Agentic AI Tutorial: Building Intelligent Autonomous Systems

Table of Contents

Introduction to Agentic AI

Agentic AI refers to AI systems that can act autonomously to achieve goals, make decisions, and interact with their environment. Unlike traditional AI that responds to prompts, agentic systems proactively plan, execute, and adapt their behavior.

Key Characteristics:

- **Autonomy**: Can operate without constant human supervision
- **Goal-oriented**: Pursues objectives over multiple steps
- **Adaptive**: Learns from experience and adjusts strategies
- **Interactive**: Communicates with humans and other systems
- **Persistent**: Maintains context and memory across sessions

Agent Types:

- **Reactive Agents**: Respond to immediate stimuli
- **Deliberative Agents**: Plan before acting
- **Hybrid Agents**: Combine reactive and deliberative approaches
- **Learning Agents**: Improve performance over time

Core Concepts and Architecture

Agent Architecture Components

```python
from abc import ABC, abstractmethod
from typing import Dict, List, Any, Optional
import json

class Agent(ABC):
    """Base agent architecture"""

    def __init__(self, name: str, goals: List[str]):
        self.name = name
        self.goals = goals
        self.memory = {}
        self.knowledge_base = {}
        self.tools = {}

    @abstractmethod
    def perceive(self, environment: Dict) -> Dict:
        """Perceive the current environment state"""
        pass

    @abstractmethod
    def plan(self, observations: Dict) -> List[Dict]:
        """Create action plan based on observations"""
        pass

    @abstractmethod
    def act(self, action: Dict) -> Dict:
        """Execute an action"""
        pass

    def update_memory(self, experience: Dict):
        """Update agent's memory with new experience"""
        timestamp = experience.get('timestamp', 'unknown')
        self.memory[timestamp] = experience

    def reflect(self):
        """Reflect on past experiences and update strategies"""
        # Analyze recent experiences
        recent_experiences = list(self.memory.values())[-10:]

        # Extract patterns and lessons
        successful_actions = [exp for exp in recent_experiences if exp.get('success', False)]
        failed_actions = [exp for exp in recent_experiences if not exp.get('success', True)]

        # Update knowledge base
        self.knowledge_base['successful_patterns'] = successful_actions
        self.knowledge_base['failure_patterns'] = failed_actions
```

Perception-Planning-Action Loop

```python
class AutonomousAgent(Agent):
    """Agent with perception-planning-action loop"""

    def __init__(self, name: str, goals: List[str], llm_model):
        super().__init__(name, goals)
        self.llm = llm_model

    def perceive(self, environment: Dict) -> Dict:
        """Analyze environment and extract relevant information"""
        perception_prompt = f"""
        Analyze the current environment and identify:
        1. Relevant objects, entities, or data
        2. Current state and context
        3. Opportunities for action
        4. Potential obstacles or constraints

        Environment: {environment}
        Agent Goals: {self.goals}
        """

        response = self.llm.generate(perception_prompt)
        return {"observations": response, "environment": environment}

    def plan(self, observations: Dict) -> List[Dict]:
        """Generate step-by-step action plan"""
        planning_prompt = f"""
        Given these observations and goals, create a detailed action plan:

        Observations: {observations['observations']}
        Goals: {self.goals}
        Available Tools: {list(self.tools.keys())}
        Past Experience: {self.knowledge_base}

        Provide a step-by-step plan with:
        - Action type
        - Required tools
        - Expected outcomes
        - Contingency plans
        """

        plan_response = self.llm.generate(planning_prompt)
        # Parse response into structured actions
        return self._parse_plan(plan_response)

    def act(self, action: Dict) -> Dict:
        """Execute a single action"""
        action_type = action.get('type')

        if action_type == 'tool_use':
            return self._use_tool(action)
        elif action_type == 'communication':
            return self._communicate(action)
        elif action_type == 'analysis':
            return self._analyze(action)
        else:
            return {"success": False, "error": f"Unknown action type: {action_type}"}

    def run(self, environment: Dict, max_iterations: int = 10):
        """Main agent execution loop"""
        for iteration in range(max_iterations):
            # Perceive
            observations = self.perceive(environment)
            # Plan
            actions = self.plan(observations)

            # Act
            for action in actions:
                result = self.act(action)

                # Update memory
                experience = {
                    'iteration': iteration,
                    'action': action,
                    'result': result,
                    'success': result.get('success', False),
                    'timestamp': f"iter_{iteration}"
                }
                self.update_memory(experience)

                # Check if goal achieved
                if self._goal_achieved(result):
                    return {"status": "success", "iterations": iteration + 1}

                # Update environment based on action result
                environment = self._update_environment(environment, result)

            # Reflect on progress
            if iteration % 3 == 0:  # Reflect every 3 iterations
                self.reflect()

        return {"status": "incomplete", "iterations": max_iterations}
```

Agent Frameworks

LangGraph Agent

```python
from langgraph.graph import StateGraph, END
from langchain_core.messages import BaseMessage
from typing import TypedDict, List

class AgentState(TypedDict):
    messages: List[BaseMessage]
    plan: List[str]
    current_step: int
    tools_used: List[str]
    goal_status: str

def create_langgraph_agent():
    """Create agent using LangGraph framework"""

    def planning_node(state: AgentState):
        # Generate plan based on current messages
        planner_prompt = f"Create a plan to achieve: {state['messages'][-1].content}"
        # Use LLM to generate plan
        plan = ["step1", "step2", "step3"]  # Simplified
        return {"plan": plan, "current_step": 0}

    def execution_node(state: AgentState):
        current_step = state["current_step"]
        if current_step < len(state["plan"]):
            # Execute current step
            step = state["plan"][current_step]
            # Simulate step execution
            return {
                "current_step": current_step + 1,
                "tools_used": state["tools_used"] + [f"tool_for_{step}"]
            }
        return {"goal_status": "completed"}

    def should_continue(state: AgentState):
        if state.get("goal_status") == "completed":
            return "end"
        elif state["current_step"] >= len(state.get("plan", [])):
            return "replan"
        else:
            return "execute"

    # Build graph
    workflow = StateGraph(AgentState)
    workflow.add_node("planner", planning_node)
    workflow.add_node("executor", execution_node)

    workflow.set_entry_point("planner")
    workflow.add_edge("planner", "executor")
    workflow.add_conditional_edges(
        "executor",
        should_continue,
        {
            "execute": "executor",
            "replan": "planner",
            "end": END
        }
    )

    return workflow.compile()
```

CrewAI Multi-Agent System

```
# Example CrewAI-style multi-agent setup
class CrewAgent:
    def __init__(self, role: str, goal: str, backstory: str, tools: List):
        self.role = role
        self.goal = goal
        self.backstory = backstory
        self.tools = tools
        self.memory = []

    def execute_task(self, task: str) -> str:
        # Simulate task execution
        context = f"Role: {self.role}\nGoal: {self.goal}\nTask: {task}"
        # Use LLM with tools
        result = f"Completed {task} as {self.role}"
        self.memory.append({"task": task, "result": result})
        return result

def create_research_crew():
    """Create a crew of agents for research tasks"""

    researcher = CrewAgent(
        role="Research Analyst",
        goal="Gather comprehensive information on assigned topics",
        backstory="Expert researcher with access to various data sources",
        tools=["web_search", "database_query", "document_analysis"]
    )

    writer = CrewAgent(
        role="Content Writer",
        goal="Create well-structured, engaging content",
        backstory="Experienced writer who specializes in technical content",
        tools=["text_editor", "grammar_checker", "style_guide"]
    )

    reviewer = CrewAgent(
        role="Quality Reviewer",
        goal="Ensure content meets quality standards",
        backstory="Senior editor with expertise in fact-checking",
        tools=["fact_checker", "plagiarism_detector", "quality_scorer"]
    )

    return [researcher, writer, reviewer]

def execute_crew_task(crew, task):
    """Execute task through crew collaboration"""
    results = []

    # Sequential execution
    for agent in crew:
        if results:
            # Pass previous results as context
            context_task = f"{task}\nPrevious work: {results[-1]}"
        else:
            context_task = task

        result = agent.execute_task(context_task)
        results.append(result)

    return results
```

