNED MODELS

We implemented MPC using our learned models as explained in SM Algorithm F.1. We applied the algorithm in a receding horizon manner by iteratively planning for a fixed horizon (see SM Table F.2), applying the first action of the sequence, and increasing the horizon by one step, reusing the shifted optimal trajectory computed in the previous iteration. We typically performed between 3 and 51 optimization iterations N from each initial state, with additional $N \cdot$ horizon iterations at the very first initial state, to *warm-up* the fully-random initial action sequence.

F.1.2. BASELINE MUJOCO-BASED PLANNER

As a baseline planning approach we used the iterative Linear-Quadratic-Gaussian (iLQG) trajectory optimization approach proposed in (Tassa et al., 2014). This method alternates between forward passes (rollouts) which integrate the dynamics forward for a current control sequence and backwards passes which consists of perturbations to the control sequence to improve upon the recursively computed objective function. Note that in the backwards pass, each local perturbation can be formulated as an optimization problem, and linear inequality constraints ensure that the resulting control trajectory does not require controls outside of the range that can be feasibly generated by the corresponding degrees of freedom in the MuJoCo model. The overall objective optimized corresponds to the total cost over J a finite horizon:

$$J(x_0, U) = \sum_{t=0}^{T-1} \ell(x_t, u_t) + \ell(x_T) \tag{1}$$

where x_0 is the initial state, u_t is the control signal (i.e. action) taken at timestep t, U is the trajectory of controls, $\ell(\cdot)$ is the cost function. We assume the dynamics are deterministic transitions $x_{t+1} = f(x_t, u_t)$.

While this iLQG planner does not work optimally when the dynamics involve complex contacts, for relatively smooth dynamics as found in the swimmer, differential dynamic programming (DDP) style approaches works well (Tassa et al., 2008). Relevant cost functions are presented in SM Section F.2.

F.2. Planning configuration

Name	Task	Planning horizon	Reward to maximize (summed for all timesteps)
Pendulum	Balance	50, 100	Negative angle between the quaternion of the pendulum and the target quaternion corresponding to the balanced position. (0 when balanced at the top, < 0 otherwise).
Cartpole	Balance	50, 100	Same as Pendulum-Balance calculated for the pole.
Acrobot	Swing up	100	Same as Pendulum-Balance summed for both acrobot links.
Swimmer	Mover towards target	100	Projection of the displacement vector of the Swimmer head from the previous timestep on the target direction, The target direction is calculated as the vector joining the head of the swimmer at the first planning timestep with the target location. The reward is shaped (0.01 contribution) with the negative squared projection on the perpendicular target direction.
Cheetah	Move forward	20	Horizontal component of the absolute velocity of the torso.
	Vertical position	20	Vertical component of the absolute position of the torso.
	Squared vertical speed	20	Squared vertical component of the absolute velocity of the torso.
	Squared angular speed	20	Squared angular velocity of the torso.
Walker2d	Move forward	20	Horizontal component of the absolute velocity of the torso.
	Vertical position	20	Vertical component of the absolute position of the torso.
	Inverse verticality	20	Same as Pendulum-Balance summed for torso, thighs and legs.
	Feet to head height	20	Summed squared vertical distance between the position of each of the feet and the height of Walker2d.
Jaco	Imitate Palm Pose	20	Negative dynamic-state loss (as described in Section D.7.4) between the position-and-orientation of the body representing the JACO palm and the target position-and-orientation .
	Imitate Full Pose	20	Same as Jaco-Imitate Palm Pose but summed across all the bodies forming JACO (see SM Section D.7.4).

F.3. Reinforcement learning agents

Our RL experiments use three base algorithms for continuous control: DDPG (Lillicrap et al., 2016), SVG(0) and SVG(N) (Heess et al., 2015). All of these algorithms find a policy π that selects an action a in a given state x by maximizing the expected discounted reward,

$$Q(\mathbf{x}, \mathbf{a}) = \mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t r(\mathbf{x}, \mathbf{a})\Big], \tag{2}$$

where $r(x, a)$ is the per-step reward and γ denotes the discount factor. Learning in all algorithms we consider occurs off-policy. That is, we continuously generate experience via the current best policy π, storing all experience (sequences of states, actions and rewards) it into a replay buffer $\mathcal{B}$, and minimize a loss defined on samples from $\mathcal{B}$ via stochastic gradient descent.

