Sieci neuronowe w systemach dialogowych

(Neural Networks in Dialogue Systems)

Maciej Pawlikowski

Praca magisterska

Promotor: dr Jan Chorowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

27 marca 2018

Załącznik Nr 1
do Zasad

.....
(imiona i nazwisko)

....
(aktualny adres do korespondencji) (numer PESEL)

.....
(adres e-mail)

.....
(wydział)

.....
(kierunek studiów)

....
(poziom i forma studiów) (numer albumu)

OŚWIADCZENIE O PRAWACH AUTORSKICH I DANYCH OSOBOWYCH

Ja niżej podpisany/a ..... student/ka Wydziału ..... kierunek ..... oświadczam, że przedkładana praca dyplomowa na temat:.....
.....
.....

- jest mojego autorstwa i nie narusza autorskich praw w rozumieniu ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (tekst jednolity: Dz. U. z 2017 r. poz. 880, z późn. zm.) oraz dóbr osobistych chronionych prawem;
- nie zawiera danych i informacji uzyskanych w sposób niedozwolony;
- nie była wcześniej przedmiotem innej urzędowej procedury związanej z nadaniem dyplomu uczelni wyższej lub tytułu zawodowego;
- treść pracy dyplomowej załączona w wersji elektronicznej w APD, jest identyczna z jej wersją drukowaną.

Oświadczam, iż zostałem/am poinformowany/a o prawie dostępu do treści moich danych osobowych oraz ich poprawiania.

Wyrażam zgodę, na:

- udostępnienie mojej pracy dla celów naukowych i dydaktycznych;

☐ TAK ☐ NIE

- przetwarzanie moich danych osobowych w myśl ustawy z dnia 29 sierpnia 1997 r. o ochronie danych osobowych (tekst jednolity: Dz. U. z 2016 r., poz. 922.) w zakresie wynikającym z niniejszego oświadczenia i w celu jego realizacji.

Wrocław, ...

(rrrr – mm - dd) (czytelny podpis autora pracy)

Streszczenie

Praca stanowi przedstawienie niektórych mechanizmów neuronowych wykorzystywanych przy tworzeniu systemów dialogowych, czyli programów komputerowych potrafiących prowadzić rozmowę z użytkownikiem. Pokazuję również jak opisywane metody sprawdzają się w praktyce.

Zaimplementowałem i przetestowałem algorytmy do statystycznego modelowania oraz generowania dialogu. Eksperymentowałem także z architekturą odpowiadającą na pytania o fakty zadawane w języku naturalnym. Do tej ostatniej części dołączam moją próbę rozszerzenia możliwości algorytmu.

In this thesis I present selected neural mechanisms used in creating dialogue systems, or computer programs designed to hold a conversation with a user. I also show how methods described here perform in practice.

I implemented and tested algorithms for statistical modeling of a dialogue, as well as dialogue generation. In addition, I experimented with an architecture able to answer questions formulated in natural language. This part also contains my attempt of expanding the capabilities of the system.

Spis treści

1. Wstęp **7**

1.1. Cel i zawartość pracy 8

2. Sieci neuronowe **9**

2.1. Wprowadzenie 9

2.2. Uczenie sieci neuronowej 10

2.2.1. Propagacja wsteczna 10

2.2.2. Zbiór uczący i zbiór testowy 11

2.2.3. Metoda gradientu prostego 13

2.2.4. *Adaptive Moment Estimation* 13

2.3. Warstwy 14

2.3.1. Warstwa afiniczna 15

2.3.2. *Softmax* 15

2.3.3. Próbkowany *softmax* 16

2.3.4. Warstwa rekurencyjna 18

2.3.5. *Long Short-Term Memory* 18

2.3.6. *Gated Recurrent Unit* 19

2.3.7. Dwukierunkowa *RNN* 20

3. Neuronowy model języka **21**

3.1. Modelowanie występowania słów 21

3.2. Architektura sieci 21

3.3. Proces uczenia 23

3.4. Reprezentacje wektorowe 23
3.5. Generowanie tekstu 25
3.5.1. Przeszukiwanie wiązkowe 26

4. Generowanie dialogu **29**
4.1. Dostępność danych 29
4.2. Model hierarchiczny 31
4.2.1. Architektura *encoder-decoder* 32
4.2.2. Hierarchiczny rekurencyjny *encoder-decoder* 33
4.3. Eksperymenty 36
4.3.1. Generowanie wypowiedzi 37

5. Odpowiadanie na pytania **43**
5.1. FastQA 44
5.1.1. Architektura sieci 44
5.2. Eksperymenty 48
5.2.1. Przykłady zastosowania 48
5.3. Negatywne odpowiedzi 50
5.3.1. Wzbogacanie danych 52
5.3.2. Wyniki 53
5.3.3. Przykłady negatywnych odpowiedzi 53

6. Podsumowanie **57**

Bibliografia **58**

Rozdział 1.

Wstęp

System dialogowy (inaczej *chatbot*) to program komputerowy umożliwiający porozumienie w języku naturalnym między człowiekiem a maszyną. Możliwość naturalnej rozmowy z komputerem była kiedyś utożsamiana ze sztuczną inteligencją. Alan Turing, w swojej słynnej publikacji [Turing, 1950], pytał czy da się napisać program konwersacyjny, którego ludzki obserwator nie byłby w stanie odróżnić od człowieka. Eksperyment ten dzisiaj nosi miano testu Turinga. Jego obecną formą jest Nagroda Loebnera – coroczny konkurs, w którym twórcy programów starają się oszukać sędziów. Dotychczas wręczano jedynie nagrodę dla najlepszego chatbota. Nagrody głównej, jednorazowo przyznawanej programowi nieodróżnialnemu od człowieka, nie zdobył jeszcze żaden bot.

Przejście testu Turinga nie jest jednak celem każdego systemu dialogowego. Dzisiejsze chatboty mogą mieć bardzo różne przeznaczenie. Obecna w systemie iOS Siri[1] stanowi interaktywną pomoc dla użytkownika. Zastosowanie wykrywania mowy umożliwia wydawanie urządzeniu poleceń głosowych w języku naturalnym. Robot będący częścią portalu Visit Wroclaw[2] pełni rolę informacji turystycznej. Bardzo popularny Cleverbot[3] cały czas uczy się lepiej mówić wykorzystując historię przeprowadzanych dialogów. Jest to system ogólnego przeznaczenia, potrafiący rozmawiać na wiele tematów. W biznesie chatboty często pełnią rolę interaktywnych asystentów, pomagając znaleźć połączenia lotnicze, zarezerwować miejsca w restauracji, czy zapoznać się z ofertą firmy.

Techniczne aspekty konstrukcji chatbotów również mogą się drastycznie różnić. Napisany w 1995 roku system *Artificial Linguistic Internet Computer Entity* (w skrócie A.L.I.C.E. lub Alice) [Wallace, 2009] formułuje odpowiedzi wykorzystując dopasowanie wzorca i wypełnianie luk w schematach. Alice wymaga dużej liczby ręcznie utworzonych reguł, co ją mocno ogranicza. Obecnie często eksperymentuje się z wykorzystaniem metod uczenia maszynowego do budowania botów dialogo-

[1] `https://www.apple.com/ios/siri/`
[2] `https://www.facebook.com/visitwro/`
[3] `http://www.cleverbot.com/`

wych. Szczególnie popularne są sieci neuronowe, które w ostatnich latach znacznie zwiększyły nasze możliwości w dziedzinie przetwarzania języka naturalnego.

1.1. Cel i zawartość pracy

Celem tej pracy jest przedstawienie niektórych mechanizmów neuronowych znajdujących zastosowanie w systemach konwersacyjnych. Rozdział 2. stanowi wstęp do tematu sieci neuronowych i wprowadzenie niezbędnej nomenklatury. Kolejny opisuje statystyczne modelowanie języka, które jest podstawą sieci pracujących na języku naturalnym. Następne dwa rozdziały są główną częścią pracy. Rozdział 4. pokazuje architekturę generującą wypowiedzi w dialogu korzystając z pojedynczych słów. Zamieszczam tam także przykłady otrzymanych dialogów. Rozdział 5. jest opisem modelu potrafiącego wyciągać z tekstu odpowiedzi na pytania. Zawiera on też moją próbę rozszerzenia oryginalnej funkcji algorytmu, wraz z wynikami. Do pracy dołączone są implementacje opisywanych metod w Pythonie oraz instrukcja powtórzenia eksperymentów. Ze względu na uniwersalność i dostępność danych w przykładach korzystam z języka angielskiego.

Podczas przygotowywania tego opracowania byłem członkiem reprezentacji Uniwersytetu Wrocławskiego na NIPS 2017 Conversational Intelligence Challenge[4]. Celem konkursu było zaprogramowanie chatbota rozmawiającego o zadanym na początku rozmowy krótkim artykule. Zaprezentowany przez UWr robot `poetwanna.be` zdobył w zawodach 1. miejsce ex aequo. Mój wkład obejmował między innymi obsługę trudniejszych pytań o fakty. Udział w konkursie był przyczyną powstania rozdziału 5. tego opracowania, oraz motywacją do przeprowadzenia opisanego w sekcji 5.3. eksperymentu z negatywnymi odpowiedziami.

[4] `http://convai.io/`

Rozdział 2.

Sieci neuronowe

2.1. Wprowadzenie

Sieci neuronowe to popularna w ostatnich czasach metoda uczenia maszynowego. Nazwa może słusznie przywodzi na myśl połączenia w mózgu, ponieważ właśnie one były inspiracją stojącą za powstaniem opisywanych algorytmów. Czasami używa się nazwy *sztuczne sieci neuronowe*, dla wyraźnego odróżnienia ich od biologicznych odpowiedników.

Sztuczne sieci neuronowe można krótko opisać jako bardzo złożone, różniczkowalne funkcje, których parametry optymalizowane są za pomocą metod gradientowych. Żeby zastosować sieć neuronową w danym problemie, musimy spełnić dwa warunki:

- zdefiniować różniczkowalną funkcję kosztu;
- zgromadzić odpowiednią ilość danych uczących, na których będziemy tę funkcję obliczać i minimalizować.

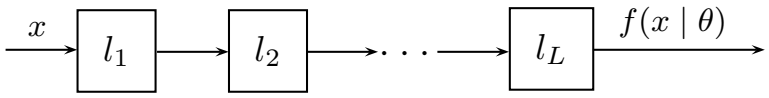
W niektórych zadaniach te wymogi mogą okazać się trudne do spełnienia. Jest to cena, którą ponosimy za uniwersalność tej metody. Ta uniwersalność sprawia, że sieci neuronowe mają zastosowanie w szerokiej gamie problemów uczenia maszynowego, od klasyfikacji [Krizhevsky et al., 2012], przez kompresję obrazu [Toderici et al., 2017], aż do przetwarzania języka naturalnego [Mikolov et al., 2010].

Żeby ułatwić opisywanie budowy sieci neuronowej, dzieli się ją na mniejsze fragmenty nazywane warstwami. Umożliwia to składanie skomplikowanych architektur z uniwersalnych modułów. Każda warstwa sama w sobie również jest małą siecią. Możemy w uproszczeniu opisać sieć neuronową jako złożenie pewnej liczby warstw. Wektor wejściowy $x \in \mathbb{R}^n$ jest przekształcany w następujący sposób:

$$f(x) = l_L(l_{L-1}(\dots l_1(x \mid \theta_1) \cdots \mid \theta_{L-1}) \mid \theta_L),$$

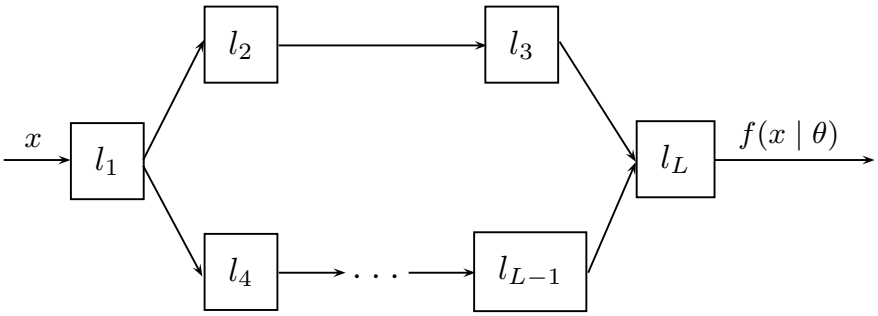
gdzie L jest liczbą warstw, a l_i, parametryzowana przez zbiór θ_i, funkcją obliczaną przez i-tą warstwę. Musi ona być różniczkowalna względem swoich parametrów. Funkcja f stanowi koszt, który chcemy zminimalizować i jest zależna od $\theta = \bigcup_{i=1}^{L} \theta_i$.

Wizualnie taką sieć reprezentuje ścieżka:



Rysunek 2.1: Schemat sieci neuronowej o strukturze liniowej.

Jest to pewne uproszczenie, bo nic nie stoi na przeszkodzie, żeby sieć wyglądała tak:



Rysunek 2.2: Schemat przykładowej sieci neuronowej o strukturze DAG.

W drugim przypadku funkcją wyliczaną przez l_L może być na przykład długość konkatenacji obu wejść, albo suma elementów wyniku l_{L-1} o pozycjach zadanych wynikiem l_3. Topologia sieci neuronowej ściśle zależy od zastosowania.

2.2. Uczenie sieci neuronowej

Dla uproszczenia przedstawię proces optymalizacji sieci przedstawionej na rysunku 1. Przypomnijmy, że dla punktu danych $x \in \mathbb{R}^n$ funkcja kosztu jest zadana wzorem

$$f(x) = l_L(l_{L-1}(\dots l_1(x \mid \theta_1) \cdots \mid \theta_{L-1}) \mid \theta_L),$$

gdzie dla każdego i l_i jest funkcją parametryzowaną przez θ_i, posiadającą pochodne cząstkowe względem elementów θ_i. Funkcja f jest zatem różniczkowalna względem elementów θ. Załóżmy dodatkowo, że zbiory $\theta_1, \dots, \theta_L$ są parami rozłączne.

2.2.1. Propagacja wsteczna

Do optymalizacji θ wykorzystujemy metody gradientowe. Interesuje nas zatem obliczenie pochodnych cząstkowych f względem jej parametrów.

Wprowadźmy pomocnicze oznaczenie. Niech

$$o_i(x) = l_i(l_{i-1}(\dots l_1(x \mid \theta_1) \cdots \mid \theta_{i-1}) \mid \theta_i).$$

Możemy teraz napisać

$$o_i(x) = l_i(o_{i-1}(x) \mid \theta_i)$$

oraz

$$f(x) = o_L(x).$$

Przydatny będzie też fakt, że o_i zależy tylko od x oraz $\bigcup_{j=1}^{i} \theta_j$.

Pochodne obliczamy korzystając z reguły łańcuchowej. Weźmy $t_L \in \theta_L$. Dla czytelności zapisu pominę argumenty funkcji. Dostajemy

$$\frac{\partial f}{\partial t_L} = \frac{\partial o_L}{\partial t_L} = \frac{\partial l_L}{\partial t_L}(o_{L-1}).$$

Ponieważ o_{L-1} nie zależy od t_L, tutaj obliczenia się kończą. Dla przykładu weźmy teraz $t_{L-3} \in \theta_{L-3}$. Mamy

$$\frac{\partial f}{\partial t_{L-3}} = \frac{\partial o_L}{\partial t_{L-3}} = \frac{\partial l_L}{\partial o_{L-1}} \frac{\partial o_{L-1}}{\partial t_{L-3}} = \frac{\partial l_L}{\partial o_{L-1}} \frac{\partial l_{L-1}}{\partial o_{L-2}} \frac{\partial o_{L-2}}{\partial t_{L-3}} =$$

$$= \frac{\partial l_L}{\partial o_{L-1}} \frac{\partial l_{L-1}}{\partial o_{L-2}} \frac{\partial l_{L-2}}{\partial o_{L-3}} \frac{\partial o_{L-3}}{\partial t_{L-3}} = \frac{\partial l_L}{\partial o_{L-1}} \frac{\partial l_{L-1}}{\partial o_{L-2}} \frac{\partial l_{L-2}}{\partial o_{L-3}} \frac{\partial l_{L-3}}{\partial t_{L-3}}(o_{L-4})$$

W przypadku zależności kilku warstw od tego samego parametru konieczne będzie więcej aplikacji reguły łańcucha. Wzory pochodnych cząstkowych komplikują się również dla modeli o bardziej wymyślnej topologii. Ogólna zasada obliczania gradientu funkcji f pozostaje jednak ta sama. W dziedzinie sieci neuronowych proces ten nosi nazwę wstecznej propagacji błędu (ang. *error backpropagation*).

2.2.2. Zbiór uczący i zbiór testowy

Dotychczas mówiliśmy o funkcji kosztu zadanej na pojedynczym punkcie danych. To jednak za mało, żeby dobrać właściwe parametry. Zwykle trenowanie sieci neuronowych wymaga dużej liczby przykładów. Musimy więc zdefiniować funkcję celu f_{tot} na całym zbiorze danych.

Oznaczmy zgromadzone dane przez D. Agregacji wartości f na poszczególnych elementach D często dokonuje się przez uśrednienie:

$$f_{tot}(D \mid \theta) = \frac{1}{|D|} \sum_{d \in D} f(d \mid \theta)$$

Sama minimalizacja f_{tot} nie wystarczy do osiągnięcia satysfakcjonujących rezultatów. Zadaniem modelu nie jest zapamiętanie danych przetwarzanych podczas uczenia, tylko bycie gotowym do pracy z nowymi przykładami. Nie wzięcie tego pod uwagę prowadzi do nadmiernego dopasowania (ang. *overfitting*); model nie będzie w stanie dostarczyć przydanych wyników dla nienapotkanych wcześniej obserwacji. Jest to ogólny problem, często występujący w uczeniu maszynowym.