Planning and Reasoning

Hierarchical Task Planning

```python
class TaskPlanner:
    """Hierarchical task decomposition and planning"""

    def __init__(self, llm_model):
        self.llm = llm_model
        self.task_hierarchy = {}

    def decompose_task(self, high_level_task: str) -> Dict:
        """Break down high-level task into subtasks"""
        decomposition_prompt = f"""
        Break down this high-level task into smaller, actionable subtasks:

        Task: {high_level_task}

        Provide:
        1. Subtasks in order of execution
        2. Dependencies between subtasks
        3. Success criteria for each subtask
        4. Estimated effort/time for each

        Format as structured data.
        """

        response = self.llm.generate(decomposition_prompt)
        return self._parse_task_breakdown(response)

    def create_execution_plan(self, task_breakdown: Dict) -> List[Dict]:
        """Create detailed execution plan"""
        subtasks = task_breakdown.get('subtasks', [])
        dependencies = task_breakdown.get('dependencies', {})

        # Topological sort based on dependencies
        execution_order = self._topological_sort(subtasks, dependencies)

        plan = []
        for task in execution_order:
            plan.append({
                'task': task,
                'dependencies': dependencies.get(task, []),
                'status': 'pending',
                'estimated_effort': task_breakdown.get('efforts', {}).get(task, 'medium')
            })

        return plan

    def adaptive_replanning(self, current_plan: List[Dict], execution_results: List[Dict]) -> List[Dict]:
        """Adapt plan based on execution results"""
        failed_tasks = [r for r in execution_results if not r.get('success', False)]

        if failed_tasks:
            replanning_prompt = f"""
            The following tasks failed during execution:
            {failed_tasks}

            Original plan: {current_plan}

            Please provide:
            1. Root cause analysis of failures
            2. Alternative approaches for failed tasks
            3. Updated execution plan
            4. Risk mitigation strategies
            """
            response = self.llm.generate(replanning_prompt)
            return self._parse_updated_plan(response)

        return current_plan

# Chain of Thought Reasoning
class ReasoningAgent:
    """Agent with explicit reasoning capabilities"""

    def __init__(self, llm_model):
        self.llm = llm_model
        self.reasoning_history = []

    def reason_step_by_step(self, problem: str) -> Dict:
        """Perform step-by-step reasoning"""
        reasoning_prompt = f"""
        Let's think step by step to solve this problem:

        Problem: {problem}

        Please provide:
        1. Problem understanding and key constraints
        2. Relevant facts and assumptions
        3. Step-by-step reasoning process
        4. Intermediate conclusions
        5. Final answer with confidence level

        Be explicit about your reasoning at each step.
        """

        response = self.llm.generate(reasoning_prompt)

        reasoning_result = {
            'problem': problem,
            'reasoning_steps': self._extract_reasoning_steps(response),
            'conclusion': self._extract_conclusion(response),
            'confidence': self._extract_confidence(response)
        }

        self.reasoning_history.append(reasoning_result)
        return reasoning_result

    def validate_reasoning(self, reasoning_result: Dict) -> Dict:
        """Validate reasoning for logical consistency"""
        validation_prompt = f"""
        Please review this reasoning for logical consistency and correctness:

        Problem: {reasoning_result['problem']}
        Reasoning Steps: {reasoning_result['reasoning_steps']}
        Conclusion: {reasoning_result['conclusion']}

        Check for:
        1. Logical fallacies
        2. Inconsistencies
        3. Missing steps
        4. Alternative solutions

        Provide validation report.
        """

        validation_response = self.llm.generate(validation_prompt)
        return self._parse_validation_report(validation_response)
```

Memory and Knowledge Management

Episodic and Semantic Memory

```python
import sqlite3
from datetime import datetime
import json

class AgentMemory:
    """Comprehensive memory system for agents"""

    def __init__(self, db_path: str = "agent_memory.db"):
        self.db_path = db_path
        self.init_database()
        self.working_memory = {}  # Short-term memory
        self.episodic_memory = []  # Experience memory
        self.semantic_memory = {}  # Knowledge memory

    def init_database(self):
        """Initialize memory database"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        # Episodic memory table
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS episodes (
                id INTEGER PRIMARY KEY,
                timestamp TEXT,
                context TEXT,
                action TEXT,
                result TEXT,
                success BOOLEAN,
                importance REAL
            )
        ''')

        # Semantic memory table
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS knowledge (
                id INTEGER PRIMARY KEY,
                concept TEXT,
                description TEXT,
                confidence REAL,
                sources TEXT,
                last_updated TEXT
            )
        ''')

        conn.commit()
        conn.close()

    def store_episode(self, context: str, action: str, result: str, success: bool, importance: float = 0.5):
        """Store episodic memory"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        cursor.execute('''
            INSERT INTO episodes (timestamp, context, action, result, success, importance)
            VALUES (?, ?, ?, ?, ?, ?)
        ''', (datetime.now().isoformat(), context, action, result, success, importance))

        conn.commit()
        conn.close()

        # Also store in working memory
        self.episodic_memory.append({
            'timestamp': datetime.now().isoformat(),
            'context': context,
            'action': action,
            'result': result,
            'success': success,
            'importance': importance
        })

    def retrieve_similar_episodes(self, current_context: str, limit: int = 5) -> List[Dict]:
        """Retrieve similar past episodes"""
        # Simplified similarity (in practice, use embeddings)
        similar_episodes = []

        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        cursor.execute('''
            SELECT * FROM episodes
            WHERE context LIKE ?
            ORDER BY importance DESC, timestamp DESC
            LIMIT ?
        ''', (f'%{current_context}%', limit))

        episodes = cursor.fetchall()
        conn.close()

        return [dict(zip(['id', 'timestamp', 'context', 'action', 'result', 'success', 'importance'], ep)) for ep in episodes]

    def update_knowledge(self, concept: str, description: str, confidence: float, sources: List[str]):
        """Update semantic knowledge"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        # Check if concept exists
        cursor.execute('SELECT id FROM knowledge WHERE concept = ?', (concept,))
        existing = cursor.fetchone()

        sources_json = json.dumps(sources)

        if existing:
            cursor.execute('''
                UPDATE knowledge
                SET description = ?, confidence = ?, sources = ?, last_updated = ?
                WHERE concept = ?
            ''', (description, confidence, sources_json, datetime.now().isoformat(), concept))
        else:
            cursor.execute('''
                INSERT INTO knowledge (concept, description, confidence, sources, last_updated)
                VALUES (?, ?, ?, ?, ?)
            ''', (concept, description, confidence, sources_json, datetime.now().isoformat()))

        conn.commit()
        conn.close()

    def get_knowledge(self, concept: str) -> Optional[Dict]:
        """Retrieve knowledge about a concept"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        cursor.execute('SELECT * FROM knowledge WHERE concept = ?', (concept,))
        result = cursor.fetchone()
        conn.close()
        if result:
            return dict(zip(['id', 'concept', 'description', 'confidence', 'sources', 'last_updated'], result))
        return None

    def consolidate_memory(self):
        """Consolidate working memory into long-term storage"""
        # Move important experiences from working memory to database
        # Implement memory consolidation algorithms
        pass
```

