Large language models and transformers

B. Mehlig, Department of Physics, University of Gothenburg, Sweden

Sources

Linander, *The mathematics behind large language models* (2023)
Penke, *A mathematician's introduction to transformers and large language models* (2022)

Language models

Linander, *The mathematics behind large language models* (2023)

Language model: predicts next word in a sentence.

$$\text{The quick brown fox jumps over the lazy dog.}$$ wikipedia.org

$$x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad x_7 \quad x_8 \quad x_9$$

A language model parameterises the probability $p(x_{t+1}|x_t, \ldots, x_1)$. Parameters are determined by training. For example, for $x_1, x_2, x_3 = \text{the}\,, \text{quick}\,, \text{brown}\,,$

$$p(x_4 = \text{fox}) \gg p(x_4 = \text{butter})$$

Assign probability to sentence:

$$p(x_T, \ldots, x_1) = \prod_t^T p(x_t|x_{t-1}, \ldots, x_1)$$

Encoding

Linander, *The mathematics behind large language models* (2023)

For the machine to interpret text, need to map vocabulary to numbers.

Lipton et al. (2015) arxiv:1506.00019

One possibility is to map the words (*tokens*) in the dictionary to integers (*token ID*) as follows

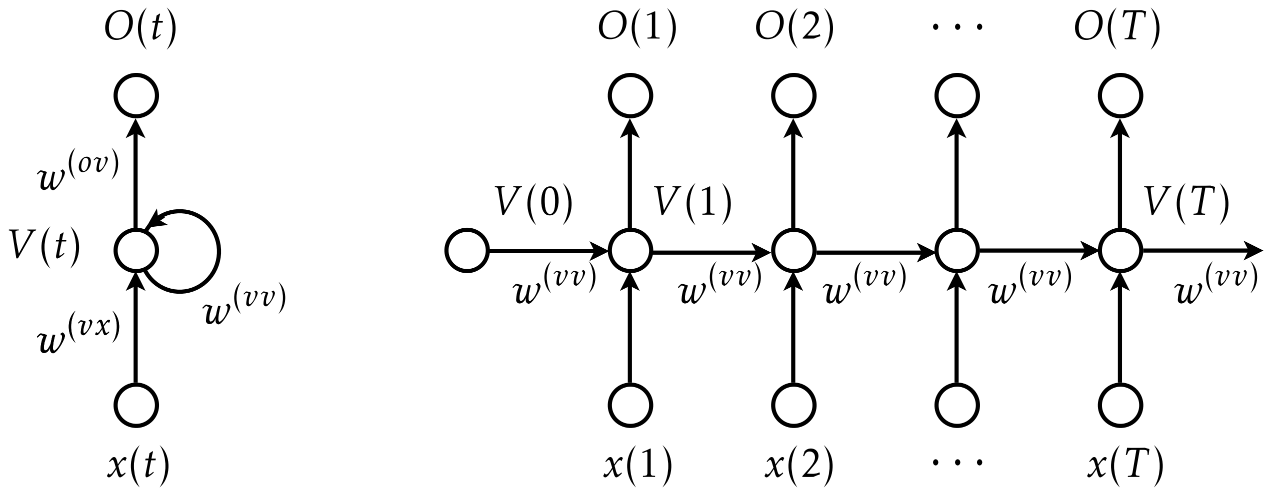
$$\mathrm{a} = 1\,, \mathrm{aardvark} = 2\,, \ldots$$

In a second step, one embeds the words to row vectors. Schematically

word	a	$\mathrm{aardvark}$	$\cdots$
token ID	1	2	$\cdots$
embedding	$[0.3, 0.8, 0.1]$	$[0.9, 0.2, 0.4]$	$\cdots$

Training recurrent networks

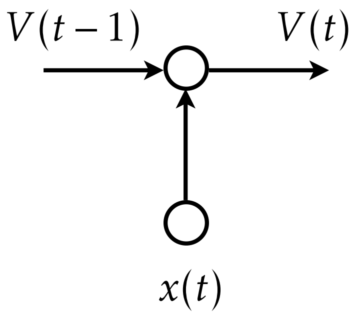
Unfold the recurrent network for machine translation.
Train it using data set of translated sentences by backpropagation through time.

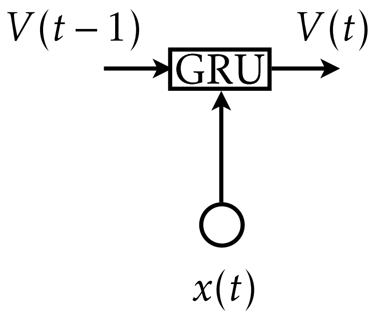


Vanishing-gradient problem

Hochreiter & Schmidhuber, Neural Computation **9** (1997)

Create short cuts by replacing hidden neurons with units that can either map input non-linearly or short cut it, $V(t) = V(t-1)$, depending on trainable parameters.





$$z_m(t) = \sigma\Big(\sum_k w^{(zx)}_{mk} x_k(t) + \sum_j w^{(zv)}_{mj} V_j(t-1)\Big)\,,$$

$$r_n(t) = \sigma\Big(\sum_k w^{(rx)}_{nk} x_k(t) + \sum_j w^{(rv)}_{nj} V_j(t-1)\Big)\,,$$

$$h_i(t) = g\Big(\sum_k w^{(hx)}_{ik} x_k(t) + \sum_j w^{(hv)}_{ij} r_j(t) V_j(t-1)\Big)\,,$$

$$V_i(t) = [1 - z_i(t)] h_i(t) + z_i(t) V_i(t-1)\,.$$

Different versions: LSTM or GRU.

