Aprendizado por Reforço
Deep Q-Networks: Atari e AlphaZero

Valdinei Freire

(EACH - USP)

Regressão

Em problemas de regressão, considera-se o seguinte modelo:

$$Y = f(\mathbf{X}) + \epsilon(\mathbf{X}),$$

onde $\mathbf{X}$ é um vetor de variáveis independentes observadas, Y é uma variável de resposta também observada, e ϵ é uma variável aleatória com esperança 0.

A esperança condicional de Y para um vetor dado $\mathbf{x}$ de $\mathbf{X}$ é chamada a função regressão de Y sobre $\mathbf{X}$.

Existe uma função $f : \mathbf{X} \to \mathbb{R}$ tal que a esperança condicional de Y_i dados os vetores $\mathbf{x}_1, \ldots, \mathbf{x}_n$ tem a forma $\mathrm{E}(Y_i) = f(\mathbf{x}_i)$ para todo $i = 1, \ldots, n$.

Método dos Mínimos Quadrados

Considere o conjunto de amostras de variáveis independentes $\mathbf{x}_1, \ldots, \mathbf{x}_n$, respostas correspondentes $y_1, \ldots, y_n$ e um modelo $f(\mathbf{x}; \theta)$ parametrizado em θ.

Encontre o parâmetro θ^* definido por:

$$\theta^* = \arg\max_{\theta \in \Omega} \sum_{i=1}^{n} (f(\mathbf{x}_i; \theta) - y_i)^2$$

Por Gradiente Ascendente temos:

$$\theta_{t+1} = \theta_t - \beta_t \sum_{i=1}^{n} \nabla_\theta f(\mathbf{x}_i; \theta_t)(f(\mathbf{x}_i; \theta_t) - y_i).$$

Generalização

Aproximação de função:

- Considere uma classe de funções parametrizada diferenciáveis $C : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^m \to \mathbb{R}$
- A ideia é que $m \ll |\mathcal{S}| \times |\mathcal{A}|$
- Dado vetor de parâmetros $\theta \in \mathbb{R}$ aproximamos:

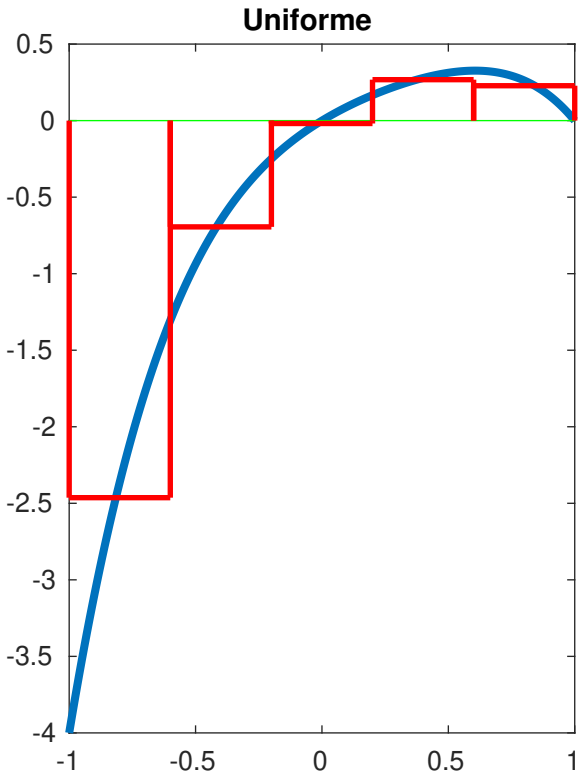
$$Q(s, a) \approxeq C(s, a; \theta)$$

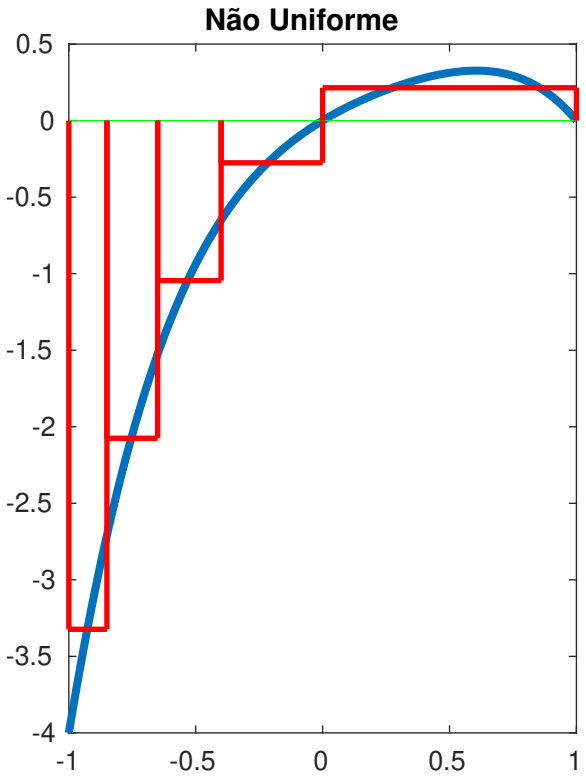
- Dado um vetor inicial de parâmetros $\theta \in \mathbb{R}^m$:

$$\begin{aligned} \delta &\leftarrow r_t + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s_t, a_t) \\ &= r_t + \gamma \max_{a' \in \mathcal{A}} C(s_{t+1}, a'; \theta) - C(s_t, a_t; \theta) \end{aligned}$$

$$\theta \leftarrow \theta + \alpha_t \delta \nabla_\theta C(s_t, a_t; \theta)$$

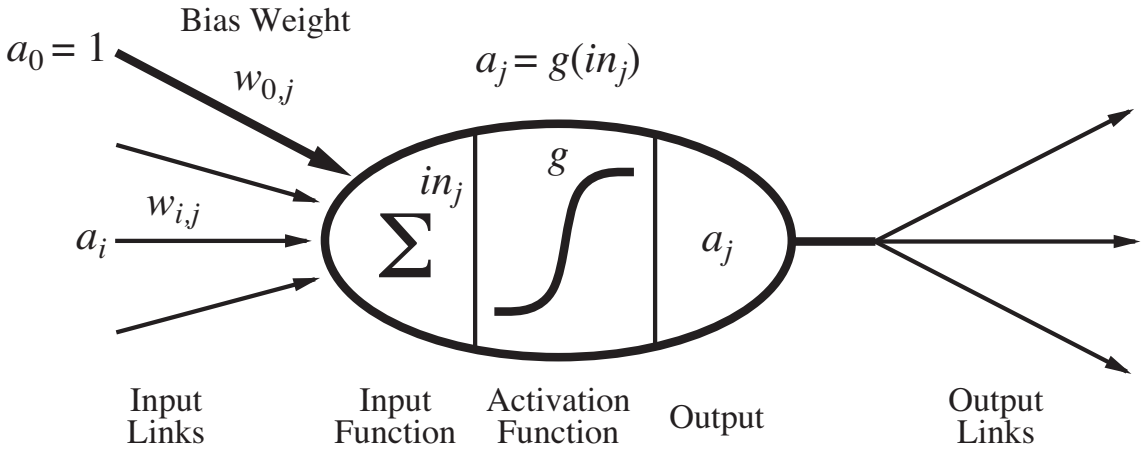
Aproximação por Grades





Perceptron: Modelo Computacional

1943: McCulloch e Pitts



Perceptron é ativado quando uma combinação linear de sua entrada atinge um *threshold*

$$a_j = g\left(\sum_{i=0}^{n} w_{i,j} a_i\right)$$

Função de Ativação

- Logito

$$g(x) = \frac{e^x}{1 + e^x}$$

- Tangente Hiperbólica

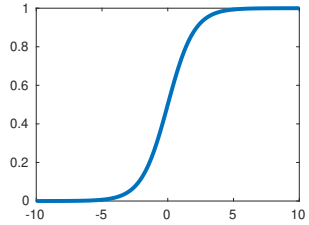
$$g(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- ReLU (Rectified Linear Unit)

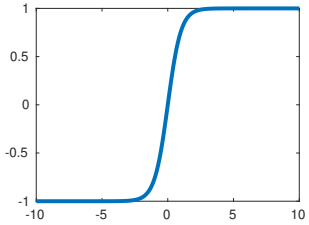
$$g(x) = \max\{0, x\}$$

Função de Ativação

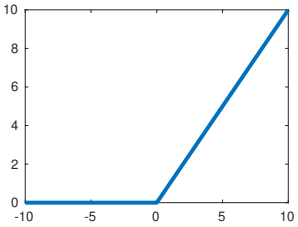
- Logito



- Tangente Hiperbólica

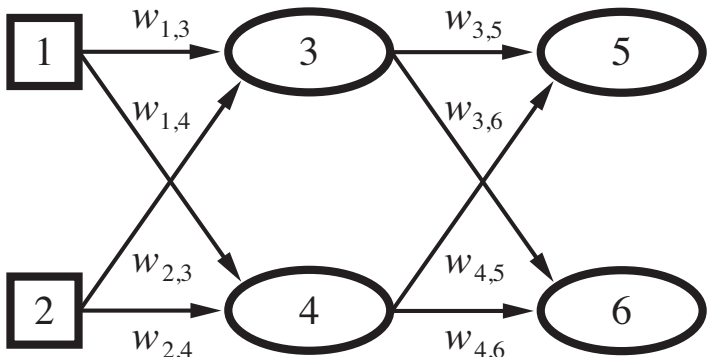


- ReLU (Rectified Linear Unit)



Multilayer Feed-Forward Network

E se a entrada de um perceptron for a saída de outros perceptrons?

