Рекуррентные нейронные сети. Работа с текстами и временными рядами.

Санкт-Петербургский государственный университет
Кафедра статистического моделирования
Семинар «Статистическое и машинное обучение»

Санкт-Петербург, 2025

Рекуррентные нейронные сети

Рекуррентные нейронные сети целенаправлено создавались для работы с последовательностями векторов или иначе **токенов**. В качестве последовательности токенов могут быть представлены, например, такие данные:

- Текст
- Временной ряд
- Звуковая дорожка
- Видео
- Последовательность аминокислот в белке

Особенно популярным стало в своё время применение рекуррентных сетей к задачам **NLP (обработки естественного языка)**, включающего работу с текстом и звуком.

Режимы работы RNN

Есть несколько способов работать с последовательными данными:

- **Many-to-One**. На вход подается последовательность объектов, на выходе один объект. Пример: тематическая классификация текстов.
- **One-to-Many**. На вход подается один объект, на выходе последовательность объектов. Пример: генерация подписи к изображению.
- **Many-to-Many**. На входе и выходе последовательности, в общем случае различной длины. Пример: перевод с одного языка на другой.

Отдельно выделяют в силу особенно простой его реализации **синхронизированный вариант Many-to-Many**, когда длины входной и выходной последовательности равны.

Эмбеддинги

Поскольку самой распостранённой задачей для RNN стала работа с текстами, а всякая нейронная сеть требует тензор на вход, то нужно векторизовать словарь токенов. Известные методы вроде TF-IDF и модификаций, как выяснилось, не являются самыми лучшими способами решить эту задачу.

Эмбеддинг — это представление токена в виде вектора так, что в вектор становится заключено семантическое значение токена. Обычно семантический смысл основан на контексте, в котором токен встречается.

Первая реализация алгоритма для получения эмбеддингов — **Word2Vec**.

Word2Vec

Word2Vec — это нейросеть без нелинейностей с одним скрытым слоем. Она имеет два варианта архитектуры:

- **Skip-Gram**. Предсказывает контекст по целевому слову.
- **CBOW**. Предсказывает слово по окружающему контексту.

Вход и выход являются векторами с размерностью, равной размеру словая V, а скрытый слой имеет размерность будущего эмбеддинга d. Одна из матриц весов или усреднение обоих матриц и является матрицей эмбеддингов размерности $V \times d$.

Первоначально для токенов из словаря используется one-hot кодирование.

Устройство простейшей RNN

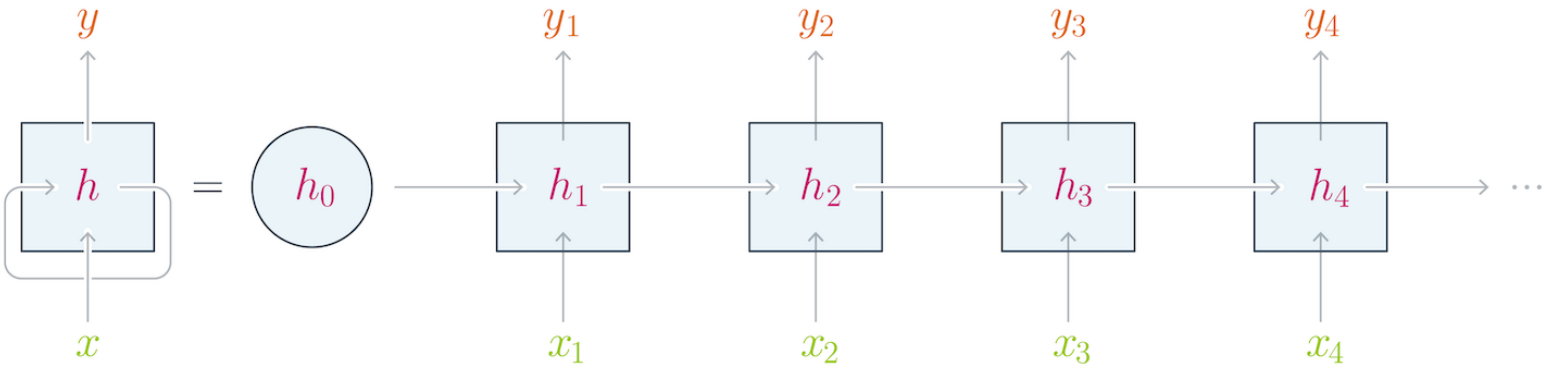

Рис.: Схема рекуррентной нейронной сети.

