# Нейросети для работы с последовательностями

Ph@DS

# 1. Постановка задачи

Пусть имеется некоторая последовательность векторов $x_1, x_2, \ldots, x_t$.

**Задача**: По входной посл-ти получить некоторый вектор, содержащий информацию о подаваемой последовательности.

*Вход*: $x_1, x_2, \ldots, x_t$, где $x_i$ — вектор размера $d'$.

*Выход*: вектор $e$ размера $d$.

Вектор $e$ кодирует информацию об $x_1, x_2, \ldots, x_t$.

$d$ —гиперпараметр.

# Conv1d

# 2. 1D-Convolution layer

На вход приходит некоторая посл-ть из векторов $\{x_1, x_2, \ldots, x_t\}$.

Каждый объект представляется вектором его признаков.
Запишем объекты в матрицу $X$ размера $(t, d')$,

где $X_i$ — $i$-ая строка матрицы $X$.
В отличие от картинок в матрице $X$ соседние по горизонтали числа

не упорядочены, однако соседние по вертикали — упорядочены.

Например:

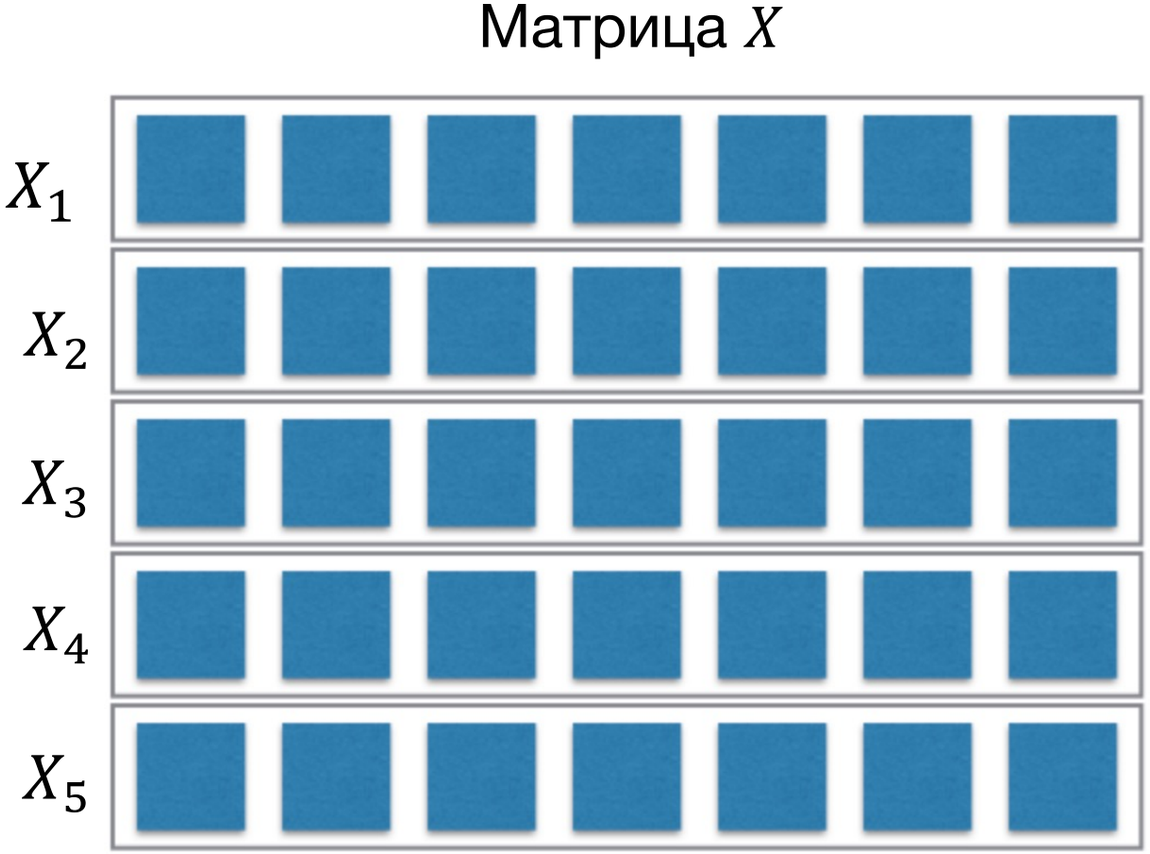
- У временных рядов есть упорядоченность по времени,

но нет упорядоченности внутри признаков одного элемента ряда.

Поэтому для $X$ не совсем правильно делать 2D-Conv.
Для таких $X$ применяется 1D свёртка.

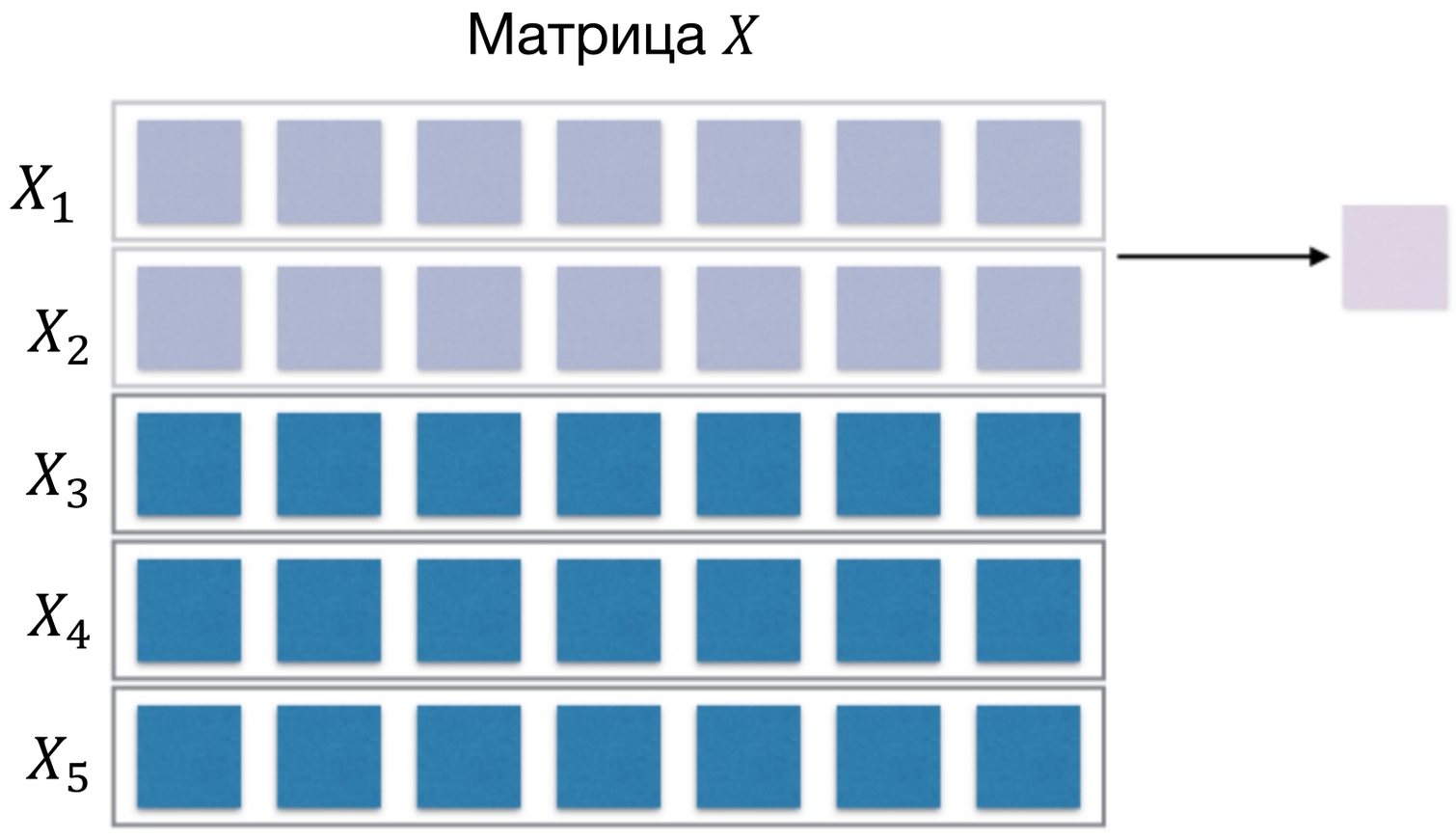
# 2. 1D-Convolution layer

1D-Conv очень похож на 2D-Conv,
но фильтр двигается теперь только по одной размерности.




# 2. 1D-Convolution layer

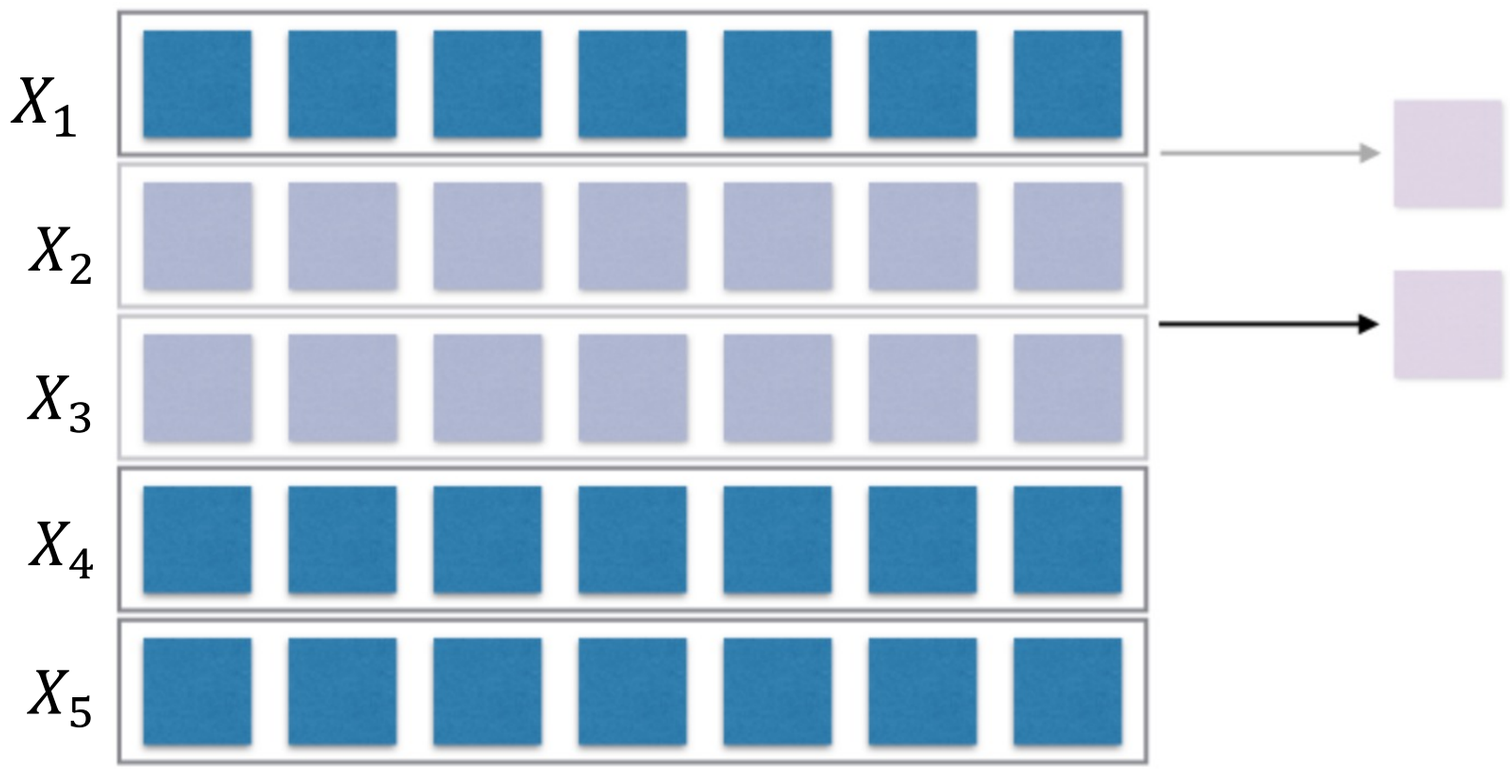
1D-Conv очень похож на 2D-Conv,
но фильтр двигается теперь только по одной размерности.




Фильтр применяется к $[X_1, X_2]$ ,
считается скалярное произведение фильтра и подматрицы $X$.

# 2. 1D-Convolution layer

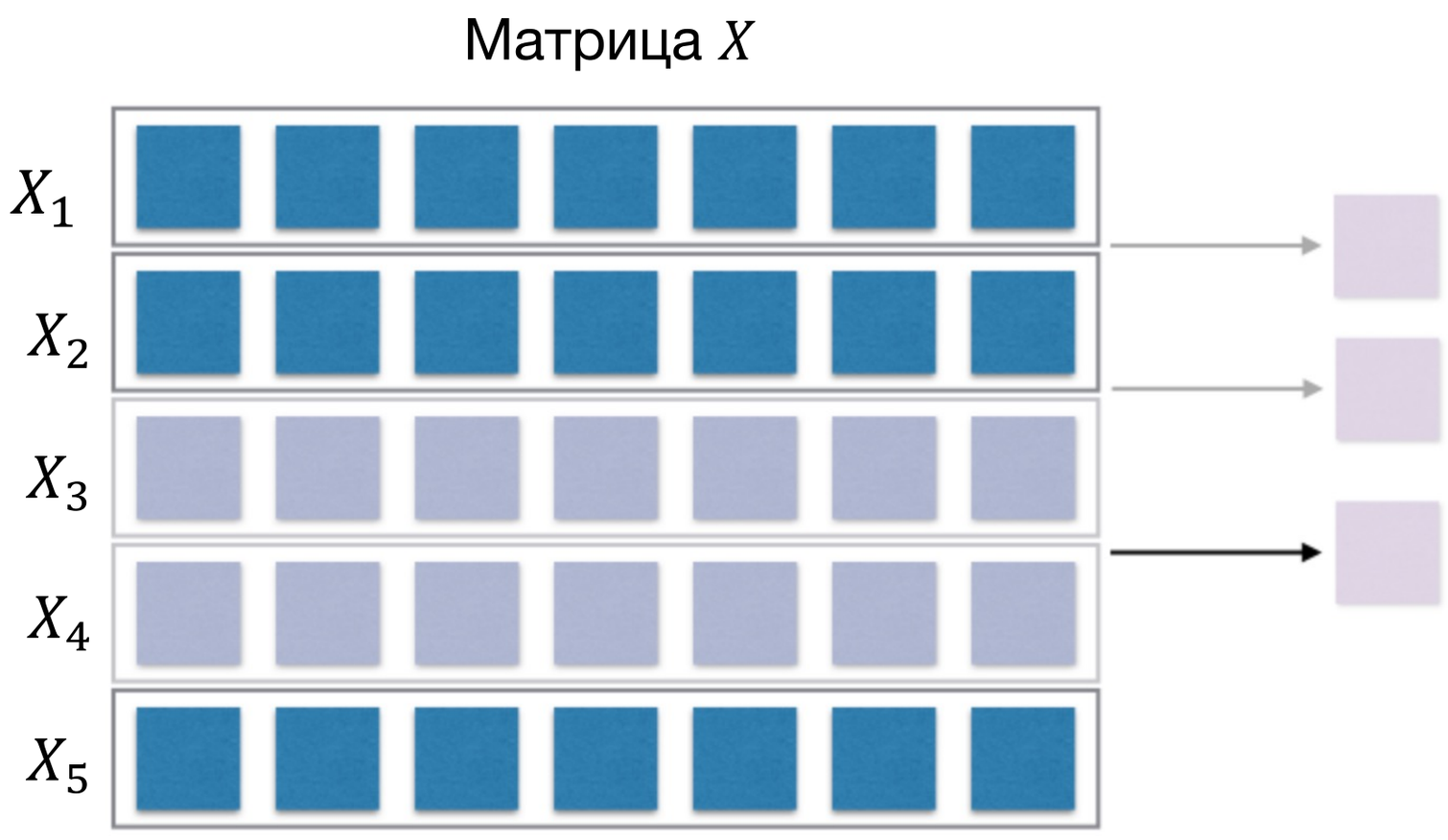
1D-Conv очень похож на 2D-Conv,
но фильтр двигается теперь только по одной размерности.




Фильтр применяется к $[X_2, X_3]$.

# 2. 1D-Convolution layer

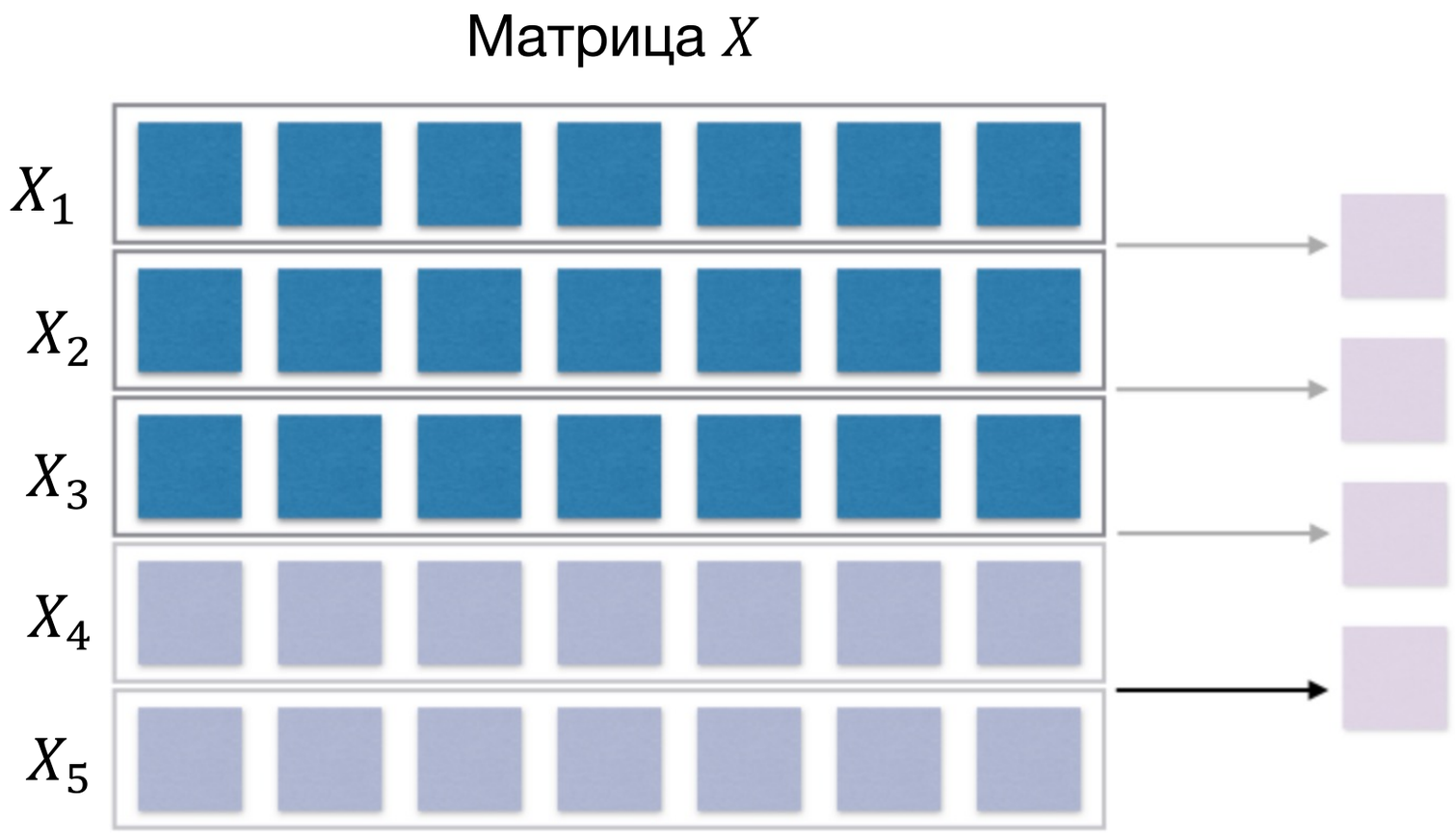
1D-Conv очень похож на 2D-Conv,
но фильтр двигается теперь только по одной размерности.




Фильтр применяется к $[X_3, X_4]$.

# 2. 1D-Convolution layer

1D-Conv очень похож на 2D-Conv,
но фильтр двигается теперь только по одной размерности.




Фильтр применяется к $[X_4, X_5]$.
В итоге получаем вектор.

# 2. 1D-Convolution layer

Пусть вход — матрица $X$ размера t×$d'$.

В случае временных рядов $t$ — кол-во элементов в последовательности, $d'$ — размер вектора.
Тогда фильтр $F$ имеет размер $K \times d'$, где $K$ — гиперпараметр.

$K$ обычно равен 3, 4, 5.

***Алгоритм 1D свёртки:***

- Фильтр скользит по первой размерности входа.
- Считается скалярное произведение текущей части и фильтра.
- Полученное значение записывается в одну ячейку выхода.

Выход — вектор размерности $t - K + 1$

***Формула 1D-свёртки:*** $(A * F)_i = \sum_{j=1}^{K} \sum_{r=1}^{M} F_{jr} \cdot A_{i+j-1,r} + b$

***Мотивация:***
Как и для картинок смотрим на набор соседних элементов
и пытаемся найти в них паттерны.
Для временных рядов это объекты из окрестности момента времени t.

