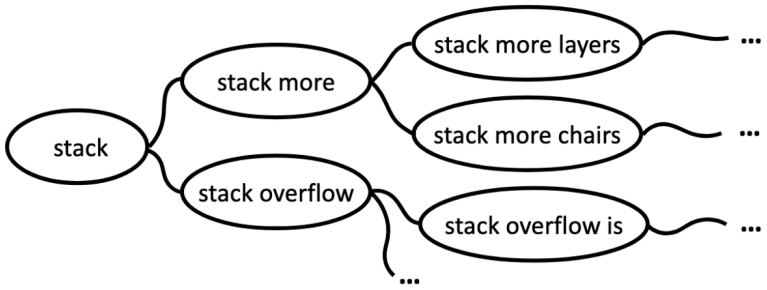
Обработка последовательностей с помощью нейронных сетей

Нерсес Багиян

14 ноября 2019 г.

Мы уже уделили достаточно много внимания нейронным сетям для работы с картинками и обычными табличными данными. Данные методы отлично справляются с поставленными задачами и легко обучаются. Однако, возникает следующий вопрос: "Как правильно обрабатывать данные последовательной природы?". В качестве примера давайте рассмотрим очень популярную в последние годы задачу языкового моделирования. В данной задаче по контексту нам требуется смоделировать продолжение:



Формализуем задачу:

У нас есть первые K_i слов i - ого предложения (длиной T_i) и мы хотим предсказать следущие $T_i - K_i$ слов. Тогда, если попробовать записать это все в привычной нам нотации, то мы получим следующую запись:

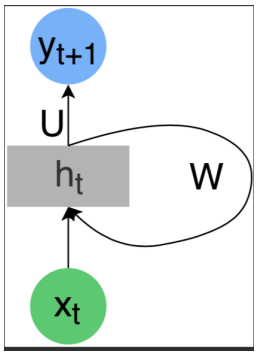
$$X = \{\{x_{t_j}\}_{j=1}^{K_i}, \{x_{t_s}\}_{s=1}^{T_i-K}\}_{i=1}^{l}$$

Внимательный студент легко заметит, что в данной записи $x_i = \{x_{t_j}\}_{j=1}^{K_i}$, что говорит нам о том, что наши тексты имеют разной длины. Это вполне логично, так как каждый человек чаще всего пишет ровно столько сколько ему захочется. Что же делать? Как вариант давайте фиксировать последовательность текста, например, 100 слов и подавать каждый из этих 100 эмбеддингов (размером 100) на вход полносвязному слою. Но сколько параметров мы получим в таком случае? Правильнный ответ больше миллиона. Однако, тут вы можете меня спросить: "1 миллион параметров? С текущими вычислительными мощностями это должно быть легко обучать". Но все не так просто,

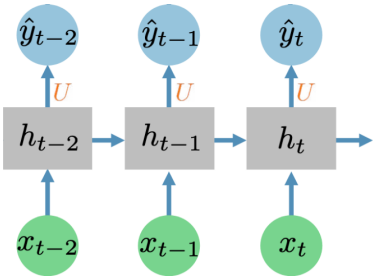
так как, во-первых, это всего лишь один слой (а мы убедились на примере, что чаще всего работает правило the deeper – the better), а во-вторых фиксируя окно размером сто мы можем терять важную для нас информацию. Все эти проблемы плавно подводят нас к следующему виду моделей – рекурентным сетям.

1 Рекурентные нейронные сети

В данном параграфе мы будем продолжать разбираться с RNN на примере языковой модели.



Как следует из названия – в данной архитектуре должна быть некая рекуррентность. Давайте разберемся откуда она берется. Один из самых простых способов понять это – представить, что вы используете один и тот же MLP, с одной и той же матрицей весов W, в которой входом является последовательность токенов (один токен за один проход) и некоторый вектор скрытого состояние h_t. Можно развернуть данную рекурентность и это будет выглядеть следующим образом:



