

Gymnázium Christiana Dopplera, Zborovská 45, Praha 5

ROČNÍKOVÁ PRÁCE  
**Regresní neuronové sítě**

Vypracoval: Lukáš Čaha  
Třída: 8.M  
Školní rok: 2017/2018  
Seminář: Seminář z programování

Prohlašuji, že jsem svou ročníkovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím s využíváním práce na Gymnáziu Christiana Dopplera pro studijní účely.

V Praze dne 20. února 2018

Lukáš Čaha

# Obsah

<b>1</b>	<b>Úvod</b>	<b>4</b>
<b>2</b>	<b>Základní pojmy</b>	<b>5</b>
2.1	Neuron . . . . .	5
2.1.1	Jádro . . . . .	5
2.1.2	Synapse . . . . .	6
2.2	Síť . . . . .	6
2.3	Pohyb dat . . . . .	7
2.3.1	Vstup . . . . .	7
2.3.2	Výstup . . . . .	7
<b>3</b>	<b>Vstup</b>	<b>9</b>
3.1	Design vstupní vrstvy . . . . .	9
3.1.1	Velikost vrstvy . . . . .	9
3.2	Scaling . . . . .	9
3.3	Data . . . . .	9
3.3.1	Trénovací . . . . .	9
3.3.2	Testovací . . . . .	10
<b>4</b>	<b>Forward-propagation</b>	<b>11</b>
4.1	Typy sítí . . . . .	11
4.1.1	Feed-forward . . . . .	11
4.1.2	Deep learning . . . . .	12
<b>5</b>	<b>Back-propagation</b>	<b>13</b>
5.1	Loss function . . . . .	13
5.2	Prakticky . . . . .	13
<b>6</b>	<b>Overfitting</b>	<b>15</b>
6.1	Řešení . . . . .	15
<b>7</b>	<b>Doporučení k programování</b>	<b>16</b>
7.1	Jazyk . . . . .	16
7.1.1	Vývojové prostředí . . . . .	16
7.1.2	Knihovny . . . . .	16
7.2	Git . . . . .	17

<b>8 Konvoluční předzpracování</b>	<b>18</b>
8.1 Konvoluce . . . . .	18
8.2 Sdružování . . . . .	18
8.3 Vložení do sítě . . . . .	19
<b>9 Závěr</b>	<b>20</b>
<b>Literatura</b>	<b>21</b>
<b>Přílohy</b>	<b>22</b>

# 1. Úvod

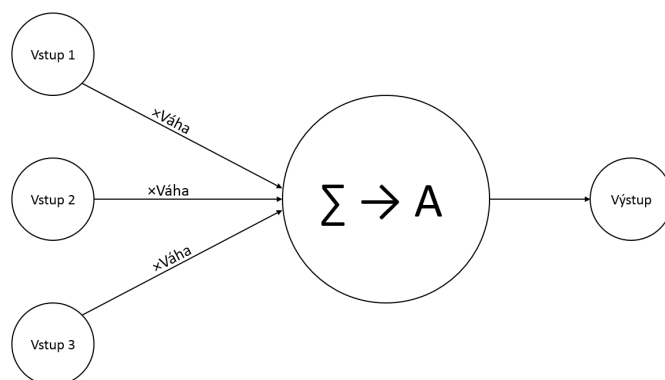
Ve světě se nachází mnoho dat v mnoha podobách a v dnešní době se dostáváme do bodu, kdy nestačíme všechny třídit a využívat na 100 %.

Neuronové sítě jsou vrcholem lidské práce v oblasti informačních technologií. Mohl bych je přirovnat k lidskému mozku. A důvodem proč je zde zmiňuji je právě jejich všestranost. Sítí můžeme pouštět dva typy dat. Jedak lidmi vyhodnocené, a poté nevyhodnocené u nichž budu chtít výsledek. Neuronové sítě se podle prvního typu dat naučí jaká je souvislost mezi vstupem a výstupem a pak můžou přibližně určit výstupy dat druhého typu. Pokud byla v první řadě síť správně designovaná můžeme očekávat výsledky s poměrně velikou přesností a rychlostí zpracování, na jakou jsem zvyklí u počítačů.

Touto prací bych chtěl rozebrat neuronové sítě na úroveň pochopitelnou i pro středoškoláky, kteří by chtěli začít se strojovým učením, což je obor zahrnující moderní způsoby práce s umělou inteligencí.

