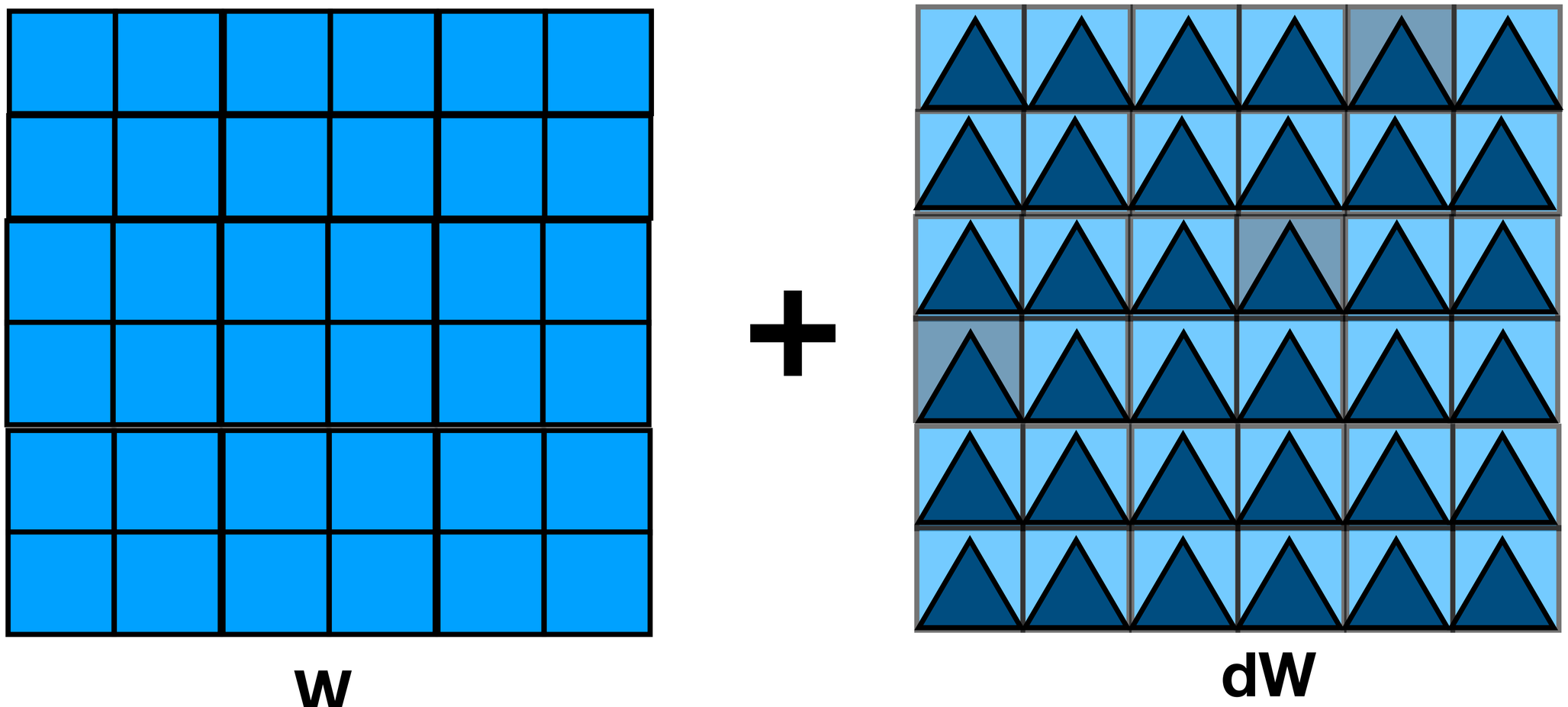
# LoRA: Low-Rank Adaptation of Large Language Models

