Scaling Up Part II: Mixture of Experts

Scaling Up

- Last lecture, we learnt that scaling up the language model can
  - Improve the model's capacity
  - Make the model training converge faster (in terms of number of flops)
- Downside: During inference time, the cost is much higher.

Reducing Inference Cost

- In the these two lectures, we will learn multiple techniques to reduce inference cost:
- This lecture: Mixture of Experts model architecture (more efficient on the MLP layer)
- Next lecture: KV caching, VLM and paged attention (more efficient on the attention layer)

Mixture of Experts

- GPT-4: 8x200B mixture of expert model.
- GPT-3.5: 8x20B mixture of expert model.

Motivation

- Human knowledge is “sparse”
  - The model is an AGI, but
  - When we ask the model “write me an essay about the history of CMU.”
    - The model only need to extract a tiny fraction of the knowledge it stored, centered around CMU.
- We want to make the model more “inference efficient” by only using a “tiny fraction of model” for each prompt (the exact fraction differ for every prompt).

“Sparsified Inference”

- How do we make sure that for every prompt, only a tiny fraction of the model weights are “used” to increase efficiency?
- The mixture of expert architecture.

Mixture of Experts: Definition

- A mixture of Expert layer with M experts, top-k routing is defined as the following:
- Given input x in R^d, the output of MoE(x) is computed as:
  - (1). Compute r = Router(x) in R^M, where Router is a linear function.
  - (2). Compute s = softmax(r).
    - And sk in R^M such that sk[i] = 1 if i is in the top-k largest entry of s, otherwise sk[i] = 0.
    - If k > 1, compute s'[i] = s[i] sk[i] / sum_j s[j] sk[j] (expert normalization). Otherwise s' = s.
  - (3). Compute Expert_i(x) for each i such that sk[i] = 1, where Expert_i is a MLP layer.
  - (4). Compute MoE(x) = sum_i Expert_i(x) * sk[i] * s'

