WEDT projekt

Autorzy:

Daniel Iwanicki
Emil Bałdyga
Kamil Zych

Temat: Rekurencyjna sieć neuronowa z różnymi parametrami.

Środowisko działania

Projekt był rozwijany na laptopie z dostępem do pamięci RAM o pojemności 16GB i partycji SWAP o pojemności kolejnych 16 GB, oraz z dostępem do karty graficznej **GTX 1050Ti**.

Pewien etap projektu został zrealizowany także na platformie Google Colab, jednak jej jedyną zaletą był fakt, że nie występowało przegrzewanie się własnego komputera a w rezultacie efekt throttlingu. Nie licząc wspomnianej zalety mieliśmy w niej dostęp do mniejszej ilości pamięci RAM, szybkość trenowania była porównywalna, a pozostawienie sieci do trenowania przez dłuższy czas często powodowała, że środowisko przestawało działać i wymagało to jego restartu.

Projekt został napisany w języku **Python** w wersji **3.7** przy użyciu biblioteki **PyTorch** rozwijanej przez firmę Facebook.

Przygotowanie danych

Praca zespołu rozpoczęła się od zamiaru wybrania danych, na których sieć LSTM będzie trenowana.

Zagadnienie to okazało się dużo bardziej skomplikowane, niż z początku się wydawało, gdyż trudność sprawiło nam znalezienie odpowiedniego korpusu, który spełniał wszystkie następujące warunki:

- posiadał odpowiednią liczbę przykładów trenujących: dostatecznie dużą, by była to reprezentatywna próba, ale jednocześnie na tyle małą, by obliczenia na naszych skończonych zasobach obliczeniowych trwały w akceptowalnym czasie
- był zrównoważony,

Ostatecznie nasz zespół zdecydował się skorzystać z korpusu udostępnianego przez firmę Amazon złożonego z recenzji różnych produktów, które to (recenzje) klasyfikowane były do jednej z dwóch klas: **pozytywna** lub **negatywna**.

Istota projektu w żaden sposób nie ucierpiała z powodu zadowolenia się przez nas zbiorem klasyfikowanym binarnie, zaś nam pozwoliła skorzystać ze zbioru który bardzo łatwo i bez straty można było zrównoważyć i którego prostota niesłychanie ułatwiła nam diagnozowanie występujących w trakcie pracy błędów.

Udostępniony przez firmę Amazon korpus składa się z ponad 142 milionów przykładów, jednak z wyżej wymienionych powodów wydzieliliśmy jego podzbiór:

Zbiór trenujący wykorzystany przez nas był złożony z **800 000 elementów**.

Zbiór testujący złożony był z **200 000 elementów**.

Reprezentacja przykładów trenujących i walidacyjnych

Z początku wykorzystywaliśmy gotowy, wytrenowany model **word2vec** o nazwie **gensim**, który przypisywał słowom 300-wymiarowe wektory. Niestety szybko okazało się, że choć reprezentacja słów jest w ten sposób bardzo dobra, to wykorzystanie tego modelu nie miało szans powodzenia - aby program przez większość czasu nie zajmował się tłumaczeniem słów tylko mógł zająć się uczeniem, konieczne było wcześniejsze przygotowanie danych do postaci gotowych macierzy. Aby uniknąć każdorazowej ich generacji która trwała niepomijalną ilocź czasu, wykorzystaliśmy bibliotekę **pickle** do zapisu na dysk twardy macierzy, co znacznie skróciło czas ich wczytywania do pamięci RAM w kolejnych próbach nauki.

Niestety okazało się, że o ile taki model przechowywania danych pozwalał na szybki dostęp, o tyle chcąc wykorzystać w modelu *batch* o rozmiarze większym niż 1 (czyli podawać do sieci na raz więcej niż jeden przykład trenujący), tablicę przykładów o różnej długości trzeba było dopełnić zerami, by każdy przykład był jednakowo długi. Po tej operacji macierze przestawały się mieścić w pamięci RAM i po kilku kolejnych próbach porzuciliśmy koncepcję wykorzystania sieci gensim.

Ostatecznie w celu translacji słów na wektory wykorzystaliśmy na wejściu sieci warstwę **Embedding**, co rozwiązało problem wektoryzacji słów.

