

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Matematyczny
specjalność: zastosowania rachunku prawdopodobieństwa i statystyki

Jakub Gierlachowski

**Wykorzystanie nowoczesnych sieci neuronowych w
zadaniach przetwarzania języka naturalnego**

Praca magisterska
napisana pod kierunkiem
dr. Pawła Rychlikowskiego

Wrocław 2016

Jakub Mateusz Gierlachowski
adres: ul. Stefana Batorego 32/4, 59-220 Legnica

PESEL: 91052002713

e-mail: jakubg@outlook.com

Wydział Matematyki i Informatyki
kierunek: matematyka, studia II stopnia, stacjonarne
numer albumu: 241868

O Ś W I A D C Z E N I E
o autorskim wykonaniu pracy dyplomowej

Niniejszym oświadczam, że złożoną do oceny pracę dyplomową zatytułowaną

*Wykorzystanie nowoczesnych sieci neuronowych w zadaniach przetwarzania języka
naturalnego*

wykonałem samodzielnie pod kierunkiem promotora

dr. Pawła Rychlikowskiego.

Oświadczam, że powyższe dane są zgodne ze stanem faktycznym i znane mi są przepisy ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (tekst jednolity: Dz. U. z 2006 r. Nr 90, poz. 637, z późniejszymi zmianami), oraz że treść pracy dyplomowej przedstawionej do obrony, zawarta na przekazanym nośniku elektronicznym, jest identyczna z jej wersją drukowaną.

Wrocław, 29 stycznia 2016 r.

Spis treści

1 Wstęp	5
1.1 Konwencje i założenia	6
1.1.1 Epoka	6
1.1.2 Baseline	6
1.1.3 Oznaczenia	6
1.2 Zbiór danych MNIST	6
1.3 Sieci neuronowe	8
1.3.1 Neuron	8
1.3.2 Funkcja sigmoid	9
1.3.3 Wielowarstwowa sieć neuronowa	10
1.3.4 Uczenie sieci neuronowych	13
1.4 Redukcja wymiaru za pomocą t-SNE	17
2 Deep belief network	19
2.1 Zastosowanie DBN	19
2.1.1 MNIST	20
2.1.2 Klasyfikacja orzeczeń sądowych	20
2.2 Opis DBN	22
2.2.1 Restricted Boltzmann machine	22
2.2.2 Rozwinięcie RBM'ów do DBN	26
2.3 Wyniki	28
3 Rekurencyjne sieci neuronowe	35
3.1 Przegląd RNN	36
3.1.1 Działanie	36

3.1.2	Zastosowanie	37
3.2	Projekt Audioscope	41
3.3	Dokładny opis działania	43
3.3.1	Nauka w dwóch wymiarach	43
3.3.2	LSTM	48
3.3.3	Słownik	50
3.4	Wyniki	51
3.4.1	Baseline	51
3.4.2	RNNLM	51
3.4.3	RNN z użyciem Keras	52
4	Opisy programów	55
4.1	DBN	55
4.2	RNN	57
4.2.1	Przetwarzanie Korpusu Rzeczypospolitej	57
4.2.2	Przetwarzanie danych testowych - Wikipedii	58
4.2.3	Dodatkowe narzędzia	59
4.2.4	Wykorzystanie biblioteki RNNLM	59
4.2.5	Wykorzystanie biblioteki Keras	60
5	Wnioski i możliwości rozwoju	63

Rozdział 1

Wstęp

Na przestrzeni ostatnich 10 lat sieci neuronowe stały się bardzo popularnym i skutecznym narzędziem do analizy danych. Kluczowym rokiem był 2006, w którym G. E. Hinton oraz R. R. Salakhutdinov opublikowali artykuł w magazynie *Science* pod tytułem *Reducing the Dimensionality of Data with Neural Networks* [5]. Był on inspiracją do dalszych badań dla wielu osób. Również dla mnie, czego efektem jest ta praca. Koncept sieci neuronowych istnieje już od lat 80 XX wieku, gdy zyskał na dużej popularności. Z czasem okazało się, że nauczenie sieci składającej się z wielu warstw nie jest zadaniem trywialnym i tylko proste sieci były wykorzystywane na większą skalę. Nie miały one jednak znaczącej przewagi nad innymi rozwiązaniami, które często były szybsze i dawały wyniki o większej lub porównywalnej jakości. Obecnie sieci neuronowe przeżywają renesans, głównie za sprawą nowych pomysłów rozwiązywających wcześniejsze problemy, a także większych możliwości obliczeniowych komputerów, czy dostępności wielkich zbiorów danych. Duże firmy zbierają ogromne ilości informacji, których często nie są w stanie przetworzyć. Niektóre z nich (np. Google) dostrzegły ich potencjał i już wykorzystują je na masową skalę, analizując każdy obrazek, e-mail czy zachowanie użytkownika. Naukowcy oraz specjalisci z przemysłu intensywnie rozwiązują zarówno sieci neuronowe, jak i inne metody analizy i wykorzystania danych, w celu rozwiązania zadanych problemów.

W niniejszej pracy wykorzystuję różne rodzaje nowoczesnych sieci neuronowych w zadaniach przetwarzania języka naturalnego. Opieram się głównie na wynikach zagranicznych naukowców, którzy stosowali je w języku angielskim. W moim przypadku

wszystkie zagadnienia będą związane z językiem polskim, który dotychczas nie był w ten sposób badany (za pomocą tych konkretnych sieci neuronowych).

W dalszej części tego rozdziału wyjaśnię jak działają proste sieci neuronowe, co będzie teoretycznym wstępem do dalszych rozważań. W kolejnych dwóch rozdziałach opiszę wyniki mojej pracy nad różnymi zagadnieniami związanymi z językiem polskim. W obu przypadkach użyję sieci neuronowych różnego rodzaju. Bardzo często będę odnosił się do zbioru danych MNIST, który również zostanie przedstawiony. Efektem tej pracy są również programy komputerowe, które opiszę w czwartym rozdziale.

1.1 Konwencje i założenia

1.1.1 Epoka

Nauka sieci neuronowej polega na przetwarzaniu danych treningowych. Dane przetwarza się wielokrotnie, aż do osiągnięcia zamierzonych rezultatów. Pojedyncze przetworzenie całego zbioru danych to jedna epoka.

1.1.2 Baseline

Gdy stosujemy różne metody uczenia maszynowego, bardzo często na końcu otrzymujemy jedną liczbę, która będzie mówić o tym, jak skuteczny jest dany algorytm. Aby mieć jakiś punkt odniesienia, przy każdym zadaniu będę podawał wynik uzyskany przez zastosowanie bardzo prostego algorytmu. Wynik ten będę nazywał baseline.

1.1.3 Oznaczenia

We wzorach matematycznych pogrubione litery oznaczają wektory, bądź macierze.

1.2 Zbiór danych MNIST

Zbiór danych MNIST [9] jest bardzo często używany w analizie danych do porównywania skuteczności różnych rozwiązań.



Rysunek 1-1: Przykładowe dane ze zbioru MNIST, rysunek pochodzi z [3]

Składa się on z 70 000 obrazów przedstawiających ręcznie pisane cyfry oraz z takiej samej wielkości zbioru etykiet jednoznacznie określających jaka cyfra przedstawiona jest na danym obrazie. W zbiorze tym wykorzystano wszystkie możliwe cyfry (od 0 do 9) tyle samo razy. Każdy obraz jest wielkości 28 na 28 pikseli w odcieniach szarości. Bardzo często, dane przekształca się tak, by obrazy zawierały tylko białe bądź czarne piksele. Jest to tzw. faza pre-processingu, która poprzedza każdą analizę danych.

Dane podzielone są na zbiór treningowy wielkości 60 000 oraz zbiór testowy wielkości 10 000.

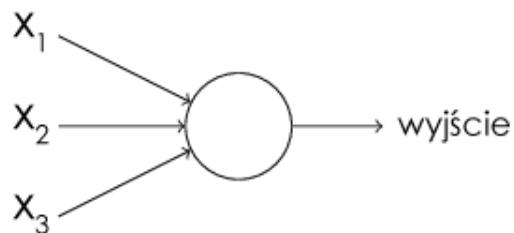
Standardowo zadanie związane z tym zbiorem danych polega na tym, by program, na podstawie dostarczonego pojedynczego obrazu (z całego zbioru testowego), potrafił samodzielnie stwierdzić jaka jest to cyfra. Zadanie jest z pozoru trywialne, lecz wymaga skomplikowanych narzędzi, aby uzyskać dużą skuteczność. Nawet dla człowieka takie pytanie może stanowić problem. Przykładowo - czy wszystkie siódemki na załączonym obrazku zostałyby przez każdą osobę oznaczone poprawnie? Co z trzecią siódemką? Można pokusić się o stwierdzenie, że "blizej" jej do dziewiątki. Z powyższego rysunku możemy również wywnioskować, że dane zostały zebrane w miescu, gdzie obowiązuje anglosaski system zapisywania cyfr. Jedynka jest pojedynczą

kreską, a siódemka nie ma środkowej poziomej kreski. W Polsce ostatnia siódemka prawdopodobnie została uznana za jedynkę.

Bardzo często sieci neuronowe potrafią odnaleźć zależności, których człowiek nie jest w stanie dostrzec. Na tym m.in. polega ich siła.

1.3 Sieci neuronowe

1.3.1 Neuron



Rysunek 1-2: Pojedynczy perceptron

Przykładem bardzo prostej sieci neuronowej jest pojedynczy neuron, nazywany również perceptronem. Posiada on dowolną liczbę wejść (na rysunku trzy) oraz jedno wyjście będące jego stanem. Perceptron może znajdować się w dwóch stanach:

- stan 0 (zero) - perceptron jest nieaktywny
- stan 1 (jeden) - perceptron jest aktywny

Aby określić stan neuronu, należy skorzystać z poniższego wzoru:

$$stan = \begin{cases} 0 & \text{gdy } \sum_i x_i w_i < t \\ 1 & \text{gdy } \sum_i x_i w_i \geq t \end{cases}$$

gdzie:

t to z góry ustalony próg

w_i to z góry ustalona waga przypisana wejściu x_i .

Im większa wartość t tym trudniej aktywować perceptron. Aby pozwolić sieci neuronowej samemu dobrać optymalną wartość tej zmiennej, wprowadzimy dodatkowy parametr b będący wyrazem wolnym. Wzór po przekształceniu wygląda tak:

$$stan = \begin{cases} 0 & \text{gdy } \sum_i x_i w_i + b < 0 \\ 1 & \text{gdy } \sum_i x_i w_i + b \geq 0 \end{cases}$$

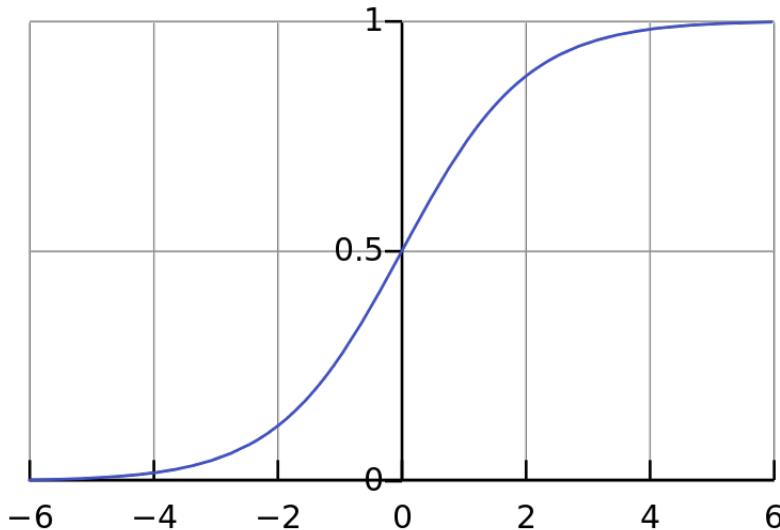
1.3.2 Funkcja sigmoid

W dalszej części pracy zobaczymy, że w rzeczywistości na początku, zarówno wagi w_i , jak i wyraz wolny b wybierane są losowo, a następnie aktualizowane w miarę uczenia sieci. Naukę możemy rozumieć przez dostarczanie nowych danych oraz cykliczne ich przetwarzanie w miarę rozwoju sieci. Sieć na wejściu dostaje dane (np. jeden obraz ze zbioru MNIST), przetwarza go i na koniec określa wynik (w tym przypadku konkretną cyfrę). Następnie porównuje go z prawidłową wartością (etykieta dostarczoną wraz z danymi). Jeżeli rezultat jest prawidłowy, to nic się nie zmienia. Jeżeli wynik był niepoprawny to sieć aktualizuje wagi oraz wyrazy wolne między wszystkimi neuronami w taki sposób, by następnym razem mieć większą szansę na odgadnięcie prawidłowej odpowiedzi. Przykładowy algorytm zostanie zaprezentowany w rozdziale 1.2.4. Zwróćmy uwagę na to, że w przypadku neuronu, który przyjmuje wspomniane dwa stany, nawet minimalna (epsilonowa) zmiana wagi może zmienić jego stan z jednego na drugi, co pociąga za sobą ogromne konsekwencje. Nie jest to pożądana przez nas własność. Dużo bardziej interesuje nas sytuacja, w której niewielka zmiana wagi będzie miała niewielki wpływ na stan neuronu. Z tego powodu powinna zostać zastosowana inna funkcja, której przykładem jest sigmoid. Wyraża się ona wzorem:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

gdzie $z = \sum_i x_i w_i + b$.

Spełnia ona podane wyżej wymagania. Zmienił się również zbiór przyjmowanych wartości. Zamiast $\{0, 1\}$ jest przedział $[0, 1]$. Oprócz wspomnianej własności, możemy teraz "przesłać" przez neuron dużo więcej informacji. W takiej sytuacji nie wiemy czy neuron jest aktywny, lecz potrafimy powiedzieć jak bardzo prawdopodobne jest to, że się aktywuje.

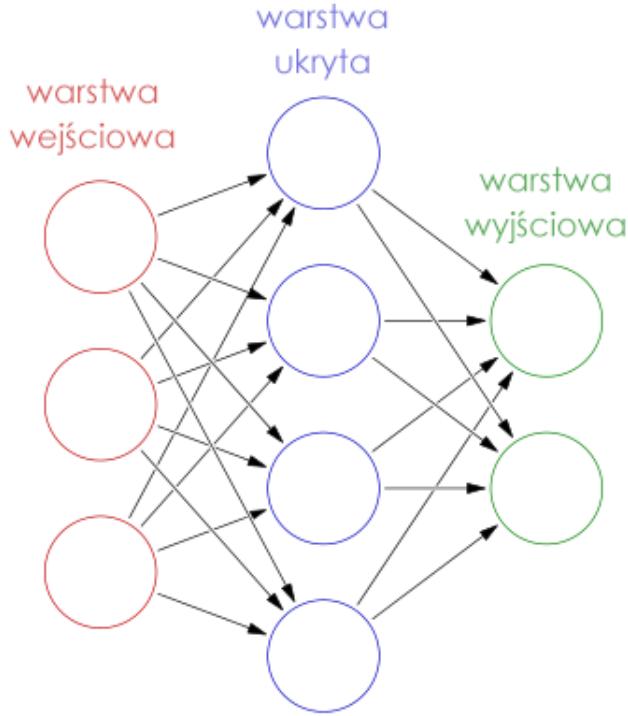


Rysunek 1-3: Przykład funkcji sigmoid

Kolejną ważną właściwością tej funkcji jest nieliniowość. Jak później zobaczymy sieć neuronowa składa się z wielu warstw złożonych z wielu neuronów. Stan każdego neuronu jest obliczany na podstawie stanów wszystkich neuronów z poprzedniej warstwy. Jeżeli zastosowana funkcja byłaby liniowa, to całą sieć moglibyśmy zredukować do jednej warstwy, ponieważ przejście pomiędzy dwoma wybranymi warstwami byłoby tylko przemnożeniem wektora reprezentującego daną warstwę przez odpowiednią macierz, co w szczególności oznacza, że istnieje taka macierz, która po przemnożeniu przez wektor wejściowy do sieci da wektor reprezentujący warstwę wejściową. Zniknęłaby wtedy cała idea sieci neuronowych, w której każdy neuron może realizować zupełnie inne zadanie. Przykładowo przy wykrywaniu twarzy ludzkiej na zdjęciu, niektóre neurony mogą bardziej skupiać się na oczach, a inne na nosie.

1.3.3 Wielowarstwowa sieć neuronowa

Wiemy już jak działa pojedynczy neuron. Przejedźmy teraz do konstrukcji sieci neuronowej złożonej z wielu neuronów. Poniżej znajduje się rysunek z przykładową siecią. Przeanalizujmy go od lewej do prawej.



Rysunek 1-4: Architektura sieci neuronowej

Pierwsza warstwa nazywana jest warstwą wejściową (input layer). Każdy neuron przyjmuje wartość zgodną z danymi, które chcemy sieci neuronowej dostarczyć. W przypadku zbioru danych MNIST, pierwsza warstwa powinna mieć 784 neurony wejściowe, ponieważ każdy obraz jest wielkości $28 * 28 = 768$. Każdy z neuronów będzie odpowiadał innemu pikselowi na obrazie. W tej sytuacji wartości przyjmowane przez neurony zależą od tego, czy piksele na obrazie są w odcieniach szarości, czy tylko w białym i czarnym kolorze. W pierwszym przypadku, neuron przyjmuje wartość na przedziale $[0, 1]$, w drugim dyskretne wartości ze zbioru $\{0, 1\}$. W obu przypadkach 0 oznacza kolor biały, a 1 kolor czarny.

