**Optimization Ideas for LLMs**

**Overview:**

* **Why Optimize Large Language Models?**
* **Model compression**

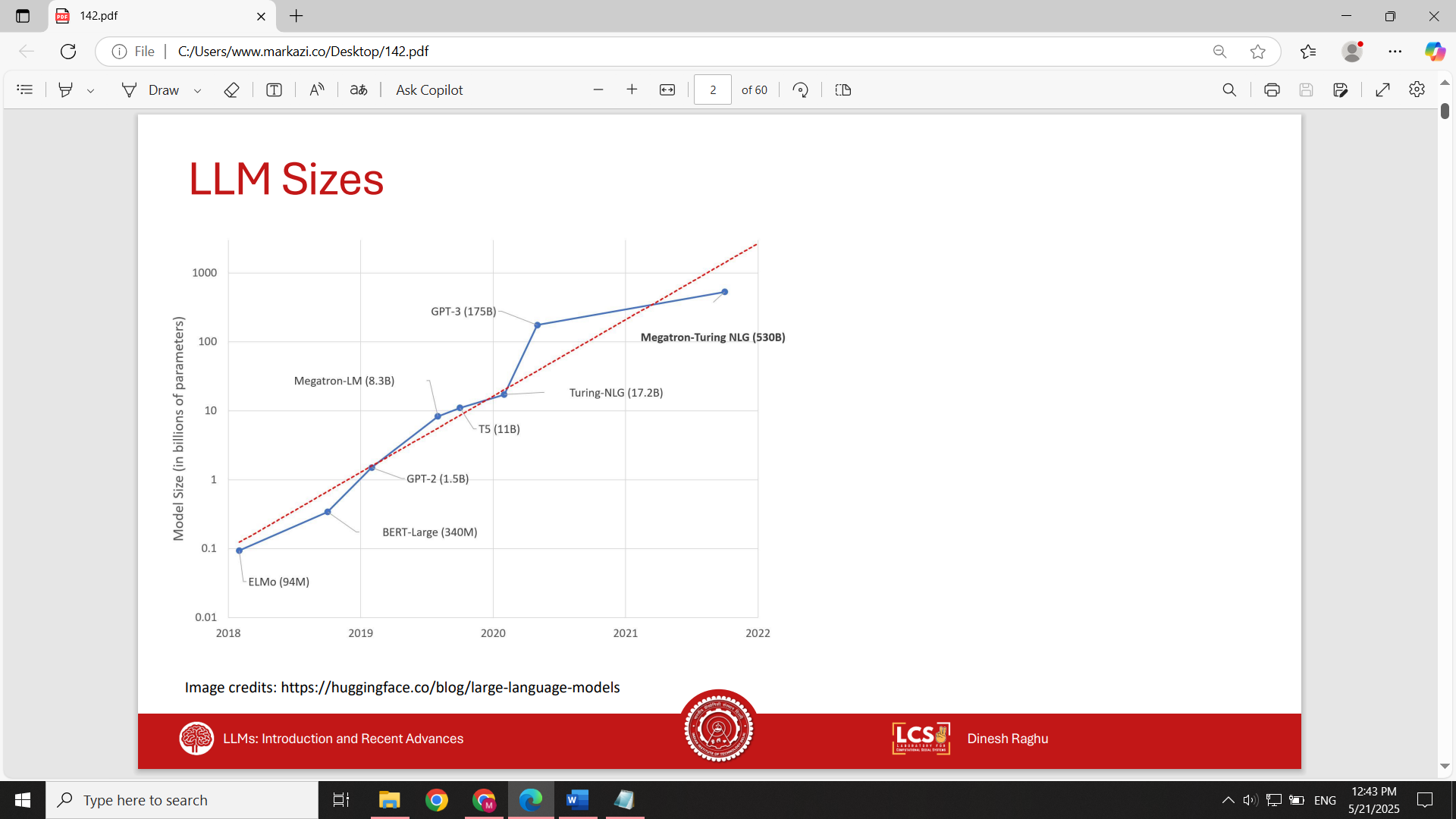
1. **Quantization**
2. **Pruning**
3. **Distillation**

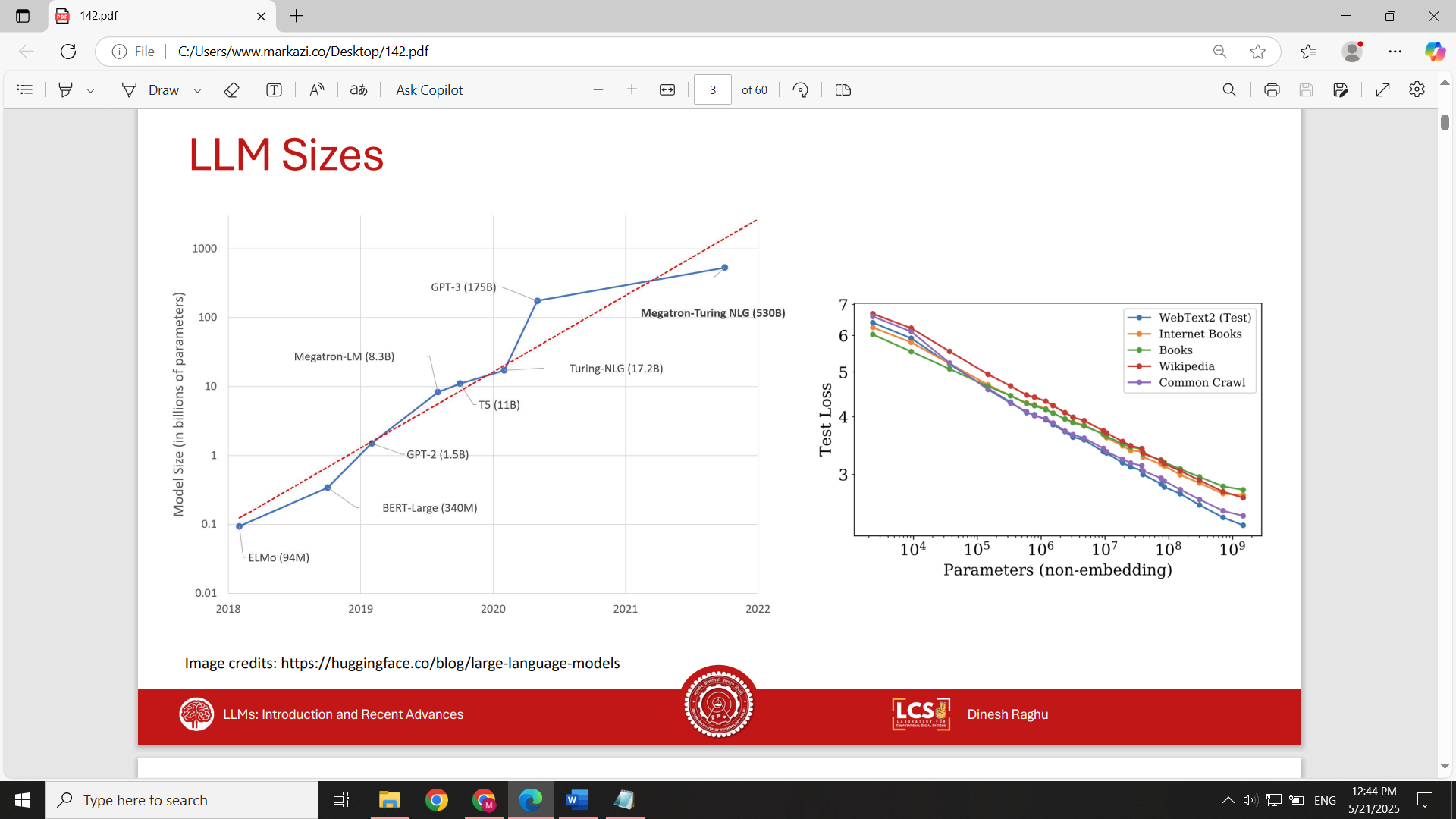
* **Mixture of experts**

**Why Optimize Large Language Models?**

1. **Growing Model Sizes and Performance**

Over the past few years, the size of large language models (LLMs) has been increasing exponentially. As the number of parameters in these models grows, we also see significant improvements in performance metrics, such as lower test loss on standard language modeling benchmarks. This trend highlights the growing capability of LLMs to perform complex tasks. However, this performance boost comes at the cost of increased resource demands, making deployment and widespread adoption more difficult.





1. **Hardware Limitations**

As models grow larger, they demand more powerful hardware for both training and inference. This means organizations often need to invest in expensive GPUs, which significantly increases costs. For many smaller organizations or startups, the requirement for advanced hardware can become a major barrier, preventing them from leveraging state-of-the-art LLMs.

1. **Inference Latency**

Larger models introduce higher inference latency, which negatively affects real-time applications. In scenarios like chatbots, long response times—such as 30 to 40 seconds—are unacceptable in a world where users expect instant interaction. This problem becomes worse when a single task may involve multiple LLM calls. For example, a model might first generate code, then another model might execute and evaluate the result before producing the final output. Each step adds to the latency, making the end-to-end process slow and less user-friendly.

1. **Inference Cost**

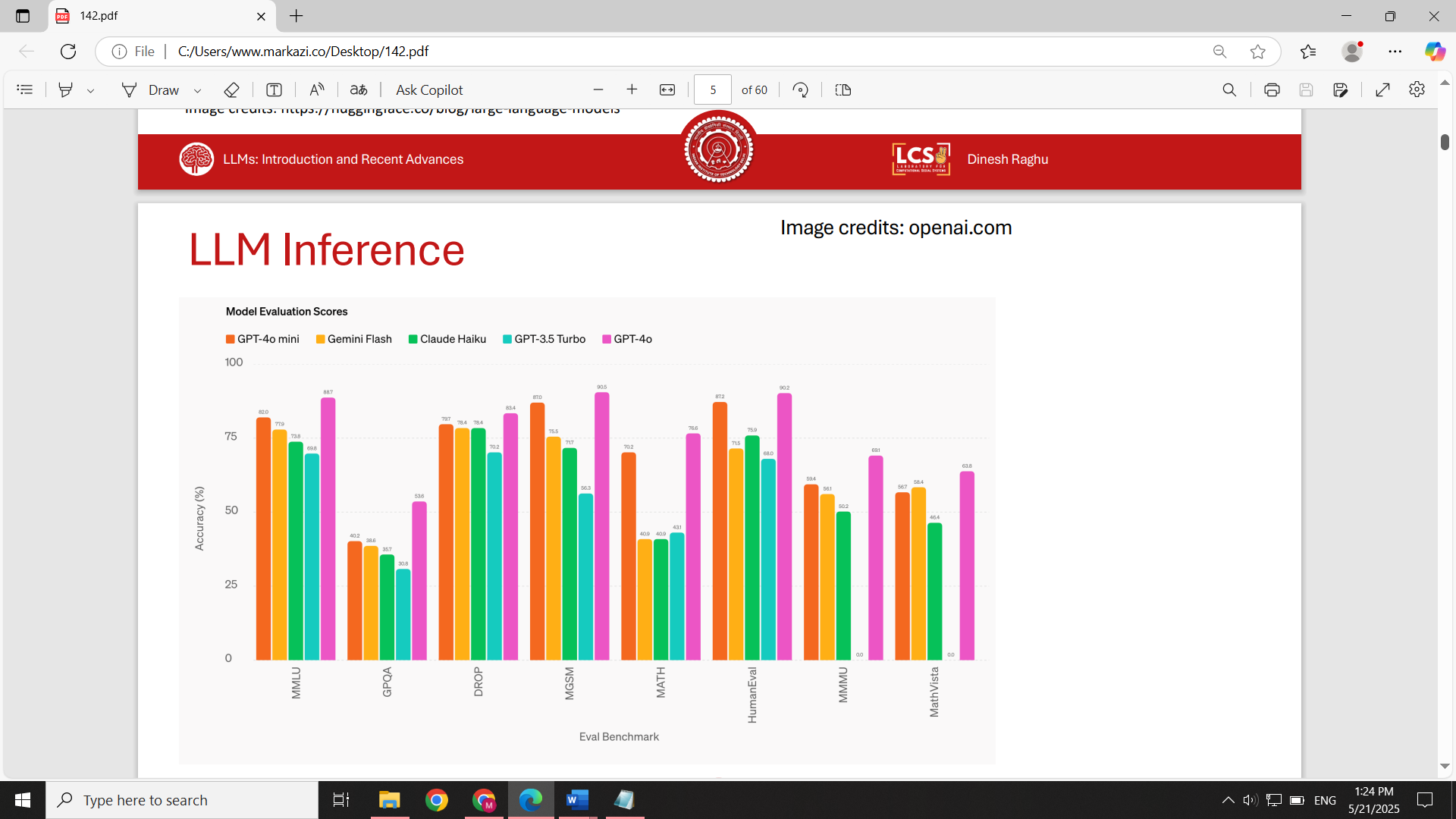
The cost of running inference on large models can be extremely high. For a product or service using LLMs, the cost of serving each user must remain below the revenue that user generates. However, as model size increases, even slight improvements in accuracy often come with disproportionately higher serving costs. This makes it difficult to maintain a profitable business if relying solely on large-scale LLMs without optimization.

