Machine Learning Department
School of Computer Science
Carnegie Mellon University

Pretraining vs. finetuning + Modern Transformers

(RoPE, GQA, Longformer)

Matt Gormley & Pat Virtue

Lecture 4

Jan. 27, 2025

Reminders

- Homework 0: PyTorch + Weights & Biases
  - Out: Wed, Jan 17
  - Due: Mon, Jan 27 at 11:59pm
- Quiz 1: Wed, Jan 29
- Homework 1: Generative Models of Text
  - Out: Mon, Jan 27
  - Due: Mon, Feb 10 at 11:59pm

Two parts: Deep Learning and Language Modeling

Recap So Far

Deep Learning

- AutoDiff
  - is a tool for **computing gradients** of a differentiable function, b = f(a)
  - the key building block is a **module** with a forward() and backward()
  - sometimes define f as **code** in forward() by chaining existing modules together
- Computation Graphs
  - are another way to define f (more conducive to slides)
  - so far, we saw two (deep) computation graphs
    - 1) RNN-LM
    - 2) Transformer-LM
    - (Transformer-LM was kind of complicated)

Language Modeling

- key idea: condition on previous words to **sample the next word**
- to define the **probability** of the next word…
  - …n-gram LM uses collection of massive 50k-sided **dice**
  - …RNN-LM or Transformer-LM use a **neural network**

- Learning an LM
  - n-gram LMs are easy to learn: just **count** co-occurrences!
  - a RNN-LM / Transformer-LM is trained by optimizing an objective function with SGD; compute gradients with AutoDiff

Recall…

LEARNING A TRANSFORMER LM

Learning a Language Model

<u>Question</u>: How do we **learn** the probabilities for the n-Gram Model?
<u>Answer</u>: From data! Just **count** n-gram frequencies

…the **cows eat** grass…
…our **cows eat hay** daily…
…factory-farm **cows eat** corn…
…on an organic farm, **cows eat hay** and…
…do your **cows eat** grass or corn?...
…what do **cows eat if** they have…
…**cows eat** corn when there is no…
…which **cows eat which** foods depends…
…if **cows eat** grass…
…when **cows eat** corn their stomachs…
…should we let **cows eat** corn?...

$p(w_t \mid w_{t-2} = \text{cows}, w_{t-1} = \text{eat})$

w_t	$p(\cdot \mid \cdot, \cdot)$
corn	4/11
grass	3/11
hay	2/11
if	1/11
which	1/11

MLE for n-gram LM

- This counting method gives us the **maximum likelihood estimate** of the n-gram LM parameters
- We can derive it in the usual way:
  - **Write the likelihood** of the sentences under the n-gram LM
  - **Set the gradient to zero** and impose the constraint that the probabilities sum-to-one
  - **Solve** for the MLE

Learning a Language Model

MLE for Deep Neural LM

- We can also use maximum likelihood estimation to learn the parameters of an RNN-LM or Transformer-LM too!
- But **not in closed form** – instead we follow a different recipe:
  - Write the **likelihood** of the sentences under the Deep Neural LM model
  - Compute the **gradient** of the (batch) likelihood w.r.t. the parameters **by AutoDiff**
  - Follow the negative gradient using **Mini-batch SGD** (or your favorite optimizer)

MLE for n-gram LM

- This counting method gives us the **maximum likelihood estimate** of the n-gram LM parameters
- We can derive it in the usual way:
  - **Write the likelihood** of the sentences under the n-gram LM
  - **Set the gradient to zero** and impose the constraint that the probabilities sum-to-one
  - **Solve** for the MLE

SGD and Mini-batch SGD

Recall...

Algorithm 1 SGD

Initialize $\theta^{(0)}$
$s = 0$
for $t = 1, 2, \ldots, T$ **do**
    for $i \in$ shuffle$(1, \ldots, N)$ **do**
        Select the next training point (x_i, y_i)
        Compute the gradient $g^{(s)} = \nabla J_i(\theta^{(s-1)})$
        Update parameters $\theta^{(s)} = \theta^{(s-1)} - \eta g^{(s)}$
        Increment time step $s = s + 1$
    Evaluate average training loss $J(\theta) = \frac{1}{n} \sum_{i=1}^{n} J_i(\theta)$
return $\theta^{(s)}$

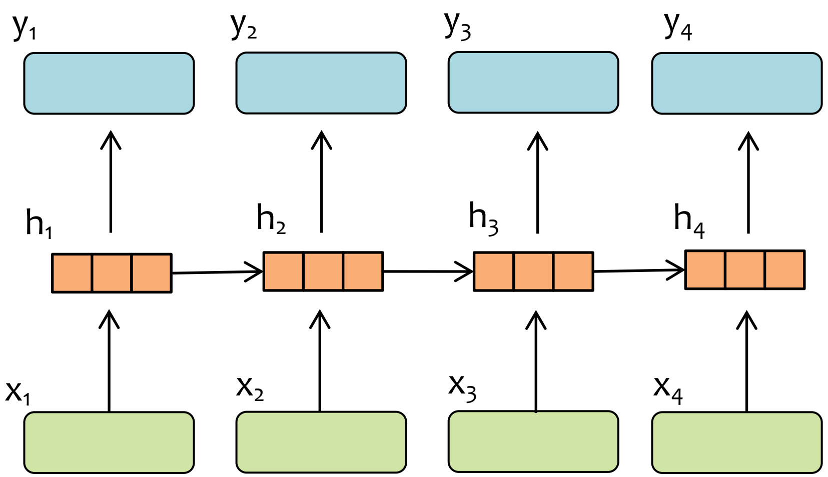
Recall...

SGD and Mini-batch SGD

Algorithm 1 Mini-Batch SGD

Initialize $\theta^{(0)}$
Divide examples $\{1, \dots, N\}$ randomly into batches $\{I_1, \dots, I_B\}$
where $\bigcup_{b=1}^{B} I_b = \{1, \dots, N\}$ and $\bigcap_{b=1}^{B} I_b = \emptyset$
$s = 0$
for $t = 1, 2, \dots, T$ **do**
    for $b = 1, 2, \dots, B$ **do**
        Select the next batch I_b, where $m = |I_b|$
        Compute the gradient $g^{(s)} = \frac{1}{m} \sum_{i \in I_b} \nabla J_i(\theta^{(s)})$
        Update parameters $\theta^{(s)} = \theta^{(s-1)} - \eta g^{(s)}$
        Increment time step $s = s + 1$
    Evaluate average training loss $J(\theta) = \frac{1}{n} \sum_{i=1}^{n} J_i(\theta)$
return $\theta^{(s)}$

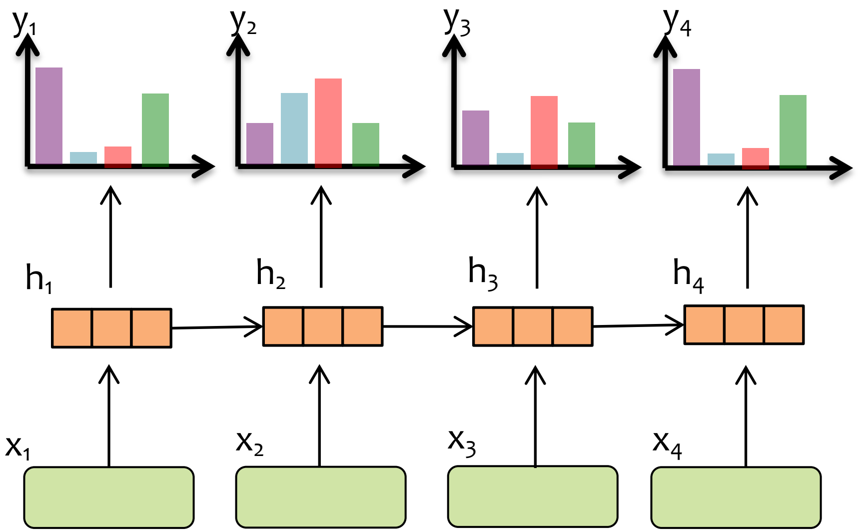
RNN



Algorithm 1 Elman RNN

procedure FORWARD$(x_{1:T}, W_{ah}, W_{ax}, b_a, W_{yh}, b_y)$
    Initialize the hidden state h_0 to zeros
    for t in 1 to T **do**
        Receive input data at time step t: x_t
        Compute the hidden state update:
            $a_t = W_{ah} \cdot h_{t-1} + W_{ax} \cdot x_t + b_a$
            $h_t = \sigma(a_t)$
        Compute the output at time step t:
            $y_t = W_{yh} \cdot h_t + b_y$

RNN



Algorithm 1 Elman RNN

procedure FORWARD($x_{1:T}, W_{ah}, W_{ax}, b_a, W_{yh}, b_y$)

  Initialize the hidden state h_0 to zeros

  for t in 1 to T **do**

    Receive input data at time step t: x_t

    Compute the hidden state update:

      $a_t = W_{ah} \cdot h_{t-1} + W_{ax} \cdot x_t + b_a$

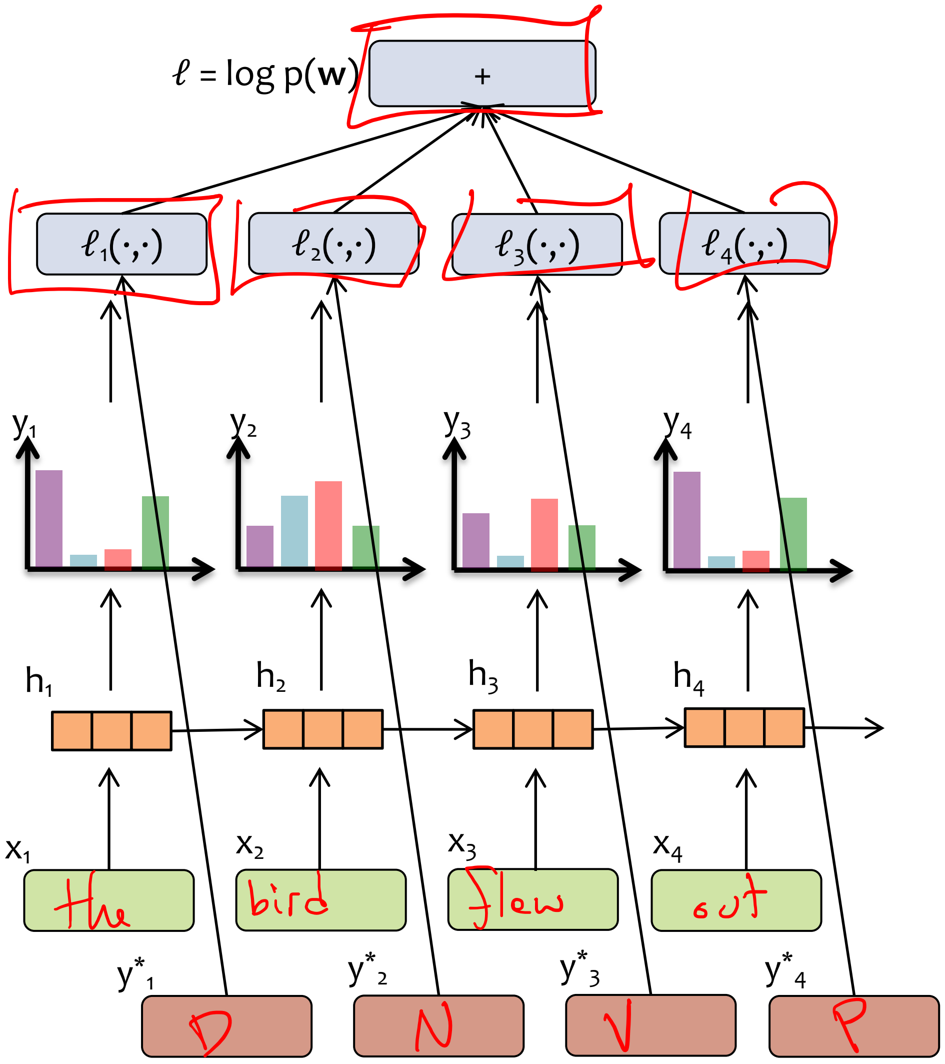
      $h_t = \sigma(a_t)$

    Compute the output at time step t:

      $y_t = \text{softmax}(W_{yh} \cdot h_t + b_y)$

RNN + Loss

How can we use this to compute the loss for an RNN-LM?



Algorithm 1 Elman RNN + Loss

procedure FORWARD($x_{1:T}, y^*_{1:T} W_{ah}, W_{ax}, b_a, W_{yh}, b_y$)

  Initialize the hidden state h_0 to zeros

  for t in 1 to T **do**

    Receive input data at time step t: x_t

    Compute the hidden state update:

      $a_t = W_{ah} \cdot h_{t-1} + W_{ax} \cdot x_t + b_a$

      $h_t = \sigma(a_t)$

    Compute the output at time step t:

      $y_t = \text{softmax}(W_{yh} \cdot h_t + b_y)$

    Compute the cross-entropy loss at time step t:

      $\ell_t = -\sum_{k=1}^{K} (y_t^*)_k \log((y_t)_k)$

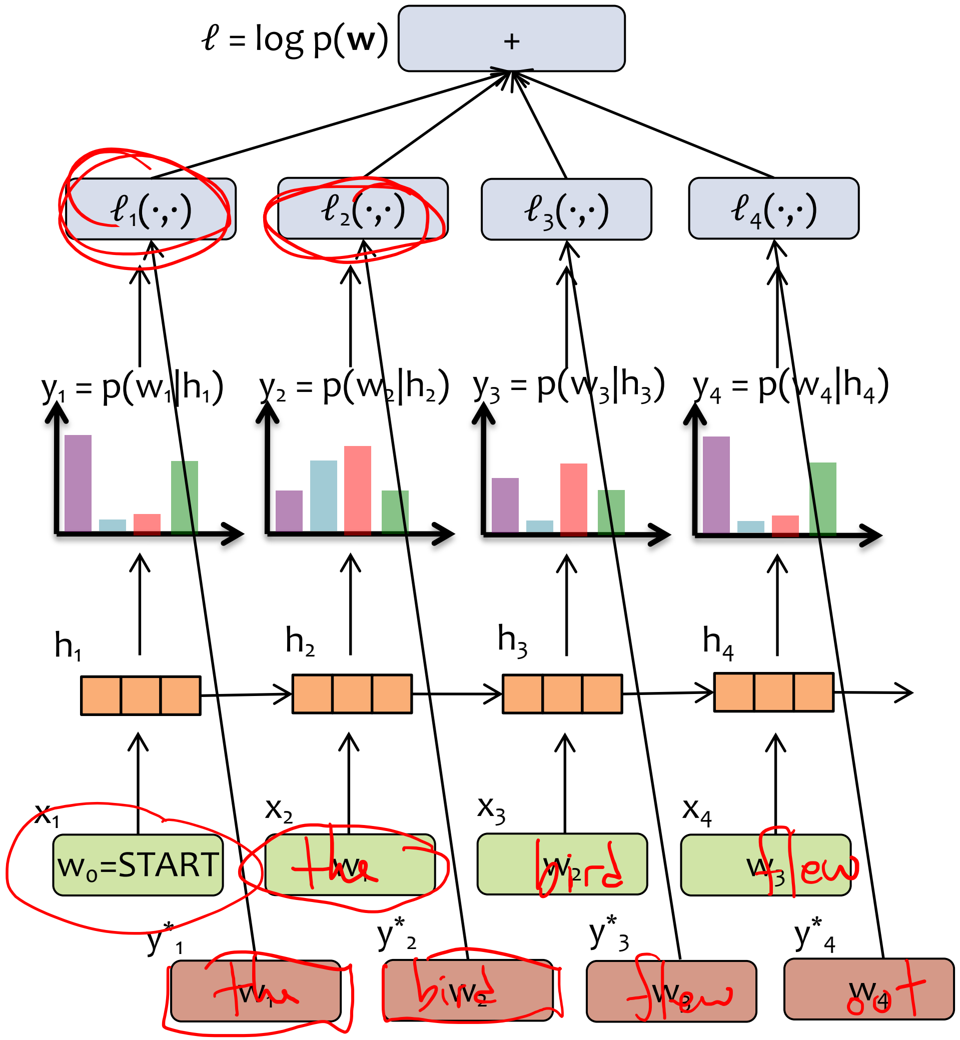
  Compute the total loss:

    $\ell = \sum_{t=1}^{T} \ell_t$

RNN-LM + Loss

How can we use this to compute the loss for an RNN-LM?

log p(**w**) = log p(w_1, w_2, w_3, … , w_T)
= log p(w_1 | h_1) + … + log p(w_2 | h_T)



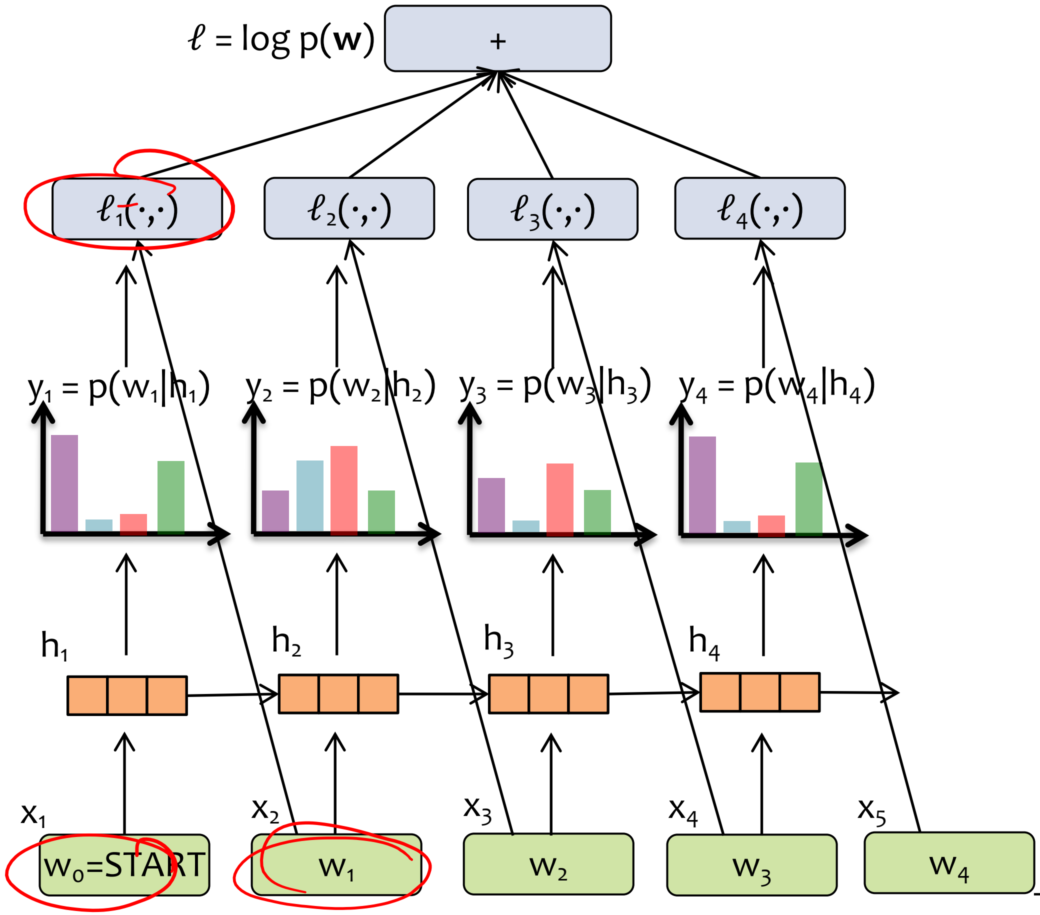
Algorithm 1 Elman RNN + Loss

1: **procedure** FORWARD($x_{1:T}, y^*_{1:T} W_{ah}, W_{ax}, b_a, W_{yh}, b_y$)
2: Initialize the hidden state h_0 to zeros
3: **for** t in 1 to T **do**
4: Receive input data at time step t: x_t
5: Compute the hidden state update:
6: $a_t = W_{ah} \cdot h_{t-1} + W_{ax} \cdot x_t + b_a$
7: $h_t = \sigma(a_t)$
8: Compute the output at time step t:
9: $y_t = \text{softmax}(W_{yh} \cdot h_t + b_y)$
10: Compute the cross-entropy loss at time step t:
11: $\ell_t = -\sum_{k=1}^{K} (y^*_t)_k \log((y_t)_k)$
12: Compute the total loss:
13: $\ell = \sum_{t=1}^{T} \ell_t$

RNN-LM + Loss

How can we use this to compute the loss for an RNN-LM?

$$\log p(\mathbf{w}) = \log p(w_1, w_2, w_3, \ldots, w_T)$$
$$= \log p(w_1 \mid h_1) + \ldots + \log p(w_2 \mid h_T)$$



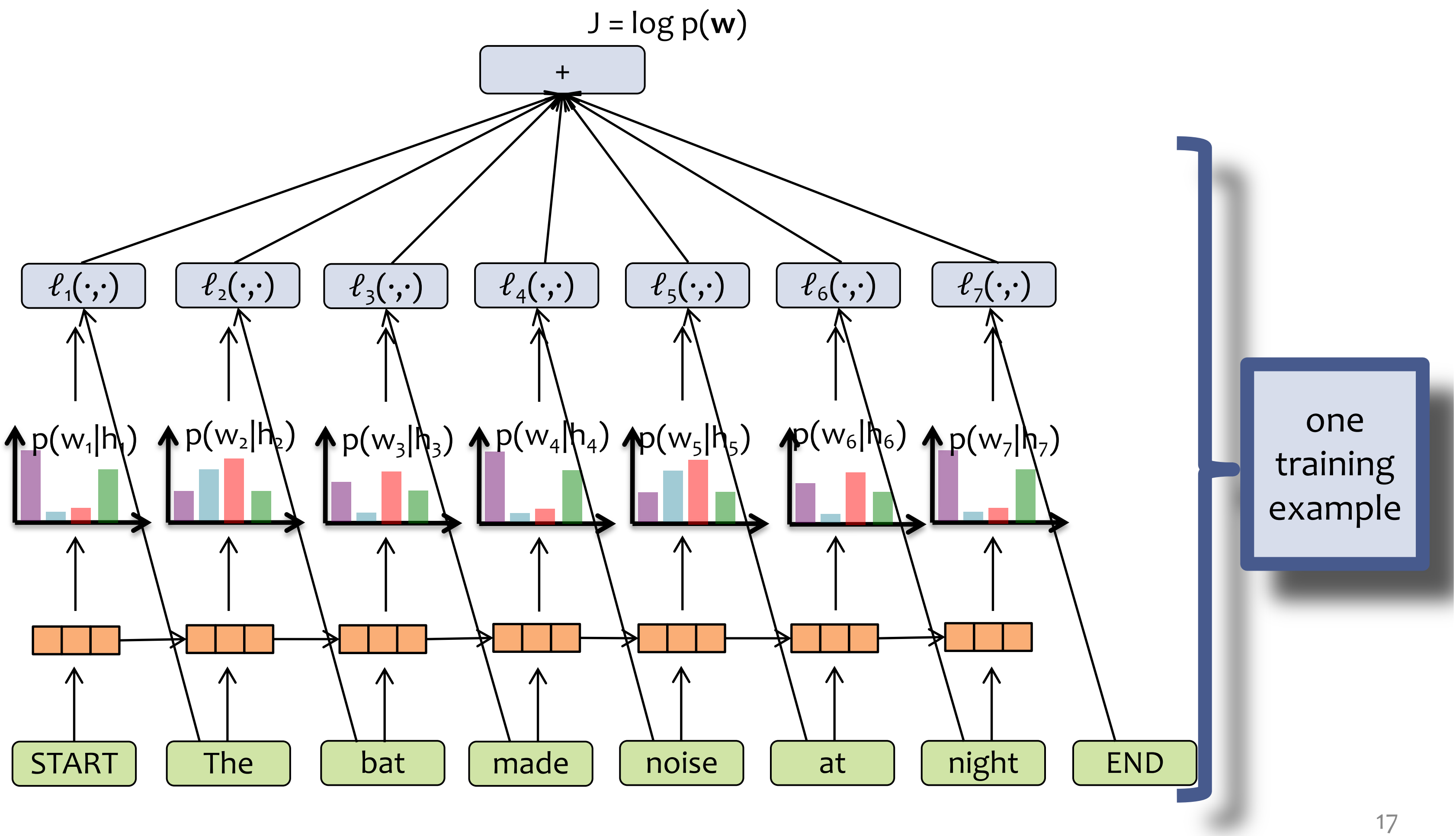
Algorithm 1 Elman RNN + Loss

procedure FORWARD($x_{1:T}, y^*_{1:T} W_{ah}, W_{ax}, b_a, W_{yh}, b_y$)
- Initialize the hidden state h_0 to zeros
- **for** t in 1 to T **do**
  - Receive input data at time step t: x_t
  - Compute the hidden state update:
    - $a_t = W_{ah} \cdot h_{t-1} + W_{ax} \cdot x_t + b_a$
    - $h_t = \sigma(a_t)$
  - Compute the output at time step t:
    - $y_t = \text{softmax}(W_{yh} \cdot h_t + b_y)$
  - Compute the cross-entropy loss at time step t:
    - $\ell_t = -\sum_{k=1}^{K} (y^*_t)_k \log((y_t)_k)$
- Compute the total loss:
  - $\ell = \sum_{t=1}^{T} \ell_t$

Learning an RNN-LM

- Each training example is a sequence (e.g. sentence), so we have training data D = {$\mathbf{w}^{(1)}$, $\mathbf{w}^{(2)}$, …, $\mathbf{w}^{(N)}$}
- The objective function for a Deep LM (e.g. RNN-LM or Tranformer-LM) is typically the log-likelihood of the training examples:
  $$J(\boldsymbol{\theta}) = \Sigma_i \log p_{\boldsymbol{\theta}}(\mathbf{w}^{(i)})$$
- We train by mini-batch SGD (or your favorite flavor of mini-batch SGD)

$$\log p(\mathbf{w}) = \log p(w_1, w_2, w_3, \dots, w_T)$$
$$= \log p(w_1 \mid h_1) + \log p(w_2 \mid h_2) + \dots + \log p(w_2 \mid h_T)$$



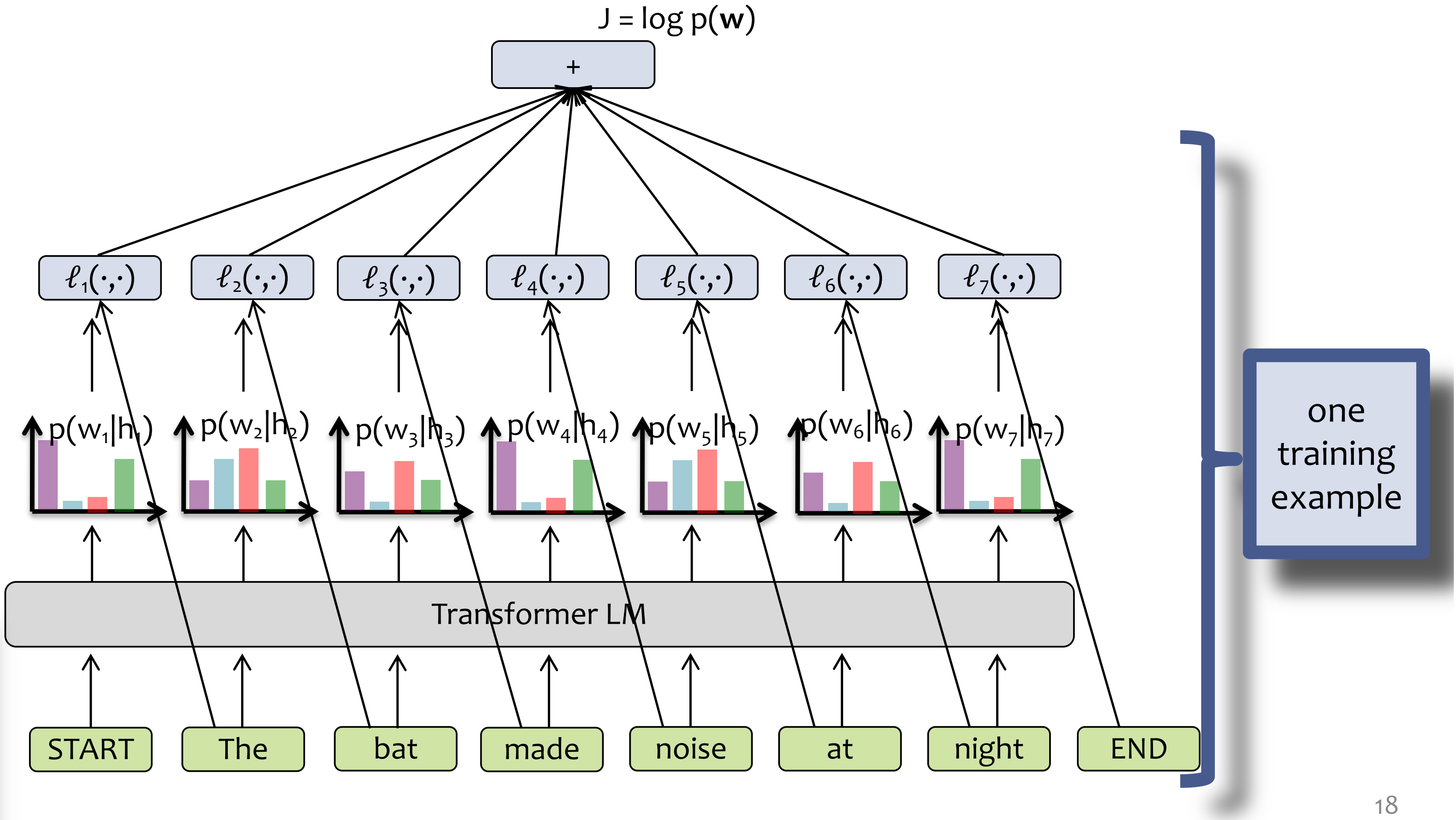
Learning a Transformer LM

- Each training example is a sequence (e.g. sentence), so we have training data D = {$\mathbf{w}^{(1)}$, $\mathbf{w}^{(2)}$, … , $\mathbf{w}^{(N)}$}
- The objective function for a Deep LM (e.g. RNN-LM or Tranformer-LM) is typically the log-likelihood of the training examples:

$$J(\boldsymbol{\theta}) = \Sigma_i \log p_{\boldsymbol{\theta}}(\mathbf{w}^{(i)})$$

- We train by mini-batch SGD (or your favorite flavor of mini-batch SGD)

$$\log p(\mathbf{w}) = \log p(w_1, w_2, w_3, \ldots , w_T)$$
$$= \log p(w_1 \mid h_1) + \log p(w_2 \mid h_2) + \ldots + \log p(w_2 \mid h_T)$$



Training a Transformer-LM is the same, except we swap in a different deep language model.

