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

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

Финансовый университет, 2021 г.

При подготовке лекции использованы материалы:

- ...

v 0.1 15.04.21

Разделы:

Нормализация

- к оглавлению

In [1]:

```
# 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'))
```

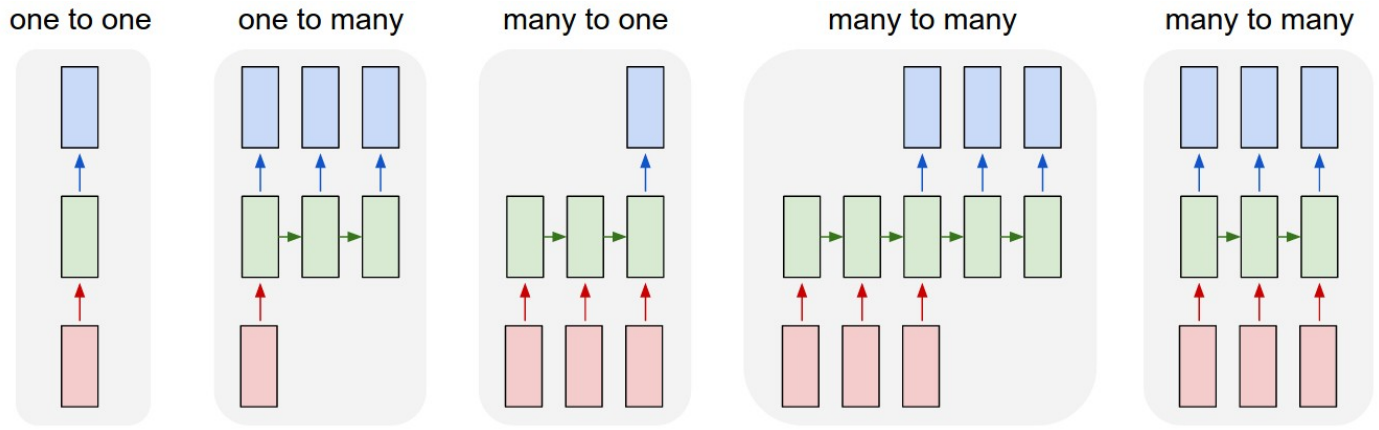
Out[1]:

Введение

- 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.

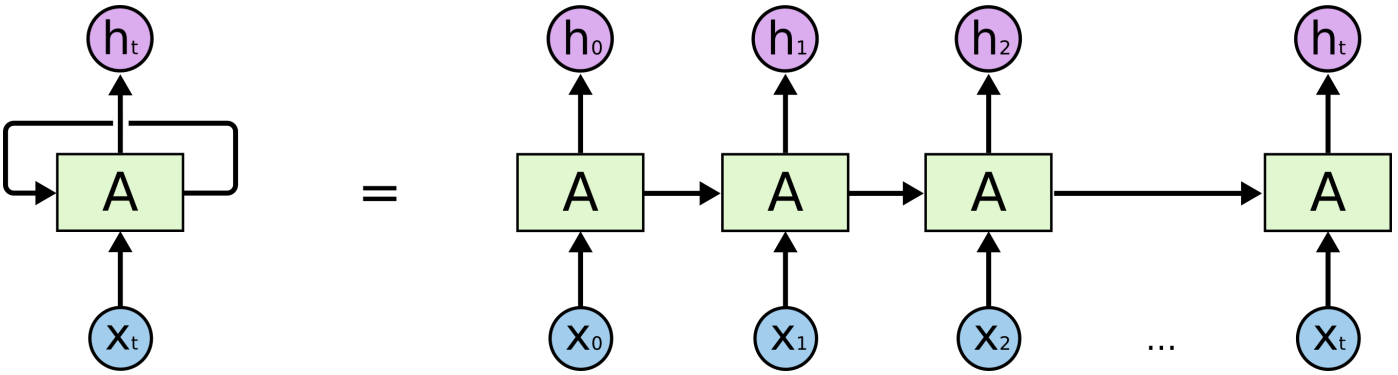
1. один вход, один выход (классический случай)
2. один вход, последовательность выходов
3. последовательность входов, один выход
4. последовательность входов, затем последовательность выходов
5. синхронизированные последовательности входов и выходов

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 сосотояния, 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

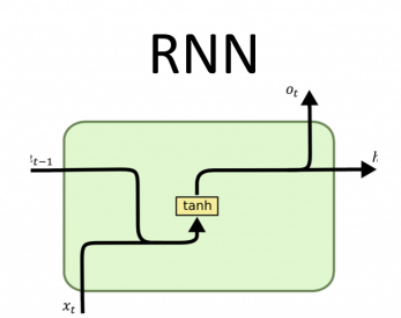
- Ткая архитектура сети позволяет обрабатывать серии событий во времени или последовательные пространственные цепочки произвольной размерности



