Studies in feature learning through the lens of sparse Boolean functions

Ben Edelman (Harvard)

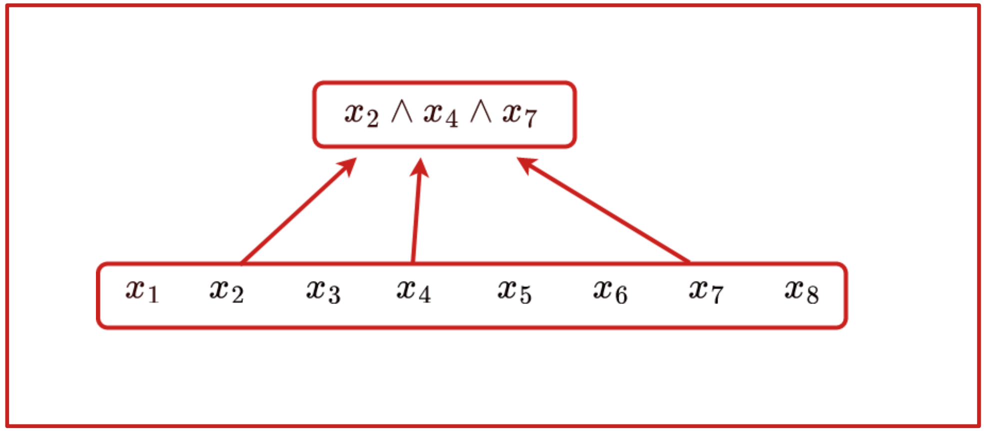
mlfoundations.org

Seminar in Mathematics, Physics & Machine Learning

IST

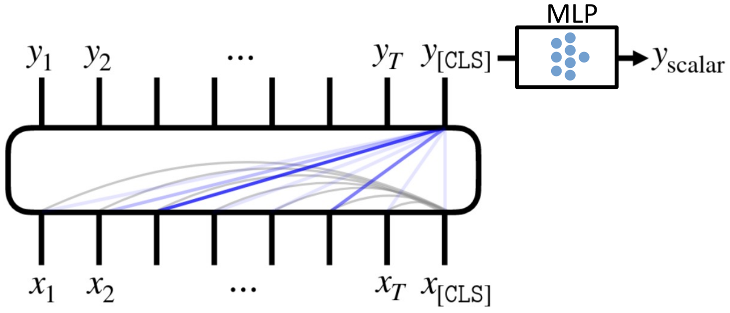
Themes

- What drives feature learning in modern and classic architectures?
  - Understanding capacity & expressivity & optimization
- Approach: focus on idealized synthetic tasks. Specifically, running theme of **sparse functions**
  - Implicit theme: can we understand representation learning as circuit learning?



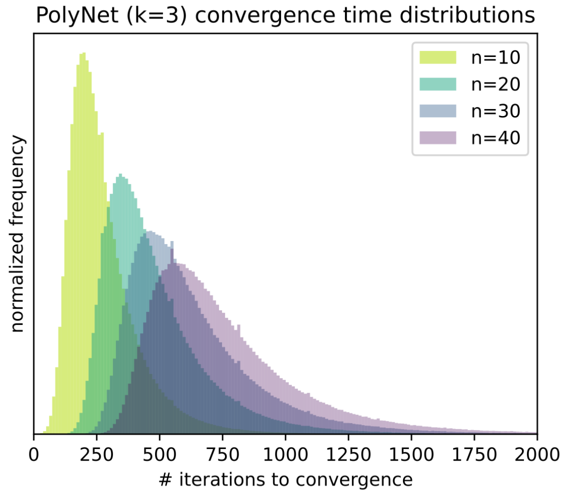
Part 1: Self-Attention & Transformers

Joint work with Surbhi Goel, Sham Kakade, & Cyril Zhang



Part 2: Parities & Emergence

Joint work with Boaz Barak, Surbhi Goel, Sham Kakade, Eran Malach, & Cyril Zhang



Part 1

Inductive Biases and Variable Creation in Self-Attention Mechanisms, ICML ‘22

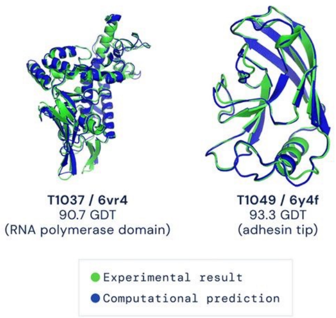
with Surbhi Goel, Sham Kakade, & Cyril Zhang

The Self-Attention Revolution

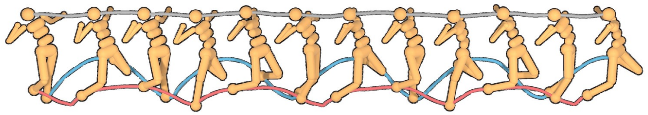
Language

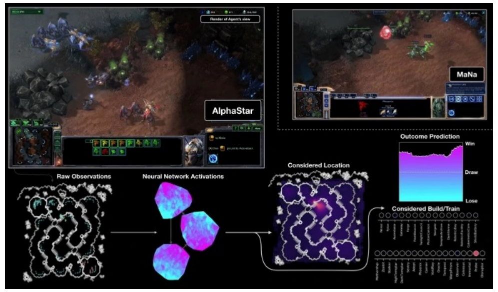
OpenAI GPT-3

Computational biology

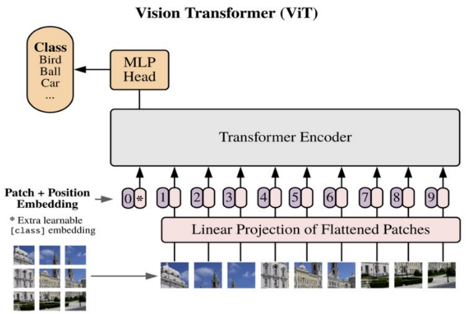


Reinforcement learning

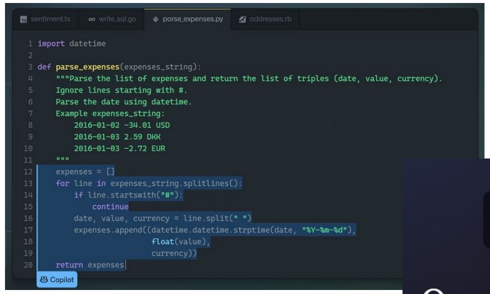




Computer vision



Automated programming



AlphaCode

OpenAI Codex

Mathematics

Question: A line parallel to $y = 4x + 6$ passes through $(5, 10)$. What is the y-coordinate of the point where this line crosses the y-axis?

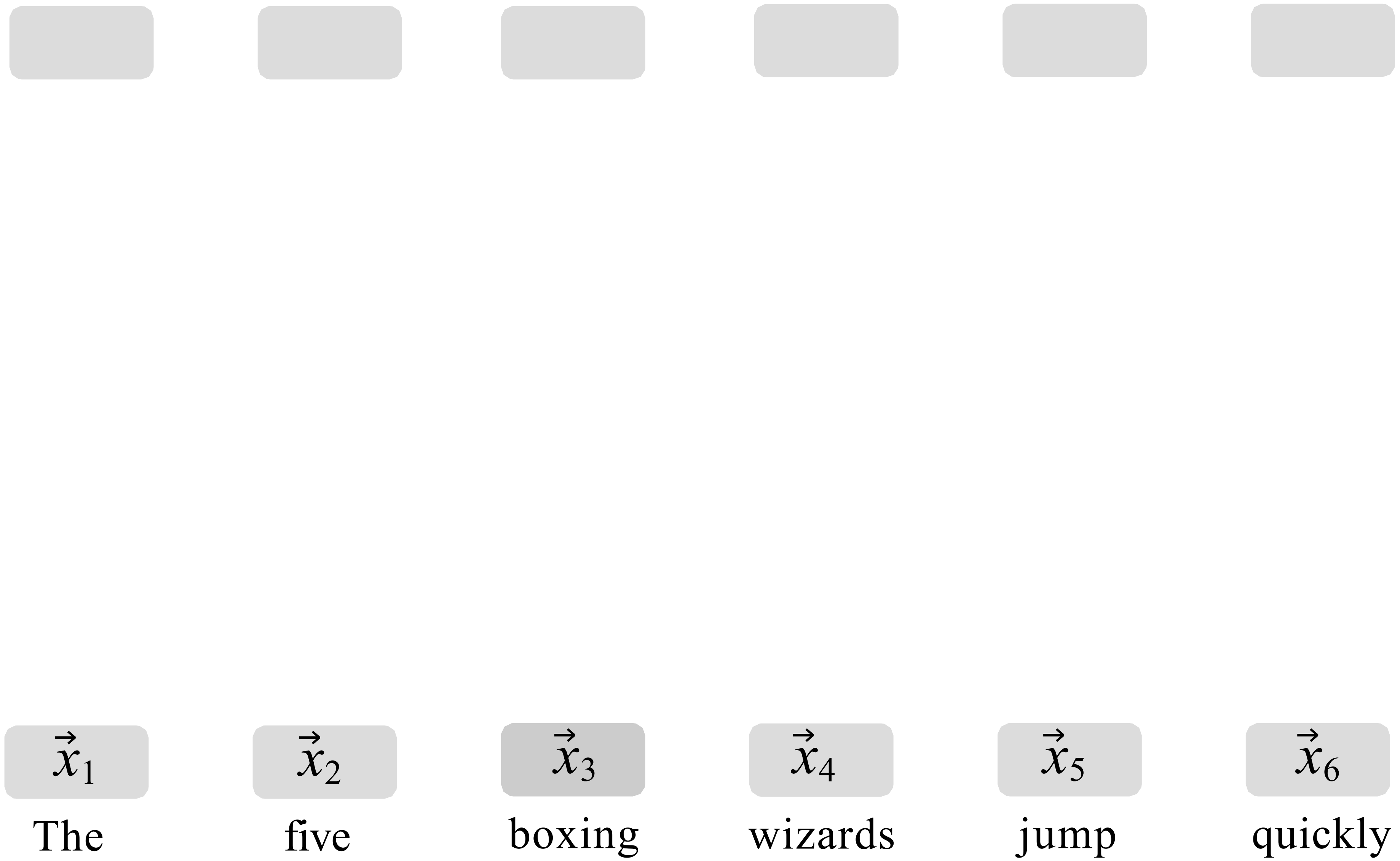
Model output: The line is parallel to $y = 4x + 6$, which means that it has the same slope as $y = 4x + 6$. Since the line passes through $(5, 10)$, we can use the point-slope form of a line to find the equation of the line:

$$y - 10 = 4(x - 5)$$
$$y - 10 = 4x - 20$$
$$y = 4x - 10$$

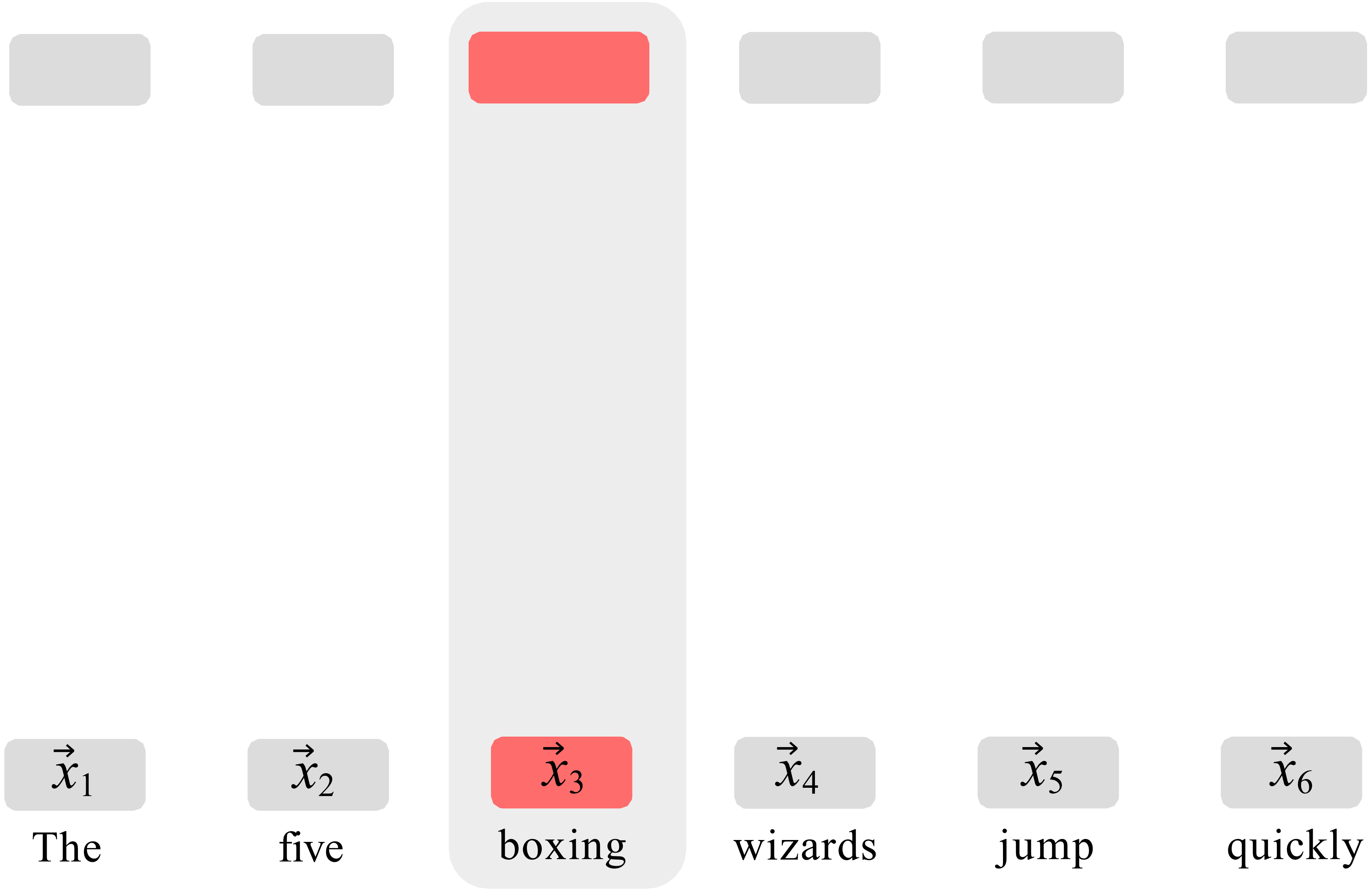
Now that we have the equation of the line, we can find the y-coordinate of the point where the line crosses the y-axis by substituting $x = 0$ into the equation:

$$y = 4 \cdot 0 - 10 = \boxed{-10}.$$

etc.


$\vec{x}_1$ $\vec{x}_2$ $\vec{x}_3$ $\vec{x}_4$ $\vec{x}_5$ $\vec{x}_6$
The five boxing wizards jump quickly