## Efficient Fine-tuning of LLMs

**Sarath Chandra** **18th Jan 2024**

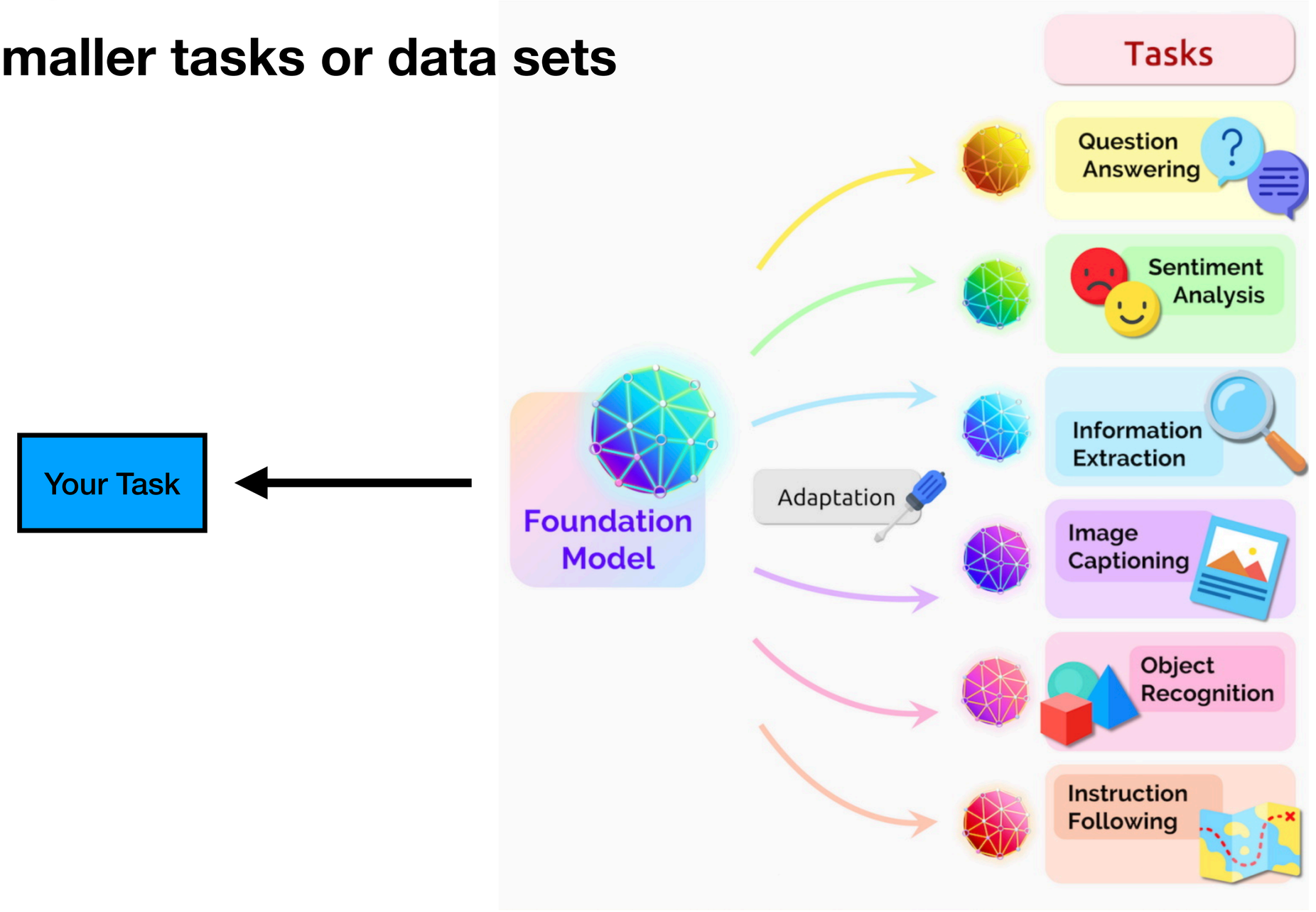
# What happens during Back-propagation?