F.3.1. DDPG

The DDPG algorithm (Lillicrap et al., 2016) learns a deterministic control policy $\pi = \mu_\theta(s)$ with parameters θ and a corresponding action-value function $Q^\mu_\phi(s, a)$, with parameters ϕ. Both of these mapping are parameterized via neural networks in our experiments.

Learning proceeds via gradient descent on two objectives. The objective for learning the Q function is to minimize the one-step Bellman error using samples from a replay buffer, that is we seek to find $\arg\min_\phi L(\phi)$ by following the gradient,

$$\begin{aligned} \nabla_\phi L(\phi) &= \mathbb{E}_{(\mathbf{x}_t, \mathbf{a}_t, \mathbf{x}_{t+1}, r_t) \in \mathcal{B}} \Big[\nabla_\phi \big(Q^\mu_\phi(\mathbf{x}_t, \mathbf{a}_t) - y \big)^2 \Big], \\ \text{with } \; y &= r_t + \gamma Q^\mu_{\phi'}(\mathbf{x}_{t+1}, \mu_{\theta'}(\mathbf{x}_{t+1})) \end{aligned} \tag{3}$$

where ϕ' and θ' denote the parameters of target Q-value and policy networks, that are periodically copied from the current parameters, this is common practice in RL to stabilize training (we update the target networks every 1000 gradient steps). The objective for learning the policy is performed by searching for an action that obtains maximum value, as judged by the learned Q-function. That is we find $\arg\min_\theta L(\theta)$ by following the deterministic policy gradient (Lillicrap et al., 2016),

$$\nabla_\theta L^{\text{DPG}}(\theta) = \mathbb{E}_{\mathbf{x}_t \in \mathcal{B}} \Big[- \nabla_\theta Q^\mu_\theta(\mathbf{x}_t, \mu_\theta(\mathbf{x}_t)) \Big]. \tag{4}$$

F.3.2. SVG

For our experiments with the familiy of Stochastic Value Gradient (SVG) (Heess et al., 2015) algorithms we considered two-variants a model-free baseline SVG(0) that optimizes a stochastic policy based on a learned Q-function as well as a model-based version SVG(N) (using our Graph Net model) that unrolls the system dynamics for N-steps.

SVG(0) In the model-free variant learning proceeds similarly to the DDPG algorithm. We learn both, a parametric Q-value estimator as well as a (now stochastic) policy $\pi_\theta(\mathbf{a}|\mathbf{x})$ from which actions can be sampled. In our implementation learning of the Q-function is performed by following the gradient from Equation (3), with $\mu(x)$ replaced by samples $\mathbf{a} \sim \pi_\theta(\mathbf{a}|\mathbf{x})$.

For the policy learning step we can learn via a stochastic analogue of the deterministic policy gradient from Equation (4), the so called stochastic value gradient, which reads

$$\nabla_\theta L^{\text{SVG}}(\theta) = -\nabla_\theta \mathbb{E}_{\substack{\mathbf{x}_t \in \mathcal{B} \\ \mathbf{a} \sim \pi_\theta(\mathbf{a}|\mathbf{x}_t)}} \Big[Q^\pi_\theta(\mathbf{x}_t, \cdot) \Big]. \tag{5}$$

For a Gaussian policy (as used in this paper) the gradient of this expectation can be calculated via the reparameterization trick (Kingma & Welling, 2014; Rezende et al., 2014).

SVG(N) For the model based version we used a variation of SVG(N) that employs an action-value function – instead of the value function estimator used in the original paper. This allowed us to directly compare the performance of a SVG(0) agent, which is model free, with SVG(1) which calculates policy gradients using a one-step model based horizon.