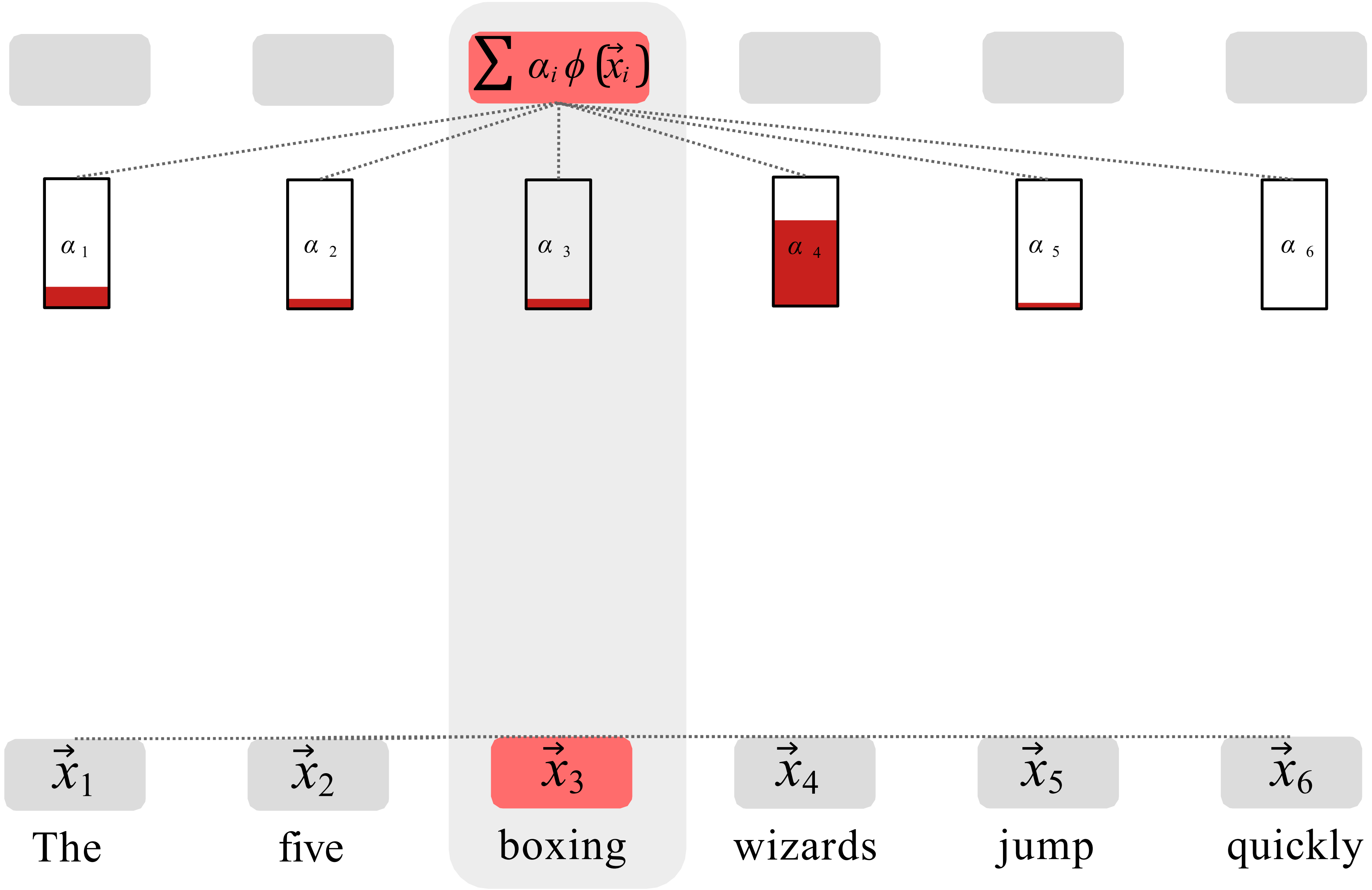
$\vec{x}_i \in \mathbb{R}_d$

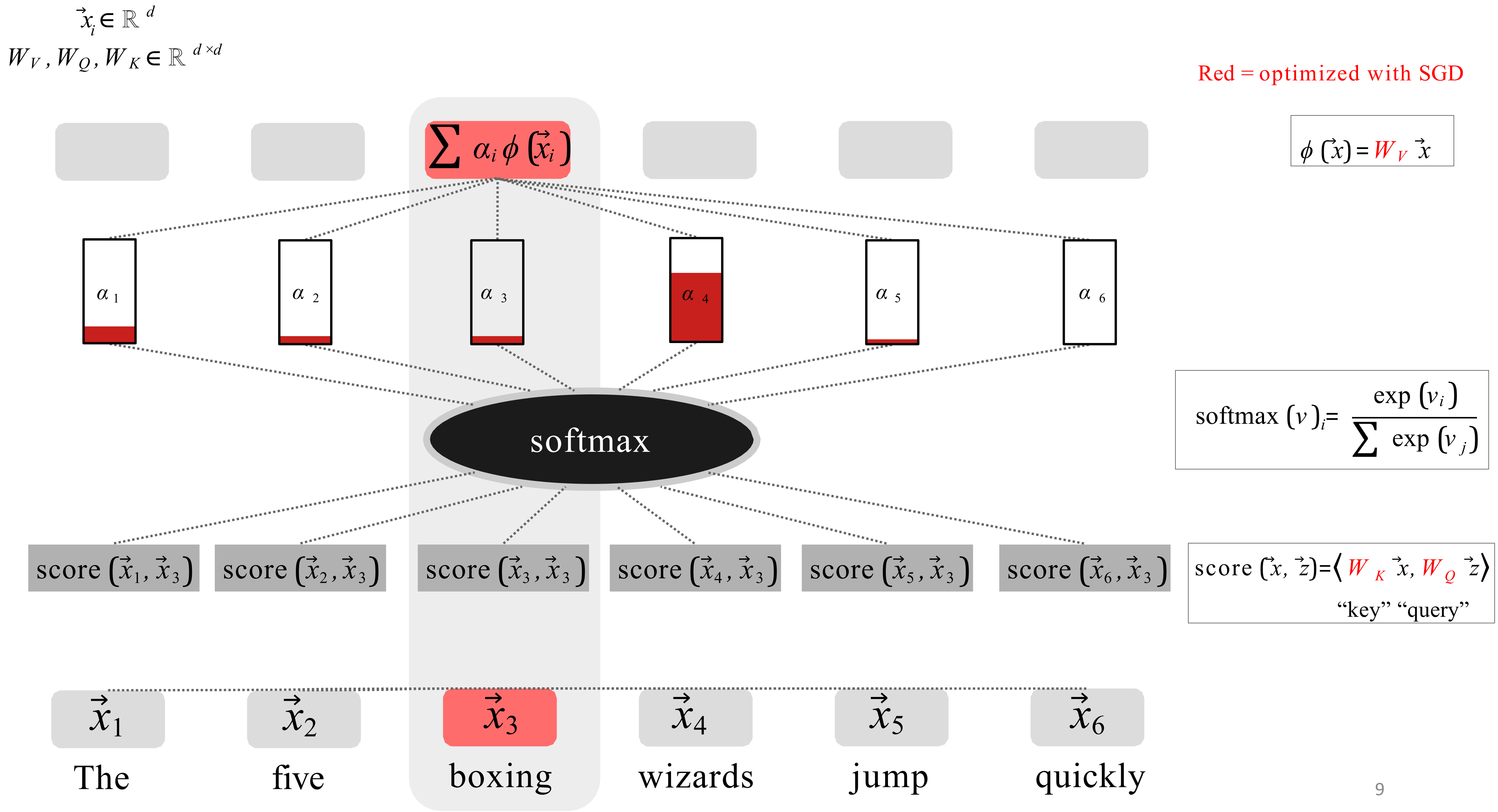


$\vec{x}_i \in \mathbb{R}^d$
$W_V \in \mathbb{R}^{d \times d}$

Red = optimized with SGD

$\phi(\vec{x}) = W_V \vec{x}$





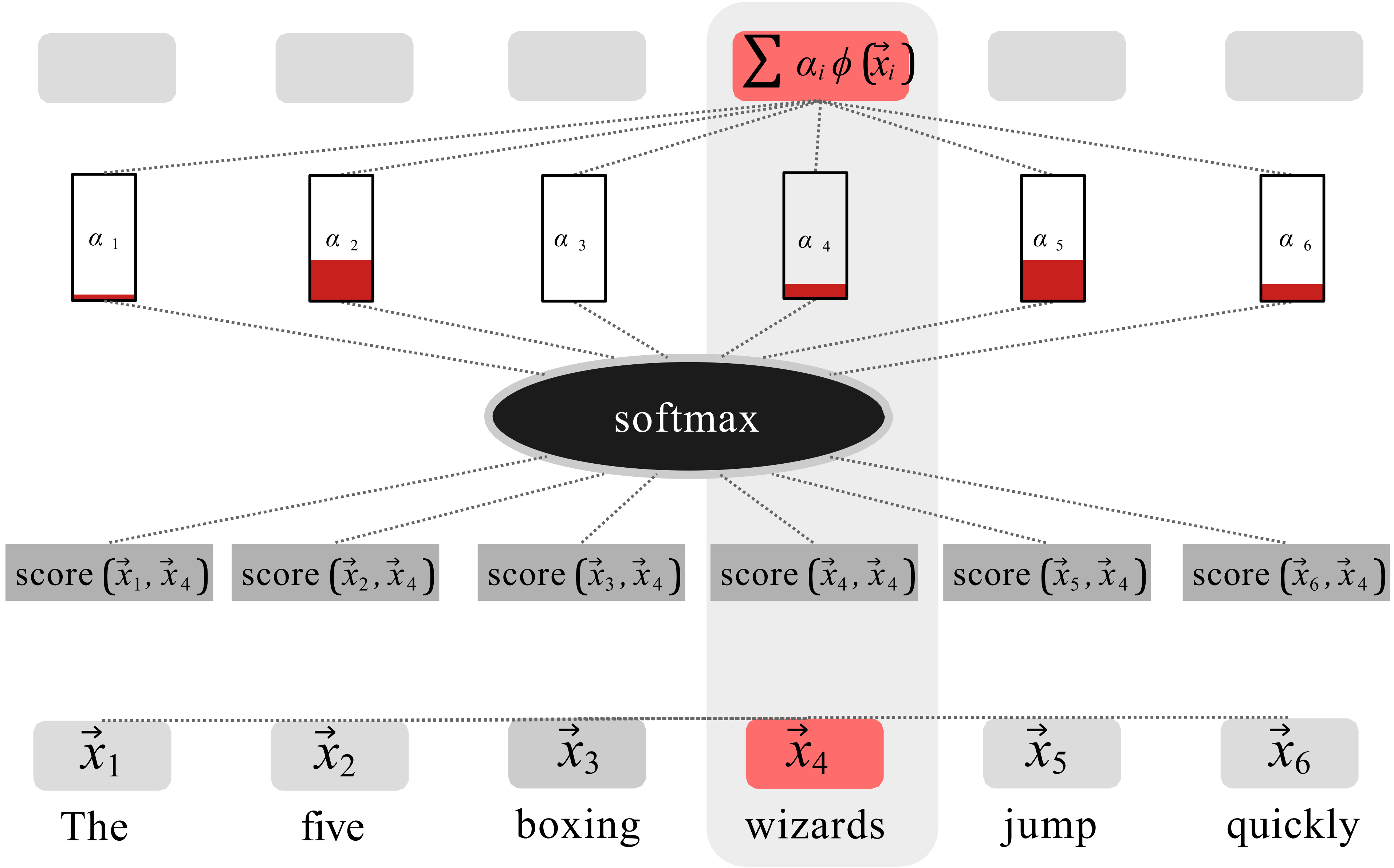
$\vec{x}_i \in \mathbb{R}^d$
$W_V, W_Q, W_K \in \mathbb{R}^{d \times d}$
Red = optimized with SGD
$\sum \alpha_i \phi(\vec{x}_i)$
$\phi(\vec{x}) = W_V \vec{x}$
α_1
α_2
α_3
α_4
α_5
α_6
softmax
$\text{softmax}(v)_i = \frac{\exp(v_i)}{\sum \exp(v_j)}$
$\text{score}(\vec{x}_1, \vec{x}_3)$
$\text{score}(\vec{x}_2, \vec{x}_3)$
$\text{score}(\vec{x}_3, \vec{x}_3)$
$\text{score}(\vec{x}_4, \vec{x}_3)$
$\text{score}(\vec{x}_5, \vec{x}_3)$
$\text{score}(\vec{x}_6, \vec{x}_3)$
$\text{score}(\vec{x}, \vec{z}) = \langle W_K \vec{x}, W_Q \vec{z} \rangle$
“key” “query”
$\vec{x}_1$
$\vec{x}_2$
$\vec{x}_3$
$\vec{x}_4$
$\vec{x}_5$
$\vec{x}_6$
The
five
boxing
wizards
jump
quickly

$\vec{x}_i \in \mathbb{R}^d$

$W_V, W_Q, W_K \in \mathbb{R}^{d \times d}$

Red = optimized with SGD

$\phi(\vec{x}) = W_V \vec{x}$



$$\text{softmax}(v)_i = \frac{\exp(v_i)}{\sum \exp(v_j)}$$

$$\text{score}(\vec{x}, \vec{z}) = \langle W_K \vec{x}, W_Q \vec{z} \rangle$$

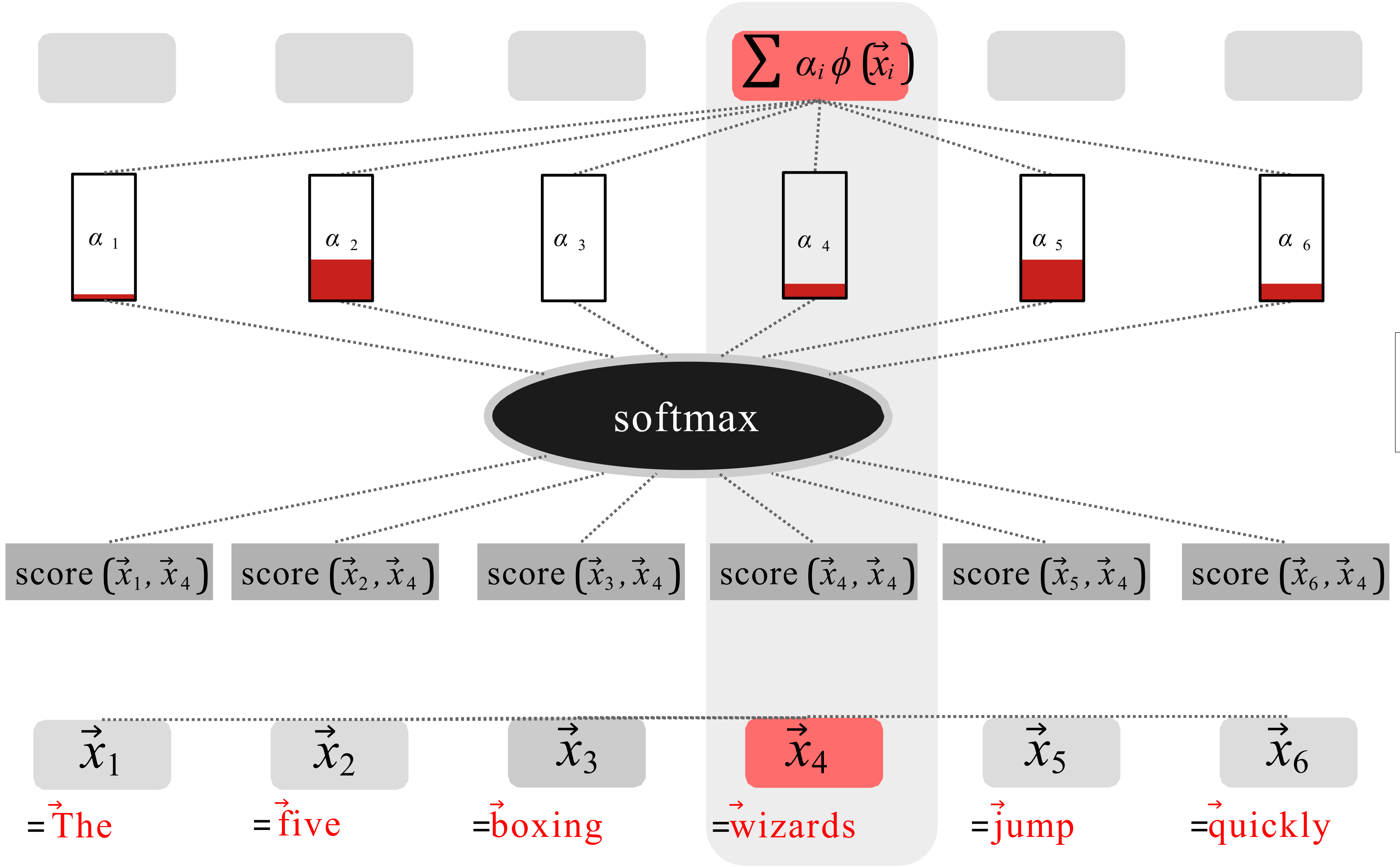
"key" "query"

$\vec{x}_i \in \mathbb{R}^d$

$W_V, W_Q, W_K \in \mathbb{R}^{d \times d}$

Red = optimized with SGD

$\phi(\vec{x}) = W_V \vec{x}$

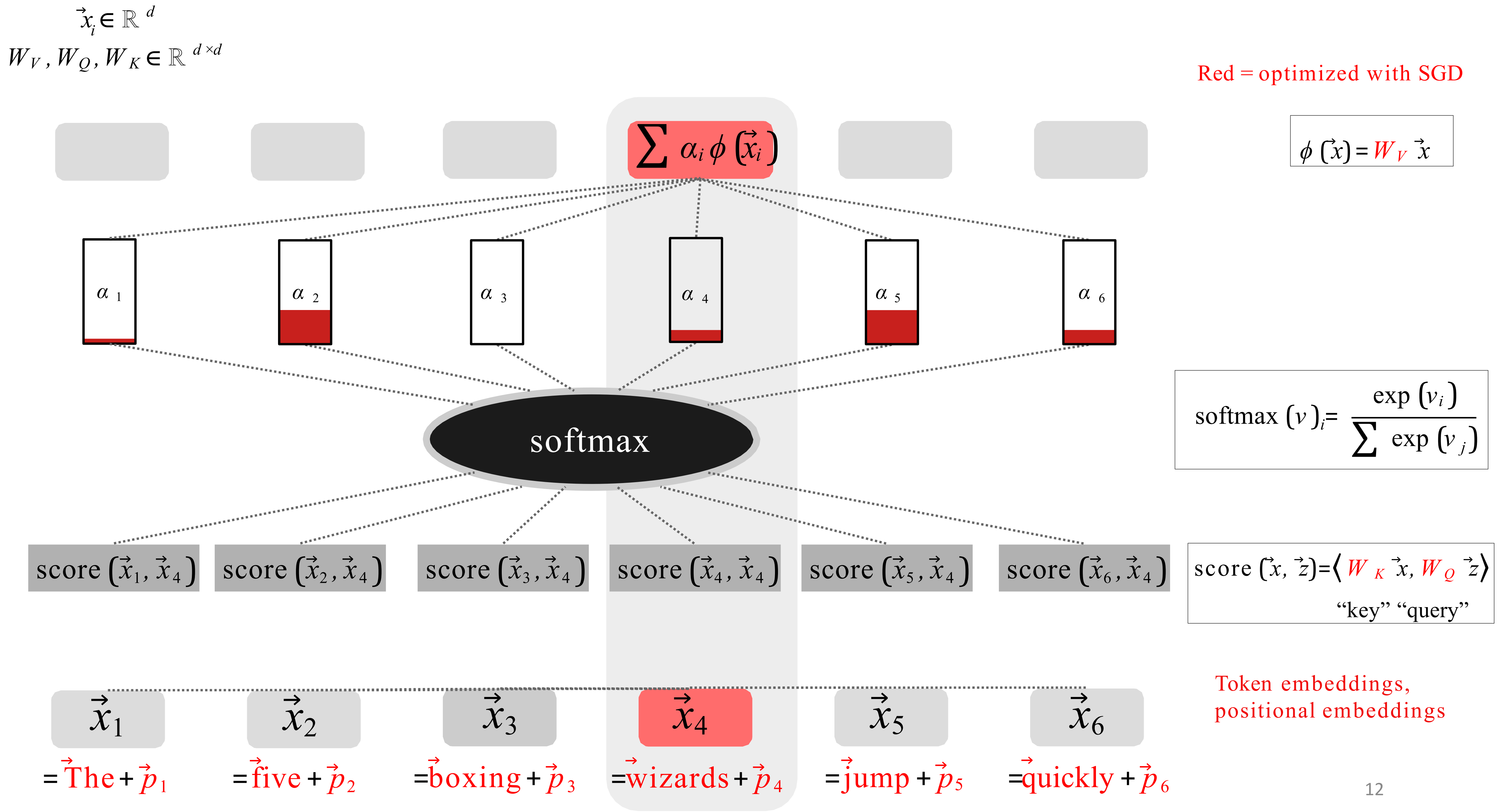


$$\text{softmax}(v)_i = \frac{\exp(v_i)}{\sum \exp(v_j)}$$

$$\text{score}(\vec{x}, \vec{z}) = \langle W_K \vec{x}, W_Q \vec{z} \rangle$$

"key" "query"

Token embeddings


$\vec{x}_i \in \mathbb{R}^d$
$W_V, W_Q, W_K \in \mathbb{R}^{d \times d}$
Red = optimized with SGD
$\sum \alpha_i \phi(\vec{x}_i)$
$\phi(\vec{x}) = W_V \vec{x}$
α_1
α_2
α_3
α_4
α_5
α_6
softmax
$\text{softmax}(v)_i = \frac{\exp(v_i)}{\sum \exp(v_j)}$
$\text{score}(\vec{x}_1, \vec{x}_4)$
$\text{score}(\vec{x}_2, \vec{x}_4)$
$\text{score}(\vec{x}_3, \vec{x}_4)$
$\text{score}(\vec{x}_4, \vec{x}_4)$
$\text{score}(\vec{x}_5, \vec{x}_4)$
$\text{score}(\vec{x}_6, \vec{x}_4)$
$\text{score}(\vec{x}, \vec{z}) = \langle W_K \vec{x}, W_Q \vec{z} \rangle$
“key” “query”
Token embeddings, positional embeddings
$\vec{x}_1$
$\vec{x}_2$
$\vec{x}_3$
$\vec{x}_4$
$\vec{x}_5$
$\vec{x}_6$
$= \vec{\text{The}} + \vec{p}_1$
$= \vec{\text{five}} + \vec{p}_2$
$= \vec{\text{boxing}} + \vec{p}_3$
$= \vec{\text{wizards}} + \vec{p}_4$
$= \vec{\text{jump}} + \vec{p}_5$
$= \vec{\text{quickly}} + \vec{p}_6$

