Lab 7 - Travis Hand

Preliminary

Packages to install

- ucimlrepo (for grabbing datasets)
- transformers
- autogen
- jupyter (if you aren't on the latest version, there's a dependency in tqdm that complains)

Install with pip

- pip install ucimlrepo transformers autogen jupyter

From lab 3

- pandas, numpy, matplotlib.pyplot, seaborn, tqdm, torch, sklearn

Install everything with pip

- pip install ucimlrepo transformers autogen pandas numpy matplotlib seaborn tqdm torch scikit-learn jupyter

```
# If you need to install stuff on colab
!pip install ucimlrepo transformers autogen pandas numpy matplotlib seaborn tqdm torch scikit-learn jupyter
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: ucimlrepo in /Users/travishand/Library/Python/3.12/lib/python/site-packages (0.0.7)
Requirement already satisfied: transformers in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-pack
Requirement already satisfied: autogen in /Users/travishand/Library/Python/3.12/lib/python/site-packages (0.9.1.post0)
Requirement already satisfied: pandas in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (
Requirement already satisfied: numpy in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (1
Requirement already satisfied: matplotlib in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packag
Requirement already satisfied: seaborn in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
Requirement already satisfied: tqdm in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (4.
Requirement already satisfied: torch in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (2
Requirement already satisfied: scikit-learn in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-pack
Requirement already satisfied: jupyter in /Users/travishand/Library/Python/3.12/lib/python/site-packages (1.1.1)
Requirement already satisfied: certifi>=2020.12.5 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/sit
Requirement already satisfied: filelock in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
Requirement already satisfied: huggingface-hub<1.0,>=0.23.2 in /Library/Frameworks/Python.framework/Versions/3.12/lib/pyth
Requirement already satisfied: packaging>=20.0 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-p
Requirement already satisfied: pyyaml>=5.1 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packa
Requirement already satisfied: regex!=2019.12.17 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site
Requirement already satisfied: requests in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
Requirement already satisfied: safetensors>=0.4.1 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/sit
Requirement already satisfied: tokenizers<0.20,>=0.19 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Requirement already satisfied: fsspec>=2023.5.0 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
Requirement already satisfied: typing-extensions>=3.7.4.3 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python
Requirement already satisfied: ag2==0.9.1post0 in /Users/travishand/Library/Python/3.12/lib/python/site-packages (from aut
Requirement already satisfied: anyio<5.0.0,>=3.0.0 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/si
Requirement already satisfied: asyncer==0.0.8 in /Users/travishand/Library/Python/3.12/lib/python/site-packages (from ag2=
Requirement already satisfied: diskcache in /Users/travishand/Library/Python/3.12/lib/python/site-packages (from ag2==0.9.
Requirement already satisfied: docker in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (
Requirement already satisfied: httpx<1,>=0.28.1 in /Users/travishand/Library/Python/3.12/lib/python/site-packages (from ag
Requirement already satisfied: pydantic<3,>=2.6.1 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/sit
Requirement already satisfied: python-dotenv in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-pac
Requirement already satisfied: termcolor in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-package
Requirement already satisfied: tiktoken in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
Requirement already satisfied: idna>=2.8 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-package
Requirement already satisfied: sniffio>=1.1 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-pack
Requirement already satisfied: httpcore==1.* in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-pac
Requirement already satisfied: h11<0.15,>=0.13 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-p
Requirement already satisfied: annotated-types>=0.4.0 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Requirement already satisfied: pydantic-core==2.20.1 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/
Requirement already satisfied: python-dateutil>=2.8.2 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Requirement already satisfied: pytz>=2020.1 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-pack
Requirement already satisfied: tzdata>=2022.7 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-pa
Requirement already satisfied: contourpy>=1.0.1 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
Requirement already satisfied: cycler>=0.10 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-pack
Requirement already satisfied: fonttools>=4.22.0 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site
Requirement already satisfied: kiwisolver>=1.3.1 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site
Requirement already satisfied: pillow>=8 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-package
Requirement already satisfied: pyparsing>=2.3.1 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
Requirement already satisfied: networkx in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
Requirement already satisfied: jinja2 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (
Requirement already satisfied: setuptools in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packag
Requirement already satisfied: sympy==1.13.1 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-pac
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/sit
Requirement already satisfied: scipy>=1.6.0 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-pack
Requirement already satisfied: joblib>=1.2.0 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-pac
Requirement already satisfied: threadpoolctl>=3.1.0 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/s
```

