Searching for a Better Policy Architecture for the OpenAI Bipedal Walker Environment

Rustam Galljamov [1] **Boris Belousov** [1] **Michael Lutter** [1]

Abstract

Deep Learning yielded multiple neural architectures effectively targeting the specifics of input data in different domains and achieved breakthrough results in the past decade. Deep Reinforcement Learning brought these successes to the Reinforcement Learning setting and raised big hopes to solve previously unreachable problems, many of which are coming from the field of robotics. An architecture specialized to the properties of input data from robotic systems indeed was not developed so far. In this work, we investigate the question if sparse architectures are better suited for robot input data compared to the default choice of a fully connected Multilayer Perceptron (MLP). We find hints of an optimal sparsity in an MLP and reduce the size of the policy network by 85% retaining the desired performance.

1. Introduction

Deep Reinforcement Learning (deep RL) introduces deep neural networks to the Reinforcement Learning (RL) setting (Li, 2018). This combination yielded agents outperforming humans in Atari Games (Mnih et al., 2015) and raised the hope to manage the complexity of bipedal locomotion in robotic systems (Peng et al., 2018; Xie et al., 2018).

Deep Learning (DL) began with the introduction of the Multilayer Perceptron (MLP) that still plays an important role today. The most prominent successes of DL in the last decade indeed were achieved by introducing neural architectures considering the specifics of data in a given domain and thus guiding the networks to learn better features:

- To deal with images, Convolutional Neural Networks (CNNs) introduce layers consisting of many identical copies of a neuron that apply different filters to the input image and preserve its spacial structure for subsequent layers (Sermanet et al., 2012).
- Recurrent Neural Networks (RNNs) brake up the pure forward structure of vanilla networks and utilize a memory concept to deal with time series data (Mikolov et al., 2010).

These architectures found their way in deep RL, e.g. by using CNNs as feature extractors for visual input data (Mnih et al., 2015), but deep RL is also applied to robotics and there is no established architecture targeting at the specifics of sensory data in robotic systems.

Some approaches introduce strong inductive biases in architecture design by considering the kinematic structure of the robot to control. Gaier & Ha (2019) successfully applied Neural Architecture Search to the OpenAI Bipedal Walker Environment, a simulated 2D bipedal robot (Brockman et al., 2016). The found architecture show clear characteristics

1. high sparsity
2. each action is a function of its own unique features
3. presence of many residual connections

These observation raise the question whether these properties are specific to the utilized search algorithm and the considered architecture building blocks or general characteristics of an architecture for Reinforcement Learning applied to robotics. In this paper, we would like to initiate the investigation of this question and focus on the property of high sparsity. Therefore, we implement and evaluate multiple versions of sparse architectures for the policy network and compare them with the default choice of a fully connected MLP. Our goal is to find an architecture that will beat the vanilla network on the Bipedal Walker Environment considering performance, stability and smoothness of the learned controllers.

2. Related Work

Neural Architecture Search. One possibility to fulfill our goal of finding a sparse architecture for the policy network

[1]Department of Computer Science, Technische Universität Darmstadt, Germany. Correspondence to: Rustam Galljamov <rustam.galljamov@gmail.com>.

Robot Learning Integrated Project

that will outperform the fully connected Multilayer Perceptron is to use algorithms dedicated to find specific architectures. The field summing up those approaches is called Neural Architecture Search, or NAS for short (Zoph & Le, 2016).

The search consists of three main stages. At the beginning, the search space is defined by choosing architecture building blocks, possible connections between the components and the desired properties of the architecture. Next, a search strategy has to be chosen. The process of finding an architecture usually starts with the simplest possible architecture building it up iteration by iteration in a bottom-up fashion. The two main approaches for choosing the modification of the architecture for the next iteration are evolutionary and Reinforcement Learning algorithms. In the former case, several possible mutations are produced, in the latter a policy provides a distribution over possible next adjustments. The last step of an iteration is the performance estimation. All candidate architectures are tested on the task the architecture is optimized for. When searching an architecture for machine learning applications, the architectures would optimally need to be trained and evaluated on unseen data. Most of the time, this procedure would be too costly computationally and in terms of required time. Therefore, recent research is targeted at reducing the cost of evaluation (Elsken et al., 2018).