$\vec{x}_i \in \mathbb{R}^d$

$W_V, W_Q, W_K \in \mathbb{R}^{d \times d}$

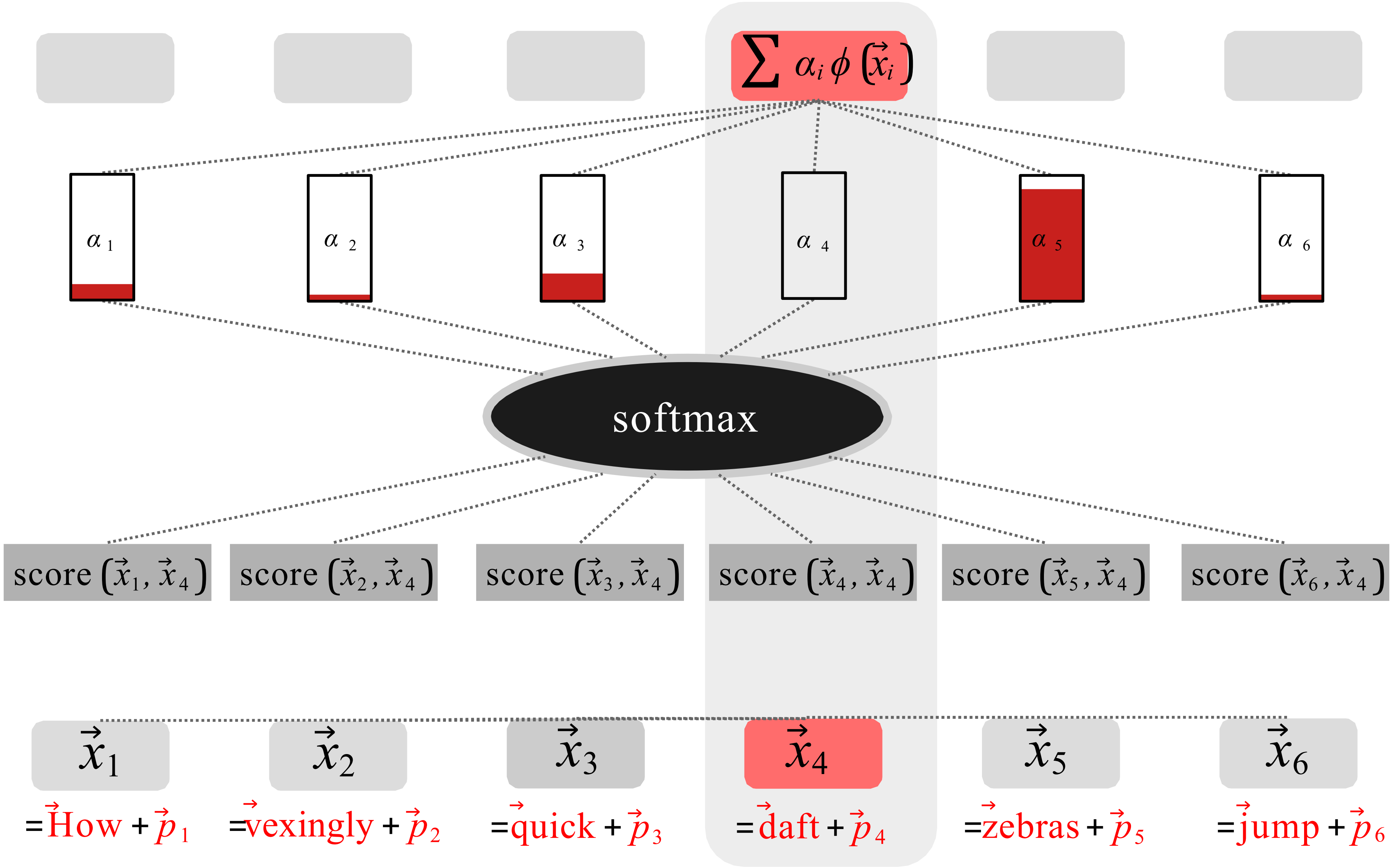
Red = optimized with SGD

$\phi(\vec{x}) = W_V \vec{x}$

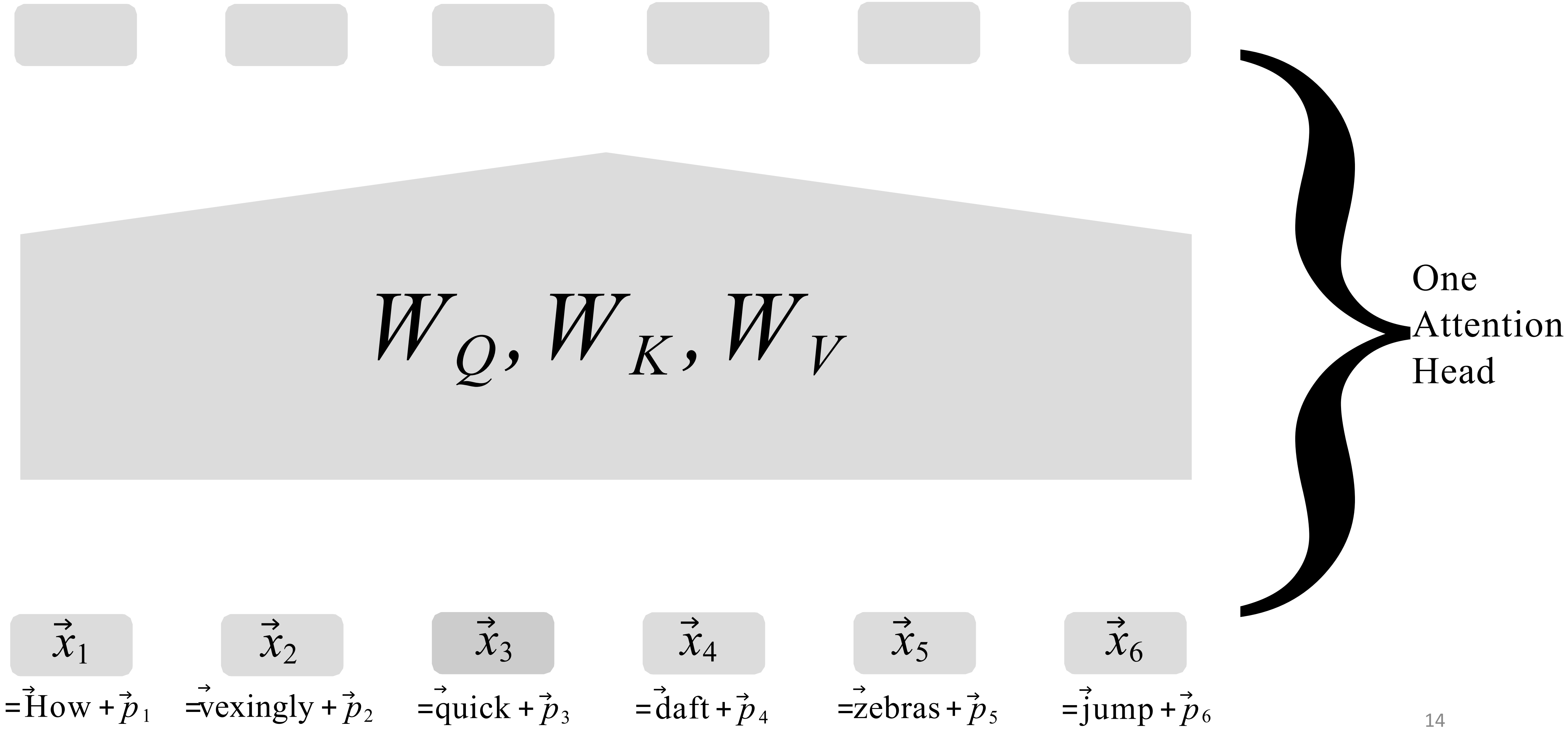
$\text{softmax}(v)_i = \frac{\exp(v_i)}{\sum \exp(v_j)}$

$\text{score}(\vec{x}, \vec{z}) = \langle W_K \vec{x}, W_Q \vec{z} \rangle$

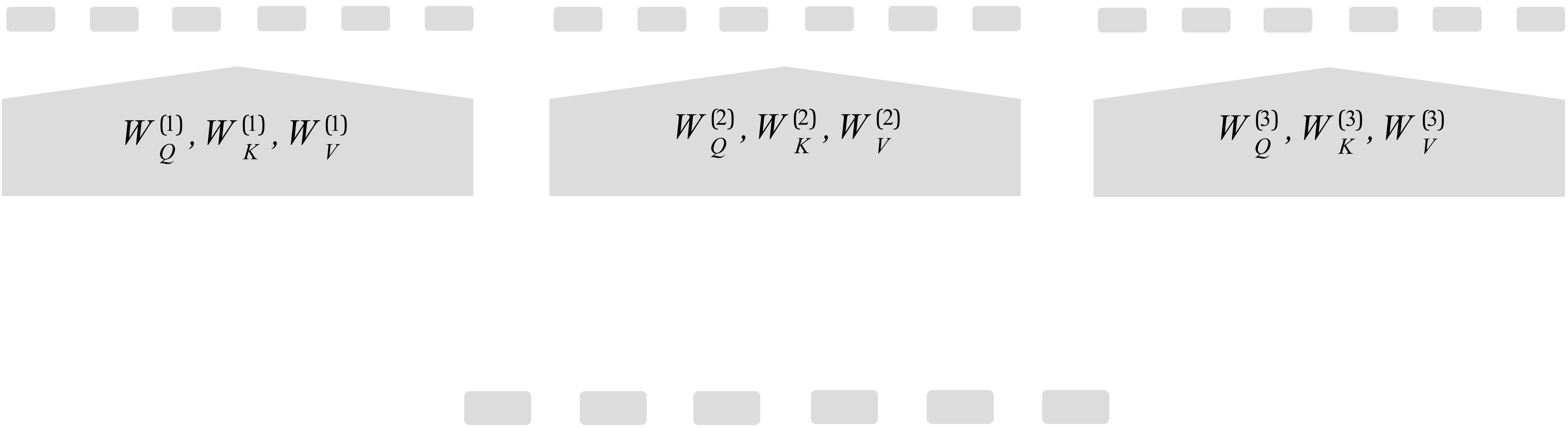
"key" "query"



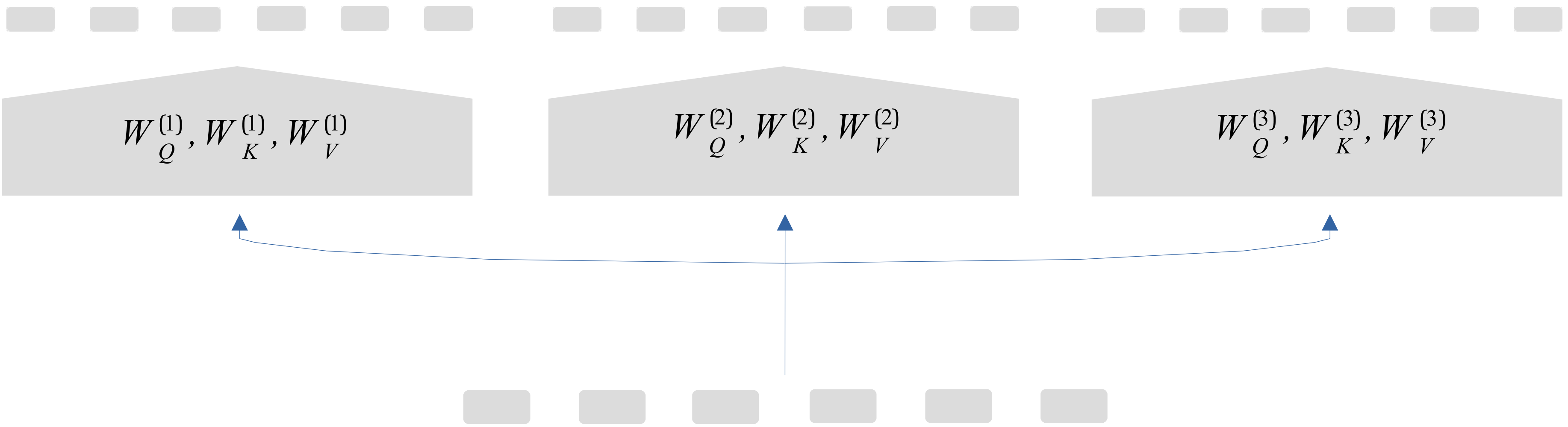
Token embeddings, positional embeddings


W_Q, W_K, W_V
$\vec{x}_1$
$\vec{x}_2$
$\vec{x}_3$
$\vec{x}_4$
$\vec{x}_5$
$\vec{x}_6$
$= \vec{\text{How}} + \vec{p}_1$
$= \vec{\text{vexingly}} + \vec{p}_2$
$= \vec{\text{quick}} + \vec{p}_3$
$= \vec{\text{daft}} + \vec{p}_4$
$= \vec{\text{zebras}} + \vec{p}_5$
$= \vec{\text{jump}} + \vec{p}_6$
One Attention Head

