Лекция 7
Временные ряды и рекуррентные нейронные сети

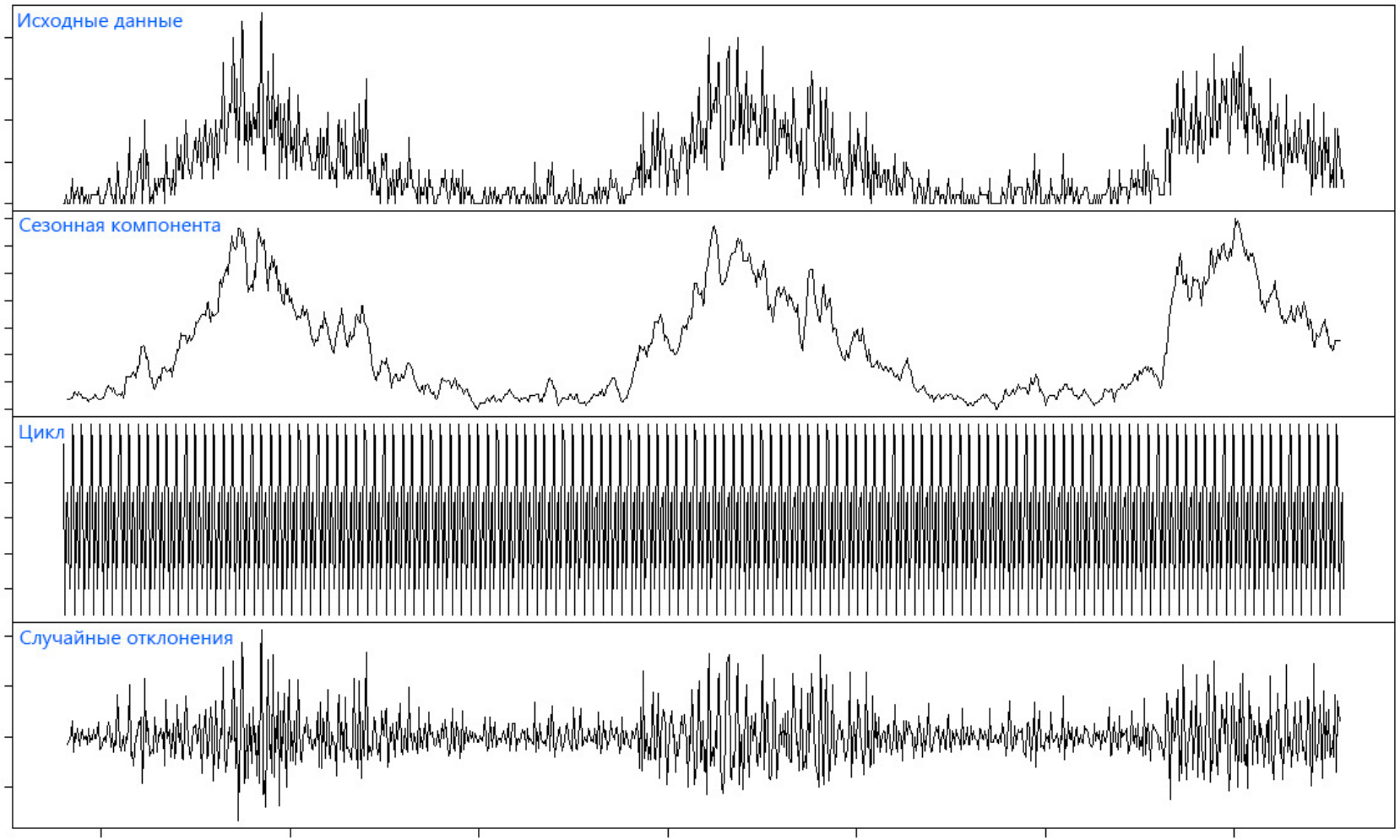
Разработка нейросетевых систем

Канев Антон Игоревич

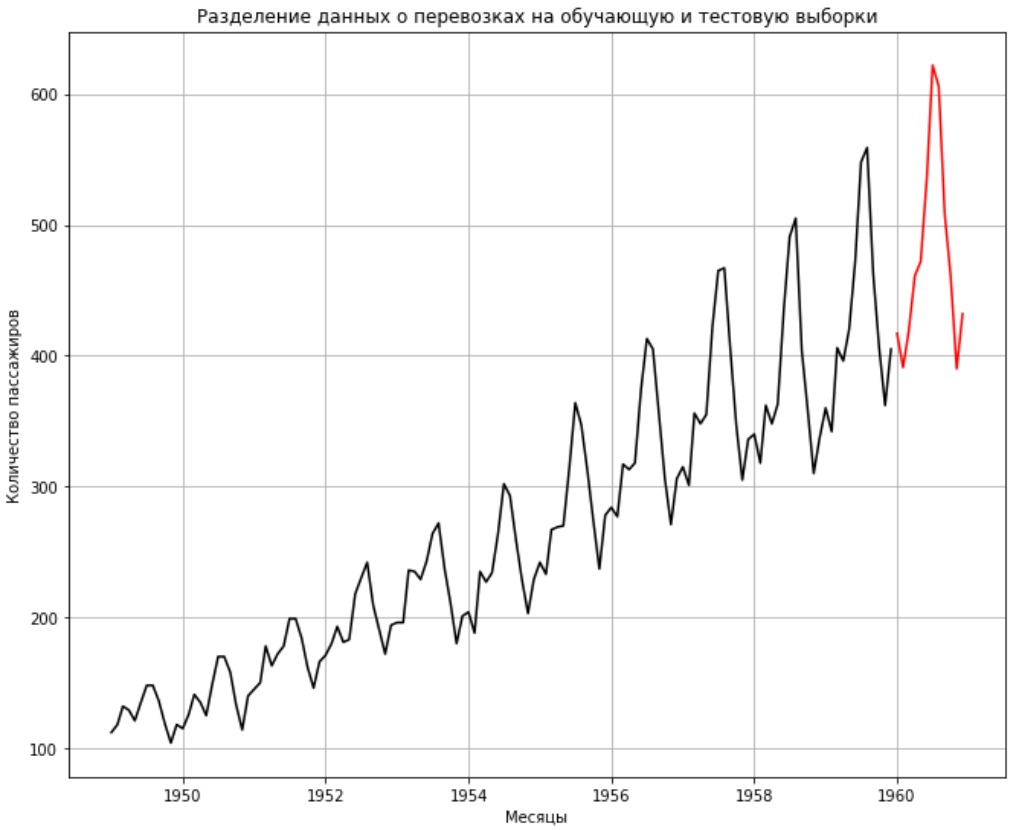
Временной ряд

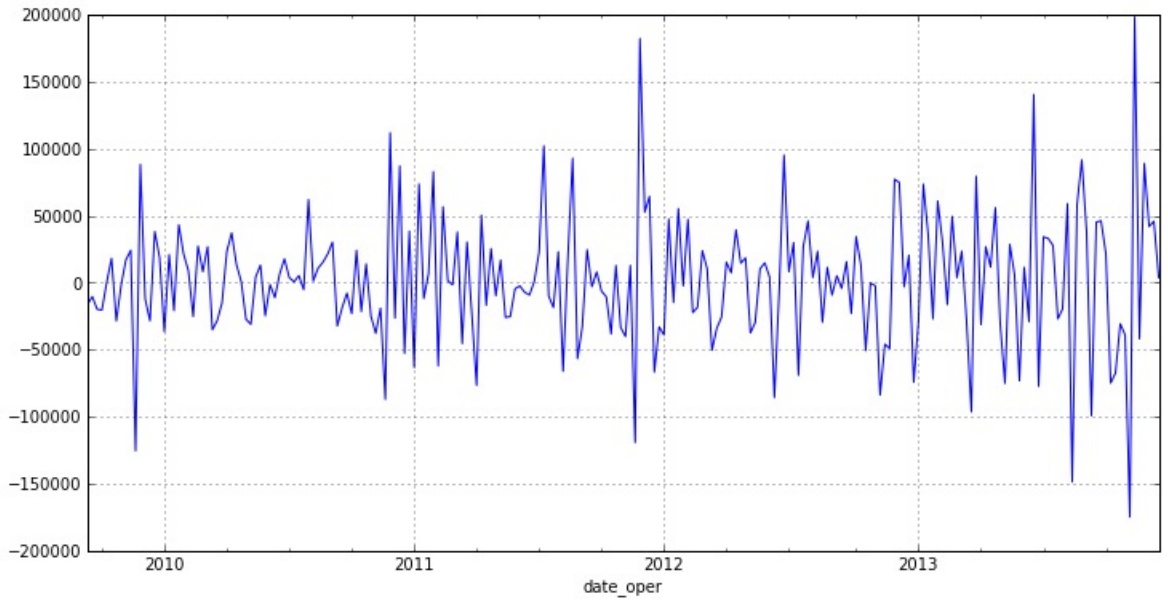
- **Временной ряд** — собранный в разные моменты времени статистический материал о значении каких-либо параметров.
- Временной ряд существенно отличается от простой выборки данных, так как при анализе учитывается взаимосвязь измерений со временем, а не только статистическое разнообразие и статистические характеристики выборки.
- Стационарный временной ряд – ряд, у которого не меняются математическое ожидание и дисперсия со временем.

Преобразование временных рядов



Стационарность временного ряда





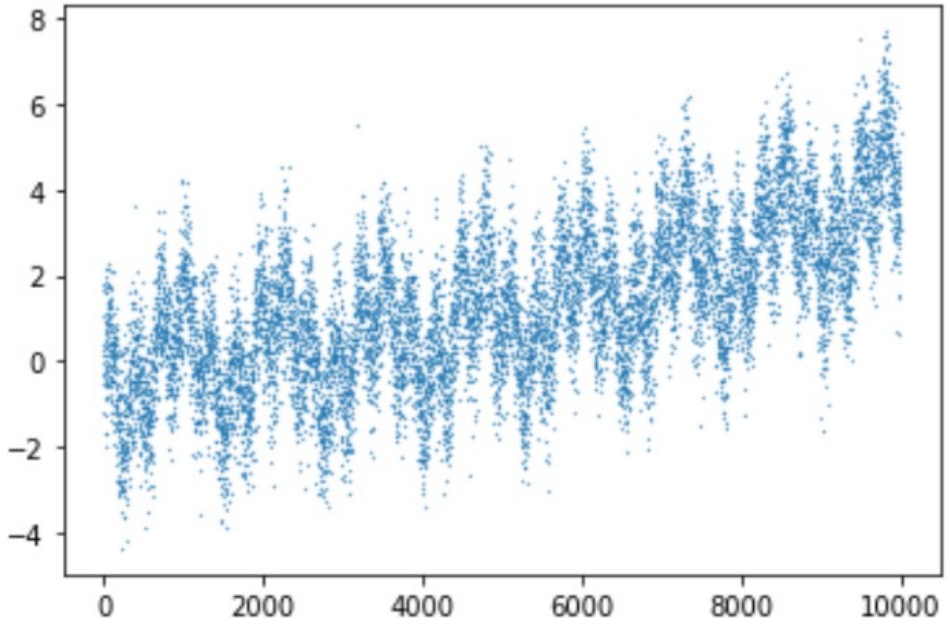
- Стационарный ряд имеет постоянный тренд и только случайная компонента
- В нестационарном может меняться тренд или присутствует циклическая компонента

Синтетические данные

```
X = np.arange(10000)
y = np.sin(X/50)-np.sin(X/200)+(2*X/X.size)**2
y += np.random.normal(scale=1.0, size=y.size)
plt.scatter(X, y[:10000], s=0.1)

df = pd.Series(y)
df = df.diff().dropna()
```

- Первая часть лабораторной посвящена анализу синтетических данных
- Создадим ряд со всеми тремя компонентами

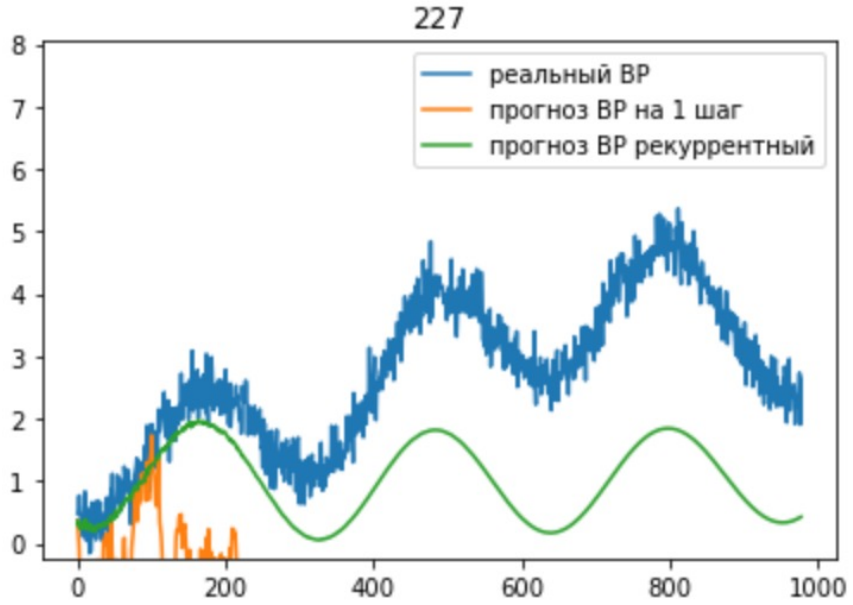


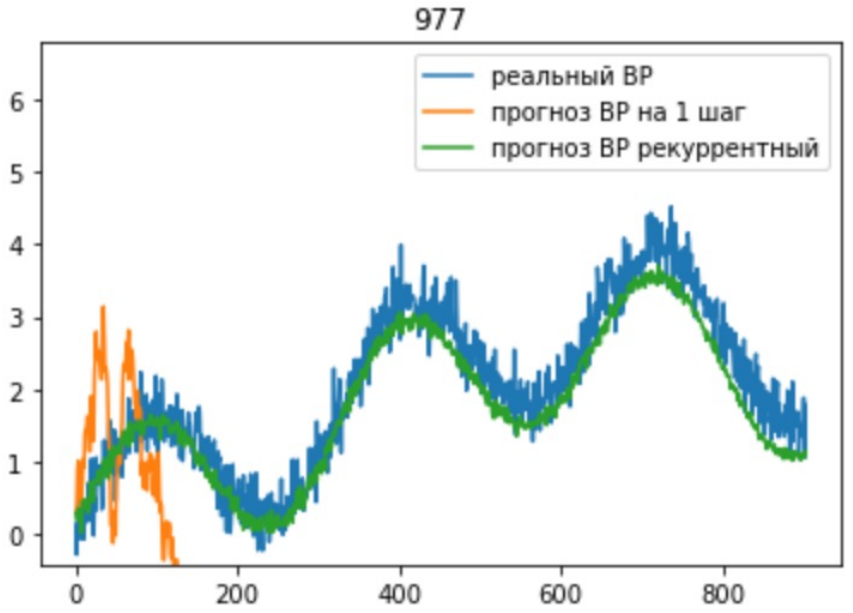
Модель авторегрессии

- **Авторегрессионная (AR) модель** (autoregressive model) — модель временных рядов, в которой значения временного ряда в данный момент линейно зависят от предыдущих значений этого же ряда.
- Авторегрессионный процесс порядка p (AR(p)-процесс) определяется следующим образом

