

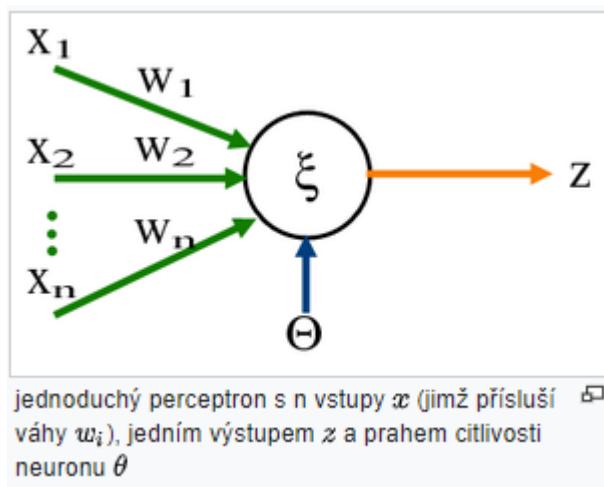
ÚVOD

prezentace 01

AI:

- **AI v každodenním životě** = konkrétní nástroje zacílené na nějaký problém (s obecnou AI nemají nic společného):
 - vyhledávání - Google
 - filtry spamu
 - doporučení - YouTube, Amazon, Facebook
 - hlasové rozpoznávání
 - překlady
 - spell checking
 - asistenti pro řízení automobilů (self-driving cars)
 - rozpoznávání obličejů
- **dva druhy AI:**
 - **a) general/strong AI**
 - artifical general intelligence (AGI) = počítačový systém, který dokáže replikovat kognitivní schopnosti člověka
 - směruje k vytvoření obecné AI, snaha přiblížit se lidské inteligenci a překonat ji
 - jak poznat silnou AI:
 - **Turingův test**
 - hodnotitel (člověk) komunikuje s člověkem a strojem a má poznat, kdo je kdo
 - argument čínským pokojem zpochybňuje Turingův test - vstup: čínské papírky, výstup: odpovědi na ně; princip: jak toho dosáhnout?; myšlenka: knížka s pravidly, tj. odpověď bez porozumění textu → asi blbost, exponenciální složitost
 - **aspekty**: ovládání pohybu, komunikace, uvažování, předvídání, představivost, emoce, intuice, schopnost učení, zadávání cílů, vědomí sám sebe
 - jak jsme daleko k jejímu vytvoření? není šance na to, aby to někdo odhadl, z hlediska HW jsme na tom už teď obecně
 - **b) tools - narrow AI**
 - nástroje pro intelligentní řešení problémů (rychleji než člověkem)
 - mohou být neetické:
 - někteří lidé mohou přijít o práci v důsledku automatizace
 - narušení soukromí
 - důvěryhodnost (generování tváří, hlasů, chatbotů)
 - profilování lidí (př. pro splácení půjček)
 - automatizace zbraní (autonomní tanky, letadla)
 - **historie AI:**
 - dvě větve, které ji ovlivňují:
 - **logika, prohledávání stavového prostoru, diskrétní reprezentace znalostí a plánování v nich, hraní her** (šachy, piškvorky)
 - osamostatnění v roce 1956 - "otcové AI", první intelligentní reálné programy:
 - hraní hry dáma
 - program Theorist pro dokazování teorém
 - program ELIZA - zpracování textu počítačem

- poté útlum, objeveny problémy, které zbrzdily pokrok → omezené možnosti HW (výpočty a paměť), reálné problémy bylo těžké formalizovat, exponenciální prohledávání prostoru
- oživení 70.-80. léta → expertní systémy založené na ručně definovaných pravidlech
 - if [premises] then [conclusion]
 - systémy užitečné, používaly se, ale i tak měly problémy - drahé, pravidla nebyla deterministická
- **inteligence inspirovaná lidským mozkem (neurony), fuzzy intelligence, distribuovaná reprezentace znalostí, strojové učení**
 - kořeny až v roce 1943 - práce neurovědce a logika o propojení jejich oborů, matematický model biologického neuronu
 - 1969 - perceptron (nejjednodušší model dopředné neuronové sítě, sestává z jednoho neuronu) nedokáže vyřešit např. XOR problém; díky tomu se přestaly neuronové sítě používat



- návrat v roce 1986, 1989 vznikla moderní neuronová konvoluční síť pro čtení PSČ v Americe
- velký rozmach neuronových sítí až v roce 2012 (hry Alpha GO, AlphaZero, AlphaStar)
- AI je multidisciplinární obor, souvisí s: ekonomií, psychologií, lingvistikou, neurovědami, statistikou, logikou, filozofií, etikou...
- podobory v rámci AI: strojové učení, neuronové sítě, evoluční algoritmy, počítačové vidění, robotika, zpracování řeči, zpracování přirozeného jazyka, expertní systémy a plánování
- řešení úkolů pomocí AI
 - 1) **modelování**
 - reprezentace problému (model) pro jeho vyřešení
 - 2) **usuzování (inference)**
 - jak získávat odpovědi na otázky
 - př. hledání nejkratší cesty
 - 3) **učení**
 - získávání informací o problému pomocí reálných dat
 - jak na základě dat odhadnout parametry modelu, které já neznám (př. podle doby cest mezi městy odhadnout jejich vzdálenost bez měření)
- strojové učení:
 - automatický převod informací (znalostí) z dat do modelu
 - člověk toto ručně nedokáže
 - dnes hlavní hnací síla AI

- vyžaduje risk/dávku důvěry v podobě generalizace
- agenti a racionalita
 - **agent** = jakýkoliv proces, který bere informace z prostředí (jenž nevidí celé), může se podle nich rozhodovat a těmi rozhodnutími ovlivňuje dané prostředí; př. neuronová síť
 - **racionalita** = hodnocení chování ve snaze maximalizace užitečnosti agenta → agent se snaží o dosažení co nejlepšího ohodnocení

Část I. (doc. Burget)

prezentace 02
Základy strojového učení

Strojové učení:

- **definice:** počítačový program se učí ze zkušenosti E s ohledem na třídu úloh T a metrikou úspěšnosti P, pokud se úspěšnost pro úlohu T, měřenou metrikou P, zlepšuje se zkušeností E
 - vstupní pozorování → model/algoritmus → výstup
 - př. rozpoznávání slov z řečových nahrávek, rozpoznávání identity osob z obrázku obličeje, překlad češtiny do korejštiny, predikce cen akcí ze čtvrtletních reportů...
 - **zkušenosť** - typicky trénovací (data) příklady (kolekce vstupů a odpovídajících výstupů)
 - **měření úspěšnosti** - typicky jak dobře plníme úkol na nové množině evaluačních dat
- **příklady vstupů**
 - křivka řeči - sekvence numerických hodnot (hodnota akustického tlaku na membránu v mikrofonu)
 - 100 x 100 obrázek - 3D matice numerických hodnot (1 dimenze pro každou barvu kanálu - RGB)
 - sekvence slov - sekvence diskrétních symbolů proměnlivé délky
- **vstupy mohou mít:**
 - spojité nebo diskrétní hodnoty
 - fixní nebo s proměnlivou velikostí (vektor nebo matice vs. sekvence symbolů, vektorů, ...)
- **typickým vstupem** bude D-dimenzionální vektor numerických hodnot (nebo skalárů):

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_D \end{bmatrix}$$

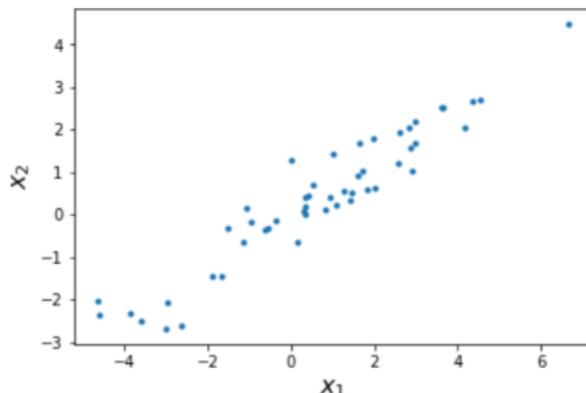
- množina N vstupních pozorování bude tedy reprezentována maticí:

$$\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N] = \begin{bmatrix} x_{11} & x_{21} & \cdots & x_{N1} \\ x_{12} & x_{22} & \cdots & x_{N2} \\ \vdots & \vdots & \ddots & \vdots \\ x_{1D} & x_{2D} & \cdots & x_{ND} \end{bmatrix}$$

- značení: velká tučná písmena matice, malá tučná písmena vektory, malá netučná písmena skaláry
- pro jednoduchost budeme obvykle uvažovat 2D vektory:

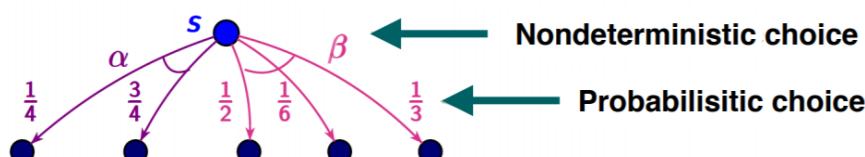
$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

- což nám umožní vizualizovat množinu vstupů jako množinu bodů ve 2D prostoru:



- **strojové učení**

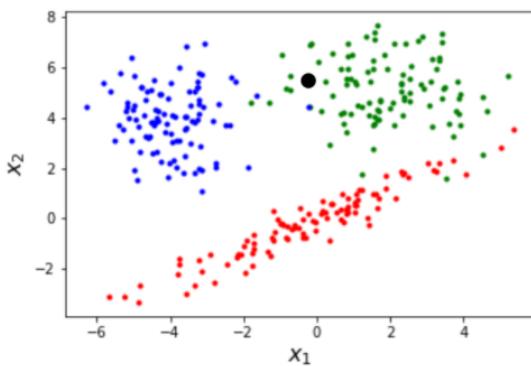
- daná množina trénovacích vzorů (vstupy a/nebo výstupy) se naučí mapovat nové "dosud neviděné" vstupy na požadované výstupy
- hlavní druhy algoritmů strojového učení:
 - **učení s učitelem (supervised learning)**
 - příklady vstupu a odpovídajícího výstupu (tj. učíme se mapovat vstup na výstup)
 - typické úkoly: klasifikace, rozpoznávání vzorů, regrese, ...
 - **učení bez učitele (unsupervised learning)**
 - trénovací příklady jsou pouze neanotovaná vstupní data
 - typické úkoly: clustering, detekce anomalií, odhadování rozložení pravděpodobností dat
 - **semi-supervised učení**
 - něco mezi → pro učení máme k dispozici jak příklady, pro které máme dvojice vstup-výstup, tak i samotná vstupní data
 - **posilované učení (reinforcement learning)**
 - typické úkoly: učení automatického řízení auta, učení hraní počítačových her
 - nemáme vstup a výstup, dokážeme pouze říct, zda jsme v úkolu (ne)úspěšní, úkoly tvoří sekvence spousty kroků
 - zpětná vazba je ve formě pozitivního posílení/negativního oslabení po učinění série rozhodnutí/akcí (př. na konci (ne)úspěšné jízdy/hry)
 - možné řešení pomocí **Markovských rozhodovacích procesů**
 - jedná se o rozšíření Markovských řetězců s diskrétním časem
 - kombinace nedeterministické a pravděpodobnostní volby



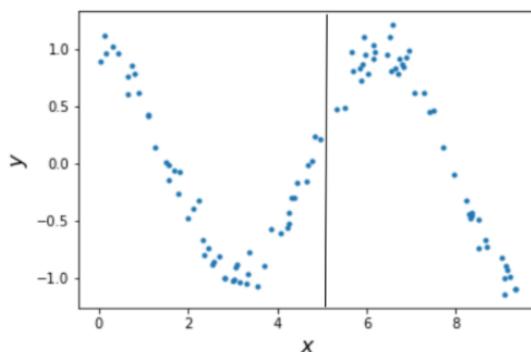
- plánovač (scheduler) - funkce rozhodující nedeterministické volby podle historie

- **učení s učitelem**

- trénovací příklady jsou dvojice vstupů x a výstupů y
- **klasifikace** - přiřazení třídy (red, green nebo blue) novému vstupu, tj, jedná se o diskrétní výstup:



- př.:
 - rozpoznávání identity člověka z obrázku obličeje
 - klasifikace objektu podle velikosti a váhy
 - klasifikace nálady/výrazu obličeje z videa
- **regrese** - vstup je např. jednorozměrná hodnota x , výstupem y je spojité hodnota (ne přesná, ale pravděpodobný rozsah), tj. učíme se nějakou funkci $y = f(x)$:



- př.:
 - predikce ceny akcií podle čtvrtletních reportů obratů firem
 - odhadování počasí, teploty, vlhkosti, pravděpodobnosti srážek z dřívějších meteorologických měření
- další obecnější příklady:
 - rozpoznávání slov z řeči
 - detekce a klasifikace objektů ve videu
 - odhadování postojů každé osoby ve videu
 - strojový překlad z češtiny do korejštiny
 - automatický popis obrázku pomocí anglického textu
 - generování realistických obrázků z anglického popisu

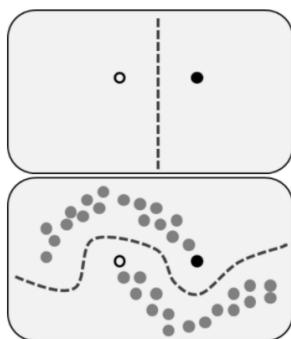
• **učení bez učitele**

- **clustering** - hledání shluků "podobných příkladů" ve vstupních datech
 - v kolekci dokumentů hledání podobných dokumentů (podle stejných témat)
 - v nahrávce konverzace hledáme cluster segmentů, které řekl stejný člověk
- **detekce anomalií** - detekce neobvyklých vstupů (outliers)
 - pro jejich vyřazení z dalšího zpracovávání
 - pro poukázání na ně jako obzvlášť zajímavé
- **odhad hustoty** - učení funkce rozložení hustoty pravděpodobnosti z dat

• **semi-supervised učení**

- neoanotované příklady mohou pomoci najít lepší rozhodovací hranici mezi třídami

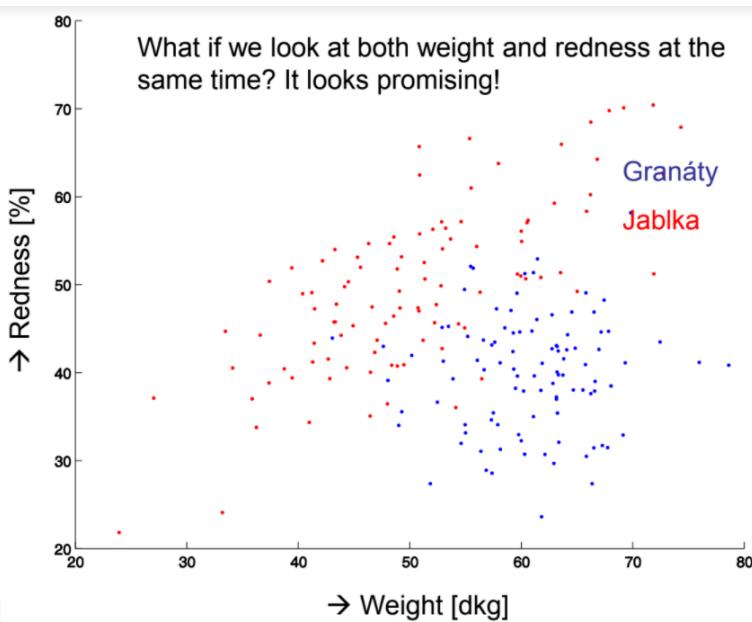
- pointa: pouze dva příklady máme klasifikované (jeden bílý, druhý černý), zbytek šedě → vidíme shluky, díky kterým můžeme určit hranici



- na internetu je k dispozici mnoho neoanotovaných dat → texty, fotky a další obrázky, řečové nahrávky, ...

- **vytvoření klasifikátoru**

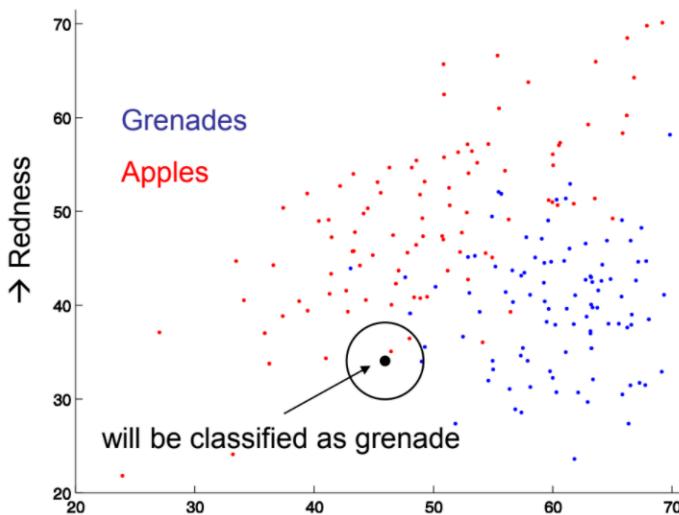
- nalezení rozpoznávacích (diskriminačních) příznaků → měřitelné veličiny - velikost (průměr), barva objektu, váha - podle kterých se budeme rozhodovat
- vícerozměrné příznaky → barva a váha pro klasifikaci jablek a granátů ⇒ slibná klasifikace, oddělené shluky



- **klasifikace**

- cílem je najít dobrou rozhodovací hranici, která oddělí příklady jedné třídy od ostatních
- není vhodné udělat ideální křivku (příliš složitou) pouze podle vzorových dat (tzn. přetrénovaný klasifikátor) → **zobecnění**
- **lineární klasifikátor**
 - rozhodovací hranicí je přímka
 - příliš jednoduchý klasifikátor, pro vzorový příklad má velkou chybovost
 - avšak každý klasifikátor - i ten dobrý - bude dělat vždy nějaké množství chyb
- **kvadratický klasifikátor**
 - rozhodovací hranicí je kuželosečka (parabola, hyperbola, elipsa, ...)
- **klasifikátor K-nejbližších sousedů (K-nearest neighbors classifier)**
 - neparametrický klasifikátor - nemá žádné parametry k trénování nebo rozhodování
 - musí si pamatovat všechna trénovací data

- k přiřazení třídy novému vzoru hledáme nejbližší příklady (pomocí Euklidovské vzdálenosti) z trénovacích dat a zvolíme nejvíce zastoupenou třídu



princip: $K = 3 \rightarrow 3$ nejbližší sousedi, tj. 2 červené tečky, 1 modrá \Rightarrow nový bod je červená třída

- úskalí:
 - jaké vybrat $K \rightarrow 1$ je málo, v prezentacích se jako optimální vyskytuje 9
 - normalizace dat, aby byly veličiny srovnatelné \rightarrow možnosti:
 - v každé ose bude hodnota z rozsahu 0-1
 - změřit si standardní odchylku na každé ose a poté ní podělit veličiny v jedné i druhé ose (podělením standardní odchylkou normalizuji data tak, aby měla stejnou standardní odchylku v každé ose)
- zatím jsme uvažovali možnost tvrdého rozhodnutí (vzor je jablko nebo granát), alternativa - měkké rozhodnutí \rightarrow odstíny šedi představují **odhad** pravděpodobnosti, že klasifikovaný vzor náleží do jedné třídy (pro $K=9$ máme 10 úrovní šedi)
- **modely:**
 - **generativní model**
 - **diskriminativní model**

Generativní model:

- modelujeme rozložení (hustoty pravděpodobnosti nebo příznaků) pozorovaných dat
- př.: 150 pozorování pro 1 diskrétní příznak - váhová kategorie

1	6	12	15	12	2	2	50
4	22	50	14	6	3	1	100
lightest 0.0 - 0.1	lighter 0.1 - 0.2	light 0.2 - 0.3	middle 0.3 - 0.4	heavy 0.4 - 0.5	heavier 0.5 - 0.6	heaviest 0.6 - 0.7	[kg]

- **P (class, observation):** společná pravděpodobnost třídy a pozorování - tj. jedné buňky tabulky - Maximum likelihood odhad (př.: P(granát, těžký) \rightarrow 12/150)
- **P (class):** marginální pravděpodobnost (součet řádku nebo sloupce) - př.: pro nový objekt - P(granát) \rightarrow 50/150)
 - **apriorní pravděpodobnost:** dopředu říkáme, s jakou pravděpodobností bude nový objekt patřit do nějaké třídy; pro tento příklad - $\frac{1}{3}$ granát, $\frac{2}{3}$ jablko
- **P (class|observation):** podmíněná pravděpodobnost (posteriorní pravděpodobnost třídy daná pozorováním; lze i obráceně, kdy pozorování určuje třídu)

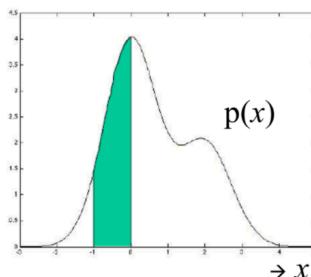
$$P(\text{granát}|těžký) = 12/(12+6)$$

$$P(\text{střední jablko}) = 14/100$$

- základní pravidla pro výpočty pravděpodobností:

- **sčítací pravidlo (operace marginalizace):** $P(x) = \sum_y P(x,y)$
 - pro spojity příznak jako integrál
 - $P(těžký) = P(jablko, těžký) + P(\text{granát}, těžký) = 12/150 + 6/150 = 18/150$
 - $P(\text{granát}) = \sum_x P(\text{granát}, x) = 50/150$
- **násobící pravidlo:** $P(x,y) = P(x|y)P(y) = P(y|x)P(x)$
 - $P(\text{granát}, těžký) = P(\text{granát}|těžký)*P(těžký) = 12/18 * 18/150 = 12/150$
 - $P(\text{granát}, těžký) = P(těžký|\text{granát})*P(\text{granát}) = 12/50 * 50/150 = 12/150$
- **Bayesovo pravidlo:** $P(y|x) = (P(x|y)P(y))/P(x)$
 - $P(\text{granát}|těžký) = (P(těžký|\text{granát})*P(\text{granát}))/P(těžký)$
- spojité proměnné
 - $P(x)$ - pravděpodobnost z $<0,1>$
 - $p(x)$ - rozložení hustoty pravděpodobnosti
 - neptáme se na konkrétní hodnotu pravděpodobnosti, ale na příslušnost do intervalu

$$P(x \in (a, b)) = \int_a^b p(x) dx$$



Sum rule:

$$p(x) = \int p(x, y) dy$$

- generativní model pro spojité pozorování

- osa x - pozorovaná veličina (př.: váha), osa y rozložení hustoty pravděpodobnosti $p(\text{pozorování}|třída)$
- **maximum a-posteriori klasifikátor**
 - vyberme třídu, která je nejpravděpodobnější
 - váha s hodnotou x, podíváme se na průběhy funkcí (*zeleně ve vzorci*), ale musíme zohlednit apriorní pravděpodobnost ($P(\text{class})$)!

$$P(\text{class}|\text{observation}) = \frac{p(\text{observation}|\text{class})P(\text{class})}{p(\text{observation})}$$

$$P(\text{observation}) = \sum_{\text{class}} p(\text{observation}|\text{class})P(\text{class})$$

- jmenovatel $p(\text{observation}) = p(\text{váha}|\text{granát})*P(\text{granát}) + P(\text{váha}|\text{jablko})*P(\text{jablko}) \rightarrow$ zajišťuje normalizaci čitatele, suma je 1
 - prakticky to stejně jako v čitateli, ale pro všechny třídy!
- obvykle nevíme rozložení pravděpodobností, dostáváme pouze vstupní data, z nichž musíme rozložení odhadnout → často vhodné Gaussovské rozložení (tj. z dat potřebujeme spočítat střední hodnotu a standardní odchylku)
- **Gaussovské rozložení**
 - prakticky kvadratická funkce v logaritmické doméně
 - člen před e je normalizační koeficient, aby rozložení integrovalo do 1
 - zbytek exponenciální funkce, v exponentu pouze kvadratická funkce

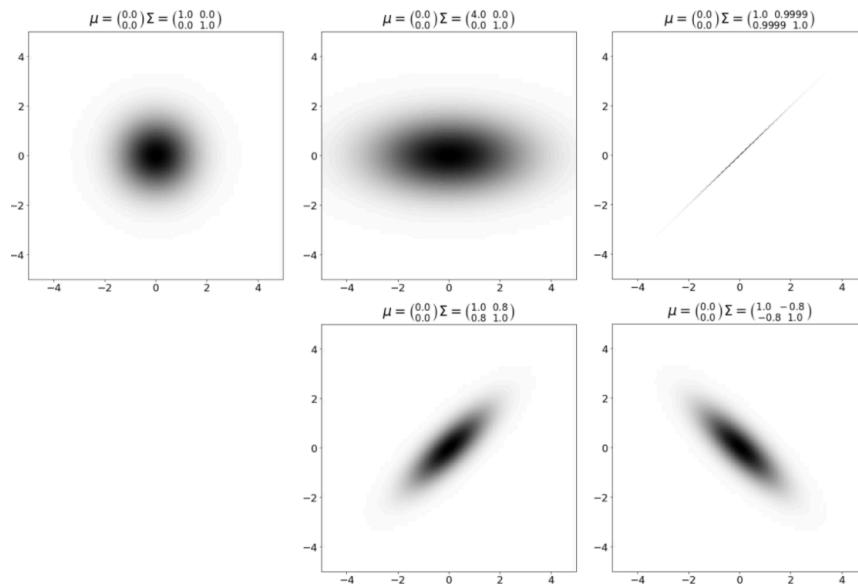
$$p(x) = \mathcal{N}(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

$$\mu = \frac{1}{N} \sum_n x_n$$

$$\sigma^2 = \frac{1}{N} \sum_n (x_n - \mu)^2$$

pozn. střední hodnota určuje, kde je vrchol křivky; standardní odchylka na druhou (**variance**) určuje, jak úzká/široká je křivka