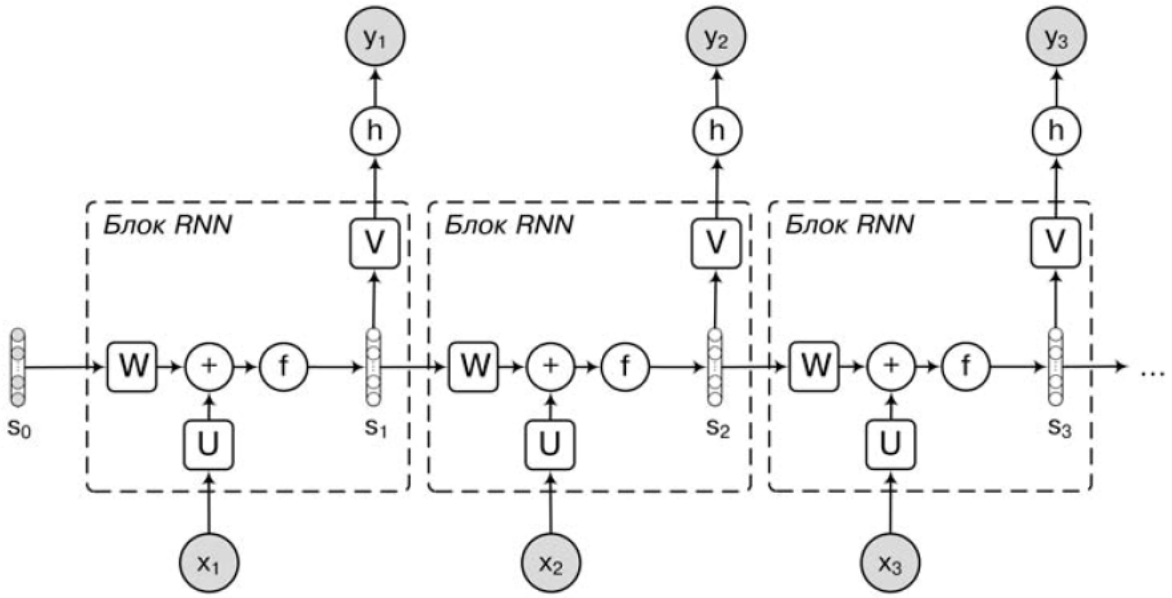
Принцип устройства узла RNN

Трудность рекуррентной сети:

- если учитывать каждый шаг времени, то становится необходимым для каждого шага времени (последовательности) создавать свой слой нейронов, что создает серьёзные 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



Распространение ошибок в узле RNN



Распространение ошибок в узле RNN

$$\begin{aligned} \boldsymbol{a}_t &= \boldsymbol{b} + W\boldsymbol{s}_{t-1} + U\boldsymbol{x}_t, & \boldsymbol{s}_t &= f(\boldsymbol{a}_t), \\ \boldsymbol{o}_t &= \boldsymbol{c} + V\boldsymbol{s}_t, & \boldsymbol{y}_t &= h(\boldsymbol{o}_t), \end{aligned}$$

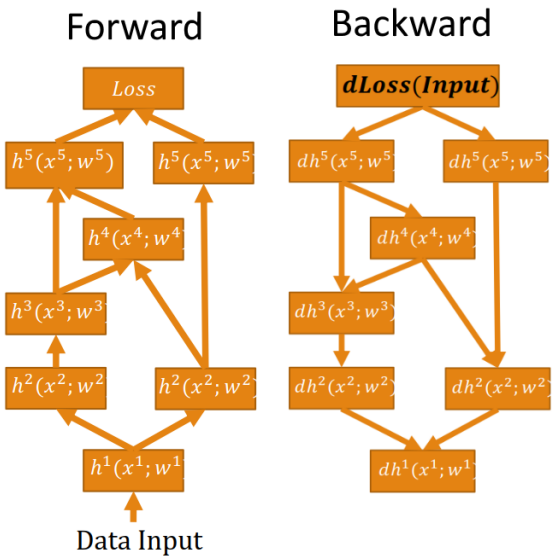
где f — это нелинейность собственно рекуррентной сети (обычно σ, tanh или ReLU), а h — функция, с помощью которой получается ответ (например, softmax).

