Augmenting Sampling Based Controllers with Machine Learning

Joose Rajamäki
Aalto University
Helsinki, Finland
joose.rajamaki@aalto.fi

Perttu Hämäläinen
Aalto University
Helsinki, Finland
perttu.hamalainen@aalto.fi

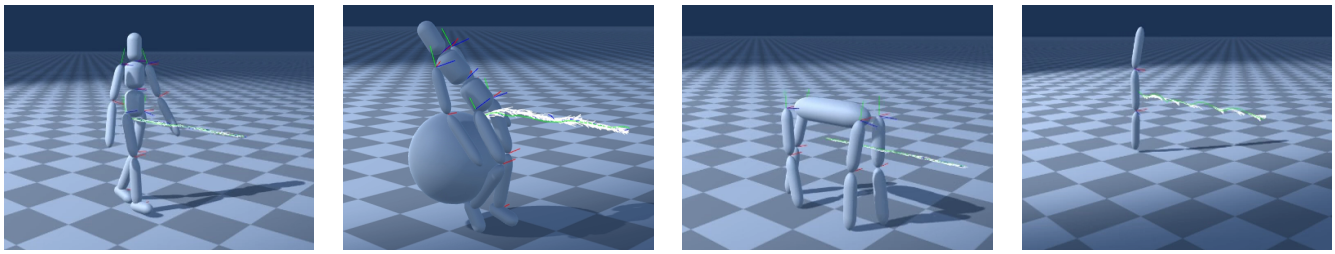


Figure 1: A humanoid, a quadruped and a monoped controlled by our learning backed sampling based model predictive controller. The various characters learn a stable gait in under a minute on a 4-core desktop computer.

ABSTRACT

Efficient learning of 3D character control still remains an open problem despite of the remarkable recent advances in the field. We propose a new algorithm that combines planning by a sampling-based model-predictive controller and learning from the planned control, which is very noisy. We combine two methods of learning: 1) immediate but imprecise nearest-neighbor learning, and 2) slower but more precise neural network learning. The nearest neighbor learning allows to rapidly latch on to new experiences whilst the neural network learns more gradually and develops a stable representation of the data. Our experiments indicate that the learners augment each other, and allow rapid discovery and refinement of complex skills such as 3D bipedal locomotion. We demonstrate this in locomotion of 1-, 2- and 4-legged 3D characters under disturbances such as heavy projectile hits and abruptly changing target direction. When augmented with the learners, the sampling based model predictive controller can produce these stable gaits in under a minute on a 4-core CPU. During training the system runs real-time or at interactive frame rates depending on the character complexity.

CCS CONCEPTS

• **Computing methodologies** → **Online learning settings**; *Physical simulation*;

KEYWORDS

Model Predictive Controller, Monte Carlo Tree Search, Reinforcement Learning

SCA '17, Los Angeles, CA, USA

DOI: 10.1145/3099564.3099579

ACM Reference format:
Joose Rajamäki and Perttu Hämäläinen. 2017. Augmenting Sampling Based Controllers with Machine Learning. In *Proceedings of SCA '17, Los Angeles, CA, USA, July 28-30, 2017,* 9 pages.
DOI: 10.1145/3099564.3099579

1 INTRODUCTION

It is a long standing dream to make game or animation characters develop skills on their own. That would open up new avenues for games and free animators to command characters on a high level instead of laboriously editing movement. Reinforcement learning has lately been showing promise for making characters develop skills on their own in the same manner as humans do by exploring various options. On the other hand, recent trajectory optimization approaches [Hämäläinen et al. 2015; Tassa et al. 2014] allow characters to act based on high-level goals without any learning phase, although movement quality leaves room for improvement. Combining these approaches would allow a character to build up new skills fast and learn from relevant experience [Mordatch et al. 2015] instead of going through the baby-like phase of extremely slow learning in the beginning.

Recent work has demonstrated that sampling based model predictive controllers (SMPC) can be used to control full human characters in real-time [Hämäläinen et al. 2014, 2015] and under challenging conditions. We further develop these controllers and augment them with different learners. Our approach has the following characteristics:

- a gradient-free, sampling-based model predictive controller (SMPC) that can naturally handle black-box dynamics simulators with contact discontinuities;
- no need to pre-train the controllers with data that can be hard to obtain;
- combination of using fast but imprecise nearest neighbor learning and slow but precise neural network training;

Even though reinforcement learning has demonstrated techniques to accelerate learning [Pritzel et al. 2017] and asynchronous

methods can be used to utilize parallelization [Mnih et al. 2016], the algorithms still require up to hours or days to learn even a basic viable policy. Recently, evolutionary strategies have been demonstrated to train fast because they can be trivially parallelized. For instance Salimans et al. [2017] demonstrated training the OpenAI 3D characters in 10 minutes using evolutionary strategies by parallelizing the computation to 1440 parallel workers. To the best of our knowledge, no previous work has demonstrated all the strengths of our system, i.e., 1) producing complex behavior such as stable 3D bipedal and quadrupedal locomotion in under a minute on a single CPU, 2) rapid online learning while the character can be interacted with, 3) ability to control a black-box simulation with complex contact discontinuities, and 4) support for reducing the sampling budget after a couple of minutes of training for fully real-time operation. Our primary technical contribution is to combine two types of machine learning with sampling based control, and the quantitative evaluation shows that the combination of learners works better than either learner alone.

2 RELATED WORK

In this section we briefly review reinforcement learning methods, sampling based controllers and various approaches for synthesizing physically based locomotion for animated characters. We focus on approaches that simulate or otherwise handle full dynamics, as opposed to primarily kinematic approaches such as Motion Graphs [Kovar et al. 2002] and Motion Fields [Lee et al. 2010], which do not handle the environment's dynamics and can perform unphysical movements.

2.1 Reinforcement learning

Reinforcement learning is a way to learn to act in a given environment. It has demonstrated remarkable achievements in the recent years such as learning to play Atari games [Mnih et al. 2015] and mastering the game of Go [Silver et al. 2016]. The basic setting is that an agent chooses an action a in a state s and observes a new state s' and a reward r. A good general introduction to reinforcement learning is given by Sutton and Barto [1998]. In reinforcement learning the methods which model both the cost function and a policy are called actor-critic methods. They train an actor which is a mapping from states to actions $a = \texttt{actor}(s)$ and a critic which tells the expected reward that we get by executing action a in state s, i.e. $Q(s, a) = \texttt{critic}(s, a)$. These are usually implemented as neural networks and from here on we use the **bold typewriter font** to denote neural networks.