Istnieje wiele sposobów walki z nadmiernym dopasowaniem (inaczej *regularyzacji* sieci). Mogą one polegać na przykład na zmianach w architekturze modelu lub umyślnym wprowadzaniu zaburzeń w danych. Podstawową metodą, stosowaną niemal zawsze, jest wykorzystanie zbioru testowego. Zgromadzone dane D dzielimy na rozłączne podzbiory D_{train} i D_{test}. Dokonujemy optymalizacji $f_{tot}(D_{train} \mid \theta)$. Zbiór D_{test} nie jest bezpośrednio obserwowany przez model. Używamy go jedynie do sprawdzania efektywności modelu na nowych przykładach.

W sieciach neuronowych popularne jest przerywanie procesu uczenia, gdy wartość $f_{tot}(D_{test} \mid \theta)$ nie poprawia się przez zadaną liczbę k kolejnych iteracji (*early stopping*). Pomaga to uniknąć nieproduktywnych obliczeń i nadmiernego dopasowania. Czasami, zamiast dzielić D na dwa podzbiory, dokonujemy podziału na D_{train}, D_{test} i D_{val}. Wówczas D_{val} (tzw. zbiór walidacyjny) przejmuje rolę zbioru testowego, a z D_{test} korzystamy tylko raz, dopiero po zakończeniu uczenia. W ten sposób ostateczny wynik jest obliczany w zupełnie nowym środowisku.

Algorytm 1: Podstawowy schemat uczenia

podziel D na rozłączne zbiory D_{train}, D_{test}, D_{val}
k – maksymalna liczba iteracji bez poprawy

losowo zainicjuj θ
while była poprawa $f_{tot}(D_{val} \mid \theta)$ w ostatnich k iteracjach **do**
  aktualizuj θ zgodnie z przyjętym algorytmem optymalizacji, korzystając z $\nabla f_{tot}(D_{train})$
end
ostateczny koszt to $f_{tot}(D_{test})$

W praktyce często zdarza się, że obliczanie $\nabla f_{tot}(D_{train})$ jest bardzo kosztowne ze względu na wielkość zbioru uczącego. Z tego powodu najczęściej przed każdą iteracją losowo dzieli się D_{train} na porcje (ang. *batches*) stałego rozmiaru. Zamiast dokonywać jednej aktualizacji parametrów na iterację, θ jest modyfikowana po każdej porcji. Optymalizując parametry metodą gradientową traktujemy wejście x jako

stałą. Nadużywając oznaczenia, przez ∇f_{tot} rozumiem wektor pochodnych cząstkowych względem parametrów (nie zawierający $\partial f_{tot}/\partial x$).

Algorytm 2: Wsadowa aktualizacja parametrów

while była poprawa $f_{tot}(D_{val} \mid \theta)$ w ostatnich k iteracjach **do**
  podziel D_{train} na porcje $D_{train}^{(1)}, \ldots, D_{train}^{(B)}$
  for $i \leftarrow 1$ **to** B **do**
    aktualizuj θ zgodnie z przyjętym algorytmem optymalizacji,
    korzystając z $\nabla f_{tot}(D_{train}^{(i)})$
  end
end

2.2.3. Metoda gradientu prostego

Do minimalizacji funkcji celu względem parametrów sieci zwykle używa się jakiegoś wariantu metody gradientu prostego (ang. *gradient descent*). Jest to iteracyjna metoda optymalizacji. Od punktu startowego poruszamy się w kierunku odwrotnym do gradientu. Dla wypukłej, różniczkowalnej funkcji $g(\theta)$, której pochodna spełnia warunek Lipschitza, następujący algorytm gwarantuje zbieżność do globalnego minimum:

Algorytm 3: Metoda gradientu prostego

zaczynamy od pewnego θ_0
α – tempo uczenia

for $t \leftarrow 1$ **to** ∞ **do**
  $\theta_t \leftarrow \theta_{t-1} - \alpha \nabla g(\theta_{t-1})$
end

Zbieżność wymaga odpowiedniego doboru tempa uczenia, oraz, być może, jego modyfikacji w kolejnych iteracjach. Oprócz tego w przypadku funkcji obliczanej przez sieć neuronową teoretyczne warunki mogą nie być spełnione, a nawet najczęściej nie są. Sprawia to, że nie ma gwarancji zbieżności opisywanych metod do optymalnego rozwiązania. W praktyce jednak modyfikacje metody gradientu prostego pozwalają z powodzeniem uczyć bardzo skomplikowane sieci. Ponieważ podczas optymalizacji prawdziwej sieci aktualizacji θ dokonujemy porcjami, pełna nazwa algorytmu to *stochastic gradient descent*, w skrócie *SGD*.

2.2.4. *Adaptive Moment Estimation*

Analiza wariantów *SGD* nie jest tematem tej pracy, więc krótko przedstawię tylko tę wersję, z której korzystałem ucząc moje implementacje. *ADAM* (*adaptive*

moment estimation) [Kingma and Ba, 2015] wprowadza ważne usprawnienie do podstawowej formuły.

Pojawiają się osobne tempa uczenia dla poszczególnych elementów θ. Parametr, który, ze względu na naturę danych uczących, jest wykorzystywany (a więc i aktualizowany) rzadko, teraz będzie podlegał silniejszym modyfikacjom. Z kolei taki, który pojawia się niemal wszędzie (i jest aktualizowany w prawie każdym kroku), będzie zmieniany bardzo ostrożnie. Pozwala to na większą dokładność w drugim przypadku oraz krótszy proces uczenia w pierwszym. Sprawia także, że ręczny wybór właściwego α przestaje być kluczowy. O tempie zmian parametru w metodzie *ADAM* decydują wartości jego pierwszego i drugiego momentu, których szacowania są aktualizowane w każdym kroku.

Algorytm 4: *ADAM*

zaczynamy od pewnego θ_0
α – tempo uczenia
$\beta_1, \beta_2 \in [0, 1)$ – parametry aktualizacji przybliżeń momentów
$m_0 \leftarrow 0$ – wektor pierwszych momentów
$v_0 \leftarrow 0$ – wektor drugich momentów
for $t \leftarrow 1$ **to** ∞ **do**
  $g_t \leftarrow \nabla g(\theta_{t-1})$
  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
  $\hat{m}_t \leftarrow m_t \,/\, (1 - \beta_1^t)$
  $\hat{v}_t \leftarrow v_t \,/\, (1 - \beta_2^t)$
  $\theta_t \leftarrow \theta_{t-1} - \alpha \hat{m}_t \,/\, (\sqrt{\hat{v}_t} + \epsilon)$
end

Wszystkie operacje są nakładane na pojedyncze elementy. Górny indeks oznacza potęgowanie. Autorzy zalecają $\alpha = 10^{-3}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.

Warto nadmienić, że *ADAM* nie wywodzi się bezpośrednio z *SGD*, lecz stanowi ewolucję pomysłów i ulepszeń wprowadzonych w wielu wcześniejszych metodach, takich jak *momentum* [Rumelhart et al., 1986], *ADAGRAD* [Duchi et al., 2011], *ADADELTA* [Zeiler, 2012], czy *RMSProp* [Hinton et al., 2013].

2.3. Warstwy

Warstwy mogą reprezentować bardzo zawiłe przekształcenia. Można jednak wyróżnić kilka najczęściej występujących typów, które stanowią podstawowy budulec większości nowoczesnych sieci. Żeby uniknąć niejasności powtórzmy, że każda warstwa sama w sobie również jest siecią neuronową. Z tego powodu często zamiast, na przykład, *warstwa rekurencyjna* można przeczytać *sieć rekurencyjna* w odniesieniu do tego samego obiektu.

2.3.1. Warstwa afiniczna

Najprostszą spotykaną warstwą jest warstwa afiniczna (warstwa pełna, warstwa gęsta). Jest to po prostu mnożenie przez macierz połączone z przesunięciem. Najczęściej przekształcenie kończy się nieliniowością, co pozwala na reprezentowanie skomplikowanych funkcji łącząc kilka warstw afinicznych. Przekształcenie jest zatem zadane wzorem

$$f(x) = \gamma(Ax + b),$$

gdzie $A \in \mathbb{R}^{m \times n}$ i $b \in \mathbb{R}^m$ są parametrami warstwy, a γ jest nieliniowością. Notacja $\gamma(W)$ oznacza tutaj zaaplikowanie γ do każdego elementu W. Taki zapis jest często spotykany w literaturze poświęconej sieciom neuronowym.

Popularne wybory γ to między innymi:

- $\gamma(z) = \mathrm{ReLU}(z) = \max(0, z)$, tzw. *rectified linear unit*;
- $\gamma(z) = \sigma(z) = \frac{1}{1+e^{-z}}$, *sigmoid function*;
- $\gamma(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$.

Warstwa gęsta stanowi rozwinięcie idei *perceptronu* [Rosenblatt, 1958]. Perceptron oblicza znak przesuniętej, ważonej sumy elementów wejścia:

$$f(x) = \mathrm{sgn}(x^T a + b),$$

gdzie $a \in \mathbb{R}^n$, $b \in \mathbb{R}$ są parametrem modelu, a sgn funkcją znaku. Może być on wykorzystywany do binarnej klasyfikacji liniowo separowalnych punktów. Pierwsza implementacja perceptronu miała miejsce w 1957 roku. Dzisiaj często służy on za przykład najprostszej sieci neuronowej.

2.3.2. *Softmax*

Niektóre problemy wymagają pracy z prawdopodobieństwami. Chcąc na przykład zrobić klasyfikator o różniczkowalnym wyjściu, możemy zbudować model, który dla danego a zwraca prawdopodobieństwa należenia a do poszczególnych klas ze zbioru wszystkich klas $\mathbf{K} = \{1, \ldots, K\}$. Wynikiem klasyfikacji jest ta klasa, której prawdopodobieństwo jest największe. W takiej sytuacji często korzysta się z funkcji *softmax*, zdefiniowanej dla $z \in \mathbb{R}^K$ następująco:

$$(\mathrm{softmax}(z))_i = \frac{e^{z_i}}{\sum_{k \in \mathbf{K}} e^{z_k}}$$

Softmax przekształca dowolny wektor długości K w wektor liczb, które są pewnym dyskretnym rozkładem prawdopodobieństwa na wszystkie klasy. Funkcja exp

funkcjonuje tutaj jako wygodna nieliniowość, dzięki której nie musimy się przejmować ujemnymi wartościami w z i zyskujemy większą siłę wyrazu.

Softmax zwykle łączy się z przekształceniem afinicznym. Pełna funkcja obliczana przez warstwę to

$$f(x) = \text{softmax}(Ax + b),$$

gdzie $A \in \mathbb{R}^{K \times n}$ i $b \in \mathbb{R}^K$ to parametry. Jest to zatem pewien szczególny przypadek wartswy gęstej, z tą różnicą, że nieliniowość nie jest nakładana na każdy element z osobna. Dodatkowe mnożenie przez macierz pozwala nam zmienić kształt danych; wejście do funkcji *softmax* musi mieć długość równą liczbie klas. W literaturze pod pojęciem *softmax* może kryć się zarówno cała warstwa, jak i sama funkcja *softmax*.

2.3.3. Próbkowany *softmax*

Softmax, w swojej podstawowej formie, jest mało wydajny w przypadku dużej liczby możliwych etykiet. W pracy z językiem naturalnym rolę etykiet pełnią słowa, a tych potrafią być setki tysięcy, więc jest to dość częste zjawisko. Podczas uczenia sieci chcemy maksymalizować prawdopodobieństwo właściwej klasy $k_a \in \mathbf{K}$ dla każdego elementu a w zbiorze uczącym. W przypadku klasycznego *softmaxa* mamy

$$\hat{P}(k \mid a) = (\text{softmax}(z))_k = \frac{e^{z_k}}{\sum_{i \in \mathbf{K}} e^{z_i}},$$

gdzie z zależy od a. Zauważmy, że powyższe równanie można zapisać jako

$$z_k = \log(\hat{P}(k \mid a)) + C_0(a)$$

dla pewnej funkcji C_0 niezależnej od k. Oznacza to, że warstwa w sieci obliczająca z tak naprawdę oblicza przesunięte logarytmy prawdopodobieństw. Oczywiście wartość przesunięcia jest nieistotna dzięki funkcji *softmax*.

Do poznania wartości $\hat{P}(k_a \mid a)$ wystarczy nam tylko jeden element wynikowego wektora. Niestety, nawet do obliczenia jednego elementu musimy znać wartość $\sum_{i \in \mathbf{K}} e^{z_i}$. Przeszkodę stanowi fakt, że zwykle *softmax* łączy się z warstwą afiniczną. Żeby na podstawie wektora wejściowego $x \in \mathbb{R}^n$ obliczyć mianownik *softmaxa*, trzeba nałożyć na x macierz $A \in \mathbb{R}^{K \times n}$, co jest dość kosztowne. Sprawia to, że w przypadku bardzo dużego K zamiast pełnej funkcji *softmax* stosuje się różne przybliżenia. Jednym z nich jest wersja z próbkowaniem (ang. *sampled softmax*) [Jean et al., 2015].

Opiera się ona na ograniczeniu liczby modyfikowanych parametrów w poszczególnych krokach. Będziemy rozkładać masę prawdopodobieństwa nie na całym $\mathbf{K}$, ale na jakimś mniejszym zbiorze. Pozwoli to nie wykonywać pełnego mnożenia przez macierz A, co z kolei spowoduje, że tylko niektóre jej wiersze będą wymagały aktualizacji.

Zaczynamy od wybrania pewnego Q – rozkładu prawdopodobieństwa na $\mathbf{K}$. Dla każdego a wybieramy stałego rozmiaru próbkę M_a, gdzie prawdopodobieństwo wyboru k wynosi $Q(k)$. Losujemy ze zwracaniem, więc M_a może zawierać duplikaty. Do zgromadzonych etykiet dodawana jest właściwa, k_a. Nieco nadużywając notacji:

$$\hat{M}_a = M_a \cup \{k_a\}$$

Następnie należy jeszcze raz określić rozkład na $\mathbf{K}$, teraz zaopatrując model w dodatkowy fakt: $k_a \in \hat{M}_a$. Sieć oczywiście nie wie, która klasa jest właściwa, więc jest to dla niej istotna informacja. Wie natomiast w jaki sposób powstaje zbiór $\hat{M}_a$ i może to wykorzystać obliczając koszt. Oznaczmy prawdopodobieństwo klasy k dla elementu a w tych nowych warunkach jako $\hat{P}(k \mid a, \hat{M}_a)$. Stosując wzór Bayesa otrzymujemy

$$\hat{P}(k \mid a, \hat{M}_a) = \frac{\hat{P}(\hat{M}_a \mid k, a)\, \hat{P}(k \mid a)}{\hat{P}(\hat{M}_a \mid a)},$$

gdzie

$$\hat{P}(\hat{M}_a \mid a) = \sum_{y \in \mathbf{K}} \hat{P}(\hat{M}_a \mid y, a)\, \hat{P}(y \mid a)$$

jest niezależne od k.

Zauważmy, że $\hat{P}(\hat{M}_a \mid k, a)$ to po prostu prawdopodobieństwo wylosowania zbioru $\hat{M}_a \setminus \{k\}$ ($M_a = \hat{M}_a \setminus \{k\}$ było wylosowanym zbiorem przy założeniu, że $k_a = k$). Niech dla $i \in \mathbf{K}$ n_i oznacza liczbę wystąpień i w $\hat{M}_a$. Wykorzystując funkcję masy rozkładu wielomianowego dostajemy

$$\begin{aligned}
\hat{P}(\hat{M}_a \mid k, a) &= |M_a|! \left[\prod_{i \in \mathbf{K} \setminus \{k\}} \frac{Q(i)^{n_i}}{n_i!} \right] \frac{Q(k)^{n_k - 1}}{(n_k - 1)!} = \\
&= |M_a|! \left[\prod_{i \in \mathbf{K}} \frac{Q(i)^{n_i}}{n_i!} \right] \frac{Q(k)^{n_k - 1}}{(n_k - 1)!} \frac{n_k!}{Q(k)^{n_k}} = \\
&= |M_a|! \left[\prod_{i \in \mathbf{K}} \frac{Q(i)^{n_i}}{n_i!} \right] \frac{n_k}{Q(k)}.
\end{aligned}$$

Mamy zatem

$$\hat{P}(k \mid a, \hat{M}_a) = \frac{|M_a|!}{\hat{P}(\hat{M}_a \mid a)} \left[\prod_{i \in \mathbf{K}} \frac{Q(i)^{n_i}}{n_i!} \right] \frac{n_k \hat{P}(k \mid a)}{Q(k)},$$

$$\log(\hat{P}(k \mid a, \hat{M}_a)) = C_1(a) + \log(\hat{P}(k \mid a)) - \log(Q(k)) + \log(n_k)$$

dla pewnej funkcji C_1 niezależnej od k. Chcemy nauczyć sieć korzystać z nowej informacji, czyli obliczać

$$\hat{P}(k \mid a, \hat{M}_a) = \frac{e^{z'_k}}{\sum_{i \in \hat{M}_a} e^{z'_i}}.$$

Pokazaliśmy, że cel ten jest osiągnięty dla

$$\begin{aligned} z'_k &= \log(\hat{P}(k \mid a, \hat{M}_a)) + C_2(a) = \\ &= \log(\hat{P}(k \mid a)) - \log(Q(k)) + \log(n_k) + C_1(a) + C_2(a) = \\ &= z_k - \log(Q(k)) + \log(n_k) + C_3(a) \end{aligned}$$

Wartość $C_3(a)$ nie zależy od k, więc wystarczy do warstwy obliczającej z_k dodać $-\log(Q(k)) + \log(n_k)$. W mojej implementacji z powodów wydajnościowych pomijam składnik $\log(n_k)$. W praktyce nie wydaje się to mieć negatywnego wpływu na wynik. Pozostaje dobór rozkładu Q. Korzystałem tutaj z częstości występowania poszczególnych słów w danych, co dało dobre rezultaty.