$$X_t = c + \sum_{i=1}^{p} a_i X_{t-i} + \varepsilon_t$$

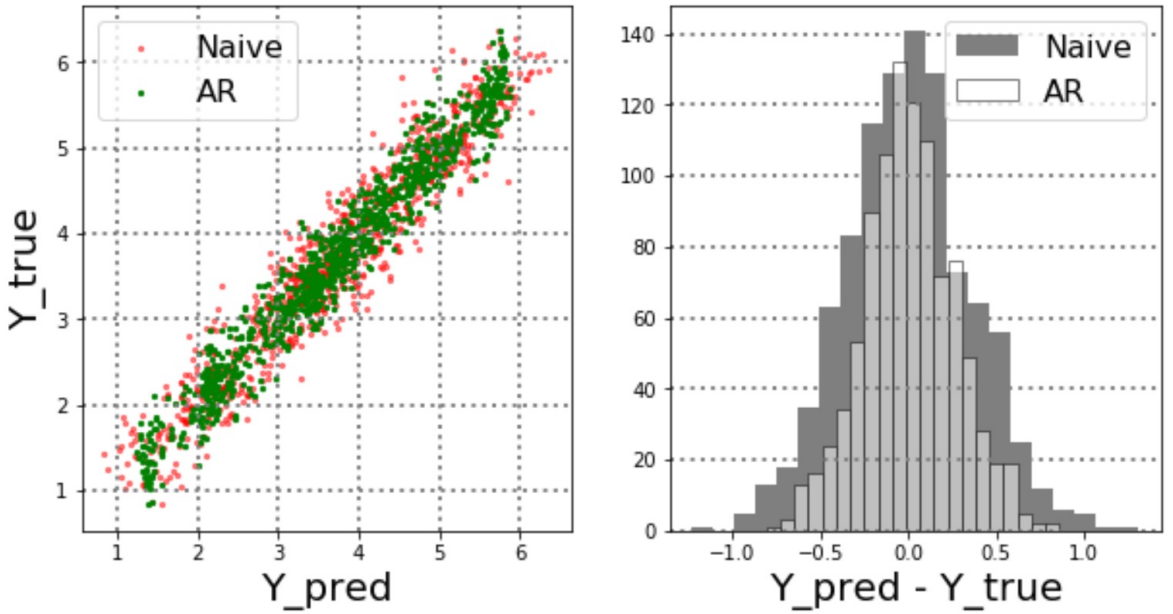
- *a* - ***параметры*** модели (коэффициенты авторегрессии)
- c - **постоянная** (часто для упрощения предполагается равной нулю)
- e - **белый шум**

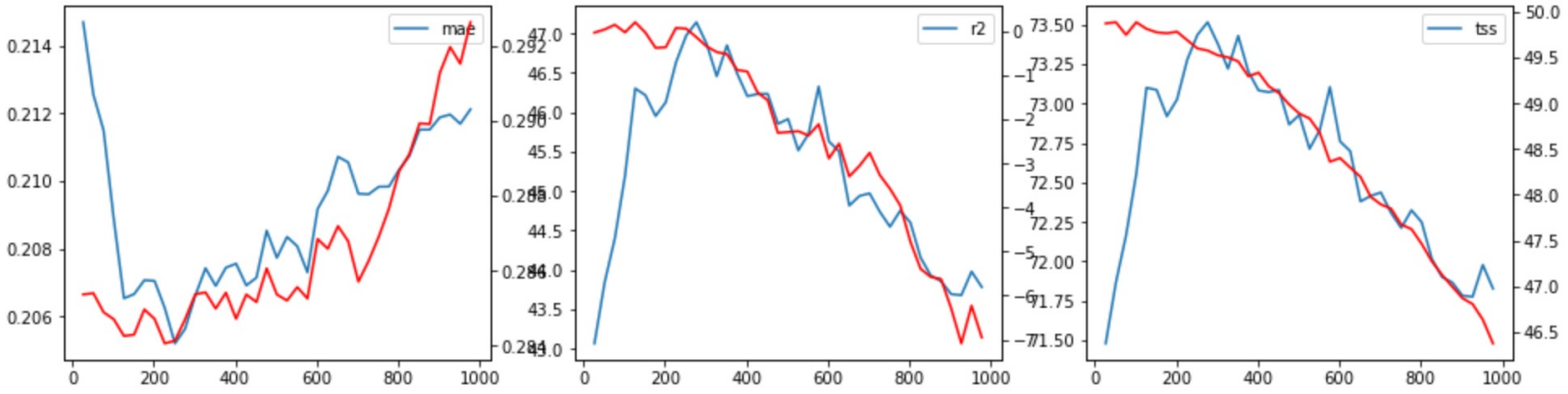




Результаты

- Выведем график плотности распределения **разности** предсказанного и истинного значения





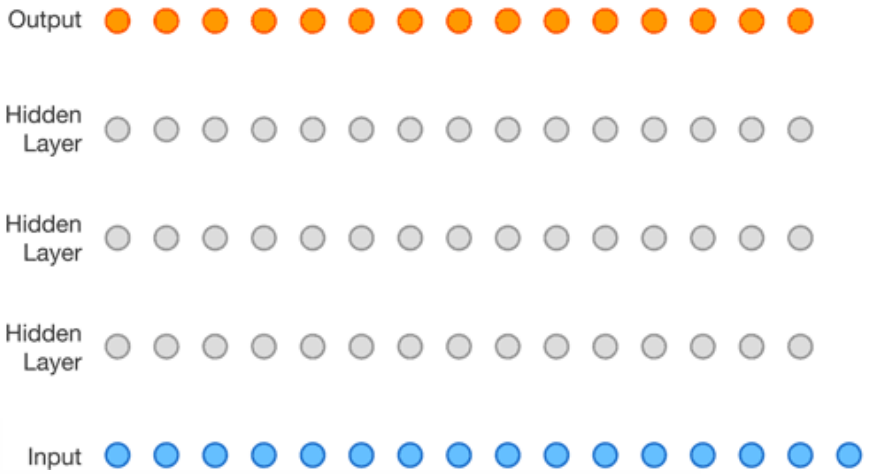
- Выведем графики средней абсолютной ошибки и коэффициента детерминации

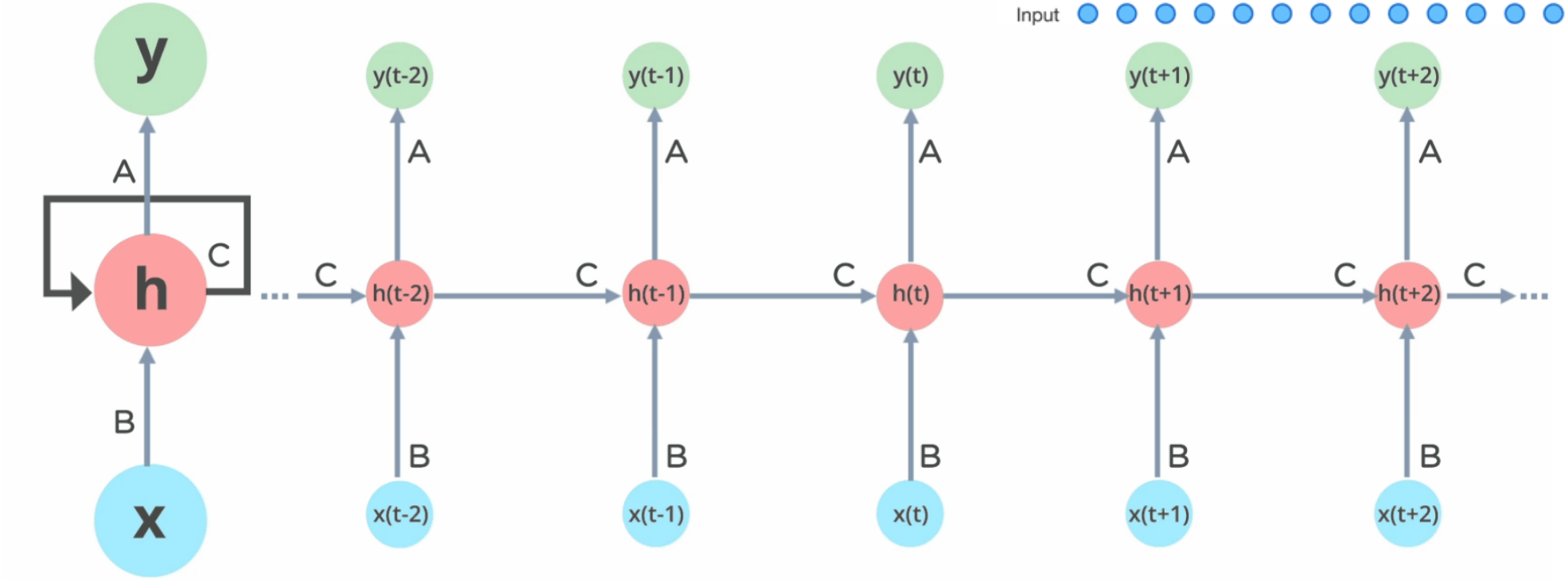
Рекуррентная нейронная сеть

- **Рекуррентные нейронные сети** (*Recurrent neural network*; *RNN*) — вид нейронных сетей, где связи между элементами образуют направленную последовательность.
- Благодаря этому появляется возможность обрабатывать серии событий во времени или последовательные пространственные цепочки.
- В отличие от многослойных перцептронов, рекуррентные сети могут использовать свою внутреннюю память для обработки последовательностей произвольной длины.

Рекуррентная нейросеть

- Рекуррентная нейросеть позволяет реализовать память
- GRU, LSTM





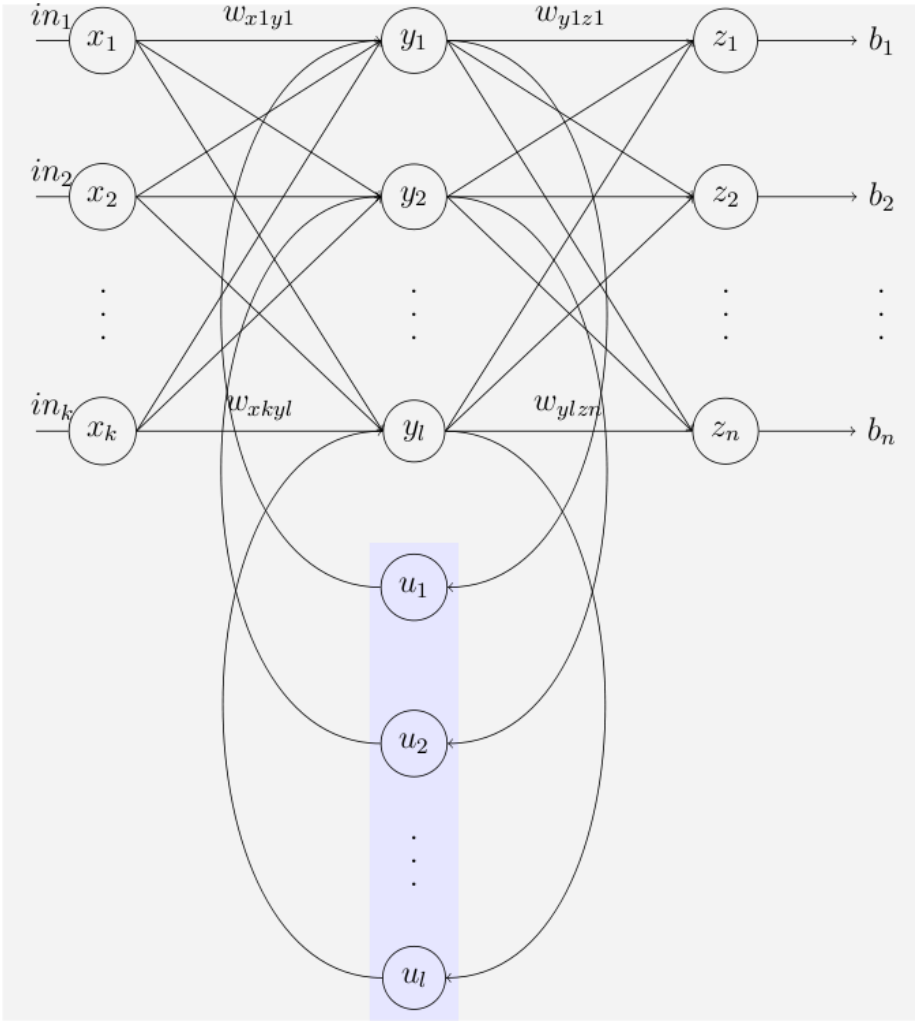
Рекуррентная нейронная сеть

- Сеть Элмана

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + b_y)$$

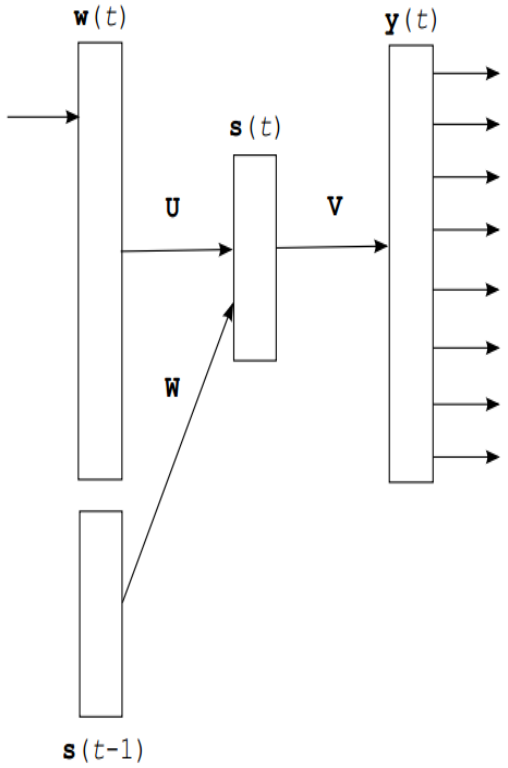
- Сеть Джордана

$$h_t = \sigma_h(W_h x_t + U_h y_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + b_y)$$



Архитектура: простая RNN

- Входной слой, скрытый слой с реккурентными соединениями и выходной слой
- В теории скрытый слой может обладать неограниченной памятью
- Также называется сетью Элмана (*Finding structure in time*, Elman 1990)



