Reinforcement Learning Upside Down: Don't Predict Rewards - Just Map Them to Actions

NNAISENSE/IDSIA Technical Report

Jürgen Schmidhuber
The Swiss AI Lab, IDSIA, USI & SUPSI
NNAISENSE, Lugano, Switzerland

23 June 2020 (based on version v1 of 5 Dec 2019)
Earlier drafts: 21 Dec, 31 Dec 2017, 20 Jan, 4 Feb, 9 Mar, 20 Apr, 16 Jul 2018

Abstract

We transform reinforcement learning (RL) into a form of supervised learning (SL) by turning traditional RL on its head, calling this ꓤꓡ or Upside Down RL (UDRL). Standard RL predicts rewards, while ꓤꓡ instead uses rewards as task-defining inputs, together with representations of time horizons and other computable functions of historic and desired future data. ꓤꓡ learns to interpret these input observations as commands, mapping them to actions (or action probabilities) through SL on past (possibly accidental) experience. ꓤꓡ generalizes to achieve high rewards or other goals, through input commands such as: *get lots of reward within at most so much time!* ꓤꓡ can also learn to improve its exploration strategy. A separate paper [63] on first experiments with ꓤꓡ shows that even a pilot version of ꓤꓡ can outperform traditional baseline algorithms on certain challenging RL problems.

We also conceptually simplify an approach [60] for teaching a robot to imitate humans. First videotape humans imitating the robot's current behaviors, then let the robot learn through SL to map the videos (as input commands) to these behaviors, then let it generalize and imitate videos of humans executing previously unknown behavior. This *Imitate-Imitator* concept may actually explain why biological evolution has resulted in parents who imitate the babbling of their babies.

Rewards, time horizons, and desired futue states as inputs

A different approach to learning by demos

Note: This is a minor update of recent work [56].

Contents

1 Basic Ideas

Traditional RL machines [24, 67, 76] learn to predict rewards, given previous actions and observations, and learn to transform those predictions into rewarding actions. Our new method UDRL or ꓤꓡ is radically different. It does not predict rewards at all. Instead it takes rewards *as inputs*. More precisely, the ꓤꓡ machine observes commands in form of *desired rewards and time horizons*, such as: *"get so much reward within so much time."* Simply by interacting with the environment, it learns through gradient descent to map self-generated commands of this type to corresponding action probabilities. From such self-acquired knowledge it can extrapolate to solve new problems such as: *"get even more reward within even less time."* Remarkably, a simple ꓤꓡ pilot version already outperforms traditional RL methods on certain challenging problems [63].

Let us outline this new principle in more detail. An ꓤꓡ agent may interact with its environment during a single lifelong trial. At a given time, the history of actions and vector-valued [43, 44] costs (e.g., time, energy, pain & reward signals) and other observations up to this time contains all the agent can know about the present state of itself and the environment. Now it is looking ahead up to some future horizon, trying to obtain a lot of reward until then.

For all past pairs of times *(time1 $<$ time2)* it can retrospectively [1, 36] invent additional, consistent, vector-valued *command* inputs for itself, indicating tasks such as: achieve the already observed rewards/costs between *time1* and *time2*. Or: achieve more than half this reward, etc.

Just append actual rewards as "commands" to historical trajectories

Now it may simply use gradient-based SL to train a differentiable general purpose computer C such as a recurrent neural network (RNN) [73, 78, 39][53] to map the time-varying sensory inputs, augmented by the special *command* inputs defining time horizons and desired cumulative rewards etc, to the already known corresponding action sequences.

If the experience so far includes different but equally costly action sequences leading from some start to some goal, then C will learn to approximate the conditional expected values (or probabilities, depending on the setup) of appropriate actions, given the commands and other inputs.

The single life so far may yield an enormous amount of knowledge about how to solve all kinds of problems with limited resources such as time / energy / other costs. Typically, however, we want C to solve user-given problems, in particular, to get lots of reward quickly, e.g., by avoiding hunger (negative reward) caused by near-empty batteries, through quickly reaching the charging station without painfully bumping against obstacles. This desire can be encoded in a user-defined command of the type *(small desirable pain, small desirable time)*, and C will generalize and act based on what it has learned so far through SL about starts, goals, pain, and time. This will prolong C's lifelong experience; all new observations immediately become part of C's growing training set, to further improve C's behavior in continual [38] online fashion.

For didactic purposes, well first introduce formally the basics of ꓤꓡ for deterministic environments and Markovian interfaces between controller and environment (Sec. 3), then proceed to more complex cases in a series of additional Sections.

A separate paper [63] describes the concrete ꓤꓡ implementations used in our first experiments with ꓤꓡ, and presents remarkable experimental results.

2 Notation

More formally, in what follows, let m, n, o, p, q, u denote positive integer constants, and h, i, j, k, t, τ positive integer variables assuming ranges implicit in the given contexts. The i-th component of any real-valued vector, v, is denoted by v_i.

To become a general problem solver that is able to run arbitrary problem-solving programs, the controller C of an artificial agent must be a general-purpose computer [14, 7, 68, 35]. Artificial

Decision Transformers will effectively replace this by a transformer

recurrent neural networks (RNNs) fit this bill, e.g., [53]. The life span of our C (which could be an RNN) can be partitioned into trials $T_1, T_2, \ldots$. However, possibly there is only one single, lifelong trial. In each trial, C tries to manipulate some initially unknown environment through a sequence of actions to achieve certain goals. Let us consider one particular trial and its discrete sequence of time steps, $t = 1, 2, \ldots, T$.

Essentially, current inputs and future targets

At time t, during generalization of C's knowledge so far in Step 3 of Algorithm A1 or B1, C receives as an input the concatenation of the following vectors: a sensory input vector $in(t) \in \mathbb{R}^m$ (e.g., parts of $in(t)$ may represent the pixel intensities of an incoming video frame), a current vector-valued [44, 46] cost or reward vector $r(t) \in \mathbb{R}^n$ (e.g., components of $r(t)$ may reflect external positive rewards, or negative values produced by pain sensors whenever they measure excessive temperature or pressure or low battery load, that is, hunger), the previous output action $out'(t-1)$ (defined as an initial default vector of zeros in case of $t = 1$; see below), and extra variable task-defining input vectors $horizon(t) \in \mathbb{R}^p$ (a unique and unambiguous representation of the current look-ahead time), $desire(t) \in \mathbb{R}^n$ (a unique representation of the desired cumulative reward to be achieved until the end of the current look-ahead time), and $extra(t) \in \mathbb{R}^q$ to encode additional user-given goals (as we have done since 1990, e.g., [45, 57, 52]).

At time t, C then computes an output vector $out(t) \in \mathbb{R}^o$ used to select the final output action $out'(t)$. Often (e.g., Sec. 3.1.1) $out(t)$ is interpreted as a probability distribution over possible actions. For example, $out'(t)$ may be a *one-hot* binary vector $\in \mathbb{R}^o$ with exactly one non-zero component, $out'_i(t) = 1$ indicates action a^i in a set of discrete actions $\{a^1, a^2, \ldots, a^o\}$, and $out_i(t)$ the probability of a^i. Alternatively, for even o, $out(t)$ may encode the mean and the variance of a multi-dimensional Gaussian distribution over real-valued actions [77], from which a high-dimensional action $out'(t) \in \mathbb{R}^{o/2}$ is sampled accordingly, e.g., to control a multi-joint robot. The execution of $out'(t)$ may influence the environment and thus future inputs and rewards to C.

all(t) is the total input at time t

Let $all(t)$ denote the concatenation of $out'(t-1), in(t), r(t)$. Let $trace(t)$ denote the sequence $(all(1), all(2), \ldots, all(t))$.

