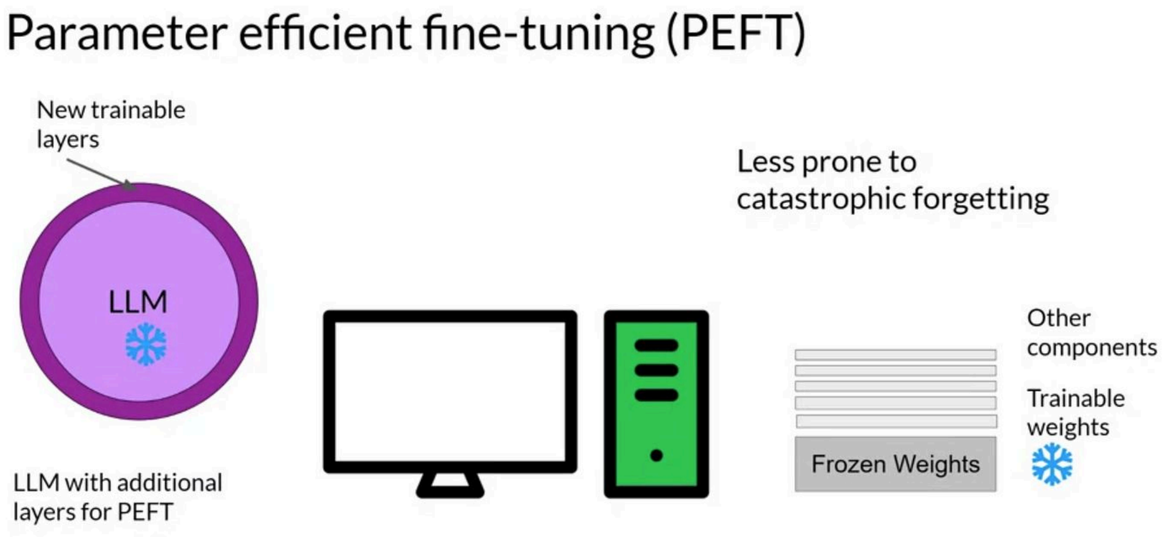
AI BIOINNOVATE
Problem Statement PS1 Protein Engineering