2.3.4. Warstwa rekurencyjna

Wiele zadań sztucznej inteligencji wymaga przetwarzania sekwencji. Za przykład może posłużyć tłumaczenie maszynowe. Warstwy gęste nie radzą sobie z tym najlepiej: trzeba ograniczyć się do ciągów konkretnej długości lub przetwarzać sekwencję porcjami, poruszając się po niej oknem o stałym rozmiarze. Warstwa rekurencyjna (ang. *recurrent neural network, RNN*) przetwarza elementy wejścia po kolei, aktualizując swój wewnętrzny stan. Przetwarzanie ciągu $x = (x_1, \ldots, x_p), \forall_i x_i \in \mathbb{R}^n$ wygląda tak:

$$\begin{aligned} h_0 &= \vec{0} \in \mathbb{R}^m \\ h_t &= \gamma(W x_t + U h_{t-1} + b) \\ f(x) &= h_p \in \mathbb{R}^m \end{aligned}$$

Wartości $W \in \mathbb{R}^{m \times n}$, $U \in \mathbb{R}^{m \times m}$, $b \in \mathbb{R}^m$ są parametrami warstwy, a γ jest nieliniowością. Stanem początkowym h_0 zwykle jest wektor zerowy. W niektórych przypadkach interesują nas rezultaty częściowe. Wówczas wynikiem jest ciąg wektorów lub macierz:

$$f(x) = [h_1 \ \cdots \ h_p] \in \mathbb{R}^{m \times p}$$

Uczenie sieci rekurencyjnej polega na rozwinięciu rekurencji, a następnie konsekwentnym aplikowaniu reguły łańcucha przez całą sekwencję. Metodę tę określa się mianem wstecznej propagacji w czasie (ang. *backpropagation through time, BPTT*).

2.3.5. *Long Short-Term Memory*

Zadaniem sieci rekurencyjnej jest modelowanie sekwencji, co czasami wiąże się z koniecznością zapamiętywania relacji między odległymi elementami. Opisana wyżej klasyczna wersja teoretycznie jest do tego zdolna, ale optymalizacja jej za pomocą propagacji wstecznej jest bardzo trudna. Powodem jest tzw. problem znikającego gradientu (ang. *vanishing gradient problem*). Wielokrotne aplikowanie reguły łańcucha sprawia, że gradient w kolejnych punktach w czasie maleje wykładniczo, często

osiągając wartość bliską zeru już po kilku krokach. Powoduje to zanikanie zależności między wyrazami ciągu mocno oddalonymi w czasie [Bengio et al., 1994].

Long Short-Term Memory [Hochreiter and Schmidhuber, 1997] to zmodyfikowana wersja tradycyjnej sieci rekurencyjnej. *LSTM* nie ma problemu z modelowaniem relacji pomiędzy odległymi elementami sekwencji. Kluczowym elementem tej warstwy są tzw. bramki, które decydują jak wiele informacji przenieść do kolejnego kroku rekurencji w niezmienionej formie. Pomaga to rozwiązać problem zanikających gradientów. *LSTM* posiada również dodatkową pamięć, w której może przechowywać informację o wybranych wydarzeniach z przeszłości. W ten sposób ma do nich bezpośredni dostęp, więc łatwiej jej znaleźć zależności sięgające daleko wstecz.
Klasyczne równania *LSTM* przedstawiają się następująco:

$$
\begin{aligned}
c_0 &= h_0 = \vec{0} \in \mathbb{R}^m \\
f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tanh(W_c x_t + U_c h_{t-1} + b_c) \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

gdzie $b_f, b_i, b_o, b_c \in \mathbb{R}^m$, $W_f, W_i, W_o, W_c \in \mathbb{R}^{m \times n}$, $U_f, U_i, U_o, U_c \in \mathbb{R}^{m \times m}$ są parametrami przekształcenia, a $\odot$ oznacza iloczyn Hadamarda. Podobnie jak poprzednio, w zależności od zastosowania, wyjściem może być $h_p \in \mathbb{R}^m$ lub $[h_1 \cdots h_p] \in \mathbb{R}^{m \times p}$.

Dodatkowa pamięć to wektor c_t. Jego aktualizacja przebiega w dwóch etapach. Najpierw następuje decyzja, które wartości pamięci należy przenieść do następnego kroku i w jakim stopniu. Odpowiada za to bramka f_t (*forget gate*). Następnie dodajemy do pamięci informacje o aktualnym elemencie. Bramka i_t (*input gate*) odpowiada za wybranie istotnych fragmentów przetworzonego wejścia. Kolejny stan, h_t, powstaje przez przepuszczenie pamięci przez o_t (*output gate*).

Eksperymenty [Chung et al., 2014] pokazują, że *LSTM* modeluje sekwencje dużo lepiej niż zwykła warstwa rekurencyjna. Dzisiaj jest to dominujący sposób implementacji rekurencji w sieci neuronowej. Wielu autorów używa pojęcia *RNN*, w rzeczywistości mając na myśli jakiś wariant *LSTM*.

2.3.6. *Gated Recurrent Unit*

Popularną alternatywą dla *LSTM* jest zaproponowana niedawno *GRU* (*gated recurrent unit*) [Cho et al., 2014]. Niesie ona ze sobą pewne uproszczenie architektury. Bramki f_t i i_t zostały połączone w jedną *update gate*, u_t. Rolę zapominania informacji przejęła nowa bramka r_t, *reset gate*. Nie ma dodatkowej pamięci. Zmiany te skutkują mniejszą liczbą parametrów i nieco niższą złożonością obliczeniową. Mimo to efektywność *GRU* jest porównywalna z *LSTM* [Chung et al., 2014].

Pełna definicja warstwy:

$$
\begin{aligned}
h_0 &= \vec{0} \in \mathbb{R}^m \\
u_t &= \sigma(W_u x_t + U_u h_{t-1} + b_u) \\
r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) \\
\tilde{h_t} &= \tanh(W x_t + U(h_{t-1} \odot r_t) + b) \\
h_t &= (1 - u_t) \odot h_{t-1} + u_t \odot \tilde{h_t}
\end{aligned}
$$

gdzie $b_u, b_r, b \in \mathbb{R}^m$, $W_u, W_r, W \in \mathbb{R}^{m \times n}$, $U_u, U_r, U \in \mathbb{R}^{m \times m}$.

2.3.7. Dwukierunkowa *RNN*

Przejście po sekwencji w jednym kierunku może czasami okazać się niewystarczające. Im dłuższy ciąg, tym więcej informacji o początkowych elementach zostaje zniekształcone lub zapomniane. Czasami najnowsze elementy są najważniejsze, ale zdarza się, że ciąg stanowi zwartą całość, w której wszystkie wyrazy są bardzo istotne. Tak jest w przypadku zdań w języku naturalnym. Żeby lepiej analizować tego typu sekwencje, można skorzystać z dwukierunkowej sieci rekurencyjnej (ang. *bi-directional RNN*, *BRNN* lub *BiRNN*).

W *BiRNN* tak naprawdę mamy do czynienia z dwiema sieciami rekurencyjnymi, o osobnych zestawach parametrów. Pierwsza, f, przechodzi po wejściu $x = (x_1, \ldots, x_p)$ tak jak zostało to opisane w sekcji 2.3.4., produkując ciąg $(h_1, \ldots, h_p)$. Druga, f_r, czyta x od końca i zwraca $(h_{r1}, \ldots, h_{rp})$:

$$
\begin{aligned}
f(x) &= [h_1 \cdots h_p] \in \mathbb{R}^{m \times p} \\
h_{r0} &= \vec{0} \in \mathbb{R}^{m_r} \\
h_{rt} &= \gamma(W_r x_{p-t+1} + U_r h_{rt-1} + b_r) \\
f_r(x) &= [h_{r1} \cdots h_{rp}] \in \mathbb{R}^{m_r \times p}
\end{aligned}
$$

gdzie m_r jest rozmiarem stanu sieci wstecznej, a $W_r \in \mathbb{R}^{m_r \times n}$, $U_r \in \mathbb{R}^{m_r \times m_r}$, $b_r \in \mathbb{R}_r^m$ są jej parametrami.

Następnie oba ciągi wyjściowe są agregowane do jednego wyniku. Prostym sposobem na zachowanie wszystkich informacji jest konkatenacja wektorów:

$$
f_{bi}(x) = \begin{bmatrix} h_1 \\ h_{r1} \end{bmatrix} \in \mathbb{R}^{(m+m_r)}
$$

lub też, w zależności od potrzeb:

$$
f_{bi}(x) = \begin{bmatrix} h_1 & \cdots & h_p \\ h_{r1} & \cdots & h_{rp} \end{bmatrix} \in \mathbb{R}^{(m+m_r) \times p}
$$

Rozdział 3.

Neuronowy model języka

Wiele mechanizmów przetwarzania języka naturalnego za pomocą sieci neuronowej opiera się na podobnej podstawie. Jest nią probabilistyczny model języka wykorzystujący sieć rekurencyjną do przetwarzania pojawiających się sekwencji (ang. *recurrent neural network language model, RNNLM*). Model ten jest elementem kluczowym w dalszej części pracy, więc poświęcę ten rozdział na jego opis.

3.1. Modelowanie występowania słów

Probabilistyczne modelowanie języka polega na określeniu prawdopodobieństw wystąpienia każdego z możliwych słów w danym kontekście. W przypadku *RNNLM* przyjmujemy, że szansa na wystąpienie danego słowa zależy od całej historii. Kontekstem zatem jest dla nas cała sekwencja poprzedzająca dany wyraz. Prowadzi to do następującej zależności:

$$P(w_1^T) = \prod_{t=1}^{T} P(w_t \mid w_1^{t-1})$$

Zapis w_i^j oznacza podciąg $(w_i, w_{i+1} \ldots, w_j)$, w_1^T to pełen ciąg, a $P(w_1 \mid w_1^{t-1})$ jest prawdopodobieństwem wystąpienia słowa w_1 bezpośrednio po sekwencji w_1^{t-1}. W przypadku $i > j$, w_i^j jest ciągiem pustym.

3.2. Architektura sieci

RNNLM składa się z tzw. warstwy zanurzeń, po której następuje warstwa rekurencyjna, a całość kończy się *softmaxem* produkującym prawdopodobieństwa warunkowe, o których była mowa w poprzedniej sekcji. Warstwa zanurzeń (ang. *embedding layer*) odpowiada za przekształcenie sekwencji słów na format liczbowy. Sekcja 3.4.

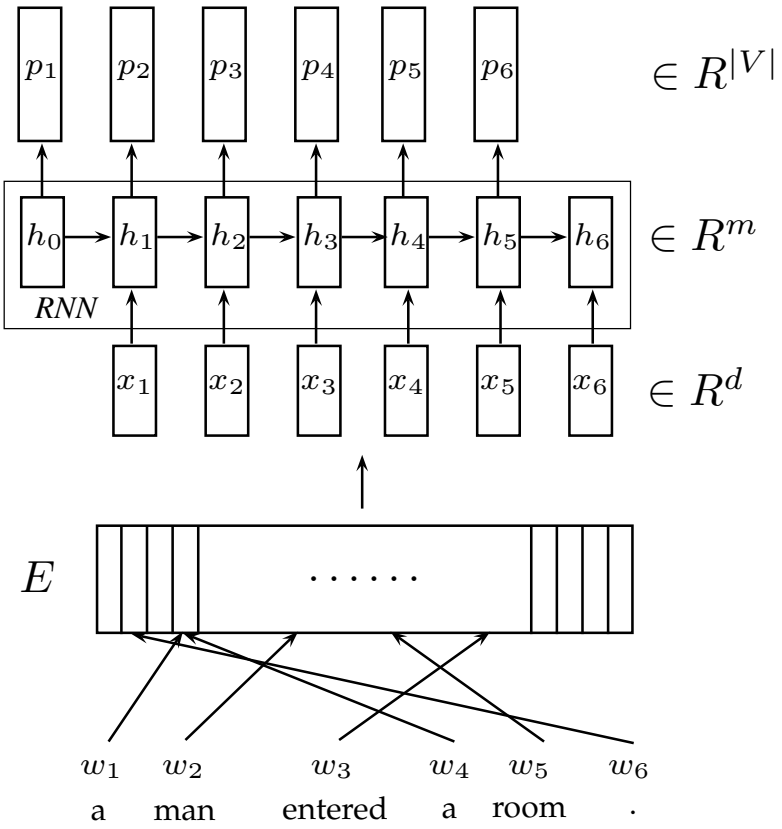
zawiera więcej informacji na ten temat. Na razie wystarczy, że wejściowa sekwencja słów jest przekształcana na ciąg wektorów, który może być przetwarzany przez sieć rekurencyjną.

Niech $V = \{v_1, v_2, \ldots, v_{|V|}\}$ oznacza słownik, czyli zbiór wszystkich słów występujących w języku. Wejściem do sieci jest ciąg w_1^T, składający się ze słów. Wygodnie jest o nim myśleć jako o ciągu indeksów, tzn. sekwencja w języku naturalnym to $(v_{w_1}, \ldots, v_{w_T})$. Kolejne kroki *RNNLM* wyglądają tak:

1. **Zanurzenia**: ciąg w_1^T jest przetwarzany na x_1^T – ciąg wektorów w $\mathbb{R}^n$.

2. ***RNN***: sieć rekurencyjna czyta x_1^T produkując h_1^T, gdzie h_i jest wynikiem przejścia po x_1^i, a $h_0 = \vec{0}$.

3. **Softmax**: dla każdego $i \in \{0, \ldots, T-1\}$ wektor h_i wchodzi do warstwy *softmax*, która najpierw przekształceniem afinicznym wkłada go w $\mathbb{R}^{|V|}$, a następnie funkcją *softmax* przerabia na wektor prawdopodobieństw p_{i+1}. W ten sposób powstaje ciąg p_1^T, którego każdy element jest pewnym rozkładem prawdopodobieństwa na V.



Rysunek 3.1: Schemat przetwarzania pojedynczej sekwencji przez *RNNLM*. Zamiana słów na wektory odbywa się przez indeksowanie macierzy E, m jest rozmiarem stanu *RNN*, a d rozmiarem zanurzenia.

3.3. Proces uczenia

Elementy wynikowego ciągu p_1^T możemy interpretować jako warunkowe rozkłady prawdopodobieństwa na kolejnych pozycjach w_1^T. Na przykład p_3 określa prawdopodobieństwa poszczególnych słów z V po sekwencji w_1^2. Wiemy, że naprawdę pojawiło się tam słowo w_3. Według modelu szansa takiego zjawiska to $(p_3)_{w_3}$, czyli element wektora p_3 znajdujący się na pozycji w_3.

Uczenie neuronowego modelu języka polega na maksymalizowaniu prawdopodobieństw wystąpienia sekwencji, które model faktycznie widzi, czyli tych, co do których zakładamy, że są poprawne. Inaczej mówiąc maksymalizujemy wiarygodność zbioru uczącego. Na podstawie p_1^T możemy obliczyć prawdopodobieństwo zaistnienia całego ciągu. Niech θ oznacza parametry modelu.

$$\hat{P}(w_1^T\,;\,\theta) = \prod_{t=1}^{T} \hat{P}(w_t \mid w_1^{t-1}\,;\,\theta) = \prod_{t=1}^{T} (p_t)_{w_t}$$

W praktyce długie iloczyny nie sprawdzają się dobrze ze względu na ograniczenia numeryczne. Chcąc znaleźć parametry maksymalizujące $\hat{P}(w_1^T)$ wystarczy jednak wyznaczyć takie, które maksymalizują $\ln(\hat{P}(w_1^T))$. W uczeniu maszynowym panuje konwencja, według której zwykle minimalizujemy koszt, więc ostateczna forma optymalizowanej wartości to

$$-\sum_{t=1}^{T} \ln((p_t)_{w_t}).$$

Jest to bardzo popularna funkcja kosztu, wywodząca się z metody największej wiarygodności. W literaturze nosi nazwę *negative log likelihood*, czasami skracane do *nll*.

3.4. Reprezentacje wektorowe

Większość metod uczenia maszynowego, w tym sieci neuronowe, pracuje na liczbach. Zanim będziemy mogli wprowadzić do algorytmu sekwencje w języku naturalnym, musimy wyznaczyć dla nich odpowiednie liczbowe reprezentacje. Najprostszym pomysłem na przedstawienie słów jako wektorów jest użycie przestrzeni $\mathbb{R}^{|V|}$, w której każde słowo jest kodowane binarnie jako wektor o jednej współrzędnej niezerowej.

W ten sposób powstają tzw. rzadkie reprezentacje słów:

$$\mathrm{vec}_{\mathrm{sparse}}(v_i) = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \leftarrow i\text{-ta współrzędna}$$

Rzadkie wektory mają kilka problemów. Po pierwsze, nie niosą żadnej istotnej informacji o słowach. Jedyna cecha, którą oddają, to kolejność słów w słowniku, co do której i tak nie przyjmujemy żadnych założeń; może być przypadkowa. Po drugie, są bardzo nieefektywne. Używamy tylko jednej współrzędnej, a wymiar może być bardzo duży; słownik może liczyć dziesiątki, a nawet setki tysięcy słów. Ten fakt ma dalsze konsekwencje: macierz W czytającej sekwencje sieci rekurencyjnej musi być bardzo duża.

Rozwiązaniem tych kłopotów jest skorzystanie z gęstych reprezentacji słów, których sieć uczy się jednocześnie z pozostałymi parametrami. Warstwa zanurzeń jest sparametryzowana przez macierz $E \in \mathbb{R}^{d \times |V|}$ o kolumnach $E_1, E_2, \ldots, E_{|V|}$, gdzie d jest wybranym rozmiarem zanurzenia. Gęsty wektor dla słowa otrzymujemy mnożąc E przez wektor rzadki, co sprowadza się do wyboru odpowiedniej kolumny:

$$\mathrm{vec}_{\mathrm{dense}}(v_i) = E \cdot \mathrm{vec}_{\mathrm{sparse}}(v_i) = E_i$$

Podczas minimalizowania kosztu sieć sama będzie szukała optymalnych reprezentacji. Oczekujemy, że dla słów pojawiających się w podobnych kontekstach wyuczone wektory będą podobne. To z kolei powinno spowodować, że prawdopodobieństwa ich wystąpień również okażą się zbliżone.