Language Modeling

An aside:

- State-of-the-art language models currently tend to rely on **transformer networks** (e.g. GPT-2)
- RNN-LMs comprised most of the early neural LMs that **led to** current SOTA architectures

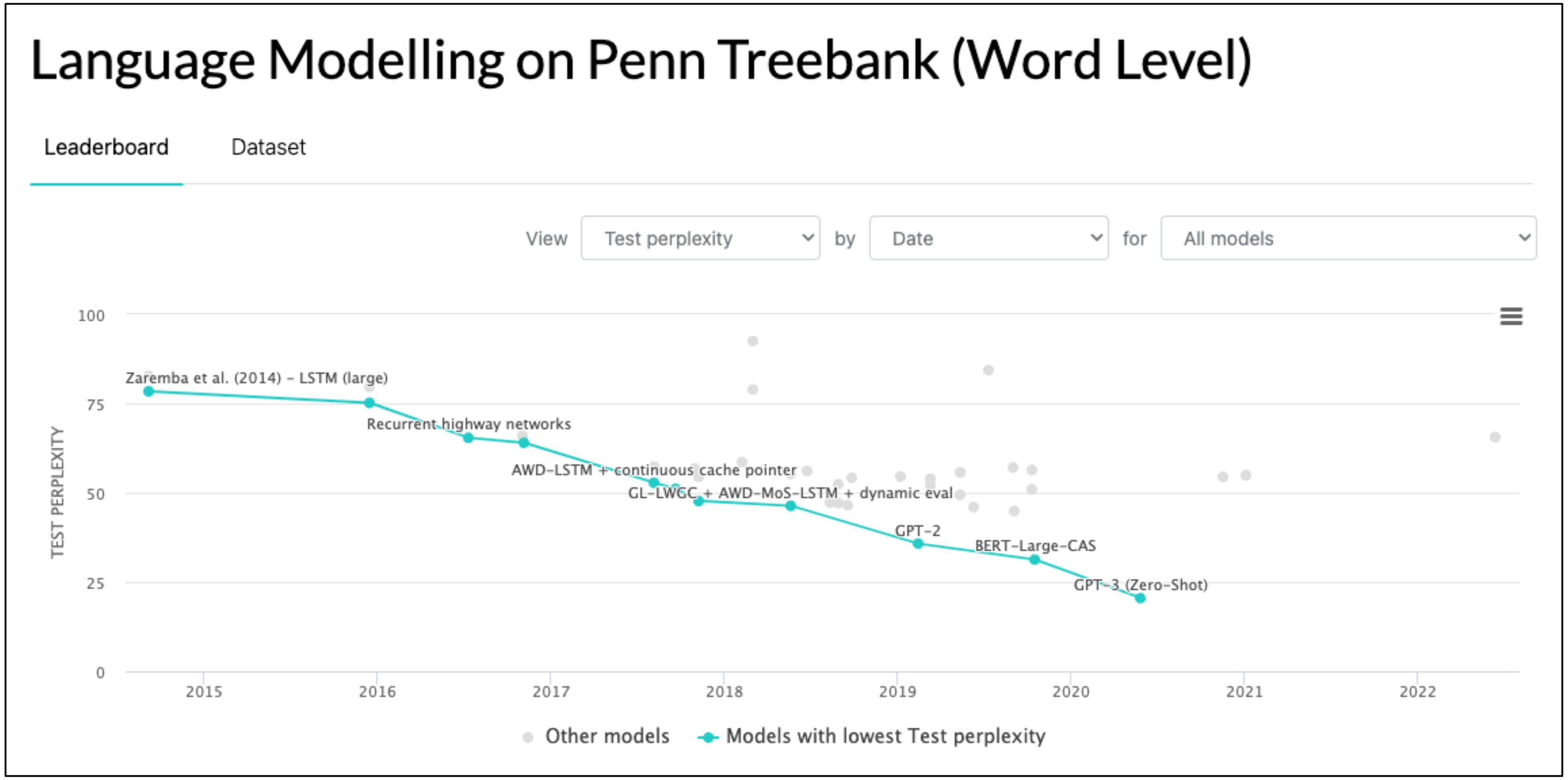


Figure from https://paperswithcode.com/sota/language-modelling-on-penn-treebank-word

PRE-TRAINING VS. FINE-TUNING

The Start of Deep Learning

- The architectures of modern deep learning have a long history:
  - 1960s: Rosenblatt's 3-layer multi-layer perceptron, ReLU)
  - 1970-80s: RNNs and CNNs
  - 1990s: linearized self-attention
- The spark for deep learning came in 2006 thanks to **pre-training** (e.g., Hinton & Salakhutdinov, 2006)

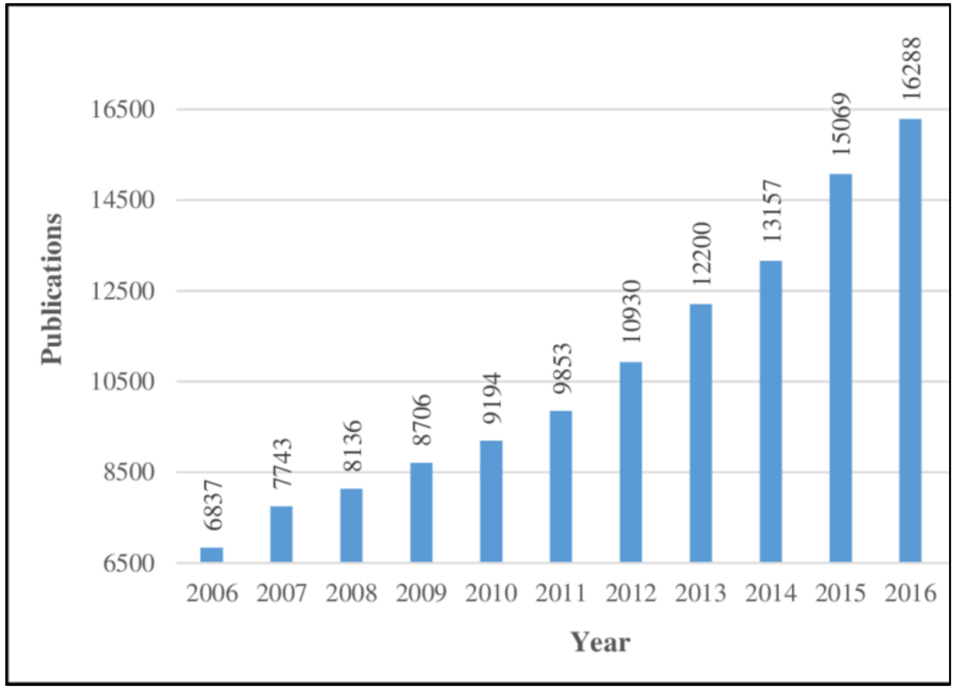


Figure from Vargas et al. (2017)

Pre-Training vs. Fine-Tuning

Definitions

Pre-training

- randomly initialize the parameters, then…
- *option A*: unsupervised training on very large set of unlabeled instances
- *option B*: supervised training on a very large set of labeled examples

Fine-tuning

- initialize parameters to values from pre-training
- (optionally), add a prediction head with a small number of randomly initialized parameters
- train on a specific task of interest by backprop

Example: Vision Models

Pre-training

- Example A: unsupervised autoencoder training on very large set of unlabeled images (e.g. MNIST digits)
- Example B: supervised training on a very large image classification dataset (e.g. ImageNet w/21k classes and 14M images)

Fine-tuning

- object detection, training on 200k labeled images from COCO
- semantic segmentation, training on 20k labeled images from ADE20k

Example: Language Models

Pre-training

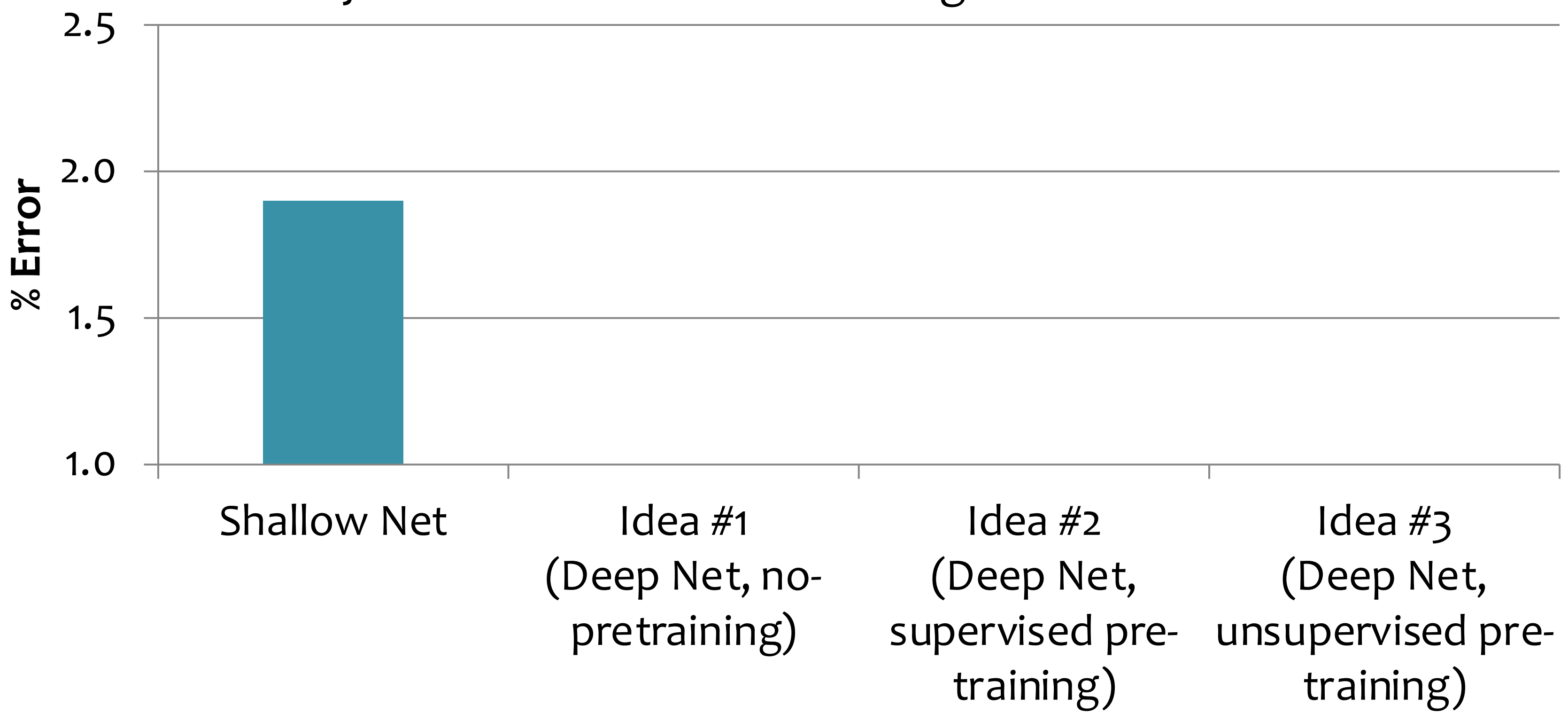
- unsupervised pre-training by maximizing likelihood of a large set of unlabeled sentences such as…
- The Pile (800 Gb of text)
- Dolma (3 trillion tokens)

Fine-tuning

- MMLU benchmark: a few training examples from 57 different tasks ranging from elementary mathematics to genetics to law
- code generation, training on ~400 training examples from MBPP

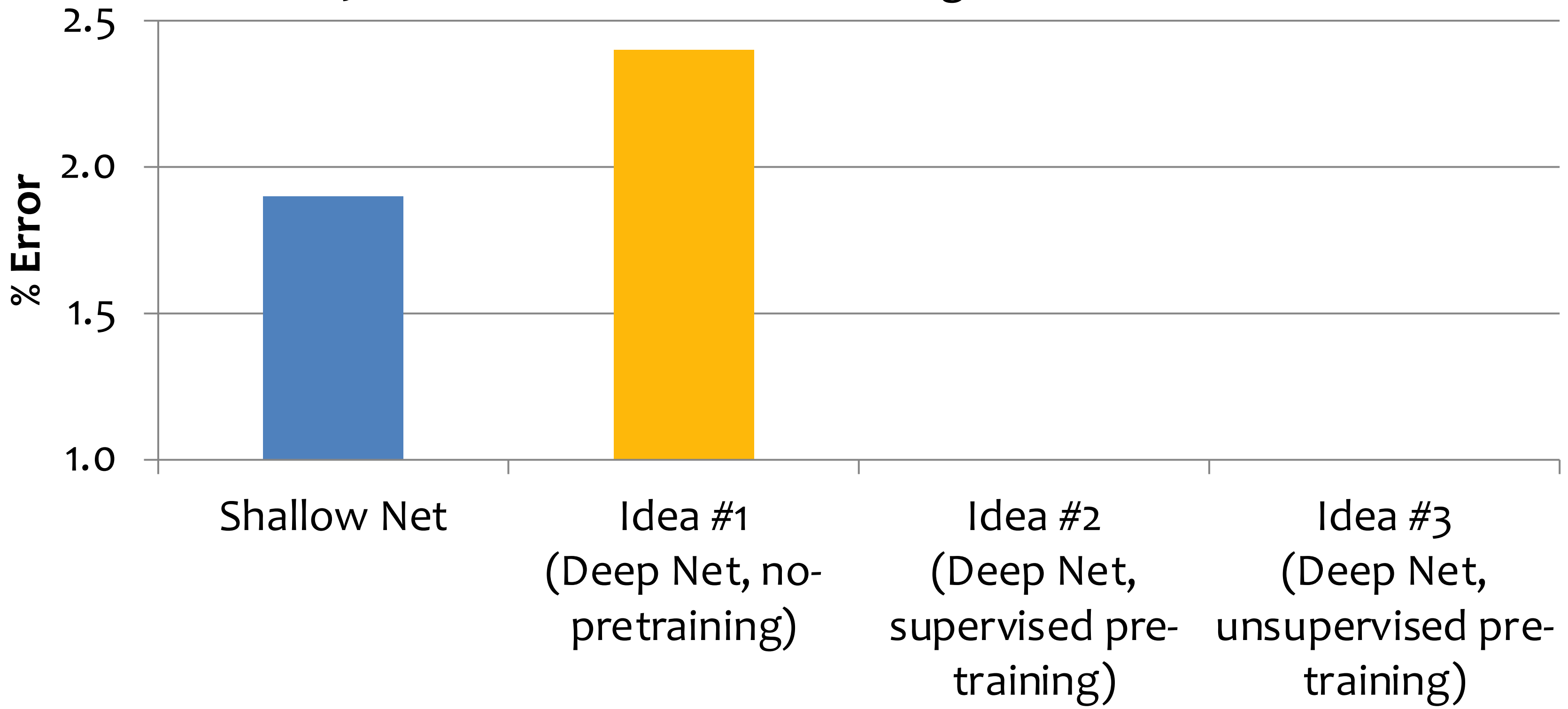
Pre-Training and Fine-Tuning on MNIST

- Results from Bengio et al. (2006) on MNIST digit classification task
- Percent error (lower is better)
- Some methods first do pre-training
- Every method includes fine-tuning on labeled data



