

Рекуррентные нейросети

Малева ТВ. Лекция 3



Определение

- Рекуррентные нейронные сети (Recurrent Neural Networks, RNN) – сети с обратными или перекрестными связями между различными слоями нейронов
- Типичные задачи (машинный перевод, анализ тональности, временные ряды)

RNN

- Один из нюансов работы с **нейронными сетями** (а также CNN) заключается в том, что они работают с предварительно заданными параметрами. Они принимают входные данные с фиксированными размерами и выводят результат, который также является фиксированным.
- RNN обеспечивают последовательности с вариативными длинами как для входа, так и для вывода.

RNN

Входные данные отмечены красным, нейронная сеть RNN — зеленым, а вывод — синим.

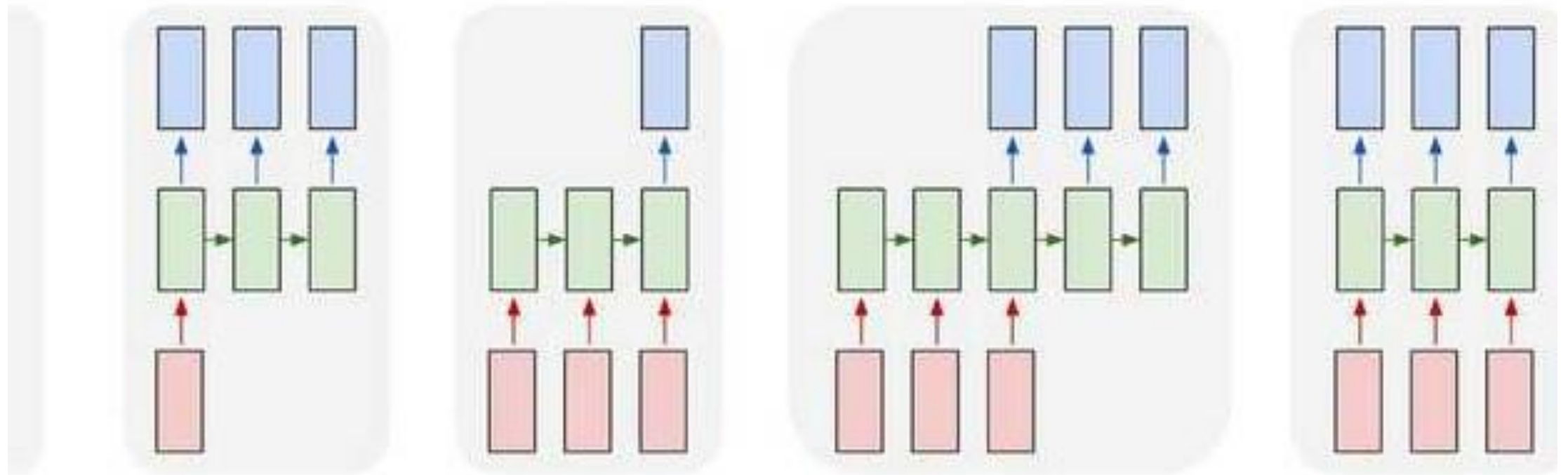
один к одному

один к многим

многие к одному

многие к многим

многие к многим



Динамическая система

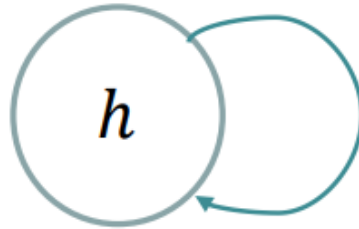
Классическая форма динамической системы:

$$h^{(t)} = f(h^{(t-1)}; \theta),$$

где $h^{(t)}$ – состояние системы в момент времени t ,

θ – множество параметров

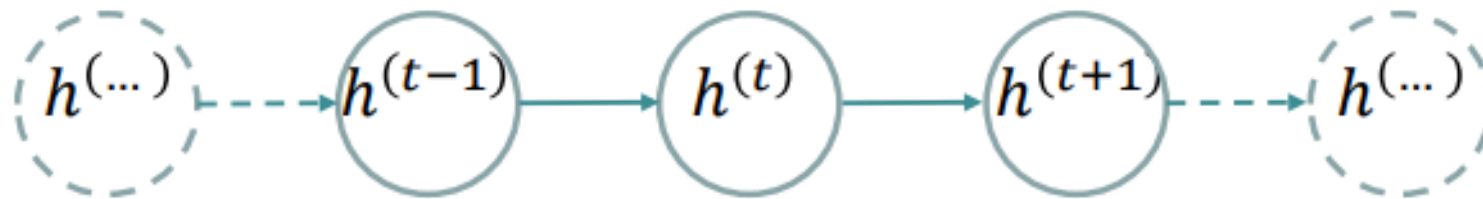
Система может быть представлена в виде рекуррентной сети



Приведенное уравнение является рекуррентным, поскольку состояние в каждый следующий момент времени зависит от состояния в предыдущий момент

Динамическая система для промежутка времени $[1, t]$

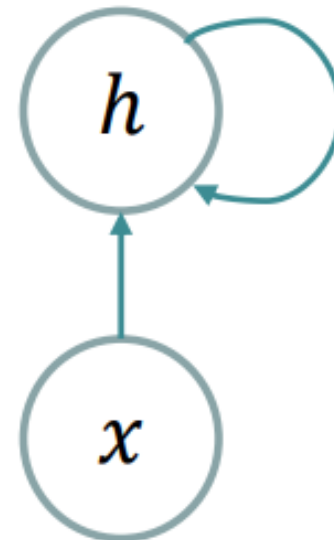
$$h^{(\tau)} = f(h^{(\tau-1)}; \theta) = f(f(h^{(\tau-2)}; \theta); \theta) = \dots = f(f(\dots f(h^{(1)}; \theta); \theta); \theta)$$



Динамическая система с внешним параметром $x(t)$

$$h^{(t)} = f(h^{(t-1)}; x^{(t)}; \theta)$$

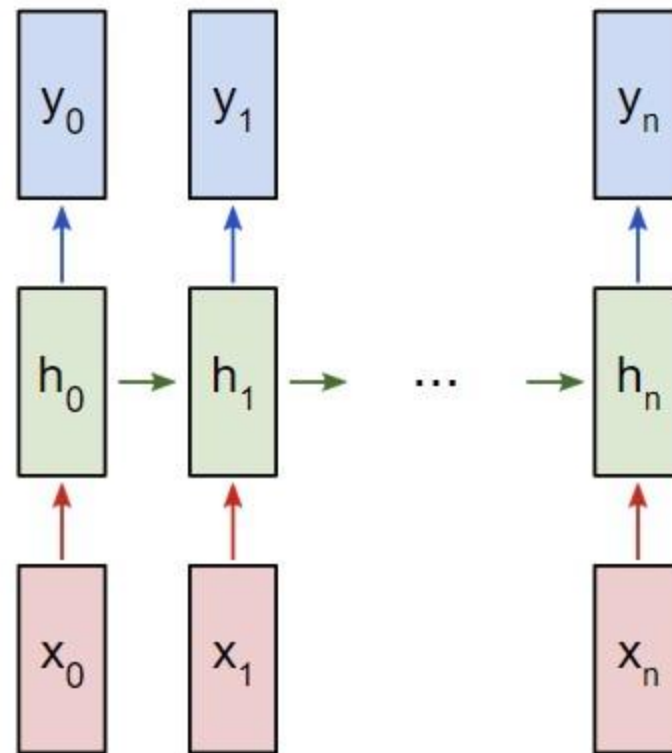
Соответствующая рекуррентная сеть имеет вид:



RNN

- Рекуррентные нейронные сети **RNN** работают путем итерированного обновления скрытого состояния h , которое является **вектором**, что также может иметь произвольный размер.
- Стоит учитывать, что на любом заданном этапе t :
 - Следующее скрытое состояние h_t подсчитывается при помощи предыдущего h_{t-1} и следующим вводом x_t ;
 - Следующий вывод y_t подсчитывается при помощи h_t .