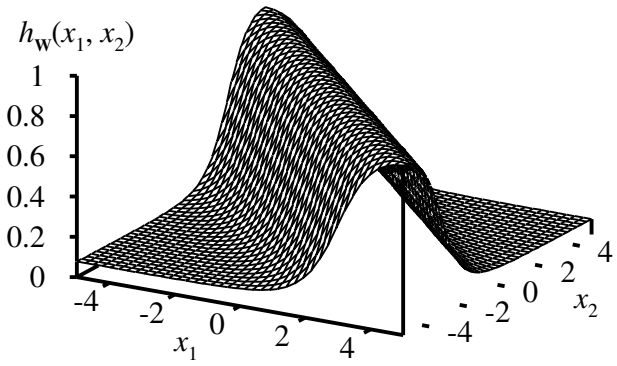


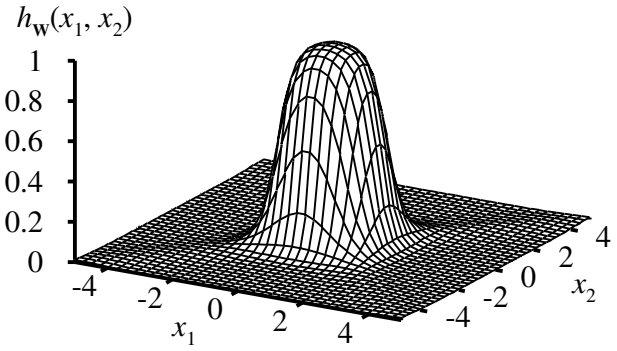
Pode-se pensar a rede organizada em várias camadas (*layers*): camadas de saída e camadas ocultas.

Aproximador de função universal

Multilayer Feed-Forward Network com uma única camada oculta e a quantidade de perceptrons adequada é um aproximador de função universal.

Função Booleana: $\frac{2^n}{n}$ perceptrons ocultos são necessários.





Aproximador de função universal

Função Contínua: Seja I_m um subconjunto compacto de $\mathbb{R}^m$, e $C(I_m)$ a classe de funções contínuas no domínio I_m. Defina:

$$F(x) = \sum_{i=1}^{N} v_i g(\mathbf{w}_i^\top \mathbf{x} + b_i),$$

onde $v_i, b_i \in \mathbb{R}$, $\mathbf{w}_i \in \mathbb{R}^m$, e $g : \mathbb{R} \to \mathbb{R}$ é uma função limitada e estritamente monotônica.

Então, dada uma função $f \in C(I_m)$ e $\varepsilon > 0$, existe inteiro N e constantes $v_i, b_i \in \mathbb{R}$ e vetores reais $\mathbf{w}_i \in \mathbb{R}^m$ tais que:

$$|F(\mathbf{x}) - f(\mathbf{x})| < \varepsilon$$

para todo $x \in I_m$.

Multilayer Feed-Forward Network

Redes acíclicas ou de alimentação direta

- Representam uma função de sua entrada atual.
- Não têm nenhum estado interno além dos pesos.

Redes cíclicas ou recorrentes.

- Utilizam suas saídas para realimentar suas entradas.
- Níveis de ativação da rede formam um sistema dinâmico.
- Podem admitir memória de curto e longo prazo.

Aprendizado: Back-Propagation

Considere a perda de um único par de exemplo $(\mathbf{x}, \mathbf{y})$:

$$Loss(h_{\mathbf{w}}) = (\mathbf{y} - h_{\mathbf{w}}(\mathbf{x}))^2 = \sum_{k=1}^{K} (y_k - (h_{\mathbf{w}}(\mathbf{x}))_k)^2 = \sum_{k=1}^{K} (Err_k)^2$$

Busca na direção do Gradiente:

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha_t \nabla_{\mathbf{w}_t} Loss(h_{\mathbf{w}_t})$$

Considere o perceptron k da camada de saída e o peso na entrada j deste perceptron, e seja $\Delta_k = Err_k g'(in_k)$ temos que:

$$\frac{\partial Loss(h_{\mathbf{w}_t})}{\partial w_{j,k}} = -2Err_k \frac{\partial g(in_k)}{\partial w_{j,k}} = -2Err_k g'(in_k) a_j = -2a_j \Delta_k$$

Aprendizado: Back-Propagation

Considere o perceptron j da camada oculta e o peso na entrada i deste perceptron, com relação ao erro da k-ésima saída temos que:

$$\begin{aligned}
\frac{\partial Loss_k(h_{\mathbf{w}_t})}{\partial w_{i,j}} &= -2Err_k\frac{\partial g(in_k)}{\partial w_{i,j}} = -2(y_k - a_k)\frac{\partial g(in_k)}{\partial w_{i,j}} \\
&= -2(y_k - a_k)g'(in_k)\frac{\partial in_k}{\partial w_{i,j}} = -2\Delta_k w_{j,k}\frac{\partial a_j}{\partial w_{i,j}} \\
&= -2\Delta_k w_{j,k}\frac{\partial g(in_j)}{\partial w_{i,j}} = -2\Delta_k w_{j,k}g'(in_j)\frac{\partial in_j}{\partial w_{i,j}} \\
&= -2\Delta_k w_{j,k}g'(in_j)a_i
\end{aligned}$$

Aprendizado: Back-Propagation

Para o erro associado a todas saídas temos que:

$$\frac{\partial Loss(h_{\mathbf{w}_t})}{\partial w_{i,j}} = \sum_{k=1}^{K} -2\Delta_k w_{j,k} g'(in_j) a_i = -2\Delta_j a_i$$

onde $\Delta_j = g'(in_j) \sum_{k=1}^{K} w_{j,k}\Delta_k$.

Algoritmo: Back-Propagation

1. Inicializa todos os pesos com valores aleatórios
2. Propaga *forward* a entrada na rede para computar a saída de cada perceptron

$$a_j \leftarrow g(in_j) = g\left(\sum_i w_{i,j} a_i\right)$$

3. Propaga *backward* os deltas da camada de saída até a camada de entrada

camada de saída: $\Delta_k \leftarrow g'(in_k)(y_k - a_k)$

camada oculta: $\Delta_j \leftarrow g'(in_j) \sum_{k=1}^{K} w_{j,k} \Delta_k$

4. Atualiza todos os pesos $w_{i,j}$ na rede

$$w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta_j$$

5. Repete até algum critério de convergência

Algoritmo: Back-Propagation

Modo de Treinamento:

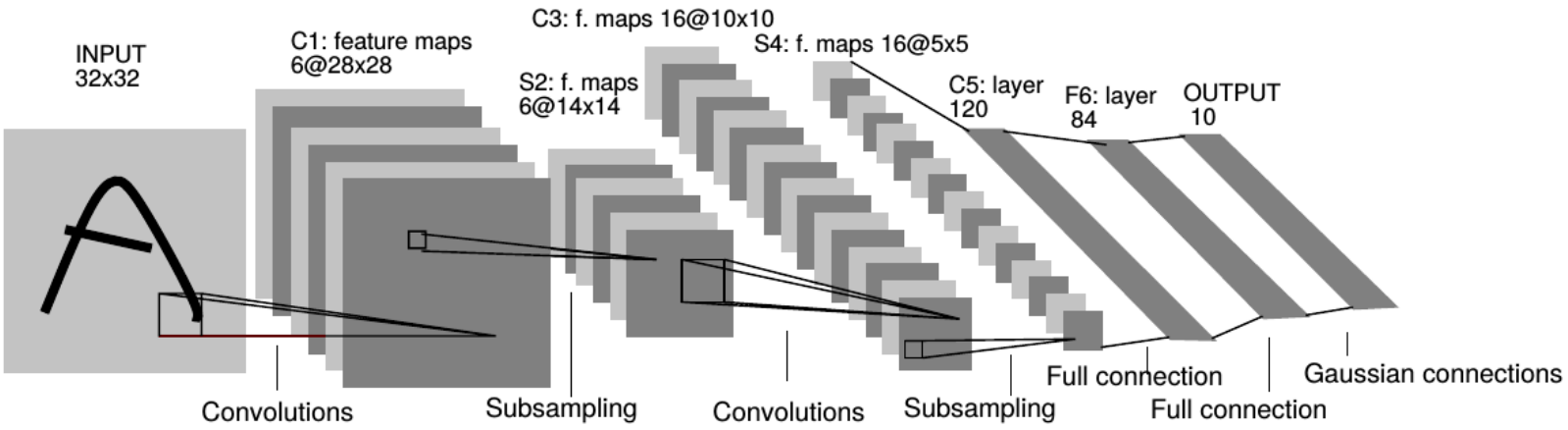
- batch (lote ou batelada)
- serial
- mini-batch