In continuous action spaces, reinforcement learning is mostly based on the policy gradient [Silver et al. 2014] or the continuous actor critic learning automaton (CACLA) [Van Hasselt and Wiering 2007] algorithm. In policy gradient methods one updates the policy to the direction of the gradient of **critic**. In CACLA one updates the policy towards the actions that have a positive temporal difference, i.e. towards the actions that seem to be better than previously thought. Even though these methods learn to perform well they require stabilizing techniques like experience replay and target **critic** to decorrelate the data and to prevent divergence [Mnih et al. 2015]. Recent techniques exist to make learning much faster [Pritzel et al. 2017], but still pure reinforcement learning takes a long time to learn.

2.2 Planning and learning

In high-dimensional and/or continuous-control settings reinforcement learning has remained somewhat less successful of an approach. Instead, planning and learning has shown promise to handle learning in these kind of environments in a reasonable time, e.g. [Zhang et al. 2016]. Here we mean by planning and learning the interleaved processes of optimizing trajectories and learning the controls in a supervised manner. Furthermore, the combination of planning with reinforcement learning has been demonstrated to be effective. For instance mastering the game of Go was performed by a combination of Monte Carlo tree search (MCTS) and reinforcement learning [Silver et al. 2016].

In continuous-control tasks guided policy search (GPS) is one way to perform planning and learning [Levine and Koltun 2013; Zhang et al. 2016]. GPS utilizes differential dynamic programming (DDP) [Mayne 1966], which constructs a quadratic cost model around a nominal trajectory. GPS operates by sampling from the Gaussian information projection of this quadratic model and using the cost estimates to estimate the policy gradient. Recent advances in sampling based control also include a Monte Carlo variant of DDP [Rajamäki et al. 2016] and its unscented variant [Manchester and Kuindersma 2016] that could be used in exploration very much in the same way as guided policy search, but circumventing the requirement of a differentiable transition model. Similarly to guided policy search, ideas of alternating trajectory optimization and learning the state-to-control mappings of the optimized trajectory have been used by Mordatch et al. [2015], who demonstrate a distributed approach to interleaving planning and learning. However, these algorithms are computationally heavy. We build our learning on top of a real-time sampling based controller simplified from that of Hämäläinen et al. [2015]. Thus, we differ from the most planning algorithms, which are based on DDP and thus require the dynamics to be differentiable.

Sampling based controllers and trajectory optimization have been used for a long time in the animation research. For example Liu et al. [2015, 2010] use a sampling based controller to construct the controls for a character tracking a reference animation. Sampling has also been used as an oracle to build controllers from a library of motion capture clips [Liu et al. 2016]. In contrast, we abstain from using motion capture data. The behavior that our algorithms exhibit arises solely from optimizing a given objective function. Recently introduced sampling based controllers [Hämäläinen et al. 2014, 2015] have also demonstrated capability of controlling humanoid characters in real-time without any pre-processing or reference animation, albeit having much noise and not demonstrating long stable walk cycles. Their approach is interesting in the sense that the characters are able to act in a given environment already before learning anything. We aim to build on this approach and produce a real-time sampling based model predictive controller that learns.

To sum up, our model predictive controller does not use heavy optimization procedures such as DDP or its popular variant, iterative linear-quadratic regulator (iLQR) [Li and Todorov 2004]. Instead our controller is based on only sampling and guiding the sampling

process. Everything we do runs in real-time or near real-time on a single multi-core CPU. Unlike pure reinforcement learning algorithms, we rely on our sampling-and-selection to produce good controls that we model in a supervised manner, i.e. we do not model the expected reward at all by using a `critic` . The goal of this work is to guide the sampling process instead of using a sampling or optimization process to find a control policy, which is performed eg. by GPS and [Mordatch et al. 2015]. Thus, our algorithm does not need to take into account that all of our data is generated off-policy.

2.3 Physically based locomotion

Synthesizing physically based locomotion has been a long standing goal of both scientists and artists, and various approaches exist. Gait optimization has been done for example by Wampler and Popović [2009]. They optimize a gait for a given character by considering its physical properties using covariance matrix adaptation evolution strategy (CMA-ES) [Hansen 2015]. A similar type of approach is employed by Geijtenbeek et al. [2013] who optimize muscle routing and locomotion for various gaits. In contrast to these previous approaches, we also handle more diverse actions including rapid turns and recovery from impacts. Our sampling-based model-predictive controller can also start acting already when learning starts, which provides a rapid iteration cycle for the cost or reward function designer.

Apart from the previously mentioned evolutionary computing approaches to gait optimization, there are various ways to break the movement to more manageable subtasks. For instance many algorithms utilize zero moment points [Vukobratović and Borovac 2004]. One way is to perform planning for simple models such as spring loaded inverted pendulum (SLIP) [Mordatch et al. 2010]. Impressive motion results have been demonstrated using state machines that segment movement to different phases. Most notable of these is the SIMBICON [Yin et al. 2007] that can be used for biped control. SIMBICON state machines have also been used as a building block by for example Peng et al. [2015, 2016]; Wang et al. [2010]. In robotics, dynamic motion primitives [Ijspeert et al. 2013; Schaal 2006] have been widely researched. For the sake of generality, to abstain from manually breaking down movement to more manageable bits. Instead we explore and learn raw motor commands.

3 METHODS

This section presents our algorithms. First, in Section 3.1 we present our sampling based model predictive controller and how the sampling is informed by three information sources. The most straightforward of these is using the trajectories of the previous sampling iteration. The more long-lasting and temporally more dispersed information sources are presented in Sections 3.2 and 3.3. The fast but imprecise learning, which stores the old best actions for approximate nearest neighbor query is discussed in Section 3.2. Section 3.3 presents the more gradual training process of a multilayer neural network.

3.1 Fixed Depth Informed Monte Carlo Tree Search

Our sampling based model predictive controller comes in the form of a process we call Fixed Depth Informed Monte Carlo Tree Search (FDI-MCTS). This process is explained in Algorithm 1. In each iteration we sample controls (target angles for the simulated character's joint motors) for T time steps and N trajectories simulated forward from the current state, in order to predict the future and test the effectiveness of different controls. Although N simulations advance in parallel in multiple threads, the trajectories form a tree; low-reward (high-cost) trajectories are pruned before the planning horizon and their simulation resources assigned to new forks of high-reward (low-cost) trajectories. The trajectories, pruning, and forking can be seen at work on the supplemental video, where we visualize the simulated trajectories of the characters' centers of mass. We run one iteration per simulation time step.