The high computational requirements in applying NAS for our purposes are a strong argument against this approach (Elsken et al., 2018). Moreover, (Yu et al., 2019) question the supremacy of this approach over Random Search in multiple applications.

Fortunately, (Gaier & Ha, 2019) overcame these barriers and successfully applied NAS to a Reinforcement Learning task, one of which was the *Bipedal Walker* from OpenAI Gym (Brockman et al., 2016). Building on the NEAT algorithm that simultaneously updates the weights and the architecture at each iteration (Stanley & Miikkulainen, 2002) and drastically reducing the cost of performance estimation, their algorithm found an architecture that performed on par with the traditional fully connected MLP on several deep RL tasks. The authors' motivation was to find architectures that even without training the weights would reach high performance. Therefore the architectures were evaluated at each iteration by setting all weights in the network to a fixed weight from the sequence $[-2, -1, -0.5, 0.5, 1, 2]$ and averaging the results. This approach gave the yielded architectures the name of Weight Agnostic Neural Networks (WANNs).

The architecture found for the *Bipedal Walker* environment show clear characteristics: The networks are very sparse with most neurons combining only a single or small fraction of possible inputs. Further, it is worth noticing the high amount of residual connections and the fact that each output neuron combines a distinct set of features in contrast to the outputs in a fully connected network where all actions consider the same inputs. Each of these observations make sense in the context of the considered task and have a sound explanation.

Throughout this work, we put our focus on the property of high sparsity. One intuition of neural networks is their ability to learn meaningful high level features by combining the input data. In contrast to images, where individual pictures bear no useful information, the kinematics and forces of individual joints as well as other sensor output in a robotic system is an important component to describe the state of the robot and choose an appropriate action. Therefore, it is expected that high level features can be built up by combining only a few inputs which corresponds to a high sparsity of the policy network.

Sparse Neural Networks. The most common way to introduce sparsity in a neural network is referred to as network pruning (LeCun et al., 1990). The main goal of approaches in this area is often the increase of memory and computational efficiency, e.g. to deploy complex networks on embedded systems (Han et al., 2015). The pruning happens after training following different heuristics. An effective and often used approach is called Magnitude Pruning (Guo et al., 2016). Here, the weights with the smallest magnitude are considered to be less important in the considered model and are thus removed.

Following this and other pruning approaches the network size is reported to be reduced by up to 90% without significantly reducing performance (Frankle & Carbin, 2018). An increase of performance after pruning was not yet reported reducing the direct relevance of this field to our goal of finding a better performing sparse architecture. Furthermore, retraining the architecture after pruning by far missed the performance of the original network until (Frankle & Carbin, 2018) presented the Lottery Hypothesis. They found out that a pruned network can still be trained to the same or even higher performance compared to its dense counterpart when the remaining weights are initialized with exactly the same values as for the original training. This finding raises the hope of existing sparse architectures that can be trained from scratch and achieve a desired performance.

Attention Mechanisms. Talking about sparsity as remaining only a fraction of possible input connections compared to the fully connected network suggests the question of how to choose the values to remain and prune. Introduced in the context of encoder-decoder networks for sequence modelling, attention mechanisms do exactly that (Vaswani et al., 2017b). Given all the hidden states of a recurrent encoder at different timesteps of the input sequence, the few input states relevant for each position of the decoder output se-

quence are chosen by learning a distribution over the input importance. This approach introduces new parameters into the model but reduces the number of considered inputs and is therefore worth investigating in our approach of finding performant sparse architectures.

3. Methods

3.1. Sparsity Definitions

Official Definition of Sparsity. The official definition of sparsity from graph theory is the fraction of zero values in a matrix.

Our Definition of Sparsity. Our intuition of neural networks learning high level features by combining raw input data asks for a slight modification of the official definition. A single neuron y in the hidden layer combines the n inputs $\vec{x}$ in a weighted sum s before applying the activation function ϕ, resulting in the following simplified equation (bias term ignored):

$$s(\vec{x}) = \vec{w}\vec{x} = \sum_{i=1}^{n} w_i x_i = w_1 x_1 + ... + w_n x_n$$

$$y(\vec{x}) = \phi(s)$$

By normalizing the input data and choosing the tanh activation function in each hidden unit, the inputs to each layer are guaranteed to be in a similar range of about $[-1 : 1]$. Therefore, the absolute value of the i'th weight $|w_i|$ determines the contribution of the i'th input to the constructed high level feature.