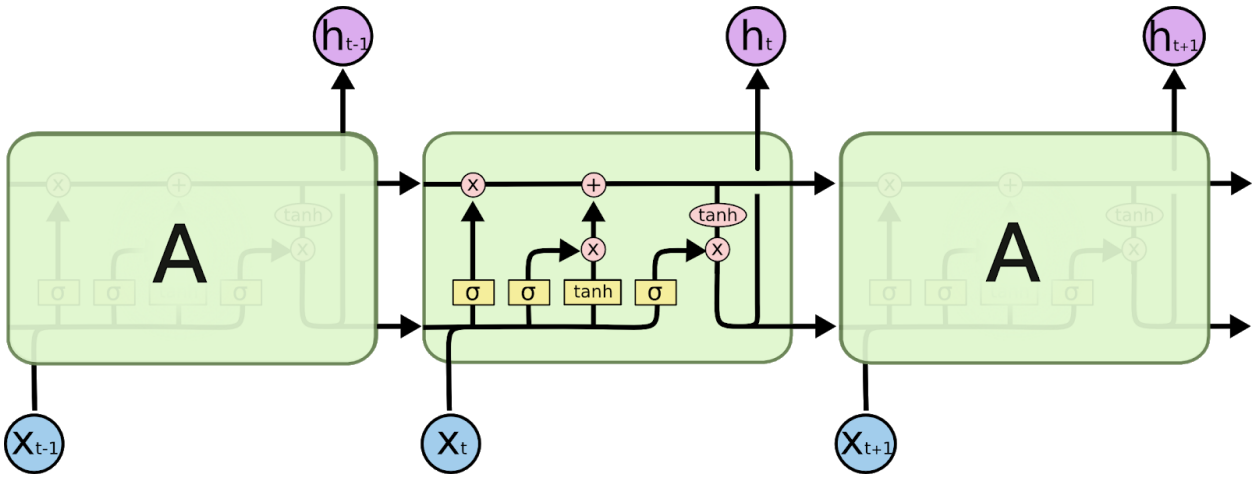
Tensorflow vs Pytorch

Jak już zostało wspomniane, zdecydowaliśmy się wykorzystać bibliotekę PyTorch, jednak decyzja ta poprzedzona została wcześniejszą próbą implementacji warstwy LSTM w bibliotece Tensorflow. Niestety na pewnym etapie zaniechaliśmy prób ze względu na znaczące trudności związane z wymiarami kolejnych warstw sieci oraz danych wejściowych, warstwy ukrytej i danych wyjściowych.

Kolejnym powodem wyboru biblioteki Pytorch był dla nas fakt, iż Tensorflow, w przeciwieństwie do Pytorch, nie ma możliwości tworzenia dynamicznych grafów obliczeniowych dla sieci. Możliwość zmiany struktury sieci podczas jej działania była zaś dla naszego projektu (jak nam się przez większą część czasu pracy wydawało) niezbędna, skoro w zależności od rozkładu POS danego, aktualnie przetwarzanego słowa, miało ono być propagowane przy pomocy odpowiedniej warstwy z własną macierzą wag.

Ostatecznie zrealizowaliśmy projekt na grafie statycznym, co będzie szczegółowo opisane w dalszej części.

LSTM - wektor stanu i wektor wyjścia.



Źródło: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

W trakcie analizowania sposobu działania sieci LSTM naszą uwagę zwróciła pewna nieścisłość między teorią, a rzeczywistymi realizacjami warstw LSTM. Korzystając z gotowej, bibliotecznej implementacji warstwy LSTM osobno można podać rozmiar wejścia, warstwy ukrytej oraz wyjścia, podczas gdy według teorii (która została zilustrowana powyżej) rozmiar **wyjścia** oraz **warstwy ukrytej** musi być jednakowy.

Wynika to z faktu, że aby możliwa była operacja przemnożenia wektorów stanu oraz wektora wyjścia, które następuje na końcowym etapie środkowej instancji warstwy narysowanej powyżej, rozmiary tych wektorów muszą być jednakowe.

