Reinforcement Learning

Abe Vos

Mei 2022

Vorige keer

Model-free prediction/control

	Signaal	Evaluatie	Controle
DP	$p(s', r\|s, a)$	Policy Evaluation	Policy/Value Iteration
MC	G	MC Prediction	MC Control
TD	R_t of G_t^λ	TD(0), TD(λ)	Sarsa, Q-learning

Kwaliteitsfunctie

- $Q(s, a)$
- Verwachte beloning voor actie a in staat s
- Beleid: $\pi(s) = \arg\max_a Q(s, a)$
- Tabel met een rij voor elke (s, a)

Tabulaire Q

- Q-waarde voor alle staten en acties $\mathcal{S} \times \mathcal{A}$
- Heel groot voor staten met veel dimensies
- *Discrete* staten/acties

Kwantisatie

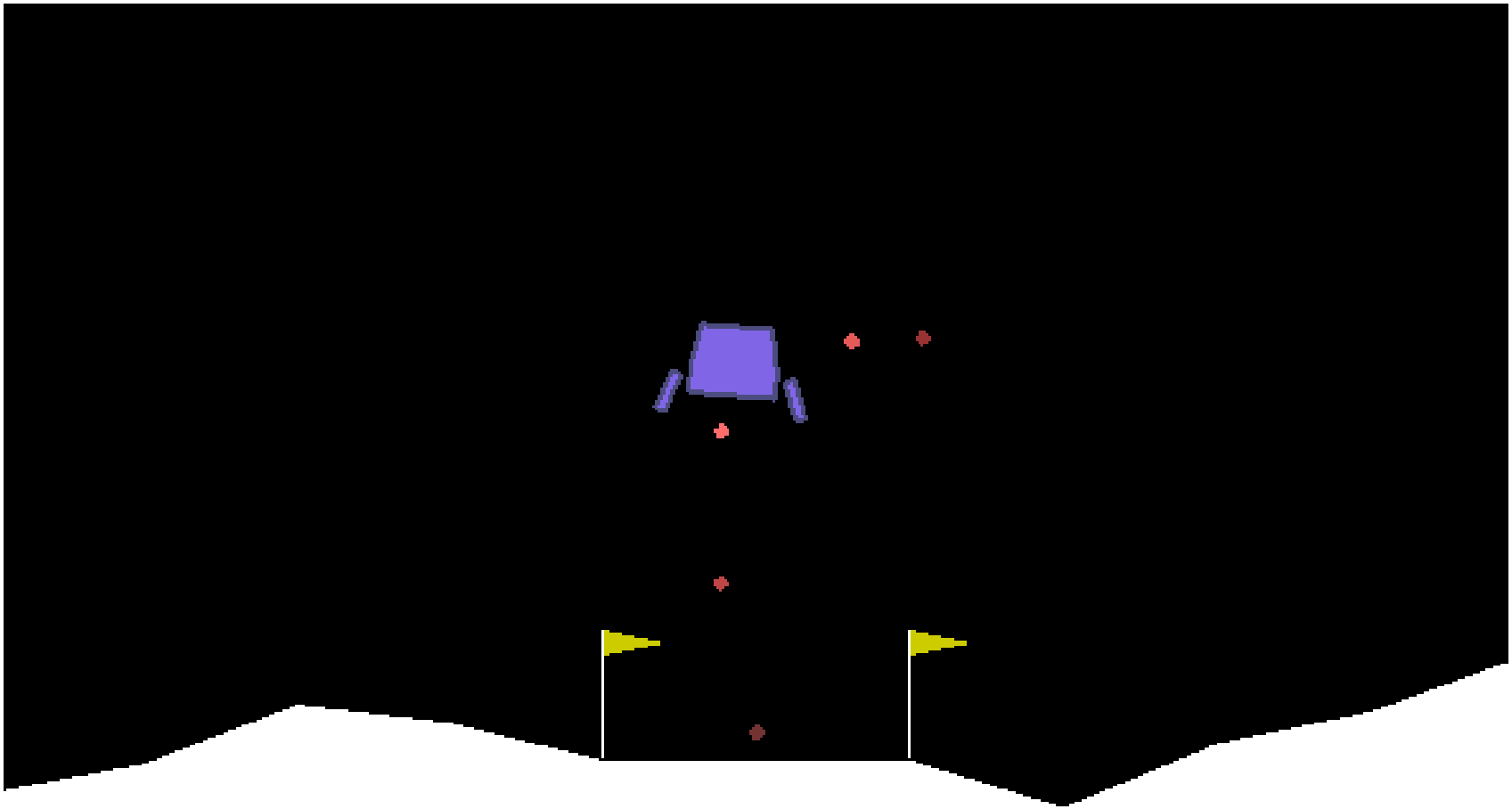


Figure 1: Lunar Lander

Lunar Lander

- Statenruimte van acht dimensies:
  - Positie (x, y)
  - Snelheid (x, y)
  - Hoek en draaisnelheid
  - Staan de poten wel/niet op de grond? (2 booleans)
- Vier acties:
  - Links, rechts, omhoog
  - Doe niets (omlaag)

Continue staten

- Tussen 0 en 1 zijn oneindig veel decimale getallen
- Lunar Lander gebruikt 6 continue staat variabelen
- Oneindig grote tabel voor $V(S)/Q(S, A)$?

Kwantisatie

- Deel continue staat op in discrete secties
- Afronden
- Hoeft niet per se op hele waarden

Afronden

- Bepaal op hoeveel decimalen we willen afronden, bijv.: 2
- Vermenigvuldig een continue waarde om decimalen links van punt te krijgen
  - $0.1736 * 10^2 = 17.36$
- Gooi de overgebleven decimalen weg

Afronden in Python

```
def quantize(x, decimals=2):
    return int(x * 10 ** decimals)
```

Tile coding

- Kwantisatie legt een raster over de statenruimte
  - Schaal van raster bepaalt “grofheid” van afrondingen
- Gebruik meerdere rasters
  - Verschuif, draai en schaal elk raster
- Rasters hoeven niet uit vierkanten te bestaan