3 Deterministic Environments With Markovian Interfaces

For didactic purposes, we start with the case of deterministic environments, where there is a Markovian interface [46] between agent and environment, such that C's current input tells C all there is to know about the current state of its world. In that case, C does not have to be an RNN - a multilayer feedforward network (FNN) [22, 53] is sufficient to learn a policy that maps inputs, desired rewards and time horizons to probability distributions over actions.

The following Algorithms A1 and A2 run in parallel, occasionally exchanging information at certain synchronization points. They make C learn many cost-aware policies from a single behavioral trace, taking into account many different possible time horizons. Both A1 and A2 use local variables reflecting the input/output notation of Sec. 2. Where ambiguous, we distinguish local variables by appending the suffixes "$[A1]$" or "$[A2]$," e.g., $C[A1]$ or $t[A2]$ or $in(t)[A1]$.

A1 = Experience collection, no training involved -> trying to achieve set targets

Algorithm A1: Generalizing through a copy of C (with occasional exploration)

1. Set $t := 1$. Initialize local variable C (or $C[A1]$) of the type used to store controllers.
2. Occasionally sync with Step 3 of Algorithm A2 to set $C[A1] := C[A2]$ (since $C[A2]$ is continually modified by Algorithm A2).
3. **Execute one step:** Encode in $horizon(t)$ the goal-specific remaining time, e.g., until the end of the current trial (or twice the lifetime so far [21]). Encode in $desire(t)$ a desired cumulative reward to be achieved within that time (e.g., a known upper bound of the maximum possible cumulative reward, or the maximum of (a) a positive constant and (b) twice the maximum cumulative reward ever achieved before). C observes the concatentation of $all(t), horizon(t), desire(t)$ (and $extra(t)$, which may specify additional commands - see Sec. 3.1.6 and Sec. 4). Then C outputs a probability distribution $out(t)$ over the next possible actions. Probabilistically select $out'(t)$ accordingly (or set it deterministically to one of the most probable actions). In exploration mode (e.g., in a constant fraction of all time steps), modify $out'(t)$ randomly (optionally, select $out'(t)$ through some other scheme, e.g., a traditional algorithm for planning or RL or black box optimization [53, Sec. 6] - such details are not essential for ⅂ꓤ). Execute action $out'(t)$ in the environment, to get $in(t+1)$ and $r(t+1)$.
4. Occasionally sync with Step 1 of Algorithm A2 to transfer the latest acquired information about $t[A1], trace(t+1)[A1]$, to increase C[A2]'s training set through the latest observations.
5. If the current trial is over, exit. Set $t := t + 1$. Go to 2.

desired reward signal

Algorithm A2: Learning lots of time & cumulative reward-related commands

This is also, in some sense, a world model

A2 = training code, follow actual trajectories, append actual signals, and train to predict these

1. Occasionally sync with A1 (Step 4) to set $t[A2] := t[A1], trace(t+1)[A2] := trace(t+1)[A1]$.
2. **Replay-based training on previous behaviors and commands compatible with observed time horizons and costs: For** all pairs $\{(k, j); 1 \leq k \leq j \leq t\}$: train C through gradient descent-based backpropagation [29, 25, 71][53, Sec. 5.5] to emit action $out'(k)$ at time k in response to inputs $all(k)$, $horizon(k)$, $desire(k)$, $extra(k)$, where $horizon(k)$ encodes the remaining time $j - k$ until time j, and $desire(k)$ encodes the total costs and rewards $\sum_{\tau=k+1}^{j+1} r(\tau)$ incurred through what happened between time steps k and j. (Here $extra(k)$ may be a non-informative vector of zeros - alternatives are discussed in Sec. 3.1.6 and Sec. 4.)
3. Occasionally sync with Step 2 of Algorithm A1 to copy $C[A1] := C[A2]$. Go to 1.

3.1 Properties and Variants of Algorithms A1 and A2

3.1.1 Learning Probabilistic Policies Even in Deterministic Environments

In Step 2 of Algorithm A2, the past experience may contain many different, equally costly sequences of going from a state uniquely defined by $in(k)$ to a state uniquely defined by $in(j+1)$. Let us first focus on discrete actions encoded as *one-hot* binary vectors with exactly one non-zero component (Sec. 2). Although the environnment is deterministic, by minimizing mean squared error (MSE), C will learn *conditional expected values*

$$out(k) = E(out' \mid all(k), horizon(k), desire(k), extra(k))$$

of corresponding actions, given C's inputs and training set, where E denotes the expectation operator. That is, due to the binary nature of the action representation, C will actually learn to estimate *conditional probabilities*

$$out_i(k) = P(out' = a_i \mid all(k), horizon(k), desire(k), extra(k))$$

of appropriate actions, given C's inputs and training set. For example, in a video game, two equally long paths may have led from location A to location B around some obstacle, one passing it to the left, one to the right, and C may learn a 50% probability of going left at a fork point, but afterwards there is only one fast way to B, and C can learn to henceforth move forward with highly confident actions, assuming the present goal is to minimize time and energy consumption.

Hard to discover policies using only scalar rewards, but easier if you predict actions that achieve a specific reward based on history

ꓤꓤ is of particular interest for high-dimensional actions (e.g., for complex multi-joint robots), because SL can easily deal with those, while traditional RL does not. See Sec. 6.1.3 for learning probability distributions over such actions, possibly with statistically dependent action components.

3.1.2 Compressing More and More Skills into C

In Step 2 of Algorithm A2, more and more skills are compressed or collapsed into C, like in the chunker-automatizer system of the 1991 neural history compressor [47], where a student net (the "automatizer") is continually re-trained not only on its previous skills (to avoid forgetting), but also to imitate the behavior of a teacher net (the "chunker"), which itself keeps learning new behaviors.

3.1.3 No Problems With Discount Factors

No gamma involved here -- rewards targets on finite horizons

Some of the math of traditional RL [24, 67, 76] heavily relies on problematic discount factors. Instead of maximizing $\sum_{\tau=1}^{T} r(\tau)$, many RL machines try to maximize $\sum_{\tau=1}^{T} \gamma^\tau r(\tau)$ or $\sum_{\tau=1}^{\infty} \gamma^\tau r(\tau)$ (assuming unbounded time horizons), where the positive real-valued discount factor $\gamma < 1$ distorts the real rewards in exponentially shrinking fashion, thus simplifying certain proofs (e.g., by exploiting that $\sum_{\tau=1}^{\infty} \gamma^\tau r(\tau)$ is finite).

ꓤꓤ, however, explicitly takes into account observed time horizons in a precise and natural way, does not assume infinite horizons, and does not suffer from distortions of the basic RL problem.

3.1.4 Representing Time / Omitting Representations of Time Horizons

what is p?

What is a good way of representing look-ahead time through $horizon(t) \in \mathbb{R}^p$? The simplest way may be $p = 1$ and $horizon(t) = t$. A less quickly diverging representation is $horizon(t) = \sum_{\tau=1}^{t} 1/\tau$. A bounded representation is $horizon(t) = \sum_{\tau=1}^{t} \gamma^\tau \tau$ with positive real-valued $\gamma < 1$. Many distributed representations with $p > 1$ are possible as well, e.g., date-like representations.

Episodic reward targets

In cases where C's life can be segmented into several time intervals or *episodes* of varying lengths unkown in advance, and where we are only interested in C's total reward per episode, we may omit C's $horizon()$-input. C's $desire()$-input still can be used to encode the desired cumulative reward until the time when a special component of C's $extra()$-input switches from 0 to 1, thus indicating the end of the current episode. It is straightforward to modify Algorithms A1/A2 accordingly.

3.1.5 Computational Complexity

The replay [28] of Step 2 of Algorithm A2 can be done in $O(t(t+1)/2)$ time per training epoch. In many real-world applications, such quadratic growth of computational cost may be negligible compared to the costs of executing actions in the real world. (Note also that hardware is still getting exponentially cheaper over time, overcoming any simultaneous quadratic slowdown.) See Sec. 3.1.8.