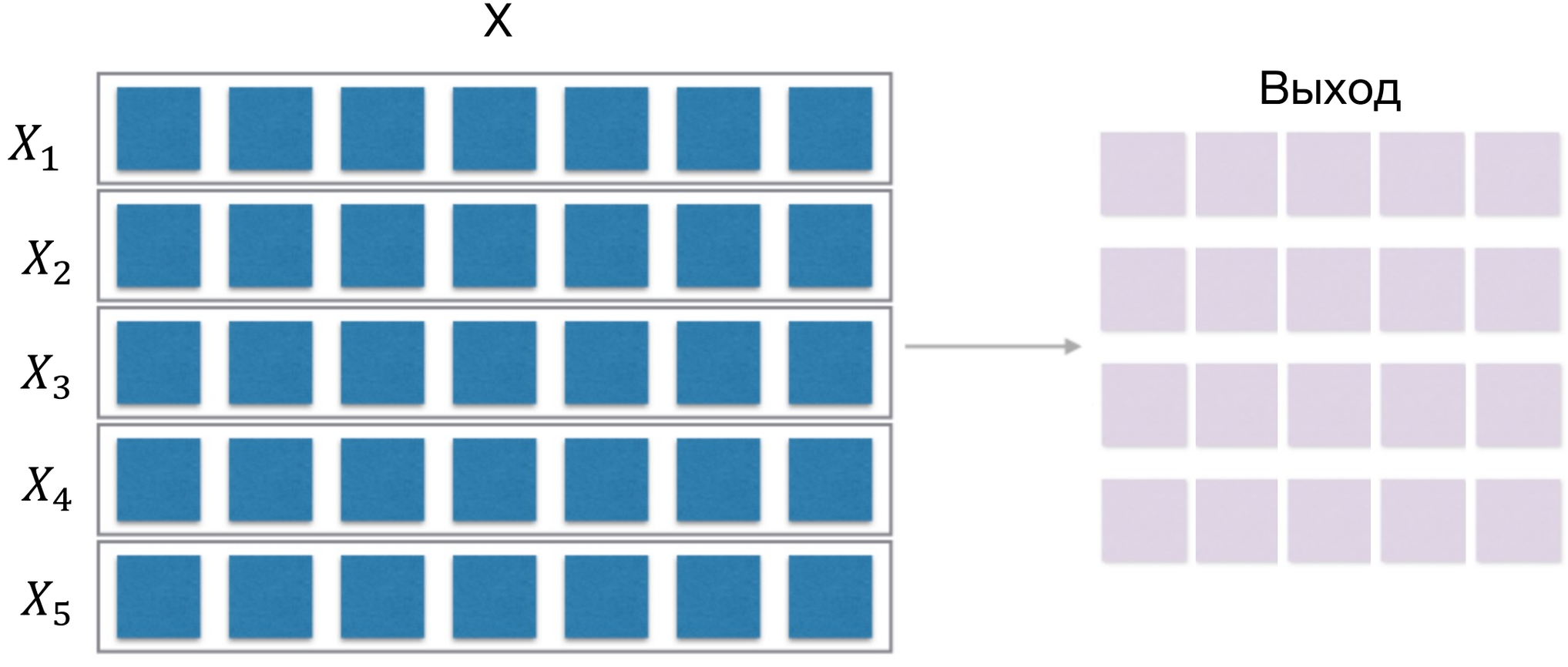
# 2. Много фильтров

Как и для 2D свёртки, применяем несколько фильтров одного размера.

Карта — матрица из векторов, полученных от разных фильтров.
Полученная матрица обладает тем же свойством.

У нее одна ось имеет порядок, другая - нет

К такой матрице можно и далее применить 1D свёртку.



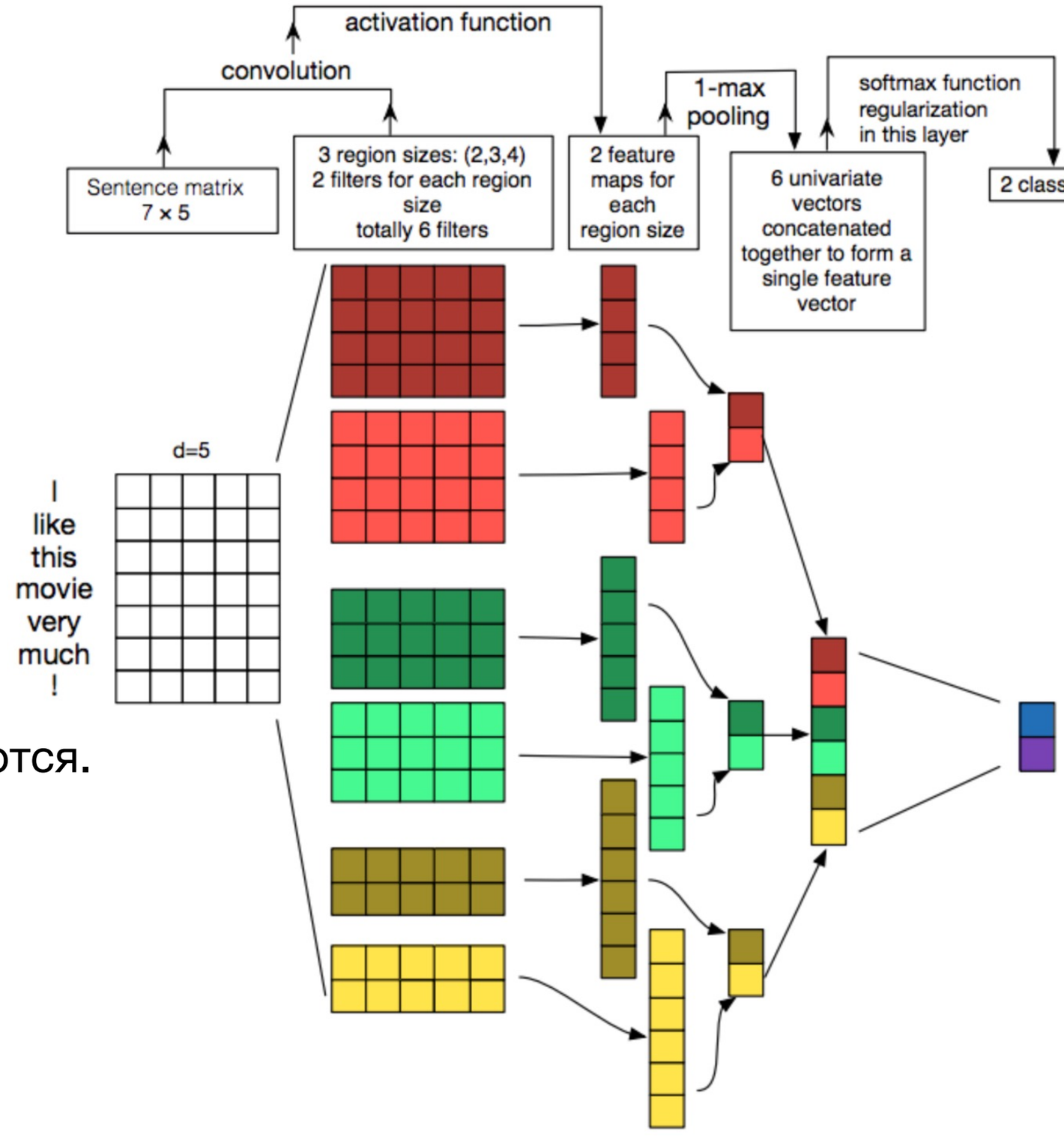
# 2. Фильтры разных размеров

Можем применять свёртки разных размеров.

Обычно это делается так:

Для каждого размера фильтра:

1. Применяется свёртка
   с фильтром этого размера.
2. К полученной карте
   применяется pooling.
   Получается вектор длины
   равной кол-ву фильтров
   в свёртке.
3. Все такие векторы
   от свёрток разного размера
   конкатенируются или складываются.



# 2. Используемые техники

Аналогично 2D-Conv, применяются:

- Padding
- Stride
- Свёртка с $K = 1$.

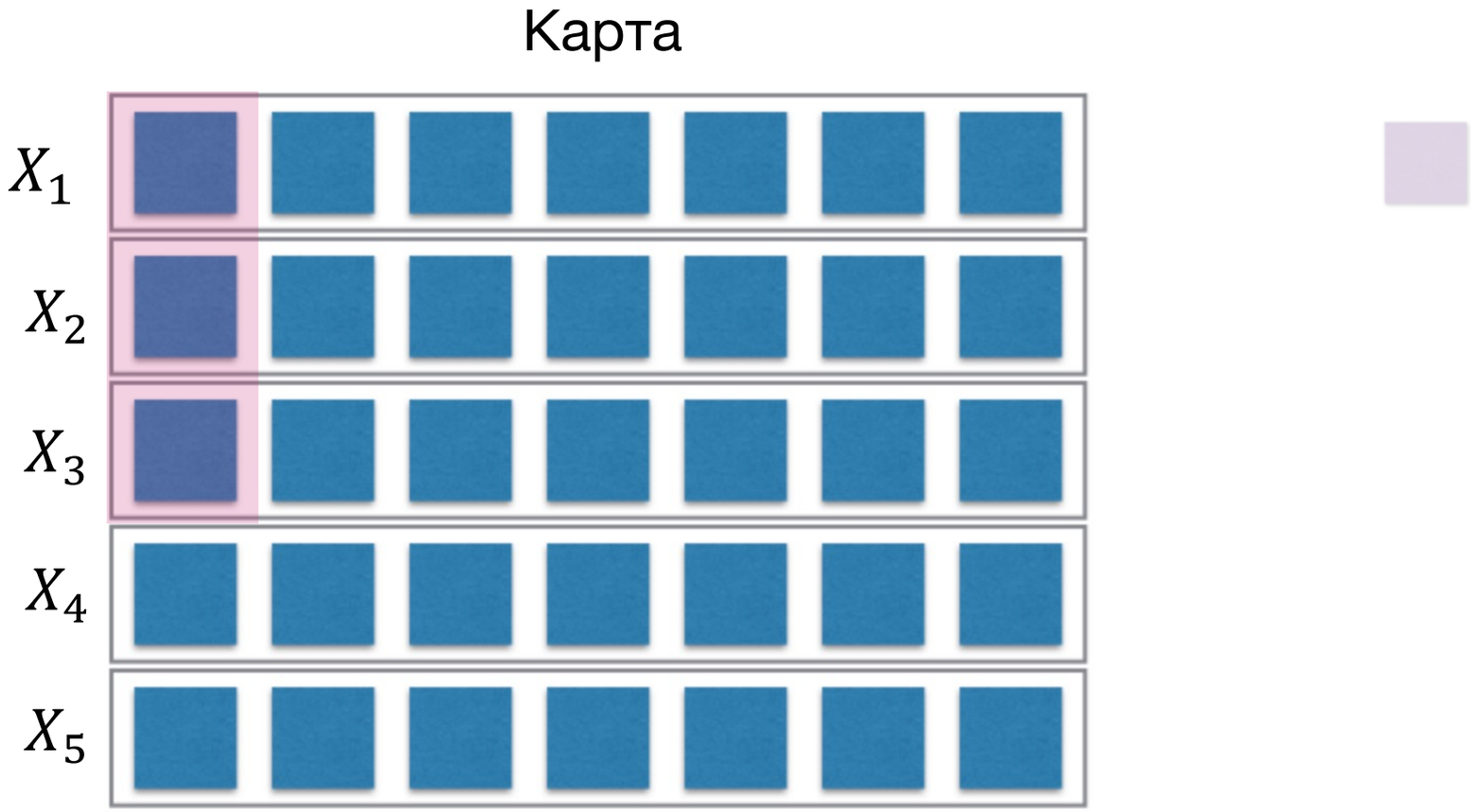
Аналог 2D свёртки 1x1.

- Все остальное, что применяется для Conv 2D.

# 2. 1D-Pooling layer

1D-Pooling скользит окном размера $(1, k)$ по карте и считает максимум.

$k$ — гиперпараметр.



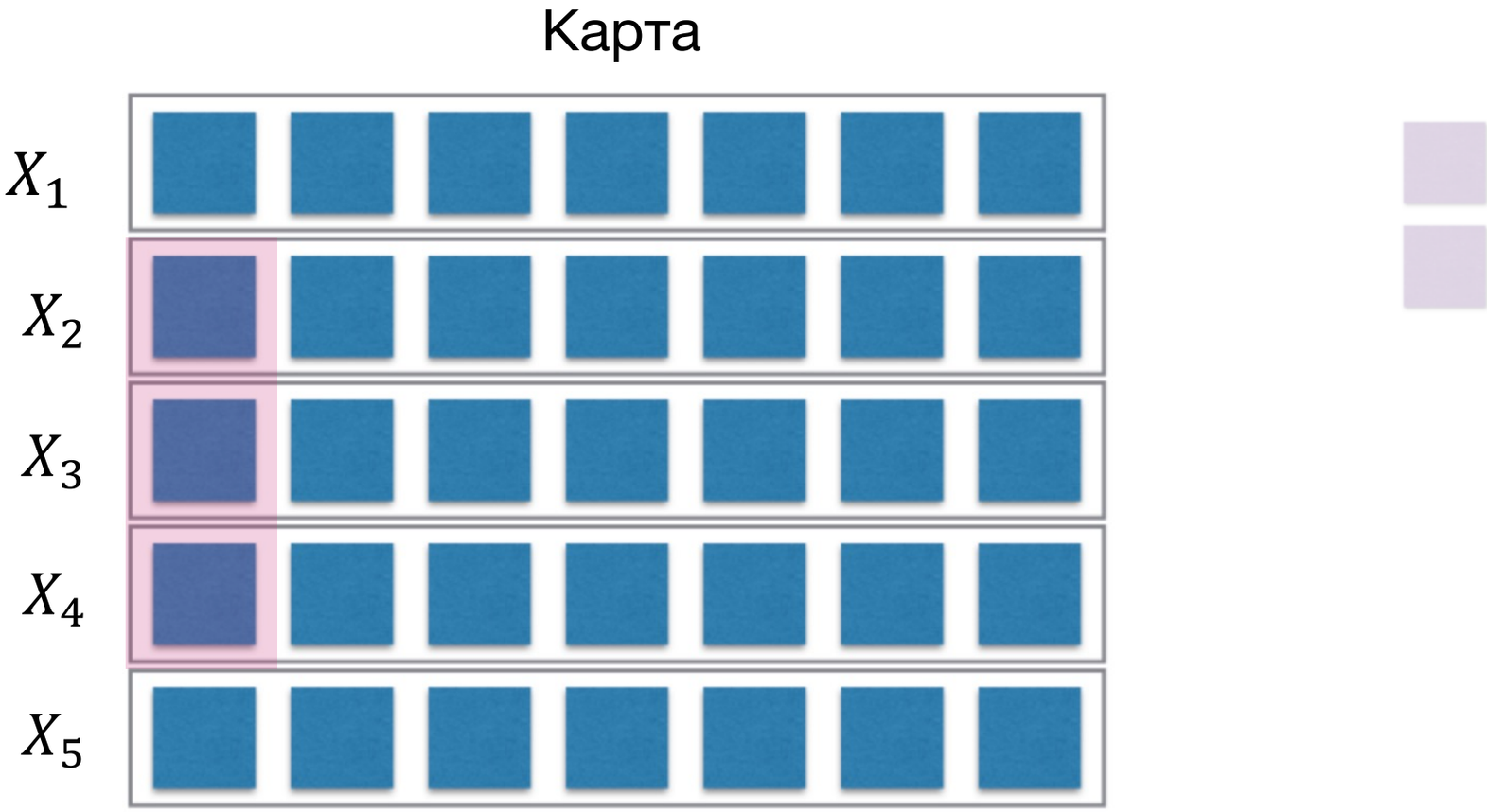

*Интерпретация*:
Нашелся ли паттерн, соответствующий определенному фильтру,

среди $k$ подряд идущих объектов в посл-ти.

# 2. 1D-Pooling layer

1D-Pooling скользит окном размера $(1, k)$ по карте и считает максимум.

$k$ — гиперпараметр.

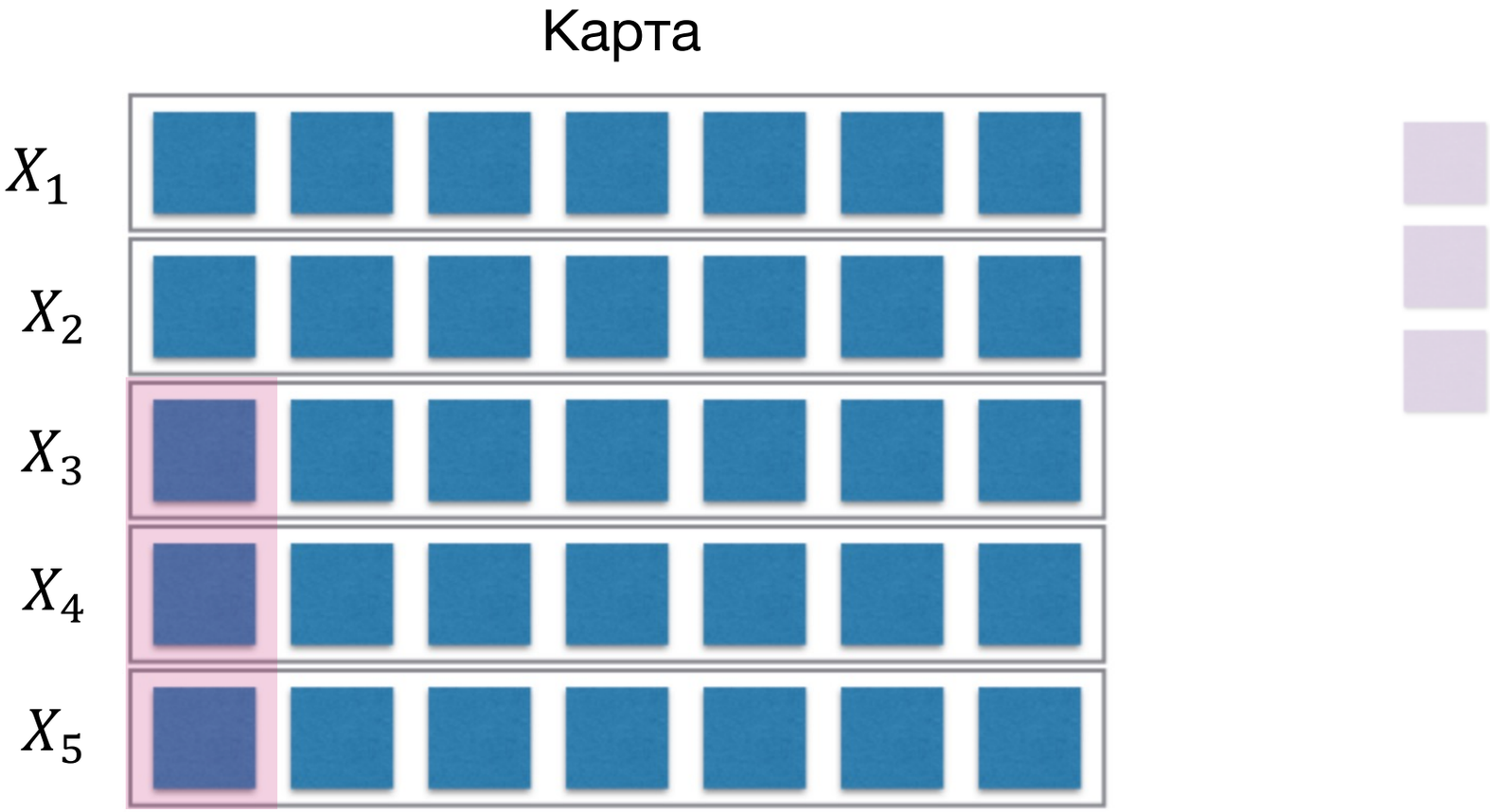



*Интерпретация*:
Нашелся ли паттерн, соответствующий определенному фильтру, среди $k$ подряд идущих объектов в посл-ти.

# 2. 1D-Pooling layer

1D-Pooling скользит окном размера $(1, k)$ по карте и считает максимум.

$k$ — гиперпараметр.

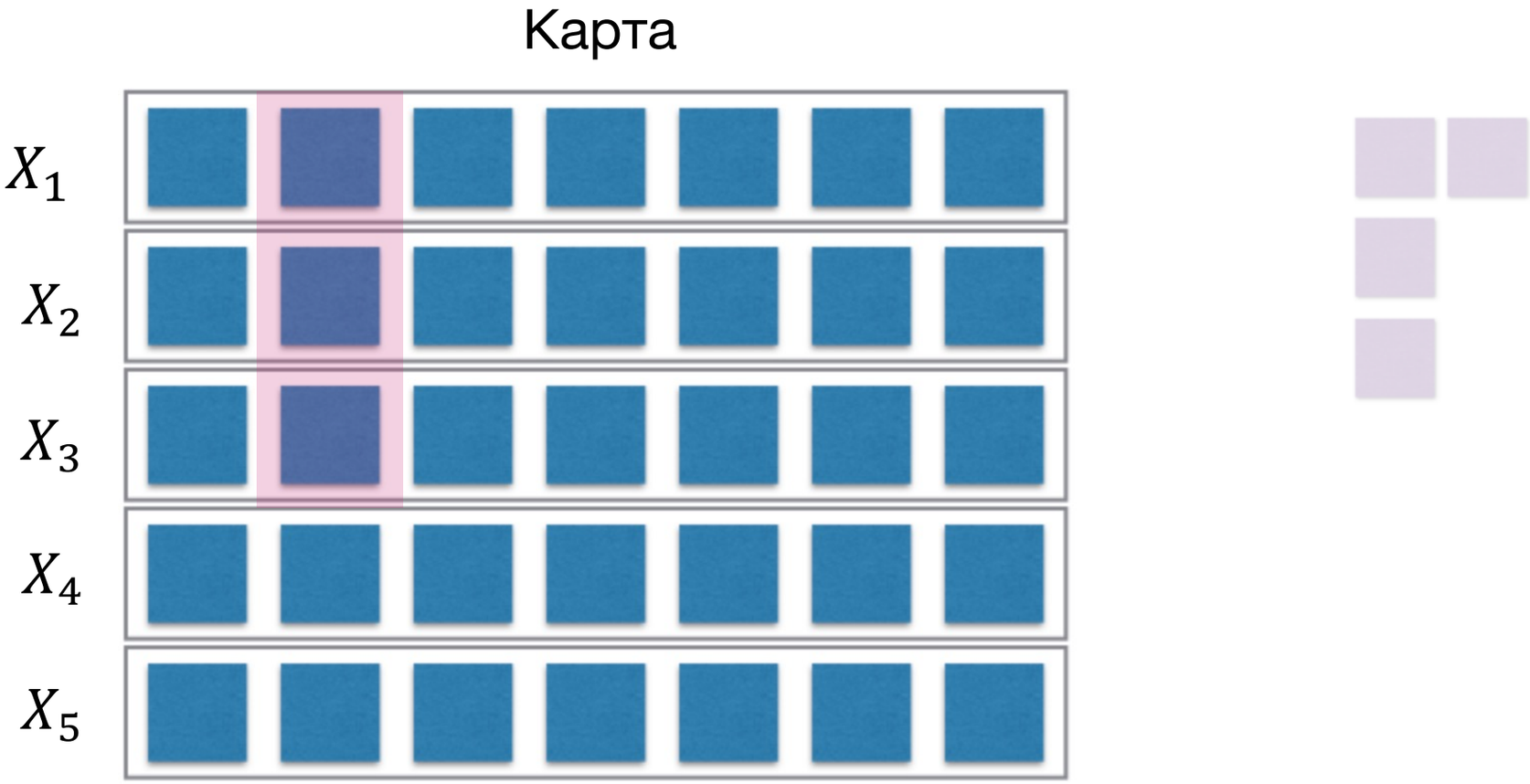



*Интерпретация*:
Нашелся ли паттерн, соответствующий определенному фильтру, среди $k$ подряд идущих объектов в посл-ти.

# 2. 1D-Pooling layer

1D-Pooling скользит окном размера $(1, k)$ по карте и считает максимум.

$k$ — гиперпараметр.