RNN



RNN

- на каждом шаге RNN использует один и тот же вес.
- типичная классическая RNN использует только три набора параметров веса для выполнения требуемых подсчетов:
 1. W_{xh} используется для всех связей $x_t \rightarrow h_t$
 2. W_{hh} используется для всех связей $h_{t-1} \rightarrow h_t$
 3. W_{hy} используется для всех связей $h_t \rightarrow y_t$
- Для рекуррентной нейронной сети также используют два смещения:
 1. b_h добавляется при подсчете h_t
 2. b_y добавляется при подсчете y_t

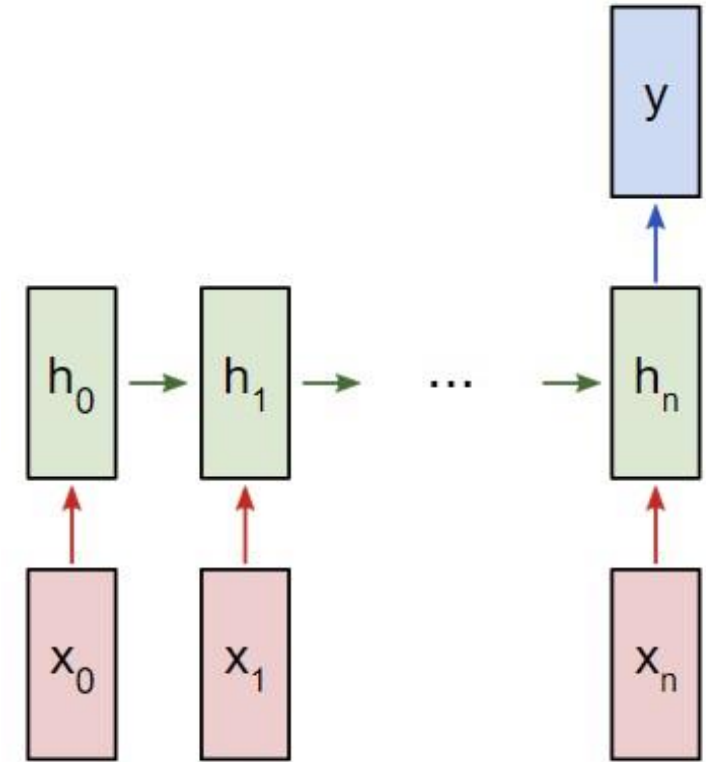
Вычисление весов скрытого состояния и смещения RNN

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$y_t = W_{hy}h_t + b_y$$

Анализ тональности

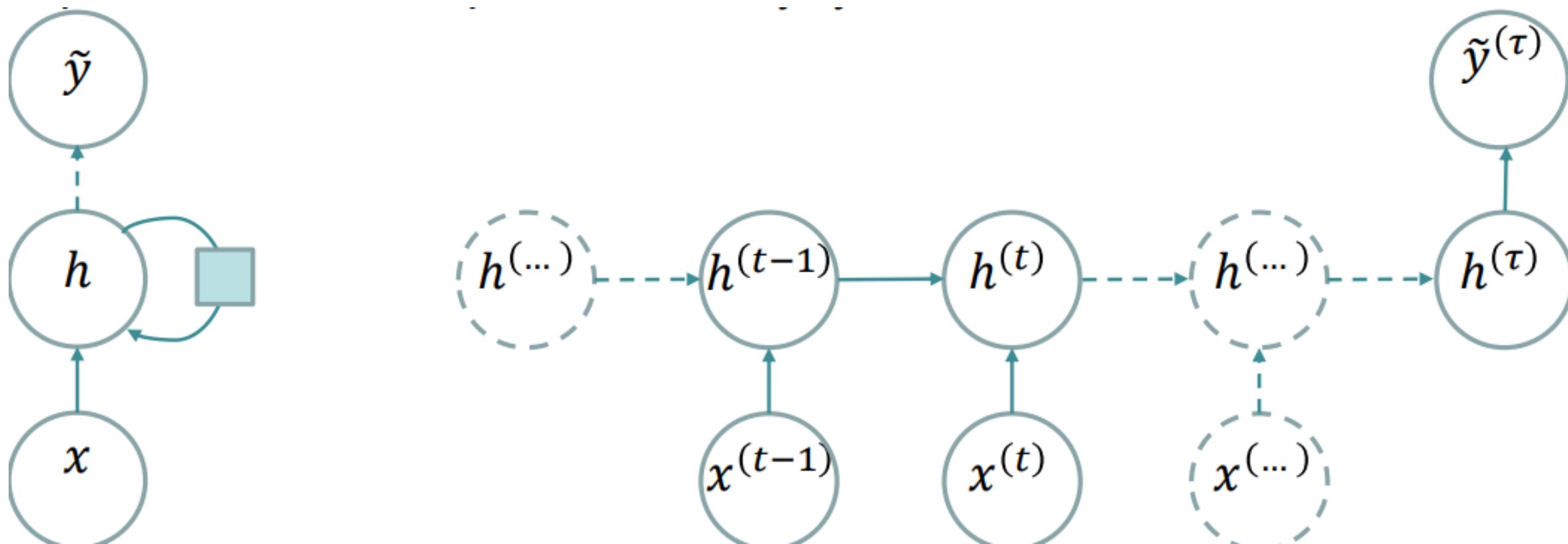
- Сеть Многие - к одному
- Каждый x_i будет вектором, представляющим определенное слово из текста. Вывод y будет вектором, содержащим два числа. Одно представляет позитивное настроение, а второе — негативное.
- Задача классификации



RNN

- на каждом шаге RNN использует один и тот же вес.
- типичная классическая RNN использует только три набора параметров веса для выполнения требуемых подсчетов:
 1. W_{xh} используется для всех связей $x_t \rightarrow h_t$
 2. W_{hh} используется для всех связей $h_{t-1} \rightarrow h_t$
 3. W_{hy} используется для всех связей $h_t \rightarrow y_t$
- Для рекуррентной нейронной сети также используют два смещения:
 1. b_h добавляется при подсчете h_t
 2. b_y добавляется при подсчете y_t

RNN многие к одному



Сети Элмана и Джордана

- Рекуррентная нейронная сеть с зависимостью скрытых нейронов является простейшей и называется рекуррентной сетью Элмана (Elman's network)
- Рекуррентная сеть, имеющая рекуррентные зависимости между выходным элементом текущего момента и скрытым элементом следующего момента, называется сетью Джордана (Jordan's network)

Уравнения, описывающие сеть Элмана

Уравнения, описывающие внутреннее состояние и выход сети, которая получена в результате развертывания сети Элмана во времени, имеют следующий вид:

$$\begin{aligned}a^{(t)} &= Ux^{(t)} + Wh^{(t-1)} + b, \\h^{(t)} &= f(a^{(t)}) = f(Ux^{(t)} + Wh^{(t-1)} + b), \\o^{(t)} &= Vh^{(t)} + c, \quad \tilde{y}^{(t)} = g(o^{(t)}) = g(Vh^{(t)} + c),\end{aligned}$$

