# Přírodou inspirované algoritmy

Jarek Kusák

May 2025

## 1 Základy strojového učení

### Vysvětlit rozdíly mezi jednotlivými typy strojového učení (učení s učitelem, bez učitele, zpětnovazební učení)

**Učení s učitelem (Supervised Learning)** K dispozici jsou vstupy a odpovídající výstupy. Cílem je naučit model, aby na základě vstupu predikoval správný výstup. Typickými úlohami jsou klasifikace a regrese.

**Učení bez učitele (Unsupervised Learning)** K dispozici jsou pouze vstupy. Model hledá v datech skrytou strukturu. Typickými úlohami jsou shlukování (clustering) nebo redukce dimenze.

**Zpětnovazební učení (Reinforcement Learning)** Agent interaguje s prostředím. Pozoruje stav, vybírá akci, dostává odměnu a snaží se nalézt strategii, která maximalizuje kumulovanou odměnu v čase.

### Vysvětlit rozdíly mezi jednotlivými úlohami strojového učení s učitelem (regrese, klasifikace)

**Regrese** Cílem je předpovědět spojitou (reálnou) hodnotu. Příklad:

$$\text{Cena nemovitosti} = f(\text{parametry nemovitosti})$$

**Klasifikace** Cílem je přiřadit vstup do jedné ze tříd. Příklad:

$$\text{Zvíře na obrázku} \in \{\text{pes}, \text{kočka}, \text{pták}\}$$

### Popsat typy příznaků (kategorické, ordinální, číselné) a jejich předzpracování

**Kategorické příznaky (Nominal)** Nemají přirozené pořadí (např. barva). *Předzpracování*: One-hot encoding.

**Ordinální příznaky (Ordinal)** Mají přirozené pořadí (např. velikost trička: S, M, L). *Předzpracování*: Label encoding nebo vhodné číselné hodnoty.

**Číselné příznaky (Numerical)** Reálná čísla (např. věk, váha). *Předzpracování*: Škálování (standardizace nebo min-max škálování).

# 2 Zpětnovazební učení

## Popsat problém zpětnovazebního učení intuitivně (agent, prostředí, hledání strategie)

Agent se pohybuje v prostředí. V každém čase:

- pozoruje stav $s_t$,
- vybírá akci $a_t$,
- dostane odměnu $r_t$ a prostředí přejde do nového stavu $s_{t+1}$.

Cílem agenta je naučit se strategii $\pi$, která mu umožní vybírat akce vedoucí k maximální kumulované odměně.

## Popsat prostředí jako Markovský rozhodovací proces

Prostředí je formálně modelováno jako **Markovský rozhodovací proces (MDP)** definovaný čtveřicí:

$$(S, A, P, R)$$

- $S$: množina stavů,
- $A$: množina akcí,
- $P_a(s, s')$: pravděpodobnost přechodu do stavu $s'$ po akci $a$ ve stavu $s$ (splňuje markovskou vlastnost - závisí pouze na aktuálním stavu s a akci a, nikoli na předchozích),
- $R_a(s, s')$: je funkce odměn - odměna za přechod.

## Definovat zpětnovazební učení jako problém hledání strategie maximalizující celkovou odměnu agenta

Cílem agenta je nalézt optimální politiku (strategii) $\pi^*$, která maximalizuje očekávanou diskontovanou odměnu:

$$\sum_{t=0}^{\infty} \gamma^t R_t$$

kde $\gamma$ je diskontní faktor (typicky $\gamma < 1$) a zajišťuje konečnost součtu.

## Definovat pojmy hodnota stavu a hodnota akce

**Hodnota stavu** $V^\pi(s)$: Očekávaná diskontovaná odměna ze stavu $s$ při následování strategie $\pi$:

$$V^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s\right]$$

**Hodnota akce** $Q^\pi(s,a)$: Očekávaná diskontovaná odměna ze stavu $s$ po provedení akce $a$ a dále následování strategie $\pi$:

$$Q^\pi(s,a) = E\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a\right]$$

## Popsat základní myšlenku Monte-Carlo metod

Monte Carlo a TD jsou metody vyhodnocující zvolenou strategii.

Monte-Carlo metody:

- Provádějí celé epizody až do konce.
- Po skončení epizody se spočítá **celková diskontovaná odměna** (tzv. návrat) $G$, např.:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$$

- Tato odměna se použije pro aktualizaci hodnoty:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[G - Q(s,a)\right]$$

- Odměna se zaznamenává a průměruje pro aktualizaci $Q(s,a)$.
- Fungují dobře u problémů s krátkými epizodami, ale špatně konvergují u problémů s vysokou variancí nebo dlouhými epizodami.

## Vysvětlit Q-učení (včetně vzorce pro aktualizaci matice Q)

Je to Temporal Differnece metoda (stejně jako SARSA níže) - rozdíl oproti Monte-Carlo metodám je ten, že se nečeká na skončení celé epizody, ale hned po obdržení odměny a uvidění nového stavu okamžitě aktualizujeme odhad hodnoty.

Q-učení je **off-policy** metoda, která přímo odhaduje optimální $Q^*$ pomocí:

$$Q_{\text{new}}(s_t,a_t) \leftarrow (1-\alpha)Q(s_t,a_t) + \alpha\left[r_t + \gamma \max_a Q(s_{t+1},a)\right]$$

kde $\alpha$ je rychlost učení a $\max_a Q$ reprezentuje odhad nejlepší možné budoucí odměny.

## Popsat algoritmus SARSA

SARSA je **on-policy** algoritmus, který aktualizuje $Q$ podle:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right]$$

Používá aktuální politiku k výběru akce $a_{t+1}$, takže reflektuje reálné chování agenta.

## Vysvětlit, jak použít Q-učení v prostorech se spojitými stavy a akcemi

Pro spojité prostory je klasická tabulka nevhodná. Používají se dvě řešení:

- **Diskretizace**: Rozdělení spojitého prostoru do intervalů (např. pozice auta v Mountain Car).
- **Funkční aproximace**: Použití modelu (např. neuronové sítě) pro aproximaci $Q(s, a)$.

To umožňuje škálovat Q-učení na problémy s velkými nebo spojitými prostory.

