PŘÍRODOU INSPIROVANÉ ALGORITMY

- Učení s učitelem – máme data s labely
  - ↳ klasifikace
  - ↳ regrese … chceme nějakou spojitou hodnotu
- Učení bez učitele – data bez labelů
  - ↳ clustering
  - ↳ generování podobných dat
- Zpětnovazebné učení
  - → agent se má naučit chovat se co nejlépe v daném prostředí
  - → zpětná vazba = reward za akce (skóre ve hře)
- Zpětnovazebné učení
  - → agent dělá akce, za to dostává odměny a mění tím stav prostředí

Def: Markovský rozhodovací proces je čtveřice (S, A, P, R)

- S… stavy
- A… akce
- P… $P_a(s, s')$ = pst. přechodu do s' při provedení a ve stavu s
- R… $R_a(s, s')$ = odměna za →

Markovský předpoklad: P závisí pouze na a, s (ne na předchozích akcích…)

Def: Policy agenta je $\pi: S \times A \to [0,1]$. Píše se $a \sim \pi(s)$

↳ $\pi(a, s)$ = pst, že v s provedu a ↳ pstní distribuce

⇒ Cíl je maximalizovat celkovou odměnu

$$R^\pi = \sum_{s=0}^{\infty} \gamma^s \underbrace{R_{a_s}(s_s, s_{s+1})}_{r_s}, \quad a_s \sim \pi(s_s).$$

↑ $\gamma < 0$ … diskontní faktor ⇒ aby to konvergovalo

Def: Hodnota stavu s je $V^\pi(s) := \mathbb{E}[R^\pi \mid s_0 = s]$

Hodnota akce ve stavu je $Q^\pi(s, a) := \mathbb{E}[R^\pi \mid s_0 = s \;\&\; a_0 = a]$

⇒ cíl: Najít π^*, aby $V^{\pi^*}(s) = \max_{\pi} V^{\pi}(s)$.

→ z modice Q může přečíst nejlepší akci, ale potom málo exploruje

- **ε-greedy policy:** Akce bude s pravděpodobností

  $(1-\varepsilon)$ ... $\operatorname{argmax}_a Q(s,a)$ ... exploitace

  ε ... náhodná akce ... explorace

- **Monte-Carlo metody** → pro natrénování Q

  → dělám hodně simulací – mám Q^π, chci $Q^\pi(s,a)$ vylepšit

  ⇒ provedu ve stavu s akci a a pokračuji π, dokud nedojdu do cíle

  ↳ někam si zapíšu, kolik to vyjde

  ⇒ nakonec z těch hodnot udělám hladově nový Q

- **Q-learning**

Bellmanovy rovnice:

$$V^\pi(s) = \mathop{\mathbb{E}}_{a,s_1}\left[r_0 + \sum_{t=1}^{\infty} \gamma^t r_t \mid s_0 = s\right] = \mathop{\mathbb{E}}_{a,s_1}\left[r_0 + \gamma V^\pi(s_1) \mid s_0 = s\right]$$

$$= \sum_a \pi(s,a) \cdot \sum_{s'} P_a(s,s') \cdot \left(R_a(s,s') + \gamma \cdot V^\pi(s')\right)$$

↳ možné akce, možné příchody r_0

Zlepšení: provedu v s akci a, dostanu r a přenesu se do s'

$$V(s) \leftarrow V(s) + \alpha\left(r + \gamma V(s') - V(s)\right)$$, α je parametr učení

→ Q-learning vylepšuje Q^π, funguje stejně

$$Q^\pi(s,a) = \mathop{\mathbb{E}}_{s'}\left[r_0 + \gamma \mathop{\mathbb{E}}_{a'}\left[Q^\pi(s',a')\right]\right]$$

$$= \sum_{s'} P_a(s,s') \cdot \left(R_a(s,s') + \gamma \sum_{a'} \pi(s',a') \cdot Q^\pi(s',a')\right)$$

π = vezmi nejlepší akci z Q pro daný stav

Zlepšení:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)\right)$$

learning-rate; nová hodnota; původní hodnota

↳ vezmu nejlepší akci co mám

- **SARSA**

→ Q-learning + té smyčce pro zlepšení vezme nejlepší akci podle Q matice

↳ policy je implicitně určena Q maticí ⇒ je to ~~off-policy alg.~~

→ sarsa místo argmax vezme akci danou policy:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(r + \gamma Q(s',a') - Q(s,a)\right)$$

$a \sim \pi(s)$ $\qquad$ $a' \sim \pi(s')$

⇒ Q-learning vezme $a := \underset{a}{\text{argmax}}\, Q(s,a)$

Problémy: Může být hodně stavů, spojité stavy / akce ⇒ diskretizace

↳ Q nejde natrénovat

Deep Q-learning – DQL

→ řeší problém hodně stavů, místo Q-matice má neuronku

⇒ myšlenka: neuronka dostane stav a vrátí vektor odměn

↳ pro ∀ akci predikuje odměnu ⇒ vlastně řádek Q matice

↳ policy = vezme nejlepší akci

→ nechám ho hrát a ukládám si hodnoty

s ... kde jsem byl
a ... co jsem udělal
s' ... kam jsem se dostal
r ... odměna

} experience buffer T

↳ sbírané za několik her

→ parametry té neuronky

⇒ chci $Q^\theta(s,a) = r + \gamma \max_{a'} Q^\theta(s',a')$

↳ $Q^\theta(s,a)$ znamená: výhodnost sítě s θ ve stavu s a vem hodnotu akce a

⇒ loss $= \sum_{(s,a,s',r)\in T} \left(r + \gamma \max_{a'} Q^\theta(s',a') - Q^\theta(s,a)\right)^2$ ... MSE

$\max_{a'} Q^\theta(s',a')$ … odhad, $Q^\theta(s,a)$ … realita

- **Target network**: abych si ty parametry θ neměnil pod rukama při trénování

Sám mám 2 sítě: Q^{θ^-} a Q^θ

⇒ vylepšuju θ ale beru akci $\max_{a'} Q^{\theta^-}(s',a')$

⇒ jednou za čas nastavím $Q^{\theta^-} := Q^\theta$

• DDPG = Deep Deterministic Policy Gradient

→ řeší spojité akce: „otoč volantem o 17°" místo „rohni doprava"

