TRANSFORMERS 1

Lecture 11: COMPSCI/DATA 182: Deep Learning

10/03/2024

Tokenization

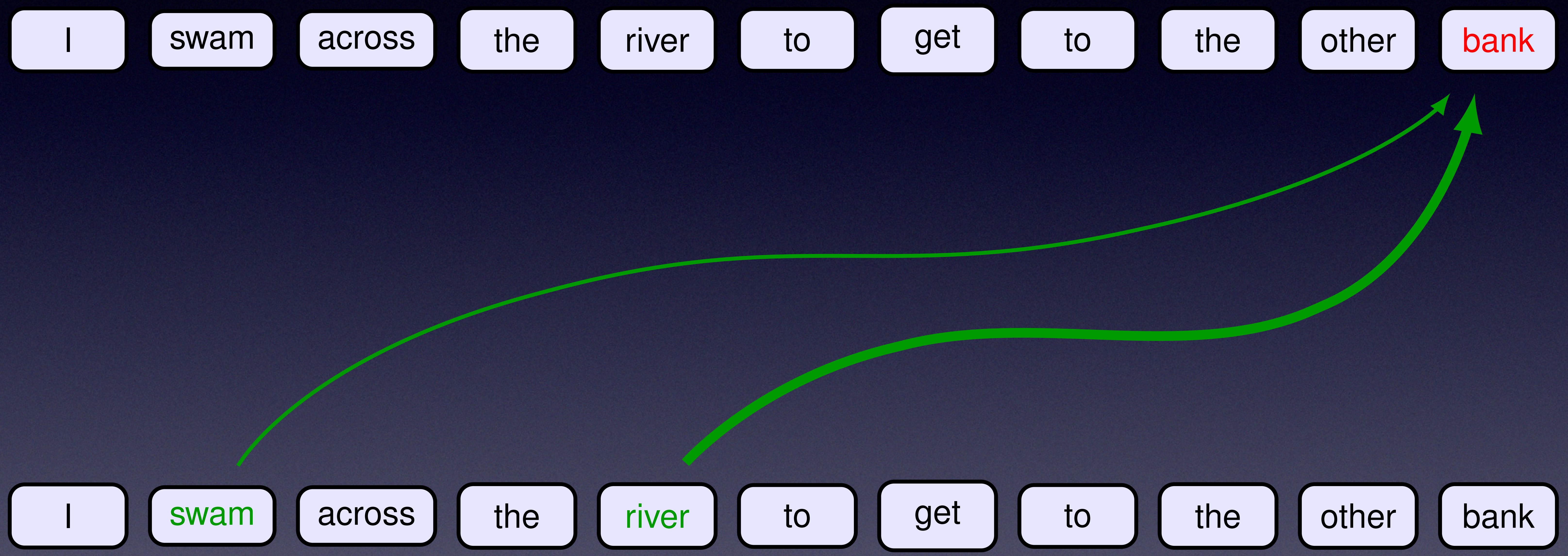
```
tokenizer = create_tokenizer()
input_text = ["hello there"]
tokens = tokenizer(input_text)

print(tokens)

> [0 387 2991 25 888 99 -1]
```

- “hello” is broken up into **two** tokens : 387 2991
- “world” into **three** tokens : 25 888 99
- 0 and -1 are START and END

