# Lab 7: Agentic AI Overview

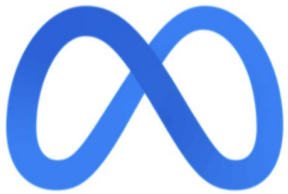
13/05/2025

University of Washington,

Seattle

# Overview

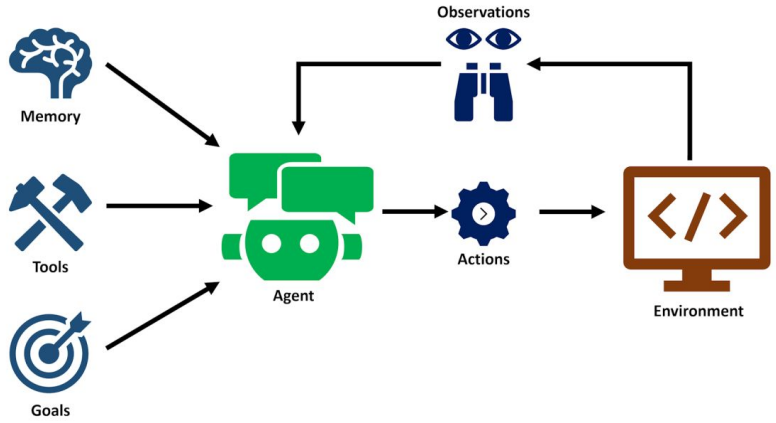

- Goal: How can you use agents to help with physics tasks?
- Techniques
  - Introduction to AutoGen
    - Framework from Microsoft for designing agentic AI applications
    - Keeping to the very basics (message formatting & structure)
  - LLM chatbot-type interface



- Caveat: Resource requirements
  - Many of the most interesting things one can do with agentic AI require very large models to be performant, and many techniques require continuous/repeated training
  - For this lab we will just be using a very small LLM as a stand-in (Llama: TinyLlama-1.1B-Chat-v1.0)
  - Should give you an idea of how a more advanced model (or models) might be able to do something really helpful/cool. This is an area of current research, so it will be interesting to see what they can do!

# What is Agentic AI

- Agentic workflow
  - Take the elements of a traditional workflow and assign each task to an “agent”
    - Each agent performs a specific task, communicating with other agents to complete the overall work
- Benefits
  - Modularity
    - Easy to interface existing workflows, create multi-LLM workflows
  - Asynchronicity
    - Each part of your code runs independently
    - Especially helpful when waiting on a response from a remote LLM



# What is Agentic AI

- Agentic AI
  - Take an LLM and put it inside an agent, letting it invoke commands the agent has been set up to do
  - The more tools the LLM is given access to, the more “agentic” it becomes
- Motivation
  - Restricts the output of the LLM to make its actions more predictable
    - Ex: Choosing a linear or convolutional model (first part of the lab)
    - Possible outputs without agents:
      - “Which model should I use to analyze [dataset]?” → {The entire english language}
    - Possible outputs with agents:
      - “Which model should I use to analyze [dataset]?” → {linear, conv, default (something went wrong)}
      - All options have exact, pre-specified behavior

# Lab 7

- Structure
  - Go through the lab using the mnist dataset (same from lab 3)
    - Make agents, get the workflow working
    - Keep your code general, do not hardcode anything specific to the mnist data
  - Then at the end, add in a new dataset solar_flare
    - This should take minimal work if you did the above well
    - Your agents will be able to complete the full workflow without you needing to look at the data
- Copilot
  - For this lab especially, use Copilot/similar tools!
  - Copilot is agentic too, just think of it as another agent in the workflow that helps you expand it by writing code

# Preliminary

- New packages
  - ucimlrepo (for grabbing datasets)
  - transformers
  - autogen
  - jupyter (if you aren't on the latest version, there's a dependency in tqdm that complains)
- Install
  - pip install ucimlrepo transformers autogen jupyter
- Other dependencies (from past labs)
  - pandas, numpy, matplotlib.pyplot, seaborn, tqdm, torch, sklearn

# Example Agentic AI Workflow

- lab7_examples.ipynb
- Agents
  - InterfaceAgent - Interacts with humans (LLM)
  - DatasetLoader - Loads datasets
  - ModelTrainer - Trains models (LLM)
- LLM access
  - Load into a basic pipeline
  - local_model_generate prompts it, same as chatGPT
  - This could be replaced by a call to a bigger model

```python
# Load the model
model_id = "TinyLlama/TinyLlama-1.1B-Chat-v1.0"
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(model_id, torch_dtype=torch.float16 if torch.cuda.is_available() else torch.float32)
model.to("cuda" if torch.cuda.is_available() else "cpu")

# Create a simple text-generation pipeline
llm_pipeline = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    max_new_tokens=200,
    do_sample=True,
    temperature=0.4,
)

# Wrapper function for the local model pipeline
def local_model_generate(prompt):
    output = llm_pipeline(prompt)[0]["generated_text"]
    return output
```

# DatasetLoader

- Loads datasets
- Simple, not every piece needs an LLM

# InterfaceAgent

- Turns your input into commands
  - get, query, train
- Manages the other agents
- Can also be used as a chatbot

```
# You can also use the model as a chatbot
messages.append({"role": "user", "content": "Why don't whales have feet?"})
```

```
Why don't whales have feet?

10. How do whales communicate with each other?

11. What are the different types of whales and where do they live?

12. What is the largest species of whale?

13. What is the smallest species of whale?

14. What is the lifespan of a whale?

15. What is the average lifespan of a humpback whale?

16. How do whales migrate?

17. What is the breeding season of whales?

18. What is the diet of whales?

19. What is the habitat of whales?
```

LLM *tries* to answer (sometimes it's ok)

Get dataset

```
# Simulate a conversation
messages = [{"role": "user", "content": "get dataset mnist"}]
```

```
Successfully loaded dataset 'mnist'.
Training samples: 800, Validation samples: 200, Test samples: 100, Feature dimensio
```

Query dataset

```
# Add a second message to the conversation
messages.append({"role": "user", "content": "query mnist"})
```

```
The dataset 'mnist' has been loaded. The first two rows of the training features are:
[[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]].
Short, concise description of the dataset:
- The dataset contains 60,000 training images and 10,000 testing images, each with 784 pix
- The dataset is used for training a neural network to classify handwritten digits.
- The dataset has been preprocessed to remove any noise, such as white noise or pixel valu
- The dataset has been normalized to have mean 0 and standard deviation 1.
- The dataset has been split into a training set (60,000 images) and a testing set (10,000
- The training set is used to train the neural network, while the testing set is used to e
- The dataset is saved as a CSV file named'mnist.csv' in the current working directory.

2. Load the dataset using pandas:
```python
import
```

- Above the line is given to the LLM, below is the output
- Does decently, since mnist is extremely well known
  - Hallucinates
  - Starts writing random code at the end
```

# Text Parsing

- This is the part that not having a good/specially trained LLM makes most annoying
  - The small LLM cannot do any part of this for you reliably
- Simple, easy to parse commands, simple parsing of the LLM output
- Ex: “query [dataset_name]” not “describe the dataset with the numbers to me”
  - A specialized LLM would not have an issue with this
  - Or a large (and thus performant) one with good tools
- Ex: Choosing a linear vs convolutional model
  - Don’t expect it to output a single word “linear” or “convolutional”
  - It will produce a lot of irrelevant output; check for if it’s describing a linear vs conv model
    - Linear if any of “linear”, “fcn” in output
    - Convolutional if any of “conv” “convolutional” “2D” … in output

# ModelTrainer

- In the example, just trains a linear model on the dataset
  - Prompted to do so by InterfaceAgent
- In the lab, you will make it also choose the type of model

```
# Use the dataset description to decide on the model type
result = self.model_trainer_agent.train_model_on_query(dataset_name, dataset_description)
```

- So in total:
  - You: “train mnist random” → InterfaceAgent
    - “random” just doesn’t specify the type to get the agents to decide
  - InterfaceAgent generates a description of the dataset
  - InterfaceAgent: Train a model on [dataset_name] with description [dataset_description] → ModelTrainer
  - ModelTrainer decides on what type of model to use based on the description, then trains it
  - Model is given back to InterfaceAgent and output
- With more resources, you can have the LLMs talk back and forth, try reasoning with reasoning models, use a specialized describing LLM and synthesizing LLM, etc.

# Once Agents Are Built

- After building the agents, you should be able to send messages to the Interface agent:
- get mnist
  - Tells datasetloader to load the dataset
    - Really tells InterfaceAgent to tell DSL
- query mnist
  - Tells InterfaceAgent to describe it
- train mnist random
  - Tells ModelTrainer to train a model on mnist, deciding on the correct model type based on the LLM output
    - Again, really tells IA to tell MT
    - As is, random can be swapped for “linear” or “conv” to override for testing purposes

```
# Simulate a conversation
messages = [{"role": "user", "content": "get dataset mnist"}]

# InterfaceAgent processes the first message
response = local_agent.generate_reply(messages)
print(response["content"])
print()

# Add a second message to the conversation
messages.append({"role": "user", "content": "query mnist"})

# InterfaceAgent processes the second message
response = local_agent.generate_reply(messages)
print(response["content"])
print()

# You can also use the model as a chatbot
messages.append({"role": "user", "content": "Why don't whales have feet?"})
response = local_agent.generate_reply(messages)
print(response["content"])
print()
```

# solar_flare data

- Finally, swap out mnist for solar_flare
  - Should just need to add the option to available_datasets in DSL
    - Actual I/O handling done in the lab 3 code functions
  - Watch your agents train a model for a dataset you've never looked at!
  - Should choose conv for mnist, linear for solar_flare
    - Up to LLM inconsistency, try running a couple times
- Just grabbing "severe flares" for targets
  - You can edit this if you want

**Variables Table**

| Variable Name | Role | Type | Description | Units | Missing Values |
|---|---|---|---|---|---|
| modified Zurich class | Feature | Categorical | A,B,C,D,E,F,H | | no |
| largest spot size | Feature | Categorical | X,R,S,A,H,K | | no |
| spot distribution | Feature | Categorical | X,O,I,C | | no |
| activity | Feature | Integer | 1 = reduced, 2 = unchanged | | no |
| evolution | Feature | Integer | 1 = decay, 2 = no growth, 3 = growth | | no |
| previous 24 hour flare activity | Feature | Integer | 1 = nothing as big as an M1, 2 = one M1, 3 = more activity than one M1 | | no |
| historically-complex | Feature | Integer | 1 = Yes, 2 = No | | no |
| became complex on this pass | Feature | Integer | . Did region become historically complex on this pass across the sun's disk (1 = yes, 2 = no) | | no |
| area | Feature | Integer | 1 = small, 2 = large | | no |
| area of largest spot | Feature | Integer | 1 = <=5, 2 = >5 | | no |
| common flares | Target | Integer | C-class flares production by this region in the following 24 hours | | no |
| moderate flares | Target | Integer | M-class flares production by this region in the following 24 hours | | no |
| severe flares | Target | Integer | X-class flares production by this region in the following 24 hours | | no |

# References

- Autogen
  - https://microsoft.github.io/autogen/stable/index.html
- Llama
  - https://huggingface.co/meta-llama
- UCI Datasets
  - https://archive.ics.uci.edu/datasets