Inference, Deployment, and Compression

CS4787/5777 Lecture 23 — Fall 2025

Review: Why we should care about inference

- **Train once, infer many times**
  - Many production machine learning systems just do inference
- Often want to run inference on **low-power edge devices**
  - Such as cell phones, security cameras
  - **Limited memory** on these devices to store models
- Need to get **responses to users quickly**
  - On the web, users won't wait more than a second

Review: Metrics for Inference

- Important metric: **accuracy**
  - Inference accuracy can be **close to test accuracy** — if data from same distribution
- Important metric: **throughput**
  - **How many examples** can we classify in some amount of time
- Important metric: **latency**
  - **How long** does it take to get a prediction for a single example
- Important metric: **model size**
  - **How much memory** do we need to store/transmit the model for prediction
- Important metric: **energy use**
  - **How much energy** do we use to produce each prediction
- Important metric: **cost**
  - How much money will all this cost us

Tradeoffs

- When designing an ML system for inference, there are **trade-offs** among all these metrics!
  - Most “techniques” do not give free improvements, but have some trade-off where some metrics get better and others get worst
- There is **no one-size-fits-all “best” way to do ML inference**.
- We need to decide **which metric we value the most**
  - Then keep that in mind as we design the system

Improving the performance of inference

What tools do we have in our toolbox?

Altering the batch size

- Just like with learning, we can **make predictions in batches**
- Increasing the batch size helps **improve parallelism**
  - Provides more work to parallelize and an additional dimension for parallelization
  - This improves **throughput**
- But increasing the batch size can make us do more work before we can return an answer for any individual example
  - Can **negatively affect latency**

Neural Network Compression

- Find an **easier-to-compute network** with similar accuracy
  - Or find a network with **smaller model size**, depending on the goal

- Most compression methods are **lossy**, meaning that the compressed network may sometimes predict differently

- **Many techniques** for doing this
  - We'll see some in the following slides

Simple Technique: “Old-School” Compression

- Just apply a standard lossless compression technique to the weights of your neural network.
  - **Huffman coding** works here, for example.
  - Even something very general like **gzip** can be beneficial.
- This **lowers the stored model size without affecting accuracy**
- But this does mean we **need to decompress eventually**, so it comes at the cost of some compute & can affect start-up latency.

Problem with Lossless Compression

- Something like gzip or bz2 doesn't know the difference between **useful** and **useless** information
  - It preserves everything losslessly

- Low-order bits of neural network weights are like a random signal that contains little useful information
  - Not only are they unnecessary, they're **hard to compress**!

- Compressor can't distinguish **important** from **unimportant** weights—needs to preserve both!

Low-precision arithmetic for inference

- Very simple: just **use low-precision arithmetic in inference**
  - This discards the low-usefulness low-order bits of original format

- Can make any signals in the model low-precision

- Simple **heuristic for compression**: keep lowering the precision of signals until the accuracy decreases
  - Can often get down below 16 bit numbers with this method alone

- **Binarization** is low-precision arithmetic in the extreme
  - Even some hardware support with binary TensorCores

Models in low-precision

openai/**gpt-oss-20b**

Tensors	Shape	Precision
model.layers.0 (4)		
model.layers.0.input_layernorm.weight	[2 880]	BF16
model.layers.0.mlp (2)		
model.layers.0.mlp.experts (6)		
model.layers.0.mlp.experts.down_proj_bias	[32, 2 880]	BF16
model.layers.0.mlp.experts.down_proj_blocks	[32, 2 880, 90, 16]	U8
model.layers.0.mlp.experts.down_proj_scales	[32, 2 880, 90]	U8
model.layers.0.mlp.experts.gate_up_proj_bias	[32, 5 760]	BF16
model.layers.0.mlp.experts.gate_up_proj_blocks	[32, 5 760, 90, 16]	U8
model.layers.0.mlp.experts.gate_up_proj_scales	[32, 5 760, 90]	U8
model.layers.0.mlp.router (2)		
model.layers.0.mlp.router.bias	[32]	BF16
model.layers.0.mlp.router.weight	[32, 2 880]	BF16
model.layers.0.post_attention_layernorm.weight	[2 880]	BF16
model.layers.0.self_attn (5)		
model.layers.0.self_attn.k_proj (2)		
model.layers.0.self_attn.k_proj.bias	[512]	BF16
model.layers.0.self_attn.k_proj.weight	[512, 2 880]	BF16
model.layers.0.self_attn.o_proj (2)		
model.layers.0.self_attn.o_proj.bias	[2 880]	BF16

