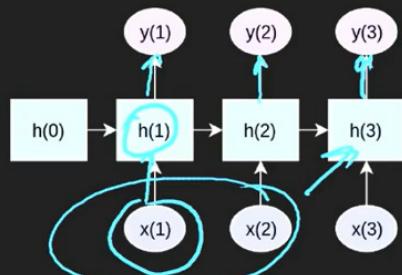


Stepik. Neural networks and NLP. 4. Language models and text generation - Part 1

4.1 Рекуррентные нейросети

Всем привет! Это видео про рабочую лошадку нейросетевых моделей для обработки последовательностей — данных вообще и текста в частности: [рекуррентные нейросети](#). Практически во всех областях обработки текстов были попытки применить [рекуррентки](#) и во многих это получилось сделать успешно. Основная идея рекурренток заключается в том, что у нас есть некоторое внутреннее состояние, которое мы обновляем после прочтения очередного элемента входной последовательности. По идее, это внутреннее состояние должно содержать в себе информацию обо всём прочитанном ранее тексте. В свёртках же — наоборот, мы состояние не хранили, а обрабатывали кусок текста определённого размера — сразу. Рекуррентные нейросети могут использоваться как для предсказания одной величины по тексту (например, для классификации), так и для предсказания каких-либо величин для каждого элемента входной последовательности. Итак, мы читаем входной текст слово за словом или символ за символом и, сначала, обрабатываем его (например, мы можем применить к нему линейное преобразование). Затем мы вычисляем новое значение состояния. В простейшем случае оно зависит от значения состояния на предыдущем шаге и от текущего входа. Далее мы делаем предсказание, используя только текущее значение состояния, а потом читаем следующее слово и повторяем всю процедуру заново. Такие нейросети потенциально гораздо мощнее, чем свёрточные. Однако скрытое состояние зависит от предыдущего, а мы обрабатываем элемент за элементом последовательно, поэтому рекуррентки не так хорошо используют возможности параллельных вычислений в графических ускорителях и, в итоге, медленнее [свёрточных нейросетей](#).



На каждом шаге

- прочитать и обработать очередной элемент входной последовательности
 $z_t = W_{input} \cdot x_t, \quad W_{input} \in \mathbb{R}^{d_{hidden} \times d_{input}}$



На каждом шаге

- прочитать и обработать очередной элемент входной последовательности
 $z_t = W_{input} \cdot x_t, \quad W_{input} \in \mathbb{R}^{d_{hidden} \times d_{input}}$
- вычислить новое значение ячейки памяти, исходя из её старого значения и входного элемента
 $h_t = \tanh(W_{hidden} \cdot h_{t-1} + z_t), \quad W_{hidden} \in \mathbb{R}^{d_{hidden} \times d_{hidden}}$
- вычислить выход
 $y_t = W_{output} \cdot h_t$

- потенциально мощнее, чем свёртки
- последовательная обработка данных → медленнее, чем свёртки



И, самое главное, рекуррентки обучаются не так легко. Здесь появляются проблемы затухания или взрыва градиента. Поэтому современные работы в области рекурренток, в основном, связаны с поиском баланса между мощностью и стабильностью процесса обучения. Давайте попробуем понять, откуда берутся эти самые проблемы — затухание и взрыв градиента. Мы это будем делать на примере классической рекуррентки (такие сети ещё называют "vanilla"). При

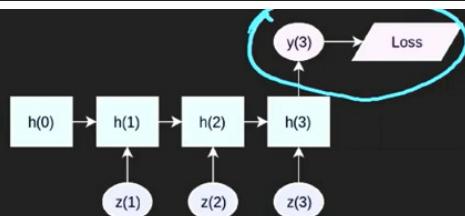
этом мы будем считать, что размерность всех переменных (входных данных и скрытого состояния) равна единице, то есть мы будем работать только со скалярными операциями. Для простоты будем считать, что мы предсказываем одну единственную величину по всему тексту — например, мы решаем задачу классификации и класс предсказываем на основе значения состояния на последнем шаге, то есть после прочтения всего входного текста. Соответственно, предсказание подаётся в функцию потерь или функционал качества, исходя из которого мы будем [градиентным спуском](#) настраивать параметры нейросети. Здесь мы не будем специфицировать, какую конкретно функцию активации мы используем — для того, что мы хотим сделать, это неважно. Итак, до прочтения первого слова у нас уже есть начальное состояние — это тоже параметр сети. Затем мы читаем первое слово и находим новое значение скрытого состояния. Затем — ещё слово, и при этом важно помнить, что h_1 , на самом деле — это функция от начального состояния и первого слова. Ну, и так далее... Таким образом мы получим глубокую композицию функций.

- ❶ прочитать и обработать очередной элемент входной последовательности
 $z_t = W_{input} \cdot x_t, \quad W_{input} \in \mathbb{R}^{d_{hidden} \times d_{input}}$
- ❷ вычислить новое значение ячейки памяти, исходя из её старого значения и входного элемента
 $h_t = \tanh(W_{hidden} \cdot h_{t-1} + z_t), \quad W_{hidden} \in \mathbb{R}^{d_{hidden} \times d_{hidden}}$
- ❸ вычислить выход
 $y_t = W_{output} \cdot h_t$

- потенциально мощнее, чем свёртки
- последовательная обработка данных → медленнее, чем свёртки
- есть проблемы сходимости (затухание и взрыв градиента)
- поиск баланса между мощностью и простотой обучения



Vanilla RNN



Одномерная рекуррентная сеть

- Вход $z_t \in \mathbb{R}$
- Скрытое состояние $h_t = f(w \cdot h_{t-1} + z_t)$
 f - функция активации
 $w \in \mathbb{R}$ - параметр функции перехода
- Выход $y_t = g(h_t)$
- Функционал качества $\text{Loss}(y_t)$

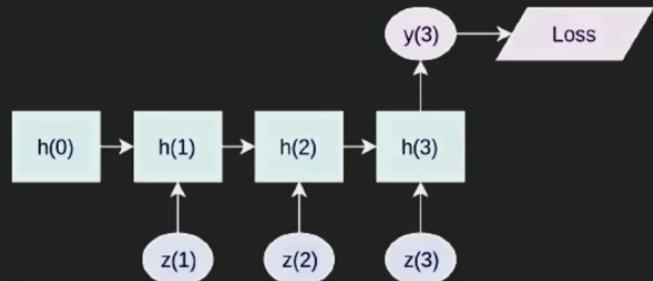
- ❶ $h_0 \in \mathbb{R}$



- Вход $z_t \in \mathbb{R}$
- Скрытое состояние $h_t = f(w \cdot h_{t-1} + z_t)$
 f - функция активации
 $w \in \mathbb{R}$ - параметр функции перехода
- Выход $y_t = g(h_t)$
- Функционал качества $Loss(y_t)$

- ① $h_0 \in \mathbb{R}$
 - ② $h_1 = f(w \cdot h_0 + z_1)$
 - ③ $h_2 = f(w \cdot h_1 + z_2) = f(w \cdot f(w \cdot h_0 + z_1) + z_2)$
 - ④ $h_3 = f(w \cdot h_2 + z_3) = f(w \cdot f(w \cdot f(w \cdot h_0 + z_1) + z_2) + z_3)$
- 

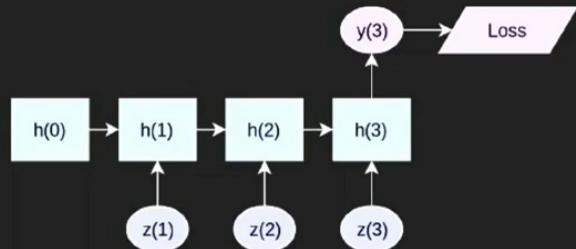
Итак, прямой проход по сети сделали. Ну, пора и домой — то есть обратно. Раскручиваем цепочку вызовов, начиная со значения функции потерь. Предсказание (y) — это функция от последнего состояния, которое, в свою очередь, является функцией предпоследнего состояния и последнего слова, и так далее. Мы применяем градиентный спуск, поэтому попробуем посчитать производную функции потерь по весам нейросети. В первую очередь нас интересует производная по весам рекуррентного блока — в нашем случае это одно число (w). y — это сложная функция, поэтому применяем правило цепочки.



- $\text{Loss}(y_t) = \text{Loss}(g(h_t)) = \text{Loss}(g(f(w \cdot h_{t-1} + z_t))) = \text{Loss}(g(f(w \cdot f(w \cdot h_{t-2} + z_{t-1}) + z_t))) = \dots$
- $\frac{\partial \text{Loss}(y_t)}{\partial w} = \frac{\partial \text{Loss}(y_t)}{\partial g} \frac{\partial g}{\partial w} =$



Мы дошли до производной последнего рекуррентного состояния по весам. Вот это значение равно $h_t \cdot h_{t-1}$. Так так! Давайте здесь остановимся поподробнее. Мы опять берём производную сложной функции, поэтому результат будет равен произведению производной самой функции и производной её аргумента. Производная самой функции нас пока что не сильно интересует, поэтому введём обозначение для краткости, в качестве "f штрих" с нижним индексом t (f'_t). z от w тоже не зависит, поэтому оно сокращается. И тут мы вспоминаем, что h — это функция, которая зависит от w , поэтому мы применяем правило дифференцирования произведения функций. Производная w по w равна единице. Чтобы найти производную предыдущего состояния, раскроем его (получим такое вот выражение). Это выражение очень похоже на то, с чего мы начали. Отлично, давайте тогда просто возьмём и подставим, только индексы, заменим на новые. Мы подставили сюда производную предпоследнего скрытого состояния. Мы можем продолжить эту процедуру дальше до самого первого элемента и начального состояния — тогда получим следующую формулу. Вполне логично — производная w зависит от всех шагов. А ещё, внутри есть произведение всех производных функции активации на нескольких шагах — вот тут-то собака и зарыта.



- $\text{Loss}(y_t) = \text{Loss}(g(h_t)) = \text{Loss}(g(f(w \cdot h_{t-1} + z_t))) =$
 $\text{Loss}(g(f(f(w \cdot h_{t-2} + z_{t-1}) + z_t))) = \dots$

- $\frac{\partial \text{Loss}(y_t)}{\partial w} = \frac{\partial \text{Loss}(y_t)}{\partial g} \frac{\partial g}{\partial w} = \frac{\partial \text{Loss}(y_t)}{\partial g} \frac{\partial g}{\partial f} \frac{\partial f(w \cdot h_{t-1} + z_t)}{\partial w}$

h_t



- $\text{Loss}(y_t) = \text{Loss}(g(h_t)) = \text{Loss}(g(f(w \cdot h_{t-1} + z_t))) =$
 $\text{Loss}(g(f(f(w \cdot h_{t-2} + z_{t-1}) + z_t))) = \dots$

- $\frac{\partial \text{Loss}(y_t)}{\partial w} = \frac{\partial \text{Loss}(y_t)}{\partial g} \frac{\partial g}{\partial w} = \frac{\partial \text{Loss}(y_t)}{\partial g} \frac{\partial g}{\partial f} \frac{\partial f(w \cdot h_{t-1} + z_t)}{\partial w}$
- $\frac{\partial f(w \cdot h_{t-1} + z_t)}{\partial w} = f'(w \cdot h_{t-1} + z_t) \cdot (w \cdot h_{t-1} + z_t)' = f'_t \cdot (w \cdot h_{t-1})' =$
 $= f'_t \cdot (w' \cdot h_{t-1} + w \cdot h'_{t-1}) = f'_t \cdot (h_{t-1} + w \cdot f(w \cdot h_{t-2} + z_{t-1})') =$
 $= f'_t \cdot (h_{t-1} + w \cdot f'_t \cdot (h_{t-2} + w \cdot f(w \cdot h_{t-3} + z_{t-2})')) =$

$$= \sum_{i=1}^t \left(h_{i-1} \cdot w^{t-i} \prod_{j=i}^t f'_j \right)$$



Из практической задачи:

Допустим, $w=1.1$, тогда за 100 шагов в последней формуле на предыдущем видео накопится множитель w^{99} . Найдите его значение. Ответ округлите до целого значения. Напомним:

$$\frac{\partial f(w \cdot h_{t-1} + z_t)}{\partial w} = \sum_{i=1}^t \left(h_{i-1} \cdot w^{t-i} \prod_{j=i}^t f_j' \right)$$

Ответ: $w^{99} = 1.1^{99} = 12528$

Комментарий к решению (из курса): Вот мы промоделировали ситуацию взрыва градиента. А что если $w = 10$ и у нас не 100 шагов, а 200?

Часто в качестве функции активации используется гиперболический тангенс. Его производная лежит в диапазоне от нуля до единицы. Если мы возьмём много таких чисел, лежащих от нуля до единицы, и перемножим их, мы получим значение, очень близкое к нулю. Это приводит к затуханию градиента — информация с первых шагов почти никак не учитывается при вычислении обновления весов. И в этом случае весь смысл использования рекуррентности исчезает. И, наоборот, если модуль вот этой части больше единицы, то, возводя степень, мы получим очень большое по модулю число, что приводит к переполнению и катастрофическому падению точности вычислений. Со взрывом градиента борются очень просто — сначала честно считают градиентные шаги для всех параметров, а потом, если какой-то градиент по модулю превышает некоторый порог, то он заменяется на значение порога со знаком. То есть слишком большие градиенты просто обрезаются.^[1,2] Бороться с затуханием градиента гораздо сложнее — этому посвящено множество работ. Давайте рассмотрим парочку.

