Attention

Lecture 4
EN.705.743: ChatGPT from Scratch

Lecture Outline

Attention

- Recap
- RNN Inspiration
- Self-Attention (simplified)
- Queries, Keys, and Values
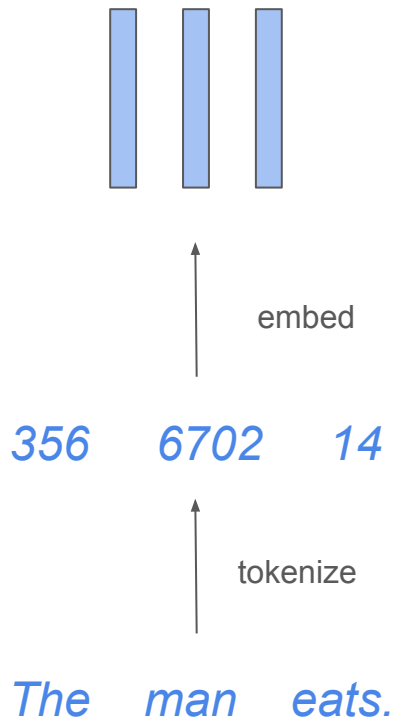- Multi-head Attention

Recap

Recap

Starting with text, we convert the text into integer ids (tokenizer).

For each id, we learn a vector representation (embedding).

Now our text has been converted into vectors, and we need to build a model that can perform computations on sequences of vectors.

(3 x N) matrix
Note: I use long rectangles as "vectors"

embed

356   *6702*   *14*

tokenize
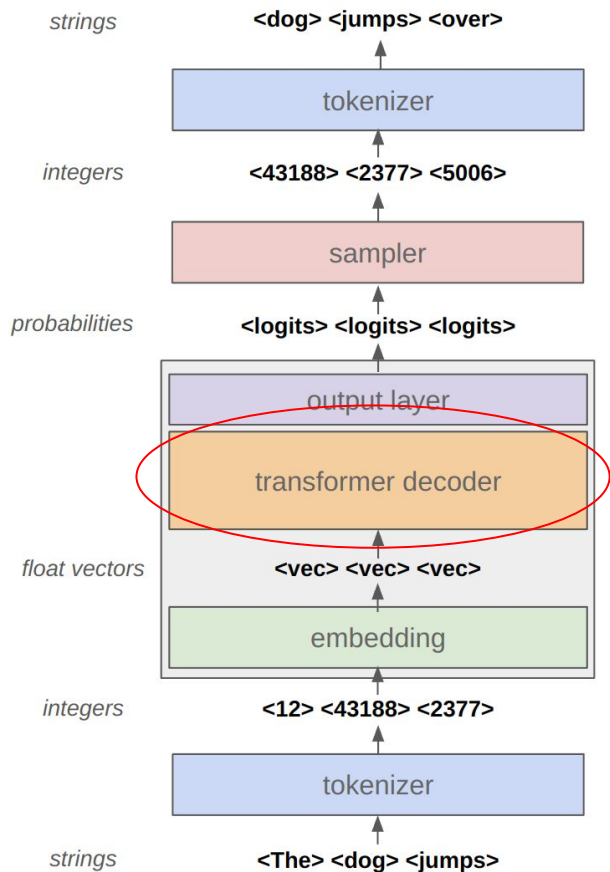
The   man   eats.

Recap

Starting with text, we convert the text into integer ids (tokenizer).

For each id, we learn a vector representation (embedding).

Now our text has been converted into vectors, and we need to build a model that can perform computations on sequences of vectors.

strings   **<dog> <jumps> <over>**

| tokenizer |

integers   **<43188> <2377> <5006>**

| sampler |

probabilities   **<logits> <logits> <logits>**

| output layer |

| transformer decoder |

float vectors   **<vec> <vec> <vec>**

| embedding |

integers   **<12> <43188> <2377>**

| tokenizer |

*strings*   **<The> <dog> <jumps>**

WARNING

This is the most math-heavy lecture. There is a lot of tensor manipulation going on.

If you get lost, come back to these slides in your own time. Sometimes it is better to think about these diagrams without my voice buzzing in your ear :)
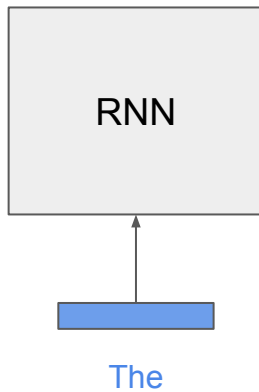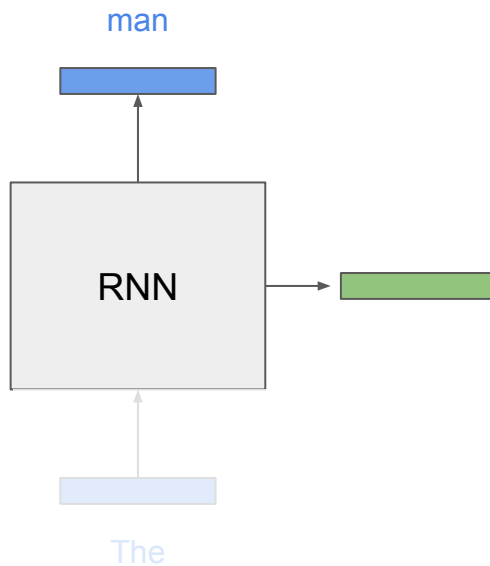
RNNs

How to process a sequence of vectors?

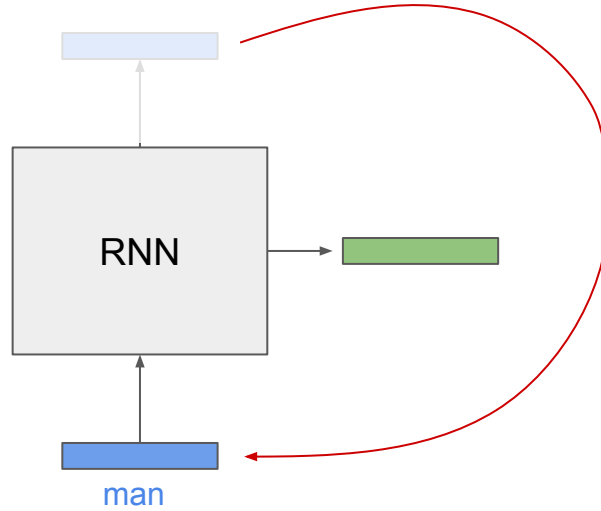Before transformers, the common practice was to use a recurrent neural network (RNN):
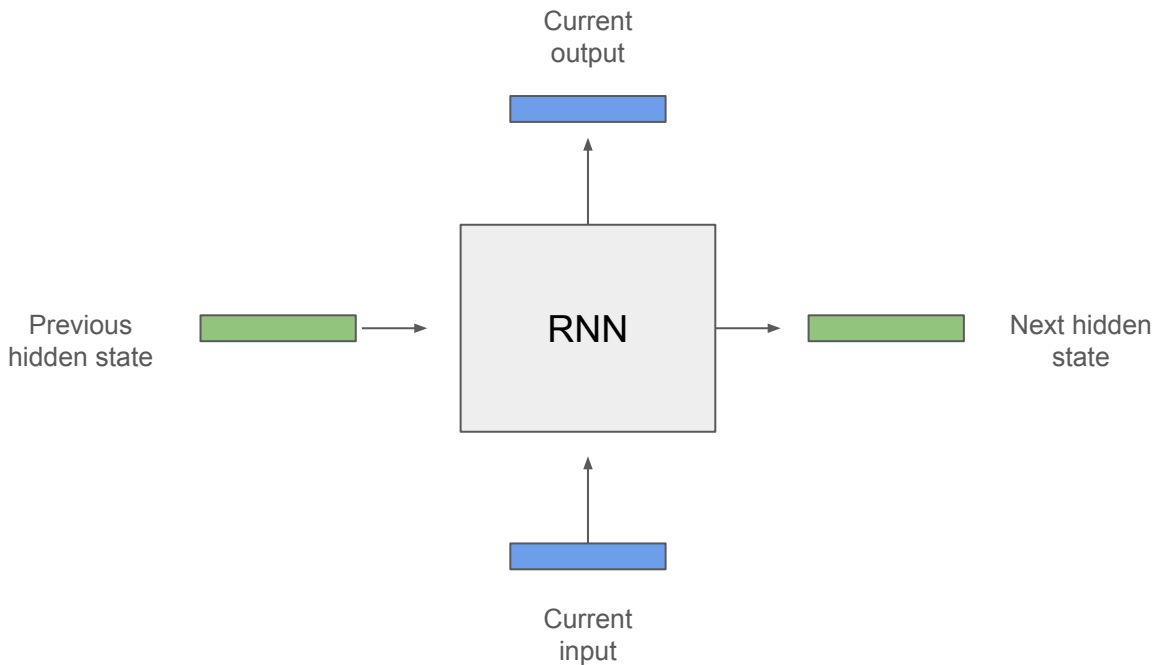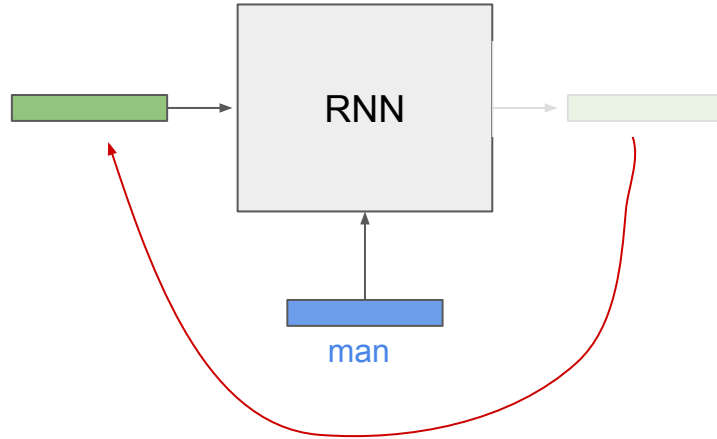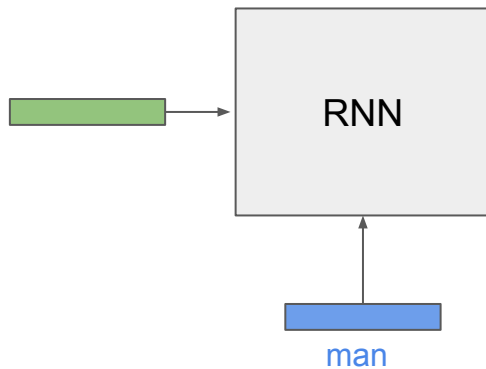
RNN Example: Text Generation

