УНИВЕРСИТЕТ
искусственного
интеллекта

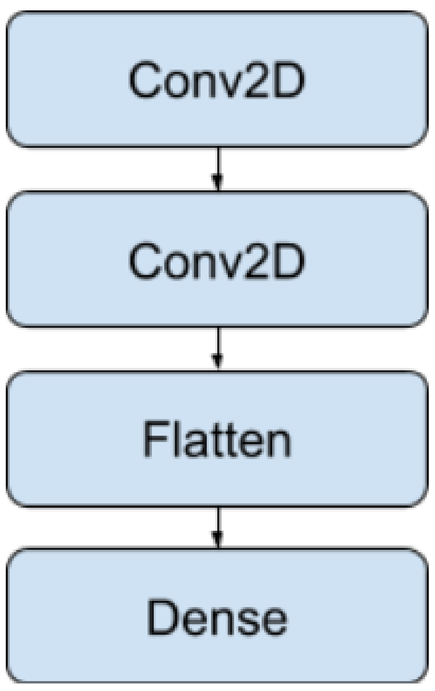
Рекуррентные сети и одномерная свертка

Функциональное программирование (API)

В Keras существует две модели программирования:

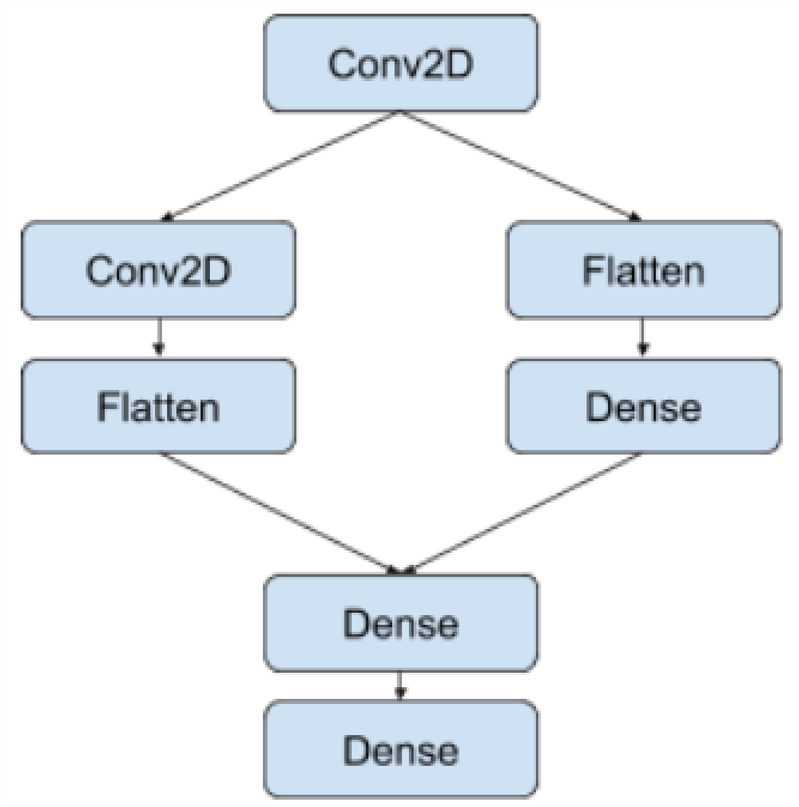
- Последовательное
- Функциональное

С последовательным API мы уже знакомы. В Keras — это модель Sequential, которая представляет собой последовательное создание слоев. Эта модель ограничена тем, что не может принимать на вход или выход несколько слоев или использовать одни и те же слои в разных частях архитектуры.



Функциональный API представляет более гибкую модель.

Функциональное программирование (API)



Модели определяются путем создания слоев и их непосредственного соединения друг с другом, а затем определения модели, которая задает слои, выступающие в качестве входных и выходных данных.

Весь функциональный API строится на следующей конструкции:

```
слой = Layer(аргументы слоя)(предыдущий слой)
```

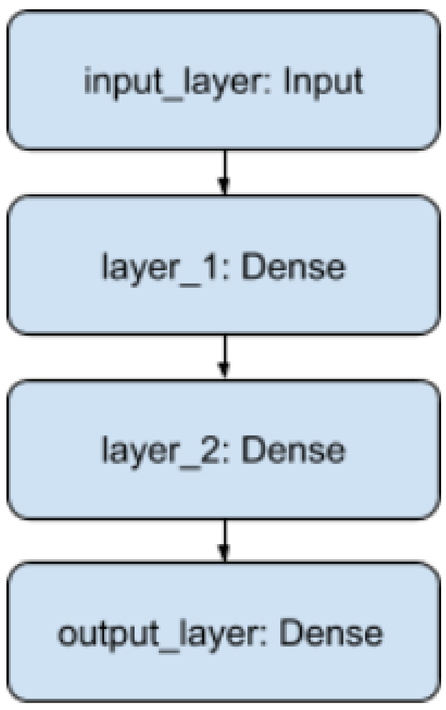
Рассмотрим на примере:

```
input_layer = Input(shape=(1000, )) # Входной слой
layer_1 = Dense(16, activation='relu')(input_layer) #
Полносвязный слой на 16 нейронов
layer_2 = Dense(16, activation='relu')(layer_1) #
Полносвязный слой на 16 нейронов
output_layer = Dense(1, activation='sigmoid')(layer_2) #
Выходной слой
```

В отличие от последовательной модели, вы должны создать отдельный входной слой, который определяет форму входных данных: input_layer. Далее создаём слой layer_1 и в скобках указываем, после какого слоя он идет (после input_layer). Затем создаем слой layer_2 и в скобках указываем, после какого слоя он идет (после layer_1), и по той же схеме добавляем последний слой output_layer.

Схема примера имеет следующий вид:

Функциональное программирование (API)



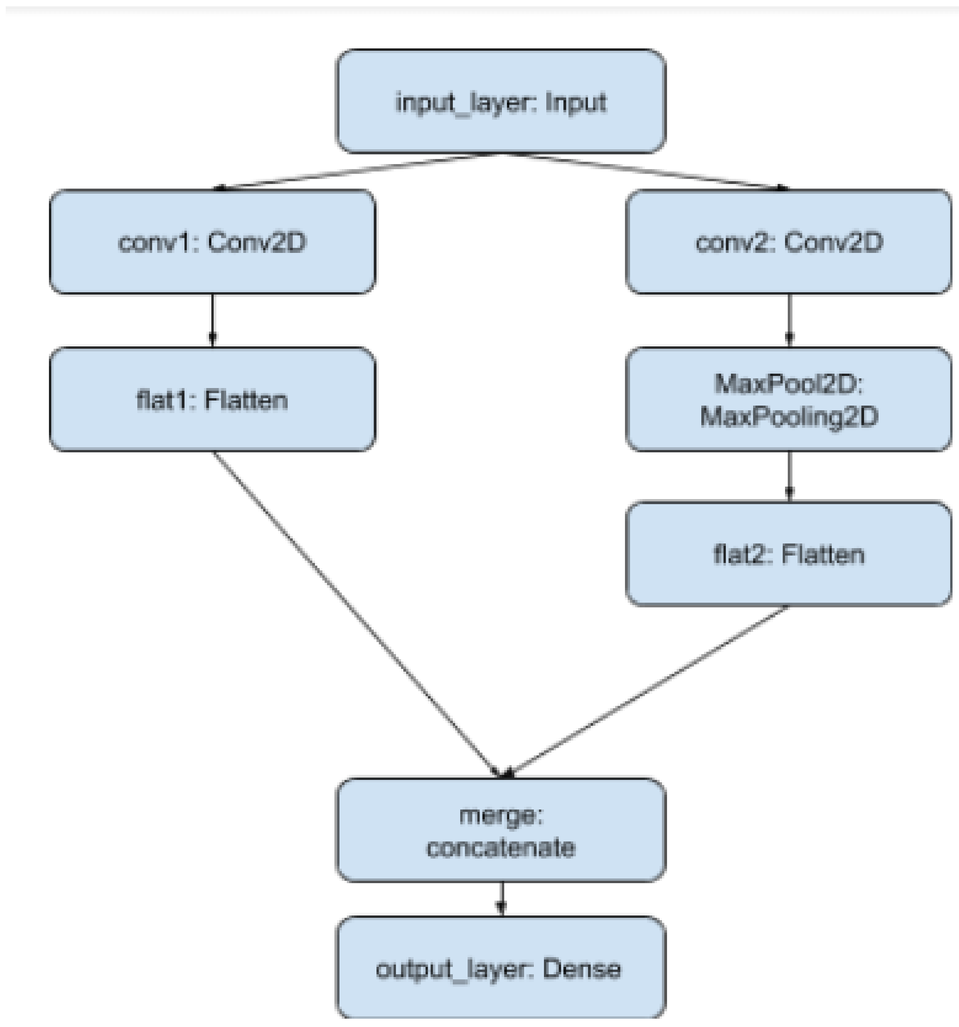
Keras имеет класс Model, который вы можете использовать для создания модели из созданных вами слоев. Требуется указать только входной и выходной слои:

```
from keras.models import Model
model = Model(inputs=input_layer, outputs=output_layer)
```

Модель с общими слоями

Несколько слоев могут совместно использовать вывод из одного слоя. Или два слоя могут быть соединены в один. Рассмотрим на примере:

```
# Входной слой
```



Функциональное программирование (API)

```
input_layer = Input(shape=(64,64,1))
# Левая ветвь
conv1 = Conv2D(64, kernel_size=4, activation='relu')
(visible)
flat1 = Flatten()(conv1)
# Правая ветвь
conv2 = Conv2D(32, kernel_size=8, activation='relu')
(visible)
MaxPool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
flat2 = Flatten()(MaxPool2)
# Слой конкатенации
merge = concatenate([flat1, flat2])
# Выходной слой
output_layer = Dense(1, activation='sigmoid')(merge)
```

Схема нашей сети:

Слои conv1 и conv2 — это слои, которые следуют из входного слоя отдельно друг от друга.

А слой merge (concatenate) соединяет слои flat1 и flat2.

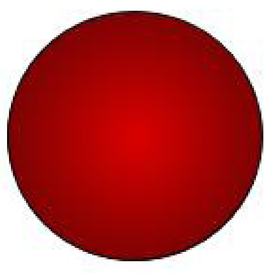
`keras.layers.concatenate(input, axis=-1)` — функциональный слой, который соединяет список тензоров на входе. В качестве входа берется список тензоров одинаковой формы, за исключением оси конкатенации, и возвращается один тензор, конкатенация всех входов.

Функциональное программирование хорошо применимо в генеративных сетях.

Рекуррентные сети

Рекуррентные сети (RNN) — это сети, у которых есть «память», учитывающая предшествующую информацию.

Рассмотрим пример: у нас есть видео какого-то объекта, например, летящего мяча. Видео разбито на кадры, и нам нужно понять, в какую сторону летит мяч. Если мы посмотрим на какой-нибудь кадр, то увидим следующую картину:



Мяч просто находится в какой-то точке. Если посмотрим другой кадр, то увидим, что мяч тоже будет находиться в какой-то точке.

Чтобы понять, в какую сторону двигается мач, нам нужно оценить несколько кадров, то есть посмотреть положение мяча, например, на трех последовательных кадрах:

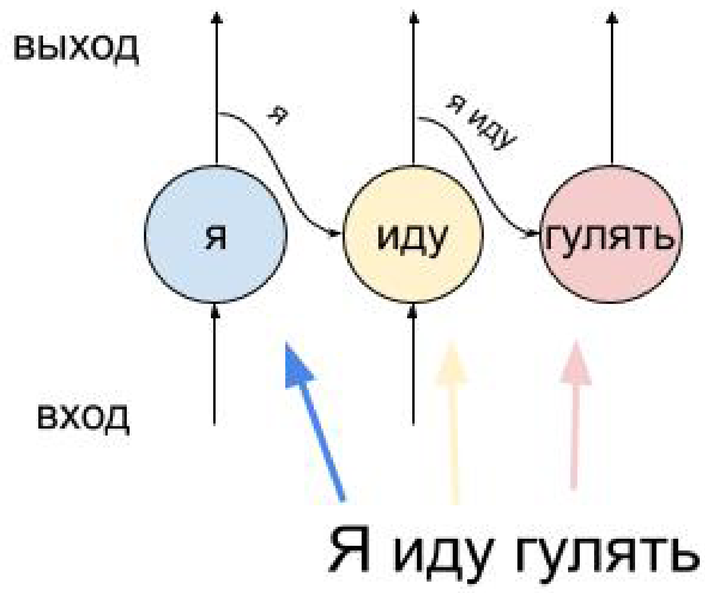
Глядя на предыдущие кадры, понимаем, что шар двигается влево.

Рекуррентные сети основаны на том, что делают вычисления на основе нынешних данных с учетом предыдущих. Отсюда следует определение.

Рекуррентные нейронные сети — вид нейронных сетей, где связи между элементами образуют направленную последовательность.

Теперь рассмотрим, как они устроены, на примере одного нейрона. Допустим, нам нужно предсказать слово в предложении «Я иду ...». Наш нейрон принимает на вход первое слово, обрабатывает его и подаёт на выход, запоминает, и при обработке следующего слова также подаст на вход выход обработки первого слова («я»). Обработав информацию с учетом первого и второго слова, нейрон что-то подаёт на выход и запоминает этот выход («я иду»), чтобы подать его на вход нашего искомого слова. Дальше алгоритм подбирает слово «гулять» по смыслу. И всё на основе последовательных связей.

Рекуррентные сети



Это всё выполняет один нейрон! Он создаёт как бы свою копию («виртуальный» нейрон) Благодаря этому мы можем получить выход не только из последнего «виртуального» нейрона, но и на любом шаге.

Главная проблема такого подхода — это **проблема исчезающего градиента**.

Проблема исчезающего градиента — это проблема потери информации. Чем длиннее последовательность, тем больше вероятность того, что

информация о первых значениях будет теряться.

Для предсказания слов в коротких последовательностях, например, «я иду ...», «поэт Александр Сергеевич ...» и т. д. такой проблемы не возникает. Но если мы будем иметь дело с текстом:

*«В детстве я несколько лет провел во **Франции**.*

...

<Несколько предложений или абзацев>

...

Я хорошо говорю по …»,

то мысль о Франции потеряется, и подбор правильного слова по смыслу «*Я хорошо говорю по **французски***» маловероятен.

Решает эту проблему рекуррентный слой LSTM.

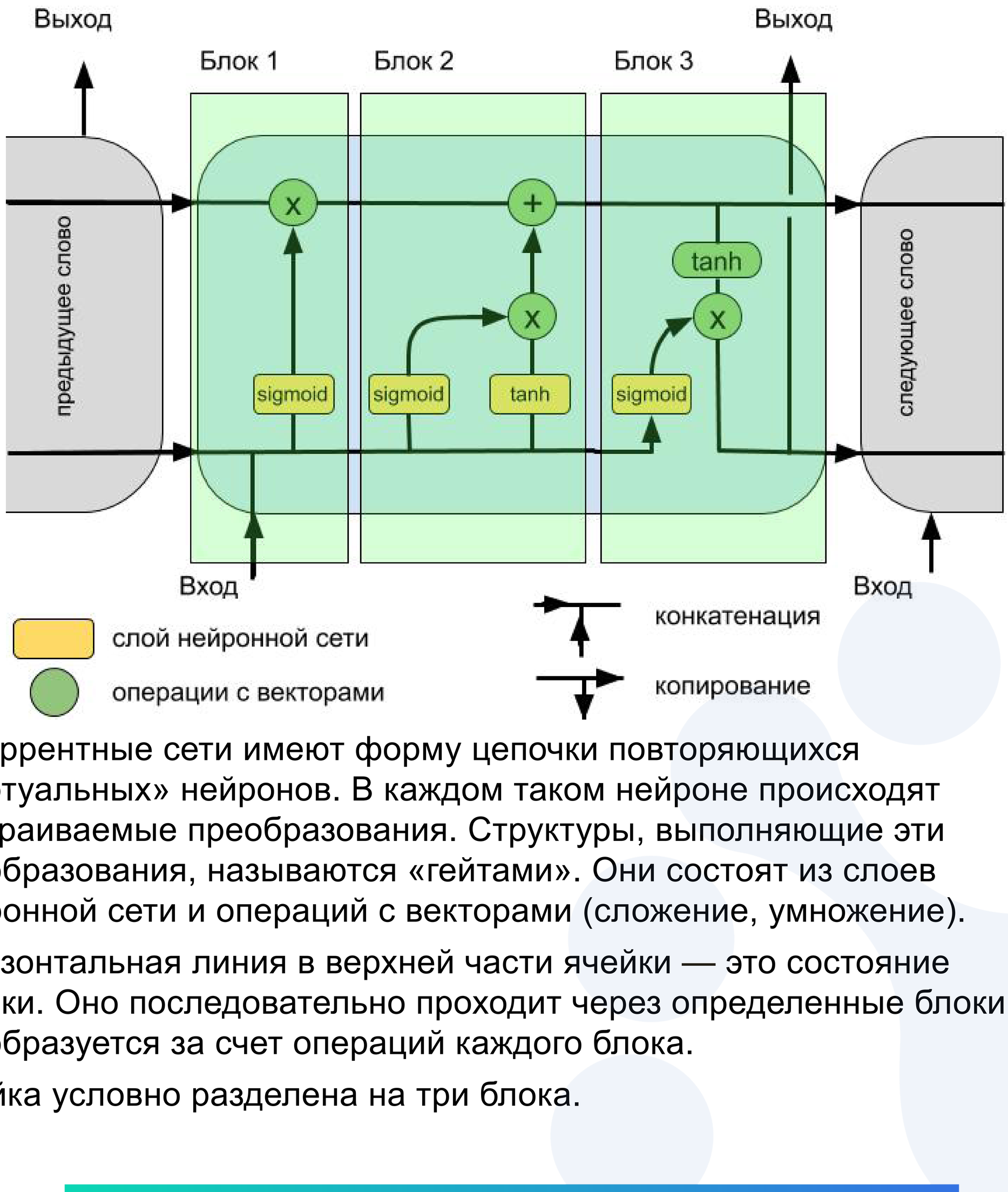
Рекуррентные сети

Сети LSTM

LSTM (*long short-term memory)*, дословно «*долгая краткосрочная память*» — тип рекуррентной нейронной сети, способной обучаться долгосрочным зависимостям.

LSTM специально разработаны для устранения проблемы исчезающего градиента. Их специализация — запоминание информации в течение длительных периодов времени.

Рассмотрим принцип работы LSTM сетей.



Рекуррентные сети имеют форму цепочки повторяющихся «виртуальных» нейронов. В каждом таком нейроне происходят настраиваемые преобразования. Структуры, выполняющие эти преобразования, называются «гейтами». Они состоят из слоев нейронной сети и операций с векторами (сложение, умножение).

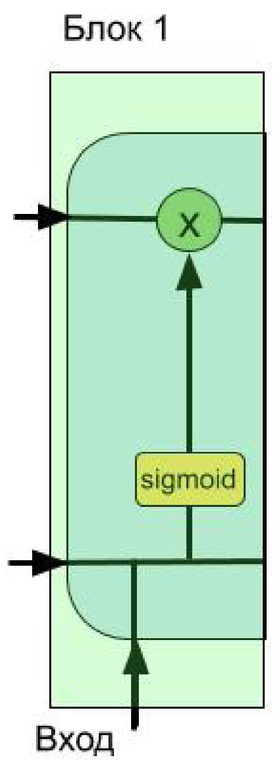
Горизонтальная линия в верхней части ячейки — это состояние ячейки. Оно последовательно проходит через определенные блоки и преобразуется за счет операций каждого блока.

Ячейка условно разделена на три блока.

Рекуррентные сети

Блок 1. Слой потери.

Информация, пришедшая из предыдущего нейрона (предыдущее слово), в совокупности с входными данными (текущее слово) проходит через сигмоиду и умножается на текущее значение состояния ячейки (оно пришло из предыдущей ячейки). Этот блок определяет, сколько информации из предыдущих ячеек нужно забыть, а сколько пропустить дальше.

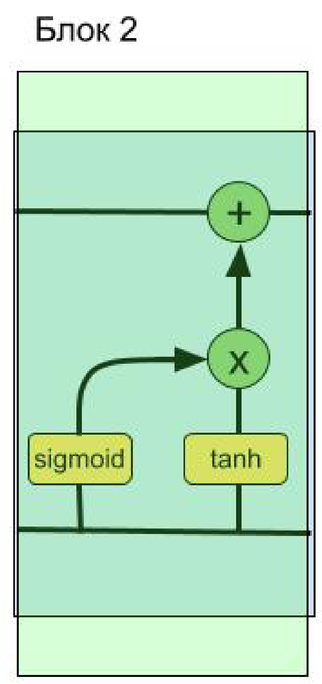


На выходе слоя sigmoid получаются значения от 0 до 1.

0 — это не пропускать ничего, 1 — это пропустить всё. Пропускается некоторая часть информации (в зависимости от значения).

Блок 2. Слой сохранения.

Этот блок отвечает за информацию, которая пойдет в следующую ячейку.



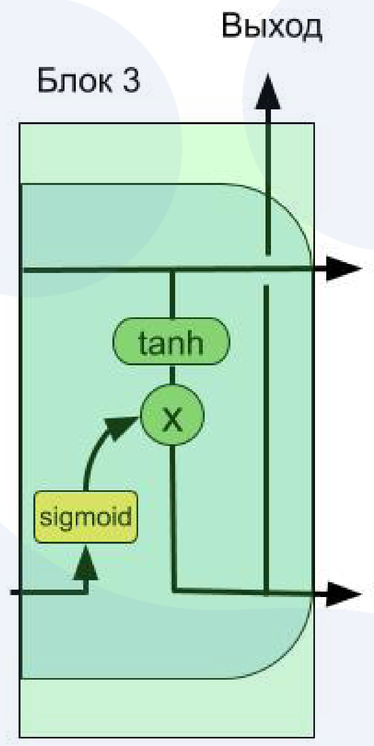
В слое сохранения происходит фиксация обновленных данных.

В **sigmoid** также определяется, сколько информации нужно забыть, а сколько пропустить. Слой **tanh** создает вектор новых значений (перемножаем на значение sigmoid (0,1)), которые добавляются в состояние ячейки.

Блок 3. Слой обновления.

Этот блок отвечает за то, что пойдёт в следующую ячейку.

Пропускаем состояние ячейки через **tanh** (чтобы разместить все значения в интервале [-1, 1]) и умножаем на выходной сигнал sigmoid.



Теперь поговорим о **Выходе** ячейки. Так как каждая такая ячейка — это «копия» нашего нейрона, то на выходе может быть либо одно значение (результирующая всех таких виртуальных нейронов), либо список выходов каждой ячейки (на рисунке это вертикальная стрелка в правом верхнем углу «Выход») .

Для применения LSTM на практике не обязательно знать принцип его работы.

Рекуррентные сети

LSTM слой в Keras

В Keras LSTM слой добавляется следующим образом:

```
# Вариант 1
model = Sequential()
model.add(Embedding(50, 10, input_length=1000))
model.add(LSTM(32))
```

Слой Embedding имеет размерность 1000x10, а слой LSTM имеет размерность 32. В этом случае слой LSTM на выходе каждого нейрона имеет только одно значение.

Если при создании слоя LSTM мы укажем значение параметра *return_sequences=True*, то на выходе каждого нейрона будет не одно значение, а несколько (равное размеру предыдущего слоя).

```
# Вариант 2
model = Sequential()
model.add(Embedding(50, 10, input_length=1000))
model.add(LSTM(32, return_sequences=True))
```

В данном случае размерность после слоя LSTM будет 1000x32.

Двунаправленные рекуррентные сети Bidirectional

Для предсказания слов в текстах довольно удобно применять сети, которые знают, что было ДО текущего слова. Но еще лучше, если мы будем знать и то, какие слова стоят ПОСЛЕ предсказываемого. Это можно сделать при помощи двунаправленной RNN. Для этого в Keras существует обертка *Bidirectional().* Она обрабатывает последовательность не только с начала до конца, но и наоборот.

```
model = Sequential()
# Двунаправленная RNN
model.add(Bidirectional(LSTM(64, return_sequences=True),
input_shape=(5, 10)))
model.add(Bidirectional(LSTM(32)))
model.add(Dense(10))
```

Это та же LSTM, только работающая в две стороны.

Рекуррентные сети

Сети GRU

GRU можно рассматривать как более простую версию сетей LSTM. Он включает в себя много схожих понятий, но имеет меньше параметров, и поэтому при одном и том же размере скрытого слоя GRU обучается быстрее.

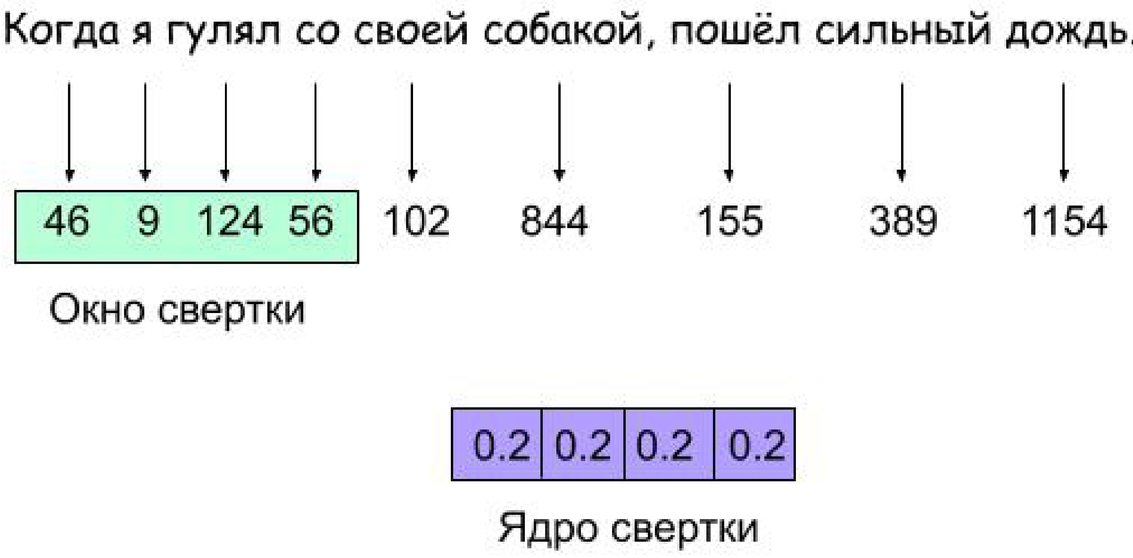
```
# Одномерная свертка
model = Sequential()
model.add(Embedding(50, 10, input_length=1000))
model.add(GRU(16, return_sequences=True))
```

Так же, как и LSTM, имеет параметр return_sequences.

Управляемый рекуррентный нейрон и сети долгой краткосрочной памяти имеют сопоставимую точность, а в некоторых случаях GRU даже точнее (но такое бывает довольно редко). Поэтому, прежде чем использовать LSTM, можно попробовать сделать нейронку на GRU, чтобы сэкономить время.

Одномерная свертка Conv1D

Рассмотрим пример одномерной свертки на сети для обработки текстов.



Создавая слой, мы указываем следующие параметры: окно свертки (фильтры) и ядро свертки. Чтобы посчитать все значения свертки, нужно каждый элемент окна умножить на каждый элемент ядра и сложить их. После этого смещаемся на один элемент вправо и повторяем процедуру, и так до тех пор, пока не дойдем до последнего элемента последовательности. Рассмотрим пример на картинке: первое значение свертки будет считаться как N = 46*0.2 + 9*0.2 + 124*0.2 + 56*0.2 = 47.

И далее так считаем все значения нашей последовательности.

В Keras этот слой применяется следующим образом:

```
# Одномерная свертка
model = Sequential()
model.add(Embedding(50, 10, input_length=1000))
model.add(Conv1D(20, 5, activation='relu'))
```

При создании слоя Conv1D указывается количество фильтров (filters=20) и размер ядра (kernel_size=5).

Преимущества одномерной сверточной нейронной сети:

- время обучения значительно ниже, чем у рекуррентных нейронных сетей

Недостатки одномерной сверточной нейронной сети:

- нет возможности «запомнить» нужные данные на длительный срок
- недостаток можно устранить с помощью механизма «внимания»