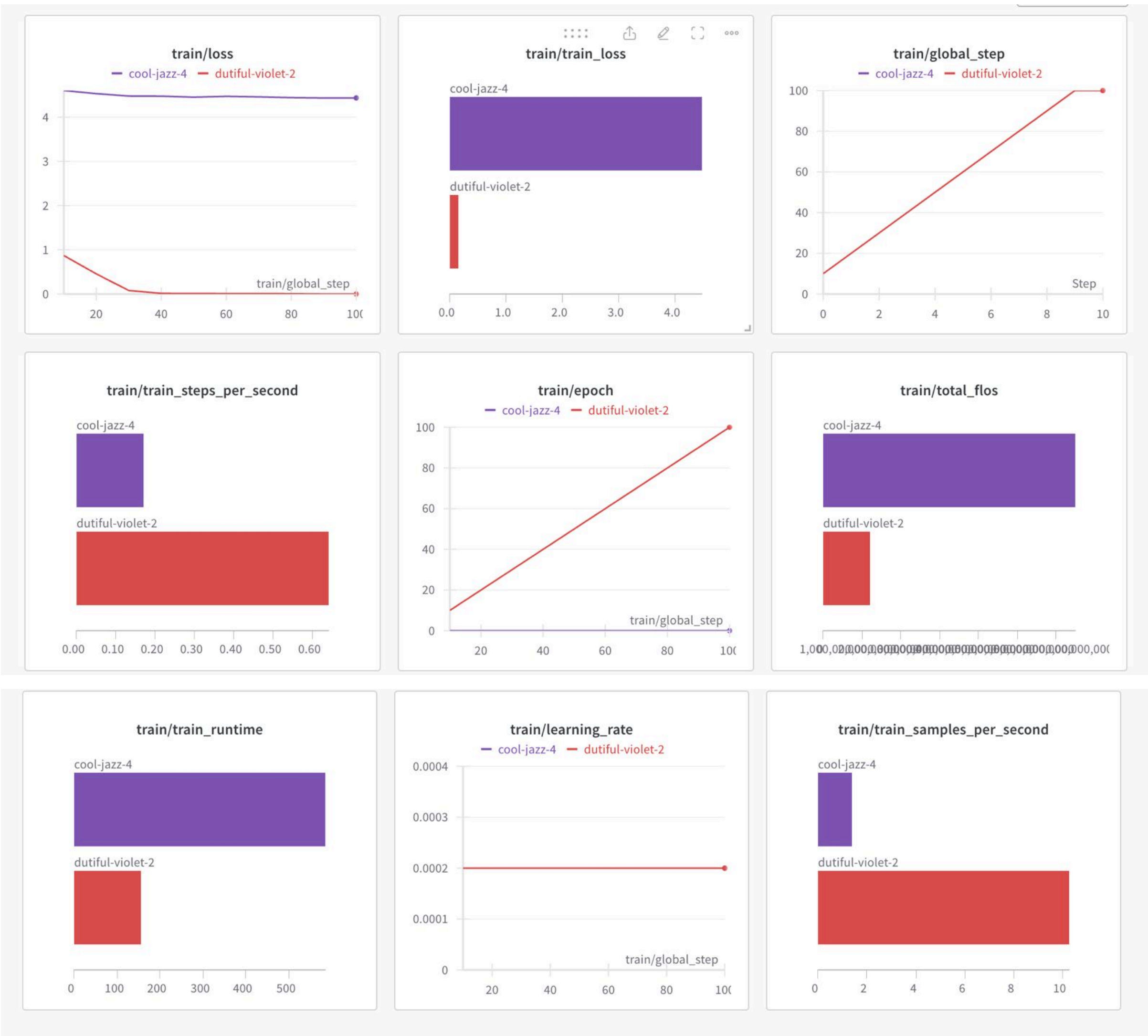
- *Protein Engineering*

  PE1 - Design and implement a model that can generate realistic and meaningful protein sequences based on a few given examples. Participants are provided with a small dataset of protein sequences for training, and they need to fine-tune a pre-trained language model or develop a novel model to achieve accurate and diverse protein sequence generation.

  1. Dataset : dataset_ENZ

Team Name Thunderflow
Team Members :
Mihir Panchal
Arnav Panicker
Hemant Singh





1. Setup and Library Installation

In [1]:

```
!pip install -q -U trl transformers accelerate git+https://github.com/huggingface/peft.git
!pip install -q datasets bitsandbytes einops wandb
```

```
Installing build dependencies ... done
Getting requirements to build wheel ... done
Preparing metadata (pyproject.toml) ... done
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 150.9/150.9 kB 3.0 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 8.2/8.2 MB 30.2 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 270.9/270.9 kB 17.5 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 507.1/507.1 kB 28.8 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 79.6/79.6 kB 8.7 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 115.3/115.3 kB 15.6 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 134.8/134.8 kB 17.1 MB/s eta 0:00:00
Building wheel for peft (pyproject.toml) ... done
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 105.0/105.0 MB 7.2 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 44.6/44.6 kB 5.3 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.2/2.2 MB 82.1 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 196.4/196.4 kB 22.8 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 254.1/254.1 kB 28.5 MB/s eta 0:00:00
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 62.7/62.7 kB 8.2 MB/s eta 0:00:00
```

This code installs necessary libraries and dependencies. The libraries include trl, transformers, accelerate, peft, datasets, bitsandbytes, einops, and wandb.

2. Dataset Loading

Dataset

In [2]:
```
from datasets import load_dataset

dataset_name = 'Arnav2612/Proteins'
dataset = load_dataset(dataset_name, split="train")
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:88: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), se
t it as secret in your Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
Downloading readme:   0%|          | 0.00/21.0 [00:00<?, ?B/s]
Downloading data:   0%|          | 0.00/44.7M [00:00<?, ?B/s]
Generating train split: 0 examples [00:00, ? examples/s]
```

This code uses the Hugging Face datasets library to load a protein dataset named 'Arnav2612/Proteins' and specifies the training split.