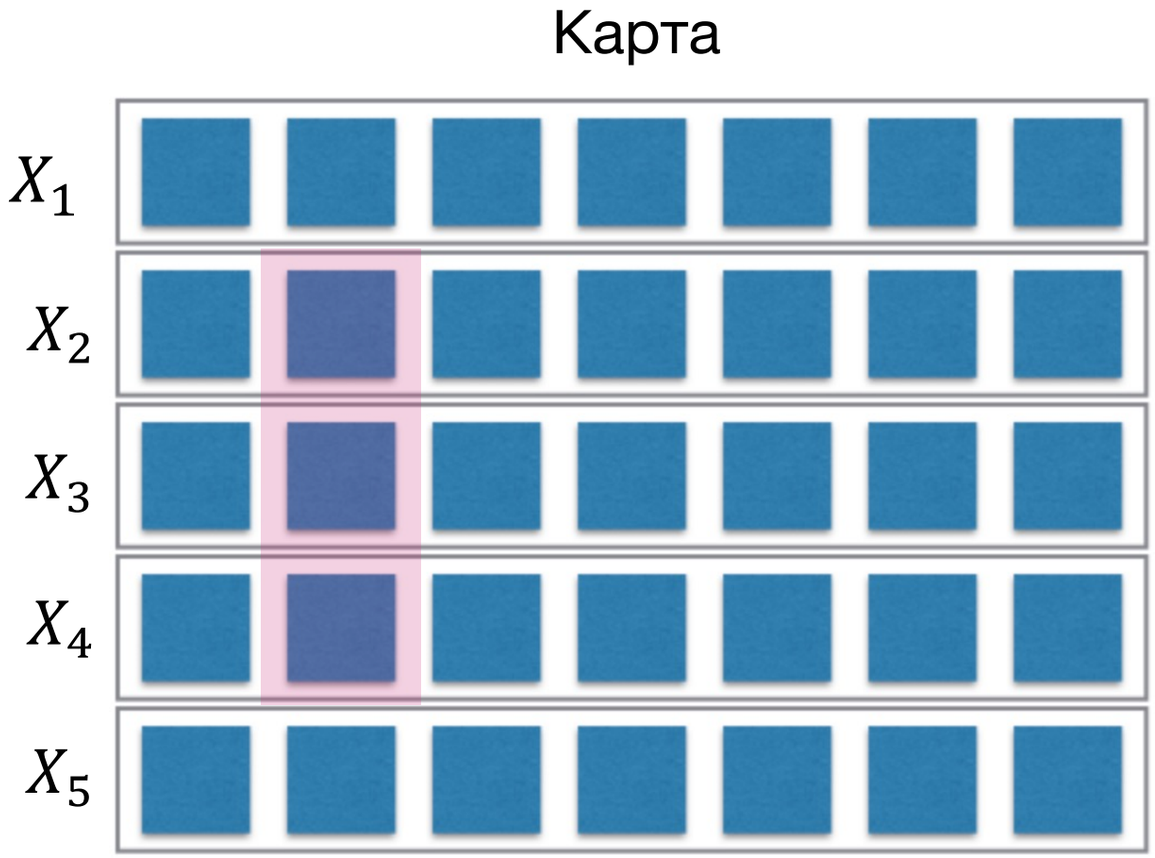



*Интерпретация*:
Нашелся ли паттерн, соответствующий определенному фильтру, среди $k$ подряд идущих объектов в посл-ти.

# 2. 1D-Pooling layer

1D-Pooling скользит окном размера $(1, k)$ по карте и считает максимум.

$k$ — гиперпараметр.

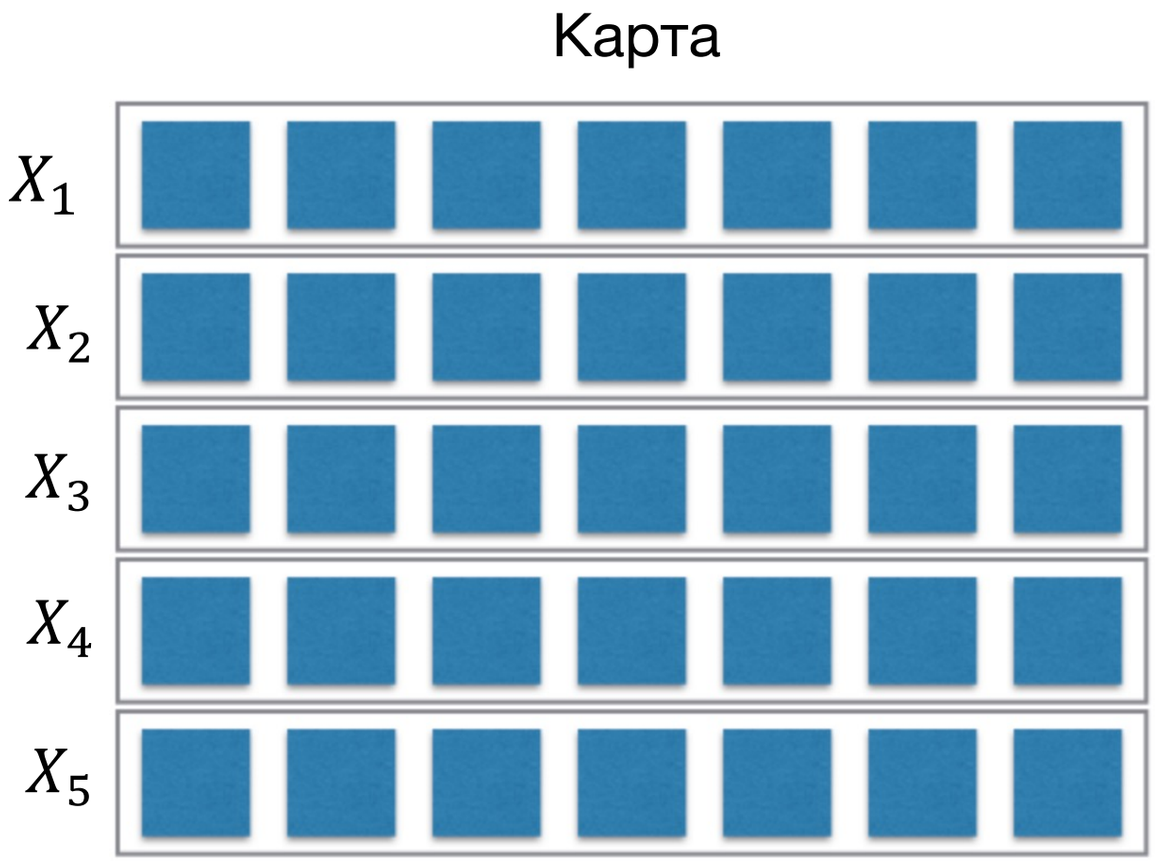



*Интерпретация*:
Нашелся ли паттерн, соответствующий определенному фильтру, среди $k$ подряд идущих объектов в посл-ти.

# 2. 1D-Pooling layer

1D-Pooling скользит окном размера $(1, k)$ по карте и считает максимум.

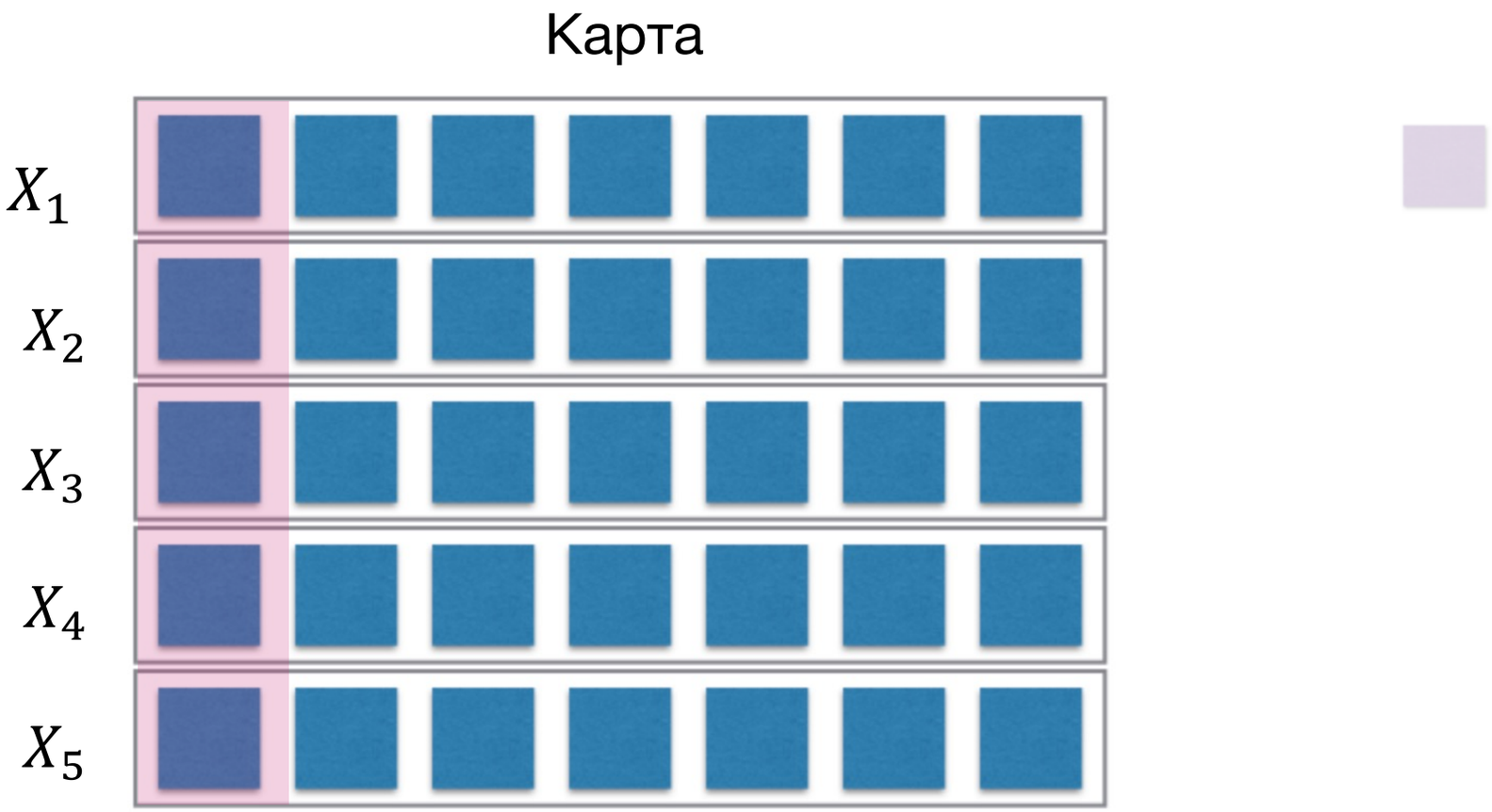
$k$ — гиперпараметр.




*Интерпретация*:
Нашелся ли паттерн, соответствующий определенному фильтру, среди $k$ подряд идущих объектов в посл-ти.

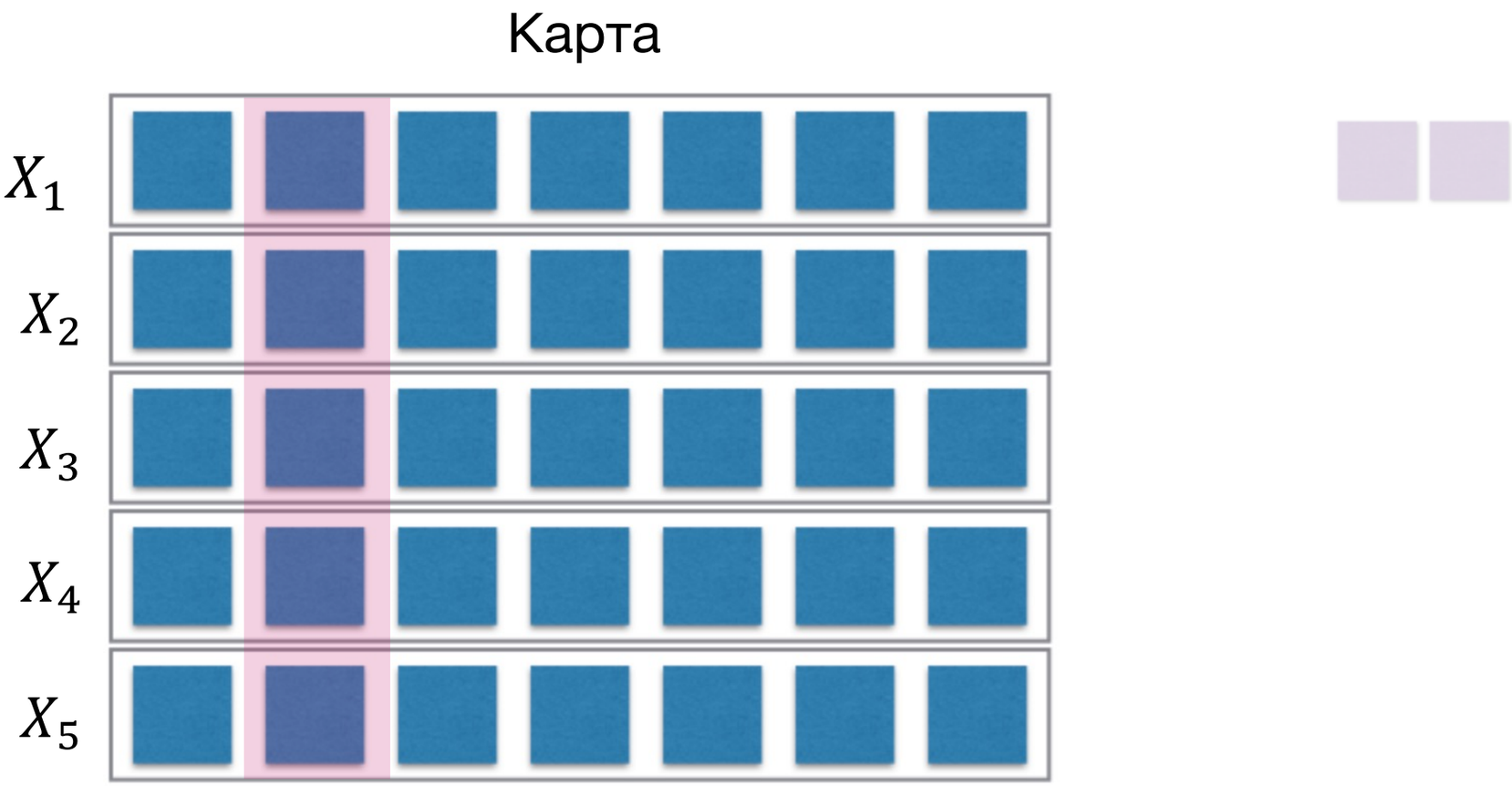
# 2. 1D-GlobalPooling layer

1D-GlobalPooling считает максимум (сумму, среднее) по размерности последовательности.



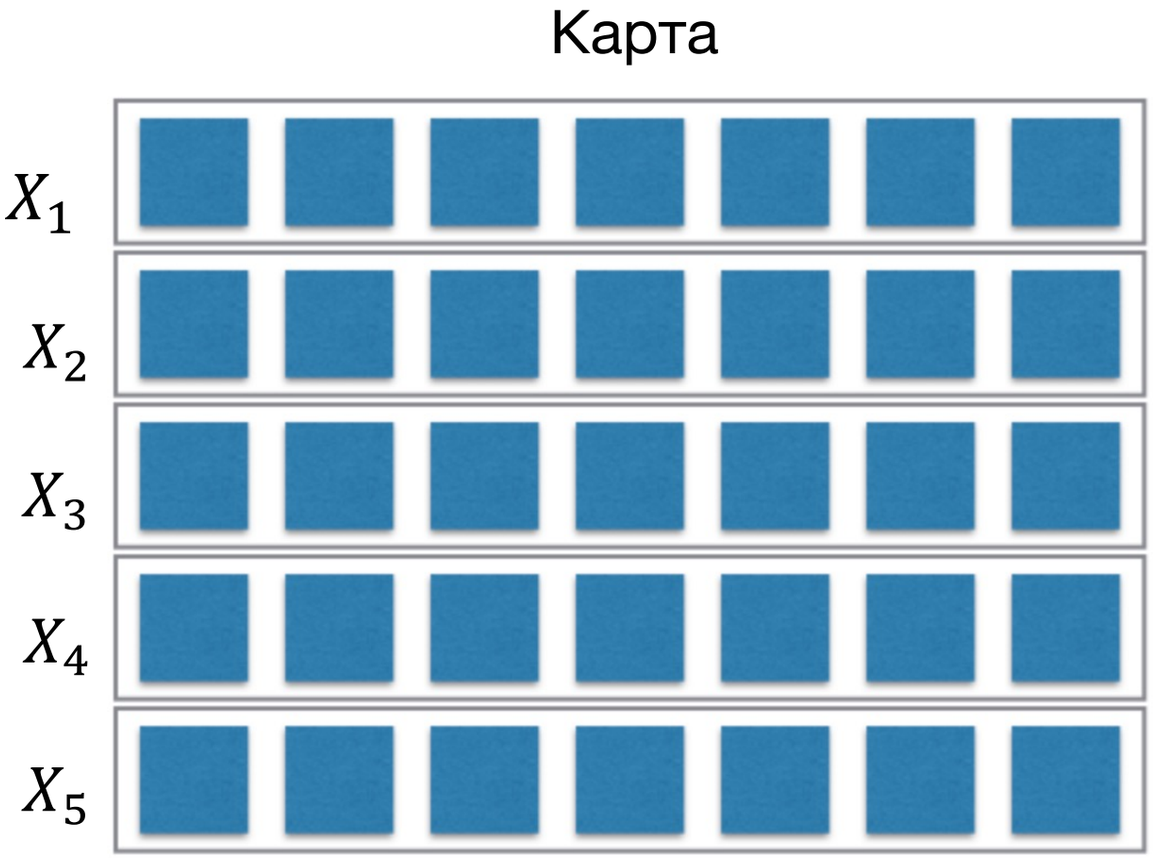
# 2. 1D-GlobalPooling layer

1D-GlobalPooling считает максимум (сумму, среднее) по размерности последовательности.



# 2. 1D-GlobalPooling layer

1D-GlobalPooling считает максимум (сумму, среднее) по размерности последовательности.



# 2. Итоги

**Плюсы**:

# 2. Итоги

**Плюсы**:

- Позволяет кодировать информацию о последовательности
- Поиск паттернов в последовательности

**Минусы**:

# 2. Итоги

**Плюсы**:

- Позволяет кодировать информацию о последовательности
- Поиск паттернов в последовательности

**Минусы**:

- Ограниченность контекста
- Количество параметров: $d(d'K + 1)$ – линейно растет с увеличением размера окна

Хотелось бы научиться кодировать последовательности неограниченной длины

# Recurrent Neural Networks

# RNN

1. Vanila RNN

2. LSTM

3. GRU

4. Bi-RNN и Deep RNN

# 1. RNN

Пусть имеется некоторая последовательность.

Например, последовательность из эмбеддингов слов.
**Задача**: По входной посл-ти получить некоторый вектор, содержащий информацию о подаваемой последовательности.

*Вход*: $x_1, x_2, \ldots, x_t$, где $x_i$ — вектор размера $d'$.

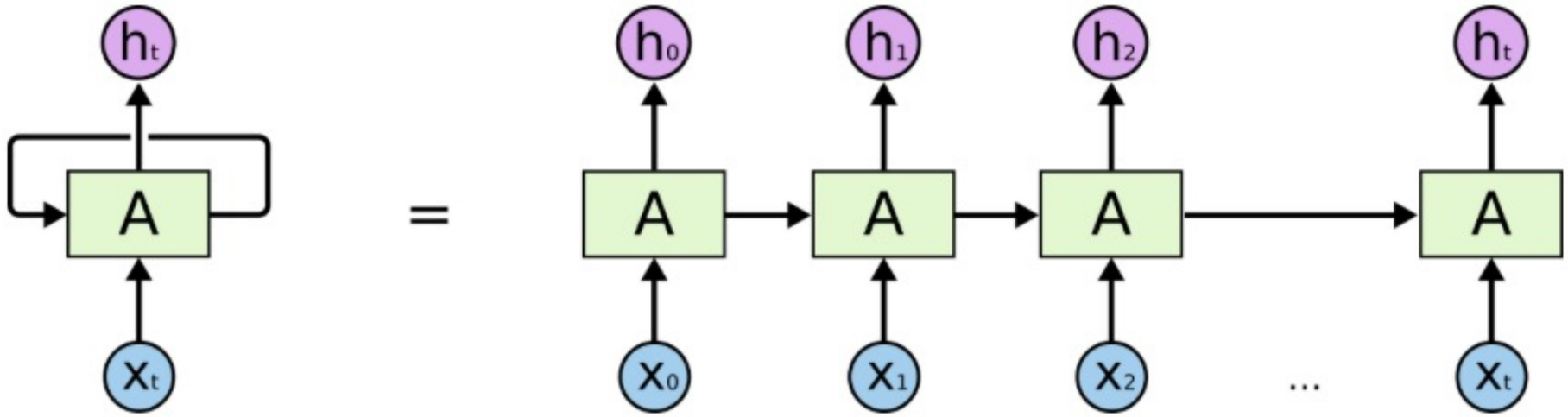
*Выход*: вектор $e$ размера $d$.

Вектор $e$ кодирует информацию об $x_1, x_2, \ldots, x_t$.

$d$ —гиперпараметр.

# 1. RNN

Recurrent Neural Network — слой, умеющий обрабатывать последовательности.

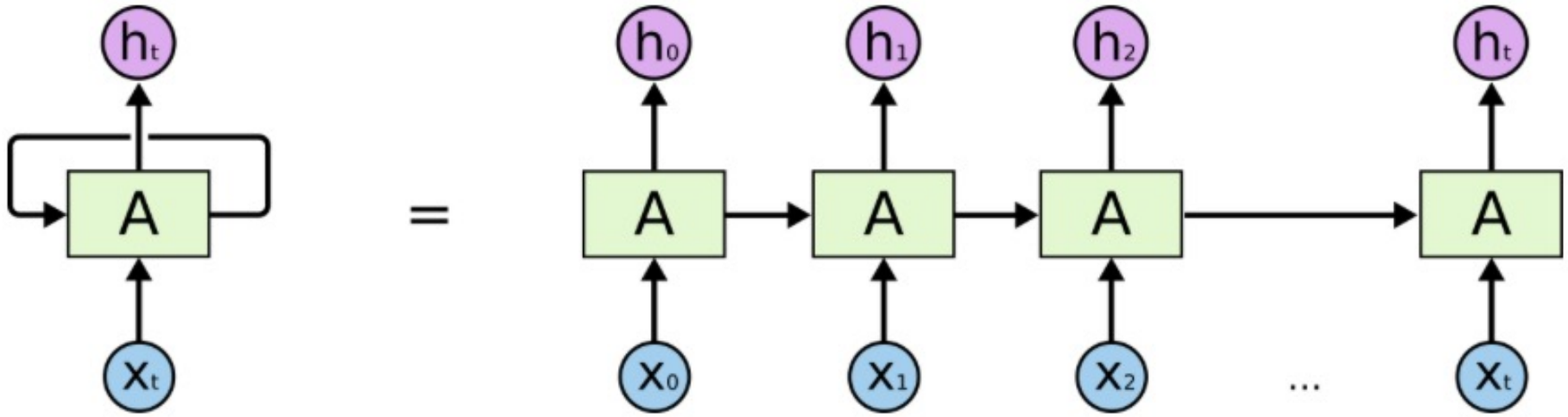



Здесь $x_t$ — входной элемент во время $t$.
$h_t$ — скрытое состояние во время $t$.
$W_h, W_x$ — обучаемые веса (матрицы).

$$h_t = \tanh(h_{t-1} W_h + x_t W_x)$$

# 1. RNN

Recurrent Neural Network — слой, умеющий обрабатывать последовательности.