RNN Example: Text Generation

RNN Example: Text Generation

RNN Example: Text Generation

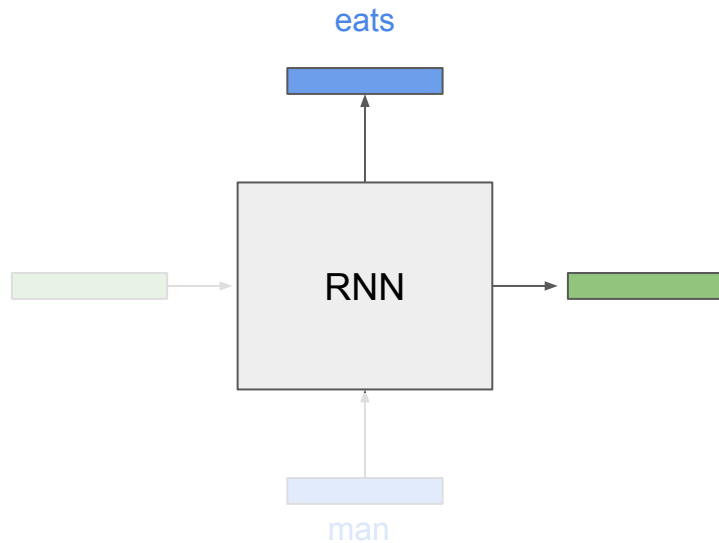# RNN Example: Text Generation

RNN Example: Text Generation

eats
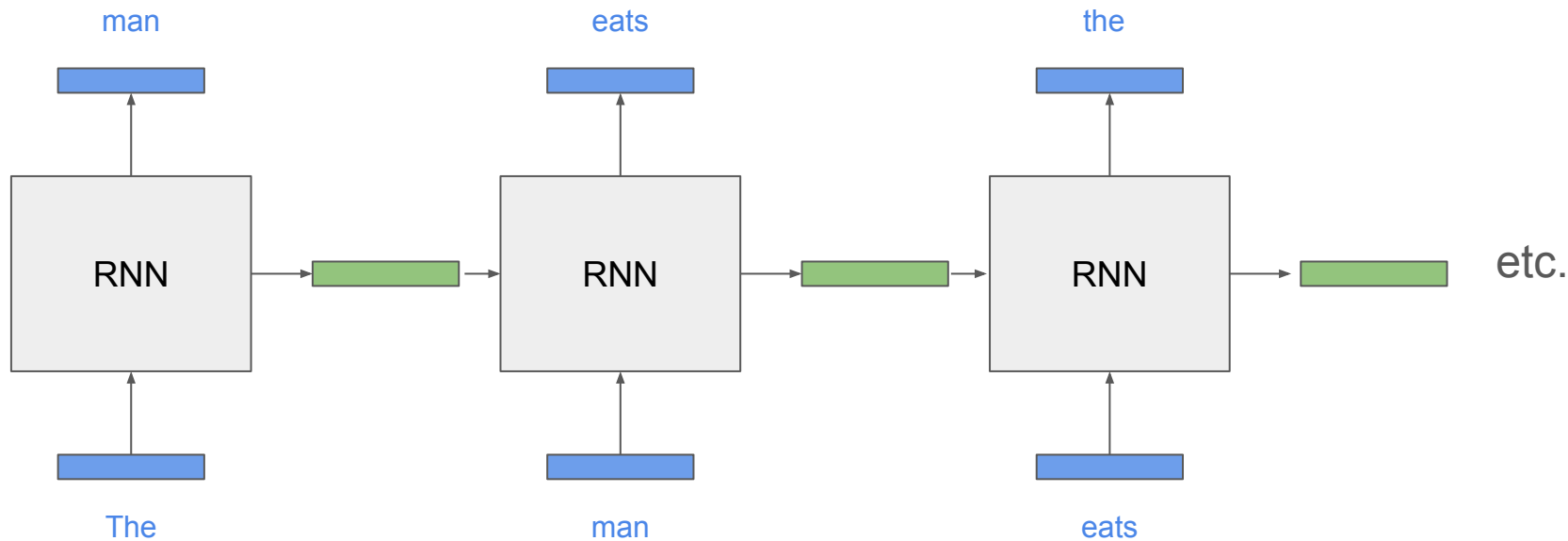
RNN

man

RNN Example: Text Generation "Unrolled" View

The RNN is the same model shown multiple times. We are maintaining a hidden state (green) which is kept between passes through the model.

RNN Example: Translation

In this case, we need to ingest multiple words before we output anything. A good example is translation:

RNN Example: Translation

In this case, we need to ingest multiple words before we output anything. A good example is translation:

Hidden state is changed 3 times.

Mein   Name   ist   Edward

RNN → RNN → RNN → RNN → RNN → RNN → RNN → RNN

My   name   is   Edward

There is a hidden state being passed between steps. For the output "Mein", the hidden state needs to reference "My" from four steps ago.
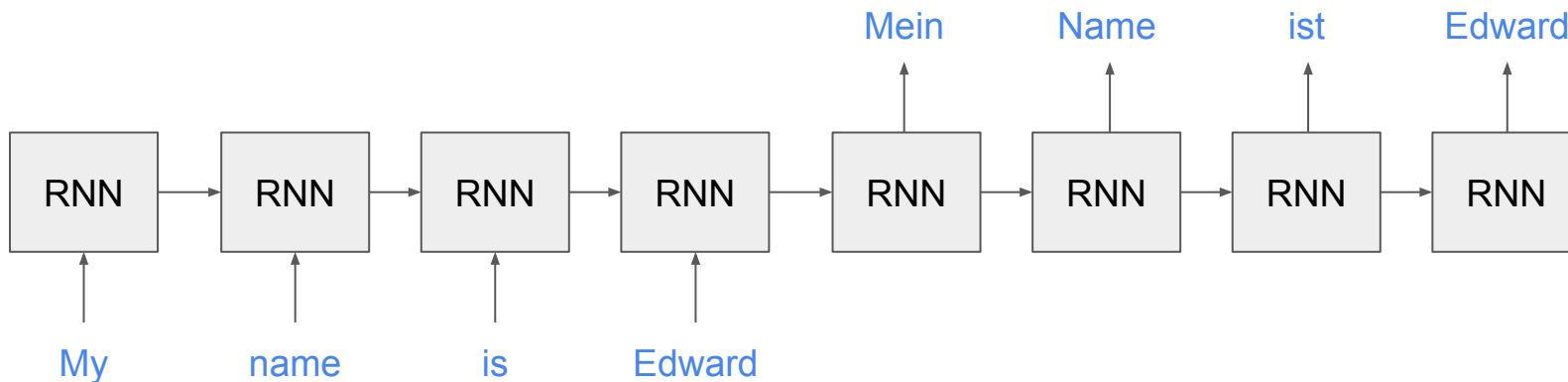
RNN Problems

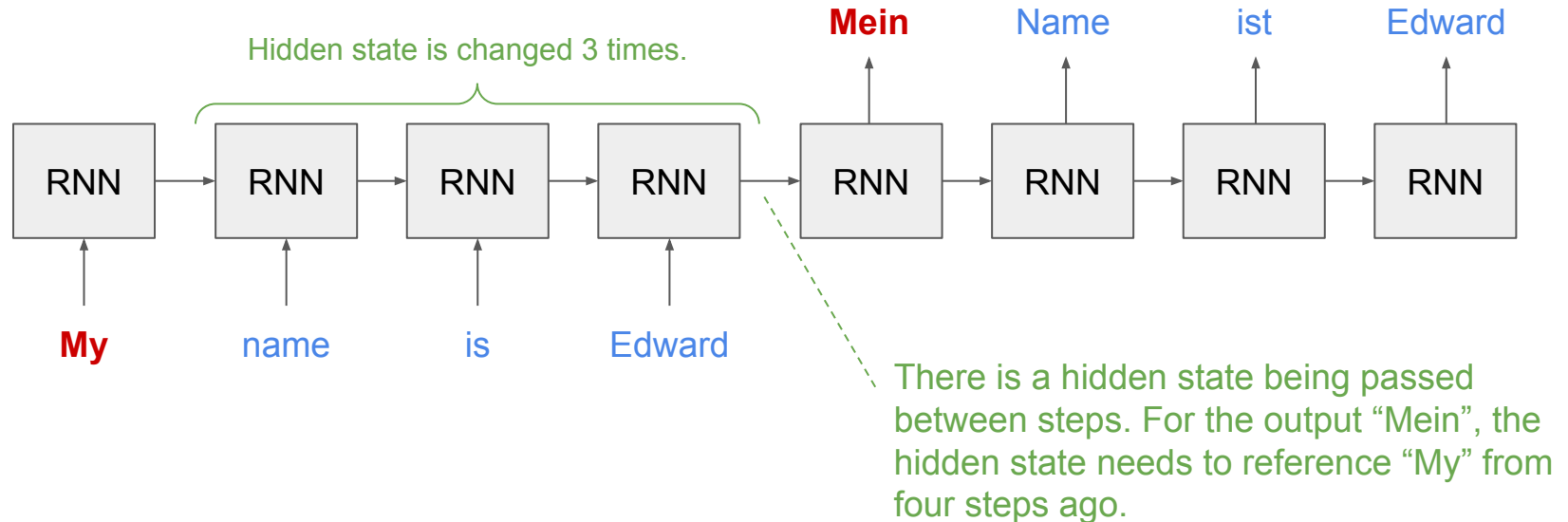This issue of "referencing" past words becomes a problem when text gets really long. Imagine processing this text:

Catherine II[a] (born Princess Sophia Augusta Frederica von Anhalt-Zerbst; 2 May 1729 – 17 November 1796),[b] most commonly known as **Catherine the Great**,[c] was the reigning empress of Russia from 1762 to 1796.[1] She came to power after overthrowing her husband, Peter III. Under her long reign, inspired by the ideas of the Enlightenment, Russia experienced a renaissance of culture and sciences, which led to the founding of many new cities, universities, and theatres, along with a large-scale immigration from the rest of Europe and with the recognition of Russia as one of the great powers of Europe. In **her** accession to power and her rule of the empire, Catherine often relied on her noble favourites, most notably Count Grigory Orlov and Grigory Potemkin.

The understand "her" (red), our hidden state would need to have held on to some concept of "Catherine the Great" from ~80 passes ago. (Or, imagine having to ingest all of this text before translating!)

Self-Attention (Simplified)

RNN "comparison" view

Another way to understand this problem is that we need to be able to make comparisons between words that may be separated by some distance.