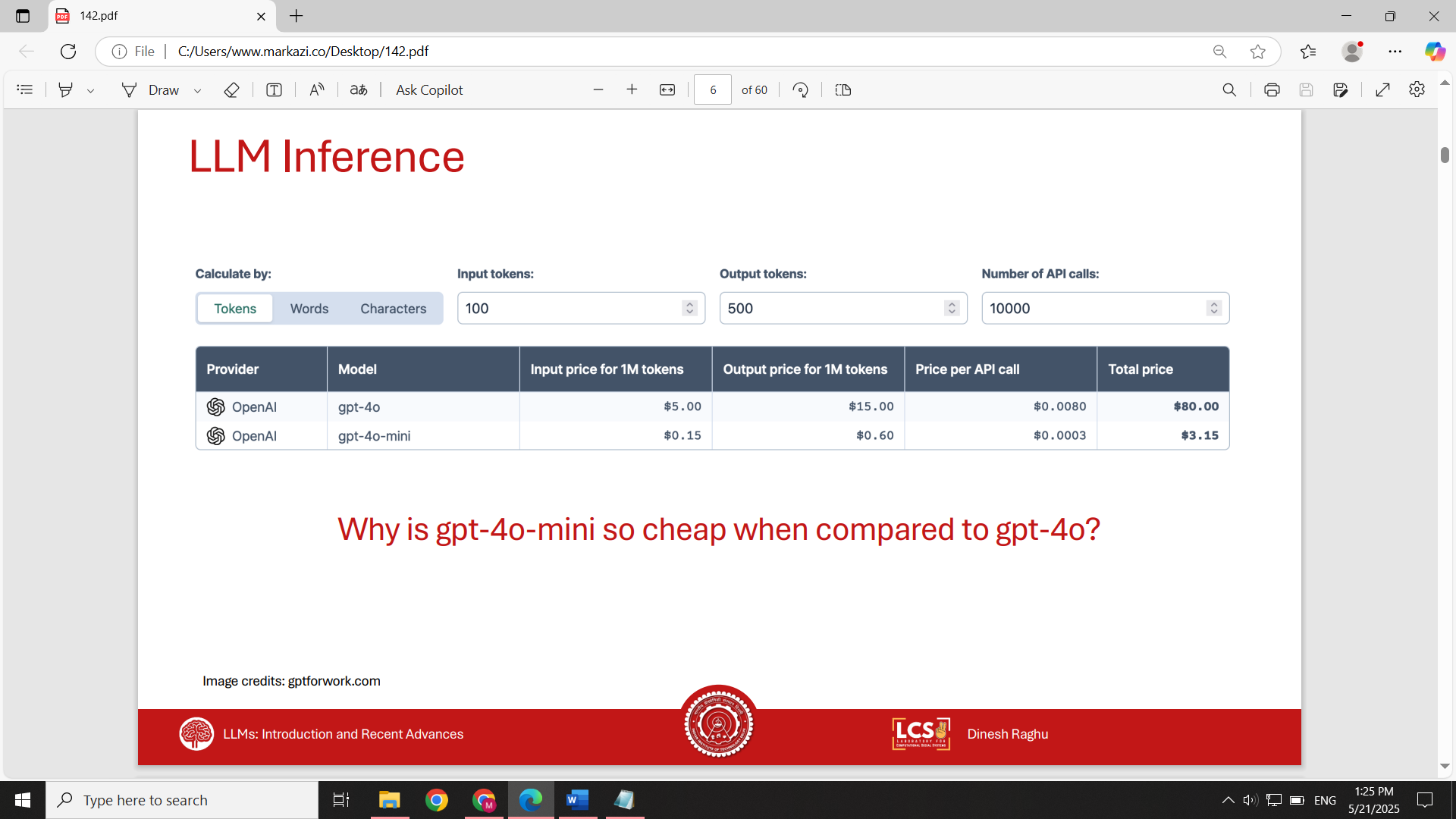
1. **Environmental and Sustainability Concerns**

Inference with large-scale LLMs also has serious environmental implications. Running these models requires vast amounts of energy and computational resources. Data centers hosting such models often consume large amounts of electricity and water for cooling, leading to high carbon emissions. Unless the energy is sourced sustainably, the environmental footprint of LLM inference becomes a growing concern, raising ethical and ecological questions about widespread AI deployment.

**A Practical Example: GPT-4 vs. GPT-4 Mini**

OpenAI recently introduced GPT-4 Mini, a smaller variant of GPT-4. While GPT-4 Mini performs slightly worse—around 6 points lower on the MMLU benchmark and 3 points lower on MGSM—it still outperforms the previous generation model, GPT-3.5. The major advantage is in cost: for the same usage scenario, GPT-4 may cost around $80, while GPT-4 Mini only costs about $3. This example demonstrates that a minor drop in performance can lead to a dramatic reduction in cost, making the smaller model a much more commercially viable option.

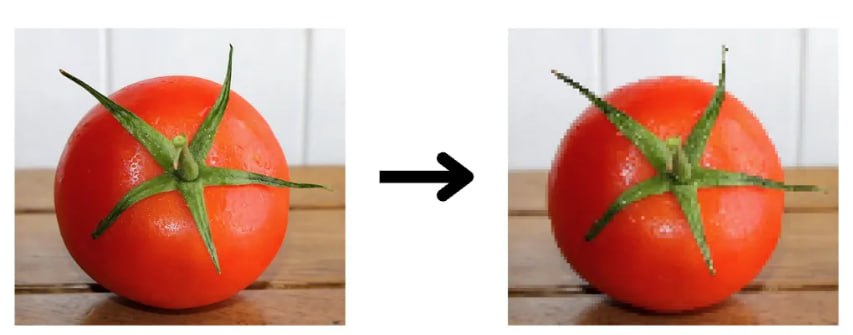




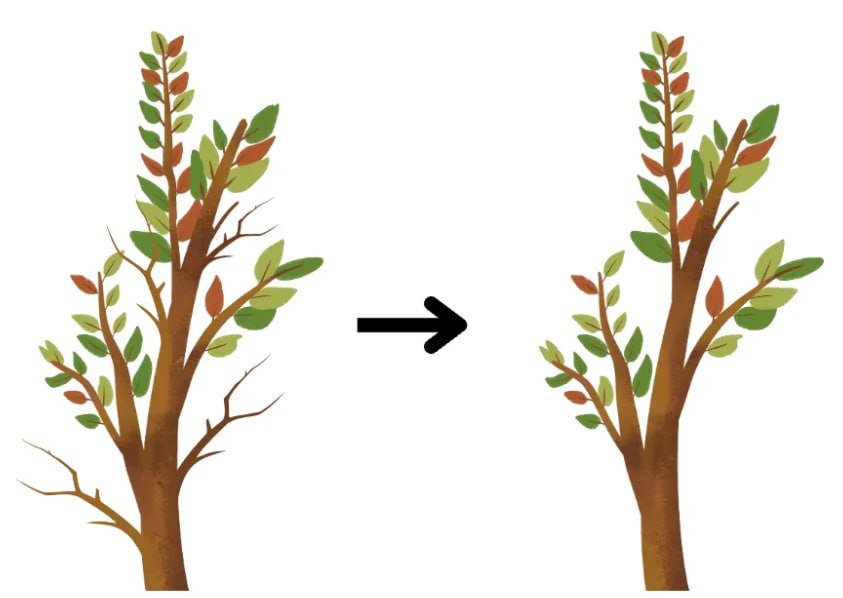
**How can we deploy LLMs in a cost-effective and efficient manner without severely compromising performance?**

This is where model optimization techniques like quantization, pruning, and distillation come into play. These methods aim to reduce model size and computational demand while retaining as much of the original model's accuracy and capabilities as possible.

**Quantization:** reduces the number of bits used to represent model parameters (weights) and activations. For example, converting a model from 32-bit floating-point (FP32) to 8-bit integers (INT8), or even 4-bit formats. This reduces memory usage and inference cost significantly. While it can be slightly lossy, the tradeoff is often negligible in terms of accuracy.



**Pruning:** eliminates unnecessary weights or entire structures (e.g., neurons or heads) from the model:

* **Unstructured pruning** removes individual weights
* **Structured pruning** removes entire layers or attention heads

The goal is to reduce model size and computation, ideally without a significant drop in performance. Pruning is inherently lossy, but often very effective for inference efficiency.

**Distillation:** involves training a smaller student model to mimic the behavior of a larger teacher model. This is the least lossy model compression technique, often maintaining close to the original model's performance. However, it is also the most expensive:

* Requires full training of a large teacher model
* Needs inference over large datasets to extract teacher predictions
* Involves re-training the student model using those predictions

Distillation offers the best balance of size reduction and performance retention but comes with higher compute and training cost. Sometimes, the teacher model may be overparameterize, simplifying the model results in better performance.

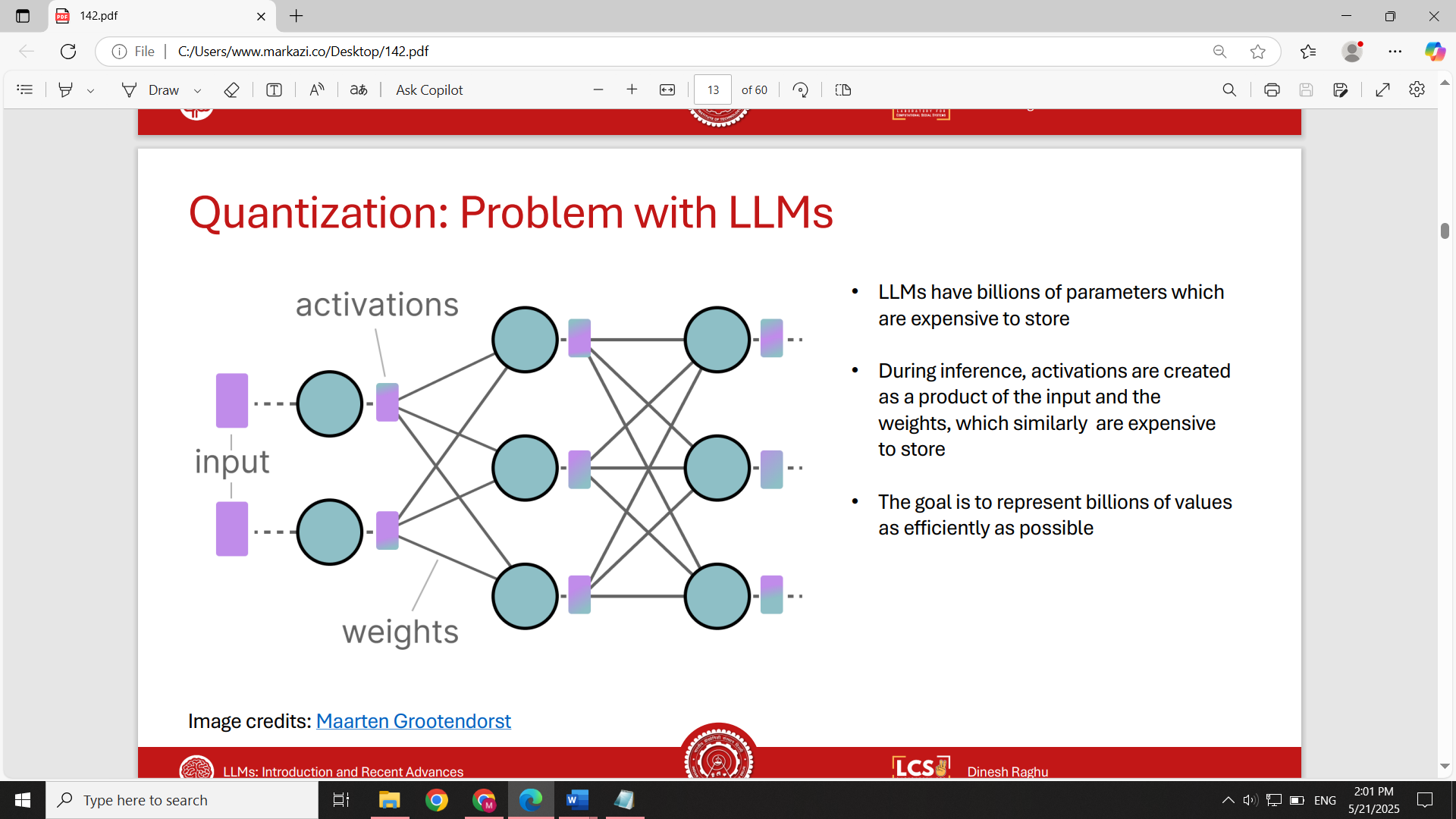
**Quantization**

Quantization is a technique used to reduce the memory footprint and inference cost of LLMs by representing values using lower-precision formats. Before diving into the quantization process, it's helpful to first understand how inference works in LLMs and how floating point numbers are represented in computing systems.

In a typical neural network, including LLMs:

* Inputs are passed through layers of weights (parameters).
* Activations are generated as the result of operations.
* During inference, both weights and activations must be stored and processed, consuming significant memory and compute resources.