We utilized a lot of the framework presented by Hämäläinen et al. [2015], but simplify to avoid having to convert cost function values to probabilities through exponentiation, which brings with it the danger of underflows and parameters that are difficult to tune.

Hämäläinen et al. [2015] use products of Gaussian probability densities to combine information. A single information source is presented in the form of a single Gaussian density. Multiple information sources can be combined by multiplication of the densities. The product of two densities $\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_1, \Sigma_1)$ and $\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_2, \Sigma_2)$ is a Gaussian function itself, and when properly normalized it is a Gaussian density [Bromiley 2003], i.e.:

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \Sigma) = \frac{1}{Z}\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_1, \Sigma_1)\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_2, \Sigma_2^2) \quad (1)$$

$$\Sigma = \Sigma_1^{-1} + \Sigma_2^{-1} \quad (2)$$

$$\boldsymbol{\mu} = \Sigma\left(\Sigma_1^{-1}\boldsymbol{\mu}_1 + \Sigma_2^{-1}\boldsymbol{\mu}_2\right) \quad (3)$$

Here Z is a normalizing constant that ensures that $\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \Sigma)$ integrates to one. Σ_i are covariance matrices and $\boldsymbol{\mu}_i$ are the means. To simplify computation, we only use diagonal covariance matrices.

The probabilistic framework of [Hämäläinen et al. 2015] is conceptually somewhat involved. They also have state-space kernel functions which have to be adjusted by the user. This can be a hard task because distances in high dimensional spaces are not intuitive. On the contrary, Monte Carlo tree search is simple and straightforward, and instead of evaluating state space kernels, we use nearest neighbor searches. Our MCTS's major difference to a standard Monte Carlo tree search is that we grow the tree synchronously in time, i.e. we proceed step by step in time and each transition that we evaluate at time step t has to be continued from a state $\mathbf{s}$ that was an end state $\mathbf{s}'$ of time step $t-1$. This has two desirable properties: 1) running the computationally intensive dynamics simulations in parallel is trivial, and 2) we ensure that N trajectories reach the planning horizon T. Multiple long trajectories avoids short-sighted control, which would easily lead to the loss of balance in 3D characters [Hämäläinen et al. 2014, 2015]. Our easy to understand search method, presented in Algorithm 1, is not a central contribution to the field but rather a starting point. The main improvements lie in combining a real-time capable sampling-based controller with two sorts of learning that happen in the background, and make the sampling perform much better.

Algorithm 1 Fixed-Depth Informed MCTS (FDI-MCTS)

1: **for** Time step $t = 0...T$ **do**
2: Compute the current current reward $r_{\text{current}} = \sum_{\tau=0}^{t-1} r_\tau$ for each trajectory
3: Find the maximum reward $r_{\max}$ so far
4: **for** Trajectory index $n = 0...N$ **do**
5: **if** $r_{\text{current}} < r_{\max} - r_{\text{threshold}}$ and $n \in \mathcal{I}_{\text{prunable}}$ **then** //If we are here the current trajectory will be pruned and another one forked.
6: Select a trajectory to fork from the existing ones that have $r_{\text{current}} > r_{\max} - r_{\text{threshold}}$
7: Construct a sampling distribution for the current state $\mathbf{s}$ depending on the trajectory role
8: Sample action $\mathbf{a}$ for the current trajectory
//Here f denotes the environment.
9: Perform transition $\mathbf{s}' = f(\mathbf{s}, \mathbf{a})$
10: Observe the reward r
11: Compute the rewards of the trajectories $\mathcal{R}_n = \sum_{t=0}^{T} r_{t,n}$
12: Sort the trajectories to descending order by $\mathcal{R}_n$
13: Store K best trajectories
14: Send the best trajectory's first transition's control $\mathbf{a}_{\text{best}}$ to the character to use
15: Send the current state and used control pair $(\mathbf{s}, \mathbf{a})$ to the fast and slow learners

The trajectory pruning greediness can be controlled by the parameter $r_{\text{threshold}}$. It is the reward threshold that sets the maximum deviation from the current time highest-reward trajectory (reward accumulated from $t = 0$). Any trajectory whose cost is $r_{\text{threshold}}$ below the maximum reward will be pruned and a higher reward trajectory will be forked instead. $r_{\text{threshold}} = 0$ corresponds to fully greedy search. $r_{\text{threshold}} = \infty$ corresponds to no pruning or forking. We have found it useful to adjust the threshold adaptively. In all the simulations of the supplemental video, the threshold is set to double the the cost of the previous iteration's best trajectory. Note that our implementation uses non-negative costs and the rewards can be thought of as negated costs.

Trajectory roles. There are many roles for the trajectories of Algorithm 1. Depending on the role, constructing the sampling distribution on line 7 is done differently. Next we explore the different roles and their sampling distributions. To construct a sampling distribution $\mathcal{N}_{\text{sampling}}$ we start with $\mathcal{N}(\mathbf{a}_{\text{init}}, \Sigma_{\text{init}})$. For all particle roles we multiply $\mathcal{N}_{\text{sampling}}$ by $\mathcal{N}(\mathbf{a}_{t-1}, \Sigma_{\text{diff}})$ to promote action continuity. We also have another term $\mathcal{N}(\mathbf{a}_{\text{acc}}, \Sigma_{\text{acc}})$ promoting continuity. This term is presented in Section 4.1 where we also present the controller structure. Following this, the additional information sources may be incorporated through similar multiplications. The trajectories with different information sources can also mix through the pruning and forking mechanism.

Previous iteration best trajectory. One of the samples replicates the old best trajectory up to the moment $T - 1$. This trajectory is the only one that does not belong to the set $\mathcal{I}_{\text{prunable}}$, i.e. it cannot be pruned to fork other trajectories. This also means that we ignore the initially constructed sampling distribution.

