ML Paper Summaries

Ario Zareinia

April 2024

1 Fundamentals

1.1 Algorithm Papers

1.1.1 [INC] Probabilistic Recurrent State-Space Models

Prerequisites: The reader should have a very strong understanding of linear algebra, probability, stochastic processes, state space models and deep learning. Understanding of graphical models would be an asset.

Motivation: State space models have successfully been used to learn time series and system dynamics. Deterministic models, such as RNNs, have been extremely successful, as have linear probabilistic state space models. However, generic nonlinear state space models are difficult to robustly train, especially in high dimensions. A nonlinear, probabilistic state space model that is easy to train and apply is desirable.

Basic Idea (buckle up, this is gonna be dense): The Probabilistic Recurrent State Space Model (or PR-SSM) is a nonlinear Gaussian process state-space model that uses variational inference to predict latnt state. Yes, that is a mouthful, which is why we will break down the pieces you need to know before describing PR-SSM.

A Gaussian Process is a distribution over functions $f\colon \mathbb{R}^D \to \mathbb{R}$, uniquely and fully defined by a mean function $m(\cdot)\colon \mathbb{R}^D \to \mathbb{R}$ and a covariance function $k(\cdot,\cdot)\colon \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$. For every finite set of points $\boldsymbol{X} = [\boldsymbol{x}_1, ..., \boldsymbol{x}_n]$, the evaluations $[f(\boldsymbol{x}_1), ..., f(\boldsymbol{x}_n)]$ are jointly Gaussian: $p(\boldsymbol{f}|\boldsymbol{X}) = \boldsymbol{N}(\boldsymbol{f}|\boldsymbol{m_X}, \boldsymbol{K_{X,X}})$, with $\boldsymbol{m_X}$ being a vector where $m_i = m(\boldsymbol{x}_i)$, and $\boldsymbol{K_{X,X}}$ being a matrix where $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$. For a new input location $\boldsymbol{x}^*$, the predictive distribution for this point is given by $p(f^*|\boldsymbol{x}^*, \boldsymbol{f}, \boldsymbol{X}) = \mathcal{N}(f^*|\mu, \sigma^2)$ where $\mu = m(\boldsymbol{x}^*) + \boldsymbol{k_{x^*,X}}\boldsymbol{K}^{-1}_{\boldsymbol{X,X}}(\boldsymbol{f} - \boldsymbol{m_X})$ and $\sigma^2 = k(\boldsymbol{x}^*, \boldsymbol{x}^*) - \boldsymbol{k_{x^*,X}}\boldsymbol{K}^{-1}_{\boldsymbol{X,X}}\boldsymbol{k_{X,x^*}}$, where $\boldsymbol{k_{x^*,X}}$ is a row vector with $k_i = k(\mathbf{x}^*, \boldsymbol{x}_i)$ ($\boldsymbol{k_{X,x^*}}$ is the column vector with the same entries).

The prediction listed above is commonly conditioned on all data $\boldsymbol{X}$; however, this is not always tractable. To alleiviate this problem, a set of P Gaussian Process targets $\boldsymbol{z} = [\boldsymbol{z}_1, ..., \boldsymbol{z}_P]$ with corresponding pseudo-input points $\boldsymbol{\zeta} = [\boldsymbol{\zeta}_1, ..., \boldsymbol{\zeta}_P]$, the true distribution can be approximated by conditioning only on this set of points: $p(f^*|\boldsymbol{x}^*, \boldsymbol{f}, \boldsymbol{X}) \approx p(f^*|\boldsymbol{x}^*, \boldsymbol{z}, \boldsymbol{\zeta})$, with $p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z}|\boldsymbol{m_\zeta}, \boldsymbol{K_{\zeta,\zeta}})$. Now, let $\boldsymbol{x}_t \in \mathbb{R}^{D_x}, \boldsymbol{u}_t \in \mathbb{R}^{D_u}, \boldsymbol{y}_t \in \mathbb{R}^{D_y}$ be the latent state, input and output at timestep t, respectively, and let $f(\cdot)$ and $g(\cdot)$ be the transition model and output model respectively. Let $\hat{\boldsymbol{x}}_t = (\boldsymbol{x}_t, \boldsymbol{u}_t)$ be a shorthand for the input to the transition model, and let $\boldsymbol{f}_{t+1} = f(\hat{\boldsymbol{x}}_t)$. Let $L_{a:b}$ denote the series of observations of quantity L_t from $t = a$ to $t = b$ inclusive. The joint distribution of all random variables in the PR-SSM model, as well as a graphical model, is given below:

$$p(\boldsymbol{y}_{1:T}, \boldsymbol{x}_{1:T}, \boldsymbol{f}_{2:T}, \boldsymbol{z}) = \left[\prod_{t=1}^{T} p(\boldsymbol{y}_t \mid \boldsymbol{x}_t)\right] p(\boldsymbol{x}_1)p(\boldsymbol{z}) \quad (8)$$
$$\left[\prod_{t=2}^{T} p(\boldsymbol{x}_t \mid \boldsymbol{f}_t)p(\boldsymbol{f}_t \mid \hat{\boldsymbol{x}}_{t-1}, \boldsymbol{z})\right],$$

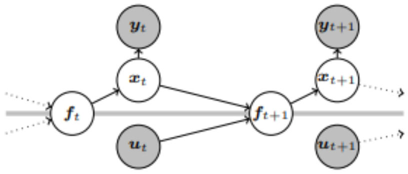


Figure 1. Graphical model of the PR-SSM. Gray nodes are observed variables in contrast to latent variables in white nodes. Thick lines indicate variables, which are jointly Gaussian under a GP prior.

Where $p(\boldsymbol{f}_t|\boldsymbol{x}\hat{_{t-1}}, \boldsymbol{z}) = \prod_{d=1}^{D_x} p(f_{t,d}|\boldsymbol{x}\hat{_{t-1}}, \boldsymbol{z}_d)$ is the likelihood across the D_x dimensions of the input, $p(\boldsymbol{y}_t|\boldsymbol{x}_t) = \mathcal{N}(\boldsymbol{y}_t|g(\boldsymbol{x}_t), \text{diag}(\sigma^2_{y,1}, ..., \sigma^2_{y,D_y}))$ is the distribution of outputs conditional on the latent state, and $p(\boldsymbol{x}_t|\boldsymbol{f}_t) = \mathcal{N}(\boldsymbol{x}_t|\boldsymbol{f}_t, \text{diag}(\sigma^2_{x,1}, ..., \sigma^2_{x,D_x}))$ is the noise of the Gaussian process. Note that g (as well as f) are fully generic here, and thus can be any model we like (including parameterized neural networks).

Unfortunately, due to the nonlinear Gaussian process dynamics model for the latent state, making a log likelihood or posterior distribution $p(\boldsymbol{f}_{2:T}.\boldsymbol{x}_{2:T}, \boldsymbol{z}|\boldsymbol{y}_{1:T})$ for PR-SSM is intractable. Thus, the paper uses variational inference to learn a distribution $q(\boldsymbol{f}_{2:T}, \boldsymbol{x}_{2:T}, \boldsymbol{z})$ that approximates p. Below, we show two formulations of q; the left one is a q from previous work, whereas the right one is the one used by PR-SSM (where $q(\boldsymbol{x}_1) = \mathcal{N}(\hat{\boldsymbol{x}_1}|\boldsymbol{\mu}_{x_1}, \boldsymbol{\sigma}_{x_1})$ and $q(\boldsymbol{z}_d) = \mathcal{N}(\hat{\boldsymbol{z}_d}|\boldsymbol{\mu}_d, \boldsymbol{\sigma}_d)$:

$$q(\boldsymbol{x}_{1:T}, \boldsymbol{f}_{2:T}, \boldsymbol{z}) = \left[\prod_{d=1}^{D_x} q(\boldsymbol{z}_d)\left[\prod_{t=2}^{T} p(f_{t,d} \mid \hat{\boldsymbol{x}}_{t-1}, \boldsymbol{z}_d)\right]\right]$$
$$\left[\prod_{t=1}^{T} q(\boldsymbol{x}_t)\right].$$

$$q(\boldsymbol{x}_{1:T}, \boldsymbol{f}_{2:T}, \boldsymbol{z}) = \left[\prod_{t=2}^{T} p(\boldsymbol{x}_t \mid \boldsymbol{f}_t)\right] q(\boldsymbol{x}_1)\cdot \quad (15)$$
$$\left[\prod_{t=2}^{T}\prod_{d=1}^{D_x} p(f_{t,d} \mid \hat{\boldsymbol{x}}_{t-1}, \boldsymbol{z}_d)q(\boldsymbol{z}_d)\right],$$

Let us briefly compare these two models. The q from previous work grows linearly with respect to the length of the time series, whereas PR-SSM's estimate does not depend on the time series length. In addition, the left q doesn't say how to initialize the first latent state variable $\boldsymbol{x}_1$, which is non-trivial, whereas $\boldsymbol{x}_1$ for PR-SSM has a defined distribution. Something more subtle is that the left model does not account for correlations between timesteps. Dealing with this was, up to the paper's publication, an open problem, one that the authors intended to address by adding the distribution $p(\boldsymbol{x}_t|\boldsymbol{f}_t)$ as a factor.

Using variational inference techniques, the authors were able to derive two things; a loss for PR-SSM (left), and an unbiased estimator for the term in the loss (right), shown below:

$$\mathcal{L}_{\text{PR-SSM}} = \sum_{t=1}^{T} \mathbb{E}_{q(\boldsymbol{x}_t)}[\log p(\boldsymbol{y}_t \mid \boldsymbol{x}_t)]$$
$$- \sum_{d=1}^{D_x} \text{KL}(q(\boldsymbol{z}_d) \parallel p(\boldsymbol{z}_d; \zeta_d)). \quad (17)$$

$$\mathbb{E}_{q(\boldsymbol{x}_t)}[\log(\boldsymbol{y}_t \mid \boldsymbol{x}_t)] \approx \frac{1}{N}\sum_{i=1}^{N} \log p(\boldsymbol{y}_t \mid \tilde{\boldsymbol{x}}_t^{(i)}). \quad (19)$$

Benefits: PR-SSM can be applied to a wide variety of systems since the transition model f and observation model g can be any model we like, including neural networks on unconventional data (e.g. images). In addition, training the model can be done flexibly, allowing the user to trade off performance for speed. PR-SSM also has many potential applications, two of which are listed below:

- state prediction in imperfect information environments: methods like AlphaZero are able to learn powerful policies through MCTS, however, MCTS becomes less effective in tasks with limited information such as DOTA or Minecraft. A state space model that allows you to predict environment dynamics in situations like this is invaluable (as evidenced by the fact Dreamer, an RL algorithm that uses state space models, was the first to be able to obtain diamonds in Minecraft without any human intervention or finetuning).

- Finance: time series data is extremely prevalent in finance, and often has a ton of latent state that affects its dynamics. For example, stock and bond prices can be modelled as a time series. Since time series are a special kind of dynamical system, they are well suited to be tackled by a state space model such as PR-SSM.

Drawbacks: Training PR-SSM on large datasets with high dimensionality can get very costly, especially when using the full gradient for updating model parameters. Thus, how exactly the model should be trained needs to be handled on a case-by-case basis depending on the domain one wishes to apply PR-SSM to.

