Learning to Learn in the Context of Spiking Neural Networks

Moritz Zanger[1]

***Abstract*— L2L abstract - I will write it in the end**

I. INTRODUCTION

Introduction to the topic

II. RECURRENT NEURAL NETWORKS (RNNS)

Artificial Neural Networks (ANNs) are networks of biologically inspired computational units , neurons, that learn according to certain learning rules thus improving their ability to perform a desired task, often classification or regression problems. Accordingly, the performance demonstrated by ANNs is based on knowledge, inferred from data (Schuster and Paliwal). Many of the data driven problems in engineering involve the recognition of time dependent patterns (e.g. speech recognition, machine translation, machine vision in real-time video material), requiring special network architectures such as the Time-Delay Neural Networks (TDNN, Waibel et al.) or Recurrent Neural Networks (RNNs). The latter revolve around the basic idea of including a loop in the networks neurons, allowing information to be passed from past steps of the networks state to the next.

A. Training RNNs

Unfolding an RNN over time results in a more approachable representation of this model as seen in fig. x. Training a model of this structure can now be achieved in a similar way to the well-known backpropagation algorithm with the additional ability to encode longer past information (Paul J. Werbos).

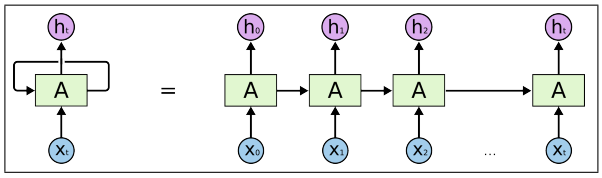


Fig. 1. Unrolled RNN (colah.github.io)

The notation of symbols in the following is according to Jian Guo. The inputs are denoted by vector $\mathbf{x}$, with components indexed by i. We similarly define the current hidden layer state as vector $\mathbf{s}$, components indexed by j, the previous hidden layer state as vector $\mathbf{s}(t-1)$, components indexed by h, the output layer as vector $\mathbf{y}$, components indexed by k. The weight matrices are written as bold uppercase Letters, where $\mathbf{V}$ maps $\mathbf{x}$ to the current state $\mathbf{s}(t)$, $\mathbf{U}$ maps the previous hidden state $\mathbf{s}(t-1)$, and $\mathbf{W}$ transforms the current state $\mathbf{s}(t)$ to the output layer $\mathbf{y}$. For clearness, the previously defined symbols can be seen in fig. x. Furthermore we describe the relation between outputs and net input function net_j or net_k of a layer as the activation functions $f(net_j(t))$ and $g(net_k(t))$ respectively for the hidden and output layer. Including biases b_j, b_k for the net input functions, we conclude that the hidden state becomes (1) and the output is described by (3).

$$s_j(t) = f(net_j(t)) \tag{1}$$

$$net_j(t) = \sum_i^l x_i(t)v_{ji} + \sum_h^m s_h(t-1)u_j h + b_j \tag{2}$$

$$y_k(t) = g(net_k(t)) \tag{3}$$

$$net_k(t) = \sum_j^m s_j(t)w_{kj} + b_k \tag{4}$$

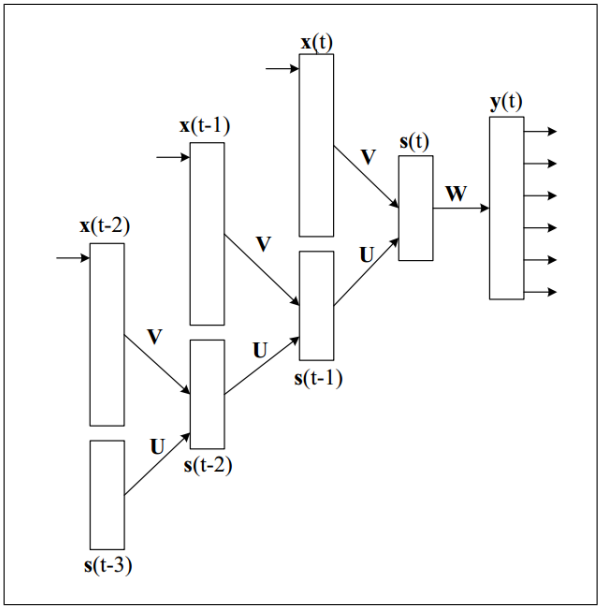


Fig. 2. An Unrolled RNN with shared weight Matrices U and V between temporal states (Guo) (colah.github.io)

