Transformers

Plan

- **Transformers**
  - Multi-Headed Self-Attention
  - Transformer Layers
  - GPT-style Decoders

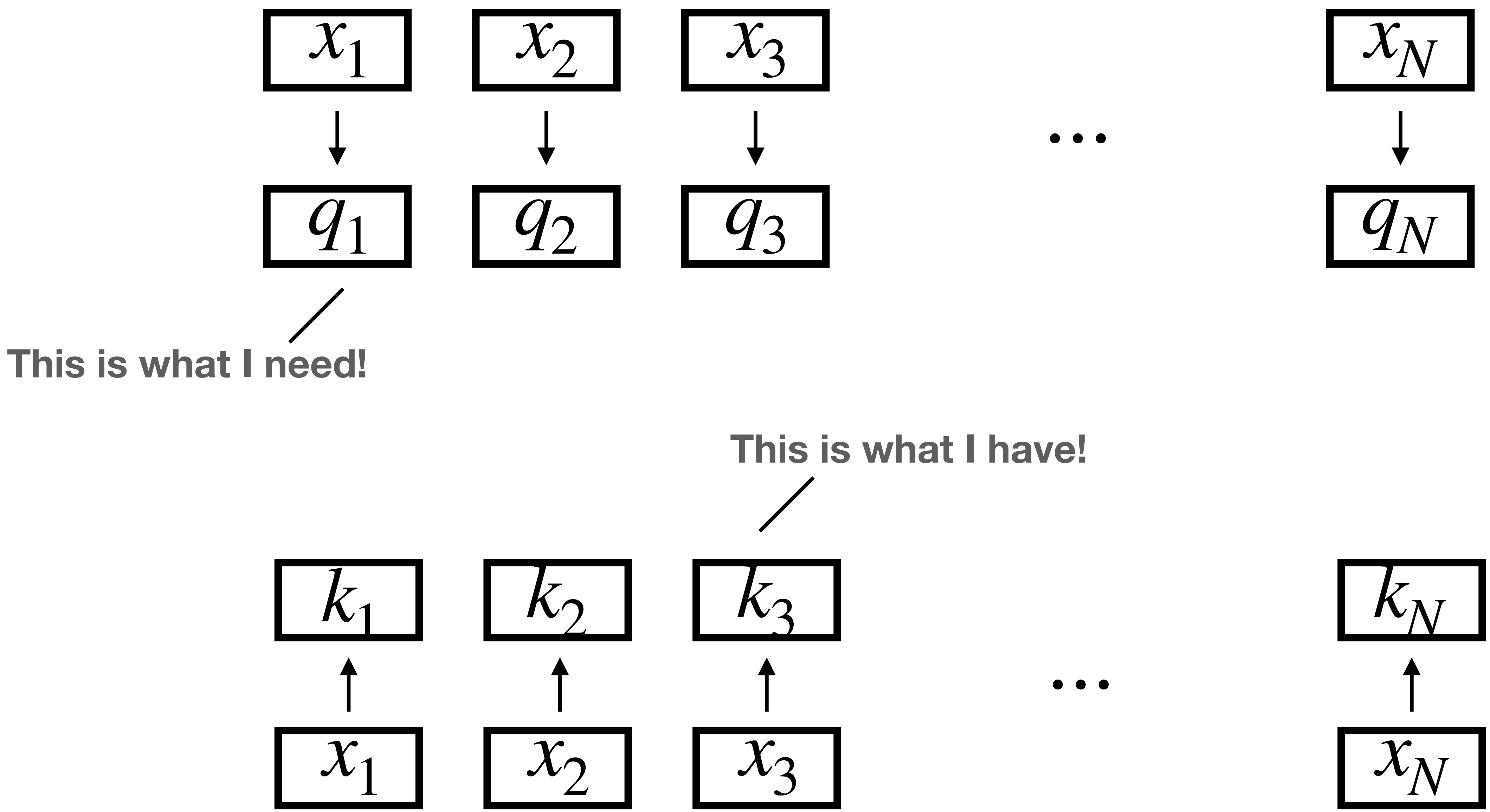
Self-Attention

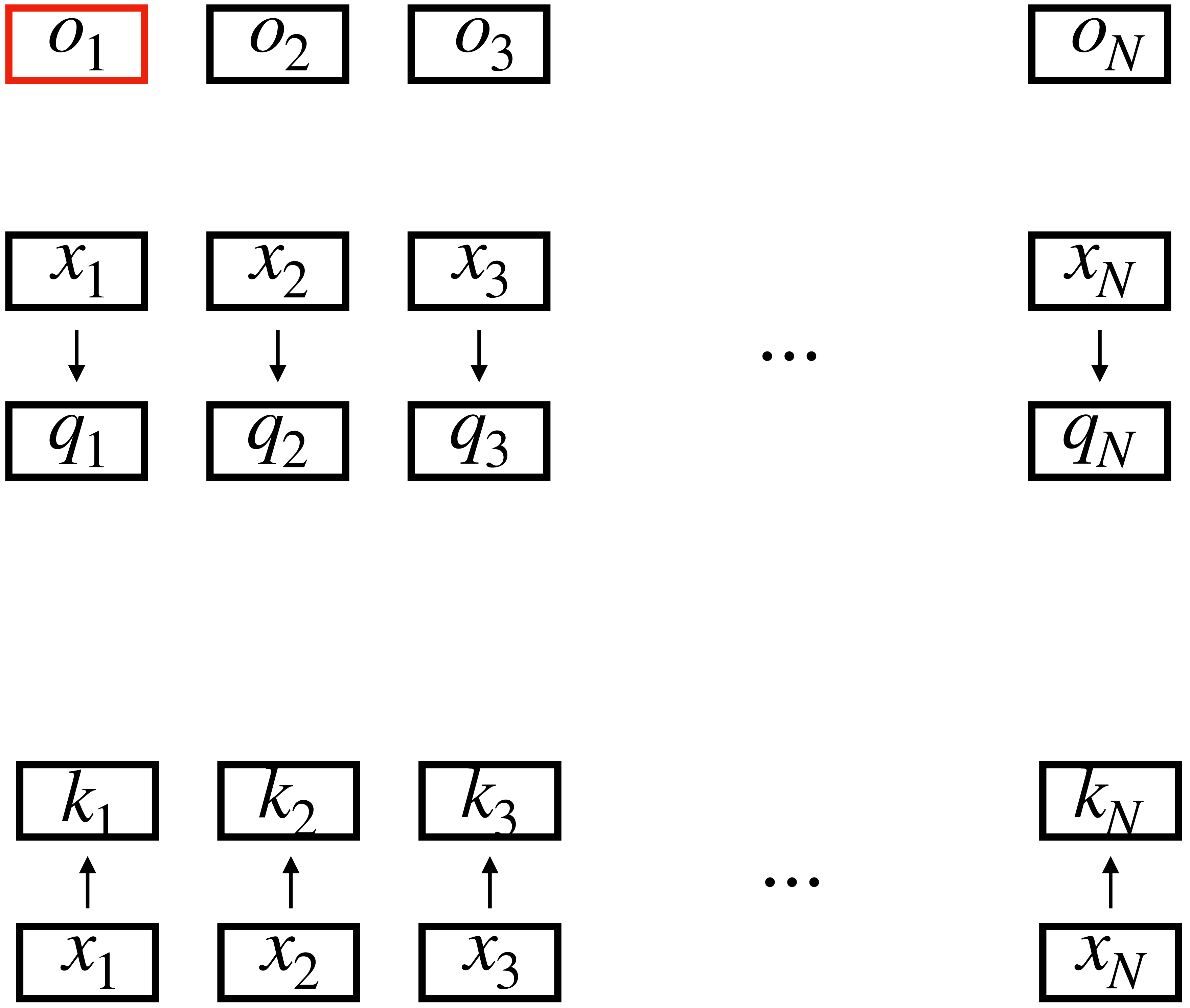
- Suppose there are N input tokens $x_1, \cdots, x_N \in \mathbb{R}^d$ (e.g. words).
- Self-attention maps the input sequence into another sequence of the same length.
- For each token x_i, we compute a **distribution** $\alpha_i \in \mathbb{R}_+^d$ over all input tokens, given by