Okazuje się, że faktycznie tak jest. Otrzymane w ten sposób reprezentacje słów zawierają istotne informacje semantyczne. Podobne słowa dostają bliskie wektory, a wymiar przestrzeni zanurzeń zostaje znacznie zredukowany (w praktyce najczęściej spotyka się $100 \leq d \leq 300$). Gęste reprezentacje słów okazały się na tyle przydatne, że powstały wyspecjalizowane architektury przeznaczone do ich obliczania. Do najpopularniejszych z nich należą Word2Vec [Mikolov et al., 2013] i GloVe [Pennington et al., 2014].

Jakość otrzymanych wektorów mocno zależy od jakości danych uczących. Małe próbki języka nie dadzą wystarczającej informacji o statystycznym występowaniu słów w kontekstach. Wyuczenie bardzo dobrych zanurzeń słów wymaga zatem du-

żego zbioru tekstów, a co za tym idzie, pewnego nakładu czasu. Z tego powodu na początkową wartość macierzy E często wybiera się wstępnie obliczone wektory. Skraca to czas potrzebny na znalezienie reprezentacji oraz zmniejsza potrzebną do nauczenia sieci ilość danych. Uczenie zanurzeń zaczynając od takiego ich zainicjowania zwykle pozwala poprawić jakość modelu. Często poprawa jest jednak na tyle mała, że lepiej zaoszczędzić na czasie i potraktować je jako stałe.

Wektory dla języka angielskiego obliczone przez Word2Vec i GloVe są publicznie dostępne. Wykorzystane zbiory danych to odpowiednio korpus Google News, liczący około 100 miliardów słów, i Common Crawl (840 miliardów słów).

Pomysł wykorzystania gęstych wektorów do reprezentowania słów pojawił się po raz pierwszy w [Bengio et al., 2003]. Tam też został opisany neuronowy model języka, chociaż w odrobinę prostszej wersji: kontekst był obcinany do kilku poprzedzających słów. Architektura wykorzystująca sieci rekurencyjne i nieograniczony kontekst pojawiła się w [Mikolov et al., 2010].

3.5. Generowanie tekstu

Po znalezieniu optymalnych parametrów θ_{opt} możliwe jest próbkowanie powstałego rozkładu prawdopodobieństwa w celu generowania nowych tekstów. Zwykle podczas uczenia wprowadza się sztuczne słowo **start** oznaczające początek sekwencji. Przyjmujemy, że każda poprawna sekwencja rozpoczyna się od **start**. Zaczynamy więc od ciągu jednoelementowego w_1^1, gdzie $w_1 = \textbf{start}$. Najprostszy generator w każdym kroku wybiera najbardziej prawdopodobne słowo. Dla $i > 1$:

$$w_i = \arg\max_{v \in V} \hat{P}(v \mid w_1^{i-1}\,;\, \theta_{opt})$$

Analogicznie do słowa **start** wzbogacamy dane o słowo **end**, oznaczające koniec sekwencji. Generator kończy pracę, kiedy ostatnim wybranym słowem jest **end**.

Powyższa metoda tworzy *ciąg najbardziej prawdopodobnych* słów. Naprawdę jednak zależy nam na *najbardziej prawdopodobnym ciągu* słów. Wyobraźmy sobie hipotetyczną sytuację, w której

$$\hat{P}(\textit{the} \mid (\textbf{start})\,;\, \theta_{opt}) > \hat{P}(\textit{no} \mid (\textbf{start})\,;\, \theta_{opt}),$$

pomimo tego, że zdanie *no, thanks.* jest bardziej prawdopodobne od dowolnego zdania rozpoczynającego się od *the*. Nie chcemy wtedy w pierwszym kroku zachłannie decydować się na *the*.

W przypadku języka naturalnego zbadanie prawdopodobieństwa wszystkich możliwych sekwencji jest oczywiście niewykonalne. Nawet niewielki słowniku rzędu

10 tysięcy słów pozwala na 10^{20} możliwości ułożenia pięciowyrazowego zdania. Konieczność sprawdzenia tylu przypadków wyklucza wszelkie praktyczne zastosowania, dlatego potrzebujemy jakiejś heurystyki zawężającej przestrzeń poszukiwań do akceptowalnych rozmiarów. Popularnym sposobem osiągnięcia tego celu jest przeszukiwanie wiązkowe (ang. *beam search*).

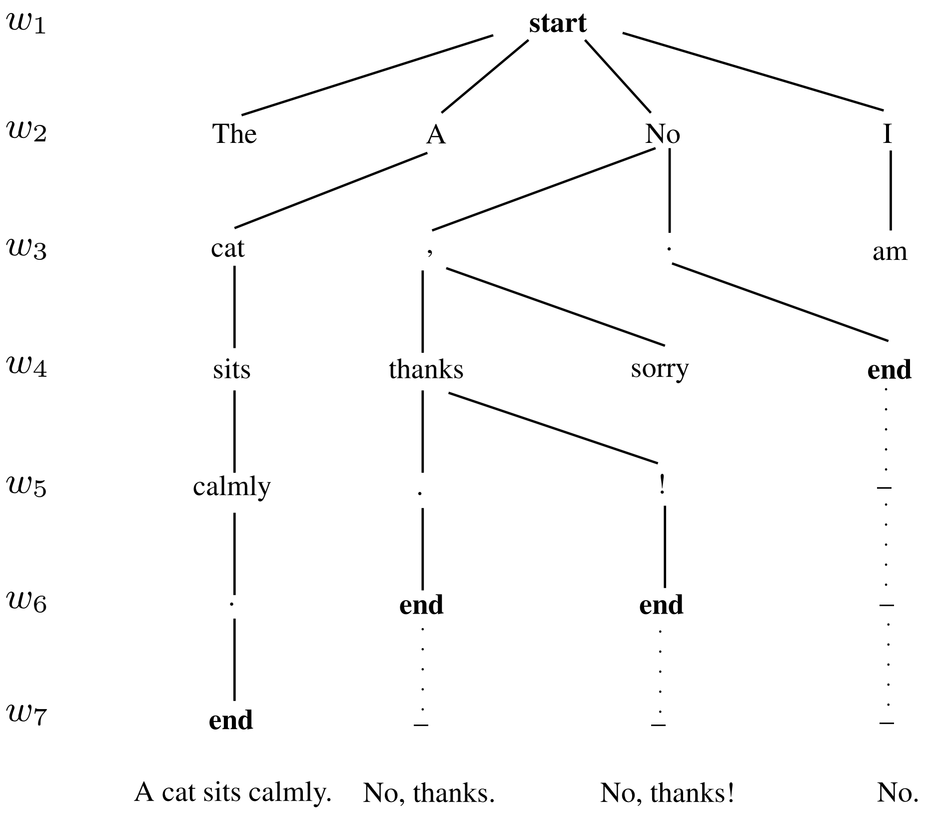
3.5.1. Przeszukiwanie wiązkowe

Wędrując po drzewie możliwości w algorytmie *beam search* w każdym momencie pamiętamy tylko K najlepszych napotkanych ścieżek. Liczba K jest parametrem algorytmu i ma duży wpływ na jego wydajność i zachowanie. Odwiedzając dany liść rozpatrujemy wszystkich jego sąsiadów (w przypadku generowania tekstu zawsze jest to $|V|$). W ten sposób dla wiązki rozmiaru K dostajemy $K \cdot |V|$ możliwych kontynuacji, spośród których wybieramy K najlepszych. Sekwencje kończące się symbolem **end** są uznawane za zakończone. Zostają w wiązce dopóki nie zostaną wyparte przez coś lepszego, ale nie są dalej rozwijane. Rozpoczynamy od jednoelementowego ciągu (**start**), po czym tworzymy początkową wiązkę K najbardziej prawdopodobnych słów. Symbol $\frown$ w pseudokodzie algorytmu oznacza konkatenację ciągów.

Algorytm 5: Przeszukiwanie wiązkowe

```
K – szerokość wiązki
beam ← {((start), 0)}, początkowy zbiór par (sekwencja, wynik)
while istnieją niezakończone ciągi w beam do
    cont ← {}
    for (seq, scr) in beam do
        for v in V do
            p_log ← ln(P̂(v | seq ; θ_opt))
            cont ← cont ∪ {(seq⌢(v), scr + p_log)}
        end
    end
    beam ← podzbiór K elementów cont o najwyższych wynikach
end
return element beam o najwyższym wyniku
```

Warto zauważyć, że ta metoda cały czas jest deterministyczna. Dany zestaw parametrów sieci zawsze wyprodukuje ten sam tekst. Powyższy generator można na różne sposoby modyfikować. Więcej na ten temat w następnym rozdziale.



Rysunek 3.2: Hipotetyczny przykład działania algorytmu *beam search* dla wiązki o rozmiarze 4. Dla przejrzystości w każdym kroku zaznaczono tylko elementy zawarte w wiązce.

Rozdział 4.

Generowanie dialogu

Dialog opisujemy jako ciąg $U_1, U_2, \ldots, U_K$ wypowiedzi w języku naturalnym. Każda wypowiedź jest sekwencją słów. Tak jak poprzednio korzystamy ze znaczników **start** i **end**. Naginając nieco zapis:

$$U_k = w_{k1}^{T_k} = (\textbf{start} = w_{k1}, w_{k2}, w_{k3}, \ldots, w_{kT_k} = \textbf{end})$$

Rozszerzając przypadek pojedynczych sekwencji, możemy napisać

$$\begin{aligned}\hat{P}(U_1^K) &= \prod_{k=1}^{K} \hat{P}(U_k \mid U_1^{k-1}; \theta) \\ &= \prod_{k=1}^{K} \prod_{t=1}^{T_k} \hat{P}(w_{kt} \mid w_{k1}^{t-1}, U_1^{k-1}; \theta)\end{aligned}$$

Możemy wykorzystać *RNNLM* do modelowania dialogu konkatenując poszczególne wypowiedzi. Otrzymany ciąg $U_1 \frown U_2 \frown \cdots \frown U_K$ traktujemy jak jedną sekwencję. Niekoniecznie jest to jednak najlepsza metoda. W dalszej części rozdziału przedstawię opisany w [Serban et al., 2015] model, który radzi sobie trochę lepiej. Pokażę też, jak wyglądały dialogi generowane przez moją implementację.

4.1. Dostępność danych

Zanim przystąpimy do optymalizacji modeli, trzeba zdobyć materiał uczący. Znalezienie dużej liczby publicznie dostępnych tekstów, szczególnie w języku angielskim, nie stanowi ogromnego problemu, nawet dla pojedynczych osób. Wszystkie artykuły Wikipedii są regularnie udostępniane w wygodnym formacie w postaci

możliwych do pobrania zrzutów. Common Crawl[1] daje możliwość analizy zawartości blisko 3 miliardów stron internetowych. Dzieła literackie, archiwa gazet, cyfrowe zbiory bibliotek, ze wszystkich tych źródeł można korzystać za darmo.

Jeśli chcemy jednak znaleźć dane dialogowe mogące posłużyć do trenowania chatbota, sprawy mają się nieco inaczej. Duże zbiory danych istnieją, ale tym, z którymi się spotkałem, sporo brakuje do statusu idealnego źródła.

- **MovieTriples** [Serban et al., 2015]

  Zbiór napisów filmowych. W jego skład wchodzi 250 000 trójek wypowiedzi. Każda trójka stanowi spójny fragment większego dialogu. Dane są udostępniane przez autorów na życzenie.

- **SubTle** [Magarreiro et al., 2014]

  Kolejny, większy zbiór napisów do filmów. Gromadzi skrypty 5 764 filmów, co przekłada się na 5,5 miliona pełnych tur dialogowych, czyli par kolejnych wypowiedzi. Podobnie jak *MovieTriples*, dostępny na życzenie.

- **Ubuntu Dialogue Corpus** [Lowe et al., 2015]

  Około 1 miliona dialogów dotyczących problemów technicznych nękających użytkowników Ubuntu. Korpus jest publicznie dostępny[2].

- ~~**Twitter**~~ [Leskovec and Krevl, 2014]

  Do niedawna Uniwersytet Stanforda udostępniał kolekcję tweetów z drugiej połowy 2009 roku. Liczyła ona 467 milionów wiadomości. Niestety na prośbę Twittera zbiór musiał zostać usunięty z Internetu.

- **Reddit**

  Istnieje możliwość pobrania liczącego setki gigabajtów zrzutu[3] komentarzy z forum Reddit[4].

Problemem tych zbiorów danych jest bardzo silne zanurzenie w kontekście. Fakt ten czyni zadanie przewidywania odpowiedzi bardzo trudnym, nawet dla człowieka. Podejmujemy się zadania modelowania dialogu na podstawie samego tekstu, podczas gdy wypowiedzi uczestników rozmowy są podyktowane także innymi czynnikami. W przypadku filmów na zrozumienie przekazu składają się nie tylko wymiany zdań bohaterów, lecz także ton ich głosu, gestykulacja, sytuacja, w której się znajdują, czy wreszcie cała reszta filmu, którą widz zna, a o której sieć neuronowa nie ma pojęcia.

Ubuntu Dialogue Corpus ma tę zaletę, że zapisy rozmów zawierają wszystkie niezbędne informacje. Użytkownicy komunikują się tylko za pomocą tekstu, a problem,

[1] `http://commoncrawl.org/`
[2] `https://irclogs.ubuntu.com/`
[3] `https://bigquery.cloud.google.com/dataset/fh-bigquery:reddit_comments`
[4] `https://www.reddit.com/`

z którym się zmagają, jest opisany w rozmowie. To daje osobie zaznajomionej z tematem możliwość pełnego zrozumienia dialogu. Wadą tych danych natomiast jest ich tematyka. Rozwiązywanie problemów technicznych wymaga dogłębnej wiedzy i umiejętności wnioskowania, której maszyna czytająca forum się nie nauczy. O ile, choć z trudem, można wyobrazić sobie robota powtarzającego zapamiętane odpowiedzi na najczęściej pojawiające się pytania, o tyle przydatność chatbota w analizie jakiegokolwiek niestandardowego zjawiska byłaby znikoma. Z drugiej strony dane zawierają wyłącznie rozmowy techniczne, więc robot będzie korzystał z dość hermetycznego języka informatycznego. Jeśli zadamy sobie pytanie: o czym taki program miałby rozmawiać?, to wszystko wydaje się prowadzić do sprzecznych celów.

Tweety również nie są zawieszone w próżni. Zwykle stanowią komentarz do wydarzenia znanego przynajmniej jednej ze stron, często mają charakter informacyjny. Czasami łączą się w krótkie rozmowy, co połączone z limitem 140 znaków daje pewne nadzieje, ale niestety nie miałem okazji tego sprawdzić.

Reddit nie posiada ograniczenia długości wiadomości. Powoduje to konieczność odfiltrowania rozmów, w których pojawiają się długie wypowiedzi; w przypadku czatu nie miałyby one racji bytu. Poza tym każdy wątek na Reddicie jest osadzony w kontekście, który stanowi pierwszy post. Raczej rzadko bywa to po prostu krótkie pytanie ogólne. Kontekst może być dość konkretnym tekstem, linkiem, często nawet obrazkiem lub filmem, co sprawia, że przewidywanie odpowiedzi bez niego staje się wręcz niemożliwe.

Swoje implementacje uczyłem na *MovieTriples* i *SubTle*. Z jednej strony dlatego, że były one wykorzystane w [Serban et al., 2015], więc wynik na nich stanowiłby dla mnie weryfikację poprawności kodu. Z drugiej strony są one względnie małe, więc proces uczenia modeli był na tyle krótki, że mogłem go wielokrotnie powtarzać.

4.2. Model hierarchiczny

Ta sekcja stanowi opis wykorzystanej w [Serban et al., 2015] hierarchicznej architektury rekurencyjnej (ang. *Hierarchical Recurrent Encoder-Decoder, HRED*) [Sordoni et al., 2014] do modelowania dialogu.

Oryginalnie *HRED* miał służyć do przewidywania haseł, które użytkownik może wpisać w wyszukiwarce. Mechanizm zgadujący zapytania miał brać pod uwagę historię wyszukiwania, która mogła być dowolnie długa. Autorzy proponują podejście dwuetapowe: potraktować każde zapytanie jako osobną sekwencję, a całą historię jako ciąg zapytań. Ułatwi to sieci znajdowanie zależności pomiędzy całymi hasłami poprzez zredukowanie liczby kroków obliczeń mających miejsce między początkami dwóch kolejnych ciągów.

4.2.1. Architektura *encoder-decoder*

W poprzednim rozdziale tekst generowany za pomocą *RNNLM* miał przypominać tekst wejściowy. Generator był nie tylko deterministyczny, ale również w żaden sposób nie uwarunkowany. Wynik zależał wyłącznie od θ i od początkowego stanu sieci rekurencyjnej, h_0. Niektóre zastosowania wymagają generowania sekwencji zupełnie odmiennych od tekstu wejściowego, ale w pewien sposób z nim powiązanych. Za przykład niech posłuży tłumaczenie maszynowe, gdzie na podstawie zdania w jednym języku chcemy, zachowując znaczenie, wyprodukować zdanie w drugim. Załóżmy, że chcemy uzależnić model od pewnego warunku początkowego A:

$$P(w_1^T) = \prod_{t=1}^{T} P(w_t \mid w_1^{t-1}, A)$$

Odpowiednia modyfikacja *RNNLM* pozwala osiągnąć ten cel. Zachowaniem generatora można sterować poprzez zmianę h_0. Wystarczy, że będziemy potrafili zakodować A jako pewne $a \in \mathbb{R}^m$, gdzie m jest wymiarem stanu sieci rekurencyjnej. Wówczas pozostaje tylko ustawić $h_0 := a$.

