

Stepik. Neural networks and NLP. 3. Basic neural network methods for working with texts

- Part 2

3.5 Основные виды нейросетевых моделей для обработки текстов

Всем привет! Этой лекцией мы начинаем цикл видео о различных архитектурах нейросетей, применяемых для обработки текстов. Конкретно сейчас мы поговорим о видах нейросетевых модулей, перечислим их, чтобы вы знали что они вообще есть, какова их область применения, преимущества и недостатки. Сначала — пару слов о задаче. Смысл использования нейросетей — в том, что они могут выделить сложные паттерны — взаимоотношения в данных. В случае текстов (а именно, отдельных предложений) такие паттерны проявляются в том, как именно слова употребляются вместе, как построены фразы. Другими словами, нейросети нужны для того, чтобы представить контекст в виде математического объекта — вектора или матрицы.

Ранее мы уже говорили о первоначальном переводе текста в матрицу с помощью [эмбеддингов](#) и методов дистрибутивной семантики. Нейросети выполняют следующий шаг: преобразование этой матрицы так, чтобы каждая строчка содержала информацию не об отдельных словах, а о фразах и их смыслах. Самый простой вид модулей — это [свёрточные блоки](#). Всё точно так же, как и в картинках, с одним отличием: используются не двухмерные свёртки, а одномерные. Подробнее о свёрточных блоках мы поговорим в следующей лекции. Свёрточные блоки хорошо подходят для нахождения в данных локальных паттернов, вне зависимости от того, где конкретно эти паттерны встречаются: в начале текста или в конце. Это называется "инвариантность к переносу". Свёрточные блоки эффективно распараллеливаются на видеокартах, поэтому они достаточно быстрые, простые, хорошо учатся. Однако они недостаточно гибкие и мощные, чтобы уметь находить широкие паттерны — например, сравнивать первое слово в предложении и последнее, игнорируя при этом слова между ними. А также, чтобы увеличить максимальную длину паттерна, нужно существенно увеличить количество параметров свёрточной сети. Более естественная идея для текстов — это [рекуррентные нейросети](#). Смысл в том, что мы читаем слово за словом или символ за символом и на каждом шаге поддерживаем некоторый вектор, который содержит информацию о всём тексте, который мы прочитали ранее. В результате могут учитываться достаточно длинные зависимости, а также длина учитываемых зависимостей не обязательно связана с количеством параметров сети. Но учить рекуррентные сети гораздо сложнее.

Цель

- Учет локальных особенностей текста, формулировок
- Локальный контекст

Вход

Текст $\rightarrow Length \times EmbeddingSize$

Выход

$NewLength \times NewSize$



- свёрточные блоки (convolutional neural networks, CNN)
 - по аналогии с изображениями
 - выявление паттернов, вне зависимости от их позиции (инвариантность к переносу в пространстве)
 - быстрые, простые, хорошо учатся
 - могут быть недостаточно гибкими
 - ограничены в ширине контекста
- рекуррентные блоки (recurrent neural networks, RNN)
 - последовательная обработка текста - слово за словом
 - на каждом шаге поддерживается и обновляется вектор скрытого состояния
 - могут учитывать длинные зависимости
 - медленные, сложно учатся



Когда какая-то обработка уже проведена, бывает полезно убрать мелкие детали и оставить только наиболее значимые. Для этого применяются блоки [пулинга](#) или агрегации. Идея их применение в текстах аналогична тому, как они используются для обработки изображений — а именно, соседние элементы матриц усредняются или заменяются на один — например, максимальный. Если же задача — классификация текстов, можно применить глобальный

пулинг, полностью абстрагировавшись от длины исходного текста и получить один вектор фиксированного размера.

- блоки объединения (агрегация, pooling)

- по аналогии с изображениями
- локальный пулинг - уменьшить длину, укрупнить детали
- глобальный пулинг - получить вектор, не зависящий от длины исходного текста



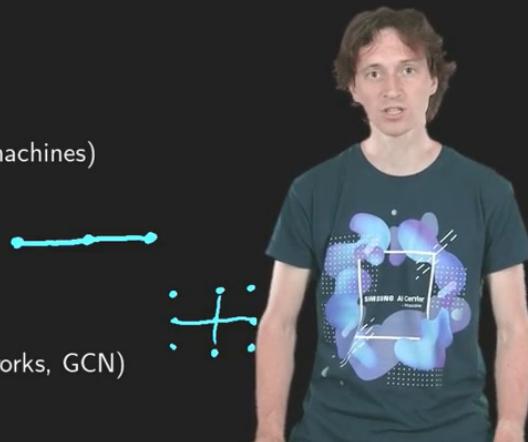
В последние годы один из прорывов был связан с новым видом нейросетевых блоков — механизмами внимания. Более того, оказалось, что достаточно лишь одного "внимания" для решения почти всех задач обработки текстов. По сути, механизм внимания осуществляет попарное сравнение элементов двух последовательностей (или элементов одной последовательности с самой собой), и позволяет выбрать только наиболее значимые их элементы, чтобы продолжить работу только с ними, а всё остальное убрать. Можно рассматривать механизм внимания как умный адаптивный пулинг. Сети "с вниманием", как правило, хорошо учатся. Дальше пошла экзотика, которую мы не будем рассматривать в рамках настоящего курса, однако, будет хорошо, если вы будете знать, что такие вещи вообще есть, каковы их области применения, преимущества и недостатки. Первый экзотический вид нейросетей — это архитектуры с памятью или нейронные машины Тьюринга. Это обобщение [рекуррентных нейросетей](#). Если в рекуррентках у нас только один вектор в ячейке, то в сетях с памятью у нас много таких векторов, и на каждом шаге мы можем выбрать, какой вектор использовать, откуда прочитать и куда записать. Потенциально такие архитектуры могут решать абсолютно любые задачи, однако процесс их обучения сходится не очень хорошо, а также, фактическая ёмкость памяти может оказаться меньше, чем вы планировали. Следующий вид блока — [рекурсивные нейросети](#). Также обобщение рекуррентных сетей, но, на этот раз, обобщение заключается не в добавлении большего количества памяти, а в том, что они работают не с последовательностями, а с деревьями. Они применяются, например, для того, чтобы сначала выполнить синтаксический анализ, а потом пройтись по построенному дереву и

[агрегировать информацию](#) из отдельных узлов. Это может быть полезно для анализа языков со свободным порядком слов — например, для русского. Однако в последнее время этот вариант не очень часто применяется. Графовые [свёрточные нейросети](#) — обобщение и свёрточных, и рекуррентных нейросетей на произвольную структуру графа. Например, в рекуррентных нейросетях у нас обычная цепочка — слово идёт за словом, и каждое связано только с непосредственными соседями. В картинках у нас решётка (4-связанная) — каждый пиксель связан только с непосредственными соседями, а графовые нейросети могут работать с любыми структурами. Этот вид нейросетей получил распространение относительно недавно и сейчас активно исследуется. Однако большая гибкость всегда приходит вместе с большей вычислительной сложностью.

- **механизм внимания (attention mechanism, self-attention)**
 - учит сколь угодно далёких зависимостей
 - попарное сравнение элементов последовательностей
 - "умная" агрегация
 - быстрый, хорошо учатся
- **архитектуры с памятью (memory neural networks, neural Turing machines)**
 - учит сложного контекста
 - потенциально, решение любых задач
 - очень сложно учатся
 - ёмкость памяти невелика
- **рекурсивные нейросети (tree-recursive neural networks)**
 - не путать с рекуррентными!
 - используются для обхода многосвязных структур (деревьев)
 - не всегда дают преимущество перед обычными RNN



- **свёрточные блоки (convolutional neural networks, CNN)**
 - по аналогии с изображениями
 - выявление паттернов, вне зависимости от их позиции (инвариантность к переносу в пространстве)
 - быстрые, простые, хорошо учатся
 - могут быть недостаточно гибкими
 - ограничены в ширине контекста
- **рекуррентные блоки (recurrent neural networks, RNN)**
 - последовательная обработка текста - слово за словом
 - на каждом шаге поддерживается и обновляется вектор скрытого состояния
 - могут учить длинные зависимости
 - медленные, сложно учатся
- **блоки объединения (агрегация, pooling)**
 - по аналогии с изображениями
 - локальный пулинг - уменьшить длину, укрупнить детали
 - глобальный пулинг - получить вектор, не зависящий от длины исходного текста
- **механизм внимания (attention mechanism, self-attention)**
 - учит сколь угодно далёких зависимостей
 - попарное сравнение элементов последовательностей
 - "умная" агрегация
 - быстрый, хорошо учатся
- **архитектуры с памятью (memory neural networks, neural Turing machines)**
 - учит сложного контекста
 - потенциально, решение любых задач
 - очень сложно учатся
 - ёмкость памяти невелика
- **рекурсивные нейросети (tree-recursive neural networks)**
 - не путать с рекуррентными!
 - используются для обхода многосвязных структур (деревьев)
 - не всегда дают преимущество перед обычными RNN
- **графовые свёрточные нейросети (graph convolutional neural networks, GCN)**



- архитектуры с памятью (memory neural networks, neural Turing machines)
 - учет сложного контекста
 - потенциально, решение любых задач
 - очень сложно учатся
 - ёмкость памяти невелика
- рекурсивные нейросети (tree-recursive neural networks)
 - не путать с рекуррентными!
 - используются для обхода многосвязных структур (деревьев)
 - не всегда дают преимущество перед обычными RNN
- графовые свёрточные нейросети (graph convolutional neural networks, GCN)
 - обобщение CNN, RNN и TRNN на произвольную структуру графа
 - новое направление, очень перспективное
 - вычислительно сложнее



Итак, в этом видео мы поговорили о роли различных нейросетевых блоков в процессе обработки текста, а также перечислили и поверхностно обсудили некоторые самые популярные виды нейросетевых модулей. В следующих видео мы подробнее остановимся на свёрточных, [рекуррентных нейросетях](#), а также на механизме внимания.

- роль нейросетей
- краткий обзор видов нейросетевых модулей



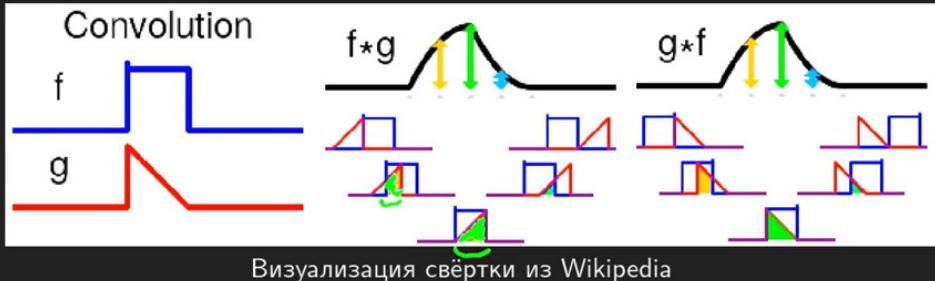
3.6 Свёрточные нейросети для обработки текстов

Всем привет! Вы открыли лекцию про [свёрточные нейросети](#) и их применение в обработке текстов. Сначала давайте вспомним о том, что такое свёртка. Это операция, выполняемая над двумя функциями или двумя сигналами. В результате применения свёртки к двум сигналам получается некий третий сигнал. В контексте нейросетей, первый сигнал — это входные данные, а второй сигнал — это bias, intercept)." > часть параметров нейросети, которые выполняют роль ядра свёртки. Давайте разберёмся как работают свёртки. Допустим, на нашей картинке синий прямоугольный импульс — это входные данные, а красный уголок — это ядро свёртки. И то, и другое, представлено векторами. Мы начинаем двигать ядро свёртки относительно нашего сигнала и для каждого значения сдвига будем считать [скалярное произведение](#) фрагмента сигнала с ядром свёртки. Таким образом мы, как бы, оцениваем, насколько текущий фрагмент сигнала похож на ядро. Слева представлена формула 1-мерной свёртки в общем виде для непрерывного случая. Здесь функция f выполняет роль ядра свёртки, а функция g — роль сигнала. Справа — формула для дискретного случая. Давайте поподробнее на нём остановимся. В дискретном случае результат применения одномерной свёртки — это вектор. Допустим, в нём L элементов. Наше ядро имеет размер k (допустим $k=3$). И есть ещё входной сигнал размера $L+k$.^[1] Раз наше ядро имеет размер k , то раз мы можем анализировать фрагменты сигнала длины k . Мы помещаем ядро так, чтобы его центральный элемент находился над i -ым элементом вектора входного сигнала. А потом вычисляем скалярное произведение ядра и фрагмента входного сигнала. А затем повторяем эту операцию для всех возможных смещений. Это позволяет нам выделять локальные паттерны безотносительно их позиции в сигнале. А ядро свёртки, при этом, описывает паттерн, который мы ищем.

[1] небольшое исправление, должно быть $L+K-1$

Дополнительные комментарии авторов к видео:

- Кроме ядра свёртки в свёрточных модулях есть ещё один набор параметров - [параметры сдвига \(bias, intercept\)](#).
- На 1:53 оговорка, должно быть "L + K - 1"

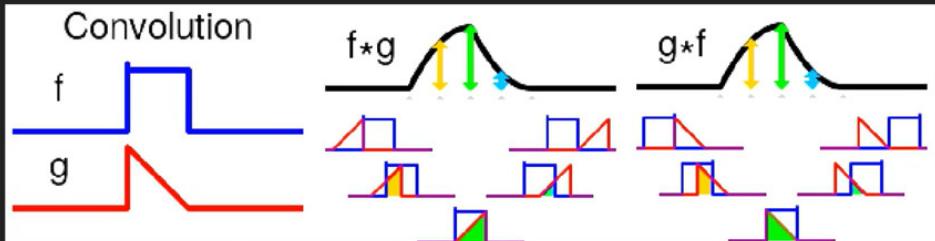


Визуализация свёртки из Wikipedia

$$(f*g)(x) = \int_{-\infty}^{\infty} f(y)g(x-y)dy$$

$$ConvOut_i = \sum_{k=1}^K InSignal_{i-K/2+k} \cdot Kernel_k \quad \forall i = 0, 1, \dots L$$

- перебираются все возможные смещения ядра относительно сигнала
- для каждого смещения измеряется сходство фрагмента сигнала с ядром



Визуализация свёртки из Wikipedia

$$(f*g)(x) = \int_{-\infty}^{\infty} f(y)g(x-y)dy$$

$$ConvOut_i = \sum_{k=1}^K InSignal_{i-K/2+k} \cdot Kernel_k \quad \forall i = 0, 1, \dots L$$

- перебираются все возможные смещения ядра относительно сигнала
- для каждого смещения измеряется сходство фрагмента сигнала с ядром
- выделение локальных особенностей сигнала (паттернов) вне зависимости от позиции
- описание искомого паттерна - ядро свёртки



Из комментариев:

Так вот почему так важен пулинг для сверточных нейросетей..

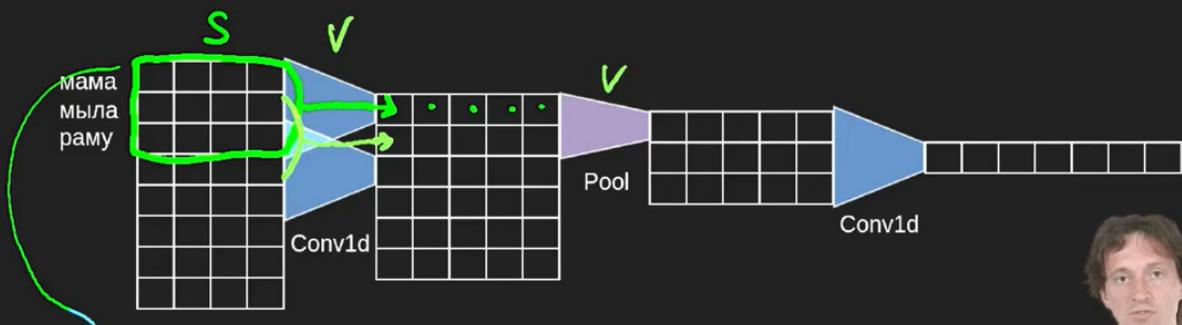
Мое дополнение:

Очень крутой на самом деле комментарий, и в целом объяснения Романа Суворова сути свертки, которое как то я не уловил на курсе по компьютерному зрению (т.е. технически там все разжевано и понятно, но сама суть...). Т.е. один из самых распространенных вариантов

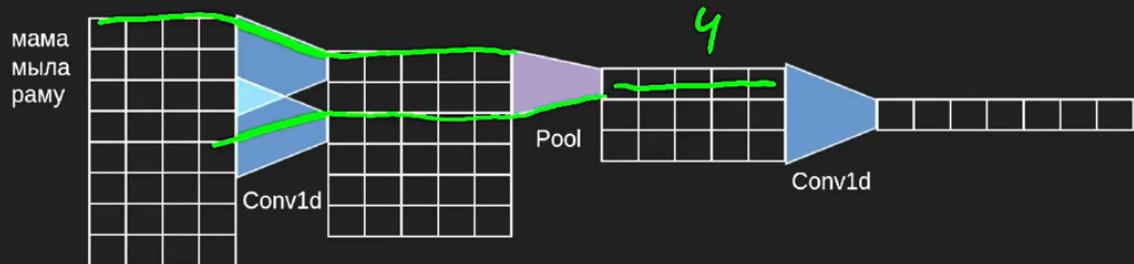
пулинга - MaxPooling, оставляет только наиболее близкие к паттерну случаи, и выкидывает все остальные, менее значимые случаи.

