Lecture 8:
Deep Learning for Games

ECS7002P - Artificial Intelligence in Games

Diego Perez Liebana - diego.perez@qmul.ac.uk
Office: CS.301

http://gameai.eecs.qmul.ac.uk

Queen Mary University of London

Outline

Deep Learning

A use case: Learning to play Atari Games

A use case: Alpha Go and Alpha Zero

Deep Learning for Games

Deep Learning

From Neural Networks to Deep Learning

Until 1998, advances were made in Neural Networks:

- Backpropagation
- Recurrent Long-Short Term Memory (LSTM) Networks
- OCR with Convolutional Neural Networks.

... but progressed stopped. Other algorithms (i.e. Support Vector Machines - SVM) were achieving similar results with fewer heuristics, tuning and generalization proofs.

Furthermore:

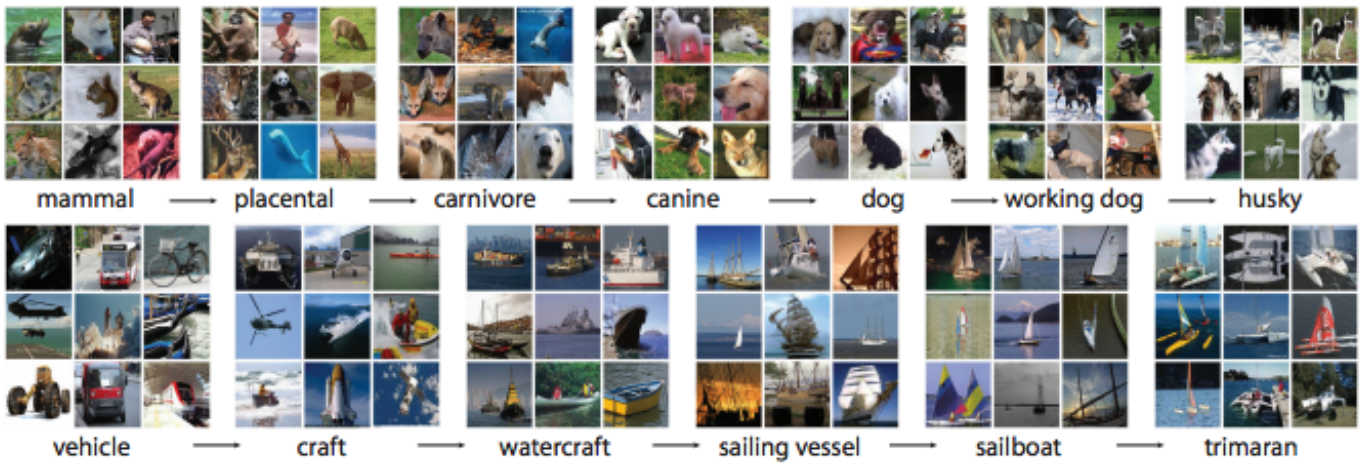
- NN weren't able to work very well beyond a few layers.
- Not enough processing power (no GPUs available)
- Lack of data: prone to overfitting.

Deep Neural Networks

The arrival of processing power and tons of data made training multi-layered neural networks easier. A NN with more than three layers (including input and output) qualifies as a **deep** neural network (DNN).

[Deng et al., 2009]

- Imagenet: Database of 16 millions of annotated images (by humans) semantically organized.

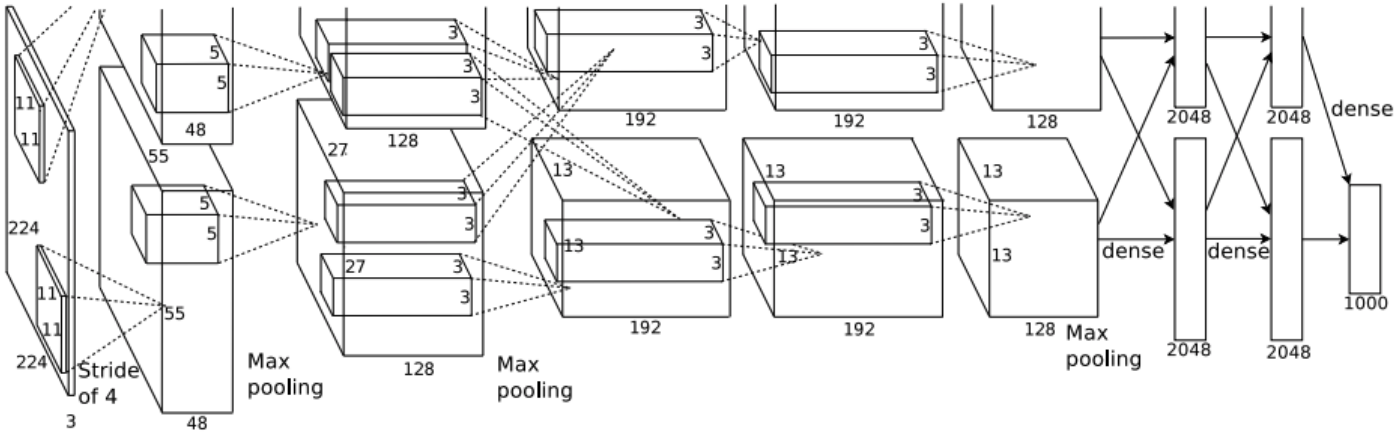


[Deng et al., 2009] Deng, Jia, et al. "Imagenet: A large-scale hierarchical image database." Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on. IEEE, 2009.

Deep Neural Networks - Alexnet

[Krizhevsky et al., 2012]

- Trains a DNN on the Imagenet set
- Uses two GPUs to train.
- Adds some theoretical improvements:
  - Rectifier Linear Units (ReLU) instead of sigmoid or tanh activations
  - Data Augmentation: techniques to increase the size of the input dataset.
  - Dropout: a form of regularization to reduce overfitting.
- Alexnet:



- Results: 16.4% classification error. Best before was 26.2%!!

[Krizhevsky et al., 2012] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

Deep Learning - how does it work?

One-line definition: *A family of* **parametric**, **non-linear** *and* **hierarchical** *representation learning functions, which are massively optimized with* **stochastic gradient descent** *to encode* **domain knowledge**.

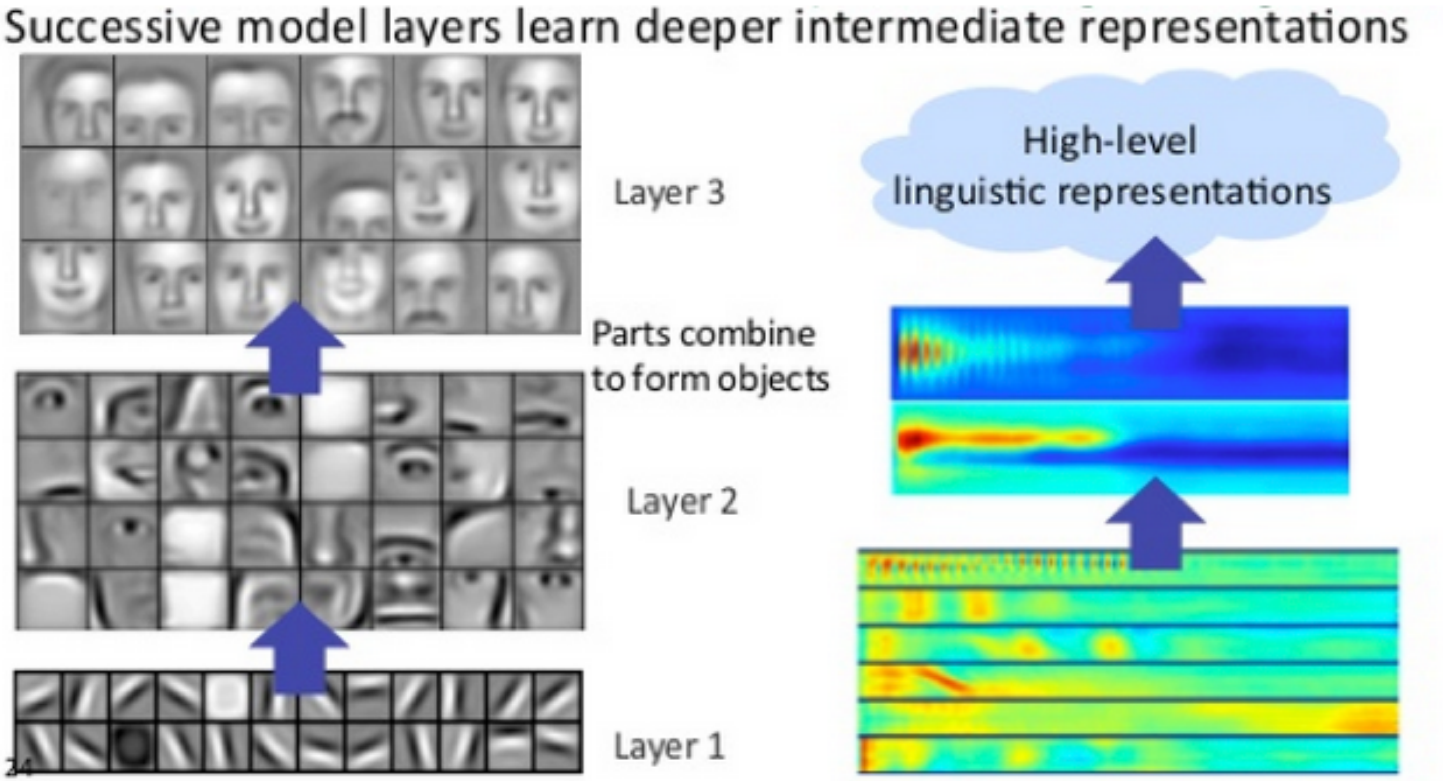
- Parametric: A set of weights to be adjusted to capture features.
- Non-linear: Non-linear function activations and other tools.
- Hierarchical: several stack layers (with different levels of complexity).
- Stochastic Gradient Descent: stochastically approximates the solution following a gradient.
- Domain Knowledge: layers capture features of the input data.

Presence of several neat tricks: dropouts, pooling, convolutions, etc.

Deep Learning - Hierarchy

Feature Hierarchy:

- Each layer of nodes trains on a distinct set of features based on the previous layer's output.
- The deeper the layer, the more complex the features can be recognized.
- Each layer aggregates and recombines features from the previous layer.



Deep Learning - Hierarchy and Learning

Feature hierarchy makes the network capable of:

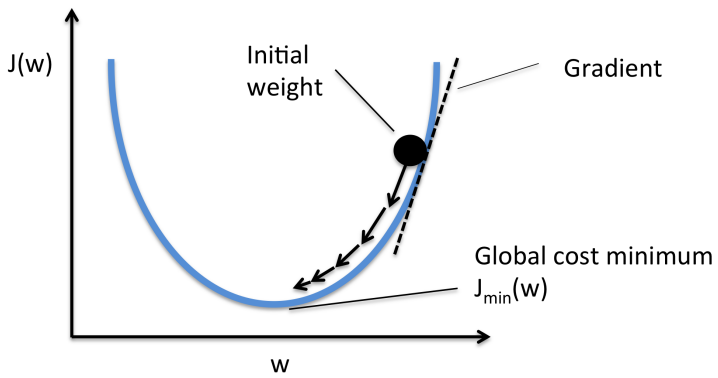
- Handling very large high-dimensional data sets.
- Discovering latent structure in unlabelled unstructured data (raw data): the World!
- This means that DL methods can perform feature extraction without human intervention. This is a huge boost in processing and capabilities of investigators.

