Рекуррентные нейронные сети. Проблема затухающего градиента

Цель занятия

В результате обучения на этой неделе:

- вы узнаете, как строится рекуррентная нейронная сеть (RNN);
- познакомитесь с основными разновидностями архитектуры RNN — LSTM (Long short-term memory), или долгая краткосрочная память; и GRU (Gated Recurrent Units), или управляемые рекуррентные блоки;
- рассмотрите проблемы затухающих и взрывающихся градиентов.

План занятия

Конспект занятия

1. Языковое моделирование

Начнем тему языкового моделирования с нескольких иллюстраций (изучить подробно их можно в статье).

Шекспир:

```
PANDARUS:
Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:
Well, your wit is in the care of side and that.

Second Lord:
They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:
Come, sir, I will make did behold your worship.
```

Отрывок из статьи по алгебраической геометрии (Latex):

Proof. Omitted. □

Lemma 0.1. *Let $\mathcal{C}$ be a set of the construction.*

Let $\mathcal{C}$ be a gerber covering. Let $\mathcal{F}$ be a quasi-coherent sheaves of $\mathcal{O}$-modules. We have to show that

$$\mathcal{O}_{\mathcal{O}_X} = \mathcal{O}_X(\mathcal{L})$$

.

Proof. This is an algebraic space with the composition of sheaves $\mathcal{F}$ on $X_{\acute{e}tale}$ we have

$$\mathcal{O}_X(\mathcal{F}) = \{morph_1 \times_{\mathcal{O}_X} (\mathcal{G}, \mathcal{F})\}$$

where $\mathcal{G}$ defines an isomorphism $\mathcal{F} \to \mathcal{F}$ of $\mathcal{O}$-modules. □

Lemma 0.2. *This is an integer $\mathcal{Z}$ is injective.*

Proof. See Spaces, Lemma ??. □

Lemma 0.3. *Let S be a scheme. Let X be a scheme and X is an affine open covering. Let $\mathcal{U} \subset \mathcal{X}$ be a canonical and locally of finite type. Let X be a scheme. Let X be a scheme which is equal to the formal complex.*

The following to the construction of the lemma follows.

Let X be a scheme. Let X be a scheme covering. Let

$$b : X \to Y' \to Y \to Y \to Y' \times_X Y \to X.$$

be a morphism of algebraic spaces over S and Y.

Исходный код ядра Linux на языке C:

```c
/*
 * If this error is set, we will need anything right after that BSD.
 */
static void action_new_function(struct s_stat_info *wb)
{
  unsigned long flags;
  int lel_idx_bit = e->edd, *sys & ~((unsigned long) *FIRST_COMPAT);
  buf[0] = 0xFFFFFFFF & (bit << 4);
  min(inc, slist->bytes);
  printk(KERN_WARNING "Memory allocated %02x/%02x, "
    "original MLL instead\n"),
    min(min(multi_run - s->len, max) * num_data_in),
    frame_pos, sz + first_seg);
  div_u64_w(val, inb_p);
  spin_unlock(&disk->queue_lock);
  mutex_unlock(&s->sock->mutex);
  mutex_unlock(&func->mutex);
  return disassemble(info->pending_bh);
}
```

Все три примера — это не текст, написанный людьми. Это все написано рекуррентной нейронной сетью. В тексте расставлено правильно форматирование, соблюдается рифма, длина строк. Второй и третий примеры компилируются. Latex освоить довольно непросто, это аналог HTML — некоторый язык разметки.

Рассмотрим, как правильно поставить задачу языкового моделирования, чтобы решать ее с помощью уже изученных техник.

Языковое моделирование — это предсказание слова или токена, которое следует за текущим текстом. **Токен** — атомарная сущность последовательности текста (слово, слог, символ, морфема).

Условимся, что сейчас мы работаем на уровне символов. Их конечное число — символы верхнего и нижнего регистра, знаки препинания, пробел. У нас есть фиксированный словарь, из которого мы на каждом шаге выбираем символ. Мы шаг за шагом генерируем новый токен, обуславливаясь на предыдущий контекст (на левый контекст).

Предположим, что мы умеем представлять текущую подпоследовательность в виде вектора фиксированной последовательности. Это нетривиальная задача, поскольку проблема последовательности в том, что она все время меняет свою длину.

Декомпозируем задачу на две части. Предположим, что мы уже умеем представлять последовательность в виде вектора, и подумаем, как нам сгенерировать следующий токен. Предсказание токена из словаря — задача классификации.

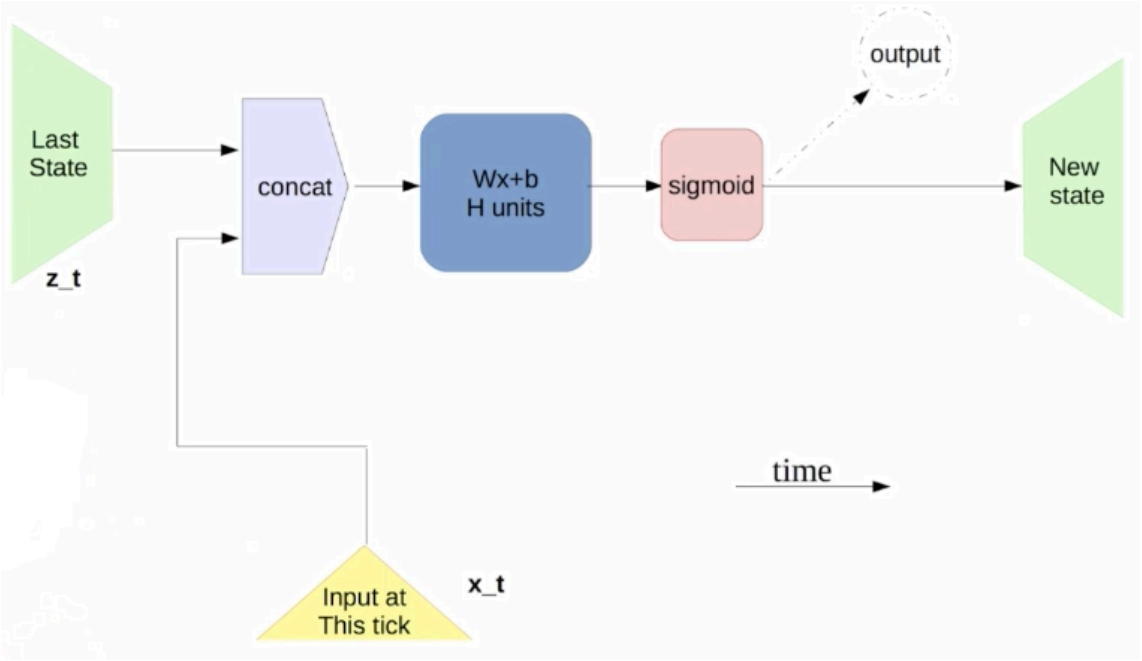
2. Рекуррентные нейронные сети

Пусть у нас есть несколько гипотез, на основании которых мы будем двигаться дальше:

1. Словарь фиксированной размерности.
2. Левый контекст — некоторый фиксированный вектор.
3. При обогащении контекста чем-то новым условимся, что и левый вектор контекста, и вектор для нового токена фиксированной размерности. И мы из них каким-то образом умеем получать новое представление.

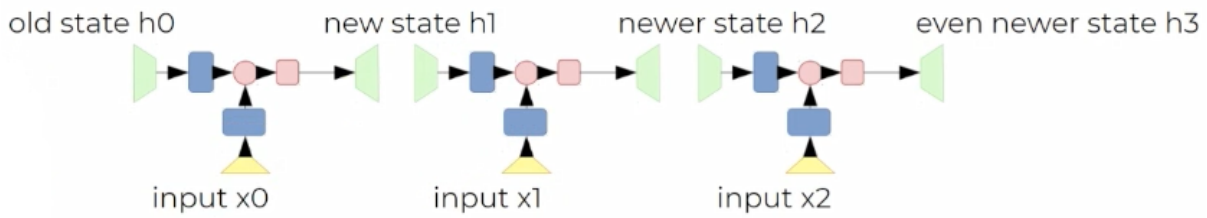
При выполнении этих условий по индукции мы сможем генерировать текст.

Какой наиболее простой способ преобразовать два вектора размерности M и N в новый вектор размерности M? Сконкатенируем два вектора и умножим на матрицу размерности M + N x M.



Разберемся, зачем нам в этой схеме сигмоида.

Рассмотрим представление рекуррентной нейронной сети, развернутой во времени:



Если последовательно развернуть данное преобразование шаг за шагом, мы получим, что преобразование линейное. Композиция линейных преобразований есть линейное преобразование. Поэтому между этими линейными преобразованиями стоит нелинейная функция — сигмоида или гиперболический тангенс.

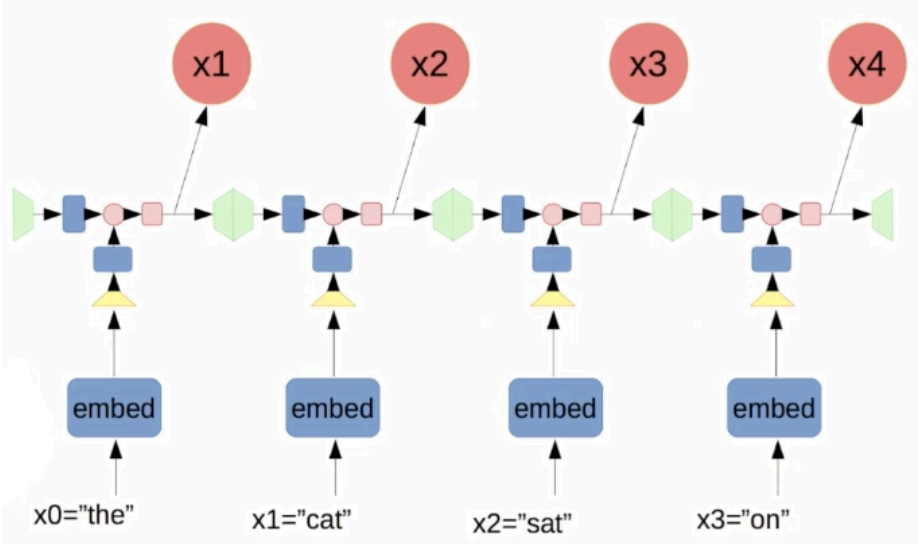
И сигмоида, и тангенс — не очень хорошие функции активации. Они насыщаются на краях и из-за этого приводят к затуханию градиентов, дороги с точки зрения количества вычислений для экспоненты. ReLU работает гораздо быстрее. Но тем не менее используют сигмоиду или гиперболический тангенс.

