Středoškolská technika 2019

Setkání a prezentace prací středoškolských studentů na ČVUT

Neurální sítě pro strojový překlad

Jan Ruman

SPŠ a VOŠ Písek, Karla Čapka 402, 397 11 Písek

Anotace

Cílem práce bylo implementovat již existující algoritmy využívající neurální sítě, analyzovat je a vytvořit vlastní model. Nejprve je řešeno získávání a vhodná úprava dat, následně způsoby a techniky učení modelů a nakonec samotná predikce. Pro jednodušší, přístupnější a přehlednější prezentaci modelu jsou dostupné webové stránky, které obsahují klíčové informace a umožňují interakci s jednotlivými modely.

Klíčová slova

Neuronové sítě; Strojové učení; Strojový překlad; Zpracování přirozeného jazyka

Annotation

The goal of this work was to implement already existing algorithms based on neural networks, analyze them and create new model. First of all, we solve the acquisition and appropriate adjustment of data, then ways and techniques for teaching models and eventually the prediction itself. For easier, more accessible and clearer model presentation, we provide a website that contains the key information about project and allows interaction with individual models.

Keywords

Neural networks; Machine learning; Machine translation; Natural language processing

Obsah

Úvod

Neuronové sítě v nedávné době začaly nacházet využití v mnoha oblastech – nejen ve zpracování obrazu, ale i v práci s textem nebo tabulkovými daty. Většina problémů, ve kterých neurální sítě podávají lepší výkony než doposud používané algoritmy, je specifická svou jednoduchostí pro člověka a zároveň náročností pro počítačový program.

Jedním z takových problémů je i překlad, i když o něm nelze mluvit jako o úplně triviálním. Aby byl překlad důvěryhodný, musí model být schopen pochopit co zdrojová věta vyjadřuje a zároveň vědět, jak tu samou myšlenku vyjádřit v jiném jazyce. Myšlenka takovéto „transformace“ textu z jednoho jazyka do druhého při zachování původního významu mě velmi zaujala.

Rozhodl jsem se tedy, že implementuji různé typy neuronových sítí, které se pro tento úkol v minulosti ukázaly jako účinné. Zajímalo mne, jak tyto modely vlastně fungují, co je potřeba pro jejich natrénování a jak dobré jsou pro samotný překlad. Tyto informace lze získat i z prací samotných autorů, mým cílem však bylo získat je vlastní experimentací. Zároveň jsem chtěl vyzkoušet vytvořit vlastní model, který jsem nakonec vytvořil na základě již existujících architektur.

Tato práce se zabývá množstvím různých aspektů týkajících se aplikace modelů pro strojové učení. Mezi ně patří například zpracování dat, implementace prostředí pro trénování modelů, ale i volba parametrů trénovacího cyklu. Na každý z těchto konceptů nahlížíme jak pohledem současné literatury, tak pohledem zkušeností získaných jejich použitím.

Vlastní pozorování různých aspektů implementace a použití modelů, alespoň tedy ta nejpodstatnější, jsou na konci jednotlivých částí, se kterými souvisí, uvedená ve stručnosti kurzívou.

Webové stránky práce – neuralmachinetranslation.rocks

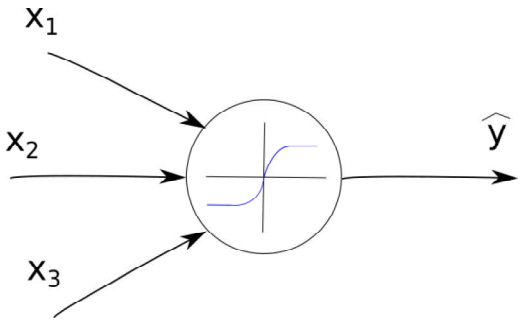
Teoretický základ

Princip činnosti neuronových sítí

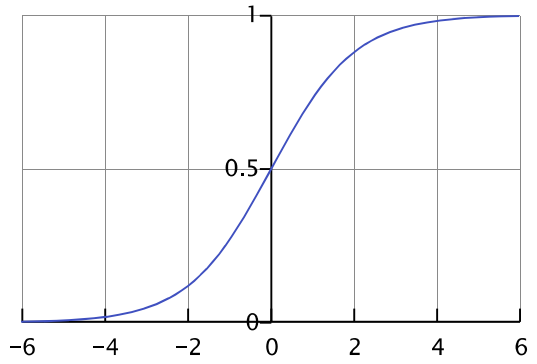
Umělé neuronové sítě jsou souborem algoritmů inspirovaných biologickými strukturami, které řídí chování živočichů. Různé architektury těchto sítí jsou používány například pro rozpoznávání objektů, analýzu sentimentu nebo predikci časových řad.

Umělý neuron

Společným základem těchto algoritmů je umělý neuron. Jeho funkcí je vytvořit jednu výstupní hodnotu $\hat{y}$ na základě několika příchozích vstupních hodnot $x_1, x_2, \ldots, x_n$. Zároveň by měl být schopný měnit vliv vstupních hodnot na jeho výstup, tedy učit se. Využívá tedy váženého součtu (váhy $w_1, w_2, \ldots, w_n$), ke kterému je následně přičten práh b. Váhy a práh se tedy učením mění.



Obrázek 1: Umělý neuron

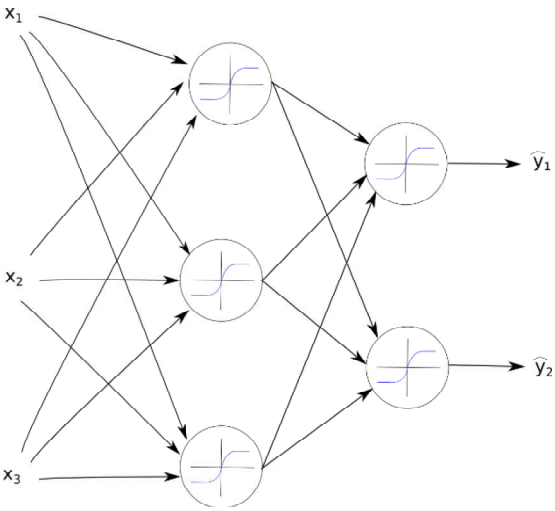


Obrázek 2: Sigmoid [11]

Tímto způsobem bychom však byli schopni modelovat pouze lineární funkce – proto na výsledek aplikujeme tzv. přenosovou funkci, která musí být nelineární a zároveň diferenciovatelná. Příkladem takové funkce je např. sigmoid, značený σ (Obrázek 2). Matematické vyjádření chování neuronu:

$$z = b + \sum_{i=0}^{n} w_i x_i$$

$$\hat{y} = \sigma(z)$$



Obrázek 3: Jednoduchá neuronová síť

Soubor více neuronů získávajících stejné vstupní hodnoty se nazývá vrstva. Postupným řazením více vrstev za sebe (kde vstupem každé následující vrstvy je výstup vrstvy předchozí) získáváme jednoduchou neurální síť (Obrázek 3).

Chybová funkce

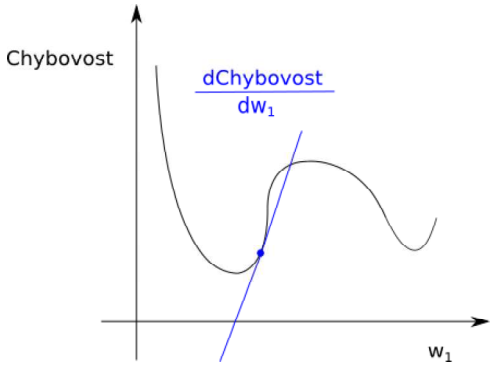
Chybová funkce nám udává míru rozdílu mezi výstupy neuronů $\hat{y}$ a hodnotami skutečnými y. Nejznámější takovou funkcí je střední kvadratická chyba:

$$MSE = \frac{1}{2m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

kde m je počet výstupních neuronů. Výstupem chybové funkce musí být jedna hodnota – na základě ní budeme měnit váhy neuronů a prahy vrstev v síti.

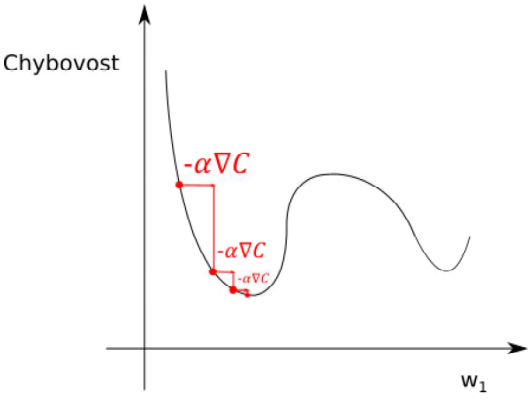
Backpropagation a SGD

Manipulací vah neuronů a prahů vrstev lze snížit chybovost výstupů. Takováto operace je vlastně závislostí chybovosti na velikosti jednotlivých vah, a tedy je sama o sobě funkcí (Obrázek 4). Jestliže chceme hodnotu chybovosti co nejmenší, hledáme tedy minimum této funkce – čehož lze (alespoň lokálně) dosáhnout pomocí částečné derivace chybovosti vůči jednotlivým vahám. Tento postup je u neurálních sítí nazýván *Backpropagation* [1]. Pro jednovrstvou neurální síť by platilo:



Obrázek 4: Backpropagation

$$\frac{\partial MSE}{\partial w_{i,j}} = \frac{\partial MSE}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial z_i}\frac{\partial z_i}{\partial w_{i,j}} = \frac{1}{m}(\hat{y}_i - y_i)\sigma'(z_i)x_j$$



Obrázek 5: SGD

Indexace – $w_{i,j}$ vyjadřuje váhu z neuronu j do neuronu i. Derivací všech vah (tenzor W) získáváme tenzor ∇C.

Síti můžeme přivádět různé kombinace vstupních hodnot, vypočítat chybovost na základě výstupu sítě a aktualizovat váhy. Velikost aktualizací lze regulovat parametrem α. Pokud budeme vstupní data vybírat náhodně a aktualizovat váhy po malém množství příkladů, získáváme *Stochastic gradient descent* [2][1] - SGD:

[1] SGD bylo sice publikováno o 35 let dříve než backpropagation, jedná se ale o nezávislý matematický koncept

$$W = W - \alpha \nabla C$$

Minimum funkce lze nalézt i analyticky, s nárůstem množství vah však výpočetní náročnost drasticky stoupá.

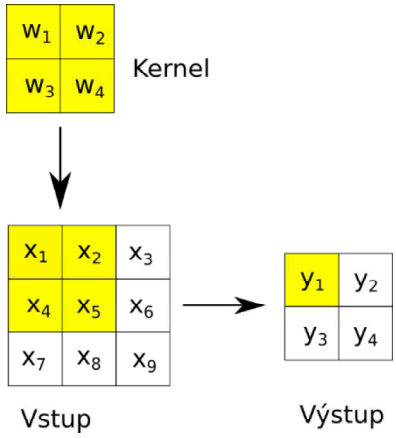
Variace neuronových sítí

Rekurentní neuronové sítě

Využívané v případech, kdy vstupem nebo výstupem je sekvence dat. Vstupní data jsou postupně přiváděna na vstup sítě, která jimi aktualizuje svůj současný stav. Výstupem může být buď stav finální, nebo stavy v jednotlivých časech.

Konvoluční neuronové sítě

Velmi dobře pracují s daty, kde záleží na lokálním kontextu. Využívají malou matici vah – kernel – která je postupně aplikována na vstupní hodnoty a následně posouvána.



Obrázek 6: Aplikace kernelu na vstupní hodnoty

Trénink

Proces, při kterém využíváme SGD pro dosažení co nejmenší chybovosti, nazýváme trénink modelu. Jako zdroj informací pro trénink využíváme tzv. soubory dat.

Soubor dat

Soubor dat je velké množství příkladů, složených ze vstupních hodnot a korespondujících hodnot výstupních. Lze jej charakterizovat na základě velikosti (tisíce až desítky milionů příkladů), kvality nebo obecnosti příkladů. Většinou je rozdělujeme na tři části: *trénovací*, využívanou k tréninku modelu; *validační*, na které testujeme účinnost modelu na datech, která nikdy neviděl; *testovací*, určenou pro závěrečnou evaluaci.

Hledání optimálních hyperparametrů

Hyperparametry jsou proměnné ovlivňující trénink a predikci, které se model nedokáže sám naučit. Za základní hyperparametry lze považovat: míru učení α, počet vrstev, počet neuronů v jednotlivých vrstvách a počet zpracovaných příkladů před aktualizací vah – batch size.

Při práci s různými soubory dat a modely vzniká potřeba různých hodnot hyperparametrů. Pro nalezení optimálních hodnot lze využít některou z těchto metod:

- Ruční hledání – autor postupně zkouší hodnoty hyperparametrů, které si myslí, že by mohli vést k optimálnímu výsledku
- Mřížkové hledání – jsou vyzkoušeny všechny kombinace různých hodnot hyperparametrů (nedoporučuje se – exponenciální nárůst výpočetní složitosti)
- Náhodné hledání – z autorem vybrané množiny možností je několik náhodně vybráno a vyzkoušeno (nepříliš efektivní metoda)
- Algoritmické metody – prohledávání prostoru možností za pomocí algoritmů schopných analyzovat vliv hodnot hyperparametrů a jejich interakce, momentálně ne příliš efektivní, avšak pravděpodobně nejvíce perspektivní; například Bayesovská optimalizace, evoluční algoritmy, …

Z dosavadních poznatků lze říci, že obecně čím větší množství dat, tím lépe bude model fungovat. Na základě toho vznikl tzv. *transfer learning*, neboli přenos učení. Jeho principem je natrénování modelu na obrovském množství dat a jeho následné dotrénování na požadovaném souboru dat.

Zpracování přirozeného jazyka

Jednou z aplikací neuronových sítí je zpracování přirozeného jazyka. Komplexita lidské řeči spočívá v obrovském množství kontextuálních závislostí, které je obtížné vyjádřit ručně psanými pravidly. Proto využíváme neuronové sítě, které jsou schopny se tyto závislosti naučit [3]. Příkladem takovýchto aplikací může být například analýza sentimentu pro klasifikaci kritiky jako pozitivní, nebo negativní, či shrnutí textu, kde je model schopný zkráceně vyjádřit myšlenku většího množství textu.

Úprava textu do formy vhodné pro neuronovou síť

Text lze vnímat jako sekvenci slov (popřípadě písmen), které nazýváme *tokeny*. Pro jednodušší trénink a překlad využíváme navíc tyto speciální tokeny:

- <unk> - používáno místo slov, která model nezná
- <pad> - paralelní komputace při tréninku vyžaduje všechny příklady v rámci jedné skupiny stejně dlouhé – tímto tokenem tedy vyplňujeme místo u kratších příkladů
- <SOS>, <EOS> - *start of string*, *end of string* – upozorňují na začátek či konec sekvence

```
['<sos>', 'quick', 'brown', 'fox', '<eos>', '<pad>', '<pad>']
```

Obrázek 7: Příklad tokenizovaného textu

Jelikož pracujeme se sekvenčními daty, využíváme většinou rekurentní neurální sítě.

Na vstup neurální sítě nemůžeme přivést samotné tokeny. Každému z nich tedy přiřadíme unikátní přirozené číslo (popřípadě nulu), tedy provedeme *numerizaci*. Takováto reprezentace je však nevhodná, neboť jedno číslo nedokáže dostatečně charakterizovat plný význam tokenu.

Číselnou reprezentaci tokenu tedy využijeme pro indexaci matice, jejíž počet řádků bude odpovídat počtu slov a počet sloupců bude udávat jak velký vektor bude každému slovu přiřazen. Získáváme tak vektorovou reprezentaci tokenu, neboli *embedding*, který je použitelný jako vstup do neuronové sítě.

