

Attention

Lecture 4

EN.705.743: ChatGPT from Scratch

Lecture Outline

Lecture 4: Attention

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

Recap

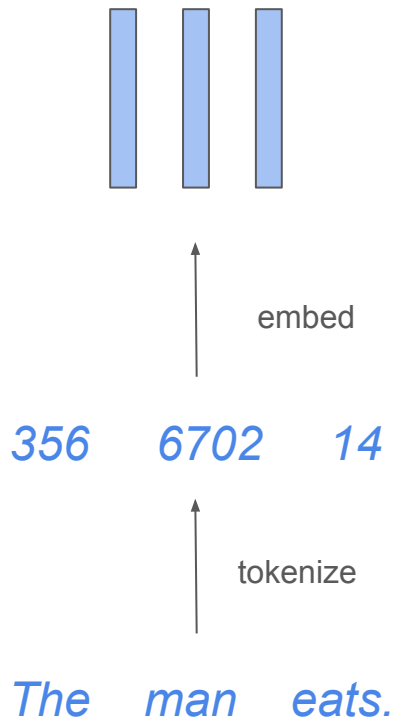
Recap

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”

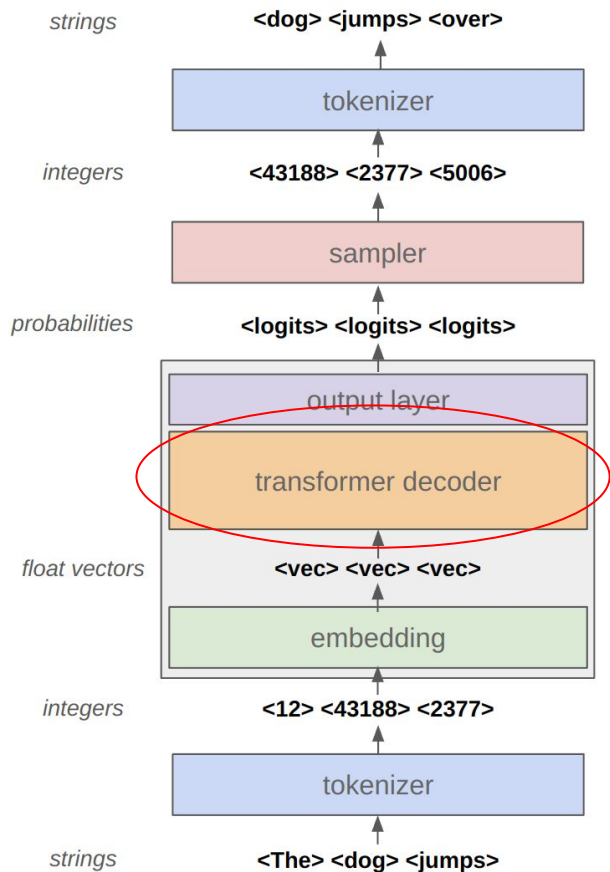


Recap

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

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

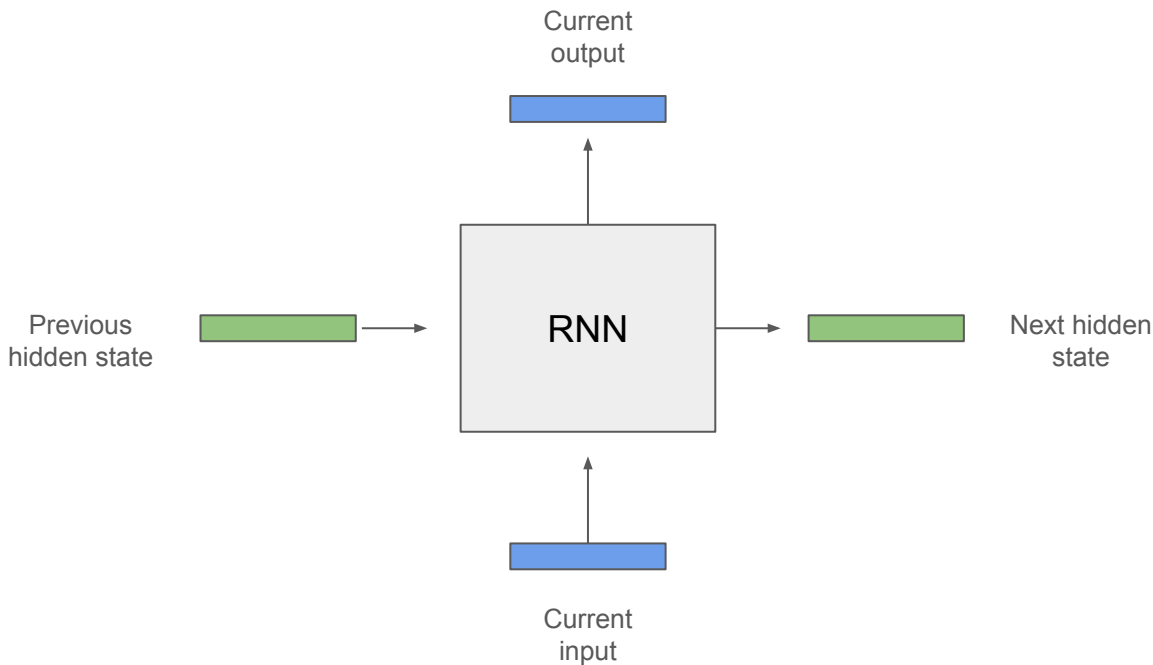
Now our text has been converted into vectors, and we need to build a model that can perform computations on sequences of vectors.



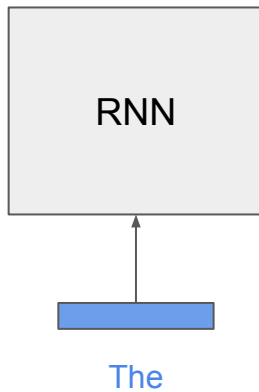
RNNs

How to process a sequence of vectors?

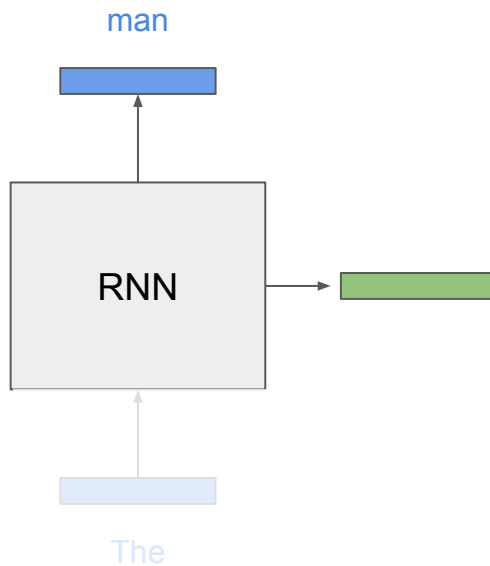
Before transformers, the common practice was to use a recurrent neural network (RNN):



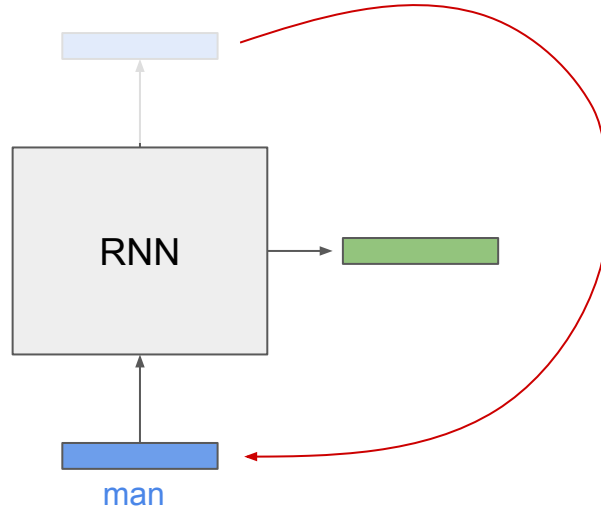
RNN Example: Text Generation



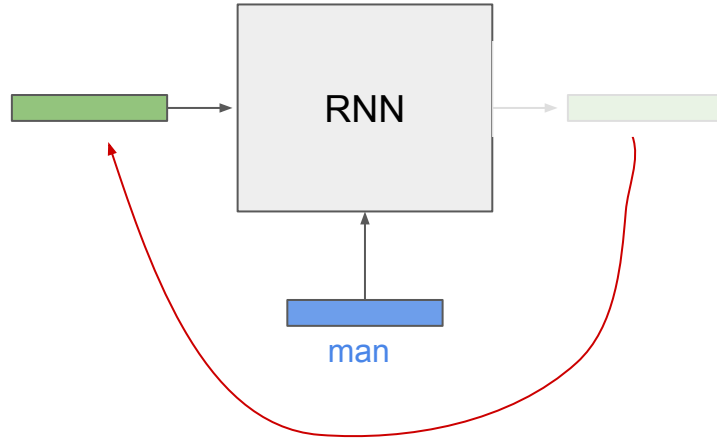
RNN Example: Text Generation



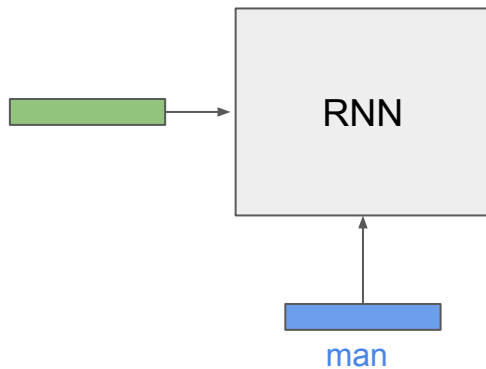
RNN Example: Text Generation



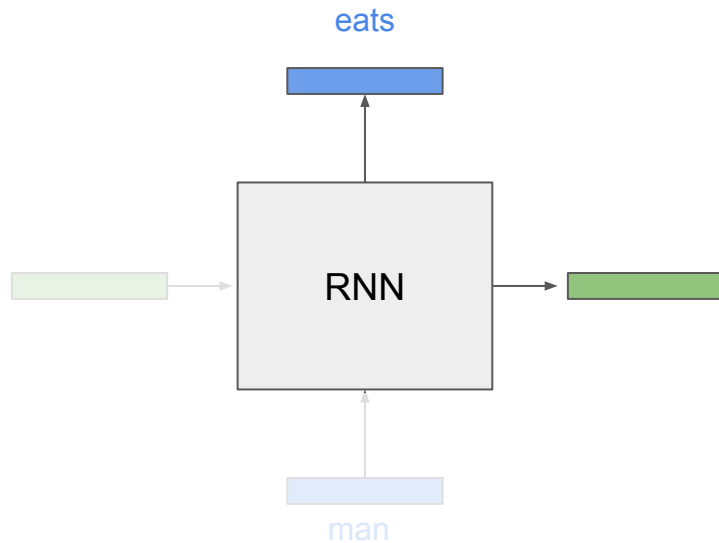
RNN Example: Text Generation



RNN Example: Text Generation

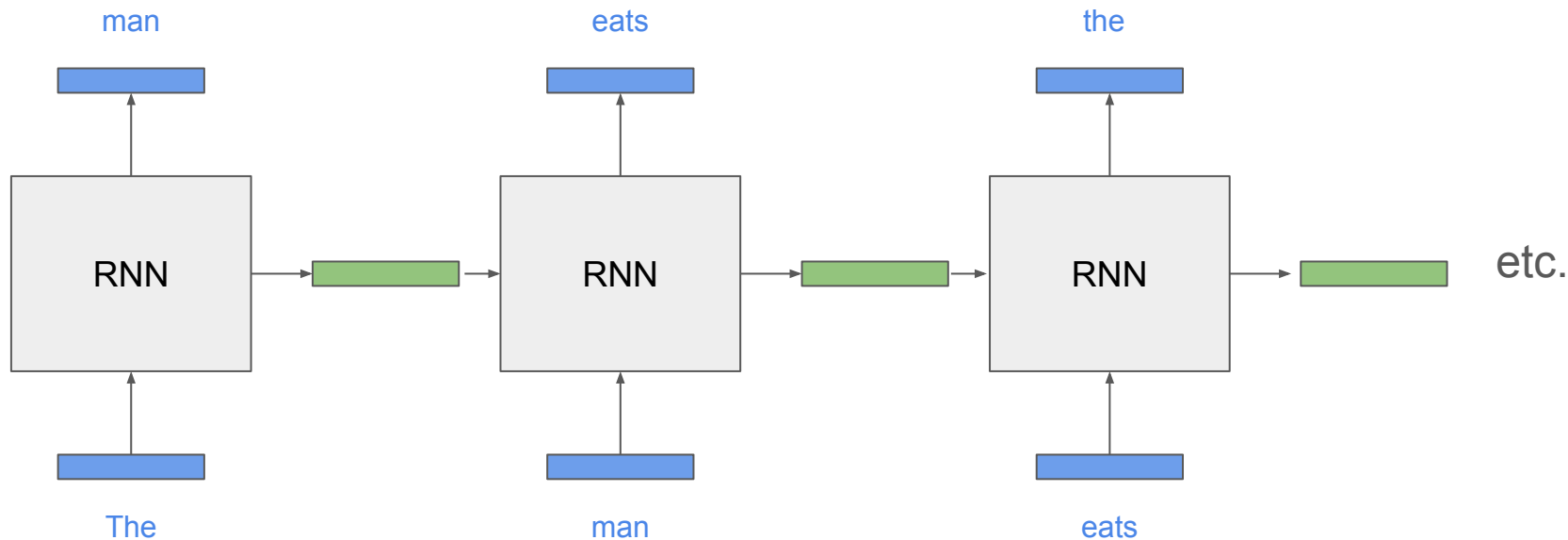


RNN Example: Text Generation



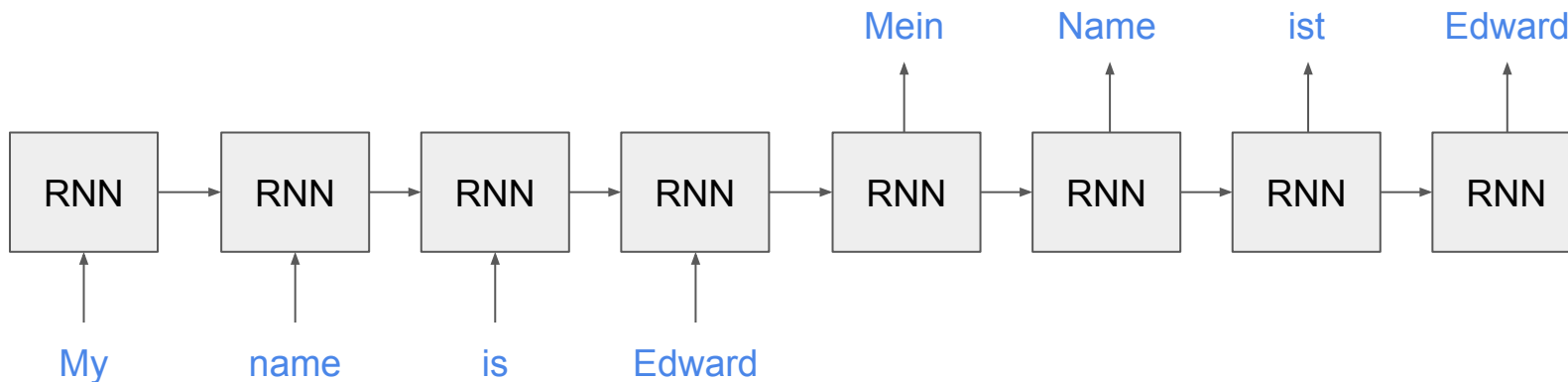
RNN Example: Text Generation “Unrolled” View

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



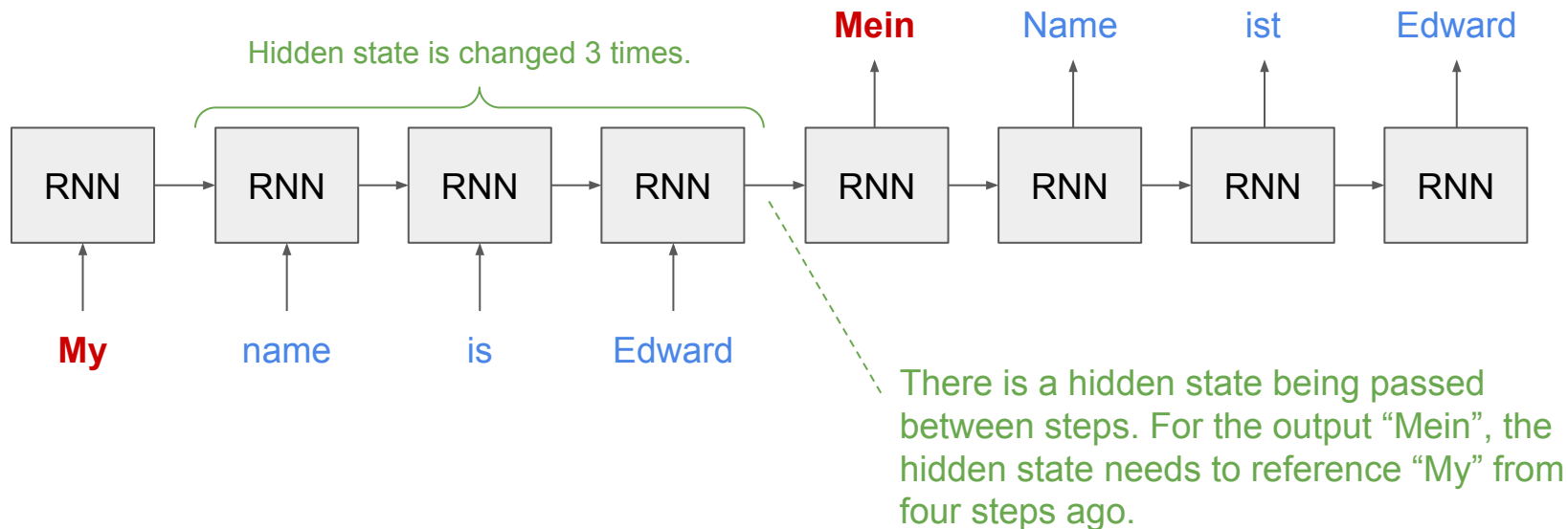
RNN Example: Translation

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



RNN Example: Translation

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



RNN Problems

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

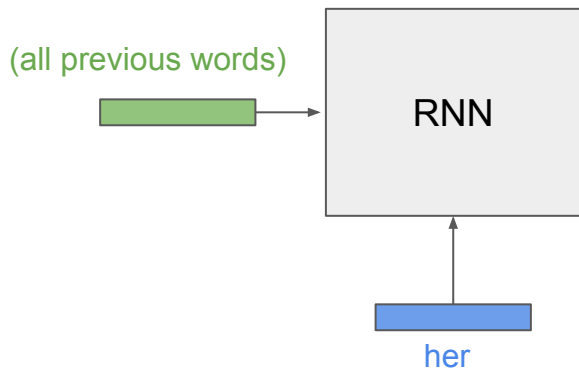
Catherine II[a] (born Princess Sophia Augusta Frederica von Anhalt-Zerbst; 2 May 1729 – 17 November 1796),[b] most commonly known as **Catherine the Great**,[c] was the reigning empress of Russia from 1762 to 1796.[1] She came to power after overthrowing her husband, Peter III. Under her long reign, 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

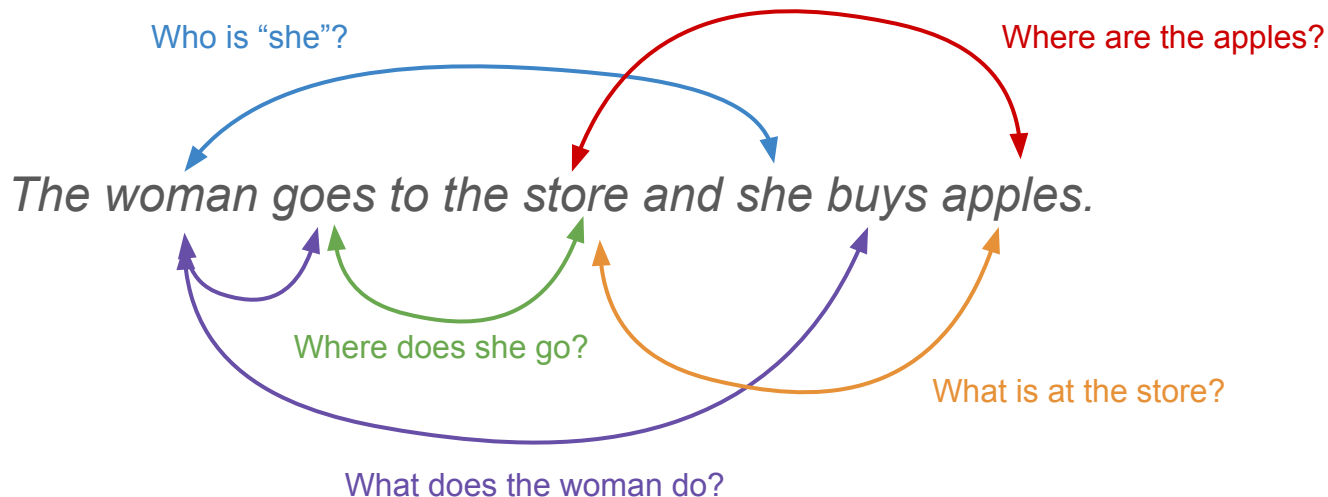
Another way to understand this problem is that we need to be able to make comparisons between words that may be separated by some distance.



We are hoping that the word “her” can correctly be compared against “Catherine the Great” that is captured in the hidden state.

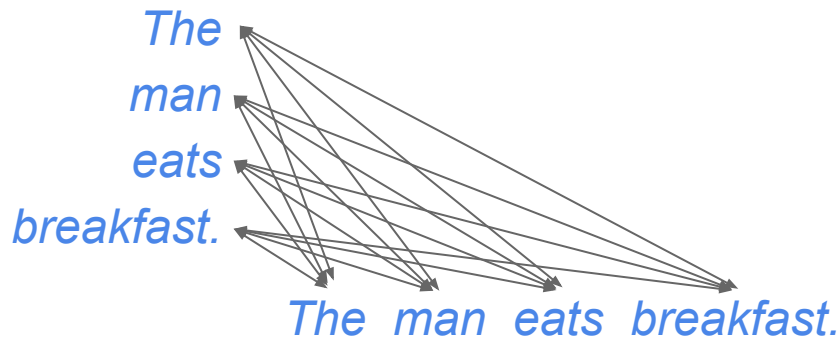
RNN “comparison” view

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

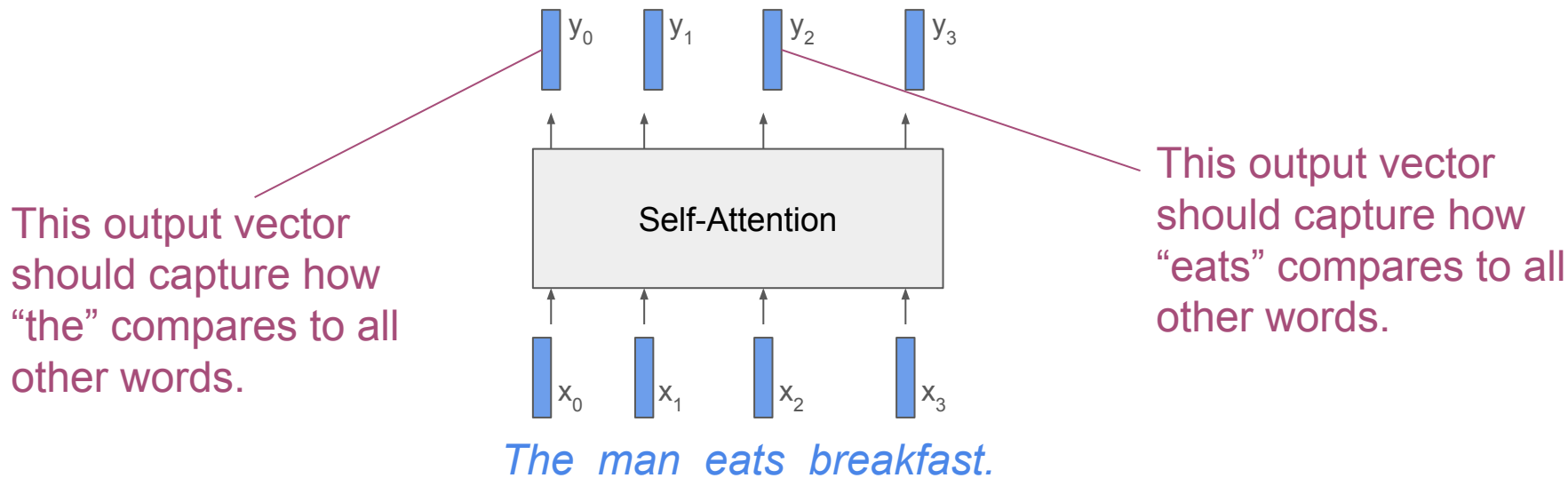
The transformer is built on a mechanism called “self-attention”. The main idea is to compare every word to every other word.



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:



Weighted Average

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

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



This weighting term somehow captures how x_i compares with x_j .

The man eats breakfast.

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:

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

Weighted Average (Continued)

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

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

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 it 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:

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

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

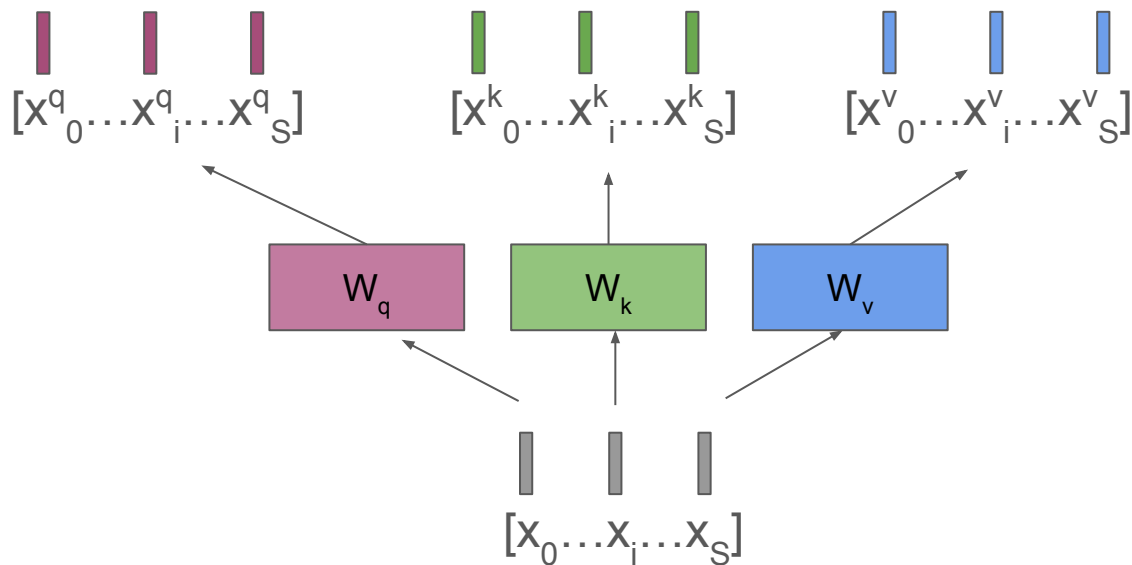
Because these are three separate roles, we learn a separate matrix transformation of x_k for each of them. Rather than using x directly we use the output of the transformations:

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

Block-Diagram Version

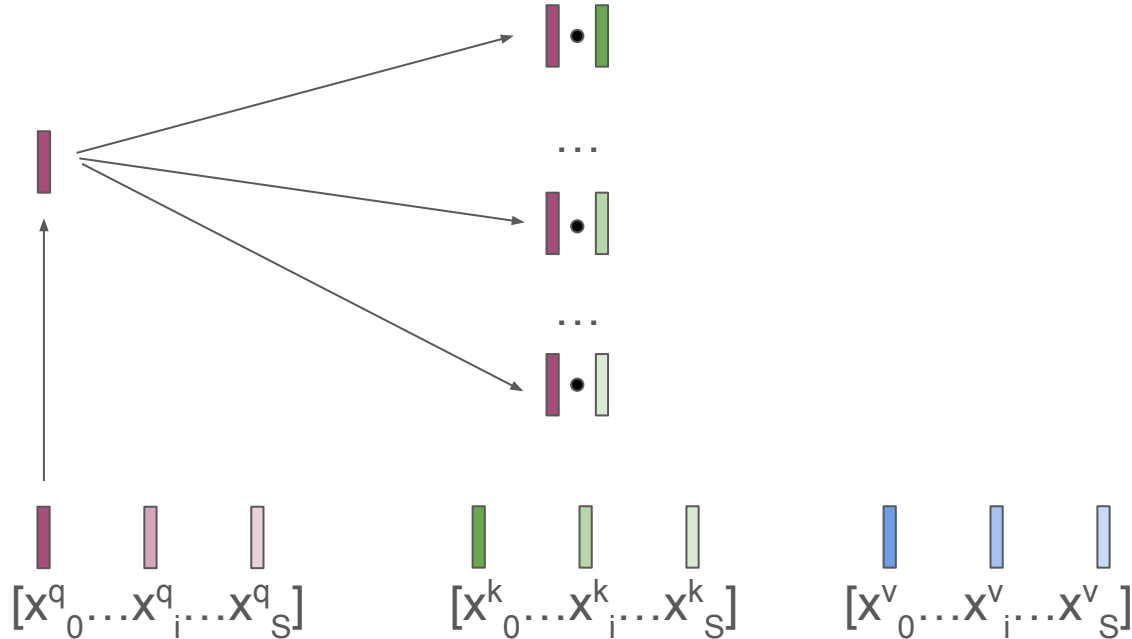
If it is helpful, we can try to draw a diagram of this. It gets messy very quickly.

Step 1: All input vectors \mathbf{x} are transformed into queries, keys and values:



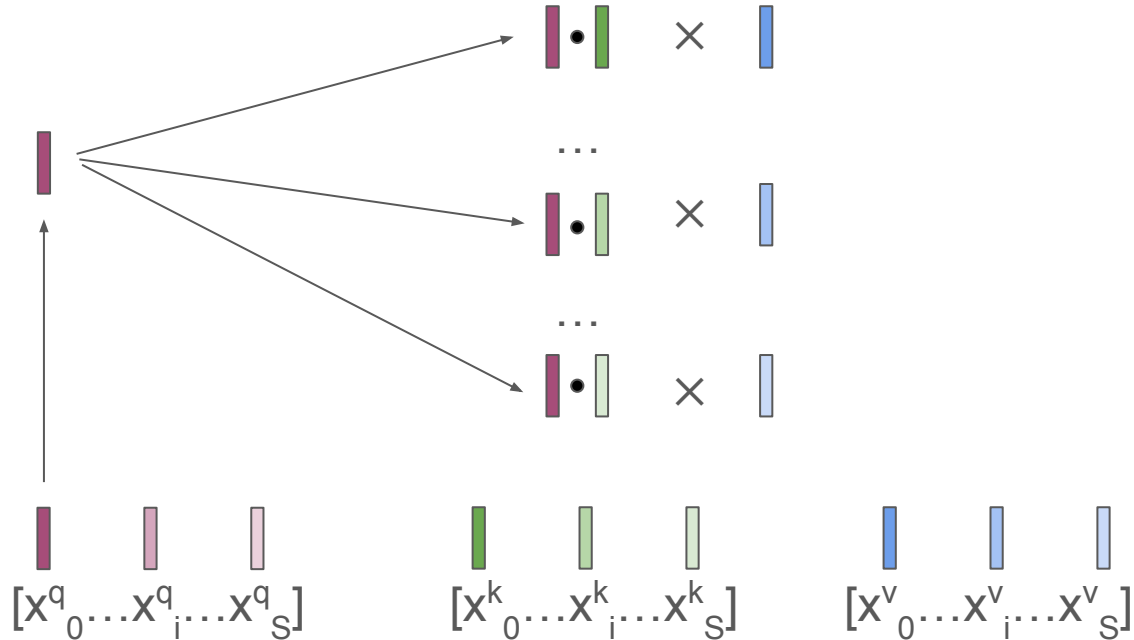
Block-Diagram Version

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



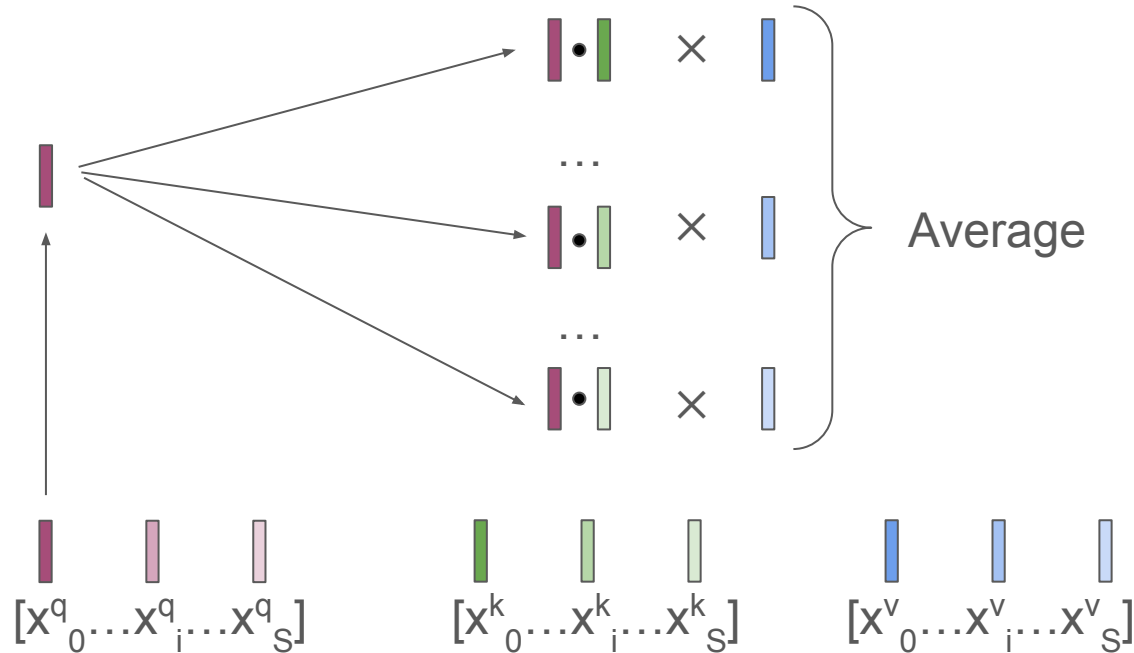
Block-Diagram Version

Step 3: This weights the values.



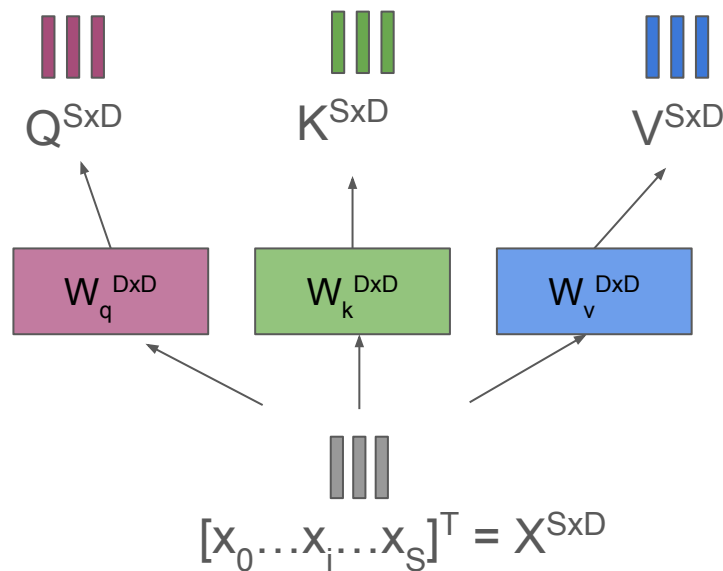
Block-Diagram Version

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



Matrix Version

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



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:

$$\text{Weighting terms (size } D \times D) = QK^T = \begin{array}{|c|c|c|c|} \hline \text{purple} & \text{purple} & \text{purple} & \text{green} \\ \hline \text{purple} & \text{purple} & \text{purple} & \text{green} \\ \hline \text{purple} & \text{purple} & \text{purple} & \text{green} \\ \hline \text{purple} & \text{purple} & \text{purple} & \text{green} \\ \hline \end{array}$$

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

$$\text{Output } Y = (QK^T)V = \begin{array}{|c|c|c|c|c|c|} \hline \text{purple} & \text{purple} & \text{purple} & \text{green} & \text{blue} & \text{blue} \\ \hline \text{purple} & \text{purple} & \text{purple} & \text{green} & \text{blue} & \text{blue} \\ \hline \text{purple} & \text{purple} & \text{purple} & \text{green} & \text{blue} & \text{blue} \\ \hline \text{purple} & \text{purple} & \text{purple} & \text{green} & \text{blue} & \text{blue} \\ \hline \end{array}$$

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} :

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

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)

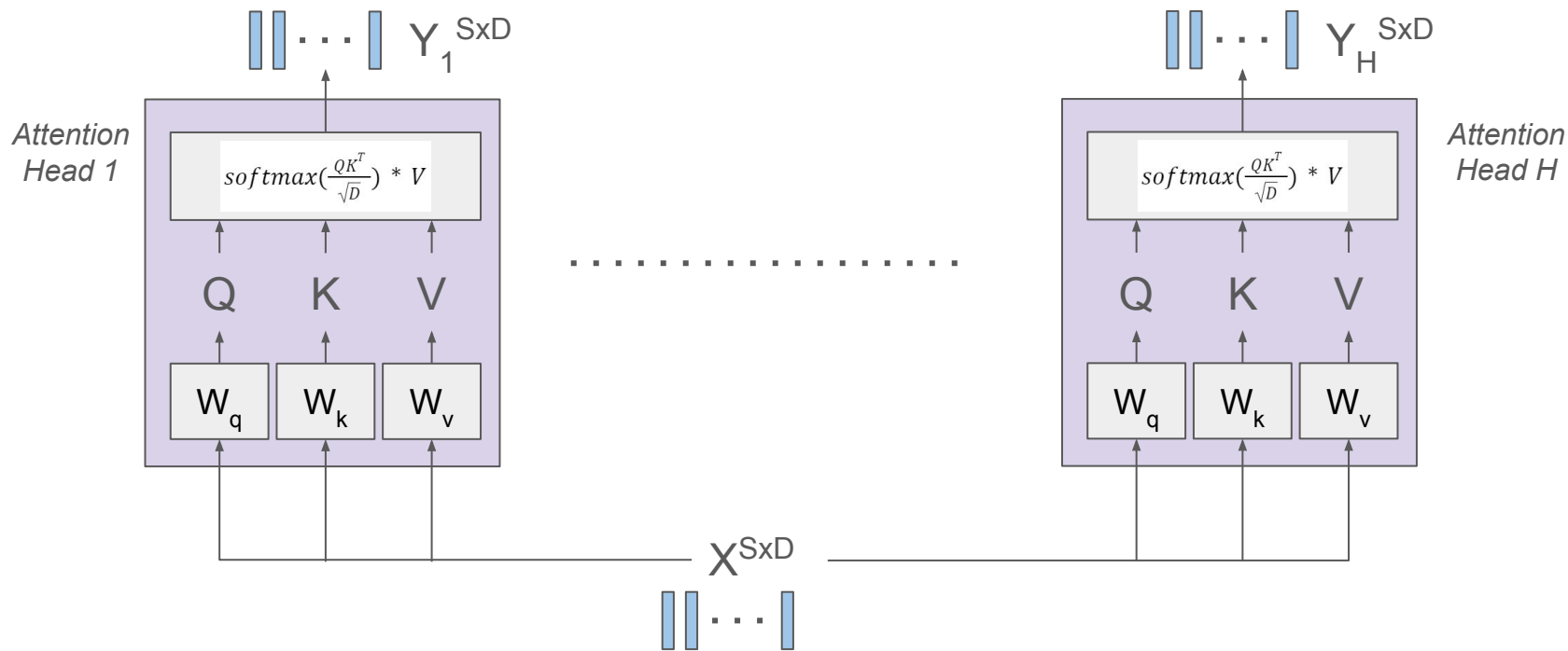
The angry man walked the poodle.

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 H different times. 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 $3 \cdot H$ weight vectors of size $D \times D$) and memory (we also have to track $3 \cdot H \cdot S$ vectors of size D).

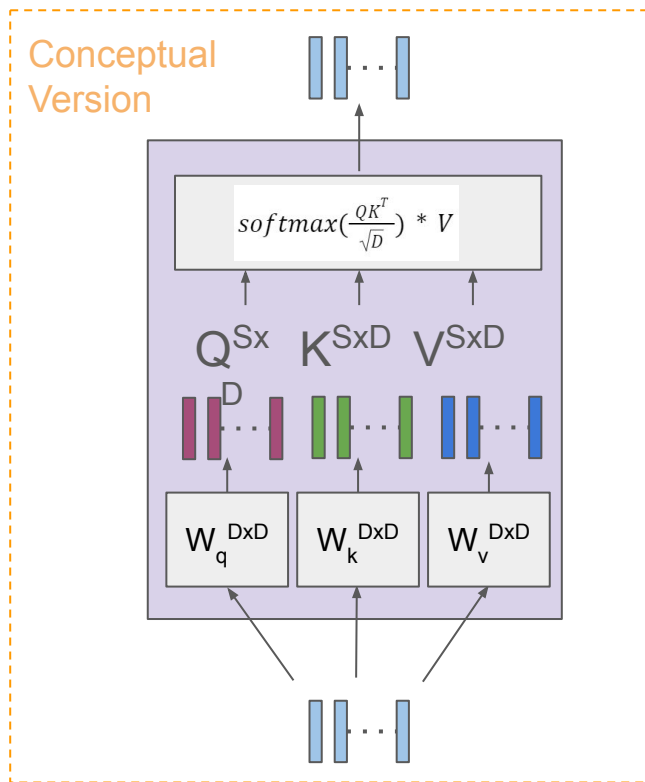
We make 3 major improvements:

- 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.

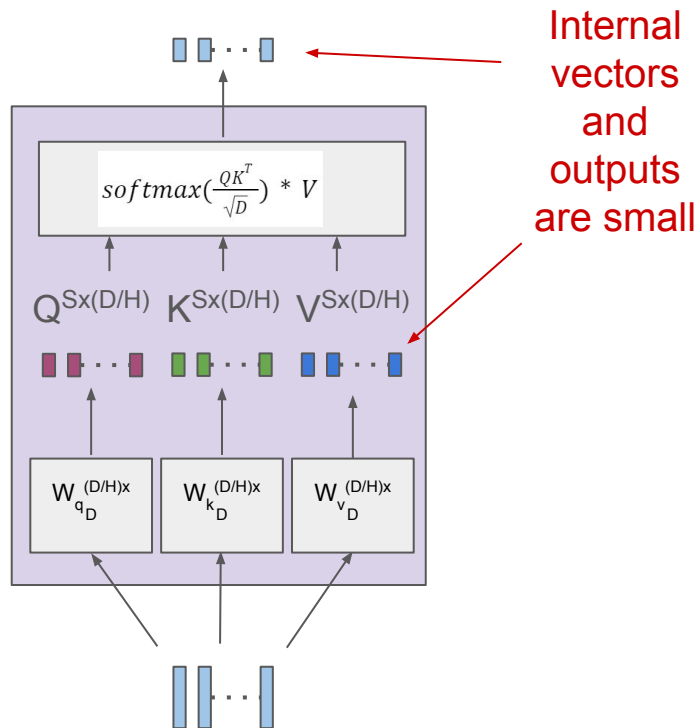
We are going to look at these one at a time.

Improvement (1): Smaller Vectors Internally

Queries, keys, and values use a smaller vector size, D/H :

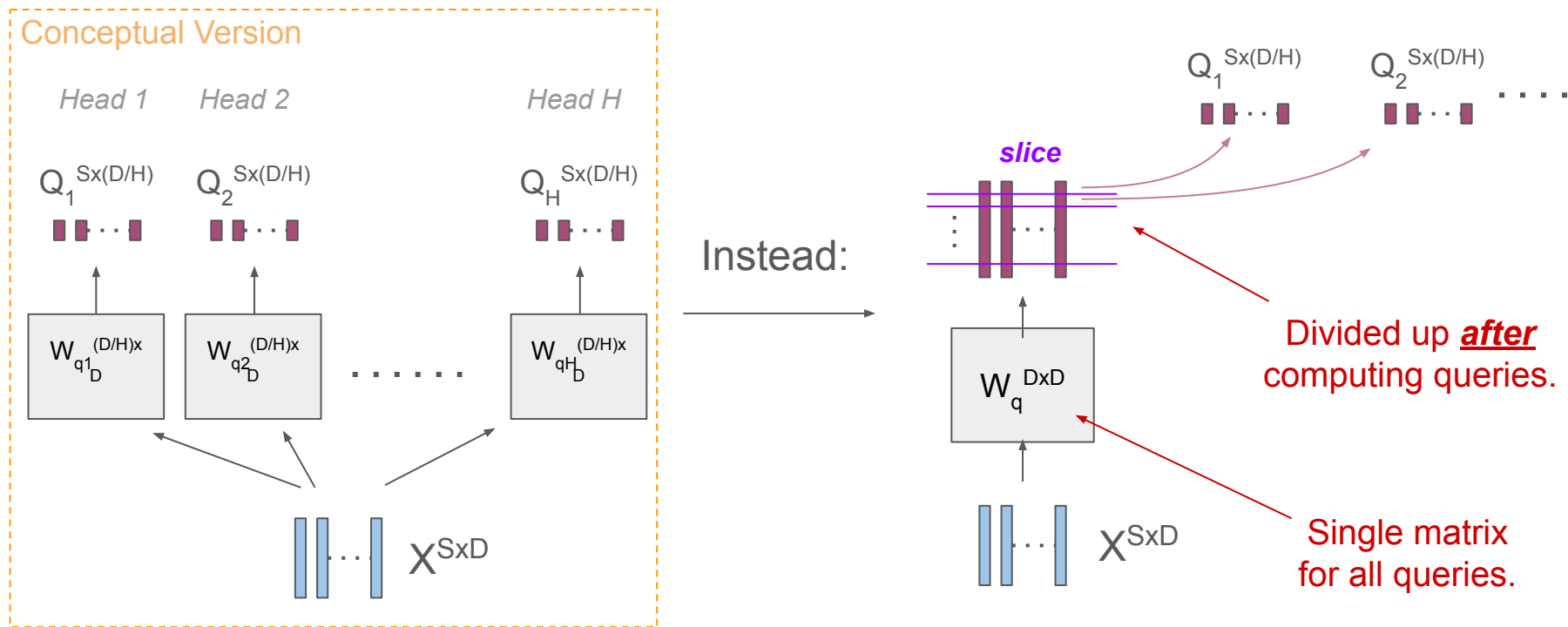


Instead:



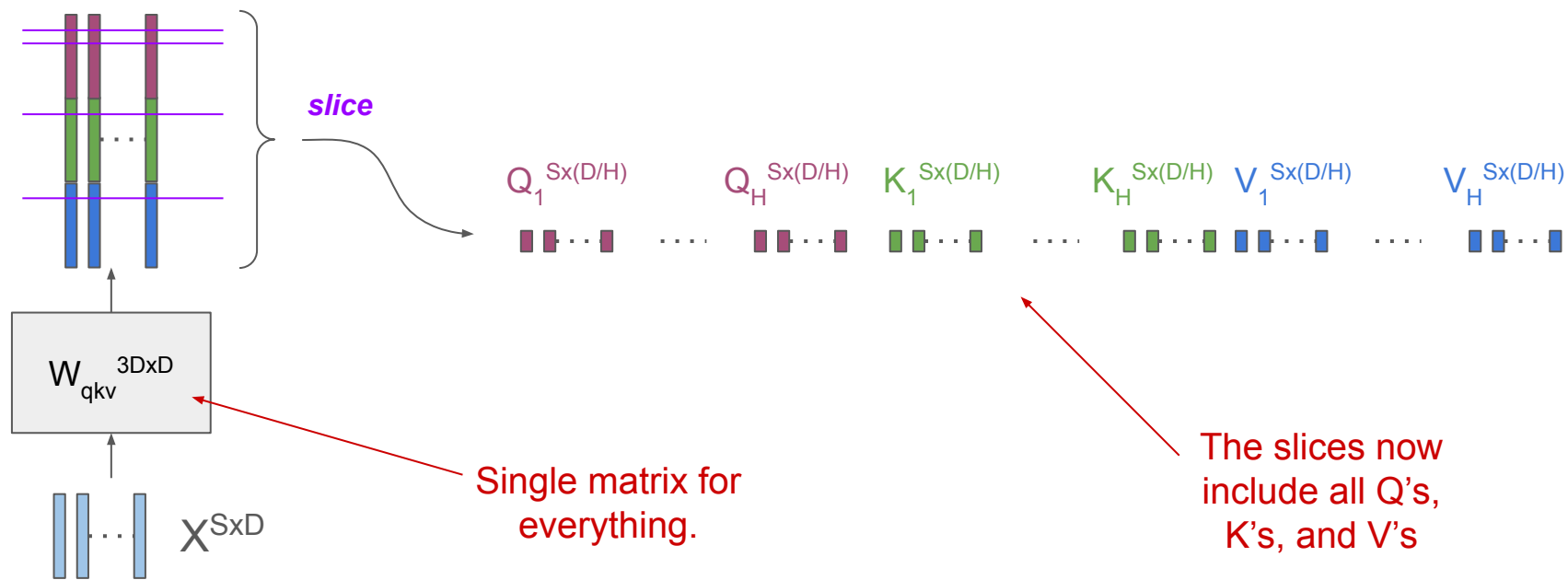
Improvement (2): Shared Weights

Instead of H matrices of size $((D/H) \times D)$, use one matrix of size $(D \times D)$ and then slice the vectors:



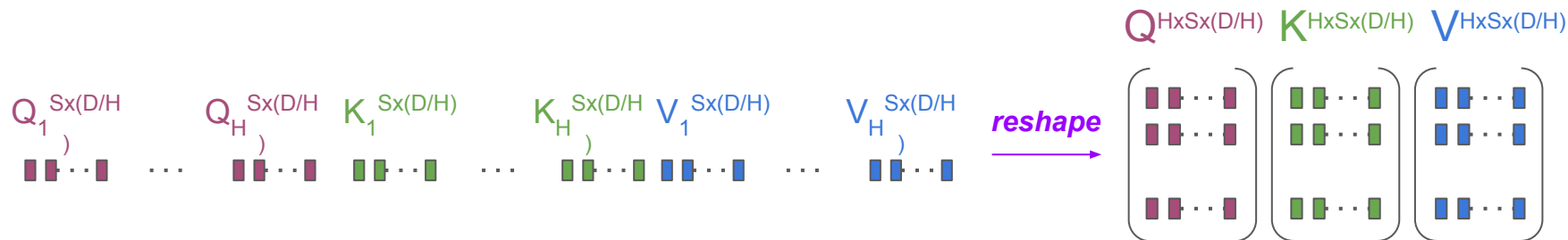
Improvement (3): Giant Q/K/V Computation

Replace $W_q^{D \times D}$, $W_k^{D \times D}$, $W_v^{D \times D}$ with a single $W_{qkv}^{3D \times D}$:



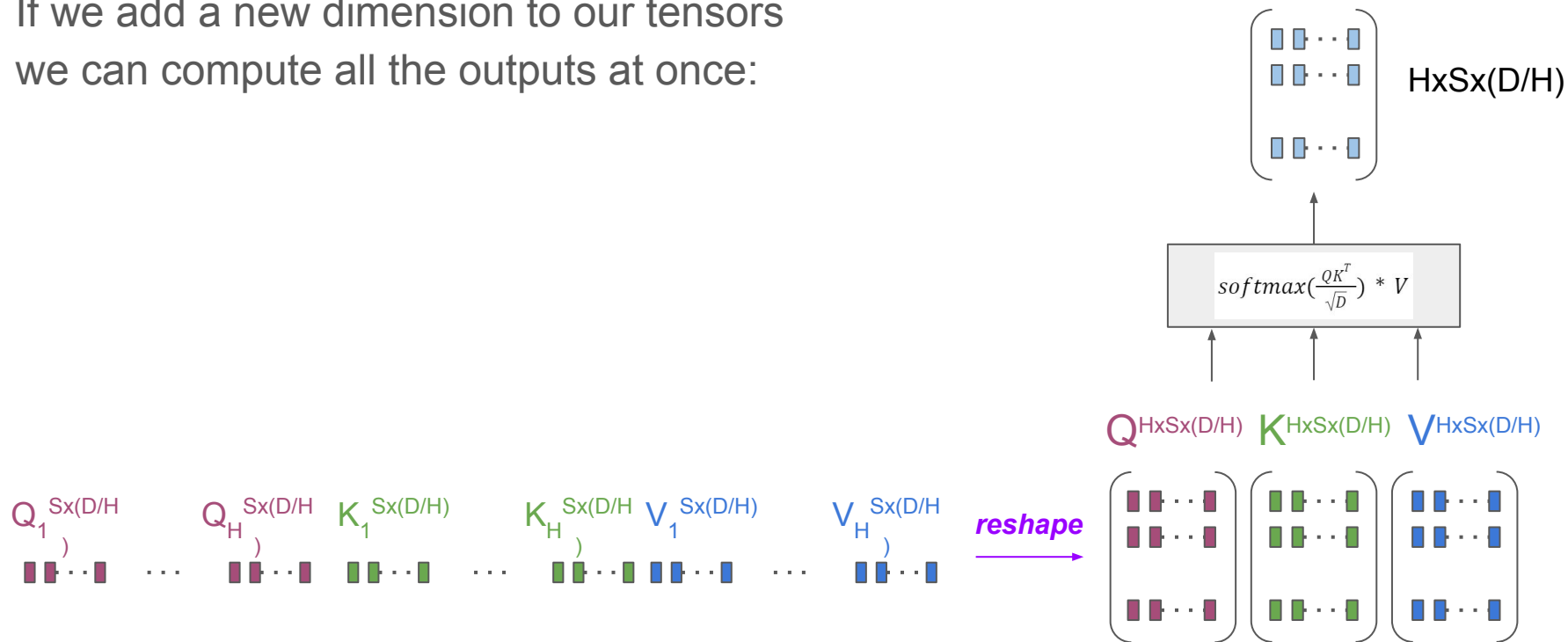
Batched Attention

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



Batched Attention

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



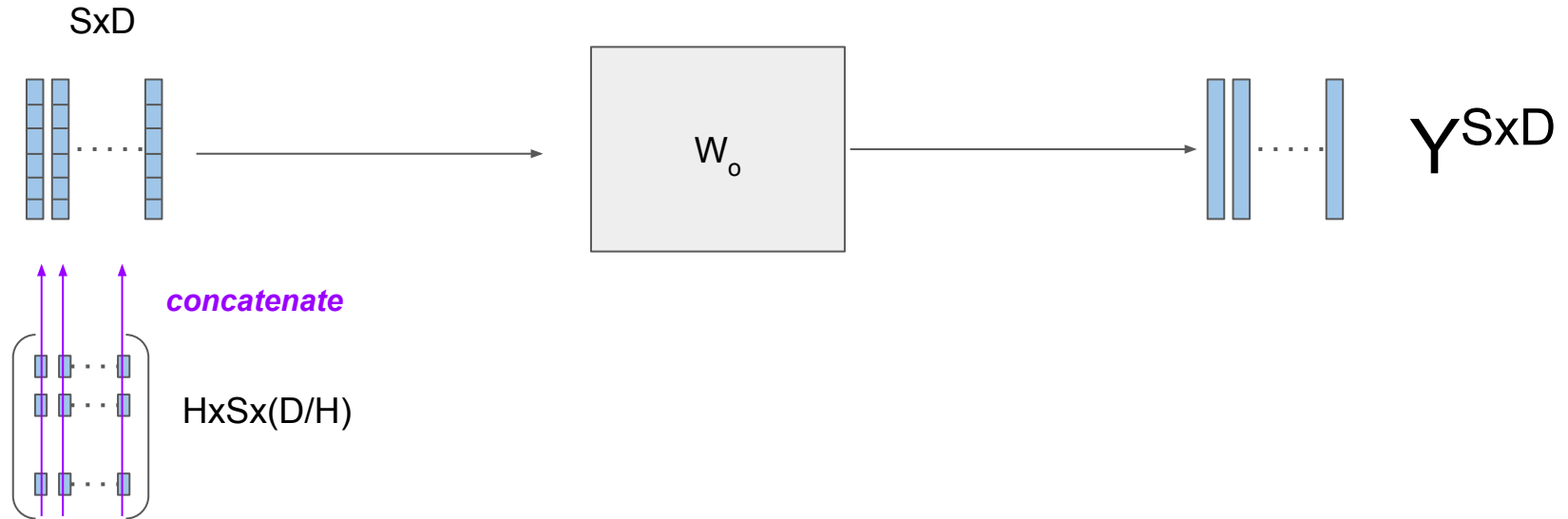
Batched Attention

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



Final Step

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



MHA Summary

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

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

Perform matrix multiplication $(\mathbf{X})(\mathbf{W}_{qkv}^T)$ to get tensor \mathbf{T} size $(S \times 3D)$

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

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

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

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

Details: Matrix Multiplication in PyTorch

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

$$A^{N \times M \times O} B^{N \times O \times P} = C^{N \times M \times P} \quad \text{same as: } N \text{ parallel instances of } A^{M \times O} B^{O \times P}$$

This also applies with additional leading dimensions:

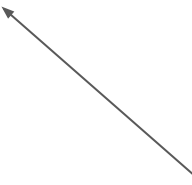
$$A^{L \times N \times M \times O} B^{L \times N \times O \times P} = C^{L \times N \times M \times P} \quad \text{same as: } L * N \text{ parallel instances of } A^{M \times O} B^{O \times P}$$

Details: Applying Attention Equation

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

The first step of applying attention is to take \mathbf{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 $[\mathbf{QK}^T]^{H \times S \times S}$



This $S \times S$ 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 $\mathbf{Q}^{H \times S \times (D/H)}$, $\mathbf{K}^{H \times S \times (D/H)}$, $\mathbf{V}^{H \times S \times (D/H)}$

The first step of applying attention is to take \mathbf{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 $[\mathbf{QK}^T]^{H \times S \times S}$

We transpose the last two axes of \mathbf{K} into $\mathbf{K}^{H \times (D/H) \times S}$. This is \mathbf{K}^T .

Multiplying together we get $(H \times S \times (D/H)) \times (H \times (D/H) \times S) \Rightarrow (H \times S \times S)$, representing $H \times S$ 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: With Batches

For a batch of training samples everything just has an extra dimension out front.

$$\mathbf{X}^{S \times D} \longrightarrow \mathbf{X}^{B \times S \times D}$$

$$\mathbf{Q}^{H \times S \times (D/H)} \longrightarrow \mathbf{Q}^{B \times H \times S \times (D/H)}$$

$$\mathbf{Y}^{S \times D} \longrightarrow \mathbf{Y}^{B \times S \times D}$$

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!

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.]])
```


Tensor Manipulation Cheatsheet: Repeating and Splicing

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])
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.]])
```