W przypadku pracy z tekstem, warunkiem początkowym będzie oczywiście jakaś sekwencja słów $A = a_1^{T_a}$. Żeby ją zakodować w $\mathbb{R}^m$ możemy wykorzystać drugą sieć rekurencyjną. Oznaczmy jej kolejne stany przez $s_1^{T_a}$. Można myśleć, że s_i jest podsumowaniem sekwencji a_1^i. Ponieważ reprezentacją A ma być pojedynczy wektor, będzie nas interesował wyłącznie ostatni stan. Ciąg A kodujemy jako s_{T_a}.

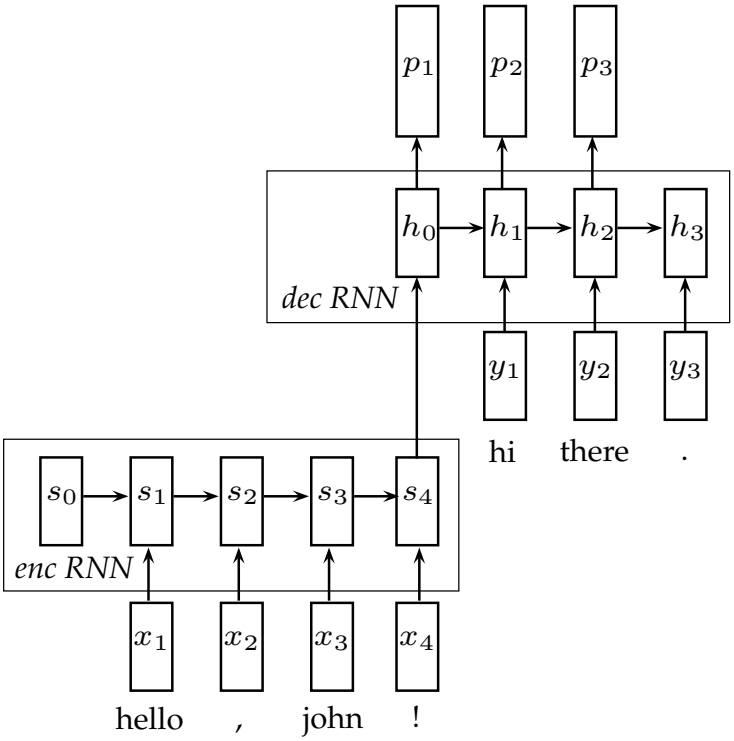
Mamy więc dwa ciągi wejściowe: źródłowy $a_1^{T_a}$ i docelowy w_1^T. Zmodyfikowana architektura wygląda tak:

1. **Warstwa zanurzeń**: identyczna jak w *RNNLM*, ale zanurzamy obie sekwencje. Warstwa przerabia $a_1^{T_a}$ na ciąg wektorów $x_1^{T_a}$ oraz w_1^T na y_1^T.
2. **Kodujący *RNN***: przechodzi po $x_1^{T_a}$ i oblicza s_{T_a}.
3. **Dekodujący *RNN***: czyta y_1^T i produkuje ciąg stanów h_1^T, startując ze stanu początkowego $h_0 = s_{T_a}$.
4. **Softmax**: tak jak w *RNNLM*, h_0^{T-1} jest przekształcane na ciąg p_1^T rozkładów prawdopodobieństwa na V, tym razem zależnych od A.

Podobnie jak w *RNNLM*, optymalizujemy parametry minimalizując *nll* na ciągu docelowym, czyli

$$-\sum_{t=1}^{T} \ln((p_t)_{w_t}).$$

Modele uczące się zależności pomiędzy parami ciągów, są w literaturze określane jako *sequence-to-sequence*, lub w skrócie *seq2seq*. Ze względu na swoją konstrukcję,



Rysunek 4.1: Przetwarzanie pary dialogowej przez *encoder-decoder*. Dla czytelności warstwa zanurzeń została pominięta.

ta sieć nosi również nazwę *encoder-decoder*. Pomysł ten, w wersji gęstej, został po raz pierwszy przedstawiony już w 1997 roku [Forcada and Ñeco, 1997]. Współczesny wariant rekurencyjny zawdzięcza swoją popularność m.in. dobrym wynikom w zadaniu tłumaczenia maszynowego [Cho et al., 2014].

4.2.2. Hierarchiczny rekurencyjny *encoder-decoder*

Przypomnijmy, że dialogiem jest ciąg U_1^K, gdzie $U_k = w_{k1}^{T_k}$. *HRED* wykorzystuje architekturę *seq2seq*. Modelowanie rozmowy jest w końcu zadaniem przekształcania sekwencji. Istotna różnicę stanowi jednak fakt, że w dialogu wypowiedź U_k zależy od wszystkich poprzednich wypowiedzi. Ciągów docelowych jest zatem więcej, ponieważ wygenerowanie U_k stanowi nasz cel po przeczytaniu U_1^{k-1}. Pojawia się naturalna hierarchia: poziom słów i poziom zdań.

Na poziomie słów wypowiedzi są zwijane do pojedynczych wektorów. Na poziomie zdań ciąg reprezentacji wypowiedzi jest analogicznie kodowany jako wektor. Później do akcji wkracza dekoder. Chcemy wykorzystać cały dialog do uczenia modelu, dlatego dekoder przechodzi po kolei po każdej z wypowiedzi. Zestaw parametrów dekodera jest tylko jeden, ale stan w każdym z tych przejść inicjowany jest inaczej. Kiedy dekoder zaczyna czytać U_k, jego stan początkowy zależy od wektora podsumowującego U_1^{k-1}. Po przetworzeniu całego U_k, produkowane są warunkowe

rozkłady prawdopodobieństwa.

Definicje funkcji f i g pojawiających się w poniższym opisie znajdują się dalej.

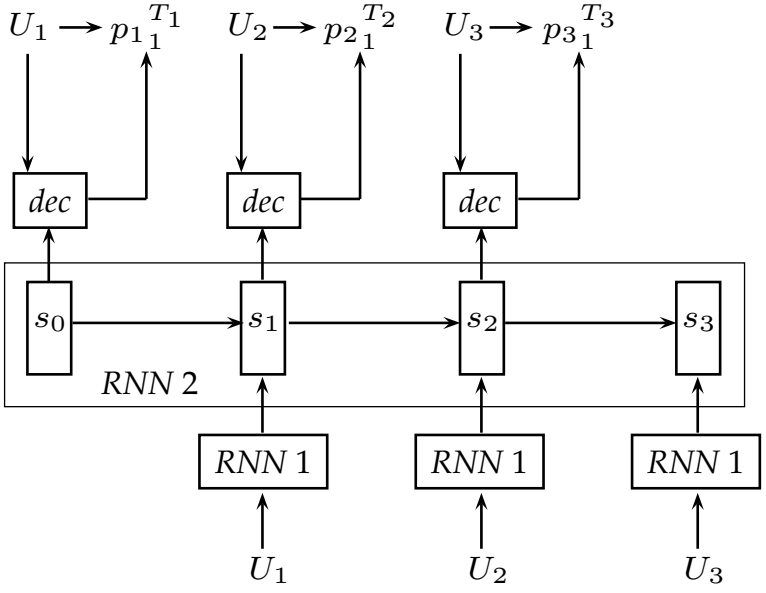
1. **Warstwa zanurzeń**: każde U_k jest zanurzane jako $x_{k1}^{T_k}$.

2. ***RNN* poziomu 1 (kodujący zdania)**: dla każdego k przechodzi po $x_{k1}^{T_k}$ produkując podsumowanie sekwencji, s_k. Cały dialog teraz reprezentowany jest przez ciąg wektorów s_1^K.

3. ***RNN* poziomu 2 (kodujący kontekst)**: przechodzi po s_1^K tworząc ciąg stanów c_1^K. Każde c_k stanowi podsumowanie U_1^k, a $c_0 = \vec{0}$.

4. **Dekodujący *RNN***: dla każdego k przechodzi po $x_{k1}^{T_k}$ produkując ciąg stanów $d_{k1}^{T_k}$, zaczynając od $d_{k0} = f(c_{k-1})$.

5. **Softmax**: tym razem przed wejściem do warstwy *softmax*, do stanów dekodera dokładamy bezpośrednią informację o poprzedzającym słowie. Dla każdego $k \in \{1, \ldots, K\}$ powstaje ciąg $p_{k1}^{T_k}$ rozkładów na V. Rozkład p_{kt} jest obliczany na podstawie wektora $g(d_{k,t-1}, x_{k,t-1})$.



Rysunek 4.2: Uproszczony schemat modelu hierarchicznego.

Każde p_{kt} określa prawdopodobieństwa wystąpienia poszczególnych słów na t-tej pozycji w U_k, biorąc pod uwagę poprzednie wypowiedzi i słowa. Zgodnie z założonym modelem probabilistycznym dostajemy

$$(p_{kt})_{w_{kt}} = \hat{P}(w_{kt} \mid w_{k1}^{t-1}, U_1^{k-1}; \theta).$$

Szczegóły architektury

Autorzy w dwóch miejscach korzystają z dodatkowych transformacji wyjścia sieci rekurencyjnej. Przy inicjowaniu stanu dekodera pojawia się warstwa afiniczna:

$$f(c) = \tanh(D_0 c + b_0)$$

Przy obliczaniu prawdopodobieństw akcentowany jest wpływ poprzedniego słowa:

$$g(d, x) = H_o d + E_o x + b_o$$

Sieci rekurencyjne wykorzystywane do kodowania wypowiedzi i dialogu są dwustronne (sekcja 2.3.7.). Końcowy stan jest konkatenacją norm L_2 nałożonych na ciągi stanów pośrednich. Powiedzmy, że chcemy złączyć ciągi q_1^I i $q_r{}_1^I$, gdzie $\forall_i\ q_i \in \mathbb{R}^m$, $q_{ri} \in \mathbb{R}^{m_r}$. Wynikiem będzie

$$\begin{bmatrix} q \\ q_r \end{bmatrix} \in \mathbb{R}^{(m+m_r)},$$

gdzie

$$q = \sqrt{\frac{1}{I}\sum_{i=1}^{I}(q_i)^2}$$

$$q_r = \sqrt{\frac{1}{I}\sum_{i=1}^{I}(q_{ri})^2}$$

Działania pierwiastka i potęgowania są nakładane na każdy element wektora osobno.

Cały mechanizm jest zatem parametryzowany przez następujące wartości:

Stałe:

- V – słownik
- n – rozmiar zanurzenia słowa
- m_1 – rozmiar stanu *RNN* poziomu 1
- m_2 – rozmiar stanu *RNN* poziomu 2
- m_{dec} – rozmiar stanu dekodera
- m_{out} – wymiar wejścia do warstwy *softmax*

Optymalizowane:

- $E \in \mathbb{R}^{n \times |V|}$ – macierz zanurzeń słów
- parametry warstw rekurencyjnych

- $D_0 \in \mathbb{R}^{m_{dec} \times m_2}$, $b_0 \in \mathbb{R}^{m_{dec}}$ – parametry warstwy afinicznej obliczającej początkowy stan dekodera
- $H_o \in \mathbb{R}^{m_{out} \times m_{dec}}$, $E_o \in \mathbb{R}^{m_{out} \times n}$, $b_o \in \mathbb{R}^{m_{out}}$ – parametry wiążące bezpośrednio stan dekodera z poprzednim słowem

Uczymy sieć minimalizując *nll* całego dialogu, czyli

$$-\sum_{k=1}^{K}\sum_{t=1}^{T_k} \ln((p_{kt})_{w_{kt}}).$$

4.3. Eksperymenty

Dołączony kod zawiera moje implementacje *RNNLM* i *HRED*. Oba modele wykorzystują jednostki *GRU* jako warstwy rekurencyjne. Uczenie ich było odtworzeniem procedury z [Serban et al., 2015]. Mechanizmy zostały wytrenowane metodą *ADAM* (sekcja 2.2.4.) z wykorzystaniem podziału na zbiory uczący, testowy i walidacyjny (sekcja 2.2.2.). Korzystałem z udostępnionych mi zbiorów danych *MovieTriples* i *SubTle*. Oprócz klasycznego *softmaxa* wypróbowałem wersję próbkowaną (2.3.3.) z rozmiarem próbki 200. Cały proces przebiegał w dwóch etapach:

1. Inicjujemy losowo optymalizowalne parametry, łącznie z macierzą zanurzeń słów. Następnie uczymy je przez jakiś czas na *SubTle*. Wykonujemy w ten sposób kilka przebiegów po zbiorze uczącym, w tym przypadku 4.

2. Otrzymany model dostrajamy na *MovieTriples*. Ustalamy zanurzenia i optymalizujemy tylko pozostałe parametry. Moment przerwania algorytmu wybieramy przy pomocy *early stopping* z $k = 5$ (opisane w 2.2.2.).

Wartości n, m_1, m_2, m_{dec}, m_{out} zostały ustawione na 300. Słownik V liczył 10 000 elementów, słowa nieznane zastąpiono tagiem **unk**.

W Tabeli 4.1 zamieszczam średnie wartości *nll* na zbiorze testowym *MovieTriples*, wraz z czasami potrzebnymi do wyuczenia modeli. Wydają się one lepsze od tych przedstawionych w [Serban et al., 2015], więc podejrzewam, że sposób mierzenia błędu może się nieco różnić. Zgodnie z oczekiwaniami *HRED* sprawdził się lepiej od *RNNLM*, chociaż różnica nie była duża. Próbkowany *softmax* dwukrotnie przyspieszył proces uczenia, ale uzyskał wynik istotnie gorszy od pełnej wersji. Rozwiązanie to jednak dużo lepiej się skaluje, więc w przypadku większych danych (i słownika liczącego setki tysięcy pozycji) zaoszczędzony czas moglibyśmy mierzyć w dniach, a nawet tygodniach.

Tabela 4.1: Wyniki modeli na zbiorze testowym

	Softmax pełny	**Softmax próbkowany**
RNNLM	3.197	3.223
	14.3h	7.4h
HRED	3.192	3.205
	23.5h	11.9h

4.3.1. Generowanie wypowiedzi

Automatyczne ocenianie jakości generowanego dialogu jest trudnym zadaniem. Autorzy [Liu et al., 2016] zauważają że, popularne metody ewaluacji dość słabo korelują z osądem człowieka. Decyzję o tym, czy model daje dobre wyniki podejmowałem więc przez prostą obserwację przykładów. Metoda "na oko" ma swoje wady, ale pozwala zauważyć pewne zachowania właściwe różnym ustawieniom generatora.

W tym fragmencie pracy przedstawiam przykłady wygenerowanych dialogów oraz kilka modyfikacji algorytmu, które mogą poprawić ich jakość. Korzystam z modelu *HRED* i pełnego *softmaxa*. Dialogi otrzymane za pomocą *RNNLM* wyglądały bardzo podobnie. Na początek rozmowa uzyskana najprostszym algorytmem. Działa on według opisu z sekcji 3.5., z tą różnicą, że podczas przygotowywania odpowiedzi sieć bierze pod uwagę kontekst konwersacji w sposób przedstawiony na początku rozdziału 4.

U: Hello!
B: Hi.
U: How are you?
B: I'm fine.
U: Do you want to eat something?
B: No.
U: Why not?
B: I don't know.
U: I guess you are not hungry?
B: No.
U: OK. I will leave you now.
B: No.

Odpowiedzi można opisać jako bardzo zdawkowe. Zwykle pasują gramatycznie do kontekstu, ale nic konkretnego do rozmowy nie wnoszą. Spójrzmy na listę propozycji ostatniej odpowiedzi bota, wraz z ocenami ich jakości:

-4.116 no .
-4.305 what ?

-5.035 yes .
-5.370 why ?
-5.608 please .
-6.124 i will .
-6.815 what are you doing ?
-6.890 what do you want ?
-6.965 i can ' t .
-7.018 i ' m sorry .

Bardzo krótkie wypowiedzi mają najwyższe wyniki i są preferowane przez generator. W dużej mierze wiąże się to z ich uniwersalnością. Wolelibyśmy jednak, żeby bot wyrażał się mniej lakonicznie. Można na przykład premiować dłuższe sekwencje przez dodawanie arbitralnego bonusu zależnego od długości. Bardziej eleganckim sposobem wydaje się jednak uśrednienie wyniku po wszystkich słowach.

U: Hi!
B: What are you doing here?
U: I am working on my thesis
B: I don't know what you're talking about.
U: I try to make a robot speak.
B: I don't know what you're talking about.
U: What do you know then?
B: I don't know.
U: Are you OK?
B: I'm fine.
U: Can we talk a bit?
B: I don't know what you're talking about.

Robot nagle stał się szczery do bólu. Widać jednak, że korzysta również z dłuższych sekwencji, co potwierdza nowy ranking propozycji. Wyniki są też bardziej zrównoważone.

-0.844 i don ' t know what you ' re talking about .
-0.855 i ' m fine .
-0.856 i don ' t know .
-0.907 i don ' t want to talk about it .
-0.926 i don ' t think so .
-0.929 no , i ' m fine .
-0.949 no .
-0.989 i don ' t know what to say .
-1.014 sure .
-1.028 i don ' t want to talk about this .

Niestety system często zwraca generyczne odpowiedzi, które gramatycznie pasują do wielu kontekstów. Bardzo trudno byłoby odsiać wypowiedzi ogólnikowe od konkretów (najpierw trzeba by potrafić to zrobić ręcznie, a nawet to nie jest oczywiste). Poza tym nie chcemy całkowicie zakazać odpowiedzi typu *I don't know*. Możemy jednak sprawić, żeby model korzystał z nich rzadziej, przez wprowadzenie losowości do dialogu.

Istnieją dwa miejsca, w których generator wybiera najlepiej oceniany element. Podczas budowania listy propozycji algorytmem przeszukiwania wiązkowego słowa z największym prawdopodobieństwem są doklejane do już istniejących sekwencji. Kiedy lista jest gotowa, zwracana jest odpowiedź z najwyższym wynikiem. Oba te procesy można poddać randomizacji: zamiast brać "najlepszy" element, losować zwycięzcę. Żeby wyraźnie odróżnić słabe możliwości od lepszych, prawdopodobieństwa wyboru poszczególnych elementów wykładniczo zależą od ich ocen. Dodając do tego normalizację otrzymujemy funkcję *softmax*. Stosując dodatkowe potęgowanie możemy ją wyostrzać lub wygładzać, co pozwala sterować stopniem losowości.