Strojový překlad

Jednou z aplikací neurálních sítí je právě strojový překlad. V minulosti byly k tomuto účelu užívány statistické modely, které si na základě dvojjazyčného souboru dat vytvořily distribuce nejpravděpodobnějších překladů individuálních slov, popřípadě n-tic slov. Svojí efektivitou je však neurální sítě překonaly [4].

Pravděpodobně nejpoužívanějším neurálním modelem pro strojový překlad je *LSTM RNN* – Long-Short Term Memory Reccurent Neural Network. V principu se jedná o rekurentní neurální síť vybavenou mechanismem pro zapamatování dlouhodobých souvislostí, který zároveň zabraňuje explozi nebo vymizení gradientu. Jedna takováto síť postupně zpracuje vstupní data a vytvoří vektorovou reprezentaci vstupních dat, která je následně použita jako vstup druhé sítě generující výstupní text. Tvorba výstupu končí tokenem <eos>.

BLEU

Pro hodnocení kvality překladu se používá skóre *BLEU*. Tato metrika bere v potaz kolik n-tic tokenů vygenerovaných modelem se shoduje se skutečnými, a zároveň bere v potaz i několik možných překladů (za podmínky, že jsou součástí souboru dat). Lze ji tedy použít pro hrubé porovnání účinnosti různých modelů.

Neurální sítě pro strojový překlad

Proces vývoje programu pro překlad lze rozdělit na několik základních částí:

- Zpracování dat
- Implementace modelů
- Trénink modelu
- Evaluace a tvorba překladů
- Uvedení do produkce

Veškeré tyto kroky by bylo možné provést bez jakýchkoliv knihoven a dalších předem vytvořených nástrojů. Pro cíl této práce by však takový přístup nebyl přínosný, spíše naopak. Ještě předtím, než se vrhneme do vývoje samotného, si proto uvedeme několik klíčových nástrojů, které tato práce používá.

Nástroje

Python

Velmi oblíbeným jazykem pro strojové učení je Python. Tento vysokoúrovňový jazyk je charakteristický svou jednoduchostí a zároveň všestranností. Velké množství optimalizovaných knihoven pro strojové učení je napsáno v tomto jazyce, což byl primární důvod pro jeho výběr k realizaci této práce.

Pytorch

Nejdůležitější knihovnou pro vznik této práce je Pytorch [5]. Zásadním nástrojem, který nabízí, je *autograd.* Tento modul umožňuje automaticky vypočítat derivace na základě dynamického grafu – jednotlivé operace provedené na hodnotách jsou zaznamenávány, a jelikož známe derivace těchto funkcí, lze získat pomocí aplikace řetězového pravidla derivaci výsledku vůči libovolné proměnné.

Pytorch nabízí i množství již implementovaných funkcí, které odstraňují nutnost psát vše od nuly, hlavní výhodou oproti vlastním funkcím však představuje pokročilá optimalizace a podpora GPU.

Torchtext

Torchtext je dceřinou knihovnou Pytorch. Jedná se o rozšíření pro lepší podporu zpracování přirozeného jazyka, které zahrnuje množství abstrakcí usnadňující práci s tímto typem dat. Jako jednu z nejdůležitějších lze označit *iterator*, usnadňující aplikaci tokenizace a numerizace na soubory dat.

Nástroje vývoje a nasazení webové aplikace

Backend:

- Flask – webový framework
- Gunicorn – server
- Nginx – proxy

Frontend:

- React – Javascript knihovna pro tvorbu uživatelského rozhraní

Soubory dat

Charakteristika

K natrénování modelů potřebujeme data. Zásadní pro tuto práci je soubor dat Multi30k [5], na jehož překlad se budeme zaměřovat. Dalším souborem dat, který budeme využívat, je IWSLT [6]. Oba tyto soubory dat jsou charakteristické svou poměrně dobrou kvalitou překladů.

Multi30k

Tento soubor dat vznikl překladem anglických popisů obrázků do němčiny. Jedná se o velmi malý soubor dat, obsahuje pouze přibližně 30 tisíc párů vět. Jelikož však obsahuje text z omezené oblasti jazyka – tedy jen popisy obrázků, a nikoliv například vzájemnou komunikaci mezi dvěma lidmi – neměla by jeho velikost činit až tak velký problém.

IWSLT

IWSLT vznikl překladem TED konferencí. Oproti Multi30k obsahuje již větší množství párů vět – okolo 200 tisíc.

Zpracování

Oba soubory dat byly původně rozděleny do souborů podle příslušného jazyka (popřípadě podle toho, jestli se jednalo o část trénovací či validační). Aby bylo možné s daty dále pracovat, bylo potřeba tyto soubory nejprve sjednotit do jedné dvojjazyčné struktury (a pokud tomu ještě tak nebylo, rozdělit na trénovací a validační část).

Při počátečním vývoji programu a hledání hyperparametrů je potřeba pracovat s daty, které lze načíst v rozumném čase. Načtení souboru Multi30k není problém, avšak IWSLT již zabere řádově minuty. Proto z něj odebereme vzorek a budeme zezačátku pracovat s ním (jako velikost vzorku zvolíme desetinu velikosti původních dat).

```
def proc_multi30k():
      train_en = 'data/Multi30k/raw/train.en'
      train_de = 'data/Multi30k/raw/train.de'

      val_en = 'data/Multi30k/raw/val.en'
      val_de = 'data/Multi30k/raw/val.de'

      train = pd.DataFrame()
      train['en'] = pd.read_table(train_en,header=None).iloc[:,0]
      train['de'] = pd.read_table(train_de,header=None).iloc[:,0]

      val = pd.DataFrame()
      val['en'] = pd.read_table(val_en,header=None).iloc[:,0]
      val['de'] = pd.read_table(val_de,header=None).iloc[:,0]

      train_path = 'data/Multi30k/processed/train.en_de'
      val_path = 'data/Multi30k/processed/val.en_de'

      train.to_csv(train_path,sep='\t',index=False)
      val.to_csv(val_path,sep='\t',index=False)
```

Figure 1: Příklad zpracování - Multi30k

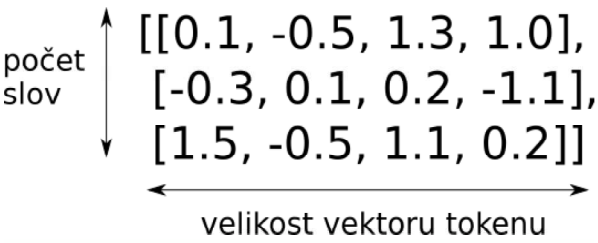
Načítání

Abychom mohli data používat pro trénink a překlad, je potřeba je vhodným způsobem načíst do paměti. Využíváme hlavně funkce knihovny torchtext – pomocí nich tato data načteme, provedeme tokenizaci a numerizaci, rozdělíme je na jednotlivé skupiny příkladů a vytvoříme *iterator*, který umožňuje jednoduchou manipulaci s daty.

Original: Vier Jungen, die in einen eingelassenen Pool springen.
Tokenized: ['vier', 'jungen', 'die', 'in', 'einen', 'eingelassenen', 'pool', 'springen', '<EOS>', '<pad>', '<pad>']
Numerized: [104, 78, 14, 5, 16, 5849, 449, 260, 2, 1, 1]

Obrázek 8: Příklad zpracování zdrojové věty z Multi30k

Během tohoto zpracování vzniká spolu s *iteratorem* i tzv. slovník, který mapuje tokeny k jejich korespondujícím číslům. Toto mapování získáme iterací přes data a postupným zaznamenáváním nových tokenů, kterým zároveň přidělujeme nová čísla. Pokud bychom chtěli zmenšit velikost slovníku (tedy zmenšit počet slov, která zná), čímž by došlo k nahrazení některých tokenů tokenem <unk>, mohli bychom přestat zaznamenávat nová slova po dosažení námi nastaveného limitu. Další možností by bylo zařadit do slovníku pouze ta slova, u nichž počet výskytů v souboru dosahuje určitého množství.



Obrázek 9: Příklad embedding matice

Tuto možnost budeme využívat u IWSLT. Počet všech unikátních slov je v tomto případě přibližně 130 tisíc. Každé toto slovo by v musel mít v embedding matici svůj řádek – pro udržení celkové velikosti modelu by pak bylo potřeba snížit hodnoty ostatních parametrů sítě, jako je například počet sloupců v této matici nebo počet neuronů v jednotlivých vrstvách.

Celkový počet slov, která model zná, je přímo úměrný velikosti modelu.

Z tohoto důvodu zahrneme do slovníku pouze slova, která se v datech vyskytují alespoň 3krát a snížíme tak počet známých slov a velikost modelu.

```
DE.build_vocab(train_ds.de, min_freq=10)
EN.build_vocab(train_ds.en, min_freq=10)
```

Figure 2: Vytvoření slovníku se slovy, která se vyskytují alespoň 10krát

Modely

Obecně

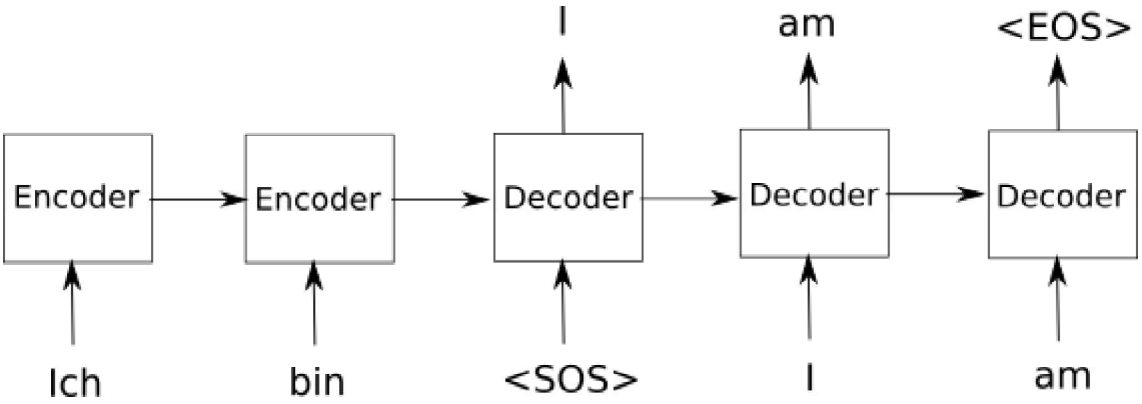
Charakteristickým znakem modelů pro strojový překlad je jejich schopnost jak zpracovávat, tak vytvářet sekvence tokenů. K tomu téměř vždy využíváme dvojici neurálních sítí, z nichž jedna má na starost extrakci významu zdrojové věty a druhá zajišťuje jeho vyjádření v cílovém jazyce. Tuto dvojici nazýváme kodér (encoder) a dekodér (decoder).

Modely pro strojový překlad většinou generují výstup token po tokenu, přičemž jeden či více předchozích tokenů slouží v daném čase jako vstup. Ve chvíli, kdy model vygeneruje jedno špatné slovo, se tedy dostává do pozice, kdy jeho vstup v dalším čase není správný. Jako řešení lze použít tzv. *teacher forcing*, který na vstup v dalším čase nepřivádí

předchozí vygenerované tokeny, nýbrž tokeny z již přeložené věty, u kterých jsme si jistí, že jsou správně. Je tedy jasné, že tuto metodu lze použít pouze při tréninku. Teacher forcing má ještě jednu výhodu – je rychlejší, a to především u nerekurentních modelů.

Celkem budeme využívat čtyř modelů – LSTM RNN [4], RNNsearch [7], ConvS2S [8] a můj model.

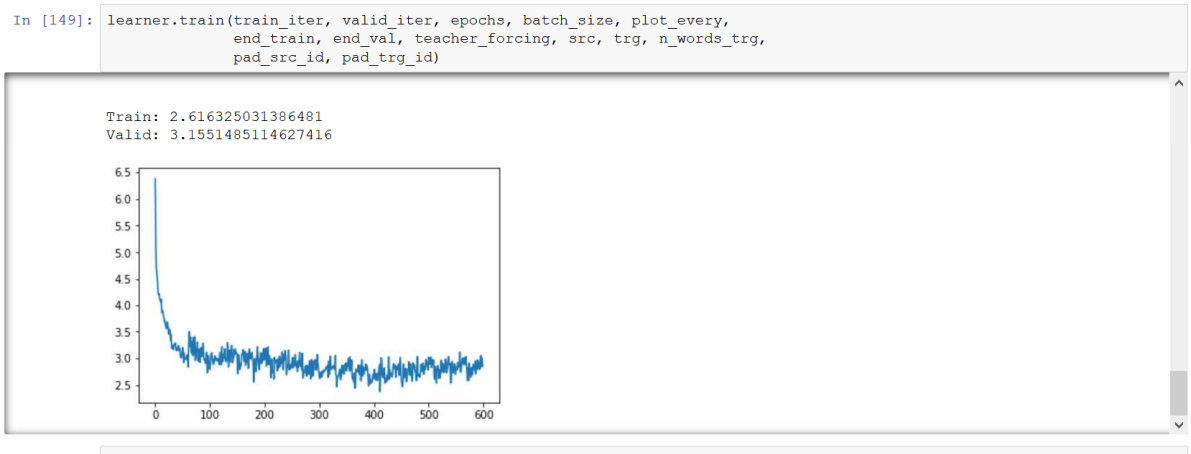
LSTM RNN



Obrázek 10: Princip LSTM RNN

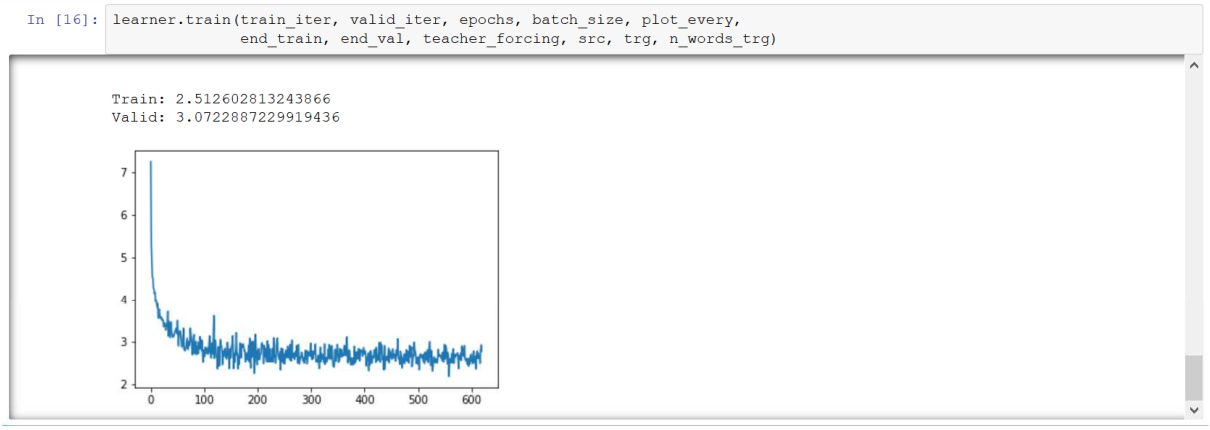
Princip

Jedná se o jeden z prvních úspěšných pokusů o aplikaci neurálních sítí na strojový překlad. Kodér i dekodér jsou tvořeny rekurentními neuronovými sítěmi. Jeho princip je poměrně jednoduchý – kodéru na vstup postupně přivádíme jednotlivá slova, pomocí nichž aktualizujeme vnitřní stav této sítě. Tento stav vložíme do dekodéru, u kterého již zaznamenáváme jeho výstupy v jednotlivých časech – tyto výstupy jsou indexy tokenů cílového jazyka. Předešlý výstupní token poté vkládáme jako vstup do sítě, čímž aktualizujeme její stav.



