CS 1053

TRANSFORMERS

1/27/26

SLIDES BORROWED/ADAPTED FROM KYLE WILSON (THANKS KYLE!)

Plan for today

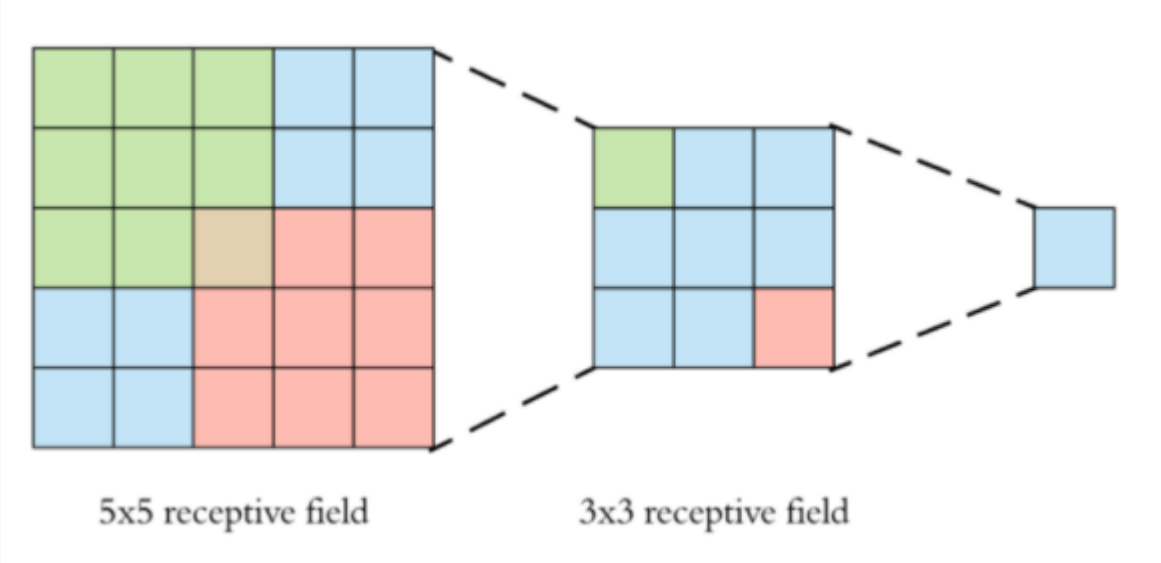
Motivation: what's wrong with CNNs?

Context: Encoders and Decoders

Transformers:

1. For language
2. For vision (???!)

What's missing from CNNs?



- Good at modeling local relationships
- Bad at modeling long-range dependencies

"receptive field" of a neuron: all pixels in the input image that affect the value of the neuron.

[Image source]

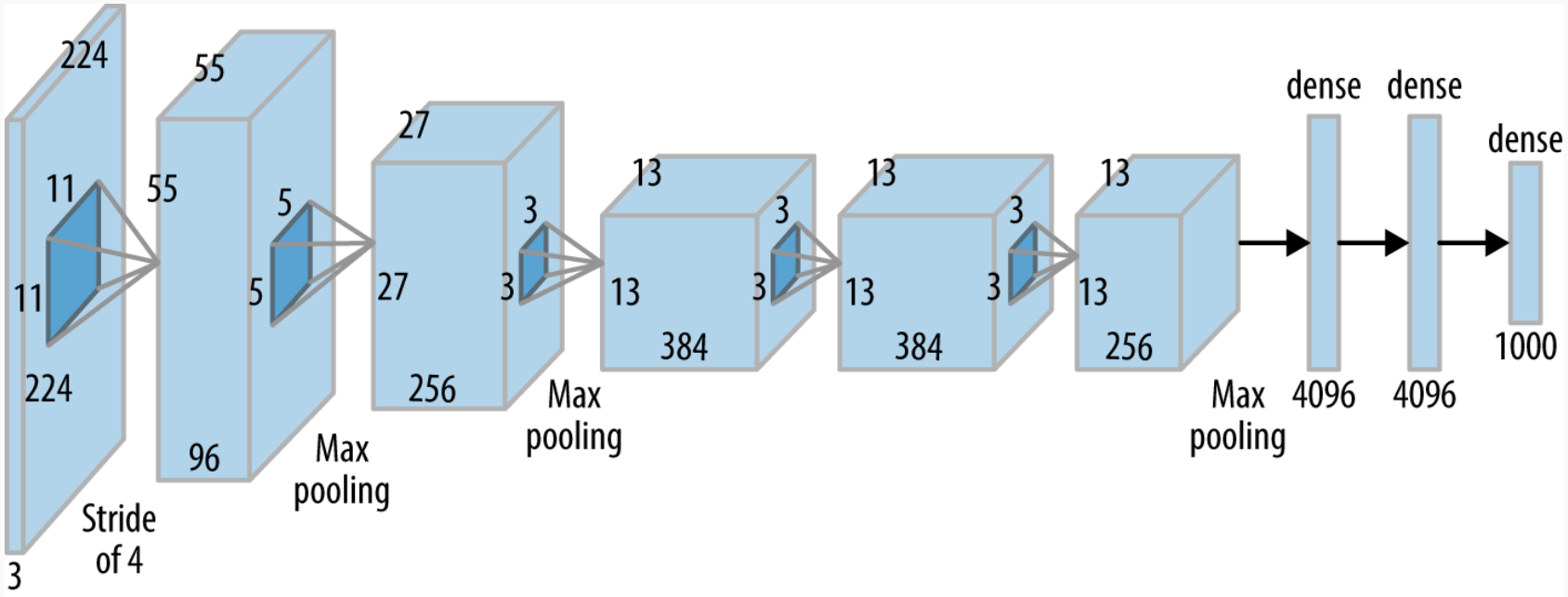
What’s missing from CNNs?

- Fitting long-range relationships within an image
- Expensive to train
  - Need lots of compute
  - Need lots of training data
- CNNs are sort of black boxes

