

# Лекция 6: Рекуррентные нейронные сети RNN (Recurrent Neural Networks)

Автор: Сергей Вячеславович Макрушин e-mail: [SVMakrushin@fa.ru](mailto:SVMakrushin@fa.ru) (<mailto:SVMakrushin@fa.ru>)

Финансовый университет, 2021 г.

При подготовке лекции использованы материалы:

- ...

v 0.1 15.04.21

## Разделы:

- [раздел 1](#)
- [раздел 2](#)
- 
- [к оглавлению](#)

---

## Нормализация

- [к оглавлению](#)

```
In [45]: # загружаем стиль для оформления презентации
from IPython.display import HTML
from urllib.request import urlopen
html = urlopen("file:./lec_v2.css")
HTML(html.read().decode('utf-8'))
```

Out[45]:

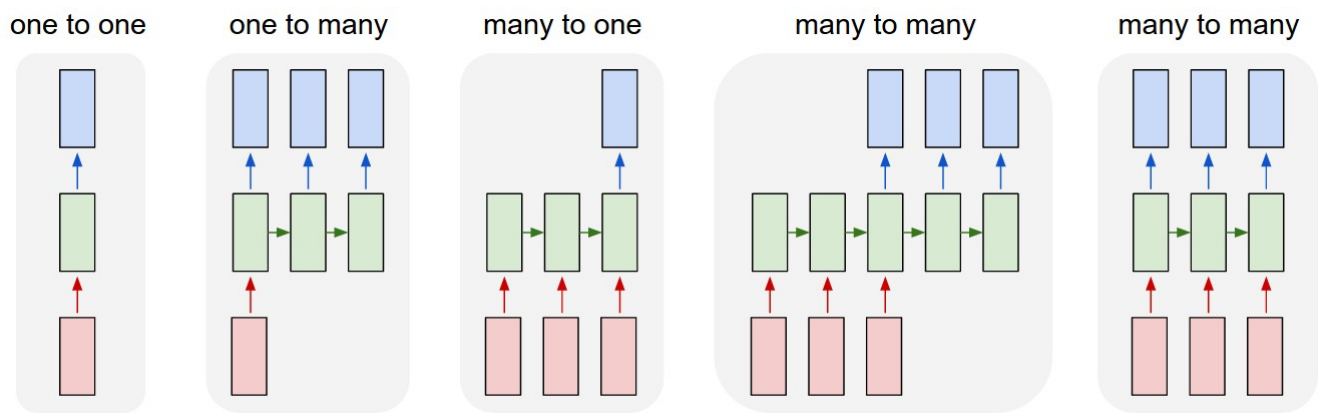
---

## Базовые подходы к RNN

- [к оглавлению](#)

### Мотивация для использования рекуррентных нейронных сетей

- Не рекуррентные архитектуры ИНС получают на вход вектор данных и пытаются по нему предсказать тот или иной результат (минимизировать ошибку).
- Важно, что входной (и выходной) вектор должен **иметь одну и ту же размерность**. Во многих задачах это не так.



### Задачи с последовательностями

Каждый прямоугольник представляет собой вектор, а стрелки представляют функции (например, умножение матриц). Входные векторы показаны красным, выходные векторы - синим, а зеленые векторы содержат состояние RNN.

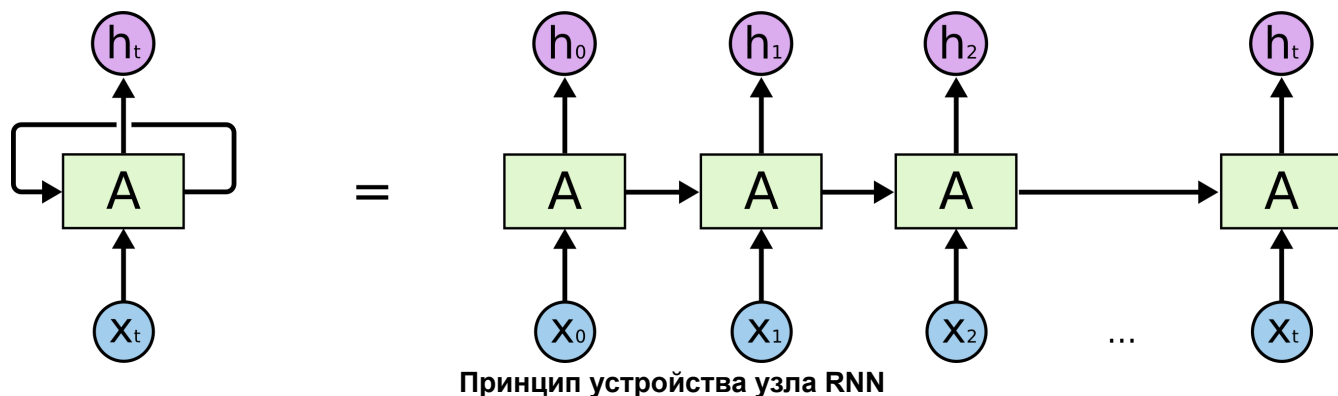
1. один вход, один выход (классический случай)

- один вход, последовательность выходов
- последовательность входов, один выход
- последовательность входов, затем последовательность выходов
- синхронизированные последовательности входов и выходов

### Рекуррентная нейронная сеть (RNN)

**Рекуррентная нейронная сеть (RNN)** - это класс искусственных нейронных сетей, в которых узел может получать входы не только от других узлов и текущих входных данных но и выходы узлов, полученные при рассмотрении предыдущих входных данных последовательности.