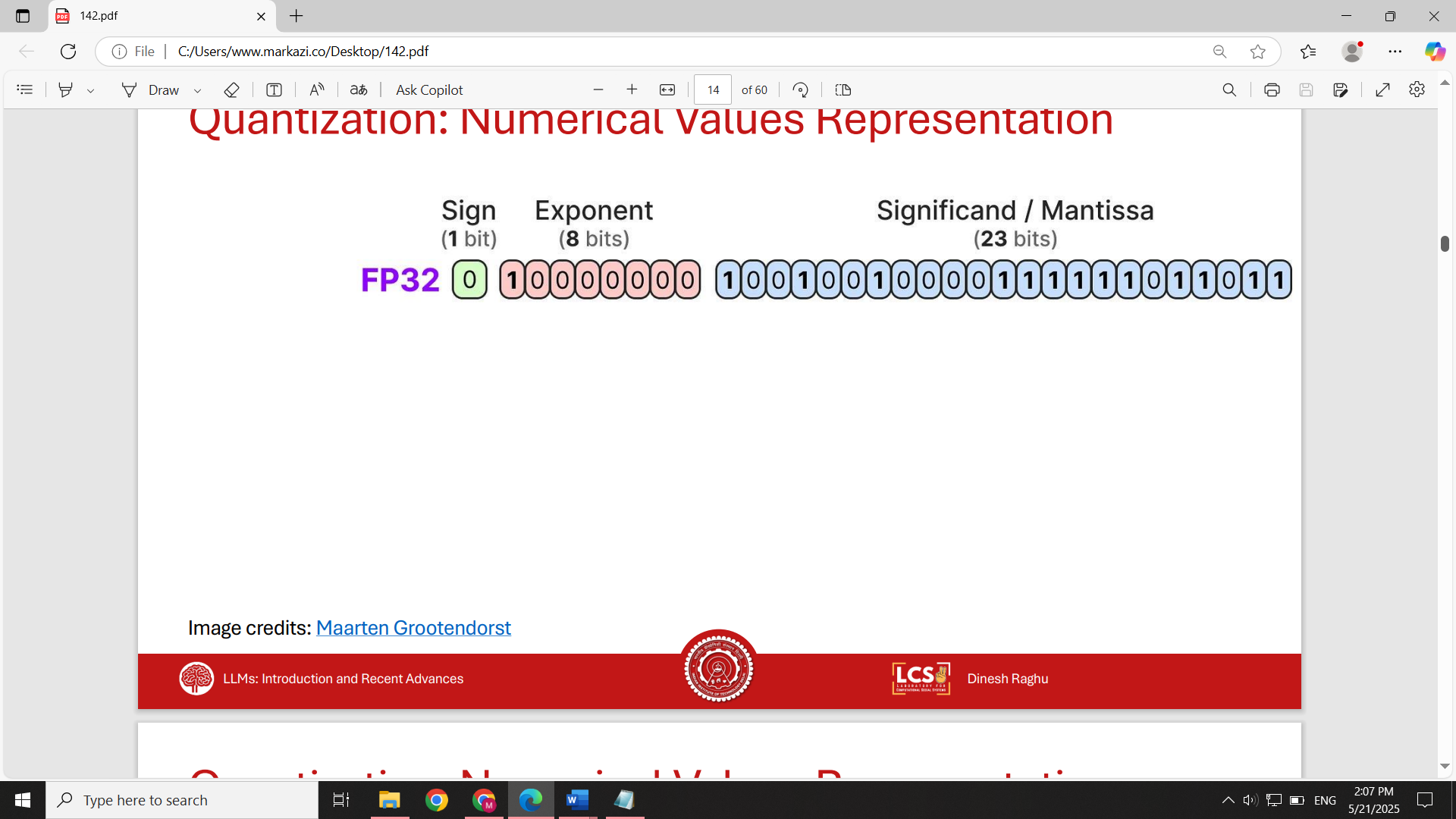
Quantization addresses this by changing how these numbers are represented and processed.



**Floating Point Representations**

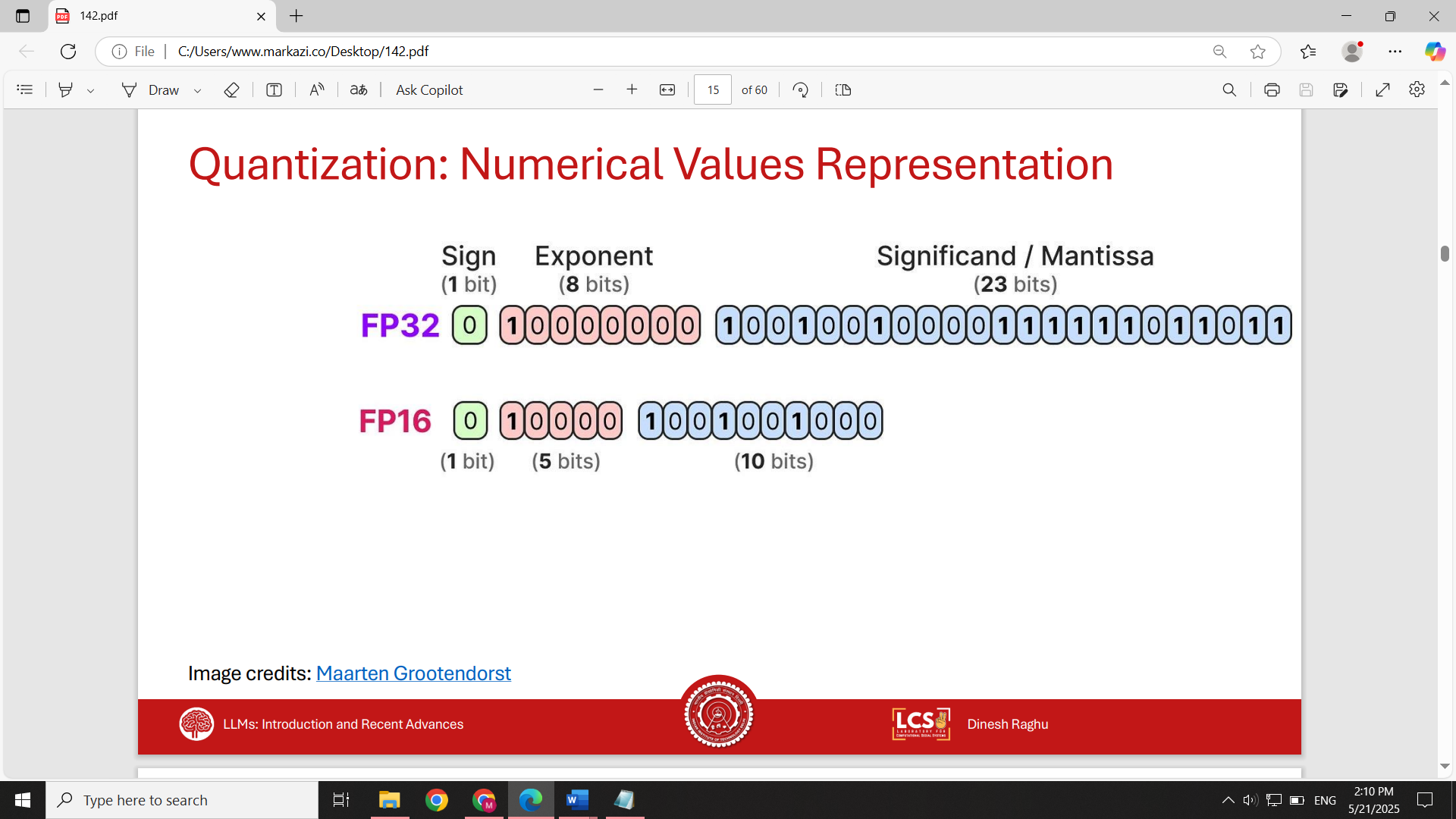
The most common format is **FP32 (32-bit floating point)**:

* 1 bit for sign
* 8 bits for exponent
* 23 bits for fraction (mantissa)
* Offers high precision and wide numeric range
* Traditionally used in scientific and machine learning applications



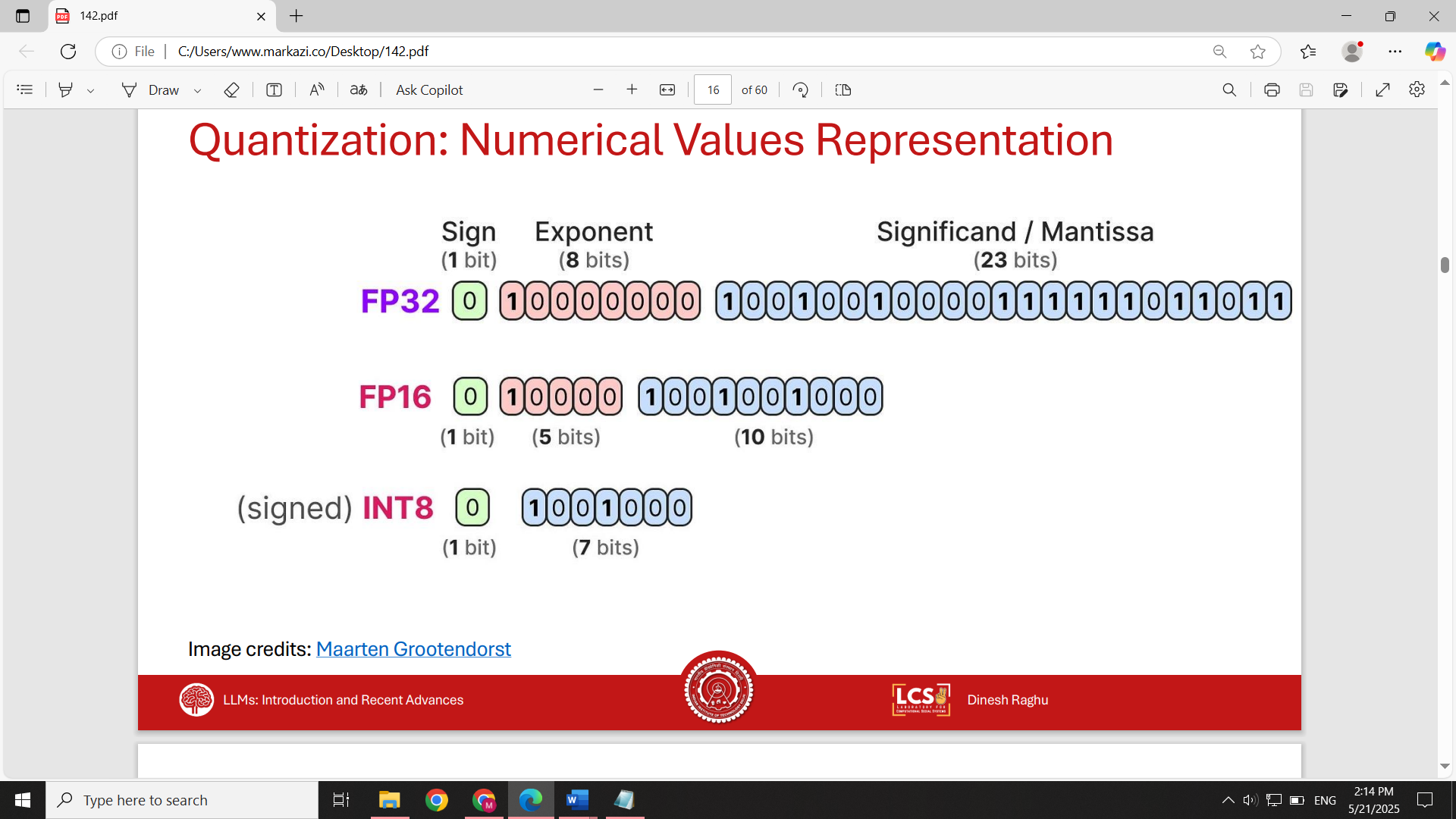
Later, **FP16 (16-bit)** became popular:

* Fewer bits for exponent and mantissa
* Lower precision and range than FP32
* Still sufficient for most deep learning tasks
* Faster computation and lower memory usage



Some applications go even further, using **INT8**:

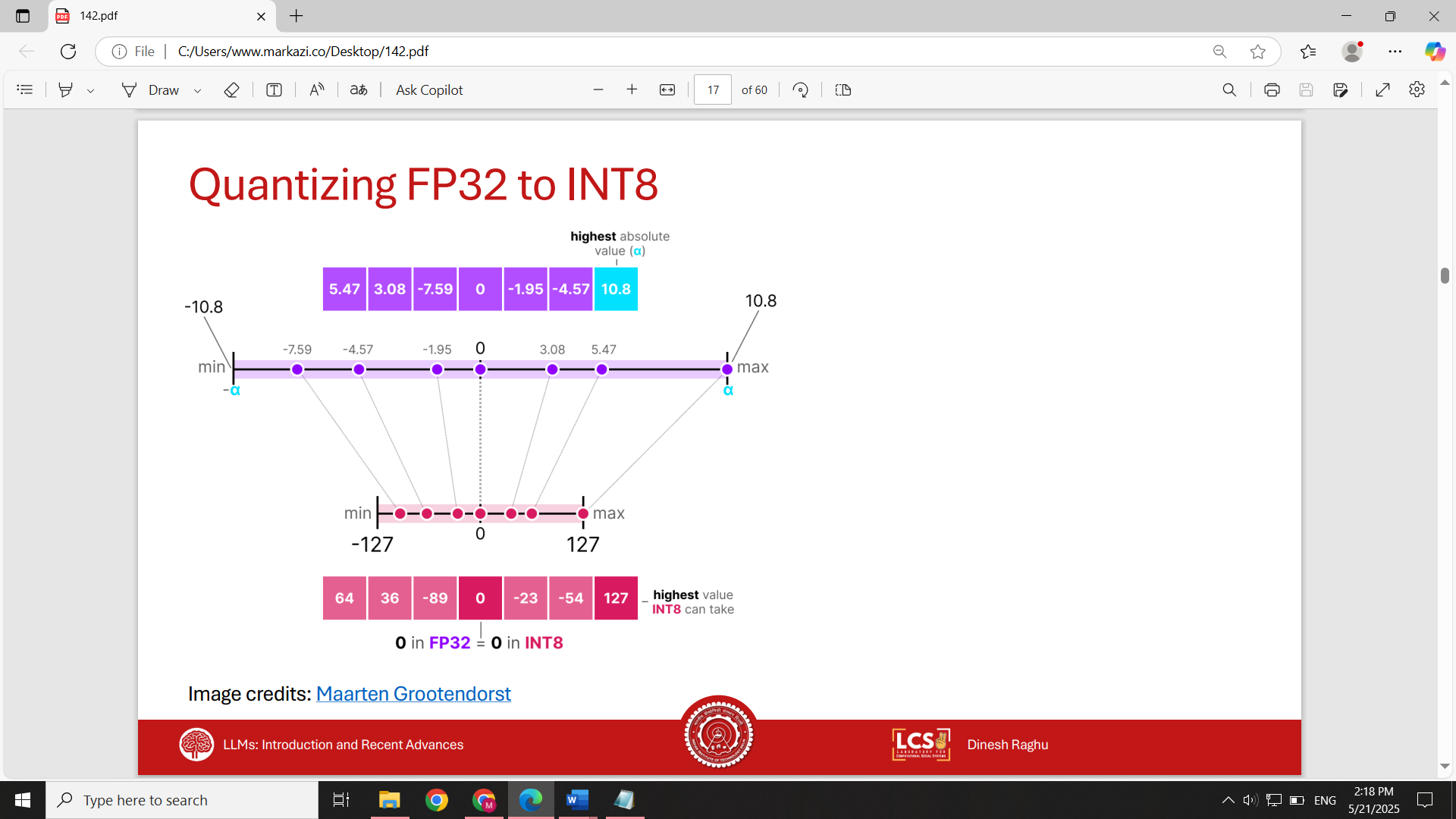
* 1 bit for sign
* 7 bits for integer value
* Much lower precision, but vastly reduced memory and compute cost