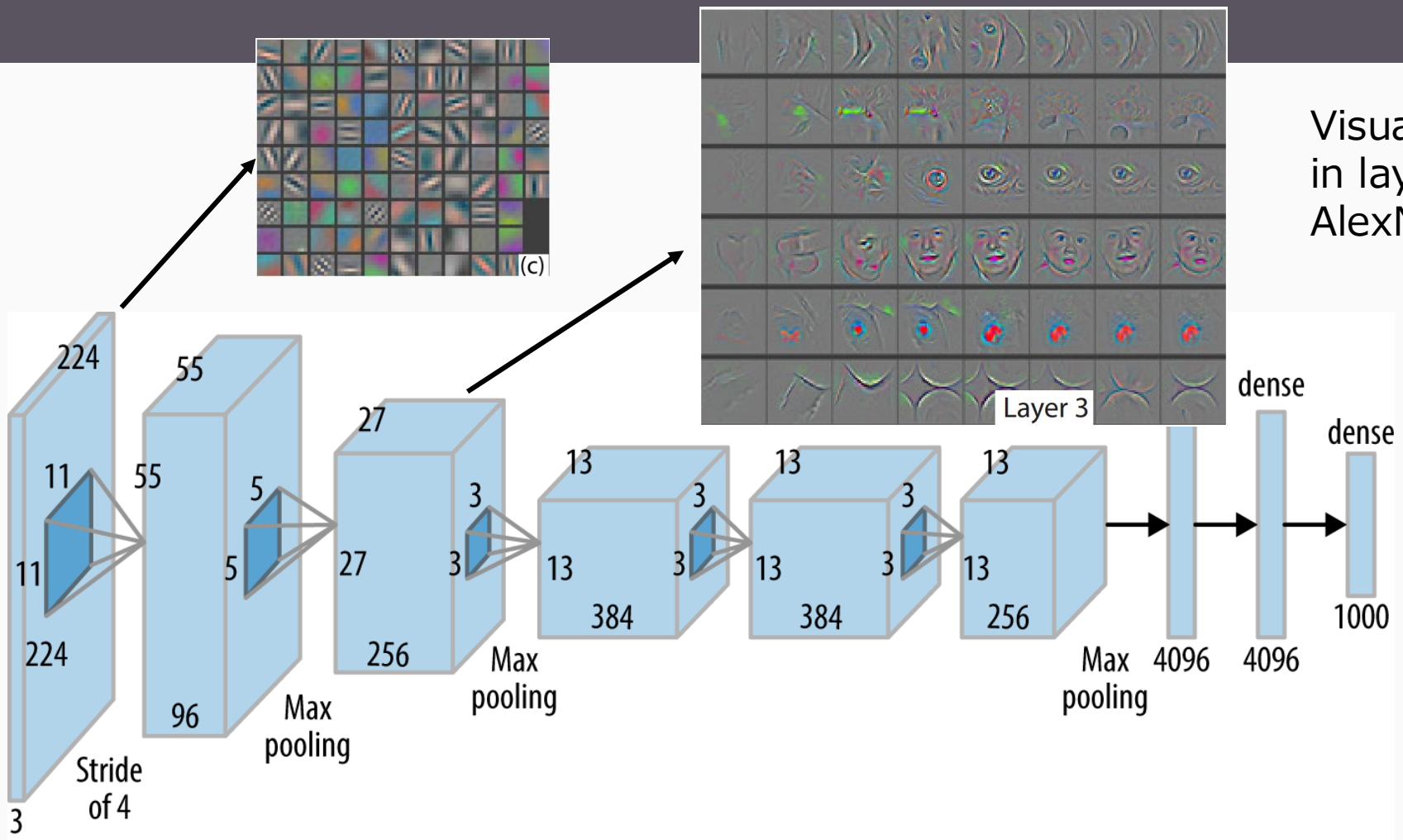
ENCODERS AND DECODERS

Recall: CNNs are hierarchical functions

- The input to layer i is the output of layer i-1



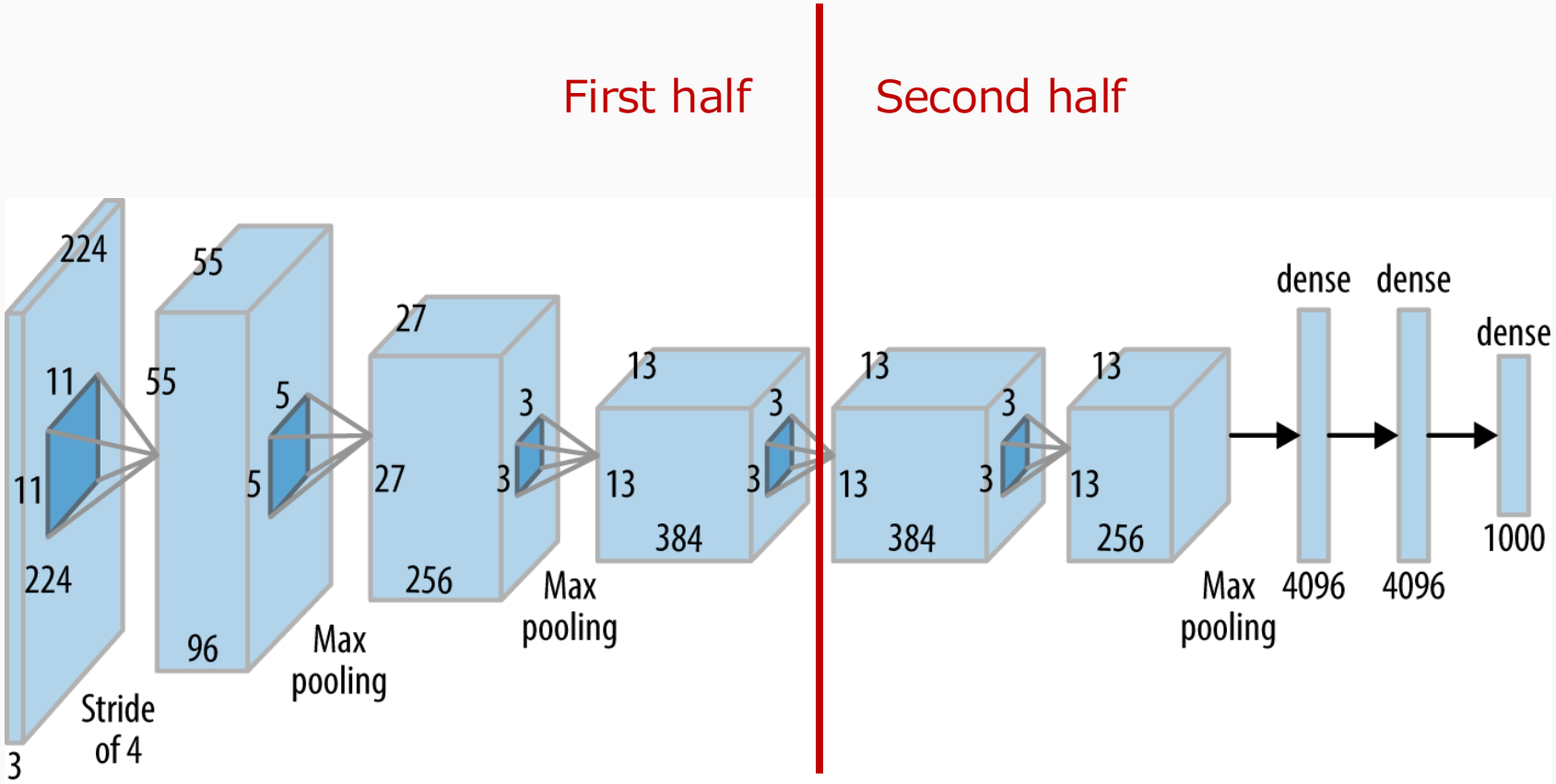
Recall: CNNs are hierarchical functions



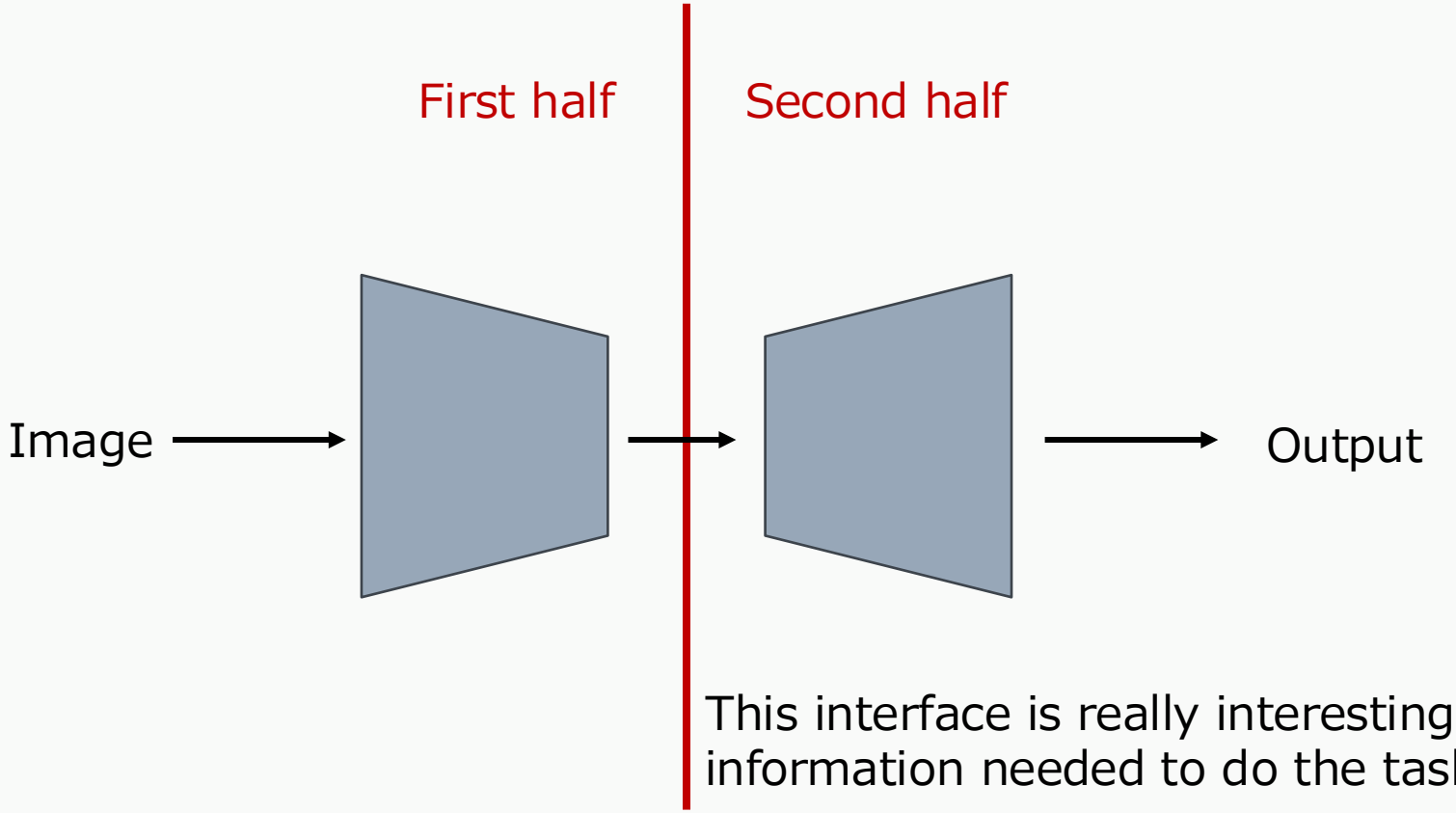
Visualizations of features in layers 1 and 3 of AlexNet. [source]

Layer 2 feature are made from Layer 1 output. Each layer gets less localized and more abstract.

