Robot Learning by Collaborative Network Training: A Self-Supervised Method using Ranking

Mason Bretan
Samsung Research America
Mountain View, CA
mason.bretan@samsung.com

Sageev Oore
Dalhousie University & Vector Institute
Halifax, Canada
sageev@vectorinstitute.ai

Siddharth Sanan
Samsung Research America
Mountain View, CA
s.sanan@samsung.com

Larry Heck
Samsung Research America
Mountain View, CA
larry.h@samsung.com

ABSTRACT

We introduce *Collaborative Network Training* – a self-supervised method for training neural networks with aims of: 1) enabling task objective functions that are not directly differentiable w.r.t. the network output; 2) generating continuous-space actions; 3) more direct optimization for achieving a desired task; 4) learning parameters when a process for measuring performance is available, but labeled data is unavailable. The procedure involves three randomly initialized independent networks that use ranking to train one another on a single task. The method incorporates qualities from ensemble and reinforcement learning as well as gradient free optimization methods such as Nelder-Mead. We evaluate the method against various baselines using a variety of robotics-related tasks including inverse kinematics, controls, and planning in both simulated and real-world environments.

KEYWORDS

Robot learning; collaborative network training; controls

ACM Reference Format:
Mason Bretan, Sageev Oore, Siddharth Sanan, and Larry Heck. 2019. Robot Learning by Collaborative Network Training: A Self-Supervised Method using Ranking. In *Proc. of the 18th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2019), Montreal, Canada, May 13–17, 2019,* IFAAMAS, 8 pages.

1 INTRODUCTION

Frequently the loss function chosen to train a neural network is not the same function that is used to measure the network's ability to perform the task it will ultimately do. This is in part due to the requirement that the loss function must be differentiable w.r.t. the output of the network. Often times this discrepancy can lead to unfavorable results because the proxy loss function used to train the network does not optimize the parameters directly for performing the specific task.

For example, consider a network tasked with generating a sequence of joint angles for a robotic manipulator enabling the end-effector to reach a specified location at a given time. One may create ground truth examples based on a conventional inverse kinematics (IK) and planning algorithm and minimize the difference between the model generated angles and this truth. While the network may converge on a local optimum allowing it to produce angles similar to that of the IK algorithm, it is quite possible it will neglect information essential to successfully performing the actual task. In other words, minimizing a loss function in joint space does not necessarily translate to success in the task space.

We introduce *Collaborative Network Training* – a new method of training neural networks with aims of enabling task objective functions that are not directly differentiable w.r.t. the network output and learning parameters when a process for measuring performance is available, but labeled data is unavailable. Our proposed method allows for direct optimization for achieving a desired task and is particularly suitable for generating continuous-space actions. The method utilizes a trio of networks that leverage an arbitrary task performance or "critic" function to rank the output of each network from worst to best for a given input. During training each network is treated as an independent agent with a unique set of parameters and each applies the information gleaned from the trio to direct its own learning and exploration. Eventually the trio converges to a single local optimum and only one network is used during inference.

We begin by describing the methodology and then show performance using experiments based on common problems in robotics—a domain which poses challenges for which our method appears to be particularly well suited. This includes inverse kinematics, torque control of a nonlinear dynamic system, and trajectory planning requiring temporal precision at the millisecond level. We show that each network's output and rank is enough to successfully train a network to perform various tasks. Success is measured by comparing the performance to other model-based strategies using supervised training with labeled data, state of the art reinforcement learning methods for deterministic continuous decision making, and more standard iterative optimization techniques such as Jacobian-based IK.

2 RELATED WORK

Sometimes the difference between a "training loss" function and a "task evaluation" function can be advantageous and may result in increased robustness and generalization overall. For example,

Proc. of the 18th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2019), N. Agmon, M. E. Taylor, E. Elkind, M. Veloso (eds.), May 13–17, 2019, Montreal, Canada. © 2019 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

in classification, test set accuracy may be used to evaluate performance, while minimizing either a softmax with negative examples or a cross-entropy loss during training can help emphasize the separation and boundaries among classes by pushing them farther apart ([8, 11, 21]). By exaggerating the boundaries in this manner the network generalizes better to data outside the training set. Classification is a scenario in which there are clearly benefits to having a training loss function that is different than the task evaluation function.

Though there can be benefits to using a provisional loss function for training, sometimes it is desirable to optimize the network parameters more directly to model the distributions seen in the data. For example, generative networks that attempt to model probabilistic distributions of real world data may minimize a loss function describing the difference between the true distribution $p(x)$ and model distribution $\hat{p}(x)$. Effective features can be gleaned from parameters of a network trained to do this ([16]). In addition, the network can produce outputs that resemble the original real-world distributions ([7, 19]). The network outputs can be further tuned with generative adversarial network (GAN) training methods in which the generator is specifically optimized to make the resulting model distribution indistinguishable from the true distribution by incorporating an additional discriminator network ([6]).