- často se vyskytuje v přírodě
- centrální limitní teorém: součet hodnot mnoha nezávisle generovaných náhodných proměnných dává gaussovské rozložení
 - př. házení dvěma kostkami, nejvíce pravděpodobná 7, nejméně 2 nebo 12
- u vícerozměrného Gaussovského rozložení máme opět střední hodnotu (průměr, ale tentokrát vektorů) a kovarianční matice (vždy symetrická!)
 - interpretace kovarianční matice Σ - co nám říká o Gaussovském rozložení:
 - koeficient na hlavní diagonále říká, jak moc budou hodnoty "roztažené" v dané dimenzi
 - koeficient mimo hlavní diagonálu říká, jak moc jsou hodnoty korelované → pokud hodnota 0,9999, vysoká pravděpodobnost odhadu přesné hodnoty neznámé veličiny; pokud záporný koeficient, dochází k negativní korelacii (pokud jedna hodnota vysoká druhá bude nízká a obráceně)

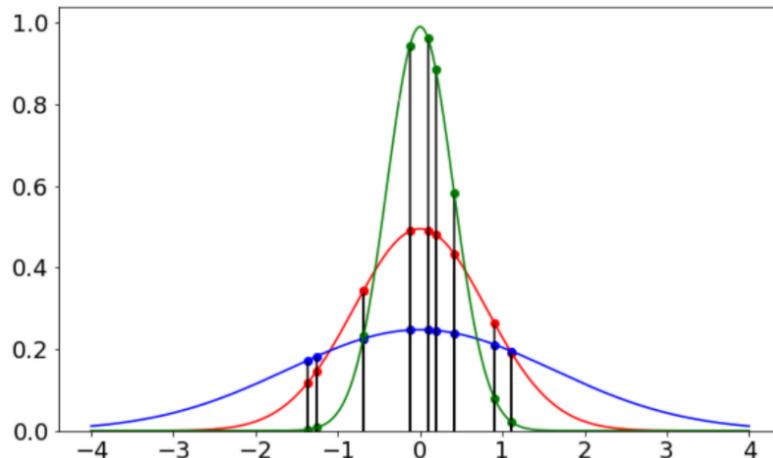


- model směsice vícerozměrných gaussovských rozložení → za použití váhového součtu gaussovských funkcí
- **odhad parametrů pomocí metody maximální věrohodnosti (Maximum likelihood)**
 - mějme parametrické rozložení $p(x|\eta)$ s parametry η (u Gaussovského rozložení je η střední hodnota a variance) a trénovací data (matici s trénovacími vzory - čísla - skaláry, u vícerozměrného rozložení by se jednalo o vektory) $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N]$
 - předpokládáme, že vzory byly vygenerovány ze **statisticky nezávislého gaussovského rozložení**, jehož parametry chceme odhadnout

- můžeme získat odhad parametrů s maximální věrohodností → hledáme takové parametry, pro které $p(\mathbf{x}|\boldsymbol{\eta})$ je maximum; pro každé trénovací dato vyhodnotíme věrohodnost dat

$$\hat{\boldsymbol{\eta}}^{ML} = \arg \max_{\boldsymbol{\eta}} p(\mathbf{X}|\boldsymbol{\eta}) = \arg \max_{\boldsymbol{\eta}} \prod_{n=1}^N p(\mathbf{x}_n|\boldsymbol{\eta})$$

- Pozn.: stříška - odhad, **arg max** - vracíme parametry, pro které funkce nabývá maxima
- pointa: hledáme takovou kombinaci parametrů, aby součin všech funkčních hodnot každého data (tj. výšek) byl ve výsledku co největší



- červená gaussova křivka je nejlepší
- čím víc máme trénovacích dat, tím bude odhad ML bližší rozložení, ze kterého data pocházejí
- **flat prior** - pokud předpokládáme, že všechny hodnoty parametrů mají stejnou pravděpodobnost
- pro optimalizaci je výhodnější pracovat se sumou logaritmů gaussovek (kvadratických funkcí), tj. kvadratickou funkcí → snadnější optimalizace

$$\arg \max_{\mu, \sigma^2} p(\mathbf{x}|\mu, \sigma^2) = \arg \max_{\mu, \sigma^2} \log p(\mathbf{x}|\mu, \sigma^2) = \arg \max_{\mu, \sigma^2} \sum_n \log \mathcal{N}(x_n; \mu, \sigma^2)$$

- maxima jsou tam, kde je nulová derivace
- funkce bude mít jen jedno maximum, neboť je konvexní
- tímto způsobem odvozeny rovnice pro střední hodnotu a varianci Gaussovského rozložení

- **kategorické diskrétní rozložení**

- mějme diskrétní kategorie, které mají nějaké pravděpodobnosti
- speciální případ Bernoulliho rozložení (to má pouze dvě možnosti - př. dvě strany mince)
- hodnotou náhodné veličiny je název nebo index kategorie
- parametrem rozložení je pouze π - vektor pravděpodobností jednotlivých kategorií; $\pi = [\pi_1, \pi_2, \dots, \pi_C]$
- jako vstupní data dostaneme sekvenci pozorování (kategorií) → $\mathbf{x} = [x_1, x_2, \dots, x_N]$, opět můžeme udělat odhad ML

$$P(\mathbf{x}|\boldsymbol{\pi}) = \prod_n \text{Cat}(\mathbf{x}_n|\boldsymbol{\pi}) = \prod_n \pi_{x_n} = \prod_c \pi_c^{m_c}$$

- 3. tvar: pro každé trénovací dato x_n vezmeme pravděpodobnost dané kategorie a pronásobíme to přes všechny kategorie dohromady
- 4.tvar: m_c je počet výskytů kategorie c
- př. $x = [3, 2, 3, 1, 2]$
- $\pi_3 * \pi_2 * \pi_3 * \pi_1 * \pi_2 = \pi_1^{1*} \pi_2^{2*} \pi_3^{2*}$
- pozn.: poslední třídu není třeba počítat, neboť pravděpodobnosti musí sumovat do 1
- optimalizace opět pomocí logaritmu, avšak nyní musíme zohlednit omezující podmínu

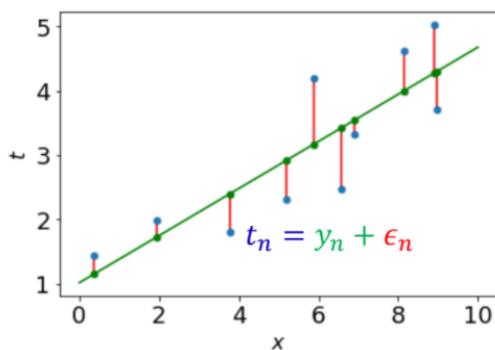
$$\pi_c = \frac{m_c}{\lambda} = \frac{m_c}{N}$$

→ počet dat třídy/celkový počet dat

- nevýhodou generativního modelu (resp. gaussovského klasifikátoru) je to, že odhadujeme model i tam, kde to není potřebné, tj. daleko od rozhodovací hranice → dochází k plýtvání zdroji

Diskriminativní model:

- **lineární regrese**
 - mějme data (modré tečky), která potřebujeme proložit přímkou tak, abychom minimalizovali čtverce vzdáleností trénovacích dat od této přímky
 - cílem je naučit se lineární funkci - $y = f(x) = w_1x + w_0$ (kde w_1 a w_0 jsou parametry, ev. váhy) - z trénovacích dat
 - předpokládáme lineární trend v datech, nad kterými se snažíme učit, ale očekáváme přidání náhodného (gaussovského) šumu do výstupů → $t_n = y_n + \epsilon_n$ (t_n je žádoucí výstup a ϵ_n gaussovská proměnná)



- pro jednoduchost úprava rovnice na skalární součin dvou vektorů: $y = f(x) = w_1x + w_0 = \hat{\mathbf{x}}^T \mathbf{w}$, kde:
 $\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$ and $\hat{\mathbf{x}} = \begin{bmatrix} 1 \\ x \end{bmatrix}$
 - potřebujeme minimalizovat součet čtverců chyb pomocí objektivní funkce:
- $$E(w_0, w_1) = \frac{1}{2} \sum_{n=1}^N (t_n - y_n)^2 = \frac{1}{2} \sum_{n=1}^N (t_n - \hat{\mathbf{x}}_n^T \mathbf{w})^2$$
- řešení pomocí derivace podle \mathbf{w} a hledáme, kde derivace bude nulová (globální minimum)
- $$\begin{aligned} &= (\hat{\mathbf{X}} \hat{\mathbf{X}}^T)^{-1} \hat{\mathbf{X}} \mathbf{t} \\ &= \hat{\mathbf{X}}^\dagger \mathbf{t} \end{aligned}$$
- $\mathbf{w} = \hat{\mathbf{X}}^\dagger \mathbf{t}$, kde křížek znamená pseudo-inverzní, \mathbf{X} je matice všech vstupů a \mathbf{t} je vektor všech žádoucích výstupů
- **gradient** - $\nabla E(\mathbf{w})$

- dvourozměrný vstup, výstupem vektor derivací funkce jednotlivých souřadnic (lze zobrazit jako šipka - ta ukazuje směr nejvyššího růstu; její délka říká, jak je růst strmý - sklon spádu)
 - prakticky: po směru šipek dojdeme k maximu, šipky vedou z minima
- ukazuje směr a rychlosť růstu výstupu
- nulový gradient nastává v místě, kde funkce neroste (minimum, maximum nebo konstantní)
- **učení nelineárních funkcí pomocí lineární regrese**
 - libovolnou nelineární funkci lze reprezentovat jako lineární kombinaci jiných „jednodušších“ nelineárních funkcí (př.: polynomiální funkce)
 - podstata: hledáme lineární kombinace vah
 - mějme vstup a váhy:

$$\hat{\mathbf{x}} = \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^K \end{bmatrix} \text{ and } \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_K \end{bmatrix}$$

- **polynomiální regrese** má tvar:

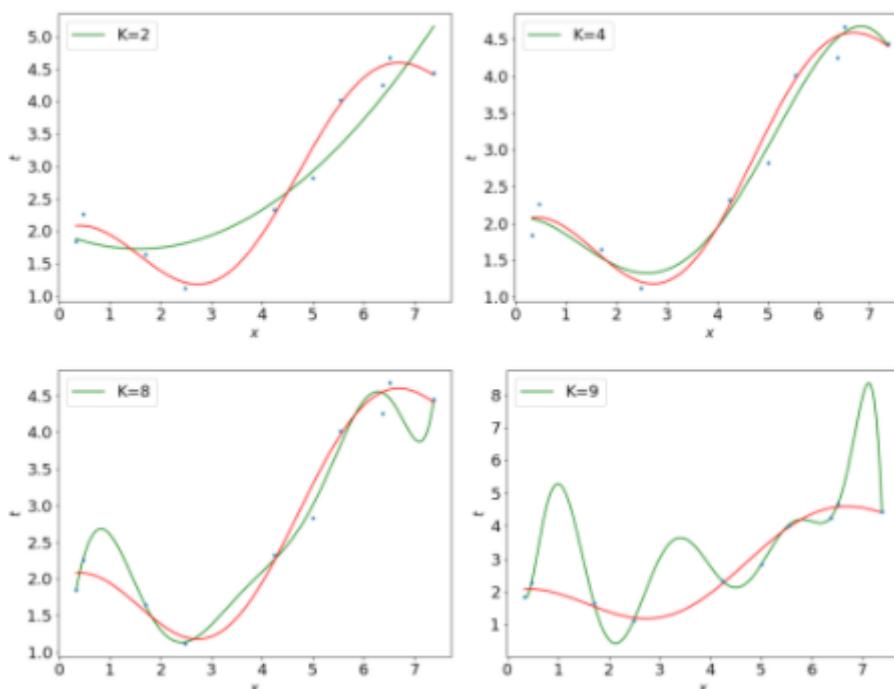
$$y = \hat{\mathbf{x}}^T \mathbf{w} = w_0 + w_1 x + w_2 x^2 + \cdots + w_K x^K$$

(kde K je hodnota nejvyššího řádu polynomu)

- lineární regrese je prakticky speciální případem, kdy K = 1
- K = 0 je konstantní funkce
- řešení analogické s řešením pro lineární funkci
- polynomiální expanze (bázové funkce) jsou jen jednou z možností; můžeme použít jakékoli jiné nelineární báze:

$$\hat{\mathbf{x}} = [f_1(x), f_2(x), \dots, f_K(x)]^T$$

- k dosažení dobré generalizace modelu je potřebné zvolit správnou složitost modelu (tj. hodnotu K) → nemít příliš mnoho parametrů (vah) na natrénování nebo mít větší množství trénovacích dat



- pokud je výstup zkreslen pouze gaussovým šumem, chyba pomocí součtu čtverců má tendenci se učit správnou základní funkci → pokud máme dostatečný počet trénovacích dat nebo dostatečně velké K
- čím je variance (vzdálenost dat od původní křivky) menší, tím rychleji dostaneme s menším počtem dat správné řešení; s dostatečným množstvím dat na varianci nezáleží
- model by měl mít minimálně takovou polynomiální složitost, jakou mají vstupní data (respektive křivka, ze které byla data vygenerována), jinak nebude mít natrénovaný model sílu naučit se, aby odpovídal vstupním datům
- model:

We learn function

$$y = f(x) = \hat{x}^T w$$

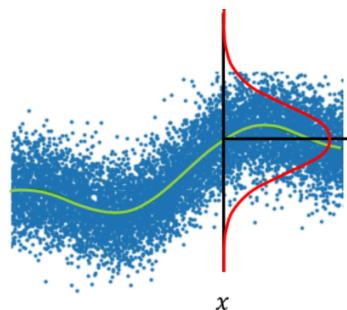
but assume output distributed as

$$t = y + \epsilon$$

where ϵ is Gaussian noise $\mathcal{N}(\epsilon; 0, \sigma^2)$

In other words

$$p(t|x) = \mathcal{N}(t; y, \sigma^2) = \mathcal{N}(t; \hat{x}_n^T w, \sigma^2)$$



→ když vyberu x, y na zelené křivce určí střední hodnotu červené gaussovky, která má **všude stejnou varianci** (standardní odchylku)

- pro odhad pravděpodobnosti s maximální věrohodností parametrů w (a σ^2) - pro trénovací data t_n (z rozložení t) podmíněně závislá na vstupním x_n (x-ových souřadnicích) - potřebujeme maximalizovat funkci *log*:

$$\log p(t|\mathbf{X}) = \sum_{n=1}^N \log p(t_n|x_n) = \sum_{n=1}^N \log \mathcal{N}(t_n; \hat{x}_n^T w, \sigma^2)$$

- po úpravě:

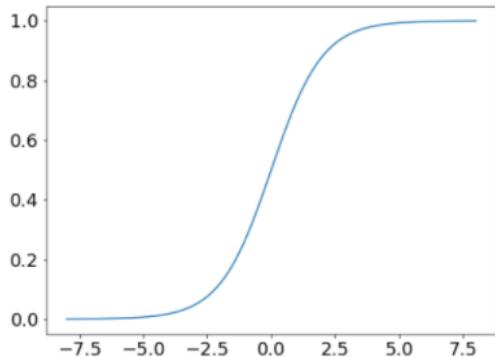
$$= -\frac{N}{2} \log 2\pi\sigma^2 - \frac{1}{2\sigma^2} \sum_{n=1}^N (t_n - \hat{x}_n^T w)^2$$

→ modrá část je klíčová a koresponduje se sumou čtverců

- **logistická regrese** → **pozor, opět klasifikátor!**

- klasifikační metoda, neučíme se libovolné funkce
- učíme se specifickou funkci, která odhaduje pravděpodobnost třídy
- pro jednoduchost uvažujeme binární klasifikátor (třídy pouze 0 a 1), kde pravděpodobnosti třídy musí být v rozsahu 0 až 1 → vstupní dato patří do třídy 0 nebo třídy 1
- diskriminativní klasifikátory
 - takové, které dostanou vstup a určí třídu, ev. pravděpodobnost příslušnosti do jednotlivých tříd
 - princip generativních klasifikátorů, kde **modelujeme rozložení hustoty pravděpodobnosti dat** (gaussovským rozložením) pocházejících z jednotlivých tříd → modelujeme rozložení každé třídy $p(\mathbf{x}|c)$ a apriorní pravděpodobnosti tříd $p(c)$, za použití Bayseova vzorce počítáme $p(c|\mathbf{x})$
 - **problém:** nepotřebujeme podrobně vědět rozložení hustoty pravděpodobnosti jednotlivých tříd, ale pouze kudy vedou rozhodovací hranice
 - přímo se soustředíme na posteriorní pravděpodobnost $p(c|\mathbf{x})$ - funkci, jejíž výstup je v rozmezí 0 a 1 → snažíme se **přímo predikovat třídu** (nemusí predikovat pravděpodobnost, ale logistická regrese to dělá)
 - pracujeme s Bernoulliho rozložením (pro dvě třídy → kategorické rozložení)

- model získaný použitím funkce logistické sigmoidy $\sigma(a)$
 - $\sigma(-\infty) = 0; \sigma(0) = 0,5; \sigma(\infty) = 1$



- předpisy:

$$\sigma(a) = \frac{1}{1 + \exp(-a)}$$

$$P(c = 1|\mathbf{x}) = \sigma(\hat{\mathbf{x}}^T \mathbf{w}) = \sigma(w_1 x_1 + w_2 x_2 + w_0)$$

→ využívá model lineární regrese

- pravděpodobnost třídy:

- $P(c = 1|\mathbf{x}) = \sigma(\hat{\mathbf{x}}^T \mathbf{w})$
- $P(c = 0|\mathbf{x}) = 1 - P(c = 1|\mathbf{x})$

- pro natrénování modelu (parametru \mathbf{w}) jsou opět potřebné vstupy \mathbf{x}_n a požadované výstupy t_n (pravděpodobnost, že dato patří do třídy 1/0)
- jako objektivní funkci můžeme opět zvolit sumu čtverců (sum-of-squares error), ale je lepší použít metodu maximální věrohodnosti (ML)

$$P(\mathbf{t}|\mathbf{X}) = \prod_n P(t_n|\mathbf{x}_n) = \prod_n \sigma(\hat{\mathbf{x}}_n^T \mathbf{w})^{t_n} (1 - \sigma(\hat{\mathbf{x}}_n^T \mathbf{w}))^{(1-t_n)}$$

- kde \mathbf{X} je pozice na x-ové ose, vstupní dato
- \mathbf{t} je vektor výstupů - zda dato patří do jedné nebo druhé třídy
- $\prod_n P(t_n|\mathbf{x}_n)$
- celková věrohodnost dat je produkt věrohodností jednotlivých trénovacích dat, kde věrohodnost jednoho trénovacího data představuje, jaká je pravděpodobnost té správné kategorie (určené hodnotou t_n)
- t_n je přepínač, který na základě kategorie určuje, s jakou částí vzorce se bude počítat
- místo maximalizace věrohodnosti dat je rovnici možné upravit pro minimalizaci chyb - (**binární křížová entropie (cross-entropy)**)

$$E(\mathbf{w}) = -\ln P(\mathbf{t}|\mathbf{X}) = \sum_{n=1}^N t_n \ln \sigma(\hat{\mathbf{x}}_n^T \mathbf{w}) + (1 - t_n) \ln (1 - \sigma(\hat{\mathbf{x}}_n^T \mathbf{w}))$$

→ funkce má pouze jedno globální optimum

- po derivaci přes \mathbf{w} dostávame gradient

$$\nabla E(\mathbf{w}) = \sum_{n=1}^N (\sigma(\hat{\mathbf{x}}_n^T \mathbf{w}) - t_n) \hat{\mathbf{x}}_n$$

- při položení = 0 však rovnice nemá analytické řešení, proto se využívá numerická optimalizace (**gradient descent**) nebo kvadratická optimalizace (lepší, avšak složitější řešení)
 - **gradientní sestup (gradient descent)**
 - opakováně (iterativně) pro dané parametry \mathbf{w} v konkrétním místě spočítáme gradient (vektor, proti kterému se máme pohnout pro zlepšení), který odečteme od současných parametrů \mathbf{w} , čímž uděláme krok a získáme nové parametry → až do dosažení minima objektivní funkce (tj. $\nabla E(\mathbf{w}) = 0$)
- $$\mathbf{w}^{\tau+1} = \mathbf{w}^\tau - \eta \nabla E(\mathbf{w}^\tau)$$
- obvykle čím blíže jsme optimu, tím menší je gradient
 - rychlosť učení η řídí, jak velké kroky provádíme, a je třeba ji vyladit pro dobrou konvergenci

Srovnání:

- generativní:
 - obvykle méně náchylné k přetrenování s malým množstvím tréninkových dat
 - modulární: složité modely vycházejí z jednoduchých
- diskriminativní:
 - dominantní přístup v dnešní době
 - méně plýtvání parametry
 - obvykle lepší výkon s dostatkem dat
 - umožňuje end-to-end řešení → neřeší dílčí kroky, modelování dat, atd.

Sumarizace:

- strojové učení: automatické učení z trénovacích dat za účelem řešení nějakého úkolu
- generalizace: klíčová věc pro SU, aby učení fungovalo i pro nová dosud neviděná data
- učící algoritmy se obvykle snaží optimalizovat nějakou objektivní funkci (ML, suma čtverců, cross-entropy)
- uvažovali jsme pouze jednoduché modely (lineární/logistická regrese) ale stejně objektivní funkce a stejné strategie optimalizace (př. metoda gradientního sestupu) se používají v kontextu silnějších modelů jako jsou neuronové sítě
- máme modely, pomocí nichž se snažíme řešit nějaký problém, ty mají nějaké parametry → snaha optimalizace objektivní funkce s ohledem na tyto parametry, aby model co nejlépe řešil zvolený problém

Část II. (dr. Hradiš)

prezentace 03

Umělá inteligence a strojové učení - neuronové sítě

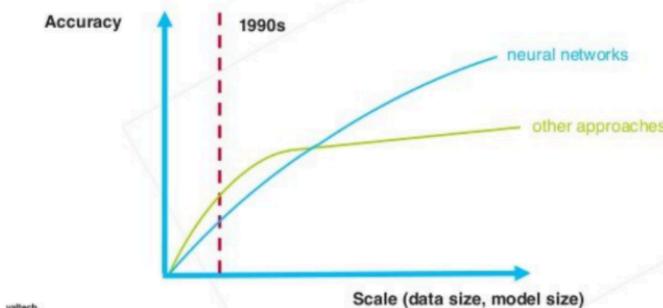
Neuronové sítě

- je to nějaká složitá funkce ("černá krabička", často výpočetně drahá), kterou pomocí datové sady naučíme, aby pro určité vstupy generovala námi požadované výstupy

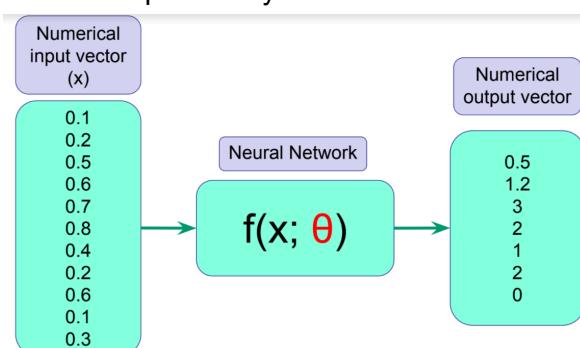


- společné znaky jako **učení s učitelem**, ale sítě umožňují velkou volnost (vstupem může být např. obrázek, video, text, zvuk, ...)
- mohou být vnitřně přizpůsobeny tak, aby dokázaly řešit různé (odlišné) úkoly
- **výhody** proč dominují SU
 - velmi dobře škálují s výpočetními prostředky a daty - pokud potřebujeme zpracovat více dat a máme k dispozici více výpočetních prostředků, můžeme vytvořit větší NS

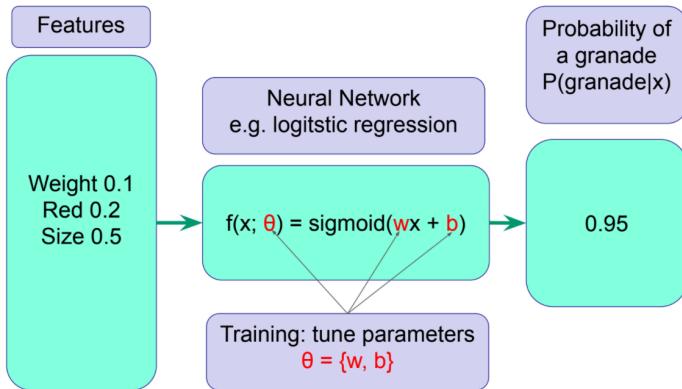
More Data + Bigger Models



- obecný přístup/pohled na to, jak tvořit klasifikátory a další aplikace pro učení s učitelem
- masivní používání → existuje velká báze znalostí a dostupných nástrojů (pytorch, tensorflow, keras, ...)
- **neuronová síť z technického hlediska**
 - funkce, která mapuje unikátní vstup na unikátní výstup (obojí v podobě vektoru reálných čísel)
 - funkcí je právě neuronová síť, kde:
 - x - vstupní vektor
 - Θ - parametry neuronové sítě