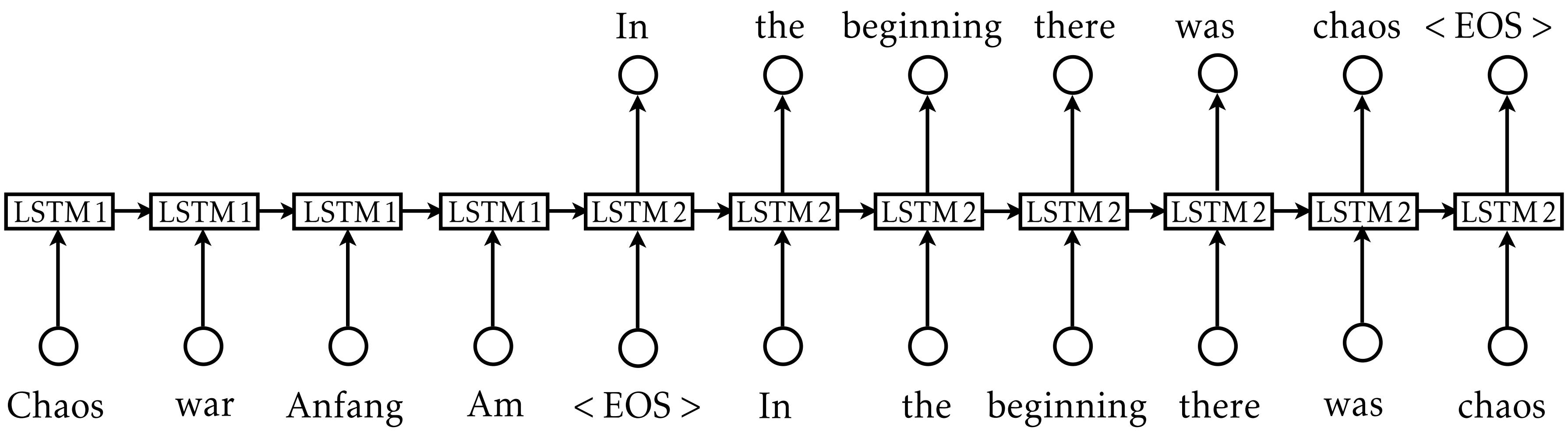
The unit short cuts when
$z_m = 1$.

Additional parameters (weights and thresholds are trained in the usual fashion).

Machine translation

Sutskever *et al.*, NIPS **27** (2014)

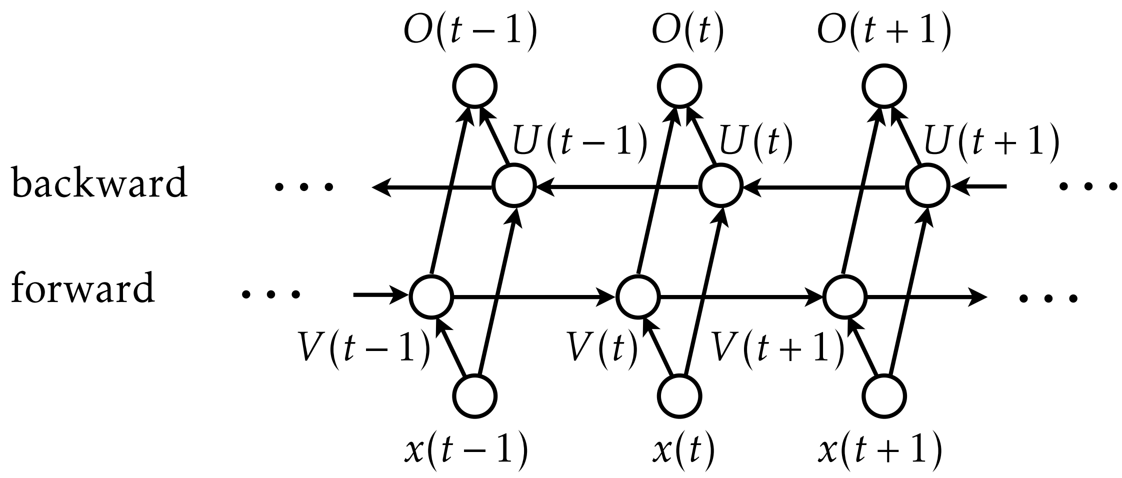
Recurrent network for machine translation



Bi-directional recurrent neural nets

Lipton et al. (2015) arxiv:1506.00019

Improved algorithm for machine translation uses bi-directional recurrent neural network.



Google translate.

Wu et al., Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation, arxiv1609.08144

Dependencies

Vaswani *et al.* (2017) arxiv:1706.03762

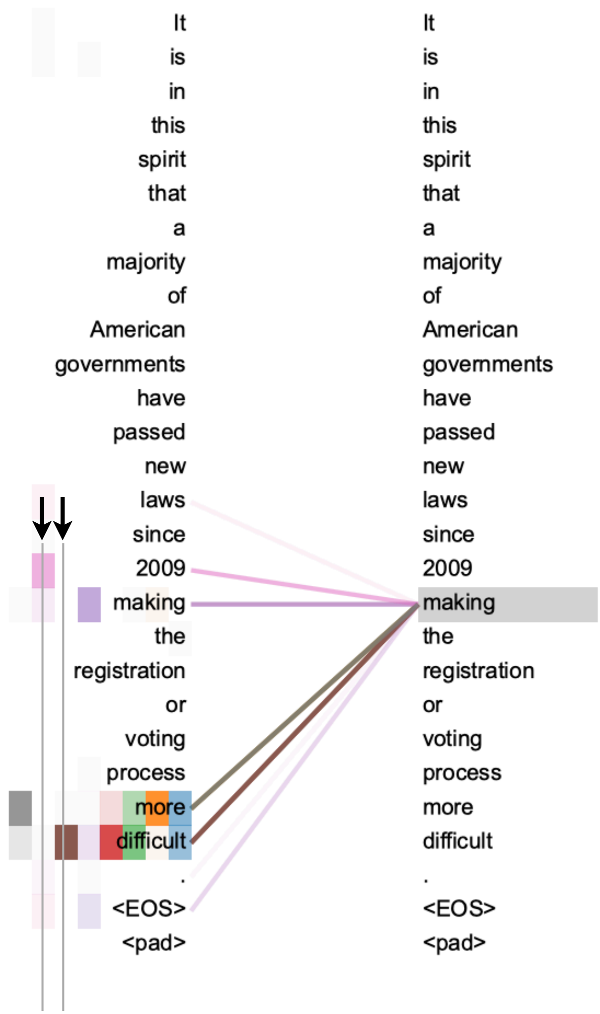
Attention mechanism designed to Represent long-range correlations.

The colours represent long-range dependencies (correlations) that the machine has learnt. Different columns (↓) correspond to different *attention heads*.

