Тема 3 «Нейронні мережі» (частина 2)

Отже, у попередньому матеріалі ми вивчили, що конволютивні нейронні мережі (НМ), або НМ зі згорткою вирішують декілька проблем:

1. Зменшують кількість параметрів для навчання на порядки і, таким чином, дозволяють тренувати НМ на величезних тренувальних вибірках.
2. Дозволяють не прив'язуватись до локальних закономірностей, а вивчати ознаки незалежно від їх позиції на зображенні.
3. Дозволяють встановити, які фільтри були натреновані і використовувати їх у інших задачах. Досить часто в певній прикладній сфері ми можемо не знати, який набір фільтрів підходить для ідентифікації певного набору ознак. Навіть, якщо ми не будемо їх використовувати у НМ, але процес тренування НМ на певні задачі можна використати, як спосіб знаходження фільтрів для «напівручної» ідентифікації певних ознак.

Архітектура у вигляді чергування конволютивних і макс-пулінг шарів, (які обирають максимальні значення), в комбінації з прихованими шарами і рівнем активації називається **VDG-архітектурою НМ**, яка свого часу показала найкращі результати на конкурсі ImageNet, який проводиться з 2012 року, і де різні команди пропонують свої архітектури алгоритмів для задачі класифікації зображень.

З появою конволютивних НМ розмір помилки найкращого алгоритму зменшився з 17% до 7%. Зараз, комбінуючи конволютивні шари та розпаралелюючи їх в деяких архітектурах вдалося зменшити помилку до 3%.

Є ще один вид НМ, які застосовуються в дещо відмінних задачах, ніж звичайні одношарові або багатошарові перцептрони чи конволютивні мережі. Кожного разу, коли звичайна НМ «дивиться» на певний приклад, вона його бачить наче вперше, оскільки всередині НМ є певний набір зв'язків, який має певну визначену конфігурацію внаслідок тренування, і кожен окремий приклад сприймається абсолютно незалежно без врахування контексту.

Але в реальному житті досить часто бувають ситуації, коли класифікація певних даних залежить від того, які приклади ми бачили перед даним прикладом.

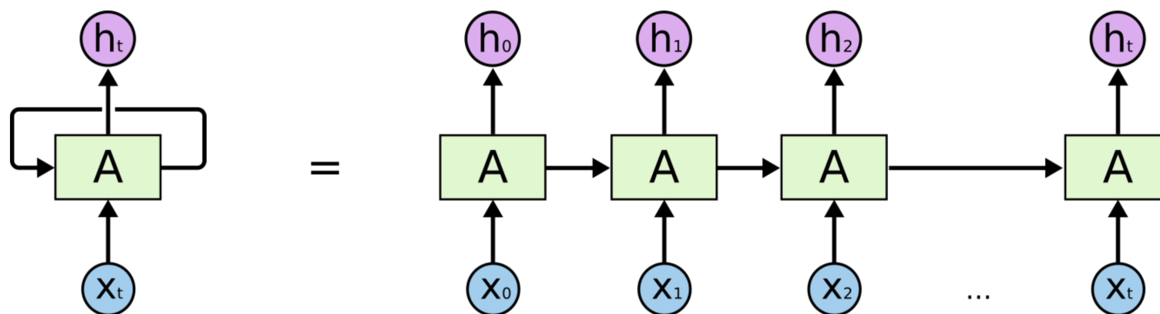
Звичайна ситуація – ми читаємо текст: ми не читаємо кожне слово наче вперше і ніби це перше слово на сторінці. Ми враховуємо, яке слово було попереднім, які слова були перед тим. Тільки в контексті послідовності ми розуміємо сенс, який записано.

Зокрема, якщо ми хочемо навчити комп’ютер частково розуміти текст, відповідно, звичайної архітектури одношарового або багатошарового перцептрона нам буде недостатньо для того, щоб НМ могла зрозуміти контекст послідовності.

І для роботи з послідовностями ми маємо декілька вимог для НМ. Перша і основна із них: НМ **має враховувати контекст**, який змінюється ітерація за ітерацією, приклад за прикладом, а також вміти ефективно репрезентувати кожен приклад у загальному контексті.

Що було запропоновано більше 50 років тому?

Першими очевидними ідеями було взяти одношаровий перцептрон і виходи мережі **зациклити** на вхід. Відповідно, кожен приклад спочатку проходитиме через мережу і призведе до певного рівня активації. Тобто береться вихід першого прихованого шару, додається на вхід, і, в результаті, сигнал другого прикладу буде змішаний з результатами обробки попереднього.