Experiments: PR-SSM was compared with previous state-space models through various different benchmarks. For the benchmarks focused on model learning, PR-SSM was either the top performing model or one of the top contenders; the results are summarized in the figure below:

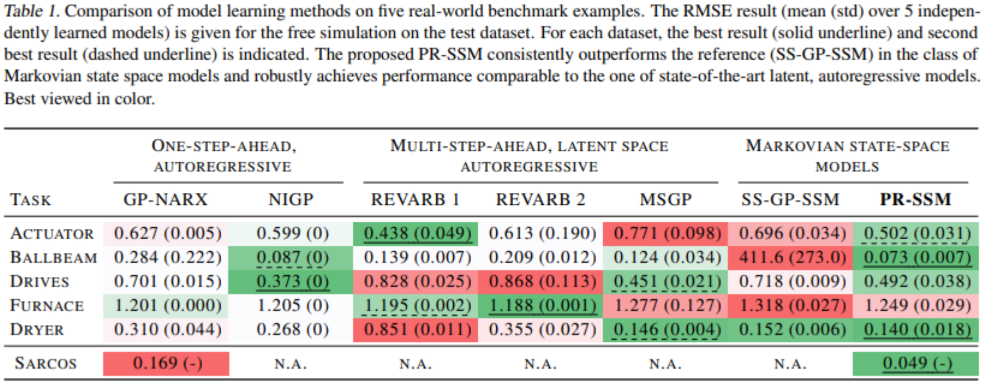


Table 1. Comparison of model learning methods on five real-world benchmark examples. The RMSE result (mean (std) over 5 independently learned models) is given for the free simulation on the test dataset. For each dataset, the best result (solid underline) and second best result (dashed underline) is indicated. The proposed PR-SSM consistently outperforms the reference (SS-GP-SSM) in the class of Markovian state space models and robustly achieves performance comparable to the one of state-of-the-art latent, autoregressive models. Best viewed in color.

	ONE-STEP-AHEAD, AUTOREGRESSIVE		MULTI-STEP-AHEAD, LATENT SPACE AUTOREGRESSIVE			MARKOVIAN STATE-SPACE MODELS	
TASK	GP-NARX	NIGP	REVARB 1	REVARB 2	MSGP	SS-GP-SSM	**PR-SSM**
ACTUATOR	0.627 (0.005)	0.599 (0)	0.438 (0.049)	0.613 (0.190)	0.771 (0.098)	0.696 (0.034)	0.502 (0.031)
BALLBEAM	0.284 (0.222)	0.087 (0)	0.139 (0.007)	0.209 (0.012)	0.124 (0.034)	411.6 (273.0)	0.073 (0.007)
DRIVES	0.701 (0.015)	0.373 (0)	0.828 (0.025)	0.868 (0.113)	0.451 (0.021)	0.718 (0.009)	0.492 (0.038)
FURNACE	1.201 (0.000)	1.205 (0)	1.195 (0.002)	1.188 (0.001)	1.277 (0.127)	1.318 (0.027)	1.249 (0.029)
DRYER	0.310 (0.044)	0.268 (0)	0.851 (0.011)	0.355 (0.027)	0.146 (0.004)	0.152 (0.006)	0.140 (0.018)
SARCOS	0.169 (-)	N.A.	N.A.	N.A.	N.A.	N.A.	0.049 (-)

Detailed graphs of the resulting predictions given these models are shown below; note that PR-SSM is better able to quantify uncertainty and is generally quite close to the ground truth output.

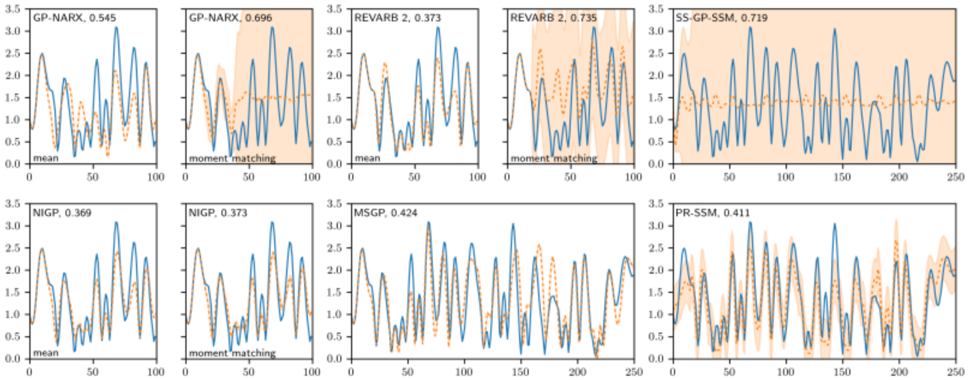


Figure 4. Free simulation results for the benchmark methods on the *Drives* test dataset. The true, observed system output (blue) is compared to the individual model's predictive output distribution (orange, mean +/- two std). Results are presented for the one-step-ahead models GP-NARX and NIGP in the left column. REVARB and MSGP (shown in the middle column) are both based on multi-step optimized autoregressive GP models in latent space. In the right column, the SS-GP-SSMs, as a model based on a Markovian latent state, is compared to the proposed PR-SSM.

Links:

Original Paper

1.2 Network Architecture Papers

1.2.1 Attention Is All You Need

Prerequisites: Understanding of residual connections, perhaps through reading the paper that introduced it, **Deep Residual Learning for Image Recognition**.

Motivation: Recurrent neural networks with attention are, at the time of this paper's publication, the state of the art in sequence processing. However, a huge problem with recurrent neural networks is that they must be trained sequentially through time, which is very inefficient for larger models. Convolutional neural networks are also used for sequence processing at this time; while they can be trained using parallel compute platforms like GPUs, their problem is that the number of operations needed to connect information from two different positions in a sequence scales in the distance between those positions, which also causes computational inefficiency. An architecture that can process sequences effectively and efficiently without the use of recurrence or convolutions is thus desired.

Basic Idea: This paper presents a new architecture, called the Transformer model, that performs sequence processing exclusively using attention mechanisms. In order to understand the Transformer, one first needs to understand attention, and from there, self-attention.

Attention is essentially a mapping that takes in 3 things; a set of queries we want to search for, a set of available keys, and a set of values for those keys, all of which are real-valued vectors. The idea behind attention is to return a set of values for each query based on how similar they are to the keys. How similarity between keys and queries is concretely computed can vary; the method used in this paper is known as Scaled Dot-Product Attention. First, some notation: let d_q be the dimensionality of the queries, d_{model} be the number of query-key-value triples the model considers simultaneously, d_k be the dimensionality of the keys, and d_v be the dimensionality of the values.

Now let $Q \in \mathbb{R}^{d_q \times d_{\text{model}}}$ be the matrix of queries, $K \in \mathbb{R}^{d_k \times d_{\text{model}}}$ be the matrix of keys, and $V \in \mathbb{R}^{d_v \times d_{\text{model}}}$ be the matrix of values. The attention function Attention(Q, K, V) is computed as follows: Attention$(Q, K, V) = \mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})V$. Breaking the pieces of this down: QK^T is a dot product of each query-key pair put into a matrix, effectively computing their similarity. The $\sqrt{d_k}$ term is present as a scaling factor to ensure the dot product doesn't grow too large in magnitude into areas where the $\mathrm{softmax}$ function has small gradients. $\mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})$ converts the similarities into coefficients that sum to 1, so that when multiplied with V, will act as a weighted sum over the possible values, producing the estimated values for each of the given queries.

While having a single attention function is nice, the power of the Transformer model comes from its exploitation of Multi Headed Attention. Multi Headed Attention uses multiple blocks of attention, each being a function of projections of the query, key and value matrices. More concretely, let h be the desired number of attention "heads". For $i = 1, ..., h$, let head$_i$ = Attention(QW_i^Q, KW_i^K, VW_i^V), where $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ are parameter matrices. We have that MultiHeadAttention(Q, K, V) = Concat(head$_1, ...,$ head$_h)W^O$, where $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$ is another parameter matrix. Intuitively, multi headed attention allows the Transformer to attend to different aspects of the sequence at different positions, thus increasing its predictive power.

The Transformer model itself is structured into two main parts; the encoder, which takes in embeddings of tokens from the sequence and converts them into semantically meaningful encodings for the model, and the decoder, which receives the encoder's output and uses it (and other information) to output a probability distribution over tokens in the sequence. A summary of the architecture is shown below.

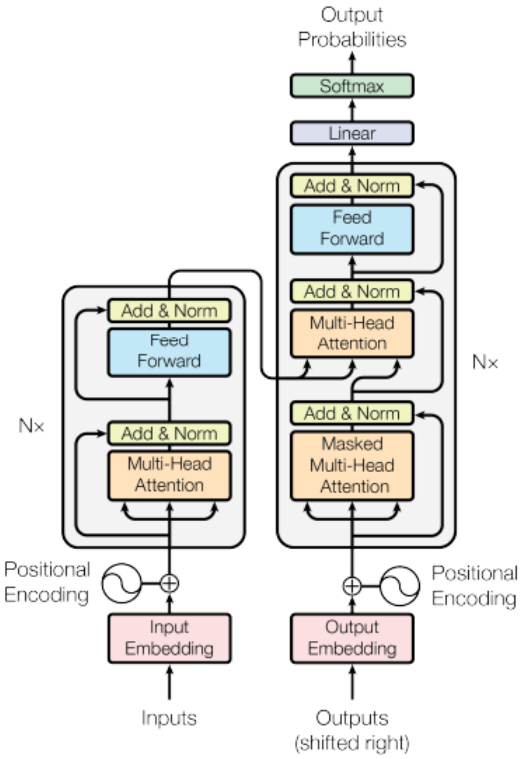


Figure 1: The Transformer - model architecture.

The encoder is a connected sequence of N identical layers. The initial input to the first encoder are the embeddings for all the tokens in the sequence. A positional encoding is attached so that there is information regarding the position of a token embedded into the input. In the first layer, the queries, keys, and values are all the same, i.e. they are simply the values of the input embeddings. These embeddings pass through various layers, including a multi headed attention block, a layer normed residual block, a feed forward network block, and another layer normed residual block. The output of this encoder layer is fed into the next encoder layer, and so on until the final layer is reached.

The decoder is also a connected sequence of N identical layers. The initial input to the first encoder are the embeddings for all the tokens in the sequence, and as before, a positional encoding is used. In the first layer, the queries, keys, and values are also the same, and are fed into the attention portion and a d_{model}-dimensional vector is outputted. However, this time, there is a mask that prevents tokens from beyond a certain point from being considered; this is because the Transformer is designed to be an autoregressor, i.e. a model that predicts a future token given past ones.

After this, the vector is passed through a layer normed residual block, and is passed as the value parameter into another multi headed attention block. The query and key parameters for this block come from the encoder's output. There is then another layer normed residual block, then a feed forward layer, followed by another layer normed residual block. The output of this decoder layer is fed as input into the next decoder layer (with the encoder's output playing the same role across all layers), until the final decoder layer is reached. At this point, the output is passed through a linear projection, then is softmaxed to produce a probability distribution over tokens.

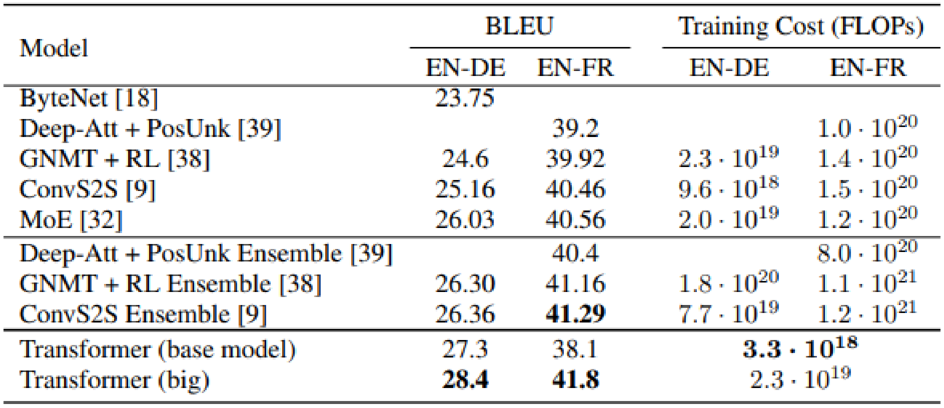
Benefits: The Transformer model addresses both problems with recurrence and convolutions; its weights are able to be trained in parallel because it is not time-dependent, and distance is not a factor when computing dependencies between tokens. Its usage of multi headed attention allows it to have each head pay ”attention” to something different within the structure of the sequences the Transformer processes. In addition, due to the architecture admitting arbitrary values for the number of encoder and decoder layers, as well as for the dimensionality constants, the Transformer can be arbitrarily scaled up or down as needed.