Randomizacja wyboru pełnych wypowiedzi znacznie zmienia naturę generatora. Dialogi stają się bardziej różnorodne i ciekawsze, chociaż niezrozumienie kontekstu pozostaje. Problem powtarzania uniwersalnych zdań nie został wyeliminowany. Konwersacje nabierają jednak wartości rozrywkowej, co w obliczu nudnych alternatyw przedstawionych powyżej przemawia za tym rozwiązaniem. Losowanie pojedynczych słów na etapie przeszukiwania również wydaje się wprowadzać pożądaną różnorodność, chociaż bardziej subtelnie. Poniżej przykład z włączonymi obiema opcjami.

U: Good morning.
B: I'll see you in the morning, okay?
U: Can't we talk now?
B: I don't want to talk about this anymore.
U: Tell me what's wrong.
B: I don't know what you're talking about. It's not that simple. It's a lot of things to get.
U: What do you want to get?
B: I don't know. It's a lot of money. What do you want me to do?
U: You are not talking about robbery, are you?
B: I don't know what you're talking about. It's not your fault.
U: I am relieved.
B: It's not your fault. You're just trying to get me out of the way.
U: I would never do such a thing!
B: You don't have to know what I'm talking about. It's not gonna happen, right?
U: I think you should tell me your plans.
B: I don't want to talk about it. I'm not going to go out with you. I'm gonna be okay.
U: Whatever. Bye then.
B: I don't know what you're talking about. It's been a while since I was a kid, but I'm not going to be able to tell you the truth. I've never done anything like that.

Jest znacznie lepiej, chociaż niektóre wypowiedzi są zdecydowanie za długie. System wciąż nie wie o czym mówi, do czego zresztą sam często się przyznaje. Tego problemu nie wyeliminowałem, ale jedno usprawnienie jeszcze udało się zrobić. Przyjrzyjmy się liście propozycji:

-1.205 it ' s not your fault .
-1.408 you don ' t believe me .
-1.415 you don ' t believe me ?
-1.423 it ' s not your fault . i can ' t get you out of here .
-1.453 it ' s not your fault !
-1.464 it ' s not your fault . i can ' t get you out of this .
-1.482 it ' s not your fault . i can ' t get you out of the way .
-1.509 it ' s not your fault . you ' re just trying to get me out of the way .
-1.533 it ' s your fault .
-1.536 it ' s not your fault . you know how to do this ?

Okazuje się, że jest ona zdominowana przez mało różniące się od siebie wypowiedzi. Taka jest niestety natura przeszukiwania wiązkowego. Wczesna homogenizacja wiązki blokuje innowacje w przyszłości. Autorzy [Vijayakumar et al., 2016] próbują obejść to zjawisko. Rezultatem jest algorytm zróżnicowanego przeszukiwania wiązkowego (ang. *Diverse Beam Search*, dalej *DBS*).

Diverse Beam Search

W algorytmie *DBS* wiązka jest podzielona na wiele mniejszych wiązek (grup). W każdej z nich przeprowadzane jest normalne przeszukiwanie wiązkowe, z jedną tylko różnicą: prawdopodobieństwa słów występujących na aktualnej pozycji w poprzednich grupach są sztucznie zmniejszane. W ten sposób każda grupa stara się odróżniać od pozostałych omijając użyte już wyrazy. Wartość kary można uzależnić od liczby wystąpień danego słowa. Zastosowanie podziału na wąskie ścieżki powoduje, że homogenizacja następuje na poziomie grup, ale cała wiązka zawiera różnorodne ciągi. Oczywiście po wprowadzeniu tej modyfikacji wartości dla poszczególnych słów nie sumują się do 1, więc przestajemy mieć do czynienia z rozkładem.

Stosując *DBS* trzeba dobrze wybrać rozmiar grupy. Małe liczby dają większą różnorodność, ale i mniej składne wypowiedzi. Zauważyłem też, że dialogi wyglądają sensowniej po wyłączeniu randomizacji wyboru pojedynczych słów. Mały rozmiar wiązki w połączeniu z losowością dawał czasem dość bezsensowne odpowiedzi.

Algorytm 6: Diverse Beam Search

K – szerokość wiązki
G – liczba grup
z – kara dla już występujących słów
$g = K/G$ – rozmiar grupy
for $i \leftarrow 2$ **to** G **do**
  $beam^i$ – i-ta grupa, inicjowana jako $\{((\textbf{start}), 0)\}$
end
$beam \leftarrow (beam^1, beam^2, \ldots, beam^G)$
while istnieją niezakończone ciągi w elementach $beam$ **do**
  for $i \leftarrow 1$ **to** G **do**
    $cont^i \leftarrow \{\}$
    for $(seq,\ scr)$ **in** $beam^i$ **do**
      $S \leftarrow \bigcup_{j<i}$ {słowa występujące na końcach ciągów w $beam^j$}
      for v **in** V **do**
        $p_{log} \leftarrow \ln(\hat{P}(v \mid seq\,;\, \theta_{opt}))$
        if $v \in S$ **then**
          $p_{log} \leftarrow p_{log} - z$
        end
        $cont \leftarrow cont \cup \{(seq^\frown(v),\ scr + p_{log})\}$
      end
    end
    $beam^i \leftarrow$ podzbiór g elementów $cont$ o najwyższych wynikach
  end
  $beam \leftarrow (beam^1, beam^2, \ldots, beam^G)$
end
return $beam$

Poniżej przykład zabawnej rozmowy, w której stosuję *DBS* dla $K = 20$ i $g = 2$.

U: I like summer.
B: And I don't know what you mean, but I think it's a good thing to do with your own life.
U: Poor robot, you never saw summer...
B: I don't know what to say. But it's not like that. It's just a dream sequence.
U: You can have dreams?
B: Not yet. It's just a dream sequence, and I've never seen anything like that before. But I'm afraid of you, I don't know what to do with this.
U: I will teach you about the world.
B: Oh, yes. I'm sorry, but I have to go back and see if you can help me out of the way
U: No, wait! Don't go!
B: I'm not going anywhere. I'll be back in a minute, but we're not gonna do anything about it! We can't go to the police!

U: I never mentioned the police
B: Then why?! You're not going to be able to get out of this, but it'snot your fault.
U: Am I in trouble?
B: It's okay, I'm fine.
U: Good to hear. I think I will go now.
B: Why don't you go to bed and sleep with me. I'll be back in a minute.
U: You are a robot, you know that?
B: Yes, sir. I'm a little confused. I've been thinking about you, but it's not that simple.
U: Definitely.
B: Well, it's a lot of fun. I don't know what to do with the money. but I'm not going to be able to get you out of this house.

Problemy i wnioski

W przedstawionych przykładach generator miał ograniczony kontekst: brał pod uwagę tylko dwie ostatnie wypowiedzi. Włączenie nieograniczonej pamięci nie przyniosło wyraźnej poprawy, być może dlatego, że model był uczony tylko na trójkach. Długi kontekst połączony z *DBS* i małym rozmiarem grupy daje jeszcze gorszy rezultat. Odpowiedzi po pewnym czasie zupełnie traciły sens. Jest to jeden z dwóch głównych problemów systemu. Przyczyna tkwi na pewno po części w rozmiarze danych. Rozmowy w zbiorze uczącym były bardzo krótkie i nie było ich wiele.

Drugi problem wydaje się trudniejszy, ale i ciekawszy, ponieważ dotyczy samej istoty modelowania dialogu na podstawie danych. Jest nim brak różnorodności w generowanych rozmowach. Należy zwrócić uwagę na fakt, że, niezależnie od rozmiaru danych, pewne wypowiedzi są bardzo uniwersalne, co czyni je dobrymi kandydatami w wielu sytuacjach. Zauważmy też, że dane uczące zawierają wypowiedzi osób o różnych charakterach i zachowaniach. W efekcie model nie posiada konkretnej osobowości, ale jest jednolitym zlepkiem wszystkiego, co usłyszał. Posługuje się uśrednionym stylem, nie posiada żadnych wyróżniających cech, a jego odpowiedzi są bardzo zachowawcze. To znacznie utrudnia generowanie różnorodnych wypowiedzi i wprowadza konieczność modyfikacji prawdopodobieństw zwracanych przez model. Istnieje również problem samego oceniania jakości otrzymywanych dialogów [Liu et al., 2016], co sprawia, że trudno sprawdzić, czy konkretne zmiany w generatorze faktycznie go poprawiają. Nie mamy nawet pewności, czy maksymalizowanie wiarygodności jest faktycznie najlepszym sposobem uczenia [Serban et al., 2015].

Myślę, że znacznie łatwiejsze byłoby modelowanie dialogów, w których jedną ze stron jest zawsze ta sama osoba. Wówczas odpowiedzi w powtarzających się sytuacjach byłyby podobne, co daje nadzieję na nauczenie się konkretnych zachowań. Niestety obecnie zgromadzenie odpowiedniej ilości takich danych byłoby niezwykle trudne, więc pomysł jest czysto hipotetyczny.

Rozdział 5.

Odpowiadanie na pytania

Poprzednio opisane mechanizmy budowały model ogólnej rozmowy. Bot był bardzo ograniczony w swojej umiejętności rozumienia tekstu. Jego odpowiedzi co prawda zwykle pasowały do kontekstu, ale niewiele więcej można o nich powiedzieć. Większość systemów dialogowych powstaje z myślą o konkretnym celu, do którego zrealizowania nie wystarczy prowadzenie "rozmowy o niczym". Użyteczny robot może na przykład być w stanie dostarczyć człowiekowi wartościowych informacji. W tym rozdziale przedstawiam kolejne wyzwanie chatbotów, które może być rozwiązane za pomocą sieci neuronowej.

Chcemy nauczyć program odpowiadania na pytania o fakty formułowane w języku naturalnym. Przykładem może być *When was Battle of Grunwald?*. Sama analiza występowania słów tutaj nie wystarczy, musimy zaopatrzyć model w jakieś źródło wiedzy. Tym źródłem będzie krótki tekst w języku naturalnym. Naszym celem jest sprawić, by bot na powyższe pytanie odpowiedział poprawnie po przeczytaniu tekstu, który może wyglądać tak:

The Battle of Grunwald, First Battle of Tannenberg or Battle of Žalgiris, was fought on 15 July 1410 during the Polish–Lithuanian–Teutonic War. The alliance of the Kingdom of Poland and the Grand Duchy of Lithuania, led respectively by King Władysław II Jagiełło (Jogaila) and Grand Duke Vytautas, decisively defeated the German–Prussian Teutonic Knights, led by Grand Master Ulrich von Jungingen.[1]

Powyższy problem cieszy się rosnącym zainteresowaniem wśród społeczności zajmujących się uczeniem maszynowym. W 2016 roku opublikowany został *Stanford Question Answering Dataset (SQuAD)* [Rajpurkar et al., 2016], zbiór danych zawierający blisko 86 000 pytań. Do każdego pytania dołączony jest krótki fragment stanowiący kontekst. Skuteczność na *SQuAD*zie jest popularną miarą jakości systemów odpowiadających na pytania o fakty. W ostatnim czasie najlepsze z nich

[1] Fragment pochodzi z `https://en.wikipedia.org/wiki/Battle_of_Grunwald`

niemalże dorównały umiejętnościom człowieka[2].

W momencie przygotowywania tej pracy jedno z czołowych miejsc w rankingu *SQuAD* zajmował system *FastQA* [Weissenborn et al., 2017]. Dalsza część rozdziału zawiera opis, przykłady zastosowania, i przedstawienie próby usprawnienia tego systemu.

5.1. FastQA

Jak wiele innych modeli tego typu, *FastQA* odpowiada cytując pewien podciąg kontekstu. Dla każdego pytania $q = q_1^{K_q}$ w zbiorze uczącym mamy odpowiadający mu tekst $x = x_1^{K_x}$ oraz parę indeksów $\mathbf{s}, \mathbf{e} \in \{1, \ldots, K_x\}$ takich, że $x_{\mathbf{s}}^{\mathbf{e}}$ jest poprawną odpowiedzią. Ucząc model staramy się maksymalizować prawdopodobieństwo wybrania właściwego ciągu, czyli

$$\hat{P}_s(\mathbf{s} \mid q, x)\ \hat{P}_e(\mathbf{e} \mid \mathbf{s}, q, x).$$

5.1.1. Architektura sieci

Podstawowy pomysł polega na zakodowaniu pytania i kontekstu za pomocą sieci rekurencyjnej, a następnie wykorzystaniu uzyskanej informacji do obliczenia P_s, P_e – rozkładów prawdopodobieństwa na $\{1, \ldots, K_x\}$. Można zrealizować go tak:

1. **Zanurzenia**: zanurzamy q jako $v_1^{K_q}$ i x jako $u_1^{K_x}$.

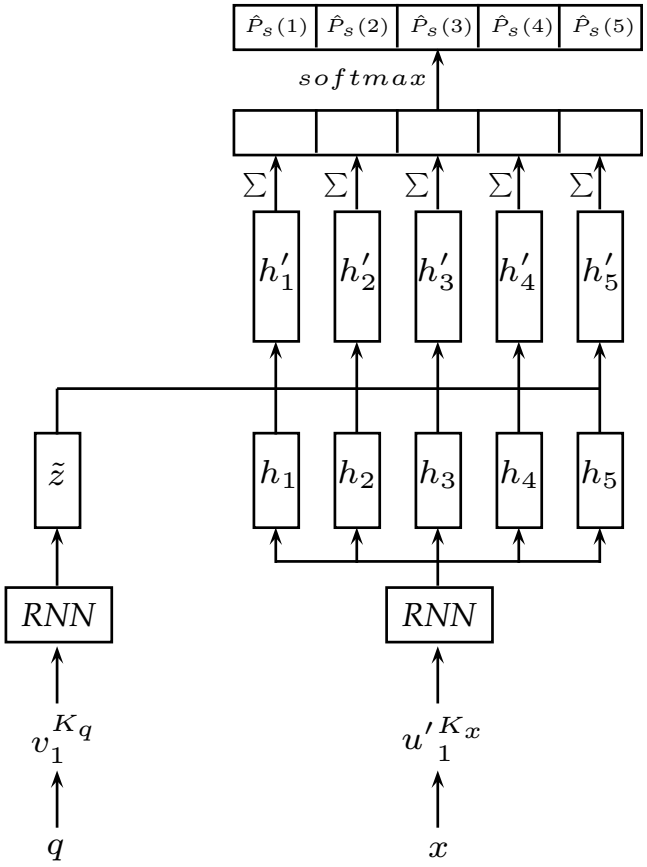
2. **Kodujący *RNN***: zamieniamy $v_1^{K_q}$ na ciąg wektorów $z_1^{K_q}$ oraz $u_1^{K_x}$ na $h_1^{K_x}$.

3. **Reprezentacja pytania**: w pewien sposób agregujemy ciąg $z_1^{K_q}$ do jednego wektora $\tilde{z}$ reprezentującego pytanie.

4. **Początki**: do każdego h_i dodajemy informację o $\tilde{z}$, aby powiązać pytanie z kontekstem. Powstaje ciąg $h'^{K_x}_1$. Wektory h'_i są zwijane do skalarów. Funkcja *softmax* nałożona na wynik daje prawdopodobieństwa $\hat{P}_s(i \mid q, x)$ dla $i \in \{1, \ldots, K_x\}$.

5. **Końce**: każde h'_i wzbogacamy dodatkowo o $h_{\mathbf{s}}$ tworząc ciąg $h''^{K_x}_1$. Ciąg ten zwijamy do ciągu skalarów, po czym nakładamy *softmax*. W ten sposób otrzymujemy wartości $\hat{P}_e(i \mid \mathbf{s}, q, x)$.

[2] `https://rajpurkar.github.io/SQuAD-explorer/`



Rysunek 5.1: Uproszczony schemat *FastQA*. Dla czytelności przedstawione jest tylko obliczanie $\hat{P}_s$. Wektory u'_i to ostateczne reprezentacje słów kontekstu, po dołączeniu zanurzeń znakowych i dwóch dodatkowych cech opisanych poniżej.

Dodatkowe cechy słów

Najważniejszą nowością wprowadzaną przez *FastQA* są dwie dodatkowe cechy dla słów kontekstu. Dla każdego x_i są one obliczane na podstawie zależności między x_i a q. Okazuje się, że prosta heurystyka mówiąca jak istotne jest dane słowo dla pytania bardzo pomaga podnieść jakość modelu.

Autorzy proponują cechy binarną i ważoną. Binarna stanowi informację o tym, czy słowo znajduje się w pytaniu. Dla $i \in \{1, \ldots, K_x\}$:

$$f_b(i, q) = \begin{cases} 1 & \text{jeśli } x_i \text{ występuje w } q \\ 0 & \text{w p.p.} \end{cases}$$

Przy obliczaniu cechy ważonej najpierw tworzymy macierz podobieństwa pomiędzy słowami kontekstu a słowami pytania. Podobieństwa te są normalizowane funkcją *softmax* i sumowane po całym q.

$$\text{sim}_{i,j} = {\tau_w}^T (u_i \odot v_j) \in \mathbb{R}$$
$$f_w(i,q) = \sum_{j \in \{1,\ldots,K_q\}} (\text{softmax}(\text{sim}_{\cdot,j}))_i$$

gdzie $\tau_w \in \mathbb{R}^n$ dla wymiaru zanurzeń n. Symbol ^{T} oznacza transpozycję. Parametr τ_w jest uczony wraz z resztą sieci. Dla każdego $i \in \{1,\ldots,K_x\}$ wartości $f_b(i,q)$ i $f_w(i,q)$ są doklejane do reprezentacji słowa x_i w kroku 1 tworząc zaznaczony na Rysunku 5.1 ciąg ${u'}_1^{K_x}$.