In order to minimize the loss of our network, we define the widely used summed squared error (5) as a loss function with desired outputs d, total number of training samples n and the number of output units o. For computational reasons, the stochastic version of gradient descent utilizes only subsets of our training collection, up to the extremum where $p = n$.

$$E = \frac{1}{2}\sum_p^n \sum_k^o (d_{pk} - y_{pk})^2 \tag{5}$$

*This work was not supported by any organization

[1]Moritz Zanger, Faculty of Mechanical Engineering, Karlsruhe Institute of Technology zanger.moritz@googlemail.com

By propagating this error backwards throughout the network, we can construct the loss functions derivative with respect to each weight in order to obtain the weight update of a specific weight for the next timestep. Applying the chain rule, the weight updates for connections between hidden and output layer Δw_{kj} are denoted in (6) and for connections between input and hidden layer Δw_{ji} in (7):

$$\Delta w_{kj} = \eta \sum_{p}^{n} \delta_{pk} s_{pj} = \eta \sum_{p}^{n} (d_{pk} - y_{pk}) g'(net_{pk}) s_{pj} \quad (6)$$

$$\Delta w_{ji} = \eta \sum_{p}^{n} \delta_{pj} x_{pi} = \eta \sum_{p}^{n} \sum_{k}^{o} \delta_{pk} w_{kj} f'(net_{pj}) x_{pi} \quad (7)$$

The terms δ_{pk} and δ_{pj} are referred to as the components of the specific error vectors per layer. The netfunction of our hidden layer net_{pj} however still depends on the previous state s_{t-1} which in turn depends on previous states itself. Unfolding the network up to a temporal depth τ with the same weights throughout the timesteps (fig.x) allows us to define an error vector for previous time steps as seen in (8).

$$\delta_{pj}(t-1) = \sum_{h}^{m} \delta_{ph}(t) u_{hj} f'(s_{pj}(t-1)) \quad (8)$$

More detailed reflections on the construction of the above equations can be found in (Guo,2013). The procedure previously used to obtain $delta_{pj}(t-1)$ allows for theoretically arbitrary depths τ to be recursively calculated. In practice, large values for τ and the subsequently generated long-term dependencies tend to be difficult to handle with gradient descent. This is due to the fact, that deriving error terms through multiple layers of an unfolded RNN de- or increases the magnitude of the resulting derivative expontially, thus creating phenomena known as vanishing and exploding gradient.

@draft Because the parameters are shared by all time steps in the network, the gradient at each output depends not only on the calculations of the current time step, but also the previous time steps. This is called Backpropagation Through Time (BPTT)vanilla RNNs trained with BPTT have difficulties learning long-term dependencies (e.g. dependencies between steps that are far apart) due to what is called the vanishing/exploding gradient problem. There exists some machinery to deal with these problems, and certain types of RNNs (like LSTMs) were specifically designed to get around them.

B. Long Short Term Memory (LSTM)

The previously described problems occuring when facing long term dependencies in RNNs are successfully tackled by the Long Short Term Memory (LSTM), first introduced by Hochreiter and Schmidhuber. As shown by Hochreiter 1991, A key idea towards avoiding the explosion or vanishing of gradients through a large number of timesteps is to enforce a constant errorflow through a single unit j by requiring the net activation derivative of j to follow equation 9 with weight w_{jj} of a single connection to itself.