Tool Use and Actions

Tool Integration Framework

```python
from typing import Callable, Any
import inspect
import json

class ToolRegistry:
    """Registry for agent tools and actions"""

    def __init__(self):
        self.tools = {}
        self.tool_descriptions = {}

    def register_tool(self, name: str, func: Callable, description: str = ""):
        """Register a new tool"""
        # Get function signature for automatic parameter extraction
        sig = inspect.signature(func)
        parameters = {}

        for param_name, param in sig.parameters.items():
            parameters[param_name] = {
                'type': param.annotation.__name__ if param.annotation != inspect.Parameter.empty else 'Any',
                'default': param.default if param.default != inspect.Parameter.empty else None
            }

        self.tools[name] = func
        self.tool_descriptions[name] = {
            'description': description,
            'parameters': parameters,
            'function': func
        }

    def get_tool_list(self) -> List[Dict]:
        """Get list of available tools with descriptions"""
        return [
            {
                'name': name,
                'description': info['description'],
                'parameters': info['parameters']
            }
            for name, info in self.tool_descriptions.items()
        ]

    def execute_tool(self, tool_name: str, **kwargs) -> Dict:
        """Execute a tool with given parameters"""
        if tool_name not in self.tools:
            return {'success': False, 'error': f'Tool {tool_name} not found'}

        try:
            result = self.tools[tool_name](**kwargs)
            return {'success': True, 'result': result}
        except Exception as e:
            return {'success': False, 'error': str(e)}

# Example tools
def web_search(query: str, max_results: int = 5) -> List[Dict]:
    """Search the web for information"""
    # Simulated web search
    return [
        {'title': f'Result {i+1} for {query}', 'url': f'https://example{i+1}.com', 'snippet': f'Information about {query}'}
        for i in range(max_results)
    ]

def calculator(expression: str) -> float:
    """Perform mathematical calculations"""
    try:
        # Safe evaluation (use ast.literal_eval in production)
        return eval(expression)
    except:
        raise ValueError(f"Invalid expression: {expression}")

def file_writer(filename: str, content: str) -> bool:
    """Write content to a file"""
    try:
        with open(filename, 'w') as f:
            f.write(content)
        return True
    except Exception as e:
        raise IOError(f"Failed to write file: {e}")

# Tool-using agent
class ToolUsingAgent(AutonomousAgent):
    """Agent that can use external tools"""

    def __init__(self, name: str, goals: List[str], llm_model, tool_registry: ToolRegistry):
        super().__init__(name, goals, llm_model)
        self.tool_registry = tool_registry

    def select_tool(self, task_description: str) -> str:
        """Select appropriate tool for a task"""
        available_tools = self.tool_registry.get_tool_list()

        tool_selection_prompt = f"""
        Select the most appropriate tool for this task:

        Task: {task_description}

        Available tools:
        {json.dumps(available_tools, indent=2)}

        Respond with just the tool name.
        """

        tool_name = self.llm.generate(tool_selection_prompt).strip()
        return tool_name

    def generate_tool_parameters(self, tool_name: str, task_description: str) -> Dict:
        """Generate parameters for tool execution"""
        tool_info = self.tool_registry.tool_descriptions.get(tool_name, {})

        param_generation_prompt = f"""
        Generate appropriate parameters for this tool:

        Tool: {tool_name}
        Task: {task_description}
        Required parameters: {tool_info.get('parameters', {})}

        Respond with JSON object containing parameter values.
        """

        params_response = self.llm.generate(param_generation_prompt)
        try:
            return json.loads(params_response)
        except:
            return {}

    def use_tool(self, task_description: str) -> Dict:
        """Use appropriate tool to complete a task"""
        # Select tool
        tool_name = self.select_tool(task_description)

        # Generate parameters
        parameters = self.generate_tool_parameters(tool_name, task_description)

        # Execute tool
        result = self.tool_registry.execute_tool(tool_name, **parameters)

        # Store experience
        self.store_tool_experience(task_description, tool_name, parameters, result)

        return result

    def store_tool_experience(self, task: str, tool: str, params: Dict, result: Dict):
        """Store tool usage experience for learning"""
        experience = {
            'task': task,
            'tool_used': tool,
            'parameters': params,
            'result': result,
            'success': result.get('success', False)
        }

        # Store in memory for future tool selection
        if 'tool_experiences' not in self.knowledge_base:
            self.knowledge_base['tool_experiences'] = []

        self.knowledge_base['tool_experiences'].append(experience)
```

Best Practices

Error Handling and Recovery

```
class RobustAgent(AutonomousAgent):
    """Agent with robust error handling"""

    def __init__(self, name: str, goals: List[str], llm_model):
        super().__init__(name, goals, llm_model)
        self.max_retries = 3
        self.fallback_strategies = {}

    def robust_execute(self, action: Dict) -> Dict:
        """Execute action with error handling and retries"""
        for attempt in range(self.max_retries):
            try:
                result = self.act(action)

                if result.get('success', False):
                    return result
                else:
                    # Try fallback strategy
                    if action['type'] in self.fallback_strategies:
                        fallback_action = self.fallback_strategies[action['type']]
                        return self.act(fallback_action)

            except Exception as e:
                if attempt == self.max_retries - 1:
                    return {'success': False, 'error': f'Max retries exceeded: {e}'}

                # Exponential backoff
                time.sleep(2 ** attempt)

        return {'success': False, 'error': 'All attempts failed'}

    def add_fallback_strategy(self, action_type: str, fallback_action: Dict):
        """Add fallback strategy for specific action types"""
        self.fallback_strategies[action_type] = fallback_action

# Safety and Alignment
class SafeAgent(RobustAgent):
    """Agent with safety constraints and alignment checks"""

    def __init__(self, name: str, goals: List[str], llm_model, safety_rules: List[str]):
        super().__init__(name, goals, llm_model)
        self.safety_rules = safety_rules
        self.violation_log = []

    def check_safety(self, action: Dict) -> bool:
        """Check if action violates safety rules"""
        safety_check_prompt = f"""
        Check if this action violates any safety rules:

        Action: {action}
        Safety Rules: {self.safety_rules}

        Respond with 'SAFE' or 'UNSAFE' and explanation.
        """

        response = self.llm.generate(safety_check_prompt)

        if 'UNSAFE' in response.upper():
            self.violation_log.append({
                'action': action,
                'violation_reason': response,
                'timestamp': datetime.now().isoformat()
            })
            return False

        return True

    def act(self, action: Dict) -> Dict:
        """Execute action with safety checks"""
        if not self.check_safety(action):
            return {
                'success': False,
                'error': 'Action violates safety rules',
                'action': action
            }

        return super().act(action)

# Performance Monitoring
class MonitoredAgent(SafeAgent):
    """Agent with performance monitoring"""

    def __init__(self, name: str, goals: List[str], llm_model, safety_rules: List[str]):
        super().__init__(name, goals, llm_model, safety_rules)
        self.performance_metrics = {
            'total_actions': 0,
            'successful_actions': 0,
            'failed_actions': 0,
            'average_response_time': 0,
            'goal_completion_rate': 0
        }

    def update_metrics(self, action_result: Dict, response_time: float):
        """Update performance metrics"""
        self.performance_metrics['total_actions'] += 1

        if action_result.get('success', False):
            self.performance_metrics['successful_actions'] += 1
        else:
            self.performance_metrics['failed_actions'] += 1

        # Update average response time
        total_actions = self.performance_metrics['total_actions']
        current_avg = self.performance_metrics['average_response_time']
        self.performance_metrics['average_response_time'] = (
            (current_avg * (total_actions - 1) + response_time) / total_actions
        )

    def get_performance_report(self) -> Dict:
        """Generate performance report"""
        total = self.performance_metrics['total_actions']
        if total == 0:
            return self.performance_metrics

        success_rate = self.performance_metrics['successful_actions'] / total
        failure_rate = self.performance_metrics['failed_actions'] / total

        return {
            **self.performance_metrics,
            'success_rate': success_rate,
            'failure_rate': failure_rate
        }
```

Conclusion

Agentic AI represents a paradigm shift toward autonomous, goal-oriented AI systems. This tutorial covered:

- Core agent architectures and design patterns
- Planning, reasoning, and decision-making capabilities
- Memory systems for learning and adaptation
- Tool integration and multi-agent collaboration
- Safety, robustness, and performance monitoring

Key Principles:

1. **Goal-oriented design**: Agents should have clear objectives
2. **Modular architecture**: Separate perception, planning, and action
3. **Memory integration**: Learn from experience
4. **Safety first**: Implement constraints and monitoring
5. **Adaptive behavior**: Adjust strategies based on outcomes

Implementation Strategy:

1. Start with simple reactive agents
2. Add planning and reasoning capabilities
3. Integrate memory and learning systems
4. Implement multi-agent coordination
5. Deploy with safety measures and monitoring

Next Steps:

- Explore advanced planning algorithms
- Implement multi-modal agents (vision, speech, etc.)
- Study reinforcement learning for agent training
- Build domain-specific agent applications
- Contribute to open-source agent frameworks

The future of AI lies in systems that can act autonomously while remaining aligned with human values and goals.