In [ ]:

In [ ]:

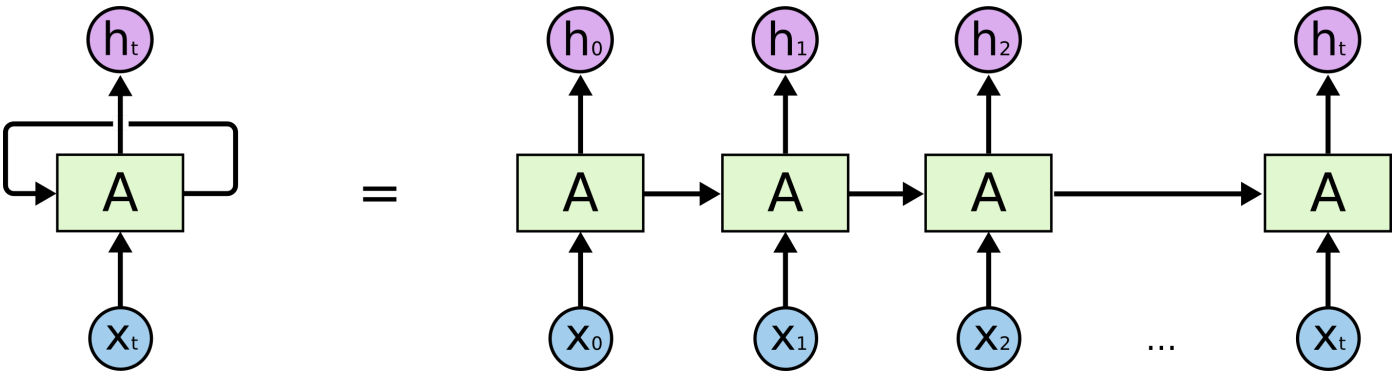
Распространение ошибки в архитектуре RNN

- 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

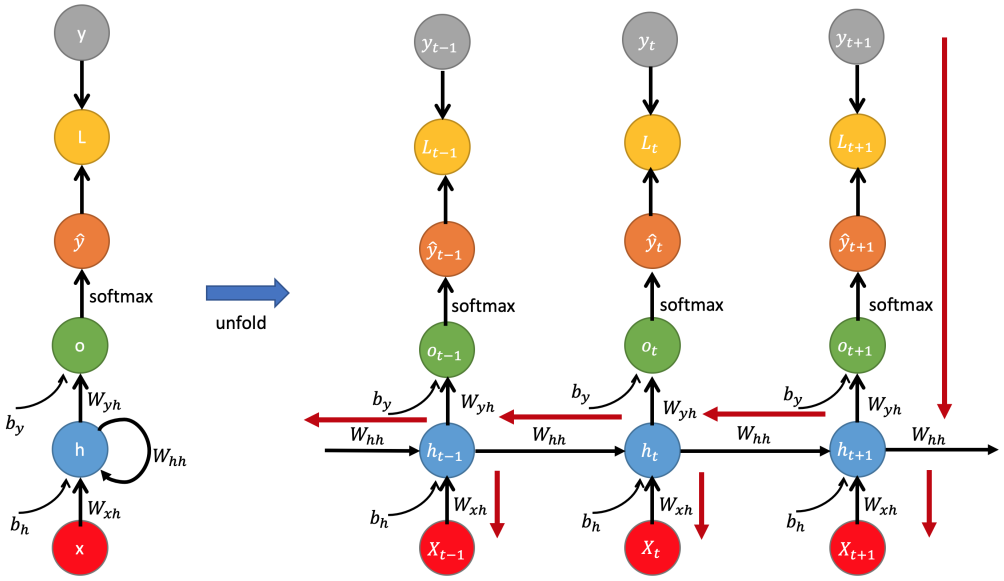
- В архитектуре RNN нейрон принимает в качестве входа результат вычисления в нем самом (через вектор состояния)
  - Важно понимать, что при этом петли в графе вычислений не образуется
  - Вычисления, которые делает рекуррентная сеть, можно развернуть обратно до начала обрабатываемой последовательности
  - 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 — разворчивается вдоль элементов последовательности $1 \dots T$ в очень-очень многоуровневую обычную сеть, в которой одни и те же веса переиспользуются на каждом уровне.