Transformers

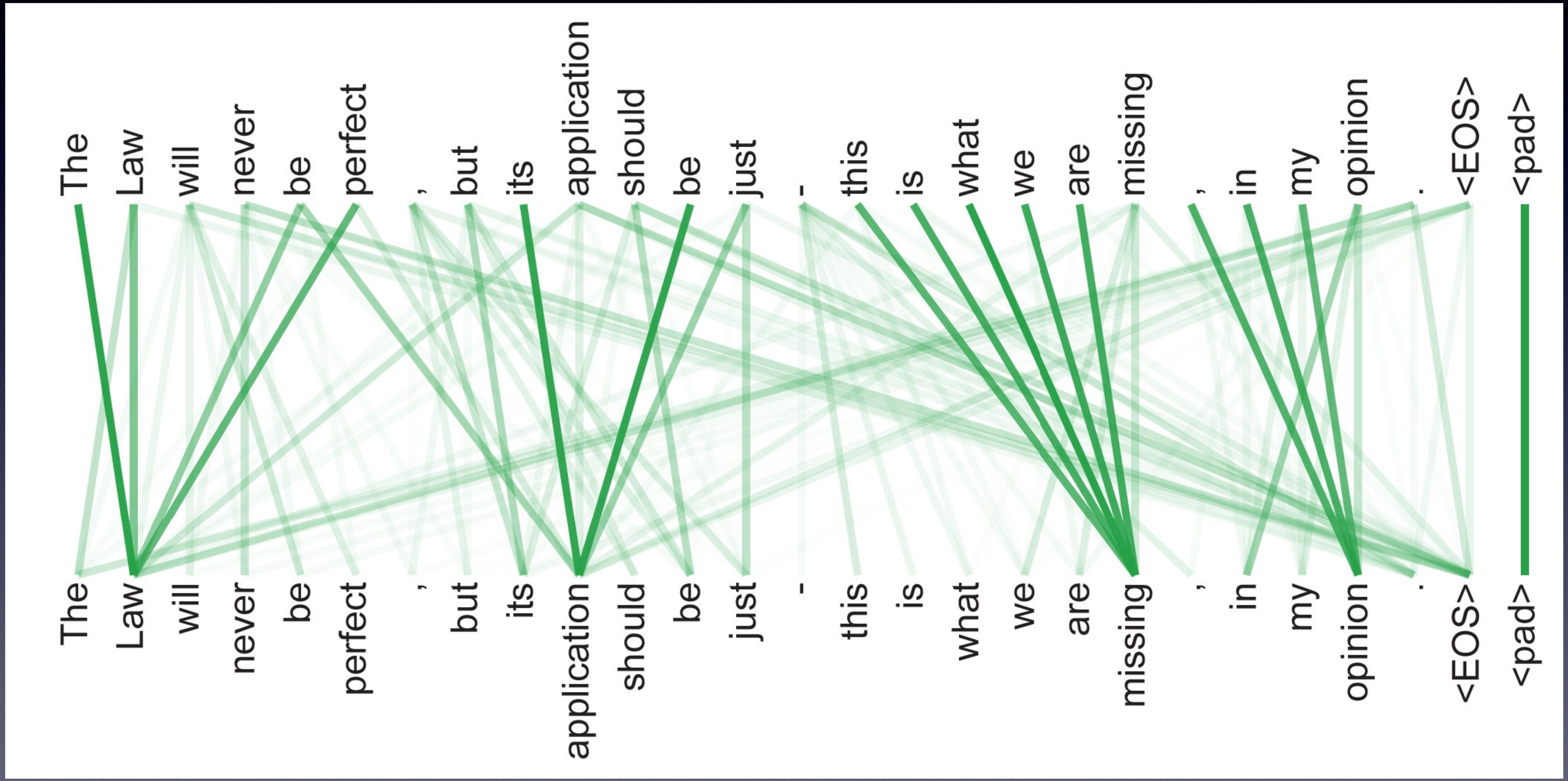
- **Transform:** a set of vectors in a space, to a set of vectors *with the same dimensionality,* in another space
  - New space is richer, better for downstream tasks
- Originally developed for NLP
  - Where we were using RNNs !
- Key advantages:
  - trained on large body of data (text, initially)
  - downstream tasks; Foundation models
  - Scalability hypothesis, leverage GPUs

Attention



- Originally an enhancement for RNNs

Attention



- Tokens could be image patches, proteins, …..