У нас есть некоторое скрытое состояние, которое кодирует левый контекст. Как закодировать левый контекст, если там пока ничего нет (предыдущее представление, когда вообще ничего пока не знаем)? Мы можем взять вектор из нулей размерности скрытого состояния.

Преобразование применяется шаг за шагом. Каждый шаг обладает одними и теми же весами. У нас линейный слой с функцией активации, который мы применяем каждый раз к каждому инпуту. Мы формализуем наше преобразование, как «обновить текущий контекст с помощью нового кусочка информации».

Почему мы используем сигмоиду или тангенс вместо ReLU? У сигмоиды и гиперболического тангенса фиксированная область значений. Независимо от входных данных мы всегда получим вектор фиксированной длины. На каждом шаге мы обновляем одни и те же веса.

В результате текст формируется следующим образом:



Теперь на каждом шаге мы можем предсказывать следующее слово, которое пока не знаем. И нам не нужна обучающая выборка, нам достаточно того текста, который был сформирован человеком. Мы можем взять любой текст и обучить на нем языковую модель (собрание литературы, википедия, новости из интернета и т. д.). Текст — это уже автоматически размеченные данные.

Рассмотрим работу рекуррентной сети формально.

Сначала у нас не было никаких данных, вектор контекста нулевой:

$$h_0 = \bar{0}$$

Прочитали первый символ, обновили вектор контекста:

$$h_1 = \sigma\left(< W_{hid}\left[h_0, x_0\right] >+ b\right)$$

где W_{hid} — матрица параметров,

σ — функция сигмоиды.

$$h_2 = \sigma\left(< W_{hid}\left[h_1, x_1\right] >+ b\right) = \sigma\left(< W_{hid}\left[\sigma\left(< W_{hid}\left[h_0, x_0\right] >+ b\right), x_1\right] >+ b\right)$$

По индукции:

$$h_{i+1} = \sigma\left(< W_{hid}\left[h_i, x_i\right] >+ b\right)$$

При многоклассовой классификации (а мы выбираем наиболее подходящее слово/токен из словаря) из кучи возможных вариантов мы для каждого шага предсказываем вероятность следующего токена для всех возможных из словаря:

$$P\left(x_{i+1}\right) = softmax\left(< W_{out}, h_i >+ b_{out}\right)$$

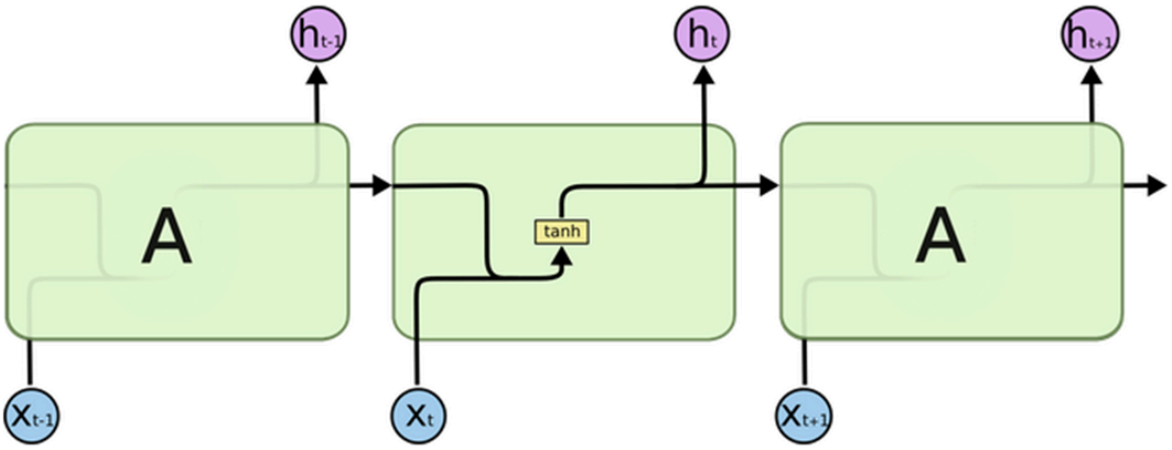
В результате исходя из базовых предположений, мы изобрели рекуррентную нейронную сеть.

У рекуррентной нейронной сети есть свои сложности, и она требует доработки.

3. RNN

Понимание того, как работает рекуррентная нейронная сеть, является ключевым фактором к пониманию обработки последовательностей. И в целом к пониманию того, как можно обрабатывать структуру в наших данных.

Рассмотрим простейшую иллюстрацию классической «ванильной» рекуррентной нейронной сети (RNN):



Данная RNN содержит лишь один вектор скрытого состояния, который кодирует всю информацию о прочитанной до текущего момента последовательности. RNN умеет обновлять это скрытое состояние посредством простой операции.

У нас есть некоторый вектор контекста, который обладает двумя свойствами:

- он фиксированной размерности;
- в него мы будем пытаться закодировать всю информацию о левом контексте в текущей последовательности.

На картинке x_t — эмбединг от нового элемента последовательности.

h_{t-1}, h_t, h_{t+1} — векторы скрытого состояния для каждого этапа RNN. На основании вектора h можно пытаться предсказывать следующий токен или метку класса для данного элемента последовательности.

Пример. Задача расстановки частей речи.

Поверх существующих состояний можно присоединить сверху еще одну RNN и получить двухслойную RNN. Это позволяет улавливать более сложные зависимости.

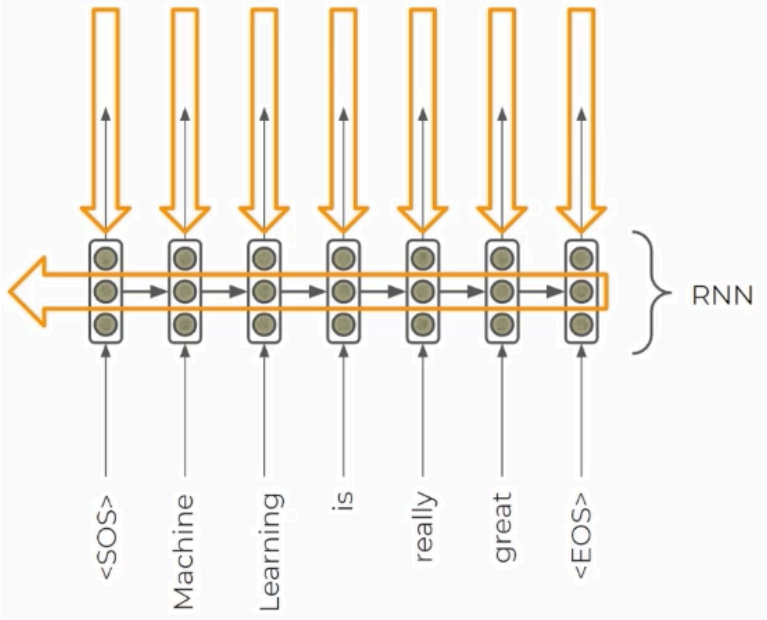
Мультислойные RNN улавливают больше информации, но приходится за это расплачиваться большим количеством операций и параметров, способностью переобучиться. Но в текстах вся информация доступна с самого начала, и машина может прочитать текст как слева направо, так и справа налево. Здесь важно, что есть упорядоченность в данных.

Статический набор данных можно обрабатывать в двустороннем порядке, получить два вектора и сконкатенировать их между собой. Теперь будет вектор, который обусловлен на весь левый контекст и на весь правый контекст.

RNN не полностью симулируют процесс человеческого восприятия, они обогащают контекст с каждым новым кусочком информации. Если мы читаем слева направо, то при прямом проходе последнее состояние будет больше обусловлено на правую часть, на конец последовательности. А если прочитаем справа налево, то вектор будет обусловлен на левую часть — начало во временном смысле нашей последовательности.

Объединив векторы между собой, мы получаем два соединенных воедино вектора, обусловленных и на левую часть последовательности, и на правую. Но при этом надо учитывать и обрабатывать в 2 раза больше информации. Расплачиваемся, как всегда, вычислительной мощностью. Мультислойные RNN на практике встречаются при построении системы машинного перевода.

Схема обучения RNN:



Если мы что-то предсказываем на каждом шаге (как, например, в задаче языкового моделирования), то на каждом шаге у нас решается некоторая задача (классификации для языкового моделирования, регрессии для предсказания временного ряда). Получаем предсказания (есть ошибка и градиенты с каждого из шагов), после чего нужно обновить параметры нашей модели. На каждом шаге параметры одинаковые, в результате их достаточно накопить (все преобразования дифференцируемые). Все градиенты влияют на один и тот же вектор весов.

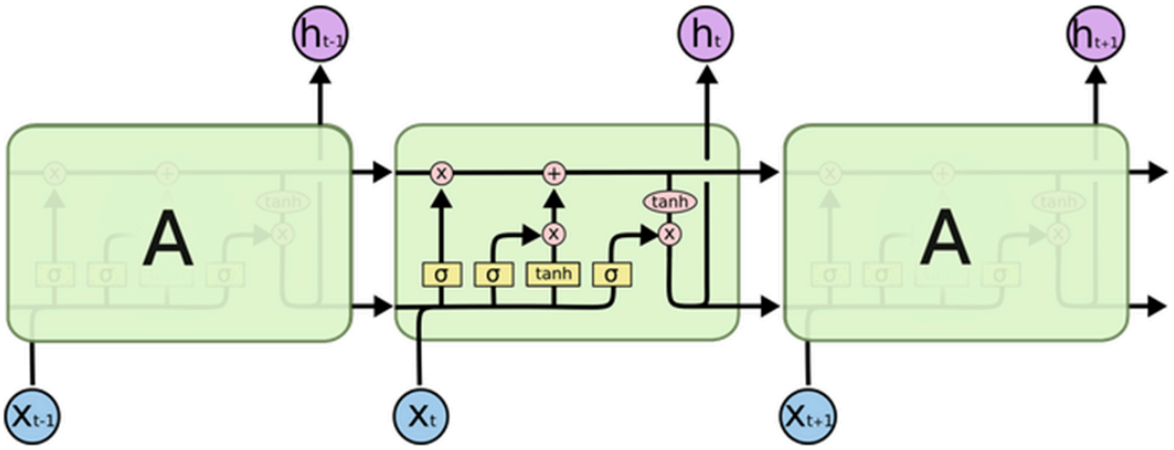
В RNN есть несколько проблем:

- Мы используем только один вектор для кодирования всего контекст.
- Функция активации (тангенс или сигмоида) зануляет нам градиенты на своих «плечах», то есть почти на всей числовой оси мы будем получать нулевые градиенты. Так или иначе, градиенты будут затухать.