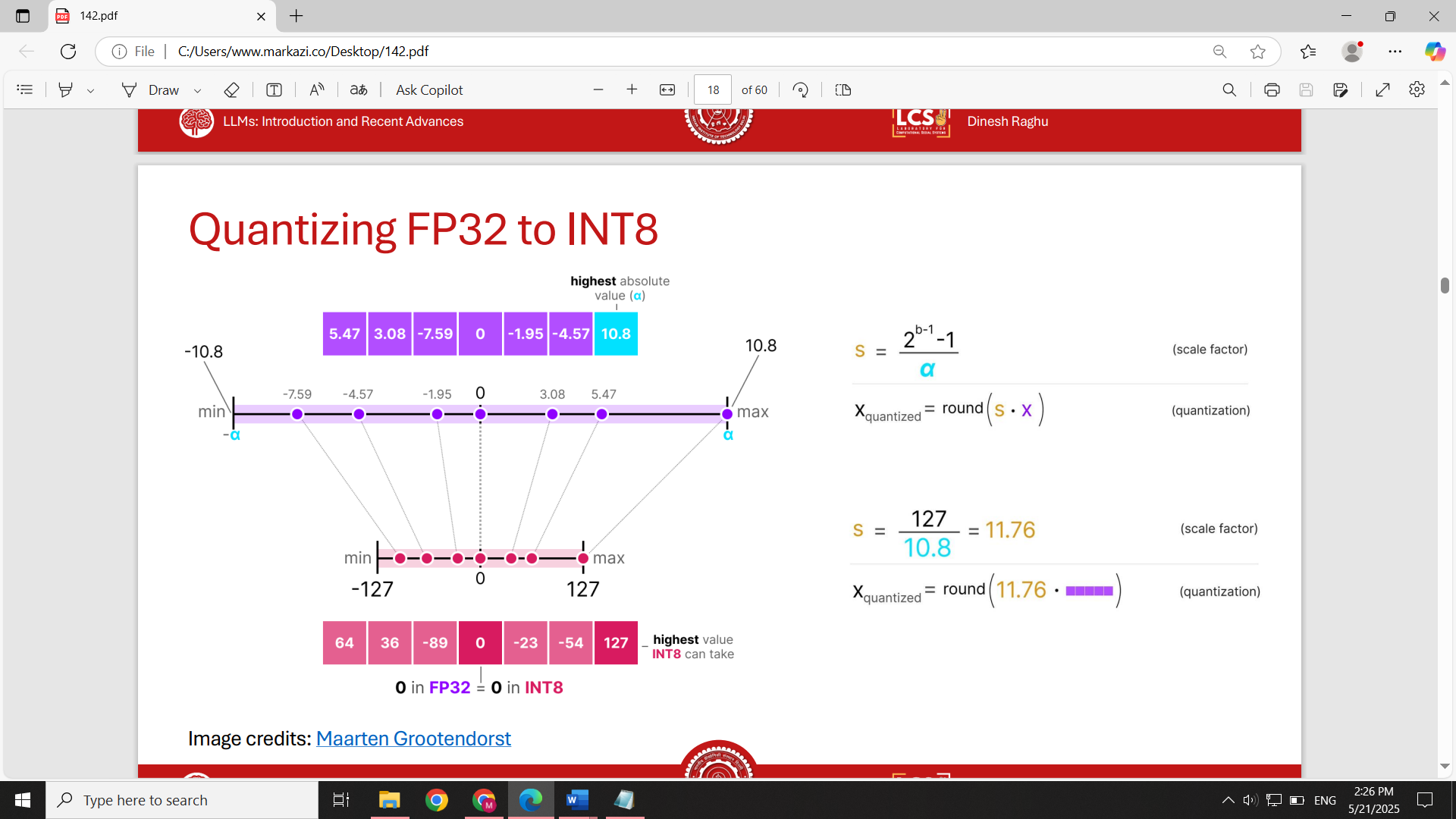
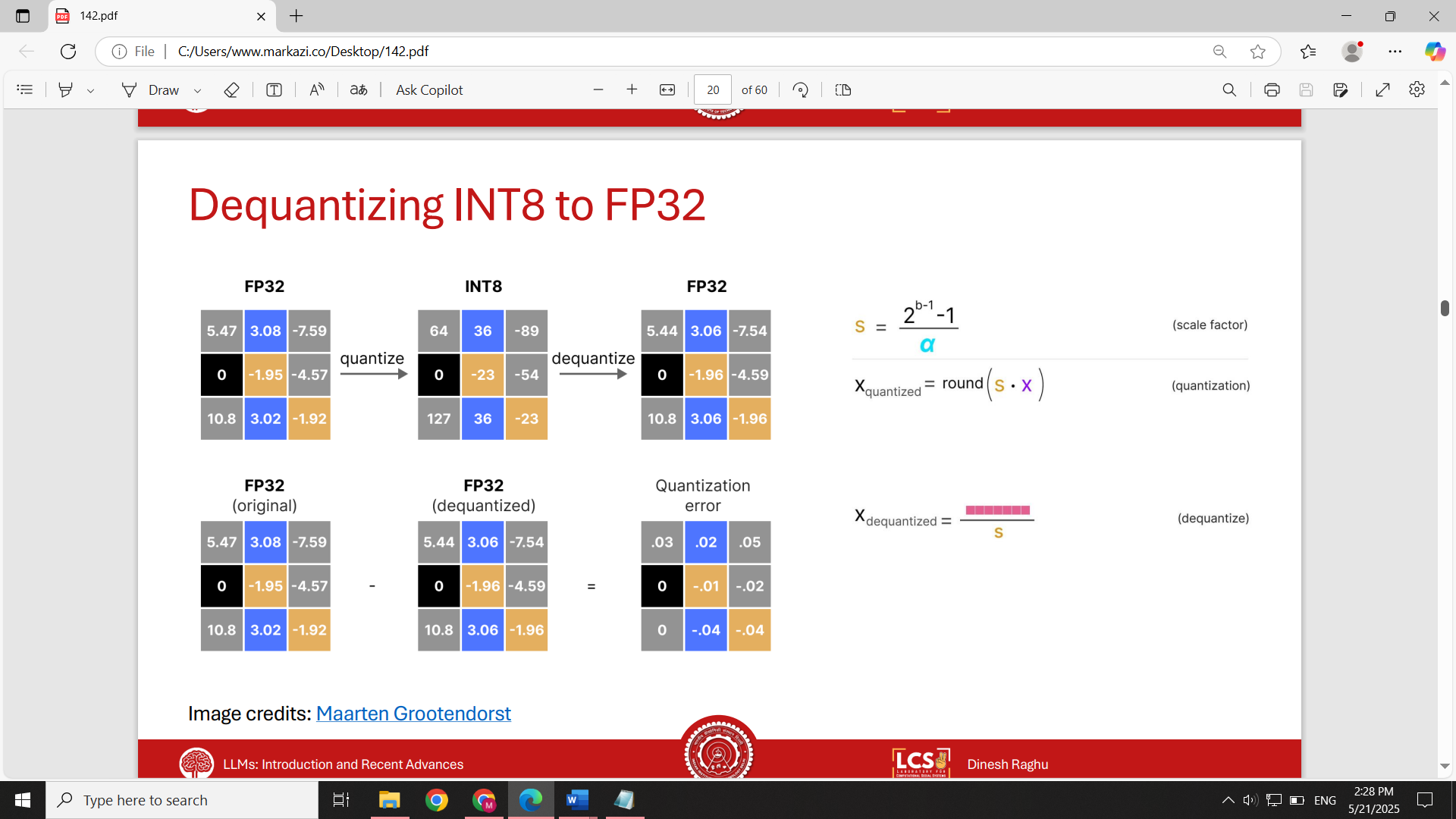
**How Quantization Works**

To convert FP32 to INT8, follow these steps:

1. **Identify the max absolute value (α)** in the vector.
2. **Symmetrically define the range** around 0 using [-α, +α].
3. **Scale the range** to [-127, +127] (for 8-bit signed integers).
4. Compute a **scale factor**:  
   scale = 127 / α
5. For each value, multiply by the scale and round:  
   quantized = round(fp32\_value × scale)



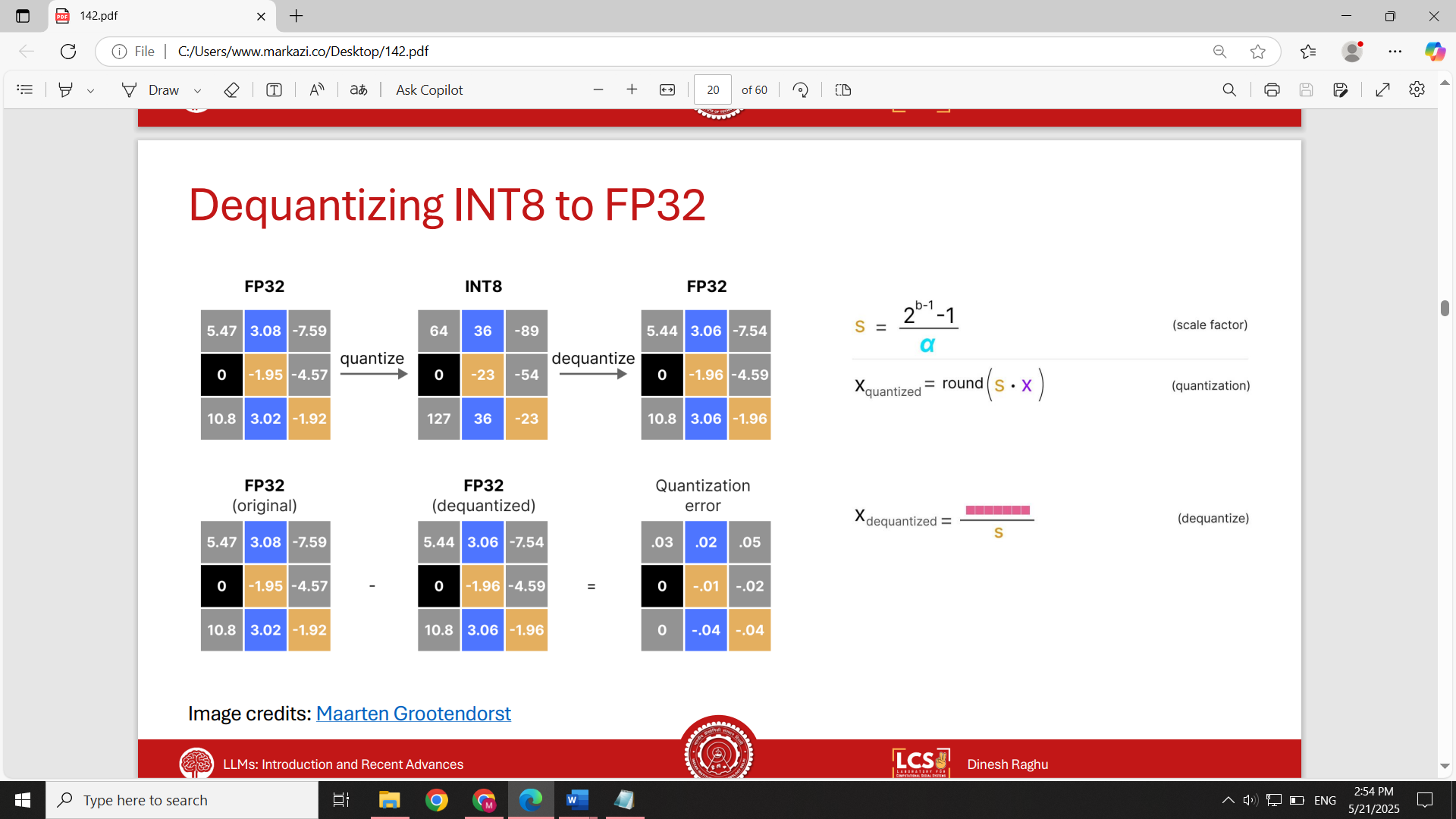
Quantization Dequantization



**Dequantization**

To recover approximate FP32 values:

* Use the quantized values and **divide by the scale factor**:  
  dequantized = quantized / scale
* This process is **lossy** due to rounding and reduced representation, introducing **quantization error**.