Drawbacks: The main drawback of the Transformer is its complexity grows very quickly with respect to the sequence length and dimensionality of the input embeddings. More concretely, if n is the sequence length, and d is the dimensionality of token embeddings, then each layer in the encoder and the decoder has $O(n^2 * d)$ complexity. This means for very long sequences, the Transformer model becomes extremely complex, which can take up a lot of time and memory in practice.

Experiments: The experiments were done in two main areas: neural machine translation, the act of using neural networks to translate between languages, and English constituency parsing, i.e. the analysis of the grammatical structure of sentences. The Transformer was compared against a variety of previous models that used recurrence and/or convolutions.

In neural machine translation, the Transformer outperformed all other models, despite training for only a fraction of the time and costing significantly less in terms of floating point operations (FLOPs). The results are summarized in the below figure:

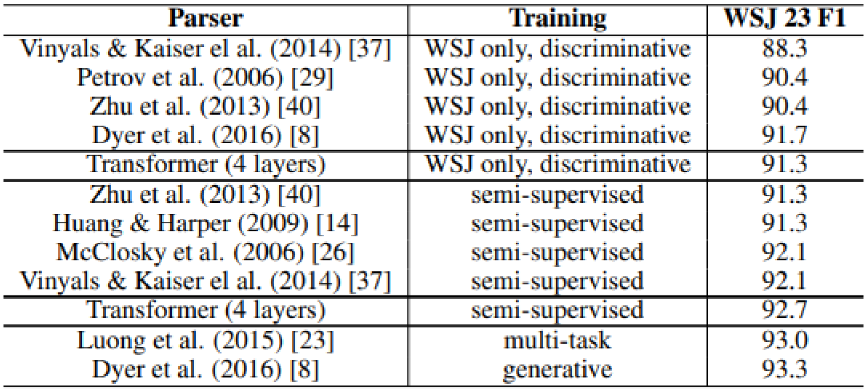
Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.



Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

In English constituency parsing, the Transformer was able to outperform almost all previous models, despite not being specifically tuned for this task. The results are summarized in the below figure:

Table 4: The Transformer generalizes well to English constituency parsing (Results are on Section 23 of WSJ)



Parser	Training	WSJ 23 F1
Vinyals & Kaiser el al. (2014) [37]	WSJ only, discriminative	88.3
Petrov et al. (2006) [29]	WSJ only, discriminative	90.4
Zhu et al. (2013) [40]	WSJ only, discriminative	90.4
Dyer et al. (2016) [8]	WSJ only, discriminative	91.7
Transformer (4 layers)	WSJ only, discriminative	91.3
Zhu et al. (2013) [40]	semi-supervised	91.3
Huang & Harper (2009) [14]	semi-supervised	91.3
McClosky et al. (2006) [26]	semi-supervised	92.1
Vinyals & Kaiser el al. (2014) [37]	semi-supervised	92.1
Transformer (4 layers)	semi-supervised	92.7
Luong et al. (2015) [23]	multi-task	93.0
Dyer et al. (2016) [8]	generative	93.3

Links:

Original Paper

1.3 Algorithm Supplements

1.4 Systems/Applications

1.5 Learning Theory

2 Computer Vision

2.1 Algorithm Papers

2.2 Network Architecture Papers

2.3 Algorithm Supplements

2.4 Systems/Applications

2.5 Learning Theory

3 NLP

3.1 Algorithm Papers

3.2 Network Architecture Papers

3.3 Algorithm Supplements

3.4 Systems/Applications

3.5 Learning Theory

4 Reinforcement Learning

4.1 Algorithm Papers

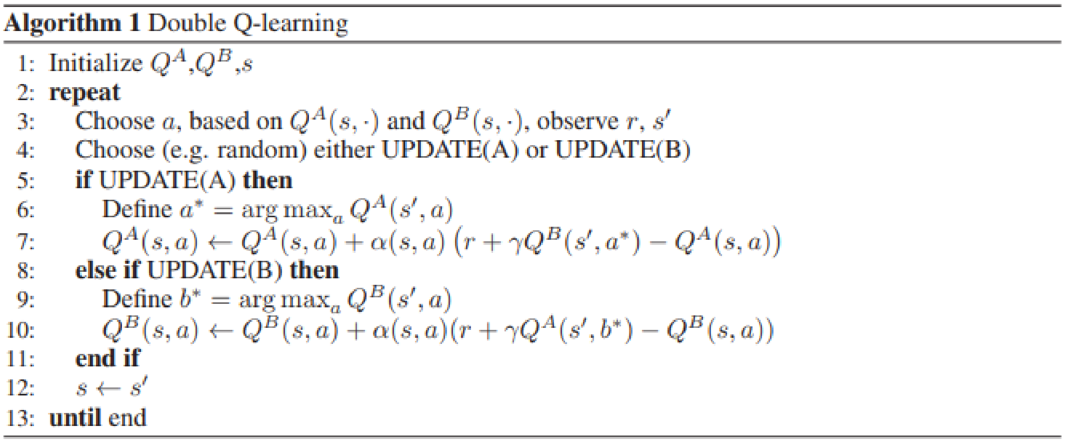
4.1.1 Deep Reinforcement Learning with Double Q-Learning

Prerequisites: an understanding of Q learning as well as Deep Q Networks is required.

Motivation: In classic Deep Q learning, the same network $Q(s, a; \theta)$ is used for both action selection and evaluation, which often leads to overestimating action values in practice. Overestimating action values can harm the performance of agents on control tasks, because the overestimates can cause actions that are actually suboptimal to be considered optimal. Thus, a method that disconnects action selection and evaluation during training would be ideal.

Basic Idea: With Double Q Learning, two networks are used: $Q^A(s, a; \theta)$ and $Q^B(s, a; \theta')$, where s is the current state, a is the action the agent wishes to take, and θ and θ' are parameters. When updating Q^A, we consider its ability to select actions, and use Q^B as an unbiased evaluation of those actions. Conversely, updating Q^B has Q^A evaluate the actions that Q^A perceives as optimal.

Pseudocode (not mine, credit goes to Hasselt): In this algorithm, Q^A and Q^B are as described above, and s is the initial state of the environment the agent resides in.



Algorithm 1 Double Q-learning

```
Initialize $Q^A$,$Q^B$,$s$
repeat
    Choose $a$, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$, observe $r$, $s'$
    Choose (e.g. random) either UPDATE(A) or UPDATE(B)
    if UPDATE(A) then
        Define $a^* = \arg\max_a Q^A(s',a)$
        $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\left(r + \gamma Q^B(s',a^*) - Q^A(s,a)\right)$
    else if UPDATE(B) then
        Define $b^* = \arg\max_a Q^B(s',a)$
        $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s',b^*) - Q^B(s,a))$
    end if
    $s \leftarrow s'$
until end
```

Benefits: Double Q Learning allows for unbiased estimators to be used in the update rule, which ensures action values are not overestimated nearly as often. This allows agents learning with Double Q Learning to usually outperform agents with classic Q learning, as they have a better estimate of action values.

Drawbacks: Double Q Learning can still underestimate action values, which means they are not completely immune to possible action value estimate errors in sampling the control task.

Experiments: Two agents were trained on various Atari 2600 games; one used a neural network with classic Q learning, the other used two neural networks and Double Q Learning. These experiments showed two things:

- Deep Q Learning's overestimation issue is quite severe, and can lead to worsened performance in practice, and

- Double Deep Q Learning mitigates this overestimation, and achieves better performance as a result.

Below is a figure which summarizes the results.

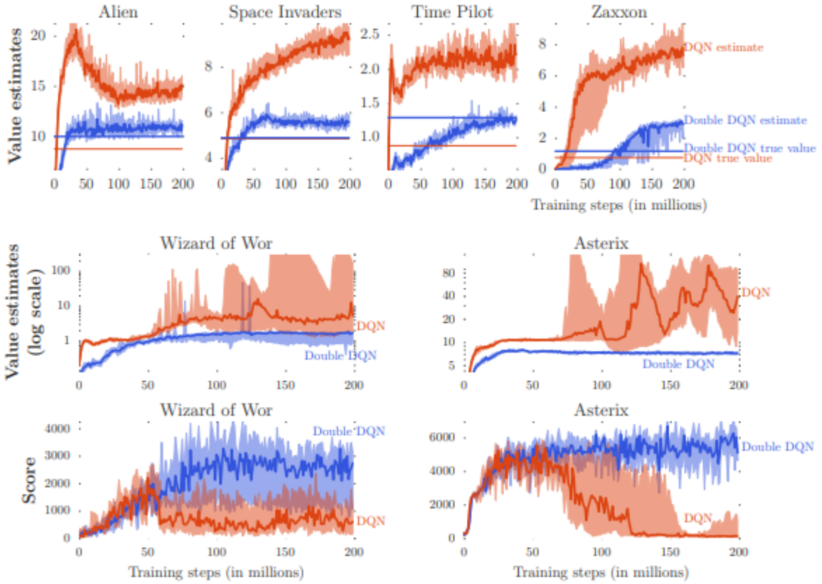


Figure 3: The **top** and **middle** rows show value estimates by DQN (orange) and Double DQN (blue) on six Atari games. The results are obtained by running DQN and Double DQN with 6 different random seeds with the hyper-parameters employed by Mnih et al. (2015). The darker line shows the median over seeds and we average the two extreme values to obtain the shaded area (i.e., 10% and 90% quantiles with linear interpolation). The straight horizontal orange (for DQN) and blue (for Double DQN) lines in the top row are computed by running the corresponding agents after learning concluded, and averaging the actual discounted return obtained from each visited state. These straight lines would match the learning curves at the right side of the plots if there is no bias. The **middle** row shows the value estimates (in log scale) for two games in which DQN's overoptimism is quite extreme. The **bottom** row shows the detrimental effect of this on the score achieved by the agent as it is evaluated during training: the scores drop when the overestimations begin. Learning with Double DQN is much more stable.

Links:
Original Paper

Double Q Learning Paper

4.1.2 IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures

Prerequisites: An understanding of Deep Q Networks, as well as asynchronous methods for deep RL, is critical.

Motivation: Asynchronous methods for deep RL, such as A3C, achieve stellar performance on control tasks like various Atari 2600 games. However, the issue with RL methods at the time was that they did not scale very well; it would sometimes take billions of frames and days of computation to master a single task. An RL algorithm capable of scaling upward without sacrificing training stability or data efficiency is desired.

Basic Idea: IMPALA uses an actor-critic based setup to learn two things; a policy π and a baseline state-value function V^{π}. The architecture contains a set of actors that repeatedly generate experience trajectories in the environment, and one or more learners that use the experience generated by the actors to update the policy. At the start of each trajectory, an actor updates its policy μ to the learner's policy π and runs it for n steps in the environment. This generates a trajectory $s_1, a_1, r_1, ..., s_n, a_n, r_n$, as well as a set of policy distributions $\mu(a_1|s_1), ..., \mu(a_n|s_n)$. These two, along with other relevant information (such as the initial state in a recurrent neural network) are sent to the learner. The learner(s) update the policy π using minibatches of trajectories. It is important to note that because of the asynchronous aspects of IMPALA, there will probably be some discrepancy between π and μ; thus, an off-policy algorithm is needed to account for this discrepancy, exploiting the high data throughput of IMPALA while maintaining data efficiency. The paper introduces an algorithm called V-trace that solves this problem.

