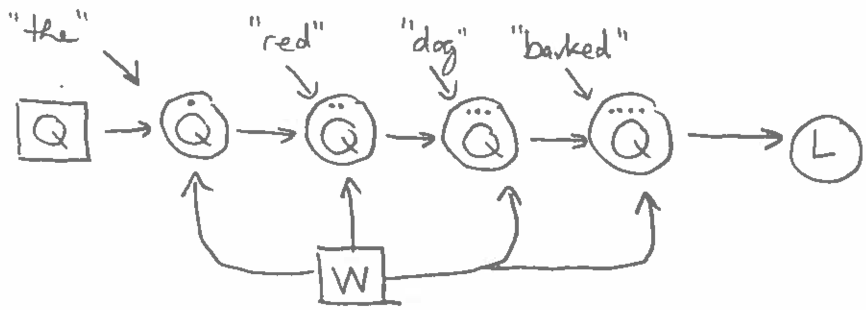
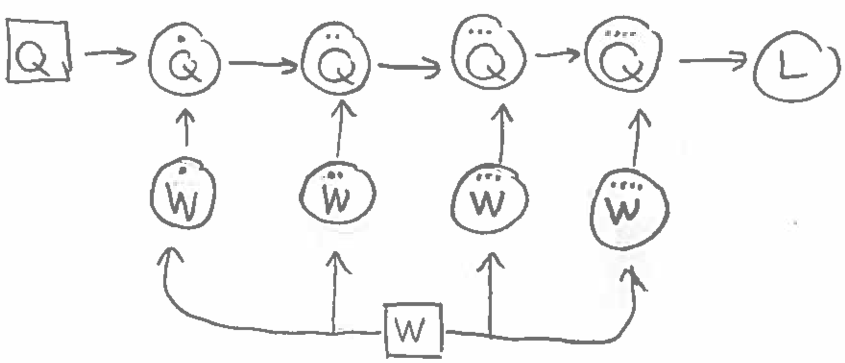
Transformer Networks

① A dominant approach to processing sequences is the RNN:



But one downside to an RNN is what happens when we need to compute partial derivatives for the model weights. Let's first create copies of W and apply the Chain Rule:



So:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \dot{W}}\frac{\partial \dot{W}}{\partial W} + \frac{\partial L}{\partial \ddot{W}}\frac{\partial \ddot{W}}{\partial W} + \frac{\partial L}{\partial \dddot{W}}\frac{\partial \dddot{W}}{\partial W} + \frac{\partial L}{\partial \ddddot{W}}\frac{\partial \ddddot{W}}{\partial W}$$

$$= \frac{\partial L}{\partial \dot{W}} + \frac{\partial L}{\partial \ddot{W}} + \frac{\partial L}{\partial \dddot{W}} + \frac{\partial L}{\partial \ddddot{W}}$$

$$= \frac{\partial L}{\partial \ddddot{Q}}\frac{\partial \ddddot{Q}}{\partial \dddot{Q}}\frac{\partial \dddot{Q}}{\partial \ddot{Q}}\frac{\partial \ddot{Q}}{\partial \dot{Q}}\frac{\partial \dot{Q}}{\partial \dot{W}} + \frac{\partial L}{\partial \ddddot{Q}}\frac{\partial \ddddot{Q}}{\partial \dddot{Q}}\frac{\partial \dddot{Q}}{\partial \ddot{Q}}\frac{\partial \ddot{Q}}{\partial \ddot{W}}$$

$$+ \frac{\partial L}{\partial \ddddot{Q}}\frac{\partial \ddddot{Q}}{\partial \dddot{Q}}\frac{\partial \dddot{Q}}{\partial \dddot{W}} + \frac{\partial L}{\partial \ddddot{Q}}\frac{\partial \ddddot{Q}}{\partial \ddddot{W}}$$

TRANSFORMER NETWORKS

2) Thus, it takes linear time to compute (linear in the length of the sequence).

I mean, that seems ok, since processing a sequence always takes time linear in the sequence, so what's the big deal?

