KATEDRA
INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

# KMI/UMIN - Umělá inteligence

Poznámky z výuky *(2025 – 2026)*
Verze z *14. ledna 2026*

**Vojtěch Netrh**
vojtanetrh@gmail.com

# Obsah

# 1 Týden 01 - neuron a perceptron

22. září 2025

## 1.1 Turingův test

Něco jako myšlenkový experiment. AI testem projde pokud lidský tazatel není po položení několika otázek schopen říct zda písemné odpovědi pochází od člověka nebo od počítače. Úplný Turingův test pak vyžaduje interakci s předměty a lidmi v reálném světě.

Počítač musí umět: zpracování jazyka, reprezentaci znalostí, automatické uvažování, strojové učení.

## 1.2 Perceptron

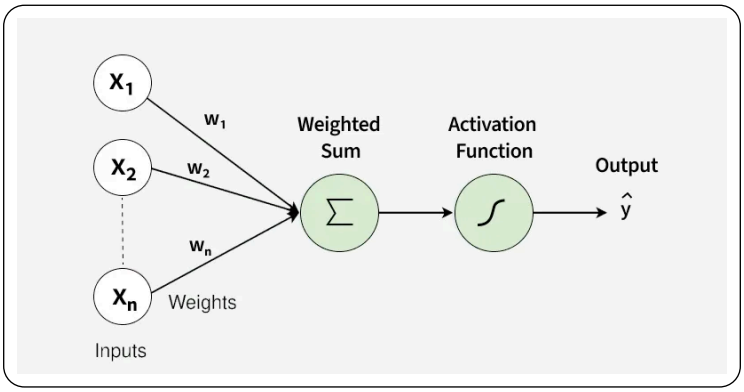
Vychází z biologického modelu neuronu. Má několik vstupů a jeden výstup. Každý vstup má přiřazenou váhu, která určuje jeho důležitost. Také ma určen práh exitace (aktivace), který rozhodne zda-li se daný neuron na dané vstupy „aktivuje“.

**Perceptron** **Definice 1**

Jednoduchý perceptron je výpočetní jednotka s prahem $\theta$, která při přijetí $n$ vstupů $x_1, x_2, ..., x_n$ přes hrany s příslušnými vahami $w_1, w_2, ..., w_n$ vydá 1 pokud platí následující:

$$\sum_{i=1}^{n} w_i x_i \geq \theta$$

a jinak 0.




Obrázek 1: Perceptron.

Občas se $-\theta$ označuje jako $b$ a říká se jí bias. Alternativně jde perceptron zapsat jako matematická funkce:

$$f(x) = \begin{cases} 1 & \text{pokud } w^T \cdot x \geq \theta \\ 0 & \text{jinak} \end{cases}$$

Funkce perceptronu je složená ze 2 matematických funkcí **agregační funkce** $\mathbb{R}^n \to \mathbb{R}$ (což je vážený součet vstupních hodnot) a **aktivační funkce** $\mathbb{R} \to \{0, 1\}$ (což je schodková funkce – dobrý vědět jak vypadá graf). Práh exitace taktéž můžeme vnímat jako další vstupní váhu.

Příklad výpočtu je jednoduchý, prostě se vynásobí složky vektoru a váhy a udělá se součet, který se pro výsledek porovná s prahem exitace.

### 1.2.1 Geometrická interpretace perceptronu

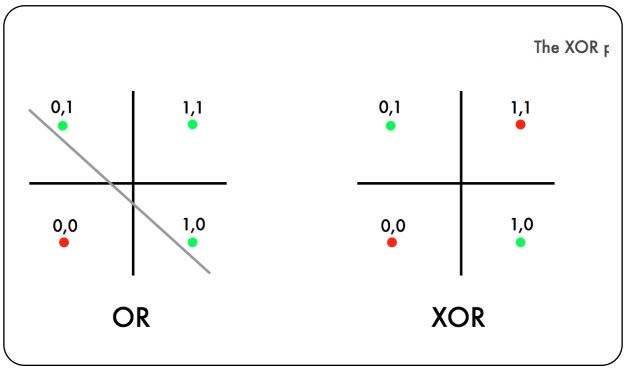
Pokud máme 2 vstupy $x$ a $y$, váhy $a$ a $b$ a $c = -\theta$, pak nám z nerovnice (ta kde byl předtím vektor a váhy) vypadne

$$ax + by + c \geq 0$$

neboli obecná rovnice poloroviny. Z toho plyne že neuron rozděluje rovinu přímkou (při 3 vstupech by rozdělil prostor polorovinou). Tím rozděluje i body do kategorií. Vektor vah $w$ slouží jako určovač směru tohoto rozdělení, zatímco práh $b$ určí vzdálenost od počátku souřadnic. Z této geometrie plyne omezení, že perceptron může řešit jen lineárně separabilní problémy.

### 1.2.2 XOR problém

Pomocí jednoho perceptronu můžeme bez problému reprezentovat AND i OR (viz grafy), ale XOR (non-ekvivalence) a ekvivalence nejde.




Obrázek 2: OR a XOR v grafu.

**Lineárně separovatelné funkce** **Definice 2**

Dvě množiny bodů $A$ a $B$ v $n$-rozměrném prostoru se nazývají lineárně separovatelné, pokud existuje $n+1$ reálných čísel $w_1, ..., w_{n+1}$ takových, že každý bod $(x_1, ..., x_n)$ v $A$ splňuje

$$\sum_{i=1}^{n} w_i x_i \geq w_{n+1}$$

a každý bod $(x_1, ..., x_n)$ v $B$ splňuje

$$\sum_{i=1}^{n} w_i x_i < w_{n+1}$$

Prostě se to dá rozdělit na ty 2 části přímkou a body musí ležet buď na jedné nebo druhé straně (proto ty znaménka nerovnosti).

**Perceptron a XOR problém** **Poznámka 3**

Perceptron může reprezentovat jen lineárně separovatelné funkce.

Řešení XOR problému je tedy použít více perceptronů a „zapojit“ je do sebe (vícevrstevné sítě). Stejné jak skládání logických funkcí pro XOR. V reálném světě problémy nejsou obvykle lineárně separabilní, ale mají šum. Tady pak nehledáme dokonalé oddělení, ale minimalizujeme chybu.

## 1.3 Učení perceptronu

Máme nějaká trénovací data a na začátku náhodně perceptron volí váhy. Probíhá v iteracích. Pokud je výstup správný, tak se váhy nezmění, jinak ano.

$P$ … pozitivní vzory (label 1)
$N$ … negativní vzory (label 0)

Hledáme vektor $w$, který má kladný skalární součin se všemi rozšířenými vektory reprezentovanými body v množině $P$ a záporný skalární součin s vektory v množině $N$.

Pro ukázku se používají klasické logické funkce (příklady a ukázky v prezentaci 01 na slajdech 60 a dál).

**Co se může perceptron naučit** **Poznámka 4**

Pokud jsou $P$ a $N$ lineárně separabilní, perceptron se Hebbovským učením naučí dokonala po konečném množství kroků.

## 1.4 Vícevrstvé neuronové sítě

Přechod k tzv. Multi-layer perceptrons umožnil řešit i neseparabilní úlohy. Organizace do vrstev je z praktických důvodů. Tyto vrstvy lze rozlišit do 3 základních kategorií: vstupní vrstva (vpodstatě ani není vrstvou perceptronů), skryté neurony a výstupní vrstva.

# 2 Týden 02 - trénování vícevrstevných sítí

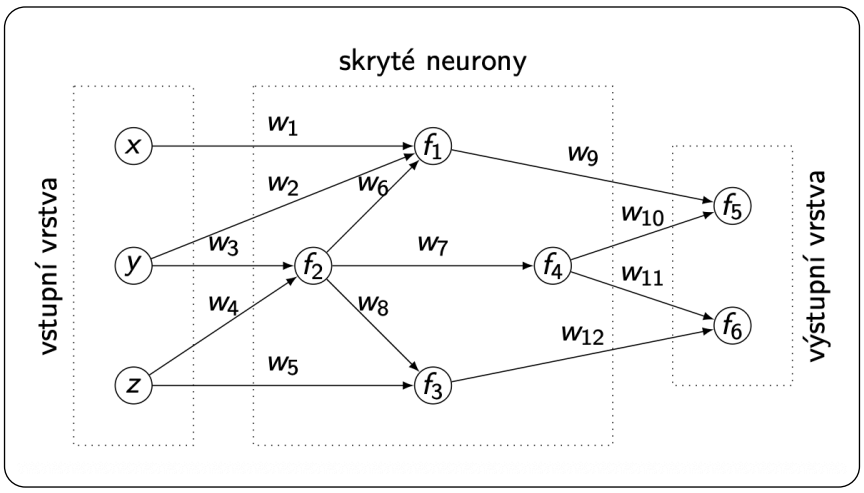
26. září 2025

## 2.1 Dopředná neuronová síť (FFNN)

Anglicky *Feedforward Neural Network*. Tuhle síť tedy dělíme do oněch 3 kategorií/vrstev. Jedná se o základní stavební kámen AI a strojového učení. Jedná se o jednosměrné sítě bez zpětné vazby. Každý skrytý uzel má danou aktivační funkci $f : \mathbb{R} \to \mathbb{R}$ a hodnotu bias $b$.

**Feedforwad Neural Network** **Definice 5**

FFNN je orientovaný acyklický graf s ohodnocenými hranami, kde každý uzel je neuron a každá hrana je váhové spojení mezi neurony.




Obrázek 3: Detailní pohled na FFNN.

Dohromady dá celou funkci $F$, která vznikne skládáním $f_1, ....$ Je vyhodnocována v bodě $x, y, z$.

Jako aktivační funkce se používají různé funkce (jednoduchá je schodková funkce).

## 2.2 Trénování vícevrstvých sítí

Nutné 3 základní pojmy:

- **Trénovací množina** – sestává z $p$ dvojic $\langle x_i, t_i \rangle$ (pro $i = 1, ..., p$) kde $x_i \in \mathbb{R}$ a $t_i \in \mathbb{R}^m$
- **Ztrátová funkce** – vlastně metoda nejmenších čtverců definována jako:

$$E = \frac{1}{2} \sum_{i=1}^{1} \| o_i - t_i \|^2$$

- **Gradientní sestup** – tím hledáme minimum chyb sítě (pomocí **derivací**). „Hýbu se“ opačným směrem něž jde gradient, abych to zmenšil (gradient ukazuje směr největšího růstu funkce). Dobře je to vidět na příkladu *[TBA z prezentace 02]*.

### 2.2.1 Výpočet gradientu

Gradient funkce je vektor skládající se z parciálních derivací funkce podle každé proměnné. Vzorec pro výpočet:

$$\nabla f = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots \right)$$

kde $\partial f$ je parciální derivace (ta říká jak rychle se mění funkce pokud měníme pouze 1 proměnnou).

Chceme aby funkce chyby sítě byla **hladká a diferencovatelná**.

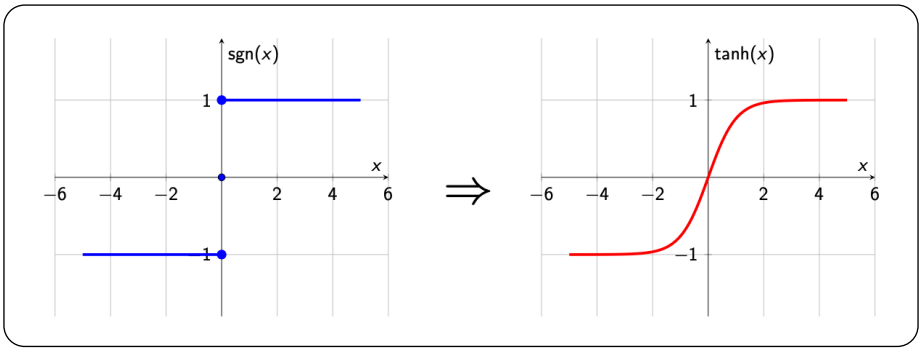
Kvůli nevhodnosti původní aktivační funkce, která byla schodková a nevhodná (má bod, kde není derivace a všude jinde je 0), ji nahradíme **sigmoidní funkcí**[1]. V podstatě vyhladíme schodkovou funkci. Pokud bychom chtěli výstupy $\{-1, 1\}$ místo $\{0, 1\}$, tak použijeme funkci *signum*, kterou když vyhladíme tak získáme *hyperbolický tangens*.