⇒ máme 2 sítě

Q^ϕ ... učí se odměnu

μ^ϕ ... učí se akci → $\mu^\phi(s)$ vrátí akci a, co maximalizuje Q ↓ vlastně policy

→ loss Q = $\sum_{(s,a,s',r)\in T} (r + \gamma Q^\phi(s', \mu^\phi(s')) - Q^\phi(s,a))^2$

→ učení μ: chci aby dávala co nejlepší akci pro stav s

⇒ max $\mathbb{E}_s[Q(s, \mu(s))]$ pomocí gradient-descent

• Policy gradient metody

→ policy ~ síť (funkce) s parametry ϕ

↳ chceme maximalizovat celkovou odměnu $\mathbb{E}$[celková odměna za běh]

⇒ dá se najít gradient téhle věci a optimalizovat to přímo

↳ vyskytují se tam kumulované odměny $G_t = r_t + r_{t+1} + \dots r_T$ → konec

! Když jsou G_t velké, tak rozptyl hodnot ∇ může být velký

⇒ trénuje se to špatně

• Actor-critic

→ místo G_t používáme něco jiného ... třeba přímo $Q^\phi(s,a)$ z DQL

→ nebo advantage:

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

↳ generická odměna za stav

↳ odměna co mi přinese konkrétní akce

→ síť pro π ... actor, vybírá akce → trénuje pomocí

→ síť pro V ... critic ... říká jak dobré jsou stavy, kam jsem se dostal

↳ A se dá vyjádřit bez Q ⇒ nepotřebuju síť pro Q

• Asynchronous Advantage Actor Critic – A3C

↳ paralelizace předešlého ... hraje víc her najednou, průměruje váhy

EVOLUČNÍ ALGORITMY

- **Genetický algoritmus**

jedinec = posloupnost 0 a 1

→ příklad: Součet podmnožiny S ... chci $S' \subseteq S$ aby $\sum S' = k$

$$fitness = -(k - \sum x_i s_i)^2 \quad \ominus \because \text{chci max fit}$$

jedinec = $\{0,1\}^{|S|}$

Algoritmus:

1. $P_0 \leftarrow$ náhodná populace
2. While not happy:
3. $f \leftarrow$ fitness (P_t)
4. Pro $i = 0, \dots |P|/2$:

$\mu_1, \mu_2 \leftarrow$ selekce (P_t, f)

$\sigma_1, \sigma_2 \leftarrow$ křížení (μ_1, μ_2)

$s_1 \leftarrow$ mutace (σ_1)

$s_2 \leftarrow$ mutace (σ_2)

$P_{t+1} = P_{t+1} \cup \{s_1, s_2\}$

→ buď musím takhle postupně dělat novou populaci

→ nebo nejprve udělat mating-pool

- **Selekce:** Ruletová

$$p_i = \frac{f_i}{\sum_j f_j}$$

×

Turnajová

1. $\mu_1, \mu_2 \leftarrow$ náhodní jedinci
2. vyberu ten s větší fitness

- **Křížení**

1) uniformní: beru náhodně bity z těch rodičů

2) jednobodové

3) n-bodové

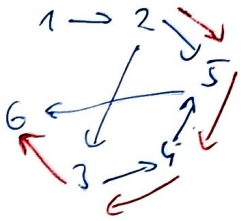
↳ klidně i více rodičů

- **Mutace:** s nějakou šancí flipnu i-tý bit

Složitější kódování jedinců

- kategorie $k \in [n]$ ... zobecněný bit
- permutace hodnot $(1, 2, \dots n)$
  - ↳ třeba pamatování si cesty v grafu – obchodní cestující

– mutace:
- prohodím 2 hodnoty
- shift podposloupnosti: 12345 → 14235
- rotace ——||—— : 123456 → 125436



↳ rozmístění cesta ⇔ obch. cest

– křížení = crossover
- skládání permutací ... velká změna
- rozložení na cykly a pak je poskládáme } nic moc
- OX (order crossover) ↳ ~ 2-bodové k.

```
12 3 | 4 5 6 7 | 8 9
4 5 2 | 1 7 9 3 | 6 8
-----------------------
2 4 5 | 1 7 9 3 | 6 8
2 1 9 | 4 5 6 7 | 3 8
```

→ doplním to, aby to byla permutace

- PMX (Partially mapped crossover)

```
1 2 3 | 4 5 6 7 | 8 9
4 5 2 | 1 7 9 3 | 6 8
-----------------------
4 2 5 | 1 7 9 3 | 8 6
      ↕ ↕ ↕ ↕
1 3 2 | 4 5 6 7 | 9 8
```

→ to co lze, opíšu
→ jinak mi řeší prostřední segment definuje mapping

- ER (Edge Recombination) – super pro obchodního cest.

→ pro ∀ vrchol udělám seznam sousedů v obou rodičích

```
* 1: 2 9 7
* 2: 1 3 5
* 3: 2 4 9 6
* 4: 3 5 8
* 5: 4 6 2
* 6: 5 7 3 8
* 7: 6 8 1 9
  8: 7 9 6 4
  9: 1 8 7 3
```

↳ vybírám vrcholy s nejmíň sousedy → pokud stejní ⇒ náhodně
→ pak procházím seznamy

4 → 5 → 2 → 3 → 6 → 1 → 7 → 8 → 9

- **Spojitá optimalizace** * spíš jedinec $\in$ n-rozměrný kompaktní interval

jedinec $\in \mathbb{R}^n$, fitness $f: \mathbb{R}^n \to \mathbb{R}$ ($\mathbb{R}$ ↑)

↳ f nemusí být spojitá, ale ten prostor ano

- křížení
  - 1-bod, 2-bod ... nic moc
  - aritmetické

$$\vec{o}_1 = w\vec{r}_1 + (1-w)\vec{r}_2$$
$$\vec{o}_2 = (1-w)\vec{r}_1 + w\vec{r}_2$$

konvexní obal ↑

1) $w \in (0,1) \Rightarrow$ konvergují k průměru → potomci $\in$ hull (populace)

2) $w \in (-1,2) \to$ musí se to dostat k konvexního obalu

- mutace

1) unbiased ... prostě vygeneruje nové číslo z toho intervalu

2) biased ... upraví hodnotu na dané pozici

↳ $x_i \leftarrow x_i + \sigma \cdot N(0,1) \to$ normální rozdělení

(σ ↓ konstanta)

- **Evoluční strategie**

– spolu s jedinci evolvuje i nějaké hyperparametry toho algoritmu

→ třeba ten σ → jak ho zvolit?