## (navíc) Tříkrokový příklad s podrobnými výpočty

**Scénář (všechny $Q$ na začátku $= 0$, výjimka $Q(S_1, B_2) = 4$)**

$$\gamma = 0.9, \qquad \alpha = 0.1$$

| t | stav $s_t$ | akce $a_t$ | odměna $r_t$ | $s_{t+1}$ |
|---|---|---|---|---|
| 0 | $\mathbf{S_0}$ | **A** | $+1$ | $S_1$ |
| 1 | $\mathbf{S_1}$ | **B** | $-1$ | $S_2$ |
| 2 | $\mathbf{S_2}$ | **C** | $+5$ | terminál |

### 1) Monte-Carlo (čekám, vidím pravdu)

**Návraty**

$$\begin{aligned} G_2 &= 5, \\ G_1 &= -1 + \gamma G_2 = -1 + 0.9 \cdot 5 = 3.5, \\ G_0 &= 1 + \gamma G_1 = 1 + 0.9 \cdot 3.5 = 4.15. \end{aligned}$$

**Aktualizace** $Q \leftarrow Q + \alpha\,(G - Q)$

$$\begin{aligned} Q(S_0, A) &\leftarrow 0 + 0.1\,(4.15 - 0) = 0.415, \\ Q(S_1, B) &\leftarrow 0 + 0.1\,(3.5 - 0) = 0.35, \\ Q(S_2, C) &\leftarrow 0 + 0.1\,(5 - 0) = 0.50. \end{aligned}$$

*„Čekám, vidím pravdu.“*

### 2) Q-learning (off-policy, „představuji si, že pak zahraju nejlepší možnou akci“)

**Krok 0** $(S_0 \to S_1)$

$$\begin{aligned} \text{cíl} &= r_0 + \gamma \max_{a'} Q(S_1, a') = 1 + 0.9 \cdot \mathbf{4} = 4.6, \\ \Delta &= 4.6 - 0 = 4.6, \\ Q(S_0, A) &\leftarrow 0 + 0.1 \cdot 4.6 = 0.46. \end{aligned}$$

**Krok 1** ($S_1 \to S_2$)

$$\begin{aligned} \text{cíl} &= -1 + 0.9 \cdot 0 = -1, \\ \Delta &= -1 - 0 = -1, \\ Q(S_1, B) &\leftarrow 0 + 0.1 \cdot (-1) = -0.10. \end{aligned}$$

**Krok 2** ($S_2 \to$ terminál)

$$\begin{aligned} \text{cíl} &= 5, \\ \Delta &= 5 - 0 = 5, \\ Q(S_2, C) &\leftarrow 0 + 0.1 \cdot 5 = 0.50. \end{aligned}$$

**3) SARSA (on-policy, „učím se z toho, co jsem reálně zahrál, včetně průzkumných kroků“)**

**Krok 0** ($S_0 \to S_1$, akce $B$ skutečně zvolená)

$$\begin{aligned} \text{cíl} &= 1 + 0.9 \cdot Q(S_1, B) = 1 + 0.9 \cdot 0 = 1, \\ \Delta &= 1 - 0 = 1, \\ Q(S_0, A) &\leftarrow 0 + 0.1 \cdot 1 = 0.10. \end{aligned}$$

**Krok 1** ($S_1 \to S_2$, akce $C$ zvolena)

$$\begin{aligned} \text{cíl} &= -1 + 0.9 \cdot Q(S_2, C) = -1 + 0 = -1, \\ \Delta &= -1 - 0 = -1, \\ Q(S_1, B) &\leftarrow 0 + 0.1 \cdot (-1) = -0.10. \end{aligned}$$

**Krok 2** (terminál)

$$Q(S_2, C) \leftarrow 0 + 0.1\,(5 - 0) = 0.50.$$

**Rekapitulace po jediné epizodě**

| $(s, a)$ | Monte-Carlo | Q-learning | SARSA |
|---|---|---|---|
| $(S_0, A)$ | 0.415 | **0.46** | 0.10 |
| $(S_1, B)$ | 0.35 | $-0.10$ | $-0.10$ |
| $(S_1, B_2)$ | 4 (beze změny) | 4 | 4 |
| $(S_2, C)$ | 0.50 | 0.50 | 0.50 |

**Snadno zapamatovatelné věty**

- **Monte-Carlo**: *„Čekám, vidím pravdu.“*
- **Q-learning**: *„Představuji si, že příště zahraju tu nejlepší možnost.“*
- **SARSA**: *„Učím se z toho, co jsem reálně zahrál, včetně průzkumných kroků.“*

# 3 Jednoduchý genetický algoritmus

## Popsat základní verzi evolučního algoritmu s binárním kódováním

Evoluční algoritmus pracuje s populací jedinců zakódovaných jako binární řetězce. Každý jedinec je hodnocen pomocí **fitness funkce**, která určuje kvalitu řešení.

Základní kroky:

- Inicializace populace náhodně.
- Vyhodnocení fitness.
- Výběr rodičů (např. ruletovou selekcí).
- Aplikace genetických operátorů (křížení a mutace).
- Vytvoření nové populace potomků.
- Opakování procesu.

## Popsat základní genetické operátory (jednobodové, n-bodové křížení, mutace)

**Jednobodové křížení** Náhodně se zvolí bod, kde se rozdělí rodiče. Potomek je složen z první části jednoho rodiče a druhé části druhého.

**n-bodové křížení** Zvolí se $n$ bodů. Potomek střídavě přebírá segmenty od obou rodičů.

**Mutace** Každý bit jedince má s malou pravděpodobností šanci se změnit (obrátit hodnotu). Mutace zvyšuje diverzitu populace.

## Popsat ruletovou a turnajovou selekci

**Ruletová selekce** Pravděpodobnost výběru je úměrná fitness:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$$

kde $f_i$ je fitness jedince $i$.

**Turnajová selekce** Náhodně se vybere několik jedinců (např. 2). S vysokou pravděpodobností (např. 80%) je vybrán ten nejlepší.

## Vysvětlit výhody a nevýhody turnajové a ruletové selekce

**Ruletová selekce**

- Výhody: reflektuje rozdíly ve fitness.

- Nevýhody: citlivá na škálování fitness, náchylná k dominanci nejsilnějších jedinců.

**Turnajová selekce**

- Výhody: robustní, funguje i pro záporné fitness.
- Nevýhody: při malých turnajích pomalejší selekční tlak.

## Vysvětlit, co je fitness funkce

**Fitness funkce** Funkce určující kvalitu řešení. Příklad:

- OneMAX: počet jedniček v binárním řetězci.
- Podmnožinový problém: fitness $= C - |\text{požadovaný součet} - \text{aktuální součet}|$ kde $C$ je konstanta.

## Popsat elitismus

**Elitismus** Zaručuje přežití nejlepších jedinců. Nejlepší jedinci jsou zkopírováni přímo do nové populace. Tím se zabrání ztrátě nejlepšího řešení kvůli genetickým operátorům.

## Popsat metody pro environmentální selekci (přenos jedinců do další populace), $(\mu, \lambda)$ a $(\mu + \lambda)$

**$(\mu, \lambda)$ selekce**

- $\mu$ rodičů vytvoří $\lambda$ potomků.
- Do další generace přejde $\mu$ nejlepších potomků.
- $\lambda > \mu$ (silný selekční tlak).

**$(\mu + \lambda)$ selekce**

- $\mu$ rodičů + $\lambda$ potomků tvoří kandidáty.
- Do další generace přejde $\mu$ nejlepších z rodičů a potomků.
- Udržuje diverzitu a chrání nejlepší řešení.

## Popsat rozšíření jednoduchého GA na problémy s celočíselným kódováním

**Celočíselné kódování**

- Jedinci nejsou binární, ale posloupnosti celých čísel.
- Křížení: obdobně jako u binárních, např. jednobodové nebo uniformní.
- Mutace: nahrazení genu náhodnou hodnotou nebo přičtení malého čísla.