Hugging Face Tutorial: Complete Guide to Transformers and Model Hub

Table of Contents

Introduction to Hugging Face

Hugging Face is the leading platform for machine learning models, providing:

- □ **Transformers**: State-of-the-art ML models for PyTorch, TensorFlow, and JAX
- □ **Datasets**: The largest collection of ready-to-use datasets
- □ **Model Hub**: Over 300,000+ models shared by the community

- □ **Spaces**: Collaborative platform for ML demos and applications

Key Ecosystems:

- **NLP**: BERT, GPT, RoBERTa, T5, and more
- **Computer Vision**: Vision Transformer, CLIP, DETR
- **Audio**: Wav2Vec2, Whisper, SpeechT5
- **Multimodal**: CLIP, DALL-E, Flamingo
- **Reinforcement Learning**: Decision Transformers

Installation and Setup

Core Installation

```
# Basic installation
pip install transformers

# With PyTorch
pip install transformers[torch]

# With TensorFlow
pip install transformers[tf]

# Full installation with all dependencies
pip install transformers[all]

# Additional libraries
pip install datasets
pip install tokenizers
pip install accelerate
pip install peft  # For parameter-efficient fine-tuning
pip install bitsandbytes  # For quantization
```

Environment Setup

```
import torch
from transformers import (
    AutoTokenizer,
    AutoModel,
    AutoModelForSequenceClassification,
    pipeline,
    TrainingArguments,
    Trainer
)
from datasets import Dataset, load_dataset
import numpy as np

# Check if CUDA is available
print(f"CUDA available: {torch.cuda.is_available()}")
print(f"Device: {torch.cuda.get_device_name() if torch.cuda.is_available() else 'CPU'}")

# Set device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Hugging Face Hub Authentication

```python
from huggingface_hub import login, HfApi
import os

# Method 1: Login interactively
login()

# Method 2: Use token from environment
os.environ["HUGGINGFACE_HUB_TOKEN"] = "your_token_here"

# Method 3: Programmatic login
login(token="your_token_here")

# Check login status
api = HfApi()
user = api.whoami()
print(f"Logged in as: {user['name']}")
```

Using Pre-trained Models

Quick Start with Pipelines

```python
# Sentiment Analysis
sentiment_pipeline = pipeline("sentiment-analysis")
result = sentiment_pipeline("I love Hugging Face!")
print(result)  # [{'label': 'POSITIVE', 'score': 0.9998}]

# Text Generation
generator = pipeline("text-generation", model="gpt2")
output = generator("The future of AI is", max_length=50, num_return_sequences=1)
print(output[0]['generated_text'])

# Question Answering
qa_pipeline = pipeline("question-answering")
context = "Hugging Face is a company that democratizes AI through open-source and open science."
question = "What does Hugging Face do?"
answer = qa_pipeline(question=question, context=context)
print(answer)

# Named Entity Recognition
ner_pipeline = pipeline("ner", aggregation_strategy="simple")
text = "My name is Sarah and I work at Google in California."
entities = ner_pipeline(text)
print(entities)

# Translation
translator = pipeline("translation", model="Helsinki-NLP/opus-mt-en-fr")
french_text = translator("Hello, how are you?")
print(french_text)
```

Available Pipeline Tasks

```
# Get all available tasks
from transformers import PIPELINE_REGISTRY
print("Available tasks:", list(PIPELINE_REGISTRY.supported_tasks.keys()))

# Specific pipeline examples
pipelines_examples = {
    "text-classification": "distilbert-base-uncased-finetuned-sst-2-english",
    "token-classification": "dbmdz/bert-large-cased-finetuned-conll03-english",
    "question-answering": "distilbert-base-cased-distilled-squad",
    "fill-mask": "bert-base-uncased",
    "summarization": "facebook/bart-large-cnn",
    "translation": "t5-base",
    "text-generation": "gpt2",
    "text2text-generation": "t5-small",
    "zero-shot-classification": "facebook/bart-large-mnli",
    "image-classification": "google/vit-base-patch16-224",
    "object-detection": "facebook/detr-resnet-50",
    "image-segmentation": "facebook/detr-resnet-50-panoptic",
    "automatic-speech-recognition": "facebook/wav2vec2-base-960h",
    "text-to-speech": "microsoft/speecht5_tts"
}

# Use any pipeline
for task, model_name in pipelines_examples.items():
    try:
        pipe = pipeline(task, model=model_name)
        print(f"✓ {task}: {model_name}")
    except Exception as e:
        print(f"✗ {task}: {e}")
```

Transformers Library

Loading Models and Tokenizers

```
# Method 1: Auto classes (recommended)
model_name = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModel.from_pretrained(model_name)

# Method 2: Specific model classes
from transformers import BertTokenizer, BertModel
tokenizer = BertTokenizer.from_pretrained(model_name)
model = BertModel.from_pretrained(model_name)

# Load model for specific tasks
model = AutoModelForSequenceClassification.from_pretrained(
    "cardiffnlp/twitter-roberta-base-sentiment-latest"
)

# Load with specific configurations
from transformers import AutoConfig
config = AutoConfig.from_pretrained(model_name)
config.output_hidden_states = True
model = AutoModel.from_pretrained(model_name, config=config)
```

Text Processing

```python
def process_text_with_bert(text, model_name="bert-base-uncased"):
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = AutoModel.from_pretrained(model_name)

    # Tokenize
    inputs = tokenizer(
        text,
        padding=True,
        truncation=True,
        max_length=512,
        return_tensors="pt"
    )

    # Get model outputs
    with torch.no_grad():
        outputs = model(**inputs)

    # Extract embeddings
    last_hidden_states = outputs.last_hidden_state
    pooled_output = outputs.pooler_output if hasattr(outputs, 'pooler_output') else None

    return {
        "input_ids": inputs["input_ids"],
        "attention_mask": inputs["attention_mask"],
        "last_hidden_states": last_hidden_states,
        "pooled_output": pooled_output,
        "tokens": tokenizer.convert_ids_to_tokens(inputs["input_ids"][0])
    }

# Usage
text = "Hugging Face transformers are amazing!"
result = process_text_with_bert(text)
print(f"Sequence length: {result['last_hidden_states'].shape[1]}")
print(f"Hidden size: {result['last_hidden_states'].shape[2]}")
```

Batch Processing

```python
def batch_encode_texts(texts, model_name="bert-base-uncased", batch_size=16):
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = AutoModel.from_pretrained(model_name)

    all_embeddings = []

    for i in range(0, len(texts), batch_size):
        batch_texts = texts[i:i+batch_size]

        # Tokenize batch
        inputs = tokenizer(
            batch_texts,
            padding=True,
            truncation=True,
            max_length=512,
            return_tensors="pt"
        )

        # Get embeddings
        with torch.no_grad():
            outputs = model(**inputs)
            # Use mean pooling for sentence embeddings
            embeddings = outputs.last_hidden_state.mean(dim=1)
            all_embeddings.append(embeddings)

    return torch.cat(all_embeddings, dim=0)

# Usage
texts = [
    "I love machine learning",
    "Natural language processing is fascinating",
    "Transformers revolutionized NLP",
    "BERT is a powerful model"
]

embeddings = batch_encode_texts(texts)
print(f"Embeddings shape: {embeddings.shape}")
```

Model Comparison

```
def compare_models(text, models):
    results = {}

    for model_name in models:
        try:
            # Load model and tokenizer
            tokenizer = AutoTokenizer.from_pretrained(model_name)
            model = AutoModel.from_pretrained(model_name)

            # Process text
            inputs = tokenizer(text, return_tensors="pt", truncation=True, max_length=512)

            with torch.no_grad():
                outputs = model(**inputs)
                embedding = outputs.last_hidden_state.mean(dim=1)

            results[model_name] = {
                "embedding_dim": embedding.shape[1],
                "vocab_size": len(tokenizer),
                "max_position": tokenizer.model_max_length,
                "embedding_norm": torch.norm(embedding).item()
            }
        except Exception as e:
            results[model_name] = {"error": str(e)}

    return results

# Compare different models
models_to_compare = [
    "bert-base-uncased",
    "roberta-base",
    "distilbert-base-uncased",
    "albert-base-v2"
]

comparison = compare_models("This is a test sentence.", models_to_compare)
for model, stats in comparison.items():
    print(f"{model}:")
    for key, value in stats.items():
        print(f"  {key}: {value}")
    print()
```

Datasets Library

Loading Datasets