1) moc velké ⇒ vlastně dělám unbiased

2) moc malé ⇒ pomalu to konverguje

→ dává smysl postupně zmenšovat ⇒ v t-té generaci $\sigma *= 0.99$

1/5-rule: chci, aby cca $\frac{1}{5}$ potomků byla lepší než rodiče

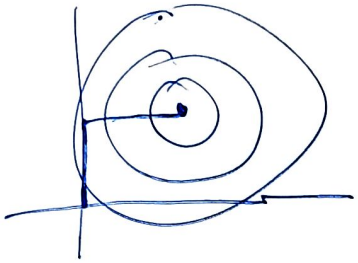
↳ pokud jich je moc lepších ⇒ zvednout σ

↳ málo ⇒ snížit σ

... $\sigma \approx 0 \Rightarrow \frac{1}{2}$ lepší ale nikam se nedostanu

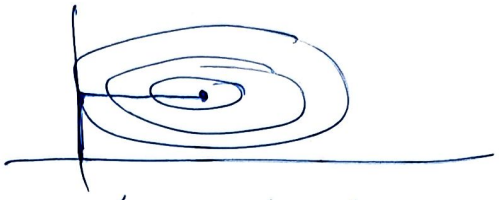
• Neseparabilní funkce

→ funkce je neparobilní ≡ lze jednotlivé proměnné optimalizovat nezávisle na sobě



$x_i += \vec{\sigma} N(0,1)$

↳ předpokládám, že fce roste stejně ve všech směrech

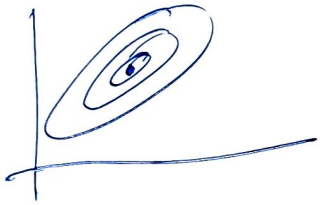


separabilní ← lze optimalizovat po složkách

↳ lepší mít σ_i pro $\forall x_i$

⇒ potom v evoluční strategii vyvíjím vektor $\vec{\sigma}$

↳ neseparabilní:



→ lze řešit kovariancí matici

↳ říká, jak na sobě závisí dvojice proměnných

• Diferenciální evoluce .. způsob jak se vypořádat s neseparabilní

→ předpokládám, že populace má podobný tvar jako ta fce

→ vyberu 4 rodiče $\mu_1, \mu_2, \mu_3, \mu_4$

$\vec{\sigma}' \leftarrow \vec{\mu}_1 + C \cdot (\vec{\mu}_2 - \vec{\mu}_3)$ ... $c \in (0,2)$, často 0.8

$\vec{\sigma} \leftarrow$ uniformní křížení (σ', μ_4, α)

↳ pst, že vyberu z σ'

↳ většinou forcenu aspoň 1

pokud fit(σ) > fit(μ_4):

nahradí μ_4 v populaci jedincem σ

⇒ v podstatě to je mutace μ_4

GENETICKÉ PROGRAMOVÁNÍ – Lineární

jedinec = posloupnost instrukcí v nějakém jednoduchém prg. jazyce
↳ jazyk "slash /A" ... jednoduchý assembler (na Github)

- 2 registry – F float
  I int ... používá se na adresaci
- pamět ... pole

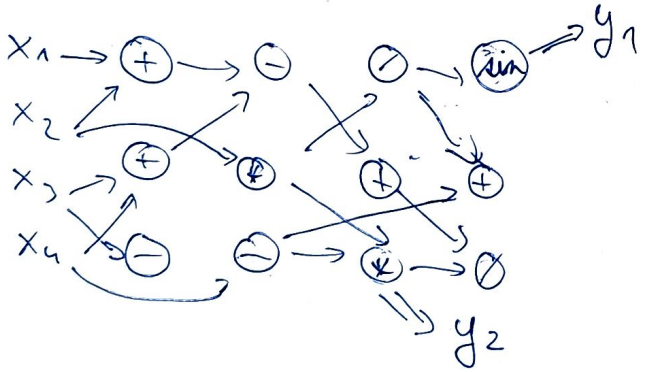
instrukce: input, save, add, output, n, jump
↳ F ← input ↳ I ← n

↳ když jumpům se může zacyklit ⇒ po nějaké době je lepší zabít

- křížení: jednobodové, n-bodové
- mutace: změna instrukce

- **Kartézské GP** *

→ mřížka $m \times n$



→ jedinec:
- * buňka
  1) funkce co počítá
  2) seznam vstupů
- pro požadované výstupy seznam buněk na kterých se počítají

- **Gramatická evoluce** *

→ náš prog. jazyk = bezkontextová gramatika

$Expr \to Var \mid (Expr + Expr) \mid (Expr * Expr) \mid \sin(Expr)$ (0, 1, 2, 3, mod 4)

$Var \to x_1 \mid x_2 \mid x_3$ (0, 1, 2, mod 3)

→ jedinec = posloupnost čísel: 2 4 5 2 3 1 2 4 1 2

→ vždy rozvíjím nejlevější neterminál

$E \xrightarrow{2} (E * E) \xrightarrow{4 \equiv 0} (V * E) \xrightarrow{5 \equiv 2} (x_3 * E) \xrightarrow{2} (x_3 * E * E) \xrightarrow{3} (x_3 * \sin(E) * E)$