Архитектура: простая RNN

$\mathbf{s}(t) = f(\mathbf{U}\mathbf{w}(t) + \mathbf{W}\mathbf{s}(t-1))$

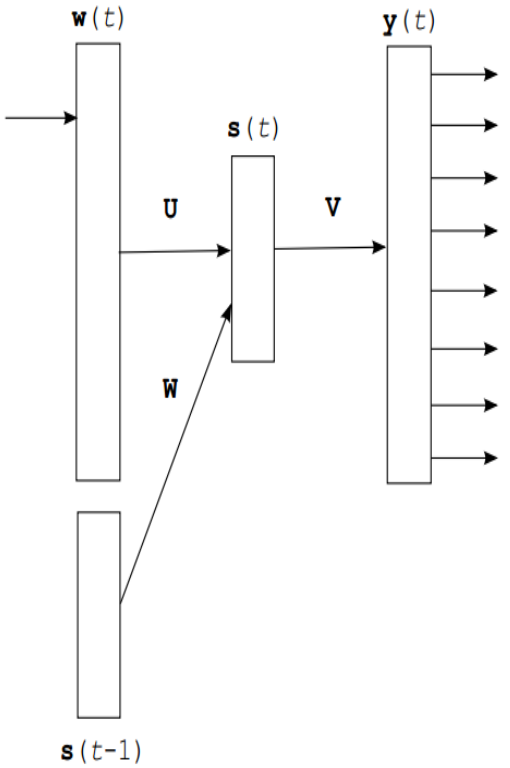
$\mathbf{y}(t) = g(\mathbf{V}\mathbf{s}(t))$

f() чаще всего сигмоидальная активационная функция:

$$f(z) = \frac{1}{1 + e^{-z}}$$

g() чаще всего softmax:

$$g(z_m) = \frac{e^{z_m}}{\sum_k e^{z_k}}$$



Обучение: обратное распространение по времени (Backpropagation through time, BPTT)

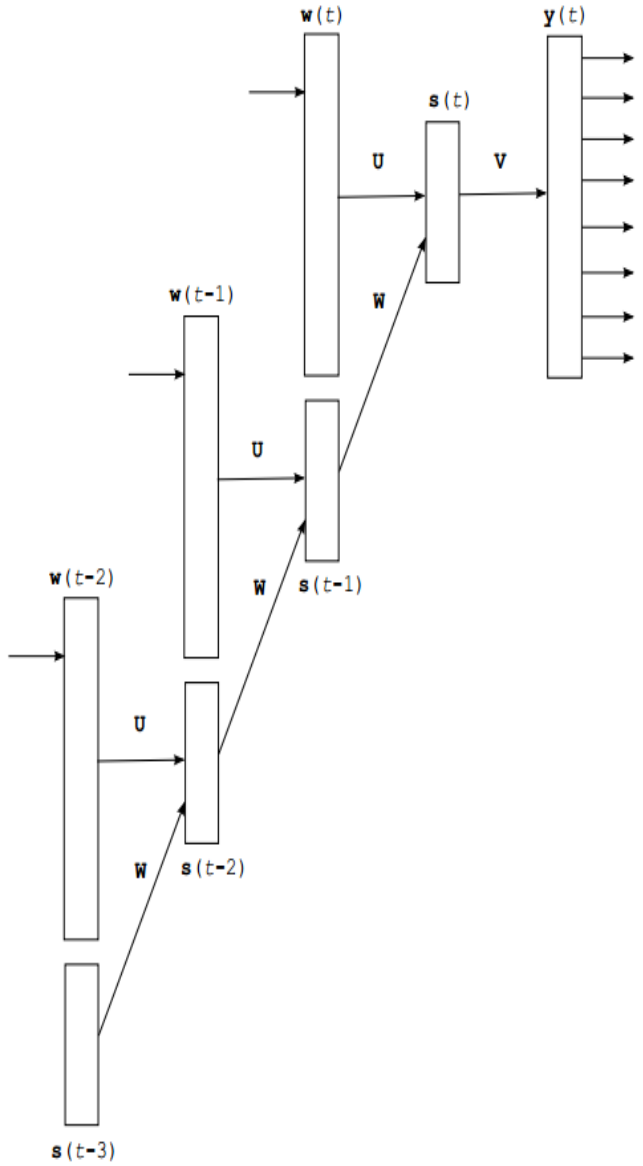
– Как обучать рекуррентные сети?

– Выходное значение зависит от состояния скрытого слоя, который зависит от всех предыдущих состояний скрытого слоя (и, следовательно, всех предыдущих входов)

– Рекуррентная сеть может рассматриваться как (очень глубокая) сеть прямого распространения с общими весами

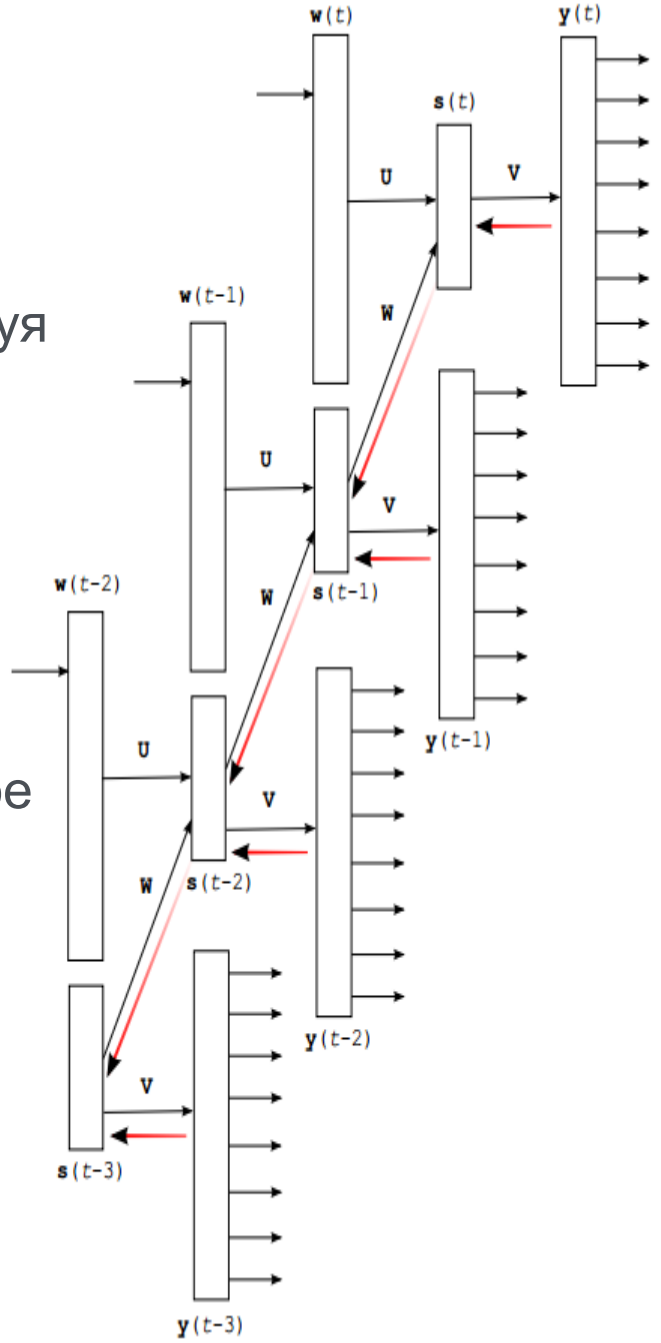
Обратное распространение по времени

- Идея заключается в том, что РНС разворачивается во времени
- Мы получаем глубокую нейронную сеть с общими весами **U** и **W**
- Часто бывает достаточно развернуть на несколько шагов (называется *укороченным BPTT*)



Обратное распространение по времени

- Мы тренируем развернутый РНС, используя обычное обр. распр-е + СГС
- На практике ограничивают количество шагов разворачивания до 5 – 10
- Эффективнее вычислить градиент после нескольких обучающих примеров (пакетное обучение)



Исчезающие градиенты

- Когда мы распространяем градиенты назад во времени, обычно их величина быстро уменьшается: это называется «проблемой исчезающего градиента»

- На практике это означает, что изучение долгосрочных зависимостей в данных затруднительно для простой RNN архитектуры

- Специальные архитектуры RNN решают эту проблему:
  - Экспоненциальная память трассировки (Jordan 1987, Mozer 1989)
  - Долгая краткосрочная память (Hochreiter & Schmidhuber, 1997))
  - будут обсуждены во второй части этой лекции

- Множество научных теорий
  - лучше инициализировать рекуррентную матрицу и использовать импульс во время обучения
- Sutskever et.al.,: On The Importance of Initialization and Momentum in Deep Learning
  - изменять архитектуру

Взрывающиеся градиенты

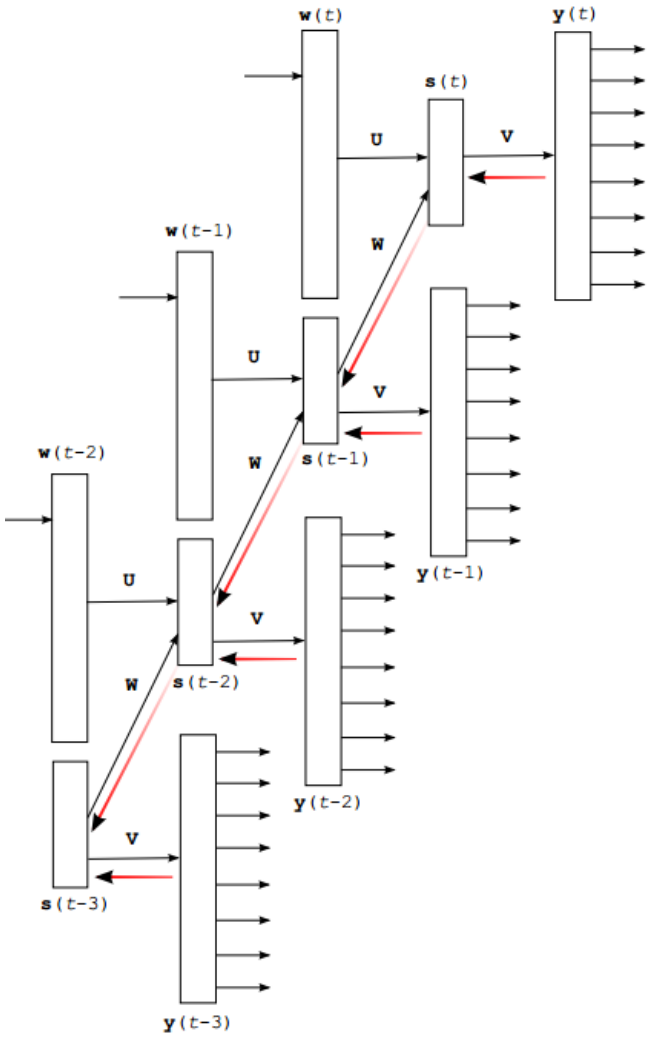
– Иногда градиенты начинают экспоненциально возрастать во время обратного распространения через рекуррентные веса

– Бывает редко, но эффект может быть катастрофическим: огромные градиенты приведут к большому изменению весов и, таким образом, разрушат то, что было изучено до сих пор

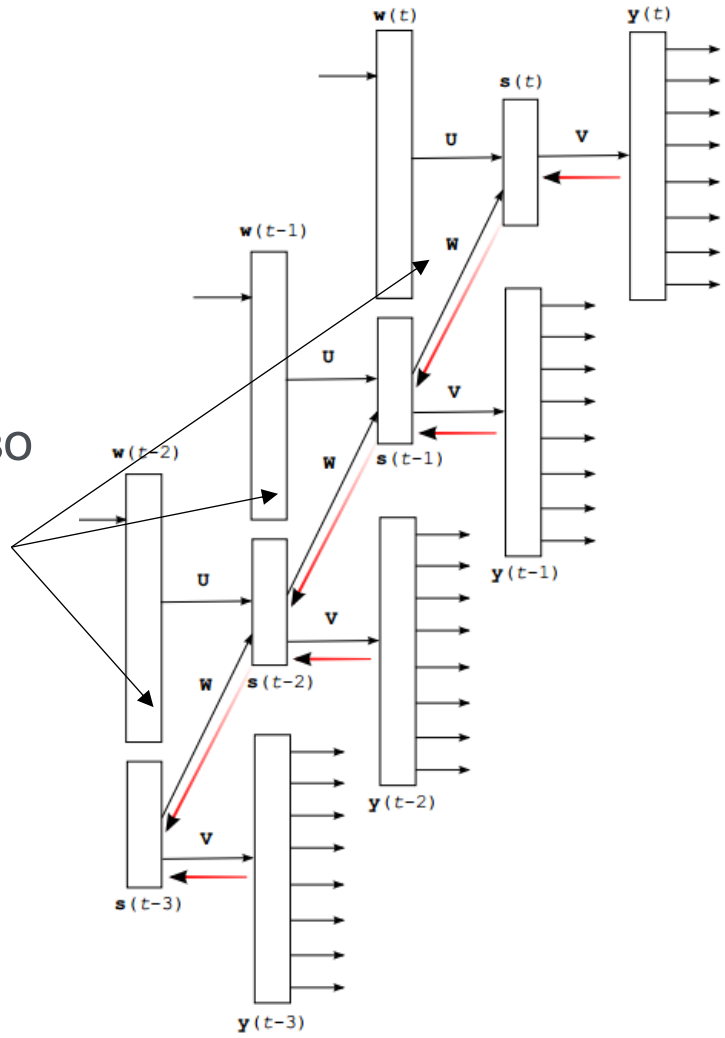
– Одна из основных причин, по которым RNN должны были быть нестабильными

– Простое решение (впервые опубликован в RNNLM toolkit в 2010): обрезать или нормализовать значения градиентов, чтобы избежать огромных изменений весов

Обучение: Взрывающиеся градиенты



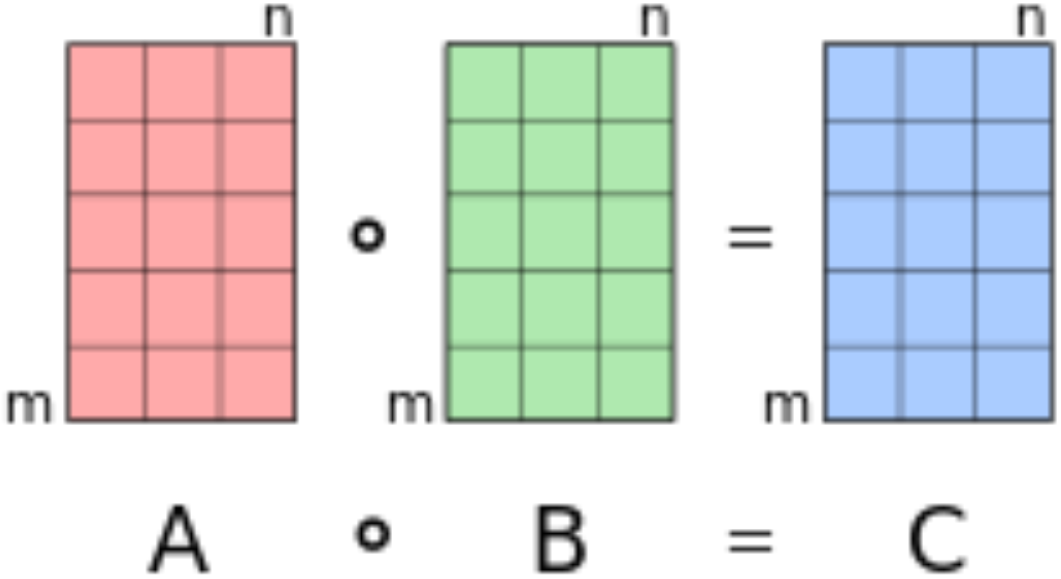
– Обрезка градиента во время обратного распространения по времени



Произведение Адамара

- Произведение Адамара оперирует двумя матрицами одинаковой размерности и создаёт новую матрицу идентичной размерности.

