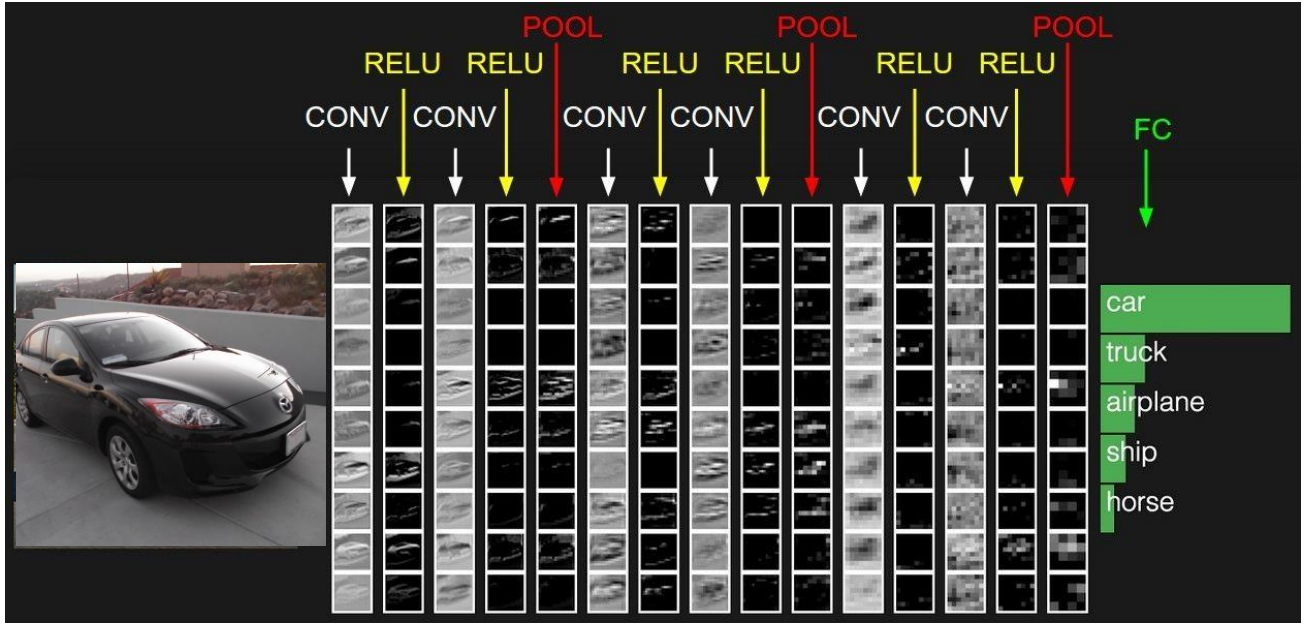
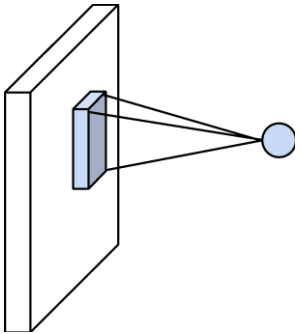
Лекция 10: Трансформеры

