Parallelism in Machine Learning Applications and Online Services

UCSB CS140, Winter 2026

Tao Yang

Outline

- Matrix-based compute-intensive neural image/audio/text processing
  - Text processing
    - Word embeddings
    - Transformer architecture for language models
    - BERT, GPT
  - Convolutional neural network for image object detection
- High throughput/ parallel online service
  - Web search engine

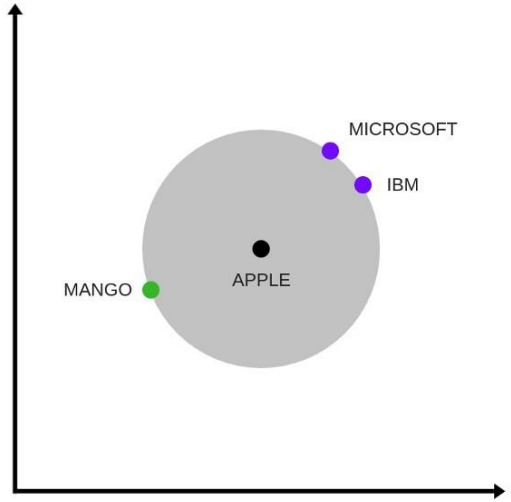
Neural Computation for Text Processing

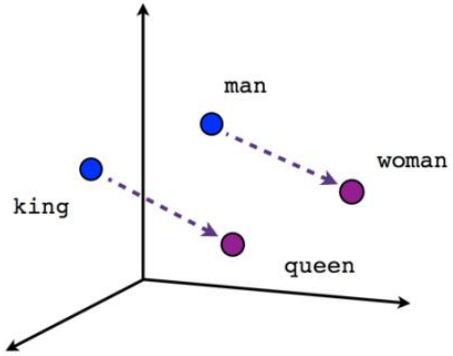
Background: Word representation with embedding for a document

Traditional Method - Bag of Words Model	Word Embeddings
• Uses one hot encoding • Each word in the vocabulary is represented by one bit position in a HUGE vector. • For example, if we have a vocabulary of 10000 words, and "Hello" is the 4th word in the dictionary, it would be represented by: 0 0 0 1 0 0 . . . . . 0 0 0 0	• Stores each word in as a point in space, where it is represented by a vector of fixed number of dimensions (e.g. 300) • For example, "Hello" might be represented as : [0.4, -0.11, 0.55, 0.3 . . . 0.1, 0.02] Word2vec from Google in 2013

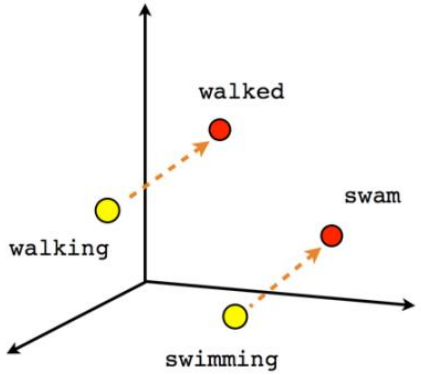
Examples on Characteristics of Word Embeddings

Numerical representations of similarities between words





Male-Female

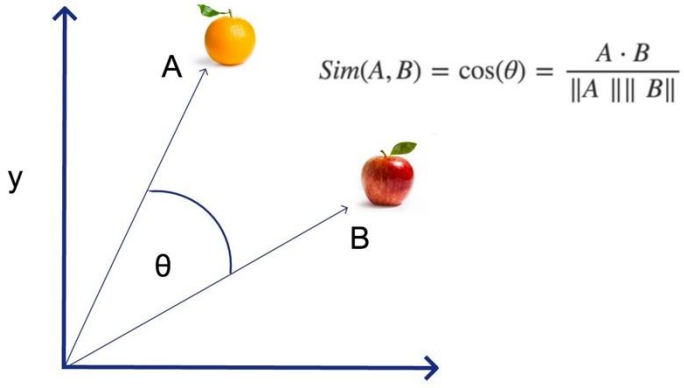


Verb tense

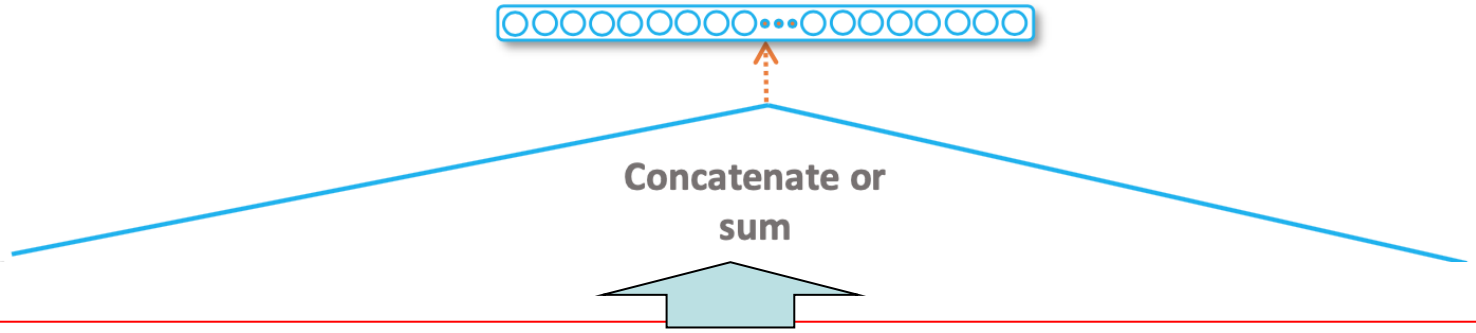
Cosine Similarity

- Model semantic similarity between two words:

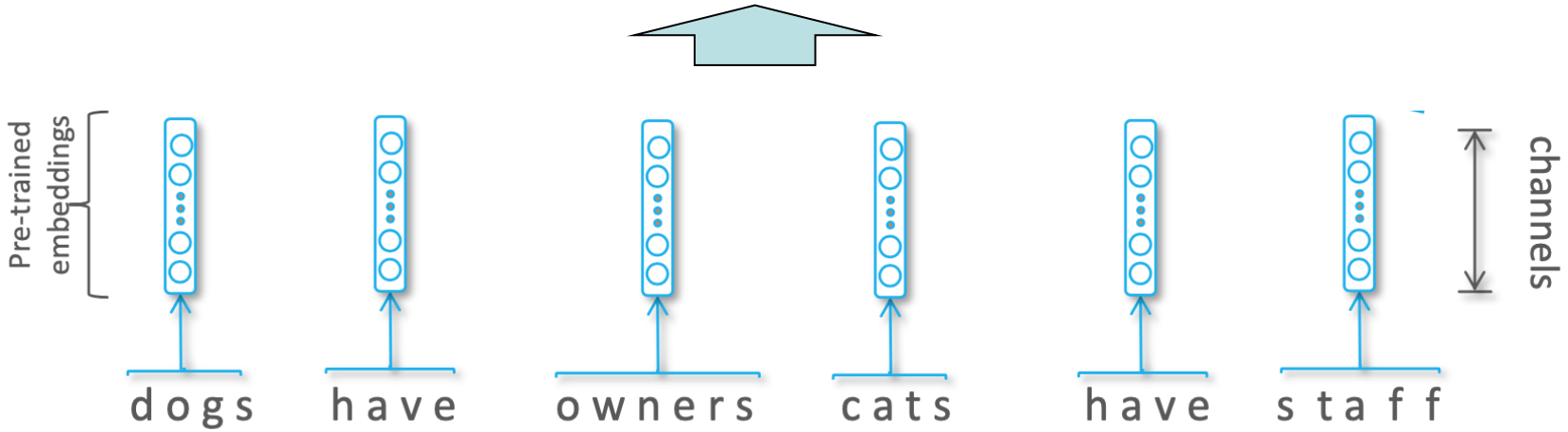
Cosine similarity('woman','man')= 0.73723527



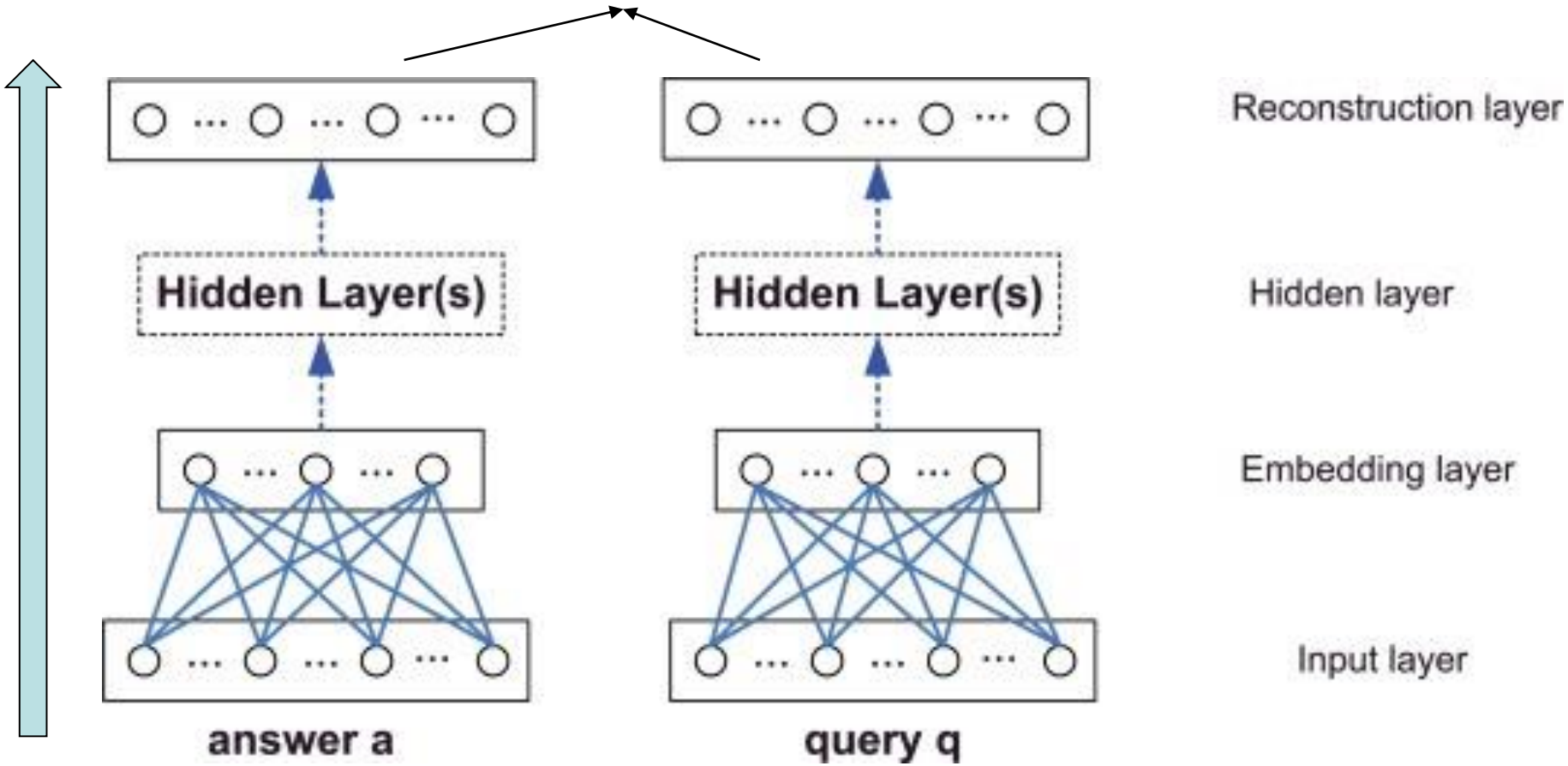
Represent a sentence with a neural matrix or vector



Additional transformation such as CNN or transformer.
Think about it as matrix-matrix multiplication



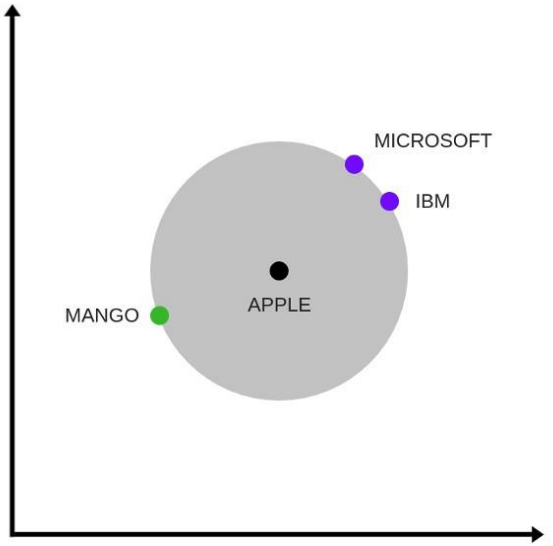
Match a query with a document answer



Static Word Embeddings vs. Contextual Representations

Word embeddings are the basis of deep learning for text understanding and NLP

Word embeddings (e.g. word2vec) are often *pre-trained* on text corpus from co-occurrence statistics