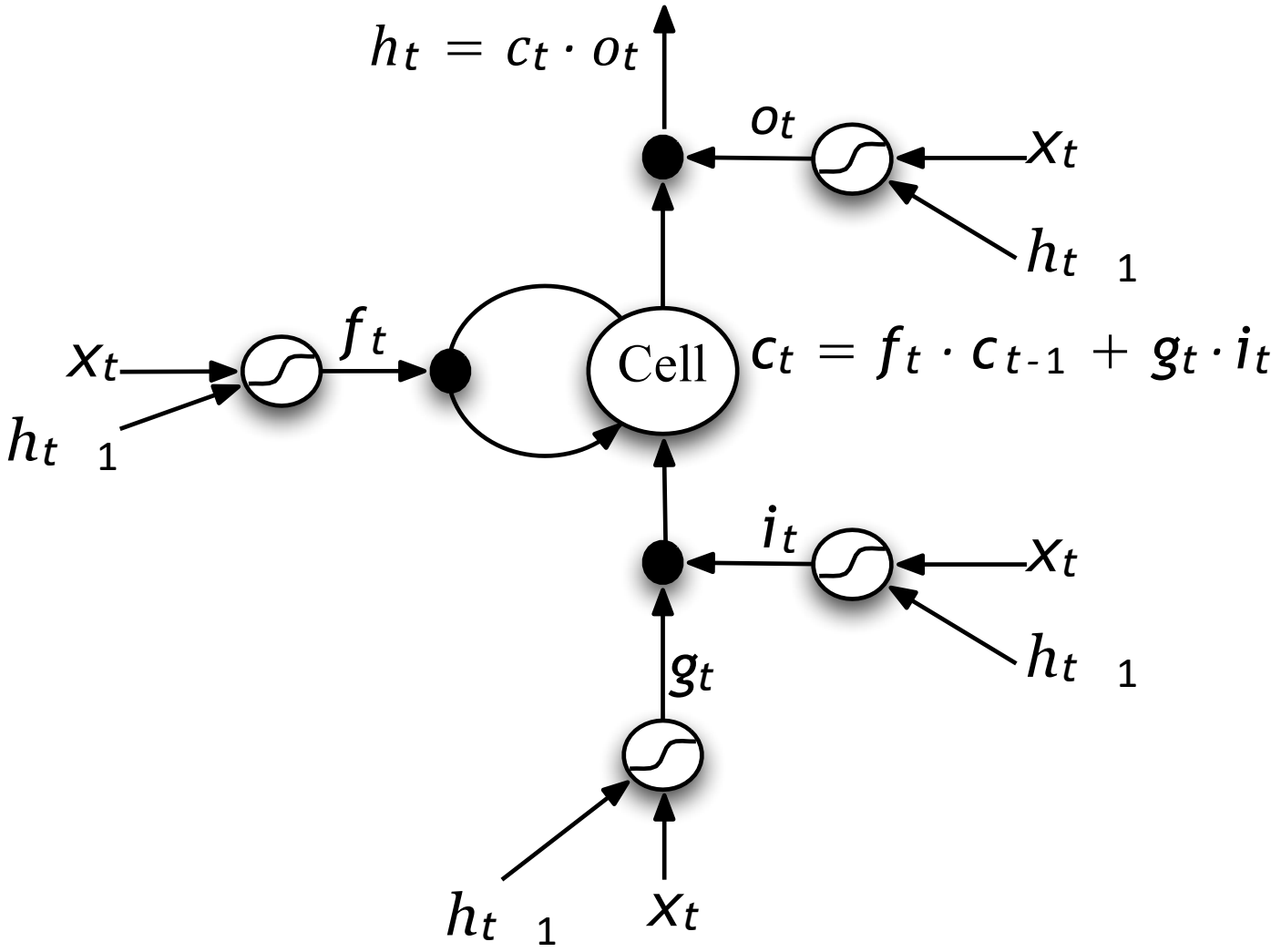
$$(A \circ B)_{i,j} = (A)_{i,j} \cdot (B)_{i,j}$$



$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \circ \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}\, b_{11} & a_{12}\, b_{12} & a_{13}\, b_{13} \\ a_{21}\, b_{21} & a_{22}\, b_{22} & a_{23}\, b_{23} \\ a_{31}\, b_{31} & a_{32}\, b_{32} & a_{33}\, b_{33} \end{bmatrix}$$

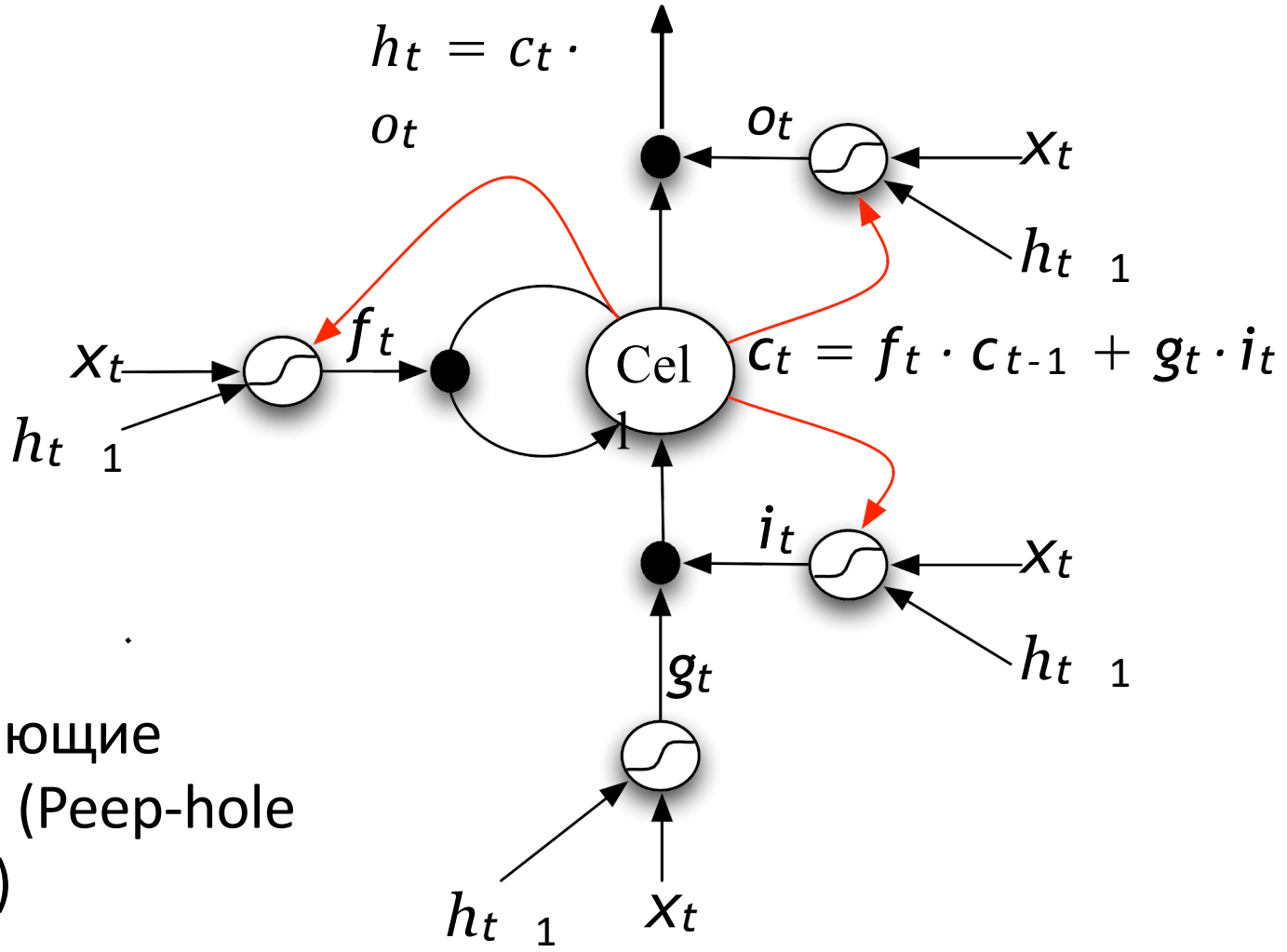
Долгая краткосрочная память (LSTM)

- Приобрела большую популярность для обработки временных рядов и текстов
- Использовалась в машинном переводе до появления трансформеров
- Имеются явные «ячейки» памяти для хранения кратковременных активаций, наличие дополнительных вентилей частично устраняет проблему исчезающего градиента
- Многослойные версии показали, что они хорошо работают над задачами, которые имеют среднесрочные зависимости



Hochreiter et.al.,1997: Long Short-Term Memory

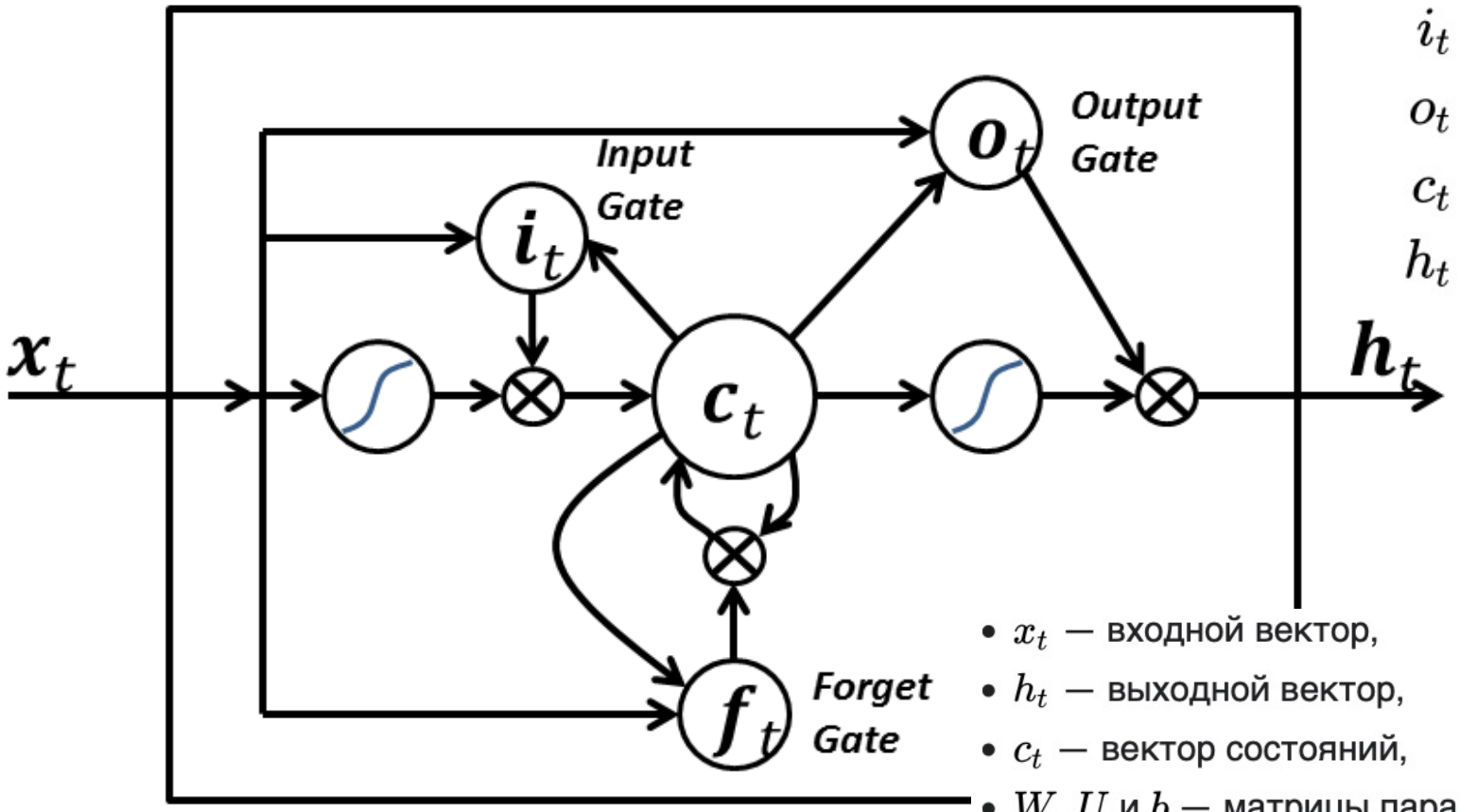
Долгая краткосрочная память (LSTM) с «глазками»



Подглядывающие соединения (Peep-hole connections)

Hochreiter et.al.,1997: Long Short-Term Memory

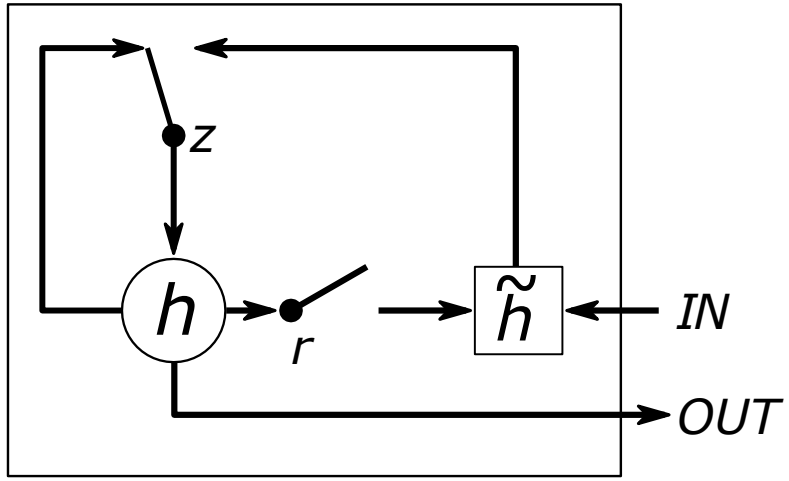
LSTM



$$f_t = \sigma_g(W_f x_t + U_f c_{t-1} + b_f)$$
$$i_t = \sigma_g(W_i x_t + U_i c_{t-1} + b_i)$$
$$o_t = \sigma_g(W_o x_t + U_o c_{t-1} + b_o)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + b_c)$$
$$h_t = o_t \circ \sigma_h(c_t)$$

- x_t — входной вектор,
- h_t — выходной вектор,
- c_t — вектор состояний,
- W, U и b — матрицы параметров и вектор,
- f_t, i_t и o_t — векторы вентилей,
  - f_t — вектор вентиля забывания, вес запоминания старой информации,
  - i_t — вектор входного вентиля, вес получения новой информации,
  - o_t — вектор выходного вентиля, кандидат на выход.

Рекуррентные нейрон с вентилями (Gated recurrent unit, GRU)



– Вентиль обновления: $z_t^j = \sigma(W_z \mathbf{x}_t + U_z \mathbf{h}_{t-1})^j$.

– Вентиль перезаписи: $r_t^j = \sigma(W_r \mathbf{x}_t + U_r \mathbf{h}_{t-1})^j$.