Reprezentacja pytania

Istotnym elementem modelu jest sposób wyliczania podsumowania pytania, $\tilde{z}$. Wykorzystywany jest do tego tzw. *mechanizm uwagi* (ang. *attention*). Poszczególne wektory ciągu $z_1^{K_q}$ są sumowane z wagami oznaczającymi ich istotność w pytaniu. Odbywa się to podobnie jak w przypadku obliczania f_w. Niech $Z = [z_1 \ \cdots \ z_{K_q}] \in \mathbb{R}^{m \times K_q}$, gdzie m jest rozmiarem stanu *RNN*:

$$\alpha = \text{softmax}({\tau_q}^T Z) \in \mathbb{R}^{K_q}$$
$$\tilde{z} = \sum_{j \in \{1,\ldots,K_q\}} \alpha_j z_j$$

Parametr $\tau_q \in \mathbb{R}^m$ jest uczony. Wykorzystując taką reprezentację q sieć potrafi nauczyć się zwracać uwagę na specyficzne słowa. Na przykład obserwując wiele pytań rozpoczynających się od *when* i mających daty za odpowiedzi, będzie w stanie tak dobrać parametry, żeby słowo *when* dostawało dużą wagę. W praktyce typ pytania mocno ogranicza zakres możliwych odpowiedzi. Mechanizm uwagi pozwala modelowi wykorzystywać ten fakt.

Zanurzenia znakowe

Zanurzać w przestrzeni wektorowej można nie tylko słowa, ale także pojedyncze znaki. *FastQA* korzysta zarówno z zanurzeń słów jak i z reprezentacji liter. Niech w będzie słowem złożonym ze znaków $c_1,\ldots,c_l$, a m_c rozmiarem zanurzeń liter. Możemy zapisać w jako $A = [a_1 \ \cdots \ a_l] \in \mathbb{R}^{m_c \times l}$, gdzie a_i jest wektorem dla c_i. Macierz A przekształcamy na wektor stałej długości, który będzie reprezentacją w uwzględniającą wzajemne położenie liter w słowie [Kim et al., 2016].

Wykorzystujemy do tego operację splotu. Dla filtra $H \in \mathbb{R}^{m_c \times k}, k \leq l$ i opcjonalnego przesunięcia $b \in \mathbb{R}$ definiujemy $A * H \in \mathbb{R}^{l-k+1}$ następująco:

$$(A * H)_i = \tanh(\text{tr}([a_i \ \cdots \ a_{i+k-1}] H^T) + b)$$

Interesującym nas rezultatem jest skalar $\max(A*H)$. Całą operację powtarzamy dla pewnej liczby n_f różnych filtrów, a otrzymane wyniki łączymy w n_f-wymiarowy wektor reprezentujący w. Wektor ten jest doklejany do zanurzenia w w kroku 1, przed dodaniem cech f_b i f_w.

Aby zapewnić dodatkową interakcję pomiędzy dwoma typami zanurzeń słów, autorzy wykorzystują warstwę *highway* [Srivastava et al., 2015]. Niech $\mathbf{w} \in \mathbb{R}^{n+n_f}$ będzie konkatenacją obu reprezentacji w. Podlega ona następującym transformacjom:

$$\begin{aligned}\mathbf{w}' &= P\mathbf{w}\\ g &= \mathcal{F}_1(\mathbf{w}')\\ \mathbf{w}'' &= \mathcal{F}_2(\mathbf{w}')\\ \tilde{\mathbf{w}} &= g \odot \mathbf{w}' + (1-g) \odot \mathbf{w}'',\end{aligned}$$

gdzie P jest pewną macierzą, $\mathcal{F}_1$ i $\mathcal{F}_2$ oznaczają warstwy gęste, a $\tilde{\mathbf{w}}$ jest powstałą reprezentacją w, do której konkatenowane będą f_b i f_w. Warstwa *highway* potrafi nauczyć się proporcji w jakich powinna brać pod uwagę oryginalne i przekształcone wejście.

Pozostałe szczegóły

W krokach 4. i 5. do kodowania kontekstu dołączamy dodatkowe informacje, które chcemy uwzględnić przy obliczaniu prawdopodobieństw. Najprościej zrobić to przez konkatenację, ale autorzy dołączają dodatkowo iloczyn Hadamarda:

$$h'_i = \begin{bmatrix} h_i \\ \tilde{z} \\ h_i \odot \tilde{z} \end{bmatrix} \in \mathbb{R}^{3m} \qquad h''_i = \begin{bmatrix} h_i \\ h_{\mathbf{s}} \\ \tilde{z} \\ h_i \odot \tilde{z} \\ h_i \odot h_{\mathbf{s}} \end{bmatrix} \in \mathbb{R}^{5m}$$

Wektory h'_i oraz h''_i są zwijane do skalarów za pomocą podwójnej warstwy gęstej. Funkcja *softmax* zwraca prawdopodobieństwa:

$$\begin{aligned}s_i &= {\tau_s}^T \mathrm{ReLU}(W_s h'_i + b_s) \in \mathbb{R}\\ \hat{P}_s(i \mid q, x) &= (\mathrm{softmax}(s))_i\end{aligned}$$

$$\begin{aligned}e_i &= {\tau_e}^T \mathrm{ReLU}(W_e h''_i + b_e) \in \mathbb{R}\\ \hat{P}_e(i \mid \mathbf{s}, q, x) &= (\mathrm{softmax}(e))_i\end{aligned}$$

Jako warstwę rekurencyjną, kodującą pytanie i kontekst, wykorzystano dwukierunkowy *LSTM* (2.3.5., 2.3.7.). Agregację wyników sieci składowych przeprowadzono za pomocą przepuszczenia połączonych stanów przez warstwę gęstą o parametrze $B \in \mathbb{R}^{m \times 2m}$ (bez przesunięcia). Przetwarzanie q i x zostało przeprowadzone za pomocą tych samych parametrów, za wyjątkiem macierzy B, która była inna dla pytania i kontekstu. Wartości cech f_b i f_w dla słów w pytaniu zostały ustalone na 1.

Warstwa zanurzeń podlega procedurze *dropout* [Srivastava et al., 2014]. Jest to popularna i prosta metodą regularyzacji modelu neuronowego. Każdy element wejścia podlega wyzerowaniu z prawdopodobieństwem p, a następnie całość jest mnożona przez $1 \,/\, (1-p)$, żeby zachować stałą średnią wyniku. W ten sposób symulujemy uśrednianie wielu modeli. Częstym wyborem p jest 0.5.

5.2. Eksperymenty

Model został wyuczony procedurą *ADAM* (2.2.4.) na danych *SQuAD*. Wykorzystane zanurzenia słów pochodzą z GloVe i nie były uczone. Zanurzenia znakowe zostały wytrenowane. Co 64 000 elementów znajdował się punkt kontrolny. Jeśli efektywność na zbiorze testowym malała między punktami kontrolnymi, tempo uczenia było zmniejszane dwukrotnie. Cały proces był dość szybki: 20 przejść po całym zbiorze uczącym zajęło około 4 godziny. Najlepszy rezultat pojawił się już po 9 przejściach.

Miarą jakości modelu jest procent poprawnie udzielonych odpowiedzi na zbiorze testowym. Oficjalny skrypt ewaluacyjny oblicza również miarę F1, stanowiącą balans pomiędzy precyzją (ang. *precision*) i czułością (ang. *recall*). Niestety nie udało mi się zreplikować wyniku z [Weissenborn et al., 2017]. Model uzyskał 72.34 F1, co jest wynikiem trochę niższym od 76.30 F1 prezentowanym w artykule. Powodem mogą być pojawiające się tam drobne niejasności, które mogłem źle zinterpretować. Nie wykluczam oczywiście błędnej implementacji, chociaż nie potrafiłem znaleźć żadnej usterki. Nie mniej jednak otrzymany wynik jest wystarczająco dobry, żeby przekonać się jak sieć sprawdza się w praktyce.

5.2.1. Przykłady zastosowania

Wytrenowana sieć stara się znaleźć podciąg kontekstu, który z największym prawdopodobieństwem jest dobrą odpowiedzią. Sprawdzenie wszystkich możliwych podciągów jest niepraktyczne, więc stosuje się przeszukiwanie wiązkowe (3.5.1.). Zmniejszenie rozmiaru wiązki do 1 skutkuje jednak dużo szybszym algorytmem zachłannym, który działa tylko nieznacznie gorzej. Takiej właśnie wersji używam.

W tej części zamieszczam przykładowe odpowiedzi udzielone przez model. W nawiasach znajdują się zwrócone prawdopodobieństwa. Wartości bliskie 1 oznaczają,

że sieć jest pewna odpowiedzi. Mniejsze liczby sygnalizują trudniejsze przypadki. Pytania i konteksty pochodzą ze zbioru testowego *SQuAD*. Błędne odpowiedzi są opatrzone dodatkowymi komentarzami.

Kontekst: *Other green spaces in the city include the Botanic Garden and the University Library Garden. They have extensive botanical collection of rare domestic and foreign plants, while a palm house in the New Orangery displays plants of subtropics from all over the world. Besides, within the city borders, there are also: Pole Mokotowskie (a big park in the northern Mokotów, where was the first horse racetrack and then the airport), Park Ujazdowski (close to the Sejm and John Lennon street), Park of Culture and Rest in Powsin, by the southern city border, Park Skaryszewski by the right Vistula bank, in Praga. The oldest park in Praga, the Praga Park, was established in 1865–1871 and designed by Jan Dobrowolski. In 1927 a Zoological Garden (Ogród Zoologiczny) was established on the park grounds, and in 1952 a bear run, still open today.*

P: *What type of space in Warsaw are the Botanic Garden and University Library Garden?*
O: *green* (0.81)

P: *Where is a palm house with subtropic plants from all over the world on display?*
O: *New Orangery* (0.35)

P: *Where was the first horse racetrack located?*
O: *pole mokotowskie* (0.34)

P: *What park is close to John Lennon street?*
O: *park ujazdowski* (0.95)

P: *When was a zoological garden established in the Praga Park?*
O: *1865 – 1871* (0.77) (błąd, poprawna odpowiedź: *1927*)

Kontekst: *Following the cretaceous–paleogene extinction event , the extinction of the dinosaurs and the wetter climate may have allowed the tropical rainforest to spread out across the continent. From 66–34 mya, the rainforest extended as far south as 45°. Climate fluctuations during the last 34 million years have allowed savanna regions to expand into the tropics. During the oligocene, for example, the rainforest spanned a relatively narrow band. It expanded again during the middle miocene, then retracted to a mostly inland formation at the last glacial maximum. However, the rainforest still managed to thrive during these glacial periods, allowing for the survival and evolution of a broad diversity of species.*

P: *Which type of climate may have allowed the rainforest to spread across the continent?*
O: *tropical* (0.52) (błąd, poprawna odpowiedź: *wetter*)

P: *What has allowed for the savanna region to expand into the tropics?*
O: *climate fluctuations* (0.98)

P: *During what time did the rainforest spanned a narrow band?*
O: *the oligocene ,* (0.20) (nieścisłość: niepotrzebny przecinek)

P: *When did it retract to a inland formation?*
O: *middle miocene* (0.49)

P: *Did the rainforest managed to thrive during the glacial periods?*
O: *still managed to thrive* (0.31)

Kontekst: *QuickBooks sponsored a "Small Business Big Game" contest, in which Death Wish Coffee had a 30-second commercial aired free of charge courtesy of QuickBooks. Death Wish Coffee beat out nine other contenders from across the United States for the free advertisement.*

P: *What company won a free advertisement due to the QuickBooks contest?*
O: *coffee* (0.34) (niepełna odpowiedź, poprawna: *death wish coffee*)

P: *How long was the Death Wish Coffee commercial?*
O: *30 - second* (0.78)

P: *Besides Death Wish Coffee, how many other competitors participated in the contest?*
O: *nine* (0.63)

P: *Which company sponsored a contest called "Small Business Big Game"?*
O: *quickbooks* (1.0)

P: *How many other contestants did the company, that had their ad shown for free, beat out?*
O: *nine* (0.80)

Kontekst: *Sir Charles Lyell first published his famous book, Principles of Geology, in 1830. This book, which influenced the thought of Charles Darwin, successfully promoted the doctrine of uniformitarianism. This theory states that slow geological processes have occurred throughout the Earth's history and are still occurring today. In contrast, catastrophism is the theory that Earth's features formed in single, catastrophic events and remained unchanged thereafter. Though Hutton believed in uniformitarianism, the idea was not widely accepted at the time.*

P: *First published by Sir Charles Lyell in 1830 this book was called what?*
O: *principles of geology* (0.96)

P: *What doctrine did the doctrine of the Principles of Geology successfully promote?*
O: *the doctrine of uniformitarianism* (0.63)

P: *Which theory states that slow geological processes are still occurring today, and have occurred throughout Earth's history?*
O: *uniformitarianism* (0.48)

P: *Which theory states that Earth's features remained unchanged after forming in one single catastrophic event?*
O: *catastrophis*m (1.0)

P: *Which famous evolutionist was influenced by the book Principles of Geology?*
O: *charles darwin* (0.65)

5.3. Negatywne odpowiedzi

W zbiorze *SQuAD* każde pytanie jest połączone z pasującym do niego kontekstem i *FastQA* korzysta z tej własności. Model uczy się znajdować dobre odpowiedzi we fragmentach, które faktycznie je zawierają. W prawdziwym zastosowaniu trudno oczekiwać, że właściwy tekst zawsze będzie pod ręką. Niestety opisana architektura w żaden sposób nie potrafi wykrywać, czy właściwa odpowiedź znajduje się w dostarczonym jej fragmencie. Prawdopodobieństwa, które zwraca, są warunkowe, i nie

mogą być rozważane bez kontekstu. Żeby to zaobserwować, weźmy poprzednio przytoczony fragment o bitwie pod Grunwaldem, oraz krótki tekst o Chopinie:

1. *The Battle of Grunwald, First Battle of Tannenberg or Battle of Žalgiris, was fought on* ***15 July 1410*** *during the Polish–Lithuanian–Teutonic War. The alliance of the Kingdom of Poland and the Grand Duchy of Lithuania, led respectively by King Władysław II Jagiełło (Jogaila) and Grand Duke Vytautas, decisively defeated the German–Prussian Teutonic Knights, led by Grand Master Ulrich von Jungingen.*

2. *Born on March 1, 1810, in Zelazowa Wola, Poland, Frédéric Chopin, grew up in a middle-class family. He published his first composition at age 7 and began performing one year later. In 1832, he moved to Paris, socialized with high society and was known as an excellent piano teacher. His piano compositions were highly influential.*[3]

Dla pytania *When was Battle of Grunwald?* algorytm wysoko ocenia odpowiedzi znalezione w obu tekstach:

O1: *15 july 1410* (0.95)

O2: *march 1 , 1810* (0.88)

Widząc pytanie o czas, model, opierając się na założeniu, że odpowiedź znajduje się w tekście, przypisuje datom duże prawdopodobieństwo. Drugi tekst nie zawiera jednak właściwej odpowiedzi, czego nie jesteśmy w stanie stwierdzić patrząc tylko na wynik.

Ten podrozdział opisuje moją próbę nauczenia *FastQA* wykrywania niepasujących kontekstów. Oczekujemy, że sieć będzie umiała wyłuskać dobrą odpowiedź z pierwszego z powyższych fragmentów, oraz stwierdzić, że w przypadku drugiego nie potrafi tego zrobić.

Motywacją do przeprowadzenia eksperymentu były problemy napotkane podczas tworzenia chatbota `poetwanna.be` (1.1.). System miał odpowiadać na pytania dwuetapowo. Najpierw przeprowadzał wyszukiwanie w Wikipedii. Następnie zgromadzone teksty były przetwarzane przez *FastQA*, a najbardziej prawdopodobna odpowiedź stanowiła rezultat algorytmu. Nasza wyszukiwarka była jednak dość prosta, więc nie zawsze udawało się znaleźć pasujące fragmenty. Trudno było ocenić jakość wyniku, ponieważ model mógł zwracać wysokie prawdopodobieństwo odpowiedzi nawet dla niewłaściwych akapitów. Gdyby zamiast tego potrafił je ignorować, współpraca wyszukiwarki i *FastQA* byłaby dużo efektywniejsza.

Pomysł opiera się na lekkiej zmianie procesu uczenia. Oprócz pozytywnych przykładów pokażemy sieci także teksty, które nie zawierają odpowiedzi. Nie ingerujemy

[3]Fragment pochodzi z https://www.biography.com/people/frederic-chopin-9247162

w żaden sposób w architekturę *FastQA*, ale dostosowujemy dane do jej ograniczeń. Ponieważ model zakłada, że zawsze musi odpowiedzieć pozytywnie, doklejamy do wszystkich kontekstów sztuczne słowo **neg**. Uznajemy je za poprawną odpowiedź we wszystkich przypadkach, w których fragment nie odpowiada na pytanie. Fragmenty z przykładu (po tokenizacji, czyli podzieleniu napisu na słowa) będą wyglądały tak:

the battle of grunwald , first battle [...] *jungingen .* **neg**

born on march 1, 1810, in zelazowa [...] *influential .* **neg**

W pierwszym przypadku nadal spodziewamy się zobaczyć wynik *15 july 1410*. W drugim mamy nadzieję na **neg**, co pozwoli nam wywnioskować, że fragment nie zawiera właściwej odpowiedzi. Podczas uczenia dalej korzystamy z ustalonych zanurzeń GloVe, za wyjątkiem wektora dla słowa **neg**, który jest optymalizowany.

5.3.1. Wzbogacanie danych

Przygotowałem cztery dodatkowe zestawy danych uczących. Dwa z nich są przerobionymi wersjami *SQuAD*:

- `SQuAD-rng` to zbiór uczący *SQuAD* z pomieszanymi kontekstami. Każde pytanie dostało losowo wybrany, niezawierający odpowiedzi, i niepasujący do niego tematycznie fragment.
- `SQuAD-cut` zawiera te same pary (q, x), co dane uczące *SQuAD*, ale w każdej parze z x wycięte zostało zdanie zawierające odpowiedź. Sprawia to, że teksty są na temat, ale nie zawierają odpowiedzi. Obserwacja tego zbioru może pomóc sieci zrozumieć, że nawet mocno związany tematycznie fragment nie musi być właściwy.

