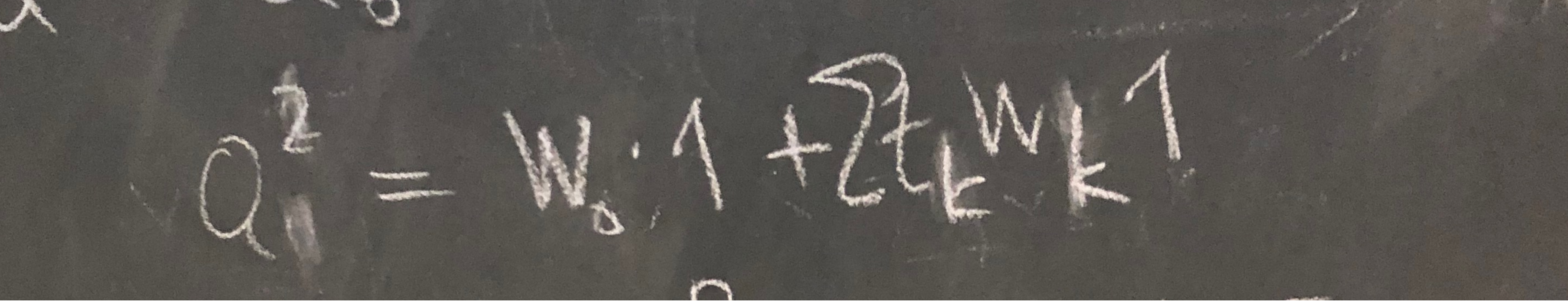
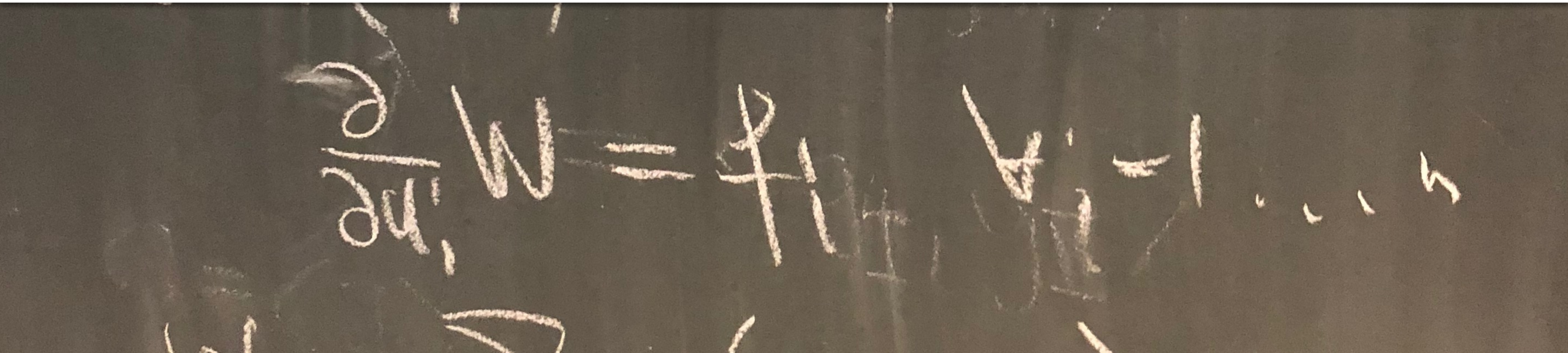


Mathematics of AlphaGo

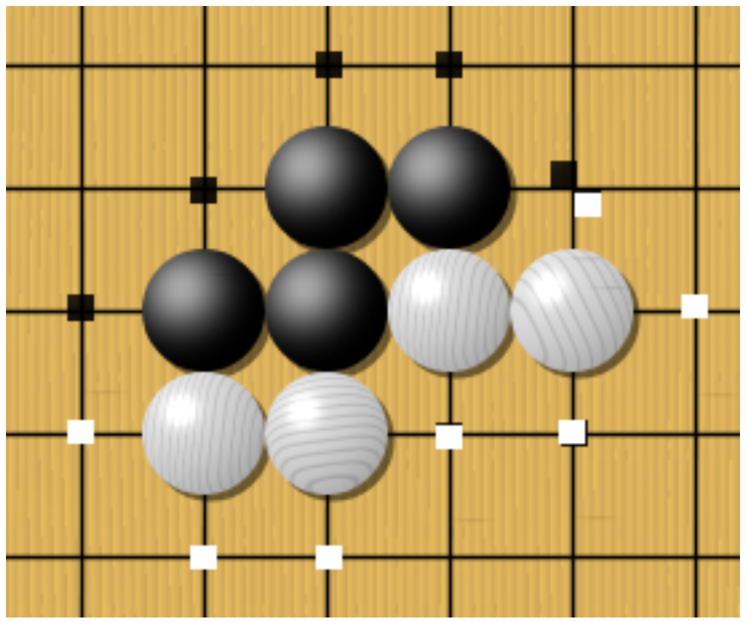
Daniel Murfet

therisingsea.org



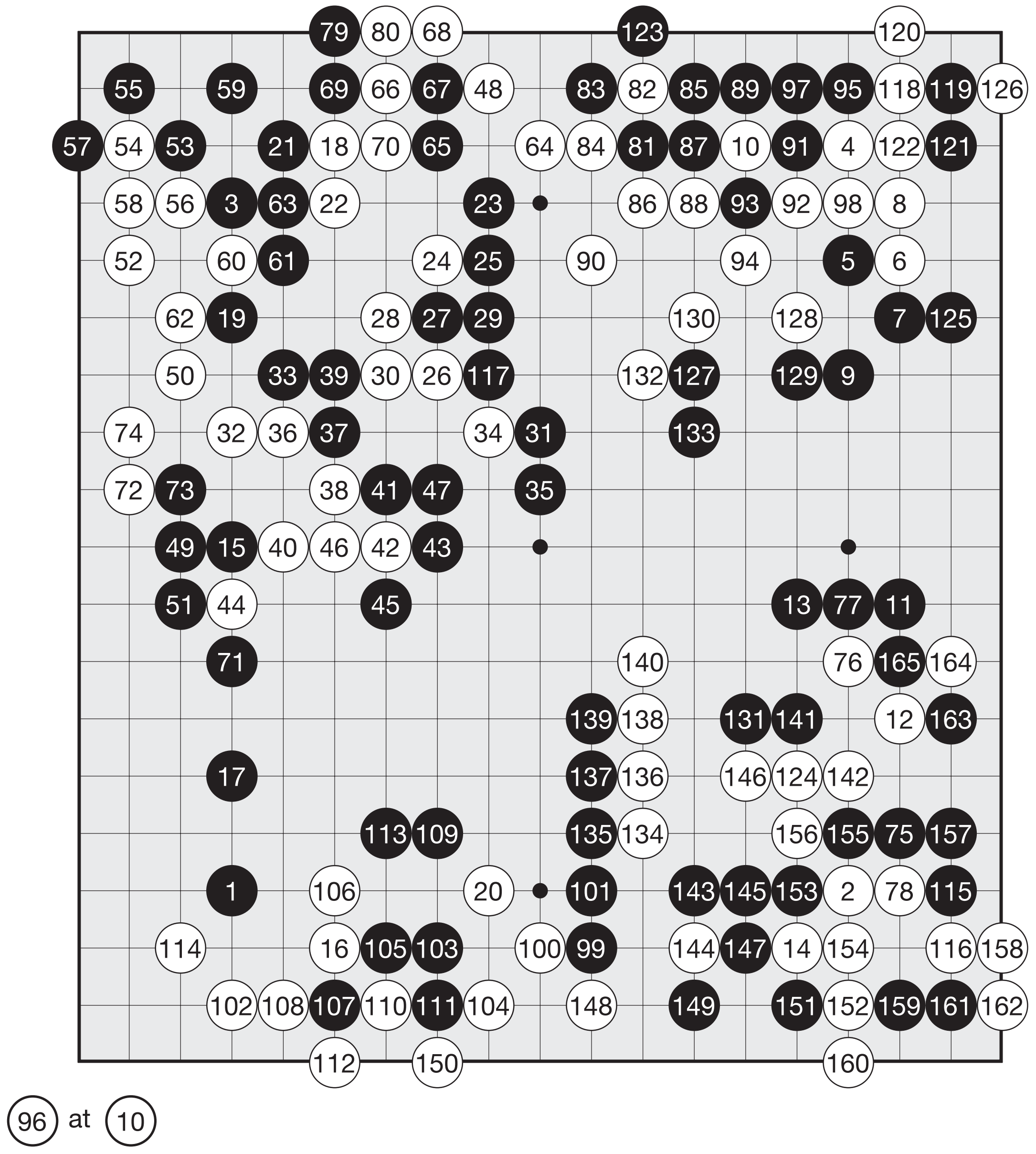
What is Go?

- Appears in Analects of Confucius (3rd century BC)
- Simple rules, complex emergent gameplay



https://upload.wikimedia.org/wikipedia/commons/9/92/Go_adjacent_stones.png

Game 4
AlphaGo (Black), Fan Hui (White)
AlphaGo wins by resignation



D. Silver et al, "Mastering the game of Go with deep neural networks and tree search", Nature 2016.

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.

[1]Google DeepMind, 5 New Street Square, London EC4A 3TW, UK. [2]Google, 1600 Amphitheatre Parkway, Mountain View, California 94043, USA.
*These authors contributed equally to this work.

What is AlphaGo?

- A program that plays Go
- AlphaGo > Lee Sedol (best of five games, March 2016).
- AlphaGo > Ke Jie (best of three games, May 2017).
- AlphaGo, AlphaGo Zero, AlphaZero
- Netflix documentary "**AlphaGo**"

Why is AlphaGo interesting?

- Most of the information in AlphaGo is in real-valued weights which are *learned* not *written.*
- Computer science is becoming a natural science (Norvig).
- This natural science is a new muse for mathematics.
- **Question**: why does deep learning work?
- **Question**: what kind of mathematical structure does this class of programs form?

Into the details…

- AlphaGo is a **neural network** which outputs, given a board configuration, a distribution over moves (policy) and an evaluation of the given board (value).

- This neural network is a function parametrised by a vector of real numbers, trained by **reinforcement learning**: a win increases the probability of contributing moves.

- This training process approximates the iteration towards the unique fixed point of a contraction mapping, defined on the complete metric space of value functions for any **alternating Markov game**.

Neural networks

Feedforward ReLU neural networks

Definition. A *feedforward ReLU neural network* $\mathcal{N}$ with input width n, output width m and depth k consists of

- A sequence of integer *widths* $n = d_1, d_2, \ldots, d_{k+1} = m$,
- A sequence of affine functions $\{A_i : \mathbb{R}^{d_i} \longrightarrow \mathbb{R}^{d_{i+1}}\}_{i=1}^{k}$.

Associated to every ReLU net $\mathcal{N}$ is a continuous function

$$f_{\mathcal{N}} : \mathbb{R}^n \longrightarrow \mathbb{R}^m \,,$$
$$f_{\mathcal{N}} = A_k \circ \mathrm{ReLU} \circ \cdots \circ \mathrm{ReLU} \circ A_1$$

where the *rectified linear unit* (ReLU) is the function

$$\mathrm{ReLU} : \mathbb{R}^d \longrightarrow \mathbb{R}^d \,,$$
$$(x_1, \ldots, x_d) \longmapsto (\max\{x_1, 0\}, \cdots, \max\{x_d, 0\}) \,.$$

ReLU networks are function approximators

Theorem (Stone-Weierstrass). *If $K \subseteq \mathbb{R}^n$ is compact, $f : K \longrightarrow \mathbb{R}^m$ is continuous and $\varepsilon > 0$, there exists a polynomial function $g : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ such that*

$$\sup_{x \in K} \|f(x) - g(x)\| < \varepsilon\,.$$

ReLU networks are function approximators

Theorem (Stone-Weierstrass). *If $K \subseteq \mathbb{R}^n$ is compact, $f : K \longrightarrow \mathbb{R}^m$ is continuous and $\varepsilon > 0$, there exists a polynomial function $g : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ such that*

$$\sup_{x \in K} \|f(x) - g(x)\| < \varepsilon\,.$$

Theorem (Hanin-Sellke '17). *If $K \subseteq \mathbb{R}^n$ is compact, $f : K \longrightarrow \mathbb{R}^m$ is continuous and $\varepsilon > 0$, there exists a ReLU net $\mathcal{N}$ with widths bounded above by $m + n$ such that*

$$\sup_{x \in K} \|f(x) - f_{\mathcal{N}}(x)\| < \varepsilon\,.$$

This is achievable with depth $O(\operatorname{diam}(K)/\omega_f^{-1}(\varepsilon))^{n+1}$.

$$\omega_f^{-1}(\varepsilon) = \sup\{\delta > 0 \,|\, \|x - y\| \leq \delta \Rightarrow \|f(x) - f(y)\| \leq \varepsilon\}$$

$$\boxed{\overline{\mathrm{ReLU}} = \overline{\mathrm{Poly}} = \mathrm{Cts}(K, \mathbb{R}^m)}$$

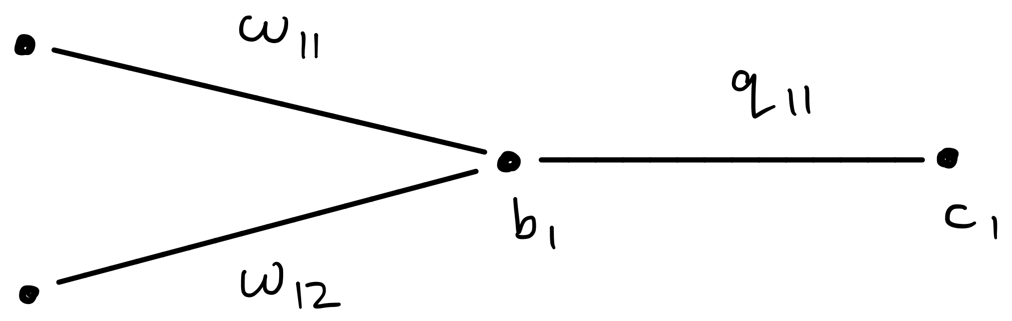
Examples of ReLU networks

$$\text{ReLU net } \mathcal{N} = \{A_i : \mathbb{R}^{d_i} \longrightarrow \mathbb{R}^{d_{i+1}}\}_{i=1}^{k} \in \prod_{i=1}^{k} M_{d_{i+1},d_i}(\mathbb{R}) \times \prod_{i=1}^{k} \mathbb{R}^{d_{i+1}}$$

$$f_{\mathcal{N}} : \mathbb{R}^n \longrightarrow \mathbb{R}^m \,,$$
$$f_{\mathcal{N}} = A_k \circ \mathrm{ReLU} \circ \cdots \circ \mathrm{ReLU} \circ A_1$$

$n = 2, m = 1, k = 2, d_2 = 1$

$$\mathbb{R}^2 \xrightarrow{A_1} \mathbb{R} \xrightarrow{A_2} \mathbb{R}$$

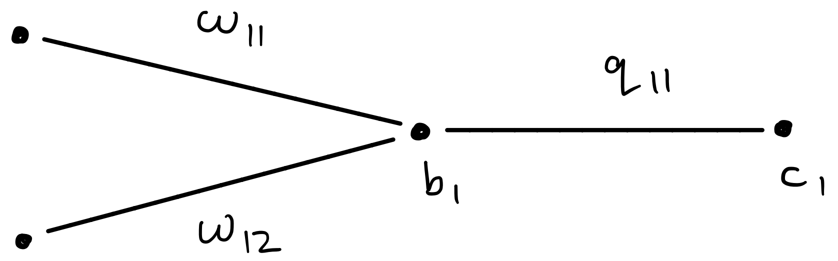


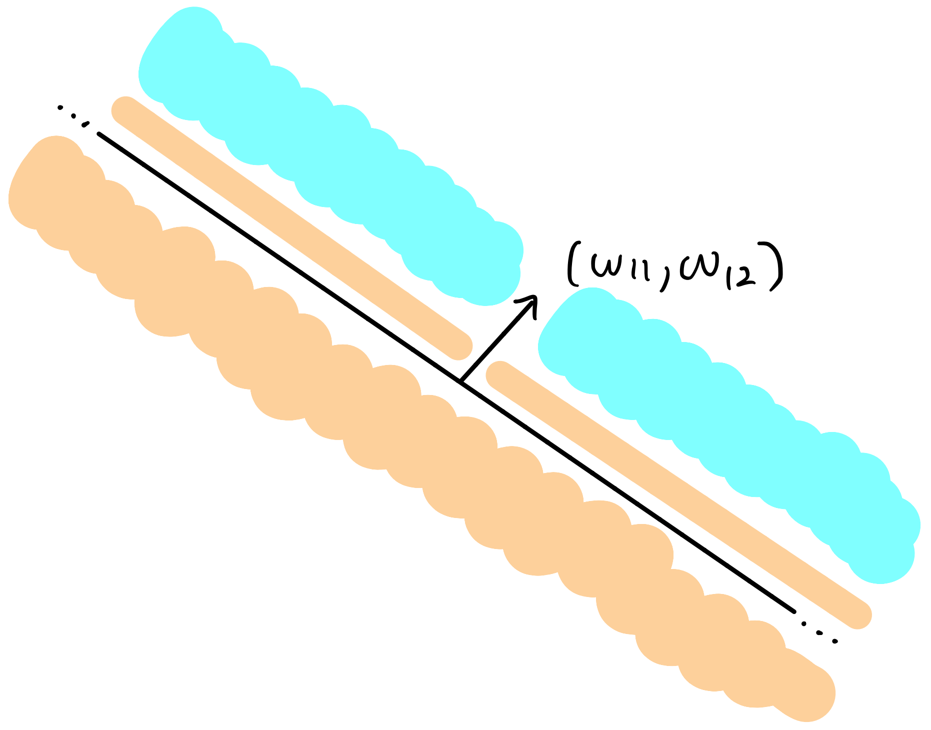
$$f_{\mathcal{N}}(x_1, x_2) = q_{11}\,\mathrm{ReLU}(w_{11}x_1 + w_{12}x_2 + b_1) + c_1$$

Examples of ReLU networks

$n = 2, m = 1, k = 2, d_2 = 1$

$$\mathbb{R}^2 \xrightarrow{A_1} \mathbb{R} \xrightarrow{A_2} \mathbb{R}$$





$c_1 < 0$
$q_{11} > 0$

🟦 >0

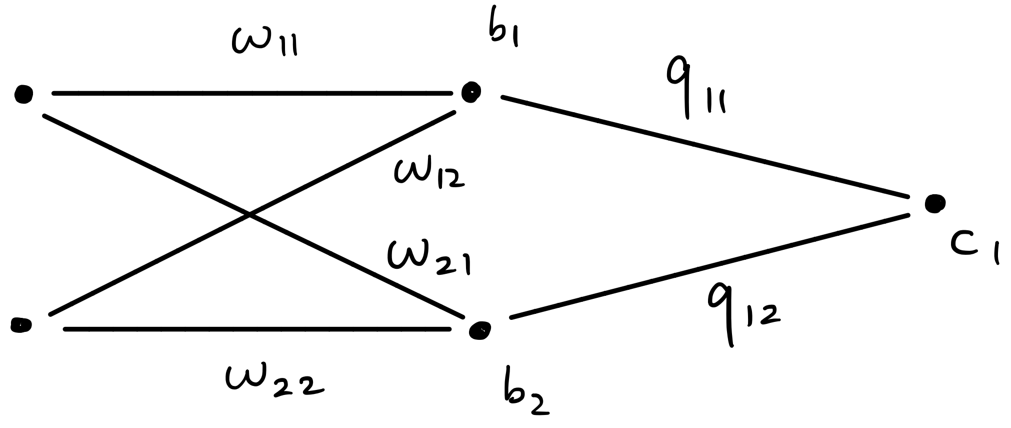
🟧 <0

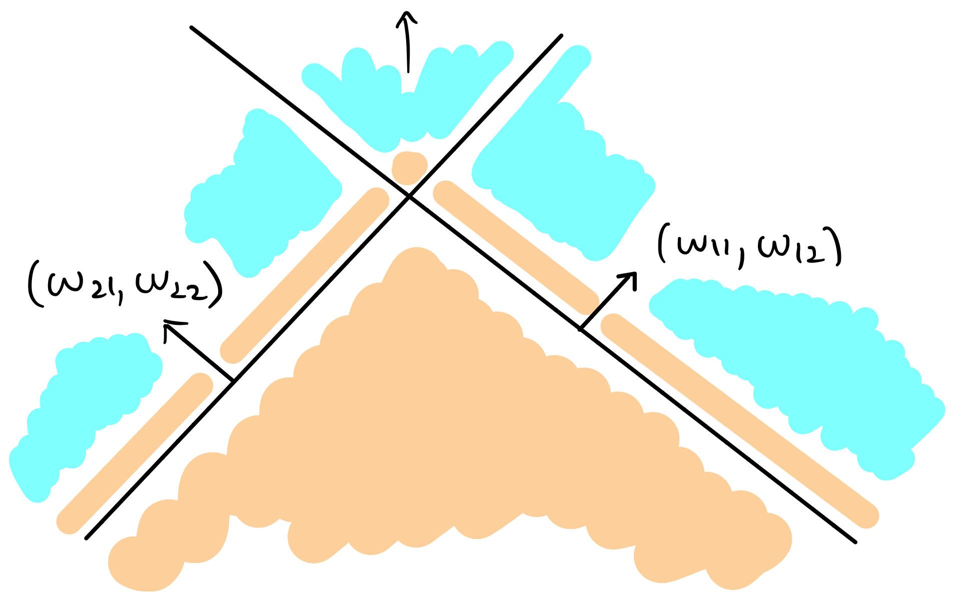
$$f_{\mathcal{N}}(x_1, x_2) = q_{11}\,\mathrm{ReLU}(w_{11}x_1 + w_{12}x_2 + b_1) + c_1$$