Проблему затухающего градиента можно решать со стороны прямого прохода, где можно «забывать» информацию, и со стороны обратного прохода, где мы можем потерять градиенты. Для этого нужно перейти от «ванильной» RNN к LSTM.

4. LSTM

LSTM (Long Short Term Memory) была впервые представлена в 1997 году.



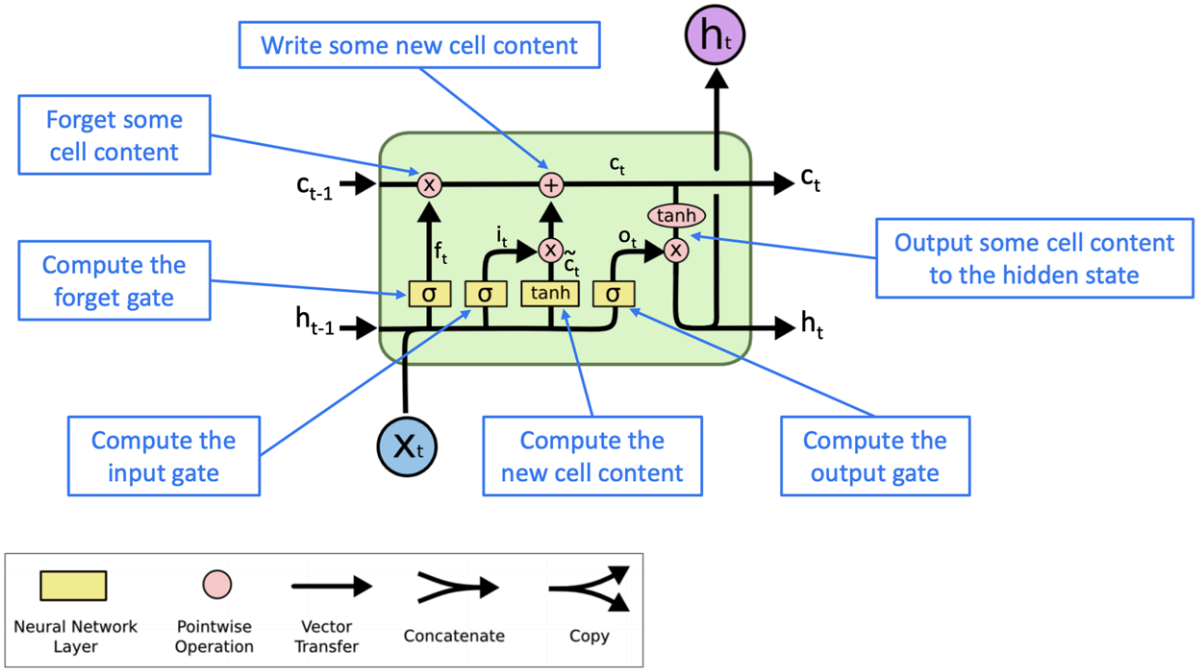
Видим, что теперь вместо одного скрытого состояния у нас их два. Назовем верхнюю стрелку долгосрочной памятью, а нижнюю — локальным контекстом. Мы используем долгосрочную память, чтобы взять кусочек информации и «положить» в долгосрочную память. Это аналог записного блокнота, куда мы фиксируем нужную информацию.

Мы накладываем на долгосрочную память важное ограничение. На пути этого вектора нет никаких функций активации. Единственные происходящие с ним изменения — поэлементное умножение и сложение. То есть никакого зануления градиентов из-за нулевых производных на краях функции активации нет.

Чтобы сохранять как можно больше «полезной» информации, нужно научиться отделять «полезную» информацию от «бесполезной».

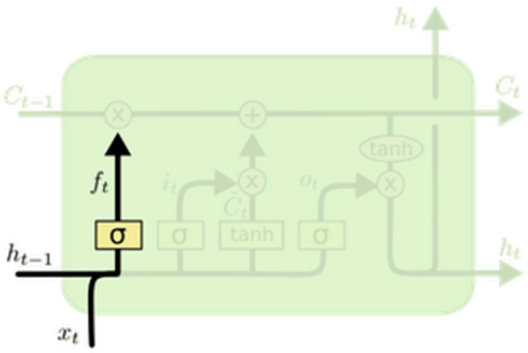
Как понять, что полезно, а что бесполезно? Это похоже на задачу бинарной классификации — логистической регрессии.

Рассмотрим ячейку LSTM:



В данной схеме умножение и сложение — поэлементные операции для векторов.

Первая операция в схеме:



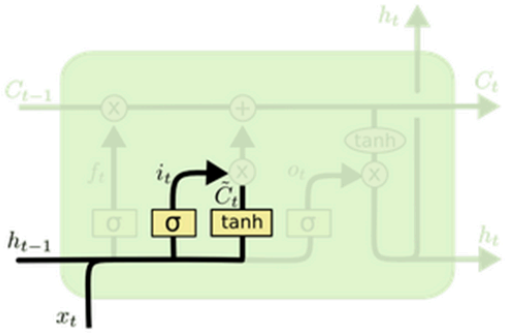
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Происходит конкатенация вектора локального контекста и вектора инпута. Теперь можно решить, что нам бесполезно в нашей долгосрочной памяти.

Выкинуть кусок информации или не выкинуть — задача бинарной классификации. Векторное представление долгосрочной памяти не упорядочено. Информация кроется в значении векторов, а не в их порядке.

В результате для этого вектора считаем N логистических регрессий, где N — число элементов вектора долгосрочной памяти. Для каждой такой логистической регрессии предсказанием является вероятность сохранить то, что лежало в долгосрочной памяти. Умножение на 1 — полностью сохраняем, на 0 — полностью забываем. Между 0 и 1 — понижаем амплитуду имеющихся значений. Бинарную классификацию «забыть/запомнить» называют **вратами забывания (forget gate)**.

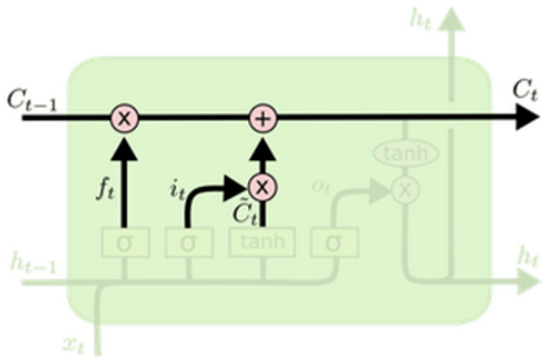
Мы выкинули все ненужное, теперь вопрос: что есть нужное?



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

Теперь мы генерируем себе кандидата на запоминание — сконкатенировали, умножили на матрицу, как в ванильной RNN. Но теперь задаем другой вопрос для бинарной классификации: что из этого мы хотим запомнить. Мы хотим запомнить только то, что будет полезно. Новая логистическая регрессия σ именно за это будет отвечать.

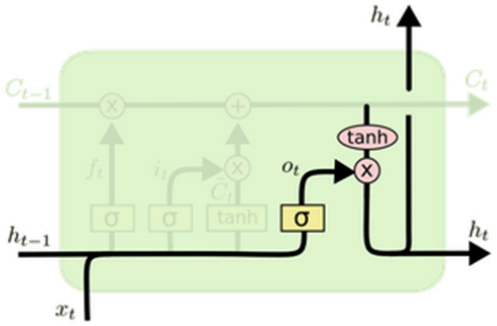
После этого обновляем вектор долгосрочной памяти: забыли и добавили то, что нужно запомнить.



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

В LSTM проблема затухающего градиента будет гораздо меньше.

Остается вопрос: что с этим дальше делать?

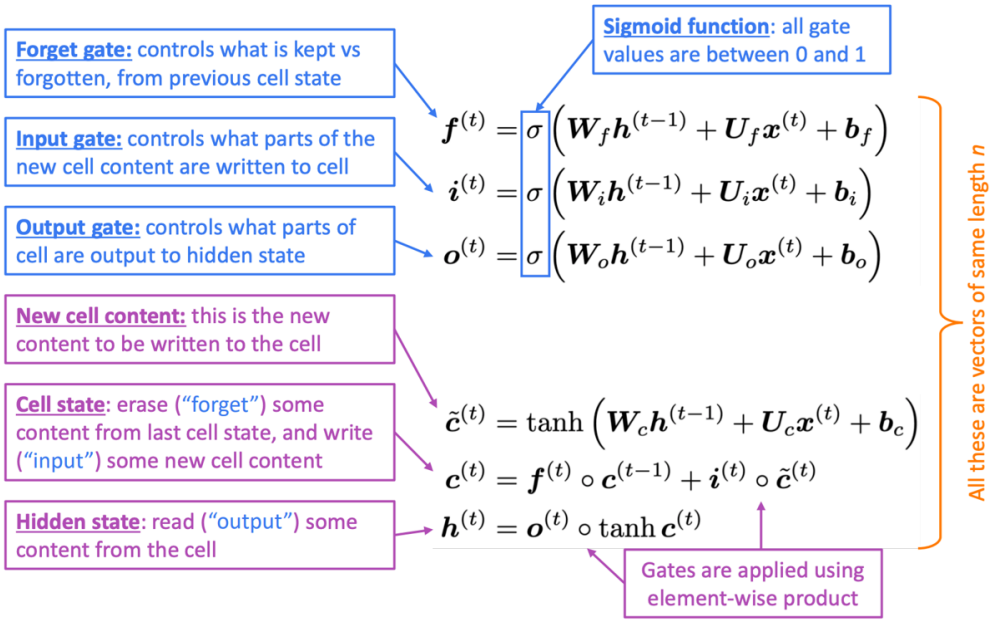


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Теперь на основании локального контекста и долгосрочной памяти объединяем «что было» и «что стало». Долгосрочную память мы обновили, все важное мы туда уже положили.