Obrázek 11: Chybovost po tréninku klasické LSTM RNN

Autoři modelu uvádějí, že získali znatelné zlepšení kvality překladu otočením pořadí slov ve zdrojové větě. Tato úprava má jednoduché vysvětlení – první slova věty zdrojové budou zakódována až nakonec, zatímco dekódována jsou ihned na začátku – tímto tedy nebude tak jednoduché tato slova zapomenout.



Obrázek 12: Chybovost po tréninku LSTM RNN s otočeným pořadím slov

Z tohoto pohledu jsou tedy dvě možnosti jak tento model implementovat. Proto jsem se rozhodl vyzkoušet obojí a výsledky porovnat, přičemž jsem nakonec dosáhl stejného závěru, jako samotní autoři – viz. Obrázky 11 a 12.

Stejně jako uvedli autoři modelu, i zde pozorujeme snížení chybovosti při otočení pořadí slov.

Výhody a nevýhody

Výhody

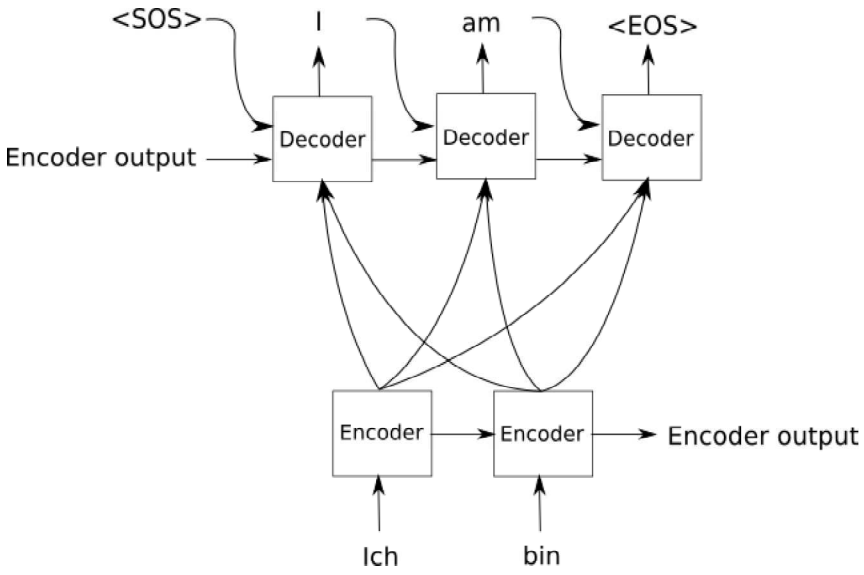
- Jednoduchý – jak na pochopení, tak na implementaci

Nevýhody

- Pomalý trénink[2]
- Horší kvalita překladu

[2] Rekurentní modely nejsou kompletně paralelizovatelné, což zpomaluje jejich trénink

RNNsearch



Obrázek 13: Princip RNNsearch

Princip

Tento model je velmi podobný LSTM RNN. Přináší však nový koncept – pozornost. Kodér již nyní nevyužíváme pouze k vytvoření stavu, ale i ke generování výstupů v jednotlivých časech – ty vycházejí z vnitřního stavu, který závisí nejen na okamžitém vstupu, ale i na vstupech okolních. Dekodér má v každém čase přístup ke všem výstupům kodéru a využívá je ke generování překladu. Výhodu zde činí fakt, že veškerý význam zdrojové věty není potřeba zkomprimovat do jednoho vektoru (= stavu), ale je možné jej distribuovat mezi výstupy kodéru.

Tato architektura využívá obousměrného zpracování – stav v daném čase není tedy ovlivněn pouze vstupy předchozími, ale i následujícími. Obousměrné zpracování aplikujeme pouze na kodér, jelikož u dekodéru neznáme následující výstupy. Jak kodér, tak i dekodér mohou mít více než jednu vrstvu.

Výhody a nevýhody

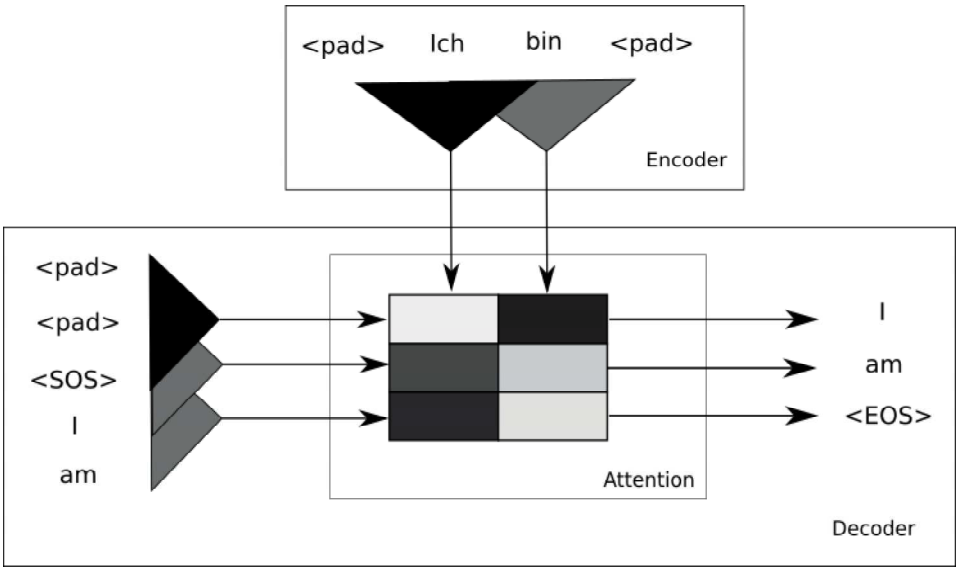
Výhody

- Stále relativně jednoduchý
- Lepší kvalita překladů

Nevýhody

- Pomalý trénink

ConvS2S



Obrázek 14: Princip ConvS2S

Princip

ConvS2S přináší do světa překladu něco velmi neočekávaného – konvoluční neuronovou síť. Oproti příkladu uvedeném v teorii se však jedná pouze o 1D konvoluci, její kernel již tedy není matice, ale vektor (v praxi to matice je, jelikož každé slovo je reprezentováno jako vektor, a ne jako číslo – pro přehlednost je však lepší o kernelu uvažovat jako o vektoru).

Velikost kernelu nám určuje na kolik slov ho budeme aplikovat (na Obrázku 14 pracujeme s velikostí kernelu 3). Jednou aplikací kernelu získáme jednu výstupní hodnotu a následně kernel posouváme o jedno slovo dál. Při kódování i dekódování přidáváme <pad> tokeny pro zachování délky věty i po aplikaci konvoluce (tedy pokud bychom aplikovali konvoluci s kernelem o velikosti 3 na větu o délce 5 a chtěli bychom jako výstup získat opět 5 hodnot, museli bychom přidat 2 <pad> tokeny). Na výstupy konvolucí kodéru i dekodéru opět aplikujeme pozornost a tím získáváme výstupní hodnoty.

Je zde ještě několik technických detailů, které je třeba brát v potaz:

- Ke klasickému embeddingu tokenů přičítáme embedding pozice tokenu ve větě (např. ve větě „Ich bin wütend“ jsou pozice jednotlivých slov [1, 2, 3])
- Kodér i dekodér mají více vrstev
- Využíváme tzv. *Residuální spoje* – k výstupu vrstvy konvolucí přičítáme jejich vstup
- Využíváme speciální inicializaci vah a normalizaci aktivací

Tento model jsem vyvíjel a testoval postupně, viděl jsem tedy, která specifika mají jaký vliv na jeho chybovost. Za nejdůležitější bych považoval embedding pozic a inicializaci vah. Zatímco embedding pozic znatelně snížil chybovost modelu po jeho tréninku, bez speciální inicializace tento model ani nebylo možné natrénovat – jakkoliv opatrně jsem manipuloval s ostatními parametry, chybovost po krátké době zkrátka explodovala.

Speciální inicializace umožnila vůbec model natrénovat. Embedding pozic znatelně snížil jeho chybovost.

Výhody a nevýhody

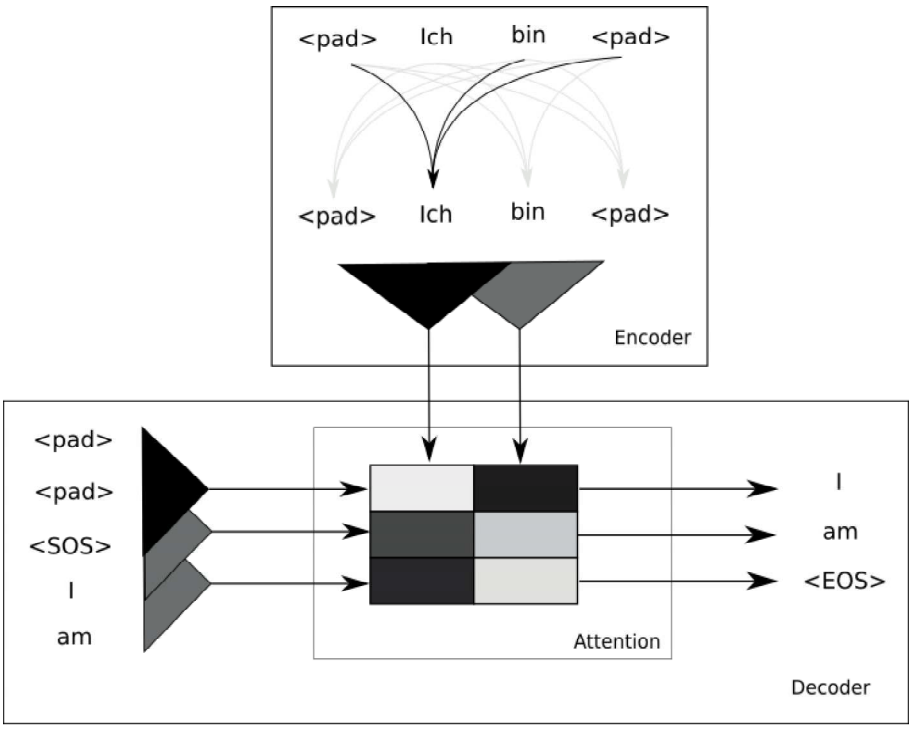
Výhody

- Lepší kvalita překladů oproti předchozím modelům
- Rychlost tréninku
- Umožňuje velké množství vrstev

Nevýhody

- Vyžaduje důkladnou inicializaci a normalizaci
- Je složitější

Můj model



Obrázek 15: Princip mého modelu

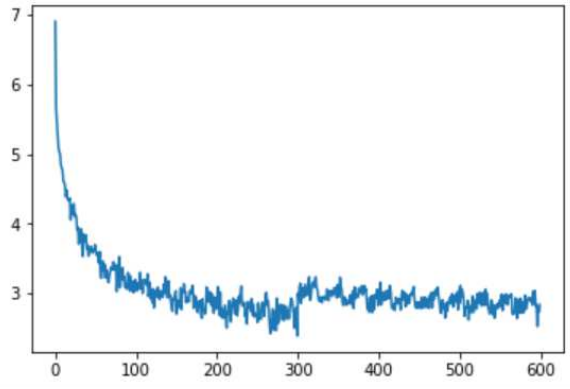
Princip

Jak už lze pozorovat z obrázku, můj model rozšiřuje architekturu ConvS2S. Jediným rozdílem je přidaný mechanizmus sebe-pozorování [9]. Jelikož konvoluční sítě kladou velký důraz na zakódování lokálního kontextu, napadlo mě, že by tato architektura mohla využít mechanizmus pro pozorování ostatních slov i přes jejich velkou vzdálenost ve větě. Využívá tedy pozornost, avšak ne vůči tokenům z jiného jazyka, ale vůči ostatním tokenům té samé věty.

Při použití modelu pro překlad jsem nezaznamenal žádné znatelné zlepšení oproti ConvS2S. Možným vysvětlením by mohlo být to, že konvoluční neuronová síť je schopna pracovat i se závislostmi mezi slovy, které jsou od sebe vzdálené, a to díky své hierarchické struktuře, která jí dává přístup k většímu množství okolních slov s každou další vrstvou. Další mechanizmus by pak v takovém případě byl zbytečný.

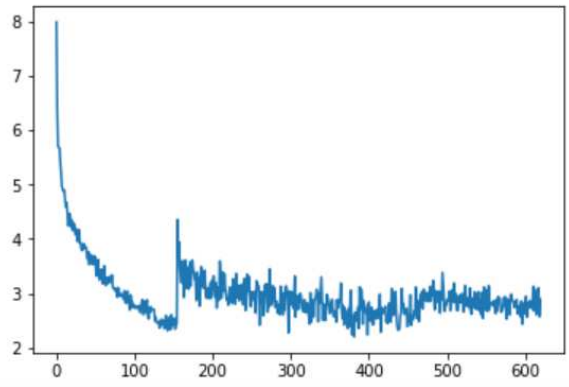
Mechanizmus sebe-pozornosti nezlepšil kvalitu překladu modelu ConvS2S.

```
Train: 2.3091248696500606
Valid: 2.8473680127750742
```



Obrázek 16: Chybovost ConvS2S

```
Train: 2.1473750230428337
Valid: 2.812874594250241
```

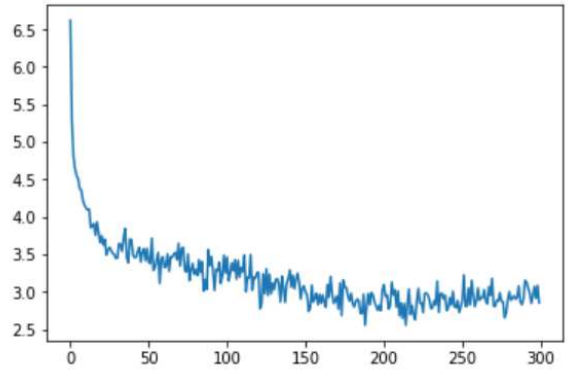


Obrázek 17: Chybovost mého modelu

Jedním z mých prvních nápadů (který jsem dokonce i realizoval) byla neurální síť založená čistě na konceptu sebe-pozornosti. Její chybovost bohužel dosahovala hodnot horších, než u LSTM RNN, rozhodl jsem se tedy tento koncept nevyužít.

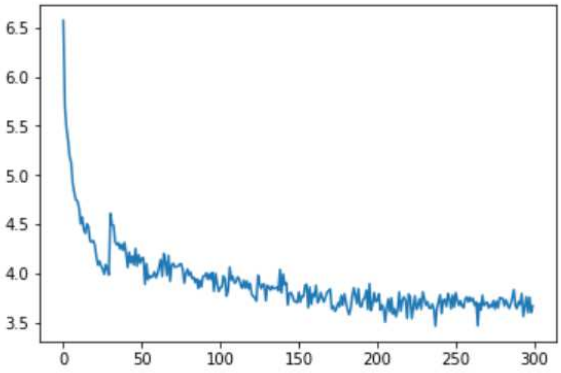
V tomto případě samotná sebe-pozornost nestačila jako základ úspěšného modelu.

```
Train: 2.7822576463222504
Valid: 3.2157924936877356
```



Obrázek 18: Chybovost LSTM RNN

```
Train: 3.253697947661082
Valid: 3.5282559712727863
```



Obrázek 19: Chybovost mého původního modelu

Výhody a nevýhody

Výhody a nevýhody mého modelu jsou stejné, jako u ConvS2S, můj model je pouze o něco málo složitější.

Trénink

Nyní použijeme data Multi30k a IWSLT pro natrénování modelů. Budeme hledat správné hyperparametry a využijeme tzv. *transfer learning*.