(all previous words)

RNN

her
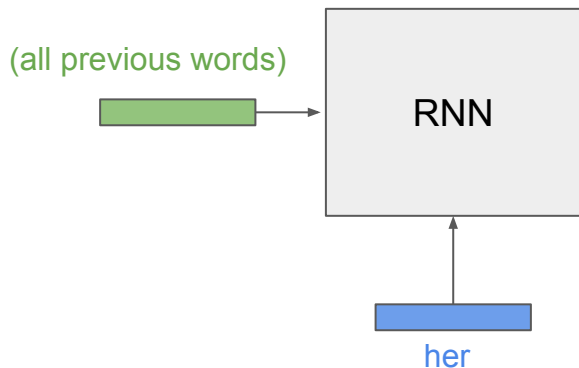
We are hoping that the word "her" can correctly be compared against "Catherine the Great" that is captured in the hidden state.
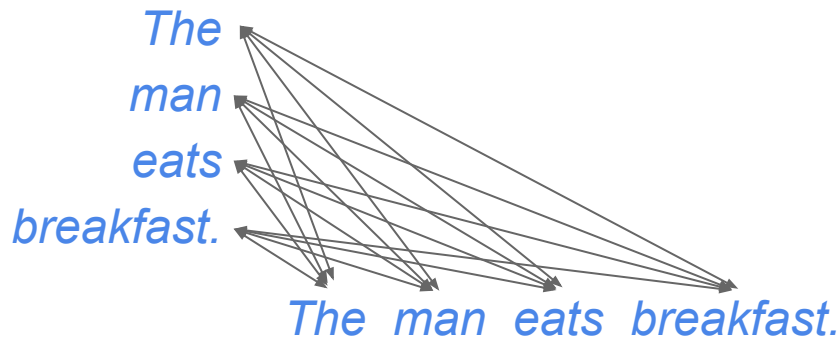
RNN "comparison" view

A lot of text understanding can be thought of as comparing words. How does word at position i compare to all others?
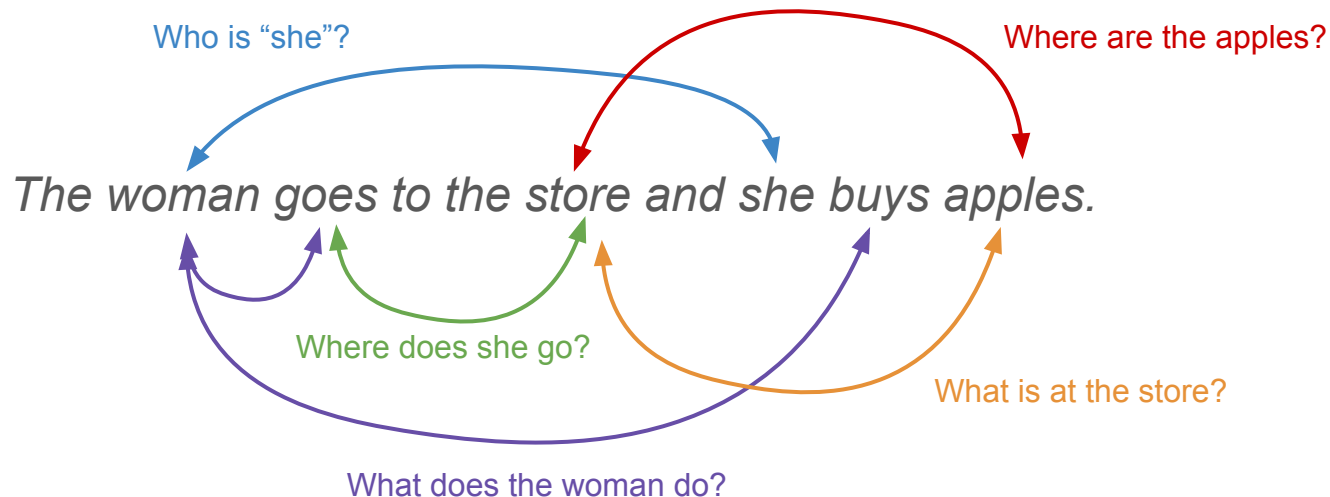
Self-Attention: Main Idea

The transformer is built on a mechanism called "self-attention". The main idea is to compare every word to every other word.

This is an N^2 operation. Much more intense than the RNN, but also much more thorough.

Block Diagram

We want a mechanism where the output at position i represents how the vector at position i compares to all the other vectors:

y_0 y_1 y_2 y_3

Self-Attention

x_0 x_1 x_2 x_3

The man eats breakfast.

This output vector should capture how "the" compares to all other words.

This output vector should capture how "eats" compares to all other words.
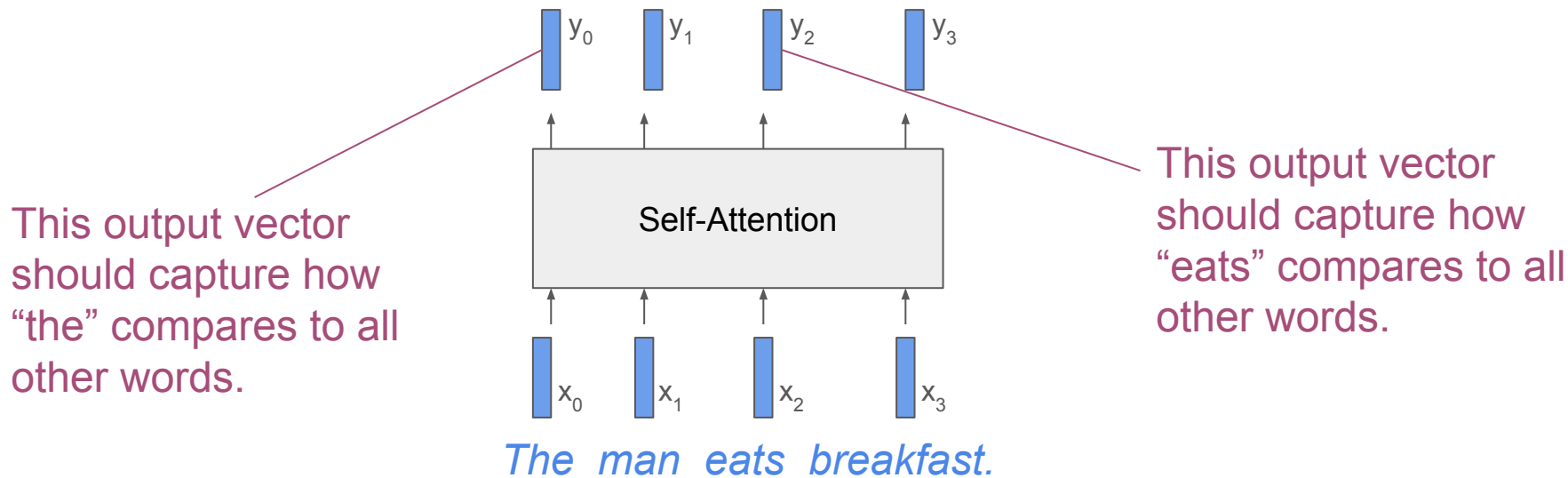
Weighted Average

Each output **y**$_i$ will be an average of all inputs $x_0 \ldots x_S$, <u>weighted by how they compare to the input at position **i**</u>.

$$y_i = \frac{1}{S} \sum_{j=0}^{S} [weight] * x_j$$

The man eats breakfast.

This weighting term somehow captures how x_i compares with x_j.

For i=0 (the word "The") we may expect a high weight for j=1 ("man") and a low weight for j=2 ("eats") and j=3 ("breakfast") since they are less related.

Comparing two vectors

We can easily compare two vectors with the dot product:

$$\text{Compare}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \bullet \mathbf{x}_j$$

Which is just an element-wise sum:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

Weighted Average (Continued)

Each output y_i will be an average of all inputs $x_0 \ldots x_S$, weighted by how they compare to the input at position i.

$$y_i = \frac{1}{S} \sum_{j=0}^{S} [weight] * x_j$$

Using the dot product to compute the weighting terms, we get:

$$y_i = \frac{1}{S} \sum_{j=0}^{S} (x_i \bullet x_j) * x_j$$

Self-Attention (Full)

Weighted Average

We perform this for each output, so x_i shows up in three places: Once in computing output i, and twice in computing the other outputs.

$$y_i = \frac{1}{S} \sum_{j=0}^{S} (x_i \bullet x_j) * x_j$$

Queries, Keys, and Values

Each time some x_k appears, it has a different role:

$$y_i = \frac{1}{S} \sum_{j=0}^{S} (x_i \bullet x_j) * x_j$$

Query
The x_k that we are computing an output for. We are **querying** how x_i relates to all other vectors.

Key
The x_k that we are comparing to our current query of interest, x_i. What "**key**" (as in database key) should we compare our query against?

Value
The x_k that we are weighting. What **value** should we be weighting against other vectors?

Queries, Keys, and Values

Because these are three separate roles, we learn a separate matrix transformation of x_k for each of them. Rather than using x directly we use the output of the transformations:

$$y_i = \frac{1}{S} \sum_{j=0}^{S} (W_q x_i \bullet W_k x_j) * W_v x_j$$

Queries, Keys, and Values

Because these are three separate roles, we learn a separate matrix transformation of x_k for each of them. Rather than using x directly we use the output of the transformations:

$$y_i = \frac{1}{S} \sum_{j=0}^{S} \left[(W_q x_i \bullet W_k x_j) * W_v x_j \right]$$

Note: The sum is over everything.

Block-Diagram Version

$$y_i = \frac{1}{S} \sum_{j=0}^{S} \left(W_q x_i \cdot W_k x_j \right) * W_v x_j$$