Due to the high information in individual sensor outputs of a robotic system, we expect each neuron to combine only a few of the inputs to form a better feature. In this context, we define sparsity of a single hidden unit as the fraction of not considered inputs in a hidden unit. By taking the average over all units in a hidden layer, we get the sparsity of the whole layer. An input x_i is defined to be not considered, when its corresponding weight w_i is below 10% of the maximum input weight of the neuron.

Explicit and implicit sparsity. Another distinction we make in the paper is that of explicit and implicit sparsity. In our search of a better architecture through sparsity we consider different approaches. A network is explicitly sparse, when a fraction of possible connections between layers is absent by construction. The sparsity of this kind of networks cannot be decreased during training but can be increased by setting the remaining weights to a negligibly small value. The minimal sparsity is a feature of the architecture.

Implicit sparsity is achieved by reducing the weights of a fully connected neural network to an absolute value that can be disregarded. Having all possible connections, the minimal sparsity of 0% can be achieved. During training the weights get updated and with them the sparsity at that point in time.

3.2. Experimental Setup

Algo, Env, Hypers, Cluster, Evaluation Metrics

The final search for a policy architecture optimized to deal with sensory data of robotic systems as inputs should be independent of an algorithm, architecture and environments. This indeed would drastically exceed the scope of the integrated project in every dimension.

Within this work, we compare multiple policy network architectures used in a single algorithm, applied to the same single Reinforcement Learning (RL) environment.

The algorithm chosen for our investigation is called Proximal Policy Optimization (PPO) and was presented by (Schulman et al., 2017). PPO is considered to be among the most popular deep RL algorithms *due to its simplicity and high reproducibility* (Li et al., 2019; Hämäläinen et al., 2018). A second reason for this choice is presented in the following.

We use the implementation of the algorithm from the open source Python library *Stable-Baselines* (Hill et al., 2018). In addition to the high quality algorithm implementations the authors provide a set of optimized hyperparameters for multiple pairs of environments and algorithms (Raffin, 2018). Our environment of choice, the *Bipedal Walker - v2* from the OpenAI Gym collection (Brockman et al., 2016), was also target of the hyperparameter optimization for different algorithms. The results of this optimization put PPO again at the best position. The Twin Delayed DDPG (TD3) algorithm (Fujimoto et al., 2018) required a significantly bigger policy architecture compared to PPO. Finally, Soft Actor-Critic (Haarnoja et al., 2018) was used with a custom policy network, so our baselines would lack optimized hyperparameters. Table X summarizes the optimized parameters that were used in all our evaluations.

The Bipedal Walker is a simple RL environment simulating a two-dimensional footless bipedal robot. The goal is to torque control the knee and hip actuators of the robot and walk over a flat slightly uneven terrain. The reward is proportional to the traveled distance with small punishments for high motor torque. The inputs consist of twelve joint and center of mass kinematics as well as two Boolean flags indicating ground contact for each leg. The additional 10 LIDAR measurements, simulating the vision part of the robot, were removed to focus on kinematic and kinetic inputs, resulting in overall 14 inputs. An episode ends when the agent reaches the end of the environment or falls down. The latter case results in a high negative reward.

3.2.1. Evaluation Metrics

Each architecture is evaluated within six runs using different seeds. The average score of an architecture is determined by averaging the results of individual runs. All evaluations are executed in parallel on the Lichtenberg-Cluster. Per run, the agent is trained for 6 Million steps. After training, each agent is observed for 100 episodes, collecting rewards and action trajectories for further investigations. We calculate all evaluation metrics per run and average them to get the mean architecture score. Due to the random generated terrain in each episode, the seeds of the environment are hold constant between all architecture evaluations in order to ensure a fair comparison.

In the following, we present

Performance. The performance of an agent is measured as the mean return over 100 episodes. The episode return is also considered in case of an early termination of the episode due to the agent falling.

Learning Curves. The learning curves show the mean reward of a batch which are collected during the training. This gives us a performance estimate of the current policy after each update. Moreover, the learning curve can be used to derive the sample efficiency and the convergence behavior of the agent.