Intro

The goal of this lab is to give you an idea of how you could use agents to help with physics tasks. It will also introduce you to AutoGen, one of the more popular frameworks at the moment for designing custom agentic workflows. We won't make use of all the tools it provides, just the very basics. Note also that many of the most interesting things one can do with AI-powered agents (see topics like retreival augmented generation (RAG)) require very large models to be performant, and many techniques require continuous/repeated training. This means large resource requirements, so for this lab we will just be using a very small LLM (Llama: TinyLlama-1.1B-Chat-v1.0). The results from this are nowhere near as good as something like chatGPT, but it 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!

Also because of the limited size of the model, the text parsing needs to be very mechanical, and in some places a bit obtuse. The better (and ideally specially trained for an agentic workflow, see RAG) your model is, the more this can be relaxed. If you're interested in working with agents in a more user-friendly way (hiding a lot of the mechanics that are on display here), check out sites like n8n or Google Gemini (be aware that these can require you to provide a lot of permissions). You can also feel free to replace TinyLlama with a call to a larger model using API keys if you have some.

Finally, we would like to emphasize the use of copilot/similar tools for this lab in particular. These are agents too! And they are definitely the most performant agents you have easy access to.

Lab

- There are two datasets, "mnist" (from Lab 3) and "solar_flare"
- Try to get everything working with mnist, then try adding solar_flare
  - This is one way agents can be helpful, since they can analyze a dataset you've never seen before and take a first crack at it much faster than you can

Imports

```
# Import necessary packages

%matplotlib inline

from autogen import ConversableAgent, AssistantAgent
from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline

import os
import certifi
os.environ['SSL_CERT_FILE'] = certifi.where()
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import tqdm
import torch
import ssl
from torch.utils.data import DataLoader, TensorDataset
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from ucimlrepo import fetch_ucirepo

# Disable SSL certificate verification
ssl._create_default_https_context = ssl._create_unverified_context
```

```
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages/pydantic/_internal/_fields.py:161: UserWarni

You may be able to resolve this warning by setting `model_config['protected_namespaces'] = ()`.
```

```
  warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages/pydantic/_internal/_fields.py:161: UserWarni

You may be able to resolve this warning by setting `model_config['protected_namespaces'] = ()`.
  warnings.warn(
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages/pydantic/_internal/_fields.py:161: UserWarni

You may be able to resolve this warning by setting `model_config['protected_namespaces'] = ()`.
  warnings.warn(
```

Lab 3 code (run and then minimize this)