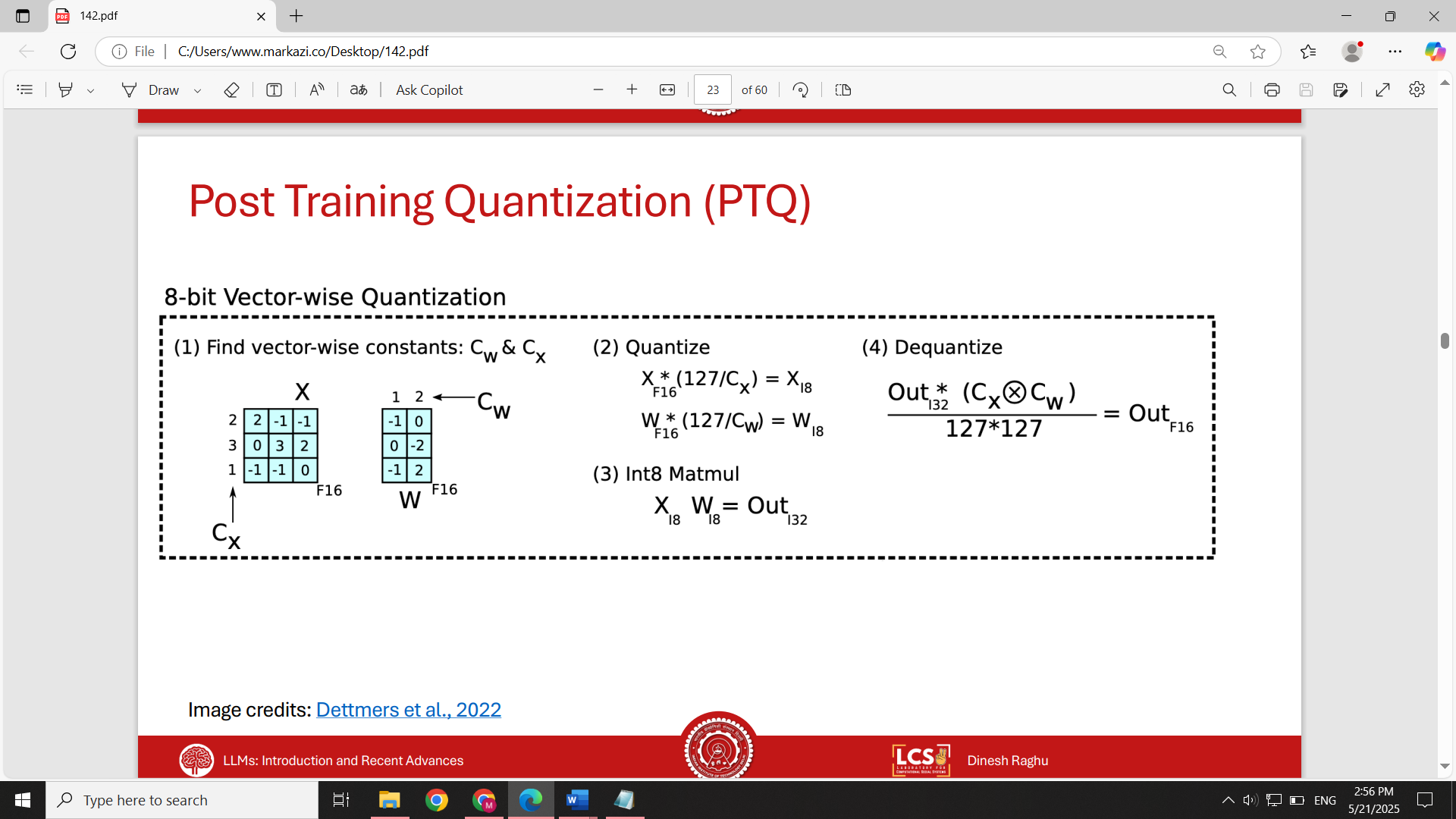


**Two Approaches to Using Quantization**

There are two ways quantization can be applied in practice:

**1. Post-Training Quantization (PTQ)**

* Train the model as usual (e.g., in FP16).
* After training, compute scale factors for weights and activations.
* Quantize the model for inference without retraining.
* This is simple and no retraining needed but may introduce more quantization error.



**Post-Training Inference Workflow**

1. Quantize inputs and weights for a given operation.
2. Perform computation in **INT8** format.
3. Dequantize the output before passing to the next layer.
4. Repeat for each operation/layer.

Despite extra quantization/dequantization steps, INT8 matrix operations are so fast that the overhead is negligible. This leads to significant speed-ups in inference.

**2. Quantization-Aware Training (QAT)**

* Simulates quantization during training to improve robustness.
* Example: **QLoRA** is a quantized fine-tuning method based on LoRA.
* Helps the model adapt to lower precision from the beginning.
* More complex but offers better accuracy after quantization.

**Some considerations about quantization**

**Symmetric vs Asymmetric Quantization**

* **Symmetric quantization**: zero in FP32 is mapped to zero in INT8. The range is centered around zero with equal positive and negative bounds.
* **Asymmetric quantization**: the zero point is not necessarily zero in the quantized range. This allows more flexibility for data with skewed distributions.
* Symmetric is simpler and used for weights (which are static after training), while **asymmetric** is often better for activations (which vary during inference).

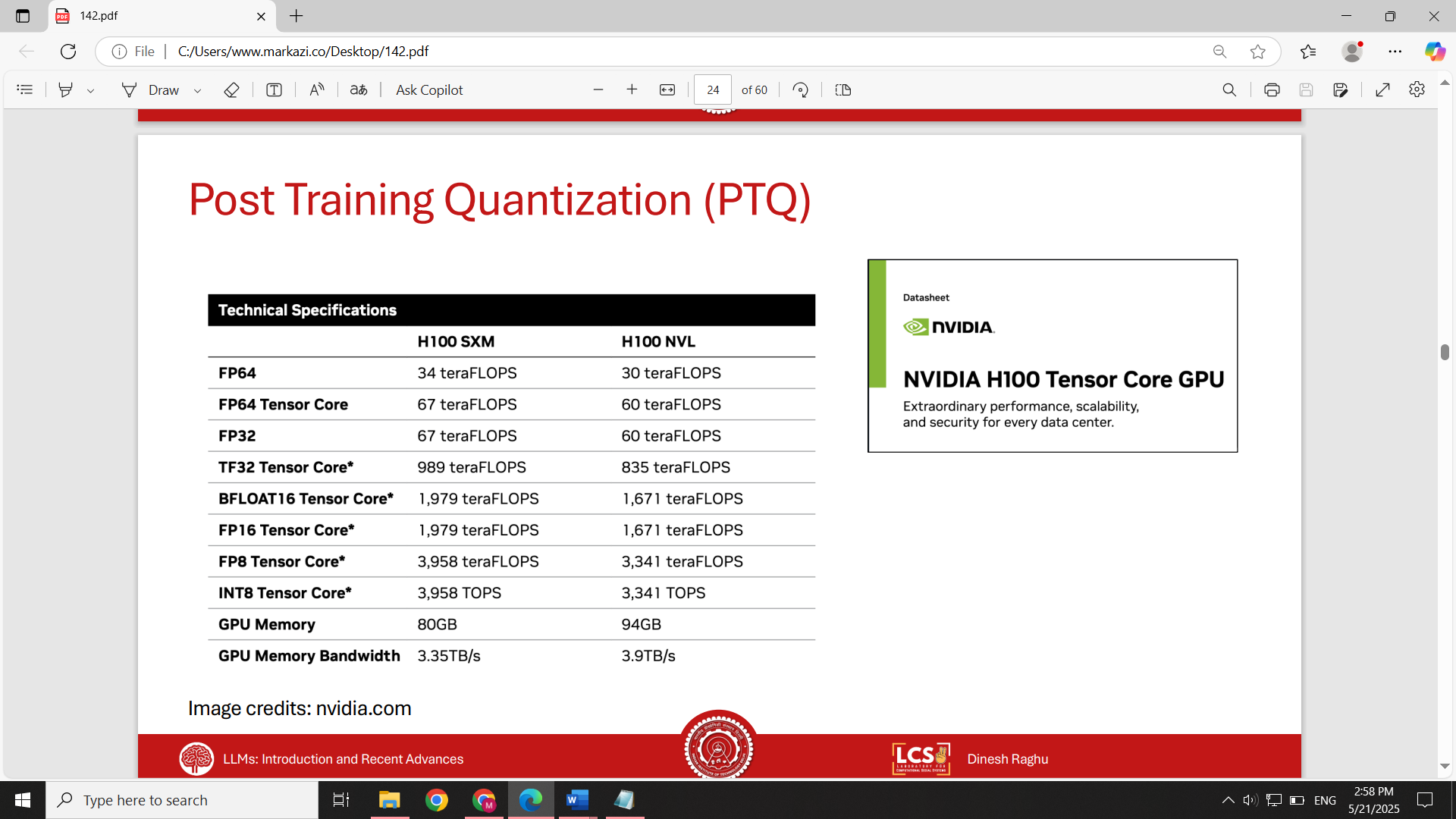
**Calibration for Activations**

* Input activations vary depending on runtime data.
* Use a small **calibration dataset** to estimate typical ranges.
* Pass data through the model to observe **activation patterns**.
* Compute scale factors based on observed activation ranges.

**Real-World Impact**

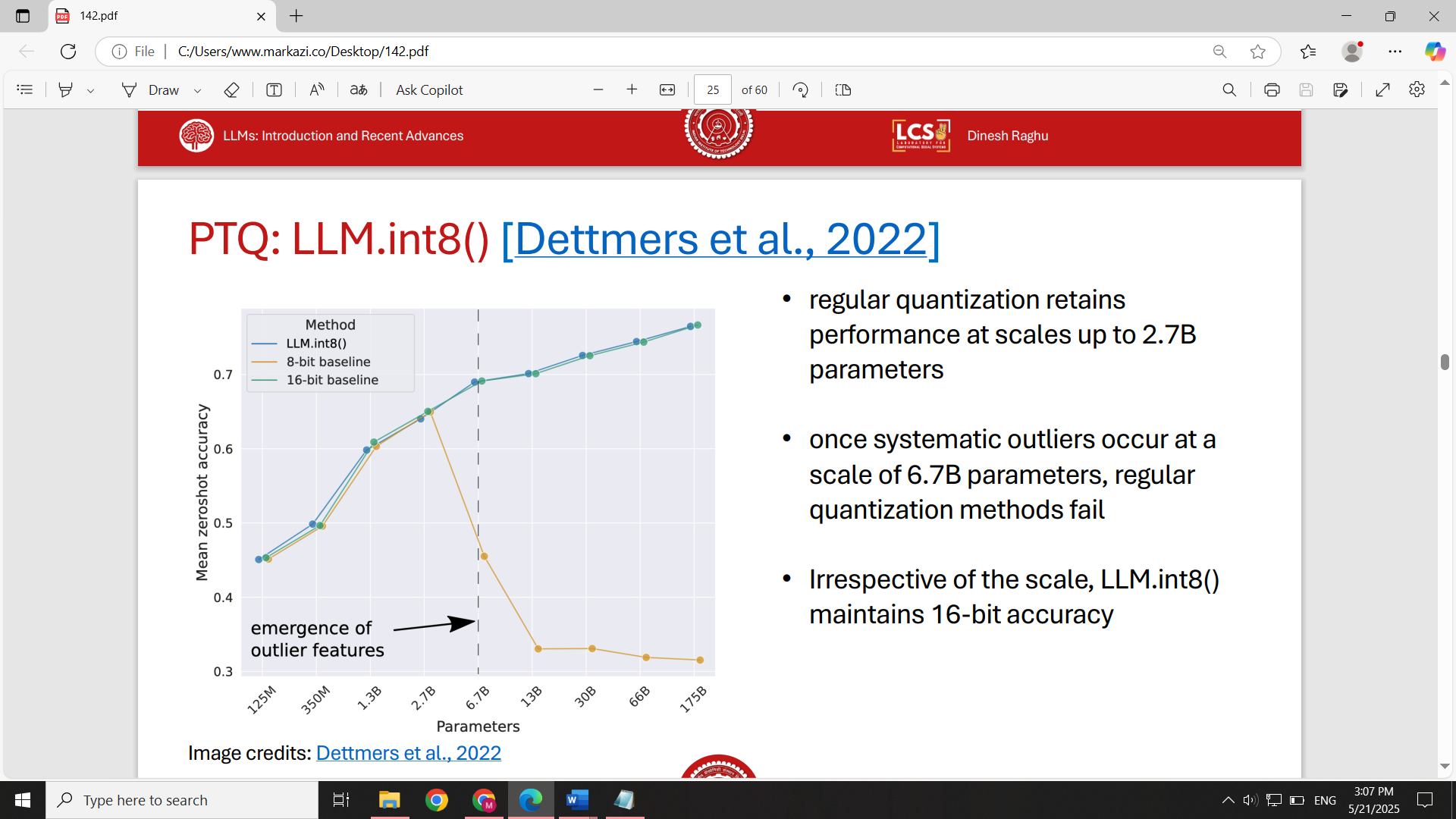
For example, converting operations from **FP16 to INT8** on GPUs like **NVIDIA H100**:

* Allows significantly **more operations per second**
* Provides better **throughput and lower latency**
* Reduces **energy consumption and cost per inference**
* Example: On **NVIDIA H100 GPUs**, FP16 gives ~1979 TFLOPs (SXM variant), but INT8 allows **more operations per watt**, leading to significantly **higher throughput** for the same energy budget.