$$\alpha_{i,j} = \frac{\exp(x_i \cdot x_j)}{\sum_{j=1}^{N} \exp(x_i \cdot x_j)}$$

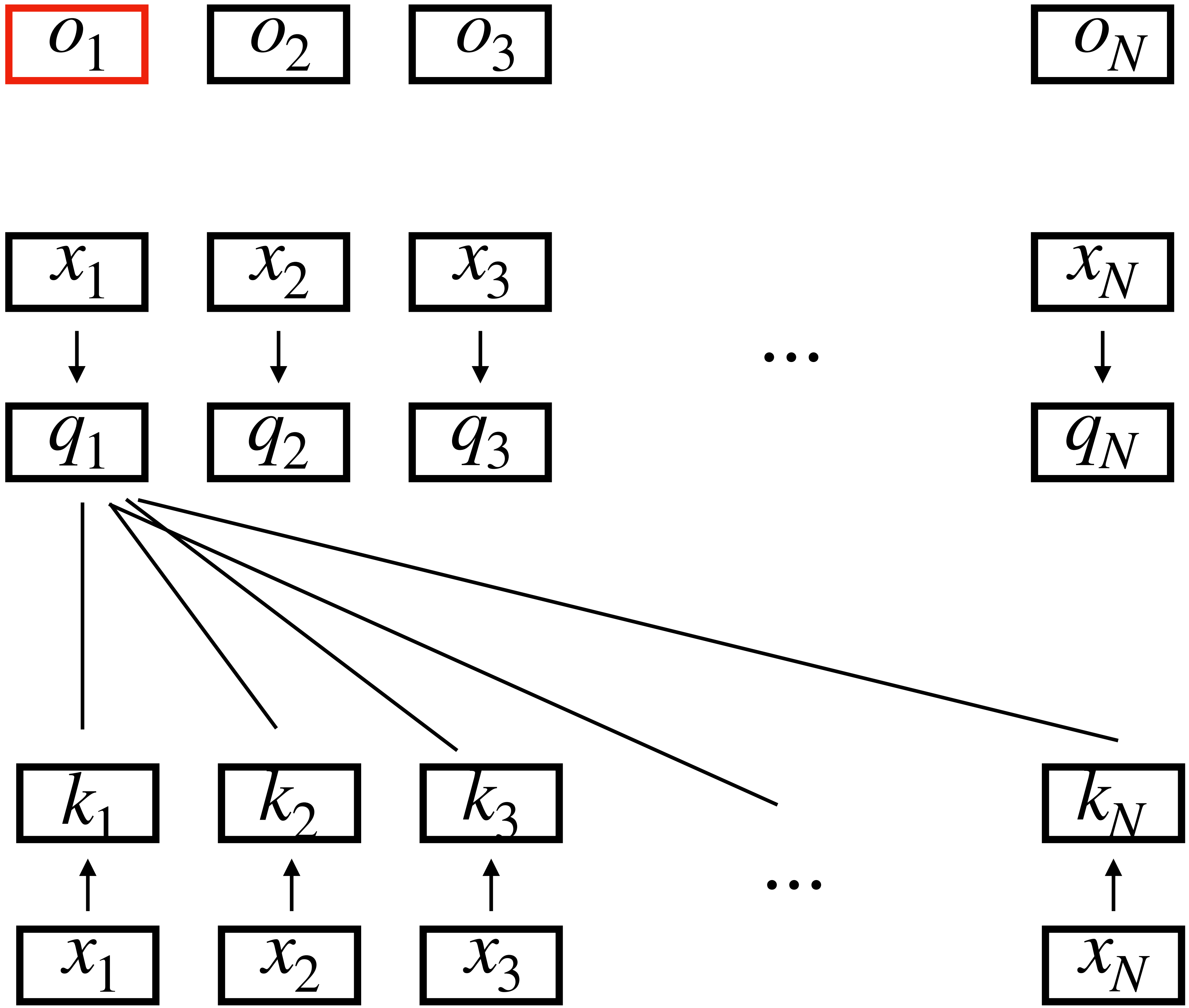
- To compute the ith output, we combine the input vectors using the attention weights α_i.
- We give ourselves more flexibility using **queries** and **keys** to compute the attention weights, and by computing a combination of **values** instead of the original input.