Learning:

- Each node in the network tries to reconstruct the input from which it draws its samples.
- They attempt to minimize the difference between what the network predicts and the input data.
- By drawing correlations between features and labeled data.
- The more data a NN can train on, the more accurate it is likely to be.
- Deep-learning networks end in an output layer: a logistic, or softmax, classifier that assigns a likelihood to a particular outcome or label.
- Training is (typically) done by **Stochastic Gradient Descent**.

Deep Learning - Gradient Descent

Gradient Descent (GD) is an optimization function that adjusts (the NN) weights according to the error caused in classification. Gradient is a measure of the slope between to variables (rise over run). NN works with the slope between the network's error and the weights. That is, *how does the error vary as the weight is adjusted*. The relationship between network Error and each of those weights is a derivative, $\frac{J(W_j)}{d_W}$, that measures the degree to which a slight change in a weight causes a slight change in the error.



$$W_j \leftarrow W_j + \alpha \frac{\delta}{\delta W_j} J(W)$$

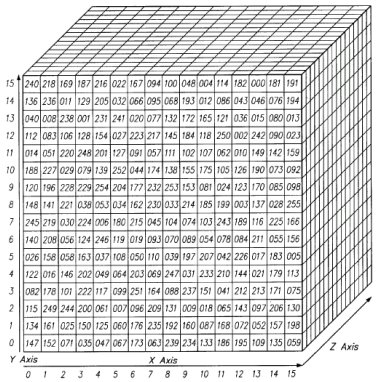
Learning happens "simply" by adjusting the model's weights in response to the error the network produces, until this error can't be reduced any more.

Stochastic Gradient Descent (SGD) works learning after using a single (or a minibatch) data sample at each iteration, while GD uses the whole data set.

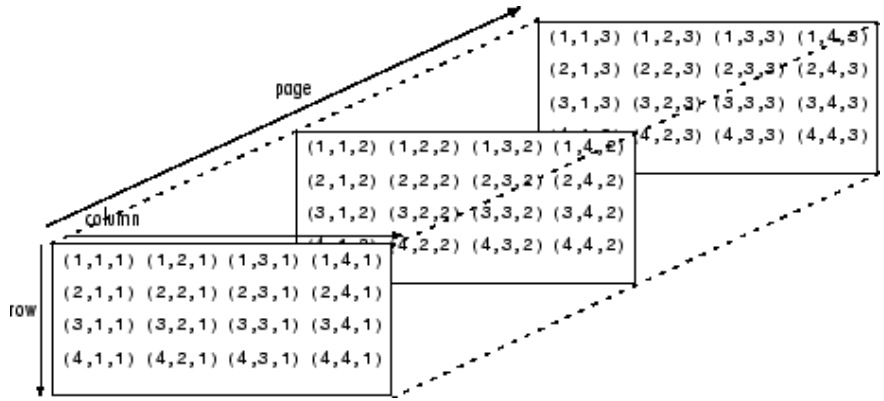
Convolutional Nets

Deep Convolutional Neural Networks (CNNs) are widely used within the Deep Learning community. They are especially well suited to classify images, cluster by similarity, object recognition within scenes, text analytics ...

Tensors are matrices of numbers used by CNNs to process images. Concretely, this are 4-Dimensional matrices.



3D



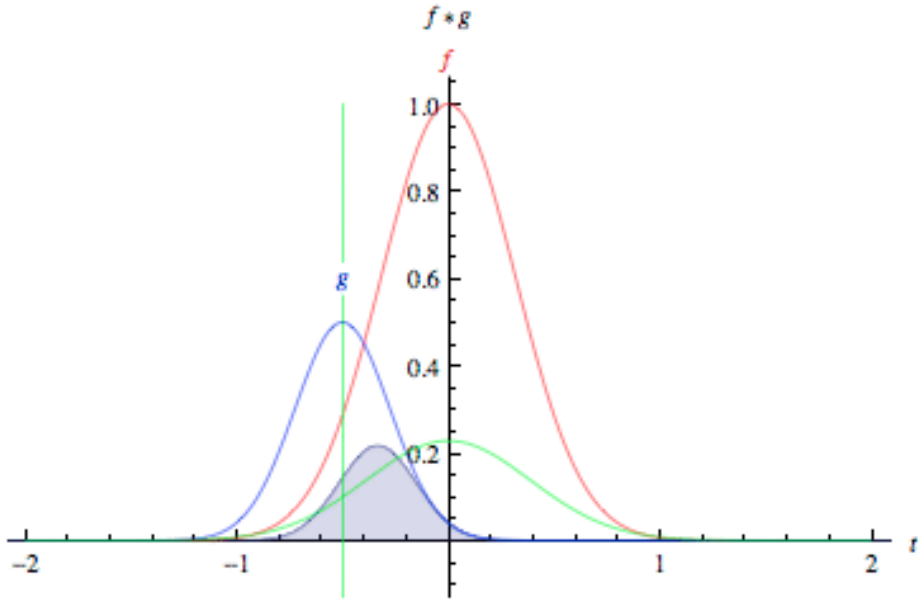
4D

4 Dimensions?

- 1^{st} and 2^{nd}: Width and Height of the image.
- 3^{rd}: Encoding *channel* (i.e. RGB) of the image.
- 4^{th}: Batch of images (input) / Feature Maps (intermediate layers).

A Convolutional Layer

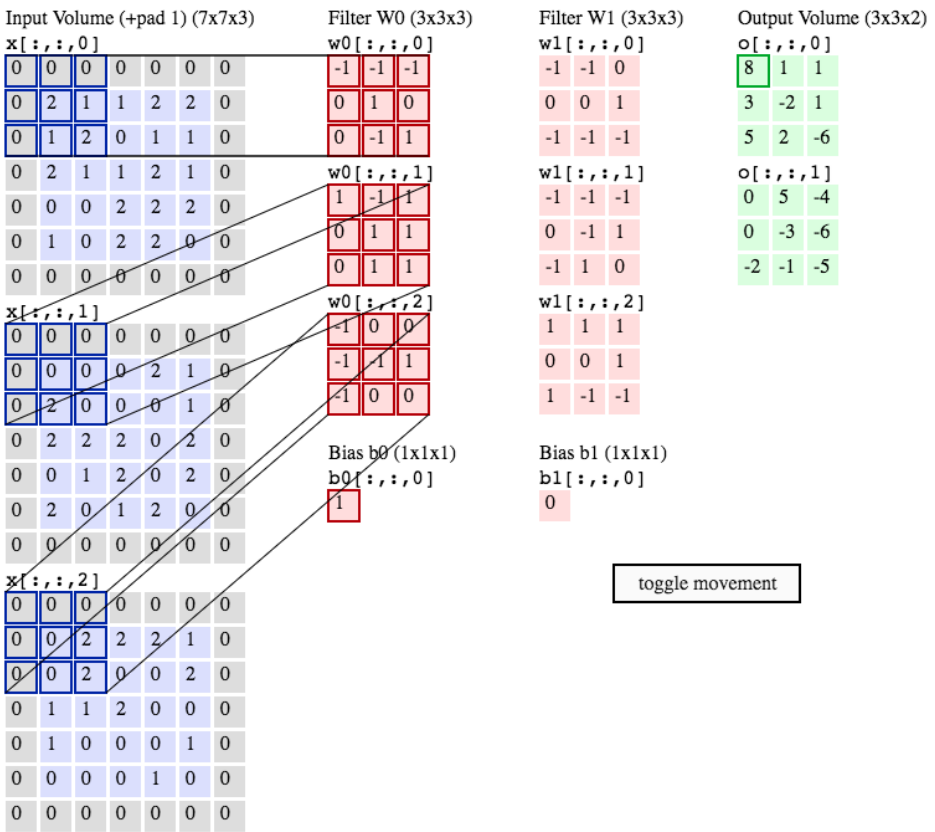
In a convolution, a filter (g) is shifted across an input signal or function (f). The product of those two functions' overlap at each point along the x-axis is their convolution ($f \times g$):



Demo: `https://skymind.ai/wiki/convolutional-network#define`

- f is the input, the "image" being analyzed.
- The filter g analyzes the image by building a feature.
- Convolutional nets pass many filters g_i over a single image f, each one of them picking a different feature.

A Convolutional Layer



In one convolutional layer, the input data (RGB data: blue matrices) is analyzed by filters (red matrices) to produce the output (in green).

Each element is computed by element-wise multiplying the highlighted input with the filter, summing it up, and then offsetting the result by the bias (it's a **dot product**).

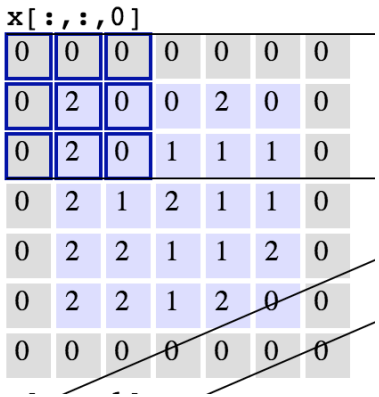
The results are gathered in another matrix, named *activation map*.

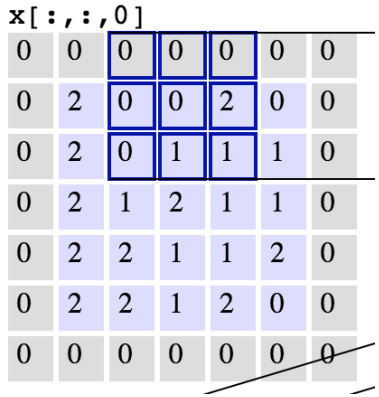
Demo: `https://cs231n.github.io/convolutional-networks/#conv`
Demo with images: `http://deeplizard.com/learn/video/k6ZF1TSniYk` (min 4:52)

Filters are like *patterns* that identify the presence of *features* in the input. Their values (W) are learnt by the DNN.

A Convolutional Layer: Stride

The *stride* is the size of the step used to move the filter across the layer (i.e. a stride of 1 means moving the filter one column at a time, and 1 row when reaching the end of the column).





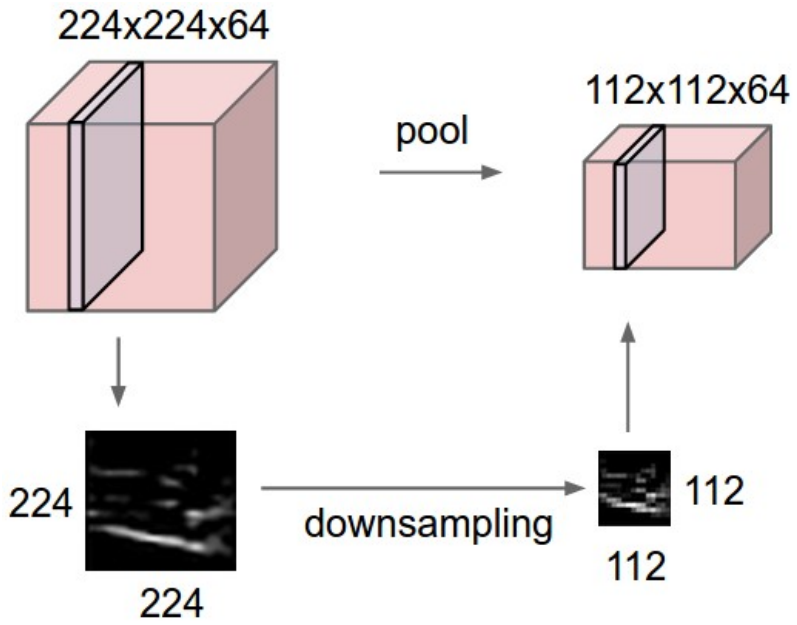
Example with stride 2

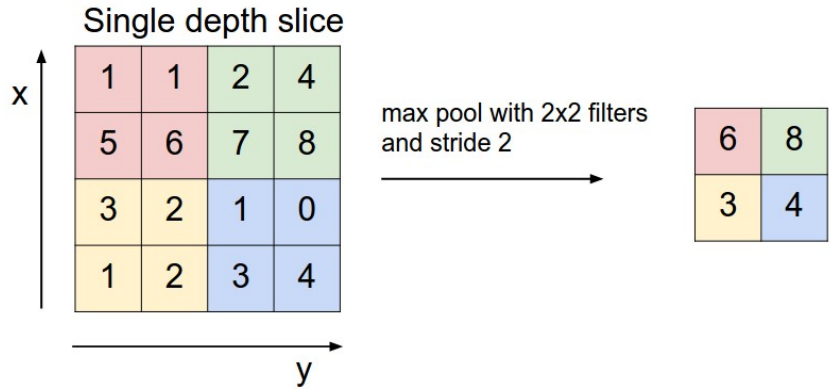
- A larger stride involves less steps, producing a small activation map.
- Viceversa for smaller strides.
- Smaller activation maps require less computation effort and time.