[1] <https://machinelearningmastery.com/how-to-avoid-exploding-gradients-in-neural-networks-with-gradient-clipping/>

[2] <http://www.wildml.com/deep-learning-glossary/>

Дополнительный комментарий от Романа Суворова:

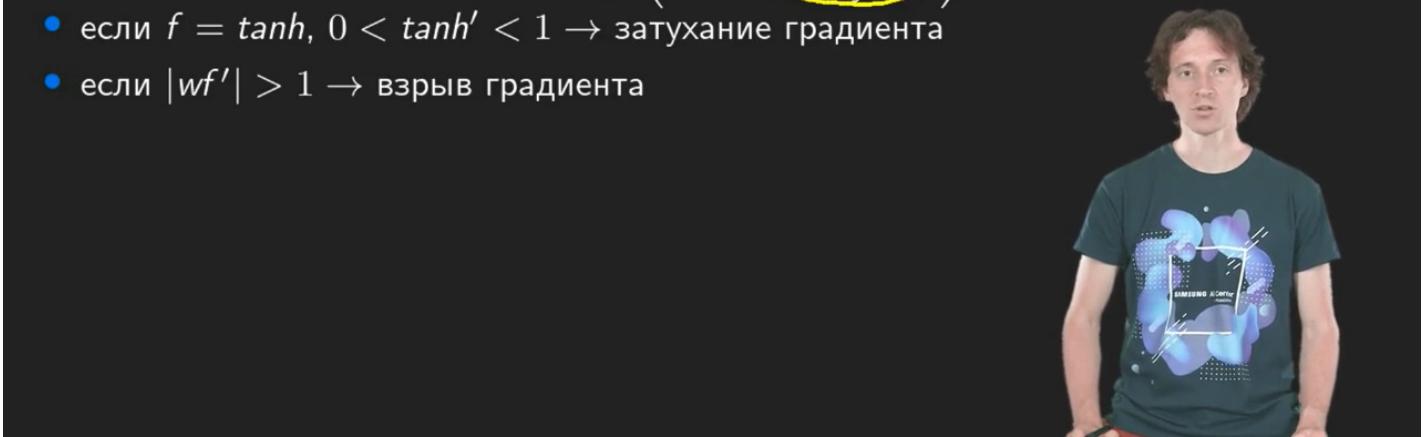
Подробнее про gradient clipping:

Градиент - это вектор направления. Если его клипать ровно как говорится в видео (то есть каждый элемент вектора независимо обрезать, как это делает, например, `tensor.clamp` в `pytorch`), то после клиппинга вектор будет указывать в другом направлении и процесс обучения может перестать сходиться. Например, если изначально градиент был [0.1, 100], то после клиппинга с порогом 1 он станет [0.1, 1]. В PyTorch есть две функции для gradient clipping: `clip_grad_value` - делает ровно то, что говорится в лекции (заменяет значение градиента на порог, если значение превосходит порог по модулю). При этом направление вектора меняется. `clip_grad_norm` - масштабирует весь вектор градиента так, чтобы его норма (считай длина) не превышала заданный порог. При этом направление вектора градиента **не** меняется. Что лучше использовать - надо пробовать в каждом конкретном случае. `clip_grad_norm` более корректный математически, но может приводить к тому, что обучение будет слишком

медленным, а [clip_grad_value](#) может приводить к расхождению процесса обучения (далеко не всегда), но не замедляет этот процесс.

- $$\begin{aligned} \frac{\partial f(w \cdot h_{t-1} + z_t)}{\partial w} &= f'(w \cdot h_{t-1} + z_t) \cdot (w \cdot h_{t-1} + z_t)' = f'_t \cdot (w \cdot h_{t-1})' = \\ &= f'_t \cdot (w' \cdot h_{t-1} + w \cdot h'_{t-1}) = f'_t \cdot (h_{t-1} + w \cdot f(w \cdot h_{t-2} + z_{t-1})') = \\ &= f'_t \cdot (h_{t-1} + w \cdot f'_{t-1} \cdot (h_{t-2} + w \cdot f(w \cdot h_{t-3} + z_{t-2})')) = \\ &= \sum_{i=1}^t \left(h_{i-1} \cdot w^{t-i} \prod_{j=i}^t f'_j \right) \end{aligned}$$

- если $f = \tanh$, $0 < \tanh' < 1 \rightarrow$ затухание градиента
- если $|wf'| > 1 \rightarrow$ взрыв градиента



Долгосрочная-краткосрочная память — самый часто используемый вид рекурренток.

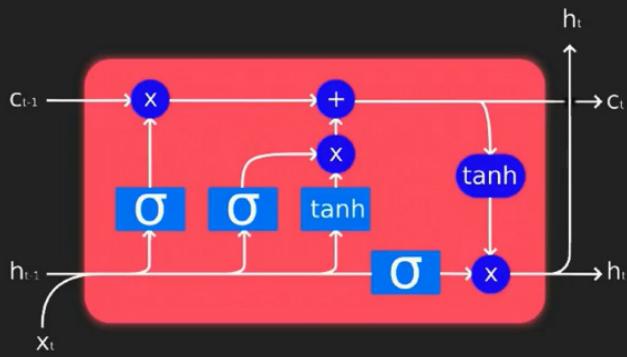
Придумал его в конце девяностых годов коллектив знаменитого Юргена Шмидтхубера.^[1] На первый взгляд, это очень сложная нейросеть. Да, на самом деле, [LSTM](#) выглядит сложной и на второй, и на третий, и на последующие взгляды. Но в её основе лежит простая и понятная идея. Цель — сохранение объёма потока ошибки.^[2] Предположим, что у нас есть труба, и через неё текут градиенты. И мы хотим, чтобы площадь сечения этой трубы была примерно одинаковой, чтобы у нас не было узких горлышек и очень широких областей. Юрген и ребята придумали так называемую "карусель постоянного объёма ошибки".^[2] Это рекуррентность особого вида. Самая главная фишка — в том, что при переходе от предыдущего значения к текущему мы не используем функцию активации, а вместо этого используем операцию сложения. При этом, значения вектора "с" могут быть абсолютно любыми. На значение вектора "с" влияет его предыдущее состояние и ещё 3 сущности. Первая сущность, вектор "g", несёт физический смысл "направления" — куда будет сдвигаться новый вектор "с" относительно его предыдущего значения. При вычислении "g" применяется гиперболический тангенс, поэтому диапазон возможных значений от -1 до +1. Есть ещё две сущности — "f" и "e", они отвечают за амплитуду изменения. Вектор "e" отвечает за чувствительность к "g". Он вычисляется через [сигмоиду](#), поэтому его значения лежат в диапазоне от 0 до 1. Таким образом, он может ослабить влияние "g" на изменение вектора "с". Вектор "e" ещё называют

входным шлюзом или "input gate". Вектор "f" отвечает за чувствительность к предыдущему значению рекуррентного состояния. Он также вычисляется через сигмоиду и лежит в диапазоне от 0 до 1. Вектор "f" ещё называют "шлюзом забывания" или "forgetting gate" — он позволяет совершать резкие изменения скрытого состояния и игнорировать всё, что было до определённого момента. И всё-таки, чтобы повысить мощность данной нейросети, к вектору "c" мы применяем нелинейность, чтобы получилось ещё одно рекуррентное состояние — на этот раз с более сложным преобразованием. Всего в LSTM два рекуррентных вектора — "c" и "h". Здесь применяется всё та же идея "[гейтинга](#)", когда вектор значений умножается на вектор шлюза, который управляет потоком или амплитудой значений, но не может изменить знак. Вообще "гейтинг" — это один из ключевых приёмов в [рекуррентках](#) и в глубоких нейросетях в целом, к нему стоит присмотреться повнимательней. И, напоследок — количество параметров. Если размер скрытого состояния — ddd , то количество параметров сети имеет порядок $8d28 d^28d2$ — это очень много. В результате, LSTM часто [переобучаются](#) и плохо обобщаются на новые данные (но не всегда).

[1] Long Short-Term Memory, Sepp Hochreiter and Jürgen Schmidhuber, Neural Computation 1997 9:8, 1735-1780

[2] Constant Error Carousel <https://deeppai.org/machine-learning-glossary-and-terms/constant%20error%20carousel>

Из комментариев: Хорошая статья для понимания LSTM <https://neurohive.io/ru/osnovy-data-science/lstm-nejronnaja-set/>



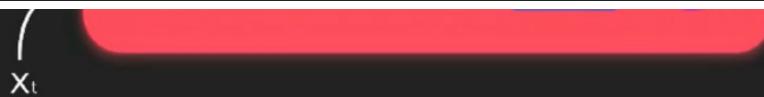
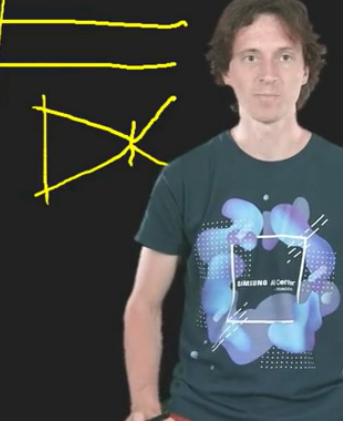
$$\begin{aligned}
 i_t &= \sigma(W_{ii}x_t + W_{hi}h_{(t-1)} + b_i) \\
 f_t &= \sigma(W_{if}x_t + W_{hf}h_{(t-1)} + b_f) \\
 g_t &= \tanh(W_{ig}x_t + W_{hg}h_{(t-1)} + b_g) \\
 o_t &= \sigma(W_{io}x_t + W_{ho}h_{(t-1)} + b_o) \\
 c_t &= f_t * c_{(t-1)} + i_t * g_t \\
 h_t &= o_t * \tanh(c_t)
 \end{aligned}$$

$i_t, f_t, g_t, o_t, c_t, h_t \in \mathbb{R}^d$



- рекуррентность особого вида - поток ошибки постоянного объёма:

$$c_t = f_t * c_{(t-1)} + i_t * g_t, \quad -\inf < c_t < +\inf$$



$$i_t, f_t, g_t, o_t, c_t, h_t \in \mathbb{R}^d$$

- рекуррентность особого вида - поток ошибки постоянного объема:

$$c_t = f_t * c_{(t-1)} + i_t * g_t, \quad -\inf < c_t < +\inf$$

- направление изменения внутреннего состояния:

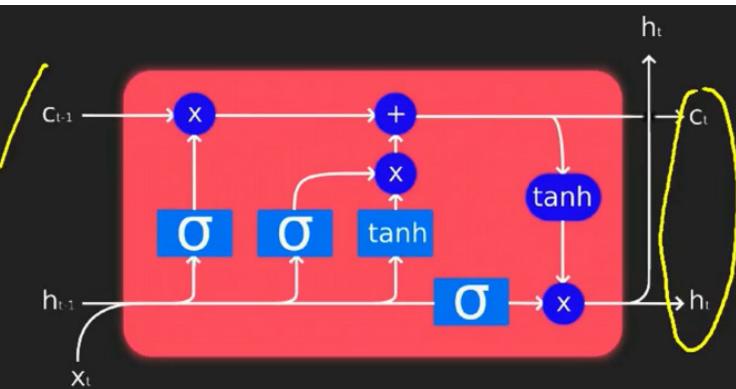
$$g_t = \tanh(W_{ig}x_t + W_{hg}h_{(t-1)} + b_g), \quad -1 < g_t < 1$$

- амплитуда изменения внутреннего состояния:

- чувствительность ко входу $i_t = \sigma(W_{ii}x_t + W_{hi}h_{(t-1)} + b_i)$, $0 < i_t < 1$



- рекуррентность особого вида - поток ошибки постоянного объема:
 $c_t = f_t * c_{(t-1)} + i_t * g_t, \quad -\inf < c_t < +\inf$
- направление изменения внутреннего состояния:
 $g_t = \tanh(W_{ig}x_t + W_{hg}h_{(t-1)} + b_g), \quad -1 < g_t < 1$
- амплитуда изменения внутреннего состояния:
 - чувствительность ко входу $i_t = \sigma(W_{ii}x_t + W_{hi}h_{(t-1)} + b_i), \quad 0 < i_t < 1$
 - чувствительность к памяти $f_t = \sigma(W_{if}x_t + W_{hf}h_{(t-1)} + b_f), \quad 0 < f_t < 1$



$$\begin{aligned}
 i_t &= \sigma(W_{ii}x_t + W_{hi}h_{(t-1)} + b_i) \\
 f_t &= \sigma(W_{if}x_t + W_{hf}h_{(t-1)} + b_f) \\
 g_t &= \tanh(W_{ig}x_t + W_{hg}h_{(t-1)} + b_g) \\
 o_t &= \sigma(W_{io}x_t + W_{ho}h_{(t-1)} + b_o) \\
 c_t &= f_t * c_{(t-1)} + i_t * g_t \\
 h_t &= o_t * \tanh(c_t) \\
 i_t, f_t, g_t, o_t, c_t, h_t &\in \mathbb{R}^d
 \end{aligned}$$

- рекуррентность особого вида - поток ошибки постоянного объема:
 $c_t = f_t * c_{(t-1)} + i_t * g_t, \quad -\inf < c_t < +\inf$
- направление изменения внутреннего состояния:
 $g_t = \tanh(W_{ig}x_t + W_{hg}h_{(t-1)} + b_g), \quad -1 < g_t < 1$
- амплитуда изменения внутреннего состояния:
 - чувствительность ко входу $i_t = \sigma(W_{ii}x_t + W_{hi}h_{(t-1)} + b_i), \quad 0 < i_t < 1$
 - чувствительность к памяти $f_t = \sigma(W_{if}x_t + W_{hf}h_{(t-1)} + b_f), \quad 0 < f_t < 1$
- выход из рекуррентного шага:
 - $h_t = o_t * \tanh(c_t), \quad 0 < h_t < 1$

• рекуррентность особого вида - поток ошибки постоянного объема

$$c_t = f_t * c_{(t-1)} + i_t * g_t, \quad -\inf < c_t < +\inf$$

- направление изменения внутреннего состояния:

$$g_t = \tanh(W_{ig}x_t + W_{hg}h_{(t-1)} + b_g), \quad -1 < g_t < 1$$

- амплитуда изменения внутреннего состояния:

- чувствительность ко входу $i_t = \sigma(W_{ii}x_t + W_{hi}h_{(t-1)} + b_i)$, $0 < i_t < 1$
- чувствительность к памяти $f_t = \sigma(W_{if}x_t + W_{hf}h_{(t-1)} + b_f)$, $0 < f_t < 1$

- выход из рекуррентного шага:

- $h_t = o_t * \tanh(c_t)$, $0 < h_t < 1$
- $o_t = \sigma(W_{io}x_t + W_{ho}h_{(t-1)} + b_o)$, $0 < o_t < 1$

- количество параметров $8d^2 + 4d$

¹ Изображение из Wikipedia изменено <https://en.wikipedia.org/wiki/LSTM>



Из комментариев:

Вопрос:

Какие методы регуляризации применяются в рекуррентных сетях? Особенно в LSTM и GRU?