For instance, the verb making depends strongly on more and $\mathrm{difficult}$, reflecting the likely sequence

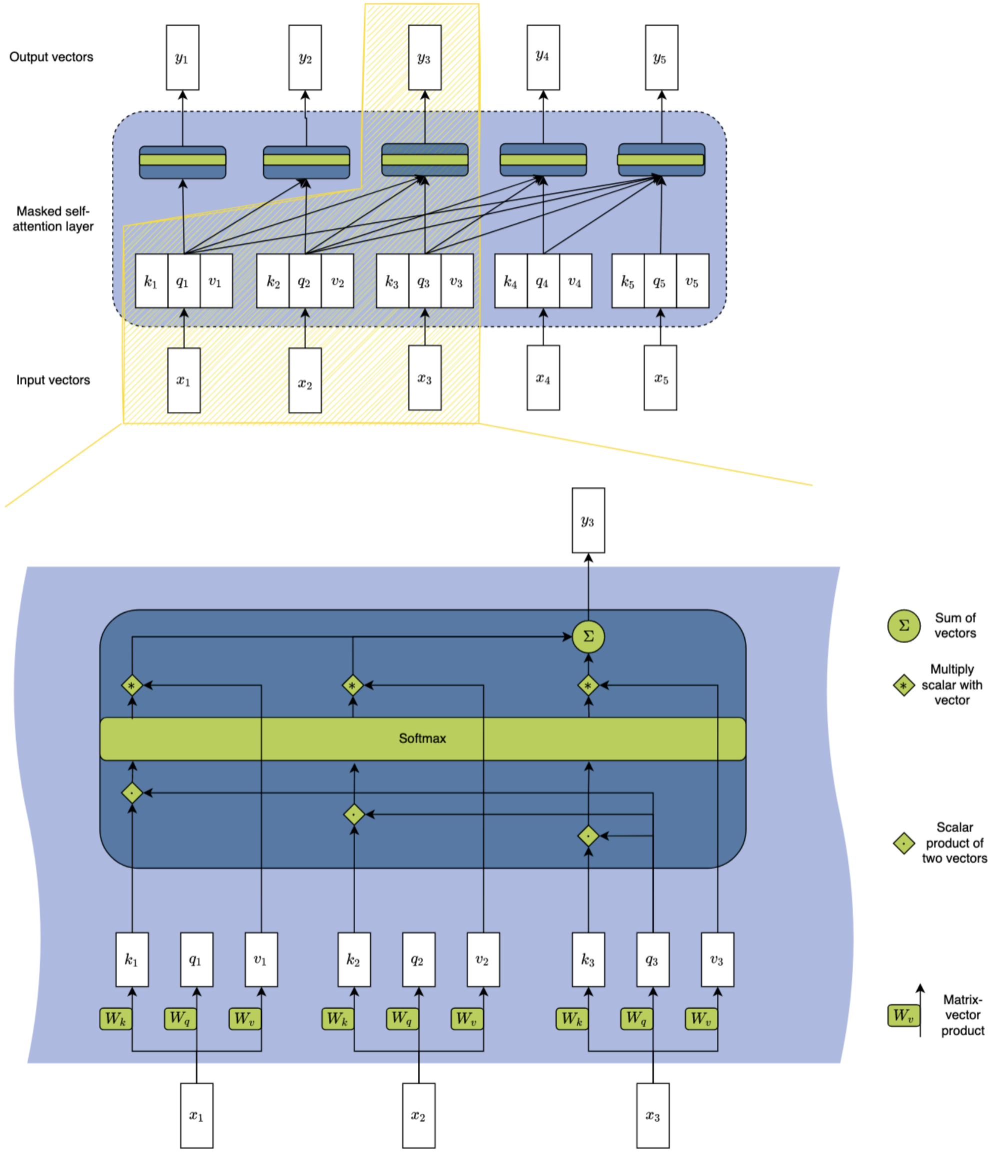
$$\mathrm{making}\ \ldots\ \mathrm{more}\ \ \mathrm{difficult}$$

How does it work?



Attention

Penke, *A mathematician's introduction to transformers and large language models* (2022)
Durafsky & Martin, *Speech and language processing* (2021)



An *attention layer* maps a sequence $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$ to an output sequence $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_T$ of the same length.

$$\boldsymbol{q}_t = \boldsymbol{x}_t \mathbb{W}^{(q)}$$

$$\boldsymbol{k}_t = \boldsymbol{x}_t \mathbb{W}^{(k)}$$

$$\boldsymbol{v}_t = \boldsymbol{x}_t \mathbb{W}^{(v)}$$

$$\alpha_{t,\tau} = \mathrm{softmax}(\mathcal{N} \boldsymbol{q}_t \cdot \boldsymbol{k}_\tau)$$

$$\boldsymbol{y}_t = \sum_\tau \alpha_{t,\tau} \boldsymbol{v}_\tau$$

Attention

Penke, *A mathematician's introduction to transformers and large language models* (2022)

The attention layer can process all words in the sequence in parallel

$$\mathbb{X} = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_T \end{bmatrix} \qquad \mathbb{Q} = \begin{bmatrix} \boldsymbol{q}_1 \\ \boldsymbol{q}_2 \\ \vdots \\ \boldsymbol{q}_T \end{bmatrix} \qquad \mathbb{K} = \begin{bmatrix} \boldsymbol{k}_1 \\ \boldsymbol{k}_2 \\ \vdots \\ \boldsymbol{k}_T \end{bmatrix} \qquad \mathbb{V} = \begin{bmatrix} \boldsymbol{v}_1 \\ \boldsymbol{v}_2 \\ \vdots \\ \boldsymbol{v}_T \end{bmatrix}$$

using the operations

$$\mathbb{Q} = \mathbb{X}\mathbb{W}^{(q)} \qquad \mathbb{K} = \mathbb{X}\mathbb{W}^{(k)} \qquad \mathbb{Y} = \mathrm{softmax}(\mathcal{N}\mathbb{Q}\mathbb{K}^{\mathsf{T}})\mathbb{V}$$

($\mathrm{softmax}$ is applied row-wise).

The matrix $\mathbb{Y}$ has the same dimension as $\mathbb{X}$.

The weight matrices $\mathbb{W}^{(q)}$, $\mathbb{W}^{(v)}$, and $\mathbb{W}^{(k)}$ are trained by backpropagation.