## GPT 3 has 175B parameters ~ 700GB



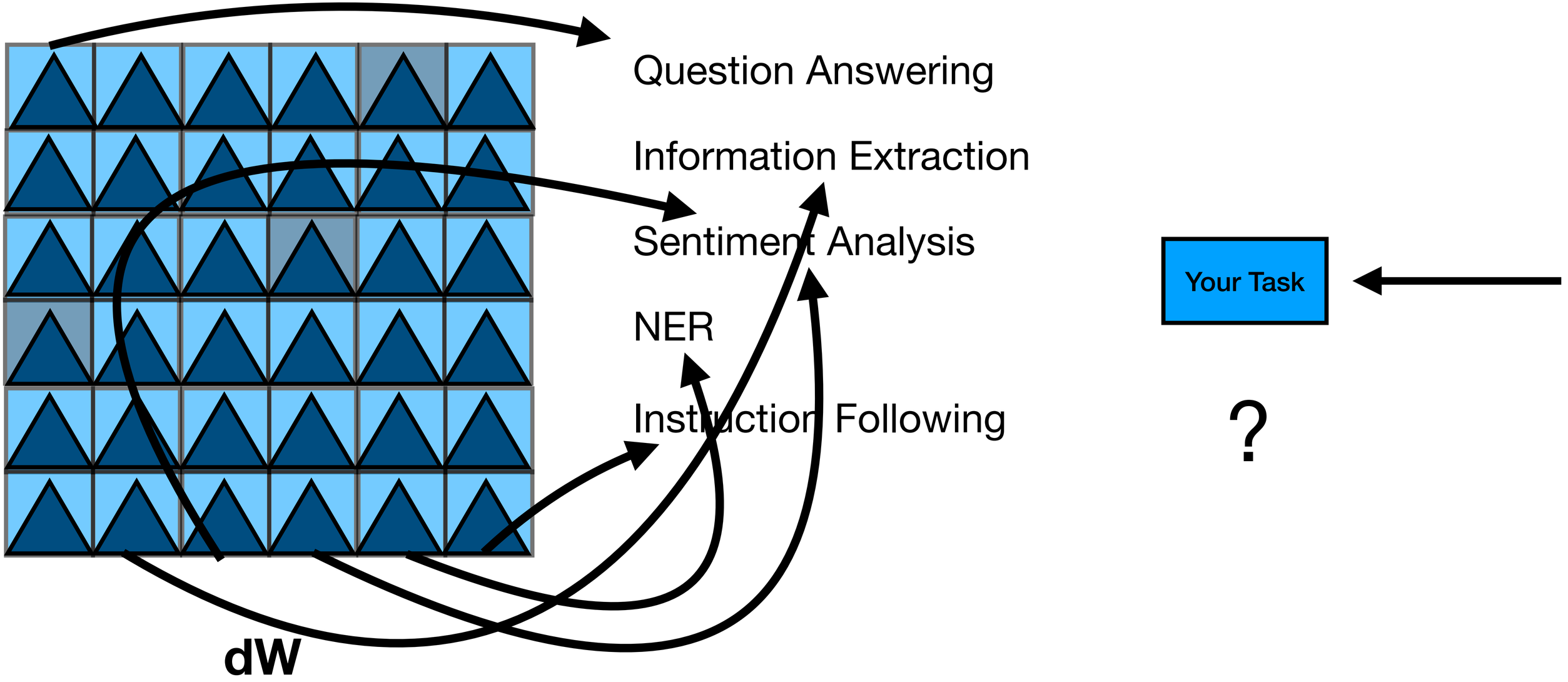
# Large Language Models may be an overkill!