Transfer learning

Pro prvotní natrénování modelů použijeme IWSLT, a následně je doladíme tréninkem na Multi30k. Během těchto dvou fází tréninku je potřeba dávat pozor na to, abychom v obou případech použili stejný slovník – jinak by se mohlo stát, že ten samý token bude mít v každé z fází tréninku přiřazená jiná čísla, a tedy i jiné vektory.

```
In [125]: t1 = time.time()
          t1-t0
Out[125]: 7761.663114309311
```

Obrázek 20: Doba tréninku nového modelu

```
In [24]: t1 = time.time()
         t1-t0
Out[24]: 1651.5400030612946
```

```
In [32]: t1 = time.time()
         t1-t0
Out[32]: 414.6447026729584
```

```
In [40]: t1 = time.time()
         t1-t0
Out[40]: 210.91653680801392
```

Obrázek 21: Části doby tréninku předtrénovaného modelu

Zásadní rozdíl, který jsem zaznamenal při využití *transfer learning* byl rozdíl rychlosti trénování mezi modelem novým a modelem předtrénovaným. Například u ConvS2S trvalo natrénovat nový model 3.5× déle, než doladit již předtrénovaný model té samé velikosti.

Předtrénovaný model lze doladit za znatelně kratší dobu, než jakou zabere trénování modelu od nuly.

ConvS2S

```
In [2]: train_iter, valid_iter, SRC, TRG = load_IWSLT_sample(64)
        src, trg = 'de','en'
        ds_name = 'IWSLT'

In [3]: n_words_src = len(SRC.vocab)
        n_words_trg = len(TRG.vocab)

In [4]: pad_src_id = SRC.vocab.stoi['<pad>']
        pad_trg_id = TRG.vocab.stoi['<pad>']

In [5]: trg_sos_id = TRG.vocab.stoi['<SOS>']
        trg_eos_id = TRG.vocab.stoi['<EOS>']
        src_eos_id = SRC.vocab.stoi['<EOS>']

In [6]: train_len = len(train_iter)
        val_len = len(valid_iter)

In [7]: model_name = 'ConvS2S'
```

T1 - wd = 0.0001, Adam, tf schedule [...]

T2 - wd = 0.0001, Adam, tf=0.5 [...]

T3 - wd = 0.0001, Adam, tf=0.0 [...]

Final [...]

Obrázek 22: Struktura tréninku - inicializace, hledání hyperparametrů, finální trénink

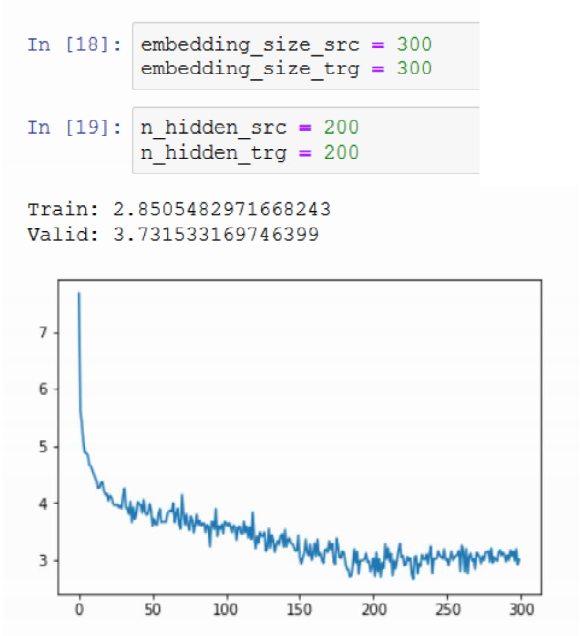
Hledání hyperparametrů

Trénink vždy začíná hledáním hyperparametrů. V tomto případě jsem zvolil jako způsob hledání manuální metodu – tedy bez jakéhokoliv algoritmu. Abych nalezl optimální sadu hyperparametrů, zkoušel jsem měnit jejich hodnoty jednu podruhé, přičemž jsem zaznamenával jejich vliv na chybovost modelu.

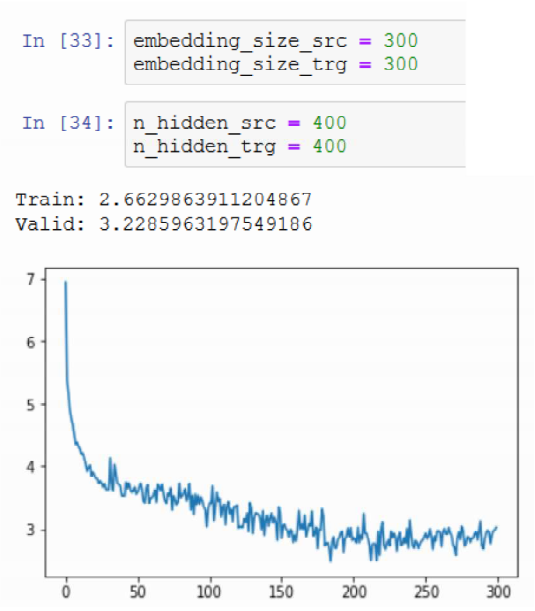
Testování hyperparametrů by bylo ideální provádět ve stejném měřítku, jako závěrečný trénink – tedy se všemi dostupnými daty a s využitím veškerého možného času. Tímto způsobem by však nebylo možné projít dostatečné množství různých kombinací hyperparametrů. S velikostí není u Multi30 problém, u IWSLT už však ano, a proto využijeme vzorek dat, který jsme již vytvořili. Jako optimální množství času pro zkoušku jednoho nastavení hyperparametrů jsem zvolil přibližně 1/8 až 1/10 délky finálního tréninku.

I přes tyto úpravy bylo hledání parametrů mnohem časově náročnější, nežli samotný finální trénink. S největší pravděpodobností tomu tak bylo kvůli tomu, že jsem ho prováděl manuálně – a jelikož každé další nastavení hyperparametrů, které jsem se rozhodl vyzkoušet, záleželo na výsledcích toho předchozího, nebylo možné dopředu navolit všechna nastavení a vyzkoušet je najednou např. přes noc.

Manuální hledání hyperparametrů bylo v tomto případě časově náročnější, než samotný trénink.



Obrázek 23: Chybovost modelu s menším počtem neuronů



Obrázek 24: Chybovost modelu s větším počtem neuronů

Co se týče samotných hodnot hyperparametrů, můžeme pozorovat dva zajímavé trendy – zvětšováním těchto hodnot (a tím zároveň i velikosti modelu) se chybovost zpravidla nezhorší a model zpravidla nemá o moc větší tendenci k tzv. *overfittingu* – velmi nízké chybovosti na trénovací a velmi vysoké chybovosti na validační části dat. Tím pádem velikost modelu už omezují pouze naše časové možnosti (větší model = delší trénink) a výpočetní výkonnost produkčního prostředí.

Zvětšování modelů zpravidla nemá záporný vliv na chybovost modelů.

Vlastní trénink

Podívejme se nyní na to, co trénink vlastně obnáší. Je potřeba zajistit načítání skupin příkladů ze souboru dat, predikci pomocí modelů, aktualizaci vah a počítání chybovosti. Pro tyto účely jsem vytvořil třídu *Learner*. Každý z modelů má svou podtřídu třídy Learner, která je uzpůsobená potřebám konkrétního modelu. Ukázku použití tohoto modulu lze vidět ve Figure 3.

```
enc = AttnEncoder(…)
dec = AttnDecoder(…)
learner = RNNsearch_Learner([enc, dec], …)
learner.train(train_iter, valid_iter, …)
```

Figure 3: Zkrácený úryvek kódu pro trénovaní modelu RNNsearch pomocí modulu Learner

V příkladech, která dáváme k překladu modelům, se vyskytují tokeny <pad>, které nenesou žádnou užitečnou informaci. Mohli bychom tedy ulehčit učení modelů tak, že bychom ignorovali jejich predikce na pozicích ve větě, kde se ve vzorovém překladu vyskytují <pad> tokeny (protože je nám vlastně stejně jedno, co na těchto místech dají jako výstup, jelikož se nacházejí za tokenem <EOS>).

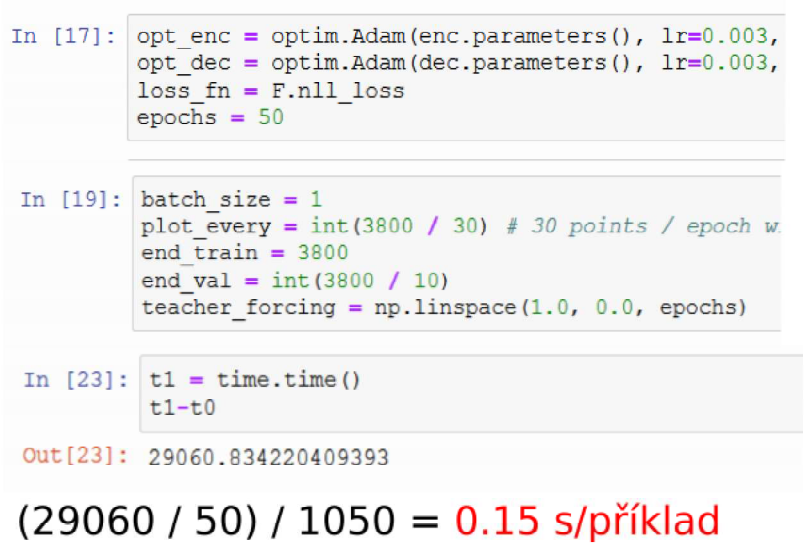
V praxi se tato úprava ukázala jako naprosto zbytečná, jelikož chybovost trénovaného modelu byla stejná jak v případě kdy jsme toto rozšíření použili, tak i v případě kdy jsme trénink ponechali tak, jak byl.

Ignorování chyb na pozicích, kde se ve vzorové větě vyskytuje <pad> token nesnížilo chybovost modelu.

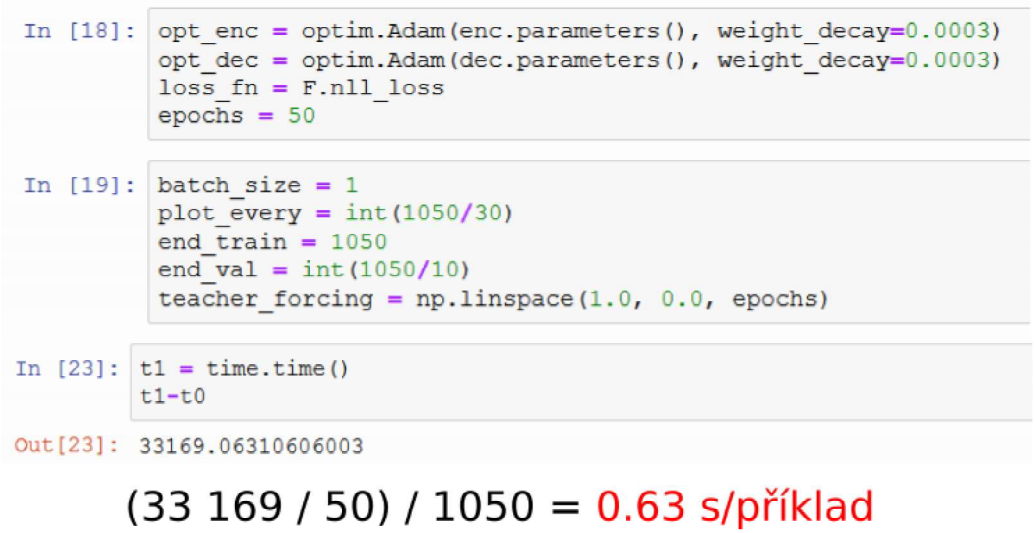
Podle teorie by měla být rychlost tréninku rekurentních modelů znatelně nižší ve srovnání s ostatními modely. Srovnáme-li si rychlost zpracování příkladu modelu RNNsearch a ConvS2S, můžeme pozorovat, že tomu tak skutečně je.

Přesně naopak však vypadá rychlost tréninku LSTM RNN ve srovnání s rychlostí ConvS2S. Důvod je jednoduchý – implementace LSTM v Pytorch je velmi dobře optimalizovaná. A jelikož ConvS2S je poměrně komplexní model a ne všechny jeho části jsou zdaleka tak dobře optimalizované, jako právě LSTM, dochází zde ke znatelnému zpomalení při zpracování příkladů.

Rychlost modelů je znatelněji více ovlivněna jejich implementací, nežli jejich vlastní strukturou; narozdíl od struktury je však implementaci možné optimalizovat.



Obrázek 25: Rychlost zpracování příkladu modelem LSTM RNN



Obrázek 26: Rychlost zpracování příkladu modelem RNNsearch

```
In [16]: opt_enc = optim.Adam(enc.parameters(), lr=0.0001, weight_decay=0.0001)
         opt_dec = optim.Adam(dec.parameters(), lr=0.0001, weight_decay=0.0001)
         # opt_enc = optim.Adam(enc.parameters())
         # opt_dec = optim.Adam(dec.parameters())
         # opt_enc = optim.SGD(enc.parameters(), lr=0.03, momentum=0.9, weight_d
         # opt_dec = optim.SGD(dec.parameters(), lr=0.03, momentum=0.9, weight_d

         loss_fn = F.nll_loss
         epochs = 50

In [17]: batch_size = 1
         plot_every = int(1800/30)
         end_train = 1800
         end_val = int(1800/10)
         teacher_forcing = np.linspace(1.0, 0.0, epochs)

In [21]: t1 = time.time()
         t1-t0

Out[21]: 27138.575883626938
```

(27138 / 50) / 1800 = 0.30 s/příklad

Obrázek 27: Rychlost zpracování příkladu modelem ConvS2S

Lze předpokládat, že při dolaďování předtrénovaného modelu již nebude potřeba tak velké míry učení α. Přesně tak tomu nebylo v případě této práce – veškeré dolaďování bylo z většiny tvořeno tréninkem s velkou mírou učení α, jelikož její snížení mělo za následek pouze zbytečné zpomalení tréninku. V případě LSTM RNN byla její hodnota při dolaďování dokonce totožná s její hodnotou při předtrénování (Obrázek 28, 29).

Pro doladění modelů bylo výhodnější použít velkou míru učení α.

```
In [17]: opt_enc = optim.Adam(enc.parameters(), lr=0.003, weight_decay=0.0003)
         opt_dec = optim.Adam(dec.parameters(), lr=0.003, weight_decay=0.0003)
         loss_fn = F.nll_loss
         epochs = 50
```

Obrázek 28: Předtrénovaní - stejný činitel učení jako dolaďování (parametr lr)

```
In [52]: opt_enc = optim.Adam(enc.parameters(), lr=0.003, weight_decay=0.0001)
         opt_dec = optim.Adam(dec.parameters(), lr=0.003, weight_decay=0.0001)
         loss_fn = F.nll_loss
         epochs = 20
```

Obrázek 29: Dolaďování - stejný činitel učení jako předtrénování (parametr lr)

Evaluace a predikce

Stejně jako jsme modelům při tréninku na vstup přiváděli příklady, tak i při predikci budeme dělat to samé, avšak model se již z těchto dat nebude učit. Zde bude důležitá evaluace modelů na validačních datech, kde budeme porovnávat výstup modelu se vzorovým překladem. Abychom byli schopni porovnávat kvalitu překladu jednotlivých modelů, využijeme skóre BLEU.

Beam search

V každém čase vždy daný model vytvoří jeden přeložený token, který bereme jako část výstupu a zároveň ho opět přivádíme dekodéru modelu jako vstup v dalším čase. Když v každém čase vybereme token, který model určil jako nejpravděpodobnější, nemusíme nutně získat větu, jejíž celková pravděpodobnost bude největší (využíváme totiž tzv. *greedy search*).

