# Week 2

fine tuning for a single task → CATASTROPHIC FORGETTING (degrade performance on other task)

1 you are ok with it

2 Fine tune on multiple tasks (50 000+ examples) → still 1 model
- Summarize
- Rate
- Identify the places

3 PARAMETER EFFICIENT FINE-TUNING

preserve weights of the original LLM & trains only a small number of task specific adapter layers & parameters

INSTRUCTION fine-tuning

model weights are updated based on labeled data with prompt completion example

FLAN specific set of instruction used to perform fine tuning

Samsun example of dataset on dialogues

---

## Evaluation Metrics

ROUGE → used for text summarization

Recall: $\frac{\text{bigram unigram}^x \text{ matches}}{\text{unigram}^x \text{ in reference}}$

Precision: $\frac{\text{unigram}^x \text{ matches}}{\text{unigram}^x \text{ in output}}$

$ROUGE_1$ → only unigrams → don't consider ordering or negation

F1 → harmonic mean of the 2

Rouge 2 → uses BIGRAMS

Rouge L → length of the longest matching subsequences

BLEU SCORE → used for text translation → average precision over multiple n-gram sizes

EVALUATION BENCHMARKS

- Glue → collection of natural language tasks (SA, Q&A)
  → SuperGlue (2019)
- MMLU ( designed for modern LLM, 2021)
- BigBench (204 tasks, 2022)
- Helm framework → trasparency of models → guideline on which one performs best at each task
  → 7 metrics across 16 scenarios, fairness bias toxicity

---

# PEFT (parameter efficient fine-tuning)

- most / all of original LLM weights are frozen, 15-20% of original LLM weights are trained
- memory requirements are more manageable ( single GPU!)
- less prone to catastrophic forgetting
- new parameters for each task are combined w/ original ones for inference
- 3 main classes

**Selective**

fine tune only a subset of original LLM param.

→ certain components
→ specific layers
→ individual param type

**Reparametrization**

Reduce # of original param by creating new low rank transformations of network weights

↓

LoRA

**Additive**

introducing new trainable layers

Adapters
↳ new layers after Attention OR FF layer

Soft Prompts
↳ manipulate the input for better performance

# LoRA

1 freeze weights of the self-Attention layer (biggest savings in trainable param, since the ↑ # is here, not FF)

2 Inject 2 rank decomposition matrices → their product has the same dimensions as the weights

3 update only param in those matrices during training

4 Add updated weights to the original ones. (you can switch them at inference time!)

5 Now the model is LoRA fine tuned!

What RANK to choose?

→ plateau after 16

→ 4-32 good trade off between reducing trainable parameters & preserving performance

---

# Prompt Tuning

→ ≠ in-context learning prompt engineering trying different phrases or including examples

Limits length of context window, lots of manual effort

⇩

→ the weights of LLM are frozen (prepended to the embedding vector representing the input text)

→ additional trainable tokens are added to your prompt ↗

soft prompt

→ supervised learning is used to determine their optimal values

→ can take any value in the multidimensional embedding space

→ 10-100K param. updated vs Million to Billions

→ train a different set of soft prompts for each task and swap them at inference time