V-trace is an off-policy algorithm created to best exploit the advantages of IMPALA. Let $\overline{p}, \overline{c}$ be hyperparameters with $\overline{p} \geq \overline{c}$, and let $\gamma \in [0, 1]$ be the discount factor. In addition, let $(s_t, a_t, r_t)_{t=l}^{t=l+n}$ be a trajectory generated by the policy μ, where l is a starting timestep. The n-steps V-trace target for $V(s_l)$ is defined as follows: $v_l = V(s_l) + \sum_{t=l}^{l+n-1}(\gamma^{t-l}(\prod_{i=s}^{t-1} c_i)p_t(r_t + \gamma V(s_{t+1}) - V(s_t)))$, where $c_i = \min(\overline{c}, \frac{\pi(a_i|s_i)}{\mu(a_i|s_i)})$ and $p_i = \min(\overline{p}, \frac{\pi(a_i|s_i)}{\mu(a_i|s_i)})$.

The weights c_i and p_i play fundamentally different roles. The weights p_i are a part of the temporal difference, $p_t(r_t + \gamma V(s_{t+1}) - V(s_t))$, and act as an interpolation between a value function for the learner policy π and the actor policy μ. The weights c_i measures how much the temporal difference at time t impacts the update of the value function at the previous time l. Intuitively, p_i impacts the nature of the value function that we consider, whereas c_i impacts the speed at which we want to adjust towards said value function.

Putting this all together, the algorithm is relatively simple; consider parametric representations of the value function V_θ and of the current policy π_ω. A trajectory beginning at time l has been generated by a behavior policy μ, along with its corresponding policy distributions. The value parameters θ are updated in the direction of $(v_l - V_\theta(s_l))\nabla_\theta V_\theta(s_l)$, and the policy parameters ω are updated in the direction of the policy gradient: $p_s \nabla_\omega \log \pi_\omega(a_l|s_l)(r_l + \gamma v_{l+1} - V_\theta(s_l))$.

Benefits: IMPALA's architecture in terms of speed is several times faster than single machine A3C, the previous state of the art in reinforcement learning. In addition, IMPALA has higher data throughput than A3C and is more robust to changing hyperparameter values and network architectures.

Drawbacks: The main drawback of IMPALA is that the discrepancy between the learner policy and actor policy necessitates the use of off-policy algorithms, limiting the algorithms that can exploit IMPALA's advantages.

Experiments: Several experiments were done comparing IMPALA to previous methods; the first tested data throughput, measuring how many frames per second various algorithms could process. IMPALA achieved higher frames per second than its A3C counterparts, even while using less CPUs. It even outperformed a GPU-assisted A3C without the use of a GPU itself. Another experiment tested how various RL frameworks performed with respect to frames consumed and time trained. In both areas, all variants of IMPALA outperformed A3C, with a significany discrepancy in performance in the time trained metric. The below figures summarize the results:

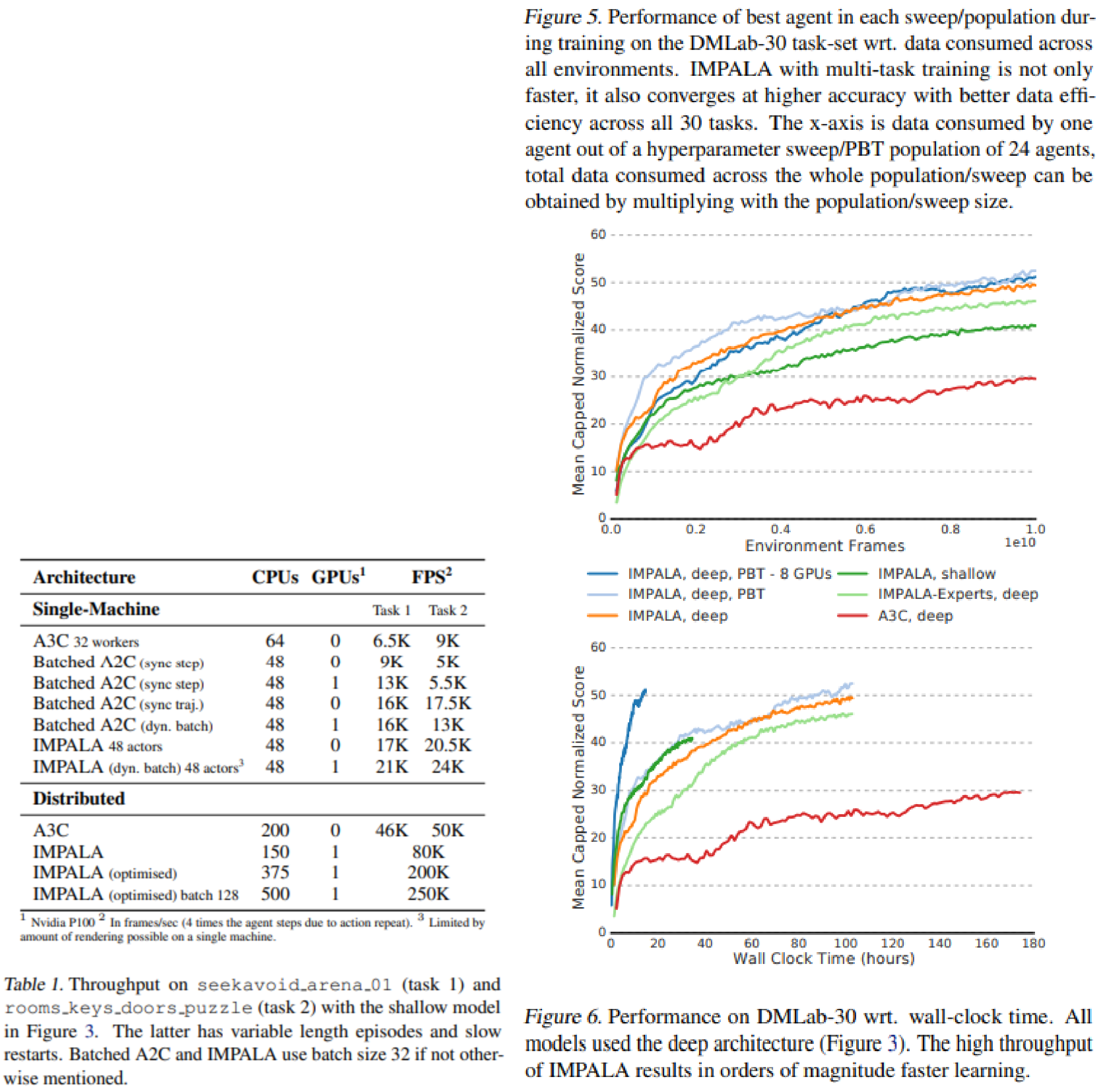


Figure 5. Performance of best agent in each sweep/population during training on the DMLab-30 task-set wrt. data consumed across all environments. IMPALA with multi-task training is not only faster, it also converges at higher accuracy with better data efficiency across all 30 tasks. The x-axis is data consumed by one agent out of a hyperparameter sweep/PBT population of 24 agents, total data consumed across the whole population/sweep can be obtained by multiplying with the population/sweep size.

Architecture	CPUs	GPUs[1]	FPS[2]	
Single-Machine			Task 1	Task 2
A3C 32 workers	64	0	6.5K	9K
Batched A2C (sync step)	48	0	9K	5K
Batched A2C (sync step)	48	1	13K	5.5K
Batched A2C (sync traj.)	48	0	16K	17.5K
Batched A2C (dyn. batch)	48	1	16K	13K
IMPALA 48 actors	48	0	17K	20.5K
IMPALA (dyn. batch) 48 actors[3]	48	1	21K	24K
Distributed				
A3C	200	0	46K	50K
IMPALA	150	1	80K	
IMPALA (optimised)	375	1	200K	
IMPALA (optimised) batch 128	500	1	250K	

[1] Nvidia P100 [2] In frames/sec (4 times the agent steps due to action repeat). [3] Limited by amount of rendering possible on a single machine.

Table 1. Throughput on `seekavoid_arena_01` (task 1) and `rooms_keys_doors_puzzle` (task 2) with the shallow model in Figure 3. The latter has variable length episodes and slow restarts. Batched A2C and IMPALA use batch size 32 if not otherwise mentioned.

Figure 6. Performance on DMLab-30 wrt. wall-clock time. All models used the deep architecture (Figure 3). The high throughput of IMPALA results in orders of magnitude faster learning.

Links:

Original Paper

4.1.3 Mastering Diverse Domains through World Models

Prerequisites: the reader should have a solid understanding of deep reinforcement learning, recurrent neural networks, and actor-critic methods.

Motivation: Up to this paper's publication, reinforcement learning algorithms had been developed that were very well suited to learning tasks within their respective domains. However, making said algorithms ready for tasks outside their domain requires careful human finetuning. A reinforcement learning algorithm that can generalize to a diverse set of tasks without the need for human intervention is desirable.

Basic Idea: The DreamerV3 algorithm is a reinforcement learning algorithm that takes a different approach compared to most. It possesses 3 components; a world model that takes in the current state and predicts future states, rewards, and whether the episode will continue, a critic that judges the value of the outcomes, and the actor uses the critic's information to make decisions. The 3 components are described in more detail below.

The world model is a Recurrent State Space Model (RSSM). Let t be the current timestep. First, the input x_t is encoded into an embedding z_t. A sequence model with internal state h_{t-1} gets a new internal state h_t from the previous state, the embedding, and the past action a_{t-1}. h_t is then used to predict future embedded inputs. The model then uses z_t and h_t together to predict future rewards $\hat{r}_t$, whether the episode will continue in the form of $\hat{c}_t \in \{0, 1\}$, and also predicts the next input $\hat{x}_t$ it will receive. The following equations concretely illustrate this description, where f_ϕ is the transition operator for the internal state, q_ϕ and p_ϕ are various distributions, and ϕ are the model parameters.

$$\begin{array}{lll} \text{RSSM} \left\{ \begin{array}{l} \text{Sequence model:} \\ \text{Encoder:} \\ \text{Dynamics predictor:} \end{array} \right. & & \begin{array}{l} h_t = f_\phi(h_{t-1}, z_{t-1}, a_{t-1}) \\ z_t \sim q_\phi(z_t \mid h_t, x_t) \\ \hat{z}_t \sim p_\phi(\hat{z}_t \mid h_t) \end{array} \\ \qquad\quad\;\; \text{Reward predictor:} & & \hat{r}_t \sim p_\phi(\hat{r}_t \mid h_t, z_t) \\ \qquad\quad\;\; \text{Continue predictor:} & & \hat{c}_t \sim p_\phi(\hat{c}_t \mid h_t, z_t) \\ \qquad\quad\;\; \text{Decoder:} & & \hat{x}_t \sim p_\phi(\hat{x}_t \mid h_t, z_t) \end{array} \tag{1}$$

Given a sequence of inputs $x_{1:T}$, actions $a_{1:T}$, rewards $r_{1:T}$ and continuation flags $c_{1:T}$, the loss for training the world model is as follows: $\mathcal{L}(\phi) = \mathbb{E}_{q_\phi}[\sum_{t=1}^{T} (\beta_{\text{pred}}\mathcal{L}_{\text{pred}}(\phi) + \beta_{\text{dyn}}\mathcal{L}_{\text{dyn}}(\phi) + \beta_{\text{rep}}\mathcal{L}_{\text{rep}}(\phi))]$, where $\mathcal{L}_{\text{pred}}(\phi) = -\log p_\phi(x_t|z_t, h_t) - \log p_\phi(r_t|z_t, h_t) - \log p_\phi(c_t|z_t, h_t)$ is the prediction loss, $\mathcal{L}_{\text{dyn}}(\phi) = \max(1, \text{KL}[\text{sg}(q_\phi(z_t|x_t, h_t)) \| p_\phi(z_t|h_t)])$ is the dynamics loss, and $\mathcal{L}_{\text{rep}}(\phi) = \max(1, \text{KL}[q_\phi(z_t|x_t, h_t) \| \text{sg}(p_\phi(z_t|h_t))])$ is the representation loss.