Výběr genetických operátorů závisí na interpretaci celočíselných genů.

# 4 Evoluční algoritmy pro spojitou a kombinatorickou optimalizaci

## Definovat problém spojité optimalizace

Spojitá optimalizace znamená hledání extrému (minima nebo maxima) funkce:

$$f : R^n \to R$$

Typicky se jedná o problémy, kde proměnné nabývají reálných hodnot a cílem je nalézt hodnoty vektoru $x \in R^n$, které optimalizují funkci $f(x)$.

## Popsat genetické operátory používané ve spojité optimalizaci - aritmetické křížení, zatížené a nezatížené mutace

**Aritmetické křížení** Potomek je získán jako vážený průměr rodičů:

$$x^{\text{potomek}} = \alpha x^{(1)} + (1 - \alpha)x^{(2)}, \quad \alpha \in [0, 1]$$

**Zatížená (biased) mutace** Např. Gaussovská mutace:

$$x_i \leftarrow x_i + \mathcal{N}(0, \sigma^2)$$

Mutace respektuje aktuální hodnotu a mění ji pouze mírně.

**Nezatížená (unbiased) mutace** Hodnota $x_i$ je nahrazena náhodnou hodnotou z povoleného rozsahu:

$$x_i \leftarrow \text{Uniform}(a_i, b_i)$$

## Vysvětlit rozdíl mezi separabilními a neseparabilními funkcemi

**Separabilní funkce** Funkce $f(x_1, \ldots, x_n)$ je separabilní, pokud lze optimalizovat každou proměnnou nezávisle, tj. má tvar:

$$f(x_1, \ldots, x_n) = f_1(x_1) + f_2(x_2) + \cdots + f_n(x_n)$$

**Neseparabilní funkce** Mezi proměnnými existují závislosti. Nelze je optimalizovat odděleně. Příkladem je funkce s rotovanými elipsovitými vrstevnicemi (vysoká podmíněnost).

## Popsat algoritmus diferenciální evoluce a jeho výhody

**Diferenciální evoluce (DE)** je algoritmus navržený pro efektivní optimalizaci v $R^n$ i u neseparabilních funkcí.

- Pro každý jedinec $x$ vybereme náhodně 3 další: $a$, $b$, $c$.

- Vytvoří se mutace:
$$v = a + F \cdot (b - c)$$
- Provádí se uniformní křížení mezi $x$ a $v$.
- Do nové populace vstupuje ten s lepší fitness.

**Výhody:**

- Invariance vůči rotacím a škálování.
- Není potřeba nastavovat konkrétní mutační rozptyl.
- Funguje i u vysoce podmíněných a neseparabilních problémů.

## Popsat mutace pro permutační kódování

**Permutační mutace** zachovávají permutační charakter jedince:

- **Swap**: prohození dvou náhodně vybraných pozic.
- **Insert**: vyjmutí prvku a vložení na jinou pozici.
- **Reverse**: otočení podposloupnosti.

## Popsat křížení používaná při permutačním kódování (OX, PMX, ER)

**Order Crossover (OX)**

- Vybere se segment z jednoho rodiče.
- Zbytek potomka je doplněn podle pořadí genů z druhého rodiče (bez duplicit).
- Používá se pro zachování relativních pozic.

**Partially Mapped Crossover (PMX)**

- Vymění se segmenty rodičů.
- Vzniklé konflikty se opravují pomocí mapování hodnot.

**Edge Recombination (ER)**

- Vytvoří se seznam sousedů každého vrcholu podle obou rodičů.
- Postupně se vybírají vrcholy s nejmenším počtem sousedů.
- Výsledkem je potomek s co nejvíce zachovanými hranami.

### Vysvětlit použití evolučních algoritmů pro problém obchodního cestujícího

**Problém obchodního cestujícího (TSP)** Cílem je najít nejkratší cestu v úplném grafu, která navštíví všechny města právě jednou a vrátí se do výchozího bodu.

**Kódování:** Permutace čísel reprezentující pořadí měst.
**Fitness:** Negativní délka trasy (protože se maximalizuje).
**Operátory:**

- Mutace: prohození měst, otočení části trasy.
- Křížení: OX, PMX, ER – zachovávají platnost permutace.

**Výhoda evolučního přístupu:** Schopnost efektivně prohledávat složité, kombinatorické prostory bez nutnosti derivací nebo konvexity.

## 5 Genetické programování

### Popsat lineární genetické programování (kódování jedince, operátory)

**Kódování:** Jedinec je posloupnost instrukcí, podobně jako assembler (Slash/A). Každá instrukce manipuluje s registry nebo pamětí.
**Operátory:**

- **Mutace:** nahrazení instrukce jinou.
- **Křížení:** výměna bloků instrukcí mezi dvěma programy.

### Popsat kartézské genetické programování (kódování jedince, operátory)

**Kódování:** Program je reprezentován jako orientovaný acyklický graf na mřížce $r \times l$ (řádky $\times$ sloupce). Každý gen kóduje:

- použitou funkci (např. +, *, sin),
- indexy vstupů (z předchozích sloupců),
- index výstupu programu.

**Operátory:**

- **Mutace:** změna funkce, vstupních indexů nebo výstupního uzlu.
- Křížení se běžně nepoužívá.

## Popsat gramatickou evoluci (kódování jedince, operátory)

**Kódování:** Jedinec je posloupnost čísel. Každé číslo určuje pomocí modulární aritmetiky (modulo počtem pravidel), jaké přepisovací pravidlo z bezkontextové gramatiky se použije pro daný neterminál.

**Operátory:**

- **Mutace:** změna jednotlivých čísel.
- **Křížení:** výměna podčástí reprezentace souvisejících s konkrétním neterminálem (zachování sémantiky).

## Vysvětlit, jak řešit problém s příliš krátkým nebo dlouhým jedincem v gramatické evoluci

- **Příliš krátký jedinec:** může skončit dříve, než je výraz dokončen — zůstanou neterminály. Takový jedinec není platný a nelze vyhodnotit.
- **Řešení:** generovat delší jedince. Pokud část genů není využita, nevadí.
- **Dlouhý jedinec:** jednoduše se ignorují geny, které nebyly při přepisu použity.

## Popsat stromové genetické programování a jeho genetické operátory

**Kódování:** Program je reprezentován jako strom. Uzly: operace (funkce), listy: vstupy (neterminály) nebo konstanty (terminály).

**Operátory:**

- **Křížení:** výměna podstromů.
- **Mutace:** nahrazení uzlu nebo podstromu jiným.

## Popsat práci s konstantami ve stromovém genetickém programování

- Používají se omezené množiny konstant jako terminály (např. 0, 1, 2).
- Další konstanty mohou být generovány pomocí aritmetiky.
- Alternativně: použít speciální uzly (např. `ConstNode(x)`) a evolučně měnit $x$ (např. pomocí Gaussovské mutace).