Examples of ReLU networks

$n = 2, m = 1, k = 2, d_2 = 2$

$$\mathbb{R}^2 \xrightarrow{A_1} \mathbb{R}^2 \xrightarrow{A_2} \mathbb{R}$$





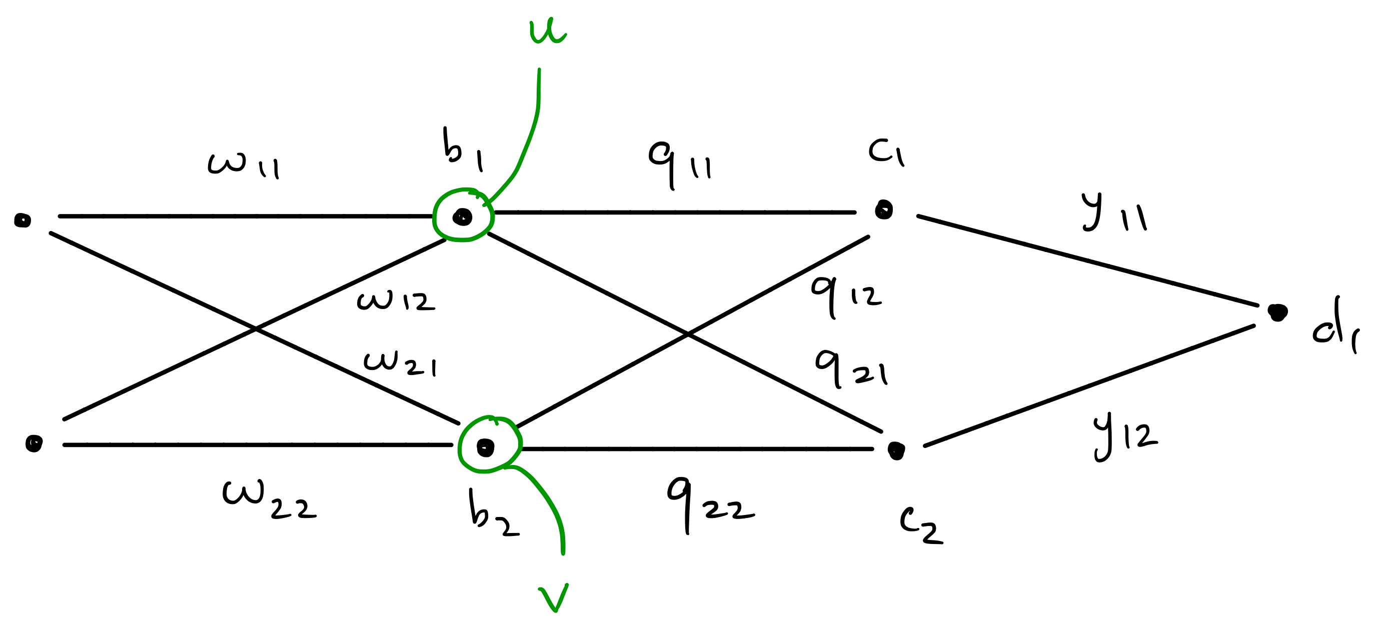
$c_1 < 0$
$q_{11} > 0$
$q_{12} > 0$

🟦 >0
🟧 <0

$$f_{\mathcal{N}}(x_1, x_2) = q_{11}\,\mathrm{ReLU}(w_{11}x_1 + w_{12}x_2 + b_1)$$
$$+ q_{12}\,\mathrm{ReLU}(w_{21}x_1 + w_{22}x_2 + b_2) + c_1$$

Examples of ReLU networks

$n = 2, m = 1, k = 3, d_2 = 2, d_3 = 2$

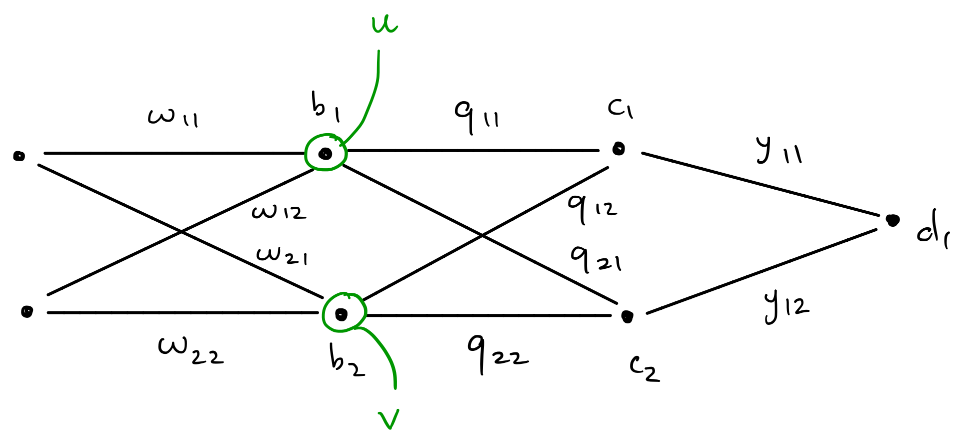


$$u = w_{11}x_1 + w_{12}x_2 + b_1 \qquad \bar{u} = \mathrm{ReLU}(u)$$

$$v = w_{21}x_1 + w_{22}x_2 + b_2 \qquad \bar{v} = \mathrm{ReLU}(v)$$

$$\begin{aligned} f_{\mathcal{N}}(x_1, x_2) = {} & y_{11}\,\mathrm{ReLU}(q_{11}\bar{u} + q_{12}\bar{v} + c_1) \\ & + y_{12}\,\mathrm{ReLU}(q_{21}\bar{u} + q_{22}\bar{v} + c_2) + d_1 \end{aligned}$$

$n = 2, m = 1, k = 3, d_2 = 2, d_3 = 2$



$$f_{\mathcal{N}}(x_1, x_2) = y_{11}\,\mathrm{ReLU}(q_{11}\bar{u} + q_{12}\bar{v} + c_1) + y_{12}\,\mathrm{ReLU}(q_{21}\bar{u} + q_{22}\bar{v} + c_2) + d_1$$

$u = w_{11}x_1 + w_{12}x_2 + b_1$

$v = w_{21}x_1 + w_{22}x_2 + b_2$

$\bar{u} = \mathrm{ReLU}(u)$

$\bar{v} = \mathrm{ReLU}(v)$

$c_1, c_2 < 0$

$d_1 > 0$

$q_{12}, q_{22} > 0$

$y_{11} < 0$

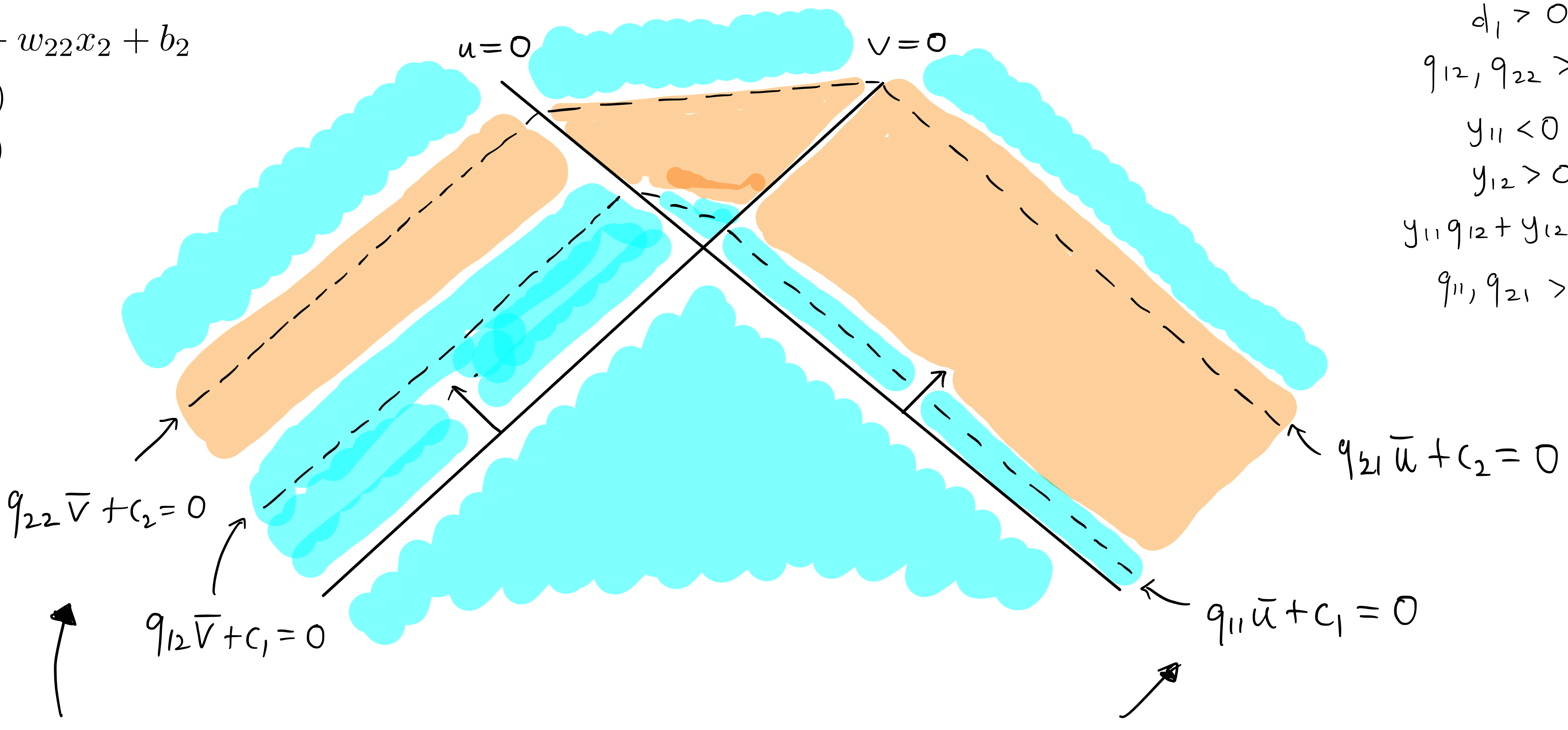
$y_{12} > 0$

$y_{11}q_{12} + y_{12}q_{22} > 0$

$q_{11}, q_{21} > 0$

🟦 >0

🟧 <0



$u=0$
$v=0$
$q_{21}\bar{u} + c_2 = 0$
$q_{22}\bar{v} + c_2 = 0$
$q_{12}\bar{v} + c_1 = 0$
$q_{11}\bar{u} + c_1 = 0$

$$f_{\mathcal{N}} = y_{11}\,\mathrm{ReLU}(q_{12}\bar{v} + c_1) + y_{12}\,\mathrm{ReLU}(q_{22}\bar{v} + c_2) + d_1$$

$$f_{\mathcal{N}} = y_{11}\,\mathrm{ReLU}(q_{11}\bar{u} + c_1) + y_{12}\,\mathrm{ReLU}(q_{21}\bar{u} + c_2) + d_1$$

AlphaGo Zero architecture

- AlphaGo uses a special kind of feedforward ReLU network called a *convolutional neural network* developed for computer vision, with constraints on the weights.
- Skip connections and batch normalisation (“residual network”) following state of the art models for computer vision tasks.
- **Input dimension:** 19x19x17 = 6137
- **Output dimension:** 19x19 (positions) + 1 (pass) + 1 (value) = 363
- **Hidden layer width**: maximum 19x19x256 = 92416
- **Network depth:** 42

Reinforcement learning (RL)

Markov Decision Processes

Definition. A *finite Markov Decision Process* (MDP) is a finite set $\mathcal{S}$ of *states*, a finte set $\mathcal{A}$ of *actions*, for each $s \in \mathcal{S}$ a subset $\mathcal{A}(s) \subseteq \mathcal{A}$ of *allowed actions* in state s, a *reward function* $R : \mathcal{S} \longrightarrow \mathbb{R}$, for each pair $s \in \mathcal{S}, a \in \mathcal{A}$ a probability distribution $P(s'|s,a)$ over states $s' \in \mathcal{S}$, and a *discount factor* $\gamma \in (0,1)$.

Definition. A *policy* is a function $\pi : \mathcal{S} \longrightarrow \Delta\mathcal{A}$ such that $\pi(a|s) = 0$ if $a \notin \mathcal{A}(s)$.

Definition. The *space of policies* $\mathcal{P}$ is the set of policies with metric

$$d_\infty(\pi, \pi') = \sup_{s \in \mathcal{S}} \sup_{a \in \mathcal{A}} |\pi(a|s) - \pi'(a|s)| \,.$$

The *discounted reward* of a sequence of states $s_0, s_1, s_2, \ldots$ is

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots$$

The goal: find a policy which maximises the expected discounted reward.

Markov Decision Processes

Definition. A *value function* is a function $v : \mathcal{S} \longrightarrow \mathbb{R}$.

Definition. The *space of value functions* $\mathcal{V}$ is the set of value functions with metric

$$d_\infty(v, v') = \sup_{s \in \mathcal{S}} |v(s) - v'(s)| \,.$$

Given a policy $\pi : \mathcal{S} \longrightarrow \Delta\mathcal{A}$ the *value* of a state $s \in \mathcal{S}$ to an agent following π is the time-discounted expected value of future rewards for trajectories starting at s

$$v_\pi(s) = \mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t R(s_t) | s_0 = s \Big] \,.$$

Theorem. *The function $\Phi_\pi : \mathcal{V} \longrightarrow \mathcal{V}$ defined by*

$$\Phi_\pi(v)(s) = R(s) + \sum_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \gamma \pi(a|s) P(s'|s,a) v(s')$$

has a unique fixed point v_π.

Policies versus value functions

$$\begin{tikzcd}\mathcal{P} \arrow[r, "\pi \longmapsto v_\pi"] & \mathcal{V}\end{tikzcd}$$

“policy evaluation”

$$v_\pi = \mathrm{fix}(\Phi_\pi)$$

$$\begin{tikzcd}\mathcal{P} & \mathcal{V} \arrow[l, "\pi_v \longleftarrow\!\!| \, v"']\end{tikzcd}$$

“greedy policy”

$$\pi_v(s) = \mathrm{argmax}_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} P(s'|s,a)v(s')$$

Optimal policies and value functions

Definition. A policy π is *optimal* if $v_\pi \geq v_\rho$ for all policies ρ.

Theorem (Bellman). *The function $\Phi : \mathcal{V} \longrightarrow \mathcal{V}$ defined by*

$$\Phi(v)(s) = R(s) + \sup_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \gamma P(s'|s,a) v(s')$$ **(Bellman operator)**

has a unique fixed point v^* and

- *for every policy $\pi \in \mathcal{P}$ we have $v^* \geq v_\pi$,*
- *the policy $\pi^* = \pi_{v^*}$ is optimal and $v_{\pi^*} = v^*$.*

- Iterating the Bellman operator exactly is usually infeasible.

- **Reinforcement learning**: the study of algorithms approximating this fixed point iteration, for classes of MDPs that we care about.

Alternating Markov games

Definition. An *alternating Markov Game* is an MDP together with a decomposition of the state space $\mathcal{S} = \mathcal{S}_0 \amalg \mathcal{S}_1$ (we write $|s| = 0$ if $s \in \mathcal{S}_0$ and $|s| = 1$ if $s \in \mathcal{S}_1$ and call these *even* or *odd* states) such that the only possible transitions change the parity:

$$P(s'|s, a) = 0 \text{ if } |s'| = |s| \,.$$

We think of a policy $\pi : \mathcal{S} \longrightarrow \Delta\mathcal{A}$ as a pair of policies $\pi_i = \pi|_{\mathcal{S}_i}$, one for the *even player* and one for the *odd player*, and $R(s)$ is the reward obtained by the even player in state s (the odd player receives $-R(s)$, so this is a zero-sum game).

Go as an alternating Markov game

Definition. The alternating Markov Game Go has

$$\mathcal{S} = \left(\{empty, white, black\}^{\{1,\dots,19\}^2}\right)^8 \times \{0,1\}$$

where the even player is black. A state

$$s = (s_0, s_{-1}, \dots, s_{-7}, p)$$

is even if $p = 0$ and odd otherwise. The set of actions $\mathcal{A}(s)$ is the set of allowed moves of player p from board configuration s_0 (taking into account the ko rule using s_{-1}). The reward $R(s)$ is only nonzero if s_0 is a terminal configuration and in that case

$$R(s) = \begin{cases} +1 & \text{black wins} \\ -1 & \text{white wins} \end{cases}.$$

Policies versus value functions

$$\begin{tikzcd} \mathcal{P} \arrow[r, "\pi \longmapsto v_\pi"] & \mathcal{V} \end{tikzcd}$$

"policy evaluation"

$$v_\pi = \mathrm{fix}(\Phi_\pi)$$

$$\begin{tikzcd} \mathcal{P} & \mathcal{V} \arrow[l, "\pi_v^\pm \longleftarrow v"'] \end{tikzcd}$$

"graded greedy policy"

$$\pi_v^\pm(s) = \mathrm{argmax}_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} P(s'|s,a)(-1)^{|s|} v(s')$$

Theorem. *Given a Markov Game the function $\Phi^{\pm} : \mathcal{V} \longrightarrow \mathcal{V}$ defined by*

$$\Phi^{\pm}(v)(s) = R(s) + (-1)^{|s|} \sup_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \gamma P(s'|s,a)(-1)^{|s|} v(s')$$

(graded Bellman operator)

has a unique fixed point v^*, which is the unique Nash equilibrium. That is, writing $\pi^* = \pi^{\pm}_{v^*}$ and $\pi^*_i = \pi^*|_{\mathcal{S}_i}$, the policy π^*_0 is optimal against π^*_1, the policy π^*_1 is optimal against π^*_0 and any such pair has the value function v^*.

- The fixed point of the graded Bellman operator is a discounted minimax. Once again, it is often impractical to evaluate this exactly (e.g. in Go).

- **Deep reinforcement learning:** approximate the policy and value functions by neural networks and approximate the iteration to this fixed point.

Deep RL for Markov games

$$\begin{tikzcd}
\mathcal{S} = \left(\{empty, white, black\}^{\{1,\dots,19\}^2}\right)^8 \times \{0,1\} \arrow[r, "\langle \pi, v \rangle"] \arrow[d, "\text{inclusion}"'] & \Delta\mathcal{A} \times \mathbb{R} \\
\mathcal{S}' = \left([0,1]^{361} \times [0,1]^{361}\right)^8 \times [0,1]^{361} \arrow[d, "\text{inclusion}"'] & \\
\mathcal{S}'' = \mathbb{R}^{6137} \arrow[r, dashed] & \mathbb{R}^{19\times 19+1} \times \mathbb{R} \arrow[uu, "\text{softmax}"']
\end{tikzcd}$$

AlphaGo neural network

$$\boxed{\begin{gathered} f_\theta : \mathbb{R}^{6137} \longrightarrow \mathbb{R}^{19\times 19+1} \times \mathbb{R} \\ (\pi_\theta(s), v_\theta(s)) := f_\theta(s) \end{gathered}}$$

Deep RL

- **Step 1**: approximate policy and value functions.
- **Step 2**: project the Bellman iteration onto these approximations.
- **Step 3**: ask someone for millions of dollars so you can run your projected Bellman iteration on fancy hardware.