The critic, $v_\psi(R_t|s_t)$, where s_t is the state being read (one can have it where $s_t = x_t$, in which case the critic receives the input directly, however, that may not be the case), is a neural network that attempts to estimate the discounted return $v_t = \mathbb{E}[v_\psi(R_t|s_t)]$ from the current actor's behavior (the expectation is with respect to a probability distribution $p_\psi(\dot{|}s_t)$. Using representations of replayed inputs, the actor and world model generate a sequence of imagined inputs $x_{1:T}$, actions $a_{1:T}$, rewards $r_{1:T}$ and continuation flags $c_{1:T}$. Let $\lambda \in [0, 1]$ be a hyperparameter. The critic learns to predict the distribution of return estimates $R_t^\lambda = r_t + \gamma c_t((1-\lambda)v_t + \lambda R_{t+1})$ ($R_T^\lambda = v_T$) using the following loss: $\mathcal{L}_(\psi) = -\sum_{t=1}^{T} \log p_\psi(R_t^\lambda|s_t)$.

The actor is the policy of the agent $\pi_\theta(a_t|s_t)$. This policy is optimized similar to the REINFORCE algorithm; its loss is as follows: $\mathcal{L}(\theta) = -\sum_{t=1}^{T}(sgn(\frac{R_t^\lambda - v_\psi(s_t)}{\max(1,S)}) \log \pi_\theta(a_t|s_t) + \eta\mathbb{H}[\pi_\theta(a_t|s_t)])$, where $\mathbb{H}$ is entropy, $\eta \in [0, 1]$ is a hyperparameter, and $S = \text{ExponentialMovingAverage}(\text{Per}(R_t^\lambda, 95) - \text{Per}(R_t^\lambda, 5), 0.99)$ is a scaling factor. Here $\text{Per}(X, n)$ denotes the nth percentile of the random variable X. The entropy term is there to encourage the actor to keep its options open when exploring.

Benefits: Because DreamerV3 is able to learn a world model for a wide variety of tasks, it is very generally applicable without the need for human finetuning. In addition, because its world model is so generic in nature, the same hyperparameters transfer across many different domains, making training much easier. The world model also enables the agent to solve tasks far more efficiently and effectively, even solving tasks that were previously unsolvable for RL algorithms.

Drawbacks: Due to the fact a lot of neural networks are present in the system, it can take a lot of memory and compute to store an effective Dreamer model.

Experiments: DreamerV3 was compared against various algorithms, such as IMPALA, Rainbow DQN, and PPO, on over 150 different tasks across 8 domains, some of which were Atari, Minecraft and Proprio Control. Dreamer beat every algorithm's performance across the same number of environment steps, with the competing algorithms even having more training data in some cases. A figure below summarizes the results.

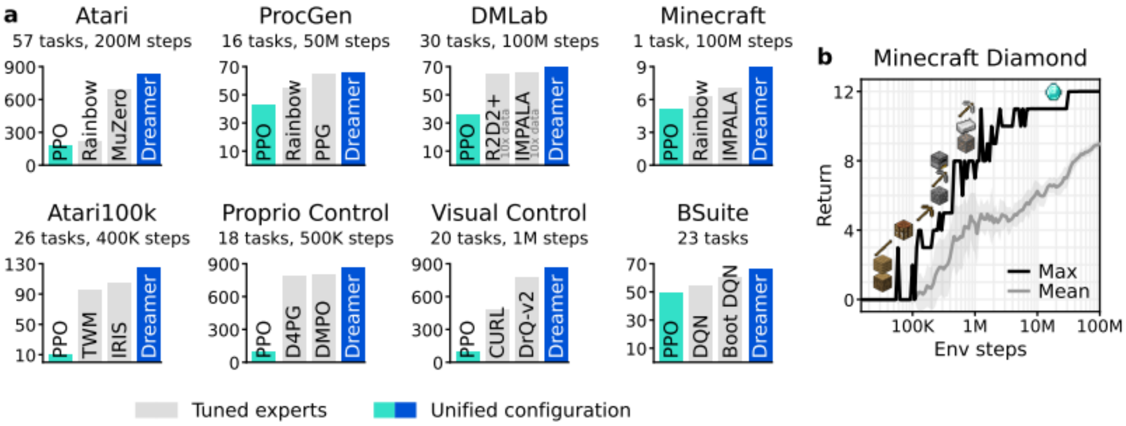


Figure 1: Benchmark summary. **a**, Using fixed hyperparameters across all domains, Dreamer outperforms tuned expert algorithms across a wide range of benchmarks and data budgets. Dreamer also substantially outperforms a high-quality implementation of the widely applicable PPO algorithm. **b**, Applied out of the box, Dreamer learns to obtain diamonds in the popular video game Minecraft from scratch given sparse rewards, a long-standing challenge in artificial intelligence for which previous approaches required human data or domain-specific heuristics.

4.2 Network Architecture Papers

4.2.1 Dueling Network Architectures for Deep Reinforcement Learning

Prerequisites: an understanding of Q learning as well as Deep Q Networks is required. Understanding of Double Q Learning is an asset but not required.

Motivation: Algorithms such as Q learning learn a joint state-action value function, enabling them to select optimal actions in given states. However, the action taken in a given state is not always important. In contrast, for bootstrapping algorithms (algorithms that sample environment runs and use those runs to estimate state and action values), the state value is considered important for every state. Thus, it would be ideal if we could obtain separate state-value and action-value estimates, and combine them if desired.

Basic Idea: Rather than computing $Q(s, a)$ directly (where s is a given state and a is a given action), we instead compute the state-value function $V(s)$ and advantage function $A(s, a)$, and exploit the fact that $Q(s, a) = V(s) + A(s, a)$ by definition. Concretely, we would have $V(s; \theta, \beta)$ be an estimate for $V(s)$ depending on parameters θ and β, and $A(s, a; \theta, \alpha)$ be an estimate for $A(s, a)$ depending on parameters θ and α (note that our estimates for $V(s)$ and $A(s, a)$ share parameters). A visualization of this idea is shown below:

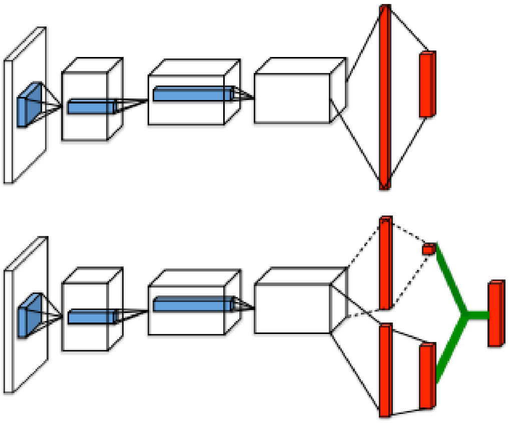


Figure 1. A popular single stream Q-network (**top**) and the dueling Q-network (**bottom**). The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; the green output module implements equation (9) to combine them. Both networks output Q-values for each action.

It may be tempting to define $Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha)$ for the output layer; however, this approach is not ideal. The main problem is that given Q, we cannot uniquely determine V or A; given guesses for V and A from Q adding a constant to V and subtracting the same constant from A would yield the same Q value. This lack of unique identifiability leads to poor performance in practice.

There are various different methods for solving this issue. One way is to define $Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + (A(s, a; \theta, \alpha) - \max_{a'} A(s, a'; \theta, \alpha))$. This leads to an advantage of 0 at the chosen action, and so at $a^* = \operatorname{argmax}_{a'} A(s, a'; \theta, \alpha)$, $Q(s, a^*; \theta, \alpha, \beta) = V(s; \theta, \beta)$, preserving the semantics of V and A while ensuring identifiability. Another way is to define $Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + (A(s, a; \theta, \alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a'; \theta, \alpha))$, where $\mathcal{A}$ is the set of possible actions. This makes V and A lose their semantics slightly, but makes optimization smoother.

Benefits: Since this architecture allows for separate estimates of $V(s)$, $A(s, a)$ and $Q(s, a)$, it allows an agent to separate the value of a state in and of itself from the advantage of actions in a given state. It is also more versatile and enables greater interpretability into how the agent perceives its environment. For example, $V(s)$ can be used to get an idea of an agent's general objective, while $A(s, a)$ can be used to see what actions the agent perceives as advantageous. In addition, since the final output of the architecture is still an estimate of $Q(s, a)$, it is compatible with any RL algorithm that uses it, such as Deep Q Networks, SARSA, or even things like Prioritized Experience Replay and Double Deep Q Networks.

Drawbacks: One must be careful when defining $Q(s, a; \theta, \alpha, \beta)$ in terms of $V(s; \theta, \beta)$ and $A(s, a; \theta, \alpha)$ to ensure that not only is identifiability preserved, but optimization through backpropagation is reasonably stable.

Experiments: The first experiment done was in a discrete environment where the true Q values were known, and the task was to obtain reasonable estimates of said Q values. 2 agents were used; one used a standard Q network architecture, while the other used a dueling Q network architecture with the same depth and similar neuron counts for each layer. As the number of actions to test increased, the dueling architecture was shown to converge faster than the classic Q network. Below is a figure that summarizes the results:

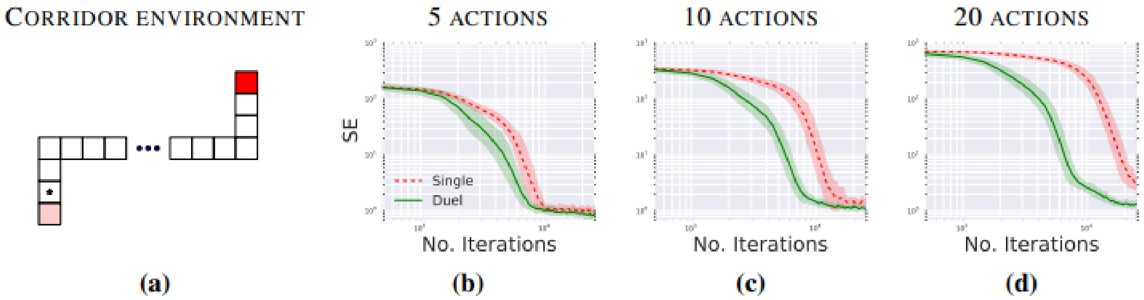


Figure 3. **(a)** The corridor environment. The star marks the starting state. The redness of a state signifies the reward the agent receives upon arrival. The game terminates upon reaching either reward state. The agent's actions are going up, down, left, right and no action. Plots **(b), (c)** and **(d)** shows squared error for policy evaluation with 5, 10, and 20 actions on a log-log scale. The dueling network (Duel) consistently outperforms a conventional single-stream network (Single), with the performance gap increasing with the number of actions.