- Mostly just lab 3 code packaged into functions

```
# Load MNIST Dataset in Numpy
def load_mnist_data():
    # 1000 training samples where each sample feature is a greyscale image with shape (28, 28)
    # 1000 training targets where each target is an integer indicating the true digit
    mnist_train_features = np.load('mnist_train_features.npy')
    mnist_train_targets = np.load('mnist_train_targets.npy')

    # 100 testing samples + targets
    mnist_test_features = np.load('mnist_test_features.npy')
    mnist_test_targets = np.load('mnist_test_targets.npy')

    # Print the dimensions of training sample features/targets
    #print(mnist_train_features.shape, mnist_train_targets.shape)
    # Print the dimensions of testing sample features/targets
    #print(mnist_test_features.shape, mnist_test_targets.shape)

    return mnist_train_features, mnist_train_targets, mnist_test_features, mnist_test_targets


def flatten_features(features):
    # Flatten the features from (28, 28) to (784,)
    return features.reshape(features.shape[0], -1)

def scale_features(features):
    scaler = StandardScaler()
    return scaler.fit_transform(features)

# More general function to load datasets, including solar_flare
def load_dataset(dataset_name: str = ""):
    if dataset_name == "mnist":
        train_features, train_targets, test_features, test_targets = load_mnist_data()
        train_features = flatten_features(train_features)
        test_features = flatten_features(test_features)
        train_features = scale_features(train_features)
        test_features = scale_features(test_features)

    elif dataset_name == "solar_flare":
        # Load the solar flare dataset
        solar_flare = fetch_ucirepo(id=89)

        # Simplifying slightly for the sake of this example
        solar_flare.data.targets = solar_flare.data.targets['severe flares']

        # Split the solar flare dataset into train and test sets (90:10 split)
        train_features, test_features, train_targets, test_targets = train_test_split(
            solar_flare.data.features, solar_flare.data.targets, test_size=0.1, random_state=42
        )

        # Onehot encode modified Zurich class, largest spot size, spot distribution
        onehot_columns = ["modified Zurich class", "largest spot size", "spot distribution"]
        for col in onehot_columns:
            onehot = pd.get_dummies(train_features[col], prefix=col)
            train_features = pd.concat([train_features, onehot], axis=1)
            train_features.drop(col, axis=1, inplace=True)

            onehot = pd.get_dummies(test_features[col], prefix=col)
            test_features = pd.concat([test_features, onehot], axis=1)
            test_features.drop(col, axis=1, inplace=True)

        # Scale the features
        train_features = scale_features(train_features)
```

```
            test_features = scale_features(test_features)

            # Convert targets to numpy arrays
            train_targets = train_targets.to_numpy()
            test_targets = test_targets.to_numpy()

        else:
            raise ValueError(f"Unknown dataset: {dataset_name}")

        # train-test split
        train_features, val_features, train_targets, val_targets = train_test_split(train_features, train_targets, test_size=0.2)

        return train_features, train_targets, val_features, val_targets, test_features, test_targets


    # Train
    def train_model(model, train_features, train_targets, validation_features, validation_targets,
                    test_features=None, test_targets=None, learning_rate=0.0015, epochs=80, batch_size=64):
        """
        Train a neural network model on the provided data.

        Parameters:
            model: PyTorch model to train
            train_features: Training features as numpy array
            train_targets: Training targets as numpy array
            validation_features: Validation features as numpy array
            validation_targets: Validation targets as numpy array
            test_features: Test features as numpy array (optional)
            test_targets: Test targets as numpy array (optional)
            learning_rate: Learning rate for optimizer
            epochs: Number of training epochs
            batch_size: Batch size for training

        Returns:
            tuple: (trained model, training loss list, validation accuracy list)
        """
        # Initialize tracking lists
        train_loss_list = np.zeros(epochs)
        validation_accuracy_list = np.zeros(epochs)

        # Convert numpy arrays to PyTorch tensors
        train_inputs = torch.from_numpy(train_features).float()
        train_targets = torch.from_numpy(train_targets).long()

        validation_inputs = torch.from_numpy(validation_features).float()
        validation_targets = torch.from_numpy(validation_targets).long()

        if test_features is not None and test_targets is not None:
            test_inputs = torch.from_numpy(test_features).float()
            test_targets = torch.from_numpy(test_targets).long()
            test_dataset = TensorDataset(test_inputs, test_targets)
            test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

        # Create dataloaders
        train_dataset = TensorDataset(train_inputs, train_targets)
        train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
        validation_dataset = TensorDataset(validation_inputs, validation_targets)
        validation_loader = DataLoader(validation_dataset, batch_size=batch_size, shuffle=False)

        # Setup optimizer and scheduler
        loss_func = torch.nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
        scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=epochs)

        # Move model to GPU if available
        if torch.cuda.is_available():
            model = model.cuda()
            train_inputs = train_inputs.cuda()
            validation_inputs = validation_inputs.cuda()
            validation_targets = validation_targets.cuda()

        # Training Loop
        for epoch in tqdm.trange(epochs):
            model.train()  # Set model to training mode
            running_loss = 0.0

            for batch_inputs, batch_targets in train_loader:
```

```
            if torch.cuda.is_available():
                batch_inputs, batch_targets = batch_inputs.cuda(), batch_targets.cuda()

            optimizer.zero_grad()  # Reset gradients to zero
            outputs = model(batch_inputs)  # Forward pass with current batch
            loss = loss_func(outputs, batch_targets)  # Compute loss
            loss.backward()  # Backward pass
            optimizer.step()  # Update weights

            running_loss += loss.item() * batch_inputs.size(0)

        # Store average epoch loss
        train_loss_list[epoch] = running_loss / len(train_dataset)
        scheduler.step()  # Update learning rate with cosine annealing

        # Compute Validation Accuracy
        model.eval()  # Set model to evaluation mode
        with torch.no_grad():
            correct = 0
            total = 0
            for val_inputs, val_targets in validation_loader:
                if torch.cuda.is_available():
                    val_inputs, val_targets = val_inputs.cuda(), val_targets.cuda()
                outputs = model(val_inputs)
                _, predicted = torch.max(outputs.data, 1)
                total += val_targets.size(0)
                correct += (predicted == val_targets).sum().item()

            validation_accuracy_list[epoch] = correct / total

    # Compute test accuracy if test data is provided
    test_accuracy = None
    if test_features is not None and test_targets is not None:
        model.eval()
        with torch.no_grad():
            correct = 0
            total = 0
            for test_inputs, test_targets in test_loader:
                if torch.cuda.is_available():
                    test_inputs, test_targets = test_inputs.cuda(), test_targets.cuda()
                outputs = model(test_inputs)
                _, predicted = torch.max(outputs.data, 1)
                total += test_targets.size(0)
                correct += (predicted == test_targets).sum().item()

            test_accuracy = correct / total

    return model, train_loss_list, validation_accuracy_list, test_accuracy


# Visualize and evaluate
def visualize_training(train_loss_list, validation_accuracy_list):
    """
    Visualize training loss and validation accuracy.

    Parameters:
        train_loss_list: List of training losses
        validation_accuracy_list: List of validation accuracies
    """
    plt.figure(figsize = (12, 6))

    # Visualize training loss with respect to iterations (1 iteration -> single batch)
    plt.subplot(2, 1, 1)
    plt.plot(train_loss_list, linewidth = 3)
    plt.ylabel("training loss")
    plt.xlabel("epochs")
    sns.despine()

    # Visualize validation accuracy with respect to epochs
    plt.subplot(2, 1, 2)
    plt.plot(validation_accuracy_list, linewidth = 3, color = 'gold')
    plt.ylabel("validation accuracy")
    sns.despine()

    plt.show()
```

Define Model(s)

- Here you should make two models, one linear and one convolutional. The agents will use these later.
- Copilot/similar tools are encouraged for this; remember that they are agents too!
- Also you can add more if you want
- Also make sure to reshape the input data to the correct shape. Each model might be given mnist data or later solar_flare data. But these have different shapes! Mnist is AxBx1, but solar_flare is just Ax1
  - You don't need to perfect this on your first pass through, just make sure both with mnist and you can revisit later

```
class fcnClassification(torch.nn.Module):

    def __init__(self, input_size=784, output_size=10): # Feel free to add parameters here
        super(fcnClassification, self).__init__()

        self.description = "" # YOUR CODE HERE (short str description of the model)

        self.linear1 = torch.nn.Linear(input_size, 512)
        self.linear2 = torch.nn.Linear(512, 256)
        self.linear3 = torch.nn.Linear(256, output_size)

        self.drop1 = torch.nn.Dropout(p=0.2)

    def forward(self, x):

        x = torch.relu(self.linear1(x))
        x = self.drop1(x)
        x = torch.sigmoid(self.linear2(x))
        x = self.drop1(x)
        x = self.linear3(x)
        # x = torch.softmax(x, dim=1)
        return x


# Uses convolutional layers
class ConvClassification(torch.nn.Module):

    def __init__(self, input_size, output_size):
        super(ConvClassification, self).__init__()

        self.description = "" # YOUR CODE HERE (short str description of the model)

        self.conv1 = torch.nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = torch.nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)

        self.pool1 = torch.nn.MaxPool2d(kernel_size=2, stride=2)

        self.fc1 = torch.nn.Linear(128 * 7 * 7, 256)
        self.fc2 = torch.nn.Linear(256, 10)

        # YOUR CODE HERE

    def forward(self, x):

        x = self.pool1(torch.relu(self.conv1(x)))
        x = self.pool1(torch.relu(self.conv2(x)))
        x = self.view(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)

        return x
```

Agents

DatasetLoader

Here is the definition of the agent that handles dataset loading. The logic is pretty simple, so nothing for you to code here

```python
class DatasetLoaderAgent(AssistantAgent):
    """
    An agent that specializes in loading and preprocessing datasets.
    """
    def __init__(self, name="DatasetLoader", **kwargs):
        self.available_datasets = ["mnist"]
        system_message = (
            "I am a dataset loading assistant. I can load and preprocess various datasets for machine learning tasks. "
            f"Currently I support: {', '.join(self.available_datasets)}. "
        )
        super().__init__(name=name, system_message=system_message, **kwargs)

        # Store loaded datasets
        self._loaded_datasets = {}

    def get_available_datasets(self):
        """Return a list of available datasets."""
        return self.available_datasets

    def load_dataset(self, dataset_name):
        """
        Load and preprocess a dataset.

        Args:
            dataset_name (str): Name of the dataset to load

        Returns:
            dict: Information about the loaded dataset and the data itself
        """
        try:
            # Load the dataset using the existing function
            train_features, train_targets, val_features, val_targets, test_features, test_targets = load_dataset(dataset_name)

            # Store the dataset
            self._loaded_datasets[dataset_name] = {
                "train_features": train_features,
                "train_targets": train_targets,
                "validation_features": val_features,
                "validation_targets": val_targets,
                "test_features": test_features,
                "test_targets": test_targets,
            }

            # Return information about the loaded dataset
            return {
                "status": "success",
                "dataset_name": dataset_name,
                "train_samples": train_features.shape[0],
                "validation_samples": val_features.shape[0],
                "test_samples": test_features.shape[0],
                "feature_dim": train_features.shape[1]
            }
        except Exception as e:
            return {
                "status": "error",
                "message": f"Failed to load dataset '{dataset_name}': {str(e)}"
            }

    def get_dataset(self, dataset_name):
        """
        Retrieve a previously loaded dataset.

        Args:
            dataset_name (str): Name of the dataset to retrieve

        Returns:
            dict: The dataset components or None if not found
        """
        return self._loaded_datasets.get(dataset_name)
```

InterfaceAgent

- Setting up the LLM

```
# 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,
    device=0 if torch.cuda.is_available() else -1
)

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

Agent definition

- This is the definition of the actual agent you will converse with
- Implement the query and get dataset commands
- You can come back to the train command after you get to the model trainer agent

```
# Agent definition
class InterfaceAgent(ConversableAgent):
    def __init__(self, name, **kwargs):
        super().__init__(name, **kwargs)

        self.dataset_loader_agent = None
        self.model_trainer_agent = None

    def set_dataset_loader_agent(self, agent):
        self.dataset_loader_agent = agent

    def set_model_trainer_agent(self, agent):
        self.model_trainer_agent = agent

    def generate_reply(self, messages):
        """
        There is a lot of obtuse text parsing and such here. I would describe the code as "technically functional".
        This is becasue the LLM is so limited. With a better LLM, and/or ideally one trained explicitly for
        agentic implementation, this would be much cleaner and more flexible.
        C.f. retrieval augmented generation, etc.
        """
        # Extract the latest user message
        user_message = messages[-1]["content"]

        # Check if the message is a "get dataset" command
        if "get dataset" in user_message:

            # YOUR CODE HERE
            # Parse the user input to extract the dataset name, e.g., "get dataset mnist"
            # Then use the DatasetLoaderAgent to load the dataset
            dataset_name = user_message.replace("get dataset", "").strip()
            result = self.dataset_loader_agent.load_dataset(dataset_name)

            # Check if the dataset was successfully loaded
            if result["status"] == "success":
                response_text = (
                    f"Successfully loaded dataset '{result["dataset_name"]}'.\n"
                    f"Training samples: {result['train_samples']}, "
                    f"Validation samples: {result['validation_samples']}, "
                    f"Test samples: {result['test_samples']}, "
                    f"Feature dimension: {result['feature_dim']}."
                )
            else:
                response_text = f"Failed to load dataset '{dataset_name}': {result['message']}"

        elif "query" in user_message:
            # YOUR CODE HERE
            # Parse the user input to extract the dataset name, e.g., "query mnist"
            query = user_message.replace("query", "").strip()
            dataset_name = query.split()[0]
            dataset = self.dataset_loader_agent.get_dataset(dataset_name)

            if dataset:
                # Prepare a prompt with dataset information
                prompt = (
                    f"The dataset '{dataset_name}' has been loaded. "
                    f"The first two rows of the training features are:\n"
                    f"{dataset['train_features'][:2]}.\n"
                    "Short, concise description of the dataset:\n"
                )
                # Generate a description using the local model
                response_text = local_model_generate(prompt)
            else:
                response_text = f"Dataset '{dataset_name}' is not loaded or does not exist."


        elif "train" in user_message:
            # Of the form "train <dataset_name> <model_type>"
            # Again, this is where a specialized LLM would be really useful to avoid this parsing
            # Model type
            parts = user_message.split()
            dataset_name, modeltype = parts[1], parts[2]

            # Use the ModelTrainerAgent to train the model
            if self.model_trainer_agent is not None:
                # If the user specified an available model type, use that
                if modeltype in self.model_trainer_agent.available_models:
                    result = self.model_trainer_agent.train_model_on_query(dataset_name, "placeholder", override=modeltype)

                # Otherwise, use the dataset description to decide on the model type
                else:

                    # YOUR CODE HERE
                    # Have the LLM generate a description of the dataset, then give that to the model trainer agent

                    dataset_description = local_model_generate(prompt=
                        f"The dataset '{dataset_name}' has been loaded. "
                        "Short, concise description of the dataset:\n"
                    )
                    # Use the dataset description to decide on the model type
                    result = self.model_trainer_agent.train_model_on_query(dataset_name, dataset_description)

                if result["status"] == "success":
                    response_text = f"Successfully trained the model '{modeltype}'."
                else:
                    response_text = f"Failed to train the model '{modeltype}': {result['response']}"
            else:
                response_text = "Model trainer agent is not set."

        # No special command, just a normal message. Can just use the model as a chatbot
        else:
            response_text = local_model_generate(user_message)

        # Return the response in the Autogen format
        return {"role": "assistant", "content": response_text}
```

Talk to the datasetloader

```
# Create instances of the agents
dataset_loader_agent = DatasetLoaderAgent(name="DatasetLoader")
local_agent = InterfaceAgent(name="LocalAgent")
local_agent.set_dataset_loader_agent(dataset_loader_agent)

# 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, querying mnist

messages.append({"role": "user", "content": "query mnist"})
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()

# Note that the output is not reliable at all since the model is tiny. Sometimes it's surprisingly good though
```

```
Successfully loaded dataset 'mnist'.
Training samples: 800, Validation samples: 200, Test samples: 100, Feature dimension: 784.

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 MNIST dataset contains 60,000 28x28 grayscale images of handwritten digits ranging from 0 to 9. The training data consis

Dataset Description:

The MNIST dataset is a classic dataset for training neural networks. It consists of 60,000 28x28 grayscale images of handwri

Why don't whales have feet?

1. Whales are marine mammals, which means they live in the ocean. They don't have feet, as they rely on their tail for balan

2. Whales have a highly specialized circulatory system that allows them to live in the ocean, where water is much cooler tha

3. Whales have very long, streamlined bodies that allow them to move through the water with minimal friction. This helps the

4. Whales also have a highly efficient respiratory system that allows them to breathe underwater without the need for specia

5. Finally, whales have a unique sense of smell that allows them to locate prey in the ocean. This helps them
```

Model Trainer

This is the agent that actually trains a model for analyzing the dataset. In InterfaceAgent, the train command should generate a description of the dataset and feed it to the train_model_on_query method below (if the user didn't explicitly specify a model type). So use the LLM to generate a recommended model type from the description ("query").

If you want, you could implement some more complex logic here, getting the LLM to recommend hyperparameters and things like that. There is also some amount of talking back and forth with itself here, in a chain-of-reasoning style. With a much bigger model, you could start to see some really interesting behavior here. This is what gives DeepSeek its power!

Note

Be careful with how you handle the input dimensions for your models. Convolutional networks want different dimensions than an fully linear network. I recommend making it so that you can feed them the data, and the model itself handles any reshaping needs.

```
class ModelTrainerAgent(ConversableAgent):
    """
    An agent that selects, creates, and trains a classification model based on the user's query.
    """
    def __init__(self, name="ModelTrainer", dataset_loader_agent=None, **kwargs):
        super().__init__(name=name, **kwargs)
        self.dataset_loader_agent = dataset_loader_agent

        self.model, self.train_loss_list, self.val_accuracy_list, self.test_accuracy = None, None, None, None

        self.available_models = ["linear", "conv"]


    def train_model_on_query(self, dataset_name, query, override=None):
        """
        Parse the query, select the model, and train it on the dataset.
        """
        try:

            # Retrieve the dataset from the DatasetLoaderAgent
            dataset = self.dataset_loader_agent.get_dataset(dataset_name)
            if not dataset:
                return f"Dataset '{dataset_name}' is not loaded or does not exist."

            # Extract dataset components
            train_features = dataset["train_features"]
            train_targets = dataset["train_targets"]
            val_features = dataset["validation_features"]
            val_targets = dataset["validation_targets"]
            test_features = dataset["test_features"]
            test_targets = dataset["test_targets"]


            # YOUR CODE HERE
            # query = local_model_generate(prompt=
            #     f"The dataset '{dataset_name}' has been loaded. "
            #     "Short, concise description of the dataset:\n")

            response = local_model_generate(f"Based on the following dataset description, recommend a model type (linear or conv

            if "linear" in response:
                model_type = "linear"
            elif "conv" in response:
                model_type = "conv"
            else:
                model_type = "linear" # Fallback to linear if no clear recommendation


            if len(train_features.shape) == 3:
                indim = train_features.shape[1]*train_features.shape[2]
            else:
                indim = train_features.shape[1]

            # Training printing
            print(f"Training model of type '{model_type}' on dataset '{dataset_name}' with input dimension {indim}.")
            print(f"Training features shape: {train_features.shape}")
            print(f"Training targets shape: {train_targets.shape}")

            # YOUR CODE BELOW (arguments to the models)

            # Select the model based on the model_type
            if model_type == "linear" or override == "linear":
                model = fcnClassification(input_size=indim, output_size=10)
            elif model_type == "conv" or override == "conv":
                model = ConvClassification()
            else: # Fallback
                model = ConvClassification()

            # Train the model
            self.model, self.train_loss_list, self.val_accuracy_list, self.test_accuracy = train_model(
                model, train_features, train_targets, val_features, val_targets,
                test_features=test_features, test_targets=test_targets
            )

            # Summarize the training results
            response = (
                f"Model '{model_type}' trained successfully on dataset '{dataset_name}'.\n"
                f"Final validation accuracy: {self.val_accuracy_list[-1]:.4f}\n"
                f"Test accuracy: {self.test_accuracy:.4f}"
            )
            print("Training response:", response)

            result = {
                "role": "assistant",
                "status": "success",
                "model": self.model,
                "train_loss_list": self.train_loss_list,
                "val_accuracy_list": self.val_accuracy_list,
                "test_accuracy": self.test_accuracy,
                "response": response
            }
            return result

        except Exception as e:
            return {"status": "error", "response": f"An error occurred during training: {str(e)}"}
```

Train a model on mnist

- Get the dataset and train a model on it
- You can try specifying, but make sure the LLM can recommend a model type itself
- The mnist dataset is very well known, so it should decide on the convolutional model itself (up to the output being bad becasue of the small model)

```
# Create instances of the agents
dataset_loader_agent = DatasetLoaderAgent(name="DatasetLoader")
model_trainer_agent = ModelTrainerAgent(name="ModelTrainer", dataset_loader_agent=dataset_loader_agent)
local_agent = InterfaceAgent(name="LocalAgent")

local_agent.set_dataset_loader_agent(dataset_loader_agent)
local_agent.set_model_trainer_agent(model_trainer_agent)

messages = [{"role": "user", "content": "get dataset mnist"}]
response = local_agent.generate_reply(messages)
print(response["content"])
print()

# Train the model
messages.append({"role": "user", "content": "train mnist random"})
response = local_agent.generate_reply(messages)
print(response["content"])
print()

# And you now have a trained model!
print(model_trainer_agent.model)
```

```
Successfully loaded dataset 'mnist'.
Training samples: 800, Validation samples: 200, Test samples: 100, Feature dimension: 784.

Training model of type 'linear' on dataset 'mnist' with input dimension 784.
Training features shape: (800, 784)
Training targets shape: (800,)
  0%|          | 0/80 [00:00<?, ?it/s]huggingface/tokenizers: The current process just got forked, after parallelism has alr
To disable this warning, you can either:
        - Avoid using `tokenizers` before the fork if possible
        - Explicitly set the environment variable TOKENIZERS_PARALLELISM=(true | false)
100%|██████████| 80/80 [00:01<00:00, 62.37it/s]Training response: Model 'linear' trained successfully on dataset 'mnist'.
Final validation accuracy: 0.8900
Test accuracy: 0.9100
Successfully trained the model 'random'.

fcnClassification(
  (linear1): Linear(in_features=784, out_features=512, bias=True)
  (linear2): Linear(in_features=512, out_features=256, bias=True)
  (linear3): Linear(in_features=256, out_features=10, bias=True)
  (drop1): Dropout(p=0.2, inplace=False)
)
```

Repeat with solar_flare

Now that everything's working, try adding the solar_flare dataset in.

- Add solar_flare to the list of available datasets in the datasetloader
  - The backend stuff for this is already in the lab 3 code
    - This would not be too bad to fit into the agents in principle
- The other agents shouldn't need any additional modification, unless you wrote mnist-specific code
  - Ignore that the models have mnist in the name, they aren't necessarily mnist specific
  - This is easy to do without intending to. The more generalized your code, the better
  - Try throwing any errors you get to copilot as a first pass. It's good at generalizing code

- In the end, you should be able to tell the agent "train 'dataset' 'random'", and it should be able to select and train a linear model if dataset=solar_flare or a convolutional model if dataset=mnist
    - Of course, up to LLM inconsistency
    - With a good model/more complex logic, it should be able to do this for many different models and any random dataset you throw at it
        - Very useful for taking a first crack at a dataset and getting a starting point
        - This entire step could also be implemented as a "copilot, figure out how to add 'dataset'" command with a model as good as copilot hooked into this

```
# YOUR CODE HERE (just more queries to the agents)
# Create instances of the agents
dataset_loader_agent = DatasetLoaderAgent(name="DatasetLoader")
model_trainer_agent = ModelTrainerAgent(name="ModelTrainer", dataset_loader_agent=dataset_loader_agent)
local_agent = InterfaceAgent(name="LocalAgent")

local_agent.set_dataset_loader_agent(dataset_loader_agent)
local_agent.set_model_trainer_agent(model_trainer_agent)

messages = [{"role": "user", "content": "get dataset solar_flare"}]
response = local_agent.generate_reply(messages)
print(response["content"])
print()

# Train the model
messages.append({"role": "user", "content": "train solar_flare conv"})
response = local_agent.generate_reply(messages)
print(response["content"])
print()

# And you now have a trained model!
print(model_trainer_agent.model)
```

```
Successfully loaded dataset 'solar_flare'.
Training samples: 1000, Validation samples: 250, Test samples: 139, Feature dimension: 23.

Training model of type 'linear' on dataset 'solar_flare' with input dimension 23.
Training features shape: (1000, 23)
Training targets shape: (1000,)
100%|██████████| 80/80 [00:00<00:00, 84.72it/s]Training response: Model 'linear' trained successfully on dataset 'solar_flar
Final validation accuracy: 0.9880
Test accuracy: 0.9928
Successfully trained the model 'conv'.

fcnClassification(
  (linear1): Linear(in_features=23, out_features=512, bias=True)
  (linear2): Linear(in_features=512, out_features=256, bias=True)
  (linear3): Linear(in_features=256, out_features=10, bias=True)
  (drop1): Dropout(p=0.2, inplace=False)
)
```