$$f'_j(net_j(t)) w_{jj} = 1.0 \quad (9)$$

Simple integration over time gives us $f_j(net_j(t)) = \frac{net_j}{w_{jj}}$, a linear function. Therefore our recurrent definition of $net_j(t+1) = w_{jj} y^j(t)$ leads us to the conclusion 10, that the unit j's activation has to remain constant (Hochreiter and Schmidhuber 1997):

$$y_j(t+1) = f_j(net(t+1)) = f_j(w_{jj} y^j(t)) = y^j(t) \quad (10)$$

This approach alone however leads to a problem that becomes evident during learning in networks with more than one neuron connected to the so far single unit j. The input weights w_{ji} from neuron i and the outgoing weights w_{kj} to neuron k now are both responsible for regulating the desired protection from yet unneeded information of previous timesteps as well as conceiving said information when deemed necessary. Take for instance a language processing network trying to estimate the importance of a recently heard subject "You", regarding the following words "are a tall, friendly, level-headed, smart person". The same weights that have just been trained on this sentence would receive very conflicting weight updates in the next learning step, when confronted with the new sentence "You are a person".
To overcome this shortcoming, Hochreiter and Schmidhuber propose a more context sensitive approach, using seperate gates for forgetting, input and output regulation within a single cell, as seen in fig. x.

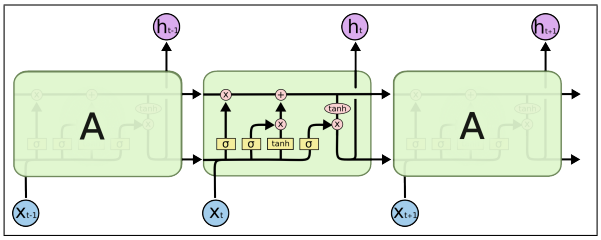


Fig. 3. (has to be edited) Inner composition of an LSTM cell (colah.github.io)

The $forgetgate$ consists of a sigmoid-activated layer f_t, essentially determining, how much of the previous output h_{t-1} should be kept within the LSTM Cell State C_t, based on its own set of weights W_f and biases b_f:

$$f_t = \sigma(W_f \times [h_{t-1}, x_t] + b_f) \quad (11)$$

The $inputgate$ is composed of the sigmoid-activated layer i_t and the hyperbolic-tangent-activated layer $\tilde{C}_t$, creating candidates for replacing the current (possibly forgotten) cell state C_{t-1} with $i_t \times \tilde{C}_t$. Each of these layers, again, have their own set of weights and biases W_i, b_i, W_c and b_c, leaving us with the respective layer outputs (12) and (13) (colah.github.improving):

$$i_t = \sigma(W_i \times [h_{t-1}, x_t] + b_i) \quad (12)$$

$$\tilde{C}_t = tanh(W_c \times [h_{t-1}, x_t] + b_c) \quad (13)$$

In order to determine whether to inhibit possibly disturbing signals to following cell states, the output h_t is filtered by applying $tanh(x)$ and multiplying with the outcome of the $outputgatelayero_t$ with weights W_o and biases b_o and results in the total output h_t:

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \quad (14)$$

$$h_t = o_t \times tanh(C_t) = o_t \times tanh(f_t \times C_{t-1} + i_t \times \tilde{C}_t) \quad (15)$$

This design allows for seperate read and write capabilities and therefore enables capture of error signals within the cell for longer periods of time than previously possible with standard RNNs (Hochreiter et al.). Many applications with record-breaking performances followed, thus proving the state-of-the-art standing of LSTMs in temporal problems. Graves and Jaitly provide an overview over variations of the vanilla LSTM, such as the introduction of peepholes by Gers and Schmidhuber or the lighter Gated Recurrent Unit (GRU) by Cho et al. Furthermore, attention-based models as proposed by Graves et al. (Natural Turing Machines), Graves et al.(Adaptive RNNs) and Neelakantan et al. () are promising more recent developtments in the field of RNNs.

@draft LSTMs dont have a fundamentally different architecture from RNNs, but they use a different function to compute the hidden state. The memory in LSTMs are called cells and you can think of them as black boxes that take as input the previous state h_{t-1} and current input x_t Internally these cells decide what to keep in (and what to erase from) memory. They then combine the previous state, the current memory, and the input. It turns out that these types of units are very efficient at capturing long-term dependencies.