К текстам свёртки применяются следующим образом. Сначала мы строим матрицу [эмбеддингов](#) слов или символов. Количество строк соответствует длине текста. Затем мы поочереди перебираем смещения i , для каждого значения смещения, мы выбираем k целых строк из входной матрицы. Затем мы вытягиваем эти строки в один вектор и вычисляем [скалярное произведение](#) с ядром. Соответственно, в данном случае, ядро будет иметь длину k , то есть ["размер ядра" умножено на "размер входного эмбеддинга"]. Так мы получили одно значение для вектора текущего шага. Чтобы получить остальные значения нам нужно взять ещё таких же ядер и тоже их применить. Затем мы переходим к другому значению смещения и повторяем такие же операции. Обычно [свёрточные блоки](#) применяют один за другим, перемешивая с блоками [пулинга](#) или агрегации, а также с функциями активации. Делают это для того, чтобы расширить пятно восприятия, то есть получить возможность обрабатывать более широкий контекст, более длинные паттерны (например, на построение вот этого вектора влияют входные вектора — вот эти, то есть на этом уровне ширина рецептивного поля равна 4, в то время, как у свёртки на первом уровне рецептивное поле равно 3).

$k \cdot S$



- хорошо учатся и сходятся



- хорошо учатся и сходятся



Из комментариев:

Вопрос:

Еще такой вопрос по поводу сверток авторам курса или тем, кто может подсказать. Так как в картиночных задачах каждый элемент входного вектора имеет интерпретацию, то можно сделать такие любопытные вещи как, обучить сеть, взять значение промежуточного слоя, использовав его как маску, наложить на входное изображение и получить области на исходном

изображении, которые наиболее важны для нейронной сети, а можно ли как-то сделать тоже самое, но для задач НЛП?

Ответ(Николая Капырина):

несомненно, и даже больше. В самых разных архитектурах для этого используются "слои внимания" (attention layers), и в современных архитектурах они позволяют не просто узнать, на что смотрела сеть (ради любопытства или оптимизации), но направить обработку следующих слоёв. То есть слой внимания вырабатывает **дополнительный набор признаков**, который конкатенируется или прямо домножается на **векторное представление** (слова, фразы, параграфа), по которому следующие слои учатся обрабатывать текст согласно заданию.

Следующие главы как раз про механизмы внимания. 😊

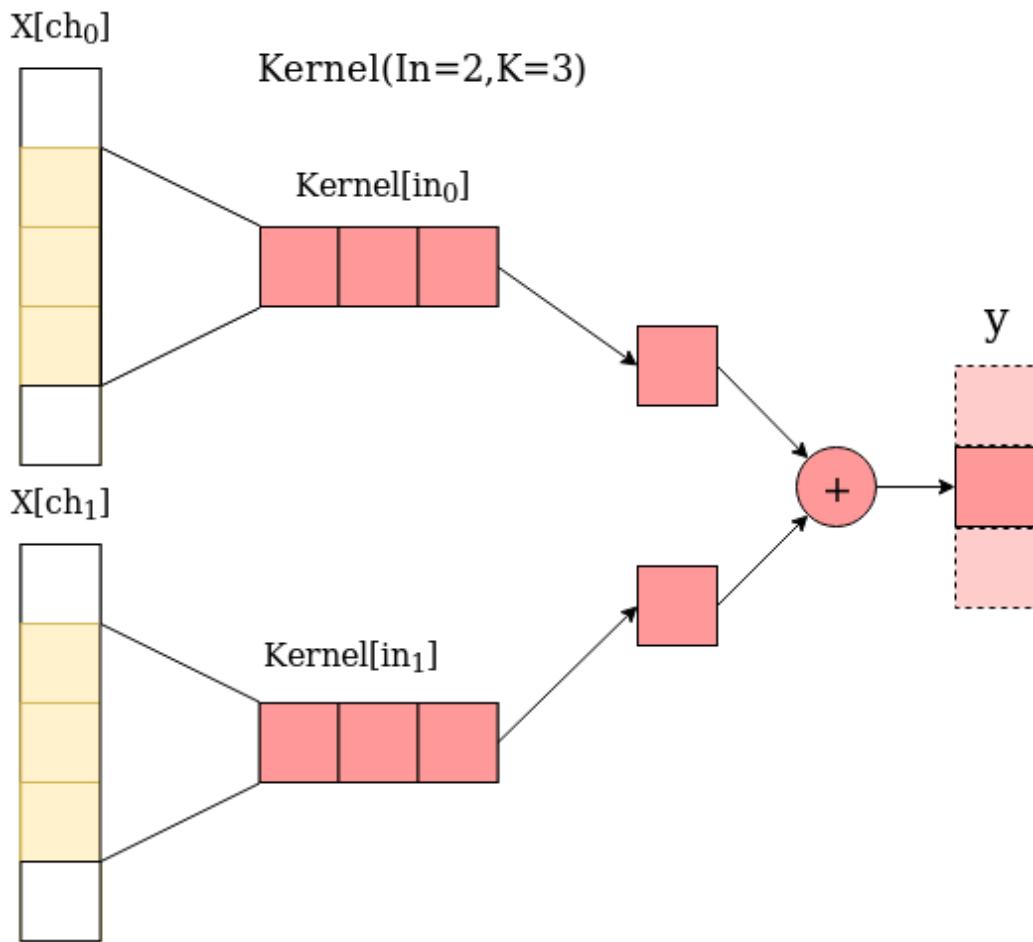
Вопрос:

Все же свертка в задачах НЛП не такая очевидная, как в картиночных задачах. Ведь в картиночных задачах каждый элемент вектора имеет физический смысл вида, яркости пикселя в RGB пространстве (или другом), а в задачах НЛП же, значение вектора это уже некоторая абстракция, которая напрямую никак не сопоставляется с каким-то словом или символом.

Ответ(Николая Капырина):

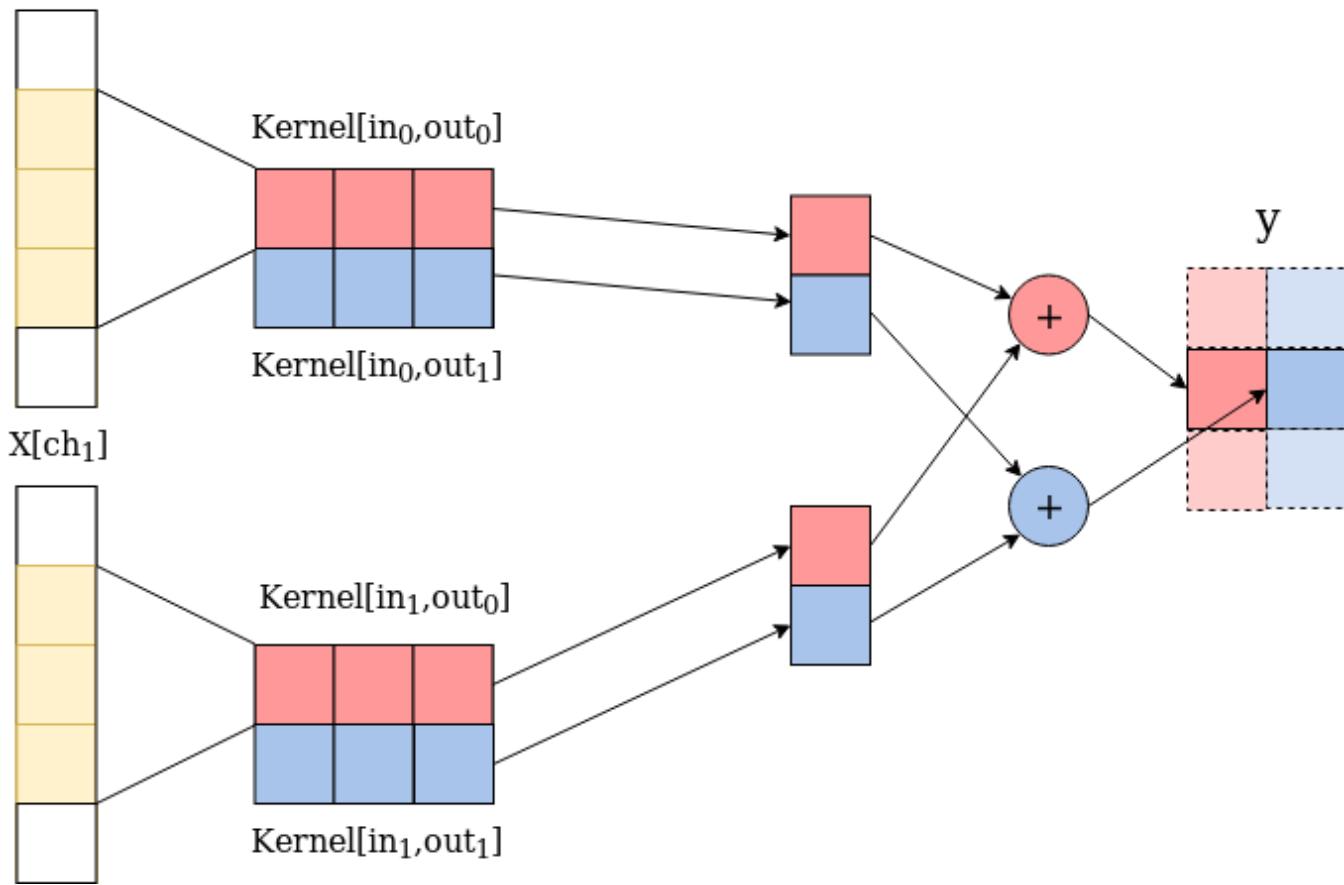
промежуточное векторное описание до каких-то пор тоже не очень специфично отражает смысл слова. До тех пор, пока не появился word2vec и стало возможным составлять смысловые пропорции типа "огурец минус зелёный плюс фиолетовый"... Свёртка просто делает что-то с числами, векторным представлением слов/предложений/параграфов, и если представление построено согласно адекватной модели языка, состав масок свёртки может означать что-то интересное.

Из практических задач (одноканальная свертка):



Модифицированный пример из прошлого шага (многоканальная свертка):

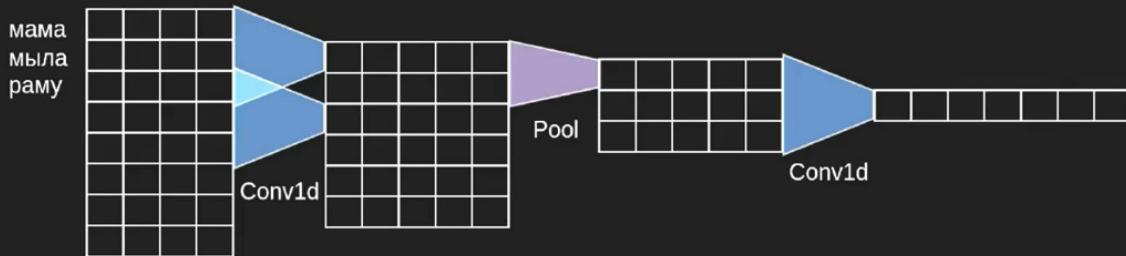
$X[ch_0]$ Kernel($In=2, Out=2, K=3$)



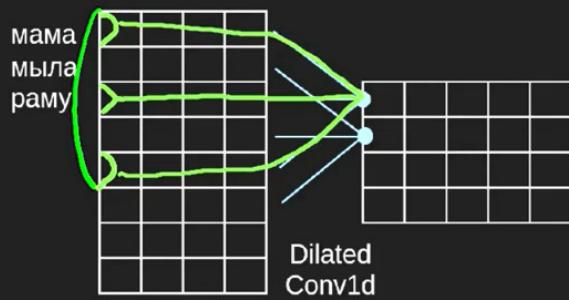
Как правило, [свёрточные нейросети](#) хорошо обучаются и сходятся. Подробнее об этой проблеме мы поговорим в лекции про [рекуррентные нейросети](#). Такие архитектуры хорошо подходят для распознавания коротких локальных особенностей, например — словосочетаний или коротких фраз. Заставить свёрточную нейросеть обрабатывать длинные предложения и учитывать широкий контекст достаточно сложно — для этого необходимо сделать очень глубокую сеть, в которой неминуемо будет много параметров, что сильно усложнит процесс обучения. Чтобы расширить пятно восприятия, можно применять "разреженные свёртки" (английский термин — "[dilated convolutions](#)"). Идея в том, что ядро свёртки применяется не к непрерывному фрагменту сигнала, а к фрагменту, из которого удалена часть элементов (как правило, удаляют элементы с чётными номерами). Таким образом почти в два раза увеличивается рецептивное поле, а количество параметров остаётся прежним. Если применять разреженные свёртки на первом слое, мы, фактически, будем игнорировать каждое второе слово, это совсем не то что мы хотим. Поэтому на первом слое применяют обычные, не прореженные свёртки, а затем на каждом новом уровне увеличивают прореживания в два раза. Таким образом, мы можем сэкономить количество параметров, увеличив рецептивное поле. Как вы могли заметить, для вычисления свёртки нам требуются элементы не только слева, но и справа. А что, если мы решаем задачу генерации текста, выдавая слово за словом, и справа от текущей позиции ничего нет? Такой подход, кстати, называется авторегрессией.

Дополнительные комментарии Романа Суворова к видео:

- В свёрточных нейросетях проблема затухания градиента тоже есть. Однако она проявляется не с ростом длины входной последовательности, а с увеличением глубины (то есть количества слоёв). Самые распространённые методы борьбы с затуханием градиента - частично-линейные функции активации (например, LeakyReLU), блоки со связями в обход нелинейностей (skip connections, residual blocks), а также нормализация (например, BatchNorm).

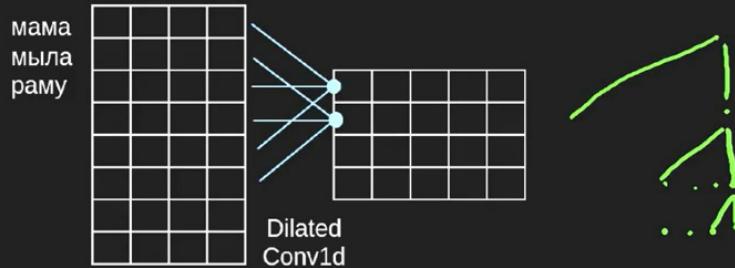


- хорошо учатся и сходятся
- нет проблемы затухающего градиента
- подходят для распознавания локальных паттернов, словосочетаний, коротких фраз
- слабо подходят для учета широкого контекста

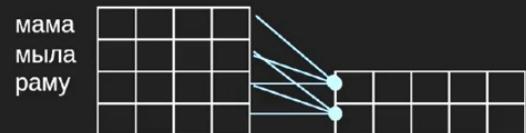


- могут учитывать более широкий контекст без увеличения числа параметров





- могут учитывать более широкий контекст без увеличения числа параметров
- шаг прореживания увеличивается с глубиной
- первый слой без прореживания, затем на каждом слое шаг увеличивается в 2 раза



- применимы в задачах генерации текста, так как учитывают только предыдущие элементы



Из комментариев к практическому заданию:

Есть очень хорошая формула, чтобы определить сколько слов получается на выходе из свёрточного слоя:

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding} - \text{dilation} \times (\text{kernel_size} - 1) - 1}{\text{stride}} + 1 \right\rfloor$$

где:

Win - количество слов на входе в свертку

Padding - это когда слева и справа добавляются фиктивные слова (Например если Padding = 2, то слева и справа добавляются по два фиктивных слова)