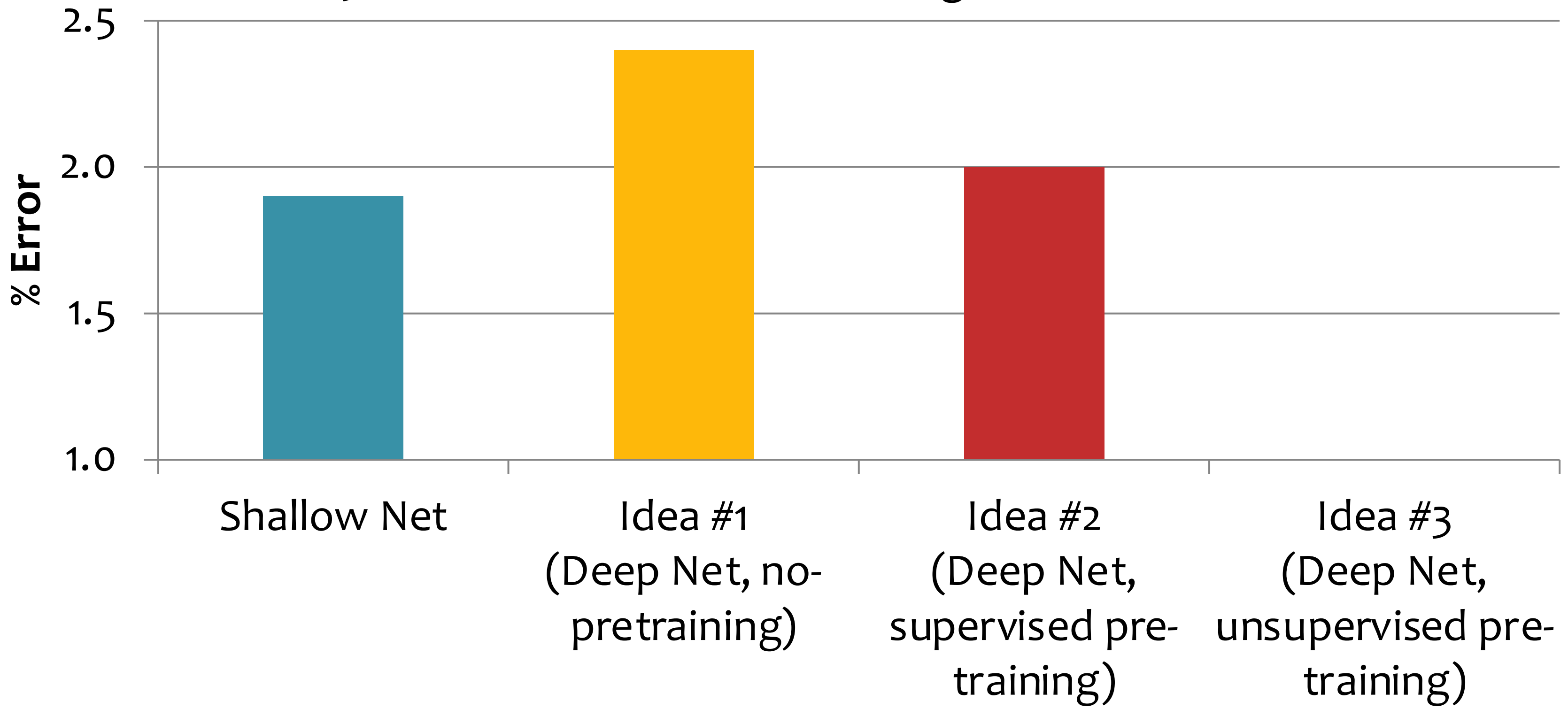
Pre-Training and Fine-Tuning on MNIST

- Results from Bengio et al. (2006) on MNIST digit classification task
- Percent error (lower is better)
- Some methods first do pre-training
- Every method includes fine-tuning on labeled data



Pre-Training and Fine-Tuning on MNIST

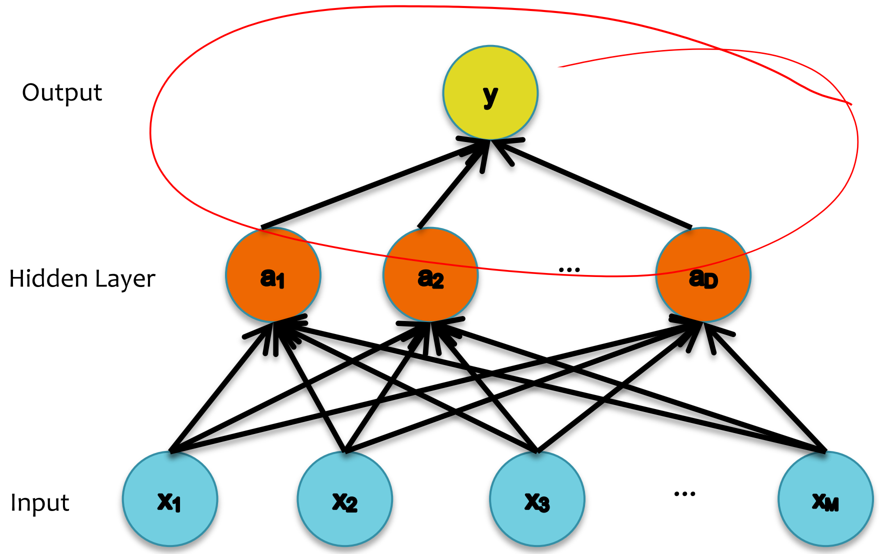
- Results from Bengio et al. (2006) on MNIST digit classification task
- Percent error (lower is better)
- Some methods first do pre-training
- Every method includes fine-tuning on labeled data



Unsupervised Autoencoder Pre-Training for Vision

Unsupervised pre-training of the first layer:

- What should it predict?
- What else do we observe?
- **The input!**

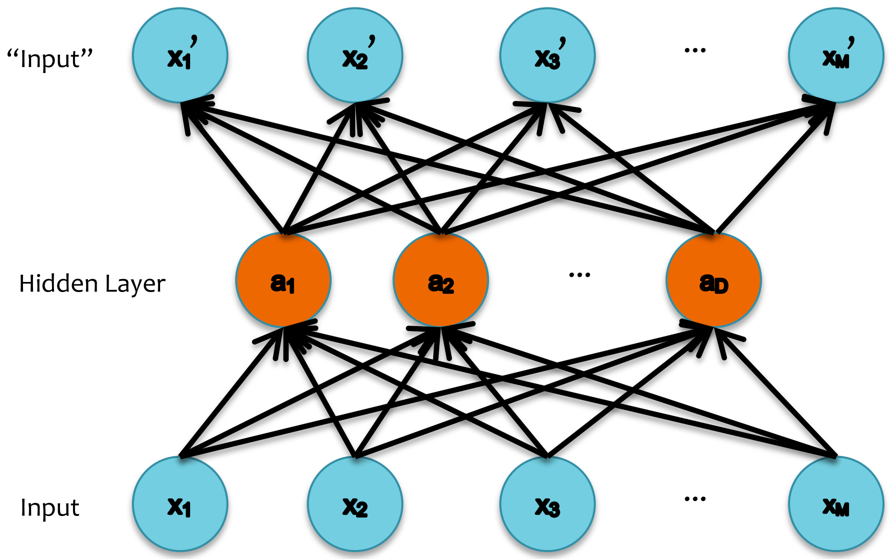


Unsupervised Autoencoder Pre-Training for Vision

Unsupervised pre-training of the first layer:

- What should it predict?
- What else do we observe?
- **The input!**

This topology defines an Auto-encoder.



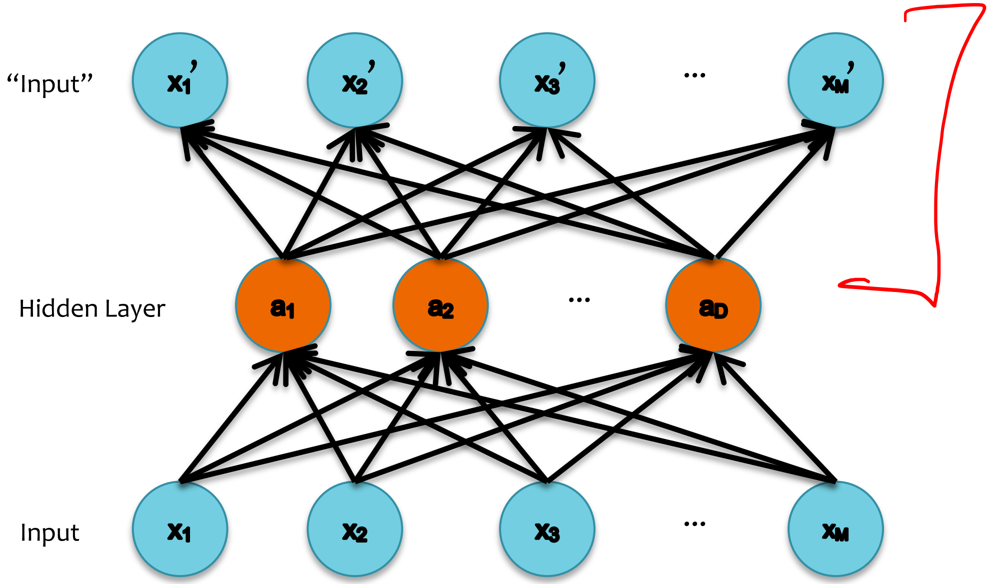
Unsupervised Autoencoder Pre-Training for Vision

Key idea: Encourage z to give small reconstruction error:

- x' is the *reconstruction* of x
- Loss = || x – DECODER(ENCODER(x)) ||^2
- Train with the same backpropagation algorithm for 2-layer Neural Networks with x_m as both input and output.

DECODER: x' = h(W'z)

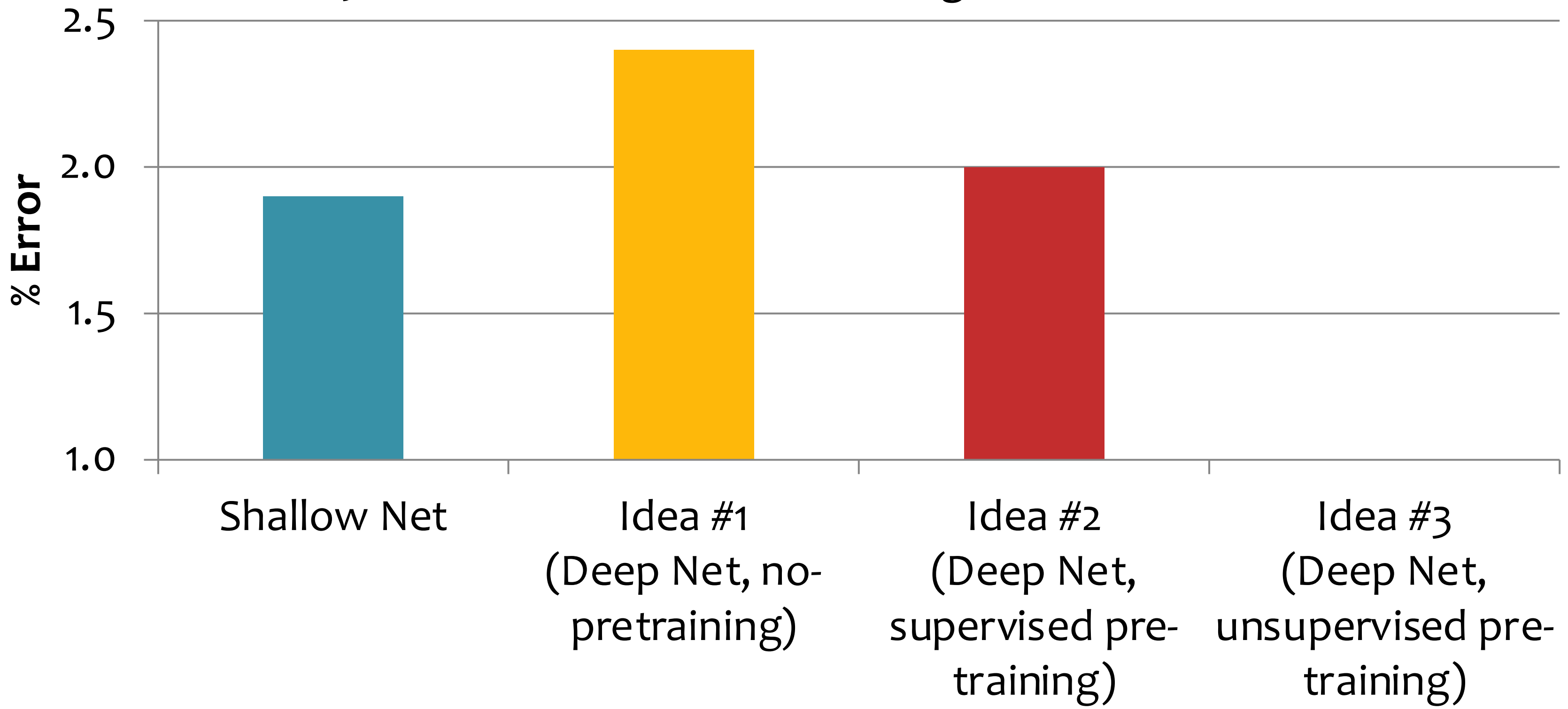
ENCODER: z = h(Wx)