Trajectories informed by previous iteration. The algorithm stores the K best trajectories of the previous iteration. These can be used up to time $T - 1$ to inform the sampling, which can also be considered as a form of evolutionary programming. They are used by searching for the transition $(\mathbf{s}, \mathbf{a}, \mathbf{s}', r)_{\text{prev}}$ whose state $\mathbf{s}$ is closest to the current state. The search is performed from the transitions of the corresponding time step of the previous iteration. We use the action to inform the sampler by multiplying the term $\mathcal{N}(\mathbf{a}_{\text{prev}}, \Sigma_{\text{variation}})$ to the sampling distribution $\mathcal{N}_{\text{sampling}}$. After each iteration the K best trajectories are selected to inform the sampling process in the next iteration.

Trajectories informed by previous iteration and approximate nearest neighbors. We have a set of previous best state-action pairs that is managed by a density forest as described in Section 3.2. From this set we find the approximate nearest neighbor state-action pair $(\mathbf{s}, \mathbf{a})_{\text{fast}}$ based on state distance. If its state is closer than the previous iteration state $\mathbf{s}_{\text{prev}}$ found above, we use it instead, i.e., we replace the $\mathcal{N}(\mathbf{a}_{\text{prev}}, \Sigma_{\text{variation}})$ above with $\mathcal{N}(\mathbf{a}_{\text{fast}}, \Sigma_{\text{variation}})$.

Trajectories informed by neural network. We use multilayer perceptron neural network **net** in two ways. One form of using **net** is to directly use the action $\mathbf{a}_{\text{ml}} = \mathbf{net}(\mathbf{s})$. The other form of using **net** is to multiply the sampling distribution $\mathcal{N}_{\text{sampling}}$ by $\mathcal{N}(\mathbf{a}_{\text{ml}}, \Sigma_{\text{variation}})$. Training the network is presented in Section 3.3.

3.2 Density forest

The data set of previously used state-control pairs is managed by a density forest, which enables fast search of approximate nearest neighbors in high-dimensional spaces [Muja and Lowe 2014]. We also perform a linear search from the 100 most recent samples. A comprehensive tutorial to the workings of density forests is provided by Criminisi et al. [2012]. We refer the reader that is unfamiliar with density forests to this tutorial and focus here on parts that are relevant to our approach.

Density forests are ensembles of decision trees. In our implementation, each tree node cuts the state space in half by a linear hyperplane. If we wish to find the approximate nearest data point of the data set stored in the density forest we recurse down each tree in the forest by seeing on which side of the split our search key is. We do a linear search from the data points returned by the trees.

The splitting planes of the forest trees are formed by doing random splits and selecting the one optimizing some criteria. We utilize a heuristic of Criminisi et al. [2012] for minimizing the differential entropy of the data on both sides of the split, i.e. minimizing:

$$\frac{\|\mathcal{D}_{\text{left}}\|}{\|\mathcal{D}\|} \log(|\Sigma_{\text{left}}|) + \frac{\|\mathcal{D}_{\text{right}}\|}{\|\mathcal{D}\|} \log(|\Sigma_{\text{right}}|). \tag{4}$$

Here $\mathcal{D}$ is the data set arriving to a node and $\mathcal{D}_{\text{left}}$ and $\mathcal{D}_{\text{right}}$ are the data sets on the "left" and "right" side of the split. Σ are the states' $\mathbf{s}$ covariance matrices for the "left" and "right" data sets. We can ensure that both data sets contain data points by sampling a normal for the splitting plane, selecting two points at random and

sampling the affine term such that the data points are on different sides of the splitting hyperplane.

As new samples arrive they are added to the leaves of the forest. If the data amount in a leaf exceeds the maximum capacity, the tree is built further starting from the leaf. One of the forest trees is being rebuilt in a background thread all the time, i.e. all the data points in the tree are gathered to one data set, the previous tree structure is discarded and the splits of the state space are computed again starting from the root of the tree. This ensures that random trees in the forest are not biased by the order in which the data arrive.

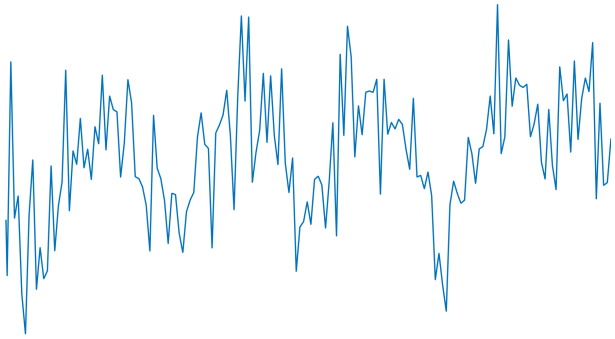
Besides working as a mechanism enabling fast nearest neighbor search, the density forest acts as a fast but imprecise volatile memory because excess data is being forgotten when the trees are rebuilt. We randomly discard data points when rebuilding the tree if the data set is bigger than the tree capacity. The forest as a whole will remember those data points much longer but as time passes the more distant history is increasingly more unlikely to be found in the nearest neighbor search. The density forest thus effectively keeps track of previously chosen actions, but the nearest neighbor search does not interpolate or extrapolate the data or ignore data points that do not fit the others.

3.3 Multilayer neural network

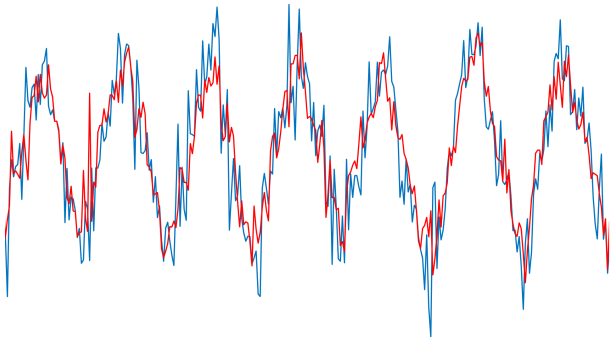
In addition to the density forest of the previous section, we also use a multilayer perceptron network to inform the sampling of Algorithm 1. We use a fully connected network of 5 layers with 100 neurons in each hidden layer. We use ELU non-linearity in the hidden layers. The training runs in a background thread using the last 2000 state-action pairs given by Algorithm 1. The data is quite noisy as illustrated in Figure 2.

We used ADAM training algorithm [Kingma and Ba 2015] to train the neural network. The algorithm is specially designed for gradient descent in noisy setting. It takes into account the "signal-to-noise ratio" of the gradient updates. It does this by exponentially smoothing the first and second order moments of the gradient updates. Smoothing the first order moment can be understood as usage of momentum, which understandably helps in a noisy setting. The parameters we used for learning were: learning rate 10^{-4}, $\beta_1 = 0.9$ and $\beta_2 = 0.99$. We trained on random minibatches whose size were a tenth of the whole data set's size.

