Fine-Tuning LLMs on Cloud: Detailed Guide with Examples and Code

Fine-tuning large language models (LLMs) on the cloud enables elastic compute, scale-out, and reproducible pipelines. This guide covers end-to-end cloud fine-tuning on three major platforms—AWS SageMaker, Google Cloud Vertex AI, and Azure ML—explaining every component, configuration, and code sample.

1. Common Components and Workflow

Across providers, the high-level steps are similar:

1. **Prepare and Upload Data**
   - Clean, split, and format your dataset (JSONL or CSV) for instruction tuning or Q/A.
   - Upload to cloud object storage (S3, GCS, Azure Blob).
2. **Containerize Training Script**
   - Write a Python training script that loads data, tokenizes, applies LoRA/QLoRA adapters, and runs training with `transformers`/`trl` or Unsloth.
   - Package dependencies in a Docker image or use curated deep learning frameworks.
3. **Configure Compute Resources**
   - Choose GPU instances (e.g., AWS p3/p4, GCP A2, Azure NC/ND) sized for memory and performance.
   - Specify multi-GPU or distributed data-parallel setup if needed.
4. **Launch Training Job**
   - Submit a training job via CLI, SDK, or console, pointing to your script, image, and data URI.
   - Set environment variables or hyperparameters (learning rate, LoRA rank, batch size, epochs).
5. **Monitor and Log**
   - Integrate with TensorBoard or W&B using built-in integrations.
   - View logs, GPU utilization, training/validation metrics in real time.
6. **Retrieve and Deploy Model**
   - After training completes, model artifacts (merged weights) are output to object storage.
   - Deploy as an endpoint (serverless or container) for inference.

2. AWS SageMaker Example

2.1 Data Preparation and Upload

```
# Clean and split locally with Hugging Face datasets
python split_and_format.py \
  --input cleaned_combined_dataset.jsonl \
  --train-out train.jsonl \
  --val-out val.jsonl \
  --test-out test.jsonl

# Upload to S3
aws s3 cp train.jsonl s3://my-bucket/datasets/medical/train.jsonl
aws s3 cp val.jsonl   s3://my-bucket/datasets/medical/val.jsonl
```

2.2 Training Script (`train.py`)

```
import os
import torch
from datasets import load_dataset
from transformers import AutoTokenizer
from trl import SFTTrainer
from transformers import TrainingArguments
from peft import get_peft_model, LoraConfig, TaskType

# 1. Load dataset from S3
train_ds = load_dataset("json", data_files={"train": os.environ["SM_CHANNEL_TRAIN"]})["tr
val_ds   = load_dataset("json", data_files={"validation": os.environ["SM_CHANNEL_VAL"]})[

# 2. Load model & tokenizer, apply 4-bit QLoRA quantization
from transformers import AutoModelForCausalLM, BitsAndBytesConfig
bnb_config = BitsAndBytesConfig(load_in_4bit=True, bnb_4bit_quant_type="nf4", bnb_4bit_us
model = AutoModelForCausalLM.from_pretrained("mistralai/Mistral-7B-v0.3", quantization_co
tokenizer = AutoTokenizer.from_pretrained("mistralai/Mistral-7B-v0.3")
tokenizer.pad_token = tokenizer.eos_token

# 3. Add LoRA adapters
lora_cfg = LoraConfig(
    task_type=TaskType.CAUSAL_LM,
    r=int(os.environ["SM_ HPARAM_LORA_R"]),
    lora_alpha=int(os.environ["SM_HP_LORA_ALPHA"]),
    target_modules=["q_proj","k_proj","v_proj","o_proj"],
    lora_dropout=float(os.environ["SM_HP_LORA_DROPOUT"]),
)
model = get_peft_model(model, lora_cfg)

# 4. Prepare Trainer arguments
training_args = TrainingArguments(
    output_dir=os.environ["SM_OUTPUT_DIR"],
    per_device_train_batch_size=int(os.environ["SM_HP_BATCH_SIZE"]),
    gradient_accumulation_steps=int(os.environ["SM_HP_ACCUM_STEPS"]),
    num_train_epochs=int(os.environ["SM_HP_EPOCHS"]),
```

```
    learning_rate=float(os.environ["SM_HP_LR"]),
    logging_steps=50,
    evaluation_strategy="steps",
    eval_steps=100,
    save_total_limit=3,
    fp16=True,
    report_to=["tensorboard","wandb"],
)

trainer = SFTTrainer(
    model=model,
    tokenizer=tokenizer,
    train_dataset=train_ds,
    eval_dataset=val_ds,
    dataset_text_field="text",
    args=training_args,
)

if __name__ == "__main__":
    trainer.train()
    # Merge adapters and save full model
    from peft import PeftModel
    merged = PeftModel.merge_and_unload(model)
    merged.save_pretrained(os.environ["SM_MODEL_DIR"])
```

2.3 Dockerfile

```
FROM pytorch/pytorch:2.0-cuda11.7-cudnn8-runtime
RUN pip install transformers datasets accelerate bitsandbytes peft trl evaluate wandb
COPY train.py /opt/ml/code/train.py
ENV PYTHONUNBUFFERED=TRUE
ENTRYPOINT ["python","/opt/ml/code/train.py"]
```