→ **problémy:**
a) na konci může obsahovat neterminály ⇒ udělám jedince dost dlouhého
b) špatně se dělají mutace a křížení – 1 změna ⇒ úplně jiný výsledek

- **Symbolická regrese** – učení s učitelem

* Vstup: množina dvojic $(\vec{x}, y)$ ⇝ chci najít f aby $f(\vec{x}) = y$

⇒ minimalizují $\sum (f(x) - y)^2$ … MSE

→ vymyslím operátory, co umím funkčně: $+, -, *, \exp, \sin, \ldots$

→ řešení řeším Kartézským GP / Gram. ev. nebo SGP

- **Stromové genetické programování**
  - terminály: vstupy $x_1, x_2, \ldots$
    konstanty: $-1, 0, 1, 2$ … jen ty nejdůležitější
    ↳ další si může vyrobit
  - neterminály: $+, -, \ldots$

→ **generování stromu**

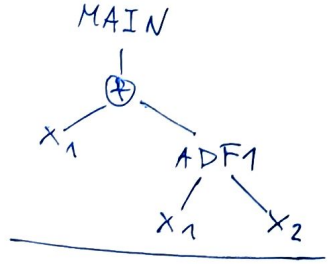
1) **full**: náhodně vyrobí strom dané hloubky → vše tam musí být terminál
2) **grow**: ——"—— s daným # neterminálů

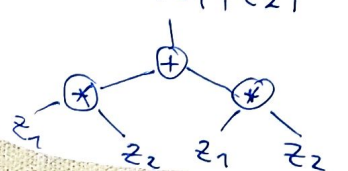
⇒ většinou se to používá půl na půl

- **křížení**: prohození podstromů
- **mutace**:
  1) nahrazení n-árního neterminálu za jiný ⊖ → ⊛
  2) nahrazení terminálu
  3) zmenšení stromu: nahrazení neterminálu jeho listem
  4) změna stromu: ——"—— náhodným stromem (malým)

→ **omezení velikosti jedince** < penalizace fitness
< nastavení limitu na hloubku / # net.

- **automaticky def. fce** – ADF





→ algoritmus může vytvářet nové neterminály – ADF
→ pro ADF nějak omezíme povolené term. a net.
→ jedinci křížíš uvnitř MAINy i ADFka
→ ADF se teoreticky mohou volat, ale musíme se vyhnout cyklům

- **Typované GP**
  - ke každému terminálu dám navíc typ
  - neterminály obsahují informace co berou a vracejí

    $<, >, ==, !=$ (float, float) → bool

    $+, -, *, /$ (float, float) → float

    $||, \&\&$ (bool, bool) → bool

    if-then-else (bool, float, float) → float → sémantický problém
  - křížení a mutace stejné, ale musím vracet typově konzistentní

- **Evoluce pravidel**
  - chci objekty rozřadit do tříd ... třeba stav hry → tah
  - → množina pravidel: podmínky na stav → tah
  - → jedinec = vektor vah k pravidlům
    - ↳ vyhodnocení = vybere tah co má největší celkovou váhu

NEURONOVÉ SÍTĚ

- preprocessing dat → standardně nějaké $x_1, \dots x_n \to y$
- číselné příznaky → naškálovat na interval $[0,1]$
  - ↳ normalizace = odečíst $\overline{x_n}$ a podělit std. dev.
- kategorické příznaky → např. 5 kategorií a $x_i = 2 \Rightarrow (0,1,0,0,0)$
- cílová hodnota – klasifikace ⇒ kategorie
  - regrese ⇒ číselná hodnota

Def: Softmax: $x_1, \dots x_n$: $x_i \mapsto \frac{e^{x_i}}{\sum_i e^{x_i}} \in (0,1)$, součet = 1

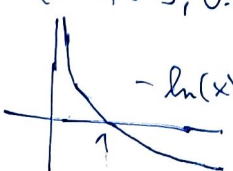
↳ sen největší nejvíc přidá

→ loss fce se jmé crossentropy

↳ chci aby se na ní roklasifikovala do správné třídy

$\Rightarrow y = (0,0,1,0)$

$p = (0.1, 0.3, 0.5, 0.2) \Rightarrow$ loss $= -\ln(0.5) = -\sum_i y_i \ln(p_i)$



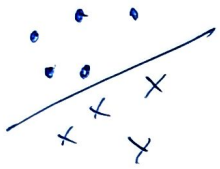
• Perceptron

počítá nejprve aktivaci $\xi := b + \sum_i x_i w_i$

⇒ pak aplikuje aktivační fci $f: \mathbb{R} \to \mathbb{R}$

→ f může být třeba $x \mapsto \begin{cases} 0, & x \le 0 \\ 1, & x > 0 \end{cases}$

→ tohle dělá separaci dat nadrovinou v prostoru $\mathbb{R}^n$



$n=3$: $ax_1 + bx_2 + cx_3 + bias = 0$

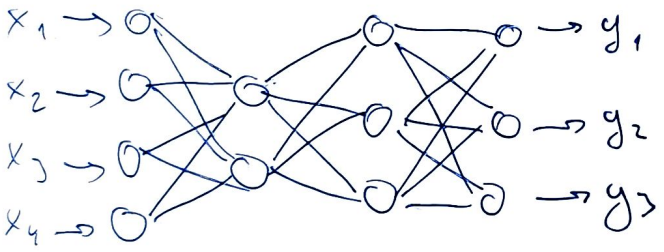
$f(\dots) = 0 \Rightarrow$ nad

$f(\dots) = 1 \Rightarrow$ pod

učení: $w_i \leftarrow w_i + r \cdot (y - y_0) x_i$ … $(y - y_0) = \begin{cases} 0 \dots \text{správně} \\ 1 \dots \xi \text{ je málo} \\ -1 \dots \xi \text{ je moc} \end{cases}$

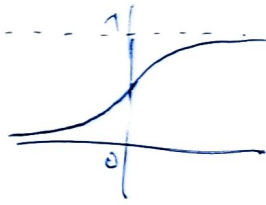
r … učení, y … label, y_0 … výstup, upravím