Transformer processing

Input: set of vectors $\{x_n\}$ of dimensionality D, $n = 1,..,N$

$\boldsymbol{x_{ni}}$: features

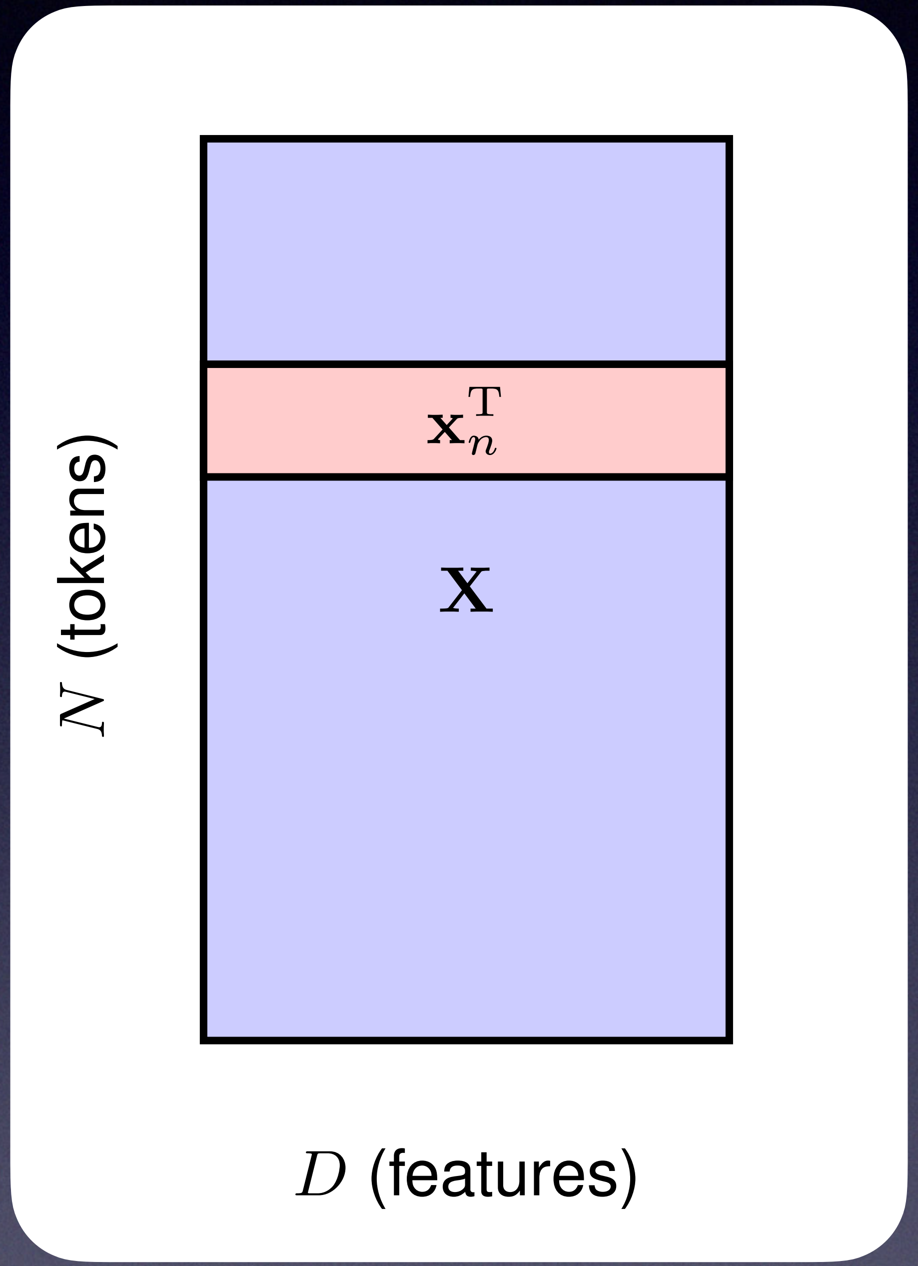
X : input data matrix $(N \times D)$

$$\boxed{\widetilde{\mathbf{X}} = \mathrm{TransformerLayer}\left[\mathbf{X}\right]}$$

$\{x_1, \ldots, x_N\}$: input in embedding space

Transform to

$\{y_1,\ldots,y_N\}$ of SAME size, in another embedding space



- There is no need to design a new (NN) architecture for a MIX of data types !