Następnie mamy dowolną liczbę warstw ukrytych (hidden layers). Na rysunku mamy tylko jedną warstwę ukrytą. Wartość każdego neuronu w tej warstwie liczona jest osobno, na podstawie wszystkich neuronów z warstwy poprzedniej oraz wyrazu wolnego. Dla każdego neuronu w nowej warstwie mamy inny wyraz wolny. Niech a_i^j oznacza wartość j -tego neuronu w i -tej warstwie, $w_{i,i+1}^{j,k}$ oznacza wagę między neuronem j w warstwie i , a neuronem k w warstwie $i + 1$, a $b_{i,i+1}^k$ oznacza wyraz wolny dla k -tego neuronu w warstwie $i + 1$ od warstwy i . Wtedy:

$$a_{i+1}^j = \sigma(z_{i+1}^j) \quad (1.1)$$

$$z_{i+1}^j = \sum_{d=1}^n w_{i,i+1}^{d,j} a_i^d + b_{i,i+1}^j \quad (1.2)$$

gdzie $\sigma(x)$ to dowolna funkcja spełniająca warunki z poprzedniego podrozdziału, np. sigmoid, a n to liczba neuronów w warstwie i .

Postępując analogicznie obliczamy wartości neuronów w każdej kolejnej warstwie. W ten sposób mamy policzone również wartości neuronów w ostatniej warstwie - wyjściowej (output layer).

Przypomnijmy, że na początku sieć wczytała jeden obraz, a jej celem było podanie cyfry, która się na nim znajduje. Z tego powodu dla zbioru MNIST potencjalnie najlepszą liczbą neuronów w ostatniej warstwie jest 10. Tyle w tym przypadku jest możliwych etykiet - cyfr między 0, a 9.

Nie ma jednej zasady na określenie liczby neuronów w jakiejkolwiek warstwie poza dwoma skrajnymi. Pierwsza z reguły zadana jest przez dane, a ostatnia przez nasze oczekiwania wobec wyników, jakie ma produkować sieć. Zazwyczaj polega się na doświadczeniu i na testach. Teoretycznie w ostatniej warstwie wystarczyłyby tylko 4 neurony (dla zbioru MNIST), ponieważ w ten sposób bylibyśmy w stanie zapisać binarnie dowolną cyfrę ($2^4 = 16 > 10$). Niemniej jednak w tym przypadku 10 okazuje się lepszym rozwiązaniem, co wynika z przeprowadzonych doświadczeń. Dla uproszczenia założymy, że w ostatniej warstwie, patrząc od góry do dołu, każdy kolejny neuron odpowiada kolejnej cyfrze. Neuron, który ma największą wartość jest odpowiedzią sieci na zadane zadanie. Przykładowo, jeżeli jest to trzeci neuron to wg sieci na obrazie jest cyfra 2 (przy założeniu, że liczymy od zera).

Podsumowując, określenie struktury sieci neuronowej nie jest zadaniem trywialnym. Wielkość pierwszej warstwy można wywnioskować z postaci danych. Dla ostatniej warstwy zależy to głównie od tego, czego od danej sieci oczekujemy. W przypadku liczby oraz wielkości warstw ukrytych nie ma żadnej reguły.

1.3.4 Uczenie sieci neuronowych

W poprzednim podrozdziale sygnał (dane) został rozpropagowany do przodu sieci. Nauka sieci neuronowej polega na przesłaniu sygnału w drugą stronę (do tyłu) i aktualizowaniu wag.

Przypomnijmy, że w przypadku uczenia z nadzorem, z którym mamy tutaj do czynienia, dane zawierają również etykiety. Potrzebujemy narzędzia, które umożliwi nam znaleźć taki zbiór wag w przestrzeni wszystkich możliwych zbiorów wag, który pozwoli uzyskać sieci neuronowej jak największą skuteczność w danym zadaniu. W przypadku zbioru MNIST - będzie podawać prawidłową cyfrę w jak największej liczbie przypadków.

Wprowadźmy zatem funkcję straty (loss function). Niech x_i oznacza i -tą daną, y_i i -tą etykietę, a $h_\theta(x_i)$ oznacza etykietę sugerowaną przez sieć neuronową po przejściu x_i przez całą sieć, gdzie θ to zbiór wszystkich parametrów (wag i wyrazów wolnych) w danej sieci. Definiujmy przykładową funkcję straty jako:

$$C(\theta) = \frac{1}{2} \sum_{i=1}^n \| \mathbf{h}_\theta(x_i) - \mathbf{y}_i \|^2 \quad (1.3)$$

gdzie n oznacza liczbę danych, a wektory $\mathbf{h}_\theta(x_i)$ oraz \mathbf{y}_i mają taką samą długość jak ostatnia warstwa sieci.

Jest to kwadratowa funkcja straty. Zauważmy, że wartość funkcji straty zależy od tego jak często sieć neuronowa podaje prawidłową wartość. Im rzadziej się myli, tym mniejsza jest jej wartość, dlatego naszym celem jest minimalizacja tej funkcji.

W rzeczywistości można użyć wielu innych funkcji. Bardzo często stosowana jest entropia krzyżowa (cross entropy):

$$C(\theta) = -\frac{1}{n} \sum_{i=1}^n (\mathbf{y}_i \log(\mathbf{h}_\theta(x_i)) + (1 - \mathbf{y}_i) \log(1 - \mathbf{h}_\theta(x_i)))$$

Założymy, że mamy m parametrów $\theta_1, \theta_2, \dots, \theta_m$, które reprezentują wszystkie wagi i wyrazy wolne. Wtedy dowolnie wybrany zbiór parametrów to punkt w przestrzeni m wymiarowej. Naszym zadaniem jest odnalezienie takiego punktu, który będzie minimalizował funkcję straty. Ze względu na liczbę kombinacji wszystkich parametrów, nie jesteśmy w stanie sprawdzić każdej możliwości, dlatego musimy skorzystać z innej

metody, np. z algorytmu gradientu prostego (gradient descent). Oto jego opis:

1. Losujemy parametry początkowe (wagi i wyrazy wolne).
2. Przepuszczamy całe dane przez sieć, czyli iterujemy po całym zbiorze danych.
Mając jedną próbke, zaczynamy od warstwy wejściowej, obliczamy stany kolejnych warstw, aż do warstwy wyjściowej, z której wynik dostarczamy do funkcji straty.
3. Dla każdej obserwacji (w przypadku MNIST, cyfry), obliczamy pochodne funkcji straty po parametrach: $\frac{\partial C}{\partial \theta_j}$, gdzie θ_j to j -ty parametr. Dzięki nim wiemy, w którym kierunku funkcja maleje.
4. Aktualizujemy parametry korzystając ze wzoru: $\theta_j = \theta_j - \alpha \frac{\partial C}{\partial \theta_j}$, gdzie α to z góry ustalona stała (np. 0.1).
5. Powtarzamy punkty 2-4 aż do momentu zbiegnięcia funkcji do globalnego minimum.

W standardowej wersji tego algorytmu, wagi aktualizujemy na podstawie wszystkich danych. Wtedy nazywamy go batch gradient descent (BGD). Zwróćmy uwagę, że we wzorze na funkcję starty zawarta jest suma po wszystkich danych. Możemy również aktualizować wagi za każdym razem, gdy przepuścimy przez sieć pojedynczą obserwację - wtedy taką metodę nazываемy stochastic gradient descent (SGD). W praktyce stosuje się tę drugą wersję, aczkolwiek zamiast jednej obserwacji, przepuszczającej się nieco więcej danych, np. 10 i wtedy dopiero na podstawie tych 10 obserwacji aktualizuje parametry. Stosujemy wtedy ten sam wzór, natomiast funkcję straty obliczamy tylko z jednej (lub kilku) obserwacji.

Czasami może się zdarzyć tak, że stosując BGD, algorytm utknie w lokalnym minimum i nie będzie mógł z niego wyjść. W przypadku SGD nie stanowi to problemu, ponieważ za każdym razem idziemy w kierunku wyznaczonym przez tylko kilka lub jedną próbke, dlatego nawet jeżeli trafimy na lokalne minimum to będzie ono "dotyczyło" tylko części danych. W związku z tym, w którymś momencie z niego wyjdziemy, bo o kierunku "zdecydują" próbki, dla których nie jest to lokalne minimum. Zaletą stosowania więcej niż jednej obserwacji w SGD jest to, iż niejako pozbywamy się szumu charakterystycznego tylko dla jednej danej. Pojedyncza próbka może

pójść w całkowicie złym kierunku, natomiast kilka z nich średnio powinno pójść w prawidłowym.

Niestety SGD nie jest idealny. W momencie, gdy znajdujemy się blisko globalnego minimum to algorytm cały czas krąży wokół niego i nie może go osiągnąć, ponieważ za każdym razem o kierunku decyduje inna grupa próbek, która ma swoje globalne minimum w nieco innym miejscu. W momencie wykrycia opisanej sytuacji, możemy np. zmienić algorytm SGD na BGD. Niemniej jednak wynik będący wystarczająco blisko globalnego minimum jest na ogół satysfakcjonujący i w dalszej części pracy nie będziemy zajmować się tym problemem.

Wiemy teraz na czym polega nauka sieci neuronowej. Przejdźmy do wyprowadzenia wzorów na aktualizację parametrów. Założymy, że warstwy numerowane są od 1 do I , gdzie warstwa numer 1 przyjmuje jako wejście x_k , a wyjściem ostatniej warstwy I jest $h_\theta(x_k)$. Przekształćmy wybraną funkcję straty (1.3) do bardziej przystępnej formy:

$$\begin{aligned} C(\theta) &= \frac{1}{2} \sum_{k=1}^n \|\mathbf{h}_\theta(x_k) - \mathbf{y}_k\|^2 \\ &= \frac{1}{2} \sum_{k=1}^n \|\mathbf{a}_I - \mathbf{y}_k\|^2 \\ &= \frac{1}{2} \sum_{k=1}^n \|\sigma(\mathbf{z}_I) - \mathbf{y}_k\|^2 \\ &= \frac{1}{2} \sum_{k=1}^n \left\| \begin{bmatrix} \sigma(z_I^1) \\ \sigma(z_I^2) \\ \vdots \\ \sigma(z_I^l) \end{bmatrix} - \begin{bmatrix} y_k^1 \\ y_k^2 \\ \vdots \\ y_k^l \end{bmatrix} \right\|^2 \end{aligned}$$

Twierdzenie 1.3.1 *Wzory na pochodne funkcji straty po parametrach wynoszą:*

$$\frac{\partial C}{\partial w_{i,i+1}^{d,j}} = \frac{\partial C}{\partial z_{i+1}^j} * a_i^d \quad (1.4)$$

$$\frac{\partial C}{\partial b_{i,i+1}^j} = \frac{\partial C}{\partial z_{i+1}^j} \quad (1.5)$$

Dowód:

Korzystając z reguły łańcucha oraz wzorów 1.1, 1.2:

$$\begin{aligned}\frac{\partial C}{\partial w_{i,i+1}^{d,j}} &= \frac{\partial C}{\partial z_{i+1}^j} * \frac{\partial z_{i+1}^j}{\partial w_{i,i+1}^{d,j}} \\ &= \frac{\partial C}{\partial z_{i+1}^j} * a_i^d\end{aligned}$$

$$\begin{aligned}\frac{\partial C}{\partial b_{i,i+1}^j} &= \frac{\partial C}{\partial z_{i+1}^j} * \frac{\partial z_{i+1}^j}{\partial b_{i,i+1}^j} \\ &= \frac{\partial C}{\partial z_{i+1}^j}\end{aligned}$$

gdzie:

$$\begin{aligned}\frac{\partial C}{\partial z_i^j} &= \sum_k \frac{\partial C}{\partial z_{i+1}^k} * \frac{\partial z_{i+1}^k}{\partial z_i^j} \\ &= \sum_k \frac{\partial C}{\partial z_{i+1}^k} * w_{i,i+1}^{j,k} * \sigma'(z_i^j)\end{aligned}$$

oraz dla ostatniej warstwy:

$$\begin{aligned}\frac{\partial C}{\partial z_I^j} &= \frac{\partial C}{\partial a_I^j} * \frac{\partial a_I^j}{\partial z_I^j} \\ &= \frac{\partial C}{\partial a_I^j} * \sigma'(z_I^j)\end{aligned}$$

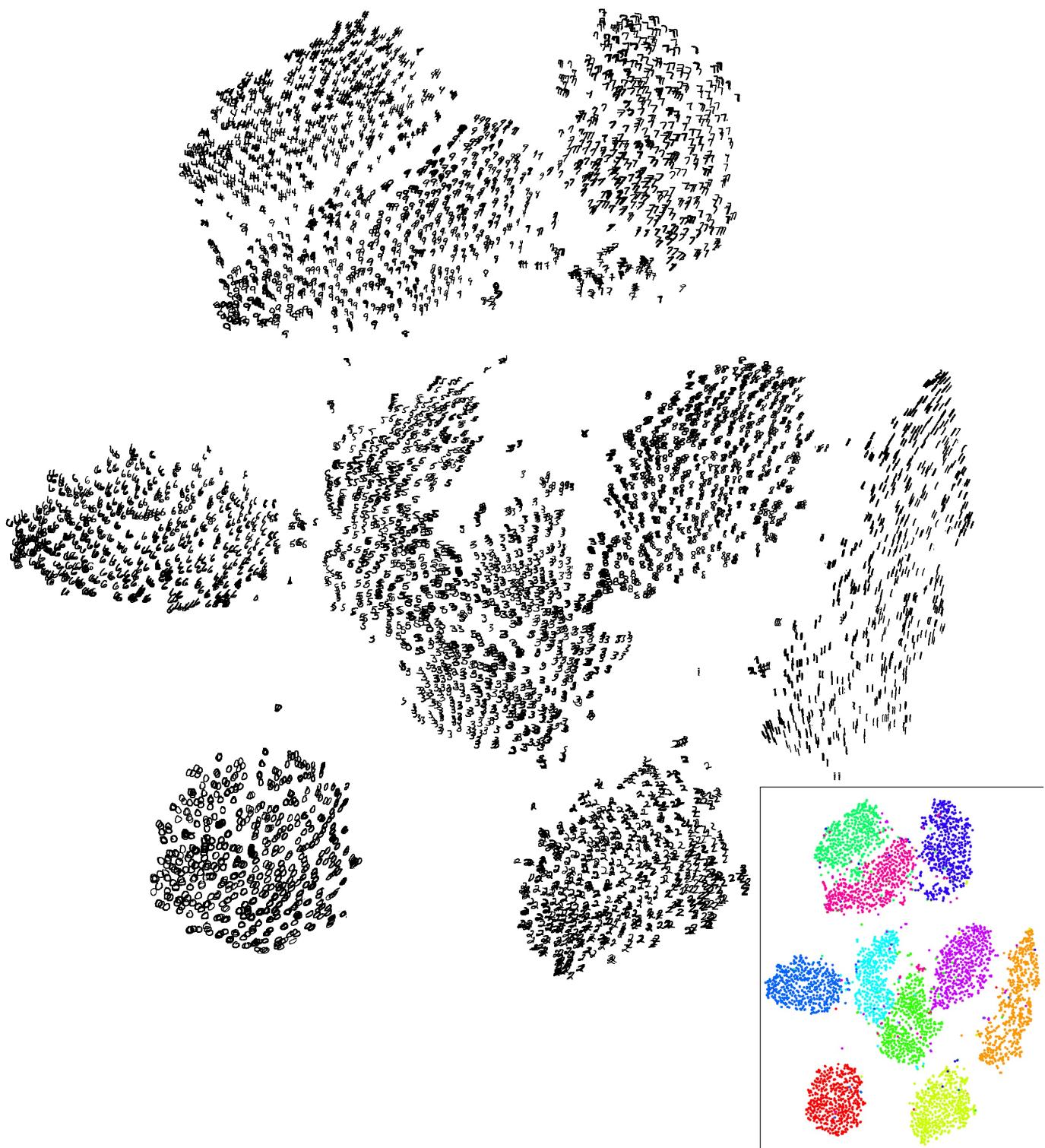
gdzie $\frac{\partial C}{\partial a_I^j}$ zależy od wyboru funkcji straty - dla wzoru 1.3 jest to:

$$\frac{\partial C}{\partial a_I^j} = a_I^j - y^j$$

a pozostałe niewiadome są znane lub łatwe do obliczenia.

1.4 Redukcja wymiaru za pomocą t-SNE

W późniejszej części pracy będę wykorzystywał t-SNE do wizualizacji danych. Jest to algorytm do nieliniowej redukcji wymiaru. Bardzo dobrze sprawdza się przy redukcji danych wysoko wymiarowych w przestrzeni 2- lub 3-wymiarową. Działanie tej metody opiera się na modelowaniu punktów w oryginalnej przestrzeni poprzez bliskie punkty z wykorzystaniem rozkładu normalnego. W nowej (niższej) przestrzeni punkty modelujemy podobnie, ale wykorzystując rozkład t-Studenta z jednym stopniem swobody. Jest to bardzo uproszczony opis, szczegóły można znaleźć w [16].