In particular, similar to Equation (5), we obtain the model based policy gradient as

$$\nabla_\theta L^{\text{SVG(N)}}(\theta) = -\nabla_\theta \mathbb{E}_{\substack{\mathbf{x}_t \in \mathcal{B} \\ \mathbf{a}_t \sim \pi_\theta(\mathbf{a}|\mathbf{x}_t) \\ \mathbf{a}_{t+1} \sim \pi_\theta(\mathbf{a}|\mathbf{x}_{t+1})}} \Big[r_t(\mathbf{x}_t, \mathbf{a}_t) + \gamma Q^\pi_\theta(\mathbf{x}_{t+1}, \mathbf{a}_t) \mid \mathbf{x}_{t+1} = g(\mathbf{x}_t, \mathbf{a}_t) \Big], \tag{6}$$

where g denotes the dynamics, as predicted by the GN and the gradient can, again, be computed via reparameterization (we refer to Heess et al. (2015) for a detailed discussion).

We experimented with SVG(1) on the swimmer domain with six links (Swimmer 6). Since in this case, the goal for the GN is to predict environment observations (as opposed to the full state for each body), we constructed a graph from the observations and actions obtained from the environment. SM Figure H.3 describes the observations and actions and shows how they were transformed into a graph.

G. Mujoco variables included in the graph conversion

G.1. Dynamic graph

We retrieved the the absolute position, orientation, linear and angular velocities for each body:

- **Global: None**
- **Nodes:** (for each body)
  Absolute body position (3 vars): mjData.xpos
  Absolute body quaternion orientation position (4 vars): mjData.xquat
  Absolute linear and angular velocity (6 vars): mj_objectVelocity (mjOBJ_XBODY, flg_local=False)
- **Edges:** (for each joint) Magnitude of action at joint: mjData.ctrl (0, if not applicable).

G.2. Static graph

We performed an exhaustive selection of global, body, and joint static properties from mjModel:

- **Global:** mjModel.opt.{timestep, gravity, wind, magnetic, density, viscosity, impratio, o_margin, o_solref, o_solimp, collision_type (one-hot), enableflags (bit array), disableflags (bit array)}.
- **Nodes:** (for each body) mjModel.body_{mass, pos, quat, inertia, ipos, iquat}.
- **Edges:** (for each joint)
  Direction of edge (1: parent-to-child, -1: child-to-parent).
  Motorized flag (1: if motorized, 0 otherwise).
  Joint properties: mjModel.jnt_{type (one-hot), axis, pos, solimp, solref, stiffness, limited, range, margin}.
  Actuator properties: mjModel.opt.actuator_{biastype (one-hot), biasprm, cranklength, ctrllimited, ctrlrange, dyntype (one-hot), dynprm, forcelimited, forcerange, gaintype (one-hot), gainprm, gear, invweight0, length0, lengthrange}.

Most of these properties are constant for all environments use, however, they are kept for completeness. While we do not include geom properties such as size, density or shape, this information should be partially encoded in the inertia tensor together with the mass.