Здесь $x_t$ — входной элемент во время $t$.
$h_t$ — скрытое состояние во время $t$.
$W_h, W_x$ — обучаемые веса (матрицы).

$$h_t = \tanh(h_{t-1} W_h + x_t W_x)$$

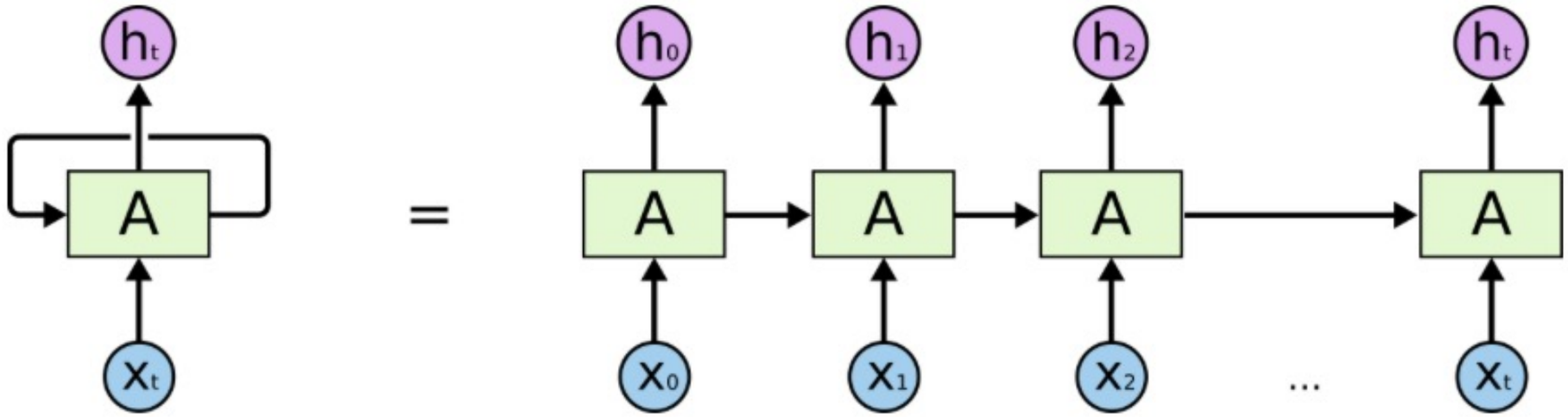
RNN на шаге $t$

- Хранит скрытое состояние $h_{t-1}$ с предыдущего шага
- Получает на вход новый элемент $x_t$
- Обновляет скрытое состояние, получая $h_t$

$h_t$ - некоторое знание обо всех уже считанных элементах посл-ти.
Используя это знание можно делать предсказания.

# 1. RNN

Recurrent Neural Network — слой, умеющий обрабатывать последовательности.




Здесь $x_t$ — входной элемент во время $t$.
$h_t$ — скрытое состояние во время $t$.
$W_h, W_x$ — обучаемые веса (матрицы).

$$h_t = \tanh(h_{t-1} W_h + x_t W_x)$$

RNN на шаге $t$

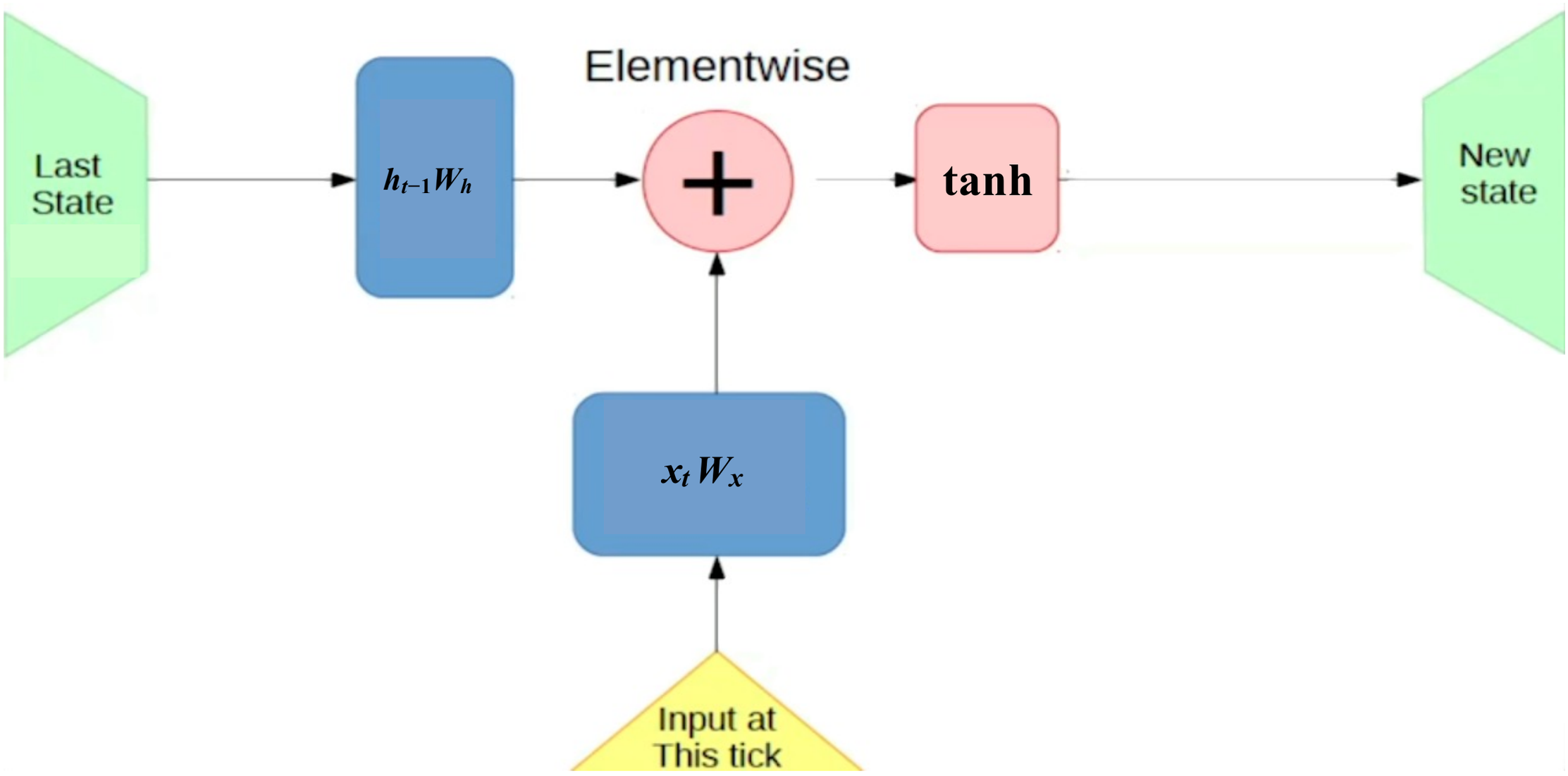
- Хранит скрытое состояние $h_{t-1}$ с предыдущего шага
- Получает на вход новый элемент $x_t$
- Обновляет скрытое состояние, получая $h_t$

$h_t$ - некоторое знание обо всех уже считанных элементах посл-ти.
Используя это знание можно делать предсказания.

**Важно:** Веса $W_h, W_x$ одни и те же для всех ячеек.

# 1. RNN

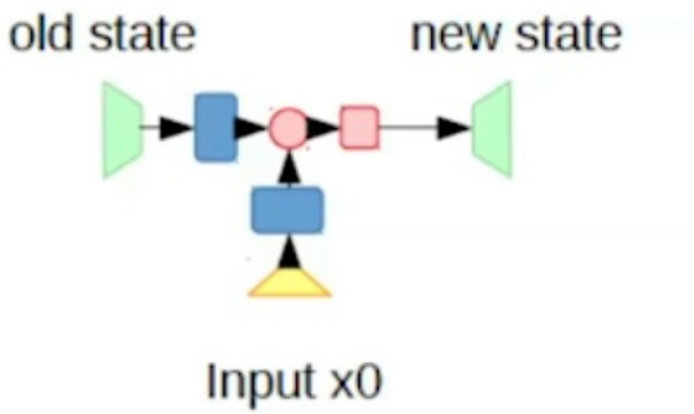
Посмотрим на одну ячейку RNN более детально.




$$h_t = \tanh(h_{t-1}W_h + x_t W_x)$$

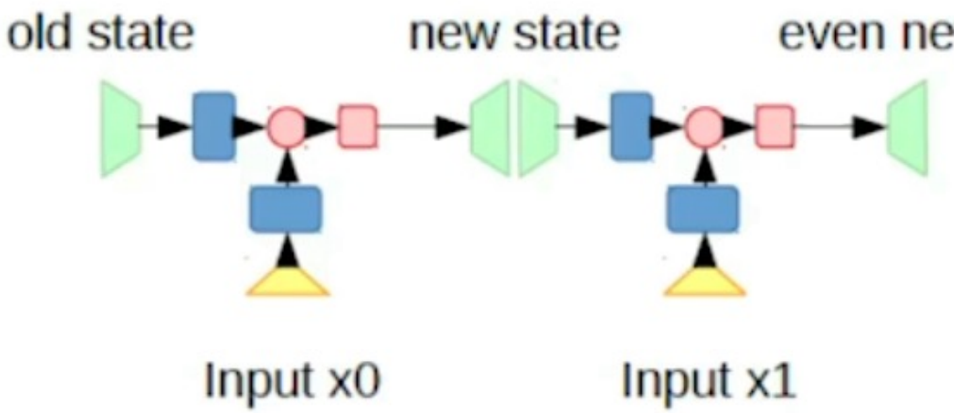
# 1. RNN

Ячейка применяется к первому элементу входной посл-ти.
Получаем новое скрытое состояние системы.



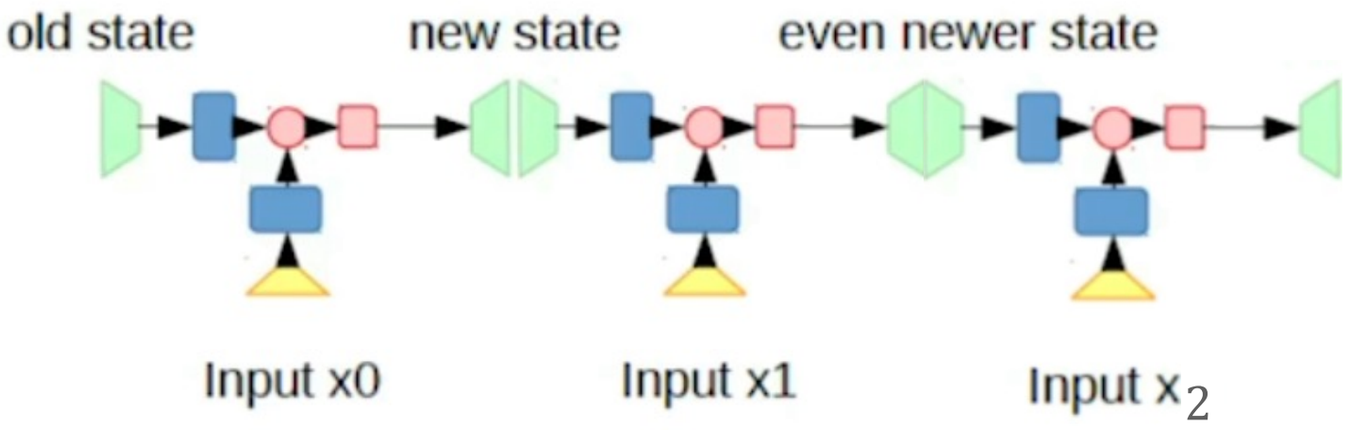
# 1. RNN

Далее ячейка применяется к новому элементу посл-ти.
Получается следующее скрытое состояние.



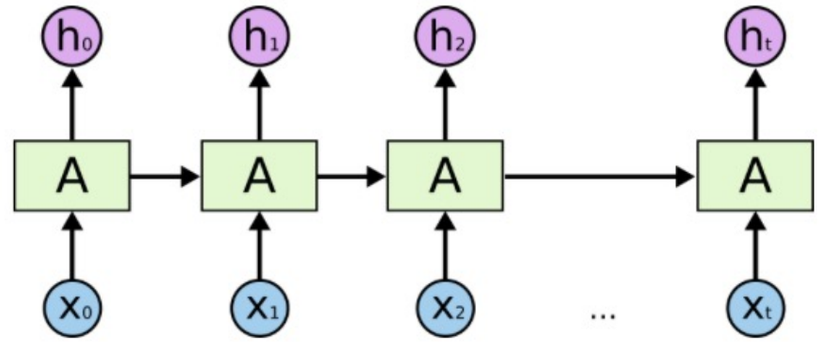
# 1. RNN

Далее ячейка применяется к новому элементу посл-ти.
Получается следующее скрытое состояние.



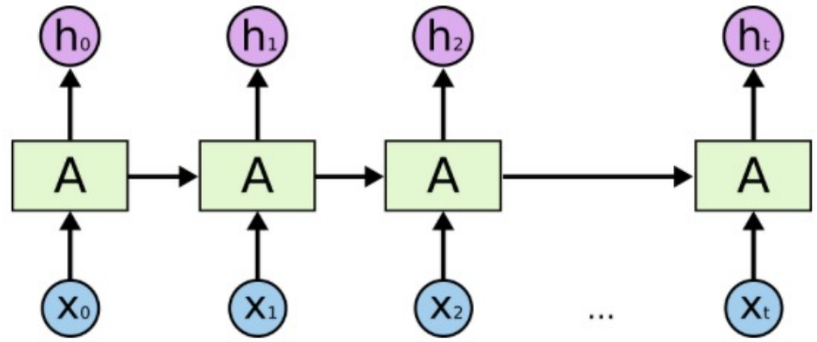
# 1. RNN

**Как решать задачи с помощью RNN?**



# 1. RNN

**Как решать задачи с помощью RNN?**



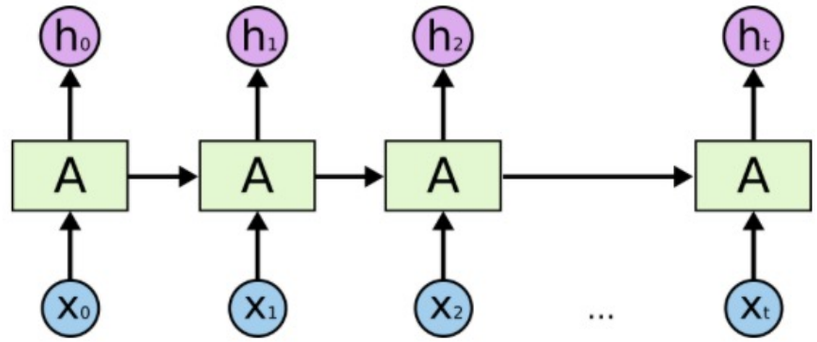

1. Предсказать таргет последовательности из последнего состояния.
Например, предсказание следующего элемента последовательности.

Рассмотрим последнее состояние $h_t$, полученное после обработки последнего $x_t$.

$h_t$ хранит информацию о всей посл-ти $\Rightarrow$ по $h_t$ можно предсказать таргет всей пос-ти.

# 1. RNN

**Как решать задачи с помощью RNN?**




1. Предсказать таргет последовательности из последнего состояния.
Например, предсказание следующего элемента последовательности.

Рассмотрим последнее состояние $h_t$, полученное после обработки последнего $x_t$.

$h_t$ хранит информацию о всей посл-ти $\Rightarrow$ по $h_t$ можно предсказать таргет всей пос-ти.

2. Сделать предсказание для каждого элемента последовательности.

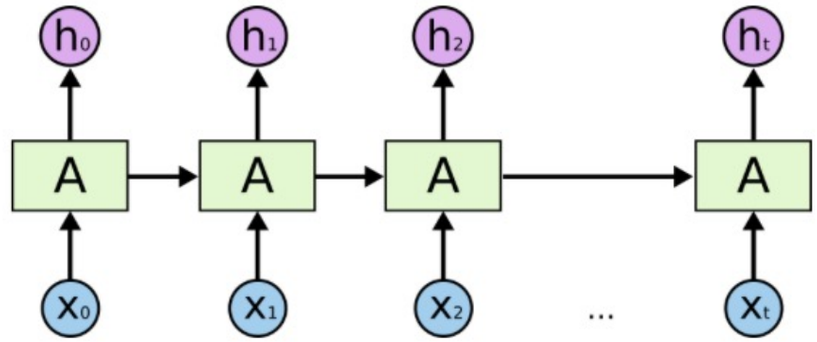
Например, задача part-of-speech tagging.

Рассмотрим все скрытые состояния, полученные на разных этапах.
Каждое $h_t$ хранит информацию о $x_1, x_2, \ldots, x_t$.

Из состояния $h_t$ можно сделать предсказание для $x_t$ или посл-ти $x_1, x_2, \ldots, x_t$.

# 1. RNN

**Как решать задачи с помощью RNN?**




1. Предсказать таргет последовательности из последнего состояния.
Например, предсказание следующего элемента последовательности.

Рассмотрим последнее состояние $h_t$, полученное после обработки последнего $x_t$.

$h_t$ хранит информацию о всей посл-ти $\Rightarrow$ по $h_t$ можно предсказать таргет всей пос-ти.

2. Сделать предсказание для каждого элемента последовательности.

Например, задача part-of-speech tagging.

Рассмотрим все скрытые состояния, полученные на разных этапах.
Каждое $h_t$ хранит информацию о $x_1, x_2, \ldots, x_t$.

Из состояния $h_t$ можно сделать предсказание для $x_t$ или посл-ти $x_1, x_2, \ldots, x_t$.

3. Предсказать таргет последовательности по всей посл-ти состояний.

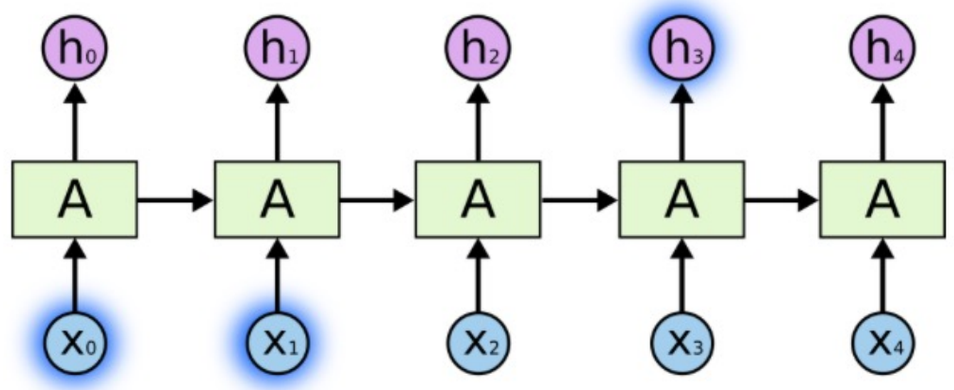
Решаемые задачи аналогичны пункту 1.

Рассмотрим все скрытые состояния, полученные на разных этапах.
К посл-ти состояний применяется некоторая функция, например усреднение.
Из полученного можно предсказать таргет для всей посл-ти.

# 1. RNN

**Как обучать?**

$$h_t = \tanh(h_{t-1} W_h + x_t W_x)$$

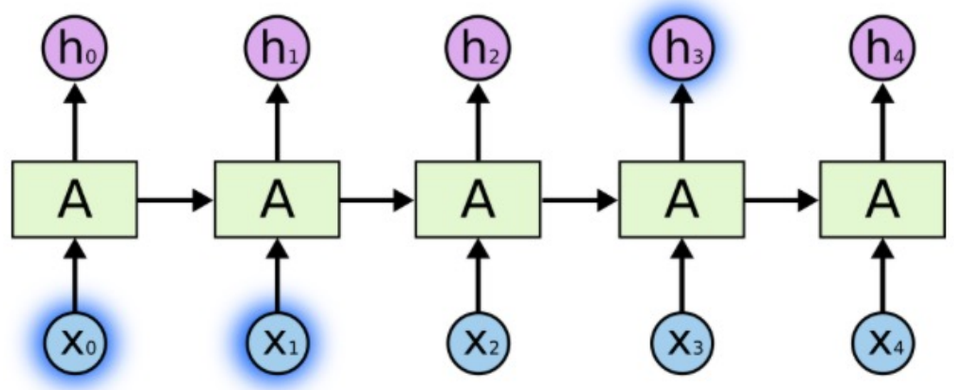


# 1. RNN

**Как обучать?**

$$h_t = \tanh(h_{t-1} W_h + x_t W_x)$$

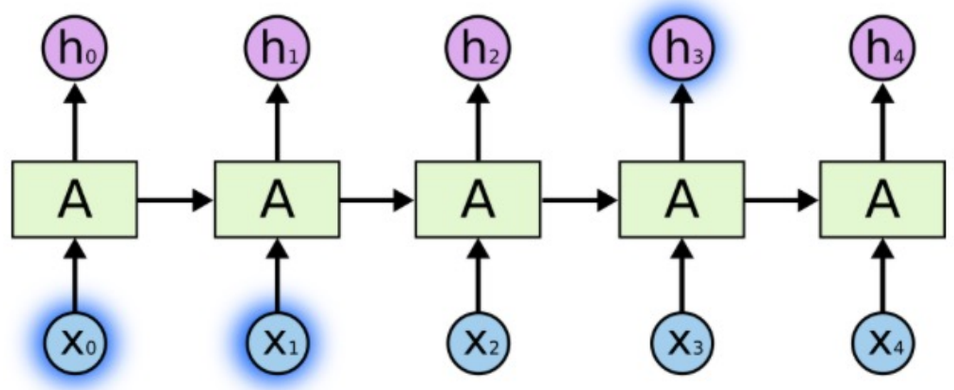
Посчитаем производные:



# 1. RNN

**Как обучать?**

$$h_t = \tanh(h_{t-1} W_h + x_t W_x)$$




Посчитаем производные:

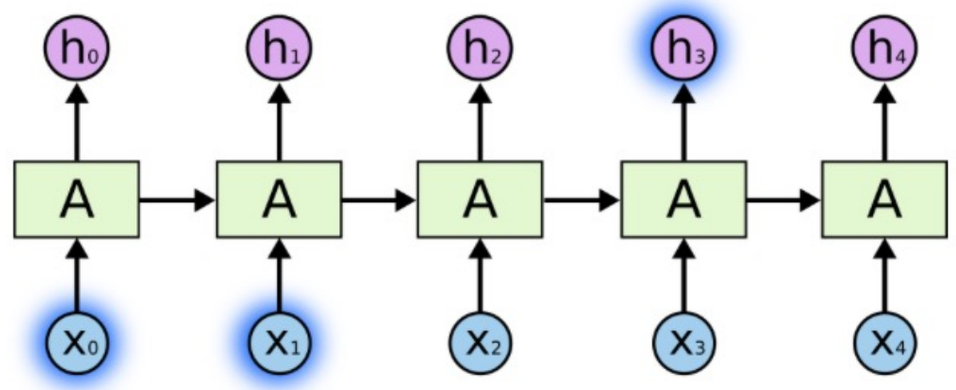
$$\frac{\partial h_t}{\partial W_h} = \sum_{k=0}^{t} \frac{\partial h_t}{\partial h_k} \cdot \frac{\partial h_k}{\partial W_h} |_{h_{k-1} isconst}$$

Здесь $\frac{\partial h_k}{\partial W_h} |_{h_{k-1} isconst}$ — градиент посчитанный, считая, что $h_{k-1}$ не зависит от $W_h$.