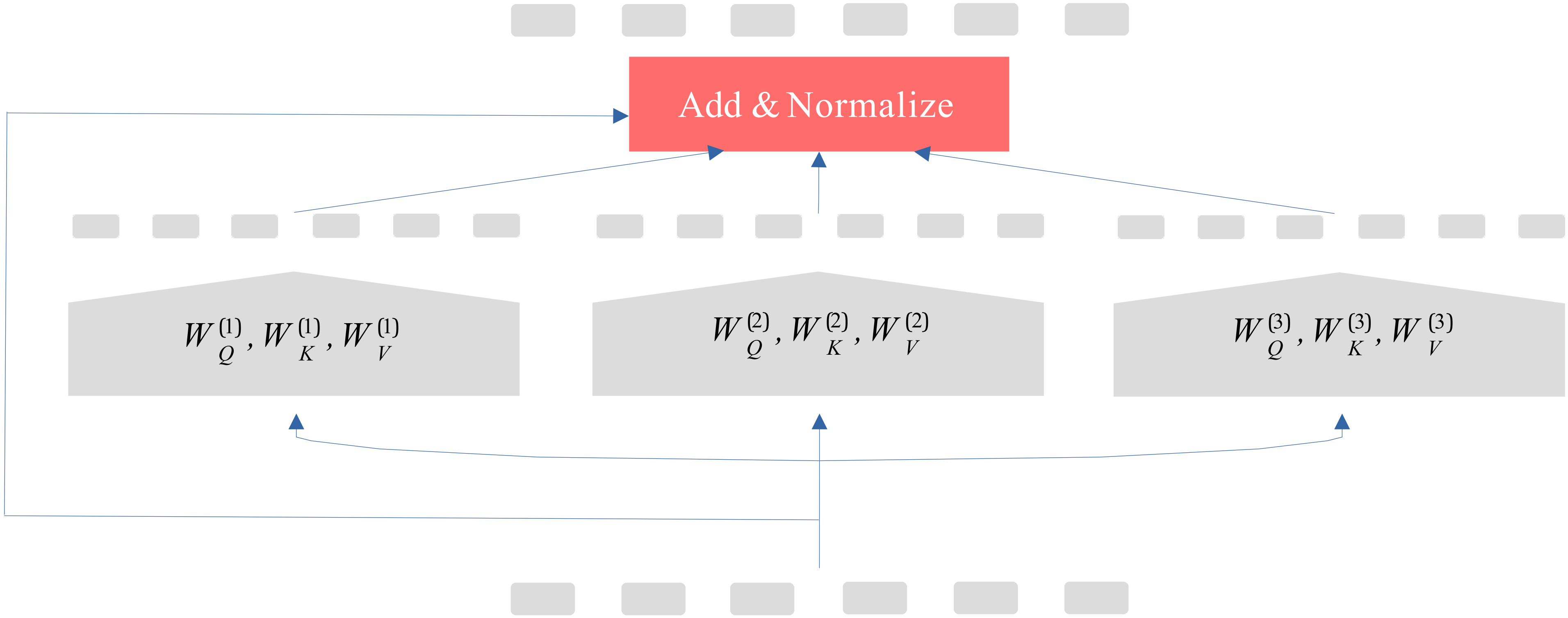
Transformer



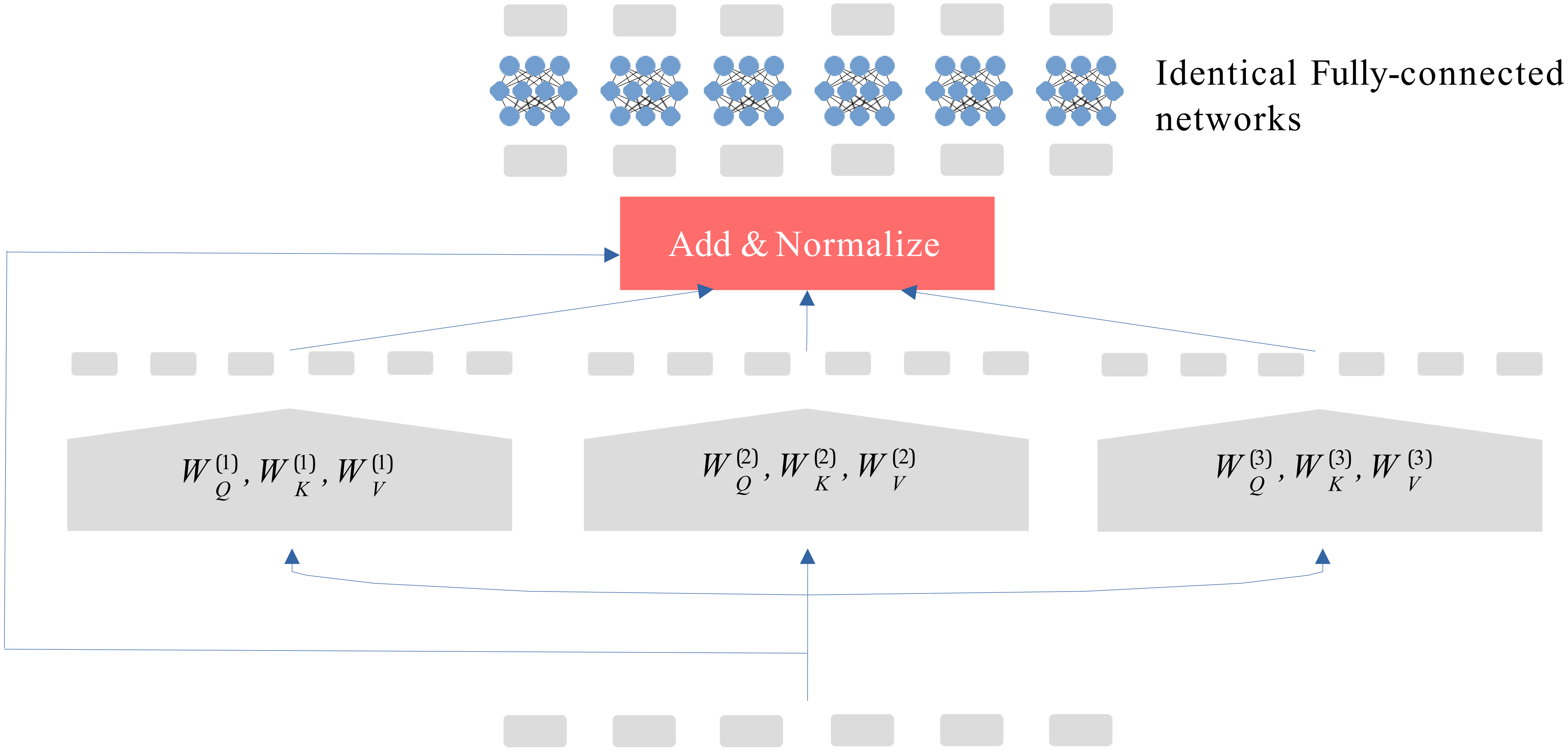
Transformer



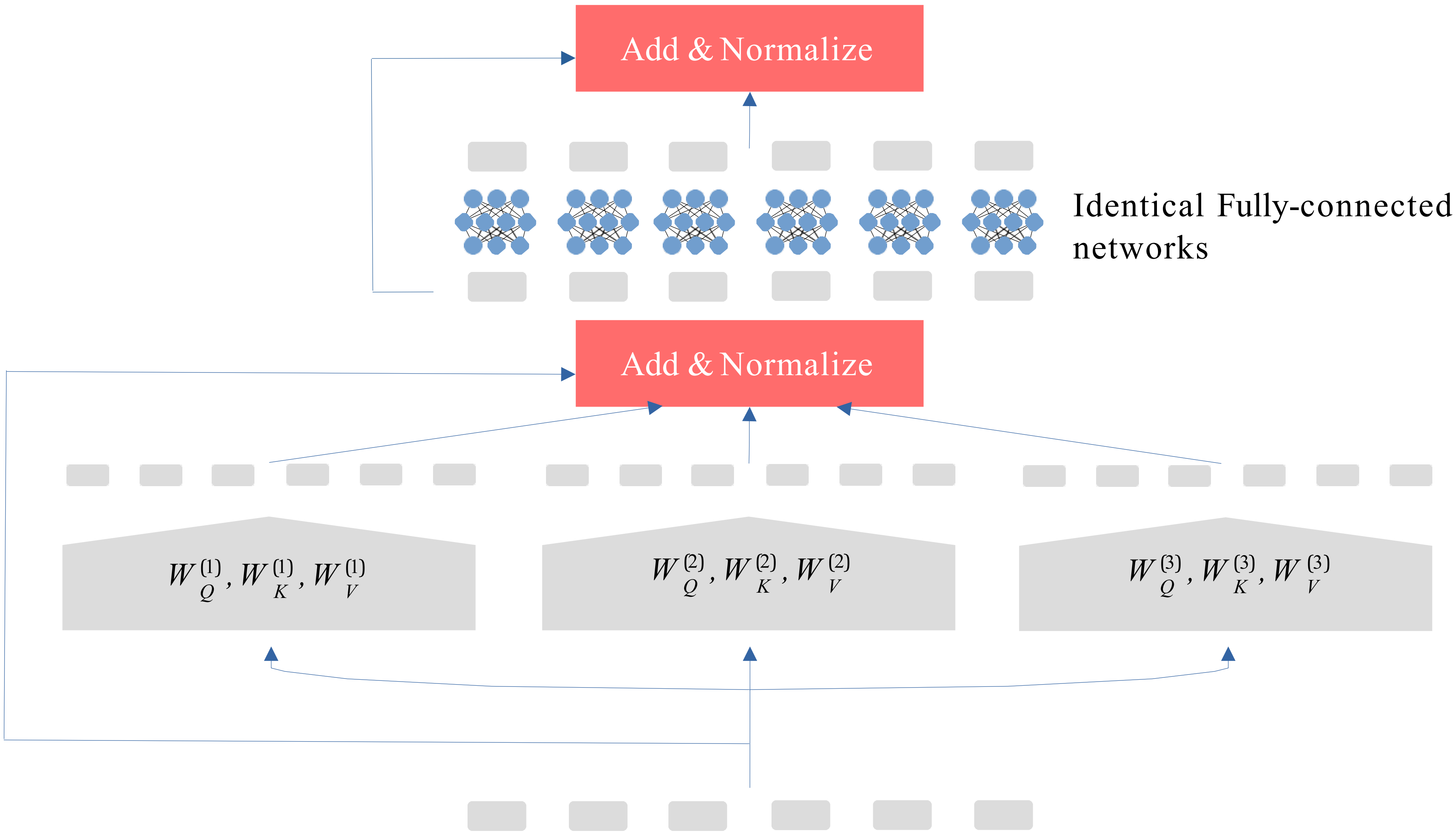
Transformer



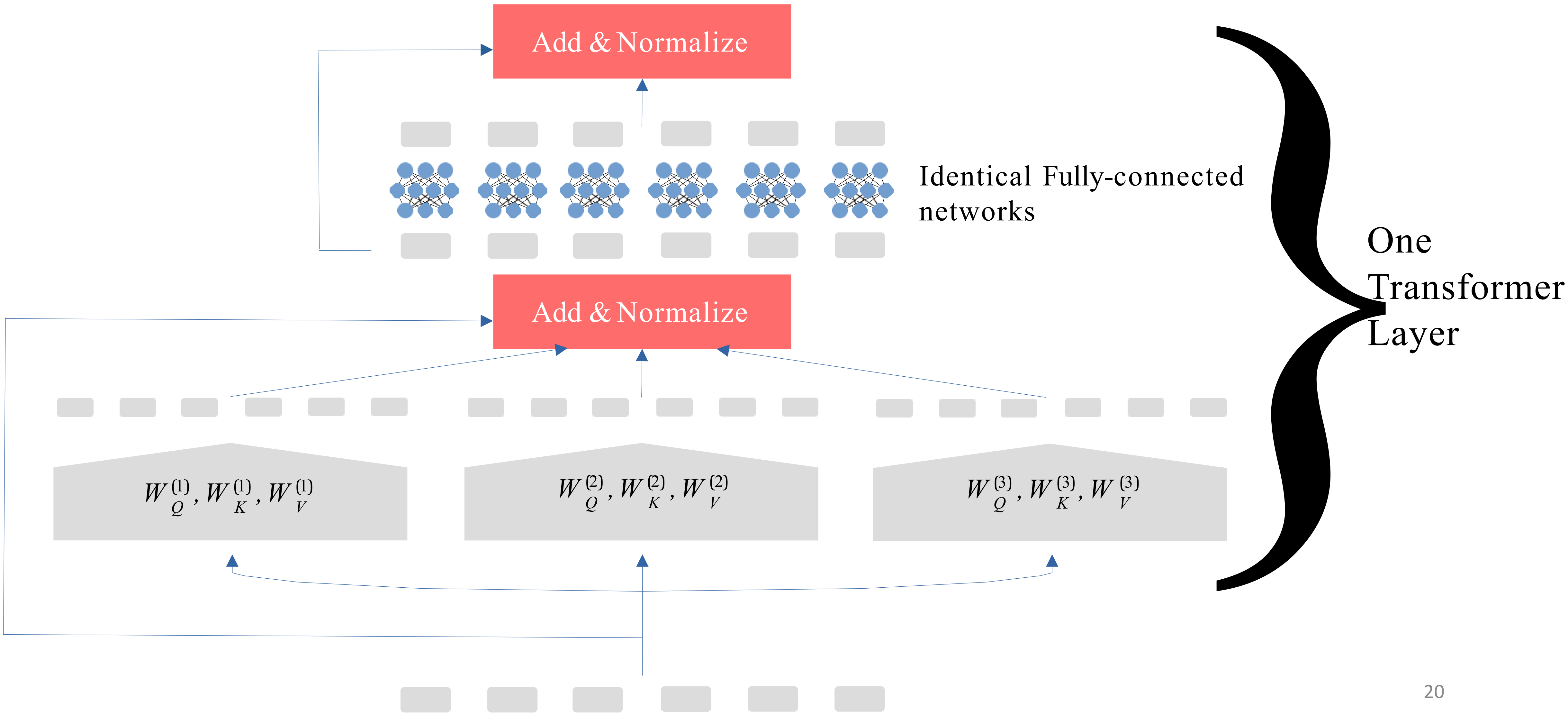
Transformer



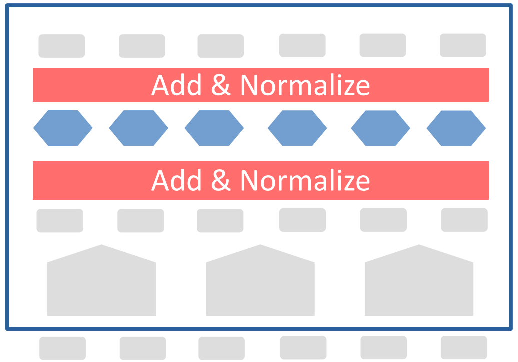
Transformer



Transformer



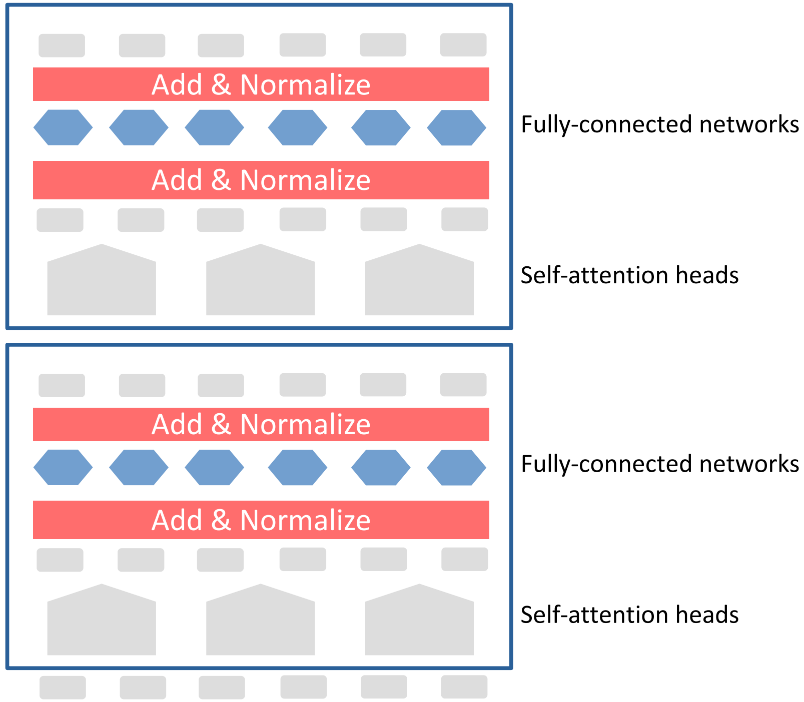
Transformer

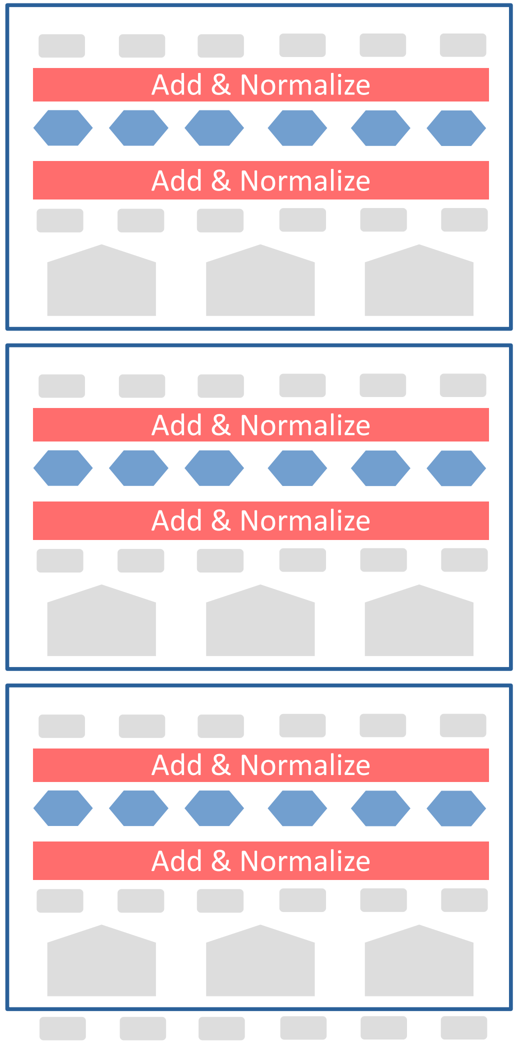


Fully-connected networks

Self-attention heads

Transformer





Fully-connected networks

Self-attention heads

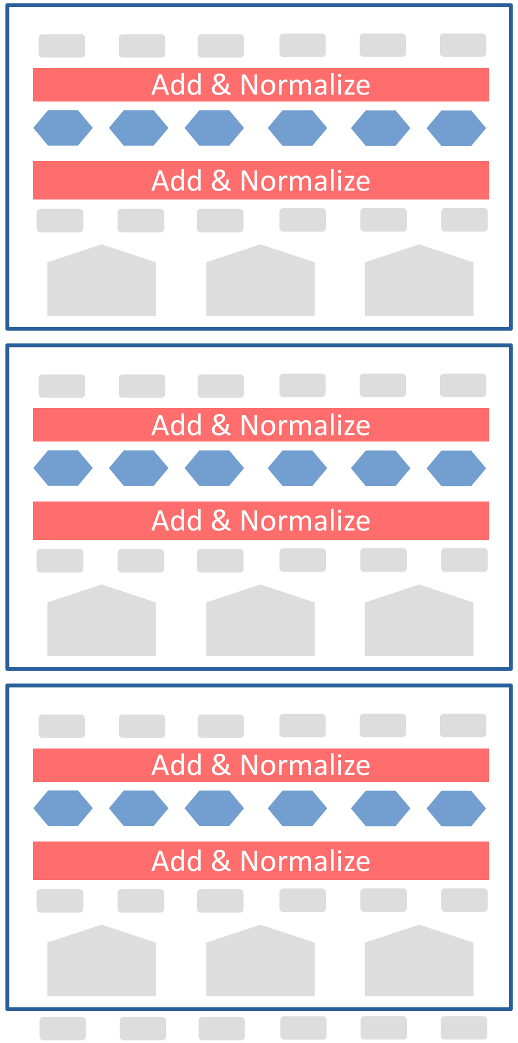
Fully-connected networks

Self-attention heads

Fully-connected networks

Self-attention heads

Transformer



Fully-connected networks

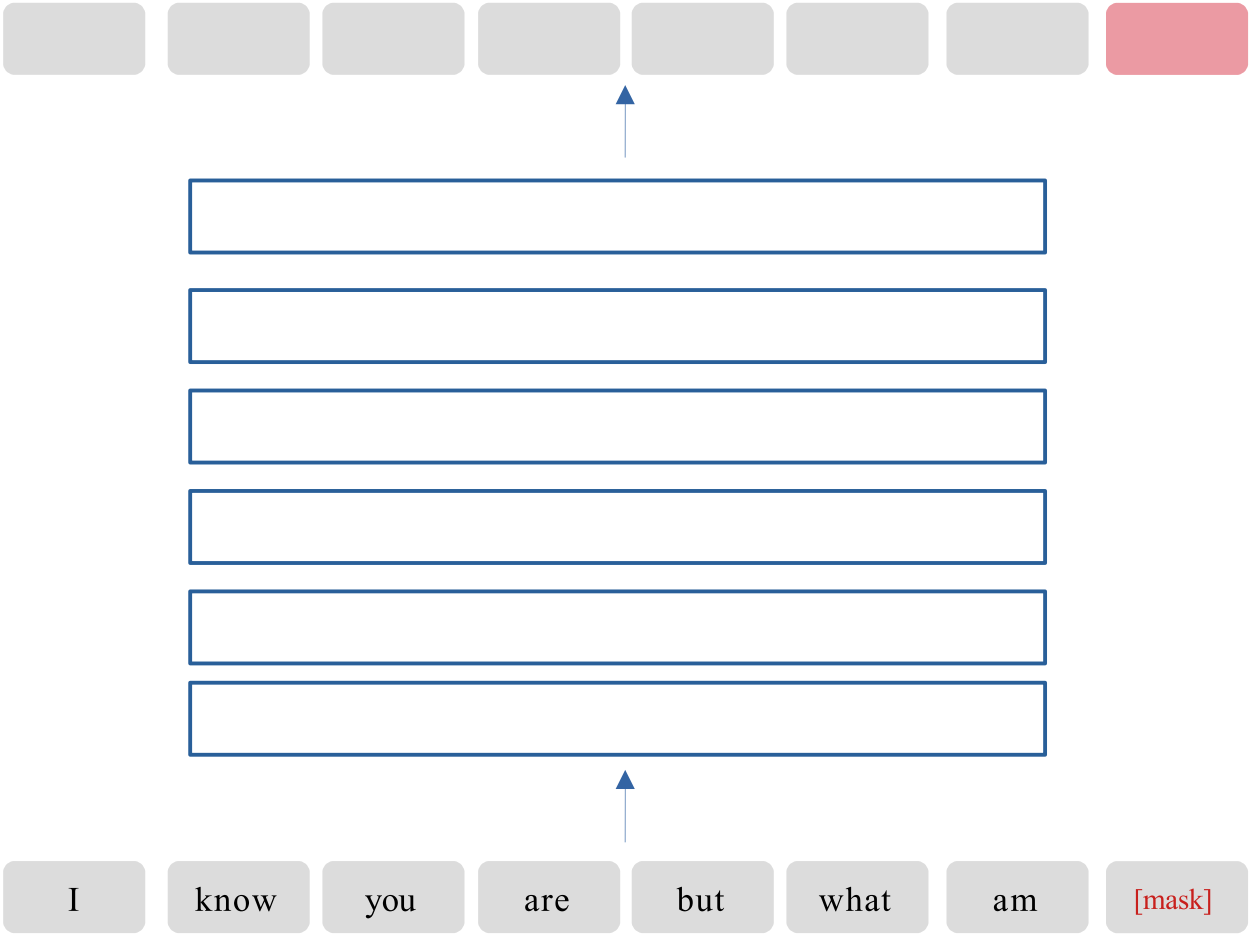
Self-attention heads

Fully-connected networks

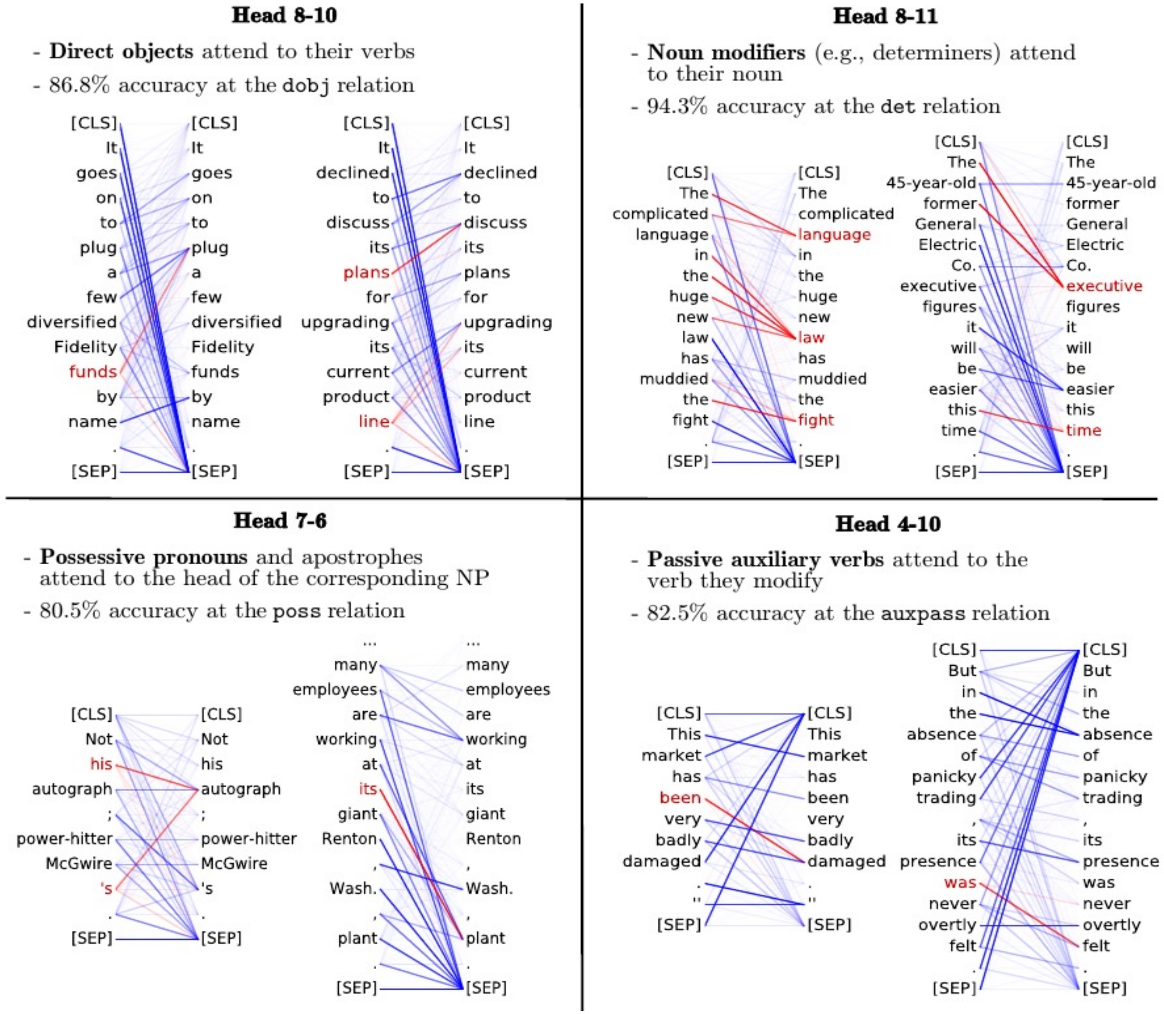
Self-attention heads

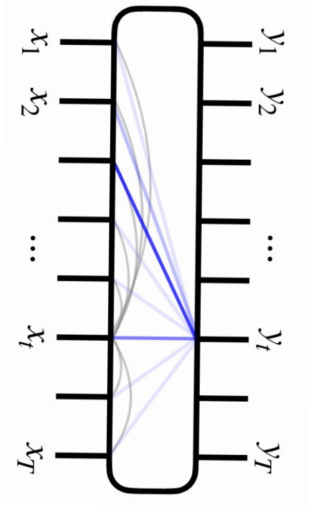
Fully-connected networks

Self-attention heads



Inductive biases of attention



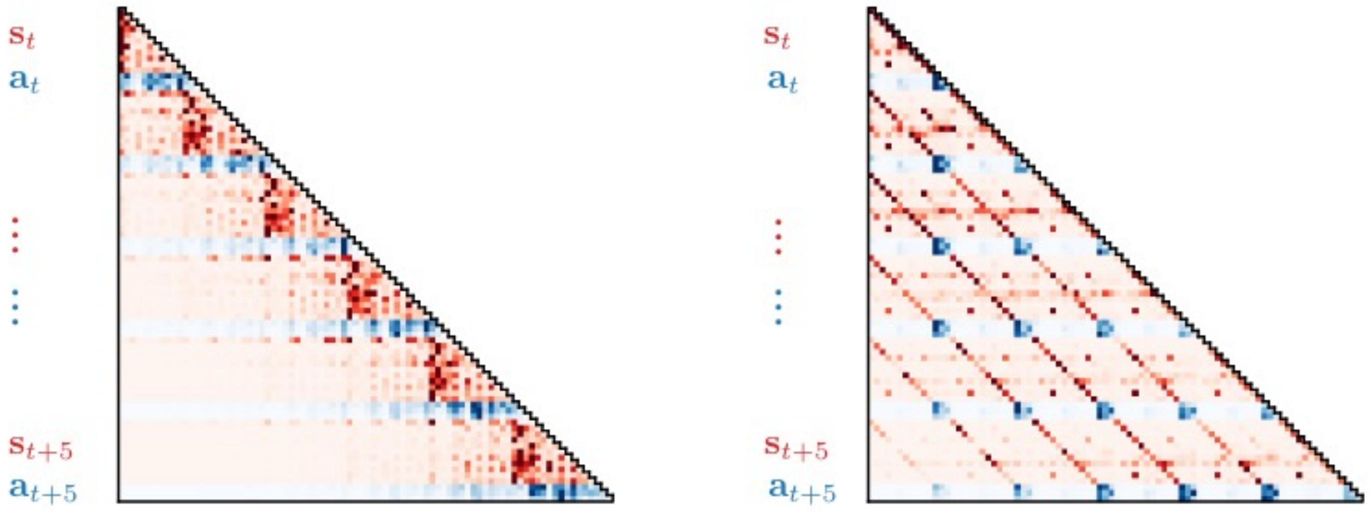


Attention weights are sparse (or close to uniform)

Source: "What Does BERT Look At? An Analysis of BERT's Attention"

Clark, Khandelwal, Levy, Manning, 2019

Inductive biases of attention



Attention weights are sparse
(or close to uniform)

Source: “Offline Reinforcement Learning as One Big Sequence Modeling Problem”
Janner, Li, Levine

Main result: Sparse variable creation

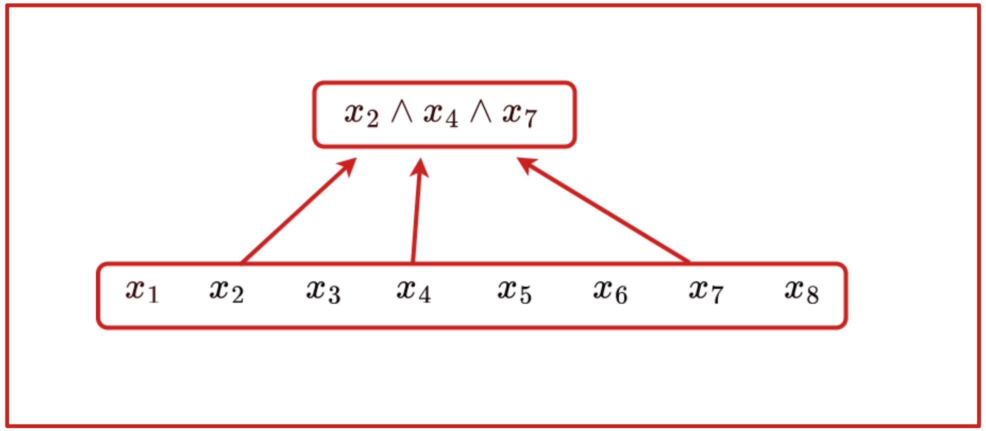
The class of **s-sparse functions of length-T inputs**

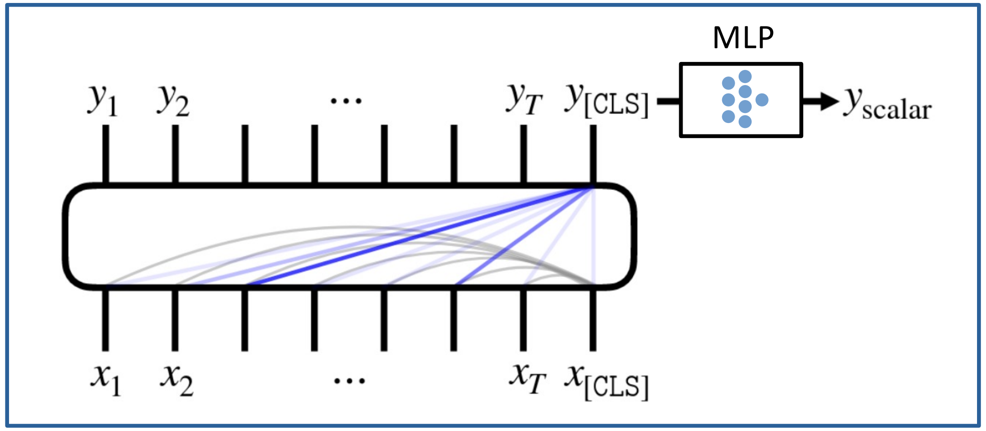
can be learned by

the class of **Transformers layers with weight norms $2^{O(s)}$**

with **sample complexity scaling as $\log(T)$**

optimal





MAIN RESULT - CAPACITY

Result for one-layer Transformers below.
For multi-layer case, there is an exp(spectral norms) factor

Theorem [informal]: *Using covering numbers as the capacity measure*

#samples needed to guarantee uniform convergence $\leq \tilde{O}\left(\frac{\text{poly}(C) \cdot \boxed{\log T}}{\epsilon^2}\right)$

Generalization error

Norm bound on weights
$\| W_V \|_2, \| W_V \|_{2,1},$
$\| W_K W_Q^\top \|_{2,1} \leq C$

Error

Sequence length