Mixed-precision inference

- Activation quantization also important!
  - **What are the tradeoffs?**

- We often see models described as **W4A16** or **W4A4**
  - This notation describes how many bits of precision are used for the weights and how many are used for the activations

- Where do we often use higher precision weights?
  - In softmax
  - In attention layers
  - Initial embedding layer **(why?)**

Post-Training Quantization

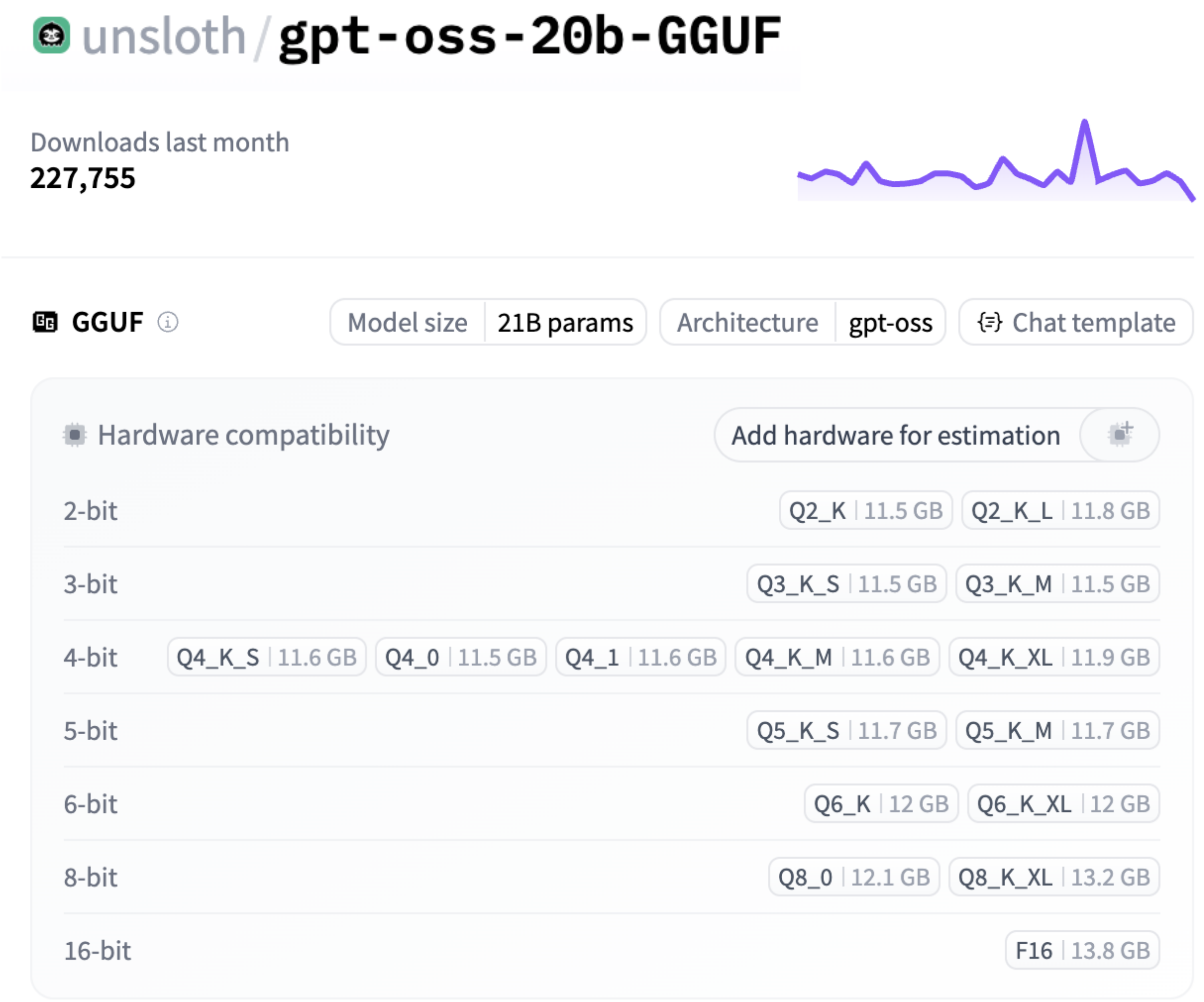
- Pretrain a model in high-precision, fine-tune it to task, then **quantize it afterwards** to the format we need