# 1. RNN

**Как обучать?**

$$h_t = \tanh(h_{t-1} W_h + x_t W_x)$$




Посчитаем производные:

$$\frac{\partial h_t}{\partial W_h} = \sum_{k=0}^{t} \frac{\partial h_t}{\partial h_k} \cdot \frac{\partial h_k}{\partial W_h} |_{h_{k-1} isconst}$$

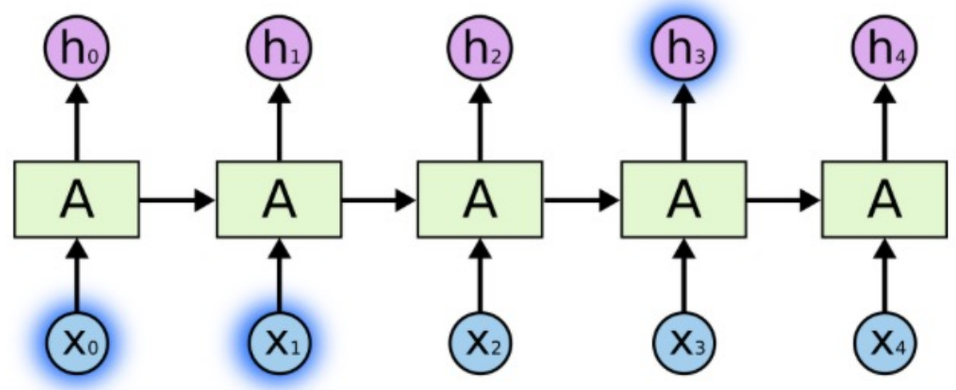
$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}}$$

Здесь $\frac{\partial h_k}{\partial W_h} |_{h_{k-1} isconst}$ — градиент посчитанный, считая, что $h_{k-1}$ не зависит от $W_h$.

# 1. RNN

**Как обучать?**

$$h_t = \tanh(h_{t-1} W_h + x_t W_x)$$




Посчитаем производные:

$$\frac{\partial h_t}{\partial W_h} = \sum_{k=0}^{t} \frac{\partial h_t}{\partial h_k} \cdot \frac{\partial h_k}{\partial W_h} |_{h_{k-1} isconst}$$

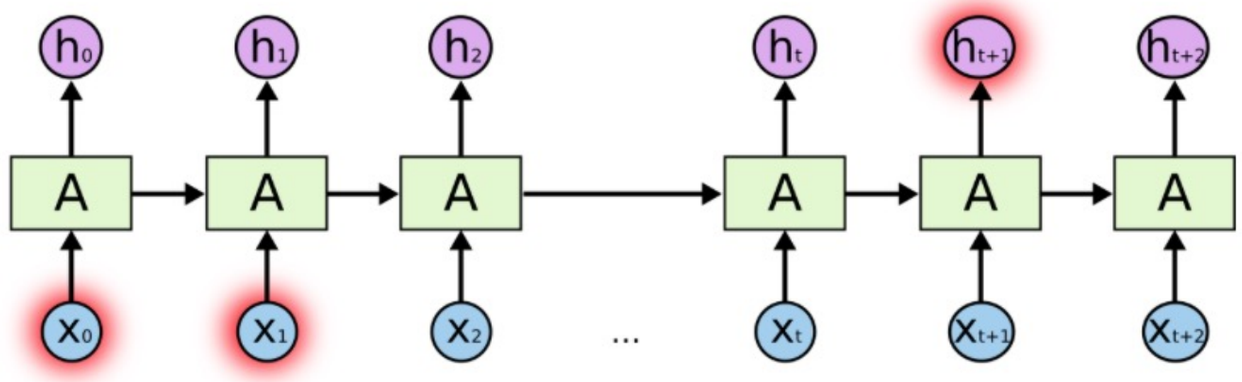
$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}}$$

**Проблемы:**

# 1. RNN

**Как обучать?**

$$h_t = \tanh(h_{t-1} W_h + x_t W_x)$$




Посчитаем производные:

$$\frac{\partial h_t}{\partial W_h} = \sum_{k=0}^{t} \frac{\partial h_t}{\partial h_k} \cdot \frac{\partial h_k}{\partial W_h} |_{h_{k-1} isconst}$$

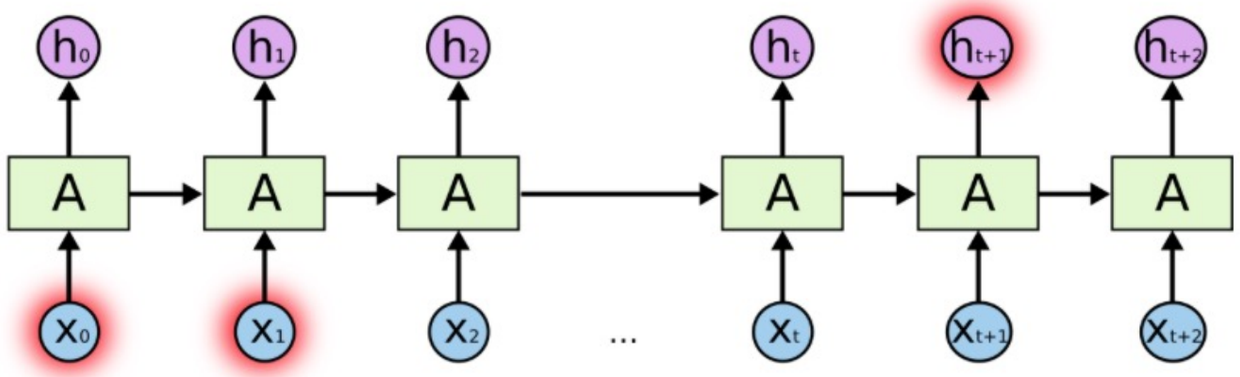
$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}}$$

**Проблемы:**

# 1. RNN

**Как обучать?**

$$h_t = \tanh(h_{t-1} W_h + x_t W_x)$$




Посчитаем производные:

$$\frac{\partial h_t}{\partial W_h} = \sum_{k=0}^{t} \frac{\partial h_t}{\partial h_k} \cdot \frac{\partial h_k}{\partial W_h} |_{h_{k-1} isconst}$$

$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}}$$

**Проблемы:**

$$\frac{\partial h_i}{\partial h_{i-1}} = tanh'(h_{i-1} W_h + x_i W_x) \cdot W_h \Rightarrow \frac{\partial h_t}{\partial h_k} = \prod_{i=k+1}^{t} tanh'(h_{i-1} W_h + x_i W_x) \cdot W_h^{t-k}$$

1. Затухание градиентов.

Если норма матрицы $W_h$ меньше 1, то $\partial h_t / \partial h_k$ может затухнуть.
Тогда градиенты не будут доходить до далеких входных элементов.

2. Взрыв градиентов.

Если норма матрицы $W_h$ больше 1, то $\partial h_t / \partial h_k$ может улететь в бесконечность.

Тогда после этого и веса улетят в бесконечность.

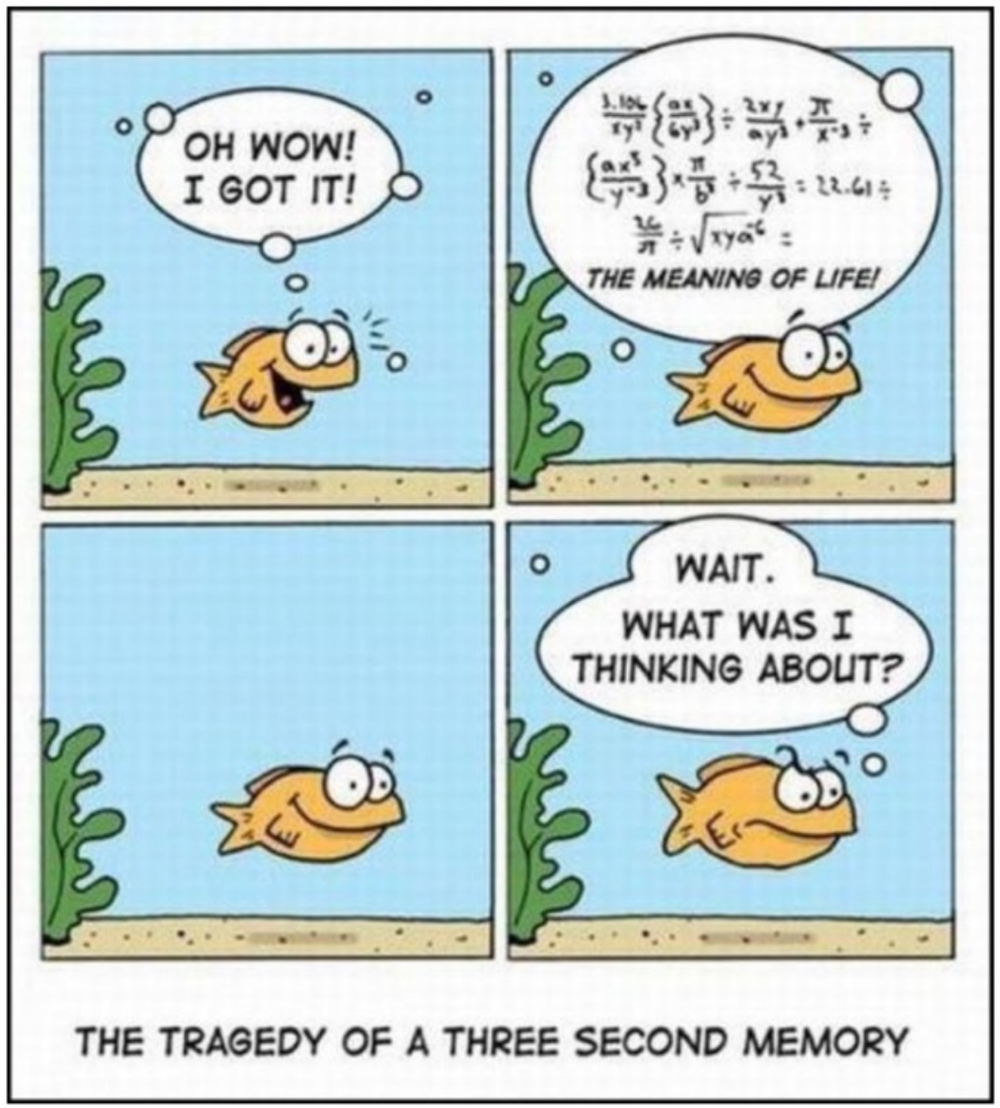
# 1. RNN

**Последствия**

1. Затухание градиентов.

Сеть не будет улавливать дальние зависимости между элементами посл-ти.
Так как градиент $\partial h_t / \partial h_k$ маленький,

то он не вносит вклад в обновление весов модели.
В итоге сеть не сможет выучить,
что нужно смотреть далеко назад.



# 1. RNN

**Последствия**

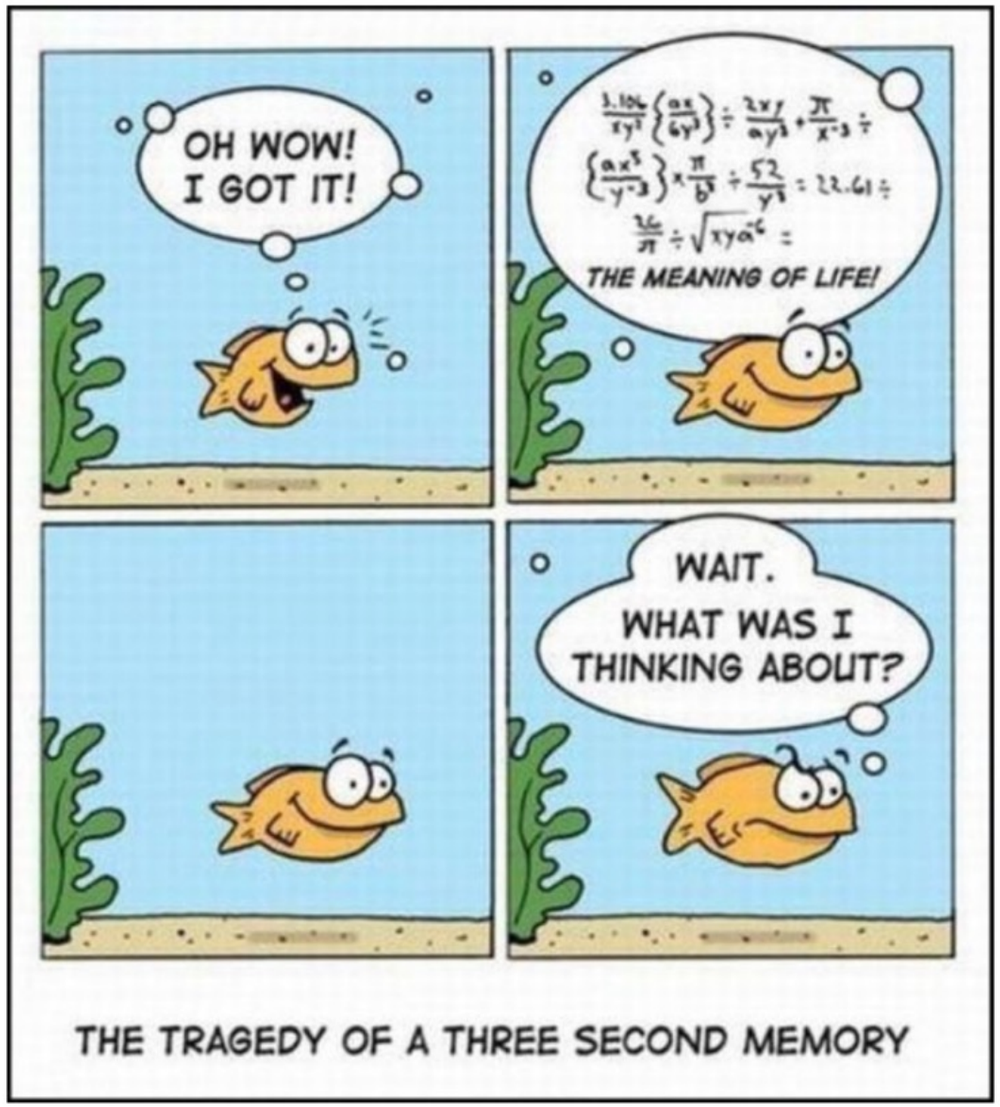
1. Затухание градиентов.

Сеть не будет улавливать дальние зависимости между элементами посл-ти.
Так как градиент $\partial h_t / \partial h_k$ маленький,

то он не вносит вклад в обновление весов модели.
В итоге сеть не сможет выучить,
что нужно смотреть далеко назад.

2. Взрыв градиентов.

Сеть прекратит учиться.



# 1. RNN

**Взрыв градиентов.**

Решение:

# 1. RNN

**Взрыв градиентов.**

Решение: **Gradient clipping**

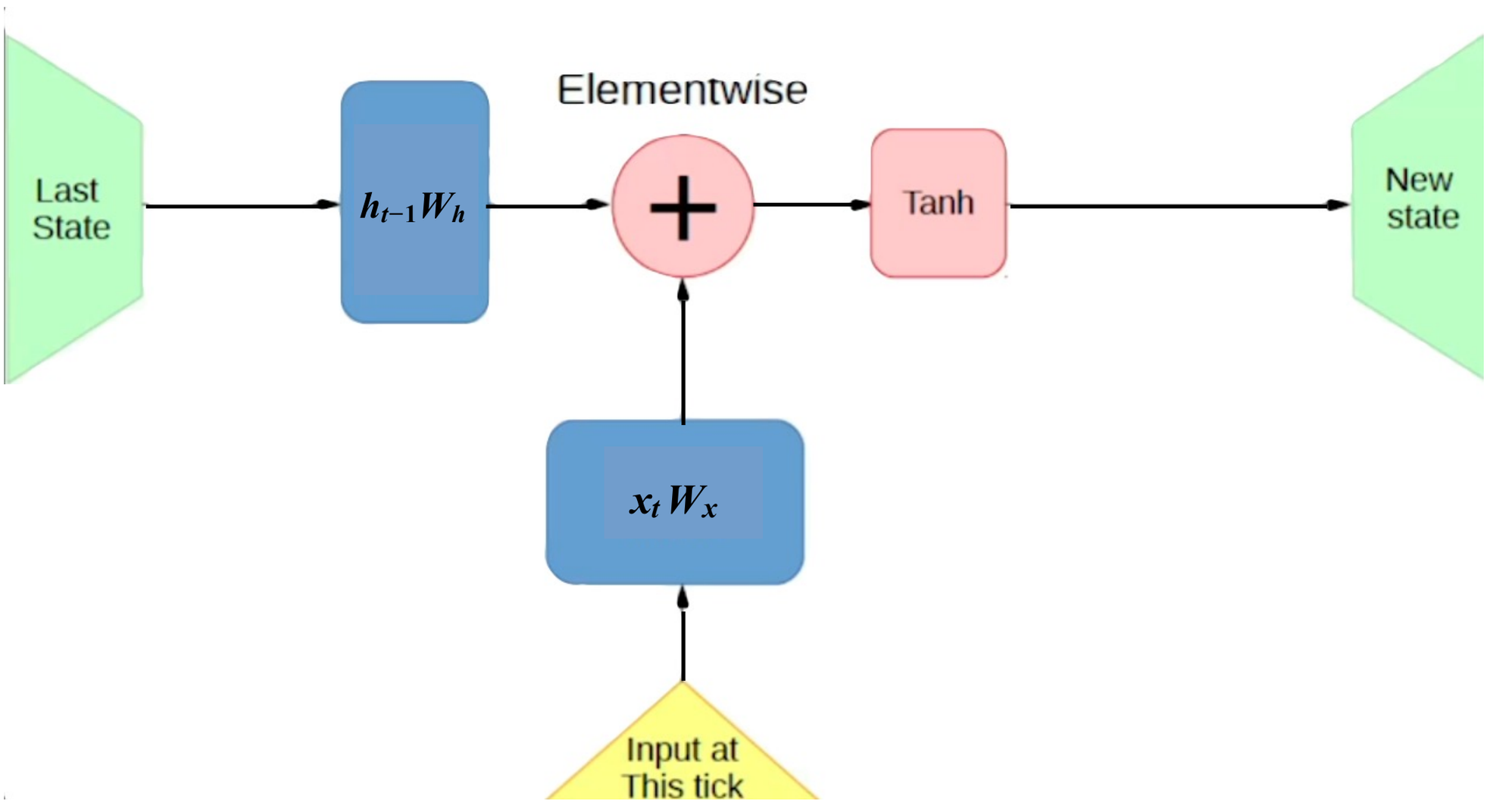
$$\| G \| > threshold \Rightarrow G = \frac{threshold \cdot G}{\| G \|}$$

threshold — гиперпараметр.
Уменьшаем threshold пока взрыв градиентов не прекратится.

# 1. RNN

**Затухание градиентов: Идея**

Как выглядела одна ячейка ранее?

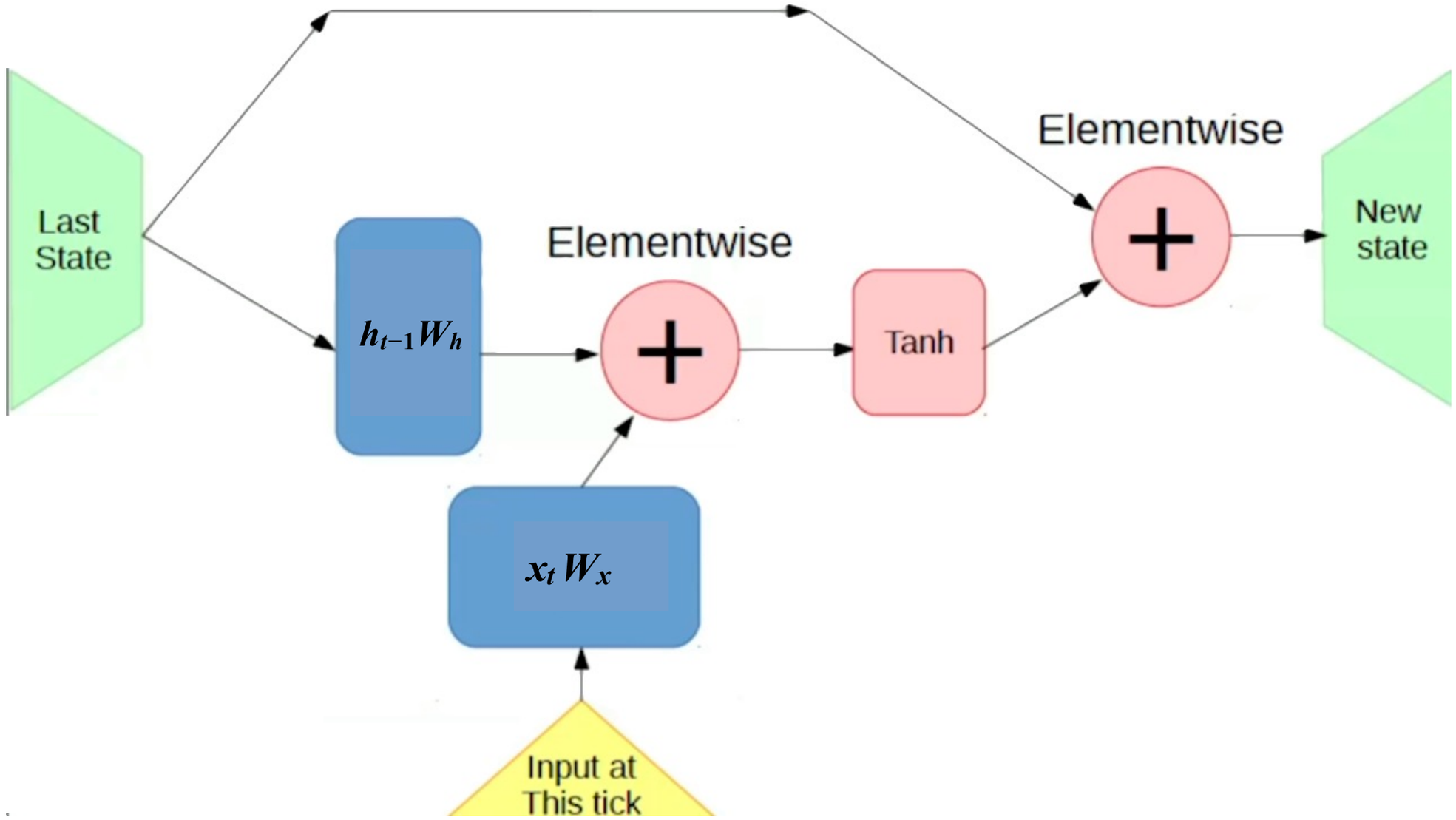


# 1. RNN

**Затухание градиентов: Идея**

Добавим skip-connection.