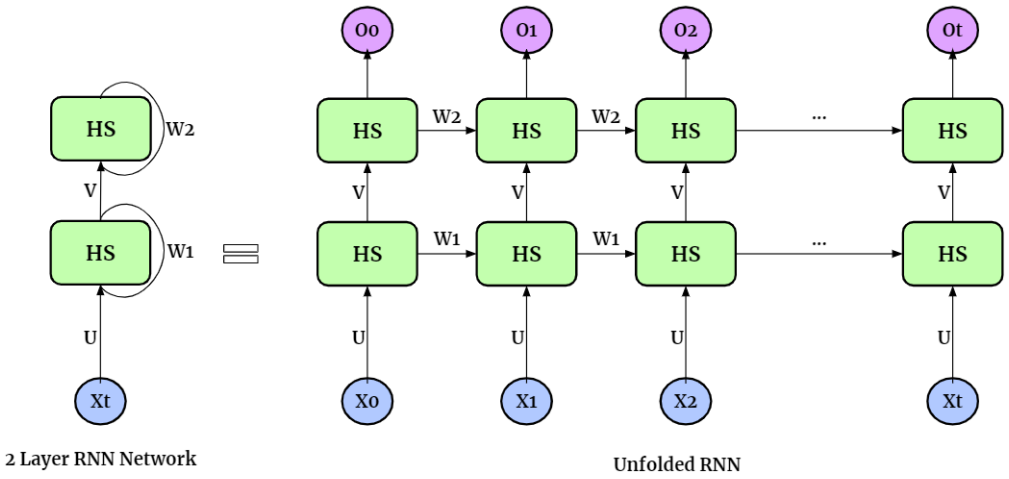
- Для хранения весов достаточно одной матрицы
- Градиенты по весам не затухают до нуля сразу же (как это бывает в обычных глубких сетях)
- Если матрица весов меняет норму вектора градиента при проходе через один «слой» обратного распространения, то при проходе через T слоев эта норма изменяется экспоненциально (т.к. веса матрицы одни и те же) это приводит:
  - к **"взрыу градиентов"** (exploding gradients), если матрица заметно увеличивает норму вектора градиента
  - к экспоненациональному затуханию градиентов (Vanishing gradients), если матрица заметно уменьшает норму вектора градиента



Распространение ошибок в узле RNN

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.

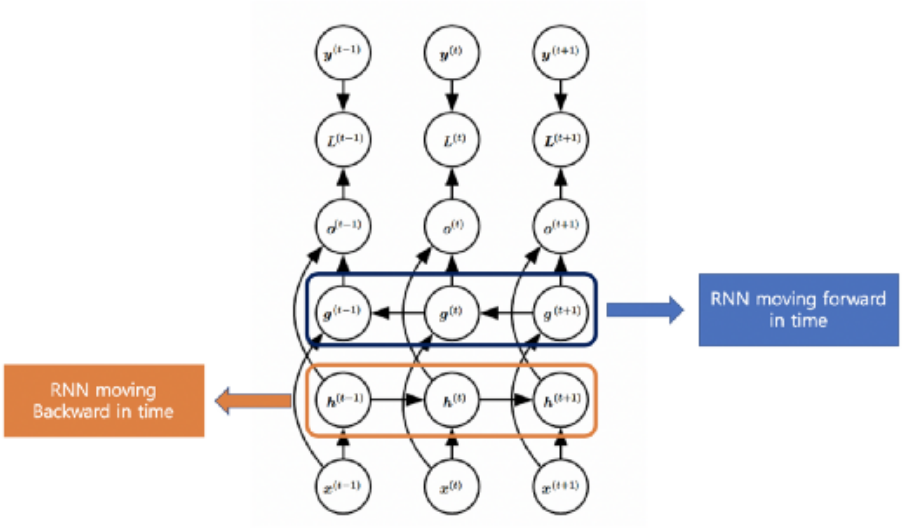


Распространение ошибок в узле RNN

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

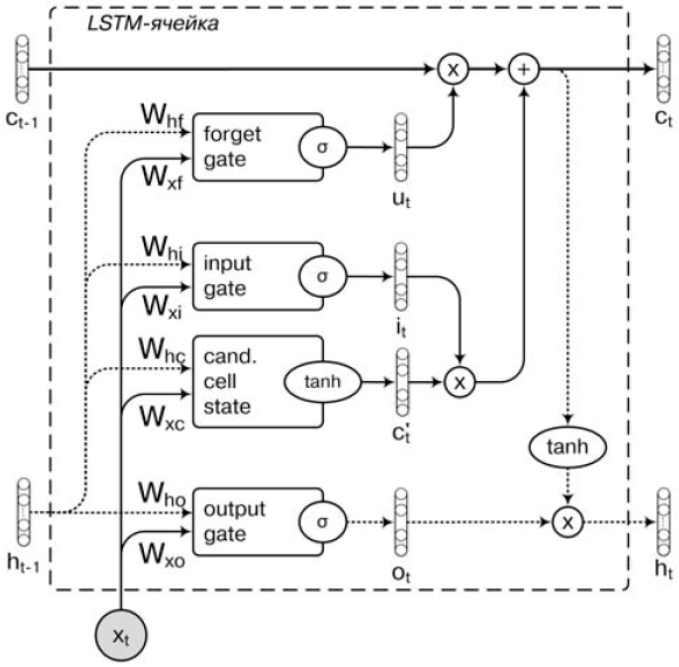
LSTM

LSTM (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 ft в начале обучения будут близки к нулю и градиенты будут свободно распространяться вдоль всей последовательности.