Attention heads

Penke, *A mathematician's introduction to transformers and large language models* (2022)

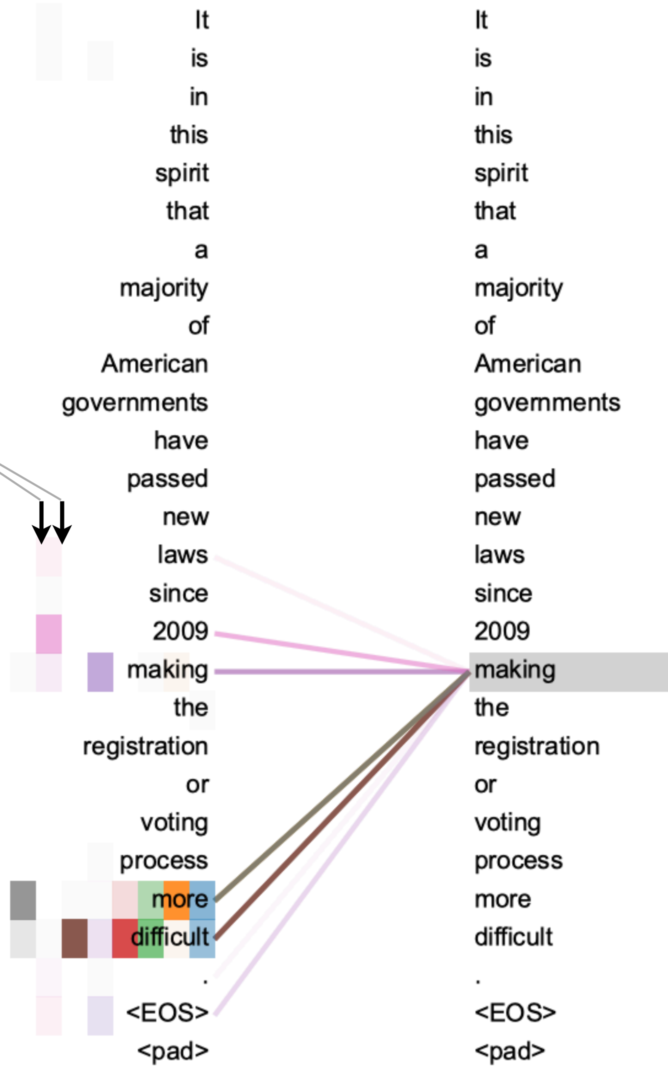
Usually one has several *attention layers* (*attention heads*) in parallel, with independent weights.

Concatenate outputs $\mathbb{Y}^{(i)}$of the different layers $[\mathbb{Y}^{(1)}, \mathbb{Y}^{(2)}, \ldots]$ and map to original dimension with another weight matrix $\mathbb{W}^{(o)}$

$$\mathbb{Y} = [\mathbb{Y}^{(1)}, \mathbb{Y}^{(2)}, \ldots]\mathbb{W}^{(o)}$$

.

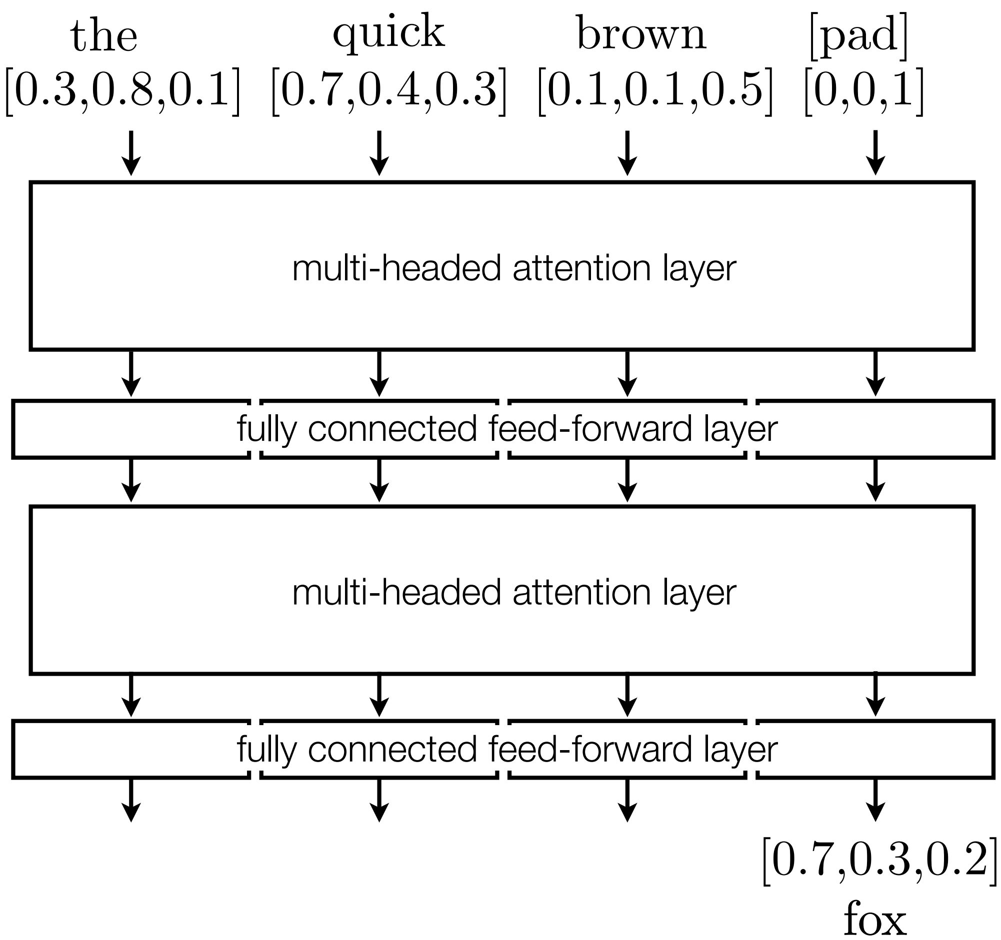
different heads



Transformers

Linander, *The mathematics behind large language models* (2023)

Transformers consist of multi-headed attention layers, fully-connected layers, and residual connections.



Training

Penke, *A mathematician's introduction to transformers and large language models* (2022)

Transformers are trained in the usual way by gradient descent, using automatic differentiation (TensorFlow, PyTorch).

Loss function based on log likelihood.

Pre-training. Language model is trained on a generic data set (e.g., WikiText) to learn general features.

www.tensorflow.org/datasets

Embedding is part of pre-training.

Fine tuning. Pre-trained model is trained further on data set of interest.