$$q_i = M_q x_i + \beta_q \qquad k_i = M_k x_i + \beta_k$$

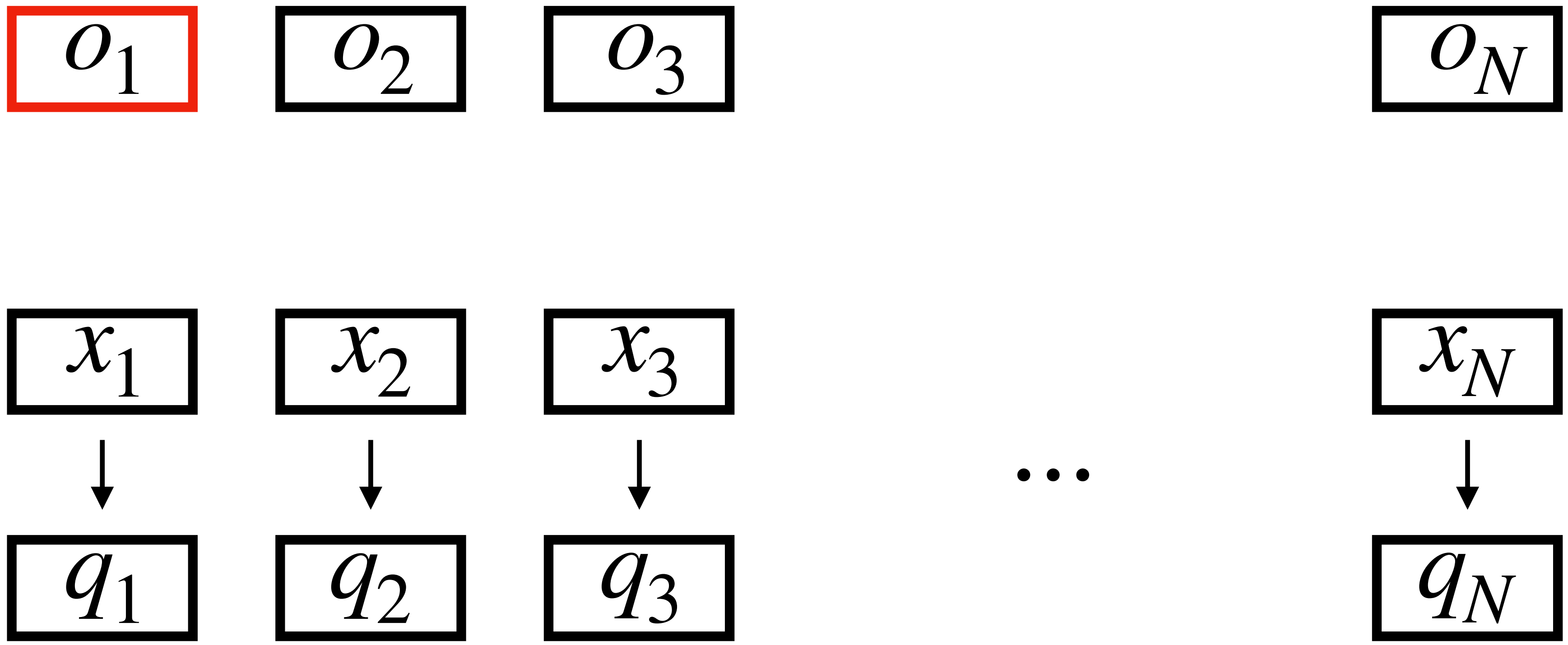




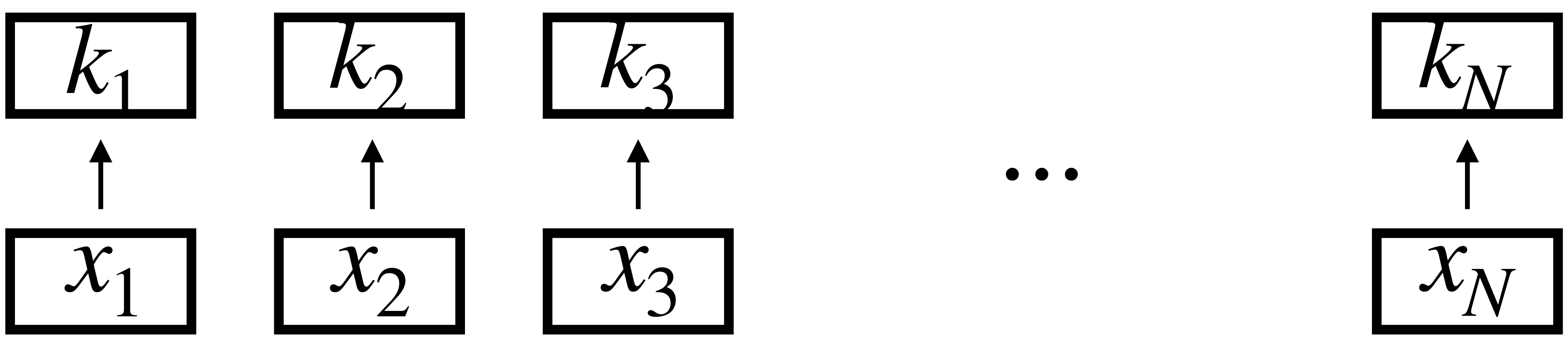
o_1 o_2 o_3 o_N
x_1 x_2 x_3 x_N
q_1 q_2 q_3 q_N
k_1 k_2 k_3 k_N
x_1 x_2 x_3 x_N



o_1
o_2
o_3
o_N
x_1
x_2
x_3
x_N
...
q_1
q_2
q_3
q_N
k_1
k_2
k_3
k_N
...
x_1
x_2
x_3
x_N