## Popsat typované genetické programování

**Typované GP:** každá funkce i terminál má specifikovaný typ vstupu a výstupu (např. `int`, `bool`).

- Zajišťuje sémantickou korektnost programů.
- Při křížení a mutacích je zachována typová konzistence.

## Popsat automaticky definované funkce

**ADF (Automatically Defined Functions):**

- Evoluce umožňuje vytvářet a používat pomocné funkce (podprogramy).
- Každá ADF může mít vlastní tělo a vstupy.
- Pomáhá se znovupoužitím kódu a škálovatelností.

## Vysvětlit, jak u automaticky definovaných funkcí zabránit zacyklení/nekonečné rekurzi

- Zakázat přímou i nepřímou rekurzi během generování nebo při mutacích.
- Omezit hloubku volání funkcí.
- Použít statickou analýzu (detekce cyklů v grafu volání).

## Porovnat jednotlivé typy genetického programování a jejich výhody/nevýhody

- **Lineární GP:**
  - \+ jednoduchá reprezentace, efektivní provádění
  - méně intuitivní pro výrazové úlohy
- **Kartézské GP:**
  - \+ vhodné pro evoluci obvodů, paralelních struktur
  - méně přehledné, omezenější křížení
- **Gramatická evoluce:**
  - \+ univerzálnost, možnost generovat libovolné jazyky
  - problém s krátkými jedinci, závislost na gramatice
- **Stromové GP:**
  - \+ velmi přirozené pro evoluci výrazů a programů
  - rostoucí složitost, bloat (zvětšování stromů)

# 6 Particle Swarm Optimization

## Popsat aktualizaci poloh a rychlostí v PSO

Každá částice $i$ má pozici $x_i$ a rychlost $v_i$. V každém kroku se aktualizuje:

$$v_i \leftarrow \omega v_i + \varphi_p r_p(p_i - x_i) + \varphi_g r_g(g - x_i)$$

$$x_i \leftarrow x_i + v_i$$

kde:

- $p_i$: nejlepší pozice částice $i$ (personal best),
- $g$: globální nejlepší pozice (nebo nejlepší v sousedství),
- $\omega$: setrvačnost (inertia),
- $\varphi_p$, $\varphi_g$: koeficienty přitažlivosti k $p_i$ a $g$,
- $r_p$, $r_g$: náhodná čísla z $[0, 1]$.

## Popsat topologie používané v PSO (globální, geometrické, sociální)

- **Globální topologie:** každá částice zná globální nejlepší řešení.
- **Geometrická topologie:** komunikace s částicemi, které jsou fyzicky blízko (v prostoru řešení).
- **Sociální topologie:** pevně daná síť sousedů (např. kruh, mřížka).

## Vysvětlit vliv topologií na algoritmus PSO

- **Globální:** rychlá konvergence, ale riziko uvíznutí v lokálním extrému.
- **Lokální (geometrická/sociální):** pomalejší konvergence, ale lepší průzkum prostoru (explorace), vyšší robustnost.

## Vysvětlit použití PSO pro řešení problémů ve spojité optimalizaci

PSO je přirozeně určené pro spojitou optimalizaci, protože pozice částic jsou reálné vektory $x_i \in R^n$. Fitness je hodnota optimalizované funkce v pozici částice.

Pro jiné typy problémů (např. diskrétní nebo kombinatorické) je potřeba modifikovat výpočet pozic a rychlostí (např. zaokrouhlování, množinové operace).

# 7 Ant Colony Optimization

## Vysvětlit základní pojmy ACO - feromon, atraktivita

- **Feromon $\tau_{xy}$:** množství stopy na hraně $(x, y)$.
- **Atraktivita $\nu_{xy}$:** heuristická hodnota (např. 1/vzdálenost).
- Pravděpodobnost výběru hrany:

$$P_{xy} \propto (\tau_{xy})^{\alpha} (\nu_{xy})^{\beta}$$

  kde $\alpha$, $\beta$ ovlivňují vliv feromonu a atraktivity.

## Popsat generování řešení pomocí ACO

Každý mravenec začíná v náhodném uzlu a iterativně vybírá další uzel podle pravděpodobnosti $P_{xy}$. Tím vygeneruje cestu (např. permutaci měst pro TSP).

## Popsat aktualizaci feromonu na základě kvality nalezeného řešení

Aktualizace feromonu v ACO probíhá ve dvou krocích:

1. **Vypařování feromonu:** Na všech hranách v grafu se sníží množství feromonu podle vzorce:

$$\tau_{xy} \leftarrow (1 - \rho) \cdot \tau_{xy}$$

   kde $\tau_{xy}$ je aktuální množství feromonu na hraně $(x, y)$ a $\rho \in (0, 1)$ je konstanta určující míru vypařování. Tento krok zabraňuje neomezené kumulaci feromonu a podporuje průzkum nových řešení.

2. **Vklad nového feromonu:** Každý mravenec, který nalezl řešení, položí feromon na hrany, které použil. Feromon je úměrný kvalitě řešení:

$$\tau_{xy} \leftarrow \tau_{xy} + \sum_{k} \Delta\tau_{xy}^{k}$$

   kde příspěvek mravence $k$ na hranu $(x, y)$ je:

$$\Delta\tau_{xy}^{k} = \begin{cases} \dfrac{Q}{L_k}, & \text{pokud mravenec } k \text{ použil hranu } (x, y), \\ 0, & \text{jinak.} \end{cases}$$

   V tomto vzorci je $Q$ konstanta (např. nastavena podle cílové úlohy) a $L_k$ je délka cesty nalezené mravencem $k$.

## Popsat použití ACO pro řešení problému obchodního cestujícího a vehicle routing problému

**Problém obchodního cestujícího (TSP):**

- Uzly: $x, y \in V$ – města.
- Hrany: $(x, y)$ – cesty mezi městy.
- Cíl: najít uzavřenou permutaci všech uzlů s minimální celkovou délkou.
- Feromon $\tau_{xy}$: síla preference mravenců pro použití hrany $(x, y)$.
- Atraktivita $\nu_{xy} = \dfrac{1}{d_{xy}}$, kde $d_{xy}$ je délka hrany $(x, y)$ (např. eukleidovská vzdálenost).
- Pravděpodobnost přechodu z $x$ do $y$:

$$P_{xy} \propto (\tau_{xy})^{\alpha} (\nu_{xy})^{\beta}$$

- Kvalita řešení $L_k$: celková délka cesty nalezené mravencem $k$:

$$L_k = \sum_{(x,y) \in \text{cesta}_k} d_{xy}$$

**Vehicle Routing Problem (VRP):**

- Uzly: $x, y \in V$ – zákazníci a sklad (depot).
- Hrany: $(x, y)$ – cesty mezi zákazníky nebo mezi skladem a zákazníky.
- Cíl: rozvést zboží všem zákazníkům s minimálním počtem vozidel a minimální celkovou délkou tras.
- Omezení:
  - kapacita vozidla $C$,
  - maximální délka trasy $D$ (volitelné).
- Feromon $\tau_{xy}$: preference pro použití hrany $(x, y)$.
- Atraktivita $\nu_{xy} = \dfrac{1}{d_{xy}}$, stejně jako u TSP.
- Mravenec generuje trasu podle:

$$P_{xy} \propto (\tau_{xy})^{\alpha} (\nu_{xy})^{\beta}$$

  dokud:
  - nedojde kapacita: $\sum_{\text{zákazníci } i} \text{demand}_i \leq C$,

– nepřekročí délku trasy: $\sum d_{xy} \leq D$.