3. Model Loading

Loading the model

In [3]:
```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfig, AutoTokenizer

model_name = "TinyPixel/Llama-2-7B-bf16-sharded"

bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.float16,
)

model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    trust_remote_code=True
)
model.config.use_cache = False
```

```
config.json:   0%|          | 0.00/626 [00:00<?, ?B/s]
model.safetensors.index.json:   0%|          | 0.00/28.1k [00:00<?, ?B/s]
Downloading shards:   0%|          | 0/14 [00:00<?, ?it/s]
model-00001-of-00014.safetensors:   0%|          | 0.00/981M [00:00<?, ?B/s]
model-00002-of-00014.safetensors:   0%|          | 0.00/967M [00:00<?, ?B/s]
model-00003-of-00014.safetensors:   0%|          | 0.00/967M [00:00<?, ?B/s]
model-00004-of-00014.safetensors:   0%|          | 0.00/990M [00:00<?, ?B/s]
model-00005-of-00014.safetensors:   0%|          | 0.00/944M [00:00<?, ?B/s]
model-00006-of-00014.safetensors:   0%|          | 0.00/990M [00:00<?, ?B/s]
model-00007-of-00014.safetensors:   0%|          | 0.00/967M [00:00<?, ?B/s]
model-00008-of-00014.safetensors:   0%|          | 0.00/967M [00:00<?, ?B/s]
model-00009-of-00014.safetensors:   0%|          | 0.00/990M [00:00<?, ?B/s]
model-00010-of-00014.safetensors:   0%|          | 0.00/944M [00:00<?, ?B/s]
model-00011-of-00014.safetensors:   0%|          | 0.00/990M [00:00<?, ?B/s]
model-00012-of-00014.safetensors:   0%|          | 0.00/967M [00:00<?, ?B/s]
model-00013-of-00014.safetensors:   0%|          | 0.00/967M [00:00<?, ?B/s]
model-00014-of-00014.safetensors:   0%|          | 0.00/847M [00:00<?, ?B/s]
```

The code loads a pre-trained model for Causal Language Modeling (AutoModelForCausalLM) named "TinyPixel/Llama-2-7B-bf16-sharded". It also uses the BitsAndBytesConfig to enable 4-bit quantization.

4. Tokenizer Loading

In [4]:
```
tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)
tokenizer.pad_token = tokenizer.eos_token
```

```
tokenizer_config.json:   0%|          | 0.00/676 [00:00<?, ?B/s]
tokenizer.model:   0%|          | 0.00/500k [00:00<?, ?B/s]
tokenizer.json:   0%|          | 0.00/1.84M [00:00<?, ?B/s]
special_tokens_map.json:   0%|          | 0.00/411 [00:00<?, ?B/s]
```

This code loads the tokenizer corresponding to the pre-trained model and sets the padding token to the end-of-sequence token.

5. PEFT Model Configuration

In [5]:
```
from peft import LoraConfig, get_peft_model

lora_alpha = 16
lora_dropout = 0.1
lora_r = 64

peft_config = LoraConfig(
    lora_alpha=lora_alpha,
    lora_dropout=lora_dropout,
    r=lora_r,
    bias="none",
    task_type="CAUSAL_LM"
)
```

The code sets up the configuration for the PEFT (Probabilistic Embeddings for Fine-Tuning) model using LoraConfig.

6. Trainer Setup

Loading the trainer

Here we will use the `SFTTrainer` from TRL library that gives a wrapper around transformers `Trainer` to easily fine-tune models on instruction based datasets using PEFT adapters. Let's first load the training arguments below.

In [6]:
```
from transformers import TrainingArguments

output_dir = "./results"
per_device_train_batch_size = 2
gradient_accumulation_steps = 4
optim = "paged_adamw_32bit"
save_steps = 100
logging_steps = 10
learning_rate = 2e-4
max_grad_norm = 0.3
max_steps = 100
warmup_ratio = 0.03
lr_scheduler_type = "constant"

training_arguments = TrainingArguments(
    output_dir=output_dir,
    per_device_train_batch_size=per_device_train_batch_size,
    gradient_accumulation_steps=gradient_accumulation_steps,
    optim=optim,
    save_steps=save_steps,
    logging_steps=logging_steps,
    learning_rate=learning_rate,
    fp16=True,
    max_grad_norm=max_grad_norm,
    max_steps=max_steps,
    warmup_ratio=warmup_ratio,
    group_by_length=True,
    lr_scheduler_type=lr_scheduler_type,
)
```

This code sets up training arguments using TrainingArguments such as batch size, optimization algorithm, learning rate, etc.