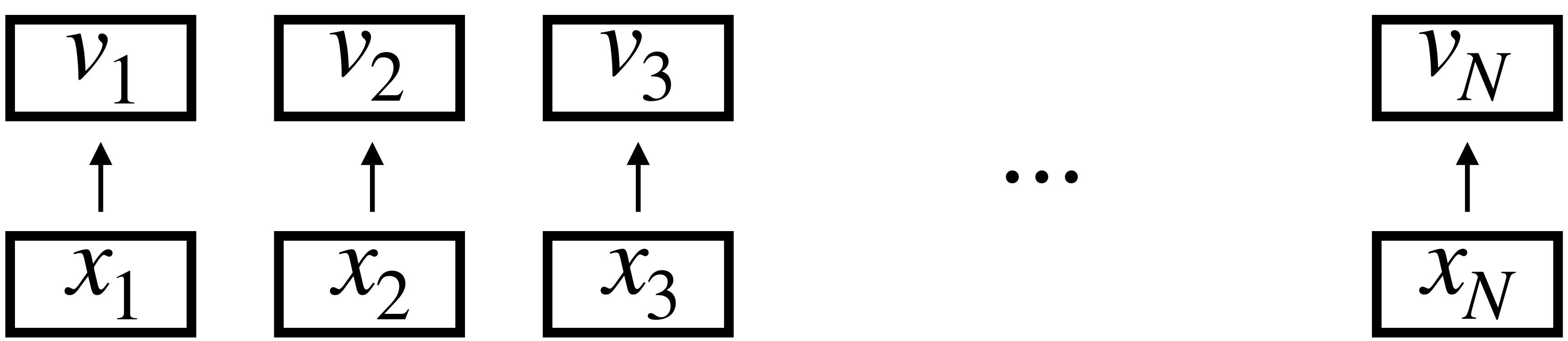


$$\alpha_{1,j} \propto \exp(q_1 \cdot k_j)$$



$$\alpha_{1,j} \propto \exp(q_1 \cdot k_j)$$

$$o_1 = \sum_{i=1}^{N} \alpha_{1,j} v_j$$



$$v_i = M_v x_i + \beta_v$$

Scaled Dot-Product Self-Attention

- Arrange the input into a $N \times D$ matrix X (each row is an element of the sequence.)

- Matrices of **queries** $Q = XM_q + \beta_q$, **keys** $K = XM_k + \beta_k$, and **values** $V = XM_v + \beta_v$. Each row is a (query / key / value).

$$\text{SA}(X) = \frac{\text{softmax}(QK^T)}{\sqrt{D}} V$$

(Multi-Head) Scaled Dot-Product Self-Attention

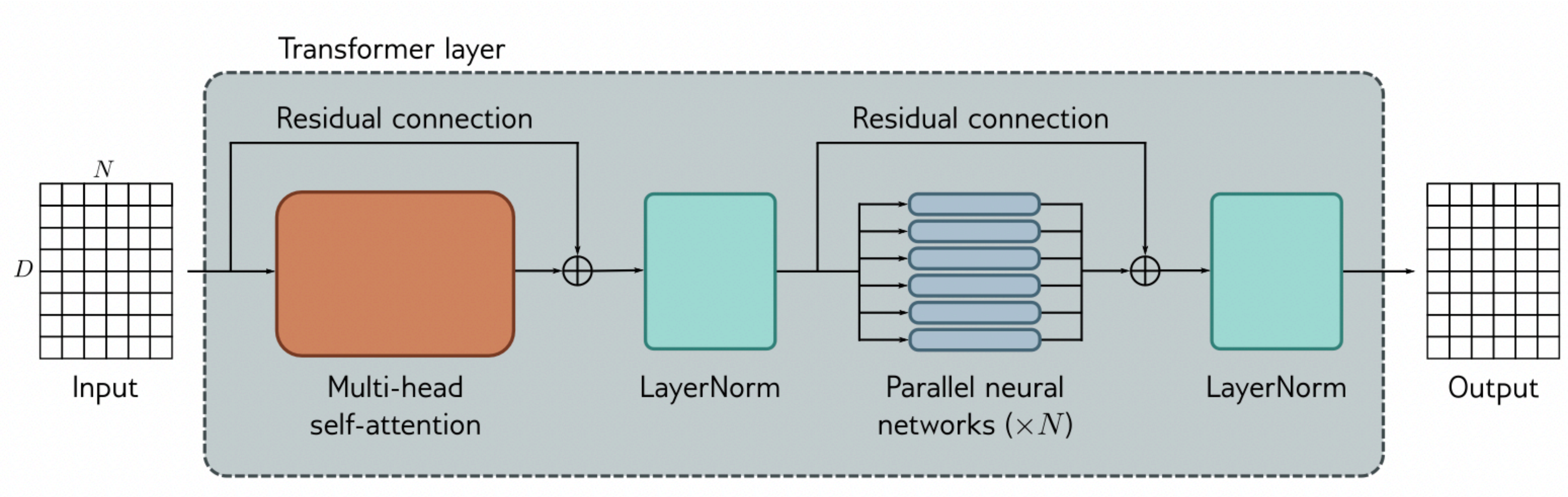
- We have H attention mechanisms (each with its own parameters), of dimension D/H.
- Compute output from each and concatenate.
- Project once more to form the output.

```python
# Compute queries, keys, and values for all heads at once.
q = jax.vmap(self.lin_q)(x).reshape(N, D // n_head, n_head)
k = jax.vmap(self.lin_k)(x).reshape(N, D // n_head, n_head)
v = jax.vmap(self.lin_v)(x).reshape(N, D // n_head, n_head)

# Attends over the values to produce the output values.
# The attention coefficients are masked using jnp.tril.
def sa(q, k, v):
    mask = jnp.triu(jnp.ones((N, N)) * -float('inf'), k=1)
    return jax.nn.softmax(q @ k.T / jnp.sqrt(D // n_head) + mask) @ v
```