## 2. Základní pojmy

### 2.1 Neuron



Obrázek 2.1: Diagram neuronu

#### 2.1.1 Jádro

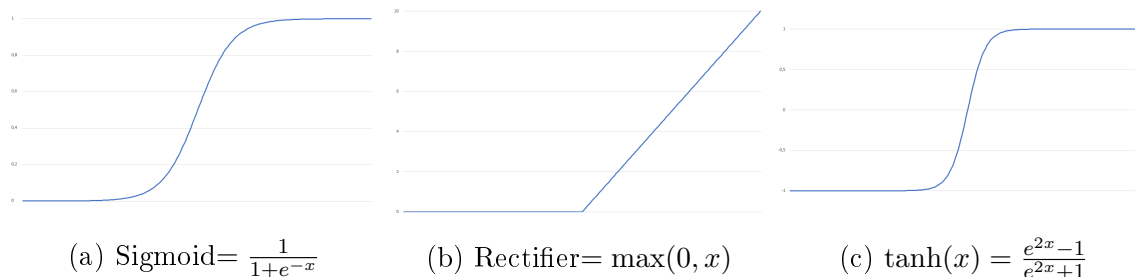
**Aktivace** je hodnota rozumně blízko nule ( $a = 0.73$ ). Tahle hodnota určuje míru zapnutosti neuronu. Více aktivované neurony můžou mít větší vliv na neurony v síti přímo následující. Aktivace neuronů jsou závislé na datech, takže není možné měnit jejich hodnoty přímo.

#### Aktivační funkce

Aktivační funkce upravuje příchozí signály, tak aby následně vytvořená hodnota zapadala řádů ostatních jednotek. Přijdou-li do neuronu 4 signály všechny s vysokou kladnou hodnotou, bude aktivace neuronu kladná a bude se blížit ostatním hodnotám vzniklým ze silných signálů.

**Signum** je nejzákladnější funkce, která může být pro tento úkol použita. Má všechny potřebné vlastnosti. Bohužel ztrácíme touto funkcí detail, který může být při tréninku sítě klíčový.

**Sigmoid** byla nejvíce používaná aktivační funkce na začátku vývoje neuronových sítí. Je také velice jednoduchá na pochopení. Dnes jí hlavně používáme pro určení pravděpodobností ve výstupní vrstvě, jelikož vrací pouze hodnoty v intervalu  $\langle 0, 1 \rangle$ .



Obrázek 2.2: Aktivační funkce

**Rectifier** je dnes velice oblíbená funkce. V odborné literatuře se můžeme setkat s názvem RELU (REctified Linear Unit). Její výhoda je, že opravdu silný vstup se nezmění na obyčejnou hodnotu jakou je například u Sigmoidu 1. Při použití této funkce se stává celý program jednoduše vypočitatelný jelikož bude použito jen porovnání, násobení a sčítání.

**Tanh** je jedna z málo funkcí, která vrací záporné hodnoty. Tento specifický znak se občas ukáže vhodný, ale pro nejzákladnější modely není důležité tuto funkci používat.

### 2.1.2 Synapse

**Synapse** je spojení mezi dvěma neurony. Toto spojení zajišťuje, že aktivace neuronu v síti je závislá na aktivacích předchozích neuronů.

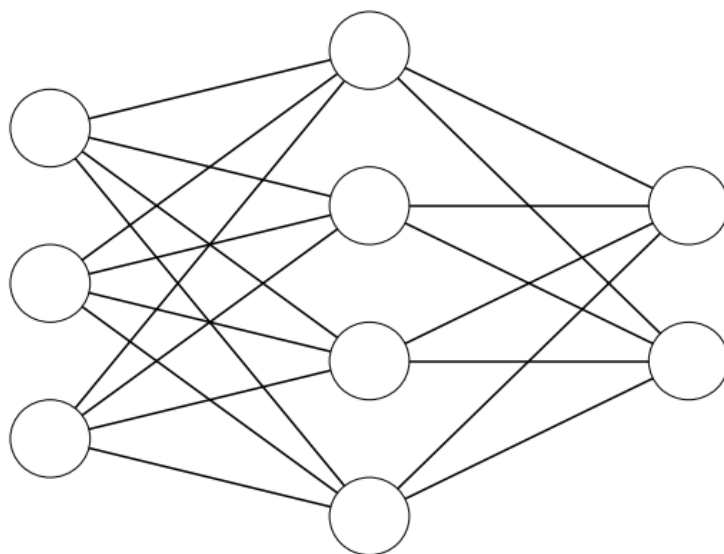
**Váha** ovlivňuje spoje mezi neurony. Váhy spojení tvoří dohromady povahu sítě. Z libovolných vstupních dat můžu upravováním síly spojení (synapsí) vyvodit libovolné výstupní data. Mezi libovolnými dvěma vrstvami je vždy plný počet synapsí a tedy i vah. Tento počet dostaneme vynásobením počtů neuronů v obou vrstvách.