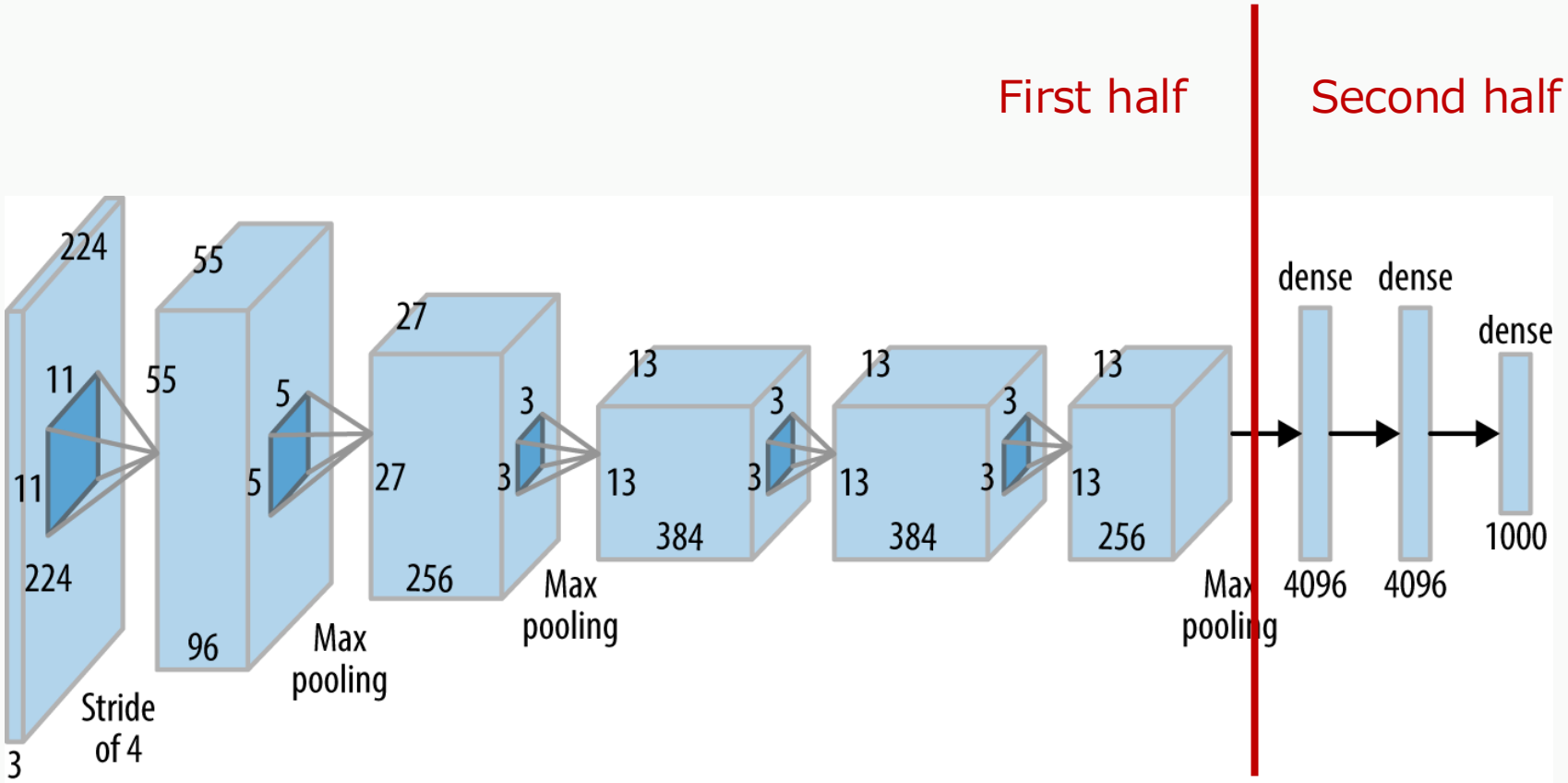
Imagine “slicing” a network in half



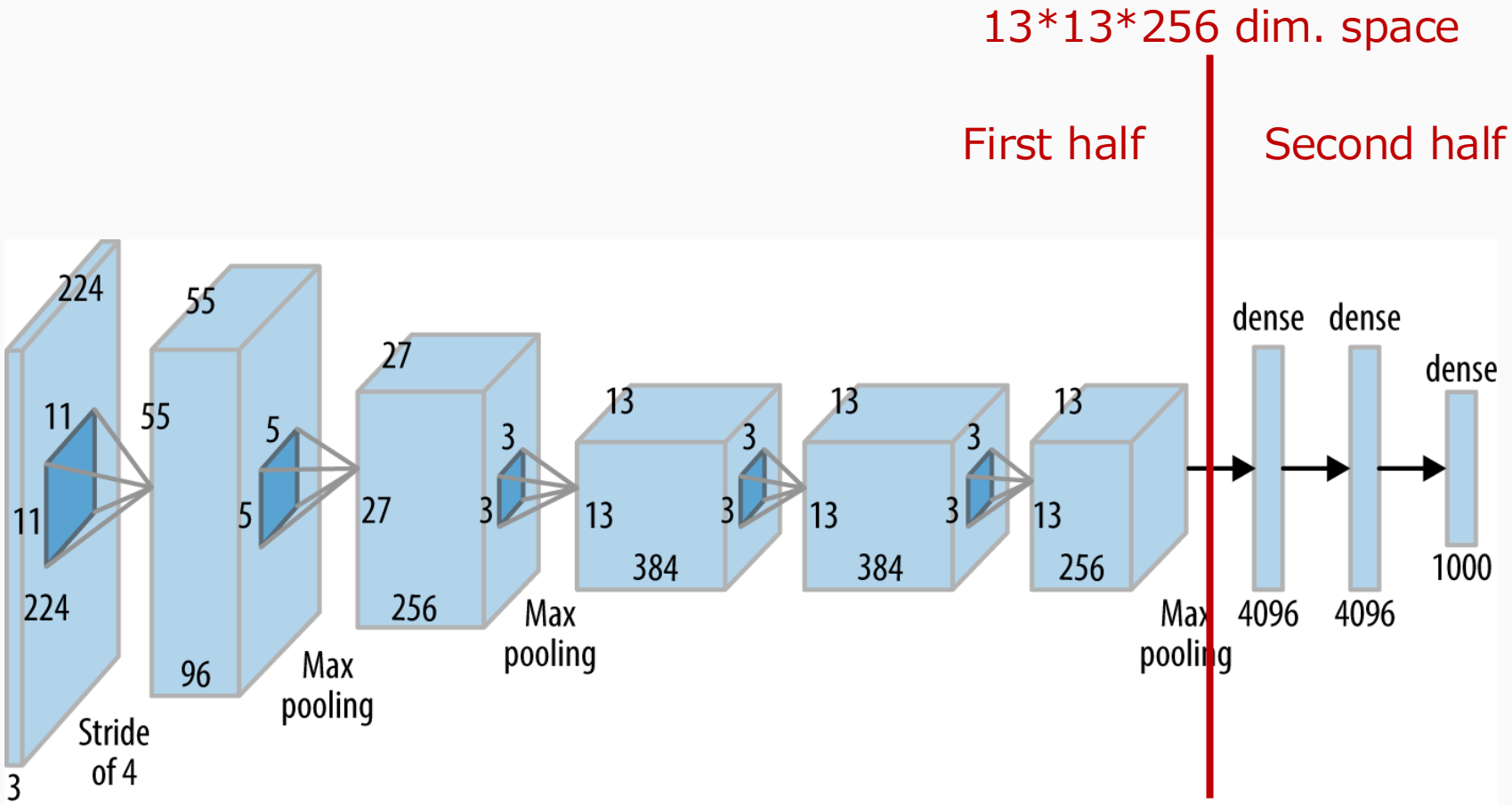
Imagine “slicing” a network in half



There's a natural place to slice this network.



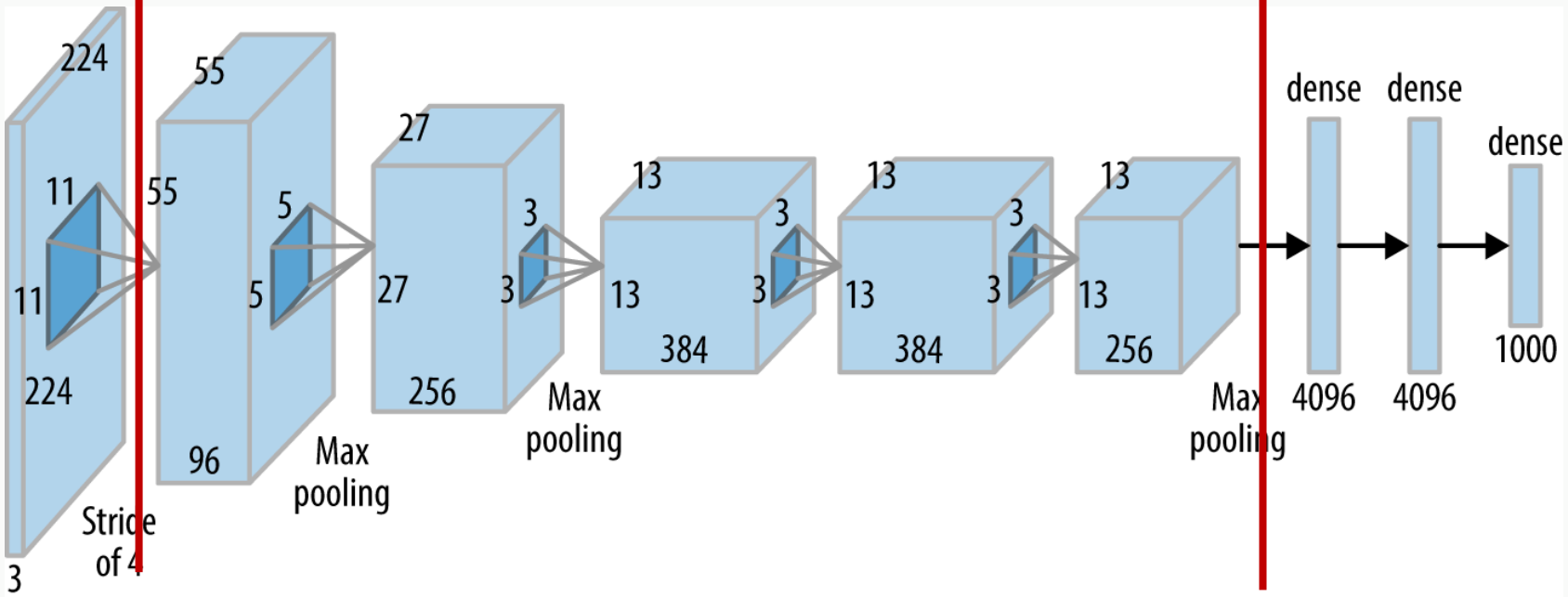
The space in-between is called the latent space.



Why are some splits special?

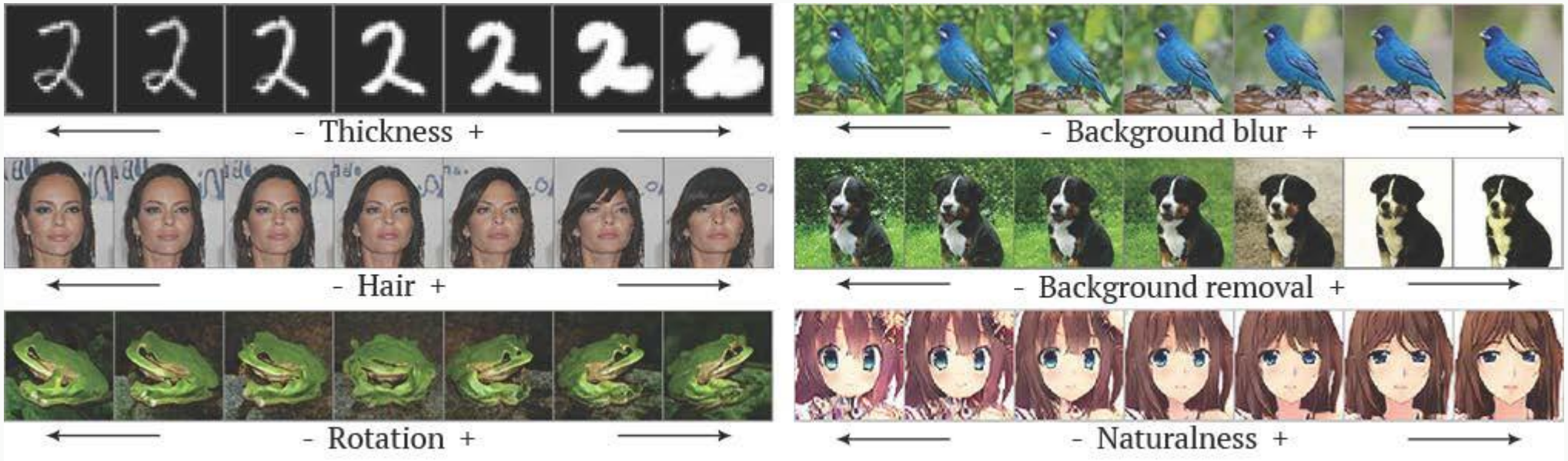
A bad latent space has diffuse information. Individual numbers are meaningless without context.

A good latent space is compressed and organized. Individual numbers are highly meaningful.



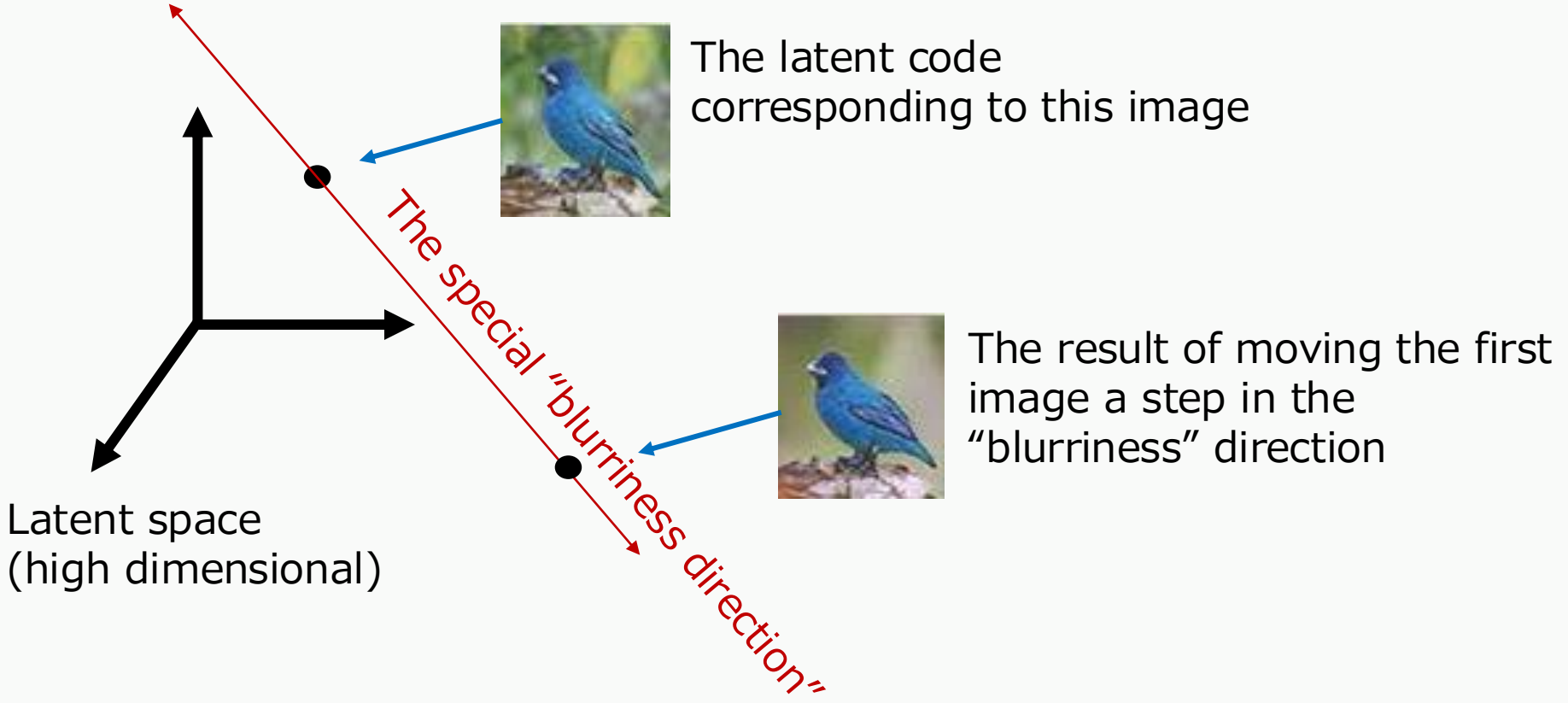
Interpretable Latent Spaces

Cool research result! They studied the latent space of an image generation model. They found directions that carried semantic meaning. Moving in a direction would create more of an effect, and moving opposite would lessen it.



[source]

Interpretable Latent Spaces



Latent Spaces: summary

- A good latent space has all the *relevant* information from the original image.
- More info stored in a much smaller space: high information density
- In practice, the higher the information density is, the more organized it must be

Latent Spaces are useful!

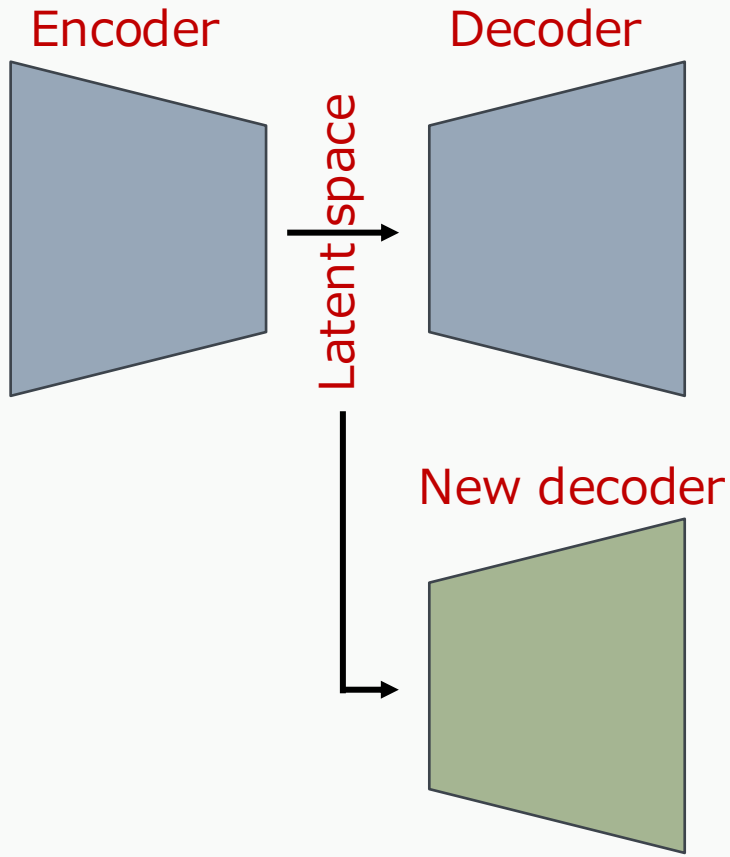
Plan:

1. Solve a really hard vision problem on the biggest dataset you can find.

2. Design the architecture to have a bottleneck

3. Cut off the part after the bottleneck

4. Repurpose the part before the bottleneck for new problems



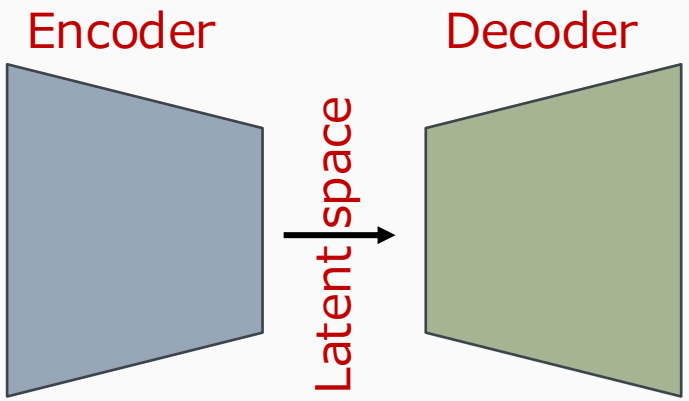
Latent Spaces are useful!

Encoder:

- Lots of parameters
- Very expensive to train
- Reuseable component
- “pretrained”

Decoder:

- Relatively simple
- Problem-specific
- Not the hard part



Latent Spaces are useful!

My mental picture:

Encoder

Expensive.
Output is a generically useful “latent space”

Problem-Specific Decoders

Often cheap.
Does one simple job.

Check-in: What's missing from CNNs?

- Fitting long-range relationships within an image
- Expensive to train
  - Need lots of compute
  - Need lots of training data
- CNNs are sort of black boxes

The encoder/decoder pattern greatly helps on both of these issues!

ATTENTION

Framing the problem

Long-range dependencies:

- Two or more areas of an image,
- that aren’t next to each other
- but need to be considered jointly to solve a problem

Example: identify pictures of a “soccer team”

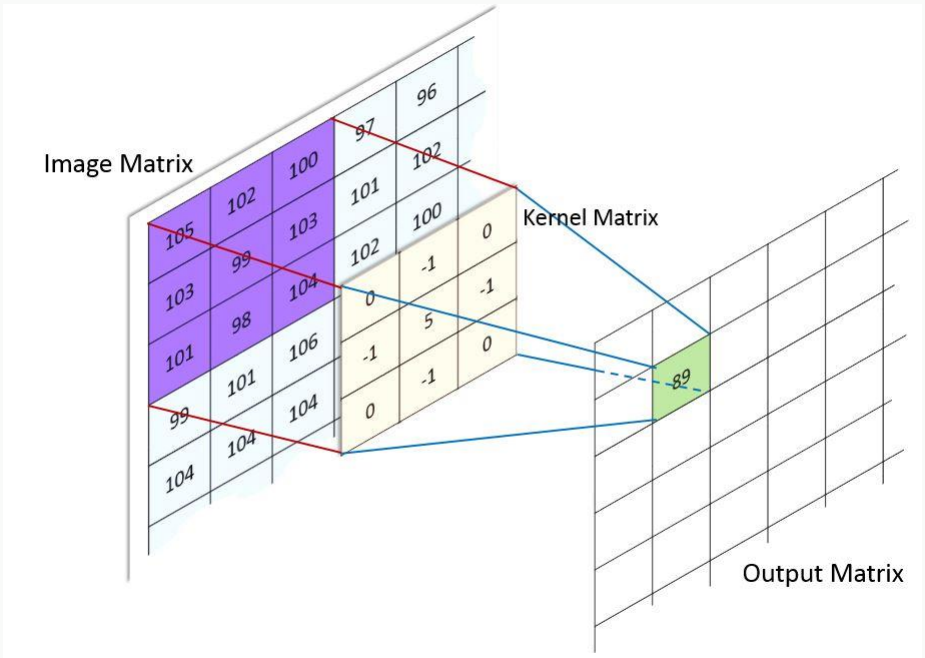
- person / ball / grass / matching uniforms...