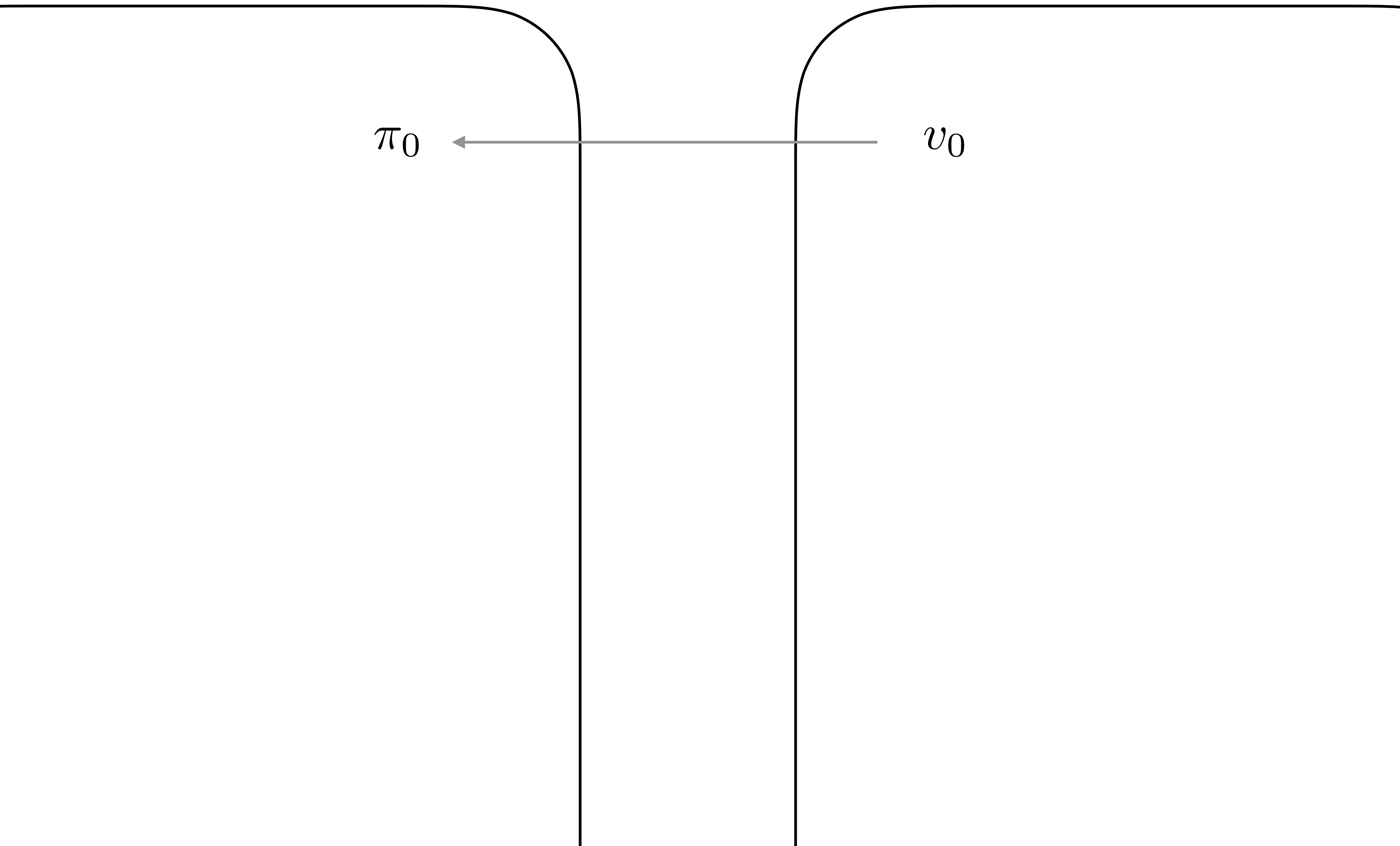
AlphaGo neural network

$$\boxed{\begin{gathered} f_\theta : \mathbb{R}^{6137} \longrightarrow \mathbb{R}^{19\times19+1} \times \mathbb{R} \\ (\pi_\theta(s), v_\theta(s)) := f_\theta(s) \end{gathered}}$$

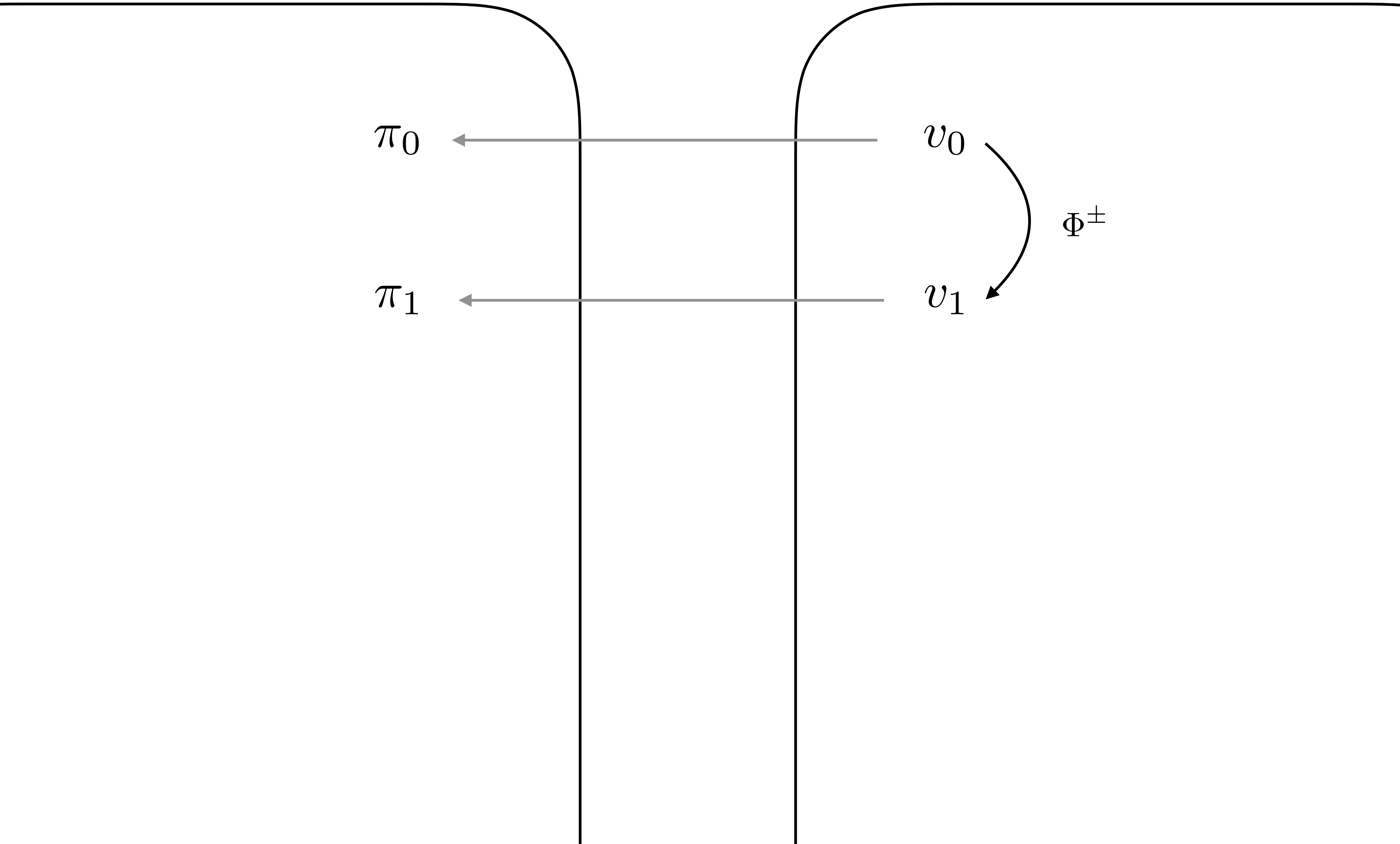
$$\begin{tikzcd}\mathcal{P} \arrow[r, shift left, "V"] & \mathcal{V} \arrow[l, shift left, "P"] \arrow[loop right, "\Phi^{\pm}"]\end{tikzcd}$$

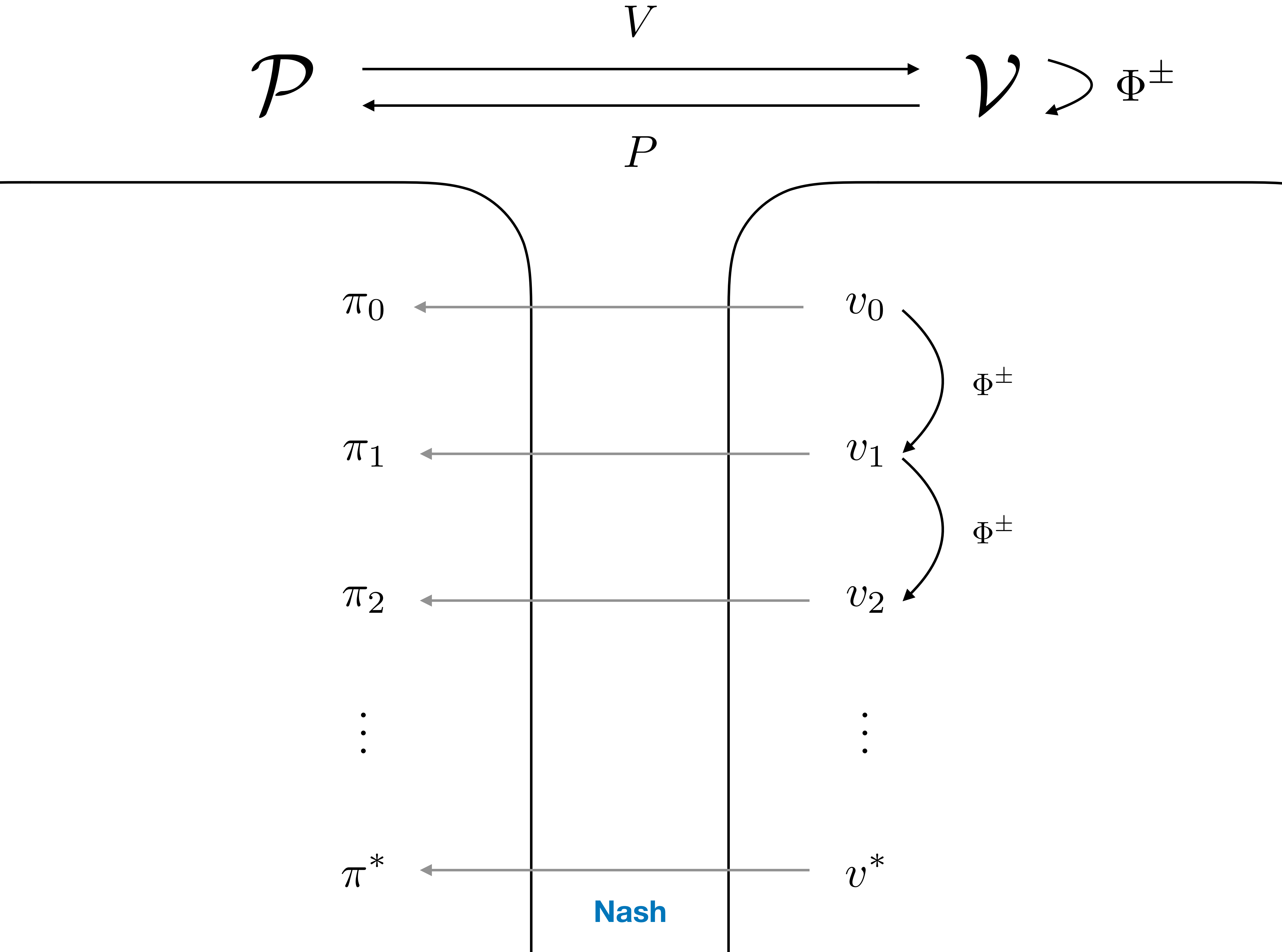
- $\mathcal{P}$ is the space of policies.
- $\mathcal{V}$ is the space of value functions.
- V maps policies to value functions $V(\pi) = v_{\pi}$.
- P maps value functions to policies $P(v) = \pi_v^{\pm}$.
- $\Phi^{\pm}$ is the graded Bellman operator.
- The fixed point v^* of $\Phi^{\pm}$ is the unique Nash equilibrium.

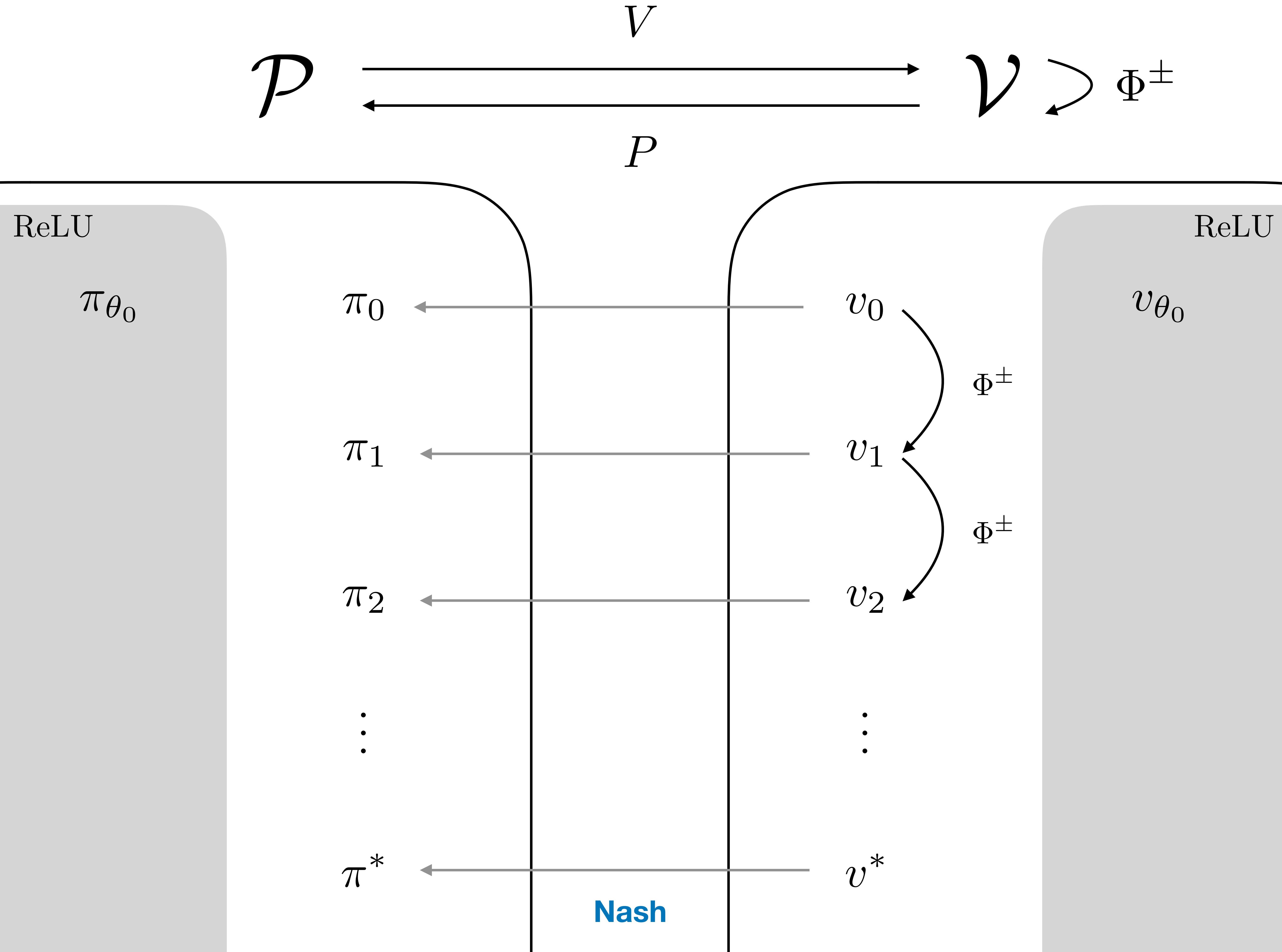
$$\begin{tikzcd}\mathcal{P} \arrow[r, shift left, "V"] & \mathcal{V} \arrow[l, shift left, "P"] \arrow[loop right, "\Phi^{\pm}"]\end{tikzcd}$$

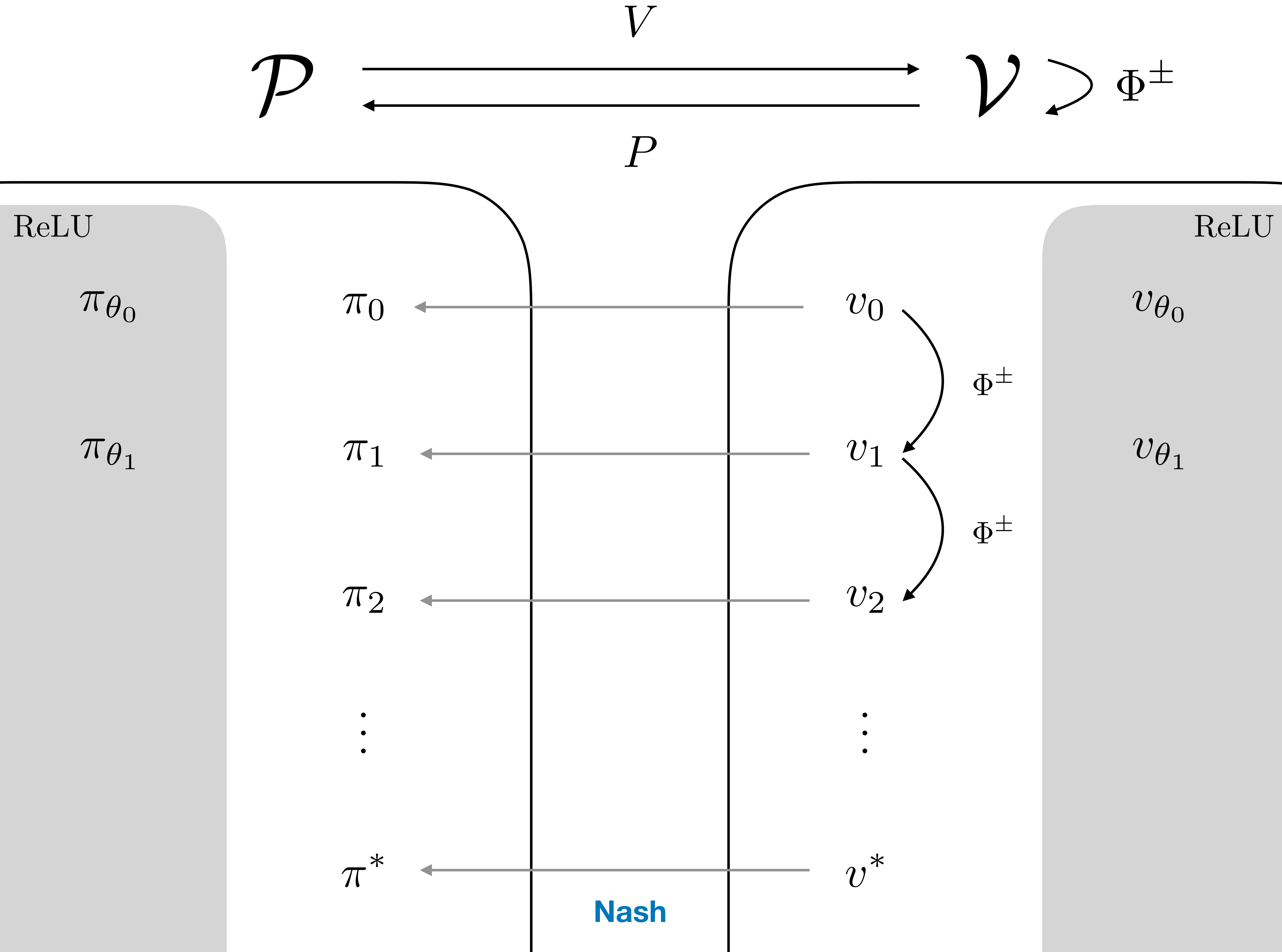


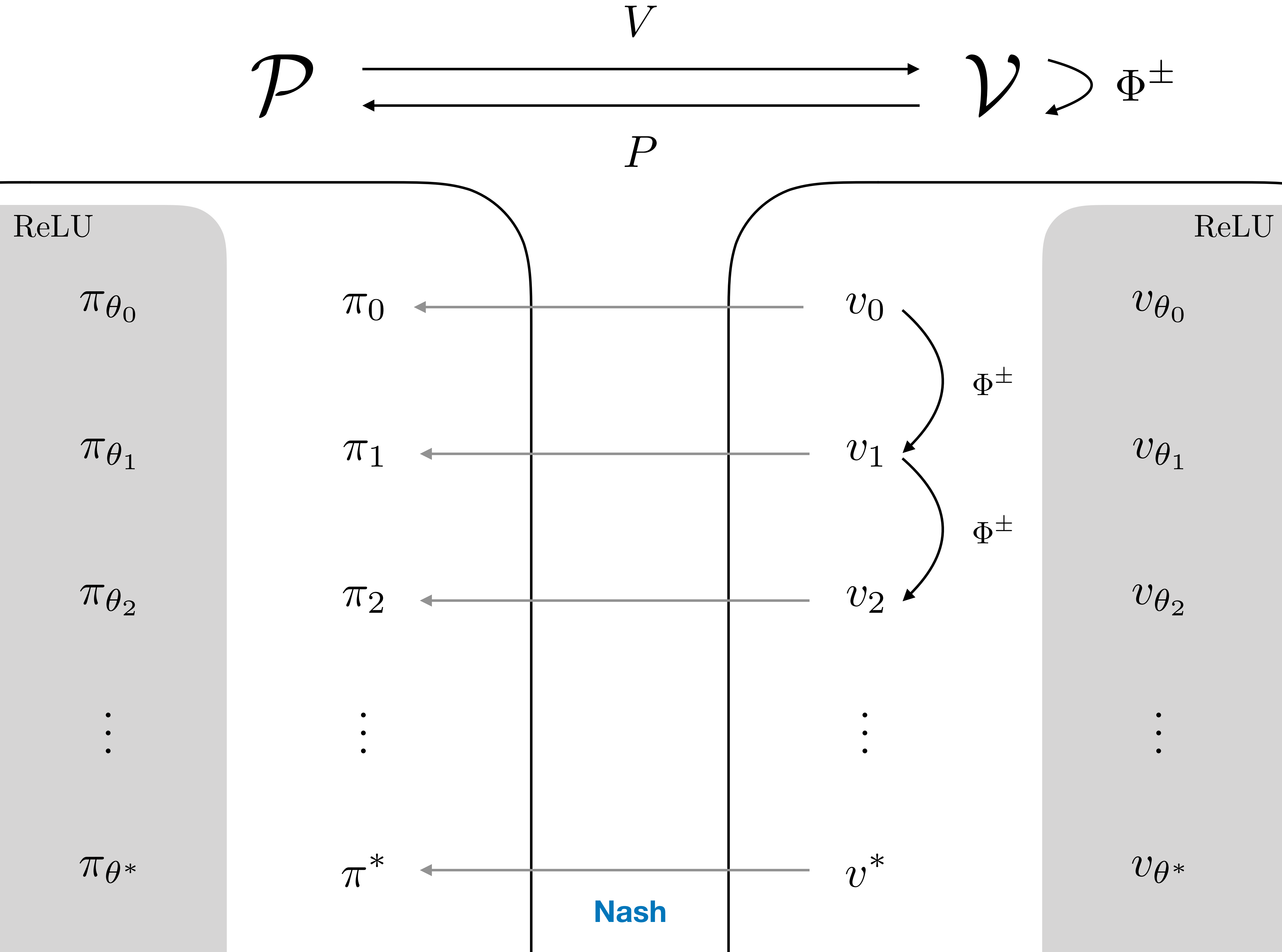
$$\begin{tikzcd}
\mathcal{P} \arrow[r, shift left, "V"] & \mathcal{V} \arrow[l, shift left, "P"] \arrow[loop right, "\Phi^{\pm}"]
\end{tikzcd}$$