Напомним: Convolutional Neural Networks

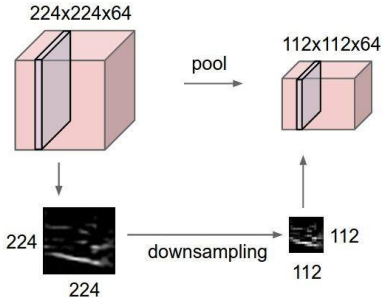


Components of CNNs

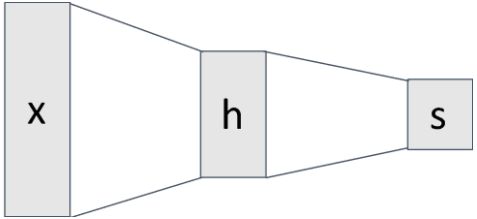
Convolution Layers



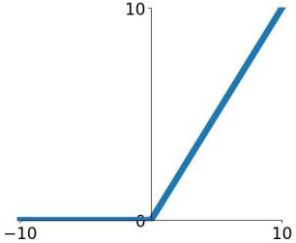
Pooling Layers



Fully-Connected Layers



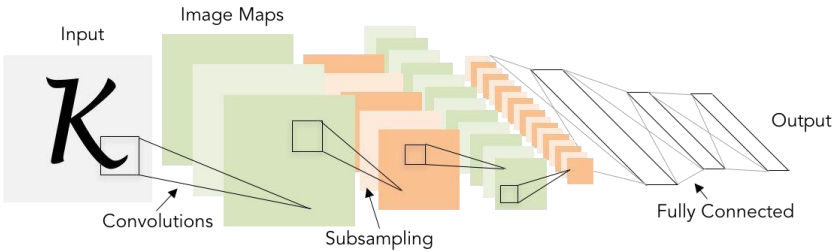
Activation Function

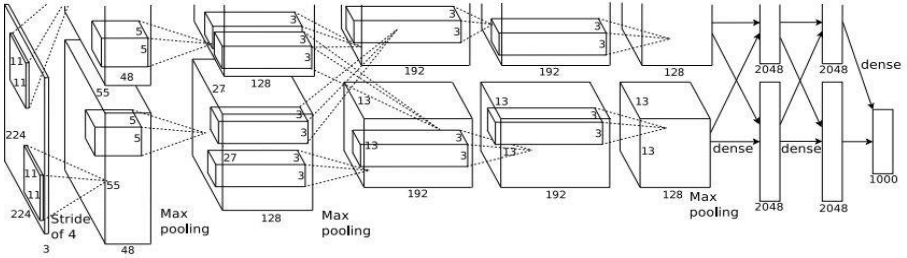


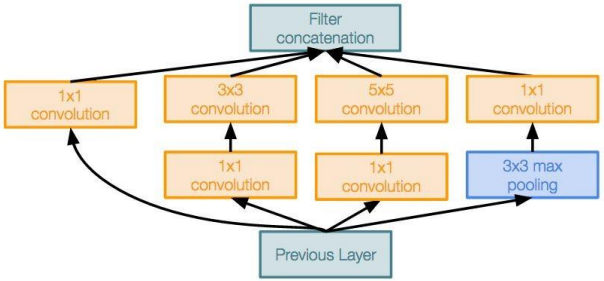
Normalization

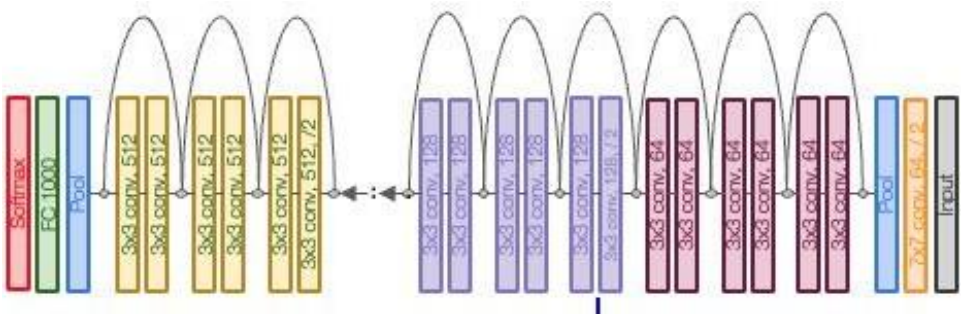
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Архитектуры CHC









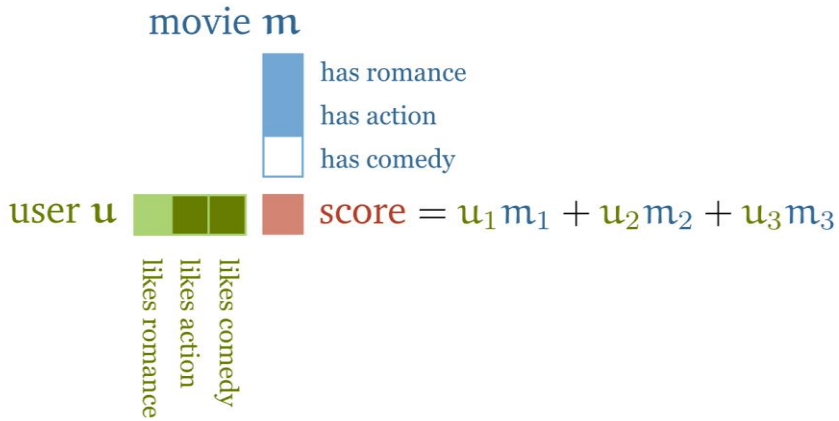
Трансформеры определение

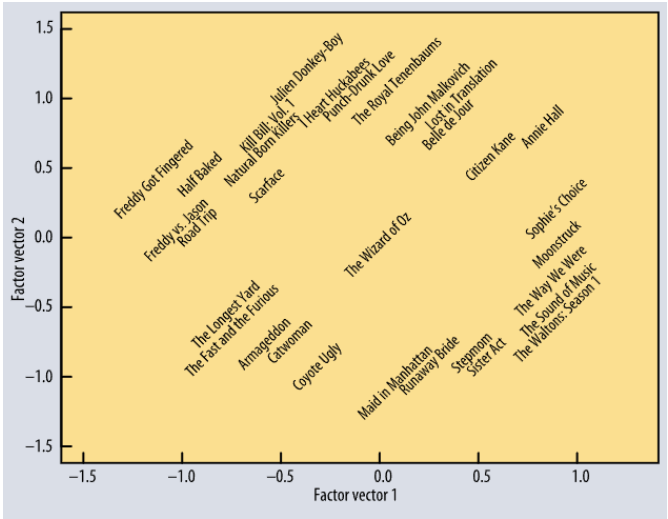
Трансформер *— нейросетевая архитектура, для обработки связанных наборов элементов, таких, как токены в последовательности пикселей изображения или слов в предложении, где эти элементы взаимодействуют только посредством* ***механизма внутреннего внимания****.*

Ashish Vaswani ... Illia Polosukhin, Attention Is All You Need, NIPS 2017
https://lilianweng.github.io/posts/2018-06-24-attention/
https://peterbloem.nl/blog/transformers
https://habr.com/ru/company/wunderfund/blog/592231/

Как работает внутреннее внимание/ self attention?

На примере рекомендательной системы фильмов:





Как работает внутреннее внимание/ self attention?

Найдем выход **y** взвесив по всем входам **x**:

$$\mathbf{y}_i = \sum_j w_{ij} \mathbf{x}_j \ .$$

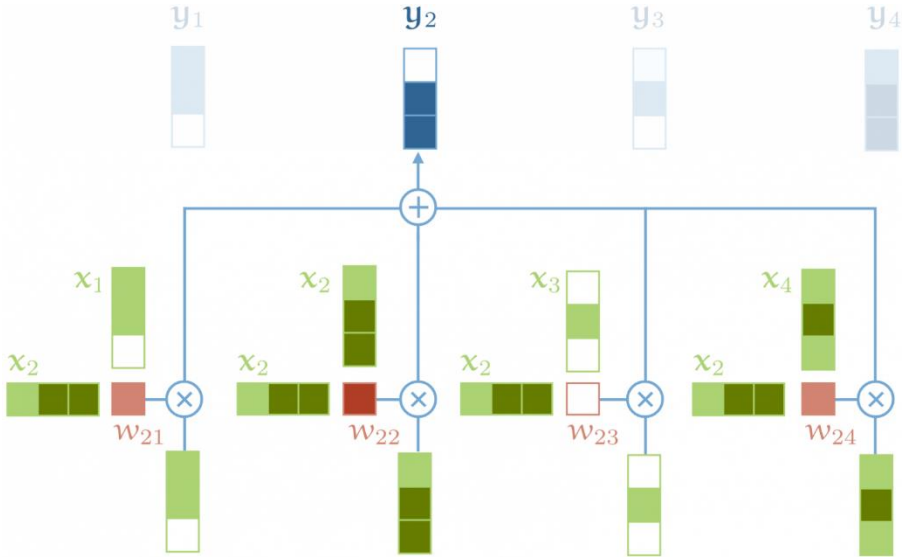
Простейший вариант весов - скалярное произведение:

$$w'_{ij} = \mathbf{x}_i{}^{\mathsf{T}} \mathbf{x}_j \ .$$

Но хорошо бы его отнормировать при помощи Softmax:

$$w_{ij} = \frac{\exp w'_{ij}}{\sum_j \exp w'_{ij}} \ .$$

Иллюстрация, для простоты без softmax):