Taxa de aprendizado:

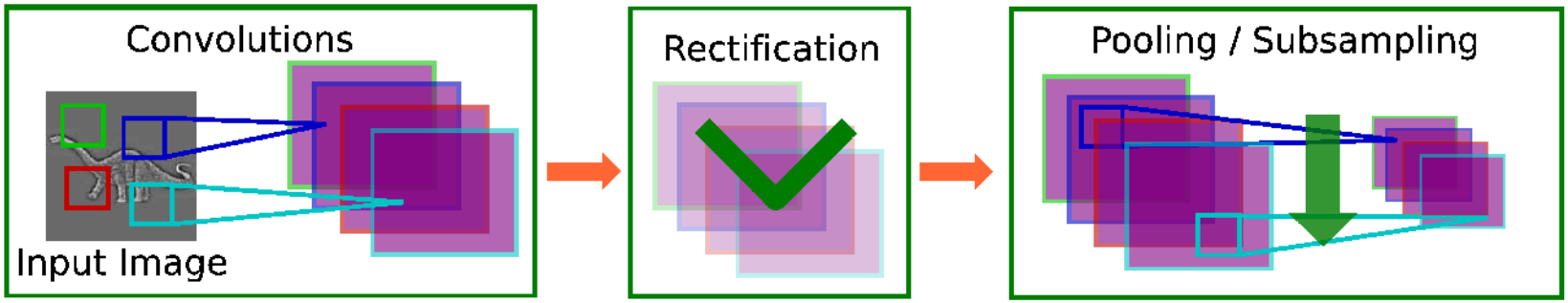
- depende da Arquitetura da rede: normalização na quantidade de pesos
- busca durante o aprendizado: decaimento e momento

Convolutional Neural Networks

Cun et. al. Gradient-based learning applied to document recognition, 1998.

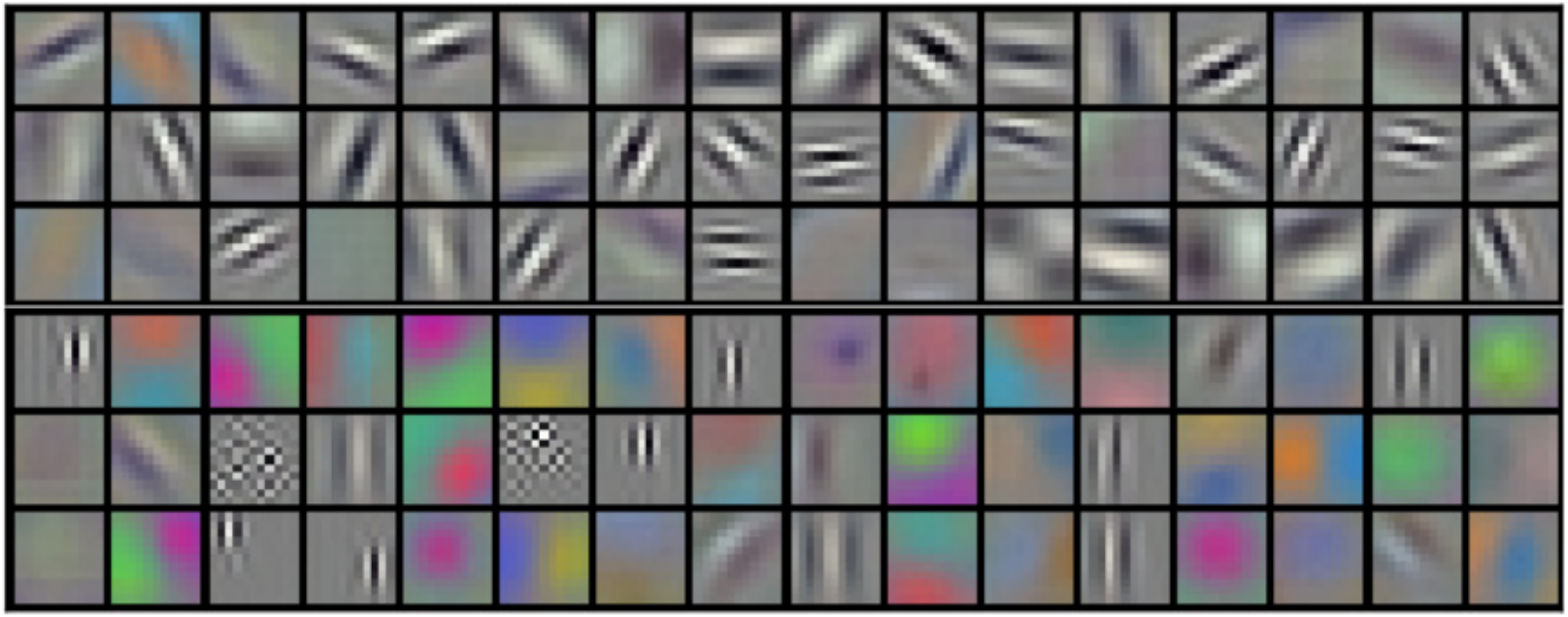


Convolutional Neural Networks



- convolution: aplica filtros para detectar texturas
- strides: como avança na imagem
- rectification: usualmente aplica-se ReLU
- pooling: diminui a quantidade de pixeis

Convolutional Neural Networks

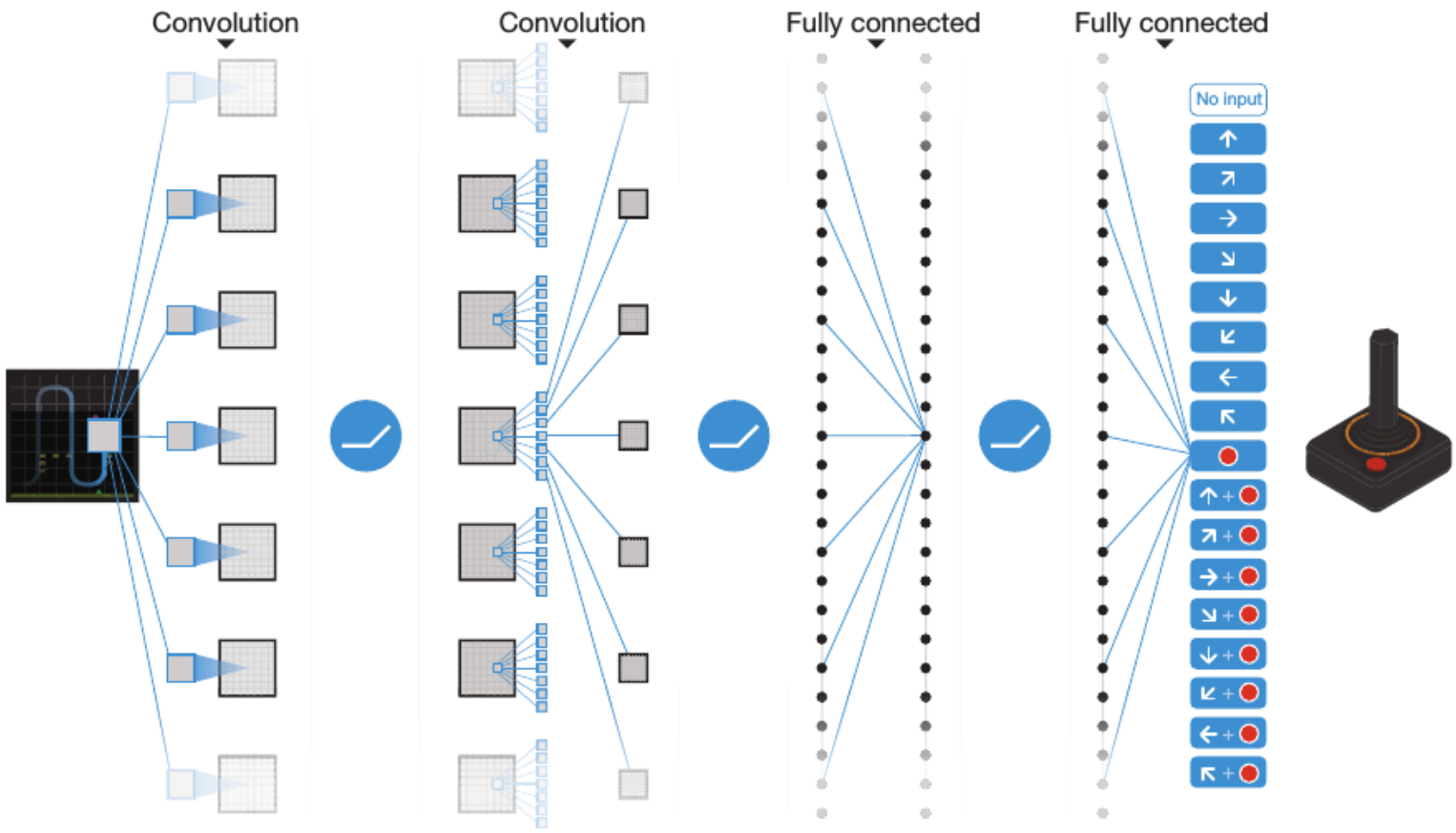


Deep Q-Networks

Mnih et. al. Human-level control through deep reinforcement learning, Nature, 2015