– Кандидат на активацию: $\tilde{h}_t^j = \tanh(W \mathbf{x}_t + U(\mathbf{r}_t * \mathbf{h}_{t-1}))^j$,

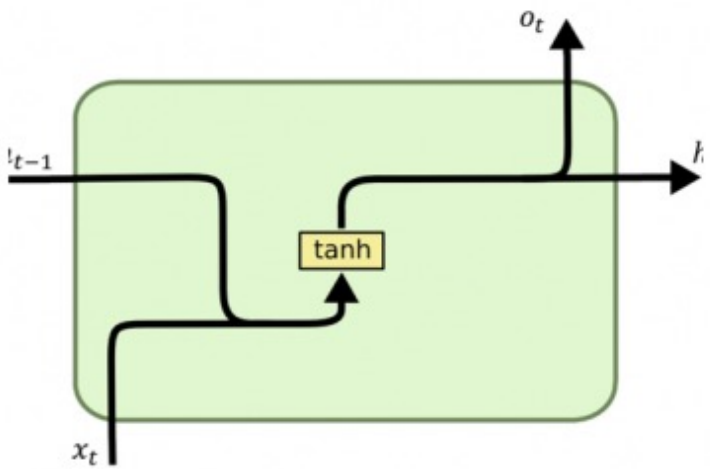
$$h_t^j = (1 - z_t^j) h_{t-1}^j + z_t^j \tilde{h}_t^j,$$

GRU

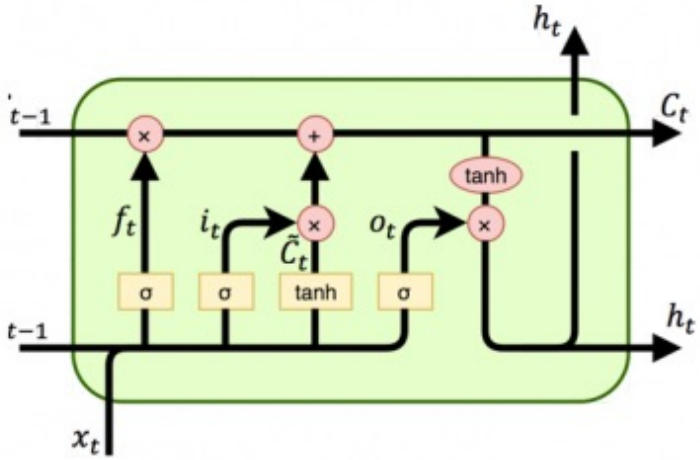
$$z_t = \sigma_g(W_z x_t + U_z h_{t-1} + b_z)$$
$$r_t = \sigma_g(W_r x_t + U_r h_{t-1} + b_r)$$
$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \sigma_h(W_h x_t + U_h(r_t \circ h_{t-1}) + b_h)$$

- x_t: входной вектор
- h_t: выходной вектор
- z_t: вектор вентиля обновления
- r_t: вектор вентиля сброса
- W, U и b: матрицы параметров и вектор

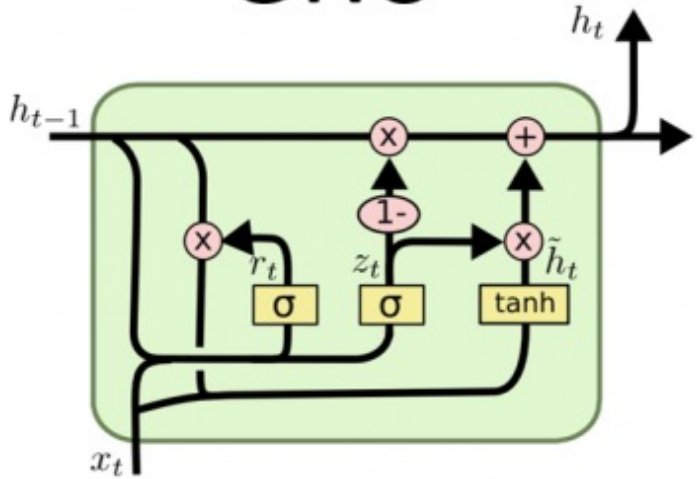
RNN



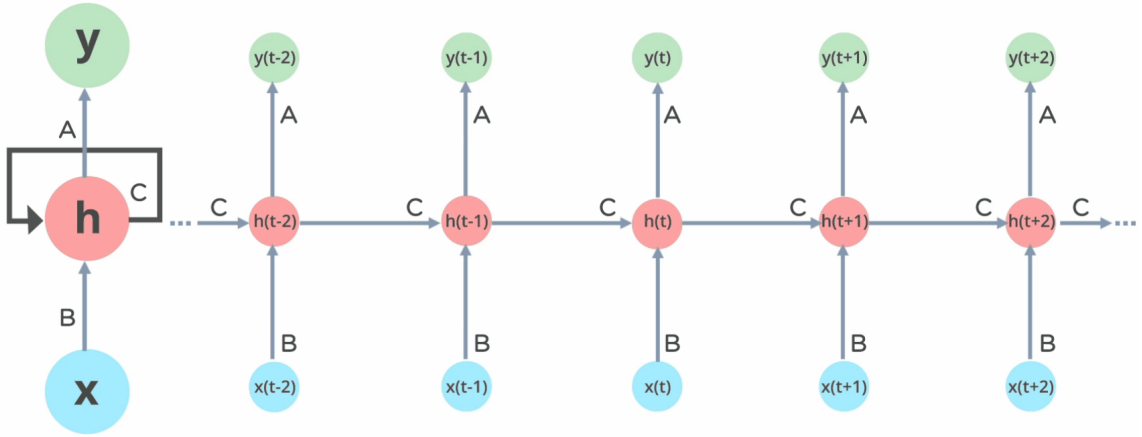
LSTM



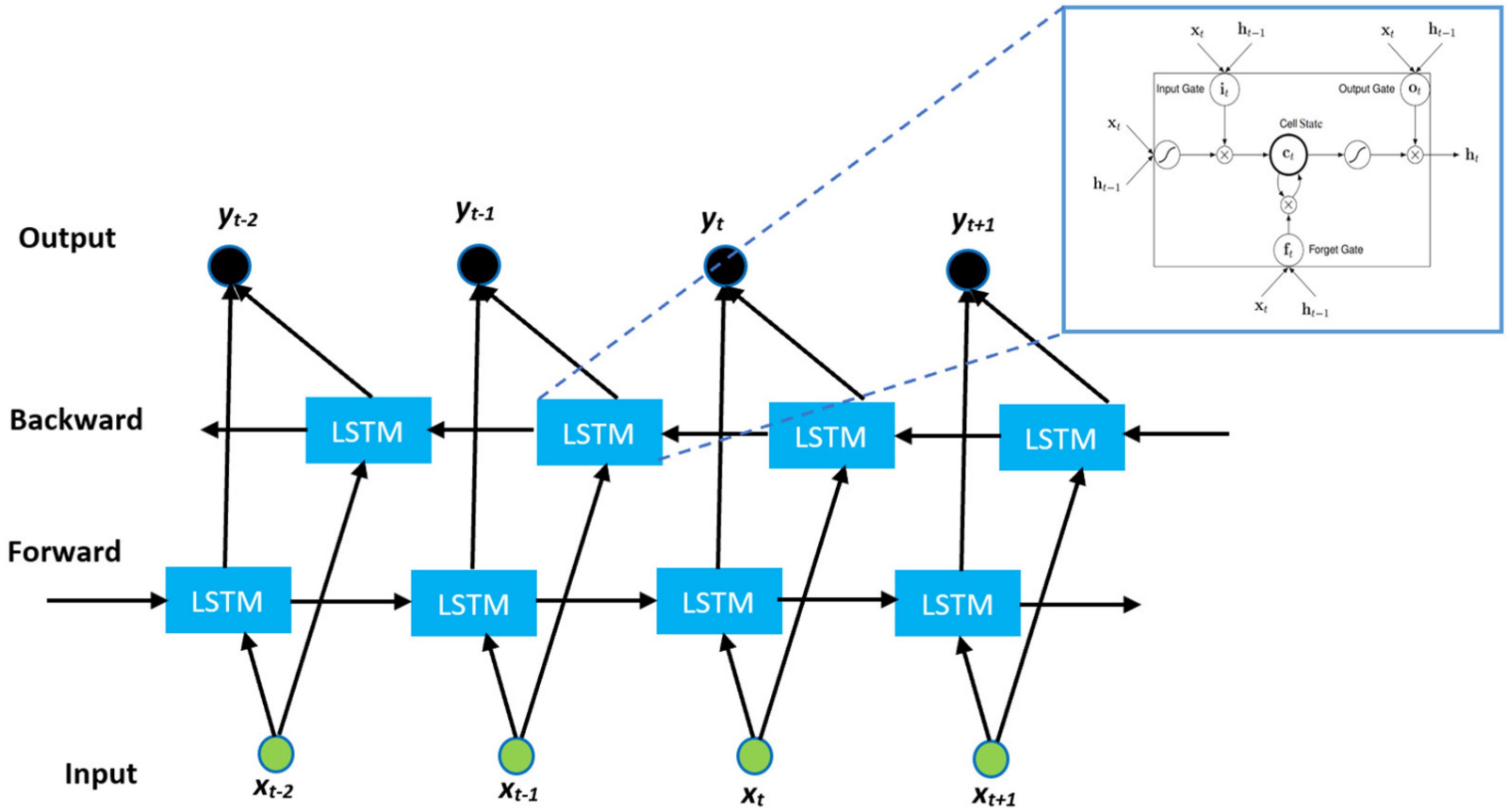
GRU



Bidirectional LSTM



- В простой RNN или LSTM мы каждый раз даем на вход следующее значение
- В двунаправленной сети у нас уже две ячейки: одна предсказывает значение, зная все предыдущие, а вторая «читает» с конца
- Двунаправленная сеть используется в известной языковой модели ELMo



LSTM из практики

```
def __init__(self, num_features, input_size, hidden_size, num_layers,
             bidirectional=True, p=0.4):
    super(LSTM, self).__init__()

    self.num_features = num_features
    self.num_layers = num_layers
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.bidirectional = bidirectional

    self.lstm = nn.LSTM(input_size=input_size,
                        hidden_size=hidden_size,
                        bidirectional=bidirectional,
                        num_layers=num_layers,
                        batch_first=True)
    self.dropout = nn.Dropout(p)

    self.fc = nn.Linear(2*hidden_size if bidirectional else hidden_size, num_features)
```

```
num_features = 1
input_size = 1
hidden_size = 16
num_layers = 2
bidirectional = True
dropout_rate = 0.2
```

Наша рекуррентная сеть **двунаправленная**

Состоит из:

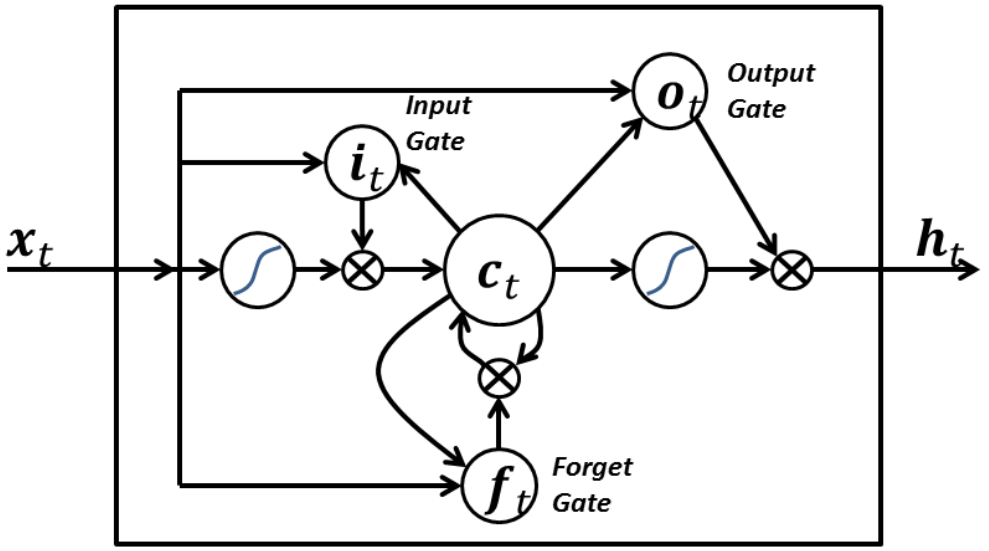
- двух слоев LSTM
- слой dropout
- полносвязный выходной слой

LSTM из практики

```
self.lstm = nn.LSTM(input_size=input_size,
                    hidden_size=hidden_size,
                    bidirectional=bidirectional,
                    num_layers=num_layers,
                    batch_first=True)
self.dropout = nn.Dropout(p)

self.fc = nn.Linear(2*hidden_size if bidirectional
```

```
lstm.weight_ih_l0 torch.Size([64, 1])
lstm.weight_hh_l0 torch.Size([64, 16])
lstm.bias_ih_l0 torch.Size([64])
lstm.bias_hh_l0 torch.Size([64])
lstm.weight_ih_l0_reverse torch.Size([64, 1])
lstm.weight_hh_l0_reverse torch.Size([64, 16])
lstm.bias_ih_l0_reverse torch.Size([64])
lstm.bias_hh_l0_reverse torch.Size([64])
lstm.weight_ih_l1 torch.Size([64, 32])
lstm.weight_hh_l1 torch.Size([64, 16])
lstm.bias_ih_l1 torch.Size([64])
lstm.bias_hh_l1 torch.Size([64])
lstm.weight_ih_l1_reverse torch.Size([64, 32])
lstm.weight_hh_l1_reverse torch.Size([64, 16])
lstm.bias_ih_l1_reverse torch.Size([64])
lstm.bias_hh_l1_reverse torch.Size([64])
fc.weight torch.Size([1, 32])
fc.bias torch.Size([1])
```

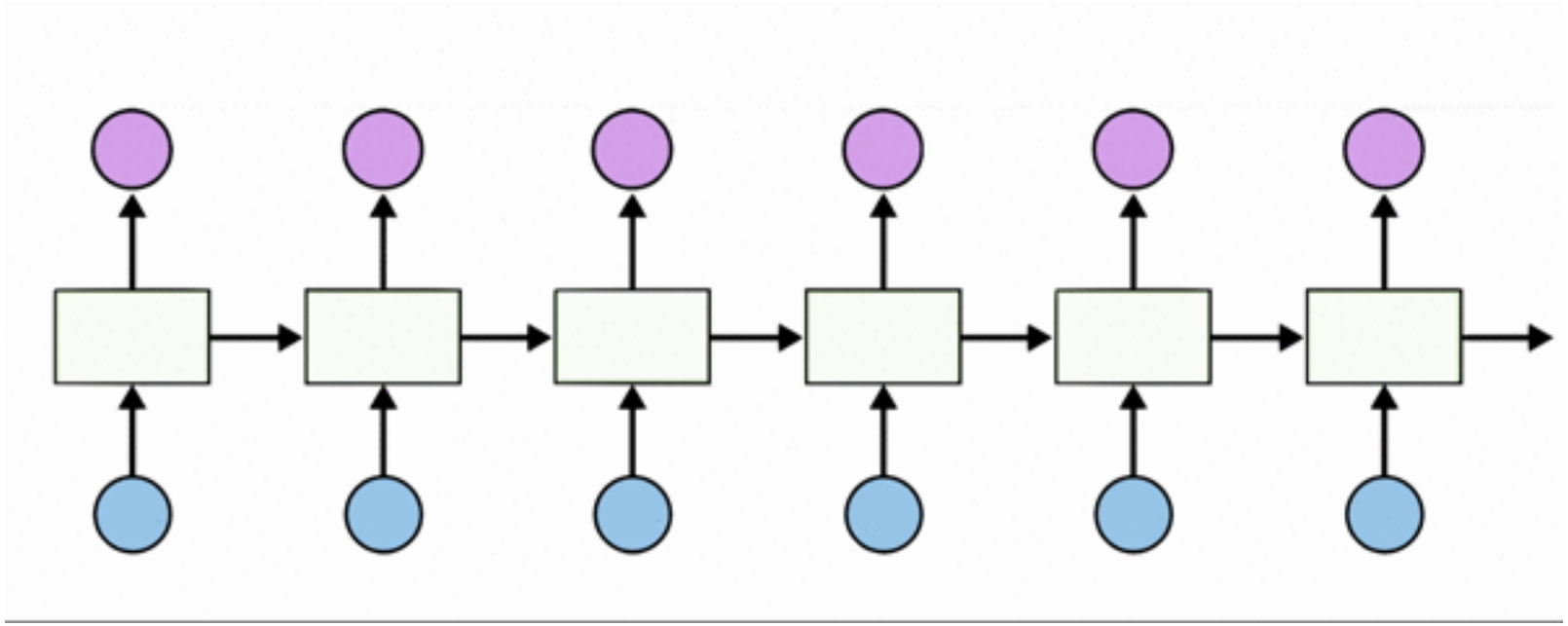


$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot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