Давайте теперь разберемся как считается каждая компонента данного слоя. Если с x_t все понятно (это эмбеддинги слов), то с h_t и y_t - нет. Как их считать? Начнем с более простого: на каждом шаге нам надо предсказывать слово из фиксированного словаря размером $|V|$. Для этого прекрасно подойдет полносвязный слой с софтмакс активацией:

$$y_t = \text{softmax}(Uh_t)$$

Однако до сих пор остается вопрос как считать h_t (скрытое состояние)? Внимательный читатель мог заметить, что каждый вектор h передается вместе с рекурентностью на вход в слой вместе с зашаренными весами. Тогда:

$$h_t = f(Vx_t + Wh_{t-1})$$

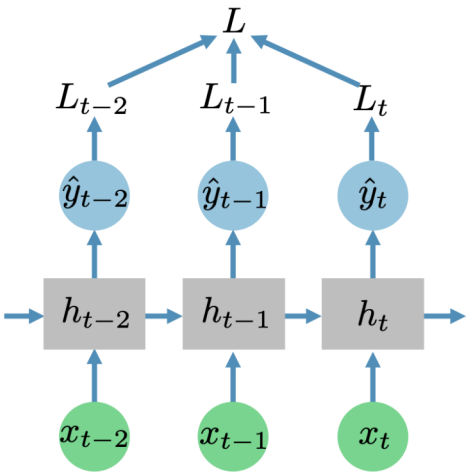
И здесь как раз таки и проявляется та самая рекуррентность из названия. Каждое скрытое состояние подсчитывается на основе предыдущего. Данная особенность позволяет хранить информацию с предыдущих шагов и тем самым, более эффективно работать с последовательностями, например, с предложениями.

2 BPTT

Давайте распишем h_{t+1}:

$$h_{t+1} = f(Vx_{t+1} + Wh_t) = f(Vx_{t+1} + \mathbf{W}f(Vx_t + \mathbf{W}h_{t-1}))$$

Как посчитать градиент по матрице W? Это задача нетривиальная (поэтому мы и выделили ей целый параграф, собственно). Начнем с самого начала, а именно с того, как считается функция потерь:



Суммарная функция потерь – это потеря по каждому предсказанию языковой модели: $L = \sum_i L_i(y_i, \hat{y}_i)$.

Напомним формулы для forward pass RNN:

$$y_t = g(Uh_t)$$
$$h_t = f(Vx_t + Wh_{t-1})$$

,

где g и f - это некоторые функции активации.

Тогда посчитаем $\frac{dL}{dU}$:

$$\frac{dL}{dU} = \sum_i \frac{dL_i}{dU} =< \text{Воспользуемся chain rule} >= \sum_i \frac{dL_i}{d\hat{y}_t}\frac{d\hat{y}_t}{dU} = \sum_i \frac{dL_i}{d\hat{y}_t}\frac{d\hat{y}_t}{dg}\frac{dg}{d(Uh_t)}\frac{d(Uh_t)}{dU}$$

Тут все довольно просто и очевидно, однако, относительно U у нас нет никакой рекурсентости. (когда мы начнем считать $\frac{dL}{dW}$ мы заметим, что вхождение этого параметра будет не единственным)

Посчитаем $\frac{dL}{dW}$:

$$\frac{dL}{dW} = \sum_i \frac{dL_i}{dW} =< \text{Рассмотрим одно слагаемое} >$$
$$\frac{dL_i}{dW} = \frac{dL_i}{d\hat{y}_i}\frac{d\hat{y}_i}{dh_i}\frac{dh_i}{dW}$$

Проблемы в множителе $\frac{dh_i}{dW}$. $h_i = f(Vx_{t+1} + Wh_{i-1})$, в h_{i-1} участвует та же самая матрица W. Давайте попробуем посчитать это, начнем с производной для h_2:

$$h_1 = f(Vx_1 + Wh_0)$$
$$h_2 = f(Vx_2 + Wh_1)$$
$$\frac{dh_2}{dW} = \frac{df}{d(Vx_2 + Wh_1)}\frac{d(Vx_2 + Wh_1)}{dW}$$

Рассмотрим слагаемое Wh_1. Обе функции зависят от W, тогда:

$$\frac{d(Wh_1)}{dW} = h_1 + W\frac{dh_1}{dW}$$

Если мы посмотрим на h_1 внимательно, то заметим, что она зависит только от h_0, которая является константой (случайно инициализируется во время обучения). Тогда обозначим это за $\frac{dh_1}{dW_*}$ (в знак того, что данное выражение уже не рассматривается как функция от W).