- rede neural convolucional + reinforcement learning
- experience replay
- double Q-learning

Deep Q-Networks



Entrada

- remoção de flickering
- considera apenas canal Y (luminância)
- reescala de 210×160 para 84×84
- estado: 4 quadros mais recentes

Arquitetura

- input: $84 \times 84 \times 4$
- camada oculta 1: 32 filtros de 8x8 com stride 4 + ReLU
- camada oculta 2: 64 filtros de 4x4 com stride 2 + ReLU
- camada oculta 3: 64 filtros de 3x3 com stride 1 + ReLU
- camada oculta 4: 512 perceptron + Relu
- output: valor Q para cada ação (entre 4 e 18)

Treinamento

- mini-batches: 32 experiências
- ϵ-greedy: decaindo linearmente entre 1 e 0.1 (10^6 quadros), depois 0.1 fixo
- treinamento com 50×10^6 quadros
- memória de replay: 10^6 quadros mais recentes
- frame-skipping: executa uma ação repetidamente por 4 quadros
- algoritmo RMSProp: taxa de aprendizado individual para cada peso

Treinamento

Estabilidade

- replays escolhidos aleatórios
- replays fora de sequência (evita correlação entre experiências)
- considera uma função Q separada e fixa para gerar exemplos de treinamento ($r + \gamma Q(s, a)$)

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

For t = 1,T **do**

With probability ε select a random action a_t

otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

Execute action a_t in emulator and observe reward r_t and image x_{t+1}

Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

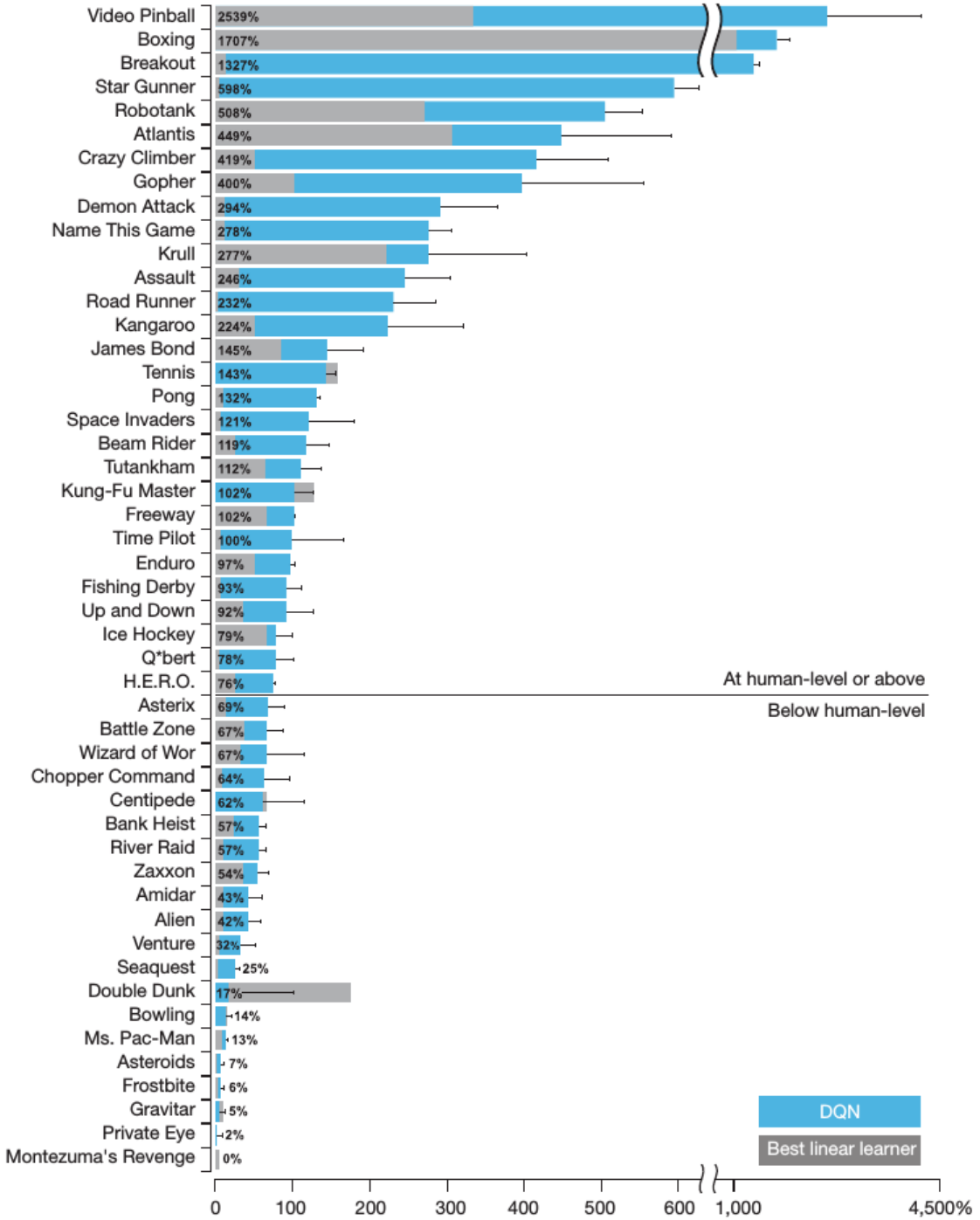
Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

Every C steps reset $\hat{Q} = Q$

End For

End For

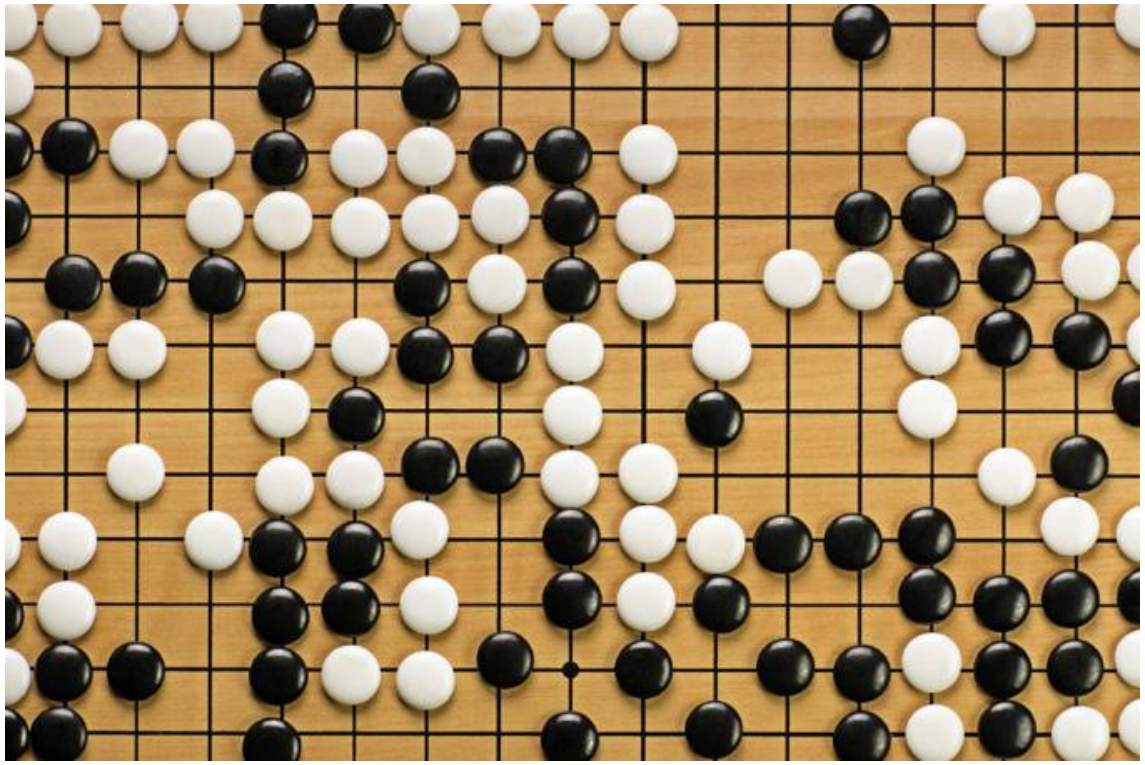

Video Pinball 2539%
Boxing 1707%
Breakout 1327%
Star Gunner 598%
Robotank 508%
Atlantis 449%
Crazy Climber 419%
Gopher 400%
Demon Attack 294%
Name This Game 278%
Krull 277%
Assault 246%
Road Runner 232%
Kangaroo 224%
James Bond 145%
Tennis 143%
Pong 132%
Space Invaders 121%
Beam Rider 119%
Tutankham 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%