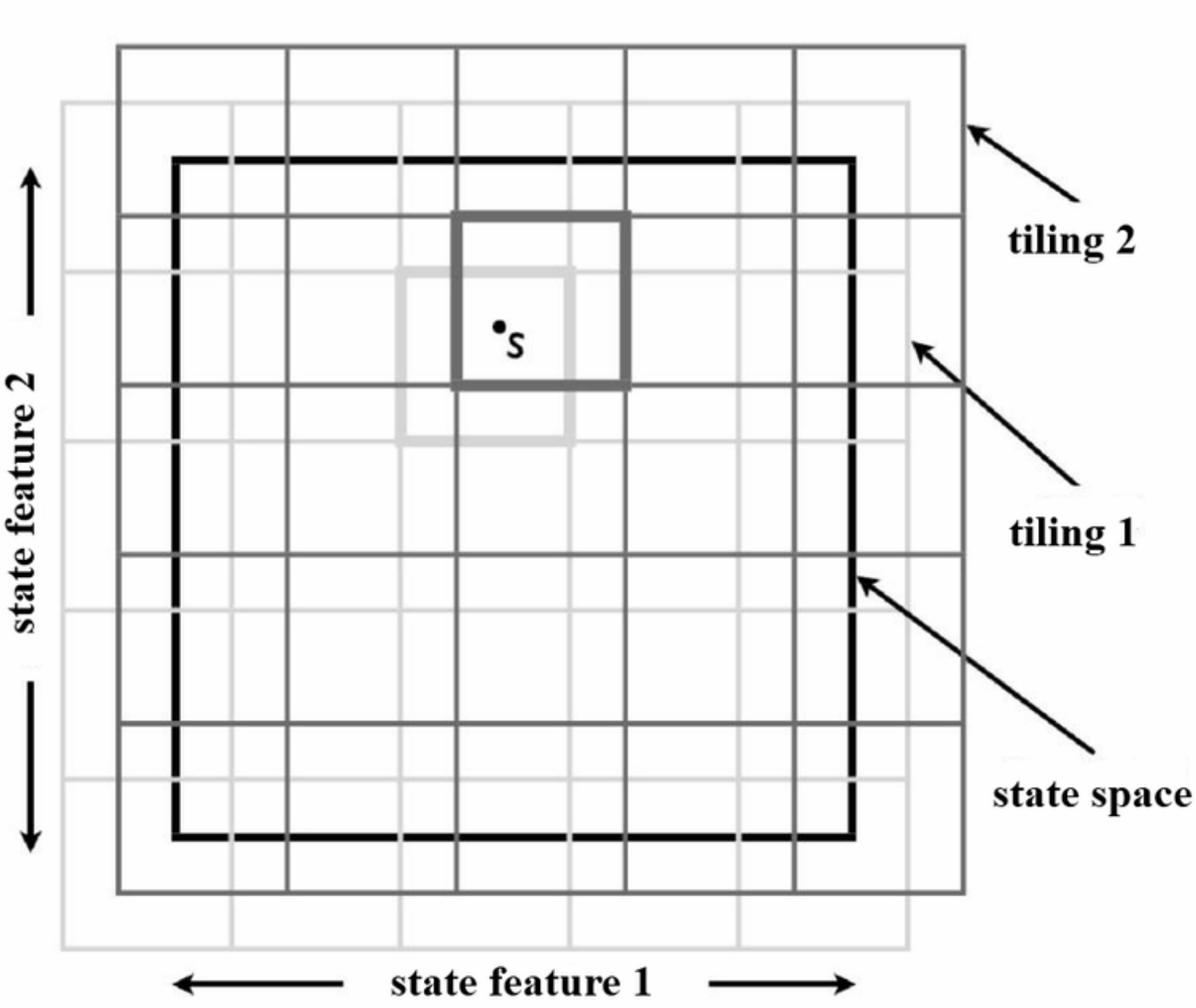


Figure 2: Voorbeeld van tile coding

De vloek der dimensies

- Tile coding maakt staten telbaar
- Lunar lander posities:
  - `x`: -400.0 tot 400.0
  - `y`: -300.0 tot 300.0
- Afronden op tientallen geeft $80 \cdot 30 = 2400$ staten
  - Met andere variabelen wordt dit nog veel groter

Grote tabulaire functies

- Veel geheugen
  - Elke staat heeft een parameter
- Elke staat moet apart bezocht en geupdate worden
  - Trainen wordt heel traag

Functie benadering

Wat we willen

- Functie om kwaliteit voor elke actie te voorspellen a.d.h.v. huidige staat en parameters
- #Parameters $\ll$ #staten
- Update voor een staat heeft invloed op "buurstaten"
  - $s \approx s' \implies \pi(s) \approx \pi(s')$

Parametrische functies

- $Q(s, a)$ is een functie van s en a
- Gebruik supervised learning
  - Q wordt een functie van s, a en $\mathbf{w}$
  - Bijvoorbeeld neuraal netwerk

Discrete acties

- Ons beleid kiest de actie met de hoogste Q-waarde
- Evalueer $Q(s, a; \mathbf{w})$ voor elke a
- Een neuraal netwerk met meerdere outputs
  - Een Q-waarde voor elke actie
  - Functie van s en $\mathbf{w}$: $Q(s; \mathbf{w})$