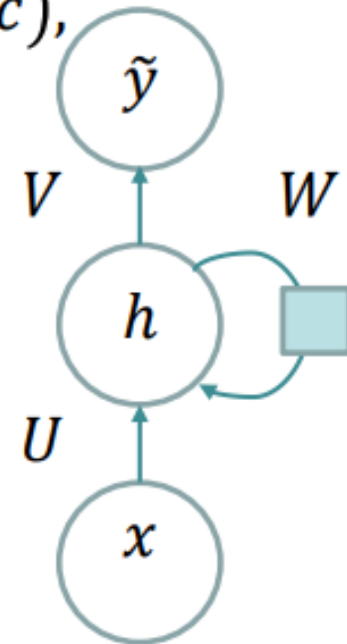
где U, W, V – матрицы весов,

b, c – вектора сдвига,

$h^{(t)}$ – вектор скрытых переменных в момент t (при обработке примера с номером t из заданной входной последовательности),

$\tilde{y}^{(t)}$ – выход сети в момент t ,

$f(\cdot), g(\cdot)$ – функции активации



Обучение сети Элмана

$$J = \frac{1}{N} \sum_{n=1}^N \sum_{t=1}^{\tau_n} d\left(\tilde{y}_n^{(t)}, y_n^{(t)}\right) \rightarrow \min_{U, W, V} \quad ,$$

где N – количество входных последовательностей,
 τ_n – количество элементов последовательности с номером n ,
 $y_n^{(t)}$ – реальный выход (разметка) в момент t при
рассмотрении последовательности с номером n ,
 $\tilde{y}_n^{(t)}$ – выход сети в момент t при получении входной
последовательности с номером n ,
 $d\left(\tilde{y}_n^{(t)}, y_n^{(t)}\right)$ – мера сходства разметки и выхода сети
(Евклидово расстояние или кросс-энтропия)

Метод обратного распространения ошибки с разворачиванием сети во времени

- Рекуррентную нейронную сеть можно развернуть во времени, тем самым, представив ее в виде сети с прямым распространением сигнала
- Для обучения параметров сети можно применить метод обратного распространения ошибки с разворачиванием сети во времени (backpropagation through time)

Метод обратного распространения ошибки с разворачиванием сети во времени

1. Прямой проход по развернутой во времени сети (проход слева направо по развернутой во времени сети)

- Выполняется вычисление скрытых состояний и выходов развернутой сети, а также градиентов функций активации.
- Сложность вычислений пропорциональна длине входной последовательности $O(\tau)$
- Распараллеливание вычислений выполнить нельзя, поскольку каждое следующее внутреннее состояние системы зависит от предыдущего

Метод обратного распространения ошибки с разворачиванием сети во времени

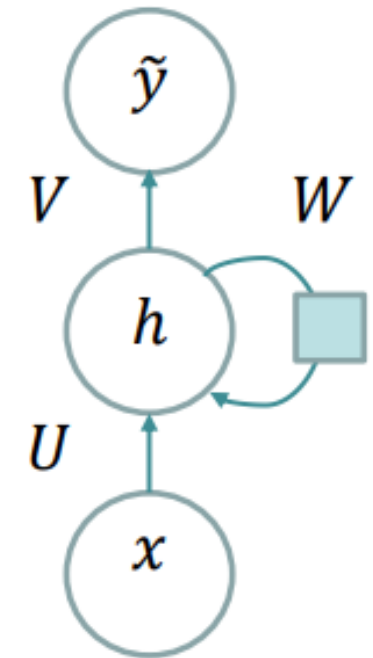
2. Вычисление значения целевой функции и градиента этой функции

3. Обратный проход развернутой во времени сети (проход справа налево по развернутой во времени сети).

Выполняется вычисление ошибки и корректировка весов сети

Преобразования на рекуррентном слое

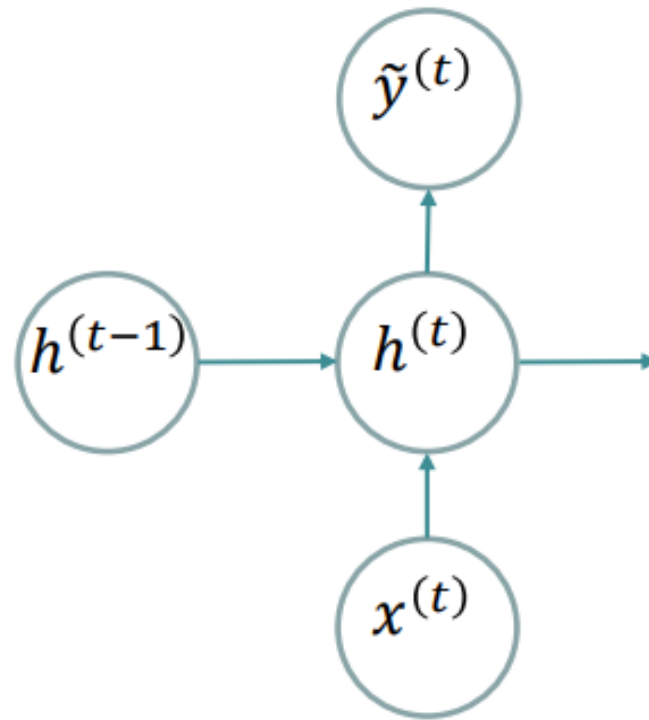
- Вычисления в большинстве типовых рекуррентных сетей можно разложить на три блока параметров и соответствующих им преобразования:
 1. Преобразование входа в скрытое состояние
 2. Преобразование от предыдущего скрытого состояния в следующее скрытое состояние
 3. Преобразование скрытого состояния в выход



Типы рекуррентных сетей

Обычная рекуррентная сеть (a conventional RNN)

$$h^{(t)} = f(Ux^{(t)} + Wh^{(t-1)} + b), \quad \tilde{y}^{(t)} = g(Vh^{(t)} + c)$$

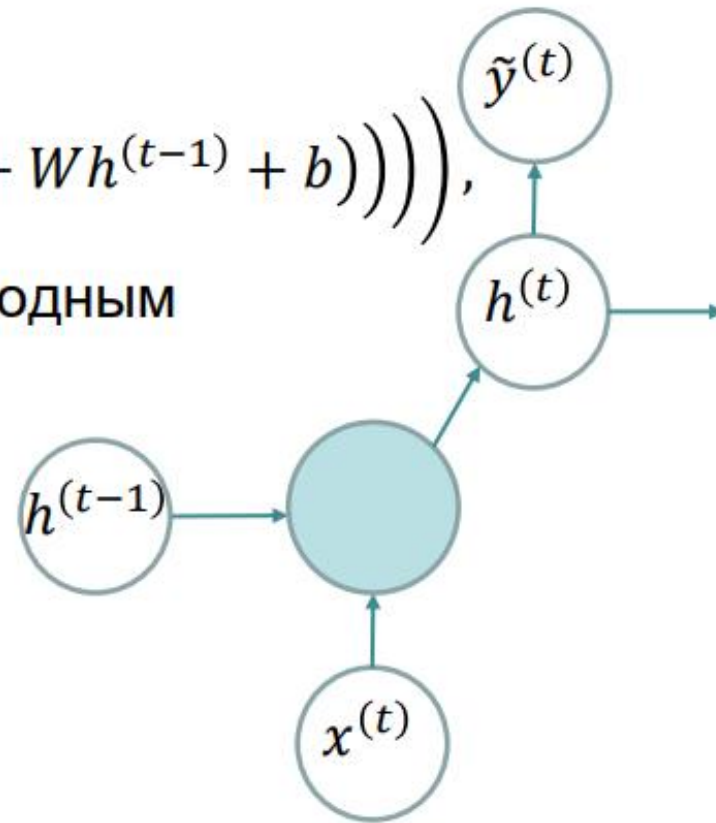


Типы рекуррентных сетей

Рекуррентная сеть с глубоким преобразованием входного сигнала в скрытый (Deep Transition RNN, DT-RNN)

$$h^{(t)} = f(Ux^{(t)} + Wh^{(t-1)} + b)$$
$$= \varphi_L \left(U_L^T \varphi_{L-1} \left(U_{L-1}^T \varphi_{L-2} \left(\dots \varphi_1 (Ux^{(t)} + Wh^{(t-1)} + b) \right) \right) \right),$$