In other experiments, two agents were trained on various Atari games. These agents would use methods ranging from classic Deep Q Networks to Double Deep Q Networks, and prioritized experience replay was also included in the experiments. Both agents were kept identical, with the exception of the architecture; one used a class deep Q network, while the other used a dueling Q network architecture. The following metric was used to evaluate the agents (the baseline score is the score of the agent with the classic architecture):

$$\frac{\text{Score}_{\text{Agent}} - \text{Score}_{\text{Baseline}}}{\max\{\text{Score}_{\text{Human}}, \text{Score}_{\text{Baseline}}\} - \text{Score}_{\text{Random}}}. \qquad (10)$$

In all scenarios, the agent with the duelling architecture tended to outperform the agent with the classic architecture; the results are summarized in two figures below:

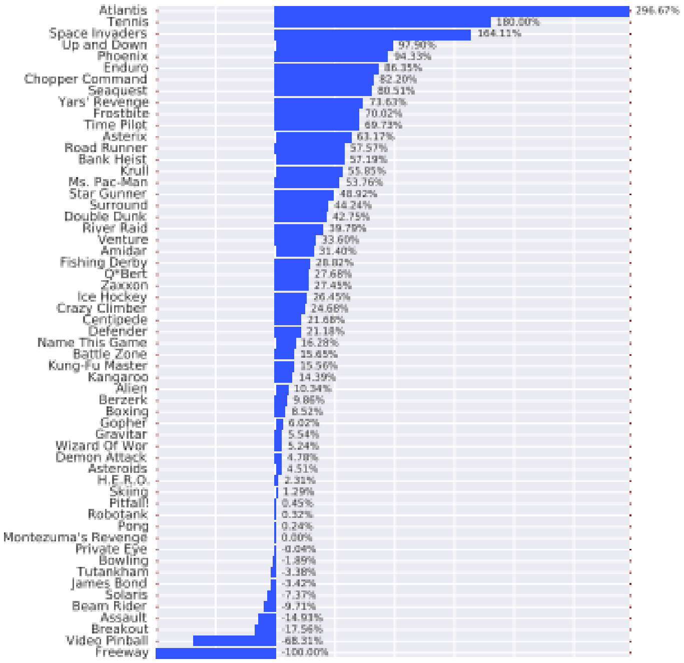


Figure 4. Improvements of dueling architecture over the baseline Single network of van Hasselt et al. (2015), using the metric described in Equation (10). Bars to the right indicate by how much the dueling network outperforms the single-stream network.

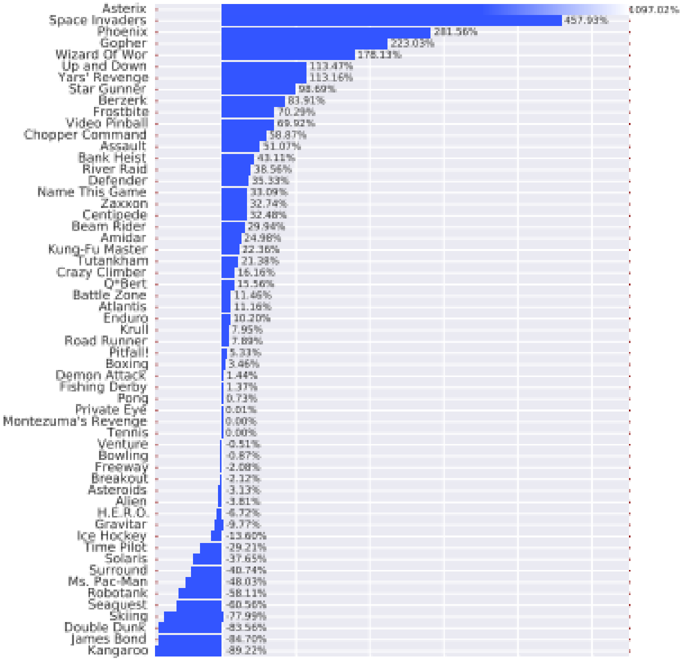


Figure 5. Improvements of dueling architecture over Prioritized DDQN baseline, using the same metric as Figure 4. Again, the dueling architecture leads to significant improvements over the single-stream baseline on the majority of games.

Links:

Original Paper

4.3 Algorithm Supplements

4.3.1 Prioritized Experience Replay

Prerequisites: an understanding of Q learning as well as Deep Q Networks is required.

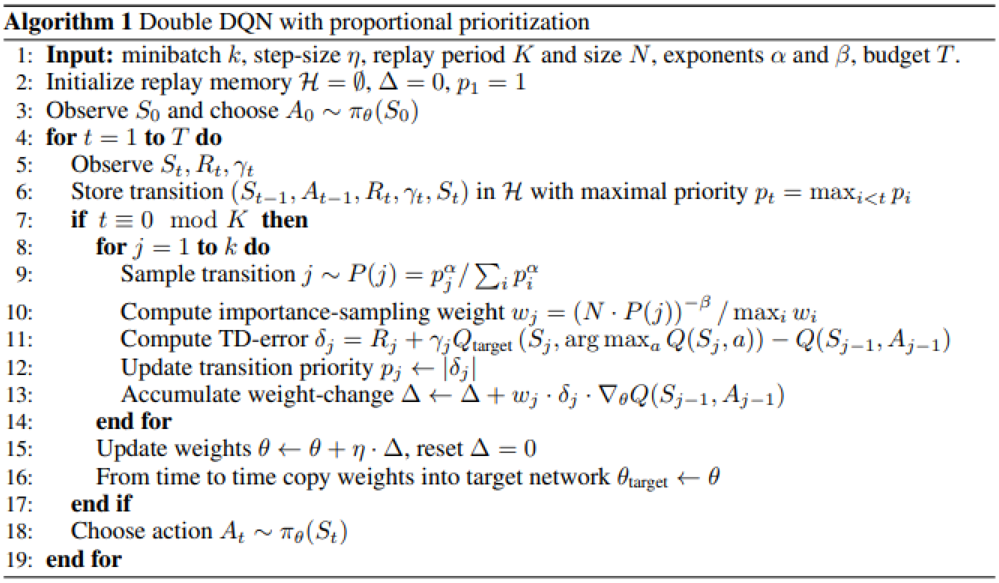
Motivation: An issue in RL is that transitions (s_t, a_t, r_t, s_{t+1}) (where s_t is the state at timestep t, a_t is the action taken at timestep t, and r_t is the reward observed at timestep t) are discarded after using them for training, even though witnessing the transition multiple times can be beneficial for learning. Experience replay addresses this issue somewhat, allowing an agent to sample past transitions from a replay buffer and train using said transitions. However, not every transition is equally as valuable. For example, some states are far more rarely encountered than others, and these states should intuitively be focused on more to ensure the agent is well prepared for all scenarios. Thus, a version of experience replay that prioritizes certain types of experiences is worth exploring.

Basic Idea: In Prioritized Experience Replay, rather than the replay buffer being sampled uniformly, certain transitions are prioritized over others. More concretely, the probability of sampling the jth transition is given by $P(j) = \dfrac{p_j^\alpha}{\sum_i p_i^\alpha}$, where $p_i > 0$ is the priority of transition i, and α is an exponent that weights how much prioritization should affect selection (so if $\alpha = 0$, this reduces to uniform selection).

There are many different possibilities for p_i, such as: $p_i = |\delta_i|+\epsilon$, where $\delta_i = r_i+\gamma Q(s_{i+1}, \underset{a}{\text{argmax}}\, Q(s_{i+1}, a))-Q(s_i, a_i)$ is the TD-error of the ith transition, and ϵ is a small positive constant. Another is $p_i = \dfrac{1}{rank(i)}$, where $rank(i)$ is the rank of transition i when sorted based on $|\delta_i|$. The latter is less sensitive to outliers.

However, this prioritization leads to a biased estimate of the experience being replayed. In order to fix this, importance-sampling based weights $w_i = w_i = (\frac{1}{N(P(i))})^{\beta}$ are used, where N is the maximum size of the replay buffer and $\beta \in (0, 1]$ is an exponent that weighs how much the weights should compensate (they will compensate fully if $\beta = 1$). These weights are also normalized to ensure the scaling of updates is always downwards.

Pseudocode (credit goes to Schaul et al): In this algorithm, k is the minibatch size for stochastic gradient descent, n is the step size for gradient descent, and K is the period to wait before accumulating the update.



Algorithm 1 Double DQN with proportional prioritization

```
Input: minibatch $k$, step-size $\eta$, replay period $K$ and size $N$, exponents $\alpha$ and $\beta$, budget $T$.
Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
Observe $S_0$ and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ to $T$ do
  Observe $S_t, R_t, \gamma_t$
  Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
  if $t \equiv 0 \mod K$ then
    for $j = 1$ to $k$ do
      Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
      Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
      Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
      Update transition priority $p_j \leftarrow |\delta_j|$
      Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
    end for
    Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
    From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
  end if
  Choose action $A_t \sim \pi_\theta(S_t)$
end for
```

Benefits: Prioritized experience replay allows practitioners to customize which transitions to prioritize, rather than simply sampling uniformly from the buffer. This can speed up convergence of RL agents.

Drawbacks: The bias introduced by prioritization must be carefully navigated, either through weighted importance sampling or another method. In addition, there is no guarantee that the metric used for prioritization is actually relevant to the given domain, so choosing the metric needs to be done carefully.

Experiments: Three agents were trained on various Atari 2600 games. They were all identical in terms of network architecture, with the only major difference being the algorithm used; one was a classic Deep Q Network agent, and two were Double Deep Q Network agents. The former and one of the latter used uniform replay sampling, while the other from the latter used prioritized experience replay. Not only did the prioritized experience replay agent converge faster than the other 2, but even had slightly improved scores in certain tasks. Some figures below visualize the effect prioritized experience replay had.

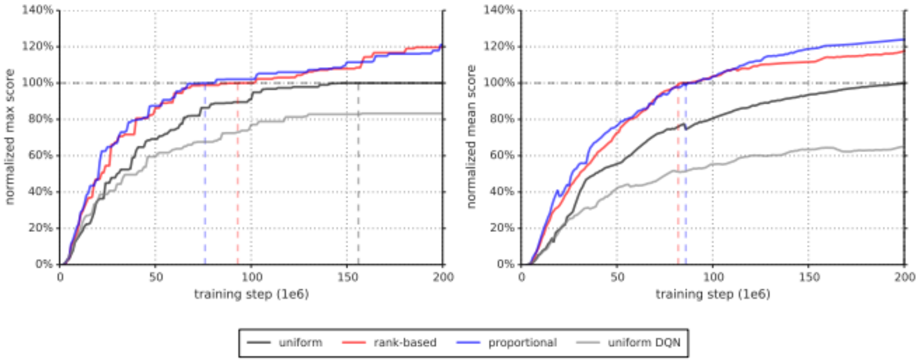
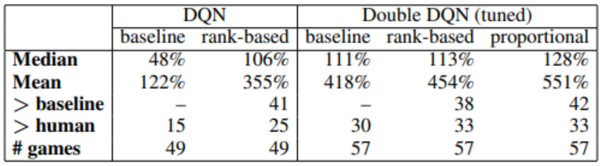


Figure 4: Summary plots of learning speed. **Left:** median over 57 games of the maximum baseline-normalized score achieved so far. The baseline-normalized score is calculated as in Equation 4 but using the maximum Double DQN score seen across training is used instead of the human score. The equivalence points are highlighted with dashed lines; those are the steps at which the curves reach 100%, (i.e., when the algorithm performs equivalently to Double DQN in terms of median over games). For rank-based and proportional prioritization these are at 47% and 38% of total training time. **Right:** Similar to the left, but using the mean instead of maximum, which captures cumulative performance rather than peak performance. Here rank-based and proportional prioritization reach the equivalence points at 41% and 43% of total training time, respectively. For the detailed learning curves that these plots summarize, see Figure 7.



	DQN		Double DQN (tuned)		
	baseline	rank-based	baseline	rank-based	proportional
Median	48%	106%	111%	113%	128%
Mean	122%	355%	418%	454%	551%
> baseline	–	41	–	38	42
> human	15	25	30	33	33
# games	49	49	57	57	57

Table 1: Summary of normalized scores. See Table 6 in the appendix for full results.