• Dopředné neur. sítě = vícevrstvý perceptron = MLP



aktivační fce:

- $\text{relu}(x) = \max(0, x)$
- $\text{sigmoid}(x) = \frac{1}{1 + e^{-\lambda x}}$
- $\tanh(x)$



… $\vec{x}$ a $\vec{w}$ určují výsledek

Gradient descent – derivuju loss fci $L(\vec{x}, \vec{y}, \vec{w})$ podle vah

„backpropagation" a jdu ve směru záporného gradientu ⇒ do minima

⇒ $w_i \leftarrow w_i - \alpha \cdot \frac{\partial \text{loss}}{\partial w_i}$ → target

w_{jk} = váha mezi j a k

$\xi_k = \sum_j w_{jk} x_j$

⇒ příklad MSE: $L = \frac{1}{2} \sum_i (y_i - t_i)^2$

Co derivuju podle vah mezi poslední skrytou vrstvou a výstupní vrstvou

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

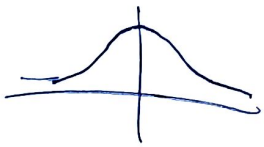
↳ záleží na f

→ pro skryté vrstvy existují nějaký rekurentní vzorec

• RBF sítě – Radial Basis Functions

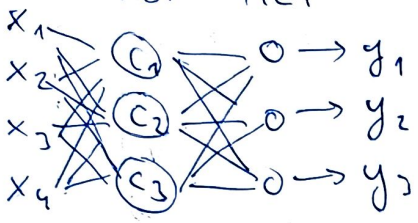
MLP ... neuron počítá $f(\sum_i w_i x_i)$

RBF ... $\rho(\|\vec{X} - \vec{C}\|)$ a aktivační fce $S(x) = e^{-\beta x^2}$ → Gaussovka



↓ kernel ↓ vstup → střed neuronu, $\|\cdot\|$ je norma ... třeba Eukl.

RBF MLP



→ C_i jsou ideálně nějaké středy clusterů

→ je to robustnější vůči změnám vstupů

→ trénování:

1) K-means pro nastavení středů a parametru β

↳ když už mám data rozdělená do clusterů

↳ pro i-tý neuron (střed) $\beta_i := \frac{1}{2\sigma_i}$, kde σ_i = průměrná vzdálenost dat v daném shluku od středu c_i

2) Trénování výstupní vrstvy

→ jen 1 vrstva a počítá něco lineárního ⇒ stačí lineární regrese

Algoritmus k-means – učení bez učitele

→ hledá clustery v datech ... pro pevný počet clusterů k

1. náhodně vyber k bodů jako středy
2. while not happy:
3. přiřaď i-tý bod joint k nejbližšímu středu
4. přepočítej středy (střed = průměr dat k němu přiřazených)

→ chci 10 RBF jednotek ⇒ $k = 10$, σ_i = průměrná vzdálenost dat příslušných ke středu c_i od něj

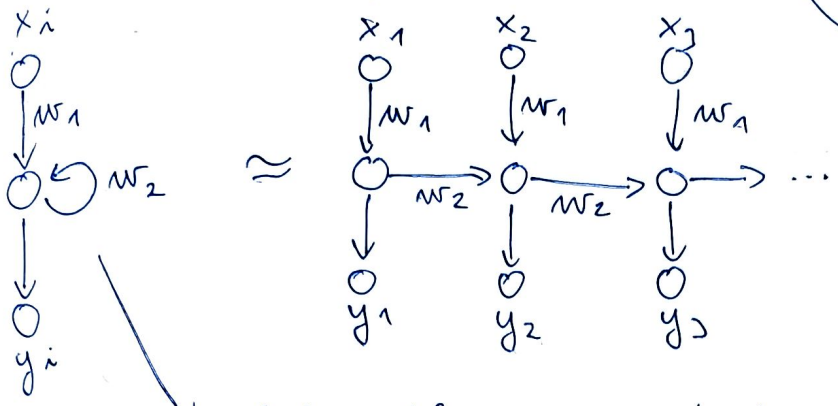
Rekurentní neuronové sítě

→ někde v tom grafu je cyklus

Vstup: nějaká posloupnost $x_1, x_2, x_3, \ldots$ → neznám délku – mění se

→ třeba slova z user inputu

→ konec = nějaký ukončovací token
↳ ve větě tečka .



→ výstup toho neuronu dostane jako vstup v dalším kroku

Trénování:

→ rozvinu síť v čase a dělám gradient descent ⇒ back prop. through time

! problém: při gradient descent se gradient z předchozí vrstvy násobí váhou

$w_2 < 1$ … vanishing gradient

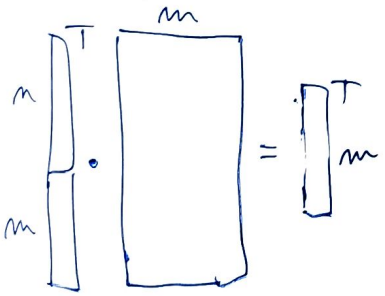
$w_2 > 1$ … exploding gradient

řešení ┌ nebudu to učit … ESN
└ nastavím na 1 a řeším to jinak … LSTM

Echo State Networks – ESN

Vstup: vektor délky n

→ má vnitřní stav = vektor délky m



⇒ náhodně vygeneruje matici $(m+n) \times m$
kterou už netrénuje

⇒ na začátku náhodný vnitřní stav

Vyhodnocení: ke vstupu přidám vnitřní stav, vynásobím maticí
⇒ dostanu nový vnitřní stav

→ potom ještě MLP vrstva jako u RBF sítí x | matice | MLP → y

⇒ vlastně transformuje vstup délky n na vektor délky m … $m > n$

Trénování: matice se netrénuje

MLP lineární regresí nebo gradient descent

- **Long Short Term Memory Networks – LSTM**

→ místo neuronů LSTM buňka → pamatuje si nějaký stav

→ buňka dostane:

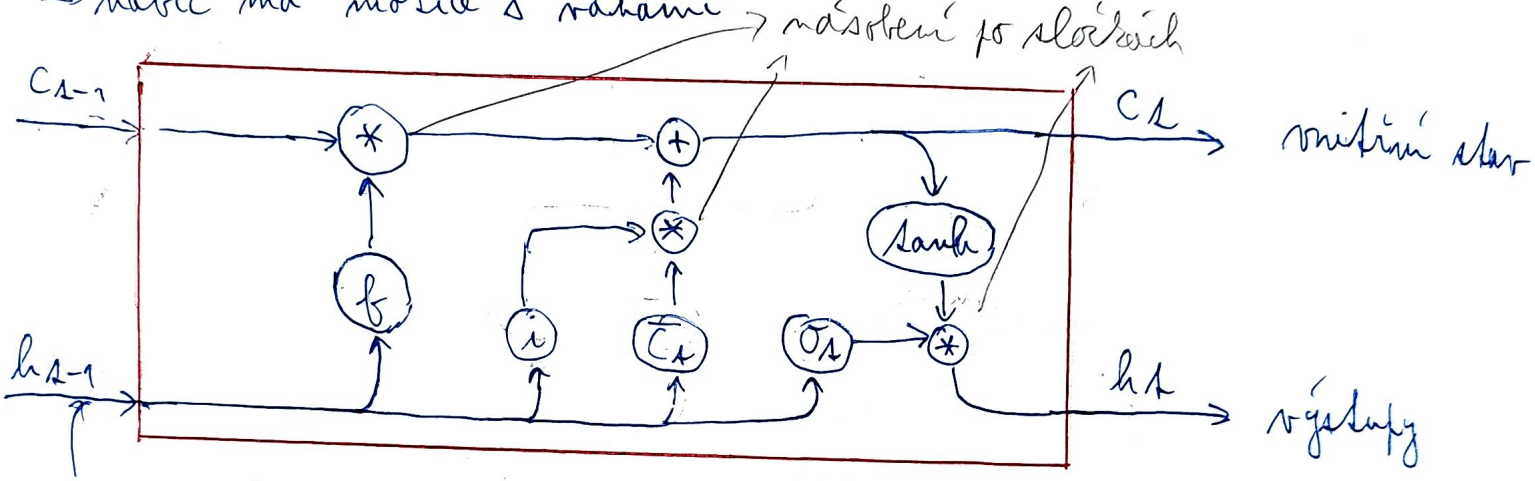
$\vec{c}_{t-1}$ ... předchozí stav

$\vec{h}_{t-1}$ ... rekurentní mezivýsledek z předchozího kroku

$\vec{x}_t$ ... vstupní vektor

↳ zřetězíme tyto vektory za sebe: $[h_{t-1}, x_t]$

→ navíc má motice s vrátky



$$\vec{f}_t = \sigma(W_f \cdot [h_{t-1}, x_t] + \vec{b}_f) \quad \text{... forget}$$

$$\vec{i}_t = \sigma(W_i \cdot [h_{t-1}, x_t] + \vec{b}_i) \quad \text{... input}$$

$$\bar{C}_t = \tanh(W_c [h_{t-1}, x_t] + \vec{b}_c) \quad \text{... kandidát na nový stav}$$

$$\Rightarrow \vec{C}_t = \vec{f}_t \circledast \vec{C}_{t-1} + \vec{i}_t \circledast \bar{C}_t \quad \text{... nový vnitřní stav}$$

$$\vec{o}_t = \sigma(W_o [h_{t-1}, x_t] + b_o) \quad \text{... output}$$

$$\Rightarrow \vec{h}_t = \vec{o}_t \circledast \tanh(\vec{C}_t) \quad \text{... výstup zohlední output a stav}$$

→ W_f, W_i, W_c, W_o jsou matice s parametry

→ b_f, b_i, b_c, b_o jsou bias vektory

→ $\circledast$ je po složkách, obdobně σ (sigmoida) a tanh

→ trénování: rozbalíme v čase a gradient descent

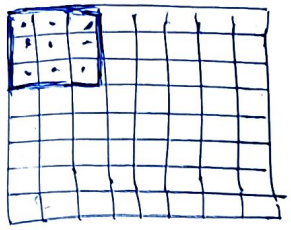
↳ v vnitřního stavu není nikde přímo násobná váha

⇒ není problém s vanishing / exploding gradient

• Konvoluční sítě – zpracování obrazu → aktivace = ReLU

→ konvoluční vrstva → zjednodušení obrázek filtrem ... n×n matice
... bias

↳ 3×3 filtr má jen 3·3+1 parametrů



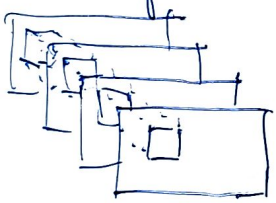
→ z 8×8 obrázku vyrobí 6×6 obrázek

↳ někdy se dělá padding

→ v jedné vrstvě mám víc filtrů
⇒ z toho obrázku vytáhne něco jiného
} ⇒ víc barevných kanálů

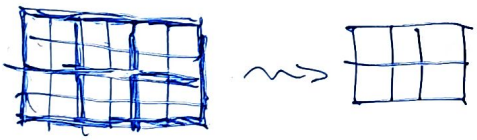
• RGB ... 3 barevné kanály
• Grayscale ... 1 kanál
• Grayscale, který projde conv2D vrstvou s 16 filtry ⇒ 16 kanálů

→ když do conv2D vrstvy jde obraz s k kanály, } k=4 ⇒ 3·3·4+1 parametrů
tak 1 filtr je n×n×k senzor