W naszym przypadku miało to istotny wpływ na strukturę sieci - wyjście składało się bowiem z jednego neuronu. Gdyby z tego powodu nasza sieć miała posiadać jeden neuron warstwy ukrytej, szansa, że czegokolwiek by się nauczyła byłaby znikoma.

Po analizie tego problemu doszliśmy do wniosku, że ustawimy rozmiar wyjścia i wektora stanu na taki, jaki powinien mieć według nas wektor stanu (czyli znacząco więcej niż 1), a przed wyjściem dołożyliśmy jeszcze warstwę **fully-connected** (w bibliotece PyTorch nazywanej **Linear**) o wyjściu o rozmiarze 1.

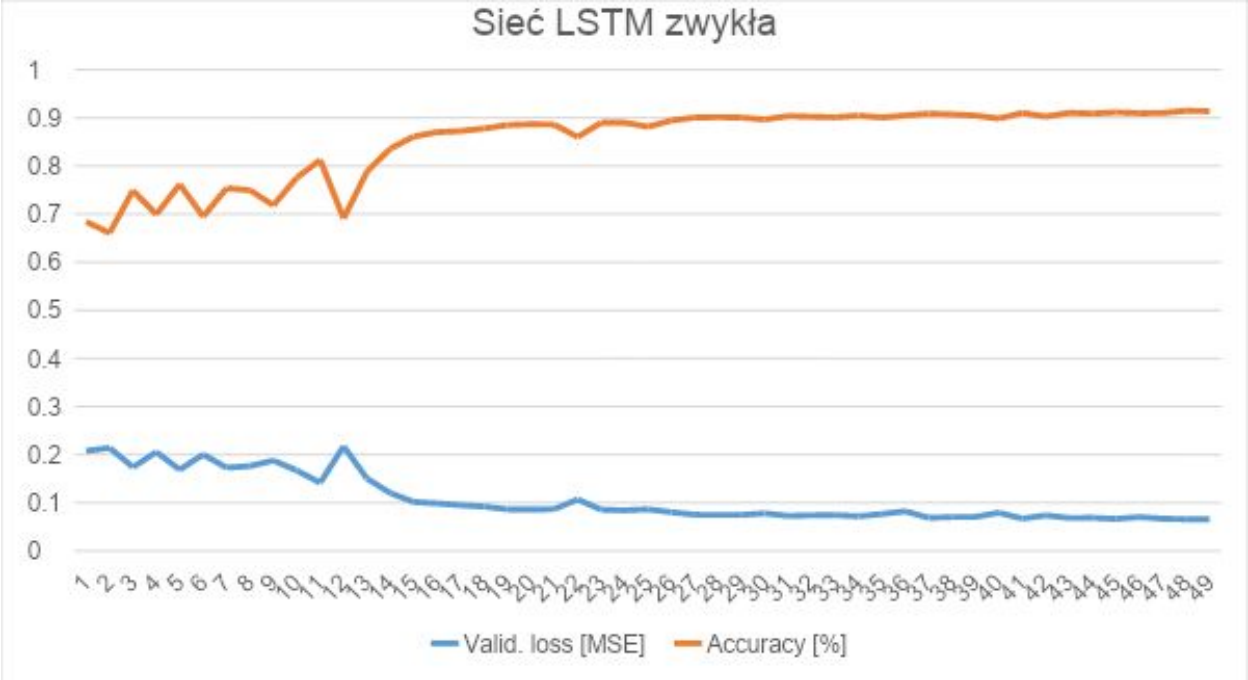
Etap 1

Pierwsza sieć uczona była przy wykorzystaniu batcha o rozmiarze 64 przykładów i warstwie ukrytej zawierającej 132 neurony.

Początkowo wybraliśmy rozmiar warstwy ukrytej (wektora stanu) na 32 neurony. Przy tym ustawieniu efektywność uczenia się sieci była bardzo niska, dlatego stopniowo zwiększyliśmy ją do rozmiaru 128 neuronów. Wartość 128 okazała się jednak momentami problematyczna, ponieważ taki sam rozmiar w niektórych przypadkach miał batch - w sytuacjach, gdy konieczne było prześledzenie sposobu działania sieci nie było oczywiste do którego wymiaru odnosi się wartość 128, przez co ostatecznie zdecydowaliśmy się zwiększyć rozmiar warstwy ukrytej do 132.