Чтобы хранить информацию о предыдущих токенах, вводится понятие внутренней памяти или скрытого состояния h_t. На каждом шаге в сеть подаётся эмбеддинг нового токена и происходит обновление состояния:

$$h_t = f(h_{t-1}W_h + x_tW_x + b_h) \tag{1}$$

RNN в различных режимах

По скрытому состоянию произвольным образом можно предсказывать выходной сигнал. В простейшем случае предсказние имеет следующий вид:

$$y_t = g(h_t W_y + b_y) \tag{2}$$

Если мы предсказываем выход только для последнего состояния, то получаем сеть в режиме Many-to-One. В общем же случае имеет место синхронизированный Many-to-Many.

Обратное распространение ошибки в RNN

Поскольку одни и те же параметры применяются несколько раз, причём к выходам, этими же параметрами и порождённым, то возникает существенная проблема: как считать градиент по параметрам? Идея **backpropagation through time (BPTT)** состоит в том, что мы сначала вычисляем производные по выходам, а затем распространяем градиенты назад по времени.

Одна из причин, по которой рекуррентные сети уступили трансформерам, состоит как раз в том, что BPTT значительно хуже распараллеливается.

BPTT, градиент по выходным весам

Итак, у нас есть некоторая функция потерь, равная сумме потерь для выхода на каждом шаге:

$$L = \sum_{t=1}^{T} L_t(\hat{y}_t, y_t) \tag{3}$$

Можем найти градиент по выходным весам:

$$\frac{\partial L}{\partial W_y} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial W_y} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial \hat{y}_t} h_t \tag{4}$$

BPTT, градиент по скрытым состояниям

Градиент по скрытым состояниям h_t имеет вид:

$$\frac{\partial L}{\partial h_t} = \frac{\partial L_t}{\partial h_t} + \frac{\partial L}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_t} \tag{5}$$

В свою очередь, имеет место следующее:

$$h_{t+1} = f(W_h h_t + W_x x_{t+1} + b) \tag{6}$$

Тогда

$$\frac{\partial h_{t+1}}{\partial h_t} = W_h^T \operatorname{diag}\left(f'(W_h h_t + W_x x_{t+1} + b)\right) \tag{7}$$

Тем самым получаем реккурентную формулу. На последнем шаге $\frac{\partial L}{\partial h_{T+1}} = 0$.

BPTT, градиент по рекуррентным и входным весам

Введём обозначение:

$$\delta_t = \left(\frac{\partial L}{\partial h_t}\right) \odot f'(W_h h_t + W_x x_{t+1} + b) \tag{8}$$

Тогда, градиент по рекуррентным весам:

$$\frac{\partial L}{\partial W_h} = \sum_{t=1}^{T} \frac{\partial L}{\partial h_t} \frac{\partial h_t}{\partial W_h} = \sum_{t=1}^{T} \delta_t \, h_{t-1}^T \tag{9}$$

Градиент по входным весам:

$$\frac{\partial L}{\partial W_x} = \sum_{t=1}^{T} \delta_t \, x_t^T \tag{10}$$

Взрыв и затухание градиента

Поскольку BPTT предполагет, что если градиент по h_t передаётся через k шагов, то происходит k умножений на W_h. Тогда имеем одну из двух ситуаций:

- Если среди собственных чисел W_h есть превосходящие по модулю единицу, то градиент растёт, быстро достигая переполнения и делая сеть не пригодной к использованию.
- В противном случае градиент быстро убывает, а веса почти не обновляются, обучения не происходит.

Проблема взрывающегося градиента в классической RNN обычно решается посредством **градиентного клиппинга**. Суть его в том, чтобы уменьшить норму градиента, если она превосходит некоторое наперёд заданное число, не меняя при этом направления градиента. Число-ограничитель является гиперпараметром.