## When adapted to smaller tasks or data sets

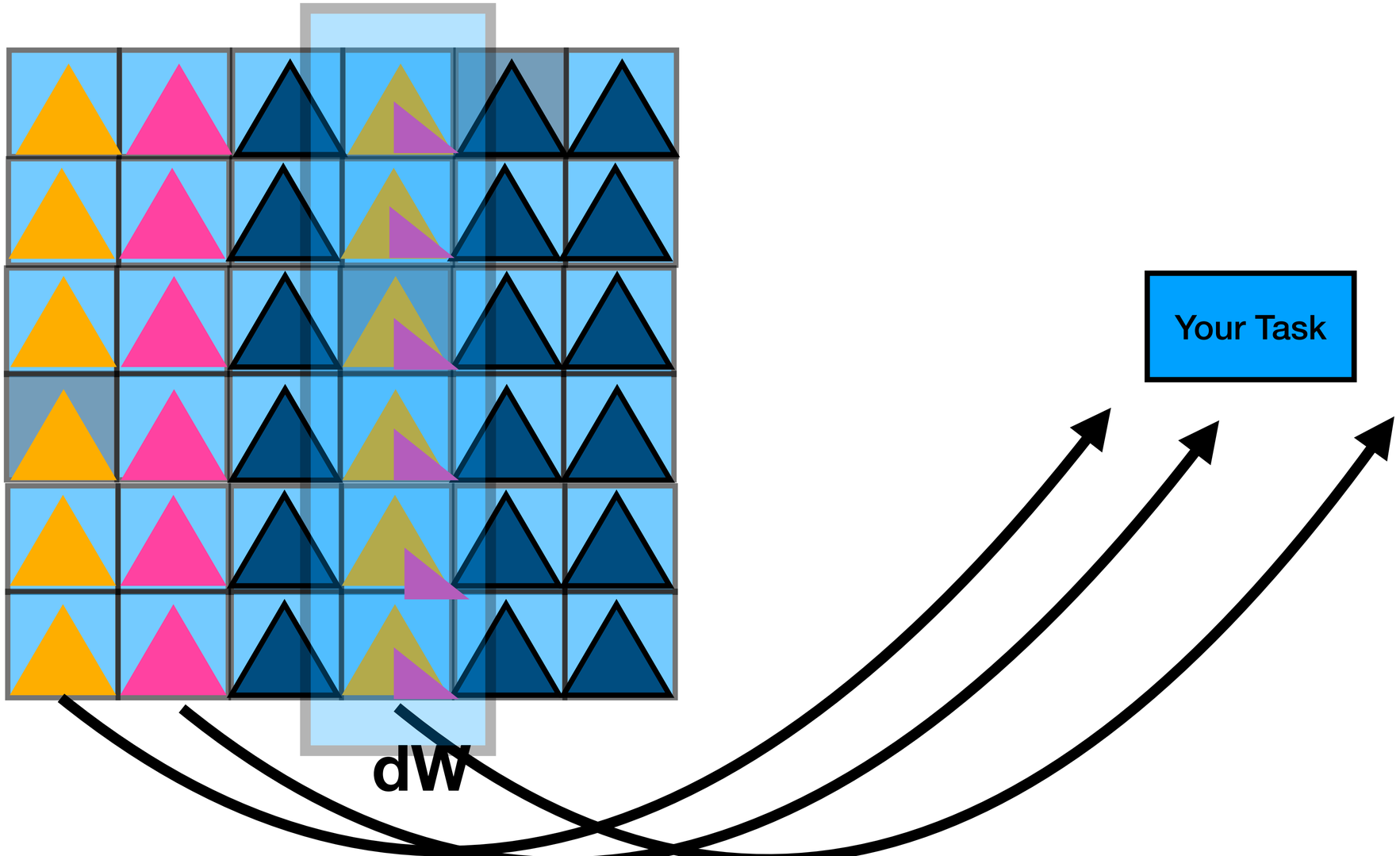


# Whats happening when Fine Tuning?

## Does it require to have the whole set of weights to adapt to your task?

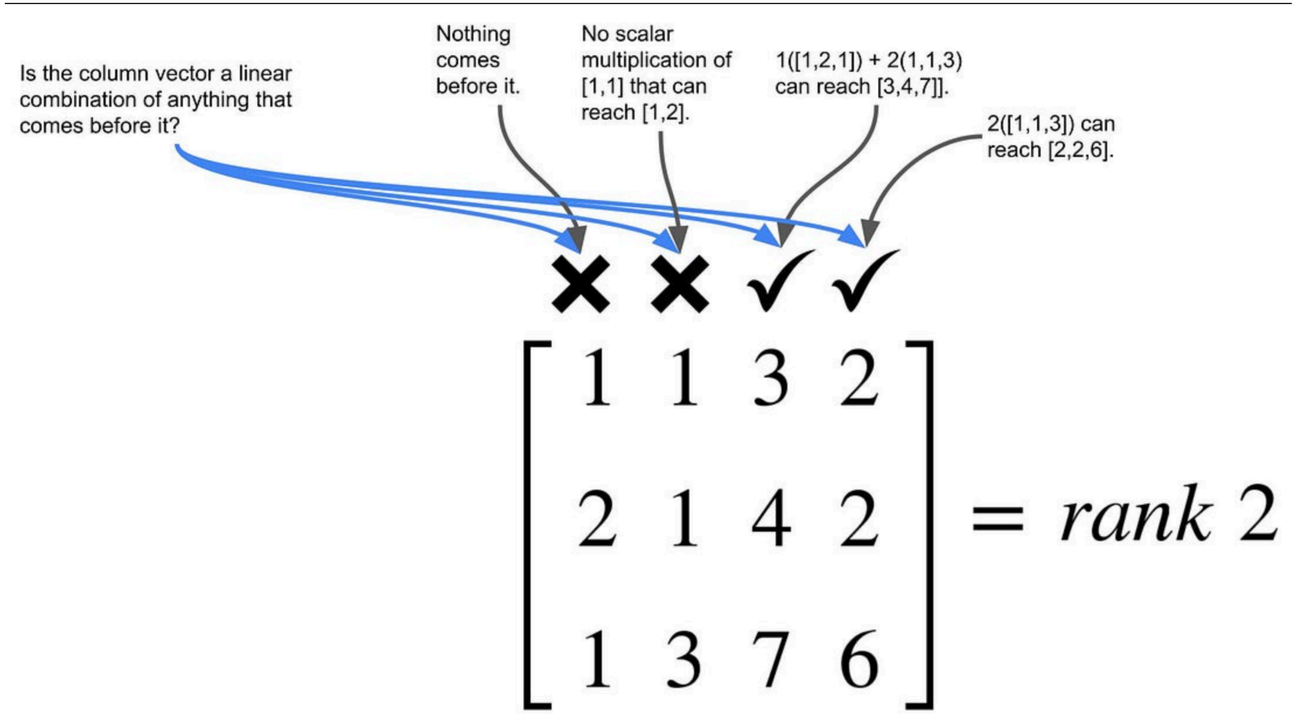


# What happens when a whole weights are fine-tuned?



# Rank of a Matrix

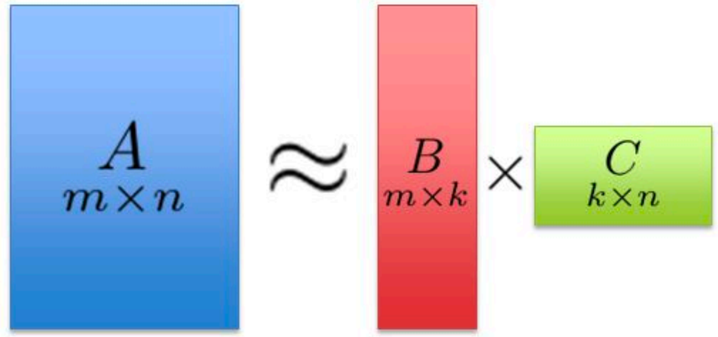
**The rank of a matrix is the number of linearly independent rows or columns in a matrix.**




## Low-Rank Decomposition: The Goal

$$\underset{m\times n}{A} \approx \underset{m\times k}{B}\ \underset{k\times n}{C}$$

where $k \ll \min\{m, n\}$



# LLM matrices have low rank

## when they are adapted to a new task (Large Language Models may become overkill!)

**Pre-trained large language models have a low “intrinsic dimension” when they are adapted to a new task, according to Aghajanyan et al. (2020)**

**Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning**

Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning

Armen Aghajanyan, Luke Zettlemoyer, Sonal Gupta

Facebook {armenag,lsz,sonalgupta}@fb.com

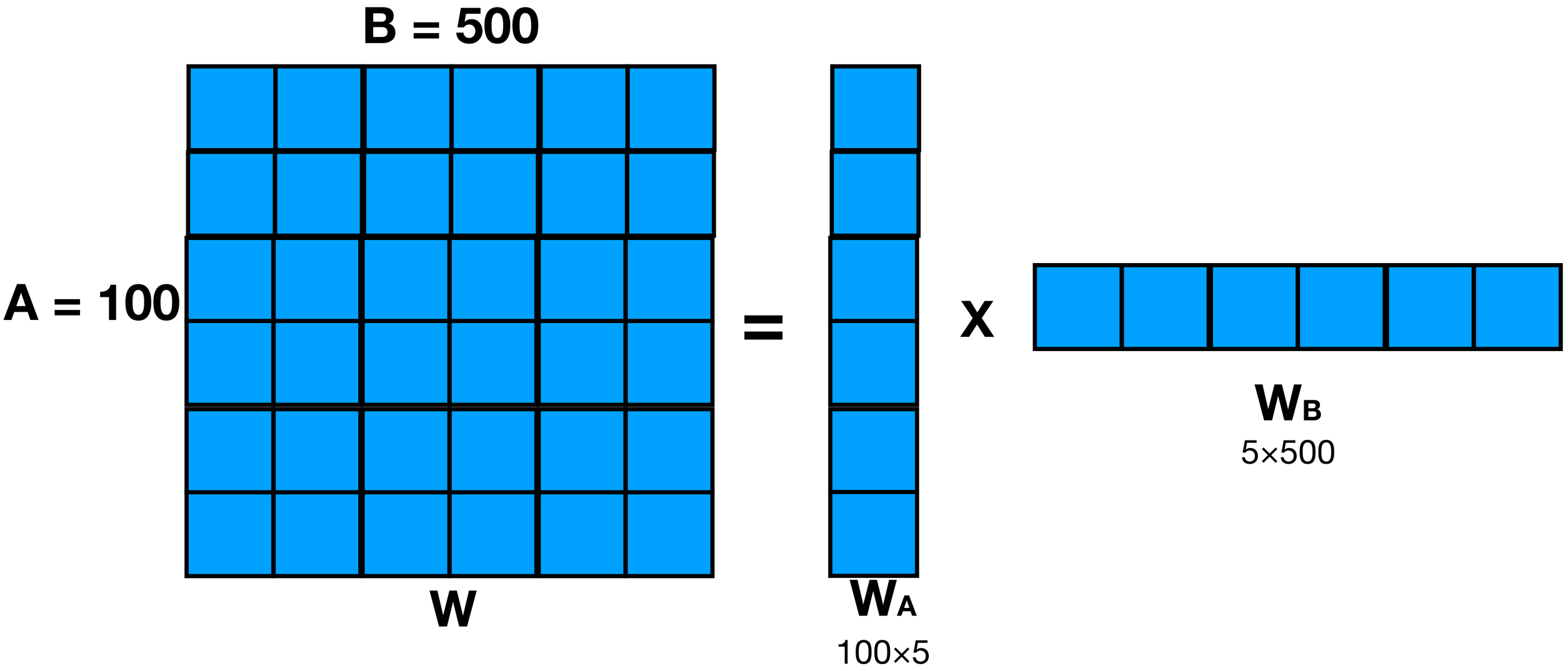
**Abstract**

Although pretrained language models can be fine-tuned to produce state-of-the-art results for a very wide range of language understanding tasks, the dynamics of this process are not well understood, especially in the low data regime. Why can we use relatively vanilla gradient descent algorithms (e.g., without strong regularization) to tune a model with hundreds of millions of parameters on datasets with only hundreds or thousands of labeled examples? In this paper, we argue that analyzing fine-tuning through the lens of intrinsic dimension provides us with empirical and theoretical intuitions to explain this remarkable phenomenon. We empirically show that common pre-trained models have a very low intrinsic dimension; in other words, there exists a low dimension reparameterization that is as effective for fine-tuning as the full parameter space. For example, by optimizing only 200 trainable parameters randomly projected back into the full space, we can tune a RoBERTa model to achieve 90% of the full parameter performance levels on MRPC. Furthermore, we empirically show that pre-training implicitly minimizes intrinsic dimension and, perhaps surprisingly, larger models tend to have lower intrinsic dimension after a fixed number of pre-training updates, at least in part explaining their extreme effectiveness. Lastly, we connect intrinsic dimensionality with low dimensional task representations and compression based generalization bounds to provide intrinsic-dimension-based generalization bounds that are independent of the full parameter count.