Sample complexity like sparse/ℓ_1 regression $\Longrightarrow$ functions not rich in the sequence

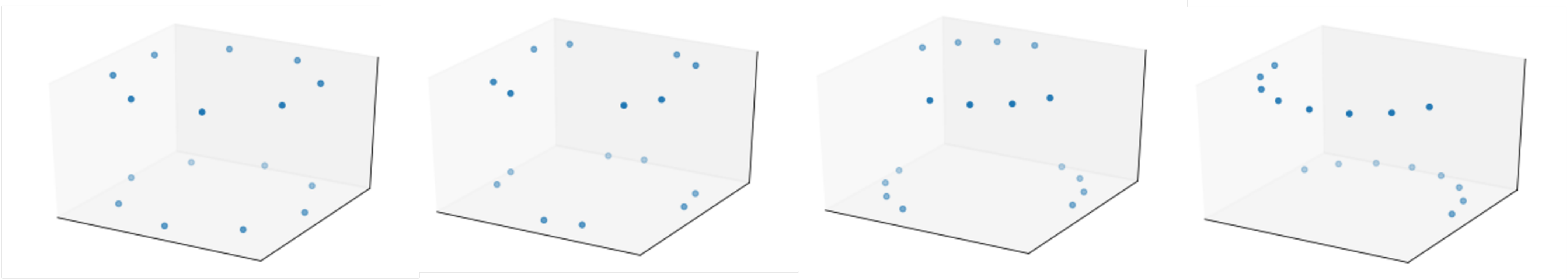
Handle "attention mechanisms" in general: extends to various choices of **ϕ** and **score**

MAIN RESULT - CAPACITY

Theorem [informal]: *Using Pseudo-dimension as the capacity measure*

Even for $d = 3$, unbounded norm attention heads require $\Omega(\log T)$ samples to guarantee uniform convergence.

Capacity larger than the number of parameters $O(d^2)$!



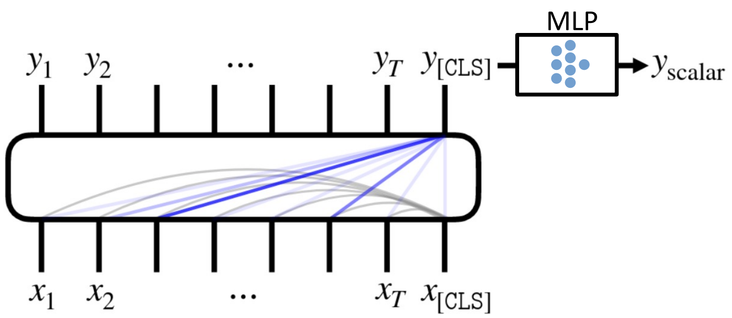
4 points that are shattered for $T = 16$

Main result - Expressivity

Any s-sparse Boolean function f can be exactly represented by a Transformer layer with weight norms $2^{O(s)}$.

If f is symmetric, only poly(s) weight norms are required.

Intuition
- `Softmax` allows sparse variable selection
- MLP allows arbitrary function to be applied



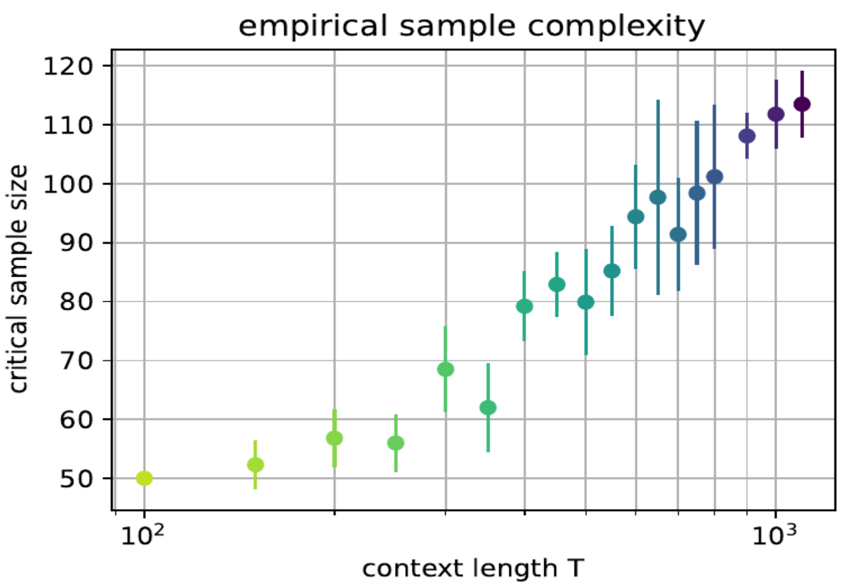
Optimization (sparse conjunctions)

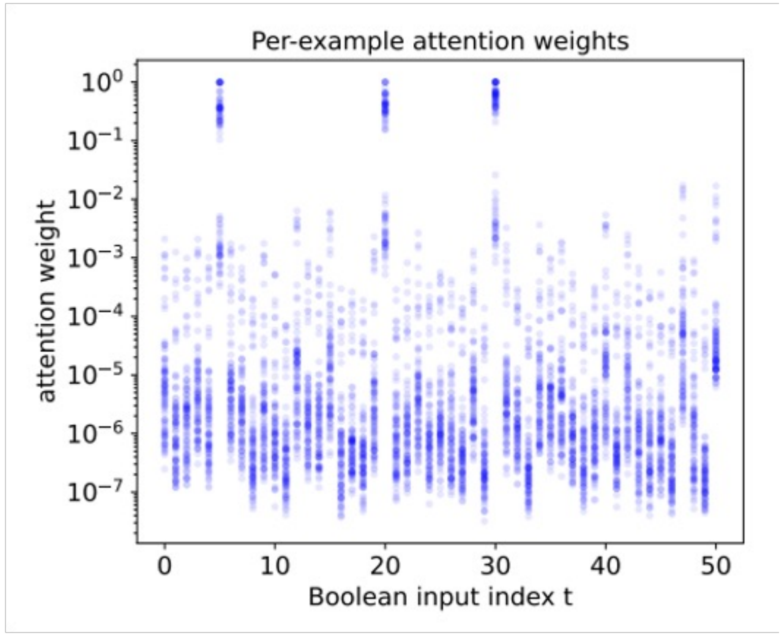
$$\vec{x} \sim \text{unif}(\{0,1\}^T)$$
$$y = x_{i_1} x_{i_2} x_{i_3}$$

Train a one-layer Transformer

- As input length T grows, how large does the training set need to be to avoid overfitting?

- Consistent with $\log(T)$ dependence in generalization bound!





Part 1 Recap

- Loose ends
  - capacity upper/lower bounds aren’t tight
  - Don’t know how to handle trainable positional encodings
- Low-norm Transformers ≈ simple circuits
  - What is the right circuit class for capturing the inductive bias of Transformers?
- Optimization!

Part 2

Hidden Progress in Deep Learning: SGD Learns Parities Near the Computational Limit, NeurIPS ‘22

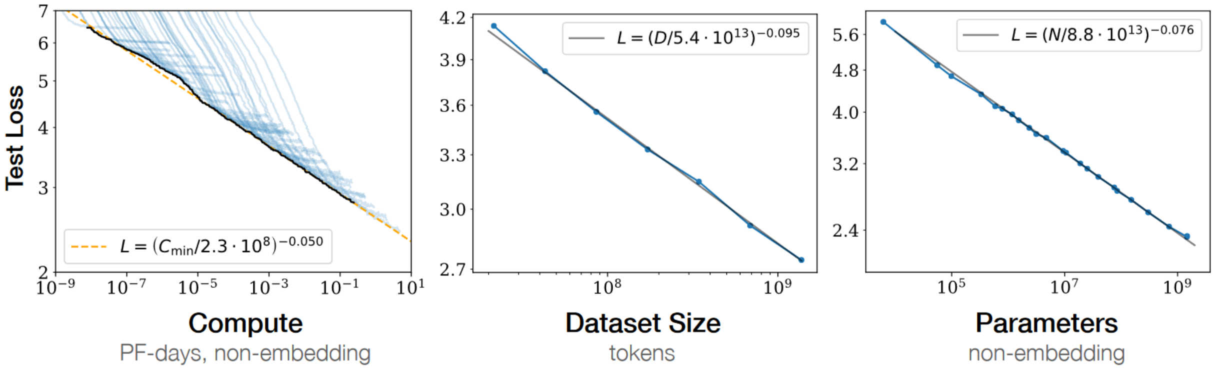
with Boaz Barak, Surbhi Goel, Sham Kakade, Eran Malach, & Cyril Zhang

"People really enjoying a machine learning seminar", painting by Pablo Picasso

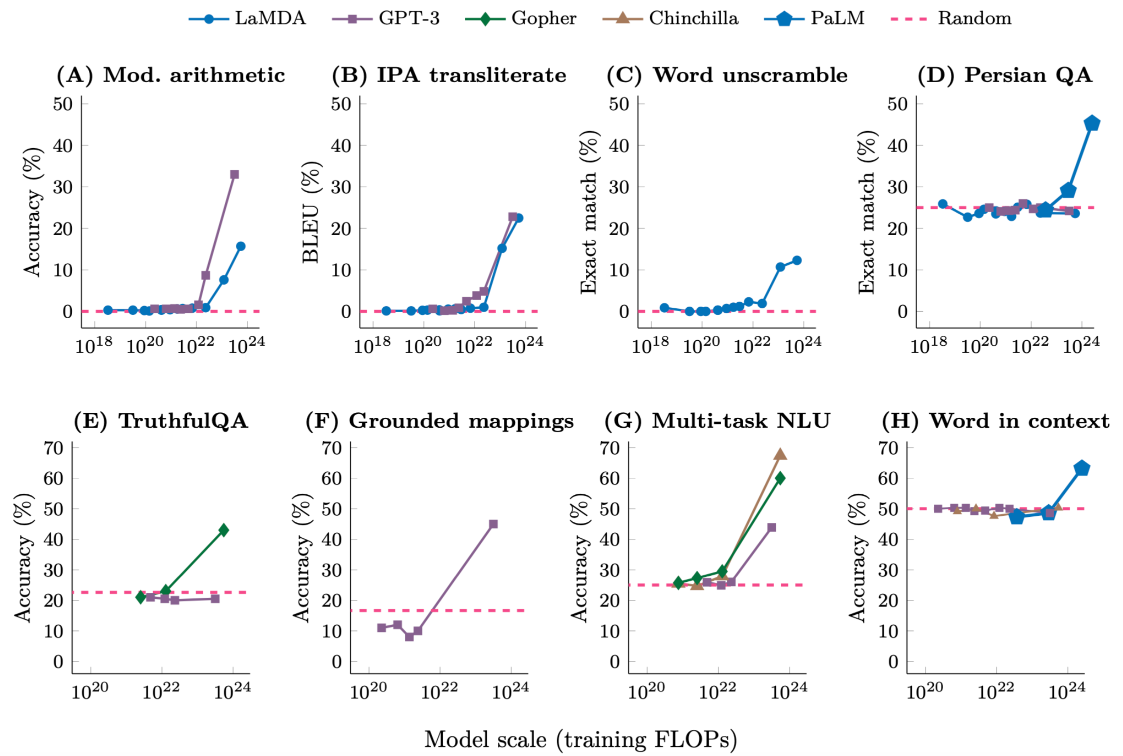
Mysteries of contemporary deep learning

1. How do neural networks learn to construct useful features?
2. How do neural networks learn to “reason” / compute “combinatorial” functions?
3. Why are there sometimes emergent breakthroughs in capabilities as resources are scaled up?

3. Emergence

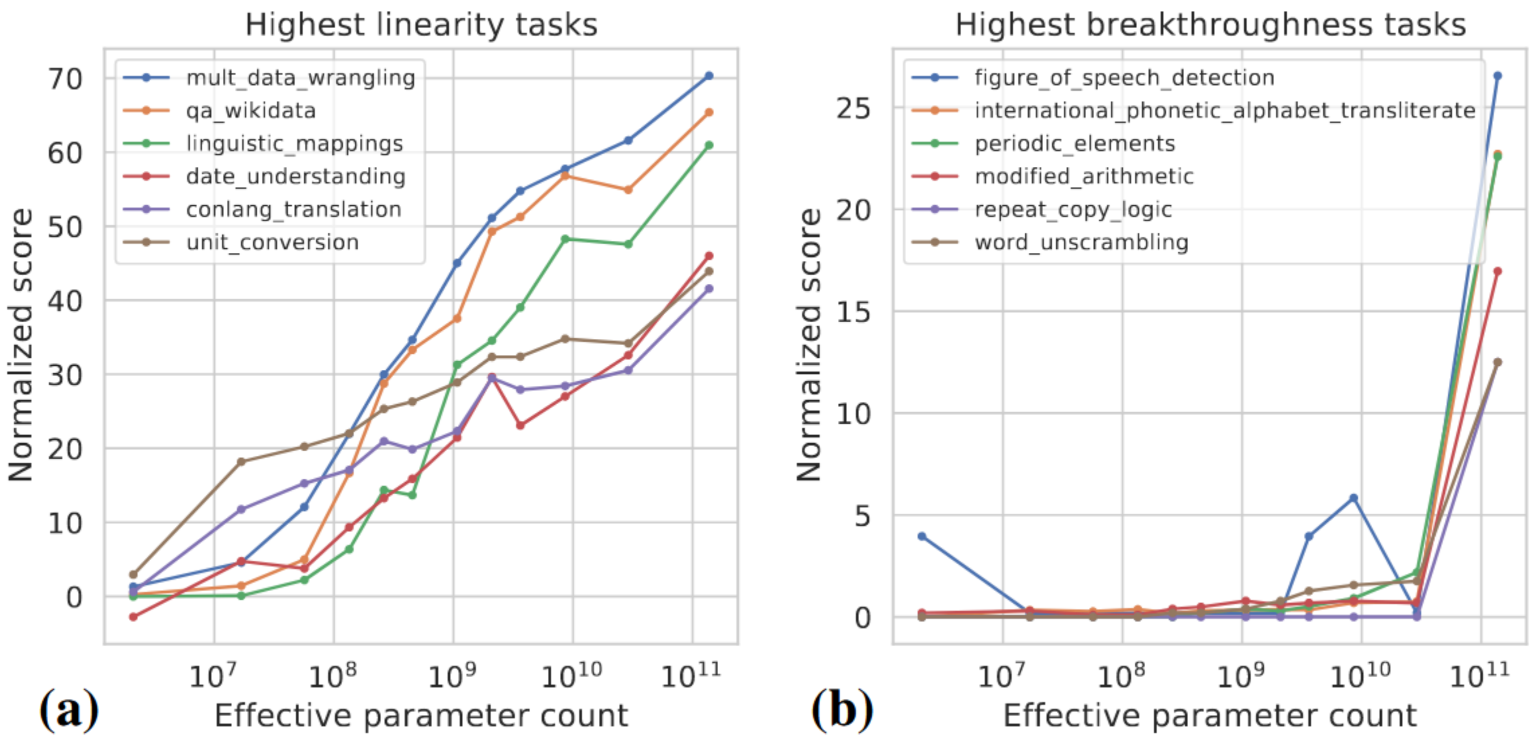


Scaling Laws for Neural Language Models
Kaplan et al. 2020



Emergent Abilities of Large Language Models
Wei et al. 2022

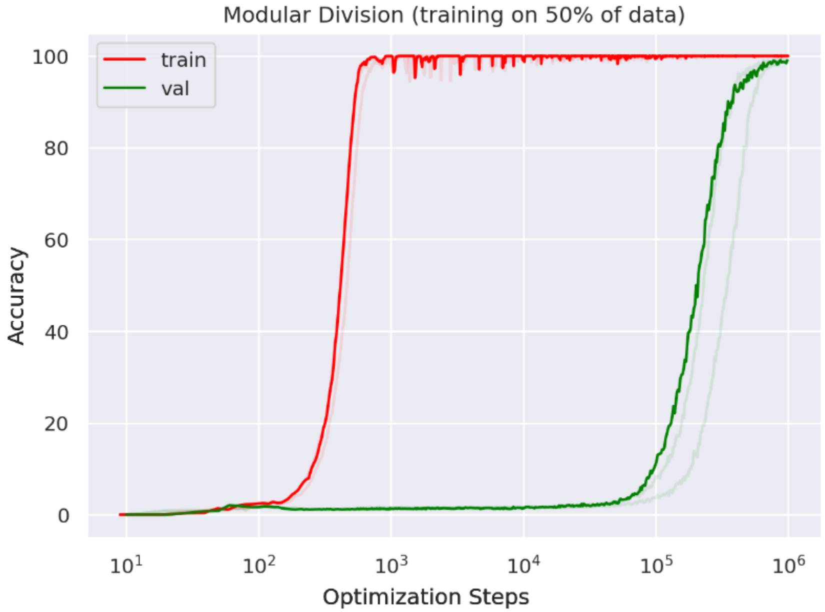
3. Emergence



Beyond the Imitation Game: Quantifying and extrapolating the capabilities of language models

435 authors 2022

3. Emergence



Grokking: Generalization Beyond Overfitting on Small Algorithmic Datasets
Power et al., 2022

Mysteries of contemporary deep learning

1. How do neural networks learn to construct useful features?
2. How do neural networks learn to “reason” / compute “combinatorial” functions?
3. Why are there sometimes emergent breakthroughs in capabilities as resources are scaled up?

Our approach

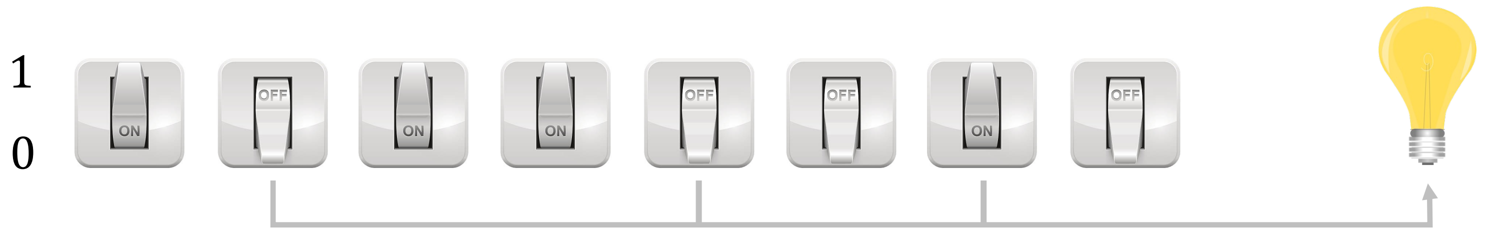
💡 Analyze a single synthetic task that exhibits these mysteries

Hidden Progress in Deep Learning: SGD Learns Parities Near the Computational Limit

Joint work with Boaz Barak, Surbhi Goel, Sham Kakade, Eran Malach, and Cyril Zhang

Learning sparse parities

Parity function $\chi_S : \{0,1\}^n \to \{0,1\}$:



$$\chi_S(x) = \sum_{i \in S} x_i \bmod 2$$

k**-way Boolean XOR**

k**-sparse parity learning problem:** given samples $(x, y) \sim \mathcal{D}_S$, recover k indices S

$\mathcal{D}_S$:

[0	1	0	0	1	1	0	1]	,	0
[0	1	1	0	1	0	1	1]	,	1
[1	0	0	0	0	0	1	0]	,	1

…

$x \sim \mathrm{Unif}(\{0,1\}^n)$ $\quad y = \chi_S(x)$

How many samples are needed to learn?

Key fact:
Parity functions are uncorrelated.
For $S \neq S'$,

$$\Pr_x[\chi_S(x) = \chi_{S'}(x)] = 1/2$$

Proof: First show for $S \cap S' = \emptyset$

k **–sparse parity function** $|S| = k,\ \chi_S : \{0,1\}^n \to \{0,1\}$

$$\chi_S(x) = \sum_{i \in S} x_i \bmod 2$$

$\mathcal{D}_S$: $x \sim \mathrm{Unif}(\{0,1\}^n)$ $y = \chi_S(x)$

Theorem: $O(k \log n)$ samples are needed.
Proof:
Suppose we draw a training set of m samples labeled by χ_S. Consider any $S' \neq S$.
Q: What's the probability that $\chi_{S'}$ is consistent with the training set?
A: $\left(\frac{1}{2}\right)^m$
Q: What's the probability that there exists *any* k –sparse parity function besides χ_S that is consistent with the training set?
A: $O\left(n^k \left(\frac{1}{2}\right)^m\right)$

How efficiently can we learn?

Theorem: $O(k \log n)$ samples are needed.

That's pretty sample-efficient!

But what about *computational efficiency*?

k **–sparse parity function** $|S| = k$, $\chi_S : \{0,1\}^n \to \{0,1\}$

$$\chi_S(x) = \sum_{i \in S} x_i \bmod 2$$

$\mathcal{D}_S$: $x \sim \mathrm{Unif}(\{0,1\}^n)$ $y = \chi_S(x)$

Computational barriers

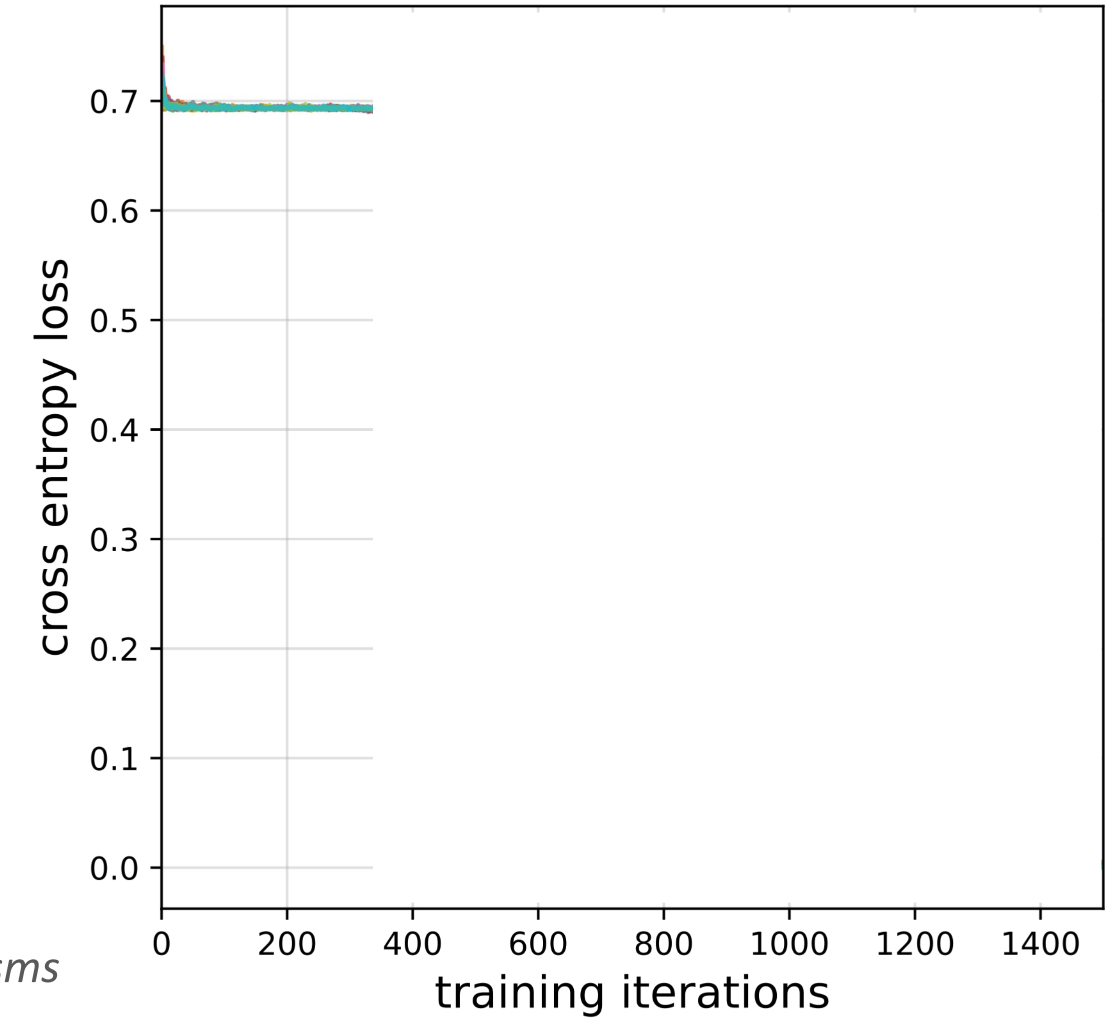
- Fastest-known algorithm for learning sparse parities using $O(k \log n)$ samples: $n^{k/2}$ running time (credited to Spielman in Klivans & Servedio 2006)
- Regardless of # samples, gradient descent on any neural network requires $n^{\Omega(k)}$ batch size or iterations (Abbe, Kamath, Malach, Sandon, Srebro 2021). Based on statistical query lower bound
- An important cryptography conjecture states: if training set labels are flipped with small constant probability, any algorithm requires $n^{\Omega(k)}$ running time (originally due to Alekhnovich 2003)

What happens when we throw deep learning at the problem?

$n = 15, k = 3$

- Train a one layer Transformer with online SGD

Note: prior works show neural networks can learn parities under assumptions on input distribution (Daniely and Malach, 2020, Frei et al., 2022, Malach et al., 2021, Shi et al., 2021)

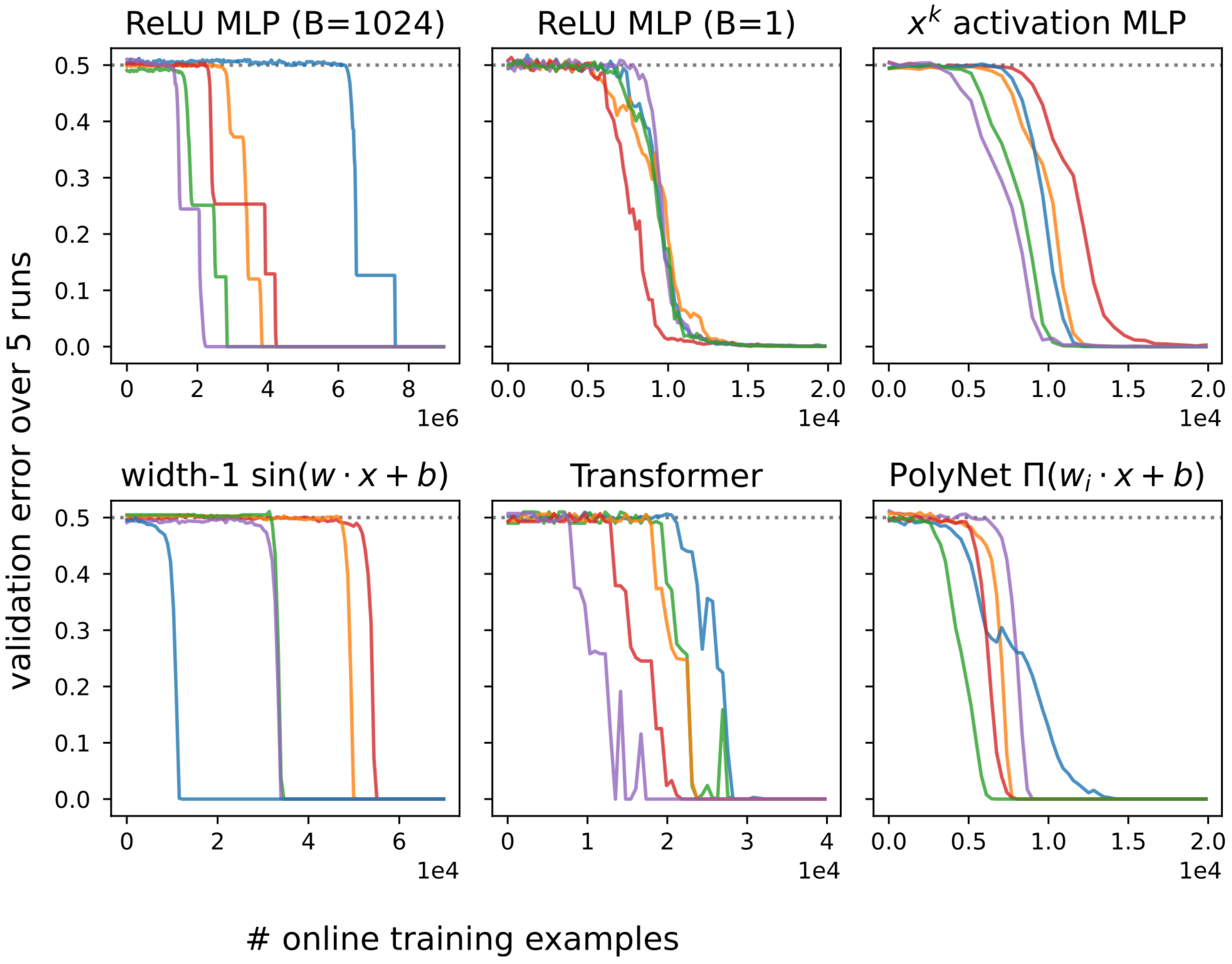


Inductive Biases and Variable Creation in Self-Attention Mechanisms
E, Surbhi Goel, Sham Kakade, and Cyril Zhang 2022

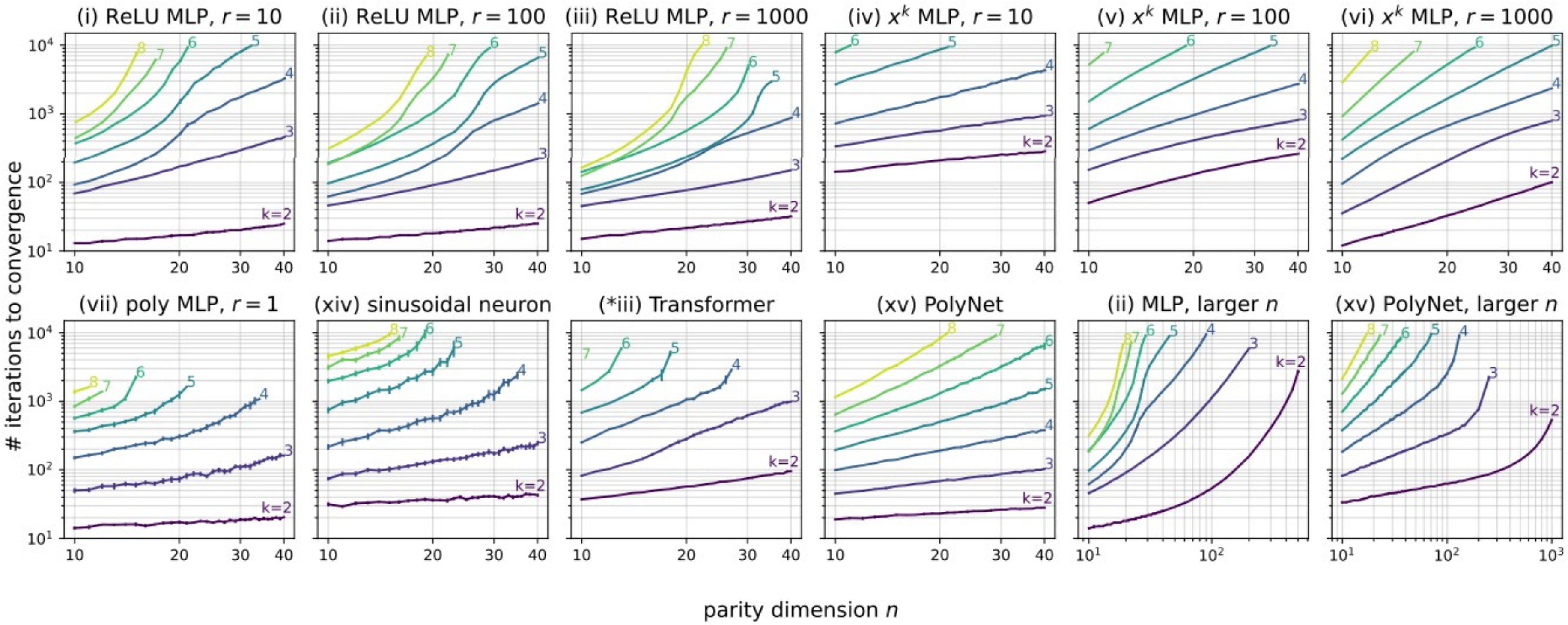
What happens when we throw deep learning at the problem?