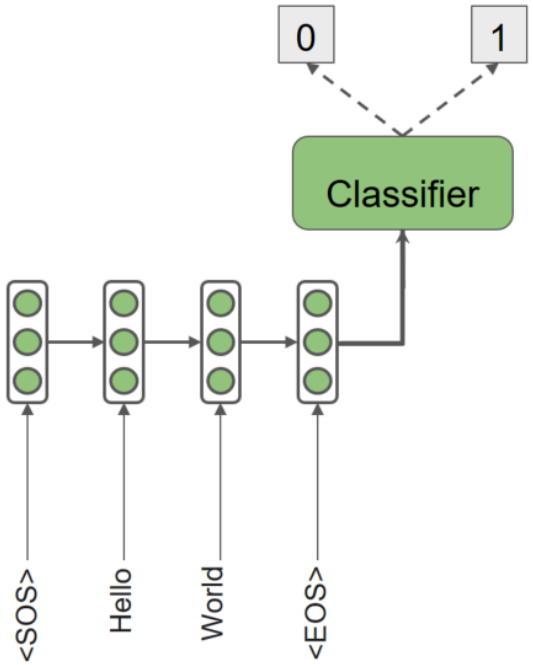
Вопрос в том, что нам важно знать оттуда на текущий момент времени, а не в будущем. На копирование долгосрочной памяти мы применяем гиперболический тангенс и генерируем еще один набор вероятностей выбрать нужный кусок информации с помощью еще одного бинарного классификатора. По нему мы определим, что на данный момент важно, а что не важно. Отсюда порождается новый вектор локального контекста h_t.

LSTM в формальном виде:



LSTM позволяет работать с гораздо более длительными последовательностями.

RNN могут использоваться в качестве энкодеров для последовательных данных.

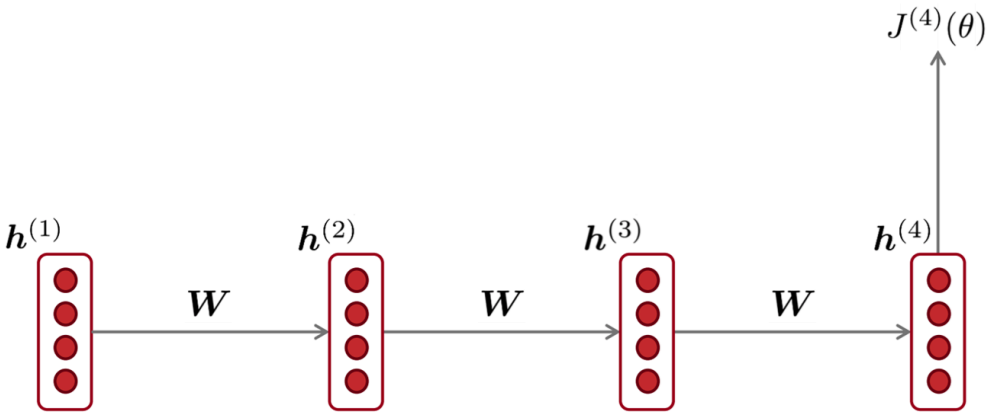


При использовании RNN для кодирования каких-то последовательностей данных мы получаем предсказание только на последнем шаге. Последнее состояние является эмбедингом всей предшествующей последовательности. По сути, оно кодирует весь левый контекст.

Градиенты текут от конца до начала и могут затухнуть где-то в середине. А значит, функция потерь не сможет повлиять на то, как модель обработала первые элементы нашей последовательности. На нее будут больше влиять данные на конце последовательности.

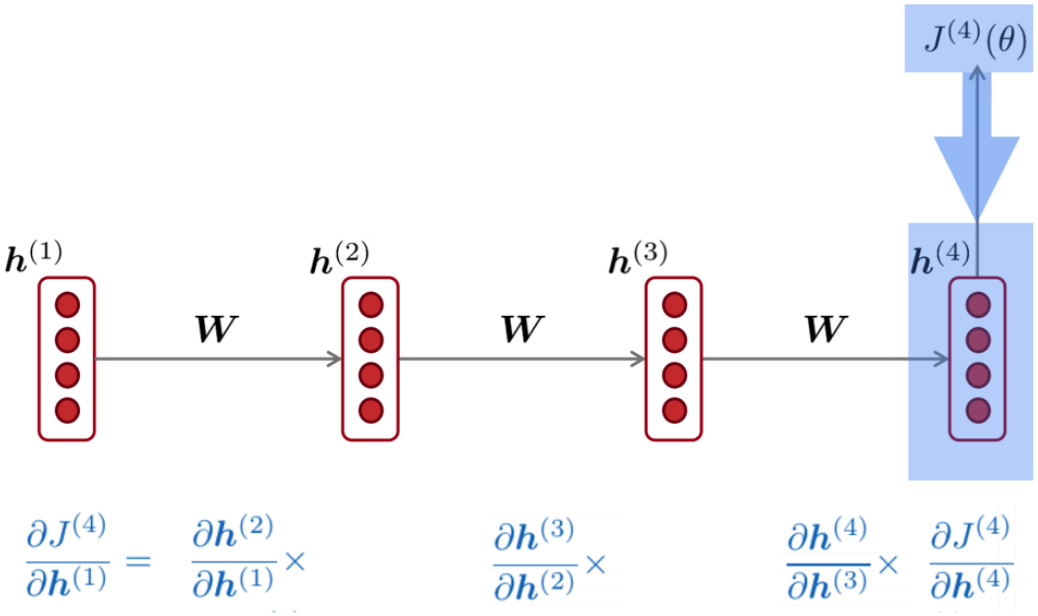
5. Проблема затухающих градиентов

Пусть у нас есть некоторая последовательность состояний:



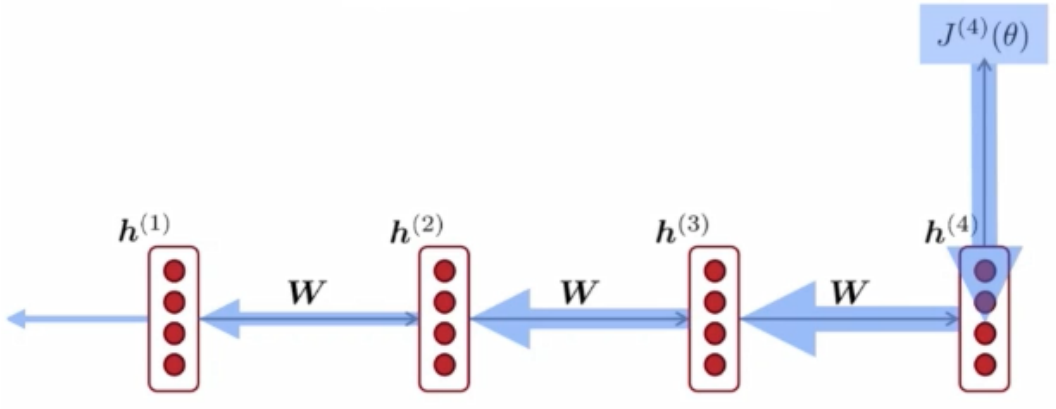
На каждом шаге применяется один и тот же вектор весов, скрытые состояния при этом разные. В данной схеме на 4-м шаге делается какое-то предсказание, оттуда приходит некоторый градиент.

Подсчет градиента на каждом этапе (BackPropagation, BackProp):



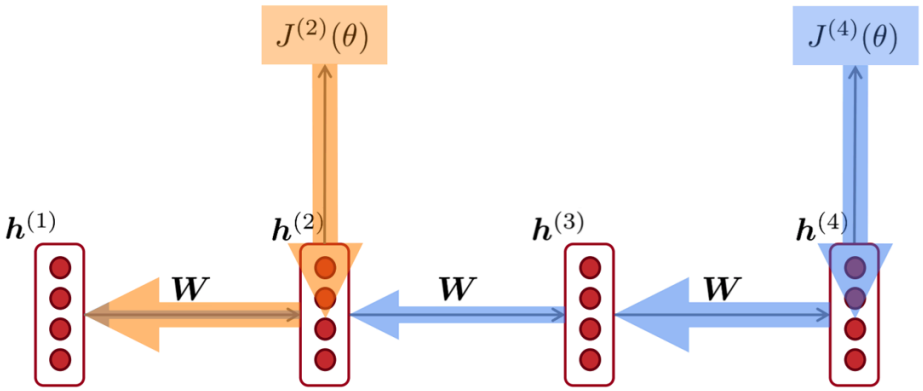
Каждая из частных производных влияет на градиент. Если любой из этих элементов близок к нулю, то произведение этих частных производных даст что-то близкое к машинному нулю. То есть градиенты затухнут.

На данной картинке толщиной стрелки показана амплитуда градиентов:



Подробнее о затухающих градиентах можно прочитать в статье.

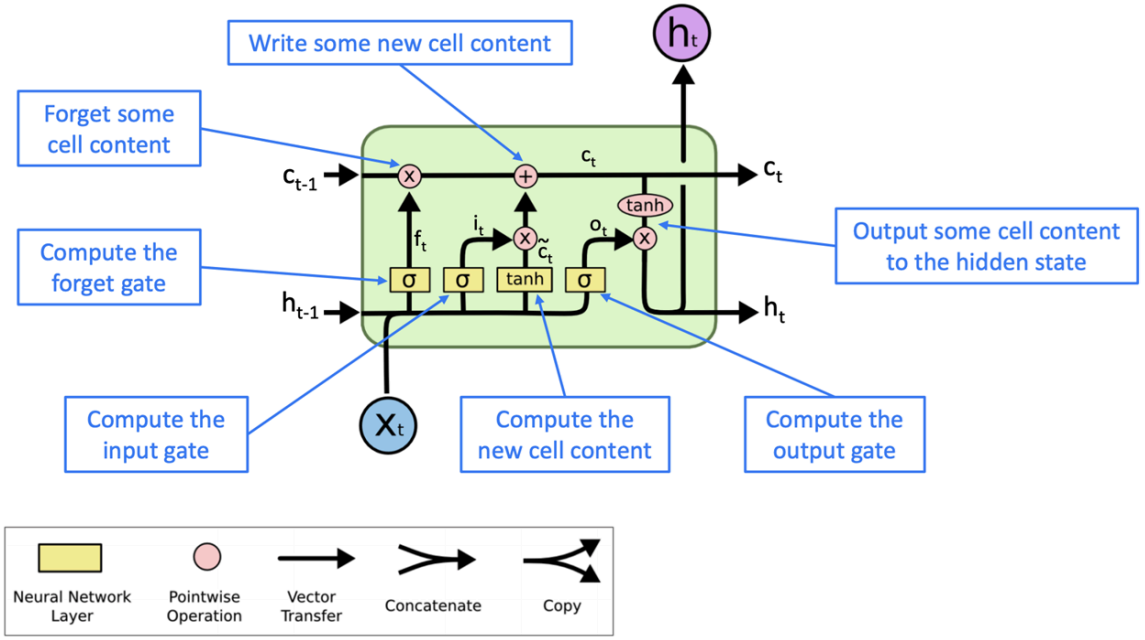
Затухающие градиенты — проблема для языкового моделирования. Обратим внимание на следующую иллюстрацию:



Здесь мы делаем предсказание на втором шаге (желтые стрелки) и на четвертом (синие стрелки). Посмотрим, каким образом градиенты от 2-го и 4-го шага придут к началу. Первый элемент последовательности повлиял на предсказание и на 2-м шаге, и на 4-м. При этом градиент с 4-го шага сильно меньше градиента со 2-го.