Как внимание выглядит в torch?

x – вход, **y** – выход

```
import torch
import torch.nn.functional as F

# представим, что имеется тензор x размера (b, t, k)
x = ...

raw_weights = torch.bmm(x, x.transpose(1, 2))
# - torch.bmm - это команда пакетного умножения матриц. Она
#   выполняет операции умножения над пакетами
#   матриц.
```

```
weights = F.softmax(raw_weights, dim=2)
```

```
y = torch.bmm(weights, x)
```

Терминология и параметризация

Вход **x** может быть использован как

- Запрос, **query**, в сравнении с другими векторами входа для получения его выходного вектора **y**
- Ключ, **key**, в сравнении с другими векторами входа для формирования весов
- Значение, **value**, для формирования взвешенной суммы для каждого из выходных векторов

$$\mathbf{q}_i = \mathbf{W}_q \mathbf{x}_i \qquad \mathbf{k}_i = \mathbf{W}_k \mathbf{x}_i \qquad \mathbf{v}_i = \mathbf{W}_v \mathbf{x}_i$$

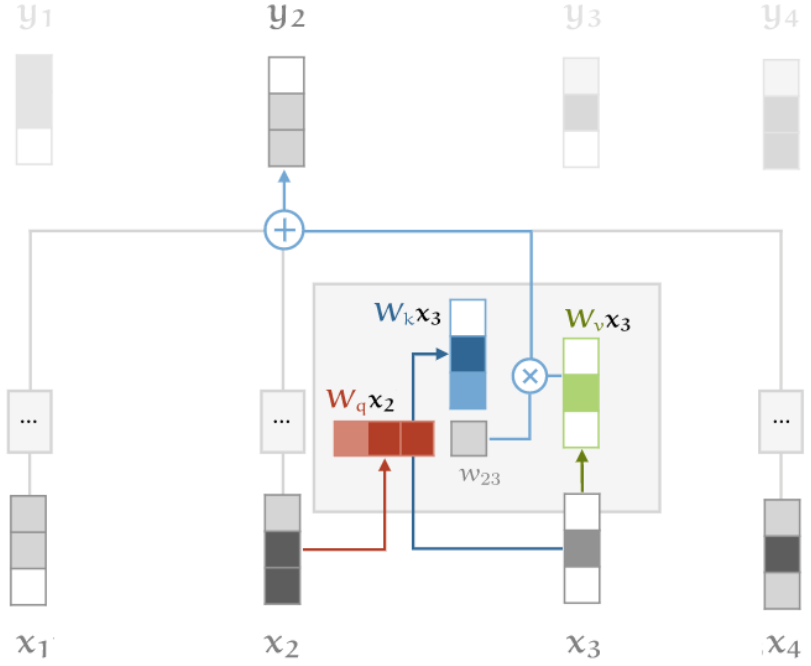
$$w'_{ij} = \mathbf{q}_i{}^T \mathbf{k}_j$$

$$w_{ij} = \text{softmax}(w'_{ij})$$

$$\mathbf{y}_i = \sum_j w_{ij} \mathbf{v}_j \,.$$

$$w'_{ij} = \frac{\mathbf{q}_i{}^T \mathbf{k}_j}{\sqrt{k}}$$

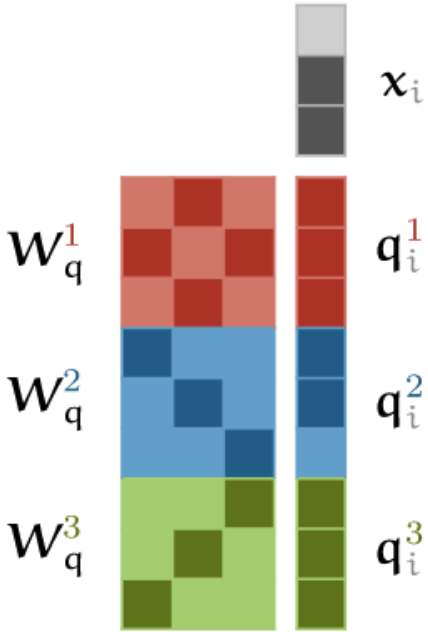
+ масштабирование для ограничения роста значения



Многоглавое внимание / multi-head attention

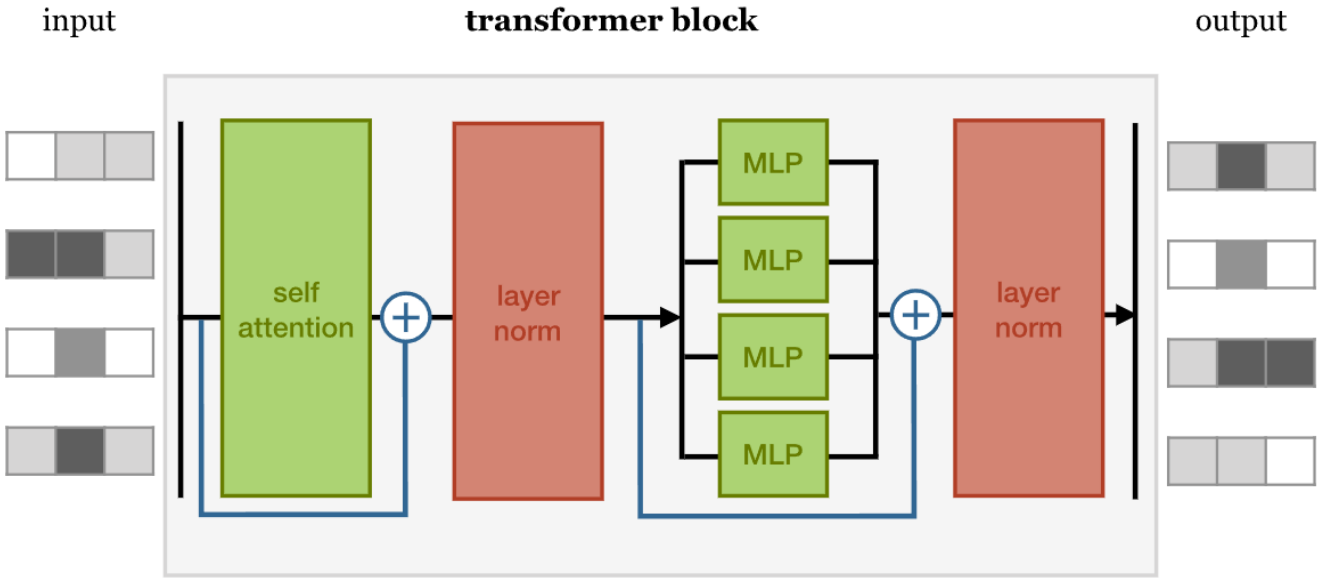
Внимание *эквивариативно* – перестановка входов дает перестановку выходов.
Как различить – Аргентина победила Ямайку, 5:0
или
Ямайка победила Аргентину, 5:0?