где L – количество слоев сети между входным и скрытым слоями

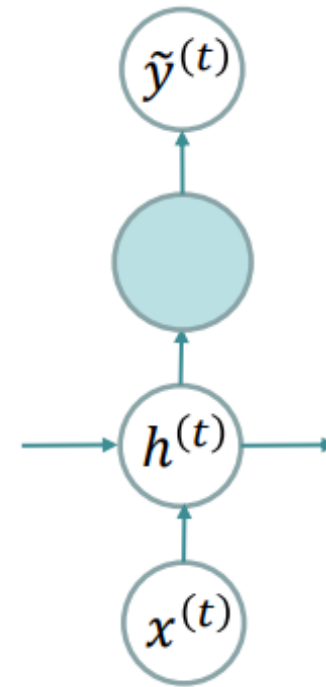


Типы рекуррентных сетей

Рекуррентная сеть с глубоким преобразованием скрытого сигнала в выходной (Deep Output RNN, DO-RNN)

$$\tilde{y}^{(t)} = g(Vh^{(t)} + c) = \psi_L \left(V_L^T \psi_{L-1} \left(V_{L-1}^T \psi_{L-2} \left(\dots \psi_1 (Vh^{(t)} + c) \right) \right) \right),$$

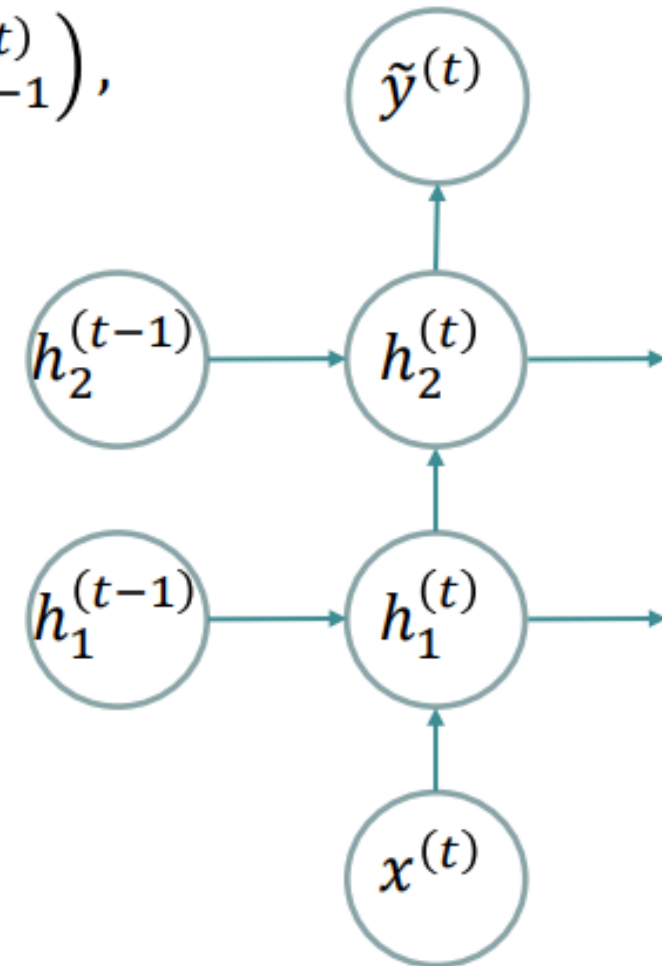
где L – количество слоев сети между скрытым и выходным слоями



Стек рекуррентных слоев

$$h_l^{(t)} = f_l \left(W_l^T h_l^{(t-1)} + U_l^T h_{l-1}^{(t)} \right),$$

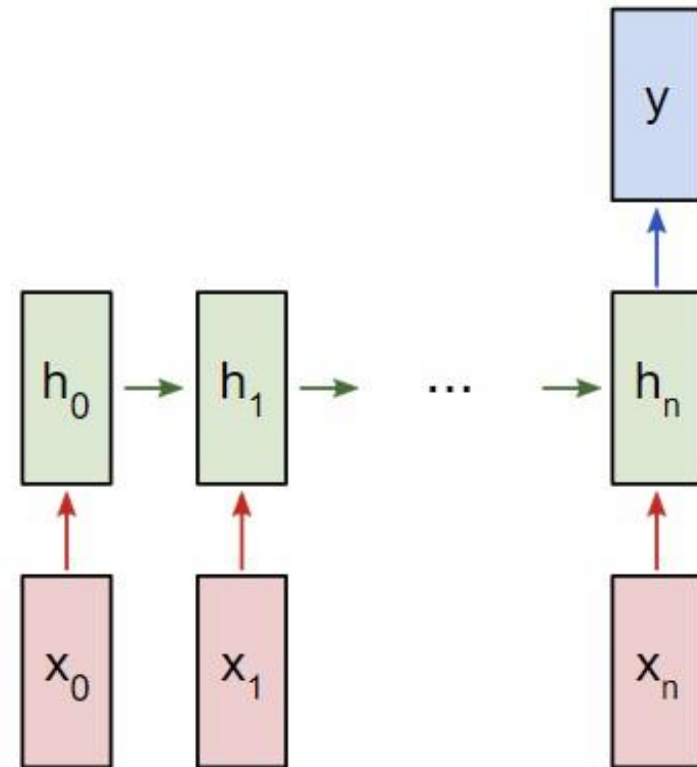
где $h_l^{(t)}$ – скрытое состояние системы
на слое с номером l в момент времени t



Анализ тональности

- Датасет

```
train_data = {  
    'good': True,  
    'bad': False,  
    'happy': True,  
    'sad': False,  
    'not good': False,  
    'not bad': True,  
    'not happy': False,  
    'not sad': True,  
    'very good': True,  
    'very bad': False,  
    'very happy': True,  
    'very sad': False,  
    'i am happy': True,  
}
```



Загрузка тренировочных словарей

- `from data import train_data, test_data`
- `# Создание словаря`
- `vocab = list(set([w for text in train_data.keys() for w in text.split(' ')]))`
- `vocab_size = len(vocab)`
- `print('%d unique words found' % vocab_size)`

Индексирование уникальных слов

- `word_to_idx = { w: i for i, w in enumerate(vocab) }`
- `idx_to_word = { i: w for i, w in enumerate(vocab) }`
- `print(word_to_idx['good'])` # 16 (это может измениться)
- `print(idx_to_word[0])` # грустно (это может измениться)

Унитарное кодирование предложений

```
def createInputs(text):  
    """  
    унитарный вектор имеет форму (vocab_size, 1)  
    """  
    inputs = []  
    for w in text.split(' '):  
        v = np.zeros((vocab_size, 1))  
        v[word_to_idx[w]] = 1  
        inputs.append(v)  
  
    return inputs
```