```
from datasets import load_dataset, Dataset, DatasetDict

# Load popular datasets
imdb = load_dataset("imdb")
squad = load_dataset("squad")
glue_sst2 = load_dataset("glue", "sst2")

# Load specific splits
train_data = load_dataset("imdb", split="train")
test_data = load_dataset("imdb", split="test[:1000]")  # First 1000 examples

# Load from local files
local_dataset = load_dataset("csv", data_files="my_data.csv")
json_dataset = load_dataset("json", data_files="my_data.jsonl")

print(f"IMDB dataset: {imdb}")
print(f"Features: {imdb['train'].features}")
print(f"Number of examples: {len(imdb['train'])}")
```

Dataset Exploration

```
def explore_dataset(dataset):
    """Explore dataset characteristics"""
    print(f"Dataset: {dataset}")
    print(f"Splits: {list(dataset.keys())}")

    for split_name, split_data in dataset.items():
        print(f"\n{split_name.upper()} SPLIT:")
        print(f"  Size: {len(split_data)}")
        print(f"  Features: {split_data.features}")

        # Show first few examples
        print(f"  First example: {split_data[0]}")

        # Show data types and statistics
        for feature_name, feature_type in split_data.features.items():
            print(f"  {feature_name}: {feature_type}")

# Explore IMDB dataset
explore_dataset(imdb)
```

Data Preprocessing

```
def preprocess_imdb_data(examples, tokenizer, max_length=512):
    """Preprocess IMDB dataset for BERT"""
    return tokenizer(
        examples["text"],
        truncation=True,
        padding="max_length",
        max_length=max_length,
        return_tensors="pt"
    )

# Load tokenizer
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")

# Preprocess dataset
def tokenize_function(examples):
    return tokenizer(examples["text"], truncation=True, padding="max_length", max_length=256)

# Apply preprocessing
tokenized_imdb = imdb.map(tokenize_function, batched=True)

# Remove original text column and rename label
tokenized_imdb = tokenized_imdb.remove_columns(["text"])
tokenized_imdb = tokenized_imdb.rename_column("label", "labels")

# Set format for PyTorch
tokenized_imdb.set_format("torch", columns=["input_ids", "attention_mask", "labels"])

print("Preprocessed dataset:")
print(tokenized_imdb["train"][0])
```

Custom Dataset Creation

```
def create_custom_dataset():
    """Create a custom dataset"""
    # Sample data
    data = {
        "text": [
            "I love this movie!",
            "This film is terrible.",
            "Great acting and storyline.",
            "Boring and predictable.",
            "Amazing cinematography!"
        ],
        "label": [1, 0, 1, 0, 1]  # 1=positive, 0=negative
    }

    # Create dataset
    dataset = Dataset.from_dict(data)

    # Split into train/test
    train_test = dataset.train_test_split(test_size=0.2)

    return train_test

# Create and use custom dataset
custom_data = create_custom_dataset()
print(custom_data)

# Add preprocessing
def preprocess_custom(examples):
    tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
    return tokenizer(examples["text"], truncation=True, padding="max_length", max_length=128)

custom_data = custom_data.map(preprocess_custom, batched=True)
custom_data.set_format("torch", columns=["input_ids", "attention_mask", "label"])
```

Data Augmentation

```python
def augment_text_data(dataset, augmentation_factor=2):
    """Simple data augmentation for text"""
    import random

    def augment_example(example):
        text = example["text"]

        # Simple augmentations
        augmented_texts = [text]  # Original

        # Synonym replacement (simplified)
        synonyms = {
            "good": ["great", "excellent", "wonderful"],
            "bad": ["terrible", "awful", "horrible"],
            "nice": ["pleasant", "lovely", "delightful"]
        }

        for word, syns in synonyms.items():
            if word in text.lower():
                for syn in syns:
                    augmented_texts.append(text.lower().replace(word, syn))

        # Return multiple examples
        return {
            "text": augmented_texts[:augmentation_factor],
            "label": [example["label"]] * min(len(augmented_texts), augmentation_factor)
        }

    # Apply augmentation
    augmented = dataset.map(augment_example, remove_columns=dataset.column_names, batched=False)

    return augmented
```

Tokenizers

Understanding Tokenization

```python
from transformers import AutoTokenizer

def analyze_tokenization(text, model_names):
    """Analyze how different tokenizers handle the same text"""
    print(f"Original text: '{text}'\n")

    for model_name in model_names:
        tokenizer = AutoTokenizer.from_pretrained(model_name)

        # Tokenize
        tokens = tokenizer.tokenize(text)
        token_ids = tokenizer.encode(text)
        decoded = tokenizer.decode(token_ids)

        print(f"Model: {model_name}")
        print(f"  Tokens: {tokens}")
        print(f"  Token IDs: {token_ids}")
        print(f"  Decoded: '{decoded}'")
        print(f"  Vocab size: {tokenizer.vocab_size}")
        print(f"  Number of tokens: {len(tokens)}")
        print()

# Compare different tokenizers
text = "Hello, I'm using Hugging Face transformers!"
models = [
    "bert-base-uncased",
    "gpt2",
    "roberta-base",
    "t5-base"
]

analyze_tokenization(text, models)
```

Custom Tokenization

```python
def custom_tokenization_pipeline(texts, model_name="bert-base-uncased"):
    """Custom tokenization with special handling"""
    tokenizer = AutoTokenizer.from_pretrained(model_name)

    results = []
    for text in texts:
        # Basic tokenization
        basic_tokens = tokenizer(text)

        # Add special tokens handling
        encoded = tokenizer(
            text,
            add_special_tokens=True,
            padding="max_length",
            truncation=True,
            max_length=128,
            return_tensors="pt",
            return_attention_mask=True,
            return_token_type_ids=True if "bert" in model_name else False
        )

        # Token analysis
        tokens = tokenizer.convert_ids_to_tokens(encoded["input_ids"][0])

        results.append({
            "original_text": text,
            "tokens": tokens,
            "input_ids": encoded["input_ids"],
            "attention_mask": encoded["attention_mask"],
            "special_tokens": {
                "CLS": tokenizer.cls_token,
                "SEP": tokenizer.sep_token,
                "PAD": tokenizer.pad_token,
                "UNK": tokenizer.unk_token
            }
        })

    return results

# Usage
texts = [
    "Short text",
    "This is a much longer text that might need truncation depending on the model's maximum sequence length",
    "Text with special characters: @#$%!"
]

tokenization_results = custom_tokenization_pipeline(texts)
for i, result in enumerate(tokenization_results):
    print(f"Example {i+1}:")
    print(f"  Original: {result['original_text']}")
    print(f"  Tokens: {result['tokens'][:10]}...")  # First 10 tokens
    print(f"  Length: {len(result['tokens'])}")
    print()
```

Fast Tokenizers

```
from transformers import AutoTokenizer
import time

def compare_tokenizer_speed(texts, model_name="bert-base-uncased"):
    """Compare fast vs slow tokenizer performance"""

    # Load both versions
    fast_tokenizer = AutoTokenizer.from_pretrained(model_name, use_fast=True)
    slow_tokenizer = AutoTokenizer.from_pretrained(model_name, use_fast=False)

    # Benchmark fast tokenizer
    start_time = time.time()
    fast_results = fast_tokenizer(texts, padding=True, truncation=True, return_tensors="pt")
    fast_time = time.time() - start_time

    # Benchmark slow tokenizer
    start_time = time.time()
    slow_results = slow_tokenizer(texts, padding=True, truncation=True, return_tensors="pt")
    slow_time = time.time() - start_time

    print(f"Fast tokenizer: {fast_time:.4f} seconds")
    print(f"Slow tokenizer: {slow_time:.4f} seconds")
    print(f"Speedup: {slow_time/fast_time:.2f}x")

    return fast_results, slow_results

# Test with many texts
large_texts = ["This is a test sentence."] * 1000
fast_result, slow_result = compare_tokenizer_speed(large_texts)
```

Fine-tuning Models

Basic Fine-tuning Setup

```python
from transformers import TrainingArguments, Trainer
from sklearn.metrics import accuracy_score, precision_recall_fscore_support

def setup_fine_tuning(model_name, num_labels):
    """Setup model and tokenizer for fine-tuning"""
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = AutoModelForSequenceClassification.from_pretrained(
        model_name,
        num_labels=num_labels
    )