Запишем теперь все вместе:

$$\frac{dh_2}{dW} = \frac{df}{d(Vx_2 + Wh_1)}\frac{d(Vx_2 + Wh_1)}{dW} = \frac{df}{d(Vx_2 + Wh_1)} * (h_1 + W\frac{dh_1}{dW}) =$$
$$= \frac{df}{d(Vx_2 + Wh_1)}h_1 + \frac{df}{d(Vx_2 + Wh_1)}W\frac{dh_1}{dW}$$

Давайте упростим выражение. Для этого честно посчитаем производную $\frac{dh_2}{dh_1}$:

$$\frac{dh_2}{dh_1} = \frac{df}{d(Vx_2 + Wh_1)}W$$

Теперь перепишем снова:

$$\frac{dh_2}{dW} = \frac{df}{d(Vx_2 + Wh_1)}h_1 + \frac{dh_2}{dh_1}\frac{dh_1}{dW}$$

Мы уже выяснили, что $\frac{dh_1}{dW}$ легко считается и договорились ее обозначит за $\frac{dh_1}{dW_*}$, тогда:

$$\frac{dh_2}{dW} = \frac{df}{d(Vx_2 + Wh_1)}h_1 + \frac{dh_2}{dh_1}\frac{dh_1}{dW_*}$$

Осталось посчиать $\frac{df}{d(Vx_2+Wh_1)}h_1$, так как данное слагаемое уже легко считается и на самом деле данное выражение уже не рассматривается как функция от W, то можно обозначить это за $\frac{df}{d(Vx_2+Wh_1)}h_1 = \frac{dh_2}{dW_*}$
Итого производная по h_2 получится следующая:

$$\frac{dh_2}{dW} = \frac{dh_2}{dW_*} + \frac{dh_2}{dh_1}\frac{dh_1}{dW_*}$$

Посчитаем производную по h_3:
У нас уже есть h_2:

$$\frac{dh_2}{dW} = \frac{dh_2}{dW_*} + \frac{dh_2}{dh_1}\frac{dh_1}{dW_*}$$

Выпишем $\frac{dh_3}{dW}$:

$$\frac{dh_3}{dW} = \frac{df}{d(Vx_3 + Wh_2)}(h_2 + W\frac{dh_2}{dW}) =$$
$$= \frac{df}{d(Vx_3 + Wh_2)}W(\frac{dh_2}{dW_*} + \frac{dh_2}{dh_1}\frac{dh_1}{dW_*}) + \frac{df}{d(Vx_3 + Wh_2)}h_2$$

Давайте упростим выражение. Для этого честно посчитаем производную $\frac{dh_2}{dh_1}$:

$$\frac{dh_3}{dh_2} = \frac{df}{d(Vx_3 + Wh_2)}W$$

Теперь перепишем снова:

$$\frac{dh_3}{dW} = \frac{df}{d(Vx_3 + Wh_2)}(h_2 + W\frac{dh_2}{dW}) = \frac{dh_3}{dh_2}(\frac{dh_2}{dW_*} + \frac{dh_2}{dh_1}\frac{dh_1}{dW_*}) + \frac{dh_3}{dW_*}$$

Результат:

$$\frac{dh_3}{dW} = \frac{dh_3}{dh_2}\frac{dh_2}{dh_1}\frac{dh_1}{dW_*} + \frac{dh_3}{dh_2}\frac{dh_2}{dW_*} + \frac{dh_3}{dW_*}$$

Тогда по индукции легко доказать, что:

$$\frac{dh_k}{dW} = \sum_{i=1}^{k}(\prod_{j=i+1}^{k}\frac{dh_j}{dh_{j-1}})\frac{dh_i}{dW_*}$$

Выдохнули.

3 LSTM

Однако, у данной реализации рекурентных сетей есть проблемы: $\prod_{j=i+1}^{k}\frac{dh_j}{dh_{j-1}}$ либо очень быстро затухает либо очень быстро взрывается. (так как много слагаемых в произведении, норма которых далеко не единица).

Что делать со взрывом? Тут все довольно просто: ограничивать норму градиентов. Если норма больше какого-то порога *threshold*, то мы просто приводим ее к значению *threshold* следующим образом:

$$\begin{aligned}&\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}\\ &\textbf{if } \|\hat{\mathbf{g}}\| \geq threshold \textbf{ then}\\ &\quad \hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|}\hat{\mathbf{g}}\\ &\textbf{end if}\end{aligned}$$

Также можно делать инициализацию из SVD разложения случайной матрицы:

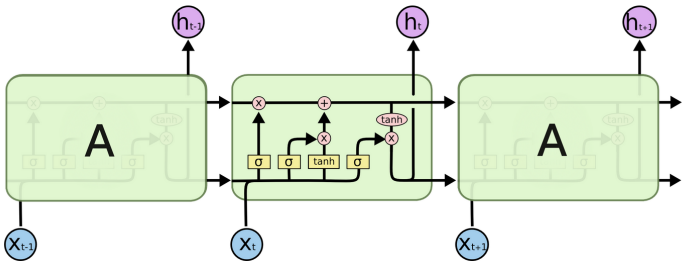
```
flat_shape = (shape[0], np.prod(shape[1:]))
a = get_rng().normal(0.0, 1.0, flat_shape)
u, _, v = np.linalg.svd(a, full_matrices=False)
# pick the one with the correct shape
q = u if u.shape == flat_shape else v
q = q.reshape(shape)
```

