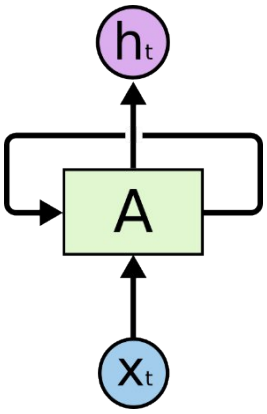
Рекуррентные нейронные сети

Рекуррентные нейронные сети (англ. Recurrent neural network; RNN) — вид нейронных сетей, где связи между элементами образуют направленную последовательность. Благодаря этому появляется возможность обрабатывать серии событий во времени или последовательные пространственные цепочки.

В отличие от многослойных перцептронов, рекуррентные сети могут использовать свою внутреннюю память для обработки последовательностей произвольной длины. Поэтому сети RNN применимы в таких задачах, где нечто целостное разбито на сегменты, например, распознавание рукописного текста или распознавание речи. Было предложено много различных архитектурных решений для рекуррентных сетей от простых до сложных. В последнее время наибольшее распространение получили сеть с долговременной и кратковременной памятью (LSTM) и управляемый рекуррентный блок (GRU).

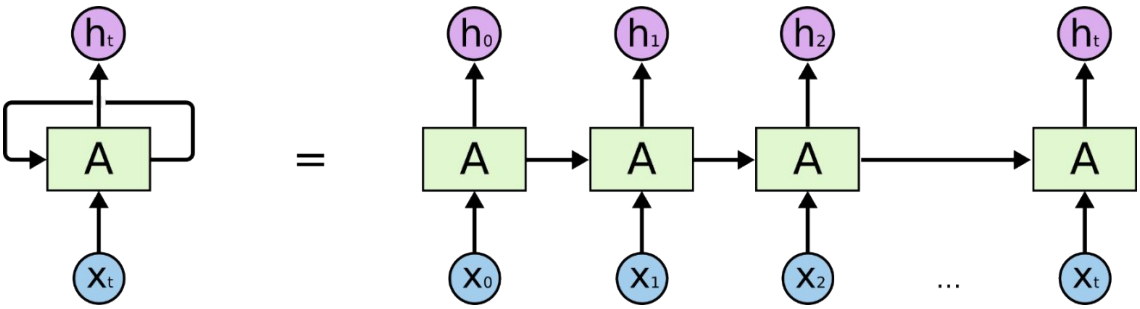


В диаграмме выше участок нейронной сети **A** получает некие данные **X** на вход и подает на выход некоторое значение **H**. Циклическая связь позволяет передавать информацию от текущего шага сети к следующему.

Существует много разновидностей, решений и конструктивных элементов рекуррентных нейронных сетей. 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. Различные подходы пытаются усовершенствовать модель исторической памяти и механизм запоминания и забывания.

Рекуррентные нейронные сети не так уж сильно отличаются от обычных нейронных сетей. Их можно представить себе, как множество копий одной и той же сети, причем, каждая копия передает сообщение следующей копии. Посмотрите, что получится, если мы развернем цикл:



Такая “цепная” сущность показывает, что рекуррентные нейронные сети по природе своей тесно связаны с последовательностями и списками.

1 Полностью рекуррентная сеть

Это базовая архитектура разработана в 1980-х. Сеть строится из узлов, каждый из которых соединён со всеми другими узлами. У каждого нейрона порог активации меняется со временем и является вещественным числом. Каждое соединение имеет переменный вещественный вес. Узлы разделяются на входные, выходные и скрытые.

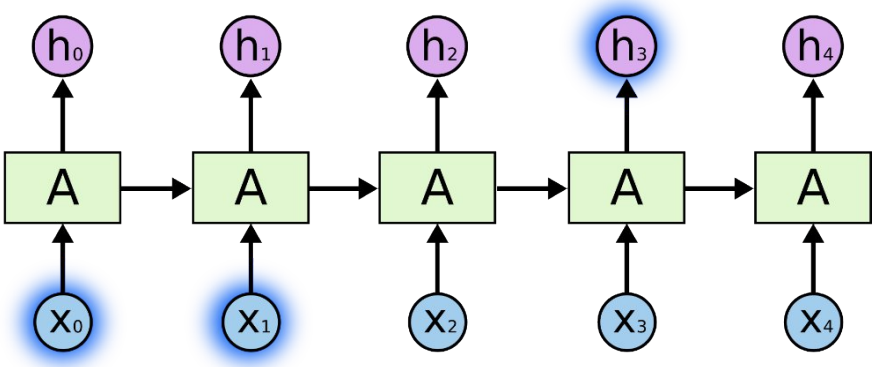
Для **обучения с учителем** с дискретным временем, каждый (дискретный) шаг времени на входные узлы подаются данные, а прочие узлы завершают свою активацию, и выходные сигналы готовятся для передачи нейроном следующего уровня. Если, например, сеть отвечает за распознавание речи, в результате на выходные узлы поступают уже метки (распознанные слова).

В **обучении с подкреплением (reinforcement learning)** нет учителя, обеспечивающего целевые сигналы для сети, вместо этого иногда используется функция годности[en] или функция оценки (reward function), по которой проводится оценка качества работы сети, при этом значения на выходе оказывает

влияние на поведение сети на входе. В частности, если сеть реализует игру, на выходе измеряется количество пунктов выигрыша или оценки позиции. Каждая цепочка вычисляет ошибку как суммарную девиацию по выходным сигналам сети. Если имеется набор образцов обучения, ошибка вычисляется с учётом ошибок каждого отдельного образца.

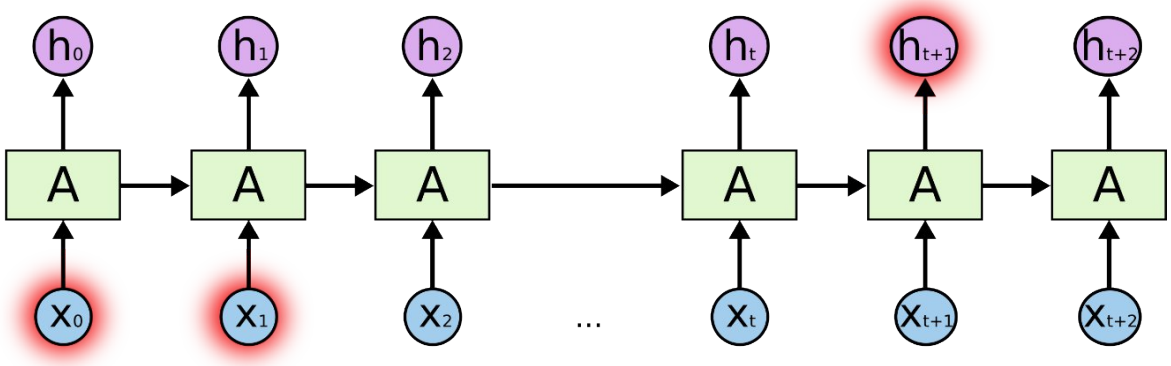
2 Проблема долгосрочных зависимостей

Одна из идей, которая делает РНС столь притягательными, состоит в том, что они могли бы использовать полученную в прошлом информацию для текущих задач. Например, они могли бы использовать предыдущие кадры видео для понимания последующих. Иногда нам достаточно недавней информации, чтобы выполнять текущую задачу. Например, представим модель языка, которая пытается предсказать следующее слово, основываясь на предыдущих. Если мы пытаемся предсказать последнее слово в предложении “Тучи на *небе*”, нам не нужен больше никакой контекст - достаточно очевидно, что в конце предложения речь идёт о небе. В таких случаях, где невелик промежуток между необходимой информацией и местом, где она нужна, РНС могут научиться использовать информацию, полученную ранее.



Но также бывают случаи, когда нам нужен более широкий контекст. Предположим, нужно предсказать последнее слово в тексте “Я вырос во Франции… Я свободно говорю по *французски*”. Недавняя информация подсказывает, что следующее слово, вероятно, название языка, но если мы хотим уточнить, какого именно, нам нужен предыдущий контекст вплоть до информации о Франции. Совсем не редко промежуток между необходимой информацией и

местом, где она нужна, становится очень большим. К сожалению, по мере роста промежутка, РНС становятся неспособны научиться соединять информацию.



Теоретически, РНС способны обрабатывать такие долговременные зависимости. Человек может тщательно подобрать их параметры, чтобы решать игрушечные проблемы такой формы. Однако, на практике, РНС не способны выучить такое.

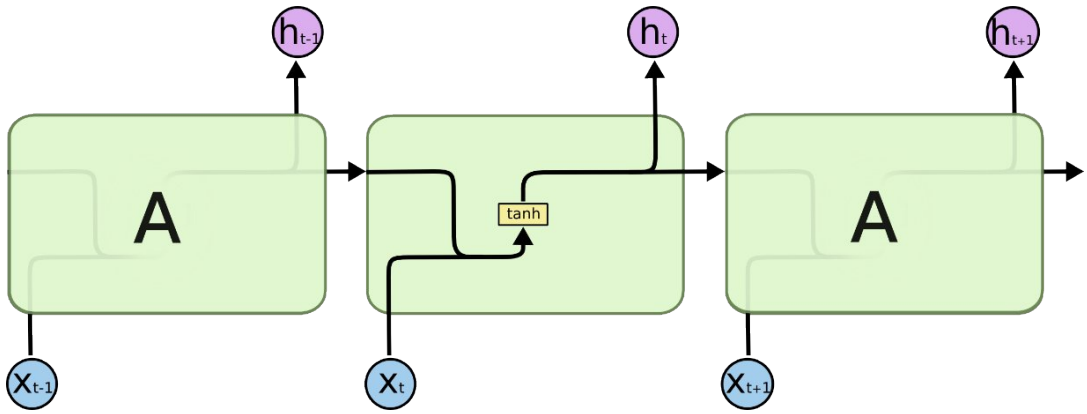
3 LSTM сети

Сети долго-краткосрочной памяти (Long Short Term Memory) - обычно просто называют “LSTM” - особый вид РНС, способных к обучению долгосрочным зависимостям. Они работают невероятно хорошо на большом разнообразии проблем и в данный момент широко применяются. LSTM специально спроектированы таким образом, чтобы избежать проблемы долгосрочных зависимостей. Запоминать информацию на длительный период времени - это практически их поведение по-умолчанию, а не что-то такое, что они только пытаются сделать.

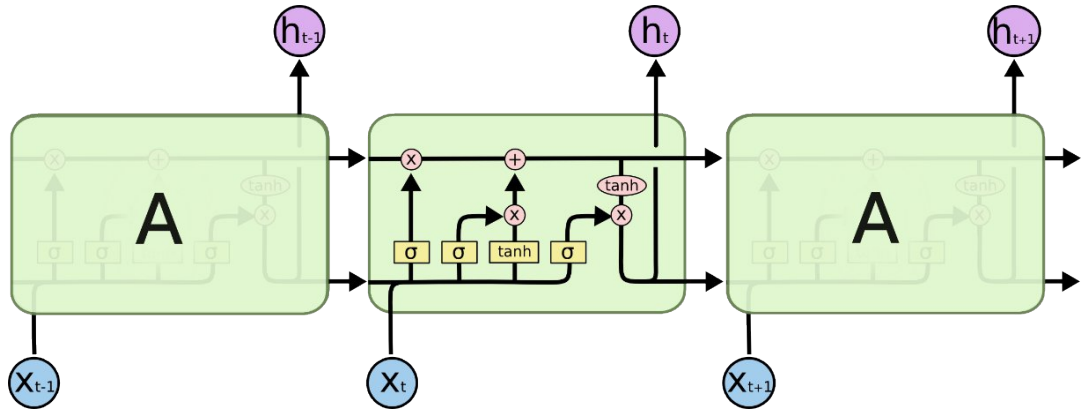
В LSTM сетях удалось обойти проблему исчезновения или зашкаливания градиентов в процессе обучения методом обратного распространения ошибки. Сеть LSTM обычно управляется с помощью рекуррентных вентилей, которые называются вентили (**gates**) «забывания». Ошибки распространяются назад по времени через потенциально неограниченное количество виртуальных слоёв. Таким образом происходит обучение в LSTM, при этом сохраняя память о тысячах и даже миллионах временных интервалов в прошлом. Топологии сетей типа LSTM

могут разрабатываться в соответствии со спецификой задачи. Всети LSTM даже большие задержки между значимыми событиями могут учитываться, и тем самым высокочастотные и низкочастотные компоненты могут смешиваться.

Все рекуррентные нейронные сети имеют форму цепи повторяющих модулей (repeating module) нейронной сети. В стандартной РНС эти повторяющие модули будут иметь очень простую структуру, например, всего один слой гиперболического тангенса (**tanhtanh**).



LSTM тоже имеют такую цепную структуру, но повторяющий модуль имеет другое строение. Вместо одного нейронного слоя их четыре, причем они взаимодействуют особым образом.



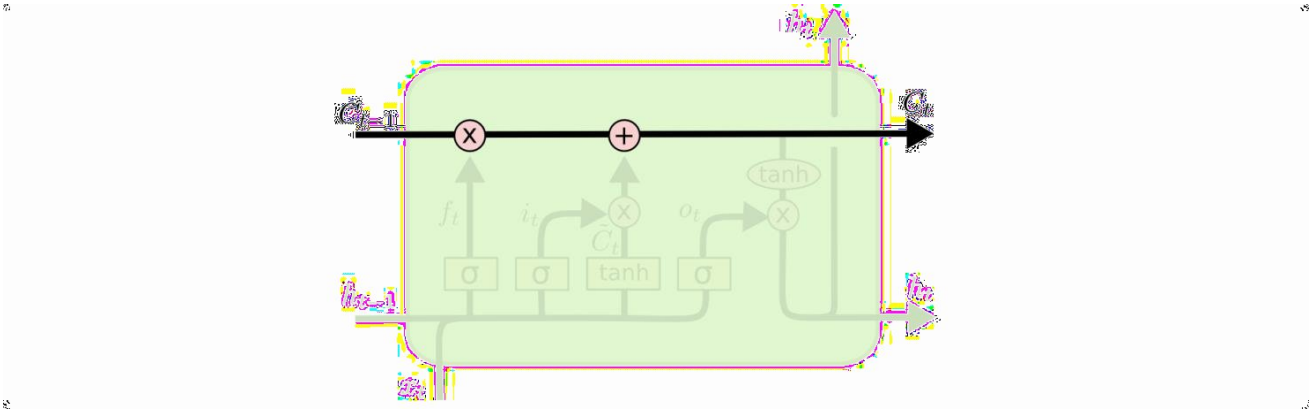
Здесь введены следующие обозначения:

В диаграмме выше каждая линия передает целый вектор от выхода одного узла к входам других. Розовые круги представляют поточечные операторы, такие

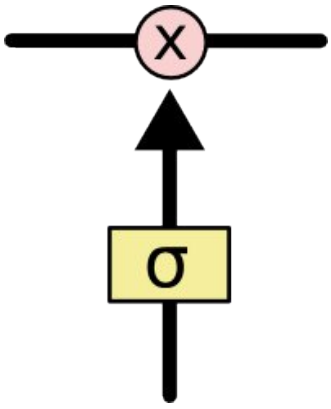
как сложение векторов, в то время, как желтые прямоугольники - это обученные слои нейронной сети. Сливающиеся линии обозначают конкатенацию, в то время как ветвящиеся линии обозначают, что их содержимое копируется, и копии отправляются в разные места.

4 Главная идея LSTM

Ключ к LSTM - клеточное состояние (cell state) - горизонтальная линия, проходящая сквозь верхнюю часть диаграммы. Клеточное состояние - это что- то типа ленты конвейера. Она движется прямо вдоль всей цепи только лишь с небольшими линейными взаимодействиями. Информация может просто течь по ней без изменений.



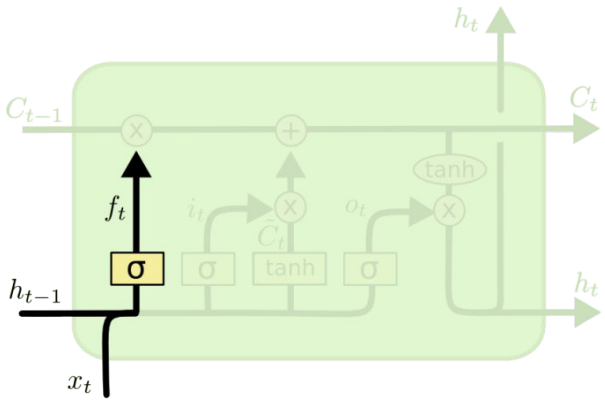
LSTM имеет способность удалять или добавлять информацию к клеточному состоянию, однако эта способность тщательно регулируется структурами, называемыми вентилями (**gates**). Вентили - это способ избирательно пропускать информацию. Они составлены из сигмоидного слоя НС и операции поточечного умножения (pointwise multiplication).



Сигмоидный слой подает на выход числа между нулем и единицей, описывая таким образом, насколько каждый компонент должен быть пропущен сквозь вентиль. Ноль - “ничего не пропускать”, один - “пропускать все”. LSTM имеет три таких вентиля, чтобы защищать и контролировать клеточное состояние.

Первым шагом в нашей LSTM будет решить какую информацию мы собираемся выбросить из клеточного состояния. Это решение принимается сигмоидным слоем, называемым “забывающим вентилем” (“**forget gate layer**”). Он получает на входе значения h_{t-1}и x_t и подает на выход число между 0 и 1. Единица означает “сохрани это полностью”, в тот время как ноль означает “избавься от этого полностью”.

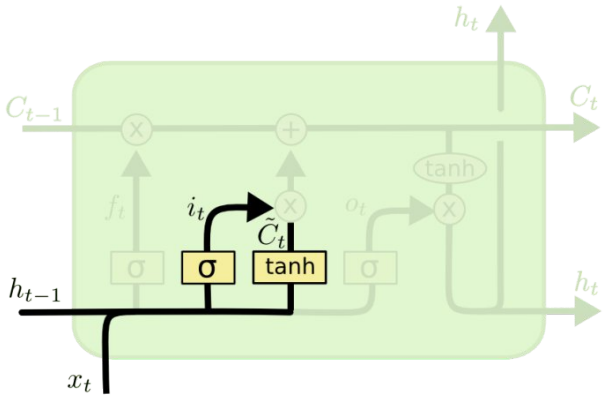
Давайте вернемся к нашему примеру языковой модели, пытающейся предсказать следующее слово, основываясь на всех предыдущих. В такой проблеме клеточное состояние может включать род подлежащего, что позволитиспользовать правильные формы местоимений. Когда мы видим новое подлежащее, мы забываем род предыдущего подлежащего.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

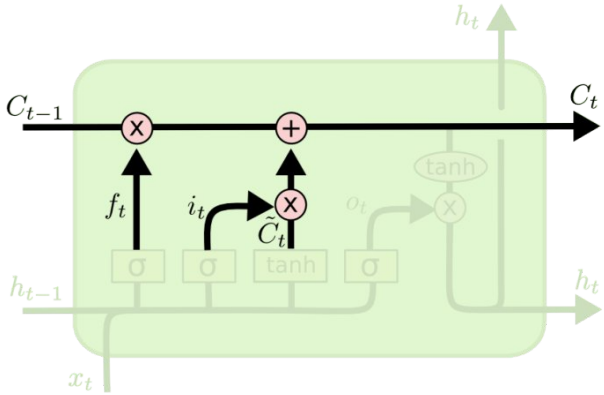
Следующим шагом будет решить, какую новую информацию мы собираемся сохранить в клеточном состоянии. Этот шаг состоит из двух частей. Во-первых, сигмоидный слой, называемый “входным вентилем” (“**input gate layer**”), решает, какие значения мы обновим. Далее, слой гирпеболического тангенса создает вектор кандидатов на новые значения σ_t, который может быть добавлен к состоянию. На следующем шаге мы соединим эти две части, чтобы создать обновление для состояния.

В примере с нашей языковой моделью мы бы хотели добавить род нового подлежащего к клеточному состоянию, чтобы заменить род старого, котороемы должны забыть.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Теперь пришла пора обновить старое клеточное состояние, σ_{t-1} новым клеточным состоянием σ_t. Все решения уже приняты на предыдущих шагах, осталось только сделать это. Мы умножаем старое состояние на f_t, забывая все, что мы ранее решили забыть. В случае с языковой моделью, это как раз то место, где мы теряем информацию о роде старого подлежащего и добавляем новую информацию, как решили на предыдущих шагах.

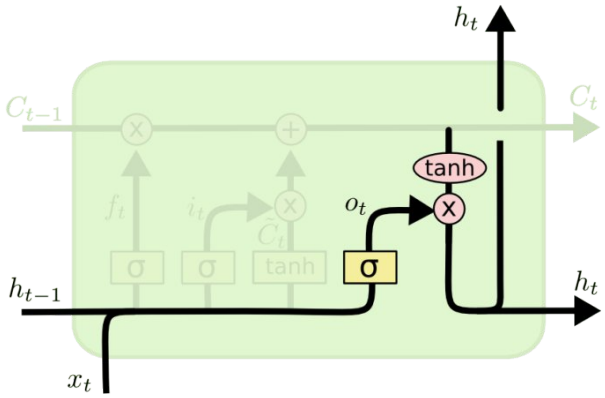


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Наконец, нам нужно решить, какой результат мы собираемся подать на выход. Этот результат будет основан на нашем клеточном состоянии, но будет его отфильтрованной версией. Сначала мы запускаем сигмоидный слой, который решает, какие части клеточного состояния мы собираемся отправить на выход. Затем мы пропускаем клеточное состояние сквозь гиперболический тангенс (**tanhtanh**) (чтобы уместить значения в промежуток от −1до 1) и умножаем его на выход сигмоидного вентиля, так что мы отправляем на выход

только те части, которые мы хотим.

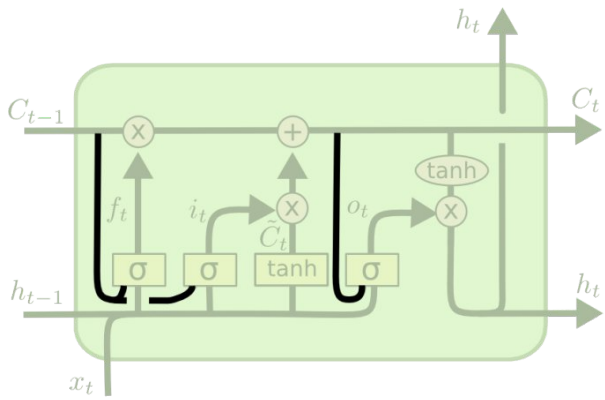
В примере с языковой моделью, если она только что видела подлежащее, она могла бы подать на выход информацию, относящуюся к глаголу (в случае, если следующее слово именно глагол). К примеру, она, возможно, подаст на выход число подлежащего (единственное или множественное). Таким образом, мы будем знать, какая форма глагола должна быть подставлена (если конечно дальше идет именно глагол).



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

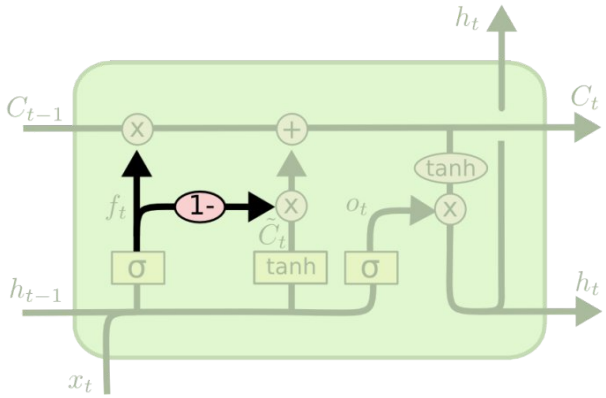
5 Разновидности LSTM сетей

В одном из популярных вариантов LSTM добавляются “глазковые соединения” (“peephole connections”). Это значит, что мы позволяем вентилям “подглядывать” за клеточным состоянием.



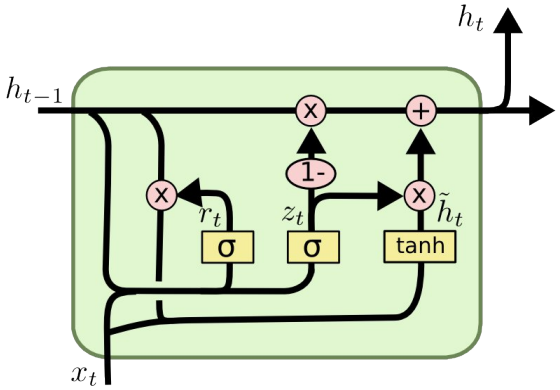
$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] + b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] + b_o\right)$$

Другая вариация - использование спаренных забывающих и входных вентилей. Вместо того, чтобы независимо решать, что забыть и куда мы должны добавить новую информацию, мы принимаем эти решения одновременно. Мы забываем что-то только в том случае, когда мы получаем что-то другое на это место. Мы получаем на вход новые значения только когда забываем что-то старое.



$$C_t = f_t * C_{t-1} + (\mathbf{1} - \boldsymbol{f_t}) * \tilde{C}_t$$

Несколько более существенно отличается от LSTM вентильная рекуррентная единица (Gated Recurrent Unit) или GRU. Она совмещает забывающие и входные вентили в один “обновляющий вентиль” (“**update gate**”). Она также сливает клеточное состояние со скрытым слоем и вносит некоторые другие изменения. Модель, получающаяся в результате, проще, чем обычная модель LSTM и она набирает популярность.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Это только некоторые из наиболее заметных вариантов LSTM. Есть множество других, например, глубинно-вентильные РНС (Depth Gated RNNs). Существует и совершенно другой подход к изучению долговременных зависимостей, например, часовые РНС (Clockwork RNNs)

6 Прогнозирование временных рядов

Задачи прогнозирования временных рядов - сложный тип проблемы прогнозирующего моделирования. В отличие от регрессионного предсказательного моделирования временные ряды также добавляют сложность зависимости последовательности от входных переменных.

Мощный тип нейронной сети, предназначенный для обработки

последовательностей называется рекуррентными нейронными сетями. Сеть с длинной короткой памятью или сеть LSTM - это тип рекуррентной нейронной сети, используемой в глубоком обучении, потому что можно успешно обучать очень большие архитектуры.

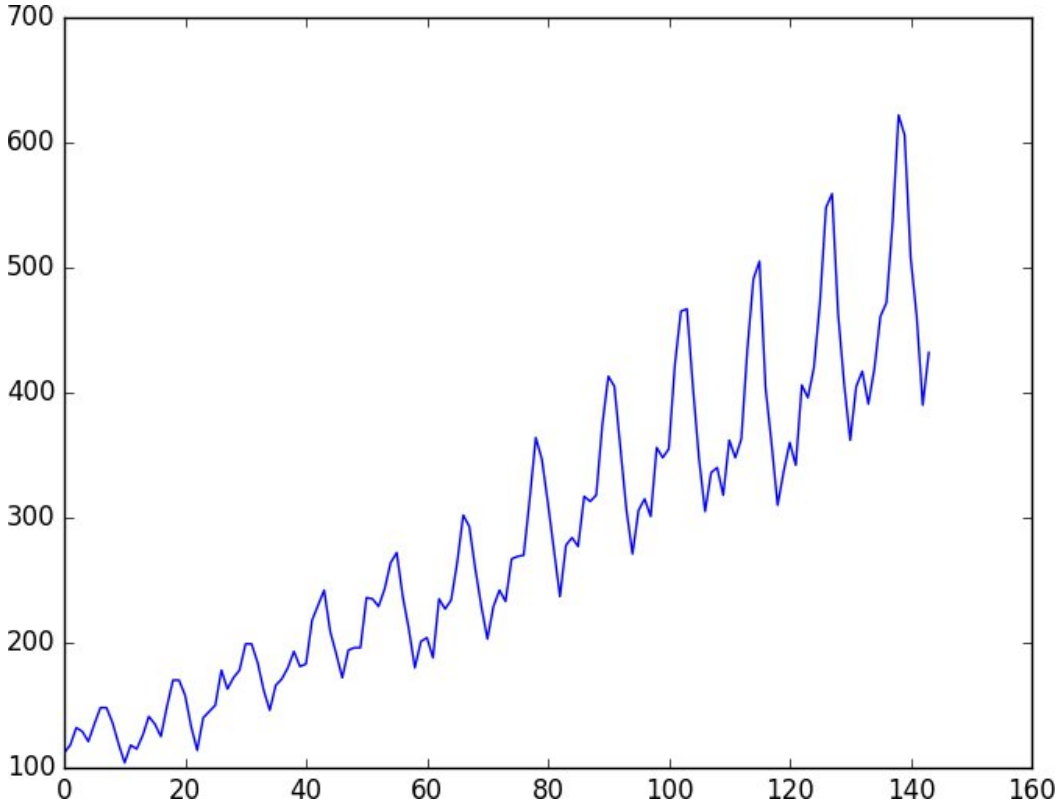
В этом разделе мы разработаем ряд LSTM для стандартной задачи прогнозирования временных рядов. Эти примеры помогут вам разработать свои собственные структурированные LSTM-сети для задач прогнозирования временных рядов.

Задача, которую мы рассмотрим это - проблема прогнозирования пассажирских авиаперевозок. Задача состоит в том зная год и месяц предсказать количество пассажиров международных авиакомпаний.

Набор данных доступен бесплатно можно скачать с адреса https://datamarket.com/data/set/22u3/international-airline-passengers-monthly-totals-in-thousands-jan-49-dec-60#!ds=22u3&display=line с именем файла « *international-airlines-passengers.csv* ». Данные варьируются от января 1949 года до декабря 1960 года или 12 лет с 144 наблюдениями. Мы можем загрузить этот набор данных с помощью библиотеки Pandas. Нам не интересна дата, учитывая, что каждое наблюдение разделяется одним и тем же интервалом в один месяц. Поэтому, когда мы загружаем набор данных, мы можем исключить первый столбец. После загрузки мы можем легко построить весь набор данных. Код для загрузки и построения набора данных приведен ниже.

```
import pandas
import matplotlib.pyplot as plt
dataset = pandas.read_csv('international-airline-passengers.csv', usecols=[1], engine='python', skipfooter=3)
plt.plot(dataset)plt.show()
```

С течением времени можно увидеть восходящий тренд в наборе данных и некоторую периодичность для набора данных, который, вероятно, соответствует периоду отпуска в северном полушарии.



Мы можем сформулировать эту задачу как задачу регрессии. То есть, учитывая количество пассажиров (в тысячах единиц) в этом месяце прогнозировать количество пассажиров в следующем месяце. Мы можем написать простую функцию, чтобы преобразовать наш единственный столбец данных в двухстолбцовый набор данных: первая колонка, содержащая количество пассажиров и второй столбец, который будет содержать количество пассажиров в следующем месяце.

Прежде чем мы начнем, давайте сначала импортируем все функции и классы, которые мы намерены использовать.

```
import numpy
import matplotlib.pyplot as pltfrom pandas import read_csv import math
from keras.models import Sequentialfrom keras.layers import Dense
```

from keras.layers import LSTM

from sklearn.preprocessing import MinMaxScaler from sklearn.metrics import mean_squared_error

Прежде чем мы что-либо сделать инициализируем генератор случайных чисел, чтобы гарантировать, что наши результаты будут воспроизводимыми.

numpy.random.seed(7)

Используем код из предыдущего раздела для загрузки набора данных в виде данных Pandas. Затем мы можем извлечь массив NumPy из фрейма данных и преобразовать целочисленные значения в значения с плавающей запятой,которые более подходят для работы с нейронной сетью.

dataframe = read_csv('international-airline-passengers.csv',usecols=[1], engine='python', skipfooter=3)

dataset = dataframe.values dataset = dataset.astype('float32')

LSTM чувствительны к шкале входных данных, особенно когда используются сигмоидные (по умолчанию) или функции активации **tanh**. Поэтому необходимо произвести масштабирование данных до диапазона от 0 до 1, также называемого нормализацией. Мы можем легко нормализовать набор данных, используя **класс** предварительной обработки **MinMaxScaler** из библиотеки **scikit-learn**.

scaler = MinMaxScaler(feature_range=(0, 1))dataset = scaler.fit_transform(dataset)

После того, как мы моделируем наши данные и оценим качество нашей модели на учебном наборе данных, нам нужно узнать насколько точен прогноз на данных которые сеть не видела. Для обычной задачи классификации или регрессии мы будем делать это с использованием перекрестной проверки. При использовании временных рядов важна последовательность значений. Простымметодом, который мы можем использовать, является разделение упорядоченного набора данных на учебный и тестовые наборы данных. Приведенный ниже код разделяет данные на учебные наборы данных с 67% наблюдений, которые мы можем использовать для обучения нашеймодели, оставляя 33% для тестирования модели.

*train_size = int(len(dataset) * 0.67) test_size = len(dataset) - train_size train, test =*

dataset[0:train_size,:], dataset[train_size:len(dataset),:]

Теперь определим функцию для создания нового набора данных, как описано выше. Функция принимает два аргумента: **набор данных**, который представляет собой массив **NumPy**, который мы хотим преобразовать в набор данных, и **look_back**, который представляет собой число предыдущих шагов времени для использования в качестве входных переменных для прогнозирования следующего периода времени - в этом случае по умолчанию -

1. Это значение по умолчанию создаст набор данных, где X - количество пассажиров в заданное время (t), а Y - количество пассажиров в следующий момент времени (t + 1).

def create_dataset(dataset, look_back=1):dataX, dataY = [], []

for i in range(len(dataset)-look_back-1): a = dataset[i:(i+look_back), 0] dataX.append(a)

dataY.append(dataset[i + look_back, 0]) return numpy.array(dataX), numpy.array(dataY)

Далее используем эту функцию для подготовки наборов данных для обучения и для тестирования нейронной сети и преобразуем данные в структуру, соответствующую входу нейронной сети.

look_back = 1

trainX, trainY = create_dataset(train, look_back) testX, testY = create_dataset(test, look_back)

trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1])) testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))

Теперь мы разработаем и настроим нашу сеть LSTM для решения этой задачи. В слой с 4 блоками LSTM или нейронами и выходной уровень, которыйна выходе дает одно значение. Для LSTM нейронов используется сигмоидальная функция активации по умолчанию и сеть обучается в течение 100 эпох.

```
model = Sequential()
model.add(LSTM(4, input_shape=(1, look_back))) model.add(Dense(1)) model.compile(loss='mean_squared_error', optimizer='adam') model.fit(trainX, trainY, epochs=100, batch_size=1, verbose=2)
```

После того, как модель прошла обучение мы можем оценить качество модели учебном и на тестовом наборах данных. Обратите внимание, что мы инвертируем предсказания перед вычислением ошибок, чтобы гарантировать, что результат выводится в тех же единицах, что и исходные данные (тысячи пассажиров в месяц).

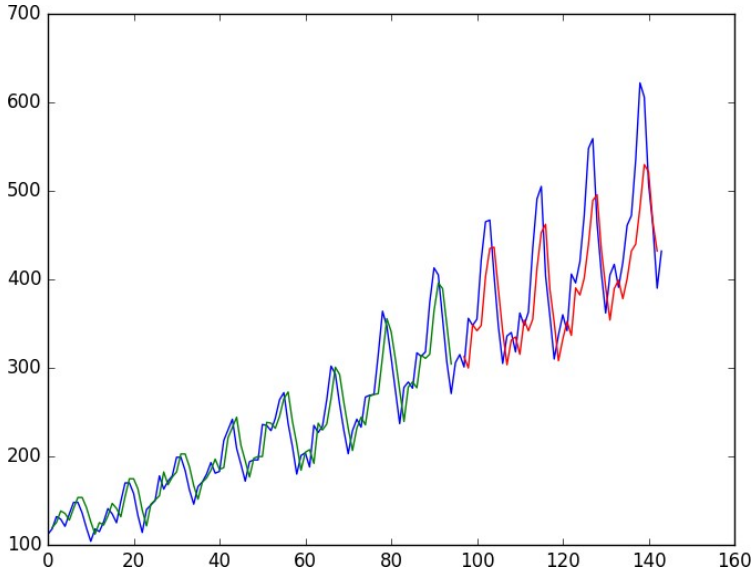
```
trainPredict = model.predict(trainX)testPredict = model.predict(testX)
trainPredict = scaler.inverse_transform(trainPredict) trainY = scaler.inverse_transform([trainY]) testPredict = scaler.inverse_transform(testPredict) testY = scaler.inverse_transform([testY])
trainScore = math.sqrt(mean_squared_error(trainY[0], trainPredict[:,0])) print('Train Score: %.2f RMSE' % (trainScore))
testScore = math.sqrt(mean_squared_error(testY[0], testPredict[:,0])) print('Test Score: %.2f RMSE' % (testScore))
```

Наконец, мы можем генерировать предсказания, используя учебные и тестовые данные, для того чтобы получить визуальное представление о качестве модели.

Из-за того, как был подготовлен набор данных, мы должны сдвинуть предсказания так, чтобы они выровнялись по оси **x** с исходным набором данных.

trainPredictPlot = numpy.empty_like(dataset)trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
*testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] = testPredict*
plt.plot(scaler.inverse_transform(dataset))
plt.plot(trainPredictPlot)plt.plot(testPredictPlot) plt.show()

Исходный набор данных отображен синим цветом, прогнозы для набора учебных данных зеленым цветом и прогнозы по тестовому набору данных красным цветом. Мы видим, что модель отлично справилась с прогнозом как учебном, так и на тестовом наборе данных.



В консоли программа выдала следующую информацию об ошибках: Train Score: 22.93 RMSE

Test Score: 47.53 RMSE