Классическая RNN. Инициализация

```
class RNN:
```

```
    # Классическая рекуррентная нейронная сеть
```

```
    def __init__(self, input_size, output_size, hidden_size=64):
```

```
        # Начальный вес
```

```
        self.Whh = np.random.randn(hidden_size, hidden_size) /  
1000
```

```
        self.Wxh = np.random.randn(hidden_size, input_size) / 1000
```

```
        self.Why = np.random.randn(output_size, hidden_size) /  
1000
```

```
        # Смещения
```

```
        self.bh = np.zeros((hidden_size, 1))
```

```
        self.by = np.zeros((output_size, 1))
```

Классическая
RNN. Прямое
распространение

```
class RNN:
```

```
    def forward(self, inputs):
```

```
        """
```

Выполнение передачи нейронной сети
при помощи входных данных

Возвращение результатов вывода и
скрытого состояния

Вывод - это массив одного унитарного
вектора с формой (input_size, 1)

```
        """
```

Классическая RNN. Прямое распространение

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$y_t = W_{hy}h_t + b_y$$

Классическая
RNN. Прямое распространение

```
class RNN:
```

```
    def forward(self, inputs):
```

```
        h = np.zeros((self.Whh.shape[0], 1))
```

```
        self.last_inputs = inputs
```

```
        self.last_hs = { 0: h }
```

```
        # Выполнение каждого шага в нейронной сети RNN
```

```
        for i, x in enumerate(inputs):
```

```
            h = np.tanh(self.Wxh @ x + self.Whh @ h + self.bh)
```

```
            self.last_hs[i + 1] = h
```

```
            y = self.Why @ h + self.by # выход, @ -  
python 3.6, __matmul__
```

```
            return y, h
```

Работа сети RNN

- `def softmax(xs):`
 - `return np.exp(xs) / sum(np.exp(xs))`
- `# Инициализация нашей рекуррентной нейронной сети RNN`
- `rnn = RNN(vocab_size, 2)`
- `inputs = createInputs('Хороший день')`
- `out, h = rnn.forward(inputs)`
- `probs = softmax(out)`

Обучение RNN

- p_c является предсказуемой вероятностью рекуррентной нейронной сети для класса correct (позитивный или негативный)
- Если для примера истинная метка True, то берем вероятность нейрона соответствующего True, иначе False

$$L = -\ln(p_c)$$

Обучение RNN

- y — необработанные входные данные нейронной сети;
- p — конечная вероятность: $p = \text{softmax}(y)$;
- c — истинная метка определенного образца текста, так называемый «правильный» класс;
- L — потеря перекрестной энтропии: $L = -\ln(p_c)$;
- W_{xh} , W_{hh} и W_{hy} — три матрицы веса в рассматриваемой нейронной сети;
- b_h и b_y — два вектора смещения в рассматриваемой рекуррентной нейронной сети **RNN**

Вычисление градиентов

$$L = -\ln(p_c) = -\ln(\text{softmax}(y_c))$$

Вычисление градиентов

$$L = -\ln(p_c) = -\ln(\text{softmax}(y_c))$$

$$\frac{\partial L}{\partial y_i} = \begin{cases} p_i & \text{if } i \neq c \\ p_i - 1 & \text{if } i = c \end{cases}$$

Вычисление градиентов

$$\frac{\partial L}{\partial y_i} = \begin{cases} p_i & \text{if } i \neq c \\ p_i - 1 & \text{if } i = c \end{cases}$$

если $p = [0.2, 0.8]$, а корректным классом является $c = 0$, то конечным результатом будет значение $[-0.8, 0.8]$.

Инициализация градиента в процессе обучения

- # Цикл для каждого примера тренировки
- for x, y in train_data.items():
 - inputs = createInputs(x)
 - target = int(y)
- # Прямое распространение
- out, _ = rnn.forward(inputs)
- probs = softmax(out)
- # Создание dL/dy
- d_L_d_y = probs
- d_L_d_y[target] -= 1
- # Обратное распространение
- rnn.backprop(d_L_d_y)

Вычисление градиентов

$$\frac{\partial L}{\partial W_{hy}} = \frac{\partial L}{\partial y} * \frac{\partial y}{\partial W_{hy}}$$

$$y = W_{hy}h_n + b_y$$

Вычисление градиентов, h_n - скрытое состояние, 1 - первый слой

$$\frac{\partial y}{\partial W_{hy}} = h_n$$

$$\frac{\partial L}{\partial W_{hy}} = \boxed{\frac{\partial L}{\partial y} h_n}$$

Вычисление градиентов, h_n - скрытое состояние, 1 - первый слой

$$\frac{\partial y}{\partial b_y} = 1$$

$$\frac{\partial L}{\partial b_y} = \boxed{\frac{\partial L}{\partial y}}$$

Классическая RNN. Обратное распространение

```
class RNN:
    def backprop(self, d_y, learn_rate=2e-2):
        """
        d_y (dL/dy) имеет форму (output_size, 1).
        """
        n = len(self.last_inputs)

        # Подсчет dL/dWhy и dL/dby.
        d_Why = d_y @ self.last_hs[n].T
        d_by = d_y
```

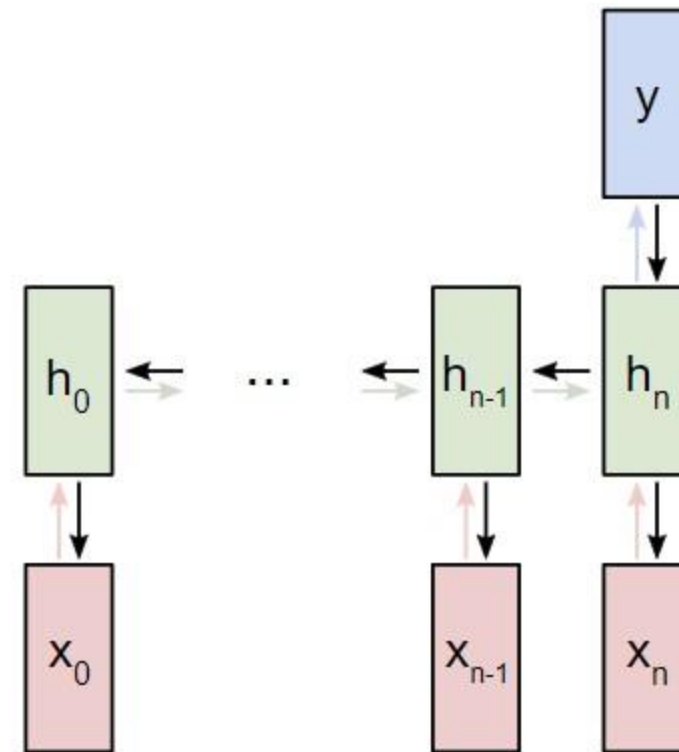
Классическая RNN. Обратное распространение

Изменение W_{xh} влияет не только на каждый h_t , но и на все y , что приводит к изменениям в L . Для того, чтобы полностью подсчитать градиент W_{xh} , необходимо провести обратное распространение через все временные шаги. Его и называют **Обратным распространением во времени** (Backpropagation Through Time (BPTT))

$$\frac{\partial L}{\partial W_{xh}} = \frac{\partial L}{\partial y} \sum_t \frac{\partial y}{\partial h_t} * \frac{\partial h_t}{\partial W_{xh}}$$

Обратное распространение во времени

$$\frac{\partial L}{\partial W_{xh}} = \frac{\partial L}{\partial y} \sum_t \frac{\partial y}{\partial h_t} * \frac{\partial h_t}{\partial W_{xh}}$$