In these training methods there must exist data representing the true distributions and a differentiable metric suitable for evaluating the quality of the networks' outputs. For example, autoencoding architectures may measure the euclidean or cosine distance between input and output and update the parameters according to one of these losses ([20]). For discrete predictive tasks such as generating letters, musical notes, or pixels, a softmax or cross-entropy describing a conditional distribution for every possible output is often used ([17, 19]). In GANs, the loss measured by the discriminator portion of the network is used to update the parameters of the generator portion ([6]).

However, sometimes measuring the performance of a network is primarily determined by the ability of its outputs to lead to the successful completion of a task rather than modeling an existing distribution. Recall the model-based IK problem described earlier and that a model trained in joint space will not necessarily result in similar performance in the task space. Reinforcement learning (RL) methods address this problem by optimizing for the specific task and generating a conditional probability describing the likelihood of a future reward for each action in a set of discrete actions ([18]).

In this RL scenario there exists a known metric for directly measuring success on the task, but it is not always clear how to efficiently define it in a manner such that it is differentiable w.r.t the network output. By breaking the output into a set of actions, the Bellman equation enables a differentiable loss function and the network learns a policy to maximize performance ([13]). Learning effective policies in this manner is feasible when the number of possible actions is relatively small, but inadequate for high-dimensional and continuous action spaces. An actor-critic method, Deep Deterministic Policy Gradient (DDPG), was introduced in ([12]) and the actor is a policy function mapping states to continuous actions. Unlike DDPG, our method does not need to learn the parameters of a critic network in parallel to the actor.

In our method of collaborative training, three networks work together to find the optimal weights for a single network on a given task. There are many examples of systems that leverage the information within a group of models as part of the inference or architecture, and where the training might be adjusted accordingly. Ensemble learning typically requires multiple learners that are trained to solve the same problem. During inference each model contributes to a collection of hypotheses which are used to make the final prediction ([22]). Conversely, the process of federated learning is to aggregate the updates made to independent decentralized models to compute a new global model ([10]). The centralized update does not rely on data, but rather the updates each decentralized model made to boost their own performance. Our method is different from both of these in that the networks rely on a signal describing their own performance relative to the others. Moreover, once the training is complete, we only need to use one of the sets of learned weights.

The procedure for collaborative network training also draws inspiration from non-gradient optimization methods including Nelder-Mead ([14]) and annealing ([9]). Like these search methods our procedure applies a heuristic which is used to direct the update of each network. Furthermore, the concepts of ordering and reflection in Nelder-Mead can also be seen in our method and help to prevent the networks from converging on an undesirable local optimum.

3 COLLABORATIVE NETWORK TRAINING

Collaborative training leverages the fact that given a collection of randomly initialized networks each network will produce unique output values for a particular input. The procedure relies on the ability to describe the outputs of each network as being simply better or worse than the outputs of the other networks within the group. Thus, a necessary component is a critic or task evaluation function that measures the performance of each network in a continuous space in order to rank the networks. We use a variety of ranking functions in our experiments.

The main premise for collaborative training is that if each network is updated in order to behave more similarly to better performing networks for a single input example, then over time and across many examples the collection will achieve a local optimum for the task. However, simply updating each network in the direction of the best performing network leads to quick convergence on sub-optimal parameters that will not perform well. In order to prevent this, the worst performing network is updated in the direction determined by its *reflection* of the best performing network.

This concept is related to exploitation and exploration processes in reinforcement learning as well as Nelder-Mead reflection. Given a collection of three networks, the middle network can be thought of as exploiting the information available within the trio, while the worst network explores new space in a direction informed by the best network.

The procedure, summarized in Figure 1, works as follows: $\mathbb{N}$ is the set containing $\{1, 2, 3, ..., N\}$ where N is the total number of networks. The parameters of each network, θ_n, are randomly initialized for $n \in \mathbb{N}$. In all of our experiments each network within an ensemble have identical architectures. Additionally, we use $N =$

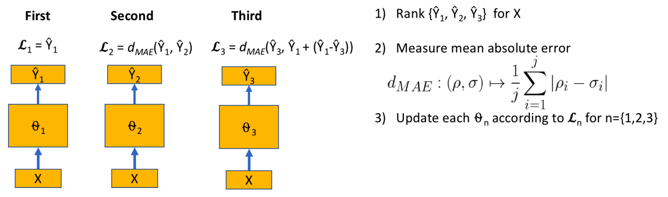
First
Second
Third
$\mathcal{L}_1 = \hat{Y}_1$
$\mathcal{L}_2 = d_{MAE}(\hat{Y}_1, \hat{Y}_2)$
$\mathcal{L}_3 = d_{MAE}(\hat{Y}_3, \hat{Y}_1 + (\hat{Y}_1 - \hat{Y}_3))$
$\hat{Y}_1$
$\hat{Y}_2$
$\hat{Y}_3$
θ_1
θ_2
θ_3
X
X
X
1) Rank $\{\hat{Y}_1, \hat{Y}_2, \hat{Y}_3\}$ for X
2) Measure mean absolute error
$d_{MAE} : (\rho, \sigma) \mapsto \frac{1}{j} \sum_{i=1}^{j} |\rho_i - \sigma_i|$
3) Update each θ_n according to $\mathcal{L}_n$ for n={1,2,3}

Figure 1: The collaborative training procedure. Three networks are ranked on their performance for a given input, x, and updated according to losses determined by their rankings. In this example, the Second ranked network is optimized to produce output closer to that of the First network, the First network remains unchanged, and the Third network is optimized to move away in a new direction informed by the First.