Problem: Word embeddings are applied in a context free manner

The taste of this apple is good → `[0.9, -.2, .6, ...]`

This apple phone looks good

Solution: Train *contextual* representations on text corpus

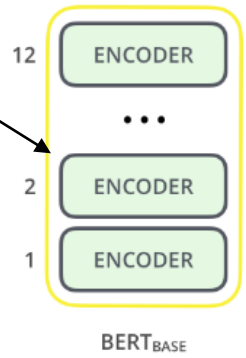
`[0.1, -.2, .6, ...]` `[0.3, .1, .3, ...]`

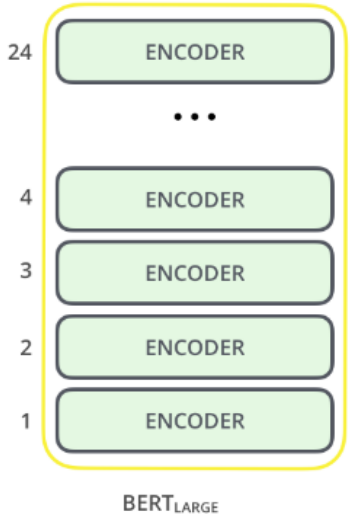
BERT: Bidirectional Encoder Representations for Transformers

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, Jacob Devlin Ming-Wei Chang Kenton Lee Kristina Toutanova Google, Proc of NAACL 2019

- BERT architecture is a multi-layer bidirectional Transformer encoder.
- 12 layers (transformer blocks) for BERT-base. 110 million parameters.
- Token embedding dimension: 768
- Unsupervised trained with books, Wikipedia :
- Masked language model: Predicted a masked word in a sentence

  "my dog is ~~cute~~"

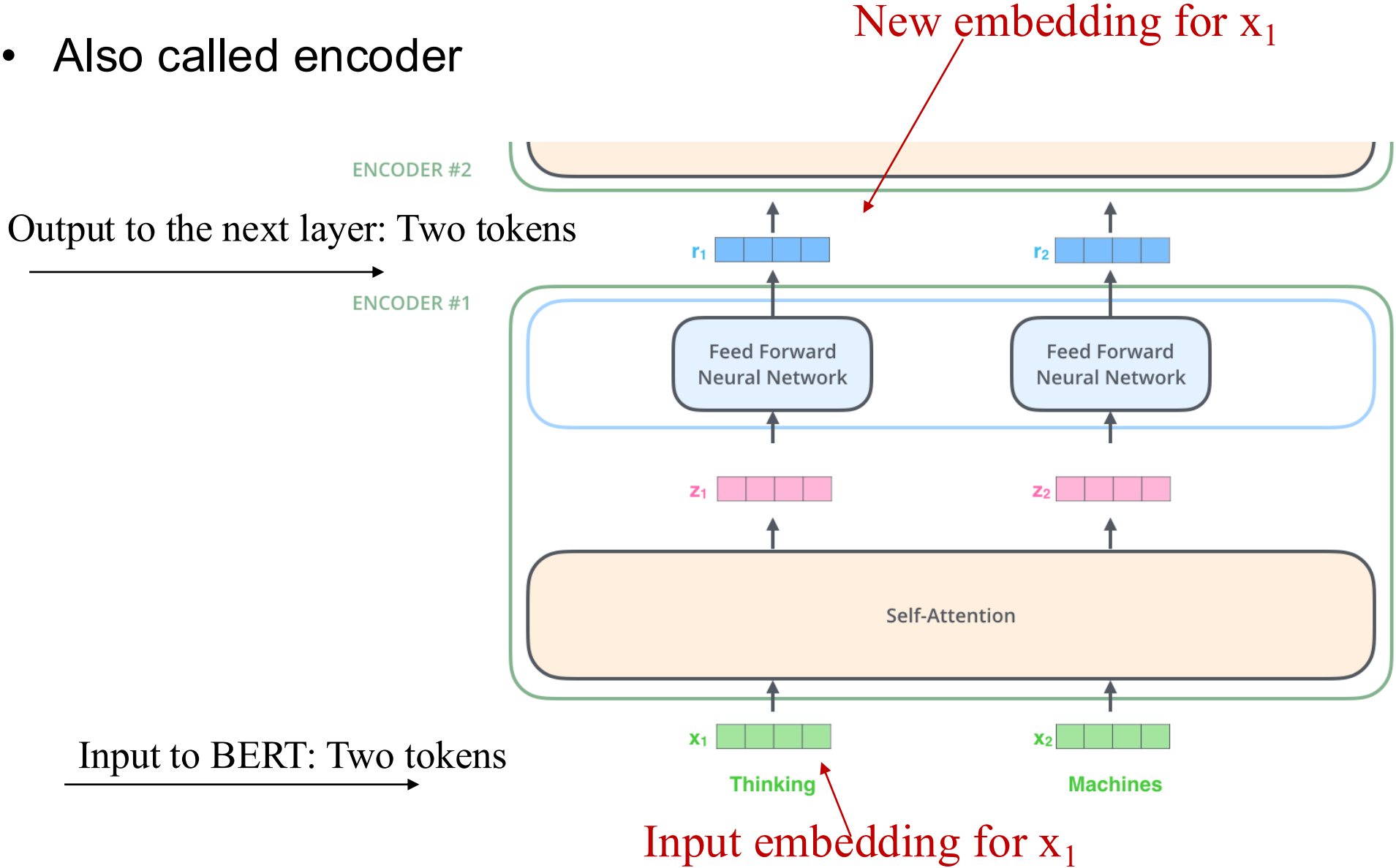




The transformer layers of BERT

https://jalammar.github.io/illustrated-transformer/

- Also called encoder



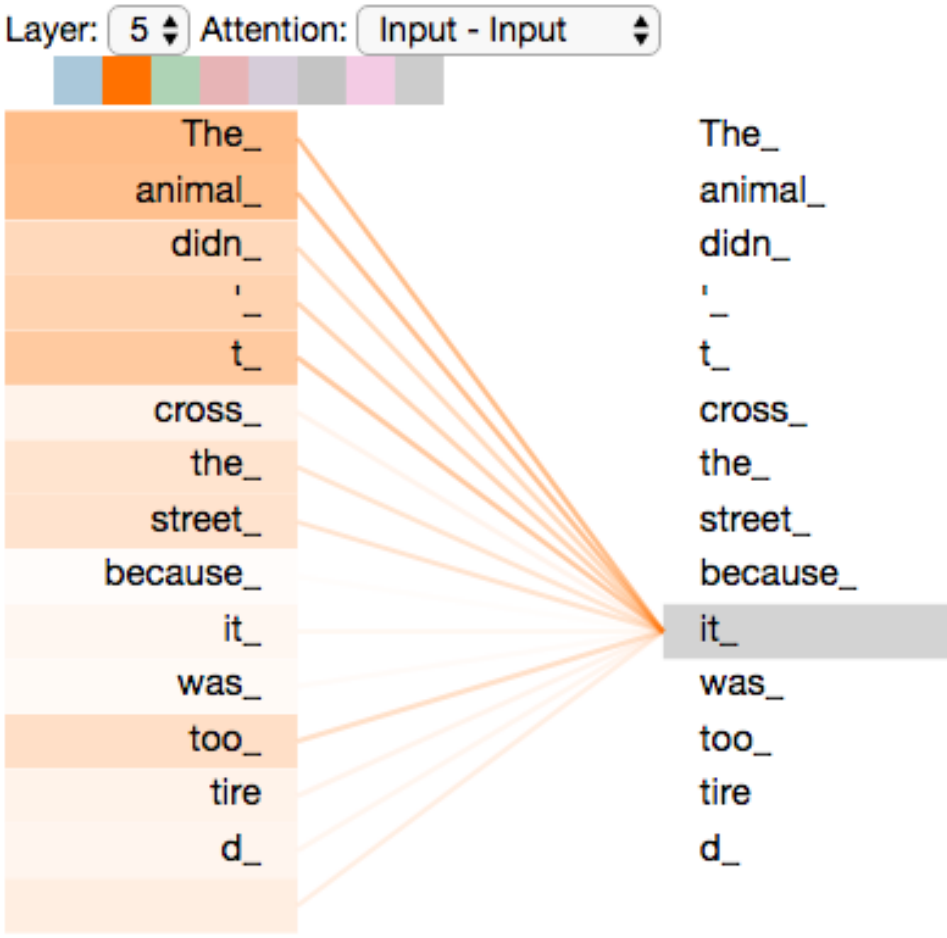
More about Attention

"The Attention is All You Need" paper, A. Vaswani et al. NIPS 2017

Self-attention is the method the Transformer uses to detect how other words semantically, syntactically, or contextually relevant to the current word

Example: "The animal didn't cross the street because it was too tired"

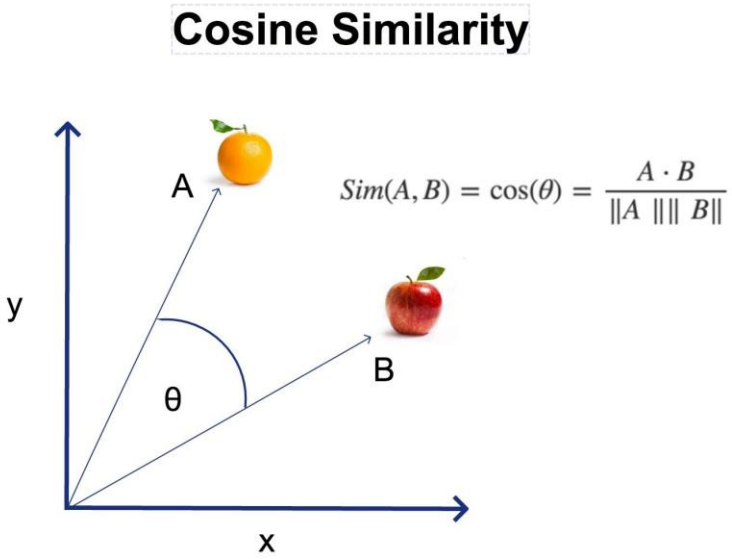
What does "it" in this sentence refer to? self-attention allows it to associate "it" with "animal".



What can be used to model semantic relevance of two word tokens?

- Every word token has an embedding vector
- Dot product of vectors A and B : their similarity or distance

- Related to cosine similarity, especially ||A||=||B||=1



- **Attention among a sequence of word tokens:**
  - $X X^T$: Similarity among row vectors of matrix X

$$X X^T = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} (x_1, x_2) = \begin{bmatrix} x_1 x_1^T & x_1 x_2^T \\ x_2 x_1^T & x_2 x_2^T \end{bmatrix}$$

Meaning of Matrix Multiplication

- ***X W***

Map row vectors of X to a space modeled by column vectors of W

$$\textit{Example:} \quad XW = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} (w_1, w_2) = \begin{bmatrix} x_1 {w_1}^T & x_1 {w_2}^T \\ x_2 {w_1}^T & x_2 {w_2}^T \end{bmatrix}$$

x_1 =(Latitude, longitude) of Seattle

w_1 =(Latitude, longitude) of Los Angeles

w_2 =(Latitude, longitude) of New York City

$Represent\ x_1\ with\ [x_1 {w_1}^T, x_1 {w_2}^T]$

→Represent Seattle as [distance to LA, distance to NYC]

→Matrix multiplication XW still represents X but in a different form

Design Options for Self Attention Computation

Assume 2 input token vectors x_1, x_2, forming matrix $X = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$
Compute the self attention of input tokens as:

- **Option 1**: Compute inner product similarity of tokens

$$X\,X^T = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} (x_1, x_2) = \begin{bmatrix} x_1 x_1^T & x_1 x_2^T \\ x_2 x_1^T & x_2 x_2^T \end{bmatrix}$$

- **Option 2** by BERT with learned parameters W^Q, W^Q, W^V, and W^O:
  1. Map X into a space called query: $Q = X\,W^Q$
  2. Map X into another space called key: $K = X\,W^Q$
  3. Map X into one more space called value : $V = X\,W^V$
  4. Compute inner product similarity of Q and K as: QK^T.
     - Normalize with softmax as attention weights of token pairs
     - Reformulate as weighted linear combination of V $Z = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$
  5. Reorganize Z linearly as outputs with single or multi heads

Softmax Normalization

Turn a vector of K real values into a vector of K real values that sum to 1.