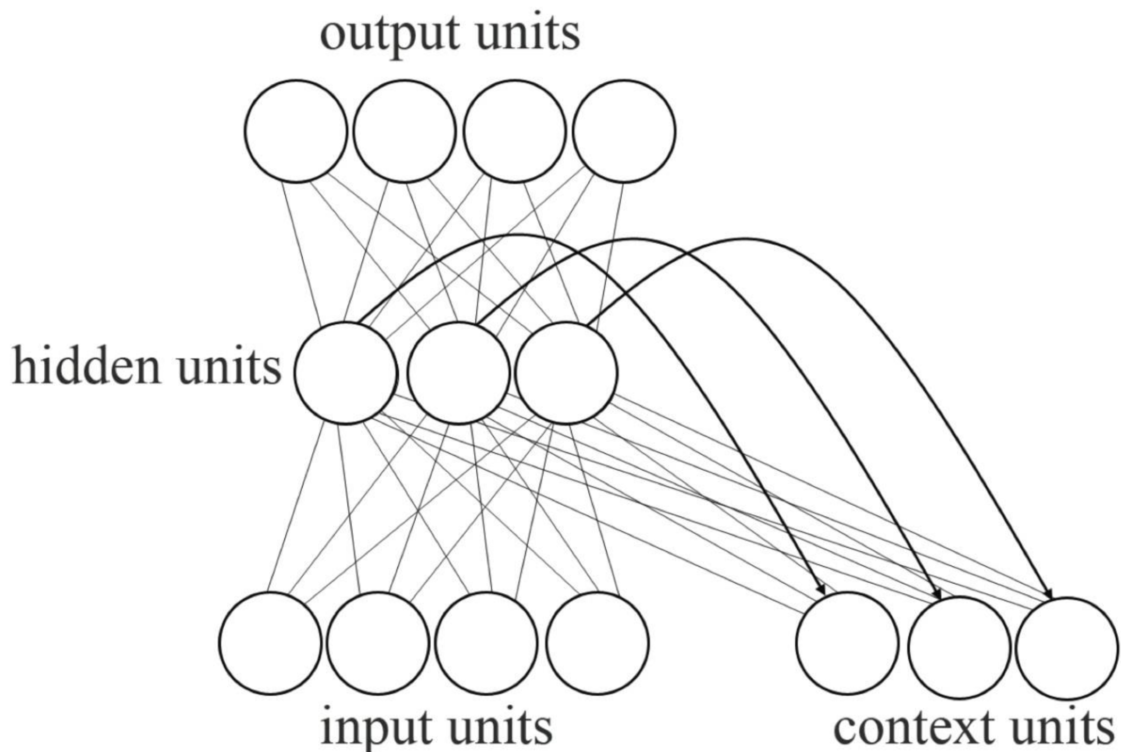


Таким чином контекст попередніх прикладів частково буде зберігатися.

Але такий підхід безпосереднього змішування сигналів призводив до того, що практична цінність таких мереж була невисока, оскільки сигнали змішувалися і навчити систему було важко. Досить часто вона не досягала конвергенції. І часто зв’язки в системі представляли собою фактично шум.

У 80-х рр. була запропонована архітектура **рекурентної нейронної мережі Елмана.** Взагалі, всі типи НМ, які використовують зворотні зв’язки, називаються рекурентними НМ, оскільки фактично це рекурентна функція, яка враховує попередні ітерації.

Архітектура рекурентної НМ Елмана, по суті, подібна до попередніх запропонованих варіантів. Єдина відмінність полягала в тому, що виходи прихованого шару з попередньої ітерації ми додаємо, змішуємо чи множимо на наступні входи не напряму, а направляємо на додатковий вхідний шар, т.зв. **шар контексту**.



Відповідно, загалом схема виглядатиме так: ми маємо вхідний шар нейронів, який проектується на прихований шар (один чи декілька), виходи прихованого шару передаються на вихідний шар, а також копіюються на шар контексту, який на наступній ітерації сприймається разом із вхідним шаром і з'єднаний із прихованим шаром.

Відповідно ми отримуємо цикл: прихований шар – шар контексту – прихований шар.

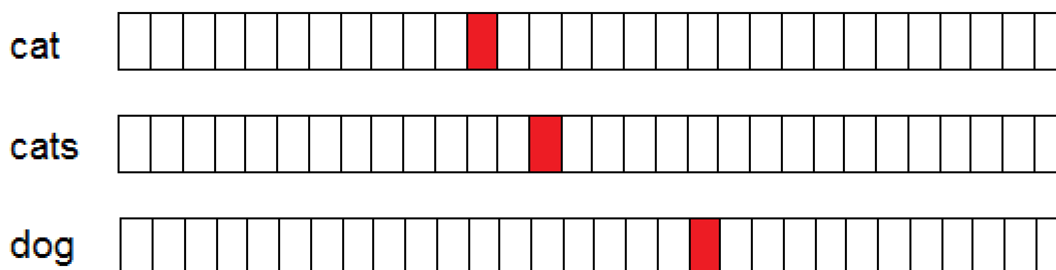
І, якщо ми спостерігаємо послідовність прикладів, відповідно, інформація в закодованому вигляді прихованого шару постійно буде циркулювати по даному колу і з кожним прикладом дещо модифікуватись.

В процесі роботи нашої НМ будуть поступати нові приклади, що змінюватимуть контекст, і контекст, який циркулює всередині мережі і зберігає інформацію про те, які були приклади, буде зберігати цінну інформацію, яка впливає на поточну класифікацію.