Because we have relatively little data, which is also noisy we employ a couple of standard regularization techniques. First we clipped the gradients to interval [-0.1,0.1] (radians, see Section 4.1 for control signal details). This diminishes the effect of single stray data points. Secondly we injected noise to the data points' states $\boldsymbol{s}$ while training. This practice has been demonstrated an effective method to attain better generalization [Mordatch et al. 2015]. The Gaussian noise that was used in the training was drawn from distribution $\mathcal{N}(\mathbf{0}, 10^{-3}\Sigma_{\mathbf{s}})$ where $\Sigma_{\mathbf{s}} = \mathrm{diag}(\sigma_{s_1}^2, ..., \sigma_{s_N}^2)$ and $\sigma_{s_i}^2$ are the variances of the training set's state vectors' dimensions.



(a) Hip control sequence initially.



(b) Hip control sequence after 1000 frames.

Figure 2: Illustrative examples of control sequences of a 3D humanoid character hip joint while walking. Actual realized control is drawn in blue and the controls given by the neural network are drawn in red.

4 RESULTS

We present here the results of comparing and testing the different components of our algorithm. Section 4.1 presents the character models, cost functions, and algorithm parameters. Section 4.2 presents quantitative analysis done using the 3D humanoid presented in Figure 4. Section 4.3 presents qualitative observations made based on the 3D characters shown in the supplemental video and in Figure 1.

4.1 Character models

In addition to qualitatively evaluating our method with the 3D characters shown in the supplemental video, we collected quantitative data with the 3D humanoid presented in Figure 4. The simulations were performed with Open Dynamics Engine. The state features, controls and algorithm settings of the tests are discussed in this section.

The state $\mathbf{s}$ that the controller receives from the characters is composed of the concatenated states of the capsules $\mathbf{s}_{\text{capsule}}$, a target speed for the character v_{target}, the feet y-positions $\mathbf{y}_{\text{feet}}$ and the feet y-velocities $\mathbf{v}_{y,\text{feet}}$. The states of the capsules $\mathbf{s}_{\text{capsule}}$ include the x-position relative to the center of mass, the y-position, the z-position relative to the center of mass, the linear velocity $\mathbf{v}$, the

rotation $\mathbf{q}$ of the capsule and the angular velocity $\boldsymbol{\omega}$. That is,

$$\mathbf{s} = \begin{bmatrix} v_{\text{target}} & \mathbf{y}_{\text{feet}} & \mathbf{v}_{y,\text{feet}} & \mathbf{s}^{\mathrm{T}}_{\text{capsule},0} & \cdots & \mathbf{s}^{\mathrm{T}}_{\text{capsule},j} \end{bmatrix}^{\mathrm{T}} \quad (5)$$

$$\mathbf{s}_{\text{capsule}} = \begin{bmatrix} x - x_{\text{com}} & y & z - z_{\text{com}} & \mathbf{v} & \mathbf{q} & \boldsymbol{\omega} \end{bmatrix}^{\mathrm{T}}. \quad (6)$$

The quantities are presented in a coordinate system that is rotated with the target velocity vector. That's why we only need the target speed v_{target} instead of the full target velocity $\mathbf{v}_{\text{target}}$.

The reward the character gets is actually a lack of punishment, i.e. a negated cost:

$$r = -\sum_i \left(\frac{\tau_i}{80}\right)^2 - \sum_j \left(\frac{\alpha_j}{10}\right)^2 - \left(\frac{\|\mathbf{v}_{\text{target}} - \mathbf{v}\|}{0.05}\right)^2 - \left(\frac{\sqrt{d_x^2 + d_z^2}}{0.025}\right)^2. \quad (7)$$

Here τ are the torques applied by simulator motors for each joint DoF and α denote the capsules' deviation from the initial pose (standing straight) measured as angular distance (degrees) between initial and current quaternions. The pose deviation cost wasn't used for the humanoid's hands and feet to allow them move more freely. The angular velocities were not included in the cost computation. The distances d_x and d_z are the x- and z-component deviations of the center of mass from the feet mean point.

The controls sampled by FDI-MCTS are reference angle vectors $\boldsymbol{\alpha}_{\text{ref}}$. The control commands passed to the physics engine are reference angular velocities computed using a P-controller as $\boldsymbol{\omega}_{\text{ref}} = K_{\text{P}}(\boldsymbol{\alpha}_{\text{ref}} - \boldsymbol{\alpha})$. These are passed to the simulator joint motors. K_{P} was set to be 10. The convergence results were produced by an Intel Xeon E3-1230 V2 CPU processor and 16 GB of RAM. The parameters used in the learning algorithms are shown in Table 1. The parameters of the controllers are presented in Table 2. The sampling distribution contains also the acceleration minimization term mentioned in Section 3.1. The term's covariance matrix is Σ_{acc}, and mean $\boldsymbol{a}_{\text{acc}} = \boldsymbol{\alpha} + \frac{1}{K_{\text{P}}}\boldsymbol{\omega}$, i.e., the mean corresponds to reference angles that would maintain the current angular velocities.

Control signal, i.e. the reference angle α_{ref} was constrained to be within the joint limits, and the controls were sampled from a clipped Gaussian distribution obeying these limits. The simulator joint motors had the torque limit of 80.0 Nm.

The character had also the so called "recovery mode". If the FDI-MCTS detects that the character is going to fall, it uses this recovery mode in which the joint torque limit is increased to 400.0 Nm and the spring constant K_{P} is increased five fold. In the recovery mode also the sampling distributions' standard deviations are multiplied by five. These changes make the character use larger and stronger movements. In addition to the previously introduced variables, the character's state vector also has an indicator variable telling whether or not the character is in the recovery mode.

4.2 Quantitative evaluation

Figures 3 presents the convergence of our method for the humanoid character with different learning components enabled or disabled. The data shows that the nearest neighbor and neural net learners augment each other; both learners improve results even alone, but their combination yields the best results.

Table 1: The parameters of the learning algorithms.