3.1.6 Learning a Lot From a Single Trial - What About Many Trials?

In Step 2 of Algorithm A2, for every time step, C learns to obey many commands of the type: get so much future reward within so much time. That is, from a single trial of only 1000 time steps, it derives roughly half a million trainig examples conveying a lot of fine-grained knowledge about time and rewards. For example, C may learn that small increments of time often correspond to small increments of costs and rewards, except at certain crucial moments in time, e.g., at the end of a board game when the winner is determined. A single behavioral trace may thus inject an enormous amount of knowledge into C, which can learn to explicitly represent all kinds of long-term and short-term causal relationships between actions and consequences, given the initially unknown environment. For example, in typical physical environments, C could automatically learn detailed maps of space / time / energy / other costs associated with moving from many locations (at different altitudes) to many target locations [57, 45, 52, 1, 36] encoded as parts of $in(t)$ or of $extra(t)$ - compare Sec. 4.1.

n(n+1)/2 combinations of time horizons

If there is not only one single lifelong trial, we may run Step 2 of Algorithm A2 for previous trials as well, to avoid forgetting of previously learned skills, like in the POWERPLAY framework [52, 64].

3.1.7 How Frequently Should One Synchronize Between Algorithms A1 and A2?

It depends a lot on the task and the computational hardware. In a real world robot environment, executing a single action in Step 3 of A1 may take more time than billions of training iterations in Step 2 of A2. Then it might be most efficient to sync after every single real world action, which immediately may yield for C many new insights into the workings of the world. On the other hand, when actions and trials are cheap, e.g., in simple simulated worlds, it might be most efficient to synchronize rarely.

3.1.8 On Reducing Training Complexity by Selecting Few Relevant Training Sequences

To reduce the complexity $O(t(t+1)/2)$ of Step 2 of Algorithm A2 (Sec. 3.1.5), certain SL methods will ignore most of the training sequences defined by the pairs (k, j) of Step 2, and instead select only a few of them, either randomly, or by selecting *prototypical* sequences, inspired by *support vector machines* (SVMs) whose only effective training examples are the *support vectors* identified through a margin criterion [69, 58], such that (for example) correctly classified outliers do not directly affect the final classifier. In environments where actions are cheap, the selection of only few training sequences may also allow for synchronizing more frequently between Algorithms A1 and A2 (Sec. 3.1.7).

Similarly, when the overall goal is to learn a single rewarding behavior through a series of trials, at the start of a new trial, a variant of A2 could simply delete/ignore the training sequences collected during most of the less rewarding previous trials, while Step 3 of A1 could still demand more reward than ever observed. Assuming that C is getting better and better at acquiring reward over time, this will not only reduce training efforts, but also bias C towards recent rewarding behaviors, at the risk of making C forget how to obey commands demanding low rewards.

There are numerous applicable SL *tricks of the trade* (e.g., [31]) and sophisticated ways of selectively deleting past experiences from the training set to improve and speed up SL.

4 Other Properties of the History as Command Inputs

A single trial can yield even much more additional information for C than what is exploited in Step 2 of Algorithm A2. For example, the following addendum to Step 2 trains C to also react to an input command saying *"obtain* **more than** *this reward within so much time"* instead of *"obtain so much*

reward within so much time," simply by training on all past experiences that retrospectively match this command.

2b. **Additional replay-based training on previous behaviors and commands compatible with observed time horizons and costs for Step 2 of Algorithm A2: For** all pairs $\{(k, j); 1 \leq k \leq j \leq t\}$: train C through gradient descent to emit action $out'(k)$ at time k in response to inputs $all(k)$, $horizon(k)$, $desire(k)$, $extra(k)$, where one of the components of $extra(k)$ is a special binary input $morethan(k) := 1.0$ (normally 0.0), where $horizon(k)$ encodes the remaining time $j - k$ until time j, and $desire(k)$ encodes *half* the total costs and rewards $\sum_{\tau=k+1}^{j+1} r(\tau)$ incurred between time steps k and j, or 3/4 thereof, or 7/8 thereof, etc.

That is, C now also learns to generate probability distributions over action trajectories that yield *more than* a certain amount of reward within a certain amount of time. Typically, their number greatly exceeds the number of trajectories yielding *exact* rewards, which will be reflected in the correspondingly reduced conditional probabilities of action sequences learned by C.

A natural corresponding modification of Step 3 of Algorithm A1 is to encode in $desire(t)$ the maximum conditional reward ever achieved, given $all(t), horizon(t)$, and to activate the special binary input $morethan(t) := 1.0$ as part of $extra(t)$, such that C can generalize from what it has learned so far about the concept of obtaining *more than* a certain amount of reward within a certain amount of time. **Thus RL can learn to improve its exploration strategy in goal-directed fashion.**

4.1 Desirable Goal States / Locations

For e.g., HER

Yet another modification of Step 2 of Algorithm A2 is to encode within parts of $extra(k)$ a final desired input $in(j+1)$ (assuming $q > m$), like in previous work where extra inputs are used to define goals or target locations [57, 45, 52, 1, 36], such that C can be trained to execute commands of the type *"obtain so much reward within so much time and finally reach a particular state identified by this particular input."* See Sec. 6.1.2 for generalizations of this.

The natural corresponding modification of Step 3 of Algorithm A1 is to encode such desired inputs [57] in $extra(t)$, e.g., a goal location that has never been reached before.

4.2 Infinite Number of Computable, History-Compatible Commands

Obviously there are infinitely many other computable functions of subsequences of $trace(t)$ with binary outputs *true* or *false* that yield *true* when applied to certain subsequences. In principle, such *computable predicates* could be encoded in Algorithm A2 as unique commands for C with the help of $extra(k)$, to further increase C's knowledge about how the world works, such that C can better generalize when it comes to planning future actions in Algorithm A1. In practical applications, however, one can train C only on finitely many commands, which should be chosen wisely.

Note the similarity to POWERPLAY (2011) [52, 64] which allows for *arbitrary computable task specifications* as extra inputs to an RL system. Since in general there are many possible tasks, POWERPLAY has a built-in way of selecting new tasks automatically and economically. POWERPLAY, however, not only looks backwards in time to find new commands compatible with the observed history, but can also actively set goals that require to obtain new data from the environment through interaction with it.

5 Probabilistic Environments Effectively, replace individual trajectories by sample averages

In probabilistic environments, for two different time steps $l \neq h$ we may have $all(l) = all(h)$, $out(l) = out(h)$ but $r(l+1) > r(h+1)$, due to *"randomness"* in the environment. To address this, let us first discuss *expected* rewards. Given $all(l), all(h)$ and keeping the Markov assumption of Sec. 3, we may use C's command input $desire(.)$ to encode a desired *expected* immediate reward of $1/2[r(l+1) + r(h+1)]$ which, together with $all(h)$ and a $horizon()$ representation of 0 time steps, should be mapped to $out(h)$ by C, assuming a uniform conditional reward distribution.

More generally, assume a finite set of states $\{s^1, s^2, \ldots, s^u\}$, each with an unambiguous encoding through C's $in()$ vector, and actions $\{a^1, a^2, \ldots, a^o\}$ with one-hot encodings (Sec. 2). For each pair (s^i, a^j) we can use a real-valued variable z_{ij} to estimate [18] the expected immediate reward for executing a^j in s^i. This reward is assumed to be independent of the history of previous actions and observations (Markov assumption [65]).

z_{ij} can be updated incrementally and cheaply whenever a^j is executed in s^i in Step 3 of Algorithm A1, and the resulting immediate reward is observed. The following simple modification of Step 2 of Algorithm A2 trains C to map *desired expected* rewards (rather than plain rewards) to actions, based on the observations so far.