**INT8 Limitation Beyond 2.7B Parameters**

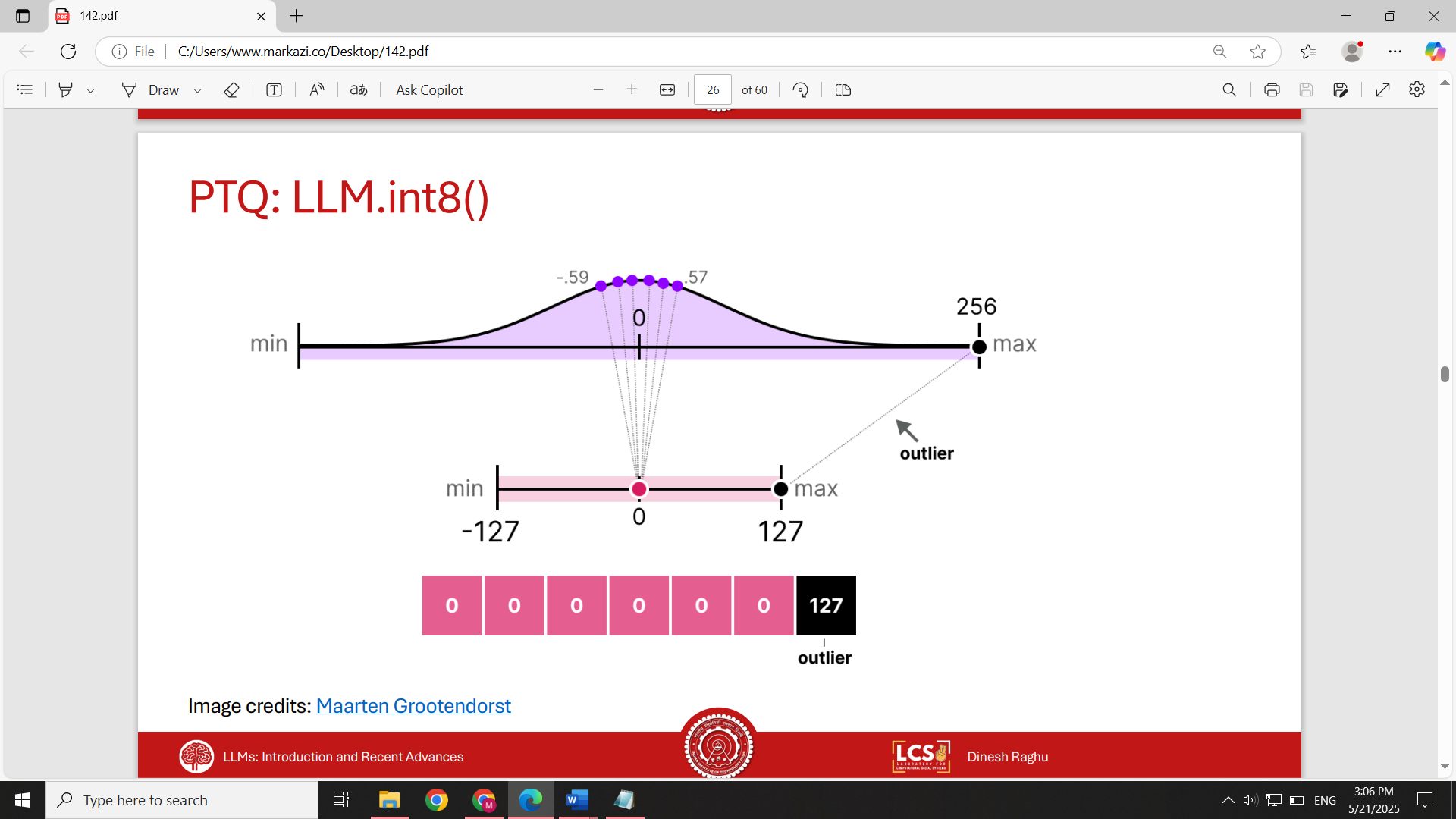
* Around **2.7B parameters**, INT8 starts **losing accuracy**.
* Larger models (e.g., 6B+) showed **a significant drop** in performance compared to FP16 (green line in plot).



Accuracy

**Why the Drop? Outliers in Activation Vectors**

* Smaller models: vectors are mostly **centered and bounded**.
* Larger models: **outliers** (very large values) become more common.
* INT8 quantization must account for outliers when computing scale, leading to significant loss of information.

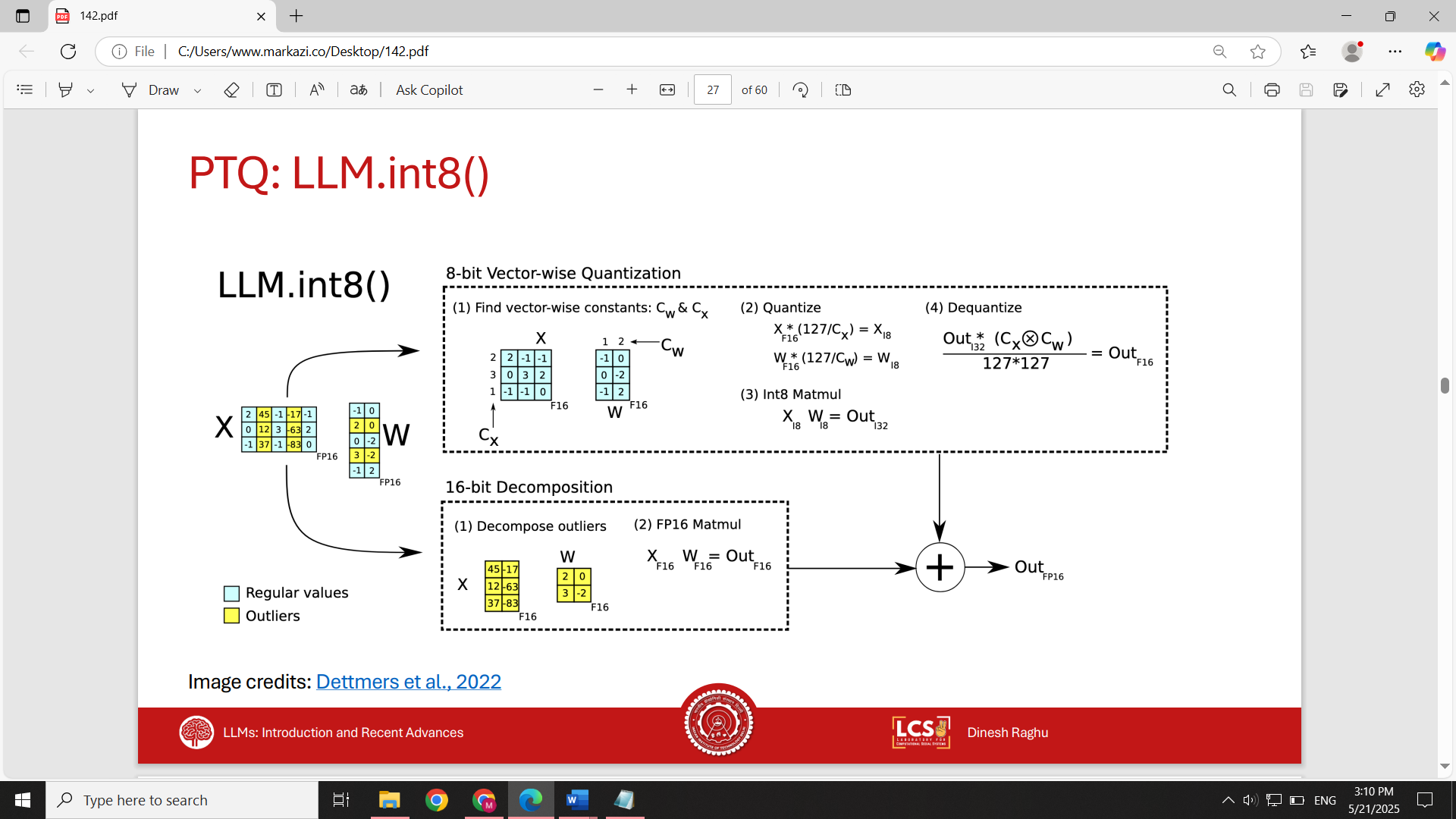


**LLM.int8() Solution: Outlier-Aware Quantization**

**separate outliers** before quantization. Method:

* Identify outlier elements (above a threshold).
* Exclude them from computing scale factors.
* Quantize remaining values as usual.
* Handle outliers separately using original precision.

this fixes the performance dip while retaining INT8 benefits.



**QLoRA: Memory-Efficient Quantization-Aware Fine-Tuning**

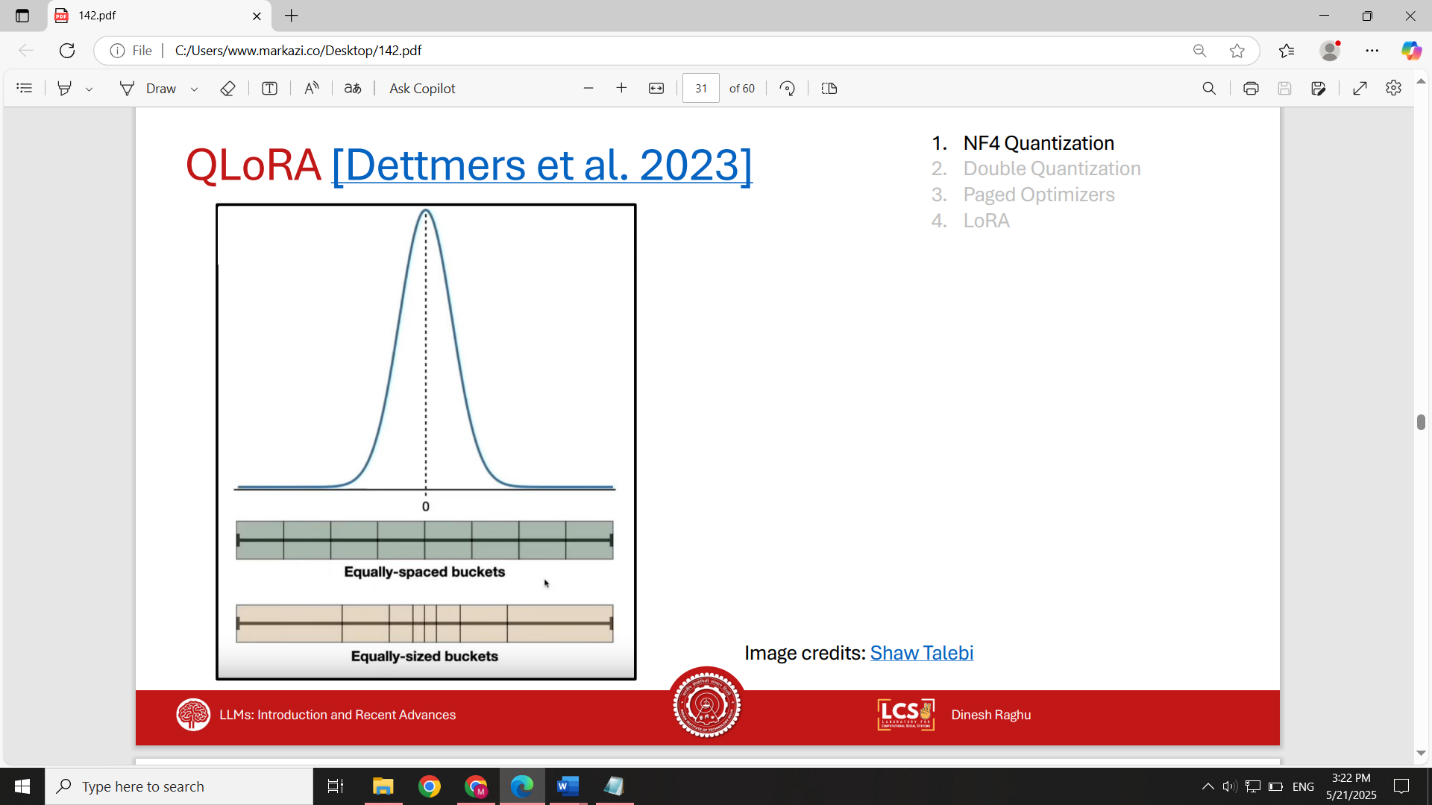
QLoRA = LoRA + advanced quantization strategies.

Reduces memory footprint dramatically, enabling fine-tuning a 65B model in ~48GB GPU memory (vs. 780GB).

Maintains FP16-level accuracy for most tasks.