- Many techniques for this!
  - Typical to use a **development set** to gather activation statistics then compress based on those statistics
  - For example, **adaptive rounding** estimates the second moment matrix of activations entering a linear layer, then uses that matrix to decide how to round
  - Data-free methods eschew a development set

Post-Training Quantization: Open Source

- It's now become popular to do this and release quantized models online in a variety of precisions
  - Targets **efficient local inference**

- You don't have to quantize the model yourself anymore



Quantization Aware Training (QAT)

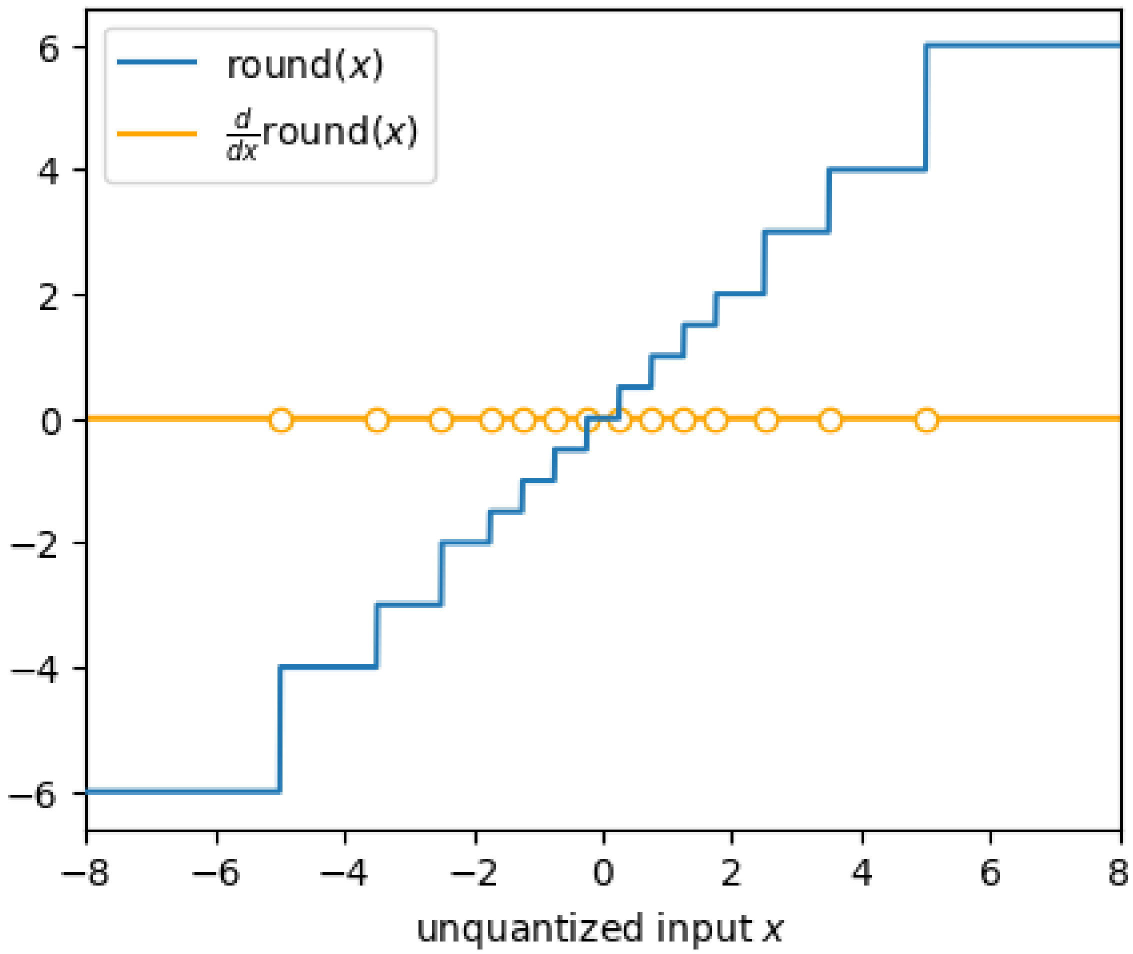
- Train a model in with **low-precision weights**
  - No need to quantize afterwards
- Almost always we keep the optimizer states (e.g. Adam weight buffer, momentum buffer, second moment buffer) in **higher precision during training**
  - The weights used in the forward pass are a quantized version of the weights stored in the optimizer