Существует много разных вариантов LSTM: * * *

In [ ]:

```
```

In [103]:

```
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 [139]:

```
# 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
# 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 [130]:

```
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 [131]:

```
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 [132]:

```
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))
```

In [140]:

```
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 [141]:

```
# 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 [142]:

```
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 [143]:

```
device
```

Out[143]:

```
device(type='cuda')
```

In [144]:

```
int_to_vocab[422]
```

Out[144]:

```
'себе.'
```

Основной цикл:

In [146]:

```
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: 8.648141860961914
Epoch: 0/200 Iteration: 100 Loss: 8.640448570251465
Epoch: 0/200 Iteration: 150 Loss: 8.291933059692383
Epoch: 0/200 Iteration: 200 Loss: 7.8641743659973145
Epoch: 0/200 Iteration: 250 Loss: 7.96994686126709
Epoch: 0/200 Iteration: 300 Loss: 8.477243423461914
```

```
Epoch: 0/200 Iteration: 350 Loss: 8.105463027954102
Epoch: 0/200 Iteration: 400 Loss: 8.251829147338867
Epoch: 0/200 Iteration: 450 Loss: 7.938296318054199
Epoch: 0/200 Iteration: 500 Loss: 7.646656036376953
Epoch: 0/200 Iteration: 550 Loss: 7.409121513366699
Epoch: 1/200 Iteration: 600 Loss: 6.838401794433594
Epoch: 1/200 Iteration: 650 Loss: 6.988497734069824
Epoch: 1/200 Iteration: 700 Loss: 6.953343868255615
Epoch: 1/200 Iteration: 750 Loss: 6.912400722503662
Epoch: 1/200 Iteration: 800 Loss: 6.809935092926025
Epoch: 1/200 Iteration: 850 Loss: 6.687542915344238
Epoch: 1/200 Iteration: 900 Loss: 6.830479145050049
Epoch: 1/200 Iteration: 950 Loss: 6.851489543914795
```

In [147]:

```
words
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-147-730c51b3e7e0> in <module>
----> 1 words

NameError: name 'words' is not defined
```

In [ ]:

```

```

In [40]:

```

```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-40-0dcfc47a1015> in <module>
      1 for _ in range(100):
----> 2     ix = torch.tensor([[choice]]).to(device)
      3     output, (state_h, state_c) = model(ix, (state_h, state_c))
      4
      5     _, top_ix = torch.topk(output[0], k=top_k)

NameError: name 'choice' is not defined
```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

Type *Markdown* and LaTeX: α^2

```
, который становится .
```

-
- 123

- 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.

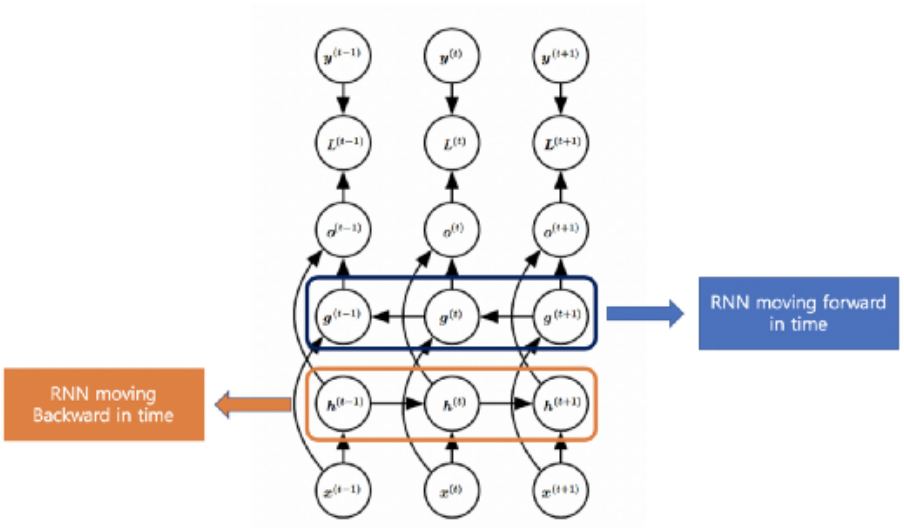
c_0 = 0 и h_0 = 0 (\circ обозначает [[произведение Адамара]]): :

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

[[Функция активации|Функции активации]]:

- \sigma_g: на основе [[Сигмоида|сигмоиды]].
- \sigma_c: на основе [[Гиперболический тангенс|гиперболического тангенса]].
- \sigma_h: на основе гиперболического тангенса, но в работе о глазках (смотровых отверстиях) для LSTM предполагается, что \sigma_h(x) = x.