Attention **coefficients**

$$\mathbf{y}_n = \sum_{m=1}^{N} a_{nm}\mathbf{x}_m$$

$$a_{nm} \geqslant 0$$

$$\sum_{m=1}^{N} a_{nm} = 1.$$

Self attention

$$a_{nm} = \frac{\exp(\mathbf{x}_n^{\mathrm{T}}\mathbf{x}_m)}{\sum_{m'=1}^{N}\exp(\mathbf{x}_n^{\mathrm{T}}\mathbf{x}_{m'})}$$

$$\mathbf{Y} = \mathrm{Softmax}\left[\mathbf{X}\mathbf{X}^{\mathrm{T}}\right]\mathbf{X}$$

Back to Learning, hence **Parameters !**

$$\widetilde{\mathbf{X}} = \mathbf{X}\mathbf{U}$$

$$\mathbf{X}\mathbf{U}\mathbf{U}^{\mathrm{T}}\mathbf{X}^{\mathrm{T}}$$

$$\mathbf{Y} = \mathrm{Softmax}\left[\mathbf{X}\mathbf{X}^{\mathrm{T}}\right]\mathbf{X} \rightarrow \mathbf{Y} = \mathrm{Softmax}\left[\mathbf{X}\mathbf{U}\mathbf{U}^{\mathrm{T}}\mathbf{X}^{\mathrm{T}}\right]\mathbf{X}\mathbf{U}$$