AlphaGo

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

- Monte Calor Tree Search
- Aprendizado por Reforço
- Aprendizado Supervisionado

Jogo GO



- objetivo: conquistar maior parte do tabuleiro
- jogadores inserem uma peça por vez alternadamente
- pedras cercadas por pedras adversárias são capturadas

Jogos de Informação Perfeita

- Jogos de Informação Perfeita possuem uma função valor ótima v^*
- v^* representa a chance de vencer quando todos os jogadores jogam de forma ótima
- pode-se obter a função valor recursivamente utilizando minimax

Jogos de Informação Perfeita

- complexidade: b (largura) quantidade jogadas legais, d (profundidade) comprimento do jogo

- árvore de busca contém aproximadamente b^d sequências de movimentos

- Go ($b \approx 250$ e $d \approx 150$) e Xadrez ($b \approx 35$ e $d \approx 80$)

AlphaGo

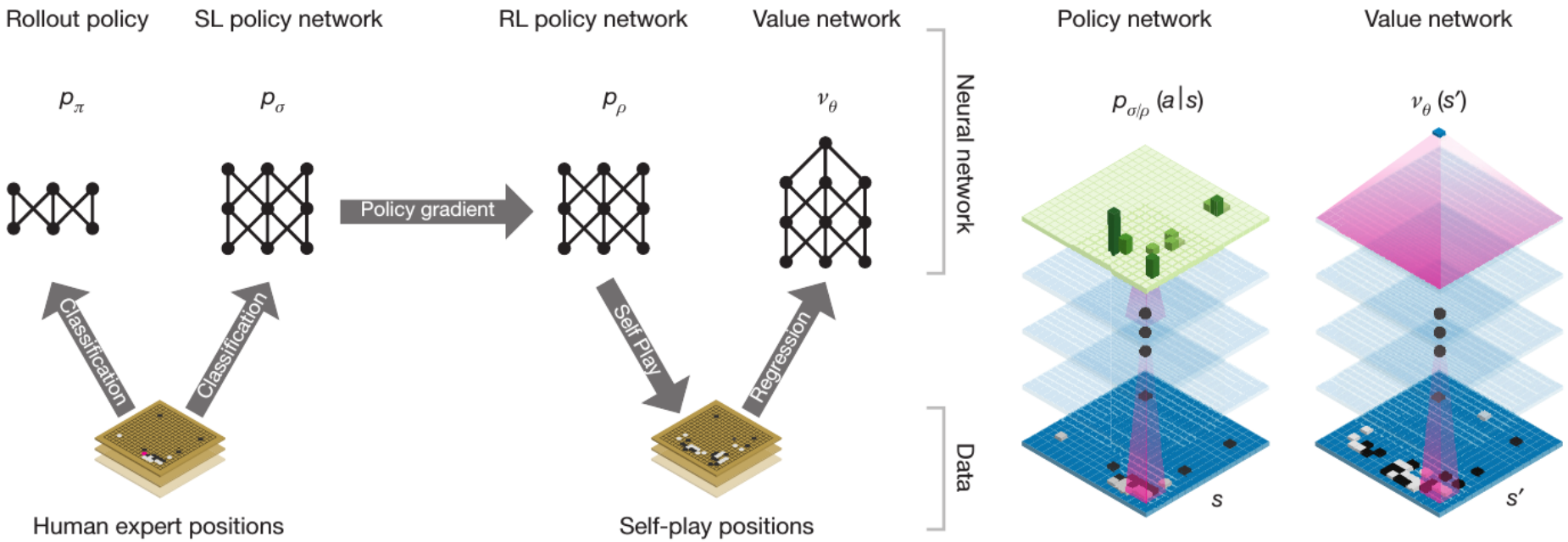
Como diminuir profundidade?

- avalia a configuração do tabuleiro
- trunca a árvore de busca, utilizando uma função valor aproximada $v(s) \approx v^*$

Como diminuir a largura?

- escolha ações de uma distribuição de probabilidades $p(a|s)$

AlphaGo - Treinamento



AlphaGo - Treinamento

Aprendizado Supervisionado

- Dados: 30 milhões de configurações do KGS Go Server com jogadas de especialistas
- p_σ: política precisa
- p_π: política rápida

AlphaGo - Treinamento

Política Precisa p_σ

- 13 camadas convolucionais e retificadores não-lineares
- uma camada softmax na saída
- entrada: várias representações do tabuleiro (localização das peças, liberdade das peças, quando as peças foram jogadas, pedras capturadas, etc.)

AlphaGo - Treinamento

Política Rápida p_π

- softmax linear
- entrada: padrões locais (salva capturas, conectado com movimento anterior, etc.)

Resultados

- p_σ: $3ms$ por ação, 57% de acurácia
- p_π: $2\mu s$ por ação, 24.2% de acurácia

AlphaGo - Treinamento

Aprendizado por Reforço

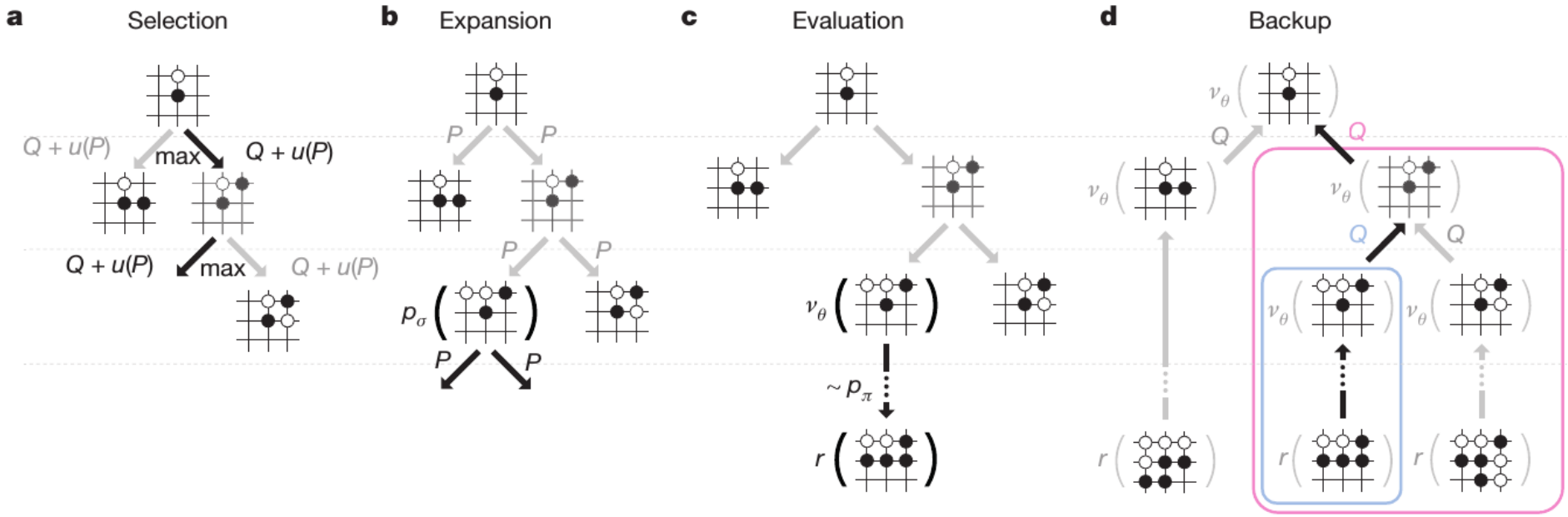
- busca da política baseado em gradiente
- p_ρ: mesma estrutura de p_σ e mesmo valor inicial
- jogos entre a política atual e alguma versão anterior escolhida aleatoriamente
- p_ρ venceu 80% dos jogos contra p_σ

AlphaGo - Treinamento

Função Valor v_θ

- aproxime $v_\theta(s) \approx v^{p_\rho}(s) \approx v^*(s)$
- arquitetura similar a arquitetura de p_ρ
- minimização do MSE
- 30 milhões de exemplos de jogos diferente gerados a partir de *self-play*