2.4 Launch SageMaker Training

```
import sagemaker
from sagemaker.estimator import Estimator

role = sagemaker.get_execution_role()
image_uri = "<YOUR_ECR_IMAGE_URI>"

estimator = Estimator(
    image_uri=image_uri,
    role=role,
    instance_count=1,
    instance_type="ml.p4d.24xlarge",
    volume_size=200,
    hyperparameters={
        "batch_size": 4,
        "accum_steps": 8,
        "epochs": 3,
        "lr": 2e-4,
        "lora_r": 16,
```

```
        "lora_alpha": 32,
        "lora_dropout": 0.05,
    },
    input_mode="File",
    output_path="s3://my-bucket/models/medical-mistral/",
)

# Specify data channels
estimator.fit({
    "train": "s3://my-bucket/datasets/medical/train.jsonl",
    "validation": "s3://my-bucket/datasets/medical/val.jsonl",
})
```

3. Google Cloud Vertex AI Example

3.1 Data in GCS

```
gsutil cp train.jsonl gs://my-bucket/datasets/medical/train.jsonl
gsutil cp val.jsonl   gs://my-bucket/datasets/medical/val.jsonl
```

3.2 Training Script

Use the same `train.py` above, parameterized via environment variables.

3.3 Submit Custom Job

```
gcloud ai custom-jobs create \
  --region=us-central1 \
  --display-name=medical-mistral-ft \
  --worker-pool-spec=machine-type=a2-highgpu-1g,replica-count=1,executor-image-uri=gcr.io
  --args="--SM_CHANNEL_TRAIN=gs://my-bucket/datasets/medical/train.jsonl","--SM_CHANNEL_V
```

Vertex AI will handle provisioning TPU/GPUs, mounting GCS, and running your container.

4. Azure ML Example

4.1 Data in Azure Blob

```
az storage blob upload -c datasets --account-name mystorageaccount --file train.jsonl --n
az storage blob upload -c datasets --account-name mystorageaccount --file val.jsonl --nam
```

4.2 Define Azure ML Components

```
from azure.ai.ml import MLClient
from azure.ai.ml.entities import (
    Environment, CommandComponent, AmlCompute, BatchEndpoint, BatchDeployment
)
from azure.identity import DefaultAzureCredential

ml_client = MLClient(DefaultAzureCredential(), subscription_id, resource_group, workspace

# Define environment
env = Environment(name="llm-finetune-env")
env.docker.base_image = "mcr.microsoft.com/azureml/openmpi4.1.0-ubuntu20.04:latest"
env.python.user_managed_dependencies = False
env.python.conda_dependencies.add_pip_package("transformers")
# ... add other packages

# Define compute
compute = AmlCompute(name="gpu-cluster", size="STANDARD_ND40rs_v2", min_instances=0, max_
ml_client.begin_create_or_update(compute).result()

# Define component
component = CommandComponent(
    name="llm-finetune",
    environment=env,
    code=".",  # directory with train.py
    command="""python train.py \
      --SM_CHANNEL_TRAIN azureml://datastores/workspaceblob/paths/medical/train.jsonl \
      --SM_CHANNEL_VAL   azureml://datastores/workspaceblob/paths/medical/val.jsonl \
      --SM_HP_LR=2e-4 --SM_HP_BATCH_SIZE=4""",
    instance_count=1,
    instance_type="STANDARD_ND40rs_v2",
)
ml_client.components.create_or_update(component)
```

4.3 Submit Job

```
from azure.ai.ml import Input

job = ml_client.create_or_update_job(
    component,
    inputs={
      "train": Input(type="uri_file", path="azureml://datastores/workspaceblob/paths/medi
      "validation": Input(type="uri_file", path="azureml://datastores/workspaceblob/paths
    }
)
ml_client.jobs.stream(job.name)
```

5. Monitoring, Logging, and Cleanup

- **TensorBoard**: Mount the log directory as a volume and forward port to view `localhost:6006`.
- **Weights & Biases**: Runs are logged automatically; view online dashboards.
- **Cost Management**: Tear down compute when done; configure spot/preemptible instances to reduce costs.

6. Key Configuration and Best Practices

Aspect	Recommendation
Instance Type	Use A100/P4d for large models; use mixed GPU clusters for distributed training
Spot/Preemptible	Leverage spot instances (Delta in SageMaker/Vertex AI/Azure ML) to reduce cost by 50–70%
Data I/O	Use high-throughput object storage (S3/GCS/Blob) and caching to minimize bottlenecks
Hyperparameter	Pass via environment variables or CLI args; track via W&B or TensorBoard
Distributed Training	Use DeepSpeed or PyTorch DistributedDataParallel for multi-node jobs
Security	Use IAM roles/service accounts with least privilege; encrypt data at rest and in transit
Reproducibility	Version control Docker image, script, dataset, and track experiments via ML metadata

By following this guide and adapting the code samples to your project, you can reliably and efficiently fine-tune LLMs at scale on any major cloud platform, with full transparency of every step, parameter, and best practice.