Stability. A fall is the worst case scenario in a bipedal walking robot. Therefore, we count the number of falls during the 100 evaluation episodes as a metric of instability. As the falls are most probably caused by the randomness of the ground shape in each episode the inverse of this metric corresponds to the robustness of the learned controller to terrain changes.

Policy Smoothness. We estimate the policy smoothness by taking the mean smoothness over all 4 motor trajectories and 100 evaluation episodes. (Balasubramanian et al., 2015) compare different approaches for estimating smoothness of trajectories and among others propose the Log Dimensionless Jerk (LDLJ), which we use in our work with the following formula, where $v(t)$ is the first derivative of the considered trajectory, t_1 and t_2 the start and end times of the movement and v_{peak} the maximum derivative on the examined interval:

$$\text{LDLJ} = ln|\frac{(t_2 - t_1)^5}{v_{peak}^2} \int_{t_1}^{t_2} |\frac{d^2 v(t)}{dt^2}|^2 dt|$$

Sparsity Evolution. With the sparsity being the main property of the proposed architectures to investigate, we save the weight matrices of each hidden layer every 200 000 timesteps during training and calculate their sparseness, resulting in 30 recordings per architecture. This procedure is indeed only relevant for implicitly sparse architectures as the explicit sparsity can be estimated by a constant value. By plotting the sparsity of each recording over the training time we observe how the sparsity change over training time, which we call the evolution of sparsity. The calculation of sparsity is carried out according to our definition in 3.1.

3.3. Architectures

In this section we present the architectures considered in our evaluation. After starting with the fully connected Multilayer Perceptron (MLP) as our baseline, we take the logic next step of introducing implicit sparsity by using L1-Regularisation (Liu et al., 2019). Next, we investigate the effect of sparse weight initialization by setting a fraction of the weights to zero before training begins. After that, explicit sparsity becomes the target of our attention where we combine the input features in a binary tree like architecture with different number of considered inputs per hidden unit. This approach opens the question of how to choose the inputs to be combined to form a better feature which leads us to the introduction of attention mechanisms to automatically detect the best input tuples for each hidden unit.

3.3.1. Implicit Sparsity Architectures

Fully Connected MLP. Our search for a better architecture starts with the baseline, the fully connected Multilayer Perceptron (MLP) with two hidden layers, each containing 64 units. The input layer consists of 14 inputs and we have four outputs, having $14 \times 64 + 64 \times 64 + 64 \times 4 = 5248$ parameters. The number of hidden units remains constant for all architectures with implicit sparsity.

L1 Regularisation. Fully connected MLPs have a high number of parameters and thus tend to overfit to the training data. As the training and test environment in Reinforcement Learning (RL) are often the same, especially in simulation studies, this hasn't been an issue so far in deep RL. In other areas of Machine Learning this situation is handled using regularization techniques. (Liu et al., 2019) recently evaluated different regularization methods in RL application. L1 Regularisation is implemented by extending the loss function with a term punishing non zero weights. While focusing on improving generalization, this effect can be interpreted as increasing the implicit sparsity of the weight matrices in the hidden layers, which is subject of our investigation.

Sparse Weight Initialization. Another approach to investigate the effect of implicit sparsity in fully connected MLPs is to initialize the weight matrices of both hidden layers sparsely. After initializing the weight matrices to be orthogonal as in earlier, in the next step a fraction of random chosen weights is set to zero. Two settings are evaluated within this approach: setting 70 and 50% of weights to zero after initialization. An important final note is that all values

remain trainable after initialization and thus the implicit sparsity can be changed during training.

3.3.2. Explicitly Sparse Architectures

In this subsection we describe architectures that no longer use fully connected hidden layers combining all inputs in a hidden unit but instead are constructed to combine only a few of the inputs.

Binary Tree Architectures. The search for a better architecture with explicit sparsity starts at the extreme case of only combining two inputs in each hidden unit, hence the name *binary*. The hypothesis behind this approach is that a high level feature can be constructed by combining only a small fraction of inputs. Simultaneously, having only two features in each hidden unit a residual connection, propagation of one of the inputs, can easily be learned during training by setting a single weight to zero. In a fully connected setting 13 weights would need to be set to zero during training to accomplish the same effect.