Решение – несколько блоков внимания, проще всего их понимать как несколько параллельных блоков.



Блок трансформера

Трансформер *— нейросетевая архитектура, для обработки связанных наборов элементов, таких, как токены в последовательности пикселей изображения или слов в предложении, где эти элементы взаимодействуют только посредством* ***механизма внутреннего внимания****.*



Блок трансформера - pytorch

```
class TransformerBlock(nn.Module):
  def __init__(self, k, heads):
    super().__init__()

    self.attention = SelfAttention(k, heads=heads)

    self.norm1 = nn.LayerNorm(k)
    self.norm2 = nn.LayerNorm(k)

    self.ff = nn.Sequential(
      nn.Linear(k, 4 * k),
      nn.ReLU(),
      nn.Linear(4 * k, k))

  def forward(self, x):
    attended = self.attention(x)
    x = self.norm1(attended + x)

    fedforward = self.ff(x)
    return self.norm2(fedforward + x)
```

Нормализация – LayerNorm

LAYERNORM

CLASS torch.nn.LayerNorm(*normalized_shape, eps=1e-05, elementwise_affine=True, device=None, dtype=None*) [SOURCE]

Applies Layer Normalization over a mini-batch of inputs as described in the paper Layer Normalization

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated over the last *D* dimensions, where *D* is the dimension of `normalized_shape`. For example, if `normalized_shape` is `(3, 5)` (a 2-dimensional shape), the mean and standard-deviation are computed over the last 2 dimensions of the input (i.e. `input.mean((-2, -1))`). γ and β are learnable affine transform parameters of `normalized_shape` if `elementwise_affine` is `True`. The standard-deviation is calculated via the biased estimator, equivalent to *torch.var(input, unbiased=False)*.

• NOTE

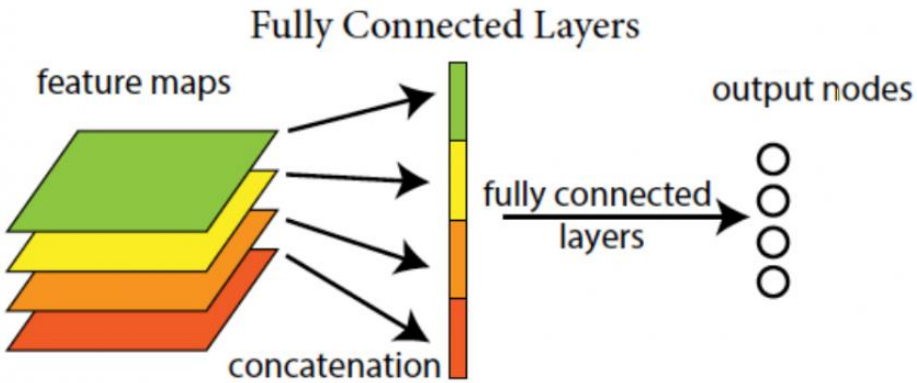
Unlike Batch Normalization and Instance Normalization, which applies scalar scale and bias for each entire channel/plane with the `affine` option, Layer Normalization applies per-element scale and bias with `elementwise_affine`.

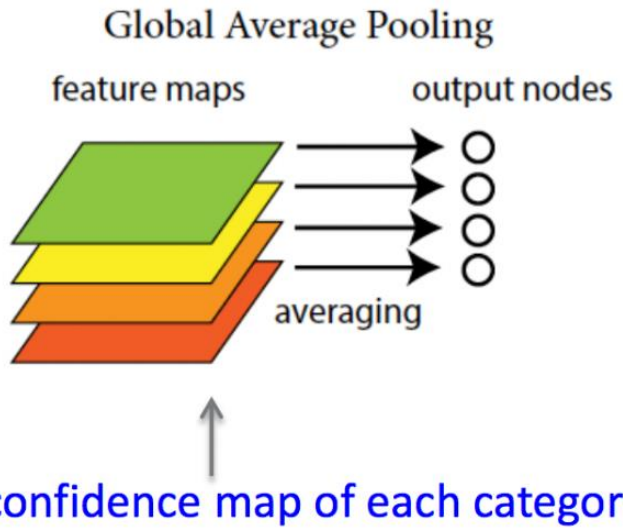
Замена FC – Global Average Pooling (GAP)

- GAP заменяет полносвязные слои, приэтом сеть становится полносвязной.
- GAP необучаемый!
- Активации сверточных слоев становятся картами конфидентности.