- U: matrix of (learnable) parameters

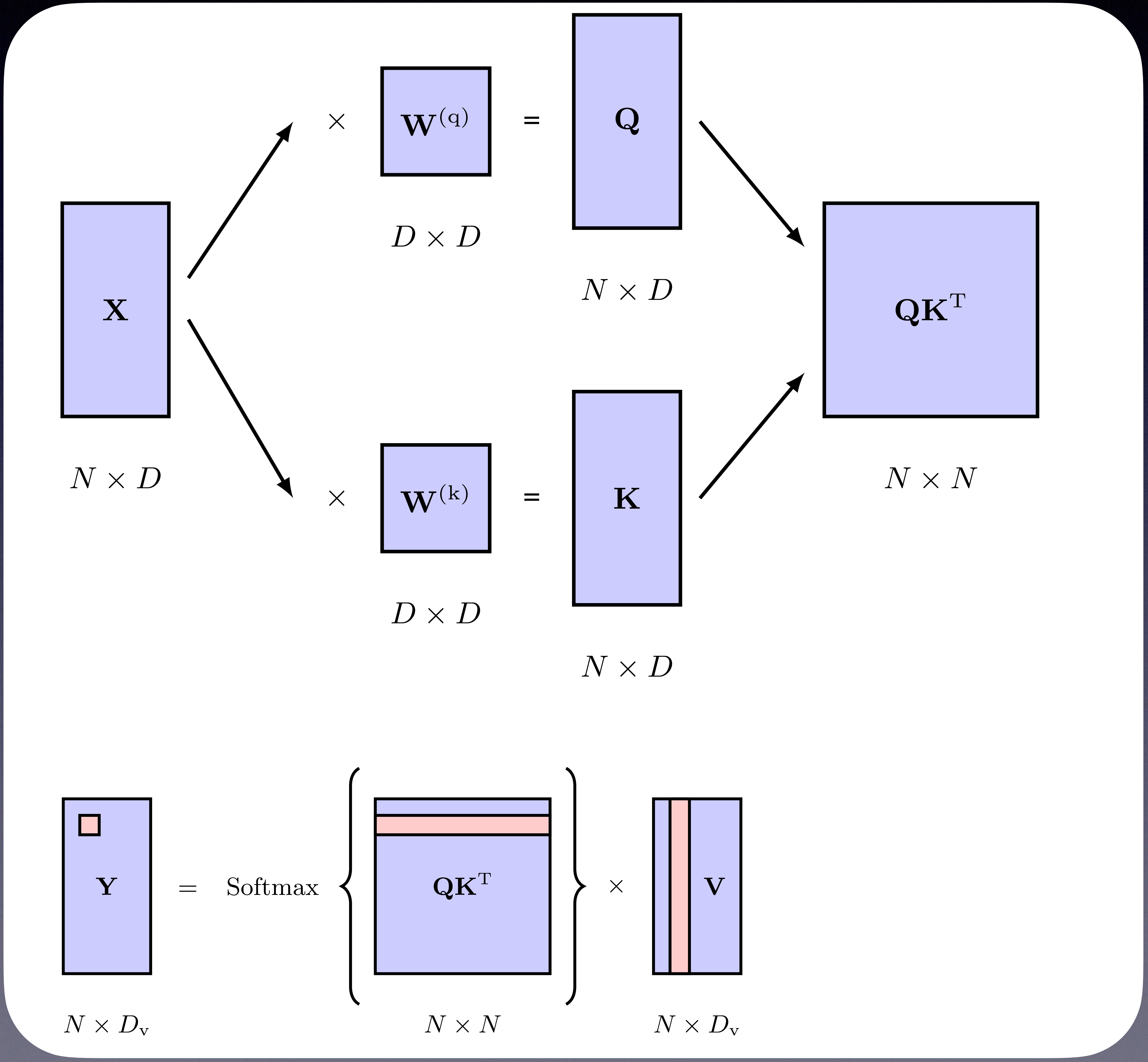
Introducing .. the Attention **Head**

- The fundamental *trinity* : **{Query, Key, Value}**

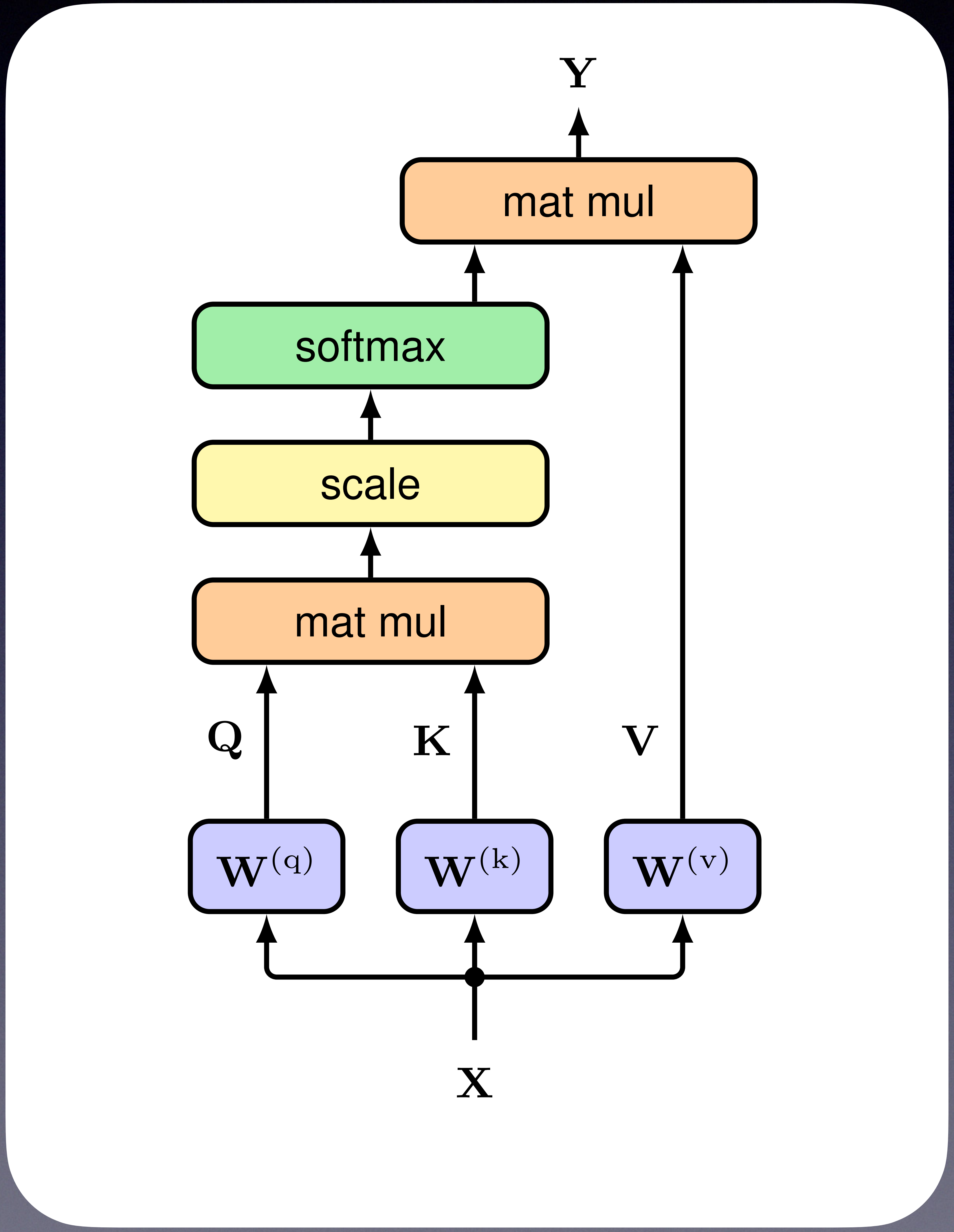
$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{(\mathrm{q})}$$
$$\mathbf{K} = \mathbf{X}\mathbf{W}^{(\mathrm{k})}$$
$$\mathbf{V} = \mathbf{X}\mathbf{W}^{(\mathrm{v})}$$