Нейронна мережа Елмана. Приклад роботи.

Наприклад, припустимо у нас є задача прогнозування наступного символу і ми хочемо навчити систему дописувати слова.

У нас є словник – 30 символів. Будь-який текст ми можемо зобразити, як набір бінарних векторів розмірністю 30. Відповідно, якщо ми спостерігаємо в тексті символ А, а в нашому словнику це елемент з індексом 0, відповідно нульовий елемент в нашому векторі буде 1, всі решта 0. Якщо спостерігаємо С, то другий елемент – 1, всі інші 0. Такий метод кодування називається «один гарячий приклад» - one hot coding.



Відповідно, якщо ми маємо певне речення, чи просто слово, тобто послідовність певних символів, ми кодуємо його у вигляді наших one hot-векторів. Якщо слово складається із 10 букв, ми отримуємо послідовність із 10 векторів розмірністю 30, де кожному символу відповідає 1 активний біт.

Отже, розмірність вхідного шару в НМ повинна бути 30. Якщо ми читаємо слово «нейронний», то на першій ітерації активуємо індекс, який відповідає букві Н, пропускаємо даний імпульс через НМ до прихованого шару. Прихований шар активується і на виході ми маємо вихідний шар розмірністю теж 30, тому що на виході нашою міткою буде індекс наступної букви.

Відповідно, якщо ми хочемо навчити нашу НМ слову «нейронний», ми очікуємо, що після букви Н наша мережа видасть очікування індексу букви Е.

Як ми хочемо, щоб система працювала в ідеалі?

Ми даємо певну послідовність індексів літер, на виході отримуємо індекси наступної літери. Даємо індекс наступної літери на вхід, отримуємо на виході очікування наступного індексу і т.д. Таким чином ми мали б розкодувати слово. І така мережа вирішувала б задачу продовження послідовності по впізнаному паттерну.

Якби ми тренували цю НМ як одношаровий чи багатошаровий перцептрон, кожна буква б сприймалася б абсолютно окремо, і після кожної букви Н, якщо система бачила лише декілька прикладів і в усіх після Н ішла буква Е, система б на виході видавала б після букви Н однакове очікування розподілу літер, які ідуть за нею, і постійно видавала б букву Е.

Але, якщо ми хочемо, щоб наша система враховувала попередній контекст, оскільки в залежності від того, що за слово, змінюються наступні літери, ми з'єднуємо прихований шар із додатковим шаром контексту.

Це ніби ще один додатковий вхідний шар. Він буде однакової розмірності з прихованим шаром. На першій ітерації активація цього шару контексту нульова, а на другій – міститиме активацію попереднього прихованого шару. Відповідно, коли ми прогнали букву Н, отримали на виході певне очікування, скорегували методом зворотного поширення помилки ваги, передали контекст на шар контексту і букву Е з очікуванням наступної букви Й ми передаємо вже з врахуванням попереднього контексту букви Н.

В результаті, повторюючи сотні ітерацій і подаючи нашій НМ текст, з часом вона вивчить певні закономірності. Як правило, найбільш вагомими будуть чисто статистичні закономірності певних складів, які частіше зустрічаються.

З часом, при вдалій тренувальній вибірці і підходящій архітектурі, наша НМ зможе прогнозувати закінчення слів по початку слова.

Такі НМ можна використовувати для класифікації послідовностей, коли у нас є послідовність певних векторів, подій і нам потрібно їх класифікувати. Досить часто рекурентні мережі використовуються у т.зв. **natural language processing** – обробка природної мови, оскільки в природну мову людей досить важко формалізувати простим набором певних чітких правил, які можна розкласти, і зрозуміти, який сенс у цьому реченні.

LSTM-архітектура нейронних мереж

Класична НМ Елмана, яку ми щойно розглянули, досить ефективна з одношаровим чи багатошаровим перцептроном, але й вона має недоліки.

Основний недолік – **згасання впливу прикладів по віддаленню**.

Як правило максимальний вплив на відповідь нашої НМ на певній ітерації мають приклади, які були на попередній ітерації, на двох ітераціях назад і т.д., і чим далі, тим цей вплив зменшується.

Відповідно, наша НМ має досить коротку пам'ять. На відміну від одношарового перцептрону вона її хоча б має, але згасання є серйозним недоліком. І досить часто бувають ситуації, коли інформація, важлива для правильного прогнозування, знаходиться не на найближчих прикладах, а на 10-20-30 ітерацій назад.

Як вирішити проблему забування і одночасно не перетворити НМ в надгігантську архітектуру, яку буде важко натренувати?