Pierwszym etapem prac było ustalenie wartości odniesienia do jakiej był porównywany przyszły wynik sieci LSTM-POS. W tym celu wytrenowaliśmy zwykłą sieć z warstwą LSTM. Zdecydowaliśmy się nie korzystać z gotowej implementacji warstwy LSTM oferowanej przez bibliotekę PyTorch, tylko napisać tę warstwę sami przy użyciu dostępnych w bibliotece operacji macierzowych, aby po rozbudowaniu warstwy o interpretację tagów POS mieć pewność, że nie “zyskaliśmy”, ani też nie “straciliśmy” żadnej nieplanowanej cechy jaką mogła mieć domyślna, biblioteczna implementacja sieci LSTM.

Na poniższym wykresie przedstawiono wyniki uczenia sieci - na czerwono *accuracy*, czyli stosunek poprawnie zaklasyfikowanych przykładów walidacyjnych do wszystkich przykładów walidacyjnych, na niebiesko błąd średniokwadratowy.



Jak widać, sieć LSTM osiągnęła skuteczność rzędu 91%, co uznaliśmy za sukces. Skuteczność na poziomie ok 90-92% ustabilizowała się mniej więcej w połowie pierwszej epoki.

Czas uczenia: **ok. 5h**

Etap 2

Etap drugi składał się z poniższych “podetapów”.

1. Przede wszystkim kluczową kwestią był odpowiedni rozkład danych i przypisanie im tagów POS. W pierwszym odruchu skorzystaliśmy z biblioteki **nltk**, jednak szybko byliśmy zmuszeni z tego zrezygnować - liczba tagów w tej bibliotece była zbyt duża (35). Oznaczało to, że proporcjonalnie do liczby tagów wzrośnie liczba parametrów w sieci, a to rodziło wątpliwości, czy chcąc zachować dużą warstwę ukrytą, liczba przykładów trenujących w porównaniu do liczby parametrów sieci jest wystarczająca, by mogła się ona skutecznie wytrenować. Co więcej, niektóre tagi POS były zbyt szczegółowe i występowały tak rzadko, że przypadała na nie zbyt mała ilość przykładów.

   Z tego powodu wybraliśmy bibliotekę do tagowania słów, która korzystała z 19stu tagów POS, o nazwie **spacy**.

2. Podobnie jak podzieliliśmy dane trenujące i walidacyjne, utworzyliśmy także macierze POS im odpowiadające.

3. Następnie napisano sieć, która przyjmowała macierz POS oraz macierz wejściową i dla każdego słowa zmieniała macierze wag podczas uczenia.

   Niestety, ta wersja sieci nie była efektywna, ponieważ ucząc się tylko jednego przykładu jednocześnie, jedna epoka trwałaby około 4000h. Wykorzystanie batchy nie było możliwe - na raz bowiem sieć nie mogła przetwarzać wielu przykładów jednocześnie, ponieważ dla każdego słowa w każdym przykładzie musiała instrukcją **if** sprawdzić które wagi powinny zostać użyte do uczenia.

4. Ostatecznie problem z punktu 3. udało się brawurowo rozwiązać, co zostało szczegółowo opisane poniżej.

LSTM-POS i super-macierze

Jak wspomniano, aby przetestować skuteczność tej sieci konieczne było zgrupowanie przykładów w batch'e. Wpadliśmy na następujący pomysł:

Niech wagi macierzy będą zbiorem dwuwymiarowych wag połączonymi w jeden trójwymiarowy tensor.

Dane wejściowe będą z kolei tak zmodyfikowane, aby wektory reprezentujące słowa zostały zanurzone w kolejnym wymiarze, a ich odległość od początku brzegu tensora oznaczałyby, jaki tag mają w rozkładzie POS.