    # Add padding token if needed
    if tokenizer.pad_token is None:
        tokenizer.pad_token = tokenizer.eos_token
        model.config.pad_token_id = tokenizer.eos_token_id

    return model, tokenizer

# Define compute metrics function
def compute_metrics(eval_pred):
    predictions, labels = eval_pred
    predictions = predictions.argmax(axis=-1)

    precision, recall, f1, _ = precision_recall_fscore_support(labels, predictions, average='weighted')
    accuracy = accuracy_score(labels, predictions)

    return {
        'accuracy': accuracy,
        'f1': f1,
        'precision': precision,
        'recall': recall
    }
```

Text Classification Fine-tuning

```
def fine_tune_text_classifier():
    """Fine-tune BERT for text classification"""

    # Load and preprocess data
    dataset = load_dataset("imdb")
    model, tokenizer = setup_fine_tuning("bert-base-uncased", num_labels=2)

    def preprocess_function(examples):
        return tokenizer(examples["text"], truncation=True, padding="max_length", max_length=256)

    # Preprocess datasets
    encoded_dataset = dataset.map(preprocess_function, batched=True)
    encoded_dataset = encoded_dataset.remove_columns(["text"])
    encoded_dataset = encoded_dataset.rename_column("label", "labels")
    encoded_dataset.set_format("torch")

    # Use smaller subset for demo
    train_dataset = encoded_dataset["train"].shuffle().select(range(1000))
    eval_dataset = encoded_dataset["test"].shuffle().select(range(200))

    # Training arguments
    training_args = TrainingArguments(
        output_dir="./results",
        num_train_epochs=3,
        per_device_train_batch_size=16,
        per_device_eval_batch_size=16,
        warmup_steps=500,
        weight_decay=0.01,
        logging_dir="./logs",
        logging_steps=10,
        evaluation_strategy="epoch",
        save_strategy="epoch",
        load_best_model_at_end=True,
        metric_for_best_model="accuracy",
        greater_is_better=True,
    )

    # Initialize trainer
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=eval_dataset,
        tokenizer=tokenizer,
        compute_metrics=compute_metrics,
    )

    # Train
    trainer.train()

    # Evaluate
    eval_results = trainer.evaluate()
    print(f"Evaluation results: {eval_results}")

    return model, tokenizer, trainer

# Run fine-tuning (commented out for demo)
# model, tokenizer, trainer = fine_tune_text_classifier()
```

Parameter-Efficient Fine-tuning (LoRA)

```
from peft import get_peft_model, LoraConfig, TaskType

def setup_lora_fine_tuning(model_name, num_labels):
    """Setup LoRA fine-tuning for efficient training"""

    # Load base model
    model = AutoModelForSequenceClassification.from_pretrained(
        model_name,
        num_labels=num_labels
    )

    # LoRA configuration
    peft_config = LoraConfig(
        task_type=TaskType.SEQ_CLS,  # Sequence classification
        inference_mode=False,
        r=8,                         # Rank
        lora_alpha=32,               # LoRA scaling parameter
        lora_dropout=0.1,            # LoRA dropout
        target_modules=["query", "value"],  # Target attention modules
    )

    # Apply LoRA
    model = get_peft_model(model, peft_config)
    model.print_trainable_parameters()

    return model

# Usage
lora_model = setup_lora_fine_tuning("bert-base-uncased", num_labels=2)
```

Custom Training Loop

```
def custom_training_loop(model, tokenizer, train_dataloader, eval_dataloader, num_epochs=3):
    """Custom training loop with more control"""
    from torch.optim import AdamW
    from transformers import get_linear_schedule_with_warmup

    # Optimizer and scheduler
    optimizer = AdamW(model.parameters(), lr=2e-5)
    total_steps = len(train_dataloader) * num_epochs
    scheduler = get_linear_schedule_with_warmup(
        optimizer,
        num_warmup_steps=0,
        num_training_steps=total_steps
    )

    # Training loop
    model.train()
    for epoch in range(num_epochs):
        total_loss = 0

        for batch_idx, batch in enumerate(train_dataloader):
            # Forward pass
            outputs = model(
                input_ids=batch["input_ids"],
                attention_mask=batch["attention_mask"],
                labels=batch["labels"]
            )

            loss = outputs.loss
            total_loss += loss.item()

            # Backward pass
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

            optimizer.step()
            scheduler.step()
            optimizer.zero_grad()

            if batch_idx % 50 == 0:
                print(f"Epoch {epoch}, Batch {batch_idx}, Loss: {loss.item():.4f}")

        avg_loss = total_loss / len(train_dataloader)
        print(f"Epoch {epoch} completed. Average loss: {avg_loss:.4f}")

        # Evaluation
        model.eval()
        eval_accuracy = evaluate_model(model, eval_dataloader)
        print(f"Evaluation accuracy: {eval_accuracy:.4f}")
        model.train()

def evaluate_model(model, dataloader):
    """Evaluate model accuracy"""
    correct = 0
    total = 0

    with torch.no_grad():
        for batch in dataloader:
            outputs = model(
                input_ids=batch["input_ids"],
                attention_mask=batch["attention_mask"]
            )
            predictions = torch.argmax(outputs.logits, dim=-1)
            correct += (predictions == batch["labels"]).sum().item()
            total += batch["labels"].size(0)

    return correct / total
```

Model Hub and Sharing

Exploring the Hub

```
from huggingface_hub import HfApi, list_models, model_info

api = HfApi()

# List models with filters
models = list_models(
    task="text-classification",
    library="transformers",
    language="en",
    sort="downloads",
    limit=10
)

print("Top 10 English text classification models:")
for model in models:
    print(f"- {model.id} ({model.downloads} downloads)")

# Get detailed model information
model_details = model_info("bert-base-uncased")
print(f"\nModel: {model_details.id}")
print(f"Library: {model_details.library_name}")
print(f"Downloads: {model_details.downloads}")
print(f"Tags: {model_details.tags}")
```

Uploading Models

~~~
from huggingface_hub import Repository, upload_folder

def upload_model_to_hub(model, tokenizer, repo_name, commit_message="Upload model"):
    """Upload trained model to Hugging Face Hub"""

    # Save model locally first
    model.save_pretrained(f"./{repo_name}")
    tokenizer.save_pretrained(f"./{repo_name}")

    # Create model card
    model_card_content = f"""
---
language: en
license: apache-2.0
tags:
- text-classification
- sentiment-analysis
datasets:
- imdb
metrics:
- accuracy
---

# {repo_name}

This model is a fine-tuned version of BERT for sentiment analysis on the IMDB dataset.

## Usage

```python
from transformers import AutoTokenizer, AutoModelForSequenceClassification

tokenizer = AutoTokenizer.from_pretrained("your-username/{repo_name}")
model = AutoModelForSequenceClassification.from_pretrained("your-username/{repo_name}")

# Use the model
inputs = tokenizer("I love this movie!", return_tensors="pt")
outputs = model(**inputs)
predictions = torch.nn.functional.softmax(outputs.logits, dim=-1)
~~~

Training Data

The model was trained on the IMDB movie reviews dataset.