As a high level feature might require more that two inputs, the next hidden layer combines units having a disjoint set of input features. This allows to combine 2^h inputs in h hidden layers. We start with three hidden layers, expecting that no high level feature would require more that 8 inputs.

To ensure a fair comparison, the number of hidden inputs per layer is chosen to be 70, only 6 units more compared to previous architectures. An additional hidden layer does not contradict the fairness due to the significantly smaller parameter count of 916 compared to 5248 in fully connected architectures. With 14 inputs, 7 units in the first hidden layer are enough to have every input combined. To achieve a comparable size of hidden layers, each input tuple is therefore combined in 10 units, resulting in overall 70 units in the first hidden layer. The second hidden layer then combines the i'th and the $(i + 10)$'th input to ensure combining only features with disjoint input sets.

A variation of the binary tree architecture is one that is combining 4 inputs in each unit of the first hidden layer and 3 inputs in the second hidden layer. This way up to 12 inputs are combined in order to build a high level feature.

Even these approaches achieve high sparsity, the results are expected to depend on the order of the inputs as we combine subsequent inputs in each unit. But a good feature in the first hidden layer might need to combine the first and 8th input, which is only accomplished in the second hidden layer. To overcome this barrier, we introduce attention mechanisms.

3.3.3. Architectures with Attention

Combining only a fraction of inputs instead of all as in the fully connected settings raises the question which features to combine in a hidden unit. Where an expert might be able to choose groups of features that have to be combined together we let the network choose the inputs itself by implementing an attention mechanism (Vaswani et al., 2017b). Every input of every hidden unit gets a weight between 0 and 1 signalizing how important this input for the formation of a feature is. This can be simply accomplished in a vectorized fashion for the whole layer by introducing an attention matrix $\vec{A}$ with the shape of the weight matrix $\vec{W}$ that is pointwise multiplied with $\vec{W}$. Without additional constraints the pointwise multiplication of two matrices could be summed up into a single one giving no room for an interpretation of $\vec{A}$ as an attention matrix. Interested in the relevant inputs to form a hyper-feature we want $\vec{A}$ to have zero values for features to ignore (no attention) and a one for required features (full attention. We propose two different ways to transform the matrix in order to achieve the desired properties.

Softmax Attention. The concept of attention is best known in the context of encoder-decoder networks for sequence modeling (Vaswani et al., 2017a). Here, the encoder states at different timesteps are combined into a single state representation. A softmax function is used to distribute the attention across all states. The straightforward implementation of attention for our purposes therefore was to apply a softmax function to the attention weights of each hidden unit. This corresponds to the row-wise application of the softmax function to the attention matrix $\vec{A}_w$.

This approach introduces three new hyperparamters. The obvious one is the initialization of the attention matrix. We set all values of the attention matrix to zero. This way all inputs have equal attention before the training and even small changes of the attention weights during Backpropagation can significantly change the distribution of attention. Expecting to have only a few relevant inputs, a spiky distribution is preferred which can be achieved by scaling the matrix before applying the softmax function. This approach is known in the literature as temperature τ (He et al., 2018). Finally, having a distribution, the attention over possible inputs sums up to one. This means that in average, a single input is considered in a hidden layer. To allow for more considered inputs, we scale the attention matrix after applying the softmax function with the number of expected number of inputs required to form a high level feature, denoted as k. The following equations gives an overview over all parameters and computations for the i'th hidden unit, corresponding to the i'th column of $\vec{A}$:

$$\vec{A}_i = k\ softmax(\frac{1}{\tau}\vec{A}_{w,i})$$

Remoid Attention. Another approach to turn $\vec{A}_w$ into an attention matrix with ones for full attention and zeros for inputs to ignore is to transform each element with a sigmoid

function σ:

$$\vec{A} = \sigma(\vec{A_w}) \;\; with \;\; \sigma(x) = \frac{1}{1+e^{-x}}$$

To reduce the computational costs the sigmoid function is approximated as a sum of two ReLU functions, giving it the name *remoid* ρ. With the variable slope α we further get a hyperparameter for controlling how fast an attention weight converges to 0 or 1:

$$\rho(x) = relu(\alpha x) - relu(\alpha x - 1)$$

The attention weight matrix $\vec{A_w}$ is then initialized with $\frac{1}{\alpha}$ resulting in all attention weights being at 0.5 during initialization and thus exactly in the middle between 0 and 1.