## 2.2 Síť

Síť se skládá z několika vrstev, které jsou navzájem propojené.

**Vrstva** je několik neuronů, které se navzájem neovlivňují, ale jsou ovlivněny stejnými neurony a zároveň ovlivňují stejné neurony.

**Bias** je míra vlivu nezávislého na datech. Tato externí síla se stará o vyrovnaní sítě s menším počtem neuronů a tím uspoří výpočetní výkon. V překladu je bias šum, který rozostřuje data, aby se výsledná síť nepřizpůsobila až příliš moc trénovacím datům.



Obrázek 2.3: Diagram jednoduché sítě

## 2.3 Pohyb dat

### 2.3.1 Vstup

#### Typy vstupních dat

**Trénovací data** jsou data u nichž používáme vstupy i výstupy pro vylepšování sítě. Pokud výsledná síť uvidí znovu tato data bude na nich mít mnohem lepší výsledky, jelikož je trénovaná speciálně na tyto data a až jako vedlejší produkt je trénink na data podobná.

**Testovací data** je soubor vstupů, u nichž je známý i výsledek. Ten ale nikdy není ukázán síti, slouží totiž pro porovnání výsledku sítě s pravidlým výsledkem. Takto získává uživatel statistiky o kvalitě sítě.

**Produkční data** jsou důvod proč síť vůbec programujeme. Tyto data dostává síť během běžného používání a počítá k nim výsledky. Není však možnost určit jak by tyto výsledky měli vyjít, a proto nám už zbývá pouze doufat, že síť funguje jak popisuje teorie.

**Scaling** je metoda upravení hodnot z našich vstupních dat, tak aby v síti tato data vystupovala pouze jako aktivace. Dobrým příkladem je vstupní hodnota věk. V našich datech se vyskytuje člověk s maximální věkem 100 a minimálním 0. Odpovídající hodnoty aktivace potom budou  $100 \rightarrow 1.0$  a  $0 \rightarrow 0.0$ .

### 2.3.2 Výstup

**Back-scaling** je forma získání dat zpět z neuronové sítě. Pokud zrovna trénujeme, není nutné data získávat a pak je porovnávat s očekávanými výsledky, lepší způsob je očekávané



výsledky převést na jazyk, kterým komunikuje síť. Tímto samozřejmě myslím použít scaling a převést výsledek na hodnotu mezi nulou a jedničkou.

## 3. Vstup

### 3.1 Design vstupní vrstvy

#### 3.1.1 Velikost vrstvy

Do vstupní vrstvy musíme dát dostatečné množství neuronů, aby mohli obsáhnout všechny důležité informace. Pokud budu například používat jako data obrázky o velikosti  $28 \times 28$ px, tak vhodný počet vstupních neuronů je 784, takto neztratím žádná data a dokonce budou mít takhle všechny pixely rovnocenný vliv. Pokud budu, ale používat jako data databázi s údaji o osobě, bude mi stačit neuronů zhruba stejný počet jako je sloupců v databázi.

### 3.2 Scaling

Jak již dobře víme, scaling je používán na získání hodnot mezi 0 a 1 ze vstupních dat. Naším vstupním číslem by síť totiž nemusela rozumět hned od začátku, jelikož se věk uvádí v řádu desítek, ale výplaty dosahují přibližně o 4 řády více. Vlivy těchto vstupních hodnot budou značně odlišné, což způsobí nechtěné nepřesnosti. Až později pochopíme jak funguje teoretické učení sítě zjistíme, že se tomuto kroku můžeme vyhnout, avšak z praktických zkušeností pak usoudíme, že scaling je docela užitečný krok.

### 3.3 Data

U začátku designování a programování sítě stojí vždy nějaká data. Každý trochu pokročilý programátor jistě dokáže vyjmenovat desítky případů, kdy mu znalost dat ulehčila práci. Tato znalost je u neuronových sítí naprosto zásadní. Když nevím co moje data znamenají nemůžu čekat, že to strojové učení odhadne za mě.