2* **Replay-based training on previous behaviors and commands compatible with observed time horizons and expected costs in probabilistic Markov environments for Step 2 of Algorithm A2: For** all pairs $\{(k, j); 1 \leq k \leq j \leq t\}$: train C through gradient descent to emit action $out'(k)$ at time k in response to inputs $all(k)$, $horizon(k)$, $desire(k)$ (we ignore $extra(k)$ for simplicity), where $horizon(k)$ encodes the remaining time $j - k$ until time j, and $desire(k)$ encodes the *estimate of the total expected costs and rewards* $\sum_{\tau=k+1}^{j+1} E(r(\tau))$, where the $E(r(\tau))$ are estimated in the obvious way through the $z_{..}$ variables corresponding to visited states / executed actions between time steps k and j.

If randomness is affecting not only the immediate reward for executing a^j in s^i but also the resulting next state, then *Dynamic Programming* (DP) [4] can still estimate in similar fashion cumulative expected rewards (to be used as command inputs encoded in $desire()$), given the training set so far. This approach essentially adopts central aspects of traditional DP-based RL [24, 67, 76] without affecting the method's overall order of computational complexity (Sec. 3.1.5).

From an algorithmic point of view [62, 26, 27, 50], however, randomness simply reflects a separate, unobservable oracle injecting extra bits of information into the observations. Instead of learning to map *expected* rewards to actions as above, C's problem of partial observability can also be addressed by adding to C's input a unique representation of the current time step, such that it can learn the *concrete* reward's dependence on time, and is not misled by a few lucky past experiences.

It is most natural to consider the case of probabilistic environments as a special case of partially observable environments discussed next in Sec. 6.

6 Partially Observable Environments Use RNNs and expectations both

In case of a non-Markovian interface [46] between agent and environment, C's current input does not tell C all there is to know about the current state of its world. A recurrent neural network (RNN) [53] or a similar general purpose computer may be required to translate the entire history of previous observations and actions into a meaningful representation of the present world state. Without loss of generality, we focus on C being an RNN such as LSTM [19, 12, 17, 53] which has become highly commercial, e.g., [41, 79, 70, 34]. Algorithms A1 and A2 above have to be modified accordingly,

resulting in Algorithms B1 and B2 (with local variables and input/output notation analoguous to A1 and A2, e.g., $C[B1]$ or $t[B2]$ or $in(t)[B1]$).

Algorithm B1: Generalizing through a copy of C (with occasional exploration)

1. Set $t := 1$. Initialize local variable C (or $C[B1]$) of the type used to store controllers.

2. Occasionally sync with Step 3 of Algorithm B2 to **do:** copy $C[B1] := C[B2]$ (since $C[B2]$ is continually modified by Algorithm B2). Run C on $trace(t-1)$, such that C's internal state contains a memory of the history so far, where the inputs $horizon(k)$, $desire(k)$, $extra(k)$, $1 \leq k < t$ are retrospectively adjusted to match the observed reality up to time t. One simple way of doing this is to let $horizon(k)$ represent 0 time steps, $extra(k)$ the null vector, and to set $desire(k) = r(k+1)$, for all k (but many other consistent commands are possible, e.g., Sec. 4).

3. **Execute one step:** Encode in $horizon(t)$ the goal-specific remaining time (see Algorithm A1). Encode in $desire(t)$ a possible future cumulative reward, and in $extra(t)$ additional goals, e.g., to receive more than this reward within the remaining time - see Sec. 4. C observes the concatentation of $all(t), horizon(t), desire(t), extra(t)$, and outputs $out(t)$. Select action $out'(t)$ accordingly. In exploration mode (i.e., in a constant fraction of all time steps), modify $out'(t)$ randomly. Execute $out'(t)$ in the environment, to get $in(t+1)$ and $r(t+1)$.

4. Occasionally sync with Step 1 of Algorithm B2 to transfer the latest acquired information about $t[B1], trace(t+1)[B1]$, to increase C[B2]'s training set through the latest observations.

5. If the current trial is over, exit. Set $t := t+1$. Go to 2.

Algorithm B2: Learning lots of time & cumulative reward-related commands

1. Occasionally sync with B1 (Step 4) to set $t[B2] := t[B1], trace(t+1)[B2] := trace(t+1)[B1]$.

2. **Replay-based training on previous behaviors and commands compatible with observed time horizons and costs: For** all pairs $\{(k,j); 1 \leq k \leq j \leq t\}$ **do**: If $k > 1$, run RNN C on $trace(k-1)$ to create an internal representation of the history up to time k, where for $1 \leq i < k$, $horizon(i)$ encodes 0 time steps, $desire(i) = r(i+1)$, and $extra(i)$ may be a vector of zeros (see Sec. 4, 3.1.4, 6.1.2 for alternatives). Train RNN C to emit action $out'(k)$ at time k in response to this previous history (if any) and $all(k)$, where the special command input $horizon(k)$ encodes the remaining time $j-k$ until time j, and $desire(k)$ encodes the total costs and rewards $\sum_{\tau=k+1}^{j+1} r(\tau)$ incurred through what happened between time steps k and j, while $extra(k)$ may encode additional commands compatible with the observed history, e.g., Sec. 4, 6.1.2.

3. Occasionally sync with Step 2 of Algorithm B1 to copy $C[B1] := C[B2]$. Go to 1.

6.1 Properties and Variants of Algorithms B1 and B2

Comments of Sec. 3.1 apply in analaguous form, generalized to the RNN case. In particular, although each replay for some pair of time steps (k,j) in Step 2 of Algorithm B2 takes into account the entire history up to k and the subsequent future up to j, Step 2 can be implemented such that its computational complexity is still only $O(t^2)$ per training epoch (compare Sec. 3.1.5).

6.1.1 Retrospectively Pretending a Perfect Life So Far

Note that during generalization in Algorithm B1, RNN C always acts as if its life so far has been perfect, as if it always has achieved what it was told, because its command inputs are retrospectively adjusted to match the observed outcome, such that RNN C is fed with a consistent history of commands and other inputs.

6.1.2 Arbitrarily Complex Commands for RNNs as General Computers

Recall Sec. 4. Since RNNs are general computers, we can train an RNN C on additional complex commands compatible with the observed history, using $extra(t)$ to help encoding commands such as: *"obtain more than this reward within so much time, while visiting a particular state (defined through an extra goal input encoded in $extra(t)$ [57, 45]) at least 3 times, but not more than 5 times."*

That is, like in POWERPLAY (2011) [52], we can train C to obey essentially arbitrary computable task specifications that match previously observed traces of actions and inputs. Compare Sec. 4, 4.2. (To deal with (possibly infinitely) many tasks, POWERPLAY can order tasks by the computational effort required to add their solutions to the task repertoire.)

6.1.3 High-Dimensional Actions with Statistically Dependent Components

As mentioned in Sec. 3.1.1, TRL is of particular interest for high-dimensional actions, because SL can easily deal with those, while traditional RL does not.

Let us first consider the case of multiple trials, where $out(k) \in \mathbb{R}^o$ encodes a probability distribution over high-dimensional actions, where the i-th action component $out'_i(k)$ is either 1 or 0, such that there are at most 2^o possible actions.

C can be trained by Algorithm B2 to emit $out(k)$, given C's input history. This is straightforward under the assumption that the components of $out'(.)$ are statistically independent of each other, given C's input history.

In general, however, they are not. For example, a C controlling a robot with 5 fingers should often send similar, statistically redundant commands to each finger, e.g., when closing its hand.

To deal with this, Algorithms B1 and B2 can be modified in a straightforward way. Any complex high-dimensional action at a given time step can be computed/selected incrementally, component by component, where each component's probability also depends on components already selected earlier.