# Parameter efficiency

$W' = W + \Delta W$

$W' = W + W_A \times W_B$

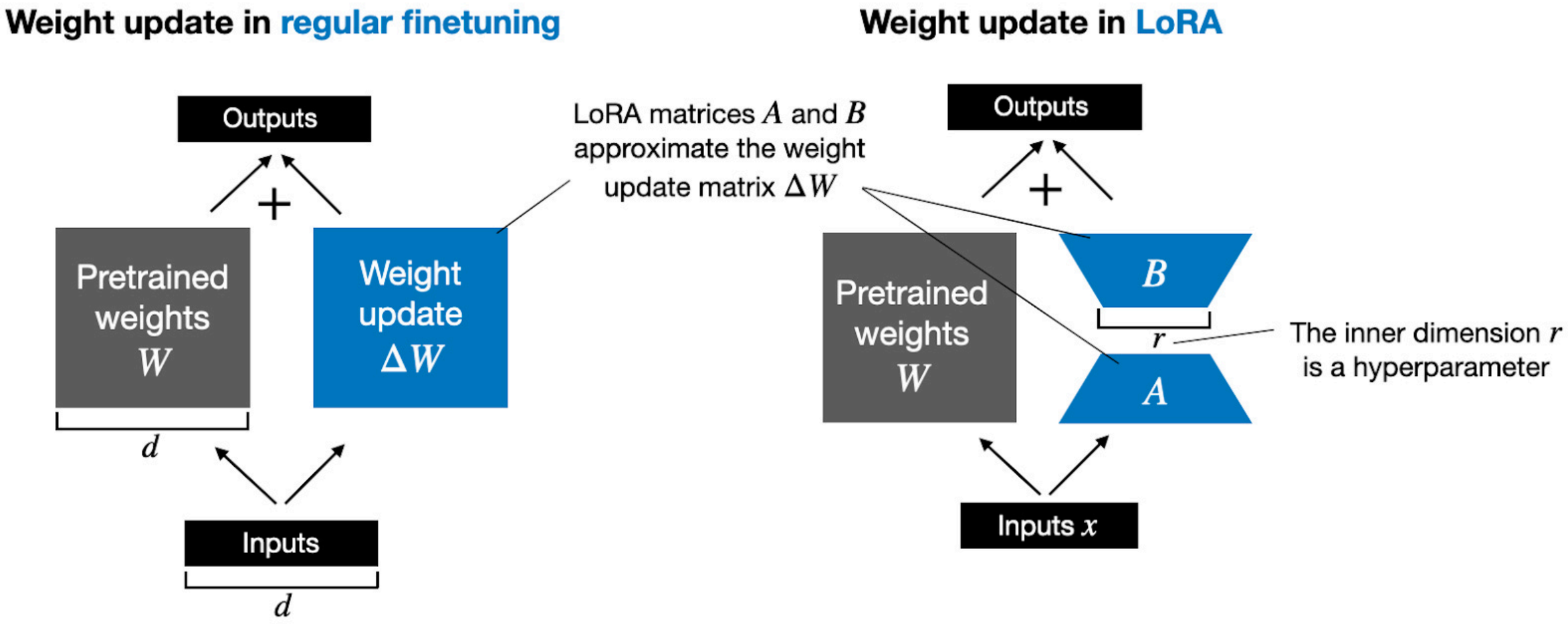



$\Delta W$ is 100×500=**50,000**

**100x5 + 5×500=3,000 parameters in total**

# Regular Finetuning vs LoRA

**7 billion parameter models can be finetuned efficiently within a few hours on a single GPU possessing 14 GB of RAM**



# Choosing the rank

**Its important to experiment with different rvalues to find the right balance**

- Rank, **r is hyperparameter** (to specify the rank of the low-rank matrices used for adaptation
- **Smaller r leads** to a simpler low-rank matrix, which results in **fewer parameters to learn during** adaptation
  - Can lead to **faster training** and potentially r**educed computational requirements**
  - may result in **lower adaptation quality**
- Smaller r is a **trade-off** between **model complexity, adaptation capacity**, and the risk of **underfitting or overfitting**