Online random forests	
Trees in random forest	5
Tree capacity	3,600
Neural network training	
maximum data set size	2000
neural network	5 hidden layers 100 neurons each fully connected ELU non-linearities

Table 2: The parameters of Fixed Depth Informed Monte Carlo Search Tree for each degree of freedom. Here r_{dof} is the range of the degree of freedom. As the covariance matrices are constrained to be diagonal the Σ here denote the matrices' diagonal elements corresponding to the degree of freedom.

$\mathbf{a}_{\text{init}}$	**0**
$\sqrt{\Sigma_{\text{init}}}$	25°
$\sqrt{\Sigma_{\text{diff}}}$	25°
$\sqrt{\Sigma_{\text{variation}}}$	$0.1 r_{\text{dof}}$
$\sqrt{\Sigma_{\text{acc}}}$	10° (Humanoid), 30° (Rest)

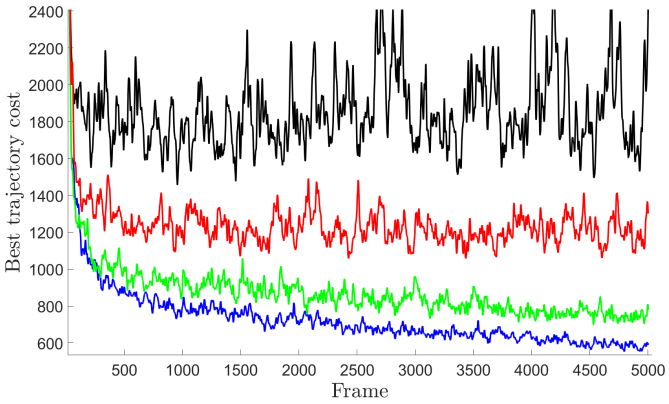


Figure 3: Convergence results when the bipedal character of Figure 4 is walking forward. The figure shows the best found trajectory cost at the given step. The curves are the smoothed medians of 20 test runs for each settings. Both learning types enabled is drawn in blue, using only the neural networks is drawn in green, using only nearest neighbor learners is drawn in red, using only the previous frame to inform is drawn in black.

In producing the graphs, we used a total of 64 sampled FDI-MCTS trajectories. Three trajectories were informed by the nearest neighbor learning when it was used. When neural network learning was present, we used three trajectories with the variation (i.e. multiplying them to the sampling distribution), and three trajectories that used `net` values as they were. Note that these trajectories were still not exactly same due to the pruning and forking. 16 trajectories

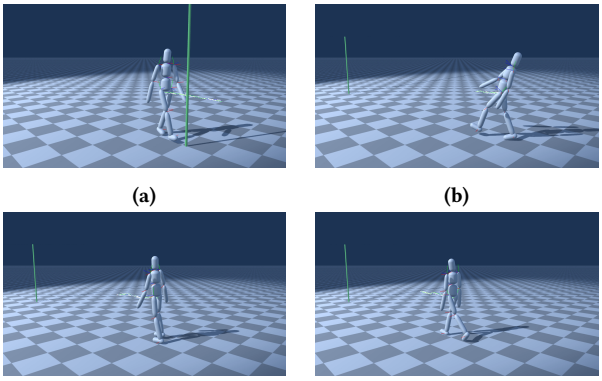


Figure 4: A full 3D humanoid character is walking towards a pole. When the pole is reached it changes location, causing an abrupt change in steering direction. Once trained, the character can produce this behavior in real-time with just 16 sampled trajectories in the Monte Carlo tree search.

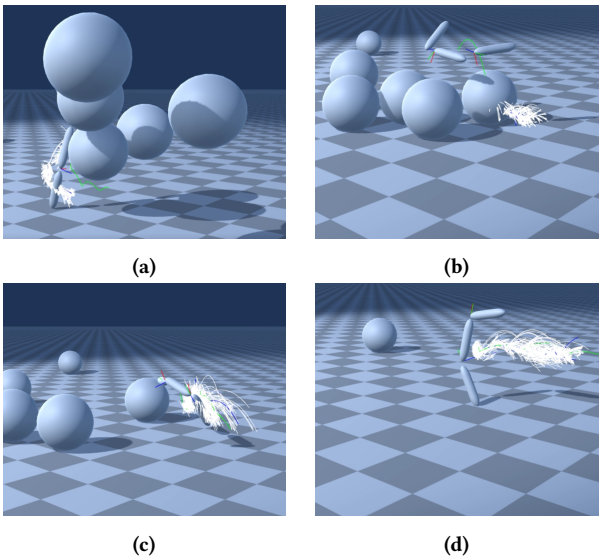


Figure 5: A monoped is thrown at with spheres. It somersaults over the spheres and resumes hopping forward.

were "free", i.e. they did not use even the previous iteration results to inform the sampling.

4.3 Qualitative evaluation

Figures 4 and 5 illustrate the tests performed with the 3D characters. Figure 4 shows a humanoid character walking towards a pole. The target pole changes location periodically, causing an abrupt steering angle change, which is known to be difficult for locomotion controllers. In the video [01:10] we can see how the character stops, changes direction and resumes walking. Figure 5 shows a monoped that tries to walk while heavy spheres are thrown at it. The monoped is sometimes able to dodge the spheres, but might require taking highly acrobatic actions such as flipping over multiple spheres, which is shown in Figure 5 and the supplemental video [01:46]. These kinds of behaviors cannot be implemented with controllers that are designed only for locomotion. On the other hand the recently introduced sampling based model predictive controllers have not demonstrated the ability to produce a stable gait [Hämäläinen et al. 2014, 2015]. The supplemental video shows more results with various characters.

The main takeaway from the experiments is that the combination of the learning algorithms and the Fixed Depth Informed Monte Carlo Tree Search produces stable policies that can be learned even under heavy disturbances. As can be seen at [00:30] and [01:30] in the video, the various characters develop a stable gait in just seconds, and continue to refine the gait in further training. The neural net learner also allows the bipedal humanoid walker to develop a natural hand swing as can be seen at [00:10] in the supplemental video.

Once a good gait is learned, the Monte Carlo tree search produces a stable gait with a much smaller sampling budget. Even with the smaller sampling budget the learning can continue. Our initial sampling budget N was 64 trajectories in all the tests, except for the monoped for which we used 256 trajectories. Simulating the monoped still takes about the same amount of CPU time as the simulation model is simpler. With the full humanoid character and a planning horizon of 1.2 seconds and a simulation time step of 1/30 s, each control iteration takes approximately 70 ms. A stable gait can be achieved in real-time with a sampling budget N of 8 samples, once gait is learned. In the video, "real-time mode" denotes this setting with smaller sampling budgets.