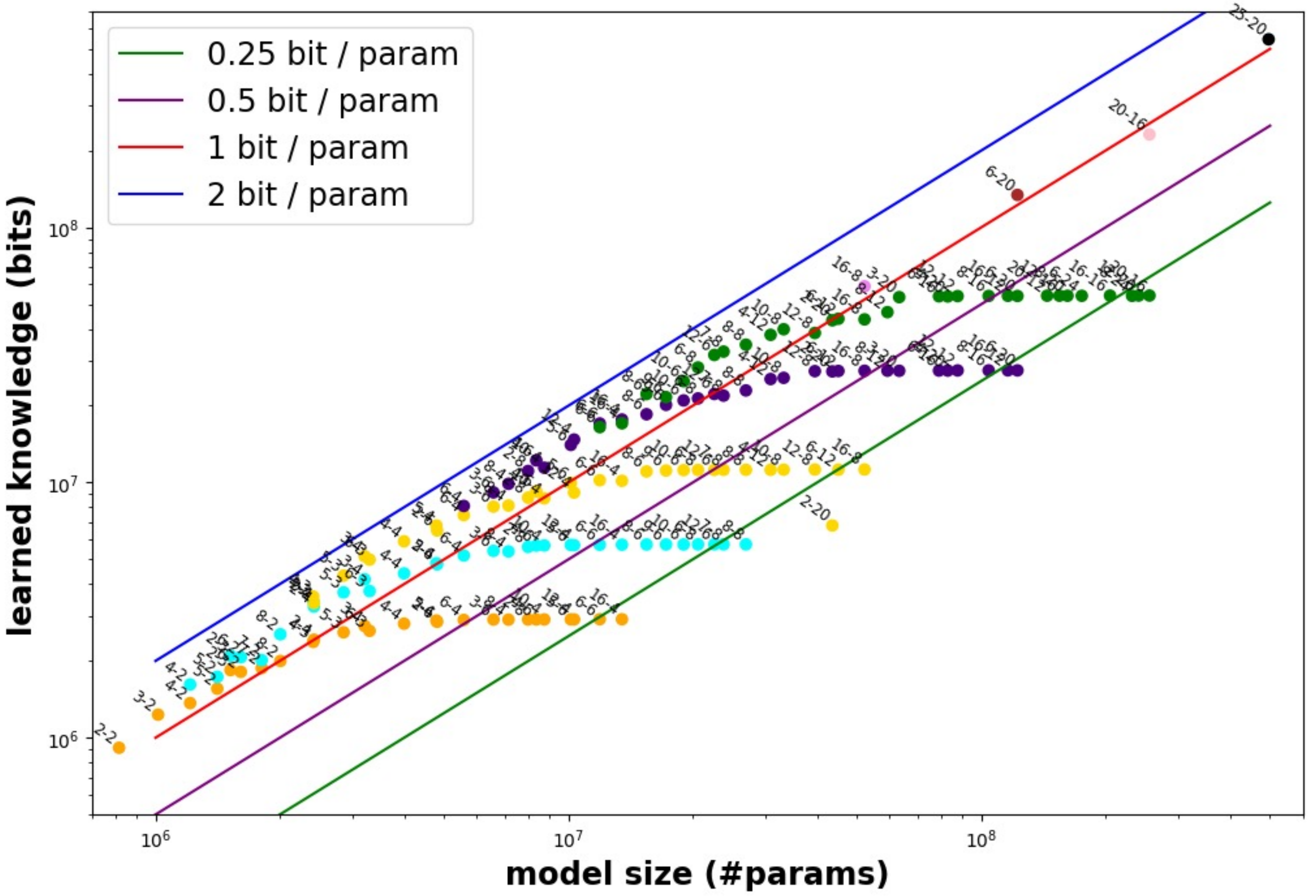
Mixture of Experts: Usage

- In transformer, using MoE layer is simple:
- We just replace the MLP layer in the transformer block with MoE layer
- A typical choice (used by OpenAI MoE model) for the MoE layer is the following:
  - Top-2 routing, M = 16
- Each MoE layer is a MLP of the following architecture:
  - Linear (d -> 2d) -> GeLU -> Linear (2d -> d).
- For each x, the “active parameter” in the MLP layer is 8d^2, the same as traditional transformer (a single MLP of shape d->4d -> d)

MoE: Effective versus Total parameter

- We typically use the MoE layer to replace the MLP layer in a transformer block
- Total parameter of an MoE transformer:
  - The total number of parameters in the network.
  - Which is
    - parameters in embedding layer/LM-head
    - Parameters in the attention layer
    - Parameters in all the experts (M * parameter per expert)
- Effective parameter of an MoE transformer:
  - The maximum number of parameters that are “activated” for each token.
    - parameters in embedding layer/LM-head
    - Parameters in the attention layer
    - Parameters in activated the experts (k * parameter per expert)

Mixture of Expert: Scaling Law (1--1.5 bit /total parameter, 32 experts)



Mixture of Expert: Scaling Law

- 1--1.5 bit /total parameter, 32 experts
  - Effective parameter when using 32 experts: 1/11 of the total parameter.
    - (parameter that’s activate per token, roughly equal to inference cost)
  - Effective parameters = 1/11 total parameters
- > 5x more efficient than Dense model!!!

Mixture of Expert: Actual implementation (training)

- Main Challenge: For each token x, it can use different experts.
- We don’t want to compute expert_i(x) for all i for each x, we want to compute the set of expert_i differently for each x (the experts that are “activate” for this x)

Fast Encode + Fast Decode

- Fast Encode operation:
- Given a sequence N of vectors x_j (each in R^d), we first compute $sk(x_j)$ for each j.
- Then we reshape the input from size N x d to size
  - M x N' x d.
    - Where N' = N * k * capacity_factor / M
    - For each i in [M], the corresponding entry N' x d is the collection of x_j such that $sk(x_j)[i] = 1$.
- Then we apply expert_i on the N' x d vectors.

Expert Parallel

- Typically , MoEs are trained with Expert Parallel
- Meaning that in one layer, each expert weight is stored in different GPU nodes.
- Fast Encode
  - N x d -> M x N’ x d
- Then we can send each N’ x d tensor to each GPU node and compute the forward/backward for that expert.

Token Dropping

- Fast Encode
  - N x d -> M x N' x d
  - Where N' = N * k * capacity_factor / M
  - Usually, capacity_factor is set as 1.1 -- 1.25.
- However, if all x_j uses the same set of i as "activate" experts, then N' might not be able to contain all the x_j (token dropping)
- We want x_j to use the experts "uniformly".

Load Balancing Loss

- When training an MoE, we add the following load-balancing loss:
- Load-Balancing Loss = sum_{i in [M]} f_i * p_i
- f_i = fraction of tokens x_j such that sk(x_j)[i] = 1
  - Fraction of tokens x_j that uses expert i.
- p_i = sum_j s(x_j)[i]
  - The total probability assigned to the expert i (before normalization).
- Observation: Load balancing loss is minimized if
  - The routing is uniform
  - The probability is uniform
  - To see this: Note that sum f_i is constant, and sum p_i is constant.