Вычисление градиентов

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$\frac{d \tanh(x)}{dx} = 1 - \tanh^2(x)$$

$$\frac{\partial h_t}{\partial W_{xh}} = \boxed{(1 - h_t^2)x_t}$$

Вычисление градиентов

$$\frac{\partial h_t}{\partial W_{hh}} = \boxed{(1 - h_t^2)h_{t-1}}$$

$$\frac{\partial h_t}{\partial b_h} = \boxed{(1 - h_t^2)}$$

$$\begin{aligned}\frac{\partial y}{\partial h_t} &= \frac{\partial y}{\partial h_{t+1}} * \frac{\partial h_{t+1}}{\partial h_t} \\ &= \frac{\partial y}{\partial h_{t+1}} (1 - h_t^2) W_{hh}\end{aligned}$$

$$\frac{\partial y}{\partial h_n} = W_{hy}$$


```
def backprop(self, d_y, learn_rate=2e-2):
```

```
    n = len(self.last_inputs)
```

```
    # Вычисление dL/dWhy и dL/dby.
```

```
    d_Why = d_y @ self.last_hs[n].T
```

```
    d_by = d_y
```

```
    # Инициализация dL/dWhh, dL/dWxh, и dL/dbh к нулю.
```

```
    d_Whh = np.zeros(self.Whh.shape)
```

```
    d_Wxh = np.zeros(self.Wxh.shape)
```

```
    d_bh = np.zeros(self.bh.shape)
```

```
    # Вычисление dL/dh для последнего h.
```

```
    d_h = self.Why.T @ d_y
```

```
def backprop(self, d_y, learn_rate=2e-2):
```

```
    for t in reversed(range(n)):
```

```
        # dL/dh * (1 - h^2)
```

```
        temp = ((1 - self.last_hs[t + 1] ** 2) * d_h)
```

$$\frac{\partial L}{\partial h} = \frac{\partial L}{\partial y} * \frac{\partial y}{\partial h}$$

```
        # dL/db = dL/dh * (1 - h^2)
```

```
        d_bh += temp
```

```
        # dL/dWhh = dL/dh * (1 - h^2) * h_{t-1}
```

```
        d_Whh += temp @ self.last_hs[t].T
```

```
        # dL/dWxh = dL/dh * (1 - h^2) * x
```

```
        d_Wxh += temp @ self.last_inputs[t].T
```

```
        # dL/dh = dL/dh * (1 - h^2) * Whh
```

```
        d_h = self.Whh @ temp
```

```
def backprop(self, d_y, learn_rate=2e-2):
```

```
    for d in [d_Wxh, d_Whh, d_Why, d_bh, d_by]:  
        np.clip(d, -1, 1, out=d)
```

```
        # Обновляем вес и смещение с использованием градиентного  
        спуска.
```

```
        self.Whh -= learn_rate * d_Whh
```

```
        self.Wxh -= learn_rate * d_Wxh
```

```
        self.why -= learn_rate * d_Why
```

```
        self.bh -= learn_rate * d_bh
```

```
        self.by -= learn_rate * d_by
```

Тестирование RNN

```
def processData(data, backprop=True):  
    items = list(data.items())  
    random.shuffle(items)  
    loss = 0  
    num_correct = 0
```

Тестирование RNN

```
for x, y in items:
    inputs = createInputs(x)
    target = int(y)
    out, _ = rnn.forward(inputs) # Прямое распределение
    probs = softmax(out)
    loss -= np.log(probs[target]) # Вычисление потери / точности
    num_correct += int(np.argmax(probs) == target)
    if backprop:
        d_L_d_y = probs # Создание dL/dy
        d_L_d_y[target] -= 1
        rnn.backprop(d_L_d_y) # Обратное распределение

return loss / len(data), num_correct / len(data)
```

Обучение RNN

```
for epoch in range(1000):  
    train_loss, train_acc = processData(train_data)  
  
    if epoch % 100 == 99:  
        print('--- Epoch %d' % (epoch + 1))  
        print('Train:\tLoss %.3f | Accuracy: %.3f' % (train_loss, train_acc))  
  
    test_loss, test_acc = processData(test_data, backprop=False)  
    print('Test:\tLoss %.3f | Accuracy: %.3f' % (test_loss, test_acc))
```

RNN в KERAS

- `Keras.layers.SimpleRNN` - RNN, в которой выходные данные из предыдущего временного шага должны передаваться на следующий временной шаг.

```
tf.keras.layers.SimpleRNN(  
    units,  
    activation="tanh",  
    use_bias=True,  
    kernel_initializer="glorot_uniform",  
    recurrent_initializer="orthogonal",  
    bias_initializer="zeros",  
    kernel_regularizer=None,  
    recurrent_regularizer=None,  
    bias_regularizer=None,  
    activity_regularizer=None,  
    kernel_constraint=None,  
    recurrent_constraint=None,  
    bias_constraint=None,  
    dropout=0.0,  
    recurrent_dropout=0.0,  
    return_sequences=False,  
    return_state=False,  
    go_backwards=False,  
    stateful=False,  
    unroll=False,  
    **kwargs  
)
```

RNN В KERAS

SimpleRNN

- **units**: положительное целое число, размерность выходного пространства.
- **activation**: функция активации. По умолчанию: гиперболический тангенс (\tanh). Если вы передадите `None`, активация не будет применяться (т.е., «линейная» активация :) $a(x) = x$.
- **use_bias** : Boolean, (по умолчанию `True`), использует ли слой вектор смещения.
- **kernel_initializer** : инициализатор матрицы весов, используемый для линейного преобразования входных данных. По умолчанию: `glorot_uniform`.
- **recurrent_initializer** : Инициализатор для `recurrent_kernel` матрицы весов, используемый для линейного преобразования рекуррентного состояния. По умолчанию: `orthogonal`.
- **bias_initializer** : Инициализатор вектора смещения. По умолчанию: `zeros`.
- **kernel_regularizer** : функция регуляризатора, примененная к матрице весов. По умолчанию: `None`.
- **recurrent_regularizer** : функция регуляризатора, примененная к `recurrent_kernel` матрице весов. По умолчанию: `None`.

SimpleRNN

- **bias_regularizer** : функция регуляризатора, примененная к вектору смещения. По умолчанию: None.
- **activity_regularizer** : функция регуляризатора, применяемая к выходу слоя (его «активация»). По умолчанию: None.
- **kernel_constraint** : функция ограничения, применяемая к матрице весов. По умолчанию: None.
- **recurrent_constraint** : функция ограничения, применяемая к recurrent_kernel матрице весов. По умолчанию: None.
- **bias_constraint** : функция ограничения, применяемая к вектору смещения. По умолчанию: None.
- **dropout** : плавающее в диапазоне от 0 до 1. Доля единиц, отбрасываемых для линейного преобразования входных данных. По умолчанию: 0.
- **recurrent_dropout** : плавающее в диапазоне от 0 до 1. Доля единиц, которые нужно отбросить для линейного преобразования повторяющегося состояния. По умолчанию: 0.

SimpleRNN

- **return_sequences** : логический. Следует ли возвращать последний вывод в выходной последовательности или полную последовательность. По умолчанию: False.
- **return_state** : логическое. Возвращать ли последнее состояние в дополнение к выходу. По умолчанию: False
- **go_backwards** : Boolean (по умолчанию False). Если True, обработать входную последовательность в обратном порядке и вернуть обратную последовательность.
- **Stateful** : Boolean (по умолчанию - False). Если True, последнее состояние для каждой выборки с индексом i в пакете будет использоваться в качестве начального состояния для выборки с индексом i в следующем пакете.
- **unroll**: Boolean (по умолчанию False). Если True, сеть будет развернута, иначе будет использоваться символический цикл. Развертывание может ускорить RNN, хотя, как правило, требует больше памяти. Развертывание подходит только для коротких последовательностей.