---

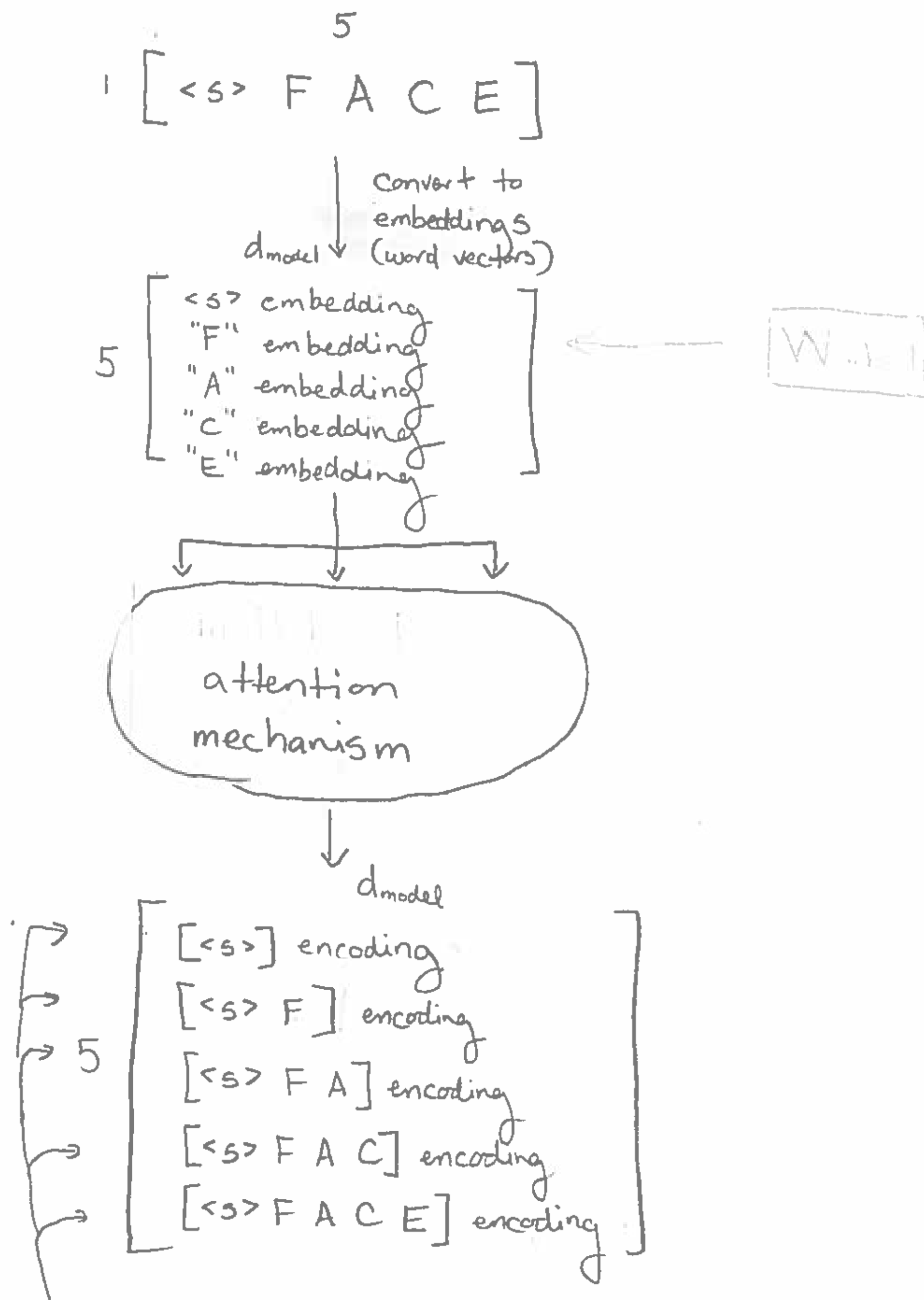
3) Well, the downside isn't that it takes linear time to compute $\frac{\partial L}{\partial W^{(k)}}$ for all k.

The downside is that it takes linear time to compute just $\frac{\partial L}{\partial \dot{W}}$. This means that if we want to parallelize the computation, and compute $\frac{\partial L}{\partial W^{(k)}}$ on k processors, it'll still take linear time.

TRANSFORMER NETWORKS

(4) A "transformer" network is a way to encode a variable-length sequence x of length n such that we do not need to sequentially encode each prefix of x in order to encode x.

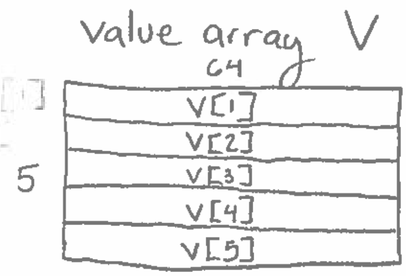
(5) At a high level, it looks as follows:



All sequence prefixes are encoded in a single forward pass!

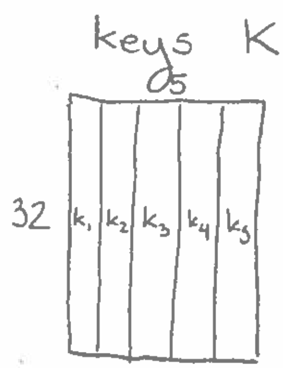
(6) Well, what does this magical "attention mechanism" do that manages to encode the prefixes in a parallelizable way?

At its core, it tries to emulate a dictionary (or hash table). Imagine we have an array of 5 "values", where each value is a length-64 real vector:



This array can be stored as a 5×64 matrix.

(7) In addition, suppose that each array position is associated with a "key", which is a length-32 real vector.



We can store the 5 keys as a 32×5 matrix.

⑧ To "look up" a value that corresponds to key k_j (i.e. $V[j]$), we can do it with matrix operations. Suppose a query $q = k_4^T$.

(a) multiply q by K:

$$\begin{array}{c} \\ 1 \end{array}\overset{32}{\begin{bmatrix} \quad k_4 \quad \end{bmatrix}} \; \begin{array}{c} \\ 32 \end{array}\overset{5}{\begin{bmatrix} k_1 & k_2 & k_3 & k_4 & k_5 \end{bmatrix}} = \begin{bmatrix} k_4 \cdot k_1 & k_4 \cdot k_2 & k_4 \cdot k_3 & k_4 \cdot k_4 & k_4 \cdot k_5 \end{bmatrix}$$

(b) $k_4 \cdot k_4 = \|k_4\|^2$ is the maximal value of the resulting vector. By applying softmax, we can further magnify the differences to obtain an "attention vector" A:

$$\text{softmax}\left(\begin{bmatrix} k_4 \cdot k_1 & k_4 \cdot k_2 & k_4 \cdot k_3 & k_4 \cdot k_4 & k_4 \cdot k_5 \end{bmatrix}\right) \approx \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

(c) now we can retrieve $V[4]$ by multiplying A by V:

$$\begin{array}{c} 1 \end{array}\overset{5}{\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \end{bmatrix}} \; \begin{array}{c} \\ \\ 5 \\ \\ \\ \end{array}\overset{64}{\begin{bmatrix} V[1] \\ V[2] \\ V[3] \\ V[4] \\ V[5] \end{bmatrix}} = \begin{bmatrix} \quad V[4] \quad \end{bmatrix}$$

TRANSFORMER NETWORKS

9) This mechanism also allows us to "hash" any query q to a corresponding array position. Observe:

(a)
$$\underset{1}{\overset{32}{\boxed{\quad q \quad}}} \begin{array}{|c|c|c|c|c|} \hline k_1 & k_2 & k_3 & k_4 & k_5 \\ \hline \end{array} = \begin{array}{|c|c|c|c|c|} \hline q\cdot k_1 & q\cdot k_2 & q\cdot k_3 & q\cdot k_4 & q\cdot k_5 \\ \hline \end{array}$$

