Практическое занятие 15-16

**РЕКУРРЕНТНАЯ LSTM СЕТЬ**

**Задание на практическую работу**

1. Изучить теорию по рекуррентным нейронным сетям (РНС), включая

ячейки LSTM и GRU.

2. Каждый студент группы должен самостоятельно написать 50 высказываний (25 – положительных, 25– отрицательных). В итоге размер

обучающей выборки должен составить: 50∙N наблюдений, где N – число студентов в группе.

Примечание: текст ниоткуда не копировать (высказывания повторяться не должны), писать самостоятельно, используя литературный язык, подойти ответственно.

3. Реализовать РНС с использованием LSTM ячейки.

4. В процессе обучения РНС на выходе должна давать оценку окраски текста (положительный или отрицательный).

5. При обучении РНС контролировать процесс переобучения и устранять его (при необходимости) путем изменения структуры сети, а также (возможно) с помощью Dropout и Batch Normalization.

6. Оценить результаты работы полученной РНС на тестовой выборке.

7. Сделать то же самое обучение РНС, но на базе ячейки GRU. Сравнить скорость обучения и качество работы с предыдущей реализацией на базе LSTM.

8. В ходе работы добиться как можно лучших результатов сентимент-анализа текстовых высказываний.

**Методические указания**

Рассмотренная простейшая рекуррентная НС – это не самая лучшая архитектура. Основным ее недостатком является быстрое «забывание» прошлого контекста. А иногда он важен. Например, такой текст:

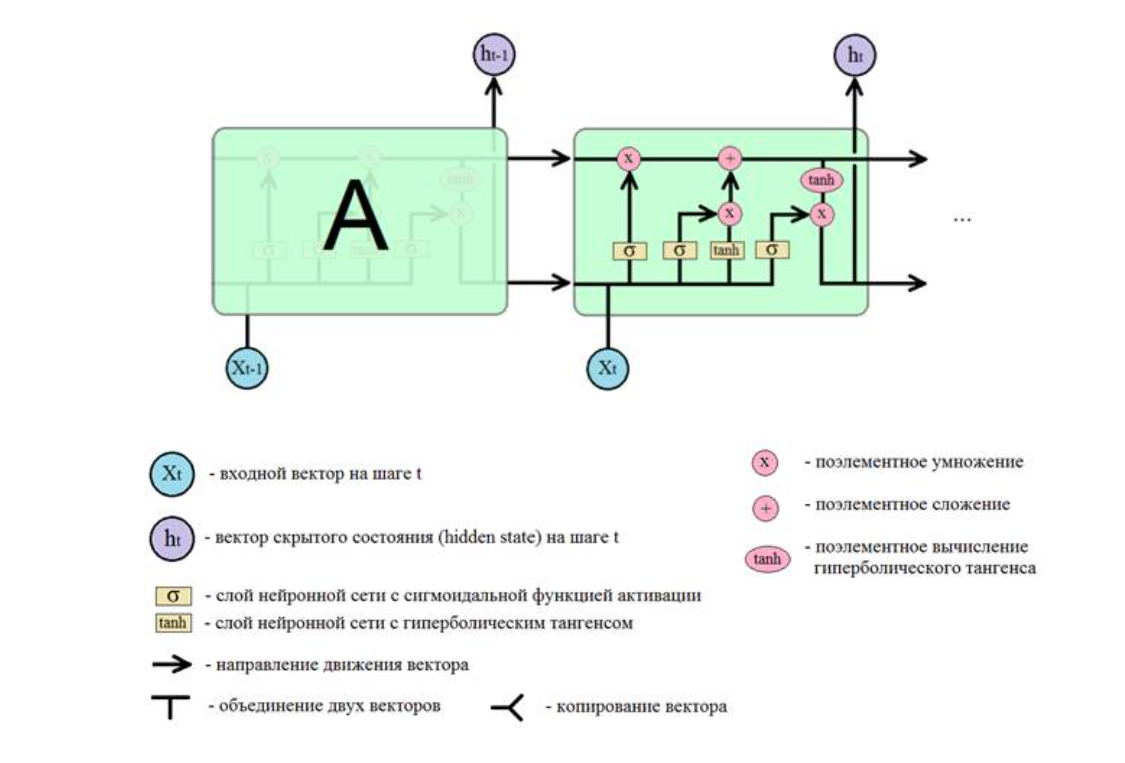
Я очень люблю программировать, …, поэтому в будущем хочу стать программистом.

Смотрите, здесь вначале упоминается любовь к программированию, а затем идут какие-то отвлеченные слова. И только в конце снова появляется слово «программистом», связанное по смыслу с программированием. Так вот, чтобы сеть имела возможность спрогнозировать это слово, исходя из контекста, она должна четко схватывать нужный прошлый контекст и не обращать внимания на второстепенный. Простейшая архитектура RNN плохо учитывает такие моменты. Она, как правило, все смешивает в одну кучу и то, что встречалось вначале быстро теряется. Хотя, исследователи говорят, что это скорее недостаток алгоритма обучения, а не архитектуры. Было показано на простых задачах, что теоретически веса можно настроить так, чтобы сеть корректно учитывала даже длинный и сложный контекст. Но на практике этого достичь не удается. Поэтому в 1997 году Зеппом Хохрайтер и Юргеном Шмидхубером (Jürgen Schmidhuber) была предложена другая архитектура известная под названием «долгая краткосрочная память»:

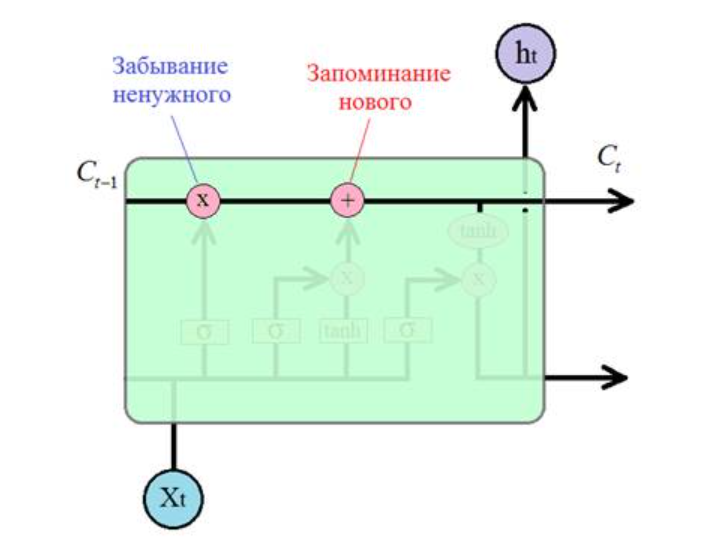
LSTM (Long short-term memory)

которая при обучении способна схватывать существенные детали прошлого контекста и сохранять их, пока они актуальны. Эта архитектура была основной «рабочей лошадкой» рекуррентных сетей вплоть до 2017 года. И сегодня остается весьма востребованной во многих прикладных задачах. Давайте с ней познакомимся поближе.