Zároveň se musím postarat, abych použil data pravdivá, ve kterých se můžou opravdu existovat korelace, které můžu potvrdit letmou úvahou. Budu-li trénovat síť na míchání barev, musím použít reálné data o míchání barev. Kdybych si výsledné barvy vymýšlel náhodně můžu čekat, že odhady sítě budou také celkem náhodné. Když se později poohlédneme za cestou od začátku do vytrénované sítě uvidíme, že dvě stejné sítě můžou být vytrénovány na dvě různé činnosti. Celkově bych to shnul upraveným příslovím: "Síť nepadá daleko od dat."

#### 3.3.1 Trénovací

Tento pojem již známe. Pojďme se tedy podívat na příklad míchání barev. Na trénovacích datech obzvlášť záleží. Musíme z existujících údajů tedy vybrat co nejvíc náhodně vzorky,

jejichž smíchání bylo ovlivněno co nejvíce faktory. Jednoduše pokud si vyberu na trénování pouze temné barvy, nemůžu potom očekávat, že síť správně smíchá dvě světlé barvy.

### 3.3.2 Testovací

Právě na těchto datech se uvidí, zda byly dosavadní kroky provedené úspěšně. Dosavadní kroky myslím celý postup, jelikož testovací data přichází skoro až na konci vývoje sítě. Jsou to právě tyto data, která nám oznámí jak dobře jsme odvedli práci trénování. Pokud se však nenachází podobné korelace mezi trénovacími a testovacími daty, bude náš test sítě neúspěšný a budeme se muset vrátit zpět ke trénování.

## 4. Forward-propagation

Je to právě tento postup, který nám umožňuje pracovat s neuronovými sítěmi. V průběhu vývoje se snažíme, aby tato fáze proběhla co nejlépe. Je to totiž výpočet toho co si síť myslí. Když tedy spustíme forward-propagation na nějakých datech dostaneme na výstup pořád pár divných čísel, podobných těm co vzniknou scalováním. To můžeme ale jednoduše změnit back-scalingem, což nám poskytne opravdu výstup, jaký by se dal čekat od živé bytosti se slušným uvažováním. Tento výstup bude zezáčátku pravděpodobně většinou chybný, ale postupem času a trénováním se dostaneme do bodu, kdy výsledky opravdu odpovídají realitě.

Nakonec když je síť ve fázi produkce, což znamená, že je využívána a už se nemění, nejsou všechny synapse tolik důležité. Dosahují totiž hodnot blízkých nule, což působí, jako by vůbec neexistovaly.

### 4.1 Typy sítí

Při náhledu do lidského mozku asi nenajdeme takhle hezky uspořádané sítě z rovnocenných neuronů. Stále však reprezentují dostatečně na to, aby to celé mohlo fungovat. Je dobré vědět, že existují i jiné typy sítí. Existují modifikace jenž umožňují pracovat s pamětí, vracejí již propočítaná data, nebo sami uspořádávají síť za běhu.

#### 4.1.1 Feed-forward

Tento typ je asi nejjednodušší případ využití neuronových sítí. Veškerá data postupují organizovaně dále do sítě a na konci dosáhnou výsledných neuronů.

Už víme, jak funguje neuron i aktivační funkce. Můžeme si tedy popsat tento zásadní algoritmus složený z toho co už známe.

Vezmeme data vložené do vstupní vrstvy, která se na diagramech značí nalevo a postupujeme podle čar doprava. Postupně vezmeme každou synapsi, co se v diagramu vyskytuje mezi vrstvami jenž právě počítáme, a vynásobíme její váhu s aktivací levého neuronu a přičteme do pravého neuronu, kde je na začátku nula. Poté co vyhodnotíme všechny synapse aplikujeme na každý neuron pravé vrstvy aktivační funkci. Když dojdeme do konce, úspěšně jsme vyhodnotili jeden vstup.