CNN

NIN

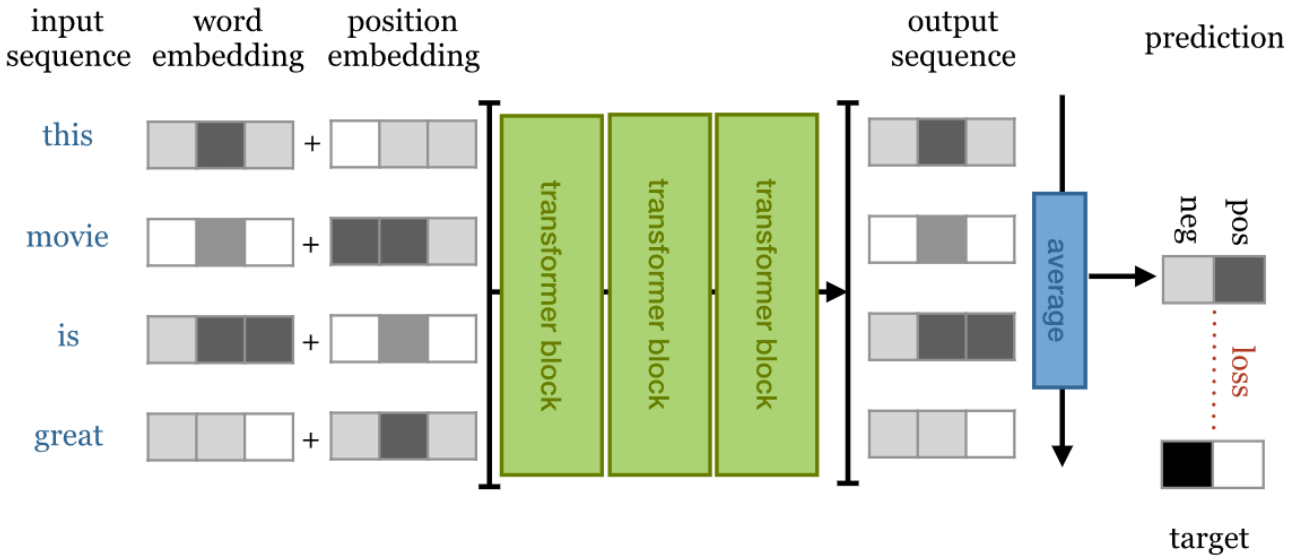




Трансформер для классификации

- Выход – применяем GAP к выходу блока трансформера.
- Вход – добавляем позицию, т.к. сеть инвариантна к перестановкам.

Используем позиционный эмбединг или позиционная кодировка.



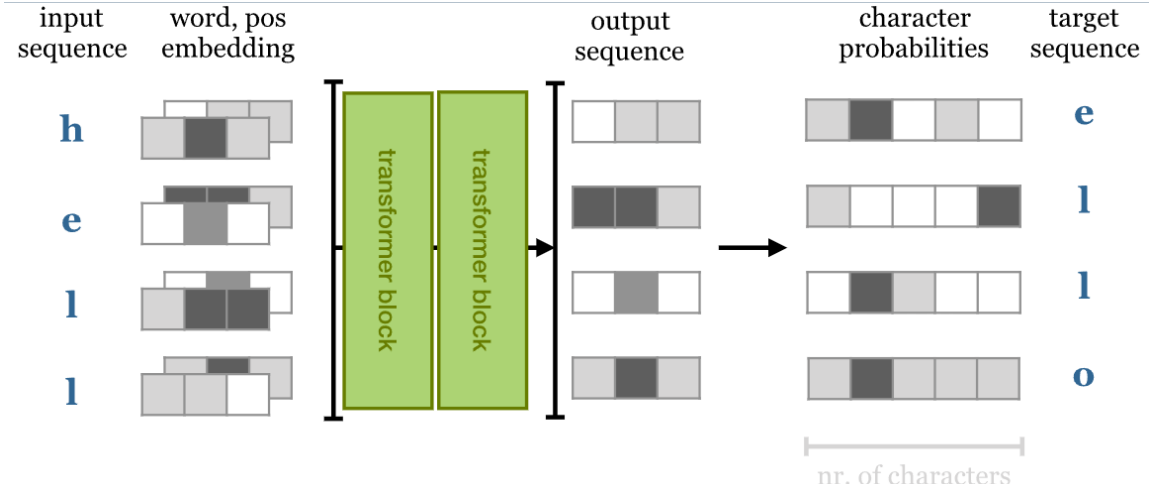
Трансформер для классификации - pytorch

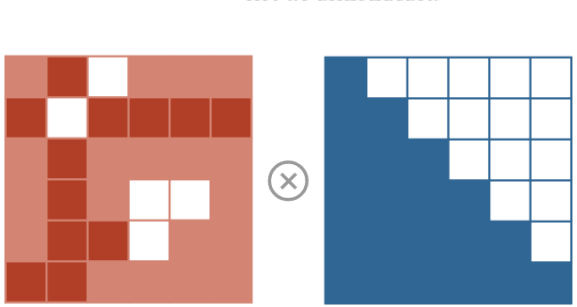
```
class Transformer(nn.Module):
    def __init__(self, k, heads, depth, seq_length, num_tokens, num_classes):
        super().__init__()

        self.num_tokens = num_tokens
        self.token_emb = nn.Embedding(num_tokens, k)
        self.pos_emb = nn.Embedding(seq_length, k)

        # Последовательность блоков трансформера, на которую
        # возлагается обязанность решения сложных задач
        tblocks = []
        for i in range(depth):
            tblocks.append(TransformerBlock(k=k, heads=heads))
        self.tblocks = nn.Sequential(*tblocks)

        # Настраиваем соответствие итоговой выходной последовательности ненормализованным
        self.toprobs = nn.Linear(k, num_classes)
```

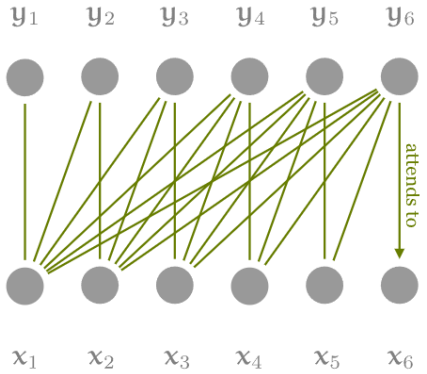
Трансформер для классификации - pytorch

```
def forward(self, x):
    """
    :param x: A (b, t) тензор целочисленных значений, представляющий
              слова (в некоем заранее заданном словаре).
    :return: A (b, c) тензор логарифмических вероятностей по
             классам (где c - это количество классов).
    """
    # генерируем эмбеддинги токенов
    tokens = self.token_emb(x)
    b, t, k = tokens.size()

    # генерируем позиционные эмбеддинги
    positions = torch.arange(t)
    positions = self.pos_emb(positions)[None, :, :].expand(b, t, k)

    x = tokens + positions
    x = self.tblocks(x)

    # Выполняем операцию усредняющего пуллинга по t измерениям и проецируем
    # на вероятности, соответствующие классам
    x = self.toprobs(x.mean(dim=1))
    return F.log_softmax(x, dim=1)
```