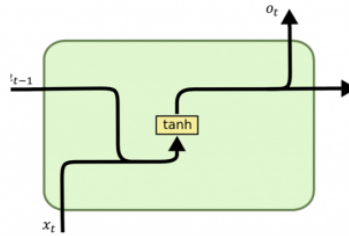
- обмен вектором внутреннего состояния, полученного на предыдущем шаге, позволяет использовать информацию о предыдущих шагах, которые сеть уже обработала
- при рассмотрении всей последовательности веса каждого узла одни и те же при рассмотрении всех входных данных последовательности
- Такая архитектура сети позволяет обрабатывать серии событий во времени или последовательные пространственные цепочки произвольной размерности



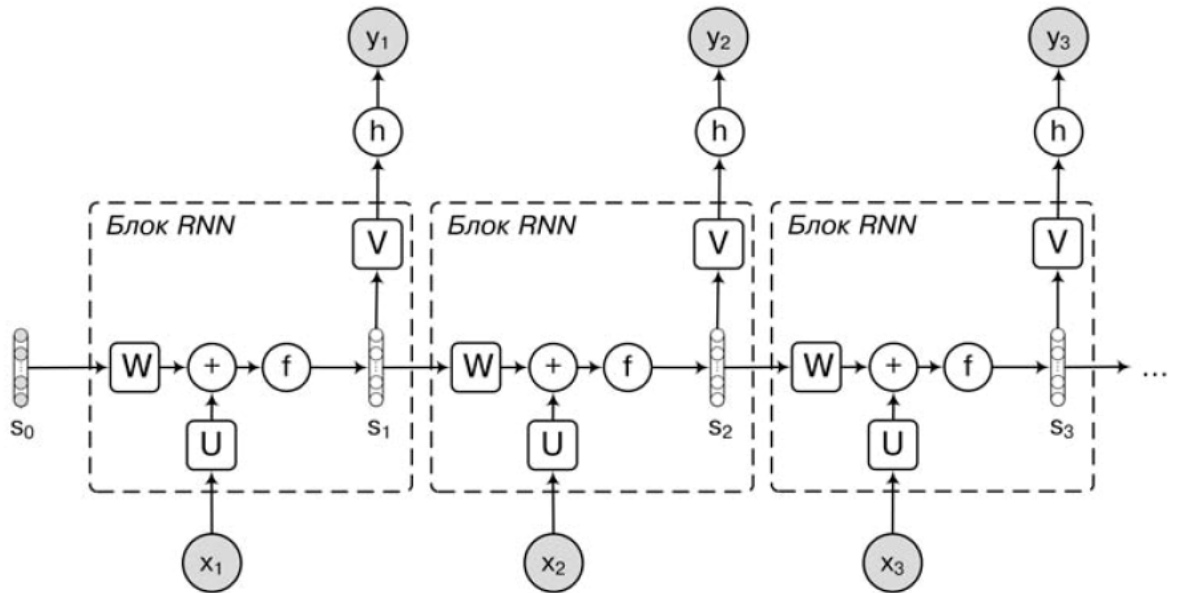
Трудность рекуррентной сети:

- если учитывать каждый шаг времени, то становится необходимым для каждого шага времени (последовательности) создавать свой слой нейронов, что создает серьезные вычислительные сложности
- многослойные реализации вычислительно неустойчивы: в них как правило либо исчезают либо зашкаливают веса
- если ограничить расчёт фиксированным временным окном, то полученные модели не будут отражать долгосрочных трендов

# RNN



## Распространение ошибок в узле RNN



## Распространение ошибок в узле RNN

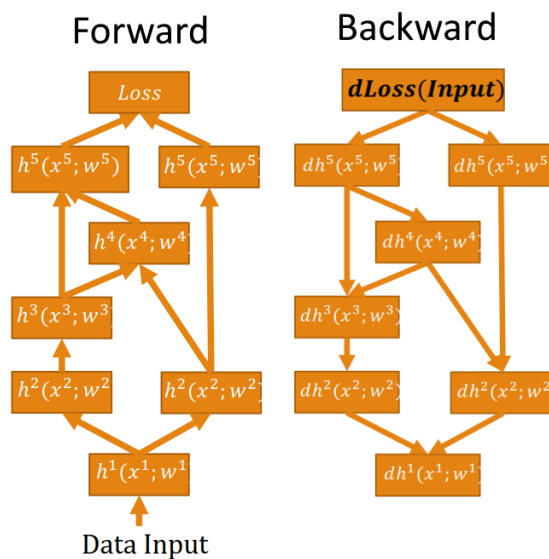
- $a_t = b + W s_{t-1} + U x_t$
- $s_t = f(a_t)$
- $o_t = c + V s_t$
- $y_t = h(o_t)$

Где:

- $f$  — это нелинейность рекуррентной сети (обычно  $\sigma$ ,  $\tanh$  или  $ReLU$ )
- $h$  — функция, с помощью которой получается ответ (например,  $softmax$ )

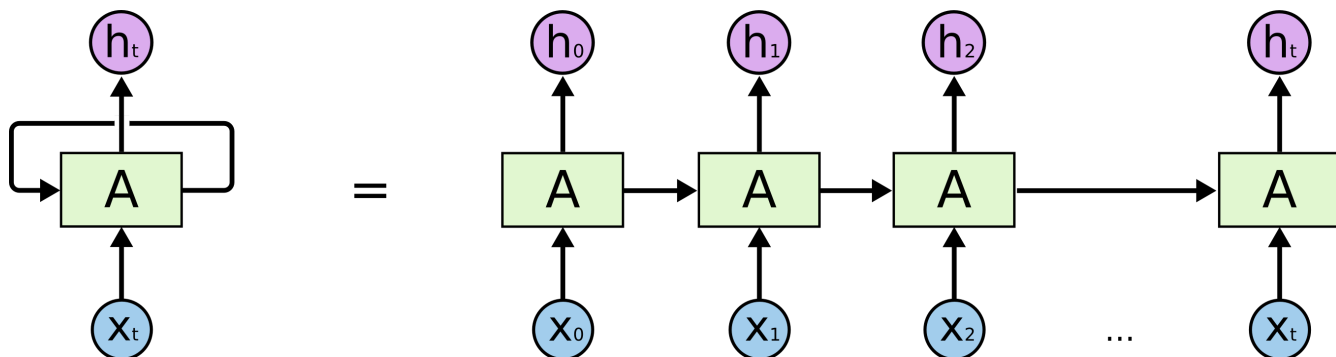
## Распространение ошибки в архитектуре RNN

- В прямой нейронной сети ошибка на конкретном нейроне вычисляется как функция от ошибок нейронов, которые используют его выходное значение формируется ациклический графы вычислений:



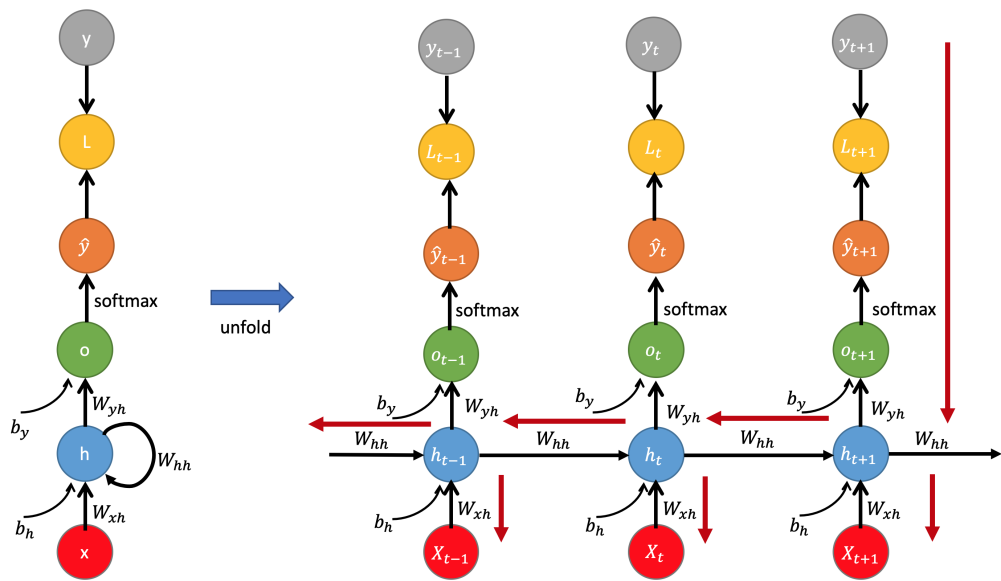
### Прямой и обратный проход процедуры обучения многослойной ИНС

- В архитектуре RNN нейрон принимает в качестве входа результат вычисления в нем самом (через вектор состояния)
  - Важно понимать, что при этом петли в графе вычислений не образуется
  - Вычисления, которые делает рекуррентная сеть, можно развернуть обратно до начала обрабатываемой последовательности
  - Можно сказать, что на каждом шаге обрабатываемой последовательности сеть создает копии самой себя



Принцип устройства узла RNN

- И на каждом последовательности мы фактически обучаем глубокую нейронную сеть, в которой столько слоев, сколько элементов в последовательности на данный момент мы уже видели
- Рекуррентная сеть — разворачивается вдоль элементов последовательности  $1 \dots T$  в очень-очень многоуровневую обычную сеть, в которой одни и те же веса переиспользуются на каждом уровне.
  - Для хранения весов достаточно одной матрицы
  - Градиенты по весам не затухают до нуля сразу же (как это бывает в обычных глубоких сетях)
  - Если матрица весов меняет норму вектора градиента при проходе через один «слой» обратного распространения, то при проходе через  $T$  слоев эта норма изменяется экспоненциально (т.к. веса матрицы одни и те же) это приводит:
    - к **"взрыву градиентов"** (exploding gradients), если матрица заметно увеличивает норму вектора градиента
    - к экспоненциальному затуханию градиентов (Vanishing gradients), если матрица заметно уменьшает норму вектора градиента



Распространение ошибок в узле RNN

## Пример реализации RNN

```
In [50]: # Импорты:
import torch
from torch import nn

import numpy as np
import csv
```

```
In [51]: # Входные данные:

text = ['hey how are you', 'good i am fine', 'have a nice day']
```

```
In [52]: with open('name2.csv', encoding='cp1251') as csvfile:
    csv_reader = csv.reader(csvfile)
    text = list(n[0].lower() for n in csv_reader)
```

```
In [53]: text[:3]
```

```
Out[53]: ['агафья', 'аглая', 'агния']
```

```
In [54]: # Кодировка символов числами:

# Join all the sentences together and extract the unique characters from the combined se
chars = set(''.join(text)+' ')

# Creating a dictionary that maps integers to the characters
int2char = dict(enumerate(chars))

# Creating another dictionary that maps characters to integers
char2int = {char: ind for ind, char in int2char.items()}
```

```
In [55]: print(char2int)
```

```
{'д': 0, 'р': 1, 'у': 2, 'ю': 3, 'т': 4, 'ф': 5, 'з': 6, 'ж': 7, 'к': 8, ' ': 9, 'е': 10, 'ь': 11, 'м': 12, 'н': 13, 'ц': 14, 'а': 15, 'л': 16, 'я': 17, 'ё': 18, 'о': 19, 'б': 20, 'и': 21, 'п': 22, 'г': 23, 'с': 24, 'в': 25}
```

In [56]: *# Выравнивание всех строк до фиксированной (максимальной) длины:*

```
maxlen = len(max(text, key=len))
print("The longest string has {} characters".format(maxlen))

# Padding

# A simple loop that loops through the list of sentences and adds a ' ' whitespace until
# the length of the longest sentence

for i in range(len(text)):
    while len(text[i])<maxlen:
        text[i] += ' '
```

The longest string has 10 characters

In [57]: *# Подготовка входных и выходных последовательностей:*

```
input_seq = [] # исходная последовательность
target_seq = [] # последовательность, смещенная на 1 (без первого символа)

for i in range(len(text)):
    # Remove last character for input sequence
    input_seq.append(text[i][: -1])

    # Remove firsts character for target sequence
    target_seq.append(text[i][1:])

    print("Input Sequence: {}\nTarget Sequence: {}".format(input_seq[i], target_seq[i]))
```

```
Input Sequence: агафья
Target Sequence: гафья
Input Sequence: аглая
Target Sequence: глая
Input Sequence: агния
Target Sequence: гния
Input Sequence: агриппина
Target Sequence: гриппина
Input Sequence: акулина
Target Sequence: кулина
Input Sequence: алевтина
Target Sequence: левтина
Input Sequence: александр
Target Sequence: лександра
Input Sequence: алина
Target Sequence: лина
Input Sequence: алла
Target Sequence: лла
Input Sequence: анастасия
Target Sequence: настасия
```