С затуханием все чуть-чуть сложнее и в 1999 году была придуман слой под названием LSTM, давайте разберемся с ним подробнее. Его название расшифровывается как

Long Short Term Memory слой. Из названия становится понятно, что данный слой содержит в себе как длинную, так и короткую память. И, как обычно, за оба вида памяти отвечают некоторые вектора, обозначим их c_t и h_t.

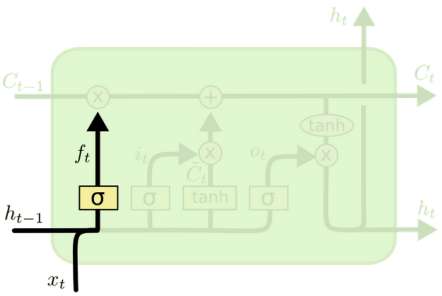
Работу данного слоя можно разделить на несколько частей:

- Работа forget gate – основываясь на информации о скрытом состоянии h_t слой выбирает, что ему нужно забыть
- Работа input gate – основываясь на входе слоя и скрытом состоянии h_t подсчитывает вклад текущего токена в решение общей задачи
- Работа output gate – выдает c_t и h_t наружу.



Разберемся, что есть что:

Forget gate

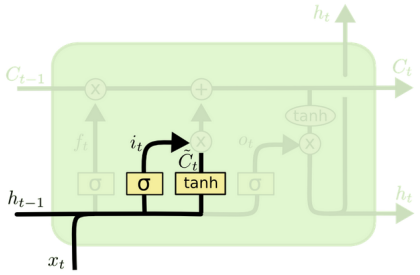


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Данная часть слоя пытается исключить ненужную информацию из скрытого состояния для того, заменив ее информацией из нового токена x_t. Если вспомнить наш пример про языковую модель, то вы можете представить как вместе с x_t появляется новая информация о происходящем в тексте и forget gate, изабавляется от старой уже нерелвантной информации.

Input gate

Input gate, в свою очередь, продолжает дело forget gate, однако вместо выучивания того, что стоит забыть, он выучивает новую информацию из входного токена x_t.

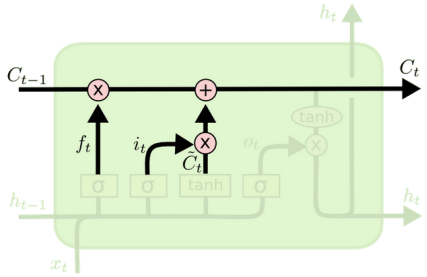


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

Далее, мы выучиваем новый вектор $\widetilde{C}_t$, основываясь на иполной информации, которую мы имеем о короткой памяти h_{t-1} и токене x_t. Нам осталось только обновить вектор длинной памяти на основе того, что мы решили забыть и оставить, а также вычислить новый вектор короткой памяти. Этим занимается

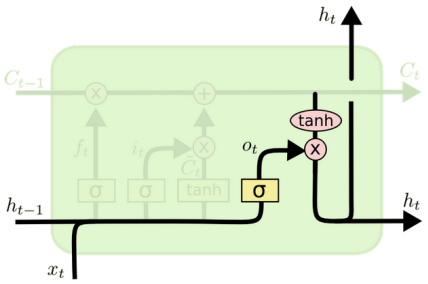
Output gate

Вычисляет c_t и h_t:



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Вектор c_t в данном случае является линейной комбинацией того, что мы решили забыть и выучить. Мы убрали оттуда старую, уже неактуальную информацию и добавили новую, более акутальную.



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Далее, используя данный вектор мы можем вычислить короткую память просто убрав из него ненужную информацию.

Подводя итог, нужно отметить несколько следующую вещь: LSTM должна бороться с затуханием градиентов. Как она решает проблему? Ответ: добавив вектор длинной памяти мы прокинули информацию от самого входа, до самого конца. Тут можно провести аналогию со skip-connection, которые прокидывают сигнал дальше и позволяют градиентам не затухать. Также, добавив cell-state, мы позволяем нейросети извлекать информацию даже с самых первых шагов, это помогает в задачах, где информация с любого шага очень критична