Dilation - собственно прореживание

Kernel_size - размер ядра свёртки

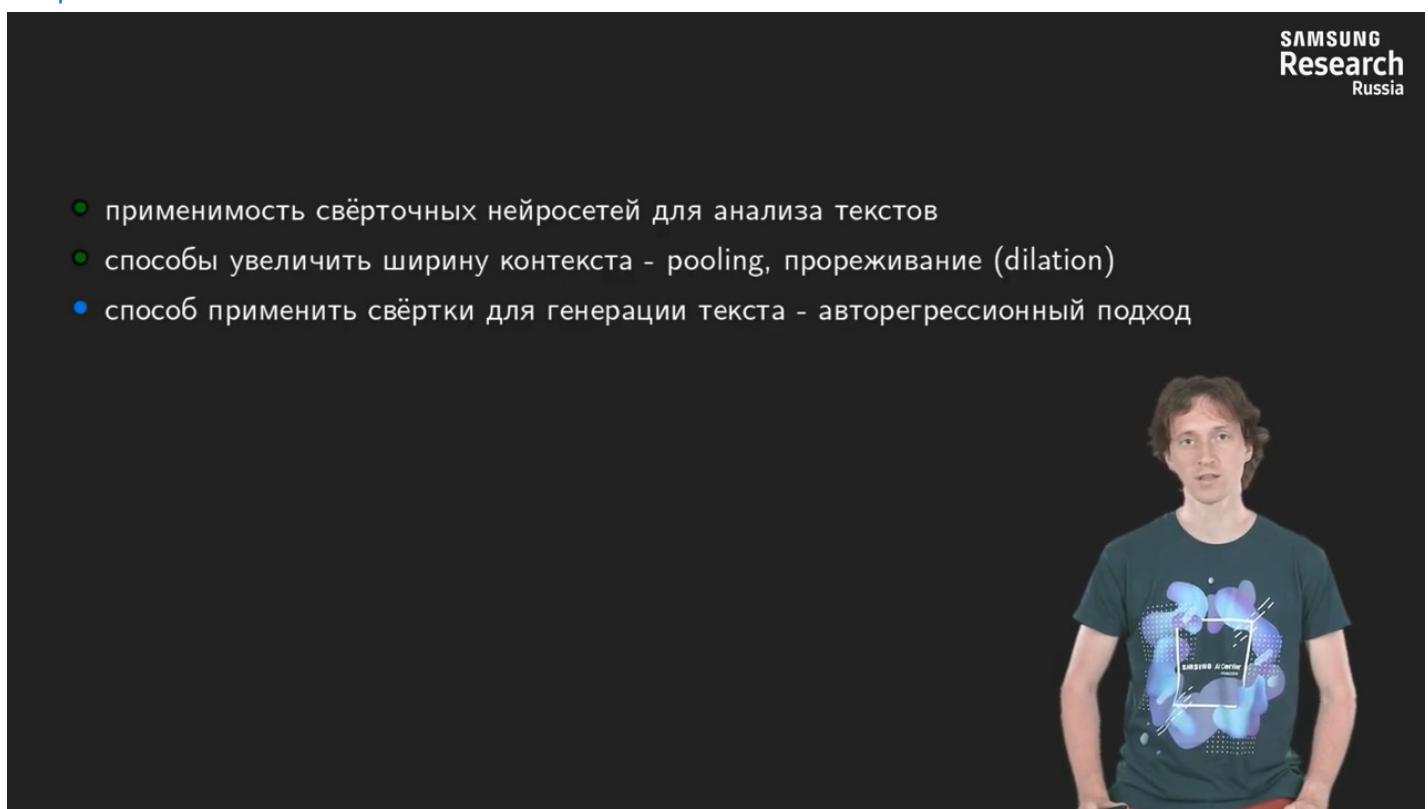
StrideStrideStride - шаг свертки

Скобки [и] означают взятие целой части. То есть если вдруг число получилось дробное, то округляем его до целого в меньшую сторону.

В этом видео мы поговорили о том, что такое свёртки, [свёрточные нейросети](#), и обсудили область применимости таких блоков для анализа текстов. Одна из проблем свёрток заключается в том, что максимальная ширина паттерна привязана к количеству параметров нейросети. Это не очень удобно. И увеличить рецептивное поле можно с помощью блоков агрегации или [пулинга](#), а также с помощью прореживания. А ещё мы упомянули авторегрессионные модели — что-то среднее между свёрточными и [рекуррентными нейросетями](#).

SAMSUNG
Research
Russia

- применимость свёрточных нейросетей для анализа текстов
- способы увеличить ширину контекста - pooling, прореживание (dilation)
- способ применить свёртки для генерации текста - авторегрессионный подход



Из комментариев (!!!ПОЛЕЗНО!!!):

Вопрос:

Редирект вопроса из телеграммской конфы(кто еще не там, присоединяйтесь!

https://t.me/stepik_nnnlp_unofficial): Из курса не смог для себя сформулировать то, а каким

собственно образом подбирается архитектура сети.

Допустим, есть задача классификации текстов - как я должен понять, какое кол-во слоев наиболее оптимально использовать, от каких характеристик корпуса отталкиваться: средняя длина текстов по корпусу быть может или еще что-то?

Вообще, как вопрос архитектуры решают супер скилловые парни и дамы?)

Несколько раз пересматривал ту часть курса где рассказывалось о понятии receptive field, но вопрос указанный выше мне это не помогло разрешить 😞

Ответ (Романа Суворова):

спасибо за вопрос! Его действительно стоило бы вынести в какое-нибудь видео в конце курса. Попробую сформулировать свой подход:

1. Если для этой задачи уже кто-то что то применял, надо взять основные идеи к себе (опираться на baseline, как сказал Николай). Читать статьи и чужой код в поисках идей.
2. Постараться выяснить, какие особенности языка могут отвечать за решение задачи (это отдельные фразы или лексический состав в целом или что-то ещё, насколько далёкие связи нужно уметь улавливать). Нужно постараться заложить своё понимание механизма (как это происходит в реальном мире) в архитектуру (модель реального мира). Например, если целевые метки сильно коррелируют с общим составом словаря, не нужно пытаться городить свёртки с большим receptive field, и вообще не надо сильно усложнять архитектуру - линейная модель + fasttext должны зарешать (если речь о классификации). Наоборот, если важен контекст, нужно постараться максимально быстро вырастить рецептивное поле за наименьшее число слоёв. Закладывание знаний и ограничений в модель ещё называется *inductive bias* - модели будет проще обобщаться, если данных не сильно много.
3. Когда сделал вариант архитектуры, добиться, чтобы она начала переобучаться на ограниченном количестве данных (например, на одном батче). Когда она сможет переобучиться - значит при большем количестве данных она сможет и обобщиться. Если модель не переобучается на одном батче, значит она вообще не учится.
4. Сравнивать train loss и validation loss - если на трейне лосс сильно меньше, то мы переобучаемся, надо облегчать модель (добавить дропаут, уменьшить количество каналов в свёртках, добавить аугментаций). Если на трейне и валидации лосс одинаковый, то мы недообучаемся, надо усилить модель. Если на валидации лосс сильно меньше, то скорее всего валидационный датасет либо проще, либо валидация вообще нерепрезентативная - надо внимательно на неё посмотреть - чудес не бывает. Если датасет маленький, важно использовать кросс-валидацию или усреднение по нескольким случайным разбивкам на трейн и тест.
5. Если хочешь увеличить количество слоёв, используй residual connections (в курсе они ещё называются "связи в обход нелинейностей") - это сильно упрощает сходимость.
6. По моему опыту свёртки проще заставить работать, чем рекуррентки, поэтому я начинаю с них обычно.

7. Можно комбинировать разные виды архитектур - сначала сделать несколько слоёв свёрток, потом добавить рекуррентку, сверху добавить crf (если задача типа ner).
8. Сначала стоит добиваться наилучшего качества любыми средствами - прикручивая сколь угодно тяжелые модели, берты, трансформеры и т.п. Когда качество устраивает - можно переключиться на сжатие полученной модели.
9. Смотреть глазами на ошибки, генерировать гипотезы насчёт "почему эта модель совершают именно эти ошибки", вносить изменения в архитектуру и проверять их экспериментально - не надо слишком много теоретизировать.
10. Ну и в целом идти от простого к сложному, за исключением случаев, когда точно по опыту знаешь, что простое не сработает. Когда есть мысль что-то добавить в архитектуру, спрашивать себя "можно ли без этого" или "есть ли что-то более простое, что можно попробовать".

Понятно, что не алгоритм (если бы он был, у нас бы не было работы), но какие-то накопленные за годы принципы.

...

кстати, Майкрософт завёз крутой сборник решений разных задач по NLP <https://github.com/microsoft/nlp-recipes>. Можно получить очень много опыта от изучения API и исходников.

3.7 Семинар: POS-тэггинг свёрточными нейросетями

#####

Предисловие из комментария (Романа Суворова):

Для данного семинара Вам потребуется ноутбук [task3_cnn_postag.ipynb](#).

Чтобы запустить ноутбук с семинара на своем компьютере:

1) Склонируйте [репозиторий курса](#):

```
git clone https://github.com/Samsung-IT-Academy/stepik-dl-nlp.git
```

2) В терминале выполните команду:

```
pip install -r requirements.txt
```

3) Запустите ноутбук:

```
ipython notebook
```

Чтобы запустить ноутбук на [Google Colab](#):

1) Скачайте ноутбук (вкладка Github, затем прописываете адрес репозитория):

The screenshot shows a GitHub search interface. At the top, there are tabs for 'Examples', 'Recent', 'Google Drive', 'GitHub', and 'Upload'. Below the tabs is a search bar with the placeholder 'Enter a GitHub URL or search by organization or user'. To the right of the search bar is a checkbox labeled 'Include private repos' and a magnifying glass icon. The search results show a repository 'Samsung-IT-Academy/stepik-dl-nlp' with the branch 'master' selected. A file named 'task1_20newsgroups.ipynb' is listed in the results. There are also icons for a refresh button and a copy button.

- 2) Запустите ноутбук.
3) Не забудьте выполнить команду `git clone` из первой (закомментированной) ячейки, чтобы выкачать на colab библиотеку `dlnlutils`
Ноутбуки также работают и на Kaggle (следуйте комментариям в ячейках ноутбука).
Ссылка на репозиторий со всеми материалами курса и инструкцией по запуску: <https://github.com/Samsung-IT-Academy/stepik-dl-nlp>

Всем привет! Сегодня мы попробуем на практике разобраться, как применять [свёрточные нейросети](#) к задачам обработки текста. Разбираться мы будем на примере задачи определения частей речи слов, по-английски она называется "[POS-tagging](#)" (part of speech tagging). Зачем эта задача нужна и в чём сложности — мы увидим в процессе семинара. Ну что ж, поехали!

Вначале мы импортируем нужные нам библиотеки — это стандартный набор: scikit-learn, torch, наша библиотечка для курса (dlnlutils), а также библиотечка pyconll — она нам потребуется для загрузки [корпуса](#). Итак, первый шаг — это скачать данные: обучающую и тестовую выборку. В этом семинаре мы будем использовать размеченный корпус, который называется "SynTag Rus". Этот корпус был размечен руками, лингвистами, и в нём содержится разметка по частям речи, по нормальным формам слов, синтаксическая разметка. Он предназначен для того, чтобы настраивать и проверять методы лингвистического анализа текстов, а именно — морфологического разбора и синтаксического разбора. Этот корпус выложен в открытый доступ на github, мы просто скачиваем. Далее, мы загружаем эти файлы. Разметка в этом корпусе представлена в [формате CoNLL](#) — это достаточно распространённый формат для того, чтобы хранить аннотированные деревья и разную лингвистическую разметку. Он используется не только в этом корпусе — он достаточно популярный. Давайте возьмём пару первых предложений и посмотрим, как выглядит разметка для задачи определения частей речи слов. На экране вы видите два предложения вместе с настоящими тэгами частей речи. Предложения разделены пустой строкой, то есть первое предложение — это, по сути, просто заголовок:

"анкета", "точка". И мы видим, что "анкета" — это существительное, а "точка" получила тэг "PUNCT", то есть — пунктуация. Второе предложение гораздо длиннее, и в нём для каждого токена проставлены тэги частей речи. Давайте посчитаем пару статистик по нашему корпусу. В первую очередь нас интересует наибольшая длина предложения и наибольшая длина токена. Также, давайте выведем несколько первых предложений для того, чтобы посмотреть, правильно ли загрузились данные, и вообще — понять, какого они характера. Решать задачу определения части речи мы будем с помощью свёрточных нейросетей, причём будем использовать нейросети, которые принимают на вход номера отдельных символов — то есть они работают не на уровне целых токенов а на уровне символов. Это вполне оправдано, потому что часть речи во многом определяется структурой слова, наличием суффиксов, окончаний определённого вида... Поэтому, если бы мы работали на уровне отдельных токенов, то мы бы просто не могли анализировать структуру слов, нам бы приходилось просто запоминать, что такое-то слово — это существительное, другое слово это просто глагол... Поэтому следующий этап обработки корпуса — это построение словаря символов.

Свёрточные нейросети и POS-теггинг

POS-теггинг - определение частей речи (снятие частеречной неоднозначности)

```
In [1]: %load_ext autoreload
%autoreload 2

import warnings
warnings.filterwarnings('ignore')

from sklearn.metrics import classification_report

import numpy as np

import pyconll

import torch
from torch import nn
from torch.nn import functional as F
from torch.utils.data import TensorDataset

import dnlputils
from dnlputils.data import tokenize_corpus, build_vocabulary, \
    character_tokenize, pos_corpus_to_tensor, POSTagger
from dnlputils.pipeline import train_eval_loop, predict_with_model, init_random_seed

init_random_seed()
```

Загрузка текстов и разбиение на обучающую и тестовую подвыборки



Загрузка текстов и разбиение на обучающую и тестовую подвыборки

```
In [2]: # !wget -O ./data https://raw.githubusercontent.com/UniversalDependencies/UD_Russian-SyntagRus/master/ru_syntagrus-ud-1.0.conllu
# !wget -O ./data https://raw.githubusercontent.com/UniversalDependencies/UD_Russian-SyntagRus/master/ru_syntagrus-ud-2.0.conllu
```

```
In [3]: full_train = pyconll.load_from_file('./data/ru_syntagrus-ud-train.conllu')
full_test = pyconll.load_from_file('./data/ru_syntagrus-ud-dev.conllu')
```

```
In [4]: for sent in full_train[2:]:
    for token in sent:
        print(token.form, token.upos)
    print()
```

```
In [5]: MAX_SENT_LEN = max(len(sent) for sent in full_train)
MAX_ORIG_TOKEN_LEN = max(len(token.form) for sent in full_train for token in sent)
print('Наибольшая длина предложения', MAX_SENT_LEN)
print('Наибольшая длина токена', MAX_ORIG_TOKEN_LEN)
```

```
In [6]: all_train_texts = [' '.join(token.form for token in sent) for sent in full_train]
print('\n'.join(all_train_texts[:10]))
```

```
In [7]: train_char_tokenized = tokenize_corpus(all_train_texts, tokenizer=character_tokenize)
char_vocab, word_doc_freq = build_vocabulary(train_char_tokenized,
```



```
In [4]: for sent in full_train[:2]:
    for token in sent:
        print(token.form, token.upos)
    print()
```

Анкета NOUN ←
PUNCT ←

Начальник NOUN
областного ADJ
управления NOUN
связи NOUN
Семен PROPN
Еремеевич PROPN
был AUX
человек NOUN
простой ADJ
, PUNCT
приходил VERB
на ADP
работу NOUN
всегда ADV
вовремя ADV
, PUNCT
здравился VERB
с ADP
секретаршей NOUN
за ADP
руки NOUN
и CCONJ
иногда ADV
даже PART
писал VERB
в ADP



```
In [5]: MAX_SENT_LEN = max(len(sent) for sent in full_train)
MAX_ORIG_TOKEN_LEN = max(len(token.form) for sent in full_train for token in sent)
print('Наибольшая длина предложения', MAX_SENT_LEN)
print('Наибольшая длина токена', MAX_ORIG_TOKEN_LEN)
```

Наибольшая длина предложения 205 ←
Наибольшая длина токена 47

```
In [6]: all_train_texts = [' '.join(token.form for token in sent) for sent in full_train]
print('\n'.join(all_train_texts[:10]))
```

Анкета .
Начальник областного управления связи Семен Еремеевич был человек простой , приходил на работу всегда вовремя , здоровался с секретаршой за руку и иногда даже писал в стенгазету заметки под псевдонимом " Муха " .
В приемной его с утра ожидали посетители , - кое-кто с важными делами , а кое-кто и с такими , которые легко можно было решить в нижестоящих инстанциях , не затрудняя Семена Еремеевича .
Однако стиль работы Семена Еремеевича заключался в том , чтобы принимать всех желающих и лично вникать в дела .
Приемная была обставлена просто , но по-деловому .
У двери стоял стол секретарши , на столе - пишущая машинка с широкой кареткой .
В углу висел репродуктор и играло радио для развлечения ожидающих и еще для того , чтобы заглушать голос , доносившийся из кабинета , так как , бессспорно , среди посетителей могли находиться и случайные люди .
Кабинет отличался скромностью , присущей Семену Еремеевичу .
В глубине стоял широкий письменный стол с бронзовыми чернильницами и перед ним два кожаных кресла .
Справа был стол для заседаний - длинный , накрытый зеленым сукном и с обеих сторон аккуратно заставленны

```
In [7]: train_char_tokenized = tokenize_corpus(all_train_texts, tokenizer=character_tokenize)
char_vocab, word_doc_freq = build_vocabulary(train_char_tokenized,
                                              max_doc_freq=1.0, min_count=5, pad_word='<PAD>')
print("Количество уникальных символов", len(char_vocab))
print(list(char_vocab.items())[:10])
```



```
In [7]: train_char_tokenized = tokenize_corpus(all_train_texts, tokenizer=character_tokenize)
char_vocab, word_doc_freq = build_vocabulary(train_char_tokenized,
                                              max_doc_freq=1.0, min_count=5, pad_word='<PAD>')
print("Количество уникальных символов", len(char_vocab))
print(list(char_vocab.items())[:10])
Количество уникальных символов 150
[('<PAD>', 0), (' ', 1), ('o', 2), ('e', 3), ('a', 4), ('т', 5), ('и', 6), ('н', 7), ('.', 8), ('с', 9)]
```

```
In [8]: UNIQUE_TAGS = ['<NOTAG>'] + sorted({token.upos for sent in full_train for token in sent if token.upos})
label2id = {label: i for i, label in enumerate(UNIQUE_TAGS)}
label2id
```

```
In [9]: train_inputs, train_labels = pos_corpus_to_tensor(full_train, char_vocab, label2id,
                                                       MAX_SENT_LEN, MAX_ORIG_TOKEN_LEN)
train_dataset = TensorDataset(train_inputs, train_labels)

test_inputs, test_labels = pos_corpus_to_tensor(full_test, char_vocab, label2id,
                                                MAX_SENT_LEN, MAX_ORIG_TOKEN_LEN)
test_dataset = TensorDataset(test_inputs, test_labels)
```

```
In [10]: train_inputs[1][:5]
```

```
In [11]: train_labels[1]
```