SimpleRNN. Аргументы вызова

- **inputs**: 3D тензор с формой [batch, timesteps, feature].
- **mask**: двоичный тензор формы, [batch, timesteps]указывающий, следует ли замаскировать данный временной шаг.
- **training**: логическое значение Python, указывающее, должен ли слой вести себя в режиме обучения или в режиме вывода. Этот аргумент передается ячейке при ее вызове. Это актуально, только если используется dropout или recurrent_dropout.
- **initial_state** : список тензоров начального состояния, передаваемых при первом вызове ячейки.

Предсказать значения временного ряда

- Моделируем периодические данные
- $N = 1000$
- $T_p = 800$
-
- `t=np.arange(0,N)`
- `x=np.sin(0.02*t)+2*np.random.rand(N)`
- `df = pd.DataFrame(x)`
- `values=df.values`
- `train,test = values[0:Tp:], values[Tp:N,:]`

Подготавливаем
данные

- Если шаг $t=1$
- $x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$

x	y
1	2
2	3
3	4
4	5
..	
9	10

Подготавливаем
данные

- Если шаг $t=3$
- $x = [1,2,3,4,5,6,7,8,9,10]$

x	y
1,2,3	4
2,3,4	5
3,4,5	6
4,5,6	7
...	
7,8,9	10

Подготавливаем данные

- `step = 4`
- `test = np.append(test,np.repeat(test[-1:],step))`
- `train = np.append(train,np.repeat(train[-1:],step))`
- **def** `convertToMatrix(data, step):`
- `X, Y = [], []`
- **for** `i in range(len(data)-step):`
- `d=i+step`
- `X.append(data[i:d,:])`
- `Y.append(data[d,:])`
- **return** `np.array(X), np.array(Y)`
- `trainX,trainY =convertToMatrix(train,step)`
- `testX,testY =convertToMatrix(test,step)`

Подготавливаем данные

- `trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))`
- `testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))`
- `trainX.shape`
- `(800, 1, 4)`

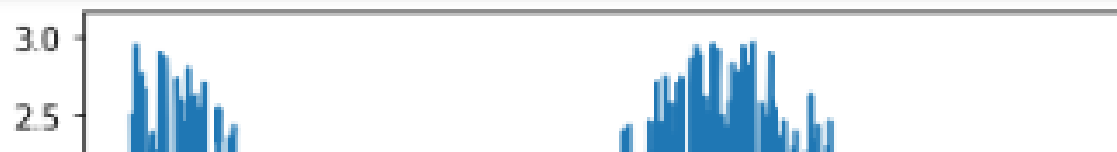
Строим сеть

- **from keras.models import Sequential**
- **from keras.layers import Dense, SimpleRNN**

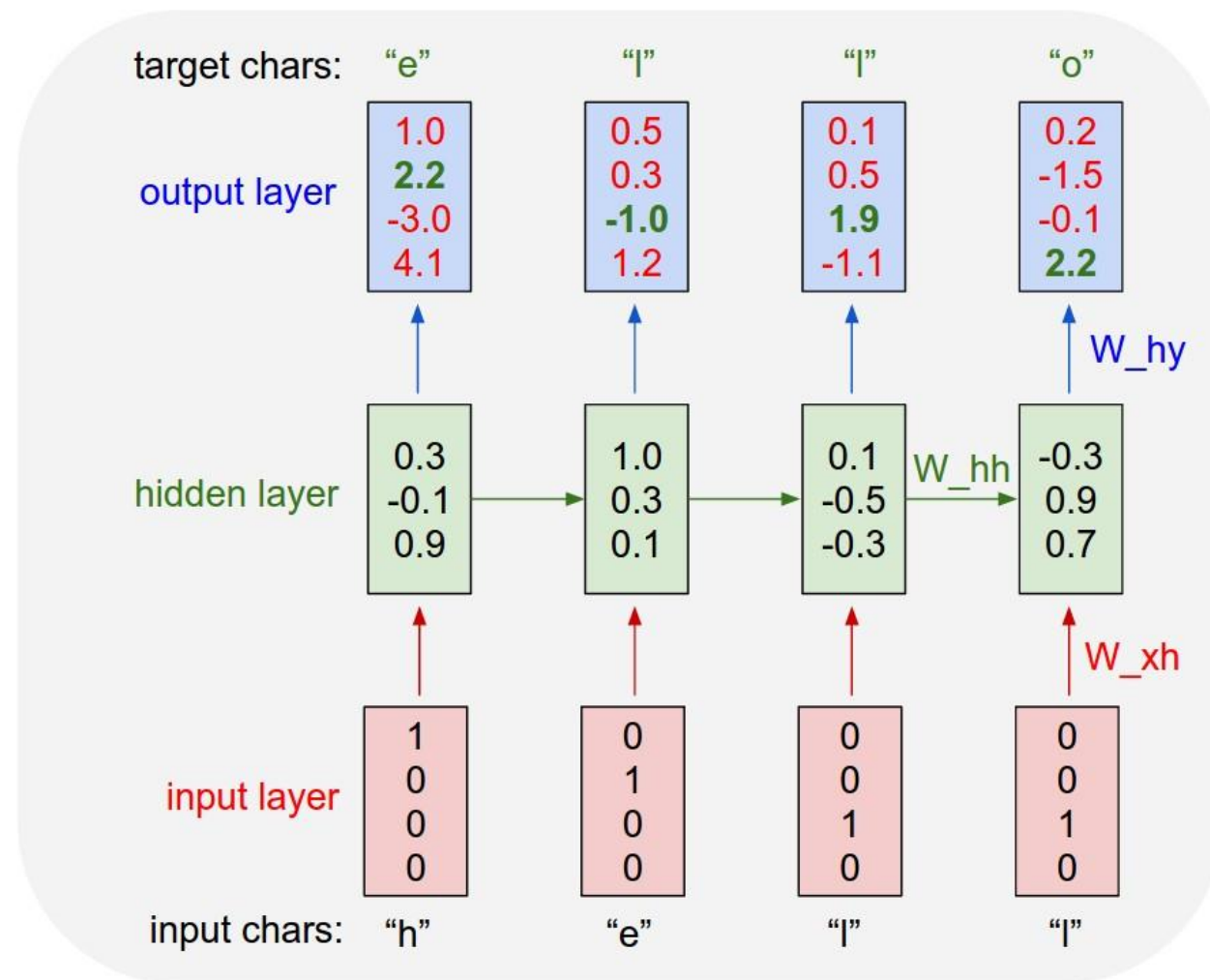
```
40 model = Sequential()
41 model.add(SimpleRNN(units=32, input_shape=(1,step), activation="relu"))
42 model.add(Dense(8, activation="relu"))
43 model.add(Dense(1))
44 model.compile(loss='mean_squared_error', optimizer='rmsprop')
45 model.summary()
46
```

Обучаем и рисуем графики

```
40  
47 model.fit(trainX,trainY, epochs=100, batch_size=16, verbose=2)  
48 trainPredict = model.predict(trainX)  
49 testPredict= model.predict(testX)  
50 predicted=np.concatenate((trainPredict,testPredict),axis=0)  
51  
52 trainScore = model.evaluate(trainX, trainY, verbose=0)  
53 print(trainScore)  
54 |  
55 index = df.index.values  
56 plt.plot(index,df)  
57 plt.plot(index,predicted)  
58 plt.axvline(df.index[Tp], c="r")  
59 plt.show()
```