Links:

Original Paper

4.3.2 Asynchronous Methods for Deep Reinforcement Learning

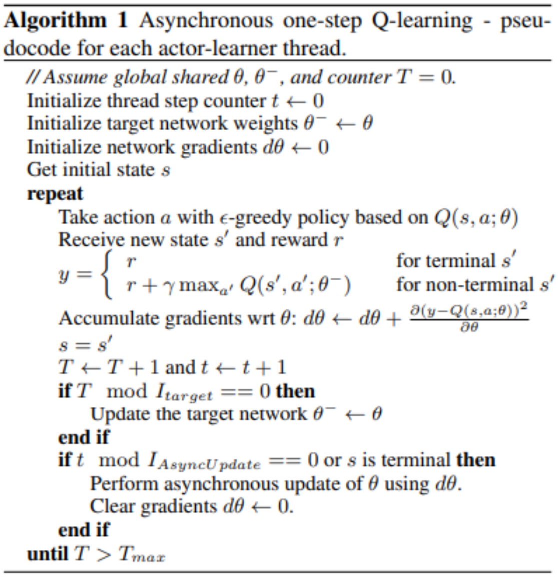
Prerequisites: an understanding of Q learning as well as Deep Q Networks is required. The reader also needs to understand actor-critic methods of reinforcement learning.

Motivation: Reinforcement learning algorithms are often trained such that a single environment run is happening synchronously, with GPUs being used to speed up compute times. This approach has a few drawbacks; one, it is very sample inefficient, as only one agent is interacting with one environment at any given time. Two, the samples obtained are often not very diverse, as the agent may begin to monotonously exploit ”optimal” actions given states. Three, GPU compute is expensive and energy intensive. A framework that exploits a multi-cored CPU, moving away from the GPU while allowing for more efficient and diverse sampling, would be helpful.

Basic Idea: This paper’s main focus is not a specific algorithm, but rather a framework for adapting existing RL algorithms into an asynchronous device. The framework involves using multiple CPU threads on a single machine, with each thread collecting data through a single run of the environment. This opens up a few possibilities for training; since each thread explores the environment independently, they can use different exploration strategies, and thus the framework enables a far broader spectrum of samples to be collected.

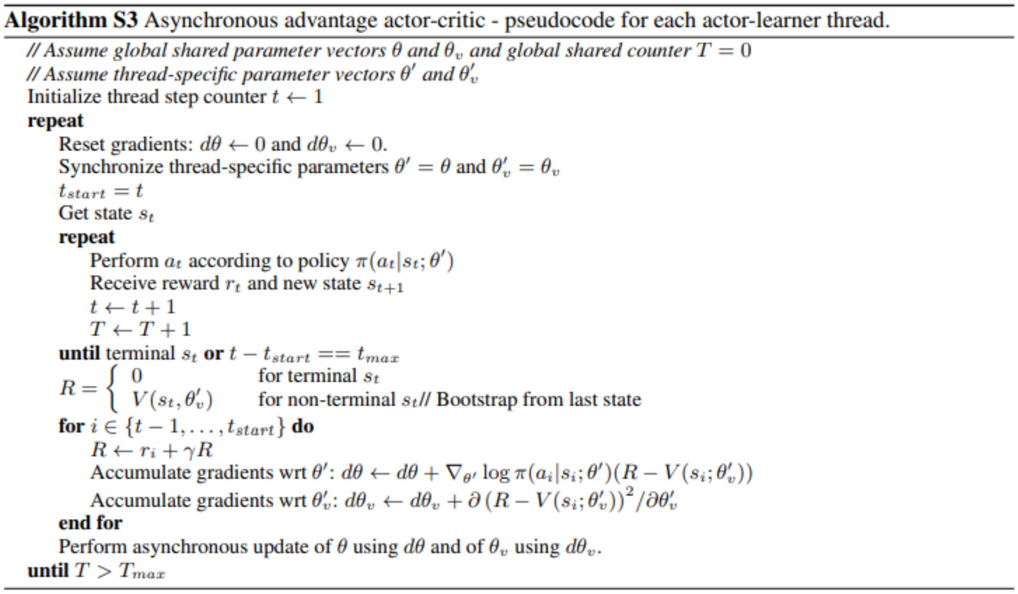
A consequence of this is that rather than relying on techniques like experience replay to stabilize learning, we can rely on the diverse exploration strategies across threads, enabling us to use this framework with on-policy algorithms like Sarsa and actor-critic methods.

Pseudocode: We present pseudocode for two algorithms in the framework; one for asynchronous one step Q learning, and another for asynchronous advantage actor critic, a new algorithm developed by the paper that surpassed the state of the art at the time.



Algorithm 1 Asynchronous one-step Q-learning - pseudocode for each actor-learner thread.

```
// Assume global shared $\theta$, $\theta^-$, and counter $T = 0$.
Initialize thread step counter $t \leftarrow 0$
Initialize target network weights $\theta^- \leftarrow \theta$
Initialize network gradients $d\theta \leftarrow 0$
Get initial state $s$
repeat
    Take action $a$ with $\epsilon$-greedy policy based on $Q(s, a; \theta)$
    Receive new state $s'$ and reward $r$
    $y = \begin{cases} r & \text{for terminal } s' \\ r + \gamma \max_{a'} Q(s', a'; \theta^-) & \text{for non-terminal } s' \end{cases}$
    Accumulate gradients wrt $\theta$: $d\theta \leftarrow d\theta + \frac{\partial (y - Q(s,a;\theta))^2}{\partial \theta}$
    $s = s'$
    $T \leftarrow T + 1$ and $t \leftarrow t + 1$
    if $T \mod I_{target} == 0$ then
        Update the target network $\theta^- \leftarrow \theta$
    end if
    if $t \mod I_{AsyncUpdate} == 0$ or $s$ is terminal then
        Perform asynchronous update of $\theta$ using $d\theta$.
        Clear gradients $d\theta \leftarrow 0$.
    end if
until $T > T_{max}$
```



Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

```
// Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared counter $T = 0$
// Assume thread-specific parameter vectors $\theta'$ and $\theta'_v$
Initialize thread step counter $t \leftarrow 1$
repeat
    Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
    Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
    $t_{start} = t$
    Get state $s_t$
    repeat
        Perform $a_t$ according to policy $\pi(a_t|s_t; \theta')$
        Receive reward $r_t$ and new state $s_{t+1}$
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    until terminal $s_t$ or $t - t_{start} == t_{max}$
    $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
    for $i \in \{t - 1, \ldots, t_{start}\}$ do
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients wrt $\theta'$: $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i; \theta')(R - V(s_i; \theta'_v))$
        Accumulate gradients wrt $\theta'_v$: $d\theta_v \leftarrow d\theta_v + \partial \left(R - V(s_i; \theta'_v)\right)^2 / \partial \theta'_v$
    end for
    Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$.
until $T > T_{max}$
```

Benefits: The benefits of this framework are numerous. For one, it enables greater sample efficiency, through multiple threads being able to run through an environment. In addition, each thread can have its own exploration policy (ϵ-greedy, Upper Confidence Bounds, or others), enabling the agent to experience the benefits of multiple strategies for exploration. Another benefit is that GPUs are not required to ensure great sample efficiency, as in practice multi-core CPUs are able to keep up. However, GPUs can still be included if desired. This framework is also adaptable to any existing reinforcement learning algorithm given the proper adjustments.

Drawbacks: Using too many CPU cores can cause other tasks to be slowed down, and so care needs to be taken when choosing the number of cores to use. As well, like with asynchronous programming in general, care needs to be taken to ensure threads do not overwrite each other's updates.

Experiments: Various different tasks were used in experiments: a set of Atari 2600 games, the TORCs simulator for racing cars, and various control tasks in the MuJoCo simulator. There were several different observations from these experiments:

The most important observation was that the asynchronous methods far outperformed their synchronous counterparts, and did so in less time and without the use of a GPU. This suggests that the improved sample efficiency and exploration abilities of a multithreaded approach far outweighed the compute benefits of a GPU. Below are some figures from the experiments; the first is a score comparison across five Atari games, and the second is a score comparison in the TORCs racing simulator:

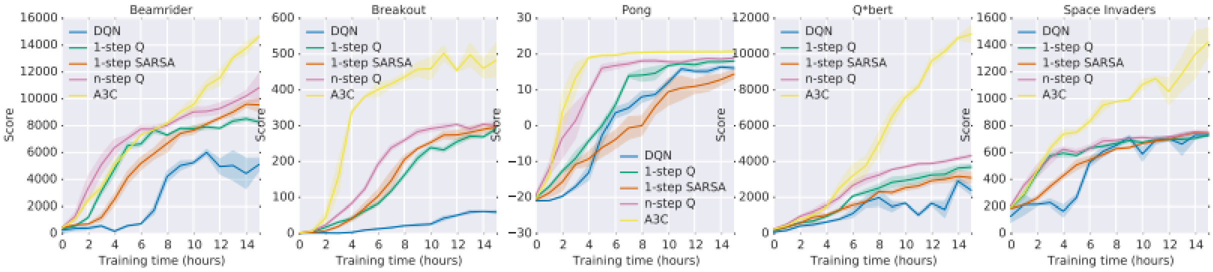


Figure 1. Learning speed comparison for DQN and the new asynchronous algorithms on five Atari 2600 games. DQN was trained on a single Nvidia K40 GPU while the asynchronous methods were trained using 16 CPU cores. The plots are averaged over 5 runs. In the case of DQN the runs were for different seeds with fixed hyperparameters. For asynchronous methods we average over the best 5 models from 50 experiments with learning rates sampled from $LogUniform(10^{-4}, 10^{-2})$ and all other hyperparameters fixed.

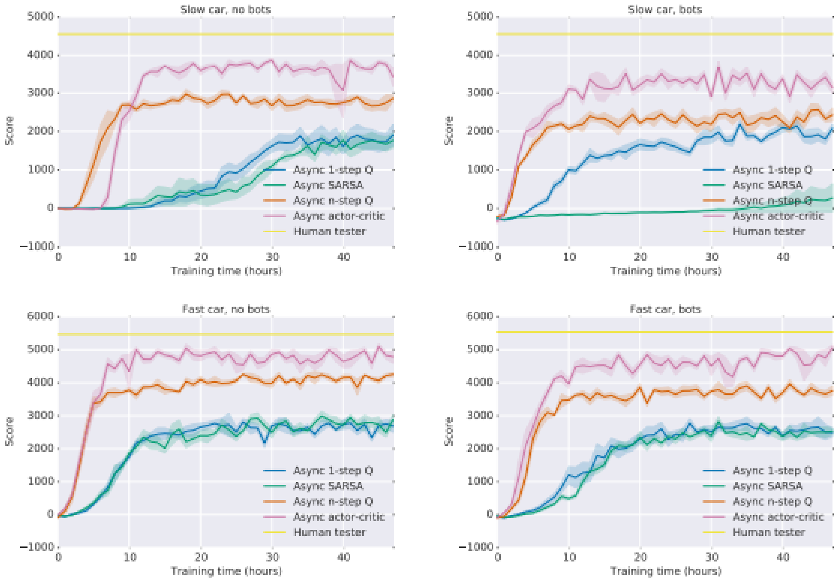


Figure S6. Comparison of algorithms on the TORCS car racing simulator. Four different configurations of car speed and opponent presence or absence are shown. In each plot, all four algorithms (one-step Q, one-step Sarsa, n-step Q and Advantage Actor-Critic) are compared on score vs training time in wall clock hours. Multi-step algorithms achieve better policies much faster than one-step algorithms on all four levels. The curves show averages over the 5 best runs from 50 experiments with learning rates sampled from $LogUniform(10^{-4}, 10^{-2})$ and all other hyperparameters fixed.