Transformer Layers

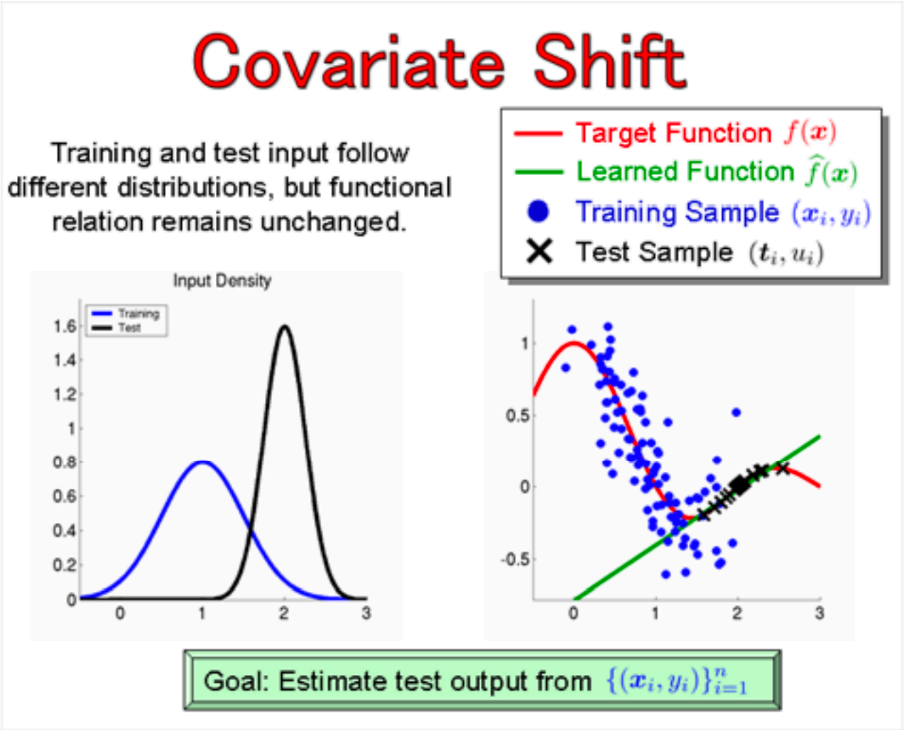
Transformer Layer



- Actually: **LayerNorm** —> Self-Attention —> **LayerNorm** —> MLP
- On Layer Normalization In the Transformer Architecture [**https://arxiv.org/pdf/2002.04745.pdf**]

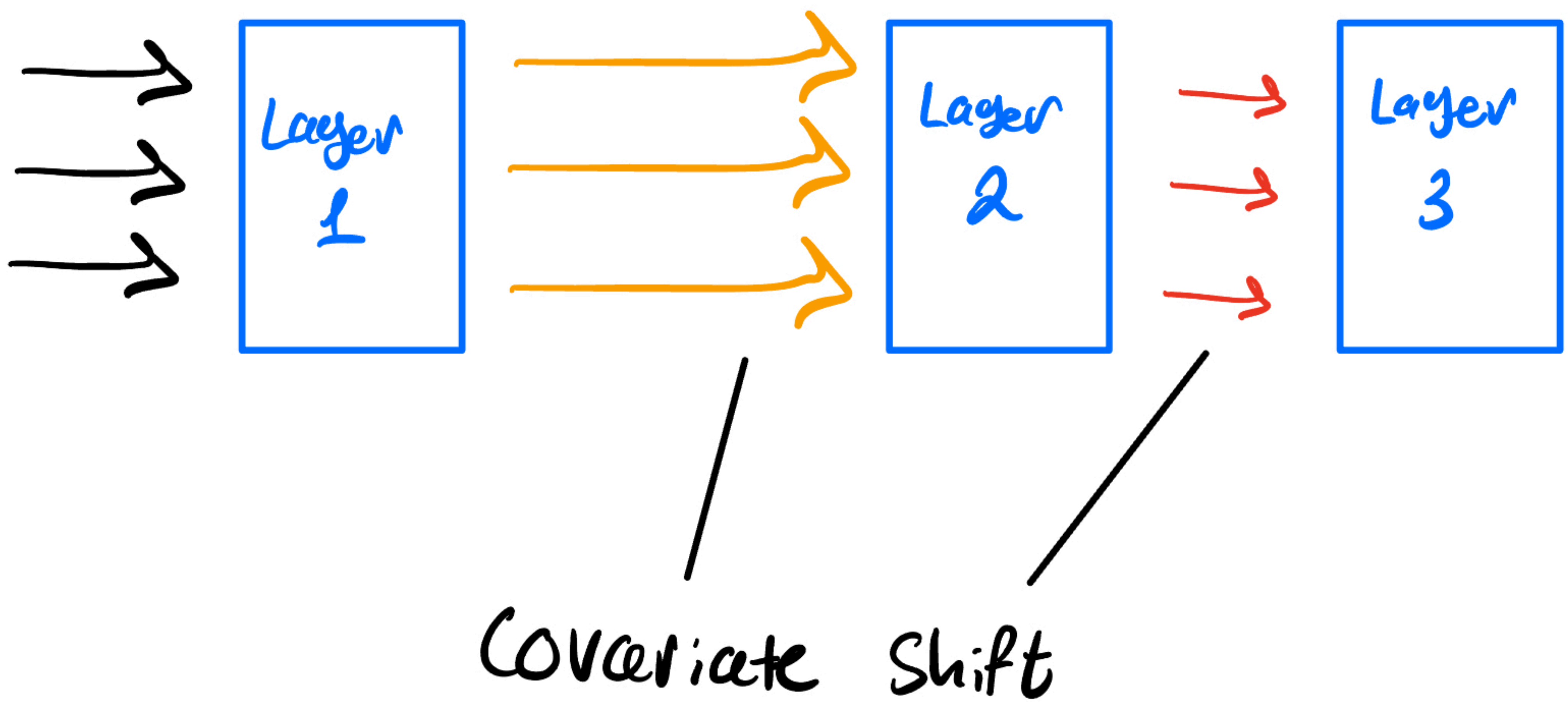
Batch Normalization

- Recall that in regression, we predict a **response** y from **covariates** (features) $x_1, \cdots, x_p$.
- What happens if the data we train on and the data we test on are different? One type of such "distribution shift" is **Covariate Shift**.
- The marginal distribution of the covariates is $q(x)$ at train time and $p(x)$ at test time. Here, $p(y \mid x) = q(y \mid x)$ - only the marginal of the covariates changes.



Batch Normalization

- **Interval Covariate Shift:** covariate shift is happening **inside** of the network.