If there is a clear hierarchy among actions

More formally, in Algorithm B1 we can decompose each time step t into o discrete *micro time steps* $\hat{t}(1), \hat{t}(2), \ldots, \hat{t}(o)$ (see [43], Sec. on *"more network ticks than environmental ticks"*). At $\hat{t}(1)$ we initialize real-valued variable $out'_0(t) = 0$. During $\hat{t}(i), 1 \leq i \leq o$, C computes $out_i(t)$, the probability of $out'_i(t)$ being 1, given C's internal state (based on its previously observed history) and its current inputs $all(t)$, $horizon(t)$, $desire(t)$, $extra(t)$ and $out'_{i-1}(t)$ (observed through an additional special *action input unit* of C). Then $out'_i(t)$ is sampled accordingly, and for $i < o$ used as C's new *special action input* at the next micro time step $\hat{t}(i+1)$.

Training of C in Step 2 of Algorithm B2 has to be modified accordingly. There are obvious, similar modifications of Algorithms B1 and B2 for Gaussian and other types of probability distributions.

6.1.4 Computational Power of RNNs: Generalization & Randomness vs. Determinism

This is an important subsection. First recall that Sec. 3.1.1 pointed out how an FNN-based C of Algorithms A1/A2 in general will learn probabilistic policies even in deterministic environments, since at a given time t, C can perceive only the recent $all(t)$ but not the entire history $trace(t)$, reflecting an inherent Markov assumption [65, 46, 24, 67, 76].

If there is only one single lifelong trial, however, this argument does not hold for the RNN-based C of Algorithms B1/B2, because at each time step, an RNN could in principle uniquely represent the entire history so far, for instance, by learning to simply count the time steps [11].

This is conceptually very attractive. We do not even have to make any probabilistic assumptions any more. Instead, ⅂ꓤ simply learns to map histories and commands directly to high-dimensional deterministic actions $out'(.) := out(.) \in \mathbb{R}^o$. (This tends to be hard for traditional RL.)

Even in seemingly probabilistic environments (Sec. 5), an RNN C could learn deterministic policies, taking into account the precise histories after which these policies worked in the past, assuming that what seems random actually may have been computed by some deterministic (initially unknown) algorithm, e.g., a pseudorandom number generator [81, 48, 49, 50, 51].

To illustrate the conceptual advantages of single life settings, let us consider a simple task where an agent can pass an obstacle either to the left or to the right, using continuous actions in [0,1] defining angles of movement, e.g., 0.0 means go left, 0.5 go straight (and hit the obstacle), 1.0 go right.

First consider an episodic setting and a sequence of trials where C is reset after each trial. Suppose actions 0.0 and 1.0 have led to high reward 10.0 equally often, and no other actions such as 0.3 have triggered high reward. Given reward input command 10.0, the agent's RNN C will learn an expected output of 0.5, which of course is useless as a real-valued action—instead one has to somehow interpret this as an action *probability* based on certain assumptions about an underlying distribution (Sec. 3, 5, 6.1.3). Note, however, that the typical popular Gaussian assumptions would not make sense here.

On the other hand, in a single life with, say, 10 subsequent sub-trials, C can learn arbitrary history-dependent algorithmic conditions of actions, e.g.: in trials 3, 6, 9, action 0.0 was followed by high reward. In trials 4, 5, 7, action 1.0 was. Other actions 0.4, 0.3, 0.7, 0.7 in trials 1, 2, 8, 10 respectively, yielded low reward. By sub-trial 11, in response to reward command 10.0, C should correctly produce either action 0.0 or 1.0 but not their mean 0.5.

In additional sub-trials, C might even discover complex conditions such as: if the trial number is divisible by 3, then choose action 0.0, else 1.0. In this sense, in single life settings, life is getting conceptually simpler, not harder. Because the whole baggage associated with probabilistic thinking and *a priori* assumptions about probability distributions and environmental resets (see Sec. 5) is getting irrelevant and can be ignored.

On the other hand, C's success in case of similar commands in similar situations at different time steps will now all depend on its generalization capability. For example, from its historic data, it must learn in step 2 of Algorithm B2 when precise time stamps are important and when to ignore them.

Sure, even in deterministic environments, C might find it useful to invent a variant of probability theory to model its uncertainty, and to make seemingly "random" decisions with the help of a self-invented deterministic internal pseudorandom generator. However, no probabilistic assumptions (such as the above-mentioned overly restrictive Gaussian assumption) should be imposed onto C a priori.

To improve C's generalization capability, well-known regularizers [53, Sec. 5.6.3] can be used during training in Step 2 of Algorithm B2. See also Sec. 3.1.8.

⅂ꓤ for RNNs or other general purpose computers without any probabilistic assumptions (Sec. 3.1.1, 5, 6.1.3) may be both the simplest and most powerful ⅂ꓤ variant.

6.1.5 RNNs With Memories of Initial Commands

There are variants of ⅂ꓤ with an RNN-based C that accepts commands such as *"get so much reward per time in this trial"* only in the beginning of each trial, or only at certain selected time steps, such that $desire(.)$ and $horizon(.)$ do not have to be updated any longer at every time step, because the RNN can learn to internally memorize previous commands. However, then C must also somehow be able to observe at which time steps t to ignore $desire(t)$ and $horizon(t)$. This can be achieved through a special marker input unit whose activation as part of $extra(t)$ is 1.0 only if the present $desire(t)$ and

$horizon(t)$ commands should be obeyed (otherwise this activation is 0.0). Thus C can know during the trial: The current goal is to match the last command (or command sequence) identified by this marker input unit. This approach can be implemented through obvious modifications of Algorithms B1 and B2.

6.1.6 Combinations with Supervised Pre-Training and Other Techniques

It is trivial to combine ꓤꓡ and SL, since both share the same basic framework. In particular, C can be pre-trained by SL to imitate teacher-given trajectories. The corresponding traces can simply be added to C's training set of Step 2 of Algorithm B2.

Similarly, traditional RL methods or AI planning methods can be used to create additional behavioral traces for training C.

For example, we may use the company NNAISENSE's winner of the NIPS 2017 "learning to run" competition to generate several behavioral traces of a successful, quickly running, simulated 3-dimensional skeleton controlled through relatively high-dimensional actions, in order to pre-train and initialize C. C may then use ꓤꓡ to further refine its behavior.

7 Compress Successful Behaviors Into a Compact Standard Policy Network Without Command Inputs

C has to learn a possibly complex mapping from desired rewards, time horizons, and normal sensory inputs, to actions. Small changes in initial conditions or reward commands may require quite different actions. A deep and complex network may be necessary to learn this. During exploitation, however, we do not need this complex mapping any longer, we just need a working policy that maps sensory inputs to actions. This policy may fit into a much smaller network.

difficulty carried through to the DT paper as well

possible fix: train a student network to map s->a direct, looking at complex trajectories of main network

Hence, to exploit successful behaviors learned through algorithms A1/A2 or B1/B2, we simply compress them into a policy network called CC, like in the 1991 chunker-automatizer system [47], where a student net (the "automatizer") is continually re-trained not only on its previous skills (to avoid forgetting), but also to imitate the behavior of a teacher net (the "chunker"), which itself keeps learning new behaviors. The POWERPLAY framework [52, 64] also uses a similar approach, learning one task after another, using environment-independent replay of behavioral traces (or functionally equivalent but more efficient approaches) to avoid forgetting previous skills and to compress or speed up previously found, sub-optimal solutions, e.g., [52, Sec. 3.1.2]. Similar for the "One Big Net" [55] and a recent study of incremental skill learning with feedforward networks [5].

Using the notation of Sec. 2, the policy net CC is like C, but without special input units for the command inputs $horizon(.)$, $desire(.)$, $extra(.)$. We immediately consider the case where CC is an RNN living in a partially observable environment (Sec. 6).