C. Attention and Augmented Recurrent Neural Networks

III. SPIKING NEURAL NETWORKS (SNN)

The above described ANN models differ from Feedforward Networks in regards of activation, connection design, learning rules and more, yet implicitly share a commonground in how they encode information in their neurons. These second generation neural networks, as commonly described in the literature, activate on continuous functions and make use of their derivability in backpropagation. Maass et al. (Maass et al.) provide a setup that integrates a different, biologically more accurate take on modelling neuron activation. These Spiking Neural Networks (SNN) employ integrate-and-fire neurons (Maass et al.) which encode information with an additional temporal factor in its activation pulses, increasing the density of encoded information.

A. Neurons - Activation and Signal Processing

The process of signal transportation within biological neurons ...(figures and Gruning and Bohte)

The fundemantal idea behind the computational units of an SNN builds on integrating a temporal factor in the representation of information. Various models of these spiking neurons, such as the integrate-and-fire model (Abbott), the Hodgkin-Huxley model (Hodgkin and Huxley), the model by Izhikevich (Izhikevich) and the Spike Response Model by Gerstner (Gerstner) exist and vary in their attempt to trade off biological accuracy and computational complexity (Grunte and Bohte). The Leaky-Integrate-and-Fire model is nowadays the most widespread apporach due to its simplicity and computational advantages. The representation of the activation process of the neron is modeled by an electrical circuit in which the membrane potential, threshold voltage, resting potential and leak rate are realized through a capacitor, gate, battery and resistance respectively. (Abbott and fig. Ponulak) At any moment, an LIF neuron has a drive v, which depends on its bias current b; its inputs $a(in)_j$ (where the index j runs from 1 to the number of inputs); and its synaptic weights, W_j (Eliasmith and Anderson, 2002). (... and so on)

B. Spike-based Neural Codes

Whilst encoding and decoding of the desired information is much simpler and intuitive in second generation neural network models, this is a larger challenge for the time-dependent neurons in an SNN, due to the arbitrary number of theorically possible ways of encoding information in the neurons. In fact the biological process of information decoding is still being researched, whereas various methods have been introduced in Neuroscience Engineering.

- Rate Coding is an approach aiming at recording spike rates during fixed time frames. This implementation of spike encoding can be seen as an analog way of interpreting spike trains in SNNs.
- Latency Coding encodes spikes based on their timing rather than their multiplicity. This encoding has for example been used in unsupervised learning [43], and supervised learning methods like SpikeProp (S. Bohte, J. Kok)
- Fully temporal codes are a more general term which includes the above mentioned approaches. It encodes information based on the precise timing of each spike in a spike train. (Gruning and Bohte)
- Gaussian Coding applies a gaussian distribution over recorded spikes of each neuron and encodes information based on their stochastic occurence.

C. Learning in Spiking Neural Networks - Synaptic Plasiticity

Whilst conventional neural networks employ a stochastic version of gradient descend to backpropagate erros throughout the network, the same approach is difficult to apply in the realm of SNNs due to their temporal dependance and the non-differentiability of spike trains. Whereas multiple learning rules adressing SNNs exist (such as Hebbian Rule, Binarization of ANNs, Conversion from ANNs and Variations of backpropagation (Pfeiffer and Pfeil)), a state-of-the-art algorithm such as backpropagation has yet to emerge. We will focus on a more biologically motivated training rule called spike-timing-dependant plasiticy (STDP). The key feature of this approach is to adjust weights between a pre- and post-synaptic neuron according to their relative spike

times within an interval of roughly tens of milliseconds in length (S. Bohte, J. Kok). Consequently a ... more on STDP

1) Backpropagation and Feedback-alignment:

2) Error Feedback:

D. Performance of SNNs)

IV. LEARNING TO LEARN (L2L)

The field of reinforcement learning (RL) has recently celebrated great success at reaching human-like and even surpassing human abilities on complex environments such as Atari and Go (Mnih et al. and Silver et al.) with the implementation of Deep Neural Networks to account for non-linear function approximation over high-dimensional action and state spaces. However Artificial Intelligence in general (Landsell and Kording) and Reinforcement Learning in particular (Dual et al.) currently suffer from two major drawbacks, that are limiting their application and design (J. Wang et al.):

- Firstly the immense volume of required training data and the relatively expensive generation of this data in often simulated environments.
- Secondly RL-algorithms often have to be heavily tailored to a specific range of tasks and various algorithms, each of which depending on numerous hyperparameters and thus requiring immense efforts compared to currently reached results.