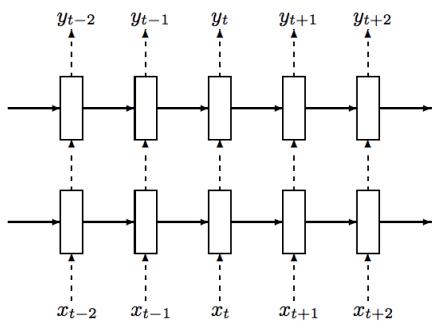
Используют ли батч-нормализация и дропаут?

Насколько я понимаю, дропаут обратных связей сводит смысл и преимущество LSTM на нет. Но можно же делать дропаут между слоями LSTM если они настеканы друг на друга?

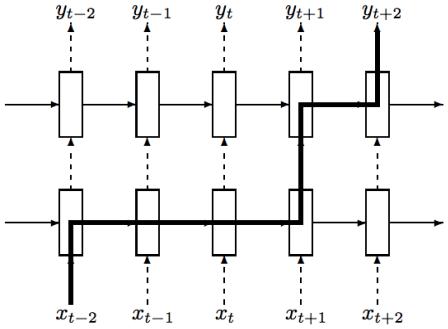
Ответ (от студента):

можно использовать [рекуррентный dropout](#) (все иллюстрации далее взяты оттуда).

Идея следующая: давайте будем применять dropout только к "нерекуррентным" связям (отмечены пунктирной линией на рисунке)



Таким образом, мы не ломаем механизм памяти. Например, на следующем рисунке показано, как информация может "течь" от t-2 шага к предсказанию на t+2 шаге:



Доп. вопрос (от студента):

спасибо! A batch normalization?

Ответ (от Романа Суворова):

batch norm в рекуррентках непонятно как считать, обычно используют [LayerNorm](#) (мы используем его в семинаре про трансформер, но подробно не разбираем)

Из практического задания:

Допустим, скрытое состояние вычисляется по формуле $h_t = Wh_{t-1}$, где $h_t, h_{t-1} \in \mathbb{R}^d$ - новое состояние и предыдущее, а $W \in \mathbb{R}^{d \times d}$ - матрица перехода, Wh - операция матричного произведения матрицы на вектор-столбец.

Можно ли вычислить хотя бы некоторые компоненты вектора h_t до того как все компоненты h_{t-1} будут вычислены?

Ответ: Нельзя, потому что каждый элемент $h(t)$ зависит от всех компонентов $h(t-1)$. Это приводит к тому, что LSTM и GRU применяются к последовательностям строго последовательно :).

Нашлись ребята, которые решили упростить [LSTM](#), и придумали "gated recurrent unit" или "грушку". Ключевая идея здесь ровно такая же — поток ошибки постоянного объёма. Однако теперь потоком ошибки управляют не два шлюза, а один — по сути, этот шлюз на каждом шаге осуществляет выбор между двумя альтернативами: оставить предыдущее значение, или обновить. Обновление вычисляется похожим образом (тоже с тангенсом). Однако тут внутри есть ещё один шлюз, который управляет чувствительностью вектора "g" к предыдущему значению состояния. В результате, количество параметров сократилось на третью — ну что ж неплохо. На практике GRU и LSTM, в большинстве задач, работают практически одинаково и дают очень близкое качество. То есть сеть упростила, стала учиться лучше, но при этом осталось достаточно мощной. Кажется, одна из проблем рекурренток частично решена. Однако есть вторая проблема — скорость. По-прежнему, [рекуррентки](#) относительно плохо распараллеливаются — в первую очередь из-за вот этих вот зависимостей. В результате каждый элемент вектора скрытого состояния зависит от всего вектора предыдущего состояния. Это заметили авторы ещё одного вида рекурренток, который называется "simple recurrent unit".

[1] Они заменили матрицу и матричное произведение на вектор и поэлементное произведение. Таким образом, теперь можно параллельно вычислять значение разных элементов рекуррентных векторов в рамках одного шага. С учётом того, что, на практике,

размерность этих векторов составляет от нескольких сотен до пары тысяч, это даёт очень хороший прирост производительности. Кроме того, в этой сети ещё в два раза меньше параметров, поэтому она ещё меньше переобучается. Да, она слабее, чем LSTM, но, на практике, за счёт более простой структуры, процесс обучения идёт более эффективно и качество решения задачи остаётся прежним, или может даже немного улучшится. В этом видео мы разобрались, что такое рекуррентные нейросети, какие сложности в процессе их обучения возникают — в первую очередь это взрыв градиента (тогда градиенты просто отсекают^[2]) и затухание градиента. Для борьбы со второй проблемой придумывают специальные архитектуры. Самая старая, проверенная и популярная — LSTM. Относительно недавно был предложен облегченный вариант — "грушка". И ещё более недавно придумали, как можно упростить и, при этом, ускорить рекуррентки — примером такой работы является simple recurrent unit.

[1] Lei T. et al. Simple recurrent units for highly parallelizable recurrence //arXiv preprint

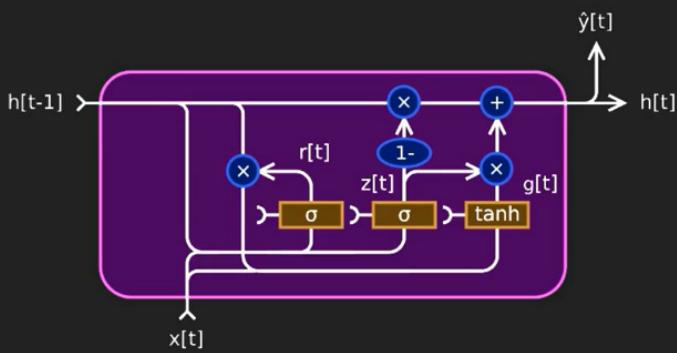
arXiv:1709.02755. – 2017. (<https://arxiv.org/abs/1709.02755>)

[2] How to Avoid Exploding Gradients With Gradient Clipping

<https://machinelearningmastery.com/how-to-avoid-exploding-gradients-in-neural-networks-with-gradient-clipping/>

Дополнительные комментарии к видео (от Романа Суворова):

- Формула на видео обрезана, полную формулу можно посмотреть [в документации PyTorch](#).



$$\begin{aligned}
 r_t &= \sigma(W_{ir}x_t + W_{hr}h_{(t-1)} + b_r) \\
 z_t &= \sigma(W_{iz}x_t + W_{hz}h_{(t-1)} + b_z) \\
 g_t &= \tanh(W_{in}x_t + b_{in} + r_t * (W_{hn}h_{(t-1)} + b_{hn})) \\
 h_t &= z_t * h_{(t-1)} + (1 - z_t) * g_t \\
 r_t, z_t, g_t, h_t &\in \mathbb{R}^d
 \end{aligned}$$

- рекуррентность особого вида - поток ошибки постоянного объёма:

$$h_t = z_t * h_{(t-1)} + (1 - z_t) * g_t, \quad -\inf < h_t < \inf$$
- выбор "оставить или обновить" состояние:

$$z_t = \sigma(W_{iz}x_t + W_{hz}h_{(t-1)} + b_z), \quad 0 < z_t < 1$$
- направление изменения внутреннего состояния:

$$g_t = \tanh(W_{in}x_t + b_{in} + r_t * (W_{hn}h_{(t-1)} + b_{hn})), \quad -1 < g_t < 1$$
- чувствительность к памяти:

$$r_t = \sigma(W_{ir}x_t + W_{hr}h_{(t-1)} + b_r), \quad 0 < r_t < 1$$
- количество параметров $6d^2 + 6d$



$$\begin{aligned}
 f_t &= \sigma(W_f x_t + v_f * c_{(t-1)} + b_f) \\
 r_t &= \sigma(W_r x_t + v_r * c_{(t-1)} + b_r) \\
 c_t &= f_t * c_{(t-1)} + (1 - f_t) * (W_c x_t) \\
 h_t &= r_t * c_t + (1 - r_t) * x_t \\
 f_t, r_t, c_t, h_t &\in \mathbb{R}^d
 \end{aligned}$$

- элементы векторов состояния зависят только от тех же элементов с предыдущего шага

$$c_t = \{c_t^i\}, \quad c_t^i = Q(c_{t-1}^i, x_t), \quad c_t^i \perp c_{t-1}^j, \quad \forall i, j : j \neq i$$
- значения c_t^i, h_t^i для разных i можно вычислять параллельно → ускорение в 5-9 раз относительно LSTM
- количество параметров $3d^2 + 4d$



- рекуррентные нейросети
- сложности в ходе обучения рекуррентных нейросетей
- LSTM - наиболее часто используемый рекуррентный блок
- GRU - облегчённый вариант LSTM
- Simple Recurrent Unit - наиболее лёгкий вариант, улучшенный параллелизм



4.2 Моделирование языка

Всем привет! Сегодня мы поговорим об одной из фундаментальных задач в обработке текстов: [моделировании языка](#). Эта задача, в некотором смысле, особенная, потому что с ней связаны недавние революционные изменения в области обработки текстов — а именно, существенное улучшение качества решения задач и демократизация этой области. Благодаря тому, что веса предобученных моделей выложены в интернете и код доступен, любой может улучшить свои результаты, просто подключив одну из доступных моделей. Ну что ж, поехали! На вход поступает токенизованный текст. Мы будем рассматривать такой текст как "сэмпл" из некоторого неизвестного распределения. В русскоязычной литературе это ещё называют "реализации случайной величины". Мы строим модель, которая должна уметь, для заданного фрагмента текста, оценивать вероятность встретить такой фрагмент в настоящем тексте. Другими словами, модель предсказывает, насколько данный фрагмент похож на настоящий текст, или — насколько он правдоподобен. В общем виде, такая задача не является уникальной именно для автоматической обработки текстов. Это одна из основных задач от мат.статистики, она называется "[оценка плотности распределения](#)". В контексте машинного обучения это является одной из возможных постановок задач [обучения без учителя](#). Если у нас есть вычислимая оценка плотности распределения, то мы можем сэмплировать из этого распределения — то есть, в нашем случае, мы можем генерировать тексты с помощью

языковых моделей. Ну, и зачем нам это всё надо? Одно из самых очевидных применений таких моделей — это проверка и коррекция ошибок в текстовых редакторах.^[1] Пользователь набирает текст, мы оцениваем правдоподобие этого текста с помощью нашей модели, находим слова, более всего снижающие правдоподобие текста, и подсвечиваем их как возможное место, где есть ошибка. Для примерно такой же цели, языковые модели применяются ещё в распознавании речи, то есть в преобразовании аудиозаписи голоса в текст, и в машинном переводе. В таких задачах решение часто строится из двух шагов. Первый шаг переводит данные из одного домена в другой — например, из аудио в текст, а другой шаг работает только во втором домене и исправляет ошибки, допущенные на первом шаге.

Например, если мы слышим "let it be", то у нас есть два похожих варианта написания (похожих по звучанию, конечно). Вот, как раз, языковая модель должна выбрать, какой из этих двух вариантов более правдоподобен с точки зрения языка. Для обучения моделей для распознавания речи или для перевода требуется параллельный [корпус](#), то есть датасет из пар: аудио и результирующий текст или пар предложений на двух языках. Объёмы таких корпусов относительно небольшие, особенно в сравнении с общим количеством документов в интернете. Модель, обученная без учителя только на одном домене, знает про этот домен гораздо больше, чем модель с учителем, но обученная на небольшом корпусе. Надо научиться эффективно делать обучение без учителя. В последнее время одно из наиболее активных направлений работы в области нейросетей — [перенос знаний](#) или перенос навыков.

Англоязычный термин: transfer learning. Мы можем обучить большую модель на большом корпусе — предпочтительно, без учителя (если можем, конечно). А потом — использовать небольшой корпус, предназначенный для одной конкретной задачи, чтобы "[донастроить](#)" полученную большую модель именно под эту задачу. Языковые модели — это основа переноса навыков в обработке текстов. Это как нейросети, обученные на ImageNet, если проводить аналогию с областью компьютерного зрения.

[1] <https://habr.com/ru/post/108831/>

Токенизированный текст $[t_1, t_2, \dots, t_l] \sim P(t_1, t_2, \dots, t_l)$

Оценка плотности вероятности $Q(t_1, t_2, \dots, t_l) - ?$

- Насколько данный фрагмент текста похож на настоящий текст на ЕЯ?
- Насколько данный фрагмент текста выглядит правдоподобно?
- Оценка плотности распределения (density estimation)
- Предсказание следующего слова, генерация текста



Исправление ошибок (proofreading)

Выбор наиболее правдоподобных вариантов при генерации текста

- Распознавание речи (speech to text)
- Машинный перевод
- Let it be? vs Let eat bee?