Downsampling layers

The next layer, after a convolutional one, is a *downsampling* layer (also known as *max pooling* or *subsampling*). The objective of this layer is to reduce the dimensionality, by taking the maximum value from patches from the activation map and discarding the rest.

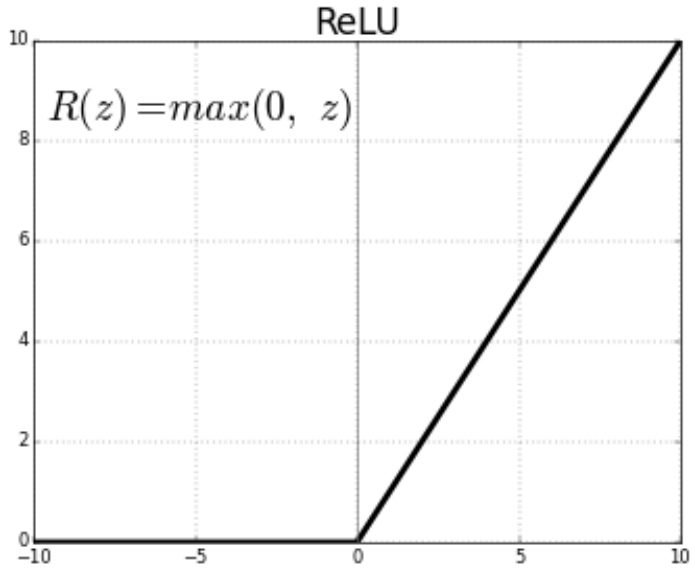
It's a trade-off between losing information and reducing the storage and processing effort required for learning.





Rectifier Linear Units (ReLU)

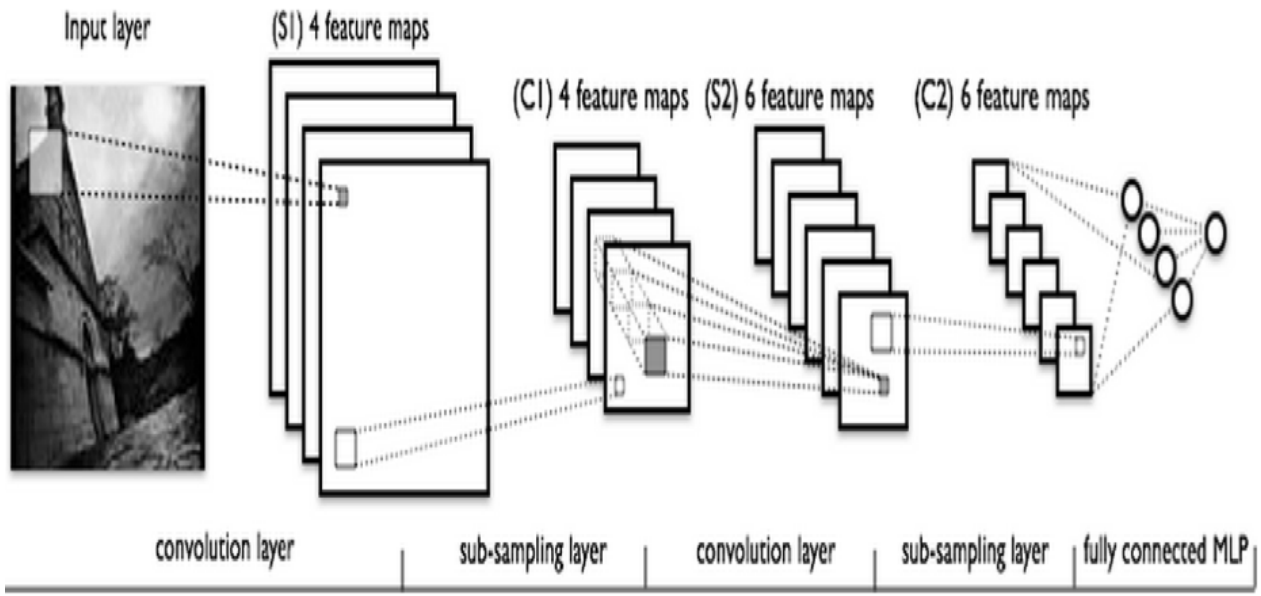
Rectifier Linear Units (ReLU) is a non-linear activation function.



A ReLU activation function can be used instead of traditional tanh or sigmoid functions. This allows the NN not to be restricted to combinations of linear functions, allowing to represent a much richer set of non-linear functions.

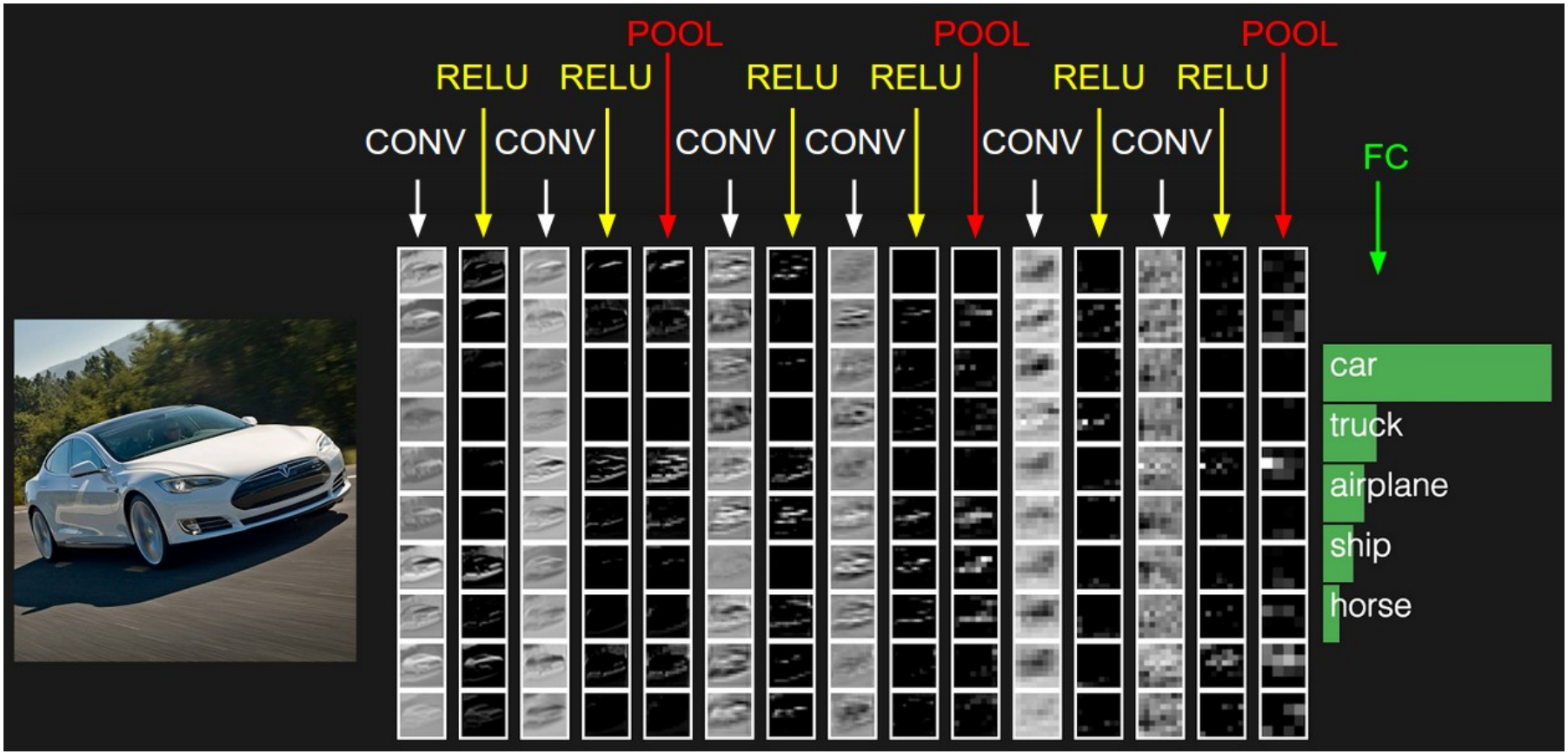
The complete Convolutional Neural Network

- Convolution and sub-sampling layers alternate, taking the data (image) as an input and reducing its dimensionality, while extracting feature maps.
- A fully connected MLP layer is appended at the end, in charge of the classification task.
  - Input: features extracted by the previous layers.
  - Output: class selected.



The complete Convolutional Neural Network

An example with ReLU activation functions:



A use case: Learning to play Atari Games

Human-level control through deep reinforcement learning

LETTER

doi:10.1038/nature14236

Human–level control through deep reinforcement learning

Volodymyr Mnih[1]*, Koray Kavukcuoglu[1]*, David Silver[1]*, Andrei A. Rusu[1], Joel Veness[1], Marc G. Bellemare[1], Alex Graves[1], Martin Riedmiller[1], Andreas K. Fidjeland[1], Georg Ostrovski[1], Stig Petersen[1], Charles Beattie[1], Amir Sadik[1], Ioannis Antonoglou[1], Helen King[1], Dharshan Kumaran[1], Daan Wierstra[1], Shane Legg[1] & Demis Hassabis[1]

The theory of reinforcement learning provides a normative account[1], deeply rooted in psychological[2] and neuroscientific[3] perspectives on animal behaviour, of how agents may optimize their control of an environment. To use reinforcement learning successfully in situations approaching real-world complexity, however, agents are confronted with a difficult task: they must derive efficient representations of the environment from high-dimensional sensory inputs, and use these to generalize past experience to new situations. Remarkably, humans and other animals seem to solve this problem through a harmonious combination of reinforcement learning and hierarchical sensory processing systems[4,5], the former evidenced by a wealth of neural data revealing notable parallels between the phasic signals emitted by dopaminergic neurons and temporal difference reinforcement learning algorithms[3]. While reinforcement learning agents have achieved some successes in a variety of domains[6–8], their applicability has previously been limited to domains in which useful features can be handcrafted

agent is to select actions in a fashion that maximizes cumulative future reward. More formally, we use a deep convolutional neural network to approximate the optimal action-value function

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t = s,\, a_t = a,\, \pi\right],$$

which is the maximum sum of rewards r_t discounted by γ at each time-step t, achievable by a behaviour policy $\pi = P(a|s)$, after making an observation (s) and taking an action (a) (see Methods)[19].

Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (also known as Q) function[20]. This instability has several causes: the correlations present in the sequence of observations, the fact that small updates to Q may significantly change the policy and therefore change the data distribution, and the correlations between the action-values (Q) and the target values $r + \gamma \max_{a'} Q(s', a')$.

Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529-533.

In a nutshell

This work presents a **Deep Q-Network**, which can learn successful *policies* directly from high-dimensional sensory inputs using end-to-end reinforcement learning.

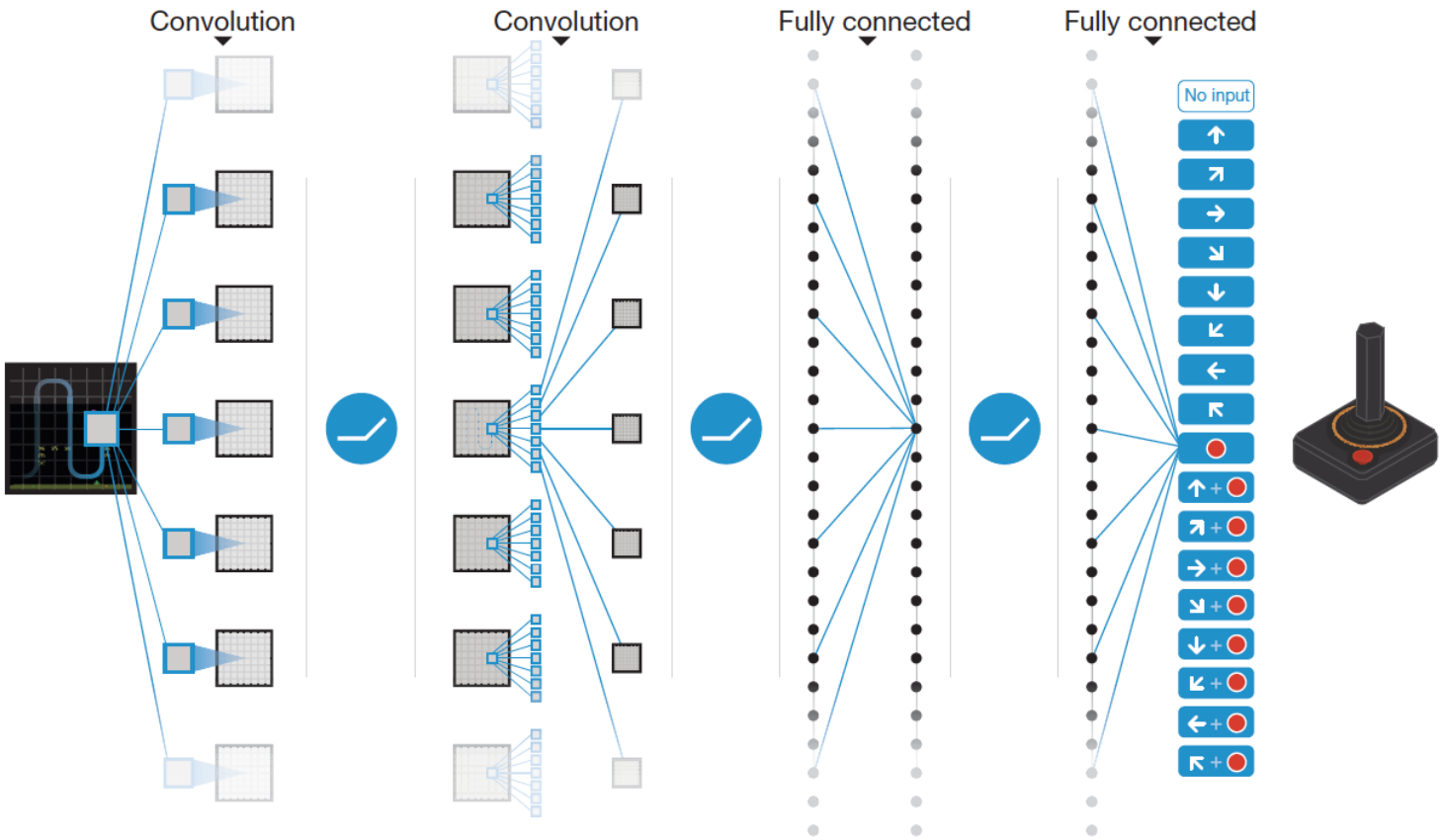
The proposed algorithm combines Deep Neural Networks (concretely, a Deep Convolutional Neural Network) and Q-Learning.

The network is tested on 49 of the games from the classic Atari 2600 collection.

The network received only the screen capture and current score of the game.

It outperformed all other (general) learning algorithms across games and achieved a level comparable to that of a professional human games testers.

Model Architecture



- Input: state representation (screen capture)
- Layers: Conv + ReLU repeated, followed by a fully connected MLP layers.
- Output: one unit for each possible action (predicted Q-values).
- It's fast: only one pass through the NN computes Q-values for all actions.

Training

Playing Breakout: `https://www.youtube.com/watch?v=Q70ulPJW3Gk`

The problem is formulated as a Markov Decision Process (MDP), with states, actions and rewards. Because the agent only observes the current screen, the task is partially observed (it is impossible to fully understand the current situation from only the current screen x_t).

Therefore, sequences of actions and observations, $x_1, a_1, x_2, \ldots, a_{t-1}, x_t$, are input to the algorithm, which then learns game strategies depending upon these sequences.

The algorithm counted with the following input:

- Input data as the screen capture of the state.
- All actions available (to build the CNN)
- Game score
- Life count (for episode termination).

Deep Q-Learning

As with any MDP, the target is to maximize accumulative reward. Concretely, the objective is to act according to $Q^*(s, a)$ following a policy π. Rather than calculating Q^* iteratively, a function approximator is used to approximate Q^*: a (deep) Q-Network. The following points characterize this algorithm:

- Model free: Solves the RL task by playing games in the emulator.
- Off-policy: learns about the greedy policy, following an ϵ-greedy policy for exploration of the search space.
- Experience replay:
  - Agent's experiences are stored.
  - Each experience is a tuple $e_t = \{s_t, a_t, r_t, s_{t+1}\}$.
  - **Stochastic** Gradient Descent (SGD): uses mini-batches (size 32) of experiences to train the network's weights *theta*.
- Two sets of weights for the network: θ and θ^-.
  - One set $\hat{Q}(\theta^-)$ is used to generate targets for the other one $Q(\theta^-)$
  - θ is updated in Q with SGD.
  - Every C time steps, $\theta_i^- = \theta_{i-1}$

Algorithm: Deep Q-Learning with Experience Replay

Initialize replay memory D

Initialize action-value function Q and target action-value function $\hat{Q}$ with random weights θ and θ^-.

For M episodes **do**:

1. Initialize $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
2. **For** T time steps **do**:
   - 2.1 Use ϵ-greedy to select action a_t from $Q(s_t, a)$.
   - 2.2 Execute action a_t in emulator, observe r_t and x_{t+1}
   - 2.3 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
   - 2.4 Store transition $\{\phi_t, a_t, r_t, \phi_{t+1}\}$ in D
   - 2.5 Sample random minibatch of J transitions $\{\phi_j, a_j, r_j, \phi_{j+1}\}$ from D
   - 2.6 **Foreach** memory reply $j \in J$ **do**:

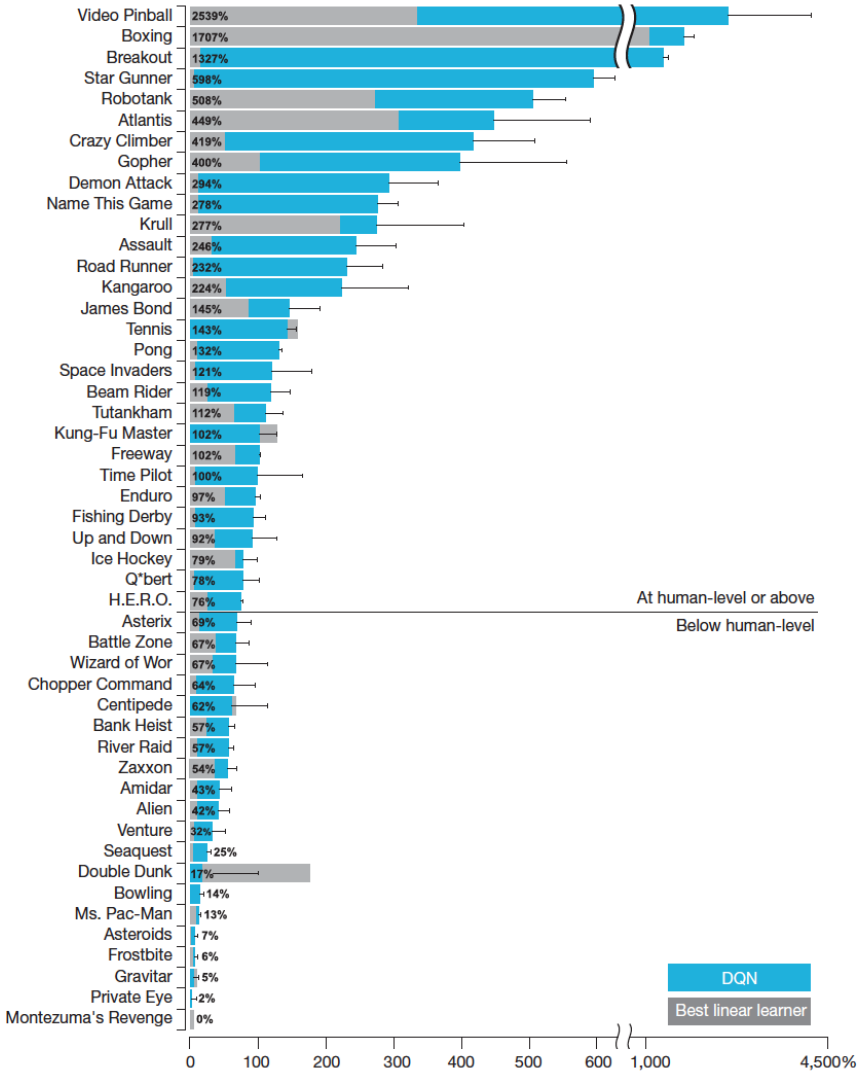
     What's the value of applying a_j from s_j (Q(s_j, a_j))?

     2.6.1 $y_j = r_j$ if episode terminates at step $j + 1$; Otherwise:

     $y_j = r_j + \max_{a'} \hat{Q}(\phi_{j+1,a'})$

     2.6.2 Perform a gradient descent step on y_j with respect to the network parameters θ.
   - 2.7 Every C steps, reset $\hat{Q} = Q$.