- В PyTorch для каждой ячейки LSTM обучаются 4 группы параметров: weight_i, weight_h, bias_i, bias_h
- Каждая группа содержит по 4 части для 3 вентилей и состояния
- В задании 2 слоя ячеек и в каждом ячейки прямого и обратного распространения

RNN для генерации текста

- Рекуррентная нейронная сеть учитывает состояние ячейки для генерации нового символа
- Здесь простой пример для генерации символов
- Обычно генерируют слова или токены

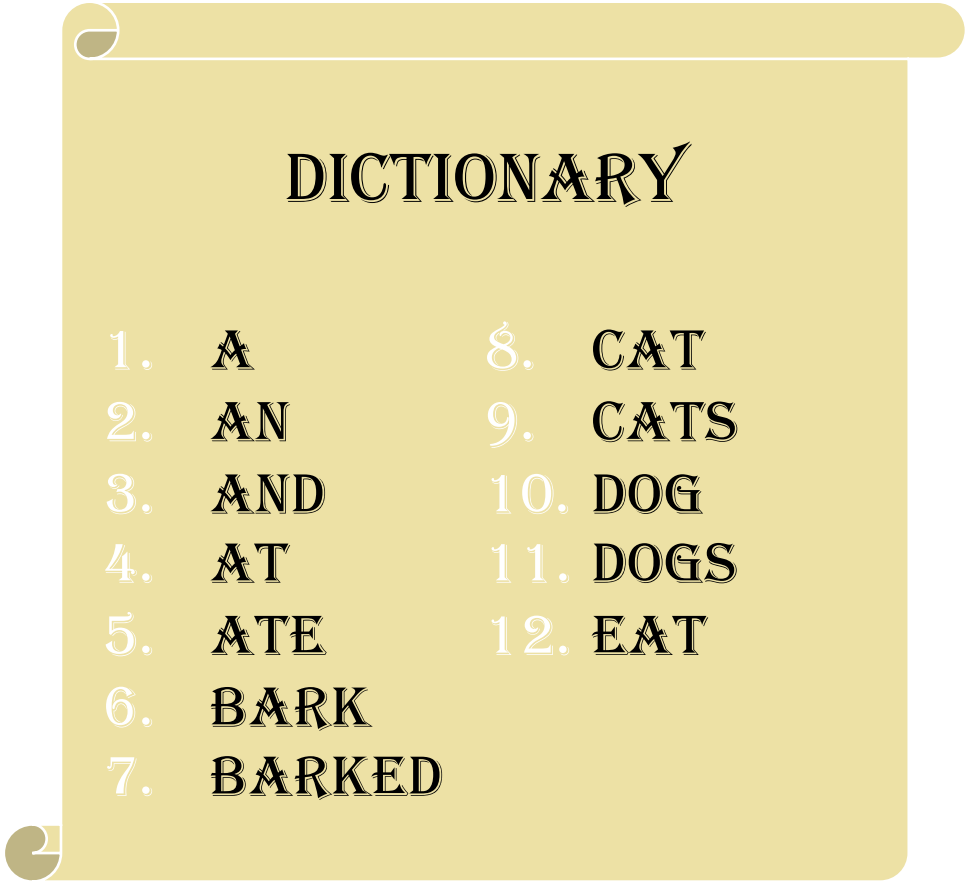


Обработка текста

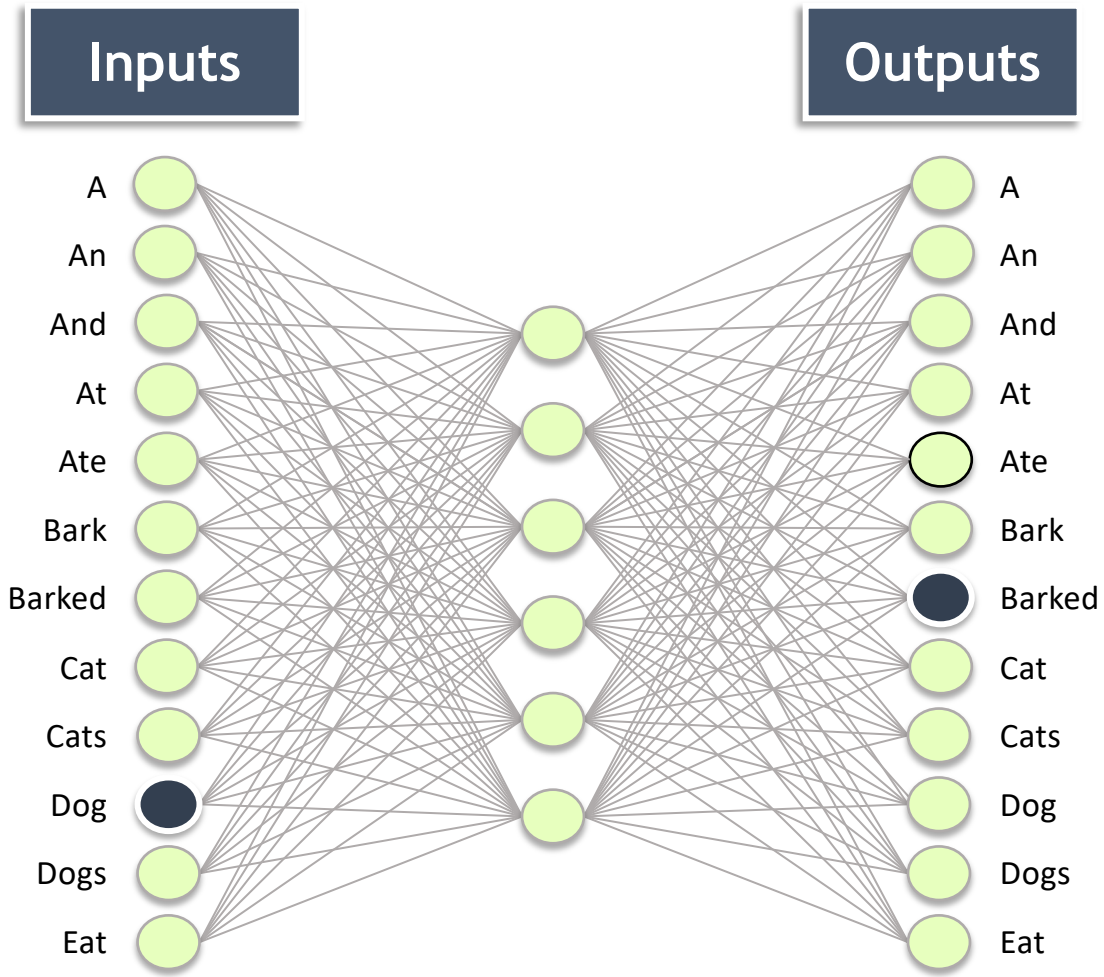
“A dog barked at a cat.”

[1, 10, 7, 4, 1, 8]

- Составляем словарь слов – нумеруем все уникальные слова из нашего текста
- Ставим в соответствие каждому уникальному слову какой-то нейрон (входной или выходной)