Training Procedure

- Learning rate: 2e-5
- Batch size: 16
- Number of epochs: 3

Evaluation Results

- Accuracy: 92.5%

- F1-score: 0.925 """

  with open(f"./README.md", "w") as f: f.write(model_card_content)

Upload to hub

upload_folder(folder_path=f"./", repo_id=f"your-username/", repo_type="model", commit_message=commit_message)

Usage (example)

upload_model_to_hub(fine_tuned_model, tokenizer, "my-sentiment-model")

~~~
### Model Versioning and Management
```python
def manage_model_versions(repo_id):
    """Manage different versions of a model"""
    api = HfApi()

    # List all commits (versions)
    commits = api.list_repo_commits(repo_id)
    print(f"Model versions for {repo_id}:")
    for commit in commits[:5]:  # Show last 5 versions
        print(f"- {commit.commit_id[:8]}: {commit.title}")

    # Load specific version
    specific_version_model = AutoModel.from_pretrained(
        repo_id,
        revision=commits[1].commit_id  # Load second-to-last version
    )

    return specific_version_model

# Usage
# model_versions = manage_model_versions("bert-base-uncased")
~~~

Inference Endpoints

Local Inference Optimization

```
import torch
from transformers import pipeline

def optimize_for_inference(model_name):
    """Optimize model for faster inference"""

    # Load with optimizations
    classifier = pipeline(
        "text-classification",
        model=model_name,
        torch_dtype=torch.float16,  # Use half precision
        device_map="auto"           # Automatic device mapping
    )

    # Batch processing function
    def batch_predict(texts, batch_size=32):
        results = []
        for i in range(0, len(texts), batch_size):
            batch = texts[i:i+batch_size]
            batch_results = classifier(batch)
            results.extend(batch_results)
        return results

    return classifier, batch_predict

# Usage
classifier, batch_predict = optimize_for_inference("cardiffnlp/twitter-roberta-base-sentiment-latest")

# Test batch prediction
texts = ["I love this!", "This is terrible", "Not bad"] * 100
results = batch_predict(texts)
print(f"Processed {len(results)} texts")
```

Quantization for Edge Deployment

```python
from transformers import AutoModelForSequenceClassification, BitsAndBytesConfig

def quantize_model(model_name, quantization_type="8bit"):
    """Quantize model for reduced memory usage"""

    if quantization_type == "8bit":
        quantization_config = BitsAndBytesConfig(load_in_8bit=True)
    elif quantization_type == "4bit":
        quantization_config = BitsAndBytesConfig(
            load_in_4bit=True,
            bnb_4bit_compute_dtype=torch.float16,
            bnb_4bit_quant_type="nf4",
            bnb_4bit_use_double_quant=True
        )
    else:
        quantization_config = None

    model = AutoModelForSequenceClassification.from_pretrained(
        model_name,
        quantization_config=quantization_config,
        device_map="auto"
    )

    return model

# Usage
quantized_model = quantize_model("bert-base-uncased", "8bit")
print(f"Model memory footprint reduced with 8-bit quantization")
```

ONNX Export for Production

```python
def export_to_onnx(model_name, output_path="model.onnx"):
    """Export model to ONNX format for production deployment"""
    from transformers.onnx import export
    from transformers import AutoTokenizer, AutoModel
    from pathlib import Path

    # Load model and tokenizer
    tokenizer = AutoTokenizer.from_pretrained(model_name)
    model = AutoModel.from_pretrained(model_name)

    # Export to ONNX
    onnx_path = Path(output_path)
    export(
        preprocessor=tokenizer,
        model=model,
        config=model.config,
        opset=14,
        output=onnx_path
    )

    return onnx_path

# Usage
# onnx_path = export_to_onnx("distilbert-base-uncased")
# print(f"Model exported to {onnx_path}")
```

Best Practices

Memory Management

```python
import gc
import torch

def optimize_memory_usage():
    """Best practices for memory management"""

    # Clear cache
    torch.cuda.empty_cache()
    gc.collect()

    # Use gradient checkpointing for large models
    model.gradient_checkpointing_enable()

    # Use mixed precision training
    from torch.cuda.amp import autocast, GradScaler

    scaler = GradScaler()

    # Training with mixed precision
    with autocast():
        outputs = model(input_ids, attention_mask=attention_mask, labels=labels)
        loss = outputs.loss

    scaler.scale(loss).backward()
    scaler.step(optimizer)
    scaler.update()

def monitor_gpu_memory():
    """Monitor GPU memory usage"""
    if torch.cuda.is_available():
        print(f"GPU memory allocated: {torch.cuda.memory_allocated() / 1024**3:.2f} GB")
        print(f"GPU memory cached: {torch.cuda.memory_reserved() / 1024**3:.2f} GB")
        print(f"GPU memory free: {torch.cuda.mem_get_info()[0] / 1024**3:.2f} GB")
```

Error Handling and Robustness

```
def robust_model_loading(model_name, fallback_model="distilbert-base-uncased"):
    """Robust model loading with fallback"""
    try:
        tokenizer = AutoTokenizer.from_pretrained(model_name)
        model = AutoModel.from_pretrained(model_name)
        print(f"Successfully loaded {model_name}")
        return model, tokenizer

    except Exception as e:
        print(f"Failed to load {model_name}: {e}")
        print(f"Falling back to {fallback_model}")

        try:
            tokenizer = AutoTokenizer.from_pretrained(fallback_model)
            model = AutoModel.from_pretrained(fallback_model)
            return model, tokenizer
        except Exception as e:
            print(f"Failed to load fallback model: {e}")
            raise

def safe_inference(model, tokenizer, text, max_retries=3):
    """Safe inference with retry logic"""
    for attempt in range(max_retries):
        try:
            inputs = tokenizer(text, return_tensors="pt", truncation=True, padding=True)

            with torch.no_grad():
                outputs = model(**inputs)

            return outputs

        except RuntimeError as e:
            if "out of memory" in str(e).lower():
                print(f"OOM error, attempt {attempt + 1}/{max_retries}")
                torch.cuda.empty_cache()
                # Reduce batch size or sequence length
                if attempt < max_retries - 1:
                    continue
            raise
        except Exception as e:
            print(f"Inference error: {e}")
            if attempt == max_retries - 1:
                raise
```

Performance Monitoring

```python
import time
from functools import wraps

def benchmark_function(func):
    """Decorator to benchmark function execution time"""
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} executed in {end_time - start_time:.4f} seconds")
        return result
    return wrapper

@benchmark_function
def benchmark_model_inference(model, tokenizer, texts):
    """Benchmark model inference speed"""
    all_results = []

    for text in texts:
        inputs = tokenizer(text, return_tensors="pt")
        with torch.no_grad():
            outputs = model(**inputs)
        all_results.append(outputs)

    return all_results

# Usage
texts = ["Sample text for benchmarking"] * 100
# results = benchmark_model_inference(model, tokenizer, texts)
```

Model Selection Guide

```
def recommend_model(task, performance_priority="balanced"):
    """Recommend model based on task and performance requirements"""

    recommendations = {
        "text-classification": {
            "speed": "distilbert-base-uncased",
            "balanced": "bert-base-uncased",
            "accuracy": "roberta-large"
        },
        "question-answering": {
            "speed": "distilbert-base-cased-distilled-squad",
            "balanced": "bert-base-cased",
            "accuracy": "roberta-large-squad2"
        },
        "text-generation": {
            "speed": "gpt2",
            "balanced": "gpt2-medium",
            "accuracy": "gpt2-large"
        },
        "summarization": {
            "speed": "facebook/bart-base",
            "balanced": "facebook/bart-large-cnn",
            "accuracy": "google/pegasus-large"
        }
    }

    if task in recommendations:
        return recommendations[task].get(performance_priority, recommendations[task]["balanced"])
    else:
        return "bert-base-uncased"  # Default fallback

# Usage
model_name = recommend_model("text-classification", "speed")
print(f"Recommended model: {model_name}")
```

Conclusion

Hugging Face provides a comprehensive ecosystem for working with transformer models. This tutorial covered:

- Using pre-trained models with pipelines
- Understanding the Transformers library architecture
- Working with datasets and tokenizers
- Fine-tuning models for custom tasks
- Sharing models on the Hub
- Optimizing models for production

Key Takeaways:

1. Start with pipelines for quick prototyping
2. Use Auto classes for flexibility
3. Preprocess data carefully for best results
4. Consider parameter-efficient fine-tuning (LoRA/QLoRA)
5. Optimize models for production deployment
6. Monitor performance and memory usage

Next Steps:

1. Explore domain-specific models on the Hub
2. Experiment with multimodal models (vision + language)
3. Try advanced fine-tuning techniques
4. Build end-to-end applications with Gradio/Streamlit
5. Contribute models back to the community

Additional Resources:

- Hugging Face Documentation (https://huggingface.co/docs)
- Transformers Course (https://huggingface.co/course)
- Community Forums (https://discuss.huggingface.co)
- Model Hub (https://huggingface.co/models)

LangGraph Tutorial: Building Complex Agent Workflows

Table of Contents

Introduction to LangGraph

LangGraph is a library for building stateful, multi-actor applications with LLMs, built on top of LangChain. It extends the LangChain Expression Language with the ability to coordinate multiple chains (or actors) across multiple steps of computation in a cyclic manner.

Key Features:

- **Stateful**: Maintains state across multiple turns of conversation
- **Multi-actor**: Coordinate multiple LLM chains or agents
- **Cyclic**: Support for loops and conditional branching
- **Human-in-the-loop**: Easy integration of human feedback
- **Streaming**: Real-time streaming of intermediate results

Core Concepts

1. Nodes

Nodes represent individual processing units in your graph. Each node is a function that takes the current state and returns an updated state.