Results


Video Pinball 2539%
Boxing 1707%
Breakout 1327%
Star Gunner 598%
Robotank 508%
Atlantis 449%
Crazy Climber 419%
Gopher 400%
Demon Attack 294%
Name This Game 278%
Krull 277%
Assault 246%
Road Runner 232%
Kangaroo 224%
James Bond 145%
Tennis 143%
Pong 132%
Space Invaders 121%
Beam Rider 119%
Tutankham 112%
Kung-Fu Master 102%
Freeway 102%
Time Pilot 100%
Enduro 97%
Fishing Derby 93%
Up and Down 92%
Ice Hockey 79%
Q*bert 78%
H.E.R.O. 76%
Asterix 69%
Battle Zone 67%
Wizard of Wor 67%
Chopper Command 64%
Centipede 62%
Bank Heist 57%
River Raid 57%
Zaxxon 54%
Amidar 43%
Alien 42%
Venture 32%
Seaquest 25%
Double Dunk 17%
Bowling 14%
Ms. Pac-Man 13%
Asteroids 7%
Frostbite 6%
Gravitar 5%
Private Eye 2%
Montezuma's Revenge 0%
At human-level or above
Below human-level
DQN
Best linear learner
0
100
200
300
400
500
600
1,000
4,500%

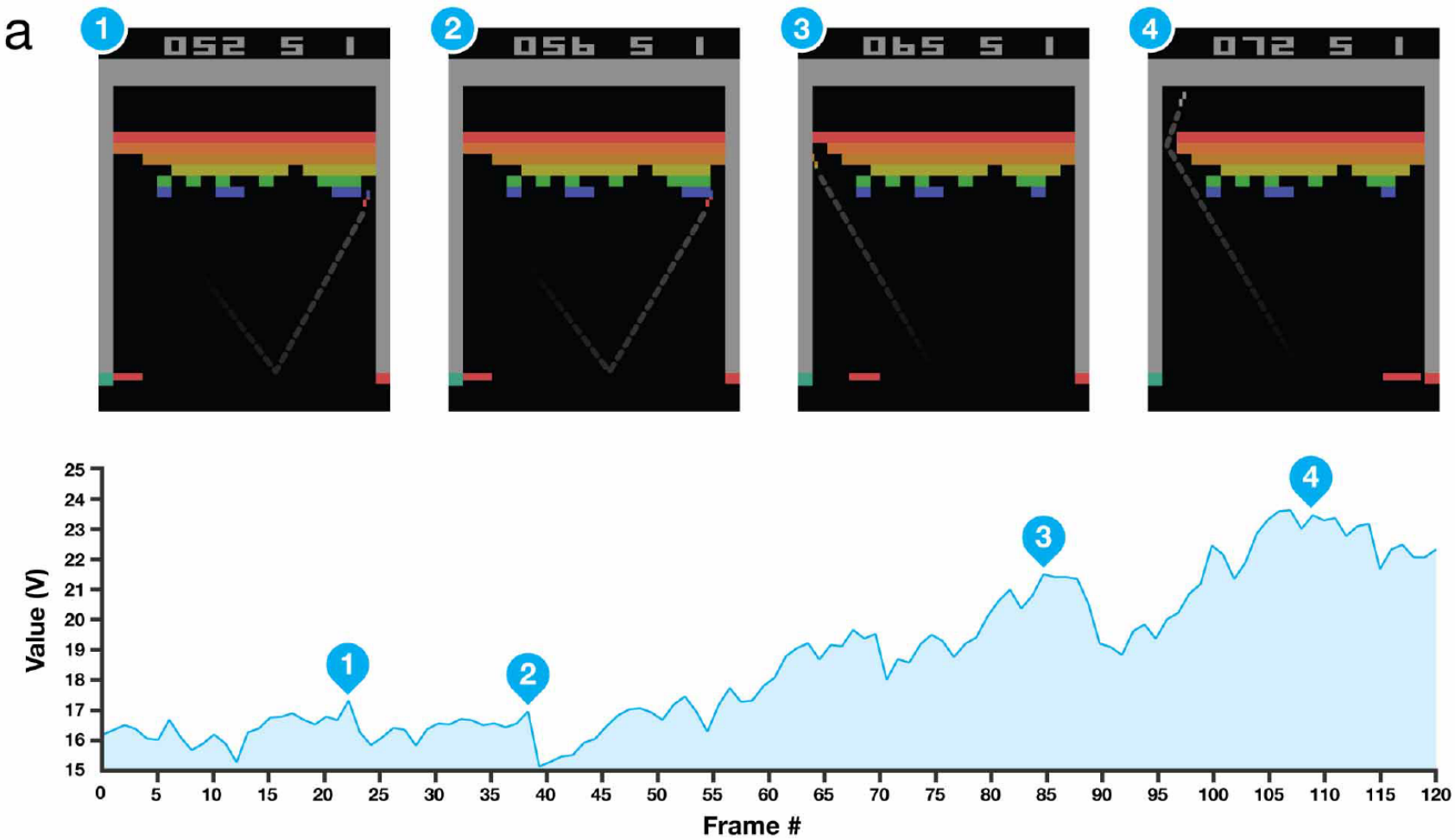
Results Table

Extended Data Table 2 | Comparison of games scores obtained by DQN agents with methods from the literature[12,15] and a professional human games tester

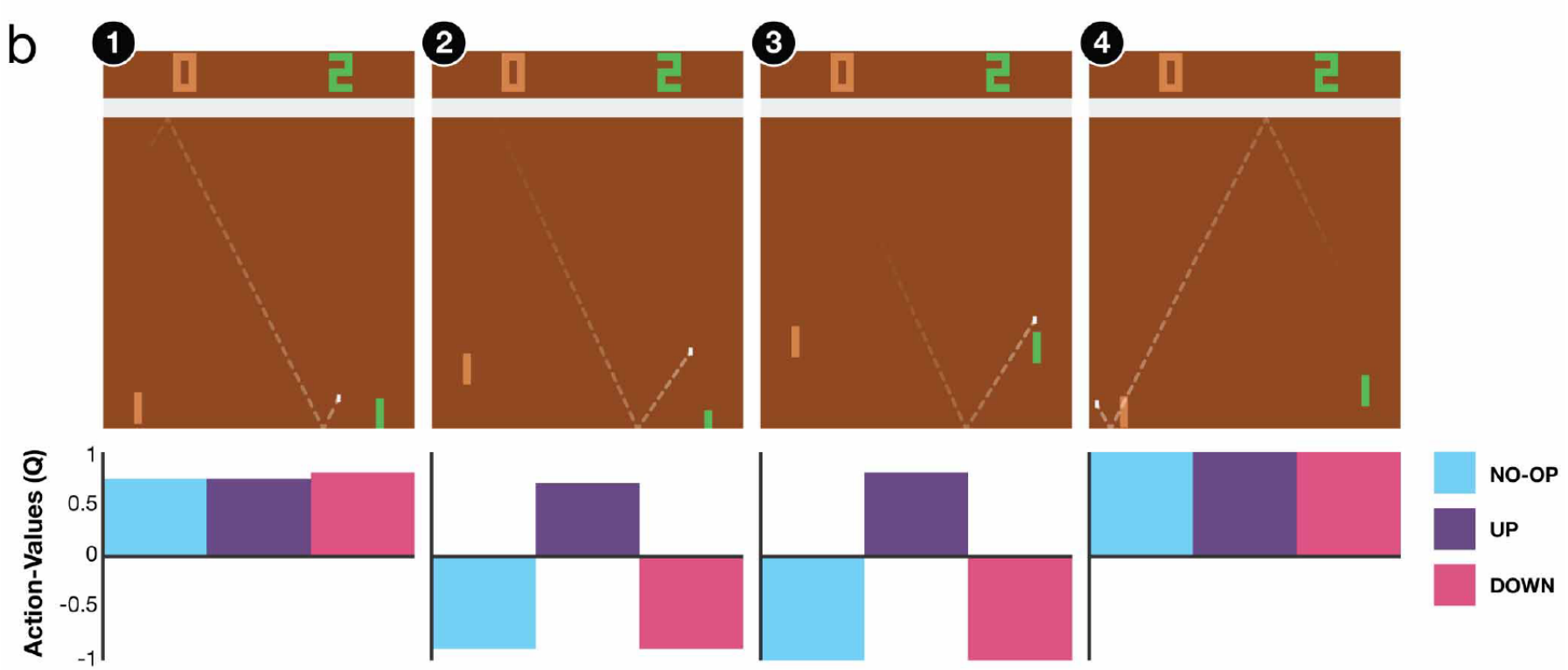
Game	Random Play	Best Linear Learner	Contingency (SARSA)	Human	DQN (± std)	Normalized DQN (% Human)
Alien	227.8	939.2	103.2	6875	3069 (±1093)	42.7%
Amidar	5.8	103.4	183.6	1676	739.5 (±3024)	43.9%
Assault	222.4	628	537	1496	3359(±775)	246.2%
Asterix	210	987.3	1332	8503	6012 (±1744)	70.0%
Asteroids	719.1	907.3	89	13157	1629 (±542)	7.3%
Atlantis	12850	62687	852.9	29028	85641(±17600)	449.9%
Bank Heist	14.2	190.8	67.4	734.4	429.7 (±650)	57.7%
Battle Zone	2360	15820	16.2	37800	26300 (±7725)	67.6%
Beam Rider	363.9	929.4	1743	5775	6846 (±1619)	119.8%
Bowling	23.1	43.9	36.4	154.8	42.4 (±88)	14.7%
Boxing	0.1	44	9.8	4.3	71.8 (±8.4)	1707.9%
Breakout	1.7	5.2	6.1	31.8	401.2 (±26.9)	1327.2%
Centipede	2091	8803	4647	11963	8309(±5237)	63.0%
Chopper Command	811	1582	16.9	9882	6687 (±2916)	64.8%
Crazy Climber	10781	23411	149.8	35411	114103 (±22797)	419.5%
Demon Attack	152.1	520.5	0	3401	9711 (±2406)	294.2%
Double Dunk	-18.6	-13.1	-16	-15.5	-18.1 (±2.6)	17.1%
Enduro	0	129.1	159.4	309.6	301.8 (±24.6)	97.5%
Fishing Derby	-91.7	-89.5	-85.1	5.5	-0.8 (±19.0)	93.5%
Freeway	0	19.1	19.7	29.6	30.3 (±0.7)	102.4%
Frostbite	65.2	216.9	180.9	4335	328.3 (±250.5)	6.2%

...

State-Value function learnt in a game (Breakout)



Action-Value function learnt in a game (Pong)



Deep Learning for Games

A use case: Alpha Go and Alpha Zero

Mastering the game of Go with deep neural networks and tree search

ARTICLE

doi:10.1038/nature16961

Mastering the game of Go with deep neural networks and tree search

David Silver[1]*, Aja Huang[1]*, Chris J. Maddison[1], Arthur Guez[1], Laurent Sifre[1], George van den Driessche[1], Julian Schrittwieser[1], Ioannis Antonoglou[1], Veda Panneershelvam[1], Marc Lanctot[1], Sander Dieleman[1], Dominik Grewe[1], John Nham[2], Nal Kalchbrenner[1], Ilya Sutskever[2], Timothy Lillicrap[1], Madeleine Leach[1], Koray Kavukcuoglu[1], Thore Graepel[1] & Demis Hassabis[1]

The game of Go has long been viewed as the most challenging of classic games for artificial intelligence owing to its enormous search space and the difficulty of evaluating board positions and moves. Here we introduce a new approach to computer Go that uses 'value networks' to evaluate board positions and 'policy networks' to select moves. These deep neural networks are trained by a novel combination of supervised learning from human expert games, and reinforcement learning from games of self-play. Without any lookahead search, the neural networks play Go at the level of state-of-the-art Monte Carlo tree search programs that simulate thousands of random games of self-play. We also introduce a new search algorithm that combines Monte Carlo simulation with value and policy networks. Using this search algorithm, our program AlphaGo achieved a 99.8% winning rate against other Go programs, and defeated the human European Go champion by 5 games to 0. This is the first time that a computer program has defeated a human professional player in the full-sized game of Go, a feat previously thought to be at least a decade away.

All games of perfect information have an optimal value function, $v^*(s)$, which determines the outcome of the game, from every board position

policies[13–15] or value functions[16] based on a linear combination of input features.

Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." Nature 529.7587 (2016): 484-489.