Z tohoto důvodu v každém čase vybereme tokenů několik (jejich počet odpovídá šířce paprsku *beam size)*. Celkovou pravděpodobnost věty získáme jako součin jednotlivých pravděpodobností, přičemž kompenzujeme na základě délky této věty (kratší věty by totiž jinak měly výhodu). Nakonec vybereme větu s největší takto upravenou pravděpodobností. Tento algoritmus se může zdát povědomí – jedná se o *beam search.*

Model	BLEU	BLEU – předtrénovaný model + beam search
LSTM RNN	11.4	12.1
RNNsearch	16.7	12.7
ConvS2S	20.8	21.5
MyModel	20.7	21.5

Tabulka 1: Porovnání skóre BLEU jednotlivých modelů (větší je lepší)

Účinnost modelů

V Tabulce 1 můžeme vidět porovnání jednotlivých modelů a jejich variací na základě skóre BLEU. Předpokládali bychom, že předtrénováním na větším souboru dat a použitím algoritmu *beam search* vylepšíme kvalitu překladu u všech modelů, tomuto pravidlu se však vymyká RNNsearch – jeho efektivita se vlivem těchto úprav naopak znatelně zhoršila.

Modely mají tendenci překládat lépe, pokud je předtrénujeme a použijeme beam search.

Originální text	**Vzorový překlad**
Ein schwarzer Mann mit weißem T-Shirt und schwarzen Kappe sitzt auf dem Bordstein und schreibt Textnachrichten.	A negro male in a white t-shirt and a black hat sitting on the curb texting.
LSTM RNN	**LSTM RNN – předtrénovaný, beam search**
a black man in a black tshirt and black and and and and down the	a black man with a black black and hat hat sitting on on on chair …
RNNsearch	**RNNsearch – předtrénovaný, beam search**
a black man in a white tshirt and black black hat on the curb	a black man with a white and black black cap cap cap on the the
ConvS2S	**ConvS2S – předtrénovaný, beam search**
a black man in a white shirt and black hat is on on on	a black man with a white and and black cap is on the curb ..
Můj model	**Můj model – předtrénovaný, beam search**
a black man in a white shirt and black hat is on on a a writing	a black man with a white and and black cap sits on on on and writing .

Tabulka 2: Příklad překladu stejné věty různými modely a jejich variacemi

Překlad

Tabulka 2 ukazuje příklad překladu věty jednotlivými modely a jejich variacemi. Tuto větu jsem vybral schválně, jelikož není natolik jednoduchá, aby jí všechny modely zvládly bez problému, ale zároveň ani natolik náročná, aby jí ani jeden model nezvládnul alespoň částečně přeložit. Velmi zhruba zde můžeme pozorovat, že skóre BLEU skutečně odpovídá kvalitě překladu.

Samozřejmě můžeme ve validační části souboru dat Multi30k nalézt jak příklady, které většina modelů přeloží téměř perfektně, tak i příklady, které ani jeden z modelů není schopen přeložit ani z části.

Originální text	Vzorový překlad	Můj model
Ein mann in einem grauen T-Shirt ruht sich aus.	A man in a gray t-shirt rests.	a man in a gray shirt is resting

Tabulka 3: Příklad téměř dokonalého překladu

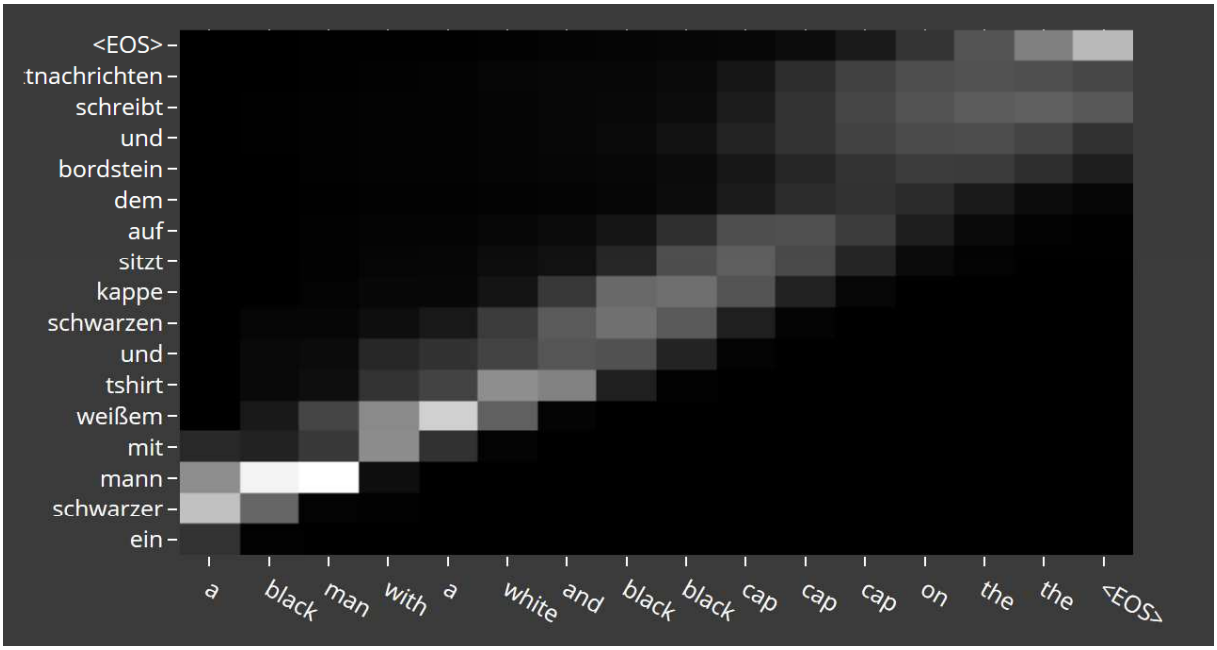
Originální text	Vzorový překlad	Můj model
Ein Mann mit einem Kapuzenshirt sitzt an einem Springbrunnen und beobachtet die Menschen der Stadt.	One man, wearing a hooded sweatshirt, sitting at a fountain watching the people of the city.	a man with a a is sitting a a a a the the

Tabulka 4: Příklad velmi nekvalitního překladu

Vizualizace

Všechny modely kromě LSTM RNN umožňují vizualizaci pozornosti. Ta má podobu matice, ve které je vidět které tokeny z originální věty nejvíce přispěly k vytvoření jednotlivých výstupních tokenů. U Mého modelu lze znázornit to samé i pro sebe-pozornost, kde však místo výstupních tokenů vznikají upravené zdrojové tokeny.

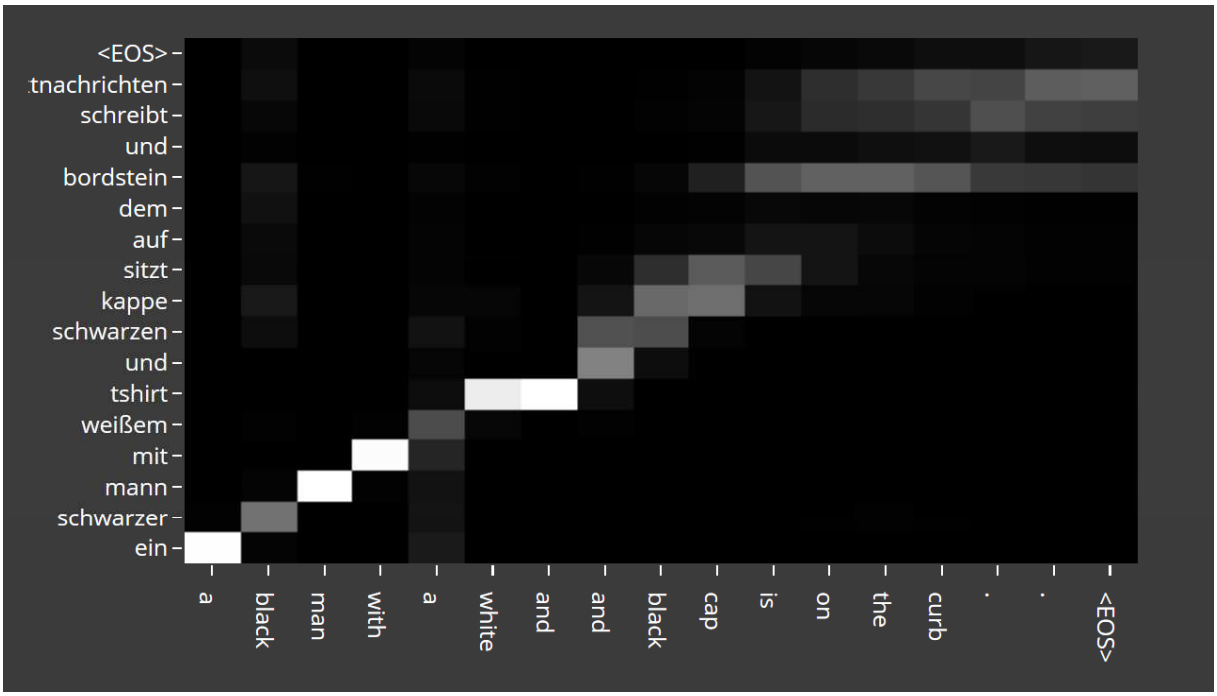
RNNsearch



Obrázek 30: Matice pozornosti modelu RNNsearch

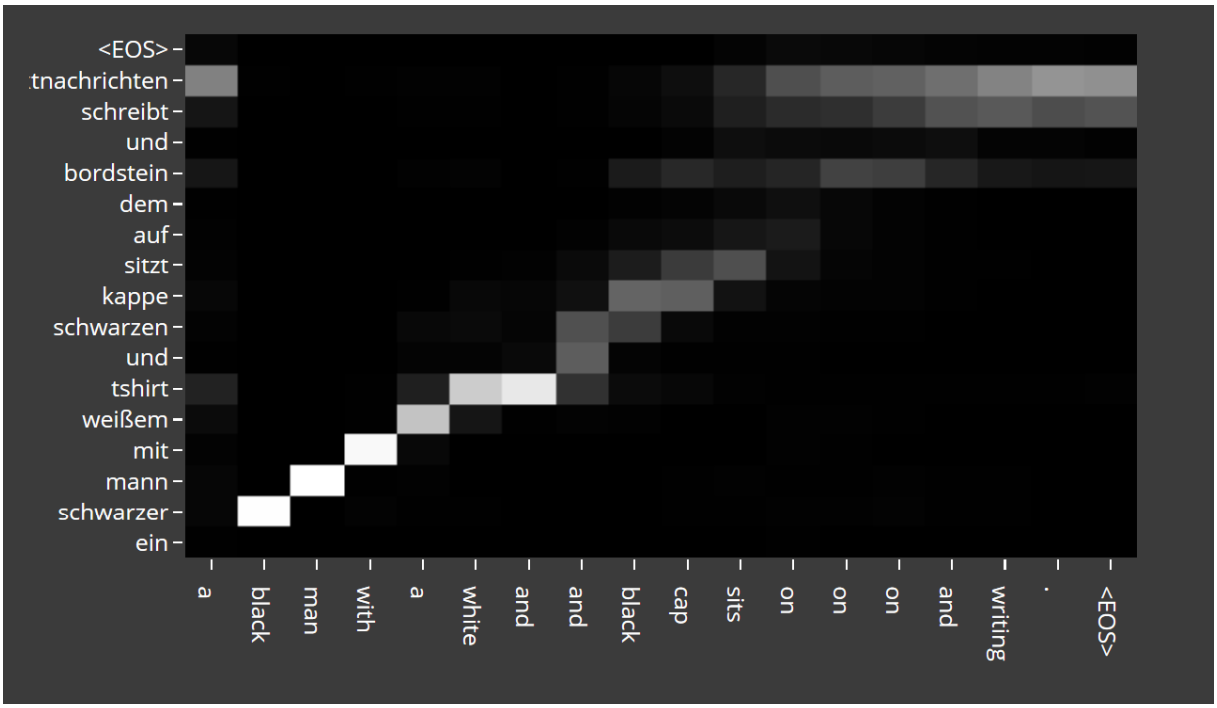
ConvS2S

Model ConvS2S využívá pozornost ve všech výstupních vrstvách, získáváme tak několik matic pozornosti. Zde si uvedeme pro příklad jednu, v příloze budou k dispozici všechny.



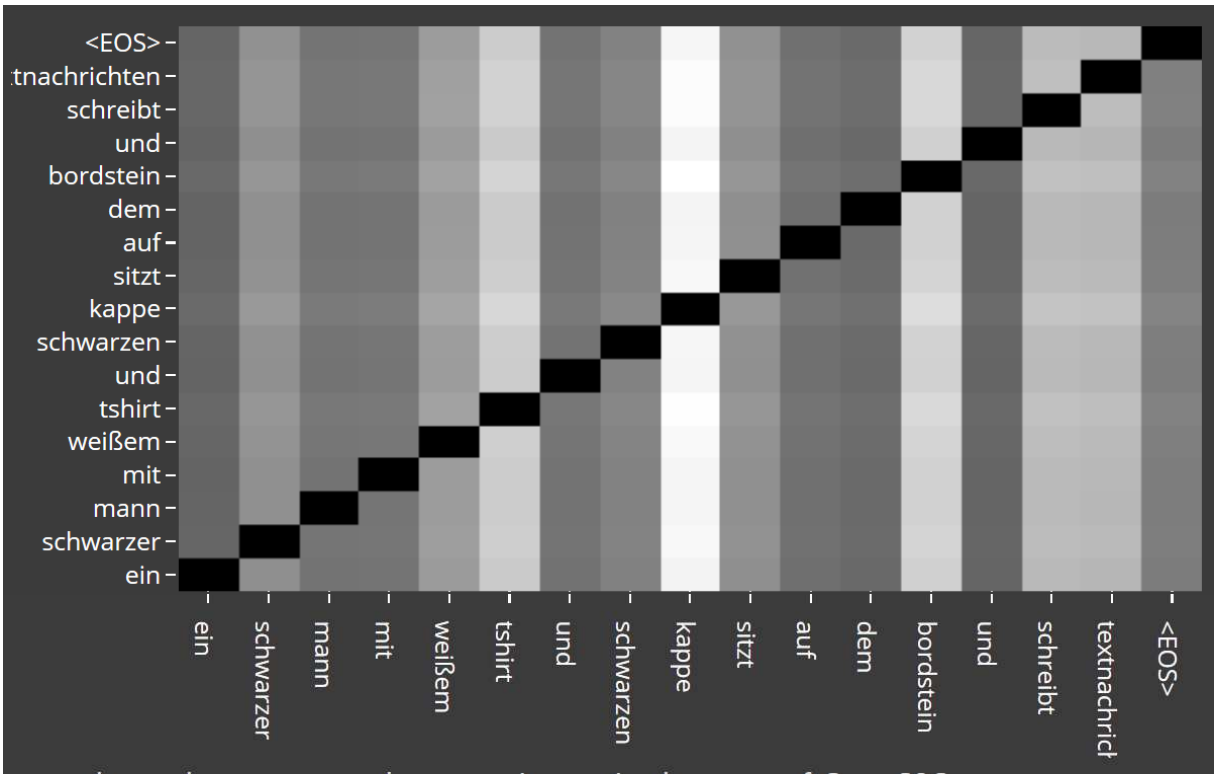
Obrázek 31: Matice pozornosti modelu ConvS2S

Můj model



Obrázek 32: Matice pozornosti mého modelu

Můj model navíc využívá sebe-pozornost, čímž získáváme další matici pozornosti navíc.



Obrázek 33: Matice sebe-pozornosti mého modelu

Produkce

Jako produkční prostředí jsem zvolil web. Důvody jsou jasné – projekt je tímto způsobem jednoduše dostupný a lze si jej vyzkoušet i bez kopírování kódu, modelů a dat. Je spousta různých způsobů, jak webové stránky realizovat, v této části bych se tedy rád zabýval tím, jak jsem k této problematice přistoupil.