$\mathcal{P}$
V
P
$\mathcal{V}$
$\Phi^{\pm}$
π_0
v_0
π_1
v_1
π_2
v_2
π^*
v^*
Nash

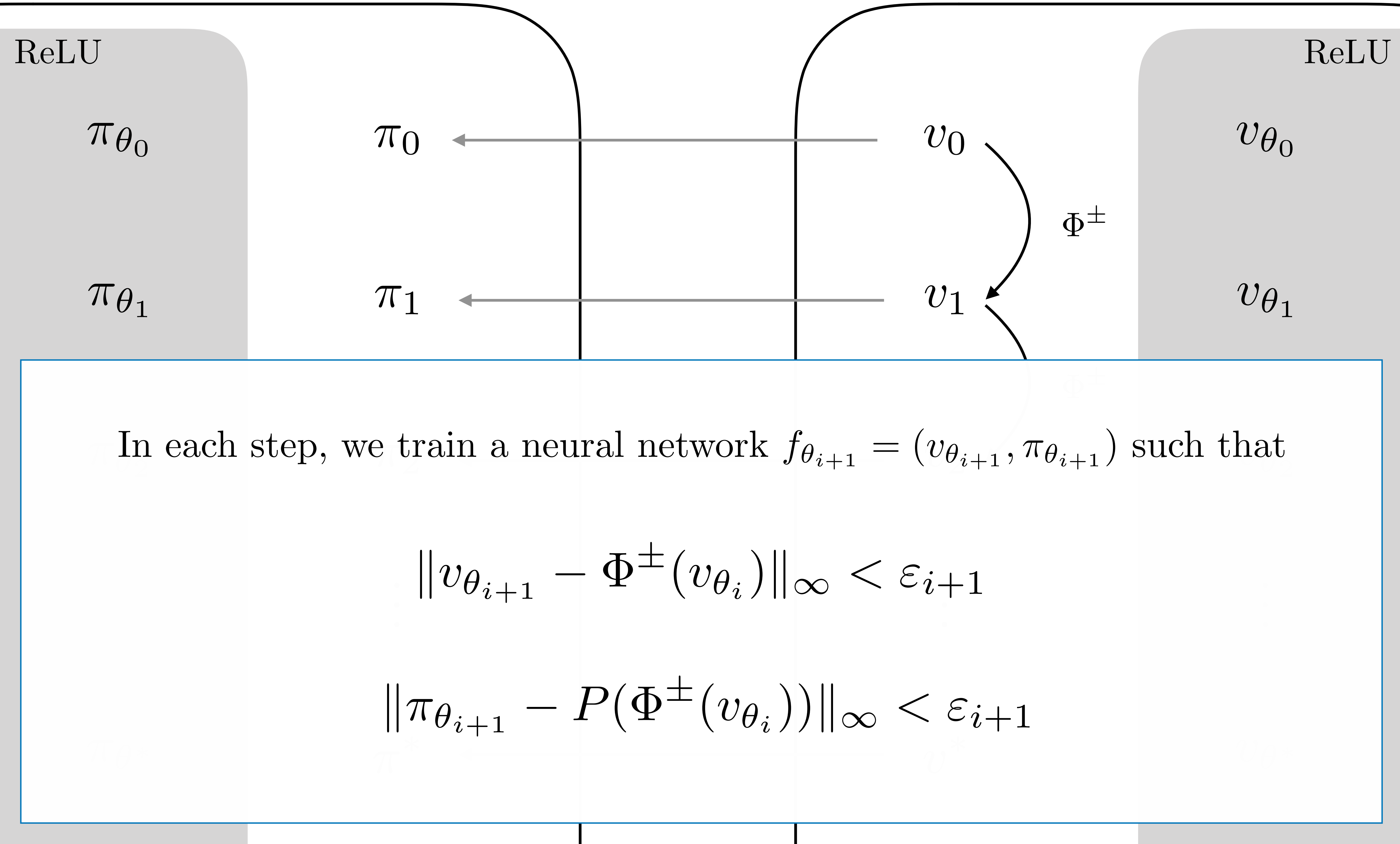

V
$\mathcal{P}$
$\mathcal{V}$
$\Phi^{\pm}$
P
ReLU
ReLU
π_{θ_0}
v_{θ_0}
π_0
v_0
$\Phi^{\pm}$
π_1
v_1
$\Phi^{\pm}$
π_2
v_2
$\vdots$
$\vdots$
π^*
v^*
Nash


$\mathcal{P}$
V
P
$\mathcal{V}$
$\Phi^{\pm}$
ReLU
ReLU
π_{θ_0}
π_{θ_1}
π_0
π_1
π_2
$\vdots$
π^*
v_0
v_1
v_2
$\vdots$
v^*
$\Phi^{\pm}$
$\Phi^{\pm}$
v_{θ_0}
v_{θ_1}
Nash



V
$\mathcal{P}$
$\mathcal{V}$
$\Phi^{\pm}$
P
ReLU
ReLU
π_{θ_0}
π_0
v_0
v_{θ_0}
$\Phi^{\pm}$
π_{θ_1}
π_1
v_1
v_{θ_1}
$\Phi^{\pm}$
π_{θ_2}
π_2
v_2
v_{θ_2}
π_{θ^*}
π^*
v^*
v_{θ^*}
Nash

$$\begin{tikzcd}\mathcal{P} \arrow[r, shift left, "V"] & \mathcal{V} \arrow[l, shift left, "P"] \arrow[loop right, "\Phi^{\pm}"]\end{tikzcd}$$

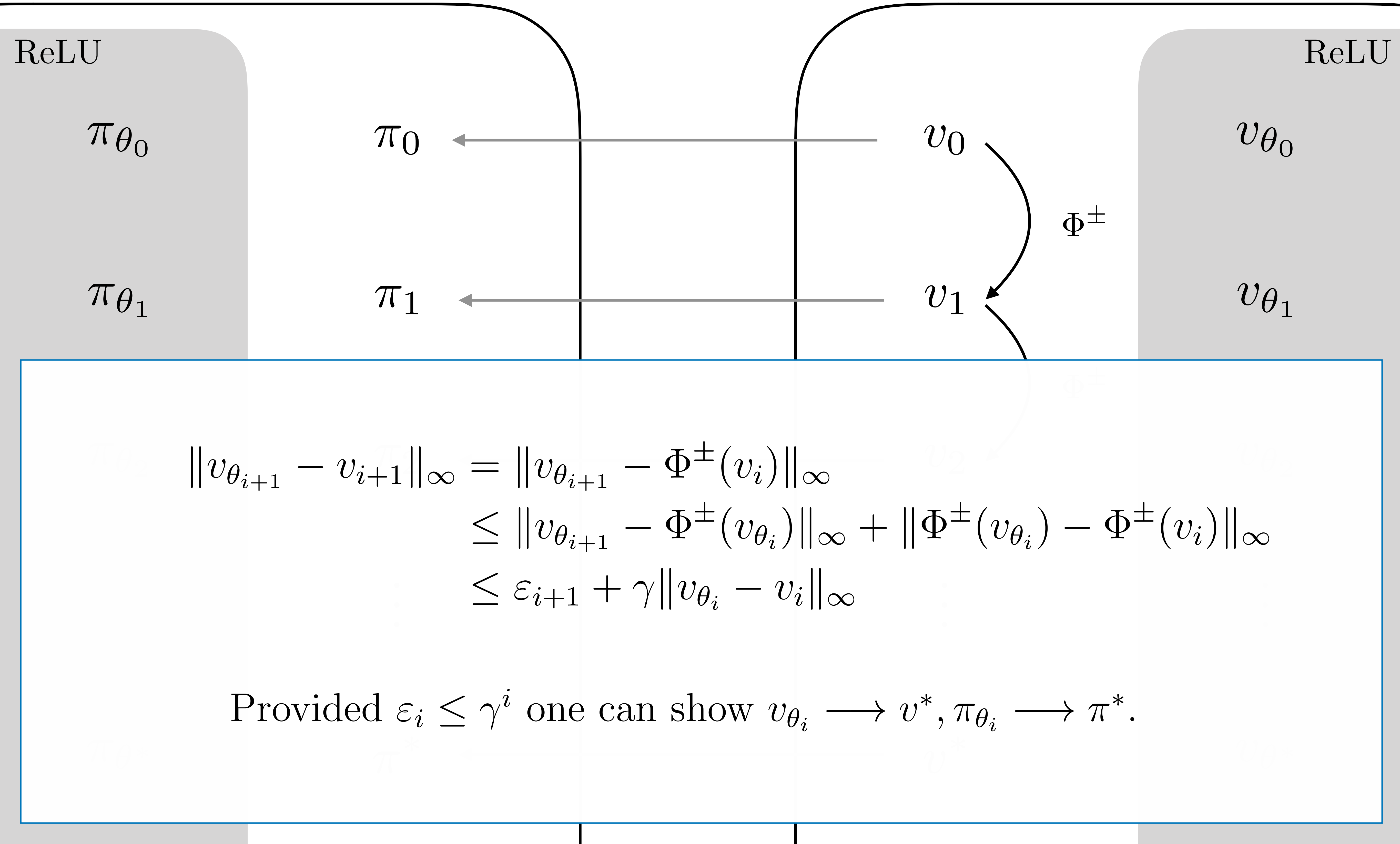


In each step, we train a neural network $f_{\theta_{i+1}} = (v_{\theta_{i+1}}, \pi_{\theta_{i+1}})$ such that

$$\|v_{\theta_{i+1}} - \Phi^{\pm}(v_{\theta_i})\|_\infty < \varepsilon_{i+1}$$

$$\|\pi_{\theta_{i+1}} - P(\Phi^{\pm}(v_{\theta_i}))\|_\infty < \varepsilon_{i+1}$$

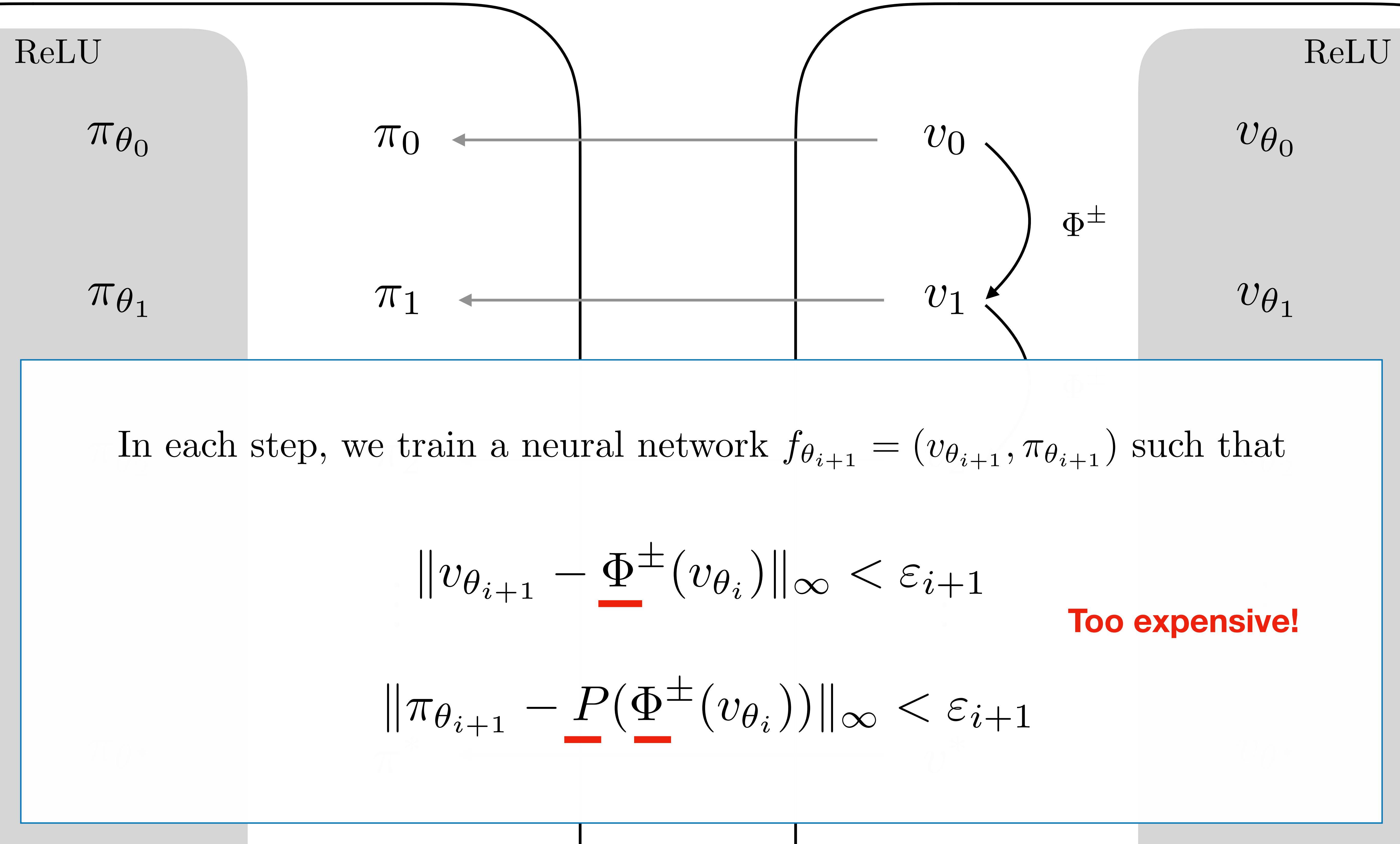
$$\begin{tikzcd}
\mathcal{P} \arrow[r, shift left, "V"] & \mathcal{V} \arrow[l, shift left, "P"] \arrow[loop right, "\Phi^{\pm}"]
\end{tikzcd}$$



$$\begin{aligned}
\|v_{\theta_{i+1}} - v_{i+1}\|_\infty &= \|v_{\theta_{i+1}} - \Phi^{\pm}(v_i)\|_\infty \\
&\leq \|v_{\theta_{i+1}} - \Phi^{\pm}(v_{\theta_i})\|_\infty + \|\Phi^{\pm}(v_{\theta_i}) - \Phi^{\pm}(v_i)\|_\infty \\
&\leq \varepsilon_{i+1} + \gamma\|v_{\theta_i} - v_i\|_\infty
\end{aligned}$$

Provided $\varepsilon_i \leq \gamma^i$ one can show $v_{\theta_i} \longrightarrow v^*, \pi_{\theta_i} \longrightarrow \pi^*$.

$$\begin{tikzcd} \mathcal{P} \arrow[r, shift left, "V"] & \mathcal{V} \arrow[l, shift left, "P"] \arrow[loop right, "\Phi^{\pm}"] \end{tikzcd}$$

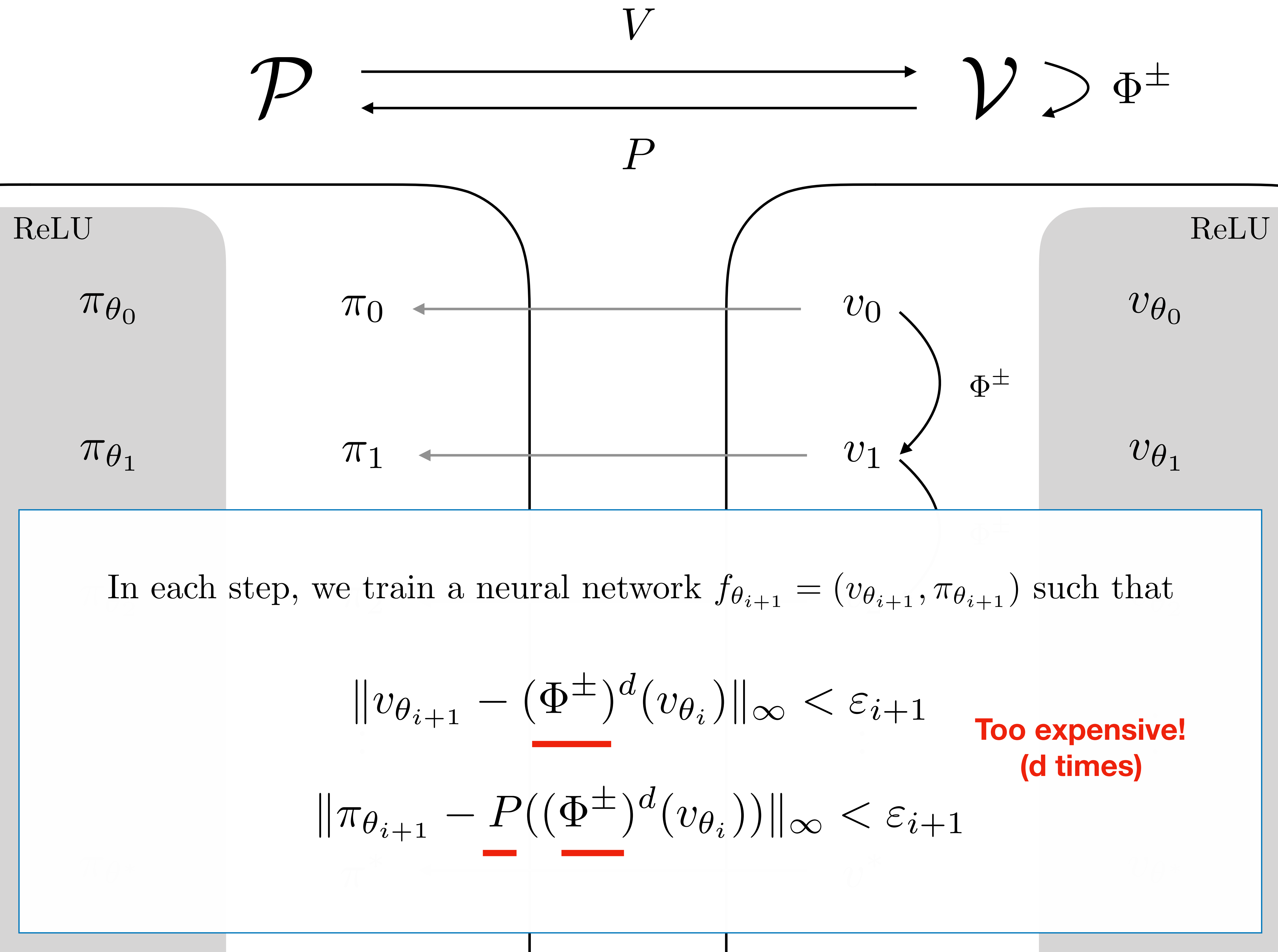


In each step, we train a neural network $f_{\theta_{i+1}} = (v_{\theta_{i+1}}, \pi_{\theta_{i+1}})$ such that

$$\|v_{\theta_{i+1}} - \underline{\Phi}^{\pm}(v_{\theta_i})\|_\infty < \varepsilon_{i+1}$$

Too expensive!

$$\|\pi_{\theta_{i+1}} - \underline{P}(\underline{\Phi}^{\pm}(v_{\theta_i}))\|_\infty < \varepsilon_{i+1}$$



$$\begin{tikzcd}\mathcal{P} \arrow[r, shift left, "V"] & \mathcal{V} \arrow[l, shift left, "P"] \arrow[loop right, "\Phi^{\pm}"]\end{tikzcd}$$