Slide adapted from Raman Arora

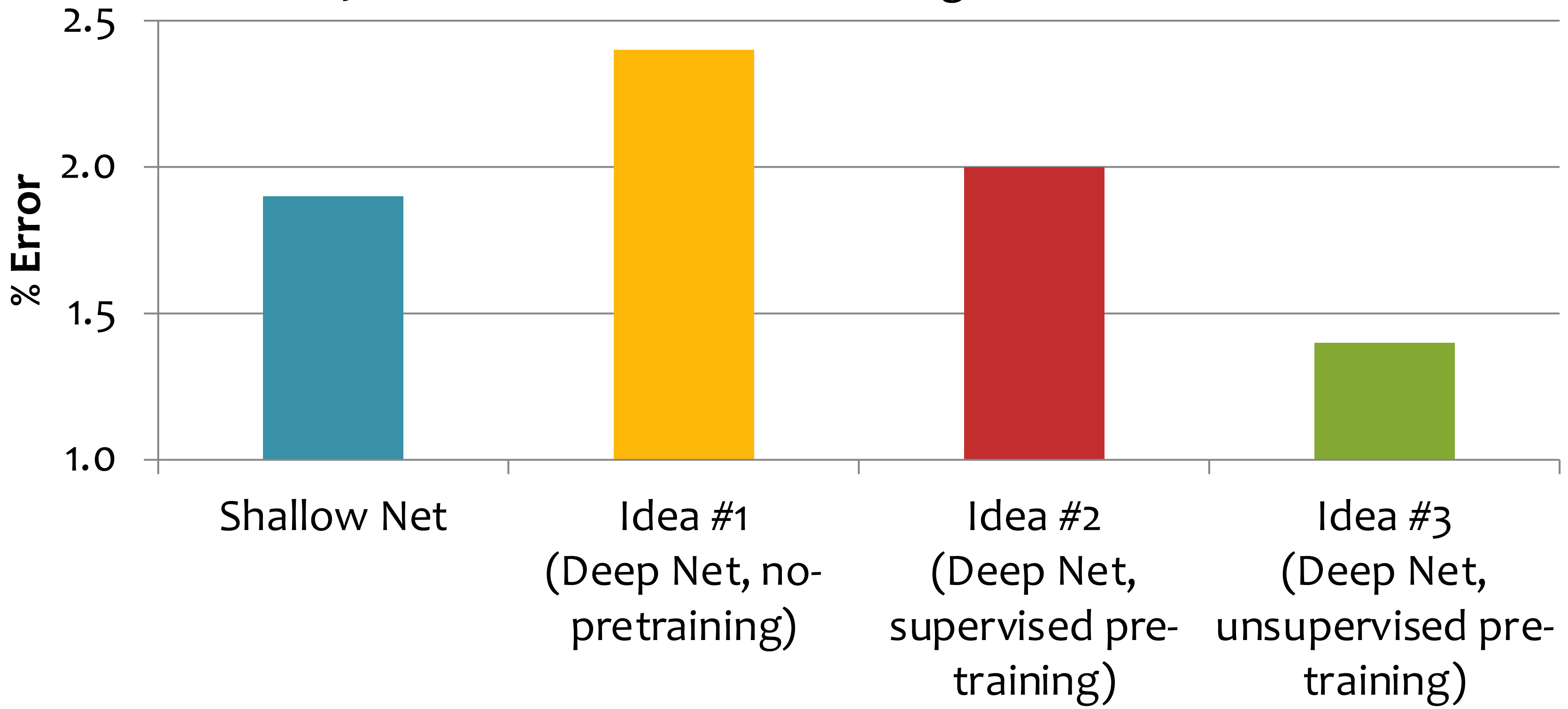
Pre-Training and Fine-Tuning on MNIST

- Results from Bengio et al. (2006) on MNIST digit classification task
- Percent error (lower is better)
- Some methods first do pre-training
- Every method includes fine-tuning on labeled data



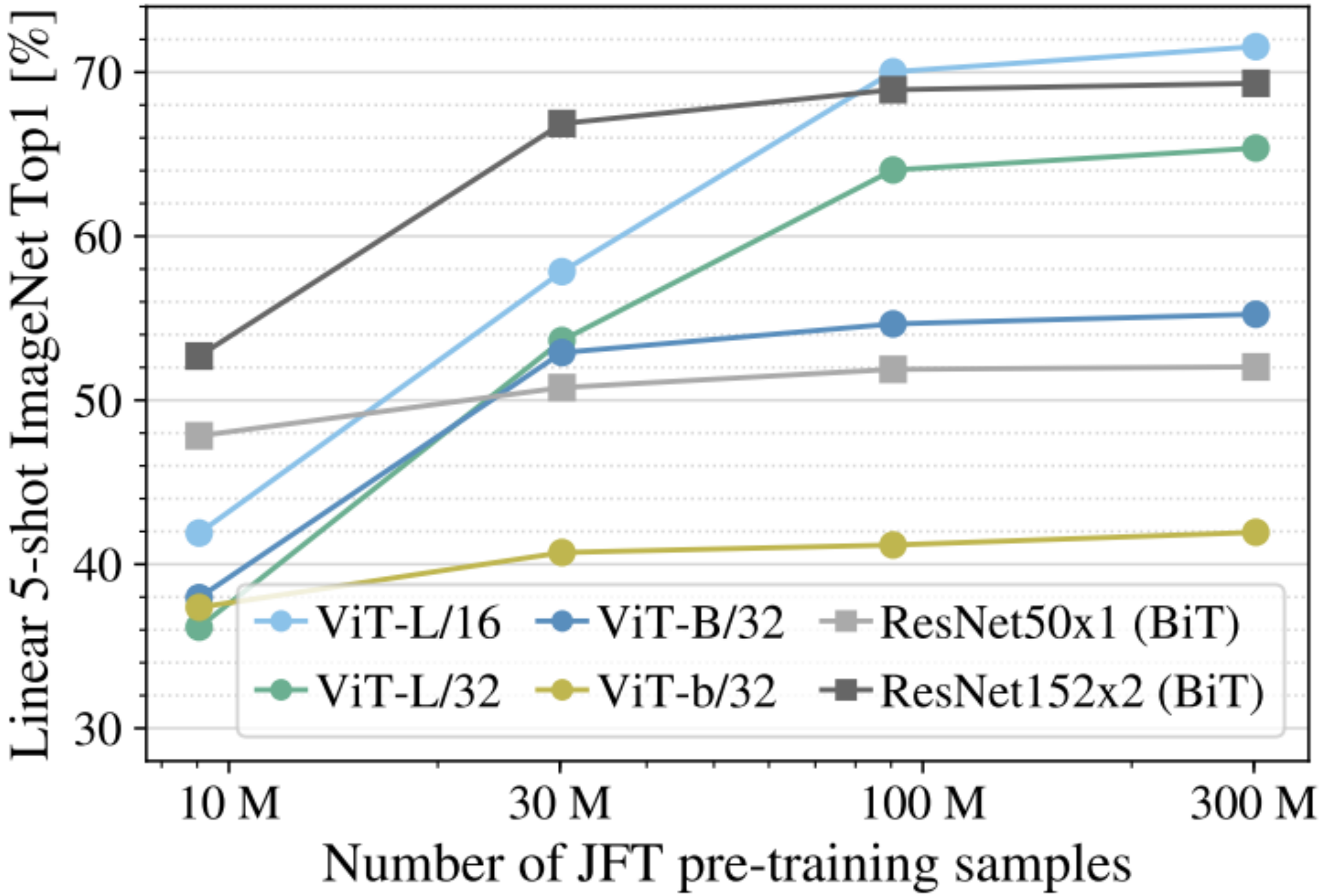
Pre-Training and Fine-Tuning on MNIST

- Results from Bengio et al. (2006) on MNIST digit classification task
- Percent error (lower is better)
- Some methods first do pre-training
- Every method includes fine-tuning on labeled data



Supervised Pre-Training for Vision

- Nowadays, we tend to just do supervised pre-training on a massive labeled dataset
- Vision Transformer's success was largely due to using a much larger pre-training dataset



Wrong?

Figure from ~~https://arxiv.org/pdf/2010.11929~~

Pre-Training vs. Fine-Tuning

Definitions

Pre-training

- randomly initialize the parameters, then…
- *option A*: unsupervised training on very large set of unlabeled instances
- *option B*: supervised training on a very large set of labeled examples

Fine-tuning

- initialize parameters to values from pre-training
- (optionally), add a prediction head with a small number of randomly initialized parameters
- train on a specific task of interest by backprop

Example: Vision Models

Pre-training

- Example A: unsupervised autoencoder training on very large set of unlabeled images (e.g. MNIST digits)
- Example B: supervised training on a very large image classification dataset (e.g. ImageNet w/21k classes and 14M images)

Fine-tuning

- object detection, training on 200k labeled images from COCO
- semantic segmentation, training on 20k labeled images from ADE20k

Example: Language Models

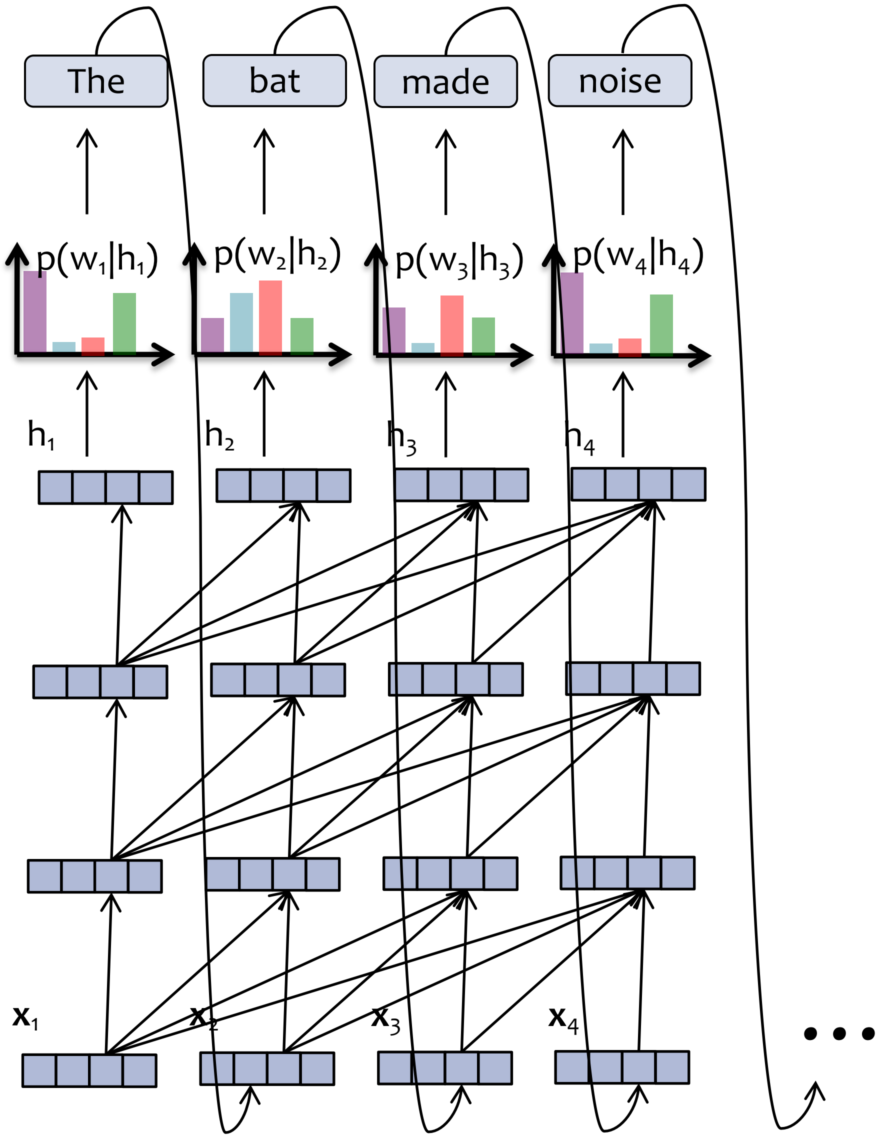
Pre-training

- unsupervised pre-training by maximizing likelihood of a large set of unlabeled sentences such as…
- The Pile (800 Gb of text)
- Dolma (3 trillion tokens)

Fine-tuning

- MMLU benchmark: a few training examples from 57 different tasks ranging from elementary mathematics to genetics to law
- code generation, training on ~400 training examples from MBPP

Unsupervised Pre-Training for an LLM



Generative pre-training for a deep language model:

- each training example is an (unlabeled) sentence
- the objective function is the likelihood of the observed sentence

Practically, we can **batch** together many such training examples to make training more efficient

Training Data for LLMs

GPT-3 Training Data:

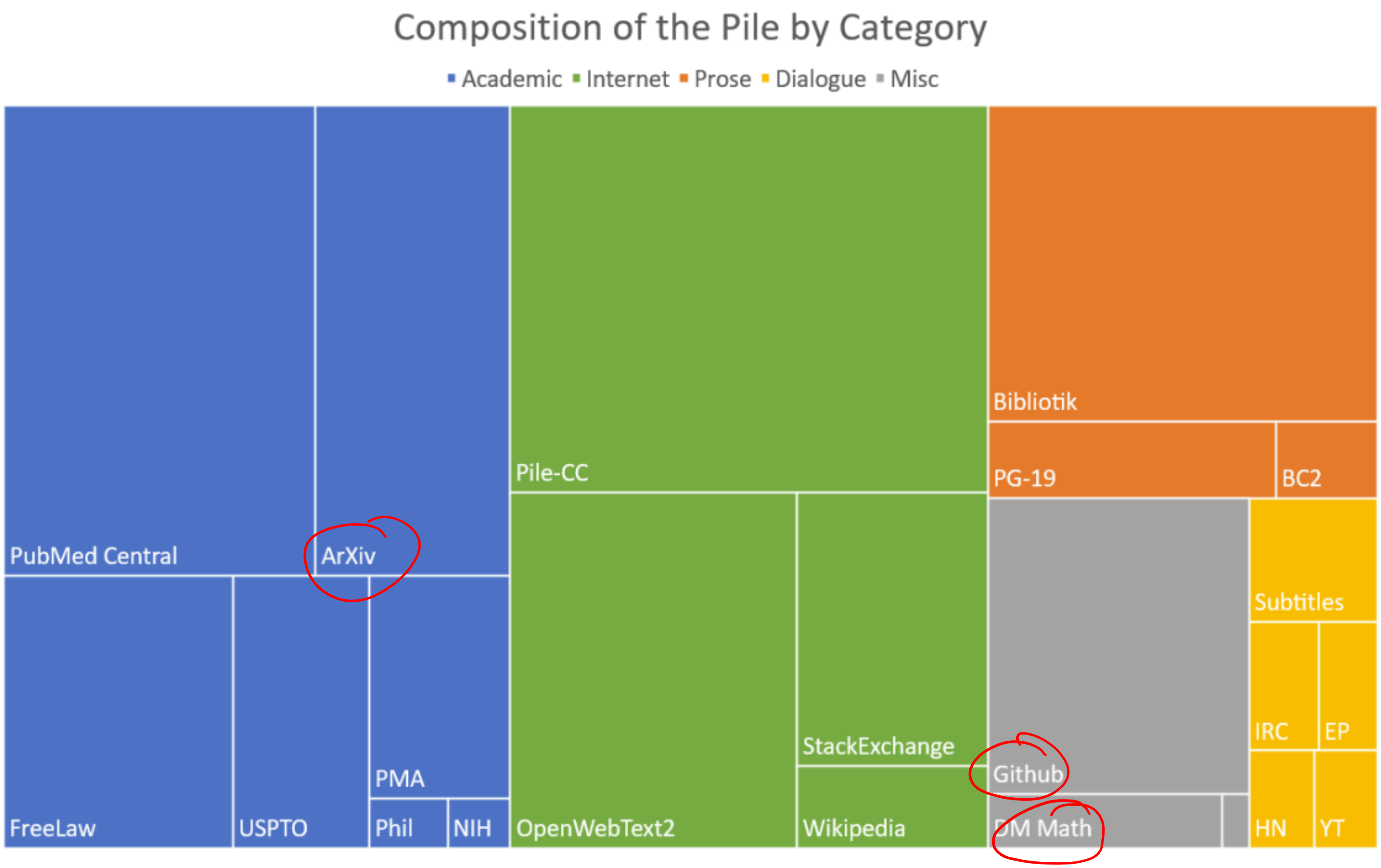
Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

