

# Technical Quiz

*Spring 25 CSS 5120 Comp Linguistics, Zhanzhan Zhao*

This quiz worths 20 points of your final grade. I offer two quiz options and you can choose either one to complete.

The submission deadline is April 20<sup>th</sup>, the end of the day.

Quiz Option I: **Fine-Tuning a Pre-trained Generative Language Model**

Quiz Option II: **Simulating a Social Phenomenon with 100+ LLM Agents**

# Quiz Option I: Fine-Tuning a Pre-trained Generative Language Model

*Spring 25 CSS 5120 Comp Linguistics, Zhanzhan Zhao*

## Overview

Fine-tuning a pre-trained **Large Language Model (LLM)** on a custom dataset allows the model to adapt to specific styles or tasks. In this assignment, you will **fine-tune a generative LLM** (such as GPT-2, LLaMA, Falcon, Mistral, etc.) on a novel text dataset of your choice. We will use *parameter-efficient fine-tuning* techniques to work within Colab's resource limits – specifically **LoRA adapters** and **8-bit model loading**. Traditional fine-tuning updates all model weights, which is impractical for very large models, but **parameter-efficient methods train only a tiny fraction of the parameters** (e.g. using **Low-Rank Adaptation, LoRA**<sup>1</sup>), drastically reducing memory and compute requirements. By using LoRA together with 8-bit quantization (via the **BitsAndBytes** library), we can fine-tune models that would normally be too large for Colab, optimizing GPU memory usage<sup>2</sup>.

You will go through the full pipeline: **dataset selection, data preprocessing, model and training setup, training with loss monitoring, and evaluation** of your fine-tuned model. The assignment is beginner-friendly – we'll use high-level libraries like Hugging Face 😊 Transformers for model training, the 😊 PEFT library for LoRA, and tools like `bitsandbytes`

---

<sup>1</sup>

<https://huggingface.co/docs/peft/en/quicktour#:~:text=PEFT%20offers%20parameter,the%20number%20of%20trainable%20parameters>

<sup>2</sup> <https://medium.com/@shitalnandre108/fine-tuning-llama-2-large-language-model-with-custom-datasets-using-google-colab-a-comprehensive-a9d68faf3bc9#:~:text=In%20summary%2C%20fine.for%20enhanced%20performance%20and%20efficiency>

for memory optimization. By the end, you'll understand how to adapt an LLM to a new dataset and assess its performance.

**Important:** Because we are using Google Colab, you can choose a *small-scale* model (around **70M parameters or a similarly small variant**) for fine-tuning for free Colab GPU, and larger models (billions of parameters) for Colab pro or pro+ membership. For example, **DistilGPT-2 has ~82 million parameters**<sup>3</sup> (a distilled small version of GPT-2), and **GPT-2 “small” has ~124M parameters**<sup>4</sup> – these are typical choices for free Colab accounts. In contrast, a LLaMA or Falcon model with 7B+ parameters would require paid versions of Colab accounts. **You are required to use a parameter-efficient method (like LoRA/adapters)** rather than full fine-tuning, to ensure training fits in memory and runs in a reasonable time frame.

The quiz is structured into **five parts**, each focusing on a step in the fine-tuning workflow. Follow the instructions in each part, and make sure to document your steps and results in a clear manner. Short explanations for each step, code snippets, and outputs (graphs, metrics, sample texts) should be included in your Colab notebook. We've also provided an **expected output** section for each part, so you know what results to aim for. Good luck, and have fun fine-tuning your own language model!

## Before You Begin: Required Background Materials

Before starting the assignment, please review the following essential resources. These materials will help you understand the fine-tuning workflow, how LoRA and other parameter-efficient techniques work, and how to implement them effectively in Colab:

### Fine-Tuning Process (Conceptual Overview)

- **Course Module:** [Fine-tuning Large Language Models – Deeplearning.ai](#)

A brief but essential video walkthrough of the fine-tuning pipeline and its components.

---

<sup>3</sup> <https://huggingface.co/distilbert/distilgpt2#:~:text=DistilGPT2%2C%20which%20has%2082%20million>

<sup>4</sup> <https://en.wikipedia.org/wiki/GPT-2#:~:text=>

## Parameter-Efficient Fine-Tuning (PEFT)

- **Video Introduction to PEFT (LoRA, Adapters):**

[YouTube: Efficient Fine-Tuning Methods Explained](#)

- **Hands-on PEFT Example (HuggingFace):**


[Colab: Finetune OPT with bitsandbytes + PEFT](#)

A complete example notebook showing how to load a model with 8-bit precision and apply LoRA adapters.

- **Student reference example notebook (adapted):**

[Colab: QLoRA-style Fine-tuning Demo](#)

Use this as inspiration — **do not copy directly**, but refer to it to understand how PEFT and int8 models are configured.

 **Tip:** Reading these resources carefully before you start will save you a lot of debugging time and help you set up your fine-tuning pipeline correctly.

---

## Part 1: Dataset Selection

In this part, you will **choose and prepare a novel dataset** for fine-tuning your language model. The dataset should consist of text samples (sentences, paragraphs, dialogues, etc.) that the model will learn to generate. You are encouraged to pick a domain or style you find interesting – the more unique, the better! Possible ideas include: movie dialogues, tech support chats, fantasy novel excerpts, news articles on a specific topic, product reviews, song lyrics, etc. **At least 1000 samples** of text are required to give the model enough to learn from.

### Instructions:

1. **Select a text dataset ( $\geq 1000$  samples):** Identify a source of data that is *new and not just a pre-packaged example*. You can curate your own dataset (e.g., scrape public domain texts or compile data from a source you have access to) or use an existing dataset from repositories like Hugging Face Datasets or Kaggle. If you use an existing dataset, make

sure to **cite the source and describe what it contains**. The key is that you should not simply reuse a trivial toy dataset from an example – demonstrate initiative in picking something novel or customizing it to your needs.

2. **Load the dataset in Colab:** Download or upload your data and load it into your notebook. You can use Python data handling (e.g., reading a text file or CSV), or the `datasets` library (`datasets.load_dataset`) if your data is available via Hugging Face. Confirm that you have at least 1000 text entries. If your dataset is large, you may sample a subset, but ensure it's still sufficiently sized.
3. **Provide a brief description:** In a markdown cell, describe your chosen dataset. Mention the domain/genre, how it was collected, and why you chose it. For example, *“This dataset contains 2,000 short news headlines from 2020 about technology. I chose it to fine-tune the model to produce tech news-style text.”*
4. **Inspect the data:** Show a **preview of the dataset**. Print out 1-2 example samples (a couple of lines of text) to give an idea of what the model will be learning. Also, print basic stats like the total number of samples and (if easily obtainable) the average length of samples (in characters or words). This helps verify the data looks correct.

### Expected Outputs (Part 1):

- A clear description of the dataset in the notebook (text explaining what the data is and its source).
- Code that loads the dataset and confirms the number of samples (print out the count, e.g., “Dataset contains 1,257 text samples.”).
- A preview of the data: for example, printing the first few lines or examples. Ensure the example text is visible so we can verify the content and format.
- (Optional) Any simple statistics or observations (e.g., average length of a sample, any cleaning performed at this stage).

## Part 2: Data Preprocessing

Now that you have your raw dataset, it's time to **preprocess the data for training**.

Preprocessing typically includes cleaning the text and converting it into a format suitable for the

model (tokenization). In language model fine-tuning, tokenization will transform text strings into sequences of token IDs that the model can understand. We will also split the dataset into training and validation (and optionally test) sets to be able to evaluate performance on unseen data.