Наша модель «предпочитает» обращать внимание на краткосрочные зависимости. Значит, модель в задаче языкового моделирования точно так же будет «страдать» и не будет способна запомнить «дальнюю» информацию. Например, если она когда-то открыла скобку, она может забыть ее закрыть через несколько шагов.

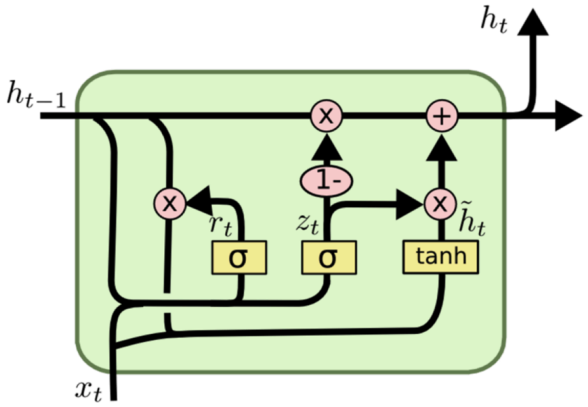
Как решить проблему затухающего градиента? Вспомним LSTM:



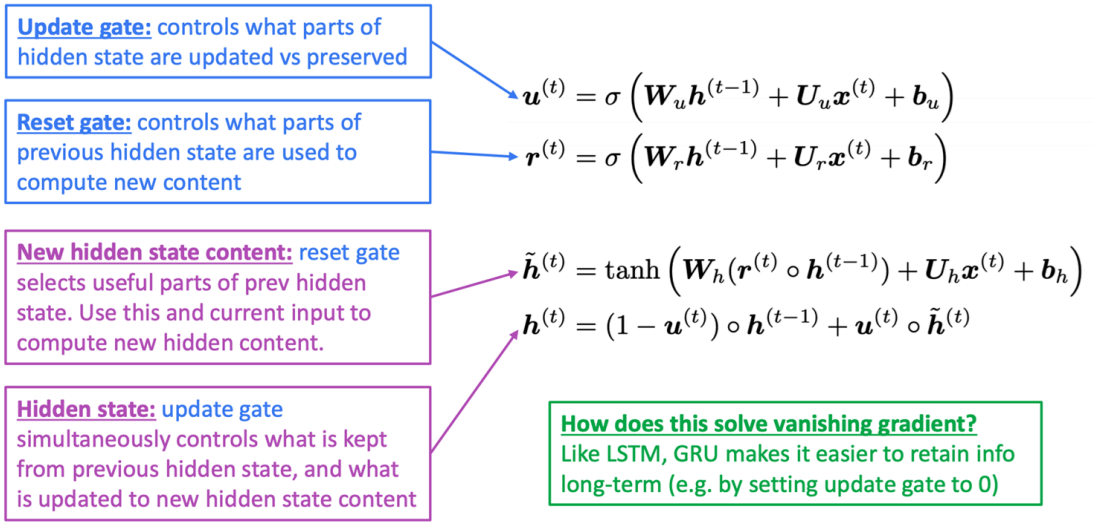
По верхней стрелке у LSTM градиент будет течь назад, практически не меняясь.

GRU

GRU (Gated Recurrent Unit) — еще один вариант доработанной версии рекуррентной нейронной сети.



В GRU чуть меньше операций. У него один поток информации, который раздваивается на каждом шаге. То есть мы берем поток информации и делаем его копию. С копией проделываем некоторые преобразования и обновляем поток, используя результат применения наших копий.



В GRU одно скрытое состояние, меньше параметров. Градиенты путешествуют сквозь вектор, не теряясь на протяжении достаточно большого числа шагов.

GRU, как и LSTM, является классическим механизмом в RNN и де-факто стандартом.

Краткие выводы для LSTM и GRU

- В LSTM чуть больше преобразований, поэтому она чуть медленнее работает, чем GRU.
- LSTM чуть лучше запоминает информацию в долгосрочном смысле.
- LSTM более популярна, широко известна и достаточно интуитивно понятна.

Проблема затухающих градиентов вне RNN

Затухающие градиенты свойственны не только RNN, они свойственны любым глубоким нейронным сетям.

Глубокие нейронные сети — основная идея работы со сложными данными. Чем дольше мы преобразуем наши данные последовательно, чем дальше от начала

преобразований будет функция потерь, тем меньше градиентов дойдет до самого начала.

Боролись с этим по-разному. В 2014-м предложили GoogleNet, которая имела 3 выхода (после первой трети, посередине и в конце). В 2015-м — ResNet, который предложил прямые или скип-коннекшены, уходивших в часть преобразований.

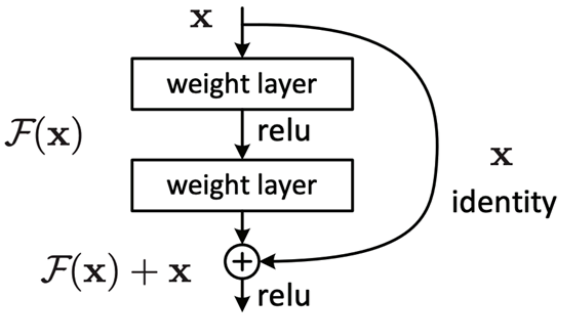


Figure 2. Residual learning: a building block.

Картинка взята из статьи. На данном изображении стрелка вокруг означает тождественное преобразование, которое позволяет обойти некоторые локальные преобразования (умножение на матрицу, сверточный слой, ReLU и т. д.). Схема очень похожа на GRU. То есть ResNet эксплуатирует ту же самую идею — использовать дополнительный поток информации, в котором будет храниться все, что нам нужно.

Чем глубже сеть, тем сложнее ее настроить.

6. Проблема взрывающихся градиентов

Если вы уже строили нейронные сети и обучали их, то можете заметить, что в какой-то момент получаются плохие результаты. Модель перестает сходиться, предсказания модели Nan, или резко подскакивает Loss и т. д. Зачастую это происходит из-за того, что на каком-то наборе данных (батче) модель делает слишком странное предсказание и получает огромные градиенты. После чего значительно меняя значения параметров, уходит в другую точку в пространстве параметров, где опять получает большую ошибку и т.д. по нарастающей. Это одна из тех причин, почему нужно нормировать данные. Для линейного слоя градиент напрямую зависит от значения данных на входе.

Ошибку можно обнаружить даже при нормировке данных. Могут быть неправильные значения границ данных, в которые мы отнормировали.

Что делать, если получаются очень большие значения градиента?

Отследить. Если очень большой градиент, то SGD может привести к слишком большому изменению. Этого можно избежать, некоторым образом повлияв на значение градиента.

$$\theta^{new} = \theta^{old} - \overbrace{\alpha}^{\text{learning rate}} \underbrace{\nabla_\theta J(\theta)}_{\text{gradient}}$$

Контролировать норму градиента — то направление, куда нужно двигаться. Чем больше норма вектора градиента, тем к большим изменениям приведет градиент.

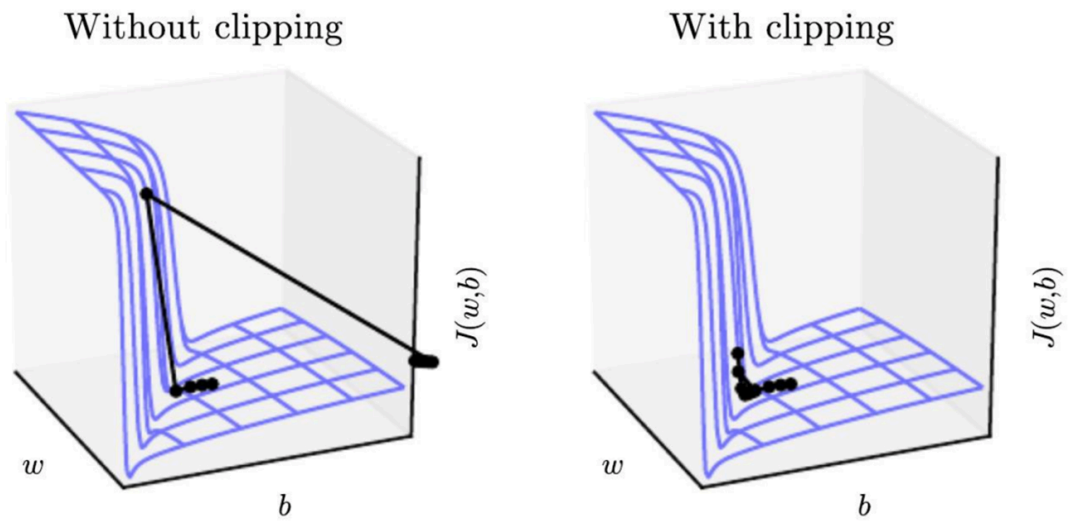
Ограничим длину этого вектора и запретим меняться весам с помощью градиента более чем на какую-то величину — трешхолд. Это называется **градиент клиппинг (обрезка градиента)**.

Algorithm 1 Pseudo-code for norm clipping

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
  $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|}\hat{\mathbf{g}}$
end if

Теперь на каждом шаге мы гарантировано получаем изменения не более чем на градиент заданной нормы. Это простое и лаконичное решение и максимально просто встраивается в любой код. Работает не только в RNN, но и в любых других нейронных сетях.

Приведем иллюстрацию решения проблемы взрывающегося градиента его обрезанием:



RMSprop и Momentum не могут помочь в решении проблемы взрывающегося градиента. Они делают усреднение по всем данным, а градиент взрывается на конкретном батче, на конкретных точках.

Откуда брать трешхолд. Это еще один гиперпараметр, константа, которая зависит от задачи в целом. Поэтому при моделировании нейронных сетей нужно следить:

- за качеством модели (precision и loss на train валидации);
- градиентом ее параметров;
- значением функции потерь;
- дисперсией градиентов, предсказаний;
- нормой вектора градиентов.

7. Языковое моделирование: реализация в Python

Изучим рекуррентные нейронные сети на Python.

Подгрузим все необходимые библиотеки:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Будем учить RNN придумывать новые имена на базе уже имеющихся:

```
# Uncomment this cell in Colab


! wget https://raw.githubusercontent.com/girafe-ai/ml-course/22f_basic/week0_09_embeddings_and_seq2seq/names -O names
```

Посмотрим, что хранится в датасете:

```
import os
start_token = " "

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

```
print ('n samples = ',len(names))
for x in names[::1000]:
```

```
    print (x)
```

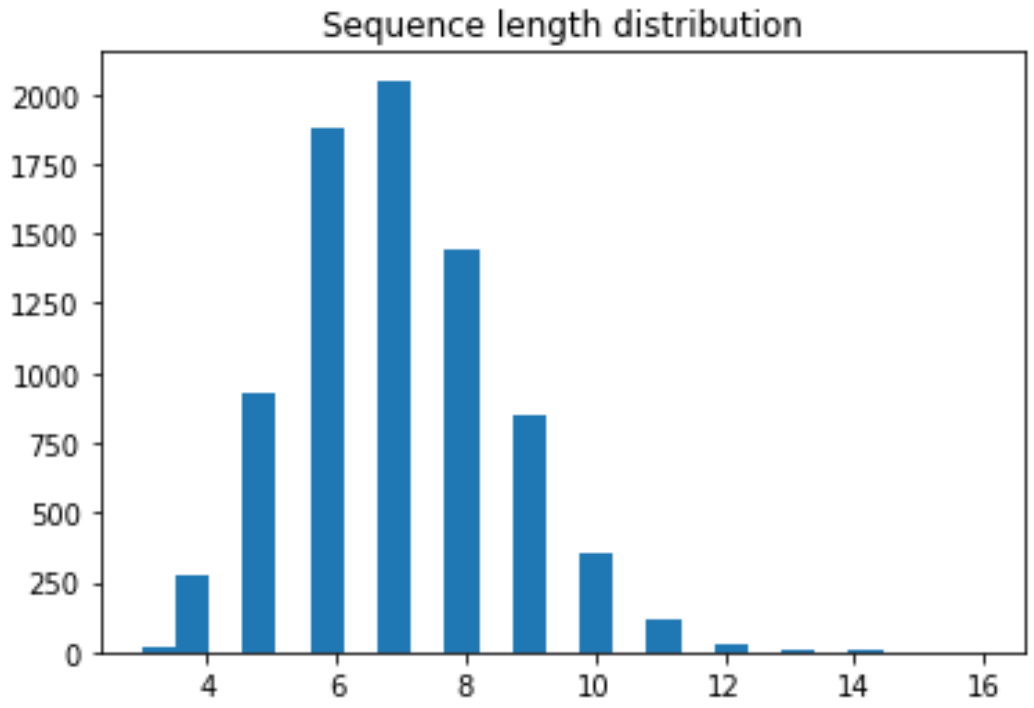
```
>>> n samples =  7944
Abagael
Claresta
Glory
Liliane
Prissie
Geeta
Giovanne
Piggy
```

Как видим, в датасете хранятся имена не очень большой длины. Начинаются с заглавной латинской буквы. Предсказательная модель это должна понимать.

Посмотрим на распределение длин имен:

```
MAX_LENGTH = max(map(len, names))
print("max length =", MAX_LENGTH)


plt.title('Sequence length distribution')
plt.hist(list(map(len, names)),bins=25);
```

```
max length = 16
```



Чаще всего в именах 7 букв.

На вход нейронной сети мы будем подавать токены. Сделаем собственный токенайзер, который будет делить имена на токены и сопоставлять им некоторый индекс:

```
#all unique characters go here
tokens = set()
for name in names:
    tokens.update(set(name))

 #<all unique characters in the dataset>

tokens = list(tokens)
```

```
num_tokens = len(tokens)
print ('num_tokens = ', num_tokens)


assert 50 < num_tokens < 60, "Names should contain within 50 and 60 unique tokens depending on encoding"
```

```
>>> num_tokens =  55
```

Вышло 55 одинаковых токенов.

Set — это набор оригинальных букв:

```
set('Anna')
```

```
>>> {'A', 'a', 'n'}
```

Машине нужно подавать не только несущие смысл токены, но и токены, которые будут указывать на новое имя и на конец имени.

```
tokens.append('<') # <SOS> начало токена
tokens.append('>') # <EOS> конец токена
tokens.append('_') # <PAD> токен падинга
```

В нашу нейронную сеть мы будем подавать имена, которые имеют разную длину. После токена EOS мы должны достраивать имена до одинаковой длины. Это будет делаться с помощью токена PAD.

Построим словарь, который будет отображать буквенному токену его индекс:

```
token_to_id = {token: idx for idx, token in enumerate(tokens)} # <dictionary of symbol -> its identifier (index in tokens list)>
```

Посмотрим на индексы вспомогательных токенов:

```
token_to_id[“>”], token_to_id[“_”], token_to_id[“<”]
```

>>> (56, 57, 55)

```
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 list"

print("Seems alright!")
```

Попробуем создавать батчи:

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

   max_len = max_len or max(map(len, names))
   max_len += 1
   names_ix = np.zeros([len(names), max_len], dtype) + pad #достраиваем паддингом
   names_ix[:, 0] = token_to_id['<'] # <SOS>

   for i in range(len(names)):
      line_ix = [token_to_id[c] for c in names[i]]
```

```
    names_ix[i, 1:len(line_ix)] = line_ix[1:]
    names_ix[i, len(line_ix)] = token_to_id['>'] # <EOS>

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

  return names_ix
```

Рассмотрим пример, как выглядят наши токены:

```
#Example: cast 4 random names to matrices, pad with zeros
print('\n'.join(names[::2000]))
print(to_matrix(names[::2000]))
```

```
>>> Abagael
 Glory
 Prissie
 Giovanne
[[55 30 23 37 51 37 39 10 56 57]
 [55 14 10  3 31  9 56 57 57 57]
 [55  1 31 50 53 53 50 39 56 57]
 [55 14 50  3 27 37 40 40 39 56]]
```

Теперь посмотрим, что мы можем из индексов токенов воспроизвести обратно в имя:

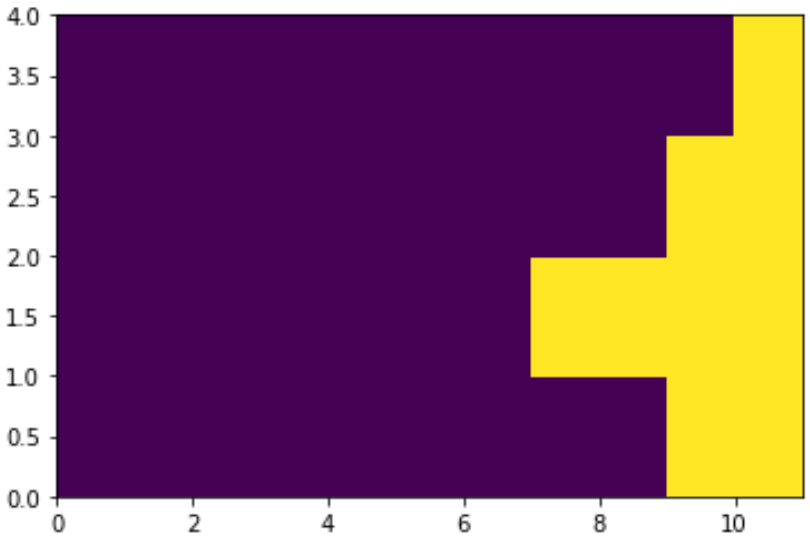
```
''.join([tokens[idx] for idx in to_matrix(names[::2000])[3]])
```

>>>'<Giovanne>'

На графике для каждого из имен покажем, какие элементы были «западены» (от англ. padding):

```
plt.pcolormesh(to_matrix(names[::2000]) == 57)
```

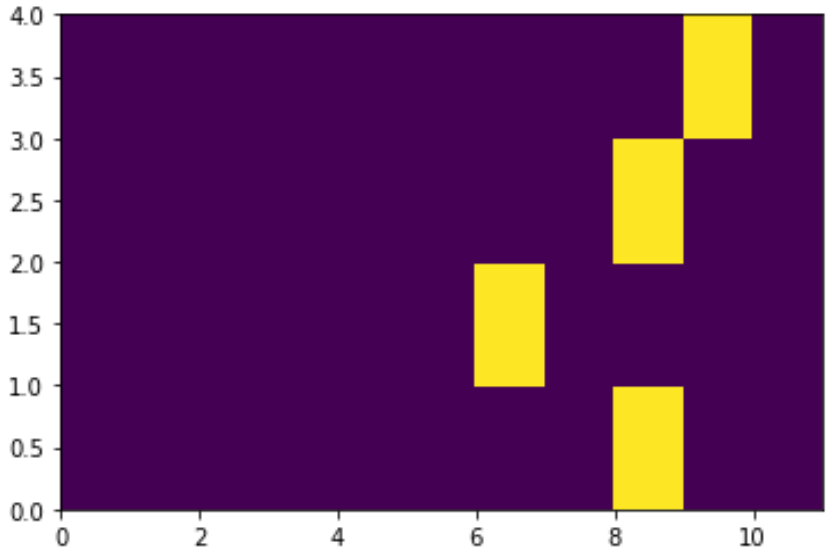
```
<matplotlib.collections.QuadMesh at 0x7fa7212eecd0>
```