Глубокие RNN

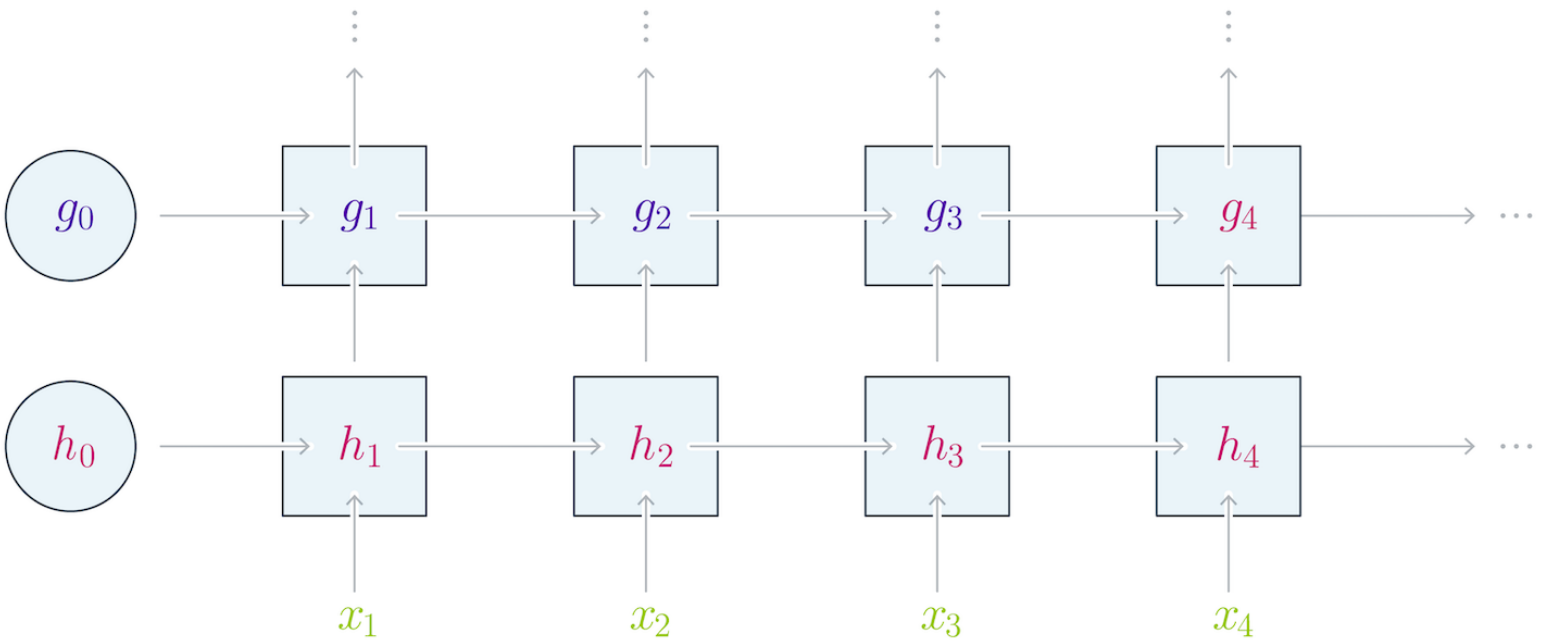

Рис.: Схема глубокой рекуррентной нейронной сети.

Глубокая RNN предполагает, что первый слой по-прежнему принимает на вход исходную последовательность, а второй — выходы первого, и так далее.

Bidirectional RNN

Иногда имеет смысл учитывать не только предыдущий контекст, но и последующий. Например, мы хотим определить синтаксическую роль каждого слова в предложении, что может зависеть и от предшествующих, и от последующих слов. В таких случаях используются **двунаправленные рекуррентные нейронные сети (bidirectional RNN)**.

При этом, можно сказать, в одно целое объединяются две сети, одна из которых принимает токены в обратном порядке. Отсюда очевидно, что такая сеть всё ещё не может решать задачи в несинхронизированном режиме Many-to-Many.