Framing the problem

Convolutions make **local comparisons**.

Deeper nets = larger receptive field.
This sort of works.

Can we do better?
Is there a model that naturally makes long-range comparisons? Need a new idea.



[image source]

Roadmap: first NLP, then Vision

Natural Language Processing also has the long-range comparisons problem.

Consider:

"The **animal** didn't cross the street because **it** was too tired."

The equations for Self-Attention arose in NLP. Then we figured out how to use the same model in vision. I'll present the NLP version of Attention first.

Illustrative Example: Attention

Some words mostly carry stand-alone meaning.

Other words' meaning are highly context-dependent.

In these examples, which words are the correct context for interpreting the word "it"?

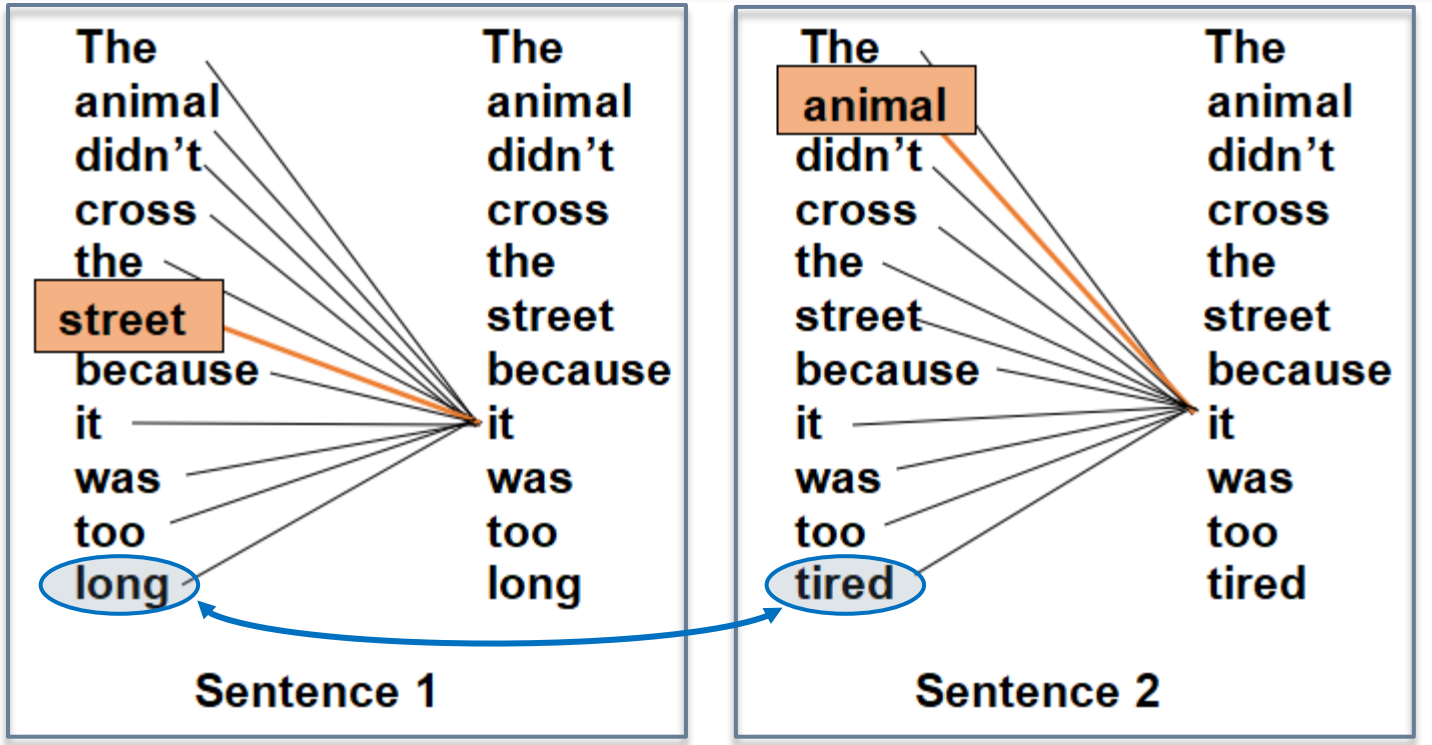


Figure 5: self-attention example

[image source]

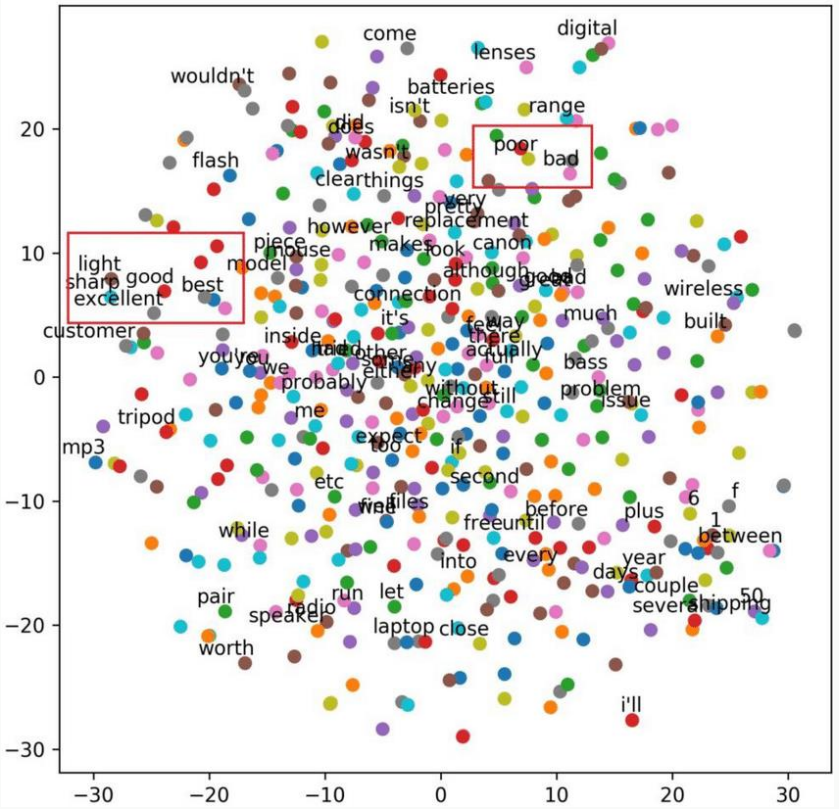
Some formalisms

We're given a sentence. We want to train an encoder to understand it.

Number the words in the sentence: $w_1, w_2, \ldots, w_{N_w}$.

Words aren't very convenient for computation. We'll **embed** each word in an N_e dimensional embedding space.

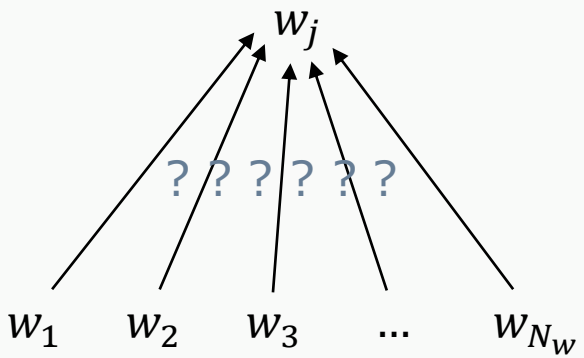
Call these embedding vectors $x_1, x_2, \ldots, x_{N_w}$



[image source]

Some formalisms

Let's pick one word (say, w_j) from the sentence. We'll just talk about how to understand that word. Which of the other words are relevant?



Is there some sort of "similarity" function $f(w_j, w_i)$ that can measure the relevance of each word w_i to understanding w_j?

We wouldn't want to code that imperatively, but it makes a great machine learning problem.

**Footnote: if this was an NLP class, we'd talk about how "words" aren't quite the right thing to use here. Instead, NLP uses "tokens". But "words" is close enough for what we're doing today.*

Learned similarity functions

The **dot product** is a sort of similarity function. It's efficient to compute and lends itself well to both theory work and to vectorized code.

$$\mathbf{a} = \langle a_1, a_2, a_3 \rangle \quad \mathbf{b} = \langle b_1, b_2, b_3 \rangle$$

$$\boxed{\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + a_3 b_3}$$

But the dot product doesn't have parameters. It can't learn an attention relationship between words.

Learned similarity functions

This is the **normal dot product**:

$$\vec{q} \cdot \vec{k} = \vec{q}\vec{k}^{\top}$$

where $\vec{q} = [q_1, q_2, \ldots, q_{N_e}]$ and $\vec{k} = [k_1, k_2, \ldots, k_{N_e}]$ are $1 \times N_e$ row vectors.

Here's the **generalized dot product**:

$$\vec{q} \cdot \vec{k} = \vec{q}W_q {W_k}^{\top}\vec{k}^{\top}$$

where W_q and W_k are $N_e \times N_s$ matrices of learnable parameters. We'll train our model to pick parameters.

Queries and keys

Let's start using the vocabulary of attention. Our word of interest is our **query:** $\vec{q} = \vec{x_j}$. We want to know if a **key** $\vec{k} = \vec{x}_i$ is relevant context to our query.

So we compute the generalized dot product $\alpha_{ij} = \vec{q} W_q {W_k}^\top \vec{k}^\top$.