И посмотрим, как распределяется токен EOS:

```
plt.pcolormesh(to_matrix(names[::2000]) == 56)
```

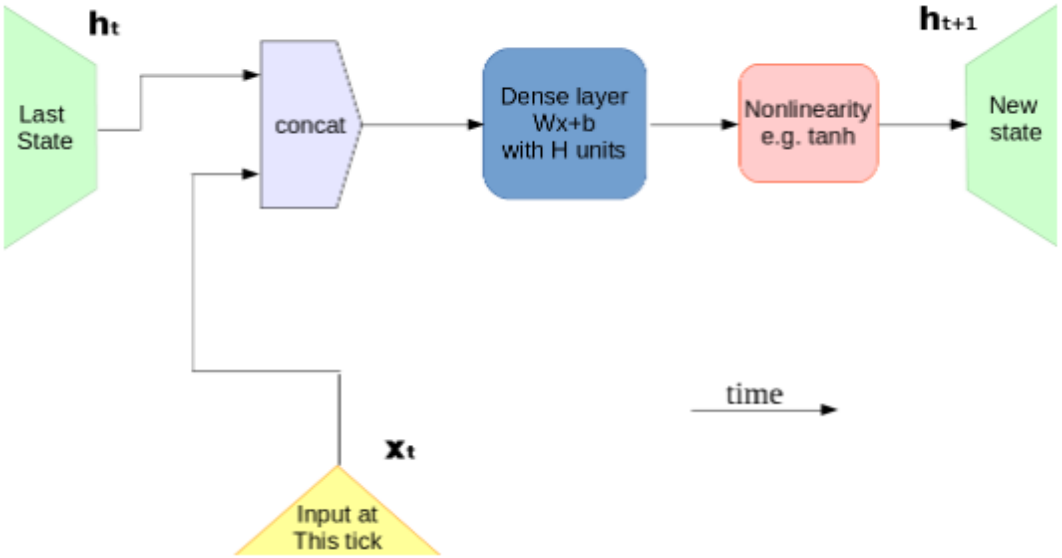
```
<matplotlib.collections.QuadMesh at 0x7fa721374590>
```

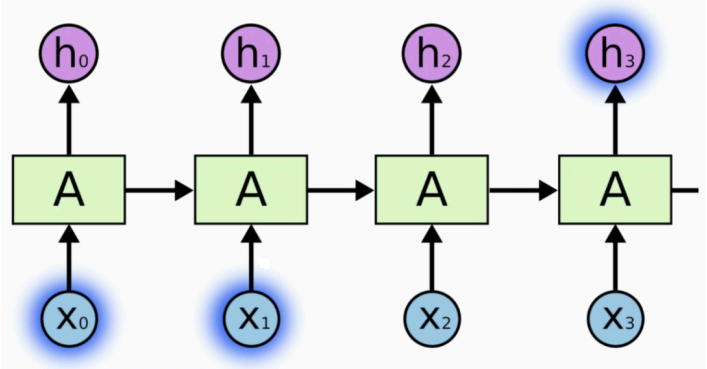


Повторим, что такое рекуррентные нейронные сети.

Архитектура Seq2Seq (Sequence to Sequence) — это сети, которые принимают на вход не всю последовательность сразу, а один элемент последовательности в один момент времени.

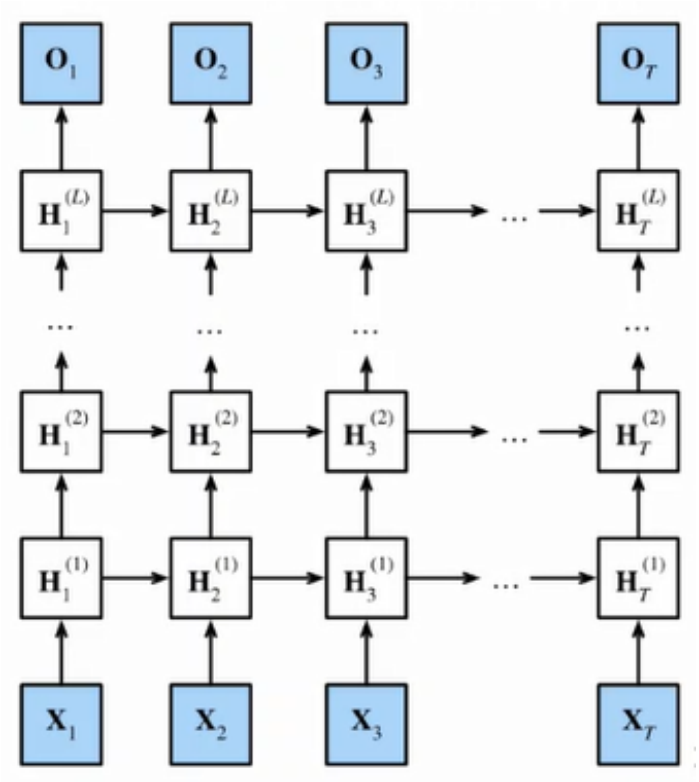
На рисунке представлена RNN-клетка, развернутая во времени:





Подаем на вход информацию x_0 в момент времени t_0, получаем какое-то предсказание h_0. В клетке генерируется скрытое состояние. На вход той же самой клетки в момент времени t_1 будем подавать x_1 и скрытое состояние с предыдущего состояния RNN-клетки. Получаем предсказание токена h_1 и подаем на вход следующего состояния новое скрытое состояние.

Для нескольких RNN-клеток это выглядит следующим образом:



Происходит стек состояний.

В PyTorch для ванильной RNN используется следующая формула:

$$h_t = \tanh(x_t W_{ih}^T + b_{ih} + h_{t-1} W_{hh}^T + b_{hh})$$

Попробуем написать RNN в коде:

```
import torch, torch.nn as nn
import torch.nn.functional as F
```

Векторное представление имени:

```
embed = nn.Embedding(4, 2)
```

Это обучающийся слой эмбединга:

```
next(embed.parameters())
```

>>> Parameter containing:

tensor([[1.6263, 0.8233],

[0.7266, 0.0388],

[0.6093, -0.1930],

[-0.3640, -1.6748]], requires_grad=True)

```
embed(torch.LongTensor([2]))
```

>>>tensor([[0.6093, -0.1930]], grad_fn=<EmbeddingBackward0>)

Создадим RNN-клетку следующим образом:

```
class CharRNNCell(nn.Module):
    """
    Implement the scheme above as torch module
    """
    def __init__(self, num_tokens=len(tokens), embedding_size=18, 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)
#линейный слой
        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, containing vector of int64
        :param h_prev: previous rnn hidden states, 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
```

```
        # hint: use torch.cat(..., dim=...) for concatenation
        x_and_h = torch.cat([x_emb, h_prev], dim=-1) #YOUR CODE HERE
        h_next = self.rnn_update(x_and_h)

        h_next = torch.tanh(h_next) #функция активации

        assert h_next.size() == h_prev.size()

        #compute logits for next character probs
        logits = self.rnn_to_logits(h_next) #еще один линейный слой

        return h_next, logits

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

Мы получили одну RNN-клетку, а нам нужно несколько таких клеток.

У нас пока один шаг, и на выходе получаем вектор скрытого состояния.

Создадим нашу модель. Раскрутим RNN-клетку во времени с помощью циклов:

```
def rnn_loop(char_rnn, batch_ix):
    """
    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_ix.size()
    hid_state = char_rnn.initial_state(batch_size)
    logits = []

    for x_t in batch_ix.transpose(0,1):
        hid_state, logits_next = char_rnn(x_t, hid_state)  # <-- here we call your one-step code
        logits.append(logits_next)

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

```
num_tokens = len(tokens)
```

```
batch_ix = to_matrix(names[:5])
batch_ix = torch.tensor(batch_ix, dtype=torch.int64)

logit_seq = rnn_loop(char_rnn, batch_ix)

# assert torch.max(logp_seq).data.numpy() <= 0
# assert tuple(logp_seq.size()) ==  batch_ix.shape + (num_tokens,)
```

В батче должны быть токены, которые берутся в определенный момент времени.

Попробуем обучить нашу нейросеть. Возьмем предсказанные значения и правильные значения:

```
predictions_logits = logit_seq[:, :-1]
actual_next_tokens = batch_ix[:, 1:]

# logp_next = torch.gather(
#     predictions_logp,
#     dim=2,
#     index=actual_next_tokens[:,:,None]
#     )

# loss = -logp_next.mean()
```

Предсказания двигаются на один токен вперед. Этот сдвиг мы учитываем:

```
loss_func = nn.CrossEntropyLoss(ignore_index=token_to_id['_'])
loss2 = loss_func(
    predictions_logits.reshape((-1, num_tokens)),
    actual_next_tokens.reshape(-1)
    )
```

```
loss2.backward()
```

```
for w in char_rnn.parameters():
    assert w.grad is not None and torch.max(torch.abs(w.grad)).data.numpy() != 0, \
```

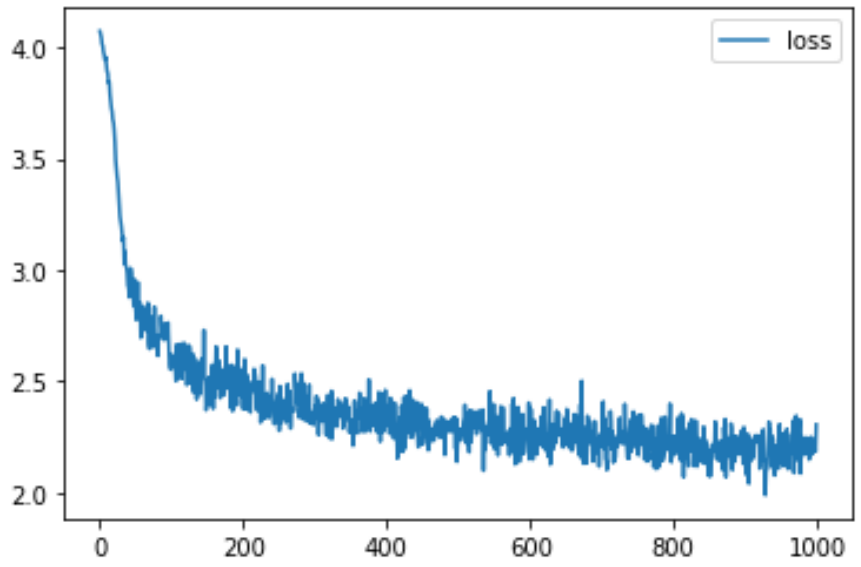
```
        "Loss is not differentiable w.r.t. a weight with shape %s. Check forward method." %
(w.size(),)
```

Теперь запустим обучающий цикл:

```
from IPython.display import clear_output
from random import sample

char_rnn = CharRNNCell()
opt = torch.optim.Adam(char_rnn.parameters())
loss_func = nn.CrossEntropyLoss(ignore_index=token_to_id['_'])

history = []
```

```
MAX_LENGTH = 16

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

    logits_seq = rnn_loop(char_rnn, batch_ix)

    # compute loss
    #<YOUR CODE>
    predictions_logits = logits_seq[:, :-1] #это и есть наши предсказания
```

```
    actual_next_tokens = batch_ix[:, 1:]

    loss = loss_func(
    predictions_logits.reshape((-1, num_tokens)),
    actual_next_tokens.reshape(-1)
    )
    loss.backward()
    opt.step()

    opt.zero_grad() #зануляем градиенты

    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."
```



Видим, что нейросеть обучается, функция ошибки падает.