Quantization Aware Training (QAT)

- How do we take derivatives?

$$\text{minimize:} \quad \ell(\text{round}(w))$$

- If we just math it out, the derivative of a rounding operation is mostly 0!

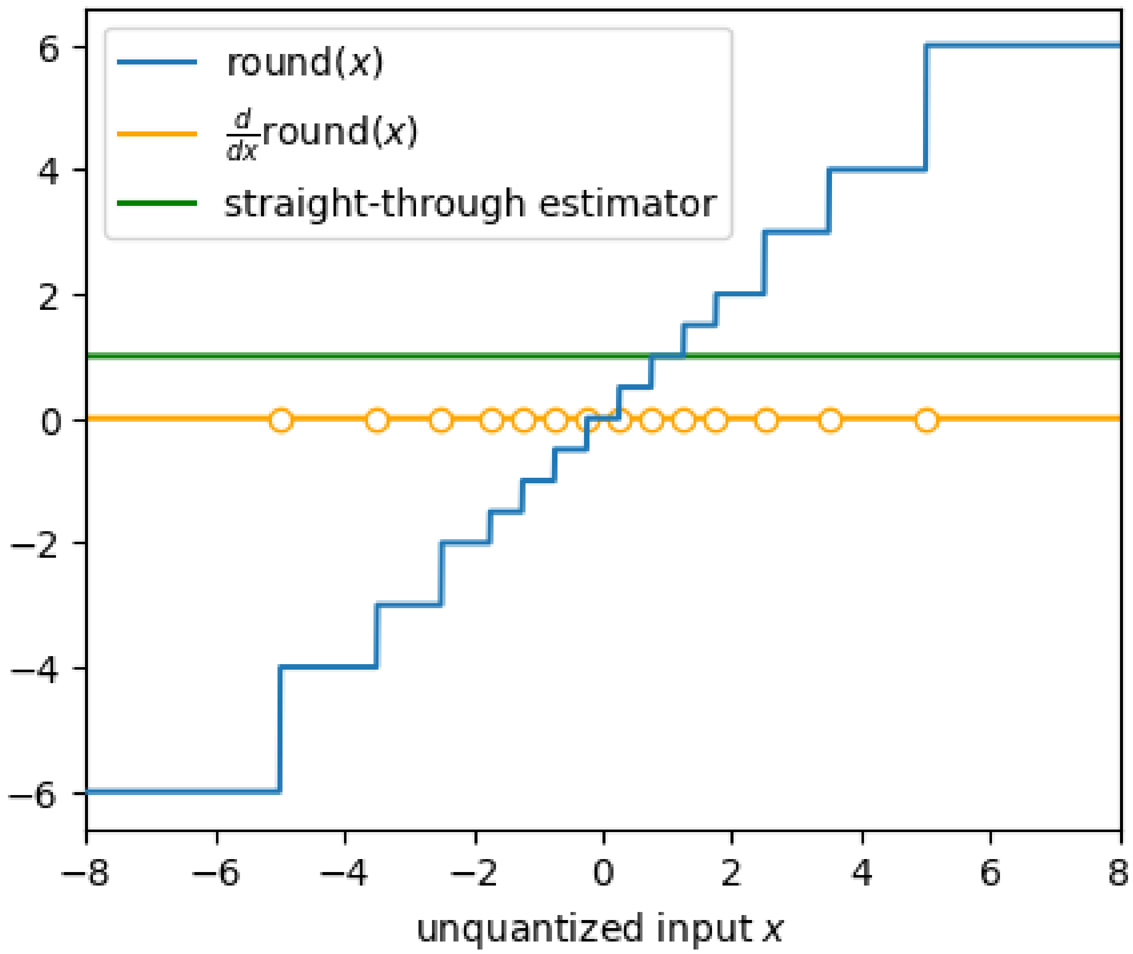


Quantization Aware Training (QAT)

- How do we take derivatives?

$$\text{minimize:} \quad \ell(\text{round}(w))$$

- **Straight-through-estimator**
    - Just imagine the rounding wasn't there when we do the forward pass



Quantization Aware Training (QAT)

- Don’t mix this up with using low-precision arithmetic during training to train more efficiently!

- QAT is about producing a **quantized model for downstream inference use**