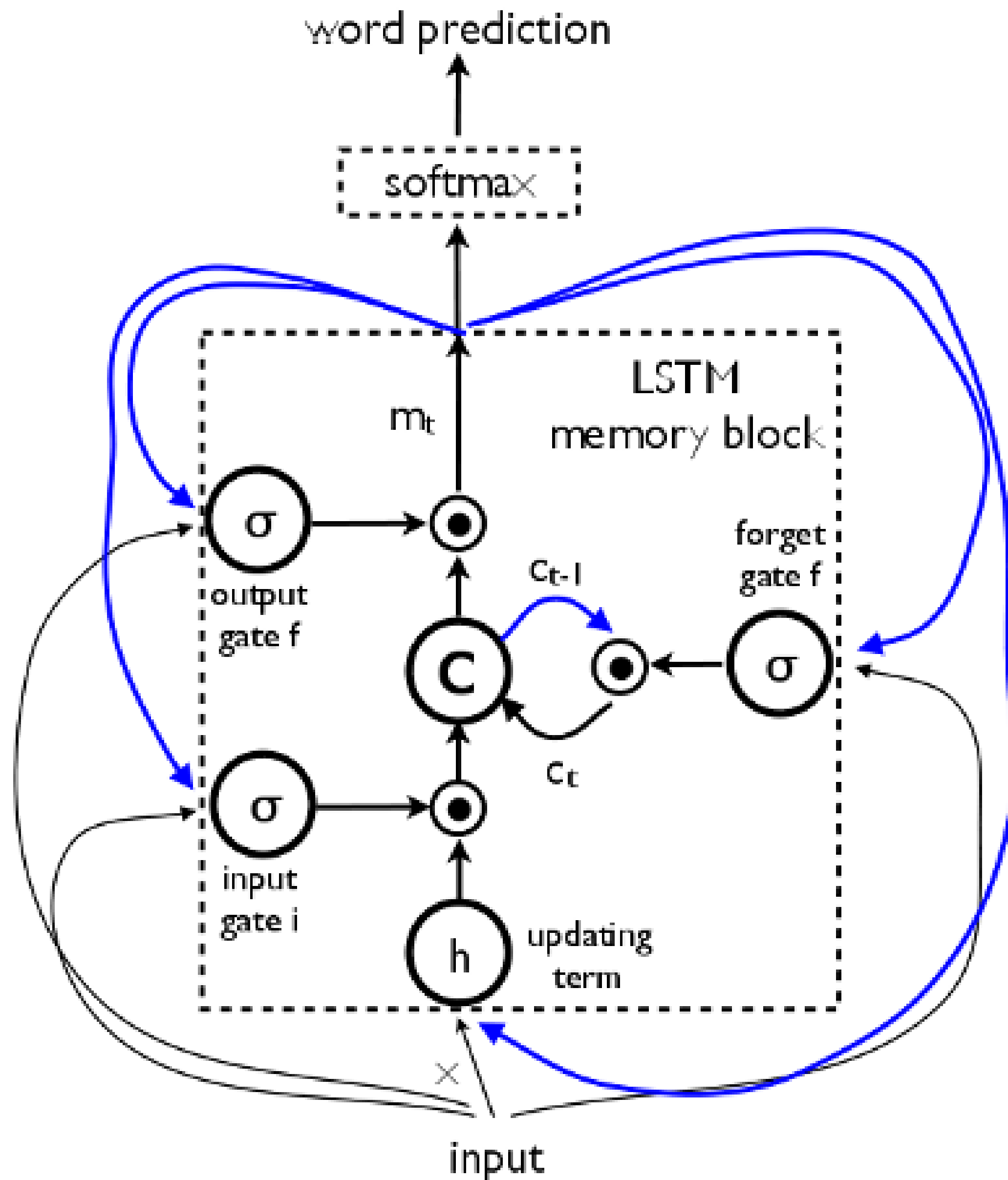
Останнім часом стала популярною архітектура рекурентних НМ, які називаються **long short-term memory neural networks - LSTM (довга короткострокова пам'ять)**.

Ідея полягає в тому, що, оскільки різні приклади в послідовності мають різний вплив на те, яку відповідь ми повинні видати на поточній ітерації, пропонується розробити метод, який буде надавати певним елементам в контексті в попередніх ітераціях **більшу вагу**, більший вплив на результат, а певним – менший.

За рахунок чого це можливо?

Якщо ми візьмемо будь-яку рекурентну НМ і розгорнемо її в часі, ми помітимо, що її можна розглядати, як послідовність великої кількості однакових НМ, кожна з яких передає свій результат на вхід наступній. І ми фактично економимо місце, стискаючи їх всі в одну цілісну структуру.

В LSTM НМ пропонується розширити класичну схему рекурентної НМ таким поняттям, як **gate** – елемент всередині НМ, який буває **memory gate** і **forget gate**, і який визначає, з якою **імовірністю** ми маємо даний приклад забути чи запам'ятати для наступних ітерацій.



На збереження чи видалення прикладу впливають попередні приклади. Якщо вони свідчать про те, що дане слово в реченні відіграє важливу роль для майбутньої класифікації наступних слів, відповідно ми надамо вагам на даній ітерації більше значення. Якщо поточне слово відіграє досить слабку роль в якості прогнозування на подальших ітераціях, ми зменшуємо відсоток впливу даного прикладу на наступні ітерації.