3 and preliminary testing did not find an increase in performance for higher N (though this needs further validation).

For a given training input, $x \in X$, where X is the set of all training inputs, the outputs of each network, $\hat{y}_n$, are sorted according to a ranking function $f : (x, \hat{y}_n) \mapsto \mu_n$ where μ_n describes the performance of $\hat{y}_n$ given x. Thus, the best performing network is θ_{w_1} where $w_1 = \underset{n}{\arg max}(\mu_n)$ and the worst performing network is θ_{w_N} where $w_N = \underset{n}{\arg min}(\mu_n)$.

We next define the mean absolute difference (MAE) between the outputs of two networks:

$$d_{MAE} : (\rho, \sigma) \mapsto \frac{1}{j} \sum_{i=1}^{j} |\rho_i - \sigma_i| \tag{1}$$

where (ρ, σ) are the output vectors of any pair of networks in our ensemble with each vector of length j.

The loss for a network in the ensemble, $\mathfrak{L}_n$, is determined by its ranking. For each of the non-worst networks, $\mathbb{N} \setminus w_N$, the loss is found by computing d_{MAE} between adjacently ranked networks That is,

$$\mathfrak{L}_{w_{n \in \mathbb{N} \setminus w_1}} = d_{MAE}(\hat{y}_{w_n}, \hat{y}_{w_{n-1}}) \tag{2}$$

where $\mathfrak{L}_{w_n}$ is differentiable w.r.t to θ_{w_n}. The goal is for each network to be trained so that a given network, θ_{w_n}, produces an output for x that more closely resembles an output from a better performing network $\theta_{w_{n-1}}$.

Finally, the loss for the worst performing network, $\mathfrak{L}_{w_N}$ is computed using

$$\mathfrak{L}_{w_N} = d_{MAE}(\hat{y}_{w_N}, \hat{y}_{w_1} + (\hat{y}_{w_1} - \hat{y}_{w_N})) \tag{3}$$

To summarize, for a given input, the best performing network remains unchanged, the second best network is updated in the direction of the best performing network, and the worst performing network is updated in the direction described by the reflection of the worst past the best.

Using these losses the collection can be trained in batch as long as each sample is evaluated individually. The same learning rate is used for each network and the parameters are optimized using AdamOptimizer. Training is stopped when the average performance across all networks no longer improves. A single network from the trio is chosen at random to use during inference. While using the averaged output across all networks may improve performance in some cases, in order to maintain fairness and more carefully evaluate and compare performance to alternative methods we do not leverage the ensemble during inference in our experiments. Thus, it is important to remember that three times as many parameters are used during training as there are during inference. That is, parameters are effectively added during training and then removed.

Relation to Nelder-Mead. In the Nelder-Mead optimization method, a function $f(\mathbf{v})$ is optimized by using a collection of N vectors $\mathbf{w}_1, \ldots, \mathbf{w}_N$, each also of dimensionality of N. Each of the $\mathbf{w}$'s is a candidate solution, and together they form a simplex that crawls through the parameter space in an amoeba-like way, essentially the worst (least-optimal) vertex pushing forward in the direction of the best one. For obvious reasons, this method does not naturally scale well to enormous parameter spaces.

However, we take inspiration from the Nelder-Mead optimization idea that mid-level solutions should move towards better ones, and the worst solutions might as well explore candidates for improving on the best solutions. Recognizing the non-linearity of the error surface and the high-dimensionality of the parameter space, we wish to have a model that "moves towards" another model but without degrading its performance on the target task; moving the parameters of model A towards the parameter values of model B—as would be done in a direct implementation of Nelder-Mead—does not make sense in this context. Furthermore, we wish to do so in a way that we do not require—as Nelder-Mead does—a simplex of the same dimensionality as that of the parameter space in which we are optimizing (having the same number of networks as parameters would be infeasible). The natural solution to this, then, is to have a small set of models, where the notion of "moving toward" is achieved by training one model on the output of another model.

4 EXPERIMENTS

4.1 A Direct Comparison to Supervised Training with Labeled Data

In our first experiments we compare collaborative training with a typical supervised method that uses labeled data. The two tasks we use are modeling the Griewank function and a forward kinematic calculation for a particular joint chain. The purpose of these initial experiments is to compare the proposed method to typical supervised training under conditions that are ideal for supervision, namely, when the objective function used for training is identical to the function used for evaluation and when the number of possible solutions to $f(x)$ is only one for each input x (zero redundancy). In both tasks, the supervised network, ϕ, and the set of three collaborative networks, $\mathbb{C} = \{\theta_1, \theta_2, \theta_3\}$ use identical architectures and see the same training samples.

Note that when evaluating our method against supervised training, getting results that are roughly comparable would be successful; a system that uses full label and gradient information should be an upper bound on how well a self-supervised system could perform. A more direct comparison will be against completely self-supervised methods and supervised methods designed to address redundant solutions (such as in IK), which we will show in the subsequent sections.