Bidirectional RNN

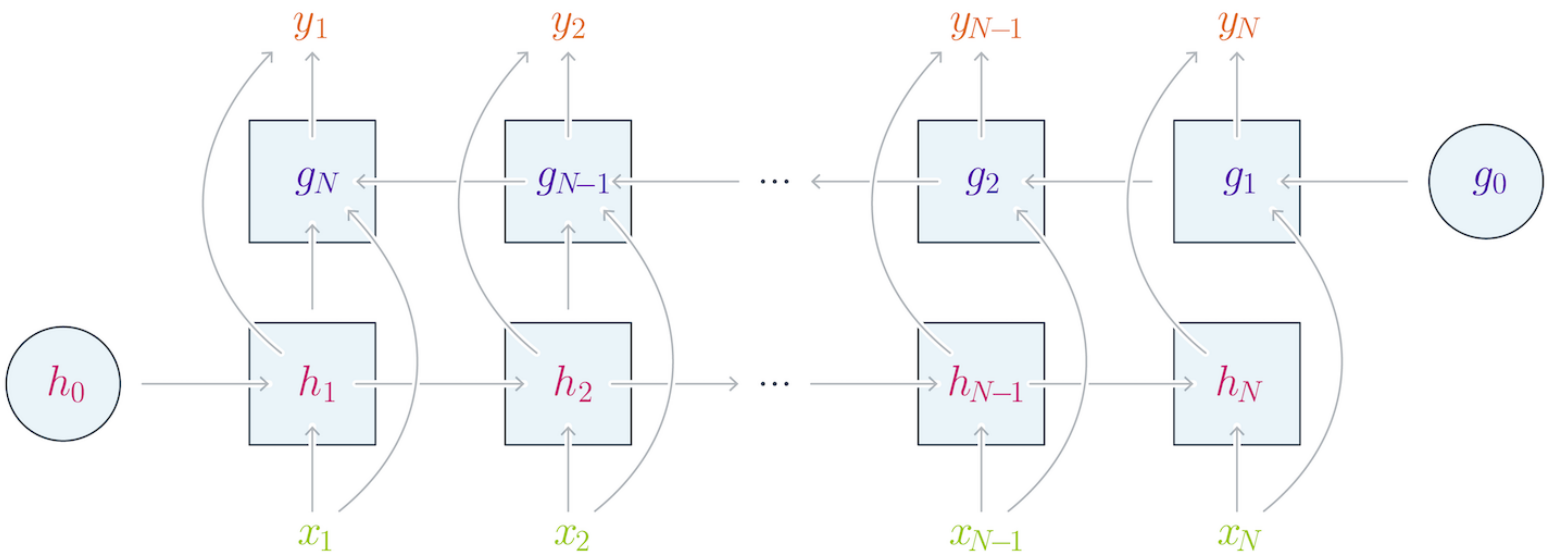

Рис.: Схема двунаправленной рекуррентной нейронной сети.

Отдельные элементы, попадая на вход какого-либо блока, могут суммироваться, усредняться или просто подвергаются конкатенации. Как правило, более сложные зависимости на практике не используются.

LSTM — долгая краткосрочная память

Поскольку взрыв и затухание градиента не позволяют обучать классические RNN на больших последовательностях, была предложена модель **долгой краткосрочной памяти (LSTM)**.
LSTM-блоки содержат **вентили (gates)**, которые используются для контроля потоков информации на входах и выходах памяти данных блоков. Эти вентили реализованы в виде логистической функции для вычисления значения в диапазоне $[0, 1]$.
Умножение на это значение используется для частичного допуска или запрещения потока информации внутрь и наружу памяти.

LSTM-блоки

Forget/input gate — вентили забывания и записи:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f), \;\; i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{11}$$

Candidate — кандидат на запись:

$$\tilde{c}_t = \tanh(W_c[h_{t-1}, x_t] + b_c) \tag{12}$$

Cell state — обновление памяти:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{13}$$

Output gate — вентиль выхода:

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{14}$$

Hidden state — новое скрытое состояние

$$h_t = o_t \odot \tanh(c_t) \tag{15}$$

LSTM-блоки

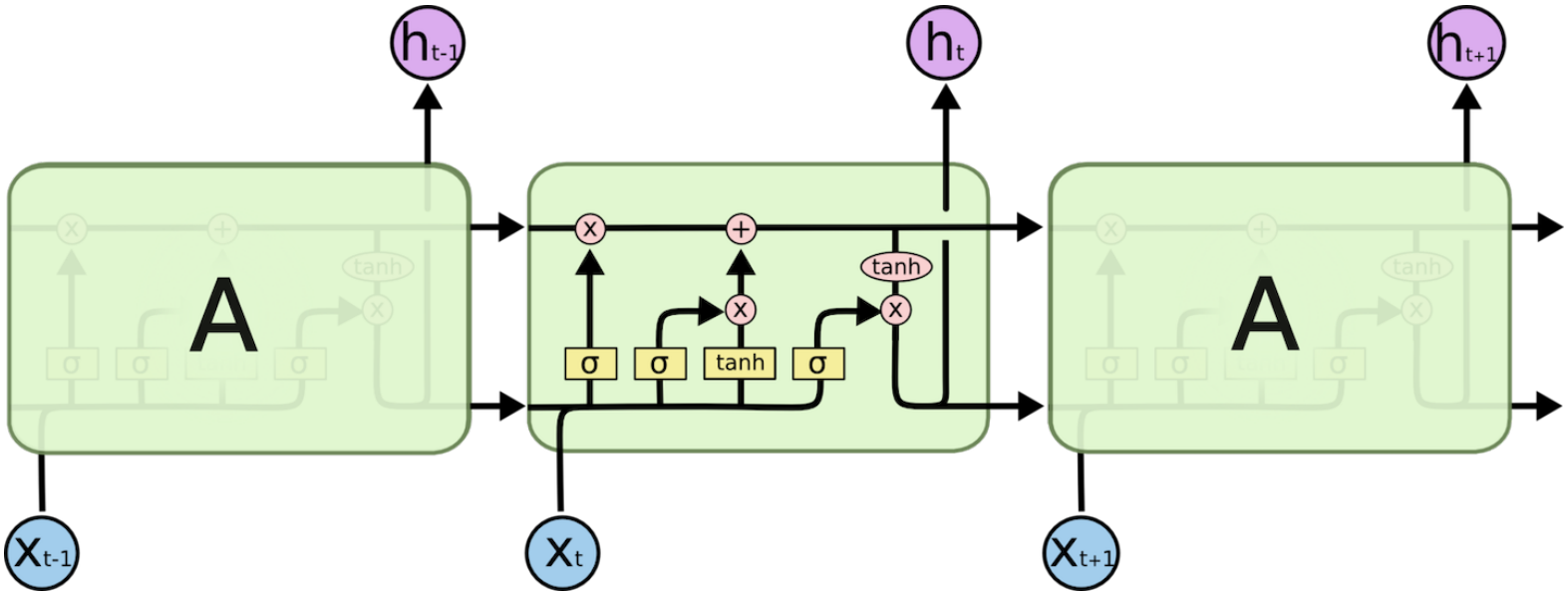

Рис.: Схема LSTM-блока.

LSTM-блоки являются довольно трудоёмкими с вычилительной точки зрения, причём как на обучении, так и на предсказании.

LSTM с «глазками»

Одна из модфиикаций, известная как **LSTM с «глазками»**, позволяет вентилям забывания, входа и выхода видеть содержимого ячейки памяти c_{t-1}:

Forget gate:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + V_f \odot c_{t-1} + b_f) \tag{16}$$

Input gate:

$$i_t = \sigma(W_i[h_{t-1}, x_t] + V_i \odot c_{t-1} + b_i) \tag{17}$$

Output gate:

$$o_t = \sigma(W_o[h_{t-1}, x_t] + V_o \odot c_t + b_o) \tag{18}$$

ConvLST

ConvLSTM — это разновидность LSTM, в которой все матричные умножения заменены на сверточные операции. Она была предложена для решения задач, где данные имеют пространственную структуру (как кадры видео).

Например, так выглядит Forget gate в ConvLSTM:

$$f_t = \sigma\big(W_f * x_t + U_f * h_{t-1} + b_f\big) \tag{19}$$

Остальные вентели изменяются аналогично.

GRU — управляемый рекуррентный блок

Управляемый рекуррентный блок (GRU) проще, чем LSTM, и использует два типа вентилей: обновляющие (update gate) и сбрасывающие (reset gate). Они управляют тем, какую информацию из прошлого состояния стоит сохранить, а какую — забыть.

Интуитивно, если выход update gate близок к единице, то новое состояние будет в основном взято из скрытого состояния в момент времени t, если ближе к нулю — в момент времени $t-1$. Вентиль сброса решает, какую часть прошлой информации «забыть» перед вычислением кандидата на новое состояние.

GRU-блоки

Update gate:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \tag{20}$$

Reset gate:

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \tag{21}$$

Кандидат на новое скрытое состояние:

$$\tilde{h}_t = \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \tag{22}$$

Итоговое скрытое сотояние:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{23}$$