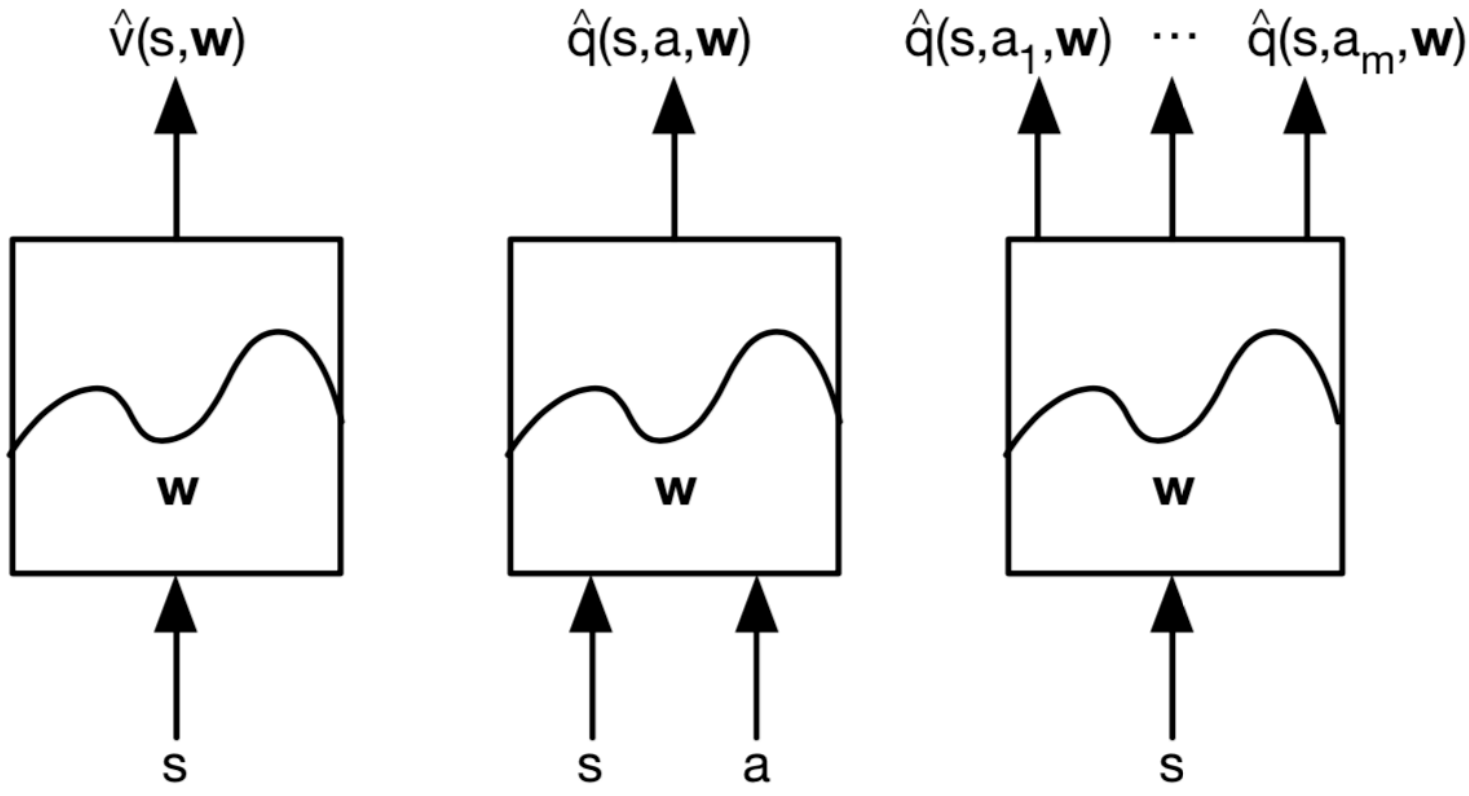


Figure 3: Type functie benaderingen

Sarsa met functiebenadering

```
Maak een differentieerbare kwaliteitsfunctie q(s, a, w)
Maak een functie voor de gradient van q(s, a, w): q'(s, a,
Kies een learning rate: lr
Initialiseer parameters w (bijvoorbeeld w=0)

Voor elke episode
    Observeer staat S en kies actie A (met bijv. epsilon gr

    Voor elke stap in de episode:
        Voer A uit, observeer R, S'

        Als S' terminaal is:
            w += lr * (R - q(S, A, w)) * q'(S, A, w)
            Ga naar volgende episode

        Kies actie A' a.d.h.v. S' en q(S', a, w)
        w += lr * (R + discount * q(S', A', w) - q(S, A, w)
```

Architectuur

Neurale netwerken

- Uitkomst van een neuron h_i met een input $\mathbf{x}_n$
  - $h_i = \sigma(w_{i1}x_{n1} + w_{i2}x_{n2} + \cdots + w_{iD}x_{nD})$
  - $h_i = \sigma(\mathbf{w} \cdot \mathbf{x}_n)$
  - $\sigma(\cdot)$ is een activatiefunctie
- Herhaal voor alle verborgen neuronen
- Gebruik $h_1, \cdots, h_K$ als input voor volgende laag

Problemen met neurale netwerken

- Neurale netwerken zijn krachtige functie benaderaars
- Veel parameters $\rightarrow$ veel manieren om een probleem op te lossen

Overfitting

- Krachtige modellen leren alle details van de training data
  - Kunnen slecht generaliseren
- Student met fotografisch geheugen
  - Bereidt examen voor door alle oude examens te onthouden
  - Kan geen nieuwe vragen beantwoorden
- Student met normaal geheugen
  - Moet onderliggende principes leren
  - Kan geleerde kennis op nieuwe problemen toepassen

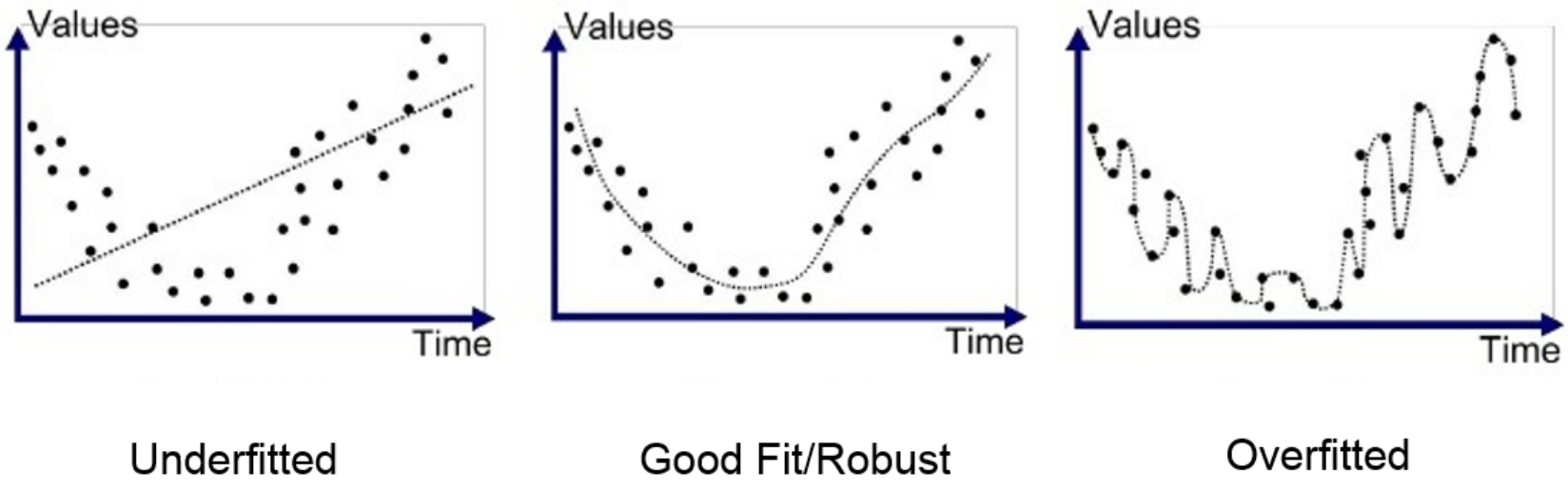


Figure 4: Overfitting en underfitting

Lokale optima

- Neurale netwerken zijn non-linear
- Error functie is *concaaf*
  - Heeft meerdere *lokale* optima
- Globaal optimimum is niet altijd makkelijk te vinden