Aby zilustrować ten koncept, poniżej przedstawiono pomocniczy rysunek:

<u>Zwykłe słowa:</u>

[A1, A2, A3, A4, A5, A6, A7]

[B1, B2, B3, B4, B5, B6, B7]

<u>Słowa w super-macierzach (rozkład POS na 5 możliwych tagów)</u>

[[0 , 0 , 0 , 0 , 0 , 0 , 0]
[0 , 0 , 0 , 0 , 0 , 0 , 0]
[0 , 0 , 0 , 0 , 0 , 0 , 0]
[A1, A2, A3, A4, A5, A6, A7]
[0 , 0 , 0 , 0 , 0 , 0 , 0]]

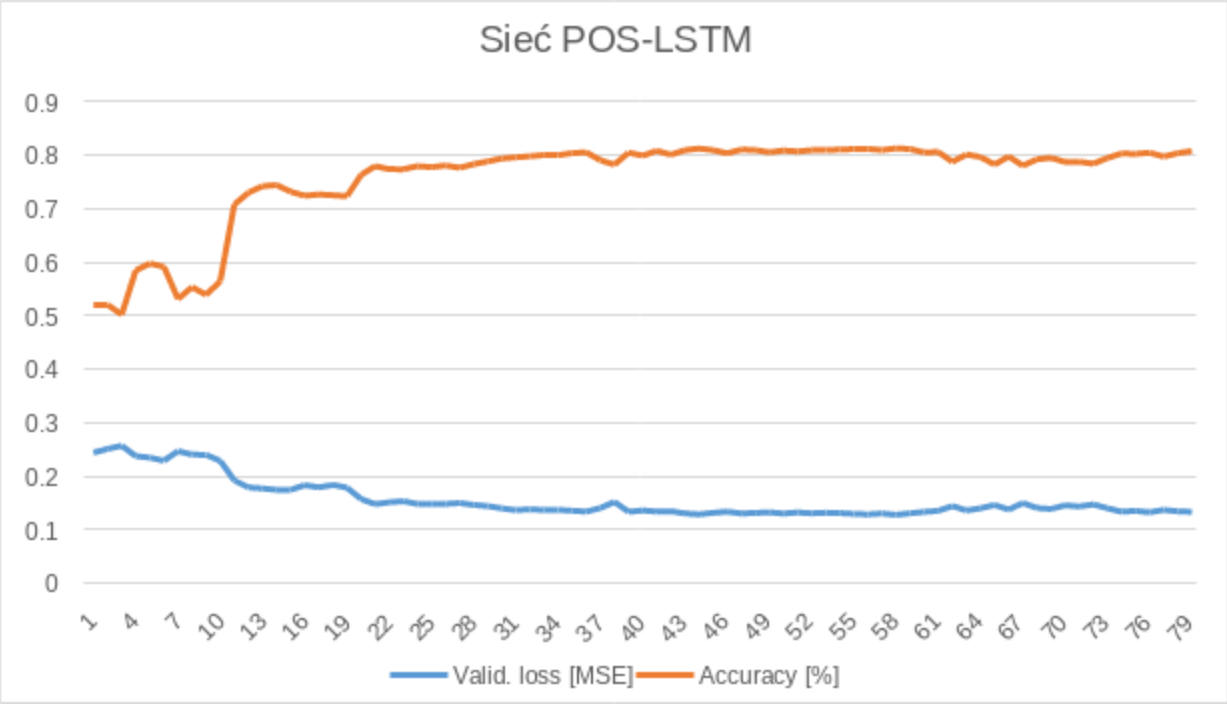
[[0 , 0 , 0 , 0 , 0 , 0 , 0]
[B1, B2, B3, B4, B5, B6, B7]
[0 , 0 , 0 , 0 , 0 , 0 , 0]
[0 , 0 , 0 , 0 , 0 , 0 , 0]
[0 , 0 , 0 , 0 , 0 , 0 , 0]]

Dzięki wykorzystaniu takiego mechanizmu, słowa byłyby mnożone przez wszystkie wagi, jednak tylko jeden “rząd” byłby niezerowy. Po takiej operacji wynik można zsumować kolumnami i otrzymać stosowny rezultat.

Trudność z powyższym rozwiązaniem polegała na tym, że skoro jedno słowo jest reprezentowane jako 2-wymiarowa macierz, to ciąg słów miał 3 wymiary, a batch słów 4 wymiary. O ile matematycznie nie jest to skomplikowane, to jednak percepcyjnie mieliśmy dużo trudności w rozwiązywaniu problemów związanych z wymiarami wag i wejścia.