Aktivační funkce by nesmí být lineární, protože by to přineslo několik problémů:

1. Mohli bychom modelovat jen lineární funkce
2. *[další důvody TBA]*

Taky nesmí být polynomická, protože by byla omezená schopnost aproximovat složitější NN.

$$\text{Funkce sigmoida: } \sigma(x) = \frac{1}{1 + e^{-x}};\ \sigma : \mathbb{R} \to (0, 1)$$




Obrázek 4: Signum a její vyhlazení hyperbolický tangens.

**Věta o univerzální aproximaci** **Theorem 6**

Standardní dopředná neuronová síť s jedinou skrytou vrstvou obsahující konečný počet neuronů dokáže aproximovat (napodobit) jakoukoli spojitou funkci na kompaktních podmnožinách $\mathbb{R}_n$ s libovolnou přesností.

## 2.3 Backproagation algoritmus

Jedná se o algoritmus zpětného šíření, který umožňuje minimalizovat chybu a správně nastavit váhy v síti.

Pro použití backpropagation je nutné rozšířit síť, tak aby počítala chybovou funkci automaticky. To se udělá připojením všech $j$ výstupních uzlů na uzel vyhodnocující funkci $\frac{1}{2}(o_{ij} - t_{ij})^2$ kde $o_{ij}$ a $t_{ij}$ je $j$-tá komponenta výstupního vektoru. Tyto výstupy jsou sečteny a je vydán výstup $E_i$ pro každý vzor $t_i$. To vše je pak sečteno a je vydána finální kvadratická chyba $E$. Tato síť tedy umí vypočítat celkovou chybu pro danou trénovací množinu.

[1] Dneska už se zase používá ReLU nebo softplus (cca od 2010)

Jediné co v síti můžeme modifikovat jsou váhy. Chybu kterou dostváme označíme jako $E$. Jelikož $E$ se počítá čistě jako složení funkcí jednotlivých uzlů, můžeme využít gradientní sestup k její minimalizaci. $E$ je spojitá a diferencovatelná funkce $\ell$ vah $w_1, w_2, ....$

Výpočet gradientu nutný k minimalizaci vypadá následovně (vlastně stejné jako předtím u funkcí):

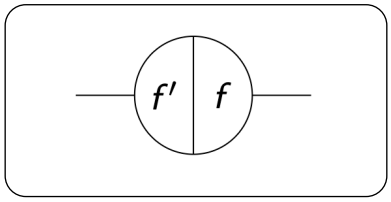
$$\nabla E = \left( \frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, ..., \frac{\partial E}{\partial w_l} \right)$$

Každá váha je upravena **inkrementem** $\delta w_i = -\gamma \frac{\partial E}{\partial w_i}$ kde $i = 1, ..., \ell$ a $\gamma$ je tzv. *učící konstanta* (udává něco jako velikost kroku).

Počet vstupních uzlů nemá vliv na fungování backpropagation. Funguje i při více korektně.

### 2.3.1 Fungování sítě

Funkce sítě je skládání funkcí, tedy řetězcové pravidlo. Každý uzel má teď 2 části. Grafický náčrt nazýváme jako *B-diagram.*

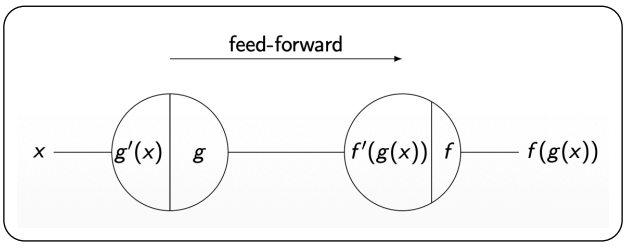



Obrázek 5: Uzel se 2 částmi

Vyhodnocování sítě probíhá ve 2 krocích:

1. **feed-forward krok**: informace (vstup x) jde zleva a každý uzel vyhodnotí $f$ i $f'$, výsledky se v něm uloží a $f$ se posílá dál
2. **backpropagation krok**: síť běží pozpátku a používají se uložené hodnoty, vstupem na pravé straně je 1, příchozí informace k uzlu je doplněna a výsledek je vynásoben hodnotou uloženou v levé části

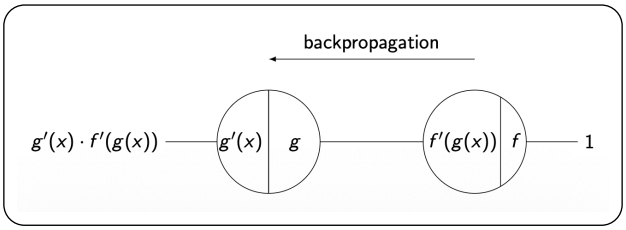
Výsledek který se shromáždí po backpropagation u vstupního uzlu je derivací síťové funkce $F$ vzhledem ke vstupu $x$.




Obrázek 6: Feed-forward krok v síti.

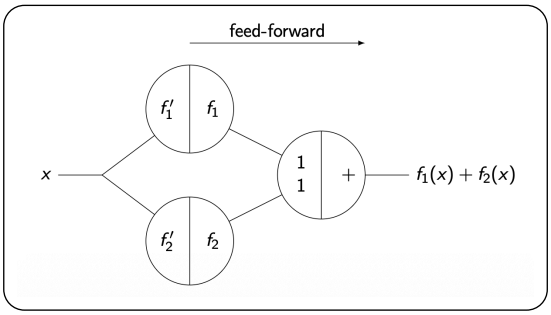
**Backproagation** **Theorem 7**

Backproagation správně počítá derivaci funkce sítě $F$ vzhledem ke vstupu $x$.

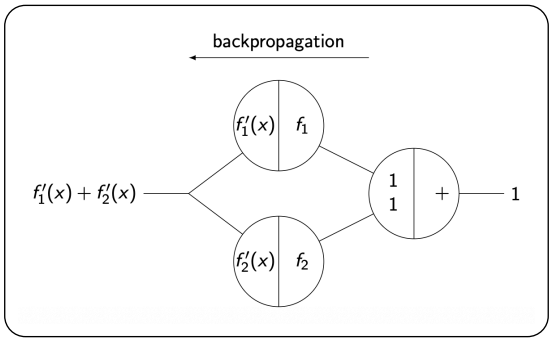



Obrázek 7: Backpropagation krok v síti.

V síti mohou nastat 3 případy navazování neuronů na sebe – **skládání funkcí** (1 neuron do 1), **sčítání funkcí** (2 neurony do 1) a **váhové hrany**. Práce s hodnotami v neuronech je přesně taková jako je název (sčítání = aplikujeme + a skládání = aplikujeme ·). U backpropagation je vstupem zprava 1.




Obrázek 8: Feed-forward sčítání funkcí.




Obrázek 9: Backproagation sčítání funkcí.

**Theorem 8**

Algoritmus backpropagation správně počítá derivaci funkce sítě $F$ vzhledem ke vstupu $x$. *[Ve slajdech je i neúplný důkaz.]*

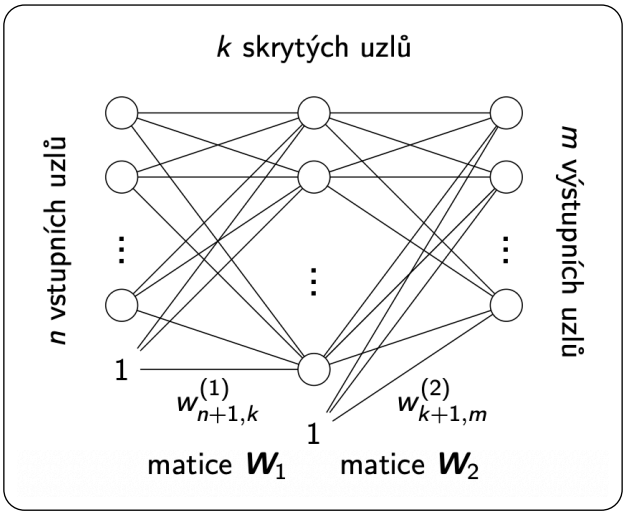
Algoritmus korektně funguje i pro více než 1 vstupní uzel (nezávislé proměnné). U 2 proměnných má síť 2 argumenty a my můžeme počítat parciální derivaci vzhledem k $x_1$ nebo $x_2$. Při backpropagation kroku se síť rozdělí na 2 podsítě, pro každou proměnnou zvlášť.

## 2.4 Vrstvené FFNN

**Vrstvená FFNN** **Definice 9**

Vrstvená FFNN je taková, kde jsou uzly uspořádány do skrytých vrstev $H_1, H_2, ..., H_k$.

Hrany ze všech vstupních uzlů vedou vždy do všech uzlů první skryté vrstvy $H_1$. Poté vše z $H_1$ vede do $H_2$. A z poslední vrstvy $H_k$ to tedy vede do výstupní vrstvy $O$ (jako output).




Obrázek 10: Síť se skrytými vrstvami.

,

### 2.4.1 Učení vrstvené FFN sítě

Po náhodném zvolení vah se použije backpropagation pro korekci. Algoritmus můžeme zapsat v těchto 4 krocích:

1. výpočet feed-forward (vzorce pro exitační hodnoty skrytých uzlů, dohromady tvoří matici)
2. backpropagation do výstupní vrstvy,
3. backpropagation do skryté vrstvy,
4. aktualizace vah.

Algoritmus skončí když je hodnota chybové funkce dostatečně nízká.

*[Tady by se dal ještě každý ten krok hodně rozepsat (viz prezentace 02, slajdy 38 a dále)]*

Pokud máme více než jeden tréninkový vzor používá se rozšířená síť pro samostatný výpočet chybové funkce pro každý z nich.

**Zjednodušení aproximace** **Poznámka 10**

Libovolnou funkci mohu aproximovat dostatečně velkou sítí s jedinou skrytou vrstvou.

## 2.5 Reziduální sítě (spojení)

Přístup k tvorbě velmi hlubokých NN, které nemají problém mizejících gradientů. Typicky se v hlubokých NN učí novou reprezentaci vrstvy $i$ nahrazením reprezentace ve vrstvě $i-1$. Tím pádem se každá vrstva musí naučit dělat něco užitečného nebo alespoň brát užitečné informace.

$$o^{(i)} = f\big(o^{(i-1)}\big) = g^{(i)}\big(W^{(i)} o^{(i-1)}\big)$$

Myšlenkou je že nová vrstva by měla předchozí reprezentaci jen narušit.

$$o^{(i)} = g_r^{(i)}\big(o^{(i-1)} + f\big(o^{(i-1)}\big)\big)$$

kde $f$ je rezidium (změna), která narušuje standardní chování přechodu $i-1$ na $i$. Pro výpočet rezidia se používá funkce, kterou je NN s jednou nelineární vrstvou kombinovaná s lineární vrstvou. Tímto dosáhneme, že reziduální sítě umožní spolehlivě učit výrazně hlubší NN.

$$f(o) = Vg(\mathrm{W}o)$$

kde $V$ a $W$ jsou naučené váhové matice.

Pokud by $V = 0$ pak by prostě reziduál $f$ zmizel a vlastně by se jen nic nedělal a výstup by se předal dál jako by vrstva neexistovala.

# 3 Týden 03 - pokračování o trénování

06. října 2025

## 3.1 Kódování vstupů

Obvykle je to přímočaré (takže 0 a 1). Většinou to jsou totiž vektory hodnot atributů. Odpovídá to prostě booleovským hodnotám.

Pokud se jedná o numerické atributy, tak se nechají v původní podobě, ale vhodně se upraví – škálování do rozsahu (třeba mezi 0 a 1), standardizace na nulový průměr, …. Pokud rostou exponenciálně namapují se na logaritmickou škálu.

Takže třeba obrázky (o rozměru $X \times Y$) vnímáme jako pixely, což jsou prostě vektory složené z 3XY celočíselných atributů, kde každý pixel má tři hodnoty, protože RGB.

**Kategorická data** Poznámka 11

**Kategorická data** nevyjadřují číselnou hodnotu, ale příslušnost ke kategorii (např. barva očí).

Mapování kategorických data na číselné hodnoty používá metodu *one-hot*. Každá kategorie se zakóduje jako binární vektor, ve kterém je jedna hodnota rovna 1 a ostatní 0. Problém by mohl nastat při velkém počtu kategorií, jelikož by se stal neefektivní (tisíce a více unikátních hodnot). V tomto případě musíme použít jinou techniku, třeba *embedding*.

## 3.2 Ztrátové funkce

Při odvození gradientů jsem použili funkci s kvadratickou chybou viz Kapitola 2.2 , což není jediná možnost. Obvykle je lepší výstup interpretovat jako pravděpodobnost, např.

$$\begin{gathered} \mathrm{p}(\text{„negative“}) = 0.1 \\ \mathrm{p}(\text{„neutral“}) = 0.2 \\ \mathrm{p}(\text{„positive“}) = 0.7 \end{gathered}$$

Chceme vysokou pravděpodobnost u správné třídy logicky.