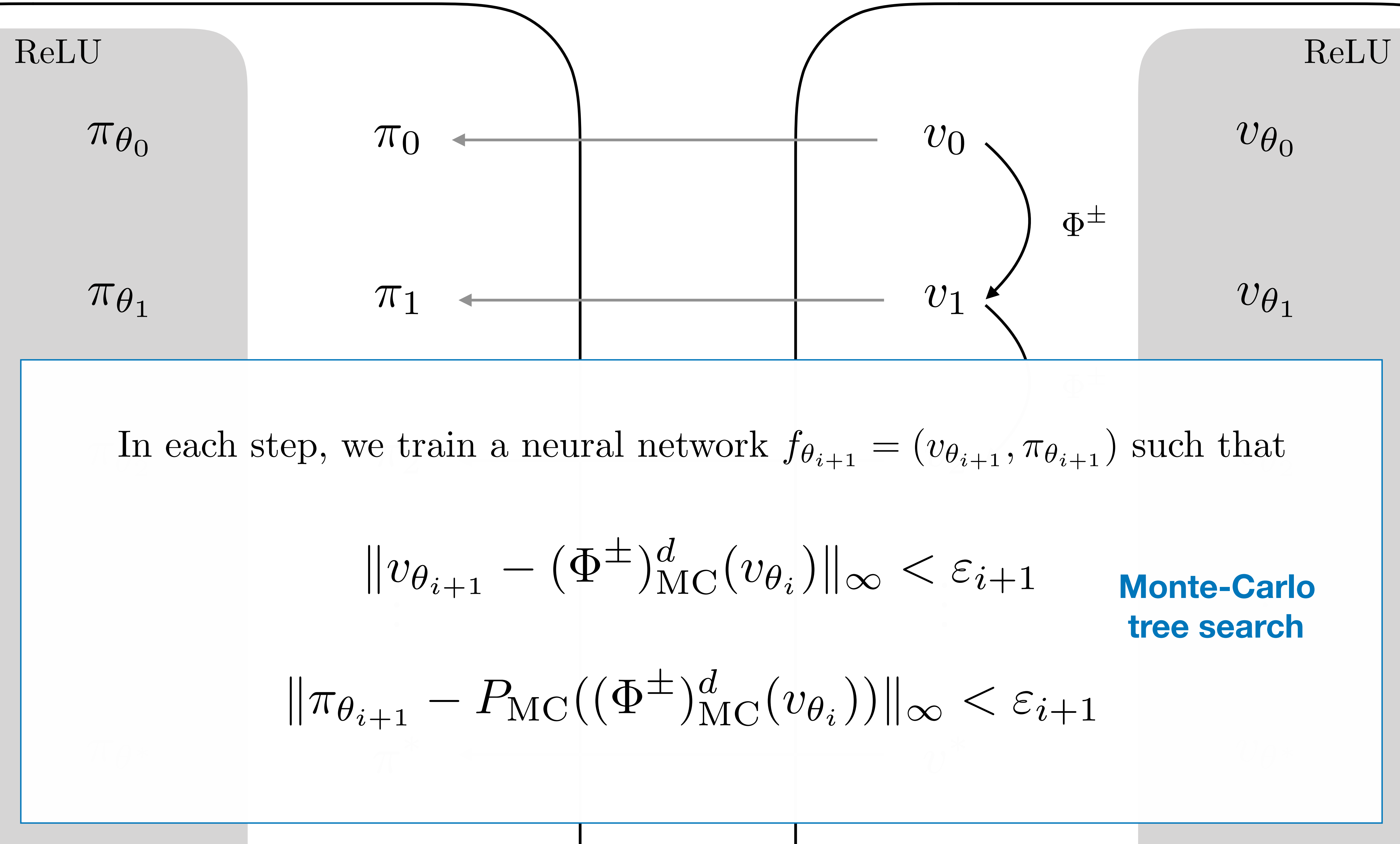
In each step, we train a neural network $f_{\theta_{i+1}} = (v_{\theta_{i+1}}, \pi_{\theta_{i+1}})$ such that

$$\|v_{\theta_{i+1}} - \underline{(\Phi^{\pm})}^d(v_{\theta_i})\|_\infty < \varepsilon_{i+1}$$

$$\|\pi_{\theta_{i+1}} - \underline{P}((\underline{\Phi^{\pm}})^d(v_{\theta_i}))\|_\infty < \varepsilon_{i+1}$$

Too expensive!
(d times)

$$\begin{tikzcd}
\mathcal{P} \arrow[r, shift left, "V"] & \mathcal{V} \arrow[l, shift left, "P"] \arrow[loop right, "\Phi^{\pm}"]
\end{tikzcd}$$



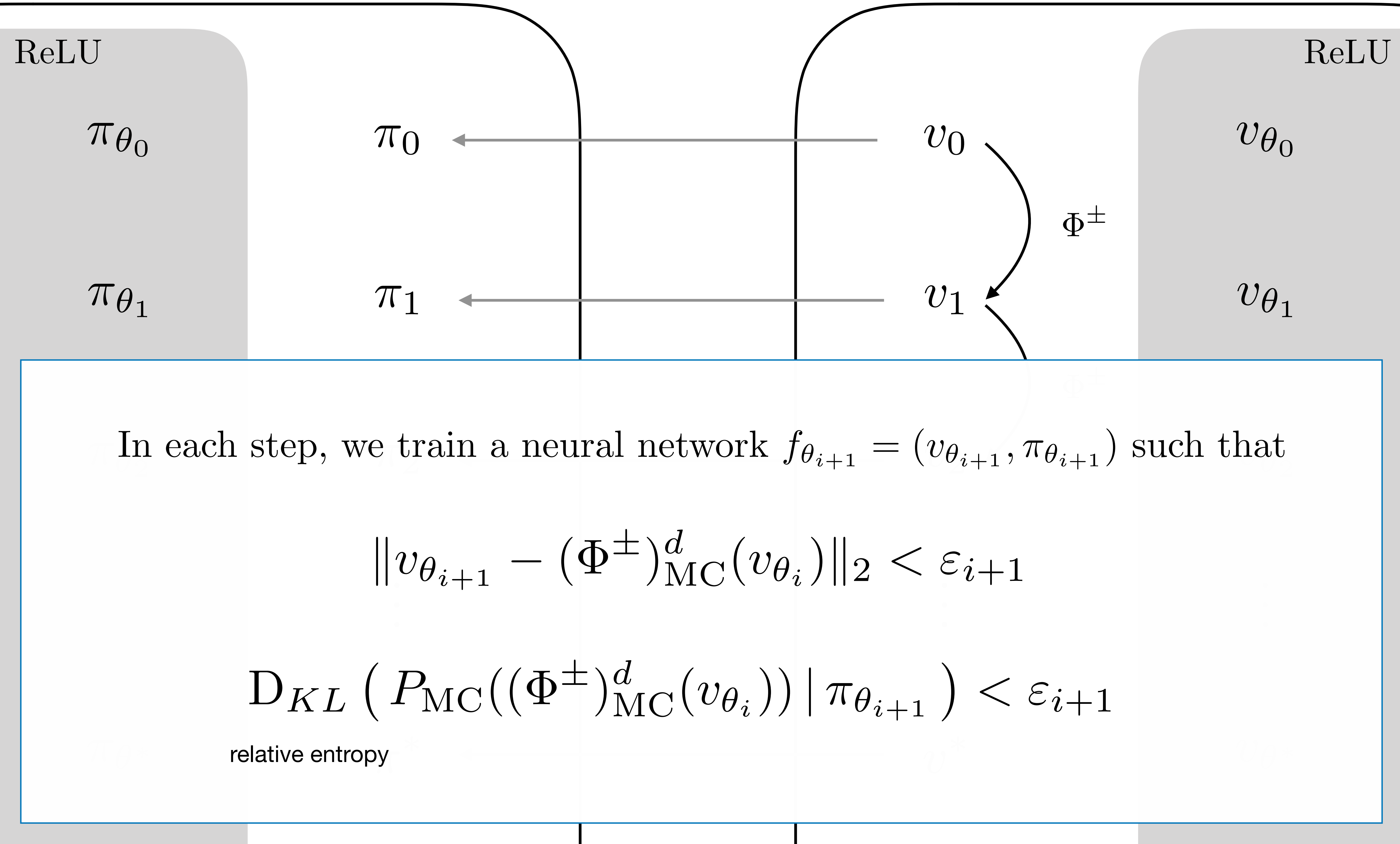
In each step, we train a neural network $f_{\theta_{i+1}} = (v_{\theta_{i+1}}, \pi_{\theta_{i+1}})$ such that

$$\|v_{\theta_{i+1}} - (\Phi^{\pm})^d_{\mathrm{MC}}(v_{\theta_i})\|_\infty < \varepsilon_{i+1}$$

Monte-Carlo tree search

$$\|\pi_{\theta_{i+1}} - P_{\mathrm{MC}}((\Phi^{\pm})^d_{\mathrm{MC}}(v_{\theta_i}))\|_\infty < \varepsilon_{i+1}$$

$$\begin{tikzcd}\mathcal{P} \arrow[r, shift left, "V"] & \mathcal{V} \arrow[l, shift left, "P"] \arrow[loop right, "\Phi^{\pm}"]\end{tikzcd}$$



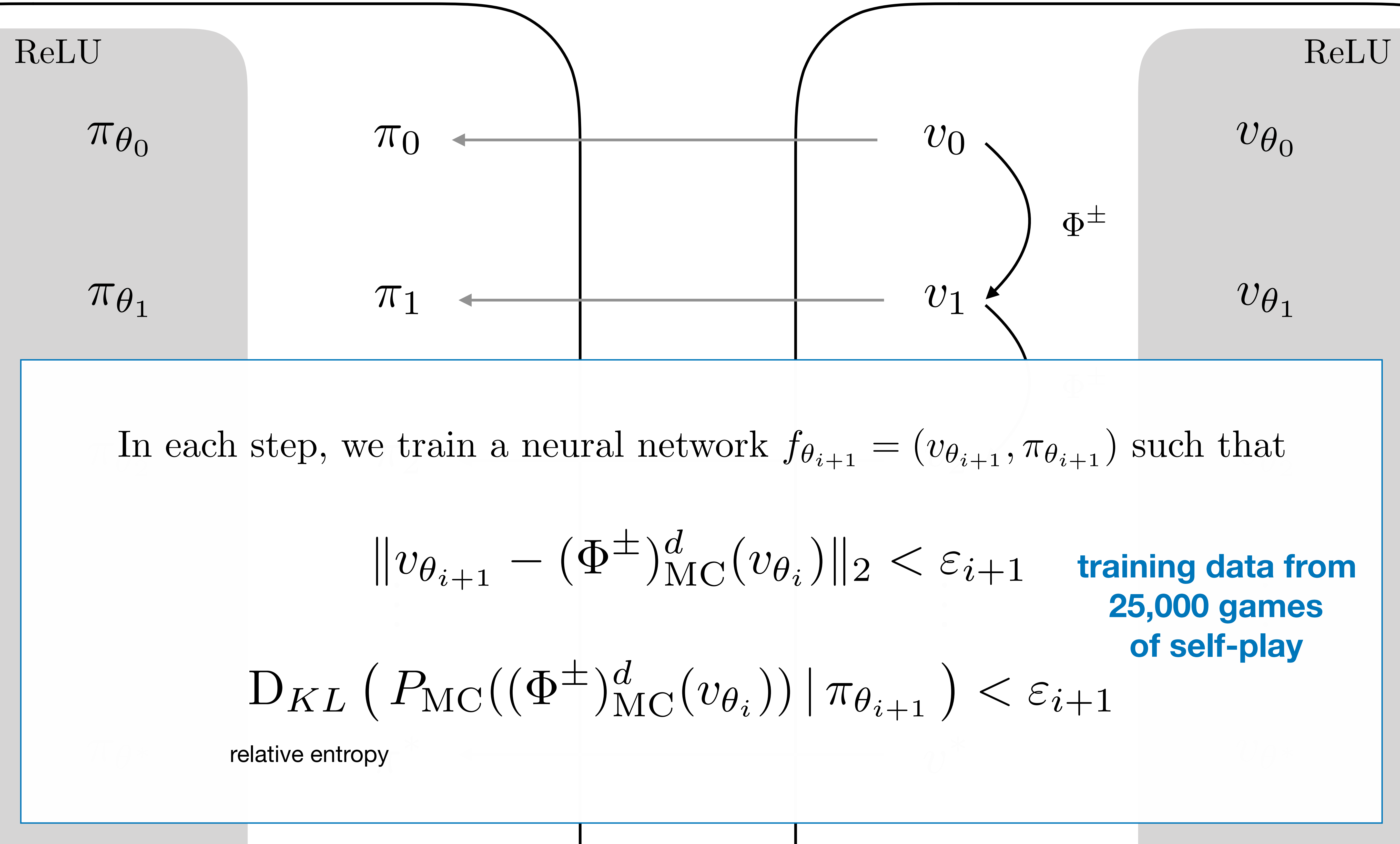
In each step, we train a neural network $f_{\theta_{i+1}} = (v_{\theta_{i+1}}, \pi_{\theta_{i+1}})$ such that

$$\|v_{\theta_{i+1}} - (\Phi^{\pm})^{d}_{\mathrm{MC}}(v_{\theta_i})\|_2 < \varepsilon_{i+1}$$

$$\mathrm{D}_{KL}\left(P_{\mathrm{MC}}((\Phi^{\pm})^{d}_{\mathrm{MC}}(v_{\theta_i})) \,|\, \pi_{\theta_{i+1}} \right) < \varepsilon_{i+1}$$

relative entropy

$$\begin{tikzcd}\mathcal{P} \arrow[r, shift left, "V"] & \mathcal{V} \arrow[l, shift left, "P"] \arrow[loop right, "\Phi^{\pm}"]\end{tikzcd}$$



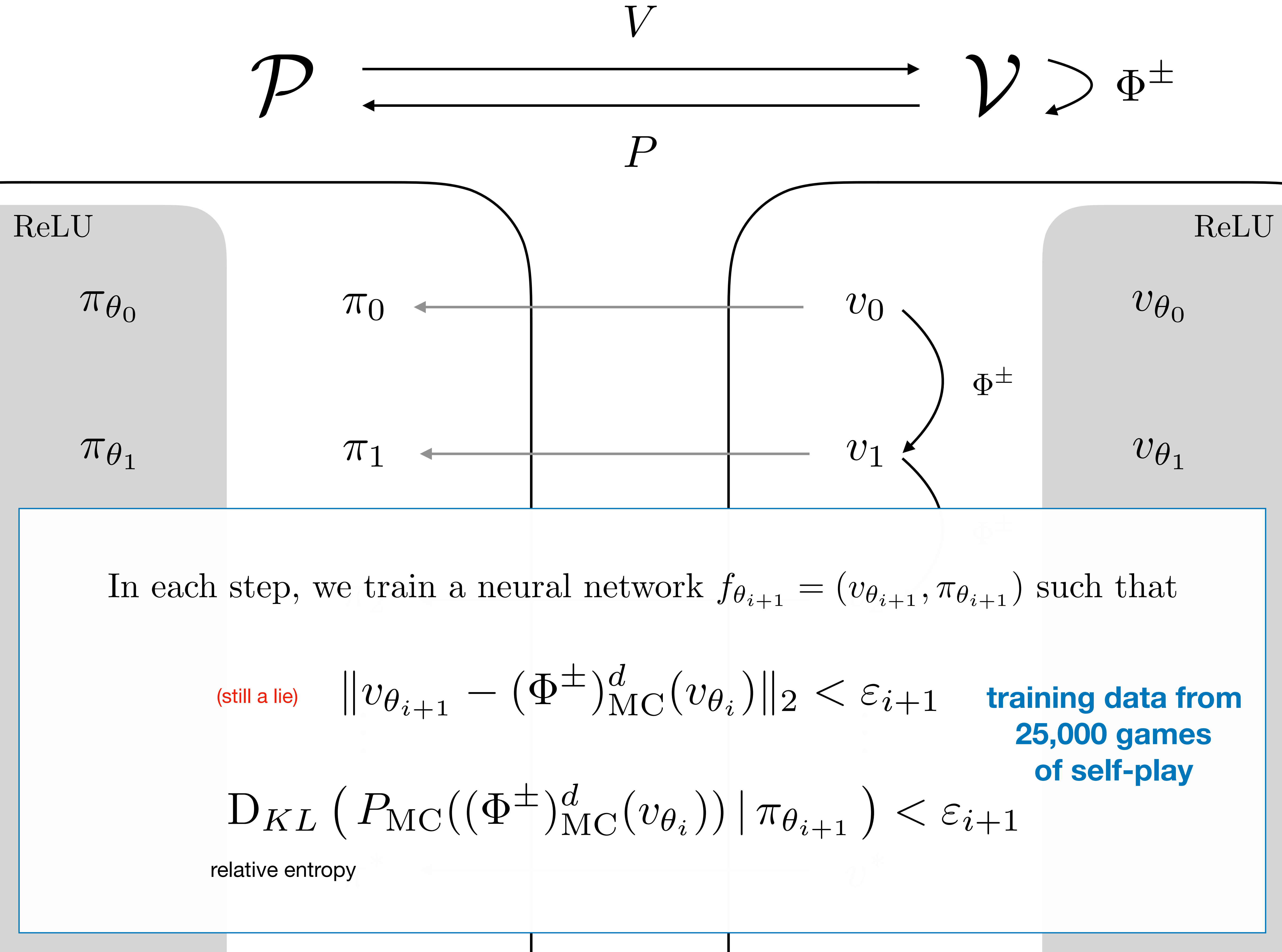
In each step, we train a neural network $f_{\theta_{i+1}} = (v_{\theta_{i+1}}, \pi_{\theta_{i+1}})$ such that

$$\| v_{\theta_{i+1}} - (\Phi^{\pm})^d_{\mathrm{MC}}(v_{\theta_i}) \|_2 < \varepsilon_{i+1}$$

training data from 25,000 games of self-play

$$\mathrm{D}_{KL}\left(P_{\mathrm{MC}}((\Phi^{\pm})^d_{\mathrm{MC}}(v_{\theta_i})) \,|\, \pi_{\theta_{i+1}} \right) < \varepsilon_{i+1}$$

relative entropy



$$\begin{tikzcd}\mathcal{P} \arrow[r, shift left, "V"] & \mathcal{V} \arrow[l, shift left, "P"] \arrow[loop right, "\Phi^\pm"]\end{tikzcd}$$

In each step, we train a neural network $f_{\theta_{i+1}} = (v_{\theta_{i+1}}, \pi_{\theta_{i+1}})$ such that

(still a lie) $$\|v_{\theta_{i+1}} - (\Phi^\pm)^d_{\mathrm{MC}}(v_{\theta_i})\|_2 < \varepsilon_{i+1}$$

training data from 25,000 games of self-play

$$\mathrm{D}_{KL}\left(P_{\mathrm{MC}}((\Phi^\pm)^d_{\mathrm{MC}}(v_{\theta_i})) \,|\, \pi_{\theta_{i+1}} \right) < \varepsilon_{i+1}$$

relative entropy

AlphaGo algorithm summary

- **Step 1**: approximate policy and value functions by a deep convolutional neural network.
- **Step 2**: project the Bellman iteration onto these approximations
    - Monte-Carlo tree search to approximate supremums
    - self-play to generate the training data

AlphaGo neural network

$$\boxed{\begin{gathered} f_\theta : \mathbb{R}^{6137} \longrightarrow \mathbb{R}^{19\times 19+1} \times \mathbb{R} \\ (\pi_\theta(s), v_\theta(s)) := f_\theta(s) \end{gathered}}$$

New programs, new math

- New kind of program: differentiable tensor programs that compute with learned representations.
- **These programs will be important for natural science**: chemistry (Molecular transformer), protein folding (AlphaFold), engineering complex quantum states, …
- Desiderata: large combinatorial search space, clear objective function, lots of ground truth data or an efficient simulator (**Hassabis** 50:10)
- **New mathematics**: Neural Tangent Kernel, tensor programs, linear logic, differential categories, …
- **Ask an expert**: **deep learning in Australia**.

References

- **(AlphaGo)** David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel and Demis Hassabis, “Mastering the game of Go with deep neural networks and tree search”, Nature 2016.

- **(AlphaGo Zero)** David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel and Demis Hassabis, “Mastering the game of Go without human knowledge”, Nature 2017.

References

- **(Deep learning text)** I. Goodfellow, Y. Bengio and A. Courville, “Deep learning”, MIT Press, 2016.
- **(Reinforcement learning text)** R. S. Sutton and A. G. Barto, “Reinforcement learning: an introduction”, 2nd edition, MIT Press, 2018.
- Rules of Go: https://www.britgo.org/intro/intro2.html
- P. Norvig, “As we may program”, talk May 2017.
- M. Sadler, N. Regan, “Game Changer: AlphaZero's Groundbreaking Chess Strategies and the Promise of AI”.
- B. Hanin and M. Sellke, “Approximating continuous functions by ReLU nets of minimal width”, arXiv:1710.11278.
- Deep reinforcement learning seminar: http://therisingsea.org/post/seminar-ch/

References

Applications to Natural Science

- P. Schwaller et al “Molecular Transformer: A Model for Uncertainty-Calibrated Chemical Reaction Prediction” ACS Cent. Sci 2019.
- (AlphaFold) R. Evans et al “De novo structure prediction with deep-learning based scoring” preprint 2018.
- G. Carleo et al “Machine learning and the physical sciences” arXiv:1903.10563.