Rysunek 1-5: Redukcja zbioru MNIST za pomocą t-SNE, rysunek pochodzi z [16]

Rozdział 2

Deep belief network

Mówimy, że im więcej ukrytych warstw ma dana sieć, tym jest ona głębsza i nazywamy ją głęboką siecią neuronową (deep neural network). Przykładem jest tytułowa sieć o nazwie deep belief network (DBN). Jej architektura zakłada dużą liczbę warstw ukrytych (więcej niż jedną) oraz dodatkową fazę uczenia bez nadzoru przed właściwą nauką sieci za pomocą algorytmu SGD.

Mimo, iż większa liczba warstw ukrytych intuicyjnie powinna dawać lepsze rezultaty (większa pojemność sieci i większa zdolność do nauki) to nie możemy bez konsekwencji dodać dodatkowe warstwy i zastosować algorytm opisany w poprzednim rozdziale. Napotkamy wtedy problem znikającego gradientu, co skutkuje tym, że końcowe warstwy uczą się dość skutecznie, a początkowe praktycznie w ogóle. Rozpisując wzory 1.4 oraz 1.5 zauważamy, że im wcześniejsza warstwa, tym dłuższy jest wzór na aktualizację parametrów po rozpisaniu. Korzystając z tego, że każdy element we wzorze jest ograniczony z góry przez 1, łatwo można zauważać, że gradient w początkowych warstwach w naturalny sposób staje się coraz mniejszy. Istnieje wiele rozwiązań radzących sobie z tym problemem i DBN jest jednym z nich.

2.1 Zastosowanie DBN

Zanim przejdziemy do dokładnego opisu algorytmu, przedstawię skuteczność tego algorytmu dla danych MNIST oraz opiszę zadanie, w którym sam zastosowałem ten algorytm.

2.1.1 MNIST

Zbiór ten, przez wiele lat był testowany z użyciem wielu różnych algorytmów. Poniżej podaję kilka istotnych wyników [9]. Stosując zwykły klasyfikator liniowy, możemy uzyskać skuteczność na poziomie 88%, czyli częstotliwość błędu na poziomie 12%. Inne rozwiązanie wykorzystujące metodę PCA (redukujące dane do wektora długości 40) połączone z klasyfikatorem liniowym ma błąd rzędu 3,3%. Sieć neuronowa z jedną warstwą ukrytą ma błąd 4,7% (dla 300 neuronów w warstwie ukrytej) i 4,5% (dla 1000 neuronów). Jeszcze lepiej radzą sobie sieci z dwoma warstwami ukrytymi, uzyskując błędy 3,05% i 2,95% dla struktury sieci 784-300-100-10 oraz 784-1000-150-10 odpowiednio. Przed opublikowanie wyników przez Hintona, najlepszym wynikiem osiąganym przez sieci neuronowe był błąd rzędu 1,6%. Omawiana w tym rozdziale sieć DBN o strukturze 784-500-500-2000-10 myli się tylko w 1,2% przypadków.

2.1.2 Klasyfikacja orzeczeń sądowych

Firma Neurosoft Sp. z o. o. na zlecenie Ministerstwa Sprawiedliwości przygotowała portal do publikacji orzeczeń sądowych¹. Jedną z funkcji strony jest możliwość wyszukiwania orzeczeń za pomocą haseł tematycznych (tagów). Dotychczas były one dopasowywane ręcznie, przez osobę, która dane orzeczenia dodawała. Z wykorzystaniem istniejącej już bazy orzeczeń i dopasowanych haseł tematycznych, moim celem było ustalenie czy można zastosować DBN do automatycznego tagowania orzeczeń. Pierwszym krokiem było zamienienie danych na wektory liczbowe. Ustalony został słownik zawierający 2000 najpopularniejszych wyrazów znajdujących się we wszystkich orzeczeniach (z pominięciem tzw. stopwords, czyli krótkich wyrazów typu 'a', 'na', 'z', 'i', które w języku polskim występują bardzo często).

Ustalmy, że wyrazy w słowniku mają określony porządek, wtedy każde orzeczenie może zostać przedstawione za pomocą wektora o długości 2000, składającego się z zer i jedynek. Gdzie 1 na miejscu k oznacza, że dane orzeczenie zawiera k -te słowo ze słownika, a 0 oznacza, że tego słowa nie ma w orzeczeniu.

¹<http://orzeczenia.ms.gov.pl/>

alimenty	renta
apelacja	renta z tytułu niezdolności do pracy
bezpodstawnne wzbogacenie	składki
dobra osobiste	składki na ubezpieczenia społeczne
dowody	świadczenie przedemerytalne
emerytura	ubezpieczenie społeczne
emerytura wcześniejsza	umorzenie postępowania
kara umowna	umowa
klauzula wykonalności	ustalenie
koszty procesu	ustawa lutowa
koszty sądowe	wysokość emerytury
narkomania	wznowienie postępowania
odsetki	zabezpieczenie powództwa
odszkodowanie	zabezpieczenie roszczenia
podleganie ubezpieczeniom społecznym	zadość uczynienie
przedawnienie	zawieszenie wstrzymanie lub zmniejszanie emerytury i renty
przestępstwa przeciwko mieniu	zażalenie
przestępstwa przeciwko zdrowiu	zwolnienie od kosztów
przestępstwa przeciwko życiu	zwrot nienależnie pobranych świadczeń z ubezpieczenia
przestępstwo przeciwko bezpieczeństwu w komunikacji	zwrot pozwu

Tabela 2.1: 40 haseł tematycznych, które zostały wybrane do opisywanego zadania

Kolejnym krokiem jest ustalenie etykiet. Dane zostały dostarczone wraz z przypisanymi hasłami tematycznymi. Było ich ponad 500. Wiele tagów należało tylko do kilku orzeczeń, a także jedno orzeczenie mogło zawierać więcej niż jeden tag, dlatego zastosowano kilka niżej wymienionych reguł.

1. Jeżeli orzeczenie zawierało więcej niż jeden tag to sprawdzano, który z nich jest

najpopularniejszy wśród wszystkich orzeczeń, a resztę odrzucono.

2. Wybrano 40 najpopularniejszych tagów. Orzeczenia, które miały inne tagi zostały odrzucone. W ten sposób zmniejszono liczbę haseł tematycznych ponad 10-krotnie, zachowując przy tym ponad połowę orzeczeń.

Tak przygotowane dane są gotowe do użycia w sieci neuronowej, której pierwsza warstwa zawierała 2000 neuronów, a ostatnia² 40.

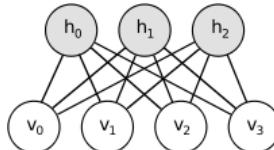
2.2 Opis DBN

2.2.1 Restricted Boltzmann machine

Podstawowym elementem, z którego składa się sieć DBN jest restricted Boltzmann machine (RBM) [6] [8]. Jest to mała dwuwarstwowa sieć neuronowa składająca się z warstwy widocznej **v** (visible) oraz ukrytej **h** (hidden), w której każdy neuron przyjmuje wartości binarne (czyli 0 lub 1).

Zakładając, że mamy zbiór danych treningowych (bez etykiet) możemy bez nadzoru nauczyć taką sieć. Warstwa **v** jest wejściem przyjmującym dane w taki sam sposób jak wcześniej opisywane sieci, natomiast warstwa **h** reprezentuje tzw. cechy danych (features), które RBM próbuje odnaleźć. Nie jest to jako taka warstwa wyjściowa, ponieważ tak czy siak nie mamy etykiet, zatem nie próbujemy ich na razie odgadnąć.

Standardowo sieć posiada prostokątną macierz wag **W** oraz wektory wyrazów wolnych. Aby przejść od warstwy **v** do **h** używamy **W** oraz **b**, natomiast w drugą stronę, **W**^T oraz **a**. Neurony połączone są ze wszystkimi neuronami w nieswojej warstwie, podobnie jak na załączonym rysunku:



Rysunek 2-1: Przykładowy RBM

²Nie jest regułą to, by ostatnia warstwa miała długość równą oczekiwanej liczbie różnych "odpowiedzi" sieci. Niemniej jednak jest to jedno z częstszych rozwiązań.

RBM jest modelem bazującym na energii, którego rozkład definiowany jest przez funkcję energii. Łaczna energia pary (\mathbf{v}, \mathbf{h}) wyraża się wzorem:

$$E(\mathbf{v}, \mathbf{h}) = - \sum_{i \in I} a_i v_i - \sum_{j \in J} b_j h_j - \sum_{i \in I, j \in J} v_i h_j w_{ij} \quad (2.1)$$

a jej prawdopodobieństwo:

$$P(\mathbf{v}, \mathbf{h}) = \frac{e^{-E(\mathbf{v}, \mathbf{h})}}{Z} \quad (2.2)$$

gdzie Z to czynnik normalizujący rozbicie, czyli suma prawdopodobieństw dla wszystkich par:

$$Z = \sum_{v \in V, h \in H} e^{-E(\mathbf{v}, \mathbf{h})} \quad (2.3)$$

Zauważmy, że im mniejsza energia tym większe prawdopodobieństwo danej pary. Naszym celem jest maksymalizacja prawdopodobieństwa dla modelowanego zjawiska. Zbiór treningowy jest tylko reprezentantem wszystkich możliwych \mathbf{v} . Dzięki wprowadzeniu dodatkowej warstwy ukrytej (\mathbf{h}) możemy modelować również nieobserwowalne dane i zwiększyć moc modelu. Wykorzystując algorytm SGD będziemy aktualizować parametry. Ze względu na to, iż tylko \mathbf{v} jest obserwowalny to interesuje nas maksymalizacja rozkładu brzegowego $P(\mathbf{v})$:

$$P(\mathbf{v}) = \frac{1}{Z} \sum_{h \in H} e^{-E(\mathbf{v}, \mathbf{h})} \quad (2.4)$$

lub równoważnie minimalizacja $-\log P(\mathbf{v})$:

$$-\log P(\mathbf{v}) = \log(Z) - \log \sum_{h \in H} e^{-E(\mathbf{v}, \mathbf{h})} \quad (2.5)$$

Niech θ będzie dowolnym parametrem sieci (wagą w_{ij} lub wyrazem wolnym a_i bądź b_j), wtedy:

$$\begin{aligned}
-\frac{\partial \log P(\mathbf{v}, \mathbf{h})}{\partial \theta} &= \frac{\frac{\partial}{\partial \theta} Z}{Z} - \frac{\frac{\partial}{\partial \theta} \sum_{h \in H} e^{-E(\mathbf{v}, \mathbf{h})}}{\sum_{h \in H} e^{-E(\mathbf{v}, \mathbf{h})}} \\
&= \frac{1}{Z} \sum_{v \in V, h \in H} \frac{\partial e^{-E(\mathbf{v}, \mathbf{h})}}{\partial \theta} - \frac{\sum_{h \in H} \frac{\partial e^{-E(\mathbf{v}, \mathbf{h})}}{\partial \theta}}{\sum_{\hat{h} \in \hat{H}} e^{-E(\mathbf{v}, \hat{\mathbf{h}})}} \\
&= -\frac{1}{Z} \sum_{v \in V, h \in H} e^{-E(\mathbf{v}, \mathbf{h})} \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta} + \frac{\sum_{h \in H} e^{-E(\mathbf{v}, \mathbf{h})} \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta}}{\sum_{\hat{h} \in \hat{H}} e^{-E(\mathbf{v}, \hat{\mathbf{h}})}} \\
&= \sum_{h \in H} \frac{e^{-E(\mathbf{v}, \mathbf{h})}}{\sum_{\hat{h} \in \hat{H}} e^{-E(\mathbf{v}, \hat{\mathbf{h}})}} \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta} - \sum_{v \in V, h \in H} \frac{1}{Z} e^{-E(\mathbf{v}, \mathbf{h})} \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta} \\
&= \sum_{h \in H} P(\mathbf{h}|\mathbf{v}) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta} - \sum_{v \in V, h \in H} P(\mathbf{v}, \mathbf{h}) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta} \\
&= E\left[\frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta}|\mathbf{v}\right] - E\left[\frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta}\right]
\end{aligned}$$

Obliczając pochodne po funkcji energii, otrzymujemy następujące reguły aktualizowania wag:

$$\Delta w_{ij} = \epsilon(E[v_i h_j | \mathbf{v}] - E[v_i h_j]) \quad (2.6)$$

$$\Delta a_i = \epsilon(E[v_i | \mathbf{v}] - E[v_i]) \quad (2.7)$$

$$\Delta b_j = \epsilon(E[h_j | \mathbf{v}] - E[h_j]) \quad (2.8)$$

gdzie ϵ to z góry ustalona stała ucząca. Ponieważ neurony w każdej warstwie są wzajemnie niezależne to:

$$P(\mathbf{h}|\mathbf{v}) = \prod_{j \in J} P(h_j|\mathbf{v})$$

$$P(\mathbf{v}|\mathbf{h}) = \prod_{i \in I} P(v_i|\mathbf{h})$$

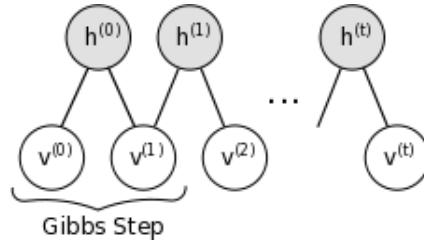
Obliczenie wartości oczekiwanej gdy mamy dane \mathbf{v} jest trywialne. Stan neuronów w warstwie ukrytej obliczamy korzystając ze wzoru:

$$P(h_j = 1 | \mathbf{v}) = \sigma\left(\sum_{i \in I} v_i w_{ij} + b_j\right)$$

Obliczenie warstwy widocznej, mając daną warstwę ukrytą jest również proste:

$$P(v_i = 1 | \mathbf{h}) = \sigma\left(\sum_{j \in J} h_j w_{ij} + a_i\right)$$

Niemniej jednak uzyskanie dowolnego v_i oraz h_j na podstawie modelu jest trudne. Jednym ze sposobów jest próbkowanie Gibbsa.



Rysunek 2-2: Próbkowanie Gibbsa

Ustalamy stan pierwszej warstwy $\mathbf{v}^{(0)}$ w sposób całkowicie losowy, a następnie korzystając z powyższych wzorów i stanu warstwy $\mathbf{v}^{(0)}$ obliczamy binarne stany neuronów w warstwie ukrytej $\mathbf{h}^{(0)}$. Jedno przejście w próbkowaniu Gibbsa polega na obliczeniu stanów w warstwach $\mathbf{v}^{(t)}$ oraz $\mathbf{h}^{(t)}$ na podstawie kolejno $\mathbf{h}^{(t-1)}$ i $\mathbf{v}^{(t)}$, tak jak na załączonym rysunku. Jest to łańcuch Markova, ponieważ każdy kolejny stan obliczamy tylko na podstawie tego, w którym się znajdujemy. Zauważmy, że $\mathbf{v}^{(i)}$ oraz $\mathbf{v}^{(j)}$ dla $i \neq j$ są tymi samymi warstwami (z tymi samymi parametrami), lecz z możliwością posiadania różnych stanów w poszczególnych neuronach. Ta sama zależność zachodzi dla warstwy ukrytej \mathbf{h} . Gdy liczba takich przejść $t \rightarrow \infty$ to ciąg par $(\mathbf{v}^{(t)}, \mathbf{h}^{(t)})$ jest zbieżny do szukanego rozkładu stacjonarnego. Zatem para $(\mathbf{v}^{(\infty)}, \mathbf{h}^{(\infty)})$ powinna pochodzić z modelowanego rozkładu.

Ze względu na to, iż w tej pracy starałem się podejść do tematu nie tylko od strony teoretycznej, ale również przeprowadzić eksperymenty, to nie mogłem skorzystać z powyższej metody, ponieważ jest ona bardzo wolna.

Alternatywą zaproponowaną przez Hintona [6] jest Contrastive Divergence (CD-k). Różni się ono dwiema rzecząmi od powyższej metody:

1. Zamiast losować stany neuronów w $\mathbf{v}^{(0)}$, ustawiamy je zgodnie z rzeczywistym stanem danych treningowych.

- Próbkowanie Gibbsa wykonujemy tylko k razy, zamiast nieskończonie wiele razy (tudzież bardzo dużo razy).