If it is helpful, we can try to draw a diagram of this. It gets messy very quickly.
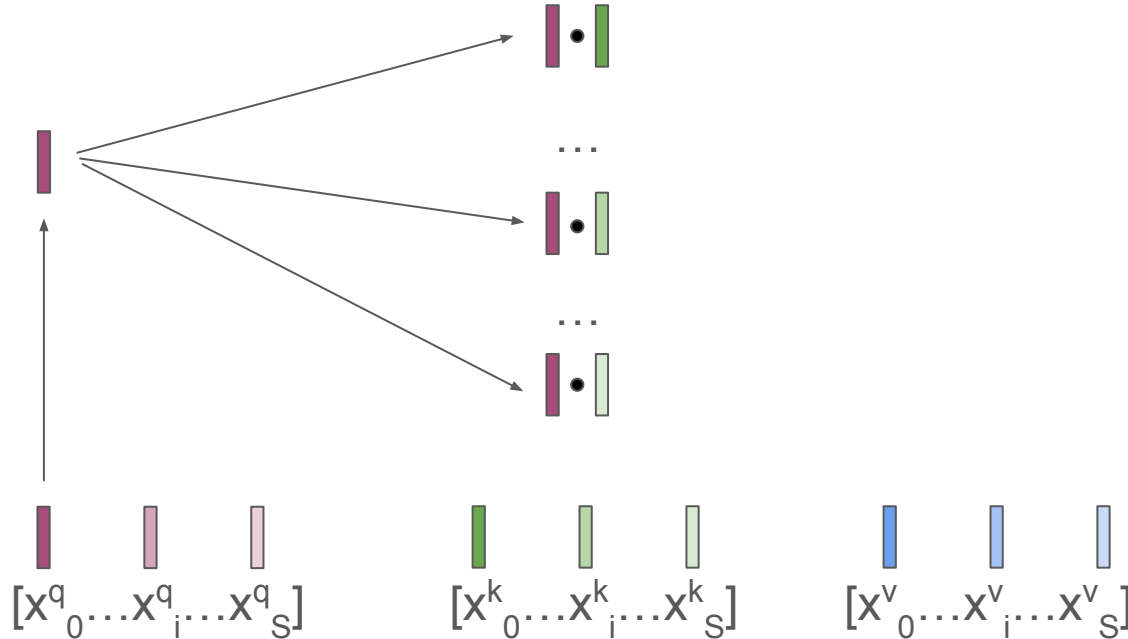
Step 1: All input vectors **x** are transformed into queries, keys and values:
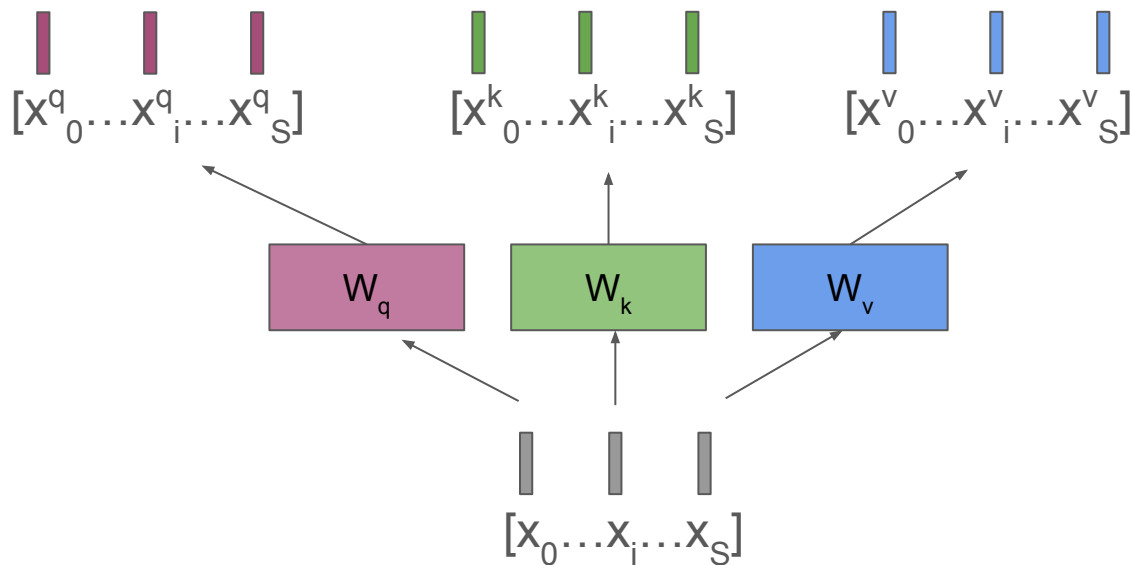
Block-Diagram Version

$$y_i = \frac{1}{S} \sum_{j=0}^{S} (W_q x_i \bullet W_k x_j) * W_v x_j$$

Step 2: Each query is compared against each key (just one query shown).

$$[x^q_0 \ldots x^q_i \ldots x^q_S]$$

$$[x^k_0 \ldots x^k_i \ldots x^k_S]$$

$$[x^v_0 \ldots x^v_i \ldots x^v_S]$$

Block-Diagram Version

$$y_i = \frac{1}{S} \sum_{j=0}^{S} (W_q x_i \bullet W_k x_j) \; * W_v x_j$$

Step 3: This weights the values.
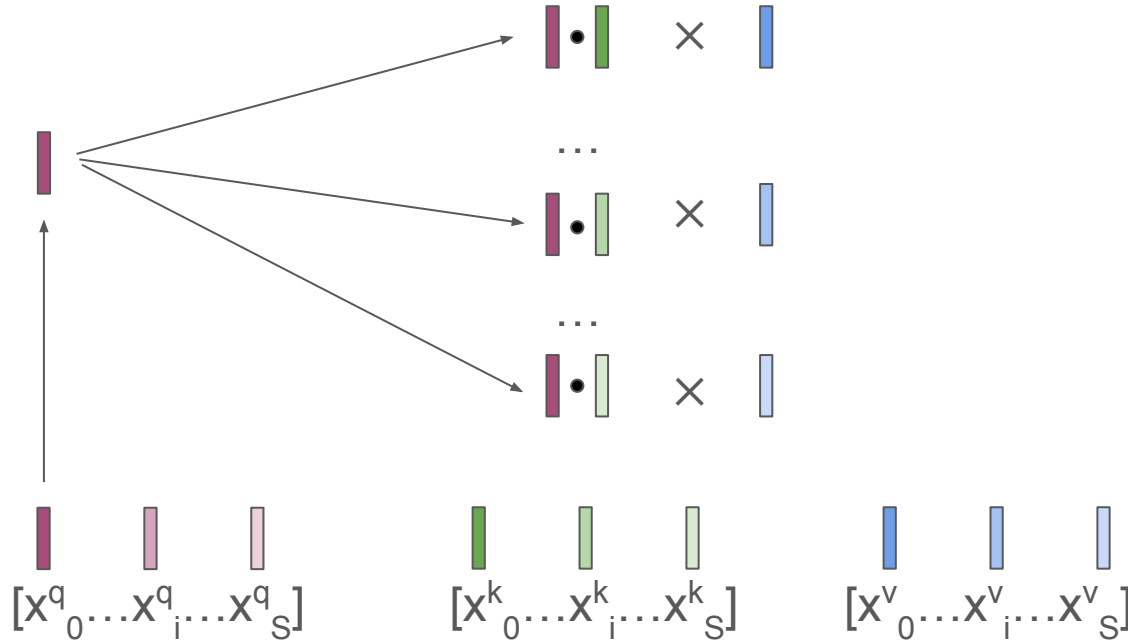
$[x^q_0 \ldots x^q_i \ldots x^q_S]$ $[x^k_0 \ldots x^k_i \ldots x^k_S]$ $[x^v_0 \ldots x^v_i \ldots x^v_S]$

Block-Diagram Version

$$y_i = \frac{1}{S} \sum_{j=0}^{S} (W_q x_i \bullet W_k x_j) * W_v x_j$$

Step 4: The output is the average of all these weighted values.

Block-Diagram Version

$$y_i = \frac{1}{S} \sum_{j=0}^{S} (W_q x_i \bullet W_k x_j) * W_v x_j$$

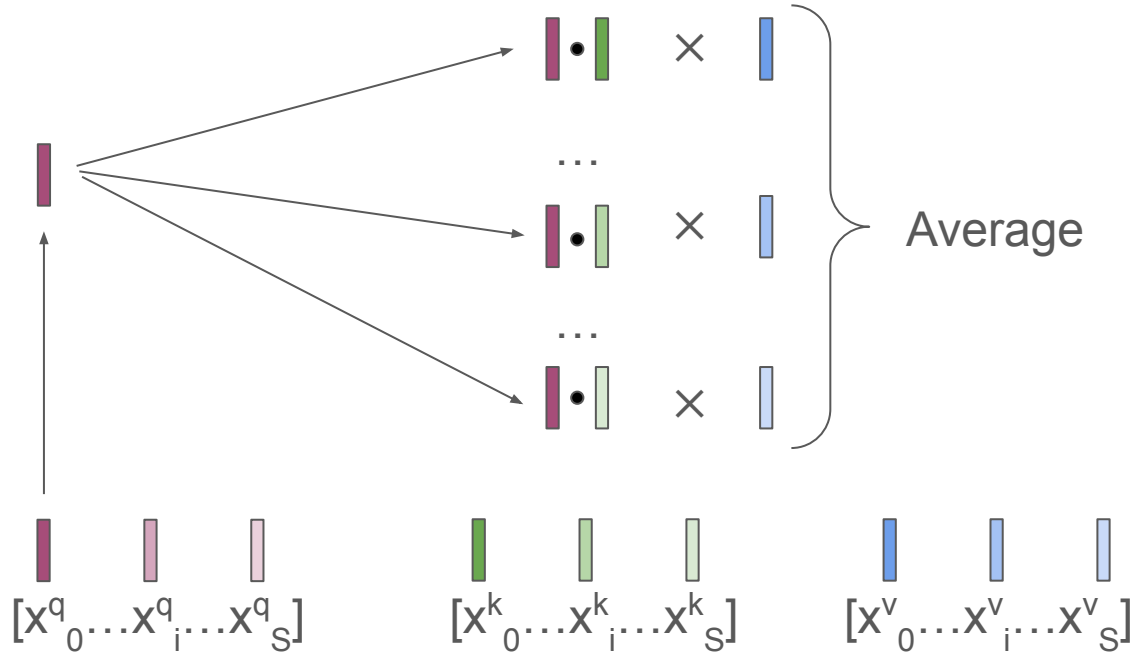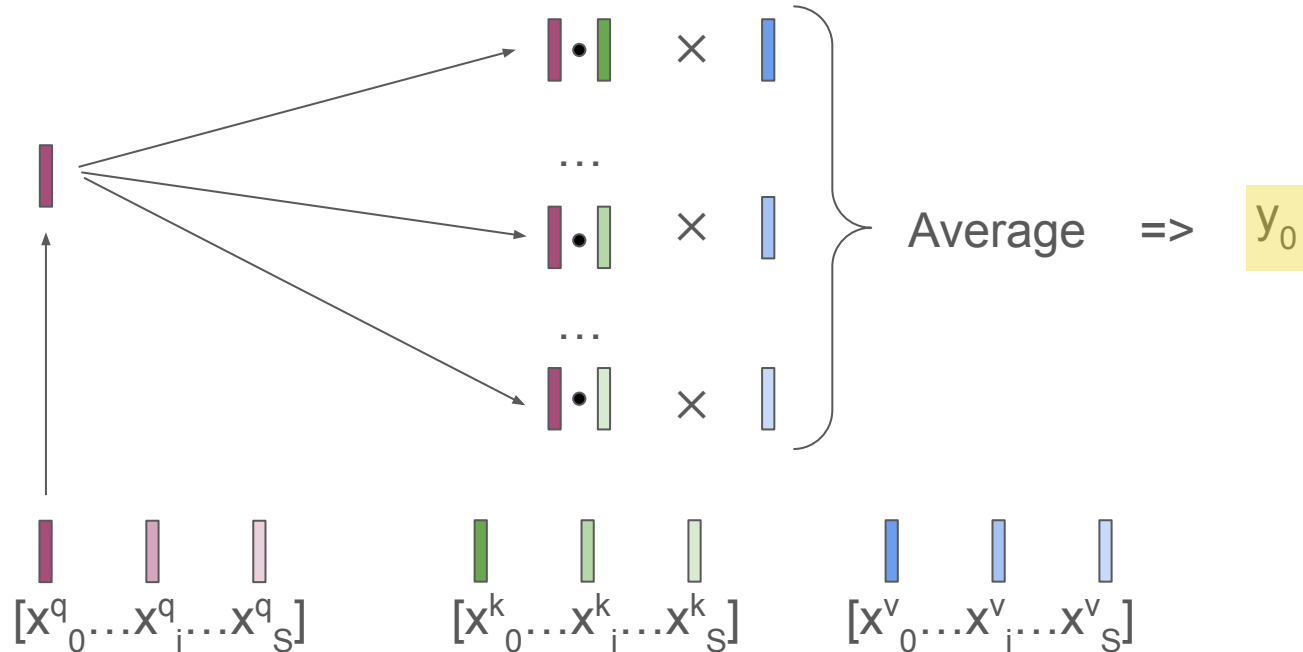**Step 4**: The output is the average of all these weighted values.