Algorithm Compress (replay-based training on previous successful behaviors):

1. **For** each previous trial that is considered successful: Using the notation of Sec. 2, **For** $1 \leq k \leq T$ **do**: Train RNN CC to emit action $out'(k)$ at time k in response to the previously observed part of the history $trace(k-1)$.

For example, in a given environment, ꓤꓡ can be used to solve an RL task requiring to achieve maximal reward / minimal time under particular initial conditions (e.g., starting from a particular initial state). Later, Algorithm Compress can collapse many different satisfactory solutions for many different initial conditions into CC, which ignores reward and time commands.

8 Imitate a Robot, to Make it Learn to Imitate You!

The concept of learning to use rewards and other goals as command inputs has broad applicability. In particular, we can apply it in an elegant and straighforward way to train robots on *learning by demonstration* tasks [80, 42, 2, 9, 60] considered notoriously difficult in traditional robotics. We'll conceptually simplify an approach [60] for teaching a robot to imitate humans.

For example, suppose that an RNN C should learn to control a complex humanoid robot with eye-like cameras perceiving a visual input stream. We want to teach it complex tasks, such as assembling a smartphone, solely by visual demonstration, without touching the robot - a bit like we'd teach a kid.

First the robot must learn what it means to imitate a human. Its joints and hands may be quite different from yours. But you can simply let the robot execute already known or even accidental behavior. Then simply imitate it with your own body! The robot tapes a video of your imitation through its cameras. The video is used as a sequential command input for the RNN controller C (e.g., through parts of $extra()$, $desire()$, $horizon()$), and C is trained by SL to respond with its known, already executed behavior. That is, C can learn by SL to imitate you, because you imitated C.

Once C has learned to imitate or obey several video commands like this, let it generalize: do something it has never done before, and use the resulting video as a command input.

In case of unsatisfactory imitation behavior by C, imitate it again, to obtain additional training data. And so on, until performance is sufficiently good. The algorithmic framework *Imitate-Imitator* formalizes this procedure.

Algorithmic Framework: Imitate-Imitator

1. **Initialization:** Set temporary integer variable $i := 0$.
2. **Demonstration:** Visually show to the robot what you want it to do, while it videotapes your behavior, yielding a video V.
3. **Exploitation / Exploration:** Set $i := i + 1$. Let RNN C sequentially observe V and then produce a trace H^i of a series of interactions with the environment (if in exploration mode, produce occasional random actions). If the robot is deemed a satisfactory imitator of your behavior, exit.
4. **Imitate Robot:** Imitate H^i with your own body, while the robot records a video V^i of your imitation.
5. **Train Robot:** For all $k, 1 \leq k \leq i$ train RNN C through gradient descent [53, Sec. 5.5] to sequentially observe V^k (plus the already known total vector-valued cost R^k of H^k) and then produce H^k, where the pair (V^k, R^k) is interpreted as a sequential command to perform H^k under cost R^k. Go to Step 3 (or to Step 2 if you want to demonstrate anew).

It is obvious how to implement variants of this procedure through straightforward modifications of Algorithms B1 and B2 along the lines of Sec. 4, e.g., using a gradient-based sequence-to-sequence mapping approach based on LSTM, e.g., [17, 66, 79].

Of course, the *Imitate-Imitator* approach is not limited to videos. All kinds of sequential, possibly multi-modal sensory data could be used to describe desired behavior to an RNN C, including spoken commands, or gestures. For example, observe a robot, then describe its behaviors in your own language, through speech or text. Then let it learn to map your descriptions to its own corresponding behaviors. Then describe a new desired behavior to be performed by the robot, and let it generalize from what it has learned so far.

Once the robot has learned to execute command (V^k, R^k) through behavior H^k, standard ⅂ʁ without a teacher can be used to further refine H^k, by commanding the robot to produce similar behavior under different cost $\hat{R}^k$ (of the same dimensionality as R^k). If necessary, the robot is trained to obey the commands through an additional series of trials. For example, a robot that already knows how to assemble some object may now learn by itself to assemble it faster or with less energy.

The central idea of the present Sec. 8 on what we'd like to call *show-and-tell robotics* or *watch-and-learn robotics* or *see-and-do robotics* may actually explain why biological evolution has evolved parents who imitate the babbling of their babies: the latter can thus quickly learn to translate input sequences caused by the behavior of their parents into action sequences corresponding to their own equivalent behavior. Essentially they are learning their parent's language to describe behaviors, then generalize and translate previously unknown behaviors of their parents into equivalent own behaviors.

9 Relation of Upside Down RL to Previous Work

Using SL for certain aspects of RL dates back to the 1980s and 90s [72, 32, 23, 75, 74, 40, 33]. In particular, like ⅂ʁ, our early end-to-end-differentiable recurrent RL machines (1990) also observe vector-valued reward signals as sensory inputs [43, 44, 46]. What is the concrete difference between those and ⅂ʁ? The earlier systems [43, 44, 46] also use gradient-based SL in RNNs to learn mappings from costs/rewards and other inputs to actions. But unlike ⅂ʁ they do not have *desired* rewards as *command* inputs, and typically the training depends on an RNN-based predictive world model M (which predicts rewards, among other things) to compute gradients for the RNN controller C. ⅂ʁ, however, does not depend at all on good reward predictions (compare [54, Sec. 5]), only on the generalization ability of the learned mapping from previously observed rewards and other inputs to action probabilities.

What is the difference to our early multi-goal RL systems (1990) which also had extra input vectors used to encode possible goals [57]? Again, it is essentially the one mentioned in the previous paragraph: ⅂ʁ does not require additional predictions of reward.

What is the difference to our early end-to-end-differentiable hierarchical RL (HRL) systems (1990) which also had extra task-defining inputs in form of start/goal combinations, learning to invent sequences of subgoals [45]? Unlike ⅂ʁ, such HRL also needs a predictor of costs/rewards (called an evaluator), given start/goal combinations, to derive useful subgoals through gradient descent.

What is the difference to hindsight experience replay (HER, 2017) [1] extending experience replay (ER, 1991) [28]? HER replays paths to randomly encountered potential goal locations, but still depends on traditional RL. HER's controller neither sees extra real-valued $horizon$ and $cost$ inputs nor general computable predicates thereof, and thus does not learn to generalize from *known* costs in the training set to *desirable* costs in the generalization phase. (HER also does not use an RNN to deal with partial observability through encoding the entire history). Similar considerations hold for hindsight policy gradients [36].

What is the difference to RUDDER [3] which also uses gradient-based SL in RNNs to perform contribution analysis, mapping rewards to state-action pairs? Unlike ⅂ʁ, RUDDER does not use desired rewards as command input for an SL model.

To summarise, as discussed above, mapping rewards [43, 44, 46] and goals [57] (plus other inputs) to actions is not new. But traditional RL methods [24, 67, 76] do not have *command* inputs in form of *desired* rewards, and most of them need some additional method for learning to select actions based on predictions of future rewards. For example, a more recent system [8] also predicts future measurements (possibly rewards), given actions, and selects actions leading to best predicted measurements, given goals. A characteristic property of ⅂ʁ, however, is its very simple shortcut: it learns directly from (possibly accidental) experience the mapping from rewards to actions.

ꟻꓤ is also very different from traditional black box optimization (BBO) [37, 59, 20, 10] such as neuroevolution [30, 61, 15, 13] which can be used to solve complex RL problems in partially observable environments [16] through iterative discovery of better and better parameters of an adaptive controller, yielding more and more reward per trial. ꟻꓤ does not even try to modify any weights with the objective of increasing reward. Instead it just tries to understand from previous experience through standard gradient-based learning how to translate (desired) rewards etc into corresponding actions. Unlike BBO, ꟻꓤ is also applicable when there is only one single lifelong trial; the new observations of any given time step can immediately be used to improve the learner's overall behavior.