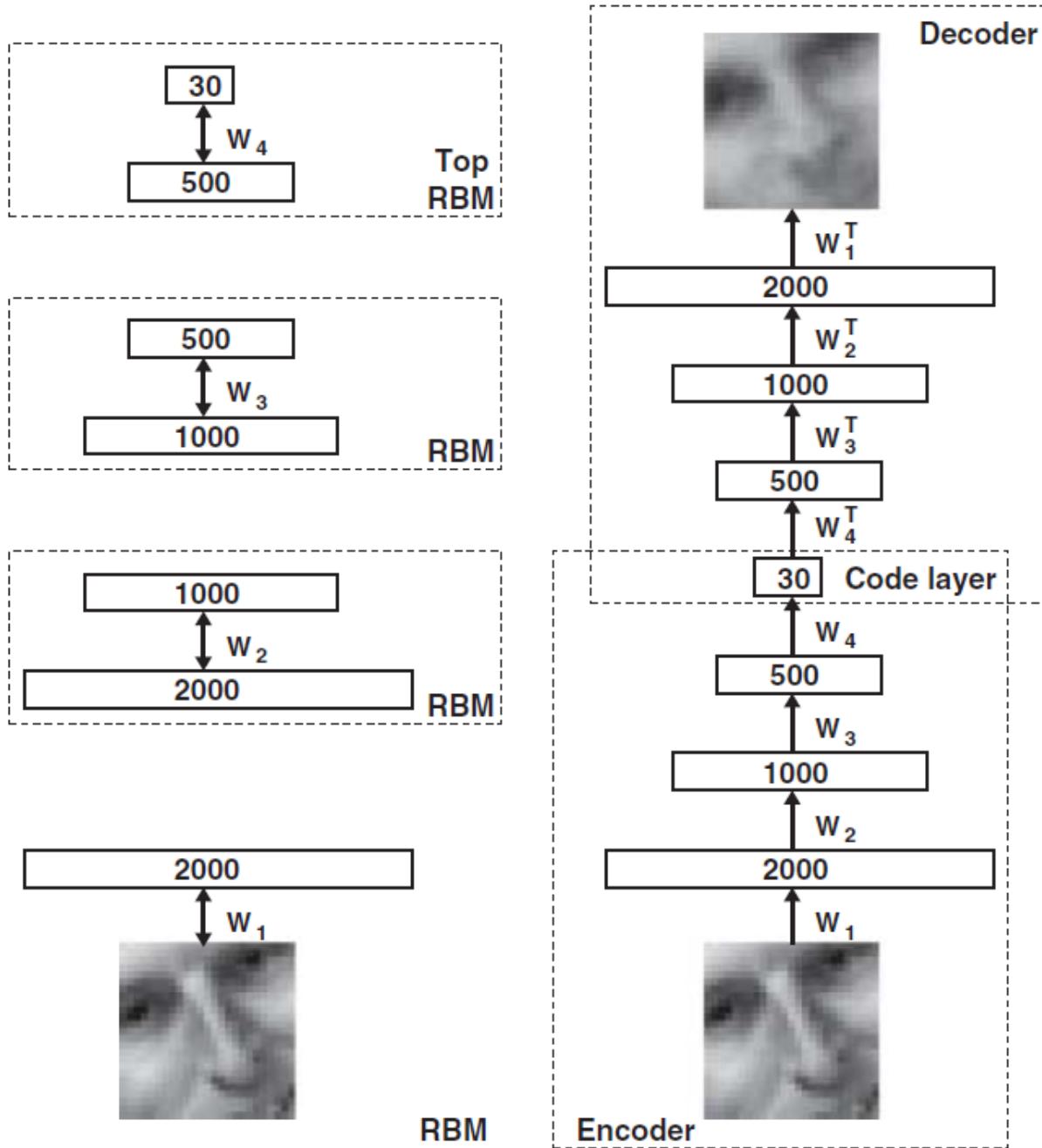
Co więcej, w praktyce okazuje się, że już dla $k = 1$ można uzyskać dobre rezultaty.

Podsumowując, RBM składa się z dwóch warstw - widocznej, która jest takiej samej długości jak dane wejściowe oraz wyjściowej, która jest ustalonego przez nas rozmiaru. Jedna epoka nauki polega na iteracji po wszystkich danych treningowych i aktualizowaniu parametrów, korzystając ze wzorów (2.6) (2.7) oraz (2.8). Możemy wykonać dowolną liczbę epok nauki. W praktyce kilkadziesiąt jest wystarczające.

2.2.2 Rozwinięcie RBM'ów do DBN

Założymy, że mamy nauczonych $k-1$ RBM'ów R_1, R_2, \dots, R_{k-1} , których warstwy widoczne są długości odpowiednio $d_1^v, d_2^v, \dots, d_{k-1}^v$, a warstwy ukryte długości $d_1^h, d_2^h, \dots, d_{k-1}^h$. Po skończonej nauce R_{k-1} przepuszczamy przez niego dane treningowe tak, by były reprezentowane przez wektory długości d_{k-1}^h i traktujemy je jako dane wejściowe do następnego RBM'a R_k . Nowy RBM R_k musi mieć warstwę widoczną długości d_{k-1}^h (czyli $d_k^v = d_{k-1}^h$) oraz warstwę ukrytą dowolnej długości. Od tego momentu nauka R_k przebiega dokładnie tak samo opisane to zostało w poprzednim podrozdziale. Ostatni RBM musi mieć taką długość warstwy ukrytej jak warstwa wyjściowa całej sieci DBN.

Gdy wszystkie RBM'y są nauczone (każdy osobno - na rysunku poniżej po lewej stronie) to możemy je rozwinać i stworzyć sieć DBN (na rysunku po prawej stronie - tylko *Encoder*). Polega to na tym, żestawiamy je jeden na drugim i scalamy warstwy ukryte z widocznymi wyższymi RBM'ów, co jest możliwe dzięki temu, że w każdym przypadku dbaliśmy o to, by długości warstw się zgadzały. Po rozwinięciu możemy przejść do właściwej nauki (fine-tuningu), która działa dokładnie tak samo jak uczenie sieci neuronowych opisane w poprzednim rozdziale, czyli za pomocą algorytmu propagacji wstecznej.



Rysunek 2-3: Rozwinięcie RBM'ów, rysunek pochodzi z [5]

Istnieje również możliwość nauki bez nadzoru. Wtedy zanim przejdziemy do nauki, musimy wykorzystać te same RBM'y. Na powstałej już sieci, stawiamy je ponownie, ale w odwrotnej kolejności, tak by stworzyły odbicie lustrzane (na rysunku po prawej stronie - połączony *Encoder* i *Decoder*). Wtedy warstwa wyjściowa jest dokładnie

takiej samej długości jak warstwa wyjściowa sieci. Znów możemy zastosować algorytm propagacji wstecznej, lecz zamiast etykiet używamy danych treningowych, czyli oczekujemy, że na wyjściu powinien znajdować się ten sam egzemplarz treningowy - jeżeli nie to odpowiednio modyfikujemy parametry. Po nauce, zabieramy górną połowę warstw i mamy gotową sieć. Uważny czytelnik zwróci uwagę na to, że opisana sieć tak na prawdę uczy się identyczności. Nie jest to jednak zadanie trywialne, ponieważ z reguły, środkowa warstwa (oznaczona na rysunku przez *Code layer*) jest istotnie mniejsza od pierwszej i ostatniej, a co za tym idzie, musimy w niej zmieścić wszystkie informacje przesyłane przez sieć.

Do zadania z orzeczeniami sądowymi użyłem naukę z nadzorem, ponieważ wykorzystuje ona więcej danych. Dzięki temu możemy spodziewać się lepszych rezultatów.

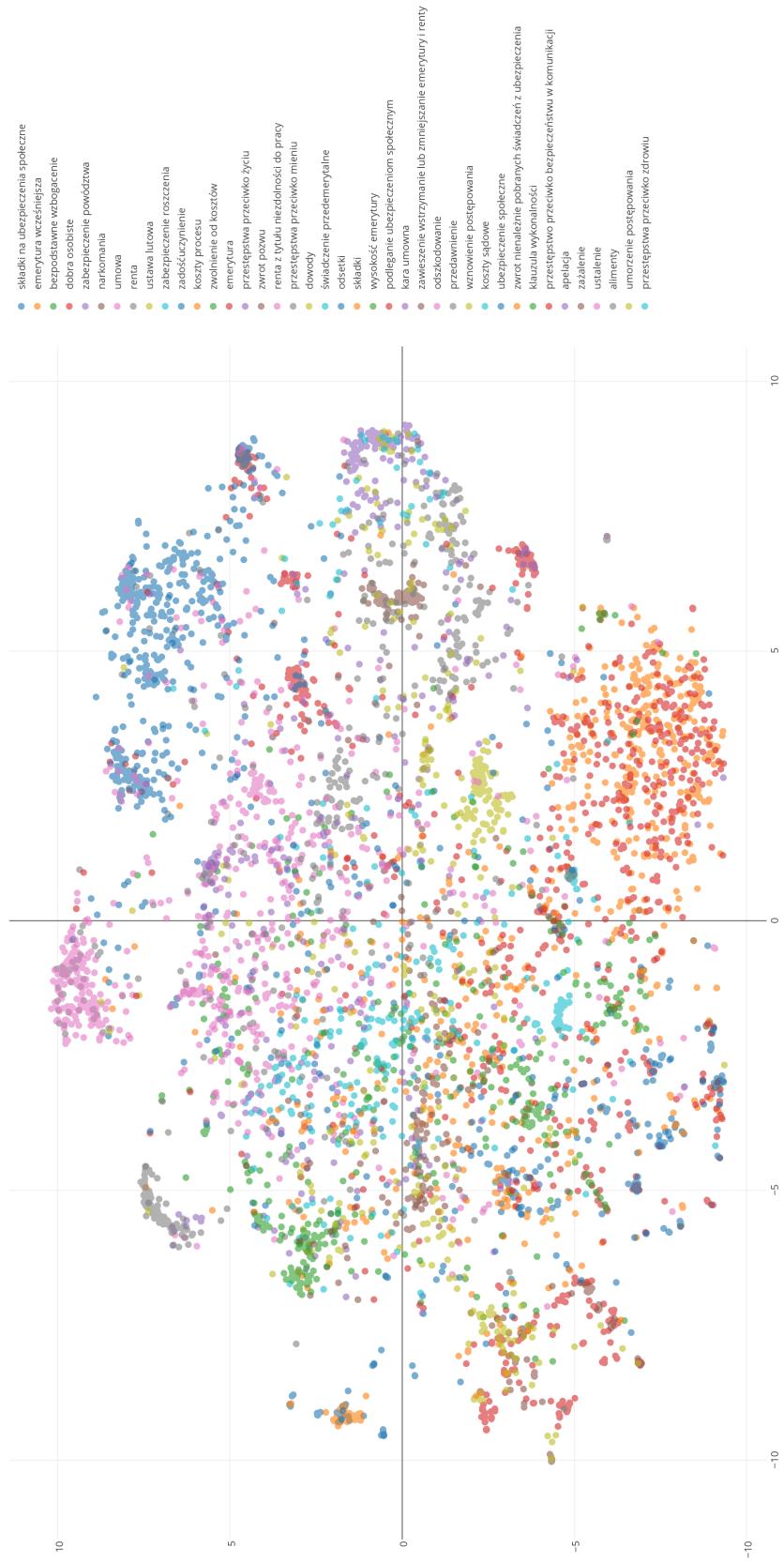
2.3 Wyniki

Najprostszym algorytmem, który można potraktować jako baseline jest wybieranie kategorii, która w danych występuje najczęściej. W tym przypadku są to **emerytury**, które dotyczą **6,17%** wszystkich orzeczeń.

Początkowym założeniem było stworzenie sieci neuronowej, która dla zadanego orzeczenia będzie potrafiła określić hasło tematyczne, które do niego najbardziej pasuje. Dane zostały podzielone na treningowe (80%) i testowe (20%). Przeprowadzonych zostało wiele eksperymentów z różnymi parametrami oraz strukturami sieci, jednakże nie udało się uzyskać wyników, które byłyby istotnie lepsze od wspomnianego wyżej i ostatecznie nie zostały one zapisane. Nie było to jednak niespodziewane, ponieważ musimy pamiętać o tym, że etykiety do danych dodawane były przez bardzo dużo ludzi i każdy określał je subiektywnie. Nie musi to oznaczać, że sieć nie potrafi poprawnie określić etykiety. Oznacza to jedynie tyle, że nie określa je w taki sam sposób jak wspomniana grupa ludzi. Nie był to też koniec eksperymentów. Uzyskane przez sieć neuronową dane z ostatniej warstwy wykorzystane zostały przeze mnie z sukcesem do wizualizacji zaprezentowanej na Rysunku 2-4.

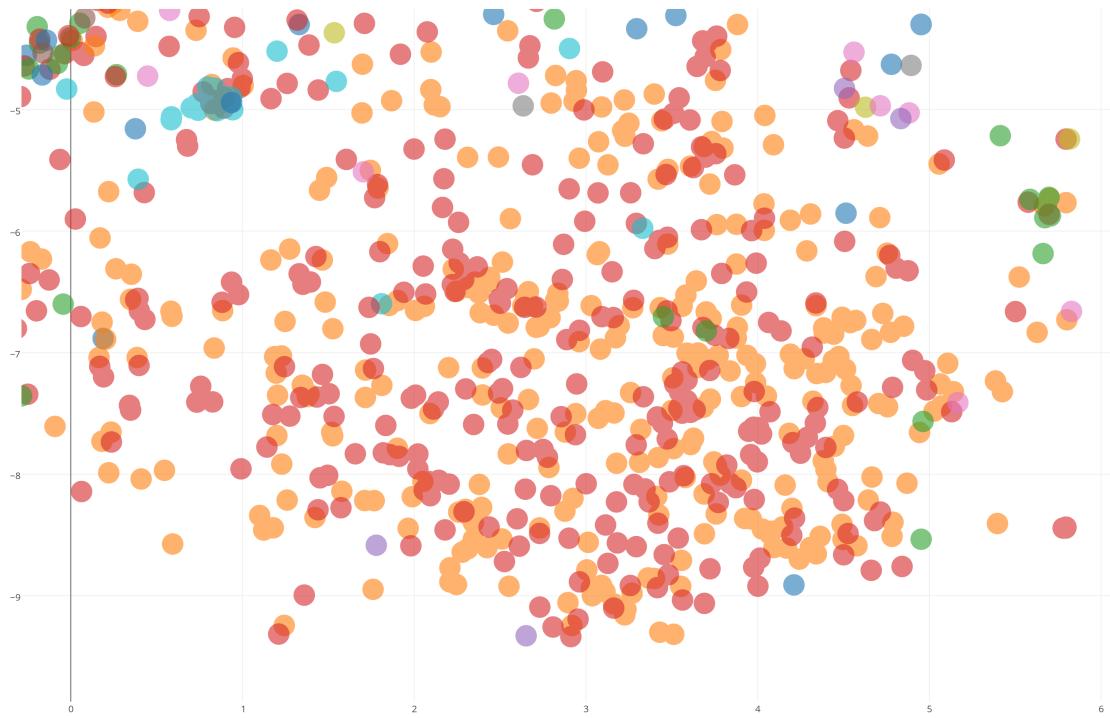
Przypomnijmy, że efektem przepuszczenia jednego orzeczenia przez sieć jest wektor o długości 40, z elementami będącymi liczbami leżącymi na przedziale [0, 1]. Mówiąc prościej, każde orzeczenie to punkt w przestrzeni 40-wymiarowej. Sieć, która

wygenerowała omawiane dane miała strukturę 2000-500-250-125-40, czyli składała się z trzech warstw ukrytych o długości kolejno 500, 250, 125 oraz warstwy wejściowej o długości 2000 i warstwy wyjściowej o długości 40. Używając t-SNE zredukowałem wszystkie orzeczenia (reprezentowane przez wektory 40 elementowe) do przestrzeni 2-wymiarowej i przedstawiłem na poniższym wykresie.



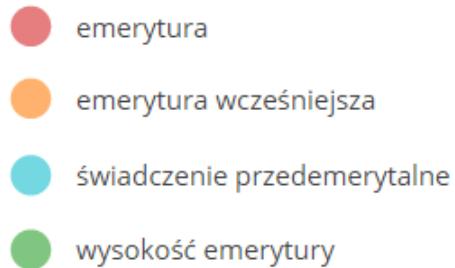
Rysunek 2-4: Orzeczenia sądowe, DBN + t-SNE

Wiele orzeczeń o tej samej tematyce (tym samym kolorze) zgrupowanych jest w tym samym miejscu, co świadczy o tym, że sieć (mimo dodatkowej redukcji wymiaru - z 40 do 2) potrafiła wyciągnąć informacje z orzeczeń. Niektóre grupy skupiają się w jednym miejscu i mieszają z innymi, często równie licznymi, grupami. Kilka takich przypadków jest dość ciekawych. Spójrzmy na jeden z nich.



Rysunek 2-5: Orzeczenia sądowe - emerytury

Po dokładnym sprawdzeniu legendy okazuje się, że prawie wszystkie punkty, mimo różnych kolorów, należą do bardzo zbliżonych kategorii. Poniżej część legendy z kilkoma z nich. Pierwsze dwie stanowią większość punktów na wykresie. Podobna sytuacja zachodzi w wielu innych miejscach na pełnym wykresie.