AlphaGo - Execução



AlphaGo - Execução

Monte Carlo Tree Search

- cada nó armazena: função valor $Q(s,a)$, ocorrências $N(s,a)$, e probabilidade *a priori* $P(s,a) = p_\sigma(s,a)$

- a árvore é atravessada por simulação até encontrar um nó folha s_L:

$$a_t = \arg\max_{a \in \mathcal{A}} \{Q(s_t,a) + u(s_t,a)\}$$

- função bônus:

$$u(s,a) \propto \frac{P(s,a)}{1 + N(s,a)}$$

- a partir do nó folha s_L, um *rollout* com a política p_π é executado para obter um resultado z_t

- Cada nó folha obtém o valor:

$$V(s_L) = (1 - \lambda)v_\theta(s_L) + \lambda z_L$$

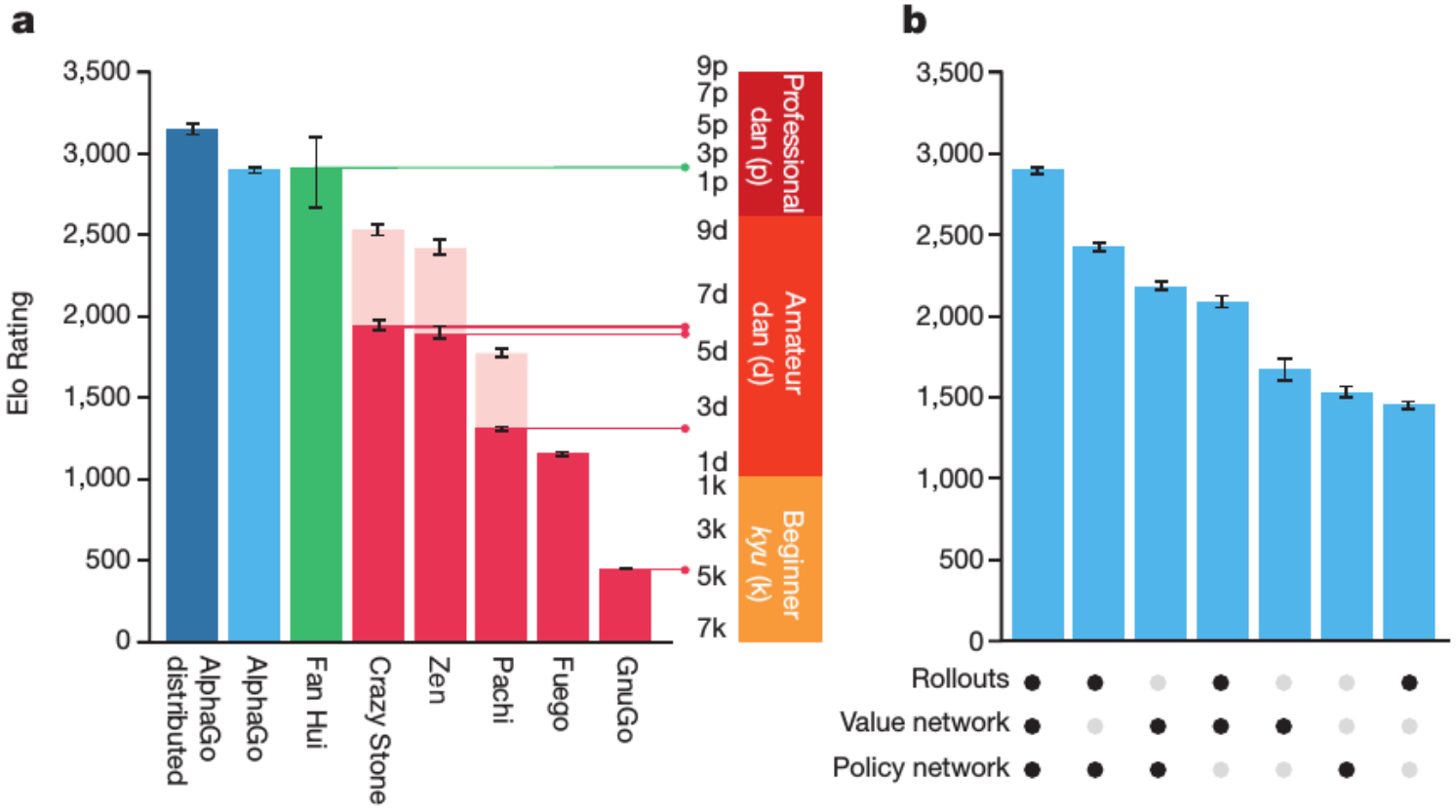
- Os valores das arestas $Q(s, a)$ visitadas são atualizados com $V(s_L)$

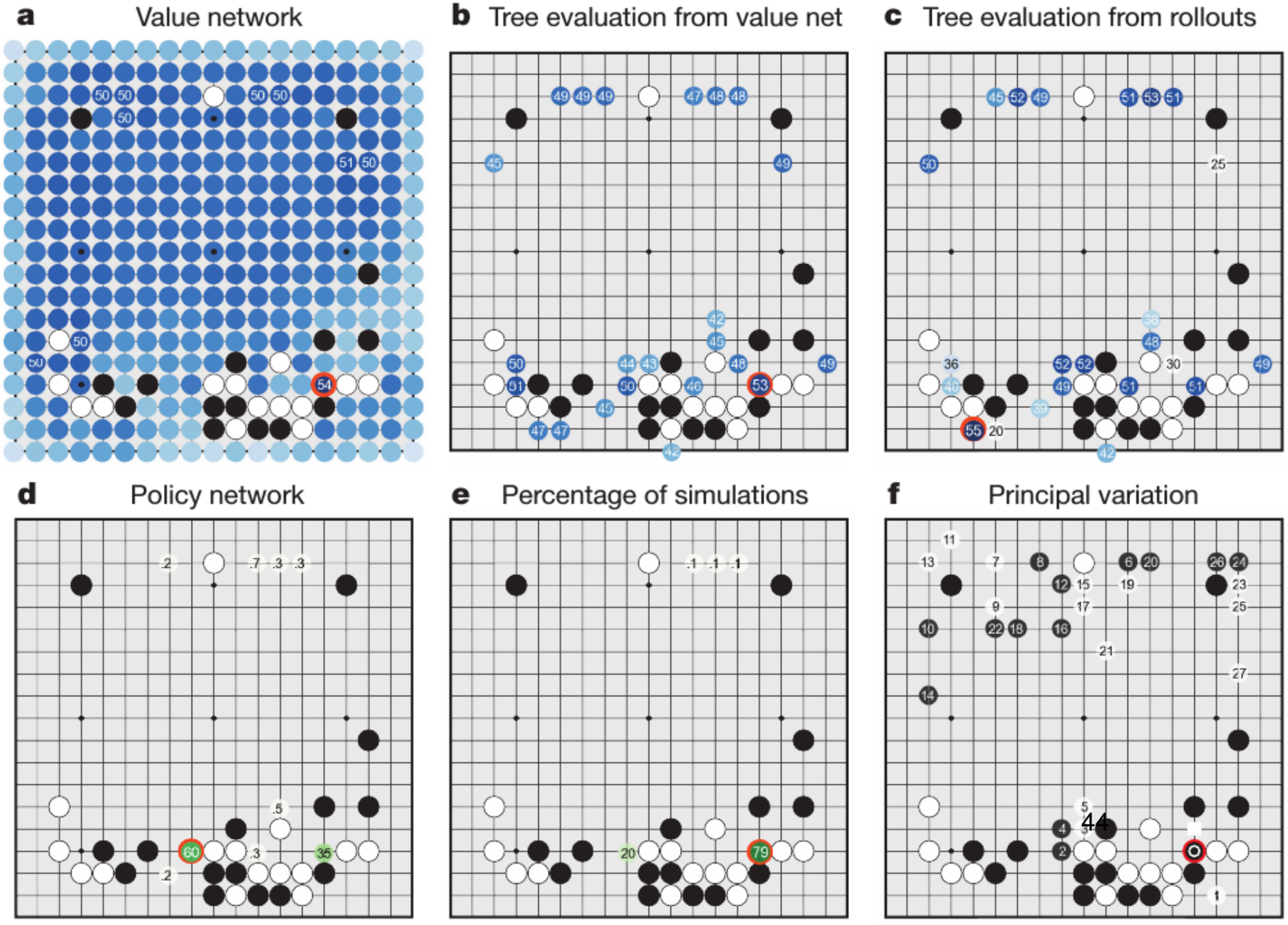
AlphaGo - Experimentos

Configuração:

- multi-thread assíncrono
- única máquina: 40 threads, 48 CPUs, e 8 GPUs
- múltiplas máquinas: 40 threads, 1202 CPUs e 176 GPUs