Average => y_0

$[x^q_0 \ldots x^q_i \ldots x^q_S]$ $[x^k_0 \ldots x^k_i \ldots x^k_S]$ $[x^v_0 \ldots x^v_i \ldots x^v_S]$

Queries, Keys, and Values

It is worth pausing to consider why we need different transformations of X for each of these roles.

The **query** is the easiest to understand. When we are computing output i, this is a representation of x_i that we compare against all other entries.

What is the role of the key vs the value?

The man eats breakfast.

The **key** is used to understand how each word is related to others. The key for "man" might help us understand that man is the subject. "The" and "man" have a specific functional relationship.

**Functionally:** How should we attend to this word?

The **value** is what we are actually weighting and using to construct our output. The value for "man" may want to capture what "man" actually means.

Functionally: How should we represent this word?

Average

Weighting Terms:
Queries x Keys

Values: What we are actually weighting and propagating forward in the network.
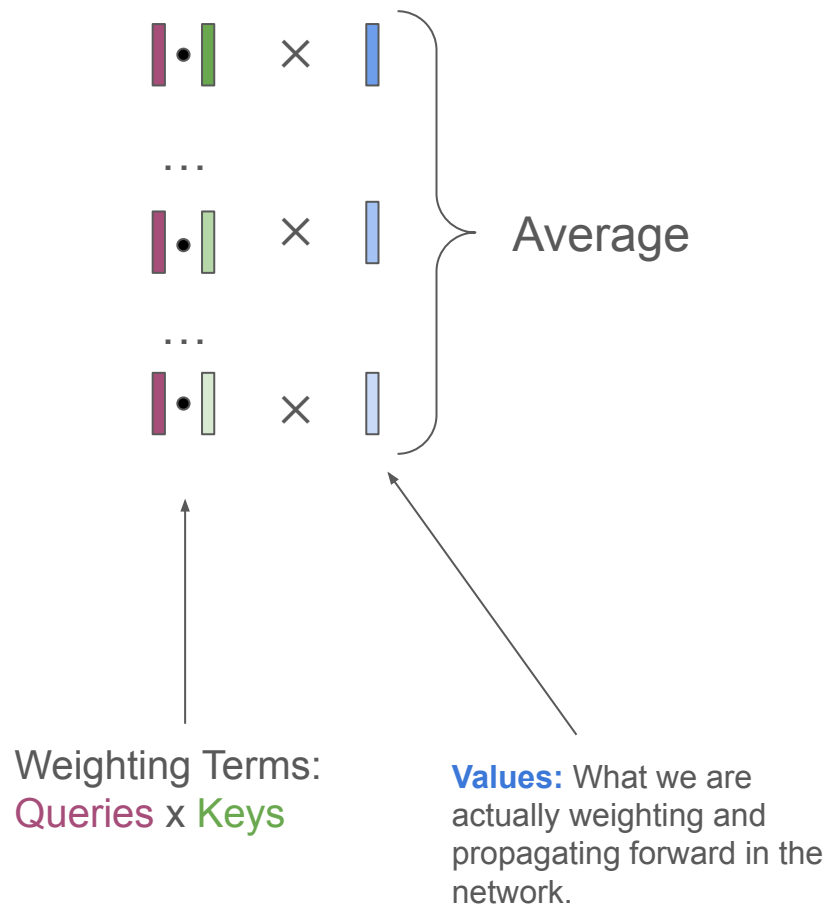
Matrix Version

This is wayyyyyyyy simpler in matrix form. We can re-write many dot product operations as matrix multiplication. Let's look at these as matrices:

Q^{SxD} K^{SxD} V^{SxD}

W_q^{DxD} W_k^{DxD} W_v^{DxD}

$[x_0 \ldots x_i \ldots x_S]^T = X^{SxD}$
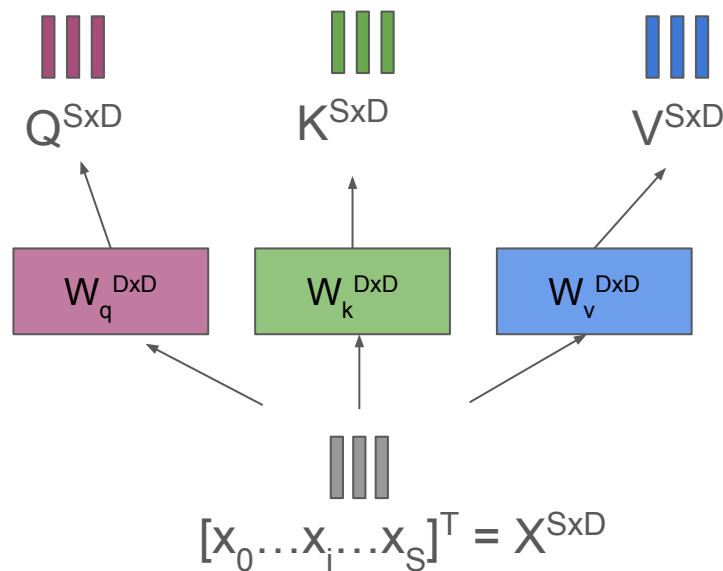
Each row of the matrix is one of our vectors of length D.

I am showing the rectangles representing data transposed because it is easier to draw a sequence of vectors this way.

Matrix Version

To perform all dot products between queries and keys we can use matrix multiplication:

$$\text{Weighting terms (size SxS)} = QK^T =$$

And we can then weight the values by matrix-multiplying these weights by the values:

$$\text{Output } Y = (QK^T)V =$$

Some Minor Adjustments

To make our weights nicer, we use a softmax function to have them sum to 1. If we use the raw dot product, we could get weighting terms that are really large.

For further stability, we scale the dot products themselves in relation to the dimension of the vectors (D). Remember, the dot product is a sum, so its magnitude is related to how long our vectors are. We correct this by dividing by sqrt(D):

$$Y = softmax(\frac{QK^T}{\sqrt{D}}) * V$$

Multi-Head Attention

Just in case it wasn't complicated enough already.

Capturing Multiple Relations per Word

What if a word has multiple relations we want to capture? (This is most of the time)

The angry man walked the poodle.

When computing an output for "man", we may expect high weights for "angry" and "walked" - both describe the man. However, if we average these together they may not be meaningful.

To correct for this, we compute self-attention <u>H different times</u>. This allows us to learn up to H different ways that a word relates to other words. (Typical values of H are smallish powers of two: 16, 32, 64. The largest GPT3 model uses 96.)

Multihead Attention (conceptual version)

We have H separate self-attention blocks (heads):
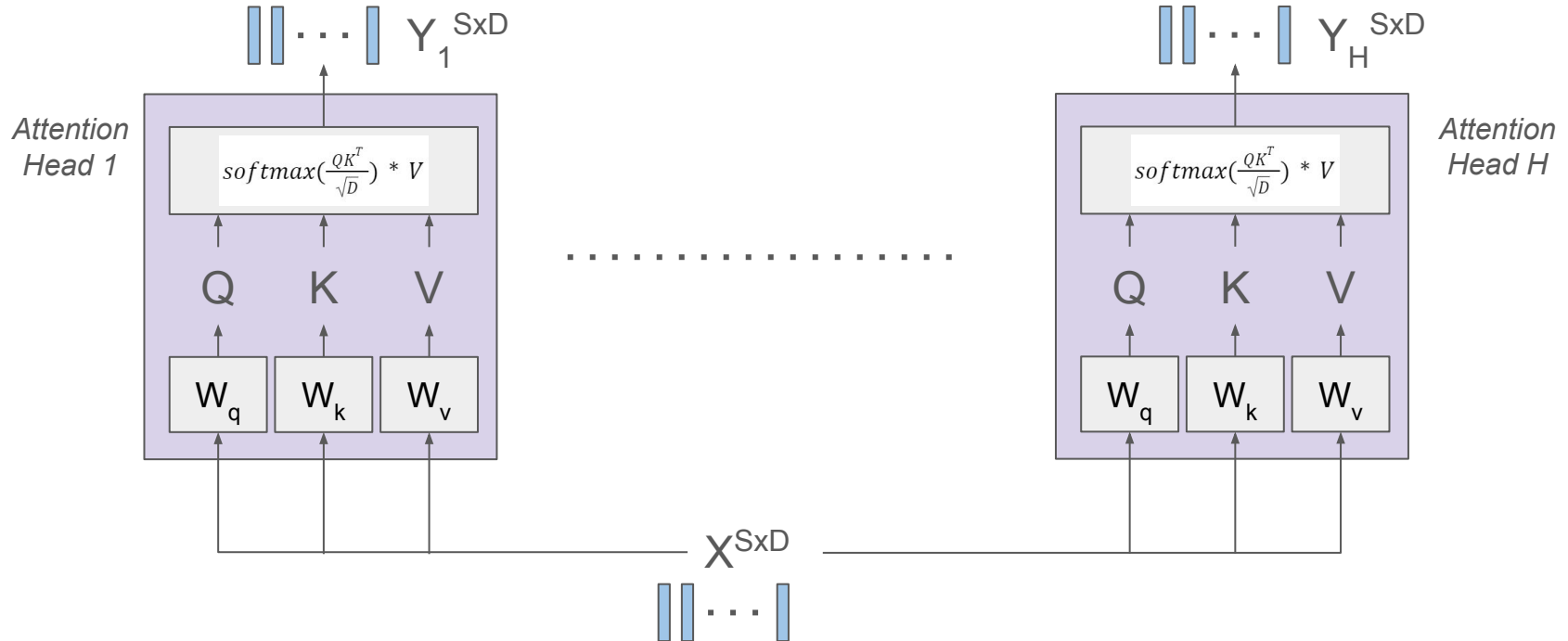
Efficiency Improvements

The conceptual version on the previous slide is incredibly expensive, both in terms of computation (we have 3*H weight vectors of size DxD) and memory (we also have to track 3*H*S vectors of size D).

We make 3 major improvements:

1) Each attention head uses a reduced vector size D/H instead of D
2) We compute all queries with a single matrix (and keys and values)
3) We further combine all Q, K, V computation into a single giant computation.