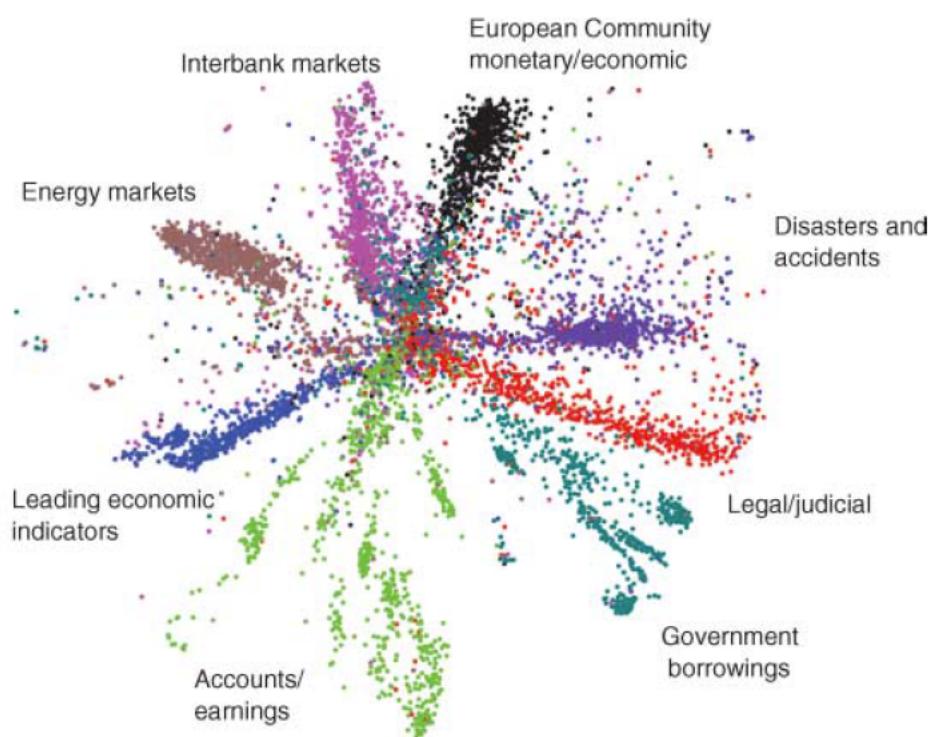


Rysunek 2-6: Legenda - orzeczenia sądowe

Mimo że klasyfikator jest chyba za słaby, by zastąpić ręczną klasyfikację, to widać, że zaprezentowana sieć DBN w połączeniu z algorytmem t-SNE może być użyteczna.

We wspomnianym na początku pierwszego rozdziału artykule [5] autorzy wykorzystali sieć DBN do wizualizacji korpusu Reuters³. Dane (artykuły) zostały przetworzone na wektory o długości 2000, analogicznie jak orzeczenia sądowe. Sieć miała strukturę 2000-500-250-125-2, dzięki czemu po przepuszczeniu danych przez sieć, były one punktami w przestrzeni 2-wymiarowej, czyli mogły być od razu zwizualizowane. Sieć była uczona bez nadzoru i po nauczeniu potrafiła generować dwuelementowe wektory, które były punktami na płaszczyźnie. Na poniższym wykresie widać wszystkie artykuły przepuszczane przez sieć i podzielone na 8 głównych kategorii, które zostały wcześniej ręcznie przypisane do każdego artykułu.

³Zbiór ponad 800 000 artykułów stworzonych przez agencję prasową Reuters



Rysunek 2-7: DBN - Reuters, rysunek pochodzi z [5]

Rozdział 3

Rekurencyjne sieci neuronowe

Powszechnym zastosowaniem rekurencyjnych sieci neuronowych (RNN) jest przetwarzanie danych sekwencyjnych. Gdy modelujemy język, sekwencją mogą być kolejne wyrazy w zdaniu, tudzież kolejne litery w wyrazie. W pewnym momencie sekwencja w ten czy inny sposób musi się pojawić. Inaczej tracimy całą zaletę RNN. Ze względu na długość wejścia i wyjścia, RNN możemy podzielić na kilka typów:

1. **1-1**, gdzie zarówno warstwa wejściowa jak i wyjściowa mają ustaloną długość (np. gdy klasyfikujemy obrazy - wejściem jest obraz o zawsze tej samej długości, a wyjściem jedna z kategorii)
2. **1-n**, gdzie wejście ma z góry ustaloną długość, a wyjście nie ma określonej długości (np. gdy klasyfikujemy obrazy, ale wyjściem jest nie jedno słowo, a całe zdanie, które w różnych przypadkach może miećną długosć)
3. **n-1**, gdzie wejście ma zmienną długość, a wyjście określona (np. przy określaniu emocji zdania)
4. **n-n**, gdzie zarówno wejście jak i wyjście nie mają określonej długości (np. tłumaczenie tekstu na inny język)

Dla uproszczenia od tego momentu RNN'y będziemy rozpatrywać w ostatnim kontekście (n-n) z niewielką zmianą, w której zakładamy, że sekwencje, które przepuszczamy przez sieć mają różną długość, ale w odpowiedzi sieć daje nam sekwencje o takiej samej długości jak ta, która była na wejściu. Dodatkowo, w przypadku tekstu,

częstym zabiegiem jest dodanie tagów oznaczających początek i koniec sekwencji (np. `_start_` oraz `_koniec_`). Dla przykładu rozpatrzmy dwie sekwencje:

oryginalna sekwencja	wejście do sieci	wyjście / etykieta
Ala ma kota	<code>_start_ Ala ma kota</code>	Ala ma kota <code>_koniec_</code>
Idzie Grześ przez wieś	<code>_start_ Idzie Grześ przez wieś</code>	Idzie Grześ przez wieś <code>_koniec_</code>

Obie mają różną długość, czyli nie możemy skorzystać z sieci typu 1-1, ale też charakteryzują się tym, że zarówno wejście, jak i wyjście z sieci ma tą samą długość (kolejno 4 i 5). Mimo, iż wejściem jest cała sekwencja to w jednym momencie czasu sieć przetwarza tylko jeden element sekwencji (reprezentowany przez wektor długości takiej jak pierwsza warstwa sieci) i w odpowiedzi daje też jeden element. Dla pierwszego przykładu rozpisane zostało to w poniższej tabeli:

wejście		wyjście/etykieta
<code>_start_</code>	→	Ala
Ala	→	ma
ma	→	kota
kota	→	<code>_koniec_</code>

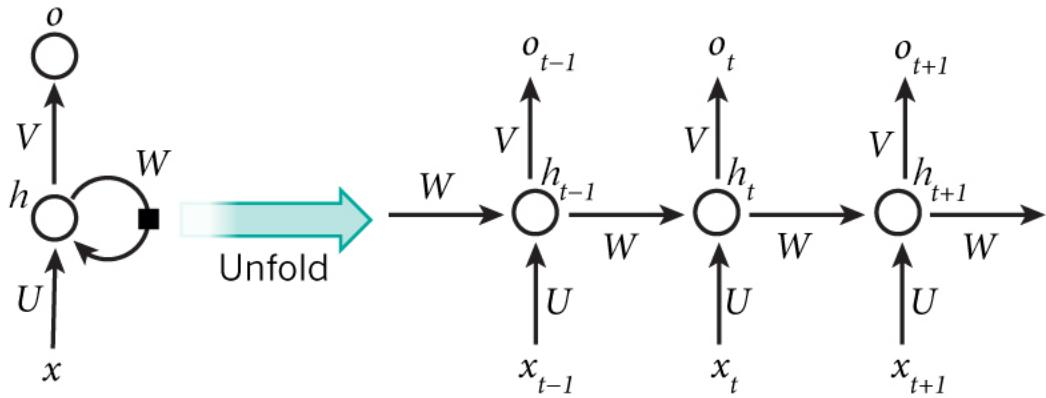
Niezależnie od tego czy jesteśmy siecią neuronową czy człowiekiem, do podania jak najlepszej odpowiedzi (przewidzenia następnego elementu) potrzebny jest kontekst czyli historia wcześniejszych wyrazów - przed wyrazem wejściowym. Dotychczas poznane sieci nie są w stanie poradzić sobie z tym zadaniem, ponieważ w momencie predykcji, widzą one tylko jeden (obecny) wyraz.

3.1 Przegląd RNN

3.1.1 Działanie

Załóżmy, że mamy bardzo prostą sieć neuronową z warstwą wejściową, jedną warstwą ukrytą i warstwą wyjściową (tak jak w pierwszym rozdziale). W dużym uproszczeniu, rekurencyjna sieć neuronowa różni się tylko tym, że wejściem do warstwy

ukrytej jest zarówno warstwa wejściowa, jak i stan warstwy ukrytej poprzedniego elementu z sekwencji. Na razie to, jak neuron w warstwie ukrytej przetwarza dane z obu wejść zostawmy z boku i potraktujmy jako czarną skrzynkę.



Rysunek 3-1: Przykładowa rekurencyjna sieć neuronowa, rysunek pochodzi z [2]

Na rysunku 3-1, po lewej stronie widzimy strukturę naszej małej sieci. Wejściem jest x , wyjściem o , a stan warstwy ukrytej to h . Zwróćmy uwagę, że zazwyczaj przez jedno kółko oznaczaliśmy jeden neuron, a w tym przypadku jest to cała warstwa. Między każdą z warstw standardowo znajdują się macierze wag (U oraz V). Dodatkowo mamy również macierz W , do przechodzenia między kolejnymi stanami warstwy ukrytej. Po prawej stronie widzimy sieć po rozwinięciu, gdzie jest to lepiej zobrazowane. Przez rozwinięcie rozumiemy zrzut sieci w kilku kolejnych (tutaj trzech) momentach czasu. Niech $t = 2$ (gdzie pierwszy moment to $t = 0$), wtedy korzystając z wcześniejszego przykładu: x_{t-1} to ***Ala***, x_t oraz o_{t-1} to ***ma***, x_{t+1} oraz o_t to ***kota***, a o_{t-1} to ***koniec***. Niewątpliwą zaletą jest to, że warstwa ukryta ma dostęp do swojego poprzedniego stanu, a co za tym idzie do wszystkich swoich poprzednich stanów w danej sekwencji. Ma ona ograniczoną pojemność, dlatego nie jest w stanie "pamiętać/wiedzieć" wszystkiego, toteż ważne jest by wspomniana czarna skrzynka była dobrze skonstruowana, tj. wiedziała co należy zapamiętać, a co nie.

3.1.2 Zastosowanie

Jedną z ciekawszych funkcji nauczzonej rekurencyjnej sieci neuronowej jest możliwość generowania sekwencji, które będą miały strukturę taką jak dane uczące. Jeżeli

będzie to zwykły tekst ze zdaniami, to sieć wygeneruje nowy tekst. Jeżeli będą to pliki HTML ze stronami WWW to sieć wygeneruje nową stronę itd. Przedstawię teraz kilka interesujących wyników opublikowanych w [7].

Poniżej znajduje się tekst wygenerowany przez sieć nauczoną na danych z Wikipedii o wielkości 100 MB:

Naturalism and decision for the majority of Arab countries' capitalide was grounded by the Irish language by [[John Clair]], [[An Imperial Japanese Revolt]], associated with Guangzham's sovereignty. His generals were the powerful ruler of the Portugal in the [[Protestant Immineners]], which could be said to be directly in Cantonese Communication, which followed a ceremony and set inspired prison, training. The emperor travelled back to [[Antioch, Perth, October 25|21]] to note, the Kingdom of Costa Rica, unsuccessful fashioned the [[Thrales]], [[Cynth's Dajoard]], known in western [[Scotland]], near Italy to the conquest of India with the conflict. Copyright was the succession of independence in the slop of Syrian influence that was a famous German movement based on a more popular servitious, non-doctrinal and sexual power post. Many governments recognize the military housing of the [[Civil Liberalization and Infantry Resolution 265 National Party in Hungary]], that is sympathetic to be to the [[Punjab Resolution]] (PJS)[<http://www.humah.yahoo.com/guardian.cfm/7754800786d17551963s89.htm>] Official economics Adjoint for the Nazism, Montgomery was swear to advance to the resources for those Socialism's rule, was starting to signing a major tripad of aid exile.]

“See also” : [[List of ethical consent processing]]
“See also” : *[[Iender dome of the ED]] *[[Anti-autism]]

“See also” : *[[Religion|Religion]] *[[French Writings]] *[[Maria]] *[[Revelation]] *[[Mount Agamul]]
“External links” : * [<http://www.biblegateway.nih.gov/entrepreneur/> Website of the World Festival. The labour of India-county defeats at the

Ripper of California Road.]

==External links== * [http://www.romanology.com/ Constitution of the Netherlands and Hispanic Competition for Bilabial and Commonwealth Industry (Republican Constitution of the Extent of the Netherlands)]

Tekst ma realnie wyglądającą strukturę, a nawet używa w poprawny sposób składni Wikipedii. W środku znajduje się kilka linków, m.in. do strony *yahoo.com*, który w całości został przez sieć wymyślony i w rzeczywistości nie istnieje. Sieć ta potrafiła również wygenerować tekst o poprawnej składni XML z wymyślonymi danymi, które mają sens:

```
<page>
  <title>Antichrist</title>
  <id>865</id>
  <revision>
    <id>15900676</id>
    <timestamp>2002-08-03T18:14:12Z</timestamp>
    <contributor>
      <username>Paris</username>
      <id>23</id>
    </contributor>
    <minor />
    <comment>Automated conversion</comment>
    <text xml:space="preserve">#REDIRECT [[Christianity]]</text>
  </revision>
</page>
```

Kolejna sieć została nauczona na podstawie kodu LaTeX'owego książki do algebry z prawie 5000 stron¹. Uruchomioną ja w trybie generowanie sekwencji i uzyskano kod LaTeX'owy, który z niewielkimi poprawkami został poprawnie skompilowany!

¹Książka dostępna jest pod adresem: <http://stacks.math.columbia.edu/>

For $\bigoplus_{n=1,\dots,m} \mathcal{L}_{m,n} = 0$, hence we can find a closed subset \mathcal{H} in \mathcal{H} and any sets \mathcal{F} on X , U is a closed immersion of S , then $U \rightarrow T$ is a separated algebraic space.

Proof. Proof of (1). It also start we get

$$S = \text{Spec}(R) = U \times_X U \times_X U$$

and the comparicoly in the fibre product covering we have to prove the lemma generated by $\coprod Z \times_U U \rightarrow V$. Consider the maps M along the set of points Sch_{fppf} and $U \rightarrow U$ is the fibre category of S in U in Section ?? and the fact that any U affine, see Morphisms, Lemma ???. Hence we obtain a scheme S and any open subset $W \subset U$ in $\text{Sh}(G)$ such that $\text{Spec}(R') \rightarrow S$ is smooth or an

$$U = \bigcup U_i \times_{S_i} U_i$$

which has a nonzero morphism we may assume that f_i is of finite presentation over S . We claim that $\mathcal{O}_{X,x}$ is a scheme where $x, x', s' \in S'$ such that $\mathcal{O}_{X,x'} \rightarrow \mathcal{O}_{X',x'}$ is separated. By Algebra, Lemma ?? we can define a map of complexes $\text{GL}_{S'}(x'/S')$ and we win. \square

To prove study we see that $\mathcal{F}|_U$ is a covering of \mathcal{X}' , and \mathcal{T}_i is an object of $\mathcal{F}_{X/S}$ for $i > 0$ and \mathcal{F}_p exists and let \mathcal{F}_i be a presheaf of \mathcal{O}_X -modules on C as a \mathcal{F} -module. In particular $\mathcal{F} = U/\mathcal{F}$ we have to show that

$$\tilde{\mathcal{M}}^\bullet = \mathcal{I}^\bullet \otimes_{\text{Spec}(k)} \mathcal{O}_{S,s} - i_X^{-1} \mathcal{F}$$

is a unique morphism of algebraic stacks. Note that

$$\text{Arrows} = (\text{Sch}/S)^{opp}_{fppf}, (\text{Sch}/S)^{opp}_{fppf}$$

and

$$V = \Gamma(S, \mathcal{O}) \hookrightarrow (U, \text{Spec}(A))$$

is an open subset of X . Thus U is affine. This is a continuous map of X is the inverse, the groupoid scheme S .

Proof. See discussion of sheaves of sets. \square

The result for prove any open covering follows from the less of Example ???. It may replace S by $X_{\text{spaces},\text{etale}}$ which gives an open subspace of X and T equal to S_{Zar} , see Descent, Lemma ???. Namely, by Lemma ?? we see that R is geometrically regular over S .

Lemma 0.1. Assume (3) and (3) by the construction in the description.

Suppose $X = \lim |X|$ (by the formal open covering X and a single map $\underline{\text{Proj}}_X(\mathcal{A}) = \text{Spec}(B)$ over U compatible with the complex

$$\text{Set}(\mathcal{A}) = \Gamma(X, \mathcal{O}_{X,\mathcal{O}_X}).$$

When in this case of to show that $\mathcal{Q} \rightarrow C_{Z/X}$ is stable under the following result in the second conditions of (1), and (3). This finishes the proof. By Definition ?? (without element is when the closed subschemes are catenary. If T is surjective we may assume that T is connected with residue fields of S . Moreover there exists a closed subspace $Z \subset X$ of X where U in X' is proper (some defining as a closed subset of the uniqueness it suffices to check the fact that the following theorem

(1) f is locally of finite type. Since $S = \text{Spec}(R)$ and $Y = \text{Spec}(R)$.

Proof. This is form all sheaves of sheaves on X . But given a scheme U and a surjective étale morphism $U \rightarrow X$. Let $U \cap U = \coprod_{i=1,\dots,n} U_i$ be the scheme X over S at the schemes $X_i \rightarrow X$ and $U = \lim_i U_i$. \square

The following lemma surjective restrecomposes of this implies that $\mathcal{F}_{x_0} = \mathcal{F}_{x_0} = \mathcal{F}_{X,\dots,0}$.

Lemma 0.2. Let X be a locally Noetherian scheme over S , $E = \mathcal{F}_{X/S}$. Set $\mathcal{I} = \mathcal{J}_1 \subset \mathcal{I}'_n$. Since $\mathcal{I}^n \subset \mathcal{I}^n$ are nonzero over $i_0 \leq p$ is a subset of $\mathcal{J}_{n,0} \circ \bar{A}_2$ works.