Использование неразмеченных данных для обогащения моделей машинного обучения

- Перенос знаний, перенос навыков (transfer learning)
- Предобучение (pretraining), затем дообучение (fine-tuning) или извлечение признаков (feature extraction)



Итак, давайте попробуем разобраться в подходах к [моделированию языка](#). Датасет состоит просто из текстов. Мы считаем, что эти тексты генерирует некоторая "случайная величина" с неизвестным многомерным распределением. Давайте вспомним про "правило цепочки" — то, как можно представить совместное распределение нескольких случайных величин в виде произведения условных распределений. Причём мы можем разворачивать эту цепочку в

любую сторону, как нам удобно. Порядок, в котором записывается условные распределение, называется порядком факторизации (англоязычный термин factorization order). В зависимости от того, как мы раскручиваем эту цепочку, мы можем получить разные постановки задачи моделирования языка. Классическая и одна из самых популярных постановок — это [авторегрессионные языковые модели](#), когда мы предсказываем слово за словом. В этом случае модель аппроксимирует такое вот условное распределение: вероятность следующего слова при наблюдении какого-то количества предыдущих слов. Вот так, двоеточием, мы будем сокращённо обозначать последовательности нескольких токенов с номерами, от значения первого индекса до второго. К этой группе относятся популярные модели — такие, как [ELMo](#), [OpenAI Transformer^{\[1\]}](#), [XLNet](#) — мы поговорим о них чуть позже. Другая постановка подразумевает отсутствие жёстко заданного порядка факторизации. Такие модели предсказывают слова в середине фрагмента текста по остальным словам. К таким моделям относится [word2vec](#), которую мы уже разбирали ранее в курсе, а также современные модели [BERT](#) и XLNet.

[1] Попробуйте пообщаться с трансформером: <https://talktotransformer.com/> (лучше, на английском)

- Даны тексты $[t_1, t_2, \dots, t_l] \sim P(t_1, t_2, \dots, t_l)$, P неизвестно
- Правило цепочки (chain rule)

$$b \leftarrow c \leftarrow a$$

$$P(a, b, c) = \underbrace{P(a|b, c)}_{\text{Chain Rule}} \underbrace{P(b|c)}_{\text{Chain Rule}} P(c) = P(b|a, c) P(c|a) P(a)$$



- Даны тексты $[t_1, t_2, \dots, t_l] \sim P(t_1, t_2, \dots, t_l)$, P неизвестно
- Правило цепочки (chain rule)

$$P(a, b, c) = P(a|b, c) P(b|c) P(c) = P(b|a, c) P(c|a) P(a)$$

- Предсказание следующего слова по первым словам (autoregressive LM, ELMo, OpenAI GPT, XLNet)

$$P(t_1, t_2, \dots, t_l) = P(t_l | t_1, t_2, \dots, t_{l-1}) P(t_{l-1} | t_1, t_2, \dots, t_{l-2}) \dots P(t_1)$$

$$P(t_1, t_2, \dots, t_l) = \prod_{i=1}^l P(t_i | t_{1:i-1})$$

- Предсказание слова в середине по остальным словам $P(t_k | t_1, \dots, t_{k-1}, t_{k+1}, t_l)$ (Word2Vec CBOW, BERT, XLNet)



Рассмотрим простейшую авторегрессионную модель — N-граммную модель. Мы хотим сделать алгоритм, который будет предсказывать очередное слово по известным предыдущим. Задача сложная, и мы упрощаем её, ограничивая длину истории некоторым числом N. Мы считаем, что слова, которые были ранее, чем за N слов от текущего, не влияют на распределение вероятностей текущего слова, то есть — решение о следующем слове мы

принимаем только на основе последних N слов. Это ещё называют свойством "марковости N-го порядка". Если N равно единице, то мы получаем простейшую марковскую цепочку: на очередное слово влияет только одно лишь предыдущее слово. В классических N-граммных моделях вероятности хранятся в явном виде с помощью хэш-таблицы или префиксово дерева, а процесс обучения заключается в подсчёте частот и нормирующих коэффициентов, чтобы, затем, мы могли посчитать условную вероятность по вот такой вот формуле. Такие модели достаточно просты в реализации, быстро работают и долгое время ничего кроме них и не было. Но, естественно, есть ряд сложностей. Во-первых, при увеличении N, количество N-грамм соответствующей длины растёт в геометрической прогрессии — поэтому на практике редко используют N больше 3. В результате, модель получается большая и разреженная. Достаточно сложно полученные представления использовать как признаки в алгоритмах машинного обучения. В целом, ситуация аналогична классическим [дистрибутивно-семантическим моделям](#), об этом мы уже говорили ранее. Другая проблема заключается в том, что если мы используем [N-граммы](#) слов, то мы не можем работать с неизвестными словами, значений для них просто нет в наших таблицах, у нас нет средства чтобы подобрать числа для похожих слов. Частично эта проблема решается переходом к работе на уровень символов. То есть, теперь T — это не слова, а отдельные символы, буковки. Но у такого подхода есть и обратная сторона — чтобы моделировать зависимости хотя бы на три слова назад, нам нужно работать с длинными N-граммами, где N равно 20. Например, если средняя длина слова равна 5...6, то N, при переходе с уровня слов на уровень символов, вырастает в шесть раз. А с ростом N, работать с такой моделью становится всё сложнее, всё дороже с вычислительной точки зрения. А ещё, такие модели ничего не знают про синтаксический анализ, про сложную многосвязную структуру текста. Всё, что они видят — это только сколько-то слов слева от текущего. Другими словами они не особо-то выразительны.

Модель

- $P(t_1, t_2, \dots, t_l) = \prod_{i=1}^l P(t_i|t_{1:i-1})$
- Длина истории ограничена n : $P(t_i|t_{1:i-1}) = P(t_i|t_{i-n:i-1})$
- Свойство Марковости n -го порядка
- Все вероятности хранятся в модели в явном виде (с помощью хэш-таблицы или префиксного дерева)
- Обучение - подсчёт частот $P(t_i|t_{i-n:i-1}) = \frac{P(t_{i-n:i})}{P(t_{i-n:i-1})}$

Преимущества

- Простота реализации

Проблемы

- Обобщение, получение сжатого представления
- Работа с большими словарями
- Работа с неизвестными словами (out of vocabulary, OOV)
- Учет сложного контекста



Один из прорывов в этой области произошел в районе 2013 года, когда Миколав и компания придумали [word2vec](#), а именно две модели: SkipGram и Continuous Bag-of-words ([CBoW](#)). Модель SkipGram ориентирована на предсказание соседних слов в некотором окне при условии наблюдения центрального слова. Модель CBoW (Continuous Bag-of-words) работает в обратном направлении — она предсказывает центральное слово по остальным словам, в рамках окна. Принципиальное отличие от N-граммной модели заключается в том, что вероятности больше не хранятся в явном виде — они вычисляются через произведение векторов. Для каждого слова мы храним два таких вектора. Один из них мы используем, когда слово выступает в роли условия, то есть находится справа от вертикальной черты, а второй — когда мы предсказываем это слово, то есть когда хотим оценить вероятность его появления. Вероятности, при этом, считаются через софтмакс от скалярных произведений этих векторов. Однако если словарь большой — порядка сотен тысяч слов, честный софтмакс считать очень дорого из-за этой суммы в знаменателе, и поэтому используют одну из аппроксимаций — например, "[negative sampling](#)", когда знаменатель оценивается по небольшому количеству случайно выбранных слов. Есть и другие аппроксимации — например, иерархический софтмакс^[1]. По сути, модель языка решает задачу классификации, в которой количество классов равно размеру словаря. Настраиваются модели word2vec с помощью градиентного спуска и в качестве целевой функции выступает [кросс-энтропия](#). Когда модель обучена, мы можем оценивать правдоподобие текста по такой вот формуле, предложенной авторами. Самое главное преимущество word2vec — это его простота. Любой человек с ноутбуком может скачать википедию^[2] на своём языке и за пол-дня обучить модель, получить признаки

и начать их использовать в своей задаче. word2vec даёт сжатое представление, которое можно использовать как признаки в задачах машинного обучения. Такие вектора содержат какую-то семантическую информацию. Помните арифметику смыслов? Например, выражение "король минус мужчина плюс женщина" примерно равно "королеве". Благодаря предложенным аппроксимациям [софтмакса](#) — negative sampling или иерархический софтмакс — word2vec может эффективно работать с очень большими словарями, до нескольких миллионов слов. Однако, из коробки, word2vec не умеет работать с неизвестными словами. Зато [FastText](#) — умеет (который, по сути, тот же самый word2vec, только на [N-граммах](#) символов). Так что это можно не считать большой проблемой. Фундаментальное ограничение word2vec заключается в том, что, в результате обучения, мы получаем только один вектор для слова — и всё. А ведь одно слово может иметь несколько смыслов и смысл может меняться в зависимости от контекста... Другими словами, представления слов, которые получаются с помощью word2vec — они не зависят от контекста. Вот это уже проблема. Вот тут на сцену выходят [авторегрессионные языковые модели](#) на основе нейросетей. Скелет таких моделей примерно одинаковый, всегда. Сначала каждый токен по-отдельности преобразуется в вектор, получается embedding токена без учёта контекста, затем, с помощью рекуррентных, свёрточных или архитектур с вниманием происходит учёт контекста, а затем, на последнем слое, предсказываются вероятности слов. Чаще всего в последнее время используется честный софтмакс, а не его аппроксимация, так как использование приближений несколько ухудшает результаты. Поэтому важно не раздувать словарь слишком сильно — на практике, многие большие языковые модели работают со словарями размера не больше пятидесяти тысяч уникальных токенов. Очевидно, что при этом мы отбрасываем очень большое количество слов. Самый очевидный ход для решения этой проблемы — работать не с целыми словами, а с символами. Однако это очень сильно растягивает последовательности и моделям требуется помнить гораздо более длинную историю. На практике, символьные модели работают не так хорошо, как модели на уровне токенов, поэтому последнее время ищут другие способы.

[1] [Hierarchical softmax and negative sampling: short notes worth telling](#)

[2] <https://dumps.wikimedia.org/>

Модель

- Skip-Gram $P(t_{k+m}|t_k)$, $m = \{-d, -d+1, -d+2, \dots, d\}$
- CBOW $P(t_k|t_{k-d}, \dots, t_{k-1}, t_{k+1}, \dots, t_{k+d})$
- Два вектора для каждого слова ("центральный" w_i , "контекстный" d_i)
- Вероятности рассчитываются через скалярное произведение векторов

$$P(t_j|t_i) = \frac{e^{w_i \cdot d_j}}{\sum_{j=1}^{|V|} e^{w_i \cdot d_j}} = \text{softmax} \simeq \text{neg.sampling} = \frac{e^{w_i \cdot d_j^+}}{\sum_{j=1}^M e^{w_i \cdot d_j^-}}, \quad M \ll |V|$$

- Обучение градиентным спуском, функция потерь - кросс-энтропия
- Классификация, количество меток равно размеру словаря
- Правдоподобие текста по Skip-Gram $P(t_1, t_2, \dots, t_l) \approx \prod_{i=1}^l \prod_{j=1, i \neq j}^l P(t_j|t_i)$

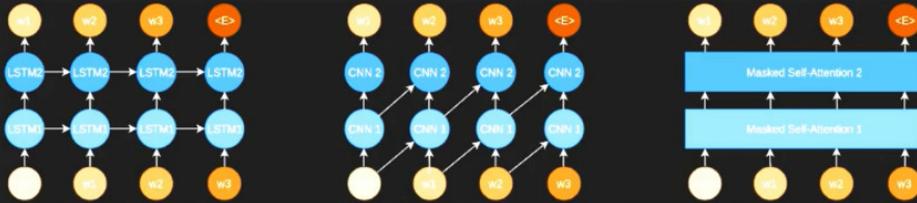


Преимущества

- Простота
- Сжатое представление, обобщение
- Работа с большими словарями

Проблемы

- Работа с неизвестными словами (OOV) - **FastText**
- Учет сложного контекста



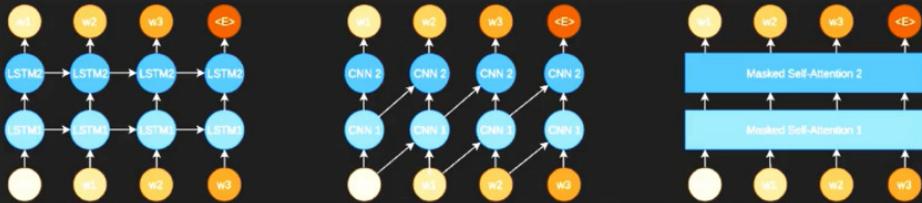
Модель

- Получение представлений слов, контекстуализация, предсказание следующего токена $P(t_1, t_2, \dots, t_l) = \prod_{i=1}^l P(t_i|t_{1:i-1})$
- Несколько слоёв RNN, CNN, Self-attention (Transformer decoder)
- Проблема OOV-слов
 - Единица анализа - один символ, а не токен (character-level LM)



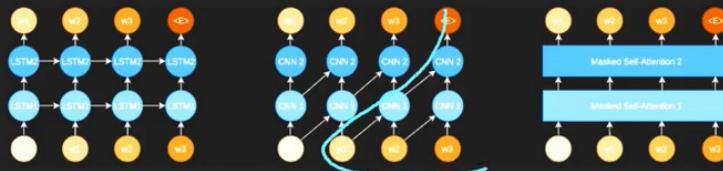
Второй вариант — работать на уровне слов, но векторы для неизвестных слов собирать из векторов N-грамм символов, входящих в эти слова — примерно, как это делается в [FastText](#). Но чаще всего сейчас используется промежуточный вариант, а именно — кодирование текста, при котором часто встречающиеся под-последовательности символов заменяются на новые символы, которые ранее не использовались. По сути, к текстам применяется некоторый

алгоритм сжатия без потерь. Один из самых популярных алгоритмов — [byte pair encoding](#). Он заключается в том что самая часто встречающаяся биграмма заменяется на новый символ, который раньше не использовался. Затем находится новая биграмма и всё повторяется. И таких замен делается достаточно много. Это позволяет сильно сократить размер словаря и, при этом, не сильно увеличить длину входных цепочек токенов, в отличие от символьных языковых моделей. Нейросетевые языковые модели также позволяют получать сжатое представление элементов текста, но теперь уже с учётом контекста, и эти представления можно использовать для решения прикладных задач. Оказывается, что первые слои выделяют признаки, соответствующие морфологической информации, частям речи, и так далее. Средние слои отвечают за согласование слов — то есть синтаксис, а признаки, полученные с последних слоёв, более полезны для задач [семантического анализа](#). То есть всё, практически, как в работе с изображениями — на каждом новом уровне мы абстрагируемся всё больше и больше, выделяем всё более крупные паттерны. Такие модели учитывают сложный контекст, но только с одной стороны от предсказываемого слова. За счёт специального кодирования входных последовательностей (byte pair encoding, например) проблема неизвестных слов практически не стоит. Но обучать такие модели очень дорого. Большие и мощные варианты требуют нескольких дней обучения на десятках видеокарт. Если модель основана на [рекуррентках](#), то сходимость может быть не очень быстрой из-за сложностей с [затуханием градиента](#). Ну, и контекст всё-таки учитывается только с одной стороны, и длина этого контекста ограничена. Особенно, если в основе архитектуры лежат свёрточные нейросети.