4. Results

We investigated multiple approaches to introduce sparsity in the policy network architecture and their effect on several evaluation metrics. Our goal was to beat the fully connected Multilayer Perceptron (MLP) on the Bipedal Walker Environment, which in our experimental setup with fixed hyperparameters optimized for the MLP architecture turned out to be a hard to beat baseline. Having not significantly outperformed it, most architectures reached the same level of performance even without hyperparameter tuning.

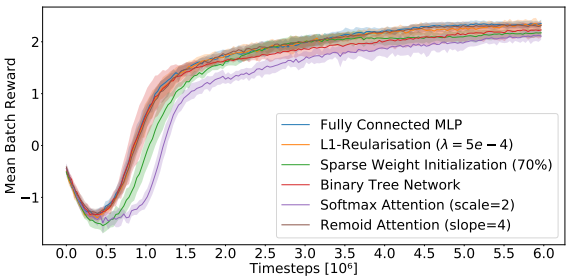

Figure 1. Learning Curves of evaluated architectures.

Learning Curves. Figure 1 shows the high similarity of learning curves for different approaches. Here, we see that sparse initialization of the weight matrices delays the steep increase in mean batch reward. The delay is proportional to the percentage of the weights set to zero during initialization. The learning curve steepness of architectures with the Softmax attention mechanism is proportional to the average number of considered inputs in a hidden unit as can also be seen in figure 1. The sparsely constructed binary tree architectures reduce the size of the policy network by up to 85% with no significant loss in performance and other evaluation metrics. The corresponding learning curve indeed shows the same steep slope as the MLP for the first one million training steps but decreases thereafter.

The performance and policy smoothness plots are not displayed due to insignificant differences between compared approaches. Figure 2 shows a slight tendency to the important role of initialization and the potential of the Remoid Attention, both having learned to control the robot with the minimal probability of a fall.

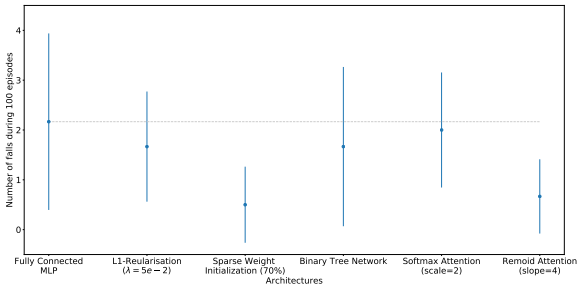

Figure 2. Comparison of Instability of the learned controller measured by the number of falls in 100 episodes.

Sparsity Evolution Comparison. The main insight within our comparison is the convergence of sparsity to the same value across different approaches, shown in figure 3. Starting with different sparsities at the training's beginning, all architectures incorporating implicit sparsity converge to a similar value at the training's end. This value seems to dependent on the size of the hidden layer and is about 20% for the first hidden layer with 896 parameters and about 30% for the second one with a size of 4096 connections. At the same time, the sparsity of the original fully connected MLP remains almost constant during training. Introducing L1-Regularization slightly increases the sparsity during the training approaching the same reported values but does not reach these.

5. Discussion

Deep Reinforcement Learning (RL) introduced deep network architectures into the reinforcement learning setting. While the successful data specific architectures like Convolutional Neural Networks (CNNs) and Recurrent Networks (RNNs) found their way into RL, no architectures were developed targeting at the specifics of input data from robotic systems. Inspired by the high sparsity of Weight Agnostic Neural Networks (WANNs) on a walking robot environment we investigated different approaches to introduce sparsity to the policy architecture. Our focus laid on exploring different methods with the goal to find architectures that are worth to further exploit.