Трансформер для генерации текста



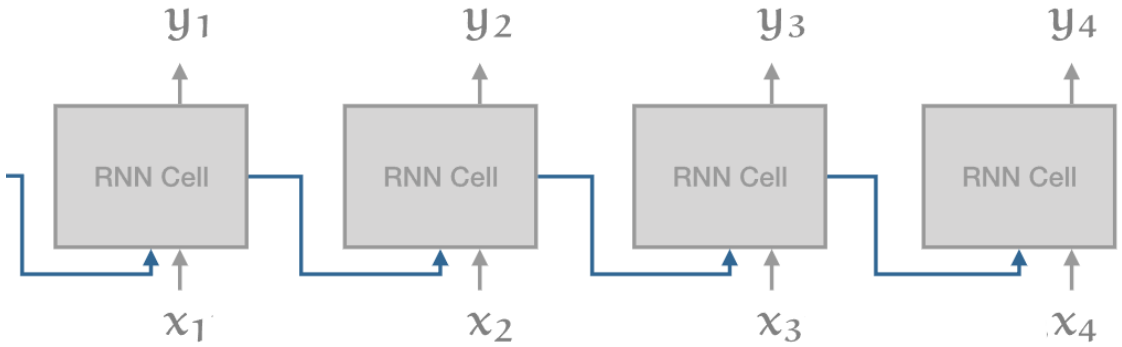




Маска не дает вниманию смотреть в будущее!

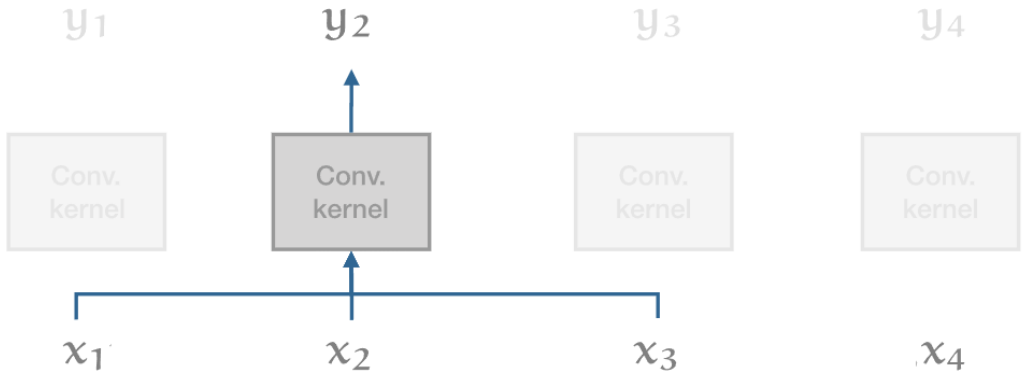
Между CONV и RNN

Рекуррентность это слабость RNN – для того, чтобы посчитать x(n), нужно посчитать x(n-1)



Сверточная сеть позволяет считать все параллельно, но с малым полем зрения.

Трансформеры – решают обе проблемы!



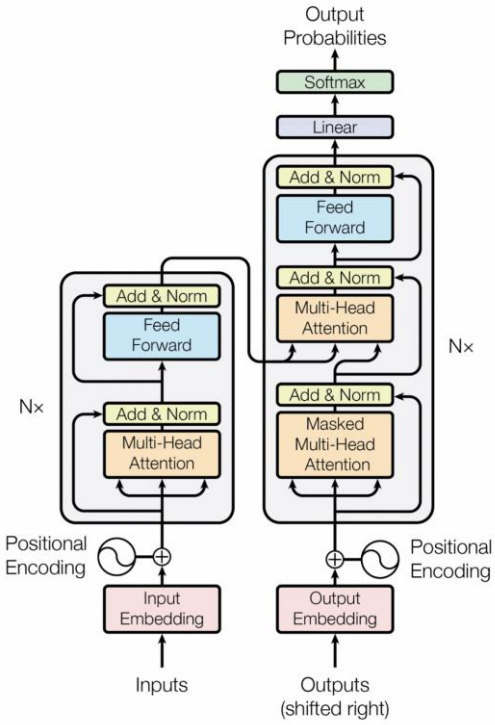
Применение: GPT

Hugging Face

Для работы с трансформерами используем Hugging Face! https://huggingface.co/

```
# Сначала установим библиотеку transformers
!pip install transformers

from transformers import GPT2LMHeadModel, GPT2Tokenizer
import torch
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Для наглядности будем работать с русскоязычной GPT от Сбера.
# Ниже команды для загрузки и инициализации модели и токенизатора.
model_name_or_path = "sberbank-ai/rugpt3large_based_on_gpt2"
tokenizer = GPT2Tokenizer.from_pretrained(model_name_or_path)
model = GPT2LMHeadModel.from_pretrained(model_name_or_path).to(DEVICE)
```

Пример кода для ruGPT3



Исходный трансформер

Языковое моделирование

Hugging Face

```
# prompt engineering for QA
text = "Вопрос: 'Сколько будет 2+2?'\nОтвет:"
input_ids = tokenizer.encode(text, return_tensors="pt").to(DEVICE)
out = model.generate(input_ids, do_sample=False)

generated_text = list(map(tokenizer.decode, out))[0]
print(generated_text)

#>>> Вопрос: 'Сколько будет 2+2?'
#>>> Ответ:  '2+2=4'
```