Obrázek 34: Domovská stránka webu

Frontend

Kromě interaktivního překladu a vizualizace je jednou z funkcí uživatelského rozhraní i představení a stručný popis práce. Stránky jsou rozděleny podle oblastí projektu, kterou se zabývají:

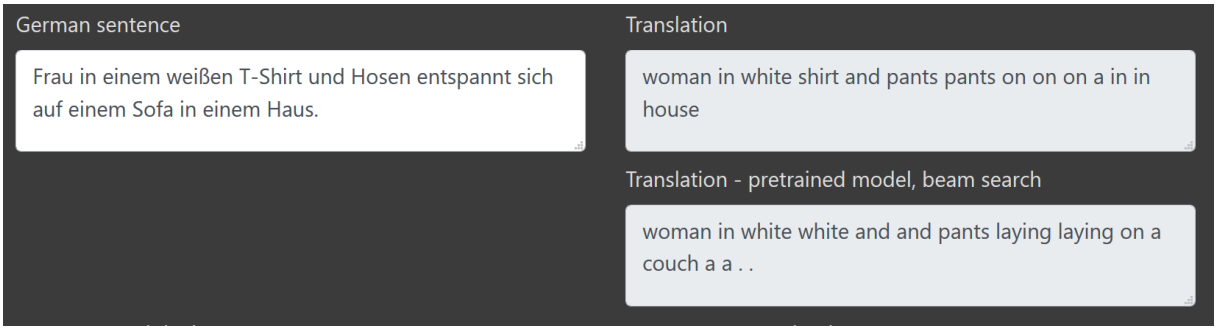
- Domovská stránka
- Úvod
- Data
- Modely
    - LSTM RNN
    - RNNsearch
    - ConvS2S
    - Můj model
- Závěr

Primární funkcí stránek je však samotný překlad pomocí jednotlivých modelů. Z tohoto důvodu, ale také proto, že informace uvedené na stránkách jsou zároveň uvedené i v této dokumentaci, se budeme v této části věnovat téměř výhradně procesu překladu.

Vytvoření překladů předem

Na stránkách se vždy zobrazuje překlad náhodné věty z validační části souboru dat. Jelikož máme k těmto větám přístup již nyní, můžeme si dopředu připravit jejich překlady. Takto zamezíme časové prodlevě, která by vznikla, pokud bychom tuto větu překládali až při návštěvě webových stránek uživatelem.

Tyto překlady (i s veškerými daty potřebnými pro vizualizace) uchováváme v .tsv souborech. Jsou seřazeny tak, že jejich pořadí odpovídá pořadí vět v souboru se zdrojovými větami, takže lze překlady jednoduše vyhledávat pomocí jejich indexů.



Obrázek 35: Překlad věty v produkci

Překlad uživatelem zadané věty

Při zadávání nové věty si lze všimnout okamžité reakce uživatelského rozhraní, kde se místo původního překladu zobrazí symbol načítání. Po jakékoliv změně textu ve vstupním poli začne běžet dvousekundový odpočet, který se obnoví, pokud se text během tohoto intervalu znovu změní. Pokud k dalším změnám nedojde, nová věta se pošle na server pro překlad.

Abychom tuto funkcionalitu mohli implementovat, využíváme asynchronní volání funkce pro překlad. Při každém volání této funkce zaznamenáme novou podobu věty, aktualizujeme stav fronty a začne odpočet intervalu. Poté, co interval skončí, zkontrolujeme stav fronty – pokud se ve frontě nenalézá žádná novější věta, odesíláme současnou větu k překladu, jinak ji zahazujeme.

Komunikace se serverem

Komunikaci mezi klientem a serverem realizujeme pomocí funkce *fetch.* V této aplikaci rozeznáváme dva typy klientských požadavků na server:

- Počáteční získání náhodného páru vět s jejich překlady
- Překlad uživatelem zadané věty

V prvním případě nejprve požádáme server o zdrojovou větu a vzorový překlad, přičemž získáme i index tohoto páru. Následují požadavky na překlad této věty pomocí jednotlivých modelů. Požadavky obsahují index věty, takže jí není nutné znovu překládat a stačí pouze vyhledat příslušné překlady.

```
const proxy = "https://neuralmachinetranslation.rocks/api";
…
fetch(`${proxy}/data`)
       .then(res => res.json())
       .then(res => this.changeAllSentences(res));
```

Figure 4: Použití funkce fetch() pro překlad věty při načtení stránek

Ve druhém případě odešleme serveru požadavek obsahující zdrojovou větu a použitý model. Na místo indexu vložíme hodnotu „null“, abychom na backendu byli schopni zjistit, že se jedná o překlad nové věty.

Zobrazení dat

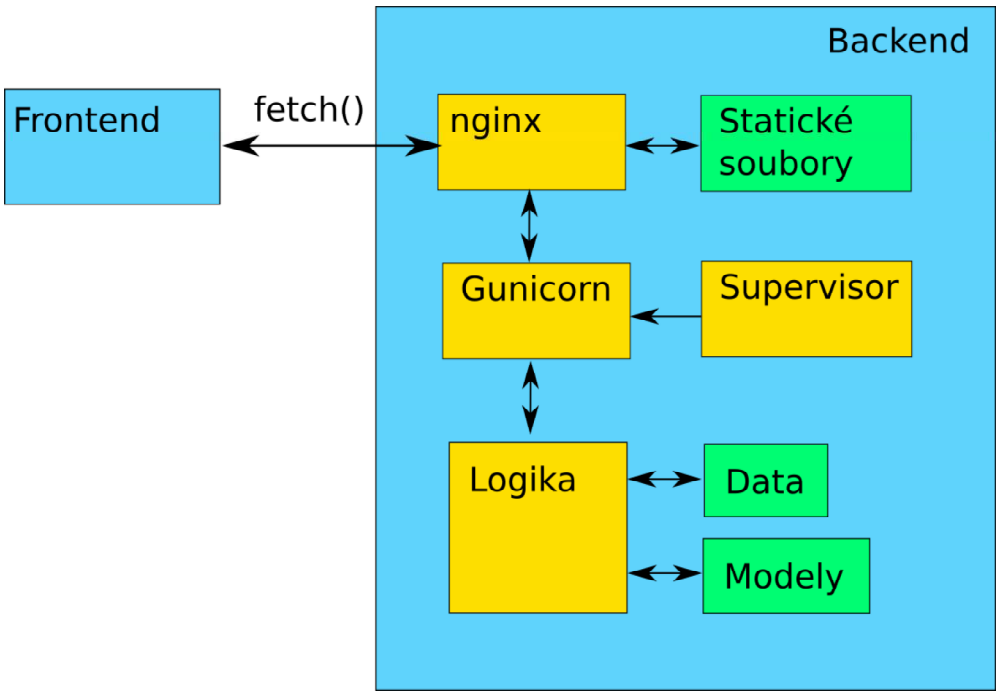
Překlady vět jsou zobrazované v polích bez se zablokovanou editací. U některých modelů lze zobrazit i vizualizace – matice pozornosti. Potřebujeme tedy nějaký způsob, jak je vykreslit. Jako nástroj jsem využil react-plotly – knihovnu pro vykreslování grafů určenou pro React.

Backend

Funkcemi backendu je poskytovat validační data spolu s připravenými překlady a překládat věty nové. V zásadě se jedná o server směrující žádosti klienta a odpovědi, získané skrze logiku backendu.

Struktura

Je možné, že se uživatel bude chtít připojit ke klientovi přes port 80 (http). Jelikož však chceme, aby naše aplikace byla zabezpečená, používáme port 443 (https). Přesměrování z portu 80 na port 443 řeší program *nginx*, který zároveň zajišťuje i poskytování přístupu ke statickým souborům.



Obrázek 36: Struktura aplikace

Je potřeba aby server neustále naslouchal a správně předával požadavky logice backendu. Zároveň musí umět zpracovat i více požadavků najednou. Tyto úkoly řeší *gunicorn*.

V případě neočekávaného restartu serveru je nutné obnovit veškeré procesy. K tomu slouží program *Supervisor*, který zajištuje nepřerušovaný chod aplikace na serveru. Narozdíl od *gunicorn* se *nginx* zapíná při startu sám, pro udržení jeho chodu tedy *Supervisor* nepoužíváme.

Sdružením těchto funkcí získáváme kompletní backend aplikace. Jeho blokové schéma lze vidět na Obrázku 36.

Činnost

Při startu serveru se načtou modely a datová struktura obsahující validační data. Při požadavku na příklad věty z validační části dat se vybere náhodný index, pomocí něhož se získá věta z datové struktury. Věta, její vzorový překlad a index jsou potom odeslány zpět klientovy.

Další požadavky na překlad jsou realizovány dvěma různými způsoby na základě toho, jestli je jejich součástí i index věty. Pokud ano, příslušné překlady jsou získány ze souborů, pokud ne, věta je přeložena pomocí modelů načtených v paměti.

Závěr

V současné době se strojové učení stalo velmi dostupnou disciplínou. Její myšlenku zpracovává mnoho autorů v podáních jak přístupných široké veřejnosti, tak i velmi rigorózních. Obrovské množství dat je veřejně dostupné. Výpočetní zdroje jsou levné. Každý si tak může vyzkoušet vytvořit program, který lze něco naučit.

Samozřejmě je otázkou, jak velké využití tato technologie bude mít. Množství praktických aplikací již existuje – radiologie, překlad, autonomní vozidla, … Neurální sítě však ještě zdaleka nevládnou tolika oblastem, kolik jim moderní publicistika má tendenci přisuzovat. Další velmi zajímavou oblastí je umělá obecná inteligence – vyvineme někdy program, který bude schopný se svým chováním podobat člověku?

V této práci jsme si vyzkoušeli aplikovat neuronové sítě na překlad mezi dvěma lidskými jazyky. Od počáteční úpravy dat, přes implementaci a trénink modelů jsme se dostali až k samotnému překladu a nasazení do produkce. Zároveň jsme měli možnost si ověřit tvrzení uváděná v současné literatuře. V praxi bychom na základě výsledků, získaných pomocí evaluace, mohli mezi modely vybrat ten nejvhodnější a použít ho pro požadovanou aplikaci.

Osobně jsem byl kvalitou překladu zklamán. Ve chvíli, kdy je vstupní věta delší, než několik slov nebo obsahuje méně známá slova, mají modely problém větu přeložit. Občas ani jednoduché věty nejsou schopné přeložit.

V průběhu tvorby tohoto projektu mě napadlo několik způsobů, jak zlepšit kvalitu překladu – některé jsem stihl implementovat (beam search), některé už bohužel ne. Jedním z těch, na které mi už nezbyl čas, je natrénování neurální sítě jako jazykového modelu.

Jazykový model má za úkol předpovědět další token na základě tokenů předchozích. Pokud bychom tímto způsobem natrénovali jak kodér, tak dekodér (každý z nich jako jazykový model příslušného jazyka), mohli bychom následně pouze vyměnit výstupní vrstvu sítě a dotrénovat na dvojjazyčném souboru dat. Tímto způsobem totiž kodér i dekodér dopředu vědí, jaká slova patří do jakého kontextu, a tedy už pouze jenom víceméně hledají, jakou formu má daná fráze v jiném jazyce. Zda-li by se však toto uspořádání skutečně osvědčilo zůstává otázkou.

I když tomu výsledky v této práci nemusí úplně odpovídat, zastávám názor, že strojový překlad bude v budoucnosti realizován výhradně neurálními sítěmi (pokud tedy nevznikne nová rodina algoritmů, která by neurální sítě svými schopnostmi překonala). Výsledky, které ve svých publikacích autoři modelů prezentují, vypadají totiž velmi slibně, a od roku 2016 využívá k překladu neuronové sítě i Google Translate[3].

Jako velmi zajímavou oblast strojového překladu nyní vnímám překlad „bez dozoru“ [10]. Místo toho, abychom modelu ukázali překlady vět, vytvoříme jazykové modely daných jazyků a hledáme, která slova mají podobné vztahy vůči ostatním slovům. Velkou výhodou této metody je, že k natrénování nepotřebujeme páry přeložených vět – stačí nám jeden soubor dat textu v jednom jazyce a jeden soubor dat textu v druhém jazyce.

[3] https://blog.google/products/translate/found-translation-more-accurate-fluent-sentences-google-translate/

Použitá literatura

1. **David E. Rumelhart, Geoffrey E. Hinton & Ronald J. Williams.** Learning representations by back-propagating errors. *Nature.* Letters to nature, 1986, Vol. 323, .

2. **Robbins, Herbert and Monro, Sutton.** A Stochastic Approximation Method. *The Annals of Mathematical Statistics.* 1951, Vol. 22, 3.

3. **Bengio, Yoshua, et al.** A neural probabilistic language model. *The Journal of Machine Learning Research.* 2003, Vol. 3.

4. *Sequence to Sequence Learning with Neural Networks.* **Le, Ilya Sutskever and Oriol Vinyals and Quoc V.** Montreal, CA : Proc. NIPS, 2014.

5. *Automatic differentiation in PyTorch.* **Paszke, Adam and Gross, Sam and Chintala, Soumith and Chanan, Gregory and Yang, Edward and DeVito, Zachary and Lin, Zeming and Desmaison, Alban and Antiga, Luca and Lerer, Adam.** 2017.

6. **Elliott, Desmond and Frank, Stella and Sima'an, Khalil and Specia, Lucia.** *Multi30K: Multilingual English-German Image Descriptions.* Berlin, Germany : Association for Computational Linguistics, 2016.

7. *In Proc. of EAMT.* **M. Cettolo, C. Girardi, and M. Federico.** Trento, Italy : WIT3: Web Inventory of Transcribed and Translated Talks, 2012.

8. **Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio.** Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473.* 2014.

9. *Convolutional Sequence to Sequence Learning.* **Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, Yann N. Dauphin.** 2017.

10. *Attention Is All You Need.* **Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin.** s.l. : NIPS, 2017.

11. **Lample, Guillaume and Denoyer, Ludovic and Ranzato, Marc'Aurelio.** Unsupervised Machine Translation Using Monolingual Corpora Only. 2017.

12. **Qef.** File:Logistic-curve.svg. *Wikimedia.* [Online] 7 4, 2014. [Cited: 3 7, 2019.] https://commons.wikimedia.org/wiki/File:Logistic-curve.svg.