  - It might also be more efficient to train, which would be good
  - But that’s incidental to the method

Pruning

- **Remove activations** that are usually zero
  - Or that don't seem to be contributing much to the model
  - Good heuristic: remove **the smallest X% of weights**

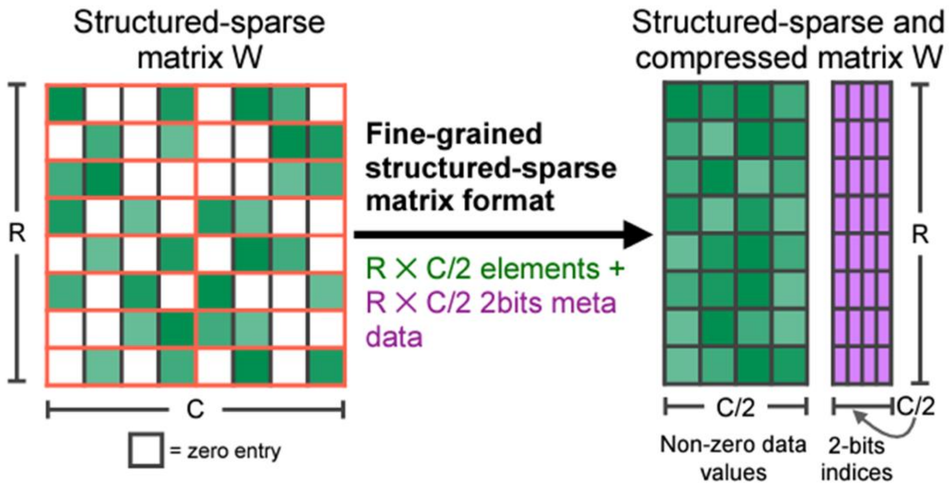
- Effectively creates **a smaller model**
  - This makes it easy to retrain, since we're just training a smaller network

- There's always the question of whether training a smaller model in the first place would have been as good or better.
  - But usually pruning is observed to produce benefits.