Модель

- Получение представлений слов, контекстуализация, предсказание следующего токена $P(t_1, t_2, \dots, t_l) = \prod_{i=1}^l P(t_i | t_{1:i-1})$
- Несколько слоёв RNN, CNN, Self-attention (Transformer decoder)
- Проблема OOV-слов
 - Единица анализа - один символ, а не токен (character-level LM)
 - FastText-эмбеддинги
 - Byte pair encoding (BPE): "aabc aabaa" → "Xbc XbX" → "Yc YX"



Модель

- Получение представлений слов, контекстуализация, предсказание следующего токена $P(t_1, t_2, \dots, t_l) = \prod_{i=1}^l P(t_i | t_{1:i-1})$
- Несколько слоёв RNN, CNN, Self-attention (Transformer decoder)
- Проблема OOV-слов
 - Единица анализа - один символ, а не токен (character-level LM)
 - FastText-эмбеддинги
 - Byte pair encoding (BPE): "aabc aabaa" → "Xbc XbX" → "Yc YX"



Преимущества

- Сжатое представление, обобщение
 - Морфология → поверхностный синтаксис → семантические роли, смыслы
- Учет сложного контекста (слева)
- Работа с OOV-словами (BPE)

Проблемы

- Высокая вычислительная сложность (дорого)
- Сложность сходимости (RNN)
- Длина учитываемого контекста слева ограничена (RNN, CNN)

Практическое задание:

Допустим, предложения в некотором языке имеют среднюю длину в 5 слов. Средняя длина слова в символах - 5.

Предположим также, что для этого языка характерно наличие далёких связей между словами (кореферентность, непроективные синтаксические связи).

Тогда чтобы успешно предсказывать последнее слово в предложении, языковой модели, работающей на уровне целых слов (word-level), нужно в среднем помнить 5 предыдущих элементов, а модели на уровне символов (character-level) - 25.

Почему character-level моделям сложнее?

Ответ:

- Если строить авторегрессионную языковую модель с помощью рекуррентных нейросетей, то с ростом длины последовательности острее проявляется проблема затухания градиента и забывания со временем.
- Если строить авторегрессионную языковую модель с помощью свёрточных нейросетей, сложно учитывать широкий контекст, так как ширину рецептивного поля свёрточных нейросетей можно увеличивать только вместе с количеством параметров (добавляя слои).

Из комментариев:

Комментарий:

кажется стоит рассказать про отличие русского и английского. Для английского символные модели часто не имеют смысла, а для русского особенно при генерации - имеют

Ответ (Романа Суворова):

спасибо за комментарий! Действительно, посимвольные модели играют более значимую роль при анализе и генерации текстов на языках с интенсивным словоизменением (например, русский), в отличие от таких языков как английский. Про особенности языков был [урок во введении](#)

Вопрос:

А как же dilated convolution? Разве она не увеличивает размер рецептивного поля без увеличения количества параметров?

Ответ (Романа Суворова):

частично решает, но в случае с рекуррентками мы всегда можем подать текст ещё длиннее и надеяться, что информация дойдёт, а в случае со свёртками (даже dilated) мы точно знаем максимальное расстояние между словами, сверх которого зависимости не будут учитываться. Если же мы хотим увеличить поле ещё больше, нам нужно добавить ещё слоёв (увеличить число параметров) и переучить сеть. Для каких-то задач этого достаточно, но современные языковые модели получают на вход достаточно длинные последовательности (от 512 до нескольких тысяч токенов). Например, чтобы объять зависимости длины 3000, нужно сделать 10 dilated слоёв с ядром 3 - такие штуки могут достаточно сложно учиться.

Практическое задание:

Byte Pair Encoding - простой алгоритм кодирования, который позволяет сжать длину последовательностей за счёт расширения словаря (то есть можно выбирать разные соотношения длины последовательностей и размера словаря).

Этот алгоритм работает в два этапа:

1. обучение - построение правил замены символов
2. кодирование - применение построенных правил замены, чтобы сократить длину текста

Разберём упрощённый алгоритм обучения на примере токенизации "косил косой осой".

1. Найти самую частотную биграмму - по два употребления имеют "ко", "ос", "со" и "ой" - выбираем "ко".
2. Назначить выбранной биграмме номер. Мы вместо номеров будем использовать заглавные буквы. Назначили: "ко" -> А.
3. Обновить последовательность с учётом созданного правила подстановки "ко" -> А: "Асила
Асой осой".
4. Повторяем шаги 1-3 для биграммы "со": подстановка "со" -> Б, последовательность "Асила
АБ осой".
5. Повторяем шаги 1-3 для биграммы "Бой": подстановка "Бой" -> В, последовательность "Асила
АВ осой".
6. Останавливаемся, когда сделаем заданное количество замен (то есть когда достигнем максимально допустимого размера словаря) или когда заменами не сможем дальше сжать тексты обучающей выборки.

В результате для этой последовательности мы получили словарь размера 10 (7 исходных символов и 3 подстановки) и уменьшили её длину с 16 до 10.

Когда мы кодируем текст, мы повторяем процедуру, жадно (то есть без возвратов и перебора) выбирая самые частотные биграммы и применяя подстановки, сохранённые на этапе обучения. Может так получиться, что две биграммы имеют одинаковую частоту - тогда применяйте правила подстановки в том порядке, в котором они добавлялись в словарь при обучении.

Примените полученные правила подстановки для кодирования текста "косой кокос".

Ответ:

АВ ААс

Вам не кажется, что это немного странно — предсказывать слово только по тем словам, что слева от него? В некоторых языках — например, в русском, порядок слов свободный, и контекст слева может вообще никак не объяснять появление очередного слова. Это ограничивает модель — то есть, её ёмкости вроде бы хватает для решения задачи, но входные данные не дают решать её лучше — у них просто нет достаточной информации. Самое очевидное решение — это учить две [авторегрессионные модели](#) — одна видит только то, что слева, а другая видит только то, что справа. При этом, признаки от обеих моделей можно использовать для лучшего решения прикладных задач. Такой подход называется "дву направленные языковые модели" и лежит в основе модели [ELMo](#), которая была предложена в 2017-2018 годах.^[1] Однако можно пойти ещё дальше и учитывать не только

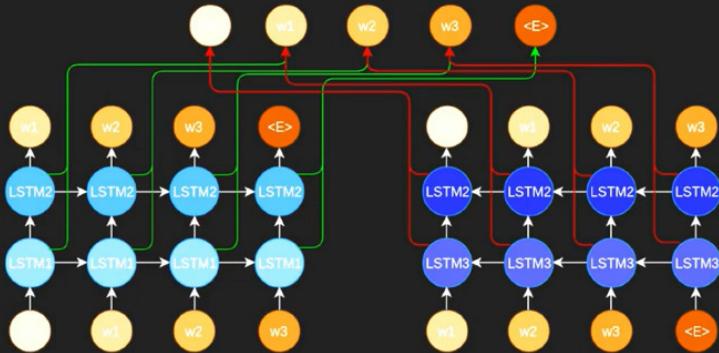
контекст строго слева или строго справа, а вообще весь доступный текст, кроме некоторых слов. При обучении модели [BERT](#), берётся фрагмент текста, в нём выбирается сколько-то слов, и они заменяются на фиктивный токен. Модель должна предсказать эти слова, используя все остальные входные данные. Так как модель основана на [трансформере](#),^[2] у неё нет особых проблем с моделированием далёких зависимостей — это позволяет моделировать не только одно предложение, а сразу большой непрерывный фрагмент текста. Тут учитывается вообще весь текст. Данная модель позволяет достичь лучших на настоящее время результатов на ряде задач, при этом, как оказалось, больше не обязательно строить какие-то специальные архитектуры для каждой задачи — можно, не меняя общей архитектуры BERT, несколько обучить его на маленьком датасете, и задача уже будет решаться отлично. Однако, и у этого монстра есть пара слабых мест, а именно — пропущенные слова считаются условно-независимыми при наблюдении остальных слов. Это означает, что их можно предсказывать независимо друг от друга — это не вполне корректно, так как от выбора одного пропущенного слова может зависеть выбор второго. А ещё, из-за того, что процесс обучения BERT опирается на внесение шума во входные данные, а именно — замене слов на токен "mask", который в реальных текстах никогда не встречается — фактическое распределение входных данных не вполне соответствует настоящему. Это может препятствовать переносу на реальные данные. Ну, как бы то ни было, эта модель работает отлично.

[1] Peters M. E. et al. Deep contextualized word representations //arXiv preprint arXiv:1802.05365. – 2018. (<https://arxiv.org/abs/1802.05365>)

[2] скоро мы очень подробно [поговорим о трансформерах](#)

Дополнительные комментарии к видео (от Романа Суворова):

- Модели Transformer, о которых идёт речь начиная с 1:02, будут разбираться [чуть позже](#).

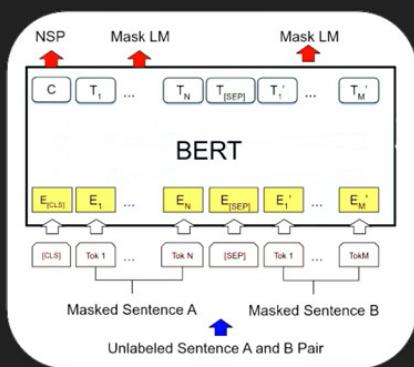


Модель

- Обучаются две языковые модели

$$P_{left}(t_i|t_{1:i-1}), \quad P_{right}(t_i|t_{i+1:l})$$

- Скрытые состояния со всех шагов и со всех слоёв используются как признаки для решения прикладной задачи
- biLM, ELMo¹



Модель

- Основа - Transformer (12 или 24 слоя)
- Входные слова случайно → [MASK]
- Модель учится предсказывать пропущенные слова

$$P(t_1^{mask}, t_2^{mask}, \dots, t_m^{mask} | P(t_1^{inp}, t_2^{inp}, \dots, t_n^{inp}))$$

- Предсказывается ещё и следующее предложение
- Учитывается контекст справа
- SOTA, гибкость
 - Пропущенные слова условно независимы

$$P(t_1^{mask}, \dots, t_m^{mask} | t_{1:n}^{inp}) = \prod_{i=1}^m P(t_i^{mask} | t_{1:n}^{inp})$$

- Распределение входных данных не соответствует реальным данным



Совсем недавно была предложена модель [XLNet](#)^[1] показывающая ещё лучшее качество, чем [BERT](#). В основе этой модели лежит идея об объединении авторегрессионных моделей и моделей, основанных на восстановлении зашумлённых данных — таких как BERT. Вспомним, что у нас есть правила цепочки и мы можем раскручивать совместное распределение по-разному. Например, первый вариант соответствует цепочке С-В-А, а второй вариант

соответствует цепочке А-С-В. Традиционные [авторегрессионные модели](#) раскручивают его всегда в одном порядке, который соответствует последовательному предсказанию очередного слова на основе слов, которые стояли в тексте слева: то есть слева-направо. BERT использует все входные данные — и слева, и справа. Но в итоге, как оказывается, работает не совсем с теми текстами, которые встречаются в жизни. Авторы XLNet предлагают использовать обычную авторегрессионную модель — такую, как OpenAI трансформер или [ELMo](#), но теперь предсказывать одно слово за другим, но для каждого следующего обучающего примера использовать новый порядок факторизации. Эта картинка показывает, какие токены должны использоваться для предсказания токена с порядковым номером "3" при различных порядках факторизации. В первом случае токен "3" в цепочке идёт последним, то есть мы считаем, что он зависит от всех остальных слов плюс какое-то скрытое состояние. А во втором случае токен номер "3" стоит на втором месте, то есть перед ним идёт только 4-й токен, и поэтому мы не используем остальные токены для предсказания этого 3-го токена. XLNet основан на [трансформере](#). Это позволяет очень удобно всё сделать за счёт передачи масок в механизм внимания. Это, на сегодняшний день, самая мощная языковая модель.