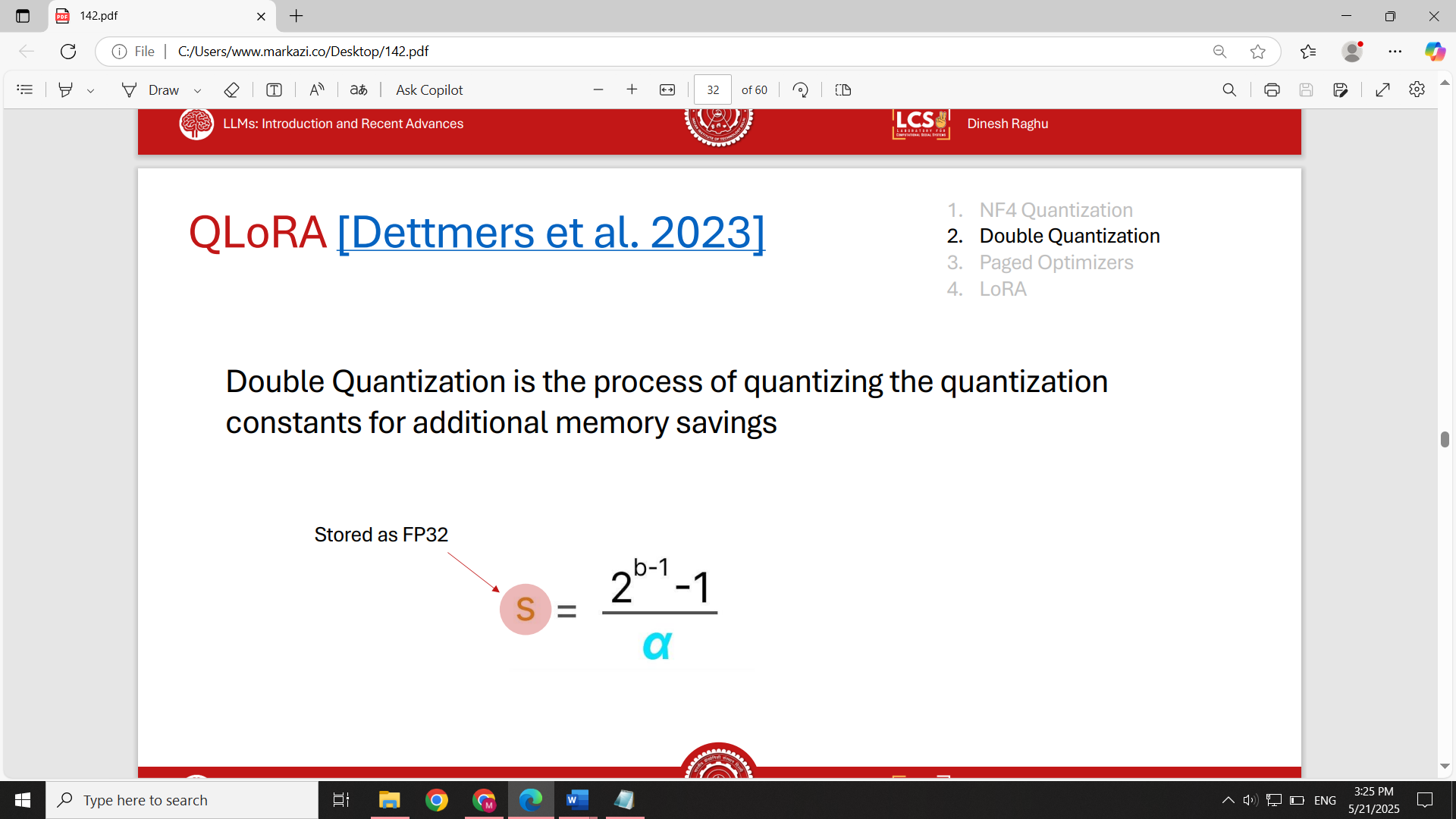
**How Do We Use Quantization in QLoRA**

**NF4 (Normal Float 4-bit Quantization)**

* Assumes **normal distribution** of weights.
* Creates **non-uniform** buckets tailored to this distribution.
* More efficient than uniform FP4.

**Double Quantization (DQ)**

* Quantizes not only weights, but also **quantization constants**.
* Helps save memory with **minimal accuracy loss**.

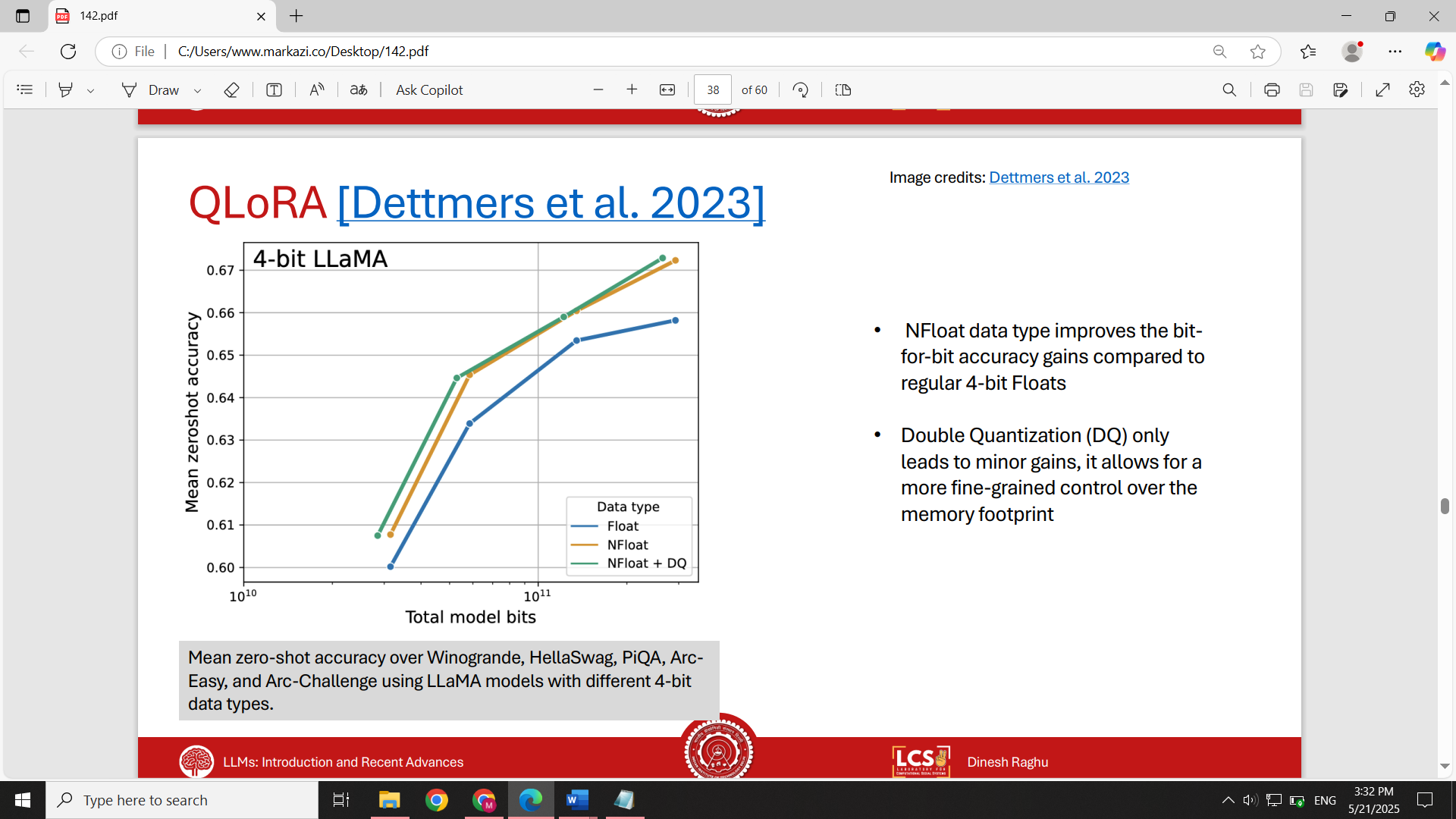


**Mixed Precision Training**

Combines multiple precisions (INT8, NF4, FP32) for optimal compute performance, model accuracy and memory efficiency.

**Performance Insights**

* **FP4 vs. NF4 vs. NF4 + DQ:**
  + FP4 shows a **clear drop** in accuracy.
  + NF4 performs **significantly better**.
  + NF4 + DQ matches or slightly outperforms **BF16** — while saving memory.



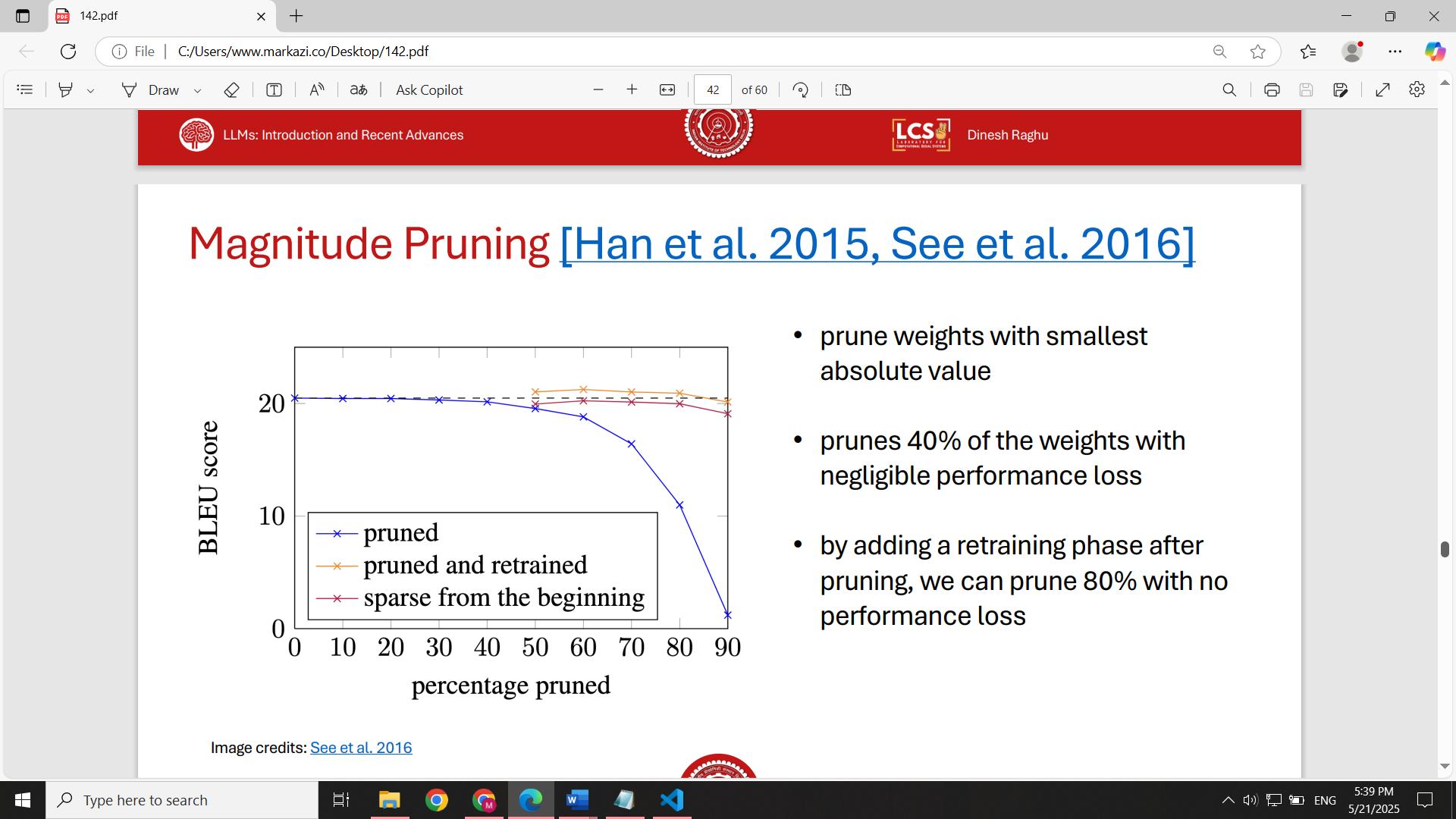
Accuracy

**What is Pruning?**  
Pruning refers to the process of removing parts of a neural network (typically weights or components) to reduce its size and computational demands—ideally without significantly affecting performance.