W końcowej fazie udało się nam sprowadzić zagadnienie do postaci, w której w żadnym momencie nie operowaliśmy czterowymiarowym tensorem.

Niestety ponownie pojawił się problem bardzo dużych macierzy w pamięci. 16GB nie wystarczyło, aby je zaalokować w pamięci RAM, na szczęście udało się je zmieścić w pamięci SWAP, co jednak negatywnie wpłynęło na wyraźnie wolniejszy czas dostępu do tychże tensorów.



Sieć LSTM-POS osiągnęła skuteczność rzędu 80%. Jest to mniej niż skuteczność standardowej sieci LSTM którą zrealizowaliśmy w pierwszym etapie projektu.

Również błąd średniokwadratowy okazał się być dwukrotnie wyższy od standardowej sieci LSTM.

Czas uczenia: **ok. 13h**

Dyskusja wyników i wnioski

Standardowa sieć LSTM osiągnęła lepszy wynik od sieci LSTM-POS, co w pierwszym odruchu budzi zdziwienie. Przecież skoro do sieci dodaliśmy więcej macierzy wag, to w najgorszym wypadku powinny one wszystkie wytrenować się do podobnych wartości i osiągnąć wynik analogiczny do sieci standardowej LSTM.

Nasza hipoteza wyjaśniająca gorszy wynik jest następująca:

Ze względu na nierównomierną liczbę różnych tagów POS jakie zostały przypisane do słów przez sieć, słowa otagowane najrzadziej występującymi tagami okazały się liczebnościowo niewystarczające do generalizacji wiedzy przez macierze wag. Z tego względu mogło nastąpić nadmierne dopasowanie do danych trenujących, co spowodowało spadek wyniku dla danych walidacyjnych.

Być może jednym z rozwiązać byłaby próba zbalansowania danych pod kątem POS lub wykorzystanie innego mechanizmu tagowania słów.

Wybrane trudności

Poza wymienionymi wcześniej problemami, największym mankamentem praktycznej realizacji sieci przy użyciu biblioteki PyTorch były dwie rzeczy:

- **Wersja PyTorch-CPU nie działała jak należy - propagacja wsteczna trwała ok. 50 razy dłużej niż propagacja zwykła przez sieć.**
  Według informacji jakie udało nam się znaleźć w Internecie problem ten pojawiał się nie tylko u nas, jednak nie posiadał ogólnie przyjętego rozwiązania.
- **Zainstalowanie biblioteki CUDA niezbędnej do uruchomienia PyTorch lub Tensorflow przy wykorzystaniu karty graficznej na systemie Linux stwarzało co najmniej duże problemy.**
  Wymagane było m.in. zainstalowanie starej wersji kompilatora gcc, instalacja starych sterowników Nvidia, a i tak za pierwszym razem po zainstalowaniu CUDA Toolkit system Ubuntu przestał się uruchamiać.
- **Pojawiające się bez ostrzeżenia wartości NaN w wagach**.
  Jak się okazało, częściowo problem wynikał z inicjalizacji wag, częściowo z konieczności zadeklarowania, że cała sieć powinna przetwarzać macierze ze wspomaganiem GPU, a po części była wynikiem zbyt dużych wartości gradientu obliczanego podczas propagacji wstecznej.
- **Wycieki pamięci w pamięci GPU.**
  Problem ten był o tyle uciążliwy, że bardzo trudno określić, czy wyciek nastąpił, co więcej, jeszcze trudniej określić z jakiej przyczyny. Prawdopodobnie rekurencyjna sieć neuronowa nawet jeśli przestaje korzystać z wartości zaalokowanych w pamięci GPU musi wcześniej wykonać operację “**detatch**”, aby na pewno zostały one “odczepione” od grafu obliczeniowego, następnie ręcznie wywołać funkcję “**empty_cache”**.

Załączniki

- Kod sieci LSTM
- Kod sieci LSTM-POS
- Kod trenujący sieć LSTM
- Kod trenujący sieć LSTM-POS
- Kod wspierający debugowanie wycieków pamięci w GPU
- Zbiór danych testowych
- Zbiór danych walidacyjnych
- Zapis standardowego wyjścia na konsoli z trenowania sieci LSTM

- Zapis standardowego wyjścia na konsoli z trenowania sieci LSTM-POS