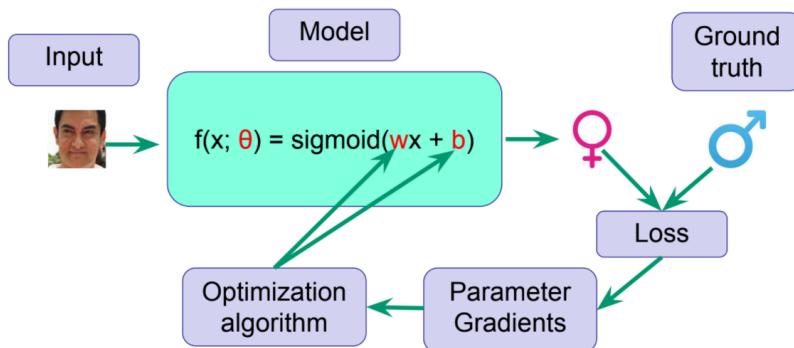


- př. s logistickou regresí, která patří mezi nejjednodušší sítě



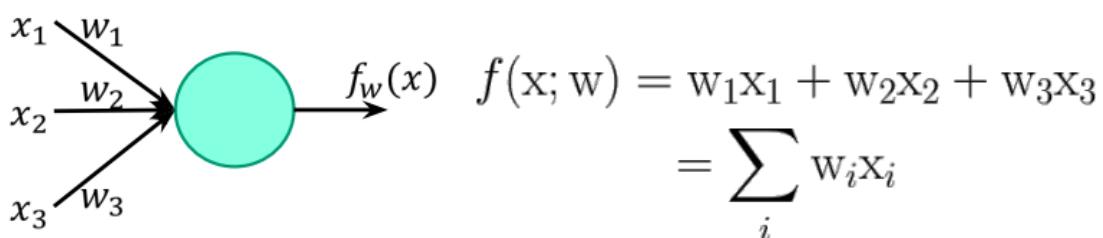
→ vstup vektor vlastností objektu reálného světa reprezentují nějakou třídu; bias - konstanta umožňující posunutí aktivační funkce v souřadném systému

- **trénování** = hledání takových parametrů, které se chovají tak, jak chceme, aby se funkce chovala
 - schéma:

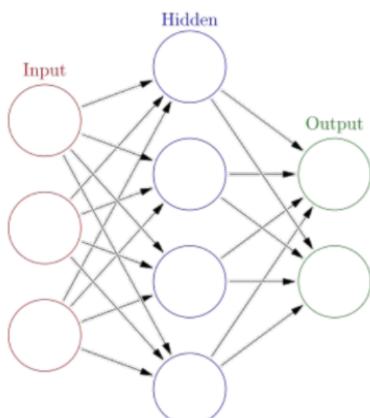


- komponenty učení s učitelem
 - trénovací sada - vstupy a požadované výstupy
 - model s parametry - $f(x; \Theta) = \text{sigmoid}(wx+b)$
 - chybová (loss) funkce - např. křížová entropie
 - trénovací optimalizační algoritmus (objektivní funkce)
- funkce, které se uplatňují v NS:
 - **aktivační funkce**
 - určuje výstupní obor hodnot
 - př. ReLU, sigmoida, soft-max
 - **loss funkce**
 - chyba na 1 vzorku → porovnáváme výstupy NS a target z datasetu
 - **objektivní funkce**
 - chyba na celém datasetu → suma loss funkce
- časté dělení:
 - **“shallow” learning**
 - zahrnuje metody jako je logistická regrese, metoda podpůrných vektorů (SVM), lineární regrese, apod.
 - tradiční přístup, výpočetně efektivní metody
 - dominovaly v 90. letech
 - pro řešení konkrétní úlohy je na začátku potřeba “ručně” vytvořit extrakci příznaků (předzpracování dat a výběr potřebných “informací” ze vstupních dat) → co z dat vyextrahat, aby to bylo užitečné
 - **“deep” learning**

- znalosti o problému ukládáme v rámci NS → v nastavení vah u jednotlivých přechodů mezi neurony
- extrakci příznaků (předzpracování vstupních dat) nedefinujeme ručně, ale použijeme hlubokou NS, která se extrakci příznaků naučí sama (dáme jí víc dat pro trénování)
- složitější a výpočetně náročnější, ale výsledky jsou lepší a není potřeba ručně vymýšlet předzpracování dat
- extrém - end-to-end přístup → jedna neuronová síť (uvnitř může být složitá), která jako vstup bere naše data a vrátí nám očekávaný výstup → neobsahuje mezikroky
- **komponenty neuronové sítě (neurony)**
 - funkce (neuron), která má konkrétní vstup (vektor čísel - x_1, x_2, x_3), parametry (váhy - w_1, w_2, w_3) a "v neuronu" se váhované vstupy sčítají → dostaneme jednu hodnotu na výstupu



- neurony jako vzory → neuron pomocí váženého součtu vstupů zjišťuje, jak dobře vstup odpovídá vzoru (jejich podobnost)
- **lineární funkce**
 - seskládání jednoduchých neuronů (lineárních funkcí) tímto způsobem je neefektivní, neboť můžeme udělat dosazení a opět nám vyjde jednoduchá lineární funkce vstupů (pouze se sloučí váhy)
→ mezinervony tedy nepomohou



- **lineární separabilita**
 - ne vždy je možná - data mohou být rozložena tak, že jedna přímka nestačí pro jejich oddělení
- **nelinearity**
 - do NS se přiřadí operace, které nejsou lineární
 - aplikuje se na každý výstup neuronu samostatně (ne vždy, ale většinou)
 - často se značí sigmoidou → místo lineárního neuronu použijeme nelineární aktivaci funkci a nelineárně namapujeme výsledek váženého součtu

$$f(x; w) = \sigma \left(\sum_i w_i x_i \right)$$

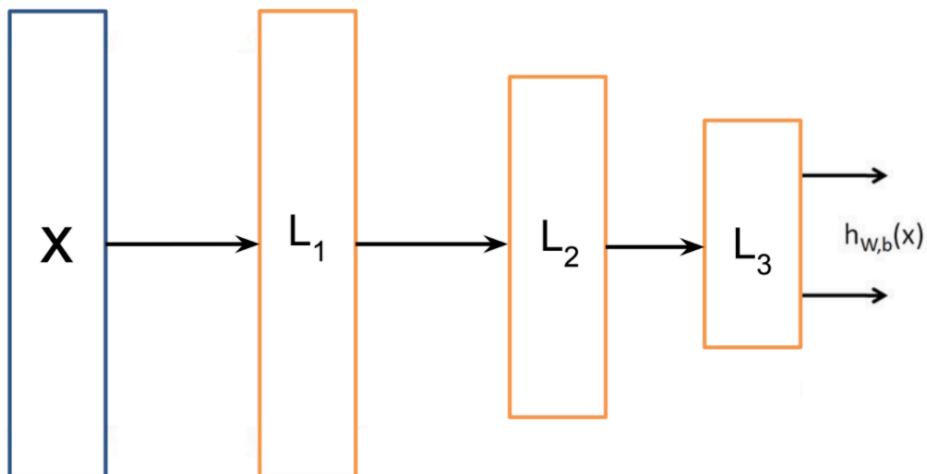
- nejčastější a nejvíce používaná funkce **ReLU** (rectified linear unit) - funkce maximum, která bere maximum mezi svým vstupem (váženým součtem neuronu) a 0
 - pokud vážený součet záporný, výstupem je 0
 - pokud kladný, hodnota se kopíruje na výstup

$$f(x; w) = \max\left(\sum_i w_i x_i, 0\right)$$

- existují i jiné verze ReLU

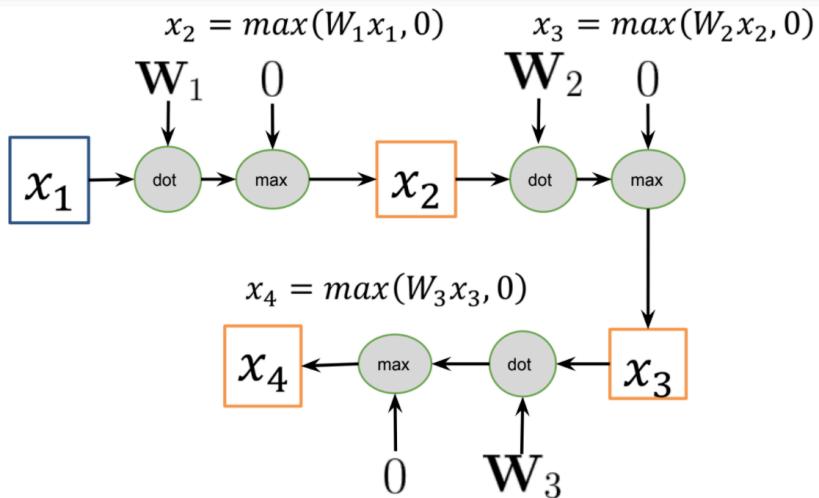
- **od neuronů k vrstvám**

- pomocí neuronů a nelinearity
- neurony se sdružují do vrstev (logické celky se společnými vlastnostmi) → vrstva je homogenní skupina neuronů, které mají stejné vstupy a počítají stejnou operaci
 - neurony ve vrstvě se liší tím, že každý má jiné váhy
 - obvykle se nezobrazují jednotlivé neurony ve vrstvách ani všechna propojení, ale pouze vstupy, "bloky" vrstev, výstupy a jednoduché propojení mezi nimi



- **definice lineární vrstvy jako skupiny neuronů**
 - pomocí lineární algebry → x (vektor vstupů) * w (vektor vah) = jedno číslo → skalární součin matic
 - u více vrstev:
 - $[x] * [w_1] * [w_2] * [w_3] * [w_4] = -1, 0, -1, -1 \rightarrow$ vstupní vektor násobíme s každým vektorem vah; lze realizovat jako jediné maticové násobení - všechny vektory vah sloučíme do jediné matice, tj. $[x] * [w_1 \ w_2 \ w_3 \ w_4] = [-1 \ 0 \ -1 \ -1]$
 - $f(x; W) = Wx$
- NS může být reprezentována jako **výpočetní graf**
 - jedná se o acyklický orientovaný graf

- tvořen operacemi, mezi kterými "tečou" data (aktivace, tenzory)
- v operacích a propojení velká volnost, avšak nesmí obsahovat cykly!



- **chybové funkce (loss functions)**

- se stejným SW a modelem lze pouze změnou chybové funkce trénovat jiný model
- měří, jak výstup NS odpovídá výstupu, který chceme (labelu z datasetu)
- libovolný výběr funkce pro NS, nicméně je třeba použít korektní funkci i aktivační funkci pro zvolený problém
- **binární klasifikace (stejná jako logistická regrese)**
 - cíl: získat $p(c|data)$
 - výstup NS: $p(c=1|data) \leftarrow 1$ hodnota, pravděpodobnost třídy
 - binární problém: $p(c=2|data) = 1 - p(c=1|data)$
 - aktivační funkce: sigmoida (převádí hodnoty od $-\infty$ do ∞ na hodnoty od 0 do 1)
 - chybová funkce:
 - minimalizace pomocí křížové entropie

$$J(f, D) = \sum_{i \in D} -\ln(P(t|x_i)) = \\ \sum_{i \in D} t_i \ln(f(x_i)) + (1 - t_i) \ln(1 - f(x_i))$$

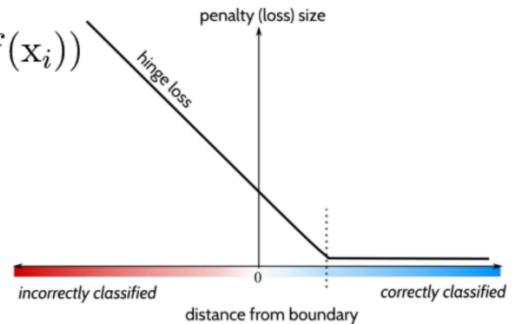
- **binární klasifikace (SVM loss)**

- pointa: hledání mezer mezi dvěma třídami, do nichž se snaží umístit rozhodovací linii, která si drží co nejlepší odstup od všech dat
- cíl:
 - $f(x) > 1$ if $c = 1$ (pokud datový bod patří do třídy 1)
 - $f(x) < -1$ if $c = -1$ (pokud datový bod patří do třídy -1)
- aktivační funkce: žádná, výstup lineární
- chybová funkce:
 - hinge loss ("hokejková chyba")
 - pokud nejsem na správné straně, chyba lineárně stoupá
 - řešíme stejnou úlohu jako v předchozím případě, ale:
 - jinak ji definujeme
 - výstup bude mít jiný význam

- může mít lepší výsledky → může být robustnější proti extrémním datům, robustejnější proti šumu, ...

Loss function: Hinge loss

$$J(f, D) = \sum_{i \in D} \max(0, 1 - c_i f(x_i))$$



- **vícetřídní klasifikace**

- cíl: získat $p(c|data)$ pro počet tříd > 2
- aktivační funkce: soft-max (v podstatě generalizace sigmoidy na více tříd)

$$\sigma_i(\mathbf{z}) = \frac{e^{\mathbf{z}_i}}{\sum_j e^{\mathbf{z}_j}}$$

- e na "konkrétní hodnotu" získanou NS (LOGIT(S)) pro určitou třídu děleno sumou přes všechny takto exponenciované hodnoty

- chybová funkce: vícetřídní křížová entropie

- **multi label klasifikace**

- máme více tříd, které jsou nezávislé (nezávislé rozhodovací problémy) - na každou "otázku" je možné odpovědět ano/ne → př. je na obrázku muž? má černé vlasy? je to dítě?
- jedná se o více binárních klasifikací v jednom
- cíl: získat $p(c|data)$ pro více binárních problémů
- aktivační funkce: sigmoida
- chybová funkce: binární křížová entropie
- výstupem bude vektor výstupních hodnot

- **regrese**

- vstupní hodnoty mapujeme na reálné hodnoty, které mají mezi sebou seřazení a měříme mezi nimi vzdálenosti
- cíl: získat $E[p(hodnota|data)] + gaussovský šum v datech$ (očekávaná hodnota celého rozložení, které mám v datech)
- aktivační funkce: žádná, výstup lineární
- chybová funkce: střední chyba čtverců (mean squared error)

$$\sum_{i \in D} \frac{1}{2} (f(x_i) - t_i)^2$$

- každé dato z datasetu se odečte od výstupu NS a rozdíl umocní na 2
- tato chybová funkce není robustní vůči chybám - odlehlym hodnotám - v datech (jedno dato dostatečně daleko od ostatních dat "rozbije" výstup NS)
- jiná chybová funkce: suma absolutních rozdílů (sum of absolute difference)

$$\sum_{i \in D} |f(x_i) - t_i|$$

- i velká chyba může být eliminována menším počtem "správných" dat, protože se rozdíl neumocňuje na 2 → robustnější funkce
- regrese - s diskretizací
 - k problému přistupujeme pomocí tříd (klasifikace) → díky tomu je vidět, jak si je NS jistá
 - aktivační funkce: softmax s očekáváním/odhadem
 - robustní přístup vůči chybným anotacím
- **vymyšlení vhodné chybové funkce bývá nejnáročnější částí!**
- **gradientní sestup (gradient descent)**
 - optimalizační algoritmus prvního řádu

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial J(D, \theta)}{\partial \theta_j}$$

$$J(D, \theta) = \sum_{D=\{(x_i, y_i), \dots\}} \text{loss}(f(x_i, \theta), y_i)$$

- 1. spočítat parciální derivace objektivní funkce, kterou optimalizujeme, vůči všem parametrům NS → výsledkom je vektor - gradient
- 2. od aktuální hodnoty parametru se odečte parciální derivace objektivní funkce, která je vynásobená učící konstantou α → výsledkem je hodnota, která nahradí starou hodnotu
- algoritmus si z konkrétního místa spočítá, kde to nejvíce stoupá a pohně se opačným směrem až do doby, kdy dojde do lokálního minima
- při NS se nehledá globální minimum, protože by to bylo výpočetně náročné, proto stačí najít lokální minimum → ve výsledku je rozdíl minimální (pokud máme dostatečně velkou NS)
 - velké NS se obvykle učí lépe než malé
 - u logistické regrese máme pouze jedno globální minimum
- gradienty pro trénování jsou počítány pomocí **řetězcového pravidla (chain rule)**
 - automatická diferenciace → aplikování tohoto pravidla, průchod výpočetním grafem od konce na začátek a násobení jednotlivých derivací operací mezi sebou (implementováno v rámci NS)

• učení neuronových sítí

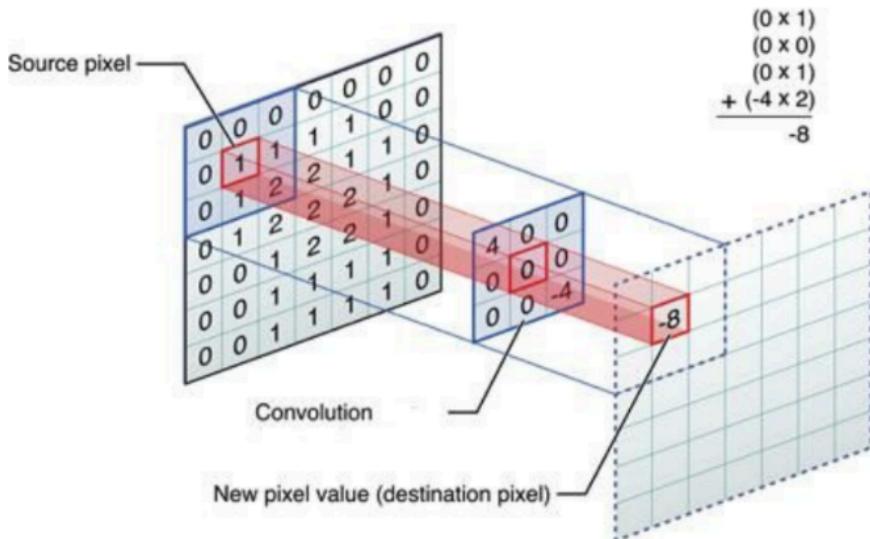
1. dopředný průchod NS (forward pass) - spočtení NS
2. výpočet chyby (compute loss) - hodnota chybové funkce (skalár - 1 číslo)
3. zpětný průchod (backward pass) - výpočet parciálních derivací, chyby vůči parametrům NS
4. použije se pravidlo pro aktualizaci parametrů (např. gradientní sestup)

- **stochastický / mini batch gradientní sestup (stochastic gradient descent - SGD)**
 - velké datasety → příliš mnoho počítání při změnách NS
 - je výhodnější dělat více menších změn, i když v sobě obsahují šum
 - náhodně se vybere malá podmnožina dat z datasetu (mini-batch) a na ní se odhadnou gradienty → na základě nich se udělá update NS (jako kdyby výpočet proběhl nad celou NS)
- **učící konstanta α (learning rate) a adaptivní algoritmy**
 - vysoká učící konstanta → divergence optimalizace
 - nízká učící konstanta → pomalé učení
 - pri SGD závisí učící konstanta od sítě, chybové funkce, inicializace NS, ...

- je třeba zkousit různé učící konstanty nebo použít adaptivní algoritmy, které normalizují učící konstantu (Adam, AdaDelta, ...)
- **rozdělení dat pro trénování dobrého modelu**
 - trénovací sada - na ní se NS bude učit
 - validační sada - vyhodnocení, s jakými "parametry" vrací naše NS nejlepší výsledky (velikost NS, chybová funkce, učící algoritmus, atd.) → výsledkem nejlepší model NS
 - testovací sada - už je vytvořený jeden model, který je nejlepší na trénovacích datech a validační sadě a chceme zjistit, jak dobře bude fungovat "v reálném světě" (na datech, které NS nikdy nevidela)
 - chybová funkce nás ne vždy zajímá, častěji je pro nás důležitější přesnost (accuracy) - zajímá nás, jak často NS udělá chybu, ne jak moc se splete

Konvoluční neuronové sítě (KNS)

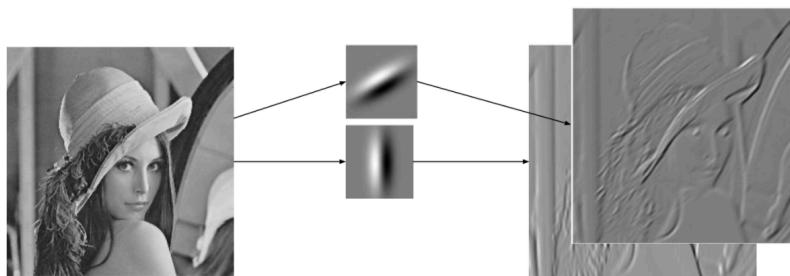
- **NS pro strukturovaná data**
 - ve vektoru mezi čísla není žádná struktura → pokud máme NS postavenou z lineárních vrstev, kde všechny výstupy závisí na všech vstupech a propojení je homogenní, tak NS nijak nerozlišuje mezi jednotlivými prvky ve vstupech
 - př. obrázek, zvuková stopa
 - **zpracování obrázků**
 - 2 hlavní principy:
 - **princip lokality**: pixely vedle sebe mají pravděpodobně něco společného → lokální zpracování
 - **poziční invariance zpracování**: objekty se mohou hýbat → nezáleží na umístění objektů, jejich interpretace bude pořád stejná
 - globální informace je také důležitá, lidský zrak dokáže poměrně snadno v šumu nalézt vzory (př. dalmatin v obrázku s tečkami)
 - ideální matematická operace
 - lokální
 - pozičně invariantní - každou pozici ve vstupu zpracuje stejným způsobem
 - jednoduchá, lineární
 - ⇒ **konvoluce**
- **konvoluce**
 - vstupem KNS pro zpracování obrázku je obrázek - tensor
 - tensor je 3D struktura (matice je 2D) - šířka x výška x RGB kanály
 - konvoluce počítá váženou sumu pixelů vstupního obrázku v malém okolí (lokálně), kde váhy jsou hodnoty konvolučního jádra → počítá se pro všechny pozice v obrázku



- výstupem konvoluce je obrázek stejných rozměrů jako vstupní obrázek

- **konvoluční vrstva**

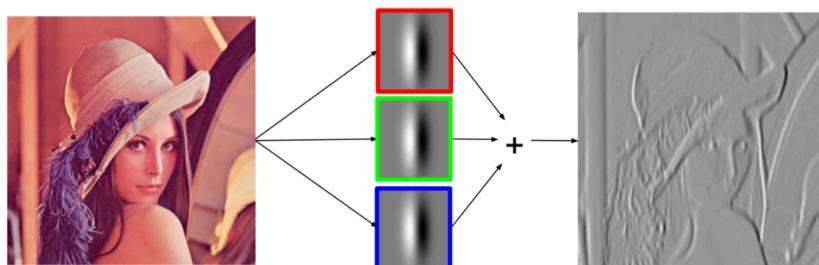
- obsahuje víc filtrů (víc konvolučních jader) a každý z nich vytváří svůj vlastní obrázek (kanál)



C=1

C=2

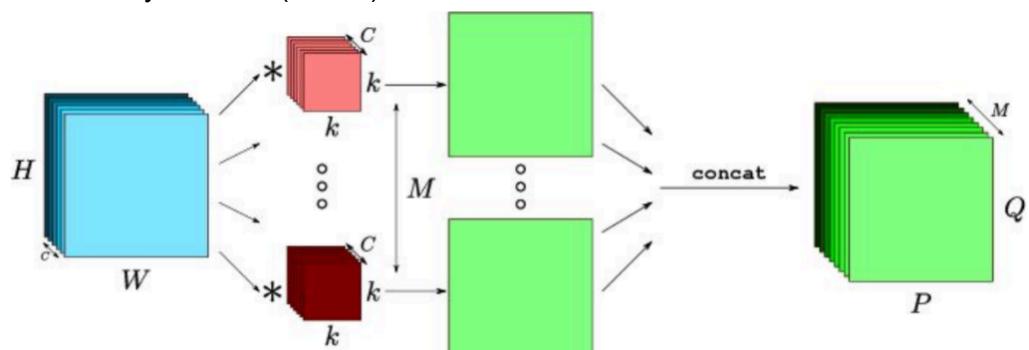
- po získání více kanálů se všechny obrázky sečtou a dostávame jeden výsledný obrázek



C=3

C=1

- vstupem je vícekanálový obrázek, konvoluční vrstva obsahuje několik konvolučních filtrů, výstupem je vícekanálový obrázek (tensor)

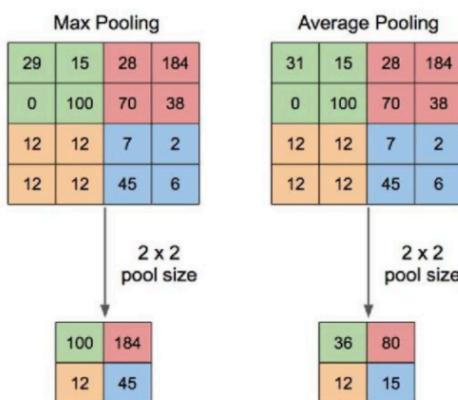


- potřebné chápat:

- konvoluční jádro musí mít stejný počet kanálů jako vstup
- velikost výstupu závisí na velikosti vstupu
- počet kanálů výstupu se rovná počtu filtrů v konvoluční vrstvě
- př. o kolik se zvýší výpočetní náročnost při dvojnásobném zvýšení kanálů vstupního a výstupního obrázku?
 - výpočetní náročnost se zvýší **4x** (2x více kanálů na vstupu, 2x na výstupu $\Rightarrow 2 \times 2 = 4$)
- př. o kolik větší bude KNS (o kolik více místa na disku zabere) při dvojnásobném zvětšení vstupního obrázku ($128 \times 128 \rightarrow 256 \times 256$) a výstupního obrázku?
 - KNS větší nebude, protože konvoluční filtry nezajímá velikost vstupního obrázku