Table from http://arxiv.org/abs/2005.14165

Training Data for LLMs

The Pile:

- An open source dataset for training language models
- Comprised of 22 smaller datasets
- Favors high quality text
- 825 Gb ≈ 1.2 trillion tokens



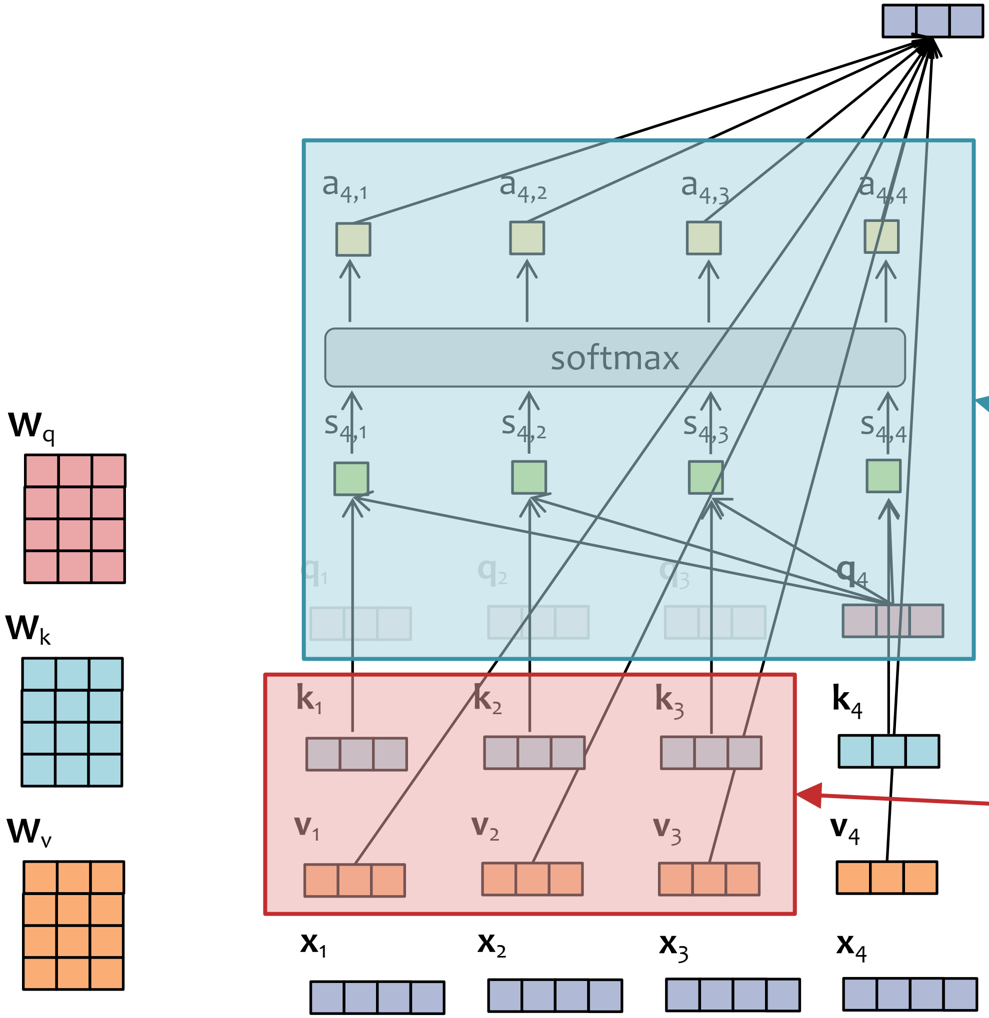
MODERN TRANSFORMER MODELS

Modern Tranformer Models

- PaLM (Oct 2022)
  - 540B parameters
  - closed source
  - Model:
    - SwiGLU instead of ReLU, GELU, or Swish
    - **multi-query attention** (MQA) instead of multi-headed attention
    - **rotary position embeddings**
    - **shared input-output embeddings** instead of separate parameter matrices
  - Training: **Adafactor** on 780 billion tokens
- Llama-1 (Feb 2023)
  - collection of models of varying parameter sizes: 7B, 13B, 32B, 65B
  - semi-open source
  - Llama-13B outperforms GPT-3 on average
  - Model compared to GPT-3:
    - **RMSNorm** on inputs instead of LayerNorm on outputs
    - **SwiGLU** activation function instead of ReLU
    - **rotary position embeddings (RoPE)** instead of absolute
  - Training: **AdamW** on 1.0 – 1.4 trillion tokens
- Falcon (June - Nov 2023)
  - models of size 7B, 40B, 180B
  - first fully open source model, Apache 2.0
  - Model compared to Llama-1:
    - ~~(GQA) instead of multi-headed attention~~ (MHA) or **grouped query attention** ~~**multi-query attention** (M~~QA)
    - **rotary position embeddings** (worked better than Alibi)
    - **GeLU** instead of SwiGLU
  - Training: AdamW on up to 3.5 trillion tokens for 180B model, using **z-loss** for stability and **weight decay**
- Llama-2 (Aug 2023)
  - collection of models of varying parameter sizes: 7B, 13B, 70B.
  - introduced Llama 2-Chat, fine-tuned as a dialogue agent
  - Model compared to Llama-1:
    - **grouped query attention (GQA)** instead of multi-headed attention (MHA)
    - context length of 4096 instead of 2048
  - Training: AdamW on 2.0 trillion tokens
- Mistral 7B (Oct 2023)
  - outperforms Llama-2 13B on average
  - introduced Mistral 7B – Instruct, fine-tuned as a dialogue agent
  - truly open source: Apache 2.0 license
  - Model compared to Llama-2
    - **sliding window attention** (with W=4096) and grouped-query attention (GQA) instead of just GQA
    - context length of 8192 instead of 4096 (can generate sequences up to length 32K)
    - **rolling buffer cache** (grow the KV cache and the overwrite position i into position i mod W)
  - variant Mixtral offers a **mixture of experts** (roughly 8 Mistral models)

In this section we'll look at four techniques:
1. key-value cache (KV cache)
2. rotary position embeddings (RoPE)
3. grouped query attention (GQA)
4. sliding window attention

Key-Value Cache



$$\mathbf{x}'_4 = \sum_{j=1}^{4} a_{4,j}\mathbf{v}_j$$

$$\mathbf{a}_4 = \text{softmax}(\mathbf{s}_4)$$

$$s_{4,j} = \mathbf{k}_j^T\mathbf{q}_4/\sqrt{d_k}$$

$$\mathbf{q}_j = \mathbf{W}_q^T\mathbf{x}_j$$

$$\mathbf{k}_j = \mathbf{W}_k^T\mathbf{x}_j$$

$$\mathbf{v}_j = \mathbf{W}_v^T\mathbf{x}_j$$

- At each timestep, we reuse all previous keys and values (i.e. we need to cache them)
- But we can get rid of the queries, similarity scores, and attention weights (i.e. we can let them fall out of the cache)

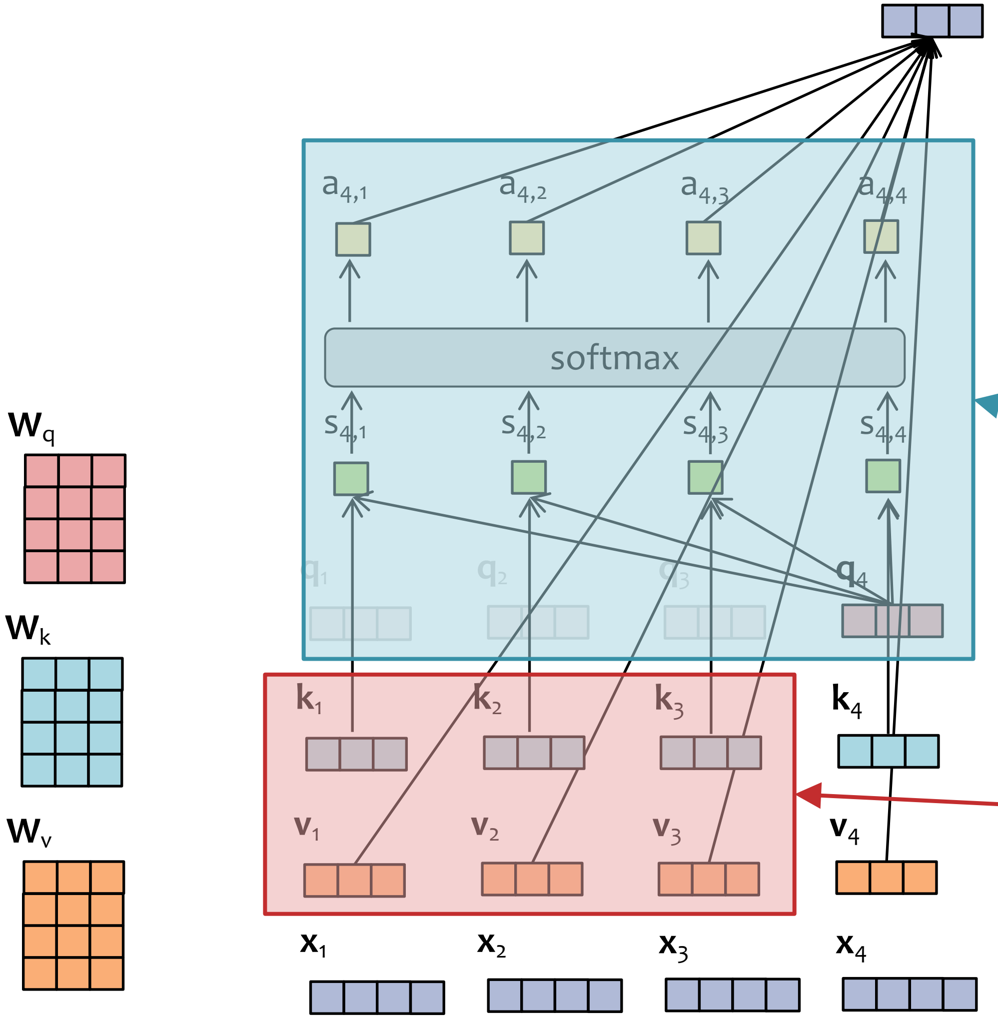
Discarded after this timestep

Computed for previous time-steps and reused for this timestep

Key-Value Cache

$$\mathbf{X}'_t = \mathbf{A}_t\mathbf{V} = \text{softmax}(\mathbf{Q}_t\mathbf{K}^T/\sqrt{d_k})\mathbf{V}$$

- At each timestep, we reuse all previous keys and values (i.e. we need to cache them)
- But we can get rid of the queries, similarity scores, and attention weights (i.e. we can let them fall out of the cache)



$$\mathbf{A}_t = \text{softmax}(\mathbf{S}_t)$$

$$\mathbf{S}_t = \mathbf{Q}_t\mathbf{K}^T/\sqrt{d_k}$$

$$\mathbf{Q}_t = \mathbf{X}_t\mathbf{W}_q$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}_k$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}_v$$

$$\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_t]^T$$

Discarded after this timestep

Computed for previous time-steps and reused for this timestep

ROTARY POSITION EMBEDDINGS (ROPE)

Rotary Position Embeddings (RoPE)

Q: Why does this slide have so many typos?

A: I'm really not sure. I very meticulously type up the latex for my slides myself and think carefully about all the things I put in them.

RoPE attention:

wrong

$$f_q(\mathbf{x}_t, m) \triangleq \mathbf{R}_{\Theta,\boxed{m}} \mathbf{W}_q^T \mathbf{x}_t$$

wrong

$$f_k(\mathbf{x}_j, m) \triangleq \mathbf{R}_{\Theta,\boxed{m}} \mathbf{W}_k^T \mathbf{x}_j$$

wrong

$$s_{t,j} = f_k(\mathbf{x}_j, \boxed{m})^T f_q(\mathbf{x}_t, \boxed{m}) / \sqrt{|\mathbf{k}|},$$

wrong

$$\forall j, t \text{ where } \boxed{m = t - j}$$

$$\mathbf{a}_t = \text{softmax}(\mathbf{s}_t), \forall t$$

where $\mathbf{W}_k, \mathbf{W}_q \in \mathbb{R}^{d_{model} \times d_k}$, and the rotary matrix $\mathbf{R}_{\Theta,m} \in \mathbb{R}^{d_k \times d_k}$ is given by:

$$R_{\Theta,m} = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d_k/2} & -\sin m\theta_{d_k/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d_k/2} & \cos m\theta_{d_k/2} \end{pmatrix}$$

The θ_i parameters are fixed ahead of time and defined as below:

wrong

$$\Theta = \{\theta_i = 10000^{\boxed{-2^{i-1}}/d}, i \in [1, 2, \ldots, d/2]\}$$

Rotary Position Embeddings (RoPE)

Q: Why does this slide have so many typos?

A: I'm really not sure. I very meticulously type up the latex for my slides myself and think carefully about all the things I put in them.