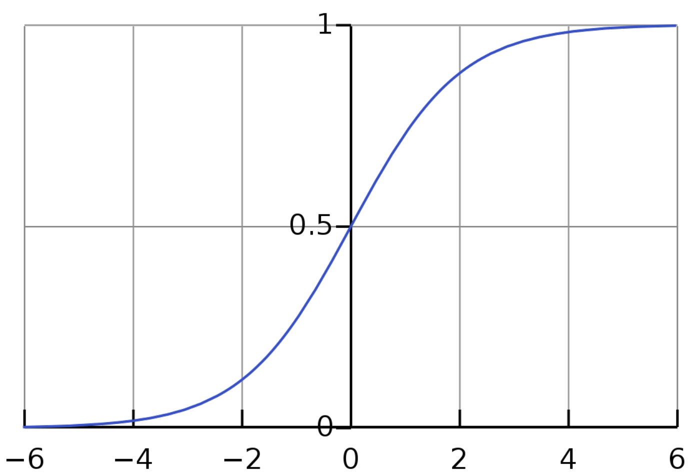
Batch Normalization

- Solution: shift and scale the input to each layer.

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_B^{(k)}}{\sqrt{(\sigma_B^{(k)})^2 + \epsilon}}$$

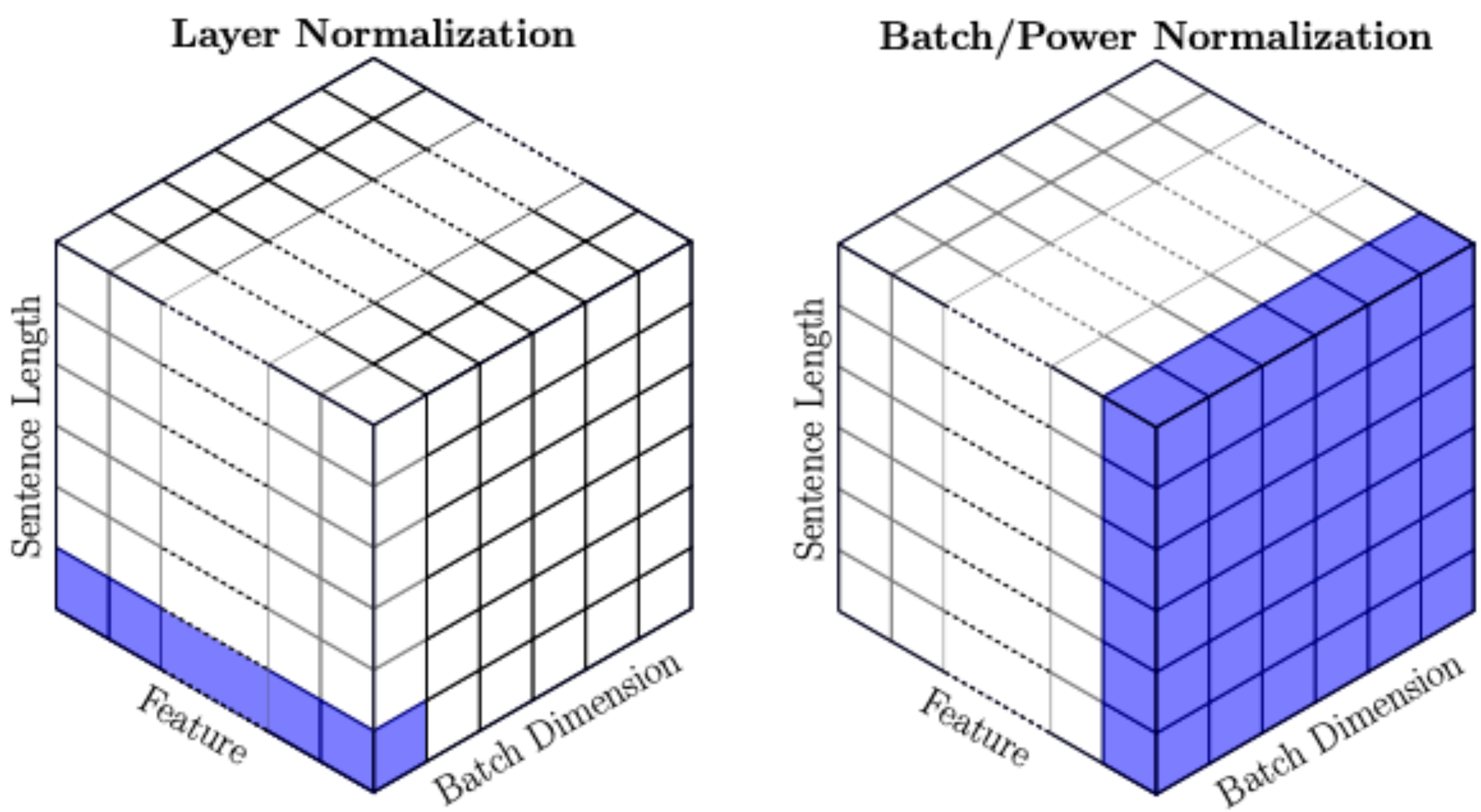
$$y_i^{(k)} = \gamma^{(k)} \hat{x}_i^{(k)} + \beta^{(k)}$$

- Gamma(s) and beta(s) are learned parameters used to ensure that “the transformation inserted in the network can represent the identity transform.”