In [58]: *# One hot энкодер (для батча примеров фиксированной длины)*

```
def one_hot_encode(sequence, dict_size, seq_len, batch_size):
    # Creating a multi-dimensional array of zeros with the desired output shape
    features = np.zeros((batch_size, seq_len, dict_size), dtype=np.float32)

    # Replacing the 0 at the relevant character index with a 1 to represent that character
    for i in range(batch_size):
        for u in range(seq_len):
            features[i, u, sequence[i][u]] = 1
    return features
```

```
In [59]: # print(input_seq, dict_size, seq_len, batch_size)
```

```
In [60]: # Построение входного и выходного тензора:
# batch_size x seq_len x dict_size ; seq_len = maxlen - 1

dict_size = len(char2int)
seq_len = maxlen - 1
batch_size = len(text)

# Symbol seq to int seq:
for i in range(len(text)):
    input_seq[i] = [char2int[character] for character in input_seq[i]]
    target_seq[i] = [char2int[character] for character in target_seq[i]]

input_seq = one_hot_encode(input_seq, dict_size, seq_len, batch_size)
input_seq = torch.from_numpy(input_seq)
# print(f"Input shape: {input_seq.shape} --> (Batch Size, Sequence Length, One-Hot Encod

# --- --- --- ---
# target_seq = one_hot_encode(target_seq, dict_size, seq_len, batch_size)
target_seq = torch.Tensor(target_seq) # from list; without one-hot encoding

# print(f"Target shape: {input_seq.shape} --> (Batch Size, Sequence Length, One-Hot Encod
```

```
In [61]: # torch.cuda.is_available() checks and returns a Boolean True if a GPU is available, else
is_cuda = False # torch.cuda.is_available()

# If we have a GPU available, we'll set our device to GPU. We'll use this device variable
if is_cuda:
    device = torch.device("cuda")
    print("GPU is available")
else:
    device = torch.device("cpu")
    print("GPU not available, CPU used")
```

GPU not available, CPU used

### Модель:

```
torch.nn.RNN(*args, **kwargs)
```

For each element in the input sequence, each layer computes the following function:

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

Shape:

Input:

- **input** of shape (seq\_len, batch, input\_size): tensor containing the features of the input sequence. The input can also be a packed variable length sequence.
- **h\_0** of shape (num\_layers \* num\_directions, batch, hidden\_size): tensor containing the initial hidden state for each element in the batch. Defaults to zero if not provided. If the RNN is bidirectional, num\_directions should be 2, else it should be 1.

Output:

- **output** of shape (seq\_len, batch, num\_directions \* hidden\_size): tensor containing the output features (h\_t) from the last layer of the RNN, for each t.
- **h\_n** of shape (num\_layers \* num\_directions, batch, hidden\_size): tensor containing the hidden state for t = seq\_len.

Parameters:

- `input_size` - The number of expected features in the input `x`
- `hidden_size` - The number of features in the hidden state `h`
- `num_layers` - Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two RNNs together to form a stacked RNN, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1
- `nonlinearity` - The non-linearity to use. Can be either `'tanh'` or `'relu'`. Default: `'tanh'`
- `bias` - If `False`, then the layer does not use bias weights `b_ih` and `b_hh`. Default: `True`
- `batch_first` - If `True`, then the input and output tensors are provided as `(batch, seq, feature)`. Default: `False`
- `dropout` - If non-zero, introduces a Dropout layer on the outputs of each RNN layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- `bidirectional` - If `True`, becomes a bidirectional RNN. Default: `False`

```
In [62]: class Model(nn.Module):
    def __init__(self, input_size, output_size, hidden_dim, n_layers):
        super(Model, self).__init__()

        # Defining some parameters
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers

        # Defining the Layers
        # RNN Layer
        self.rnn = nn.RNN(input_size, hidden_dim, n_layers, batch_first=True)
        # Fully connected Layer
        self.fc = nn.Linear(hidden_dim, output_size)

    def forward(self, x):

        batch_size = x.size(0)

        #Initializing hidden state for first input using method defined below
        hidden = self.init_hidden(batch_size)

        # Passing in the input and hidden state into the model and obtaining outputs
        out, hidden = self.rnn(x, hidden)

        # Reshaping the outputs such that it can be fit into the fully connected layer
        # print('d1: ', out.size())
        out = out.contiguous().view(-1, self.hidden_dim)

        # print('d2: ', out.size())
        out = self.fc(out)
        # print('d3: ', out.size())

        return out, hidden

    def init_hidden(self, batch_size):
        # This method generates the first hidden state of zeros which we'll use in the j
        hidden = torch.zeros(self.n_layers, batch_size, self.hidden_dim).to(device)
        # We'll send the tensor holding the hidden state to the device we specified ear
        return hidden
```



```
In [63]: # Создаем инстанс модели с установленными гиперпараметрами

model = Model(input_size=dict_size, output_size=dict_size, hidden_dim=12, n_layers=1)
# We'll also set the model to the device that we defined earlier (default is CPU)
model = model.to(device)

# Define hyperparameters
n_epochs = 100
lr=0.01

# Define Loss, Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
```

```
In [64]: output, hidden = model(input_seq)
```

```
In [65]: # target_seq.view(-1), target_seq.size()
```

```
In [66]: # Training Run
input_seq = input_seq.to(device)

for epoch in range(1, n_epochs + 1):
    optimizer.zero_grad() # Clears existing gradients from previous epoch
    #input_seq = input_seq.to(device)
    output, hidden = model(input_seq)

    output = output.to(device)
    target_seq = target_seq.to(device)

    loss = criterion(output, target_seq.view(-1).long())
#     loss = criterion(output.view(-1), target_seq.view(-1).long())

    loss.backward() # Does backpropagation and calculates gradients
    optimizer.step() # Updates the weights accordingly

    if epoch%10 == 0:
        print('Epoch: {}/{}. ....'.format(epoch, n_epochs), end=' ')
        print("Loss: {:.4f}".format(loss.item()))
```

```
Epoch: 10/100 .... Loss: 2.7077
Epoch: 20/100 .... Loss: 2.1426
Epoch: 30/100 .... Loss: 1.9227
Epoch: 40/100 .... Loss: 1.7293
Epoch: 50/100 .... Loss: 1.5687
Epoch: 60/100 .... Loss: 1.4620
Epoch: 70/100 .... Loss: 1.3818
Epoch: 80/100 .... Loss: 1.3205
Epoch: 90/100 .... Loss: 1.2726
Epoch: 100/100 .... Loss: 1.2330
```

```
In [69]: def predict(model, character):
# One-hot encoding our input to fit into the model
character = np.array([[char2int[c] for c in character]])
character = one_hot_encode(character, dict_size, character.shape[1], 1)
character = torch.from_numpy(character)
character = character.to(device)

out, hidden = model(character)

prob = nn.functional.softmax(out[-1], dim=0).data
# Taking the class with the highest probability score from the output
# TODO: select top_k

char_ind = torch.max(prob, dim=0)[1].item()

return int2char[char_ind], hidden
```

```
In [70]: def sample(model, out_len, start='hey'):
model.eval() # eval mode
start = start.lower()
# First off, run through the starting characters
chars = [ch for ch in start]
size = out_len - len(chars)

# Now pass in the previous characters and get a new one
for ii in range(size):
    char, h = predict(model, chars)
    chars.append(char)

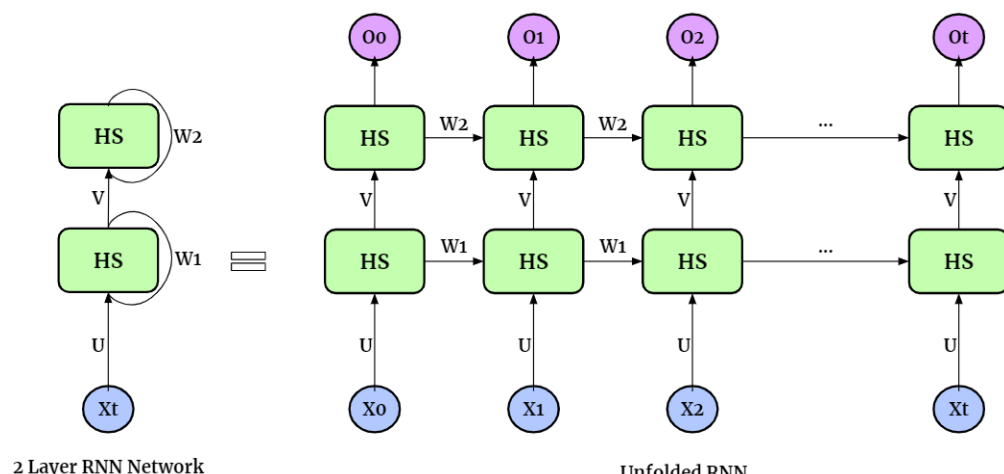
return ''.join(chars)
```

```
In [82]: sample(model, 45, 'иван')
```

```
Out[82]: 'ивана'
```