Botvinick et al. explain these weak spots in AI with a need for low learning rates and the bias-variance trade-off. Low learning rates are necessary to prevent both catastrophic interference (discarding previously reached successful configurations) and overfitting (Hardt, M. et al.). The bias-variance trade-off is a phenomenon describing the contrary working directions of efficiency-driving biases or priors and performing on a wider range of tasks.

——— maybe more basics from hochreiter et al. ———

With some approaches addressing these issues existing, Landsell and Kording argue, that these L2L approaches can be categorized into either Learning to Optimize or Structure Learning (Landsell and Kording). Learning to Optimize focusses on the general adaption of network parameters to achieve efficient learning rules on arbirary Task classes without hand-selection. Similiarly to the way gradient descent applies small changes of the weights in an NN in order to minimize loss functions, the design of AI systems can be viewed as an optimization problem itself, that requires parameter optimization to ensure a well performing algorithm. Structure Learning on the other hand makes use of structural similarities within a finite family of tasks to reach higher data efficiency due to its prior adaptnedness to the given family of tasks(Landsell and Kording).

A. Learning to Reinforcement Learn (metal-RL) and RL

A high level architecture consisting of a learner (performing on the task itself) and a meta-learner (adjusting

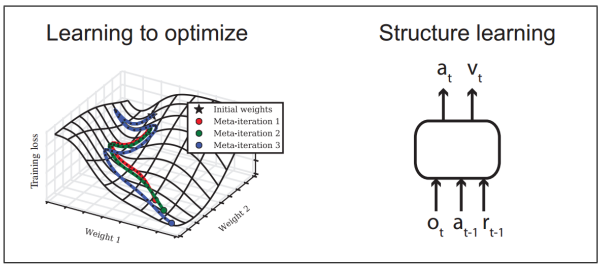


Fig. 4. Types of learning-to-learn in AI. Learning-to-learn can be roughly divided into learning to optimize and structure learning. In AI, hyperparameter optimization is an example of learning to optimize (Maclaurin et al. 2015), while a recurrent neural network taking rewards, actions and observations can often be used to perform structure learning (Wang et al.)(Landsell et al.).

the learner) is inherent to most implementations of L2L (Landsell and Kording) and has been refined in various ways to create new L2L Systems, as will be explained in the following section.

Wang et al. as well as Duan et al. introduced frameworks that can be thought of as generating an RL algorithm of their own and provide agents, who are given a predesigned prior to efficiently learn any task $T \epsilon F$ (in the original papers denoted as a Markov Decision Process (MDP) $m \epsilon M$) from a family of interrelated fasks F(i.e. a set of MDPs M).

In their attempt to design an algorithm, capable of performing well on a set M of Markov Decision Processes (MDPs), Duan et al. implement a nested system in which learning an RL algorithm is regarded as a reinforcement learning problem itself, hence the name RL(Duan et al.). The agent performing on a randomly drawn separate MDP $m \epsilon M$ from the distribution $\rho_M : M \longrightarrow R_+$ is represented as a recurrent neural network (RNN) which outputs the probability distribution over the tasks action-space π (policy) based on a function $\phi(s, a, r, d)$ of the tuple (state, action, reward, termination flag) (Duan et al.). On a higher abstraction layer, this RNN is being optimized by an implementaion of Trust Region Policy Optimization (TRPO), a state-of-the-art DRL algorithm (Schulman et al., 2015) with several advantages regarding stability and hyperparameter dependance.