Генерация текста RNN



Генерация текста

```
1 import pandas as pd
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from keras.models import Sequential
5 from keras.layers import Dense, SimpleRNN, Activation
6
```

```
1 fn = open('abc.txt', 'rb')
2 lines = []
3 for t in fn:
4     t = t.strip().lower()
5     t = t.decode()
6     # t = t.decode('ascii', 'ignore')
7     if len(t) == 0:
8         continue
9     lines.append(t)
10 fn.close()
11 text = ' '.join(lines)
```

```
1 text
```

и некстати. У нее опустились, завяли черты и по сторонам лица п
осы, она задумалась в унылой позе, точно грешница на старинной к
зала она. — Вы же первый меня не уважаете теперь. На столе в ном
за себя заметы и ставь эти на свое. Прощаю, по крайней мере, с

Подготовка данных

```
1 chars = set ([c for c in text])
2 nbch = len(chars)
3 ch2ind = dict((c,i) for i,c in enumerate(chars))
4 ind2ch = dict((i,c) for i,c in enumerate(chars))
```

```
1 slen = 10
2 step = 1
3 inch = []
4 labch = []
5 for i in range(0, len(text)-slen, step):
6     inch.append(text[i:i+slen])
7     labch.append(text[i+slen])
```

```
1 x = np.zeros((len(inch),slen,nbch),dtype = np.bool)
2 y = np.zeros((len(inch),nbch),dtype = np.bool)
3 for i,j in enumerate(inch):
4     for j1,ch in enumerate(j):
5         x[i,j1,ch2ind[ch]] = 1
6     y[i,ch2ind[labch[i]]] = 1
```

Генерация текста. Сеть

```
1 model = Sequential()
2 model.add(SimpleRNN(128, return_sequences = False, input_shape=(slen, nbch), unroll=True))
3 model.add(Dense(nbch))
4 model.add(Activation('softmax'))
5 model.summary()
```

Model: "sequential_5"

Layer (type)	Output Shape	Param #
=====	=====	=====
simple_rnn_4 (SimpleRNN)	(None, 128)	27008
dense_4 (Dense)	(None, 82)	10578
activation_3 (Activation)	(None, 82)	0
=====	=====	=====

Total params: 37,586

Trainable params: 37,586

Non-trainable params: 0

```
1 model.compile(loss='categorical_crossentropy', optimizer = 'adam')
```

Генерация текста. Обучение

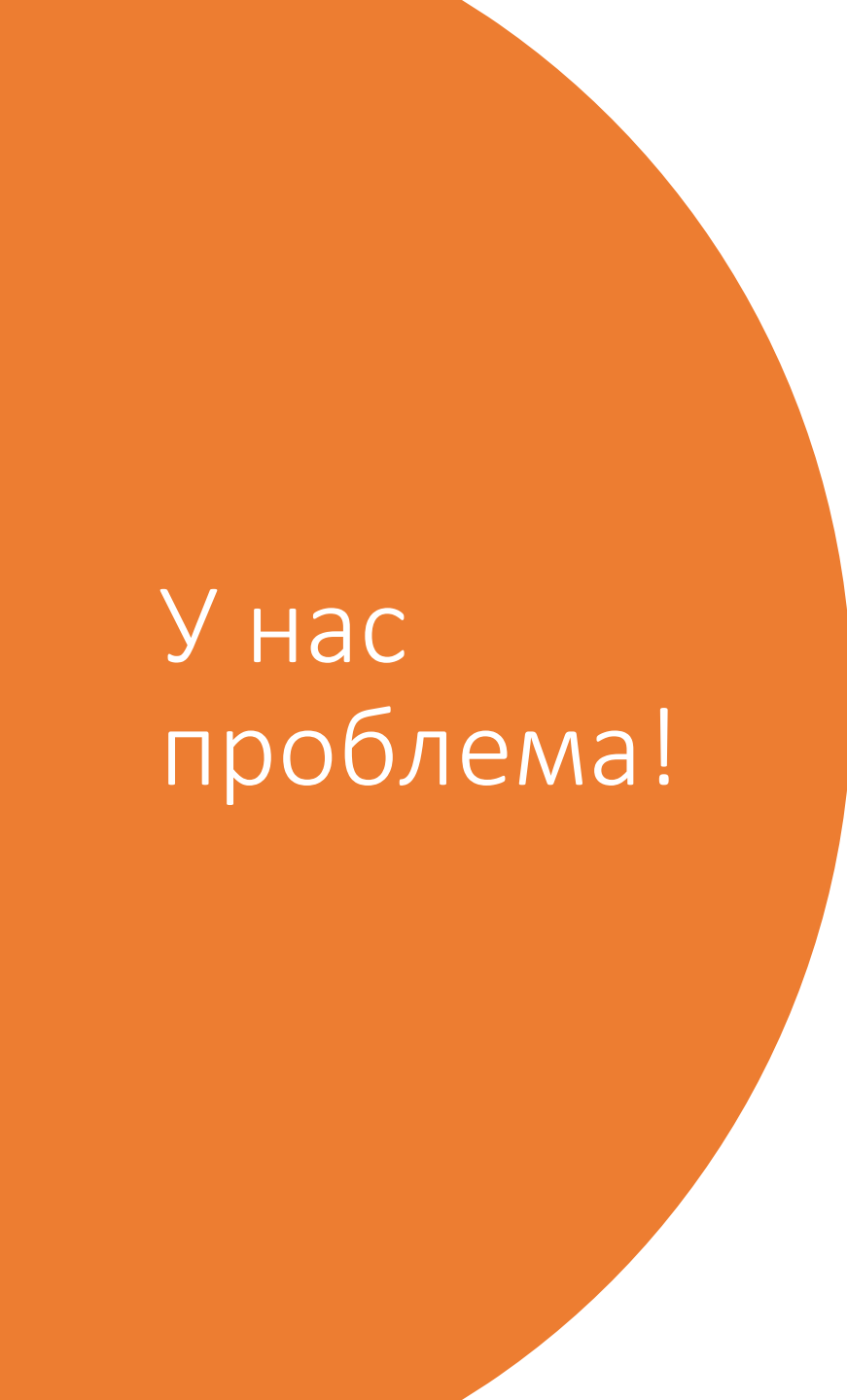
```
1 model.compile(loss='categorical_crossentropy',optimizer = 'adam')
```

```
1 for iter2 in range(25):
2     print("=" * 40)
3     print("Iteration #: %d" % (iter2))
4     model.fit(x,y,batch_size = 128, epochs = 1)
5     testid = np.random.randint(len(inch))
5     testch = inch[testid]
7     print(testch,end = " ")
3     for i in range(50):
3         xtest = np.zeros((1,slen,nbch))
3         for i,ch in enumerate(testch):
1             xtest[0,i,ch2ind[ch]] = 1
2         pred = model.predict(xtest,verbose = 0)[0]
3         ypred = ind2ch[np.argmax(pred)]
4         print(ypred,end = "")
5         testch = testch[1:]+ypred
5     print()
7
```

Iteration #: 16

Задание

1. Воспроизведите код из лекционного примера для "временного ряда"
2. Возьмите наблюдения за курсом каких-либо акций и предскажите стоимость акции на день вперед с помощью простой RNN
3. Воспроизведите RNN для анализа бинарной тональности коротких выражений на русском языке. Для базы данных каждый предлагает по 10 экземпляров для обучающего и тестового словарей, обмен данными через облако.
4. Воспроизведите простую RNN для генерации текста

A large orange circle is positioned on the left side of the slide, partially cut off by the edge.

У нас
проблема!

- При наличии достаточно длинных входных последовательностей в процессе обучения сеть «забывает» информацию об удаленных объектах
- В некоторых случаях возникает необходимость, чтобы сеть «помнила» информацию об объектах, находящихся в начале последовательности
- Нужны рекуррентные сети с памятью!