Теперь перейдем к самому интересному — к восстановлению предсказания. Для этого есть функция generate_sample:

```
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, logits_next = char_rnn(x_sequence[:, -1], hid_state)
        p_next = F.softmax(logits_next / temperature, dim=-1).data.numpy()[0] #вероятность взять токен

        # sample next token and push it back into x_sequence
        next_ix = np.random.choice(num_tokens, p=p_next)
        next_ix = torch.tensor([[next_ix]], dtype=torch.int64)
        x_sequence = torch.cat([x_sequence, next_ix], dim=1)
        if next_ix == token_to_id['>']:
          break
    return ''.join([tokens[ix] for ix in x_sequence.data.numpy()[0]])
```

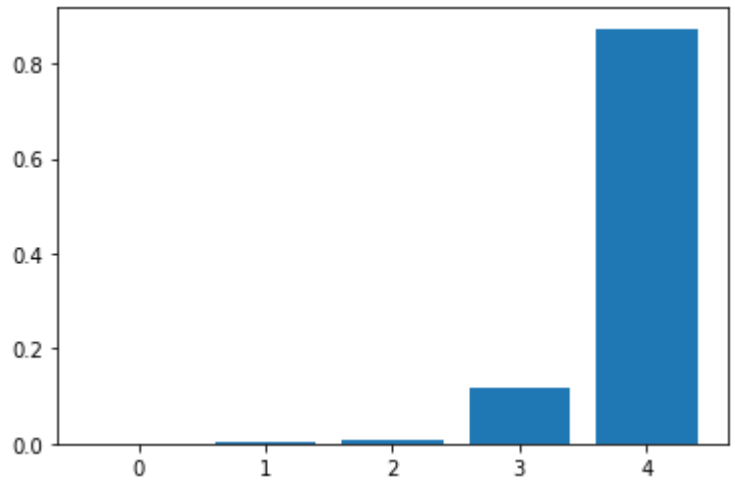
seed_phrase — то, с чего должно начинаться следующее имя. В начале токен начала предложения.

temperature позволяет внести элемент случайности в выбор токенов. Если ее не будет, каждый раз будет браться токен с наибольшей вероятностью.

```
def softmax(x, temp=1.):
  exps = np.exp((x - x.max()) / temp)
  return exps / exps.sum()
```

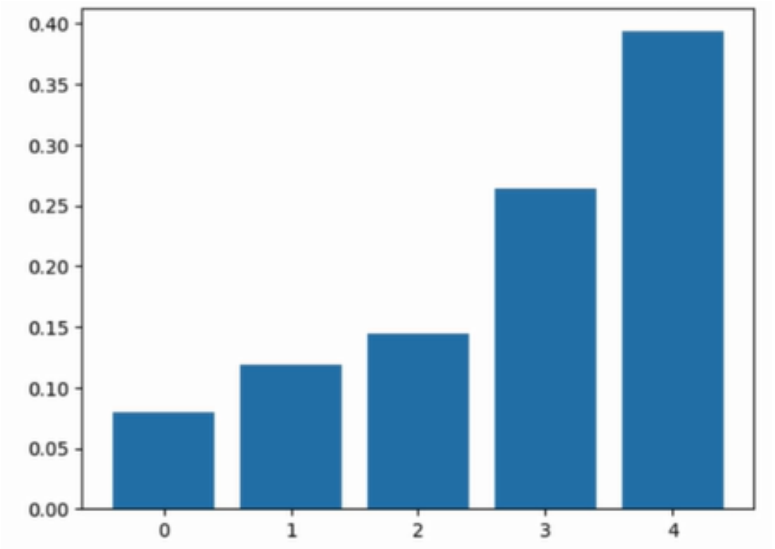
При temperature = 1:

```
plt.bar(np.arange(5), softmax(np.array([1, 3 , 4, 7, 9]), 1))
```

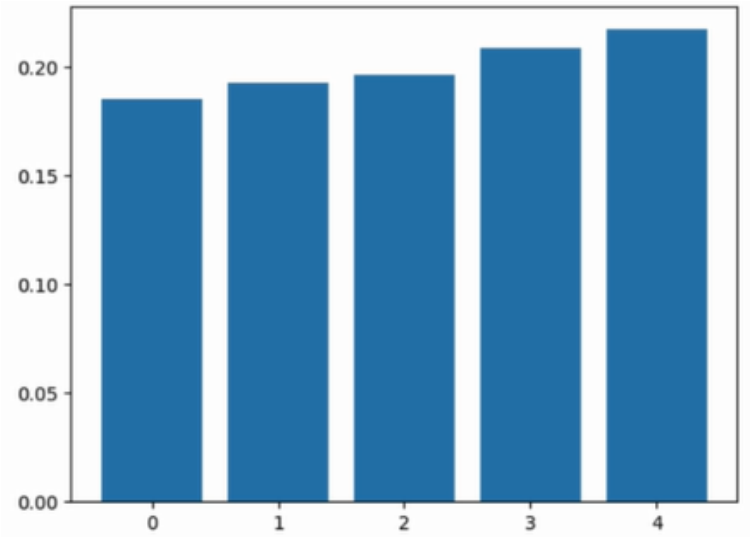


При temperature = 5:

```
plt.bar(np.arange(5), softmax(np.array([1, 3 , 4, 7, 9]), 5))
```



```
plt.bar(np.arange(5), softmax(np.array([1, 3 , 4, 7, 9]), 50))
```



При увеличении вероятности токены попадают равновероятно:

```
for _ in range(10):
    print(generate_sample(char_rnn, temperature=0.1)[1:-1])
```

>>>Berrie
Alenna
Annelle
Allin
Alenna
Aline
Sherie
Alenne
Alenna
Sharie

При низкой temperature очень часто генерируется Alenna и Sherie.

Попробуем сгенерировать с какой-то последовательности:

```
for _ in range(50):
    print(generate_sample(char_rnn, seed_phrase='<Rad'))
```

```
>>><Rada-eranna>
<Radey>
<Radise>
<Radde>
<Rad>
<Rady>
<Rady>
<Radeic>
<Radaphr>
<Rady>
<Radbda>
...
```

Разумеется, каждый раз писать цикл с нуля не хочется. В PyTorch уже есть RNN:

```
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.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 = []
```

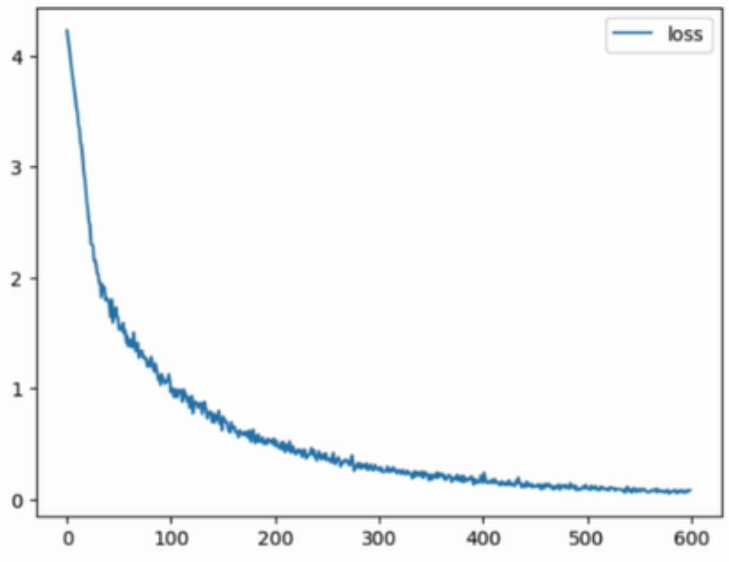
Это более компактная запись того, что мы уже делали:

```
# the model applies over the whole sequence
batch_ix = to_matrix(sample(names, 32), max_len=MAX_LENGTH)
batch_ix = torch.LongTensor(batch_ix)

logp_seq = model(batch_ix)

# compute loss.
loss = F.nll_loss(logp_seq[:, 1:].contiguous().view(-1, num_tokens),
                  batch_ix[:, :-1].contiguous().view(-1))

loss.backward()
```

```
MAX_LENGTH = 16

 for i in range(1000):
    batch_ix = to_matrix(sample(names, 32), max_len=MAX_LENGTH)
```

```
    batch_ix = torch.tensor(batch_ix, dtype=torch.int64)


    logp_seq = model(batch_ix)
    loss = F.nll_loss(logp_seq[:, 1:].contiguous().view(-1, num_tokens), batch_ix[:,
:-1].coniguous().view(-1))
    loss.backward()
    opt.step()
    opt.zero_grad()


#    # compute loss
#    #<YOUR CODE>
#    predictions_logp = #<YOUR CODE>
#    actual_next_tokens = #<YOUR CODE>


#    loss = ###YOUR CODE


#    # train with backprop


#    #<YOUR CODE>


    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."
```



Видим, что RNN обучается.

Домашнее задание. Самостоятельно потестировать RNN.

Дополнительные материалы для самостоятельного изучения

1. The Unreasonable Effectiveness of Recurrent Neural Networks
2. Understanding LSTM Networks
3. Machine Translation, Sequence-to-Sequence and Attention
4. On the difficulty of training recurrent neural networks
5. Deep Residual Learning for Image Recognition
6. Список имен