: $n = 50, k = 3$

- One hidden-layer (width$= 100$) ReLU MLP
- One hidden-layer (width$= 100$) $a \mapsto a^k$ MLP
- Sinusoidal neuron $x \mapsto \sin(w^\top x)$
- One layer Transformer
- PolyNet: $x \mapsto \prod_{i=1}^{k} (w_i^\top x)$

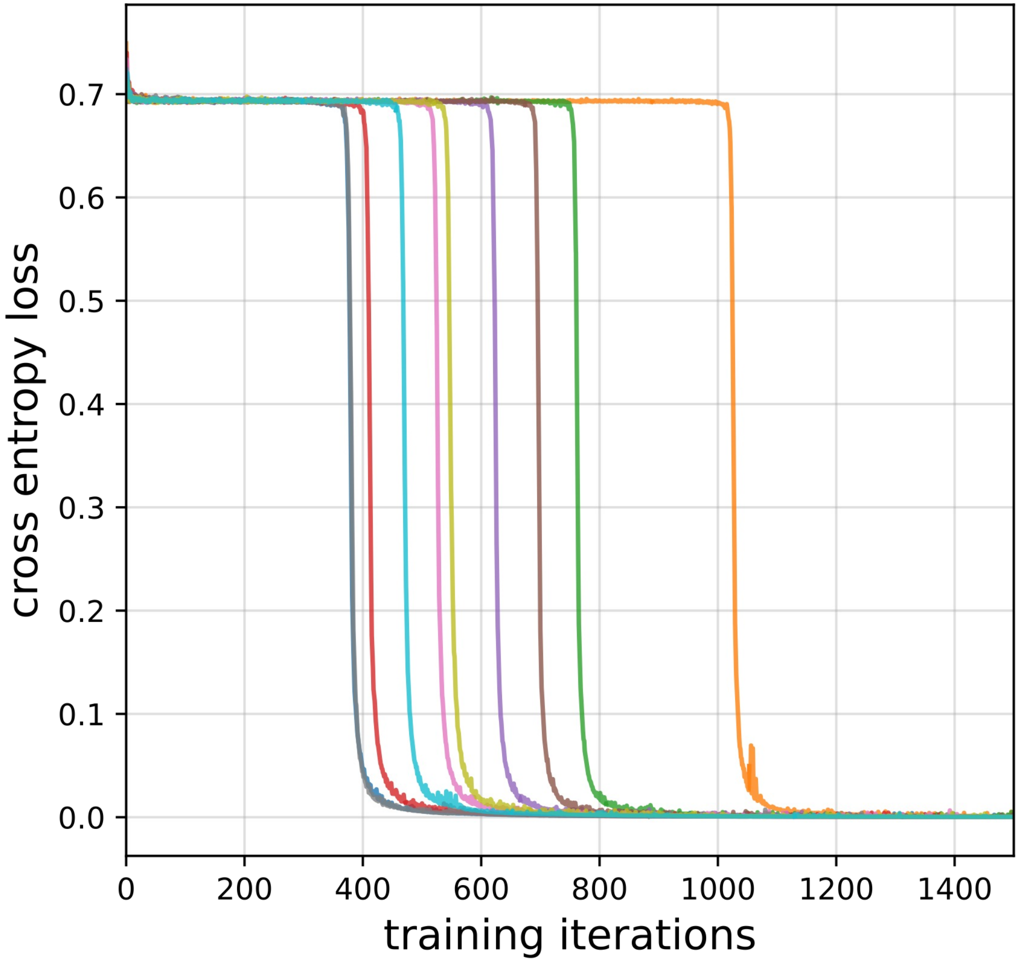


What happens when we throw deep learning at the problem?



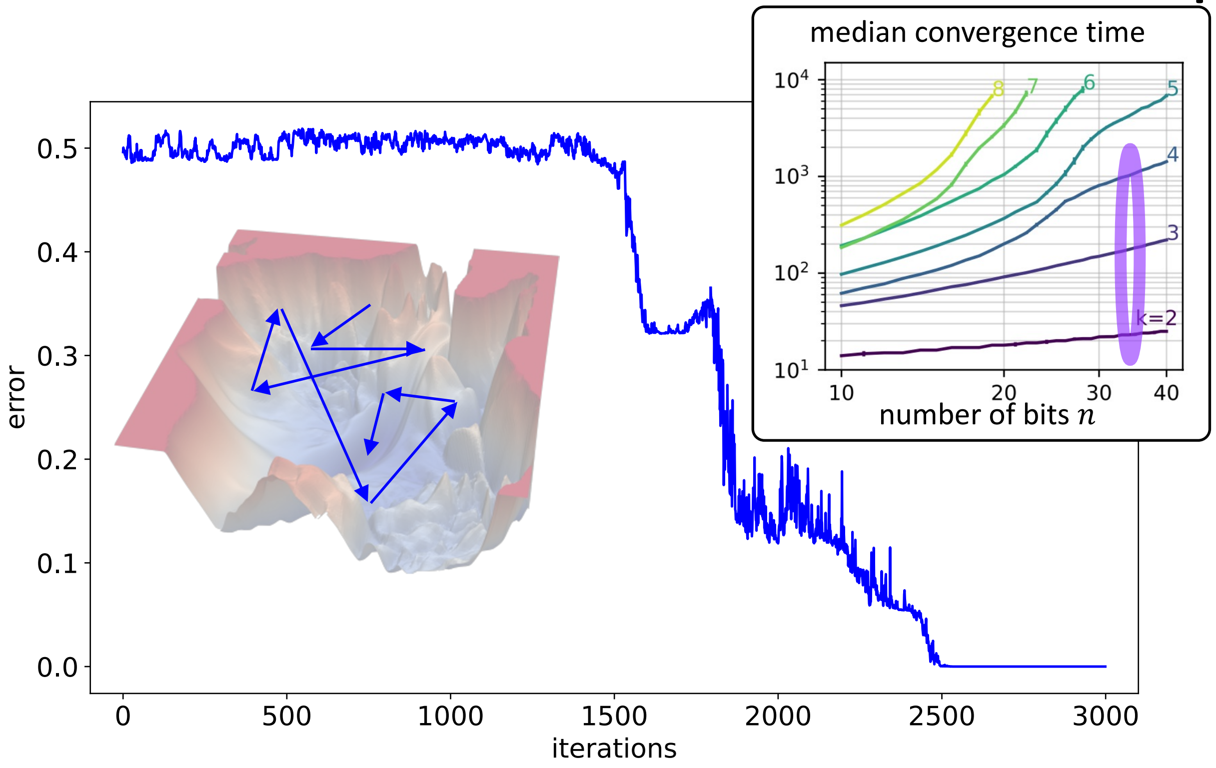
- Across a wide range of architectures/initializations/batch sizes, SGD on neural networks learns sparse parities; for small instances, # iterations looks like $n^{O(k)}$

What's the mechanism behind the breakthrough: Implicit random search? (or hidden progress???)



Note: the network does need to learn features---if we used linear classification on fixed features (i.e. kernel methods), the number of features would need to be $n^{\Omega(k)}$.

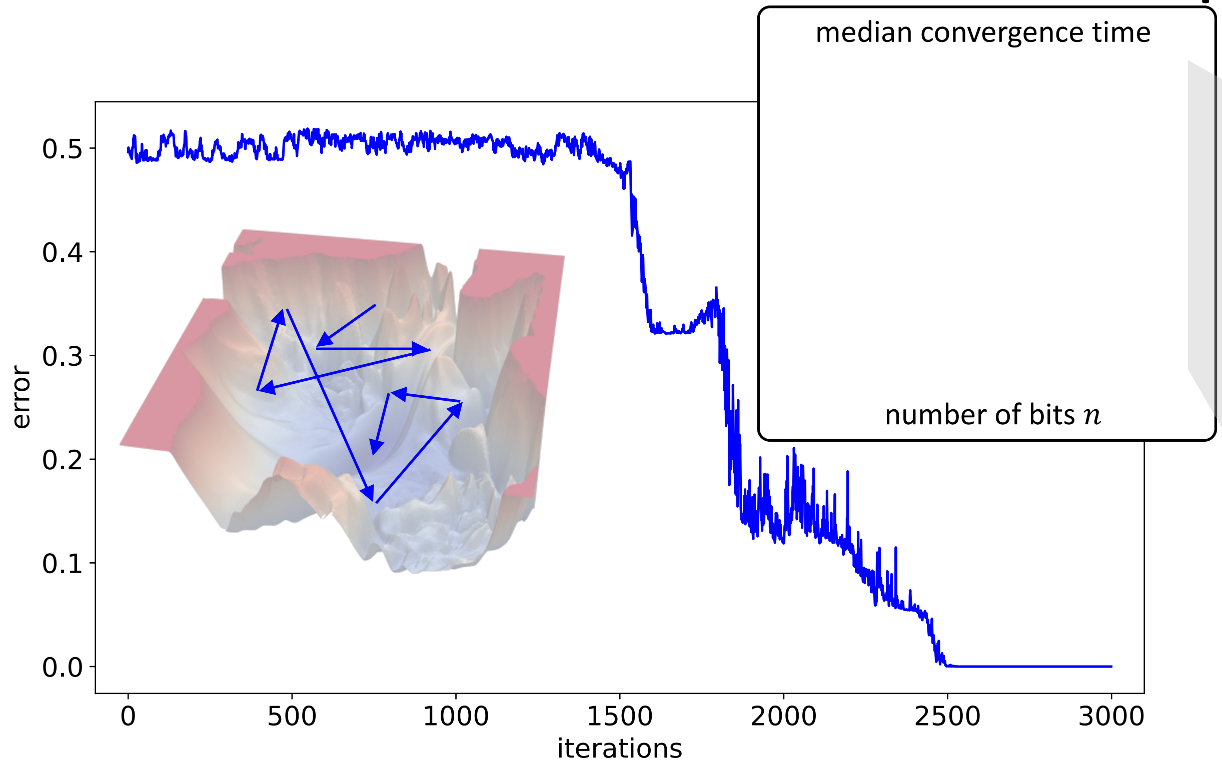
How does SGD learn sparse parities?

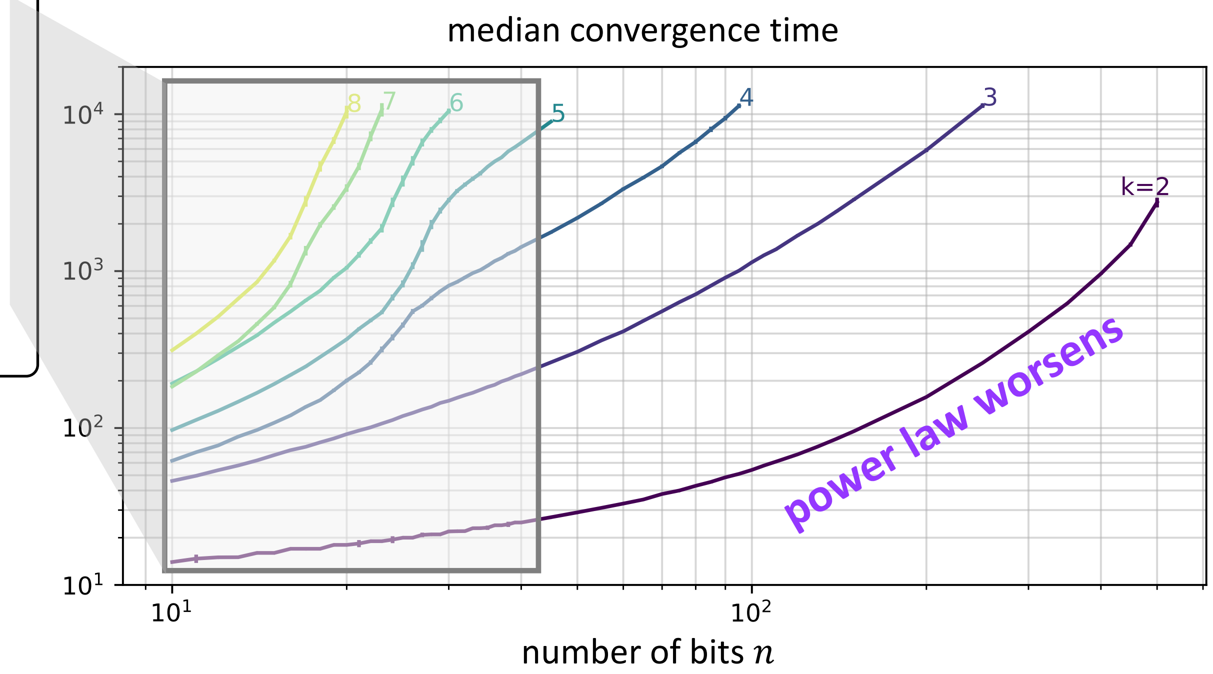


same architecture
same algorithm
different behavior for each k

- **Hypothesis:** randomness of SGD implements random search (à la Langevin)
  - **…how does the network adapt to the sparsity k?**

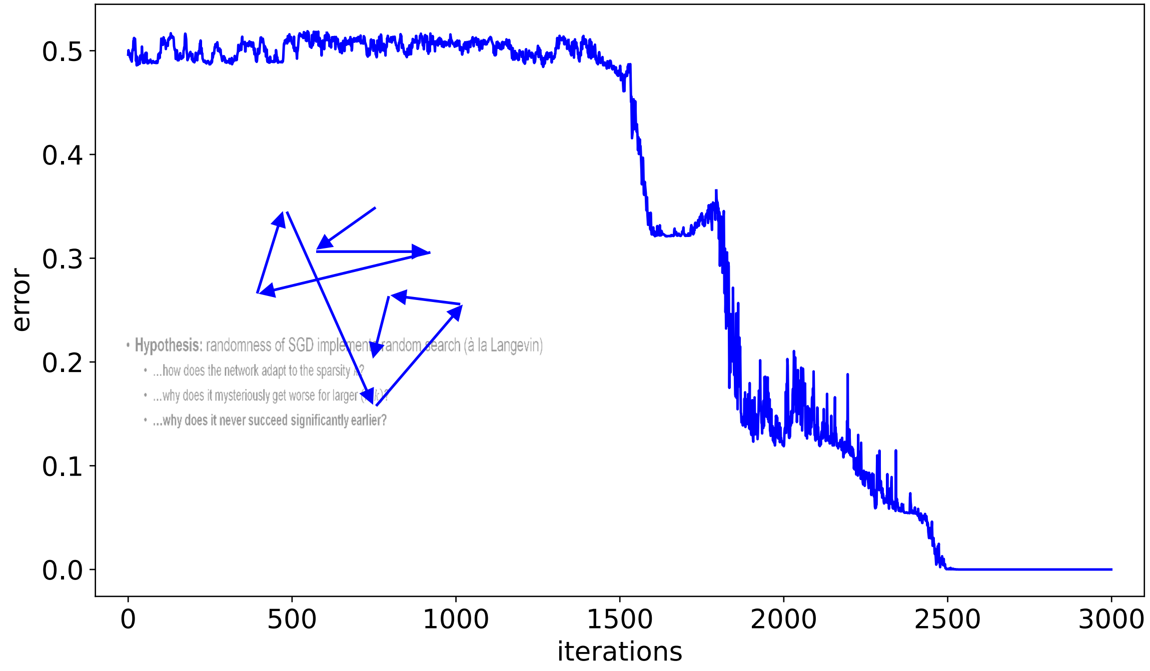
How does SGD learn sparse parities?

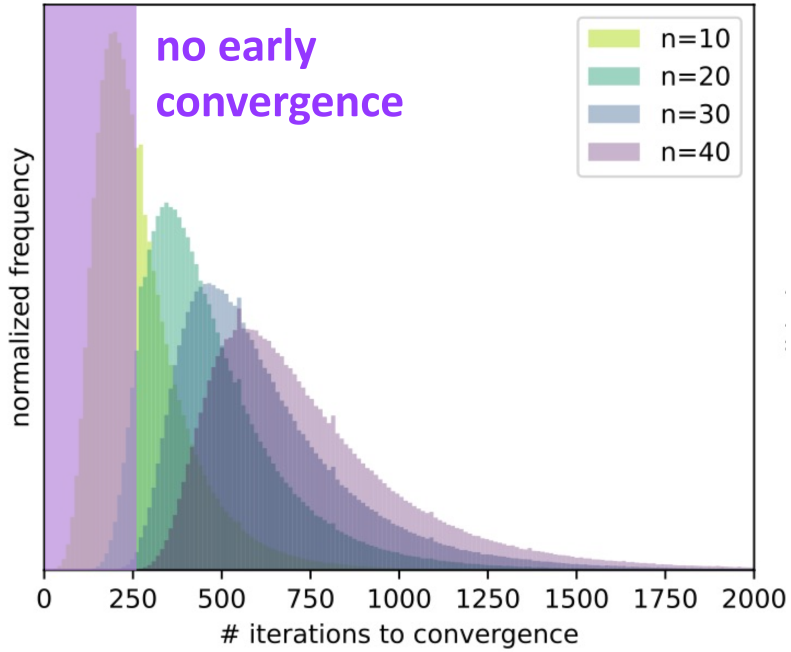




- **Hypothesis:** randomness of SGD implements random search (à la Langevin)
  - …how does the network adapt to the sparsity k?
  - **…why does it mysteriously get worse for larger** (n, k)**?**

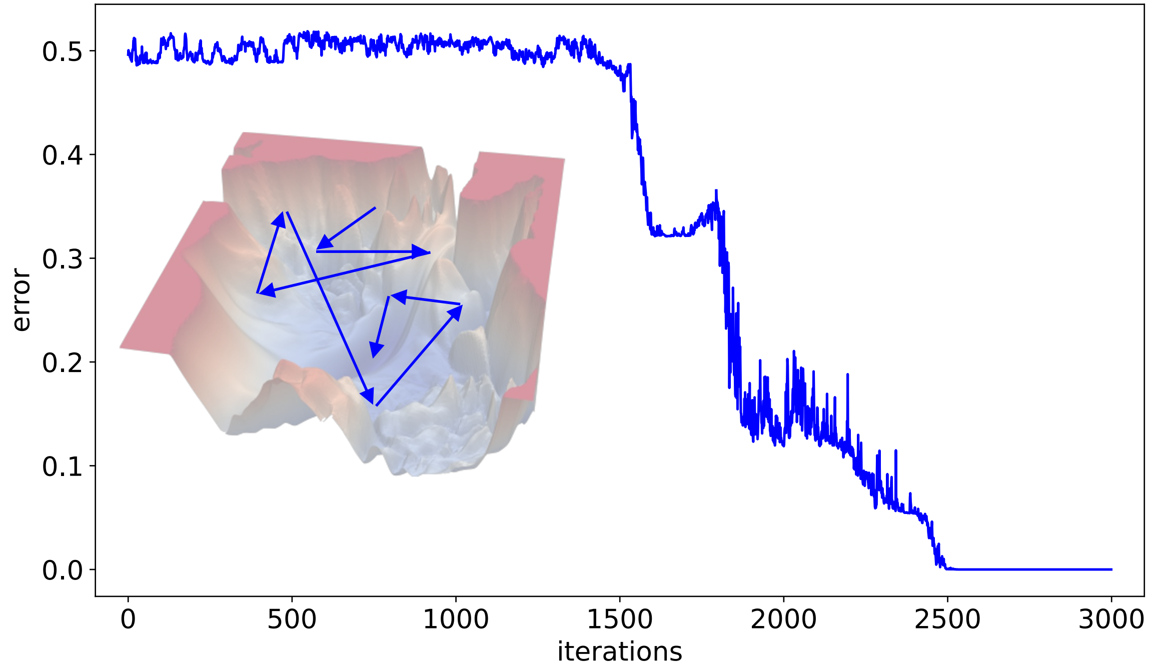
How does SGD learn sparse parities?