Poté se vrací zpět do skladu a zahajuje novou trasu.

- Kvalita řešení $L_k$: celková délka všech tras mravence $k$:

$$L_k = \sum_{\text{všechny trasy}} \sum_{(x,y)\in\text{trasa}} d_{xy}$$

# 8 Perceptronové neuronové sítě

## Popsat, jakým způsobem počítá jeden perceptron

Perceptron je základní neuron, který počítá výstup na základě váženého součtu vstupů a prahu (biasu). Aktivace se spočítá jako:

$$\xi = \sum_{i=1}^{n} w_i x_i + b$$

a výstup je:

$$f(\xi) = \begin{cases} 1, & \xi \geq 0 \\ 0, & \text{jinak} \end{cases}$$

Bias $b$ lze absorbovat do váhového vektoru přidáním vstupu $x_0 = 1$, čímž dostaneme:

$$f\left(\sum_{i=0}^{n} w_i x_i\right)$$

## Popsat algoritmus pro trénování perceptronu a vysvětlit, kdy konverguje

Perceptron se učí pomocí aktualizace vah podle chyby:

$$w_i \leftarrow w_i + \eta(y - f(x))x_i$$

kde $y$ je požadovaný výstup, $f(x)$ výstup perceptronu, $\eta$ je rychlost učení.

**Konvergence:** Algoritmus konverguje (najde řešení), pokud jsou trénovací data lineárně separabilní.

## Popsat strukturu vícevrstvých perceptronových sítí

Vícevrstvá perceptronová síť (MLP) je složena z:

- vstupní vrstvy,
- jedné nebo více skrytých vrstev,
- výstupní vrstvy.

Každý neuron ve vrstvě přijímá vstupy z předchozí vrstvy a výstup posílá do vrstvy následující. Výpočty probíhají dopředně bez zpětných vazeb.

## Popsat aktivační funkce používané ve vícevrstvých perceptronech (sigmoida, tanh, ReLU)

- **Sigmoida (logistická):**

$$f(x) = \frac{1}{1 + e^{-\lambda x}}, \quad f'(x) = \lambda f(x)(1 - f(x))$$

- **Hyperbolický tangens (tanh):**

$$f(x) = \tanh(x), \quad f'(x) = 1 - f^2(x)$$

- **ReLU (Rectified Linear Unit):**

$$f(x) = \max(0, x), \quad f'(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

## Popsat metodu pro trénování neuronových sítí (backpropagation)

**Backpropagation** je metoda pro výpočet gradientu chybové funkce vůči váhám v síti. Skládá se ze dvou kroků:

1. **Dopředný průchod:** spočítá se výstup celé sítě.
2. **Zpětný průchod:** spočítají se derivace chyby a upraví se váhy:

$$w_{ij} \leftarrow w_{ij} - \alpha \frac{\partial L}{\partial w_{ij}}$$

   kde $L$ je chybová funkce a $\alpha$ je rychlost učení.

## Odvodit pravidla pro adaptaci vah v metodě zpětného šíření

Uvažujeme výstupní neuron s aktivací:

$$\xi_k = \sum_j w_{jk} x_j, \quad y_k = f(\xi_k)$$

Chyba:

$$E = \frac{1}{2} \sum_k (y_k - t_k)^2$$

Derivace podle váhy:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial y_k} \cdot \frac{\partial y_k}{\partial \xi_k} \cdot \frac{\partial \xi_k}{\partial w_{jk}} = (y_k - t_k) f'(\xi_k) x_j$$

Definujeme:

$$\delta_k = (y_k - t_k) f'(\xi_k)$$

a dostaneme adaptační pravidlo:

$$w_{jk} \leftarrow w_{jk} - \alpha \delta_k x_j$$

Pro skrytou vrstvu analogicky:

$$\delta_j = f'(\xi_j) \sum_k \delta_k w_{jk}$$

a aktualizace vah:

$$w_{ij} \leftarrow w_{ij} - \alpha \delta_j x_i$$

**Poznámka:** V případě více vstupních vzorů se jednotlivé příspěvky sčítají (nebo se použije minibatch).

# 9 RBF sítě

## Popsat RBF jednotku a architekturu RBF sítě

RBF (Radial Basis Function) jednotka počítá svou aktivaci na základě vzdálenosti vstupu $x$ od středu $c_i$:

$$y_i = \rho(\|x - c_i\|)$$

kde $\rho$ je radiální aktivační funkce – typicky Gaussova:

$$\rho(x) = \exp\left(-\beta \|x - c_i\|^2\right)$$

Architektura RBF sítě:

- **Skrytá vrstva:** složená z RBF jednotek.
- **Výstupní vrstva:** lineární kombinace výstupů skrytých jednotek.

Výstup celé sítě:

$$\varphi(x) = \sum_{i=1}^{N} w_i \rho(\|x - c_i\|)$$

## Vysvětlit princip trénování RBF sítí

Trénink probíhá ve třech krocích:

1. **Určení středů** $c_i$ pomocí shlukovacího algoritmu (např. $k$-means).
2. **Nastavení parametru** $\beta_i$ pomocí vzorce:

   $$\beta_i = \frac{1}{2\sigma_i^2}$$

   kde $\sigma_i$ je průměrná vzdálenost bodů ve shluku od středu $c_i$.
3. **Trénink výstupní vrstvy** – lineární regrese (např. metoda nejmenších čtverců).

### Popsat algoritmus $k$-means

Algoritmus $k$-means slouží ke shlukování dat:

1. Náhodně vybereme $k$ počátečních středů $m_1, \ldots, m_k$.
2. Opakujeme:
   - Každý vstupní bod přiřadíme nejbližšímu středu (dle Euklidovské vzdálenosti).
   - Nové středy vypočítáme jako průměr bodů přiřazených do daného shluku.
3. Ukončíme po konvergenci (středy se nemění) nebo po pevném počtu iterací.

### Porovnat geometrické vlastnosti RBF sítí a vícevrstvých perceptronů

- **RBF sítě:** vytvářejí rozhodovací oblasti pomocí součtu lokálních funkcí (např. Gaussovek), aktivují se jen pro vstupy blízké svému středu.
- **Vícevrstvé perceptrony (MLP):** tvoří rozhodovací hranice pomocí nelineárního složení lineárních operací (např. hyperplochy).
- **Shrnutí:**
  - RBF sítě jsou vhodné pro interpolaci a lokální učení.
  - MLP lépe aproximují globální nelineární funkce.