(b)
$$\text{softmax}\left(\begin{array}{|c|c|c|c|c|} \hline q\cdot k_1 & q\cdot k_2 & q\cdot k_3 & q\cdot k_4 & q\cdot k_5 \\ \hline \end{array} \right) \approx \begin{array}{|c|c|c|c|c|} \hline 0 & 1 & 0 & 0 & 0 \\ \hline \end{array}$$

if q is closer to k_2 than to the other keys

(c)
$$\underset{1}{\overset{5}{\begin{array}{|c|c|c|c|c|} \hline 0 & 1 & 0 & 0 & 0 \\ \hline \end{array}}} \quad 5 \overset{64}{\begin{array}{|c|} \hline V[1] \\ \hline V[2] \\ \hline V[3] \\ \hline V[4] \\ \hline V[5] \\ \hline \end{array}} = \begin{array}{|c|} \hline \quad V[2] \quad \\ \hline \end{array}$$

10) Note also that some queries will return, rather than a single array element, a linear combination of array elements, e.g. suppose

$$q\cdot k_1 = 2.0 \qquad q\cdot k_4 = 1.8$$
$$q\cdot k_2 = 2.4 \qquad q\cdot k_5 = -12.5$$
$$q\cdot k_3 = -10$$

Then:

$$\text{softmax}\left(\begin{array}{|c|c|c|c|c|} \hline q\cdot k_1 & q\cdot k_2 & q\cdot k_3 & q\cdot k_4 & q\cdot k_5 \\ \hline \end{array} \right) \approx \begin{array}{|c|c|c|c|c|} \hline .30 & .45 & 0 & .25 & 0 \\ \hline \end{array}$$

and

$$\begin{array}{|c|c|c|c|c|} \hline .3 & .45 & 0 & .25 & 0 \\ \hline \end{array} \begin{array}{|c|} \hline V[1] \\ \hline V[2] \\ \hline V[3] \\ \hline V[4] \\ \hline V[5] \\ \hline \end{array} = \begin{array}{|c|} \hline .3V[1] + .45V[2] + .25V[4] \\ \hline \end{array}$$

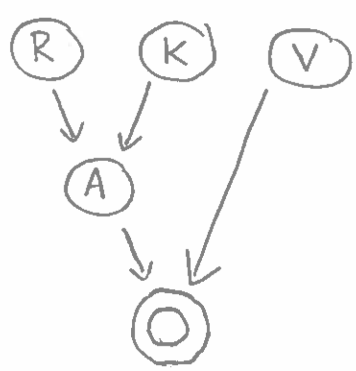
(11) It's straightforward to extend this mechanism to handle multiple queries at once by stacking query vectors into a query matrix:

(a)
$$\begin{array}{cc} & 32 \\ 3 & \begin{bmatrix} q_1 \\ \hline q_2 \\ \hline q_3 \end{bmatrix} \end{array} \begin{array}{cc} & 5 \\ 32 & \begin{bmatrix} k_1 & k_2 & k_3 & k_4 & k_5 \end{bmatrix} \end{array} = \begin{array}{cc} & 5 \\ 3 & \begin{bmatrix} q_1\cdot k_1 & q_1\cdot k_2 & q_1\cdot k_3 & q_1\cdot k_4 & q_1\cdot k_5 \\ q_2\cdot k_1 & q_2\cdot k_2 & q_2\cdot k_3 & q_2\cdot k_4 & q_2\cdot k_5 \\ q_3\cdot k_1 & q_3\cdot k_2 & q_3\cdot k_3 & q_3\cdot k_4 & q_3\cdot k_5 \end{bmatrix} \end{array}$$

(b)
$$\begin{bmatrix} \text{softmax}\left(\begin{bmatrix} q_1\cdot k_1 & q_1\cdot k_2 & q_1\cdot k_3 & q_1\cdot k_4 & q_1\cdot k_5 \end{bmatrix}\right) \\ \text{softmax}\left(\begin{bmatrix} q_2\cdot k_1 & q_2\cdot k_2 & q_2\cdot k_3 & q_2\cdot k_4 & q_2\cdot k_5 \end{bmatrix}\right) \\ \text{softmax}\left(\begin{bmatrix} q_3\cdot k_1 & q_3\cdot k_2 & q_3\cdot k_3 & q_3\cdot k_4 & q_3\cdot k_5 \end{bmatrix}\right) \end{bmatrix} \underset{\approx}{\text{e.g.}} \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

(c)
$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} V[1] \\ V[2] \\ V[3] \\ V[4] \\ V[5] \end{bmatrix} = \begin{bmatrix} V[2] \\ V[1] \\ V[5] \end{bmatrix}$$

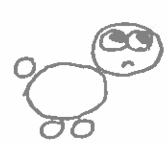
(12) Expressed as a causal diagram, the attention mechanism is:



$$A = \text{softmax}(RK,\ \dim = 1)$$

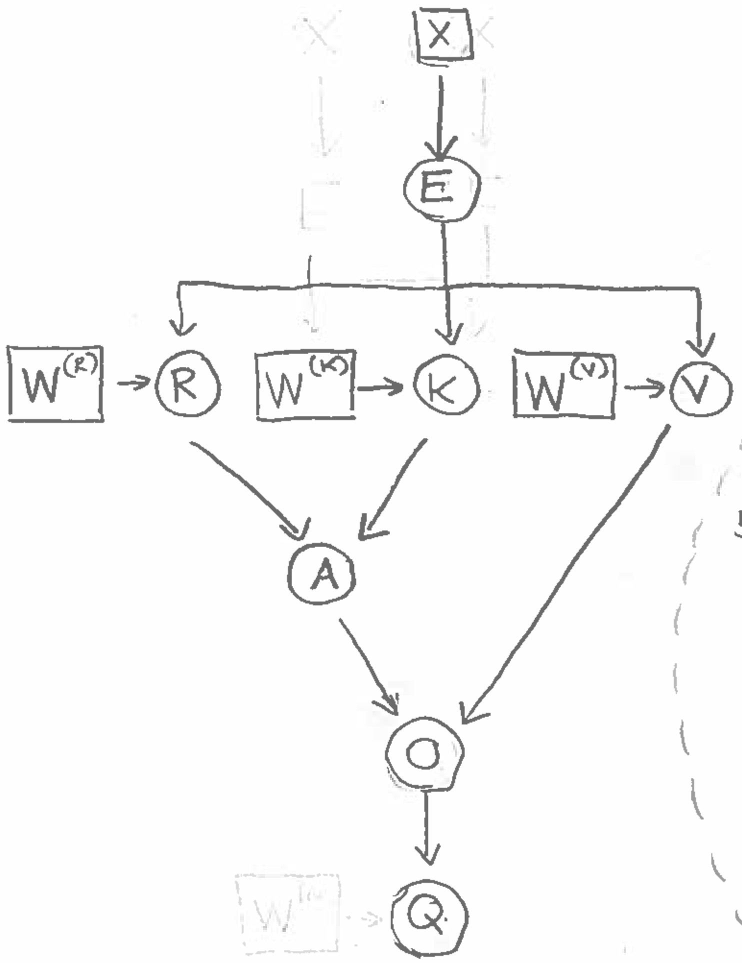
$$O = AV$$

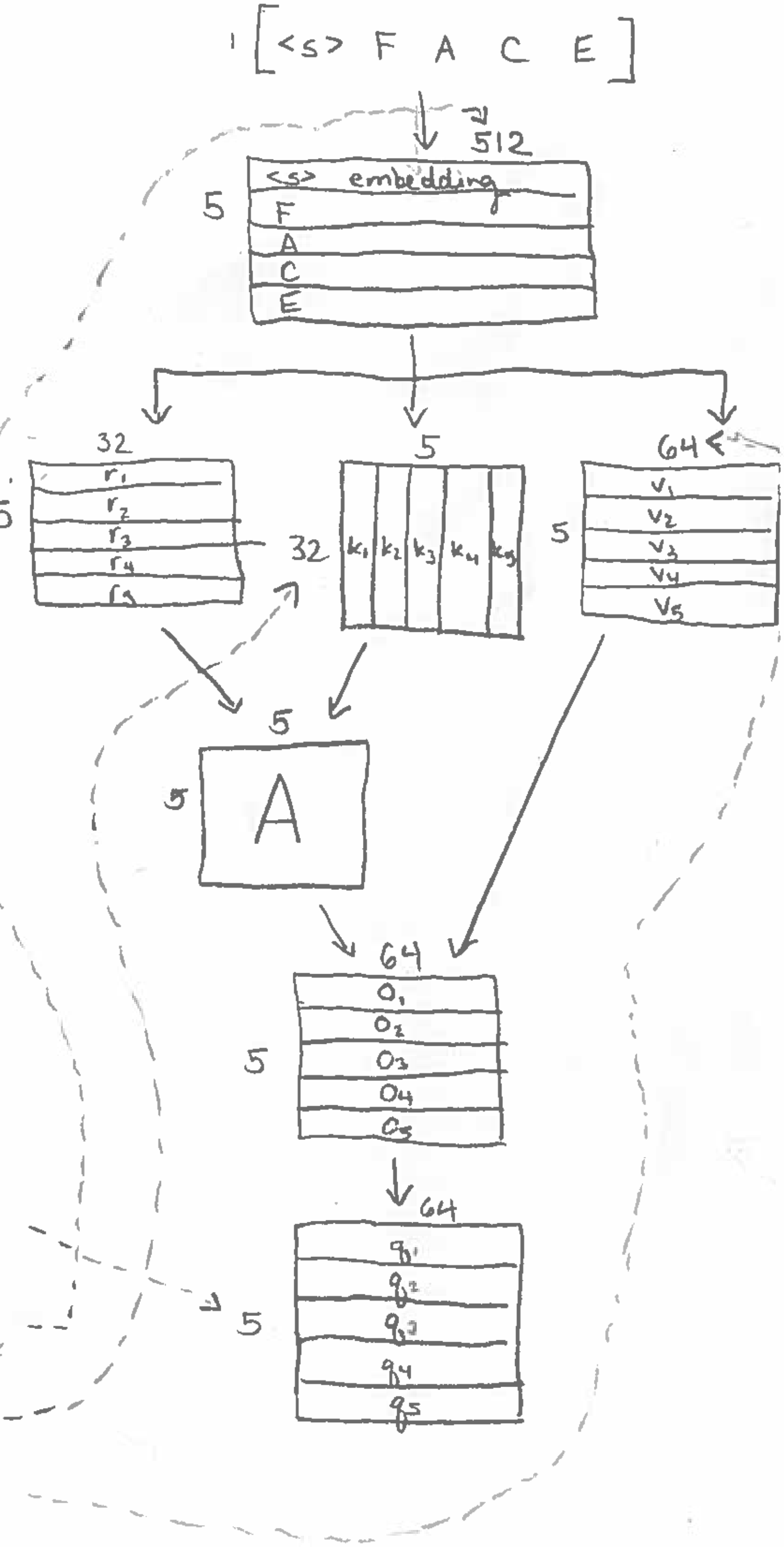
this means we apply softmax to each row



Transformer Networks

(13) Let's see what happens if we put this attention mechanism into our encoder:





There are four "dimensionalities" in the encoder:

$|X|$: length of the input seq

d_{model} : length of the embedding

d_{key} : length of each "key"

d_{value} : length of each "value"

TRANSFORMER NETWORKS

(14) In the causal diagram, no matter what the size of the input sequence, the longest path has only 6 nodes:

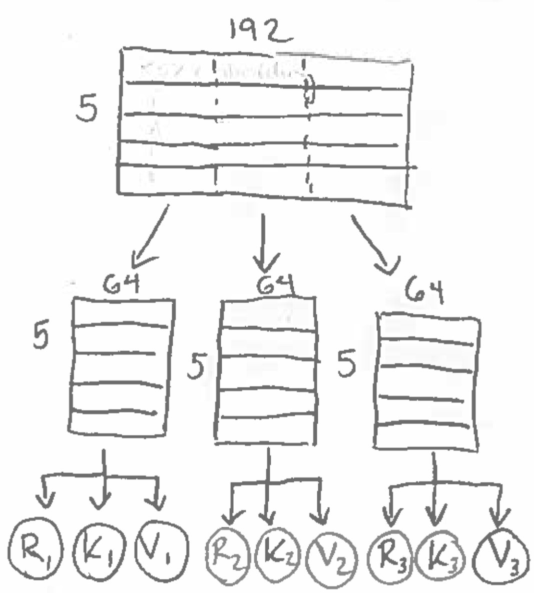
$$\boxed{X} \to E \to R \to A \to O \to Q$$

Contrast this to the RNN encoder, whose longest path was linear in the length of the input, e.g.

$$\boxed{Q} \to \overset{1}{Q} \to \overset{2}{Q} \to \cdots \to \overset{n}{Q}$$

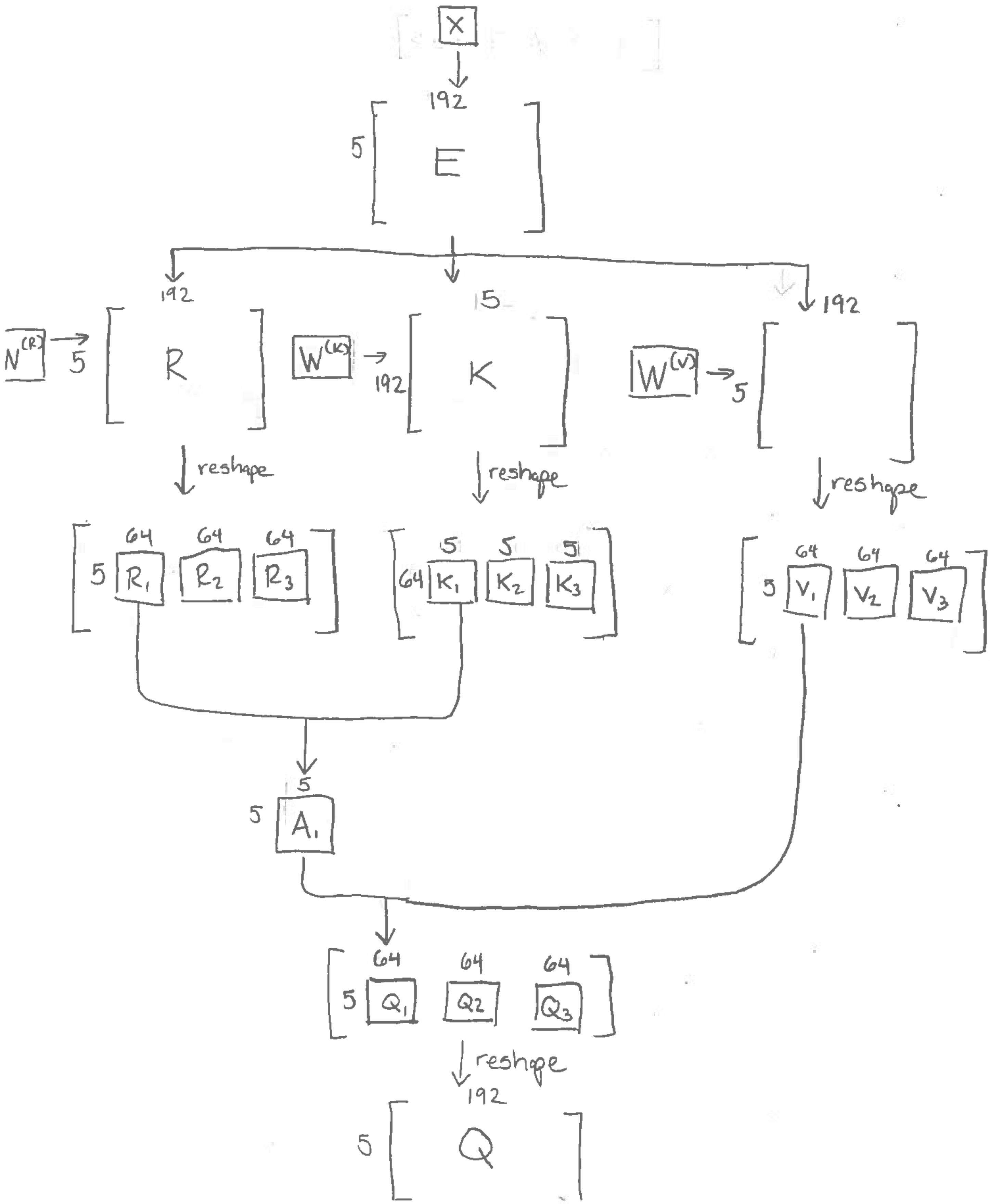
Hence, we can compute any quantity of the transformer encoder in constant time, while computing an RNN state variable requires linear time.

(15) A generalized version of the transformer's attention mechanism splits the embedding matrix into equal-sized chunks and runs a separate attention mechanism on each chunk; e.g.:



TRANSFORMER NETWORKS

(16) This "multi-head" attention mechanism can be efficiently implemented with tensor operations:



(17) Next, let's take a closer look at the embedding process:

$$\begin{bmatrix} \langle s \rangle & F & A & C & E \end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix} \langle s \rangle \text{ embedding} \\ \text{"F" embedding} \\ \text{"A" embedding} \\ \text{"C" embedding} \\ \text{"E" embedding} \end{bmatrix}$$

(18) One straightforward approach is to just use pretrained word vectors from software like word2vec. However, the transformer network does not encode anything about the position of the words, thus:

$$\begin{bmatrix} \langle s \rangle \text{ embedding} \\ F \\ A \\ C \\ E \end{bmatrix} \text{ is treated just like } \begin{bmatrix} \langle s \rangle \text{ embedding} \\ C \\ A \\ F \\ E \end{bmatrix}$$

This "bag-of-words" assumption is problematic for applications for which word order matters.