Вспомогательная свёрточная архитектура



Словарь символов — это "dict" питоновский, который отображает подстроку, содержащую единственный элемент (это сам символ) в номер этого символа. Таким образом мы можем каждый токен представить как список чисел. Также нам опять пригодится фиктивный символ, который будет означать отсутствие символа. Он будет использоваться для того, чтобы уравнять длины всех токенов и всех предложений. Как мы видим, всего у нас в [корпусе](#) 150 уникальных символов. Самый частотный символ — это пробел, затем идут гласные. Аналогичную процедуру проделаем и с метками частей речи. Для того, чтобы составить обучающую выборку, нам нужно преобразовать строковые метки частей речи в их номера. Как мы видим, всего в корпусе 17 частей речи и мы добавляем сюда фиктивную "часть речи" — опять же, для того, чтобы выравнивать длины предложений. Что же значат эти метки частей речи? "VERB" — это глагол, "PROPN" — это имя собственное, "NUM" — это числительное, "NOUN" — это существительное... Ну, и наконец, мы перекладываем весь исходный корпус в специальные структуры, для того, чтобы подавать их в наш цикл для обучения. То есть мы перекладываем их в pytorch Dataset. Мы используем стандартный TensorDataset, он принимает на вход список [тензоров](#), то есть, для того чтобы использовать этот Dataset, нам нужно подготовить эти тензоры. Первый — это тензор идентификаторов символов, а второй — это идентификаторы меток частей речи. Давайте рассмотрим поподробнее, как именно мы составляем эти тензоры. Вот — эта функция, которая нас интересует — она на вход получает список токенизованных предложений. На самом деле, это не просто список токенизованных предложений, а это результаты разбора исходного корпуса в [формате CoNLL](#), то есть в нём токены имеют дополнительную информацию, а именно — информацию о частях речи. То есть эта функция

предназначена для преобразования обучающей и тестовой выборки в 2 тензора тензора — тензор входных идентификаторов символов и тензор выходных идентификаторов тэгов. Кроме списка исходных предложений нам нужно отображение из символов в номера символов, а также отображение из меток в номера; нам нужна оценка максимальной длины предложения и максимальной длины токена в символах. Создаём заготовки для тензоров — сначала мы эти тензоры инициализируем нулями. Тензор номеров символов имеет следующую размерность: [количество предложений (всего в обучающей выборке) на количество токенов в предложений на максимальную длину токена]. Но, кроме этого, мы добавляем ещё 2 дополнительных колоночки к каждому токену. Эти колоночки дополнительные мы будем заполнять нулями — они нам нужны для того, чтобы указать нейросети, что определённая [N-грамма](#) символов встречается именно в начале токена или именно в конце токена, но не в середине. То есть это нужно для того, чтобы нейросеть умела отличать начало слова и конец слова от середины слова.

```
In [7]: train_char_tokenized = tokenize_corpus(all_train_texts, tokenizer=character_tokenize)
char_vocab, char_doc_freq = build_vocabulary(train_char_tokenized,
                                              max_doc_freq=1.0, min_count=5, pad_word='<PAD>')
print("Количество уникальных символов", len(char_vocab))
print(list(char_vocab.items())[:10])
Количество уникальных символов 150
[('<PAD>', 0), (' ', 1), ('o', 2), ('e', 3), ('a', 4), ('т', 5), ('и', 6), ('н', 7), ('.', 8), ('с', 9)]
```

```
In [8]: UNIQUE_TAGS = ['<NOTAG>'] + sorted({token.upos for sent in full_train for token in sent if token.upos})
label2id = {label: i for i, label in enumerate(UNIQUE_TAGS)}
label2id
```

```
In [9]: train_inputs, train_labels = pos_corpus_to_tensor(full_train, char_vocab, label2id,
                                                       MAX_SENT_LEN, MAX_ORIG_TOKEN_LEN)
train_dataset = TensorDataset(train_inputs, train_labels)

test_inputs, test_labels = pos_corpus_to_tensor(full_test, char_vocab, label2id,
                                                MAX_SENT_LEN, MAX_ORIG_TOKEN_LEN)
test_dataset = TensorDataset(test_inputs, test_labels)
```

```
In [10]: train_inputs[1][:5]
```

```
In [11]: train_labels[1]
```

Вспомогательная свёрточная архитектура



```
[('<PAD>', 0), (' ', 1), ('o', 2), ('e', 3), ('a', 4), ('т', 5), ('и', 6), ('н', 7), ('.', 8), ('с', 9)]
```

```
In [8]: UNIQUE_TAGS = ['<NOTAG>'] + sorted({token.upos for sent in full_train for token in sent if token.upos})
label2id = {label: i for i, label in enumerate(UNIQUE_TAGS)}
label2id
```

```
Out[8]: {'<NOTAG>': 0,
         'ADJ': 1,
         'ADP': 2,
         'ADV': 3,
         'AUX': 4,
         'CCONJ': 5,
         'DET': 6,
         'INTJ': 7,
         'NOUN': 8,
         'NUM': 9,
         'PART': 10,
         'PRON': 11,
         'PROPN': 12,
         'PUNCT': 13,
         'SCONJ': 14,
         'SYM': 15,
         'VERB': 16,
         'X': 17}
```

```
In [9]: train_inputs, train_labels = pos_corpus_to_tensor(full_train, char_vocab, label2id,
                                                       MAX_SENT_LEN, MAX_ORIG_TOKEN_LEN)
train_dataset = TensorDataset(train_inputs, train_labels)

test_inputs, test_labels = pos_corpus_to_tensor(full_test, char_vocab, label2id,
                                                MAX_SENT_LEN, MAX_ORIG_TOKEN_LEN)
test_dataset = TensorDataset(test_inputs, test_labels)
```



```
In [9]: train_inputs, train_labels = pos_corpus_to_tensor(full_train, char_vocab, label2id,
                                                    MAX_SENT_LEN, MAX_ORIG_TOKEN_LEN)
train_dataset = TensorDataset(train_inputs, train_labels)

test_inputs, test_labels = pos_corpus_to_tensor(full_test, char_vocab, label2id,
                                                MAX_SENT_LEN, MAX_ORIG_TOKEN_LEN)
test_dataset = TensorDataset(test_inputs, test_labels)
```

```
In [10]: train_inputs[1][:5]
```

...

```
In [11]: train_labels[1]
```

...

Вспомогательная свёрточная архитектура

```
In [12]: class StackedConv1d(nn.Module):
    def __init__(self, features_num, layers_n=1, kernel_size=3, conv_layer=nn.Conv1d, dropout=0.0):
        super().__init__()
        layers = []
        for _ in range(layers_n):
            layers.append(nn.Sequential(
                conv_layer(features_num, features_num, kernel_size, padding=kernel_size//2),
                nn.Dropout(dropout),
                nn.LeakyReLU()))
        self.layers = nn.ModuleList(layers)

    def forward(self, x):
        """x - BatchSize x FeaturesNum x SequenceLen"""

```



```
import torch
from torch.utils.data import TensorDataset

from dlnputils.pipeline import predict_with_model
from .base import tokenize_corpus

def pos_corpus_to_tensor(sentences, char2id, label2id, max_sent_len, max_token_len):
    inputs = torch.zeros((len(sentences), max_sent_len, max_token_len * 2), dtype=torch.long)
    targets = torch.zeros((len(sentences), max_sent_len), dtype=torch.long)

    for sent_i, sent in enumerate(sentences):
        for token_i, token in enumerate(sent):
            targets[sent_i, token_i] = label2id.get(token.upos, 0)
            for char_i, char in enumerate(token.form):
                inputs[sent_i, token_i, char_i + 1] = char2id.get(char, 0)

    return inputs, targets
```



Из комментариев:

Вопрос:

И всё-таки, зачем нам добавлять в начало 0, если в данном случае наша нейронная сеть всегда принимает целые слова, а не n-граммы символов, и, соответственно, первый символ всегда будет соответствовать началу слова?

Ответ (Романа Суворова):

замечание очень правильное!

Не смотря на то, что вся нейросеть видит только целые слова, из-за того, что она свёрточная, она на самом деле работает с n-граммами символов. Представьте первый слой такой сети - это свёртка с окном 3, то есть на одну выходную позицию влияет 3 рядом стоящих символа.

Допустим, нам дают на вход текст "косил или покос" (не ищите смысл!). В этом тексте есть две биграммы, которые выступают как в качестве окончания - "ил" и "ос" - так и в середине слова. Окончания очень важны для определения частей речи в русском языке, поэтому важно отличать случаи, когда мы видим какую-то n-грамму в начале, середине или конце слова.

Доп. комментарий (студента):

наверное, объяснение было бы понятнее, если бы мы развили пример текста до представления предложения как списка токенов без пробелов ['косил', 'или', 'покос'], т.е. после токенизатора пробелы убираются. Для этого и нужны вышеописанные нули в начале, конце слов: чтоб ядро, ходящее по n символам, понимало что оно на границе слова.

В остальном, процедура заполнения [тензоров](#) достаточно простая — мы просто итерируемся по всем предложениям, по всем токенам в предложении, и для каждого токена кладём в соответствующую ячейку тензора меток идентификатор метки, а также кладём в соответствующую ячейку входного тензора идентификатор очередного символа. Ну что ж, тензоры мы построили, положили их датасет, давайте теперь посмотрим, как данные будут выглядеть для нейросети во время обучения. Для этого давайте выведем на экран фрагмент тензора, представляющего второе предложение из обучающей выборки — первые пять слов этого предложения. Мы видим прямоугольный тензор, каждая строчка этого тензора представляет один токен и содержит номера символов, которые в этом токене используются, в том порядке, в котором они встречались в самом токене. Обратить внимание здесь нужно на две вещи — первая: это стартовый нолик, он нам нужен для того, чтобы указать нейросети, что это — начало токена. Всё, что после последнего символа — заполняется нулями. Количество значимых элементов в каждой такой строчке у нас отличается. У нас есть короткие и длинные токены. Так выглядит входной тензор. Давайте теперь посмотрим, как выглядит тензор меток. Целевой тензор для этого же предложения — мы выбрали второе предложение из обучающей выборки, и это уже не двухмерный тензор, это одномерный тензор, то есть это просто список чисел, каждое число представляет номер тэга соответствующего токена. Для первого токена из этого предложения мы должны будем предсказать класс номер "8", для 2 — класс номер "1". Для токенов фиктивных, ненастоящих, мы всегда должны будем предсказывать "0". Напомню, что фиктивные токены у нас не являются частью исходного предложения, они используются для того, чтобы выровнять длины всех предложений и иметь возможность упаковывать предложения разной длины в прямоугольный тензор и обрабатывать в пакетном режиме в нейросети. Это нужно для того, чтобы эффективно использовать возможности современных вычислителей — видеокарт. Перед тем, как перейти непосредственно к решению нашей задачи, давайте определим вспомогательный нейросетевой модуль. Он состоит из свёрток,

функции активации и дропаута. Давайте сначала посмотрим на функцию "forward", то есть на то, как этот модуль работает. Здесь мы видим, что в основе модуля лежит набор блоков — одинаковых блоков. Давайте попробуем изобразить этот модуль графически. У нас в начале есть "x", он подаётся в некий блок "layer 1", затем выход "layer 1" приplusplusывается к его же входу. Это всё подаётся в следующий — такой же блок "layer 2" и снова приplusplusывается... наверное, вам это что-то напоминает. Это простенький [ResNet](#). Зачем это нужно?

Использование вот этих связей (skip connection) ускоряет сходимость, а также позволяет нам сделать нейросеть более глубокой. Другими словами, без "skip connection" мы можем сделать нейросеть максимум из 5...9 блоков глубиной, но, используя "skip connection", мы можем делать нейросеть произвольной глубины, она по-прежнему будет обучаться. Давайте теперь посмотрим, как же устроен каждый из вот этих блоков. Каждый из этих блоков реализуется с помощью модуля pytorch "sequential" — это базовый модуль, который берёт список других модулей и выполняет их по-очереди, передавая результат первого во второй, из второго в третий, и так далее. В этом блоке первый слой — это свёрточный слой. Это одномерные свёртки — для текстов чаще всего используются одномерные свёртки. По умолчанию, мы говорим, что размер ядра равен 3. При этом, свёрточный слой не меняет количества каналов — он принимает одно и то же число каналов и возвращает одно и то же число каналов. Кроме того, здесь мы используем padding. Мы просим, чтобы размерность тензора вообще никак не менялась, то есть по умолчанию при нулевом padding свёртки немного сжимают тензор по пространственному измерению. Здесь мы просим, чтобы размерность тензора оставалась прежней. Для этого, перед тем, как применять свёртки, нужно добавить какое-то количество нулей в начало и в конец тензора. Реализацию свёрточного модуля мы рассмотрим попозже. Второй слой в нашем блоке — это dropout. Он нужен для того, чтобы нейросеть меньше [переобучалась](#). В режиме обучения dropout зануляет случайные ячейки тензора. Когда нейросеть обучена, dropout ничего не делает. Третий слой нашего блока — это функция активации. Здесь мы используем Leaky ReLU, часто это — неплохой выбор. Таким образом, мы определили достаточно универсальный свёрточный модуль, который можно использовать в абсолютно разных ситуациях и, в рамках этого семинара, мы будем использовать его, как минимум, в двух разных случаях. Давайте теперь перейдём к конкретным архитектурам для определения частей речи токенов и попробуем что-нибудь уже обучить!

```
import torch
from torch.utils.data import TensorDataset

from dlnlp.utils.pipeline import predict_with_model
from .base import tokenize_corpus

def pos_corpus_to_tensor(sentences, char2id, label2id, max_sent_len, max_token_len):
    inputs = torch.zeros((len(sentences), max_sent_len, max_token_len + 2), dtype=torch.long)
    targets = torch.zeros((len(sentences), max_sent_len), dtype=torch.long)

    for sent_i, sent in enumerate(sentences):
        for token_i, token in enumerate(sent):
            targets[sent_i, token_i] = label2id.get(token.upos, 0)
            for char_i, char in enumerate(token.form):
                inputs[sent_i, token_i, char_i + 1] = char2id.get(char, 0)

    return inputs, targets

class POSTagger:
    def __init__(self, model, char2id, id2label, max_sent_len, max_token_len):
        self.model = model
        self.char2id = char2id
        self.id2label = id2label
        self.max_sent_len = max_sent_len
        self.max_token_len = max_token_len

    def __call__(self, sentences):
        tokenized_corpus = tokenize_corpus(sentences, min_token_size=1)

        inputs = torch.zeros((len(sentences), self.max_sent_len, self.max_token_len + 2), dtype=torch.long)
```



```
In [9]: train_inputs, train_labels = pos_corpus_to_tensor(full_train, char_vocab, label2id,
                                                       MAX_SENT_LEN, MAX_ORIG_TOKEN_LEN)
train_dataset = TensorDataset(train_inputs, train_labels)

test_inputs, test_labels = pos_corpus_to_tensor(full_test, char_vocab, label2id,
                                               MAX_SENT_LEN, MAX_ORIG_TOKEN_LEN)
test_dataset = TensorDataset(test_inputs, test_labels)
```

```
In [10]: train_inputs[1][:5]
```

```
In [11]: train_labels[1]
```

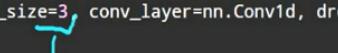


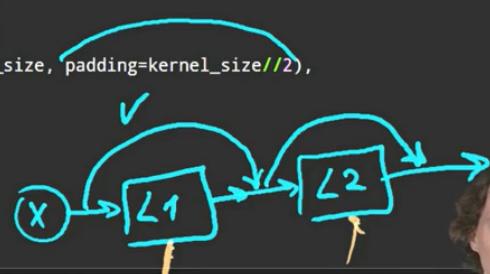


Вспомогательная свёрточная архитектура

```
In [12]: class StackedConv1d(nn.Module):
    def __init__(self, features_num, layers_n=1, kernel_size=3, conv_layer=nn.Conv1d, dropout=0.0):
        super().__init__()
        layers = []
        for _ in range(layers_n):
            layers.append(nn.Sequential(
                conv_layer(features_num, features_num, kernel_size, padding=kernel_size//2),
                nn.Dropout(dropout),
                nn.LeakyReLU()))
        self.layers = nn.ModuleList(layers)

    def forward(self, x):
        """x - BatchSize x FeaturesNum x SequenceLen"""
        for layer in self.layers:
            x = x + layer(x)
        return x
```