Types of Pruning

- **Neuron pruning**
  - Remove whole rows and columns from weight matrices
- **Unstructured weight pruning**
  - Remove weights, making them 0
  - Results in a **sparse model**
- **Structured weight pruning**
  - E.g. 2:4 sparsity where 2 of every block of 4 weights must be 0

https://developer.nvidia.com/blog/structured-sparsity-in-the-nvidia-ampere-architecture-and-applications-in-search-engines/

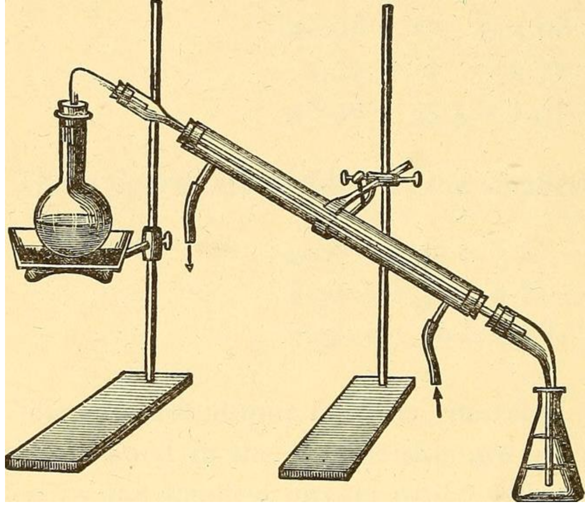


Fine-Tuning after Compression

- Powerful idea: apply a lossy compression operation, then **retrain the model** to improve accuracy

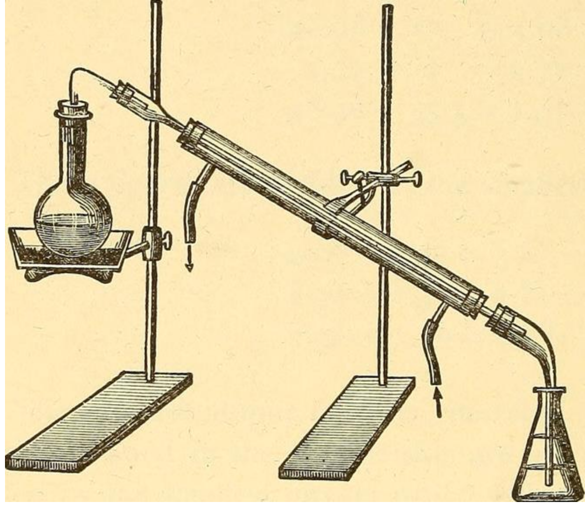
- A general way of "getting back" accuracy lost due to lossy compression.

Knowledge distillation



- Idea: take a large/complex model (the teacher) and **train a smaller network** (the student) to match its output
  - Hinton et. al. “Distilling the Knowledge in a Neural Network.”
- Often used for distilling **ensemble models** into a single network
  - Ensemble models average predictions from multiple independently-trained models into a single better prediction
  - Ensembles often **win Kaggle competitions**
- Can also **improve the accuracy** in some cases.

Knowledge distillation



- Idea: take a large/complex model and **train a smaller network to match its output**
  - E.g. Hinton et. al. "Distilling the Knowledge in a Neural Network."

- In **generative AI**, it's often used to improve smaller student models in a pretrained model family by using larger models in the same family as the teacher
  - E.g. Gemini-Nano was trained by distilling from larger Gemini models.