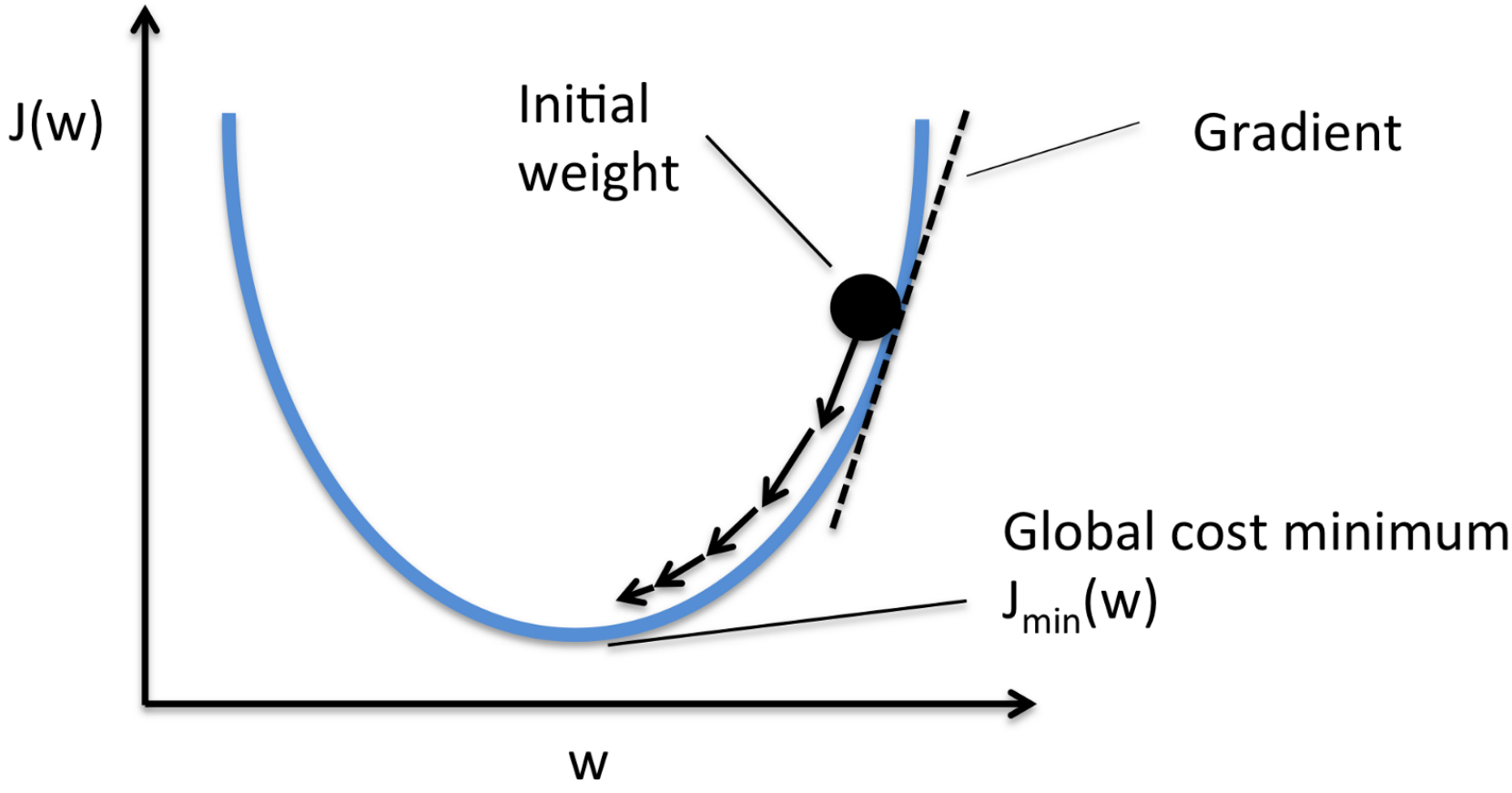
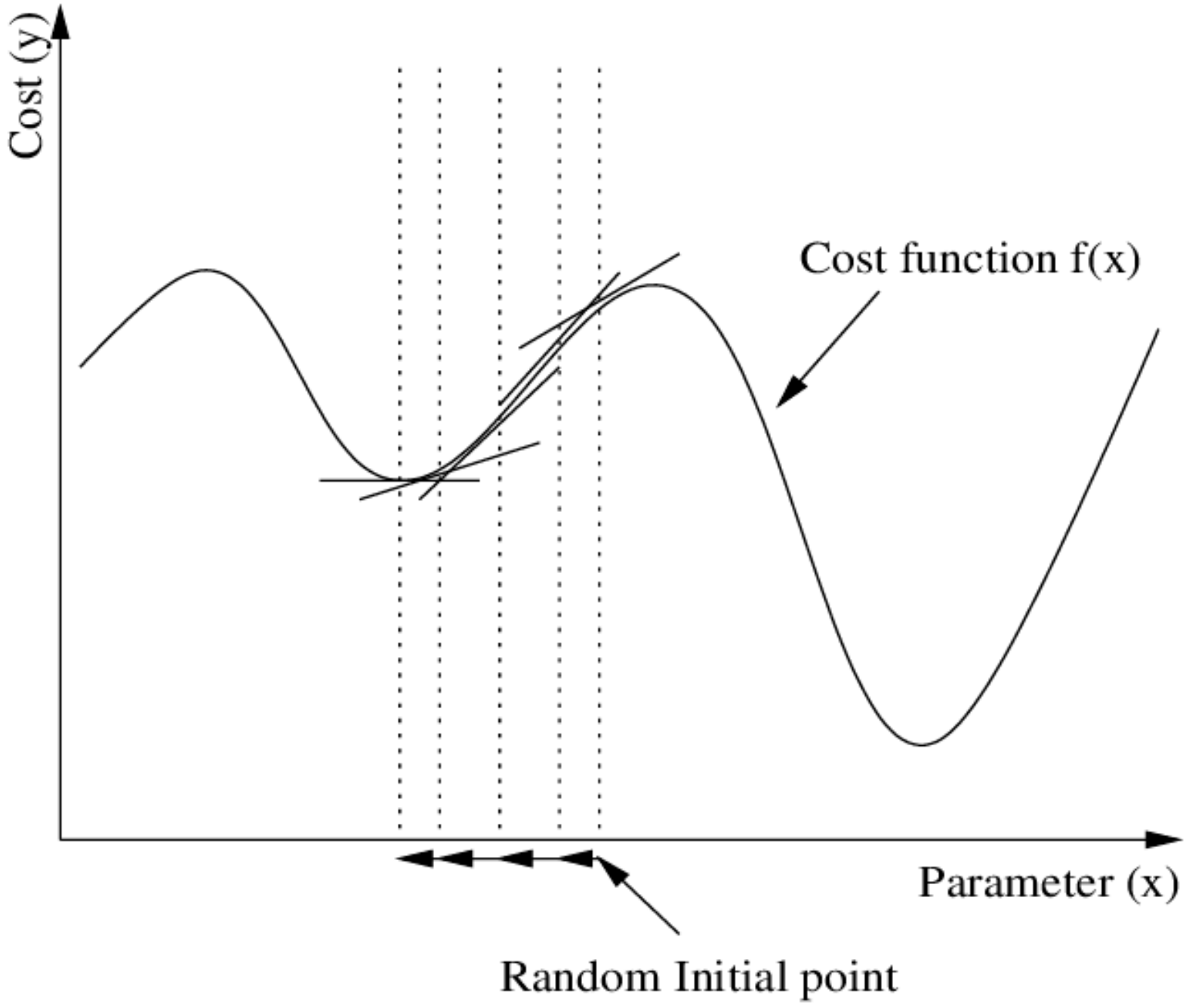