This two-step procedure is more efficient and less prone to overfitting, compared with training from scratch for a specific data set.

Inference. Using the trained large language model, user input is converted to output (ChatGPT).

Transformer architectures

	encoder-decoder for machine translation Vaswani *et al.* (2017) arxiv:1706.03762	(2017)
GPT	generative pre-trained transformer Radford *et al.* (2018) *Improving understanding by generative pre-training*	(2018)
BERT	bidirectional encoder representations from transformers Devlin *et al.* (2018) arxiv:1810.04805	(2018)
GPT-2	trained on larger data sets Radford *et al.* (2019)	(2019)
GPT-3	even larger data set, 10^{11} training parameters, improved training, with *few-shot learning* Brown *et al.* (2020)	(2020)
OpenGPT-X	open-source European large language model opengpt-x.de	(2022)

Few-shot learning

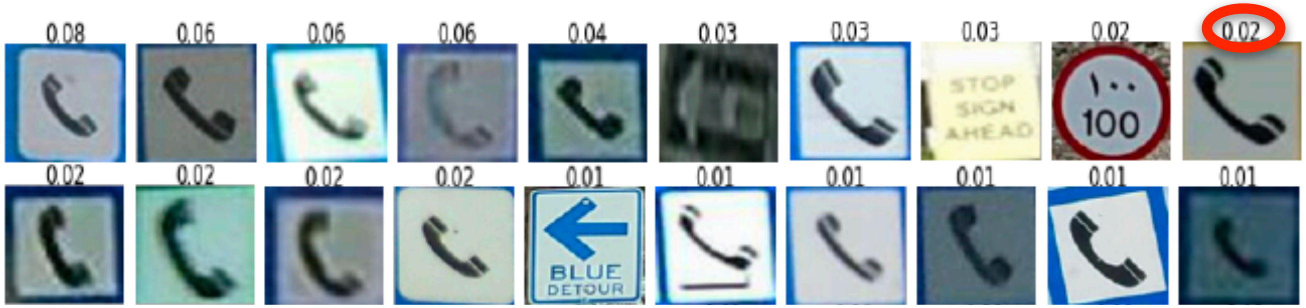
Jaghouar, Gustafsson, Mehlig, Werner & Gustavsson, DAGM GCPR Pattern Recognition (2022)

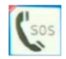
Usually the network needs to see an image many times to recognise it reliably.

Can networks learn to recognise rare patterns? Example: rare traffic signs.

Training set contains only few of the rare sign .

Let network try to find this sign in large test set. Result:



Network output in channel  (between 0 and 1). Outputs sum to unity over all channels.

Ordered according to output in channel  . All outputs small. But the largest of these are still meaningful. See above paper for references to *few-shot learning*.

Training data sets

Linander, *The mathematics behind large language models* (2023)
Touvron *et al.* (2023), arxiv:2302.13971

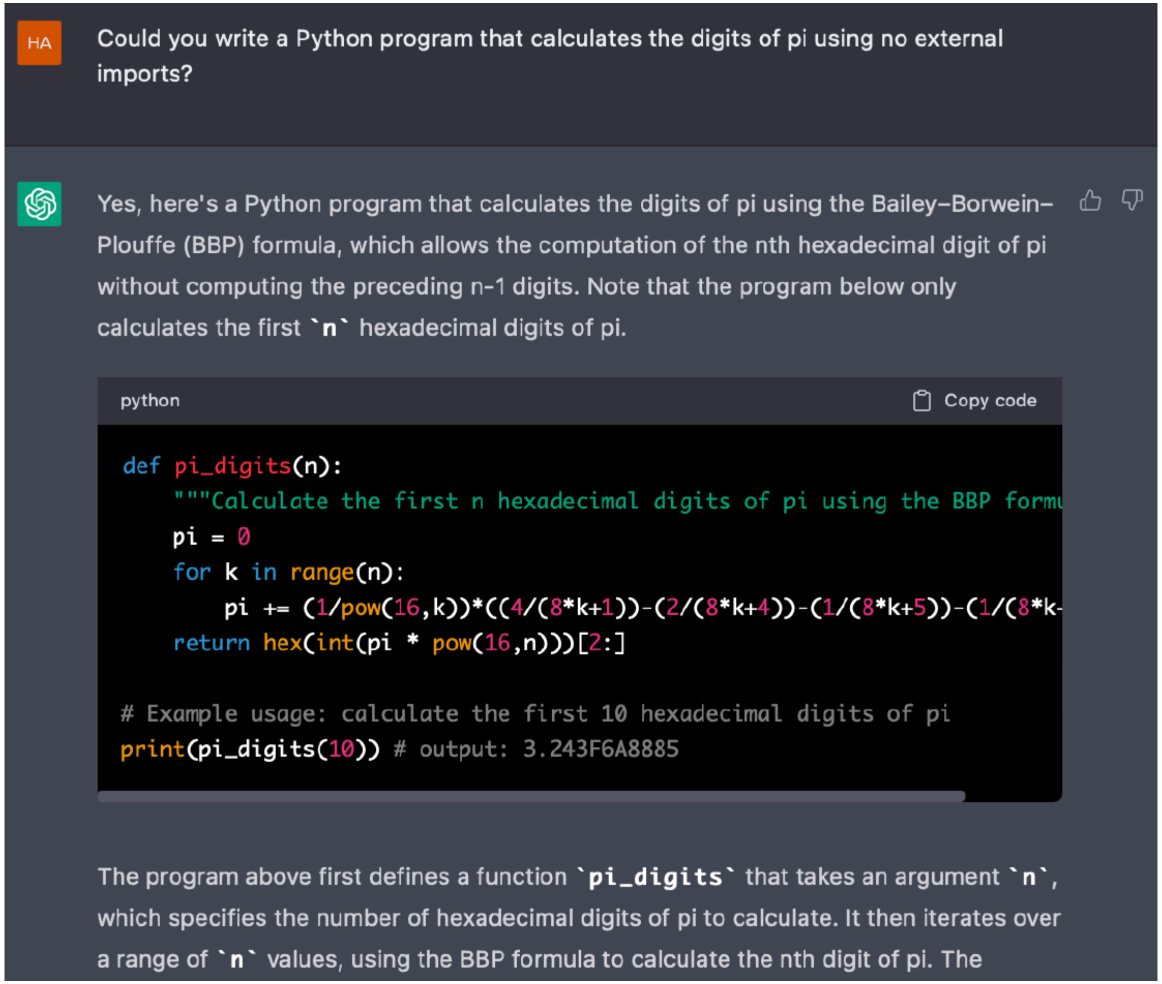
Dataset	Sampling prop.	Epochs	Disk size
CommonCrawl	67.0%	1.10	3.3 TB
C4	15.0%	1.06	783 GB
Github	4.5%	0.64	328 GB
Wikipedia	4.5%	2.45	83 GB
Books	4.5%	2.23	85 GB
ArXiv	2.5%	1.06	92 GB
StackExchange	2.0%	1.03	78 GB

In total $\sim 10^{12}$ tokens. Touvron *et al.* (2023), arxiv:2302.13971

ChatGPT

Linander, *The mathematics behind large language models* (2023)

ChatGPT is based on GPT-3. Additional fine tuning using *reinforcement learning* with Human feedback.