Текст

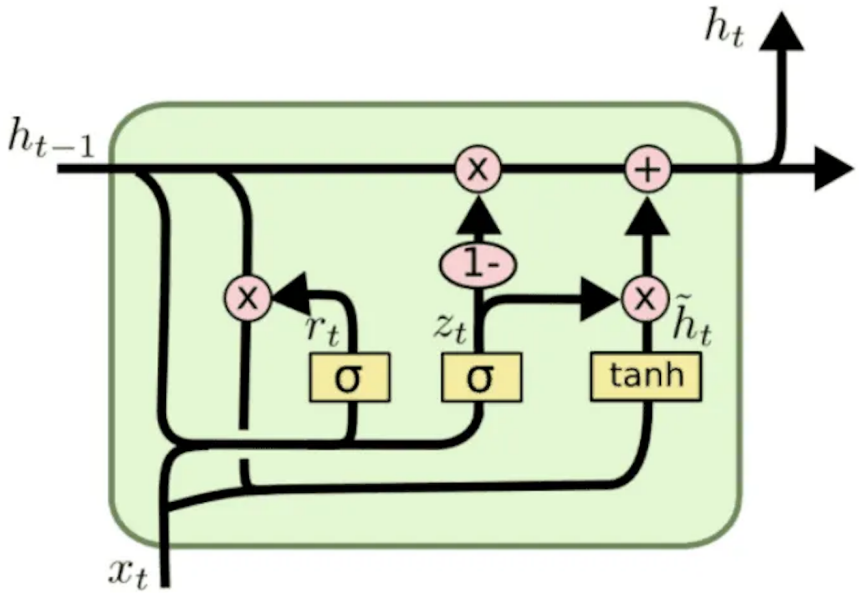

Рис.: Схема GRU-блока.

В итоге GRU имеет меньше параметров, чем LSTM, и при прочих равных, быстрее обучается. GRU и LSTM показывают сопоставимое качество на многих задачах, включая генерацию музыки, распознавание речи, многие задачи обработки естественного языка.

Несинхронизованный вариант Many-to-Many

Несинхронизованный вариант Many-to-Many, иначе называемый задачей **Seq2Seq** (поскольку мы из одной последовательности произвольной длины получаем другую), требует некоторых модификаций рассмторенных инструментов. Для задач типа Seq2Seq применяется архитектура Encoder-Decoder.

Энкодер читает входное предложение токен за токеном и обрабатывает их с помощью блоков рекуррентной сети. Выход последнего блока становится контекстным вектором.

Архитектура декодера аналогична энкодеру. При этом каждый блок декодера должен учитывать токены, сгенерированные к текущему моменту, и также информацию об исходной последовательности.

Encoder-Decoder

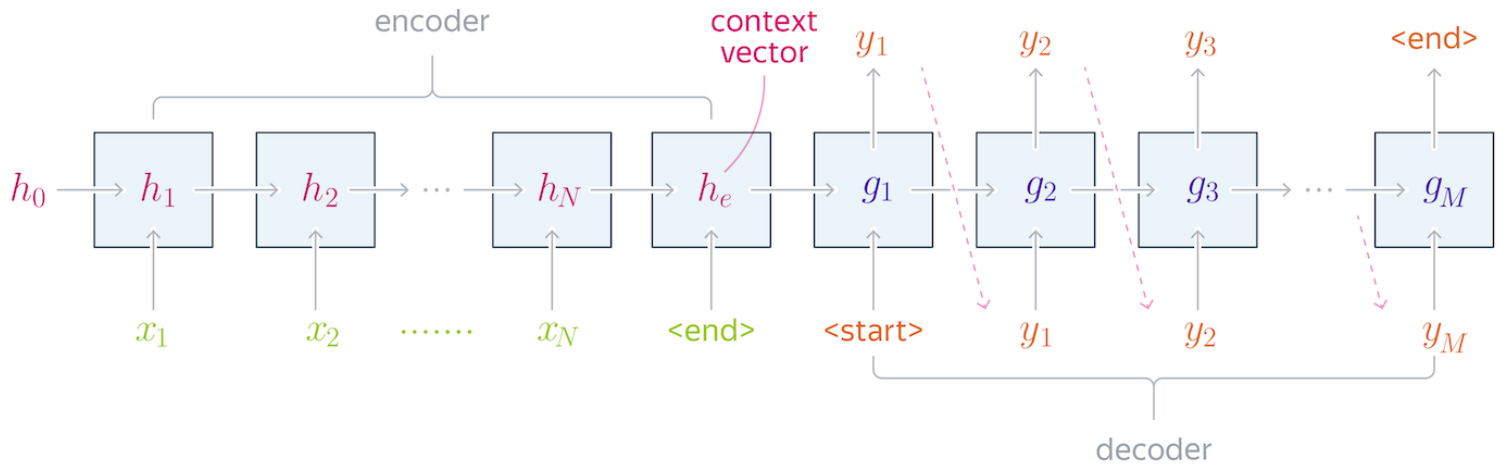

Рис.: Схема архитектуры Encoder-Decoder.

Согласно предложенной схеме, есть максимальная длина последовательности, которую может сгенерировать декодер. Генерация продолжается, пока не будет сгенерирован спецтокен остановки или не будет достигнута максимальная длина. Очевидно, если взять декодер без энкодера, то мы получим решение задачи One-to-Many.