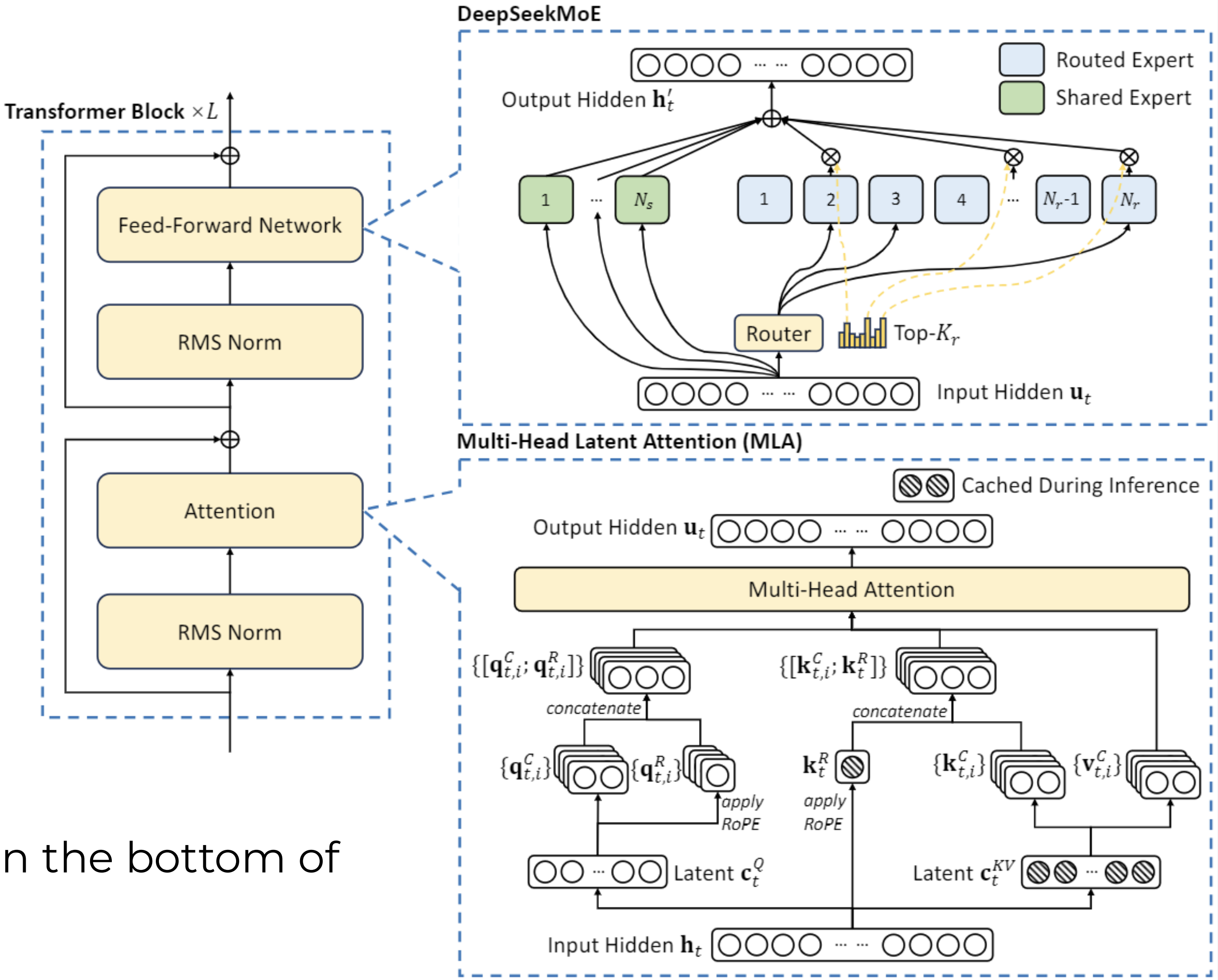
Efficient architectures

- Some neural network architectures are **designed to be efficient at inference time**
  - Early examples: MobileNet, ShuffleNet, SqueezeNet
  - Common for vision
- Networks are often based on sparsely connected neurons
  - This limits the number of weights which makes models smaller and easier to run inference on
- To be efficient, we can just **train one of these networks in the first place** for our application.