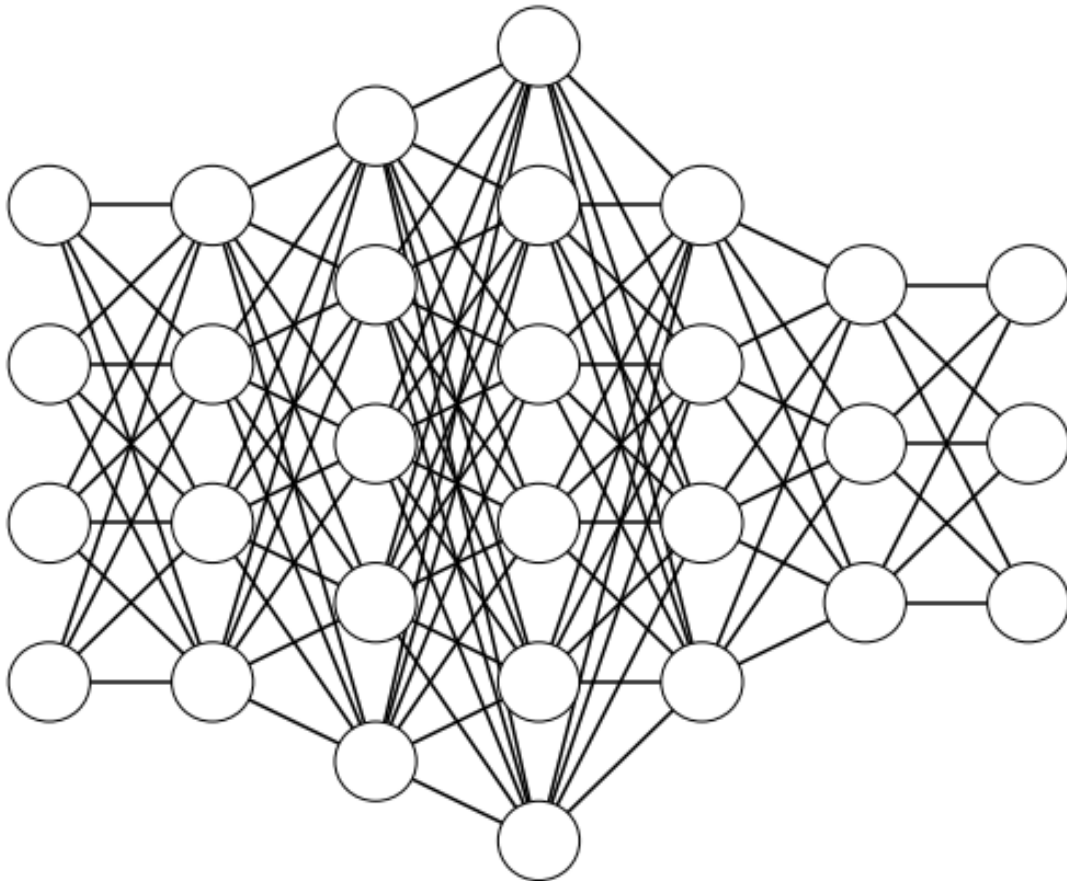
Při reprezentaci postupu pomocí programu můžeme využít násobení matice vrstvy s maticí vah a dostat tak výstupní vrstvu. Tímto způsobem můžeme zapsat zpracování libovolné vrstvy feed-forward modelu na jeden řádek.

$$A_{vstup} \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} \times W_{vstup} \begin{bmatrix} 0 & 1 & -1 & 1 & 0 \\ 3 & 1 & 0 & 0 & 2 \\ -1 & 2 & -3 & 0 & -1 \\ 1 & 2 & -1 & -2 & 0 \end{bmatrix} = A_{vrstva_1} \begin{bmatrix} 7 & 5 & -14 & -7 & 1 \end{bmatrix}$$

Na tuto matici stačí uplatnit aktivační funkci a pokračovat dalším násobením.

### 4.1.2 Deep learning

Od slova deep (hluboké) jsou tyto sítě používány na zpracování komplikovanějších dat, jako je například rozpoznávání znaků. Uvnitř mají totiž více vrstev a každá funguje jako vstupní vrstva pro další. Takto si můžou jednotlivé vrstvy předpracovat data a vrstva s výstupem už jen posbírá skoro hotové výsledky.



Obrázek 4.1: Více vrstev tvoří hloubku sítě

## 5. Back-propagation

Použitím této metody můžeme síť opravdu něco naučit. Stručně řečeno si síť porovná výsledky s očekávaními a poté upraví váhy v síti, tak aby nám další feed-forward fungoval trochu lépe.

### 5.1 Loss function

Celá síť je naprosto definovaná svým rozložením a vahami jednotlivých synapsí. Rozložení je fixní, takže pokud chceme ze sítě dostat maximum musíme upravovat váhy. Není však žádný jasný směr, kterým se vydat. Váh v síti je často od stovek k tisícům. Proto si zavedeme funkci loss neboli ztrátu. Tato funkce nám říká, jak moc je výsledek špatný. A najítím minimální hodnoty můžeme najít ideální stav sítě.