## 10 Rekurentní neuronové sítě

### Definovat, co jsou rekurentní neuronové sítě

Rekurentní neuronová síť (RNN) je taková, ve které mohou výstupy neuronů ovlivňovat výpočty v budoucích časových krocích. Obsahuje zpětné vazby a stav, který se přenáší mezi časovými kroky. Umožňuje pracovat s posloupnostmi vstupů libovolné délky.

### Vysvětlit, kde se rekurentní sítě používají

RNN se používají především tam, kde je důležité zachytit závislosti v posloupnostech:

- předpovídání časových řad,
- strojový překlad,
- rozpoznávání řeči,
- generování textu.

## Vysvětlit trénování rekurentních sítí pomocí algoritmu zpětného šíření v čase

Trénování se provádí pomocí algoritmu **Backpropagation Through Time (BPTT)**:

1. Síť se „rozvine v čase“, tj. vytvoří se její kopie pro každý časový krok.
2. Na rozvinutou síť se aplikuje běžný algoritmus backpropagation.
3. Gradienty se zpětně propagují přes všechny časové kroky.

## Vysvětlit problém s explodujícími a mizejícími gradienty

- Pokud jsou rekurentní váhy $w$ menší než 1, gradient se s každým krokem násobí malým číslem a **mizí**.
- Pokud jsou větší než 1, gradient **roste exponenciálně**.
- Oba případy vedou k tomu, že učení se buď zastaví, nebo je nestabilní.

## Popsat princip Echo State sítí

Echo State Network (ESN) má:

- velkou nelineární rekurentní vrstvu s náhodnými vahami,
- tyto váhy se během trénování nemění,
- pouze výstupní vrstva se trénuje (např. lineární regrese).

Aktualizace vnitřního stavu:

$$s_t = f(W[s_{t-1}, x_t])$$

kde $s_t$ je vnitřní stav, $x_t$ vstup a $f$ nelineární aktivační funkce (např. tanh).

## Popsat trénování Echo State sítí

Trénuje se pouze výstupní vrstva:

$$y_t = W_{\text{out}} \cdot s_t$$

- Sbírají se stavy $s_t$ během běhu sítě.
- Poté se trénuje $W_{\text{out}}$ pomocí lineární regrese (např. pomocí Moore-Penrose pseudoinverze).

### Popsat LSTM buňku a architekturu LSTM sítě

Každá LSTM buňka má vstupní, zapomínací a výstupní bránu. Vstupy jsou předchozí výstup $h_{t-1}$ a aktuální vstup $x_t$.

$$
\begin{aligned}
f_t &= \sigma(W_f[h_{t-1}, x_t] + b_f) \quad \text{(zapomínací brána)} \\
i_t &= \sigma(W_i[h_{t-1}, x_t] + b_i) \quad \text{(vstupní brána)} \\
\tilde{C}_t &= \tanh(W_c[h_{t-1}, x_t] + b_c) \quad \text{(kandidát stavu)} \\
C_t &= f_t * C_{t-1} + i_t * \tilde{C}_t \quad \text{(nový stav)} \\
o_t &= \sigma(W_o[h_{t-1}, x_t] + b_o) \quad \text{(výstupní brána)} \\
h_t &= o_t * \tanh(C_t) \quad \text{(výstup buňky)}
\end{aligned}
$$

### Vysvětlit výhody Echo State sítí a LSTM sítí v porovnání se základními RNN

- **ESN:** jednoduchý trénink, rychlé učení, stačí lineární regrese, dobře funguje pro lineárně separabilní úlohy.
- **LSTM:** odolné vůči mizejícím gradientům, dokáže uchovávat dlouhodobé závislosti, vhodné pro složité posloupnosti.

## 11 Konvoluční sítě

### Popsat funkci konvolučního filtru

Konvoluční filtr aplikuje na malé části obrázku (např. $3 \times 3$) stejnou lineární operaci – vážený součet hodnot pixelů. Tento filtr se posouvá po celém obrázku (s daným krokem – *stride*) a výsledkem je tzv. feature map.

Výstup jednoho filtru:

$$y_{i,j} = \sum_{u,v} w_{u,v} \cdot x_{i+u,j+v}$$

kde $w_{u,v}$ jsou váhy filtru a $x_{i+u,j+v}$ jsou pixely vstupního obrázku.

### Popsat funkci pooling vrstev v konvoluční síti

Pooling vrstvy slouží ke zmenšení rozměrů a redukci výpočetní složitosti. Nejčastěji používaný je max-pooling, který z každého podokna (např. $2 \times 2$) vybere maximální hodnotu:

$$y_{i,j} = \max_{(u,v) \in \text{okno}} x_{i+u,j+v}$$

Pooling zajišťuje translatační invarianci a zmenšuje citlivost na malé posuny ve vstupu.

## Popsat architekturu konvoluční neuronové sítě

Typická konvoluční síť (CNN) se skládá z:

- **Vstupní vrstvy** – obrázek (např. $100 \times 100 \times 3$).
- **Střídání konvolučních a pooling vrstev** – postupně extrahují stále abstraktnější reprezentace obrazu.
- **Plně propojené vrstvy** – na konci sítě, pracují s výslednou zkomprimovanou reprezentací.
- **Výstupní vrstva** – např. softmax pro klasifikaci.

## Vysvětlit, co jsou matoucí vzory a jak ovlivňují praktické nasazení neuronových sítí

Matoucí vzory (adversarial examples) jsou vstupy, které jsou na pohled téměř totožné s běžnými vstupy, ale způsobí, že neuronová síť udělá chybnou predikci.

I velmi malé změny ve vstupu mohou způsobit zásadní změnu ve výstupu modelu, což je bezpečnostní problém – zejména např. v autonomním řízení nebo rozpoznávání obličejů.

## Popsat algoritmus FGSM pro vytváření matoucích vzorů

**FGSM (Fast Gradient Sign Method)**:

- Spočítá se gradient ztrátové funkce $J(\theta, x, y)$ podle vstupu $x$.
- Vstup se upraví směrem, který nejvíc zvětšuje chybu:

$$x' = x + \epsilon \cdot \text{sign}(\nabla_x J(\theta, x, y))$$

- Malé $\epsilon$ (např. $< 0.1$) stačí k výrazné změně výstupu sítě.

## Vysvětlit myšlenku přenosu uměleckého stylu

Cílem je převést obsah jednoho obrázku do stylu jiného (např. fotografie do stylu Van Gogha).

- **Obsah** – reprezentován aktivacemi ve vyšších vrstvách sítě.
- **Styl** – reprezentován korelacemi aktivací (Gramovy matice).
- Optimalizační problém: najít nový obrázek, jehož aktivace odpovídají obsahu zdrojového obrázku a zároveň stylu cílového.

### Vysvětlit základní fungování generative adversarial networks (podle rozsahu v textu na webu)

**GAN (Generative Adversarial Network)** se skládá ze dvou sítí:

- **Generátor (G)** – snaží se generovat realistické obrázky.
- **Diskriminátor (D)** – snaží se rozlišit, zda je obrázek reálný (z trénovacích dat), nebo falešný (z generátoru).