### Instructions:

1. **Clean and normalize text (if needed):** Examine if your dataset text needs cleaning. This could include lowercasing (if case shouldn't matter), removing unwanted characters or HTML tags, fixing encoding issues, etc. If the text is already clean, this step may be minimal. *Document any cleaning steps you perform.* For example, you might use regex or Python string operations to remove citations, or strip extra whitespace.
2. **Split the dataset into train/validation/test:** It's important to evaluate the model on data it wasn't trained on. Split your dataset into a **training set** and a **validation set** at minimum. For example, you could use an 80/20 split (80% train, 20% val) or 90/10 split if data is limited. If you want, you can set aside a small test set as well for final evaluation (e.g., 10% validation, 10% test, 80% train). You can do the split randomly (ensure it's reproducible by setting a seed) or use any dataset utility (HuggingFace's `dataset.train_test_split` if using their `datasets` object). After splitting, print the number of samples in each subset to verify (e.g., "900 in train, 100 in val").
3. **Initialize the tokenizer:** Load the tokenizer associated with your chosen model. For example, if using a GPT-2 model from HuggingFace, you can do:

```
(python)
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained(<model_name>)
```

This will download the tokenizer vocab and config. Many tokenizers (especially for GPT-style models) might not have a default pad token. For text generation tasks, we often set the EOS (end-of-sequence) token as the pad token. For example, if using GPT-2:

```
(python)
tokenizer.pad_token = tokenizer.eos_token
```

This avoids issues during batching (since GPT-2's model head can ignore pad tokens if set this way).

4. **Tokenize the dataset:** Convert your text data into token IDs for the model. You can tokenize each sample in the train and validation sets. If using HuggingFace Datasets, you can use `dataset.map` with a `tokenize` function. For example:

```
(python)
def tokenize_function(examples):
    return tokenizer(examples["text"], truncation=True,
padding="max_length", max_length=256)
tokenized_datasets = raw_datasets.map(tokenize_function, batched=True)
```

This will add tokenized representations to the dataset. If you're not using HF Datasets, you can loop over your list of texts and use `tokenizer()` on each, then create lists/arrays of input IDs. Choose an appropriate `max_length` for truncation/padding – e.g., 128 or 256 tokens, depending on how long your samples are and memory constraints. (Shorter `max_length` will train faster but might truncate longer samples; use your judgment based on the data).

Make sure to tokenize both the training and validation sets (you can tokenize the test set too, if you have one, for later evaluation).

5. **Prepare for causal language modeling:** Since we are fine-tuning a **causal LLM** (one that predicts the next token in a sequence), the typical data format consists of **input token sequences where the model learns to predict the next token**. In the simplest case, the model is trained to predict every next token from a sequence (e.g., language modeling). In such setups, we **do not need to specify separate labels** — the model just tries to predict the next token in the input, and libraries like HuggingFace's `Trainer` handle this automatically (by shifting `input_ids` to create `labels`).

However, in many real-world fine-tuning tasks — especially for **prompt-response**, **instruction-following**, or **question-answering (Q-A)** setups — you may want to structure your dataset as a **prompt + response** pair. In this case: The input is the concatenation of prompt and response. The label (what you want the model to learn to generate) is usually **only the response**, with the **prompt portion masked out** (e.g., using `-100` in the label array to ignore those tokens during loss computation). This ensures the model learns to **generate the response given the prompt**, rather than trying to reproduce the prompt too. For example:

```
(Python)
example = {"prompt": "Q: What is the capital of
France?\nA:", "response": " Paris"}

input_text = example["prompt"] + example["response"]

input_ids=tokenizer(input_text,return_tensors="pt").input_ids.squeeze()

labels = input_ids.clone()
# Mask the prompt tokens
prompt_len = len(tokenizer(example["prompt"]).input_ids)

labels[:prompt_len] = -100
```

If you’re using your own training loop, you must manually shift `input_ids` and prepare the `labels` like above. If you use the `HuggingFace Trainer`, you still need to format your dataset so that it includes the correct `labels` if your use case is Q-A or prompt-response generation. This is crucial when you only want the model to learn from a specific part of the input (e.g., the answer).

6. **Double-check tokenization output:** It’s a good idea to print a tokenized sample to verify the process. Take one sample text and show: the original text, the list of token IDs, and maybe a decoded version of the token IDs back to text (to ensure nothing funky happened). This sanity check helps confirm that your preprocessing is correct. Also confirm the shape of your dataset now (e.g., number of examples and length of each) and ensure it matches expectations.

## Expected Outputs (Part 2):

- Confirmation of the dataset split sizes (e.g., `printout`: “Training set: 900 samples, Validation set: 100 samples”).
- Some example code output showing the tokenizer is loaded (for instance, printing `tokenizer.vocab_size` or a special token).
- A printed example of a tokenized text: for example, output a small dictionary or list like `{'input_ids': [..., ..., ...], 'attention_mask': [..., ..., ...]}` for one sample, along with the original text. Alternatively, print the first 20 token IDs of a sample and their decoded tokens.
- A brief note of any preprocessing decisions (e.g., “All text lowercased,” “URLs removed,” or “No special cleaning was needed as the text was already clean.”). This can be in markdown or code comments.



- The dataset ready for training: e.g., you might show that you have a `train_dataset` and `val_dataset` object or lists of token tensors ready to feed into the model.

## Part 3: Fine-Tuning Setup (Model & Training Configuration)

In this part, you will set up the model and training configuration for fine-tuning. This includes selecting a pre-trained model checkpoint to start from, loading it with a parameter-efficient setup (LoRA in 8-bit mode), and preparing the training settings (hyperparameters, optimizer, etc.). We will use the 😊 *Transformers* library to load the model and the 😊 *PEFT* library to apply LoRA. We'll also configure training details like number of epochs, batch size, and learning rate.

### Instructions:

1. **Choose a pre-trained model:** Select a **base LLM** (Large Language Model) that you will fine-tune. Your choice should balance **model size**, **training time**, and **Colab resource limits**. There are two main paths depending on your access to **Google Colab**:

🟢 **Option A: Using Free Colab (or Colab with limited memory)** Free Colab sessions typically provide **one GPU (like Tesla T4 or K80)** with **~12-16GB of RAM**, so you should choose a **smaller-scale model** (under ~125M parameters) and use **8-bit loading and LoRA** for memory-efficient training. Suitable models include:

- **GPT-2 Small** (`gpt2`) – ~124M parameters
- **DistilGPT-2** (`distilgpt2`) – ~82M parameters
- **OPT-125M** (`facebook/opt-125m`) – ~125M parameters
- **GPT-Neo 125M** (`EleutherAI/gpt-neo-125M`) – ~125M parameters
- **Pythia-70M** (`EleutherAI/pythia-70m`) – ~70M parameters (very lightweight)

**Tip:** Use `load_in_8bit=True` and `device_map='auto'` when loading the model for optimal memory efficiency.

🟡 **Option B: Using Colab Pro or Colab Pro+ (paid plans)** If you're using **Colab Pro** or **Pro+**, you may access faster GPUs (e.g., Tesla P100, V100, A100) and longer runtimes. This allows you to experiment with **medium-scale models** that require more VRAM, especially when using parameter-efficient methods like **LoRA** or **QLoRA** with quantization. Examples:

- **LLaMA 2 7B (with QLoRA)** – e.g., via `TheBloke/Llama-2-7B-GPTQ`
- **Mistral 7B (quantized)** – e.g., `TheBloke/Mistral-7B-Instruct-v0.1-GPTQ`
- **Falcon-1B** (`tiiuae/falcon-rw-1b`) – manageable for Pro-level setups
- **Pythia-410M** (`EleutherAI/pythia-410m`) – a larger but feasible model

⚠️ Be cautious: full fine-tuning on 7B+ models is not feasible without 4-bit quantization and careful memory handling. Always use quantized versions and LoRA for training.

### 🧠 General Tips

- Always check model size on Hugging Face – avoid full fine-tuning for models over 1B parameters if you are not a Colab Pro member.
- Prefer **instruction-tuned models** (e.g., `falcon-rw-1b`, `dolly-v2-3b`, `mpt-7b-instruct`) if you're working with tasks like summarization, Q&A, or chatbots.
- Look for models with tags like `GPTQ`, `AWQ`, or `int4` for 4-bit versions that are easier to fit in Colab.

2. **Install necessary libraries:** At the top of your Colab, ensure you have installed all required packages. For example, in a code cell:

```
(bash)
!pip install transformers datasets peft bitsandbytes
```

This installs HuggingFace Transformers, Datasets, PEFT (for LoRA), and BitsAndBytes (for 8-bit quantization). You should also import PyTorch (`import torch`). Verify that you have a GPU runtime in Colab by running `!nvidia-smi` – the presence of a GPU (like Tesla T4, etc.) is needed for training to be feasible.