What is the difference between ꟻꓤ and POWERPLAY (2011) [52, 64]? Like ꟻꓤ, POWERPLAY does receive extra command inputs in form of arbitrary (user-defined or self-invented) computable task specifications, possibly involving start states, goal states, and costs including time. It even orders (at least the self-invented) tasks automatically by the computational difficulty of adding their solutions to the skill repertoire. But it does not necessarily systematically consider all previous training sequences between all possible pairs of previous time steps encountered so far by accident. See also Sec. 4.2.

Of course, we could limit POWERPLAY's choice of new problems to problems of the form: *choose a unique new command for C reflecting a computable predicate that is true for some already observed action sequence (Sec. 4.2), and add the corresponding skill to C's repertoire, without destroying previous knowledge.* Such an association of a new command with a corresponding skill or policy will cost time and other resources; POWERPLAY will, as always, prefer new skills that are easy to add. (Recall that one can train C only on finitely many commands, which should be chosen wisely.)

Note also that at least the strict versions of POWERPLAY insist that adding a new skill does not decrease performance on (replays of) previous tasks, while ꟻꓤ's occasional sychronization of Algorithms A1/A2 and B1/B2 does not immediately guarantee this, due to limited time between synchronizations, and basic limitations of gradient descent. Nevertheless, in the long run, Algorithms A2/B2 of ꟻꓤ will keep up with the stream of incoming new observations from Algorithms A1/B1, and thus won't forget previous skills of C due to constant retraining, much like POWERPLAY.

10 Experiments

A separate paper [63] describes the concrete implementations used in our first experiments with a pilot version of ꟻꓤ, and presents remarkable experimental results.

11 Conclusion

Traditional RL predicts rewards, and uses a myriad of methods for translating those predictions into good actions. ꟻꓤ shortcuts this process, creating a direct mapping from rewards, time horizons and other inputs to actions. Without depending on reward predictions, and without explicitly maximizing expected rewards, ꟻꓤ simply learns by gradient descent to map task specifications or commands (such as: *get lots of reward within little time*) to action probabilities. Its success depends on the generalization abilities of deep / recurrent neural nets. Its potential drawbacks are essentially those of traditional gradient-based learning: local minima, underfitting, overfitting, etc. [6, 53]. Nevertheless, experiments in a separate paper [63] show that even our initial pilot version of ꟻꓤ can outperform traditional RL methods on certain challenging problems.

A related *Imitate-Imitator* approach is to imitate a robot, then let it learn to map its observations of the imitated behavior to its own behavior, then let it generalize, by demonstrating something new, to be imitated by the robot.

12 Acknowledgments

I am grateful to Paulo Rauber, Sjoerd van Steenkiste, Wojciech Jaskowski, Rupesh Kumar Srivastava, Jan Koutnik, Filipe Mutz, and Pranav Shyam for useful comments. This work was supported in part by a European Research Council Advanced Grant (no: 742870).

References

[1] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba. Hindsight experience replay. *Preprint arXiv:1707.01495*, 2017.

[2] B. D. Argall, S. Chernova, M. Veloso, and B. Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, 2009.

[3] J. A. Arjona-Medina, M. Gillhofer, M. Widrich, T. Unterthiner, J. Brandstetter, and S. Hochreiter. Rudder: Return decomposition for delayed rewards. In *Advances in Neural Information Processing Systems (NIPS)*, pages 13544–13555, 2019.

[4] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1st edition, 1957.

[5] G. Berseth, C. Xie, P. Cernek, and M. V. de Panne. Progressive reinforcement learning with distillation for multi-skilled motion control. In *Proc. International Conference on Learning Representations (ICLR); Preprint arXiv:1802.04765v1*, 2018.

[6] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[7] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[8] A. Dosovitskiy and V. Koltun. Learning to act by predicting the future. In *Proc. International Conference on Learning Representations (ICLR 2017)*, 2017.

[9] Y. Duan, M. Andrychowicz, B. Stadie, O. J. Ho, J. Schneider, I. Sutskever, P. Abbeel, and W. Zaremba. One-shot imitation learning. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1087–1098, 2017.

[10] L. Fogel, A. Owens, and M. Walsh. *Artificial Intelligence through Simulated Evolution*. Wiley, New York, 1966.

[11] F. A. Gers and J. Schmidhuber. Recurrent nets that time and count. In *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, volume 3, pages 189–194. IEEE, 2000.

[12] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 2000.

[13] T. Glasmachers, T. Schaul, Y. Sun, D. Wierstra, and J. Schmidhuber. Exponential natural evolution strategies. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 393–400. ACM, 2010.

[14] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[15] F. J. Gomez. *Robust Nonlinear Control through Neuroevolution*. PhD thesis, Department of Computer Sciences, University of Texas at Austin, 2003.

[16] F. J. Gomez, J. Schmidhuber, and R. Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, 9(May):937–965, 2008.

[17] A. Graves, M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, and J. Schmidhuber. A novel connectionist system for improved unconstrained handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5), 2009.

[18] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning*. Springer Series in Statistics, 2009.

[19] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997. Based on TR FKI-207-95, TUM (1995).

[20] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[21] M. Hutter. *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability*. Springer, Berlin, 2005. (On J. Schmidhuber's SNF grant 20-61847).

[22] A. G. Ivakhnenko and V. G. Lapa. *Cybernetic Predicting Devices*. CCM Information Corporation, 1965.

[23] M. I. Jordan. Supervised learning and systems with excess degrees of freedom. Technical Report COINS TR 88-27, Massachusetts Institute of Technology, 1988.

[24] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: a survey. *Journal of AI research*, 4:237–285, 1996.

[25] H. J. Kelley. Gradient theory of optimal flight paths. *ARS Journal*, 30(10):947–954, 1960.

[26] A. N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of Information Transmission*, 1:1–11, 1965.

[27] M. Li and P. M. B. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications (2nd edition)*. Springer, 1997.

[28] L.-J. Lin. Programming robots using reinforcement learning and teaching. In *Proceedings of the Ninth National Conference on Artificial Intelligence - Volume 2*, AAAI'91, pages 781–786. AAAI Press, 1991.

[29] S. Linnainmaa. The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's thesis, Univ. Helsinki, 1970.

[30] G. Miller, P. Todd, and S. Hedge. Designing neural networks using genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 379–384. Morgan Kauffman, 1989.

[31] G. Montavon, G. Orr, and K. Müller. *Neural Networks: Tricks of the Trade*. Number LNCS 7700 in Lecture Notes in Computer Science Series. Springer Verlag, 2012.

[32] P. W. Munro. A dual back-propagation scheme for scalar reinforcement learning. *Proceedings of the Ninth Annual Conference of the Cognitive Science Society, Seattle, WA*, pages 165–176, 1987.

[33] N. Nguyen and B. Widrow. The truck backer-upper: An example of self learning in neural networks. In *Proceedings of the International Joint Conference on Neural Networks*, pages 357–363. IEEE Press, 1989.

[34] J. Pino, A. Sidorov, and N. Ayan. Transitioning entirely to neural machine translation. *Facebook Research Blog, 2017, https://code.facebook.com/posts/289921871474277/transitioning-entirely-to-neural-machine-translation/*.

[35] E. L. Post. Finite combinatory processes-formulation 1. *The Journal of Symbolic Logic*, 1(3):103–105, 1936.

[36] P. Rauber, F. Mutz, and J. Schmidhuber. Hindsight policy gradients. *Preprint arXiv:1711.06006*, 2017.

[37] I. Rechenberg. Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Dissertation, 1971. Published 1973 by Fromman-Holzboog.

[38] M. B. Ring. *Continual Learning in Reinforcement Environments*. PhD thesis, University of Texas at Austin, Austin, Texas 78712, August 1994.