Mathematics

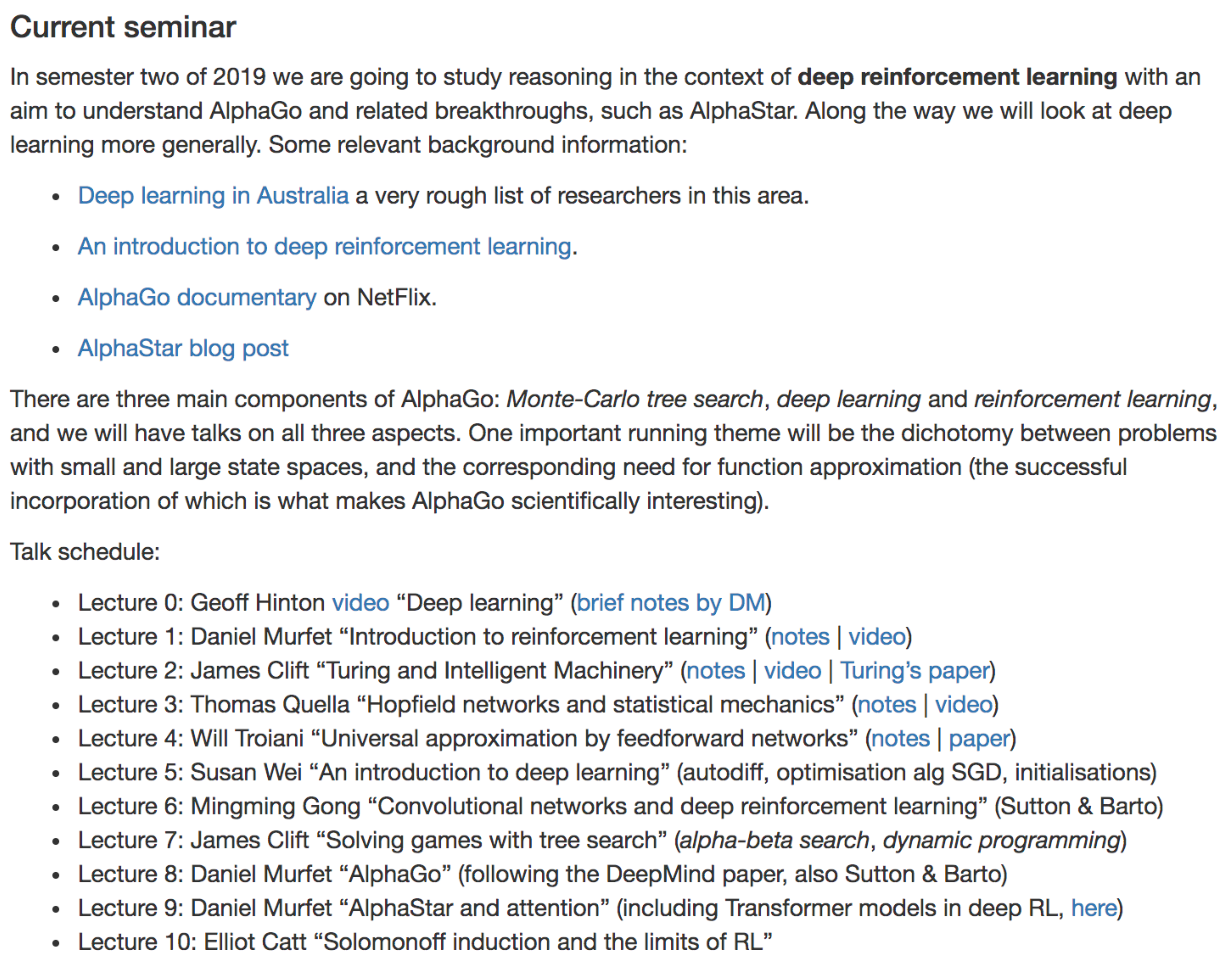
- A. Jacot, F. Gabriel, and C. Hongler “Neural tangent kernel: Convergence and generalization in neural networks” In Advances in neural information processing systems, pages 8571–8580, 2018.
- G. Yang “Tensor Programs I: Wide Feedforward or Recurrent Neural Networks of Any Architecture are Gaussian Processes” arXiv: 1910.12478.

Bonus slides

AlphaGo: conclusion

- AlphaGo is the product of: neural networks, reinforcement learning and Monte-Carlo tree search.
- **AlphaGo**: included hand-coded features by Go experts
- **AlphaGo Zero**: “zero” human knowledge about Go
- **AlphaZero**: also plays Chess and Shogi (same hyperparameters, new weights trained from scratch).
- Less search than standard chess engines (10,000s vs 10,000,000s).
- Raw network play: Elo 3055 (**9 dan**, **#486**)
- Is AlphaGo good news for Go?

http://therisingsea.org/post/seminar-ch/



Current seminar

In semester two of 2019 we are going to study reasoning in the context of **deep reinforcement learning** with an aim to understand AlphaGo and related breakthroughs, such as AlphaStar. Along the way we will look at deep learning more generally. Some relevant background information:

- Deep learning in Australia a very rough list of researchers in this area.
- An introduction to deep reinforcement learning.
- AlphaGo documentary on NetFlix.
- AlphaStar blog post

There are three main components of AlphaGo: *Monte-Carlo tree search*, *deep learning* and *reinforcement learning*, and we will have talks on all three aspects. One important running theme will be the dichotomy between problems with small and large state spaces, and the corresponding need for function approximation (the successful incorporation of which is what makes AlphaGo scientifically interesting).

Talk schedule:

- Lecture 0: Geoff Hinton video "Deep learning" (brief notes by DM)
- Lecture 1: Daniel Murfet "Introduction to reinforcement learning" (notes | video)
- Lecture 2: James Clift "Turing and Intelligent Machinery" (notes | video | Turing's paper)
- Lecture 3: Thomas Quella "Hopfield networks and statistical mechanics" (notes | video)
- Lecture 4: Will Troiani "Universal approximation by feedforward networks" (notes | paper)
- Lecture 5: Susan Wei "An introduction to deep learning" (autodiff, optimisation alg SGD, initialisations)
- Lecture 6: Mingming Gong "Convolutional networks and deep reinforcement learning" (Sutton & Barto)
- Lecture 7: James Clift "Solving games with tree search" (*alpha-beta search*, *dynamic programming*)
- Lecture 8: Daniel Murfet "AlphaGo" (following the DeepMind paper, also Sutton & Barto)
- Lecture 9: Daniel Murfet "AlphaStar and attention" (including Transformer models in deep RL, here)
- Lecture 10: Elliot Catt "Solomonoff induction and the limits of RL"

Theorem. *The function* $\Phi_\pi : \mathcal{V} \longrightarrow \mathcal{V}$ *defined by*

$$\Phi_\pi(v)(s) = R(s) + \sum_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \gamma \pi(a|s) P(s'|s,a) v(s')$$

has a unique fixed point v_π*.*

Proof. Since $\mathcal{V}$ is a complete metric space, it suffices by the Banach fixed point theorem to prove that Φ_π is a γ-contraction mapping. But for $v, v' \in \mathcal{V}$

$$\begin{aligned} |\Phi_\pi(v)(s) - \Phi_\pi(v')(s)| &\leq \sum_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \gamma \pi(a|s) P(s'|s,a) |v(s') - v'(s')| \\ &\leq \sum_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \gamma \pi(a|s) P(s'|s,a) d_\infty(v, v') \\ &= \gamma d_\infty(v, v')\,. \end{aligned}$$

Hence $d_\infty(\Phi_\pi(v), \Phi_\pi(v')) \leq \gamma d_\infty(v, v')$. $\square$

$$v = v_{\theta_i} \quad \Phi^{\pm}(v)(s) = R(s) + (-1)^{|s|} \sup_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \gamma P(s'|s,a)(-1)^{|s|} v(s')$$

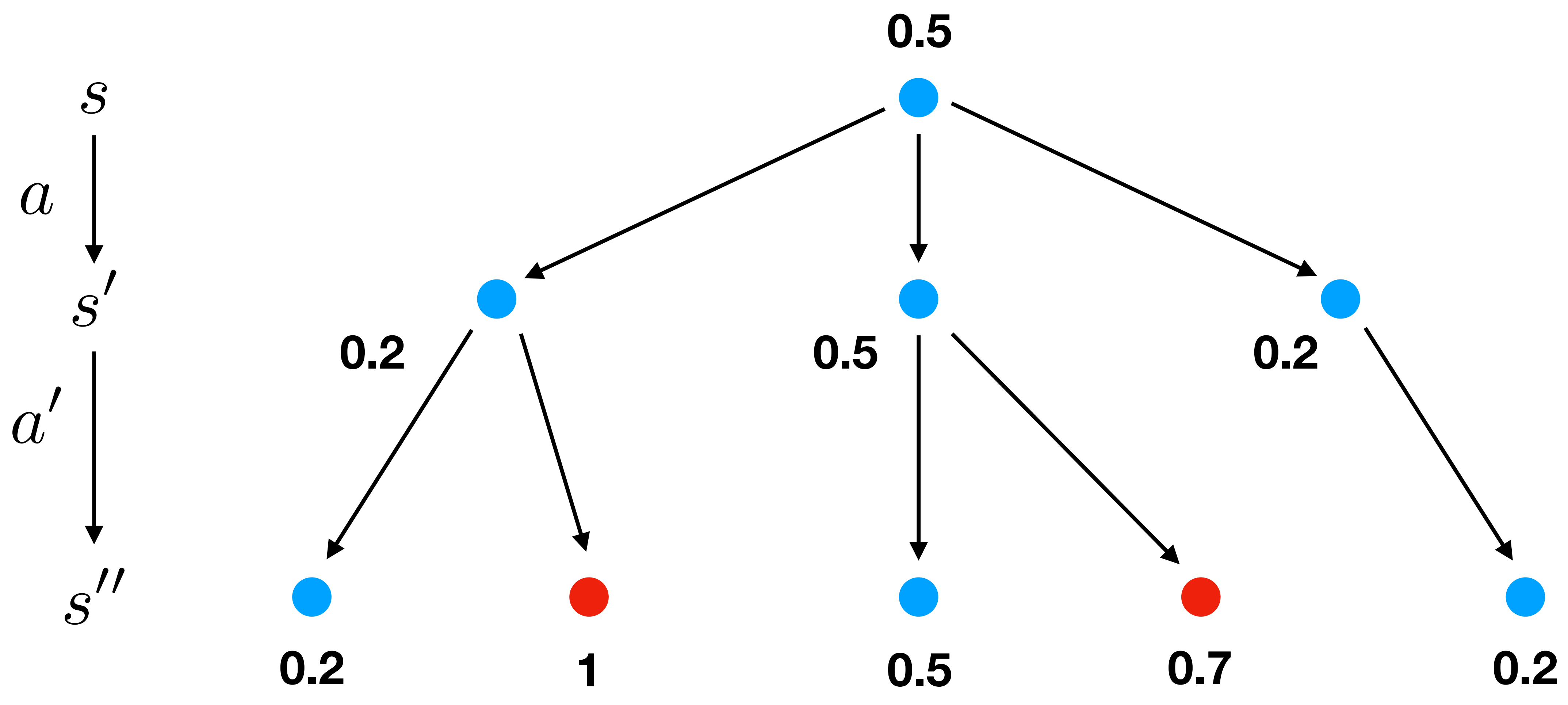
$$s \in \mathcal{S}_0 \quad (\Phi^{\pm})^2(v)(s) = R(s) + \gamma \sup_{a \in \mathcal{A}(s)} \left[R(s') + \gamma \inf_{a' \in \mathcal{A}(s')} v(s'') \right]$$

general

$$(\Phi^{\pm})^2(v)(s) \approx \sup_{a \in \mathcal{A}(s)} \inf_{a' \in \mathcal{A}(s')} v(s'')$$

Go, non-terminal

$\gamma \approx 1$



$$v = v_{\theta_i} \qquad \Phi^{\pm}(v)(s) = R(s) + (-1)^{|s|} \sup_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \gamma P(s'|s,a)(-1)^{|s|} v(s')$$

$$s \in \mathcal{S}_0 \qquad (\Phi^{\pm})^2(v)(s) = R(s) + \gamma \sup_{a \in \mathcal{A}(s)} \left[R(s') + \gamma \inf_{a' \in \mathcal{A}(s')} v(s'') \right]$$

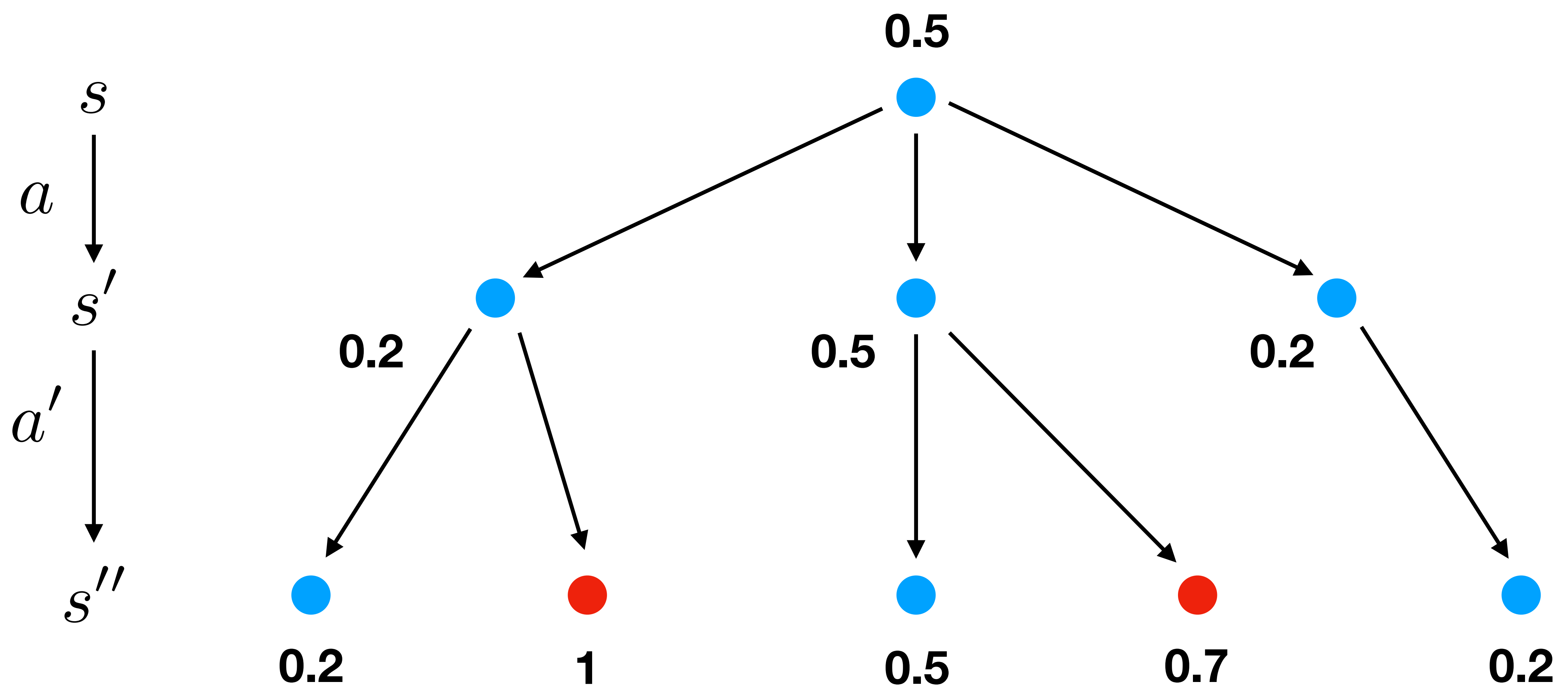
general

$$(\Phi^{\pm})^2(v)(s) \approx \sup_{a \in \mathcal{A}(s)} \inf_{a' \in \mathcal{A}(s')} v(s'')$$

Go, non-terminal

$\gamma \approx 1$

How to truncate these?



$$v = v_{\theta_i} \quad \Phi^{\pm}(v)(s) = R(s) + (-1)^{|s|} \sup_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}} \gamma P(s'|s,a)(-1)^{|s|} v(s')$$

$$s \in \mathcal{S}_0 \quad (\Phi^{\pm})^2(v)(s) = R(s) + \gamma \sup_{a \in \mathcal{A}(s)} \Big[R(s') + \gamma \inf_{a' \in \mathcal{A}(s')} v(s'') \Big]$$

general

⚡ ⚡

$$(\Phi^{\pm})^2(v)(s) \approx \sup_{a \in \mathcal{A}(s)} \inf_{a' \in \mathcal{A}(s')} v(s'')$$

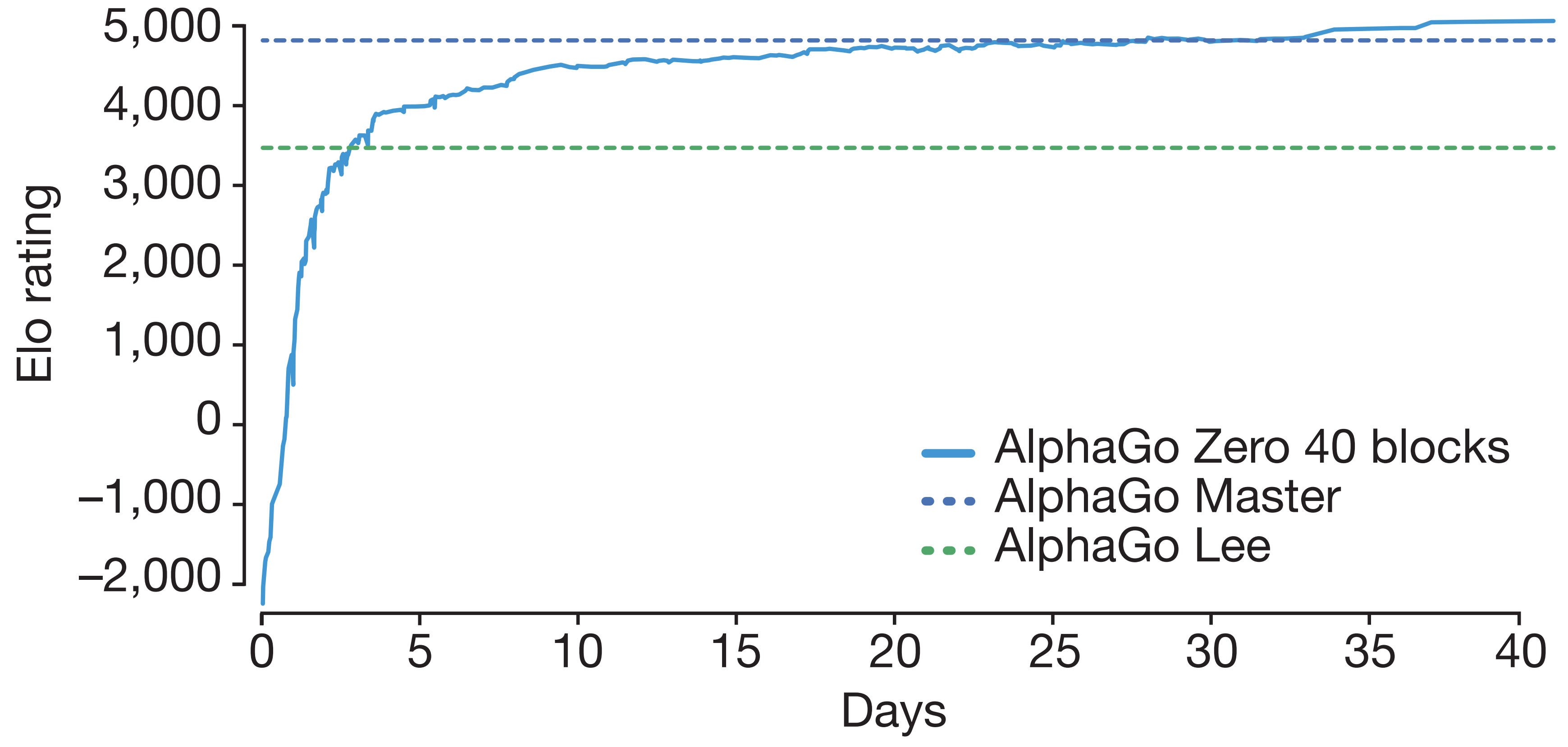
Go, non-terminal

$\gamma \approx 1$

How to truncate these?

