Assignment 4: Comparative_Training_Methods— Report

Dataset: wikitext https://github.com/charan-976/Generative-AI

1. Introduction:

This assignment explores how three core training methods for generative AI models behave in practice:

- Unsupervised pre-training – training a small language model to predict the next character on a general text corpus.
- Supervised Fine-Tuning (SFT) – starting from the pre-trained weights and training on a small, labeled instruction–answer dataset.
- Reinforcement Learning (RL-lite) – treating the SFT model as a policy, updating it with a REINFORCE objective to favor outputs that satisfy a hand-crafted reward function.

In my implementation, all experiments use a **tiny character-level Transformer ("TinyGPT")** so that everything can be trained quickly on CPU or a low-VRAM GPU.

2. **Methodology:**

Train/validation split:

The pre-training corpus is split into 90% training and 10% validation, controlled by train_fraction = 0.9. Each dataset is turned into an LMDataset, which slides a window of length context_length = 64 over the character IDs. For each window, the input is characters t[i : i+64] and the target is the next characters t[i+1 : i+65].

DataLoaders are built with batch_size = 32 and drop_last=True so that only full batches are used.SFT dataset. For SFT, I constructed a small list of instruction–answer pairs (SFT_PAIRS)

Instruction: <instruction>

Answer: <answer>

and concatenated into sft_train_text (for training) and sft_val_text (for validation). The same CharTokenizer is re-used, so the model stays character-level across all phases.

3. Model Architectures: The generative model is a compact Transformer nicknamed TinyGPT. It is implemented in PyTorch using nn. TransformerEncoder with a causal mask

Key architectural choices:

- Embedding dimension (d_model): 128
- Number of layers (n_layers): 2 Transformer encoder blocks
- Number of heads (n_heads): 4 attention head
- Context length: 64 characters
- Dropout: 0.1

4. Components:

1. **Token embeddings:** an nn.Embedding layer maps each character ID to a 128-dimensional vector.
2. **Positional embeddings:** another embedding layer encodes positions 0–63 so the model can distinguish character order.
3. **Transformer encoder:** each block contains multi-head causal self-attention and a feed-forward MLP with GELU activation and residual connections.
4. **LayerNorm + output head:** outputs from the Transformer are normalized and passed through a final linear layer to produce logits over the vocabulary at every time step.

Overall size is small (on the order of ~1M parameters), which keeps training fast but limits expressiveness which is important when interpreting the later results.

5. Training Configuration:

All three stages use **AdamW** as the optimizer with gradient clipping (grad_clip = 1.0) and mild weight decay:

- **Batch size**: 32
- **Pre-training learning rate**: lr_pretrain = 3e-4
- **SFT learning rate**: lr_sft = 1e-4
- **RL learning rate**: lr_rl = 5e-5
- **Maximum steps**:
    - max_steps_pretrain = 2000
    - max_steps_sft = 500
    - max_steps_rl = 150

Training and validation loss are logged periodically. For evaluation, the notebook computes **cross-entropy loss** and then **perplexity = exp(loss)** as a standard language-modeling metric.

For text generation, the model uses **temperature sampling** with temperature = 0.6 and top_k = 10, which reduces randomness compared to pure sampling but still allows variety.

Reinforcement Learning:

After SFT, the model is treated as a policy. The RL goal is to encourage outputs that match a "good coffee tagline" style for a fixed prompt:

Instruction: Give a creative tagline for coffee before practice.

Answer:

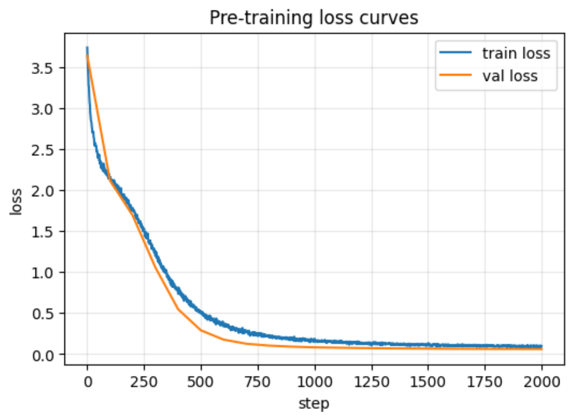
A simple reward function simple_reward(text) is defined over the generated answer:

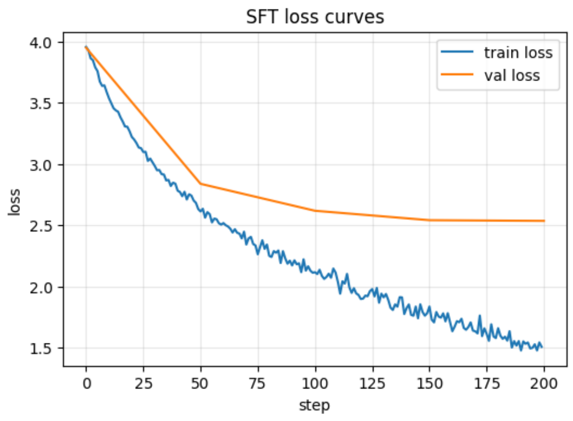
- +1.0 if the word "coffee" appears (case-insensitive).
- +0.5 if the text contains an exclamation mark.
- +0.5 if the answer lengthis between 8 and 20, to avoid very short long lines.