Предсказание частей речи на уровне отдельных токенов

```
In [13]: class SingleTokenPOSTagger(nn.Module):
    def __init__(self, vocab_size, labels_num, embedding_size=32, **kwargs):
        super().__init__()
        self.char_embeddings = nn.Embedding(vocab_size, embedding_size, padding_idx=0)
        self.backbone = StackedConv1d(embedding_size, **kwargs)
        self.global_pooling = nn.AdaptiveMaxPool1d(1)
        self.out = nn.Linear(embedding_size, labels_num)
        self.labels_num = labels_num

    def forward(self, tokens):
```

Из Комментариев:

Комментарий №1 (Алексей Сильвестров):

Надеюсь, эта визуализация будет полезна слушателям при разборе семинара:

В начале мы фиксируем максимальную длину предложения SentenceSize:

$$Batch = \begin{pmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,S} \\ w_{2,1} & w_{2,2} & \dots & w_{2,S} \\ w_{3,1} & w_{3,2} & \dots & w_{3,S} \end{pmatrix} \in \mathbb{R}^{Batch \times SentenceSize}$$

Каждое слово состоит из букв + тэги начала и конца слова, все они имеют свой embedding vector:

$$w_{1,1} = (c_{1,1} \ c_{1,2} \ \dots \ c_{1,k}) = \begin{pmatrix} [e_{1,1}] & [e_{1,2}] & \dots & [e_{1,k}] \\ [e_{2,1}] & [e_{2,2}] & \dots & [e_{2,k}] \\ \dots & \dots & \dots & \dots \\ [e_{E,1}] & [e_{E,2}] & \dots & [e_{E,k}] \end{pmatrix} \in \mathbb{R}^{EmbedSize \times WordSize}$$

Таким образом Batch представляет собой тензор размерности 4:

$$Batch = \left(\begin{pmatrix} [e_{1,1}] & [e_{1,2}] & \dots & [e_{1,k}] \\ [e_{2,1}] & [e_{2,2}] & \dots & [e_{2,k}] \\ \dots & \dots & \dots & \dots \\ [e_{E,1}] & [e_{E,2}] & \dots & [e_{E,k}] \end{pmatrix} \dots \dots \right) \in \mathbb{R}^{Batch \times SentenceSize \times EmbedSize \times WordSize}$$

Первая нейросеть принимает на вход одно слово и не учитывает его контекст (соседние слова), поэтому для нее мы переделываем батч:

1. Транспонируем embedding vectors для каждого слова
2. Один элемент батча - одно слово

$$Batch' = \left(\begin{pmatrix} [e_{1,1} \ e_{2,1} \ \dots \ e_{E,1}] \\ [e_{1,2} \ e_{2,2} \ \dots \ e_{E,2}] \\ \dots \\ [e_{1,k} \ e_{2,k} \ \dots \ e_{E,k}] \end{pmatrix} \dots \dots \right) \in \mathbb{R}^{Batch \times SentenceSize \times WordSize \times EmbedSize}$$

Комментарий №2 (Георгий Господинов):

Когда нейросеть обучена, dropout ничего не делает.

Кажется, стоит отметить момент про коррекцию весов у обученной с dropout'ом сети. В train-time нейроны присутствуют в архитектуре с вероятностью p , в test-time присутствуют все нейроны, но их веса необходимо скорректировать. Сеть "привыкла" к активациям не от всех нейронов, следовательно, веса модели нужно занизить.

Таким образом, исходящие веса умножаются на вероятность p , это иллюстрирует рисунок из оригинальной статьи(<http://jmlr.org/papers/volume15/srivastava14a.old/srivastava14a.pdf>):

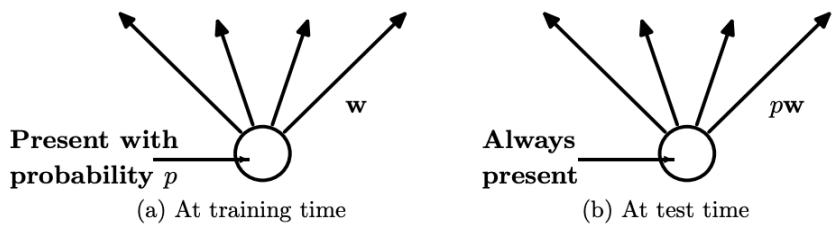


Figure 2: **Left:** A unit at training time that is present with probability p and is connected to units in the next layer with weights \mathbf{w} . **Right:** At test time, the unit is always present and the weights are multiplied by p . The output at test time is same as the expected output at training time.

Также есть хорошее обсуждение на

stackoverflow: <https://datascience.stackexchange.com/questions/44293/how-does-dropout-work-during-testing-in-neural-network>

Добавлю, что в **pytorch** перед обучением модели ее необходимо перевести в режим обучения (критически важно для слоев DropOut и BatchNorm):

```
model.train()
```

Перед оценкой качества/применением модели необходимо вызвать

```
model.eval()
```

UPD: в треде ниже есть интересная информация о том, как Dropout реализован в PyTorch.

Ответ (Романа Суворова):

очень хорошее замечание, спасибо! Идея в том, чтобы сохранить норму каждого вектора признаков (чтобы не было различий между распределением на обучении и на инференсе). Только прошу обратить внимание на важный момент: в PyTorch в модуле Dropout параметр p соответствует вероятности зануления элемента (выше p - сильнее dropout), в отличие от упомянутой статьи, где p - это вероятность сохранения элемента (см рисунок в сообщении выше - "a unit ... is present with probability p "). Поэтому в PyTorch по идеи должно происходить масштабирование не на p , а на $(1 - p)$ (но по исходному коду не нашел этого места, оно где-то в библиотеке cudnn скорее всего).

Ответ (Романа Суворова):

хотя вот покопал исходники немного ещё - и увидел, что в eval режиме [никакого масштабирования нет](#). Хотя я не уверен, что используется именно этот код.

Ответ (Романа Суворова):

новая инфа откопалась, коллеги помогли! Классически дропаут реализуется домножением активаций на $\text{binary_noise} \sim \text{Bernoulli}(p)$ (binary_noise может быть либо 0 либо 1 для каждого элемента). При этом в eval режиме надо масштабировать активации на p . Есть другой подход -

[inverted dropout](#) - когда мы домножаем активации на $\text{scaled_noise} \sim \text{Bernoulli}(p) / p$ (scaled_noise может быть либо 0, либо $1/p$). В этом случае в eval режиме активации масштабировать не надо!

Кстати, если внимательно почитать [код PyTorch](#) (чуть ниже первого места, на которое я прикреплял ссылку), то мы увидим, что там реализован как раз inverted dropout.
Кажется, теперь разобрались!

Давайте предпримем первую попытку решить нашу задачу. Для этого нам необходимо описать нейросетевой модуль для pytorch. И особенность этого модуля будет заключаться в том, что он будет предсказывать метки частей речи токенов, используя информацию, содержащуюся только в самих токенах. Другими словами, эта модель никак не будет использовать контекст, в котором слово употребляется. Опять же, давайте начнём с метода "forward". Для начала, получим переменные, представляющие форму исходного [тензора](#). На вход нам приходит трёхмерный тензор, размерности которого соответствуют [размеру батча, наибольшей длине предложения в токенах и наибольшей длине каждого токена в символах]. Далее мы склоняем первое и второе измерения, чтобы получить двухмерный тензор. Другими словами, мы забываем о том, что токены у нас были как-то объединены в предложения (для этой модели нам совершенно неважно). Далее мы используем embedding-слой для того, чтобы для каждого символа получить вектора. Таким образом, мы снова получаем трёхмерный тензор, который соответствует [количеству токенов в батче на длину каждого токена на размер вектора для символа]. Затем мы транспонируем этот тензор для того, чтобы мы могли его подать в свёрточную нейросеть. В pytorch принятая следующая конвенция о порядке размерностей: сначала идёт размер батча, затем идёт количество признаков для каждого элемента (в данном случае — это размер вектора), а затем идёт какое-то количество размерностей, соответствующих самим элементам, то есть дальше идут пространственные измерения. В текстах мы работаем с одномерными данными, то есть у нас, в данном случае это длина токена. После всех этих операций, переменная "char embeddings" содержит векторы для отдельных символов. Пока что, эти векторы не содержат информации о том, в каком контексте используется каждый символ — это просто какие-то априорные знания о том, что это вообще за символ. В начале обучения эти вектора инициализируются случайными числами. Затем эти векторы мы передаём в свёрточный модуль, для того, чтобы учесть локальный контекст — то есть ту ситуацию, в которой каждый конкретный символ используется. Для этого мы применяем "backbone"-сетку, роль которой выполняет простенький [ResNet](#), который мы определили чуть выше. В результате мы получаем трёхмерный тензор такой же размерности, кладём его в переменную features, и эта переменная содержит векторы символов уже с учётом их контекста. Тут нужно вспомнить, что эти теги нам нужно предсказывать не для каждого символа, а для каждого токена. Поэтому нам нужно как-то агрегировать признаки символов в токене, чтобы получить вектор токена. Для этого мы используем pooling — в данном случае это max pooling. Как работает max pooling? Допустим, у нас есть некоторая матричка, строки в этой

матричке представляют отдельные символы в токене, а столбцы — это признаки этих символов. И max pooling делает из этого один вектор — количество элементов в этом векторе соответствует количеству столбцов в исходной матрице, и каждый элемент получен взятием функции максимума из соответствующего столбца. В результате применения global [пулинга](#) мы получаем тензор, на этот раз — уже двумерный, каждая строчка этого тензора представляет отдельный токен. Ну и наконец, по каждому токену нам нужно принять решение касательно его метки. Для этого признаки токенов мы передаём в ещё один нейросетевой модуль, который называется out — это просто полносвязный блок. В результате применения этого блока мы получаем также двухмерный тензор, но у него уже размер строки не "embedding size", а "количество меток частей речи". Далее мы меняем форму этого тензора, преобразуем его в трёхмерный, то есть "вспоминаем" о том, что у нас есть предложения, и транспонируем для того, чтобы порядок измерений соответствовал порядку измерений в исходном тензоре "tokens". В чём же физический смысл того, что мы только что рассмотрели? Физический смысл того, что мы сейчас описали, заключается в том, чтобы рассмотреть все возможные [N-граммы](#) символов, которые встречаются в каждом токене, и по ним попробовать определить часть речи. Благодаря тому, что основная наша нейросеть (backbone) содержит "skip connections", N-граммы, которые учитываются этой нейросетью, по сути, имеют различную длину. Например, если мы используем размер ядра свёртки, равный 3, то первый блок учитывает трёхграммы, второй блок уже учитывает пятиграммы, а третий — семиграммы, соответственно. При этом, благодаря тому, что есть "skip connection", информация о трёхграммах не теряется, она прорасыпываться до самого конца.

SAMSUNG
Research
Russia


Предсказание частей речи на уровне отдельных токенов

```
In [13]: class SingleTokenPOSTagger(nn.Module):
    def __init__(self, vocab_size, labels_num, embedding_size=32, **kwargs):
        super().__init__()
        self.char_embeddings = nn.Embedding(vocab_size, embedding_size, padding_idx=0)
        self.backbone = StackedConv1d(embedding_size, **kwargs)
        self.global_pooling = nn.AdaptiveMaxPool1d(1)
        self.out = nn.Linear(embedding_size, labels_num)
        self.labels_num = labels_num

    def forward(self, tokens):
        """tokens - BatchSize x MaxSentenceLen x MaxTokenLen"""
        batch_size, max_sent_len, max_token_len = tokens.shape
        tokens_flat = tokens.view(batch_size * max_sent_len, max_token_len)

        char_embeddings = self.char_embeddings(tokens_flat) # BatchSize*MaxSentenceLen x MaxTokenLen x EmbSize
        char_embeddings = char_embeddings.permute(0, 2, 1) # BatchSize*MaxSentenceLen x EmbSize x MaxTokenLen
        features = self.backbone(char_embeddings)

        global_features = self.global_pooling(features).squeeze(-1) # BatchSize*MaxSentenceLen x EmbSize

        logits_flat = self.out(global_features) # BatchSize*MaxSentenceLen x LabelsNum
        logits = logits_flat.view(batch_size, max_sent_len, self.labels_num) # BatchSize x MaxSentenceLen x LabelsNum
        logits = logits.permute(0, 2, 1) # BatchSize x LabelsNum x MaxSentenceLen
        return logits
```

```
In [14]: single_token_model = SingleTokenPOSTagger(len(char_vocab), len(label2id),
                                                embedding_size=64, layers_n=3,
                                                kernel_size=3, dropout=0.3)
print('Количество параметров', sum(np.product(t.shape) for t in single_token_model.parameters()))
```

Предсказание частей речи на уровне отдельных токенов

```
In [13]: class SingleTokenPOSTagger(nn.Module):
    def __init__(self, vocab_size, labels_num, embedding_size=32, **kwargs):
        super().__init__()
        self.char_embeddings = nn.Embedding(vocab_size, embedding_size, padding_idx=0)
        self.backbone = StackedConv1d(embedding_size, **kwargs)
        self.global_pooling = nn.AdaptiveMaxPool1d(1)
        self.out = nn.Linear(embedding_size, labels_num)
        self.labels_num = labels_num

    def forward(self, tokens):
        """tokens - BatchSize x MaxSentenceLen x MaxTokenLen"""
        batch_size, max_sent_len, max_token_len = tokens.shape
        tokens_flat = tokens.view(batch_size * max_sent_len, max_token_len)

        char_embeddings = self.char_embeddings(tokens_flat) # BatchSize*MaxSentenceLen x MaxTokenLen x EmbSize
        char_embeddings = char_embeddings.permute(0, 2, 1) # BatchSize*MaxSentenceLen x EmbSize x MaxTokenLen

        features = self.backbone(char_embeddings)
        global_features = self.global_pooling(features).squeeze(-1) # BatchSize*MaxSentenceLen x EmbSize
        logits_flat = self.out(global_features) # BatchSize*MaxSentenceLen x LabelsNum
        logits = logits_flat.view(batch_size, max_sent_len, self.labels_num) # BatchSize x MaxSentenceLen x LabelsNum
        logits = logits.permute(0, 2, 1) # BatchSize x LabelsNum x MaxSentenceLen
        return logits

In [14]: single_token_model = SingleTokenPOSTagger(len(char_vocab), len(label2id),
                                                embedding_size=64, layers_n=3,
                                                kernel_size=3, dropout=0.3)
print('Количество параметров', sum(np.product(t.shape) for t in single_token_model.parameters()))
```

Из комментариев:

Вопрос:

далее мы меняем форму этого тензора, преобразуем его в трёхмерный, то есть "вспоминаем" о том, что у нас есть предложения, и транспонируем для того, чтобы порядок измерений соответствовал порядку измерений в исходном тензоре "tokens". (видео 4:20)

В докстринге forward написан изначальный

порядок: """tokens - BatchSize x MaxSentenceLen x MaxTokenLen""""

И перед последней операцией logits.permute(0, 2, 1) у нас размер

BatchSize x MaxSentenceLen x LabelsNum, т.е. у нас уже есть нужный порядок измерений.

Зачем делать logits.permute(0, 2, 1) превращая в # BatchSize x LabelsNum x MaxSentenceLen - непонятно.

Ответ:

это просто по сигнатуре функции [torch.nn.functional.cross_entropy](#), которую мы за лосс функцию в следующем видео берем как лосс функцию нужно — там в тензоре input измерения идут в порядке: количество элементов в батче, количество классов, и уже потом все пространственные измерения (в нашем случае, количество слов в предложении).

Вопрос:

Всем, привет! Вопрос от новичка в колабе:) делаю все по инструкции, все работает до того момента, пока есть свободное ОЗУ. Но память заканчивается просто очень быстро - после первого запуска нейронки уже забивается под 11гБ и на следующем шаге все рушится. При этом колаб автоматом мне не предлагает увеличить память до 24 гБ как это пишут в

источниках. Вот собственно вопрос - можно ли что-то сделать, чтобы озу меньше расходовалось или нужно как-то увеличить память?

Ответ:

Сам задал вопрос- сам ответил :)

В итоге пришлось много гуглить и я пришел к такому выходу. Не претендую, что это лучший вариант, но пока что смог и вдруг кому-то поможет:

- 1) все расчеты по обработке и созданию датасета выполнял на своем железе
- 2) затем датасет сохранил в пайторчевский файл и скопировал его на гуглдиск
- 3) запустил колаб и сконнектил его с гуглдиском
- 4) импортировал в колаб датасеты (трейн и валидацию)
- 5) запустил обучение модели, проверил качество

В итоге расход озу на колабе составил около 7гб, а этого вполне хватает для отладки этой модели и следующей

А если нужно менять данные, то меняю на железе и копирую в гугль диск. Немного муторно, зато памяти хватает

Если нужно поиграть с обученной моделью, то ее нужно сохранить и скопировать себе на железо. Поиграть стоит не дорого - можно и на своем железе 😊

Ответ:

мб покажете код части 4? Сходу не видно разницы между тем, что сразу в памяти весь датасет посчитать и посчитать его у себя, потом загрузить на гугл драйв, а потом целиком в память загрузить %) Или вы на части его дробили, когда в файл записывали? Ведь как-то еще надо случайные батчи оттуда выбирать будет...