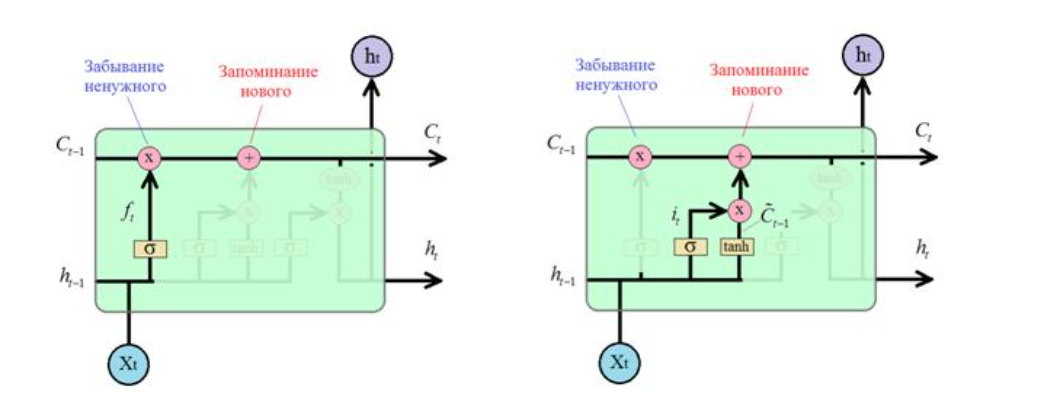
В базовом исполнении рекуррентный блок LSTM выглядит следующим образом (эти блоки еще называют ячейками):

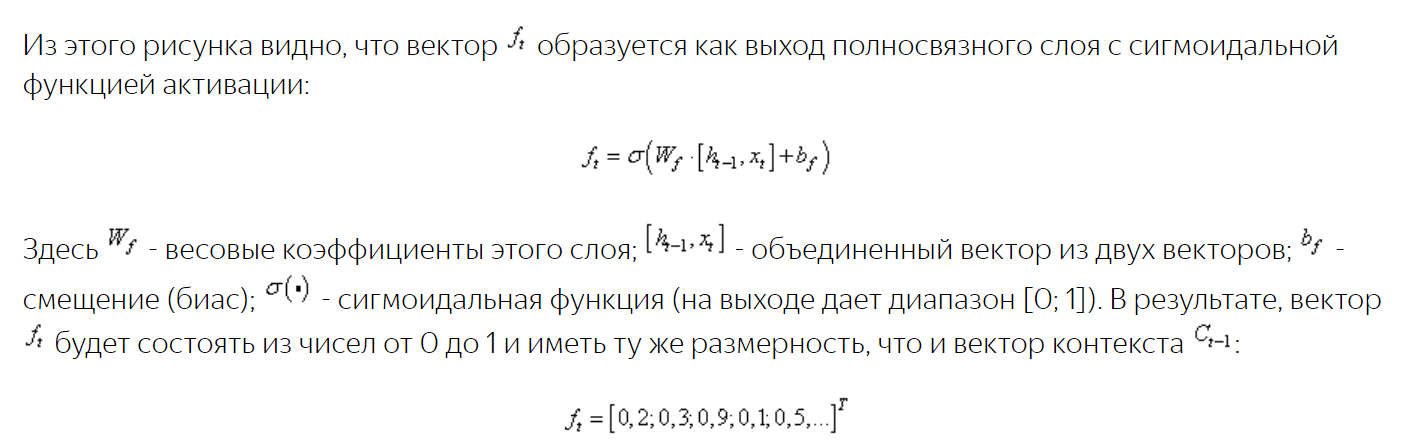


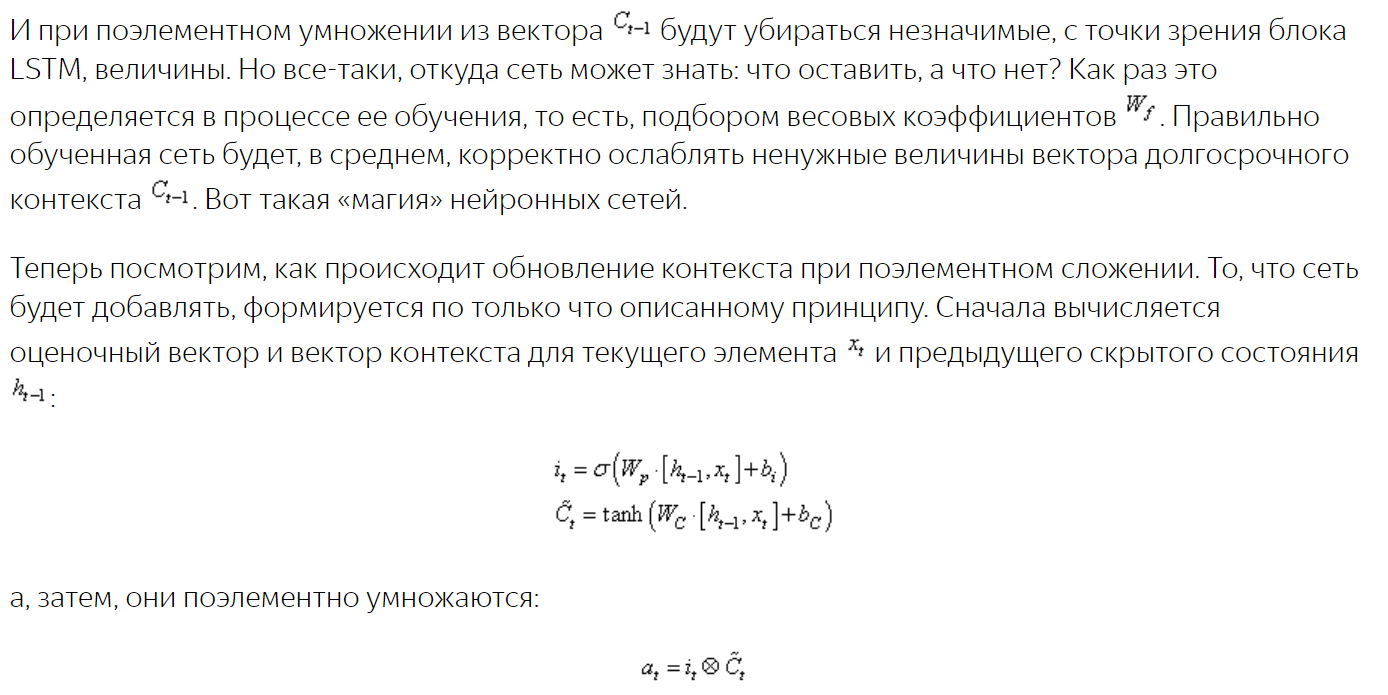
Давайте теперь разберем принцип работы этой ячейки. Отличительной чертой архитектуры LSTM является наличие вот этой верхней трубы, по которой, как бы движется вектор контекста:

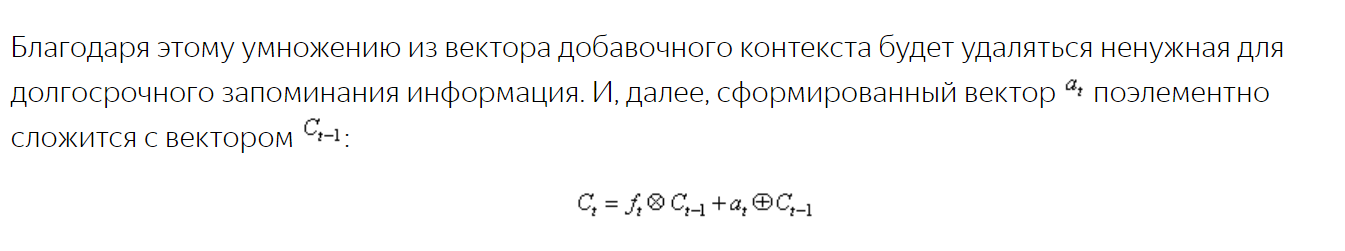


Благодаря ее наличию LSTM-блок имеет возможность сохранять и передавать долгосрочный контекст дальше по рекурсии. Естественно, возникают вопросы: как и почему при поэлементном умножении происходит забывание чего-то ненужного, а при поэлементном сложении – запоминание чего-то нового? Сначала разберемся с забыванием. На вход операции умножения поступают два вектора: один - вектор контекста, а второй - - оценочный вектор:

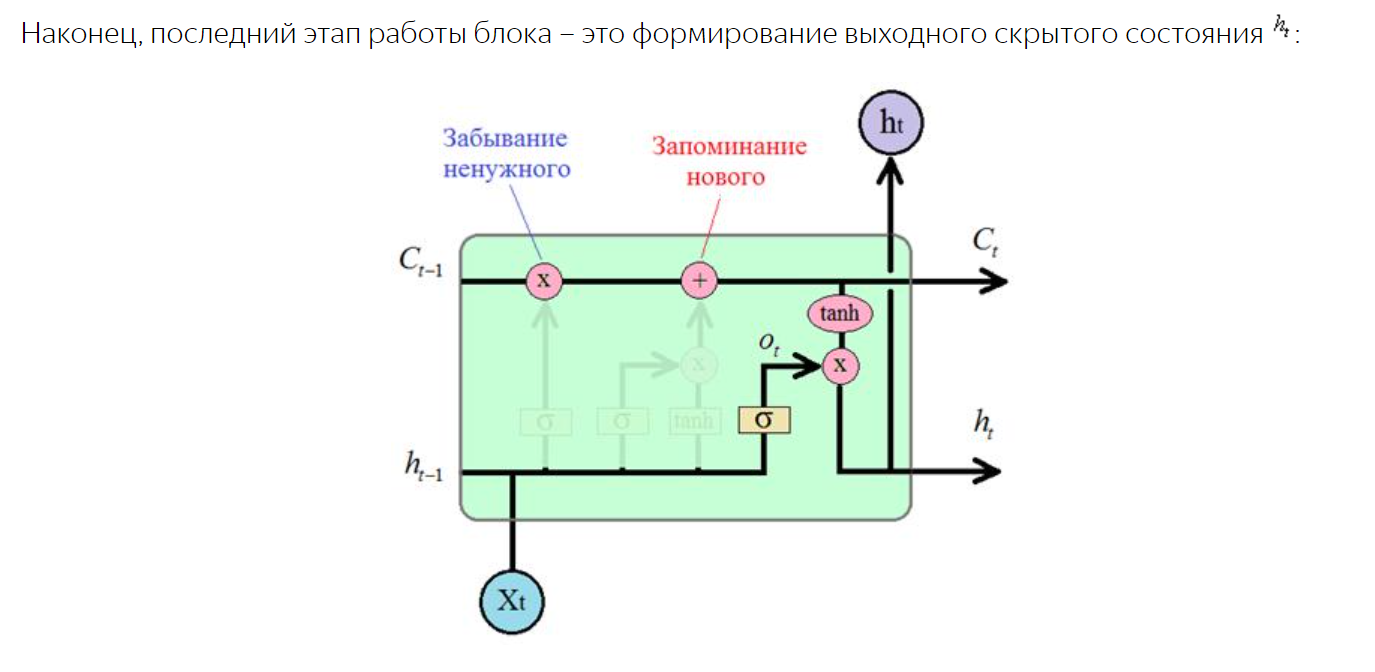


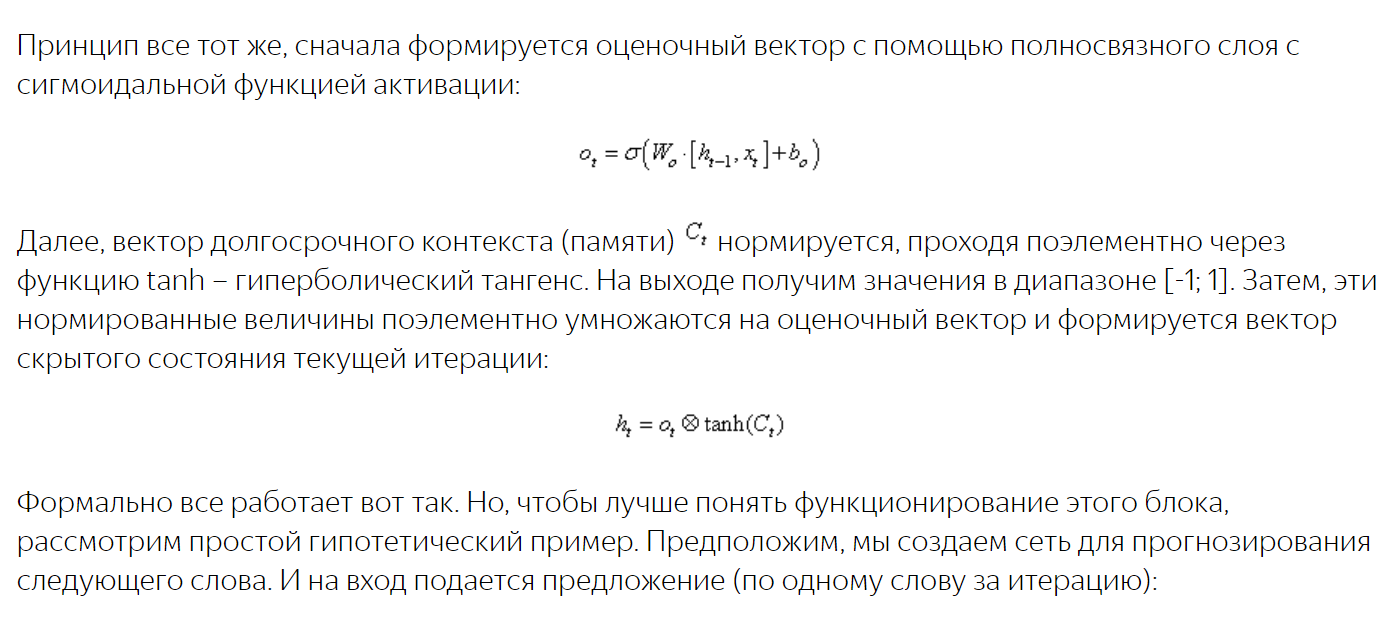




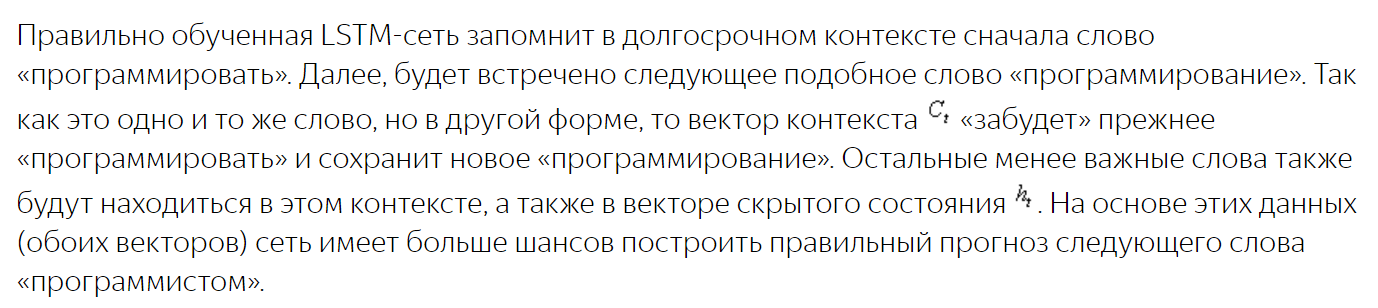


Так в блоке LSTM меняется долгосрочный контекст от итерации к итерации.





Я очень люблю программировать, программирование – это круто, поэтому в будущем хочу стать программистом.

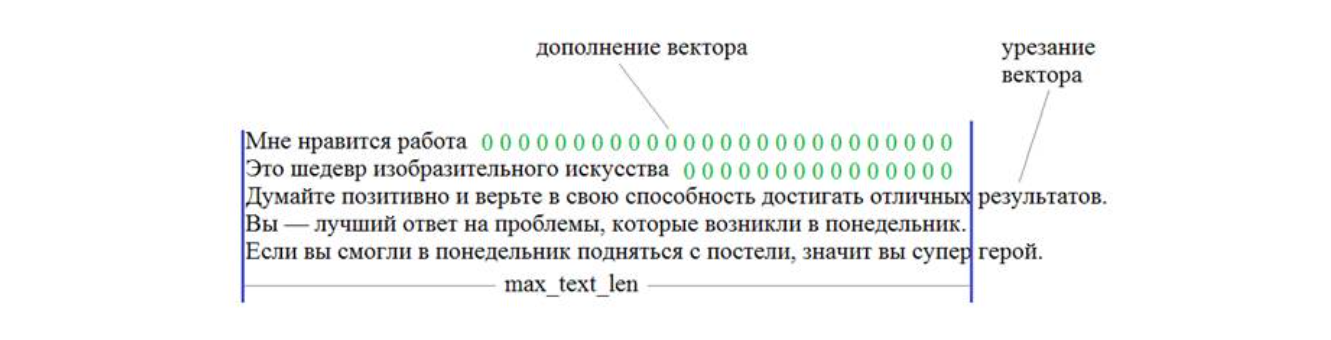


Конечно, слушая о порядке работы блока LSTM и описание примера, невольно напрашивается вопрос: а почему сеть будет работать именно так, как мы хотим, как мы задумываем? Почему бы после корректного обучения ей не работать как-то иначе, по другим принципам, используя ту же самую архитектуру? Здесь есть одна «великая магия». Элементы LSTM-блока при обучении будут приобретать те признаки, которым их легче обучить. Например, гораздо проще реализовать «забывание» контекста через умножение, чем через сложение, поэтому именно умножение в итоге будет отвечать за «забывание». А сложение – за добавление чего-то нового. Здесь разделение обязанностей происходит по принципу: где проще, там и делается. Это как вода, которая течет по своему руслу. Ей же никто не говорил течь именно так, по ложбинке? Это определяется законами физики, гравитацией, в результате которой мы наблюдаем именно такое ее поведение – ей «проще» течь по руслу, чем по холмам. Также и весовые коэффициенты НС приобретают функциональность, которую им проще реализовать. В результате, архитектура LSTM работает в целом так, как задумано ее создателями.

На самом деле вся эта магия хорошо сочетается с алгоритмом градиентного спуска и если детально разобрать процесс обучения НС, то это станет вполне очевидным фактом.

Мы рассмотрели с вами лишь базовую (простейшую) архитектуру LSTM-блока. На практике применяют разные их разновидности. Но все они работают по похожему принципу. На следующем занятии мы продолжим эту тему и рассмотрим решение задачи сентимент-анализа коротких текстовых сообщений с помощью архитектуры LSTM.

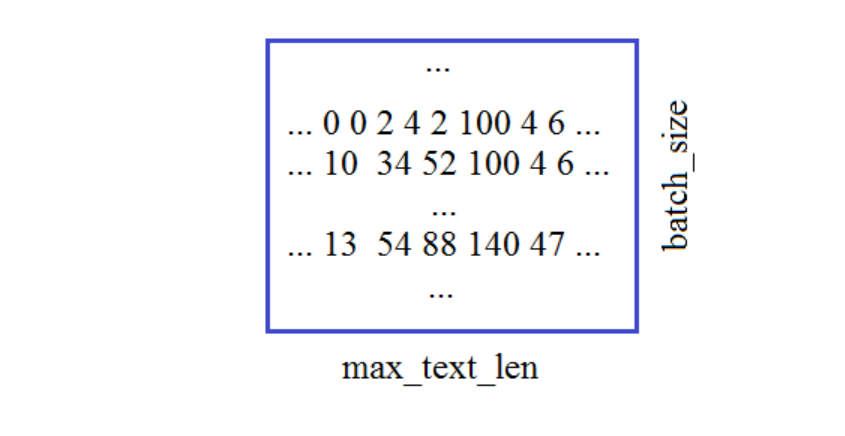
На этом занятии продолжаем тему обработки текстовой информации и построим сеть для оценки эмоциональной окраски текста: положительный или отрицательный. Для начала нам нужно подготовить обучающую выборку. Что она из себя будет представлять? Смотрите, у нас есть два файла: с положительными и отрицательными короткими высказываниями. Длины фраз различны. Но на вход сети нужно подавать тензор со строго определенными размерами. Поэтому, короткие фразы дополняются нулями, а длинные – обрезаются:



Переменная max\_text\_len как раз и определяет размер каждой фразы (по числу слов). В итоге, у нас будет входной двумерный тензор, размерностью

(batch\_size, max\_text\_len)

содержащий индексы слов фразы из сформированного словаря:



Давайте теперь посмотрим, как можно сформировать такой тензор. Вначале загрузим тексты с положительными и отрицательными высказываниями (они будут находиться в отдельных файлах: train\_data\_true и train\_data\_false):

with open('train\_data\_true', 'r', encoding='utf-8') as f:

texts\_true = f.readlines()

texts\_true[0] = texts\_true[0].replace('\ufeff', '') #убираем первый невидимый символ

with open('train\_data\_false', 'r', encoding='utf-8') as f:

texts\_false = f.readlines()

texts\_false[0] = texts\_false[0].replace('\ufeff', '')

Далее, сразу объединим списки высказываний в единый список и вычислим длину каждого из них:

texts = texts\_true + texts\_false

count\_true = len(texts\_true)

count\_false = len(texts\_false)

print(count\_true, count\_false)

Теперь нам нужно разбить эти высказывания на слова. Для этого воспользуемся уже знакомым инструментом Tokenizer и положим, что максимальное число слов будет равно 1000:

maxWordsCount = 1000

tokenizer = Tokenizer(num\_words=maxWordsCount, filters='!–"—#$%&amp;()\*+,-./:;<=>?@[\\]^\_`{|}~\t\n\r«»',

lower=True, split=' ', char\_level=False)

tokenizer. fit\_on\_texts (texts)

По идее, мы здесь могли бы и не задавать максимальное число слов, тогда эта величина была бы определена автоматически при парсинге текста. Но данный параметр имеет один существенный плюс: из всех найденных слов мы оставляем 999 наиболее часто встречаемых (maxWordsCount-1), то есть, отбрасываем редкие слова, которые особо не нужны при обучении НС.

Конечно, в данном случае, останутся все найденные слова, т.к. их общее число меньше 1000. Вообще, этот параметр устанавливается с позиции «здравого смысла». Например, при большой обучающей выборке хорошим выбором будет значение в пределах:

maxWordsCount = 20000

Итак, мы разбили текст на слова и для примера выведем их начальный список с частотами появления:

dist = list(tokenizer.word\_counts.items())

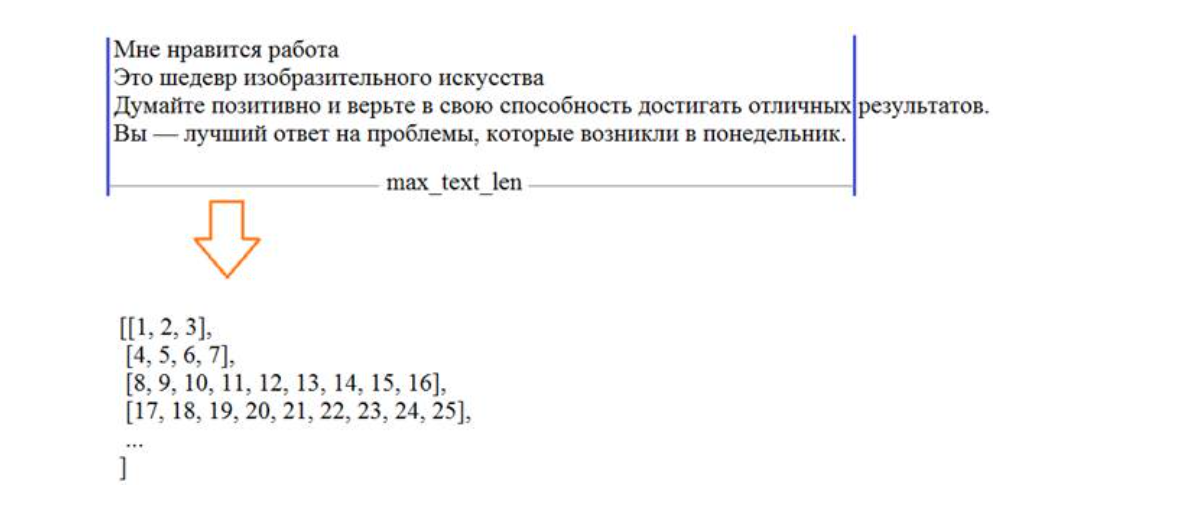
print(dist[:10])

print(texts[0][:100])

Далее, преобразуем текст в последовательность чисел в соответствии с полученным словарем. Для этого используется специальный метод класса Tokenizer:

data = tokenizer.texts\_to\_sequences(texts)

На выходе получим двумерный массив чисел объекта numpy:



Теперь, нам нужно выровнять все эти векторы до длины max\_text\_len. Для этого используется еще один встроенный метод pad\_sequences, который обрезает массив data до длины max\_text\_len и добавляет нули для коротких векторов:

max\_text\_len = 10

data\_pad = pad\_sequences(data, maxlen=max\_text\_len)

print(data\_pad)

print(data\_pad.shape)

Все, получили двухмерный тензор обучающей выборки. Сформируем еще один тензор для требуемых выходных значений сети. Кодировать ответы будем по правилу:



Сформировать набор таких выходных данных можно следующим образом:

X = data\_pad

Y = np.array([[1, 0]]\*count\_true + [[0, 1]]\*count\_false)

print(X.shape, Y.shape)

Все, у нас есть обучающая выборка и требуемые выходные значения. Для лучшего обучения перемешаем все эти высказывания, чтобы на вход подавались вперемешку и положительные и отрицательные. Так сеть будет лучше обучаться:

indeces = np.random.choice(X.shape[0], size=X.shape[0], replace=False)

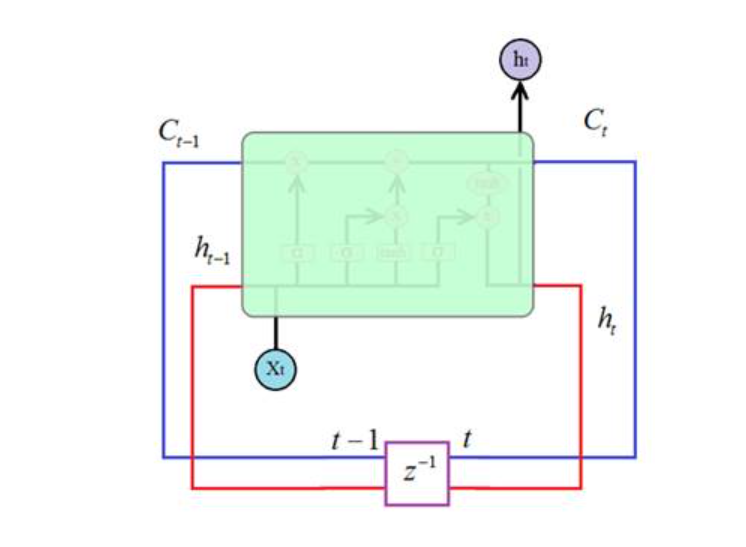
X = X[indeces]

Y = Y[indeces]

Осталось создать модель рекуррентной сети. Мы воспользуемся рекуррентным слоем LSTM, о котором говорили на предыдущем занятии. Для его создания в Keras используется класс:

keras.layers.LSTM(units, …)

В качестве первого параметра (units) указывается число нейронов в каждом полносвязном слое внутри LSTM-ячейки:



Они же будут формировать размерность выходного вектора . Например, создадим LSTM-слой с 64 нейронами. Тогда размерность выходного вектора будет равна 64 элемента. Далее, добавим еще один такой же слой с 32 нейронами. У нас получится стек из двух рекуррентных слоев.

Подробное описание параметров LSTM слоя в Keras смотрите на странице русскоязычной документации:

https://ru-keras.com/recurrent-layers/

На выходе поставим полносвязный слой с двумя нейронами и функцией активации softmax. Определим оптимизацию по Adam с шагом сходимости 0,0001 (одна десятитысячная):

model = Sequential()

model.add(Embedding(maxWordsCount, 128, input\_length = max\_text\_len))

model.add(LSTM(64, activation='tanh', return\_sequences=True))

model.add(LSTM(32, activation='tanh'))

model.add(Dense(2, activation='softmax'))

model.summary()

model.compile(loss='categorical\_crossentropy', metrics=['accuracy'], optimizer=Adam(0.0001))

Готово. Запускаем процесс обучения:

history = model.fit(X, Y, batch\_size=32, epochs=50)

И давайте теперь посмотрим, что у нас получилось. Сформируем какой-нибудь текст и преобразуем его во входной формат нашей сети:

t = "я люблю позитивное настроение".lower()

data = tokenizer.texts\_to\_sequences([t])

data\_pad = pad\_sequences(data, maxlen=max\_text\_len)

Посмотрим, что осталось из этого предложения (здесь могут использоваться слова, которых нет в нашем словаре – словарь формируется при парсинге текстов):

print( sequence\_to\_text(data[0]) )

А функция sequence\_to\_text будет следующая:

reverse\_word\_map = dict(map(reversed, tokenizer.word\_index.items()))

def sequence\_to\_text(list\_of\_indices):

words = [reverse\_word\_map.get(letter) for letter in list\_of\_indices]

Мы здесь сначала сформировали словарь, в котором сначала идут индексы, а затем, слова. С его помощью функция sequence\_to\_text преобразует последовательность индексов в слова.

Ну и, наконец, пропускаем входной вектор через сеть и на выходе получаем результат:

res = model.predict(inp)

print(res, np.argmax(res), sep='\n')

Конечно, сейчас будут получаться не очень хорошие результаты классификации текстов на положительные и отрицательные. Это связано, прежде всего, с малым объемом выборки. Я не нашел в интернете открытую базу русских текстов с их классификацией, поэтому пришлось создавать самому. Терпения у меня хватило только на 80 высказываний одного и другого типа. Это очень мало. Нужно несколько десятков тысяч. Но для этого мне потребовались бы месяцы упорной работы. Поэтому то, что здесь есть – это лишь демонстрация, пример решения задачи сентимент анализа текстов, показан общий подход начальной обработки текста и формирования обучающего множества.

Программа:

import os

os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2'

import numpy as np

import re

from tensorflow.keras.layers import Dense, LSTM, Input, Dropout, Embedding

from tensorflow.keras.models import Sequential

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.preprocessing.text import Tokenizer, text\_to\_word\_sequence

from tensorflow.keras.preprocessing.sequence import pad\_sequences

with open('train\_data\_true', 'r', encoding='utf-8') as f:

texts\_true = f.readlines()

texts\_true[0] = texts\_true[0].replace('\ufeff', '') #убираем первый невидимый символ

with open('train\_data\_false', 'r', encoding='utf-8') as f:

texts\_false = f.readlines()

texts\_false[0] = texts\_false[0].replace('\ufeff', '') #убираем первый невидимый символ

texts = texts\_true + texts\_false

count\_true = len(texts\_true)

count\_false = len(texts\_false)

total\_lines = count\_true + count\_false

print(count\_true, count\_false, total\_lines)

maxWordsCount = 1000

tokenizer = Tokenizer(num\_words=maxWordsCount, filters='!–"—#$%&amp;()\*+,-./:;<=>?@[\\]^\_`{|}~\t\n\r«»', lower=True, split=' ', char\_level=False)

tokenizer.fit\_on\_texts(texts)

dist = list(tokenizer.word\_counts.items())

print(dist[:10])

print(texts[0][:100])

max\_text\_len = 10

data = tokenizer.texts\_to\_sequences(texts)

data\_pad = pad\_sequences(data, maxlen=max\_text\_len)

print(data\_pad)

print( list(tokenizer.word\_index.items()) )

X = data\_pad

Y = np.array([[1, 0]]\*count\_true + [[0, 1]]\*count\_false)

print(X.shape, Y.shape)

indeces = np.random.choice(X.shape[0], size=X.shape[0], replace=False)

X = X[indeces]

Y = Y[indeces]

model = Sequential()

model.add(Embedding(maxWordsCount, 128, input\_length = max\_text\_len))

model.add(LSTM(128, return\_sequences=True))

model.add(LSTM(64))

model.add(Dense(2, activation='softmax'))

model.summary()

model.compile(loss='categorical\_crossentropy', metrics=['accuracy'], optimizer=Adam(0.0001))

history = model.fit(X, Y, batch\_size=32, epochs=50)

reverse\_word\_map = dict(map(reversed, tokenizer.word\_index.items()))

def sequence\_to\_text(list\_of\_indices):

words = [reverse\_word\_map.get(letter) for letter in list\_of\_indices]

return(words)

t = "Я люблю позитивное настроение".lower()

data = tokenizer.texts\_to\_sequences([t])

data\_pad = pad\_sequences(data, maxlen=max\_text\_len)

print( sequence\_to\_text(data[0]) )

res = model.predict(data\_pad)

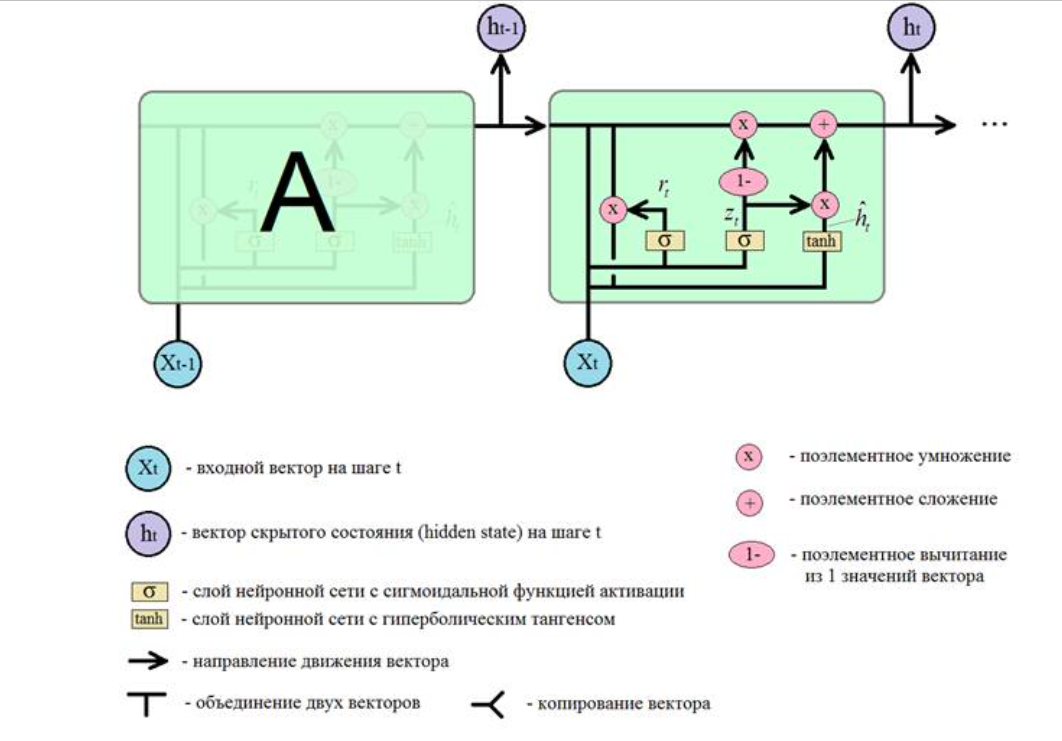
print(res, np.argmax(res), sep='\n')

**РЕКУРРЕНТНЫЕ БЛОКИ GRU. ПРИМЕР ИХ РЕАЛИЗАЦИИ В ЗАДАЧЕ СЕНТИМЕНТ-АНАЛИЗА**

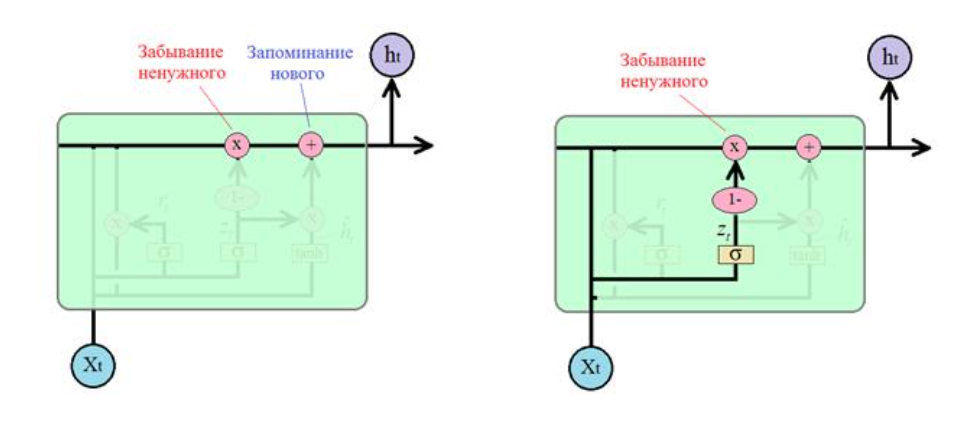
На предыдущих занятиях мы с вами познакомились с популярной рекуррентной архитектурой LSTM. Она эффективна при анализе долгосрочного контента, но имеет один существенный недостаток – большое число настраиваемых параметров (весовых коэффициентов). Это приводит к большим затратам памяти и длительному процессу обучения таких сетей. Поэтому в 2014 году было предложено упрощение LSTM, которое стало известно под названием управляемые рекуррентные блоки:

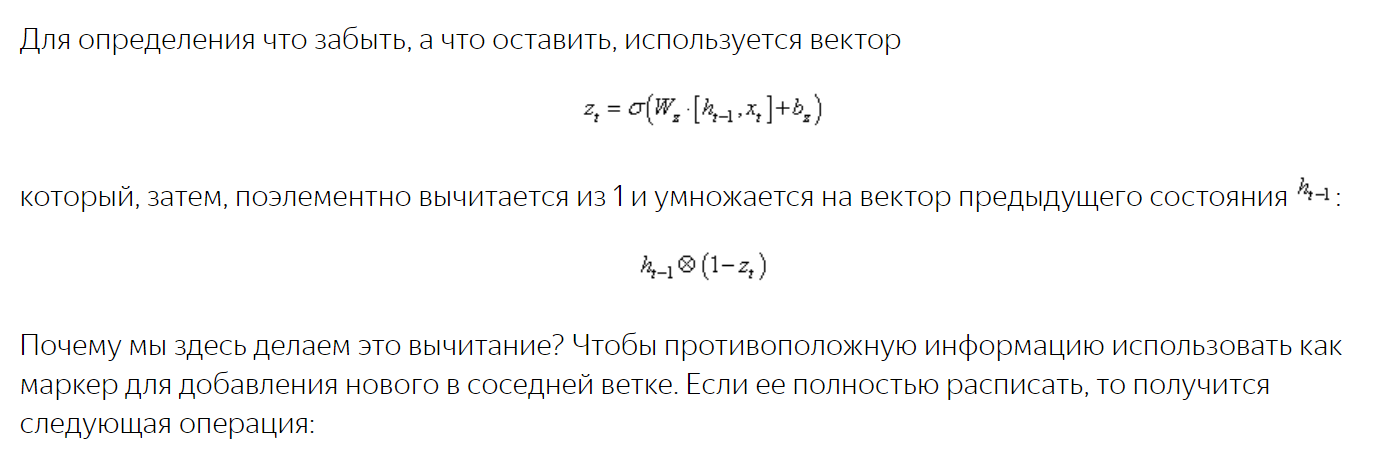
Gated Recurrent Units (GRU)

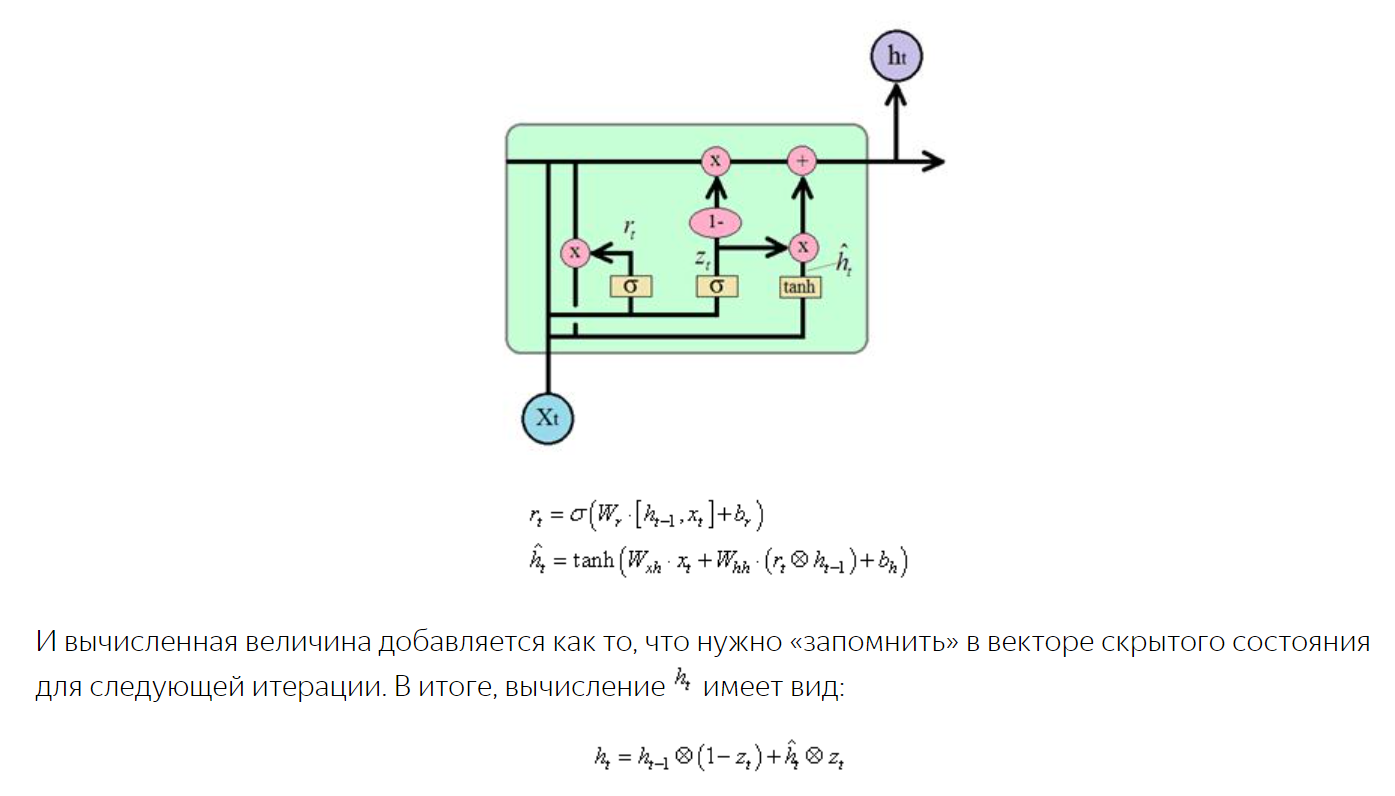
По эффективности блоки GRU сравнимы с LSTM во многих практических задачах: моделирования музыкальных и речевых сигналов, обработка текста и так далее. Существует несколько вариантов этого блока и, как всегда, мы рассмотрим их классическую архитектуру:



Работа этого блока (ячейки) похожа на блок LSTM только здесь долгосрочный элемент памяти реализован не отдельным каналом, а в самом векторе скрытого состояния https://proproprogs.ru/htm/neural_network/files/rekurrentnye-bloki-gru-primer-realizacii-v-zadache-sentiment-analiza.files/image003.png. Наверху мы также видим две поэлементные операции умножения и сложения для забывания ненужной информации и запоминания новой:







Эта формула и определяет принцип работы ячейки GRU. Если кто из вас знаком с калмановской фильтрацией случайных сигналов:

https://youtu.be/nLVgrFq8obM

Формула очень похожа на реализацию фильтра Калмана в дискретном времени. В некотором смысле она работает по этому же принципу: отбрасывает случайные (незначительные) детали и сохраняет главное (важное).

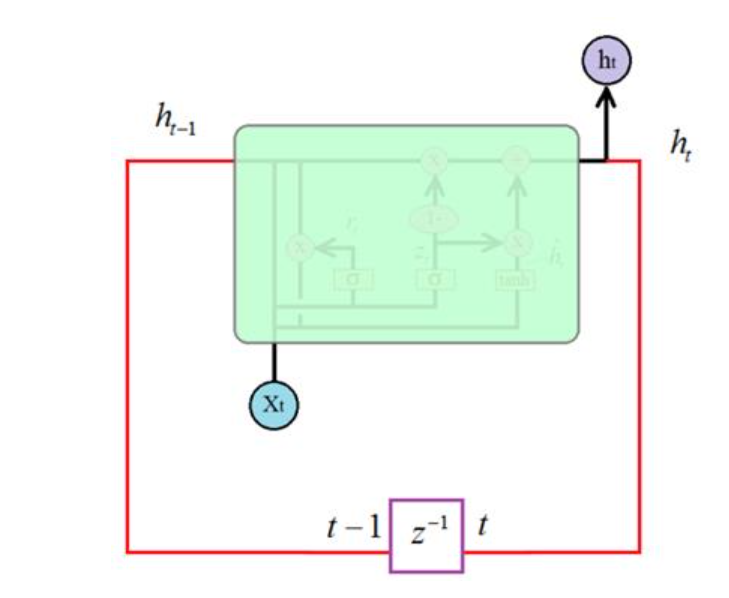
Какую же сеть LSTM или GRU выбирать для практического применения? Все зависит от поставленной задачи, но общая рекомендация такая: сначала лучше воспользоваться сетью GRU, так как она быстрее обучается и если точность решения задачи оказывается недостаточной, то есть смысл попробовать сеть LSTM.

Общим преимуществом сетей LSTM и GRU является решение проблемы исчезающего градиента, характерная для простейшей рекуррентной НС, когда при увеличении числа итераций величина градиента стремится к нулю. В LSTM и GRU благодаря наличию ячейки памяти (возможности хранения долгосрочного контента) градиент не уходит ниже определенного уровня, так как он поддерживается значением сигнала сохраняемого контента.

В Keras рекуррентный слой из ячеек GRU формируется подобно ранее рассмотренному LSTM слою. Для этого используется класс:

keras.layers.GRU(units, …)

в котором первый обязательный параметр units определяет число нейронов внутри GRU ячейки:



Подробное описание параметров GRU слоя в Keras смотрите на странице русскоязычной документации:

<https://ru-keras.com/recurrent-layers/>

Модель сети из предыдущего занятия для сентимент-анализа коротких текстов, можно записать в следующем виде:

model = Sequential()

model.add(Embedding(maxWordsCount, 128, input\_length = max\_text\_len))

model.add(GRU(128, return\_sequences=True))

model.add(GRU(64))

model.add(Dense(2, activation='softmax'))

model.summary()

Как видите, на уровне Keras это делается элементарно: вместо класса LSTM пишется класс GRU и все.

Одним из общих недостатков блоков LSTM и GRU является их переобучение. Так как каждая ячейка содержит большое число нейронов, то сеть, на их основе также получается большой. А это, как мы знаем, прямой путь к переобучению. Чтобы этого избежать, во-первых, нужно контролировать этот эффект по выборке валидации. И, во-вторых, применять инструменты:

Dropout и Batch Normalization

для его предотвращения. Причем, в Keras Dropout можно использовать и для внутренних слоев ячеек, определяя следующие параметры классов LSTM и GRU:

* dropout – для нейронов, связанных с входным вектором https://proproprogs.ru/htm/neural_network/files/rekurrentnye-bloki-gru-primer-realizacii-v-zadache-sentiment-analiza.files/image012.png;
* recurrent\_dropout – для нейронов, связанных с состоянием https://proproprogs.ru/htm/neural_network/files/rekurrentnye-bloki-gru-primer-realizacii-v-zadache-sentiment-analiza.files/image003.png.

По умолчанию оба параметра равны нулю (Dropout отключен). Также можно использовать Dropout между рекуррентными слоями. Инструмент Batch Normalization используется только между слоями (внутри ячеек он не применяется).

Ссылки: [LSTM - долгая краткосрочная память (proproprogs.ru)](https://proproprogs.ru/neural_network/lstm-dolgaya-kratkosrochnaya-pamyat),

[Как делать сентимент-анализ рекуррентной LSTM сетью (proproprogs.ru)](https://proproprogs.ru/neural_network/kak-delat-sentiment-analiz-rekurrentnoy-lstm-setyu)