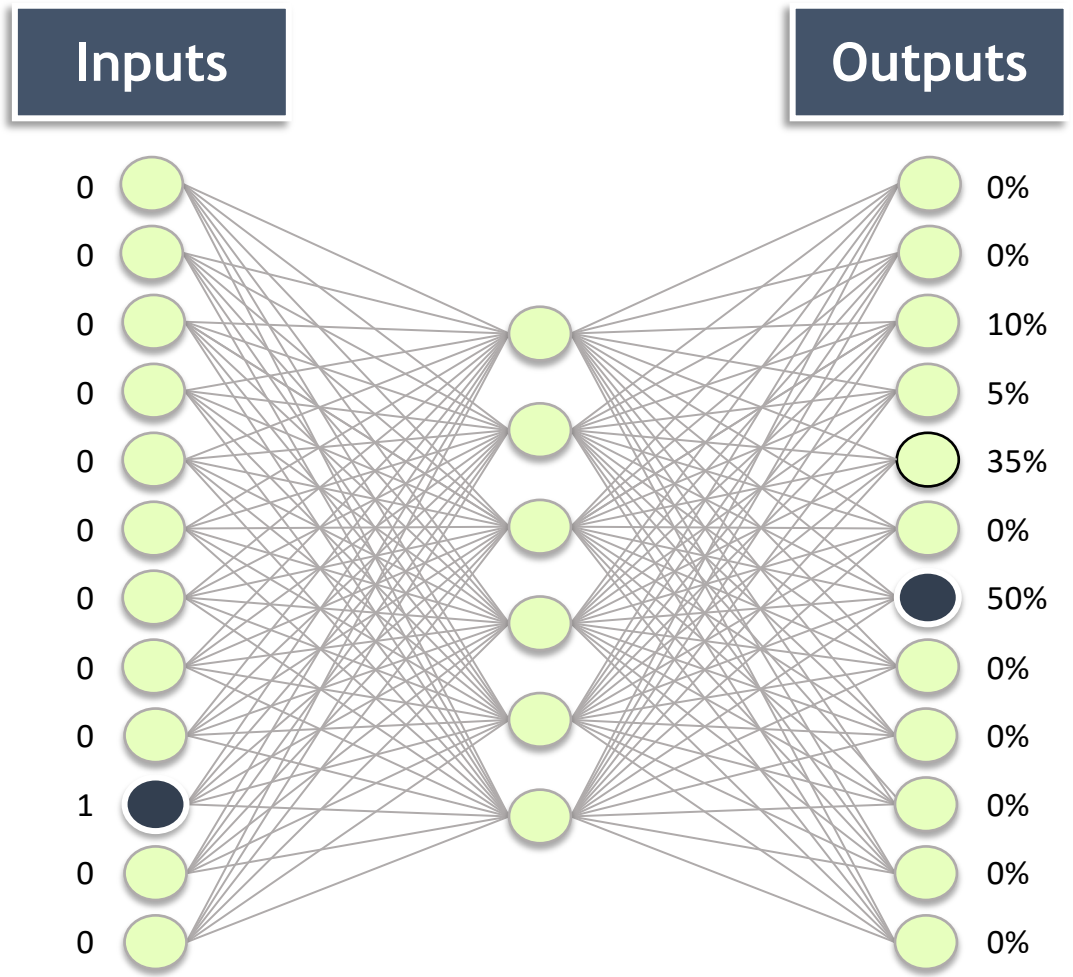
Полносвязная сеть для классификации



DICTIONARY

1. A
2. AN
3. AND
4. AT
5. ATE
6. BARK
7. BARKED
8. CAT
9. CATS
10. DOG
11. DOGS
12. EAT

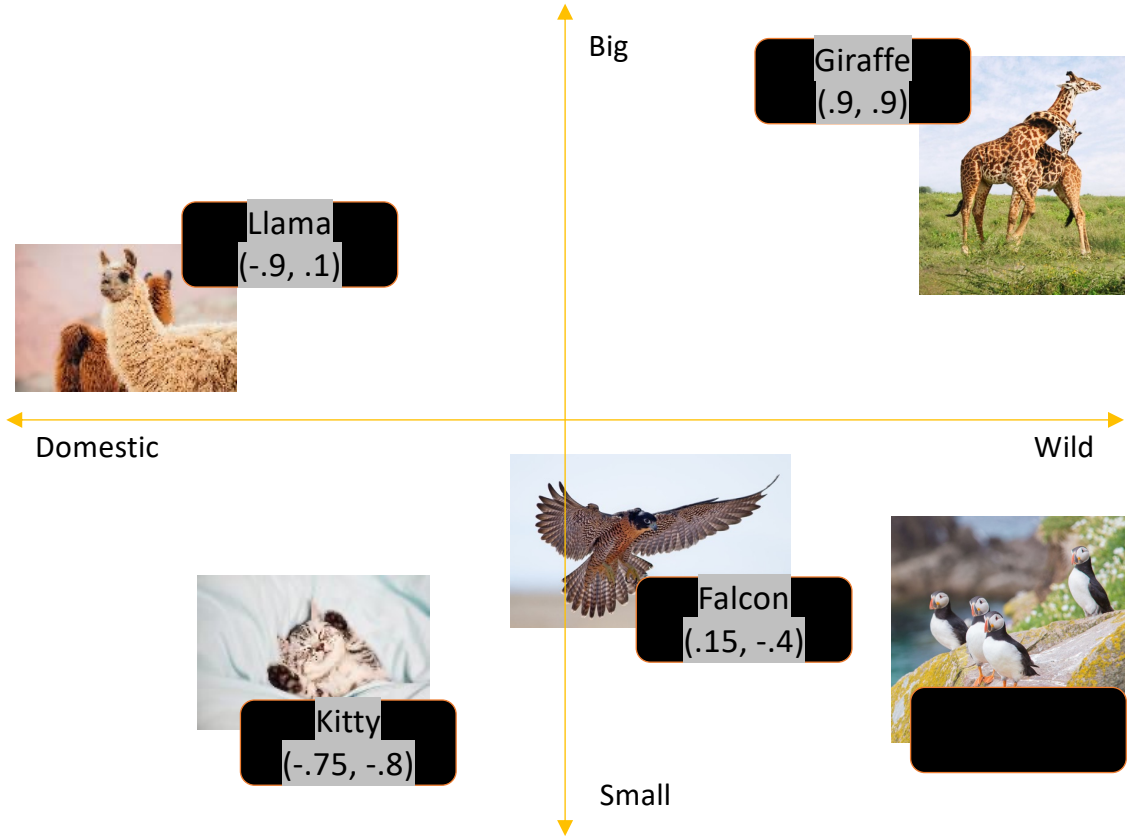
Полносвязная сеть для классификации



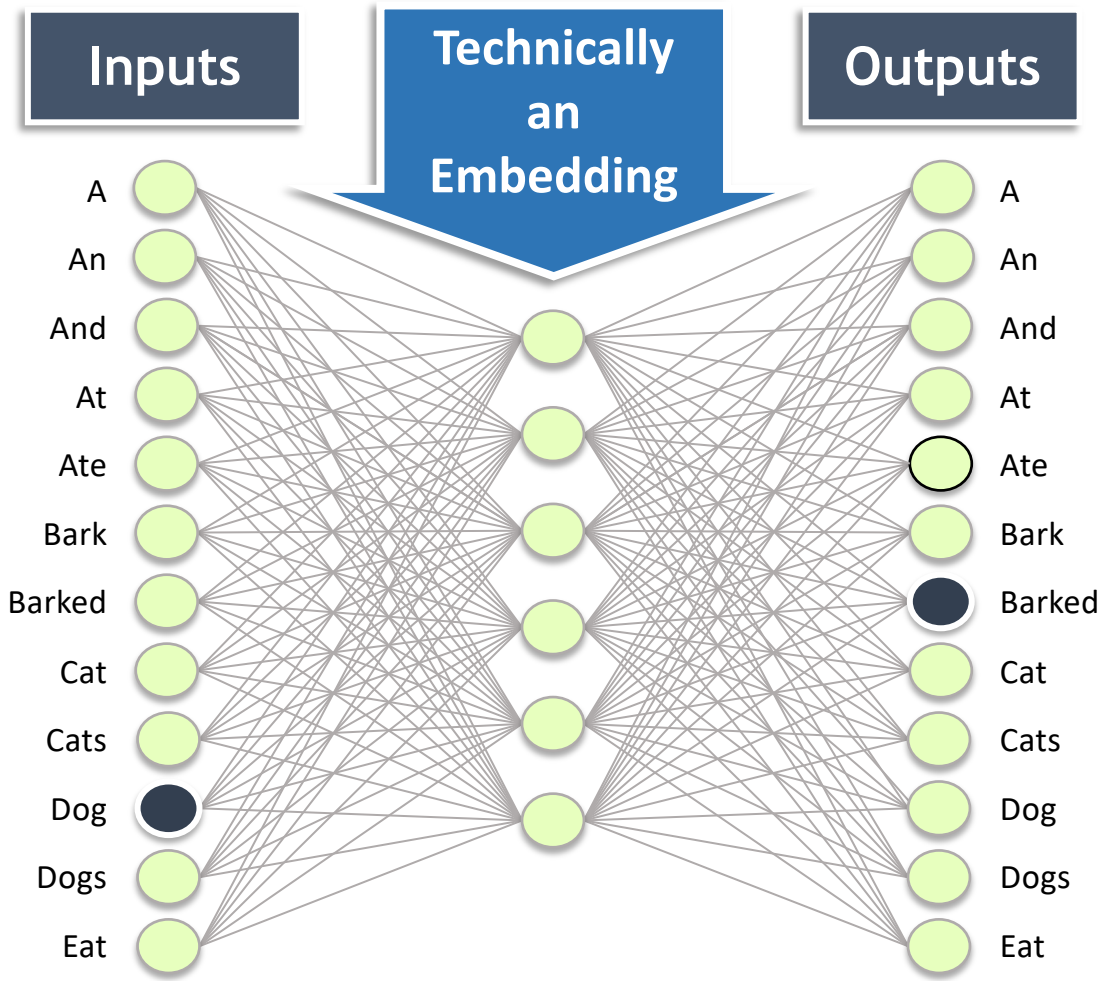
DICTIONARY

1. A
2. AN
3. AND
4. AT
5. ATE
6. BARK
7. BARKED
8. CAT
9. CATS
10. DOG
11. DOGS
12. EAT

Замена словаря на embedding



Добавляем embedding в полносвязную сеть



DICTIONARY

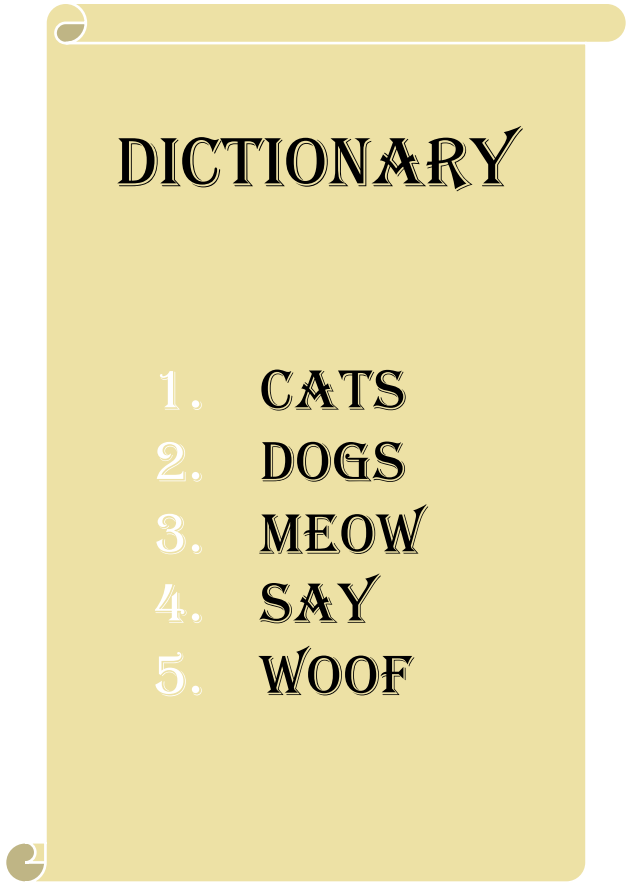
1. A
2. AN
3. AND
4. AT
5. ATE
6. BARK
7. BARKED
8. CAT
9. CATS
10. DOG
11. DOGS
12. EAT

Recurrent Neural Networks

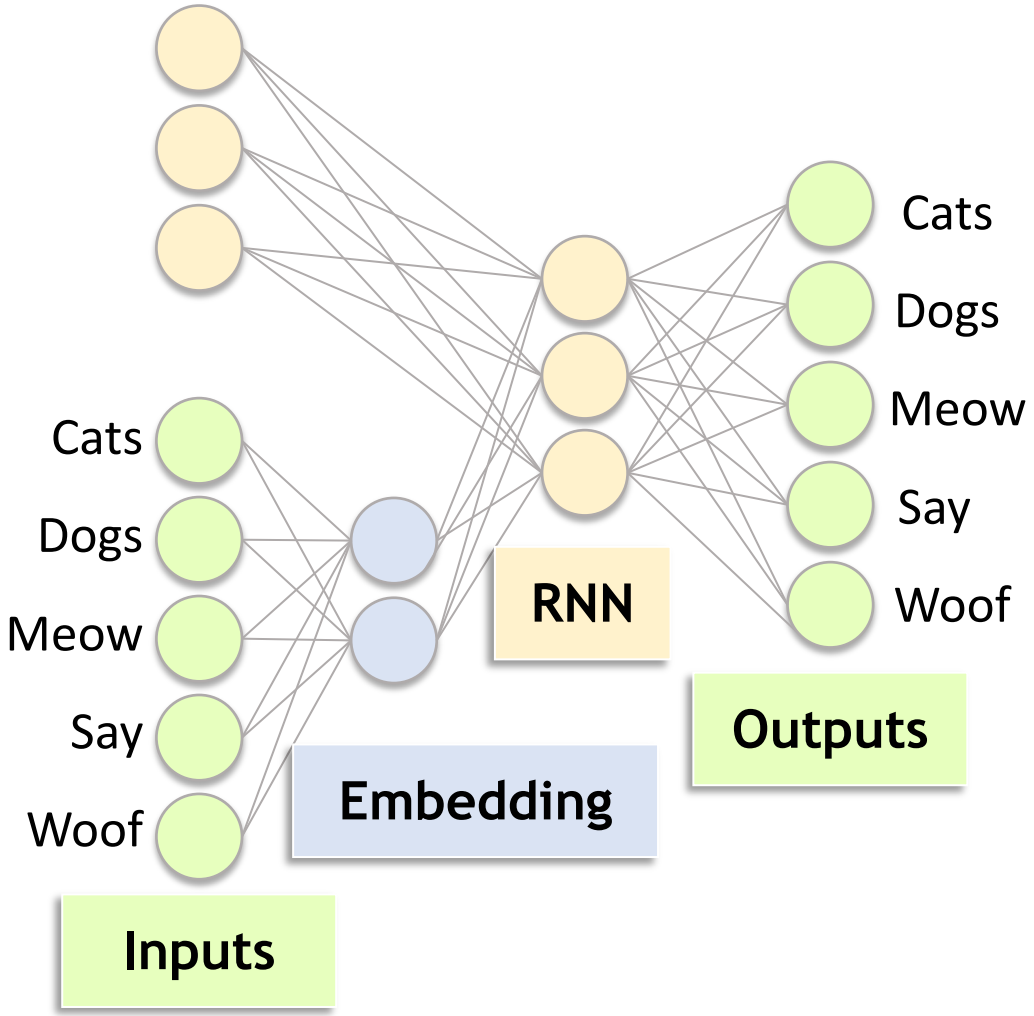
- Усложним задачу: теперь в обоих случаях есть одно и то же слово ”say ” и сеть должна запомнить, что было до этого
- Для этого будем использовать RNN

“Cats say ___.”

“Dogs say ___.”



RNN для генерации текста



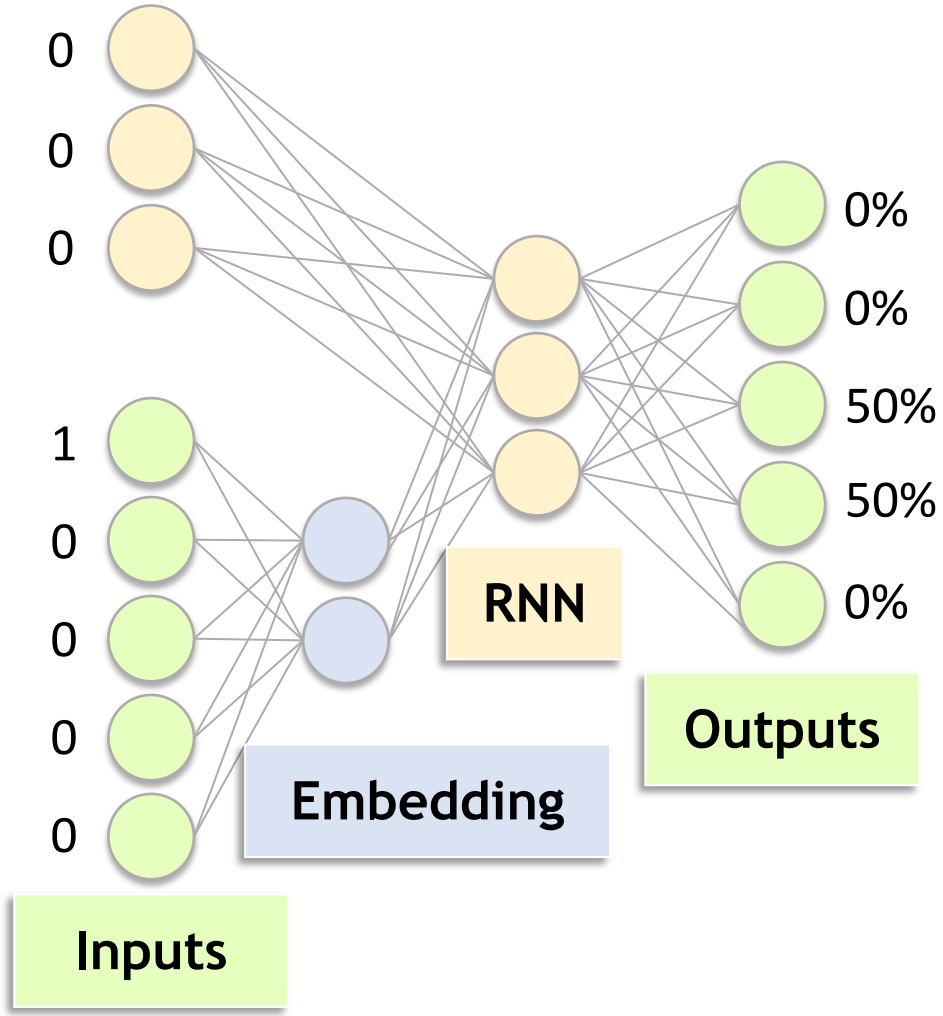
“Cats say ___.”

“Dogs say ___.”

DICTIONARY

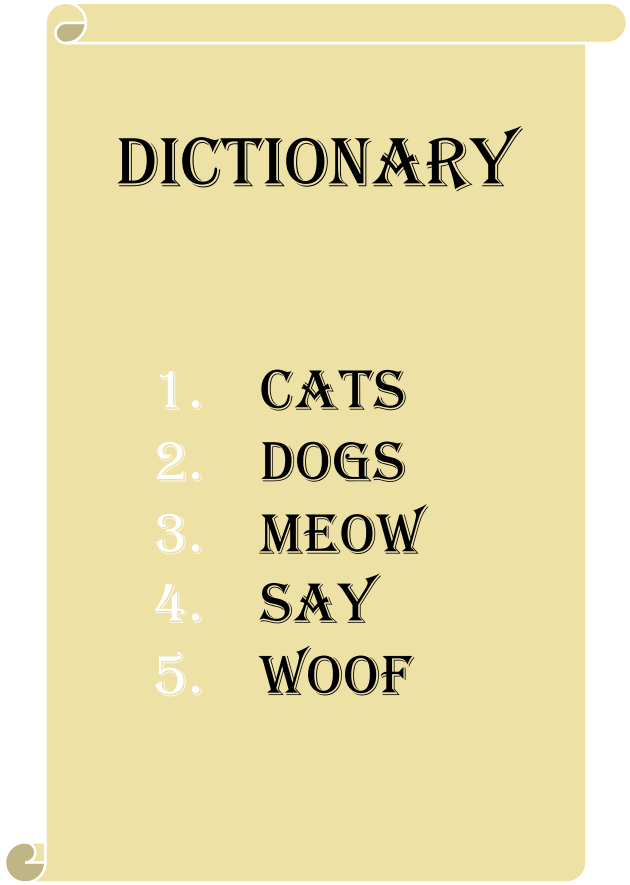
1. CATS
2. DOGS
3. MEOW
4. SAY
5. WOOF

Начальное слово и состояние

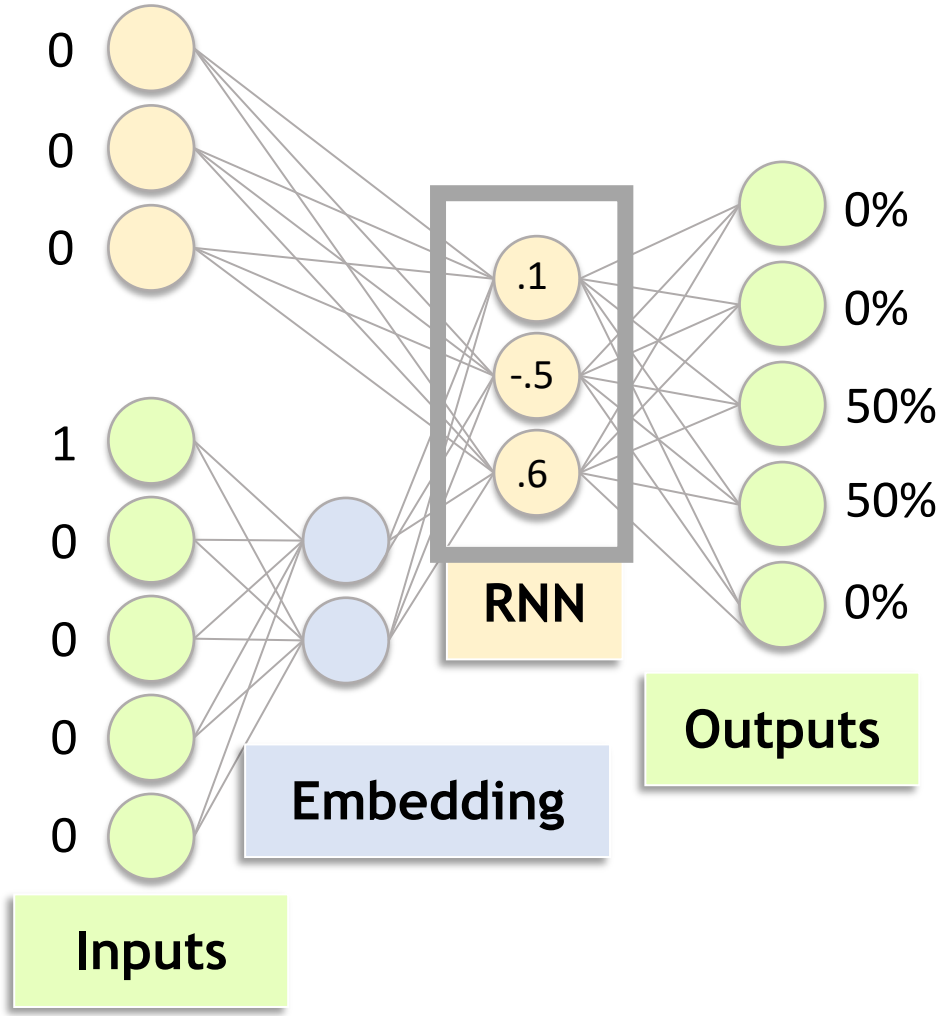


“Cats say ___.”