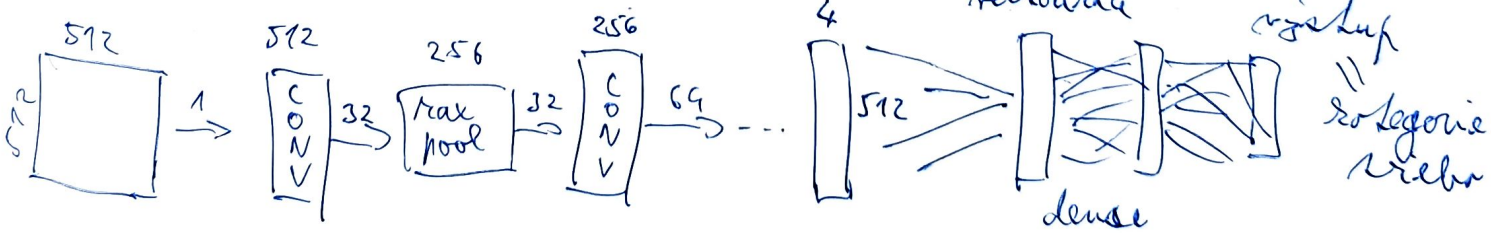
⇒ nakonec zase spočte 1 hodnotu
Relu (vynásob po složkách a sečti)

→ pooling vrstvy – zmenšují rozměry obrázku



Max Pooling = vezme maximum

→ většinou 2×2 filtr, který s jakým se posouvá = stride
se stride = 2



→ postupně zmenšuju dimenzi obrázku a zvětšuju # kanálů
→ nakonec to vektorizuju a zpracování přes dense vrstvy

- **Matoucí vzory – FGSM = Fast Gradient Sign Method**
  - → mám obrázek a síť ho přiřadí do kategorie
  - → spočítám gradient loss fce podle pixelů toho obrázku
    - ↳ tedy pro 256×256 a RGB (3 kanály) je 256·256·3 proměnných
  - → jen udělám znaménko toho gradientu a vynásobím šumem → R=0 v 255, G=0 v 255, B=0 v 255
    - ↳ 3 kanály a 2 znaménka ⇒ 8 barev pro šum
  - → ke každému pixelu přičtu znaménko gradientu v daném pixelu
  - ⇒ obrázek + $\frac{1}{128}$ · šum = matoucí obrázek → maximalizuje loss
    - ↳ malé ε, pro člověka to vypadá stejně

- **Přenos uměleckého stylu**
  - – mám fotku, chci aby byla ve stylu Picassa
  - → ukazuje se, že vnitřní sítě
    - aktivace ve vnitřních vrstvách ~ obsah
    - korelace mezi aktivacemi ~ styl
  - ⇒ optimalizační problém – chci vytvořit obrázek, který
    - má při průchodu sítí aktivace jako ta fotka
    - má korelace mezi aktivacemi jako obrazy s daným stylem

- **Generative Adversarial Networks**
  - → mají dvě sítě
  - Generátor: generuje obrázky, snaží se maximalizovat chybu diskriminátoru
  - Diskriminátor: snaží se rozhodnout, zda obrázek na vstupu patří do trénovací množiny, nebo je od generátoru
    - ⇒ 2 kategorie ⇒ loss = cross-entropy

NEUROEVOLUCE

→ vyráběme / trénujeme neuronky pomocí evolučních alg.

- **Evoluce vah**
  - máme síť s fixní topologií ... vlastně spojitá optimalizace
  - pro učení s učitelem je gradient-descent superior
  - **zpětnovazebné učení**
    - problém je vyhodnocení prostředí (trvá dlouho)
    - → evoluční alg. se snadno paralelizují ⇒ #jedinců = k · #jader
    - **prostředí s řídkými odměnami**
      - odměnu dostanu až na konci, ne průběžně
        - ↳ Q učení se bude trénovat moc dlouho
          - ↳ hodnota akcí se bude propagovat pomalu
      - evolučního algoritmu nevadí, že odměna je na konci
        - ↳ fitness stejně počítá až při vygenerování nové populace

- **NEAT** = **Neuro-evolution of Augmented Topologies**
  - jedinec = neuronová síť s daným počtem vstupů a výstupů
    - ↳ pamatuje si uzly a hrany
      - ↳ (odkud, kam, váha, enabled, inovační no.)
        - ↳ globální ID

- **mutace:**
  1) upravení vah – pomocí evoluce vah
  2) přidání hrany – spojí 2 nespojené uzly
  3) přidání uzlu – rozdrobí nějakou hranu:

  4) disable hrany

- **křížení** – pověsíná inovační čísla

p_1: 1 2 3 4 5

p_2: 1 2 3 4 5 6 7 9 10 → zarovná je pod sebe

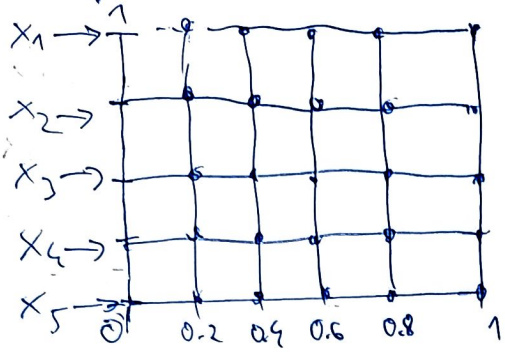
⇒ p_1: 1 2 3 4 5 | 9 ~~⇝~~ 1 2 3 4 5 6 7 9 10 $fit(p_2) > fit(p_1)$

p_2: 1 2 3 4 5 | 6 7 9 10

náhodně vybere z obou → odpovídající kolo s větší fitness

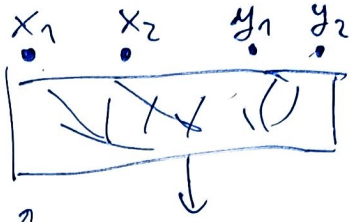
- **chrání inovaci**
  - → když vznikne nová hrana, tak se to nejspíš rozbije → mění fit
  - → jedinci jsou rozděleni do DRUHŮ podle podobnosti
  - → fitness jedince se dělí velikostí jeho druhu
  - ⇒ po křížení a mutacích vybere z druhu nějakého reprezentanta a ostatní přiřadím