Efficient architectures: Mixture of Experts

https://github.com/deepseek-ai/DeepSeek-V2/tree/main

- Use only some weights in the MLP block for each token
- Choose which weights to use with a router layer

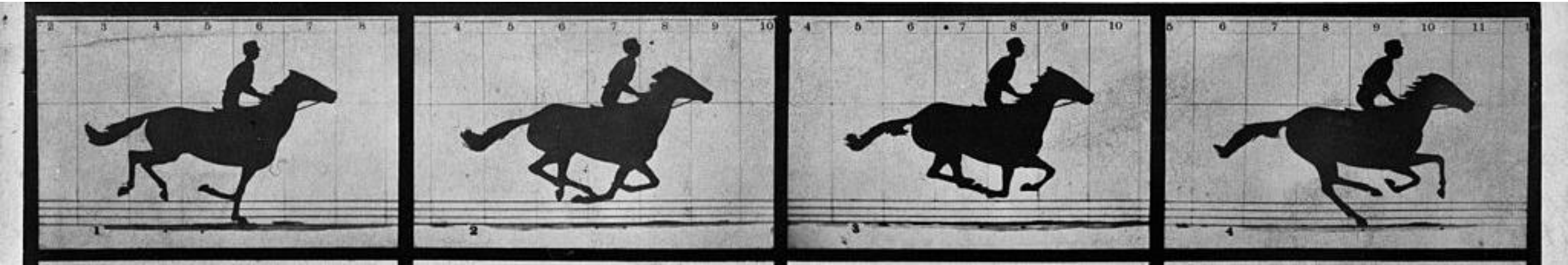


(Also note the cool KV cache compression on the bottom of this figure via a low-rank decomposition)

Re-use of computation

- For video and time-series data, there is a lot of **redundant information** from one frame to the next.

- We can try to **re-use some of the computation** from previous frames.
  - This is less popular than some of the other approaches here, because it is not really general.



Re-use of computation: KV Cache

- The **KV cache** is the <u>ur</u>-example of this
  - Reuses computation to process later tokens in the sequence
  - Attention operation is designed for this!

- Can also target KV cache with other compression methods we've already seen
  - KV cache quantization
  - Sparse KV caching

Speculative Decoding

- Large language model inference latency is bottlenecked by memory bandwidth
  - Need to read the whole model to make each token prediction

- **Can we get around this bottleneck and infer faster?**

- Idea: **generate multiple tokens at a time** from a lightweight "draft" model and then check them with the big model
  - Only requires one forward pass of the big model

- **What are the tradeoffs of this method?**

The last resort for speeding up DNN inference

- **Train another, faster type of model** that is not a deep neural network
  - For some real-time applications, you can't always use a DNN
- If you can get away with **a linear model**, it's almost always much faster.
- Also, **decision trees** tend to be quite fast for inference.
- ...but with how technology is developing, we're **seeing more and more support for fast DNN inference** (especially on edge hardware) so this will become less necessary.

Where do we run inference?

Inference on the cloud

- Most inference today is run on **cloud platforms**
- The cloud supports **large amounts of compute**
  - And makes it easy to access it and make it reliable
- This is a good place to put AI that needs to think about data
- For interactive models, **latency** is critical

Inference in the AI cloud

- **AI cloud platforms** sell generations, fine-tuning, predictions, etc. by the token
- This is the place you can get predictions from major proprietary models
  - E.g. ChatGPT
- Lots of requests means **high throughput**!
  - Often lower cost

Inference on edge devices

- Inference can run on your **laptop or smartphone**
  - Here, the size of the model becomes more of an issue
  - Limited smartphone memory
- This is good for **user privacy and security**
  - But not as good for companies that want to keep their models private
- Also can be used to deploy **personalized models**

Inference on sensors

- Sometimes we want **inference right at the source**
  - On the sensor where data is collected

- Example: a surveillance camera taking video
  - Don’t want to stream the video to the cloud, especially if most of it is not interesting.

- **Energy use** is very important here.