Another important finding was that the methods achieved robustness to different hyperparameter choices, and did not diverge or collapse due to bad hyperparameter selection. This showed the robustness of asynchronous methods to diverse hyperparameter choices. A plot that illustrates this is shown below (where the x axis is learning rate, and the y axis is game score):

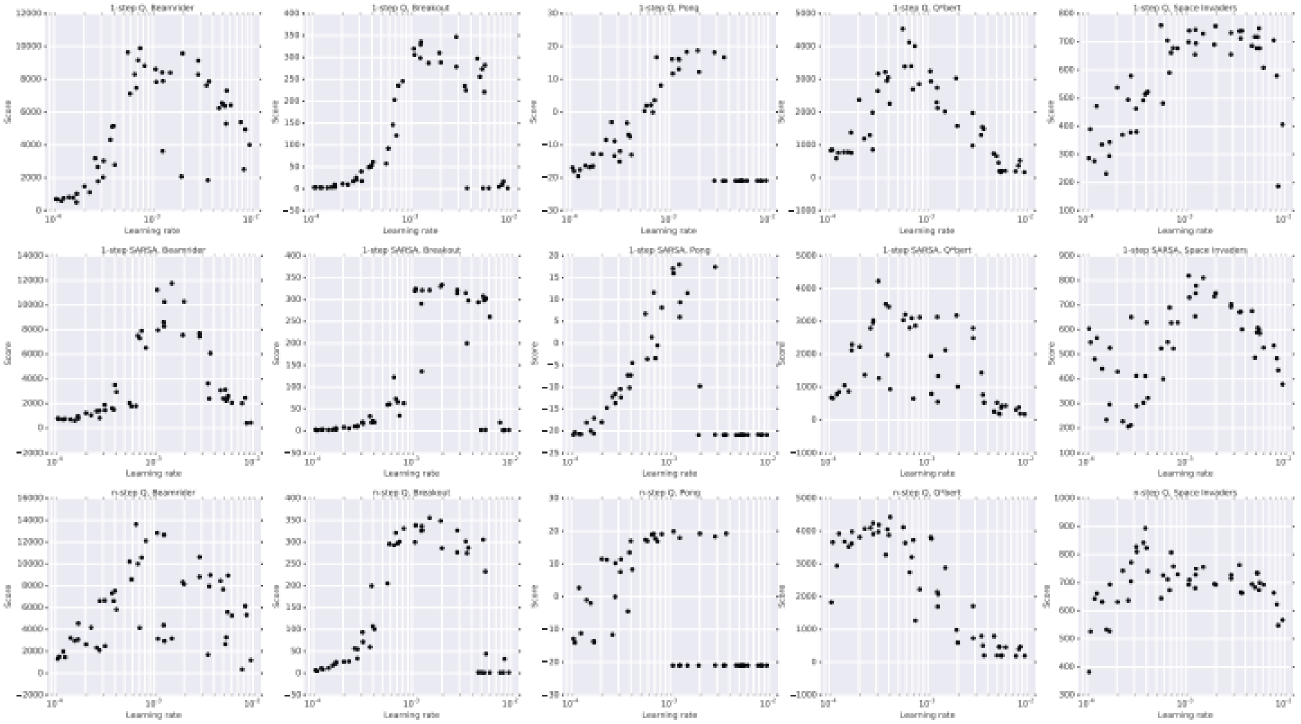


Figure S11. Scatter plots of scores obtained by one-step Q, one-step Sarsa, and n-step Q on five games (Beamrider, Breakout, Pong, Q*bert, Space Invaders) for 50 different learning rates and random initializations. All algorithms exhibit some level of robustness to the choice of learning rate.

Links:

Original Paper

4.4 Systems/Applications

4.4.1 Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm

Prerequisites: An understanding of Monte Carlo Tree Search, as well as an understanding of reinforcement learning and deep reinforcement learning.

Motivation: Many agents have been created that have mastered playing board games; however, most of these agents have used handcrafted heuristics to measure performance. At the time of this paper's publication, AlphaGo Zero had achieved superhuman performance without the use of these heuristics, and additionally, all through self play (when an agent plays against itself). An agent that can be generalized to learn a wide variety of games, and potentially other similar tasks, would be revolutionary.

Basic Idea: AlphaZero is a reinforcement learning algorithm designed to be used to learn board games using only the game rules as background knowledge and nothing else. The first component of AlphaZero is a neural network $f(s;\theta)$, where θ are the parameters of the network. This neural network takes the board state s as input, and outputs two things; a vector of probabilities $\mathbf{p}$, where $p_i = \mathbb{P}(a|s)$ is the probability for the ith action (note that this setup requires a finite number of available actions for each state), and a scalar $v_s \in \mathbb{R}$ that represents the expected game outcome from this state, i.e. whether the agent should expect to win, lose or draw.

The parameters θ are trained via self-play and reinforcement learning. Games are done by selecting moves for both players using Monte Carlo Tree Search, $a_t\pi_t$, where a_t and π_t are the action and Monte Carlo Tree Search probabilities at timestep t. When the tree search finishes a game, the outcome z is recorded: -1 for a loss, 1 for a win, and 0 for a draw. The network parameters θ are then updates so that the value v_s is closer to z, and that the probabilities $p_i = \mathbb{P}(a|s)$ are closer to the search probabilities π_t. Concretely, the parameters are updated to minimize the following loss: $L(\theta) = (z - v_{s_t})^2 - \pi_t^T \log(\mathbf{p}) + c\|\theta\|^2$, where c is a constant controlling weight regularization.

Benefits: AlphaZero is able to achieve superhuman performance in a diverse set of board games, without the use of handcrafted heuristics or move orderings. The description of the method allows various RL algorithms and methods to be intertwined with AlphaZero, such as IMPALA and A3C, making it highly versatile. In addition, the Monte Carlo Tree Search portion of AlphaZero makes it highly adaptable to differing strategies opponents may apply.

Drawbacks: As a game increases in complexity, it becomes harder to learn and usually results in higher dimensionality; especially since AlphaZero typically learns through self-play, it can take hours of training before an effective policy network f is trained.

Experiments: AlphaZero was placed against leading engines for various games. For Chess, it was pitted against Stockfish, one of the most powerful chess engines in the world at the time. For Shogi, it was pitted against Elmo, a popular Shogi engine. For Go, it was pitted against AlphaGo Zero, a previous Go agent trained through reinforcement learning and hand-crafted heuristics.

In all 3 cases, not only did AlphaZero win against its opponent far more times than it lost, but it was able to do so with less than half a day of training time in each game (more specifically, 2 hours in Chess, 4 in Shogi, and 8 in Go). Below are 2 figures that summarize the results of the experiments.

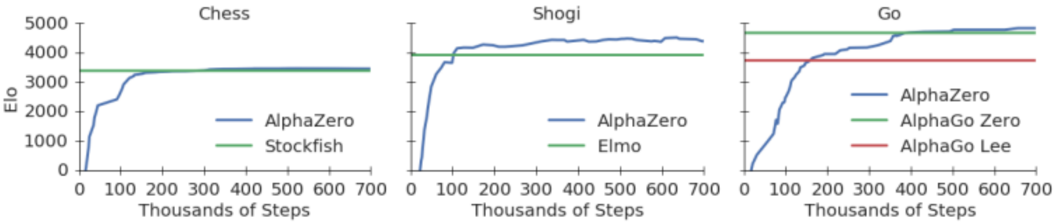
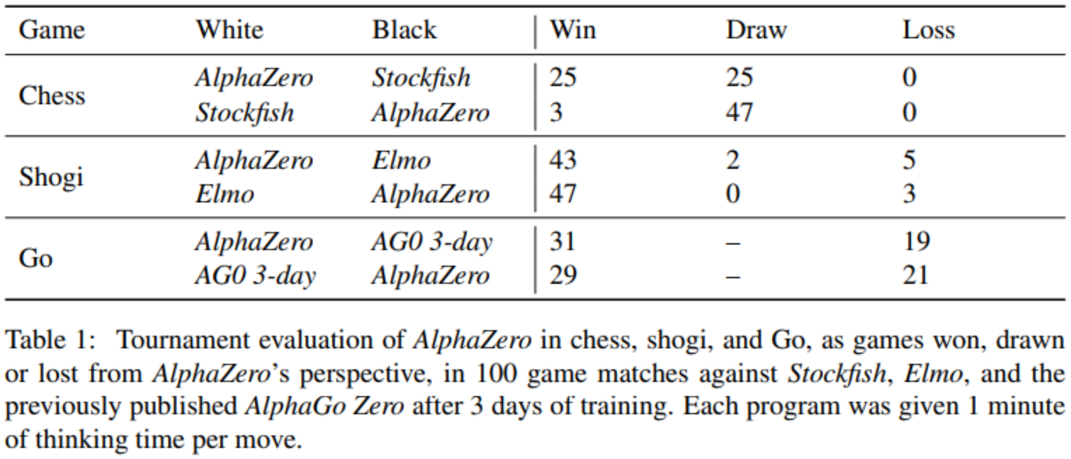


Figure 1: Training *AlphaZero* for 700,000 steps. Elo ratings were computed from evaluation games between different players when given one second per move. **a** Performance of *AlphaZero* in chess, compared to 2016 TCEC world-champion program *Stockfish*. **b** Performance of *AlphaZero* in shogi, compared to 2017 CSA world-champion program *Elmo*. **c** Performance of *AlphaZero* in Go, compared to *AlphaGo Lee* and *AlphaGo Zero* (20 block / 3 day) (*29*).



Game	White	Black	Win	Draw	Loss
Chess	*AlphaZero*	*Stockfish*	25	25	0
	Stockfish	*AlphaZero*	3	47	0
Shogi	*AlphaZero*	*Elmo*	43	2	5
	Elmo	*AlphaZero*	47	0	3
Go	*AlphaZero*	*AG0 3-day*	31	–	19
	AG0 3-day	*AlphaZero*	29	–	21

Table 1: Tournament evaluation of *AlphaZero* in chess, shogi, and Go, as games won, drawn or lost from *AlphaZero*'s perspective, in 100 game matches against *Stockfish*, *Elmo*, and the previously published *AlphaGo Zero* after 3 days of training. Each program was given 1 minute of thinking time per move.

Links:

Original Paper

4.5 Learning Theory

4.5.1 Policy Gradient Methods for Reinforcement Learning with Function Approximation

Prerequisites: An understanding of reinforcement learning, deep learning, and policy gradients is very important.

Main Theoretical Result(s): if the policy $\pi(a; s, \theta)$ and value function $f(s, a; \theta')$ are arbitrary function approximators (or equivalently, neural networks) parameterized by θ and θ' respectively, under certain constraints, the approximators through training eventually converge to the optimal policy and value function.

Links:

Original Paper

4.5.2 Policy invariance under reward transformations: theory and application to reward shaping

Prerequisites: An understanding of reinforcement learning is the most important here.

Main Theoretical Result(s): This paper introduces a type of reward shaping that doesn't change the optimal policy under a specific Markov Decision Process. The basic idea is as follows: suppose you have some reward shaping function $F\colon S \times A \times S \to \mathbb{R}$, where S, A is your set of states and set of actions respectively. F is said to be a **potential-based** shaping function if there is a function $\Phi\colon S \to \mathbb{R}$ such that $F(s, a, s') = \gamma\Phi(s') - \Phi(s)$. In practice, this means one can define Φ directly, and use $\gamma\Phi(s') - \Phi(s)$ as an additive reward to an existing Markov Decision Process, which enriches the reward space while ensuring the optimal policy is unaltered.

Links:

Original Paper