Rotary Position Embeddings (RoPE)

- Rotary position embeddings are a kind of **relative** position embeddings
- Key idea:
  - break each d-dimensional input vector into d/2 vectors of length 2
  - rotate each of the d/2 vectors by an amount scaled by m
  - m is the absolute position of the query or the key

1
2
3
4
5
6

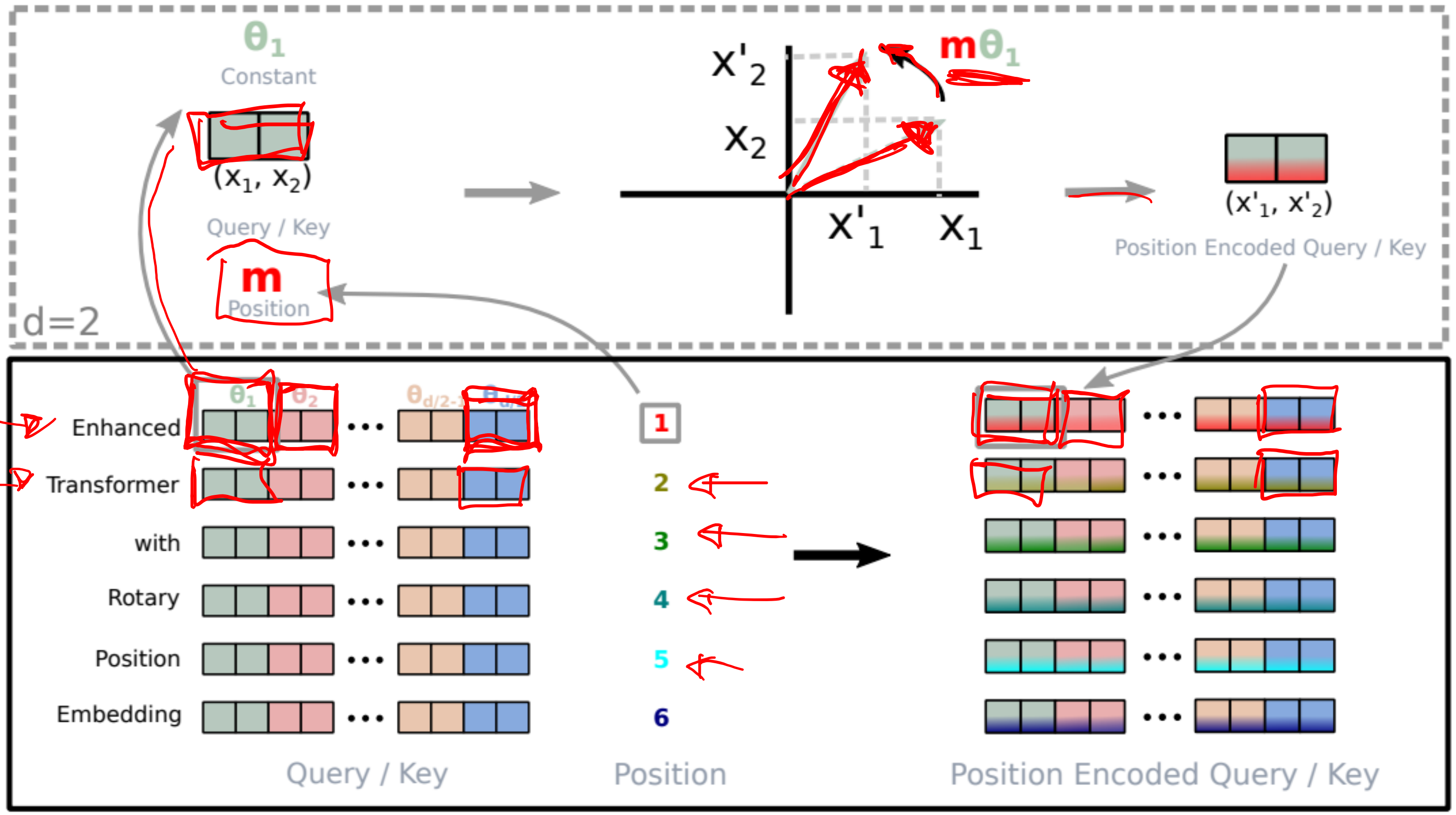


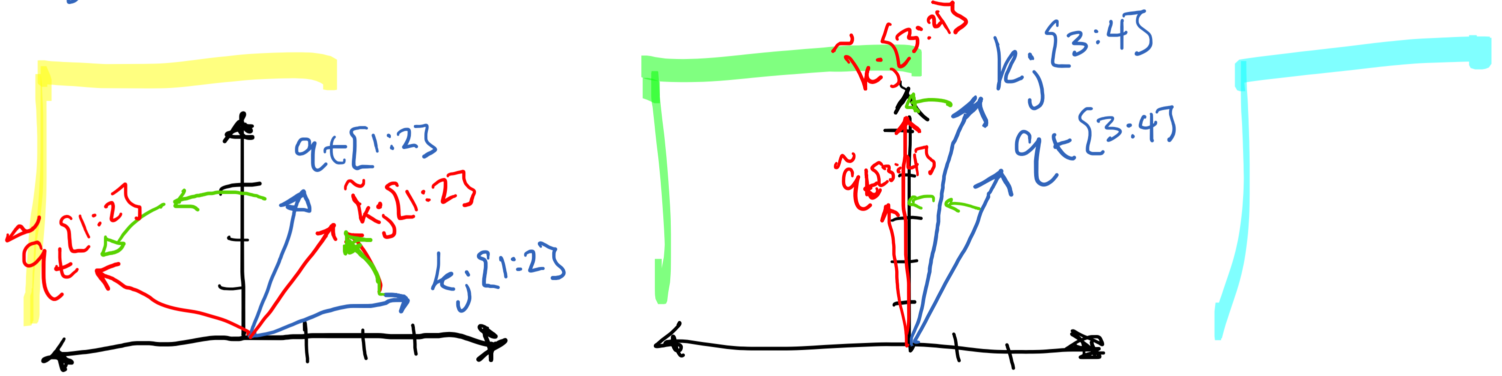
Figure from http://arxiv.org/abs/2104.09864

Rotary Position Embeddings (RoPE)

the	bird	flew	out
1	2	3	4

$q_t = W_q x_t = [1, 3, 2, 4, 4, 7]$ $\quad t = 4$

$k_j = W_k x_j = [3, 1, 2, 5, -2, 1]$ $\quad j = 2$



$\tilde{q}_t = f(q_t, t) = R_{\Theta,t}\, q_t$

$\tilde{k}_j = f(k_j, j) = R_{\Theta,j}\, k_j$

Rotary Position Embeddings (RoPE)

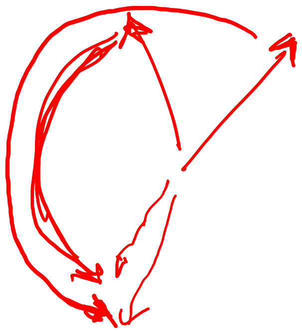
Standard attention:

$$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j, \forall j$$

$$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j, \forall j$$

$$s_{t,j} = \mathbf{k}_j^T \mathbf{q}_t / \sqrt{|\mathbf{k}|}, \forall j, t$$

$$\mathbf{a}_t = \text{softmax}(\mathbf{s}_t), \forall t$$



RoPE attention:

$$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j, \forall j \qquad \mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j, \forall j$$

$$\tilde{\mathbf{q}}_j = \mathbf{R}_{\Theta,j} \mathbf{q}_j \qquad \tilde{\mathbf{k}}_j = \mathbf{R}_{\Theta,j} \mathbf{k}_j$$

$$s_{t,j} = \tilde{\mathbf{k}}_j^T \tilde{\mathbf{q}}_t / \sqrt{d_k}, \forall j, t$$

$$\mathbf{a}_t = \text{softmax}(\mathbf{s}_t), \forall t$$

where $\mathbf{W}_k, \mathbf{W}_q \in \mathbb{R}^{d_{model} \times d_k}$. Herein we use $d = d_k$ for brevity.

For some fixed absolute position m, the rotary matrix $\mathbf{R}_{\Theta,m} \in \mathbb{R}^{d_k \times d_k}$ is given by:

$$R_{\Theta,m} = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d_k/2} & -\sin m\theta_{d_k/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d_k/2} & \cos m\theta_{d_k/2} \end{pmatrix}$$

The θ_i parameters are fixed ahead of time and defined as below.

$$\Theta = \{\theta_i = 10000^{-2(i-1)/d}, i \in [1, 2, \ldots, d/2]\}$$

Rotary Position Embeddings (RoPE)

Standard attention:

$$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j, \forall j$$
$$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j, \forall j$$
$$s_{t,j} = \mathbf{k}_j^T \mathbf{q}_t / \sqrt{|\mathbf{k}|}, \forall j, t$$
$$\mathbf{a}_t = \text{softmax}(\mathbf{s}_t), \forall t$$

RoPE attention:

$$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j, \forall j \qquad \mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j, \forall j$$
$$\tilde{\mathbf{q}}_j = \mathbf{R}_{\Theta,j} \mathbf{q}_j \qquad \tilde{\mathbf{k}}_j = \mathbf{R}_{\Theta,j} \mathbf{k}_j$$
$$s_{t,j} = \tilde{\mathbf{k}}_j^T \tilde{\mathbf{q}}_t / \sqrt{d_k}, \forall j, t$$
$$\mathbf{a}_t = \text{softmax}(\mathbf{s}_t), \forall t$$

Because of the block sparse pattern in $\mathbf{R}_{\theta,m}$, we can efficiently compute the matrix-vector product of $\mathbf{R}_{\theta,m}$ with some arbitrary vector $\mathbf{y}$ in a more efficient manner:

$$\mathbf{R}_{\Theta,m}\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_{d-1} \\ y_d \end{pmatrix} \odot \begin{pmatrix} \cos m\theta_1 \\ \cos m\theta_1 \\ \cos m\theta_2 \\ \cos m\theta_2 \\ \vdots \\ \cos m\theta_{d/2} \\ \cos m\theta_{d/2} \end{pmatrix} + \begin{pmatrix} -y_2 \\ y_1 \\ -y_4 \\ y_3 \\ \vdots \\ -y_d \\ y_{d-1} \end{pmatrix} \odot \begin{pmatrix} \sin m\theta_1 \\ \sin m\theta_1 \\ \sin m\theta_2 \\ \sin m\theta_2 \\ \vdots \\ \sin m\theta_{d/2} \\ \sin m\theta_{d/2} \end{pmatrix}$$

Matrix Version of RoPE

RoPE attention:

$$\mathbf{q}_j = \mathbf{W}_q^T \mathbf{x}_j, \forall j \qquad \mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j, \forall j$$

$$\tilde{\mathbf{q}}_j = \mathbf{R}_{\Theta,j} \mathbf{q}_j \qquad \tilde{\mathbf{k}}_j = \mathbf{R}_{\Theta,j} \mathbf{k}_j$$

$$s_{t,j} = \tilde{\mathbf{k}}_j^T \tilde{\mathbf{q}}_t / \sqrt{d_k}, \forall j, t$$

$$\mathbf{a}_t = \text{softmax}(\mathbf{s}_t), \forall t$$

Matrix Version:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_q \qquad \mathbf{K} = \mathbf{X}\mathbf{W}_k$$

$$\tilde{\mathbf{Q}} = g(\mathbf{Q}; \Theta) \qquad \tilde{\mathbf{K}} = g(\mathbf{K}; \Theta)$$

$$\mathbf{S} = \tilde{\mathbf{Q}}\tilde{\mathbf{K}}^T / \sqrt{d_k}$$

$$\mathbf{A} = \text{softmax}(\mathbf{S})$$

Goal: to construct a new matrix $\tilde{\mathbf{Y}} = g(\mathbf{Y}; \Theta)$ such that $\tilde{\mathbf{Y}}_{m,\cdot} = \mathbf{R}_{\Theta,m} \mathbf{y}_m$

$$\mathbf{C} = \left[\begin{array}{ccc|ccc} 1\theta_1 & \cdots & 1\theta_{\frac{d}{2}} & 1\theta_1 & \cdots & 1\theta_{\frac{d}{2}} \\ \vdots & & \vdots & \vdots & & \vdots \\ N\theta_1 & \cdots & N\theta_{\frac{d}{2}} & N\theta_1 & \cdots & N\theta_{\frac{d}{2}} \end{array}\right]$$

$$\begin{aligned} \tilde{\mathbf{Y}} &= g(\mathbf{Y}; \Theta) \\ &= \left[\ \mathbf{Y}_{\cdot,1:d/2} \mid \mathbf{Y}_{\cdot,d/2+1:d}\ \right] \odot \cos(\mathbf{C}) \\ &\quad + \left[\ -\mathbf{Y}_{\cdot,d/2+1:d} \mid \mathbf{Y}_{\cdot,1:d/2}\ \right] \odot \sin(\mathbf{C}) \end{aligned}$$

Matrix Version of RoPE

Q: Is this slide correct?

A: I'm really not sure.

But I did write it myself!

$$\mathbf{k}_j = \mathbf{W}_k^T \mathbf{x}_j, \forall j$$

$$\tilde{\mathbf{k}}_j = \mathbf{R}_{\Theta,j} \mathbf{k}_j$$

Matrix Version:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_q \qquad \mathbf{K} = \mathbf{X}\mathbf{W}_k$$

$$\tilde{\mathbf{Q}} = g(\mathbf{Q}; \Theta) \qquad \tilde{\mathbf{K}} = g(\mathbf{K}; \Theta)$$

$$\mathbf{S} = \tilde{\mathbf{Q}}\tilde{\mathbf{K}}^T / \sqrt{d_k}$$

$$\mathbf{A} = \text{softmax}(\mathbf{S})$$

Goal: to construct a new matrix $\tilde{\mathbf{Y}} = g(\mathbf{Y}; \Theta)$ such that $\tilde{\mathbf{Y}}_{m,\cdot} = \mathbf{R}_{\Theta,m} \mathbf{y}_m$

$$\mathbf{C} = \left[\begin{array}{ccc|ccc} 1\theta_1 & \cdots & 1\theta_{\frac{d}{2}} & 1\theta_1 & \cdots & 1\theta_{\frac{d}{2}} \\ \vdots & & \vdots & \vdots & & \vdots \\ N\theta_1 & \cdots & N\theta_{\frac{d}{2}} & N\theta_1 & \cdots & N\theta_{\frac{d}{2}} \end{array}\right]$$

$$\begin{aligned} \tilde{\mathbf{Y}} =& g(\mathbf{Y}; \Theta) \\ =& \left[\, \mathbf{Y}_{\cdot,1:d/2} \;\middle|\; \mathbf{Y}_{\cdot,d/2+1:d} \,\right] \odot \cos(\mathbf{C}) \\ &+ \left[\, -\mathbf{Y}_{\cdot,d/2+1:d} \;\middle|\; \mathbf{Y}_{\cdot,1:d/2} \,\right] \odot \sin(\mathbf{C}) \end{aligned}$$

RoPE

Pat's RoPE Demo:

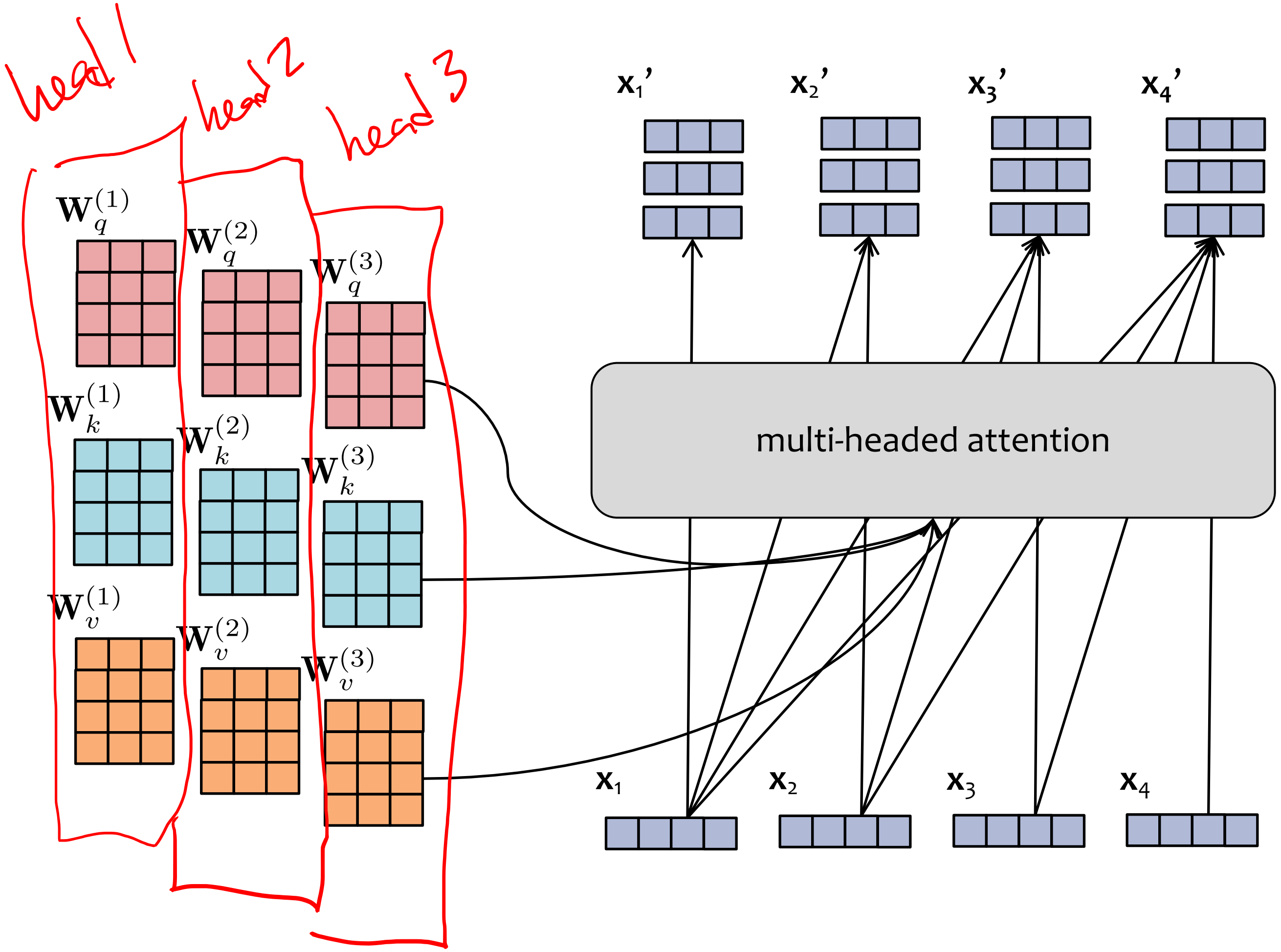
https://www.desmos.com/calculator/z1fuchfpej