7. SFTTrainer Initialization

Then finally pass everthing to the trainer

In [7]:
```
from trl import SFTTrainer

max_seq_length = 256

trainer = SFTTrainer(
    model=model,
    train_dataset=dataset,
    peft_config=peft_config,
    dataset_text_field="sequence",
    max_seq_length=max_seq_length,
    tokenizer=tokenizer,
    args=training_arguments,
)
```

```
Map:   0%|          | 0/89560 [00:00<?, ? examples/s]
/usr/local/lib/python3.10/dist-packages/trl/trainer/sft_trainer.py:290: UserWarning: You passed a tokenizer with `padding_side` not equal to `right` to the SFTTrainer. This might lead to some unexpected behaviour due to overflow issues when training a model in half-precision. You might consider adding `tokenizer.padding_side = 'right'` to your code.
  warnings.warn(
```

We will also pre-process the model by upcasting the layer norms in float 32 for more stable training

The code initializes an instance of the SFTTrainer for training the model using the provided dataset and PEFT configuration.

8. Model Pre-processing

Now let's train the model! Simply call `trainer.train()`

In [9]:
```
import torch
torch.cuda.empty_cache()
```

In [10]:
```
trainer.train()
```

```
wandb: Logging into wandb.ai. (Learn how to deploy a W&B server locally: https://wandb.me/wandb-server)
wandb: You can find your API key in your browser here: https://wandb.ai/authorize
wandb: Paste an API key from your profile and hit enter, or press ctrl+c to quit:
 ..........
wandb: Appending key for api.wandb.ai to your netrc file: /root/.netrc
```

Tracking run with wandb version 0.16.2

Run data is saved locally in `/content/wandb/run-20240120_064256-zdlnu6c2`

Syncing run **cool-jazz-4** to Weights & Biases (docs)

View project at https://wandb.ai/flash26/huggingface

View run at https://wandb.ai/flash26/huggingface/runs/zdlnu6c2

```
You're using a LlamaTokenizerFast tokenizer. Please note that with a fast tokenizer, using the `__call__` method is faster than using a method to encode the text followed by a call to the `pad` method to get a padded encoding.
```

[100/100 09:13, Epoch 0/1]

Step	Training Loss
10	4.602100
20	4.530400
30	4.477300
40	4.473500
50	4.451200
60	4.469500
70	4.457700

This loop upcasts the layer norms in the model to float32 for more stable training.

9. Model Training

Out[10]:
```
TrainOutput(global_step=100, training_loss=4.477074356079101, metrics={'train_runtime': 584.3136, 'train_samples_per_second': 1.369, 'train_steps_per_second': 0.171, 'total_flos': 6483083746787328.0, 'train_loss': 4.477074356079101, 'epoch': 0.01})
```

In [11]:
```
model_to_save = trainer.model.module if hasattr(trainer.model, 'module') else trainer.model  # Take care of distributed/par
model_to_save.save_pretrained("outputs")
```

In [12]:
```
lora_config = LoraConfig.from_pretrained('outputs')
model = get_peft_model(model, lora_config)
```

The code trains the model using the configured trainer.
This code saves the trained model to a directory named "outputs" and loads the PEFT model configuration from the saved directory.

10. Sequence Generation

```
In [16]: sequence = "MSTAGKVIKCKAAVLWGKAAKLADNLA"
         device = "cuda:0"

         inputs = tokenizer(sequence, return_tensors="pt").to(device)
         outputs = model.generate(**inputs, max_new_tokens=50)
         print(tokenizer.decode(outputs[0], skip_special_tokens=True))

MSTAGKVIKCKAAVLWGKAAKLADNLAKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKL
```

This code generates a sequence using the trained model given an input sequence

11. Sequence Similarity Calculation

```
In [18]: from difflib import SequenceMatcher

         def compute_sequence_identity(seq1, seq2):
             matcher = SequenceMatcher(None, seq1, seq2)
             match = sum(match_size for opcode, start1, end1, start2, end2 in matcher.get_opcodes() if opcode == 'equal' for match_s
             total_length = len(seq1)
             identity = match / total_length
             return identity

         # Example usage
         reference_sequence = "MSTAGKVIKCKAAVLWGKAAKLADNLA"
         generated_sequence = "MSTAGKVIKCKAAVLWGKAAKLADNLAKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKLKL"

         identity = compute_sequence_identity(reference_sequence, generated_sequence)
         print(f"Sequence Identity: {identity}")

Sequence Identity: 1.0
```

This code defines a function (compute_sequence_identity) to calculate the sequence identity between two protein sequences. It then applies this function to compare a reference and a generated sequence.

This code appears to be a training script for a protein language model using PEFT with transformer-based architecture, 4-bit quantization, and additional features for stable training. The training is carried out using the SFTTrainer from the TRL library, and the trained model is saved for future use. The script also includes sequence generation and identity calculation for evaluation purposes.