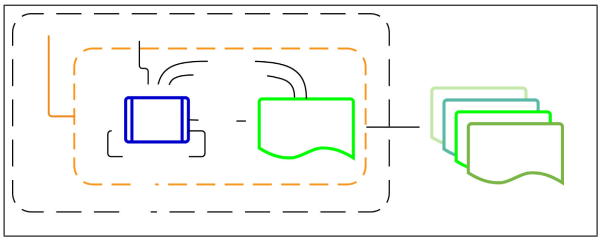


Fig. 5. Schematic of Meta-reinforcement Learning, Illustrating the Inner and Outer Loops of Training. The outer loop trains the parameter weights θ, which determine the inner-loop learner (Agent, instantiated by a recurrent neural network) that interacts with an environment for the duration of the episode. For every cycle of the outer loop, a new environment is sampled from a distribution of environments, which share some common structure (Botvinick et al.).

Wang et al. define a similiar setup in which a RL-

algorithm is responsible for learning the weights of a nested RNN. Both, inner and outer loop in this framework draw their learning experience from the reward information generated by the actions of the RNN (Wang et al.), where the RNN holds information on the previously chosen action and the subsequent rewards. However the process of learning in each of these loops is realized differently and results in specializations of different scopes. While the wrapping RL-algorithm used to optimize the weights of the RNN operates over the entire set of episodes, that is to say all MDPs M, learning of the nested RNN within a single task m is based on the inner recurrent dynamics of the network. The policy outputs π of this network can be viewed as an RL-algorithm on its own, resulting in the name meta-RL. For the implementation of this framework Wang et al. used a LSTM according to Hochreiter and Schmidhuber (Hochreiter and SChmidhuber, 1997) to account for the inner RNN, while both synchronous asynchronous advantage actor critics (A2C and A3C) (Mnih et al.) were employed to learn its parameters. The observation vectors of experiment environments were either directly fed to the LSTM one-hot-encoded or passed through an additional deep encoder model (Wang et al.). Experiments on a series of bandit problem and two MDP-centered problems with implementation architectures as described above showed, that meta-RL delivers competitive results compared to problem-specific algorithms (Thompson sampling, UCB, Gittins) while operating on a wider set of tasks.

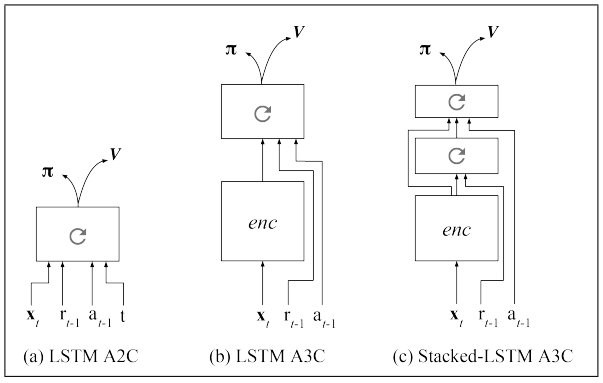


Fig. 6. Advantage actor-critic with recurrence. In all architectures, reward and last action are additional inputs to the LSTM. For non-bandit environments, observation is also fed into the LSTM either as a one-hot or passed through an encoder model [3-layer encoder: two convolutional layers (first layer: 16 8x8 filters applied with stride 4, second layer: 32 4x4 filters with stride 2) followed by a fully connected layer with 256 units and then a ReLU non-linearity. See for details Mirowski et al. (2016)]. For bandit experiments, current time step is also fed in as input. = policy; v = value function. A3C is the distributed multi-threaded asynchronous version of the advantage actor-critic algorithm (Mnih et al., 2016); A2C is single threaded. (a) Architecture used in experiments 1-5. (b) Convolutional-LSTM architecture used in experiment 6. (c) Stacked LSTM architecture with convolutional encoder used in experiments 6 and 7 (Wang et al.)