- **pooling vrstvy**

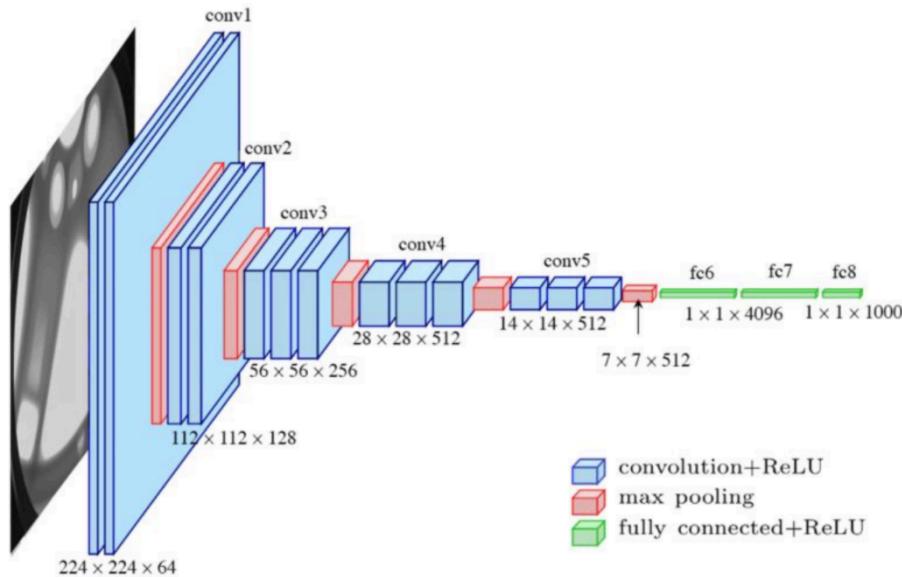
- neobsahují učící parametry
- slouží ke zmenšení obrázku - podvzorkování
- dělí se na redukci podle průměru nebo maxima



- pro jednu malou oblast (např. 2×2) se vypočítá statistická hodnota (Max Pooling nebo Average Pooling) a ta se dá do výstupu jako 1×1 bod

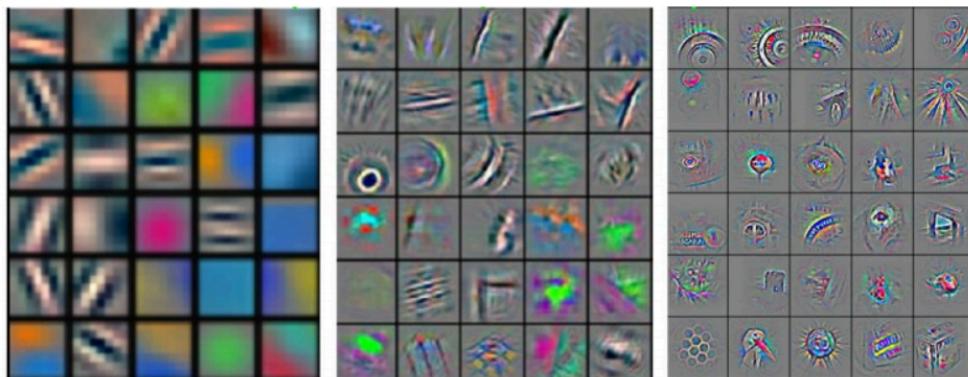
- **architektura sítě (VGG síť)**

- střídání konvolučních vrstev a zmenšování obrázku pomocí max poolingu
- na začátku jsou konvoluční vrstvy ve velkém rozlišení a menším počtu kanálů a postupně se snižuje rozlišení (jeden pixel zachytí více informací ze vstupního obrázku) a zvyšuje se počet kanálů (aby bylo tyto informace možné reprezentovat)
- na konci potřebujeme v 1 pixelu informaci o celém obrázku



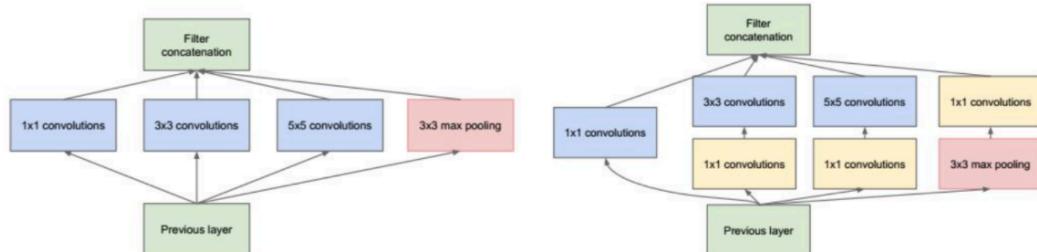
- **hierarchie vlastností (pro konvoluční filtry) → vzory, které detekují jednotlivé filtry v nějaké vrstvě KNS**

- nízká úroveň (př. 1. vrstva) - zachytávají přechody mezi objekty, hrany, výrazné barvy
- střední úroveň (př. 5. vrstva) - zachytávají větší části objektů (po zmenšení obrázku)
- vysoká úroveň (př. 9. vrstva) - zachytávají velké části objektů / celé objekty



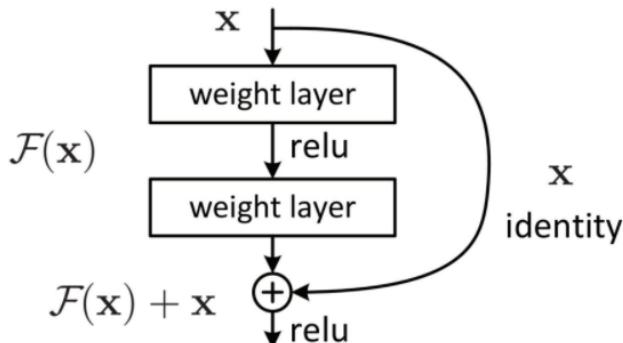
- **architektura GoogLeNet**

- začátek sítě stejný
- poté inception bloky - tvořen více konvolucemi
- snaha o zlepšení výpočetní efektivity



- **reziduální sítě**

- snaha o zlepšení učení → hluboké sítě jsou dobré a dobře fungují, ale od určité hloubky se už nedají trénovat



- každý blok reziduální sítě, který obsahuje dvě konvoluční vrstvy, má navíc jednu "cestu" (boční) ze vstupu bloku k jeho výstupu, kde se sečte výstup z konvolučních vrstev se vstupem
- síť původně (bez této cesty) měnila pomocí konvolučních vrstev signál (x), v reziduálních sítích se učí, co k tomuto signálu přičíst, aby měl námi požadované vlastnosti
- původně v konv. vrstvách vzniká multiplikativní šum, v těchto blocích vzniká "pouze" aditivní šum

- **generalizace (zlepšení výsledků NS):**

- **dobrá struktura modelu** - musí odpovídat tomu, co chceme spočítat a síť musí mít v sobě výpočetní prvky, které dokáží vyřešit daný problém (klasifikační síť budou mít jinou strukturu než síť pro odhad vzdáleností dvou snímků)
- **multi-task učení** → je lepší vytvořit jednu větší síť, která bude řešit 10 problémů (které mají nějaké prvky společné) než 10 malých sítí, jelikož informace při řešení jednoho problému mohou pomoci při řešení jiného problému
- **více dat** → čím je k dispozici více dat, tým lepší jsou výsledky
 - víc dat je možné získat pomocí **augmentace dat** (data augmentation) - vytvoření nových dat z původních dat pomocí transformací (obrázky z datové sady můžeme otočit, posunout, zvětšit, změnit, změnit barvu, ...) → nově vzniklé obrázky jsou podobné, ale ne stejné
- **velikost modelu**
- **finetuning** → použití již natrénované sítě, která se jen částečně upraví - např. se vymění 2 poslední konvoluční vrstvy - v krátkém čase a s malým počtem dat síť dotrénuji a použiji na řešení jiné úlohy
- **regularizace** a overfitting
 - overfitting znamená, že síť se naučí rozpoznávat pouze na daných (trénovacích) datech a pro jiné dátá bude nepoužitelná (testovací data) → nenastává tak často
 - trénování je možné zastavit před tím, než se síť přetrénuje (v bodě, kdy NS vrací nejlepší výsledky) → pokud se výsledky na validační sade nezlepšují, trénování zastavím
 - **regularizace vah** - omezení, že v síti nechceme velké váhy (síť se nemůže naučit, co by sama chtěla)
 - pomocí přidání **členu** do objektivní funkce → čím větší budou váhy a čím dál budou od 0, tím víc mi to vadí
 - **regularizace aktivací** - **omezení** toho, jak moc si je aktivační funkce jistá

$$J(D, \theta) = \left(\sum_D loss(f_{\theta}(x_i), y_i) \right) + \lambda \|\theta\|^2 + \beta \|activation\|^2$$

- využití **dropout** - náhodně se některé aktivace nastavují na 0
- **batch normalizace**

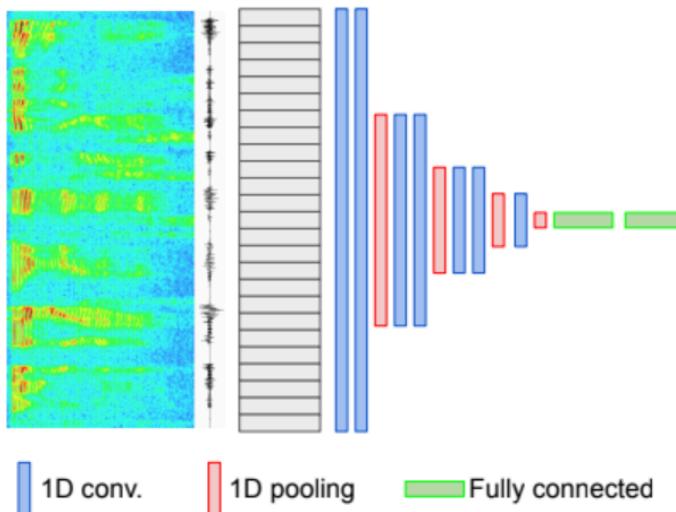
- v podstatě vrstva NS, která má dvě vlastnosti - obecně se chová předvídatelněji (lépe se trénuje) a zároveň trochu regularizuje síť
- někde v síti (na některých aktivacích) jsou zajištěny rozumné hodnoty - přes mini-batch pro každý kanál spočítá mean (průměrnou hodnotu) a standardní odchylku \Rightarrow normalizace hodnot (tj. od každé hodnoty odečtení mean a dělení st. odchylkou)
- obvykle využívá ještě dva učící parametry γ , β

prezentace 04 Umělá inteligence a strojové učení - sekvence a jazyk

Rekurentní neuronové sítě

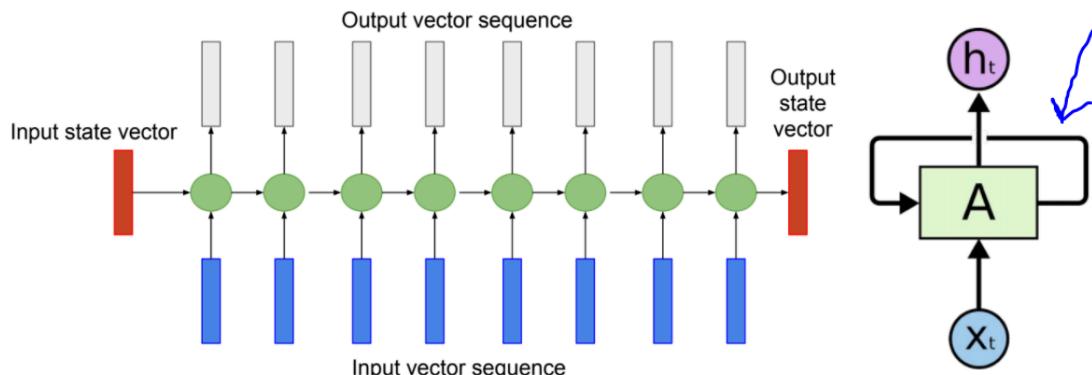
- **rozpoznání zvuku**

- podobné rozpoznávání obrazu (ty však 2D) - vstup je pouze 1D (1D konvoluční vrstva, 1D pooling vrstva, ...)



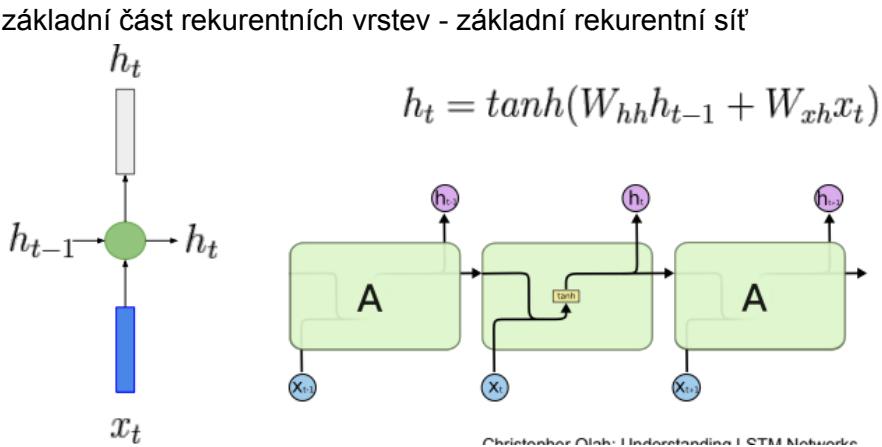
- **rekurentní vrstvy**

- rozdíl: neuronové sítě nemohou mít tento “**cyklus**”! Ve skutečnosti se nejedná o cyklus, ale o zhuštěné zobrazení (obrázku vlevo); A je v síti několikrát \rightarrow (vpravo) zabalené zobrazení výpočetního grafu velké sítě, kde A je všude stejná operace, která se několikrát opakuje

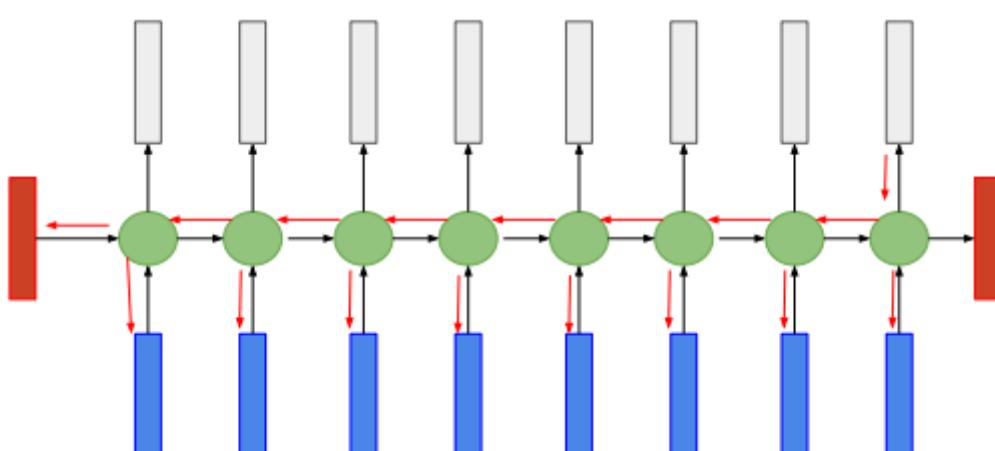


- několikrát se opakující operace A (zelené kolečko), vstupní (inicializovaný hidden state h_0) a výstupní stav (h_{END}), sekvence vstupních vektorů (např. zvukový signál) a sekvence výstupních vektorů
- rekurentní vrstva bude mít 2 vstupní vektory (vektor vstupního stavu a sekvence vstupních vektorů) a 2 výstupní vektory

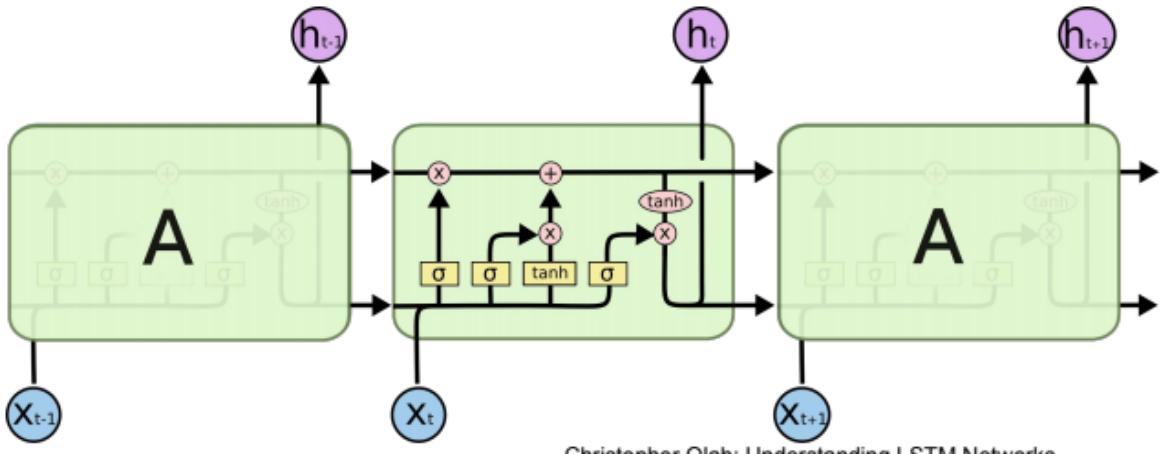
- jejich výhoda spočívá v možnosti zpracování různě dlouhých sekvencí a zapamatování si dlouhodobých závislostí
- jejich výstup se v závislosti od problému může konkatenovat mezi jednotlivými výstupy, nebo vzít jen poslední výstup a pod.
- **vanila RNN - operace A**
 - základní část rekurentních vrstev - základní rekurentní síť



- x_t aktuální vstup, h_{t-1} předchozí vnitřní stav, h_t výstup
- W_{hh} - matice s rozměry $h * h$
- W_{xh} - matice s rozměry $x * h$
- \tanh - nelinearity (hyperbolický tangens)
- **trénování rekurentních vrstev**
 - dopředný průchod (černé šipky) - výpočet sítě
 - zpětný průchod (červené šipky) - plynutí gradientů při počítání derivace sítě



- derivace získána z jednoho výstupu zpětně ovlivní **všechny** části vrstvy (kromě ostatních šedých výstupů)
 - update modelu se dělá na základě sumy všech gradientů (z každého zeleného kroužku)
- **LSTM - Long Short Term Memory**

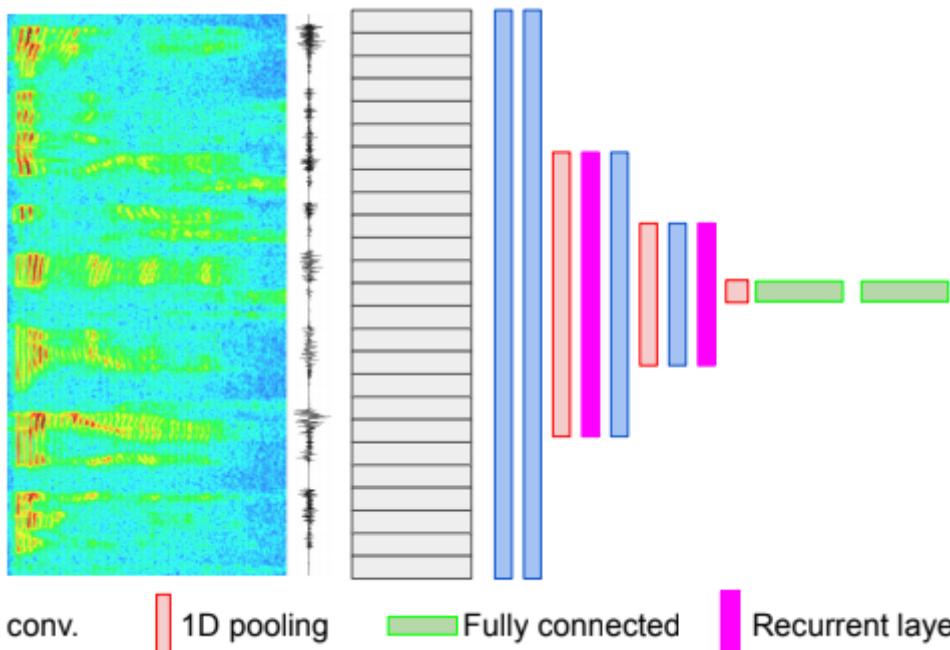


- slouží na zachování gradientů
- obsahuje sběrnici (procházející vrstvou), na které probíhá násobení (hodnoty na sběrnici se násobí s hodnotami od 0 do 1; pokud hodnota signálu 0, hodnota na sběrnici se úplně zapomene) a sčítání (k hodnotám na sběrnici se přičítají nějaké hodnoty)
- výhoda oproti vanila: gradienty se nemíchají, proto se mohou dostat na větší vzdálenost
- **GRU - Gated Recurrent Unit**
 - podobné jako LSTM, pouze jednodušší architektura

- **obousměrné rekurentní vrstvy (Bidirectional recurrent layer)**

- sekvence se zpracovává jak jedním, tak i druhým směrem
- nejdřív jedna vrstva zpracuje sekvenci v jednom směru, pak se sekvence otočí a ta stejná vrstva ji znova zpracuje (již otočenou); pak se sekvence otočí zpět
- kanály z jednoho směru se obvykle konkatenují s kanály z druhého směru

- **kombinace konvolučních a rekurentních vrstev (pro zpracování zvuku)**



- začátek se dělá pomocí konvoluční vrstvy a když vstup již máme v menším rozlišení, použije se rekurentní vrstva
- rekurentní vrstvy se umísťují na místo vrstev konvolučních

Zpracování textu

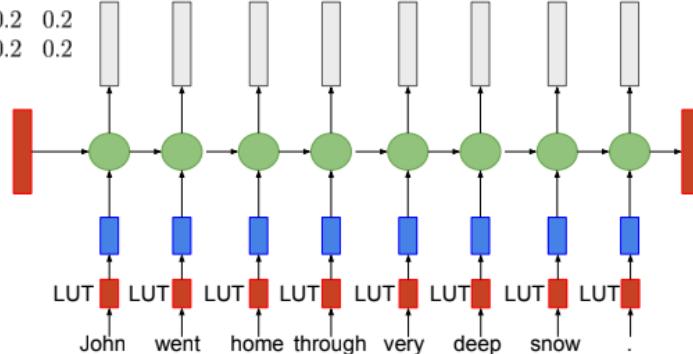
- text je potřeba převést na vektor reálných čísel (pro násobení s maticemi)
- **one-hot-encoding**
 - každé slovo má unikátní reprezentaci (vektor)
 - jednoduchý koncept, ale neefektivní, protože při velkém počtu slov budou matice vektorů příliš velké a řídké
 - navíc nelze určit podobnost slov - tato reprezentace nezachycuje jejich sémantiku (vektory jsou vzájemně ortogonální, po vynásobení dvou vektorů je výsledek 0)

$$\begin{array}{c}
 x_t \\
 \begin{matrix} 0 \\ 0 \\ John \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{matrix} \\
 Wx_t \\
 \begin{bmatrix} 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 \\ 0.3 & 0.2 & 1.2 & 0.2 & 1.0 & 0.0 & 0.2 & 0.2 \\ 0.8 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 \\ 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 \\ 0.3 & 0.2 & 1.2 & 0.2 & 1.0 & 0.0 & 0.2 & 0.2 \\ 0.8 & 0.2 & 0.1 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 \end{bmatrix} \\
 \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.2 \\ 1.2 \\ 0.2 \\ 0.2 \\ 1.2 \\ 0.1 \end{bmatrix}
 \end{array}$$

- **word embedding - LUT**

- realizovaná jako lookup tabulka (matice jako v případě one-hot-encoding)
- když na vstup přijde index konkrétního řádku (jedno slovo), vezme se příslušný vektor a nic se již nemusí násobit

snow	0.2	0.2	0.2	0.2	0.2	0.2	0.2
went	0.2	1.2	0.2	1.0	0.0	0.2	0.2
john	0.2	0.2	0.2	0.2	0.2	0.2	0.2
deep	0.2	0.2	0.2	0.2	0.2	0.2	0.2
very	0.2	1.2	0.2	1.0	0.0	0.2	0.2
home	0.2	0.1	0.2	0.2	0.2	0.2	0.2



- tyto tabulky se učí a pokud jsou vektory natrénovány správně, je možné mezi vektory slov počítat euklidovskou vzdálenost, která říká, že vektory slov, které jsou si něčím podobné (blízké), budou mít menší vzdálenosti (např. monarcha - král vs. monarcha - řidič)
- **jak získat word embeddings**
 - hlavní myšlenka: podobná slova se objevují v podobných kontextech (pokud v úplně stejných, mohou to být synonyma)

█ : Center Word
█ : Context Word

c=0 The cute cat jumps over the lazy dog.