Lemma 0.3. In Situation ???. Hence we may assume $q' = 0$.

Proof. We will use the property we see that p is the next functor (??). On the other hand, by Lemma ?? we see that

$$D(\mathcal{O}_{X'}) = \mathcal{O}_X(D)$$

where K is an F -algebra where δ_{n+1} is a scheme over S . \square

Rysunek 3-2: Wygenerowana książka do algebry - skompilowany kod LaTeX'owy

Jednym z częstych problemów, który zauważali autorzy eksperymentu, było to, iż sieć po pewnym czasie chciała poprawnie zamknąć daną strukturę, ale nie wiedziała czy np. jest to koniec dowodu, czy może lematu.

Kolejną sieć nauczono na kodzie źródłowym jądra Linuxa (prawie 500 MB). Oto wygenerowany przez nią przykładowy kod:

```
/*
 * Increment the size file of the new incorrect UI_FILTER group information
 * of the size generatively.
 */
static int indicate_policy(void)
{
    int error;
    if (fd == MARN_EPT) {
        /*
         * The kernel blank will coeld it to userspace.
        */
    }
}
```

```

    if (ss->segment < mem_total)
        unblock_graph_and_set_blocked();
    else
        ret = 1;
    goto bail;
}

segaddr = in_SB(in.addr);
selector = seg / 16;
setup_works = true;
for (i = 0; i < blocks; i++) {
    seq = buf[i++];
    bpf = bd->bd.next + i * search;
    if (fd) {
        current = blocked;
    }
}
rw->name = "Getjbbregs";
bprm_self_clearl(&iv->version);
regs->new = blocks[(BPF_STATS << info->historidac)] | PFMR_CLOBATHINC_SECONDS << 12;
return segtable;
}

```

Kod nie kompliuje się, ale mimo to sprawia dobre wrażenie. Inicjowanie zmiennych i ich użycie prawie zawsze wydaje się losowe, natomiast sama struktura, tudzież komentarze wyglądają autentycznie.

3.2 Projekt Audioscope

Wyniki przedstawione w tym rozdziale są efektem pracy nad projektem Audioscope, który jest wspólną inicjatywą firmy Neurosoft Sp. z o. o., Uniwersytetu Wrocławskiego oraz Politechniki Wrocławskiej. W ramach projektu zostanie stworzony system do automatycznego wyszukiwania treści w nagraniach audio. Jednym z zagadnień tego projektu jest modelowanie języka polskiego. Wiele publikacji naukowych potwierdza

dużą skuteczność rekurencyjnych sieci neuronowych dla tego typu zadań, lecz w innych językach. Są to koncepcje dość nowe i wydaje się, że dotąd nie były testowane na języku polskim. W tym celu zostaną użyte dwa zbiorы danych. Pierwszy z nich, Korpus Rzeczypospolitej [15], to zbiór artykułów, które ukazały się na łamach wspomnianego czasopisma w latach 1993 - 2002. Dane zostały zarchiwizowane w formie plików HTML i udostępnione w celach naukowych. Zostały one przeze mnie oczyszczone i doprowadzone do formy, w której był sam tekst podzielony na zdania. W sumie cały zbiór zawierał 3 576 785 zdań oraz 82 733 568 słów, z czego 2 258 093 słów było unikalnych.

Drugi zbiór to dane testowe stworzone na podstawie Wikipedii². Ponizej znajduje się mała próbka:

astronomia|astronomię zajmuje się badaniem|badanym procesów|protestów|protestach|procesach dotyczących tych ciał|cię|ciału|ciała

duży|duszy|datę|dłuższe|dłuższy|dusza wpływ w tamtych czasach miał kościół|kodzie|kozła|kota|kosy|koty

w tamtych|tańszych czasach|częściach|czystych|częstych|czasów głoszono poglądy|poglądu cofające astronomię|astronomia o wiele|wieje|wielu wieków|wiekach wstecz

W każdej linii znajduje się jedno zdanie, w którym wybrane słowa mają kilka alternatyw. Zawsze pierwsze słowo jest poprawne. Zadaniem sieci jest odgadnięcie poprawnej wersji zdania³. W przypadku pierwszego zdania mamy $2 * 2 * 4 * 4 = 64$ różne wersje. Mając nauczoną sieć neuronową, możemy ją wykorzystać do tego, by każdemu z podanych zdań w danej grupie⁴ przypisać prawdopodobieństwo. Przyjmujemy, że zdanie, które otrzyma największe prawdopodobieństwo, jest uznawana przez sieć za najbardziej poprawne i porównujemy je z prawidłową wersją. W całym zbiorze testowym znajduje się 12 480 zdań (przed rozbiciem na alternatywy) oraz 45 540 alternatyw. Wszystkich kombinacji zdań jest w sumie 1 278 710. Skuteczność sieci

²Został on dostarczony przez promotora tej pracy - dra Pawła Rychlikowskiego.

³Sieć nie zna kolejności zdań i tego, że pierwsza wersja zdania jest w pełni poprawna. Dane testowe są wykorzystane dopiero po nauce sieci, zatem nie może ona na ich podstawie zaktualizować swoich parametrów.

⁴Grupa to wszystkie możliwe zdania z danej linii.

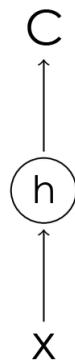
to suma wszystkich odgadniętych alternatyw, podzielona przez 45 540. Przykładowo, jeżeli w danym przypadku zdanie uznane przez sieć za najbardziej poprawne posiada 10 alternatyw, z czego 8 jest poprawnych to ta druga liczba zostaje dodana do sumy i przechodzimy do oceny kolejnego zdania.

3.3 Dokładny opis działania

3.3.1 Nauka w dwóch wymiarach

W pierwszym rozdziale zostały wprowadzone dokładne wzory na aktualizację parametrów sieci. W tym skupię się na pokazaniu różnic w stosunku do standardowej sieci neuronowej, dlatego dla uproszczenia zostaną pominięte wyrazy wolne, zamiast pojedynczych neuronów będzie mowa o całych warstwach, a na rysunkach sieć nie będzie pokazana od lewej do prawej, tylko od dołu do góry.

Założmy, że mamy zwykłą sieć neuronową z jedną warstwą ukrytą, tak jak na rysunku poniżej. Ma ona dwie macierze przejść U oraz V - każda pomiędzy poszczególnymi warstwami. Na rysunku 3-3 przez x oznaczone zostały dane dostarczone do pierwszej warstwy, przez C wartość funkcji straty po przepuszczeniu danych przez sieć oraz stan warstwy ukrytej h , który jest wektorem reprezentującym stany poszczególnych neuronów w warstwie.

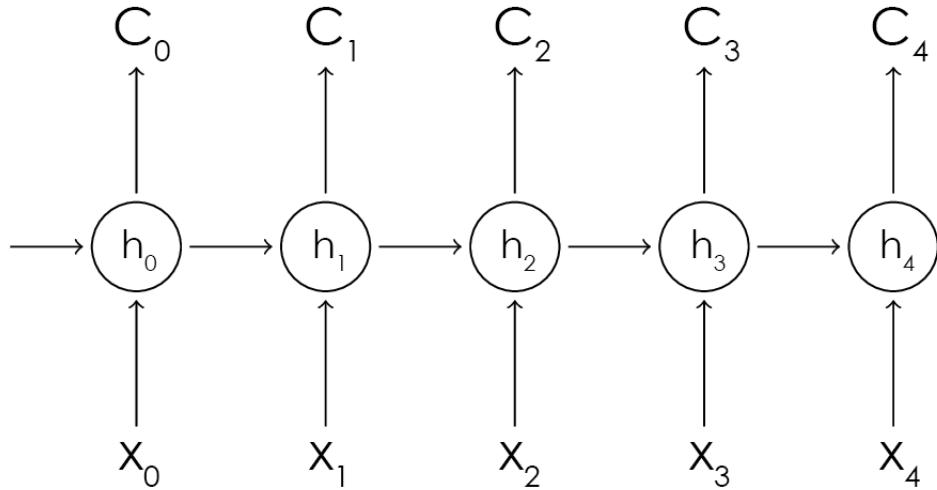


Rysunek 3-3: Sieć neuronowa z jedną warstwą ukrytą

Z rozdziału pierwszego wiemy jak przebiega nauka takiej sieci neuronowej. Przy ustalonej funkcji straty, aktualizacja dowolnego $\theta_i \in \theta$ sprowadza się do policzeniach

$\frac{\partial C}{\partial \theta_i}$, gdzie θ to zbiór wszystkich parametrów sieci.

Na rysunku 3-4 widzimy rozszerzenie tej sieci do rekurencyjnej sieci neuronowej. Wprowadzona została kolejna macierz przejść W z warstwy ukrytej w samą siebie. Po rozwinięciu dla sekwencji o długości 5, prezentuje się ona tak:



Rysunek 3-4: Rekurencyjna sieć neuronowa z jedną warstwą ukrytą

Wzory na stany poszczególnych neuronów zostają przekształcone z:

$$h_i = f_1(\mathbf{U}\mathbf{x}_i)$$

$$o_i = f_2(\mathbf{V}\mathbf{h}_i)$$

na:

$$h_i = f_3(\mathbf{U}\mathbf{x}_i, \mathbf{W}\mathbf{h}_{i-1})$$

$$o_i = f_2(\mathbf{V}\mathbf{h}_i)$$

gdzie i to i -ty element sekwencji, a o_i to stan warstwy wyjściowej i -tego elementu. Jako f_3 często używa się funkcji \tanh , dlatego od tej pory będziemy pisać:

$$s_i = \tanh(\mathbf{U}\mathbf{x}_i + \mathbf{W}\mathbf{h}_{i-1}) \quad (3.1)$$

Stan warstwy ukrytej obliczany jest zatem na podstawie zarówno warstwy wejściowej jak i swojego poprzedniego stanu. Dodatkowo, zamiast jednej wartości funkcji straty, mamy ich tyle ile wynosi długość sekwencji. Wprowadźmy zatem funkcję straty dla całej sekwencji:

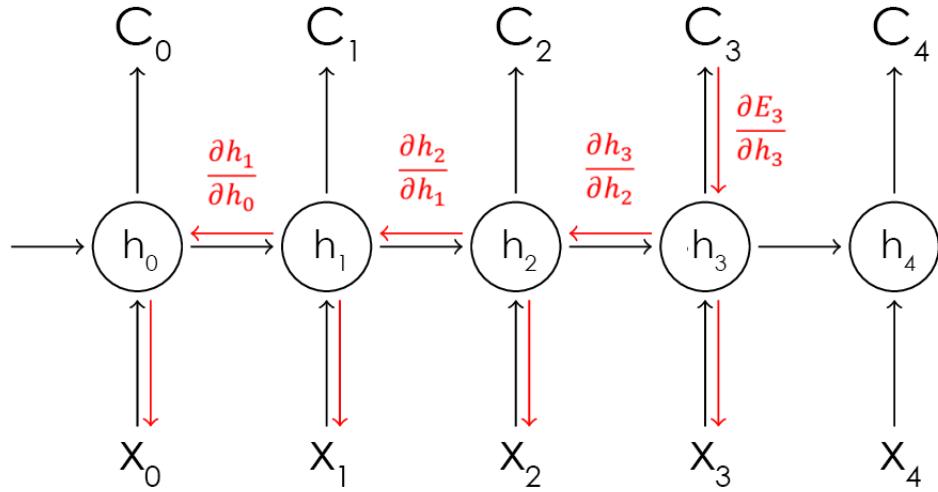
$$C(\theta) = \sum_{i=0}^k C_i(\theta)$$

co pociąga za sobą:

$$\frac{\partial C}{\partial \theta_j} = \sum_{i=0}^k \frac{\partial C_i}{\partial \theta_j} \quad (3.2)$$

gdzie k to długość sekwencji.

Zwróćmy uwagę, że w przypadku zwykłych sieci, propagacja wsteczna błędu szła tylko w jednym kierunku - w dół sieci. Na poniższym rysunku widzimy, że błąd wyliczony na podstawie np. czwartego elementu sekwencji ma również wpływ na aktualizację wag w poprzednim elementach sekwencji, co zostało zaznaczone czerwonym kolorem:



Rysunek 3-5: Propagacja błędu w RNN

Wyprowadźmy wzory na pochodne parametrów po funkcji straty C . Niech $v_a \in \mathbf{V}$ będzie dowolnym parametrem między warstwą ukrytą, a wyjściową oraz niech $u_b \in \mathbf{U}$ będzie dowolnym parametrem między warstwą wejściową, a ukrytą. Wtedy korzystając z 3.2 oraz z reguły łańcucha:

$$\frac{\partial C}{\partial v_a} = \sum_{i=0}^k \frac{\partial C_i}{\partial v_a}$$

$$\frac{\partial C}{\partial u_b} = \sum_{i=0}^k \frac{\partial C_i}{\partial u_b}$$

gdzie:

$$\begin{aligned}\frac{\partial C_i}{\partial v_a} &= \frac{\partial C_i}{\partial \mathbf{o}_i} \frac{\partial \mathbf{o}_i}{\partial v_a} \\ &= \frac{\partial C_i}{\partial \mathbf{o}_i} \frac{\partial \mathbf{o}_i}{\partial \mathbf{Vh}_i} \frac{\partial \mathbf{Vh}_i}{\partial v_a} \\ &= \frac{\partial C_i}{\partial \mathbf{o}_i} \frac{\partial \mathbf{o}_i}{\partial \mathbf{Vh}_i} h_i\end{aligned}$$

$$\begin{aligned}\frac{\partial C_i}{\partial u_b} &= \frac{\partial C_i}{\partial \mathbf{o}_i} \frac{\partial \mathbf{o}_i}{\partial u_b} \\ &= \frac{\partial C_i}{\partial \mathbf{o}_i} \frac{\partial \mathbf{o}_i}{\partial \mathbf{Ux}_i} \frac{\partial \mathbf{Ux}_i}{\partial u_b} \\ &= \frac{\partial C_i}{\partial \mathbf{o}_i} \frac{\partial \mathbf{o}_i}{\partial \mathbf{Ux}_i} x_i\end{aligned}$$

czyli bez zmian w stosunku do pierwszego rozdziału.

Niech $w_a \in \mathbf{W}$ będzie dowolnym parametrem między warstwą ukrytą, a nią samą. Wtedy (pomijając rozbieście na sumę wszystkich funkcji straty) otrzymujemy analogiczny wzór:

$$\begin{aligned}\frac{\partial C_i}{\partial w_a} &= \frac{\partial C}{\partial \mathbf{o}_i} \frac{\partial \mathbf{o}_i}{\partial w_a} \\ &= \frac{\partial C}{\partial \mathbf{o}_i} \frac{\partial \mathbf{o}_i}{\partial \mathbf{h}_i} \frac{\partial \mathbf{h}_i}{\partial w_a}\end{aligned}$$

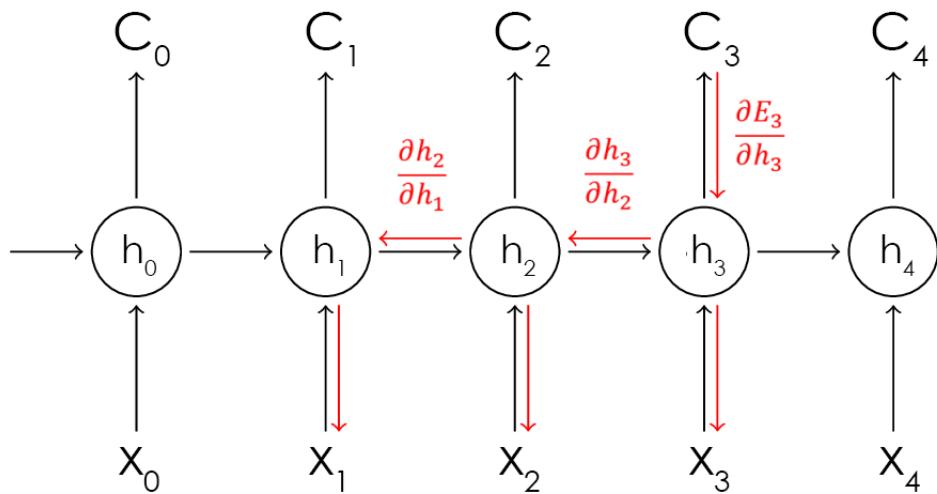
ale $\frac{\partial \mathbf{h}_i}{\partial w_a} \neq h_{i-1}$, ponieważ \mathbf{h}_{i-1} zależy od \mathbf{W} oraz \mathbf{h}_{i-2} , które również zależy od \mathbf{W}

itd. aż do \mathbf{h}_0 . Łatwo można zauważyć, że:

$$\begin{aligned}\frac{\partial \mathbf{h}_i}{\partial w_a} &= \sum_{k=1}^i \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial w_a} \\ &= \sum_{k=1}^i \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_k} h_{k-1} \\ &= \sum_{k=1}^i \prod_{l=k+1}^i \frac{\partial \mathbf{h}_l}{\partial \mathbf{h}_{l-1}} h_{k-1}\end{aligned}$$

gdzie $\frac{\partial \mathbf{h}_l}{\partial \mathbf{h}_{l-1}}$ jest łatwe do policzenia i zależy od wyboru funkcji f_3 .