## Построение многослойных нейронных сетей на базе архитектуры RNN

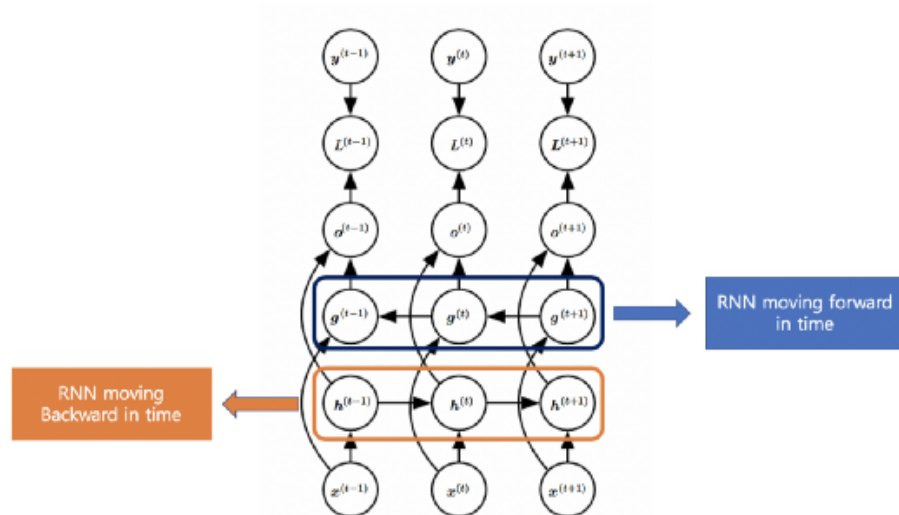
- Рассмотрим всю рекуррентную сеть как слой и используем ее выходы как входы для следующего рекуррентного слоя.
- Мотивация: каждый слой действует в своем собственном «масштабе времени», примерно как каждый слой сверточной сети действует в своем масштабе, на свой размер окна входов.



### Распространение ошибок в узле RNN

**Двунаправленные рекуррентные сети (bidirectional RNN):** для входной последовательности запустим RNN (обычно с разными весами) два раза: один слой будет **читать последовательность слева направо**, а другой — **справа налево**

- Матрицы весов для двух направлений абсолютно независимы и между ними нет взаимодействия
- Ограничение: данный подход возможен только для последовательностей, которые даны сразу целиком (например для предложений естественного языка).
- Мотивация в том, чтобы получить состояние, отражающее контекст и слева, и справа для каждого элемента последовательности (например для отнесения слова к части речи т.к. важно анализировать все предложение, и слева, и справа от слова)
- Вместо классической рекуррентной сети из трех матриц, может использоваться любая другая конструкция, например LSTM или GRU.



Распространение ошибок в узле RNN

## LSTM

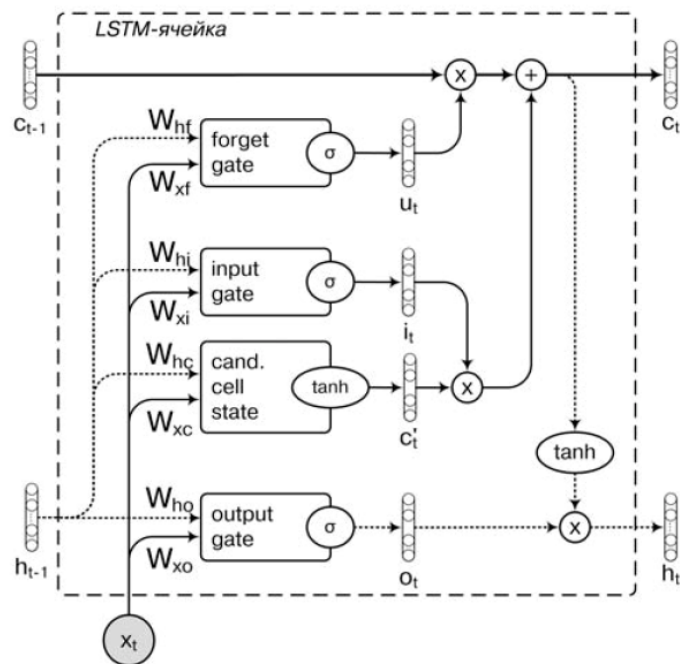
### LSTM

LSTM (Long Short-Term Memory)