- **Hyper NEAT**



→ mám „substrát" neuronů rozmístěných v $[0,1]^2$ ... linspace

→ natrénuji pomocí NEATu síť, která dostane souřadnice dvou bodů a dá váhu



⇒ v 1 iteraci vyrobím hrany s váhami podle mezi všemi dvojicemi uzlů
↳ malé váhy zahodím úplně
⇒ loss této sítě ≈ fitness té NEAT sítě - chci ji minimalizovat

- **Deep NEAT**
  - jednotky nejsou uzly a hrany ale celé vrstvy
    ↳ pomocí evoluce trénuji hyperparametry těch vrstev

- **Co Deep NEAT**
  - pomocí Deep NEATu vyrábím moduly
  - navíc pomocí NEATu vyrábím blueprinty pro sítě
    ⇒ pak do toho blueprintu nastrkám moduly, co sama posbírá
    ↳ tohle udělám několikrát, průměr loss funkcí po natrénování je moje fitness

- **Novelty Search**
  - místo toho, abych evoluci jedince rozhodoval podle fitness, tak posbírám novelty = jak moc nové je to jeho řešení

→ bludiště ... pokud došel na nové místo ⇒ vysoká novelty
→ je dobré střídat novelty a fitness ⇒ explorace x exploitace
↳ novelty mě dostane z lokálního optima
↳ fitness zdokonalí aktuální optimum

PŘÍRODNÍ ALGORITMY

- **Particle Swarm Optimization** – PSO
  - → inspirované pohybem hejna ptáků / ryb
  - částice = 2 vektory $\in \mathbb{R}^n$ … poloha $\vec{x}$ a rychlost $\vec{v}$
    - ↳ navíc si pamatují nejlepší navštívenou pozici $\vec{p_b}$
    - ↳ globálně mám uloženou globální nejlepší pozici $\vec{g_b}$
  - → hodnota pozice určuje nonciální fitness $f: \mathbb{R}^n \to \mathbb{R}$
  - → částice se hýbe v prostoru a je přitahována k těm dobrým místům

$$\vec{v} \leftarrow \omega \cdot \vec{v} + \varphi_p \cdot r_p \cdot (\vec{p_b} - \vec{x}) + \varphi_g \cdot r_g \cdot (\vec{g_b} - \vec{x})$$

- $\omega, \varphi_p, \varphi_g$ … parametry
- r_p, r_g … random $\in (0, 1)$ → může udělat i $\vec{r_p}$ = random $\in (0,1)^n$
  - ↳ pak násobím po složkách

→ směr

$$\vec{x} \mathrel{+}= \vec{v}$$

Topologie – jak spolu částice komunikují

- **globální** – ukládám si globální nejlepší řešení $\vec{g_b}$
- **geometrické** – komunikují spolu částice, co jsou blízko u sebe
- **sociální** – předem je určeno, které částice se kamarádí

→ bodré rychle konverguje, ale nedostane se z lokálního optima

• Ant Colony Optimization – ACO

→ mravenci prozkoumávají prostředí a překládají feromon ← metafora
↳ když najdou jídlo ⇒ hodně feromonu
→ ostatní mravenci bloudí a mají sendenci jít sami, kde je hodně feromonu
→ typicky se používá na hledání cest v grafu – obchodní cestující ↑ Hamiltonovská kružnice

jedna iterace:

1) ∀ mravenec vygeneruje nějaké řešení

1. začne v náhodném vrcholu → rozhoduje se, kam dál
→ pravděpodobnost přechodu $x \to y$ je úměrná

$$(F_{x,y})^{\alpha} \cdot (V_{x,y})^{\beta} \quad \dots \alpha, \beta \text{ jsou konstanty citlivosti}$$

$F_{x,y}$ = feromon na hraně $x-y$
$V_{x,y}$ = volné za hranu $x-y$ → $\frac{1}{\text{délka}(x,y)}$

2) update feromonu

$$U_{x,y} \leftarrow Q \cdot \sum_{k} \frac{1}{L_k} \quad \dots k \text{ je mravenec co prošel přes hranu } x,y$$

L_k = kvalita řešení ... délka nejkratší cesty
Q je konstanta → menší = lepší

$$F \leftarrow (1-S) \cdot F + U$$

↳ vypařování feromonu ... $S = 5\%$

⇒ ∀ mravenec nakreslí feromon, lepší řešení ⇒ víc feromonu

ARTIFITIAL LIFE

- soft → simulace
- hard → roboti
- wet → v laborce ve zkumavce

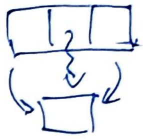
• **Cellular Automata**

- políčko má jednu z k barev a pravidla na změnu barev

Game of life

pravidla: mrtvá & #sousedů = 3 ⇒ obživne
živá & #sousedů ∉ {2,3} ⇒ umře

1D-automaty



→ pokud uvažujeme pouze pravidla s kontextem 1:

$\Rightarrow 2^{2^3} = 2^8 = 256$ možných 1D automatů

→ existuje turing complete automat

• **Langtons ant**

→ černobílý nekonečný grid, na kterém běhá mravenec

- černá ⇒ flip color, turn left, step
- bílá ⇒ flip color, turn righ, step

→ mravenec vypadá, že běhá náhodně

⇒ conjecture: nakonec vždy začne opakovat posloupnost kroků, která vytvoří „dálnici" a mravenec uteče do nekonečna

→ také je turing complete

• **Simulace života** – Tierra

- jedinec ~ program (posloupnost instrukcí) v paměti

⇒ 32 instrukcí – aritmetika, podmínky, skoky, NOP0, NOP1

↳ jump je udělaný pomocí NOPů → hledá vždy complement

Jump	→ hledá
NOP0	NOP1
NOP0	NOP1
NOP1	NOP0
NOP0	NOP1

→ jedinec může číst kód jiných jedinců, ale ne write
→ na začátku jediný jedinec, co se pořád kopíruje
↳ malá šance, že se nějaká instrukce změní
→ je tam smrtka co zabíjí staré jedince
→ vznikli „paraziti", co používají kód jiných jedinců ↗ JUMP