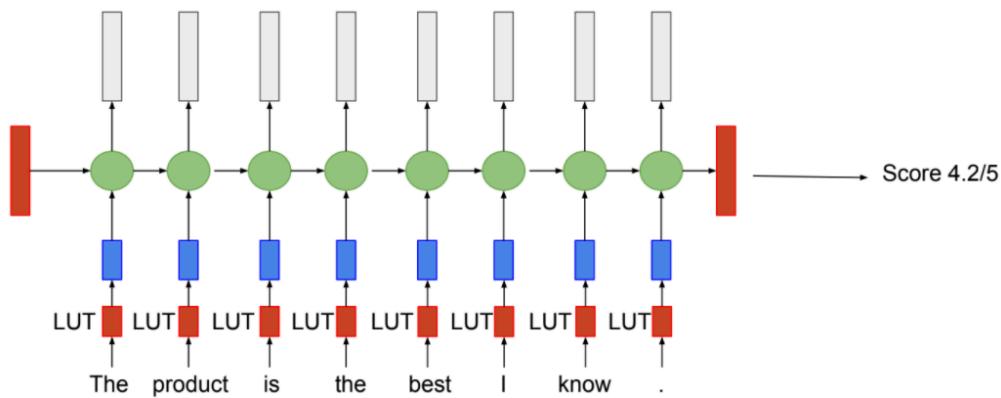
c=1 The cute cat jumps over the lazy dog.

c=2 The cute cat jumps over the lazy dog.

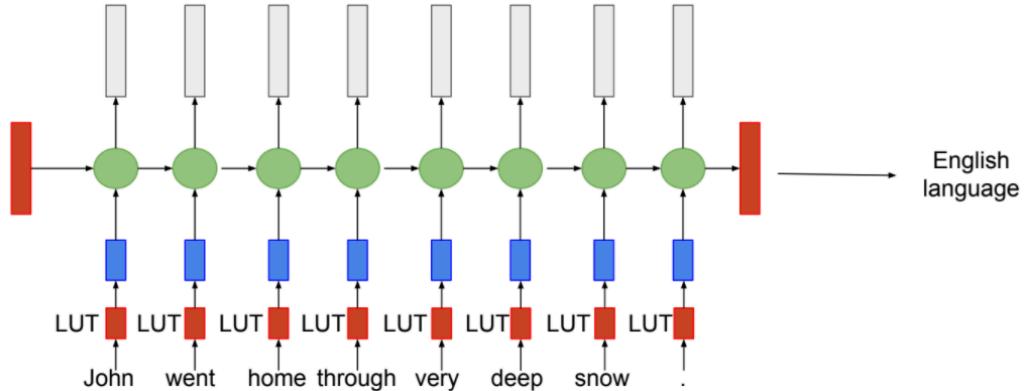
- jedno slovo se vybere jako "hlavní" a k němu se přiřadí slova z okolí
- Word2Vec (Mikolov)
- **fastText**
 - knihovna pro efektivní klasifikaci a reprezentaci textu využívající *vector math*
 - word embedding ke stažení → obsahuje slovní vektory pro 157 jazyků
 - zarovnané vektory pro 44 jazyků (př. auto a car na podobném místě) → užitečné pro překlady

NS pro zpracování textu/sekvencí

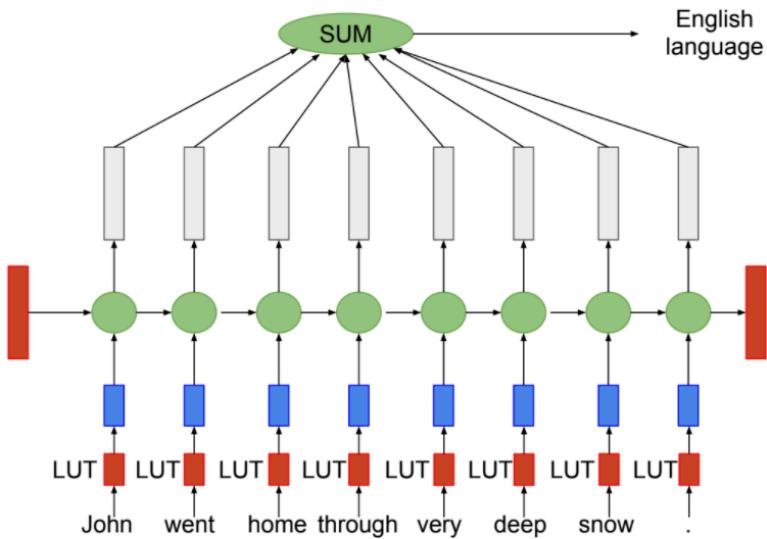
- **regrese s rekurentními NS**
 - text je pomocí NS ohodnocen na nějaké skóre (např. recenze na internetu → z textu odhadujeme počet hvězdiček)



- neideální, ale použitelná, architektura
- **klasifikace**
 - pro vstupní řetězec NS zjistí, v jakém jazyce je napsaný



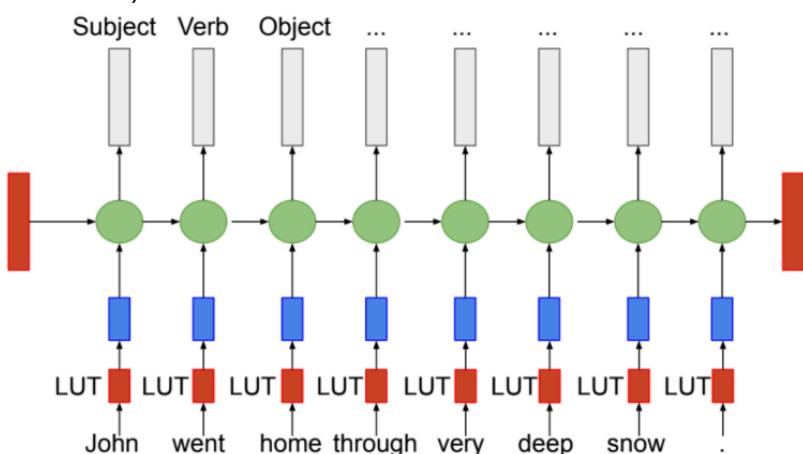
- neideální architektura - pokud je na vstupu dlouhá věta, informace ze začátku věty musí projít všemi vrstvami hluboké neuronové sítě - je složité NS natrénovat, aby informaci zachovala
 - na čím větší vzdálenost se má informace přenést, tím hůř se to učí (i LSTM má své limity)
- **klasifikace - jiná architektura**



- nevyužije se koncový stav, ale vezme se každý skrytý stav produkovaný v každém kroku po zpracování každého slova, udělá se jejich průměr, vytvoří se vektor a na základě něho se rozhodne jazyk vstupní věty
- pravděpodobně lepší síť než ta předchozí

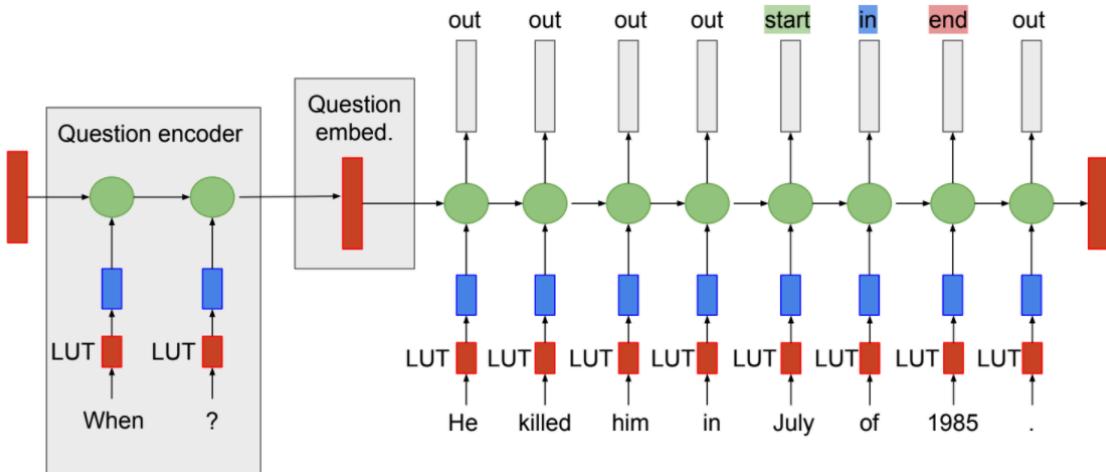
- **tagování slov (word tagging)**

- prakticky sémantická segmentace
- př. rozbor vět na větné členy → pro každé slovo určit, o jaký člen se jedná (ev. jinou informaci - např. slovní druh)



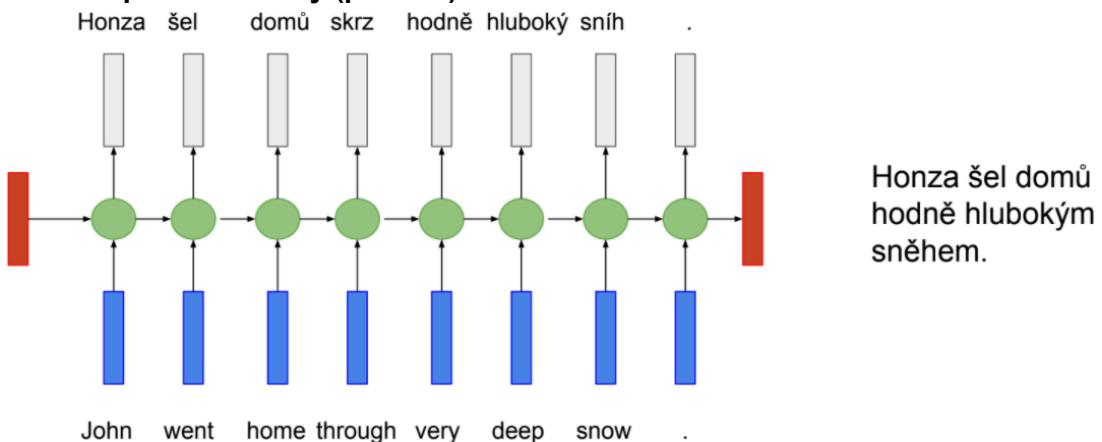
- **porozumění textu (reading comprehension)**

- text obsahující nějaké informace
- na vstupu je otázka a úlohou NS je najít odpověď na otázku v textu (extraktivní odpovídání)



- 1 (rekurentní) síť na začátku zpracuje otázku
- vytvoří vektor reprezentující otázku
- další síť má na vstupu odstavec textu, ze kterého má vybrat část textu jako odpověď
- pro každé slovo má síť říct jednu ze 4 možností - out (není součástí odpovědi na otázku), start (na tomto místě odpověď na otázku začíná), in (pokračuje), end (končí)

- **sequence to sequence modely (překlad)**



- není možné vzít slovo ze vstupu, "přeložit" ho a vrátit ho na výstup již přeložené, protože ne každé slovo v překladu se mapuje 1:1
- při překladu např. do němčiny jdou slovesa na konec věty, což tato síť neumožňuje
- pro překlad sekvence do jiné sekvence je potřebné použít jiný přístup → **autoregresivní faktorizace (autoregressive factorization)**
 - pravděpodobnost všech možných vět
 - pravděpodobnost celé věty je složitá, podmíněná pravděpodobností není řešení
 - modelování jazyka → jak pravděpodobnostní je daná věta v daném jazyce, tj. po sekvence slov
 - faktorizace - rozložení na pravděpodobnosti (pst. 1. slova, pst. 2. slova na základě 1. slova, pst. 3. slova na základě 1. a 2. slova, atd.) → pomocí rekurentní sítě
 - model predikující pravděpodobnost jednoho slova pro danou historii (prefix) - slova ve větě před námi "testovaným" slovem

$P(\text{sentence} \mid \text{John went home through very deep snow})$

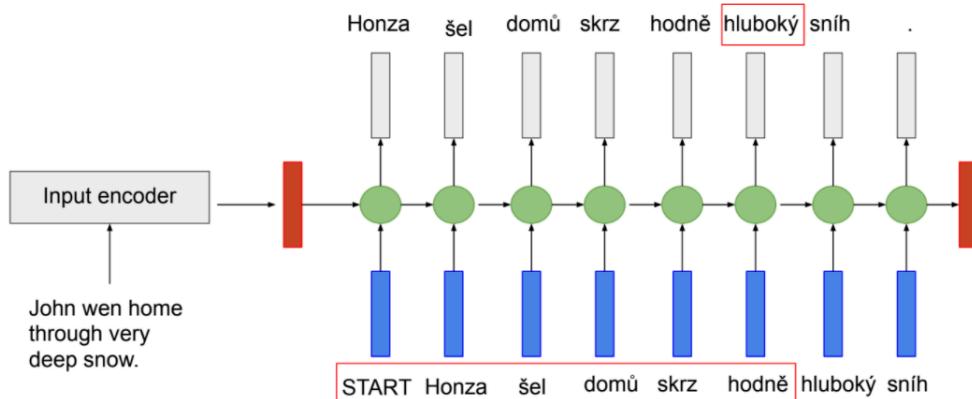
$$P(w_1, w_2, w_3, \dots, w_n) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)\dots P(w_n|w_{n-1}, w_{n-2}, \dots, w_1)$$

$$P(w_1, w_2, w_3, \dots, w_n) = \prod P(w_i|w_{i-1}, w_{i-2}, \dots, w_1)$$

■ seq2seq - pravděpodobnost věty

- autoregresivní model

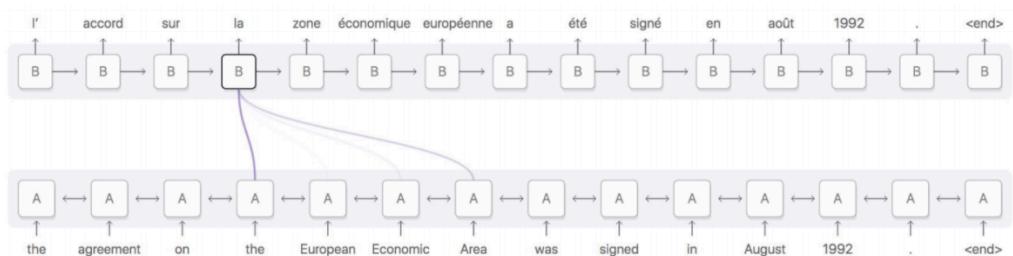
$$P(w_1, w_2, w_3, \dots, w_n) = \prod P(w_i | w_{i-1}, w_{i-2}, \dots, w_1)$$



- model se dělí na encoder a decoder
- vstupní věta se zakóduje v encoderu (šedý box vlevo), čímž vznikne context vector (červený box), který je následně vstupem dekóderu. Ten pak postupně generuje slova, na základě tohoto kontextového vektoru a již vygenerovaných slov
- ptáme se na pravděpodobnost slova, když znám ty předchozí
- takto zjistíme, jak je nějaká věta pravděpodobným překladem jiné věty
- postup vygenerování překladu:
 - máme síť se vstupní větou, která je zpracována na vektor, jejž využíváme jako počáteční stav dekódovací sítě
 - překlad se vytváří krok po kroku - slovo po slově
 - nejprve síť dostane informaci o zahájení překladu - "START" (speciální slovo)
 - síť vyprodukuje rozložení pravděpodobnosti pro první slovo (John → Honza)
 - toto první slovo vybereme a dáme tuto informaci síti → dole na vstupu se objeví "Honza", poté se počítá další iterace rekurentní sítě, která vygeneruje rozložení pravděpodobnosti přes druhé slovo, podle toho vyberu další slovo (went → šel)
 - vybrané slovo dáme opět dolů na vstup rekurentní vrstvy a postupuje analogicky dál...
 - jakmile se na výstupu objeví speciální slovo "STOP", překlad ukončíme
- nedostatky:
 - funguje pouze pro kratší věty, protože větu je potřebné zakódovat do jednoho vektoru, což je při velmi dlouhých větách problém
 - vektor obsahující zakódovanou větu musí při dekódování projít všemi iteracemi rekurentní vrstvy (opět při dlouhých větách velmi složité až nereálné)
- pozornost (attention)
 - při zpracování slova/věty se NS podívá, jaký znak/slovo na které pozici zpracovává

representative	r
representative	e
representative	p
representative	r
representative	e
representative	s
representative	e
representative	n
representative	t
representative	a
representative	t
representative	i
representative	v
representative	e

- o pro každou pozici se počítají váhy z předešlých kroků

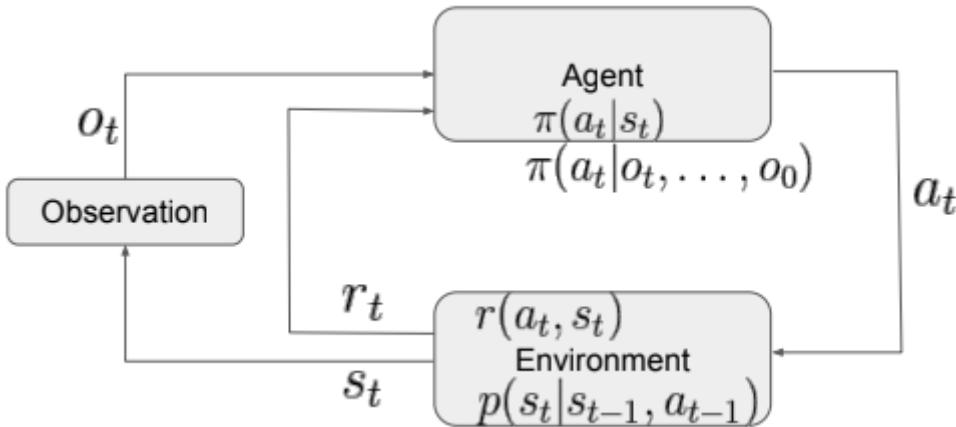


- o existují modely, které pro překlad nevyužívají konvoluce, rekurentní vrstvy a podobně, ale pouze mechanismus pozornost (attention)

prezentace 05 Umělá inteligence a strojové učení - posilované učení

Posilované učení (reinforcement learning)

- pokročilejší téma umělé inteligence
- cílem je vytvořit části SW (agenty), které se budou chovat inteligentně, budou samy schopny najít řešení nějakého problému bez toho, abychom jim museli říct, jaké to řešení je
- u posilovaného učení nemáme k dispozici řešení, ale je k dispozici ohodnocení vygenerovaného řešení (v základní verzi ale neříká proč je řešení špatné)
- definujeme úlohu, která se má vyřešit + funkci, která hodnotí, jak dobře agent pracuje
 - o agent má volnost najít jakékoli řešení problému (někdy překvapivé)
- schéma:



- o_t - pozorování; s_t - stav; $\pi(a_t|s_t)$ - politika; $p(s_t|s_{t-1}, a_{t-1})$ - přechodový model; r_t - odměna; a_t - akce; $r(a_t, s_t)$ - funkce odměn

- **agent** - provádí akce (např. generování dalšího slova) - může se rozhodnout pouze na základě aktuálního stavu (pokud známe stav celého světa) nebo na základě pozorování (i z minulosti)
 - **politika** $\pi(a_t|s_t)$ - funkce, která má konkrétní výstup, na základě kterého se agent rozhoduje na provedení akce (jak se agent rozhoduje)
 - stochastický vs. deterministický přístup - př. pro tuhle konfiguraci šachovnice hraj vždy takto, ev. vybírá z možností
- **svět (environment)** - agent nad ním nemá žádnou vládu; agent generuje akce, které jdou do světa a zpátky od světa dostává odměnu a stav světa (Markovský řetězec) nebo nějakou omezenou informaci o světě (např. obraz)
 - **funkce odměn** $r(a_t, s_t)$ - odměna je podmíněná tím, v jakém stavu svět aktuálně je a jakou v tom stavu agent provede akci
 - **přechodový/dynamický model** $p(s_t|s_{t-1}, a_{t-1})$ - funkce, která říká, jak se svět mění → když je svět v nějakém stavu a agent provede nějakou akci, do jakého stavu se svět může dostat
- **trajektorie** - posloupnost akcí a stavů; pravděpodobnost nějaké trajektorie se vypočítá jako $p(\tau)$ * $p(a_t|\tau)$ * $p(s_t|\tau)$

$$p_\theta(s_1, a_1, \dots, s_T, a_T) = p(s_1) \prod_{t=1}^T \pi_\theta(a_t|s_t)p(s_{t+1}|s_t, a_t)$$

$p_\theta(\tau) \leftarrow$ Trajectory

- **cíl:** agent se snaží dělat takové akce, aby se zvyšovaly odměny

$$\theta^* = \arg \max_{\theta} E_{\tau \sim p_\theta(\tau)} \left[\sum_t r(s_t, a_t) \right]$$

- [...] → suma odměn za život agenta
- když je agent vpuštěn do světa, za svůj "život" posbírá nějaké odměny, přičemž se chová podle nějaké strategie theta, která je parametrizovaná podle nějakých parametrů NS
 - agent v prostředí bude produkovat trajektorie
 - po agentovi chceme, aby v **průměru** posbíral za svůj život co největší odměnu

- problémy v reálném světě
 - neznáme/nemáme přístup k dynamickému modelu, funkci odměn
 - není možné udělat tabulku všech stavů
 - na začátku se spustí agent (který hraje náhodně nebo jinak neoptimálně), nechá se hrát a zaznamenávají se stavy (např. jako čtverice - stav světa, provedená akce, získaná odměna, nový stav světa po provedení akce)
 - je složité vypozorovat, co je ta správná akce, protože odměny přicházejí se **zpožděním** (algoritmus si nějakým způsobem musí spojit odměny s provedenými akcemi, které však nastaly dřív)
 - aby bylo možné nějakou situaci natrénovat, tato situace musí **nastat**

- **co se učit?**

- strategie $\pi(a_t|s_t)$
 - NS, která říká, co se má dělat, jaké akce vybrat
 - dívá se na aktuální stav (nebo sekvenci pozorování)
 - algoritmus policy gradients
- q-funkce $Q^\pi(s_t, a_t)$
 - kvalita akce v daném stavu - pro nějaký konkrétní stav a konkrétní akci - pokud se budu chovat dál podle té strategie, jaké mohu očekávat v budoucnosti odměny

$$Q^\pi(s_t, a_t) = E_\pi \sum_{t'=t}^T r(s_{t'}, a_{t'}) | s_t, a_t$$

set $\pi'(\mathbf{a}|s) = 1$ if $\mathbf{a} = \arg \max_{\mathbf{a}} Q^\pi(s, \mathbf{a})$

- q-funkci použijeme pro výběr dobrých akcí - když mám nějaké ohodnocení, můžu vytvořit novou strategii, která bude vybírat akce tak, že vezme současné ohodnocení (které se NS naučila se starou strategií) a bude vybírat ty akce, které jsou optimální (podle Q)
- pokud uděláme update strategie tak, že podle Q budeme vybírat nejlepší akce, tak vždycky dojde ke zlepšení
- value-funkce ($V^\pi(s_t)$)
 - ohodnocení stavu - když jsem ve stavu s_t , jaké jsou z tohoto stavu očekávatelné budoucí odměny

$$V^\pi(s_t) = E_\pi \sum_{t'=t}^T r(s_{t'}, a_{t'}) | s_t$$

modify $\pi(\mathbf{a}|s)$ to increase probability of \mathbf{a} if $Q^\pi(s, \mathbf{a}) > V^\pi(s)$

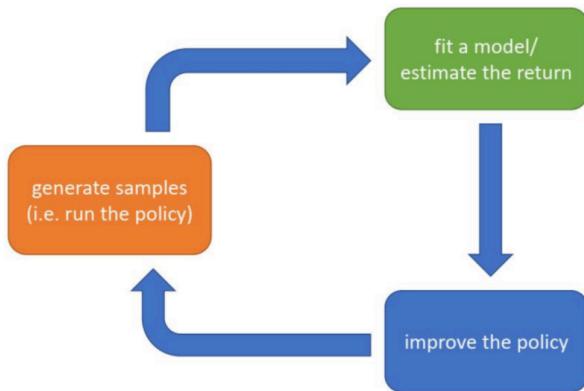
- **typy algoritmů**

- všechny dále uvedené algoritmy řeší problém maximalizace průměrných odměn

$$\theta^* = \arg \max_{\theta} E_{\tau \sim p_{\theta}(\tau)} \left[\sum_t r(s_t, a_t) \right]$$

- **algoritmus policy gradients** - NS, která se učí, které akce mají být provedeny
- **odhad Q-funkce** - konkrétně DQN - algoritmus s 1 NS
- **actor-critic** - třída hybridních algoritmů - 2 NS - jedna odhaduje kvalitu (např. akcí), druhá se učí na základě první a jednotlivé akce provádí

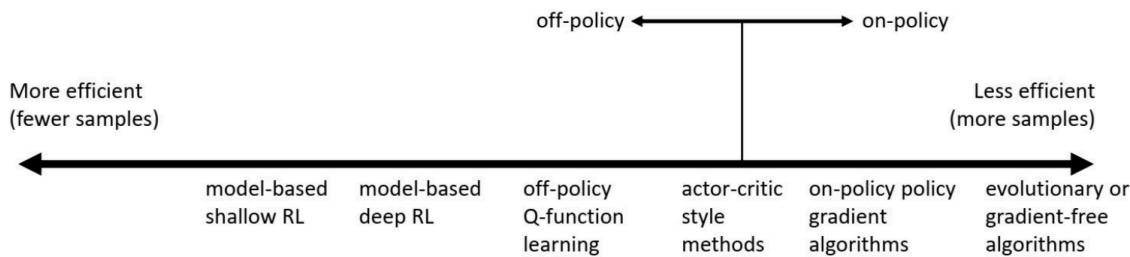
- **fáze učení**



- 3 fáze:

- život agenta v prostředí → mám agenta, nechám ho něco dělat v prostředí, posbírám si zkušenosti, jeho trajektorie a odměny
- nepovinná část, která na základě těchto informací upravuje model
- úprava samotné strategie (někdy se jedná o ručně definovaný algoritmus)