We are going to look at these one at a time.

Improvement (1): Smaller Vectors Internally

Queries, keys, and values use a smaller vector size, D/H:
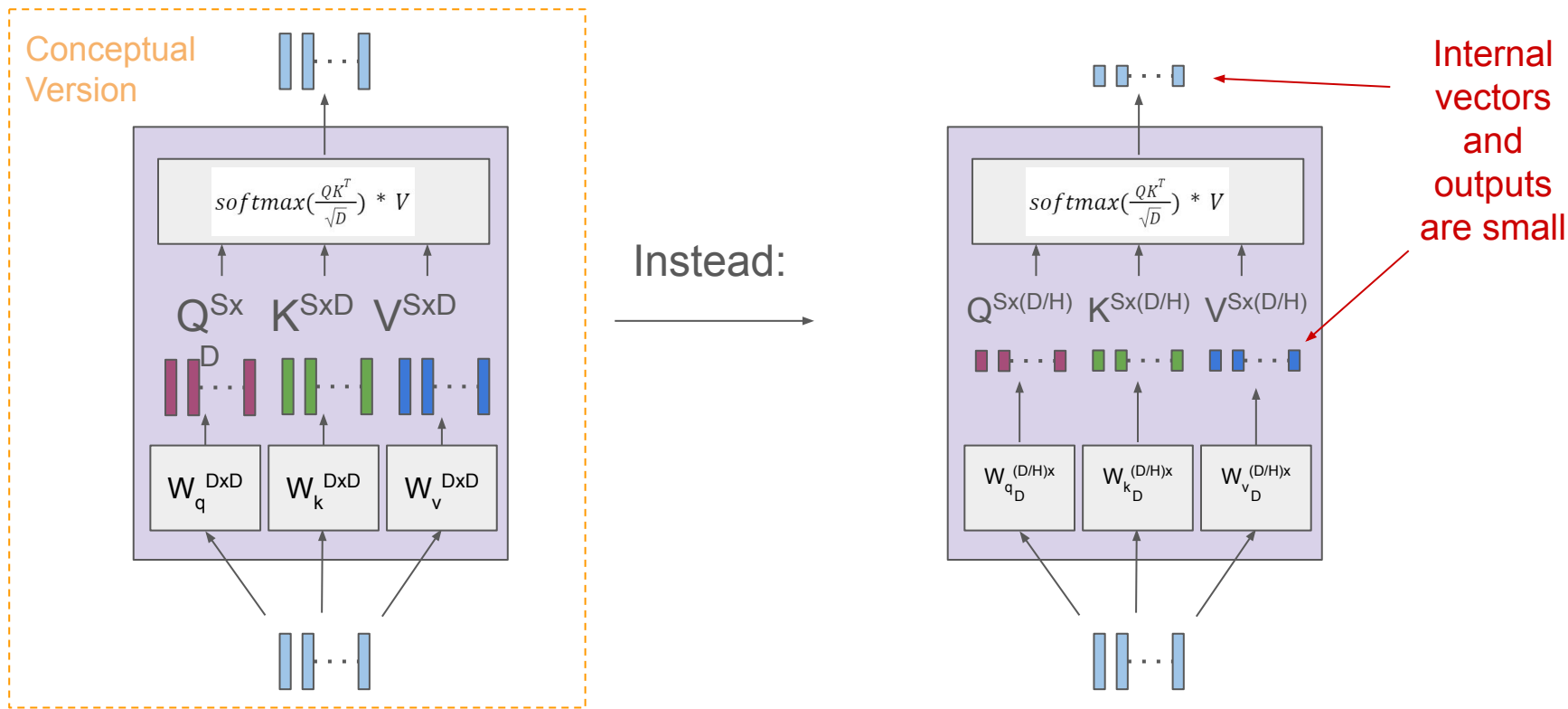
Improvement (2): Shared Weights

Instead of H matrices of size ((D/H)xD), use one matrix of size (DxD) and then slice the vectors:

Conceptual Version

Head 1   Head 2   Head H

$Q_1^{Sx(D/H)}$ $Q_2^{Sx(D/H)}$ $Q_H^{Sx(D/H)}$

$W_{q1}^{(D/H)x}{}_D$ $W_{q2}^{(D/H)x}{}_D$ $W_{qH}^{(D/H)x}{}_D$

X^{SxD}

Instead:

$Q_1^{Sx(D/H)}$ $Q_2^{Sx(D/H)}$

slice

W_q^{DxD}

X^{SxD}

Divided up **_after_** computing queries.

Single matrix for all queries.
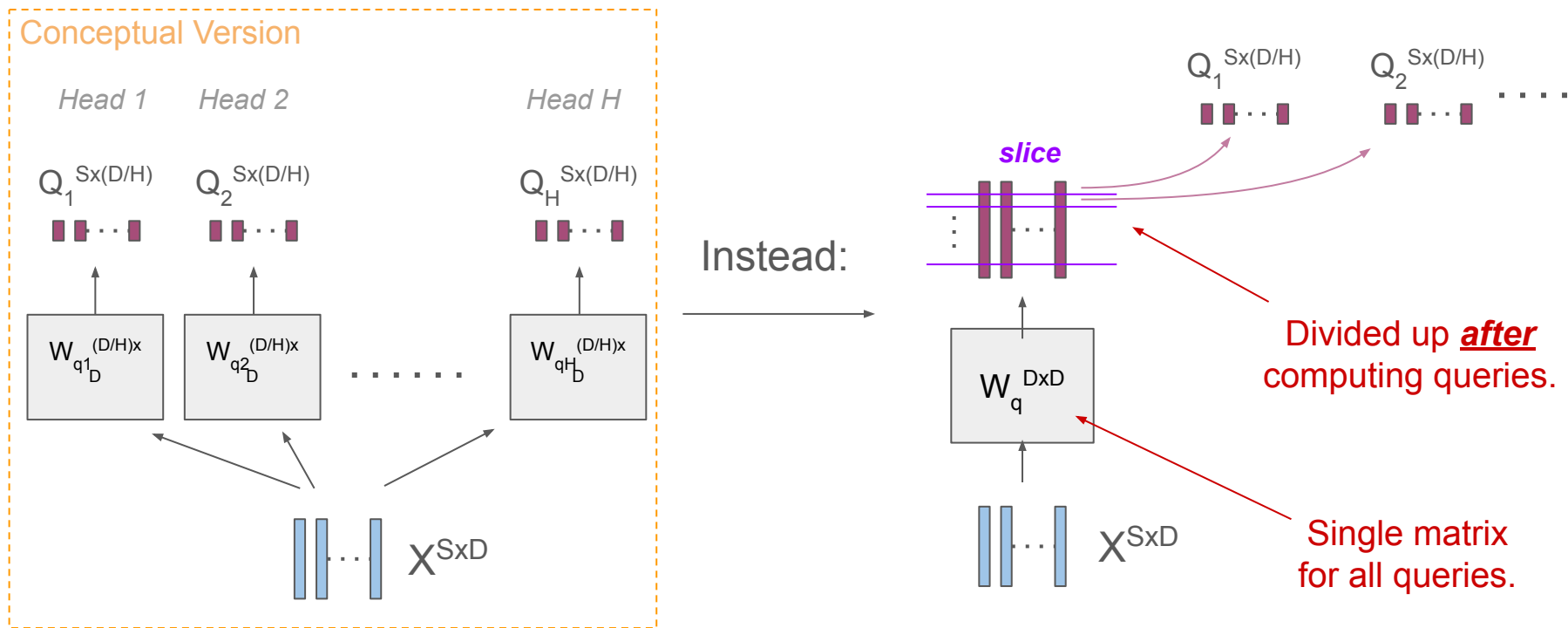
Improvement (3): Giant Q/K/V Computation

Replace W_q^{DxD}, W_k^{DxD}, W_v^{DxD} with a single W_{qkv}^{3DxD} :
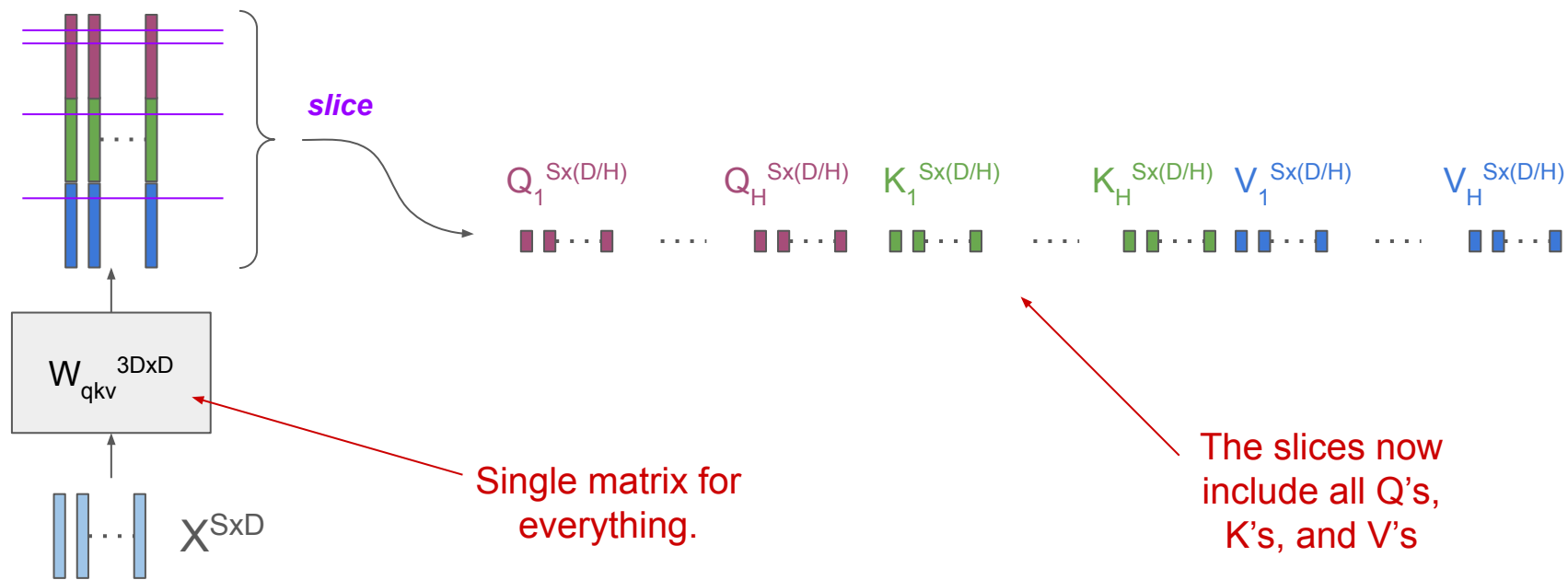
slice

$Q_1^{Sx(D/H)}$ $Q_H^{Sx(D/H)}$ $K_1^{Sx(D/H)}$ $K_H^{Sx(D/H)}$ $V_1^{Sx(D/H)}$ $V_H^{Sx(D/H)}$

W_{qkv}^{3DxD}

X^{SxD}

Single matrix for everything.