Summary: we have a learned similarity function that measures whether a given key is relevant context for a given query. Matrices W_q and W_k parameterize the function.

Queries and keys, vectorized

Recall: we compute the generalized dot product $\alpha_{ji} = \vec{q} W_q {W_k}^\top \vec{k}^\top$.

Do that for all keys, not just one. If we stack the keys row by row into an $N_w \times N_e$ matrix, then we can compute

$$[\alpha_{j1} \quad \dots \quad \alpha_{jN_w}] = \vec{q} W_q W_k^\top \begin{bmatrix} \vec{k}_1 \\ \vec{k}_2 \\ \vdots \\ \vec{k}_{N_w} \end{bmatrix}^\top$$

Since the keys are just our embedded words, let's write $\begin{bmatrix} \vec{k}_1 \\ \vec{k}_2 \\ \vdots \\ \vec{k}_{N_w} \end{bmatrix}^\top = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \\ \vdots \\ \vec{x}_{N_w} \end{bmatrix}^\top = X,$

so the expression becomes $\vec{q} W_q (X W_k)^\top$.

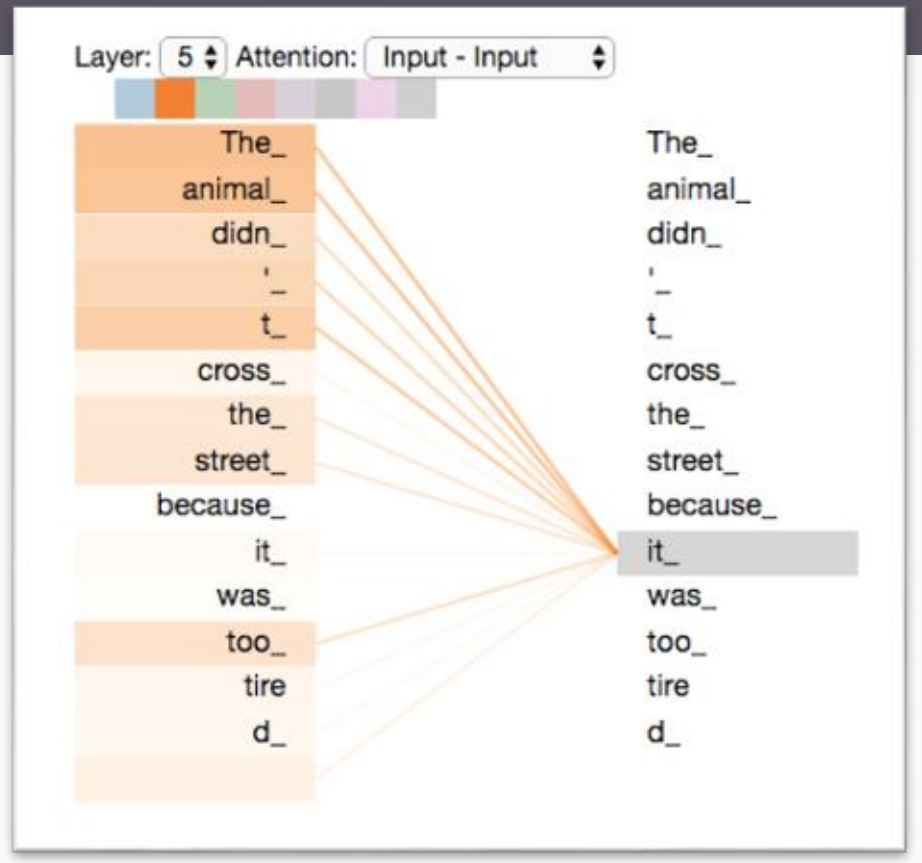
Queries and keys, interpreted

Let's interpret this expression:

$$[\alpha_{j1} \quad \dots \quad \alpha_{jN_w}] = \vec{q}W_q(XW_k)^\top$$

Each α_{ji} is a number representing how relevant word w_i is for understanding word w_j.

Bigger numbers mean highly relevant context (pay attention to this word!) and smaller numbers are not relevant.



[Actual attention in a real trained transformer, courtesy of Jay Alammar]

Softmax

The vector $[\alpha_{j1} \quad \cdots \quad \alpha_{jN_w}]$ is real-valued. Each number is a decimal.

The softmax function transforms real-valued vectors into probability vectors.

```
>>> x
array([-8.96,  8.84, -2.15,  4.1 ])
>>> # do softmax
>>> y = np.exp(-x); y = y / np.sum(y)
>>> y
array([9.99e-01, 1.85e-08, 1.10e-03, 2.13e-06])
>>> sum(y)
1.0
```

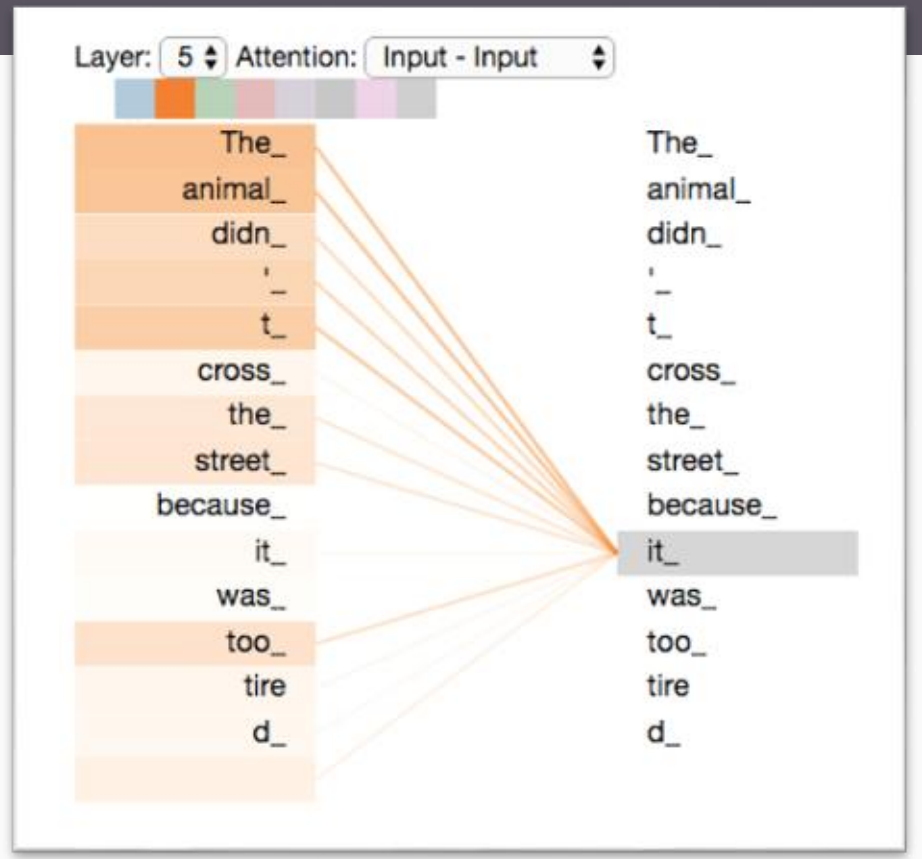
real-valued vector

all positive numbers

that sum to 1

Softmax

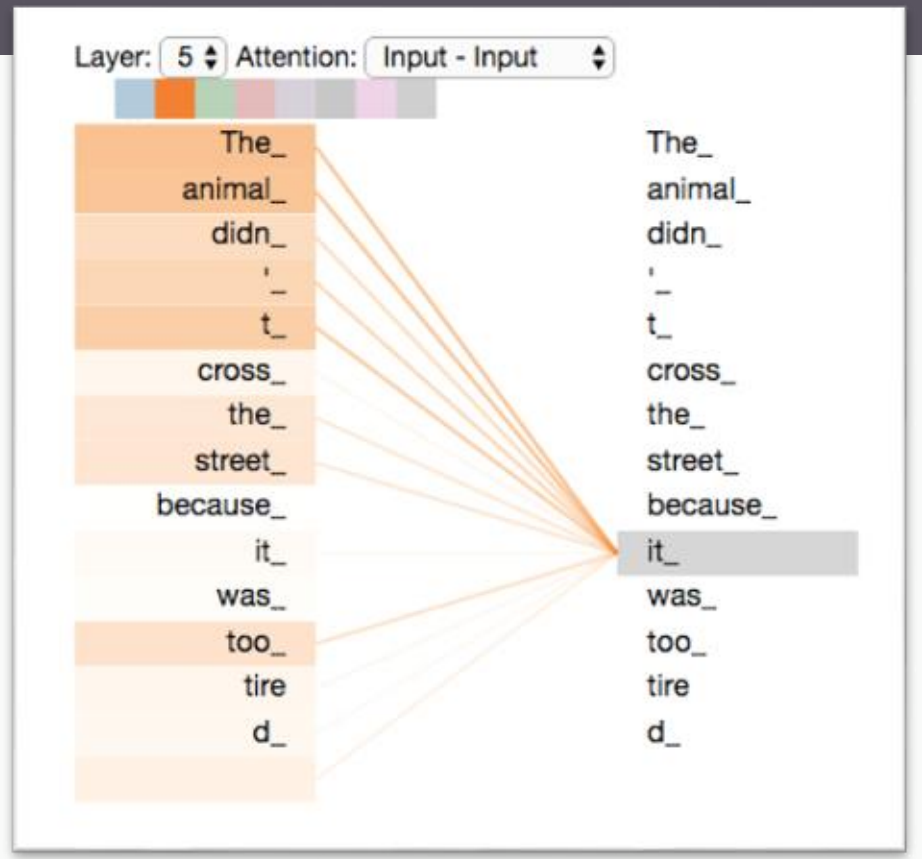
softmax($[\alpha_{j1} \quad \cdots \quad \alpha_{jN_w}]$) splits word w_j's attention over all of the input words, and w_j's total attention adds up to 100%.



Decide on the meaning of a word

Our model is going to interpret words like this:

1. Assign each word a provisional “on-its-own meaning”
2. Reassign final meanings based on the attention calculation from before.



Decide on the meaning of a word

1. Assign each word a provisional "on-its-own meaning"

   Our lingo for a word meaning will be **value**. The value $\vec{v}j$ of word w_j is

   $$\vec{v}_j = \vec{x}_j W_v$$

   Where W_v is a $N_w \times N_o$ matrix of learnable parameters.

2. Reassign final meanings based on the attention calculation from before:

   $$\vec{z}_j = \sum \alpha_{ji} \vec{v}_i$$

   $\vec{z}_j$ is the output. It's our "meaning" of word w_j.

More vectorizing:

Vectorize $\vec{v}_j = \vec{x}_j W_v$: We get $V = XW_v$.

Vectorize $[\alpha_{j1} \quad \cdots \quad \alpha_{jN_w}] = \vec{q} W_q (XW_k)^\top$. We get

$$\begin{bmatrix} \alpha_{11} & \cdots & \alpha_{1N_w} \\ \vdots & \ddots & \vdots \\ \alpha_{N_w 1} & \cdots & \alpha_{N_w N_w} \end{bmatrix} = \left(XW_q\right)(XW_k)^\top$$

The final form of the attention equations:

With input $w_1, \ldots, w_{N_w}$, embed the words to get row vectors $x_1, \ldots, x_{N_w}$. Stack these into $N_w \times N_e$ matrix X.

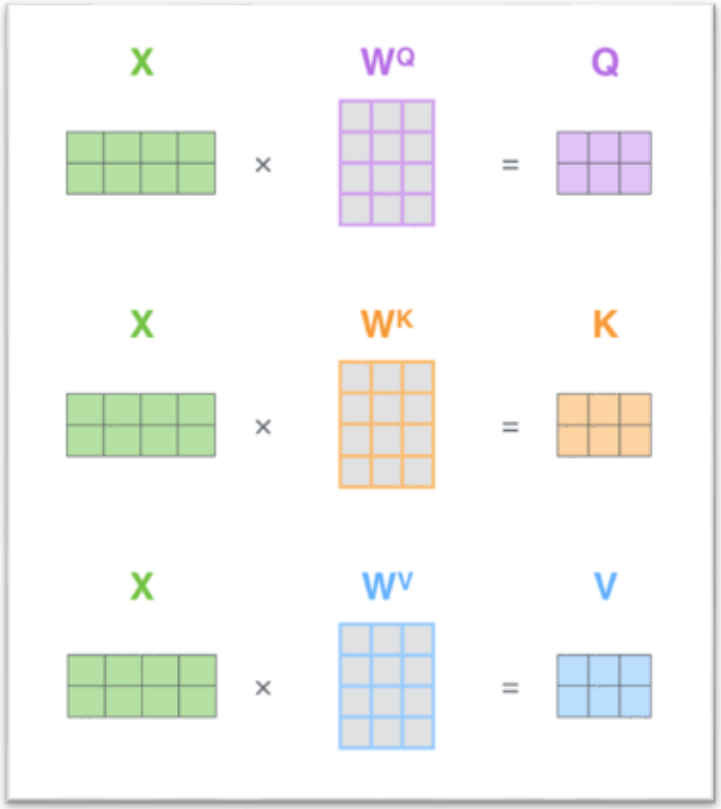
Write $Q = XW_q$, $K = XW_k$, and $V = XW_v$.

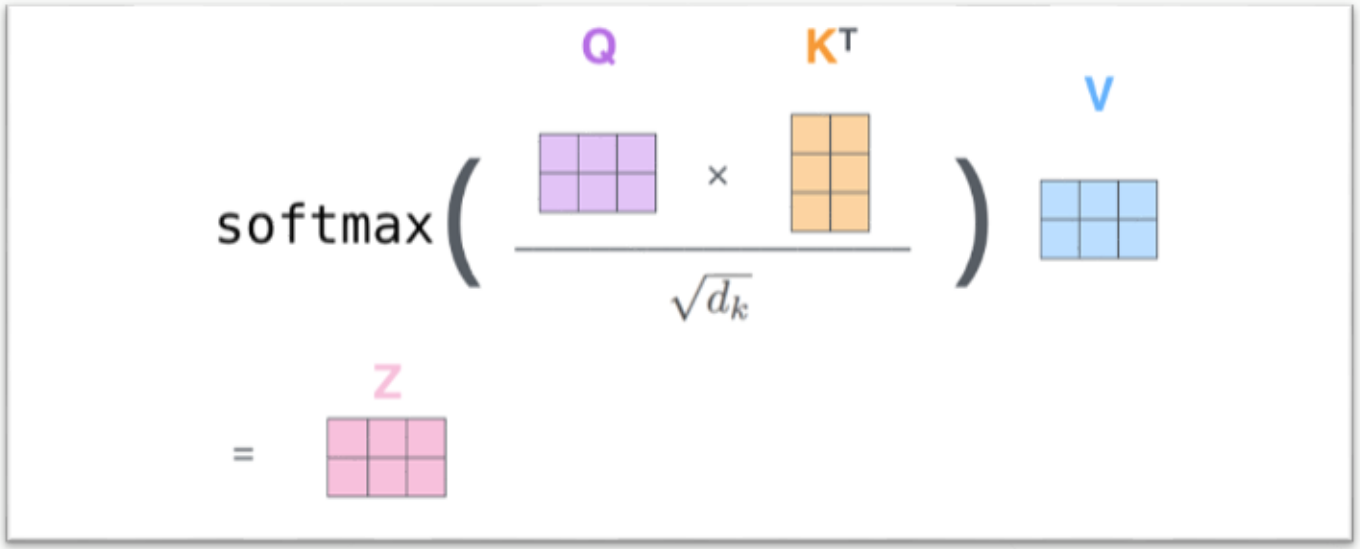
Return $Z = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_s}}\right)V$

I snuck this $\sqrt{d_s}$ in. It's not conceptually important, but does give more stable training (for stats reasons). d_s is the dimension of our similarity dot product: W_q and W_k are $N_e \times N_s$ matrices.

There's a minor notational ambiguity here. Understand this to mean that softmax is being applied row-by-row to this matrix. So each query's attention adds up to 100%

Attention equations, visualized:





[figures courtesy of Jay Alammar]

TRANSFORMERS

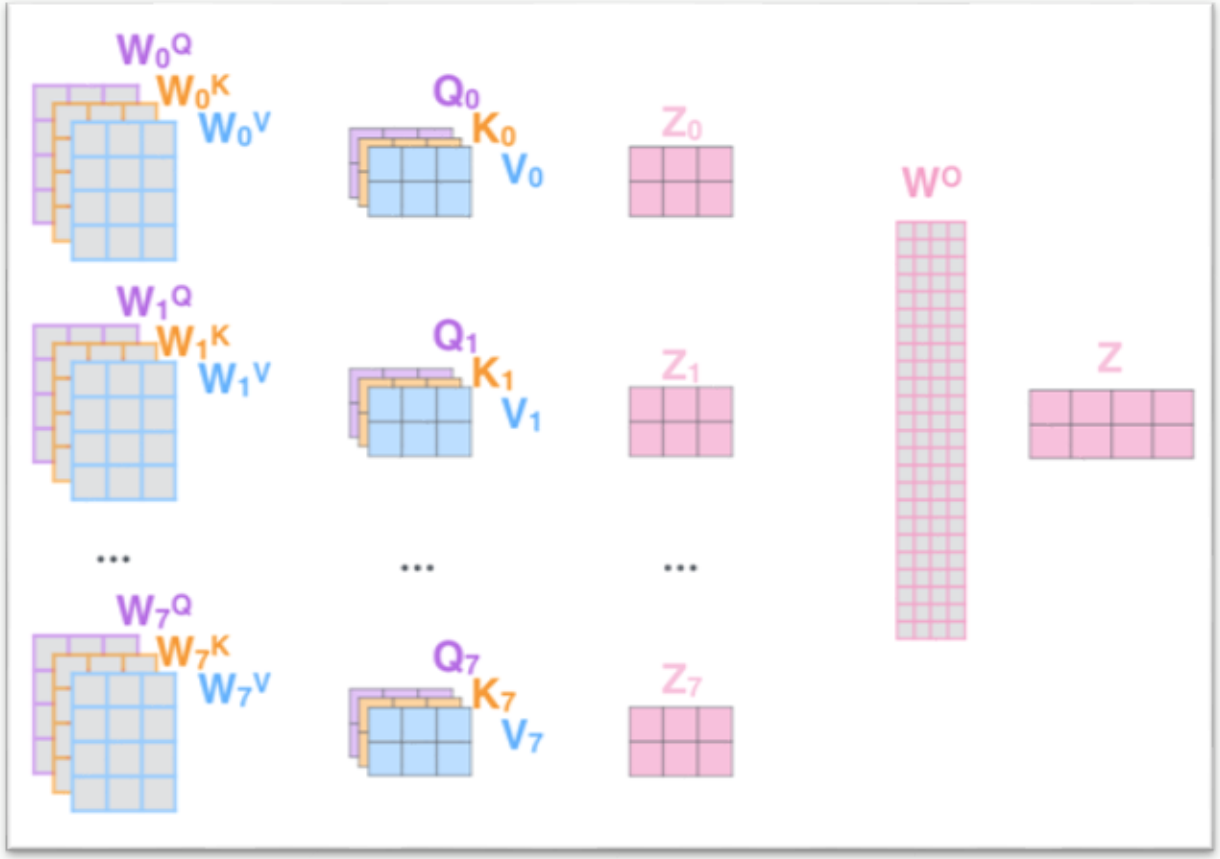
Small tweak, big performance boost:

Multi-headed Attention:

Do the attention equations multiple times in parallel.

Learn different weights for each “attention head”.