- Let $\pi = \pi_{\theta_i}, v = v_{\theta_i}$ be the current policy and value function.
- To approximate $(\Phi^{\pm})^d(v)(s)$ we need to approximate the d nested supremums.
- AlphaGo searches over sequences of $\leq d$ actions using an algorithm which prioritises those paths likely to contribute to chain of suprema according to π, v.
- This algorithm is called *Monte-Carlo tree search.*
- The search looks at 1600 paths to evaluate this approximation

$$(\Phi^{\pm})^d_{\mathrm{MC}}(v)(s) \approx (\Phi^{\pm})^d(v)(s)\,.$$



AlphaGo Zero (40 blocks)
5185

Ke Jie (#1 human)
3687

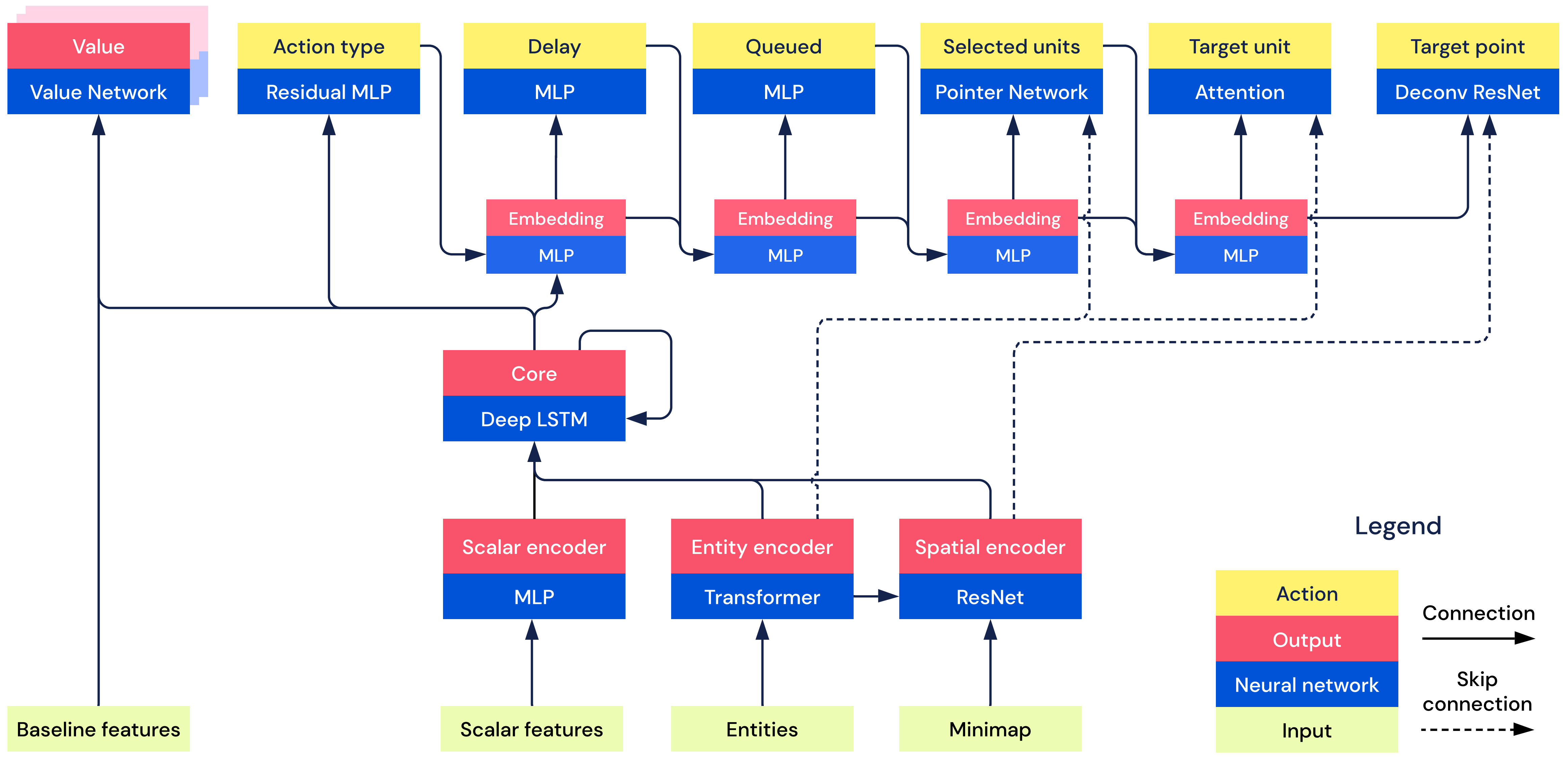
$$P(a \text{ defeats } b) = \frac{1}{1+\exp(c_{\text{elo}}(e(b)-e(a))},$$

$$P(KJ > AGZ) = 0.023$$

Humankind has accumulated Go knowledge from millions of games played over thousands of years, collectively distilled into patterns, proverbs and books. In the space of a few days, starting *tabula rasa*, AlphaGo Zero was able to rediscover much of this Go knowledge, as well as novel strategies that provide new insights into the oldest of games.

D. Silver et al, "Mastering the game of Go without human knowledge", Nature 2017.

AlphaStar network architecture



O. Vinyals et al "Grandmaster level in StarCraft II using multi-agent reinforcement learning", Extended data.

Alternating Markov games

Definition. An *alternating Markov Game* is an MDP together with a decomposition of the state space $\mathcal{S} = \mathcal{S}_0 \amalg \mathcal{S}_1$ (we write $|s| = 0$ if $s \in \mathcal{S}_0$ and $|s| = 1$ if $s \in \mathcal{S}_1$ and call these *even* or *odd* states) such that the only possible transitions change the parity:

$$P(s'|s,a) = 0 \text{ if } |s'| = |s|\,.$$

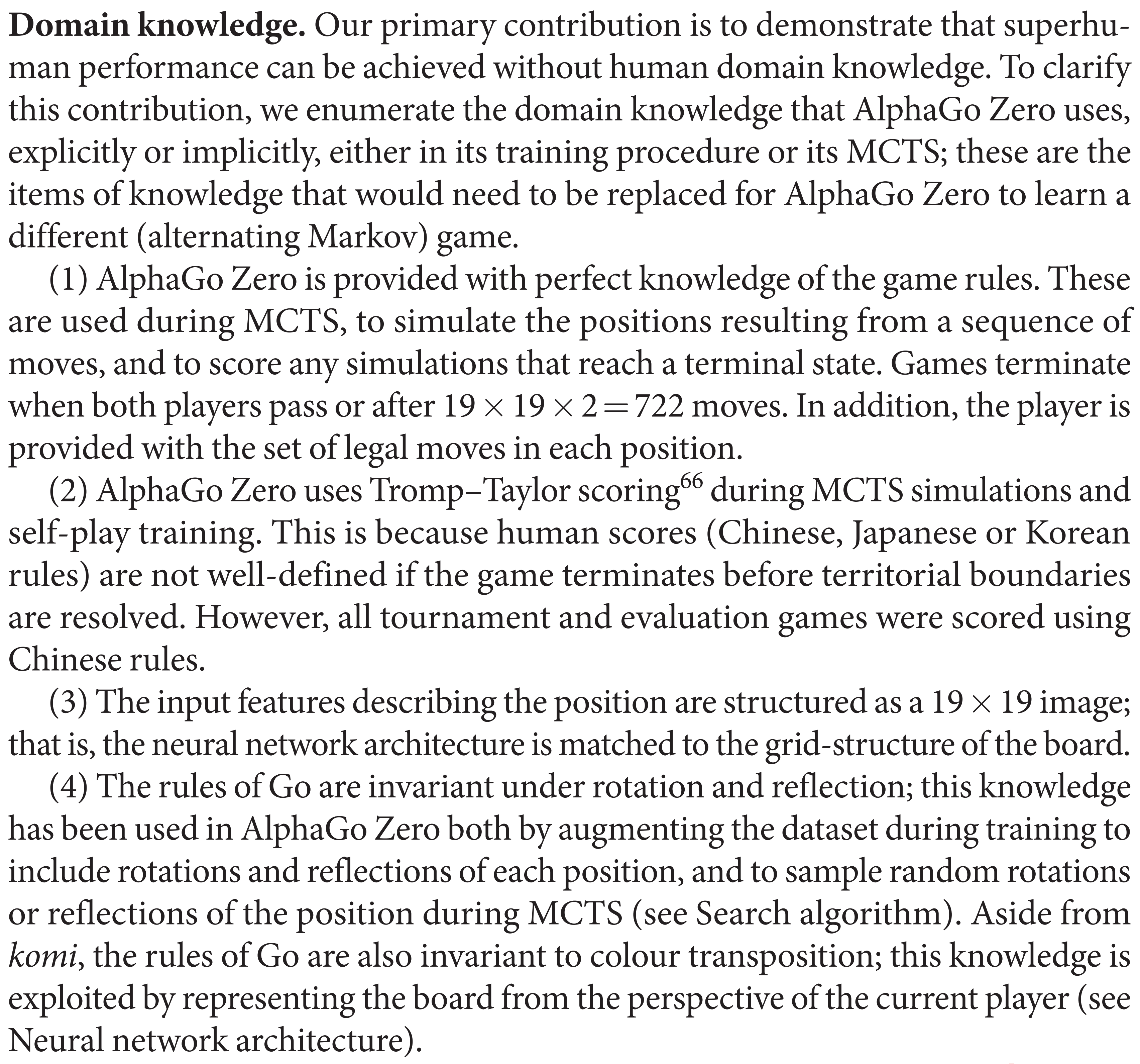
We think of a policy $\pi : \mathcal{S} \longrightarrow \Delta\mathcal{A}$ as a pair of policies $\pi_i = \pi|_{\mathcal{S}_i}$, one for the *even player* and one for the *odd player*, and $R(s)$ is the reward obtained by the even player in state s (the odd player receives $-R(s)$, so this is a zero-sum game).

Remark. If $(\mathcal{S} = \mathcal{S}_0 \amalg \mathcal{S}_1, \mathcal{A}, R, P)$ is an alternating Markov game and $\pi_1 : \mathcal{S}_1 \longrightarrow \Delta\mathcal{A}$ is a policy for the odd player, then there is a Markov Decision Process

$$(\mathcal{S}_0, \mathcal{A}_0, R|_{\mathcal{S}_0}, P^{\pi_1})$$

where the odd player acts as the environment for the even player, so for $s \in \mathcal{S}_0$

$$P^{\pi_1}(s''|s,a) = \sum_{a' \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s'|s,a)\pi_1(a'|s')P(s''|s',a')\,.$$



Domain knowledge. Our primary contribution is to demonstrate that superhuman performance can be achieved without human domain knowledge. To clarify this contribution, we enumerate the domain knowledge that AlphaGo Zero uses, explicitly or implicitly, either in its training procedure or its MCTS; these are the items of knowledge that would need to be replaced for AlphaGo Zero to learn a different (alternating Markov) game.

(1) AlphaGo Zero is provided with perfect knowledge of the game rules. These are used during MCTS, to simulate the positions resulting from a sequence of moves, and to score any simulations that reach a terminal state. Games terminate when both players pass or after $19 \times 19 \times 2 = 722$ moves. In addition, the player is provided with the set of legal moves in each position.

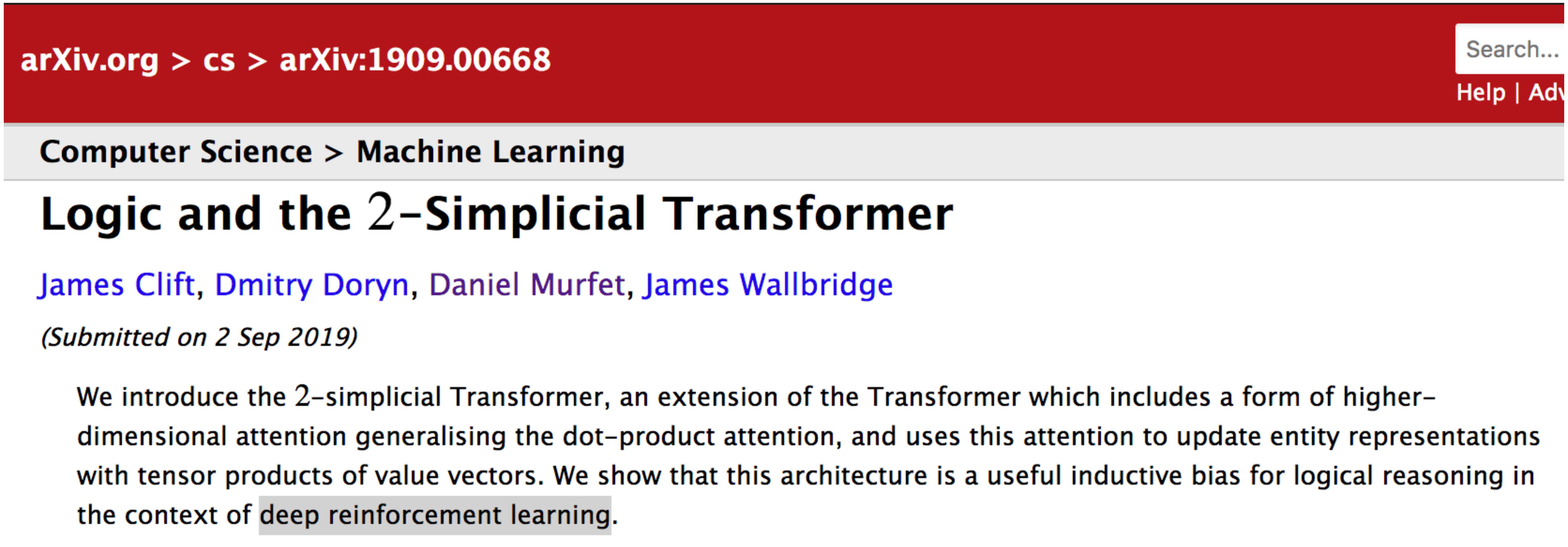
(2) AlphaGo Zero uses Tromp–Taylor scoring[66] during MCTS simulations and self-play training. This is because human scores (Chinese, Japanese or Korean rules) are not well-defined if the game terminates before territorial boundaries are resolved. However, all tournament and evaluation games were scored using Chinese rules.

(3) The input features describing the position are structured as a 19×19 image; that is, the neural network architecture is matched to the grid-structure of the board.

(4) The rules of Go are invariant under rotation and reflection; this knowledge has been used in AlphaGo Zero both by augmenting the dataset during training to include rotations and reflections of each position, and to sample random rotations or reflections of the position during MCTS (see Search algorithm). Aside from *komi*, the rules of Go are also invariant to colour transposition; this knowledge is exploited by representing the board from the perspective of the current player (see Neural network architecture).

+ input representation (7 prior moves)

D. Silver et al "Mastering the game of Go without human knowledge", Methods.



arXiv.org > cs > arXiv:1909.00668

Search...

Help | Adv

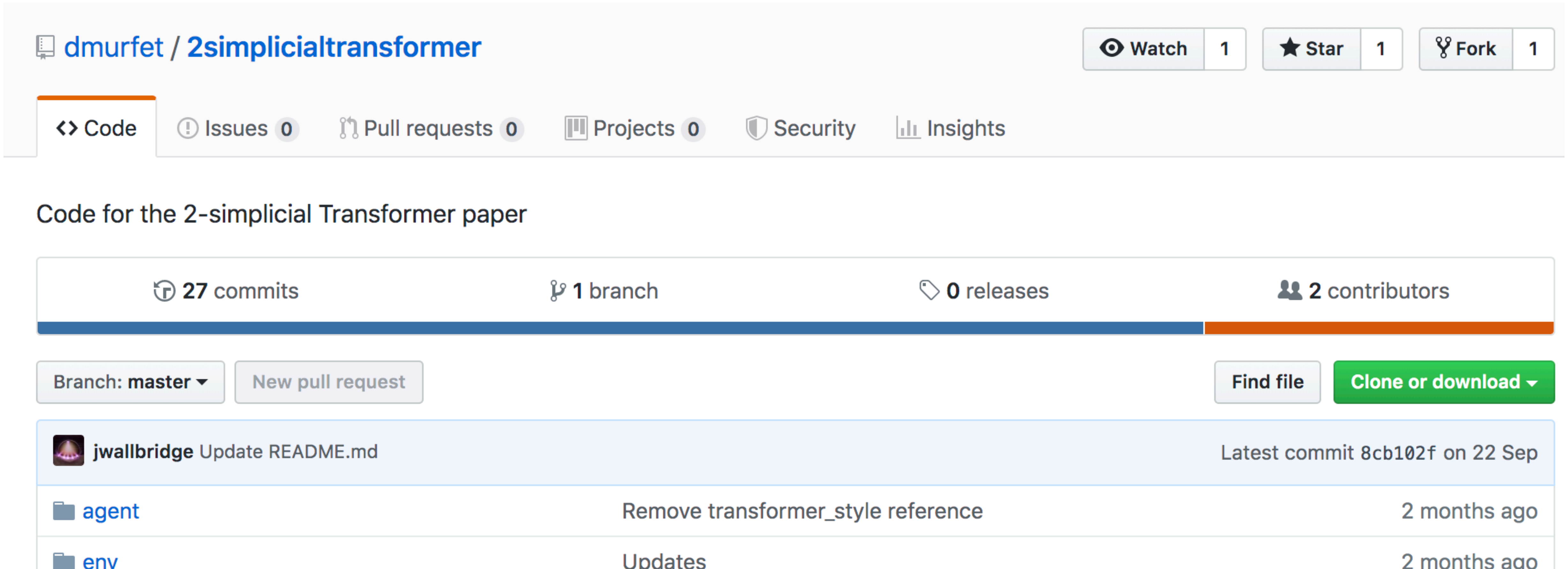
Computer Science > Machine Learning

Logic and the 2-Simplicial Transformer

James Clift, Dmitry Doryn, Daniel Murfet, James Wallbridge

(Submitted on 2 Sep 2019)

We introduce the 2-simplicial Transformer, an extension of the Transformer which includes a form of higher-dimensional attention generalising the dot-product attention, and uses this attention to update entity representations with tensor products of value vectors. We show that this architecture is a useful inductive bias for logical reasoning in the context of deep reinforcement learning.



dmurfet / 2simplicialtransformer

Watch 1 | Star 1 | Fork 1

Code | Issues 0 | Pull requests 0 | Projects 0 | Security | Insights

Code for the 2-simplicial Transformer paper

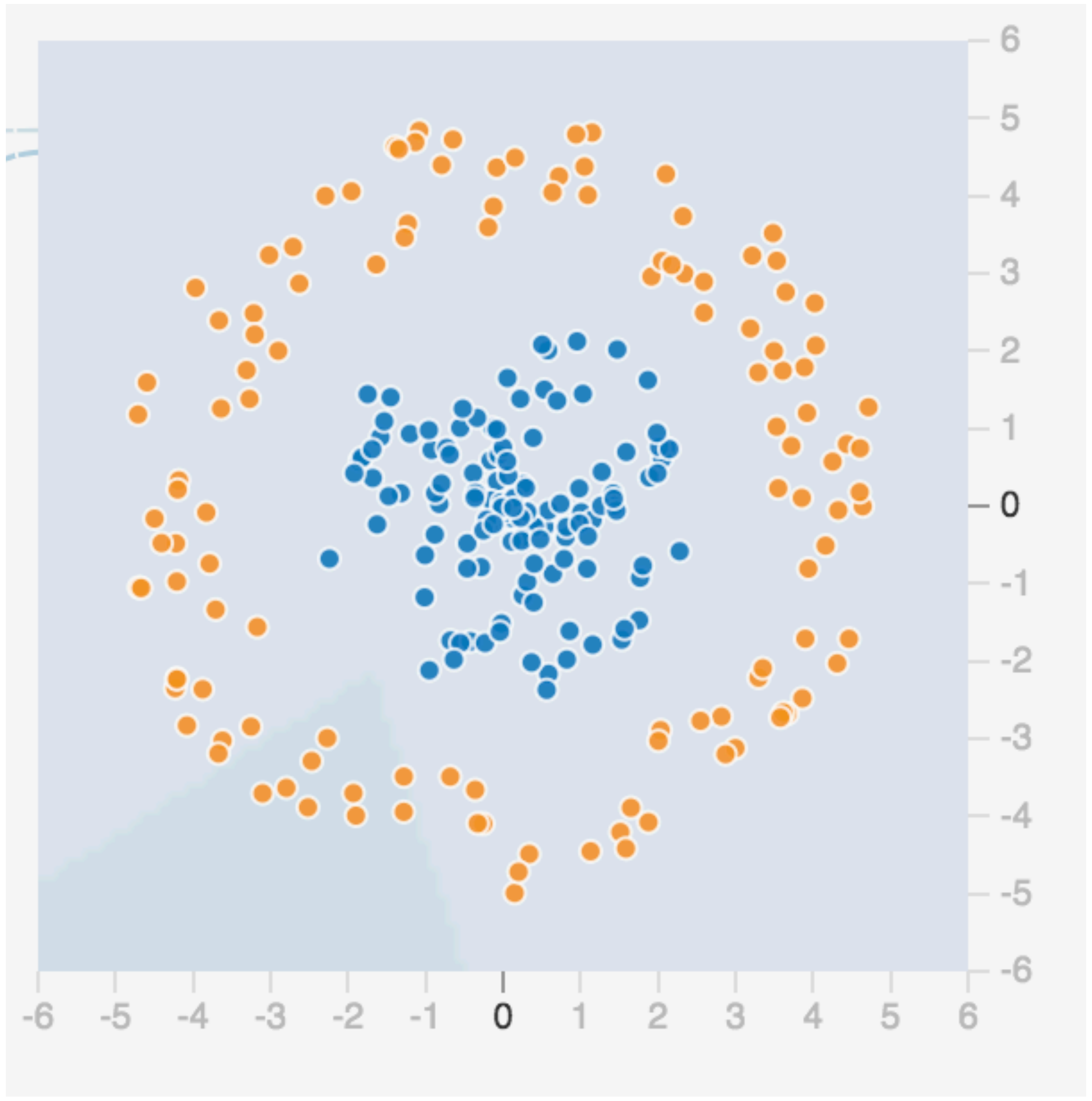
27 commits | 1 branch | 0 releases | 2 contributors

Branch: master | New pull request | Find file | Clone or download

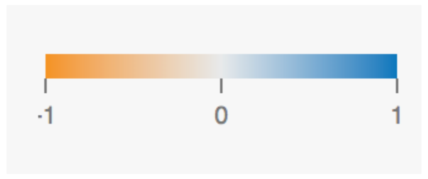
jwallbridge Update README.md		Latest commit `8cb102f` on 22 Sep
agent	Remove transformer_style reference	2 months ago
env	Updates	2 months ago

Theorem (Hanin-Sellke '17). *If $K \subseteq \mathbb{R}^n$ is compact, $f : K \longrightarrow \mathbb{R}^m$ is continuous and $\varepsilon > 0$, there exists a ReLU net $\mathcal{N}$ with widths bounded above by $m+n$ such that*