The slices now include all Q's, K's, and V's

Whole Sequence

If we add a new dimension to our tensors
we can compute all the outputs at once:

$Q_1^{Sx(D/H)}$ $\cdots$ $Q_H^{Sx(D/H)}$ $K_1^{Sx(D/H)}$ $\cdots$ $K_H^{Sx(D/H)}$ $V_1^{Sx(D/H)}$ $\cdots$ $V_H^{Sx(D/H)}$

reshape →

$Q^{HxSx(D/H)}$ $K^{HxSx(D/H)}$ $V^{HxSx(D/H)}$
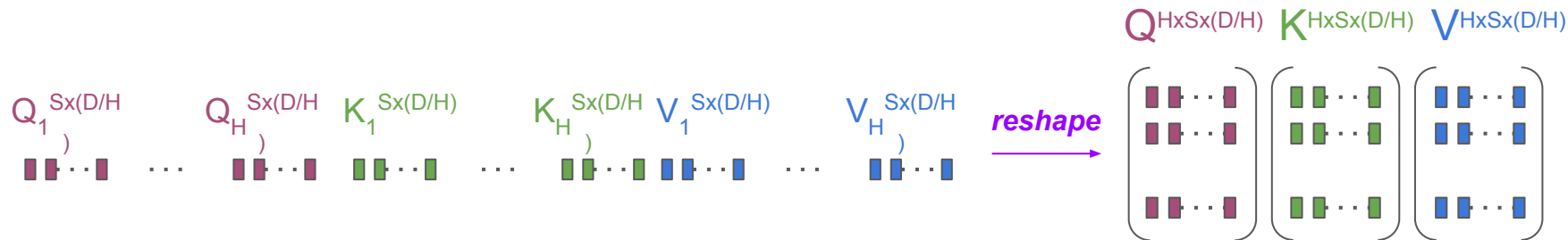
Whole Sequence

If we add a new dimension to our tensors
we can compute all the outputs at once:

Whole Sequence

If we add a new dimension to our tensors
we can compute all the outputs at once:

The dimensions
are a bit screwy
here but you get
the idea.

Swap first two
axes and then
concatenate.

SxD

concatenate

HxSx(D/H)

$$softmax(\frac{QK^T}{\sqrt{D}}) * V$$

$Q^{HxSx(D/H)}$ $K^{HxSx(D/H)}$ $V^{HxSx(D/H)}$

$Q_1^{Sx(D/H)}$ $Q_H^{Sx(D/H)}$ $K_1^{Sx(D/H)}$ $K_H^{Sx(D/H)}$ $V_1^{Sx(D/H)}$ $V_H^{Sx(D/H)}$
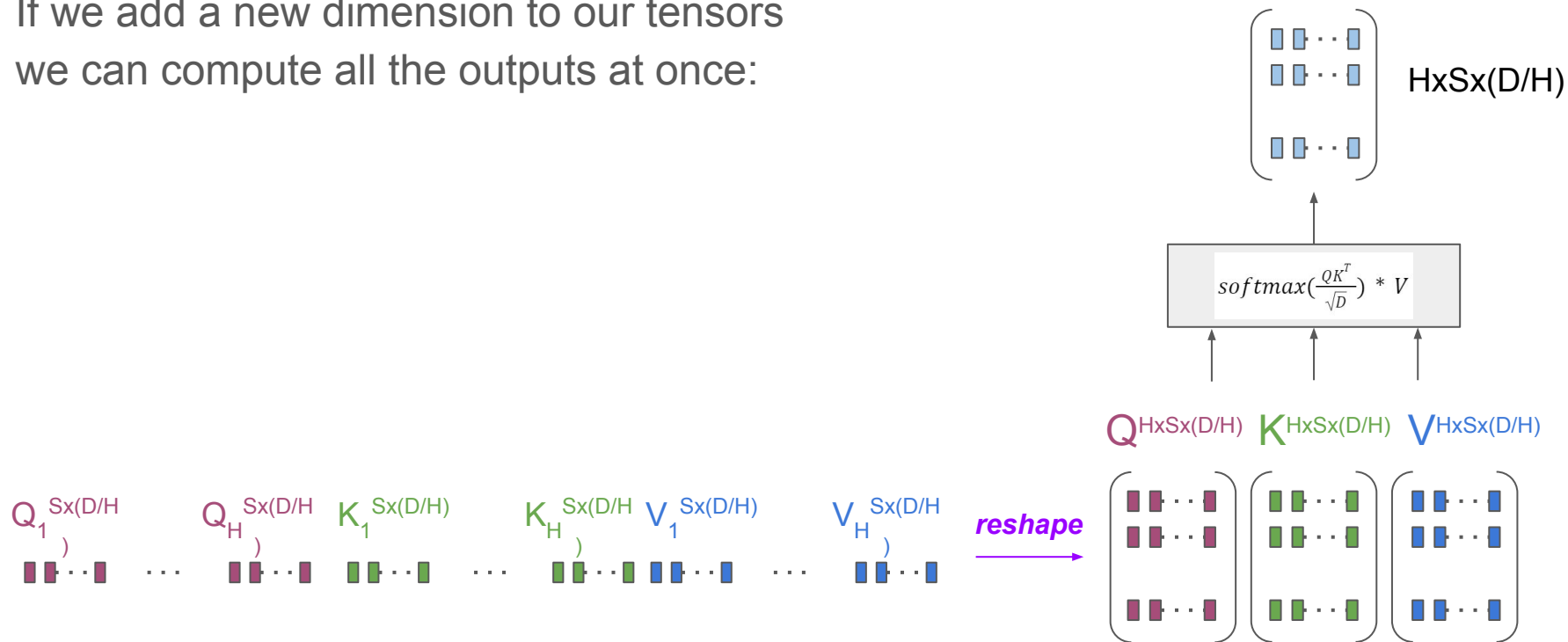
reshape
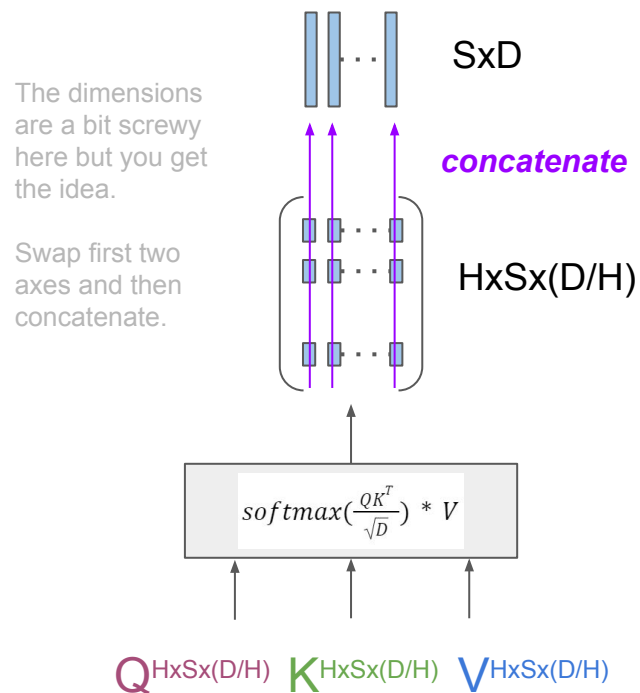
Final Step

The final step is to run our vectors through one more DxD matrix so that the concatenated sections can be fully merged into meaningful vectors:

SxD

W_o

Y^{SxD}

concatenate

HxSx(D/H)
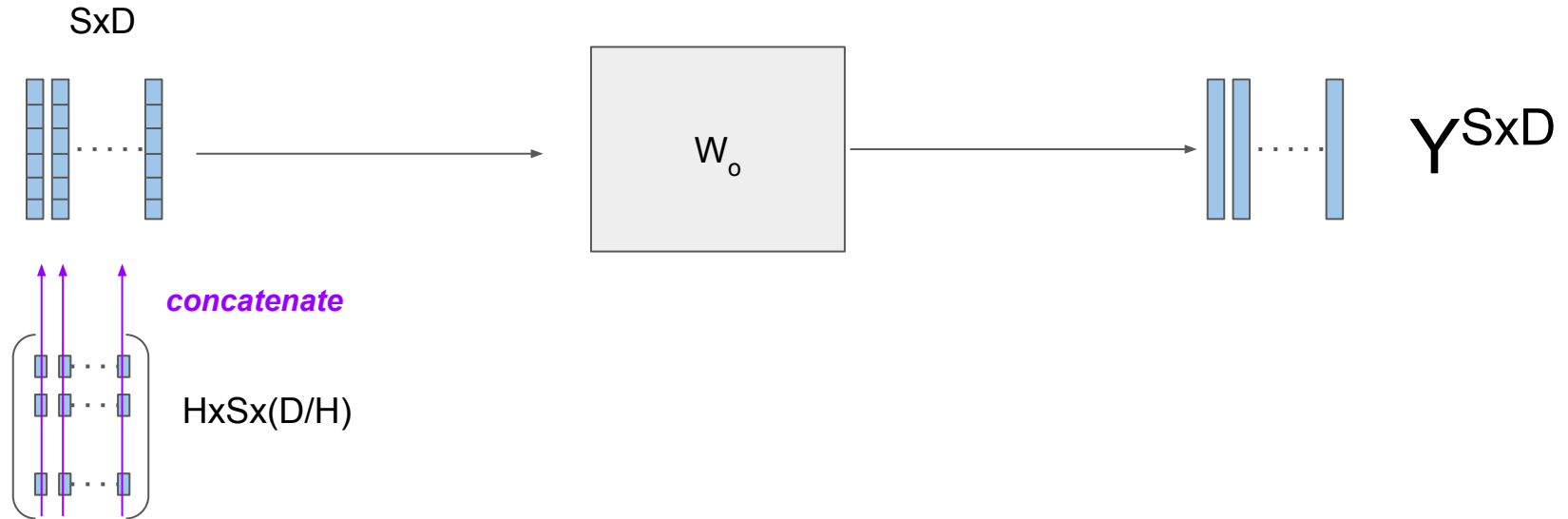
MHA Summary

Input is $\mathbf{X}^{SxD}$, comprised of S input vectors, length D.