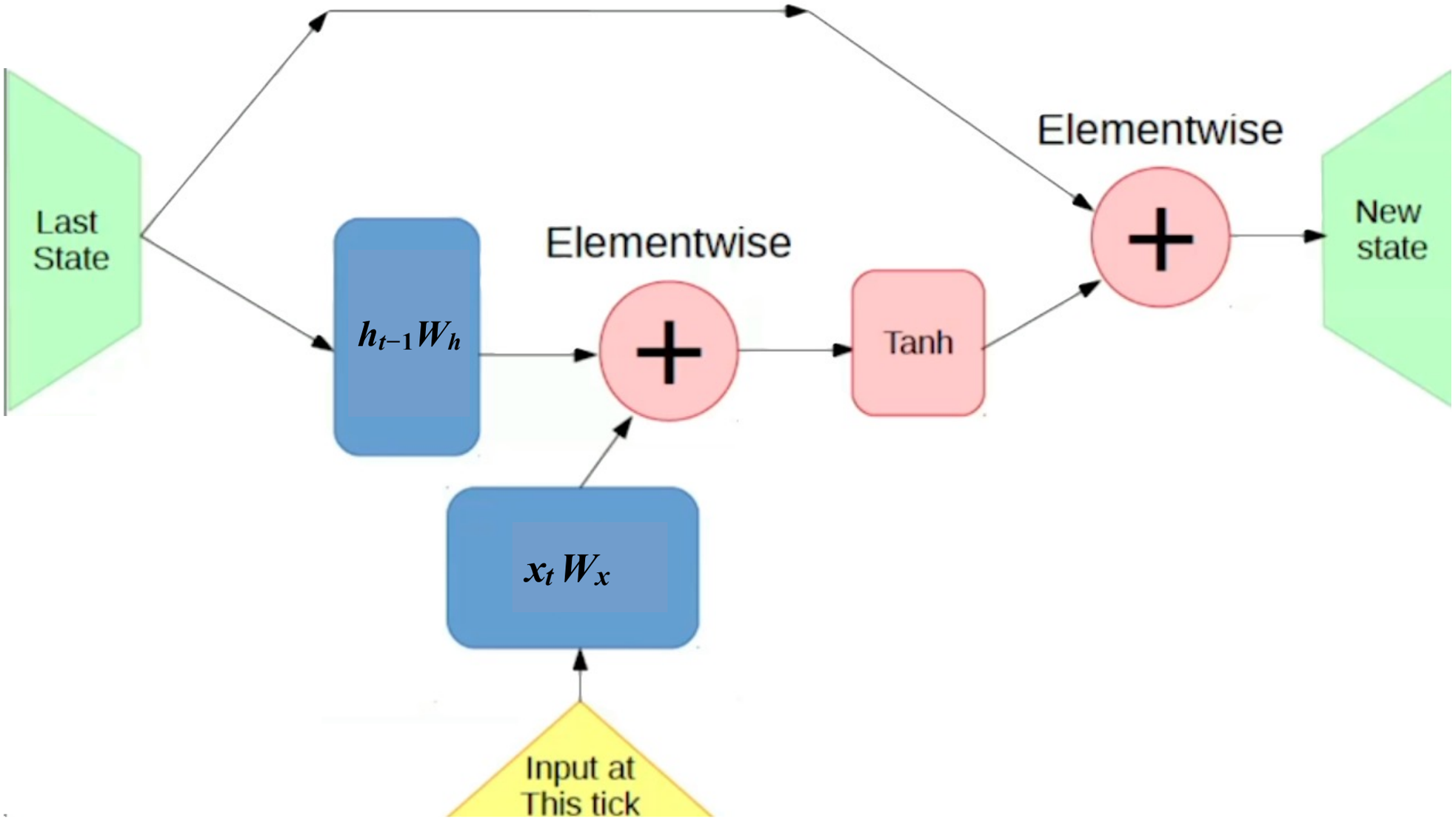
Тогда градиенты не будут затухать.



# 1. RNN

**Затухание градиентов: Идея**

Добавим skip-connection.
Тогда градиенты не будут затухать.

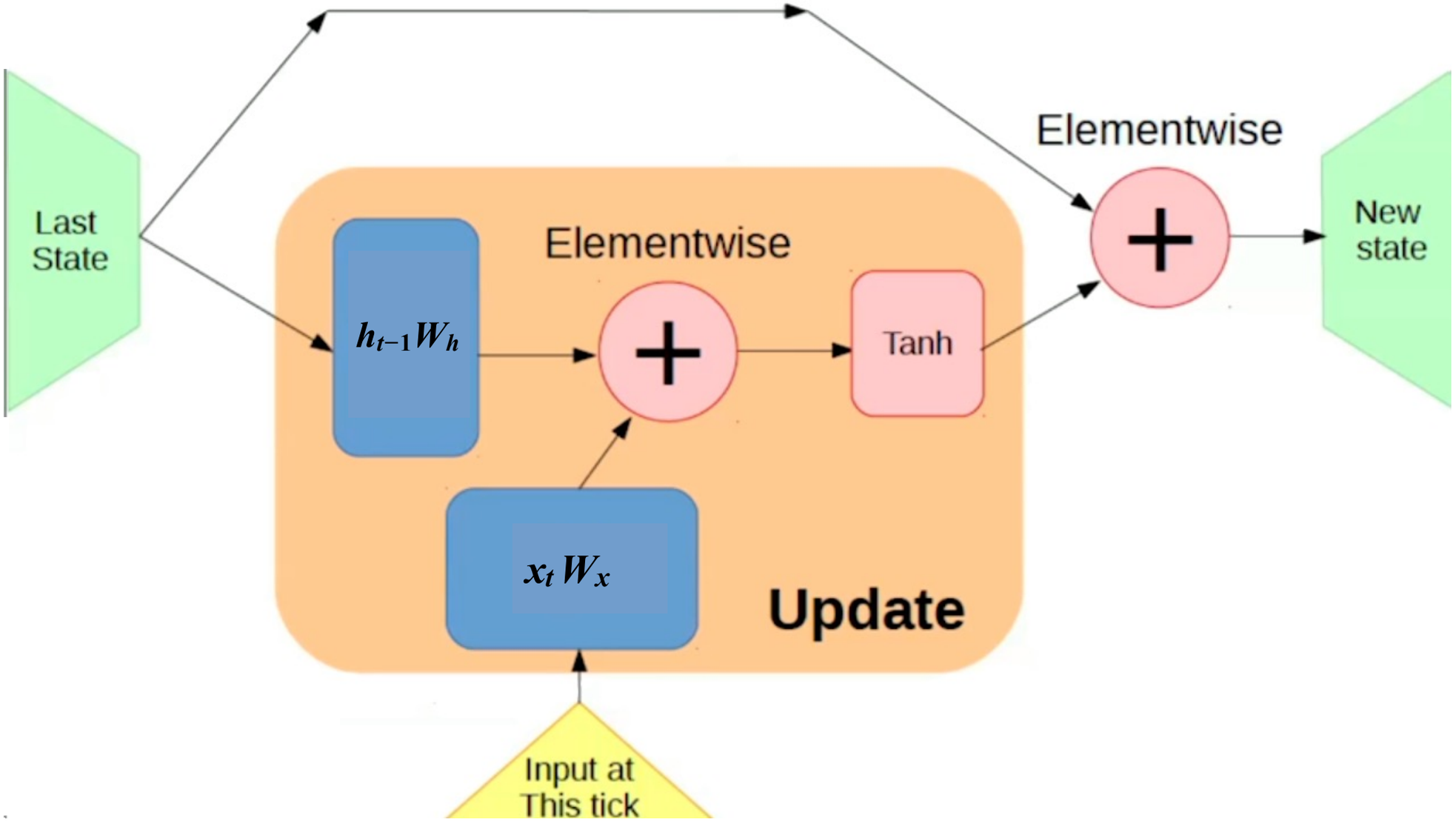



Однако такая сеть менее полезна.
Ранее сеть могла что-то занулить или прибавить много новой информации.
Сейчас же такое сделать очень сложно.

# 1. RNN

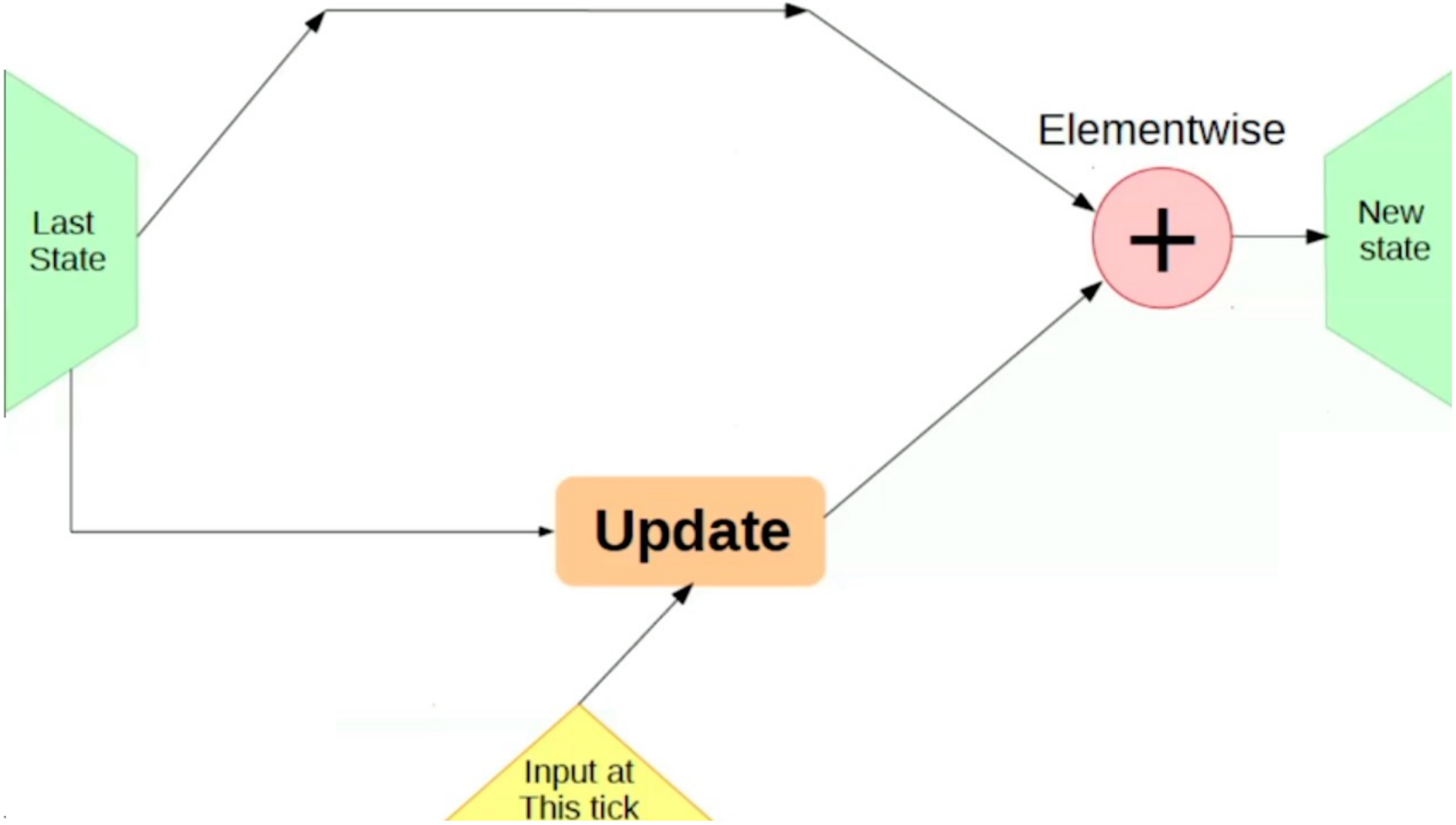
**Затухание градиентов: Идея**

Пусть *update* — преобразование,
применяемое к предыдущему состоянию и текущему входу.



# 1. RNN

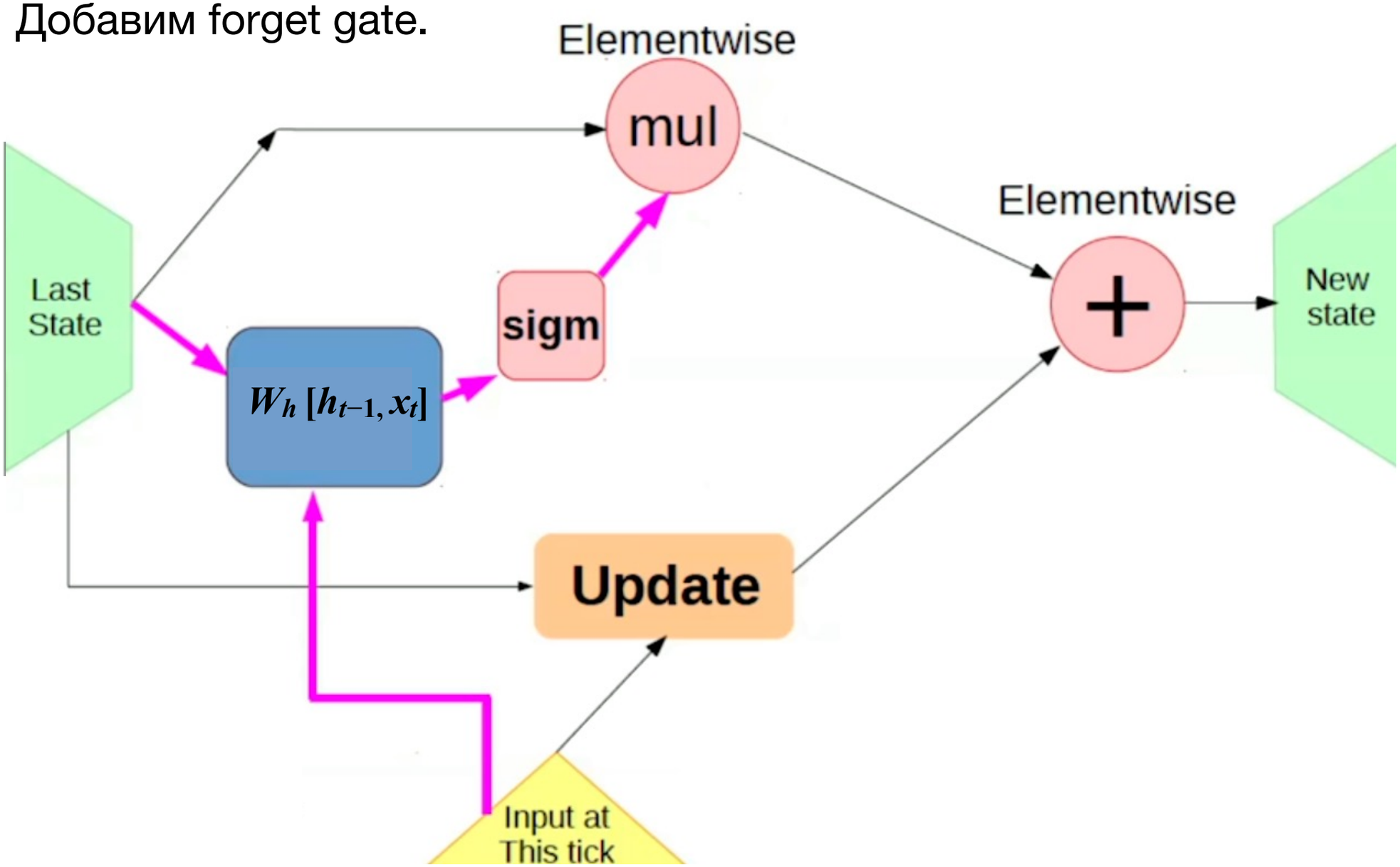
## Затухание градиентов: Идея



# 1. RNN

**Затухание градиентов: Идея**

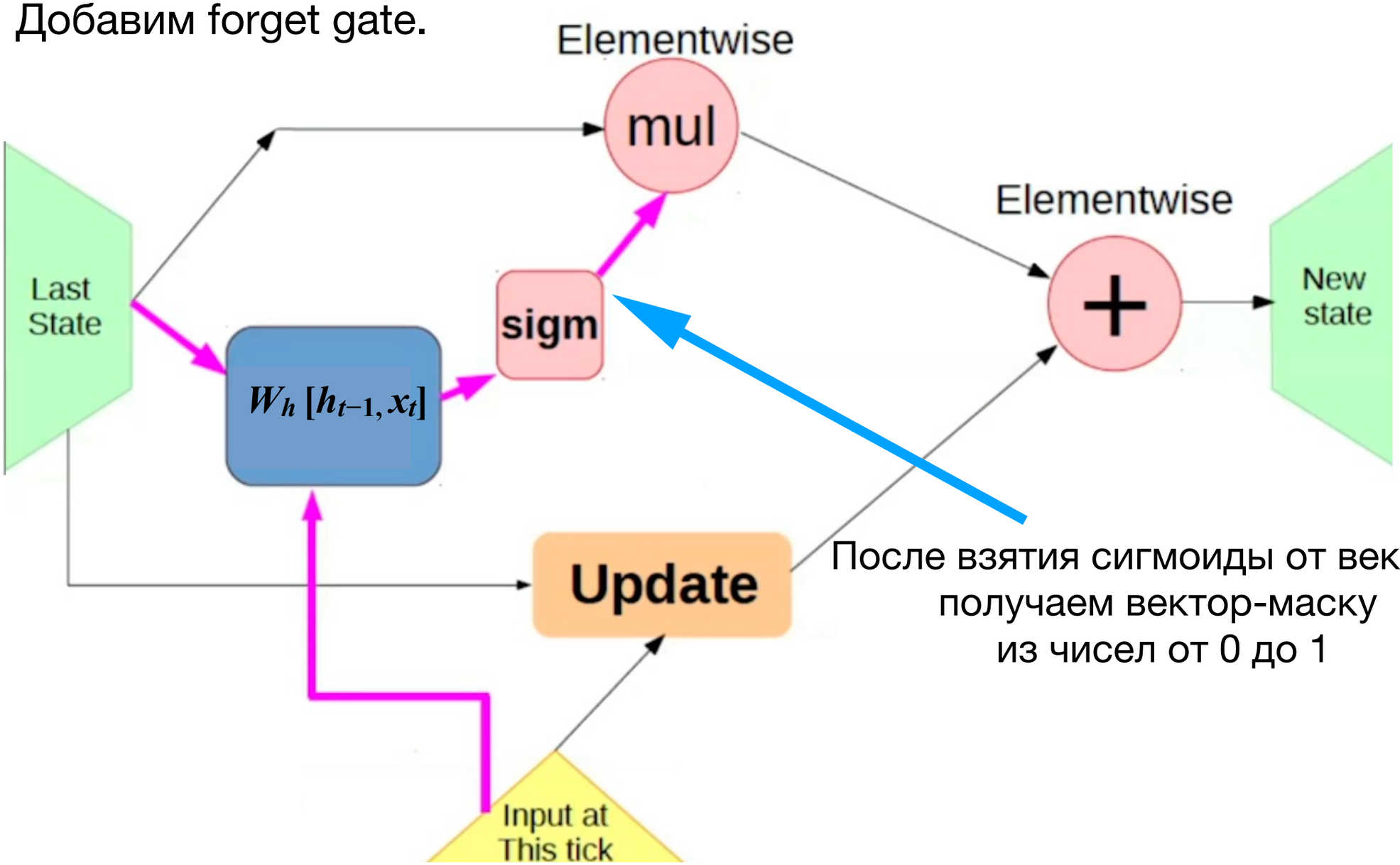
Добавим forget gate.



# 1. RNN

**Затухание градиентов: Идея**

Добавим forget gate.