- обычные рекуррентные сети очень плохо справляются с ситуациями, когда нужно что-то «запомнить» надолго: влияние скрытого состояния или входа с шага  $t$  на последующие состояния рекуррентной сети экспоненциально затухает
- LSTM хорошо приспособлена к обучению на задачах классификации, обработки и прогнозирования временных рядов в случаях, когда важные события разделены временными лагами с неопределённой продолжительностью и границами
- вместо одного-единственного числа, на которое влияют все последующие состояния, используется специального вида ячейка моделирующая "долгую память"
  - LSTM моделирует процессы записи и чтения из этой "ячейки памяти"
  - у ячейки не один набор весов, как у обычного нейрона, а сразу несколько

В LSTM есть три основных вида узлов, которые называются гейтами:

- входной (input gate)
- забывающий (forget gate)
- выходной (output gate)
- рекуррентная ячейка со скрытым состоянием



Структура LSTM

Переменные:

- $x_t$  — входной вектор во время  $t$
- $h_t$  — вектор скрытого состояния во время  $t$
- $c_t$  — вектор ячейки состояния во время  $t$
- $W_i$  — матрицы весов, применяющиеся ко входу
- $W_h$  — матрицы весов, в рекуррентных соединениях;  $b$  - векторы свободных членов
- candidate cell state:  $c'_t = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_{c'})$
- input gate:  $i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$
- forget gate:  $f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$
- output gate:  $o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$
- cell state:  $c_t = f_t \odot c_{t-1} + i_t \odot c'_t$
- block output:  $h_t = o_t \odot \tanh(c_t)$

### Специфика ячейки памяти в LSTM

- Вход в LSTM:
  - входные данные  $x_t$
  - скрытое состояние  $h_{t-1}$
  - вектор "ячейки памяти" (cell)  $c_t$ :
- Кандидат на новое значение памяти, полученный из входа и предыдущего скрытого состояния вектор:  $c'_t$ 

$$c_t = f_t \odot c_{t-1} + i_t \odot c'_t$$
- Новое значение  $c_t$  получается как линейная комбинация из старого с коэффициентами из забывающего гейта  $f_t$  и нового кандидата  $c'_t$  с коэффициентами из входного гейта  $i_t$ .
- Покомпонентное умножение приводит к тому, что на очередном шаге может быть перезаписана только часть "памяти" LSTM-ячейки и какая это будет часть, тоже определяет сама ячейка.
  - LSTM-ячейка может не просто выбрать, записать новое значение или выкинуть его, а еще и сохранить любую линейную комбинацию старого и нового значения, причем коэффициенты могут быть разными в разных компонентах вектора.
  - Решения ячейка принимает в зависимости от конкретного входа.

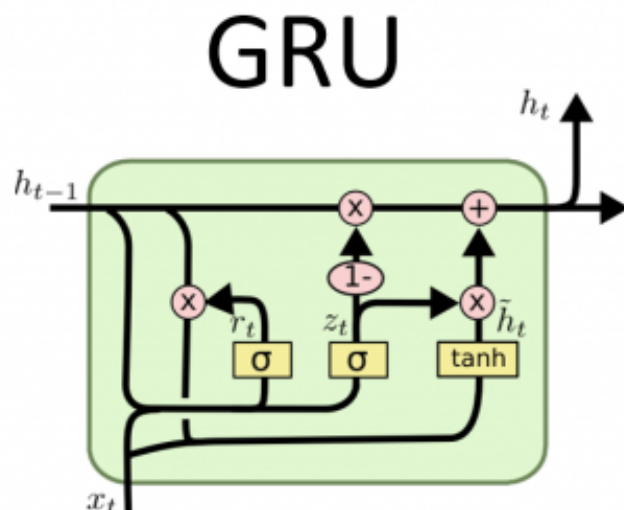
Так распространяется градиент ошибки для  $c_t$ , если рассматривать ее без забывающего гейта:

$$\frac{\partial c_t}{\partial c_{t-1}} = 1$$

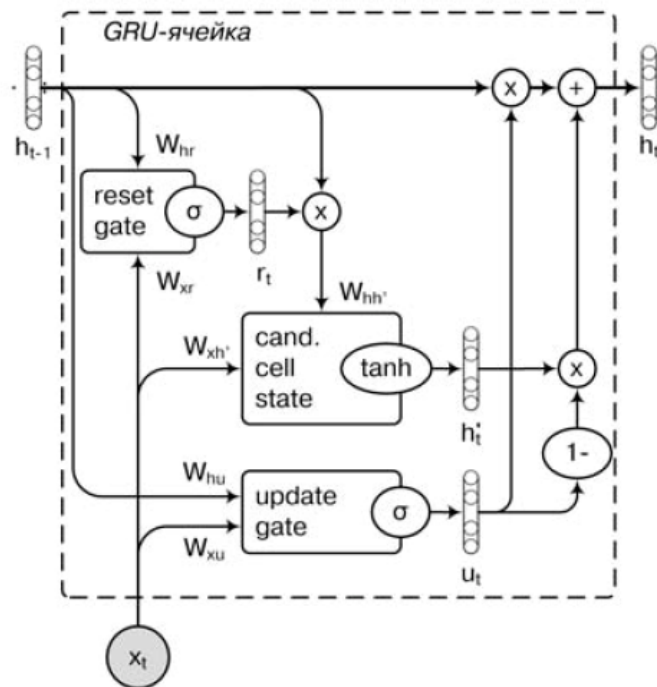
- в рекурсивном вычислении состояния ячейки нет никакой нелинейности: т.е. ошибки в сети из LSTM пропагируются без изменений, и скрытые состояния LSTM могут, если сама ячейка не решит их перезаписать, сохранять свои значения неограниченно долго
- **Важно:** хотя обычно веса нейронной сети инициализируются маленькими случайными числами свободный член забывающего гейта  $b_f$ 
  - все LSTM-ячейки изначально будут иметь значение  $f_t$  около 1/2
  - ошибки и память будут затухать экспоненциально. Поэтому свободный член  $b_f$  нужно инициализировать большими значениями, около 1 или даже 2: тогда значения забывающих гейтов  $f_t$  в начале обучения будут близки к нулю и градиенты будут свободно распространяться вдоль всей последовательности.