Наприклад, в розмовних реченнях дуже часто певні займенники (слова типу АЛЕ, АБО, ТОМУ) відіграють менш важливу роль для розуміння того, про що взагалі йдеться, ніж такі слова, як ФРАНЦІЯ, ГУЛЯТИ, КОХАТИ, які досить сильно звужують контекст. Тому перші – матимуть менший вплив, ніж другі.

LSTM-архітектура нейронних мереж. Приклад роботи.

За допомогою LSTM НМ можна вирішити задачу генерування тексту. Відомі багато красивих прикладів, коли на НМ передається велика кількість тексту – набір книжок (ми розглядаємо книжку як послідовність індексів слів), і дані послідовності ітерація за ітерацією передаються LSTM-нейромережі. І їй формують задачу прогнозування індексу наступного слова.

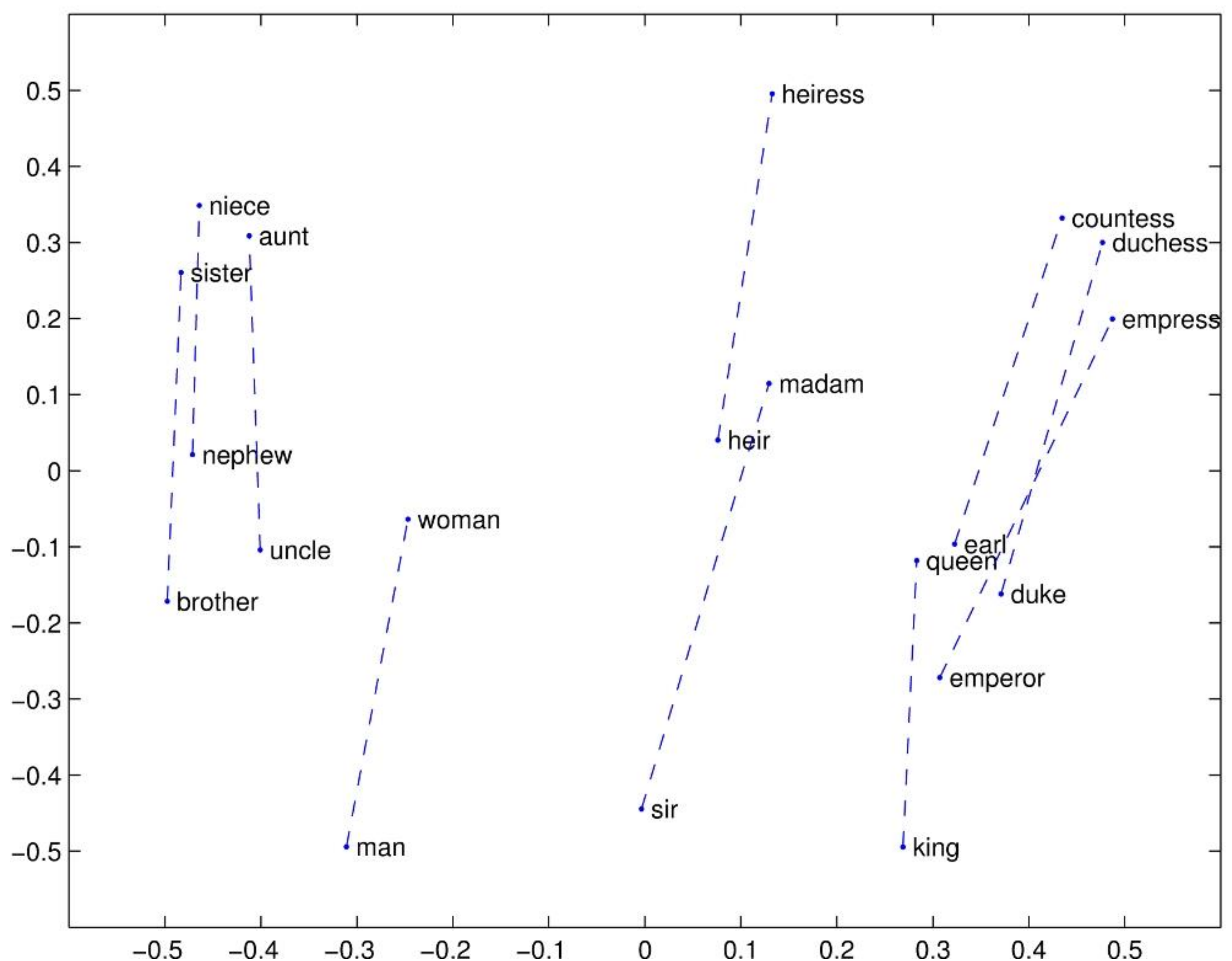
В процесі НМ вивчає досить складні закономірності і, відтворивши процес генерування тексту, тобто даючи на вхід певне слово, а не тренувальну вибірку, ми беремо очікування, що видала мережа, і передаємо його на її вхід. Відповідь змінюється і ітерація за ітерацією ми отримуємо послідовність слів. При достатньо великій кількості тренувальних прикладів і їх різноманітності, результати тренування НМ вражали. Відомі навіть ситуації, коли нейромережі вдавалося згенерувати тексти, подібні до віршів Шекспіра (після опрацювання текстів Шекспіра, звісно).