Figure 5: Gradient descent in een convexe functie


Cost (y)
Cost function f(x)
Parameter (x)
Random Initial point

Lineaire voorspeller

- Neuraal netwerk met 1 laag
  - Geen activatie functie
- Heeft maar 1 optimum
  - Is dus altijd globaal optimum
- Makkelijke afgeleide:
  - $f(\mathbf{x}; \mathbf{w}) = \mathbf{w} \cdot \mathbf{x}$
  - $\nabla_{\mathbf{w}} f(\mathbf{x}; \mathbf{w}) = \mathbf{x}$
- Kan geen non-lineaire verbanden leren

Feature engineering

Lineare functies

- Weinig parameters, weinig overfitting
- Zal voor veel situaties wel underfitten
- Geef model zelf non-lineaire informatie d.m.v. features

Simpele features

- Een staat met features $s_1, s_2, \cdots, s_D$
- Maak nieuwe features:
  - $\phi_{dd} = s_d^2$
  - $\phi_{cd} = s_c s_d$
- Parameters voor oorspronkelijke en nieuwe features
  - $\mathbf{w} = w_1, \cdots, w_D, w_{D+1}, \cdots, w_{D+D^2}$
- Bereken output:
  - $Q(s; \mathbf{w}) = \sum_{d=1}^{D} w_d s_d + \sum_{c=1}^{D} \sum_{d=1}^{D} w_{cd} \phi_{cd}$
- Non-lineaire features, functie is nog steeds lineair in *parameters*

Andere features

- Tile-coding
  - Elke tegel is een feature met waarde 0 of 1
- Radial basis function
  - Verdeel K punten door de statenruimte $m_1, \cdots, m_K$
  - Bereken de afstand van s tot elk punt:
    - $\phi_k(s) = \sqrt{\sum_{d=1}^{D}(s_d - m_{kd})^2}$
    - Pythagoras in D dimensies
  - Andere metrieken zijn mogelijk
  - Tile-coding met “fuzzy” grenzen

Deep Reinforcement Learning

Features leren

- Feature engineering kan tijdrovend zijn
- Vergt ook domeinkennis
- Bestaat er een algoritme om automatisch features te leren?

Terug naar neurale netwerken

- Verborgen neuronen in een neuraal netwerk zijn “geleerde” features
- Gebruike simpelere features
  - Neuraal netwerk kan non-lineaire verbanden in features vinden
  - Bijvoorbeeld pixels

Dodelijke Triade

- Methoden voor minder variantie en grotere statenruimtes
  - Bootstrapping (TD)
  - Off-policy training (Q-learning)
  - Functie benadering
- Combinatie van alle drie zorgt voor instabiliteit

Deep Q Network

- Gebruik neuraal netwerk om Q-functie te leren
- Observeer staat als pixels
- Q-learning met neuraal netwerk: combineert bootstrapping, off-policy training *en* functie benadering
- Action replay: train netwerk op oude ervaringen