Пример: 2+2=4

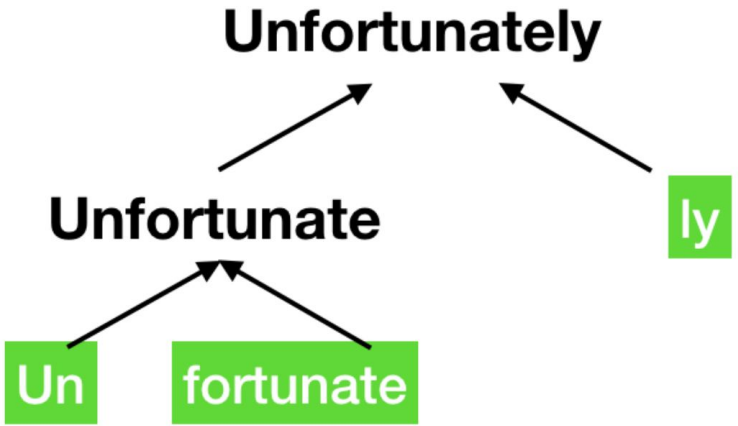
```
# prompt engineering for Translation
text = "По-русски: 'кот', по-английски:"
input_ids = tokenizer.encode(text, return_tensors="pt").to(DEVICE)
out = model.generate(input_ids, do_sample=False)

generated_text = list(map(tokenizer.decode, out))[0]
print(generated_text)

#>>> По-русски: 'кот', по-английски: 'cat'
```

Пример: кот –> cat

Токенизация

Hugging Face

```
# Изначальные текст
text = "Токенизируй меня"
# Процесс токенизации с помощью токенайзера ruGPT-3
tokens = tokenizer.encode(text, add_special_tokens=False)
# Обратная поэлементая токенизация
decoded_tokens = [tokenizer.decode([token]) for token in tokens]

print("text:", text)
print("tokens: ", tokens)
print("decoded tokens: ", decoded_tokens)

#>>> text:            Токенизируй меня
#>>> tokens:          [789, 368, 337, 848, 28306, 703]
#>>> decoded tokens:  ['Т', 'ок', 'ени', 'зи', 'руй', ' меня']
```



Код

Иллюстрация

Архитектура GPT

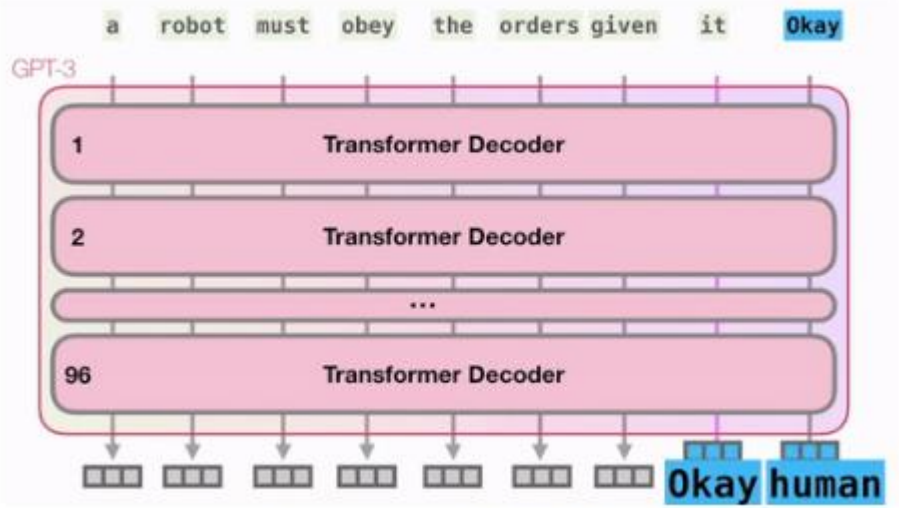
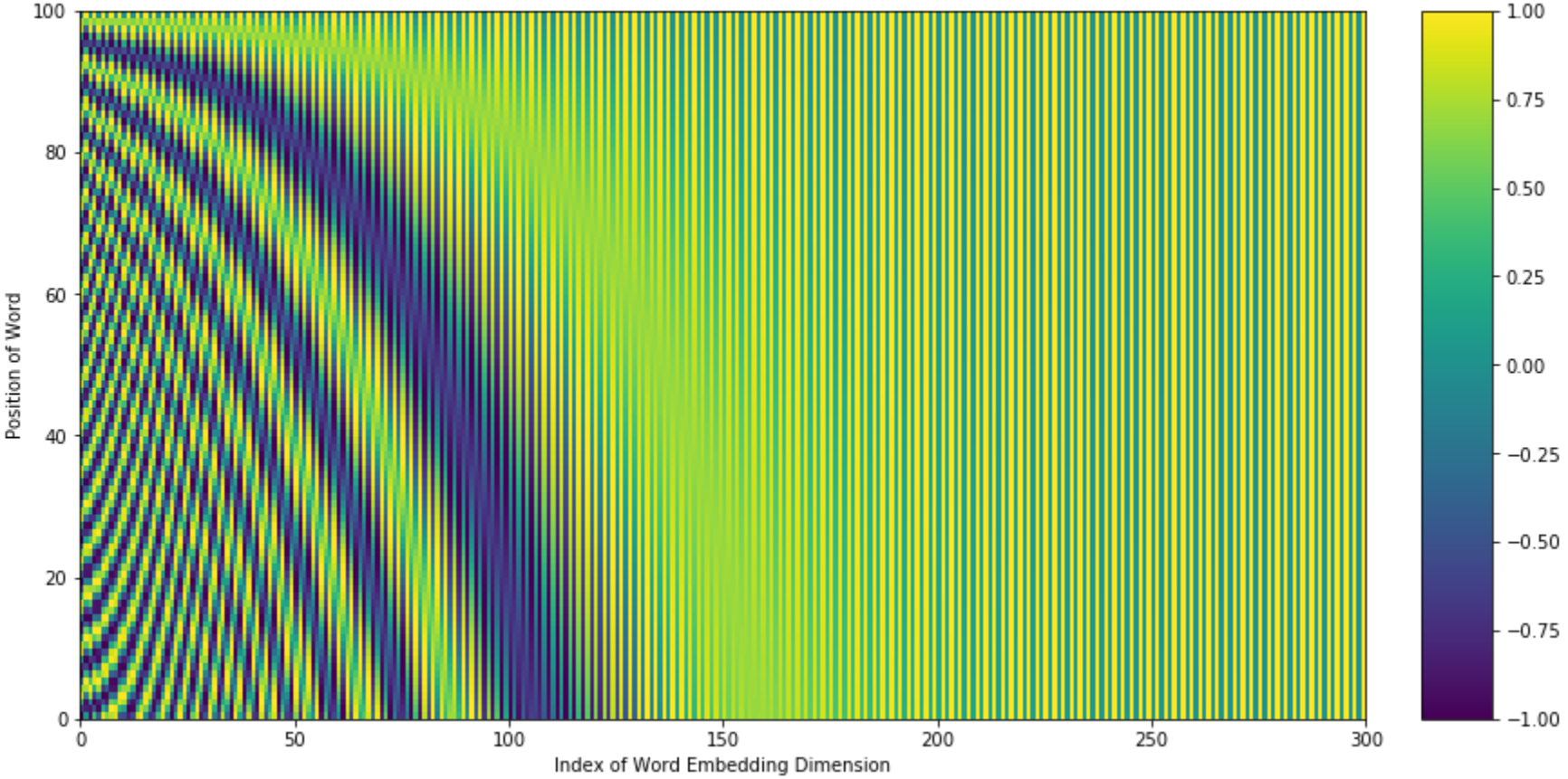


Схема работы GPT (продолжение текста)

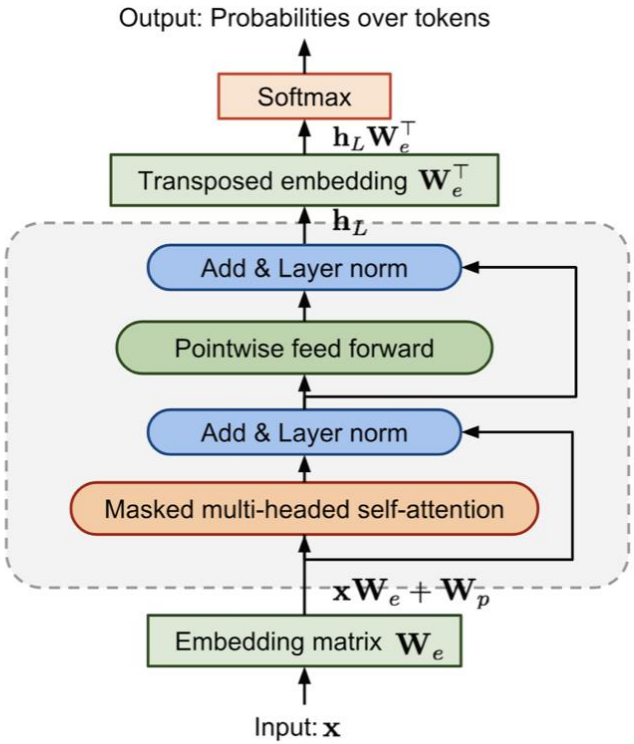
Последовательность шагов GPT

1. Токенизация
2. Получение эмбедингов (похоже на word2vec)
3. Добавляем эмбединг позиции
4. Пропускаем через последовательность Transformer Decoding Blocks
5. Умножаем выход на вход? берем softmax – получаем вероятности следующего токена
6. Делаем семплинг из распределения

Позиционные эмбеденги



Transformer Decoder Block

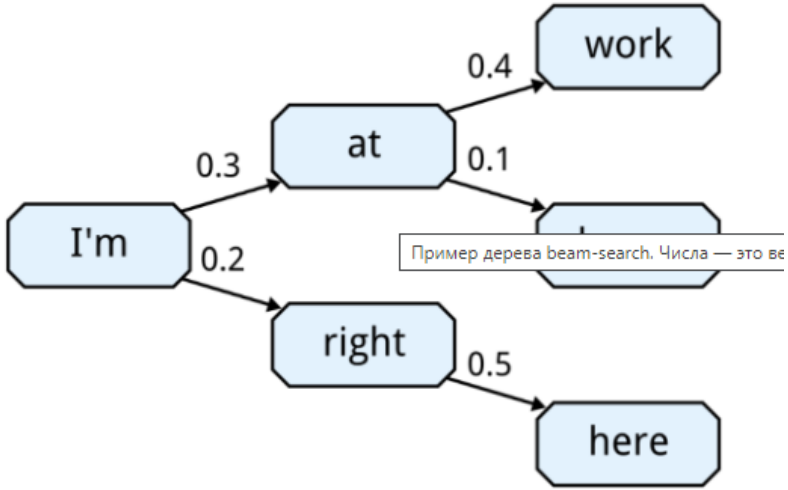


Получение результата - argmax

```
# Пример аргмаксного сэмплирования
out = model.generate(input_ids,
                     do_sample=False,
                     max_length=30)

# Декодирование токенов
generated_text = list(map(tokenizer.decode, out))[0]
print(generated_text)

#>>> Определение: "Нейронная сеть" - это компьютерная программа, которая
#>>> позволяет создавать и анализировать нейронные сети.
```

Получение результата – beam search

```
# Пример генерации с помощью beam-search
out = model.generate(input_ids,
                     do_sample=False,
                     num_beams=5,
                     max_length=30)

# Декодирование токенов
generated_text = list(map(tokenizer.decode, out))[0]
print(generated_text)

#>>> Определение: "Нейронная сеть" - это сеть, состоящая из множества
#>>> нейронов, соединенных друг с другом.
```



Пример дерева beam-search. Числа — это вероятности токенов.

Получение результата – семплинг

```
# Пример вероятностного сэмплирования
out = model.generate(input_ids,
                     do_sample=True,
                     temperature=1.3,
                     max_length=30)

# Декодирование токенов
generated_text = list(map(tokenizer.decode, out))[0]
print(generated_text)

#>>> Определение: "Нейронная сеть" - это модель, при которой каждый
#>>> элемент сети имеет "исполнительный элемент" - "всплеск.
```

```
# Пример вероятностного сэмплирвоания с ограничением
out = model.generate(input_ids,
                     do_sample=True,
                     temperature=1.3,
                     top_k=20,
                     top_p=0.8,
                     max_length=30,
                    )
# Декодирование токенов
generated_text = list(map(tokenizer.decode, out))[0]
print(generated_text)

#>>> Определение: "Нейронная сеть" - это совокупность объектов и программ,
#>>> объединенных единой целью и имеющих функциональный характер.
```