**Kvadratická ztrátová funkce** je nejčastěji používaná funkce. Její jednoduchost a lehká pochopitelnost jí staví nad všechny ostatní funkce, které mohou být pro určité případy lepší.  $y$  je očekávaný výstup,  $\hat{y}$  je vypočítaný výstup.

$$L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$$

V případě, že máme více výstupních neuronů, je  $L$  rovna sumě jednotlivých ztrát neuronů. Lineární multiplikátor  $\frac{1}{2}$  je zde pouze pro zjednodušení pozdější derivace.

### 5.2 Prakticky

Pokud bych chtěl vysvětlit tento postup bez použití složité matematiky, můžu použít uvažování o vlivu jednotlivých chyb na výsledek sítě. Spočítám si pro každý neuron ve výstupní vrstvě o kolik a jakým směrem ho musím změnit, abych dosáhl správného výsledku. Pomocí synapsí vedoucích do předchozí vrstvy vypočítám, jak moc ovlivnili neurony v předchozí vrstvě tento výsledek.

Neurony v předchozí vrstvě označím  $a_1$  až  $a_m$ , neurony v této vrstvě  $b_1$  až  $b_n$ . Váhu mezi neuronem  $a_k$  z první vrstvy a neuronem  $b_l$  z této vrstvy označím  $w_{k,l}$ .

Algoritmus v programu by mohl vypadat následovně:

1. V každém z neuronů  $a_1$  až  $a_m$  si vytvořím proměnnou  $d = 0$ , do které budu zapisovat, jak chtějí neurony  $b$ , aby vypadala aktivace (její relativní změna)
2. Vezmu neuron  $b_1$  a do proměnné  $d_1$  neuronů  $a_1$  až  $a_m$  připišu hodnotu váhy mezi neurony  $d_1 = d_1 + w_{k,1}$
3. Opakuji pro neurony  $b_2$  až  $b_n$

Tímto jsem do každého neuronu v předposlední vrstvě získal předsavu výstupní vrstvy o ideálním stavu pro tento vstup. Postup zopakuju pro všechny ostatní vrstvy, s tím, že současná předchozí vrstva se stane výstupní vrstvou. Když dojdeme k počítání vytoužených aktivací mezi vstupní vrstvou a první počítací vrstvou nemůžeme už měnit nic jiného, než váhy. Proto nastavím váhy tak, aby z aktivovaných vstupních neuronů byly kladné váhy do neuronů co mají být aktivované a negativní váhy do neuronů co mají být vypnuté. Ze slabě aktivovaných vstupních neuronů nemusím měnit nic, protože nemají vliv na současnou situaci. Místo toho si je můžu ponechat pro jiné vstupní data, protože tam můžou mít vliv větší.

Tímto postupem se můžu přiblížit k vytoužené síti, ale nevyužívám její plnou komplexitu. Proto můžu upravit i váhy v jiných vrstvách, než je mezi vstupní a první. Použiju úplně stejný postup. Pokud je neuron aktivovaný více nastavím váhu z něj do výstupů, co mají být aktivované trochu větší a do výstupů co mají být méně aktivované nastavím menší váhu.

Během upravování všech vah používám významnou konstantu nazvanou "learning rate", která určuje, jak rychle se bude síť učit. Touto konstantou násobím všechny změny, jelikož rychlé změny v síti můžou způsobit zapomenutí informací, jenž se naučil dříve.

## 6. Overfitting

Po tom co síť vytrénujeme s velkou přesností nad 95% na trénovacích datech, můžeme se radovat z velkého úspěchu. To ale není ještě výhra. Přesnost sítě se totiž musí hodnotit na datech testovacích. Během tréningu se totiž síť jako vedlejší produkt učí nazpaměť vstupy. Proto musíme v testu mít jiná data.

To znamená, že pokud máme vysokou úspěšnost na tréningu, ale nízkou na testu jsme obětí overfittingu. Do lidského uvažování to můžu přeložit, jako že program usoudil, že mu nedáváme dostatek dat na zjištění souvislostí a tak se naučil všechny vstupní data nazpaměť. Samozřejmě to znamená, že pokud programu nechám vyhodnotit i jednoduchou úlohu, tak nám nebude schopen odpovědět, pokud už takový příklad neviděl.