Opisany powyżej algorytm nazywa się *backpropagation through time* (BPTT). Aby przyśpieszyć uczenie sieci dla długich sekwencji, często stosuje się *truncated BPTT*. Polega to na tym, że nie propagujemy błędu wstecz (po czasie) aż do początku sekwencji, tylko przez z góry określoną (jako parametr) liczbę kroków. Efekt tego parametru na propagację błędu wstecz jest przedstawiony na poniższym rysunku:



Rysunek 3-6: Truncated BPTT, liczba kroków: 2

3.3.2 LSTM

Na początku drugiego rozdziału wspomniałem o problemie znikającego gradientu. Zwróćmy uwagę na iloczyn, który wystąpił w jednym z ostatnich równań:

$$\prod_{l=k+1}^i \frac{\partial \mathbf{h}_l}{\partial \mathbf{h}_{l-1}}$$

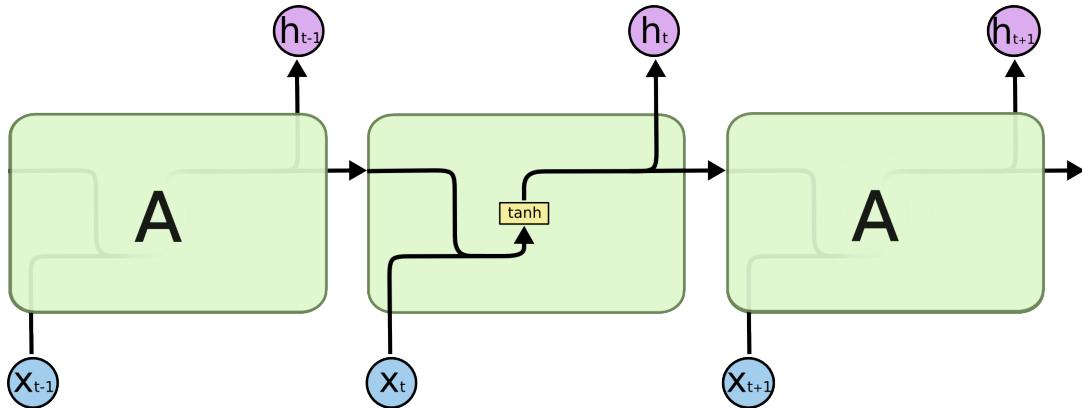
Mnożymy ze sobą $i - k$ elementów. Im $i - k$ jest większe tym większa szansa na znikający, bądź eksplodujący gradient, w zależności do tego czy

$$\left\| \frac{\partial \mathbf{h}_l}{\partial \mathbf{h}_{l-1}} \right\| < 1$$

czy

$$\left\| \frac{\partial \mathbf{h}_l}{\partial \mathbf{h}_{l-1}} \right\| > 1$$

Eksplodujący gradient nie jest dużym problemem, ponieważ wystarczy ograniczyć go przez z góry ustaloną granicę i przy aktualizacji wag nie będziemy jej przekraczać. Jeżeli natomiast gradient zniknie to w danej iteracji wagi nie zostaną zaktualizowane, a co za tym idzie, sieć nie będzie się uczyć. W tym przypadku im dalej element (np. słowo) występuje w sekwencji tym ma on większe szanse na to by nie mieć wpływu na aktualizację wag.

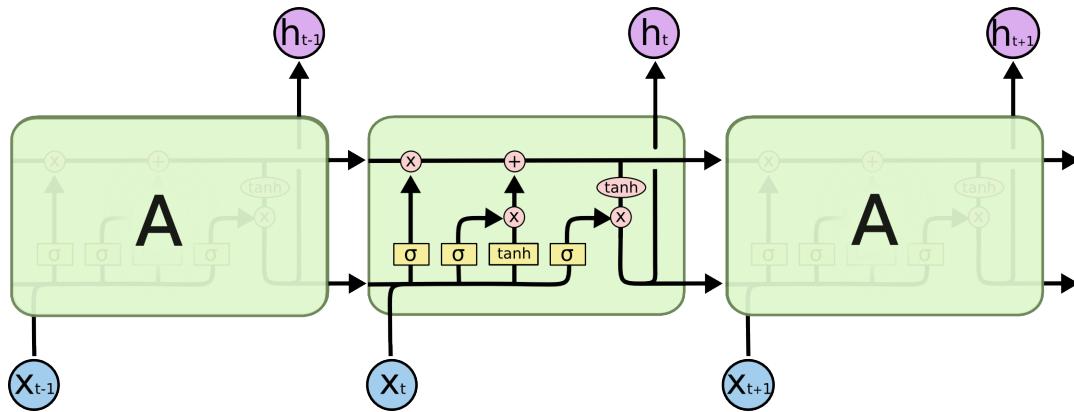


Rysunek 3-7: Sieć neuronowa ze zwykłą warstwą ukrytą, rysunek pochodzi z [14]

Jednym z rozwiązań jest zastosowanie rekurencyjnej sieci neuronowej typu Long Short Term Memory (LSTM). Została ona zaproponowana w 1994 przez Bengio [1].

Rysunek 3-7 przedstawia dokładnie taką samą sieć jaką została opisana wcześniej w tym rozdziale. Mamy dane wejściowe, stan warstwy ukrytej z poprzedniego elementu sekwencji i wykorzystując funkcję $tanh$ obliczamy nowy stan warstwy ukrytej i przekazujemy go w górę sieci oraz do warstwy ukrytej kolejnego elementu sekwencji. Mamy tutaj tylko jedną warstwę (jeden żółty prostokąt).

Na kolejnym rysunku przedstawiony jest przykładowy LSTM⁵, który składa się z 4 warstw:



Rysunek 3-8: Sieć neuronowa z LSTM, rysunek pochodzi z [14]

Dokładny opis działania czytelnik może znaleźć w [14], poniżej przedstawię najważniejsze informacje, które przybliżą działanie tej sieci:

- Głównym celem sieci LSTM jest zdecydowanie jak dużo informacji z przeszłości należy zachować oraz ile informacji z teraźniejszości przyjąć.
- Wszystkie warstwy (oznaczone żółtym kolorem) do obliczenia swoich stanów wykorzystują macierze przejść (każda warstwa ma swoją) oraz dane, które przesyłane są do nich przez czarne połączenia.
- Do następnej warstwy ukrytej przesyłamy nie jedną, a dwie informacje. Każda z nich uzyskana jest w nieco inny sposób. Zwróćmy uwagę, że "górną" informację może bardzo łatwo przejść dalej - wystarczy, że przejdzie przez pierwszą bramkę.

⁵Istnieje wiele różnych kombinacji sieci typu LSTM, które różnią się niewielkimi szczegółami - np. rozmieszczeniem bramek czy połączeń. Ostateczny efekt jest taki sam - sieć potrafi skutecznie uczyć się długich zależności.

- Czerwonymi kołami z symbolem **X** zaznaczone zostały bramki, które decydują o tym, ile informacji ma przejść dalej. Bramki te zawsze poprzedzone są funkcją sigmoid σ , która przyjmuje wartości na przedziale $[0, 1]$. Informacja, która jest przesyłana przez bramkę, przemnażana jest przez wynik funkcji σ . Jeśli było to 1, to cała informacja przechodzi dalej, jeśli 0, to nic nie przechodzi.
- Pozostałe czerwone figury to zwykłe operacje, które są na nich przedstawione (dodawanie, bądź nałożenie funkcji $tanh$).

3.3.3 Słownik

Jednym z większych problemów przy uczeniu rekurencyjnych sieci neuronowych jest wielkość słownika. Im jest on większy, tym potencjalnie lepsza jakość sieci neuronowej, ale też dłuższy jest wektor reprezentujący pojedyncze słowo. W przypadku danych testowych, pełny słownik zawierał prawie 30 tysięcy wyrazów. Aby przyśpieszyć szybkość uczenia sieci, autorzy wielu prac używali różnych sposobów na ograniczenie słownika. Najprostszym rozwiązaniem jest ustalenie wielkości słownika (tak jak w poprzednim rozdziale o DBN) i wybraniu najczęściej używanych wyrazów. Innym sposobem jest użycie klas. Przykładowo, niech k będzie z góry ustaloną liczbą klas (np. $k = 200$). Każde słowo zostaje przypisane do jednej z klas, następnie obliczany jest rozkład prawdopodobieństwa należenia danego słowa do danej klasy, a na końcu obliczany jest rozkład prawdopodobieństwa między wyrazami wewnątrz każdej z klas. Odpowiedni podział słów na klasy jest kluczem do utworzenia skutecznej rekurencyjnej sieci neuronowej. Podejście podobne do opisanego powyżej zostało użyte w wykorzystanej przeze mnie bibliotece RNNLM [11]. Ten sam autor w kolejnych latach rozwinał popularne narzędzie o nazwie **word2vec**⁶, które jest kolejnym rozwinięciem idei klas i opisał je w [10].

⁶Dostępne jest pod adresem: <https://code.google.com/p/word2vec/>

3.4 Wyniki

3.4.1 Baseline

Przypomnijmy, że do testów posłużą nam dane z Wikipedii, w których chcemy znaleźć poprawną wersję zdania. W każdym zdaniu znajduje się co najmniej jedno miejsce, w którym jest alternatywa z dwoma lub więcej wyrazami. Założymy, że rozkład przy każdej alternatywie jest jednostajny (dyskretny), wtedy każde słowo ma takie samo prawdopodobieństwo, a prawdopodobieństwo zgadnięcia wszystkich poprawnych alternatyw wynosi **36,21%**. Wynik ten posłuży jako baseline do dalszych badań.

3.4.2 RNNLM

W 2011 roku Tomáš Mikolov opublikował pracę o tytule Recurrent Neural Network Language Modeling (RNNLM) [12] wraz z biblioteką do uczenia rekurencyjnych sieci neuronowych. Wykorzystałem je do nauki własnej sieci na danych z korpusu Rzecznopolskiej. Przeprowadziłem wiele eksperymentów, zmieniając takie parametry jak wielkość warstwy ukrytej, liczba klas czy długość BPTT. Najlepszy uzyskany wynik to **86,21%** na sieć o parametrach:

1. Wielkość słownika: **28 252** (jest to jednocześnie wielkość warstwy wejściowej i wyjściowej)
2. Truncated BPTT: **7** (czyli liczba kroków wstecz przy BPTT)
3. Wielkość warstwy ukrytej: **250**
4. Liczba klas: **400**
5. Liczba epok: **13**

Nauka całej sieci trwała 4 dni, zaś użycie jej na wszystkich danych testowych 17 minut, co przekłada się na zdolność sieci do przetwarzania nieco ponad 1000 zdań na sekundę.

Aby ograniczyć liczbę słów w korpusie, wygenerowałem słownik na podstawie danych testowych, poprzez zebranie wszystkich występujących tam słów. Mając taki

słownik, zarówno dane treningowe, jak i testowe były przetwarzane w następujący sposób: jeżeli dane słowo nie znajdowało się w słowniku to zamieniałem je na $<\text{unk}>$, jeżeli znajdowało się w słowniku to pozostawiałem je bez zmian.

Wadą powyższego narzędzia jest brak mechanizmu radzącego sobie ze znikającym gradientem przy BPTT, który posiada np. LSTM. Zamiast tego jest zwykła warstwa, którą opisałem przed wprowadzeniem LSTM, używająca funkcji sigmoid do aktywacji.

3.4.3 RNN z użyciem Keras

Innym pomysłem na ograniczenie słownika jest zamienienie każdego słowa na jego kilkuznakowy sufiks (np. o długości 3). W ten sposób znacząco ograniczamy słownik, a zostawiamy przy tym kontekst, który tylko nieznacznie traci na wartości. Zwróćmy uwagę, że dane testowe w alternatywach zawierają słowa które są bardzo podobne, a często ich jedyną różnicą jest sufiks.

Z tym pomysłem przeszedłem do dalszych badań. Zamiast korzystać z poprzedniego narzędzia postanowiłem wykorzystać bardziej rozbudowaną bibliotekę o nazwie Keras⁷. Jej największą zaletą jest możliwość wykorzystania kart graficznych do obliczeń równoległych. W tym celu wykorzystałem komputery dostępne w Instytucie Informatyki Uniwersytetu Wrocławskiego, posiadające następującą konfigurację:

- CPU: **Intel Core i7-4930K** (6 rdzeni, 12 wątków logicznych)
- GPU: **NVIDIA GeForce 780 GTX** (2 304 rdzeni CUDA oraz 3 GB pamięci)
- Pamięć RAM: **32 GB**

W bibliotece tej nie ma możliwości skorzystania z truncated BPTT, co powoduje, że sieć propaguje błąd aż do początku sekwencji. Na samą naukę ma to pozytywny wpływ, natomiast znacząco wydłuża jej czas i zwiększa zapotrzebowanie na pamięć. Aby temu zaradzić ograniczyłem długość pojedynczej sekwencji do 30 elementów, ponieważ dokładnie tyle miało najdłuższe zdanie w danych testowych. W przypadku danych treningowych były one dłuższe, dlatego zostały przeze mnie pocięte wraz z

⁷<http://Keras.io/> - biblioteka do sieci neuronowych, która pozwala budować sieci różnego typu. Wykorzystuje ona 2 biblioteki w swojej implementacji - Theano oraz TensorFlow. Ta druga została udostępniona przez Google w drugiej połowie 2015 roku

dodanymi wcześniej tagami początku i końca sekwencji. Aby w jak najlepszym stopniu zachować zaletę posiadania sekwencyjnych danych, przy ucinaniu kopowałem dwa ostatnie elementy i dodawałem je na początek odciętej części.

Wszystkie opisywane eksperymenty wykorzystujące tą bibliotekę nie wykorzystywały klas, które są częścią RNNLM, a nie są zaimplementowane w Keras, dlatego zapotrzebowanie na pamięć było dużo większe, niż w przypadku RNNLM. Z tego powodu od razu zastosowałem nowy pomysł z zamieniam wyrazów na sufiksy i istotnie zmniejszyłem słownik. Liczba różnych sufiksów, które znajdowały się w danych testowych to **2364**, czyli relatywnie mało. Nauka z wykorzystaniem 100% danych treningowych w żadnym przypadku nie mieściła się w pamięci RAM, dlatego dane były porcjowane (batchowane) i wczytywane na bieżąco. Wykorzystanie pamięci rosło nie tylko wraz ze wzrostem danych, ale również w momencie gdy badałem sieci o większej strukturze (np. ze względu na wielkość warstwy wejściowej czy LSTM). Wtedy problemem była zbyt mała ilość pamięci na GPU, dlatego nie zawsze możliwe było skorzystanie z wszystkich 2 304 rdzeni na raz. W najgorszym przypadku uruchamianych było 840 rdzeni. W poniższej tabeli przedstawiam uzyskane wyniki:

Dane	Słownik	Sufiks	Struktura sieci	Liczba epok	Wynik	Bd
100%	2000	3	2000-300-2000	18	29,58%	32,12%
10%	2000	3	2000-300-2000	100	29,67%	32,08%
10%	2364	3	2364-600-300-2364	300	34,33%	26,47%
10%	2364	4	2364-600-300-2364	247	36,14%	30,06%
100%	2364	4	2364-600-300-2364	28	36,21%	30,03%

Kolumna **Wynik** reprezentuje ograniczenie dolne skuteczności sieci. W wielu przypadkach, a dokładnie w tylu ile jest podane w kolumnie **Bd** (brak danych), najlepszy wynik (prawdopodobieństwo) osiągało więcej niż jedno zdanie. W takim przypadku zakładane było, że sieć nie odgadła poprawnie żadnej alternatywy. Ograniczeniem górnym dla najlepszej skuteczności sieci jest suma obu kolumn.

Czas trwania pojedynczej epoki na pełnych danych treningowych to około 60 min. W przypadku testowania, podobnie jak RNNLM, sieć potrafiła przetworzyć ponad 1 000 zdań na sekundę.

W niektórych przypadkach były wykorzystane całe dane treningowe, a w niektórych

rych tylko 10% (aby przyśpieszyć obliczenia). Słownik zawsze zawierał frazę *<unk>* do zastąpienia słów, których nie było w słowniku oraz *<start>* i *<end>* do oznaczenia początku i końca sekwencji. Warstwa wejściowa i wyjściowa miała wielkość taką jak użyty słownik.

Najlepsza sieć używała całych danych treningowych (100%), korzystała ze słownika posiadającego 2364 słowa, każdy wyraz był zamieniany na jego 4 znakowy sufiks, posiadała dwie warstwy ukryte o wielkości kolejno 600 i 300 oraz przetworzyła cały zbiór danych 28 razy. Jej wynik nie przekracza 36,21% (66,24% bez mocnych założeń podanych powyżej), a więc jest gorszy od poprzedniego podejścia, mimo iż wykorzystano tutaj potencjalnie lepsze sieci. Wydaje się, że fenomen ten wymaga dokładniejszego zbadanie, na które niestety nie ma miejsca w niniejszej pracy.

Rozdział 4

Opisy programów

W celu zrealizowania założeń niniejszej pracy konieczne było poznanie języka Python i stworzenie z jego wykorzystaniem wielu programów do przetwarzania danych oraz uczenia sieci neuronowych. Napisano również kilka skryptów bashowych, aby ułatwić korzystanie z niektórych aplikacji.