Ще одна ефективна властивість LSTM НМ та взагалі рекурентних НМ це те, що в процесі ітеративної обробки прикладів, мережа постійно на виході видає певний вектор ознак. Тобто, отримавши на вхід певне слово, ми маємо вектор контексту, отримавши на вхід наступне слово – цей вектор контексту змінюється.

Відповідно, якщо ми натренували нашу рекурентну НМ на достатній кількості прикладів, ми можемо отримати певний енкодер. Припустимо, ми маємо словник на 100 000 слів і НМ, що натренована на задачі прогнозування наступного слова. Надаючи на вхід такої НМ, наприклад, слова в закодованому вигляді one hot-кодування і розмірності векторів 100 000, в якому тільки один елемент активний, ми будемо отримувати після кожної ітерації вектор контексту, який буде відмінний від попередніх. Фактично ми перекодували наш текст в послідовність стиснених векторів вже не розмірністю 100 000, а, наприклад, розмірністю 100 (якщо розмірність нашої LSTM НМ була 100). Відповідно ми перекодовуємо кожне слово в певний стовимірний вектор. А надалі послідовність таких векторів можна аналізувати конволютивними НМ, оскільки ми можемо аналізувати біграми, триграми, чотириграми вже на більш високоабстрактному рівні репрезентації, аналізувати вже не просто індекси слів в словнику, а аналізувати їхній контекст.

Тому, як правило, в задачах аналізу емоційної оцінки коментарів на певному ресурсі або статті, ми робимо тренувальну вибірку, де є декілька тисяч точно відомих позитивних та негативних коментарів. Ми створюємо

словник, в якому кодуємо кожне слово індексом. Далі, використовуючи рекурентну НМ, ми можемо створити енкодер, який перекодує наші індекси слів в щось, що має більше сенсу для НМ: не просто номер в словнику, а вектор, в якому відстань між двома близькими векторами буде відповідати відстані між двома досить подібними словами. Даний процес називається **embedding** – створення простору, в якому приклади будуть розміщені відповідно до їхньої пов’язаності. У випадку мови одним із перших етапів обробки тексту є створення правильних багатовимірних репрезентацій слів. One hot репрезентація є хорошою для початкового навчання певної системи, але аналізувати таку репрезентацію досить важко, оскільки нам нічого не говорить інформація, що дане слово у словнику із індексом 1000, а інше – 10000, хоча, можливо, вони синоніми. Натомість, якщо ми тренуємо систему, в якій подібні слова будуть відповідати векторам, які знаходяться поряд, то, аналізуючи відстань між векторами, ми можемо аналізувати контекст, який є в реченні.



Отже, один із перших етапів розуміння тексту – це створення багатовимірної векторної репрезентації слів, а потім– конструювання НМ, які ітерація за ітерацією аналізують послідовності цих багатовимірних векторів слів і знаходять там закономірності.

Далі ми тренуємо рекурентну НМ класифікатора, або над нею можемо сконструювати конволютивну НМ, оскільки, фактично, якщо ми маємо речення із 20 слів і кожному слову, припустимо, відповідає 200-вимірний вектор, відповідно речення ми можемо репрезентувати як певну картинку розмірністю 200 на 20 і проходитися, як ми пам'ятаємо з випадку НМ зі згорткою, певним віконцем по даній картинці і вивчати фільтри певних типових зворотів, мовленнєвих конструкцій, закономірностей. І ми це робимо вже не вручну, а НМ буде самостійно все ефективніше підбирати фільтри, що корелюють із нашою задачею, методом регресії або методом зворотного поширення помилки. Даний підхід можна використовувати для задач аналізу емоційного забарвлення в тексті.

Тема 3 «Сучасні бібліотеки машинного навчання» Сучасні бібліотеки машинного навчання. NumPy. Scikit-learn.

В попередніх лекціях ми ознайомились з тим, що таке нейромережі, які існують найбільш популярні типові алгоритми вирішення задач машинного навчання (МН), якими методами їх можна навчати та оптимізувати, розглянули базові архітектури.

Хоча машинне навчання – це прикладна наука, оскільки саме явище МН виникло внаслідок того, що людині потрібно використовувати для вирішення прикладних задач, з теоретичної точки зору нашими інструментами в МН є алгоритми.

Алгоритми мають бути реалізовані певною мовою програмування, в певному середовищі, розраховані на певний вид «заліза» і т.п.

Які ж нині існують інструменти, що дозволяють реалізовувати алгоритми МН?

Якщо людина розуміє принцип роботи певного алгоритму, то **будь-яка мова програмування** може підійти для реалізації того чи іншого методу і вирішення прикладної задачі. Звісно, це буде потребувати додаткового часу, можливо ефективність реалізації буде гіршою за аналоги. Але розуміння логіки процесу робить вибір мови чи сфери застосування алгоритму - необмеженими.