$$\mathbf{Y} = \mathrm{Softmax}\left[\mathbf{X}\mathbf{X}^{\mathrm{T}}\right]\mathbf{X}$$

$$\mathbf{Y} = \mathrm{Softmax}\left[\mathbf{Q}\mathbf{K}^{\mathrm{T}}\right]\mathbf{V}$$



Single Attention-Head: Information flow

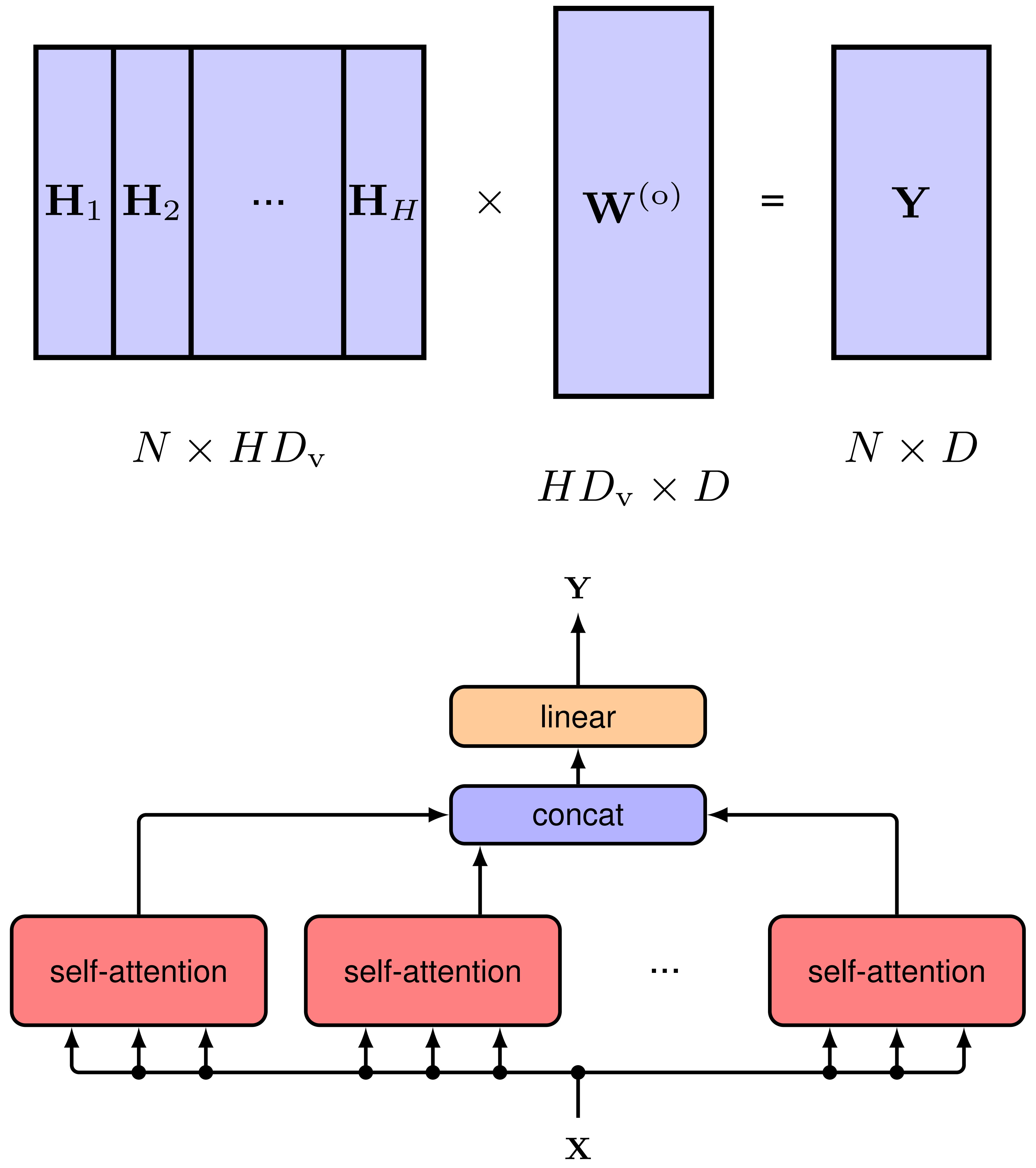


Multi-head Attention

$$\mathbf{H}_h = \mathrm{Attention}(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h)$$

$$\mathbf{Q}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{q})}$$
$$\mathbf{K}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{k})}$$
$$\mathbf{V}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{v})}.$$

$$\mathbf{Y}(\mathbf{X}) = \mathrm{Concat}\left[\mathbf{H}_1, \ldots, \mathbf{H}_H\right] \mathbf{W}^{(\mathrm{o})}$$