### 6.1 Řešení

Na předejití overfittingu existuje několik triků. Nejlepší by bylo použít od každého alespoň trochu.

- Přidáním vzorových dat
- Kontrolou, jestli jsou data relevantní
- Odstraněním vstupních hodnot, které nemají velký vliv
- Zastavením tréningu dřív, kdy test dosahuje dostatečně vysokých výsledků



## 7. Doporučení k programování

### 7.1 Jazyk

Jako vývojový jazyk jsem si zvolil Python. Konkrétně používám verzi 3.6.3 na operačním systému Windows 7 64-bit. Tento jazyk jsem si zvolil především, protože je v oboru výrazně preferovaný. Tomu odpovídá i množství knihoven, které pro Python v oblasti strojového učení vzniklo. Běh Pythonu není nejrychlejší, ale při odhadu úspory času na běhu u jiných jazyků zase ztrácím čas na psaní velkého množství kódu.

#### 7.1.1 Vývojové prostředí

Pro vývoj používám prostředí Spyder, které již podle názvu "Scientific PYthon Development EnviRonment" bylo navrženo čistě pro práci s Pythonem. Dva nejdůležitější nástroje poskytnuté prostředím jsou IPython console a Variable explorer. Instalaci jsem provedl skrz Pythnovou platformu pro data science jménem Anaconda.

**IPython console** umožňuje mít program zapnutý po celou dobu práce s modelem neuronové sítě. Ze začátku se zapne Python kernel, do kterého postupně nahráváme řádky kódu a ty se vyhodnocují. Styl jakým se nejčastěji píše kód z tohoto nástroje velmi benefituje. Jednotlivé části kódu na sebe totiž navazují, a proto je píšeme postupně. S náhledem do již vyhodnocené části kódu se nám následující program bude psát i opravovat rychleji. Tento nástroj opravdu oceníme, až budeme chtít upravit výpis dlouho trénované neuronové sítě, bez nutnosti pouštět celý program znovu.

**Variable explorer** nám dovolí dívat se živě na proměnné, které v programu existují. Největší uplatnění je asi na začátku, když se snažíme dostat data ze souboru do vstupní vrstvy a můžeme se dívat v jaké fázi se zrovna nachází.

#### 7.1.2 Knihovny

Knihovny usnadňují práci s programem. Můžeme si je představit právě jako knihy z knihovny plné funkcí, které jsou většinou velmi jednoduché, ale i přes to vyžadují čas na vytvoření a organizaci zdrojových souborů. Navíc jsou dost často optimalizované na to co dělají.

**Pandas** čte vstupní soubory a nahrává data do lehce čitelných proměnných.

**NumPy** poskytuje značné množství jednoduchých matematických operací a funkcí, ze kterých je strojové učení složené.

**Scikit-learn** je podstatná knihovna pro práci s daty. Do tohoto oboru spadají i neuronové sítě. Využívám ji na rozdělení dat na testovací a trénovací, a také pro určení chyby a účinnosti vytrénované sítě.

**Keras** je velmi pokročilá knihovna určená přímo pro neuronové sítě. Běží na základě knihoven Tensorflow od Googlu a Theano. Umožní nám zaměřit se přímo na stavění sítí a modelů, místo toho, abychom všechno psali od začátku můžeme začít pouze s teoretickými znalostmi.

**MatPlotLib** není knihovna, co by se značně podílela na funkčnosti. Poskytuje nám však značné možnosti ve vykreslování statistických údajů pomocí grafů.

## 7.2 Git

Pro zálohování a verzování používám Git. Konkrétně webovou službu GitHub `github.com/LukasCaha/GCHD_prace`, kde se nachází veškerý kód asociovaný k této práci.

## 8. Konvoluční předzpracování

Aplikováním následujícího postupu získáme vstupní hodnoty do neuronové sítě. Hlavní důvody, proč to děláme jsou rovnou dva. Prvním je přečtení souvislostí mezi jednotlivými pixely. Tato znalost je naprosto klíčová, protože jinak by se síť mohla zaměřit na výrazné, ale s obrázkem naprosto nesouvisející prvky (například šum v nočních snímcích). Druhým důvodem je zmenšení počtu informací, ale zachování informací rozhodujících pro správnou identifikaci obrázku. Tento postup nám tedy zrychlí zpracování bez ztráty kvality.