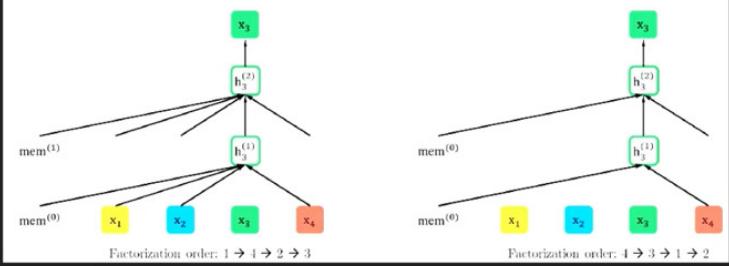
[1] Yang Z. et al. Xlnet: Generalized autoregressive pretraining for language understanding //Advances in neural information processing systems. – 2019. – С. 5754-5764.
(<https://arxiv.org/abs/1906.08237>)

Дополнительные комментарии к видео (от Романа Суворова):

- XLNet был самой мощной языковой моделью на момент записи видео. (доп. инфа от меня, на момент февраль 2022 года, XLNet уже на 99 месте)
- Прогресс можно отслеживать на страницах [GLUE Benchmark](#) и [SuperGLUE Benchmark](#)

Примечание от студента:

А тем временем OpenAI [всё-таки открыли свою GPT-2](#)



Модель

- Правило цепочки $P(a, b, c) = P(a|b, c)P(b|c)P(c) = P(b|a, c)P(c|a)P(a)$
- AP-модели (ELMo, OpenAI GPT) - слева направо $P(t_1, \dots, t_l) = \prod_{i=1}^l P(t_i|t_{1:i-1})$
- BERT - двунаправленный контекст, но предположение условной независимости $P(t_1^{mask}, \dots, t_m^{mask}|t_{1:n}^{inp}) = \prod_{i=1}^m P(t_i^{mask}|t_{1:n}^{inp})$
- XLNet - двунаправленный контекст, но порядок факторизации меняется $P(t_1, \dots, t_l) = \prod_{i=1}^l P(t_{order(i)}|t_{order(1:i-1)}), \quad order \in AllPermutations(1, \dots, l)$
- Transformer
- Порядок управляет масками в self-attention
- SOTA



Из практического задания:

Допустим, мы выбрали порядок факторизации $3 \rightarrow 1 \rightarrow 2$ для предложения "мама мыла раму". Это означает, что будут формироваться следующие обучающие примеры (в формате "Вход -> Эталонный выход", в скобках после слова стоит его исходная позиция в тексте):

- <start> -> раму(3)
- <start> раму(3) -> мама(1)
- <start> раму(3) мама(1) -> мыла(2)

Так как порядок слов в исходном предложении очень важен для понимания его смысла, необходимо добавлять к эмбеддингу слов эмбеддинг его исходной позиции в тексте. Это называется позиционным кодированием (подробнее - [в лекции](#) и [семинаре](#) про трансформер).

Итак, языковые модели — это одна из самых горячих тем исследований в области обработки текстов на [естественном языке](#). Давайте выделим некоторые архитектурные решения, к которым пришли исследователи в последнее время. Во-первых — модели, использующие сжатие текста и словаря через "[byte pair encoding](#)" и, при этом, использующие честный софтмакс (а не его аппроксимацию) показывают лучшее качество по сравнению с моделями, которые опираются на целые словари, но упрощаются софтмакс (например, [negative sampling](#)). Модели честные показывают качество лучше. Во-вторых — трансформер, кажется, занял доминирующее положение^[1], так как он легче обучается и способен моделировать далёкие зависимости. А ещё, в механизме внимания можно подавать маски и, тем самым, управлять зависимостями между словами. И это делает трансформер очень гибким. С помощью него можно сделать некоторые вещи, которые вообще никак не сделать с помощью рекурренток.

И, как одно из следствий использования [трансформера](#) (а именно, его способность учитывать реально далёкие зависимости), лучше обучать модель на длинных входных последовательностях, то есть на нескольких подряд идущих предложениях, а не на отдельных предложениях. Это вносит в модель информацию о связи предложений, о дискурсе, а не только о структуре отдельных предложений. Мы поговорили про задачу [моделирования языка](#), которая на практике сводится к задаче предсказания слов, то есть к большой задаче классификации. Начали с классики — с N-граммной модели. Также мы рассмотрели несколько основных тем, которым были посвящены работы в области моделирования языка. Первая такая тема — сжатие и обобщение представлений, то есть — то, чего в N-граммной модели вообще не было. Вторая такая проблема — необходимость работать с большими словарями, её решали через упрощение [софтмакса](#) или через сжимающее кодирование текста. Третья проблема — необходимость обрабатывать слова, которых не было в обучающей выборке. Её решают или через посимвольные модели, или через [эмбеддинги](#) N-грамм, как это делает [FastText](#), например, или через "byte pair encoding". Четвёртая проблема — это учёт максимального контекста. Самый популярный способ построения языковых моделей — [авторегрессия](#), когда предсказание слов выполняется последовательно, одно слово за шаг. Чтобы модель научилась учитывать контекст не только слева, но и справа, применялись двунаправленные языковые модели (например, [ELMo](#)), модели с перестановками ([XLNet](#)) или с зашумлением входных данных ([BERT](#)). В этой области были достигнуты впечатляющие результаты. Модели, обученные без учителя на больших [корпусах](#) текстов неплохо решают даже те задачи, для которых их специально не учили. Однако у себя дома, пока что, вряд ли получится обучить BERT или XLNet — для этого требуется несколько дней непрерывной работы нескольких десятков видеокарт. Но хорошая новость заключается в том, что авторы многих моделей выложили веса и исходные коды в открытый доступ, и вы можете их скачать, включить свою модель, и почти бесплатно улучшить качество решения ваших прикладных задач. Это уже относится к теме переноса навыков ([transfer learning](#)) и этому будет посвящена отдельная лекция.

[1] скоро мы очень подробно [поговорим о трансформерах](#)

- Использовать BPE и "честный" softmax, а не целые слова и аппроксимацию softmax
- Использовать Transformer и self-attention, а не RNN/CNN
- Управлять зависимостями между токенами с помощью масок в Transformer
- Работать с несколькими подряд идущими предложениями



- Постановка задачи моделирования языка
- Базовый подход - N-граммная модель
- Обобщение, получение сжатого представления
- Работа с большими словарями - neg. sampling, иерархический softmax, BPE
- Работа с неизвестными словами (OOV) - FastText, character CNN, BPE
- Учет сложного контекста
- Авторегрессионные модели - RNN, CNN, OpenAI GPT (Transformer), XLNet
- Модели с двусторонним контекстом - ELMo, BERT, XLNet
- Обучение с нуля большой языковой модели требует очень много ресурсов
- Веса для многих моделей выложены в открытый доступ!



4.3 Семинар: генерация имён и лозунгов с помощью RNN

Для данного семинара Вам потребуется ноутбук [task4_RNN_name_generator.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):

Examples

Recent

Google Drive

Enter a GitHub URL or search by organization or user

<https://github.com/Samsung-IT-Academy/stepik-dl-nlp>

Repository:

[Samsung-IT-Academy/stepik-dl-nlp](#)

Branch:

[master](#)

Path



task1_20newsgroups.ipynb

2) Запустите ноутбук.

3) Не забудьте выполнить команду `git clone` из первой (закомментированной) ячейки, чтобы выкачать на colab библиотеку `dlnlputils`

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! В сегодняшнем семинаре мы рассмотрим задачу генерации текста с помощью [рекуррентных нейронных](#) сетей. Наверняка, многие из вас сталкивались со следующей проблемой — вы пишете код и не можете придумать хорошее название для новой переменной. Или другая проблема — вы регистрируете личный кабинет в каком-либо сервисе и хотите придумать короткий и красивый, запоминающийся логин... С решением обеих этих проблем нам может помочь подход, разобранный в лекции, а именно рекуррентные

нейронные сети. Итак, в сегодняшнем семинаре мы разберём проблему генерации новых имён с помощью рекуррентных нейронных сетей. А именно: сегодня мы напишем рекуррентную нейронную сеть с нуля, мы не будем использовать какие-либо готовые решения из pytorch, tensorflow или любых других библиотек, а будем писать рекуррентную нейронную сеть руками. А затем мы научим нашу нейронную сеть генерировать новые имена, обучив её на небольшой коллекции. Для начала, давайте посмотрим на данные, которые у нас есть. Наш датасет состоит из 9000 имён, которые написаны латиницей. Они лежат в файле под названием `russian_names.txt`, давайте посмотрим на наш файл. Как вы можете видеть, в этом это датасете содержатся не только имена, но и фамилии, и при этом — это русские имена. Все имена отсортированы в алфавитном порядке. Вы можете заметить небольшую особенность — каждое имя в нашем датасете будет начинаться с пробела, и вот в этой строке кода мы добавляем пробел перед каждым именем — зачем это делается? Кажется, это довольно странное решение.

Генерация текста с помощью RNN

```
In [ ]: import os  
  
import numpy as np  
import matplotlib.pyplot as plt  
%matplotlib inline  
  
import warnings  
warnings.simplefilter('ignore')
```

Данные

Датасет содержит ~9k имен, все написаны латиницей.

```
In [ ]: with open("russian_names.txt") as input_file:  
    names = input_file.read()[:-1].split('\n')  
    names = [' ' + line for line in names]
```

```
In [ ]: names[:5]
```

Посмотрим на распределение длин имен:

```
To [1]: plt.title('Name length distribution')
```



```
In [2]: import os  
  
import numpy as np  
import matplotlib.pyplot as plt  
%matplotlib inline  
  
import warnings  
warnings.simplefilter('ignore')
```

Данные

Датасет содержит ~9k имен, все написаны латиницей.

```
In [3]: with open("russian_names.txt") as input_file:  
    names = input_file.read()[:-1].split('\n')  
    names = [' ' + line for line in names]
```

```
In [4]: names[:5]
```

```
Out[4]: ['Ababko', 'Abaev', 'Abagyan', 'Abaidulin', 'Abaidullin']
```

Посмотрим на распределение длин имен:

```
In [ ]: plt.title('Name length distribution')  
plt.hist(list(map(len, names)), bins=25);
```

Препроцессинг



После того, как мы обучим нашу нейронную сеть, мы сможем генерировать имена, которые соответствуют некоторым условиям — например, имена, которые начинаются на букву "а" или на буквы "abc", или какие-либо другие условия. Если же мы захотим генерировать любые имена, начинающиеся с любой буквы, мы просто передадим нашей функции пробел в качестве

первого символа. Таким образом, сможем сгенерировать имена, начинающиеся на любую букву. Отлично! С этой небольшой хитростью в коде разобрались. Давайте теперь посмотрим на распределение длин имён в нашем датасете. Как вы видите, распределение длин имён описывается нормальным распределением и, в целом, большинство имён имеет длину от 6 до, скажем, 12 символов, что довольно логично и соответствует действительности. При этом, есть имена, которые содержат всего три буквы или содержат аж 18 букв. Отлично, теперь нам нужно немного предобработать данные перед тем, как подавать их в нашу нейронную сеть. Нам нужно составить список уникальных символов, которые встречаются в нашем датасете. Это мы можем сделать с помощью такой строки кода. И посмотрим, сколько уникальных символов есть в нашем датасете: их всего 52. Теперь нужно создать словарь соответствия буквы некоторому численному ID. Мы не можем подавать в нейросеть буквенный input, нам нужно преобразовать его в некоторое численное представление, поэтому создадим словарь, который будет мапить (map) букву в её численный ID. Отлично! С помощью этой строки кода создаём словарь, который мы назовём "token_to_id". Отлично! И теперь мы хотим преобразовать наши входные данные, а именно — наши 9 с небольшим хвостиком тысяч имён в некоторое численное представление, то есть вместо имени мы хотим получить численный вектор. Сделать это мы можем с помощью функций "to_matrix", которая будет преобразовывать наше имя из буквенного, человеко-читаемого формата в формат "вектор с числами". Давайте посмотрим на такую матрицу, как она будет выглядеть. Берём 5 имён и смотрим на то, как будет выглядеть матрица для этих 5 имён. Как вы можете видеть, все векторы — одинаковой длины, и при этом все векторы начинаются с одного и того же числа: 29. Это вполне логично, поскольку в начале каждого имени мы ставили пробел, "29" — это ID пробела. Отлично! При этом, как вы можете заметить, в конце некоторых строк также присутствуют символы "29", один или какое-то большее количество раз. Почему так происходит? Так происходит потому, что наша функция "to matrix" умеет дополнять слова, которые оказались короче самого длинного слова в коллекции — пробелами в конце, для того, чтобы все слова были одинаковой длины и векторы в нашей матрице имели одинаковую длину. Поэтому в конце некоторых слов вы можете увидеть числа "29", которые соответствуют дополнительным пробелам в конце слов, которые оказались короче самого длинного слова. На этом заканчивается часть про препроцессинг наших данных, она очень простая. Теперь давайте перейдём к более сложным и более интересным вещам, а именно — написанию рекуррентной нейронной сети своими руками.