AlphaGo - Experimentos




a Value network
b Tree evaluation from value net
c Tree evaluation from rollouts
d Policy network
e Percentage of simulations
f Principal variation

AlphaGo Zero

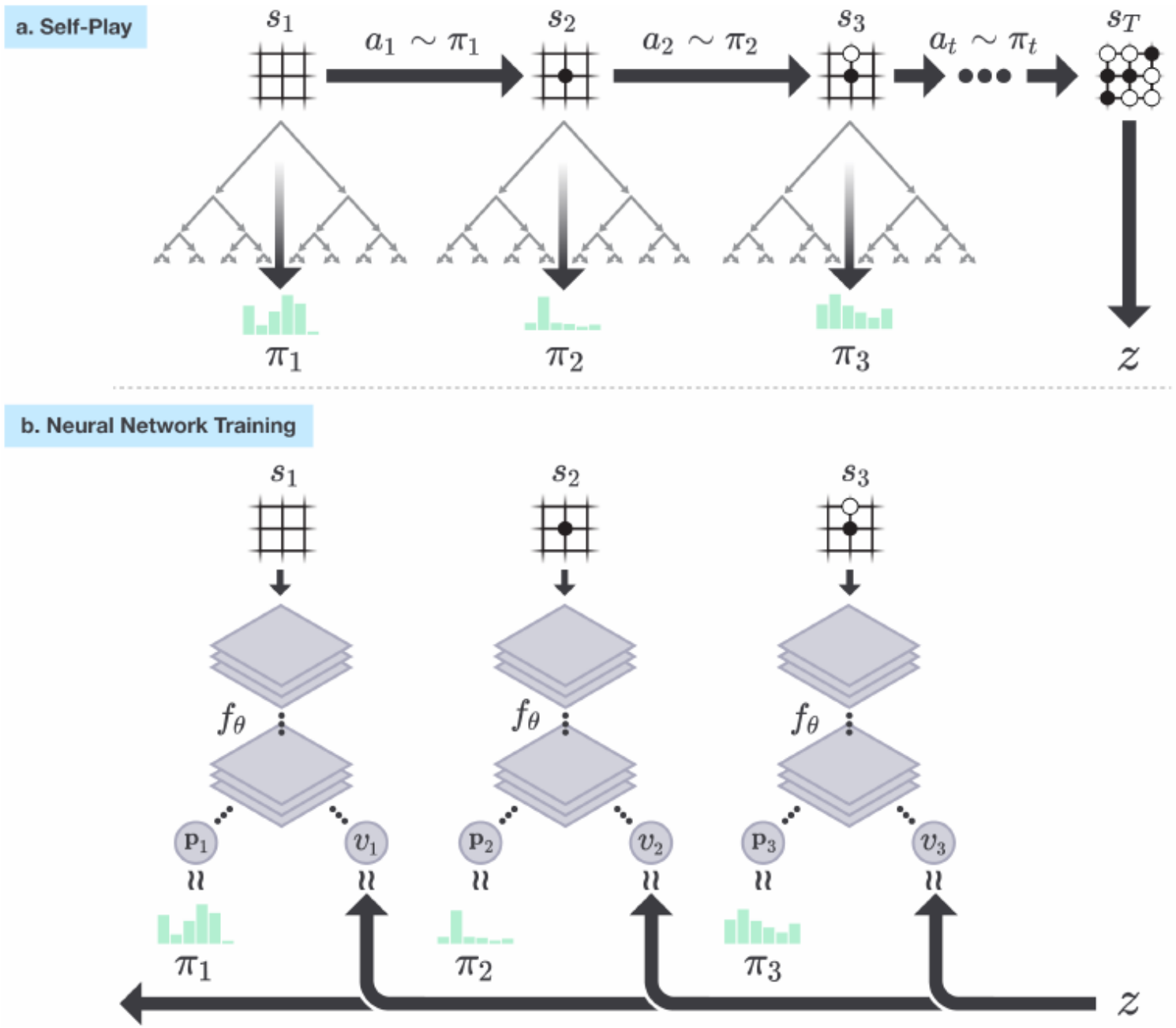
Silver et. al. Mastering the Game of Go without Human Knowledge, Nature, 2017.

- treinamento apenas por *self-play*
- apenas representação direta do tabuleiro
- uma única rede para política e função valor
- uma busca em árvore mais simples baseada apenas nessa rede

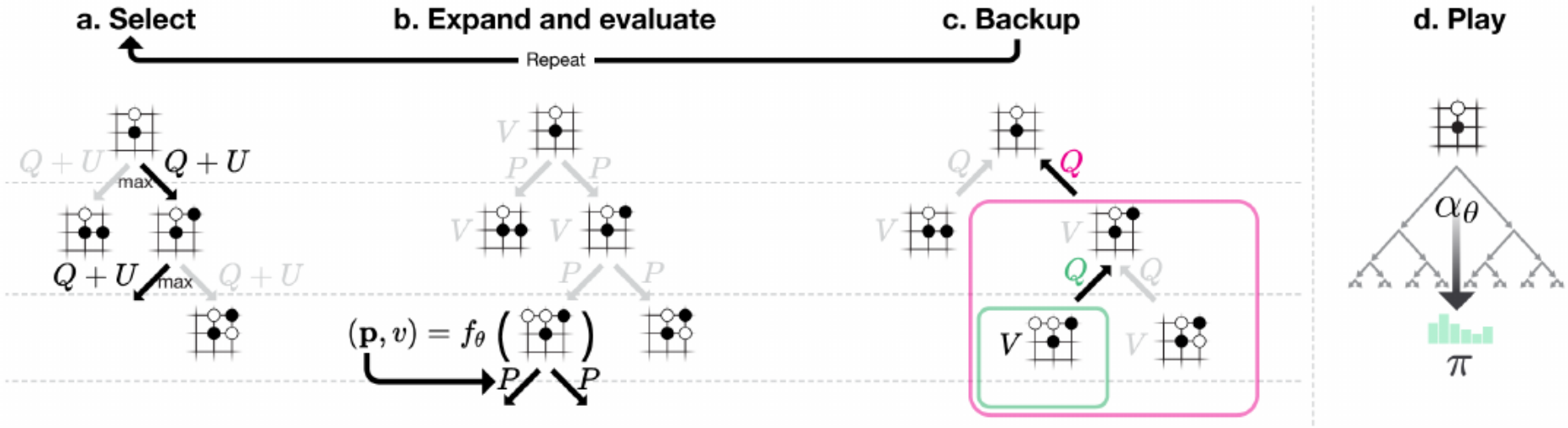
AlphaGo Zero

Novo algoritmo de aprendizado por reforço:

- incorpora busca com *lookahead* dentro do *loop* de treinamento
- melhoria mais rápida
- maior precisão
- maior estabilidade


a. Self-Play
s_1
$a_1 \sim \pi_1$
s_2
$a_2 \sim \pi_2$
s_3
$a_t \sim \pi_t$
s_T
π_1
π_2
π_3
z
b. Neural Network Training
s_1
s_2
s_3
f_θ
f_θ
f_θ
p_1
v_1
p_2
v_2
p_3
v_3
π_1
π_2
π_3
z

AlphaGo Zero - Busca



Não ocorre *rollout*, assume-se o valor $v_\theta(s_L)$ da folha

A política π é exponencialmente proporcional à frequência de ocorrência $N(s,a)$ e a temperatura τ decai com as iterações:

$$\pi(s,a) = \alpha_\theta(s,a) \propto N(s,a)^{1/\tau}$$

AlphaGo Zero - Treinamento

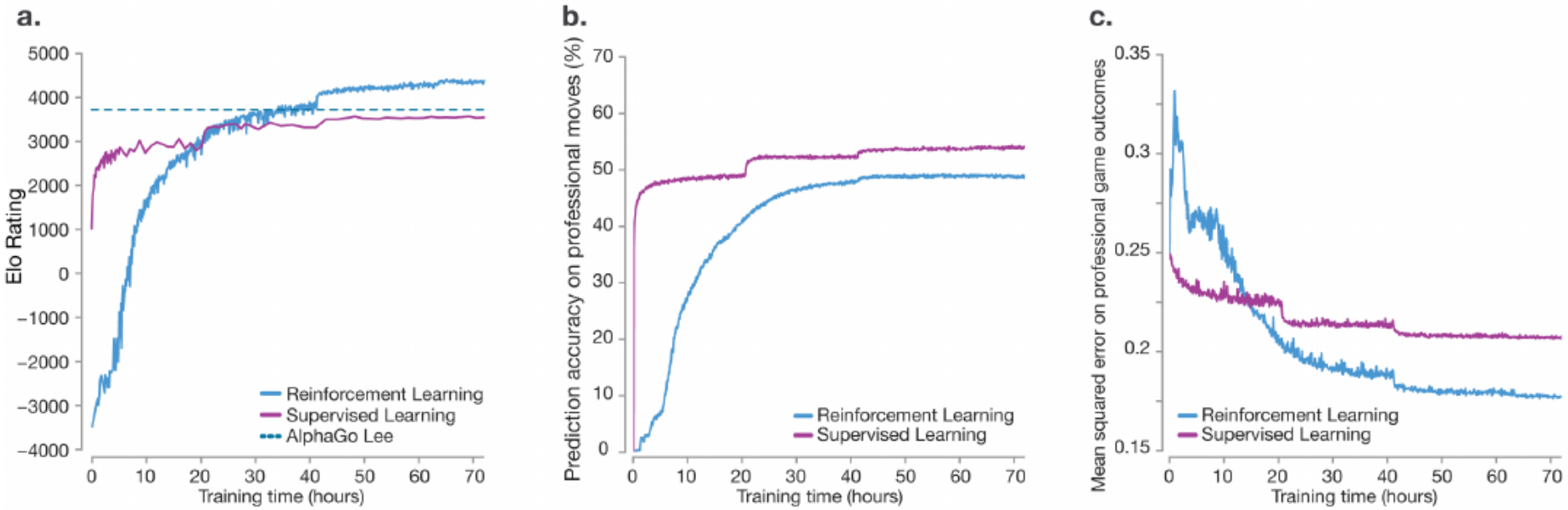
- jogos possuem um limite T de jogadas ou resignação, sempre um jogador vence
- para cada tempo t é armazenado como (s_t, π_t, z_t), sendo que $z_t = \pm r_T$
- rede neural considera 20 blocos residuais de camadas convolucionais
- dados são selecionados uniformemente dentre os passos dos últimos jogos

- função de perda

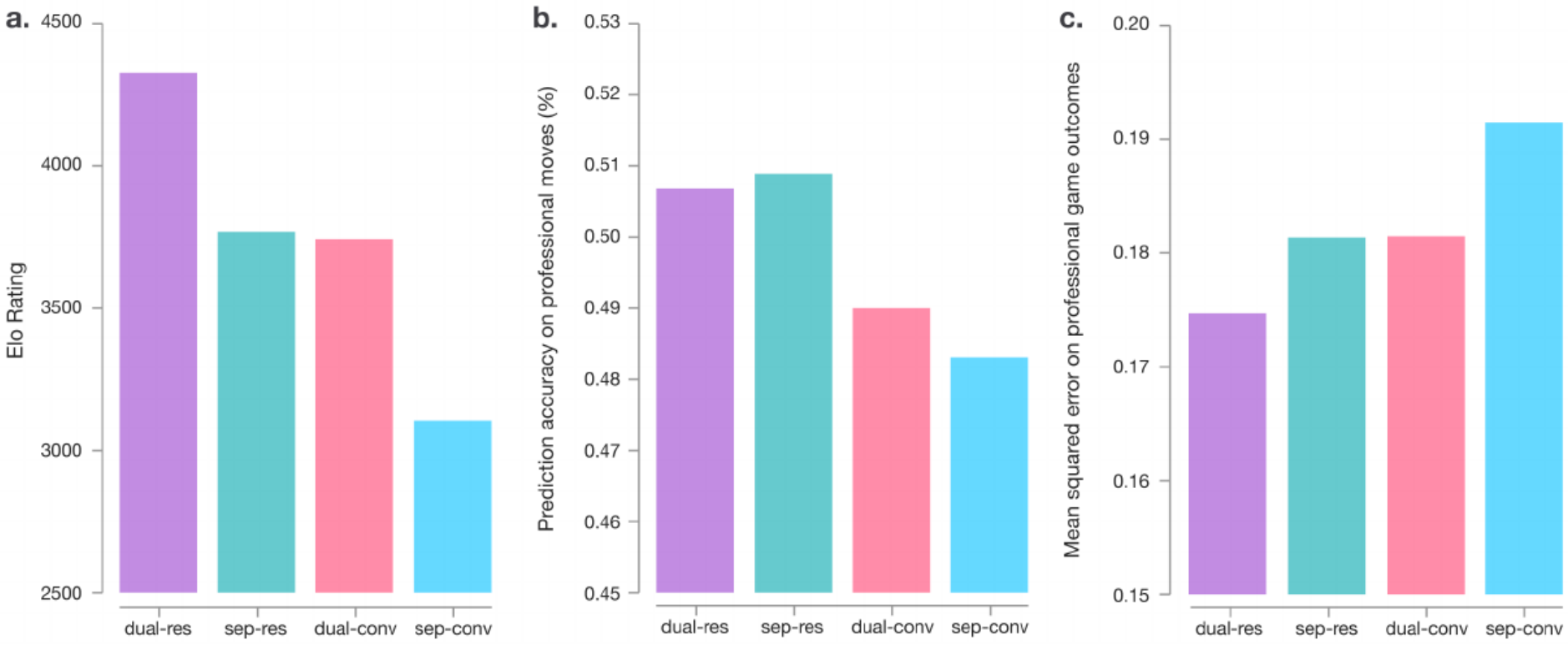
$$l = (z - v)^2 - \pi^\top \log p + c||\theta||^2$$

- o melhor θ é mantido por meio de um teste no fim de cada treinamento

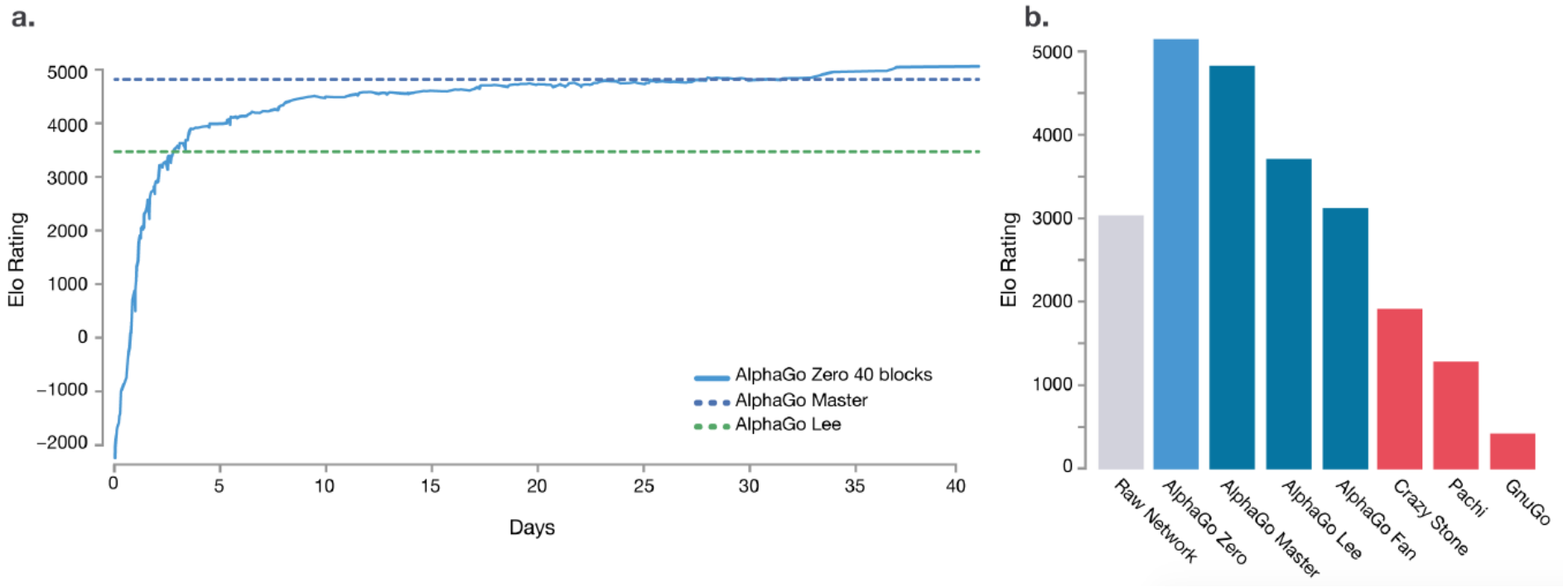
AlphaGo Zero - Experimentos



AlphaGo Zero - Experimentos



AlphaGo Zero - Experimentos



AlphaGo Zero - Experimentos

Conhecimento do domínio:

- conhecimento perfeito das regras do jogo (movimentos e decisão sobre vencedor)
- imagem estruturada do tabuleiro
- invariância de rotação, reflexão e transposição de peças

AlphaZero

Silver et. al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play, Science, 2018

- AlphaZero considera o resultado esperado, enquanto AlphaGo considera probabilidade de vencer
- Não considera nenhuma característica específica do domínio (invariância de rotação, reflexão, etc.)
- AlphaZero mantém uma única rede neural que é atualizada continuamente