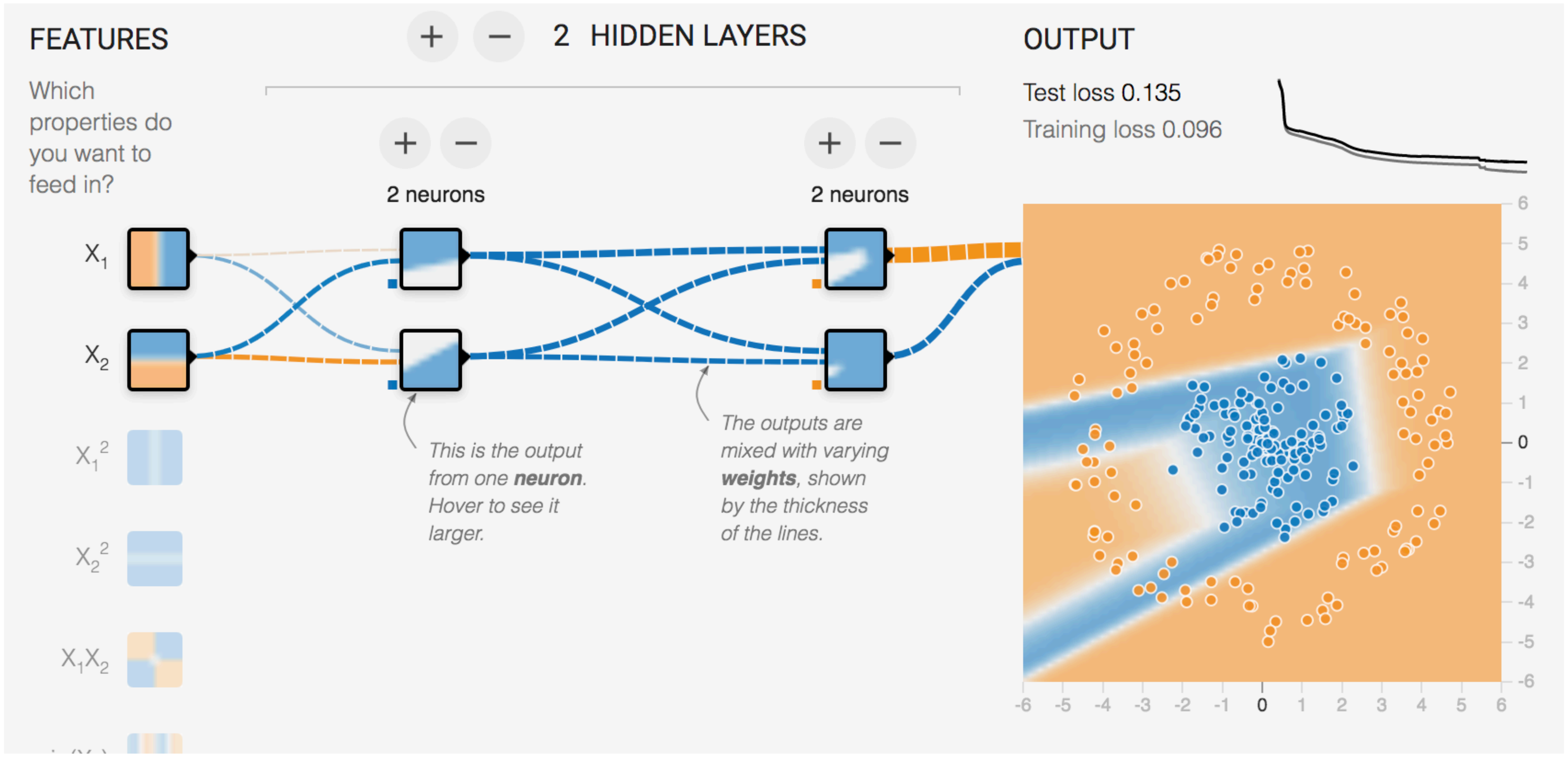
$$\sup_{x \in K} \| f(x) - f_{\mathcal{N}}(x) \| < \varepsilon \, .$$



$$f : [-6,6] \times [-6,6] \longrightarrow [-1,1]$$



playground.tensorflow.org



FEATURES

Which properties do you want to feed in?

2 HIDDEN LAYERS

OUTPUT

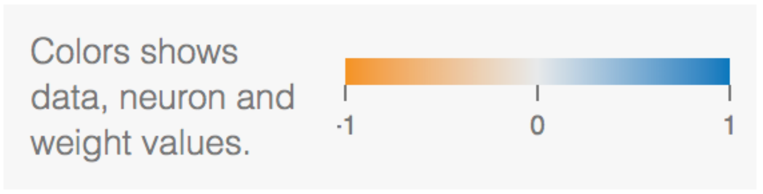
Test loss 0.135

Training loss 0.096

*This is the output from one **neuron**. Hover to see it larger.*

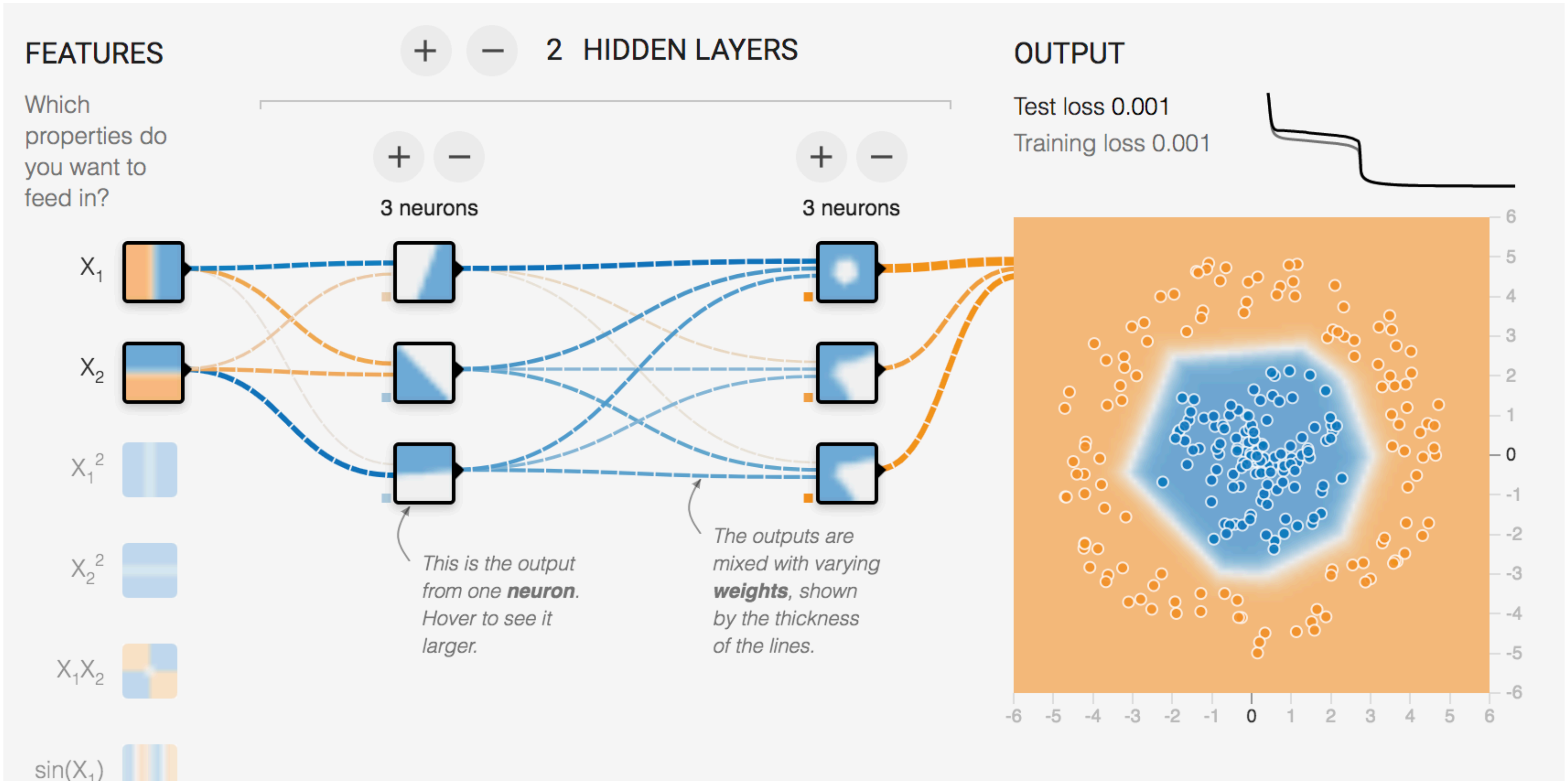
*The outputs are mixed with varying **weights**, shown by the thickness of the lines.*

Not quite!



Colors shows data, neuron and weight values.

playground.tensorflow.org



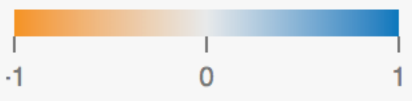
Test loss 0.001
Training loss 0.001

*This is the output from one **neuron**. Hover to see it larger.*

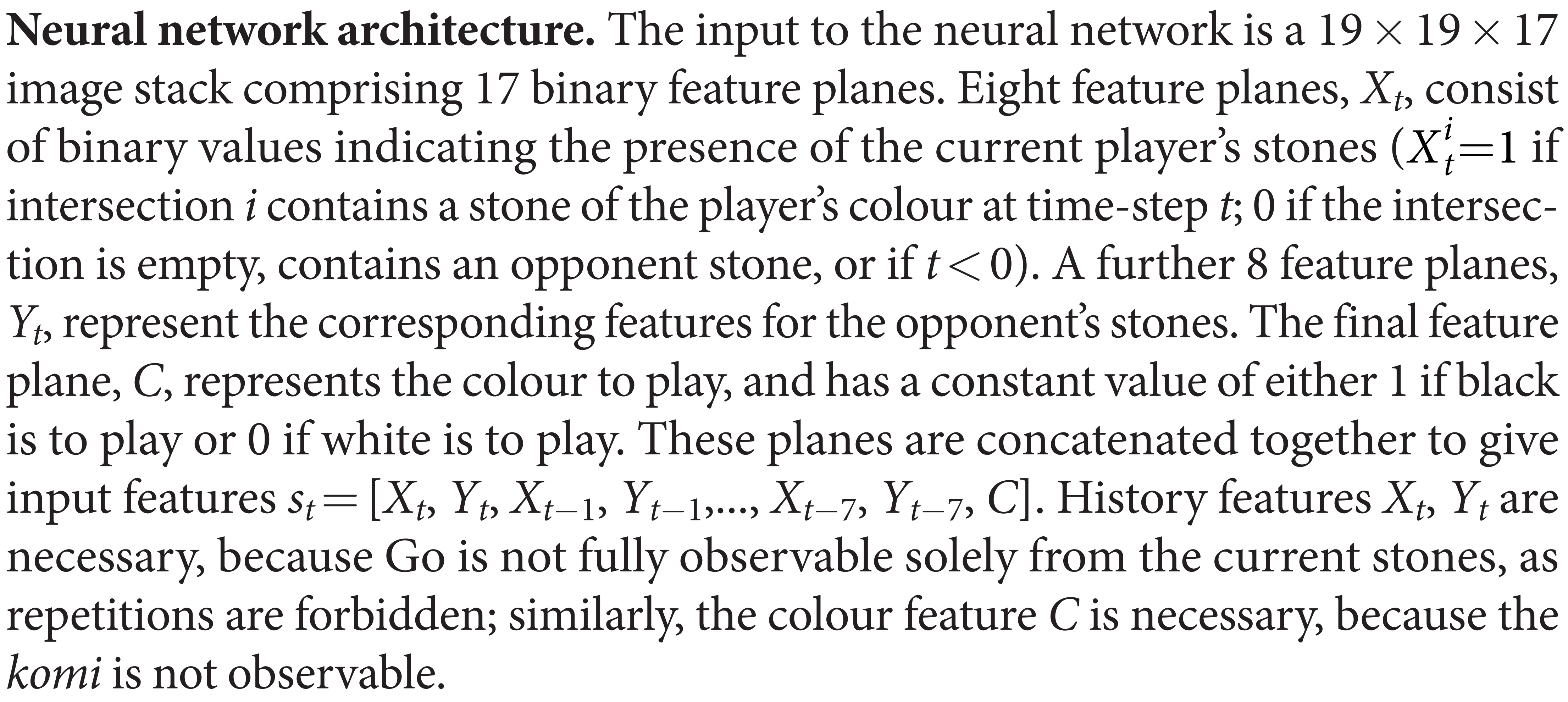
*The outputs are mixed with varying **weights**, shown by the thickness of the lines.*

That's better

Colors shows data, neuron and weight values.



$$\mathbb{R}^n = \mathbb{R}^{19\times19\times17}$$



Neural network architecture. The input to the neural network is a $19 \times 19 \times 17$ image stack comprising 17 binary feature planes. Eight feature planes, X_t, consist of binary values indicating the presence of the current player's stones ($X_t^i = 1$ if intersection i contains a stone of the player's colour at time-step t; 0 if the intersection is empty, contains an opponent stone, or if $t < 0$). A further 8 feature planes, Y_t, represent the corresponding features for the opponent's stones. The final feature plane, C, represents the colour to play, and has a constant value of either 1 if black is to play or 0 if white is to play. These planes are concatenated together to give input features $s_t = [X_t, Y_t, X_{t-1}, Y_{t-1}, \ldots, X_{t-7}, Y_{t-7}, C]$. History features X_t, Y_t are necessary, because Go is not fully observable solely from the current stones, as repetitions are forbidden; similarly, the colour feature C is necessary, because the *komi* is not observable.

D. Silver et al "Mastering the game of Go without human knowledge", Methods.

The input features s_t are processed by a residual tower that consists of a single convolutional block followed by either 19 or 39 residual blocks[4].

The convolutional block applies the following modules:

(1) A convolution of 256 filters of kernel size 3×3 with stride 1

(2) Batch normalization[18]

(3) A rectifier nonlinearity

Each residual block applies the following modules sequentially to its input:

(1) A convolution of 256 filters of kernel size 3×3 with stride 1

(2) Batch normalization

(3) A rectifier nonlinearity

(4) A convolution of 256 filters of kernel size 3×3 with stride 1

(5) Batch normalization

(6) A skip connection that adds the input to the block

(7) A rectifier nonlinearity

The output of the residual tower is passed into two separate 'heads' for computing the policy and value. The policy head applies the following modules:

(1) A convolution of 2 filters of kernel size 1×1 with stride 1

(2) Batch normalization

(3) A rectifier nonlinearity

(4) A fully connected linear layer that outputs a vector of size $19^2 + 1 = 362$, corresponding to logit probabilities for all intersections and the pass move

The value head applies the following modules:

(1) A convolution of 1 filter of kernel size 1×1 with stride 1

(2) Batch normalization

(3) A rectifier nonlinearity

(4) A fully connected linear layer to a hidden layer of size 256

(5) A rectifier nonlinearity

(6) A fully connected linear layer to a scalar

(7) A tanh nonlinearity outputting a scalar in the range $[-1, 1]$

The overall network depth, in the 20- or 40-block network, is 39 or 79 parameterized layers, respectively, for the residual tower, plus an additional 2 layers for the policy head and 3 layers for the value head.

D. Silver et al "Mastering the game of Go without human knowledge", Methods.

Appendix: Ko rule

Diagram 12

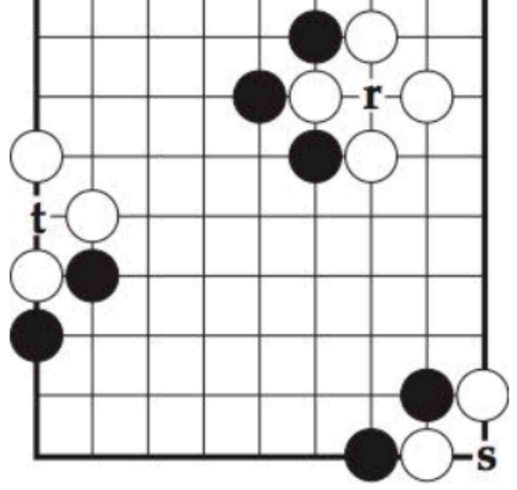
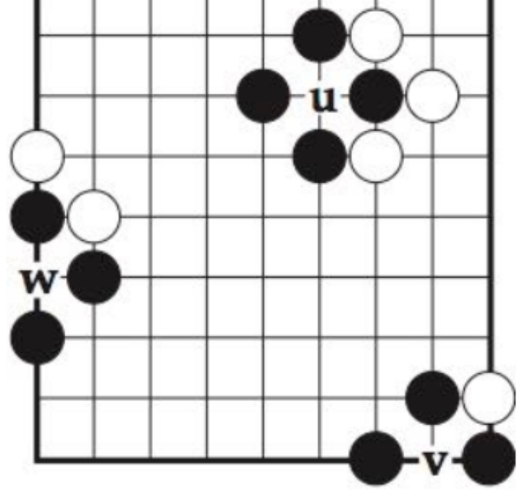
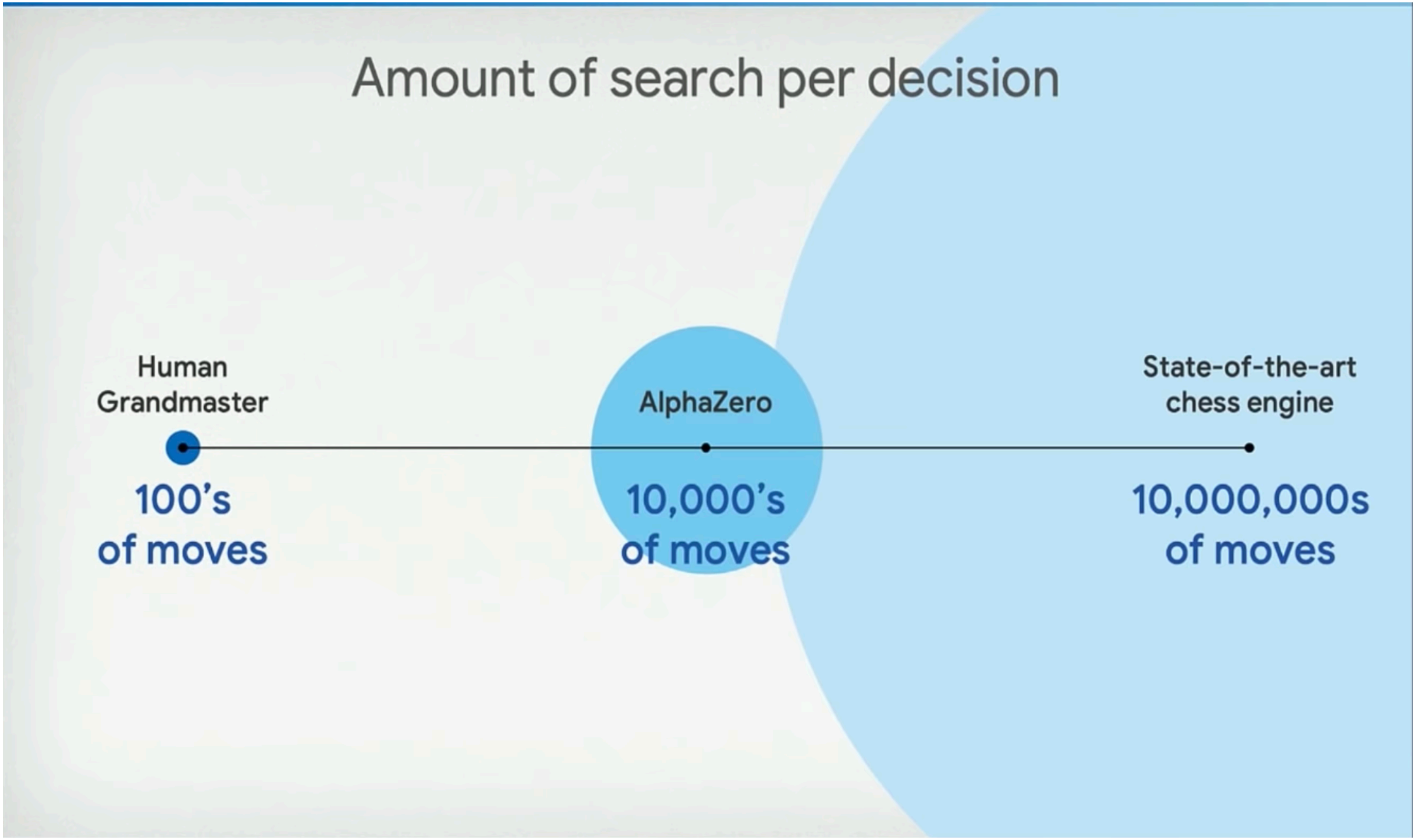


Diagram 13



The *ko* rule

At the top of **Diagram 12**, Black can capture a stone by playing at **r**. This results in the situation at the top of **Diagram 13**. However, this stone is itself vulnerable to capture by a White play at **u** in **Diagram 13**. If White were allowed to recapture immediately at **u**, the position would revert to that in **Diagram 12**, and there would be nothing to prevent this capture and recapture continuing indefinitely. This pattern of stones is called **ko** - a Japanese term meaning eternity. Two other possible shapes for a ko, on the edge of the board and in the corner, are also shown in this diagram.



D. Hassabis, "The power of self-learning systems", talk MIT March 20, 2019 (link)