Obě sítě se trénují proti sobě:

$$\min_G \max_D E_{x\sim p_{\text{data}}}[\log D(x)] + E_{z\sim p_z}[\log(1 - D(G(z)))]$$

Po trénování se používá pouze generátor $G$ pro generování nových vzorků. GANy jsou schopny tvořit velmi realistické obrázky a používají se i pro stylizace a přenos stylu.

## 12 Neuroevoluce

### Popsat použití evolučních algoritmů pro trénování vah v neuronové síti

Při fixní architektuře sítě lze trénovat její váhy pomocí evolučních strategií. Každý jedinec reprezentuje vektor vah, který se mění mutacemi:

$$w_i \leftarrow w_i + \mathcal{N}(0, \sigma^2)$$

Tento přístup je dobře paralelizovatelný a vhodný zejména pro:

- prostředí s řídkou odměnou (delayed reward),
- trénink bez gradientu (black-box optimalizace),
- distribuované výpočty bez potřeby GPU.

### Popsat algoritmus NEAT – reprezentace jedince, operátory

NEAT (NeuroEvolution of Augmenting Topologies) vyvíjí jak váhy, tak i strukturu sítě. Jedinec je seznam spojení:

- **Spoj:** (zdroj, cíl, váha, aktivní/neaktivní, inovační číslo)

**Mutace:**

- Přidání spoje (nové inovační číslo)
- Přidání neuronu – rozdělí stávající spoj na dva

**Křížení:**

- Geny se párují podle inovačních čísel.
- Geny v obou rodičích: náhodně vybrány.
- Geny pouze v jednom rodiči: převzaty z lepšího jedince.

## Vysvětlit význam inovačních čísel v algoritmu NEAT

Inovační číslo slouží k identifikaci původu spojení – dvě spojení se stejným inovačním číslem mají společnou historii a lze je zarovnat při křížení. Pomáhá to:

- rozlišit homologické (odpovídající) spoje,
- spravedlivě kombinovat rozdílné struktury.

## Popsat myšlenku algoritmu HyperNEAT

HyperNEAT rozšiřuje NEAT tak, že místo přímého vyvíjení vah používá generující síť (CPPN – compositional pattern producing network). Ta pro každé spojení dostane na vstup souřadnice zdrojového a cílového neuronu a na výstup vrací váhu spoje.

Výhody:

- Mnohem pravidelnější struktury.
- Možnost vývoje větších sítí.

## Popsat rozšíření algoritmu NEAT pro hledání architektur neuronových sítí (algoritmus CoDeepNEAT)

CoDeepNEAT:

- Jedinci reprezentují strukturu neuronové sítě na úrovni modulů (bloků).
- Každý uzel jedince odkazuje na předdefinovaný nebo evolučně vytvořený blok.
- Kombinací těchto bloků vzniká výsledná architektura sítě.

Používá se pro NAS (Neural Architecture Search) – návrh topologií před tréninkem gradientní metodou.

## Vysvětlit myšlenku algoritmu Novelty Search

Novelty Search neoptimalizuje přímo fitness, ale maximalizuje odlišnost chování:

$$\text{novelty}(x) = \frac{1}{k} \sum_{i=1}^{k} \text{dist}(x, x_i)$$

kde $x_i$ jsou nejbližší jedinci v archivu chování.

Použití:

- prostředí s mnoha lokálními optimy,
- kreativní úlohy (např. generování map, robotických strategií).

### Porovnat Novelty Search a evoluční algoritmy a diskutovat jejich výhody/nevýhody

**Klasické EA:**

- + Optimalizují konkrétní cíl.
- Náchylné k uvíznutí v lokálních optimech.

**Novelty Search:**

- + Lepší explorace prostoru.
- + Nachází řešení, na která by fitness vůbec nevedla.
- Může generovat "divná" řešení, která nic neřeší.

## 13 Hluboké zpětnovazební učení

### Vysvětlit algoritmus hlubokého Q-učení

Hluboké Q-učení (DQN) aproximuje Q-funkci neuronovou sítí $Q_\theta(s, a)$. Minimalizuje se chybová funkce:

$$\mathcal{L}(\theta) = E_{(s,a,r,s')}\left[\left(r + \gamma \max_{a'} Q_{\theta^-}(s', a') - Q_\theta(s, a)\right)^2\right]$$

kde $Q_{\theta^-}$ je cílová síť (target network), aktualizovaná méně často.

### Porovnat Q-učení a hluboké Q-učení

- **Q-učení:** udržuje tabulku $Q(s, a)$.
- **DQN:** využívá neuronovou síť pro aproximaci $Q$.

DQN se lépe škáluje na prostředí s velkým nebo spojitým stavovým prostorem.

### Popsat triky s experience replay a target network

- **Experience replay:** Jednotlivé přechody $(s, a, r, s')$ získané při interakci s prostředím se ukládají do paměti (bufferu). Při trénování se pak náhodně vzorkují minibatche z této paměti. Tento přístup narušuje časovou závislost mezi po sobě jdoucími přechody a umožňuje efektivnější využití předchozích zkušeností, čímž se snižuje rozptyl v trénovacích datech a zvyšuje stabilita učení.
- **Target network:** Používají se dvě neuronové sítě – primární síť $Q_\theta$ pro aktuální aproximaci hodnoty $Q$ a cílová síť $Q_{\theta^-}$ pro výpočet cílových hodnot při učení. Cílová síť má zafixované parametry $\theta^-$, které se kopírují z $\theta$ jen periodicky. Tím se snižuje riziko oscilací nebo divergence v důsledku neustále se měnících cílových hodnot.

## Vysvětlit vliv experience replay a target network na hluboké Q-učení

Obě techniky řeší zásadní problémy při trénování neuronové sítě v rámci Q-učení, kde se vstupy i výstupy neustále mění:

- **Experience replay** zajišťuje, že trénovací data nejsou silně korelovaná v čase (např. podobné stavy jdoucí po sobě), což by vedlo k nestabilnímu učení. Díky náhodnému výběru přechodů je učení robustnější a efektivněji využívá celé spektrum předchozích zkušeností.
- **Target network** zabraňuje situaci, kdy se při trénování síť snaží přizpůsobit svým vlastním, právě měnícím se predikcím. Pomocí zafixované sítě $Q_{\theta^-}$ se cílové hodnoty nemění v každém kroku, což stabilizuje výpočet ztráty (loss) a tím i aktualizace vah.

Výsledkem použití těchto technik je výrazně stabilnější a spolehlivější trénování hluboké neuronové sítě v rámci algoritmu DQN.