Griewank Function We train a network to approximate a widely used function for optimization testing, the Griewank Function. This function is often used because of its many local minima and maxima. In this experiment each network uses a feed-forward architecture with three hidden layers each with 200 nodes and rectified linear units (ReLU) on all layers. The first method of training (ϕ) uses labels derived from a first order Griewank function

$$f(x) = 1 + (1/4000) \cdot x^2 - cos(x) \tag{4}$$

and the second method employs collaborative training ($\mathbb{C}$) using a critic function to describe the performance of each network for a given input, $x \in [-600, 600]$ (the typical testing bounds for this benchmark function), by measuring the absolute distance between $f(x)$ and $g_{\theta_n}(x)$ where $g_{\theta_n}(x)$ is the resulting output for the network θ_n.

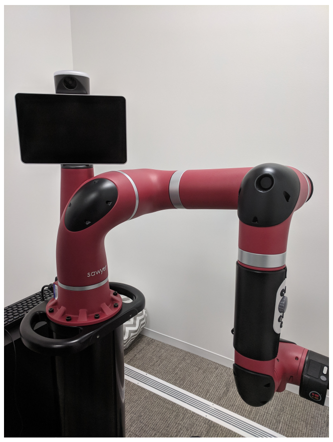

Figure 2: Sawyer robot has seven degrees of freedom.

Forward Kinematics In the second experiment the networks are tasked with predicting the 3-d coordinates of a robotic arm. We use the kinematic chain as defined by the Rethink Robotics' Sawyer Robot which is a seven degree-of-freedom (DoF) robotic arm with a single end effector. The arm is shown in Figure 2 and thus x is a vector of length seven and $g_{\theta_n}(x)$ is a vector of length three. Labels for ϕ are gathered using typical forward kinematic calculations for a random set of joint angles and the critic function is the euclidean distance between the actual coordinates and those generated by $g_{\theta_n}(x)$. Each network has three hidden layers each with 200 nodes with ReLU activation.

Results. The results based on 10,000 sample test sets for each experiment are shown in Tables 1 and 2 and Figures 3 and 4. In the single variate case (Griewank) the supervised network, ϕ, performs equivalently to each individual network in the ensemble. In the case in which both the inputs and outputs are multivariate (forward kinematics), the ϕ network unsurprisingly learns the fastest. However, the differences between ϕ and the collaboratively trained networks are not substantial despite $\{\theta_1, \theta_2, \theta_3\}$ not training on labeled data.

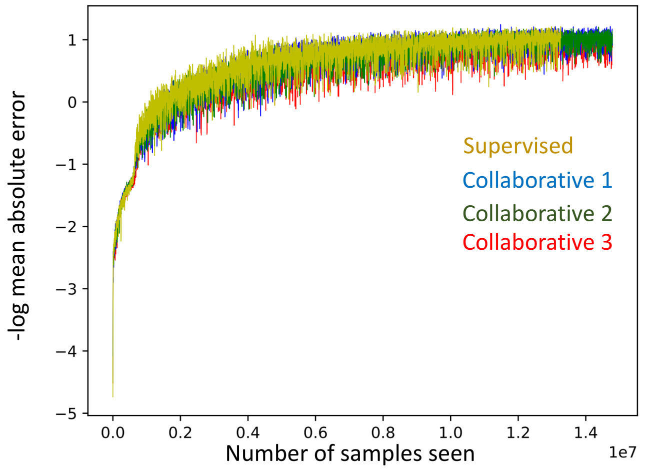

Figure 3: Losses for Griewank experiment are shown here. The x-axis indicates the number of samples seen and the y-axis indicates the performance. In this experiment the performance is given by the negative log mean absolute error between the network generated output for a given input, x, and the true value determined by the Griewank function.

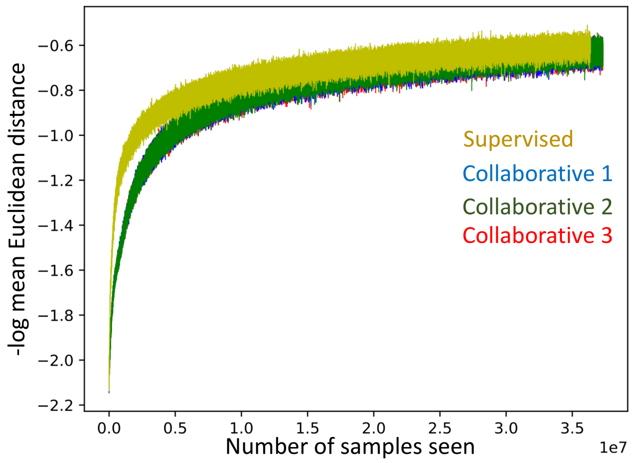

Figure 4: Losses for forward kinematics experiment are shown here. The x-axis indicates the number of samples seen and the y-axis indicates the performance. In this experiment the performance is given by the negative log of the euclidean distance between the network generated output for a given input, x, and the true value determined by the forward kinematic calculation for the Sawyer robot.