TRANSFORMER NETWORKS

(19) Looking at the transpose of the embedding matrix:

$$\begin{array}{ccc} \langle s \rangle \text{ embedding } \downarrow & & \swarrow \text{ "E" embedding} \\ & \begin{bmatrix} 0.8 & 2.2 & -0.1 & -2.8 & 0.0 \\ -0.4 & 2.1 & -0.5 & -2.4 & 0.0 \\ 1.2 & 0.0 & 3.0 & 0.0 & -1.8 \end{bmatrix} & \end{array}$$

we could try to encode "position" by adding some function of the position to each dimension:

$$\begin{bmatrix} 0.8 + f_1(1) & 2.2 + f_1(2) & -0.1 + f_1(3) & -2.8 + f_1(4) & 0.0 + f_1(5) \\ -0.4 + f_2(1) & 2.1 + f_2(2) & -0.5 + f_2(3) & -2.4 + f_2(4) & 0.0 + f_2(5) \\ 1.2 + f_3(1) & 0.0 + f_3(2) & 3.0 + f_3(3) & 0.0 + f_3(4) & -1.8 + f_3(5) \end{bmatrix}$$

(20) A common way to implement this "positional encoding" is to use sinusoids of different frequencies and offsets:

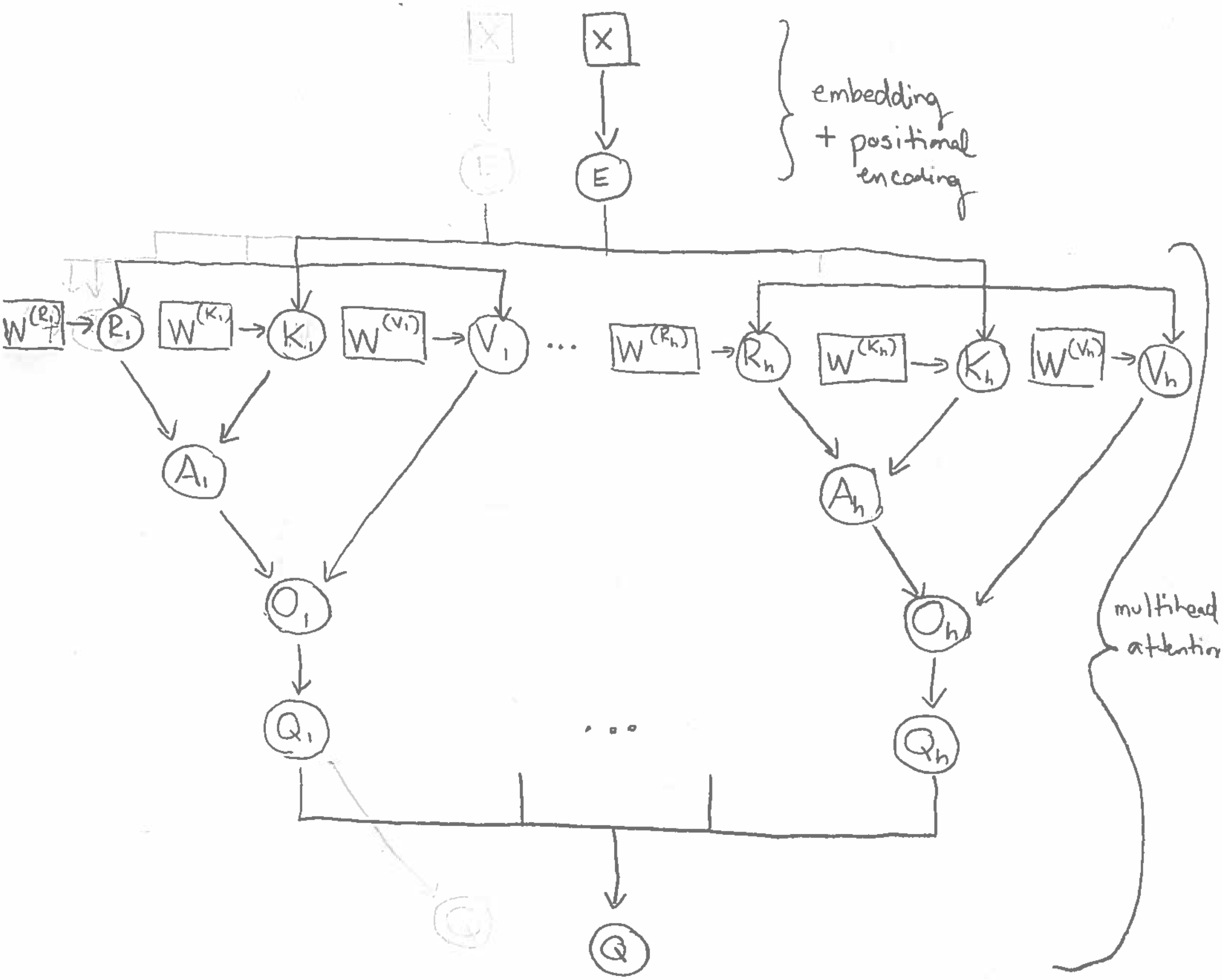
$$\begin{matrix} 0.8 & 2.2 & -0.1 & -2.8 & 0.0 \\ +0 & +1 & +0 & -1 & +0 \end{matrix}$$

$$\begin{matrix} -0.4 & 2.1 & -0.5 & -2.4 & 0.0 \\ +1 & +0 & -1 & 0 & +1 \end{matrix}$$