H. Supplementary figures

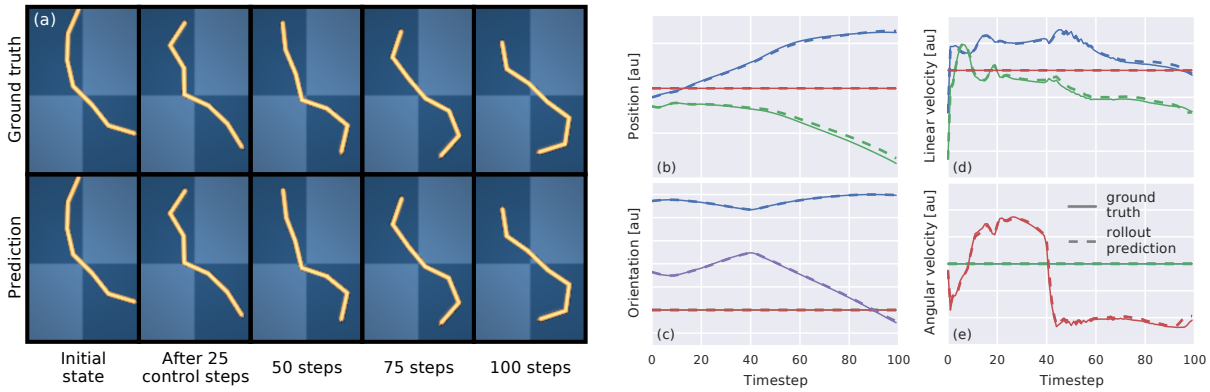

Figure H.1. Model trained on Swimmer6 trajectories under random control evaluated on a trajectory generated by a DDPG agent. Trajectories are also available in video [link-P.F.S6(D)]. (Left) Key-frames comparing the ground truth and predicted sequence within a 100 step trajectory. (Right) Full state sequence prediction for the third link of the Swimmer, consisting of Cartesian position (3 vars), quaternion orientation (4 vars), Cartesian linear velocity (3 vars) and Cartesian angular velocity (3 vars). The full prediction contains such 13 variables for each of the links, that is 78 variables.

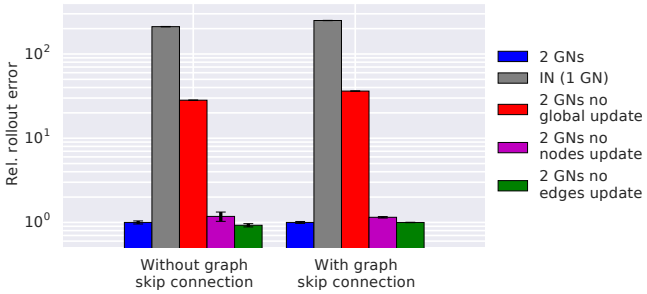

Figure H.2. Ablation study of the architecture using the rollout error over 20 step test sequences in Swimmer6 to evaluate relative performance. The performance of the architecture used in this work (a sequence of two GNs, blue) is compared to: an Interaction Network (IN) (Battaglia et al., 2016), which is equivalent in this case to a single GN (grey), and a sequence of GNs where the first GN is not allowed to update either the global (red), the nodes (purple) or the edges(green) of the output graph. Results are shown both for a purely sequential connection of the GNs, and for a model with a graph skip connection, where the output graph of the first GN, is concatenated to the input graph, before feeding it into the second GN. The results show that the performance of the double GN is far superior than that of the equivalent IN. They also show that the global update performed by the GN is necessary for the model to perform well. We hypothesize this is due to the long range dependencies within the graph that exist within swimmer, and the ability of the global update to quickly propagate such dependencies across the entire graph. Similar results may have been obtained without global updates by using a deeper stack of GNs to allow information to flow across the entire graph. Each model was trained from three different seeds. The figure depicts the mean, and the standard deviation of the asymptotic performance of the three seeds.

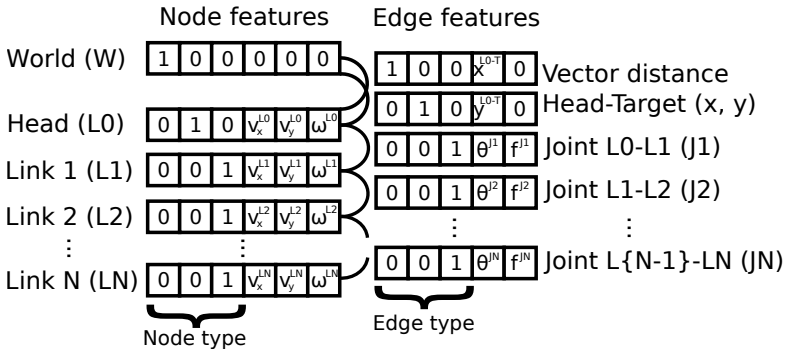

Figure H.3. Arrangement as a graph of the default 25-feature observation and 5 actions provided in the Swimmer 6 task from the DeepMind Control Suite (Tassa et al., 2018). The observation consists of: *(to_target)* the distance between the head and the target projected in the axis of the head ($x^{\text{L0-T}}$, $y^{\text{L0-T}}$), *(joints)* the angle of each joint JN between adjacent swimmer links LN-1 and LN (θ^{JN}) and *(body_velocities)* the linear and angular velocity of each link LN projected in its own axis (v_x^{LN}, v_y^{LN}, ω^{LN}). The actions consists of the force applied to each of the joints (f^{JN}) connecting the links. Because our graphs are directed, all of the edges were duplicated, with an additional -1, 1 feature indicating the direction.