На всякий случай напишу то, что я делал, вдруг тоже кому-то поможет. Проблема в том, что какие-то объекты занимают очень много места в памяти (перемножив размерности, можно увидеть, что объект train_inputs занимает примерно 3 гигабайта в памяти!), поэтому я не держал целиком датасет и переменные all_train_texts, train_char_tokenized в памяти, а вычислял значения на лету. Для последних двух переменных это делается просто через генератор (вставлять код в степике так себе идея, но надеюсь будет понятно):

```
def symbol_generator(texts):  
    for text in texts:  
        yield list(text.text)  
train_char_tokenized = symbol_generator(full_train)
```

— это позволяет не держать в памяти переменные all_train_texts, train_char_tokenized и сэкономить где-то 400 мб памяти. Для датасета нужно написать что-то такое:

```
from torch.utils.data import Dataset
```

```
class SymbolDataset(Dataset):  
    def __init__(self, texts, char2id, label2id, max_sent_len, max_token_len):  
        self.texts = texts  
        self.label2id = label2id  
        self.char2id = char2id  
        self.max_sent_len = max_sent_len
```

```

    self.max_token_len = max_token_len
def __len__(self):
    return len(self.texts)
def __getitem__(self, item):
    text = self.texts[item]
    inputs = torch.zeros(self.max_sent_len, self.max_token_len + 2, dtype=torch.long)
    labels = torch.zeros(self.max_sent_len, dtype=torch.long)
    for token_i, token in enumerate(text):
        labels[token_i] = self.label2id.get(token.upos, 0)
        word = token.form
        for char_i, char in enumerate(word):
            inputs[token_i, char_i + 1] = self.char2id.get(char, 0)
    return inputs, labels
train_dataset = SymbolDataset(full_train, char_vocab, label2id, MAX_SENT_LEN, MAX_ORIG_TOKEN_LEN)
test_dataset = SymbolDataset(full_test, char_vocab, label2id, MAX_SENT_LEN, MAX_ORIG_TOKEN_LEN)

```

Теперь оно нормально считается. Конечно, тут вопрос скорости встает, так как каждый раз создавать и вычислять тензоры, да еще и в цикле — не самое эффективное решение. Быстрее ли это, чем читать много файлов с диска, пока не знаю, скорость выполнения одних и тех же команд в колабе может отличаться раза в два-три)

Давайте попробуем обучить эту нейросеть и посмотрим, как хорошо мы можем определять часть речи слова, не используя информацию о его контексте. Сначала мы создаём экземпляр описанного нами нейросетевого модуля, передаём ему в конструктор количество уникальных символов в датасете, количество меток, рабочий размер модели, то есть каждый символ мы будем представлять вектором из 64 элементов, дальше мы указываем количество [свёрточных блоков](#), то есть глубину нашей нейросети (мы будем использовать глубину равную трём), размер ядра свёртки, а также вероятность дропаута. Вероятность равная 0.3 означает, что на каждом проходе по нейросети в режиме обучения будет, случайным образом, зануляться примерно треть активации. Как мы видим, наша нейросеть достаточно маленькая, в ней меньше пятидесяти тысяч элементов (по современным меркам, это очень маленькая нейросеть). Как вы думаете, сможет ли она вообще хоть что-то выучить? Далее мы используем нашу стандартную функцию для тренировки нейросетей, которая описана в нашей библиотеке специально для этого курса, передаём туда модель, датасеты, а также указываем функцию потерь — мы будем использовать [кросс-энтропию](#), эта функция потерь используется в многоклассовых задачах. Ну что ж, поехали! Обучение нейросети требует определённого времени, и в предыдущих семинарах нейросеть обучалась достаточно быстро даже на обычном процессоре. Здесь нам уже требуется видеокарта. То есть, она будет учиться и на обычном процессоре, но это потребует большего времени. Полное обучение описанной нейросети до сходимости на этом [корпусе](#) требует примерно одного часа работы видеокарты. Мы не будем ждать всё это время — мы просто загрузим модельку, которую я обучил заранее и оценим её качество. Для того, чтобы оценить качество, мы используем обученную нейросеть, чтобы получить предсказание для обучающей выборки, а дальше используем функцию

"classification_report" из библиотеки scikit-learn. Эта функция выдаёт самые стандартные метрики качества классификации (а именно, точность, полнота и [f-мера](#)) для каждого класса. А также, для каждого класса выдаётся количество примеров. Как мы видим, в этом случае датасет имеет сильно скошенное распределение классов, другими словами у нас есть очень частые классы и таких классов мало, а есть очень редкие классы. В таком случае нам вообще нет смысла использовать "[accuracy](#)", то есть долю верно угаданных ответов — она абсолютно неинформативна. Как вы видите, на обучающей выборке она равна единице, хотя есть классы, которые не очень хорошо определяются, на самом деле. В случае сильно скошенного распределения классов, самое правильное — это считать сразу несколько метрик, устойчивых к распределению классов (в данном случае, это — точность, полнота и f-мера) и смотреть на них всех. Часто бывает удобно получить не большую пачку цифр, а всего лишь одно число, по которому мы сможем понять, насколько хорошо модель работает. В данном случае лучше всего нам подходит "macro-среднее". Что значит "macro-среднее"? Это значит, что сначала мы считаем каждую метрику по каждому классу, а потом усредняем. Macro-среднее более устойчиво к скошенным распределениям классов.

```
In [14]: single_token_model = SingleTokenPOSTagger(len(char_vocab), len(label2id),
                                                embedding_size=64, layers_n=3,
                                                kernel_size=3, dropout=0.3)
print('Количество параметров', sum(np.product(t.shape) for t in single_token_model.parameters()))
Количество параметров 47826
```

```
In [15]: (best_val_loss,
          best_single_token_model) = train_eval_loop(single_token_model,
                                                       train_dataset,
                                                       test_dataset,
                                                       f'cross_entropy',
                                                       lr=5e-3,
                                                       epoch_n=10,
                                                       batch_size=64,
                                                       device='cuda',
                                                       early_stopping_patience=5,
                                                       max_batches_per_epoch_train=500,
                                                       max_batches_per_epoch_val=100,
                                                       lr_scheduler_ctor=lambda optim: torch.optim.lr_scheduler.ReduceLROnPlateau(optim, 'val'))
```

```
In [16]: torch.save(best_single_token_model.state_dict(), './models/single_token_pos.pth')
In [17]: single_token_model.load_state_dict(torch.load('./models/single_token_pos_best.pth'))
```



```
max_batches_per_epoch_train=500,
max_batches_per_epoch_val=100,
lr_scheduler_ctor=lambda optim: torch.optim.lr_scheduler.ReduceLROnPlateau(optim, 'val'))
```

Эпоха 0
 Эпоха: 501 итераций, 120.19 сек
 Среднее значение функции потерь на обучении 0.07971616704277174
 Среднее значение функции потерь на валидации 0.036421469071566466
 Новая лучшая модель!

Эпоха 1
 Эпоха: 501 итераций, 121.13 сек
 Среднее значение функции потерь на обучении 0.02936516877657877
 Среднее значение функции потерь на валидации 0.026192707839504916
 Новая лучшая модель!

Эпоха 2
 Эпоха: 501 итераций, 121.15 сек
 Среднее значение функции потерь на обучении 0.024633088314574874
 Среднее значение функции потерь на валидации 0.02537446160164505
 Новая лучшая модель!

Эпоха 3
 Эпоха: 501 итераций, 121.15 сек
 Среднее значение функции потерь на обучении 0.022628229849427164
 Среднее значение функции потерь на валидации 0.021124084406338707
 Новая лучшая модель!

Эпоха 4
 Эпоха: 501 итераций, 121.13 сек
 Среднее значение функции потерь на обучении 0.021418659499811078
 Среднее значение функции потерь на валидации 0.020120816849319653



SAMSUNG
Research
Russia

Эпоха 9
Эпоха: 501 итераций, 121.14 сек
Среднее значение функции потерь на обучении 0.018222528461463557
Среднее значение функции потерь на валидации 0.017878032584517898
Новая лучшая модель!

```
In [16]: torch.save(best_single_token_model.state_dict(), './models/single_token_pos.pth')

In [17]: single_token_model.load_state_dict(torch.load('./models/single_token_pos_best.pth'))
```

```
In [18]: train_pred = predict_with_model(single_token_model, train_dataset)
train_loss = F.cross_entropy(torch.tensor(train_pred),
                           torch.tensor(train_labels))
print('Среднее значение функции потерь на обучении', float(train_loss))
print(classification_report(train_labels.view(-1), train_pred.argmax(1).reshape(-1),
                            target_names=UNIQUE_TAGS))
print()

test_pred = predict_with_model(single_token_model, test_dataset)
test_loss = F.cross_entropy(torch.tensor(test_pred),
                           torch.tensor(test_labels))
print('Среднее значение функции потерь на валидации', float(test_loss))
print(classification_report(test_labels.view(-1), test_pred.argmax(1).reshape(-1),
                            target_names=UNIQUE_TAGS))
```

Предсказание частей речи на уровне предложений (с учётом контекста)

```
In [191]: class SentenceLevelPOSTagger(nn.Module):
```

	precision	recall	f1-score	support
<NOTAG>	1.00	1.00	1.00	9136391
ADJ	0.93	0.95	0.94	85589
ADP	1.00	0.99	1.00	81963
ADV	0.86	0.95	0.90	44101
AUX	0.85	0.89	0.87	7522
CCONJ	0.89	0.98	0.93	30432
DET	0.79	0.90	0.84	21968
INTJ	0.89	0.41	0.56	78
NOUS	0.99	0.95	0.97	214497
NUM	0.96	0.97	0.97	13746
PART	0.97	0.79	0.87	26651
PRON	0.94	0.82	0.87	38438
PROPN	0.83	0.97	0.90	32401
PUNCT	1.00	1.00	1.00	157989
SCONJ	0.80	0.76	0.78	16219
SYM	1.00	1.00	1.00	840
VERB	0.97	0.96	0.96	97670
X	0.97	0.70	0.81	375
accuracy			1.00	10006870
macro avg	0.92	0.89	0.90	10006870
weighted avg	1.00	1.00	1.00	10006870

Среднее значение функции потерь на валидации 0.012743492610752583
precision recall f1-score support

<NOTAG>	1.00	1.00	1.00	1231232
ADJ	0.92	0.95	0.93	11222
ADP	1.00	0.99	1.00	10585
ADV	0.85	0.94	0.89	6165

Из комментариев:

Комментарий №1:

Думаю, не мешало бы в данном случае остановиться на функции потерь, подробнее.

Насколько я понял мы в качестве меток (labels, targets) не используем one-hot-encoding, а просто числа (Размер одного targets равен max_sent_len), а из нейросети получаем лоджиты LabelsNum x MaxSentenceLen

Т.е допустим [[1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12], макс длина предложения

[2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13],

[3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14]]

[1,

2, - вот это типа POS метка первого токена в предложении

3]

<https://pytorch.org/docs/stable/torch.html#cross-entropy>

<https://pytorch.org/docs/stable/torch.nn.html#torch.nn.CrossEntropyLoss>

- **input** (*Tensor*) - (N, C) where $C = \text{number of classes}$ or (N, C, H, W) in case of 2D Loss, or $(N, C, d_1, d_2, \dots, d_K)$ where $K \geq 1$ in the case of K-dimensional loss.
- **target** (*Tensor*) - (N) where each value is $0 \leq \text{targets}[i] \leq C - 1$, or $(N, d_1, d_2, \dots, d_K)$ where $K \geq 1$ for K-dimensional loss.

Пример. вот такой

```
>>> loss = nn.CrossEntropyLoss() # nn.CrossEntropyLoss() это почти аналог F.cross_entropy
>>> input = torch.randn(3, 5, requires_grad=True)
>>> target = torch.empty(3, dtype=torch.long).random_(5)
>>> output = loss(input, target)
>>> output.backward()
```

Ответ:

тоже думаю, что можно остановиться подробнее

для одного токена кросс-энтропия имеет вид:

$$CE(y, \hat{y}) = - \sum_i y_i \log \hat{y}_i$$

y - вектор с единицей на позиции t , остальные элементы - нули

$\hat{y} = \text{softmax}(z) = \frac{e^z}{\sum_k e^{z_k}}$, где z - вектор, полученный в результате последнего линейного слоя

$$CE(y, \hat{y}) = - \sum_i y_i \log \hat{y}_i = - \log \frac{e^{z_t}}{\sum_k e^{z_k}} = -z_t + \log(\sum_k e^{z_k})$$

для целого предложения или батча токенов кросс-энтропия будет вектором, а не числом

torch.nn.functional.cross_entropy в качестве значения возвращает среднее по батчу

на примере одного предложения:

```
tokens = train_dataset.tensors[0][1]
tokens = tokens.view(1, tokens.shape[0], tokens.shape[1]) # torch.Size([1, 205, 49])

labels = train_dataset.tensors[1][1]
labels = labels.view(1, labels.shape[0]) # torch.Size([1, 205])

pred = single_token_model(tokens) # torch.Size([1, 18, 205])

F.cross_entropy(pred, labels) # tensor(0.0664, grad_fn=<NllLoss2DBackward>

np_pred = pred.data.numpy() # (1, 18, 205)
```

```
np_labels = labels.data.numpy() # (1, 205)

np.mean([
    np.log(np.exp(np_pred[0, :, i]).sum()) - np_pred[0, label, i]
    for i, label in enumerate(np_labels[0])
]) # 0.06644829
```

Мы видим, что, несмотря на то, что модель была очень простая и она никак не учитывала контексты, на обучающей выборке она показывает достаточно высокую f-меру (0.9). Что же на валидационной выборке? Мы видим, что на валидационной выборке macro-средняя [f-мера](#) не сильно меньше, всего лишь на 1% (0.89 — достаточно неплохо). Ранее, во вводных лекциях в этом курсе, мы говорили о так называемой частиречной омонимии. Частиречная [омонимия](#) — это когда слова "с разной частью речи" пишутся абсолютно одинаково. Например, "мама мыла раму": здесь "мыла" — это глагол. Или "в ванной не было мыла": здесь "мыла" — это уже существительное. Такие случаи принципиально невозможно отлавливать, используя только информацию изнутри слова — нам нужно учитывать контекст в предложении. Давайте опишем вторую модельку, которая учитывает такой контекст. Опять же, давайте посмотрим на метод forward. Первая часть этого метода — ровно такая же, как и в предыдущей модели. Берём номера символов, делаем выборку из таблицы [эмбеддингов](#), получаем векторы символов. Потом, с помощью свёрточной нейросети, которая в данном случае называется "single token backbone", получаем векторы символов с учётом их контекста. Затем используем глобальный пулинг для того, чтобы получить вектор токена. Далее мы проделываем примерно то же самое, но уже на уровне токенов в предложении. Сначала мы немного изменяем форму и транспонируем тензор признаков таким образом, чтобы получить трёхмерный тензор с размерностями "количество предложений в батче", "количество признаков для каждого токена" и "количество токенов в предложении". Этот тензор содержит признаки токенов без учёта их контекста. Далее мы эти признаки подаём в другой свёрточный модуль для того, чтобы учесть это самый контекст. Для учёта контекста символов и для учёта контекста слов мы используем две разных нейросети, но архитектура их одинакова — это ровно та самая [свёрточная сеть](#) со "skip connections", которую мы описали в начале нашего семинара. Несмотря на то, что архитектура у этих модулей одинаковая, веса у них отличаются. На старте обучения они инициализируются случайно, а затем — настраиваются. В результате мы получаем также трёхмерный тензор, той же размерности, но теперь — вектор для каждого токена уже содержит информацию о его контексте. И, наконец, нам нужно принять решение о части речи каждого токена. Для этого мы проецируем признаки каждого токена в пространство классов. Для этого мы используем одномерный свёрточный блок с ядром свёртки 1, то есть он за раз видит только один токен. Физический смысл того, что мы описали сейчас, заключается в том, чтобы сначала проанализировать структуру каждого слова, найти там какие-то суффиксы и окончания (за это отвечает первая часть) а затем — смешать информацию о структуре каждого слова с контекстом, в котором это слово употребляется. Давайте посмотрим, насколько сильно

это вообще влияет. Может быть, явление частиречной омонимии и не такое уж, серьёзное — может, я тут вам всё преувеличиваю. Сначала мы создаём экземпляр класса, который только что описали, и передаём туда, на самом деле, все те же самые параметры — это количество символов, количество выходных меток, количество признаков для каждого символа, но теперь это ещё и количество признаков для каждого токена (оно у нас одинаково), а также два набора параметров для свёрточных модулей. Первый набор параметров — для анализа символов на уровне каждого токена, а второй набор параметров — для анализа контекста токенов. Мы решили задать одни и те же параметры, потому что это неплохо работает. В результате мы получили нейросеть, в которой в два раза больше параметров, но по-прежнему это очень маленькое число для современных нейросетей. Далее мы используем всё тот же цикл обучения — здесь никаких существенных отличий нет. Если вы потом внимательно посмотрите на вывод обучения, вы увидите, что для второй модели, которая учитывает контекст токенов, значение функции потерь во время обучения падает гораздо быстрее и, в итоге, достигает существенно меньшего значения. Давайте сразу перейдём к метрикам качества на валидационной выборке. Здесь у нас macro f-мера, равна 0.93, а для первой модели, как мы помним, она была равна 0.89. То есть, мы получили прирост в 4%, добавив всего лишь ещё один свёрточный модуль для учёта контекста токенов. 4%, на фоне 90% — казалось бы, не очень впечатляюще. Давайте теперь посмотрим собственными глазами, в каких же случаях эти 4% действительно играют роль. Для того, чтобы вручную поэкспериментировать с обученными моделями мы сделали специальный класс "[POS tagger](#)", который принимает на вход а обученную модель, а также те же самые параметры — это отображение символов в их идентификаторы, это количество уникальных тэгов. Кроме этого, он на вход принимает отображение символов в их номера, отображение из номеров тэгов обратно в их строковое представление, а также статистики [корпуса](#) — максимальную длину предложения и максимальную длину токена. И мы создаём два экземпляра этого класса для модели, обученной на отдельных токенах, и для модели, которая учитывает контекст. Класс "POS tagger" достаточно простой. В конструктор он принимает все эти параметры, которые мы сейчас описали, и у него есть ещё метод — этот метод применяет нашу обученную модель к переданным предложениям для того, чтобы получить части речи токенов. Для того, чтобы определить части речи токенов, сначала мы токенизуем переданное нам предложение, затем делаем то, что мы делали при обучении — а именно, перекладываем информацию о символах в тензор, затем, в цикле, применяем нашу модель для каждого предложения. В результате применения нашей модели к каждому предложению мы получаем трёхмерный тензор, первая размерность этого [тензора](#) соответствует количеству предложений, вторая — это количество меток, а третья — это максимальная длина предложения. Таким образом, для каждого предложения и для каждого токена у нас есть распределение вероятностей по меткам. И здесь мы просто выбираем для каждого токена наиболее вероятную метку. Ну, и напоследок — мы преобразовываем полученную информацию в формат, удобный для человека, то есть мы преобразовываем номера тэгов в их строковое название.