Layer Normalization

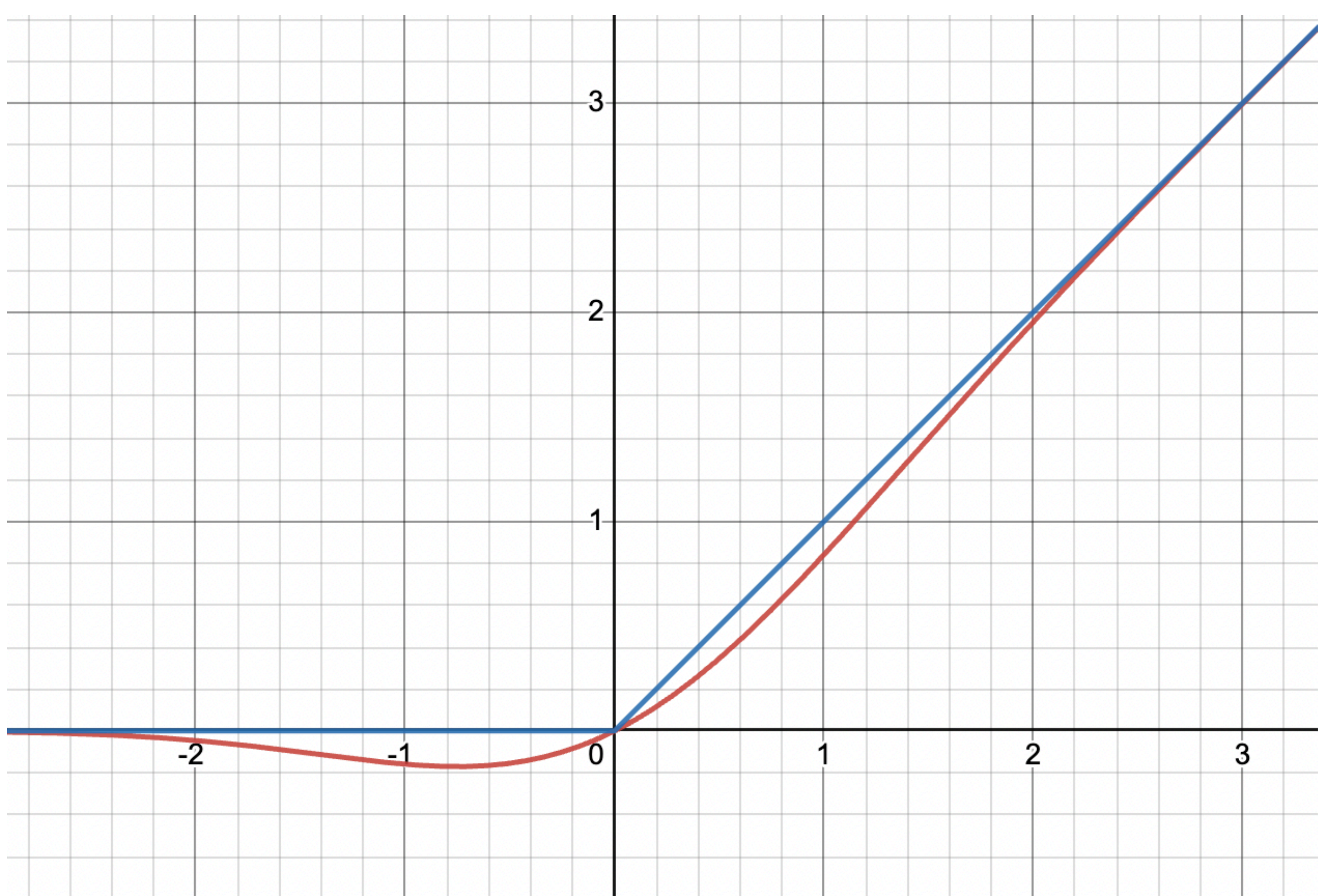
- In BatchNorm, we standardize over each **feature** separately, **across the batch**.
- In LayerNorm, we standardize over each **batch example** separately, across the **features**.