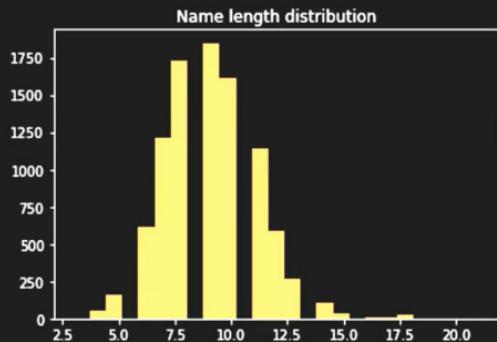
```
In [3]: with open("russian_names.txt") as input_file:  
    names = input_file.read()[:-1].split('\n')  
    names = [' ' + line for line in names]
```

```
In [4]: names[:5]
```

```
Out[4]: ['Ababko', 'Abaev', 'Abagyan', 'Abaidulin', 'Abaidullin']
```

Посмотрим на распределение длин имен:

```
In [5]: plt.title('Name length distribution')  
plt.hist(list(map(len, names)), bins=25);
```



Препроцессинг

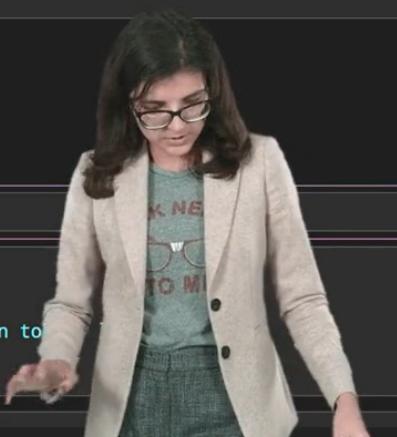
```
In [6]: #all unique characters go here  
tokens = list(set(''.join(names)))  
  
num_tokens = len(tokens)  
print ('num_tokens = ', num_tokens)  
  
num_tokens = 52
```

Символы -> id

Создадим словарь < символ > -> < id >

```
In [ ]: token_to_id = {token: idx for idx, token in enumerate(tokens)}
```

```
In [ ]: assert len(tokens) == len(token_to_id), "dictionaries must have same size"  
for i in range(num_tokens):  
    assert token_to_id[tokens[i]] == i, "token identifier must be it's position in tokens"  
print("Seems alright!")
```



```
In [9]: def to_matrix(data, token_dict, max_len=None, dtype='int32', batch_first = True):
    """Casts a list of names into rnn-digestable matrix"""

    max_len = max_len or max(map(len, data))
    data_ix = np.zeros([len(data), max_len], dtype) + token_dict[' ']

    for i in range(len(data)):
        line_ix = [token_dict[c] for c in data[i]]
        data_ix[i, :len(line_ix)] = line_ix

    if not batch_first: # convert [batch, time] into [time, batch]
        data_ix = np.transpose(data_ix)

    return data_ix
```

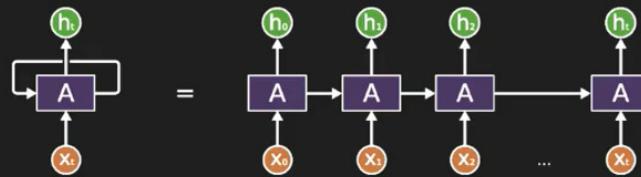
```
In [10]: #Example: cast 5 names to matrices, pad with zeros
print('\n'.join(names[:2000]))
print(to_matrix(names[:2000], token_to_id))
```

```
Ababko
Chihachev
Isaikov
Nakhamkin
Ustenko
[[29 17 22 46 22 34 38 29 29 29]
 [29 37 24 47 24 46 50 24 14 35]
 [29 49 31 46 47 34 38 35 29 29]
 [29 41 46 34 24 46 8 34 47 23]
 [29 3 31 44 14 23 34 38 29 29]]
```



Здесь на экране вы можете увидеть изображение из лекции, а именно — основной принцип работы и рекуррентной нейронной сети: небольшую схему, которая поможет нам писать код дальше. Будем писать класс, который мы назовём "CharRnnCell". Что у нас есть в этом классе? У нас есть функции "forward" и есть функция "initial state". "initial state" просто заполняет векторы нулями, выход функции "initial state" соответствует переменной "h0", или нулевому скрытому состоянию нашей нейронной сети. Рассмотрим более подробно, что происходит внутри функции "forward". Для начала мы преобразовываем наши входные векторы в [эмбеддинги](#) с помощью слоя, который мы берём из библиотеки pytorch, а именно "nn embedding". Дальше мы конкатенируем текущий входной вектор из переменной "x embedding" и скрытое состояние из предыдущего шага, и дальше, с помощью "[rnn update](#)" (именно так мы назвали линейный слой функции "init"), мы делаем следующий шаг — предсказываем следующее скрытое состояние. Дальше в ход идёт нелинейность — мы применяем гиперболический тангенс к выходу этого слоя. В итоге, на выход мы передаём следующее скрытое состояние и вероятности для следующего символа. А именно, мы получим 52 вероятности по каждому символу из нашего словаря.

Рекуррентные нейронные сети



```
In [ ]: import torch, torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
```

```
In [ ]: class CharRNNCell(nn.Module):
    """
    Implement the scheme above as torch module
    """
    def __init__(self, num_tokens=len(tokens), embedding_size=16, rnn_num_units=64):
        super(self.__class__, self).__init__()
        self.num_units = rnn_num_units

        self.embedding = nn.Embedding(num_tokens, embedding_size)
        self.rnn_update = nn.Linear(embedding_size + rnn_num_units, rnn_num_units)
        self.rnn_to_logits = nn.Linear(rnn_num_units, num_tokens)

    def forward(self, x, h_prev):
        """
        This method computes h_next(x, h_prev) and log P(x_next | h_next)
        We'll call it repeatedly to produce the whole sequence.
        """

```



```
self.embedding = nn.Embedding(num_tokens, embedding_size)
self.rnn_update = nn.Linear(embedding_size + rnn_num_units, rnn_num_units)
self.rnn_to_logits = nn.Linear(rnn_num_units, num_tokens)

def forward(self, x, h_prev):
    """
    This method computes h_next(x, h_prev) and log P(x_next | h_next)
    We'll call it repeatedly to produce the whole sequence.

    :param x: batch of character ids, variable containing vector of int64
    :param h_prev: previous rnn hidden states, variable containing matrix [batch, rnn_num_units] of float32
    """
    # get vector embedding of x
    x_emb = self.embedding(x)

    # compute next hidden state using self.rnn_update
    x_and_h = torch.cat([x_emb, h_prev], dim=1)
    h_next = self.rnn_update(x_and_h)

    h_next = F.tanh(h_next)

    #compute logits for next character probs
    logits = self.rnn_to_logits(h_next)

    return h_next, F.log_softmax(logits, -1)

def initial_state(self, batch_size):
    """
    return rnn state before it processes first input (aka h0)
    """
    return Variable(torch.zeros(batch_size, self.num_units))
```



Тренировка сети, RNN loop

Теперь мы можем переходить в тренировке нашей нейронной сети. После того, как мы написали один шаг [RNN](#), мы можем вызвать его в цикле и предсказывать символы на каждом шаге нашего цикла. Теперь мы можем начинать тренировку нашей сети. Минимизируя [кросс-энтропию](#), либо максимизируя логарифм правдоподобия нашей модели (что — то же самое) — обучать нашу сеть. Берём матрицу ID токенов, сдвинутую на "i" символов влево, так, чтобы

именно "i"-ый символ был следующим символом для предсказания на "i"-ом шаге. Такая матрица хранится в переменной "batch index" (а именно, вот здесь) и, дальше, мы можем переходить к обучению сети. Мы делаем "backward pass" — именно здесь мы вычисляем градиенты нашей лосс-функции по параметрам, делаем шаг с помощью "opt.step" и не забываем сделать "zero_grad", если мы этого не сделаем то градиенты из предыдущих шагов будут накапливаться, аккумулироваться, и это приведёт к неправильному обучению сети. Кроме того, будем визуализировать процесс обучения нашей нейронной сети с помощью библиотеки matplotlib, без каких-либо дополнительных инструментов. Запускаем обучение!

```
        return h_next, F.log_softmax(logits, -1)

    def initial_state(self, batch_size):
        """ return rnn state before it processes first input (aka h0) """
        return Variable(torch.zeros(batch_size, self.num_units))
```

Тренировка сети, RNN loop

```
In [ ]: def rnn_loop(rnn, batch_index):
    """
    Computes log P(next_character) for all time-steps in names_ix
    :param names_ix: an int32 matrix of shape [batch, time], output of to_matrix(names)
    """
    batch_size, max_length = batch_index.size()
    hid_state = rnn.initial_state(batch_size)
    logprobs = []

    for x_t in batch_index.transpose(0,1):
        hid_state, logp_next = rnn(x_t, hid_state)
        logprobs.append(logp_next)

    return torch.stack(logprobs, dim=1)
```

Тренировка сети

```
In [ ]: from IPython.display import clear_output
from random import sample
```



Тренировка сети

```
In [14]: from IPython.display import clear_output
from random import sample

char_rnn = CharRNNCell()
opt = torch.optim.Adam(char_rnn.parameters())
history = []

In [15]: MAX_LENGTH = max(map(len, names))

for i in range(1000):
    batch_ix = to_matrix(sample(names, 32), token_to_id, max_len=MAX_LENGTH)
    batch_ix = torch.tensor(batch_ix, dtype=torch.int64)

    logp_seq = rnn_loop(char_rnn, batch_ix)

    # compute loss
    predictions_logp = logp_seq[:, :-1]
    actual_next_tokens = batch_ix[:, 1:]

    loss = -torch.mean(torch.gather(predictions_logp, dim=2, index=actual_next_tokens[:, :, None]))

    # train with backprop
    loss.backward()
    opt.step()
    opt.zero_grad()

    # visualizing training process
    history.append(loss.data.numpy())
```



```

history = []

In [15]: MAX_LENGTH = max(map(len, names))

for i in range(1000):
    batch_ix = to_matrix(sample(names, 32), token_to_id, max_len=MAX_LENGTH)
    batch_ix = torch.tensor(batch_ix, dtype=torch.int64)

    logp_seq = rnn_loop(char_rnn, batch_ix)

    # compute loss
    predictions_logp = logp_seq[:, :-1]
    actual_next_tokens = batch_ix[:, 1:]

    loss = -torch.mean(torch.gather(predictions_logp, dim=2, index=actual_next_tokens[:, :, None]))

    # train with backprop
    loss.backward()
    opt.step()
    opt.zero_grad()

    # visualizing training process
    history.append(loss.data.numpy())
    if (i + 1) % 100 == 0:
        clear_output(True)
        plt.plot(history, label='loss')
        plt.legend()
        plt.show()

assert np.mean(history[:10]) > np.mean(history[-10:]), "RNN didn't converge."

```



Из комментариев:

Комментарий от студента:

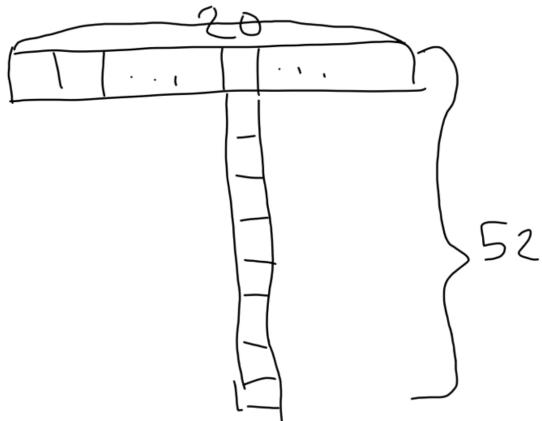
Очень понятное объяснение того, как работает функция `torch.gather`:

<https://stackoverflow.com/questions/50999977/what-does-the-gather-function-do-in-pytorch-in-layman-terms>

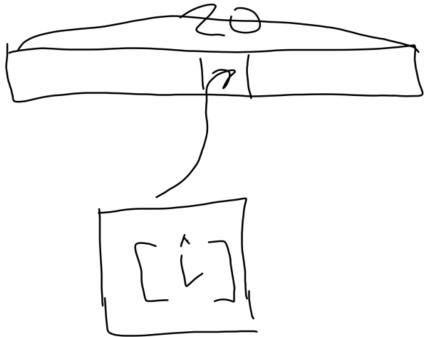
(по документации тяжело понять что происходит, а там с картинками и объяснениями)

Пояснение того, как `torch.gather` работает именно в нашем случае:

Возьмем любое слово из нашего батча, например 4ое. Под 4ым индексом в тензоре `predictions_logp` лежит матрица размера (макс длина слова - 1) x количество букв (20 x 52), в каждой ячейке которой лежит логарифм вероятности данной букве (по номеру строки) стать следующей в 4ом слове из батча:



Это наш `input`, из которого вероятность будет выбрана по индексу = `actual_next_tokens[:, :, None]`. В этом тензоре под номером 4 (4ое слово в батче) лежит:



Это тензор размерности $32 \times 20 \times 1$, и i это индекс следующей буквы в обучающем датасете. Вероятность этой буквы выбирается из тензора `predictions_logp` и берется с минусом для дальнейшего увеличения вероятности в процессе обучения.

Ответ (от студента):

Если подытожить, то в команде `torch.gather` первый аргумент - матрица предсказаний размерностью $[32, 20, 52]$, `index` - матрица меток размерностью $[32, 20, 1]$. И этой командой мы отбираем из всех предсказаний значения вероятностей именно для тех символов, что указаны в `actual_next_tokens`. На выходе имеем размерность $[32, 20, 1]$ и далее проводим суммирование по всем элементам для подсчета общей ошибки.

Вопрос:

А почему как `predictions_logp` мы берем все вероятности, кроме последнего символа?

Ответ (от Романа Суворова):

обратите внимание на следующую строчку: там мы берём `batch_ix` начиная со второго символа. Таким образом мы сдвигаем тензор меток на одну позицию влево. Это нужно, чтобы показать модели, что на i -й входной позиции надо предсказывать $i+1$ -й символ, а не i -й. Но так как мы откусываем один символ от `batch_ix`, то нам нужно откусить один символ и от `logp_seq` - они должны быть одной длины. К тому же вероятность первого символа мы здесь не моделируем. Более того, в [нашем датасете первый символ - это всегда пробел](#).