	Среднее значение функции потерь на обучении	precision	recall	f1-score	support
<NOTAG>	0.011244023218750954	1.00	1.00	1.00	9136391
ADJ		0.93	0.95	0.94	85589
ADP		1.00	0.99	1.00	81963
ADV		0.86	0.95	0.90	44101
AUX		0.85	0.89	0.87	7522
CCONJ		0.89	0.98	0.93	30432
DET		0.79	0.90	0.84	21968
INTJ		0.89	0.41	0.56	78
NOUN		0.99	0.95	0.97	214497
NUM		0.96	0.97	0.97	13746
PART		0.97	0.79	0.87	26651
PRON		0.94	0.82	0.87	38438
PROPN		0.83	0.97	0.90	32401
PUNCT		1.00	1.00	1.00	157989
SCONJ		0.80	0.76	0.78	16219
SYM		1.00	1.00	1.00	840
VERB		0.97	0.96	0.96	97670
X		0.97	0.70	0.81	375
accuracy				1.00	10006870
macro avg		0.92	0.89	0.90	10006870
weighted avg		1.00	1.00	1.00	10006870



	Среднее значение функции потерь на валидации	precision	recall	f1-score	support
<NOTAG>	0.012743492610752583	1.00	1.00	1.00	1231232
ADJ		0.92	0.95	0.93	11222
ADP		1.00	0.99	1.00	10585
ADV		0.85	0.94	0.89	6165
AUX		0.84	0.88	0.86	1106
CCONJ		0.89	0.98	0.93	4410
DET		0.77	0.88	0.82	3085
INTJ		1.00	0.27	0.43	11
NOUN		0.98	0.95	0.96	27974
NUM		0.95	0.95	0.95	1829
PART		0.96	0.79	0.87	3877
PRON		0.94	0.79	0.86	5598
PROPN		0.82	0.94	0.88	4438
PUNCT		1.00	1.00	1.00	22694
SCONJ		0.77	0.74	0.76	2258
SYM		1.00	1.00	1.00	53
VERB		0.96	0.95	0.95	13078
X		0.96	0.83	0.89	105
accuracy				1.00	1349720
macro avg		0.92	0.88	0.89	1349720
weighted avg		1.00	1.00	1.00	1349720



Предсказание частей речи на уровне предложений (с учётом контекста)

```
In [19]: class SentenceLevelPOSTagger(nn.Module):
    def __init__(self, vocab_size, labels_num, embedding_size=32,
                 single_backbone_kwarg={}, context_backbone_kwarg={}):
        super().__init__()
        self.embedding_size = embedding_size
        self.char_embeddings = nn.Embedding(vocab_size, embedding_size, padding_idx=0)
        self.single_token_backbone = StackedConv1d(embedding_size, **single_backbone_kwarg)
        self.context_backbone = StackedConv1d(embedding_size, **context_backbone_kwarg)
        self.global_pooling = nn.AdaptiveMaxPool1d(1)
        self.out = nn.Conv1d(embedding_size, labels_num, 1)
        self.labels_num = labels_num

    def forward(self, tokens):
        """tokens - BatchSize x MaxSentenceLen x MaxTokenLen"""
        batch_size, max_sent_len, max_token_len = tokens.shape
        tokens_flat = tokens.view(batch_size * max_sent_len, max_token_len)

        char_embeddings = self.char_embeddings(tokens_flat) # BatchSize*MaxSentenceLen x MaxTokenLen x EmbSize
        char_embeddings = char_embeddings.permute(0, 2, 1) # BatchSize*MaxSentenceLen x EmbSize x MaxTokenLen
        char_features = self.single_token_backbone(char_embeddings)

        token_features_flat = self.global_pooling(char_features).squeeze(-1) # BatchSize*MaxSentenceLen x EmbSize
        token_features = token_features_flat.view(batch_size, max_sent_len, self.embedding_size)
        token_features = token_features.permute(0, 2, 1) # BatchSize x EmbSize x MaxSentenceLen
        context_features = self.context_backbone(token_features) # BatchSize x EmbSize x MaxSentenceLen

        logits = self.out(context_features) # BatchSize x LabelsNum x MaxSentenceLen
        return logits
```



```
In [20]: sentence_level_model = SentenceLevelPOSTagger(len(char_vocab), len(label2id), embedding_size=64,
                                                       single_backbone_kwarg=dict(layers_n=3,
                                                       kernel_size=3,
                                                       dropout=0.3),
                                                       context_backbone_kwarg=dict(layers_n=3,
                                                       kernel_size=3,
                                                       dropout=0.3))
print('Количество параметров', sum(np.product(t.shape) for t in sentence_level_model.parameters()))
Количество параметров 84882
```

```
In [21]: (best_val_loss,
           best_sentence_level_model) = train_eval_loop(sentence_level_model,
                                                       train_dataset,
                                                       test_dataset,
                                                       F.cross_entropy,
                                                       lr=5e-3,
                                                       epoch_n=10,
                                                       batch_size=64,
                                                       device='cuda',
                                                       early_stopping_patience=5,
                                                       max_batches_per_epoch_train=500,
                                                       max_batches_per_epoch_val=100,
                                                       lr_scheduler_ctor=lambda optim: torch.optim.lr_schedul...
```



```
In [22]: torch.save(best_sentence_level_model.state_dict(), './models/sentence_level_pos.pth')
```

```
In [23]: sentence_level_model.load_state_dict(torch.load('./models/sentence_level_pos_best.pth'))
```

```
Эпоха 6
Эпоха: 501 итераций, 124.75 сек
Среднее значение функции потерь на обучении 0.015094459519727144
Среднее значение функции потерь на валидации 0.01196445744668789
Новая лучшая модель!
```

```
Эпоха 7
Эпоха: 501 итераций, 124.75 сек
Среднее значение функции потерь на обучении 0.01438999031400847
Среднее значение функции потерь на валидации 0.011267489726000493
Новая лучшая модель!
```

```
Эпоха 8
Эпоха: 501 итераций, 124.75 сек
Среднее значение функции потерь на обучении 0.014113767176658331
Среднее значение функции потерь на валидации 0.011351724819160334
```

```
Эпоха 9
Эпоха: 501 итераций, 124.97 сек
Среднее значение функции потерь на обучении 0.013891586998548217
Среднее значение функции потерь на валидации 0.01066051164439114
Новая лучшая модель!
```

```
In [22]: torch.save(best_sentence_level_model.state_dict(), './models/sentence_level_pos.pth')
```

```
In [23]: sentence_level_model.load_state_dict(torch.load('./models/sentence_level_pos_best.pth'))
```

```
In [24]: train_pred = predict_with_model(sentence_level_model, train_dataset)
train_loss = F.cross_entropy(torch.tensor(train_pred),
                           torch.tensor(train_labels))
```



```
Среднее значение функции потерь на валидации 0.007575111463665962
precision      recall   f1-score   support
```

<NOTAG>	1.00	1.00	1.00	1231232
ADJ	0.96	0.94	0.95	11222
ADP	1.00	1.00	1.00	10585
ADV	0.94	0.94	0.94	6165
AUX	0.87	0.96	0.91	1106
CCONJ	0.95	0.98	0.97	4410
DET	0.91	0.93	0.92	3085
INTJ	0.67	0.36	0.47	11
NOUN	0.98	0.98	0.98	27974
NUM	0.96	0.97	0.96	1829
PART	0.97	0.91	0.94	3877
PRON	0.96	0.92	0.94	5598
PROPN	0.95	0.96	0.96	4438
PUNCT	1.00	1.00	1.00	22694
SCONJ	0.88	0.95	0.92	2258
SYM	1.00	1.00	1.00	53
VERB	0.96	0.97	0.97	13078
X	0.99	0.77	0.87	105
accuracy			1.00	1349720
macro avg	0.94	0.92	0.93	1349720
weighted avg	1.00	1.00	1.00	1349720

Применение полученных теггеров и сравнение

```
In [25]: single_token_pos_tagger = POSTagger(single_token_model, char_vocab,
                                             UNIQUE_TAGS, MAX_SENT_LEN, MAX_ORIG_TOKEN_LEN)
sentence_level_pos_tagger = POSTagger(sentence_level_model, char_vocab)
```



ПРИМЕНЕНИЕ ПОЛУЧЕННЫХ ИСКЛЕВОДИТЕЛЬСТВ

```
In [25]: single_token_pos_tagger = POSTagger(single_token_model, char_vocab,
                                         UNIQUE_TAGS, MAX_SENT_LEN, MAX_ORIG_TOKEN_LEN)
sentence_level_pos_tagger = POSTagger(sentence_level_model, char_vocab,
                                       UNIQUE_TAGS, MAX_SENT_LEN, MAX_ORIG_TOKEN_LEN)
```

```
In [26]: test_sentences = [
    'Мама мыла раму.',
    'Косил косой косой косой.',
    'Глокая куздра штеко будланула бокра и куздрячит бокрёнка.',
    'Сыпала Калуша с Калушатами по напушке.',
    'Пирожки поставлены в печь, мама любит печь.',
    'Ведро дало течь, вода стала течь.',
    'Три да три, будет дырка.',
    'Три да три, будет шесть.',
    'Сорок сорок'
```

```
test_sentences_tokenized = tokenize_corpus(test_sentences, min_token_size=1)
```

```
In [27]: for sent_tokens, sent_tags in zip(test_sentences_tokenized,
                                         single_token_pos_tagger(test_sentences)):
    print(' '.join('{}-{}'.format(tok, tag) for tok, tag in zip(sent_tokens, sent_tags)))
print()
```

...

```
In [28]: for sent_tokens, sent_tags in zip(test_sentences_tokenized,
                                         sentence_level_pos_tagger(test_sentences)):
    print(' '.join('{}-{}'.format(tok, tag) for tok, tag in zip(sent_tokens, sent_tags)))
print()
```

...



```
for sent_i, sent in enumerate(sentences):
    for token_i, token in enumerate(sent):
        targets[sent_i, token_i] = label2id.get(token.upos, 0)
        for char_i, char in enumerate(token.form):
            inputs[sent_i, token_i, char_i + 1] = char2id.get(char, 0)

return inputs, targets

class POSTagger:
    def __init__(self, model, char2id, id2label, max_sent_len, max_token_len):
        self.model = model
        self.char2id = char2id
        self.id2label = id2label
        self.max_sent_len = max_sent_len
        self.max_token_len = max_token_len

    def __call__(self, sentences):
        tokenized_corpus = tokenize_corpus(sentences, min_token_size=1)

        inputs = torch.zeros((len(sentences), self.max_sent_len, self.max_token_len + 2), dtype=torch.long)

        for sent_i, sentence in enumerate(tokenized_corpus):
            for token_i, token in enumerate(sentence):
                for char_i, char in enumerate(token):
                    inputs[sent_i, token_i, char_i + 1] = self.char2id.get(char, 0)

        dataset = TensorDataset(inputs, torch.zeros(len(sentences)))
        predicted_probs = predict_with_model(self.model, dataset) # SentenceN x TagsN x MaxSentLen
        predicted_classes = predicted_probs.argmax(1)

        result = []
        for sent_i, sent in enumerate(tokenized_corpus):
```



Из комментариев:

Вопрос:

очему в случае SingleTokenPOSTagger в out используется полно связный слой, а в out из SentenceLevelPOSTagger свертка? Есть какая-то принципиальная разница? Спасибо.

Ответ:

разницы нет, применить к матрице A размера $Cin \times Lin$ (в нашем случае число каналов — это размерность вектора эмбеддинга, а длина последовательности — это максимальная длина предложения) одномерную свертку K с числом выходных каналов Cout — это то же, что представить K как матрицу размера $Cin \times Cout$ и умножить A.T на K (в pytorch в Conv1d размерности расставлены в порядке, обратном тому, который мы в прошлом разделе смотрели [вот тут](#) можно глянуть).

Для демонстрации мы с ребятами подобрали несколько коварных предложений. Во-первых, это "мама мыла раму" — здесь омонимичное слово "мыла". Второе предложение — это "косил косой косой косой" — здесь вообще непонятно, что происходит. Затем — известное предложение "глокая куздра штеко будланула бокра и кудрячит бокрёнка", а также, ещё несколько примеров в таком же духе. Таким образом, мы имеем возможность проверить, во-первых — насколько модель хорошо определяет части речи слов для неизвестных слов. Предложений про "глокую куздрю" в обучающей выборке не было. Таким образом это тест на неизвестные слова, а ещё есть несколько тестов на контекст — на учёт контекста, а именно здесь: "ведро дало течь" — это существительное, а "вода стала течь" — это глагол. "Сорок сорок" — здесь одно из слов числительное, а другое — существительное. Ну что ж, давайте посмотрим, как наши модели отработали на этих предложениях. Мы видим, что, например, в последнем предложении "сорок сорок", модель назначает просто наиболее вероятный тэг: слово "сорок" всё-таки чаще используются как числительное. Аналогично и в других случаях. Но хорошая новость — в том, что для неизвестных слов модель отработала просто отлично. Хотя допустила пару ошибок, а именно — "штеко" — это не существительное, а наречие. Ну что ж, теперь — вторая модель. Мы сразу видим, что "сорок сорок" уже разобраны более-менее правильно, то есть одно из этих слов — это числительное, а второе — это существительное. Также в предложении про "глокую куздрю" исправлена ошибка с наречием, в предложении про "пирожки" эта нейросеть уже смогла различить ситуации, в которых и слово "печь" используется как существительное, и как глагол. Забавно, что во втором аналогичном предложении нейросеть тоже отличила две ситуации, но перепутала. То есть, в том случае, где должен был быть глагол, получилось существительное, а во втором случае должно было быть существительное, а получился глагол. Предложение про "косого косого", который "косил косой" тоже разобрано чуть лучше, хотя и не идеально. Предложения про "три да три будет шесть" или "будет дырка" разобрались так же, предположительно — потому, что мы использовали только три слоя свёрток для учёта контекста токенов и, возможно, нейросети просто не хватило рецептивного поля, то есть для слова "три" она не могла увидеть, что в конце предложения есть слово "шесть". Это просто очень далеко. Вы можете это проверить сами, добавив больше свёрток. Мы увидели, что, действительно, для задачи определения частей речи, учитывать контекст токенов действительно важно, в определённых случаях. А также мы, в принципе, убедились, что архитектура, которую мы описали, работает для этой задачи и достигает приемлемых показателей качества.

```

test_sentences = [
    'Мама мыла раму.',
    'Косил косой косой косой.',
    'Глокая куздра штеко будланула бокра и куздрячит бокрёнка.', ✓
    'Сыпала Калуша с Калушатами по напушке.',
    'Пирожки поставлены в печь, мама любит печь.',
    'Ведро дало течь, вода стала течь.',
    'Три да три, будет дырка.',
    'Три да три будет шесть.',
    'Сорок сорок'
]
test_sentences_tokenized = tokenize_corpus(test_sentences, min_token_size=1)

```

```
In [27]: for sent_tokens, sent_tags in zip(test_sentences_tokenized,
                                         sentence_level_pos_tagger(test_sentences)):
    print(' '.join('{}-{}'.format(tok, tag) for tok, tag in zip(sent_tokens, sent_tags)))
    print()
    ...