A notable characteristic of both previously described setups is that the learning rate of the nested RNN is chosen lower compared to the outer optimization loop, consequently preventing the agent from overfitting to a single task m, yet gathering knowledge from the entire MDP space M (Botvinick et al.).

more from :
Meta-SGD: Learning to Learn Quickly for Few-Shot Learning (Li et al.)
Meta-learning in Reinforcement Learning (Schweighofer and Doya)
Learning to Learn without Gradient Descent by Gradient Descent (Chen et al.) Learning to Learn Using Gradient Descent (Hochreiter et al.)

B. L2L in the Context of Spiking Neural Networks

tbd . . . Long short-term memory and Learning-to-learn in networks of spiking neurons (Bellec et al.) Biologically inspired alternatives to backpropagation through time for learning in recurrent neural nets (Bellec et al.) Embodied Neuromorphic Vision with Event-Driven Random Backpropagation (Kaiser et al.)

C. Implications for Neuroscience and Psychology

That is, having to learn their complete knowledge about the world from scratch, whereas the human brain has undergone a long history of evolutionary development, adjusting its learning paradigms to the challenges it faces (Duan et al.).

more from:
Prefrontal cortex as a meta-reinforcement learning system (WAng) Reinforcement Learning, Fast and Slow (Botvinick et al.)
Biologically inspired alternatives to backpropagation through time for learning in recurrent neural nets (Bellec et al.)
Towards learning-to-learn (Landsell and Kording)

V. APPLICATIONS OF L2L AND SNNS IN ROBOTICS

Robotics has undergone many successful developments in the recent past with advances being pushed from numerous fields of engineering, including that of machine learning. Yet the design process is still a tedious and highly taylored one, requiring many domain experts. Many of the underlying algorithms in the control, motion planning and sensoric interpretation require suitable setups of the environment with little room for variation. For example industrial manipulator robots can perform outstandingly when placed in a fixed production line, yet recognizing and grasping everyday objects in a kitchen or workshop poses a much higher challenge, as it requires the skill to make sense of broad environments with numerous imaginable tasks. Furthermore the dominating problems of applying RL in robotics can be summarized by the following problem classes (https://towardsdatascience.com/reinforcement-learning-for-real-world-robotics):

- Sample efficiency
- Sim2Real

- Reward Specification
- Safety

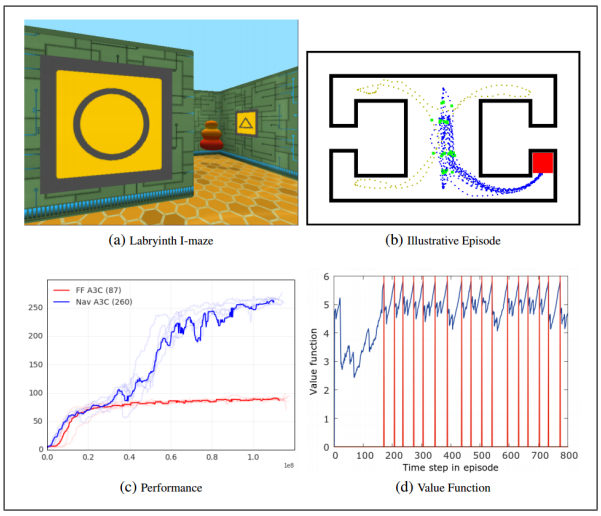
The previous sections revealed a high potential in L2L frameworks in terms of sample efficiency and generalization, thus constituting feasible answers to expensive data generation or overfitting to simulator-specific features. However this further implies a reflection on the scalability of said L2L approaches in order to evaluate their applicability in the often very high-dimensional task spaces faced in robotics.

Wang et al. examine meta-RL's ability to detect abstract task structures in large scale problems by adapting a well-known behavioural experiment described by Harlow (Harlow, 1949) to a visual fixation task. In Harlows experiment, monkeys were presented two unfamiliar objects, with one hiding a bowl filled with food and while the other holds an empty bowl. The monkeys were allowed to choose one of the objects and received the reward, if present. Despite switching the objects for new unkown objects in each episode, upon replaying several trials in several episodes of this game, the animals showed a general understanding of the underlying structure of the problem. After beginning a new episode with new objects, the monkeys would, inevitably, take one random guess but managed to succeed in the following trials of the episode (Botvonick et al.).