[39] A. J. Robinson and F. Fallside. The utility driven dynamic error propagation network. Technical Report CUED/F-INFENG/TR.1, Cambridge University Engineering Department, 1987.

[40] T. Robinson and F. Fallside. Dynamic reinforcement driven error propagation networks with application to game playing. In *Proceedings of the 11th Conference of the Cognitive Science Society, Ann Arbor*, pages 836–843, 1989.

[41] H. Sak, A. Senior, K. Rao, F. Beaufays, and J. Schalkwyk. Google voice search: faster and more accurate. *Google Research Blog, 2015, http://googleresearch.blogspot.ch/2015/09/google-voice-search-faster-and-more.html*.

[42] S. Schaal. Is imitation learning the route to humanoid robots? *Trends in Cognitive Sciences*, 3(6):233–242, 1999.

[43] J. Schmidhuber. Making the world differentiable: On using fully recurrent self-supervised neural networks for dynamic reinforcement learning and planning in non-stationary environments. Technical Report FKI-126-90 (revised), Institut für Informatik, Technische Universität München, November 1990. (Revised and extended version of an earlier report from February.).

[44] J. Schmidhuber. An on-line algorithm for dynamic reinforcement learning and planning in reactive environments. In *Proc. IEEE/INNS International Joint Conference on Neural Networks, San Diego*, volume 2, pages 253–258, 1990.

[45] J. Schmidhuber. Learning to generate sub-goals for action sequences. In T. Kohonen, K. Mäkisara, O. Simula, and J. Kangas, editors, *Artificial Neural Networks*, pages 967–972. Elsevier Science Publishers B.V., North-Holland, 1991.

[46] J. Schmidhuber. Reinforcement learning in Markovian and non-Markovian environments. In D. S. Lippman, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3 (NIPS 3)*, pages 500–506. Morgan Kaufmann, 1991.

[47] J. Schmidhuber. Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4(2):234–242, 1992. (Based on TR FKI-148-91, TUM, 1991).

[48] J. Schmidhuber. A computer scientist's view of life, the universe, and everything. In C. Freksa, M. Jantzen, and R. Valk, editors, *Foundations of Computer Science: Potential - Theory - Cognition*, volume 1337, pages 201–208. Lecture Notes in Computer Science, Springer, Berlin, 1997, submitted 1996.

[49] J. Schmidhuber. Algorithmic theories of everything. Technical Report IDSIA-20-00, quant-ph/0011122, IDSIA, Manno (Lugano), Switzerland, 2000. Sections 1-5: see [50]; Section 6: see [51].

[50] J. Schmidhuber. Hierarchies of generalized Kolmogorov complexities and nonenumerable universal measures computable in the limit. *International Journal of Foundations of Computer Science*, 13(4):587–612, 2002.

[51] J. Schmidhuber. The Speed Prior: a new simplicity measure yielding near-optimal computable predictions. In J. Kivinen and R. H. Sloan, editors, *Proceedings of the 15th Annual Conference on Computational Learning Theory (COLT 2002)*, Lecture Notes in Artificial Intelligence, pages 216–228. Springer, Sydney, Australia, 2002.

[52] J. Schmidhuber. POWERPLAY: Training an Increasingly General Problem Solver by Continually Searching for the Simplest Still Unsolvable Problem. *Frontiers in Psychology*, 2013. (Based on arXiv:1112.5309v1 [cs.AI], 2011).

[53] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. Published online 2014; 888 references; based on TR arXiv:1404.7828 [cs.NE].

[54] J. Schmidhuber. On learning to think: Algorithmic information theory for novel combinations of reinforcement learning controllers and recurrent neural world models. *Preprint arXiv:1511.09249*, 2015.

[55] J. Schmidhuber. One big net for everything. *Preprint arXiv:1802.08864 [cs.AI]*, February 2018.

[56] J. Schmidhuber. Reinforcement learning upside down: Don't predict rewards–just map them to actions. *Preprint arXiv:1912.02875*, 5 Dec 2019.

[57] J. Schmidhuber and R. Huber. Learning to generate artificial fovea trajectories for target detection. *International Journal of Neural Systems*, 2(1 & 2):135–141, 1991. (Based on TR FKI-128-90, TUM, 1990).

[58] B. Schölkopf, C. J. C. Burges, and A. J. Smola, editors. *Advances in Kernel Methods - Support Vector Learning*. MIT Press, Cambridge, MA, 1998.

[59] H. P. Schwefel. Numerische Optimierung von Computer-Modellen. Dissertation, 1974. Published 1977 by Birkhäuser, Basel.

[60] P. Sermanet, C. Lynch, J. Hsu, and S. Levine. Time-contrastive networks: Self-supervised learning from multi-view observation. *Preprint arXiv:1704.06888*, 2017.

[61] K. Sims. Evolving virtual creatures. In A. Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 1994)*, Computer Graphics Proceedings, Annual Conference, pages 15–22. ACM SIGGRAPH, ACM Press, jul 1994. ISBN 0-89791-667-0.

[62] R. J. Solomonoff. A formal theory of inductive inference. Part I. *Information and Control*, 7:1–22, 1964.

[63] R. K. Srivastava, P. Shyam, F. Mutz, W. Jaskowski, and J. Schmidhuber. Training agents using upside-down reinforcement learning. *NNAISENSE Technical Report 201911-02, arXiv:1912.02877*, 2019. *NeurIPS 2019 Deep RL workshop*.

[64] R. K. Srivastava, B. R. Steunebrink, and J. Schmidhuber. First experiments with PowerPlay. *Neural Networks*, 41(0):130 – 136, 2013. Special Issue on Autonomous Learning.

[65] R. Stratonovich. Conditional Markov processes. *Theory of Probability And Its Applications*, 5(2):156–178, 1960.

[66] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. Technical Report arXiv:1409.3215 [cs.CL], Google, 2014. NIPS'2014.

[67] R. Sutton and A. Barto. *Reinforcement learning: An introduction*. Cambridge, MA, MIT Press, 1998.

[68] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 41:230–267, 1936.

[69] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, 1995.

[70] W. Vogels. Bringing the Magic of Amazon AI and Alexa to Apps on AWS. *All Things Distributed, 2016, http://www.allthingsdistributed.com/2016/11/amazon-ai-and-alexa-for-all-aws-apps.html*.

[71] P. J. Werbos. Applications of advances in nonlinear sensitivity analysis. In *System modeling and optimization*, pages 762–770. Springer, 1982.

[72] P. J. Werbos. Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man, and Cybernetics*, 17, 1987.

[73] P. J. Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1, 1988.

[74] P. J. Werbos. Backpropagation and neurocontrol: A review and prospectus. In *IEEE/INNS International Joint Conference on Neural Networks, Washington, D.C.*, volume 1, pages 209–216, 1989.

[75] P. J. Werbos. Neural networks for control and system identification. In *Proceedings of IEEE/CDC Tampa, Florida*, 1989.

[76] M. Wiering and M. van Otterlo. *Reinforcement Learning*. Springer, 2012.

[77] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[78] R. J. Williams and D. Zipser. Gradient-based learning algorithms for recurrent networks and their computational complexity. In *Back-propagation: Theory, Architectures and Applications*. Hillsdale, NJ: Erlbaum, 1994.

[79] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *Preprint arXiv:1609.08144*, 2016.

[80] M. Yeasin and S. Chaudhuri. Automatic robot programming by visual demonstration of task execution. In *Proc. 8th International Conference on Advanced Robotics, ICAR'97*, pages 913–918. IEEE, 1997.

[81] K. Zuse. *Rechnender Raum*. Friedrich Vieweg & Sohn, Braunschweig, 1969. English translation: *Calculating Space,* MIT Technical Translation AZT-70-164-GEMIT, Massachusetts Institute of Technology (Proj. MAC), Cambridge, Mass. 02139, Feb. 1970.