Hampus Linander (2023)

Reinforcement learning

Sutton & Barto, *Reinforcement Learning: An Introduction*, MIT Press (2018)
Mehlig, *Machine learning with neural networks*, CUP (2021)

Supervised learning requires labelled data (targets $\boldsymbol{t}^{(\mu)}$)
Unsupervised learning does not need labels.

Reinforcement learning: only partial feedback in terms of a reward function, e.g.

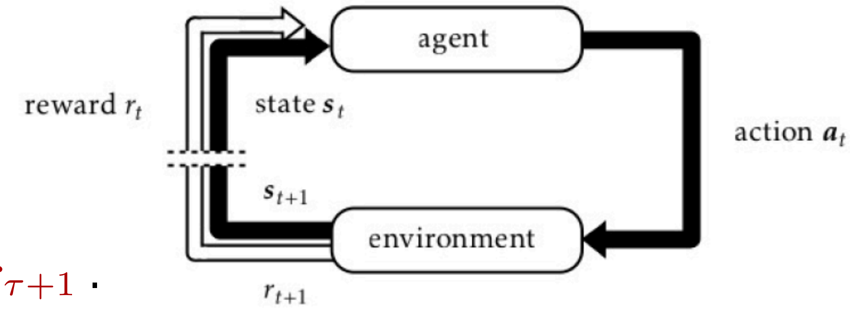
$$r = \begin{cases} +1 & \text{reward if all outputs correct} \\ -1 & \text{penalty otherwise} \end{cases}$$

(*immediate* reward)

Learning by trial and error.

Sequential decision process. Estimate expected *future* reward.

Agent explores a sequence of states $\boldsymbol{s}_0, \boldsymbol{s}_1, \boldsymbol{s}_2, \ldots$
through a sequence of actions $\boldsymbol{a}_0, \boldsymbol{a}_1, \boldsymbol{a}_2, \ldots$ and
receives rewards $r_1, r_2, r_3, \ldots$.
Goal: estimate expected *future* reward $R_t = \sum_{\tau=t}^{T-1} r_{\tau+1}$.

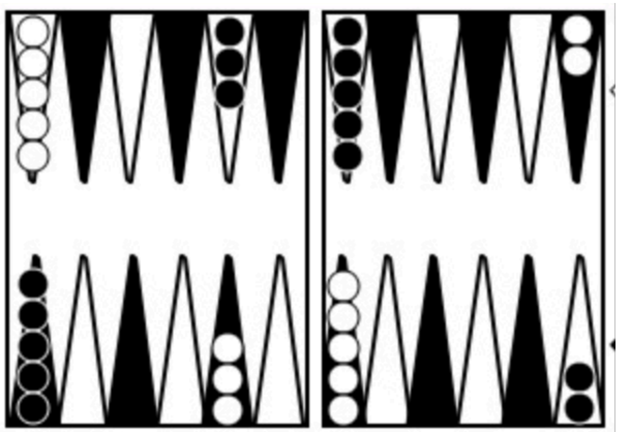


Method: iteratively improve estimate of expected future reward, given state $\boldsymbol{s}_t$ and action $\boldsymbol{a}_t$.

Backgammon

Tesauro, NIPS (1991)

Reinforcement learning allows computers to learn to play board games.



Program	Training Games	Opponents	Results
TDG 1.0	300,000	Robertie, Davis, Magriel	-13 pts/51 games (-0.25 ppg)
TDG 2.0	800,000	Goulding, Woolsey, Snellings, Russell, Sylvester	-7 pts/38 games (-0.18 ppg)
TDG 2.1	1,500,000	Robertie	-1 pt/40 games (-0.02 ppg)

Table 1. Results of testing TD-Gammon in play against world-class human opponents. Version 1.0 used 1-play search for move selection; versions 2.0 and 2.1 used 2-ply search. Version 2.0 had 40 hidden units; versions 1.0 and 2.1 had 80 hidden units.

Tesauro, Communications of the ACM (1995)

Practical Issues in Temporal Difference Learning

Gerald Tesauro
IBM Thomas J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598
tesauro@watson.ibm.com

Abstract

This paper examines whether temporal difference methods for training connectionist networks, such as Suttons's TD(λ) algorithm, can be successfully applied to complex real-world problems. A number of important practical issues are identified and discussed from a general theoretical perspective. These practical issues are then examined in the context of a case study in which TD(λ) is applied to learning the game of backgammon from the outcome of self-play. This is apparently the first application of this algorithm to a complex nontrivial task. It is found that, with zero

AlphaGo

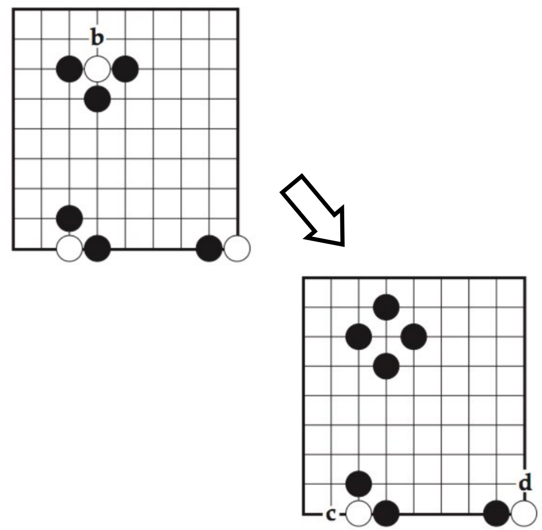
Silver *et al.* Nature (2016,2017)

Reinforcement learning allows computers to learn to play board games.