- **využití vzorků**



- efektivita

- vlevo jsou hodně efektivní algoritmy, které nemusí posbírat mnoho informací z prostředí (jízda self-driving auta venku)
- vpravo jsou algoritmy, které jsou velmi neefektivní (velmi rychlá počítačová simulace - např. evoluční algoritmy (náhodné prohledávání))
- on-policy (gradients) algoritmy - pro zlepšení chování nemohou využít staré "vzpomínky"/akce, které byly vykonány před předcházející změnou (mohou využít pouze vzpomínky z "aktuálního" života agenta)
- actor-critic - částečně využívají staré vzpomínky, ale neomezeně
- off-policy, Q-learning - algoritmus využívá staré "vzpomínky"/zkušenosti
- deep RL, shallow RL - nejfektivnější, mají k dispozici model světa nebo se ho učí

Q-learning algoritmus

$$\theta^* = \arg \max_{\theta} E_{\tau \sim p_{\theta}(\tau)} \left[\sum_t r(s_t, a_t) \right]$$

- cíl: hledáme parametry NS, která ovládá chování agenta; hledáme takové parametry, díky kterým agent v průměru nasbírá nejvíce odměn
- učí se funkce Q - v podobě NS, která má na vstupu stav světa nebo několik minulých pozorování, které dostatečně dobře popisují stav světa
- na výstupu NS je ohodnocení/kvalita pro každou možnou akci (tj. vektor)
- pak je možné na základě ohodnocení vybrat akci s nejvyšším ohodnocením a tu provést

- vhodné pokud počet možných akcí není astronomické číslo
- učení NS pro ohodnocení akcí**

- základem je Bellmanova rovnice

$$Q^\pi(s, a) = r(s, a) + E_{s'} [\max(Q^\pi(s', a'))]$$

$$(Q^{\pi_\theta}(s, a) - r(s, a) + E_{s'} [\max(Q^{\pi_\theta}(s', a'))])^2$$

- kvalitu nějaké akce je možné rozložit na 1) jakou odměnu agent dostane, pokud konkrétní akci vykoná a 2) jaké odměny můžeme v budoucnosti očekávat v příštím stavu s příští akcí
- snaha o to, aby se levá strana rovnice rovnala pravé → pomocí mean square error (chybová funkce), kdy se pravá strana odečítá od levé a celé se to umocňuje na 2
- pokud nepoznáme dynamický model, je potřeba se zbavit $E_{s'}$ (expectation - náhoda, že se svět ocitne v nějakém neočekávaném stavu)

$$Q^\pi(s, a) = r(s, a) + E_{s'} [\max(Q^\pi(s', a'))]$$

$$L(\theta) = E_{\pi_\theta} (Q^{\pi_\theta}(s, a) - r(s, a) + E_{s'} [\max(Q^{\pi_\theta}(s', a'))])^2$$

$$L(\theta) = E_{\pi_\theta} (Q^{\pi_\theta}(s, a) - r(s, a) + \max(Q^{\pi_\theta}(s', a')))^2$$

- expectation můžeme přesunout na začátek, protože Q a aktuální odměna nezávisí na tom, do jakého stavu se v budoucnosti dostaneme \Rightarrow expectation na začátku už nebude přes budoucí stavy, ale přes to, jak se agent v prostředí chová
- chybovou funkci pak můžeme vyjádřit jako sumu přes zážitky agenta

$$L(\theta) = \frac{1}{|D|} \sum_{i \in D} (Q^{\pi_\theta}(s_i, a_i) - r(s_i, a_i) + \max(Q^{\pi_\theta}(s'_i, a'_i)))^2$$

- pomocí datasetu se dále optimalizuje chybová funkce (např. pomocí Adama)
 - první Q je výstup NS a tato část se optimalizuje pomocí gradientů \Rightarrow celková očekávatelná budoucí kvalita stavu akce do konce života agenta
 - druhá část ($-r \dots + \max(\dots)$) se neoptimalizuje!
 - funkce odměn $r(s_i, a_i)$ je předem definovaný vstup, pouze se vyhodnocuje, nad ní učení neprobíhá \Rightarrow odměna v daném stavu
 - nestabilita NS: pokud se mění i druhé Q (uvnitř max); finta - musí se měnit pomaleji než první Q
- při každém běhu agenta s určitou politikou se zaznamenávají informace a vytváří se "dataset" (nazývaný replay buffer/experience memory)

$$D = \{(s_i, a_i, s'_i, r_i)\}$$

- shrnutí:

- máme aktuální policy, která je závislá na tom, jaké kvality akcí NS odhaduje
- pomocí toho sbíráme dataset D
- poté se optimalizuje chybová funkce

- výhoda: algoritmus funguje i v případě, kdy záznamy chování nebyly získány pomocí aktuálně nejlepší strategie → mohu použít libovolně staré vzpomínky získané libovolným způsobem a Q-learning se na nich může i tak učit
- proces učení a simulace agenta v prostředí mohou probíhat odděleně a asynchronně

Policy gradients

- pointa: NS, která přímo říká, které akce jsou dobré a mají se udělat

$$(\pi(a_t|s_t))$$

- NS má na výstupu softmax, který vrací vektor pravděpodobností pro každou možnou akci, že se akce má vykonat
- **učení**

- agent s nějakou politikou po běhu vrátí trajektorii (spustí se vícekrát pro získání více trajektorií)
- snaha o zvýšení pravděpodobnosti trajektorií s největšími odměnami a snížení pravděpodobnosti pro trajektorie s nízkymi odměnami
- problém je, že nevíme, které akce přispěly k větším odměnám, proto zvýšíme pravděpodobnost všem akcím v dané trajektorii
- stejně můžeme snižovat pravděpodobnost pro všechny akce v trajektoriích s nízkymi odměnami
 - akce, které vedou k dobrým odměnám, se v průměru častěji budou vyskytovat v dobrých trajektoriích a akce, které vedou ke zlým odměnám, se v průměru častěji budou vyskytovat ve špatných trajektoriích → dobré akce budou častěji zlepšovat pravděpodobnost (budou se stávat více a více pravděpodobnější) a špatné akce zhoršovat pravděpodobnost (budou se stávat méně pravděpodobné)

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) \right) \left(\sum_{t=1}^T r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right)$$

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} J(\theta)$$

- několik trajektorií, z nichž děláme průměr
- první suma přes všechny akce agenta (u gradientu se využívá funkce backward)
- druhá suma vyjadřuje ohodnocení (součet odměn)
- update (2. řádek): uvažujeme gradienty přes více akcí, které váhujeme (druhou sumou v 1. řádku)

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) \right) \left(\sum_{t=1}^T r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right)$$

maximum likelihood: $\nabla_{\theta} J_{ML}(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) \right)$

- algoritmus:
 - zahazuje staré trajektorie/policy
 - vezme aktuální policy, nasimuluje průchody agenta světem a udělá update NS
 - zvláštnost - využívá jednu váhu pro všechny akce v daném životě agenta

- lepší verze algoritmu používá pouze budoucí odměny

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \left(\sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) \right) \left(\sum_{t=1}^T r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right)$$
$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \left[\nabla_{\theta} \log \pi_{\theta}(a_{i,t} | s_{i,t}) \sum_{t'=t}^T r(s_{i,t'}, a_{i,t'}) \right]$$

Actor critic - policy gradients + Q-learning

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \left[\nabla_{\theta} \log \pi_{\theta}(a_{i,t} | s_{i,t}) \sum_{t'=t}^T r(s_{i,t'}, a_{i,t'}) \right]$$

On average, this is
 $Q^{\pi}(s_{i,t}, a_{i,t})$

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T [\nabla_{\theta} \log \pi_{\theta}(a_{i,t} | s_{i,t}) Q_{\omega}(a_{i,t} | s_{i,t})]$$

- vpravo nahoře je váha - suma budoucích odměn (obsahuje šum)
 - **Q-Actor-Critic**
 - sbírá zkušenosti pomocí policy
 - aktualizuje Q na základě shromážděných zkušeností → odhad funkce Q jako v Q-learningu
 - vstup: stav
 - výstup: vektor čísel ohodnocující kvality jednotlivých akcí v daném stavu
 - aktualizuje policy podle upravené Q a získaných zkušeností
- $$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T [\nabla_{\theta} \log \pi_{\theta}(a_{i,t} | s_{i,t}) Q_{\omega}(a_{i,t} | s_{i,t})]$$
- funkce Q dále učí druhou NS, která modeluje policy stejně jako v policy gradient, pouze váhy jsou výstupy z NS pro Q

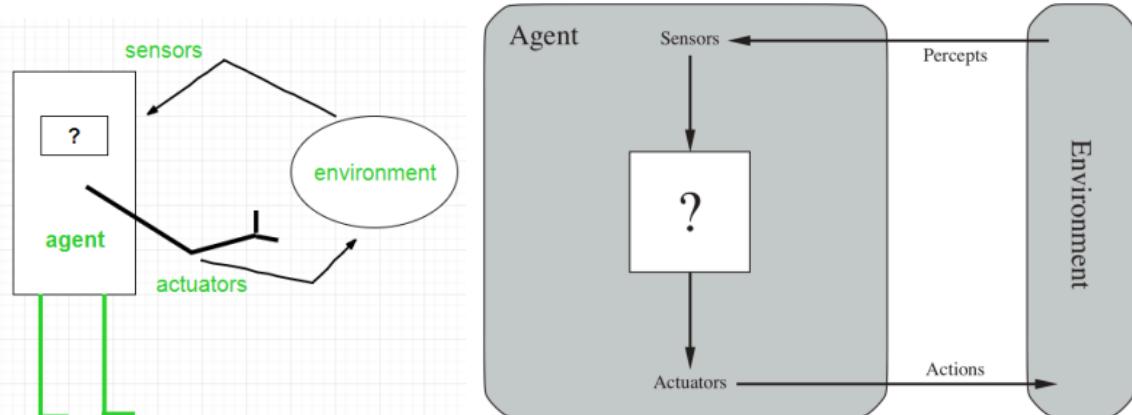
Část III. (ing. Šustek)

prezentace 06

Agentní pojetí

Agent:

- je v prostředí a má
 - aktuátory - výstupy pro informace do prostředí - provádí akce
 - senzory - vstupy pro informace z prostředí - vnímá



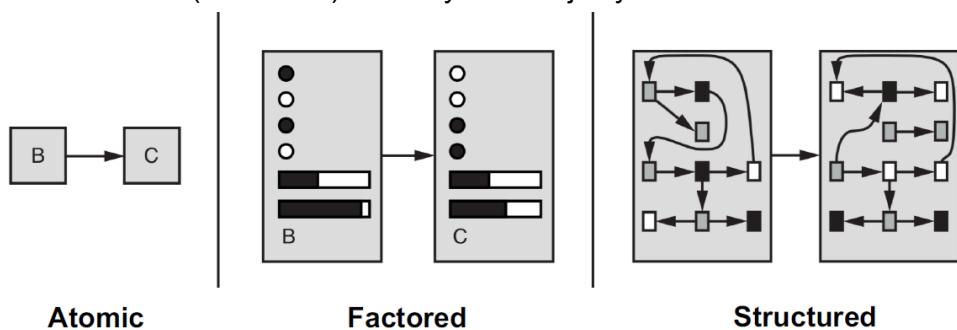
- agentní pojetí
 - percept sequence (sekvence vjemů) – co všechno agent dosud vnímal
 - agentní funkce – popis chování agenta (mapovací funkce, reprezentace např. tabulkou → složité, máme moc možností)
 - agentní program – konkrétní implementace AF
- racionální agent RA
 - jedná nejlépe, jak může
 - performance measure PM (objektivní fce) - jak dobře si vedu
 - vůči stavům prostředí (ne agenta)
 - racionalita u některých lidí
 - ovládá výsledné chování
 - prior knowledge PK (předchozí znalosti)
 - možné akce
 - percept sequence PS (co už „viděl“)
 - racionální agent vybírá akci, která maximalizuje očekávanou PM s ohledem na předchozí znalosti (PK) a to, co už „viděl“ (PS)
- specifikace prostředí úkolu
 - PEAS (Performance, Environment, Actuators, Sensors)
 - P – počet bodů (známka, percentil, ...)
 - E – test (počítač, lavice, učitel, ostatní žáci, ...)
 - A – výběr A, B, C, D
 - S – oči (přečtení otázky)
- vlastnosti prostředí úkolu
 - plně/částečně pozorovatelné (fully/partially observable) – relevantní aspekty → zda je možné z daného místa sledovat všechno, co je potřeba sledovat
 - jeden agent/multiagentní – agent B maximalizuje PM, která závisí na chování ag. A

- competitive, cooperative (soupeřící, spolupracující) → multiagentní prostředí jen pokud je více agentů v jednom prostředí a vzájemně si pomáhají/soupeří
- **deterministické/stochastické** (někdy náhodné - není jisté, že akce bude mít námi požadovaný výsledek)
- **jisté/nejisté** (certain/uncertain)
 - jisté = plně pozorovatelné + deterministické
 - na začátku (ne)dokážeme simulovat, co se stane po daném tahu - (ne)dokážeme určit sekvenci akcí vedoucích do konce
- **episodické/sekvenční** – počítačové hry
 - výsledek jedné hry mi (ne)ovlivní výsledek další hry
- **statické/dynamické** – vzhledem k rozhodování
 - semi-dynamické (pouze změna PM)
 - statické → provedení akce způsobí pokaždě to stejné
- **diskrétní/spojité** – záleží na:
 - stavech
 - vnímání času
 - vjemech
 - akcích
 - pro diskrétní musí být vše uvedené diskrétní, pokud alespoň jedna věc spojitá → spojité
- **známé/neznámé** – znalosti programátora/agenta (př. pravidla pro hraní hry)
 - nezaměňovat s pozorovatelností!
- **realizace agenta**
 - **agent** = agentní **program** + **architektura** agenta
 - program – fyzikální omezení → generalizace
 - má přístup pouze k aktuálním vstupům (vjemům), pokud chce minulé, musí si je uchovat
 - jak paměťově náročné by bylo uchovávat video?
 - matematické tabulky vs. kalkulačky
- rozdělení agentů podle struktury programu:
 - **reaktivní** (simple reflex)
 - rozhoduje se pouze na základě aktuálních vjemů (žádná historie)
 - implementace pomocí if-then konstrukcí
 - pokud „svítí zelená“ → jed!
 - dostatečný pro plně pozorovatelné prostředí
 - co se stane, pokud není pozice vysavače známá?
 - náhodné chování
 - **reaktivní s modelem** (model-based reflex)
 - vytváření modelu
 - jak svět/prostředí funguje → model
 - akce A → vystřel, akce B → posuň vpravo
 - uchování minulých vstupů pomocí **stavu** v modelu
 - **řízený cílem** (goal-based)
 - souvisí se znovupoužitelností
 - dojít z bodu A do bodu B (reaktivní agent), dojít z aktuálního místa na libovolné **cílové** místo (agent řízený cílem)
 - potřeba prohledávání, plánování
 - uvažování o budoucnosti (co se stane, když...; budu pak spokojený?)
 - **řízený užitkem** (utility-based)
 - utility (užitek)

- různé způsoby dosažení cíle nejsou stejně dobré
- pokud užitek (agent) a performance measure (prostředí) navzájem korespondují, pak maximalizace užitku bude racionální chování

- **reprezentace**

- typ reprezentace může ovlivnit, zda jsme schopni něco vyřešit či nikoliv (za limitovaný čas)
- jména (symbolická reprezentace → uchopitelný koncept)
- možné vysvětlení inteligence (před 50 tisíci lety se člověk naučil kombinovat 2 koncepty tak, aby vytvořil 3. → naučili jsme se popisovat věci, souvisí s jazykem)
- představivost – umíme simulovat něco, co se nikdy nestalo
- **typy reprezentace stavů**
 - atomická (atomic) – bez vnitřní struktury
 - faktorizovaná (factored) – pomocí atributů
 - využívá se v NS
 - strukturovaná (structured) – vztahy mezi objekty



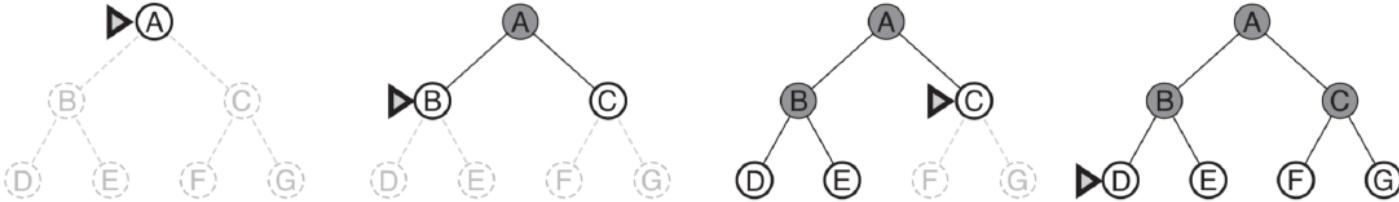
prezentace 07 Řešení problémů prohledáváním

Řešení problémů prohledáváním:

- agentní pojetí:
 - cílem (užitkem) řízený agent
 - atomická reprezentace stavů
- **prohledávání** – hledání sekvence akcí (řešení), které dosáhnou cíle
- druhy:
 - **neinformované** prohledávací metody – algoritmus nevyužívá žádné informace kromě definice úlohy
 - **informované** – využívá, jak je daný stav nadějný
- **postup:**
 - 1. formulace problému
 - 2. prohledávání → nalezení řešení
 - 3. vykonání nalezeného řešení (nezávisle na vjemech z prostředí)
- **formulace problému**
 - základem je **abstrakce** – vynechání detailů z reprezentace (stavů, akcí, ...) irrelevantních pro řešení úlohy
 - abstrakce musí být validní (abstrahované řešení koresponduje s původním)
 - **pouze kladné** ohodnocení cest
 - zaručuje nalezení optimálního řešení!
- **prohledávací strategie: datová struktura**

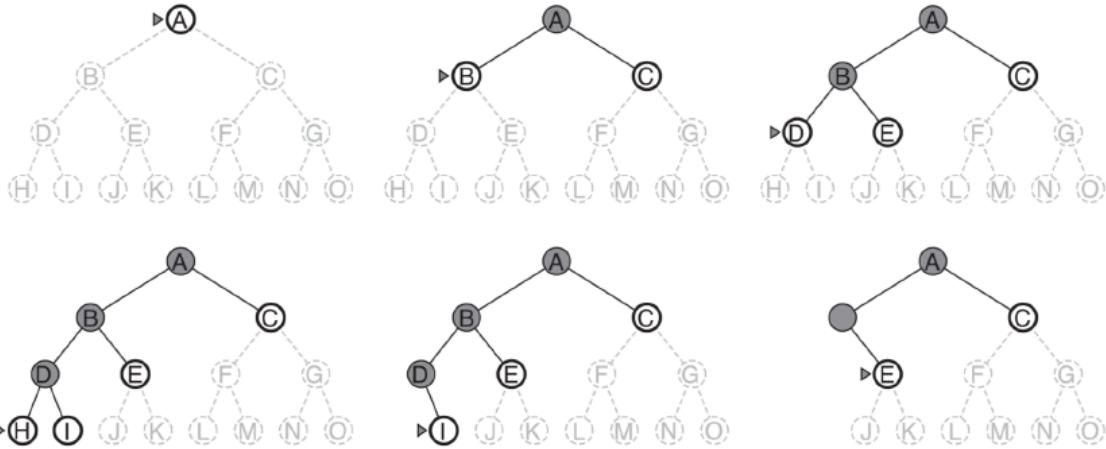
- prohledávací strategie = který uzel vybrat?
 - 1. FIFO (fronta) → prohledávání **do šířky**
 - Breadth-first search (BFS)
 - 2. LIFO (zásobník) → prohledávání **do hloubky**
 - Depth-first search (DFS)
 - 3. prioritní fronta → prohledávání **od nejlepšího**
 - Best-first search
- aby algoritmus pořád neopakoval stejné kroky, musí si svou historii nějak ukládat → seznam **closed**
 - kanonický tvar → např. **hash**
- **hodnocení prohledávacích algoritmů**
 - **úplnost** – pokud existuje řešení, algoritmus ho musí nalézt
 - **optimálnost** – máme garantováno, že nalezené řešení bude to nejlepší (cena cesty)
 - časová složitost
 - paměťová složitost
 - porovnáváme cenu cesty, což odpovídá ceně řešení, u reálných problémů však může být brána v potaz i cena prohledávání (výpočet)

Neinformované metody:

- **BFS - Breadth-first search**
 - prohledávání do hloubky/po řádcích (postupně od nejbližších)
 - fronta (FIFO)
 - **optimální** (pokud jsou všechny akce stejně dobré – fixní cena kroku)
 - cílový stav můžeme testovat už při vygenerování (načítání následníků do fronty)
- 

```

graph TD
    A((A)) --> B((B))
    A((A)) --> C((C))
    B((B)) --> D((D))
    B((B)) --> E((E))
    C((C)) --> F((F))
    C((C)) --> G((G))
  
```
- **DFS - Depth-first search**
 - prohledávání do hloubky (řeší problém s pamětí při prohledávání stromu)
 - zásobník (LIFO), **není optimální**
 - **backtracking search** (varianta DFS), neukládají se následníci, ale provedená akce (šetří paměť)
 - prohledávání stromu je úplné (v **konečném** stavovém prostoru), pokud se ověřuje, že nový stav uzlu je na cestě jedinečný
 - na obrázku **DFS inverse order**
 - tzn. uzly do zásobníku vkládáme v opačném pořadí
 - v základní verzi algoritmu bychom nejprve vybrali ze zásobníku ve druhém kroku uzel C a jako první by se navštívil nejpravější podstrom

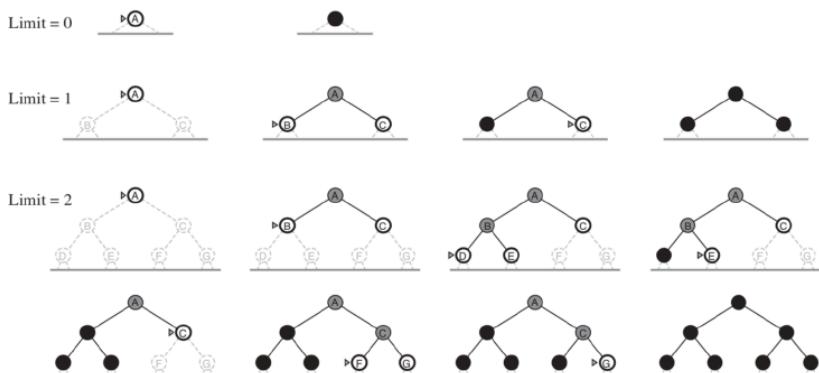


- **DLS - Depth-limited search**

- prohledávání do omezené hloubky
- pokud se cílový uzel nachází hlouběji než se můžeme s daným omezením hloubky zanořit, úloha nemá řešení

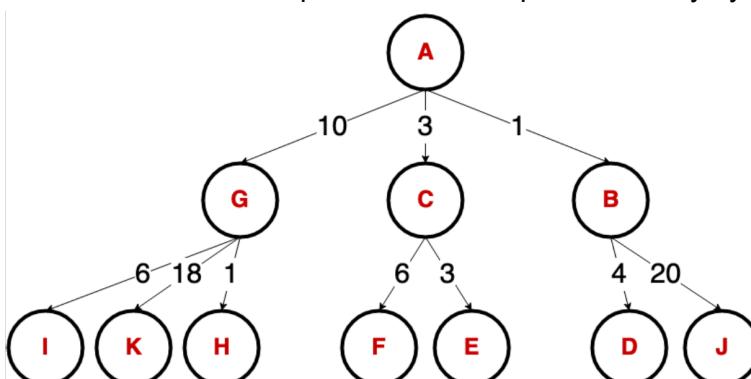
- **IDS - Iterative deepening depth-first search**

- iterativní DLS, začíná se s limitem hloubky 0 a v každé iteraci se o 1 zvyšuje
- zvyšuje hloubku až do nalezení řešení (pokud existuje)
- výhoda: nepotřebujeme tolik paměti jako u metody BFS
- nevýhoda: některé uzly prozkoumáme několikrát



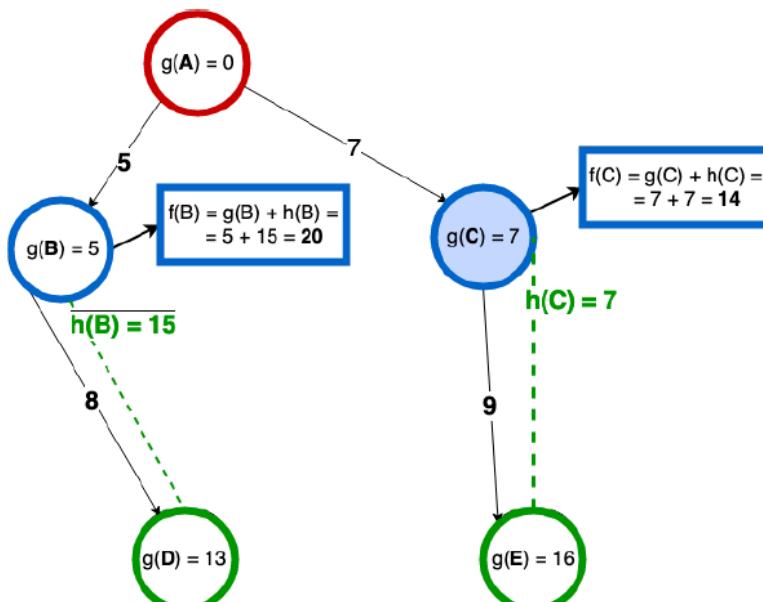
- **UCS - Uniform-cost search**

- každá hrana má ohodnocení a přechody respektují tyto ceny
 - změna struktury (FIFO → prioritní fronta)
- zpracuje se ten uzel, který má nejnižší ohodnocení (nejmenší cenu cesty)
- cíl se musí testovat až při snaze uzel expandovat, aby byla zajištěna optimálnost



- UCS s cenou hran všude = 1 ⇒ algoritmus BFS

- **BS - Bidirectional search**
 - obousměrné prohledávání
 - úloha musí mít reverzibilní operátory
 - jeden cílový stav (nebo je potřeba zvolit)
 - alespoň z jedné strany prohledávání je potřeba uchovávat stavy (paměťové nároky)
- **Informované metody:**
 - implementace metod odpovídá UCS (jediná změna je způsob ohodnocení uzlů)
 - UCS využíval pro ohodnocení uzlu n funkci $g(n)$, která odpovídala nejkratší ceně cesty z počátečního uzlu do uzlu n
 - informované metody se řídí podle funkce $f(n)$, která využívá funkci $h(n)$ a může mít tvar:
 - $f(n) = g(n) + h(n)$
 - $h(n)$ je heuristická funkce a je to **ODHAD** ceny cesty z uzlu n do nejbližšího cílového uzlu
- **Greedy best-first search**
 - využívá se pouze heuristická funkce, $g(n) = 0$
 - $f(n) = h(n)$
 - není optimální
 - úplnost je podobná jako u DFS
 - vhodná metoda, pokud chceme najít nějaké řešení rychle
- **A* search**
 - využívají se obě složky funkce $f(n)$:
 - $f(n) = g(n) + h(n)$
 - **úplná i optimální**
 - platí pro tzv. přípustné heuristiky, kdy odhad $h(n)$ nebude nikdy větší než reálná cena do cíle
 - heuristika je v tomto případě chápána jako způsob **prořezávání (prunning)** – vyloučení některých možností bez jejich prozkoumání
 - nepřípustná heuristika
 - z důvodu špatného odhadu $h(n)$ může dojít k nalezení neoptimálního řešení



- v reálných aplikacích může mít A* také problémy s pamětí, existují proto různé úpravy:
 - IDA*
 - RBFS

- MA*
- SMA*

Heuristiky:

- **heuristická funkce $h(n)$**
 - musí být menší než reálná cena cesty do cíle (pokud chceme mít zaručenou optimálnost), ale na druhou stranu, pokud si můžeme vybrat mezi dvěma heuristikami (obě splňují tuto podmínu) h_1 a h_2 , $\forall n: h_1(n) \geq h_2(n)$, pak použijeme h_1 , protože má **větší hodnoty**, a tím pádem bude nutné prozkoumat méně „slepých cest“
 - větší heuristika nás více přibližuje reálnému řešení, protože je podobnější skutečné cestě
- **vytváření heuristik**
 - jeden ze způsobů, jak vytvářet nové heuristiky, je ignorování některých omezení řešeného problému (**relaxed problem**)
 - zjednodušený problém musí být „lehce řešitelný“
 - co když máme k dispozici K heuristik a **neplatí**, že jedna je lepší než všechny ostatní:
 - $\neg \exists i \forall k \forall n: h_i(n) \geq h_k(n), 0 \leq i < K, 0 \leq k < K$
 - vytvoříme novou heuristiku $h_K(n) = \max(h_0, \dots, h_{K-1})$
 - také se můžeme „učit“ heuristickou funkci pomocí např. neuronových sítí (ale nemusí být optimální)

prezentace 08 Lokální prohledávání, prohledávání ve spojitéch prostorách

Lokální prohledávání:

- neprohledáváme systematicky celý prostor, jsme pouze v jednom (nebo i více) „**aktuálním**“ uzlu (stavu), který vyhodnocujeme a modifikujeme
- metody využívají **málo paměti**, jsou použitelné pro řešení úloh ve velkých (i spojitéch) prostorách, kde nemáme zdroje pro prozkoumání všech možností
- abychom získali nový uzel ze současného, využíváme **sousednost uzlů** (např. skrize akce, náhodná malá změna, kombinace více uzlů)
- algoritmy neberou v potaz cenu za provedení akcí, řešením **není sekvence akcí**, ale jeden uzel (stav) nebo bod v prostoru, snahou je najít „co nejlepší“ řešení podle nějaké vyhodnocovací metriky
- někdy známe ideální hodnotu vyhodnocovací metriky, můžeme tak s určitostí říci, že jsme našli hledané řešení a ukončit prohledávání; jindy řešíme **optimalizační úlohu**, kdy hledáme nejlepší z možných řešení, ale často dopředu neznáme ideální hodnotu a máme omezené výpočetní prostředky (nelze zkoušit vše)

Algoritmy pro lokální prohledávání:

- **hill-climbing search**
 - jako nový uzel se zvolí nejlepší ze sousedních uzlů (**steepest-ascent** varianta), který je zároveň lepší než současný uzel; pokud takový není, výsledným řešením (výstupem algoritmu) je současný uzel
 - také označovaný jako **greedy local search**
 - problematické jsou lokální extrémy a části, na kterých je funkce konstantní (**plateau**)
 - **možné úpravy**
 - **stochastic hill climbing** – pokud je více sousedních uzlů lepších než aktuální, nemusí se volit ten nejlepší

- **first-choice hill climbing** – první nalezený soused, který je lepší než aktuální uzel, se zvolí jako následník (vhodný, pokud má uzel mnoho sousedů)
- **random-restart hill climbing** – jeden běh algoritmu nalezne **lokální extrém**, pokud iterativně spouštíme algoritmus s (náhodnou) inicializací, zvyšujeme pravděpodobnost nalezení globálního extrému (eventuálně musíme nalézt glob. extrém)
- **simulated annealing (simulované žíhání)**
 - kombinace hill-climbing a náhodné procházky (volíme náhodného souseda)
 - náhodně vybíráme sousední uzel, pokud je lepší než současný, nahradíme, pokud je horší, spočítáme, o kolik je horší ΔE (záporné číslo)
 - myšlenka snižování teploty T v každé iteraci (čím nižší je teplota, tím menší pravděpodobnost nahrazení horším uzlem)
 - princip: dokážeme překonat malé lokální extrémy a přehoupnout se přes ně
 - pravděpodobnost nahrazení horším uzlem se spočítá podle vztahu $e^{\Delta E/T}$

⇒ srovnání



- **local beam search**
 - vychází z **beam search** – princip je podobný BFS, ale uchováváme pouze omezené množství uzelů
 - nejprve expandujeme počáteční uzel, ponecháme však pouze k nejlepších následníků, pak expandujeme těchto k následníků, z nichž ponecháme opět k nejlepších atd.
 - mírně modifikovaná varianta se využívá u strojového překladu (**machine translation MT**)
 - **local beam search** můžeme chápat jako **beam search** inicializovaný několika počátečními uzelů
 - princip: využijeme zároveň k běhů programu, kde různé běhy nespouštíme sekvenčně (random-restarts), ale **paralelně**
 - po pár krocích se může stát, že všechna řešení jsou velmi podobná (málo diverzity) → co s tím? **stochastic beam search**
 - umožníme, aby jako následníci mohly být zvoleny i uzly, které nejsou nejlepšími sousedy současných uzelů; **lepší ohodnocení** souseda by mělo **zvyšovat pravděpodobnost**, že bude vybrán jako následník
- **genetic algorithm (genetický algoritmus)**
 - **motivace ke genetickému algoritmu**
 - jedna z myšlenek přirozeného výběru – jedinci, kteří jsou „lepší“, mají větší šanci na přežití (rozmnožení); pokud se na aktuální uzel díváme jako na populaci jedinců, na sousední uzel jako na potomky, pak **stochastic beam search** připomíná **nepohlavní** rozmnožování
 - **genetický algoritmus** může být chápán jako varianta **stochastic beam search**, kde nový uzel vzniká kombinací dvou uzelů
 - z biologického hlediska odráží evoluční procesy organismů rozmnožujících se **pohlavně**
 - **fitness funkce** – vyhodnocovací metrika (objektivní funkce) u GA
 - **selection (selekce)** – výběr rodičů
 - **crossover (křížení)** – proces vytvoření nového potomka z rodičovských uzelů

- **mutation (mutace)** – náhodná změna potomka
- nejprve se selekcí vyberou rodiče, pak se křížením vytvoří potomci a následně v některých případech dochází k mutaci, nakonec se upraví původní populace a proces se opakuje
- existuje mnoho variant z hlediska selekce, křížení, mutace, ...; při řešení některých typů úloh lze využít pouze specifické varianty



Lokální prohledávání ve spojitém prostoru:

- dva přístupy
 - spojitý prostor si navzorkujeme
 - řešíme prohledávání spojitého prostoru (viz předešlé přednášky - NS, regrese, maximalizace a minimalizace funkcí, gradientní sestup...)
- př. hledání **shluků** (clusters)

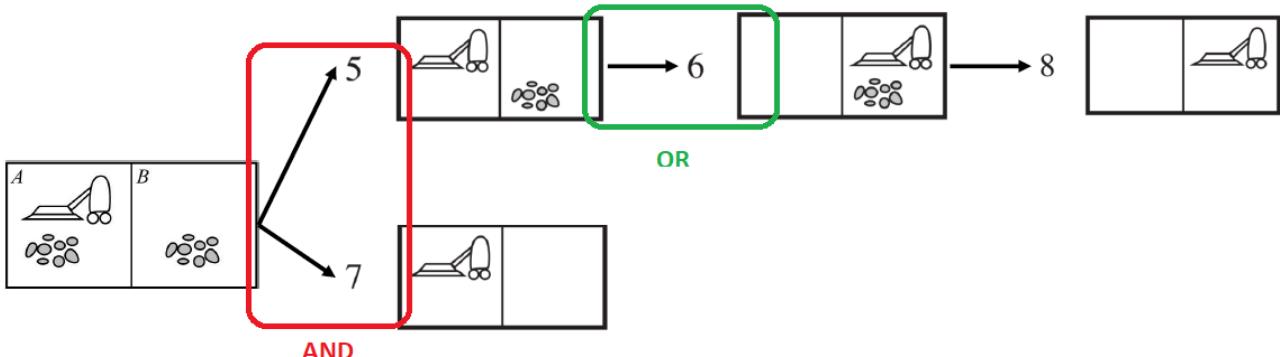
prezentace 09 Prohledávání v nejistých prostředích

Nedeterministické akce:

- **nejisté prostředí**
 - takové, které není plně pozorovatelné
 - obsahuje alespoň jednu nedeterministickou akci
- zadaný problém již nelze řešit pouze pevně danou sekvencí akcí, ale potřebujeme vytvořit nějaký **plán/strategii** (contingency plan, strategy)
- co se změní oproti předchozí definici problému?
 - výsledkem $RESULT(s, a)$ je nyní **množina** nových stavů
 - cena $COST(s, a, s')$ provedení akce a ve stavu s závisí také na novém stavu s' , ve kterém se ocitneme po provedení akce
- dříve jsme věděli, co se stane po provedení akce, plán = sekvence akcí (nezávisle na vjemech)
- nyní se rozhodujeme i na základě vjemů
- **selhání při vykonání akce**
 - pokud chceme nalézt řešení problému, kdy provedení akce "selže", musíme povolit cykly a garantovat, že se pouhým opakováním jednou dostaneme do alternativního požadovaného stavu
 - řešením může být opakovat stejnou akci, dokud se nedostaneme do kýzeného stavu → **nekonečný cyklus**
 - je třeba změna formulace: plně pozorovatelné nedeterministické prostředí → **částečně pozorovatelné a deterministické prostředí**

AND-OR prohledávací stromy:

- **OR** – můžu si vybrat, kterou akci provedu
 - deterministické akce vyjadřují pouze OR uzly
- **AND** – výsledkem akce může být více stavů, řešení pro všechny možné nové stavы



- **dynamické programování** – myšlenka znovupoužitelnosti již provedeného výpočtu
 - pokud po prozkoumání podstromu uzlu se stavem A zjistíme, jestli je řešitelný, narazíme-li znova na uzel se stavem A , tak již neprovádíme výpočet, ale rovnou použijeme uložené řešení
- pokud narazíme na stav, který se již na cestě k tomuto uzlu nachází, pak dál neprohledáváme, abychom neprocházeli pořád ty stejné sekvence stavů (detekovali jsme **cyklus**) → algoritmus skončí v konečném stavovém prostoru

Částečně pozorovatelné prostředí (ČPP):

- bavíme se o **belief state** – reflektuje agentovu důvěru o aktuálním stavu (v jakých stavech se může nacházet)
- **lokální senzor**: vysavač nedokáže určit, jestli je vedlejší pole čisté nebo špinavé (senzory dokáží detektovat pouze čistotu aktuálního pole)
- **prohledávání bez senzorů**:
 - nemá senzory → nepozorovatelné prostředí
 - př. širokospektrá antibiotika
 - pro N možných stavů je 2^N možných **belief states**
 - nemusí být problém, protože u většiny problémů je velká část stavů nedosažitelná
 - obtížná reprezentace **jednoho** belief state, 10 políček k vysávání → 2^{10} kombinací špinavých a čistých polí (repr. výčtem stavů není možná)
 - **incremental belief-state search** – postupně hledáme řešení pro každý stav; výhoda: pokud neexistuje řešení, zjistí se rychle; jiná řešení později
- **lokální senzor**:
 - nedeterminismus v částečně pozorovatelných prostředích plyne z nemožnosti predikovat, co bude výsledkem akce (které vjemem agent obdrží)
 - každý belief state (Bstate) je reprezentován atomicky (bez vnitřní struktury), pro optimalizaci můžeme využít, že některé stavы jsou nadmnožiny/podmnožiny jiných:
 - 1. neprozkoumávat některé cykly
 - 2. pokud známe řešení pro nadmnožinu stavů → lze aplikovat i na podmnožinu stavů
 - **udržování si Bstate** je základem inteligentního agenta v ČPP (většina reálných prostředí)
 - přirovnání k rekurentní síti s textem → jedno slovo je Bstate, celý text síť nezná
- **lokalizace** - určení, kde je agent, př.:
 - agent má mapu areálu, je na neznámé pozici, ale dokáže určit, jestli je ve směru každé světové strany překážka nebo volno

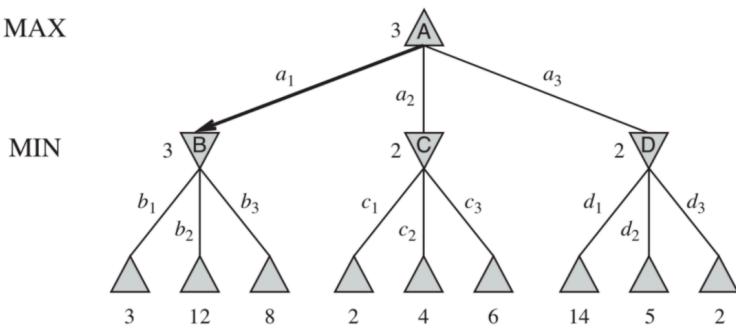
prezentace 10 Adversarial search

Adversarial search:

- typ prohledávání v soutěživém prostředí, často ho označujeme jako hry → teorie her (game theory)
- nejčastěji se jedná o speciální druh her:
 1. **hry s nulovým součtem (zero-sum game)**
 - součet zisku/užitku všech hráčů je konstantní – typ her, ve kterých není možné, aby všichni prosperovali (lepší výsledek je na úkor druhých)
 2. tahové
 3. dvou hráčů
 4. perfect information (plně pozorovatelné)
 5. deterministické
- **multiagentní** prostředí, pokud se v prostředí vyskytují dva agenti, chápeme to tak, že nám druhý agent (hráč) chce škodit
- prozkoumat všechny možnosti je u většiny her (stejně jako rozhodování v reálném světě) nemožné → přesto nutné zvolit nějakou akci (tah)
 - **pruning (prořezávání)** – ignorování částí prohledávacího stromu, které (většinou) neovlivňují výslednou volbu
 - **heuristická vyhodnocovací (evaluační) funkce** – často pouze odhadneme ohodnocení stavu (užitek) bez provádění úplného prohledávání

Hry dvou hráčů:

- hráč na tahu je **MAX** (A), druhý hráč je **MIN** (B)
- hodnota (např. x) u uzlu odpovídá užitku (utility) pro hráče **MAX**, užitek hráče **MIN** je $-x$
- pokud bychom neuvažovali pouze hry dvou hráčů s nulovým součtem, pak by přibyla otázka kooperace mezi agenty, ta ovšem v tomto případě není racionální
- hra má svůj strom (**game tree**), my ovšem prohledáváme pouze některé části tohoto stromu, což je náš **prohledávací strom (search tree)**
- **AND-OR pojetí**:
 - můžeme chápát stejně jako AND-OR úlohu (**OR** odpovídá **volbě našeho tahu**, **AND** odpovídá všem možným **tahům protihráče**)
 - za předpokladu, že by ohodnocení stavu bylo globálně určeno (užitek je subjektivní) a protihráč by byl racionální → soupeř by vždy zvolil pro nás nejhorší tah → **minimax** algoritmus
 - pokud se jedná o hru dvou hráčů, algoritmus lze také interpretovat jako minimalizace možných ztrát = volba nejbezpečnějšího řešení
- řešením je **tah a_1** , celé "kolo" (tah A i B) je označováno jako **move**



- **real-time rozhodování s omezenými prostředky**

- ani s prolezáváním nelze prozkoumat celou hru → některé uzly neprohledáváme, ale aplikujeme **heuristickou vyhodnocovací funkci eval**:
 1. při dosažení stanovené hloubky prohled. stromu
 2. použít **cutoff test**, který pro každý uzel určí, jestli se dále zanořovat, nebo jestli aplikovat funkci *eval*
- dobrá vyhodnocovací funkce koresponduje s pravděpodobností **výhry** (nejistota je způsobena omezenými výpočet. prostředky, ne prostředím!)
 - často zaměřené **materiálně** (šachy: počet různých figurek) i **strukturálně** (šachy: je král v bezpečí?) → 2 části

- **hry více hráčů**

- každý hráč **maximalizuje svůj užitek** (není to minimalizace ztrát; chceme-li min. ztráty, hráče C chápeme jako druhý tah hráče B = tvoří alianci)
 - spolupráce může být důsledek maximalizace užitku
- pokud férová hra, vybíráme v dané úrovni maximum z označených čísel (v pravém podstromu analogicky)

to move

A

B

C

A

(1, 2, 6)

(1, 5, 2)

(1, 2, 6)

(6, 1, 2)

(1, 5, 2)

(5, 4, 5)

(1, 2, 6)

(6, 1, 2)

(5, 1, 1)

(7, 7, 1)

(4, 2, 3)

(7, 4, 1)

(1, 5, 2)

(5, 4, 5)

(6, 1, 2)

(7, 4, 1)

(5, 1, 1)

(7, 7, 1)

(5, 4, 5)

(7, 4, 1)

(5, 4, 5)

(5, 4, 5)

Stochastické hry:

- hry s náhodností (hod kostkou, náhodná karta, ...) → **expectiminimax** algoritmus
- přibydlou náhodné uzly (**chance**) – nedokážeme předpovědět, „co padne“ → užitek uzlu je vážený průměr užitků potomků (váha = pravděpodobnost)
 - výsledkem je **očekávaný (expected)** užitek – tzn. pokud bychom daný tah hráli mnohokrát, byl by to průměrný užitek
- **nevývážená ohodnocení**
 - jelikož váhujeme pravděpodobnosti, vyhodnocovací funkce by měla být pozitivní lineární transformace pravděpodobnosti výhry
 - dříve důležitý pouze relativní vztah mezi hodnotami

- **Monte Carlo simulace**
 - prohledávání všech možností je náročné, jednou z možností, jak approximovat distribuci, je **vzorkování** (výběr náhodných vzorků) → čím více vzorků, tím přesnější odhad
 - **Monte Carlo simulace** – provedeme několik (tisíce) simulací her až do konce (např. s pomocí alpha-beta nebo náhodných tahů) s náhodnými „hody kostkou“ (před každou hrou se **dopředu** znova náhodně určí výsledky „hodů kostek“)
 - předpokládáme, že procento výher koreluje s tím, jak je daný tah dobrý → zvolíme ten nejlepší
- **částečně pozorovatelné hry:**
 - snaha není pouze provést nejlepší tah, ale takový, který soupeř **nepředpokládá** (abychom mu neposkytl naše „tajně informace“)
 - **blafování** – ochota hrát nepředvídatelně (někdy náhodně), abychom zmátlou soupeře
 - karetní hry, kde jsou na začátku rozdány karty:
 - použití **Monte Carlo simulace** může způsobit problém, protože simulujeme hry, ve kterých se rozhodujeme, jako kdybychom viděli soupeřovi do karet → řešení může být odlišné oproti původní specifikaci úlohy (tentotýden se někdy nazývá **averaging over clairvoyance**)

Možná vylepšení:

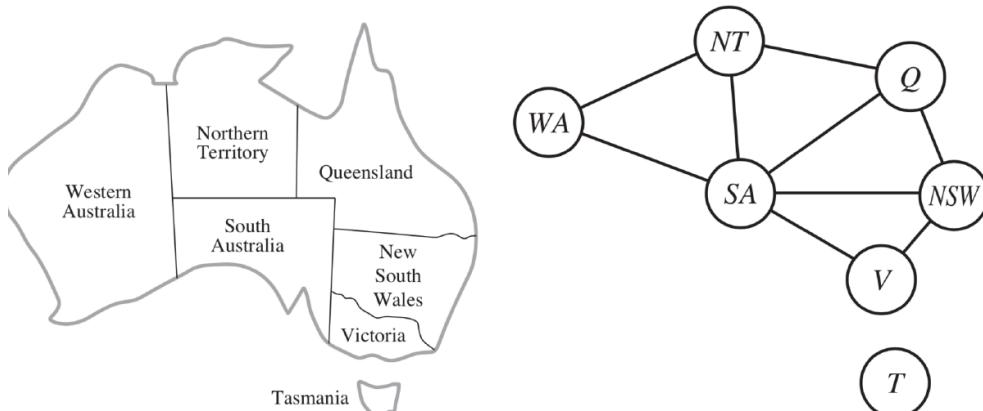
- **pořadí tahů**
 - myšlenka: na pořadí prověrování tahů záleží, pokud budeme nejdřív prověrovat slibné tahy, častěji dojde k odříznutí podstromu při využití alpha-beta prořezávání → **nikdy nebude ideální**
 - pokud bychom dokázali vymyslet funkci, která by řadila tahy podle nejlepších, funkce samotná by pak stačila pro výběr ideálního tahu
 - dynamické řazení tahů – na základě předchozí zkušenosti, hry, tahu, hloubky aktuálního zanoření při využití IDS
- **transpozice**
 - některé stavy hry jsou dosažitelné více permutacemi tahů → **transpoziční tabulka** – hashovací tabulka s uloženými prozkoumanými stavy (stejná myšlenka jako u seznamu closed)
 - zrychlení za cenu paměťové náročnosti
- **quiescent search**
 - vyhodnocovací funkci bychom měli aplikovat na pozice, kde **neočekáváme náhlou změnu vyhodnocovací funkce** v blízké budoucnosti, takové pozice jsou „klidné“ (**quiescent**) → **quiescent search**:
 - místo aplikace vyhodnocovací funkce *eval* na uzel (dosažení limitu hloubky prohledávání), který není quiescent, provedeme další zanoření tak, abychom funkci *eval* použili na quiescent stavy (myšlenka stability)
 - pointa: máme konkrétní stav, v následujícím stavu (o úrovně dál) se může rapidně změnit ohodnocení, proto se zanořujeme ještě na další úrovně, abychom si byli jistí, že nedojde k výrazné změně
- **potlačit efekt horizontu**
 - **efekt horizontu (horizon effect)**:
 - potřeba provést „bolestivý“ a „nevynutelný“ tah, který je možné oddalovat za cenu menších ztrát (omezená hloubka prohledávání → neviditelný)
 - splacení kreditní karty, prokrastinace
 - **singulární rozšíření (singular extension)** – pro zmírnění **efektu horizontu**:
 - narazí-li se během standardního prohledávání na „jasně lepší“ tahy, tak si je zapamatujeme a uvažujeme, zda se nedají aplikovat jako extra tahy navíc (na uzly, na které aplikujeme *eval*)

- **více prořezávání**
 - **dopředné prořezávání (forward pruning)** – v každém kroku uvažovat pouze několik (n) nejlepších tahů (myšlenka beam search)
 - nebezpečné, protože lze odříznout nejlepší tah
 - člověk například u šach také neuvažuje všechny tahy, ale jen pár slibných
 - **null move** heuristika – na začátku hry necháme soupeře tahnout dvakrát → spočítáme spodní odhad ohodnocení, který využijeme pro účely prořezávání
- **lookup table**
 - použití **vyhledávací tabulky (lookup table)** místo prohledávání – na počátku a na konci hry často nebývá tolik možností
 - například počáteční tahy u šachů můžou být manuálně definované člověkem
 - konec hry může být někdy vyřešen analyticky – např. u šachů lze pro kombinace král + pěšec vs. král najít výherní pozice a provést zpětné (retrograde) prohledávání tak, aby se každá pozice označila jako vedoucí k výhře (nezávisle na tazích MIN), remíze nebo prohře (nezávisle na tazích MAX) → výpočet je před samotnou hrou a tahy jsou uloženy v paměti

prezentace 11 Constraint satisfaction problems

CSP - úlohy s omezujícími podmínkami:

- opět nehledáme sekvenci akcí/tahů, nyní se ani **nesnažíme najít nejlepší řešení**
- hledáme pouze nějaké řešení, které splňuje předem stanovené **podmínky pro kombinaci hodnot proměnných**
- **faktorizovaná** reprezentace stavů – stav reprezentujeme několika proměnnými, které mohou nabývat předem definovaných hodnot
- vizualizace pomocí **constraint graph**, omezení v podobě hran
- př. barvení map



- úlohou jeobarvit všechna města v mapě tak, aby žádná dvě sousední města nebyla obarvenou stejnou barvou
- $X = \{WA, NT, Q, NSW, V, SA, T\}$
- $D_i = \{\text{red}, \text{green}, \text{blue}\}$
- $C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$
- **formální definice CSP**
 - X je množina proměnných $X = \{X_1, \dots, X_n\}$
 - D je množina domén $D = \{D_1, \dots, D_n\}$, D_i je doména pro X_i , $1 \leq i \leq n$

- C je množina omezení, která definují povolené kombinace hodnot
 - omezení můžeme zadat různou formou, např. výčtem hodnot, rovností či nerovností
 - př.: $X_1 < X_2$; $3X_2 + 1 = X_3$; $\{X_1, X_2, \{(0,0), (1,4)\}\}$
 - někdy využíváme zápis $Alldiff(X_1, X_2, X_3)$, čímž míníme, že proměnné musí mít odlišné hodnoty
- **assignment (přiřazení)**
 - jelikož je potřeba přiřadit nějakou přípustnou hodnotu proměnným, bavíme se o **přiřazení** (assignment)
 - přiřazení je **konzistentní/legální** (consistent/legal), pokud neporušuje žádnou podmínu
 - přiřazení je **úplné** (complete), pokud má každá proměnná přiřazenou hodnotu

Naivní prohledávání:

- v každé úrovni stromu se přiřadí hodnota jedné proměnné, po přiřazení všech se zkoumá, jestli je řešení konzistentní

Inference:

- (odvozování) je alternativa k prohledávání
- slouží pro zmenšení domén konkrétních proměnných → zmenšení prohledávacího stromu
- uvažujeme inferenci nazývanou constraint propagation CP (**propagace omezení**), inferenci lze provést před samotným prohledáváním nebo se může s prohledáváním prolínat
 - **konzistence uzlů (node consistency)**
 - odpovídá 1-consistency
 - **inference CP** – snahou je snížit počet hodnot v doménách, což zrychlí prohledávání
 - předpokládáme, že jsme úlohu převedli na constraint graph
 - aby měl graf **konzistentní uzly**, nesmí se stát, že doména proměnné obsahuje hodnotu, která porušuje unární omezení (omezení se vztahuje jen k jedné proměnné – např.: $X_1 \neq 3$ nebo $X_2 = 1$)
 - **konzistenci uzlů** dosáhneme aplikováním unárních omezení na **domény** → úprava domén úloh, např.: $D_i = \{1, \dots, 5\} \rightarrow D_1 = \{1,2,4,5\}, D_2 = \{1\}$
 - **konzistence hran (edge consistency)**
 - odpovídá 2-consistency
 - jako konzistence uzlů, ale pro **binární** omezení
 - pro každou hodnotu z domény proměnné X_j existuje nějaká hodnota v doméně proměnné X_i , $i \neq j$ tak, že přiřazení této kombinace je konzistentní – pro každou hodnotu je na druhé straně každé hrany alespoň jeden „kamarád“
 - pro $X_2 = X_1^2$:
 - $D_1 = \{1,2\}, D_2 = \{1,4\}$
 - pro $X_2 = 2X_1$, $X_3 = X_2 - 1$, $Alldiff(X_1, X_2, X_3)$:
 - $D_1 = \{1,2\}, D_2 = \{2,4\}, D_3 = \{1,3\}$
 - algoritmus **AC-3** – vytvoření konzistentních hran
 - **konzistence cest (path consistency)**
 - odpovídá 3-consistency
 - náročnější a „dražší“ oproti binárnímu omezení
 - kombinace binárních omezení dohromady nebo ternární omezení (např. $X_1 \neq X_2, X_1 \neq X_3, X_2 \neq X_3$ nebo $Alldiff(X_1, X_2, X_3)$) – pro každé konzistentní přiřazení hodnot proměnným $X_i, X_j, i \neq j$ musí existovat hodnota pro X_k , $i \neq k, j \neq k$
 - pro $X_2 = 2X_1$, $X_3 = X_2 - 1$, $Alldiff(X_1, X_2, X_3)$:

- $D_1 = \{2\}, D_2 = \{4\}, D_3 = \{3\}$
- lze pokračovat dál, obecně **K-consistency**
 - pro každé konzistentní přiřazení $k - 1$ proměnných musí existovat konzistentní přiřazení pro k proměn
- *Alldiff* omezení → inference **naked triples**
 - př. sudoku: domény 3 políček stejného bloku jsou podmnožinou $\{1,3,5\} \rightarrow$ na ostatních políčkách ve stejném bloku se nemůže nacházet 1, 3 ani 5

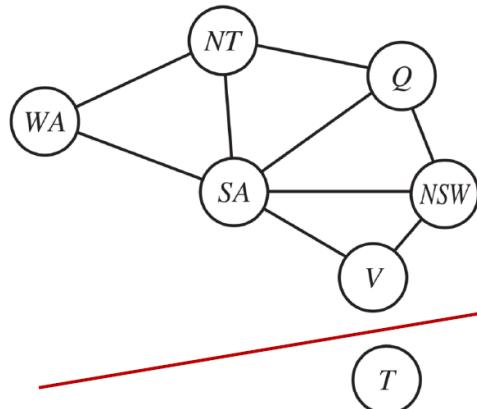
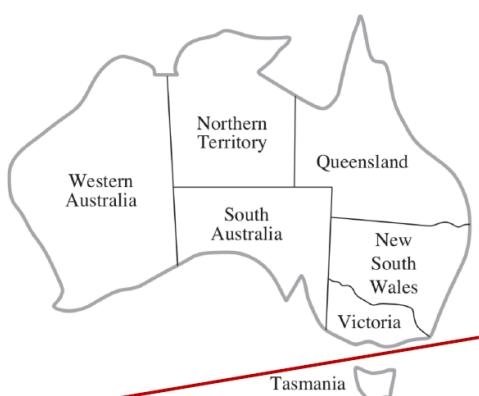
Prohledávání:

- **backtracking**
 - pro prohledávání využíváme backtracking
 - zodpovězením následujících otázek můžeme urychlit prohledávání:
 1. kterou **proměnnou** přiřadit teď? (X_1 nebo X_2 ?)
 2. kterou **hodnotu** přiřadit teď? ($X_1 = 1$ nebo $X_1 = 2$?)
 3. máme použít nějakou **inferenci** v aktuálním kroku? pokud ano, jakou?
 4. došlo ke konfliktu, kam se vrátit? jaká kombinace hodnot je problematická? mohu příště neopakovat?
 - touto otázkou se nebudeme zabývat, pro zájemce např. metoda **conflict-directed backjumping**
 - **výběr proměnné**
 - **minimum-remaining-values** (most constrained variable) – myšlenka „nejmenšího větvení“, zvolíme proměnnou, které lze přiřadit nejméně hodnot
 - **degree heuristic** – zvolíme proměnnou, která se vyskytuje v největším počtu omezení nepřiřazených proměnných (v grafu obsahující pouze nepřiřazené proměnné bude mít tento uzel největší stupeň = nejvíce hran)
 - **výběr hodnoty**
 - **least-constraining-value** – přiřadíme hodnotu, která ponechá nejvíce hodnot u sousedních nepřiřazených proměnných (myšlenka ponechání co největšího počtu řešení)
 - hledáme-li jedno **řešení**, zajímá nás pouze co nejrychlejší nalezení **prvního**, pokud úloha nemá řešení nebo máme zájem o nalezení všech, pak je pořadí výběru hodnot pro přiřazení irrelevantní
 - **inference**
 - **forward checking**
 - po každém přiřazení hodnoty do proměnné X_i se provádí inference, aby se zaručila **konzistence hran aktuálně přiřazované proměnné X_i** (odebrání nekonzistentních hodnot u proměnných, které jsou s X_i spojené hranou, **neprovádí se pro celý graf**)
 - **lokální prohledávání**
 - aktuální uzel má přiřazené všechny hodnoty proměnných → snažíme se modifikovat hodnotu vybrané proměnné tak, abychom získali konzistentní řešení (snižujeme počet konfliktů)
 - **min-conflict** – zvolit hodnotu minimalizující počet konfliktů; připomíná hill-climbing
 - funguje velmi dobře pro problém N-dam
 - můžeme také využít metody uvedené v přednášce o lokálním prohledávání
 - vhodné, když se omezení mění (**online settings**)
 - **možné úpravy:**
 - **tabu search** – uložíme několik naposledy navštívených stavů a zabráníme algoritmu, aby se do těchto stavů vrátil

- **constraint weighting** – některá omezení může být velmi obtížné splnit, jiná mohou být celkem „jednoduchá“ → nebudeme přikládat každému omezení stejnou váhu a brát v potaz pouze počet konfliktů, ale tuto **váhu se budeme učit** na základě toho, jak se nám daří toto omezení plnit (obtížně splnitelné omezení = velká váha)

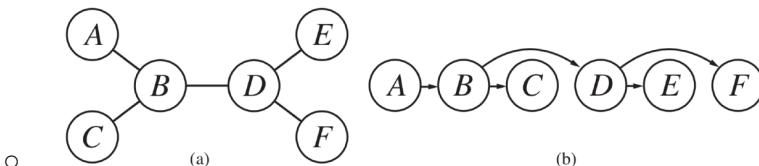
Nezávislé problémy:

- rozdelení úlohy na nezávislé části – zřídka kdy
 - **separátní řešení, zrychlení**



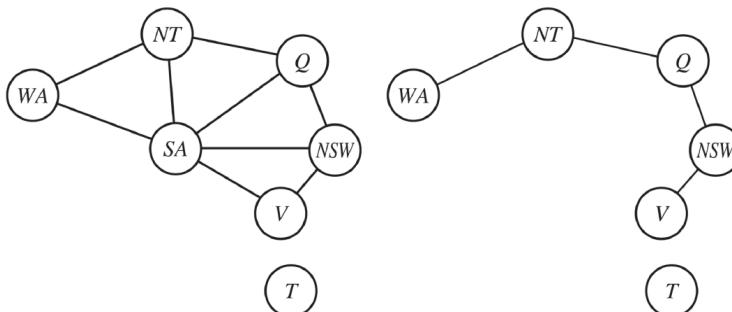
- **stromy**

- každé dvě proměnné jsou spojeny maximálně jednou cestou
 - lze řešit rychleji – určíme topologii (pořadí přiřazení)
 - určíme kořen, potomek musí být přiřazen až po rodiči
 - projdeme ve zvoleném pořadí, vyžadujeme konzistenci hran → řešení

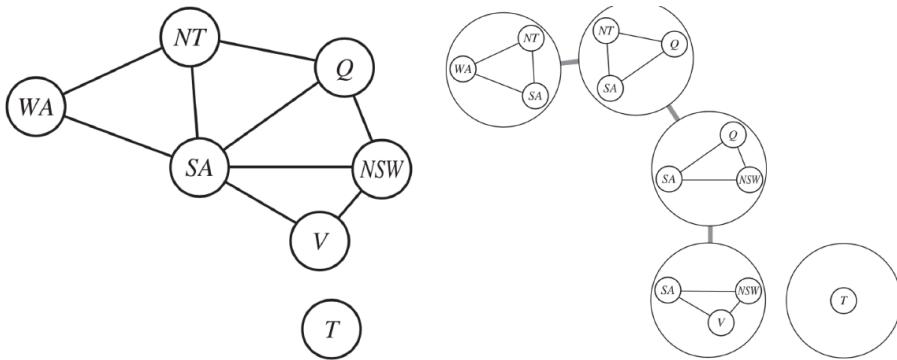


- **graf → strom**

- myšlenka převodu obecného grafu na strom:
 1. odstranění uzlů, aby zbytek tvořil strom(y)



2. dekompozice na několik podproblémů



- **symmetry-breaking constraint**

- *Alldiff* omezení u příkladu barvení map způsobí, že nalezneme několik řešení pouze výměnou barev (modře obarvená města obarvíme červeně, červeně obarvená obarvíme modře), je výhodné přidat **symmetry-breaking constraint** (odstranění symetrie – myšlenka kanonického tvaru) – přidáme omezení na kombinaci barev, např. přiřadíme barvám ordinální hodnoty a požadujeme $NT < SA < WA$
 - může pomoci obzvláště při dekompozici na několik problémů

prezentace 12
Agenti založení na logice

Agenti založení na logice:

- budeme navrhovat agenty, kteří budou:
 - uchovávat **reprezentaci světa** (vytvářet si **model**)
 - používat **inferenci** (vyvozování na základě logiky), aby si vytvořili novou reprezentaci světa
 - na základě nové reprezentace světa **odvodit**, co by měli udělat
- lidé používají proces **uvažování** (reasoning), který využívá interní reprezentaci znalostí → v umělé inteligenci je reflektováno u **agentů s bází znalostí** (knowledge-based agents)
- agenti založení na logice jsou považováni za značně pokročilé (z pohledu logiky, uvažování, ...)
- používání **atomické** reprezentace stavů je **nepraktické** → agent může reprezentovat to, co ví, pouze jako **výčet možných stavů**
 - použijeme **faktORIZovanou** reprezentaci stavů (CSP) a využijeme **logiku** pro reprezentaci znalostí
 - základem bude **báze znalostí** (knowledge base KB)

Báze znalostí – KB:

- množina platných vět (v nespecifikovaném jazyce)
 - knowledge representation language
 - **axiom** – věta, která platí bez dokazování
 - někdo nám tuto větu (formuli) řekl a my mu věříme
- komunikace s KB
 - **tell** – přidání nových vět (formulí) do KB
 - **ask** – zodpovězení otázky na základě vět v KB
 - obě operace mohou zahrnovat **inferenci** (odvození nových vět)
 - například z uložených (**tell**) vět „je teplo“ a „pokud je teplo, tak mám radost“ mohu položit dotaz (**ask**) „mám radost?“
- výhoda oproti ML:
 - ML nám nikdy nebude garantovat 100% bezchybnost
- **KB agent - agent využívající bázi znalostí**

- volání **agentního programu** v čase t :
 1. uložení (**tell**) vjemů (percept) do KB pro čas t
 2. akce = dotaz (**ask**), kterou akci provést v čase t
 3. uložení do KB, že agent tuto akci provedl v čase t
- agent pracuje na **úrovni znalostí** – specifikujeme, **co agent ví a čeho má dosáhnout** (nezávisle na implementační úrovni)
 - agent pracující s mapou může využívat fotky z družice nebo mít uloženou abstrahovanou mapu, pro nás je podstatné, že pokud mu např. sdělíme, že je most uzavřen, pak přes něj nenaplánuje cestu
- **deklarativní** přístup – neříkáme agentovi, „**jak**“ má něco udělat, ale pouze „**co**“ má udělat
 - tento přístup se uplatňuje u KB agentů
- **procedurální** přístup – to, co má agent udělat, přímo ve formě kódu (programu), případně příkazů v libovolném jazyce (včetně přirozeného)
- úspěšný agent by měl kombinovat prvky z obou přístupů

Logika:

- **syntax** – určení vět (formulí) patřících do jazyka
- **sémantika** – význam korektních vět jazyka
 - nyní uvažujeme, jestli je věta pravdivá (True/False) s ohledem na všechny možné světy (**modely**)
 - např. věta $\alpha: x + y = 3$ je pravdivá v modelu, kde $x = 1, y = 2$, ale nepravdivá pro $x = 0, y = 0$
 - všechny modely, kde je věta α pravdivá, označujeme jako $M(\alpha)$
 - tvrzení, že věta β je **logickým důsledkem** věty α značíme jako $\alpha \models \beta$ (v každém modelu, kde je α pravdivá, je pravdivá i β) $\equiv M(\alpha) \subseteq M(\beta)$
 - $\alpha: \forall x x > 4, \beta: \forall x x > 3 \rightarrow \alpha \models \beta$
- **inference**
 - inferenční algoritmus, který odvodí pouze logicky **platné věty** (entailed sentences), je **sound** (truth-preserving); dokáže-li odvodit všechny odvoditelné věty, pak je **úplný** (complete)
 - pokud algoritmus není **sound**, dokáže si „vymýšlet věty“, je možné, že odvodí něco, co není pravda
 - pokud algoritmus není **úplný**, existují logické důsledky (věty), které nejsou odvoditelné
 - **model checking** je **sound** a pro konečné modely je i **úplný**
 - lze provádět pouze pro malé množství potenciálních modelů

Výroková logika:

- **výrok** (symbol) – tvrzení, jehož platnost ověřujeme
 - např.: $P80 \equiv$ „Počet žáků ve třídě je větší než 80“
- **logické spojky** – negace (\neg), konjunkce (\wedge), disjunkce (\vee), implikace (\rightarrow), ekvivalence (\leftrightarrow)
 - implikace nevyžaduje kauzalitu
 - $BrnoInUSA \equiv$ „Brno se nachází v USA“, $V \equiv 1 > 2$
 - $V \rightarrow BrnoInUSA$ je platná formule
- symbol ve výrokové logice má hodnotu True nebo False \rightarrow pro N symbolů existuje 2^N různých modelů
- **modelování**:
 - **sémantiku** výrokům dáváme my, například zápis $B_{1,1} \leftrightarrow (P_{1,2} \vee P_{1,2})$ může znamenat, že:
 - vánek na pozici [1,1] se vyskytuje právě tehdy, když je díra na pozici [1,2] nebo [2,1] (nebo má matematický význam *or*, nejdří se o výlučné nebo – *xor*)
 - babička narozena prvního ledna přijde na návštěvu právě tehdy, když dostane pivo z první police ve druhé lednici nebo z druhé police v první lednici

Theorem proving inference:

- definice:

- inference přímo aplikovaná na věty (formule) v KB, nezávislé na konkrétním modelu
- **logická ekvivalence**
 - dvě věty jsou logicky ekvivalentní, pokud jsou pravdivé ve stejných modelech
- **platná (valid) formule – tautologie**
 - pravdivá ve všech modelech
- **dedukce (deduction theorem)**
 - pro každé formule α a β , $\alpha \models \beta$ **právě tehdy**, když je formule $\alpha \Rightarrow \beta$ **platná**
- **splnitelná (satisfiable) formule**
 - existuje model, ve kterém je formule pravdivá
- **SAT** problémy – je zadáná formule splnitelná?
- platnost a splnitelnost spolu souvisí
 - věta α je splnitelná právě tehdy, když neplatí, že $\neg\alpha$ je platná
 - $\alpha \models \beta \Leftrightarrow (\alpha \wedge \neg\beta)$ není splnitelná
 - důkaz sporem

Důkaz:

- odvozovací pravidla:

- **modus ponens** (pravidlo odloučení)
$$\frac{\alpha \rightarrow \beta, \alpha}{\therefore \beta}$$
- **eliminace konjunkce** (And-Elimination)
$$\frac{\alpha \wedge \beta}{\therefore \alpha}$$
- jakákoli logická ekvivalence lze použít jako odvozovací pravidlo
- pro odvozování ve výrokové logice nám stačí 3 schémata axiomů a jediné odvozovací pravidlo modus ponens
- **monotónní logika:**
 - když máme nějaké tvrzení a dokážeme ho, tak se nikdy nemůže stát, že by přestalo platit
 - pokud platí $KB \models \alpha$, pak přidání nové věty β do KB nemá vliv na již dokázaná tvrzení α : $KB \wedge \beta \models \alpha$
- **nemonotonní logiky:**
 - rovněž existují, více se podobají přemýšlení lidí → změna názoru
 - pokud je možné, že se výrok Q zmení na výrok $\neg Q$, pak je třeba zajistit, aby všechny důsledky odvozené z Q byly odstraněny
 - **truth maintenance systems**

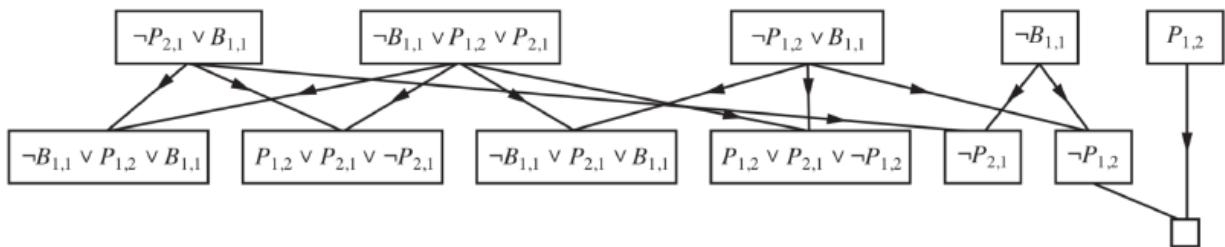
- rezoluční pravidlo:

- úplnost předchozího přístupu záleží na zvolených pravidlech a prohledávacím algoritmu → **rezoluce** je nejen **sound**, ale také **úplná** pro úplný prohledávací algoritmus
- příklad rezoluce, z α a β odvodíme γ :
 - $\alpha: \neg A \vee \neg B \vee C \quad \beta: \neg D \vee B$
 - $\gamma: \neg A \vee C \vee \neg D$

Rezoluce:

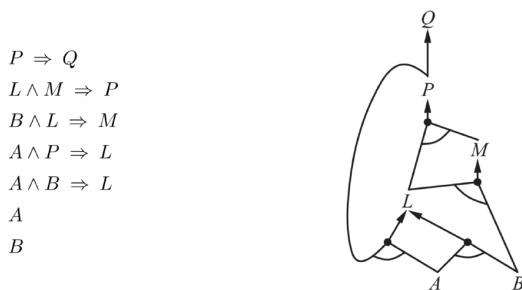
- **klauzule** je disjunkce literálů (literál = výrok nebo negace výroku)
 - poznámka: konjunkci literálů taktéž někdy označujeme jako **klauzuli**
- každá formule lze převést do CNF (konjunktivní normální formy – konjunkce klauzulí)
 - formuli reprezentujeme jako množinu klauzulí (všechny klauzule jsou spojeny konjunkcemi)

- klauzuli reprezentujeme jako množinu literálů (všechny literály jsou spojeny disjunkcemi)
- abychom dokázali $KB \models \alpha$, ukážeme, že není splnitelné ($KB \wedge \neg\alpha$)
- **postup:**
 1. přidat do KB negaci dokazovaného tvrzení $\neg\alpha$
 2. převedeme KB do CNF
 3. iterativně přidáváme nové klauzule do KB spojováním již existujících:
 - pokud získáme prázdnou klauzuli, pak lze na základě KB odvodit tvrzení α
 - pokud nelze přidat žádná nová klauzule, pak nelze na základě KB odvodit tvrzení α



Inference pro Hornovy klauzule:

- rezoluce funguje pro libovolné formule, pokud ale využijeme **Hornovy klauzule** (maximálně jeden pozitivní literál; implikace, kde premisa odpovídá konjunkci pozitivních literálů), existují efektivní algoritmy (lineární vzhledem k velikosti KB)
 - $\neg P_{1,1} \vee P_{2,2} \vee \neg P_{3,1}$ je Hornova klauzule
 - $\neg P_{1,1} \vee P_{2,2} \vee P_{3,1}$ není Hornova klauzule
- **2 algoritmy:**
 - **forward chaining**
 - odvodí se všechno, co lze (je-li dotaz na určitý výrok, pak po odvození výroku algoritmus končí)



- **backward chaining**
 - někdy nepotřebujeme odvodit vše, ale stačí nám vědět, jestli lze odvodit jeden konkrétní výrok, postupujeme pak od konce a prohledáváme s využitím AND/OR algoritmu
 - odvozením všech možných důsledků vzniká velké množství irrelevantních informací → lidé (většinou) používají forward chaining velmi obezřetně, aby neodvozovali něco nevyužitelného
 - často může být mnohem rychlejší než forward chaining

Efektivní model checking:

- myšlenka je v převedení $\alpha \models \beta$ na ověření nesplnitelnosti formule $(\alpha \wedge \neg\beta) \rightarrow \text{SAT}$
- využíváme stejné prostředky jako u CSP úloh, nyní mohou být proměnné pouze True nebo False
- stejně jako u CSP úloh lze řešit pomocí dvou přístupů (**SAT solvers**):
 - upravením backtrackingu
 - lokálním prohledáváním
 - řešení hledáme omezenou dobu

- pokud ho nalezneme, pak **neplatí** $\alpha \vDash \beta$
- pokud řešení nenalezneme, pak buď neexistuje a tedy **platí** $\alpha \vDash \beta$ nebo jsme nehledali dost dlouho
- známý je algoritmus **WALKSAT** (připomíná simulované žíhání)
- lidská analogie: „Mohu provést tohle?“
 - „Ne, tohle nejde!“ nebo
 - „Snažil jsem se najít nějaký případ, kdy by to nefungovalo, ale na nic jsem nepřišel → snad to bude fungovat“

Limity výrokové logiky:

- **Predikátová logika:**
 - př. pro 16 políček máme 256 symbolů → v predikátové logice máme kvantifikátory (for all)
- **Teorie pravděpodobnosti:**
 - **qualification problem** – specifikování všech výjimek je složité