We implemented several mechanisms to encourage high sparsity in the fully connected Multilayer Perceptron (MLP). Against our expectation, all architectures with implicit sparsity converged to the same small level of sparsity. Fully connected MLPs seem to have an optimal sparsity that is

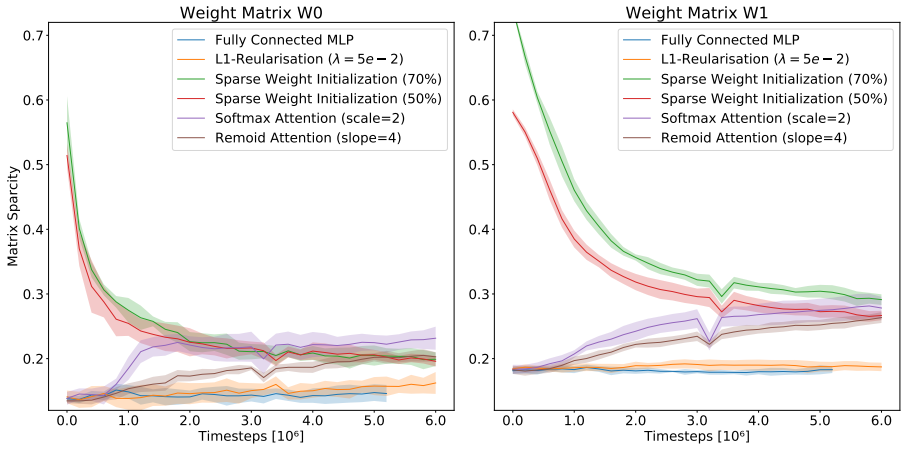

Figure 3. Sparsity Evolution of both hidden layers over training time for all compared architectures.

most probably specific to the considered RL environment. We believe, this insight should be further investigated and might be beneficial in tuning parameters of attention mechanisms and guide the pruning process of networks.

Interestingly, a similar performance can be achieved with significantly sparser architectures with only 15% of weights, when connections are dropped during architecture construction and cannot be recovered during training. This means that up to 85% of memory taken by the policy network can be saved without noticing a drop in the performance. Similar and even stronger numbers are reported by the network pruning community (Frankle & Carbin, 2018). The much lower number of parameters is expected to decrease overfitting and better generalize to changes in the environment. This property is especially important for the transition from simulation to the real robotic systems. One explanation is that combining only a few inputs to form a better feature is enough to reach a reasonable performance but having a fully connected structure the additional information can still be used for improvement. The learning curves in figure 1 support this claim. The shape of the curves of the fully connected MLP and the Binary Tree architecture is almost identical for the first 20% of the training. Thereafter, the dense network proceeds with a slightly higher slope which could be due to its higher capacity. Another explanation could lie in the properties of used activation functions. (Gaier & Ha, 2019) introduced a notion of symmetry and periodicity by allowing the absolute and trigonometric functions. This allows to capture relevant patterns with a much smaller number of parameters and thus sparser networks.

The highly similar evaluation metrics between architectures with implicit sparsity and the fully connected MLP can be explained by the fact that most architectures tend to converge to a dense network and do not significantly differ from the baseline. The preferred sparsity, which we expect to be dependent on the environment, is indeed bigger compared to the fully connected setting. This promises a room to save memory usage and increase the calculation speed, which can be significant for much bigger networks.

Our last point to discuss considers the usage of hyperparameters beyond the architecture. During all evaluations the hyperparameters of the algorithm and optimizer have been hold constant, changing only the architecture. Indeed, the used hyperparameters were chosen from the *Stable Baseline RL Zoo* (Raffin, 2018), which were optimized for the fully connected policy architecture. Having reached almost equal and often slightly better results with other architectures without tuning the hyperparameters shows the potential of the investigated architectures.

6. Limitations

The hyperparameters used during all evaluations were optimized for the combination of the fully connected MLP and the targeted environment. That results in a hard to beat baseline, but more importantly do not allow the considered architectures to show their full potential. Even the additional hyperparameters introduced with different architectures were only tuned by hand on a few runs instead of using optimization techniques like Bayesian Optimization or Random Search. Several proposed architectures are tested in their first or second version and have known areas of improvement. The attention mechanism for example is expected to be improved by utilizing a regularization punishing values that are away from one or zero, corresponding to full and no attention for an input.

The evaluation on a single environment can of course be only a first small step on the way to finding a better policy architecture targeted on the usage with robotic systems. In addition, the considered environment only supports a few

properties of state of the art physics engines like *MuJoCo* (Todorov et al., 2012). An example is the lack of friction and damping in the joints.