За останні роки виникла ціла екосистема бібліотек, в яких досить грамотно реалізовані найбільш відомі базові алгоритми МН. Нині не

обов'язково реалізовувати кожен з алгоритмів з нуля. Зрозуміло, що для людини, яка займається МН, потрібно розуміти, як працює кожен із алгоритмів.

Однією з мов програмування, яка максимально використовується для вирішення прикладних задач, є мова **Python**. Ця мова має декілька переваг. Вона досить проста у вивченні і, як правило, це мова, яка потребує низького рівня входу.

Людина, яка має базові знання з теорії алгоритмів і математики, досить просто може освоїти базовий функціонал, методи і синтаксис для того, щоб вирішувати прикладні задачі.

Саме на базі цієї мови реалізована велика кількість бібліотек, які надають у зручному виді більшість доступних алгоритмів.

Більшість із цих бібліотек використовують бібліотеку **NumPy**. Це бібліотека, яка дозволяє швидко і ефективно працювати з числовими даними, матрицями, таблицями чисел в різних форматах, проводити велику кількість типових операцій, що потрібні в процесі вирішення прикладних задач МН.

Кроки з підготовки нашого середовища розробки для запуску першого алгоритму МН:

- Встановити середовище Python.
- Налаштувати середовище та спробувати запустити Hello world!
- Підготувати інфраструктури через встановлення наборів бібліотек. У цій мові є зручний менеджер пакетів **pip,** який теж потрібно встановити.

Тепер, якщо у нас виникне необхідність використовувати можливості бібліотеки NumPy, в коді при ініціалізації достатньо буде його імпортувати.

Загалом для МН існує декілька базових бібліотек на Python, які мають досить велику перевагу у порівнянні з бібліотеками інших мов програмування. І основна з них: **дуже детальна і якісна документація**. Досить часто бувають ситуації, коли ми маємо хороші алгоритми, але не маємо документації. І потрібно витратити багато часу на розуміння, як працювати з даною реалізацією, що роблять методи тощо.

Одна з бібліотек з чудовою документацією, яка реалізує більшість типових методів МН є **scikit-learn**. В даній бібліотеці реалізовані десятки алгоритмів для задач кластеризації, регресії, класифікації, методу опорних векторів, лінійної та логістичної регресія та десятків інших.

В кожному із доступних алгоритмів є велика кількість параметрів, які ми можемо налаштувати під свою задачу.

До речі, МН завжди потребує попереднього форматування даних для того, щоб натренувати певну модель – потрібно дані попередньо підготувати в відповідному форматі, в певній векторній репрезентації.

Сучасні бібліотеки машинного навчання. Pandas.

Однією із дуже зручних бібліотек всередині мови Python, які допомагають працювати з великою кількістю табличних даних (досить часто тренувальні і тестувальні вибірки виглядають, як .csv-таблиці з сотнями тисяч і мільйонами рядків і колонок параметрів) є бібліотека **pandas**.

Вона дозволяє дуже швидко завантажувати дані, препроцесити їх (готувати у відповідний формат), щоб у зручному вигляді відправляти на опрацювання нашим алгоритмом, який ми, наприклад, вибрали з бібліотеки scikit-learn.

Припустимо, у нас стоїть задача зменшення вимірності: ми маємо багатовимірні дані і ми хочемо подивитися на їх двовимірну проекцію – карту, як дані структурно залежні одне від одного.

Дану задачу можна вирішити алгоритмом зменшення вимірності **t-SNE** (див. документацію самостійно). При встановленні бібліотеки та імпортованому методі, якщо ми маємо 2000 векторів в стовимірному форматі в діапазоні від -1 до 1, процес зменшення вимірності буде виглядати так: ми імпортуємо наші дані з .csv формату, використовуючи **pandas**, конвертуємо у вид певного **NumPy**-масиву, припустимо з float 32 типом даних і ініціалізуємо **t-SNE**-оптимізатор, т.зв. модель. Для початку можемо не вказувати специфічні гіперпараметри, а використовувати стандартні. Якщо ми хочемо отримати двовимірні точки, які відповідають кожному із стовимірних векторів так, щоб ми їх могли розглянути на площині, ми застосовуємо в нашій моделі **t-SNE** метод **fit** або **fit transform**, в який передаємо масив із нашим списком векторів.

В результаті ми отримуємо також список векторів, де кожному відповідному стовимірному елементу з аналогічним індексом буде відповідати набір двовимірних координат, які ми можемо накласти на площину і подивитися, як розподілені наші дані, щоб помітити певні закономірності.

В результаті, весь даний код процесу зменшення вимірності сумарно з імпортами бібліотек буде не більше 20-30 рядків коду.

Сучасні бібліотеки машинного навчання. Theano. TensorFlow. CUDA.

Описане рішення задачі годиться у випадку, якщо нашу задачу можна вирішити стандартними доступними методами бібліотеки **scikit-learn**.

Але, припустимо, ми хочемо сконструювати конволютивну нейронну мережу для задачі ідентифікації котиків. В описаній бібліотеці методи вирішення подібної задачі не реалізовані.

Відповідно, нам потрібно описувати «з нуля» нейронну мережу, що займає багато часу.