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. Относительная невосприимчивость к длительности временных разрывов даёт LSTM преимущество по отношению к альтернативным рекуррентным нейронным сетям, скрытым марковским моделям и другим методам обучения для последовательностей в различных сферах применения



Распространение ошибок в узле RNN

In [ ]:

хорошо, если бы

нейронная сеть могла запоминать что-то из истории приходящих на вход данных, сохранять некое внутреннее состояние, которое можно было бы потом использовать для предсказания будущих элементов последовательности

Type *Markdown* and LaTeX: α^2

Эмбеддинг (embedding) ~ векторное представление

связи между элементами образуют направленную последовательность.

в которых соединения между узлами образуют ориентированный граф во временной последовательности.

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.

Если схематично, слой RNN использует цикл for для итерации по упорядоченной по времени последовательности, храня при этом во внутреннем состоянии, закодированную информацию о шагах, которые он уже видел.

Рекуррентная нейронная сеть - это тип нейронной сети глубокого обучения, которая запоминает входную последовательность, сохраняет ее в состояниях памяти / состояниях ячеек и предсказывает будущие слова / предложения.

это класс искусственных нейронных сетей, в которых соединения между узлами образуют ориентированный граф во временной последовательности. Это позволяет ему демонстрировать динамическое поведение во времени. Полученные из нейронных сетей прямого распространения, RNN могут использовать свое внутреннее состояние (память) для обработки последовательностей входных данных переменной длины.

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.

Problem: Решение задачи машинного обучения (с учителем) требует конструирования признаков для каждой конкретной задачи!

Трудность рекуррентной сети заключается в том, что если учитывать каждый шаг времени, то становится необходимым для каждого шага времени создавать свой слой нейронов, что вызывает серьёзные вычислительные сложности. Кроме того, многослойные реализации оказываются вычислительно неустойчивыми, так как в них как правило исчезают или зашкаливают веса. Если ограничить расчёт фиксированным временным окном, то полученные модели не будут отражать долгосрочных трендов.

In [ ]:

```
```

In [107]:

```
# Импорты:
import torch
from torch import nn

import numpy as np
import csv
```

In [108]:

```
# Входные данные:

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

In [128]:

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

In [129]:

```
text[:3]
```

Out[129]:

```
['агафья', 'аглая', 'агния']
```

In [137]:

```
# Кодировка символов числами:

# Join all the sentences together and extract the unique characters from the combined sente
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 [138]:

```
print(char2int)
```

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

In [139]:

```
# 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 th
# 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 [140]:

```
# 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
```

In [141]:

```
# 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 [142]:

```
print(input_seq, dict_size, seq_len, batch_size)
```

```
['агафья   ', 'аглая    ', 'агния    ', 'агриппина', 'акулина  ', 'алевтина
', 'александр', 'алина    ', 'алла     ', 'анастасия', 'ангелина ', 'анжела
', 'анжелика ', 'анна     ', 'антонина ', 'анфиса   ', 'валентина', 'валерия
', 'варвара  ', 'василиса ', 'вера     ', 'вероника ', 'виктория ', 'галина
', 'глафира  ', 'гликерия ', 'дана     ', 'дарья    ', 'евгения  ', 'евдокия
', 'евлалия  ', 'евлампия ', 'евпраксия', 'евфросини', 'екатерина', 'елена
', 'елизавета', 'епистима ', 'ермиония ', 'жанна    ', 'зинаида  ', 'злата
', 'зоя      ', 'инга     ', 'инесса   ', 'инна     ', 'иоанна   ', 'ираида
', 'ирина    ', 'ия       ', 'капитолин', 'карина   ', 'каролина ', 'кира
', 'клавдия  ', 'ксения   ', 'лада     ', 'лариса   ', 'лидия    ', 'лилия
', 'любовь   ', 'людмила  ', 'маргарита', 'марина   ', 'мария    ', 'марфа
', 'матрёна  ', 'милица   ', 'мирослава', 'надежда  ', 'наталья  ', 'нина
', 'нонна    ', 'оксана   ', 'октябрина', 'олимпиада', 'ольга    ', 'павлина
', 'пелагея  ', 'пинна    ', 'полина   ', 'прасковья', 'рада     ', 'раиса
', 'римма    ', 'светлана ', 'серафима ', 'снежана  ', 'софия    ', 'таисия
', 'тамара   ', 'татьяна  ', 'улита    ', 'ульяна   ', 'урсула   ', 'фаина
', 'феврония ', 'фёкла    ', 'феодора  ', 'целестина', 'юлия     ', 'яна
', 'ярослава '] 25 9 103
```

In [143]:

```
# 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 Encoding

# --- --- --- ---
# 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 Encodin
```

In [144]:

```
# torch.cuda.is_available() checks and returns a Boolean True if a GPU is available, else i
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 l
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 [145]:

```
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 forw
        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 earlie
        return hidden
```

In [146]:

```
# 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 [ ]:

```
```

In [ ]:

```
```

In [147]:

```
output, hidden = model(input_seq)
```

In [148]:

```
target_seq.view(-1), target_seq.size()
```

Out[148]:

```
(tensor([ 8.,  5., 14., 24.,  4., 11., 11., 11., 11.,  8.,  2.,  5.,  4., 1
1.,
        11., 11., 11., 11.,  8., 20.,  9.,  4., 11., 11., 11., 11., 11.,
8.,
        12.,  9., 15., 15.,  9., 20.,  5., 11., 23., 13.,  2.,  9., 20.,
5.,
        11., 11., 11.,  2., 19., 18., 22.,  9., 20.,  5., 11., 11.,  2., 1
9.,
        23., 16.,  5., 20., 25., 12.,  5.,  2.,  9., 20.,  5., 11., 11., 1
1.,
        11., 11.,  2.,  2.,  5., 11., 11., 11., 11., 11., 11., 20.,  5., 1
6.,
        22.,  5., 16.,  9.,  4., 11., 20.,  8., 19.,  2.,  9., 20.,  5., 1
1.,
        11., 20., 10., 19.,  2.,  5., 11., 11., 11., 11., 20., 10., 19.,
2.,
         9., 23.,  5., 11., 11., 20., 20.,  5., 11., 11., 11., 11., 11., 1
1.,
        20., 22.,  0., 20.,  9., 20.,  5., 11., 11., 20., 14.,  9., 16.,
5.,
        11., 11., 11., 11.,  5.,  2., 19., 20., 22.,  9., 20.,  5., 11.,
5.,
         2., 19., 12.,  9.,  4., 11., 11., 11.,  5., 12., 18.,  5., 12.,
5.,
        11., 11., 11.,  5., 16.,  9.,  2.,  9., 16.,  5., 11., 11., 19., 1
2.,
         5., 11., 11., 11., 11., 11., 11., 19., 12.,  0., 20.,  9., 23.,
5.,
        11., 11.,  9., 23., 22.,  0., 12.,  9.,  4., 11., 11.,  5.,  2.,
9.,
        20.,  5., 11., 11., 11., 11.,  2.,  5., 14.,  9., 12.,  5., 11., 1
1.,
        11.,  2.,  9., 23., 19., 12.,  9.,  4., 11., 11.,  5., 20.,  5., 1
1.,
        11., 11., 11., 11., 11.,  5., 12., 24.,  4., 11., 11., 11., 11., 1
1.,
        18.,  8., 19., 20.,  9.,  4., 11., 11., 11., 18., 25.,  0., 23.,
9.,
         4., 11., 11., 11., 18.,  2.,  5.,  2.,  9.,  4., 11., 11., 11., 1
8.,
         2.,  5.,  6., 15.,  9.,  4., 11., 11., 18., 15., 12.,  5., 23., 1
6.,
         9.,  4., 11., 18., 14., 12.,  0., 16.,  9., 20.,  9.,  4., 23.,
5.,
        22., 19., 12.,  9., 20.,  5., 11.,  2., 19., 20.,  5., 11., 11., 1
1.,
        11., 11.,  2.,  9.,  1.,  5., 18., 19., 22.,  5., 11., 15.,  9., 1
6.,
        22.,  9.,  6.,  5., 11., 11., 12.,  6.,  9.,  0., 20.,  9.,  4., 1
1.,
        11.,  5., 20., 20.,  5., 11., 11., 11., 11., 11.,  9., 20.,  5.,
9.,
        25.,  5., 11., 11., 11.,  2.,  5., 22.,  5., 11., 11., 11., 11., 1
1.,
         0.,  4., 11., 11., 11., 11., 11., 11., 11., 20.,  8.,  5., 11., 1
```

```
1.,
       11., 11., 11., 11., 20., 19., 16., 16.,  5., 11., 11., 11., 11., 2
0.,
       20.,  5., 11., 11., 11., 11., 11., 11.,  0.,  5., 20., 20.,  5., 1
1.,
       11., 11., 11., 12.,  5.,  9., 25.,  5., 11., 11., 11., 11., 12.,
9.,
       20.,  5., 11., 11., 11., 11., 11.,  4., 11., 11., 11., 11., 11., 1
1.,
       11., 11.,  5., 15.,  9., 22.,  0.,  2.,  9., 20.,  5.,  5., 12.,
9.,
       20.,  5., 11., 11., 11., 11.,  5., 12.,  0.,  2.,  9., 20.,  5., 1
1.,
       11.,  9., 12.,  5., 11., 11., 11., 11., 11., 11.,  2.,  5., 18., 2
5.,
        9.,  4., 11., 11., 11., 16., 19., 20.,  9.,  4., 11., 11., 11., 1
1.,
        5., 25.,  5., 11., 11., 11., 11., 11., 11.,  5., 12.,  9., 16.,