- Two word embeddings represented as 2D vectors:
  - 1) cat
  - 2) ate
- We consider each one residing in a different position
- Each one is rotated by an amount given by theta

GROUPED QUERY ATTENTION (GQA)

Matrix Version of Multi-Headed (Causal) Attention

Recall...



$$\mathbf{X} = \text{concat}(\mathbf{X}'^{(1)}, \ldots, \mathbf{X}'^{(h)})$$

$$\mathbf{X}'^{(i)} = \text{softmax}\left(\frac{\mathbf{Q}^{(i)}(\mathbf{K}^{(i)})^T}{\sqrt{d_k}} + \mathbf{M}\right)\mathbf{V}^{(i)}$$

$$\mathbf{Q}^{(i)} = \mathbf{X}\mathbf{W}_q^{(i)}$$

$$\mathbf{K}^{(i)} = \mathbf{X}\mathbf{W}_k^{(i)}$$

$$\mathbf{V}^{(i)} = \mathbf{X}\mathbf{W}_v^{(i)}$$

$$\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_4]^T$$

Grouped Query Attention (GQA)

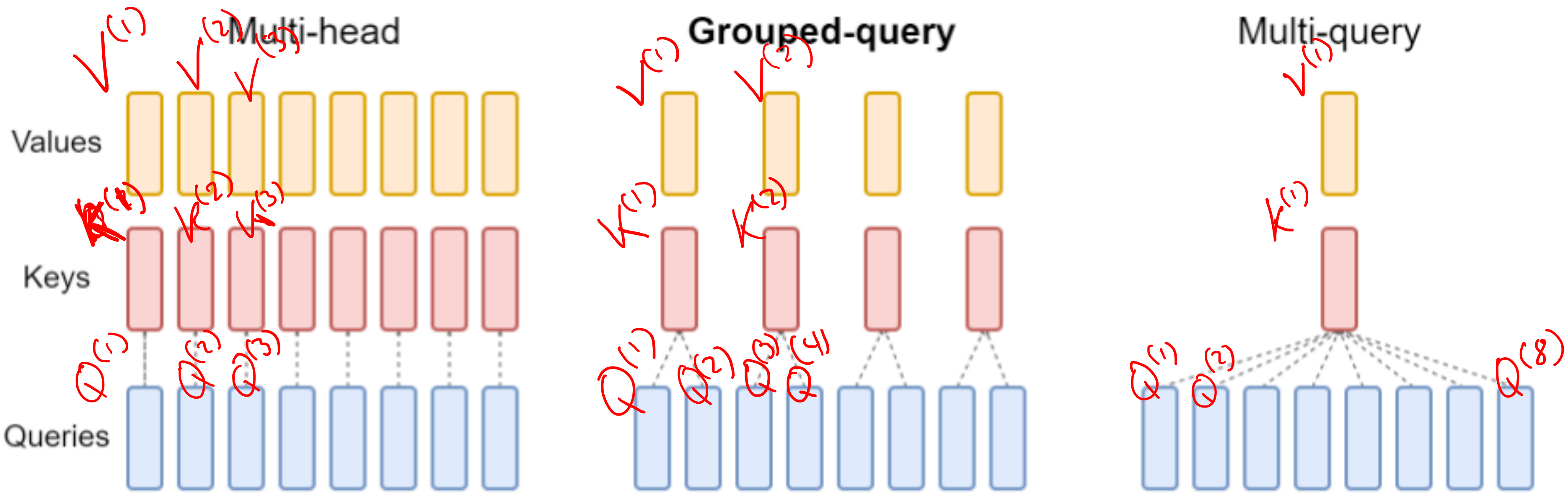
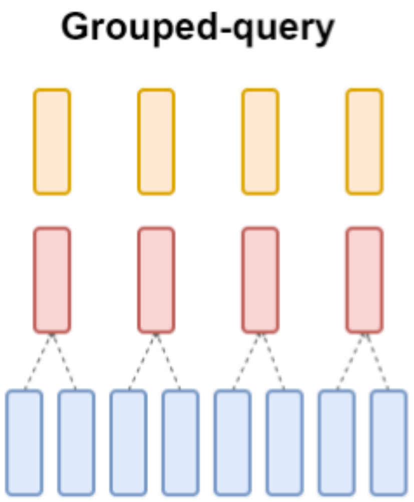


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between multi-head and multi-query attention.

Figure from http://arxiv.org/abs/2305.13245

Grouped Query Attention (GQA)



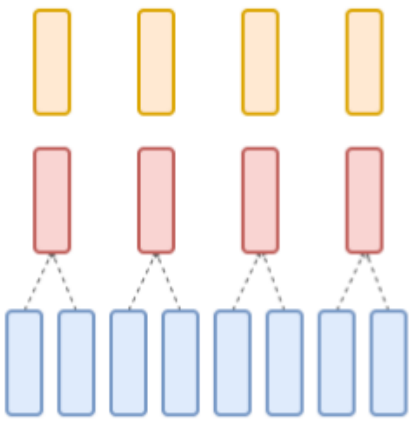
- **Key idea:** reuse the same key-value heads for multiple different query heads
- **Parameters:** The parameter matrices are all the same size, but we now have fewer key/value parameter matrices (heads) than query parameter matrices (heads)

$$\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_T]^T$$
$$\mathbf{V}^{(i)} = \mathbf{X}\mathbf{W}_v^{(i)}, \forall i \in \{1, \ldots, h_{kv}\}$$
$$\mathbf{K}^{(i)} = \mathbf{X}\mathbf{W}_k^{(i)}, \forall i \in \{1, \ldots, h_{kv}\}$$
$$\mathbf{Q}^{(i,j)} = \mathbf{X}\mathbf{W}_q^{(i,j)}, \forall i \in \{1, \ldots, h_{kv}\}, \forall j \in \{1, \ldots, g\}$$

- h_q = the number of query heads $h_q = 8$
- h_{kv} = the number of key/value heads $h_{kv} = 4$
- Assume h_q is divisible by h_{kv}
- $g = h_q / h_{kv}$ is the size of each group $g = 8/4 = 2$
  (i.e. the number of query vectors per key/value vector).

Figure from http://arxiv.org/abs/2305.13245

Grouped Query Attention (GQA)



- **Key idea:** reuse the same key-value heads for multiple different query heads
- **Parameters:** The parameter matrices are all the same size, but we now have fewer key/value parameter matrices (heads) than query parameter matrices (heads)

$$\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_T]^T$$
$$\mathbf{V}^{(i)} = \mathbf{X}\mathbf{W}_v^{(i)}, \forall i \in \{1, \ldots, h_{kv}\}$$
$$\mathbf{K}^{(i)} = \mathbf{X}\mathbf{W}_k^{(i)}, \forall i \in \{1, \ldots, h_{kv}\}$$
$$\mathbf{Q}^{(i,j)} = \mathbf{X}\mathbf{W}_q^{(i,j)}, \forall i \in \{1, \ldots, h_{kv}\}, \forall j \in \{1, \ldots, g\}$$

$$\mathbf{S}^{(i,j)} = \mathbf{Q}^{(i,j)}(\mathbf{K}^{(i)})^T / \sqrt{d_k}, \quad \forall i \in \{1, \ldots, h_{kv}\}, \forall j \in \{1, \ldots, g\}$$
$$\mathbf{A}^{(i,j)} = \text{softmax}(\mathbf{S}^{(i,j)}), \quad \forall i \in \{1, \ldots, h_{kv}\}, \forall j \in \{1, \ldots, g\}$$
$$\mathbf{X}'^{(i,j)} = \mathbf{A}^{(i,j)}\mathbf{V}^{(i)}, \quad \forall i \in \{1, \ldots, h_{kv}\}, \forall j \in \{1, \ldots, g\}$$
$$\mathbf{X}' = \text{concat}(\mathbf{X}'^{(i,j)}), \quad \forall i \in \{1, \ldots, h_{kv}\}, \forall j \in \{1, \ldots, g\}$$
$$\mathbf{X} = \mathbf{X}'\mathbf{W}_o \qquad \text{(where } \mathbf{W}_o \in \mathbb{R}^{d_{model} \times d_{model}}\text{)}$$

Figure from http://arxiv.org/abs/2305.13245

SLIDING WINDOW ATTENTION

Sliding Window Attention

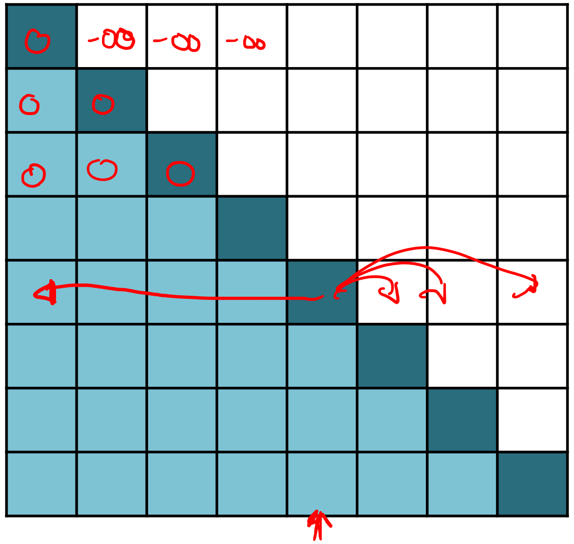
Sliding Window Attention

- also called "local attention" and introduced for the Longformer model (2020)
- **The problem:** regular attention is computationally expensive and requires a lot of memory
- **The solution:** apply a causal mask that only looks at the include a window of (½w+1) tokens, with the rightmost window element being the current token (i.e. on the diagonal)

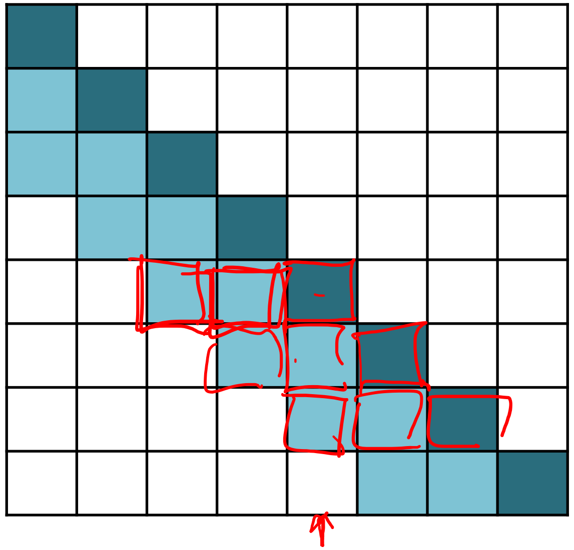
$$\mathbf{X}' = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}} + \mathbf{M}\right)\mathbf{V}$$

M =

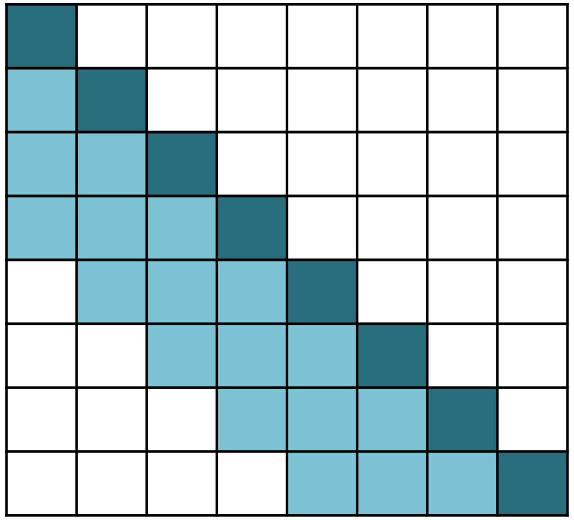
regular causal attention



sliding window attention (w=4)



sliding window attention (w=6)



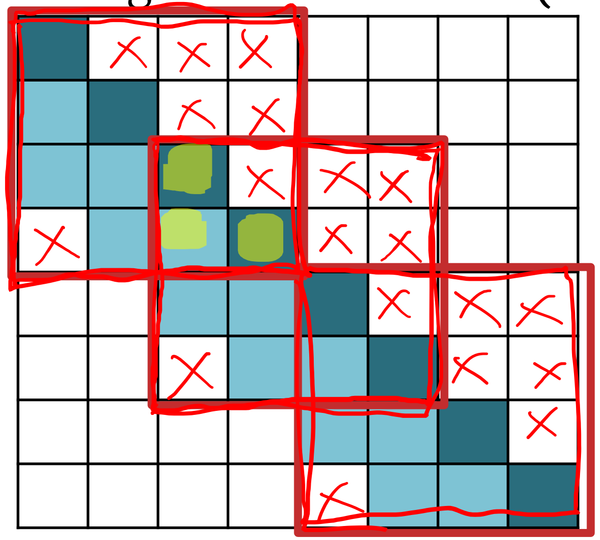
Sliding Window Attention

Sliding Window Attention

- also called "local attention" and introduced for the Longformer model (2020)
- **The problem:** regular attention is computationally expensive and requires a lot of memory
- **The solution:** apply a causal mask that only looks at the include a window of (½w+1) tokens, with the rightmost window element being the current token (i.e. on the diagonal)

$$\mathbf{X}' = \text{softmax}\left(\frac{\mathbf{QK}^T}{\sqrt{d_k}} + \mathbf{M}\right)\mathbf{V}$$

sliding window attention (w=4)



3 ways you could implement

1. *naïve implementation:* just do the matrix multiplication, but this is still slow
2. *for-loop implementation:* asymptotically faster / less memory, but unusable in practice b/c for-loops in PyTorch are too slow
3. *sliding chunks implementation:* break into Q and K into chunks of size w x w, with overlap of ½w; then compute full attention within each chunk and mask out chunk (very fast/low memory in practice)