The acceleration minimization term $\mathcal{N}(\mathbf{a}_{\text{acc}}, \Sigma_{\text{acc}})$ makes the movements smooth and prevents the humanoid from walking on its toes as it does without this term as shown in the video at [02:02]. With the term enabled, stable walking emerges much faster. Without the term, gait stabilization takes approximately 2500 frames with the full humanoid.

5 DISCUSSIONS

5.1 Remarks

Before arriving to the present form of the algorithm we observed that mixing a lot of confident information sources was generally a bad idea. If the Gaussian distribution multiplications in Algorithm 1 involved two distributions with covariance matrices close to zero, the performance was clearly bad. This is understandable as the product of narrow Gaussian distributions is a narrow distribution itself. But because the mean is the covariance matrix weighted mean, it is far from either individual distribution's mean thus causing problems if the means of the different distributions differ a lot. It has been observed also by others that averaging over two different actions can yield a poor action even though the individual actions would be good [Peng et al. 2015]. An elaborated version of this would read that mixing two confident and different information sources yields bad actions. This is probably the reason why replacing $\mathcal{N}(\mathbf{a}_{\text{prev}}, \Sigma_{\text{variation}})$ with $\mathcal{N}(\mathbf{a}_{\text{fast}}, \Sigma_{\text{variation}})$ worked better than just multiplying everything together.

An interesting observation was that the algorithm yields subpar results if the fast but precise memory's informing term, i.e.

$\mathcal{N}(\mathbf{a}_{\text{fast}}, \Sigma_{\text{variation}})$, is not overridden by the previous iteration closest sample informing term $\mathcal{N}(\mathbf{a}_{\text{prev}}, \Sigma_{\text{variation}})$, when $\mathbf{s}_{\text{prev}}$ is closer to the current state $\mathbf{s}$ than $\mathbf{s}_{\text{fast}}$. It seems that our way of combining information is a good support in the more explored parts of the space but becomes detrimental in the less explored parts of the state space.

The selection of action parametrization greatly affects performance. Using the P-controller with reference angle as control made the sampler more robust than using target velocities, which was done by Hämäläinen et al. [2015]. The subject has been more extensively studied by Peng and van de Panne [2016]. Our observations confirm their results and we recommend that close attention is paid to the selection of action space.

We used the Manhattan distances (i.e. one-norm) to measure distances. According to our intuition one-norm produces better results than two-norm and is a natural choice for measuring distances in high dimensional spaces. Exploring the effect of different norms in various applications is a widely researched topic and falls outside the scope of this paper.

5.2 Limitations

The different modes of learning allowed us to quickly improve the sampling based model predictive controller's performance and enabled natural gaits for various 3D characters is just seconds. However, the neural networks did not learn to control the 3D characters without using the sampling based model predictive controller. We witnessed the 3D humanoid taking a few steps but the character quickly drifts outside the part of the state space where it was trained and falls. With a 2D walker with similar state and control as those used for the 3D characters, the neural net learns to control walking in under a minute of training as shown in the video [02:45]. If we wish to make the character act with just the neural network without the Monte Carlo tree search, we should develop some measures to address the inevitable drift from the state distribution of the training data. Furthermore, we should address the problem of how to determine whether the control can be given to the neural networks instead of using the MCTS. Also, because our controller uses forward sampling of the dynamics of the system, we obviously cannot perform collaborative tasks that involve taking into account the actions that might be taken by others.

6 CONCLUSIONS

We presented an algorithm that combines sampling-based model-predictive control, a fast but imprecise random density forest learner and a slow but more precise neural network learner. This combination yields a controller that is capable of acting instantly and learning from experience. The capability of acting before learning is desirable, as it shortens the iteration cycle of designing cost functions and adjusting the character models. Our controller also does not require any precomputation or training data beyond what it generates through simulation.

Our algorithm appears capable of synthesizing robust behaviors for a wide range of agents. We have demonstrated 3D locomotion under heavy disturbances for a monoped, a humanoid character and a quadruped. Once trained, locomotion was stable with a significantly smaller sampling budget. Previous sampling based model predictive controllers [Hämäläinen et al. 2014, 2015] have not been demonstrated to produce a stable gait even with the initial sampling budget. Furthermore, learning appears to remove much of the sampling noise that plagues sampling-based controllers, which on the other hand have many other desirable qualities such as ability to stop walking or to recover from falling. Our work takes sampling-based based controllers closer to being a viable option in games and animation, although the sampling budgets required even after learning still provide room for improvement.

ACKNOWLEDGEMENTS

We thank our reviewers for evaluating our paper and for their feedback. We would also like to thank Kourosh Naderi for discussions and his help. This research has been supported by Academy of Finland and TEKES, the Finnish Funding Agency for Innovation (Grant number 305737).

REFERENCES

Paul Bromiley. 2003. Products and Convolutions of Gaussian Probability Density Functions. *Tina-Vision Memo* 3 (2003).

Antonio Criminisi, Jamie Shotton, and Ender Konukoglu. 2012. Decision Forests: A Unified Framework for Classification, Regression, Density Estimation, Manifold Learning and Semi-Supervised Learning. *Foundations and Trends in Computer Graphics and Vision* 7, 2–3 (2012), 81–227.

Thomas Geijtenbeek, Michiel van de Panne, and A. Frank van der Stappen. 2013. Flexible Muscle-Based Locomotion for Bipedal Creatures. *ACM Transactions on Graphics* 32, 6 (2013).

Perttu Hämäläinen, Sebastian Eriksson, Esa Tanskanen, Ville Kyrki, and Jaakko Lehtinen. 2014. Online Motion Synthesis Using Sequential Monte Carlo. *ACM Transactions on Graphics* 33, 4 (2014), 51.

Perttu Hämäläinen, Joose Rajamäki, and C Karen Liu. 2015. Online Control of Simulated Humanoids Using Particle Belief Propagation. *ACM Transactions on Graphics* 34, 4 (2015), 81.

Nikolaus Hansen. 2015. The CMA Evolution Strategy: A Tutorial. (2015). https://www.lri.fr/~hansen/cmatutorial.pdf https://www.lri.fr/~hansen/cmatutorial.pdf.