5.,
       11., 11., 11., 11.,  9., 25.,  9.,  4., 11., 11., 11., 11., 11.,
9.,
        2.,  9.,  4., 11., 11., 11., 11., 11., 17.,  7.,  0., 18., 24., 1
1.,
       11., 11., 11., 17., 25.,  6.,  9.,  2.,  5., 11., 11., 11.,  5., 1
2.,
        8.,  5., 12.,  9., 22.,  5., 11.,  5., 12.,  9., 20.,  5., 11., 1
1.,
       11., 11.,  5., 12.,  9.,  4., 11., 11., 11., 11., 11.,  5., 12., 1
4.,
        5., 11., 11., 11., 11., 11.,  5., 22., 12.,  3., 20.,  5., 11., 1
1.,
       11.,  9.,  2.,  9., 21.,  5., 11., 11., 11., 11.,  9., 12.,  0., 1
6.,
        2.,  5., 18.,  5., 11.,  5., 25., 19., 10., 25.,  5., 11., 11., 1
1.,
        5., 22.,  5.,  2., 24.,  4., 11., 11., 11.,  9., 20.,  5., 11., 1
1.,
       11., 11., 11., 11.,  0., 20., 20.,  5., 11., 11., 11., 11., 11., 2
3.,
       16.,  5., 20.,  5., 11., 11., 11., 11., 23., 22.,  4.,  7., 12.,
9.,
       20.,  5., 11.,  2.,  9.,  6., 15.,  9.,  5., 25.,  5., 11.,  2., 2
4.,
        8.,  5., 11., 11., 11., 11., 11.,  5., 18.,  2.,  9., 20.,  5., 1
1.,
       11., 11., 19.,  2.,  5.,  8., 19.,  4., 11., 11., 11.,  9., 20., 2
0.,
        5., 11., 11., 11., 11., 11.,  0.,  2.,  9., 20.,  5., 11., 11., 1
1.,
       11., 12.,  5., 16., 23.,  0., 18., 24.,  4., 11.,  5., 25.,  5., 1
1.,
       11., 11., 11., 11., 11.,  5.,  9., 16.,  5., 11., 11., 11., 11., 1
1.,
        9.,  6.,  6.,  5., 11., 11., 11., 11., 11., 18., 19., 22.,  2.,
5.,
       20.,  5., 11., 11., 19., 12.,  5., 14.,  9.,  6.,  5., 11., 11., 2
0.,
       19., 10.,  5., 20.,  5., 11., 11., 11.,  0., 14.,  9.,  4., 11., 1
1.,
       11., 11., 11.,  5.,  9., 16.,  9.,  4., 11., 11., 11., 11.,  5.,
6.,
```

```
          5., 12.,  5., 11., 11., 11., 11.,  5., 22., 24.,  4., 20.,  5., 1
1.,
         11., 11.,  2.,  9., 22.,  5., 11., 11., 11., 11., 11.,  2., 24.,
4.,
         20.,  5., 11., 11., 11., 11., 12., 16., 13.,  2.,  5., 11., 11., 1
1.,
         11.,  5.,  9., 20.,  5., 11., 11., 11., 11., 11., 19., 18., 12.,
0.,
         20.,  9.,  4., 11., 11.,  3., 23.,  2.,  5., 11., 11., 11., 11., 1
1.,
         19.,  0., 25.,  0., 12.,  5., 11., 11., 11., 19.,  2., 19., 16., 2
2.,
          9., 20.,  5., 11.,  2.,  9.,  4., 11., 11., 11., 11., 11., 11., 2
0.,
          5., 11., 11., 11., 11., 11., 11., 11., 12.,  0., 16.,  2.,  5., 1
8.,
          5., 11., 11.]),
 torch.Size([103, 9]))
```

In [ ]:

In [ ]:

In [ ]:

In [149]:

```
# 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: 2.6348
Epoch: 20/100..... Loss: 2.1549
Epoch: 30/100..... Loss: 1.9723
Epoch: 40/100..... Loss: 1.7659
Epoch: 50/100..... Loss: 1.5775
Epoch: 60/100..... Loss: 1.4626
Epoch: 70/100..... Loss: 1.3804
Epoch: 80/100..... Loss: 1.3168
Epoch: 90/100..... Loss: 1.2665
Epoch: 100/100..... Loss: 1.2247
```

In [ ]:

```

```

In [150]:

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

    return int2char[char_ind], hidden
```

In [151]:

```
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 [163]:

```
sample(model, 45, 'гут')
```

Out[163]:

```
'гутана                                          '
```

In [ ]:

In [ ]:

In [ ]: