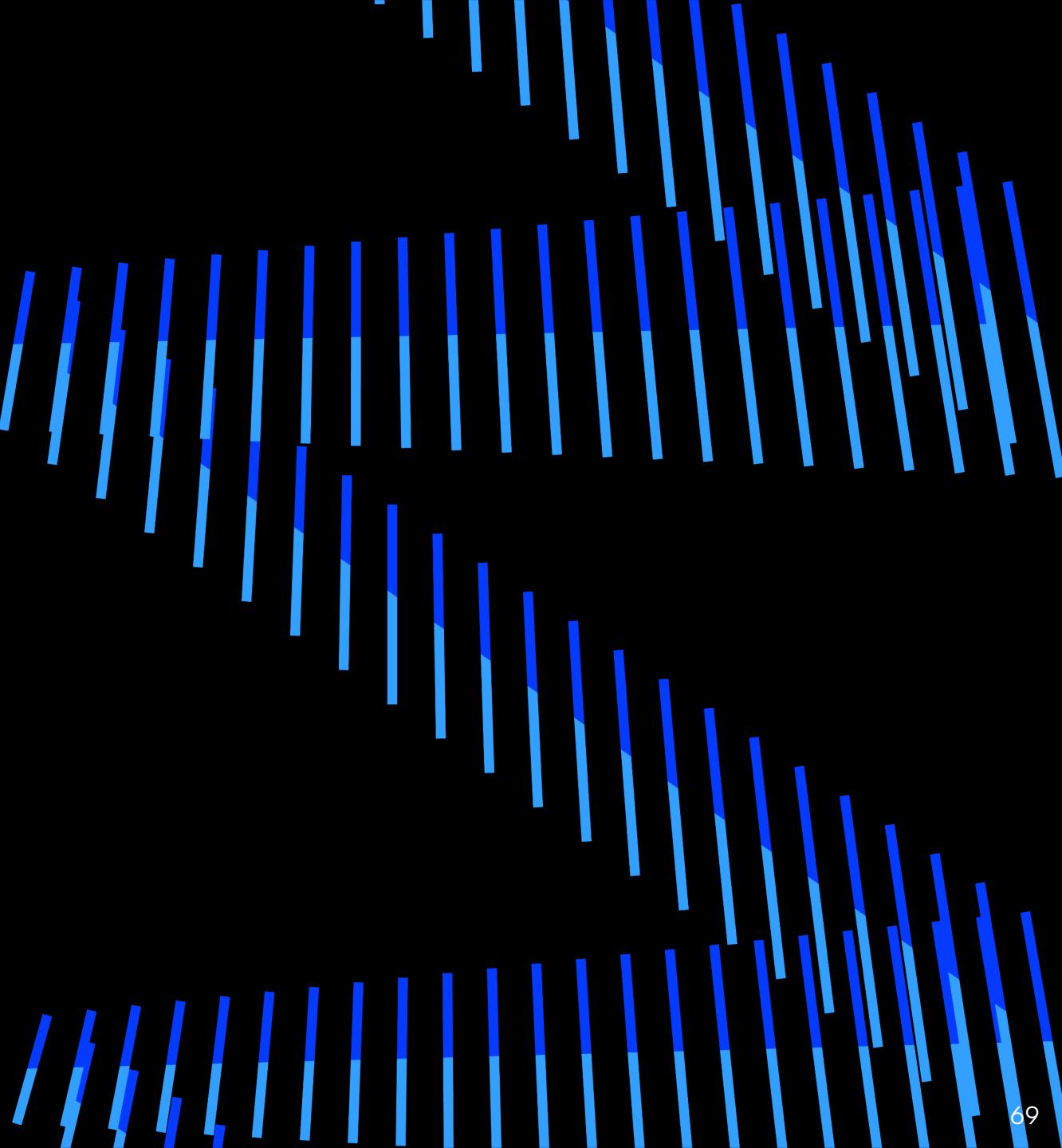


Архитектуры нейронных сетей



Обучение представлений данных

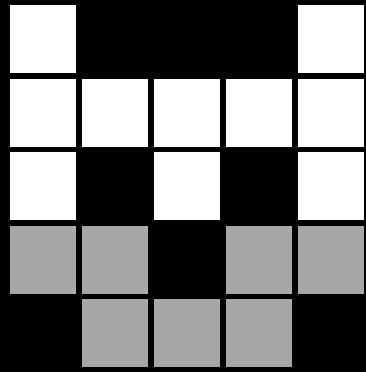


Обучение представлений данных

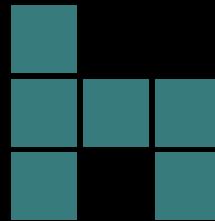
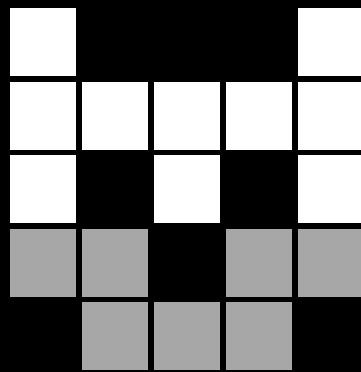


Изображения

Фильтры, свёртка, карты активаций

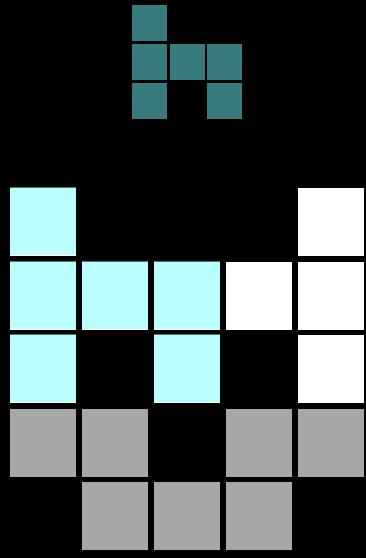


Фильтры, свёртка, карты активаций

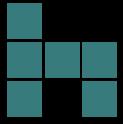


Кернел (фильтр, ядро) 3x3

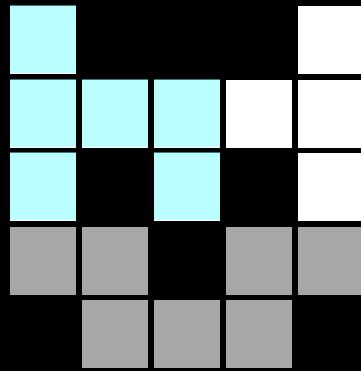
Фильтры, свёртка, карты активаций



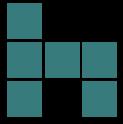
Фильтры, свёртка, карты активаций



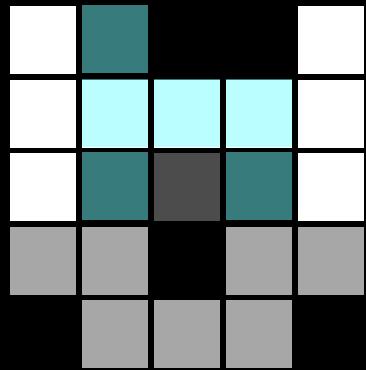
$$a = \sum_{i,j} \mathbf{w}_{ij} \mathbf{x}_{ij} = \mathbf{W}^T \mathbf{x} \quad \mathbf{x} - \text{патч изображения размером с кернел}$$



Фильтры, свёртка, карты активаций



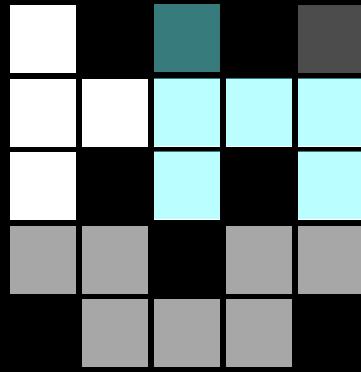
$$a = \sum_{i,j} \mathbf{w}_{ij} \mathbf{x}_{ij} = \mathbf{W}^T \mathbf{x} \quad \mathbf{x} - \text{патч изображения размером с кернел}$$



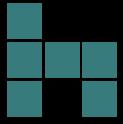
Фильтры, свёртка, карты активаций



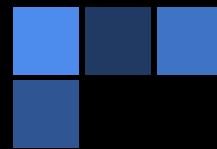
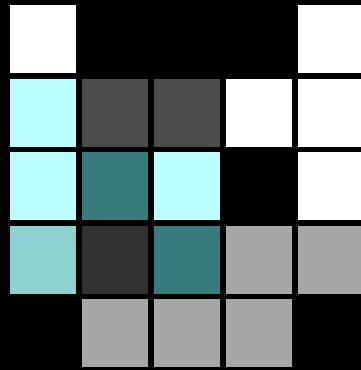
$$a = \sum_{i,j} \mathbf{w}_{ij} \mathbf{x}_{ij} = \mathbf{W}^T \mathbf{x} \quad \mathbf{x} - \text{патч изображения размером с кернел}$$



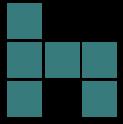
Фильтры, свёртка, карты активаций



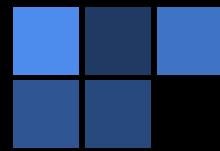
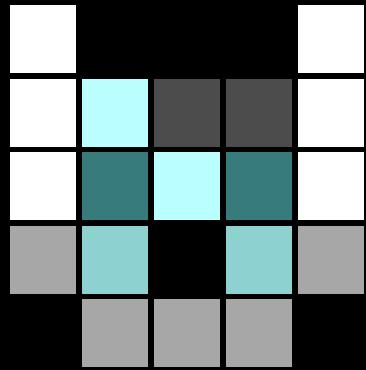
$$a = \sum_{i,j} \mathbf{w}_{ij} \mathbf{x}_{ij} = \mathbf{W}^T \mathbf{x} \quad \mathbf{x} - \text{патч изображения размером с кернел}$$



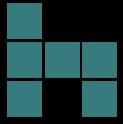
Фильтры, свёртка, карты активаций



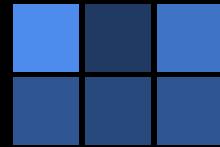
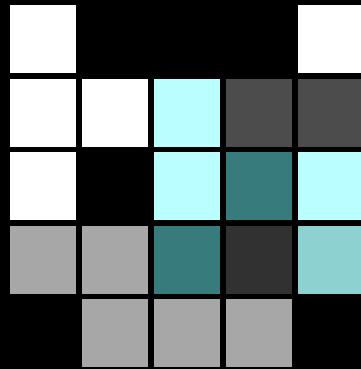
$$a = \sum_{i,j} \mathbf{w}_{ij} \mathbf{x}_{ij} = \mathbf{W}^T \mathbf{x} \quad \mathbf{x} - \text{патч изображения размером с кернел}$$



Фильтры, свёртка, карты активаций



$$a = \sum_{i,j} \mathbf{w}_{ij} \mathbf{x}_{ij} = \mathbf{W}^T \mathbf{x} \quad \mathbf{x} - \text{патч изображения размером с кернел}$$

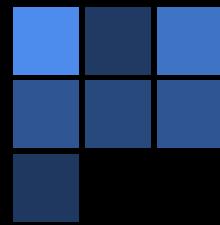
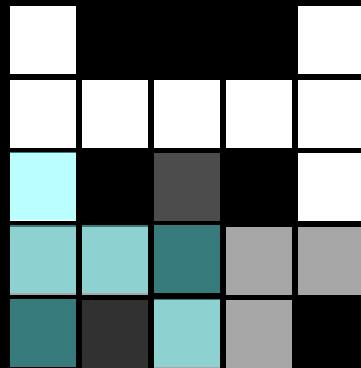


Фильтры, свёртка, карты активаций

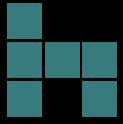


$$a = \sum_{i,j} \mathbf{w}_{ij} \mathbf{x}_{ij} = \mathbf{W}^T \mathbf{x}$$

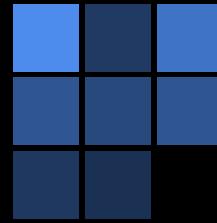
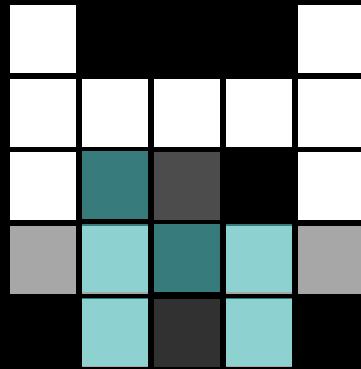
\mathbf{x} — патч изображения размером с кернел



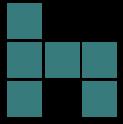
Фильтры, свёртка, карты активаций



$$a = \sum_{i,j} \mathbf{w}_{ij} \mathbf{x}_{ij} = \mathbf{W}^T \mathbf{x} \quad \mathbf{x} - \text{патч изображения размером с кернел}$$

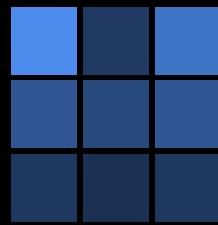
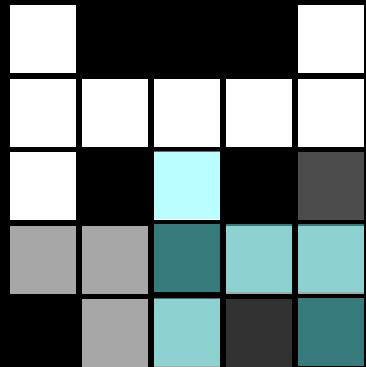


Фильтры, свёртка, карты активаций

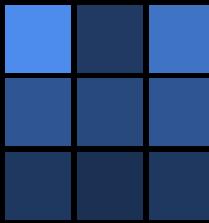
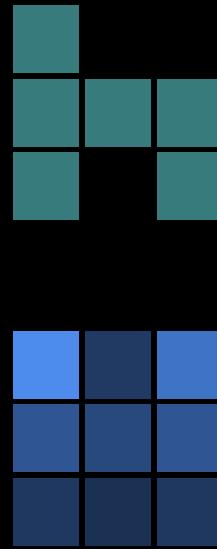
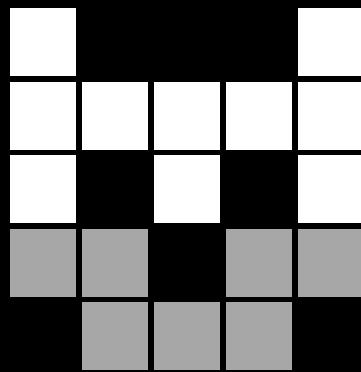


$$a = \sum_{i,j} \mathbf{w}_{ij} \mathbf{x}_{ij} = \mathbf{W}^T \mathbf{x}$$

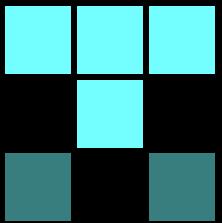
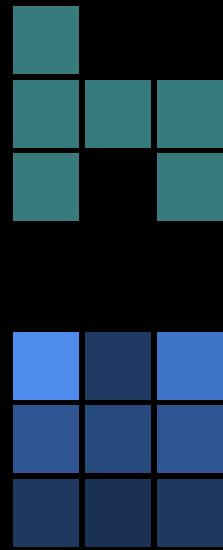
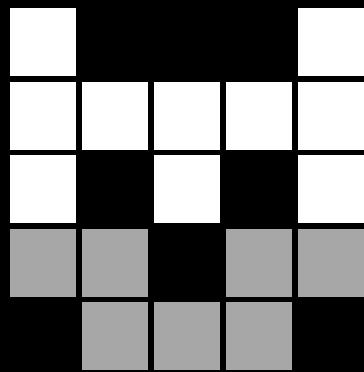
\mathbf{x} – патч изображения размером с кернел



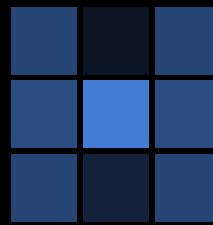
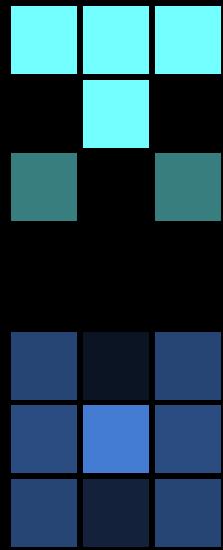
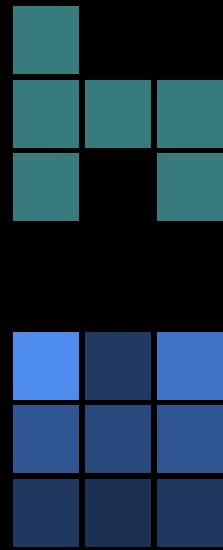
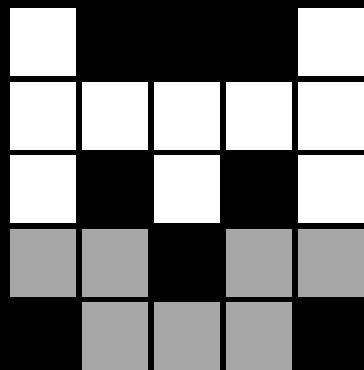
Фильтры, свёртка, карты активаций



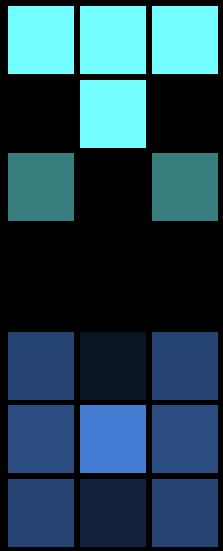
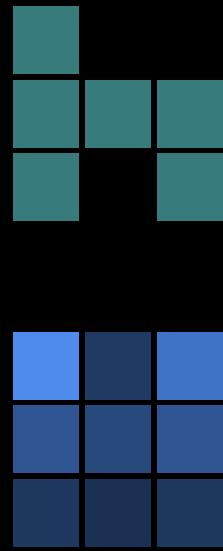
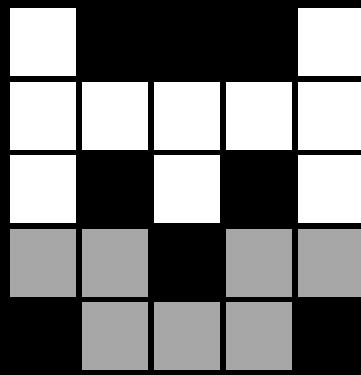
Фильтры, свёртка, карты активаций



Фильтры, свёртка, карты активаций

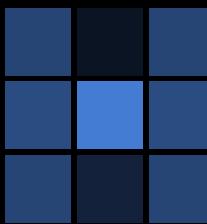
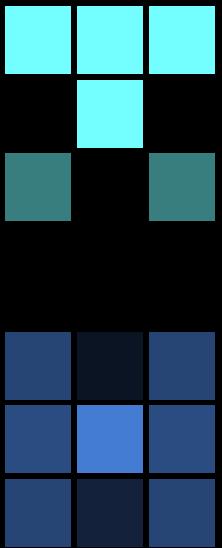
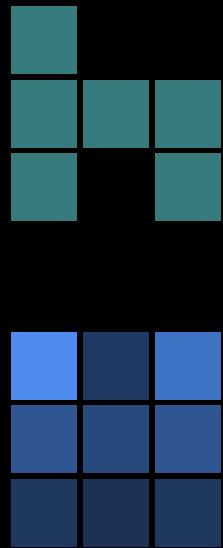
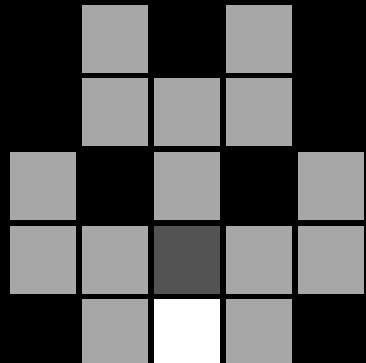
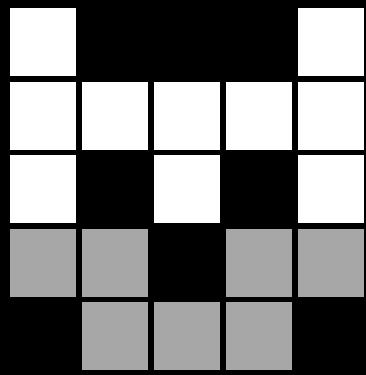


Фильтры, свёртка, карты активаций



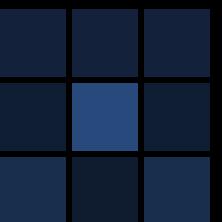
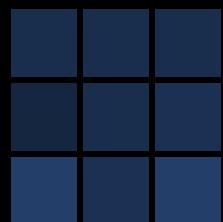
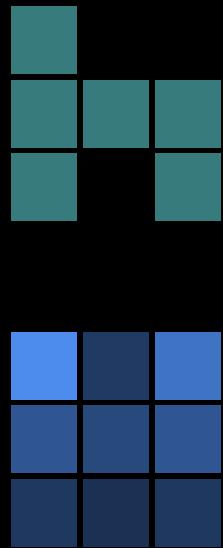
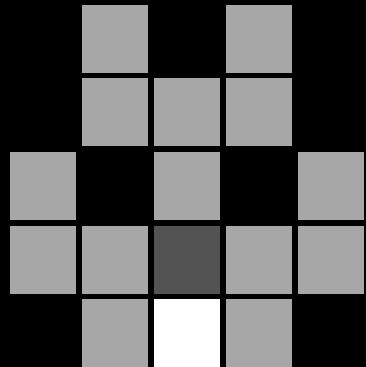
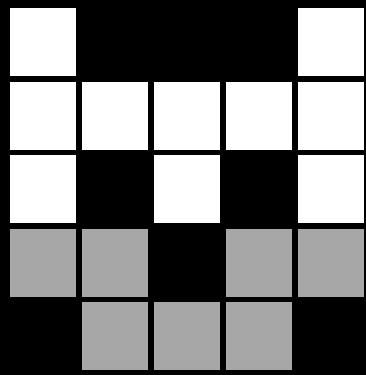
Карты активаций

Фильтры, свёртка, карты активаций



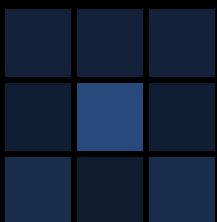
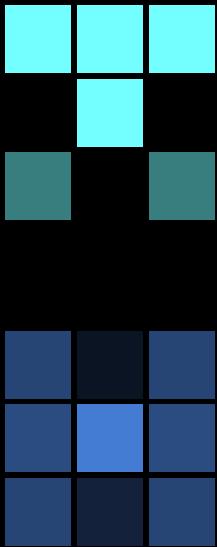
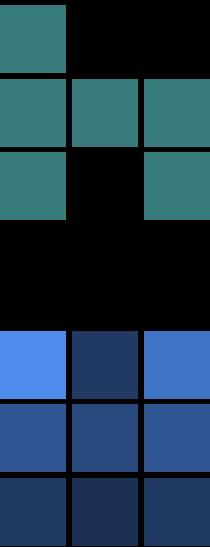
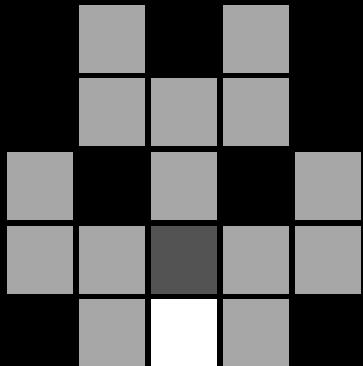
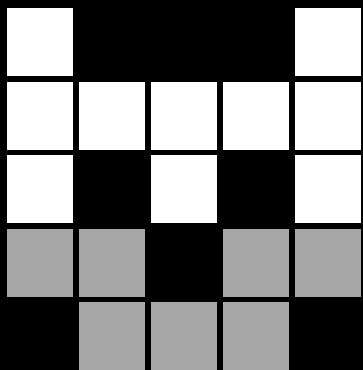
Карты активаций

Фильтры, свёртка, карты активаций



Карты активаций

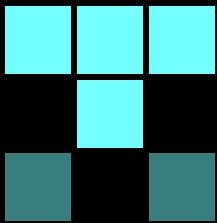
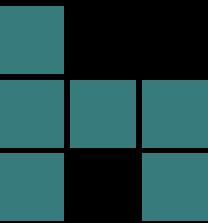
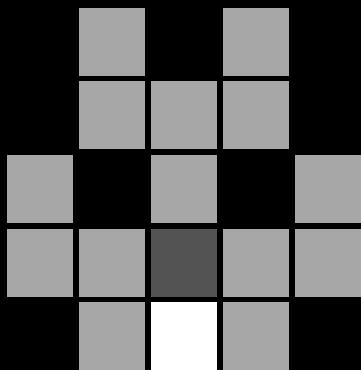
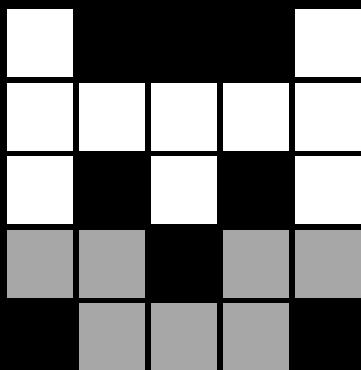
Свёрточная нейронная сеть



Max Pooling:

сжимаем пространственное разрешение, выбирая максимальную активацию в каждом канале

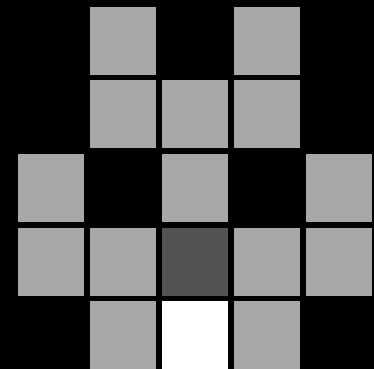
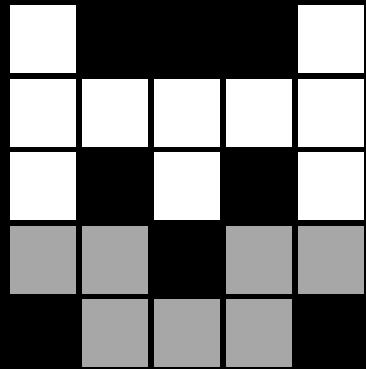
Свёрточная нейронная сеть



Max Pooling:

сжимаем пространственное разрешение, выбирая максимальную активацию в каждом канале

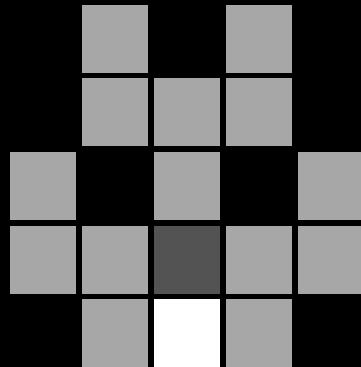
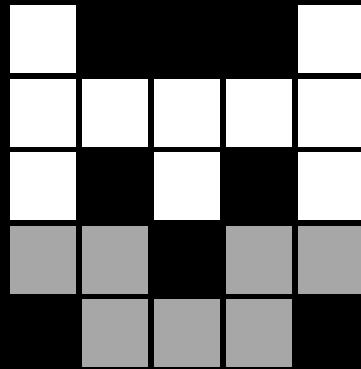
Свёрточная нейронная сеть



Max Pooling:

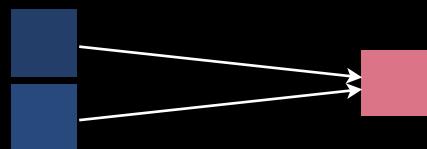
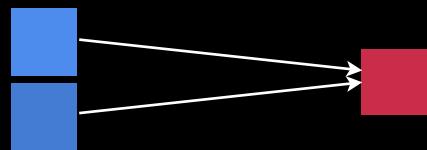
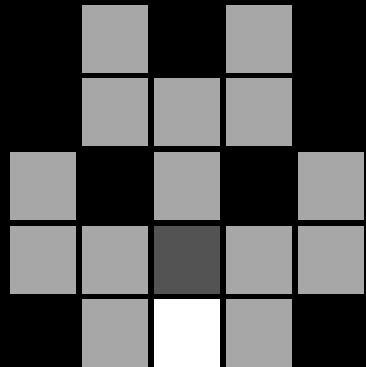
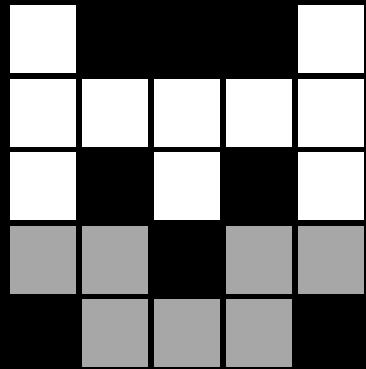
сжимаем пространственное разрешение, выбирая максимальную активацию в каждом канале

Свёрточная нейронная сеть

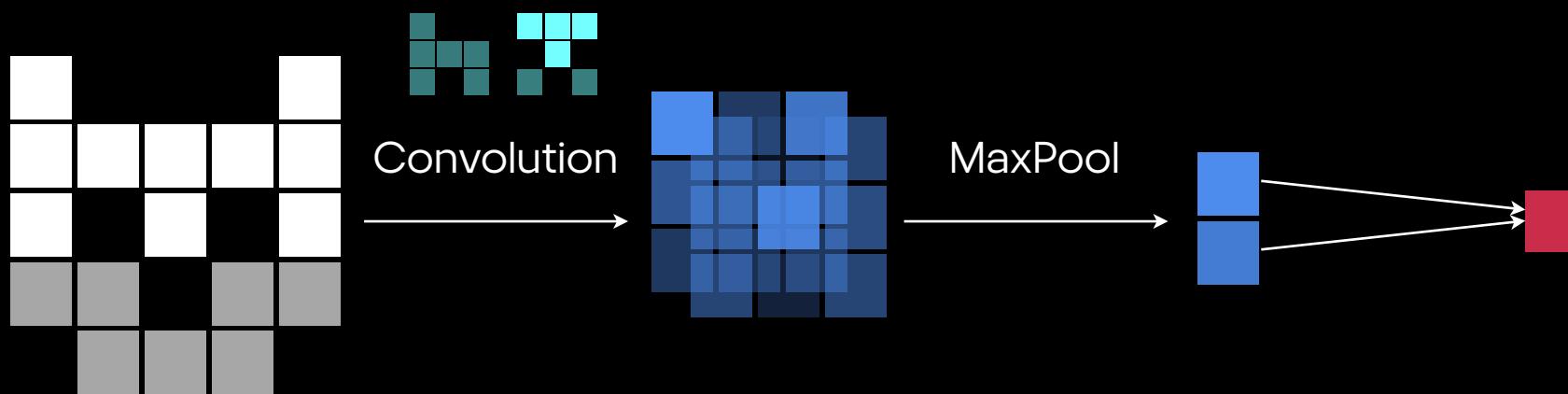


Embedding –
векторное
представление
изображения

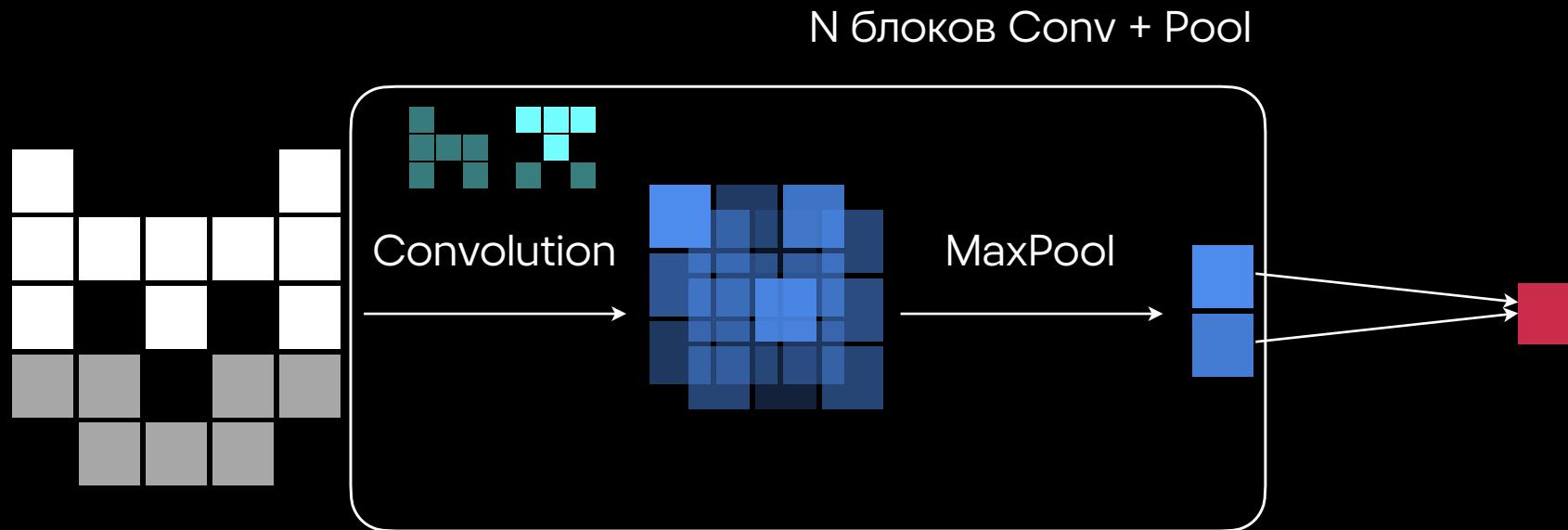
Свёрточная нейронная сеть



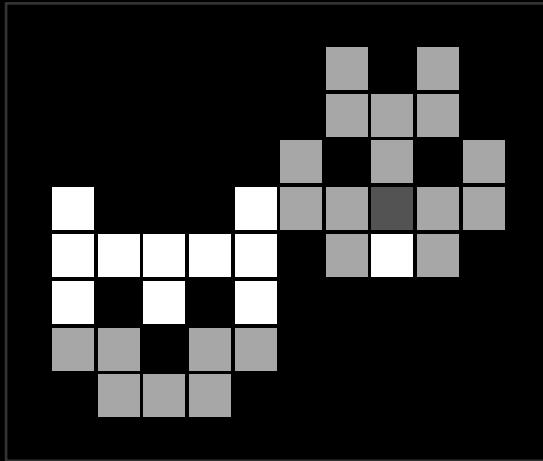
Свёрточная нейронная сеть



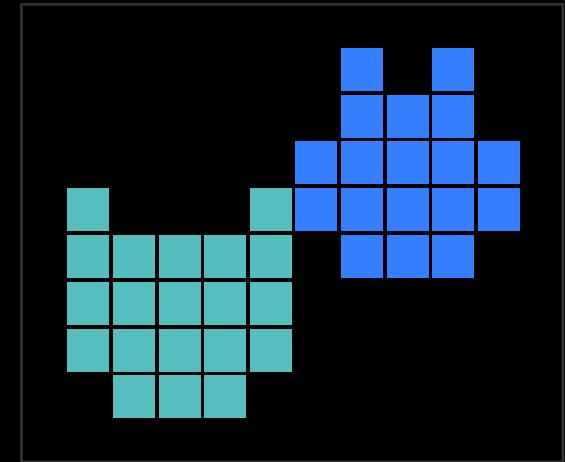
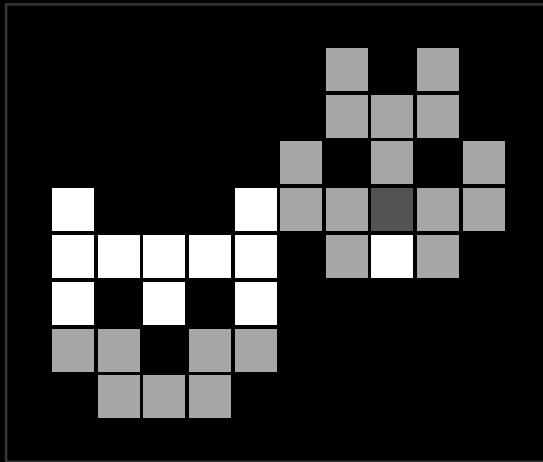
Свёрточная нейронная сеть



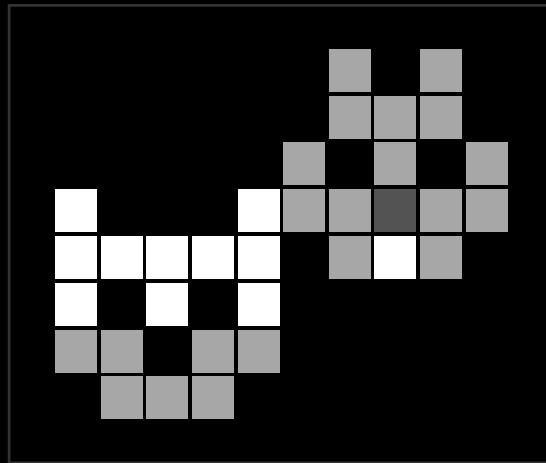
Архитектура для задачи сегментации



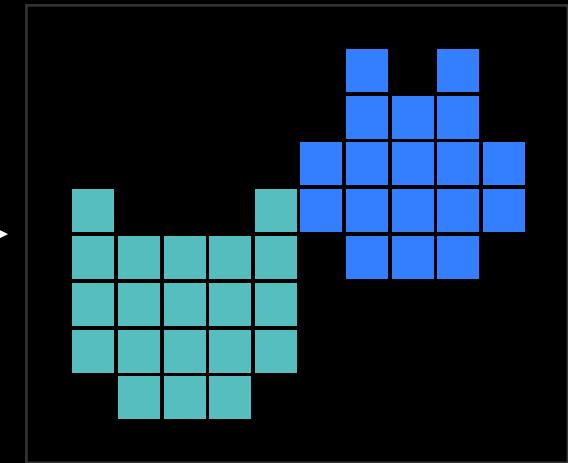
Архитектура для задачи сегментации



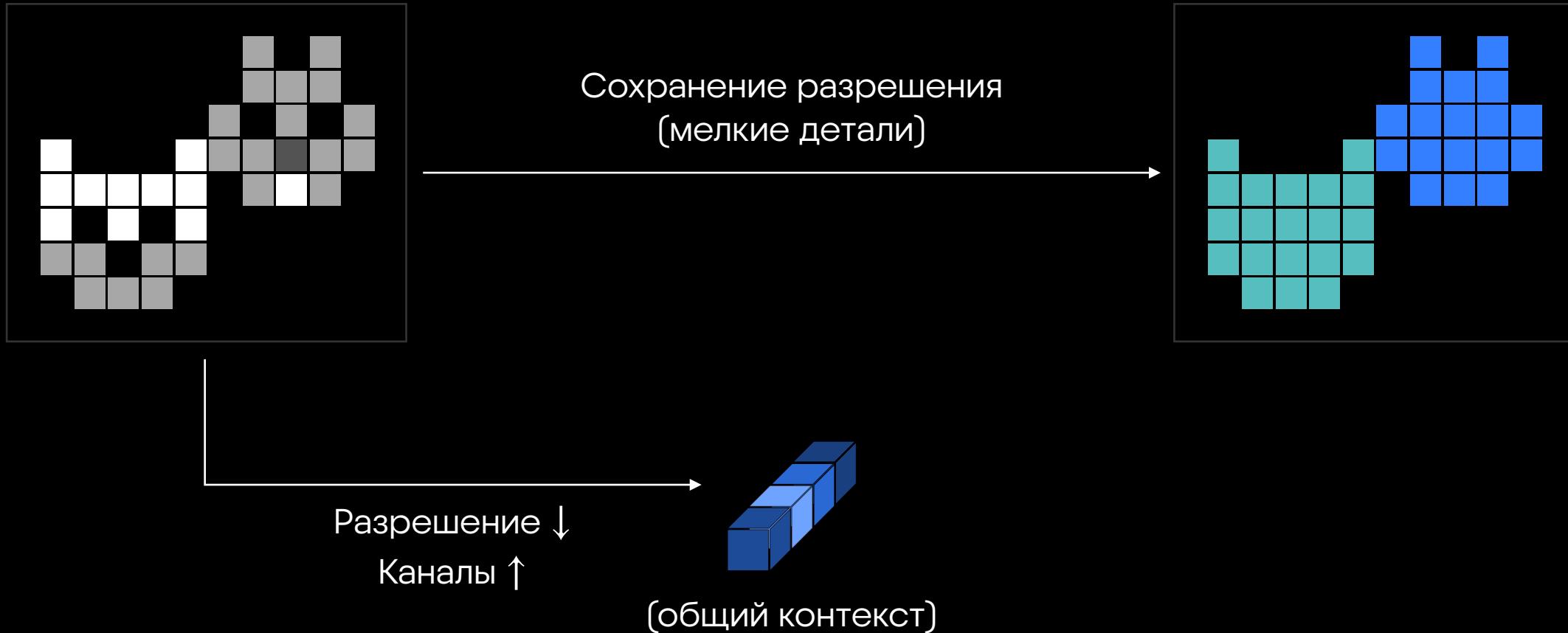
Архитектура для задачи сегментации



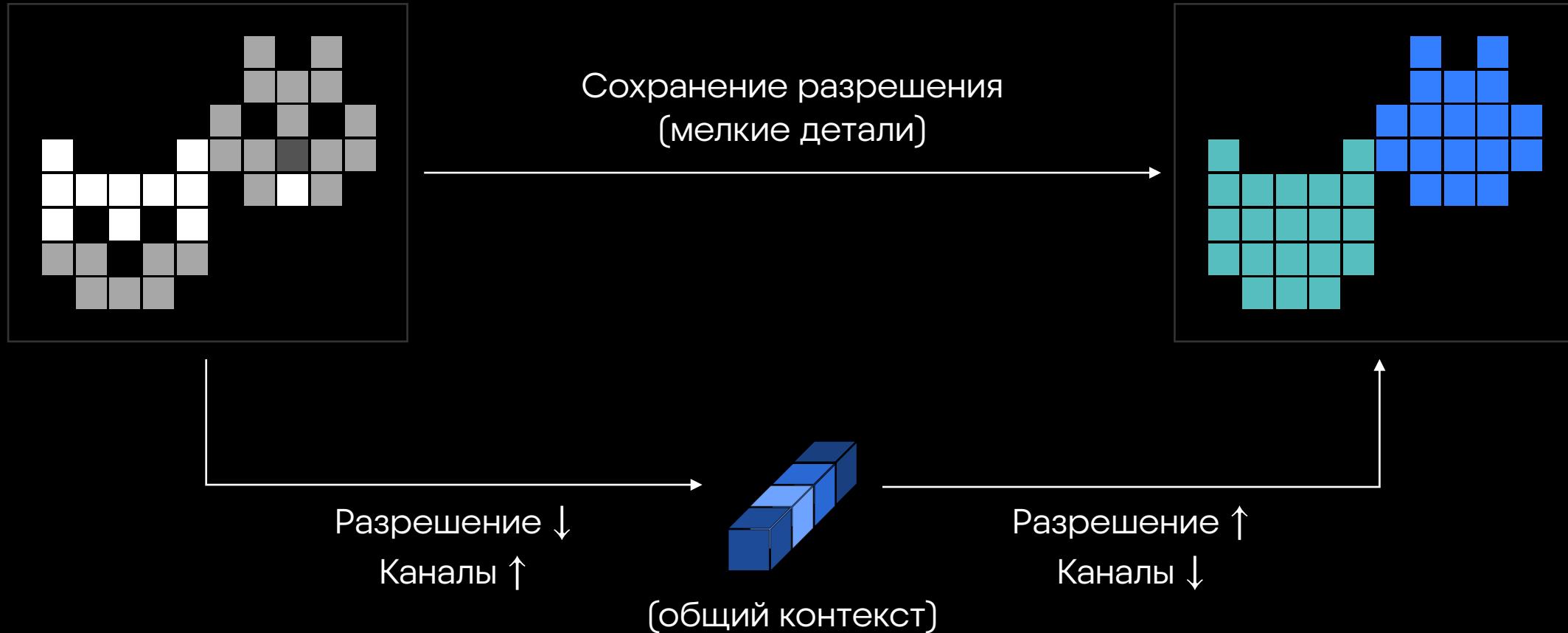
Сохранение разрешения
(мелкие детали)



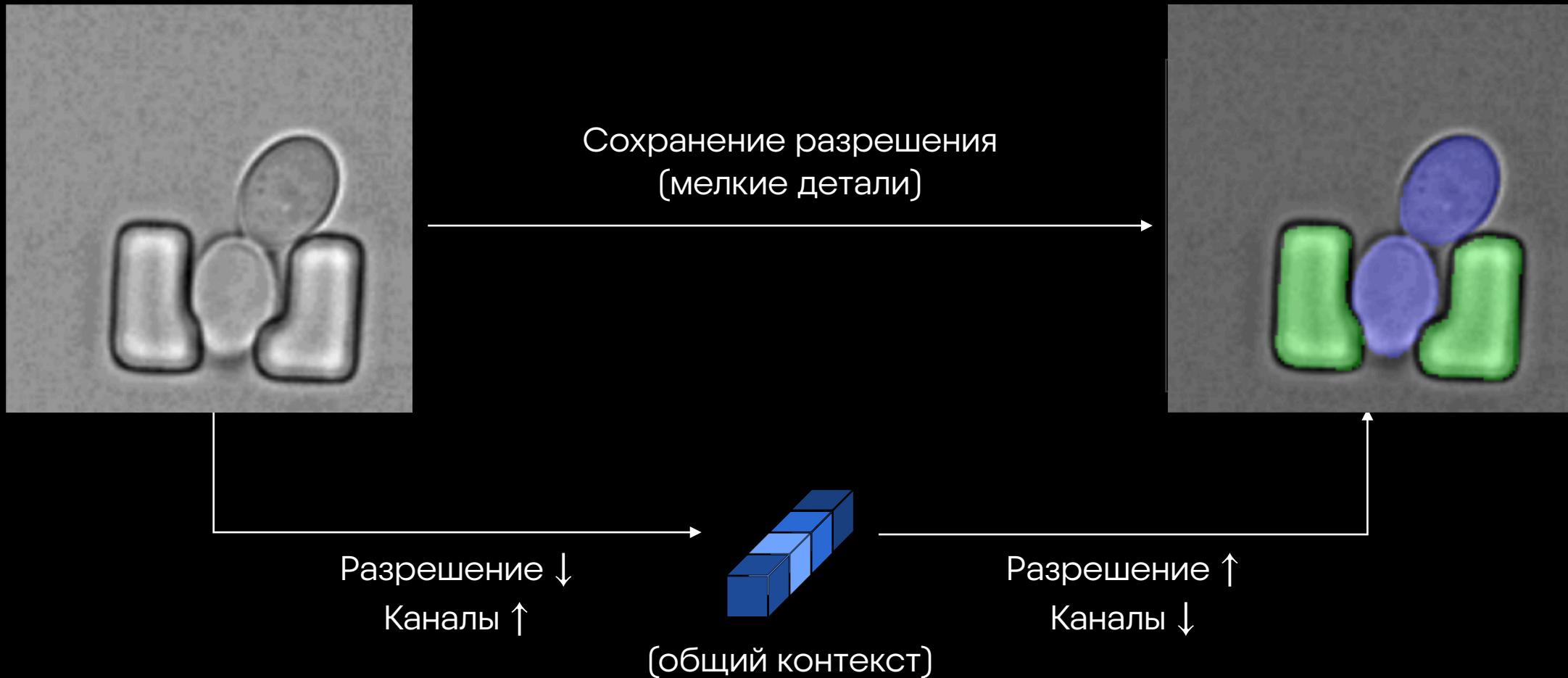
Архитектура для задачи сегментации



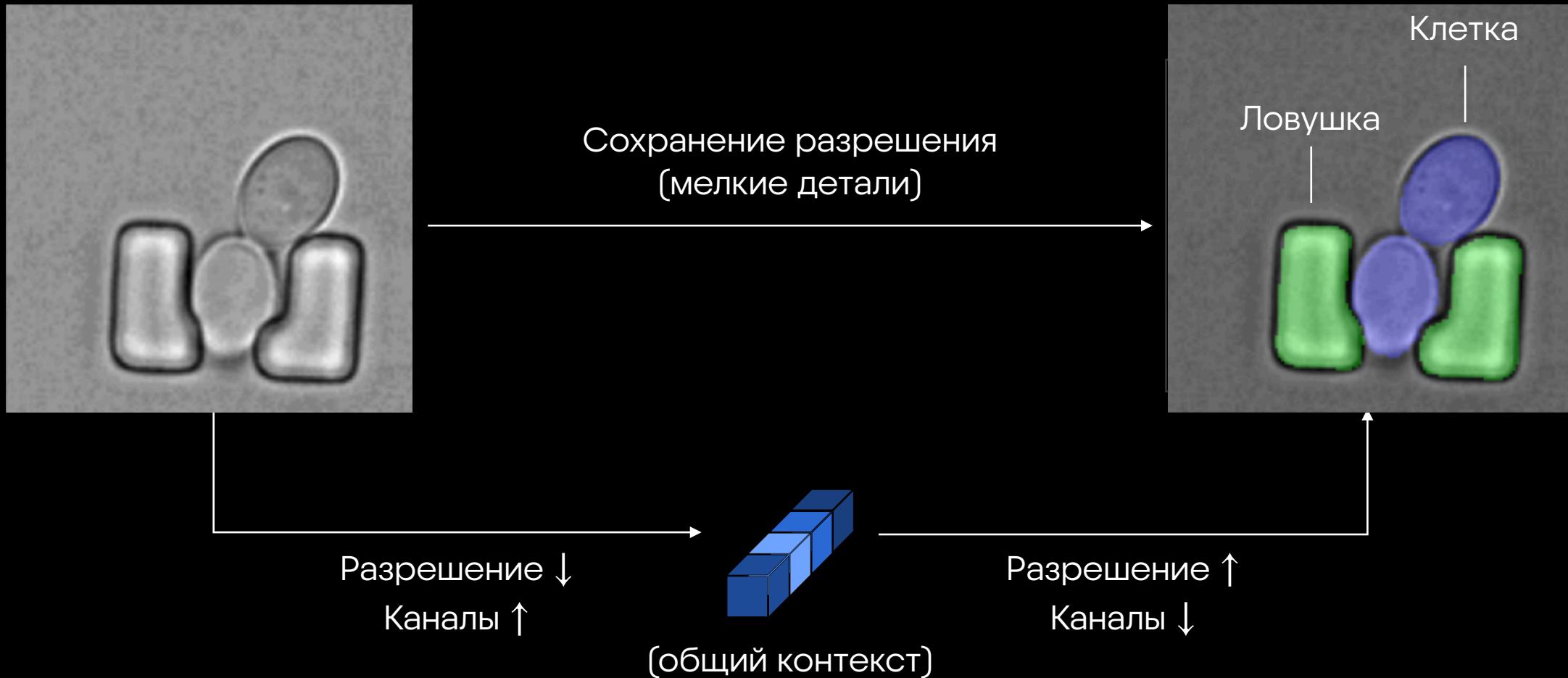
Архитектура для задачи сегментации



Проект №1. Сегментация изображений с клетками дрожжей



Проект №1. Сегментация изображений с клетками дрожжей



Обучение представлений данных



Изображения

Обучение представлений данных



QVQLVE...
DIQLTQ...
TVPPMV...

Последовательности



Изображения

Токены и их эмбеддинги

G A T T A C A

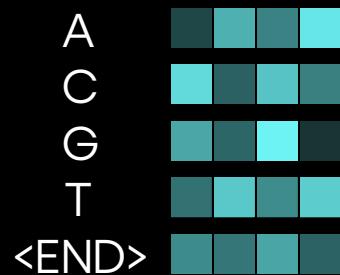
Токены и их эмбеддинги

G A T T A C A <END>

Токены — смысловые единицы последовательности
(слова, части слов, символы)

Токены и их эмбеддинги

Эмбеддинги токенов:
обучаемые векторные представления

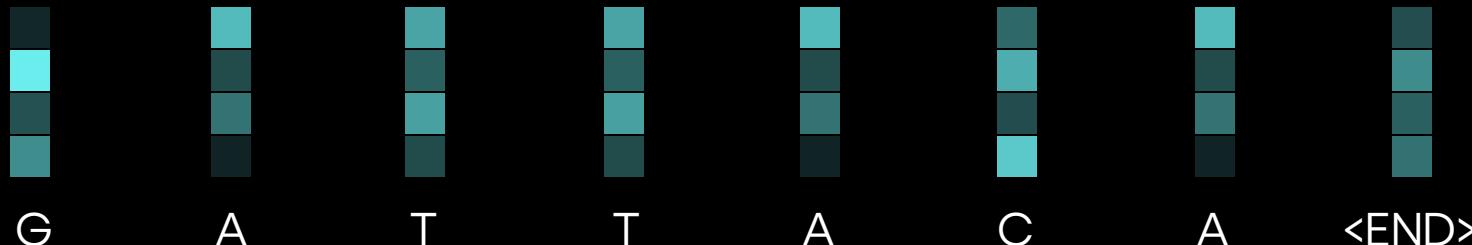
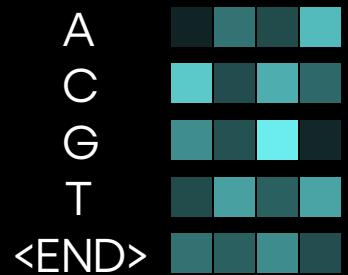


G A T T A C A <END>

Токены — смысловые единицы последовательности
(слова, части слов, символы)

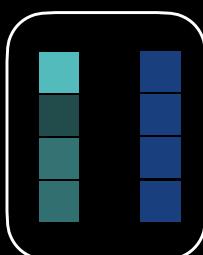
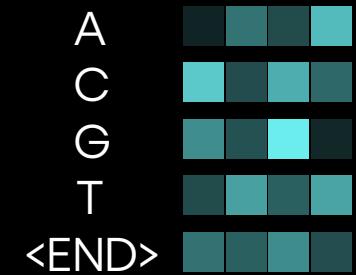
Токены и их эмбеддинги

Эмбеддинги токенов:
обучаемые векторные представления



Токены — смысловые единицы последовательности
(слова, части слов, символы)

Рекуррентная сеть (RNN)



G

A

T

T

A

C

A

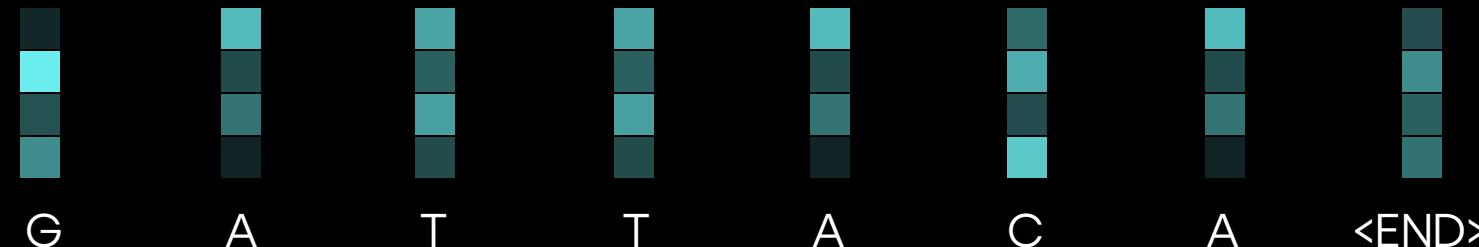
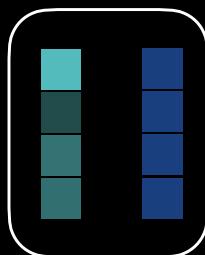
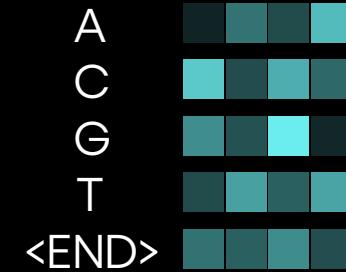
<END>

Рекуррентная сеть (RNN)

Веса рекуррентной ячейки

$$h^{(t+1)} = \tanh \left(\mathbf{W}_1 h^{(t)} + \mathbf{W}_2 x^{(t)} \right)$$

Следующее состояние Текущее состояние Эмбеддинг токена

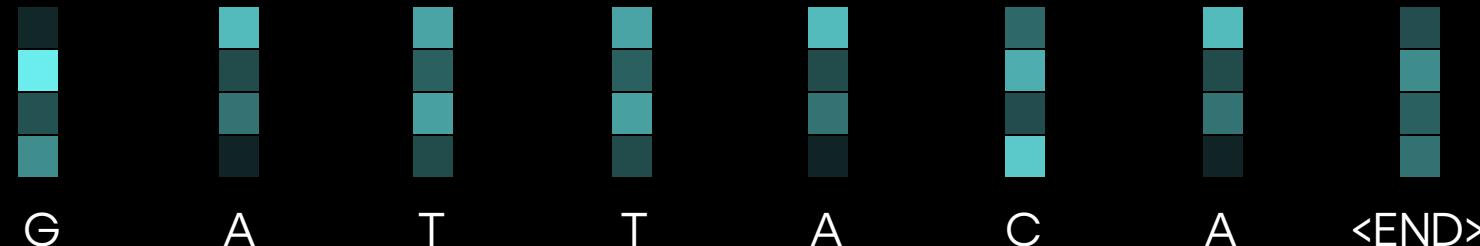
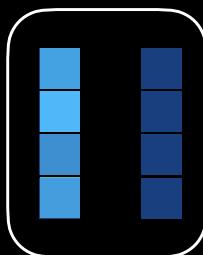
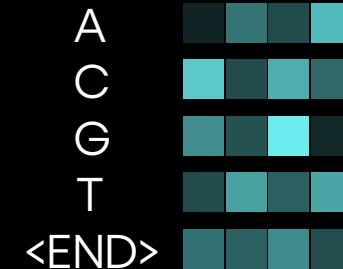


Рекуррентная сеть (RNN)

Веса рекуррентной ячейки

$$h^{(t+1)} = \tanh \left(\mathbf{W}_1 h^{(t)} + \mathbf{W}_2 x^{(t)} \right)$$

Следующее состояние Текущее состояние Эмбеддинг токена

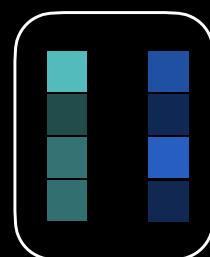


Рекуррентная сеть (RNN)

Веса рекуррентной ячейки

$$h^{(t+1)} = \tanh \left(\mathbf{W}_1 h^{(t)} + \mathbf{W}_2 x^{(t)} \right)$$

Следующее состояние Текущее состояние Эмбеддинг токена

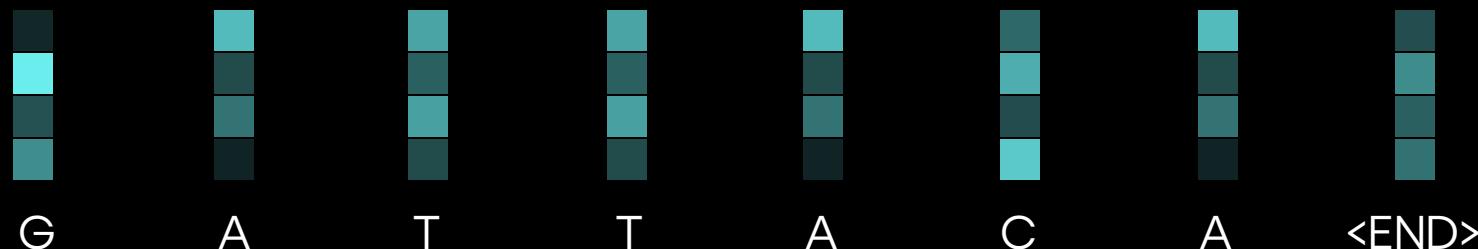
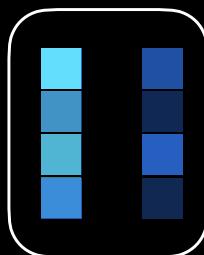


Рекуррентная сеть (RNN)

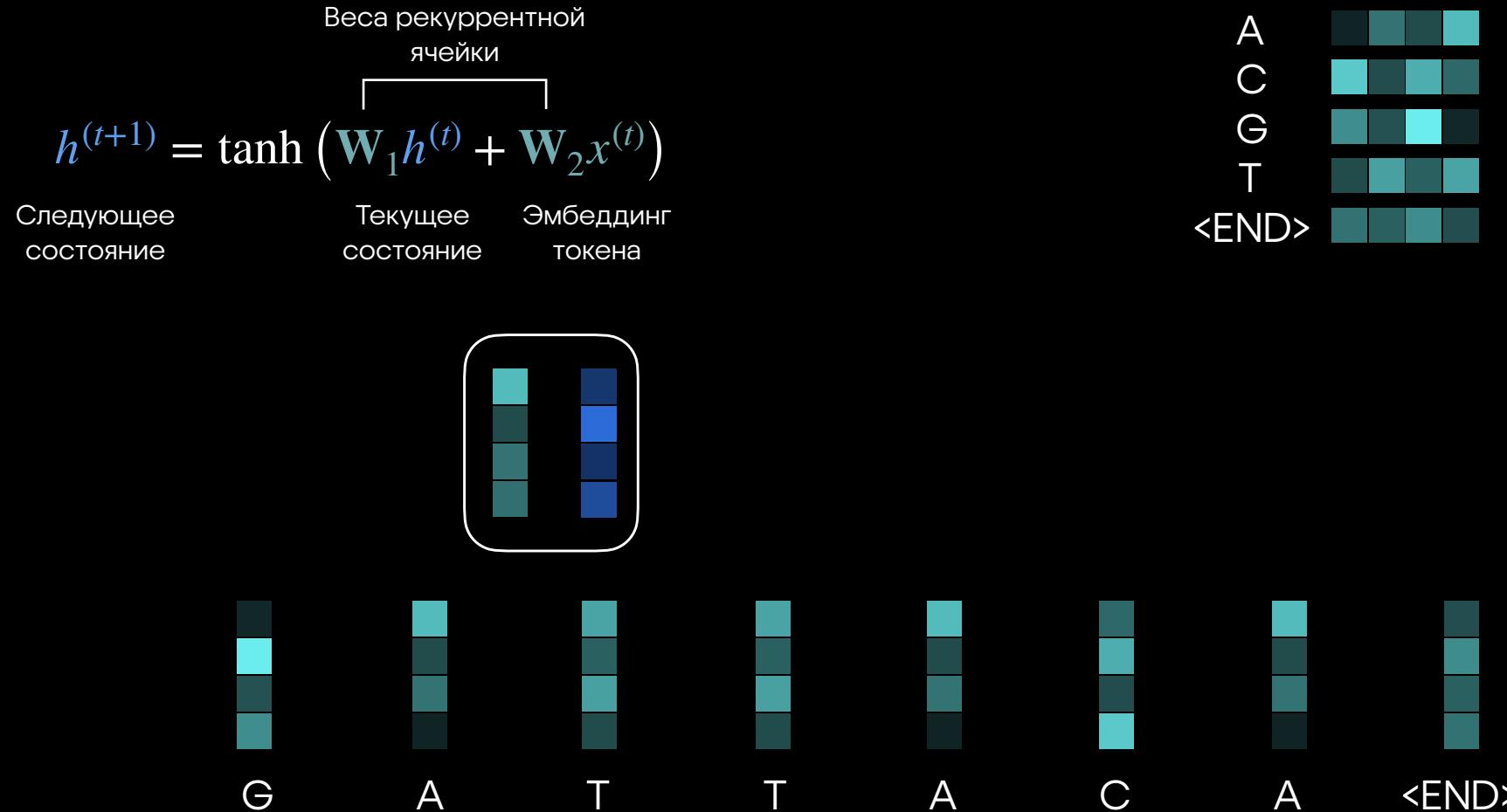
Веса рекуррентной ячейки

$$h^{(t+1)} = \tanh \left(\mathbf{W}_1 h^{(t)} + \mathbf{W}_2 x^{(t)} \right)$$

Следующее состояние Текущее состояние Эмбеддинг токена



Рекуррентная сеть (RNN)

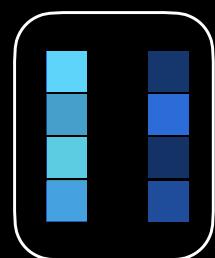


Рекуррентная сеть (RNN)

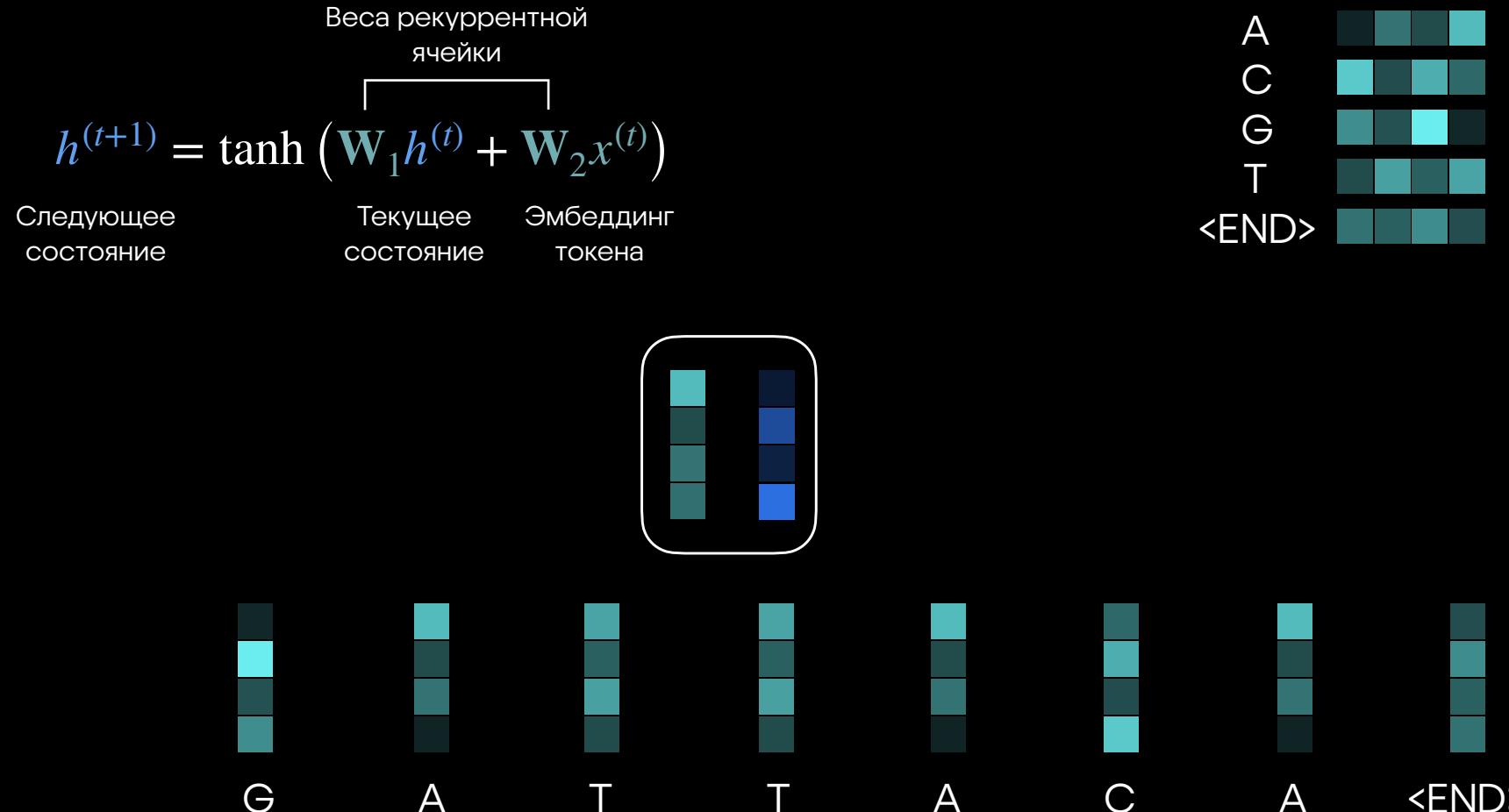
Веса рекуррентной ячейки

$$h^{(t+1)} = \tanh \left(\mathbf{W}_1 h^{(t)} + \mathbf{W}_2 x^{(t)} \right)$$

Следующее состояние Текущее состояние Эмбеддинг токена



Рекуррентная сеть (RNN)

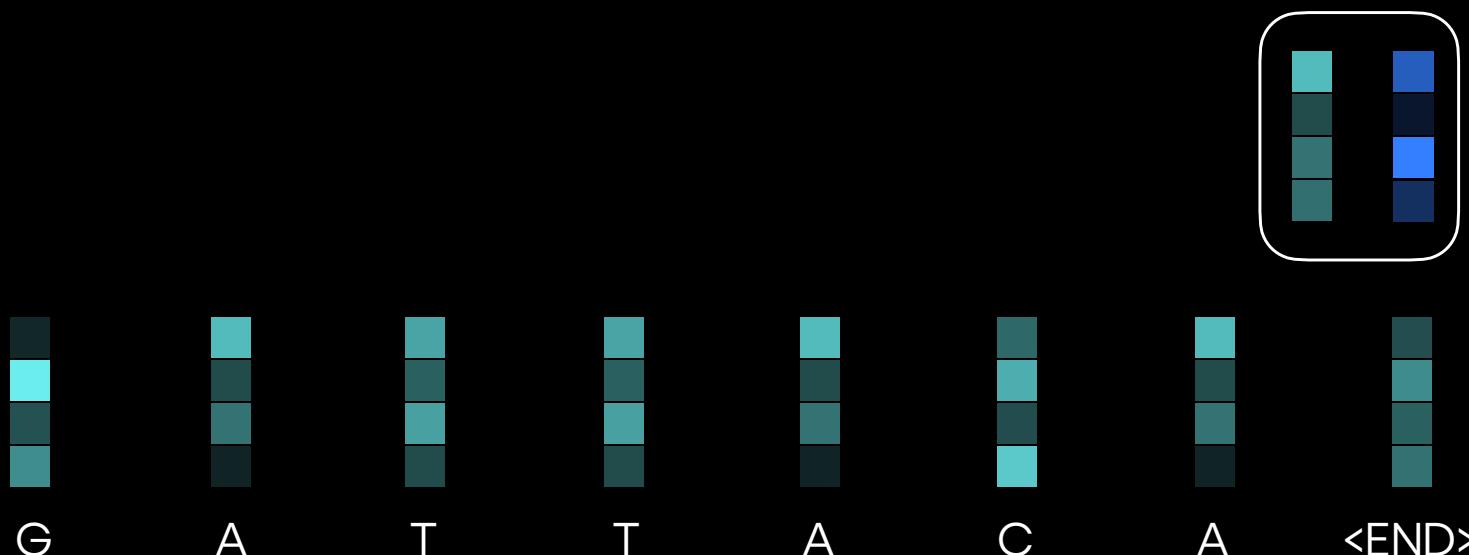


Рекуррентная сеть (RNN)

Веса рекуррентной ячейки

$$h^{(t+1)} = \tanh \left(\mathbf{W}_1 h^{(t)} + \mathbf{W}_2 x^{(t)} \right)$$

Следующее состояние Текущее состояние Эмбеддинг токена



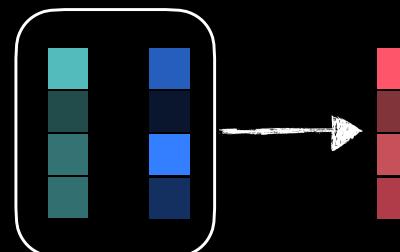
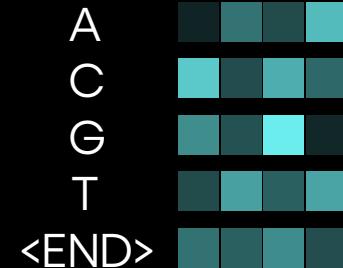
Рекуррентная сеть (RNN)

Веса рекуррентной ячейки

$$h^{(t+1)} = \tanh \left(\mathbf{W}_1 h^{(t)} + \mathbf{W}_2 x^{(t)} \right)$$

Следующее состояние Текущее состояние Эмбеддинг токена

A C G T
<END>



G A T T A C A <END>

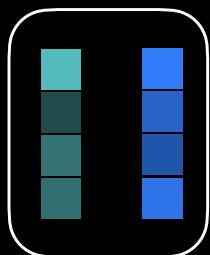
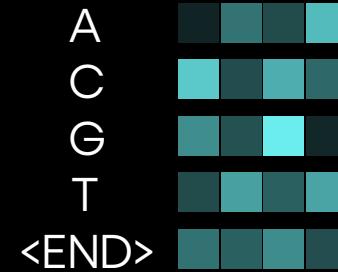


Генерация последовательности

Веса рекуррентной ячейки

$$h^{(t+1)} = \tanh \left(\mathbf{W}_1 h^{(t)} + \mathbf{W}_2 x^{(t)} \right)$$

Следующее состояние Текущее состояние Эмбеддинг токена



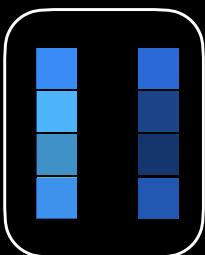
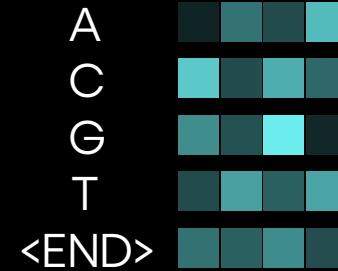
G

Генерация последовательности

Веса рекуррентной ячейки

$$h^{(t+1)} = \tanh \left(\underbrace{\mathbf{W}_1 h^{(t)} + \mathbf{W}_2 x^{(t)}}_{\text{Следующее состояние}} + \underbrace{\mathbf{b}}_{\text{Текущее состояние}} \right) + \mathbf{c}$$

Эмбеддинг токена



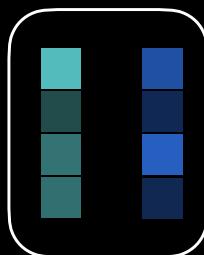
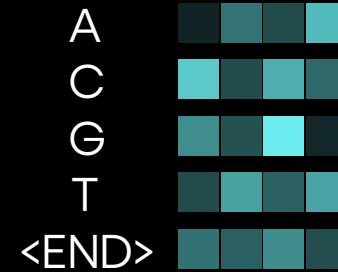
G

Генерация последовательности

Веса рекуррентной ячейки

$$h^{(t+1)} = \tanh \left(\mathbf{W}_1 h^{(t)} + \mathbf{W}_2 x^{(t)} \right)$$

Следующее состояние Текущее состояние Эмбеддинг токена



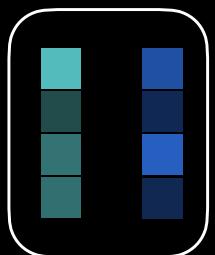
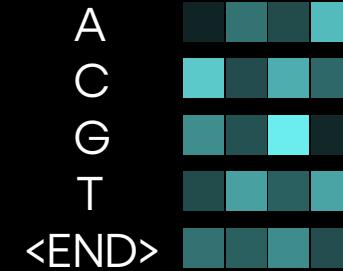
G

Генерация последовательности

Веса рекуррентной ячейки

$$h^{(t+1)} = \tanh \left(\mathbf{W}_1 h^{(t)} + \mathbf{W}_2 x^{(t)} \right)$$

Следующее состояние Текущее состояние Эмбеддинг токена



A
 C
 G
 T
 <END>

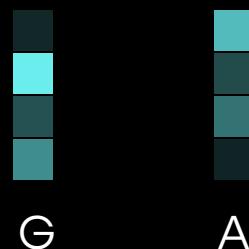
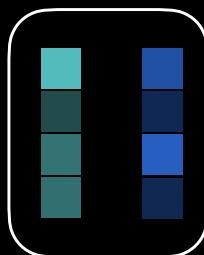
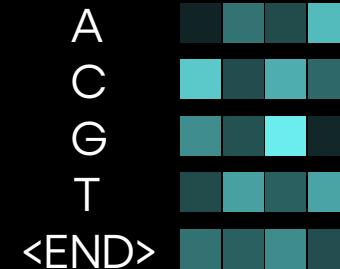
G

Генерация последовательности

Веса рекуррентной ячейки

$$h^{(t+1)} = \tanh \left(\mathbf{W}_1 h^{(t)} + \mathbf{W}_2 x^{(t)} \right)$$

Следующее состояние Текущее состояние Эмбеддинг токена

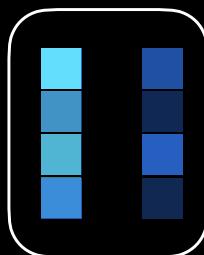
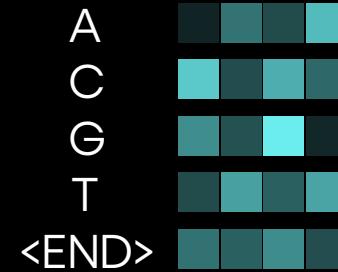


Генерация последовательности

Веса рекуррентной ячейки

$$h^{(t+1)} = \tanh \left(\mathbf{W}_1 h^{(t)} + \mathbf{W}_2 x^{(t)} \right)$$

Следующее состояние Текущее состояние Эмбеддинг токена



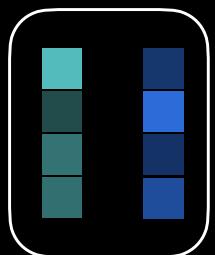
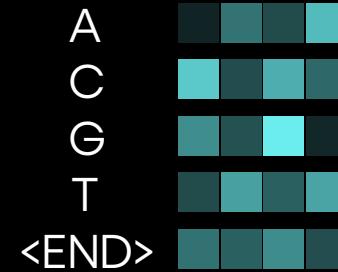
G A

Генерация последовательности

Веса рекуррентной ячейки

$$h^{(t+1)} = \tanh \left(\mathbf{W}_1 h^{(t)} + \mathbf{W}_2 x^{(t)} \right)$$

Следующее состояние Текущее состояние Эмбеддинг токена



G

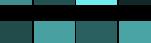
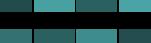
A

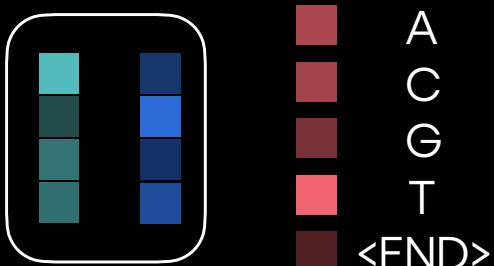
Генерация последовательности

Веса рекуррентной ячейки

$$h^{(t+1)} = \tanh \left(\mathbf{W}_1 h^{(t)} + \mathbf{W}_2 x^{(t)} \right)$$

Следующее состояние Текущее состояние Эмбеддинг токена

A 
C 
G 
T 
<END> 



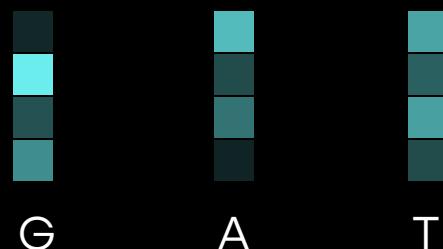
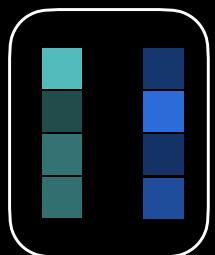
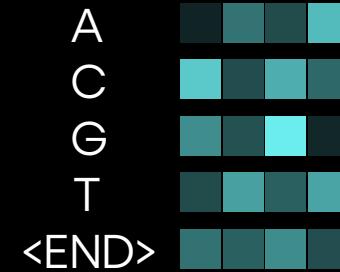
 G  A

Генерация последовательности

Веса рекуррентной ячейки

$$h^{(t+1)} = \tanh \left(\mathbf{W}_1 h^{(t)} + \mathbf{W}_2 x^{(t)} \right)$$

Следующее состояние Текущее состояние Эмбеддинг токена

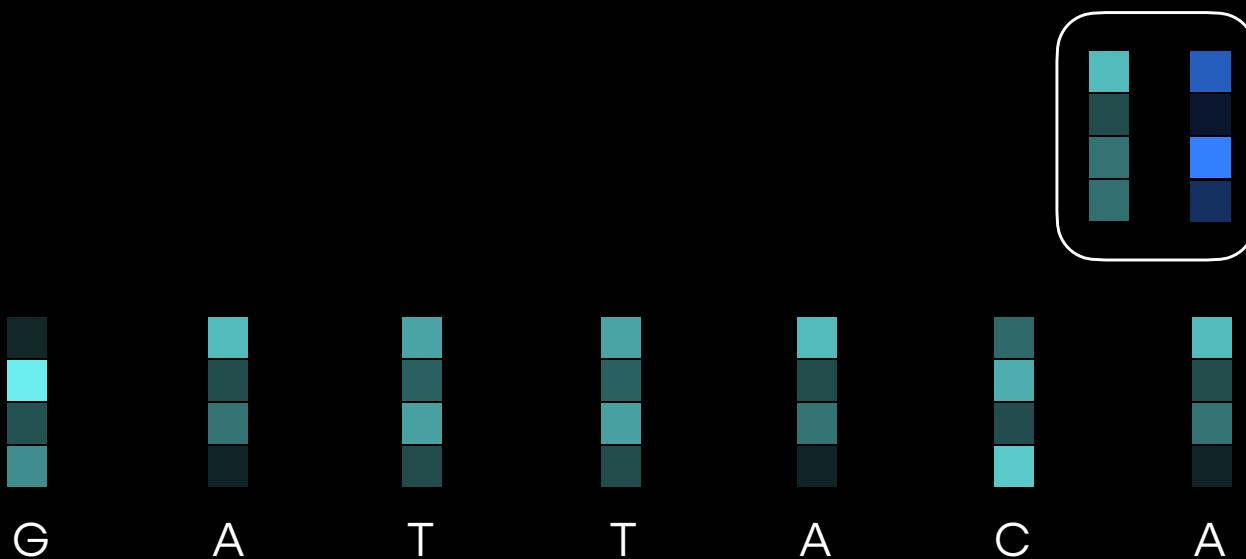
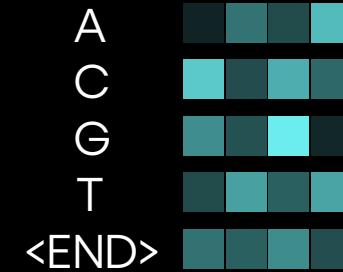


Генерация последовательности

Веса рекуррентной ячейки

$$h^{(t+1)} = \tanh \left(\underbrace{\mathbf{W}_1 h^{(t)} + \mathbf{W}_2 x^{(t)}}_{\text{Следующее состояние}} + \underbrace{\mathbf{b}}_{\text{Текущее состояние}} \right) + \mathbf{c}$$

Эмбеддинг токена

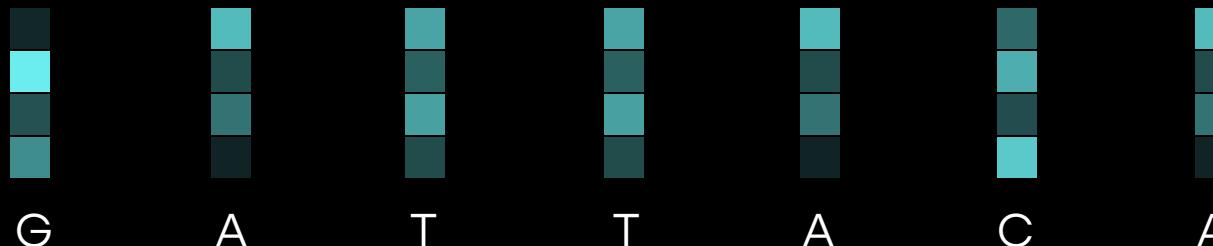
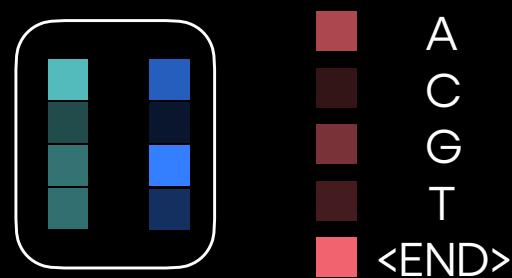
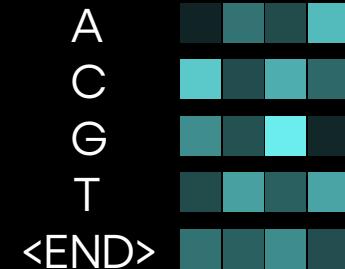


Генерация последовательности

Веса рекуррентной ячейки

$$h^{(t+1)} = \tanh \left(\mathbf{W}_1 h^{(t)} + \mathbf{W}_2 x^{(t)} \right)$$

Следующее состояние Текущее состояние Эмбеддинг токена

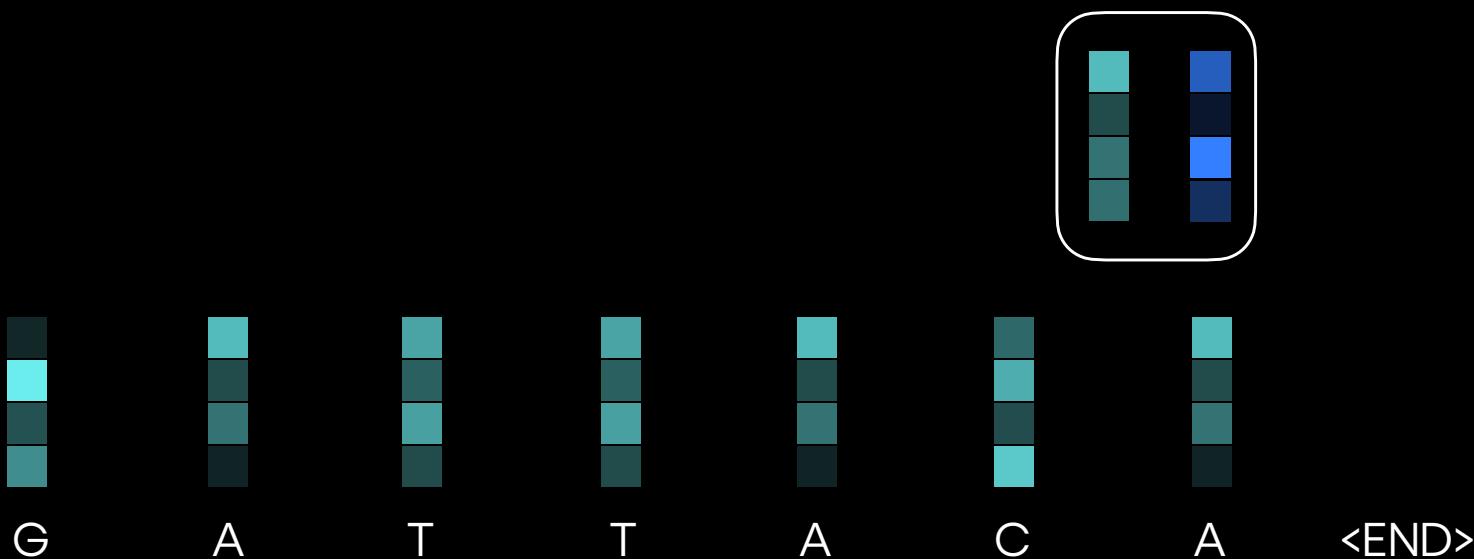


Генерация последовательности

Веса рекуррентной ячейки

$$h^{(t+1)} = \tanh \left(\underbrace{\mathbf{W}_1 h^{(t)} + \mathbf{W}_2 x^{(t)}}_{\text{Следующее состояние}} + \underbrace{\mathbf{b}}_{\text{Текущее состояние}} \right) + \mathbf{c}$$

Эмбеддинг токена



Рекуррентная сеть на pytorch

Веса рекуррентной ячейки

$$h^{(t+1)} = \tanh \left(\mathbf{W}_1 h^{(t)} + \mathbf{W}_2 x^{(t)} \right)$$

Следующее состояние Текущее состояние Эмбеддинг токена

```
class RNNCell(nn.Module):
    def __init__(self, input_dim: int, hidden_dim: int):
        super().__init__()
        self.linear = nn.Linear(
            input_dim + hidden_dim, hidden_dim
        )

    def forward(self, x: Tensor, h: Tensor) -> Tensor:
        h = torch.cat([x, h], dim=1)
        h = self.linear(h)
        return F.tanh(h)
```

```
class RNN(nn.Module):
    def __init__(self, vocab_size: int, hidden_dim: int) -> None:
        super().__init__()
        self.embed = nn.Embedding(vocab_size, hidden_dim)
        self.init_h = nn.Parameter(data=torch.randn(1, hidden_dim))
        self.rnn = RNNCell(hidden_dim, hidden_dim)
        self.lm_head = nn.Linear(hidden_dim, vocab_size)

    def forward(self, x: Tensor) -> Tensor:
        # x: B x T
        # embed(x): B x T -> B x T x hidden_dim
        B, T = x.shape

        x = self.embed(x)  # B x T x hidden_dim
        h = self.init_h.expand((B, -1))  # B x hidden_dim

        logits = [] # T x B x V
        for t in range(T):
            xt = x[:, t, :]
            h = self.rnn.forward(xt, h) # B x hidden
            y = self.lm_head(h).unsqueeze(1) # B x 1 x hidden
            logits.append(y)
            # save prediction for step t + 1

        # lm_head: B x T x hidden -> B x T x V
        return torch.cat(logits, dim=1)
```

Выучивание дальних и разреженных зависимостей

$\mathbf{X} \sim (n, d)$ – последовательность из n токенов размерности d

Выучивание дальних и разреженных зависимостей

$\mathbf{X} \sim (n, d)$ – последовательность из n токенов размерности d

$$\mathbf{h}_i = \sum_{j=1}^{2k+1} \mathbf{W}_j \mathbf{x}_{i+k+1-j} \text{ — 1D-свёртка с размером фильтра } 2k+1$$

Сумма по токенам внутри области видимости

Выучивание дальних и разреженных зависимостей

$\mathbf{X} \sim (n, d)$ – последовательность из n токенов размерности d

$$\mathbf{h}_i = \sum_{j=1}^{2k+1} \mathbf{W}_j \mathbf{x}_{i+k+1-j} - 1\text{D-свёртка с размером фильтра } 2k+1$$

Сумма по токенам внутри области видимости

$$\mathbf{h}_i = \sum_{j=1}^n g(i - j) \mathbf{x}_j$$

Сумма по всем токенам

Выучивание дальних и разреженных зависимостей

$\mathbf{X} \sim (n, d)$ – последовательность из n токенов размерности d

$$\mathbf{h}_i = \sum_{j=1}^{2k+1} \mathbf{W}_j \mathbf{x}_{i+k+1-j} - 1\text{D-свёртка с размером фильтра } 2k+1$$

Сумма по токенам внутри области видимости

$$\mathbf{h}_i = \sum_{j=1}^n g(i - j) \mathbf{x}_j \longrightarrow \mathbf{h}_i = \sum_{j=1}^n g(\mathbf{x}_i, \mathbf{x}_j) \mathbf{x}_j$$

Сумма по всем токенам

Scaled dot-product attention

$$\mathbf{h}_i = \sum_{j=1}^n g(\mathbf{x}_i, \mathbf{x}_j) \mathbf{x}_j$$

Scaled dot-product attention

$$\mathbf{h}_i = \sum_{j=1}^n g(\mathbf{x}_i, \mathbf{x}_j) \mathbf{x}_j$$

$$g(\mathbf{x}_i, \mathbf{x}_j) = \text{softmax}_j \left(\frac{1}{\sqrt{d}} \mathbf{x}_i^T \mathbf{x}_j \right) \mathbf{x}_j$$

softmax_j — применение softmax к $\left\{ g(\mathbf{x}_i, \mathbf{x}_j) \right\}_{j=1}^n$ независимо по всем i

Scaled dot-product attention

$$\mathbf{h}_i = \sum_{j=1}^n g(\mathbf{x}_i, \mathbf{x}_j) \mathbf{x}_j$$
$$g(\mathbf{x}_i, \mathbf{x}_j) = \text{softmax}_j \left(\frac{1}{\sqrt{d}} \mathbf{x}_i^T \mathbf{x}_j \right) \mathbf{x}_j$$

Self-attention

Ключи: $\mathbf{k}_i = \mathbf{W}_k^T \mathbf{x}_i$

Значения: $\mathbf{v}_i = \mathbf{W}_v^T \mathbf{x}_i$

Запросы: $\mathbf{q}_i = \mathbf{W}_q^T \mathbf{x}_i$

$$\mathbf{h}_i = \sum_{j=1}^n \text{softmax}_j \left(\frac{1}{\sqrt{d}} \mathbf{q}_i^T \mathbf{k}_j \right) \mathbf{v}_j$$

softmax_j — применение softmax к $\left\{ g(\mathbf{x}_i, \mathbf{x}_j) \right\}_{j=1}^n$ независимо по всем i

Scaled dot-product attention

$$\mathbf{h}_i = \sum_{j=1}^n g(\mathbf{x}_i, \mathbf{x}_j) \mathbf{x}_j$$

$$g(\mathbf{x}_i, \mathbf{x}_j) = \text{softmax}_j \left(\frac{1}{\sqrt{d}} \mathbf{x}_i^T \mathbf{x}_j \right) \mathbf{x}_j$$

Self-attention

Ключи: $\mathbf{k}_i = \mathbf{W}_k^T \mathbf{x}_i$

Значения: $\mathbf{v}_i = \mathbf{W}_v^T \mathbf{x}_i$

Запросы: $\mathbf{q}_i = \mathbf{W}_q^T \mathbf{x}_i$

В матричной форме

$$\mathbf{K} = \mathbf{X} \mathbf{W}_k$$

$$\mathbf{V} = \mathbf{X} \mathbf{W}_v$$

$$\mathbf{Q} = \mathbf{X} \mathbf{W}_q$$

$$\mathbf{h}_i = \sum_{j=1}^n \text{softmax}_j \left(\frac{1}{\sqrt{d}} \mathbf{q}_i^T \mathbf{k}_j \right) \mathbf{v}_j$$

$$\mathbf{H} = \text{softmax} \left(\frac{\mathbf{Q} \mathbf{K}^T}{\sqrt{k}} \right) \mathbf{V}$$

softmax_j — применение softmax к $\left\{ g(\mathbf{x}_i, \mathbf{x}_j) \right\}_{j=1}^n$ независимо по всем i

Scaled dot-product attention

В матричной форме

$$\mathbf{K} = \mathbf{X}\mathbf{W}_k$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}_k$$

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_k$$

$$\mathbf{H} = \text{softmax} \left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{k}} \right) \mathbf{V}$$

Scaled dot-product attention

В матричной форме

$$\mathbf{K} = \mathbf{X}\mathbf{W}_k$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}_v$$

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_q$$

$$\mathbf{H} = \text{softmax} \left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{k}} \right) \mathbf{V}$$

```
class ScaledDotProductAttention(nn.Module):
    def __init__(self, input_dim: int, hidden_dim: int):
        super().__init__()
        self.W_q = nn.Linear(input_dim, hidden_dim)
        self.W_k = nn.Linear(input_dim, hidden_dim)
        self.W_v = nn.Linear(input_dim, input_dim)

    def forward(self, x: Tensor) -> Tensor:
        q = self.W_q(x)
        k = self.W_k(x)
        v = self.W_v(x)
        D_k = k.shape[1]
        scores = torch.bmm(q, k.permute(0, 2, 1)) / D_k**0.5
        attention_weights = scores.softmax(-1)
        return torch.bmm(attention_weights, v)
```

Multi-head attention (MHA)

Голова внимания:

$$\mathbf{K}_i = \mathbf{X} \mathbf{W}_{k,i}$$

$$\mathbf{V}_i = \mathbf{X} \mathbf{W}_{v,i}$$

$$\mathbf{Q}_i = \mathbf{X} \mathbf{W}_{q,i}$$

$$\text{SA}_i(\mathbf{X}) = \text{softmax} \left(\frac{\mathbf{Q}_i \mathbf{K}_i^T}{\sqrt{k}} \right) \mathbf{V}_i$$

```
class ScaledDotProductAttention(nn.Module):
    def __init__(self, input_dim: int, hidden_dim: int):
        super().__init__()
        self.W_q = nn.Linear(input_dim, hidden_dim)
        self.W_k = nn.Linear(input_dim, hidden_dim)
        self.W_v = nn.Linear(input_dim, input_dim)

    def forward(self, x: Tensor) -> Tensor:
        q = self.W_q(x)
        k = self.W_k(x)
        v = self.W_v(x)
        D_k = k.shape[1]
        scores = torch.bmm(q, k.permute(0, 2, 1)) / D_k**0.5
        attention_weights = scores.softmax(-1)
        return torch.bmm(attention_weights, v)
```

Multi-head attention (MHA)

Голова внимания:

$$\mathbf{K}_i = \mathbf{X} \mathbf{W}_{k,i}$$

$$\mathbf{V}_i = \mathbf{X} \mathbf{W}_{v,i}$$

$$\mathbf{Q}_i = \mathbf{X} \mathbf{W}_{q,i}$$

$$\text{SA}_i(\mathbf{X}) = \text{softmax}\left(\frac{\mathbf{Q}_i \mathbf{K}_i^T}{\sqrt{k}}\right) \mathbf{V}_i$$

Конкатенация и репроекция:

$$\text{MHA}(\mathbf{X}) = [\text{SA}_1(\mathbf{X}) || \dots || \text{SA}_h(\mathbf{X})] \mathbf{W}_o$$

Отдельная
голова

Выходная
проекция

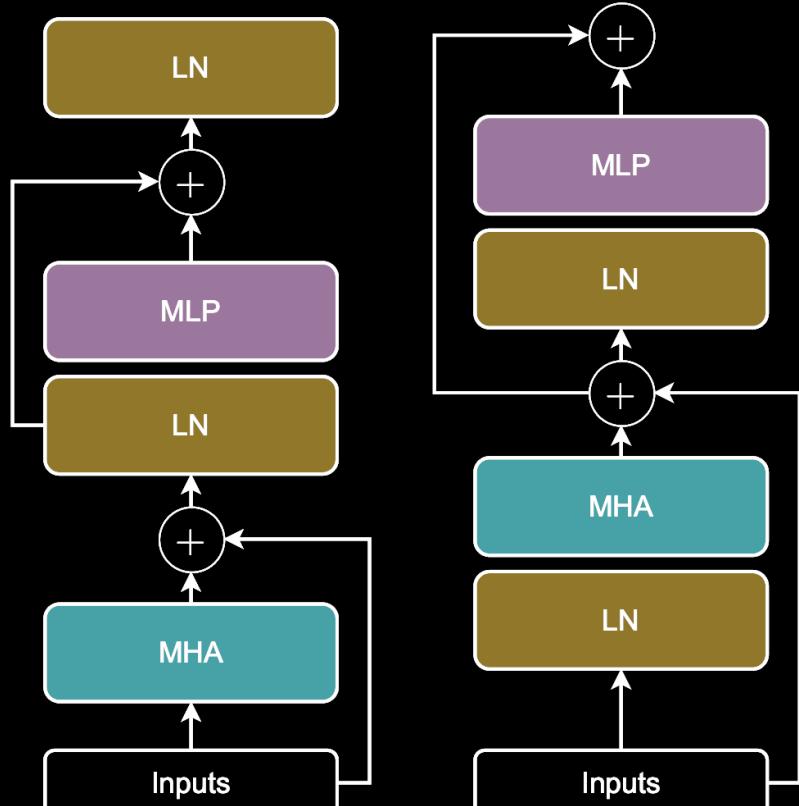
```
class ScaledDotProductAttention(nn.Module):
    def __init__(self, input_dim: int, hidden_dim: int):
        super().__init__()
        self.W_q = nn.Linear(input_dim, hidden_dim)
        self.W_k = nn.Linear(input_dim, hidden_dim)
        self.W_v = nn.Linear(input_dim, input_dim)

    def forward(self, x: Tensor) -> Tensor:
        q = self.W_q(x)
        k = self.W_k(x)
        v = self.W_v(x)
        D_k = k.shape[1]
        scores = torch.bmm(q, k.permute(0, 2, 1)) / D_k**0.5
        attention_weights = scores.softmax(-1)
        return torch.bmm(attention_weights, v)

class MultiHeadAttention(nn.Module):
    def __init__(self, input_dim: int, hidden_dim: int, n_heads: int):
        super().__init__()
        self.heads = nn.ModuleList([
            ScaledDotProductAttention(input_dim, hidden_dim)
            for i in range(n_heads)
        ])
        self.W_o = nn.Linear(n_heads * input_dim, input_dim)

    def forward(self, x: Tensor) -> Tensor:
        h = torch.cat([head(x) for head in self.heads], dim=-1)
        return self.W_o(h)
```

Постпроцессинг токенов: нормализация и MLP



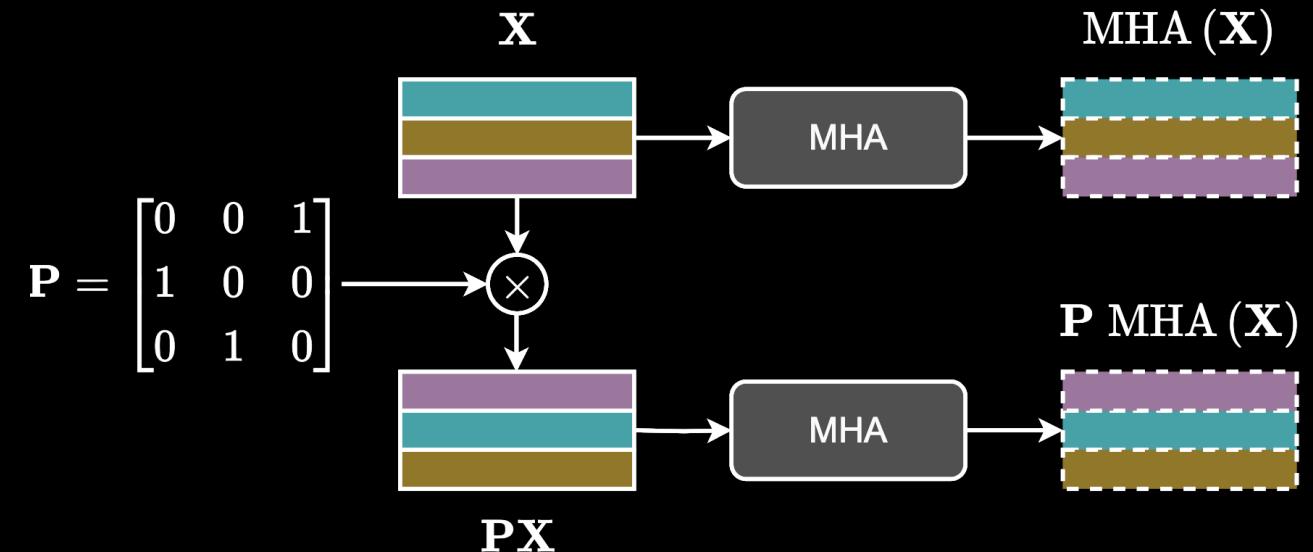
```
class TransformerLayer(nn.Module):
    def __init__(self, input_dim: int, hidden_dim: int, n_heads: int):
        super().__init__()
        self.mha = MultiHeadAttention(input_dim, hidden_dim, n_heads)
        self.norm1 = nn.LayerNorm(input_dim)
        self.mlp = nn.Sequential(
            nn.Linear(input_dim, input_dim * 4),
            nn.ReLU(inplace=True),
            nn.Linear(input_dim * 4, input_dim),
        )
        self.norm2 = nn.LayerNorm(input_dim)

    def forward(self, x: Tensor) -> Tensor:
        z = self.norm1(self.mha(x) + x)
        return self.norm2(self.mlp(z) + z)
```

Позиционные эмбеддинги

Permutation equivariance

$$\text{MHA}(\mathbf{P}\mathbf{X}) = \mathbf{P} \cdot \text{MHA}(\mathbf{X})$$



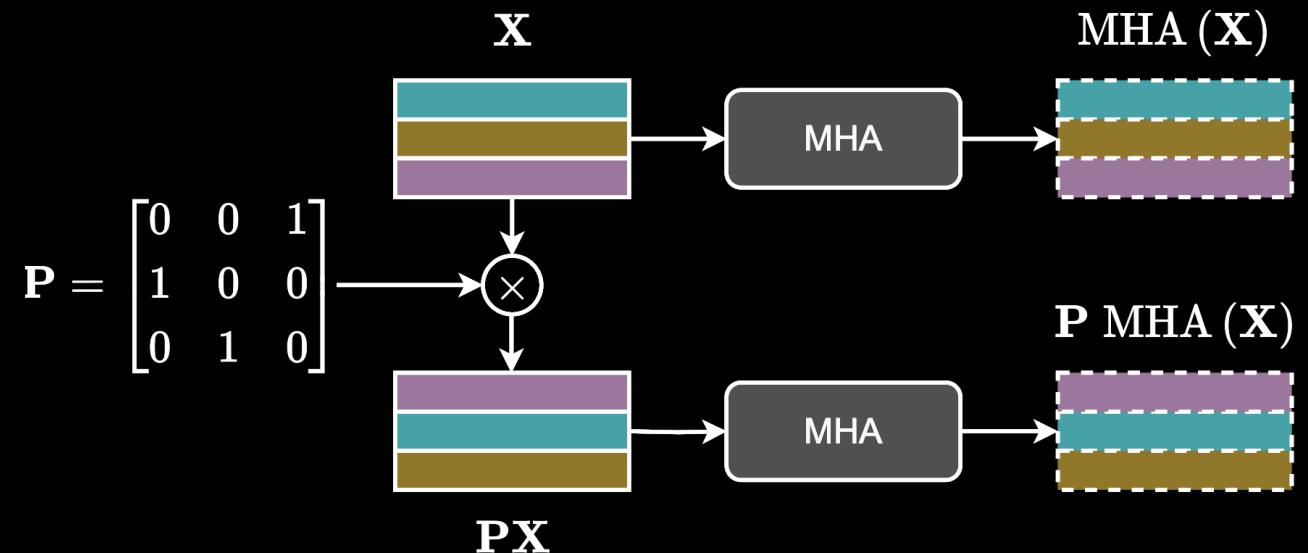
Позиционные эмбеддинги

Permutation equivariance

$$\text{MHA}(\mathbf{P}\mathbf{X}) = \mathbf{P} \cdot \text{MHA}(\mathbf{X})$$

Абсолютные позиционные эмбеддинги

$$\text{MHA}(\mathbf{P}\mathbf{X} + \mathbf{S}) \neq \text{MHA}(\mathbf{X} + \mathbf{S})$$



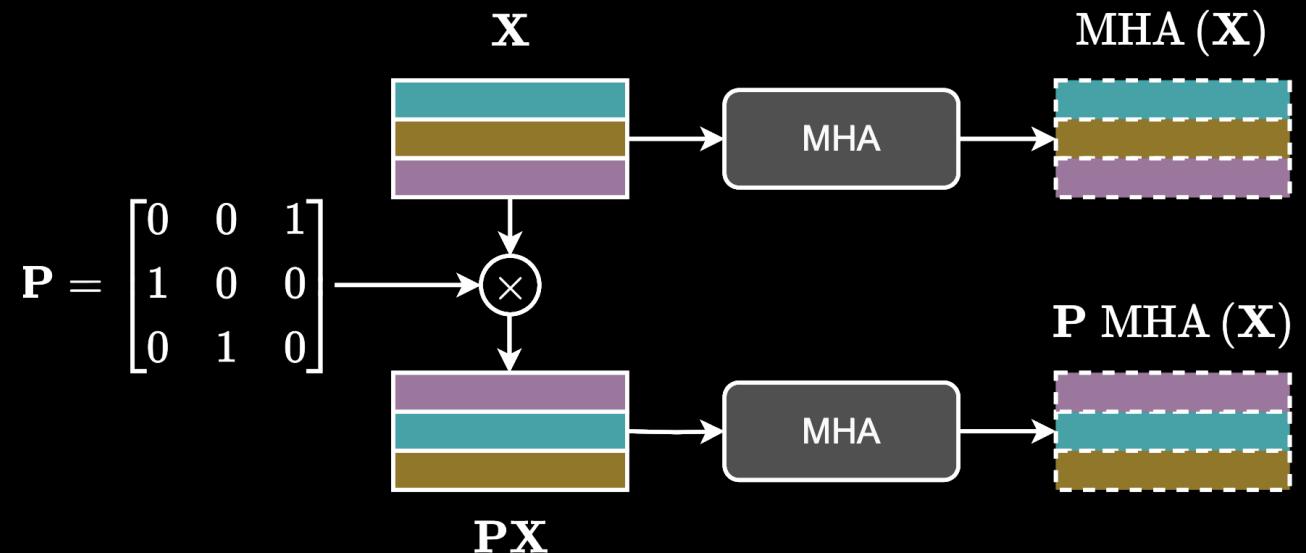
Позиционные эмбеддинги

Permutation equivariance

$$\text{MHA}(\mathbf{P}\mathbf{X}) = \mathbf{P} \cdot \text{MHA}(\mathbf{X})$$

Абсолютные позиционные эмбеддинги

$$\text{MHA}(\mathbf{P}\mathbf{X} + \mathbf{S}) \neq \text{MHA}(\mathbf{X} + \mathbf{S})$$



Относительные позиционные эмбеддинги

$$\mathbf{h}_i = \sum_{j=1}^n \text{softmax}_j(g(\mathbf{q}_i, \mathbf{k}_j))\mathbf{v}_j$$

$$g(\mathbf{x}_i, \mathbf{x}_j) \rightarrow g(\mathbf{x}_i, \mathbf{x}_j, i - j)$$

Классификация

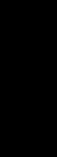
Transformer block



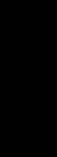
G



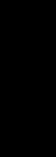
A



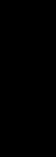
T



T



A



A

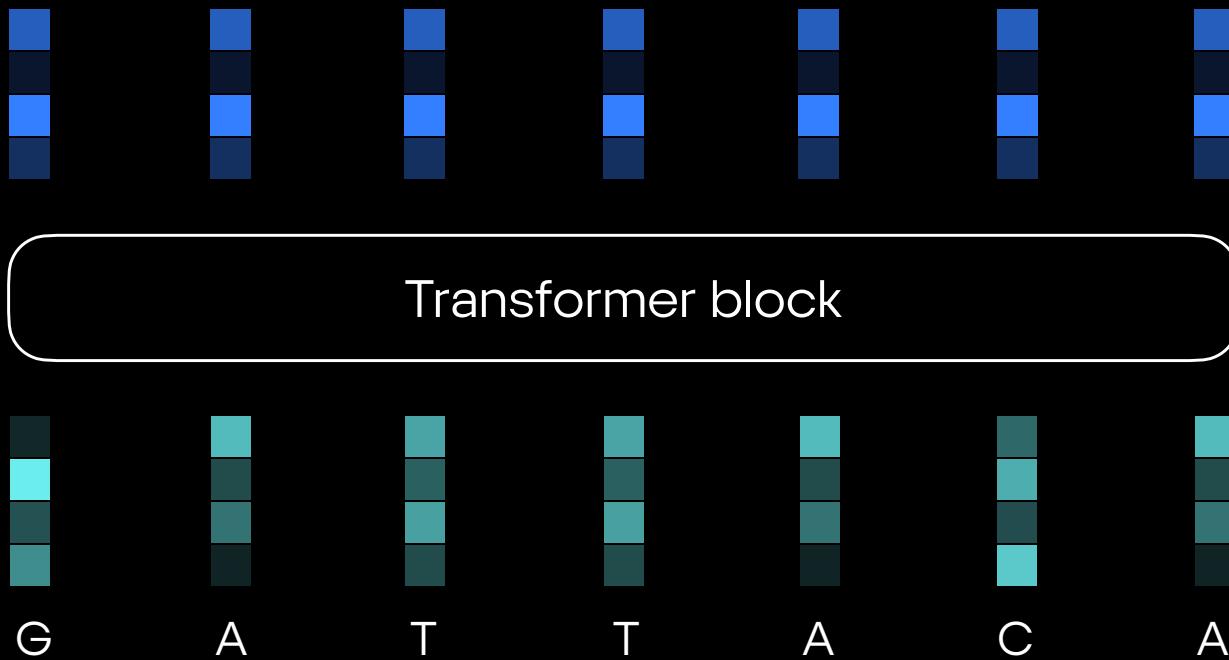


C

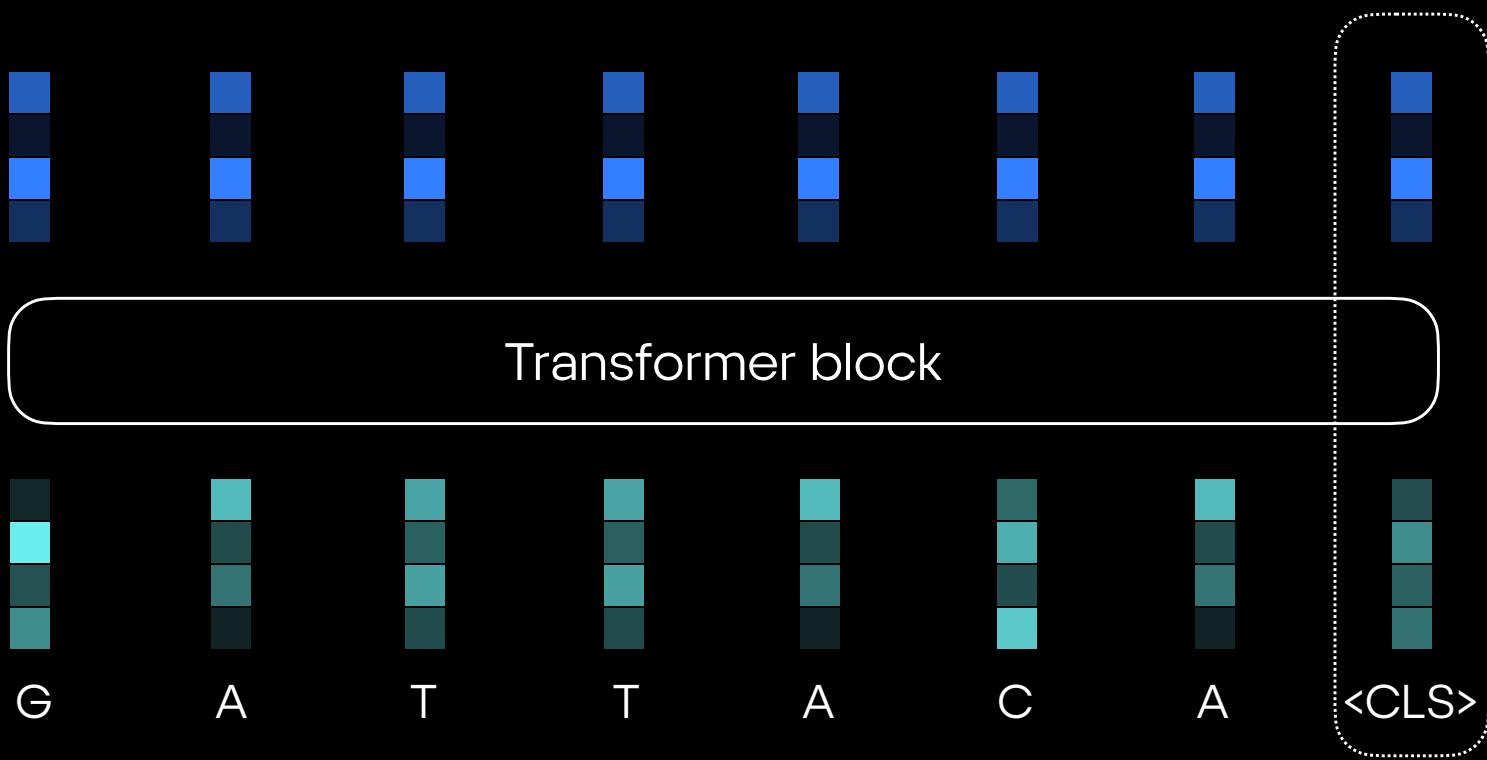


A

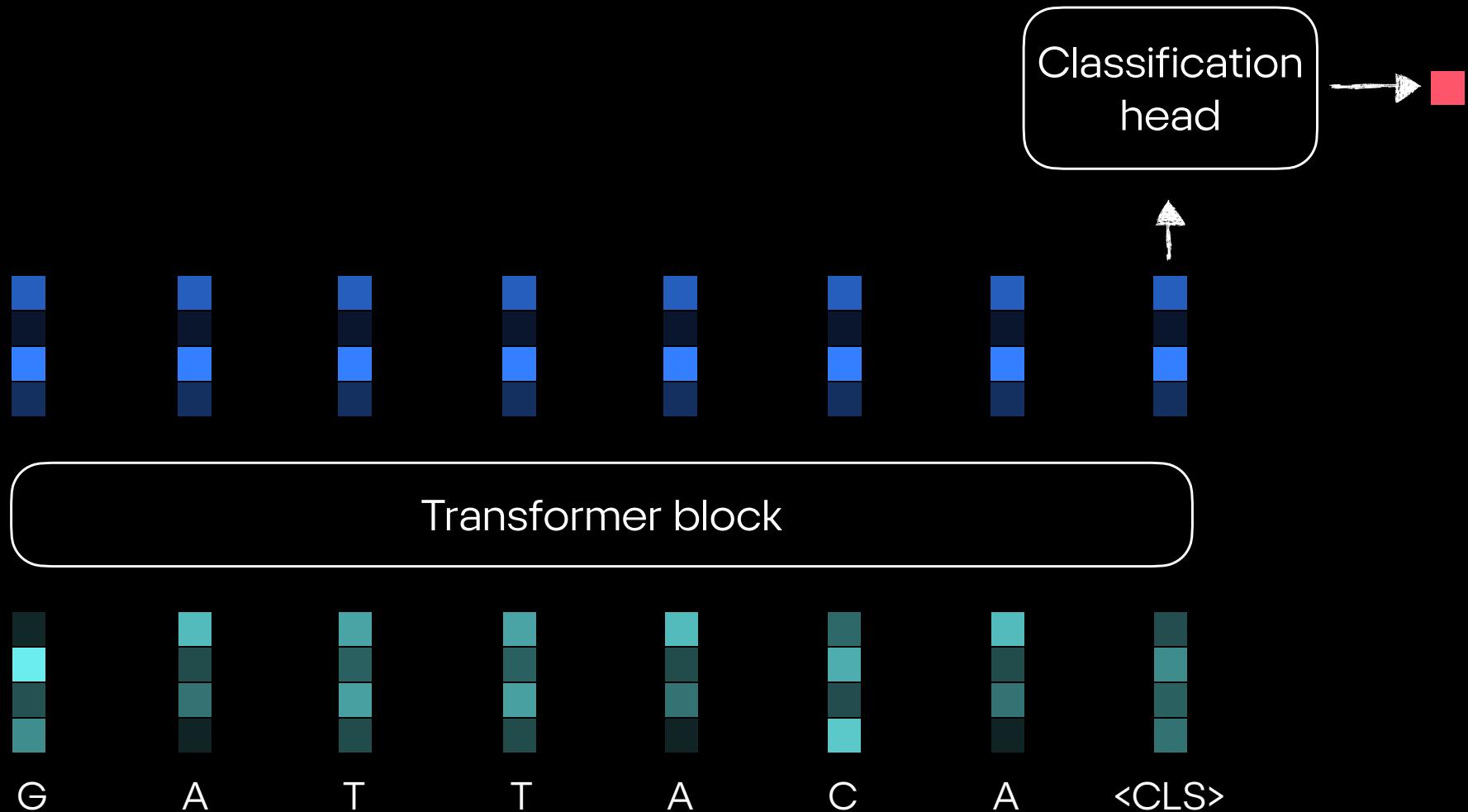
Классификация



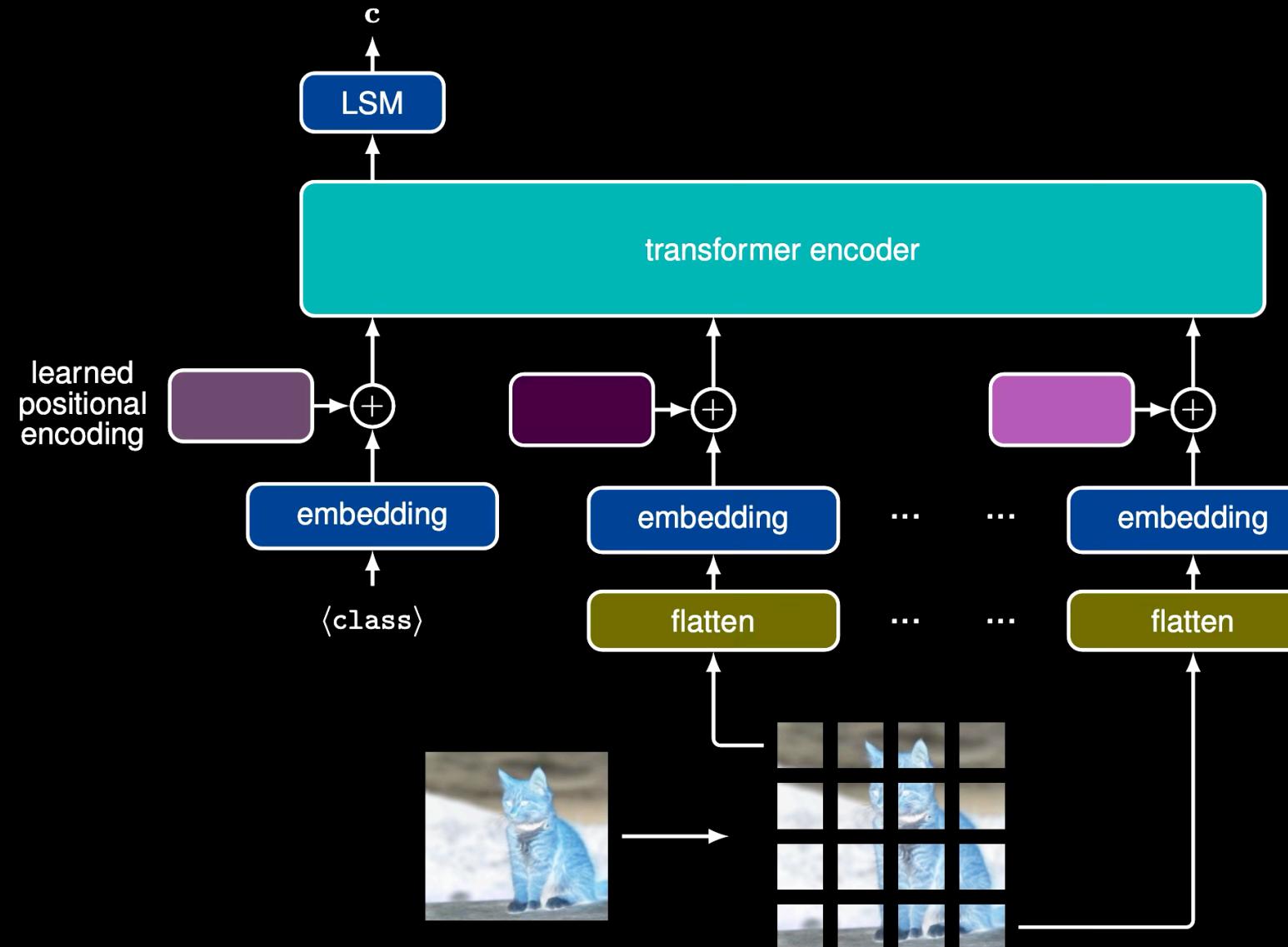
Классификация



Классификация

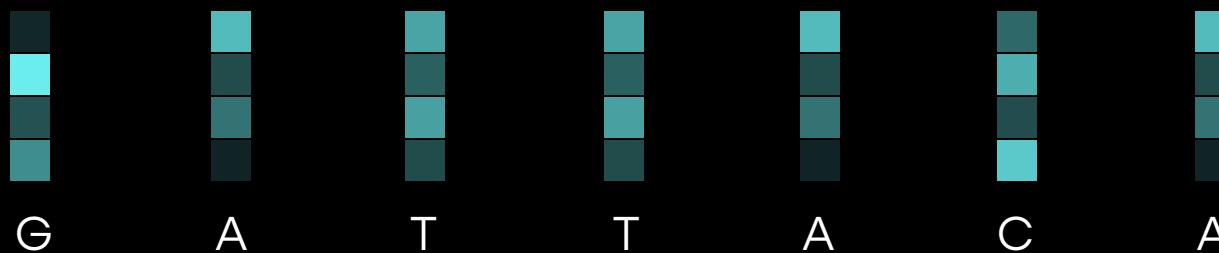


Transformer-encoder для изображений

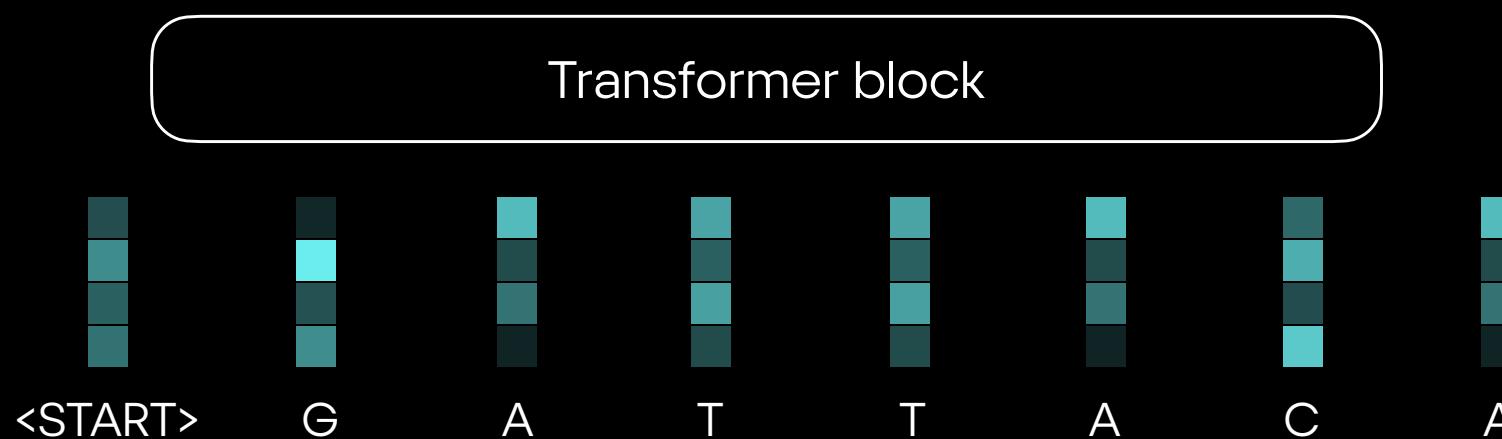


Генерация. Decoder-only transformer

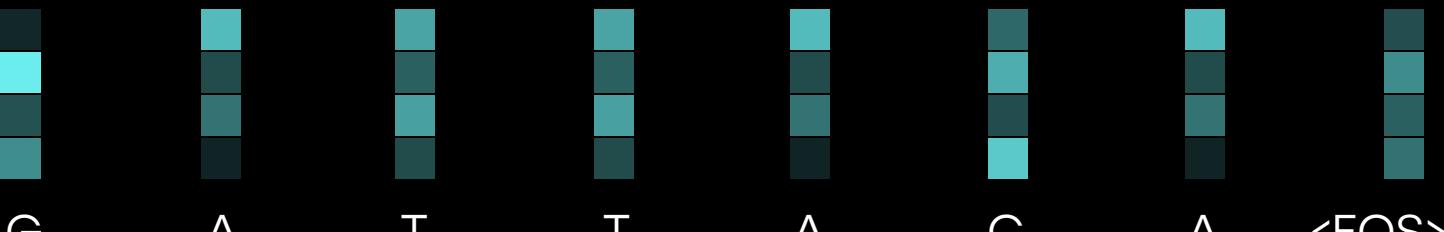
Transformer block



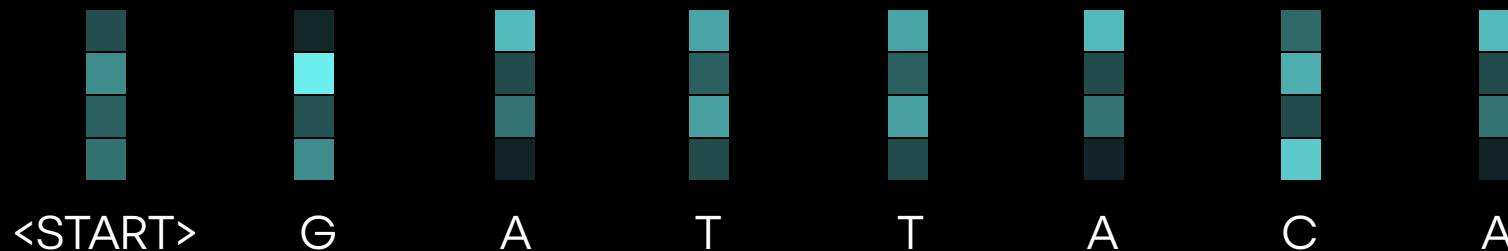
Генерация. Decoder-only transformer



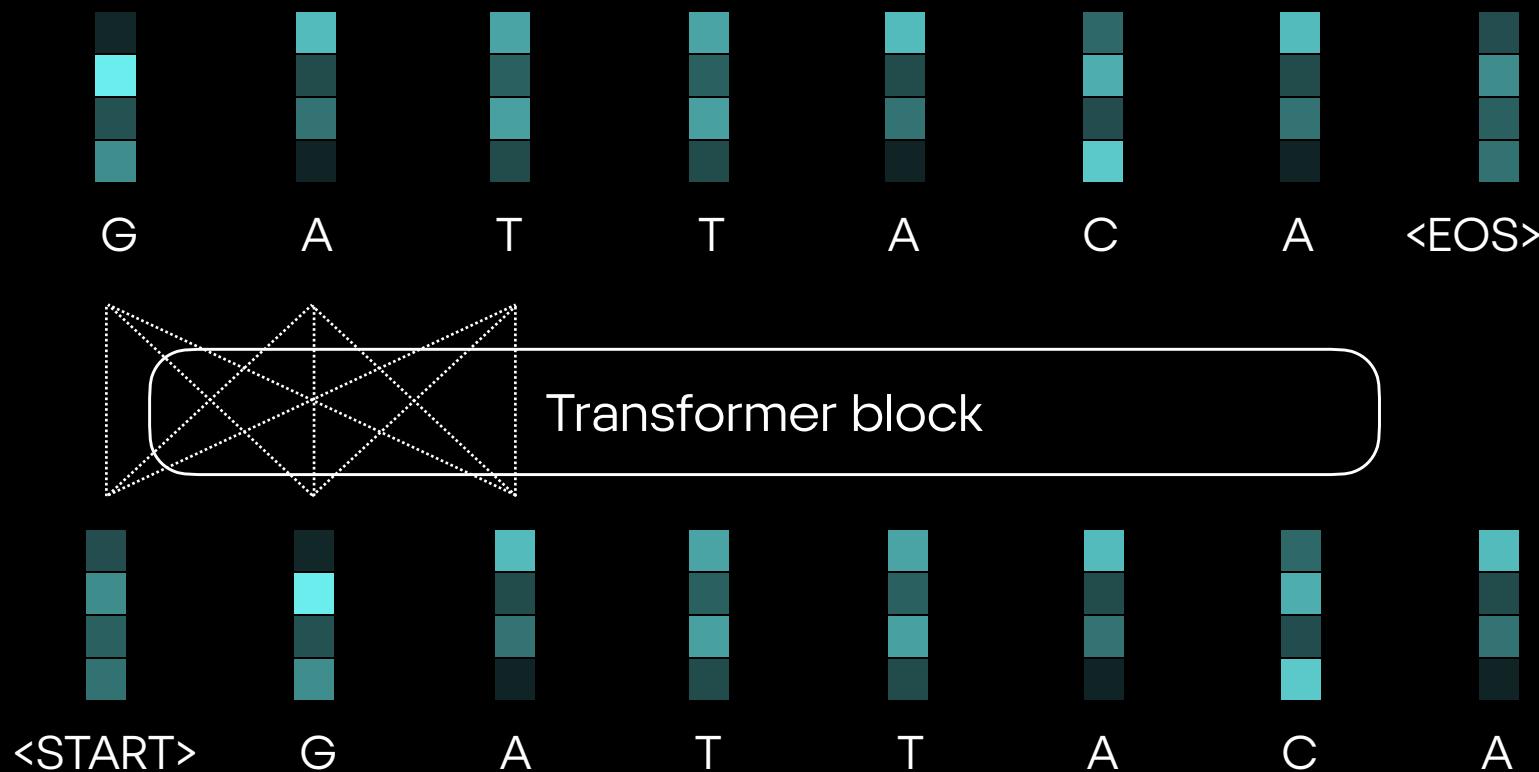
Генерация. Decoder-only transformer



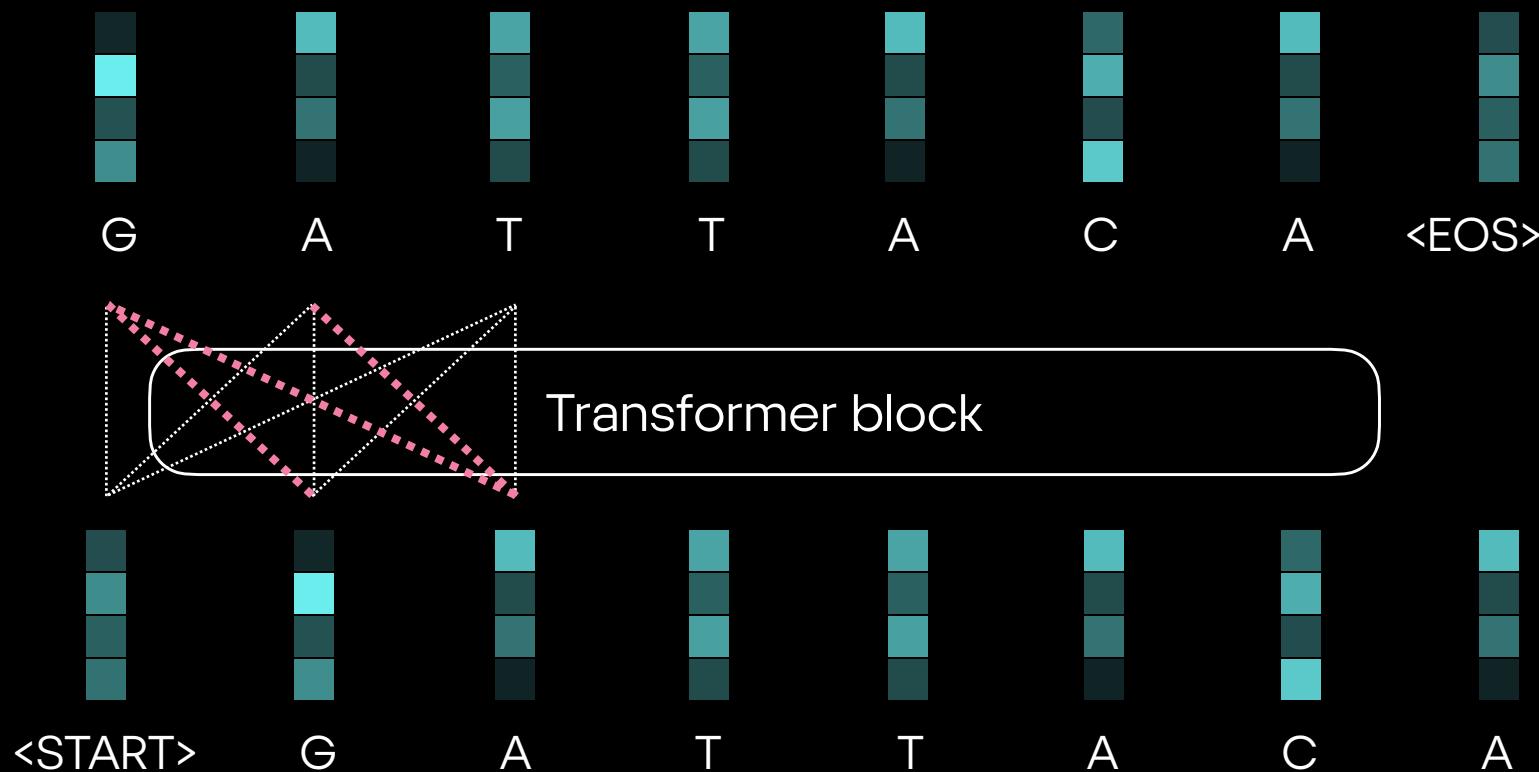
Transformer block



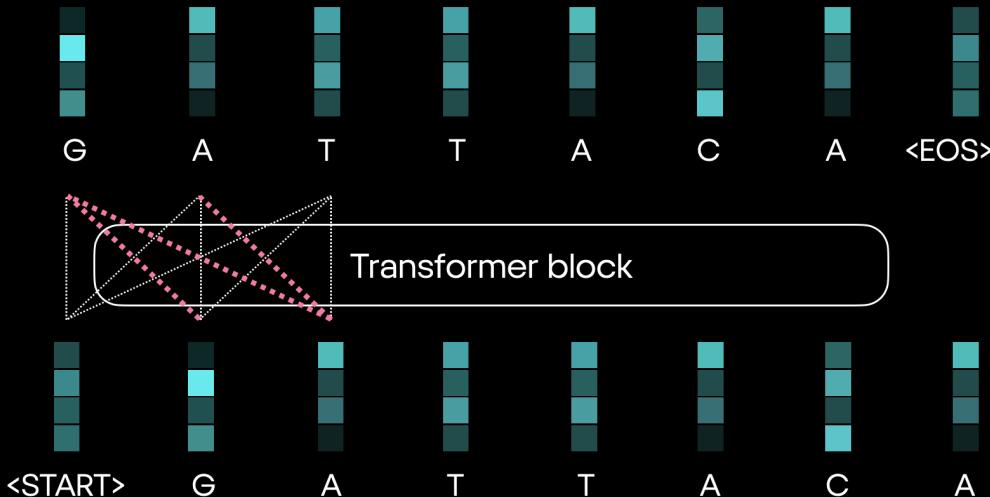
Генерация. Decoder-only transformer



Генерация. Decoder-only transformer



Генерация. Decoder-only transformer

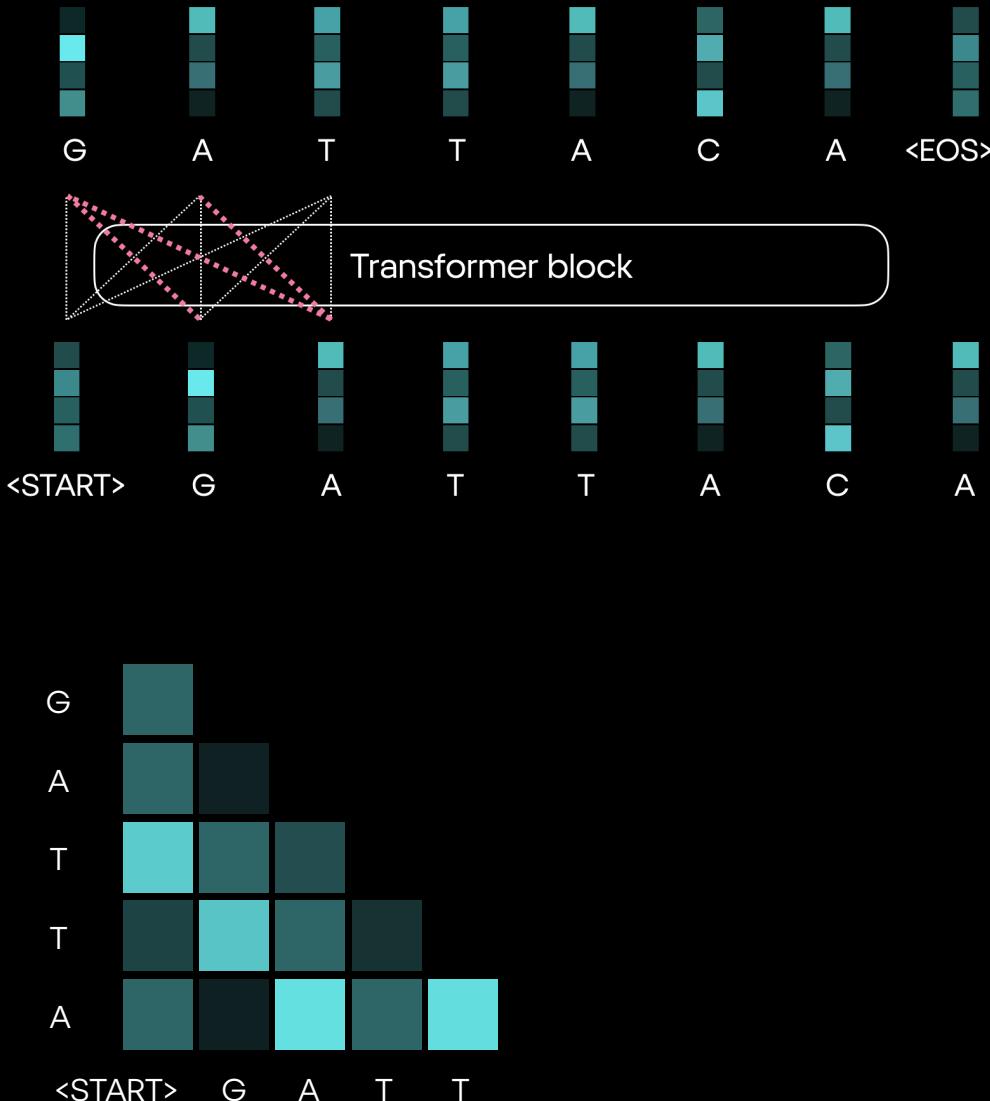


```
class MaskedSelfAttention(nn.Module):
    def __init__(self, input_dim: int, hidden_dim: int) -> None:
        super().__init__()
        self.W_q = nn.Linear(input_dim, hidden_dim)
        self.W_k = nn.Linear(input_dim, hidden_dim)
        self.W_v = nn.Linear(input_dim, input_dim)

    def forward(self, x: Tensor, mask: Tensor) -> Tensor:
        q = self.W_q(x)
        k = self.W_k(x)
        v = self.W_v(x)
        _, _, D_k = k.shape

        attention: Tensor = torch.bmm(q, k.permute(0, 2, 1)) / D_k**0.5
        # apply mask
        attention = attention.masked_fill(mask == 0, float("-inf"))
        attention_weights = attention.softmax(-1)
        return torch.bmm(attention_weights, v)
```

Генерация. Decoder-only transformer



```
class MaskedSelfAttention(nn.Module):
    def __init__(self, input_dim: int, hidden_dim: int) -> None:
        super().__init__()
        self.W_q = nn.Linear(input_dim, hidden_dim)
        self.W_k = nn.Linear(input_dim, hidden_dim)
        self.W_v = nn.Linear(input_dim, input_dim)

    def forward(self, x: Tensor, mask: Tensor) -> Tensor:
        q = self.W_q(x)
        k = self.W_k(x)
        v = self.W_v(x)
        _, _, D_k = k.shape

        attention: Tensor = torch.bmm(q, k.permute(0, 2, 1)) / D_k**0.5
        # apply mask
        attention = attention.masked_fill(mask == 0, float("-inf"))
        attention_weights = attention.softmax(-1)
        return torch.bmm(attention_weights, v)
```

Self-attention → Cross-attention

$$\mathbf{K} = \mathbf{X} \mathbf{W}_k$$

$$\mathbf{V} = \mathbf{X} \mathbf{W}_v$$

$$\mathbf{Q} = \mathbf{X} \mathbf{W}_q$$

$$\mathbf{H} = \text{softmax} \left(\frac{\mathbf{Q} \mathbf{K}^T \odot \mathbf{M}}{\sqrt{k}} \right) \mathbf{V}$$

Self-attention → Cross-attention

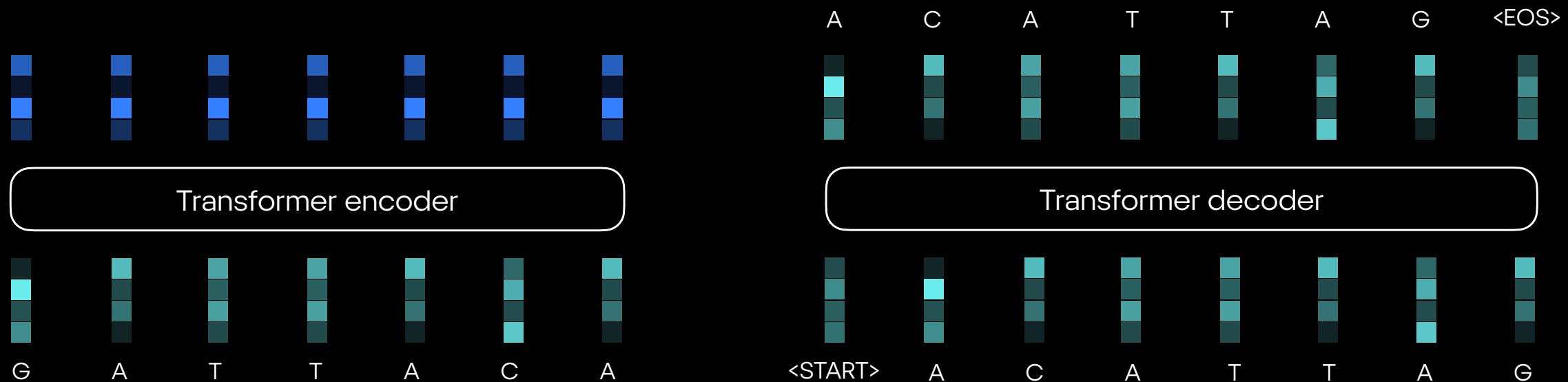
$$\mathbf{K} = \mathbf{XW}_k \quad \mathbf{K} = \mathbf{YW}_k$$

$$\mathbf{V} = \mathbf{XW}_k \quad \mathbf{V} = \mathbf{YW}_k$$

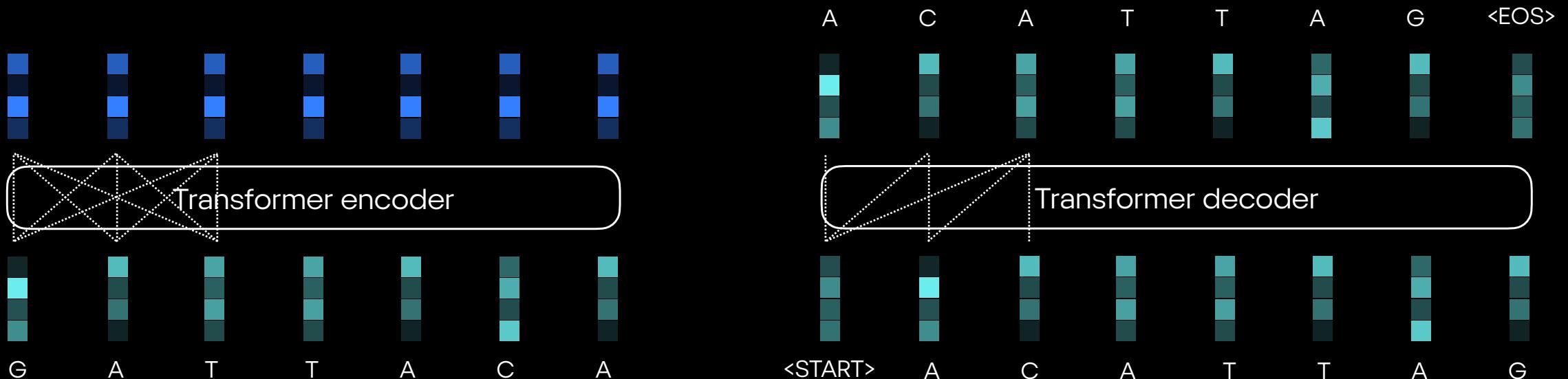
$$\mathbf{Q} = \mathbf{XW}_k \quad \mathbf{Q} = \mathbf{XW}_k$$

$$\mathbf{H} = \text{softmax} \left(\frac{\mathbf{QK}^T \odot \mathbf{M}}{\sqrt{k}} \right) \mathbf{V}$$

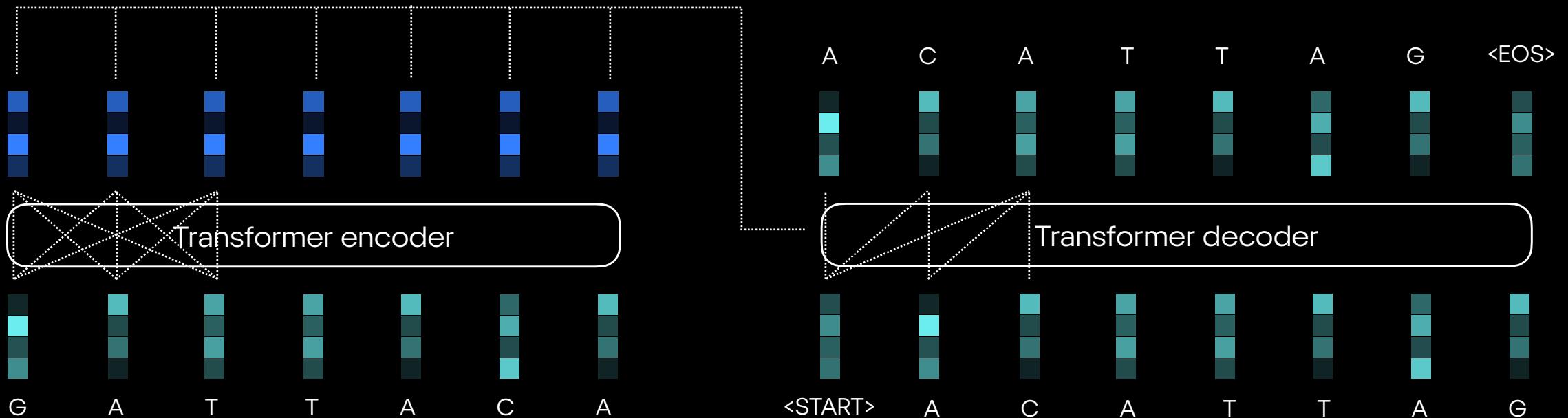
Encoder-decoder



Encoder-decoder



Encoder-decoder



Encoder-decoder

```
class MaskedAttention(nn.Module):
    def __init__(self, input_dim: int, hidden_dim: int) -> None:
        super().__init__()
        self.W_q = nn.Linear(input_dim, hidden_dim)
        self.W_k = nn.Linear(input_dim, hidden_dim)
        self.W_v = nn.Linear(input_dim, input_dim)

    def forward(
        self, q: Tensor,
        k: Tensor,
        v: Tensor,
        mask: Tensor
    ) -> Tensor:
        q = self.W_q(q)
        k = self.W_k(k)
        v = self.W_v(v)
        _, _, D_k = k.shape
        attention: Tensor = torch.bmm(
            q, k.permute(0, 2, 1)
        ) / D_k**0.5
        # apply mask
        attention = attention.masked_fill(
            mask == 0, float("-inf")
        )
        attention_weights = attention.softmax(-1)
        return torch.bmm(attention_weights, v)

class DecoderLayer(nn.Module):
    def __init__(self, input_dim: int, hidden_dim: int) -> None:
        super().__init__()
        self.self_attention = MaskedAttention(input_dim, hidden_dim)
        self.cross_attention = MaskedAttention(input_dim, hidden_dim)
        self.mlp = nn.Sequential(
            nn.Linear(input_dim, input_dim),
            nn.ReLU(inplace=True),
            nn.Linear(input_dim, input_dim),
        )
        self.norm1 = nn.LayerNorm(input_dim)
        self.norm2 = nn.LayerNorm(input_dim)
        self.norm3 = nn.LayerNorm(input_dim)

    def forward(
        self, x: Tensor,
        x_mask: Tensor,
        z: Tensor,
        z_mask: Tensor
    ) -> Tensor:
        # x – embedded targets
        # z – embedded inputs
        # 1. masked self-attention
        h = self.norm1(self.self_attention.forward(
            q=x, k=x, v=x, mask=x_mask) + x
        )
        # 2. cross-attention
        h = self.norm2(self.cross_attention.forward(
            q=h, k=z, v=z, mask=z_mask) + h
        )
        # 3. MLP
        return self.norm3(self.mlp(h) + h)
```

В библиотеке transformers всё проще

```
from transformers import AutoModelForCausalLM, AutoTokenizer  
  
checkpoint = "HuggingFaceTB/SmollM2-135M"  
tokenizer = AutoTokenizer.from_pretrained(checkpoint)  
model = AutoModelForCausalLM.from_pretrained(checkpoint)  
inputs = tokenizer.encode("Gravity is", return_tensors="pt")  
outputs = model.generate(inputs)  
print(tokenizer.decode(outputs[0]))
```

Обучение представлений данных



QVQLVE...
DIQLTQ...
TVPPMV...

Последовательности

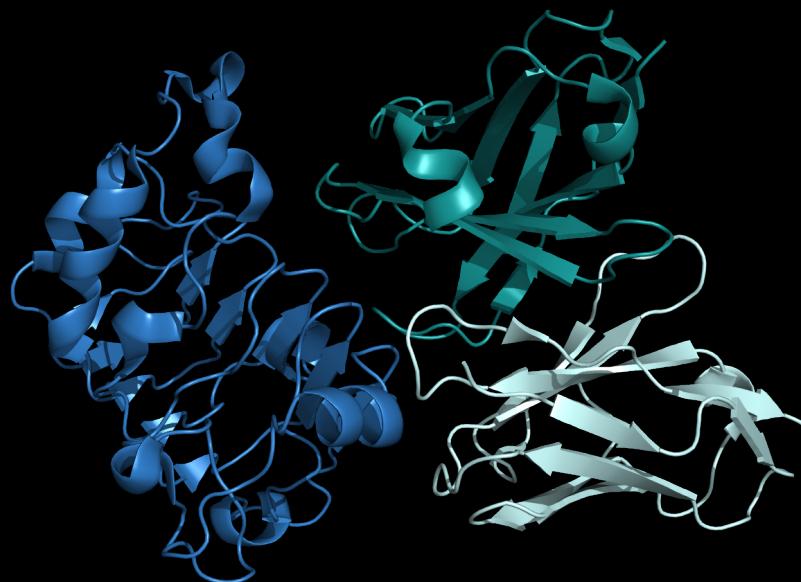


Изображения

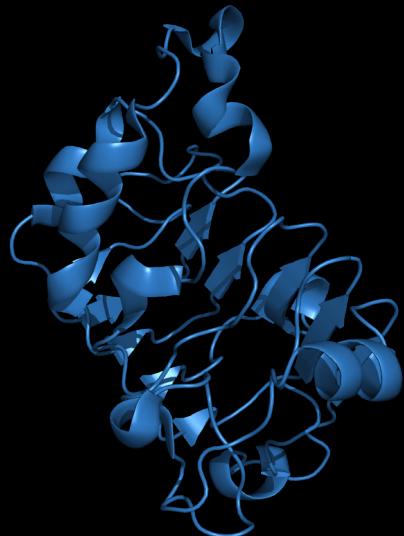
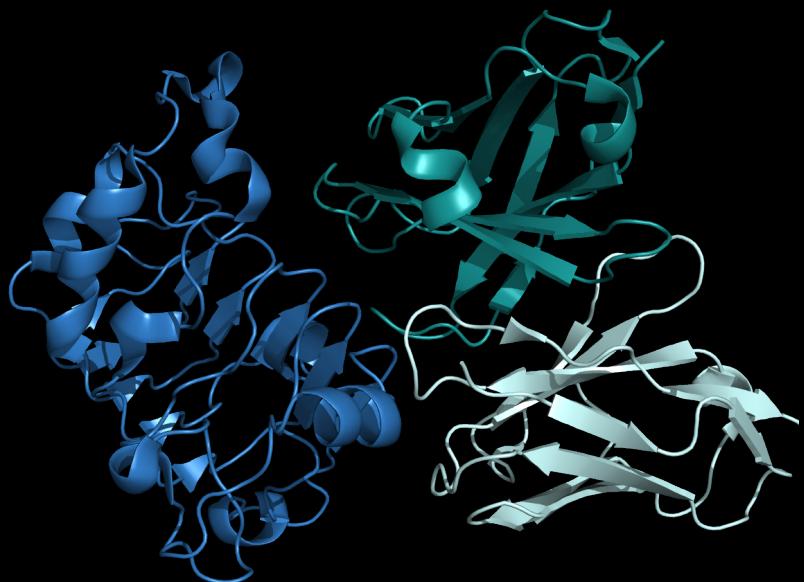
Обучение представлений данных



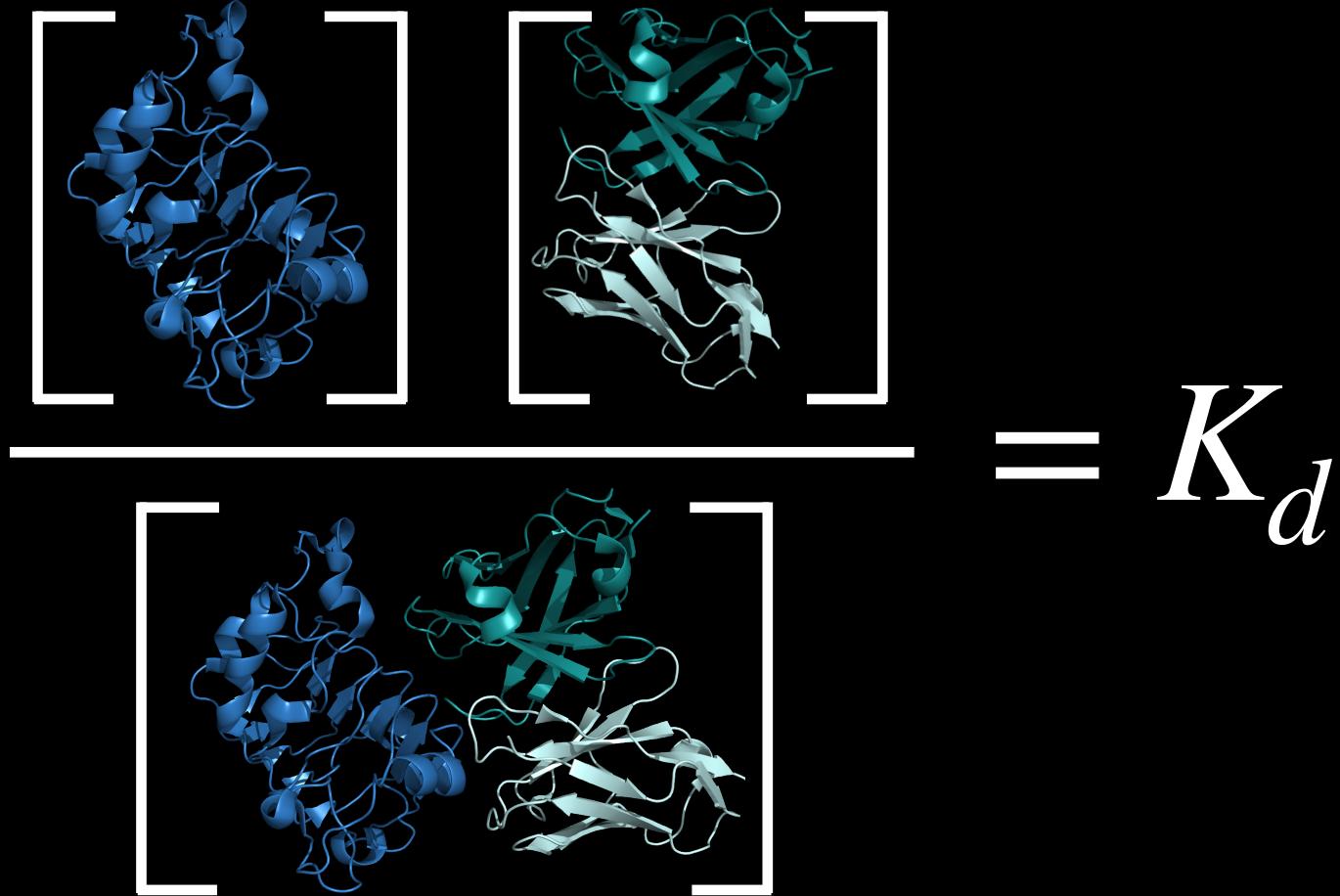
Мера аффинности: константа диссоциации



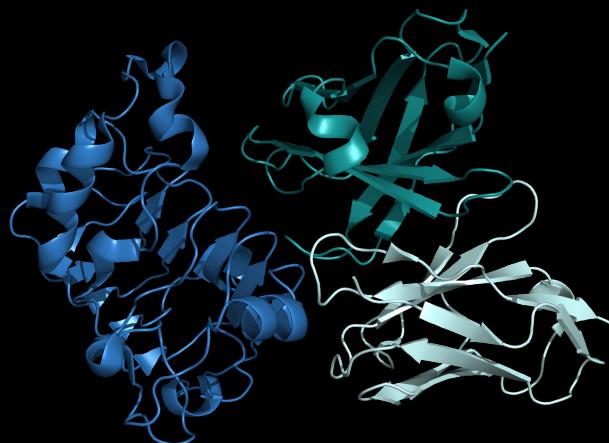
Мера аффинности: константа диссоциации



Мера аффинности: константа диссоциации

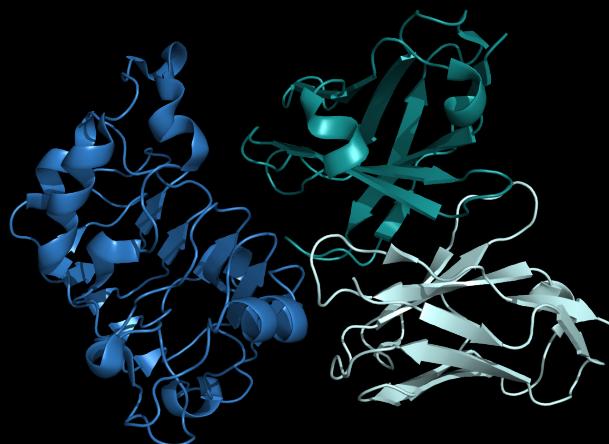


Предсказание аффинности по структуре комплекса



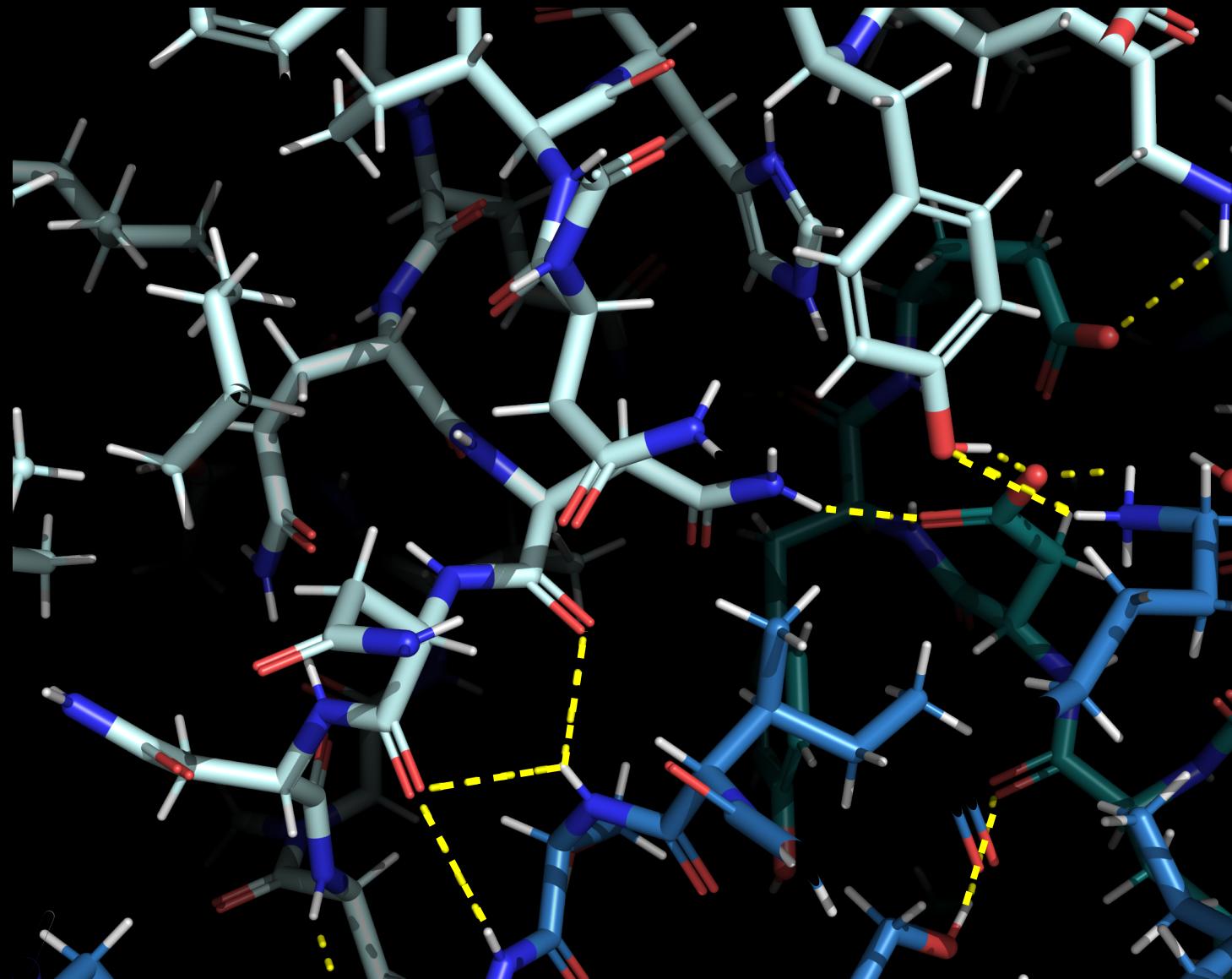
$$K_d$$

Предсказание аффинности по структуре комплекса

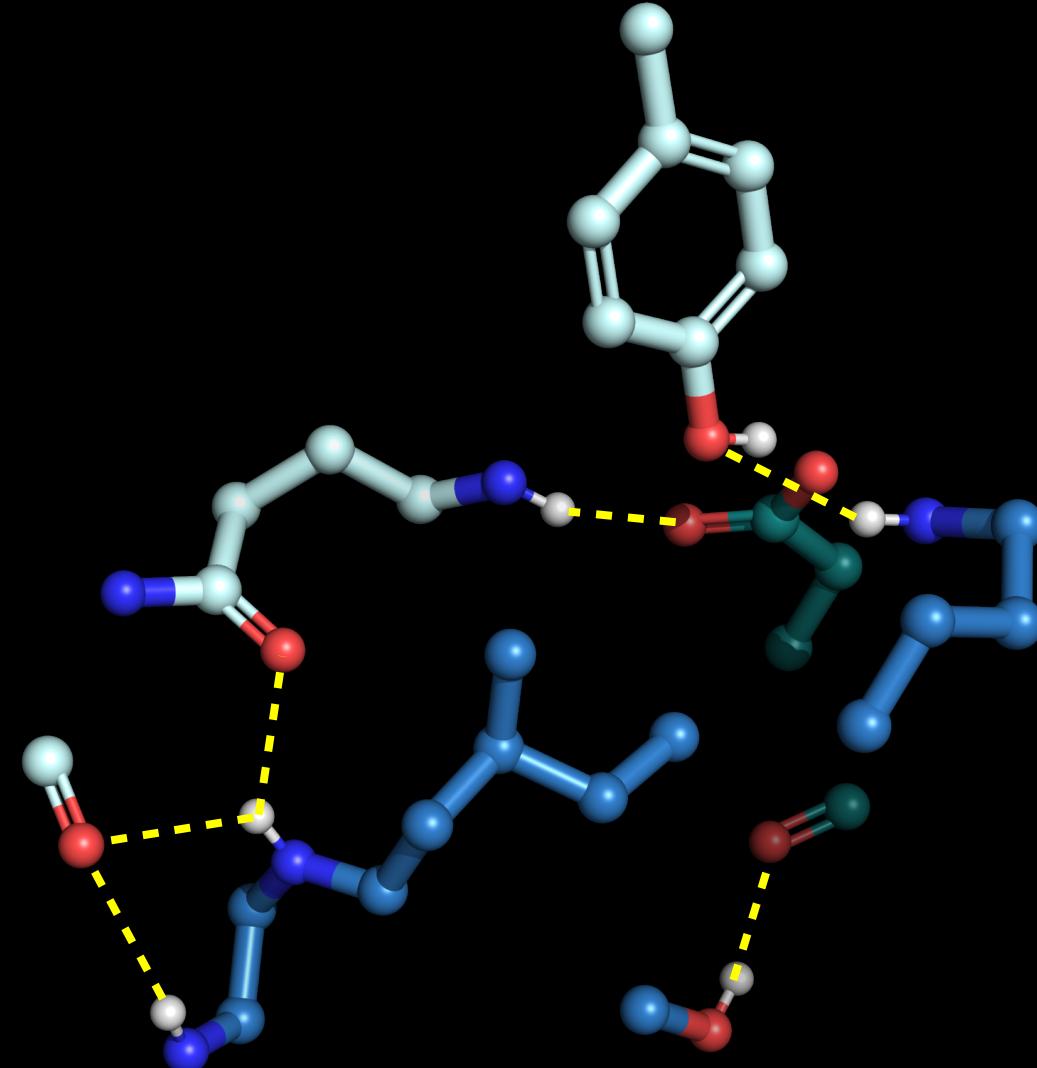


$$\Delta G = RT \ln K_d$$

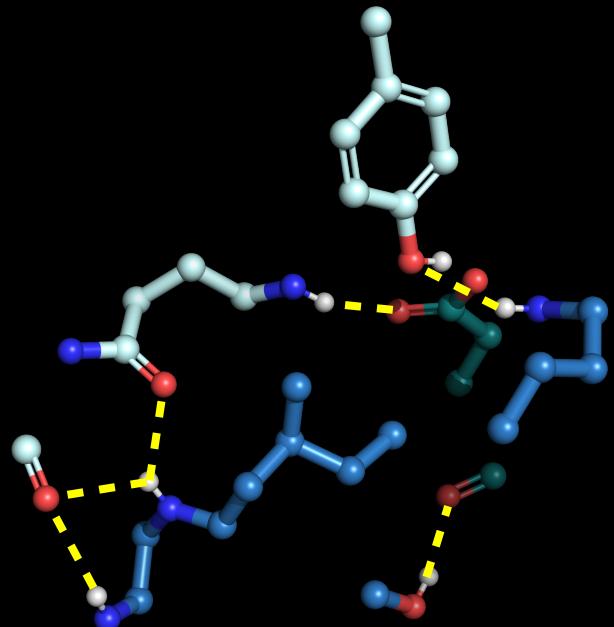
Предсказание аффинности по структуре комплекса



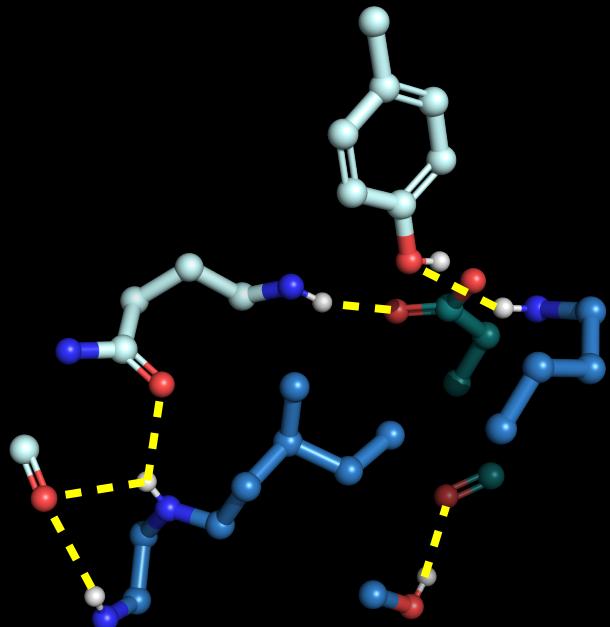
Предсказание аффинности по структуре комплекса



Предсказание аффинности по структуре комплекса

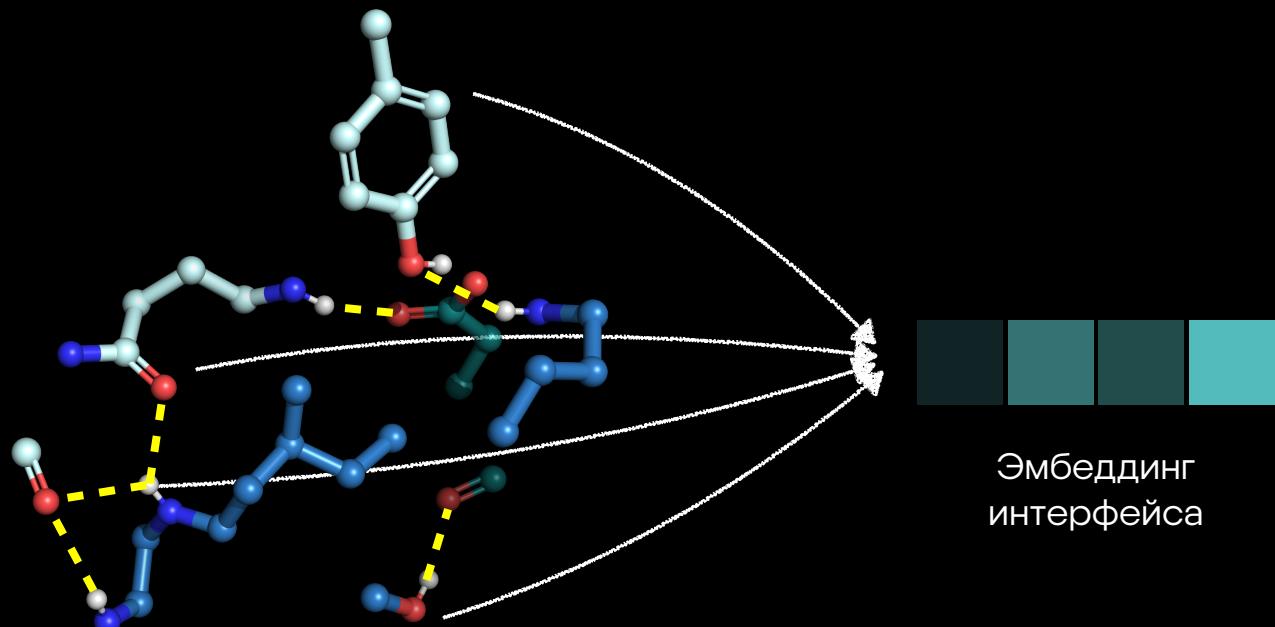


Предсказание аффинности по структуре комплекса

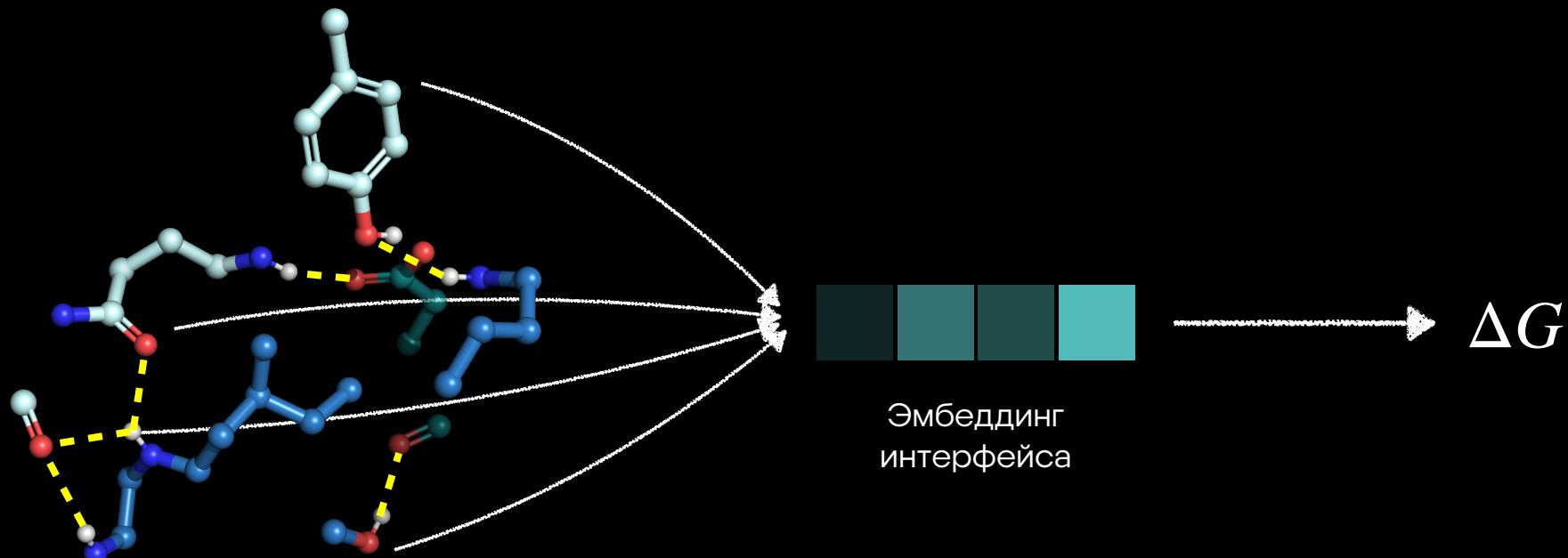


Эмбеддинг
интерфейса

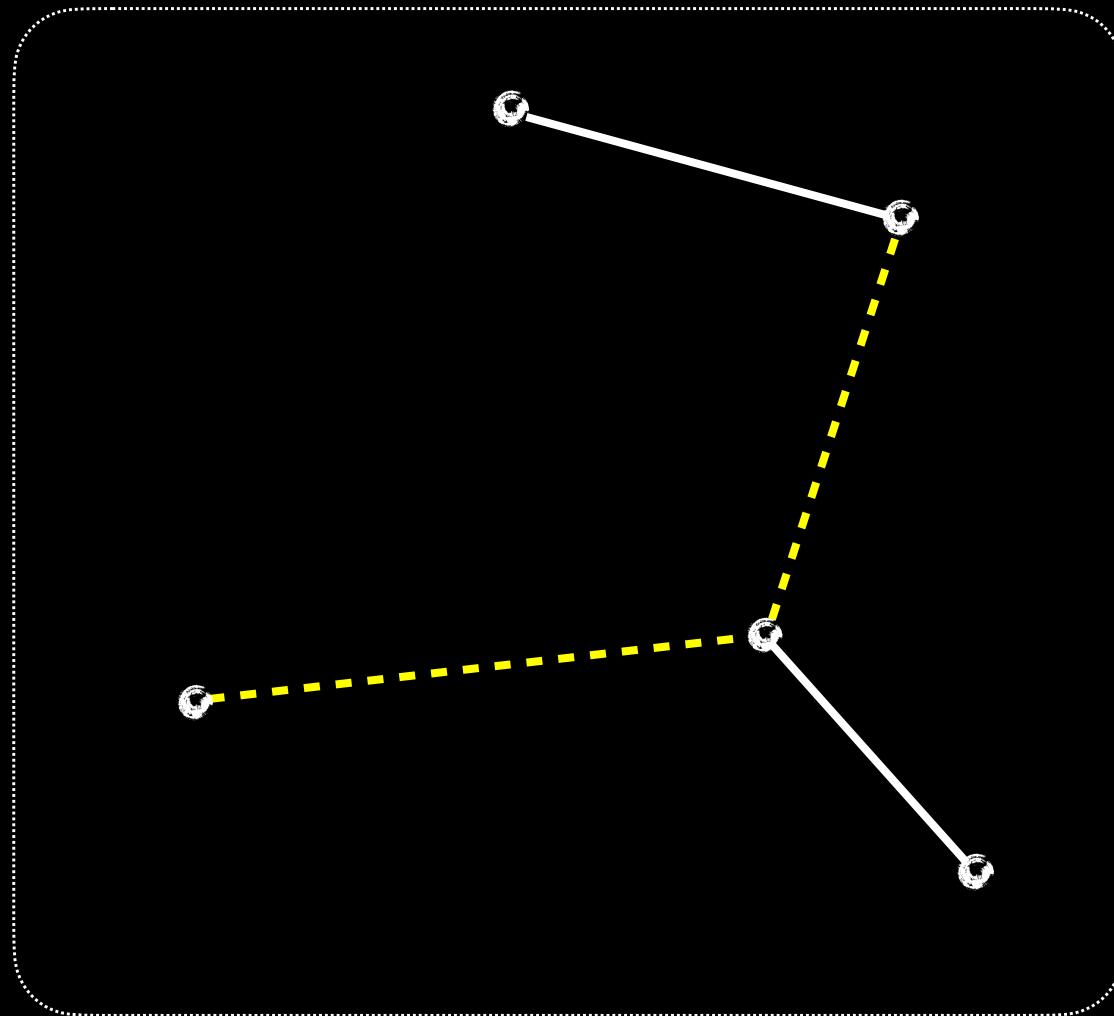
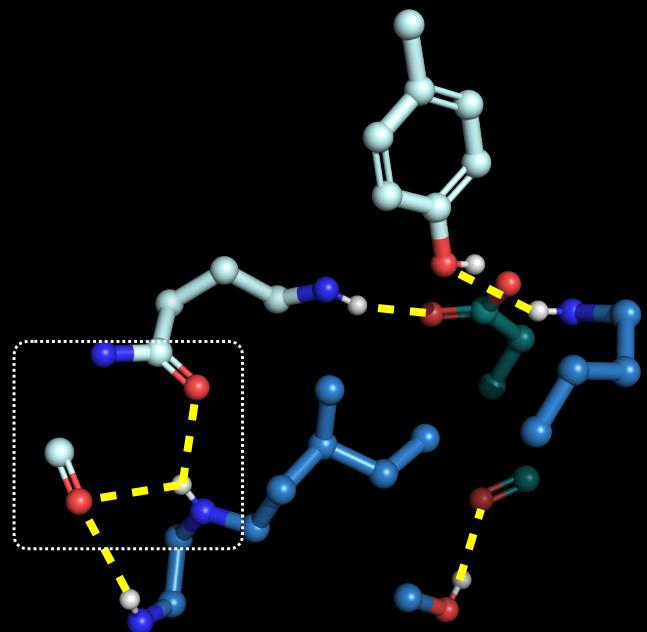
Предсказание аффинности по структуре комплекса



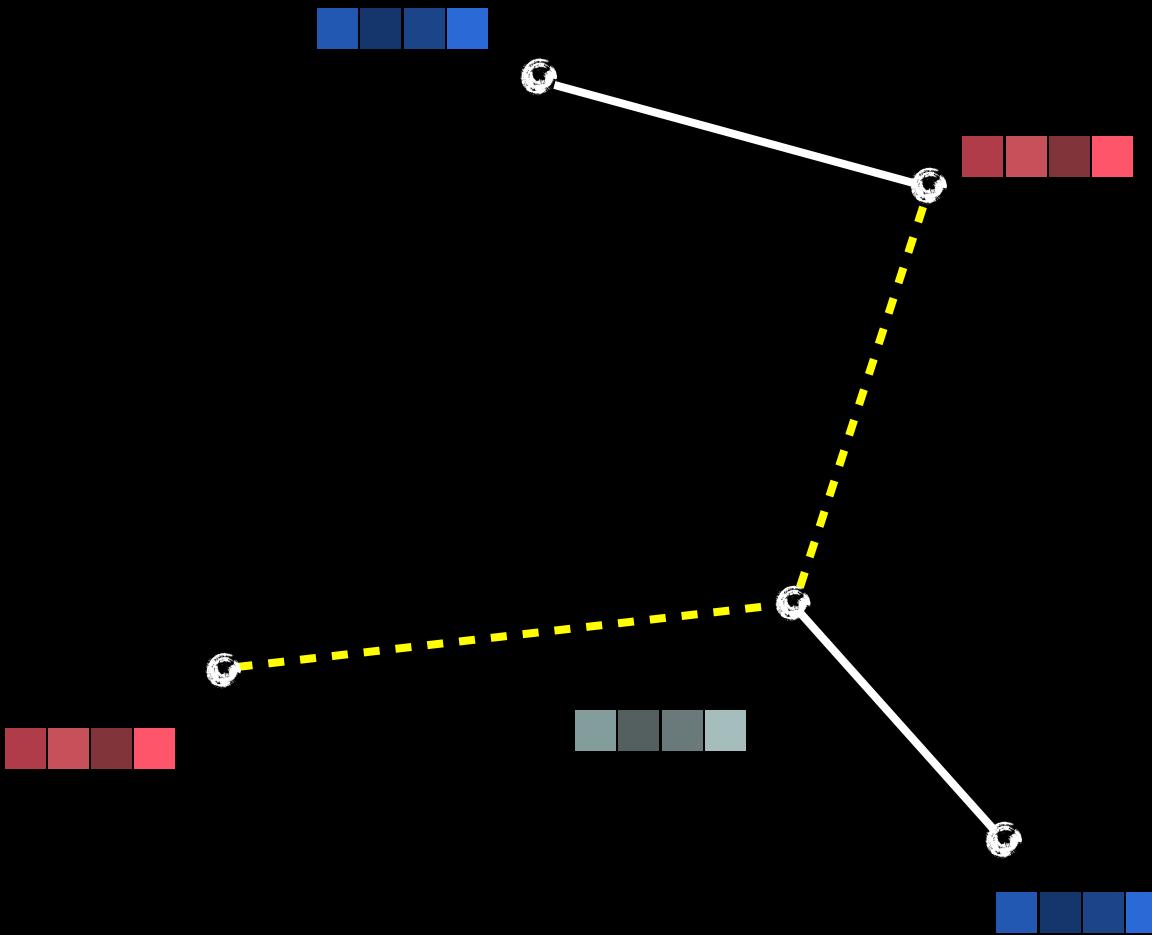
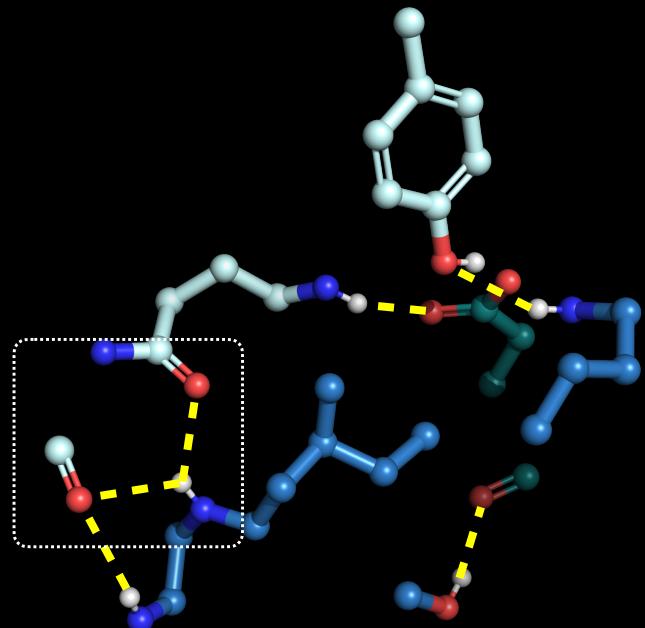
Предсказание аффинности по структуре комплекса



Graph-level regression



Graph-level regression



Graph-level regression

1. Message passing

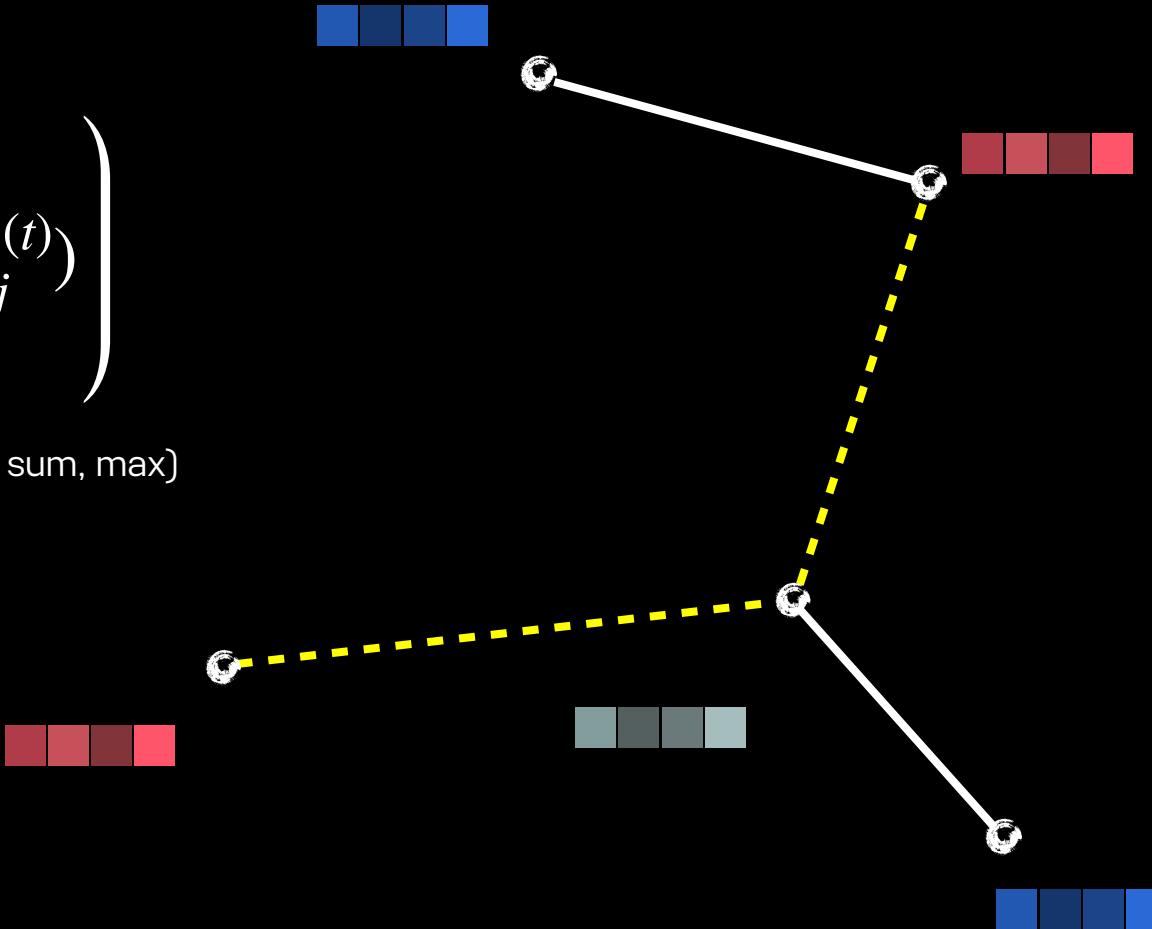
$$h_i^{(t+1)} = \phi \left(h_i^{(t)}, \bigoplus_{j \in \mathcal{N}_i} \psi(h_i^{(t)}, h_j^{(t)}) \right)$$



– функция агрегации (инвариантная; sum, max)

$j \in \mathcal{N}_i$ – соседи вершины i

ψ – функция сообщения



Graph-level regression

1. Message passing

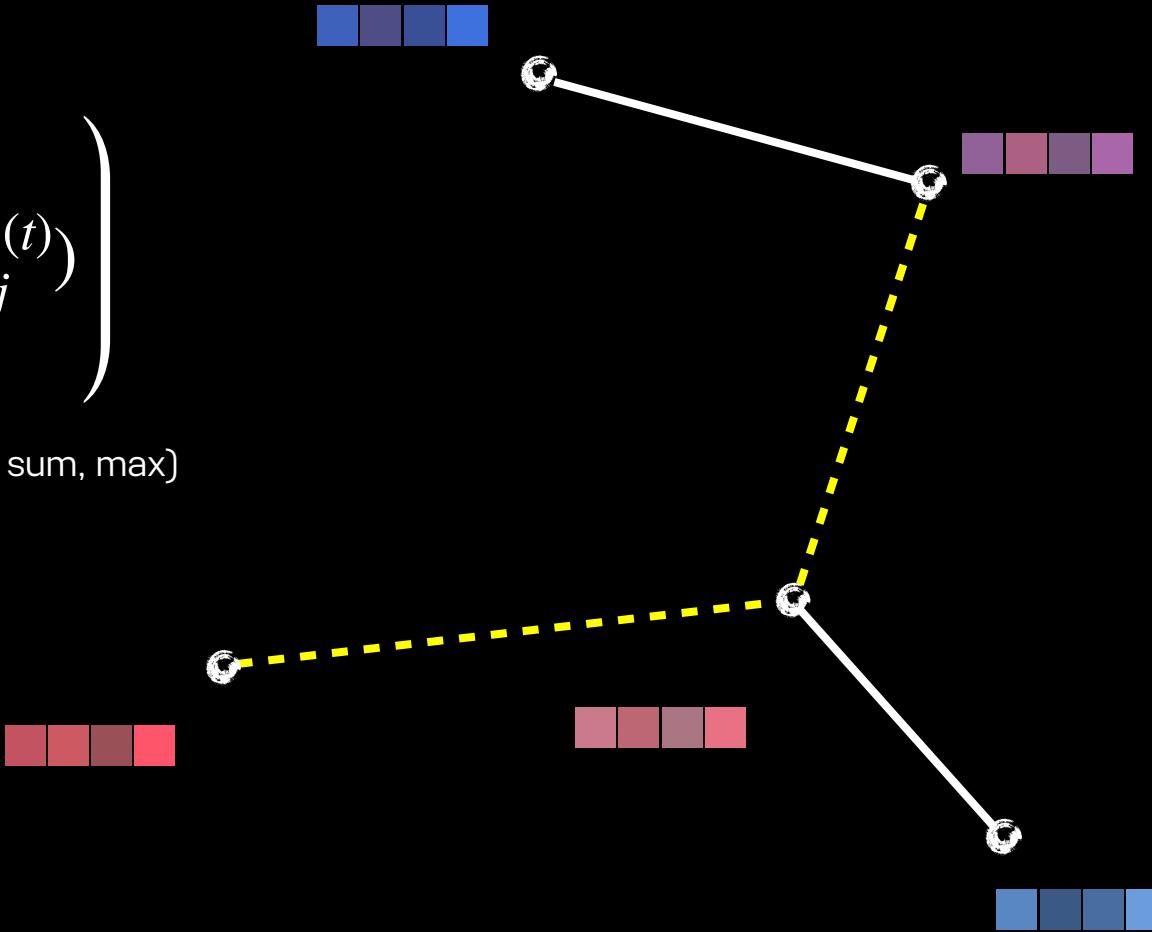
$$h_i^{(t+1)} = \phi \left(h_i^{(t)}, \bigoplus_{j \in \mathcal{N}_i} \psi(h_i^{(t)}, h_j^{(t)}) \right)$$



– функция агрегации (инвариантная; sum, max)

$j \in \mathcal{N}_i$ – соседи вершины i

ψ – функция сообщения



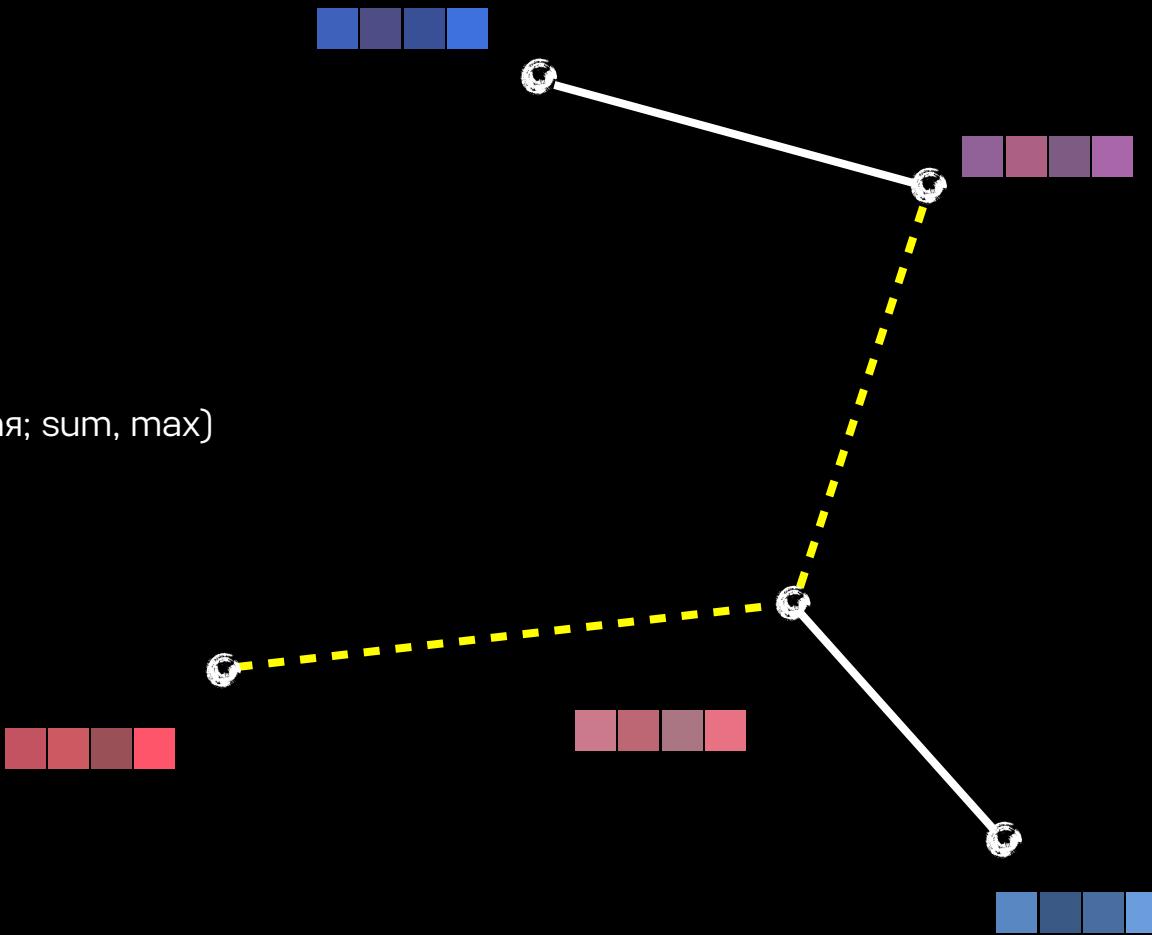
Graph-level regression

2. Readout

$$h_{\mathcal{V}}^{(t)} = \bigoplus_{j \in \mathcal{V}} h_j^{(t)}$$

$$\bigoplus_{j \in \mathcal{V}}$$

— функция агрегации (инвариантная; sum, max)



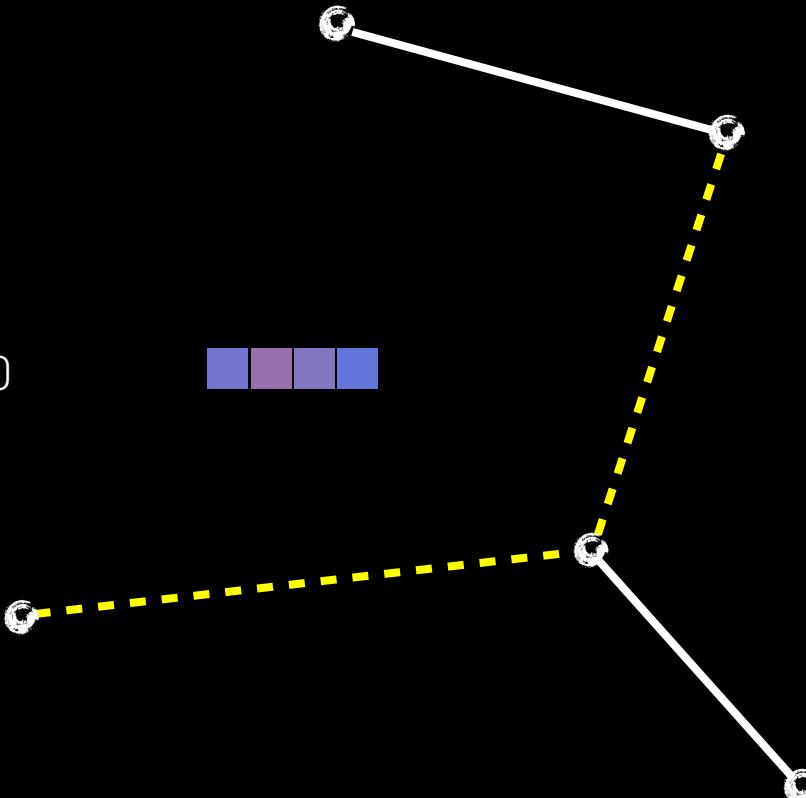
Graph-level regression

2. Readout

$$h_{\mathcal{V}}^{(t)} = \bigoplus_{j \in \mathcal{V}} h_j^{(t)}$$

$$\bigoplus_{j \in \mathcal{V}}$$

— функция агрегации (инвариантная; sum, max)



Graph-level regression

2. Readout

$$h_{\mathcal{V}}^{(t)} = \bigoplus_{j \in \mathcal{V}} h_j^{(t)}$$

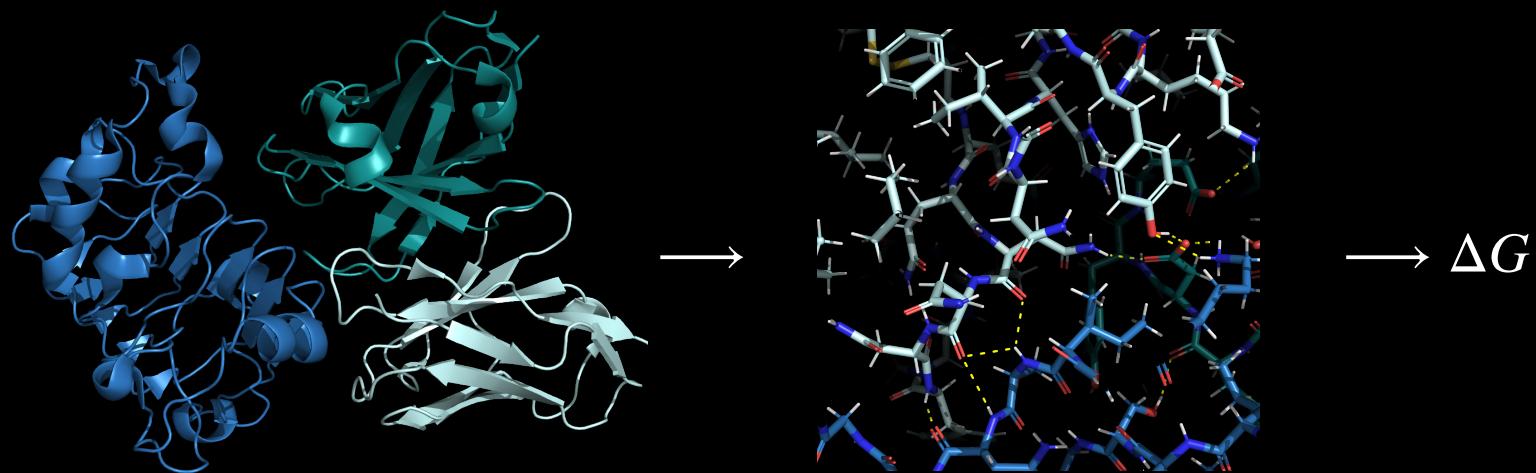
$$\bigoplus_{j \in \mathcal{V}}$$

– функция агрегации (инвариантная; sum, max)



ΔG

Предсказание свободной энергии связывания



Geometric deep learning

1. Message passing

$$h_i^{(t+1)} = \phi \left(h_i^{(t)}, \bigoplus_{j \in \mathcal{N}_i} \psi(h_i^{(t)}, h_j^{(t)}) \right)$$

2. Readout

$$h_{\mathcal{V}}^{(t)} = \bigoplus_{j \in \mathcal{V}} h_j^{(t)}$$

Geometric deep learning

1. Message passing

$$h_i^{(t+1)} = \phi \left(h_i^{(t)}, \bigoplus_{j \in \mathcal{N}_i} \psi(h_i^{(t)}, h_j^{(t)}) \right)$$

2. Readout

$$h_{\mathcal{V}}^{(t)} = \bigoplus_{j \in \mathcal{V}} h_j^{(t)}$$

В свёрточных сетях

Geometric deep learning

1. Message passing

$$h_i^{(t+1)} = \phi \left(h_i^{(t)}, \bigoplus_{j \in \mathcal{N}_i} \psi(h_i^{(t)}, h_j^{(t)}) \right)$$

В свёрточных сетях

Скалярное произведение кернела на патч с центром в пикселе i

2. Readout

$$h_{\mathcal{V}}^{(t)} = \bigoplus_{j \in \mathcal{V}} h_j^{(t)}$$

Geometric deep learning

1. Message passing

$$h_i^{(t+1)} = \phi \left(h_i^{(t)}, \bigoplus_{j \in \mathcal{N}_i} \psi(h_i^{(t)}, h_j^{(t)}) \right)$$

В свёрточных сетях

Скалярное произведение кернела на патч с центром в пикселе i

2. Readout

$$h_{\mathcal{V}}^{(t)} = \bigoplus_{j \in \mathcal{V}} h_j^{(t)}$$

Global max pooling

Geometric deep learning

1. Message passing

$$h_i^{(t+1)} = \phi \left(h_i^{(t)}, \bigoplus_{j \in \mathcal{N}_i} \psi(h_i^{(t)}, h_j^{(t)}) \right)$$

2. Readout

$$h_{\mathcal{V}}^{(t)} = \bigoplus_{j \in \mathcal{V}} h_j^{(t)}$$

В рекуррентных сетях

Geometric deep learning

1. Message passing

$$h_i^{(t+1)} = \phi \left(h_i^{(t)}, \bigoplus_{j \in \mathcal{N}_i} \psi(h_i^{(t)}, h_j^{(t)}) \right)$$

В рекуррентных сетях

Обновление рекуррентной ячейки

2. Readout

$$h_{\mathcal{V}}^{(t)} = \bigoplus_{j \in \mathcal{V}} h_j^{(t)}$$

Geometric deep learning

1. Message passing

$$h_i^{(t+1)} = \phi \left(h_i^{(t)}, \bigoplus_{j \in \mathcal{N}_i} \psi(h_i^{(t)}, h_j^{(t)}) \right)$$

В рекуррентных сетях

Обновление рекуррентной ячейки

2. Readout

$$h_{\mathcal{V}}^{(t)} = \bigoplus_{j \in \mathcal{V}} h_j^{(t)}$$

Последнее состояние

Geometric deep learning

1. Message passing

$$h_i^{(t+1)} = \phi \left(h_i^{(t)}, \bigoplus_{j \in \mathcal{N}_i} \psi(h_i^{(t)}, h_j^{(t)}) \right)$$

2. Readout

$$h_{\mathcal{V}}^{(t)} = \bigoplus_{j \in \mathcal{V}} h_j^{(t)}$$

В трансформерах

Geometric deep learning

1. Message passing

$$h_i^{(t+1)} = \phi \left(h_i^{(t)}, \bigoplus_{j \in \mathcal{N}_i} \psi(h_i^{(t)}, h_j^{(t)}) \right)$$

В трансформерах

Multi-head attention

2. Readout

$$h_{\mathcal{V}}^{(t)} = \bigoplus_{j \in \mathcal{V}} h_j^{(t)}$$

Geometric deep learning

1. Message passing

$$h_i^{(t+1)} = \phi \left(h_i^{(t)}, \bigoplus_{j \in \mathcal{N}_i} \psi(h_i^{(t)}, h_j^{(t)}) \right)$$

В трансформерах

Multi-head attention

2. Readout

$$h_{\mathcal{V}}^{(t)} = \bigoplus_{j \in \mathcal{V}} h_j^{(t)}$$

Global avg pooling

Geometric deep learning

1. Message passing

$$h_i^{(t+1)} = \phi \left(h_i^{(t)}, \bigoplus_{j \in \mathcal{N}_i} \psi(h_i^{(t)}, h_j^{(t)}) \right)$$

В трансформерах

Multi-head attention

2. Readout

$$h_{\mathcal{V}}^{(t)} = \bigoplus_{j \in \mathcal{V}} h_j^{(t)}$$

Global avg pooling

<CLS> token readout