4.2 Inverse Kinematics

Neural methods for inverse kinematics (IK) are a more interesting and challenging problem because there are multiple joint solutions for a set of end effector coordinates. Standard approaches to solving the IK problem do not include training models, but instead rely on computing the Jacobian and iteratively updating the joint positions until the end effector is within a set distance threshold of the desired

model	ϕ	θ_1	θ_2	θ_3
mean error	.125	.127	.131	.125
standard deviation	.022	.026	.018	.018

Table 1: Griewank function results. The error represents the average absolute difference between the true Griewank value and the network approximated value for 10,000 random input values $x \in X[-600, 600]$.

model	ϕ	θ_1	θ_2	θ_3
mean error	3.57	3.71	3.66	3.69
standard deviation	.41	.41	.39	.45

Table 2: Forward kinematics results. Error represents the mean euclidean distance (centimeters) between the ground truth coordinates and approximated coordinates.

coordinates. We use this methodology as a baseline and would expect resulting performance to be an upper bound for neural network based IK solvers. The lower bound for neural network based IK would be to train a single network using typical supervised methods in which the data is not controlled for redundant solutions. In the following experiments we train feed forward networks to generate a set of joint angles given the end effector coordinates.

Inverse Kinematics In this experiment we use the Sawyer Robot kinematic chain. Our hypothesis is that the collaborative training methodology will learn a mapping that will outperform a network trained on labeled data because it will optimize specifically for the task of generating a single set of joint angles that satisifies the given end effector coordinates, thus, “protecting" the learning against redundant solutions. The critic function used here is the euclidean distance between the goal coordinates, x (the input to the networks), and the coordinates produced by the set of joint angles generated from $g_{\theta_n}(x)$. This contrasts with the labeled data methods that are commonly used in previous work which will model the distribution of all possible joint angle solutions for a specific set of coordinates [2, 4]. The input for the networks is a vector of length three (coordinates of the end effector) and the outputs are vectors of length seven representing angles for each joint. We complete this experiment three times with networks of varying sizes. We use a network with three fully connected hidden layers with the following sizes for the three runs: [200, 200, 200], [1000, 1000, 1000], [3000, 3000, 3000].

While collaborative training inherently addresses the redundancy problem there are examples in the literature when a network is provided only a single joint solution for a given set of coordinates to prevent the network from learning the distribution across all solutions. By manually choosing solutions in this manner the responsibility is shifted to the individual for selecting appropriate joint angles that will minimize the potential for redundancy. One method is to select a solution when a specific joint(s) in the chain is locked at a particular angle or restricted to a given range. In [3] multiple networks are trained to support different fixed joint positions. We also include this method as a baseline for comparison. We train an ensemble of 6 networks (each with identical architectures as those used for the collaborative training) and redundancy is reduced by limiting joint solutions to a set of 6 different ranges.

model	ϕ	θ_1	θ_2	θ_3	ensemble
mean error (200)	6.4	4.20	4.10	4.15	2.98
standard deviation (200)	1.52	.71	.72	.71	.78
mean error (1000)	6.13	2.12	2.12	2.19	2.26
standard deviation (1000)	1.65	.23	.23	.23	.31
mean error (3000)	5.91	.56	.53	.54	2.07
standard deviation (3000)	1.55	.18	.15	.19	.26

Table 3: Inverse kinematics results. The error describes the resulting mean Euclidean distance (centimeters) between the end-effector position generated by the model and the end-effector position generated by Sawyer’s built-in Jacobian IK method for a random input coordinate.

Results The results of a 100 size sample test set are shown in Table 3. Results were obtained by measuring the average euclidean distance between Sawyer’s end-effector position as determined by the built-in benchmark Jacobian-based IK method for a given 3-d coordinate and the resulting joint position provided by each network. While the results are not perfect, the precision of the collaborative networks increased with the number of learn-able parameters. In the testing scenario in which the networks used 3000 unit layers the difference is within about half a centimeter. The convergence plots for the three network configurations are shown in 5.

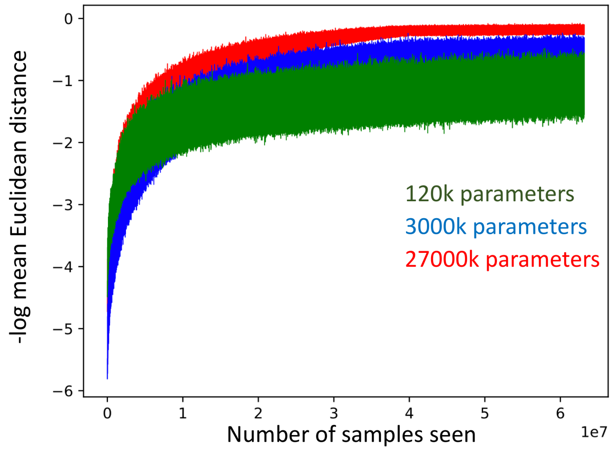

Figure 5: Average losses for collaborative networks for the three network configurations. Network parameter counts are 120k, 3000k, 27000k.

Also, in agreement with our hypothesis, each network resulting from the collaborative training method outperforms the training method using explicitly labeled data and while the ensemble of six networks also consistently performed better than the single network the performance did not increase at the same rate as the collaborative training with the increase of parameters. This suggests that there was still some redundancy in the solution space during training and this solution is not feasible with kinematic chains with a high number of DoFs and high ranges of motion. Addressing all of the possible configurations by locking joints at particular angles becomes increasingly difficult with each additional degree of freedom.

Additionally, we found that collaboratively trained networks generate joint angles that result in very smooth trajectories because without any additional constraints the training process naturally minimizes differences among joint angles for geometrically close input coordinates.

4.3 Nonlinear Controls

In this experiment we examine collaborative network training for a common nonlinear controls problem. We use the OpenAI gym pendulum-v0 environment with the task of keeping a friction-less pendulum standing up [1]. The state of the art and current high score for the task uses a variant of DDPG to learn an effective policy. We compare our method to this.

In DDPG and related reinforcement learning methods the actor network generates the next action given the current state. As experience is gained a policy that enables high rewards is eventually learned. Instead of predicting a single action and maintaining a memory of what actions lead to good results, our implementation based on collaborative training simply predicts a sequence of several actions. We do not use a memory buffer or specific exploration protocol as is typical for RL; rather, a single network generates an entire action sequence given only the initial starting condition.

The reward function for the task is determined by the angle of the pendulum and the required effort of each action. The precise equation for the reward used in the gym's environment is:

$$-(\lambda^2 + 0.1 * \lambda_d t^2 + 0.001 * a^2) \quad (5)$$

where λ is the pendulum angle and a is the action. The architecture of each network in the collaborative ensemble takes the start state as input, has three ReLU activated hidden layers with 30 nodes each (identical to current OpenAI winner), and has an output layer of 25 linear units. These 25 values in the output vector represent the next 25 actions (torques) given the initial condition. The critic function comparing performance among the networks is identical to this reward function and is accumulated over the 25 actions. This technique is related to model predictive control in that each individual action for a specific time slot is optimized while taking all other actions into account [5].

The trio does not utilize additional joint angle and velocity information after the initial state that is provided as the input. Therefore, in order to achieve good performance the networks must essentially learn a robust model about the dynamics of the pendulum system to produce a good sequence. For each subsequent action in the sequence the performance is likely to degrade as the errors in the network's model accumulate. For this reason, after training and during testing we use an overlapping method of using only the first five values in the 25-value generated action sequence. The pendulum state is read again after the five actions and a new 25 step action sequence is generated and so on (see Figure 6).

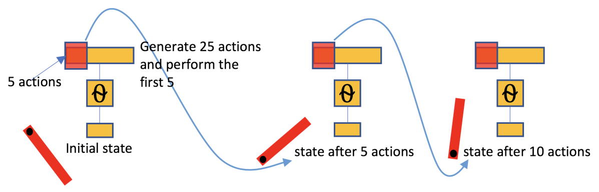

Figure 6: The 25 actions are carried out sequentially and the resulting rewards are accumulated for each step. The critic function compares the accumulated reward of each network when given the same input state. During inference the generated action sequences are used in an overlapping manner. The network generates a sequence of 25 actions, but only the first five are used. After these five actions are performed the pendulum state is read and a new set of 25 actions are produced. The first five of these are used, the pendulum state read again, and so on.

model	DDPG	θ_1	θ_2	θ_3
mean reward	-134.48	-136.41	-127.84	-128.32
standard deviation	9.07	93.25	81.79	88.72
geometric mean reward	not reported	-72.051	-73.66	-53.35

Table 4: Best 100-episode performance on the inverted pendulum task.

Results Table 4 shows the results of the best 100-episode performance on the pendulum task. The mean reward of the collaborative training method is an improvement over the DDPG method reported on OpenAI. However, the distribution of rewards from collaborative training is heavily skewed. For skewed data the geometric mean is a better indicator of central tendency and indicates improved performance over DDPG. Though collaborative training may have better overall results, the standard deviation is much higher and appears to be more prone to poorly performing outliers. Given this behavior, we ran 10,000 episodes and used a t-test (with a bonferonni post-hoc correction)to evaluate if there were significant differences between the results obtained by DDPG and the collaborative networks. The mean rewards for these 10,000 episodes are reported in the table below. The results between each collaborative network and the DDPG network were significant at the $p < .01$ level.

model	DDPG	θ_1	θ_2	θ_3
mean reward	-157.48	-143.22	-145.94	-141.23
standard deviation	19.07	97.41	86.98	101.11

Table 5: Mean rewards for 10,000 episodes on the inverted pendulum task.

4.4 Millisecond Precision for Model-based Drumstick Trajectory Generation

While the previous experiment demonstrated the methodology's ability to learn an effective policy within a controlled simulated environment, in this section we demonstrate the efficacy with a real-world system. In this experiment we attempt to train a network to generate a sequence of motor velocities resulting in the production of drumstick trajectories allowing it to strike the drum in a rhythmically controlled manner (see Figure 7.

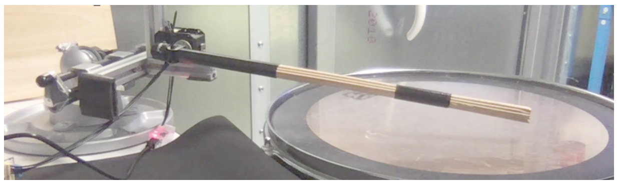

Figure 7: One end of a drumstick is secured to a motor (MX-28 Dynamixel). The goal is to train a network to control the motor so that the drumstick strikes the drum producing a rhythm described by the input vector.

In the previous experiments the critic functions used to evaluate the performance of each network were relatively straightforward. Here, we highlight a custom critic function that incorporates a method to measure the rhythmic similarity between the resulting drum strikes and the input onset sequence as well as a method to measure “goodness" when the drumstick doesn’t come into contact with the drum at all producing no onsets. To measure rhythmic similarity we compute the cosine distance between the input vector and the resulting onset vector generated from the drumstick movements. To evaluate the trajectory we measure the euclidean distance between the tip of the drumstick and two constant values representing either the ideal “rest" position or ideal “strike" position. Therefore, we assume prior knowledge of the positions of the stick and drum in the environment. This allows us to train the models in simulation by replicating the real-world setup.

Another step to better ensure adequate transfer from simulation to real-world is to use a symbolic representation of the desired onset sequence rather than the raw audio signal. In doing this the actual audio does not need to be simulated. Therefore, the input to the networks are a sequence of zeros and ones. Each value in the input vector represents an event a specific time where the temporal resolution is 10ms. We provide the networks with a sequence of 20 events representing 200ms. In the vector a one indicates an onset and zeros indicate no onsets at that time. Additionally, the starting angle of the joint is included in the input vector (see Figure 8.

Frequently a network will produce actions that are not possible. The safety mechanisms within the simulation recognizes illegal actions and prevents them. In the scenario where an action would need to exceed the velocity threshold to successfully reach the target position, the simulation moves as far as it can in the same direction at the thresholded rate. In the scenario where an action would require the robot to pass through the drum the simulation ignores the action all together and does nothing. Because of this behavior, the sequence of performed velocities are likely to be different than the sequence of generated joint angles. This is particularly true during the early stages of training. In order to encourage the networks to learn to generate legal actions we make a modification to the loss function (Equation 3) for the best performing network in the ensemble. Instead of using the *generated* joint action sequence

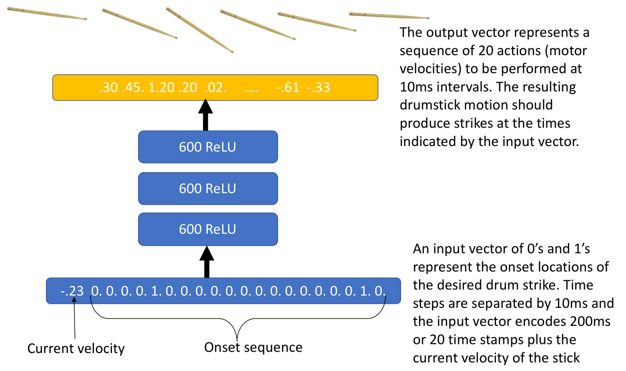
The output vector represents a sequence of 20 actions (motor velocities) to be performed at 10ms intervals. The resulting drumstick motion should produce strikes at the times indicated by the input vector.
.30 .45. 1.20 .20 .02. .... -.61 -.33
600 ReLU
600 ReLU
600 ReLU
An input vector of 0’s and 1’s represent the onset locations of the desired drum strike. Time steps are separated by 10ms and the input vector encodes 200ms or 20 time stamps plus the current velocity of the stick
-.23 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.
Current velocity
Onset sequence

Figure 8: The network is trained to produce a sequence of actions controlling the drumstick’s trajectory. The resulting trajectory should produce a rhythm that is identical to the sequence provided by the input vector.

from the best and worst performing networks, $\hat{y}_{w_1}$ and $\hat{y}_{w_3}$, we use the *effected* joint action sequence, $e(\hat{y}_{w_1})$ and $e(\hat{y}_{w_1})$, where $e(x)$ is a function representing the simulation and built-in safety constraints. Thus, the loss function for the worst performing network is defined as:

$$\mathfrak{L}_{w_3} = d_{MAE}(e(\hat{y}_{w_1}), e(\hat{y}_{w_1}) + (e(\hat{y}_{w_1}) - e(\hat{y}_{w_3}))) \tag{6}$$

Using the effected sequence is feasible because a differentiable loss function is not required.

Similarly to the previous experiment, during inference an overlapping method of using only the first few actions of the output is put in place. In this experiment, we only using the first action before recomputing another sequence of 20 actions.

Results We evaluated the system in both simulation and the real-world. During real-world inference we wanted a user to be able to provide input by drumming. To do this, the drumming audio is recorded and a spectral difference-based onset detection method ([15]) is used to create the input sequence necessary for the network (see Figure 9). This pre-processing audio analysis step introduces the potential for error by either missing onsets or producing false positives. Though, the method is relatively robust achieving roughly 90% accuracy in the controlled environment in which we performed the experiment.