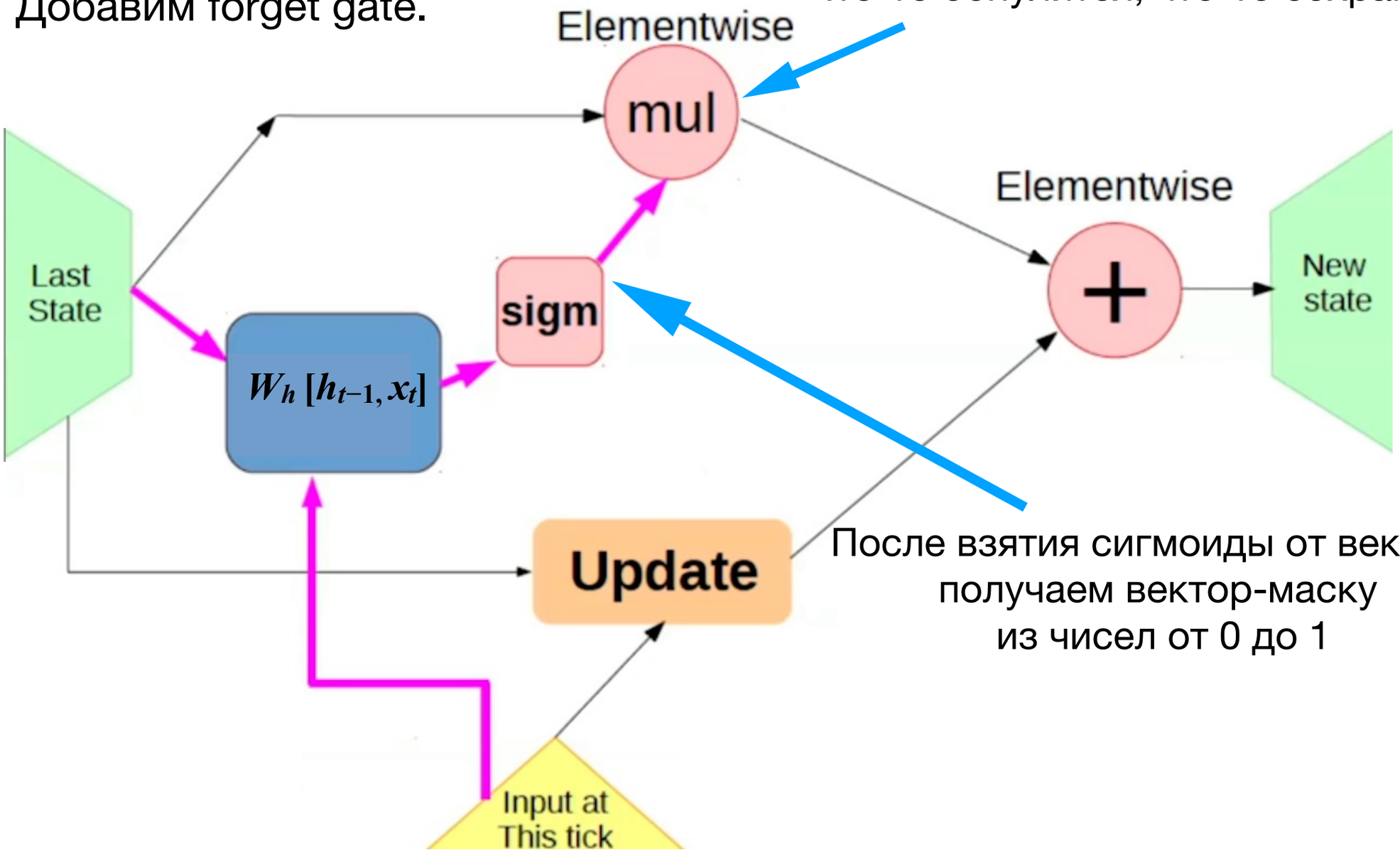

После взятия сигмоиды от вектора
получаем вектор-маску
из чисел от 0 до 1

# 1. RNN

**Затухание градиентов: Идея**

Добавим forget gate.

Поэлементно умножаем предыдущее состояние на вектор-маску. Что-то обнулится, что-то сохранится.

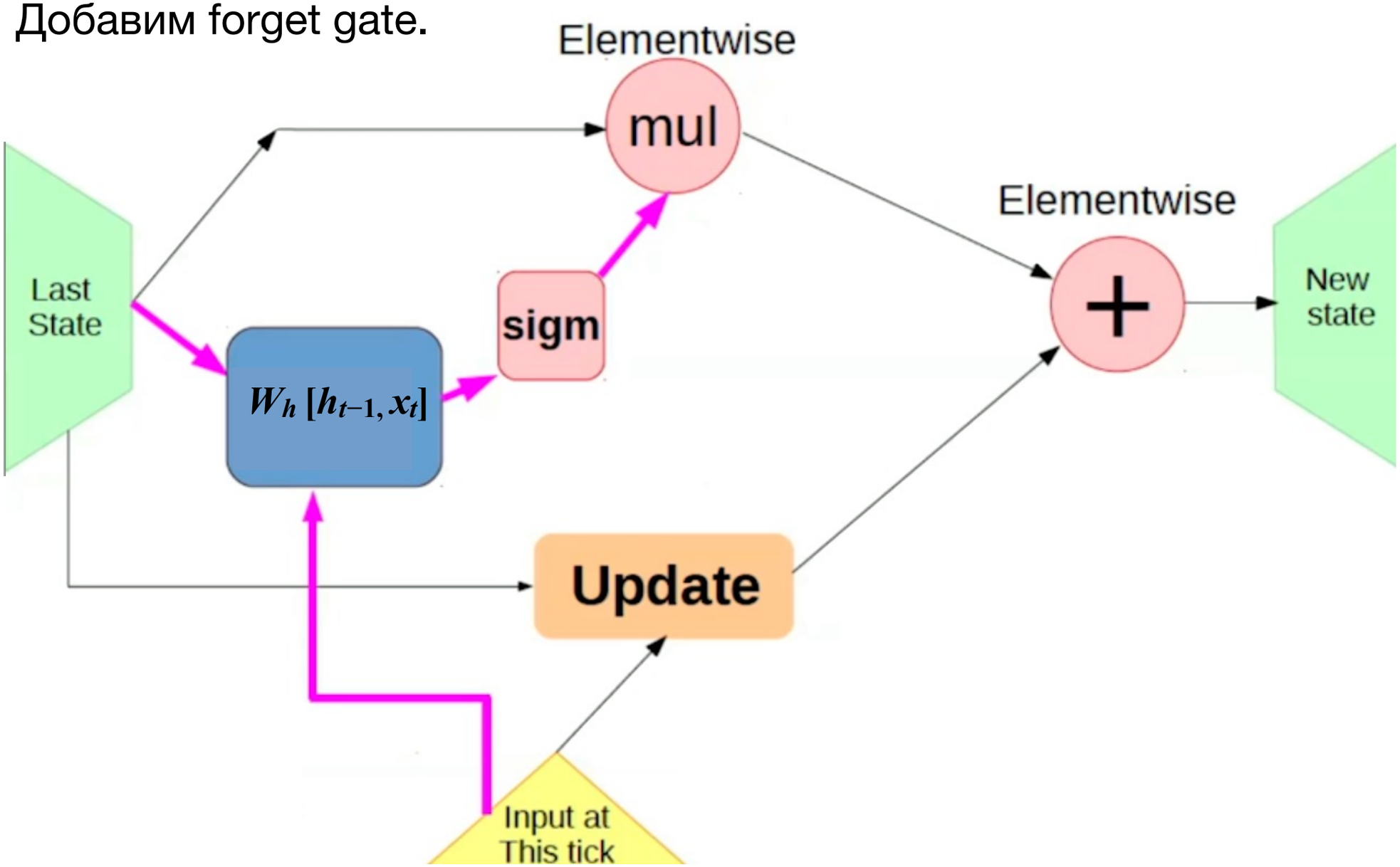



После взятия сигмоиды от вектора получаем вектор-маску из чисел от 0 до 1

# 1. RNN

**Затухание градиентов: Идея**

Добавим forget gate.




В *update* решаем, что добавить из новой информации.

В *forget* решаем, что забыть и что сохранить из предыдущего состояния.

# RNN

1. Vanila RNN

2. LSTM

3. GRU

4. Bi-RNN и Deep RNN

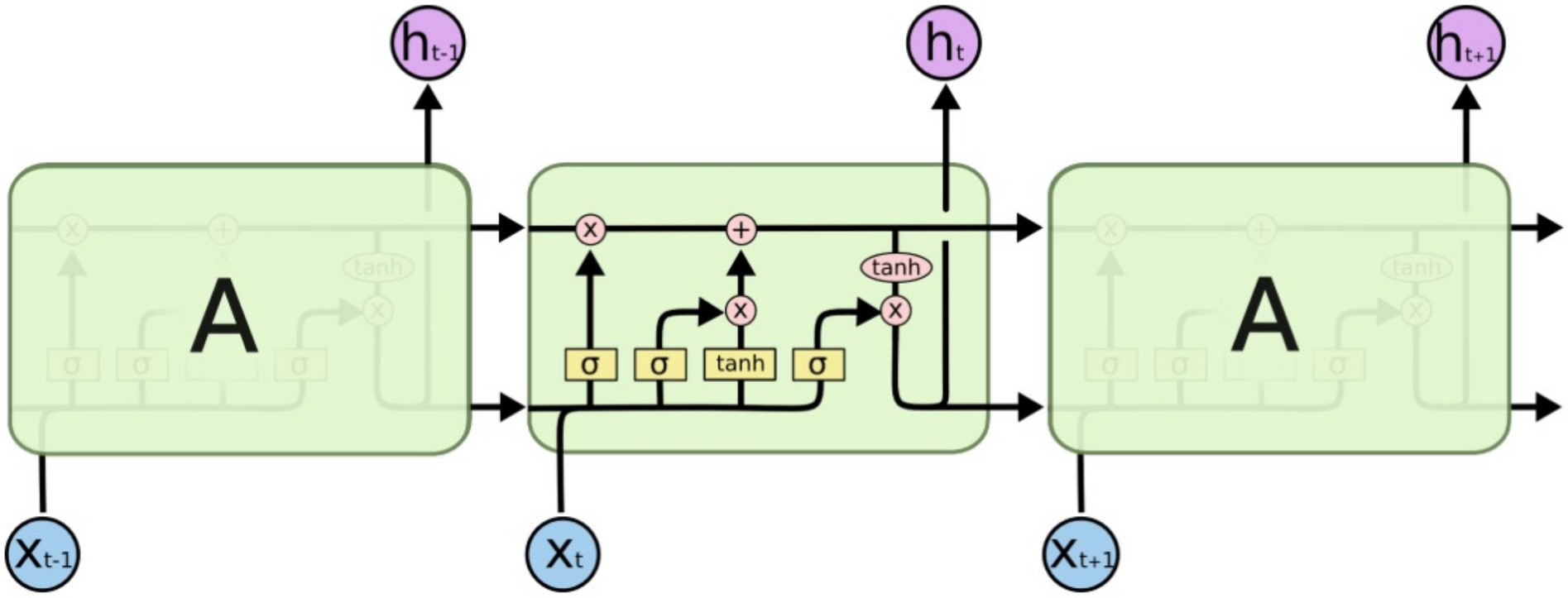
# 2. LSTM

**Long Short Term Memory**

LSTM имеет уже 2 скрытых состояния:

- cell state (приватное состояние)
- output state (публичное состояние)

Cell state — внутреннее состояние системы.
Output state — состояние, возвращаемое на выходе.



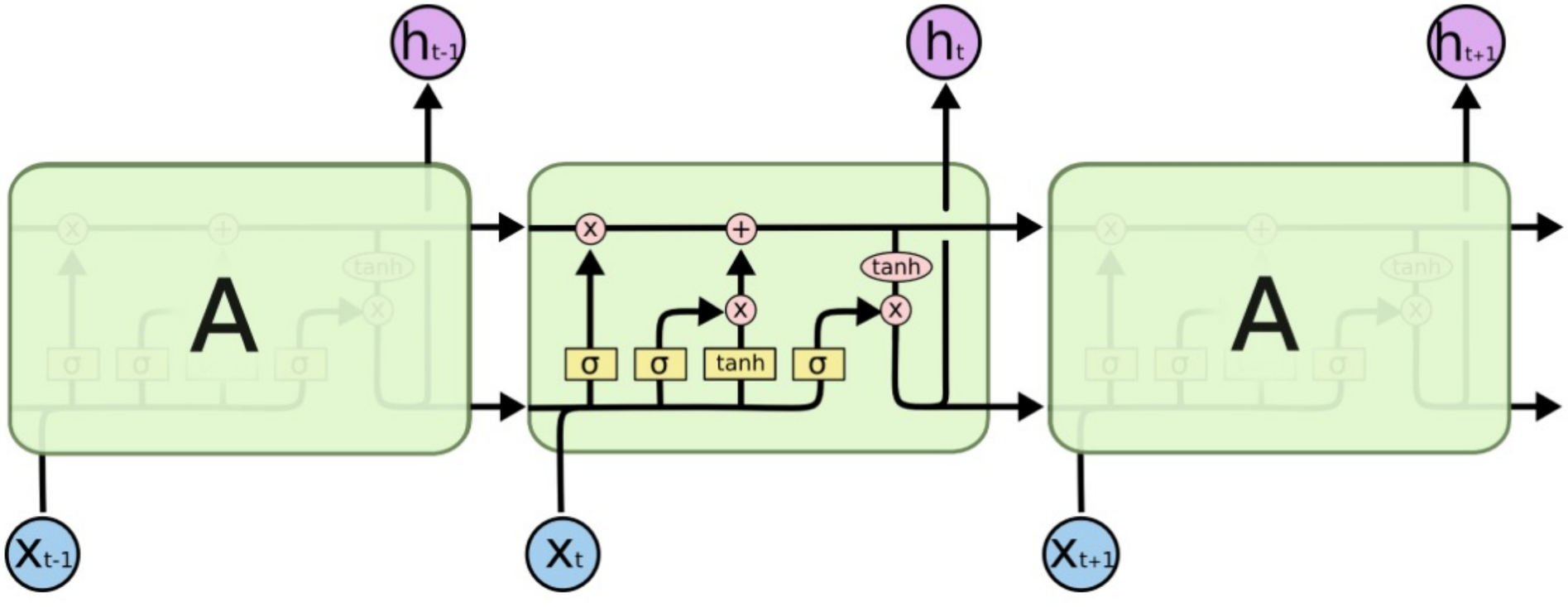
# 2. LSTM

**Long Short Term Memory**

LSTM имеет уже 2 скрытых состояния:
- cell state (приватное состояние)
- output state (публичное состояние)

Cell state — внутреннее состояние системы.
Output state — состояние, возвращаемое на выходе.

LSTM содержит в себе 4 блока:
- Update
- Forget gate
- Input gate
- Output gate



# 2. LSTM

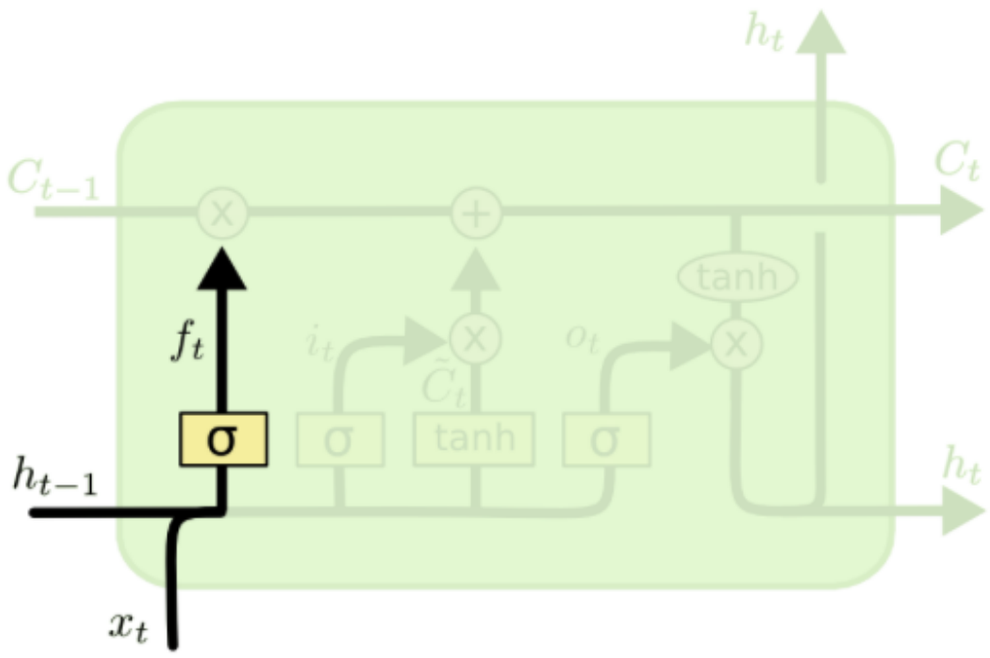
**Forget gate**

Решаем какую информацию из предыдущего cell state $C_{t-1}$ удалить и какую сохранить.

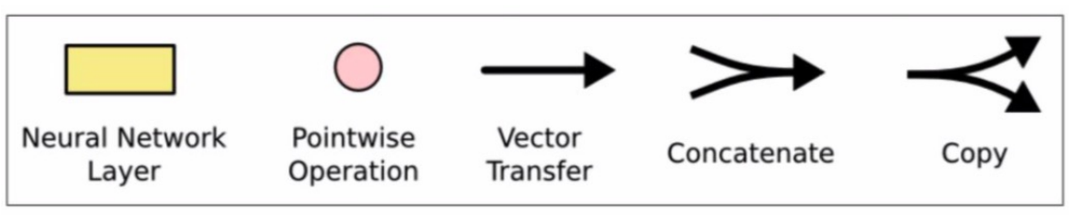
$f_t$ — вектор-маска из чисел от 0 до 1.

$C_{t-1}$ потом поэлементно домножается на вектор-маску.

Так оставляем только нужную информацию из $C_{t-1}$.




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



# 2. LSTM

**Input gate + Update**

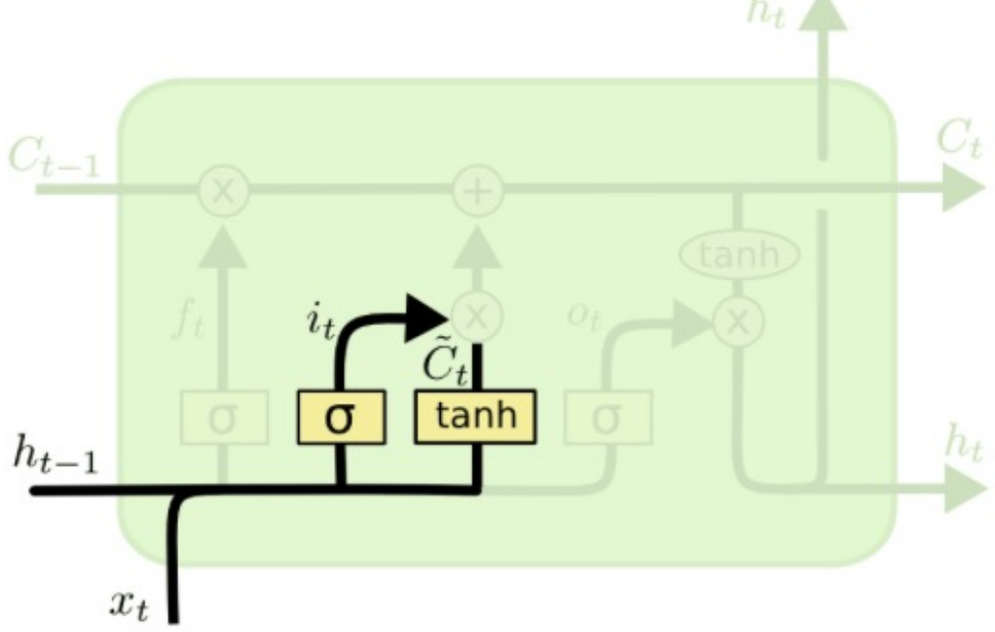
*Input gate* решает какую новую информацию добавить от текущего элемента.

$i_t$ — вектор-маска из чисел от 0 до 1, показывает какую информацию взять.

*Update gate* вычисляет новую информацию.

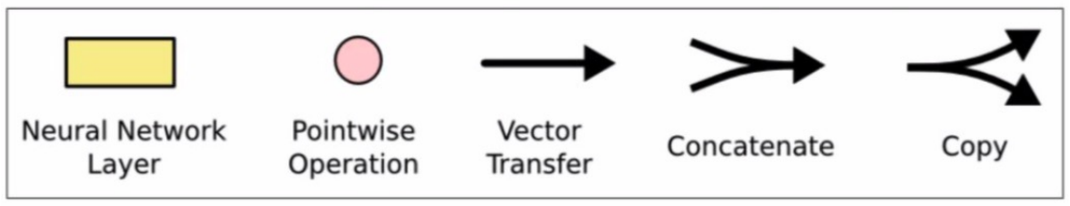
$\tilde{C}_t$ — новая информация, полученная из $h_{t-1}$ и $x_t$.

$C_t$ поэлементно домножается на вектор-маску $i_t$ и прибавляется к $C_{t-1}$.




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

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



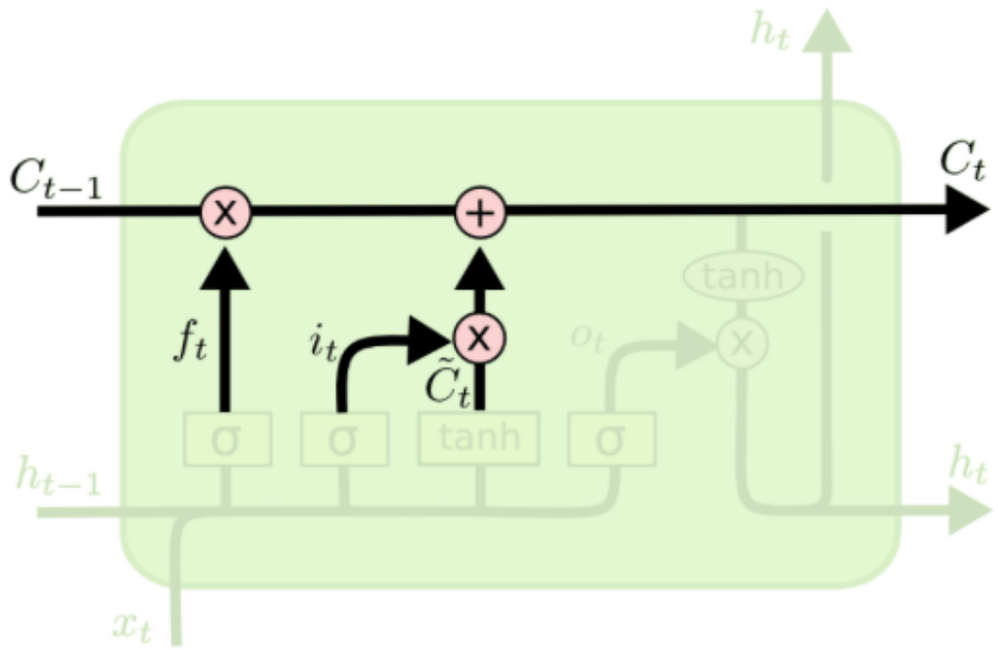
# 2. LSTM

**Обновляем cell state.**

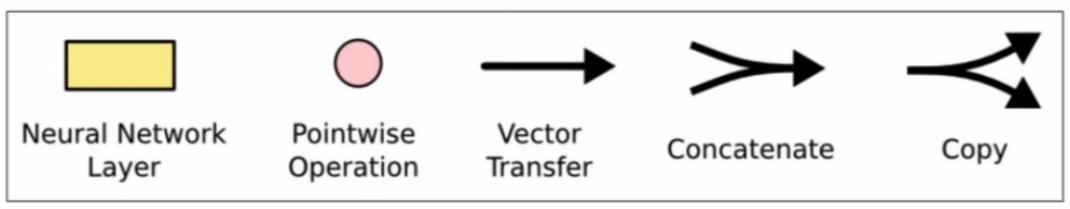
Фильтруем старое состояние $C_{t-1}$ и добавляем к нему новую информацию.

*Фильтрация*: $C_{t-1}$ домножается на вектор-маску $f_t$.

*Добавление информации*: вектор новой информации $\tilde{C}_t$ домножается на вектор-маску $i_t$ и прибавляется к $C_{t-1}$.