In a nutshell

Combines Deep Neural Networks and Monte Carlo Tree Search to achieve professional level of play in the game of Go.

Position evaluation in Go, using a value function $v(s)$ was believed to be intractable: AlphaGo uses a Value Network.

Go has an enormous branching factor: Alpha Go uses a Policy Network for sampling actions from a state, embedded in MCTS.

AlphaGo achieved a 99.8% winning rate against other Go programs, defeated the human European Go champion $5-0$, and the World Go champion $4-1$.

Training the Policy and Value Networks

All networks share the a similar structure:

- A Deep Neural Network with 13 layers.
- The input of the policy network is a simple representation of the board state s.
- Alternate between convolutional layers and rectifier nonlinearities.
- The output layer is different:
  - Policy Network: a final Softmax layer outputs a probability distribution over all legal moves a.
  - Value Network: a single prediction of the outcome as the output.

Training in three phases:

1. **Supervised learning of policy networks**:

- The policy network p_ρ is trained on randomly sampled state-action pairs with Stochastic Gradient Descent to maximize the likelihood of a human move a from s.
- Data: 30 million positions from the KGS Go Server.
- The network predicted expert moves on a held out test set with an accuracy of 57.0% using all input features (previous state of the art at 44.4%).

Training the Policy and Value Networks

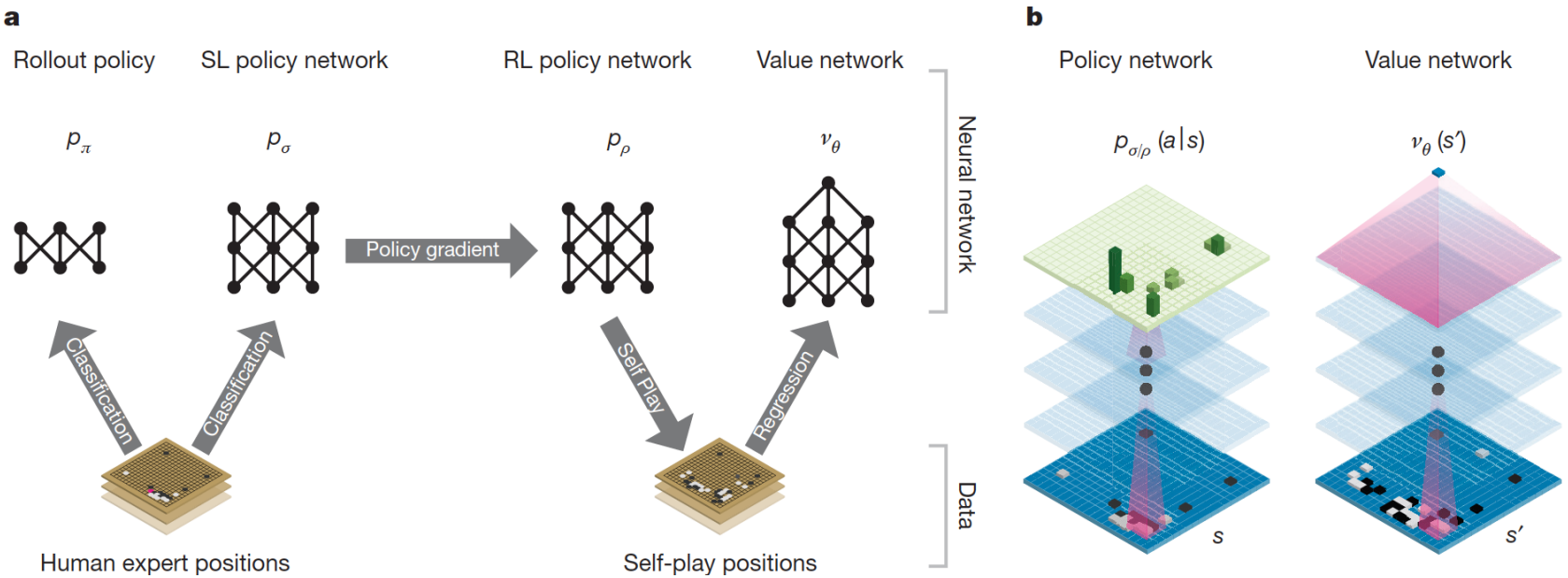
2. **Reinforcement Learning of policy networks**:

   - Using self-play RL (Policy Gradient), the trained Supervised Learning Policy network plays against a randomly selected previous iteration of itself.
   - This allows the network to reduce overfitting.
   - Performance of 85% victories against previously strongest Go player (Pachi), while other supervised learning of CNNs achieved 11% only.

3. **Reinforcement Learning of value networks**:

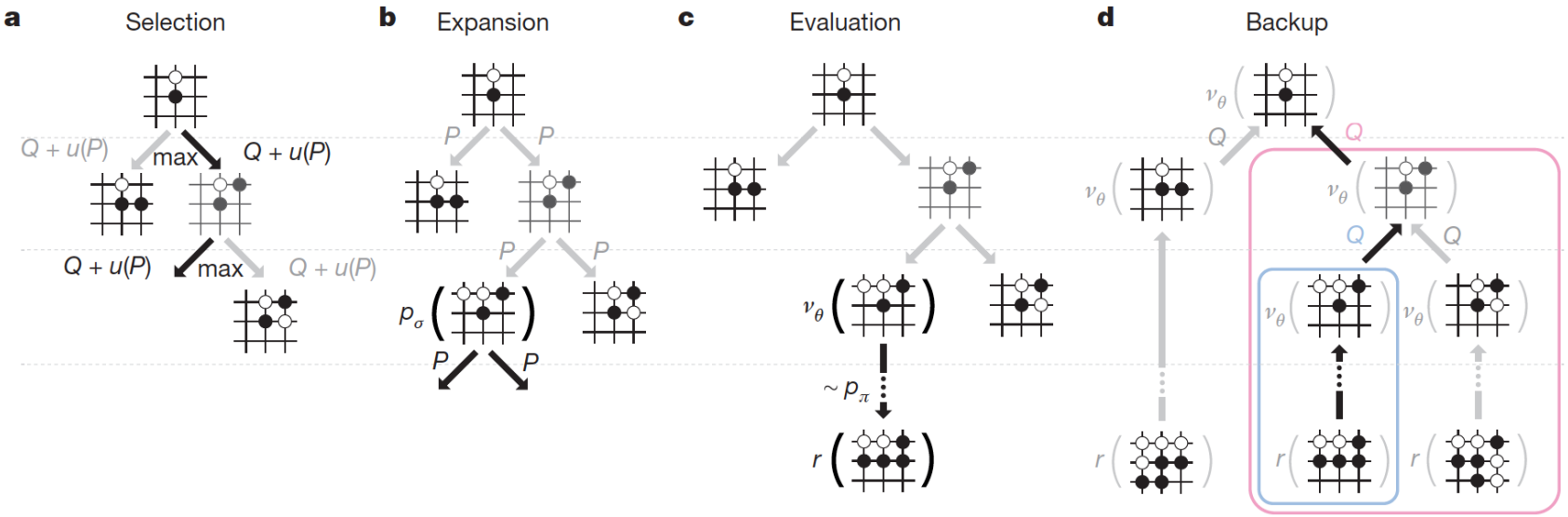
   - The value network $v_\theta(s)$ outputs a single prediction instead of a probability distribution.
   - The prediction is the outcome of a game (win or loss).
   - Trained on 30 million distinct non consecutive positions, from separated games played by the RL policy network against itself.
   - A single evaluation of $v_\theta(s)$ approached the accuracy of Monte Carlo rollouts using the RL policy network p_ρ, but using 15,000 times less computation.

Training the Policy and Value Networks



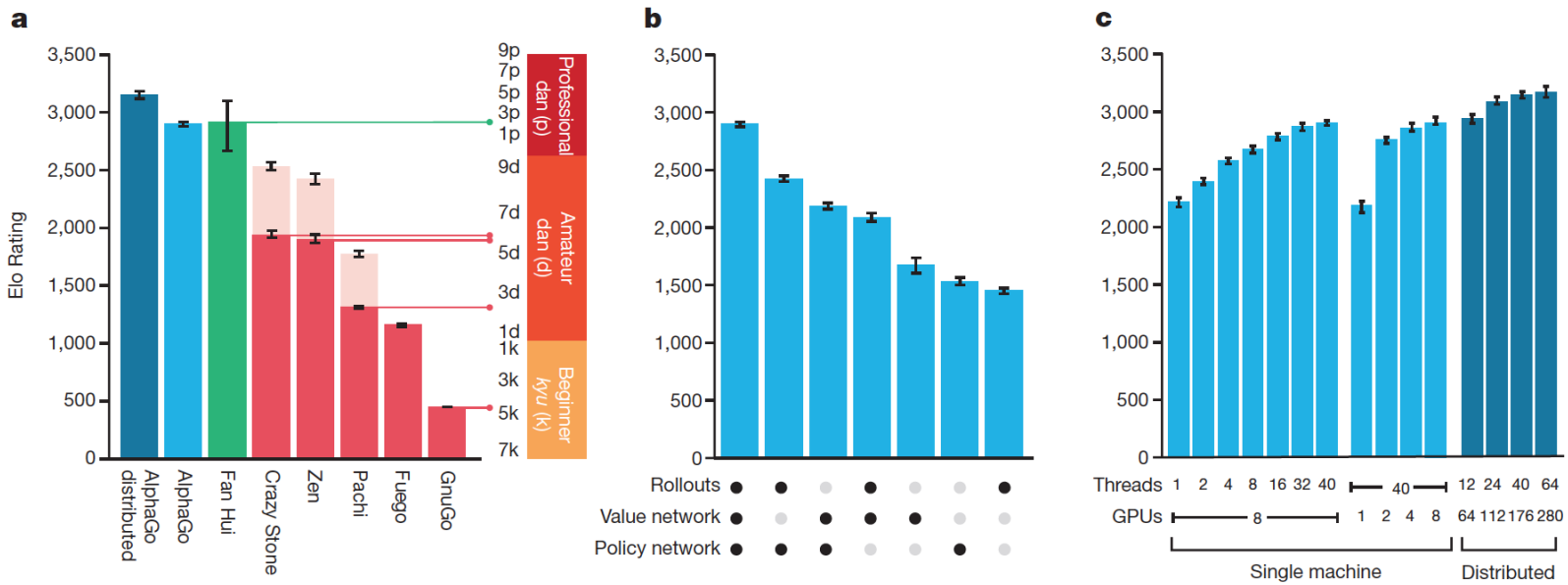
Searching with policy and value networks

MCTS uses the Value and Policy networks to select the move to play.