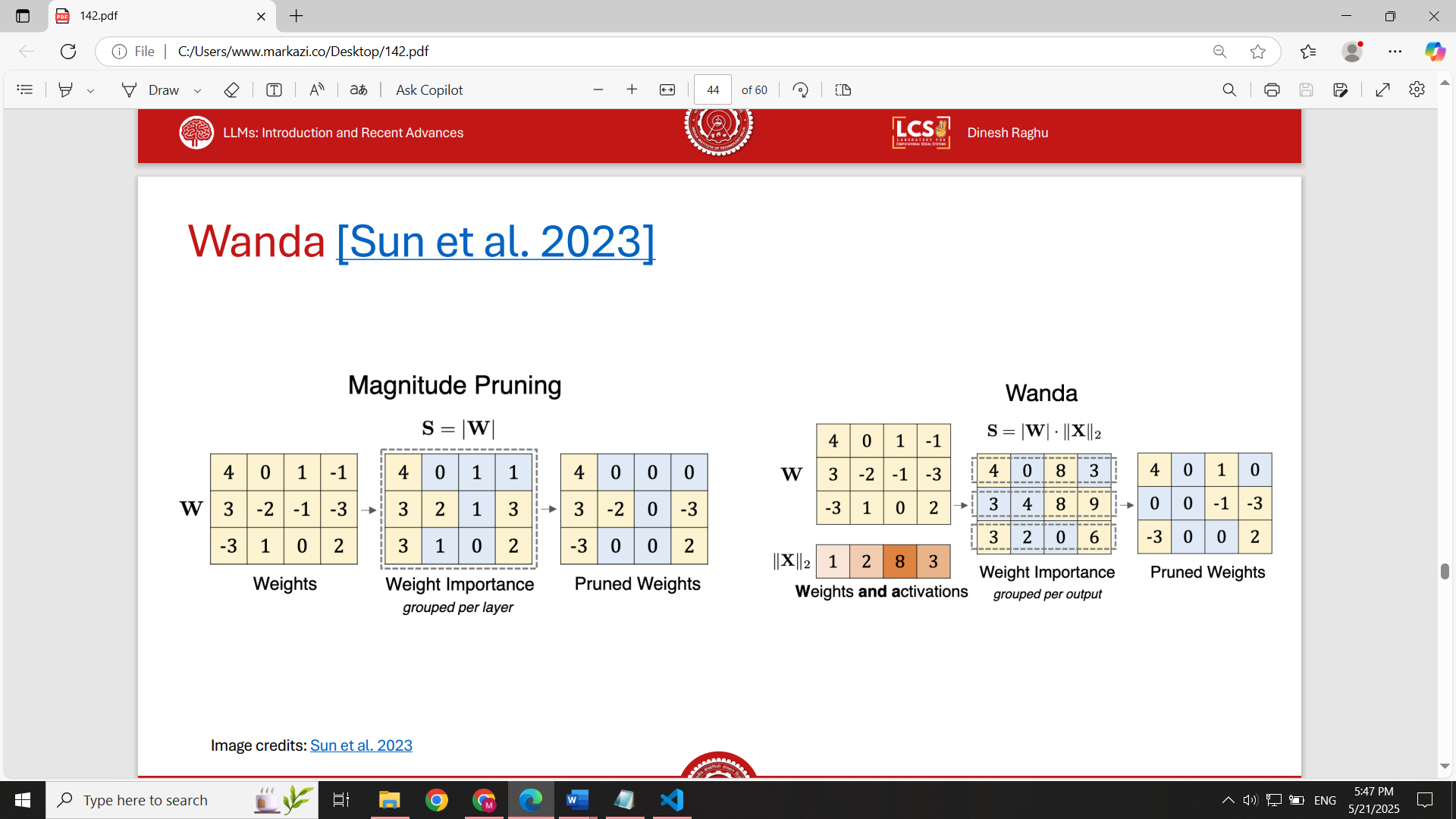
**Types of Pruning**

**1. Unstructured Pruning**

* **Definition**: removes individual weights from the network, without considering the network’s architecture.
* **Example**: *Magnitude Pruning* – simply sort weights by absolute value and remove the smallest ones.
* **Key Findings**:
  + Removing up to **40% of weights** often does **not degrade performance significantly**.
  + Heavier pruning can degrade accuracy, but **fine-tuning** can recover much of the loss—**up to 80% effectiveness**.

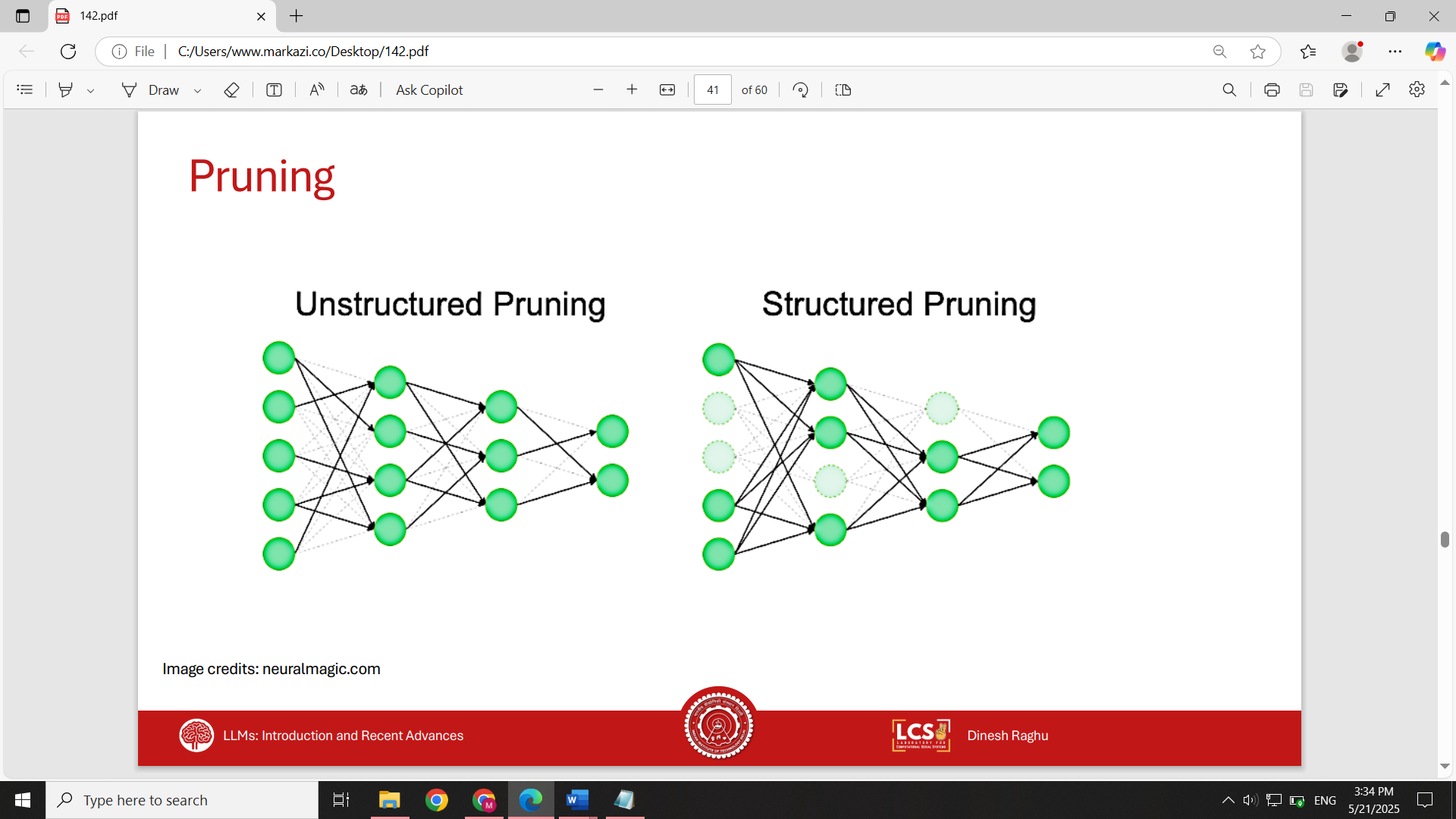


Score

* **Beyond Weights**:
  + Traditional pruning considers only **weight magnitudes**.
  + Newer approaches consider **activations** too—by pruning based on the product of **weights × activations**, more informed pruning decisions are made.
* **Limitation**:
  + **No hardware benefits** if the pruned weights are still stored and processed (e.g., storing zeros in FP16 still consumes space and compute).
  + Result: **No speedup, energy savings, or memory benefits** unless the hardware supports sparse computation.

**2. Structured Pruning**

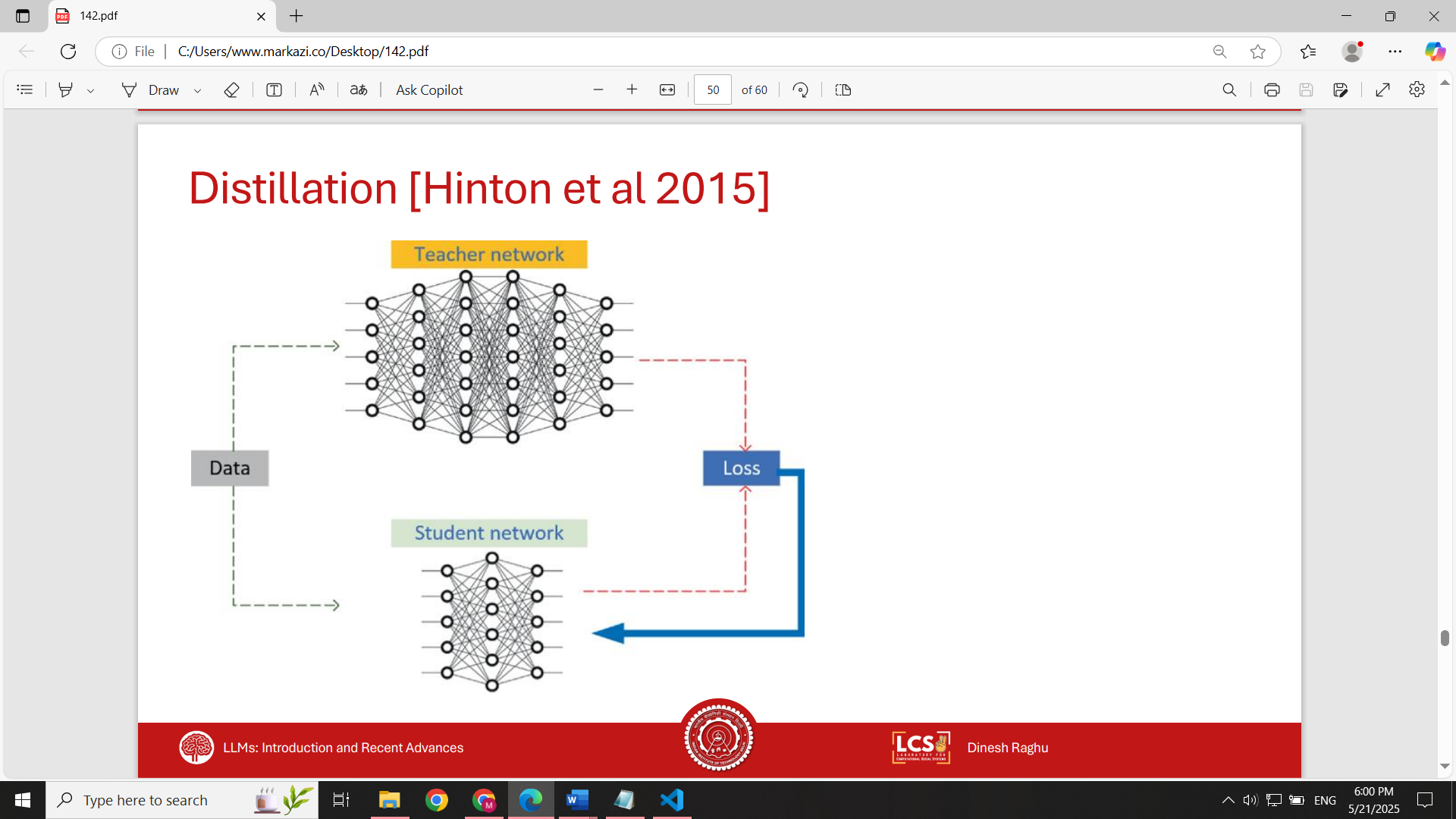
* **Definition**: Removes entire structures or blocks (e.g., neurons, attention heads, layers), aligned with hardware or model architecture.
* Works by removing structural components from transformer models:
  + Entire feedforward layers.
  + Multi-head attention blocks or individual attention heads.
* This kind of pruning leads to computational efficiency with minimal performance loss when done carefully.



**Model Compression: Distillation**

**What is Distillation?**

**Distillation** is the process of training a smaller **student** model to imitate a larger **teacher** model. The goal is to make the student **mimic** the output (and possibly internal behavior) of the teacher. This is useful for reducing model size, latency, and inference cost, while retaining performance.



**How It Works**

1. **Input** is passed through the **teacher model**, producing outputs ŷ.
2. The **same input** is passed through the **student model**, producing ŷ'.
3. The **training objective** is to minimize the difference between ŷ and ŷ'.
4. The teacher is **frozen** (not updated during training); only the student is optimized.

**Loss functions**:

* **Soft targets**: Use full probability distributions from the teacher (better performance).
* **Hard targets**: Use argmax of teacher output as labels (less effective, but used when soft labels are unavailable).

**Challenges**

* **Requires data access**: Unlike quantization, distillation needs representative or task-specific training data.
* **Data availability is limited**: Models are often open-sourced, but training data is not.
* **Compute cost**:
  + Inference on large models is **slow and expensive**.
  + Training the student requires full training loops after generating teacher outputs.

**Advantages**

* **Significant size & latency reduction**.
* Can compress extremely large models into smaller, usable ones.
* Student architecture can be **customized to meet latency constraints**.
* Often results in better performance than training the student from scratch.

**Real-World Use Case: Self-Instruct**

Giant models like **LLaMA 405B** have excellent quality, but impractical latency. We can use the large model to **generate synthetic training data**.

* Start with ~175 handcrafted instruction-task pairs.
* Use the LLM to **generate new tasks + inputs + outputs**.
* Result: A large **instruction-finetuning dataset**.

**Train a smaller model** on this dataset → faster, cheaper, still accurate.