3. **Load the pre-trained model and tokenizer:** Using the model name you chose, load the model with 8-bit precision to save memory. For example:

(python)

```
from transformers import AutoModelForCausalLM, AutoTokenizer
model_name = "distilgpt2" # or "gpt2", "EleutherAI/pythia-70m", etc.
tokenizer = AutoTokenizer.from_pretrained(model_name)
tokenizer.pad_token = tokenizer.eos_token # set pad token if needed
(for GPT-2 family)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    load_in_8bit=True,      # use 8-bit quantization for model weights
    device_map="auto"      # let HF automatically place model on GPU
)
```

The `load_in_8bit=True` flag uses the `BitsAndBytes` library to load model weights in 8-bit quantized form, greatly reducing memory usage (with minimal impact on accuracy). The `device_map="auto"` will put the model on GPU if available. If for some reason 8-bit loading isn't supported by your model choice, you can omit it (but be mindful of memory). If not using 8-bit, you might consider `model.half()` to use FP16 precision.

4. **Prepare model for LoRA fine-tuning:** When using PEFT LoRA, we typically **freeze the base model weights** and add small trainable adapter weights. HuggingFace's PEFT provides a utility to prepare the model for int8 training if applicable:

(python)

```
from peft import LoraConfig, get_peft_model,
prepare_model_for_int8_training
model = prepare_model_for_int8_training(model)
```

The above line (optional but recommended for int8) will tweak the model (e.g., adjust layer norms) to be compatible with training in 8-bit. Next, define a LoRA configuration. For a *causal language modeling* task:

```
(python)
peft_config = LoraConfig(
    task_type="CAUSAL_LM",      # type of task
    inference_mode=False,      # we are training, not just inferencing
    r=8,                        # LoRA rank (adjustable hyperparam, 4-16
are typical)
    lora_alpha=16,              # LoRA scaling factor
    lora_dropout=0.1            # dropout for LoRA layers
)
model = get_peft_model(model, peft_config)
```

This wraps your model with LoRA adapters. The `r`, `alpha`, and `dropout` are hyperparameters you can experiment with; the given values are common defaults (rank-8 adapters, 0.1 dropout). Once wrapped, **only the LoRA adapter parameters will be trainable** while the original model’s weights stay frozen. You can verify this by checking the number of trainable parameters:

```
(python)

model.print_trainable_parameters()
```

This should output something like: “*trainable params: X || all params: Y || trainable%: Z*”. Ideally, X will be much smaller than Y. For example, even on a model with over 1 billion parameters, LoRA might train fewer than 1% of them<sup>5</sup>. (E.g., only ~2.3 million out of 1.2 billion parameters were trainable in one LoRA setup, which is ~0.19%<sup>6</sup>). In your case, since the model is smaller, the absolute number of trainable params will be on the order of a few hundred thousand to a few million, depending on the LoRA rank. **Include the output of `print_trainable_parameters()` in your notebook** to confirm the reduction.

5. **Set up training configuration:** Decide on your training hyperparameters. We need to specify:

---

<sup>5</sup> <https://huggingface.co/docs/peft/en/quicktour#>

<sup>6</sup> <https://huggingface.co/docs/peft/en/quicktour#>

- **Number of epochs:** How many passes through the training data. For a dataset of this size (~1000+ samples), something like 3-5 epochs is a reasonable starting point. You can adjust based on observed training curves (if underfitting, increase epochs; if overfitting, maybe fewer).
- **Batch size:** This is **limited by available GPU memory**. With **LoRA** and a **small model** ( $\leq 125\text{M}$  parameters), you might try a batch size of **8 or 16** if your sequence length is short (e.g., 128–256 tokens). If you get **out-of-memory errors**, reduce the batch size — even **batch size 4** is acceptable. You can also use **gradient accumulation** to simulate a larger batch size while keeping memory usage low. For example, **accumulating gradients over 2 steps of batch size 4** gives you the effect of batch size 8. Set this via the `gradient_accumulation_steps` argument in the `TrainingArguments`.

💡 **For Colab Pro/Pro+ users with more memory:**

If you're using **larger models** (e.g., 410M–1B parameters) or longer sequences, start with a batch size of **2 or 4**, and use gradient accumulation (e.g., 4 steps of batch size 2 = effective batch 8). Also try reducing `max_length` if memory becomes an issue.

- **Learning rate:** A common learning rate for fine-tuning full models is in the range **1e-5 to 5e-5**. However, when using **parameter-efficient tuning** like **LoRA**, we are only training a **small number of adapter weights**, so you can often use a slightly **higher learning rate**, like **2e-5 or 3e-5**.

LoRA papers on large models sometimes use rates up to **3e-4**, but for our use case (smaller models or fewer resources), **2e-5 to 5e-5** is a safer default. Monitor the **training loss** carefully to ensure the model is actually improving; if the loss fluctuates too much or increases, reduce the learning rate.

- **Other settings:** You can set `logging_steps` (how frequently to log training loss), and `evaluation_strategy` (e.g., "epoch" to evaluate on the validation set each epoch). Also decide if you want to save checkpoints — you can save the final

model at least. Use `save_steps` or `save_total_limit` to manage checkpoints if needed.

If using the HuggingFace `Trainer`, you will create a `TrainingArguments` object. For example:

(python)

```
from transformers import TrainingArguments, Trainer
training_args = TrainingArguments(
    output_dir="model_outputs",
    num_train_epochs=4,
    per_device_train_batch_size=8,
    gradient_accumulation_steps=1,  # adjust if you need to simulate
larger batch
    learning_rate=2e-5,
    logging_steps=50,
    evaluation_strategy="epoch",
    save_strategy="epoch",
    save_total_limit=1,
    fp16=True,      # Enable mixed precision if using a GPU with tensor
cores
    report_to="none"  # (or "tensorboard" if you use TB for logging)
)
```

The above is just an example – adjust the values to your needs. `fp16=True` will use mixed precision (half-precision) training for speed if supported. We disable reporting to any online service; you can use `"tensorboard"` if you want to visualize in TensorBoard (optional).

6. **Initialize the Trainer (or training loop):** If using `Trainer`, create it by passing the model, datasets, tokenizer, and training args (and optionally a `compute_metrics` function if you plan to calculate metrics during eval):

(python)

```
trainer = Trainer(
```

```

model=model,
args=training_args,
train_dataset=tokenized_train_dataset,
eval_dataset=tokenized_val_dataset,
tokenizer=tokenizer
# (you can add compute_metrics function here if evaluating
perplexity or others on the fly)
)

```

If you prefer to write your own training loop without `Trainer`, you would need to set up an optimizer (and perhaps a scheduler), iterate over epochs and batches, and do `model.forward` and `loss.backward()`, etc. For this assignment, using `Trainer` is perfectly fine and recommended for simplicity.

At this point, you have configured everything needed for training. Before proceeding, double-check that:

- The model is on GPU (`model.device` should show `cuda`).
- The number of trainable parameters is reasonable (from the print earlier).
- The training and validation datasets are ready (and possibly small sanity-check: try a forward pass with one batch to see if it runs without error).

### Expected Outputs (Part 3):

- The model selection and loading code, with output indicating the model was loaded (e.g., logs from HuggingFace model download).
- Output from `model.print_trainable_parameters()` showing how many parameters will be updated (to confirm the use of LoRA/adapters). For full credit, this should show a **very small percentage of the total parameters** are trainable (this indicates you correctly froze the rest). Include this output in the notebook.
- A summary of your training hyperparameters (you can simply show the `TrainingArguments` by printing `training_args` which will display the configuration).
- Evidence that the model and tokenizer are ready: for instance, you might show that `tokenizer.eos_token` and `tokenizer.pad_token` are set, or print a sample of

`tokenizer.decode(model.generate(...))` for a quick test (optional). However, if you do test generation now, remember the model is not yet fine-tuned on your data – it would just produce generic text – this is optional for a baseline check.

- In short, by end of Part 3, your notebook should clearly indicate **which model** you're using and that it's prepared for training with LoRA in 8-bit mode, and how training will be carried out (which Trainer or loop config you set).

## Part 4: Model Training and Loss Monitoring

Time to **fine-tune the model** on your dataset! In this part, you will run the training process and monitor the model's performance (primarily via the training loss, and validation loss if you have a val set). Training might take some time depending on your model and dataset size, but with a small model and limited data it should be manageable on Colab (likely a few minutes to an hour). Make sure to use the GPU by moving data to CUDA if writing a custom loop (the Trainer will handle this if configured with `device_map` or if `cuda` is available).