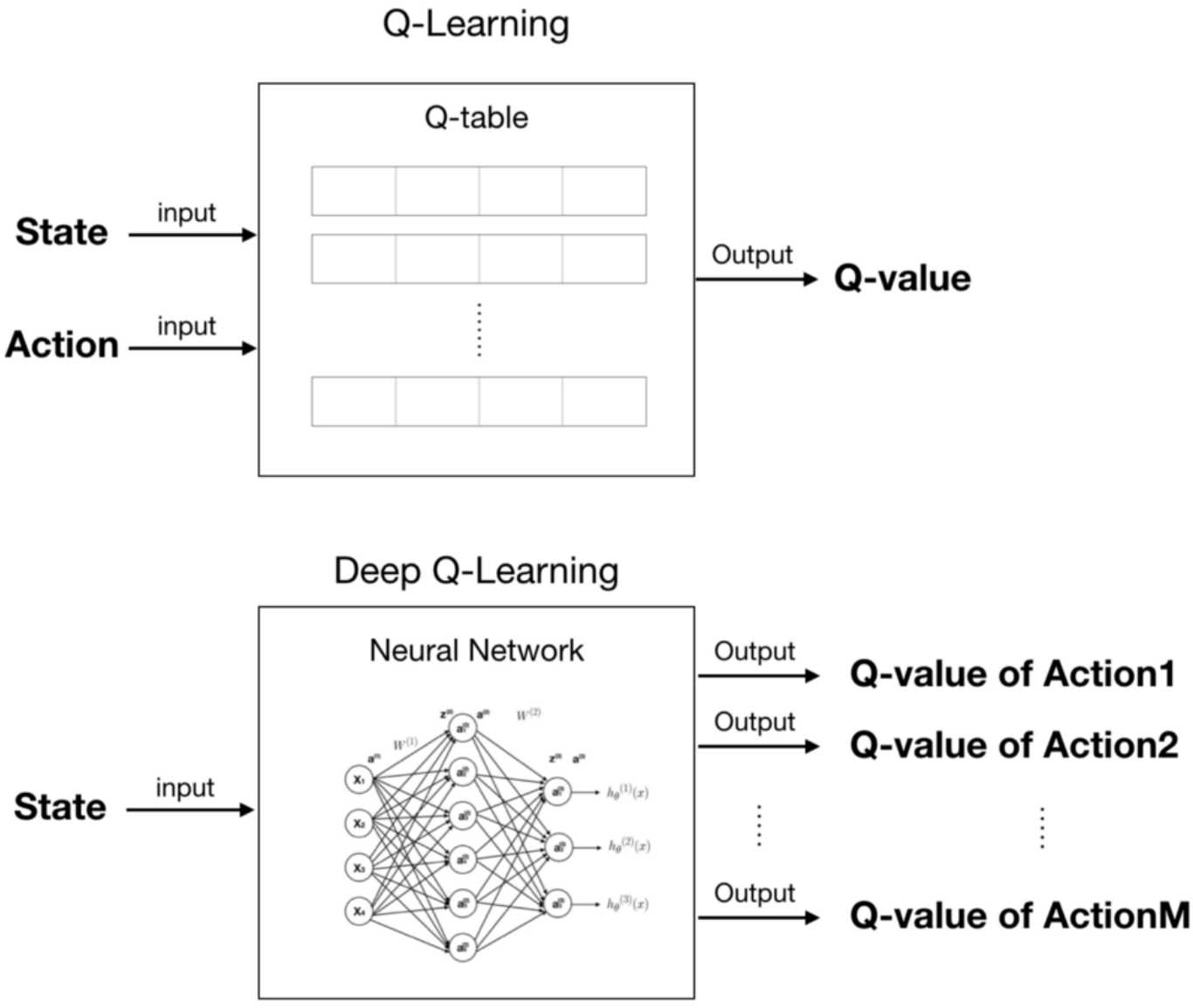


Figure 7: Deep Q Network

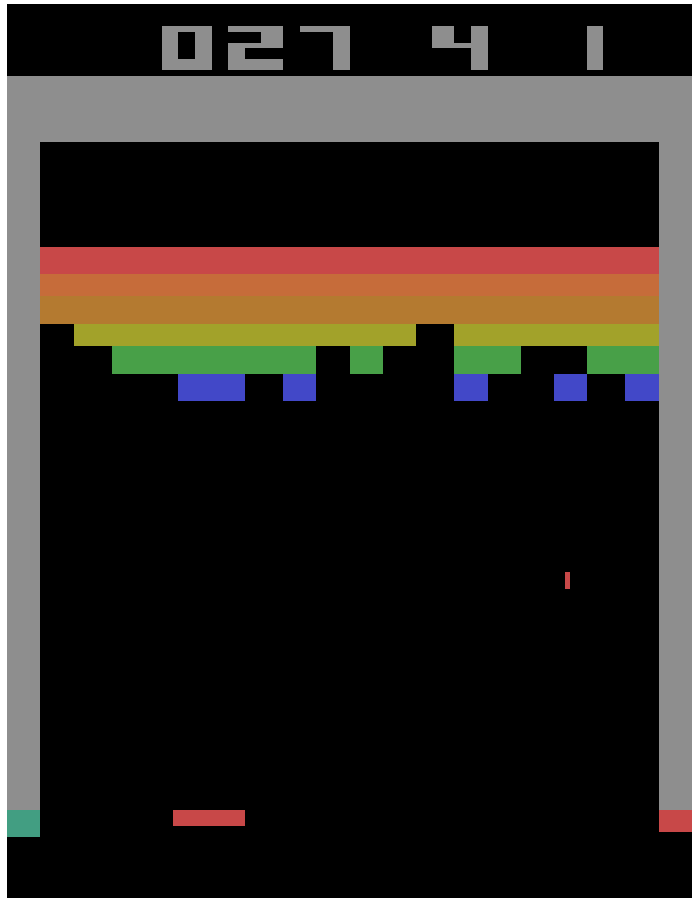


Figure 8: Breakout voor de Atari 2600

Random Search Methodes

Random search

- Alternatief voor gradient descent
  - Gradients zijn niet altijd beschikbaar
- Genereer N oplossingen $x_1, \cdots, x_N$, kies de beste x^*
  - Minimaliseert een functie $\mathcal{L}(x_n)$
- N moet heel groot zijn
- Niet praktisch voor grote zoekruimte

Simulated Annealing

- Random search gebruikt $p(X = x)$
- Vervang met $p(X_{\text{new}} = x | X_{\text{old}} = x^*)$
  - Gebruik kennis van vorige stap
  - "Goede" oplossingen liggen dichter bij elkaar
- Genereer "buurman" van vorige oplossing
  - Bijvoorbeeld $x = x^* + \epsilon, \epsilon \sim \mathcal{N}(0, 1)$
- Update wel of niet afhankelijk van temperatuur en score van x

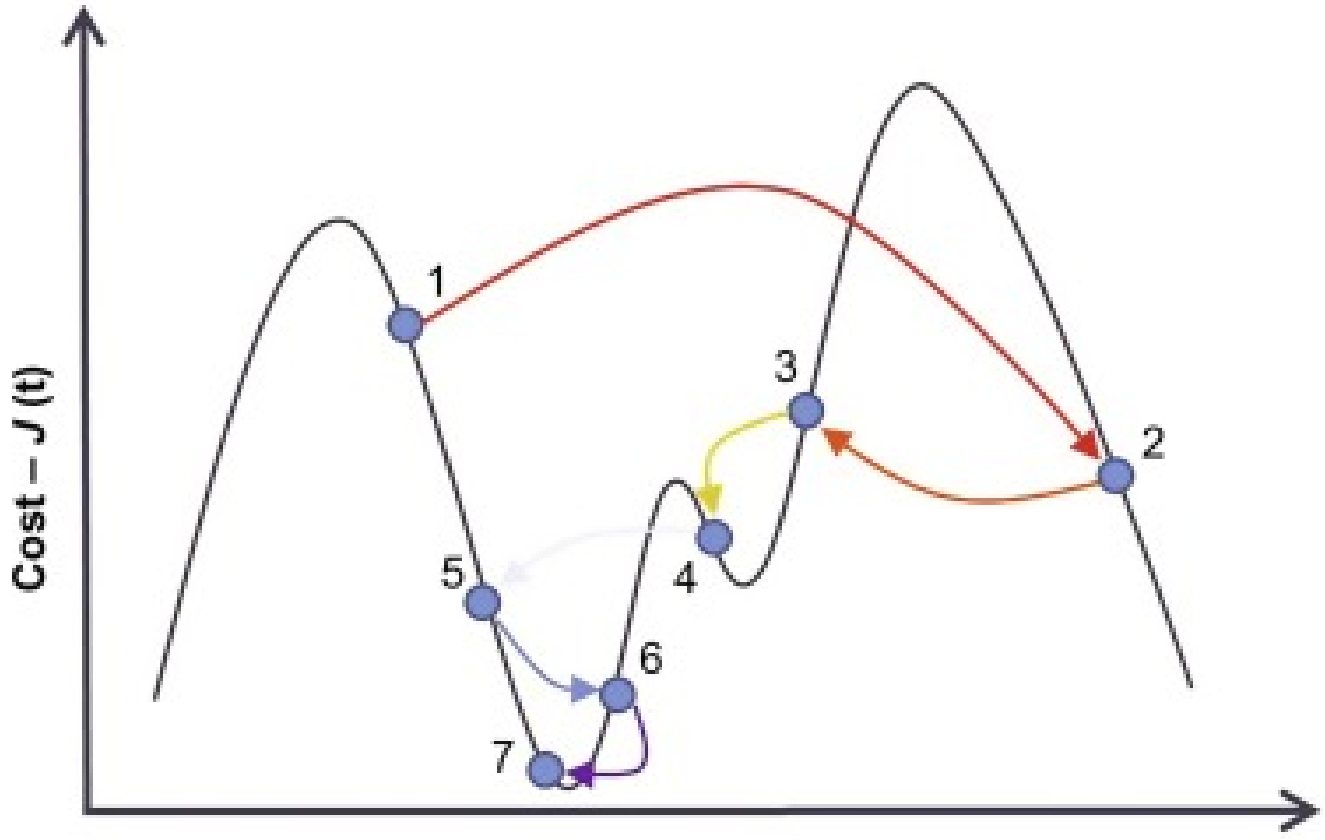
Simulated Annealing

```
Genereer x*

Voor elke n stap 1..N:
    t := 1 - n / N
    Genereer kandidaat x

    Genereer p = random.random()

    Als L(x) < L(x*) of p > exp(-(L(x) - L(x*)) / t):
        x* = x
```