Auke Jan Ijspeert, Jun Nakanishi, Heiko Hoffmann, Peter Pastor, and Stefan Schaal. 2013. Dynamical Movement Primitives: Learning Attractor Models for Motor Behaviors. *Neural Computation* 25, 2 (2013), 328–373.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. *International Conference on Machine Learning* (2015).

Lucas Kovar, Michael Gleicher, and Frédéric Pighin. 2002. Motion Graphs. In *ACM Transactions on Graphics*. ACM.

Yongjoon Lee, Kevin Wampler, Gilbert Bernstein, Jovan Popović, and Zoran Popović. 2010. Motion Fields for Interactive Character Locomotion. In *ACM Transactions on Graphics*. ACM.

Sergey Levine and Vladlen Koltun. 2013. Guided Policy Search. In *International Conference on Machine Learning*.

Weiwei Li and Emanuel Todorov. 2004. Iterative Linear Quadratic Regulator Design for Nonlinear Biological Movement Systems. In *International Conference on Informatics in Control, Automation and Robotics*.

Libin Liu, Michiel van de Panne, and KangKang Yin. 2016. Guided Learning of Control Graphs for Physics-Based Characters. *ACM Transactions on Graphics* 35, 3 (2016).

Libin Liu, KangKang Yin, and Baining Guo. 2015. Improving Sampling-Based Motion Control. In *Computer Graphics Forum*, Vol. 34. Wiley Online Library, 415–423.

Libin Liu, KangKang Yin, Michiel van de Panne, Tianjia Shao, and Weiwei Xu. 2010. Sampling-Based Contact-Rich Motion Control. *ACM Transactions on Graphics* 29, 4, Article 128 (July 2010), 10 pages. DOI: https://doi.org/10.1145/1778765.1778865

Z. Manchester and S. Kuindersma. 2016. Derivative-Free Trajectory Optimization with Unscented Dynamic Programming. In *Conference on Decision and Control*. 3642–3647. DOI: https://doi.org/10.1109/CDC.2016.7798817

David Mayne. 1966. A Second-Order Gradient Method for Determining Optimal Trajectories of Non-Linear Discrete-Time Systems. *International Journal of Control* 3, 1 (1966), 85–95.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy P Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. *arXiv preprint arXiv: 1602.01783* (2016). https://arxiv.org/abs/1602.01783

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, and others. 2015. Human-Level Control Through Deep Reinforcement Learning. *Nature* 518, 7540 (2015), 529–533.

Igor Mordatch, Martin de Lasa, and Aaron Hertzmann. 2010. Robust Physics-Based Locomotion Using Low-Dimensional Planning. *ACM Transactions on Graphics* 29, 3 (2010).

Igor Mordatch, Kendall Lowrey, Galen Andrew, Zoran Popovic, and Emanuel V Todorov. 2015. Interactive Control of Diverse Complex Characters with Neural Networks. In *Advances in Neural Information Processing Systems*. 3132–3140.

Marius Muja and David G. Lowe. 2014. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36 (2014).

Xue Bin Peng, Glen Berseth, and Michiel van de Panne. 2015. Dynamic Terrain Traversal Skills Using Reinforcement Learning. *ACM Transactions on Graphics* 34, 4 (2015), 80.

Xue Bin Peng, Glen Berseth, and Michiel van de Panne. 2016. Terrain-Adaptive Locomotion Skills Using Deep Reinforcement Learning. *ACM Transactions on Graphics* 35, 4 (2016).

Xue Bin Peng and Michiel van de Panne. 2016. Learning Locomotion Skills Using DeepRL: Does the Choice of Action Space Matter? *arXiv preprint arXiv: 1611.01055* (2016). http://arxiv.org/abs/1611.01055

Alexander Pritzel, Benigno Uria, Sriram Srinivasan, Adrià Puigdomènech, Oriol Vinyals, Demis Hassabis, Daan Wierstra, and Charles Blundell. 2017. Neural Episodic Control. *arXiv preprint arXiv:1703.01988* (2017).

Joose Rajamäki, Kousrosh Naderi, Ville Kyrki, and Perttu Hämäläinen. 2016. Sampled Differential Dynamic Programming. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*. 1402–1409. DOI: https://doi.org/10.1109/IROS.2016.7759229

Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. 2017. Evolution Strategies as a Scalable Alternative to Reinforcement Learning. *arXiv preprint arXiv:1703.03864* (2017).

Stefan Schaal. 2006. Dynamic Movement Primitives – A Framework for Motor Control in Humans and Humanoid Robotics. In *Adaptive Motion of Animals and Machines*. Springer, 261–280.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, and others. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529, 7587 (2016), 484–489.

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. 2014. Deterministic Policy Gradient Algorithms. In *International Conference on Machine Learning*.

Richard Sutton and Andrew Barto. 1998. *Reinforcement Learning: An Introduction*. Vol. 1. MIT Press.

Yuval Tassa, Nicolas Mansard, and Emo Todorov. 2014. Control-Limited Differential Dynamic Programming. In *IEEE International Conference on Robotics and Automation*. IEEE, 1168–1175.

Hado Van Hasselt and Marco A Wiering. 2007. Reinforcement Learning in Continuous Action Spaces. In *IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*. IEEE, 272–279.

Miomir Vukobratović and Branislav Borovac. 2004. Zero-Moment Point – Thirty Five Years of Its Life. *International Journal of Humanoid Robotics* 1, 01 (2004), 157–173.

Kevin Wampler and Zoran Popović. 2009. Optimal Gait and Form for Animal Locomotion. *ACM Transactions on Graphics* 28, 3, Article 60 (July 2009), 8 pages. DOI: https://doi.org/10.1145/1531326.1531366

Jack M. Wang, David J. Fleet, and Aaron Hertzmann. 2010. Optimizing Walking Controllers for Uncertain Inputs and Environments. *ACM Trans. Graph.* 29, 4, Article 73 (July 2010), 8 pages. DOI: https://doi.org/10.1145/1778765.1778810

KangKang Yin, Kevin Loken, and Michiel van de Panne. 2007. SIMBICON: Simple Biped Locomotion Control. *ACM Transactions on Graphics* 26, 3 (2007), Article 105.

Marvin Zhang, Zoe McCarthy, Chelsea Finn, Sergey Levine, and Pieter Abbeel. 2016. Learning Deep Neural Network Policies with Continuous Memory States. In *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 520–527.