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



# 2. LSTM

**Output gate**

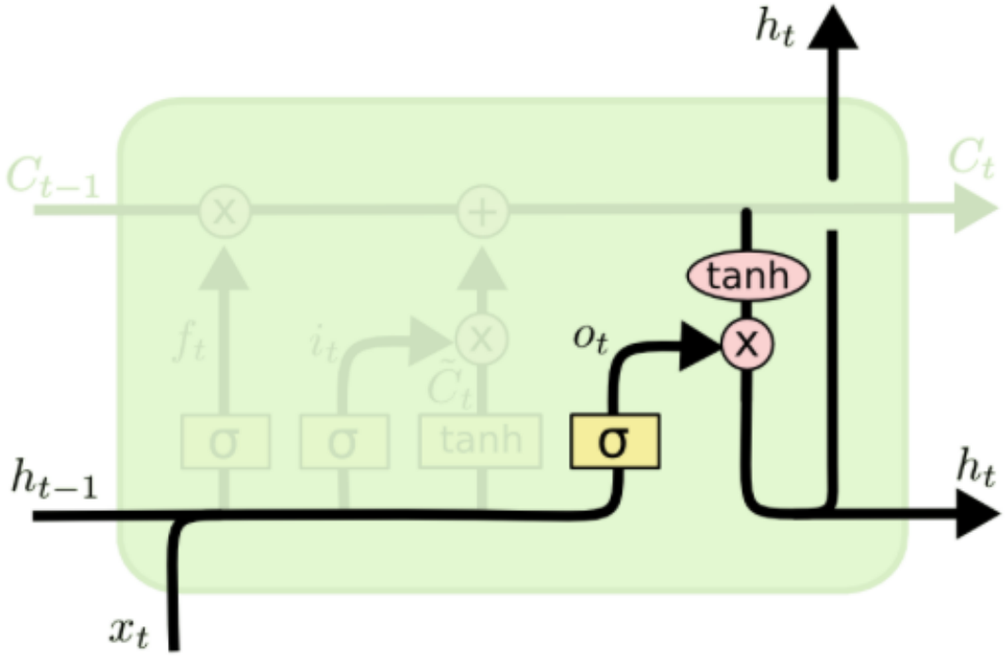
*Output gate* решает какую часть $C_t$ сделать публичным состоянием $h_t$

и выдать наружу, а какую нет.

$o_t$ — вектор-маска из чисел от 0 до 1 для фильтрации $C_t$.

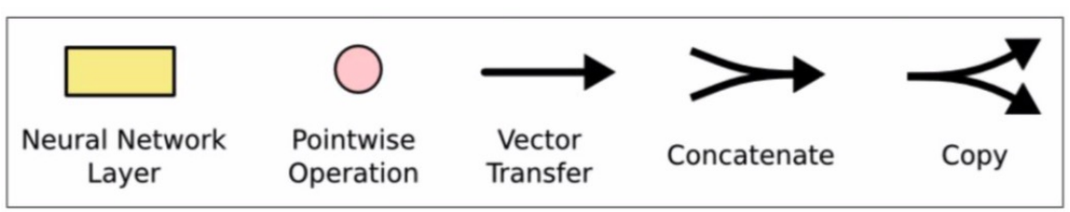
$\tanh(C_t)$ домножается на вектор-маску $o_t$ и полученное становится новым $h_t$.

То есть в качестве $h_t$ берется некоторая часть $C_t$.



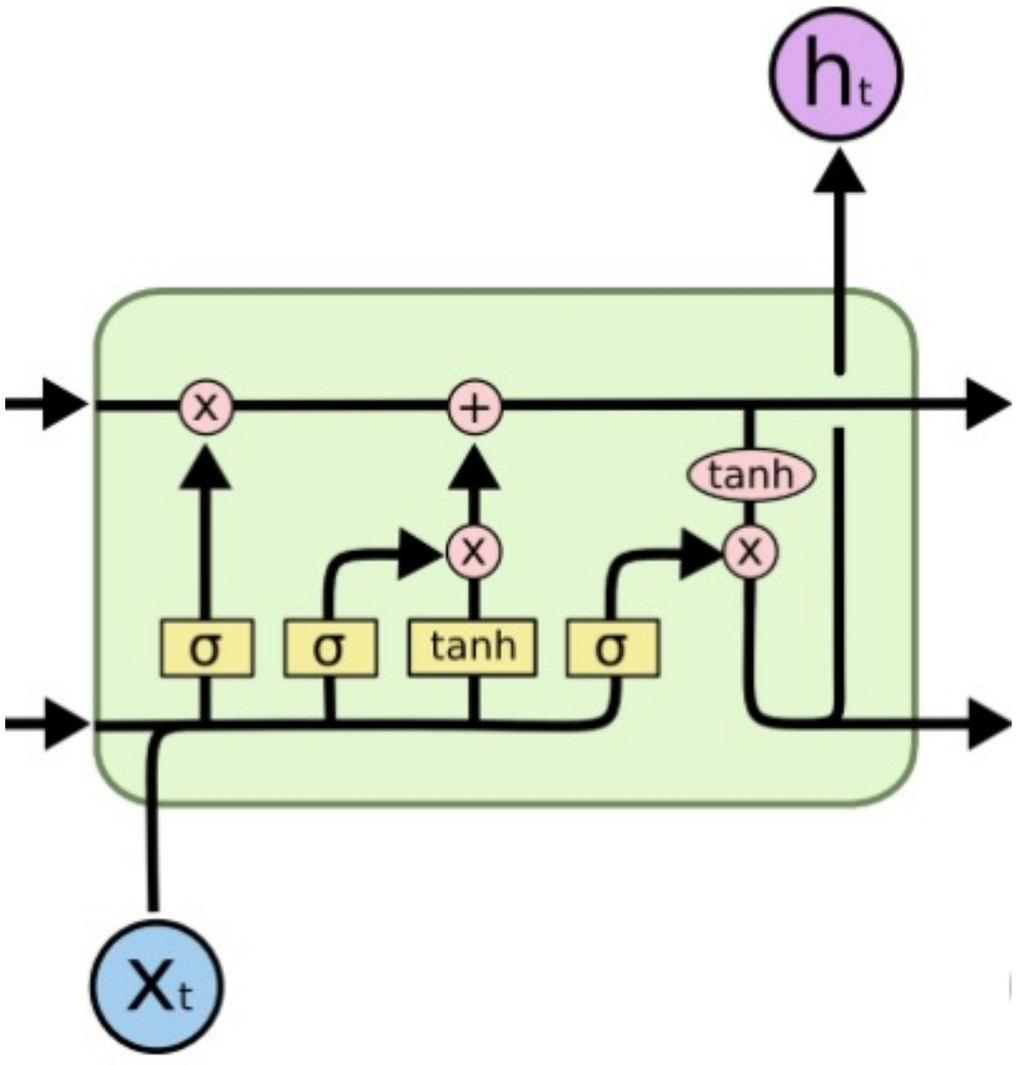

$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

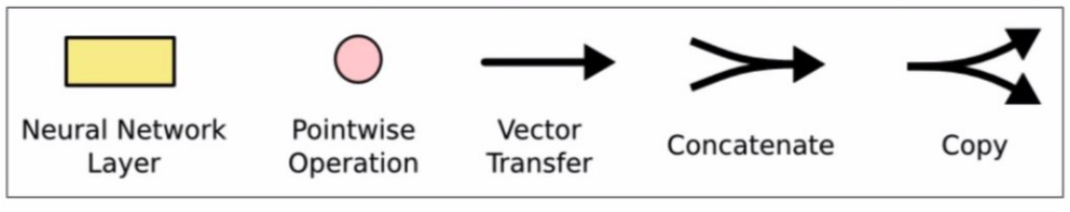
$$h_t = o_t * \tanh\left(C_t\right)$$



# 2. LSTM

**Все формулы**




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

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

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

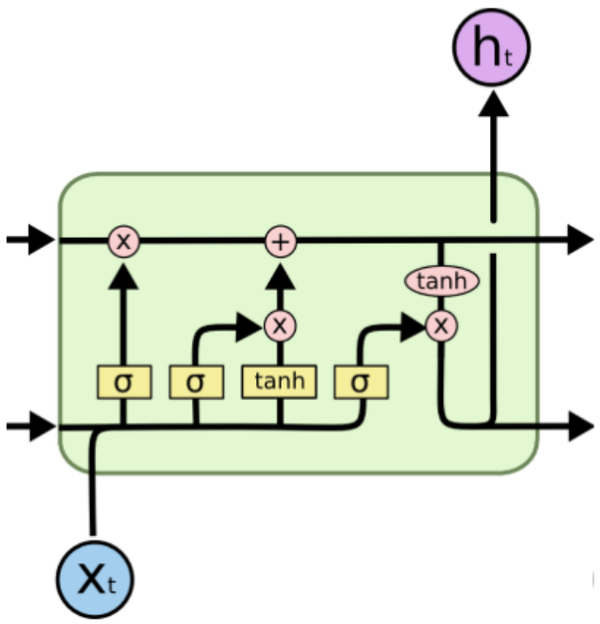
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

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

$$h_t = o_t * \tanh\left(C_t\right)$$

# 2. LSTM

**Все формулы**




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

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

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

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

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

$$h_t = o_t * \tanh\left(C_t\right)$$

or

Original version:

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

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

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

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

$$h_t = o_t * \tanh\left(C_t\right)$$

В оригинальной версии не было forget gate $f_t$ позволяющего удалить некоторую информацию из предыдущего cell state $C_{t-1}$.

# 2. LSTM

***Почему это решает проблему затухания градиентов?***

*Vanilla RNN*

$$h_t = tanh(h_{t-1}W_h + x_tW_x).$$

$$\frac{\partial h_i}{\partial h_{i-1}} = tanh'(h_{i-1}W_h + x_iW_x) \cdot W_h \Rightarrow \frac{\partial h_t}{\partial h_k} = \prod_{i=k+1}^{t} tanh'(h_{i-1}W_h + x_iW_x) \cdot W_h^t$$

Из-за $W_h^{t-k}$ градиент может занулиться.

*В LSTM*

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

Градиент $\frac{\partial C_t}{\partial C_{t-1}} = f_t + C_{t-1}\frac{\partial f_t}{\partial C_{t-1}} + \tilde{C}_t\frac{\partial i_t}{\partial C_{t-1}} + i_t\frac{\partial \tilde{C}_t}{\partial C_{t-1}}$.

Градиент состоит из четырех слагаемых, поэтому он редко близок к 0.

Градиент будет доходить до ранних входов.

⇒ Модель сможет уловить длинные зависимости при обучении
и при тестировании сможет обладать долгой памятью и смотреть далеко назад.

# RNN

1. Vanila RNN

2. LSTM

3. GRU

4. Bi-RNN и Deep RNN

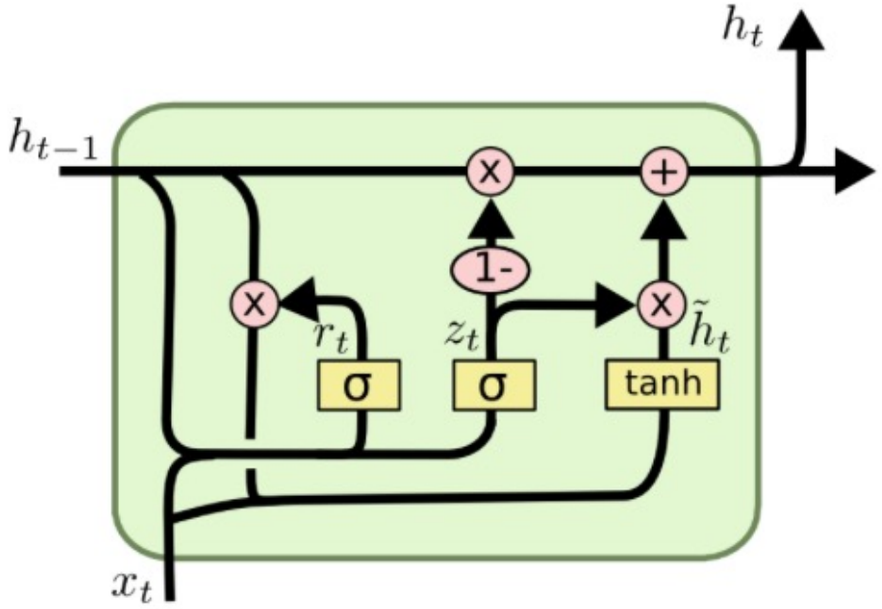
# 3. GRU

**Gated Recurrent Unit**

- Похожа на LSTM.

Тоже имеет структуру из gates.

- Содержит только одно скрытое состояние.
- Является более легковесной версией LSTM.




$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

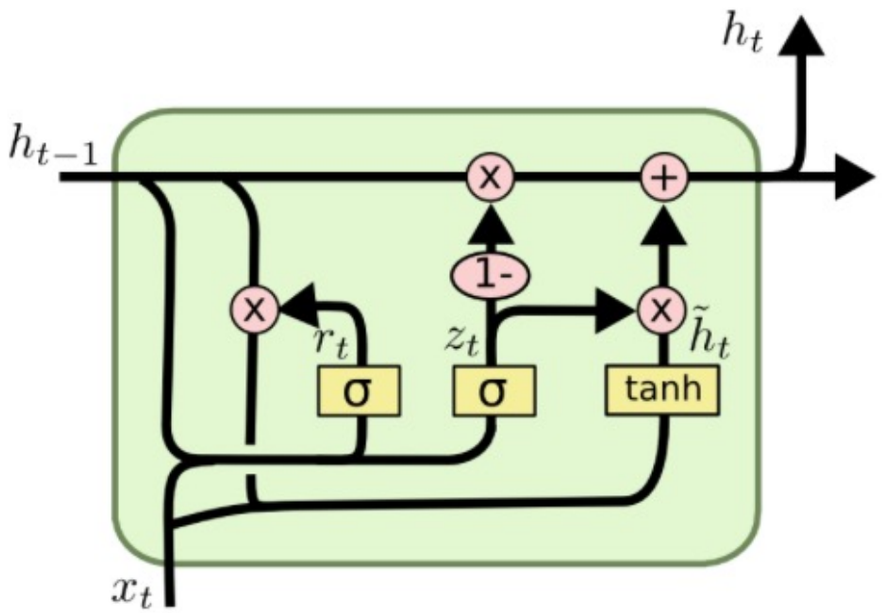
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

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

# 3. GRU

## Gated Recurrent Unit




$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

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

- Update gate $z_t$ контролирует какие части $h_{t-1}$ должны измениться.
- Reset gate $r_t$ находит какие части $h_{t-1}$ должны быть использованы

для подсчета вектора новой информации

- Вектор новой информации $h_t$ использует текущий вход $x_t$ и часть $h_{t-1}$,

выбранную с помощью reset gate, для подсчета новой информации.

- Обновление $h_t$ происходит при помощи аддитивной операции,

тем самым помогая справиться с проблемой затухающего градиента.

# 3. LSTM vs GRU

- GRU более вычислительно эффективна и имеет меньше параметров.
- Если в датасете много данных,

  то LSTM обычно показывает чуть более хороший результат.
- Если в датасете мало данных,

  то GRU обычно показывает чуть более хороший результат.
- Нельзя точно сказать, что даст лучший результат.

Сейчас существует и множество других аналогов RNN

# RNN

1. Vanila RNN

2. LSTM

3. GRU

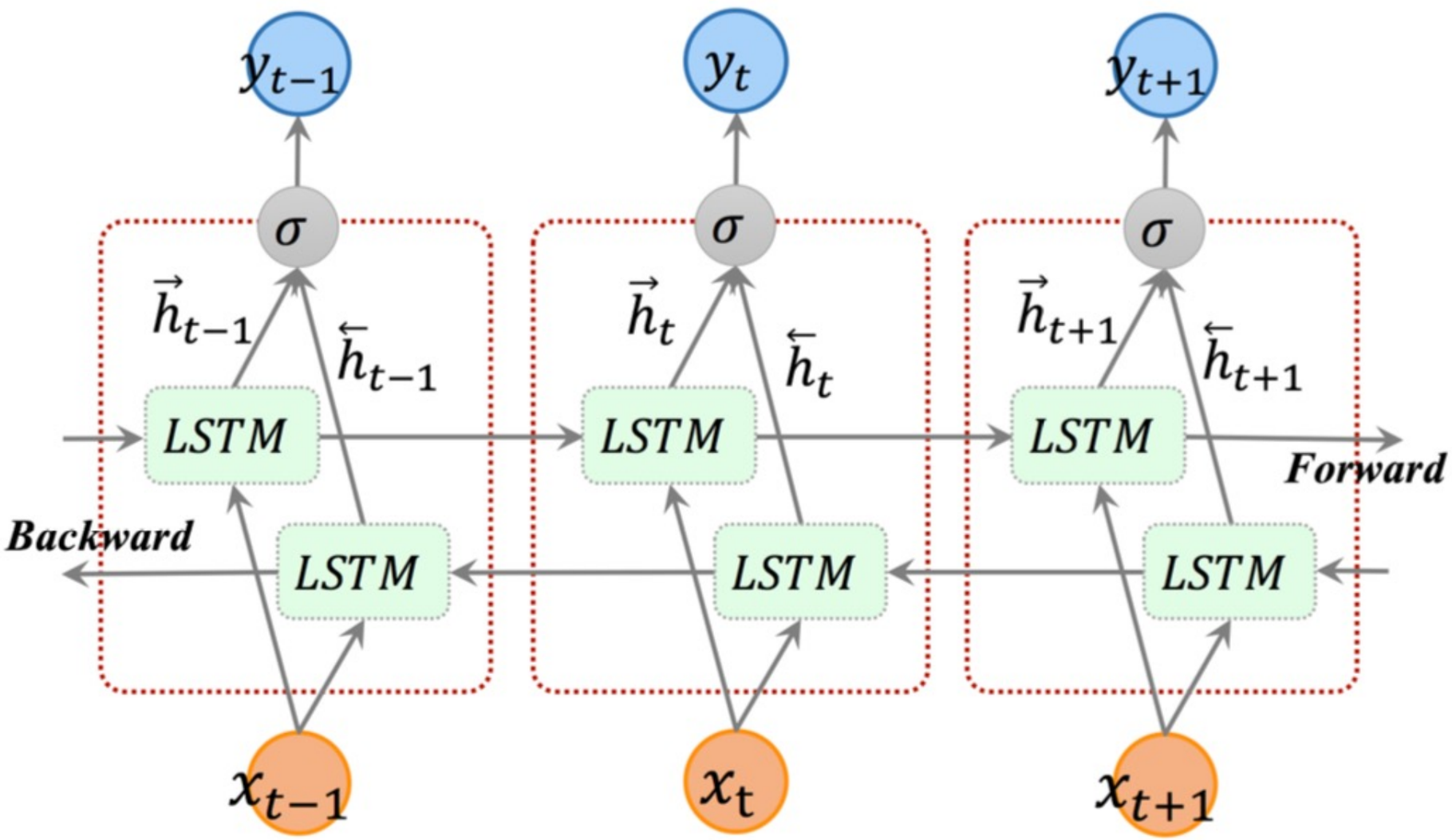
4. Bi-RNN и Deep RNN

# 4. Bi-RNN

Bidirectional RNN позволяют собрать информацию с обоих концов посл-ти.
Будем проходиться RNN по последовательности с двух концов,

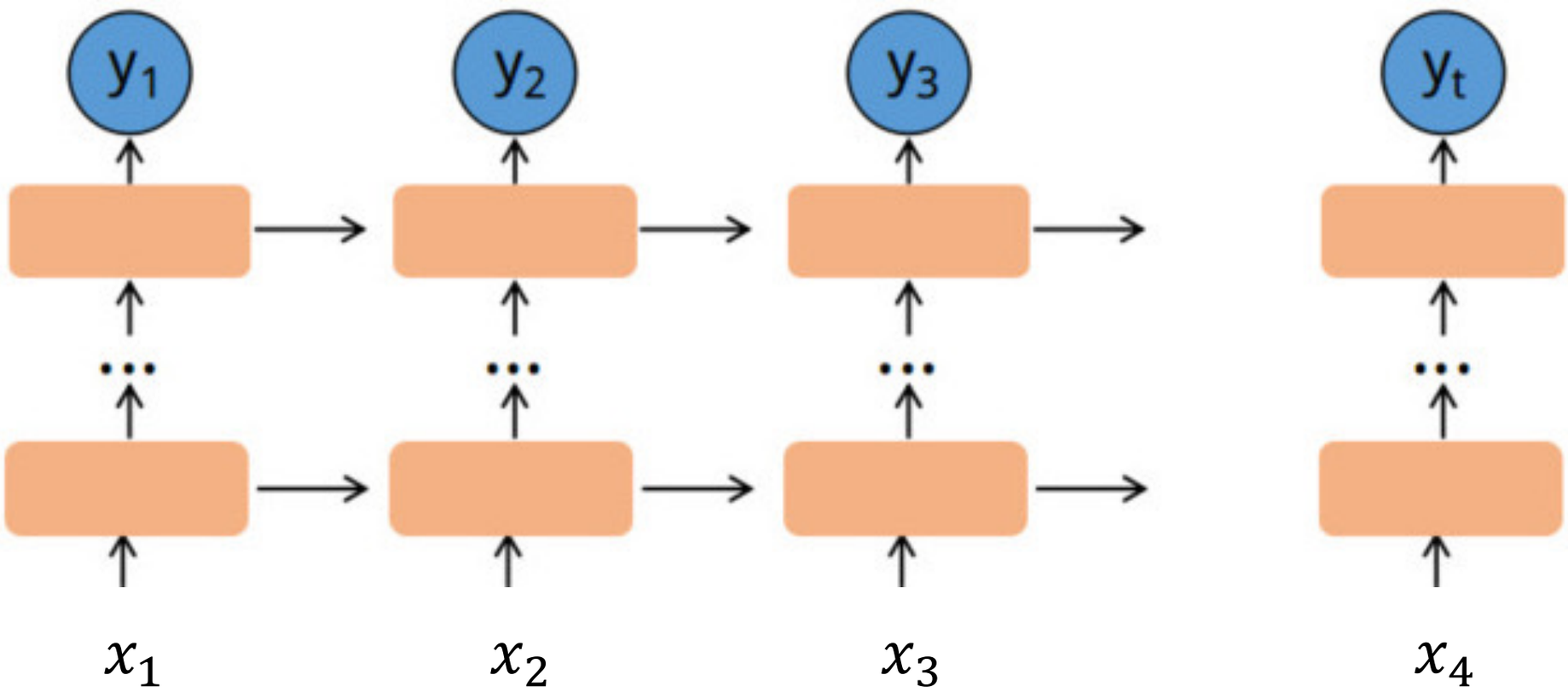
а потом сконкатенируем результат.

$[\overrightarrow{h_t}, \overleftarrow{h_t}]$ содержит информацию как о элементах левее $x_t$, так и правее $x_t$.



# 4. Многослойная RNN

Для улучшения качества RNN можно стакать друг на друга.
Состояния первой RNN подаются на вход второй.
Состояния второй RNN подаются на вход третьей.
И так далее.




Каждая RNN внутри может быть как однонаправленной,
так и двунаправленной.

# ВСЁ!