A. *Tasks in high-dimensional spaces*

Motion and path planning are fundamental problems in robotics, whether it be within a space of rich visual input, sensory data or configuration/join-spaces. Similiar to the problem described by Harlow, the navigational task in the I-maze environment as described by Mirowski et al. and Jaderberg et al. requires a understanding of the general structure of the problem in order to learn sample-efficiently on the specific task. In this case the same maze spawns a goal location on random position within the maze where the agent has to learn a motion path to the goal in as few trials as possible. The results of Wang et al. show, that an architecture of stacked LSTM is able to solve the task after having conducted one exploration run (finishing the episode in 100 timesteps) notably faster (30 timesteps) within few explotation runs. The reference baseline, a feedforward architecture A3C learner, is not able to solve the problem at all.

Duan et al. take a similiar approach in their evaluation of the feasbility of RL in high-dimensional state spaces. Again a randomly generated maze with a randomly placed target is chosen as the problem to solve for the agent. During one test run, the agent is given a number of episodes during which the maze structure and target position remain fixed. In contrast to an earlier approach to this RL-Task shown by Oh et al. RL bases its actions within a more granular action space (Duan et al.). The environments sparse reward payout design (+1 for target, -0.001 for wall hits, -0.04 per time frame) poses additional challenges to the agents learning and requires well-devloped exploration strategies in the first



(a) Labryinth I-maze (b) Illustrative Episode (c) Performance (d) Value Function

Fig. 7. a) view of I-maze showing goal object in one of the 4 alcoves b) following initial exploration (light trajectories), agent repeatedly goes to goal (blue trajectories) c) Performance of stacked LSTM (termed Nav A3C) and feedforward (FF A3C) architectures, per episode (goal = 10 points) averaged across top 5 hyperparameters. e) following initial goal discovery (goal hits marked in red), value function occurs well in advance of the agent seeing the goal which is hidden in an alcove.

episode in order to gain information on the problems ground structure. Cross-validation with a small and a larger version of the maze environment show a significant reduction in solving trajectory lengths between the first to episodes and indicate, that the RL algorithm managed to utilize previoulsy gained information to come to good solutions more quickly. However the shown results are not yet optimal as the agent still forgets, though rarely, initially explored target positions and explores further paths in the second episode. Duan et al. indicate that further imporvements might come with improved RL-algorithms as the outer-loop optimizer.

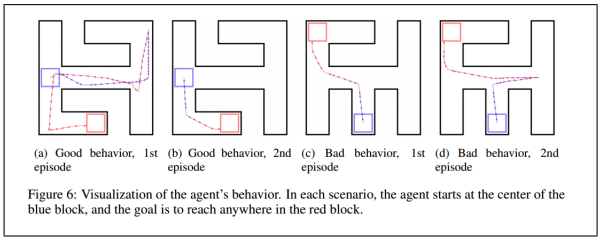

(a) Good behavior, 1st episode (b) Good behavior, 2nd episode (c) Bad behavior, 1st episode (d) Bad behavior, 2nd episode

Figure 6: Visualization of the agent's behavior. In each scenario, the agent starts at the center of the blue block, and the goal is to reach anywhere in the red block.

Fig. 8. Visualization of the agents behavior. In each scenario, the agent starts at the center of the blue block, and the goal is to reach anywhere in the red block.

more from :

Meta-SGD: Learning to Learn Quickly for Few-Shot Learning (Li et al.)

Meta-learning in Reinforcement Learning (Schweighofer and Doya)

Learning to Learn without Gradient Descent by Gradient Descent (Chen et al.) Learning Transferable Architectures for Scalable Image Recognition (Zoph et al.) OPTIMIZATION AS A MODEL FOR FEW-SHOT LEARNING (Ravi and Larochelle) One-Shot Imitation Learning (Duan et al.)

B. Speed Improvement and Few-Shot Learning

VI. CONCLUSION AND CHALLENGES

APPENDIX

Appendixes should appear before the acknowledgment.

REFERENCES

[1]