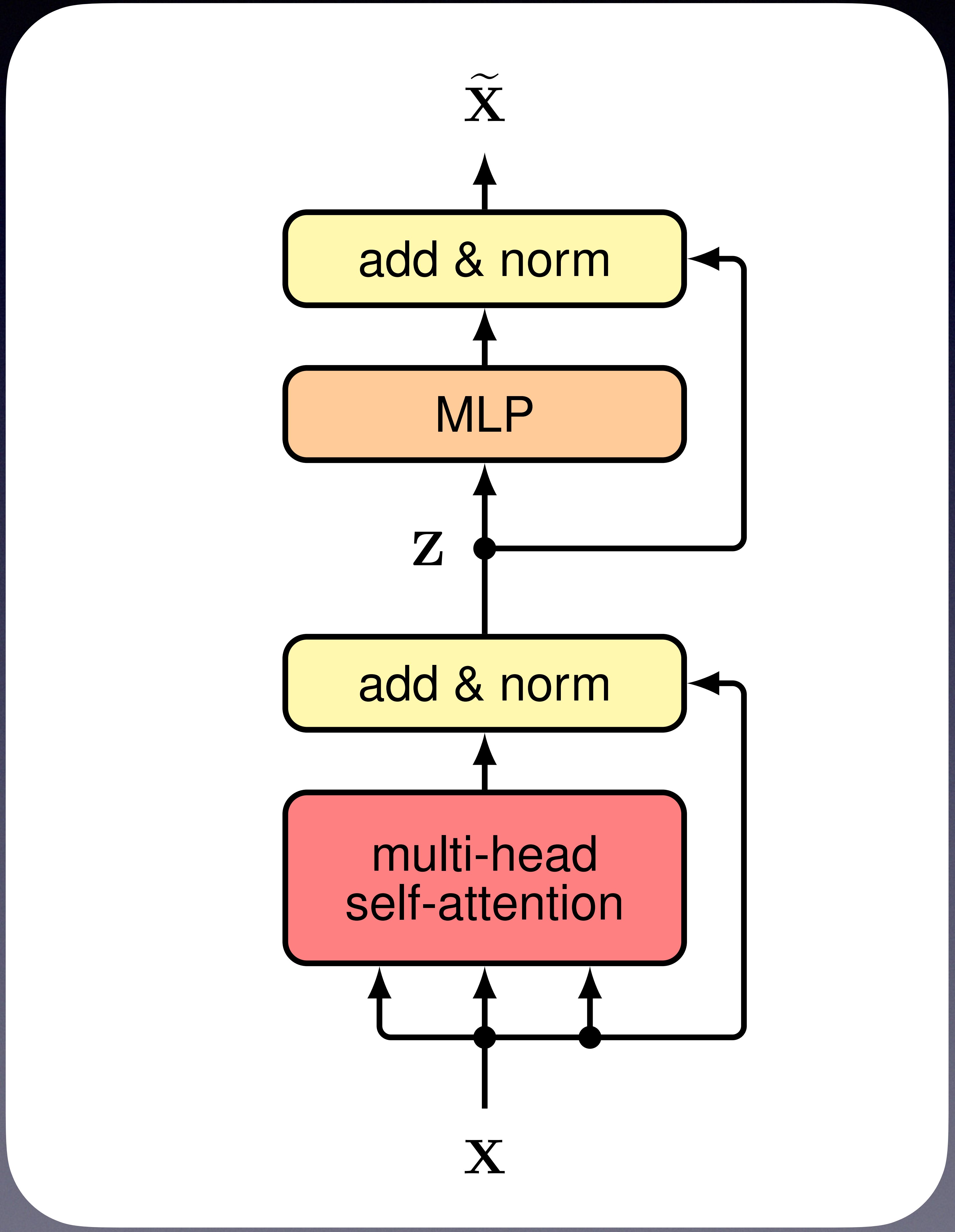


Transformer **Layers**

$$\mathbf{Z} = \text{LayerNorm}\left[\mathbf{Y}(\mathbf{X}) + \mathbf{X}\right]$$

$$\widetilde{\mathbf{X}} = \text{LayerNorm}\left[\text{MLP}\left[\mathbf{Z}\right] + \mathbf{Z}\right]$$

- Stack multiple attention layers
  - On top of each other
- Addition of original: residuals, works better
- Layer normalization: efficiency
- MLP: non-linearity
  - For instance, 2-layer FC with ReLU



Positional Encoding

- Consider:
    - 1) *The food was good, not bad at all*
    - 2) *The food was bad, not good at all*

RNN Code !

```python
import torch as th
import torch.nn as nn

class RNNLayer(nn.Module):
    def __init__(self, input_size, hidden_size, nonlinearity=th.tanh):
        """
        Initialize a single RNN layer.

        Inputs:
        - input_size: Data input feature dimension
        - hidden_size: RNN hidden state size (also the output feature dimension)
        - nonlinearity: Nonlinearity applied to the rnn output
        """
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.nonlinearity = nonlinearity

        # Initialize the weights for input to hidden and hidden to hidden
        self.W_ih = nn.Parameter(th.randn(hidden_size, input_size))  # (hidden_size, input_size)
        self.W_hh = nn.Parameter(th.randn(hidden_size, hidden_size))  # (hidden_size, hidden_size)

        # Initialize a single bias (no separate biases for W_ih and W_hh)
        self.bias = nn.Parameter(th.randn(hidden_size))  # (hidden_size,)

    def forward(self, x):
        """
        RNN forward pass

        Inputs:
        - x: input tensor (B, seq_len, input_size)

        Returns:
        - all_h: tensor of size (B, seq_len, hidden_size) containing hidden states
                 produced for each timestep
        - last_h: hidden state from the last timestep (B, hidden_size)
        """
        B, seq_len, _ = x.shape
        h = th.zeros(B, self.hidden_size)
        h_list = []
        for t in range(seq_len):
            h = self.nonlinearity(th.mm(x[:, t], self.W_ih.T) + th.mm(h, self.W_hh.T) + self.bias)
            h_list.append(h)
        all_h = th.stack(h_list, dim=1)
        last_h = h_list[-1]
        return all_h, last_h
```

- $W_{ih} \in \mathbb{R}^{\text{hidden\_size} \times \text{input\_size}}$
- $W_{hh} \in \mathbb{R}^{\text{hidden\_size} \times \text{hidden\_size}}$
- $b \in \mathbb{R}^{\text{hidden\_size}}$
- σ is a **non-linear activation function** like **tanh**

$$h_t = \sigma(W_{ih}x_t + W_{hh}h_{t-1} + b)$$

```python
class RNNCharPredictionModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        """
        Initialize the RNN Character Prediction Model.

        Inputs:
        - input_size: Number of unique characters
        - hidden_size: Number of hidden units
        - output_size: Number of unique characters (same as input_size)
        """
        super().__init__()
        self.rnn = RNNLayer(input_size, hidden_size)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        """
        Forward pass through the RNN character prediction model.
        - x: Input tensor (batch_size, seq_len, input_size)
        Returns:
        - Output logits for the next character prediction.
        """
        last_h = self.rnn(x)  # Get the last hidden state from the RNN
        out = self.fc(last_h)  # Pass it through the fully connected layer
        return out  # Return logits (unnormalized scores for each character)
```