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Example: z

$$\begin{bmatrix} 8 \\ 5 \\ 0 \end{bmatrix}$$

$$e^{z_1} = e^{8} = 2981.0 \qquad \sigma(\vec{z})_1 = \frac{2981.0}{3130.4} = 0.9523$$

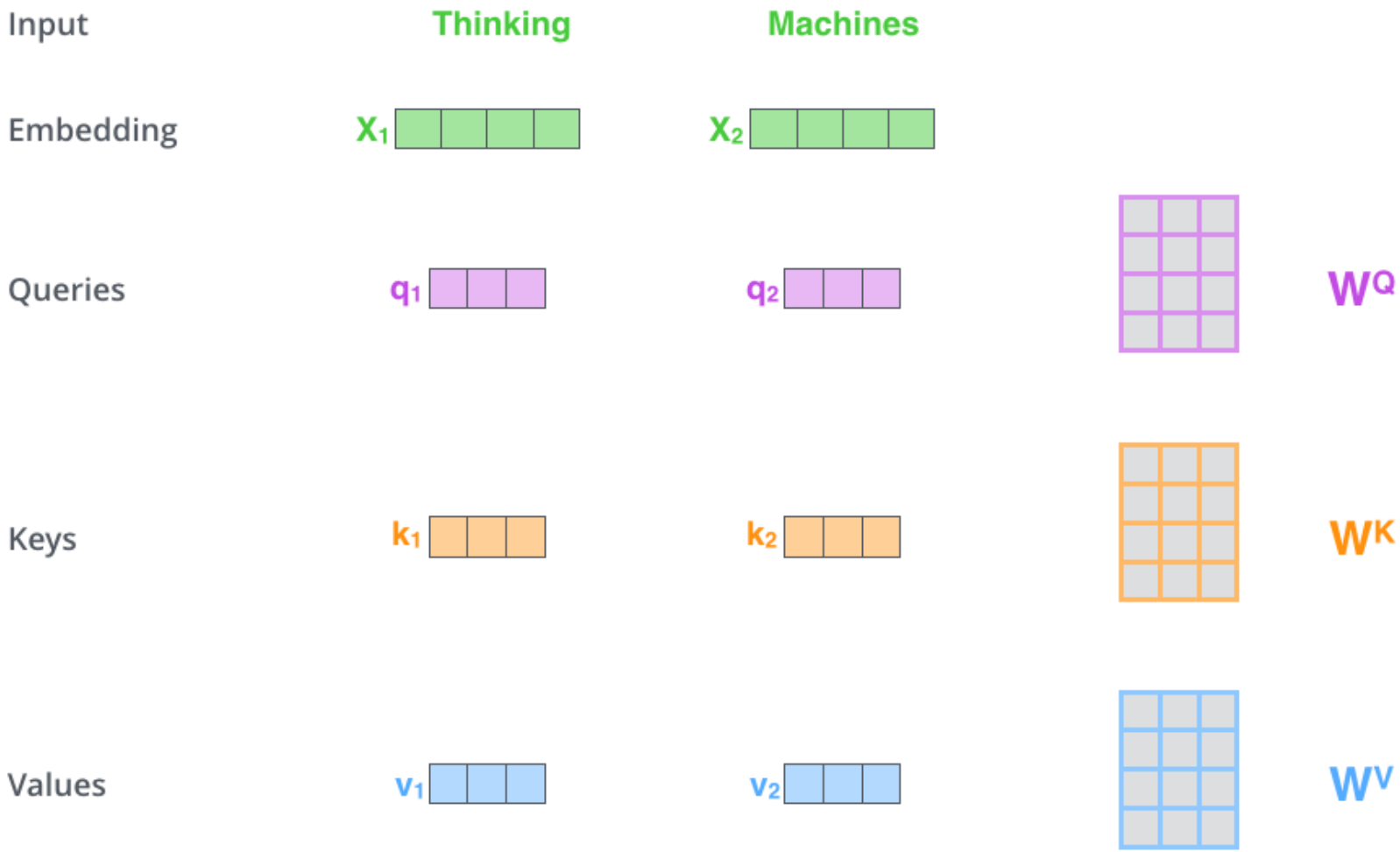
$$e^{z_2} = e^{5} = 148.4 \qquad \sigma(\vec{z})_2 = \frac{148.4}{3130.4} = 0.0474$$

$$e^{z_3} = e^{0} = 1.0 \qquad \sigma(\vec{z})_3 = \frac{1.0}{3130.4} = 0.0003$$

Purpose: Vector values may have small differences. SoftMax scaling keeps values in a range where the model can still learn from differences

Example of Transformer Computation for Self Attention

For each input token, we create a Query vector, a Key vector, and a Value vector.



Transformer Computation for Self-Attention: Compute Query, Key, and Value Matrices

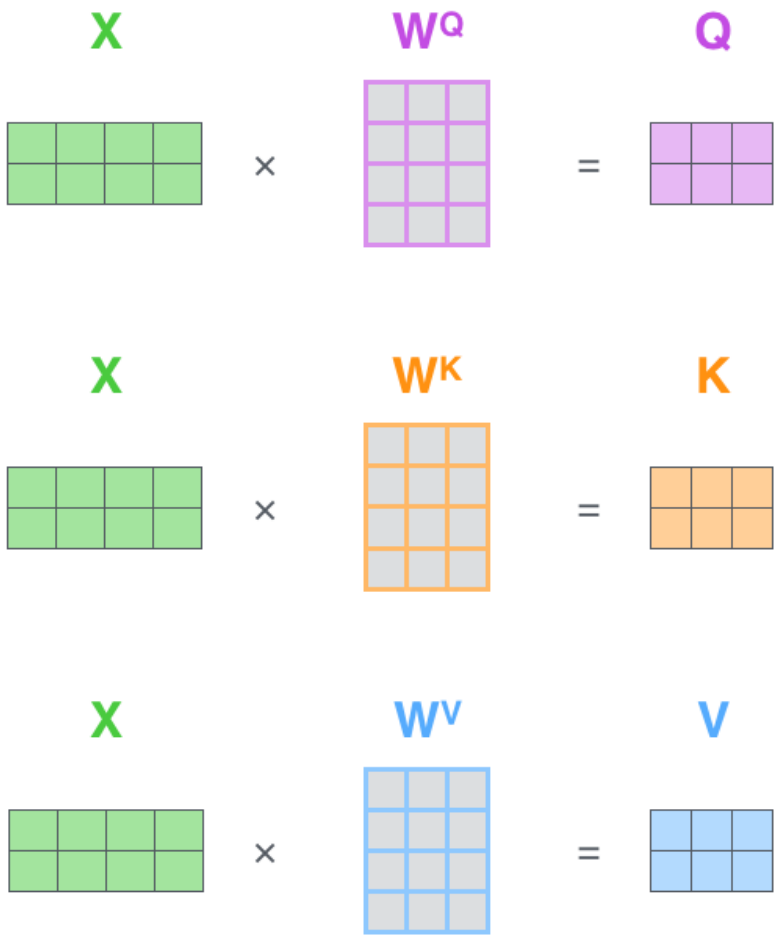
For 2 tokens x_1, x_2 , they form two rows of matrix X.

Multiply with 3 weight matrices, we get query, key, value matrices Q, K, V

Q contains q_1, q_2

K contains k_1, k_2

V contains v_1, v_2



Takeaways from Transformer Computation

- Q, K, and V represent embeddings of input X in a different form

- Attention matrix $A = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)$ represents inter-token pairwise attentions $X X^T$ in a normalized form

$$X X^T = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} (x_1, x_2) = \begin{bmatrix} x_1 x_1^T & x_1 x_2^T \\ x_2 x_1^T & x_2 x_2^T \end{bmatrix}$$

- Output $Z = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) V = AV = (A_1, A_2) \begin{pmatrix} V_1 \\ V_2 \end{pmatrix} = A_1 V_1^T + A_2 V_2^T$

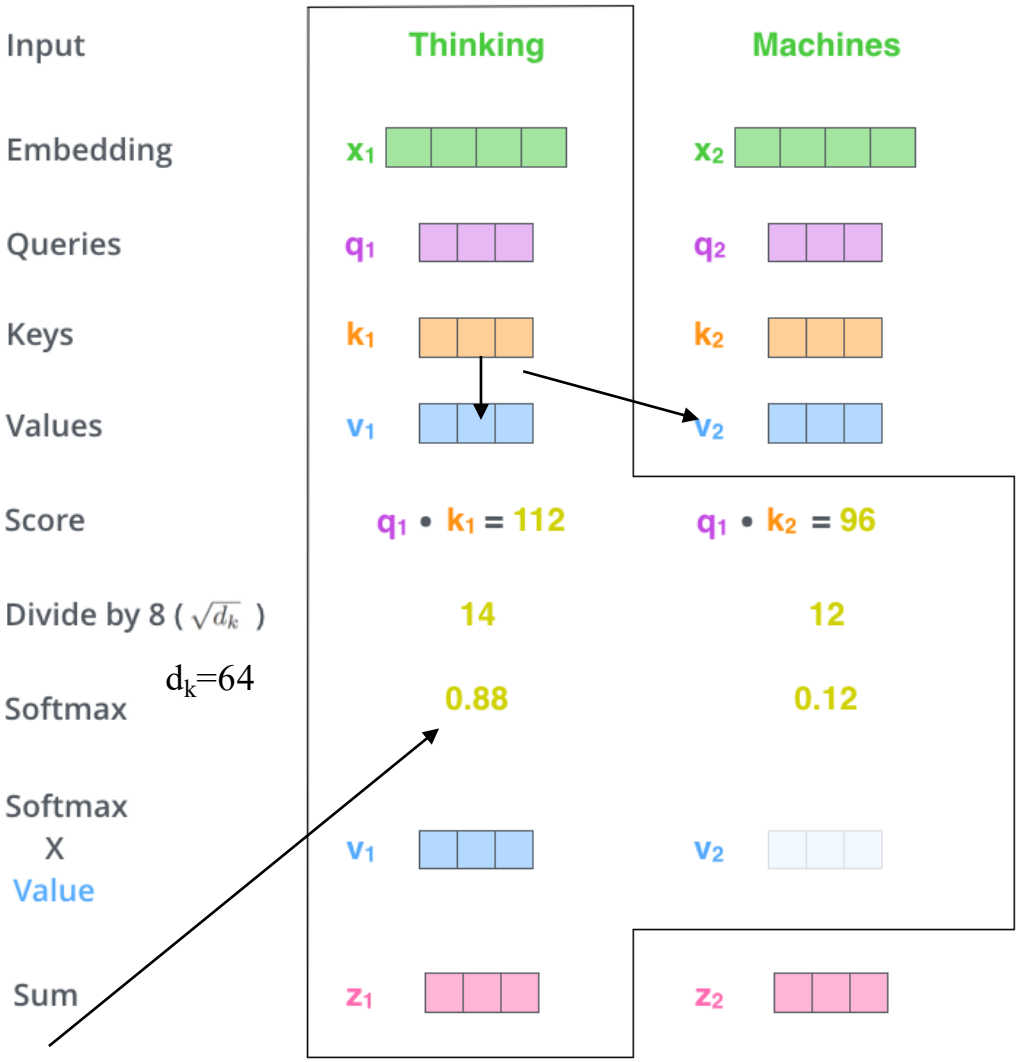
→Attention-weighted linear combination of rows V_1 and V_2 of V

→Attention-weighted linear combinations of input embeddings conceptually

Output Row 1: a new embedding for x_1

$$Z_1 \approx (x_1 x_1^T) V_1 + (x_1 x_2^T) V_2 \approx (x_1 x_1^T) x_1 + (x_1 x_2^T) x_2$$

Example: Output of Self Attention

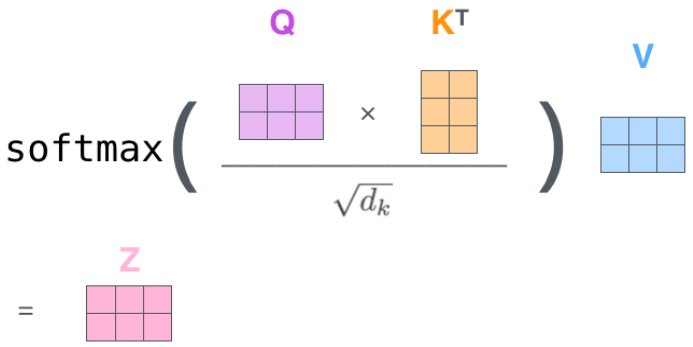


Pairwise attention weights of tokens

For 2 tokens x_1, x_2 of dimension d_k,

output z_1, z_2

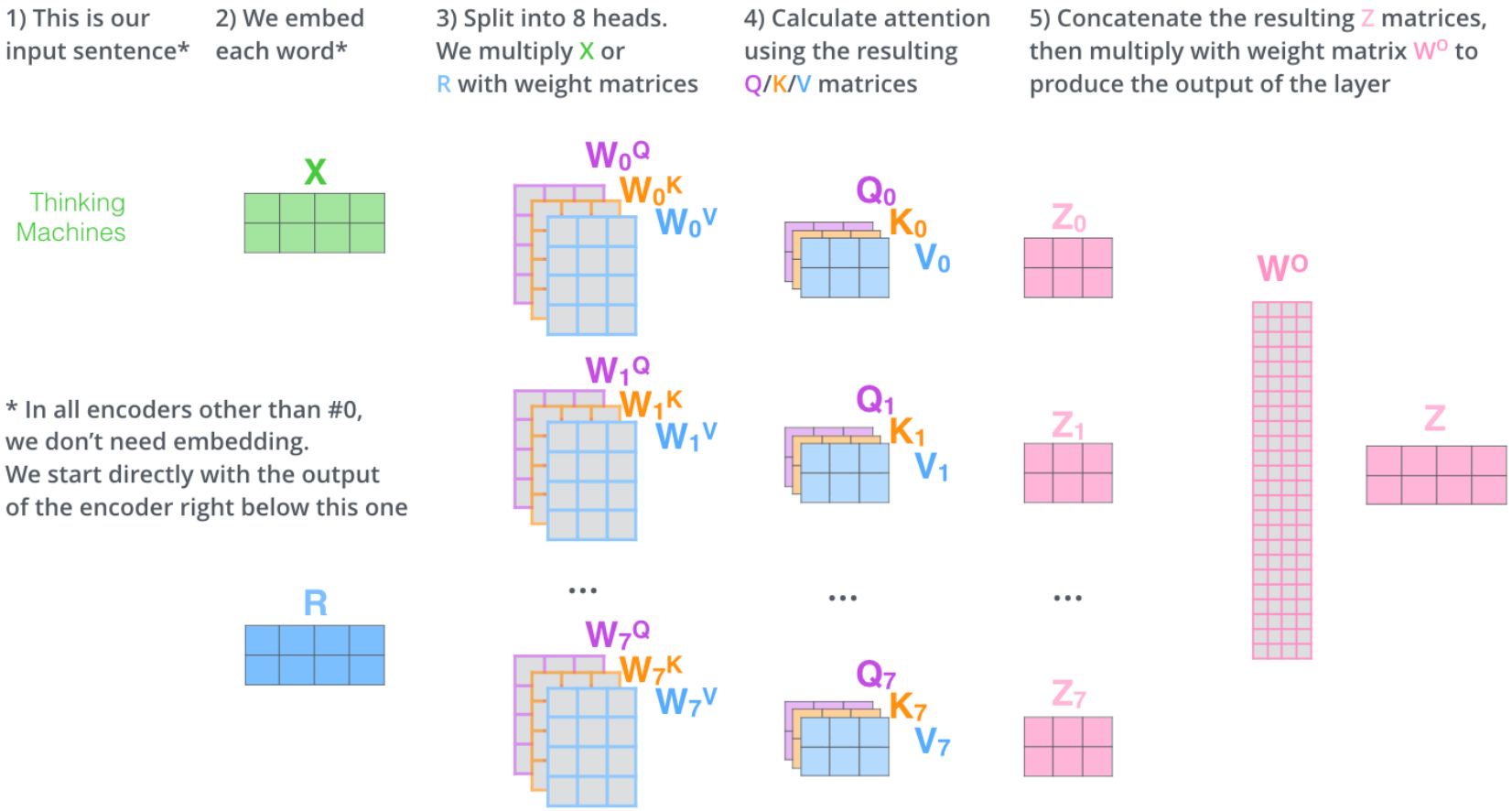
z_1 represents interaction of q_1 with k_1 , and interaction of q_1 with k_2



Multi-headed Attention: Assume 8 Heads

Repeat the same attention computation for different heads

Finally concatenate them together multiplied by weight matrix



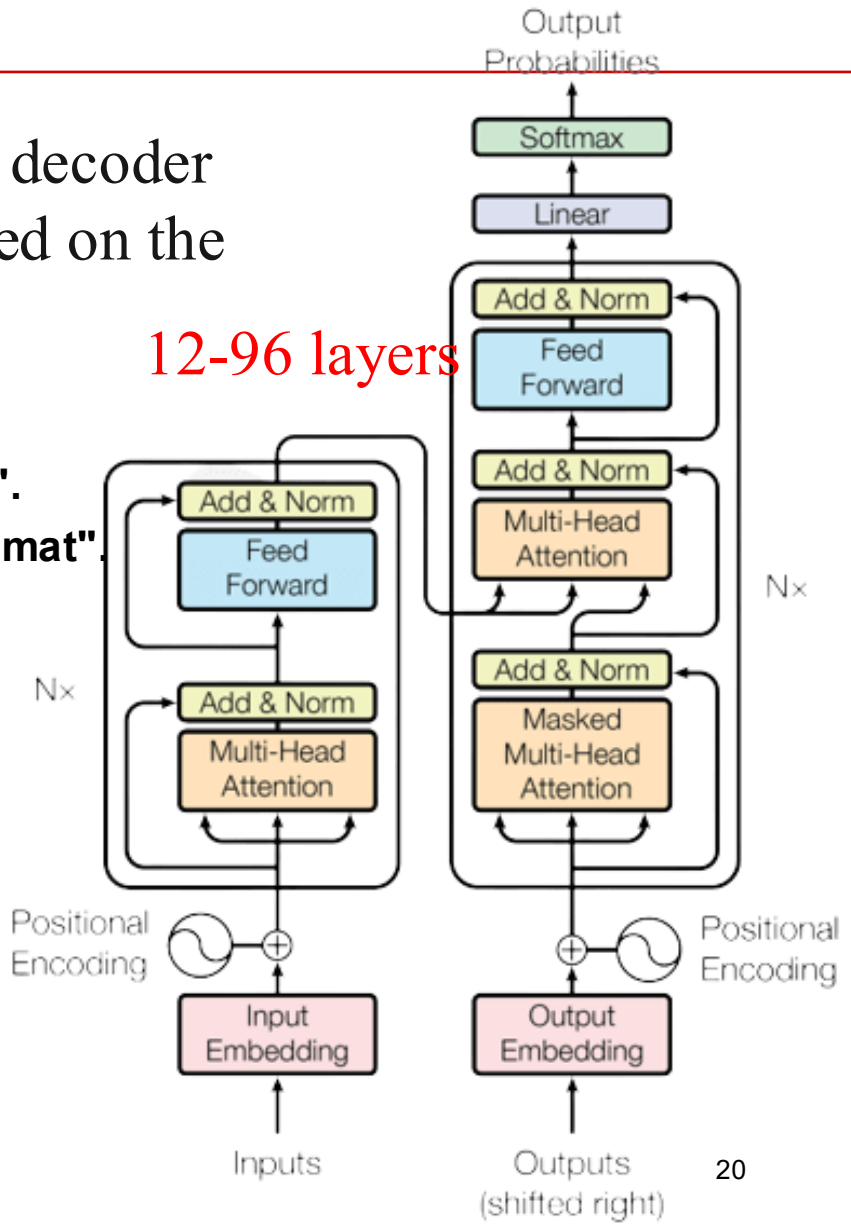
Model training learns parameters in weight matrices

GPT-2 and GPT-3 (1.5 - 175 billion parameters)

- **GPT** is an autoregressive transformer decoder
- Each token is predicted and conditioned on the previous tokens

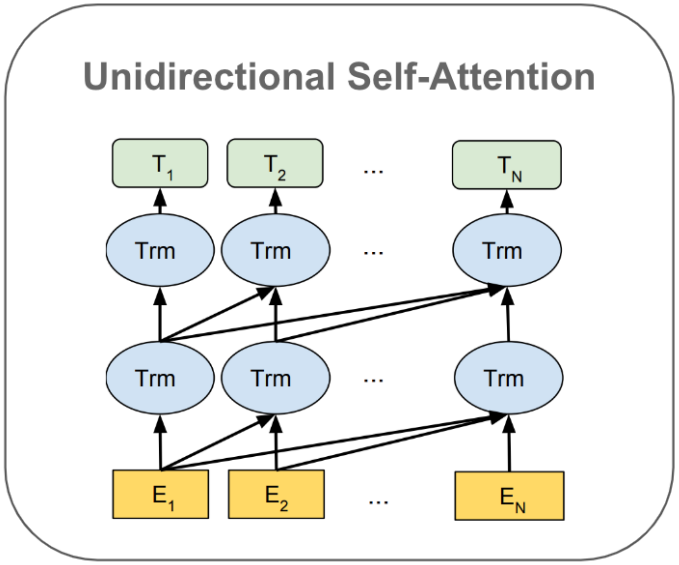
Step 1: Input: "The cat sat" → GPT predicts "on".
Step 2: Input: "The cat sat on" → GPT predicts "the".
Step 3: Input: "The cat sat on the" → GPT predicts "mat".

- Input: 2048 tokens of 50257 word vocabulary.
  - Each token embedding: 12288 dimensions
- Output: 2048 tokens
  - GPT chooses top tokens from the vocabulary with high probabilities
- Trained with common crawl, Wikipedia, GitHub, …



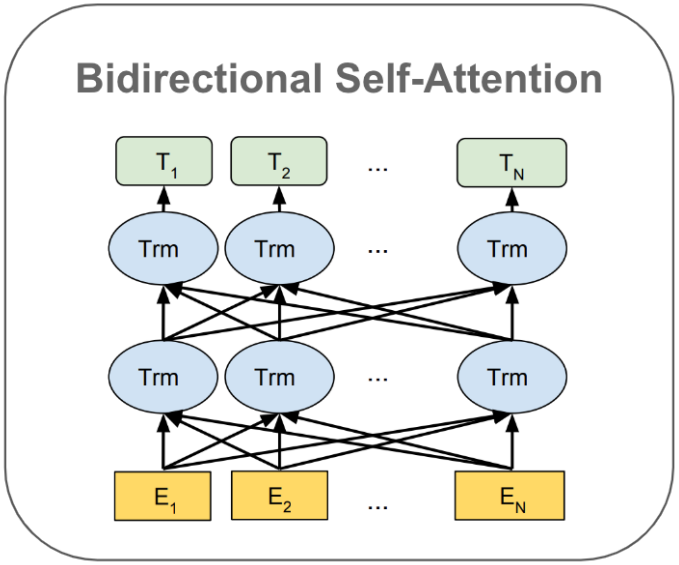
Attention computation in GPT decoding and BERT encoding

Decoder (GPT)



- Only look at the past.
- Good at generating text.

Encoder (BERT)



- Look at words before and after the current position simultaneously
- Good at understanding context

- Computing time during inference, and training (forward/backward propagation) of LLMs is dominated by matrix multiplication involving key, value, and query matrices

High Performance Computing for LLMs with GPUs

Model	Training end	Chip type	TFLOP/s (max)	Chip count	Wall clock (days)	Total time (years)	Retail (US$)	MMLU
GPT-3 175B	Apr/2020	V100	130	10,000	15 days	405y	$9M	43.9
Llama 1 65B	Jan/2023	A100	312	2,048	21 days	118y	$4M	63.4
Llama 2 70B	Jun/2023	A100	312	*2,048*	35 days	196y	$7M	68.0
Titan 200B	Apr/2023	A100	312	13,760	48 days	1,319y	$45M	70.4
GPT-4 1.7T	Aug/2022	A100	312	25,000	95 days	6,507y	$224M	86.4
Gemini	Nov/2023	TPUv4	275	57,000	100 days	15,616y	$440M	90.0
Llama 3 405B	Apr/2024	H100	989	24,576	50 days	3,366y	$125M	*85+*
GPT-5	*Apr/2024*	H100	989	*50,000*	*120 days*	*16,438y*	$612M	
Grok 2	*Jun/2024*	H100	989	20,000	*50 days*	*6,571y*	$245M	
Olympus	*Aug/2024*	H100	989					
Gemini 2	*Nov/2024*	TPUv6	*1,847*					
Grok 3	*Dec/2024*	H100	989	100,000	*50 days*	*32,855y*	$1.2B	
						Alan D. Thompson. May/2024. LifeArchitect.ai		

Table. Model training compute (see working, with sources[8]).