Существует много разных вариантов LSTM:

- LSTM без входного гейта  $i_t$
  - LSTM без забывающего гейта  $f_t$
  - LSTM без выходного гейта  $o_t$
  - LSTM без функции активации  $\sigma$  на входном гейте
  - LSTM без функции активации  $\sigma$  на выходном гейте
  - LSTM без замочных скважин
  - LSTM со связанными входным и забывающим гейтом
  - LSTM с дополнительными рекуррентными связями на каждом гейте
- 
- LSTM требует довольно значительных ресурсов
    - В обычном RNN каждая ячейка имеет один вектор скрытого состояния  $h$ , а веса представлены тремя матрицами (плюс свободные члены)
    - В LSTM-ячейке даже в базовой модели участвует сразу восемь матриц весов
  - Цель: добиться того же эффекта долгосрочной памяти и решить проблему затухающих градиентов более эффективно
  - Критически важными компонентами для успешной работы LSTM выступают
    - два гейта: выходной и забывающий
    - «память»  $c_t$  и константная ошибка, которая позволяет состоянию LSTM сохраняться надолго
  - Архитектура GRU использует идею совмещения выходного и забывающего гейта, а скрытое состояние  $h_t$  совмещает со значением памяти  $c_t$ .



Структура GRU



Структура GRU

```
In [84]: import torch
import torch.nn as nn
import torch.nn.functional as F

import numpy as np
from collections import Counter
import os
from argparse import Namespace
import collections
```

In [85]: *# code from: <https://trungtran.io/2019/02/08/text-generation-with-pytorch/>*

```
# set parameters:

# flags = Namespace(
#     train_file='oliver.txt',
#     seq_size=32,
#     batch_size=16,
#     embedding_size=64,
#     lstm_size=64,
#     gradients_norm=5,
#     initial_words=['I', 'am'],
#     predict_top_k=5,
#     checkpoint_path='checkpoint',
# )

Flags = collections.namedtuple('Flags', 'train_file seq_size batch_size embedding_size lstm_size gradients_norm initial_words predict_top_k checkpoint_path learning_rate')
# print with_class._fields

flags = Flags(
    train_file='AnnaKarenina__.txt',
    seq_size=32,
    batch_size=16,
    embedding_size=64,
    lstm_size=64,
    gradients_norm=5,
    initial_words=['Анна', 'не'],
    predict_top_k=5,
    checkpoint_path='checkpoint',
    learning_rate = 0.01
)
```

```
In [86]: def get_data_from_file(train_file, batch_size, seq_size):
    with open(train_file, 'r', encoding='utf-8') as f:
        text = f.read()
        text = text.split()

    word_counts = Counter(text)
    sorted_vocab = sorted(word_counts, key=word_counts.get, reverse=True)
    int_to_vocab = {k: w for k, w in enumerate(sorted_vocab)}
    vocab_to_int = {w: k for k, w in int_to_vocab.items()}
    n_vocab = len(int_to_vocab)

    print('Vocabulary size', n_vocab)

    int_text = [vocab_to_int[w] for w in text]
    num_batches = int(len(int_text) / (seq_size * batch_size))
    print(f'num_batches: {num_batches}')
    in_text = int_text[:num_batches * batch_size * seq_size]
    out_text = np.zeros_like(in_text)
    out_text[:-1] = in_text[1:]
    out_text[-1] = in_text[0]
    in_text = np.reshape(in_text, (batch_size, -1))
    out_text = np.reshape(out_text, (batch_size, -1))
    return int_to_vocab, vocab_to_int, n_vocab, in_text, out_text

# TODO: Lower case ??
# TODO: DataLoader ??
```

```
In [87]: def get_batches(in_text, out_text, batch_size, seq_size):
num_batches = np.prod(in_text.shape) // (seq_size * batch_size)
for i in range(0, num_batches * seq_size, seq_size):
    yield in_text[:, i:i+seq_size], out_text[:, i:i+seq_size]
```

```
In [26]: class RNNModule(nn.Module):
def __init__(self, n_vocab, seq_size, embedding_size, lstm_size):
    super(RNNModule, self).__init__()
    self.seq_size = seq_size
    self.lstm_size = lstm_size

    self.embedding = nn.Embedding(n_vocab, embedding_size)
    self.lstm = nn.LSTM(embedding_size,
                        lstm_size,
                        batch_first=True)
    self.dense = nn.Linear(lstm_size, n_vocab)

def forward(self, x, prev_state):
    embed = self.embedding(x)
    output, state = self.lstm(embed, prev_state)
    logits = self.dense(output)
    return logits, state

def zero_state(self, batch_size):
    return (torch.zeros(1, batch_size, self.lstm_size),
            torch.zeros(1, batch_size, self.lstm_size))
```

## Модель:

`torch.nn.LSTM(*args, **kwargs)`

Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.

In a multilayer LSTM, the input  $x_t^{(l)}$  of the  $l$ -th layer ( $l \geq 2$ ) is the hidden state  $h_t^{(l-1)}$  of the previous layer multiplied by dropout  $\delta_t^{(l-1)}$  where each  $\delta_t^{(l-1)}$  is a Bernoulli random variable which is 0 with probability dropout.