Agents: two players,
Environment: the opponent,
States: board configurations,
Actions: moves,
Future reward: $r = +1$ (win),
$r = -1$ (lose).

In AlphaGo's case, that involved splitting itself in half and playing millions of matches against itself, learning from each victory and loss. In one day alone, AlphaGo was able to play itself more than a million times, gaining more practical experience than a human player could hope to gain in a lifetime. In essence, AlphaGo got better at Go simply by thinking extremely hard about the problem.

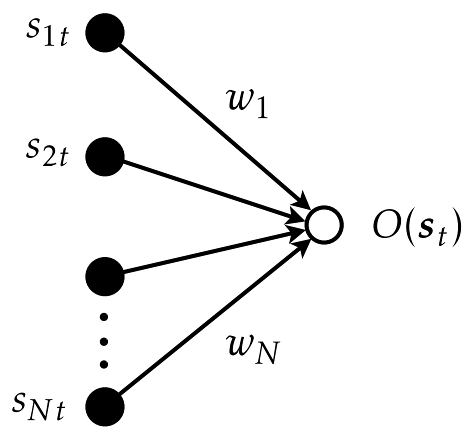
Alex Hern, in: The Guardian (2016)



British GO association

Temporal difference learning

Future reward $R_t = \sum_{\tau=t}^{T-1} r_{\tau+1}$.



Use neural network with input $\boldsymbol{s}_t$ (state) to estimate R_t ,

$$O(\boldsymbol{s}_t) = \boldsymbol{w} \cdot \boldsymbol{s}_t$$ (linear unit, weight vector $\boldsymbol{w}$)

Minimise energy function $H = \frac{1}{2}\sum_{t=0}^{T-1}[R_t - O(\boldsymbol{s}_t)]^2$ using gradient descent:

$$\delta w_m = \eta \sum_{t=0}^{T-1}[R_t - O(\boldsymbol{s}_t)]\,\frac{\partial O}{\partial w_m}$$

Trick: express error $R_t - O(\boldsymbol{s}_t)$ as sum over temporal differences

$$R_t - O(\boldsymbol{s}_t) = \sum_{\tau=t}^{T-1}[r_{\tau+1} + O(\boldsymbol{s}_{\tau+1}) - O(\boldsymbol{s}_\tau)]$$ with $O(\boldsymbol{s}_T) \equiv 0$

Temporal difference learning

Insert this expression for $R_t - O(\boldsymbol{s}_t)$ into the gradient-descent rule:

$$\delta \boldsymbol{w} = \eta \sum_{t=0}^{T-1} \sum_{\tau=t}^{T-1} [r_{\tau+1} + O(\boldsymbol{s}_{\tau+1}) - O(\boldsymbol{s}_\tau)] \boldsymbol{s}_t$$

Terms in this double sum can be summed in a different way

$$\delta \boldsymbol{w} = \eta \sum_{\tau=0}^{T-1} \sum_{t=0}^{\tau} [r_{\tau+1} + O(\boldsymbol{s}_{\tau+1}) - O(\boldsymbol{s}_\tau)] \boldsymbol{s}_t$$

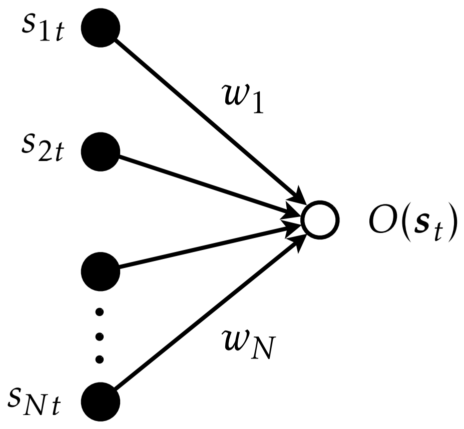
Exchange summation variables and add weights λ:

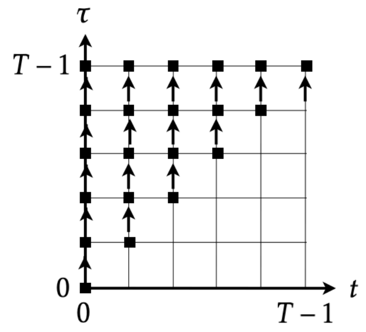
$$\delta \boldsymbol{w} = \eta \sum_{t=0}^{T-1} [r_{t+1} + O(\boldsymbol{s}_{t+1}) - O(\boldsymbol{s}_t)] \sum_{\tau=0}^{t} \lambda^{t-\tau} \boldsymbol{s}_\tau$$

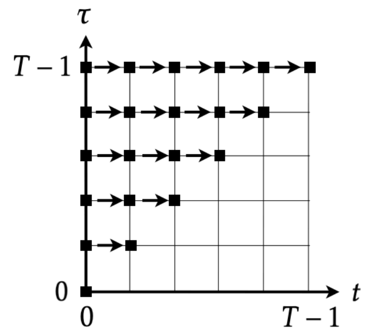
Alternative: update $\boldsymbol{w}$ (and hence O) at every time step:

$$\delta \boldsymbol{w}_t = \eta [r_{t+1} + O(\boldsymbol{w}_t, \boldsymbol{s}_{t+1}) - O(\boldsymbol{w}_t, \boldsymbol{s}_t)] \sum_{\tau=0}^{t} \lambda^{t-\tau} \boldsymbol{s}_\tau$$

This is the temporal difference learning rule TD(λ).