Cost – J (t)
1
2
3
4
5
6
7

Evolutionaire algoritmes

- Meerdere kandidaten
- Selecteer K beste op basis van fitness
  - "Fitness" is functie die we willen optimaliseren
- Genereer nakomelingen op basis van de top K
  - Mutatie: stochastische verandering aan een individu
  - Crossover: stochastische combinatie van twee individuen
- Toe te passen op gewichten van neuraal netwerk
  - Maar ook op architectuur

Genetisch algoritme

```
Genereer kandidaten X = [x1, x2, ..., xN]

Voor iedere generatie:
    Evalueer fitness van alle individuen

    Selecteer K kandidaten met beste fitness

    Genereer N - K nieuwe kandidaten met mutatie en crossov

    Vervang N - K slechtste kandidaten met nieuwe nakomelin
```

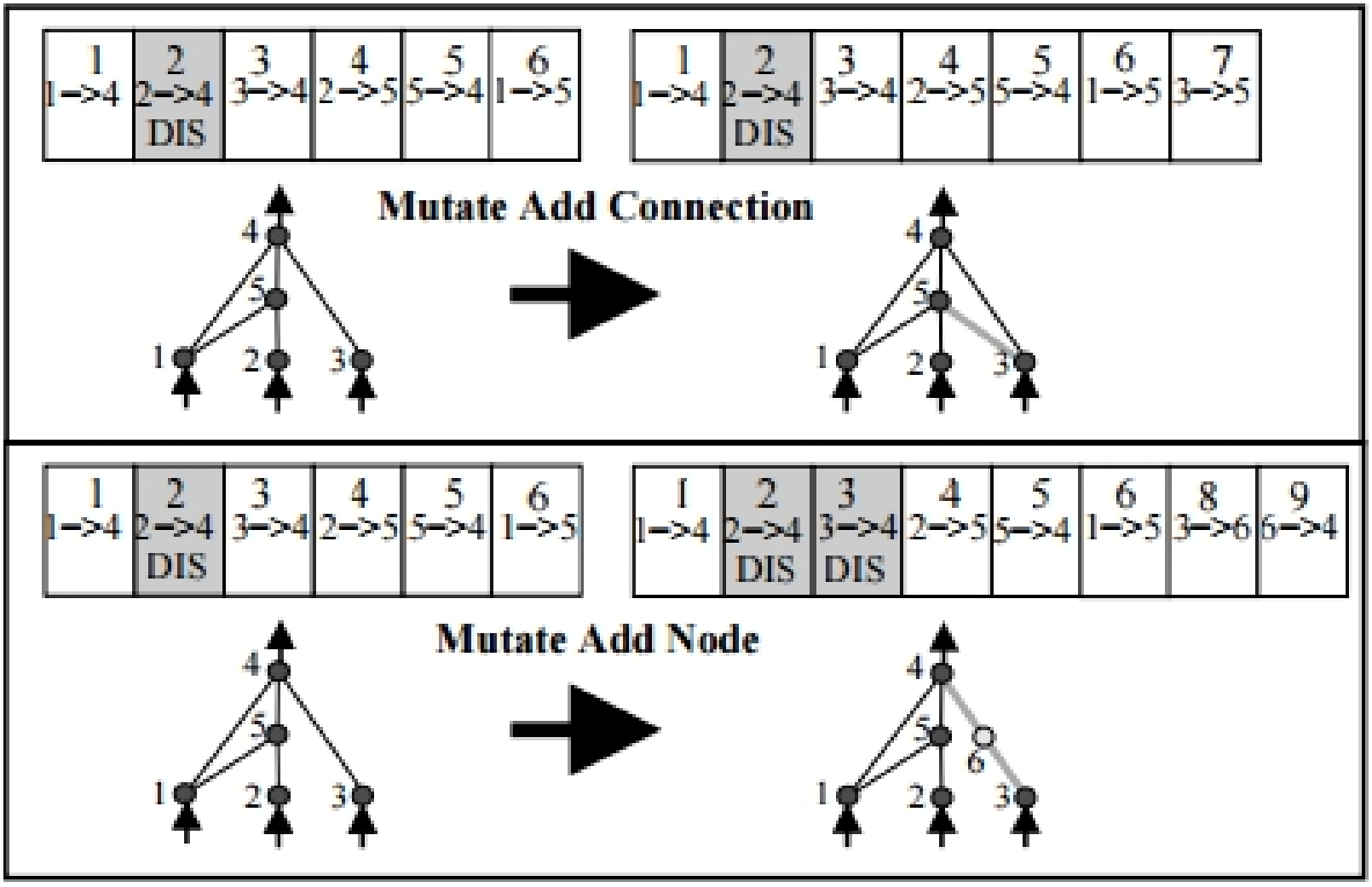


Figure 9: NEAT leert gewichten *en* architectuur

Evolutionaire strategie

- Genereer kandidaten volgens kansverdeling
- Selecteer beste kandidaten
- Update parameters van nieuwe kansverdeling met geselecteerde kandidaten
  - Monte Carlo schatting van parameters