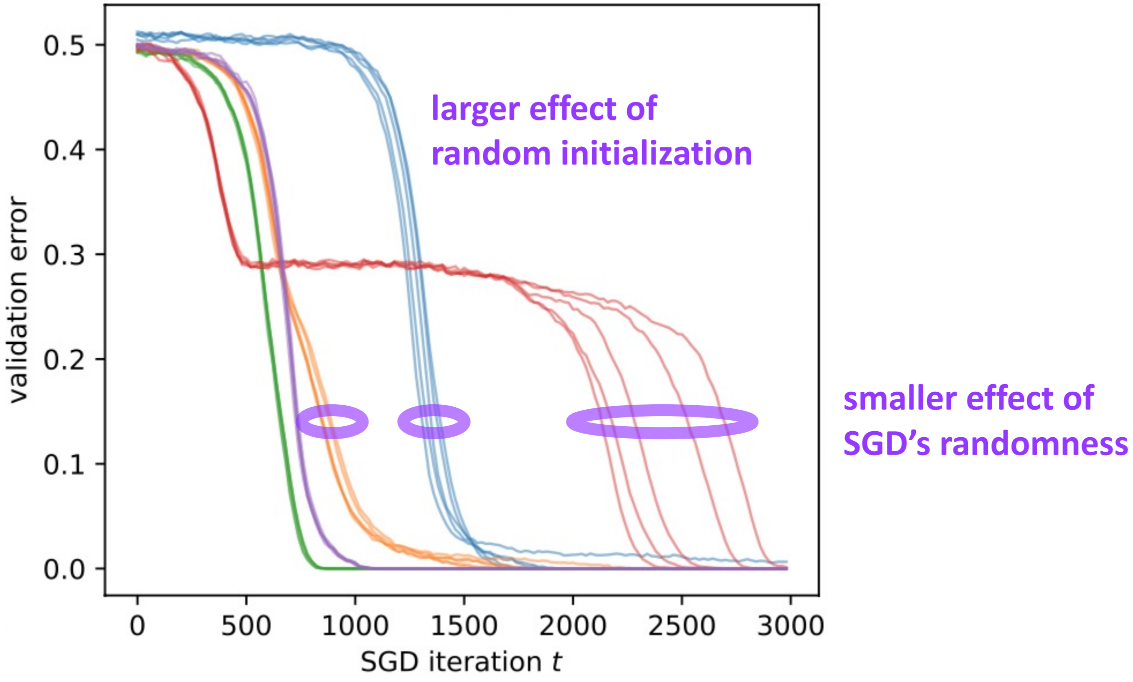




- **Hypothesis:** randomness of SGD implements random search (à la Langevin)
  - …how does the network adapt to the sparsity k?
  - …why does it mysteriously get worse for larger (n, k)?
  - **…why does it never succeed significantly earlier?**

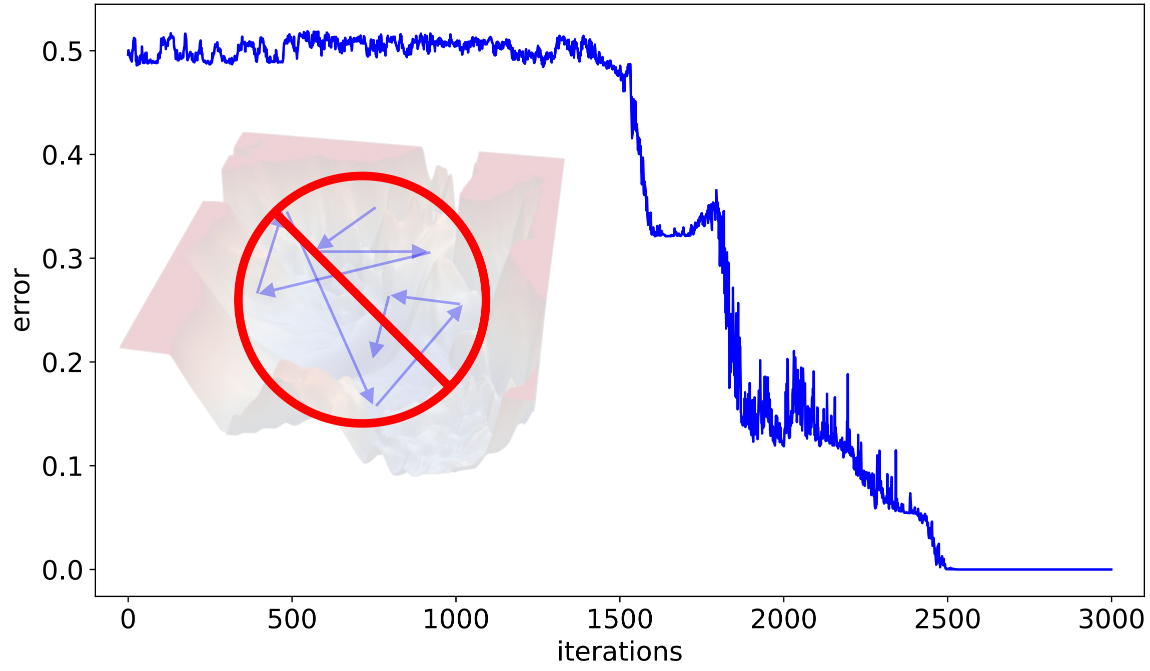
How does SGD learn sparse parities?

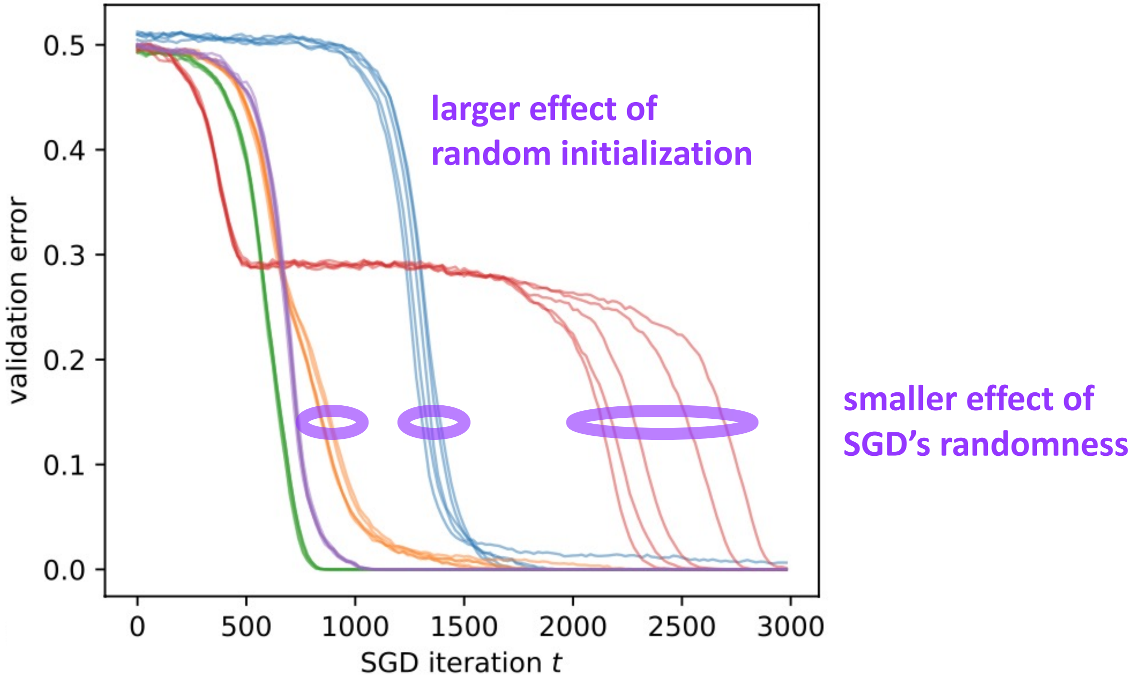




- **Hypothesis:** randomness of SGD implements random search (à la Langevin)
  - …how does the network adapt to the sparsity k?
  - …why does it mysteriously get worse for larger (n, k)?
  - …why does it never succeed significantly earlier?
  - **…why does its trajectory depend heavily on initialization?**

How does SGD learn sparse parities?

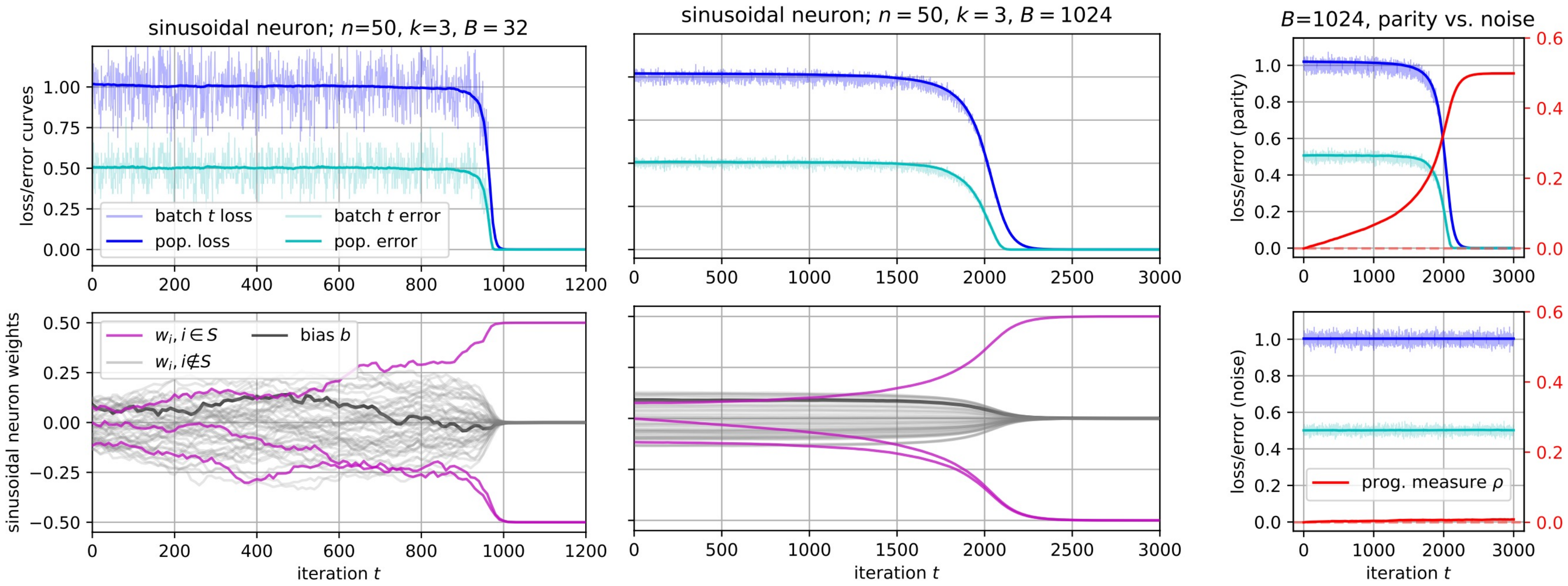




- ~~**Hypothesis:** randomness of SGD implements random search (à la Langevin)~~
  - …how does the network adapt to the sparsity k?
  - …why does it mysteriously get worse for larger (n, k)?
  - …why does it never succeed significantly earlier?
  - **…why does its trajectory depend heavily on initialization?**

It's hidden progress

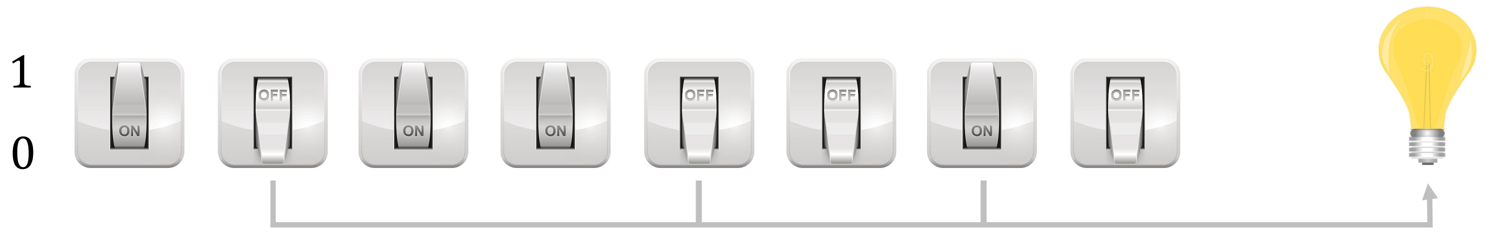
Hidden progress measure: a function of the training algorithm's state which is predictive of the time to convergence and continuously improves throughout training



Progress measure based on drift term $\|w^{(t)} - w^{(0)}\|_\infty$

Learning sparse parities

Parity function $\chi_S : \{0,1\}^n \to \{0,1\}$:



$\chi_S(x) = \sum_{i \in S} x_i \bmod 2$

k**-way Boolean XOR**

Parity learning problem: given samples $(x, y) \sim \mathcal{D}_S$, recover k indices S

$\mathcal{D}_S$:

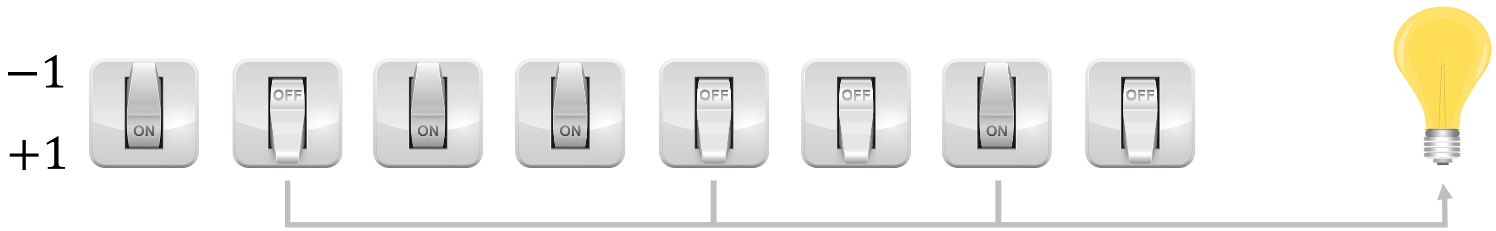
[0	1	0	0	1	1	0	1] ,	0
[0	1	1	0	1	0	1	1] ,	1
[1	0	0	0	0	0	1	0] ,	1

…

$x \sim \mathrm{Unif}(\{0,1\}^n)$ $\qquad$ $y = \chi_S(x)$

Learning sparse parities

Parity function $\chi_S : \{\pm 1\}^n \to \ \pm 1$:



$\chi_S(x) = \Pi_{i \in S} x_i$

k**-way Boolean XOR / degree-**
k **monomial**

Parity learning problem: given samples $(x, y) \sim \mathcal{D}_S$, recover k indices S

$\mathcal{D}_S$:

```
[+1  -1  +1  +1  -1  -1  +1  -1] ,   +1
[+1  -1  -1  +1  -1  +1  -1  -1] ,   -1
[-1  +1  +1  +1  +1  +1  -1  +1] ,   -1
               ...
```

$x \sim \text{Unif}(\{\pm 1\}^n)$ $\quad$ $y = \chi_S(x)$

KEY IDEA - INFORMATION IN THE GRADIENT AT STEP 1

Assume single ReLU with correlation loss $\mathbb{E}[-y\sigma(w^\top x)]$, and initialize $w = [1, \dots, 1]$

Population gradient for ith coordinate of weight vector is $\mathbb{E}[-y\sigma'(w^\top x)x_i]$

For $i \in S$, this is the $(k-1)$th order Fourier coefficient $S \backslash \{i\}$ of $x \to \sigma'(w^\top x)$

For $i \notin S$, this is the $(k+1)$th order Fourier coefficient $S \cup \{i\}$ of $x \to \sigma'(w^\top x)$

At initialization: $\sigma'(w^\top x) = \frac{\mathrm{sign}(1^\top x)+1}{2}$ *(shifted majority function)*

The Fourier gap is $\approx_k n^{-(k-1)/2}$ [O'Donnell'14] and can be detected with $\approx_k n^{k-1}$ samples
(additive)

This information is (potentially) accumulated over samples in the small batch setting

THEORETICAL RESULT - HIDDEN INFORMATION

Theorem [informal]:
One hidden-layer MLPs with ReLU activation and $2^{O(k)}$ hidden units learn k-sparse parities using large batch SGD with compute time (batch-size × run-time) scaling as $n^{O(k)}$.

(NTK requires at least $n^{\Omega(k)}$ hidden units)

Large batch: First gradient step has enough *information* for hidden units to pick out correct parity indices.

Caveat: doesn't work with standard learning rate schedule (Standard schedule results in elbow curves!)

THEORETICAL RESULT - GF/SGD

Disjoint PolyNet: $\prod_{i=1}^{k} w_i^{\top} x_i$ for $x = [x_1, \ldots, x_k]$

Theorem [informal]:

For PolyNets with $k \geq 3$ and $\epsilon > 0$, the fraction of the time it takes for the error to fall below 0.49 is at least $1 - \tilde{O}\left(\frac{1}{(n/k)^{k/2-1}}\right)$ fraction of the running time required to achieve zero error.

PolyNet $x \mapsto \prod_{i=1}^{k} (w_i^{\top} x)$

Explains phase transition in the gradient flow regime, can be extended to SGD

Small batch: Random walk with *bias* towards relevant coordinates

Mysteries of contemporary deep learning

1. How do neural networks learn to construct useful features?
2. How do neural networks learn to “reason” / compute “combinatorial” functions?
3. Why are there sometimes emergent breakthroughs in capabilities as resources are scaled up?

In some “combinatorial” tasks, like learning sparse parities, features are only useful when they are learned *together*, and to a sufficient extent. In other words: the network needs to learn from scratch to compute a certain circuit. In these situations, we may see a “phase transition” in the loss curve, even though there is hidden progress inside the black box.

Still mysteries

1. How do neural networks learn to hierarchically construct useful features?
2. How do neural networks learn to “reason” / compute complex “combinatorial” functions?
3. Why are there sometimes emergent breakthroughs in capabilities as resources like network size are scaled up?

In some “combinatorial” tasks, like learning sparse parities, features are only useful when they are learned *together*, and to a sufficient extent. In other words: the network needs to learn from scratch to compute a certain circuit. In these situations, we may see a “phase transition” in the loss curve, even though there is hidden progress inside the black box.

Thank you!

1. Inductive Biases and Variable Creation in Self-Attention Mechanisms, ICML '22

with Surbhi Goel, Sham Kakade, & Cyril Zhang

2. Hidden Progress in Deep Learning: SGD Learns Parities Near the Computational Limit, NeurIPS '22

With Boaz Barak, Surbhi Goel, Sham Kakade, Eran Malach, Cyril Zhang, NeurIPS 2022

Questions??

benjaminedelman.com

bedelman@g.harvard.edu