```

```
In [28]: for sent_tokens, sent_tags in zip(test_sentences_tokenized,
                                         sentence_level_pos_tagger(test_sentences)):
    print(' '.join('{}-{}'.format(tok, tag) for tok, tag in zip(sent_tokens, sent_tags)))
    print()
    ...

```

Свёрточный модуль своими руками

```
In [29]: class MyConv1d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, padding=0):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
```



мама-NOUN мыла-VERB раму-NOUN
 косил-VERB косой-ADJ косой-ADJ косой-ADJ ✓
 глокая-ADJ куздра-NOUN штеко-NOUN будланула-VERB бокра-NOUN и-CCONJ куздрячит-VERB бокрёнка-NOUN
 сыпала-VERB калуша-NOUN с-ADP калушатами-NOUN по-ADP напушке-ADV
 пирожки-NOUN поставлены-VERB в-ADP печь-NOUN мама-NOUN любит-VERB печь-NOUN
 ведро-NOUN дало-VERB течь-NOUN вода-NOUN стала-VERB течь-NOUN
 три-NUM да-CCONJ три-NUM будет-AUX дырка-NOUN
 три-NUM да-CCONJ три-NUM будет-AUX шесть-NUM
 сорок-NUM сорок-NUM

```
In [28]: for sent_tokens, sent_tags in zip(test_sentences_tokenized,
                                         sentence_level_pos_tagger(test_sentences)):
    print(' '.join('{}-{}'.format(tok, tag) for tok, tag in zip(sent_tokens, sent_tags)))
    print()
    ...

```



Свёрточный модуль своими руками

```
In [29]: class MyConv1d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, padding=0):
        super().__init__()
        self.in_channels = in_channels
```

```
In [28]: for sent_tokens, sent_tags in zip(test_sentences_tokenized,
                                         sentence_level_pos_tagger(test_sentences)):
    print(' '.join('{{}-{}''.format(tok, tag) for tok, tag in zip(sent_tokens, sent_tags)))
```

```
мама-NOUN мыла-VERB раму-NOUN
косил-VERB косой-ADJ косой-ADJ косой-NOUN
глокая-ADJ куздра-NOUN штеко-ADV будланула-VERB бокра-NOUN и-CCONJ куздрячит-VERB бокрёнка-NOUN
сияла-VERB калуша-NOUN с-ADP калушатами-NOUN по-ADP напушке-NOUN
пирожки-NOUN поставлены-VERB в-ADP печь-NOUN мама-NOUN любит-VERB печь-VERB
ведро-NOUN дало-VERB течь-VERB вода-NOUN стала-VERB течь-VERB
три-NUM да-PART три-NUM будет-AUX дырка-NOUN
три-NUM да-PART три-NUM будет-AUX шесть-NUM
сорок-NUM сорок-NOUN
```

Свёрточный модуль своими руками

```
In [29]: class MyConv1d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, padding=0):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
```



Давайте копнём чуть чуть глубже — посмотрим, как же можно реализовать свёртки своими руками. Для этого давайте опишем класс — все pytorch-модули наследуются от базового класса "[nn.module](#)", давайте реализуем не весь, но основной функционал стандартного модуля "[nn.conv1d](#)". При этом, давайте будем, по возможности, делать так, чтобы наш модуль можно было использовать как замену — один в один. Итак, входные параметры — это количество исходных каналов, количество результирующих каналов, размер ядра и размер паддинга. В конструкторе а мы создаём два [тензора](#) — это наши веса. "self.weight" — это ядро свёртки. В отличие от стандартного модуля, мы будем использовать здесь прямоугольную матрицу, в этой матрице количество строк равно количеству входных каналов умноженному на размер свёртки (на размер ядра), а количество столбцов этой матрицы равно количеству выходных каналов. Инициализируем этот тензор мы шумом из нормального распределения с маленькой дисперсией. Также у нас есть второй тензор параметров — это смещение. Это просто вектор размерности, равной количеству выходных каналов, его мы инициализируем нулями. Теперь давайте посмотрим на метод "forward". На вход одномерные свёртки принимают трёхмерные тензоры, первая размерность которых соответствует размеру батча, вторая — количеству входных каналов, и третья — длине последовательности. А в качестве результата возвращают также трёхмерный тензор размерности ["количество элементов в батче", "количество выходных каналов" на "новую длину последовательности"]. Она может либо оставаться прежней, либо уменьшится, либо увеличится — это зависит от размеров ядра и от паддинга. Сначала мы делаем паддинг (вот этот кусок кода отвечает за паддинг). Здесь мы реализовали паддинг только нулями. В случае одномерных свёрток, паддинг заключается в том, что мы увеличиваем

третье измерение (удлиняем тензор по третьему измерению — то есть, по длине последовательности) на указанное количество элементов, как спереди, так и сзади. Для этого мы создаём тензор нулей, а затем — конкатенируем его вместе с исходным тензором признаков, так, чтобы паддинги были и в начале тензора, и в конце. И нам нужно вычислить заново длину последовательности, переприсвоить переменной. Далее мы подготавливаем матрицу признаков. Как мы это делаем? Вот этот кусок кода отвечает за подготовку признаков. Предположим, что на вход нам пришла вот такая матрица. В ней каждый столбец соответствует какому-то элементу последовательности. Давайте их условно перенумеруем "1, 2, 3, 4, 5". И у нас ядро свёртки, допустим, равно трём. Тогда мы из исходной матрицы сформируем новую матрицу, в которой количество столбцов — новое, оно вычисляется как длина исходной последовательности минус размер ядра плюс 1 (длина исходной последовательности, конечно, берётся уже с учётом паддинга). Для примера — допустим, что у нас нету паддинга. Итак — как мы будем готовить матрицу признаков? Мы будем перебирать все смещения от 0 до размера ядра - 1 и для каждого смещения мы будем брать фрагмент исходной матрицы, начиная с этого смещения, длины равной результирующей длине. Давайте на примере — для исходной последовательности длины 5 и ядра свёртки 3, длина выходной последовательности будет также равна 3 (то есть, chunk size будет равен 3). Тогда — перебирать все смещения от 0 до 2 и выбирать фрагмент исходной матрицы (сначала мы берём такую под-матрицу, затем берём начиная со второго элемента, затем — начиная с третьего элемента). Потом, когда мы перебрали все возможные смещения, мы объединяем все эти кусочки в одну матрицу, причём объединяя её по второму измерению, то есть по измерению признаков — мы получаем, в итоге, вот такую новую матрицу, в ней три столбца, а количество строк равно количеству исходных каналов умноженное на размер ядра свёртки (здесь у нас, по сути, значения будут такие). Таким образом, мы реализовали скользящее окно, которым мы идём по данным. То есть, каждый столбец этой матрицы содержит признаки всех данных, которые попадают в скользящее окно, которым мы идём по исходной последовательности и преобразовываем исходную последовательность в каждом окне. После объединения кусочков в одну матрицу признаков мы её транспонируем и применяем ядро свёртки. Делать мы это будем с помощью функции "[torch.bmm](#)" ("BMM" расшифровывается как "batch matrix multiplication", то есть это пакетное матричное перемножение). эта функция принимает на вход два трёхмерных тензора и, для каждой пары матриц в этих тензорах, делает матричное умножение. Первый тензор — это признаки, которые мы только что составили из кусочков, и второй тензор — это наше ядро свёртки, но наше ядро свёртки — это матрица, а нам нужно получить трёхмерный тензор. Для этого мы добавляем фиктивное нулевое измерение, соответствующее размеру батча и вызываем функцию "expand" ("expand" изменяет размер тензора, но, при этом, она делает это без выделения дополнительной памяти, то есть снаружи выглядит, что тензор большой, а на самом деле это — просто плоская матрица). Итак, ядро свёртки применили, теперь добавляем смещение и транспонируем полученную матрицу так, чтобы у нас смысл измерений в результирующем тензоре соответствовал смыслу измерений во входном тензоре, то есть — сначала размер батча, потом количество каналов, а потом — пространственные измерения, то

есть длина последовательности. Всё, это дело мы передаём дальше. Давайте попробуем обучить модель для определения частей речи токенов, но свёрточный модуль заменим на наш свёрточный модуль, возьмём за основу модель, которая учитывает контекст токенов, и создадим экземпляр этой модели, и передадим туда все те же самые параметры, которые передавали и раньше, но добавим ещё один — а именно, скажем ей "используй, пожалуйста, наш модуль — не стандартный `nn.conv1d` из `pytorch`, а наш модуль, который мы только что описали". Как мы видим, количество параметров в результате мы получили ровно такое же. Обучаем нашу модель, используя ту же самую функцию. Любопытно, что модель, которая использует наш свёрточный модуль, на одну эпоху требует 52 секунды. А модель, которая использовала стандартный свёрточный модуль требовала примерно 120 секунд на одну эпоху, то есть получается, что наш модуль чуть-чуть быстрее. Эта разница имеет значение только для той версии `pytorch`, которую используем мы сейчас, в другой версии `pytorch` разница может быть совершенно другая. Наиболее вероятно, что это ускорение вызвано тем, что наша реализация свёрток узкоспециализирована, в ней нету кучи разных вариантов паддинга, в ней нет механизма прореживания ядра, а также нет страйдов — в ней нельзя задавать шаг, с которым нужно идти скользящим окном по исходной последовательности. Другими словами, она проще гораздо, чем стандартная реализация свёрток. Таким образом, иногда имеет смысл что-то реализовать самому, но надо помнить, что в машинном обучении, часто, гораздо важнее быстро проверять разные гипотезы, а для этого лучше использовать стандартные модули, которые проверены, работают надёжно в разных ситуациях, они универсальны и вам не нужно тратить время на написание своих модулей. Свои модули имеет смысл писать только тогда, когда вы точно знаете ту архитектуру, которая вам нужна для решения вашей прикладной задачи, и вы хотите её ускорить. Например — для того, чтобы уметь запускать её на мобильном телефоне.

Дополнительные комментарии к видео (от Романа Суворова):

- 7:03 - переменная в видео неправильная (должна быть `sentence_level_model_my_conv`), но количество параметров правильное

Свёрточный модуль своими руками

```
In [29]: class MyConv1d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, padding=0):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.padding = padding
        self.weight = nn.Parameter(torch.randn(in_channels * kernel_size,
                                              out_channels) / (in_channels * kernel_size),
                                  requires_grad=True)
        self.bias = nn.Parameter(torch.zeros(out_channels), requires_grad=True)

    def forward(self, x):
        """x - BatchSize x InChannels x SequenceLen"""

        batch_size, src_channels, sequence_len = x.shape
        if self.padding > 0:
            pad = x.new_zeros(batch_size, src_channels, self.padding)
            x = torch.cat((pad, x, pad), dim=-1)
            sequence_len = x.shape[-1]

        chunks = []
        chunk_size = sequence_len - self.kernel_size + 1
        for offset in range(self.kernel_size):
            chunks.append(x[:, :, offset:offset + chunk_size])

        in_features = torch.cat(chunks, dim=1) # BatchSize x InChannels x KernelSize x ChunkSize
        in_features = in_features.permute(0, 2, 1) # BatchSize x ChunkSize x InChannels x KernelSize
        out_features = torch.bmm(in_features,
                               self.weight.unsqueeze(0).expand(batch_size, -1, -1))
        out_features = out_features + self.bias.unsqueeze(0).unsqueeze(0)
        out_features = out_features.permute(0, 2, 1) # BatchSize x OutChannels x ChunkSize
```



Свёрточный модуль своими руками

```
In [29]: class MyConv1d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, padding=0):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.padding = padding
        self.weight = nn.Parameter(torch.randn(in_channels * kernel_size,
                                              out_channels) / (in_channels * kernel_size),
                                  requires_grad=True)
        self.bias = nn.Parameter(torch.zeros(out_channels), requires_grad=True)

    def forward(self, x):
        """x - BatchSize x InChannels x SequenceLen"""

        batch_size, src_channels, sequence_len = x.shape
        if self.padding > 0:
            pad = x.new_zeros(batch_size, src_channels, self.padding)
            x = torch.cat((pad, x, pad), dim=-1)
            sequence_len = x.shape[-1]

        chunks = []
        chunk_size = sequence_len - self.kernel_size + 1
        for offset in range(self.kernel_size):
            chunks.append(x[:, :, offset:offset + chunk_size])

        in_features = torch.cat(chunks, dim=1) # BatchSize x InChannels x KernelSize x ChunkSize
        in_features = in_features.permute(0, 2, 1) # BatchSize x ChunkSize x InChannels x KernelSize
        out_features = torch.bmm(in_features,
                               self.weight.unsqueeze(0).expand(batch_size, -1, -1))
        out_features = out_features + self.bias.unsqueeze(0).unsqueeze(0)
        out_features = out_features.permute(0, 2, 1) # BatchSize x OutChannels x ChunkSize
```



```
In [30]: sentence_level_model_my_conv = SentenceLevelPOSTagger(len(char_vocab), len(label2id), embedding_size=64,
                                                               single_backbone_kwargs=dict(layers_n=3,
                                                               kernel_size=3,
                                                               dropout=0.3,
                                                               conv_layer=MyConv1d),
                                                               context_backbone_kwargs=dict(layers_n=3,
                                                               kernel_size=3,
                                                               dropout=0.3,
                                                               conv_layer=MyConv1d))
print('Количество параметров', sum(np.product(t.shape) for t in sentence_level_model.parameters()))
Количество параметров 84882
```

```
In [31]: (best_val_loss,
           best_sentence_level_model_my_conv) = train_eval_loop(sentence_level_model_my_conv,
                                                               train_dataset,
                                                               test_dataset,
                                                               F.cross_entropy,
                                                               lr=5e-3,
                                                               epoch_n=10,
                                                               batch_size=64,
                                                               device='cuda',
                                                               early_stopping_patience=5,
                                                               max_batches_per_epoch_train=500,
                                                               max_batches_per_epoch_val=100,
                                                               lr_scheduler_ctor=lambda optim: torch.optim.lr_sch...
```

```
In [32]: train_pred = predict_with_model(best_sentence_level_model_my_conv, train_dataset)
train_loss = F.cross_entropy(torch.tensor(train_pred),
```



```
lr=5e-3,
epoch_n=10,
batch_size=64,
device='cuda',
early_stopping_patience=5,
max_batches_per_epoch_train=500,
max_batches_per_epoch_val=100,
lr_scheduler_ctor=lambda optim: torch.optim.lr_scheduler.Reduc...
```

```
Эпоха 0
Эпоха: 501 итераций, 52.19 сек
Среднее значение функции потерь на обучении 0.07469622310466634
Среднее значение функции потерь на валидации 0.022032036307719674
Новая лучшая модель!
```

```
Эпоха 1
Эпоха: 501 итераций, 52.45 сек
Среднее значение функции потерь на обучении 0.022592827436066196
Среднее значение функции потерь на валидации 0.019129324012832478
Новая лучшая модель!
```

```
Эпоха 2
Эпоха: 501 итераций, 52.56 сек
Среднее значение функции потерь на обучении 0.019180212177321344
Среднее значение функции потерь на валидации 0.015566333690381582
Новая лучшая модель!
```

```
Эпоха 3
Эпоха: 501 итераций, 52.68 сек
Среднее значение функции потерь на обучении 0.017376770992240984
Среднее значение функции потерь на валидации 0.01380048683927496
Новая лучшая модель!
```



Перейдём к оценке качества модели, которая использует наши свёртки. Видим, что она достигает f-меры 0.91, таким образом мы видим, что мы свёртки реализовали правильно, модели работают, всё отлично. Давайте подытожим. В этом семинаре мы рассмотрели, как можно применять свёрточные нейросети для обработки текстов. Мы описали базовую

архитектуру — свёрточный [ResNet](#), а также применили её в разных ситуациях для анализа контекста символов и для анализа контекста токенов. Демонстрировали мы работу свёрточной нейросети на задаче предсказания частей речи токенов. Эта задача относится к области лингвистического анализа, то есть анализа структуры текстов. Основная сложность в этой задаче, обычно, заключается в том, чтобы правильно определять часть речи для омонимов, то есть для слов, которые пишутся одинаково, но, на самом деле, имеют разные части речи.

Другая сложность заключается в том, чтобы правильно определять части речи для неизвестных слов, то есть каких-нибудь неологизмов или специальной редкой лексики (научной, например). В результате семинара мы увидели, что учёт контекста токенов действительно важен в определённых случаях, хотя, чаще всего, часть речи можно определить, просто посмотрев на само слово. Модели, которые мы обучили в этом семинаре — достаточно неплохие, хотя промышленные [POS тэггеры](#) работают с гораздо более высоким качеством (на уровне 0.97 f-мер). Кроме того, мы реализовали свёрточный модуль своими руками и убедились в том, что он работает. На этом на сегодня всё. Пока!