MHA stores weights $\mathbf{W}_{qkv}^{3DxD}$ and $\mathbf{W}_o^{DxD}$

Perform matrix multiplication $(\mathbf{X})(\mathbf{W}^T_{qkv})$ to get tensor $\mathbf{T}$ size (Sx3D)

Slice $\mathbf{T}$ into S vectors of length D and reshape into $\mathbf{Q}^{HxSx(D/H)}$, $\mathbf{K}^{HxSx(D/H)}$, $\mathbf{V}^{HxSx(D/H)}$

Apply the attention equation on these batches to get output $\mathbf{Y}$' size (HxSx(D/H))

Swap first two axes and concatenate along second dimension to get $\mathbf{Y}$' size (SxD)

Perform $(\mathbf{Y}')(\mathbf{W}_o^T)$ to get final output $\mathbf{Y}$ of size (SxD)

Details: Matrix Multiplication in PyTorch

In PyTorch, matrix multiplication applies to the last dimensions of the tensors, provided the first dimensions are equal:

$$A^{NxMxO}B^{NxOxP} = C^{NxMxP}$$ same as: N parallel instances of $A^{MxO}B^{OxP}$

This also applies with additional leading dimensions:
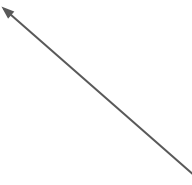
$$A^{LxNxMxO}B^{LxNxOxP} = C^{LxNxMxP}$$ same as: L*N parallel instances of $A^{MxO}B^{OxP}$

Details: Applying Attention Equation

On the previous slide we construct $\mathbf{Q}^{HxSx(D/H)}$, $\mathbf{K}^{HxSx(D/H)}$, $\mathbf{V}^{HxSx(D/H)}$

The first step of applying attention is to take QK^T. This is a bit simplistic.

What we are trying to find is: for each attention head, for each member of the sequence, an attention weight for all elements of the sequence. Thus our desired shape is $[QK^T]^{HxSxS}$

This SxS matrix is an "attention matrix", which describes how each element in the sequence relates to every other (hence S^2 entries).

Details: Applying Attention Equation

On the previous slide we construct $\mathbf{Q}^{HxSx(D/H)}$, $\mathbf{K}^{HxSx(D/H)}$, $\mathbf{V}^{HxSx(D/H)}$

The first step of applying attention is to take QK^T. This is a bit simplistic.

What we are trying to find is: for each attention head, for each member of the sequence, an attention weight for all elements of the sequence. Thus our desired shape is $[QK^T]^{HxSxS}$

We transpose the last two axes of K into $K^{Hx(D/H)xS}$. This is K^T.

Multiplying together we get (HxSx(D/H))*(Hx(D/H)xS)=>(HxSxS), representing H*S separate sets of attention weights (each size S). To normalize those weights, we apply softmax to the last dimension. In pytorch we can do this with torch.nn.functional.softmax(matrix, dim=-1)

Details: Applying Attention Equation

After dividing by sqrt(D) and taking the softmax, we have a matrix of normalized attention weights with size (HxSxS) multiplied by tensor $V^{HxSx(D/H)}$.

Following the matrix multiplication rules from a few slides ago, one of the "S" dimensions cancels and we get another tensor size (HxSx(D/H)):

$$\left[\text{[intermediate result]}^{HxSx\textbf{S}}\right]\left(V^{Hx\textbf{S}x(D/H)}\right) = \left(\text{size } HxSx(D/H)\right)$$

This represents, for each head, for each element in the sequence, the result of applying the attention equation.

Details: With Batches of Samples

For a **<u>batch</u>** of training samples everything just has an extra dimension out front.

$\mathbf{X}^{SxD}$ $\longrightarrow$ $\mathbf{X}^{BxSxD}$

$\mathbf{Q}^{HxSx(D/H)}$ $\longrightarrow$ $\mathbf{Q}^{BxHxSx(D/H)}$

$\mathbf{Y}^{SxD}$ $\longrightarrow$ $\mathbf{Y}^{BxSxD}$

Important Note

By far the most complicated part of implementing MHA is keeping track of dimensions and shapes of tensors.

Print out tensor shapes throughout your code and make sure they seem right.

Be careful of broadcasting!

Additional Resources

It can be helpful to get several perspectives on a topic when trying to understand it for the first time. I have found these explanations of MHA to be helpful:

- 3B1B Video: https://www.youtube.com/watch?v=eMlx5fFNoYc
- Peter Bloem's Blog post: https://peterbloem.nl/blog/transformers
  - This contains code- please don't just copy this. If you end up needing to reference this, carefully cite it and don't use it as a crutch.
  - This is also your reading for next week's module 5.
- Jay Alamar's Blog post: https://jalammar.github.io/illustrated-transformer/
- Bonus: Original Paper: https://arxiv.org/pdf/1706.03762

Review Assignments

Appendix: Tensor Operations Guide

Tensor Manipulation Cheatsheet: Dimensions

You can **query the shape** of a tensor and assign it to variables with **tensor.shape()**

```
>>> x = torch.Tensor([[1,2,3],[4,5,6],[7,8,9],[0,0,0]])
>>> x.shape
torch.Size([4, 3])
>>> B, N = x.shape
>>> B
4
>>> N
3
```

You can **remove a redundant dimension** of a tensor with **tensor.squeeze()**

```
>>> x = torch.Tensor([[[1,2,3]],[[4,5,6]],[[7,8,9]],[[0,0,0]]])
>>> x.shape
torch.Size([4, 1, 3])
>>> x = x.squeeze()
>>> x.shape
torch.Size([4, 3])
```

You can **add a redundant dimension** of a tensor with tensor.unsqueeze(), but a much easier option is **indexing with *None*** where you want a dimension:

```
>>> x = torch.Tensor([[1,2,3],[4,5,6]])
>>> y = x[:,None,:]
>>> y.shape
torch.Size([2, 1, 3])
>>> z = x[:,:,None]
>>> z.shape
torch.Size([2, 3, 1])
>>> k = x[None,:,None,None,:]
>>> k.shape
torch.Size([1, 2, 1, 1, 3])
```

You can **swap dimensions** of a tensor with **tensor.transpose(dim0, dim1)**

```
>>> x = torch.Tensor([[[1,2,3],[2,3,4]]])
>>> x.shape
torch.Size([1, 2, 3])
>>> y = x.transpose(0,-1)
>>> y.shape
torch.Size([3, 2, 1])
>>> z = x.transpose(1,2)
>>> z.shape
torch.Size([1, 3, 2])
```

You can **reshape** a tensor with **tensor.reshape(new_shape_dims)**. Compare with transpose below:

```
>>> x = torch.Tensor([[1,2,3],[4,5,6]])
>>> x
tensor([[1., 2., 3.],
        [4., 5., 6.]])
>>> y = x.reshape(3,2)
>>> y
tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])
>>> z = x.transpose(0,1)
>>> z
tensor([[1., 4.],
        [2., 5.],
        [3., 6.]])
```

You can **repeat data** with **tensor.repeat(list_of_repeats).**

```
>>> x = torch.Tensor([[1,2,3],[4,5,6]])
>>> y = x.repeat(2,1)
>>> y
tensor([[1., 2., 3.],
        [4., 5., 6.],
        [1., 2., 3.],
        [4., 5., 6.]])
>>> z = x.repeat(1,2)
>>> z
tensor([[1., 2., 3., 1., 2., 3.],
        [4., 5., 6., 4., 5., 6.]])
```

You can add a dimension here as well:

```
>>> k = x.repeat(2,1,2)
>>> k
tensor([[[1., 2., 3., 1., 2., 3.],
         [4., 5., 6., 4., 5., 6.]],

        [[1., 2., 3., 1., 2., 3.],
         [4., 5., 6., 4., 5., 6.]]])
>>> k.shape
torch.Size([2, 2, 6])
```

You can also combine this with *None* indexing to **insert and then repeat** along a dimension. This is a good trick.

```
>>> x = torch.Tensor([[1,2,3],[4,5,6]])
>>> y = x[None,:,:].repeat(2,1,1)
>>> y
tensor([[[1., 2., 3.],
         [4., 5., 6.]],

        [[1., 2., 3.],
         [4., 5., 6.]]])
```

You can **break up a tensor** with the **tensor.chunk(chunks, dimension)** method (technically returns views).

```
>>> x = torch.Tensor([[1,2,3],[4,5,6]])
>>> a,b = x.chunk(2,dim=0)
>>> a,b
(tensor([[1., 2., 3.]]), tensor([[4., 5., 6.]]))
>>> j,k,l = x.chunk(3,dim=1)
>>> j,k,l
(tensor([[1.],
        [4.]]), tensor([[2.],
        [5.]]), tensor([[3.],
        [6.]]))
```

The last line printed horribly. That says:
[[1],[4]], [[2],[5]], and [[3],[6]] (vertical slices).

You can **implicitly repeat a tensor** via broadcasting: if the shapes line up for some of the dimensions, the operation will broadcast to the others (when applicable).

Be extremely careful with this (in fact, I suggest you avoid it completely). Broadcasting means that an operation can run successfully even when you make a mistake in tensor dimensions!

```
>>> x = torch.Tensor([[1,2,3],[4,5,6]])
>>> x + torch.Tensor([[1,1,1],[1,1,1]])
tensor([[2., 3., 4.],
        [5., 6., 7.]])
>>> x + torch.Tensor([1,1,1])
tensor([[2., 3., 4.],
        [5., 6., 7.]])
>>> x + 1
tensor([[2., 3., 4.],
        [5., 6., 7.]])
>>> x + torch.Tensor([[1],[1]])
tensor([[2., 3., 4.],
        [5., 6., 7.]])
```

Matrix multiplication applies to the last two dimensions, and can arbitrary leading dimensions:

```
>>> x = torch.rand((2,3,4)) @ torch.rand((2,4,5))
>>> x.shape
torch.Size([2, 3, 5])
>>> x = torch.rand((7,2,3,4)) @ torch.rand((7,2,4,5))
>>> x.shape
torch.Size([7, 2, 3, 5])
```

If the leading dimensions are not equal you will get an error:

```
>>> x = torch.rand((7,2,3,4)) @ torch.rand((7,9,4,5))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: The size of tensor a (2) must match the size of tensor b (9) at non-singleton dimension 1
```

If you have a different number of leading dimensions pytorch will apply broadcasting:

```
>>> x = torch.rand((7,2,3,4)) @ torch.rand((2,4,5))
>>> x.shape
torch.Size([7, 2, 3, 5])
```

Broadcasting will also apply when you have mismatched dimensions **but one of them has size 1**. This can be really hard to find if it is causing a bug:

```
>>> x = torch.rand((7,2,3,4)) @ torch.rand((7,1,4,5))
>>> x.shape
torch.Size([7, 2, 3, 5])
```

A shorthand for transpose is **tensor.T**, but this is often not what you want for more than two dimensions. It reverses all dimensions:

```
>>> x = torch.rand((7,2,3,4)).T
>>> x.shape
torch.Size([4, 3, 2, 7])
```

You can use **tensor.mT** to only swap the last two dimensions, treating your tensor as a collection of 2D tensors. Thanks to Evan G. for this tip!

```
>>> x = torch.rand((7,2,3,4)).mT
>>> x.shape
torch.Size([7, 2, 4, 3])
```