SARSA

Temporal difference learning TD(λ)

$$\delta \boldsymbol{w}_t = \eta[r_{t+1} + O(\boldsymbol{w}_t,\boldsymbol{s}_{t+1}) - O(\boldsymbol{w}_t,\boldsymbol{s}_t)]\sum_{\tau=0}^{t}\lambda^{t-\tau}\boldsymbol{s}_\tau$$

The TD(0)-rule corresponds to the following learning rule for the network output

$$O_{t+1}(\boldsymbol{s}_t) = O_t(\boldsymbol{s}_t) + \eta[r_{t+1} + O_t(\boldsymbol{s}_{t+1}) - O_t(\boldsymbol{s}_t)]$$

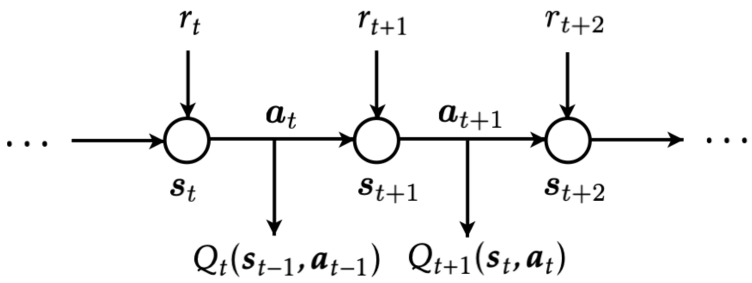
For a sequential decision process, estimate the expected future reward given $\boldsymbol{s}_t$ and $\boldsymbol{a}_t$

$$Q_{t+1}(\boldsymbol{s}_t,\boldsymbol{a}_t) = Q_t(\boldsymbol{s}_t,\boldsymbol{a}_t) + \eta\big[r_{t+1} + Q_t(\boldsymbol{s}_{t+1},\boldsymbol{a}_{t+1}) - Q_t(\boldsymbol{s}_t,\boldsymbol{a}_t)\big]$$

To update one needs $\boldsymbol{s}_t$, $\boldsymbol{a}_t$, r_{t+1} , $\boldsymbol{s}_{t+1}$, $\boldsymbol{a}_{t+1}$ (SARSA).

Problem: iteration depends upon policy for how to choose the next action, $\boldsymbol{a}_{t+1}$.
Greedy policy: choose the action one with largest $Q_t(\boldsymbol{s}_t,\boldsymbol{a}_t)$.
Stochastic policy: mainly greedy, but sometimes do something else.

Explore-versus-exploit dilemma.



Q-learning

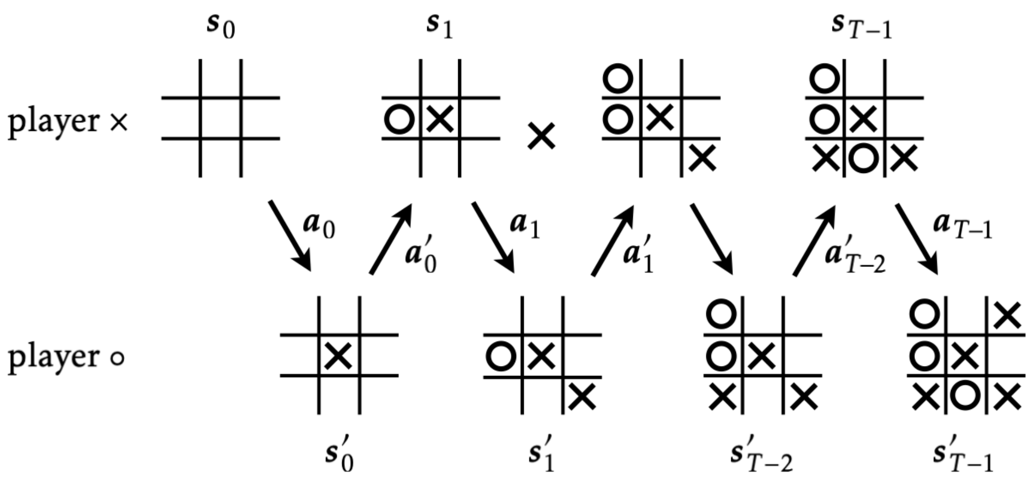
The Q-learning rule is an approximation to SARSA that does not depend on $\boldsymbol{a}_{t+1}$.

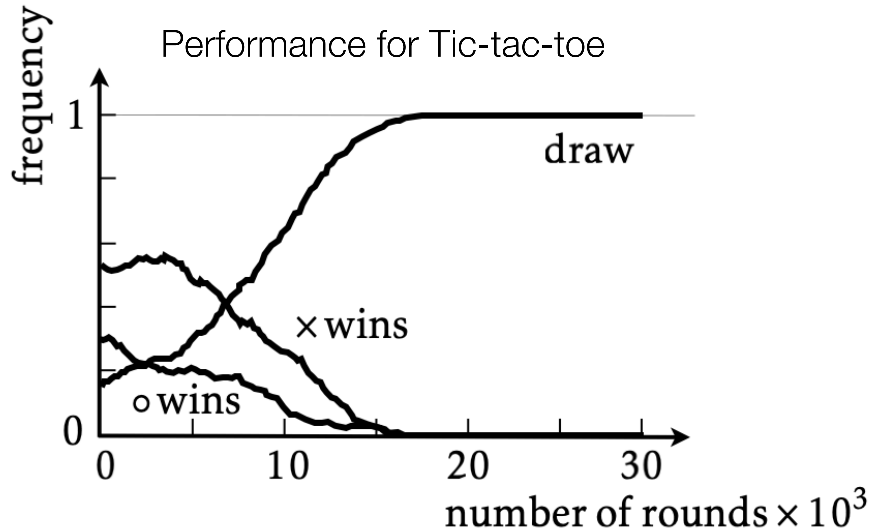
Instead one assumes that the next action, $\boldsymbol{a}_{t+1}$, is the optimal one, regardless of the policy that is currently followed:

$$Q_{t+1}(\boldsymbol{s}_t, \boldsymbol{a}_t) = Q_t(\boldsymbol{s}_t, \boldsymbol{a}_t) + \eta\big[r_{t+1} + \max_{\boldsymbol{a}} Q_t(\boldsymbol{s}_{t+1}, \boldsymbol{a}) - Q_t(\boldsymbol{s}_t, \boldsymbol{a}_t)\big]$$

Q-learning is simpler than SARSA, but approximate.

Learn to play tic-tac-toe: $r = +1$ (win), $r = 0$ (draw), $r = -1$ (lose).





Summary

This last part of the lectures is based on the sources

Linander, *The mathematics behind large language models* (2023)
Penke, *A mathematician's introduction to transformers and large language models* (2022)

Attention mechanism efficiently represents long-range dependencies (overcomes vanishing-gradient problem of recurrent nets).

Efficient because attention layer processes inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$ in parallel using matrix-vector products.

Transformers consist of stacked multi-headed attention layers, fully connected Feed-forward layers, and residual connections.

Trained in the standard fashion plus reinforcement learning.

Transformers for image analysis Liu *et al.* (2021) arxiv:2103.14030