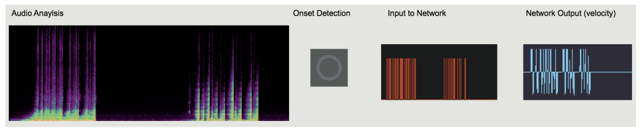

Figure 9: The processing pipeline of the real-time system includes an audio stream which is analyzed for drum onsets. The onset sequence is provided as input to the network which then produces a sequence of motor velocities.

To evaluate in simulation we generated 1000 onset sequences with random initial joint positions and measured the cosine similarity between the resulting onsets and the input sequence. It was guaranteed that the robot was physically capable of playing all of the test sequences, therefore, any errors could not be attributed to the sequences being outside of the robot's capabilities. For the real-world evaluation 3 minutes worth of drumming was recorded from a professional drummer and analyzed producing 852 onsets. The resulting onset sequence was then fed into the network (at 200ms intervals) resulting in an approximate imitation of the human drummer. Onset detection was performed on the resulting audio and cosine similarity was measured between the human and robot drummer's onsets. The results are shown in Table 6.

	Simulation	Real-world
cosine similarity	1.0	0.86

Table 6: Cosine similarities (1 is perfect) between the input rhythms and the rhythms generated by the drum strikes resulting from the motor velocity commands.

In both simulation and the real-world settings the generated onset had to occur within the 10ms window of the target as specified by the input onset sequence. Anything outside of this window (no matter how close) was treated as an error and would penalize the cosine similarity score. In simulation the network was able to produce actions that perfectly produced the desired input sequence. In the real-world scenario performance declined, but the results were still convincing and much of the decline was likely due to the onset detection.

5 CONCLUSION

In this article we introduced *Collaborative Network Training*. We demonstrate the efficacy using both simulation and real-world environments using experiments ranging from traditional (IK and inverted pendulum) to novel (drum strike trajectory). In the IK experiment the methodology addresses the problem of redundancy and achieved results similar to standard iterative Jacobian-based methods. In the simulated inverted pendulum experiment the network learned policies that achieved results on par with the state of the art reinforcement learning method, DDPG. Finally, in the drum strike trajectory planning experiment the method resulted in a policy enabling a motor to strike a drum with millisecond level precision.

The experiments presented in this work used only a single network during inference. Future directions of this work will examine potential gains of leveraging the ensemble during inference as well as including additional networks with varying architectures during training.

REFERENCES

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. Openai gym. *arXiv preprint arXiv:1606.01540* (2016).

[2] Akos Csiszar, Jan Eilers, and Alexander Verl. 2017. On solving the inverse kinematics problem using neural networks. In *Mechatronics and Machine Vision in Practice (M2VIP), 2017 24th International Conference on*. IEEE, 1–6.

[3] Bassam Daya, Shadi Khawandi, and Mohamed Akoum. 2010. Applying neural network architecture for inverse kinematics problem in robotics. *Journal of Software Engineering and Applications* 3, 03 (2010), 230.

[4] Adrian-Vasile Duka. 2014. Neural network based inverse kinematics solution for trajectory tracking of a robotic arm. *Procedia Technology* 12 (2014), 20–27.

[5] Carlos E Garcia, David M Prett, and Manfred Morari. 1989. Model predictive control: theory and practiceâĂŤa survey. *Automatica* 25, 3 (1989), 335–348.

[6] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in neural information processing systems*. 2672–2680.

[7] Alex Graves and Navdeep Jaitly. 2014. Towards end-to-end speech recognition with recurrent neural networks. In *International Conference on Machine Learning*. 1764–1772.

[8] Alex Graves and Jürgen Schmidhuber. 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks* 18, 5-6 (2005), 602–610.

[9] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. 1983. Optimization by simulated annealing. *science* 220, 4598 (1983), 671–680.

[10] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. 2016. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492* (2016).

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[12] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[14] Donald M Olsson and Lloyd S Nelson. 1975. The Nelder-Mead simplex procedure for function minimization. *Technometrics* 17, 1 (1975), 45–51.

[15] Miller S Puckette, Miller S Puckette Ucsd, Theodore Apel, et al. 1998. Real-time audio analysis tools for Pd and MSP. (1998).

[16] Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

[17] Siddharth Sigtia, Emmanouil Benetos, and Simon Dixon. 2016. An end-to-end neural network for polyphonic piano music transcription. *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)* 24, 5 (2016), 927–939.

[18] Richard S Sutton and Andrew G Barto. 1998. *Reinforcement learning: An introduction*. Vol. 1. MIT press Cambridge.

[19] Aaron Van Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. 2016. Pixel Recurrent Neural Networks. In *International Conference on Machine Learning*. 1747–1756.

[20] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. 2010. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research* 11, Dec (2010), 3371–3408.

[21] Kun Xu, Yansong Feng, Songfang Huang, and Dongyan Zhao. 2015. Semantic relation classification via convolutional neural networks with simple negative sampling. *arXiv preprint arXiv:1506.07650* (2015).

[22] Zhi-Hua Zhou. 2015. Ensemble learning. *Encyclopedia of biometrics* (2015), 411–416.