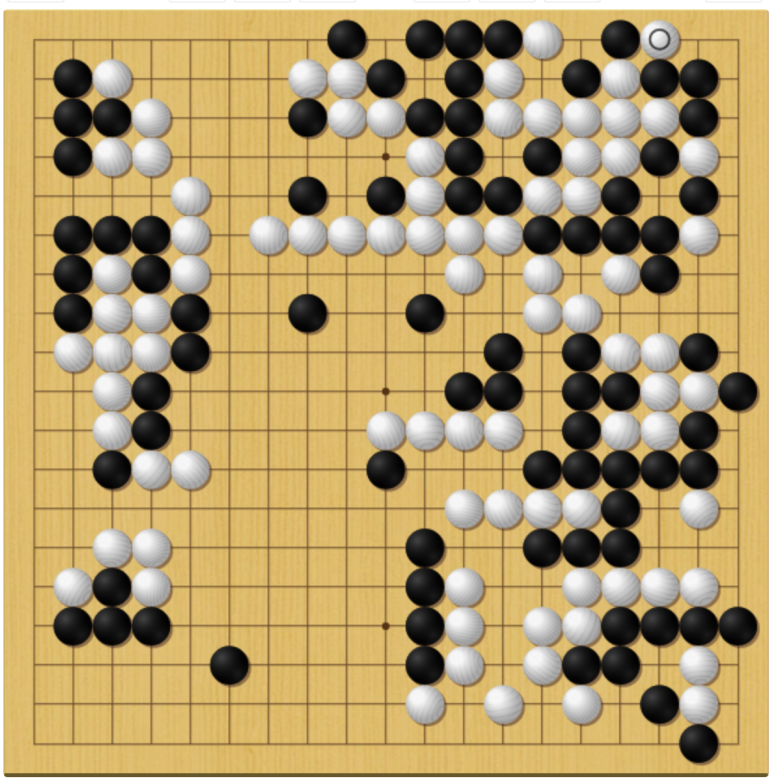
- Selection: an UCT policy.
- Expansion: adds a new node.
- Evaluation of a node combined out of two measurements:
  - By the value network $v_\theta(s)$.
  - Outcome Z of a random rollout played until a terminal node with policy p_π.
  - $V(S) = (1 - \lambda) \times v_\theta(s) + \lambda \times Z$, $(\lambda = 0.5)$.
- Back-up: update the Q value in the nodes of the tree.

Results Comparison



AlphaGo vs Lee Sedol

In March 2016, AlphaGo won 4-1 against the top Go player in the world over the past decade.



All games are available here:
`https://deepmind.com/research/alphago/alphago-games-english/`

AlphaGo Zero (Oct 2017)

AlphaGo Zero improves AlphaGo:

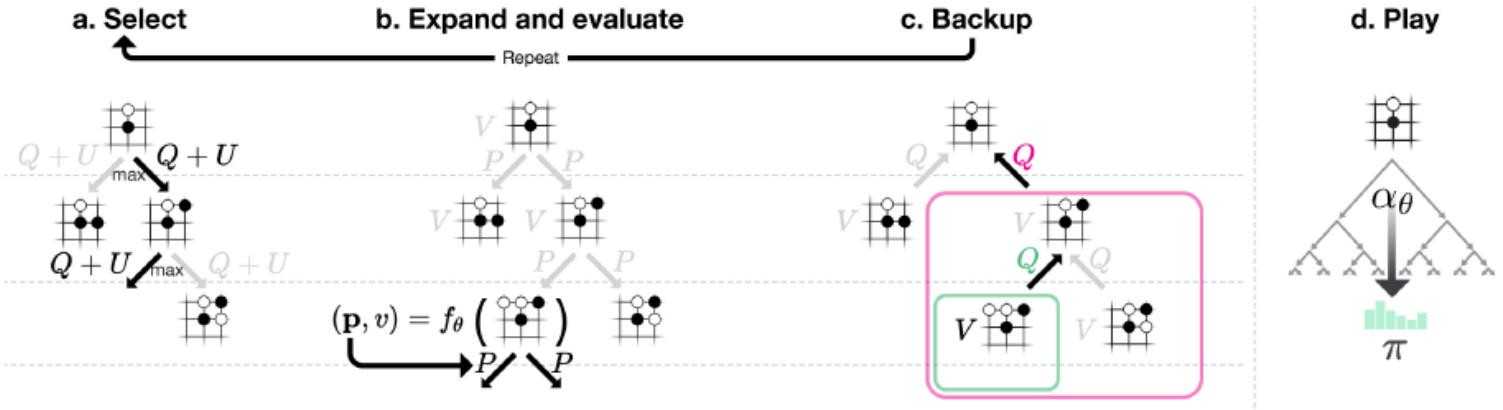
- **Without** using **human knowledge** to train the DQN, only by self-play.
- The DQN combines both the roles of the policy and value networks into a single architecture.
- The neural network is trained by a self-play reinforcement learning algorithm that uses MCTS to play each move.
  - MCTS is a way of *policy improvement*.
  - Self-play is *policy evaluation*.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... & Chen, Y. (2017). Mastering the game of go without human knowledge. Nature, 550(7676), 354–359.

`https://deepmind.com/documents/119/agz_unformatted_nature.pdf`

MCTS in AlphaGo Zero (I)

Each edge (s, a) in the search tree stores a prior probability $P(s, a)$, a visit count $N(s, a)$, and an action-value $Q(s, a)$. For each state s, MCTS is executed, guided by the neural network f_Θ.

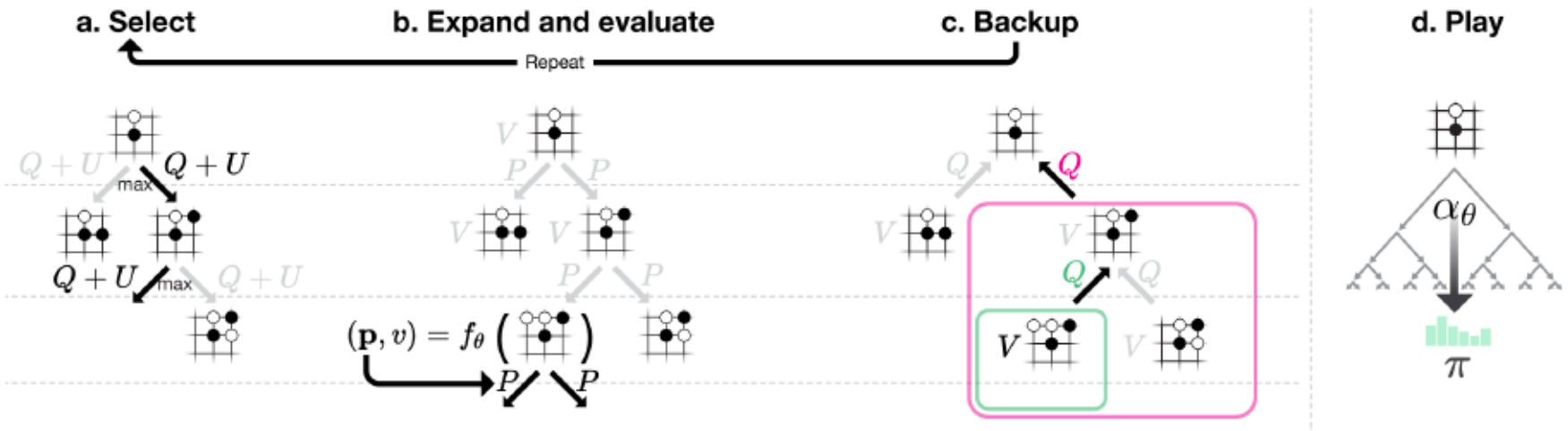


1. Selection: Uses the following UCB equation until finding a leaf node s'.

$$a_t = \arg\max_{a \in A} Q(s, a) + U(s, a); \qquad U(s, a) \propto \frac{P(s, a)}{1 + N(s, a)}$$

2. Expansion: A node is added to the tree and automatically evaluated (i.e. no rollout) by f_Θ, so that $P(s', a)$ and $V(s')$ are the outputs of the network $f_\Theta(s')$.

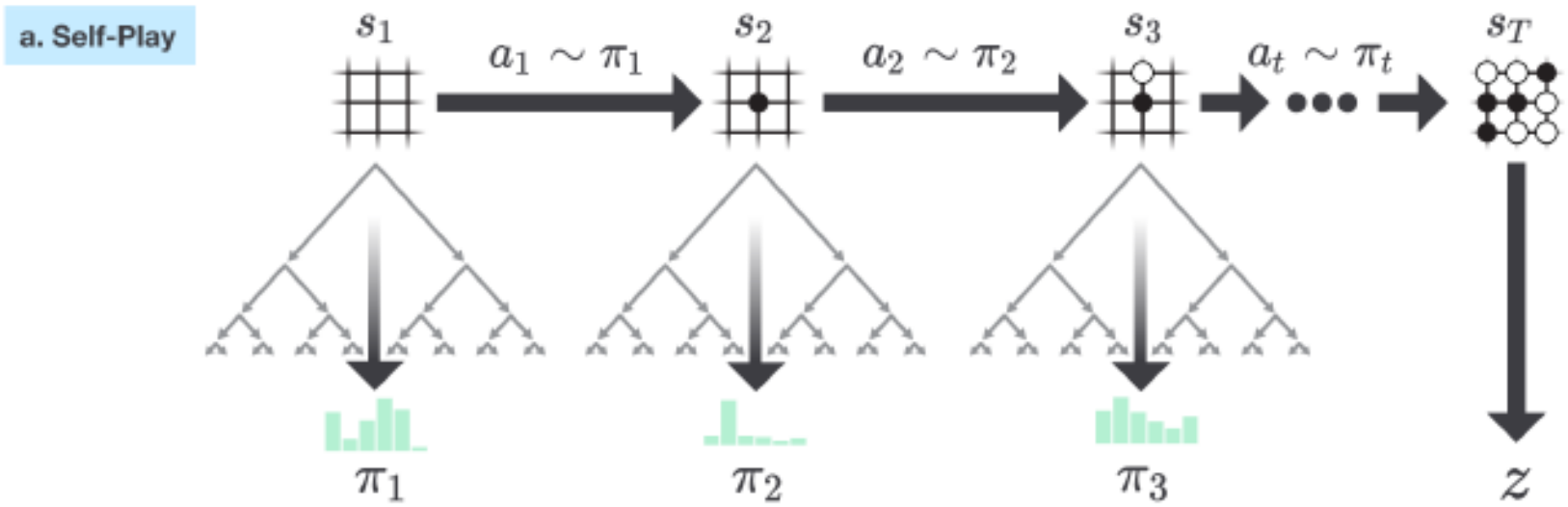
MCTS in AlphaGo Zero (II)



3. Backup: Update $N(s, a)$ and $Q(s, a)$ for each traversed node in the tree (Q running average as in vanilla MCTS).
4. Play: After all iterations have been completed, MCTS recommends a move to play with a policy $\pi = \alpha_\Theta(s)$
   - π_a is proportional to the visit count for each move: $\pi_a \propto N(s, a)^{\frac{1}{\tau}}$, where τ is a temperature parameter.