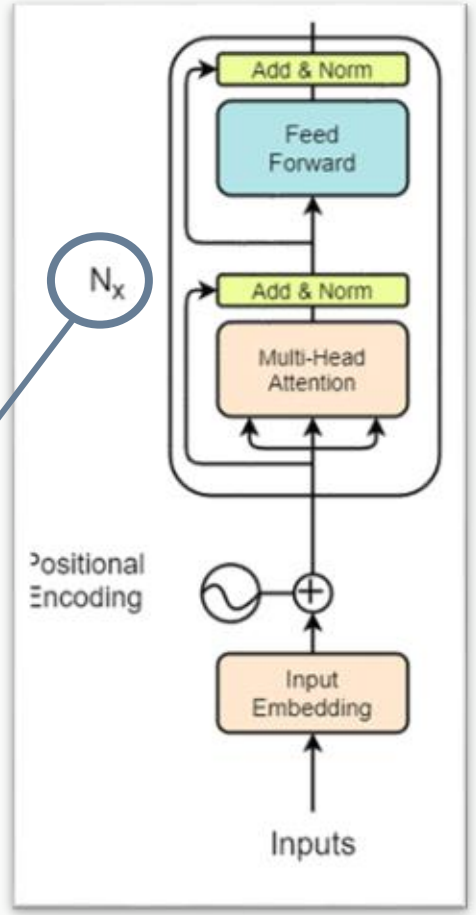
Merge the outputs from each attention head to make the combined output.



Stack attention layers, with a little nonlinearity

The feed-forward block is a simple stack of two of three linear neural network layers, with a standard nonlinearity (like ReLU) after each one.

repetition = stacked layers



I'm skipping the "Positional Encoding" today because research is moving fast in that area and it hasn't settled down yet.

[figure from the original paper introducing transformers: "Attention is all you need, Polosukhin et al.]

Does it work?

Transformer models model long-range relationships in text effortlessly.

Transformer models require mind-boggling amounts of time and data to train.

Famous Transformer models:

- ChatGPT (**g**enerative **p**re-trained **t**ransformer)
  - *Best estimate: first training run cost $12M in compute*
  - *Trained on ~45TB of text*
- Google Gemini
- Claude

VISION TRANSFORMERS

How do we adapt transformers to vision?

Transformers were invented as sequence models. Images aren't sequences like sentences are.

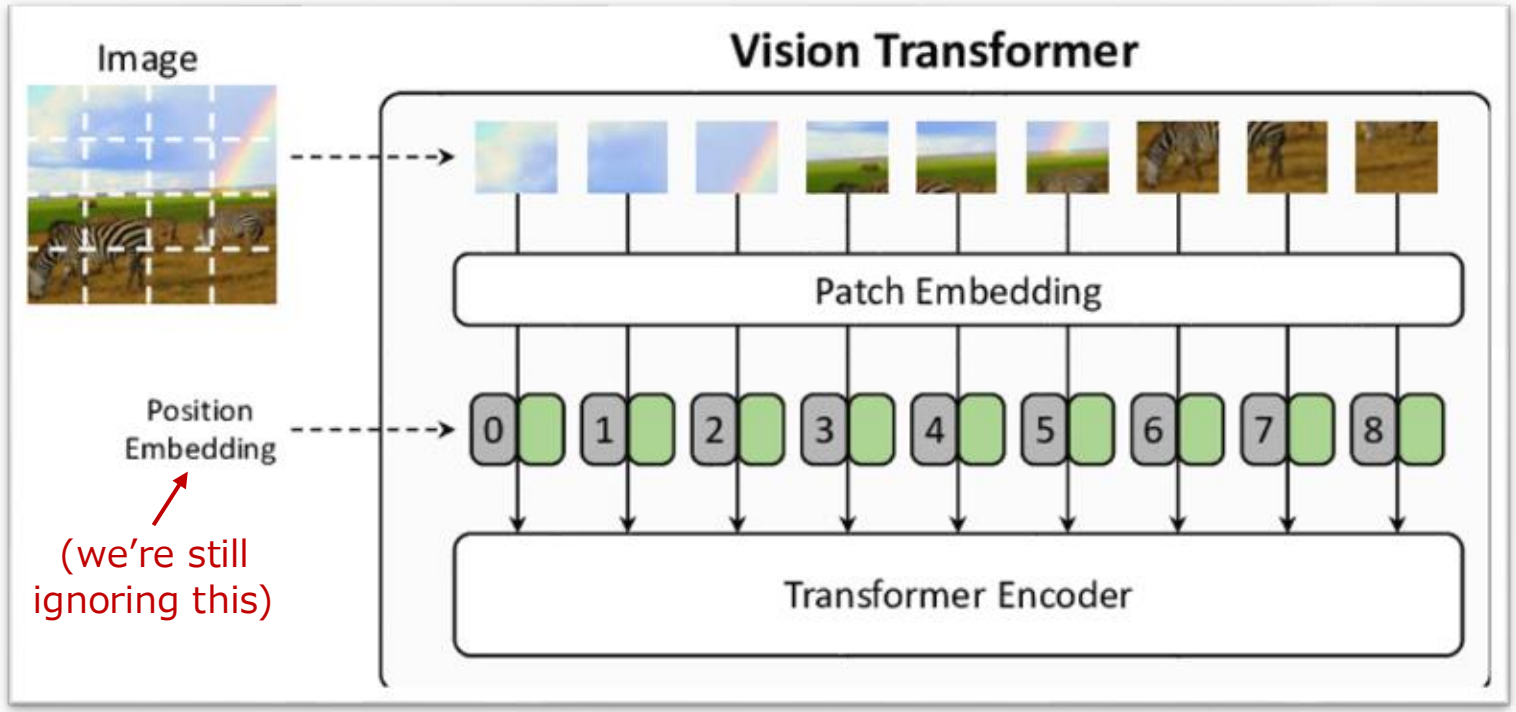
Newer way of thinking:

- A transformer is a model that takes a set of tokens and encodes their meaning as a latent space vector.

What serves the role of tokens in an image? Why not **patches**?

Image patches are our words

1. Chop the image into patches
2. Embed each patch (just like we did with words)
3. Apply the same transformer layers we used for words



[figure source]

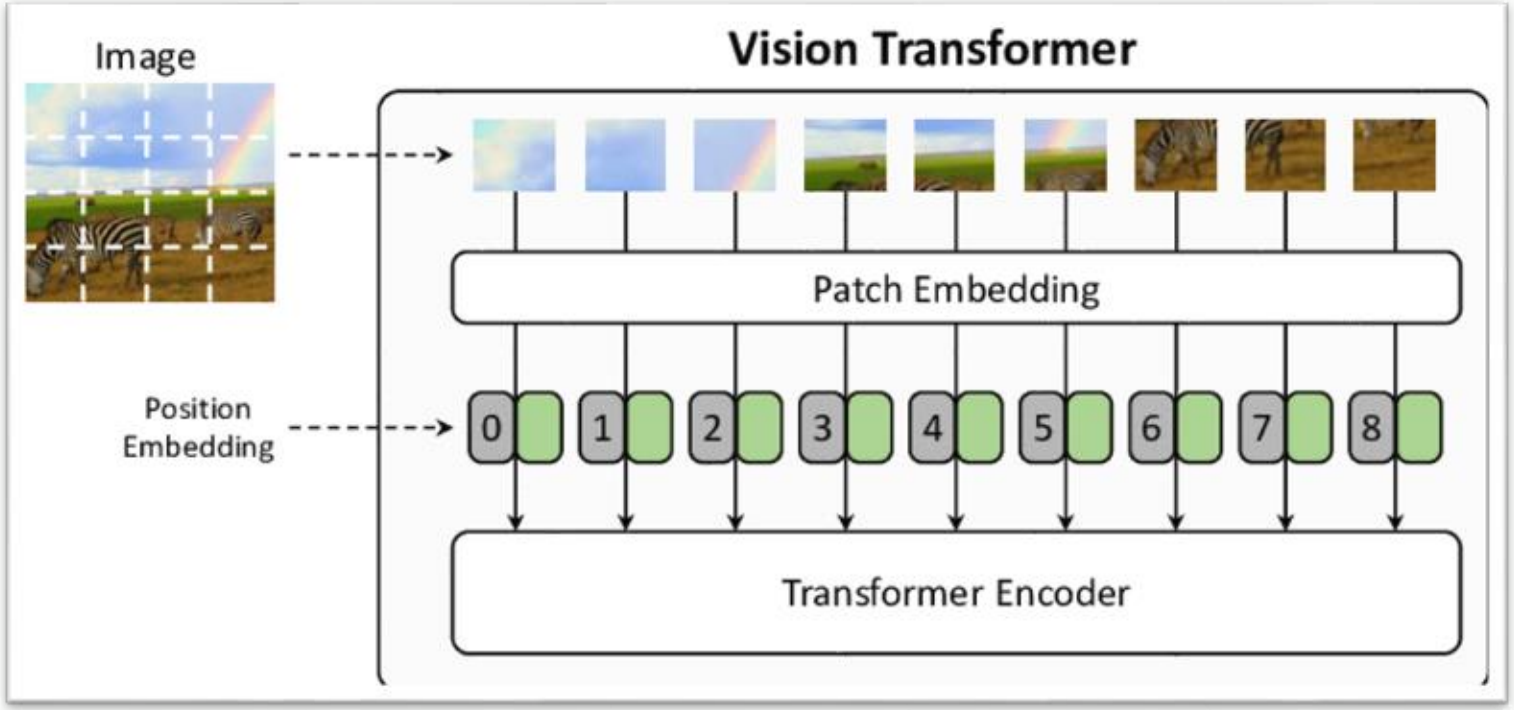
small lie! A few other things change too. But the differences are practical, not conceptual.

What does it mean?

Our model is computing a vector representing the “meaning” of each patch.

Each patch’s meaning is contextually dependent.

The Transformer learns which other patches matter for understanding this one.



[figure source]

Does it work?

Vision transformers are the encoders for lots of models you might have heard of:

- DALL-E 2 and DALL-E 3 (OpenAI)
- GPT4V (GPT4 with Vision, OpenAI)
- Segment Anything (Meta AI)
- Imagen (Google)
- Firefly (Adobe)

How do you train one?

We know training a Vision Transformer will be expensive. But we plan to reuse it as an encoder in lots of other tasks.

What tasks do we train the encoder on?

- Image classification (lots of training data)
- Masked Image Modeling (fill in holes in images. Infinite free training data!)
- Contrastive Learning (this is more advanced. Train to make similar images encode to similar vectors, and dissimilar images encode to dissimilar vectors)