Core components of pytorch

1. Tensor library for efficient computing
2. Automatic differentiation engine – utilities to differentiate computations automatically
3. Deep learning library

Defining Deep learning:

1. If cuda compatible gpu is found , torch for gpu will be automatically installed.
2. pip install torch torchvision torchaudio --index-url <https://download.pytorch.org/whl/cu126> [126 is cuda 12.6]
3. nvidia-smi =>The **nvidia-smi** (NVIDIA System Management Interface) command displays information about your **GPU**, including driver version, CUDA version, memory usage, and running processes.
4. Float 32 is generated by default and its optimized for GPU
5. Most operations are similar to GPU
6. If we carry out the computations in Pytorch , it will build a computational graph internally by default if one of its terminal nodes has requires\_grad =True
7. By default ..after calculating the gradient , the computational graph is destroyed but here using retain\_graph=True, we retain the graph as we do multiple grad calculations
8. We can ask the model not to compute the computation graph during inference , hence use torch.no\_grad()
9. For every random initialize use torch.manual\_Seed(number)
10. We don’t pass last layer to activation function in pytorch as its automatically taken care in loss functions, but for inference we can explicitly call those functions [softmax]
11. Dataset class defines how individual data records are loaded [we created tensor dataset that sits in memory]
12. DataLoader class defines dataset shuffling,batches and more
13. Cpu load and preprocess data , GPu waits if num workers =0
14. Else parallell processing can be done
15. Optimal training is num\_workers=4
16. Torch.save(model.state\_dict(),”model.pth”) -- dict object maps each layer to trainable params.. .pth , .pt are most common … model.load\_state\_dict(‘mlp.pth’) loads these params to the model of same architecture
17. GPU training – torch.cuda.is\_available() … tensor\_1.to(“cuda”) if multiple GPU … tensor\_2.to(“cuda:0”)
18. All tensors must be on same device . otherwise the computation will fail, where one tensor resides on CPU and other on GPU
19. Pytorch Distributed Data Parallel [DDP] splits input data and do the processing
20. Each gpu will receive a model,each model will receive a mini batch[distributed sampler ] non overlapping samples, get logits and gradients then synch up do weight update,
21. Multiprocessing works differently in scripts and jupyter notebook
22. Hugging face accelerate support linux os … work in colab notebook … mixed precision / deepseek
23. Accelerate handles the device placement for you, so you can remove the lines that put the model on the device (or, if you prefer, change them to use accelerator.device instead of device).
24. Then the main bulk of the work is done in the line that sends the dataloaders, the model, and the optimizer to accelerator.prepare(). This will wrap those objects in the proper container to make sure your distributed training works as intended.
25. **collate function**
26. You can pass the collate function as an argument of a DataLoader. We used the DataCollatorWithPadding function, which pads all items in a batch so they have the same length. Collate function puts together all the samples in a batch.
27. **Chat Templates:**

Base Model : predict next token

Instruct : finetuned model to follow instruction

1. **. C**hatML is one such template format that structures conversations with clear role indicators (system, user, assistant).
2. Each model chat template format for instruct differs on system message handling,message boundaries and special tokens
3. Chat templates can handle more complex scenarios beyond just conversational interactions such as tool use,multimodal inputs , function calling, multi-turn context[maintain history]
4. Key practices while using chat templates :

* Consistent formatting
* Clear role definition
* Context management[token limit]
* Error handling
* Validation

1. Supervised finetuning :

* SFT allows precise control over model’s output structure. [Generate responses in a specific chat template format, follow strict output schemas, maintain consistent styling across responses]
* Domain adaptation

1. **Data preparation**

* Input-output pair … [Input prompt,expected model response,context/metdata]

1. **SFTT Parameters**

* Training Duration parameters: num\_train\_epochs,max\_steps
* Batch Size parameters: per\_Device\_train\_batch\_size,gradient\_accumulation\_steps
* Learning rate parameters: learning\_rate,warmup\_ratio
* Monitorinh : logging steps,eval\_steps,save\_steps

1. **When using a dataset with a "messages" field (like the example above), the SFTTrainer automatically applies the model's chat template, which it retrieves from the hub. This means you don't need any additional configuration to handle chat-style conversations - the trainer will format the messages according to the model's expected template format.**
2. Packing the dataset:

**The SFTTrainer supports example packing to optimize training efficiency. This feature allows multiple short examples to be packed into the same input sequence, maximizing GPU utilization during training. To enable packing, simply set packing=True in the SFTConfig constructor. When using packed datasets with max\_steps, be aware that you may train for more epochs than expected depending on your packing configuration. You can customize how examples are combined using a formatting function - particularly useful when working with datasets that have multiple fields like question-answer pairs. For evaluation datasets, you can disable packing by setting eval\_packing=False in the SFTConfig.**

1. **Metrics to monitor :** Training Loss, Validation Loss, Learning Rate Progression, Gradient Norms
2. **After completing SFT, consider these follow-up actions:**

* **Evaluate the model thoroughly on held-out test data**
* **Validate template adherence across various inputs**
* **Test domain-specific knowledge retention**
* **Monitor real-world performance metrics**

1. **Gradient\_Accumulation\_steps: To** Increase effective batch size without using more memory.
2. **LORA :**

* Only adapter params are stored in GPU
* Base model frozen and loaded in lower precision
* Merge adapters into base model

1. LORA Configuration

* I [rank] – Dimension of low rank matrices , used for weight updates .Between 4-32.
* Lora\_alpha –
* Lora dropout - Dropout probability for LoRA layers, typically 0.05-0.1. Higher values help prevent overfitting during training.
* Bias - Options are “none”, “all”, or “lora\_only”. “none” is most common for memory efficiency.
* Target-modules -  Can be “all-linear” or specific modules like “q\_proj,v\_proj”. More modules enable greater adaptability but increase memory usage.
* When implementing PEFT methods, start with small rank values (4-8) for LoRA and monitor training loss. Use validation sets to prevent overfitting and compare results with full fine-tuning baselines when possible. The effectiveness of different methods can vary by task, so experimentation is key.

1. After training with LoRA, you might want to merge the adapter weights back into the base model for easier deployment. This creates a single model with the combined weights, eliminating the need to load adapters separately during inference.
2. Evaluation:

* General Knowledge benchmarks [MMLU,TruthfulQA]
* Light eval can be used to automate on standard benchmark datasets
* Function calling is already available in Mistral Models
* User: specify tools and query
* Model: Generate function arguments if applicable
* User: Execute function to obtain tool results
* Model: Generate final answer
* Provide tools in json
* Message has tools json,userquery
* Returns – tool with arguments to call …append in messages
* Execute and append in messages
* Finally we get the required response

1. Model Training :

* Pretrain
* Finetune to follow instruction
* Aligned to customer preference .. cant be impolite

1. We use instruct model to fine tune