Seznam obrázků

Seznam figur

Seznam tabulek

Přílohy

Kód, modely, data

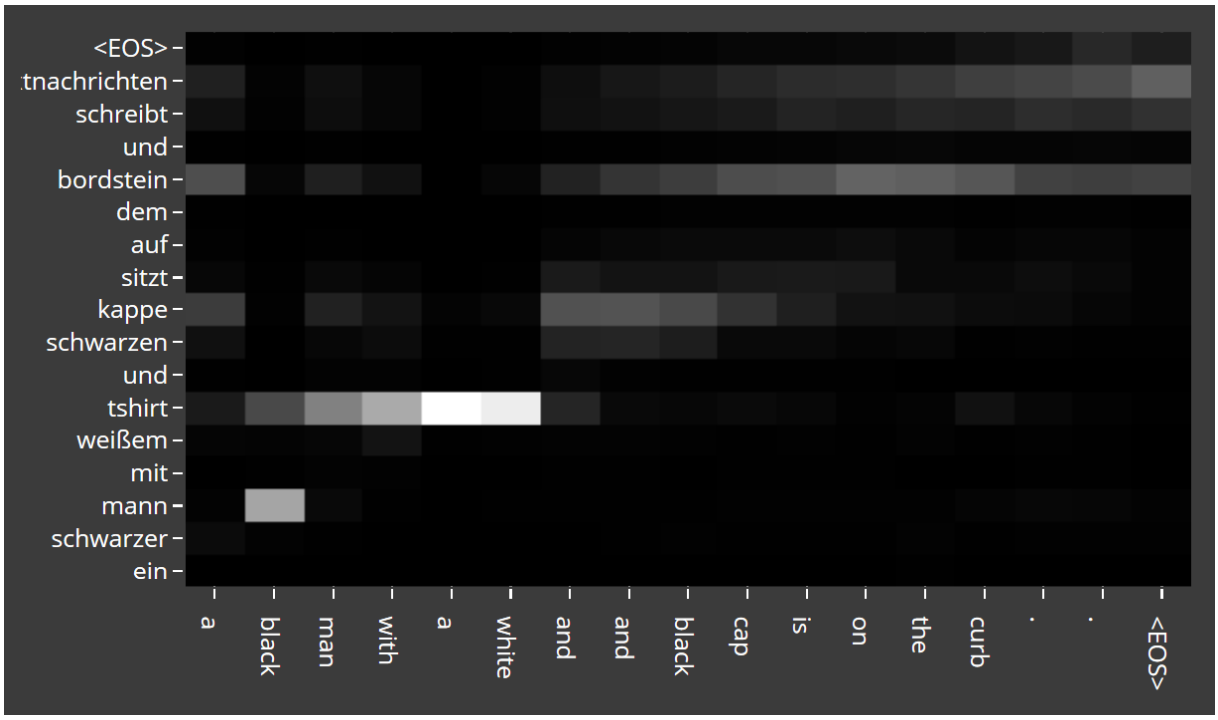
Rozhodl jsem se, že veřejně dostupný bude pouze kód této práce. Lze se jím inspirovat, není však ve formě, ze které by bylo jednoduché tuto práci reprodukovat. Modely a data tvoří totiž poměrně velké soubory (v řádech desítek až stovek MB) a samotná práce je značně členitá.

Kód práce - https://1drv.ms/f/s!AiQ5a2cXVytTm0lSx-WTdfFxry7l

V případě zájmu o celou práci mě prosím kontaktuje emailem – janrumik@gmail.com

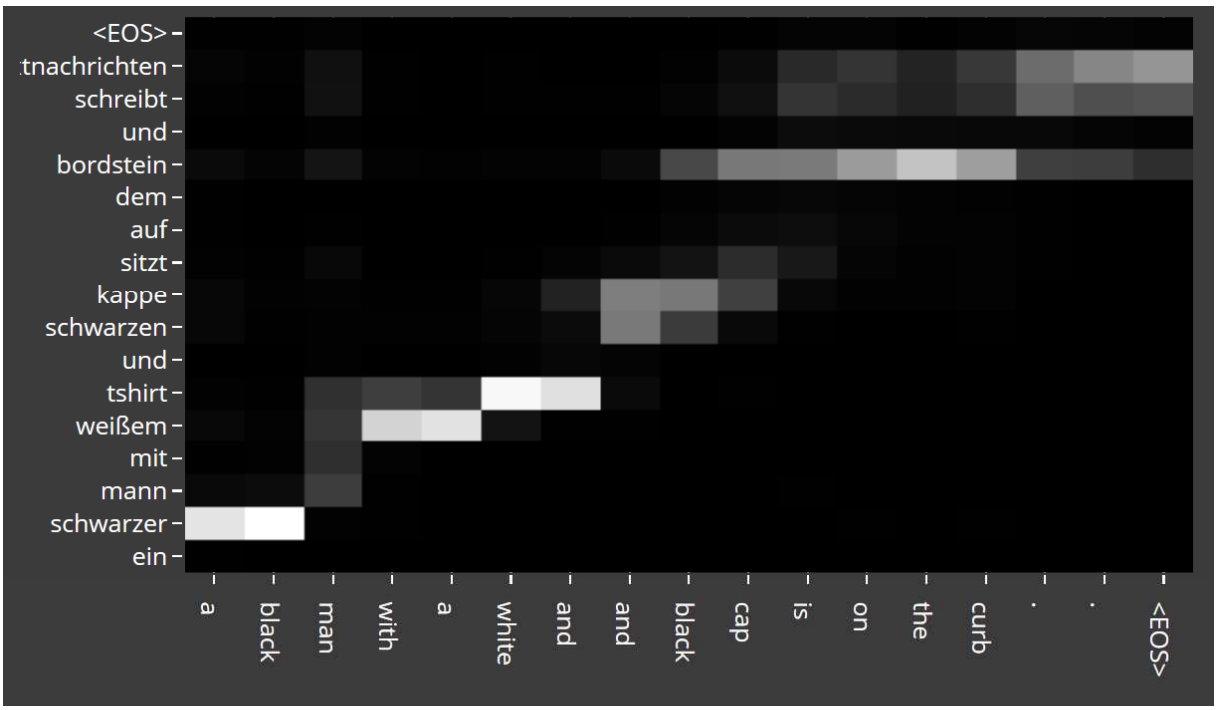
Matice pozornosti – ConvS2S

Vrstva 1



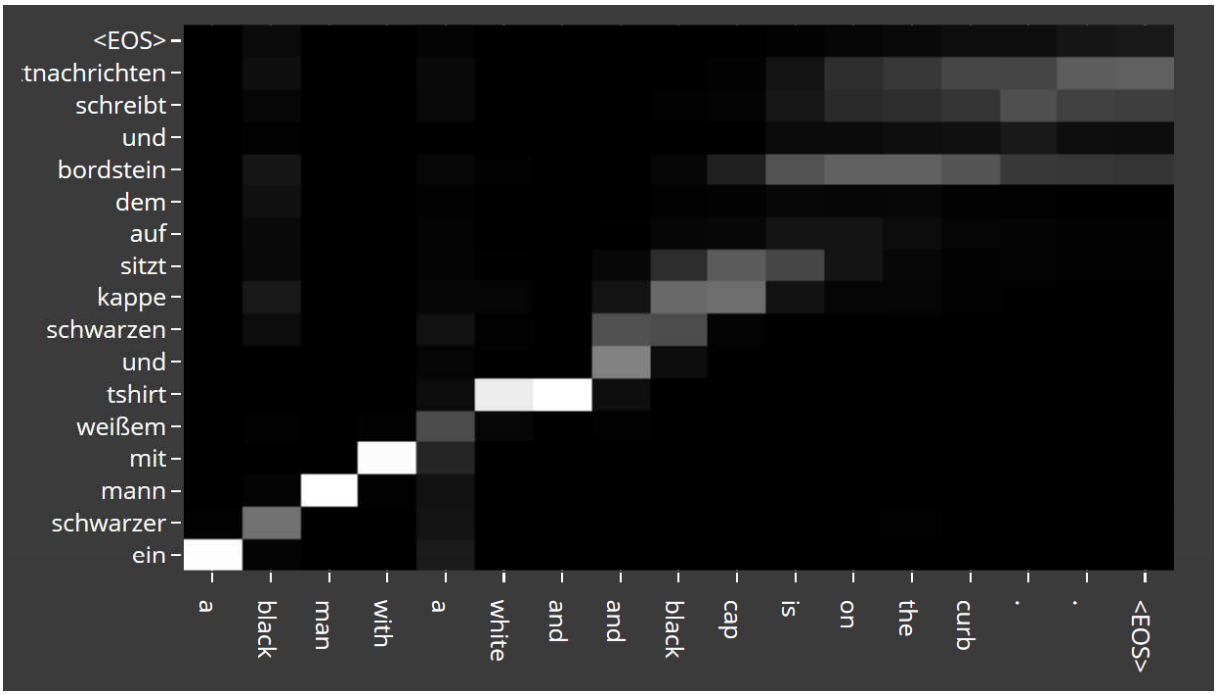
Obrázek 37: Matice pozornosti ConvS2S, vrstva 1

Vrstva 2



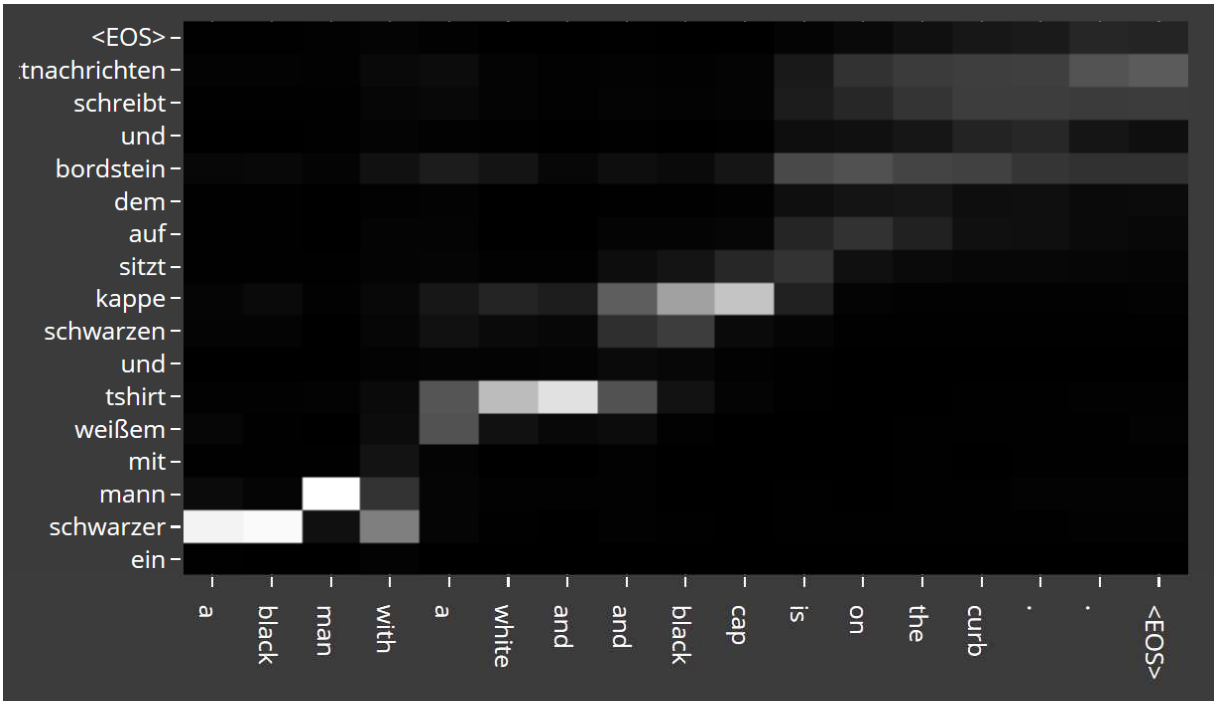
Obrázek 38: Matice pozornosti ConvS2S, vrstva 2

Vrstva 3



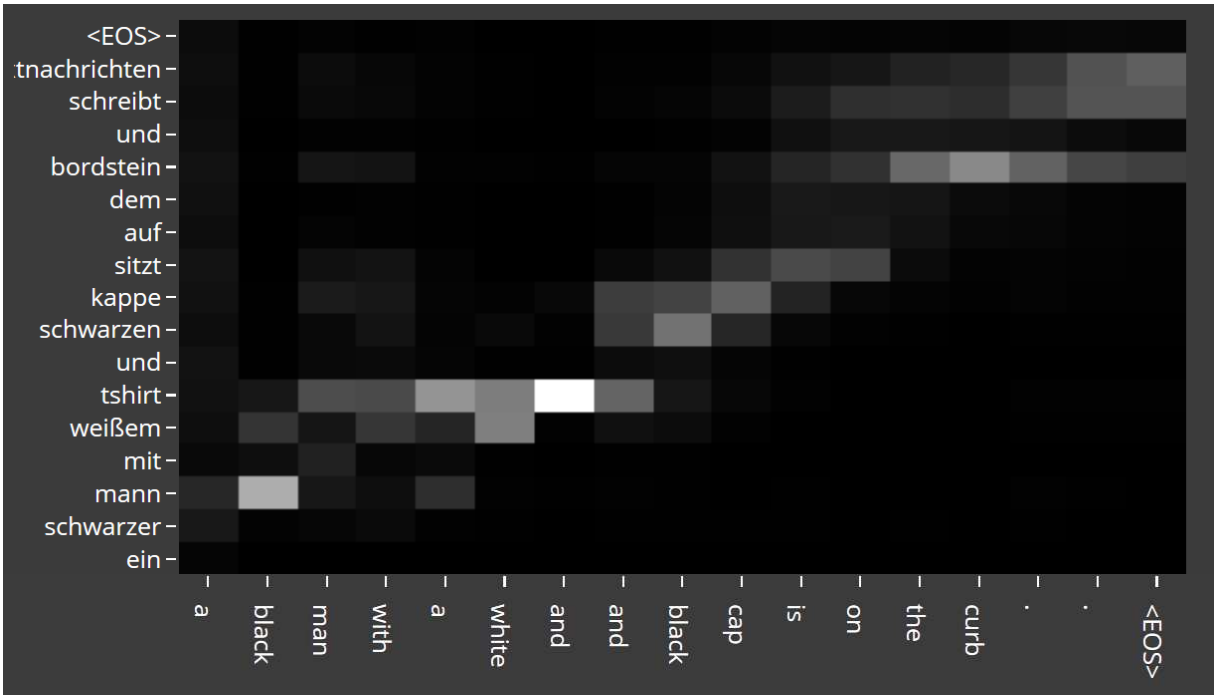
Obrázek 39: Matice pozornosti ConvS2S, vrstva 3