NVIDIA V100 Peak:
Single-Precision (FP32): 14 - 15.7 TFLOPS
Double-Precision (FP64): 7 - 7.8 TFLOPS

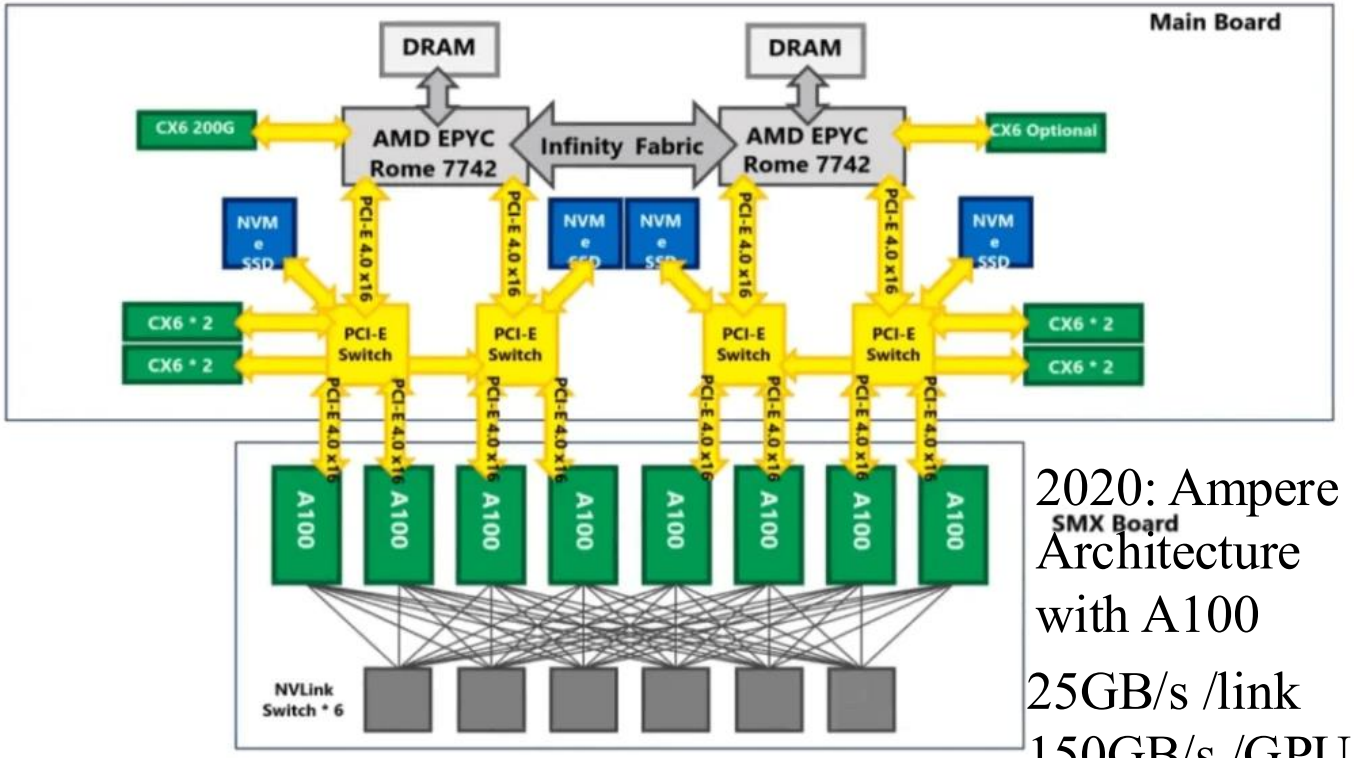
Tensor cores (FP16/Mixed Precision):
120 - 130 TFLOPS

- **V100** has 5120 Cuda cores + **640 Tensor cores** (optimized for 4x4 matrix-multiply-accumulate operations).
  - **To use Tensor cores in cuBLAS:** Call cublasGemmEx()
  - **Python PyTorch:** Set torch.backends.cuda.matmul.allow_tf32 = True

Networking and Communication Library Support in NVIDIA for GPU Clustering

NVLink provides ultra-high-speed GPU-to-GPU interconnects

- Up-to 576 GPUs in Blackwell architecture
- 900GB/s total unidirectional bandwidth per GPU



2020: Ampere Architecture with A100

25GB/s /link

150GB/s /GPU

Collective Communication Library (NCCL)

- Collective: all-gather, all-reduce, broadcast, reduce, reduce-scatter
- Point-to-point send and receive
- SPMD (single-program multiple data). Similar API as MPI.

MPI vs. NVIDIA Collective Communication Library

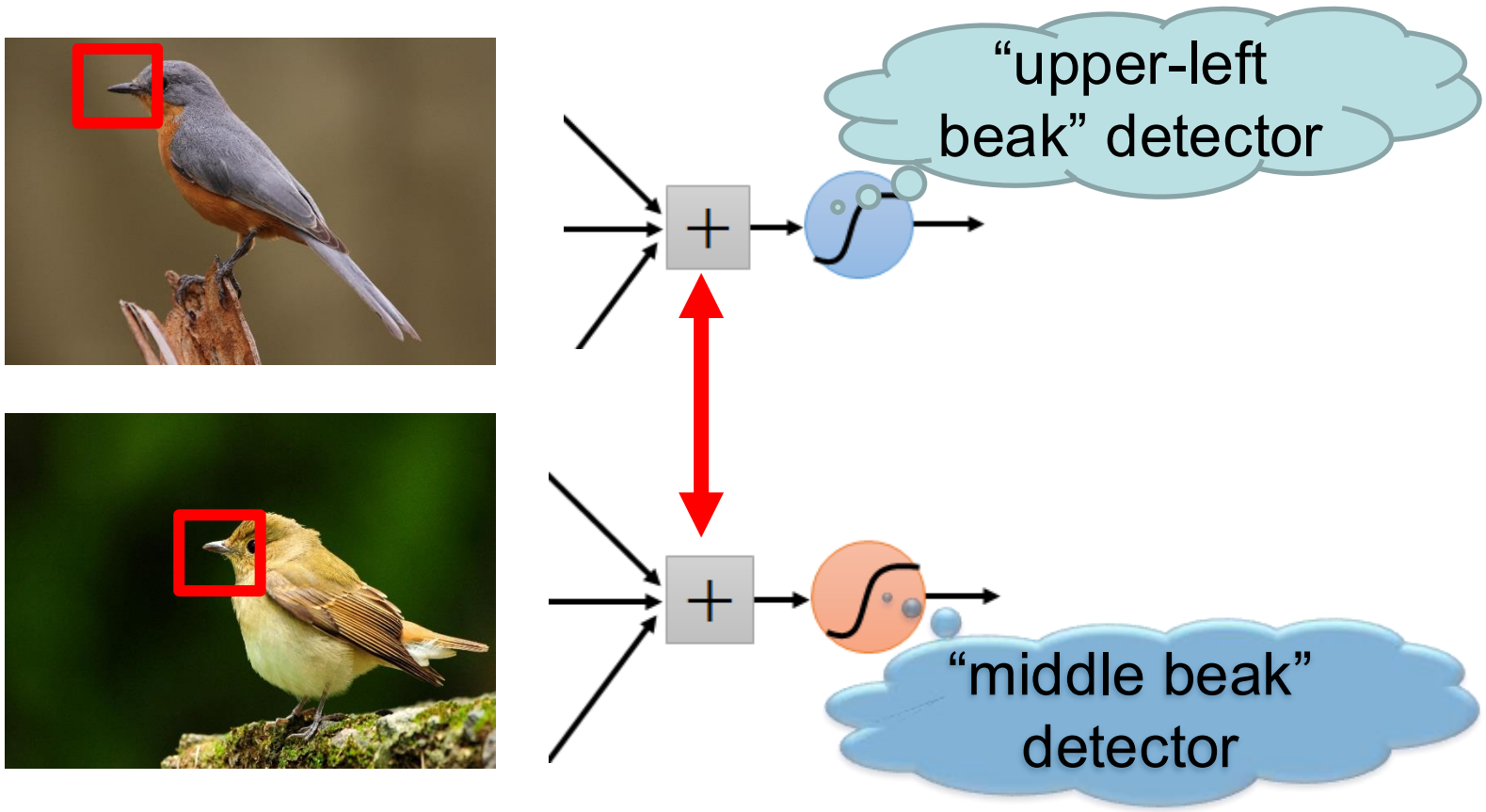
	MPI	NCCL
Who am I?	MPI_Comm_rank(MPI_COMM_WORLD, &rank); 1D process naming	ncclCommUserRank(comm, &rank); 1D GPU device naming
All Reduce	MPI_Allreduce(send_buf, recv_buf, count, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);	ncclAllReduce(d_send, d_recv, count, ncclFloat, ncclSum, comm, stream);
Send Rank 0	MPI_Send(data, count, MPI_FLOAT, 1, 99 /*tag*/, MPI_COMM_WORLD);	ncclSend(d_buf, count, ncclFloat, 1, comm, stream); No tag. Rely on the call order launched into the CUDA stream.
Receive Rank 1	MPI_Recv(data, count, MPI_FLOAT, 0, 99 /*tag*/, MPI_COMM_WORLD, &status);	ncclRecv(d_buf, count, ncclFloat, 0, comm, stream);
Execution	Normally blocking. Completed after function returns.	Asynchronous with stream. Completed after each GPU calls cudaStreamSynchronize(stream)

Outline

- Matrix-based compute-intensive neural image/audio/text processing
  - Text processing
    - Word embeddings
    - Transformer architecture for language models
    - BERT, GPT
  - Convolutional neural network for image object detection
- High throughput/ parallel online service
  - Web search engine

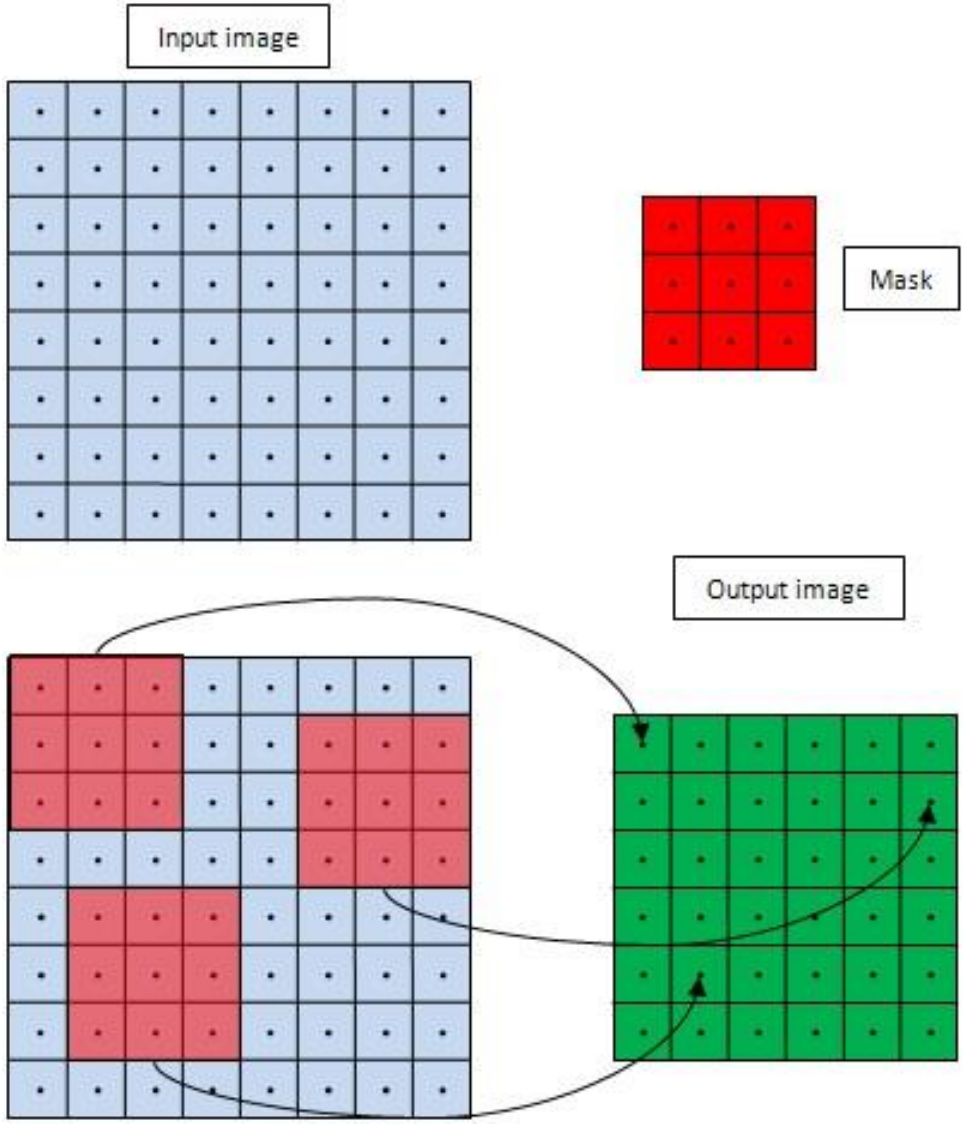
Convolutional Neural Network for Image Object Detection and Classification

An inference model using lots of “small” detectors.
Each detector must “move around”.