# Pseudo-code for Implementing LoRA

```
input_dim = 768  # e.g., the hidden size of the pre-trained model
output_dim = 768  # e.g., the output size of the layer
rank = 8  # The rank 'r' for the low-rank adaptation

W = ... # from pretrained network with shape input_dim x output_dim

W_A = nn.Parameter(torch.empty(input_dim, rank)) # LoRA weight A
W_B = nn.Parameter(torch.empty(rank, output_dim)) # LoRA weight B

# Initialization of LoRA weights
nn.init.kaiming_uniform_(W_A, a=math.sqrt(5))
nn.init.zeros_(W_B)

def regular_forward_matmul(x, W):
    h = x @ W
return h

def lora_forward_matmul(x, W, W_A, W_B):
    h = x @ W  # regular matrix multiplication
    h += x @ (W_A @ W_B) * alpha # use scaled LoRA weights
return h
```

**W_B is initialised to 0, so that ΔW = W_A X W_B = 0 at the beginning of the training**

**We can think of it as a modified forward pass for the fully connected layers in an LLM**

**alpha is a scaling factor that adjusts the magnitude of the combined result (original model output plus low-rank adaptation). This balances the pretrained model's knowledge and the new task-specific adaptation — by default, alpha is usually set to 1**

- **Reducing inference overhead**
- **A full 7B LLaMA checkpoint requires 23 GB of storage capacity, while the LoRA weights can be as small as 8 MB if we choose a rank of r = 8**

# Results

LoRA can even outperform full finetuning training only 2% of the parameters

Full finetuning

Only tune bias vectors

Prompt tuning

Prefix tuning

| Model&Method | # Trainable Parameters | WikiSQL Acc. (%) | MNLI-m Acc. (%) | SAMSum R1/R2/RL |
|---|---|---|---|---|
| GPT-3 (FT) | 175,255.8M | **73.8** | 89.5 | 52.0/28.0/44.5 |
| GPT-3 (BitFit) | 14.2M | 71.3 | 91.0 | 51.3/27.4/43.5 |
| GPT-3 (PreEmbed) | 3.2M | 63.1 | 88.6 | 48.3/24.2/40.5 |
| GPT-3 (PreLayer) | 20.2M | 70.1 | 89.5 | 50.8/27.3/43.5 |
| GPT-3 ($\text{Adapter}^{\text{H}}$) | 7.1M | 71.9 | 89.8 | 53.0/28.9/44.8 |
| GPT-3 ($\text{Adapter}^{\text{H}}$) | 40.1M | 73.2 | **91.5** | 53.2/29.0/45.1 |
| GPT-3 (LoRA) | 4.7M | 73.4 | **91.7** | **53.8/29.8/45.9** |
| GPT-3 (LoRA) | 37.7M | **74.0** | **91.6** | 53.4/29.2/45.1 |

ROUGE scores

Table 4: Performance of different adaptation methods on GPT-3 175B. We report the logical form validation accuracy on WikiSQL, validation accuracy on MultiNLI-matched, and Rouge-1/2/L on SAMSum. LoRA performs better than prior approaches, including full fine-tuning. The results on WikiSQL have a fluctuation around $\pm 0.5\%$, MNLI-m around $\pm 0.1\%$, and SAMSum around $\pm 0.2/\pm 0.2/\pm 0.1$ for the three metrics.

# QLoRA Compute-Memory Trade-offs

## QLoRA: short for quantized LoRA

- During backpropagation, QLoRA quantizes the pretrained weights to 4-bit precision.

- It was found that one can save **33% of GPU** memory when using QLoRA. However, this comes at a **39% increased training runtime**.

| | Training time | Memory used |
|---|---|---|
| **Default LoRA with 16-bit precision** | **1.85 h** | **21.33 GB** |
| **QLoRA with 4-bit precision** | **2.79 h** | **14.18 GB** |

- The modeling performance was barely affected! Learn more here:

# References


- https://sebastianraschka.com/blog/2023/llm-finetuning-lora.html
- https://ar5iv.labs.arxiv.org/html/2106.09685
- https://twitter.com/rasbt/status/1712816975083155496
- https://lightning.ai/pages/community/lora-insights/?utm_medium=social&utm_source=twitter&utm_campaign=Education_10132023
- https://magazine.sebastianraschka.com/p/practical-tips-for-finetuning-llms