### Instructions:

1. **Begin training:** If using the HuggingFace `Trainer`, simply call:

```
(python)
trainer.train()
```

This will start the fine-tuning process. The Trainer will output logs every few steps as specified (e.g., every 50 steps) and at the end of each epoch (especially since we set `evaluation_strategy="epoch"`, it will also evaluate on the validation set each epoch and report `eval_loss`). Watch the Colab output for the training progress. If you implemented a custom loop, print the loss periodically yourself (for example, every 50 iterations print the average loss so far).

2. **Monitor the loss:** Pay attention to the **training loss** as it decreases. Typically, it should start higher and go down over epochs. Also observe the **evaluation/validation loss** each epoch (if you set that up). Ideally, the validation loss should also decrease and be close to



training loss; if it diverges or increases while training loss decreases, that indicates overfitting. Given our dataset size and model, a bit of overfitting is possible if too many epochs are used – you can adjust accordingly.

- You may also use TensorBoard for monitoring if you logged metrics. (This is optional: you'd load the extension with `%load_ext tensorboard` and `%tensorboard --logdir <log_dir>` if you set it up.)
- Another simple approach: **plot the loss curve** after training. You can extract the logged history from the Trainer. For example:

```
(python)
import pandas as pd
df = pd.DataFrame(trainer.state.log_history)
print(df.columns) # see what metrics are logged
```

The `log_history` will contain entries like `{'step': ..., 'loss': ..., 'epoch': ...}` and also `{'eval_loss': ...}` at certain points. You can filter and plot these. Alternatively, keep track of losses in a list if using custom loop. Plot training loss vs. training steps (or epochs), and validation loss vs. epochs.

- **Include a training loss graph** in your notebook output. A simple Matplotlib line plot of loss over time (steps or epochs) is sufficient.
3. **Save the fine-tuned model:** Once training is complete, save your model (and the LoRA adapter weights). With the `Trainer`, you might have `trainer.save_model()` or if not, you can directly call:

```
(python)
model.save_pretrained("fine-tuned-model")
tokenizer.save_pretrained("fine-tuned-model")
```

This will save the model weights and tokenizer files to the `fine-tuned-model` directory in your Colab environment. (Since we used LoRA, this should save the full model with the LoRA applied, or possibly it saves just the adapter – ensure that you can reload it later. You might test reloading it after saving, optional).

4. **(Optional):** Record training time or any notable observations during training (like if it was unstable at first, etc.). This isn't required, but it can be useful to note if you had to restart or change any settings.
5. **Troubleshooting:** If you run into issues such as out-of-memory errors:
  - Reduce batch size or sequence length.
  - Ensure that gradient checkpointing is off (unless you explicitly use it) as it can increase memory during training.
  - Verify that only the intended params are trainable (sometimes forgetting `prepare_model_for_int8_training` can cause extra memory use).
  - If the environment crashes, you might have to tweak and try again with smaller settings. Mention any such adjustments you made in your notes.

### Expected Outputs (Part 4):

- **Training logs:** in the notebook, there should be visible logs of training progress. At minimum, the final output after `trainer.train()` showing the end of training (and perhaps final loss values). Ideally, some intermediate logging is shown too (loss values per step or per epoch).
- **Loss curve plot:** a chart (e.g., Matplotlib or similar) showing training loss over time. If you have validation loss, plot that as well (perhaps as two lines: training vs validation loss across epochs). This visual will help demonstrate the model's learning progress.
- **Final training loss and validation loss:** you can report these in text as well. For example, "Final training loss: 1.23; Final validation loss: 1.45". From the validation loss, you can compute **perplexity**: `perplexity = exp(loss)`. If your final eval loss was, say, 1.45, the perplexity is  $\exp(1.45) \approx 4.26$ . It's good to provide this metric on the validation set (or test set later) as a measure of model performance (lower perplexity is better).
- The model saving confirmation (e.g., the presence of `pytorch_model.bin` and adapter config in the output directory, you don't need to list files but mention that model is saved).
- If using `Trainer`, also consider using `trainer.evaluate()` after training to get the final eval loss cleanly and report perplexity.

- Overall, Part 4 deliverables are: evidence that the training ran successfully and the model converged (loss went down). The graph is especially important for visualization.

## Part 5: Evaluation and Results

With the model fine-tuned, it's time to **evaluate** how well it learned and demonstrate its capabilities. We'll perform both quantitative evaluation (computing some metrics) and qualitative evaluation (generating sample texts and doing a human sanity-check). This will show whether the model has adapted to your dataset's style/content.

### Instructions:

1. **Quantitative evaluation – Perplexity:** If you have a **held-out test set** (text data the model never saw during training or validation), load and preprocess it similarly to the others (tokenize it). If you didn't create a separate test set, you can use the validation set for final evaluation. Use the fine-tuned model to compute the average loss on this test set:
  - If you used `Trainer`, simply call `trainer.evaluate(test_dataset)` (or `val_dataset` if no test) which will return a dictionary including `'eval_loss'`. Use that loss.
  - If not using `Trainer`, you can manually do a forward pass on all test samples in evaluation mode (`model.eval()`) and compute the mean loss.
  - Compute **perplexity** from the loss: `perplexity = torch.exp(loss)` (or use `math.exp`). Report the perplexity value. Perplexity indicates how well the model predicts the test data (lower is better, and roughly, a perplexity of  $P$  means the model is as uncertain as having to choose among  $P$  options for the next word on average). For reference, a perplexity around say 5-50 is common for models on various corpora (the lower end being very good).
  - *Note:* Perplexity is a standard metric for language models. If your task was something like translation or a specific output with references, you might compute a metric like **BLEU** or **ROUGE** instead. For this assignment, we focus on perplexity since it's a general metric. (If your dataset/task lends itself to BLEU – e.g., if you fine-tuned on translating a sentence to another language and you have

reference translations – you can calculate BLEU score using `nltk` or `sacrebleu` library. This is optional and only if applicable.)

2. **Qualitative evaluation – Text Generation:** This is the fun part – see what your model can do! Using the fine-tuned model, generate some sample outputs:

- Pick a few **prompts or input contexts** that make sense for your domain. For example, if your dataset is news headlines, you might just provide a few words like “Breaking: New tech” and let the model continue. If the dataset is dialogue, you might start a conversation turn. If it’s Shakespearean text, give it an opening line in that style.
- Use the model’s generate function to produce text. HuggingFace pipeline or the `model.generate()` method can be used. For example:

```
(python)
prompt = "Once upon a time in a distant galaxy"
inputs = tokenizer(prompt, return_tensors='pt').to('cuda')
outputs = model.generate(**inputs, max_length=100,
do_sample=True, top_p=0.9)
generated_text = tokenizer.decode(outputs[0],
skip_special_tokens=True)
print(generated_text)
```

You may experiment with generation parameters (`max_length`, `do_sample` vs `greedy`, `temperature`, `top_k`, `top_p` etc.) to get interesting outputs. For beginners, using `do_sample=True` with a `top_p` (nucleus sampling) or `top_k` can produce more varied text, while greedy might produce the most likely but sometimes dull continuation.

- Generate at least **3 samples** showcasing your model. Ensure the prompt and the generated continuation are clearly presented in your notebook output. The generations should reflect patterns or content from your fine-tuning data. For instance, a model fine-tuned on tech news might produce a plausible tech news headline or brief article; one trained on Shakespeare might output old-style English prose, etc.

- If possible, also include a **comparison** with the original base model's output on the same prompt (optional but insightful). This way you can see the difference after fine-tuning. (To do this, you'd load the original pre-trained model without your fine-tuned weights and generate with it on the same prompt, then compare to the fine-tuned model's output. Only do this if you have time/resources.)
3. **Human assessment – Analyze the outputs:** Read the model's generated texts and give a brief assessment:
- Do they make sense grammatically and contextually?
  - Do they clearly resemble the style or domain of your training data?
  - Any obvious errors or oddities? (e.g., repetition, nonsense, going off-topic)
  - If you have a reference or expectation, how well did the model meet it? (For instance, if it's supposed to answer questions, does it give reasonable answers?)
  - You might also note if the model has any limitations, such as a smaller vocabulary or forgetting info, especially since the model is small and dataset is limited.
  - **Include a short written discussion** (a few sentences) of how the model performed. This can mention the metrics too: e.g., "The model achieved a perplexity of 5.2 on the test set, indicating reasonably confident predictions. The generated examples show that it learned the style of Shakespearean English, although some sentences are grammatically imperfect."
4. **(Optional)** If appropriate, compute an additional metric like **BLEU score** or **ROUGE** on the test set:
- This is only if your task had a defined target text for each input (which a generic language modeling task doesn't really have). For example, if you fine-tuned the model to complete a sentence given the first half (a toy task), you could compare the completion to a reference completion. Or if it's question-answer pairs, you could measure accuracy of answers.
  - BLEU is computed by comparing generated text to a reference text for overlap of n-grams. You can use `nltk.translate.bleu_score` or `sacrebleu` for this if needed. Provide the score if you do this, e.g., "BLEU score of model on test set answers: 45.3".

- If not applicable, it's fine to skip formal metrics beyond perplexity.

### Expected Outputs (Part 5):

- **Perplexity (and/or loss) on the test/validation set:** e.g., “*Test set perplexity = 12.8*”. You should show how you got this (either via `trainer.evaluate` output or a small code calculation). If using `trainer.evaluate`, you might get something like `{'eval_loss': 2.54, 'eval_runtime': ..., ...}` – then you take  $\exp(2.54) \approx 12.7$  as perplexity.
- **Sample generated texts:** At least 2-3 prompts and outputs displayed. Clearly separate the prompt from the generation (you can format it as: **Prompt:** “....” **Model output:** “....”). Make sure the output text is readable (some models might output special tokens – use `skip_special_tokens=True` in `decode` to avoid that).
- **Your analysis/comments:** A brief evaluation of how the model did, referencing the examples. For instance, note if the output is fluent, whether it stays on topic, and any differences from pre-training behavior. If the outputs are good, you can say so; if they have issues, discuss what might improve it (more data, longer training, larger model, etc.).
- (Optional) If you did a comparison with the base model output or calculated BLEU/other metrics, include those results and comment on them.
- Basically, Part 5 should demonstrate the **end result** of your fine-tuning in both number (perplexity) and language (sample generation) form, along with your interpretation.

## Grading Rubric (5 points per section)

Your work will be evaluated based on five key sections, each worth **5 points**:

- **Part 1: Dataset Selection and Preprocessing (5 pts)** – Full credit for choosing an appropriate **novel dataset with  $\geq 1000$  samples**, clearly describing the dataset and loading it successfully. We expect to see the dataset source/description and a preview of its contents. Points will be deducted if the dataset is too small, not described, or simply a copy of an example without any effort. The notebook should show that the data was split into train/val (and test if applicable), and that tokenization was applied correctly (with

evidence like example tokenized output). Efficient use of the tokenizer (with appropriate padding/truncation) and handling of any special cases (like setting pad token) will be rewarded. If the data is left unprocessed or tokenization is incorrect, points will be deducted.

- **Part 2: Fine-Tuning Setup (5 pts)** – Full credit for correctly setting up a small-scale pre-trained model for fine-tuning with parameter-efficient methods. This includes installing necessary libraries, loading the model in 8-bit, applying LoRA (or another adapter method), and freezing parameters. The output from `print_trainable_parameters()` (or equivalent) should confirm that only a small percentage of parameters will update<sup>7</sup>. Hyperparameters for training should be chosen sensibly and stated. Points are deducted for using an inappropriately large model (that doesn't fit in Colab), not using any adapter/PEFT (i.e., trying to full fine-tune all params), or misconfiguring the training setup.
- **Part 3: Training & Monitoring (5 pts)** – Full credit for successfully fine-tuning the model and documenting the training process. There should be evidence that training ran (e.g., logs or a statement of final loss), and a **plot of the training loss** over time. We expect to see that the loss decreases, showing learning. Use of a validation set and showing validation loss is a plus. Saving the model at the end is also expected. Deductions if the model was not actually trained (e.g., no proof of loss decreasing), if the plot is missing, or if the training process is not explained. Minor issues during training are okay as long as they are noted and addressed.
- **Part 4: Evaluation & Results (5 pts)** – Full credit for thorough evaluation of the fine-tuned model. This includes reporting **perplexity** on a test/val set (with calculation shown or explained) and providing **sample generated outputs** that reflect the fine-tuning. The analysis should be coherent – the student should comment on whether the outputs make sense and meet expectations. We expect at least a few example generations that demonstrate the model's behavior. Points will be deducted if no samples are shown, or if there is no commentary on the results. If only metric is given without qualitative analysis,

---

<sup>7</sup> <https://huggingface.co/docs/peft/en/quicktour#>

or vice versa, some points may be lost – we want to see both. Creative or insightful analysis will receive full credit.

Each section will be scored individually, and then summed for a total out of 25 points. Ensure that each part of your notebook addresses the points above so you can secure all the points. Good documentation, clear formatting of results, and following the instructions will all help your score. Remember, clarity and completeness are key – the grader should be able to follow your process and see the evidence of each step easily.

## Submission Instructions

You will submit your project in two parts:

### 1. A PDF file

- If you are using Colab, export your notebook as a PDF (File > Print > Save as PDF or File > Download > PDF). Make sure it includes
  - Markdown cells that clearly explain each section of your code—what it does and why.
  - Code comments to clarify each step within the code itself.
  - Clear markings to indicate any pre-existing code you reused.
  - Detailed notes on any modifications, changes, or additions you made to pre-existing code or packages, along with an explanation of why those changes were necessary.
  - *Also include your public Colab link at the first Markdown cell.*
- If you are **not using Colab**, submit a **PDF report** that explains:
  - How your code works with necessary code copies or snapshots.
  - What part of pre-existing code you re-used with necessary code copies or snapshots.
  - What part of code you modified/changed/added with detailed explanations and code copies or snapshots.
- Name your PDF: Lastname\_Firstname\_Project.pdf

### 2. A ZIP file

Include the following in a single ZIP archive:



- **README file** (`README.txt` or `README.md`) A brief document that explains:
  - How to run your code
  - Any required packages or dependencies (list them clearly)
  - Any commands to execute
  - Folder structure and where to locate important files (e.g., dataset path)
- Your **notebook (.ipynb)** and **code files**.
- The **dataset** you used for the project:
  - If it's a single file (e.g., .csv, .txt), include it directly.
  - If it's multiple files, organize them into a folder inside the ZIP.
- Name your ZIP: `Lastname_Firstname_Project.zip`

### 3. How to Submit

Upload both the **PDF** and **ZIP** files.

# Quiz Option II: Simulating a Social Phenomenon with 100+ LLM Agents

Spring 25 CSS 5120 Comp Linguistics, Zhanzhan Zhao

## Overview

Large Language Models (LLMs) can be used as autonomous agents to simulate complex social systems. In this project, you will design a simulation of **100+ LLM-powered agents** interacting to reproduce a social phenomenon of your choice – for example, opinion polarization, herd behavior, disease spread, or urban migration. Recent research shows that networks of LLM-driven agents can mimic realistic human dynamics. For instance, **LLM agents in an opinion network tend to converge on consensus unless biases are introduced, in which case fragmented camps emerge**<sup>8</sup>. Likewise, **LLM-based agents in an epidemic simulation independently chose to quarantine when sick, collectively producing multiple infection waves and “flattening” the curve, akin to real pandemics**<sup>9</sup>. This assignment will guide you through building your own multi-agent simulation, even if you have beginner-level experience

with Python and LLMs.



*Illustration: A group of LLM-powered agents engaged in a debate (here on climate change). Such multi-agent interactions can model phenomena like **opinion polarization** or **consensus formation** in a society<sup>8</sup>. Each agent is a generative model with its own persona, contributing to emergent group dynamics.*

## Quiz Objectives

<sup>8</sup> <https://github.com/yunshiuan/llm-agent-opinion-dynamics#>

<sup>9</sup>

<https://arxiv.org/abs/2307.04986#:~:text=Using%20generative%20artificial%20intelligence%20in,creates%20potential%20to%20improve%20dynamic>

By completing this quiz, you are expected to:

- Learn how to **define individual agent properties** (background, memory, goals, [actions](#)) and translate them into LLM behavior.
- Understand how to **schedule and manage interactions** among many agents over time (synchronous rounds, asynchronous events, etc.).
- Create an **external environment or context** (physical or digital) in which agents interact and influence each other.
- Observe **emergent group behavior** (e.g. formation of consensus, polarization into factions, herd-following, spread of something through the group) that arises from individual interactions.
- Practice using visualization tools to **illustrate the simulation results** (network graphs, timelines, charts), making the outcomes easy to interpret.
- Gain familiarity with **LLM tools and frameworks** that can simplify multi-agent simulations (you may adapt existing frameworks like *SimAIWorld*, *ChatArena*, or *generative\_agents* for this project).
- Consolidate your work in a **Google Colab notebook** with clear code, explanations, visualizations, and analysis, suitable for sharing and evaluation.

## **Before You Begin: Recommended Background Materials**

Before you dive into your project, we've curated a set of background materials to help you build strong conceptual foundations and gain practical coding skills. These references will give you a warm start in designing and developing LLM-based agents that exhibit complex social behaviors.

### **Core Lectures (Start Here!)**








These lectures offer essential guidance for understanding, designing, and implementing LLM agents:

- **Lecture 6 & 7** – Key concepts for building LLM agents, including decision-making and behavior modeling.

- **Lecture 2 & 3** – Prompt design and writing techniques to craft meaningful agent interactions.
- **Lecture 5** – How to choose LLM APIs, with sample code to get you started on API integration.

## **Inspirational Techniques & Frameworks**

Get hands-on with methods and tools commonly used in multi-agent LLM simulations:

- **ReAct Prompting** – A powerful prompting technique that interleaves reasoning and acting for better agent behavior.  
 [promptingguide.ai/techniques/react](https://promptingguide.ai/techniques/react)
- **AgentSociety** – A framework for simulating social interactions and emergent dynamics among LLM agents.  
 [arxiv.org/abs/2502.08691](https://arxiv.org/abs/2502.08691)
- **Oasis (CAMEL)** – A modular framework for building agents with planning, memory, and dialogue capabilities.  
 [github.com/camel-ai/oasis](https://github.com/camel-ai/oasis)
- **Generative Agent Simulations (GenAgents)** – A scalable simulation of 1,000 agents with rich internal states and memory.  
 [github.com/joonspk-research/genagents](https://github.com/joonspk-research/genagents)
- **SimAIWorld** – A GPT-powered open-world sim with autonomous agents.  
 <https://github.com/Turing-Project/SimAIWorld>
- **ChatArena** – A toolkit for running social games and language-based interactions between agents.  
 <https://github.com/Farama-Foundation/chatarena>
- **Chuang et al. (2024)** – *Simulating Opinion Dynamics with LLMs*  
Examines how polarization can emerge when LLM agents are biased.  
 <https://github.com/yunshiuan/llm-agent-opinion-dynamics>
- **Williams et al. (2023)** – *Epidemic Modeling with Generative Agents*  
Demonstrates how agents can simulate public health behaviors and multiple outbreak

phases.

🔗 <https://github.com/bear96/GABM-Epidemic>

- **Sotopia Lab (2024)** – *Awesome Social Agents*. A curated collection of papers, environments, and methods for building and evaluating social agents, with taxonomy and examples across domains like education, policy, and robotics.

🔗 <https://sotopia.world/awesome-social-agents>

## Part 0: Choose a Social Phenomenon

Pick one social phenomenon to simulate. This will be the high-level scenario that your agents inhabit. Possible choices include:

- **Opinion Polarization:** Agents hold and exchange opinions on a topic (political stance, climate change, etc.). Will the group split into opposing camps or reach a consensus?
- **Herd Behavior:** Agents make decisions by observing others (e.g. choosing products, joining trends, or panic buying). Will they end up all following a few leaders or a popular choice?
- **Disease Spread:** Agents move and interact in a community where an infection can transmit. How does an epidemic propagate and can agents' behavior (like self-isolation) slow it down?
- **Urban Migration:** Agents choose where to live based on opportunities or social networks. How do population clusters form or shift over time?

**Feel free to get creative** – you can combine elements of these or pick another social dynamic that interests you. **Describe the scenario** you've chosen and what you expect might happen in the simulation (your initial hypothesis). This will guide how you program your agents and environment.

## Part 1: Define Your LLM Agents

Design the properties and behavior rules for your agents. Each agent should be powered by an LLM (via an API or model) and have its own **persona and state**. Define at least the following for your agents:

- **Background:** A brief profile for the agent. For example, an agent could have traits like age, profession, ideology, or health status – whatever is relevant to your scenario. This background can be used in the LLM prompt to give the agent a consistent personality or starting state.
- **Memory:** A mechanism for the agent to remember past events or interactions. This could be a simple summary of the agent’s last few actions or a transcript of recent dialogues. Since LLMs have context windows, you might keep a rolling history in the prompt (e.g. last N messages) or maintain a separate memory state that you include when the agent acts.
- **Goals:** Define what each agent is trying to achieve or what drives them. Goals should tie into the phenomenon (e.g. “convince others to my opinion,” “stay healthy and avoid infection,” “maximize my social contacts,” “find a high-quality job in a city”). An agent’s goal will influence how it responds in conversations or decisions.
- **Actions:** When it is the agent’s turn to act, it chooses from a **defined set of possible actions**, depending on its **current situation** (background + memory + goals). Instead of letting the LLM freely generate behavior, you guide it through structured action options.

### Action Selection Procedure

1. Present the agent’s **current situation**:
  - Background (identity, role)
  - Memory (past interactions)
  - Goals (short-term + long-term)
2. Present the **list of possible actions** the agent can choose from.
3. Prompt the LLM:
 

*“Given the situation above, which of the following actions does the agent take and why?”*

Optionally, you can ask the LLM to choose **top 1** or **ranked top 3** actions.

### Example Action Set (can be modified per simulation)

*(Assume up to 6 distinct actions, scalable)*

Action	Description
A1: Speak	Say something to another agent (e.g., ask, inform, persuade)
A2: Move	Change location (go to a new room, join a group, leave scene)
A3: Observe	Stay quiet and gather information from surroundings

Action	Description
A4: Reflect	Update internal state or strategy based on current context
A5: Collaborate	Propose or engage in joint action with another agent
A6: Act Physically	Take physical action in the environment (e.g., pick up object, open door)

You can also **customize** actions to fit domain-specific needs (e.g., in negotiation, gaming, medical, storytelling, etc.).

- Keep it simple. You do **not** need extremely elaborate agent profiles for 100+ agents – you can create a few archetypes and clone them with slight variations. For example, in an opinion simulation you might have a set of initial beliefs distributed across agents. In a disease model, you might assign certain agents as “high social activity” and others “low activity” to vary their contact patterns. Document your agent design clearly in the notebook (perhaps create a table or list of agent types and their attributes).

**Implementation Tip:** You can represent each agent as a Python object or dictionary holding its state (background info, current memory or summary, etc.). Write a function (or class method) for the agent’s action/decision that takes in the current context (e.g. messages it received this turn or who it encounters) and produces an outcome (maybe by calling an LLM with a prompt constructed from the agent’s state). Remember that using an LLM for all 100 agents *simultaneously* can be slow or costly – consider having agents act one by one or in small groups per step, or using shorter prompts. If needed, you can limit LLM calls to the more complex decisions and use simpler random or rule-based functions for trivial interactions to reduce overhead.

## Part 2: Set Up the Environment and Simulation Schedule

Next, define the **environment** in which your agents interact, and decide on the **simulation schedule** (the timing and order of interactions). This provides the structure for how your 100+ agents will actually meet, communicate, or affect each other.

- **Environment:** Choose a representation for the world or context. This could be physical (e.g. a grid map, network of locations, or a set of regions) or purely informational (e.g. a social network graph or chat room). The environment should make it clear **who can interact with whom**. For example, in a disease or migration scenario, you might place agents on a map or graph where edges indicate contact or movement paths. In a social

opinion scenario, the environment could be a social network (friends graph) or a sequence of meetings where random pairs/groups discuss a topic. Describe how your environment is structured and how it dictates interactions. (Are interactions local? Random? Does the environment impose any rules, like limited space or resources?)

- **Simulation Schedule:** Determine how time progresses in your simulation and how agent turns are handled. You have flexibility here – **there is no single correct schedule**, but it must be clearly defined. Options include:
  - *Concurrent (Parallel) Updates:* All agents act simultaneously in each time step (you might compute their new states based on the previous step before updating everyone). This often requires making sure agents consider the old state when acting, to avoid race conditions.
  - *Discrete Time-Steps:* Simulate in rounds (e.g., day 1, day 2, ...). In each round, either all agents get one turn (in some order) to act, or a subset of agents act. Time-stepped simulation is easier to manage for beginners – you can loop over a fixed number of rounds and update agent states each round.
  - *Event-Driven (Asynchronous):* Rather than fixed rounds, have agents act whenever certain events or conditions trigger them. For example, an “event” could be an agent receiving a message or an agent reaching a location. You might use a loop that processes a queue of events. (This is more advanced; if unsure, stick to time-stepped.)

Decide on a schedule that fits your scenario. For instance, in an epidemic you may naturally go day by day (time-step). For a social network chat, you might randomly pick pairs of agents to talk in each round. **Document your choice** and how it works. If using time steps, state what one step represents in your simulation (an hour, a day, or an abstract tick). If using an event-based approach, explain what events you have and how they’re triggered.

- **Interaction Rules:** With the environment and schedule set, specify *how agents interact*. This ties together agent decision rules with the environment. For example: “At each time step, each agent will meet one of its neighbors in the network and they will exchange opinions via an LLM-facilitated dialogue,” or “In each round, randomly select 10 pairs of agents to converse,” or “Move each agent to a new location based on their goal and then resolve any encounters (contacts) at that location.” Essentially, clarify what happens in



one unit of simulation. This should align with the phenomenon (e.g. if simulating herd behavior in buying a product, maybe each step an agent decides to buy or not buy after seeing how many others bought).

**Remember:** Clearly comment and explain the environment and timeline in your code. It might help to visualize the environment setup (e.g. draw the network of agents, or map out initial agent placements) as one of your charts.

## Part 3: Implement and Run the Simulation (Google Colab)

Now you're ready to build the simulation in a Google Colab notebook. This notebook will contain all your code, outputs, and explanations. Make sure to structure it well using Markdown headings and code/comments, so someone else can follow your work easily. Your implementation should cover: the initialization of agents and environment, the simulation loop (according to your schedule), and data collection for results (so you can visualize what happened).

Key guidelines for the Colab implementation:

- **Use of LLMs:** Integrate at least one LLM to govern agent behaviors or interactions. You have flexibility in choosing the model/API. For cost-effective or free options, consider **open models** or **cheap API endpoints**. We **highly recommend using Qwen** (Alibaba's Tongyi Qianwen model) or a similar open-source model, due to Qwen's generous token allowance (it can handle very long contexts<sup>10</sup>, which means your agents can have substantial memory or long conversations). You could use Qwen via an API or through Hugging Face if available. If Qwen is not accessible, alternatives include: OpenAI's GPT-3.5<sup>11</sup>, or local models like Llama 2 if you can load them in Colab (keep in mind 100+ agents LLM calls may be slow). You do *not* need to use the most powerful model for every agent – even a smaller model can suffice for certain agents to reduce cost.
- **Frameworks (Optional):** You are **allowed (and encouraged) to use or adapt existing multi-agent simulation frameworks** to avoid reinventing the wheel. For example, **ChatArena** is a library providing multi-agent language game environments, with

---

<sup>10</sup> <https://www.prismetric.com/how-to-build-ai-agent-with-qwen-2-5/#:~:text=Unlike%20many%20models%2C%20Qwen2,driven%20interactions>

<sup>11</sup> <https://chatgpts.asia>

abstractions for multiple players and environments based on MDP (Markov Decision Process) formalisms<sup>12</sup>. It offers a flexible way to define interactions and even has a web UI for monitoring. Another example is **SimAIWorld**, an open-source project inspired by Stanford’s generative agents, which creates a virtual world of multiple GPT-based autonomous agents<sup>13</sup>. The original “**Generative Agents**” project (Park et al. 2023) demonstrated a small town simulation with agents exhibiting daily routines and memory<sup>14</sup>. You can draw inspiration or even borrow code snippets from these resources (with citation) to structure your simulation. However, make sure you **customize the scenario to your chosen phenomenon** – don’t just run a canned demo. **Be sure to check the background readings—many of them include open-source code repositories you can explore and build upon!** Using a framework might involve learning its API, so if you prefer, you can code the simulation logic manually – it’s up to you. In any case, document any external code or library you use.

- **Notebook Structure:** Organize the Colab notebook in a logical flow. A suggested structure:
  1. **Introduction** (Markdown): Briefly restate the scenario and plan.
  2. **Setup** (Code/Markdown): Import libraries, install any needed packages (e.g. `!pip install openai`, `!pip install chatarena`, etc.), and define any constants or initial data (like network structure or agent profiles).
  3. **Agent Class/Functions** (Code): Define how you represent agents and implement their decision-making (could be functions calling the LLM with prompts constructed from the agent’s state). Include some examples showing an agent prompt and the kind of response it gives, to verify your LLM integration works.
  4. **Environment & Schedule Setup** (Code): Create the environment (e.g. generate the network graph, or initialize spatial positions) and any needed scheduling helpers (like a function to get pairs of agents to interact each round, etc.).

---

<sup>12</sup> <https://github.com/Farama-Foundation/chatarena#>

<sup>13</sup> [https://blog.csdn.net/gitblog\\_00697/article/details/142131678#](https://blog.csdn.net/gitblog_00697/article/details/142131678#)

<sup>14</sup> [https://github.com/joonspk-research/generative\\_agents#](https://github.com/joonspk-research/generative_agents#)

5. **Simulation Loop** (Code): Run through the steps/events, making agents interact as per your rules. Collect data at each step – for example, log key state variables (number of infected agents, distribution of opinions, etc.) or store transcripts of some interactions for analysis. Use print statements or logging sparingly (to avoid huge outputs with 100 agents), but ensure you can tell the simulation is progressing (maybe print a summary each round or a few example agent states).
6. **Results Collection**: After the loop, organize the data you recorded into a form that can be visualized or analyzed (e.g. a list of values over time, or final states of all agents).
7. **Visualization**: Generate the required plots/graphs (see next step) and display them in the notebook.
8. **Analysis** (Markdown): Discuss what happened in the simulation. Include observations and whether they match your expectations.

Each section should be clearly labeled with Markdown headers in your Colab. Include explanatory text **before and after code blocks** to explain what the code is doing and why. This notebook should read like a mini-report as well as an executable script.

- **Debugging & Testing**: It's wise to start with a **smaller number of agents** (maybe 5–10) for initial testing, to ensure your code works and the agents behave roughly as expected, before scaling up to 100+. You might encounter issues like the LLM giving repetitive answers or taking too long; you can adjust prompts or reduce calls if needed. Once stable, increase to ~100 agents for the full simulation run. (If 100 is too slow, you can try slightly fewer, but aim for at least 100 to see rich interactions.) Monitor resource usage on Colab – you might need to request a higher-RAM runtime if using heavy models or a complex environment.

## Part 4: Observe Emergent Behavior

After running the simulation, examine the outcomes for **emergent group behaviors** – patterns or dynamics that arise from the agents' collective interactions. This is a crucial part of the assignment: you need to demonstrate that something interesting *emerged* at the group level. Consider the following when analyzing emergent behavior:

- **What patterns did you observe?** Relate this to the phenomenon you chose. For example, did you see **clusters of agreement vs. disagreement** form in an opinion

simulation? Did the population experience **multiple waves of infection** and then stabilize in a disease model? Did agents exhibit a **bandwagon effect** in herd behavior (e.g. a few choices became extremely popular)? Describe these findings qualitatively, and point out any notable events (e.g. “around time step 5, a majority of agents switched opinion to match a charismatic leader agent”).

- **Why did it happen?** Try to explain the emergent outcomes based on the rules and interactions. Maybe consensus emerged because one viewpoint was consistently reinforced, or polarization occurred because agents with similar backgrounds talked more to each other. If a trend caught on in a herd scenario, was it because those who adopted it early influenced many others? Connect the dots between agent-level rules and macro patterns. This shows you understand how local interactions can lead to global phenomena.
- **Reference real patterns or literature (optional):** It can be insightful to compare your results with known real-world outcomes or findings from research. For instance, you might note that your opinion dynamics resemble how real social networks polarize under confirmation bias<sup>15</sup>, or that the infection peaks in your simulation mirror classic SIR model epidemic curves. Such comparisons are not required, but they can enrich your analysis.
- **If no clear pattern emerged:** That’s okay – not every simulation will succeed on the first try. If your agents ended up random or no noticeable group behavior formed, discuss possible reasons and improvements. *Perhaps there wasn’t enough interaction to create a cascade effect, or the agents’ designs (background, memory, goals, and actions) are too simple.* Propose what changes could be made (e.g. stronger influence strength, longer simulation time, etc.) to potentially see the phenomenon. Reflecting on this still shows understanding.

Document these observations and explanations in a Markdown section of your notebook. This should read like the “Results and Discussion” of an experiment. Include examples or excerpts from the simulation if helpful (for instance, quote a snippet of dialogue between two agents that

---

<sup>15</sup> <https://github.com/yunshiuan/llm-agent-opinion-dynamics#:~:text=Accurately%20simulating%20human%20opinion%20dynamics,After%20inducing>

illustrates how one persuaded the other). Ensure you connect the behavior back to the concept of emergence – highlight how **no single agent was explicitly told to create that group outcome**, it arose from their interactions.

## Part 5: Visualize and Analyze Results

A picture is worth a thousand words! You are required to include **visualizations** that help demonstrate the outcomes of your simulation. Aim to create at least **2 different visualizations** (more if needed) that together capture the essence of the emergent behavior. Here are some ideas for various scenarios:

- **Timeline Chart:** Plot how a certain measure evolves over the simulation time steps. For example, in an opinion simulation, you could plot the number of agents holding each opinion over time (lines for “pro” vs “con” opinions). For a disease model, plot the counts of Susceptible, Infected, and Recovered agents vs. time (an epidemiological curve). In a migration scenario, plot population in each region vs. time. Use matplotlib or Plotly to create a clear line or bar chart with proper labels (e.g. x-axis = time step, y-axis = count or percentage). This shows the **dynamics** of the process (e.g. did one group grow or shrink?).
- **Network or Spatial Graph:** If your environment is a network or has spatial structure, visualize it. You can use **networkx** to draw a graph of agents, using colors or node sizes to indicate some agent attribute (like their final opinion, or infected/healthy status, etc.). For instance, draw a network with edges as interactions and color nodes red/blue for two factions – do you see clusters of same-color nodes (polarization)? Or plot agents on a grid/map with different symbols for moved vs. stayed in migration. A before-and-after snapshot (initial state vs final state network) can be very insightful.
- **Distribution Plot:** Show a histogram or bar chart of some final distribution across agents. For example, a bar chart of how many agents ended up in each of several opinion categories at the end, or a histogram of how many interactions each agent had (to see if a few agents dominated the conversation, a “rich-get-richer” effect).
- **Any other visualization** that makes sense for your data: maybe a heatmap of a grid (if spatial infection spread), or an animation if you’re ambitious (animated scatter plot of agents moving). Static images are fine for this assignment, but feel free to be creative if you want to show a sequence.

Make sure each visualization has: a title, labeled axes (if applicable), a legend or annotations if multiple data series are present, and a brief caption/explanation. You should also describe in your analysis text what each figure illustrates. The goal is that someone could glance at your charts and understand the high-level outcomes. Using Plotly for interactive visuals is allowed (just ensure the graphs render in the Colab and are accessible via the link), but static PNG images via matplotlib are perfectly acceptable and often easier.

**Technical note:** If using networkx for network graphs, you might need to install it (!pip install networkx) and possibly use positions or a layout algorithm for nicer appearance. For plotting timelines, pandas or matplotlib can help aggregate data per time step. If using Plotly, remember to include `plotly.io.renderers.default = 'colab'` for proper rendering in Colab.

After generating the visuals, double-check that they indeed support the story of your emergent behavior. If something is not clear, you might need to tweak what you plot or even collect additional data from the simulation. For example, if you expected a polarization but your chart shows just a single line (maybe average opinion trending to neutral), that might indicate consensus – is that what you expected or did something go differently? Use the visuals as evidence in your analysis.

## Evaluation Criteria (20 points)

Your project will be evaluated on **five key aspects** (5 points each). Make sure your submission addresses each of these clearly:

- **Agent Design (5 pts):** Quality and clarity of your agent definitions. Full points if you [defined diverse agent backgrounds](#), [included a memory mechanism](#), [specified clear goals](#), and [clear action space specifications that align with the chosen scenario](#). We will check that agent behavior is non-trivial and that you described how the LLM is used in their decision-making.
- **Simulation Setup: Schedule & Environment (5 pts):** Appropriateness of your chosen simulation schedule and environment design. Full points for a well-justified schedule (time-stepped or event-driven) that fits the phenomenon, and a clear definition of the environment (physical or digital context) enabling agent interactions. We should see that you documented how agents meet/interact each step. The design doesn't have to be complex, but it should make logical sense for the scenario.

- **Emergent Behavior Demonstration (5 pts):** Evidence of emergent group behavior and insightful analysis. Full credit if your simulation produced some recognizable group-level outcome (e.g. polarization, consensus, contagion wave, etc.) **and** you analyzed it thoughtfully. This includes describing what happened and giving a plausible explanation linking it to agent rules. Even if the result was unexpected or the pattern subtle, you get points for identifying and discussing it. (If truly nothing emerged, you can still earn partial credit by analyzing what might need to change.)
- **Code Explanation & Presentation (5 pts):** Full credit will be given to projects that demonstrate a clear understanding of the code and present it in a well-organized way.
  1. If your project is in **Colab or a Jupyter notebook**, you should include:
    1. **Markdown cells** that explain each part of the code and what it's doing.
    2. **Code comments.**
    3. Clear notes on **any modifications** you made to pre-existing code or packages, and why you made them.
  2. If your project is **not in a notebook format**, you will need to submit a **short written report** that explains:
    1. How your code works with necessary code copies or snapshots.
    2. Where you reused or adapted existing code with code copies or snapshots.
    3. What you added new codes and why with code copies or snapshots.

In all cases, your explanations should be easy to follow and show that you understand the logic of your implementation. Projects with poor or missing explanations, or that rely heavily on unmodified code without discussion, will receive lower scores.

Each section will be scored 0–5. The total project score will be the sum (out of 20 points). Strive for completeness in each area. If something didn't go as planned, explain what you did and what you learned – that's often just as important as success. We are looking for understanding and effort, not perfection.

## Submission Instructions

You will submit your project in two parts:

### 1. A PDF file

- If you are using Colab, export your notebook as a PDF (File > Print > Save as PDF or File > Download > PDF). Make sure it includes
  - Markdown cells that clearly explain each section of your code—what it does and why.
  - Code comments to clarify each step within the code itself.
  - Clear markings to indicate any pre-existing code you reused.
  - Detailed notes on any modifications, changes, or additions you made to pre-existing code or packages, along with an explanation of why those changes were necessary.
  - *Also include your public Colab link at the first Markdown cell.*
- If you are **not using Colab**, submit a **PDF report** that explains:
  - How your code works with necessary code copies or snapshots.
  - What part of pre-existing code you re-used with necessary code copies or snapshots.
  - What part of code you modified/changed/added with detailed explanations and code copies or snapshots.
- Name your PDF: Lastname\_Firstname\_Project.pdf

### 2. A ZIP file

Include the following in a single ZIP archive:

- **README file** (`README.txt` or `README.md`) A brief document that explains:
  - How to run your code
  - Any required packages or dependencies (list them clearly)
  - Any commands to execute
  - Folder structure and where to locate important files (e.g., dataset path)
- Your **notebook (.ipynb) and code files**.
- The **dataset** you used for the project:
  - If it's a single file (e.g., .csv, .txt), include it directly.
  - If it's multiple files, organize them into a folder inside the ZIP.
- Name your ZIP: Lastname\_Firstname\_Project.zip



### 3. How to Submit

Upload both the **PDF** and **ZIP** files.

We're excited to see the creative simulations you build! This project is an opportunity to explore how **simple rules and language-model interactions can lead to complex social phenomena**.

Good luck, and have fun with your AI agents! If you have any questions or need clarification, don't hesitate to ask on the class forum.