Convolutional Neural Network (CNN) for Image Classification/Object Detection

- **Input:** matrix-based pixel image
- **Apply** a pixel-wise filter mask to many positions with multiplication (dot product) and then sum
- **Purpose**: scan an image to look for specific patterns
- **Output:** new intermediate matrix image



Convolution Example

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

Dot product → 3 -1

1	0	0
0	1	1
0	0	1

∘

1	-1	-1
-1	1	-1
-1	-1	1

→ 3

Convolution example

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

Convolution with Multiple Filters

-1	1	-1
-1	1	-1
-1	1	-1

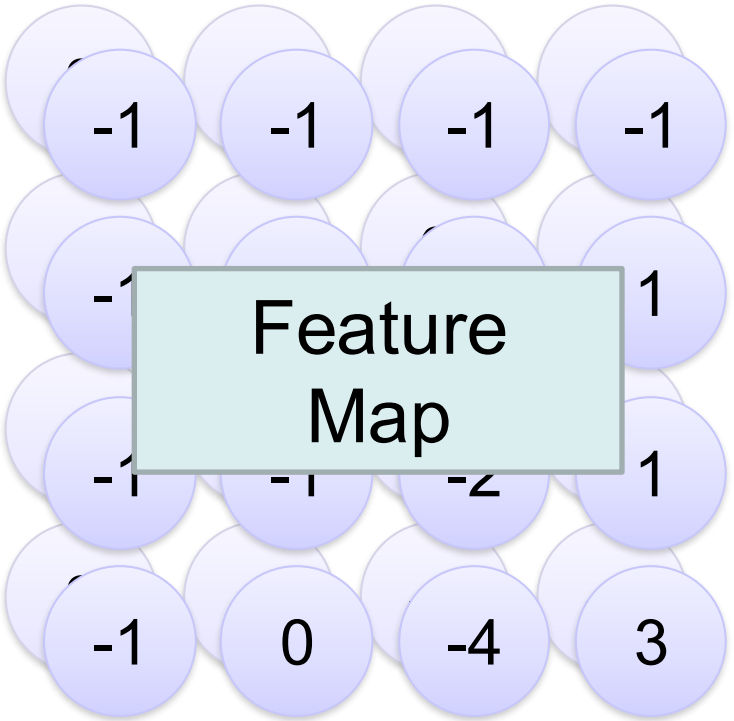
Filter 2

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

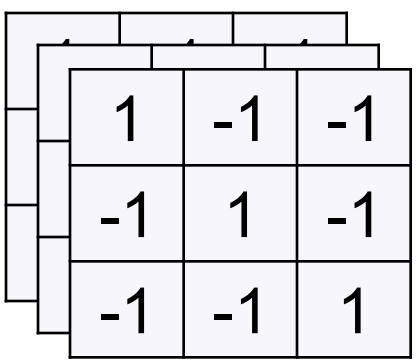
6 x 6 image

Repeat this for each filter

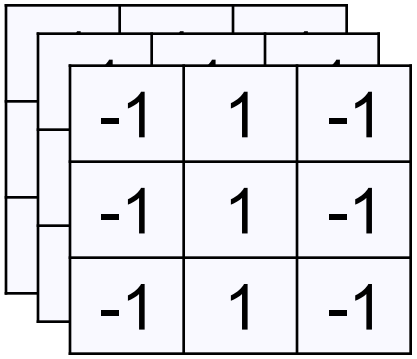


Two 4 x 4 images
Forming 2 x 4 x 4 matrix

Input color image: RGB 3 channels

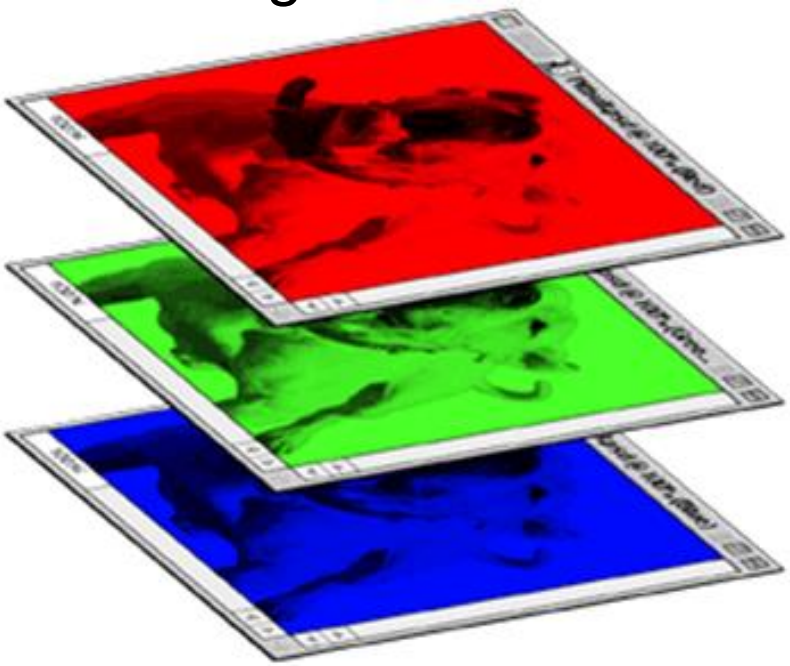


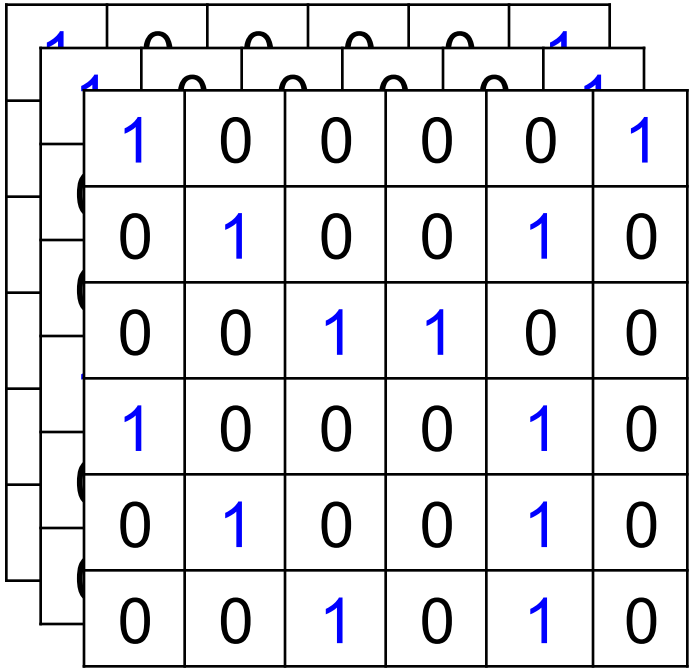
Filter 1



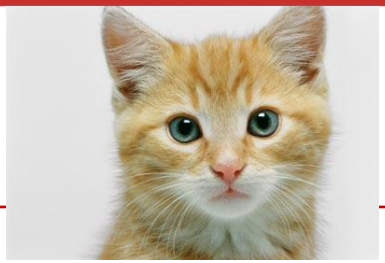
Filter 2

Color image

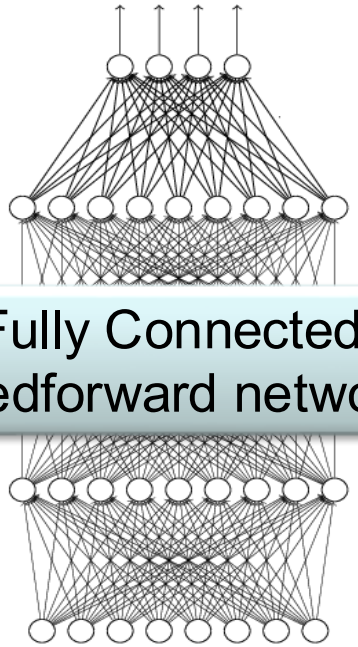




The whole CNN



cat dog ……



Fully Connected Feedforward network

Convolution

Max Pooling

Convolution

Max Pooling

Can repeat many times

Flattened

Why Pooling

- **Subsampling pixels will not change the object**

bird

Subsampling

bird

We can subsample the pixels to make image smaller

→ fewer parameters to characterize the image

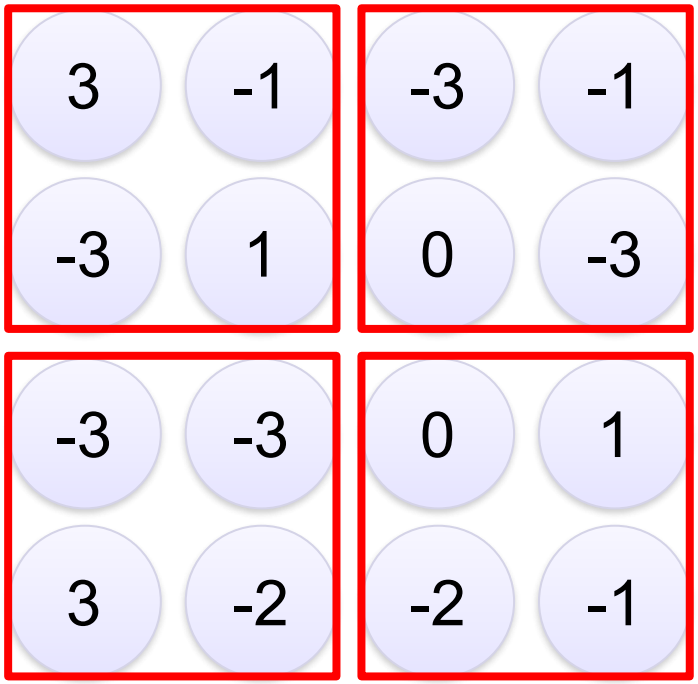
Max Pooling

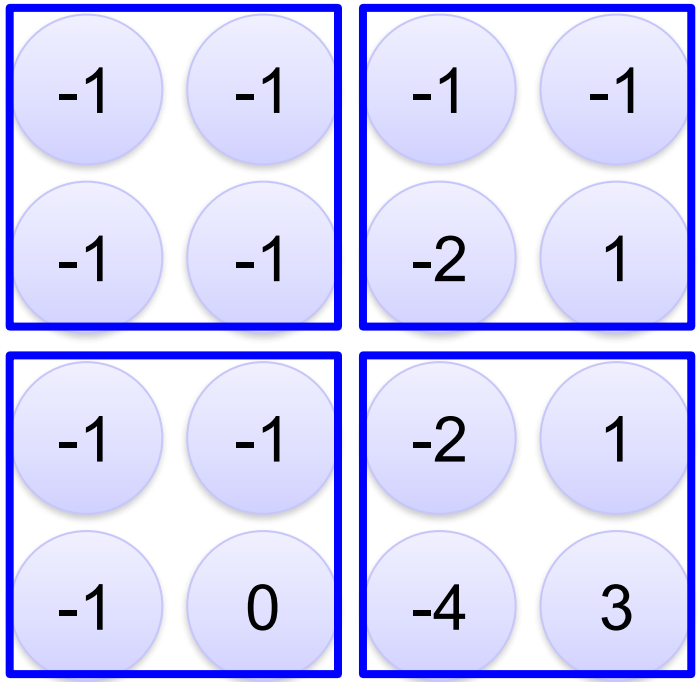
1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

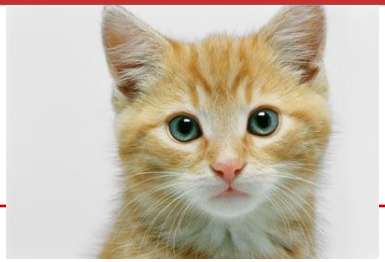
-1	1	-1
-1	1	-1
-1	1	-1

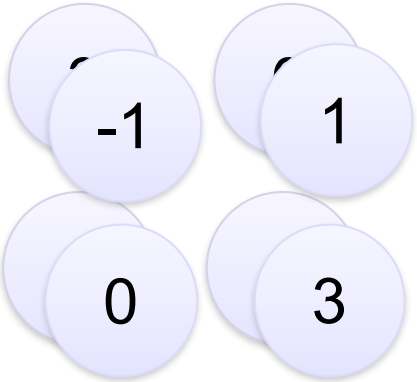
Filter 2





The whole CNN





A new image

Convolution

Max Pooling

Convolution

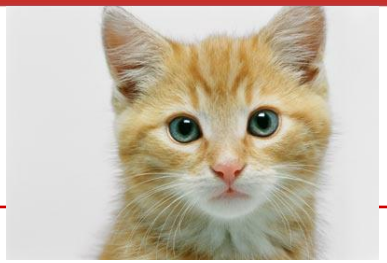
Max Pooling

Can repeat many times

Smaller than the original image

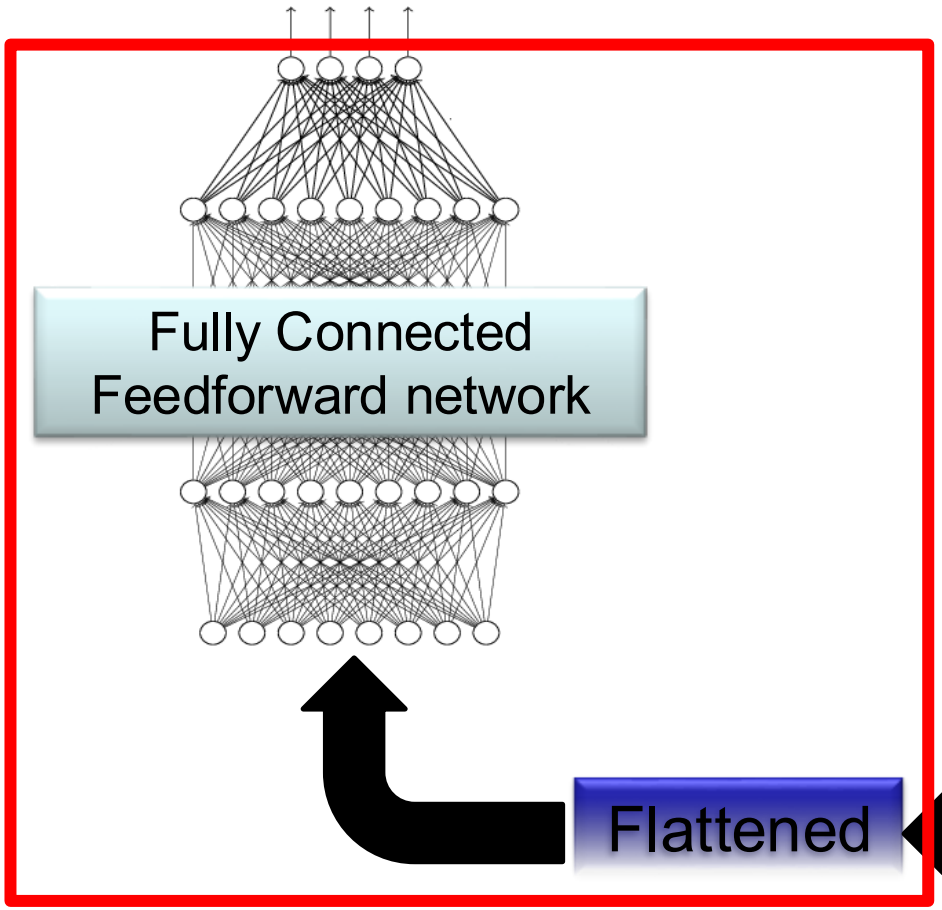
The number of channels is the number of filters

The whole CNN



Detect patterns in lower, middle, and higher layers for edges, shapes, parts, objects

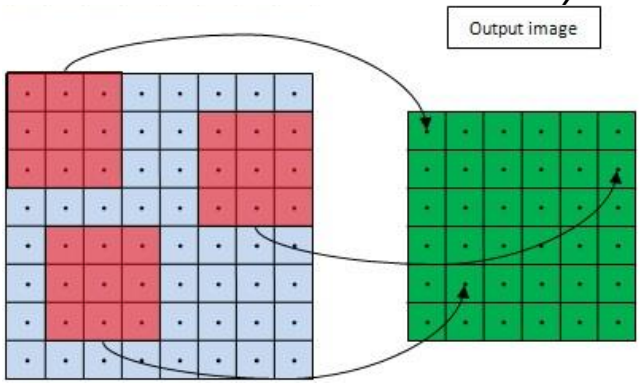
cat dog ……



Optional exercise on SGD training and model inference: simplified CNN to detect a beak represented by a triangle

- Step 1:Convolutional layer with 5x5 kernel and ReLU

$$\text{conv\_map}[i][j] = \max\left(0, \sum_{ki=0}^{4}\sum_{kj=0}^{4}(\text{input}[i+ki][j+kj] \cdot \text{conv.weights}[ki][kj]) + \text{conv.bias}\right)$$



- Step 2: Max pooling with 4x4 window

$$\text{pooled\_map}[i][j] = \max_{pi,pj \in [0,3]}(\text{conv\_map}[i \cdot 4 + pi][j \cdot 4 + pj])$$

- Step 3: Flattening with 2D-to-1D linearization

$$\text{flat\_pooled}[k] = \text{pooled\_map}\,[\lfloor k/\text{size} \rfloor]\,[k \pmod{\text{size}}]$$

Object detection output of simplified CNN

- Step 4: Object detection with a dense layer

Logit $$z = \sum_{i=0}^{N} (\text{flat\_pooled}[i] \cdot \text{dense.weights}[i]) + \text{dense.bias}$$

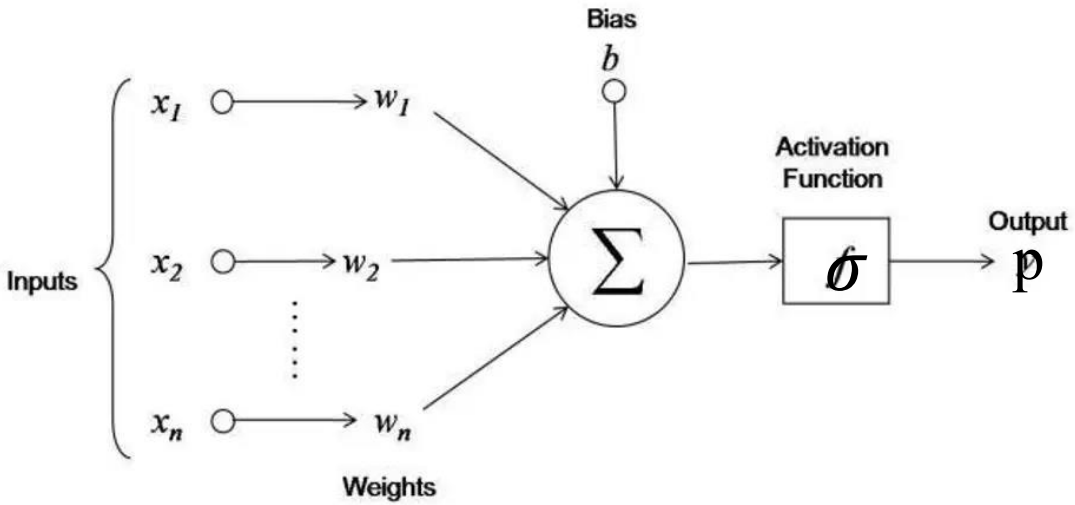
- Step 5: Convert to probability (p) via sigmoid

$$p = \sigma(z) = \frac{1}{1 + e^{-z}}$$

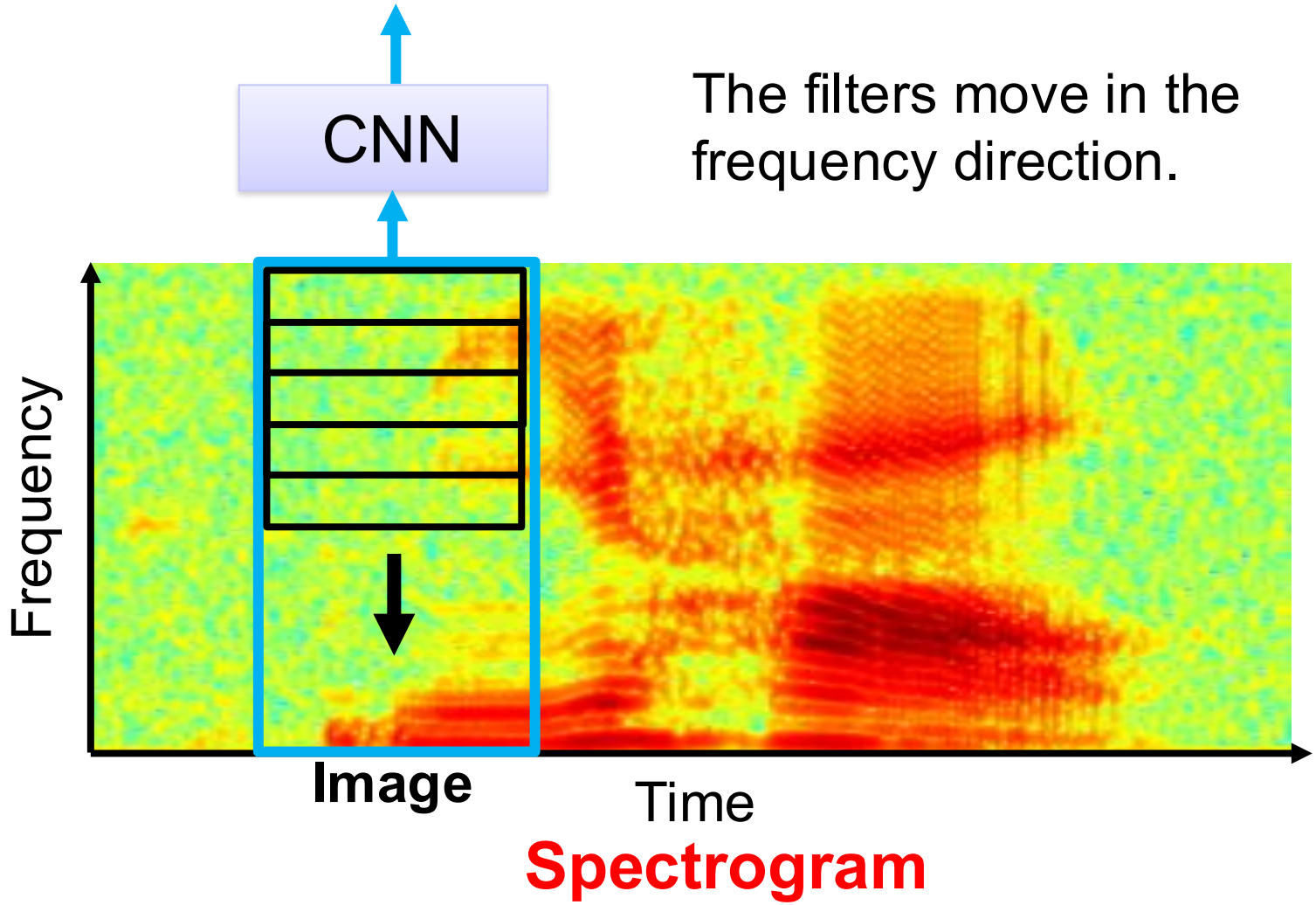
≅ 1 means object detected

≅ 0 means no object detected

Flattened image

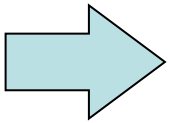


Neural network computation in speech recognition



Outline

- Matrix-based compute-intensive neural image/audio/text processing
  - Text processing
    - Word embeddings
    - Transformer architecture for language models
    - BERT, GPT
  - Convolutional neural network for image object detection