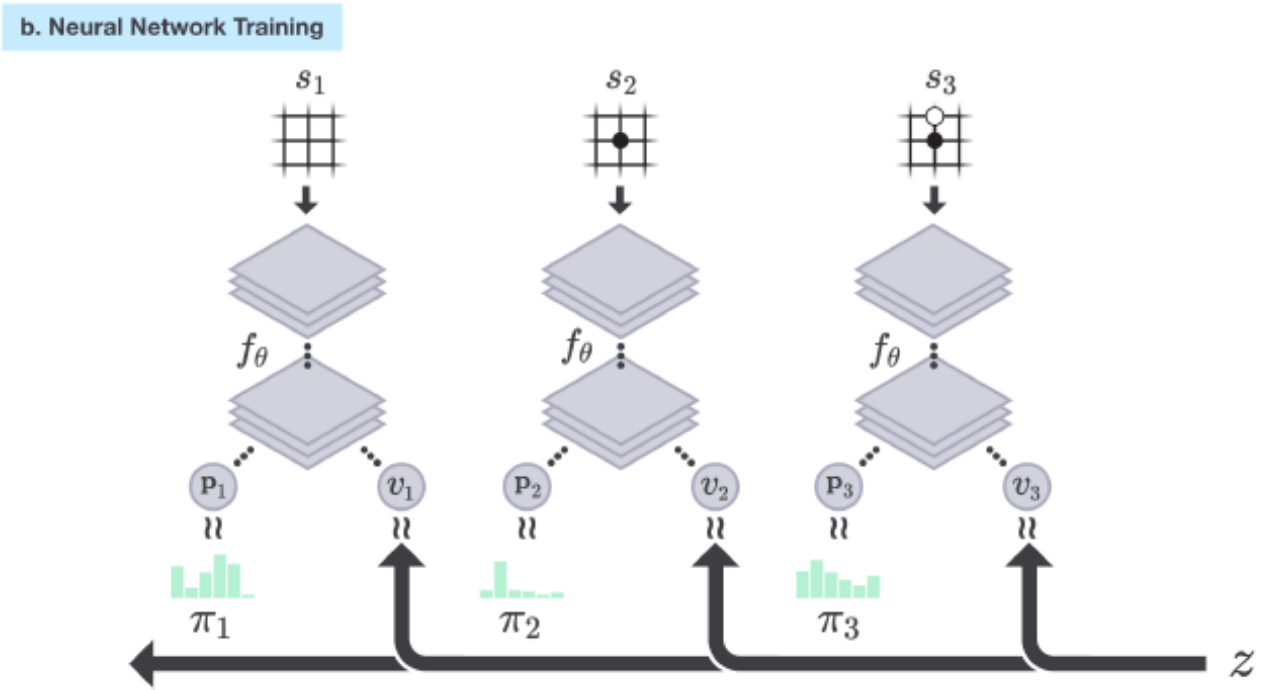
Self-Play with MCTS in AlphaGo Zero

AlphaGo Zero plays against itself using MCTS, until the end of the episode (time T).



- For each state $s_t \in s_1 ... s_T$, MCTS $\alpha_\Theta(s)$ is executed using the latest neural network $f_\Theta(s')$
- Moves a_t are selected according to the probabilities of π_t.
- The game determines the winner of the game (z) given the final state s_T.
  - Both players pass.
  - Search value drops below a resignation threshold.
  - Maximum length.

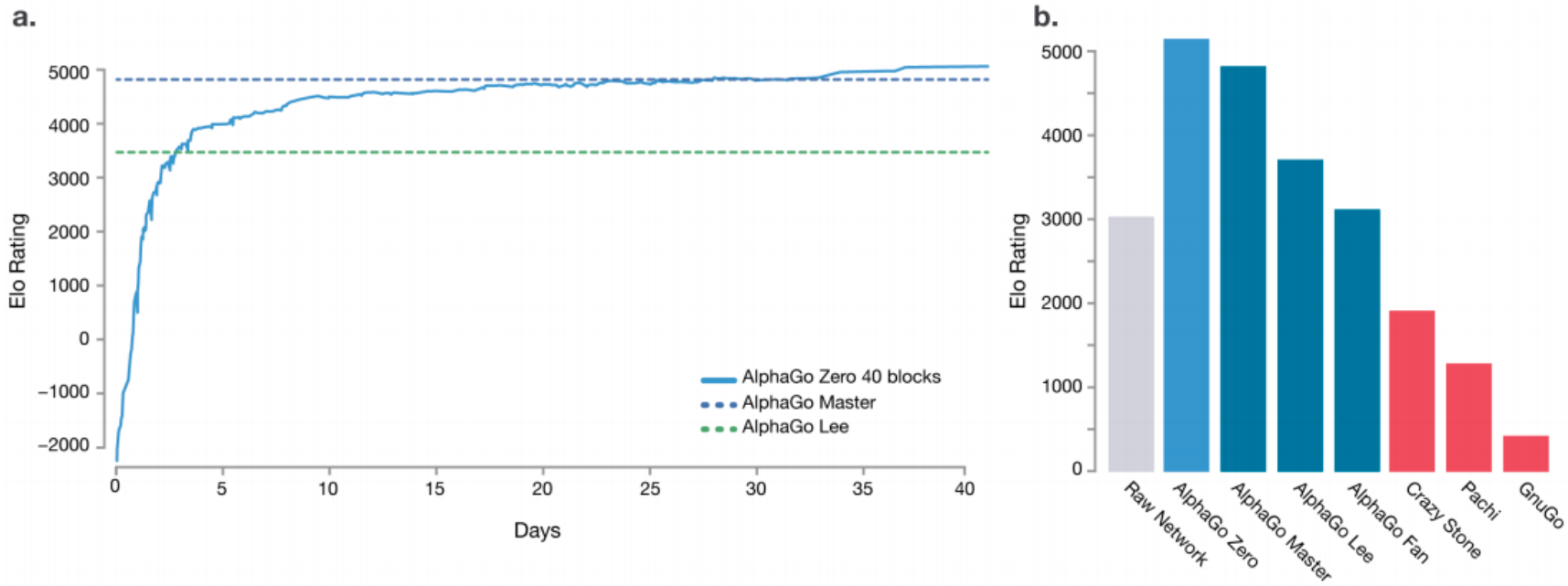
AlphaGo Zero Network Training



z is used to **train** the Neural Network. Given an uniformly sampled set of all tuples (s, π, z) from the last iteration(s) of self-play:

- The neural network takes each s_t as input.
- Data is passed through convolutional layers with parameters Θ.
- Outputs two vectors:
  - p_t: probability distribution over moves.
  - v_t: probability of the current player winning in position s_t.
- Neural Network parameters Θ are optimized so $p_t \approx \pi_t$ and $v_t \approx z$.
- The new parameters are used in the next iteration of self-play.

AlphaGo Zero Results



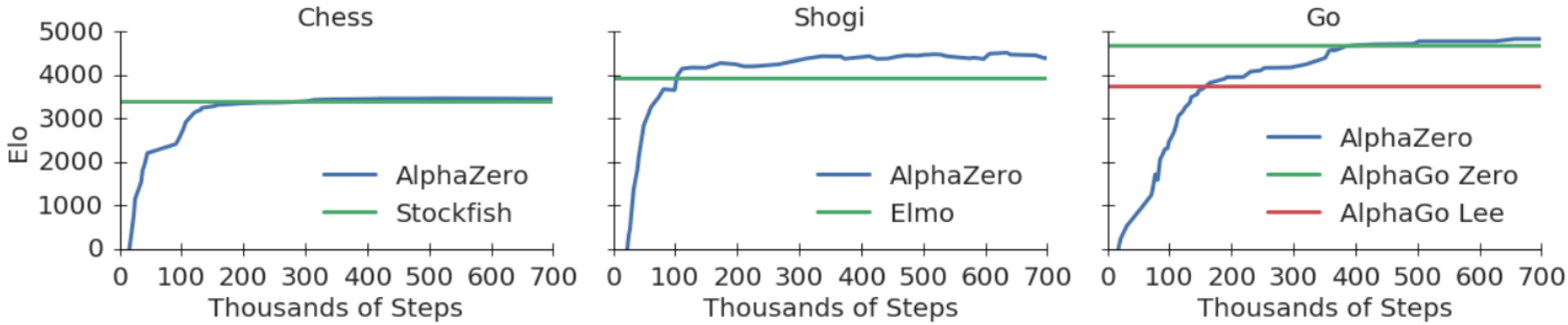
Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... & Chen, Y. (2017). Mastering the game of go without human knowledge. Nature, 550(7676), 354–359.

`https://deepmind.com/documents/119/agz_unformatted_nature.pdf`

More AlphaZero (Dec 2017)

AlphaZero for Go, Chess and Shogi:

- Same architecture, new world champions at Chess and Shogi.



Silver, D., Hubert, T., Schrittwieser, ... & Hassabis, D. (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. arXiv:1712.01815v1, 1–19.

`https://arxiv.org/pdf/1712.01815.pdf`

Deep Learning today

... can be found almost everywhere: computer vision (object classification, detection, segmentation), pose estimation, machine translation, speech recognition, language processing... and game AI!

Game(s)	Method	Network architecture	Input	Output
Atari 2600	DQN [81]	CNN	Pixels	Q-values
	DQN [82]	CNN	Pixels	Q-values
	DRQN [41]	CNN+LSTM	Pixels	Q-values
	UCTtoClassification [33]	CNN	Pixels	Action predictions
	Gorila [84]	CNN	Pixels	Q-values
	Double DQN [130]	CNN	Pixels	Q-values
	Prioritized DQN [104]	CNN	Pixels	Q-values
	Dueling DQN [135]	CNN	Pixels	Q-values
	Bootstrapped DQN [90]	CNN	Pixels	Q-values
	A3C [80]	CNN+LSTM	Pixels	Action probabilities & state-value
	UNREAL (A3C + aux. learning) [50]	CNN+LSTM	Pixels	Act. pr., state value & aux. prediction
	Scalable Evolution Strategies [103]	CNN	Pixels	Policy
	Distributional DQN (C51) [8]	CNN	Pixels	Q-values
	NoisyNet-DQN [28]	CNN	Pixels	Q-values
	NoisyNet-A3C [28]	CNN	Pixels	Action probabilities & state-value
	Rainbow [45]	CNN	Pixels	Q-values
Ms. Pac-Man	HRA [131]	CNN	Pixels (object channels)	Q-values
Montezuma's Revenge	H-DQN [62]	CNN	Pixels	Q-values
	DQN-CTS [7]	CNN	Pixels	Q-values
	DQN-PixelCNN [91]	CNN	Pixels	Q-values
Racing	Direct perception [18]	CNN	Pixels	Affordance indicators
	DDPG [73]	CNN	Pixels	Action probabilities & Q-values
	A3C [80]	CNN+LSTM	Pixels	Action probabilities & state value
Doom	DQN [59]	CNN+pooling	Pixels	Q-values
	A3C + curriculum learning [140]	CNN	Pixels	Action probabilities & state value
	DRQN + aux. learning [64]	CNN+GRU	Pixels	Q-values & aux. predictions
	DQN + SLAM [11]	CNN	Pixels & depth	Q-values
	DFP [22]	CNN	Pixels, features & goals	Feature prediction
Minecraft	H-DRLN [125]	CNN	Pixels	Policy
	RMQN/FRMQN [86]	CNN+LSTM+EM	Pixels	Q-values
	TSCL [77]	CNN+LSTM	Pixels	Action probabilities
StarCraft micromanagement	Zero Order [129]	Feed-forward NN	Local & global features	Q-values
	IQL [26]	CNN+GRU	Local features	Q-values
	BiCNet [95]	Bi-directional RNN	Shared features	Action probabilities & Q-values
	COMA [25]	GRU	Local & global features	Action probabilities & state value
RoboCup Soccer (HFO)	DDPG + Inverting Gradients [42]	Feed-forward NN	Features	Action prob. & power/direction
	DDPG + Mixing policy targets [43]	Feed-forward NN	Features	Action prob. & power/direction
2D billiard	Object-centric prediction [29]	CNN+LSTM	Pixels & forces	Velocity predictions
Text-based games	LSTM-DQN [85]	LSTM+pooling	Text	Q-values

Justesen N, Bontrager P, Togelius J, Risi S. Deep learning for video game playing. arXiv preprint arXiv:1708.07902. 2017 Aug 25.

`https://arxiv.org/pdf/1708.07902.pdf`

Acknowledgements

Most of the materials for this lecture are based on:

- The Deep Learning Book:
  `http://www.deeplearningbook.org/`
- UvA Deep Learning Course:
  `http://uvadlc.github.io/`
- Christopher Olah's tutorials in Neural Networks
  `http://colah.github.io/`
- TensorFlow and DeepLearning4J Tutorials in DNN
  `https://www.tensorflow.org/versions/r0.11/tutorials/index.html`
  `https://deeplearning4j.org/tutorials`