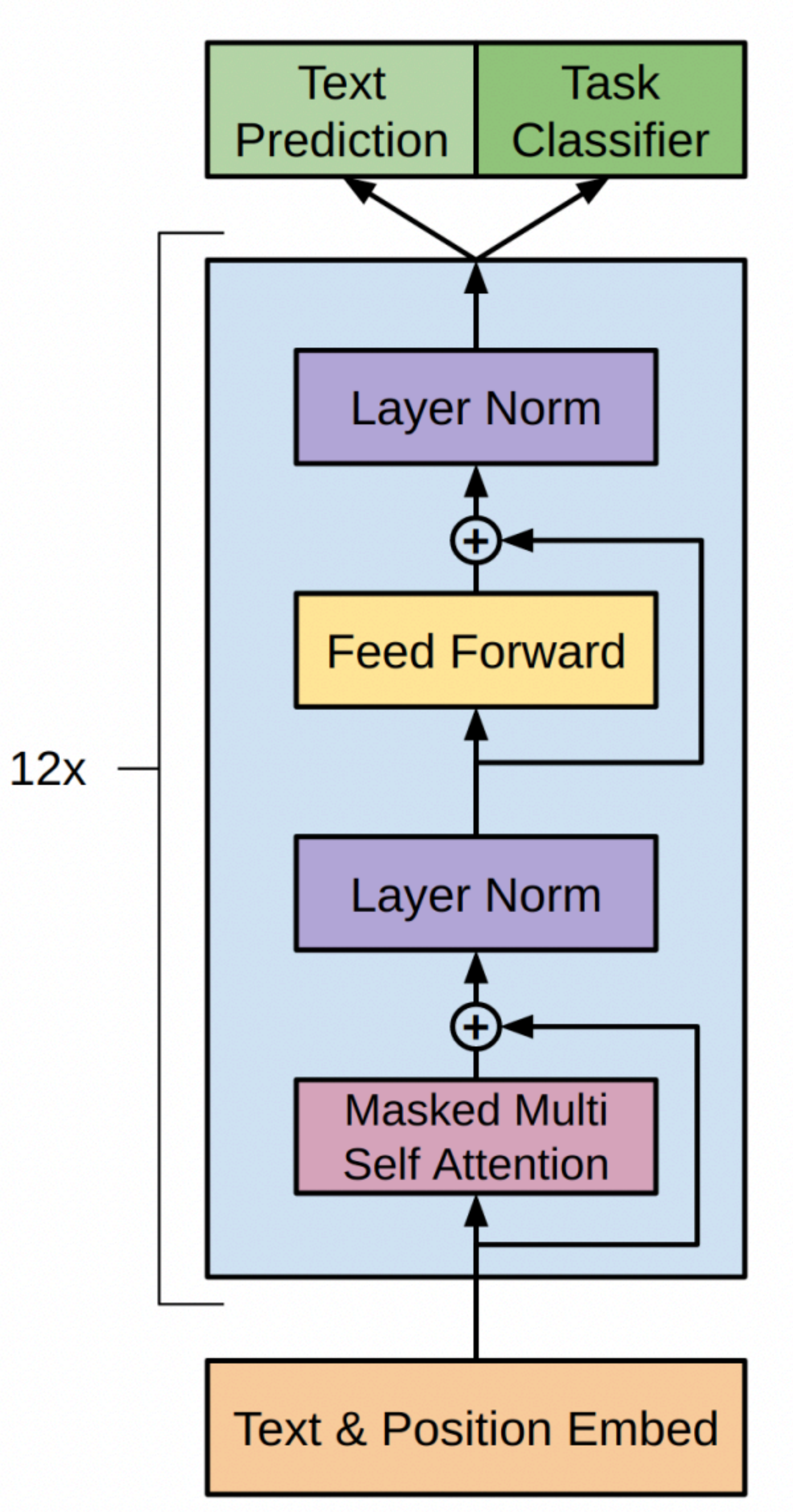
Multilayer Perceptron

- Just a FFN applied to each time-step
  - GPT2: two-layer, with hidden size 4*D.
- GeLU (Gaussian Error Linear Unit) activation [https://arxiv.org/pdf/1606.08415.pdf]

$$\text{GeLU}(x) = x\Phi(x) \approx 0.5x\left(1 + \tanh\left(\sqrt{\left(\frac{2}{\pi}\right)}\left(x + 0.044715x^3\right)\right)\right)$$



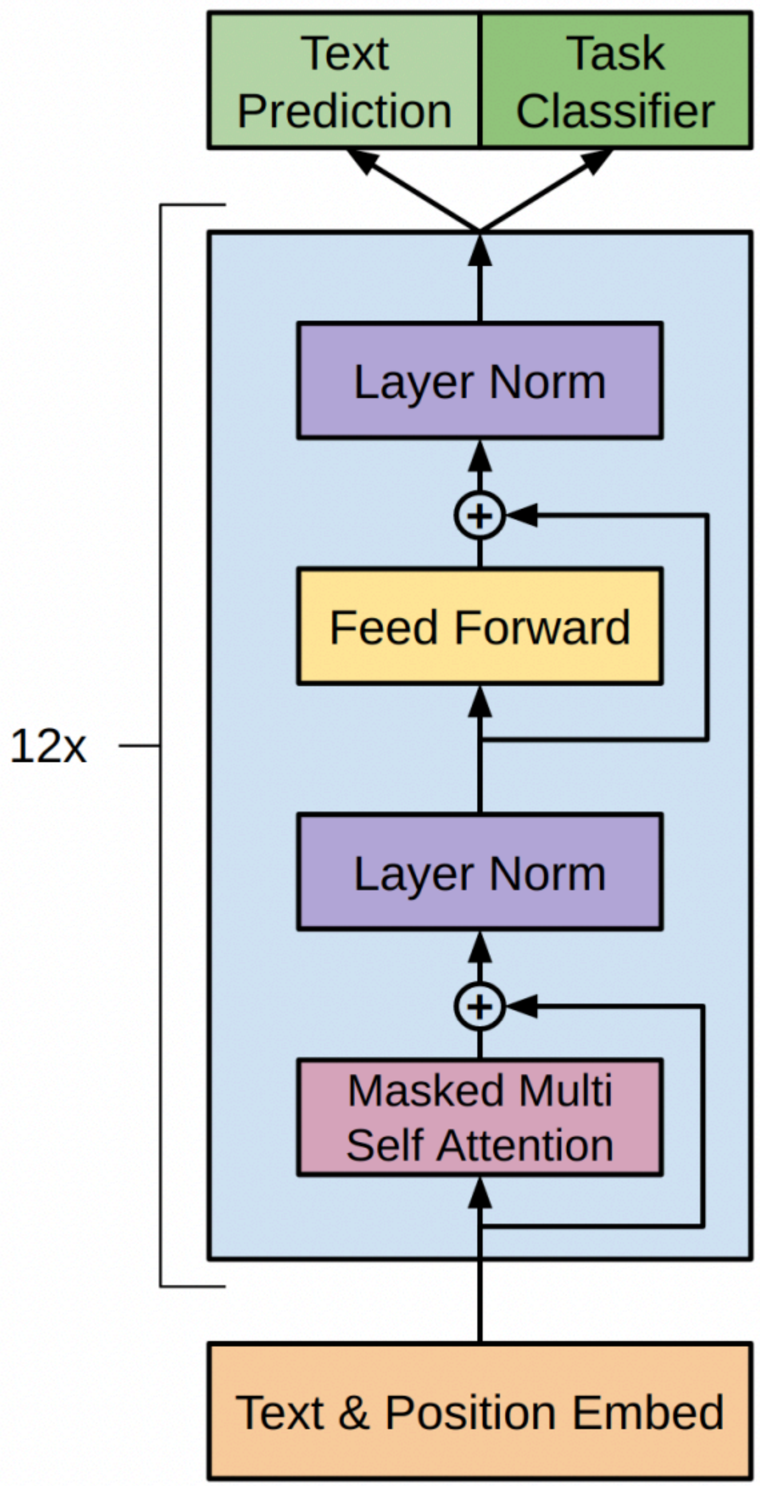
GPT2-Style Decoder



GPT2-Style Decoder

- Input string: “attention is all you need”.
- “attention is all you need” —> [1078, 1463, 318, 477, 345, 761] (indices)
- **Word Token Embeddings (WTE):** Get a (learned) embedding for each token from the embedding table.
- **Word Positional Encodings (WPE):** Get a positional encoding for each position. Can be learned or fixed.
- Sum WPE and WTE for each token. This is the input to the transformer.
- **Apply transformer.**
- **Project onto vocabulary**
- **Apply softmax** to obtain N distribution(s) over next tokens, one for each time-step.
- **Train to minimize cross-entropy loss.**

GPT2-Style Decoder




```
N = len(idx)

# Sum the word embeddings and positional encodings.
wte = self.wte(idx)
wpe = self.wpe(jnp.arange(N))
x = wte + wpe

# Forward through the transformer.
x = self.transformer(x)

# Project onto the vocabulary.
x = jax.vmap(self.vproj)(x)
```