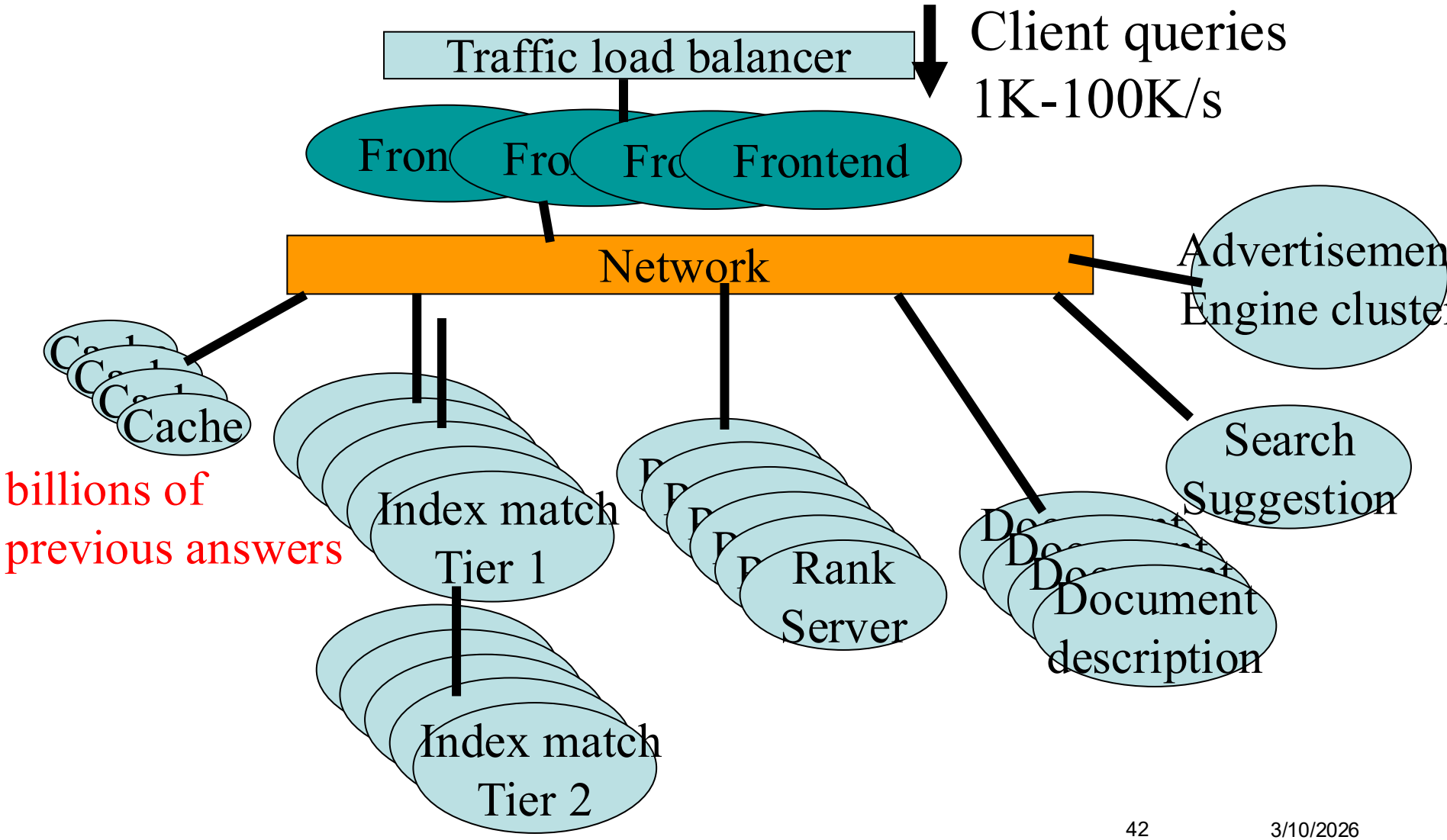


- High throughput/ parallel online service
  - Web search engine

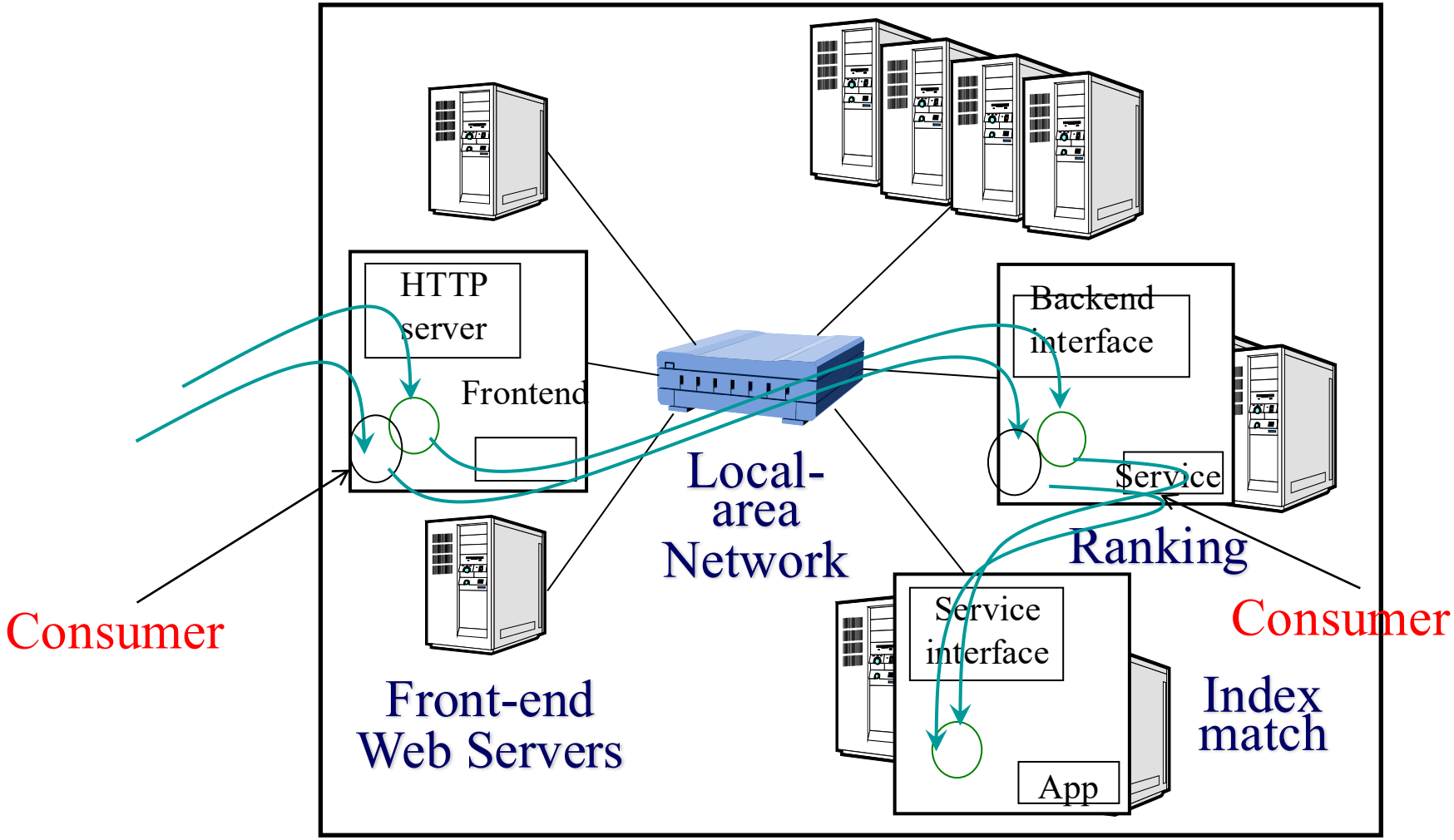
Programming Challenges for Online Services

- **Challenges/requirements for online services:**
  - Data intensive, requiring large-scale clusters.
    - 1000 to 10,000 cluster is not uncommon.
  - Incremental scalability
  - Resource management
- **7×24 availability and fault tolerance:**
  - Operation errors
  - Software bugs
  - Hardware failures

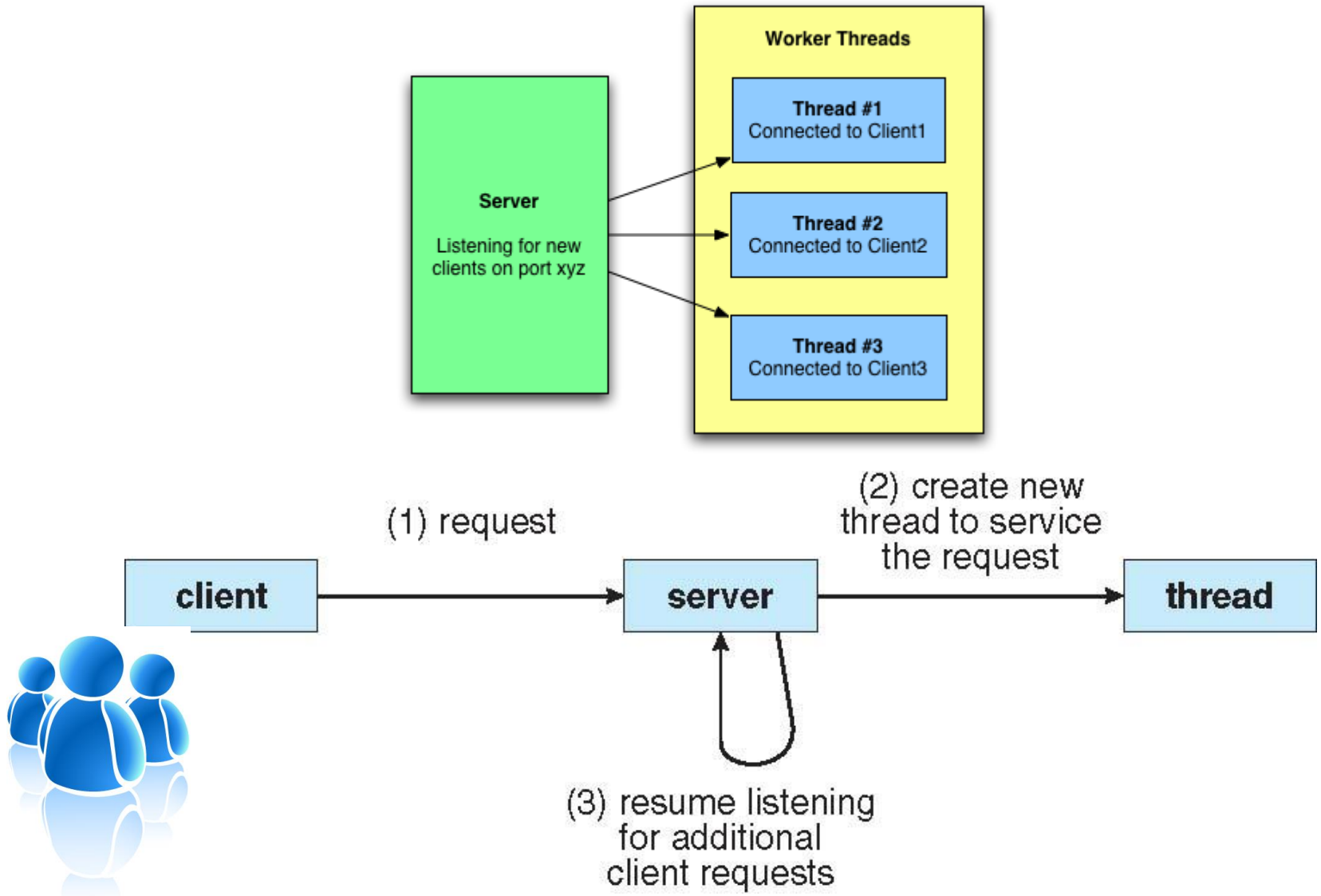
Multi-tier Web Services: Search Engine



Example of request flow in a large-scale web search engine



Multi-threaded request processing inside each server



Summary

- Parallel matrix-based machine learning applications for fast inference and iterative model training
  - Vision: image representation with matrices
    - Object detection with filter based convolutional neural network
  - Audio: speech frequency over time with matrix representation
  - Text: High dimension word and document embeddings in transformers and LLMs
  - NVIDIA NACCL: MPI-like programming for clustered GPUs
- Large-scale web services (e.g. search)
  - Threads are critical for low latency, high throughput, and fault tolerance.