### 8.1 Konvoluce

**Konvoluce** je zpracování dvou signálů na základě jejich průniku. V matematice se s konvolucí setkáváme jako s operátorem pro dvě funkce. První funkci nazýváme obraz a druhou filtr. Při filtru na obrázku můžeme dosáhnout například zvýraznění hran. Tento filtr tedy hledá hrany.

**Využití** je v mém případě hledání rysů na obrázku. Na hodně exaktní předloze můžu najít pomocí specifických filterů například přechod mezi nebem a mořem. Poté můžu do této oblasti přidat červený odstín pro zdůraznění západu slunce.

Ve spojení s neuronovými sítěmi můžu automaticky určovat obsah obrázků. Na vstupní obrázek uplatním několik předem vytrénovaných filterů. Každý z nich zvýrazní oblasti obrázku, kde našel shodu. Při designu konvoluční vrstvy nastavujeme několik parametrů.

**Velikost filteru** záleží na uvážení, jestli spolu jednotlivé části obrázku souvisí hodně nebo málo.

**Počet filterů** určuje kolik filterů bude využito pro hledání znaků v obrázcích. Čím více jich bude, tím déle program poběží, ale budeme mít více rysů na kombinování.

### 8.2 Sdružování

**Sdružování** neboli pooling je způsob vyzdvížení důležitých rysů a zanedbání informací nedůležitých. Také mírně zvýší entropii, čímž umožní dvěma mírně různým obrázkům, aby po předzpracování vypadali stejně. Tímto algoritmem také dosáhneme největší úspory času, jelikož sdružovací hlavice dělá z  $n^2$  ( $n$  je malé přizobené číslo viz. obrázek) informací informaci jednu.

Stejně jako u všech algoritmů, které zde používám je mnoho možností jak je nastavit a získat tím různé výsledky různě rychle.

**Velikost poolu** je nastavena tak, aby jsme neztratili příliš mnoho informací, ale aby program běžel dostatečně rychle. Obvykle je vhodná velikost  $2 \times 2$ . Můžeme samozřejmě použít mřížku  $3 \times 3$ . Obecně se ale doporučuje mít shodné rozměry.

## 8.3 Vložení do sítě

Nakonec vezmu všechny sdružené obrázky a poskládám je do jednorozměrného pole. Tohle pole je vstupní vrstva neuronové sítě, kterou již známe. Po dokončení trénování stačí vložit obrázek na začátek konvoluce a stejným postupem nám výjde odhad.

## 9. Závěr

Během psaní této práce jsem se naučil naprosté základy neuronových sítí a jejich fungování. Pro učení bych ze zkušeností doporučil četné video návody, které lze najít na internetu zcela zdarma. V rámci procvičování jsem vytvořil tři modely neuronových sítí s postupně se zvyšující složitostí.

První program se učil souvislosti, které vytváří logické brány mezi vstupem a výstupem. Přesnost byla okolo 70%, což bylo způsobeno nedostatečnou složitostí problému a omezeností vstupních hodnot.

[https://github.com/LukasCaha/GCHD\\_prace/blob/master/src/net\\_logicgates.py](https://github.com/LukasCaha/GCHD_prace/blob/master/src/net_logicgates.py)

Druhý pokus mířil trochu výš, kdy program určoval, zda jsou buňky v poli zaškrtnuté vertikálně symetricky. Vzhledem k jednoduchosti problému bylo jednoduché získat velký počet trénovacích a testovacích dat. To zapříčinilo úžasnou úspěšnost nad 99% na testovacích datech.

[https://github.com/LukasCaha/ANN\\_symetry](https://github.com/LukasCaha/ANN_symetry)

Poslední počín mířil na počítačovou vizi (computer vizion), kde je účelem rozpoznat, zda je na obrázku pes, nebo kočka. Tento problém mi zabral nejvíce času, jelikož jsem se potýkal s dlouhou výpočetní dobou programu. Po asi deseti hodinách se mi tréninkem podařilo získat testovací úspěšnost 80%, což považuji za veliký úspěch, vzhledem k tomu jak byl problém komplexní.

[https://github.com/LukasCaha/GCHD\\_prace/blob/master/src/cnn.py](https://github.com/LukasCaha/GCHD_prace/blob/master/src/cnn.py)

# Literatura

- [1] Birge J. R., Wets R. J.-B. (1987): Computing bounds for stochastic programming problems by means of a generalized moment problem. *Mathematics of Operations Research* **12**, 149-162.

# Přílohy