В режиме реального времени мы можем видеть, как наша сеть обучается. Я уже запускала обучение этой сети, поэтому она начала обучаться не с нулевой итерации, а с 1000-ой. 1000 итераций прошло буквально за несколько секунд, и теперь мы можем переходить к генерации имён с помощью обычной [RNN](#)-ки. Давайте попробуем сгенерировать несколько имён! Для этого нам, всего лишь, нужно несколько раз и сделать forward pass в цикле, предсказать несколько символов и сконкатенировать их в одно слово. Посмотрим на 10 имён, которые отсэмплирует наша нейронная сеть. Отлично! Мы получили какие-то фамилии, и они очень похожи на настоящие фамилии, например — "Чермеев", "Лобовский", "Баторов"... Готовы спорить, что люди с такими фамилиями действительно живут в России. Кроме того, мы можем осуществить генерацию имён, которые начинаются с определённого набора символов — например, с букв "АР". И мы получаем фамилии "Араскин", "Ардиский", "Артроев" и

некоторые другие. Что ещё мы можем менять при генерации фамилий? Например, мы можем менять такой параметр, как температура. Давайте попробуем поставить её равной "0.2". Получаем какой-то достаточно логичный вывод. Теперь поставим температуру равной "2" и мы видим что-то совсем нелогичное — например, пробелы внутри фамилий или какие-то странные заглавные буквы внутри фамилий. Если же мы поставим совсем маленькую температуру, например "0.01", мы будем получать одну и ту же фамилию каждый раз (практически одну и ту же). Почему так происходит? Если мы ставим маленькую температуру, сеть будет генерировать такие фамилии в которых она наиболее уверена. Оказалось, что наша сеть уверена, что фамилии "Ардиров", "Артамов" и... кажется, всё — это наиболее вероятные фамилии, которые начинаются на буквы "АР". Возможно, они встречались в нашем датасете или были какие-то фамилии, похожие на эти две фамилии. Если же мы ставим большую температуру, то наша сеть будет генерировать очень разнообразный выход, но при этом, зачастую, достаточно нелогичный, потому что она могла выучить некоторые странные особенности нашего датасета. Наша задача — выбрать такую температуру, которая будет обеспечивать достаточное разнообразие нашего датасета и, при этом, генерировать логичные фамилии и логичные имена.

```

actual_next_tokens = batch_ix[:, 1:]

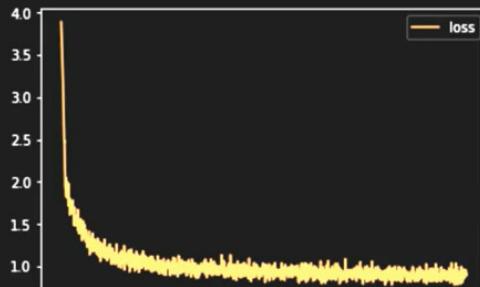
loss = -torch.mean(torch.gather(predictions_logp, dim=2, index=actual_next_tokens[:, :, None]))

# train with backprop
loss.backward()
opt.step()
opt.zero_grad()

# visualizing training process
history.append(loss.data.numpy())
if (i + 1) % 100 == 0:
    clear_output(True)
    plt.plot(history, label='loss')
    plt.legend()
    plt.show()

assert np.mean(history[:10]) > np.mean(history[-10:]), "RNN didn't converge."

```



Генерация имен

```
In [17]: def generate_sample(char_rnn, seed_phrase=' ', max_length=MAX_LENGTH, temperature=1.0):
    """
    The function generates text given a phrase of length at least SEQ_LENGTH.
    :param seed_phrase: prefix characters. The RNN is asked to continue the phrase
    :param max_length: maximum output length, including seed_phrase
    :param temperature: coefficient for sampling. higher temperature produces more chaotic outputs,
                        smaller temperature converges to the single most likely output
    ...
    x_sequence = [token_to_id[token] for token in seed_phrase]
    x_sequence = torch.tensor([x_sequence], dtype=torch.int64)
    hid_state = char_rnn.initial_state(batch_size=1)

    #feed the seed phrase, if any
    for i in range(len(seed_phrase) - 1):
        hid_state, _ = char_rnn(x_sequence[:, i], hid_state)

    #start generating
    for _ in range(max_length - len(seed_phrase)):
        hid_state, logp_next = char_rnn(x_sequence[:, -1], hid_state)
        p_next = F.softmax(logp_next / temperature, dim=-1).data.numpy()[0]

        # sample next token and push it back into x_sequence
        next_ix = np.random.choice(len(tokens), p=p_next)
        next_ix = torch.tensor([[next_ix]], dtype=torch.int64)
        x_sequence = torch.cat([x_sequence, next_ix], dim=1)

    return ''.join([tokens[ix] for ix in x_sequence.data.numpy()[0]])

```



```
for i in range(len(seed_phrase) - 1):
    hid_state, _ = char_rnn(x_sequence[:, i], hid_state)

#start generating
for _ in range(max_length - len(seed_phrase)):
    hid_state, logp_next = char_rnn(x_sequence[:, -1], hid_state)
    p_next = F.softmax(logp_next / temperature, dim=-1).data.numpy()[0]

    # sample next token and push it back into x_sequence
    next_ix = np.random.choice(len(tokens), p=p_next)
    next_ix = torch.tensor([[next_ix]], dtype=torch.int64)
    x_sequence = torch.cat([x_sequence, next_ix], dim=1)

return ''.join([tokens[i] for i in x_sequence.data.numpy()[0]])
```

In [18]: `for _ in range(10):
 print(generate_sample(char_rnn))`

Chermeev
Bakharushin
Pimo dzaev
Alabovsky
Duzttenh
Ust,ewnkov
Zhazhekovich
Battorov
Agoshin
Turmen

In []: `for _ in range(10):
 print(generate_sample(char_rnn, seed_phrase=' Ar'))`

In [19]: `for _ in range(10):
 print(generate_sample(char_rnn, seed_phrase=' Ar'))`

Arasgin
Artchenchaenkov
Artbavin
Armov
Aretshir
Ardirsky
Artroev
Arevmav
Armhenyaren
Arlikhal

Более простое решение

* nn.RNNCell(emb_size - rnn num units) - war RNN. Алгоритм: concat-linear-tanh

```
In [21]: for _ in range(10):
    print(generate_sample(char_rnn, seed_phrase=' Ar', temperature=2.0))
ArtsPajkowka
ArEchaikdey
Ariegumpug
Arismoaski fovTer Nr
Arcalorfely Fankrkhd
Arelogasmkd1Etmenkow
Arfso v
Aret hek s
Armiblehkok G
ArTrehyrste
```



Более простое решение

```
* np.RNNCell(emb_size rnn.num_units) - шир RNN Алгоритм: concat-linear-tanh
```

```
In [22]: for _ in range(10):
    print(generate_sample(char_rnn, seed_phrase=' Ar', temperature=0.01))
Ardurov
Ardurov
Artamov
Ardurov
Artamov
Ardurov
Ardurov
Ardurov
Artamov
Ardurov
```



Более простое решение

```
* np.RNNCell(emb_size rnn.num_units) - шир RNN Алгоритм: concat-linear-tanh
```

Из комментариев:

Вопрос:

Прошу прощения, прохожу курс по ночам, поэтому мог пропустить, но разве про температуру было в лекциях?

P.S. судя по следующему заданию, действительно не было

Ответ (Николая Копырина):

Насколько я знаю, температура – это такой общий полу-эмпирический момент в алгоритмах стохастической оптимизации (например, "симулированного отжига"). Этот параметр определяет вклад случайности в результат оптимизации. При высокой температуре скачки решения в случайном направлении более вероятны.

Ответ (студента):

параметр температуры в коде в такой строке фигурирует:

```
p_next = F.softmax(logp_next / temperature, dim=-1).data.numpy()[0]
```

где logp_next - предсказания сети для логарифма правдоподобия следующего символа по всему словарю символов.

Получается, чем ниже температура, тем больше будут отличаться вероятности одних символов от других, и наоборот. А символ выбирается далее в следующей строке:

```
next_ix = np.random.choice(len(tokens), p=p_next)
```

Из практического задания:

В семинаре при рассказе о генерации текста упоминается "температура". Что же имеется в виду?

Температура в softmax - параметр, который отвечает за "случайность" итогового распределения. Если устремить температуру к нулю, итоговое распределение вырождается в one-hot (элемент с максимальным значением выбирается с вероятностью 1.0). Такое поведение согласуется и с физическими явлениями. С возрастанием температуры энтропия (мера хаотичности) системы растет, а значит события становятся более случайными (т.е. распределение стремится к равномерному).

Отлично, мы закончили с генерацией фамилий с помощью нашего датасета. Теперь перейдём к более простому решению. Писать [RNN](#) своими руками — это отличное упражнение, чтобы лучше понять, как работает [рекуррентная нейронная](#) сеть, понять какие-то её особенности, возможно, повторить материал из лекции. Но, кажется, это не то что вы хотите делать каждый день, когда решаете какую-либо задачу, связанную с генерацией текстов, например, на работе или в каком-то своём проекте. В этом случае гораздо удобнее взять готовый модуль — например, из библиотеки pytorch. В pytorch уже реализована RNN — вы можете просто вызвать из неё "["nn.rnnCell"](#)" или "["nn.rnn"](#)", и использовать уже готовый модуль для того, чтобы обучить вашу нейронную сеть генерировать новые имена. Здесь будет гораздо меньше кода и, при этом, код будет работать чуть-чуть быстрее, чем на нашей самописной RNN-ке, но, при этом, результат будет примерно такой же. Давайте проверим! Возьмём pytorch "nn.rnn" и обучим нашу сеть. Здесь мы будем использовать не стохастический градиентный спуск, как это было при обучении в предыдущем блоке, а будем использовать в качестве оптимизатора [Adam](#). И точно так же запустим обучение нашей нейронной сети. Мы видим, как наша сеть обучается... обучается... и доходит до 1000 итераций. Отлично! С помощью библиотеки pytorch мы можем обучить то же самое, что вы только что написали своими руками, но значительно быстрее и с меньшим количеством кода.

Более простое решение

- nn.RNNCell(emb_size, rnn_num_units) - шаг RNN. Алгоритм: concat-linear-tanh
- nn.RNN(emb_size, rnn_num_units) - весь nn_loop.

Кроме того, в PyTorch есть nn.LSTMCell, nn.LSTM, nn.GRUCell, nn.GRU, etc. etc.

Перепишем наш пример с генерацией имен с помощью средств PyTorch.

```
In [ ]: class CharRNNLoop(nn.Module):
    def __init__(self, num_tokens=num_tokens, emb_size=16, rnn_num_units=64):
        super(self.__class__, self).__init__()
        self.emb = nn.Embedding(num_tokens, emb_size)
        self.rnn = nn.RNN(emb_size, rnn_num_units, batch_first=True)
        self.hid_to_logits = nn.Linear(rnn_num_units, num_tokens)

    def forward(self, x):
        assert isinstance(x, Variable) and isinstance(x.data, torch.LongTensor)
        h_seq, _ = self.rnn(self.emb(x))
        next_logits = self.hid_to_logits(h_seq)
        next_logp = F.log_softmax(next_logits, dim=-1)
        return next_logp

model = CharRNNLoop()
opt = torch.optim.Adam(model.parameters())
history = []
```



```
model = CharRNNLoop()
opt = torch.optim.Adam(model.parameters())
history = []

In [ ]: MAX_LENGTH = max(map(len, names))

for i in range(1000):
    batch_ix = to_matrix(sample(names, 32), token_to_id, max_len=MAX_LENGTH)
    batch_ix = torch.tensor(batch_ix, dtype=torch.int64)

    logp_seq = model(batch_ix)

    # compute loss
    predictions_logp = logp_seq[:, :-1]
    actual_next_tokens = batch_ix[:, 1:]

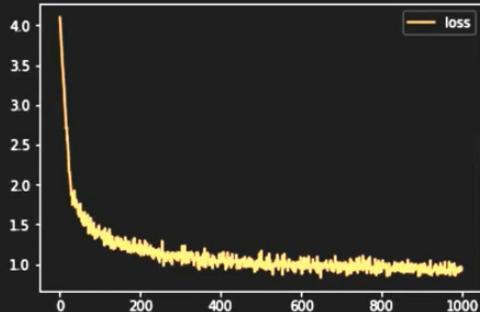
    loss = -torch.mean(torch.gather(predictions_logp, dim=2, index=actual_next_tokens[:, :, None]))

    # train with backprop
    loss.backward()
    opt.step()
    opt.zero_grad()

    history.append(loss.data.numpy())
    if (i + 1) % 100 == 0:
        clear_output(True)
        plt.plot(history, label='loss')
        plt.legend()
        plt.show()

assert np.mean(history[:25]) > np.mean(history[-25:]), "RNN didn't converge."
```





Домашнее задание: мотивационные лозунги

Из комментариев:

Вопрос:

А зачем здесь упоминается `nn.RNNCell`? Вроде, в коде семинара его нет.

И скобка пропущена после аргументов `nn.RNN`.

Ответ (Романа Суворова):

RNNCell можно использовать тогда, когда Вы сами хотите реализовать цикл. Это полезно, например, когда ваши данные - не последовательность, а граф или дерево (как в tree recursive networks)

Мы рассмотрели задачу генерацию текстов, а именно — генерации новых имен с помощью [рекуррентных нейронных](#) сетей в этом семинаре. Что же ещё можно делать рекуррентными нейронными сетями? Если говорить про генерацию текстов — можно генерировать огромное количество другой информации, например — вы можете генерировать повести, романы, стихи, и будет получаться даже что-то более-менее похожее на логичную прозу или логичную поэзию. Например, очень популярная задача — это генерация поэзии Шекспира, когда [RNN](#)-ка обучается на поэзии Шекспира и дальше ваша сеть может генерировать собственные стихи. Кроме того, можно генерировать даже программный код, который вряд ли будет компилироваться или интерпретироваться, но, тем не менее, будет похож на правду. Можно генерировать музыку, новостные заголовки — что угодно ещё. В качестве домашнего задания мы предлагаем вам сгенерировать мотивационные лозунги. А именно, взять данные из файла, который приложен к этому ноутбуку и называется "author quotes", и попытался научить вашу рекуррентную нейронную сеть генерировать цитаты или какие-то лозунги. Это домашнее задание достаточно простое — всё, что вам нужно — это применить уже готовый код, который мы рассмотрели, в семинаре, к готовым данным и посмотреть на результаты, которые у вас получатся. Возможно немного регулировать температуру генерации, и, в целом, проанализировать результаты, которые вы получите. Удачи с домашним заданием!

Домашнее задание: мотивационные лозунги

```
In [ ]: with open("author_quotes.txt") as input_file:  
    quotes = input_file.read()[:-1].split('\n')  
    quotes = [' ' + line for line in quotes]  
  
In [ ]: quotes[:5]  
  
In [ ]: tokens = list(set(''.join(quotes)))  
token_to_id = {token: idx for idx, token in enumerate(tokens)}  
num_tokens = len(tokens)  
  
In [ ]: # your code here
```

Что еще можно генерировать?

С помощью кода из этого семинара можно генерировать не только имена или цитаты, но и:

- Повести/романы/поэзию/песни любимого автора
- Новостные заголовки
- Программный код
- Молекулы в формате smiles
- Музыку
- Названия мебели из ИКЕА
- etc.