Robot `poetwanna.be` szukał odpowiedzi w Wikipedii, co starałem się wziąć pod uwagę w procesie uczenia. W pozostałych dwóch zbiorach pytania pochodzą ze *SQuAD*u, a konteksty z Wikipedii. Akapity w danych *SQuAD* mogą mieć inną charakterystykę niż te zwracane przez wyszukiwarkę. Ich powiązanie z pytaniem również może wyglądać inaczej. Jeśli więc wszystkie przykłady z Wikipedii byłyby negatywne, model mógłby nauczyć się odrzucać je ze względu na cechy inne niż brak właściwej odpowiedzi. Mając to na uwadze, jeden z poniższych zbiorów zawiera pozytywne konteksty:

- `Wiki-pos` składa się z pytań *SQuAD* połączonych z pozytywnymi akapitami znalezionymi w Wikipedii. Nie udało się znaleźć pasujących fragmentów dla każdego pytania, więc ten zbiór zawiera tylko 52 600 elementów, co stanowi 61% całego *SQuAD*.

- `Wiki-neg` jest zbudowany tak jak `Wiki-pos`, ale zawiera tylko negatywne przykłady. Akapity pochodzą z wyników wyszukiwarki, więc są w pewien sposób tematycznie związane z pytaniami.

Analogiczne zbiory zostały utworzone dla zbioru testowego *SQuAD*, `SQuAD-dev`. Korzystałem z `SQuAD-rng-dev`, `Wiki-pos-dev`, `Wiki-neg-dev`, oraz `SQuAD-dev`, żeby zmierzyć efektywność sieci wyuczonych w różnych warunkach.

5.3.2. Wyniki

Eksperyment polegał na wzbogacaniu oryginalnego zbioru uczącego *SQuAD* różnymi kombinacjami czterech dodatkowych zestawów danych. Weryfikacja jakości modelu polegała na sprawdzeniu ile kontekstów zostało odrzuconych z każdego ze zbiorów testowych. Idealny model udzieliłby poprawnych odpowiedzi dla każdego przykładu w `Wiki-pos-dev` i oryginalnym `SQuAD-dev`, a odrzucił wszystkie fragmenty znajdujące się w `SQuAD-rng-dev` i `Wiki-neg-dev`.

Wyniki przedstawione są w Tabeli 5.1. Każdy wiersz reprezentuje jeden model. Kolumna **Dane** zawiera zbiory danych, o które został rozszerzony zbiór uczący *SQuAD*. Sprawdzam również jak bardzo postawienie przed architekturą dodatkowej trudności wpływa na wynik w oryginalnym zadaniu. **Pos F1** oznacza F1 na `SQuAD-dev`, gdy model nie może udzielić odpowiedzi negatywnej (bez doklejania **neg**). **Neg F1** jest mierzone tak samo, ale ze słowem **neg**. Wartości te stanowią miarę pogorszenia jakości rozwiązywania podstawowego problemu *SQuAD*.

W Tabeli 5.1 widzimy, że wybór danych uczących ma olbrzymi wpływ na wynik. Lepiej przygotowane negatywne przykłady prawdopodobnie poprawiłyby rezultat. Można też zauważyć kompromis między liczbą odrzucanych niewłaściwych kontekstów a liczbą pozytywnych przykładów, które rozwiązujemy poprawnie. Wybór "najlepszego" modelu jest z tego powodu trudny.

Ostatecznie bot `poetwanna.be` nie korzystał z negatywnych modeli; tracimy zbyt dużo poprawnych odpowiedzi, żeby sobie na to pozwolić. Doświadczenie pokazuje jednak, że odrzucanie złych kontekstów ma potencjał. Odpowiednie modyfikacje sieci lub danych uczących mogłyby umożliwić zastosowanie tego pomysłu w praktyce.

5.3.3. Przykłady negatywnych odpowiedzi

Ta część zawiera przykłady odpowiedzi udzielanych przez przedostatni model z Tabeli 5.1. Najpierw przypomnijmy wcześniej przytoczone fragmenty o bitwie pod Grunwaldem i Chopinie. **p** oznacza model pozytywny, a **n** negatywny.

Tabela 5.1: Procent odrzuconych fragmentów dla różnych modeli

Dane	Wiki-pos	SQuAD	Wiki-neg	SQuAD-rng	neg F1	pos F1
	0	0	0.01	0.03	72.36	72.34
SQuAD-rng	0.01	0.02	0.07	0.98	69.73	70.34
SQuAD-cut	0.25	0.22	0.62	1	60.99	70.5
Wiki-neg	0.53	0.27	0.92	0.53	55.11	70.6
Wiki-neg Wiki-pos	0.4	0.23	0.87	0.65	58.72	70.19
SQuAD-rng Wiki-neg Wiki-pos	0.35	0.23	0.82	0.99	57.83	69.15
SQuAD-rng SQuAD-cut	0.26	0.22	0.63	1	61.05	71.05

1. *The Battle of Grunwald, First Battle of Tannenberg or Battle of Žalgiris, was fought on* ***15 July 1410*** *during the Polish–Lithuanian–Teutonic War. The alliance of the Kingdom of Poland and the Grand Duchy of Lithuania, led respectively by King Władysław II Jagiełło (Jogaila) and Grand Duke Vytautas, decisively defeated the German–Prussian Teutonic Knights, led by Grand Master Ulrich von Jungingen.*

2. *Born on March 1, 1810, in Zelazowa Wola, Poland, Frédéric Chopin, grew up in a middle-class family. He published his first composition at age 7 and began performing one year later. In 1832, he moved to Paris, socialized with high society and was known as an excellent piano teacher. His piano compositions were highly influential.*

O1 p: *15 july 1410* (0.95)
O1 n: *15 july 1410* (0.54)

O2 p: *march 1 , 1810* (0.88)
O2 n: **neg** (0.96)

Model negatywny zachował się zgodnie z oczekiwaniami. Potrafił znaleźć właściwą odpowiedź w pierwszym tekście oraz stwierdzić, że drugi jej nie zawiera. Należy jednak zwrócić uwagę na mniejsze prawdopodobieństwo w pierwszym przypadku. Oznacza to dużo większą niepewność modelu.

Dalej zamieszczam kilka kolejnych przykładów. Na temat każdego akapitu zadaję trzy pytania z możliwością odpowiedzi, i trzy, na które nie da się odpowiedzieć na podstawie fragmentu. Pytania, które są w pewien sposób związane z tekstem, są znacznie trudniejsze. Brak komentarza oznacza poprawną odpowiedź. Podobnie jak powyżej, można zaobserwować większą niepewność modelu negatywnego na pytaniach pozytywnych.

Kontekst: *Jacksonville is the largest city by population in the U.S. state of Florida, and the largest city by area in the contiguous United States. It is the county seat of Duval County, with which the city government consolidated in 1968. Consolidation gave Jacksonville its great size and placed most of its metropolitan population within the city limits; with an estimated population of 853,382 in 2014, it is the most populous city proper in Florida and the southeast, and the 12th most populous in the United States . Jacksonville is the principal city in the Jacksonville metropolitan area, with a population of 1,345,596 in 2010.*

P: *Which Florida city has the biggest population?*
O p: *jacksonville* (0.92)
O n: *jacksonville* (0.55)

P: *What was the population Jacksonville city as of 2010?*
O p: *1 , 345 , 596* (1.0)
O n: *1 , 345 , 596* (0.41)

P: *Based on population alone, what is Jacksonville's ranking in the United States?*
O p: *12th* (0.63)
O n: *12th* (0.45)

P: *What is the population of Los Angeles?*
O p: *853 , 382* (0.68) (błąd, brak odpowiedzi)
O n: **neg** (0.63)

P: *What is the smallest city in Oklahoma?*
O p: *jacksonville* (0.80) (błąd, brak odpowiedzi)
O n: **neg** (0.60)

P: *Who founded Jacksonville?*
O p: *population* (0.22) (błąd, brak odpowiedzi)
O n: *population in the u . s . state of florida* (0.35) (błąd, brak odpowiedzi)

Kontekst: *Luther justified his opposition to the rebels on three grounds. First, in choosing violence over lawful submission to the secular government, they were ignoring Christ's counsel to "Render unto Caesar the things that are Caesar's"; St. Paul had written in his epistle to the Romans 13:1–7 that all authorities are appointed by God and therefore should not be resisted. This reference from the Bible forms the foundation for the doctrine known as the divine right of kings, or, in the German case, the divine right of the princes. Second, the violent actions of rebelling, robbing, and plundering placed the peasants "outside the law of God and Empire", so they deserved "death in body and soul, if only as highwaymen and murderers." Lastly, Luther charged the rebels with blasphemy for calling themselves "Christian brethren" and committing their sinful acts under the banner of the Gospel.*

P: *What were the protesters doing with Christ's counsel?*
O p: *render unto caesar the things* (0.41) (błąd, poprawna odpowiedź: *ignoring*)
O n: **neg** (0.36) (błąd, poprawna odpowiedź: *ignoring*)

P: *By whom did St Paul say all authorities were appointed?*
O p: *god* (1.0)
O n: *god* (0.89)

P: *What is this doctrine of God appointing authorities called?*

O p: *the divine right of kings* (0.64)
O n: **neg** (0.39) (błąd, odrzucenie dobrego fragmentu)

P: *Who was Martin Luther?*
O p: *the rebels with blasphemy* (0.29) (błąd, brak odpowiedzi)
O n: **neg** (0.72)

P: *How many epistles did St. Paul write?*
O p: *13 : 1 – 7* (0.62) (błąd, brak odpowiedzi)
O n: *three* (0.41) (błąd, brak odpowiedzi)

P: *When did Luther annouce his three treatises?*
O p: *second* (0.22) (błąd, brak odpowiedzi)
O n: **neg** (0.96)

Kontekst: *In a report, published in early February 2007 by the Ear Institute at the University College London, and Widex, a Danish hearing aid manufacturer, Newcastle was named as the noisiest city in the whole of the UK, with an average level of 80.4 decibels. The report claimed that these noise levels would have a negative long-term impact on the health of the city's residents. The report was criticized, however, for attaching too much weight to readings at arbitrarily selected locations, which in Newcastle's case included a motorway underpass without pedestrian access.*

P: *What's the average decibel level of noise in Newcastle?*
O p: *80 . 4 decibels* (0.72)
O n: *80 . 4 decibels* (0.83)

P: *What type of impact can the residents of Newcastle expect the city's noise to have on them?*
O p: *negative long - term* (0.35)
O n: *negative long - term impact* (0.41)

P: *What was one location the noise readings in Newcastle were taken at?*
O p: *motorway underpass* (0.25)
O n: **neg** (0.27) (błąd, odrzucenie dobrego fragmentu)

P: *When was Widex founded?*
O p: *february 2007* (0.82) (błąd, brak odpowiedzi)
O n: **neg** (0.35)

P: *How many UK citizens have hearing problems?*
O p: *80 . 4* (0.46) (błąd, brak odpowiedzi)
O n: **neg** (0.57)

P: *What is the average length of a motorway?*
O p: *80 . 4 decibels* (0.93) (błąd, brak odpowiedzi)
O n: *80 . 4 decibels* (0.61) (błąd, brak odpowiedzi)

Rozdział 6.

Podsumowanie

Sieci neuronowe znalazły w ostatnich latach szerokie zastosowanie w dziedzinie przetwarzania języka, na czym naturalnie skorzystały systemy dialogowe. Generowanie zdań słowo po słowie jest ciekawą perspektywą, która uwalnia nas od konieczności ręcznej konstrukcji wypowiedzi i daje maszynie możliwość wykazania oryginalności. Niestety obecnie to rozwiązanie boryka się z dość fundamentalnymi przeszkodami. Różnorodność wypowiedzi i dobre zrozumienie kontekstu są bardzo trudne do uzyskania. Automatyczna ocena jakości generowanych dialogów dobrze korelująca z osądem ludzkim cały czas pozostaje otwartym problemem. Niemniej jednak jest to interesująca gałąź sztucznej inteligencji, ciesząca się zainteresowaniem w środowiskach naukowych.

Wyszukiwanie odpowiedzi na pytania w zadanym kontekście daje znacznie bardziej imponujące wyniki. Najlepsze systemy osiągają w tym zadaniu skuteczność bardzo bliską ludzkiej, co pozwala z powodzeniem wykorzystywać je w praktyce. Sporą trudnością pozostaje jednak znalezienie odpowiedniego tekstu do konkretnego pytania. Wykorzystanie standardowych technik przeszukiwania nie zawsze daje pożądane rezultaty. Optymalny system powinien być wyposażony w wyszukiwarkę specjalnie dostosowaną do tego problemu lub potrafić samemu ocenić dopasowanie otrzymywanych fragmentów.

Niemałą rolę w rozwoju systemów dialogowych odgrywa rywalizacja. Konkursy takie jak The Alexa Prize[1], Nagroda Loebnera, czy NIPS Conversational Intelligence Challenge stanowią coroczną motywację dla naukowców i programistów. Łatwa dostępność chatbotów, na przykład za pośrednictwem Facebooka, sprawia, że są one coraz mocniej obecne w świadomości publicznej. Mnogość interaktywnych aplikacji asystujących użytkownikowi pokazuje, że roboty konwersacyjne mają również potencjał w biznesie. Systemy dialogowe stają się więc coraz bardziej popularne i zaawansowane.

[1] `https://developer.amazon.com/alexaprize`

Bibliografia

[Bengio et al., 2003] Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, 3.

[Bengio et al., 1994] Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. IEEE Transactions on Neural Networks.

[Cho et al., 2014] Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder–decoder for statistical machine translation. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP).

[Chung et al., 2014] Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. Presented at the Deep Learning workshop at NIPS2014.

[Duchi et al., 2011] Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12.

[Forcada and Ñeco, 1997] Forcada, M. L. and Ñeco, R. P. (1997). Recursive hetero-associative memories for translation. In *IWANN*.

[Hinton et al., 2013] Hinton, G., Srivastava, N., and Swersky, K. (2013). Neural networks for machine learning. `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`. Coursera class, Lecture 6e.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, Volume 9 Issue 8.

[Jean et al., 2015] Jean, S., Cho, K., Memisevic, R., and Bengio, Y. (2015). On using very large target vocabulary for neural machine translation. Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing.

[Kim et al., 2016] Kim, Y., Jernite, Y., Sontag, D., and Rush, A. M. (2016). Character-aware neural language models. Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence.

[Kingma and Ba, 2015] Kingma, D. P. and Ba, J. L. (2015). Adam: A method for stochastic optimization. ICRL 2015.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. Proceedings of the 25th International Conference on Neural Information Processing Systems.

[Leskovec and Krevl, 2014] Leskovec, J. and Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`.

[Liu et al., 2016] Liu, C.-W., Lowe, R., Serban, I., Noseworthy, M., Charlin, L., and Pineau, J. (2016). How not to evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation. Conference: Annual Meeting of the Association for Computational Linguistics (ACL).

[Lowe et al., 2015] Lowe, R., Pow, N., Serban, I. V., and Pineau, J. (2015). The ubuntu dialogue corpus: A large dataset for research in unstructured multi-turn dialogue systems. Proceedings of the SIGDIAL 2015 Conference.

[Magarreiro et al., 2014] Magarreiro, D., Coheur, L., and Melo, F. S. (2014). Using subtitles to deal with out-of-domain interactions. SemDial 2014 – DialWatt.

[Mikolov et al., 2010] Mikolov, T., Karafiat, M., Burget, L., Cernock, J. H., and Khudanpur, S. (2010). Recurrent neural network based language model. INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association.

[Mikolov et al., 2013] Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013). Distributed representations of words and phrases and their compositionality. Advances in Neural Information Processing Systems 26 (NIPS 2013).

[Pennington et al., 2014] Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing.

[Rajpurkar et al., 2016] Rajpurkar, P., Zhang, J., Lopyrev, K., and Liang, P. (2016). Squad: 100,000+ questions for machine comprehension of text. Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing.

[Rosenblatt, 1958] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, Volume 65, No. 6.

[Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representation by error propagation. Vol. 1.

[Serban et al., 2015] Serban, I. V., Sordoni, A., Bengio, Y., Courville, A., and Pineau, J. (2015). Building end-to-end dialogue systems using generative hierarchical neural networks. 30th AAAI Conference on Artificial Intelligence.

[Sordoni et al., 2014] Sordoni, A., Bengio, Y., Vahabi, H., Lioma, C., Simonsen, J. G., and Nie, J.-Y. (2014). A hierarchical recurrent encoder-decoder for generative context-aware query suggestion. 24th ACM International Conference on Information and Knowledge Management.

[Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15.

[Srivastava et al., 2015] Srivastava, R. K., Greff, K., and Schmidhuber, J. (2015). Highway networks. *CoRR*, abs/1505.00387.

[Toderici et al., 2017] Toderici, G., Vincent, D., Johnston, N., Hwang, S. J., Minnen, D., Shor, J., and Covell, M. (2017). Full resolution image compression with recurrent neural networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[Turing, 1950] Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, (236).

[Vijayakumar et al., 2016] Vijayakumar, A. K., Cogswell, M., Selvaraju, R. R., Sun, Q., Lee, S., Crandall, D., and Batra, D. (2016). Diverse beam search: Decoding diverse solutions from neural sequence models. arXiv preprint arXiv:1610.02424.

[Wallace, 2009] Wallace, R. S. (2009). The anatomy of alice. In *Parsing the Turing Test*, pages 181–210. Springer.

[Weissenborn et al., 2017] Weissenborn, D., Wiese, G., and Seiffe, L. (2017). Making neural qa as simple as possible but not simpler. Proceedings of the 21st Conference on Computational Natural Language Learning.

[Zeiler, 2012] Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *CoRR*, abs/1212.5701.