7. Conclusion

Following our search for a policy architecture that would outperform the default fully connected Multilayer Perceptron on the Bipedal Walker Environment we explored multiple architectures. Inspired by the high sparsity in Weight Agnostic Neural Networks (WANNs) we introduced several approaches to reduce the density of a neural network. This inspiration was backed by our intuition of deep networks combining the inputs to form high level features and the assumption that robotic input data has a high degree of information. Therefore, we expected only a few inputs needed to be combined to form good hyperfeatures which is given in sparsely connected architectures. To our surprise indeed, a all network with learnable sparsity always converged to the same small sparsity showing a clear preference for dense solutions. However in line with our expectations, we found architectures that combined only 2-4 inputs in each hidden layer, thus reduced the size of the policy network by up to 85% and have not reduced the performance compared to the fully connected baseline.

References

Balasubramanian, S., Melendez-Calderon, A., Roby-Brami, A., and Burdet, E. On the analysis of movement smoothness. *Journal of neuroengineering and rehabilitation*, 12 (1):112, 2015.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym. *ArXiv*, abs/1606.01540, 2016.

Elsken, T., Metzen, J. H., and Hutter, F. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018.

Frankle, J. and Carbin, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

Fujimoto, S., Van Hoof, H., and Meger, D. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.

Gaier, A. and Ha, D. Weight agnostic neural networks, 2019.

Guo, Y., Yao, A., and Chen, Y. Dynamic network surgery for efficient dnns. In *Advances in neural information processing systems*, pp. 1379–1387, 2016.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018. URL http://arxiv.org/abs/1801.01290.

Hämäläinen, P., Babadi, A., Ma, X., and Lehtinen, J. PPO-CMA: proximal policy optimization with covariance matrix adaptation. *CoRR*, abs/1810.02541, 2018. URL http://arxiv.org/abs/1810.02541.

Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pp. 1135–1143, 2015.

He, Y.-L., Zhang, X.-L., Ao, W., and Huang, J. Z. Determining the optimal temperature parameter for softmax function in reinforcement learning. *Applied Soft Computing*, 70:80–85, 2018.

Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., and Wu, Y. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

LeCun, Y., Denker, J. S., and Solla, S. A. Optimal brain damage. In *Advances in neural information processing systems*, pp. 598–605, 1990.

Li, L., Yang, Y., and Li, B. Combine PPO with NES to improve exploration. *CoRR*, abs/1905.09492, 2019. URL http://arxiv.org/abs/1905.09492.

Li, Y. Deep reinforcement learning. *CoRR*, abs/1810.06339, 2018. URL http://arxiv.org/abs/1810.06339.

Liu, Z., Li, X., Kang, B., and Darrell, T. Regularization matters in policy optimization. *arXiv preprint arXiv:1910.09191*, 2019.

Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J., and Khudanpur, S. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533, 2015.

Peng, X. B., Abbeel, P., Levine, S., and van de Panne, M. Deepmimic: Example-guided deep reinforcement learning of physics-based character skills. *ACM Transactions on Graphics (Proc. SIGGRAPH 2018)*, 37(4), 2018.

Raffin, A. Rl baselines zoo. https://github.com/araffin/rl-baselines-zoo, 2018.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Sermanet, P., Chintala, S., and LeCun, Y. Convolutional neural networks applied to house numbers digit classification. In *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*, pp. 3288–3291. IEEE, 2012.

Stanley, K. O. and Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

Todorov, E., Erez, T., and Tassa, Y. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033. IEEE, 2012.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. *CoRR*, abs/1706.03762, 2017a. URL http://arxiv.org/abs/1706.03762.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017b.

Xie, Z., Berseth, G., Clary, P., Hurst, J., and van de Panne, M. Feedback control for cassie with deep reinforcement learning. In *Proc. IEEE/RSJ Intl Conf on Intelligent Robots and Systems (IROS 2018)*, 2018.

Yu, K., Sciuto, C., Jaggi, M., Musat, C., and Salzmann, M. Evaluating the search phase of neural architecture search, 2019.

Zoph, B. and Le, Q. V. Neural architecture search with reinforcement learning, 2016.