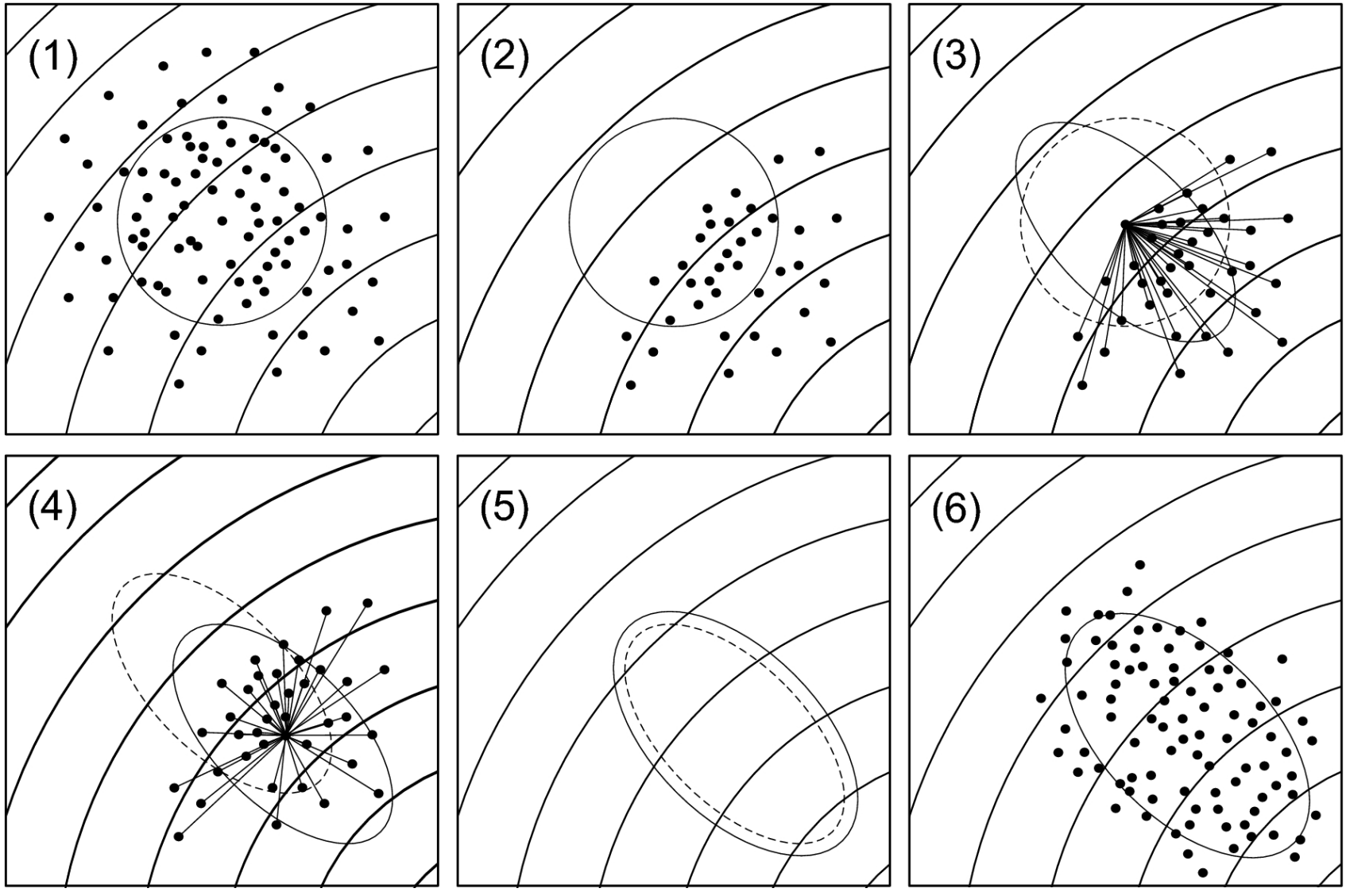


Figure 10: CMA-ES; een evolutionaire strategie

Augmented Random Search

- Leer direct een policy met random search
- Gebruikt functie benadering, zonder gradients
- Uitbreiding van Basic Random Search

Basic Random Search

```
Hyperparameters: lr, v, N
Initialiseer parameters: w = [0, 0, ..., 0]

Voor elke iteratie:
    Genereer N noise vectors met standard normaal steekproe

    for elke n in 1..N:
        G_n+ = beloning uit episode met beleid van: w + v *
        G_n- = beloning uit episode met beleid van: w - v *

    w += lr / N * sum((G_n+ - G_n-) * dn voor elke n in 1..
```

Aanpassingen

- Sorteer vectors `dn` op `max(G_n+, G_n-)`
  - Selecteer `b` beste vectoren voor update
  - Evolutie strategie
- Update stapgrootte voor BRS: `lr / N`
  - Vervang met `lr / (b * sd)`
  - `sd`: standaard deviatie van de `2 * b` beloningen
- Standaardiseer staten
  - Zorg dat staat componenten zelfde gemiddelde/standaard deviatie hebben
  - $(S - \mu)/\sigma$

Extra: Alpha Zero

Alpha Zero

- Verslaat wereldkampioenen in schaak en Go
- Leert een waarde functie en beleid
- $\mathcal{L}_t(s_t, \tilde{\pi}_t, z_t) = (v_\theta(s_t) - z_t)^2 - \tilde{\pi}_t \cdot \log(p_\theta(s_t))$
  - z_t: uitkomst van spel (-1 of 1)
  - $\tilde{\pi}_t$: schatting van optimaal beleid in t

Monte Carlo Tree Search

- Minimax algoritme met Monte Carlo Prediction
- Minimax
  - Doorzoek de “gametree” op zoek naar actie die maximale winst van tegenstander minimaliseert
  - Erg zwaar naarmate we dieper in de boom zoeken
- Monte Carlo Prediction
  - Schat de waarde van een staat door middel van simulaties

Monte Carlo Tree Search

- Node in boom voor elke actie
- Selectie: kies een punt in de boom waar we weinig van weten
- Uitbreiding: genereer nodes voor alle mogelijke acties
- Simulatie: simuleer een episode met een naïef beleid
- Update: houd het aantal gespeelde en gewonnen episodes bij voor alle nodes op het pad van de huidige tot de start node

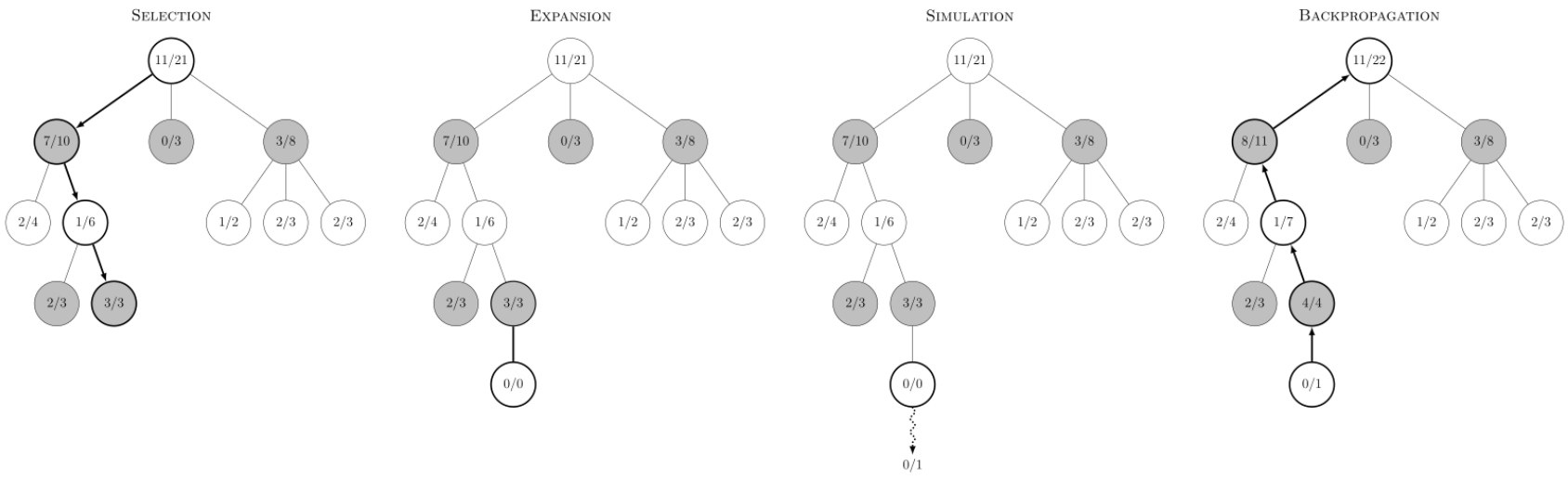


Figure 11: Een stap van MCTS