Але на даний час завдяки популярності НМ на мові Python існує велика кількість бібліотек, які на досить різних рівнях абстракції (від низькорівневих до високоабстрагованих методів описання архітектури) надають можливість конструювання різноманітних НМ.

Одна із досить поширених бібліотек низькорівневих операцій для реалізації алгоритмів НМ – бібліотека **Theano**. Вона реалізує комплексні матричні мультиплікації, швидкі методи згортки з множенням, вичленовуванням, методи регресії і всю backend-логіку роботи нейронних мереж.

Одним із ключових бонусів таких бібліотек є те, що окрім CPU-реалізації (тобто реалізації роботи алгоритму на процесорі), бібліотеки типу Theano або **TensorFlow** (від Google) є open source, тобто з відкритим кодом (можна його подивитися чи дописати модулі, яких Вам не вистачає).

Це тільки дві з найпопулярніших бібліотек з реалізації backend-логіки НМ і це «конструктор», з якого ми можемо зібрати НМ будь-якої архітектури або складності, модифікувати або скомбінувати методи навчання чи оптимізації.

Один із ключових бонусів – в них реалізована підтримка обчислень на відеокартах. Процес роботи або тренування НМ пришвидшується у 70-300 разів у порівнянні із швидкістю роботи на процесорі, оскільки він виконує операції виконує послідовно, по черзі. При виконанні обчислень на відеокарті, що являє собою систему з великої кількості маленьких процесорів, одна складна задача розбивається на велику кількість маленьких

задач і вони виконуються паралельно. І саме бібліотеки Theano, TensorFlow досить ефективно використовують можливості GPU-обчислень (обчислення на відеокартах).

Слід зауважити, що між відеокартою, мовою Python та бібліотекою, написаною на цій мові є ще один прошарок – **CUDA** – набір бібліотек, реалізований компанією NVidia, які дозволяють ефективно і швидко проводити обчислення на її відеокартах.

Отже, як виглядає структура процесу роботи чи тренування? Маємо НМ з високорівневою архітектурою, написаною мовою **Python**, використовуючи бібліотеку **Theano**, яка в свою чергу використовує функціонал **CUDA**, яка перенаправляє всі обчислення на відеокарту; функціонал **CUDA** отримує результати обчислень і конвертує їх так, щоб ми могли отримати результати в середовищі **Python**.

Сучасні бібліотеки машинного навчання. Keras. Anaconda Python

Окрім backend-реалізації нейронних мереж таких бібліотек, як Theano або TensorFlow, зараз зростає величезна кількість високорівневих методів описання НМ. Це пов’язано з тим, що попри те, що в Theano реалізована велика кількість backend-логіки, тобто всі елементи-конструктори, з яких ми можемо зібрати все, що нам заманеться, тим не менше досить часто нам потрібно заново реалізовувати певні типові конструкції і послідовності, наприклад, конволютивних мереж з типовими рівнями активації.

Тому для спрощення роботи існує вже декілька десятків високорівневих надбудов над цією бібліотекою, які дозволяють описати архітектуру і запустити тренування НМ з 15 шарів, яка показує результати на рівні найкращої НМ зразка 2013 року на конкурсі ImageNet. Весь код такої НМ, описаний на рівні високорівневого API, буде виглядати як 20 рядків коду, з яких 16 рядків – ініціалізація нашої мережі, а решта – запуск тренування і збереження.

Такої лаконічності в ініціалізації НМ можна досягнути, використовуючи бібліотеку **Keras**. Це бібліотека, яка одночасно дозволяє і використовувати або backend бібліотеки Theano, або backend бібліотеки TensorFlow. Оскільки основна логіка зберігається, а відрізняються методи реалізації, то високорівневу структуру наших НМ ми можемо описувати на

високорівневому API, а вже в залежності від потреби змінювати backend, на якому це все обчислюється – на процесорі чи відеокарті.

Ми описали тільки декілька базових популярних бібліотек.

На такому ресурсі, як **GitHub** досить часто є списки бібліотек, де щодня додають нові цікаві бібліотеки з реалізованими необхідними алгоритмами.

Ми згадали саме такі бібліотеки саме тому, що для них доступна зрозуміла і чітка документація. Особливо це важливо для людини, яка вперше займається машинним навчанням.

Кількість бібліотек зростає щодня і навіть виник певний набір бібліотек, які стабільно встановлюються людьми, які постійно займаються МН або обробкою великих даних. Були зроблені цілі пакети бібліотек, які можна встановити за один раз. Це дозволяє перетворити комп'ютер на готову станцію, де можна починати реалізовувати будь-які алгоритми по МН, дата-майнінгу, аналізу великих даних.

Один з таких пакетів – **Anaconda Python** – це повністю налаштоване середовище програмування, в якому попередньо установлені десятки і сотні бібліотек, версії яких **не конфліктують** між собою.

Не потрібно боятись використовувати те, що вже реалізовано, але треба пам'ятати, що бажано знати, як воно працює, і почитати документацію, щоб у випадку непередбачуваних результатів спробувати знайти помилку.