## Popsat algoritmus DDPG

- Pro spojité akce používá dvě sítě:
  - $Q_\phi(s, a)$ – hodnota,
  - $\mu_\theta(s)$ – deterministická politika.
- Cílová síť: $\phi^-$ a $\theta^-$
- Chybová funkce:

$$\mathcal{L}(\phi) = E_{(s,a,r,s')}\left[\left(r + \gamma Q_{\phi^-}(s', \mu_{\theta^-}(s')) - Q_\phi(s, a)\right)^2\right]$$

- Politika se trénuje jako gradient maximum hodnoty:

$$\nabla_\theta J(\theta) = E_s\left[\nabla_a Q_\phi(s, a)|_{a=\mu_\theta(s)} \nabla_\theta \mu_\theta(s)\right]$$

## Popsat actor-critic přístupy

Actor-Critic přístupy kombinují výhody hodnotových metod (např. Q-učení) a metod založených na politice (policy gradient). Cílem je efektivněji trénovat agenta pomocí dvou sítí:

- **Actor:** neuronová síť $\pi_\theta(a|s)$, která přímo modeluje politiku, tj. pravděpodobnost výběru akce $a$ ve stavu $s$. Tato síť se trénuje pomocí gradientu politiky, který říká, jak změnit parametry $\theta$ tak, aby zvyšovala očekávanou odměnu.
- **Critic:** druhá síť, která odhaduje hodnotu stavu nebo hodnotu akce:

  - buď hodnotovou funkci stavu $V_\phi(s)$ (state-value function),
  - nebo akčně-stavovou funkci $Q_\phi(s, a)$.

  Jejím úkolem je říct, jak „dobrý“ je daný stav nebo akce – tedy poskytuje zpětnou vazbu (kritiku) pro aktora.

- Cílem je maximalizovat očekávanou diskontovanou odměnu. Pokud používáme advantage $A(s_t, a_t)$, pak gradient politiky vypadá jako:

$$\nabla_\theta J(\theta) = E_{s_t,a_t} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot A(s_t, a_t)\right]$$

  Tímto způsobem actor zvyšuje pravděpodobnost akcí, které vedly k vyššímu zisku oproti očekávání (dle kritika).

## Popsat advantage a zdůvodnit, proč se používá

**Advantage** (výhoda) je veličina, která vyjadřuje, jak moc byla konkrétní akce $a$ ve stavu $s$ lepší než průměrné očekávání. Je definována jako rozdíl mezi hodnotou akce a hodnotou stavu:

$$A(s, a) = Q(s, a) - V(s)$$

Ve variantách, kde máme k dispozici pouze funkci $V(s)$, lze advantage aproximovat z přechodu:

$$A(s, a) \approx r + \gamma V(s') - V(s)$$

**Proč se používá:**

- Výrazně snižuje rozptyl gradientního odhadu při trénování politiky.
- Pokud by se místo advantage používaly přímo kumulované odměny, mohly by být hodnoty velmi rozkolísané (vysoký rozptyl), což by vedlo k neefektivnímu učení.
- Advantage koriguje tento problém tím, že hodnotí, jak moc konkrétní akce překročila (nebo nedosáhla) očekávání, a tím poskytuje stabilnější signál pro trénování.

## Popsat metodu A3C

**A3C** (Asynchronous Advantage Actor-Critic) je efektivní algoritmus pro trénování agentů pomocí paralelních procesů, který se dobře škáluje na vícejádrové procesory. Je založen na actor-critic přístupu, ale využívá několik klíčových zlepšení:

- **Asynchronní paralelismus:** Místo jediné instance agenta se spustí více agentů paralelně v různých kopiích prostředí. Každý agent má svou kopii sítě, která se průběžně trénuje.

- **Sdílení váhové centrální sítě:** Po několika krocích trénování odesílají agenti své gradienty do centrální verze neuronové sítě. Ta se aktualizuje pomocí agregovaných gradientů, a aktualizované váhy se rozesílají zpět.
- **Využití advantage:** Pro výpočet aktualizací politiky se používá odhad advantage:
$$A(s_t, a_t) = r_t + \gamma V(s_{t+1}) - V(s_t)$$
nebo delší n-kroková verze odměny.
- **Bez potřeby experience replay:** Díky asynchronnímu běhu agentů se trénovací data od různých agentů přirozeně liší a nejsou korelovaná, čímž se eliminuje potřeba bufferu zkušeností.

**Výhody A3C:**

- Lepší efektivita a škálovatelnost na vícejádrové systémy.
- Stabilnější učení díky rozmanitosti vstupů.
- Vhodné pro prostředí, kde nelze jednoduše použít experience replay (např. s nestabilními nebo proměnlivými stavy).

# 14 Artificial Life

## Vysvětlit rozdíly mezi jednotlivými typy umělého života (soft, hard, wet)

- **Soft ALife** – simulace života na počítači, modelování interakcí mezi organismy.
- **Hard ALife** – reálná fyzická realizace životních principů, např. roboti.
- **Wet ALife** – zkoumá biochemické procesy, cílem je vytvořit umělou DNA nebo chemický život.

## Vysvětlit rozdíly mezi silným a slabým umělým životem

- **Silný ALife** – tvrdí, že život je možné vytvořit v jakémkoliv médiu (i digitálním).
- **Slabý ALife** – považuje simulace jen za nástroje ke studiu reálného života, ne za skutečný život.

## Popsat 1D a 2D celulární automaty

- **1D automaty** – každý prvek (buňka) má sousedy vlevo a vpravo, stav se aktualizuje podle pravidla.
- **2D automaty** – buňky jsou uspořádány do mřížky, stav se mění podle okolních buněk (např. Game of Life).

## Popsat pravidla Game of Life

- Živá buňka s méně než dvěma nebo více než třemi sousedy umírá.
- Živá buňka se dvěma nebo třemi sousedy přežívá.
- Mrtvá buňka se třemi sousedy ožívá.

## Popsat pravidla Langtonova mravence

- Mravenec se pohybuje po mřížce se dvěma barvami.
- Na každém kroku:
  - Přepne barvu políčka.
  - Pokud byla barva bílá, otočí se doprava.
  - Pokud byla barva černá, otočí se doleva.
  - Posune se o jedno políčko vpřed.

## Vysvětlit pojem dálnice u Langtonova mravence

Po počáteční fázi chaosu se mravenec ustálí do periodického chování (104 kroků), kterému se říká **dálnice**. Mravenec se tímto způsobem pohybuje stále jedním směrem.

## Popsat systém Tierra

Tierra je simulátor digitálního života, kde jedinci jsou krátké programy běžící na emulovaném počítači s 32 instrukcemi. Podporuje paralelní běh, kopírování, mutace a evoluci.

## Popsat jedince v systému Tierra

Jedinec je program schopný samoreplikace. Má instrukce pro výpočty, skoky a dvě varianty NOP instrukcí (NOP0 a NOP1), které se používají při adresování.

## Vysvětlit způsob adresování (definování cílů skoku) v systému Tierra

Namísto adres systém používá:

- Skoková instrukce hledá nejbližší **komplementární** posloupnost NOP instrukcí (NAP0 $\leftrightarrow$ NOP1).
- Inspirace komplementárními páry v DNA.

## Popsat příklady vytvořených jedinců v systému Tierra (paraziti)

- **Paraziti** – nemají vlastní kód pro kopírování, využívají kód jiných.
- **Obrana proti parazitům** – evolucí vznikají jedinci odolní proti parazitům.
- **Hyper-paraziti** – „přesvědčí“ parazity, aby kopírovali je.