Shape:

*Input:*

- **input** of shape (seq\_len, batch, input\_size): tensor containing the features of the input sequence.
  - The input can also be a packed variable length sequence.
- **h\_0** of shape (num\_layers \* num\_directions, batch, hidden\_size): tensor containing the initial hidden state for each element in the batch.
  - If the LSTM is bidirectional, num\_directions should be 2, else it should be 1.
- **c\_0** of shape (num\_layers \* num\_directions, batch, hidden\_size): tensor containing the initial cell state for each element in the batch.
- *If (h\_0, c\_0) is not provided, both h\_0 and c\_0 default to zero*

*Output:*

- **output** of shape (seq\_len, batch, num\_directions \* hidden\_size): tensor containing the output features ( $h_t$ ) from the last layer of the LSTM, for each  $t$ .
- **h\_n** of shape (num\_layers \* num\_directions, batch, hidden\_size): tensor containing the hidden state for  $t = \text{seq\_len}$ .
- **c\_n** of shape (num\_layers \* num\_directions, batch, hidden\_size): tensor containing the cell state for  $t = \text{seq\_len}$ .



Parameters:

- `input_size` – The number of expected features in the input `x`
- `hidden_size` – The number of features in the hidden state `h`
- `num_layers` – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two LSTMs together to form a stacked LSTM, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
- `bias` – If `False`, then the layer does not use bias weights `b_ih` and `b_hh`. Default: `True`
- `batch_first` – If `True`, then the input and output tensors are provided as `(batch, seq, feature)`. Default: `False`
- `dropout` – If non-zero, introduces a Dropout layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- `bidirectional` – If `True`, becomes a bidirectional LSTM. Default: `False`
- `proj_size` – If `> 0`, will use LSTM with projections of corresponding size. Default: 0

```
In [88]: def predict(device, model, words, n_vocab, vocab_to_int, int_to_vocab, top_k=5):
    model.eval()

    state_h, state_c = model.zero_state(1)
    state_h = state_h.to(device)
    state_c = state_c.to(device)

    for w in (w for wl in words for w in wl):
        ix = torch.tensor([[vocab_to_int[w]]], dtype=torch.long).to(device)
        output, (state_h, state_c) = model(ix, (state_h, state_c))

    _, top_ix = torch.topk(output[0], k=top_k)
    choices = top_ix.tolist()
    choice = np.random.choice(choices[0])

    words_new = list()
    words_new.append(int_to_vocab[choice])

    for _ in range(100):
        ix = torch.tensor([[choice]], dtype=torch.long).to(device)
        output, (state_h, state_c) = model(ix, (state_h, state_c))

        _, top_ix = torch.topk(output[0], k=top_k)
        choices = top_ix.tolist()
        choice = np.random.choice(choices[0])
        words_new.append(int_to_vocab[choice])

    words.append(words_new)

    print('\n\n'.join(' '.join(s) for s in words))
```

```
In [89]: # def get_loss_and_train_op(model, lr=0.001):
#         criterion = nn.CrossEntropyLoss()
#         optimizer = torch.optim.Adam(model.parameters(), lr=lr)
#
#         return criterion, optimizer
```

```
In [90]: device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
int_to_vocab, vocab_to_int, n_vocab, in_text, out_text = get_data_from_file(
    flags.train_file, flags.batch_size, flags.seq_size)

model = RNNModule(n_vocab, flags.seq_size,
                  flags.embedding_size, flags.lstm_size)
model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=flags.learning_rate)

iteration = 0
```

Vocabulary size 55567  
num\_batches: 576

```
In [30]: device
```

```
Out[30]: device(type='cuda')
```

```
In [31]: int_to_vocab[422]
```

```
Out[31]: 'себе.'
```

Основной цикл:

```

In [91]: for e in range(50):
    batches = get_batches(in_text, out_text, flags.batch_size, flags.seq_size)
    state_h, state_c = model.zero_state(flags.batch_size)

    # Transfer data to device (GPU)
    state_h = state_h.to(device)
    state_c = state_c.to(device)

    for x, y in batches:
        # print(type(x), x.shape, x)
        iteration += 1

        # Tell it we are in training mode
        model.train()

        # Reset all gradients
        optimizer.zero_grad()

        # Transfer data to GPU
        x = torch.tensor(x, dtype=torch.long).to(device)
        y = torch.tensor(y, dtype=torch.long).to(device)

        logits, (state_h, state_c) = model(x, (state_h, state_c))
        loss = criterion(logits.transpose(1, 2), y)

        state_h = state_h.detach()
        state_c = state_c.detach()

        loss_value = loss.item()

        # Perform back-propagation
        loss.backward()

        # gradient clipping:
        _ = torch.nn.utils.clip_grad_norm_(
            model.parameters(), flags.gradients_norm)

        # Update the network's parameters
        optimizer.step()

        # print the loss value:
        if iteration % 50 == 0:
            print('Epoch: {}/{}'.format(e, 200),
                  'Iteration: {}'.format(iteration),
                  'Loss: {}'.format(loss_value))

        if iteration % 1000 == 0:
            predict(device, model, [flags.initial_words], n_vocab,
                    vocab_to_int, int_to_vocab, top_k=5)
        # torch.save(model.state_dict(),
        #             'checkpoint_pt/model-{}.pth'.format(iteration))

```

```

Epoch: 0/200 Iteration: 50 Loss: 8.682318687438965
Epoch: 0/200 Iteration: 100 Loss: 8.617399215698242
Epoch: 0/200 Iteration: 150 Loss: 8.296709060668945
Epoch: 0/200 Iteration: 200 Loss: 7.892895221710205
Epoch: 0/200 Iteration: 250 Loss: 7.927113056182861
Epoch: 0/200 Iteration: 300 Loss: 8.42678451538086
Epoch: 0/200 Iteration: 350 Loss: 8.1193265914917
Epoch: 0/200 Iteration: 400 Loss: 8.304059982299805
Epoch: 0/200 Iteration: 450 Loss: 7.96299409866333
Epoch: 0/200 Iteration: 500 Loss: 7.650730609893799
Epoch: 0/200 Iteration: 550 Loss: 7.475924015045166
Epoch: 1/200 Iteration: 600 Loss: 6.885001182556152

```

```
Epoch: 1/200 Iteration: 650 Loss: 7.147774696350098
Epoch: 1/200 Iteration: 700 Loss: 7.100455284118652
Epoch: 1/200 Iteration: 750 Loss: 7.00004243850708
Epoch: 1/200 Iteration: 800 Loss: 6.786848545074463
Epoch: 1/200 Iteration: 850 Loss: 6.651426792144775
Epoch: 1/200 Iteration: 900 Loss: 6.785276889801025
Epoch: 1/200 Iteration: 950 Loss: 6.836668968200684
```

In [ ]: