LLM Interface for Brain-Inspired Neural Network

This document provides detailed information about the LLM (Large Language Model) interface component of the brain-inspired neural network system. The LLM interface enables the neural network to connect with external LLM API endpoints for training and validation purposes.

Overview

The LLM interface serves as a bridge between the brain-inspired neural network and various LLM providers (OpenAI, Hugging Face, Anthropic). It enables:

1. Sending model outputs to LLMs for evaluation and feedback
2. Receiving structured feedback from LLMs to guide training
3. Translating between neural network tensor representations and natural language
4. Generating training data using LLM capabilities
5. Validating model performance using LLM-based metrics

Supported LLM Providers

The interface supports multiple LLM providers through a unified API:

- **OpenAI** (GPT-3.5, GPT-4)
- **Hugging Face** (Various open-source models)
- **Anthropic** (Claude models)

Each provider is implemented as a concrete class that inherits from the `BaseLLMProvider` abstract base class, ensuring a consistent interface regardless of the underlying LLM service.

Configuration

The LLM interface is configured through the `config.yaml` file. Here's an example configuration:

```yaml
# LLM Integration settings
llm:
  # General LLM settings
  provider: "openai"  # Options: "openai", "huggingface", "anthropic"
  api_endpoint: ""  # Leave empty to use default endpoints
  model_name: "gpt-4"
  max_tokens: 1024
  temperature: 0.7
  embedding_dim: 768

  # Provider-specific settings
  openai:
```

```yaml
    api_key: ""  # Set via environment variable OPENAI_API_KEY
    model_name: "gpt-4"
    embedding_model: "text-embedding-3-small"

  huggingface:
    api_key: ""  # Set via environment variable HF_API_TOKEN
    model_name: "mistralai/Mistral-7B-Instruct-v0.2"
    embedding_model: "sentence-transformers/all-mpnet-base-v2"

  anthropic:
    api_key: ""  # Set via environment variable ANTHROPIC_API_KEY
    model_name: "claude-3-sonnet-20240229"

  # LLM validation settings
  validation:
    prompts:
      - "Explain how neural networks process information similar to the human brain."
      - "Describe the role of neuromodulators in learning and memory."
    interval: 5  # Validate every N epochs

  # LLM training settings
  training:
    use_llm_feedback: true  # Whether to use LLM feedback during training
    feedback_weight: 0.5  # Weight of LLM feedback in loss calculation
    generate_data: false  # Whether to generate training data using LLM
```

API Keys

For security reasons, API keys should be provided via environment variables rather than in the configuration file:

- OpenAI: `OPENAI_API_KEY`
- Hugging Face: `HF_API_TOKEN`
- Anthropic: `ANTHROPIC_API_KEY`

Example:

```
export OPENAI_API_KEY="your-api-key-here"
```

Core Components

LLMInterface Class

The main class that provides a unified interface to different LLM providers:

```
from src.utils.llm_interface import LLMInterface

# Initialize the interface
```

```python
llm_interface = LLMInterface(
    api_endpoint="https://api.example.com/llm",
    model_name="gpt-4",
    max_tokens=1024,
    temperature=0.7,
    provider="openai",
    api_key=None,  # Will use environment variable
    embedding_dim=768
)

# Get a response from the LLM
response = llm_interface.get_response("What is neuromodulation?")

# Get a streaming response
def process_chunk(chunk):
    print(chunk, end="", flush=True)

llm_interface.get_response(
    "Explain neural plasticity.",
    streaming=True,
    callback=process_chunk
)
```

Provider Classes

Each LLM provider is implemented as a separate class that inherits from `BaseLLMProvider`:

- `OpenAIProvider`: For OpenAI's GPT models
- `HuggingFaceProvider`: For Hugging Face's models
- `AnthropicProvider`: For Anthropic's Claude models

These classes handle the specifics of connecting to each provider's API, formatting requests, and processing responses.

Key Features

1. Translation Between Tensors and Text

The interface provides methods to convert between neural network tensor representations and natural language:

```python
# Convert LLM response to model input tensor
model_input = llm_interface.response_to_model_input(llm_response)

# Convert model output tensor to LLM prompt
output_prompt = llm_interface.model_output_to_prompt(model_output)
```

2. LLM-Based Validation

Validate model performance using LLM feedback:

```
# Run validation
avg_score, results = llm_interface.validate_with_llm(
    model, validation_prompts, device
)
```

3. Training with LLM Feedback

Incorporate LLM feedback into the training process:

```
# Train with LLM feedback
results = llm_interface.train_with_llm_feedback(
    model, inputs, targets, optimizer, loss_fn, epochs=5
)
```

4. Training Data Generation

Generate training data using LLM capabilities:

```
# Generate training data
inputs, targets = llm_interface.generate_training_data(
    prompts, sequence_length=64, output_size=64
)
```

Integration with Training Process

The LLM interface integrates with the training process in `train.py`. During training:

1. The model processes inputs and generates outputs
2. Outputs are converted to natural language prompts
3. These prompts are sent to the LLM for evaluation
4. LLM feedback is used to adjust the loss function or provide gradients
5. The model is updated based on this feedback

During validation:

1. Validation prompts are sent to the LLM
2. LLM responses are converted to model inputs
3. The model processes these inputs
4. Model outputs are evaluated by the LLM
5. Evaluation scores and feedback are collected

Example Usage

See the example script at `examples/llm_interface_example.py` for a complete demonstration of the LLM interface capabilities.

Extending the Interface

To add support for a new LLM provider:

1. Create a new class that inherits from `BaseLLMProvider`
2. Implement the required methods: `connect()`, `send_prompt()`, `send_prompt_streaming()`, and `get_embedding()`
3. Add the new provider to the `_init_provider()` method in `LLMInterface`
4. Update the configuration file to include settings for the new provider

Error Handling

The LLM interface includes robust error handling to manage API failures, rate limits, and other issues:

- Connection failures are logged with appropriate warnings
- API errors are caught and reported
- Fallback mechanisms are provided when possible (e.g., for embeddings)

Performance Considerations

- **Caching**: Consider implementing response caching for frequently used prompts
- **Batching**: Use batch processing when possible to reduce API calls
- **Rate Limiting**: Be aware of provider rate limits and implement appropriate throttling
- **Cost**: Monitor API usage to control costs, especially for commercial providers

Limitations

- LLM responses may vary between providers and even between calls to the same provider
- API costs can be significant for large-scale training or validation
- Some providers may have limitations on request frequency or token counts
- Translation between tensor representations and natural language is inherently lossy