**Negativní logaritmus pravděpodobnosti správné třídy**:

$$\text{Loss} = -\log(p(\ \ \text{correct\_class}\ \ ))$$

$$\text{Loss} = -\log P_w(t_j \mid x_j)$$

$$\text{Loss} = -\sum_{j=1}^{N} \log P_w(t_j \mid x_j) \text{ ... pro } N \text{ příkladů}$$

### 3.2.1 Křížová entropie (cross-entropy)

Značená jako $H(P, Q)$. Jde o druh míry odlišnosti mezi 2 rozděleními pravděpodobnosti $P$ a $Q$ (tedy skutečnou a modelem odhadovanou). Používá se tam kde model rozděluje data do tříd. Takže říká jak moc se model mýlí při určování (čím nižší tím lepší).

$$H(P, Q) = -\sum_{z} P(z) \log Q(z)$$

*Není to vzdálenost, nebot'* $H(P, P) = H(P) \neq 0$.

## 3.3 Softmax vrstva

Převádí surové skóre (logity, značené $x_i$) na pravděpodobnosti tříd. Takže pro $d$ tříd potřebujeme $d$ výstupních uzlů, kde na každém máme pravděpodobnost pro danou třídu. Dohromady na všech $d$ uzlech je pak součet 1. Tuto pravděpodobnost pro třídu $i$ značíme $y_i$.

$$y_i = \frac{e^{x_i}}{\sum_{j=1}^{d} e^{x_j}}$$

Softmax je v podstatě zobecnění sigmoidy, protože ta převádí reálné číslo na hodnotu mezi 0 a 1 a používáme ji u binární klasifikace (2 třídy). *Pro $d = 2$ by se softamx zredukoval na sigmoidu.*

*[Souvislost softmax a max]*

## 3.4 Problém mizejících gradientů

Skryté vrstvy v NN obvykle používají méně aktivačních funkcí než se používá ve výstupních vrstvách. U ReLU a softplus se z pozorování věří, že pomáhají řešit problém mizejících gradientů.

Při trénování hlubokých NN (s mnoha vrstvami) se může stát, že gradienty jsou příliš malé, což způsobí zpomalení nebo zastavení učení v hlubších vrstvách. Proč? Při backpropagation gradienty klesají pro hluboké vrstvy exponenciálně (derivace aktivačních funkcí jsou mneší než 1). Což způsobí, že váhy v těchto vrstvách se neaktualizují efektivně. To se projeví na neschopnosti sítě zachytit složitější vzory.

## 3.5 Problém explodujících gradientů

Prostě gradienty naopak rostou moc agresivně a aktualizace vah se stává nekontrolovatelnou. Nastává v hlubokých nebo rekurentních NN.

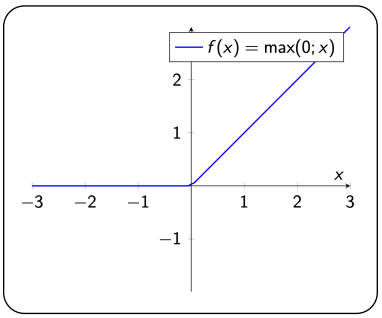
Data dělíme na trénovací a testovací množinu (slouží přesně k tomu, co je v názvu). Testovací se snaží odhadnout generalizační chybu. Může vzniknout problém, že se na testovacích datech začnou objevovat veliké chyby, protože model ztrácí schopnost generalizace (tzv. **early stopping point**). Tam bychom to měli ukončit.

Hyperparametr … cokoliv co je potřeba nastavit při vzniku sítě

**ReLU – Rectified linear unitDefinice 12**

Používá se od 2010 jako aktivační funkce.

$$\Phi(x) = \begin{cases} 0 \text{ pokud } x \leq 0 \\ x \text{ pokud } x > 0 \end{cases} = \max(0, x)$$




Obrázek 11: Graf ReLU funkce.

**Poznámka 13**

Aktivační funkce musí být nelineární. Jinak bychom mohli modelovat jen lineární funkce

**Věta o univerzální aproximaci** **Theorem 14**

Neuronová síť s jednou skrytou vrstvou může aproximovat jakoukoli spojitou funkci na kompaktním množině s libovolnou přesností, pokud má dostatečný počet neuronů.

Další možnost je funkce **softplus**, což je hladká aproximace ReLU.

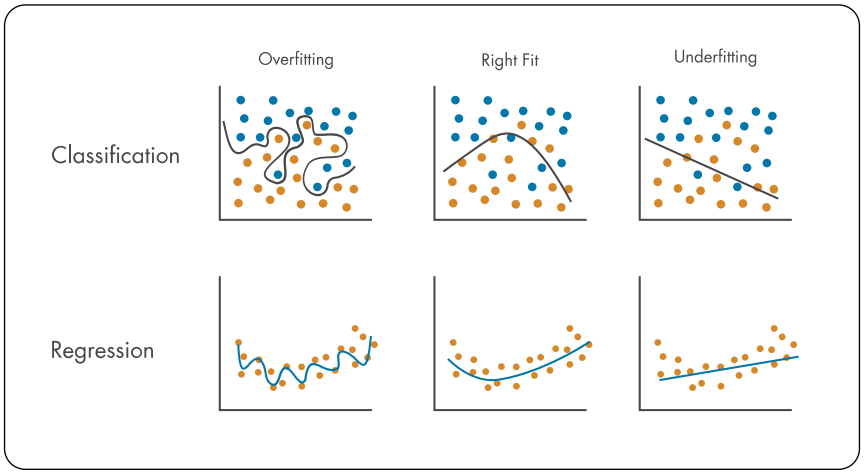
**Leaky ReLU** je modifikací ReLU, která umožňuje malé, ale nenulové gradienty i pro záporné vstupy. (*Prostě ten graf jde od středu doleva hodně mírně do záporných hodnot*).

Další pak: swish, Exponential Linear Unit.

## 3.6 Generalizace vs Memorizace

**Generalizace** … schopnost modelu se učit a aplikovat naučené věci na neviděná data. Model musí správně predikovat i na jiných než trénovacích datech.

**Memorizace** … model se učí přesně trénovací data bez snahy najít obecné vzory. Na trénovacích datech nízká chyba, na neviděných selhává. Bývá důsledkem přetrénování modelu (overfittingu).




Obrázek 12: Znázornění overfitting, underfitting a right fit.

## 3.7 Rychlé trénování (optimalizace gradientního sestupu)

Proč? Klasický gradientní sestup je pomalý, citlivý na volbu learning rate, riziko uvíznutí v lokálním minimu.

Myšlenka pochází z fyziky a využití setrvačnosti pohybu (*momentum*) => využijeme aktuální gradient i ten z předchozího kroku.

**Momentum** **Definice 15**

Vzorec pro momentum (vektor rychlosti a aktualizace vah):

$$v_t = \alpha\ v_{t-1} - \eta \nabla E(w_{t-1})$$

$$w_t = w_{t-1} + v_t$$

$\eta$ … learning rate, $\alpha$ … parametr momenta (obvykle 0.9), $\nabla E$ je gradient,

Nevýhodou je, že musíme ladit 2 parametry.

### 3.7.1 Nesterov Accelerated Gradient (NAG)

Modifikace momenta. Gradient se nepočítá v aktuálním bodě, ale v bodě předpovězeném momentem. Lepší směrová informace a rychlejší sestup (rychlejší reakce na změnu).

$$v_t = \alpha\ v_{t-1} - \eta \nabla E(w_{t-1} + \alpha\ v_{t-1})$$

$$w_t = w_{t-1} + v_t$$

*[K tomuto jsou různé ilustrace v prezentacích* **`lec03.pdf`** *okolo slajdu 40]*

### 3.7.2 Aktivní výběr trénovacích dat

Nemusíme trénovací vzory používat rovnoměrně, ale používáme, ty které přináší největší chybu. V praxi se používá spíše doplňkově.

## 3.8 Adaptivní algoritmy pro learning rate

Je to technika, kdy každá váha dostává vlastní learning rate. Protože jedno globální $\eta$ může vést k malým nebo naopak moc velkým změnám v různých směrech.

### 3.8.1 Newtonova metoda

Slouží k hledání minima chybové funkce $E(w)$. Bere v úvahu i zakřivení označované jako *Hessian* (je to matice druhých derivací; rozdíl oproti *gradient descent*). To dá rychlejší konvergenci a méně oscilací. Ale je nutný jeho výpočet, který trvá $O(n^3)$, což je nepraktické pro velké NN. Metoda se používá se jen v některých úpravách (ne čistá).

### 3.8.2 Silva & Almeida's algoritmus

Slouží ke zrychlení učení bez výpočtu Hessianu (zakřivení funkce, tzn. jak se mění gradient). Každá váha $w_i$ má vlastní learning rate $\eta_i$ a sleduje se její gradient $g_i^t = \frac{\partial E}{\partial w_i}$. Jednodušší než Newtonova metoda, ale je podobná.

**Idea**: využívá informace o druhé derivaci (což je zakřivení). Gradient je stabilní $\Rightarrow$ krok lze zvětšit; nebo gradient mění znaménko $\Rightarrow$ krok se zmenší.

$$\eta_i^{t+1} = \begin{cases} n_i^t \cdot u & \text{pokud } g_i^t \cdot g_i^{t-1} > 0 \\ n_i^t \cdot d & \text{pokud } g_i^t \cdot g_i^{t-1} < 0 \\ n_i^t & \text{jinak} \end{cases}$$

$u > 1$ … zvětšení kroku
$d < 1$ … zmenšení kroku

### 3.8.3 Delta-bar-delat (DBD)

Má za cíl adaptovat váhovou učící rychlost $\gamma$ s menšími oscilacemi než Silva & Almeida. Pokud se znaménko gradientu pro váhu dlouhodobě nemění, znamená to, že jdeme správným směrem $\Rightarrow$ zvýšíme learning rate ($\gamma = \gamma + u$, kde $u > 0$). Pokud by došlo k neshodě rázně se krok zmenší ($\gamma = \gamma \cdot d$, kde $0 < d < 1$).

Přidává nový hyperparametr $\theta$ tzv. hladicí konstanta (nutnost zvolit), která říká jak velkou váhu přiřazujeme gradientům z minulosti.

$$\overline{g}_i = (1 - \theta) \cdot g_i + \theta \cdot \overline{g}_{i-1}$$

Je pro ni uveden i pseudokód v prezentacích (slajd 55 v **`lec03`**).

### 3.8.4 Resilent backpropagation (Rprop)

Ignoruje velikost gradientu a řídí se pouze jeho znaménkem (velikost posunu určí právě adaptivní krok). Změna váhy je pevná, tedy nezávisí na velikosti gradientu. Stejné znaménko jako předtím ⇒ zvýšíme krok $\Delta_i$, opačné znaménko ⇒ snížíme krok a pokud je gradient = 0, tak s váhou nehýbeme. Pokud krokem přeskočíme minimum, využijeme *rollback*, který se vrátí před poslední update vah a aktuální gradient se nastaví na 0. Je odolný vůči mizejícím i explodujícím gradientům (kvůli ignoraci velikosti).

$$\Delta w_i^{(k)} = \begin{cases} -\Delta_i & \text{pokud } g_i^{(k)} > 0 \\ +\Delta_i & \text{pokud } g_i^{(k)} < 0 \\ 0 & \text{jinak} \end{cases}$$

### 3.8.5 AdaGrad

Upravuje learning rate na základě toho jak často se mění gradient (řídí se tedy jeho historií) zvlášť pro každý parametr. Vhodný pro řídká data (např. NLP). Akumulace $s$ (suma čtverců historických gradientů)

$$g_t = \nabla_w E_t(w_t)$$

$$s_t = s_{t-1} + g_t \odot g_t$$

a ta se pak použije v adaptivní úpravě

$$w_{t+1} = w_t - \eta \frac{g_t}{\sqrt{s_t + \varepsilon}}$$

kde $\odot$ je násobení po souřadnicích, $\varepsilon$ slouží pro stabilitu (obvykle velmi velmi malé) a $g$ jsou gradienty vzorku.

Kroky jsou kratší tam kde je historicky velká derivace (strmé místo), naopak jsou delší tam kde je derivace nízká (křivka plochá nebo neaktivní).

Má více variant – Diagonální AdaGrad, Plno-maticový AdaGrad. U nestacionárních úloh[2] může učení zvadnout (přestane se učit), protože $s_t$ pořád roste a tím pádem $\frac{\eta}{\sqrt{s_t}}$ stále klesá.

### 3.8.6 RMSProp

Nástupce AdaGrad. On místo kumulace používá exponenciální klouzavý průměr (něco jako pomalé zapomínání starých gradientů)

$$s_t = \rho s_{t-1} + (1 - \rho) g_t \odot g_t$$

kde $\rho$ je decay rate (okolo 0.9). Algoritmus díky tomu zůstává aktivní i po dlouhém čase. Přidává ale nový hyperparametr $\rho$.

### 3.8.7 Adam

Spojení RMSProp a momenta. Tím pádem kombinuje směr i měřítko. Dobře funguje už defaultně nastavený. Často používaný v deep learningu. Má dost hyperparametrů. Má 2 momenty:

- $m_t$ momentum (průměr) – vyhladí krátkodobý šum v gradientu
- $v_t$ momentum (rozptyl) – jako RMSProp

tedy škáluje kroky podle variability gradientu.

[2]Nestacionární úloha je taková, kde se pravidla hry mění v průběhu času.

$$m_t = \beta_1 m_{t-1} + (1 - \mathrm{B}_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \mathrm{B}_2) g_t \odot g_t$$

kde $\beta$ jsou hyperparametry určující rychlost exponenciálního zapomínání.

# 4 Týden 04 - Rekurentní neuronové sítě

13. října 2025

Jsou to dohromady 4 prezentace, otázka co z toho přesně vybrat.

1. **Prezentace ze dne AI** – podle mě ta není tak podstatná
2. **Rekurentní NN**
3. **LLM pro poučené uživatele**
4. **Transformer**

## 4.1 Rekurentní NN (RNN)

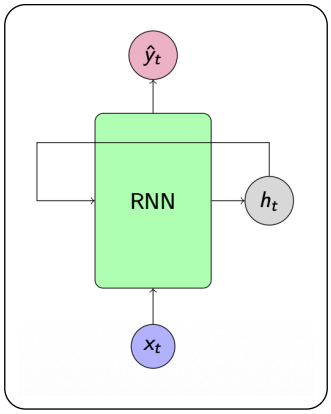
Poměrně dost dat má sekvenční povahu (čili pořadí hraje roli) – text, zvuk, časová řada, video. Na těchto datech běžné NN selhávají. RNN mají tu výhodu, že můžou modelovat závislosti mezi sekvencemi a mají schopnost pamatovat si minulost (skryté stavy $h_t$).

$$h_t = f(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t)$$

kde $x_t$ je vstup na časovém kroku $t$ a $h_{t-1}$ je skrytý stav předchozího kroku.

Používají se při strojovém překladu, rozpoznání řeči, generování textu, …

Mají **opakovanou strukturu** (tedy obsahují smyčky), což umožňuje si informace ukládat a tím pádem ta informace přetrvá v čase. Představa viz obrázek.




Obrázek 13: RNN se skrytým stavem.

### 4.1.1 Backproagationt through time (BPTT)

Aktualizace vah musí proběhnout přes časové kroky (RNN jsou model s pamětí).

Postup:

1. Výpočet feed-forward přes celou časovou sekvenci
2. Rozložení chyby v čase a zpětná propagace na každý časový krok
3. Aktualizace vah na základě gradientů chyby kumulované z různých časových kroků

Opět mohou nastat klasické problémy jako mizející nebo explodující gradienty. Řeší se pomocí pokročilejších architektur LSTM (long short-term memory) a GRU (Gated Reccurent Units) je zjednodušený LSTM.

### 4.1.2 Long Short-Term Memory (LSTM)

**LSTM** je speciální typ RNN, který dokáže uchovávat informace po velmi dlouhou dobu. Jde o speciální paměťovou buňku. Buňka má několik bran (gates) – forget gate (co se má zapomenout), input gate (které nové informace se uloží) a output gate (co z buňky se použije jako výstup). *[Je k tomu poměrně komplikovaný obrázek v prezentaci]*. Vhodnější pro složitější závislosti.

**Hadamardův součin** **Definice 16**

Prvkový součin 2 matic (vektorů) o stejné velikosti. Násobí se vždy sobě si odpovídající prvky. $(A \times B)_{i,j} = A_{i,j} \cdot B_{i,j}$

Používá se k aplikaci bran na jednotlivé složky vektorů. Tím se nezávislé aktualizují jednotlivé složky vektorů.

Důležitá je u něj asi i derivace.

**Forget gate** $f_t$ rozhoduje co se má zapomenout. Jde o vektor hodnot 0 až 1. Pokud se $f_t$ blíží nule informace jsou téměř zapomenuty.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

**Input gate** rozhoduje o přidání nových informací. Má dvě složky $i_t$, která určuje jaké hodnoty se aktualizují a $\tilde{c}_t$, což jsou nové potencionálně uložitelné informace.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

**Output gate** rozhoduje co bude na výstupu v daném časovém kroku. Kde $o_t$ rozhoduje o výstupu a $h_t$ je skrytý stav pro další časový krok.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t \times \tanh(C_t)$$

**Aktualizace paměti** pro celý LSTM pak probíhá následovně

$$c_t = f_t \times c_{t-1} + i_t \times \tilde{c}_t$$

kdy $f_t$ určuje kolik staré paměti se zapomene a $i_t$ kolik nové se přidá ($c$ jsou tedy informace).

### 4.1.3 Grated Recurrent Unit (GRU)

Oproti LSTM má méně parametrů a pouze 2 brány (reset a update). Použijeme pokud máme omezený výkon, ale potřebujeme efektivní práci s dlouhodobými znalostmi. Skrytý stav a paměť je spojena do jednoho vektoru.

**Skrytý stav** $h_t$ je kombinací nového a starého stavu.

**Reset Gate** $r_t$ rozhoduje jak moc se má předchozí skrytý stav zapomenout.

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$

**Update Gate** $z_t$ naopak říká jak moc se má skrytý stav aktualizovat o nové informace.

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$

**Aktualizace skrytého stavu**:

$$h_t = z_t \cdot h_{t-1} + (1 - z_t) \cdot \tilde{h_t}$$

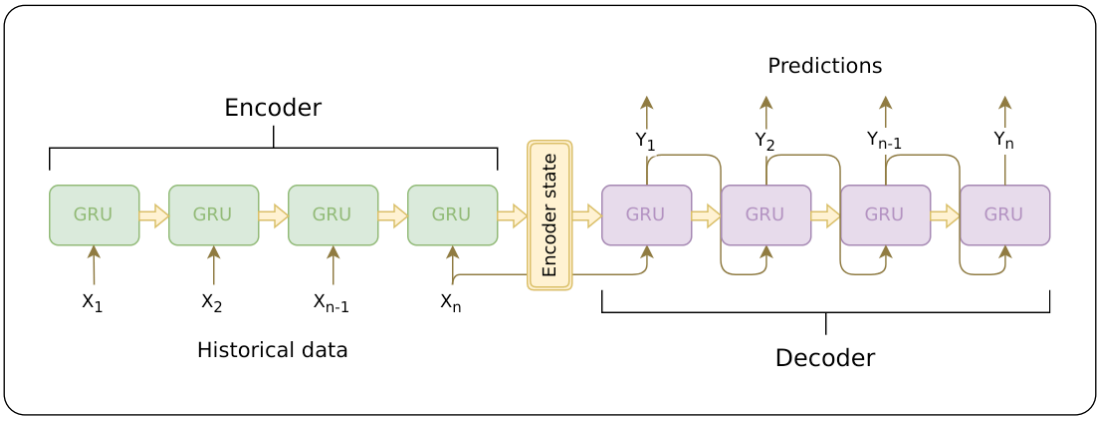
Starý stav se vždy kombinuje s novým kandidátním stavem $\tilde{h_t}$.

## 4.2 Bidirectional RNN (BiRNN)

Oproti běžným RNN se zde zpracovávají data oběma směry (budoucnost → minulost i minulost → budoucnost). Někdy mohou i informace z budoucnsoti ovlivnit současný časový krok (třeba zpracování přirozeného jazyka). Funguje prostě tak, že je rozdělená na 2 samostatné modely Forward RNN (zpracování od $t_1$ po $t_n$) a Backward RNN (zpracování od $t_n$ až po $t_1$). Výstupy se následně spojí a poskytnou úplný kontext $h_t = \left[\vec{h_t}, \overleftarrow{h_t}\right]$.

## 4.3 Seq2Seq modely

Zabývají se problémy, kde se má převést jedna sekvence na druhou (překlad, shrnutí textu). Takže se musí model naučit mapovat vstupní sekvenci na výstupní. Obvykle využívá dvě RNN (*encoder* – zpracuje vstupní sekvenci a *decoder* – generuje výstup na základě enkodéru).




Obrázek 14: Architektura seq2seq modelu.

Postup se dá rozdělit do 3 kroků:

1. Vstupní sekvence je zpracována encodérem – $x_1, x_2, ..., x_n$ na $h_1, h_2, ..., h_n$
2. Kód je poslední skrytý stav $h_n$ reprezentující celou vstupní sekvenci
3. Dekodér generuje výstup $y_1, y_2, ..., y_m$. Generování výstupu je postupné, protože každý krok závisí na předchozím výstupu.

Při dlouhých sekvencích ztrácí jednoduchý seq2seq model informace, protože se vše komprimuje do jednoho fixního vektoru. Mohou se také akumulovat chyby kvůli postupné návaznosti na vše předchozí. **Attention mechanismus** umožní modelu se zaměřit na různé části vstupní sekvence při generování každého výstupního kroku.

## 4.4 Atention mechanismy

Při každém kroku generování výstupu model zváží důležitost každého prvku ve vstupní sekvenci. Váhy (*attention*) $\alpha$ jsou určeny na základě podobnosti skrytých stavů decoderu $s_t$ a encoderu $h_i$. Výstup je pak vážená kombinace všech skrytých stavů encoderu.

$$c_t = \sum_{i=1}^{n} \alpha_{t,i} \cdot h_i$$

Podobnost mezi akrytými stavy $s_t$ a $h_i$ odhaduje tzv. **score function**

$$\text{score}(s_t, h_i) = s_t^T W h_i$$

Normalizace probíhá pomocí softmax, aby se jednalo o pravděpodobnostní rozdělení

$$\alpha_{t,i} = \frac{e^{\text{score}(s_t, h_i)}}{\sum_{k=1}^{n} e^{\text{score}(s_t, h_k)}}$$

Podle typu score function, pak nazýváme celý mechanismus. Tyto mechanismy se liší způsobem výpočtu skóre mezi dotazem (Query $Q$) a klíčem (Key $K$).

**Query** Q je dotaz reprezentující aktuální prvek v dekodéru, kterým chceme najít relevantní část vstupní sekvence. Každý prvek vytváří svůj vlastní dotaz $Q$.

**Key** $K$ je vektor reprezentující každý prvek ve vstupní sekvenc a popisujícíc jeho charakteristiky. Dotaz $Q$ se porovná s klíči $K$, aby určil jaký prvek je pro dotaz důležitý.

**Value** $V$ je vektor nesoucí skutečnou informaci, kterou chceme použít pro výstup. Výsledek v daném časovém kroku je průměr hodnot, kde váhy určuje dotaz a klíč.

- **Dot-product Attention:**

$$\text{score} = s_t^T \cdot h_i$$

- **Bilinear Attention**:

$$\text{score} = s_t^T \cdot W h_i$$

- **MLP (multi-layer perceptron) Attention**:

$$\text{score} = s_a^T \tanh(W_a[s_t; h_i])$$

### 4.4.1 Bahdanau attention mechanismus

Byl představen v článku. Architektura zahrnuje BiRNN (skryté stavy z obou směrů se spojí pro lepší zachycení kontextu). Attention skóre se počítá pomocí vícevrstvého perceptronu (MLP). Attention mechanismus se aplikuje mezi jednotlivými kroky dekodéru.

### 4.4.2 Luong attention mechanismus

Opět představen v článku. Používá jednodušší jednosměrnou RNN, attention skóre se vypočítává pomocí bilineární funkce (založena na skalárním součinu stavu dekodéru a enkodéru). Attention se aplikuje po každém kroku dekodéru a stav $s_t$ se používá k výpočtu výstupu $c_t$.

### 4.4.3 Multi-head attention mechanismus

Rozšíření standardního attention, které se využívá v transformerech. Provádí se několik výpočtů paralelně. Kdy každý výpočet (hlava) má vlastní váhy pro dotazy, klíče a hodnoty (tím umí zachytit různé aspekty).

Matice $W^O$ slouží ke shrnutí všech hlav do jediného výstupu (ten pokračuje do dalších vrstev modelu).

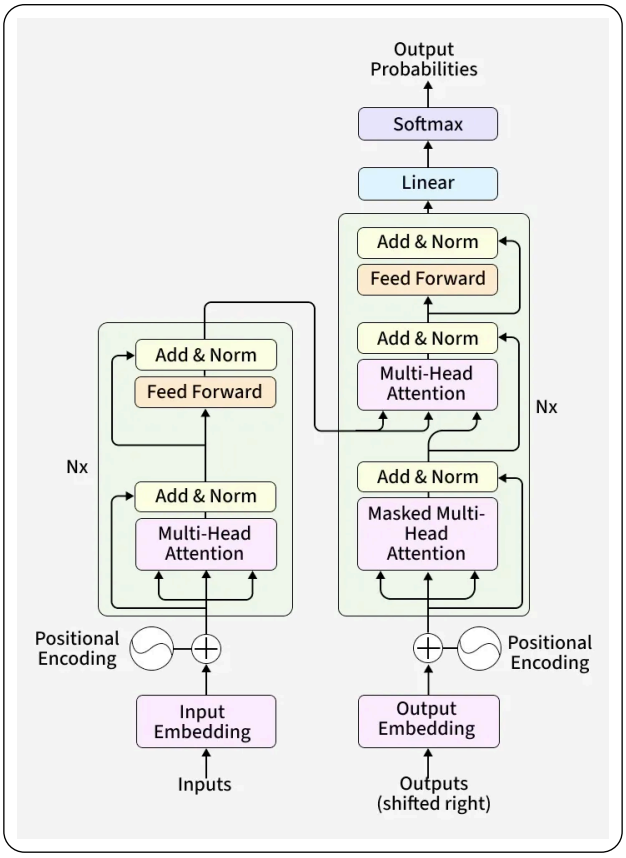
Výpočet probíhá pomocí škálovaného součtu podobností:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

kde $d_k$ je dimenze klíče $K$. Díky tomu může každý prvek v dekodéru věnovat větší pozornost relevantním částem vstupu na základě vypočtených vah.

# 5 Týden 05 - Transformer

Jedná se o model pro strojový překlad. Skládá se ze 2 hlavních komponent – *enkodéru* a *dekodéru*. Oproti RNN vidí celou sekvenci najednou, nikoliv jako zpracování jedné části po druhé.




Obrázek 15: Architektura Transformer modelu.

## 5.1 Tokenizace

Proces při kterém se text dělí na menší části tzv. *tokeny* (to může celé slovo, část slova *subword* nebo jen znak). Převod na tento formát má umožňuje textu porozumět modelům na zpracování přirozeného jazyka (NLP).

## 5.2 Word embedding (WE)

Reprezentace slov jako hustých vektorů v nízkoúrovňovém prostoru. Zachytí sémantická a syntaktické vztahy. Podobná slova ⇒ podobné vektory. Zlepší výkon NLP oproti třeba one-hot encoding. Tady v té oblasti se angažoval Tomáš Mikolov.

### 5.2.1 Word2Vec

Naučit vektory slov podle kontextu v textu. Sémantický vztah bere jako vektorovou aritmetiku (`king - man + woman ≈ queen`). Má nízkou dimenzionalitou (100-300 cca). Trénováno na velkých korpusech textu. Modely jako BERT nebo GPT jsou na tomto postaveny.

Někdy se používají 2 embeddingy. Třeba když se jedná o překlad a jsou rozdílné slovníky pro vstup a výstup. Nevýhodou je vyšší počet parametrů nebo nepřenositelnost naučených reprezentací mezi vstupem a výstupem.

**Poznámka 17**

GPT má sdílený emdebbing.

### 5.2.2 Skip-Gram

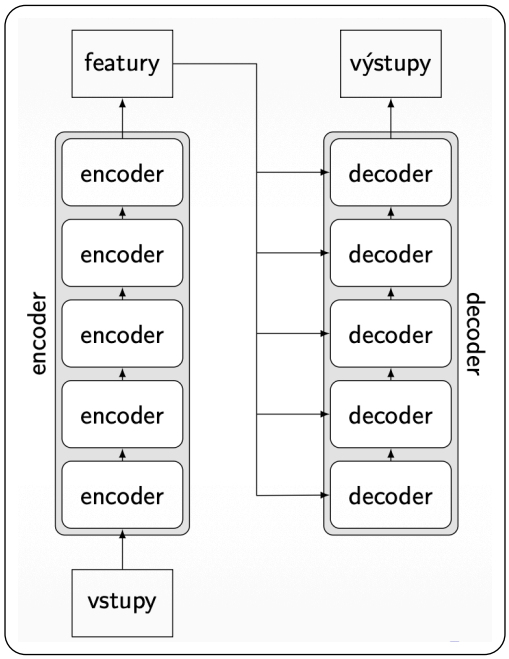
Předpovídá kontextová slova na základě daného slova.

### 5.2.3 Continous Bag of Word (CBOW)

Předpovídá dané slovo na základě jeho kontextu.

## 5.3 Paralelní zpracování v Transfomeru

Vstup do enkodéru je zpracován celý najednou. Lze to snadno paralelizovat na GPU, umožňuje delší kontext a má rychlejší trénink. Enkodér i dekodér se skládá z více identických bloků, jimiž prochází vstupní věta (poslední pak tvoří výstup).




Obrázek 16: Postup enkodéru a dekodéru v Transformeru.

## 5.4 Poziční kódování (PE)

Transfomer používá u enkodér-dekodéru mechanismy pozornosti bez rekurence. Pokud by se ale používal pouze attention mohlo by dojít k přiřazení stejné sémantiky různým větám (např. „*Tom bite a dog.*“ a „*A dog bite Tom*“). Proto se přidalo poziční kódování.

Kontatencace vektorů umožňuje modelu vidět poziční kódování nezávisle na word embeddingu. Poziční kódování nevyžaduje stejnou dimenzi jako word embedding. Často se používá funkce sin nebo cos.

**Pozice** (*pos*) je číslo 0 až maximálně definovaný počet tokenů ve větě. Pokud je max lenght = 128 a $d_{\text{model}} = 512$ pak pro každou dvojici prvků embeddingu

$$\frac{\text{pos}}{(10\ 000)^{\frac{2i}{512}}}$$

pro $i = 0...\frac{d_{\text{model}}}{2} - 1 = 255$

$$\mathrm{PE}(\mathrm{pos}, 2) = \sin\left(\frac{\mathrm{pos}}{10000^{\frac{2}{512}}}\right)$$

$$\mathrm{PE}(\mathrm{pos}, 3) = \cos\left(\frac{\mathrm{pos}}{10000^{\frac{2}{512}}}\right)$$

$$\mathrm{PE}(\mathrm{pos}, 4) = \sin\left(\frac{\mathrm{pos}}{10000^{\frac{0}{512}}}\right)$$

$$\mathrm{PE}(\mathrm{pos}, 5) = \cos\left(\frac{\mathrm{pos}}{10000^{\frac{0}{512}}}\right)$$

$$\vdots$$

Může se přecházet na relativní pozice (což je lineární operace). Toho se využívá při attention mechanismech, kde model musí určit jak daleko od sebe jsou jednotlivé pozice (pro korektní přiřazení vah).

$$\sin(a+b) = \sin(a)\cos(b) + \cos(a)\sin(b)$$

$$\cos(a+b) = \cos(a)\cos(b) - \sin(a)\sin(b)$$

## 5.5 Self Attention

Má zvážit vztah mezi všemi slovy ve větě a vyrtvořit kontextově závislé reprezentace.

Klíče (keys) $K$, dotazy (queries) $Q$, hodnoty (values) $V$. Do těchto 3 vektorů je transformováno každé slovo pomocí lineárních projekcí.

# 6 Týden 06 - Konvoluční neuronové sítě (CCN)

27. října 2025

## 6.1 Co jsou konvoluční sítě?

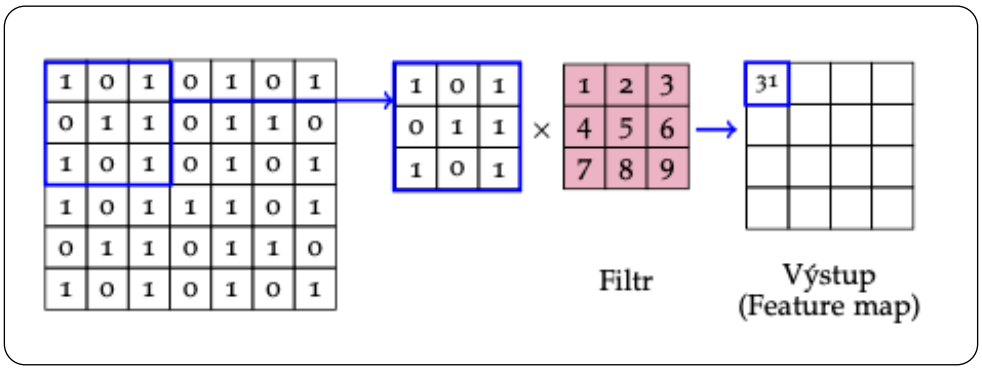
Speciální typ hluboké (čili více než 1 skrytá vrstva) neuronové sítě. Měly by efektivně zpracovávat data s **prostorovou strukturou** (videa, obrázky, časové řady, …). Měly by z takových dat extrahovat vlastnosti (hierarchické rysy) a ušetřit to mít mnoho vstupních parametrů.

Z obrázků by se totiž neměl dělat lineární vektor, který se použije ve FFNN (problémy: vysoký počet parametrů, paměťová náročnost, riziko přeučení, ignorují se lokální vlastnosti).

## 6.2 Složení CCN

### 6.2.1 Konvoluce

Konvoluce je matermatická operace. Posouvá malé jádro po celém vstupu. Jádrem je malá matice (třeba $3 \times 3$). Ta se učí detekovat lokální rys – hrana, textura, roh.




Obrázek 17: Konvoluční filtr.

Filtr udělá s vybranou částí skalární součin a výstupem je *feature map*.

**Hlavní hyperparametry** které musíme určit – velikost filtru $F$ (rozměr jádra), stride $S$ (krok posunutí), padding $P$ (výplň okrajů nulami), počet filtrů $K$ (hloubka výstupu).

**Výpočetní výstupní dimenze** ($W_{\text{out}}$) jsou nové rozměry po konvoluci (předpokládáme čtvercové matice):

$$W_{\text{out}} = \frac{W_{\text{in}} - F + 2P}{S} + 1$$

**Detaily derivace** v konvoluci: *není až tak podstatné, spíše vědět přehledově z prezentace.*

1. Výpočet gradientu vah
2. Propagace gradientu vstupu

### 6.2.2 Pooling vrstva

Slouží k provedení redukci dimenze (*downsampling*). Takže výrazně sníží prostorovou velikost a pomáhá s translační tolerancí (schopnost rozpoznat objekt bez ohledu na jeho posunutí). Vrstva se neučí (nemá žádné váhy). Existují 2 typy – *max pooling* (častější) a *average pooling*.

Má 2 hyperparametry:

- velikost okna (pool size) $F$
- krok (stride) $S$

**Max pooling**. U max poolingu se při FF v každém okně vybírá maximální hodnota (něco jako „Byl daný rys přítomen v této oblasti?“), takže dobře dokáže detekovat daný rys a odfiltrovat šum. Při backpropagation se chová jako přepínač – z vyšší vrstvy se šíří pouze do té pozice, které nesla nejvyšší hodnotu, jinde je 0 gradient.

**Average pooling**. U average poolingu se při FF vypočítá průměr a dochází spíše k vyhlazení dané oblasti. Používá se převážně v posledních vrstvách. Při backpropagation se chová jako distributor – čili gradient se rovnoměrně rozdělí přes všechny pozice (např. při okně 2×2 každá pozice obdrží $\frac{1}{4}$ gradientu).

**Detaily derivace** v pooling vrstvách: *TBA*.

## 6.3 Architektura LeNet-5

První komplentní implementace architektury CNN. Měla 5 učitelných vrstev. subsampling = pooling

## 6.4 Architektura AlexNet

První hluboká CCN (8 vrstev) ve větším měřítku. Používali ReLu (výrazné zrychlení). Použití *Dropout* pro prevenci přeučení.

## 6.5 Architektura ResNet

ResNet zavedl tzv. residual blocks (ty umožňují zkratku/přeskočení některých vrstev), který umožnil satbilní trénink sítí se stovkami vrstev.

## 6.6 Technika Dropout

Brání přeučení. Regularizační technika pro FFNN. Při každé iteraci tréninku (jak feed-forward tak backpropagation) se náhodně vypne určité procento neuronů (obvykle 50 %). Tím pádem se síť „nemůže spolehnout“ na určitou skupinu neuronů. Síť pak funguje jako trénink mnoha menších řídkých sítí sdílejících váhy.

> **Clever Hans efekt** **Poznámka 18**
>
> Nastane pokud se model naučí chybnou korelaci místo skutečného rysu. Např. vlci a psi husky – vstupem byli fotky vlků na sněhu, pokud síť dostala obrázek huskyho na sněhu označila ho chybně za vlka, protože rozlišovala podle pozadí (tedy sněhu).

## 6.7 Architektura GoogleNet

GoogleNet používá sady různě velkých filtrů, nikoliv jen jeden. Velmi hluboká síť s menším počtem parametrů. Kromě zvyšování hloubky sítě přistoupili i k jejímu rozšíření do šířky. Přidali *Inception modul*, který provádí všechny konvoluce paralelně najednou. Využili *global average pooling* k rapidnímu snížení parametrů. Už to nebyl jen lineární stoh vrstev, ale složitější graf.

## 6.8 Vizualizace a interpretace

### 6.8.1 Saliency map (mapa důležitosti)

Má určit citlivost výstupního skóre $S_c$ vzhledem ke změně vstupního pixelu $X$. Matematickým základem je analýza citlivosti – gradient skóre cílové třídy $c$.

$$M = | \frac{\partial S_c(X)}{\partial X} |$$

Vizualiazce pomocí heatmap gradientů (vyšší hodnota = důležitější). Velmi rychlá metoda, která odhalí zkreslení. Prostě jako teplotní mapa překryje obrázek.

Výpočeet gradientu $\frac{\partial S_c}{\partial X}$ se získá provedením backpropagation s jedničkovým gradientem na výstupu cílové třídy $c$.

1. Dopředný průchod vypočítá $S_c$
2. Nastaví se gradient $\frac{\partial S_c}{\partial S_c}$ na 1.0
3. Gradient se zpětně šíří až na výstupní vrstvu $X$
4. Vizualizace pomocí matice (mapy) $M$

### 6.8.2 Grad-CAM

Oproti saliency map vytváří „hrubší“ bitmapu s vysokou lokalizací ukazující jaké regiony vstupu ovlivnili finální rozhodnutí. Využívá výstup z poslední konvoluční vrstvy.

## 6.9 Povídání o cvičení

V prezentacích něco jak pracovat s Pythonem a těmi nejznámějšími knihovnami. Nevím proč je to zmíněný až v půlce semestru.

# 7 Týden 07 - Generativní a diskriminační modely

03. listopadu 2025

**Generativní modely** generují nové příklady na základě vzorů ve vstupních datech. Výzvou je model naučit efektivní zachycení vzorů na původních datech. Na vstupních datech se naučí rozdělení a podle toho vytváří vzory, které tomu rozdělení odpovídají.

**Diskriminační modely** rozlišují mezi třídami (kategoriemi) v datech.

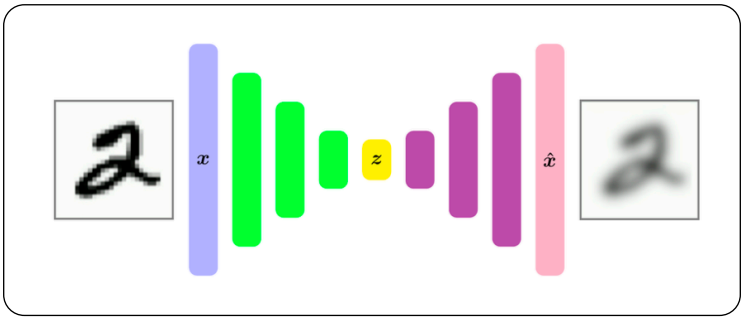
## 7.1 Autoencodery (AE)

Jedná se o neuronovou síť, která se učí komprimovat vstupní data do nižší reprezentace a poté je konstruovat zpět. Cílem je naučit model kódovat klíčová informace a odstranit redundantní data. AE mají několik použití – redukce dimenzionality (komprimace na menší počet rysů, např. pro předzpracování), odstranění šumu či generování nových dat (např. VAE mohou generovat nové realistické vzorky dat). Jsou to takový ty příklady co jsme dělali na AKTI, kde se generovali číslice na obrázku asi $16 \times 16$ pixelů.

**AE není vhodný jako generátor**. Potřebujeme VAE (Variational Autoencoder), který má latentní prostor plynulý a smysluplně uspořádaný. Nevhodnost je z několika důvodů:

1. Netrénuje latentní prostor, aby byl spojitý nebo homogenní (takže tam vznikají prádná místa)
2. Dekodér často může generovat šum, protože nemá naučenou distribuci dat v prostoru

Dělí se na 2 části – *enkodér* (převádí vstup do komprimované reprezentace) a *dekodér* (rekonstruuje původní vstupní data z oné komprimované reprezentace). Enkodér počítá deterministickou funkci $q_\Phi(z|x)$, která zobrazí vstupní data $x$ na jediný latentní vektor $z$. Dekodér zase naopak počítá funkci $p_\theta(x|z)$, která rekonstruuje původní vstupní prostor z latentního vektoru $z$




Obrázek 18: Enkodér a dekodér v AE.

Úzká skrytá vrstva $z$ uprostřed nutí NN naučit se malou latentní reprezentaci.

**Latentní reprezentace** **Poznámka 19**

Komprimované reprezentace se také nazývá jako **latentní**.

**Poznámka 20**

Autoencoder … automatická kódování data („auto“ jako „self“).

Cílem tréninku je minimalizovat rozdíl mezi původním vstupem a rekonstruovaným výstupem. Trénovací množina je složená z $p$ dvojic $\langle x_i, t_i \rangle$, kde $x_i \in \mathbb{R}^n$ a $t_i \in \mathbb{R}^m$. Ještě je tam nějaká chybová funkce.

Používá se při redukci dimenzionality (komprimace dat třeba pro předzpracování), odstranění šumu (např. rekonstrukce „rozbitých“ dat).

## 7.2 Variable Autoencoder (VAE)

Oproti AE přidá pravděpodobnostní inferenci a má stochastickou reprezentaci. Mapování není tak přímočaté jako u AE, ale vstupní data se převádí na střední hodnotu $\mu$ a odchylku $\sigma$. Výběr pak není deterministikcý, ale náhodný podle speciálního rozdělení.

Výpočet má 3 kroky:

1. funkce enkodéru – pro vstupní data se spočítají parametry latentní reprezentace (střední hodnota a směrodatná odchylka) a na jejich základě se definuje distribuce, ze které se vzorkuje
2. vzorkování (sampling) – vybere se vzorek
3. funkce dekodéru – dekodér bere vybranou latentní proměnnou a snaží se rekonstruovat původní vstupní data

Problémem může být prostřední krok vzorkování, protože náhodností neumožňuje hledat gradient (není tam derivace). Tím pádem nemůžeme optimalizovat váhy pomocí backpropagation. Řešení je použít externí šum $\varepsilon \sim \mathcal{N}(0, 1)$ a výpočet pak je $z = \mu + \sigma \odot \varepsilon$ ($\odot$ je Hadarmadův součin). Takže náhodnost je pak mimo enkodér a operace jsou deterministické.

## 7.3 Generative Adversarial Networks (GAN)

Zde se nové instance generují přímo vzorkováním. Je tvořen 2 neuronovými sítěmi (generátor a diskriminátor), které se trénují **navzájem** pomocí konkurečního učení. Každá síť má jiný úkol.

#### Generátor

Vytváří falešná data z šumu, která se co nejvíce podobají reálným (např. obrázky nebo text). Začíná se náhodným vektorem z latentního prostoru z normálního rozdělení.

#### Diskriminátor

Má za úkol označit jestli se jedná o pravá data nebo falešná data (vytvořená generátorem).

Tyto 2 části mezi sebou soupeří, jeden se snaží o co nejvíce realistiká data a druhý se snaží je co nejlépe odhalit (tzv. hra s nulovým součtem).

Proces trénování je iterativní (opakuje se dokud se nedosáhne požadované úrovně) a probíhá následovně:

1. Aktualizace parametrů diskriminátoru.
2. Diskriminátor dostane data pravá i falešná (obojí označena). Na těchto datech se učí maximalizovat svou spárvnou předpověď.
3. Po aktualizaci diskriminátoru se jeho nastavení zmrazí a aktualizuje se generátor.
4. Generátor dostane náhodný šumový vektor, na jehož základě tvoří co nejvíce realistická data.
5. Ztrátová funkce generátoru je nastavená tak, aby minimalizovala rozpoznání diskriminátorem.

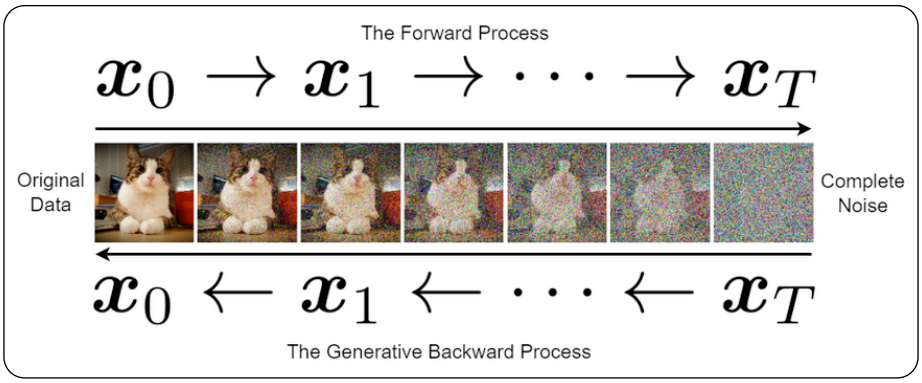
Při trénování mohou nastat problémy – nestabilita (oscialace výsledků $\Rightarrow$ nikdy se nedosáhne požadované úrovně) nebo *mode collapse* (generátor netvoří rozmanité výstupy).

**Poznámka 21**

Tento přístup generuje i „obrazy“ a „umění“. Dokonce se něco z toho i prodalo v reálné aukci.

## 7.4 Generativní modely řízené šumem (difuzní modely)

Jsou stabilnější alternativou k předchozím zmíněným modelům. Idea přístupu má 2 stochastické procesy – **dopředná difuze** přidávající Gaussův šum k čistému obrázku dokud z něj není jen šum a **zpětná difuze** trénující síť odebírat šum a získat původní obrázek.




Obrázek 19: Postup difuzního modelu.

Dopředný proces je fixní a nemá učitelné parametry. V časových krocích se prostě přidává dané množství šumu.

Zpětná difuze je jádrem tohoto přístupu (obvykle architektura U-Net). Vstupem je zašuměný obrázek $x_t$ a informace o čase $t$. Výstup sítě má predikovat přidaný šum v danémn kroku. Chybovou funkcí je Means Squared Error (MSE) mezi skutečně přidaným šumem a predikovaným šumem.

### 7.4.1 Architektura U-Net

Konvoluční síť, která má tvar písmene U – úlohy kde vstup i výstup musí mít stejné rozlišení.

*[Tady asi dopsat další větší podrobnosti z konce 6. prezentace.]*

# 8 Týden 08 - Asociativní sítě

10. listopadu 2025

**Univerzální aproximační schopnost** **Theorem 22**

Nechť $f : \mathbb{R}^n \to \mathbb{R}$ je spojitá funkce definovaná na kompantní podmnožině $X$ množiny $\mathbb{R}^n$. Pak pro každé $\varepsilon > 0$ existuje vícevrstevná NN s jednou skrytou vrstvou obsahující konečný počet neuronů.

Tzn. kdykoliv je chování systému popsatelné spojitou funkcí na kompaktní množině, lze ho libovolně popsat NN.

## 8.1 Asociativní síť (ANN)

Inspirováno lidskou pamětí. Jsou schopné se učit a následně si vybavit vzory na základě částečných/zkreslených vstupů. Jsou 2 typy **autoasociativní sítě** vybavující si celý vzor z jeho části a **heteroasocitivní sítě** asociující různé vzory mezi sebou.

### 8.1.1 Hopfieldova síť

Jde o autoasociativní síť. Umožňuje ukládat vzory a následně je vybavit z neúplných (poškozených) verzí. Má tzv. *energetickou funkci*. Síť se sanží dosáhnout stavu s minimální energií, která odpovídá následujícímu vzoru

$$E = -\frac{1}{2} \cdot \sum_{j=1}^{n} \sum_{i=1}^{n} y_i w_{ij} y_j$$

Jde o jednovrstevnou síť a všechny neurony jsou vzájemně propojeny (kromě smyček sami na sebe). Všechny váhy jsou symetrické ($w_{ij} = w_{ji}$). Neurony mají obvykle diskrétní výstup (třeba 0 a 1).

Trénink sítě se neprovádí pomocí backpropagation (takže je dost odlišný). Používá se Hebbovo pravidlo pro nastavení vah. Konkrétně:

1. Inicializuje se váhová matice ($w_{ij} = w_{ji}$) a diagonální prvky jsou 0.
2. Pro každý vzor upravíme váhy podle Hebbova pravidla (2 neurony aktivní současně $\Rightarrow$ váha mezi nimi by měla být posílena).
3. Normalizace nebo škálování váh pro předejití nestabilitě sítě. Bežný přístup je dělení celkové váhy každého spojení počtem neuronů v síti.

Mohou být 2 základní režimy aktualiazce neuronů – asynchronní (neurony se katualizují jeden po druhém v náhodném pořadí) a synchronní aktualiazce (aktualizace všech současně, něco jako paralelní zpracování).

Kapacita sítě je velmi omezená – může si efektivně ukládat přibližně 15 % vzorů z počtu neuronů (tzn. mám 100 neuronů, tak můžu uložit zhruba 15 vzorů). Obecně se tedy nikde v praxi nyní nevyužívá.

Využívalo se to pro hledání přijetelných řešení TSP. Použila se reprezentace maticí. Poměrně rozsáhlé a ve slajdech 7a rozepsané.

## 8.2 Samoorganizující se NN (SONN)

NN využívající kompetitivní učení. Je dána množina $T$ vzorů a parametr $m$, což je počet reprezentatnů a výstupních neuronů. Každý reprezentant je dán vektorem $w_j$ (*codebook word*). Jsou-li určeny $w$ a vstup $x$, tak je vstupu $x$ přiřazen reprezentant $c$, který je mu nejblíže (podle $c = \arg\min_{j=1}^{n} \| x - w_j \|$). Cílem je najít $m$ reprezentantů, pro které je

$$E = \frac{1}{|P|} \cdot \sum_{p \in P} \| x^p - w_c \|^2$$

minimální.

### 8.2.1 Kompetitivní učení

Neurony mezi sebou soutěží o výstup. Obvykle se řídí pravidlem *winner takes all* (takže se vždy aktivuje jen 1 neuron). Toto způsobí „specializaci“ neuronů na různé signály. K tomuto učení není potřeba externí označení nebo klasifikace vstupů.

### 8.2.2 Lloydův algoritmus učení (k-means shlukování)

Základní algoritmus pro shlukovou analýzu. Využívá kompetitivní učení bez učitele. Pracuje v iteracích.

1. **Inicializace reprezentatnů** – probíhá náhodně, vybere se $k$ bodů, které považujeme za počáteční středy
$$T_r = \left\{ p \in P \mid r = \arg\min_{j=1}^{n} \| x^p \| \right\} - w_j$$
2. **Přiřazení vstupích bodů** – každý vstupní bod se přiřadí nejbližšímu reprezentantu
3. **Aktualizace reprezentantů** – váhové vektory se aktualizují tak, aby lépe reprezentovaly přiřazené vstupní body, obvykle se používá průměr (těžiště) všechn bodů přiřezených k tomuto reprezentantu
$$w_r = \frac{1}{\mid T_r \mid} \sum_{x \in T_r} x$$

Krok 2. a 3. se opakují dokud nedojde ke konvergenci (algoritmus se ustáli a nedochází k dalším změnám při opakovaných výpočtech) reprezentantů.

### 8.2.3 Kohenovo učení a mapy

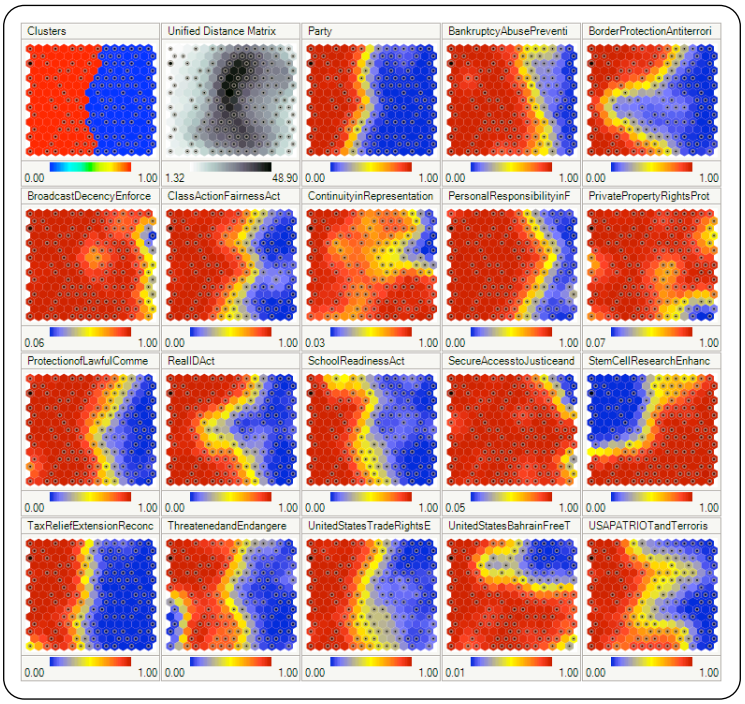
Aktualizace se provádí po každém vstupním vzoru (rychlejší a flexibilnější adaptace). Pro každý vstupní vektor $x$ se identifikuje vítězný neuron jehož váhový vektor $w_c$ je mu nejblíže podle Euklidovské vzdálenosti. Aktualizace váhového vektrou vítězného neuronů sousedů se provádí podle:

$$w_{j(t+1)} = \begin{cases} w_{j(t)} + \theta(t,i,j) \cdot \big(x - w_j(t)\big) & \text{pokud } w_j \in N_\delta(i) \text{ je vítěz} \\ w_{j(t)} & \text{jinak} \end{cases}$$

kde $\theta(t)$ je rychlost učení (v čase se zmenšuje).

**Kohenovy (samoorganizující se) mapy** zavádí topologickou strukturu reprezentantů (2D mřížka). Redukuje dimenzionalitu a přináší vizualizaci. Můžeme se na mapa dívat pomocí U--Matrix, která dává světlé (neurony si jsou velmi podobné = shluk) a tmavé (data se prudce

mění) hodnoty. Topologickou strukturu můžeme vytvořit třeba vhodnou vzáleností $d$ neuronů (a k nim brát i okolí).



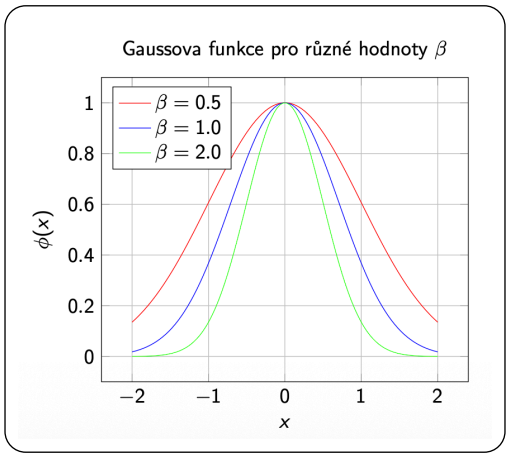

Obrázek 20: Kohenovy mapy.

### 8.2.4 RBF (radial basis function) sítě

Skládají se ze 3 vrstev – vstupní, skrytá a výstupní. Jako aktivační funkce používají radiální bázové funkce (proto ten název). Snaží se prostor pokrýt oblastmi vlivu (narozdíl od MLP, kde se dělí přímkou).

Obvykle se jedná o Gaussovu funkci:

$$\phi(x) = e^{-\beta \| x - c \|^2}$$




Obrázek 21: Gaussova funkce.

Učení takové sítě zahrnuje: určení středů RBF ve skryté vrstvě, nastavení vah mezi skrytou a výstupní vrstvou. K určení středů se používá buď k-means clustering nebo náhodný výběr.

## 8.3 Fuzzy regulátory

**Řídící systém** je soubor komponent, který řídí chování jiného systému nebo zařízení. Jeho hlavním úkolem je dosažení nebo udržení požadovaného stavu systému (např. výkon). Příkladem může být seqay robot, který reaguje na náklon (signály) a pomocí fuzzy logiky říká motorům jak mají regulovat výkon, aby seqway držel rovnováhu a jel určeným směrem. Touto konkrétní části se budeme zabývat – *controller* a ten pro nás bude **fuzzy regulátor**.

Fuzzy regulátory využívají fuzzy logiku. Takže můžeme pracovat s neurčitostí a přibližnými hodnotami.

**Fuzzy množina** **Definice 23**

Standardní fuzzy množina v universu U je zobrazení

$$A : U \to [0, 1]$$

$A(u)$ je stupeň příslušnosti prvku $u$ do množiny $A$.

**Moderní fuzzy** **Definice 24**

Nechť $L = \langle L, \leq \rangle$ je částečné uspořádáná množina. Fuzzy množina v universu $U$ je zobrazení

$$A : U \to L$$

kde $a \in L$ jsou stupně, $A(u)$ stupeň přílušnosti (čím větší, tím více tam prvek patří).

Podle toho jakou hodnotu má $A(u)$ tak tak moc tam $u$ patří (0 je nejménně a 1 nejvíce).

S takovýmito L-množinami můžeme provádět množinové operace jako standardně (zakreslíme to v grafu těch 2 „funkcí“).

**T-norma** **Definice 25**

T-norma na úplném svazu $\langle L, \leq \rangle$ je binární operace $\otimes$ na $L$, která je asociativní, komutativní, monotónni a má 1 jako neutrální prvek, t.j.

$$a \otimes (b \otimes c) = (a \otimes b) \otimes c$$

$$a \otimes b = b \otimes a$$

$$a \leq b \Rightarrow a \otimes c \leq b \otimes c$$

$$a \otimes 1 = a$$

Máme 3 nejdůležitější T-normy:

1. **Lukasiewiczova**

$$a \otimes b = \max(a + b - 1, 0)$$

2. **Gödelova**

$$a \otimes b = \min(a, b)$$

3. **Goguenova** (součinová)

$$a \otimes b = a \cdot b$$

**Lingvistická proměnná**
Jde o proměnnou, která nabývá lingvistických hodnot (prostě slova). Ke každé lingvistické hodnotě $t$ je přiřazena fuzzy množina $A_t$, která mapuje lingvistické hodnoty na číselné (fuzzy hodnoty).

### 8.3.1 Báze pravidel

Obsahuje pravidla typu **`if-then`**, které řídí rozhodovací proces. KAždé pravidlo říká do dělat v jaké situaci. Jsou formulována takto vstup = lingvistická proměnná a výstup = akce, která má být provedena. Např. **`if`** $x_1$ **`is`** $t_{1,1}$ **`and`** $x_2$ … **`then`** $y$ **`is`** $o_1$ ($x$ a $y$ jsou lingvistické proměnné a $t$, $o$ lingivistické termy).

Každá pravidlo se dá reprezentovat jako relace.

### 8.3.2 Fuzzy inferenční mechanismus

*Compositional Rule Inference (CRI)* je technika používaná v tzv. systémech založených na fuzzy pravidlech, základních součástech fuzzy regulátorů. Ve fuzzy logice je toto pravidlo považováno za pravidlo odvození.

**Definice 26**

Nechť $R$ je $L$-relace mezi $X$ a $Y$ , $A$ je $L$-množina v $X$. $L$-množina $B$ získaná z $A$ a $R$ pomocí CRI je definována jako

$$B(y) = \bigvee_{x \in X} A(x) \otimes R(x, y)$$

neboli $B = A \circ R$.

$B(y)$ je stupeň pravdivosti výroku.

### 8.3.3 Defuzzifikace

Jde o proces vytváření kvantifikovatelného výsledku v crisp logice (klasická logika), jsou-li dány fuzzy množiny.

**Těžiště (center of gravity nebo centroid)** je metoda, která vypočítá „těžiště“ fuzzy množiny, které se pak dá jako výstupní výsledek.

### 8.3.4 Takagi-Sugenův model

Každé pravidlo zde vede k výstupu, který je lineární funkcí všech vstupních proměnných , nikoliv k fuzzy množině. Takže tvar je: **`IF`** $x_1$ je $t_{1,1}$ **`and`** $x_2$ je $t_{1,2}$ **`THEN`** $y = f(x_1, x_2)$

**O univerzální aproximační schopnosti fuzzy regulátorů** **Theorem 27**

### 8.3.5 Mambdaniho model

Velmi populární model fuzzy inference. Umí pracovat s přiorozeným jayzkem jak na vstupu, tak na výstupu. Používá 2 metody, které ji odlišují *clipping* a *scaling*. Prostě v grafu se uřízne vršek pod nějakou hodnotou $\alpha$ a zůstane jen spodní část (lichoběžník), který se následně „stlačí“, aby si zachoval tvar, ale byl nížší. Clipping je oproti scalingu výrazně méně citlivější (dělá tupé tvary).

# 9 Týden 09 - Evoluční algoritmy a aplikace algoritmů v praxi

- schéma (délka schématu, hodnota schématu )
- křížení
- generace
- přežití křížení
- selekce

**Evoluční algoritmus** **Definice 28**

Třída optimalizačních metod inspirovaných principy biologické evoluce. Např. genetické algoritmy, evoluční strategie, genetické programování, diferenciální evoluce.

Nové nebo upravené vlastnosti jedince vznikají pomocí 3 mechanismů, které slouží i pro inspiraci tvorby algoritmů:

- mutace
- křížení
- přirozený výběr.

Můžeme to využít pro řešení optimalizačních problémů tím, že udržujeme populaci potenciálních řešení, na které iterativně provádíme evoluci dokud nedosáhneme požadovaného stavu.

## 9.1 Kroky a schéma evolučního algoritmu

1. Kódování kandidátních řešení. Populace je množinou kódů reprezentující tato řešení. Prakticky vše jde vždy převést na binární řetězec.
2. Počáteční populace (binární řetězce) se obvykle generuje náhodně.
3. Fitness funkce je funkce, která ohodnocuje daná řešení. Můžeme chtít neco maximalizovat nebo minimalizovat.
4. Genetické operátory provádí mutace či křížení.
5. Je nutné definovat ukončující podmínku, velikost populace, pravděpodobnost mutace, fungování křížení atd.

Tohle schéma jde zapsat nějakým pseudokódem nebo i graficky znázornit.

Příkladem může být hledání maxima funkce.

## 9.2 Tvorba nové generace

Děje se ve 3 krocích. Čím lepší řetězec, tím větší šance že půjde dál (logické).

**Reprodukce** – kopírování řetězců ze staré generace do nové. Třeba pomocí *vážené rulety* (ta ale nepřinese nic nového a můžeme mít duplicity).

**Křížení** – umožní vyměnit si informaci mezi řetězci. Pro křížení musíme mít pravděpodobnost s jakou proběhne. Má destruktivní vliv na schémata (závisí to hodně na pozicích). Delší schémata mají větší náchylnost ke ztrátě při křížení.

**Mutace** – obvykle má velmi nízkou pravděpodobnost. Zavádí genetickou variabilitu do populace. Funguje tak, že prostě změní bit v řetězci. Pravděpodobnost změny je $p_m$.

## 9.3 Schéma

Něco jako šablona která poposuje podmnožinu daného řetězce. Má stjenou délku jako ostatní řetězce. Umožní analýzu a sledování vlastností podmnožin.

**Příklad schématu** **Příklad 29**

Příklad schématu `**1*0010`. Řetězec `00100010` tam patří, `11100010` taky, ale `11001101` ne.

Pro schéma definujeme fitness v dané generaci. Je to průměr všech fitness řetězců v generaci, které odpovídají danému schématu. Tím můžeme vysledovat jak které skupiny přispívají ke zlepšení.

Může dojít k problému kdy některé schéma má velmi vysoké skóre, ale zároveň je velmi nepravděpodobné, že bychom z tohoto schématu získali ten nejlepší řetězec. Muselo by dojít k vhodné mutaci. Může dojít k problému tzv. *problém sousedních silných schémat*, kdy jsou dvě schémata velmi blízko sebe a mají vysoké skóre, ale jsou úplně opačná (např. `011*****` a `100*****`). Dalším problémem může být *problém křížení rovnocených schémat*. Generace dospěje do stavu kdy jsou řetězce odpovídající 2 schématům rovnoměrně zastoupeny (`011*****` a `100*****`). Pokud tyto 2 řetězce zkřížímě (zaměřujeme se na první 3 pozice) dojde k tomu, že potomci budou slabší. Z toho plyne, že křížení rovnocených schémat vede ke snížení kvality (zpomalí nebo zastaví pokrok k optimálnímu řešení).

**Určená pozice**: pozice ve schématu, kde je 0 nebo 1

**Délka schéamtu** $\delta(H)$: vzdálenost mezi první a poslední určenou pozicí

**Řád schématu** $o(H)$: počet určených pozic ve schématu

Řešíme otázku přežití schématu v generaci. Jak se bdue měnit počet řetězců během jednoho kroku. Máme schéma $H$, počet řetězců v dané generaci po t-tém kroku je $m(H,t)$ a populace po t-tém kroku $A(t)$. Snažíme se zjistit $m(H,t+1)$. Hlavními faktory pro ovlivnění je selekce, křížení a mutace.

**Pravděpodobnsot výběru řetězce** **Definice 30**

Je dána vztahem

$$p_i = \frac{f_i}{\sum f_i}$$

kde $f_i$ je kvalita $i$-tého řetězce a a suma je jejich součet.

**Očekávaný počet řetězců shodujících se se schématem $H$** **Definice 31**

Po reprodukci z generace $A(t)$ je tento počet dán jako:

$$m(H,t+1) = m(H,t) \cdot n \cdot \frac{f(H)}{\sum f_j}$$

kde $n$ je počet řetězců v generaci, $f(H)$ je průměrná kvalita řetězce v generaci $H$ a $f_j$ je součet kvalit.

Prostě jak reprodukce ovlivní zachování schématu.

**Průměrná kvalita řetězce** **Definice 32**

$$\overline{f} = \frac{\sum f_j}{n}$$

Pokud je kvalita nadprůměrná bude řetězců více a naopak ⇒ reprodukce odtraňuje postupně schémata s podprůměrnou kvalitou.

Můžeme spočítat i rychlost růstu schématu. Vzniká nám tím vlastně geometrická řada (což je exponenciální růst).

**Pravděpodobnost ztráty schématu při křížení** **Definice 33**

$$p_d \leq \frac{\delta(H)}{\ell - 1}$$

kde $\ell$ je délka řetězce. Pravděpodobnost že alespoň jeden potomek odpovídá schématu $H$ je $1 - p_d$.

Z předchozích poznatků plyne, že **schéma s malou délkou a vysokou kvalitou** mají největší šanci k přežití.

**Pravděpodobnost přežití schéamtu při mutaci** **Definice 34**

Pravděpodobnost že řetězec $A$ bude po mutaci stále odpovídat schématu $H$ je

$$p_s = (1 - p_m)^{o(H)}$$

**Věta o schématech** **Theorem 35**

Krátká, nadpůrměrná schémata s nízkým řádem (počet určených pozic) získávají exponenciální nárůst počtu řetězců v následujících generacích.

Schémata z předchozí věty se nazývají stavební bloky a jsou základem pro vytváření kvalitních řešení v genetických algoritmech. Mohlo by se jednat o schéma `*1*0*` (pokud by mělo nadprůměrnou kvalitu).

## 9.4 Grayův kód

Zajišťuje že 2 sousední body v prohledávaném prostoru se liší pouze na jediné pozici. To změnšuje riziko výrazných změn.

První bit je stejný jako bit binárního čísla. Další bit je XOR předchozího bitu binárního čísla a příslušného bitu binárního čísla.

**Převod čísla na Grayův kód** **Příklad 36**

Číslo `1011`. První bit bude 1. Druhý bit bude $1 \oplus 0 = 1$. Třetí bit bude $0 \oplus 1 = 1$. Čtvrtý bit bude $1 \oplus 1 = 0$. Výsledek je `1110`.

## 9.5 Elitářství

V evolučním algoritmu může dojít ke ztrátě nalezeného maxima (které může být globální). Elitářství dokáže ochránit nejelepší řetězec dané generace, že 100% přežije do další bez ohledu na výsledek rulety.

Prostě s určít $m$ nejlepších řetězců (obvykle se jedná o jediný) v dané generaci, které nejsou ovlivněny křížením ani mutací a automaticky přechází dále.

## 9.6 Podobná kvalita a fitness scaling (škálování)

Když jsou řetězce v okolí maxima a mají podobnou kvalitu, algoritmus ztratí schopnost hledat nejlepší řešení. Musíme více do podrobna rozlišit mezi řetězci mající podobnou kvalitu. To zajistí fitness scaling, který zvýrazní rozdíly mezi podobnými skóre u řetězců.

### Lineární škálování

Pro každý řetězec zajistíme přepočet podle vztahu:

$$\overline{f} = a + f \cdot b$$

$$\overline{f}_{\max} = C_{\text{mult}} \cdot f_{\text{avg}}$$

$C_{\text{mult}}$ je obvykle dáno něco jako 1.2 nebo 2.0.

$$a = f_{\text{avg}} \cdot \frac{f_{\max} - C_{\text{mult}} \cdot f_{\text{avg}}}{f_{\max} - f_{\text{avg}}}$$

$$b = \frac{(C_{\text{mult}} - 1) \cdot f_{\text{avg}}}{f_{\max} - f_{\text{avg}}}$$

Průměrná kvalita $f_{\text{avg}}$ by měla zůstat zachována, protože zajišťuje stabilitu pravděpodobnosti výběru řetězců. Takový řetězec má pravděpodobnost výběru $P = \frac{1}{n}$, kde $n$ je počet řetězců v generaci.

Po škálování můžeme některé kvality dostat záporné. To má 2 možnosti řešení – neškálovat, nebo upravit $a$ a $b$, tak aby nejmíň byla 0.

## 9.7 Problém 2-rukého ($k$-rukého) bandity

Slouží pro vysvětlení proč je exponenciáůních nárůst v oněch specifických schématech užitečný.

**Zadání problému**. Máme stroj se 2 pákami. Po zatažení za páku získáme průměrnou výhru ($m_1$ nebo $m_2$ – tyto hodnoty jsou neznámé). Máme $N$ pokusů. Cílem je navrhnout strategii, která maximulizuje celkovou výhru.

**Řešení**. Používá se tzv. *strategie průzkumu a exploatace*. Na začátku nevíme, která páka je lepší. Rozdělíme teda problém na 2 části, kde v první průzkumné pozorujeme, která páka dodá lepší výsledky a v druhé části se soustředíme na lepší páku, abychom maximalizovali zisk. Otázkou je jak najít optimální počet pokusů $\ell$, který věnujeme které části. Jde použít tento vzorec

$$N - n^* \approx \sqrt{8\pi b^4 \ln(N^2)} \cdot e^{\frac{n^*}{2b^2}}$$

kde $n^*$ je počet pokusů na horší páce, $b$ je parametr související s rozptylem výher páky, $\ln(N^2)$ je závilost na celkovém počtu $N$ pokusů.

### 9.7.1 Aplikace problému na genetické algoritmy

Genetický algoritmus jde brát jako mnoho problémů $k$-rukého bandity (jedno schéma = jedna páka). Průměrná výhra je jako kvalita schématu a počet tahů za páku je jako počet řetězců daného schématu. Je žádoucí dosáhnout exponenciálního nárůstu pokusu (u nejlepších pák), protože potom to odpovídá šíření kvalitních schémat.

# 10 Info ke zkoušce

- Povolení nahládnout do papírových materiálu na 2 minuty.
- Nemělo by se jednat o moc matiky.
- Asi né tak podrobné jako na přednášce, ale víc než jen úplně povrchově.

**Seznam témat ke zkoušce**

1. perceptron, funkce, geometrická interpretace, učení, separabilní problémy, problém XOR
2. FFNN, architektura, vrstvy, aktivační funkce, věta o aproximaci, přehledově trénink, kódování vstupů a výstupů
3. Backpropagation, v obecné architektuře, ve vrstvených sítích
4. Optimalizace gradientního sestupu: momentum, NAG, adaptivní LR, Silva&Almeida, DBD, Rprop, Adagrad, RMSProp, Adam (ne všechno zaráz, vyberu třeba dvě z nich.
5. Zpracování sekvencí: RNN, BP in time, LSTM, GRU
6. Attention mechanismy
7. Transformer, architektura, poziční kódování, residuální spojení
8. Konvoluční sítě, konvoluční vrstvy a jejich hyperparametry, saliency maps
9. Generativní NN: AE, VAE, GANs, difuzní modely
10. Fuzzy regulátory, báze pravidel, inference, Takagi-Sugeno model, Mambdaniho model.
11. Asociativní sítě, Hopfieldova síť, SONN, Kohonenovy mapy, RBF sítě
12. Genetické algoritmy