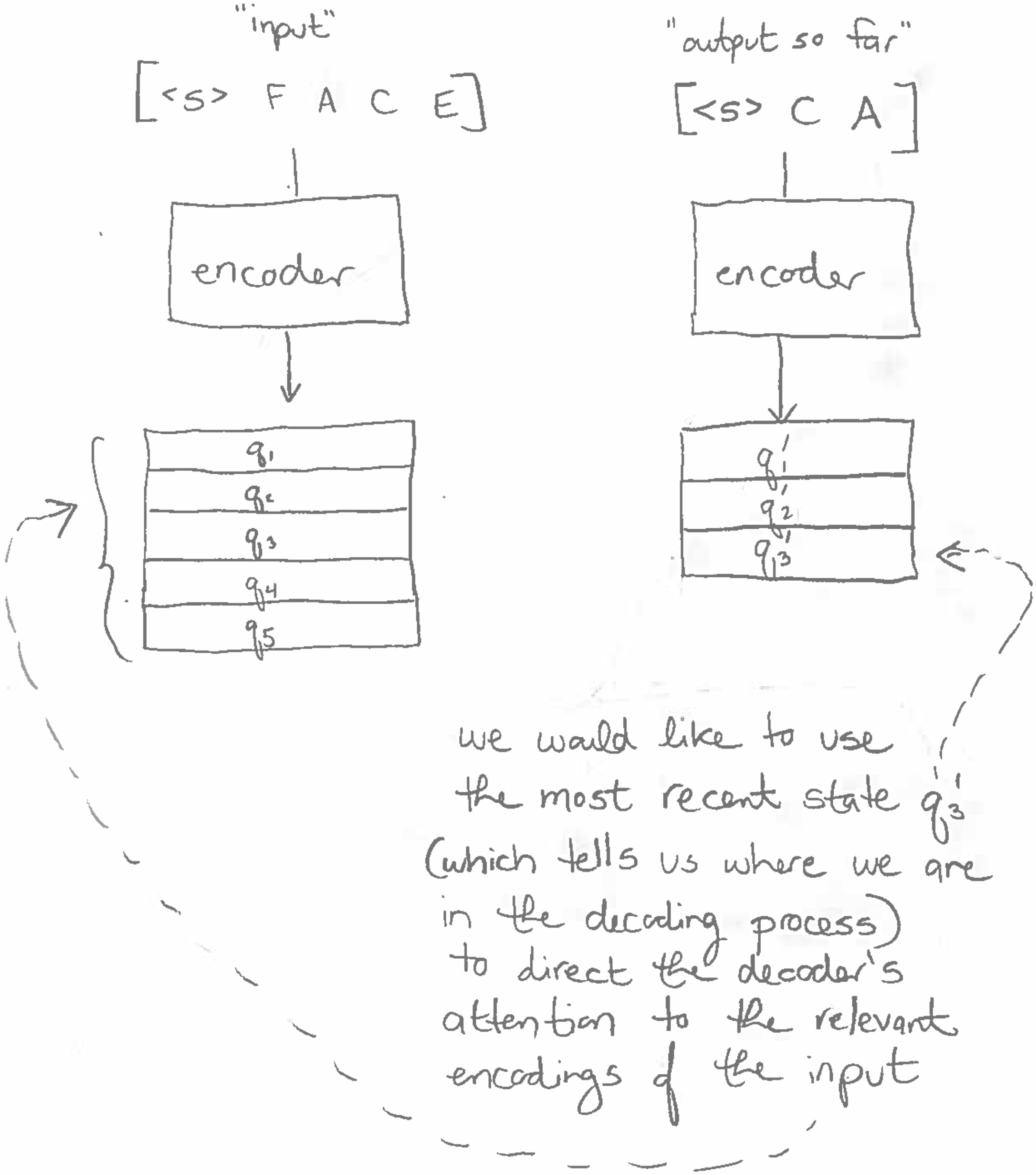
etc.

TRANSFORMER NETWORKS

21) Adding positional encodings to our overall picture, we have:

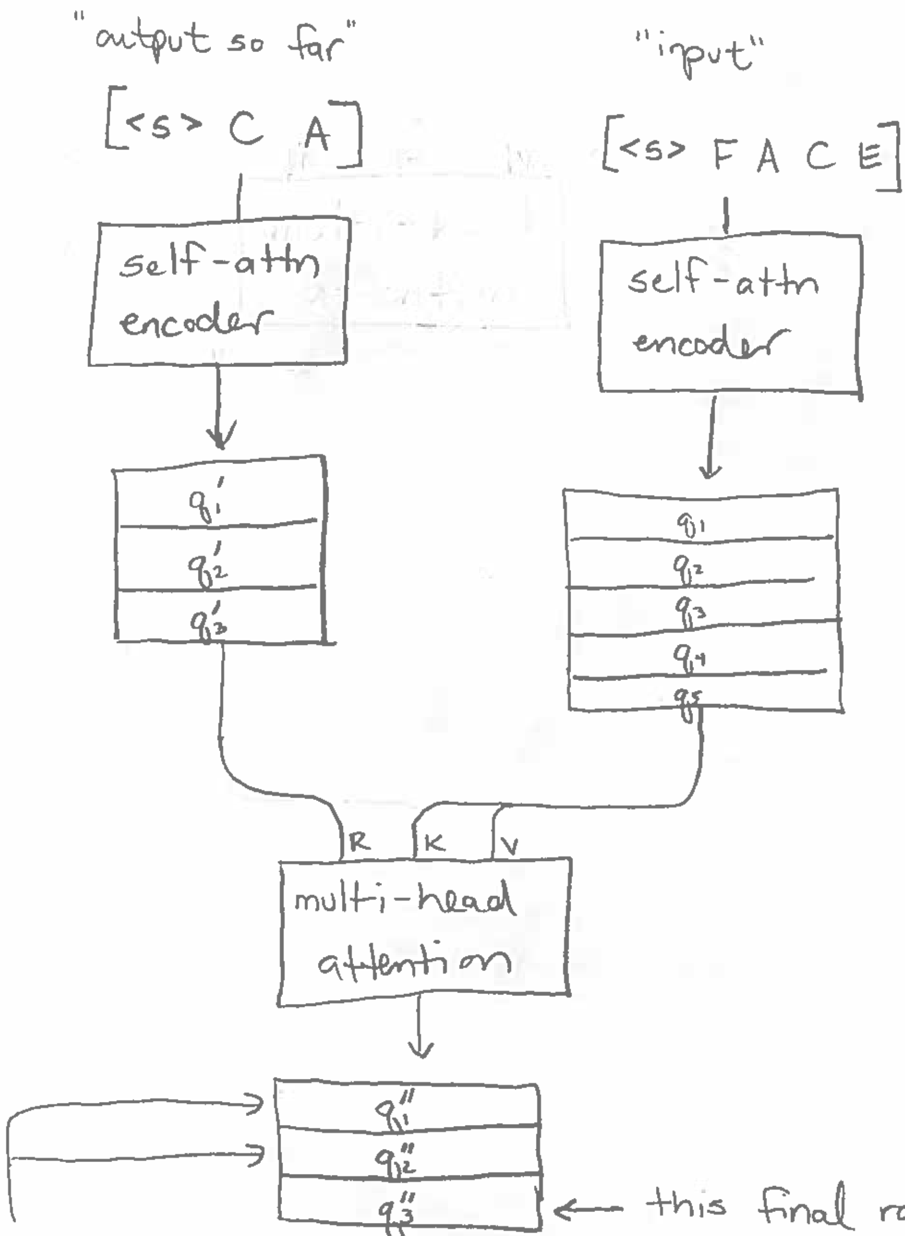


(22) Now suppose we want to use a transformer for translation. When "decoding" (producing the output language), we want to take into consideration both the input encodings and the output generated so far:



TRANSFORMER NETWORKS

(23) Again we use the multi-head attention mechanism, but rather than use the input as our request, key, and value array, we use the "output so far" as the request:

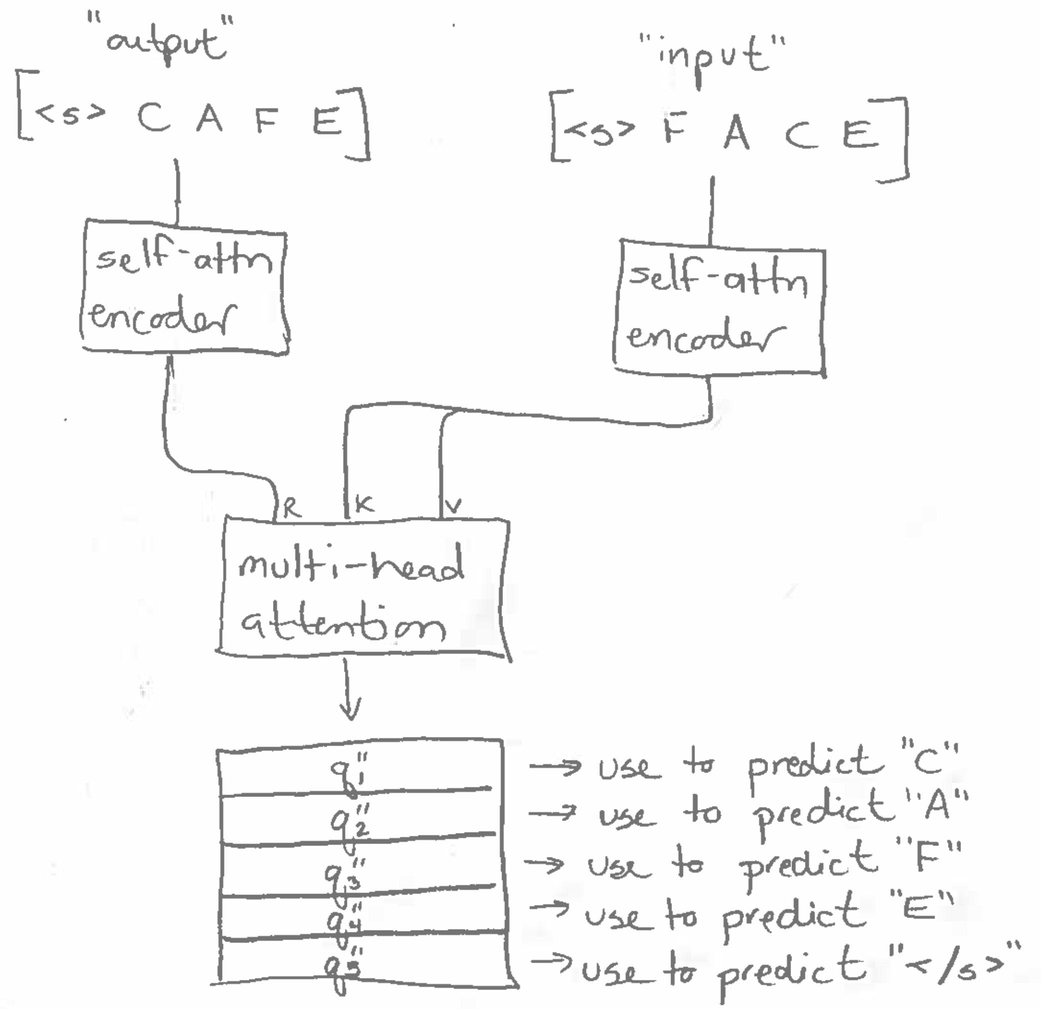


these states have already been used for prediction, and are sort of trash at this point

this final raw is then used to predict the next letter of the translation (using a linear layer + softmax)

TRANSFORMER NETWORKS

(24) During training, we know the entire output in advance, so for efficiency we could do the predictions all at once:



(25) This requires a small tweak to the output's self-attn encoder, so that earlier output states cannot "see" future output states.

Transformer Networks

(26) This modification is a "mask" matrix that zeroes out information coming from future output (compare w (16)):