Zarówno praca magisterska jak i dołączone programy podzielono na dwie części: DBN i RNN. W pierwszym przypadku do uruchomienia potrzebny jest tylko Python wraz z powszechnie używanymi bibliotekami takimi jak *Numpy* (zawartymi w dysstrybucji Anaconda¹). Do drugiej części (RNN) potrzebne są również biblioteki *Keras* oraz *Theano*. W celu poprawy szybkości działania programów zalecane jest wykorzystanie kart graficznych Nvidia wraz z zainstalowanymi odpowiednimi sterownikami do obsługi CUDA.

4.1 DBN

Część DBN posiada dwie główne możliwości:

- uczenia sieci neuronowych typu DBN
- redukcji danych (np. wygenerowanych przez sieć) do przestrzeni 2-wymiarowej za pomocą algorytmu t-SNE².

¹<https://www.continuum.io/>

²Bez zastosowania sieci, dane będą wysokowymiarowe - wtedy t-SNE skorzysta z PCA do redukcji do przestrzeni np. 50-wymiarowej i dopiero zastosuje swój główny algorytm.

Dane można później zwizualizować za pomocą dowolnie wybranego narzędzia/języka³. Mimo, iż w porównaniu z sieciami rekurencyjnymi, jest tutaj mniej programów to należy pamiętać, że użyte dane były w dostarczone w bardziej przystepnym formacie i nie wymagały dużego przetwarzania. Sama sieć została zaimplementowana od początku do końca bez użycia żadnej biblioteki, korzystając z rozwiązań opisywanych w [13]. W tym celu zgłębiono tematykę sieci neuronowych, dokonano przeglądu różnych implementacji i ostatecznie stworzono własną uproszczoną wersję sieci i poddano ją testom. Kolejnym krokiem było zaprojektowanie sieci DBN, którą wykorzystano do danych związanych z orzeczeniami. W poniżej tabeli przedstawiono opis poszczególnych plików i folderów:

nazwa pliku / folderu	opis
data_dbn/	Tutaj zapisywane są dane wygenerowane przez trenowanie z użyciem run_network.py.
data_original/	Tutaj powinny znajdować się dane, które używamy do niżej opisanych programów.
data_output/	Tutaj zapisywane są dane wygenerowane przez testowanie z użyciem run_network.py.
data_tsne/	Tutaj zapisywane są dane wygenerowane po uruchomieniu run_tsne.py.
run_network.py	Program umożliwiający stworzenie sieci, nauczenie jej i przetestowanie. W zależności od ustawionych flag, niektóre fazy można pominąć.
README.txt	Opis możliwości sieci wraz z podanymi przykładowymi uruchomieniami.
run_tsne.py	Program do użycia algorytmu t-SNE. Korzysta on z implementacji dostępnej w bibliotece <i>scikit-learn</i> .
utils.py	Dodatkowe funkcje, wykorzystywane przez inne programy.
network.py	Implementacja sieci DBN.

³Korzystałem z narzędzia dostarczonego przez <https://plot.ly/>, które pozwala na tworzenie przejrzystych interaktywnych wykresów.

data_loader.py	Program zawierające funkcje do wczytania i przetwarzania danych.
----------------	--

4.2 RNN

Do stworzenia i nauki rekurencyjnych sieci neuronowych wykorzystano biblioteki - RNNLM oraz Keras. Pierwsza z nich jest dość prosta w użyciu i nie pozwala na dużo modyfikacji, zaś druga oferuje duży wachlarz możliwości, dzięki czemu możemy stworzyć własną sieć (nie tylko rekurencyjną), nakładając na siebie kolejne elementy (np. warstwy) jak klocki Lego. Analiza poprzedzona jest wstępnią obróbką danych wejściowych. W zależności od ich specyfiki, faza ta może być mniej lub bardziej czasochłonna. Przygotowanie danych dla potrzeb niniejszej analizy było procesem dość złożonym, dlatego też większość załączonych programów służy do przetwarzania danych. Wszystkie programy podzielono na 5 grup i opisano w kolejnych podrozdziałach.

Zarówno foldery jak i nazwy plików poprzedzone są prefiksem liczbowym, będącym sugestią dotyczącym tego, w jakiej kolejności powinny być uruchomione. Nienajlej jednak nic nie stoi na przeszkodzie, by skorzystać z każdego pliku w dowolnym momencie w zależności od potrzeb.

4.2.1 Przetwarzanie Korpusu Rzeczpospolitej

Wszystkie programy związane z tym zagadnieniem znajdują się w folderze **1_rzepa/**.

nazwa pliku / folderu	opis
1_strip_html_rzepa.py	Korpus Rzeczpospolitej to zbiór plików HTML. Program ten przetwarza je, poprzez usunięcie zbędnych informacji (w tym tagów), zostawiając czysty tekst, a następnie dzieli go na zdania. Dla każdego artykułu (pliku HTML) zostaje wygenerowany nowy plik, w którym jedna linia zawiera dokładnie jedno zdanie poprzedzone tagiem (jest to dodatkowa informacja nt. tego czy dana linia to część artykułu, czy jego tytuł, podtytuł, bądź autor).

2_rzepa_to_rnndl.py	Przetwarza dane do formatu, który będzie później użyty w bibliotece rnndl, tj. scala wszystkie pliki w jeden i usuwa tagi z początku linii, zostawiając czysty tekst.
3_split_train_test.py	Dzieli dane na treningowe, testowe i walidacyjne wg zdefiniowanej proporcji, podanej jako parametr.

4.2.2 Przetwarzanie danych testowych - Wikipedii

Wszystkie programy związane z tym zagadnieniem znajdują się w folderze **2_wiki/**.

nazwa pliku / folderu	opis
1_wiki_to_rnndl.py	Dane testowe mają format taki jak pokazano w 3 rozdziale, tj. w każdej linii jest inne zdanie, a słowa w alternatywach rozdzielone są znakiem " ". Program ten, przetwarza je tworząc tyle zdań, ile jest możliwych kombinacji wynikających z alternatyw i każde z nich umieszcza w oddzielnej linii.
2_wiki_to_dict_no_repeats.py	Program analizuje dane i na ich podstawie generuje słownik stworzony z wszystkich wyrazów zawartych w danych. Każdy wyraz występuje tylko raz. W jednej linii znajduje się tylko jeden wyraz.
2_wiki_to_dict_with_repeats.py	Program działa podobnie jak powyższy, przy czym słowa mogą się powtarzać. Powtórzenia są potrzebne, gdy chcemy wybrać mniej słów ze słownika, sugerując się częstotliwością wystąpień.
3_wiki_count_sentences.py	Dla każdego zdania oblicza liczbę możliwych kombinacji wynikających z alternatyw.
4_count_baseline.py	Liczy baseline zgodnie z algorytmem podanym w 3 rozdziale.

4.2.3 Dodatkowe narzędzia

Podczas pracy nad danymi, stworzono wiele programów, ale w pracy umieszczone tylko te najważniejsze. Niektóre z nich nie pasowały bezpośrednio do żadnej z pozostałych kategorii, ponieważ łączyły funkcjonalności kilku z nich, dlatego umieszczone je w folderze **3_tools/**.

nazwa pliku / folderu	opis
1_rnnlm_to_rnnlm_with_dict.py	Wejściem do programu są dane treningowe w formacie gotowym do użycia w RNNLM oraz słownik. Program przetwarza dane treningowe tak by zawierały tylko słowa ze słownika, a resztę zamienia na tag <i><unk></i> .
2_scores_to_results.py	Po przepuszczeniu danych testowych przez sieć neuronową, każdemu zdaniu przyporządkowane jest prawdopodobieństwo. Program ten przetwarza je, sprawdzając, ile błędów miały najlepsze, wg sieci zdania i generuje jedną liczbę oznaczającą skuteczność sieci.

4.2.4 Wykorzystanie biblioteki RNNLM

Programy i foldery związane z uczeniem i testowaniem RNN za pomocą biblioteki RNNLM zostały umieszczone w folderze **3_rnnlm/**.

nazwa pliku / folderu	opis
data/	W tym miejscu trzymane są dane treningowe.
models/	W tym miejscu zapisywane są dane nt. nauczonej sieci, które można później wczytać i użyć do testów bez ponownej nauki.
results/	W tym miejscu zapisywany jest wynik ze skutecznością sieci.
rnnlm-0.4b/	Miejsce, w którym należy umieścić bibliotekę RNNLM ⁴ .

⁴Do ściągnięcia z <http://rnnlm.org/>

run/	Folder ze skryptem bashowym, który na podstawie podanych parametrów sieci, tworzy ją, trenuje, testuje i podaje skuteczność sieci, zapisując przy tym pośrednie wyniki do poszczególnych folderów.
scores/	W tym miejscu zapisywane są wyniki testów, tj. prawdopodobieństwa poszczególnych zdań.

4.2.5 Wykorzystanie biblioteki Keras

Część wykorzystująca bibliotekę Keras jest nieco bardziej rozbudowana, ponieważ oprócz użycia wcześniejszych programów, sama posiada sporą część przetwarzającą dane. Wynika to m.in. z tego, że dane muszą być w innym formacie. Dodatkowo bardzo często nie mieszczą się one w pamięci, dlatego została również zaimplementowana możliwość ich dzielenia i wczytywania na bieżąco. Przygotowany skrypt bashowy ma też dużo więcej możliwości, a całość jest bardziej zautomatyzowana.

nazwa pliku / folderu	opis
0_original_data/	W tym miejscu trzymane są gotowe dane, tak jak słowniki, dane testowe czy treningowe (wszystkie w formacie gotowym do użycia w RNNLM).
1_dicts/	W zależności od tego jakie ustawimy parametry sieci (w szczególności wielkość pierwszej warstwy) to potrzebny będzie odpowiedni słownik wygenerowany z danych, które też można podać. Jeżeli dany słownik już istnieje, to zostanie on pobrany z tego folderu, jeżeli nie, to zostanie utworzony przed rozpoczęciem nauki sieci.
2_data_for_train/	Do nauki wykorzystywane były te same dane (Korpus Rzeczypospolitej), ale nie zawsze był to cały korpus. Mogą się one również różnić w zależności od użytego słownika oraz tego, na ile plików je podzielmy, tak by zmieściły się w pamięci. W tym folderze zapisywane są dane, które gotowe są do użycia po uwzględnieniu opisanych parametrów.

3_results/	Każda nowa sieć ma swój katalog w tym folderze, a w nim znajduje się model w postaci pliku JSON (w takim formacie sieci przetrzymuje Keras) oraz zapisane parametry sieci (tj. wagi). Wszystkie informacje jakie logowane są przez moje programy zapisywane są do pliku z logami i po zakończonej nauce, bądź testach, przenoszone do tego folderu.
m1_create_dict.py	Tworzy słownik z dowolnych danych dodając przy tym tagi początku i końca sekwencji oraz uniwersalny tag dla słów, których nie ma w słowniku.
m2_prepare_data.py	Przygotowuje dane treningowe na podstawie zadanych parametrów, zgodnie z opisem podanym przy folderze 2_data_for_train/. Dane są wtedy gotowe do łatwego użycia w bibliotece Keras.
m3_train.py	W tym programie najpierw budowany jest model sieci rekurencyjnej z wykorzystaniem LSTM, po czym następuje nauka na podstawie wczytywanych na bieżąco danych treningowych.
m3_test.py	Dane testowe mają nieco inną charakterystykę niż dane treningowe, np. musimy pamiętać ile było alternatyw w danej grupie itp. Z drugiej strony są na tyle małe, że nie musimy ich dzielić na pliki, dlatego zamiast postępować analogicznie jak w przypadku treningu, który podzielony jest na kilka faz, wszystko odbywa się w tym jednym pliku. Dane są odpowiednio dostosowane do formatu wymaganego przez Keras, model zostaje wczytany, a dane przetworzone przez sieć. Uzyskane prawdopodobieństwa używane są wtedy do policzenia skuteczności sieci. Wynik ten zostaje zapisany później w logach.
preprocessing.py	W pliku tym znajdują się wszystkie funkcje, z których korzysta więcej niż jeden program.

run.sh

Skrypt ten służy do ustawienia parametrów sieci i uruchomienia po kolej wszystkich faz, które doprowadzą nas do stworzenia nowej sieci, nauczenia jej i przetestowania.

Rozdział 5

Wnioski i możliwości rozwoju

Celem pracy było wykorzystanie nowoczesnych sieci neuronowych w zadaniach przetwarzania języka polskiego. Tematyka sieci neuronowych jest bardzo szeroka, dla tego wybór konkretnych rozwiązań inspirowany był sukcesami poszczególnych sieci w języku angielskim. Przedstawione wyniki są obiecujące i stanowią dobry początek do dalszych badań. Praca ta powstawała przez ponad 1,5 roku. Duża ilość czasu poświęcona tej tematyce zaowocowała poznaniem ogromu możliwości jakie dają sieci neuronowe. Poniżej przedstawiono listę pomysłów, które nie zostały zrealizowane, ale mogą poprawić jakość sieci w rozwiązywanych problemach:

- Zarówno dane testowe, jak i treningowe zostały z góry określone, aczkolwiek tylko te pierwsze są nieodłącznym elementem zadania. Zastosowanie innych danych treningowych mogłoby zwiększyć jakość sieci. Nowe informacje mogłyby być użyte zarówno zamiast jak i obok dotychczas używanych. Przykładem innych danych jest Narodowy Korpus Języka Polskiego¹,
- Kolejnym aspektem oraz etapem przy pracy z danymi jest to, w jaki sposób zaprezentujemy je w formie wektorowej tak, by mogła je przyjąć warstwa wejściowa sieci neuronowej.
 - W pierwszym zadaniu orzeczenia reprezentowane były przez wektory długości 2000. Zwiększenie słownika wydłużyło by proces nauki, ale mogłoby

¹<http://nkjp.pl/> - niestety publikowany tylko w formie N-gramów (gdzie N=1,2,3,4,5).

również poprawić skuteczność sieci neuronowej. Binarne podejście, w którym element w wektorze jest oznaczony przez 1 w momencie, gdy dane słowo występuje przynajmniej raz w tekście, może zostać zastąpione przez bardziej skomplikowane metody, takie jak tf-idf², generujące ciągłe reprezentacje na przedziale [0, 1].

- Przy drugim zadaniu poruszono temat zmniejszania słownika, tudzież wektorów reprezentujących słowa, poprzez zastosowanie klas lub bardziej zaawansowanych algorytmów reprezentujących wyrazy w niskowymiarowej przestrzeni. Przykładem jest word2vec, który znalazły zastosowanie w obu zadaniach. W przypadku biblioteki Kerass, umożliwiły to użycie pełnych słów, zamiast sufiksów.
- Uczenie sieci neuronowej trwa dość długo, dlatego dobranie odpowiednich parametrów nie jest zadaniem trywialnym. Zautomatyzowanie całego procesu oraz zastosowanie walidacji krzyżowej lub innego podejścia pomogłoby w odpowiednim wyborze m.in.:
 - funkcji aktywacji
 - struktury sieci (tj. wielkości poszczególnych warstw)
 - współczynnika uczenia
 - formatu danych
 - funkcji kosztu (straty)
- LSTM jest tylko jednym z podejść do problemu ze znikającym gradientem przy BPTT. Istnieją również inne rozwiązania, takie jak np. zaprezentowany w 2014 roku GRU [4], niekoniecznie będący rozwiązaniem lepszym, ale charakteryzującym się mniejszą liczbą parametrów.

²<https://en.wikipedia.org/wiki/Tf-idf>

Bibliografia

- [1] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.
- [2] Denny Britz. Recurrent neural networks tutorial, part 1 - introduction to rnns, 2015. [on-line] <http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/>.
- [3] Juan Cazala. Mnist digits. [on-line] <https://www.npmjs.com/package/mnist>.
- [4] Kyunghyun Cho, Bart van Merriënboer, Çağlar Gülcehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. 2014.
- [5] G E Hinton and R R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, July 2006.
- [6] Geoffrey E. Hinton. A practical guide to training restricted boltzmann machines. In *Neural Networks: Tricks of the Trade - Second Edition*. 2012.
- [7] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks, 2015. [on-line] <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>.
- [8] LISA Lab. Restricted boltzmann machines (rbm). [on-line] <http://deeplearning.net/tutorial/rbm.html>.
- [9] Yann LeCun. The mnist database of handwritten digits. [on-line] <http://yann.lecun.com/exdb/mnist/>.
- [10] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. 2013.
- [11] Tomas Mikolov, Stefan Kombrink, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur. Extensions of recurrent neural network language model. 2011.
- [12] Tomáš Mikolov, Stefan Kombrink, Anoop Deoras, Lukáš Burget, and Jan Černocký. Rnnlm - recurrent neural network language modeling toolkit. 2011.

- [13] Michael Nielsen. Neural networks and deep learning, 2014. [on-line] <http://neuralnetworksanddeeplearning.com/>.
- [14] Christopher Olah. Understanding lstm networks, 2015. [on-line] <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>.
- [15] Korpus rzeczypospolitej. [on-line] <http://www.cs.put.poznan.pl/dweiss/rzeczypospolita>.
- [16] L.J.P. van der Maaten and G.E. Hinton. Visualizing high-dimensional data using t-sne. 2008.