Vrstva 4



Obrázek 40: Matice pozornosti ConvS2S, vrstva 4

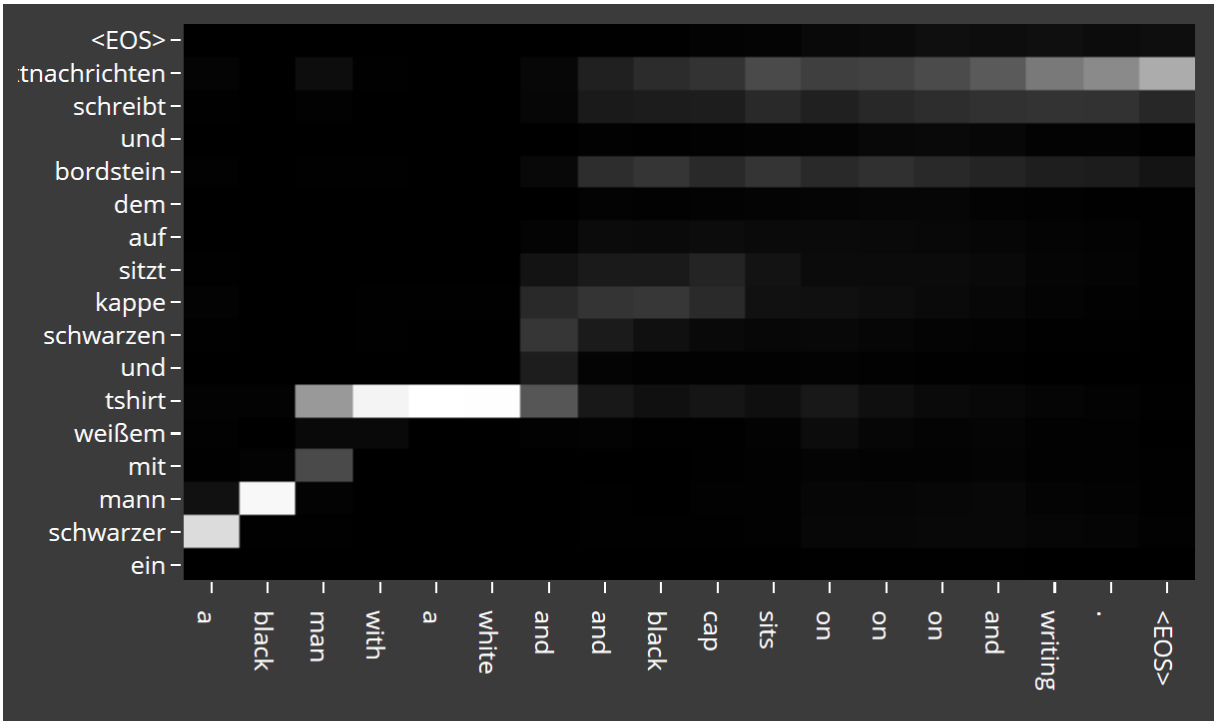
Vrstva 5



Obrázek 41: Matice pozornosti ConvS2S, vrstva 5

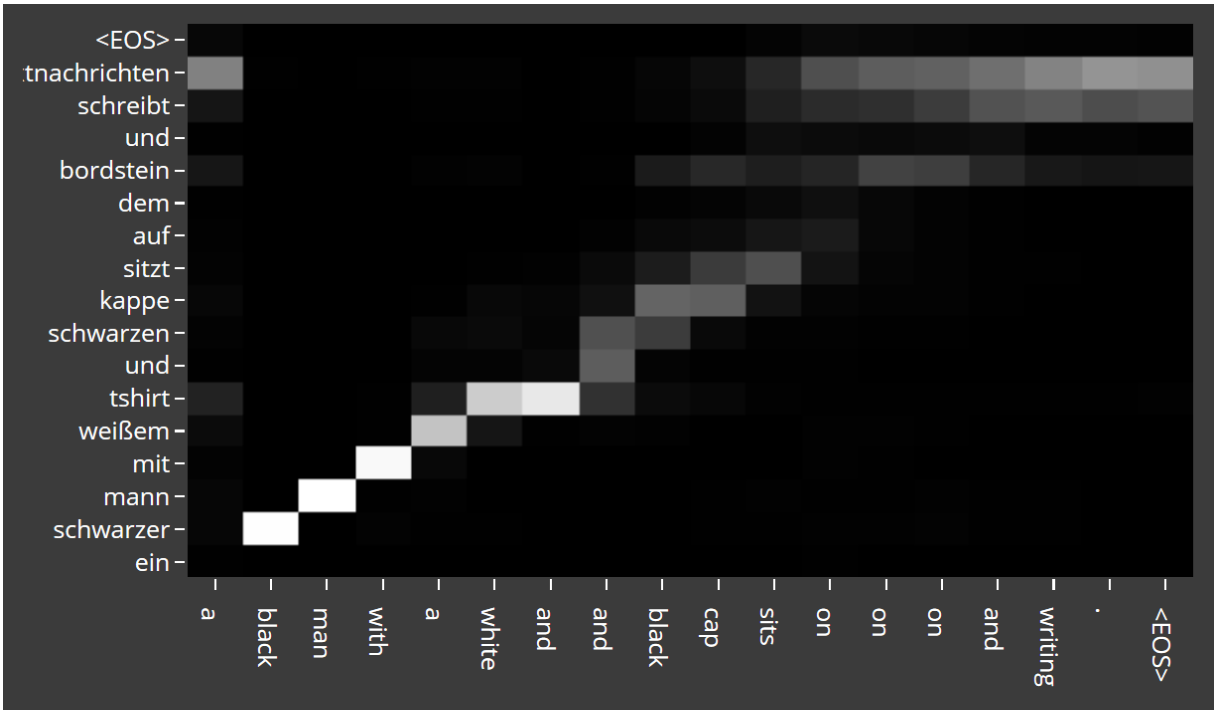
Matice pozornosti – Můj model

Vrstva 1



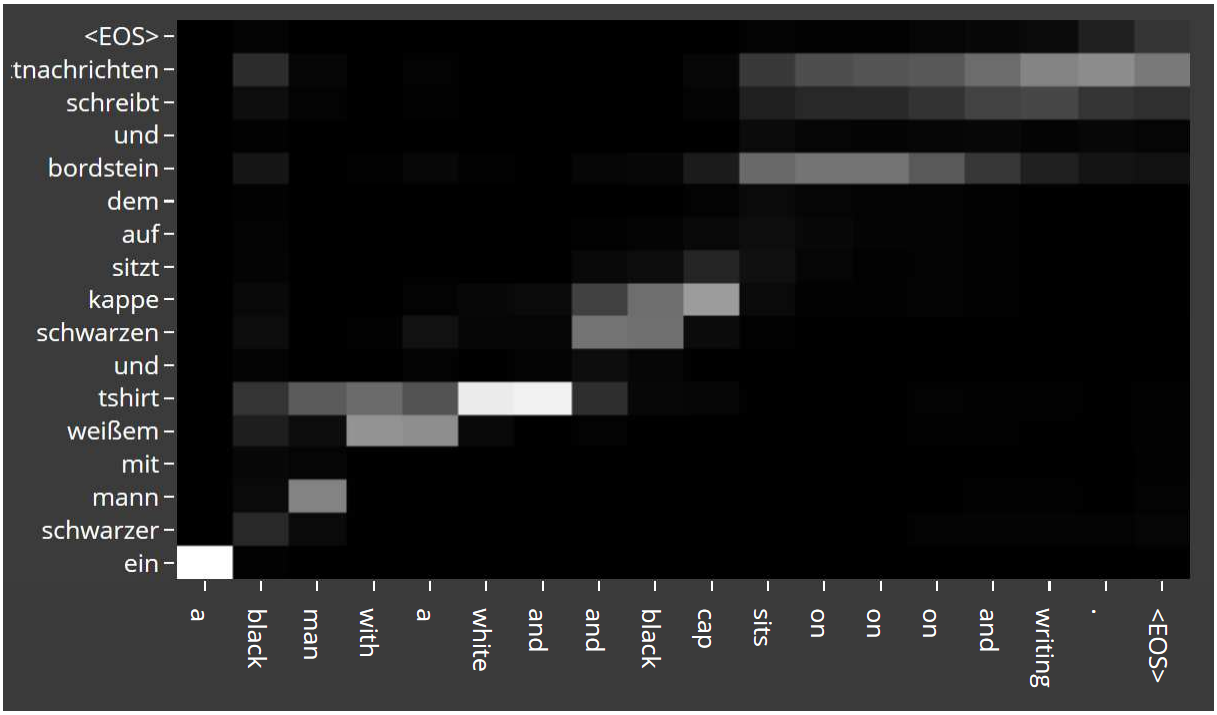
Obrázek 42: Matice pozornosti mého modelu, vrstva 1

Vrstva 2



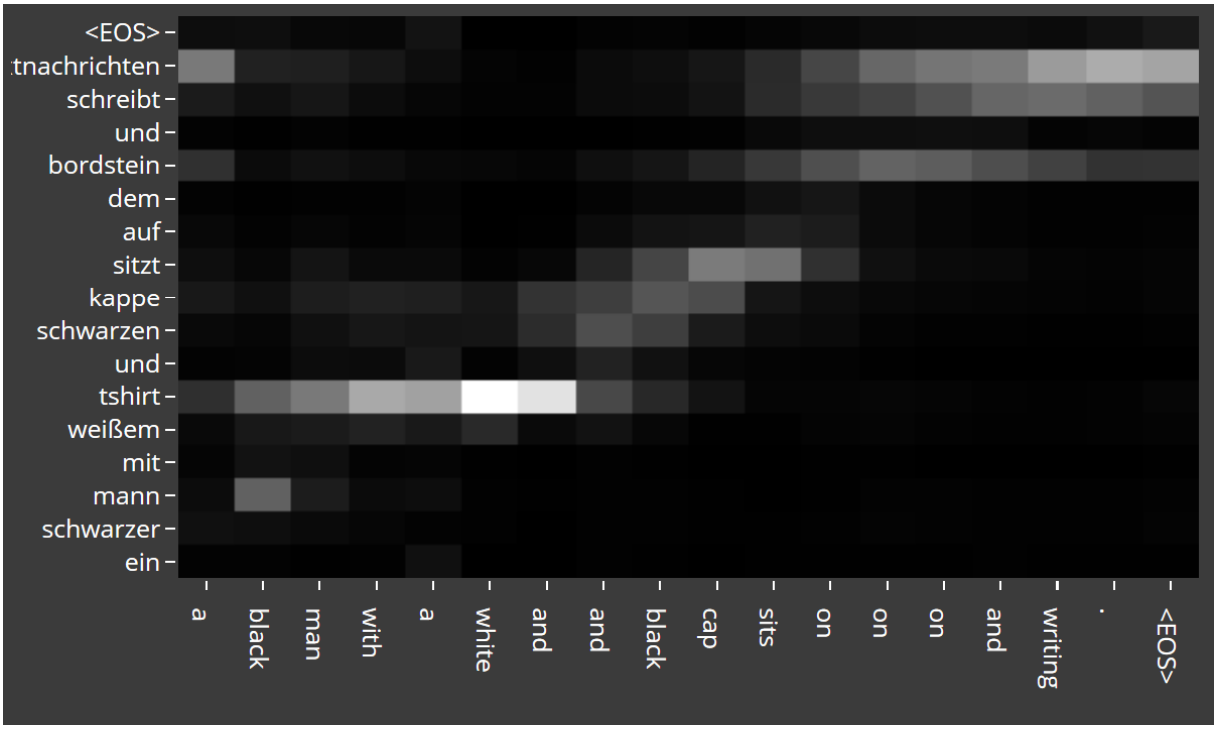
Obrázek 43: Matice pozornosti mého modelu, vrstva 2

Vrstva 3



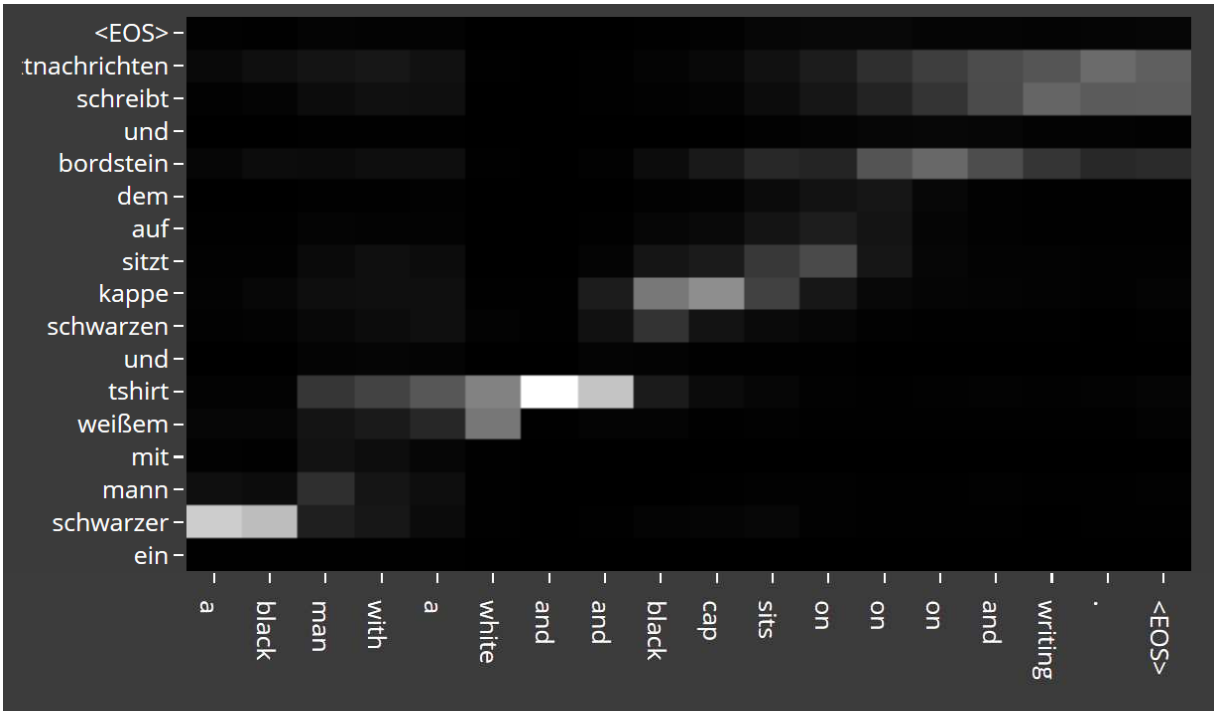
Obrázek 44: Matice pozornosti mého modelu, vrstva 3

Vrstva 4



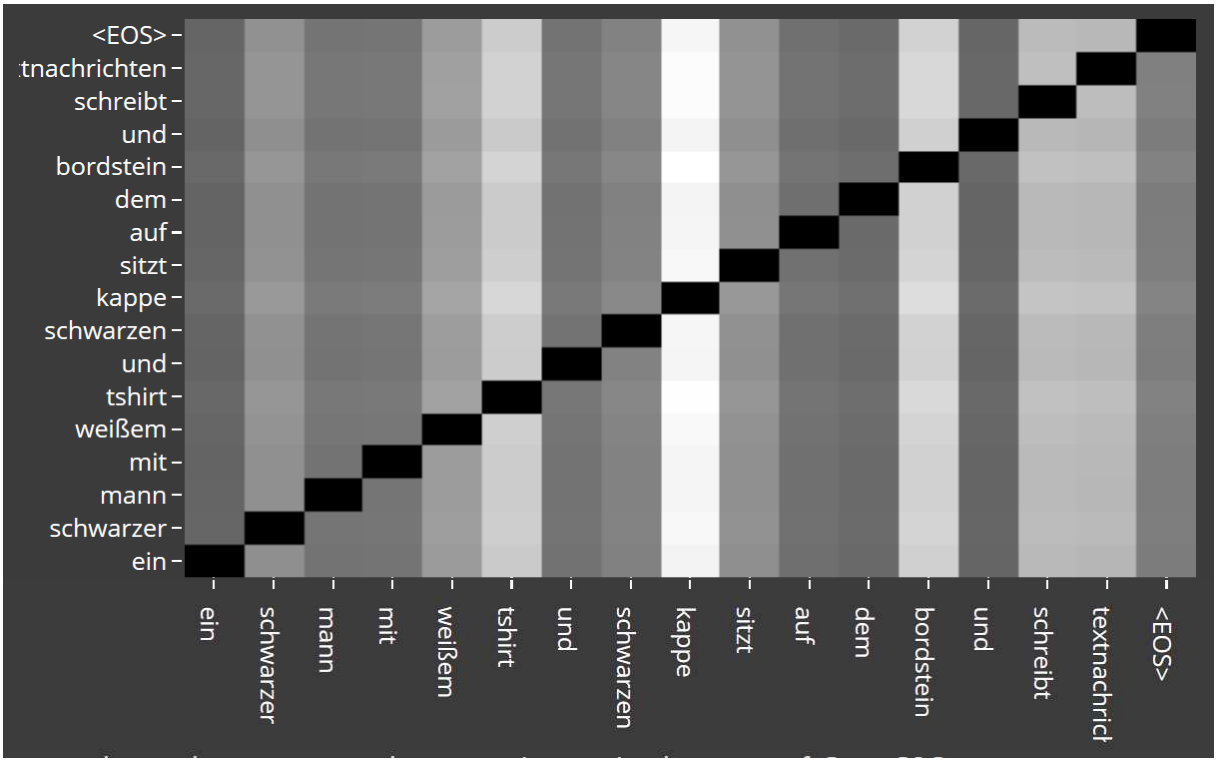
Obrázek 45: Matice pozornosti mého modelu, vrstva 4

Vrstva 5



Obrázek 46: Matice pozornosti mého modelu, vrstva 5

Sebe-pozornost



Obrázek 47: Matice sebe-pozornosti mého modelu