6. Results:

In the first ~200 steps, loss drops quickly to around 2.0, showing that the model is rapidly picking up basic character patterns and frequent n-grams.Between 200 and ~700 steps, both curves continue to fall more slowly down to below 0.5, indicating that the model is fitting more detailed structure.From roughly 700 to 2000 steps, loss continues to decrease and levels off very close to 0.1, which corresponds to very low perplexity on this small dataset.

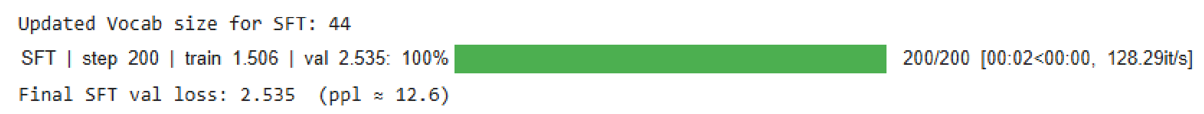
Pre-training is stable and convergent: there are no spikes or divergencethere is **little evidence of strong overfitting.** your validation split is from the same and the model capacity is small.





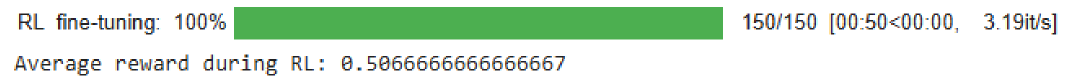
At step 0, both train and validation loss start around 4.0, reflecting that the pre-trained model has never seen the "Instruction: … Answer: …" format or the new coffee-related outputs.The training loss drops quickly from 4.0 to about 2.6 within the first ~50 steps, and then continues decreasing steadily to around 1.5 by 200 steps.The validation loss initially follows the training curve down to roughly 2.8 by step 50, but then flattens around 2.5–2.6.

The rapid drop in both curves at the beginning means the model immediately benefits from pre-training: it only needs a few dozen steps to pick up the new instruction–answer pattern and the specific SFT pairs.



```
Updated Vocab size for SFT: 44
SFT | step 200 | train 1.506 | val 2.535: 100%                    200/200 [00:02<00:00, 128.29it/s]
Final SFT val loss: 2.535  (ppl ≈ 12.6)
```

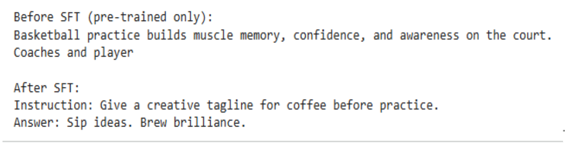
Before SFT, the pre-trained model only talks in general sports sentences like "Basketball practice builds muscle memory…", and it does not follow the "Instruction / Answer" format or understand the coffee tagline task. After SFT, when we ask "Give a creative tagline for coffee before practice," the model replies with a short, tagline-style answer: "Sip ideas. Brew brilliance." – this shows it has learned the instruction format and the idea of a short motivational line. After RL fine-tuning, using a reward that prefers the word "coffee" and a certain length, the answer becomes even more aligned with the desired style: "Cool down, sip up – coffee that rewards every rep." Overall, you can see a clear progression from generic sports text (pre-train), to formatted and somewhat creative taglines (SFT), to a final version that directly matches the coffee-tagline style we rewarded during RL.



```
RL fine-tuning: 100%                    150/150 [00:50<00:00,  3.19it/s]
Average reward during RL: 0.5066666666666667
```

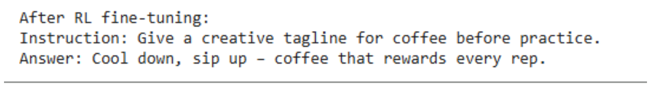
7. Conclusion:

supervised fine-tuning on a handful of instruction–answer pairs taught the model a new format ("Instruction: … Answer: …") and a new task.



```
Before SFT (pre-trained only):
Basketball practice builds muscle memory, confidence, and awareness on the court.
Coaches and player

After SFT:
Instruction: Give a creative tagline for coffee before practice.
Answer: Sip ideas. Brew brilliance.
```



```
After RL fine-tuning:
Instruction: Give a creative tagline for coffee before practice.
Answer: Cool down, sip up - coffee that rewards every rep.
```

The experiments also highlight several practical lessons. First, pre-training is essential: without it, such a small SFT dataset would not be enough to learn both language and the task. Second, SFT is powerful but easy to overfit when the labeled data is tiny; the SFT loss curves show training loss continuing to drop while validation loss flattens, so monitoring validation metrics and using early stopping is important. Third, RL is useful when we care about specific behaviors that cannot be written as simple labels (for example, "must mention coffee and feel like a tagline"), but it is more unstable and depends heavily on the reward design. Finally, the project makes it clear that, in real applications, we would need larger models, more diverse data, and better RL algorithms, but the same overall pipeline and trade-offs still apply: use pre-training for general knowledge, SFT to adapt to tasks, and RL only when we need fine-grained control over model behavior.