“Dogs say ___.”



Вычисление нового состояния



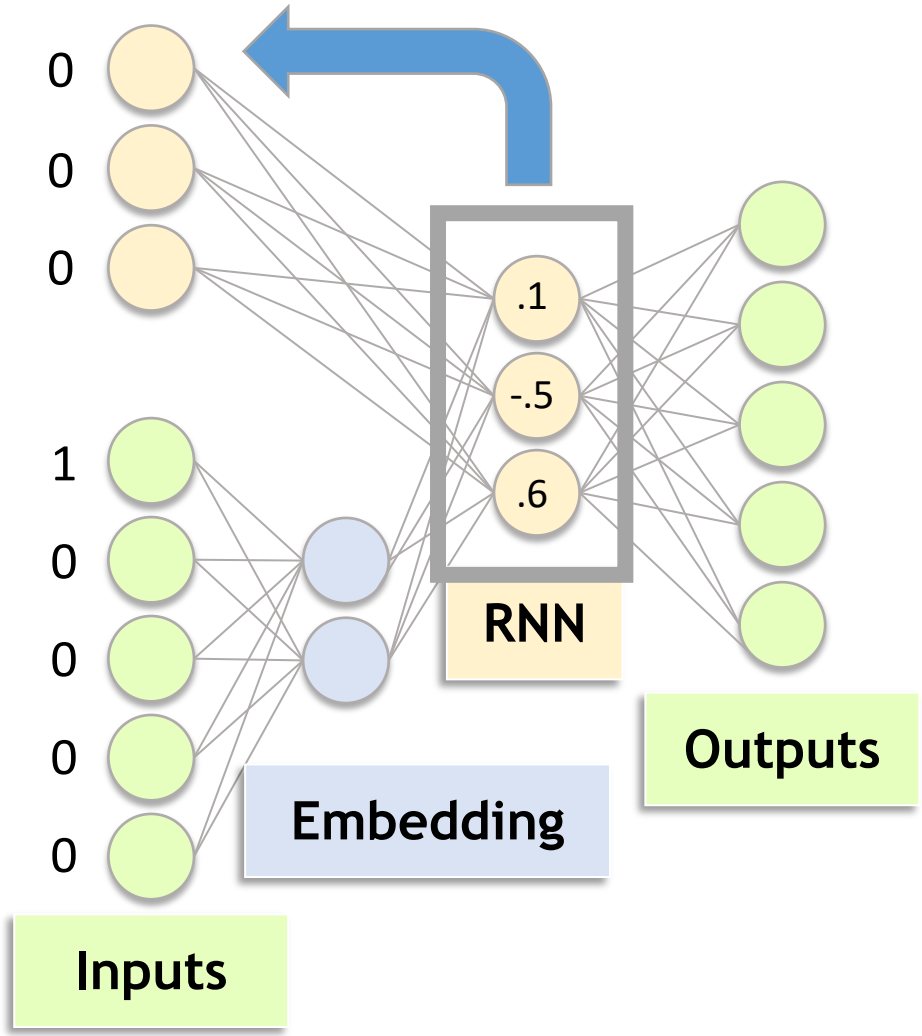
“Cats say ___.”

“Dogs say ___.”

DICTIONARY

1. CATS
2. DOGS
3. MEOW
4. SAY
5. WOOF

Запоминаем состояние



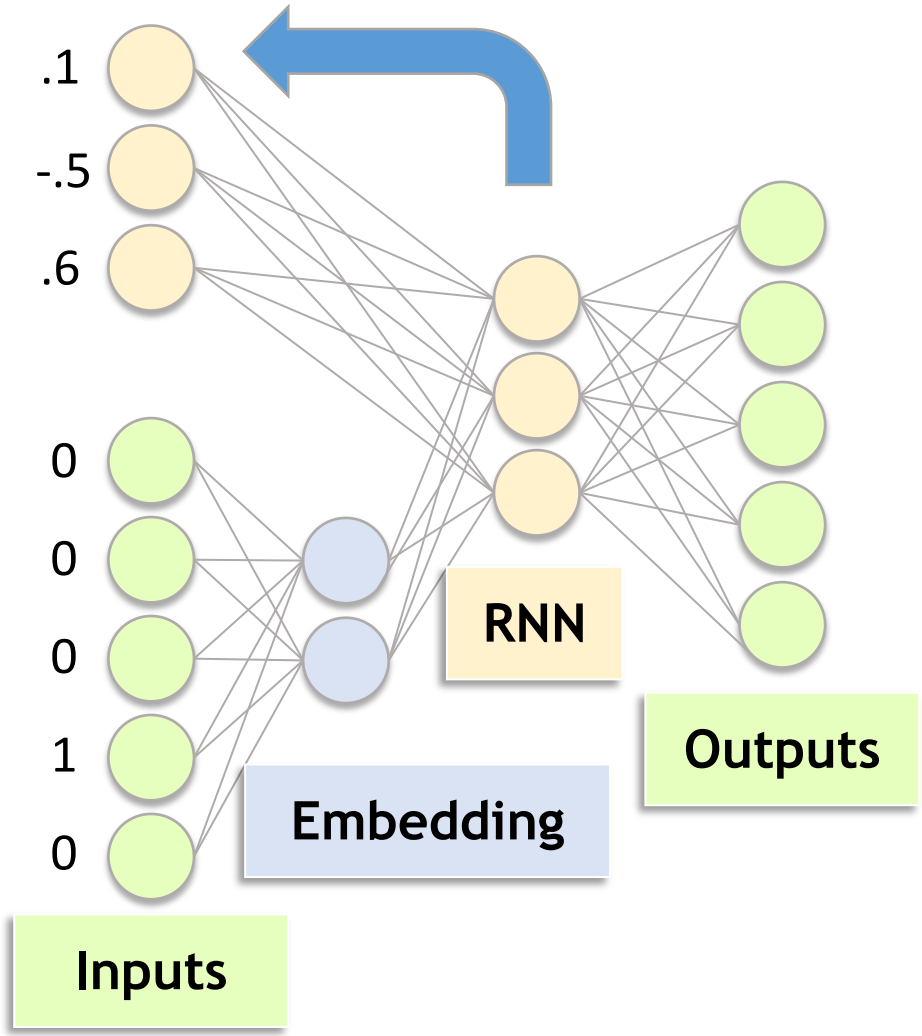
“Cats say ___.”

“Dogs say ___.”

DICTIONARY

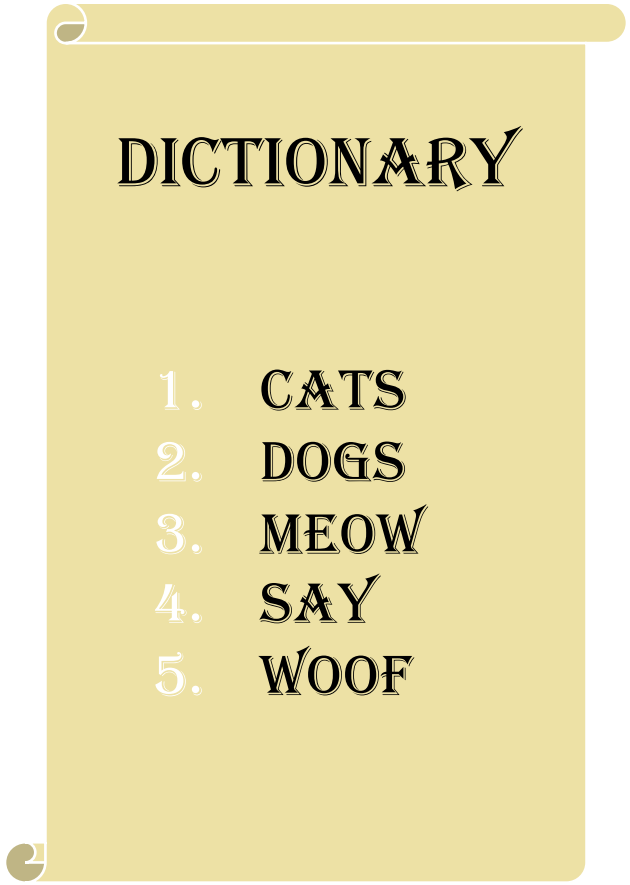
1. CATS
2. DOGS
3. MEOW
4. SAY
5. WOOF

Новое слово и старое состояние

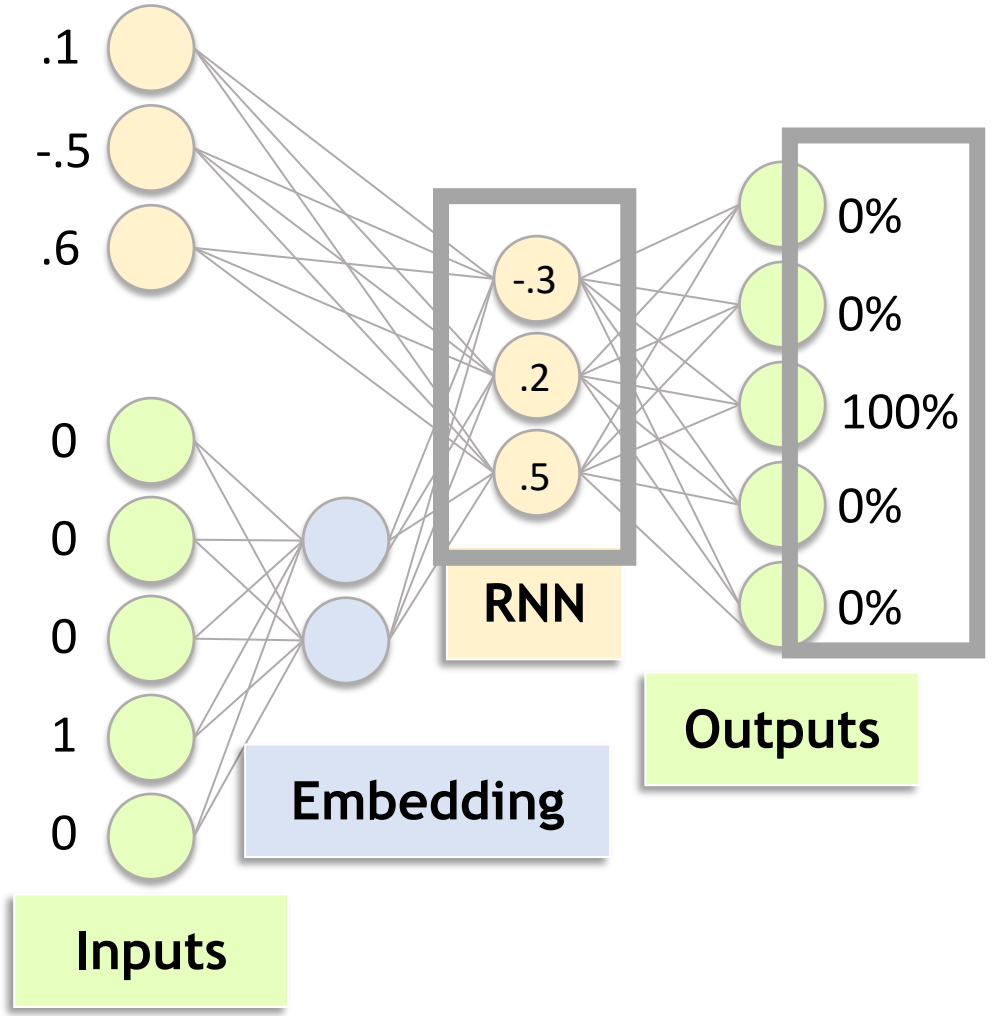


“Cats say ___.”

“Dogs say ___.”

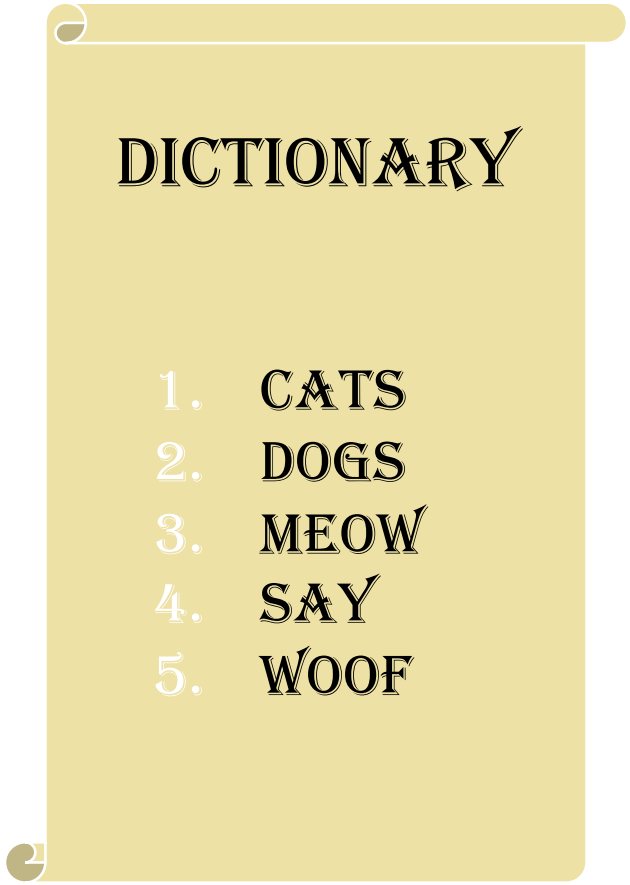


Получение ответа



“Cats say ___.”

“Dogs say ___.”



Seq2seq

- Для машинного перевода
- Состоит из двух частей, например две LSTM
- Можно добавить внимание