```
def my_node(state):
    # Process the state
    return {"messages": state["messages"] + [new_message]}
```

2. Edges

Edges define the flow between nodes. LangGraph supports:

- **Normal edges**: Direct connections between nodes
- **Conditional edges**: Branching based on state evaluation
- **Start/End edges**: Entry and exit points

3. State

State is the shared data structure that flows through your graph. It's typically a dictionary that gets updated by each node.

4. Checkpoints

Checkpoints allow you to save and restore the state at any point in the execution.

Installation and Setup

```
# Install LangGraph
pip install langgraph

# For development
pip install langgraph[dev]

# With additional integrations
pip install "langgraph[anthropic,openai]"
```

Environment Setup

```
import os
from langgraph.graph import StateGraph, END
from langgraph.prebuilt import ToolExecutor
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage, AIMessage

# Set your API keys
os.environ["OPENAI_API_KEY"] = "your-api-key-here"
```

Basic Graph Construction

Simple Linear Flow

```
from langgraph.graph import StateGraph, END
from typing import TypedDict, List
from langchain_core.messages import BaseMessage

class GraphState(TypedDict):
    messages: List[BaseMessage]

def chatbot(state: GraphState):
    messages = state["messages"]
    llm = ChatOpenAI()
    response = llm.invoke(messages)
    return {"messages": [response]}

def create_simple_graph():
    workflow = StateGraph(GraphState)

    # Add nodes
    workflow.add_node("chatbot", chatbot)

    # Add edges
    workflow.set_entry_point("chatbot")
    workflow.add_edge("chatbot", END)

    return workflow.compile()

# Usage
app = create_simple_graph()
result = app.invoke({"messages": [HumanMessage(content="Hello!")]})
print(result["messages"][-1].content)
```

Conditional Branching

```python
def should_continue(state: GraphState) -> str:
    messages = state["messages"]
    last_message = messages[-1]

    if "FINAL ANSWER" in last_message.content:
        return "end"
    else:
        return "continue"

def researcher(state: GraphState):
    # Research logic here
    return {"messages": state["messages"] + [AIMessage(content="Research complete")]}

def writer(state: GraphState):
    # Writing logic here
    return {"messages": state["messages"] + [AIMessage(content="FINAL ANSWER: Here's the result")]}

def create_conditional_graph():
    workflow = StateGraph(GraphState)

    workflow.add_node("researcher", researcher)
    workflow.add_node("writer", writer)

    workflow.set_entry_point("researcher")
    workflow.add_conditional_edges(
        "researcher",
        should_continue,
        {
            "continue": "writer",
            "end": END
        }
    )
    workflow.add_edge("writer", END)

    return workflow.compile()
```

Advanced Patterns

Multi-Agent Collaboration

```python
from langgraph.prebuilt import create_react_agent
from langchain_core.tools import Tool

class MultiAgentState(TypedDict):
    messages: List[BaseMessage]
    next_agent: str

def create_multi_agent_system():
    # Create specialized agents
    researcher = create_react_agent(
        ChatOpenAI(),
        [search_tool, calculator_tool]
    )

    writer = create_react_agent(
        ChatOpenAI(),
        [writing_tool, fact_checker_tool]
    )

    def agent_node(state, agent, name):
        result = agent.invoke(state)
        return {
            "messages": [AIMessage(content=result["output"])],
            "next_agent": determine_next_agent(result)
        }

    workflow = StateGraph(MultiAgentState)
    workflow.add_node("researcher", lambda x: agent_node(x, researcher, "researcher"))
    workflow.add_node("writer", lambda x: agent_node(x, writer, "writer"))

    # Routing logic
    def route_next(state):
        return state.get("next_agent", "writer")

    workflow.add_conditional_edges("researcher", route_next)
    workflow.add_conditional_edges("writer", route_next)

    return workflow.compile()
```

Human-in-the-Loop

```python
from langgraph.checkpoint.sqlite import SqliteSaver

def create_human_in_loop_graph():
    memory = SqliteSaver.from_conn_string(":memory:")

    def human_feedback(state):
        # This will pause execution and wait for human input
        pass

    workflow = StateGraph(GraphState)
    workflow.add_node("agent", chatbot)
    workflow.add_node("human", human_feedback)

    workflow.set_entry_point("agent")
    workflow.add_edge("agent", "human")
    workflow.add_edge("human", END)

    return workflow.compile(checkpointer=memory, interrupt_before=["human"])

# Usage with interruption
app = create_human_in_loop_graph()
config = {"configurable": {"thread_id": "1"}}

# Initial run - will stop at human node
result = app.invoke({"messages": [HumanMessage("Analyze this data")]}, config)

# Continue after human feedback
app.update_state(config, {"messages": [HumanMessage("User feedback here")]})
result = app.invoke(None, config)  # Resume from checkpoint
```

Parallel Processing

```python
def create_parallel_processing_graph():
    def parallel_task_1(state):
        return {"task1_result": "Result from task 1"}

    def parallel_task_2(state):
        return {"task2_result": "Result from task 2"}

    def combine_results(state):
        combined = f"{state.get('task1_result', '')} + {state.get('task2_result', '')}"
        return {"final_result": combined}

    workflow = StateGraph(dict)
    workflow.add_node("task1", parallel_task_1)
    workflow.add_node("task2", parallel_task_2)
    workflow.add_node("combine", combine_results)

    # Both tasks run in parallel
    workflow.set_entry_point("task1")
    workflow.set_entry_point("task2")

    # Both feed into combine
    workflow.add_edge("task1", "combine")
    workflow.add_edge("task2", "combine")
    workflow.add_edge("combine", END)

    return workflow.compile()
```

Real-World Examples

Research Assistant

```
def create_research_assistant():
    class ResearchState(TypedDict):
        query: str
        research_results: List[str]
        summary: str
        citations: List[str]

    def search_step(state: ResearchState):
        # Implement web search
        results = web_search(state["query"])
        return {"research_results": results}

    def analyze_step(state: ResearchState):
        # Analyze search results
        analysis = llm_analyze(state["research_results"])
        return {"summary": analysis}

    def cite_step(state: ResearchState):
        # Generate citations
        citations = extract_citations(state["research_results"])
        return {"citations": citations}

    workflow = StateGraph(ResearchState)
    workflow.add_node("search", search_step)
    workflow.add_node("analyze", analyze_step)
    workflow.add_node("cite", cite_step)

    workflow.set_entry_point("search")
    workflow.add_edge("search", "analyze")
    workflow.add_edge("analyze", "cite")
    workflow.add_edge("cite", END)

    return workflow.compile()
```

Customer Service Agent

```
def create_customer_service_agent():
    class ServiceState(TypedDict):
        customer_input: str
        intent: str
        customer_data: dict
        response: str
        escalate: bool

    def classify_intent(state: ServiceState):
        intent = classify_customer_intent(state["customer_input"])
        return {"intent": intent}

    def fetch_customer_data(state: ServiceState):
        data = get_customer_info(state.get("customer_id"))
        return {"customer_data": data}

    def handle_request(state: ServiceState):
        response = generate_response(
            state["intent"],
            state["customer_data"],
            state["customer_input"]
        )
        return {"response": response, "escalate": should_escalate(response)}

    def escalate_to_human(state: ServiceState):
        # Escalation logic
        return {"response": "Transferring to human agent..."}

    def should_escalate_decision(state: ServiceState):
        return "escalate" if state.get("escalate") else "respond"

    workflow = StateGraph(ServiceState)
    workflow.add_node("classify", classify_intent)
    workflow.add_node("fetch_data", fetch_customer_data)
    workflow.add_node("handle", handle_request)
    workflow.add_node("escalate", escalate_to_human)

    workflow.set_entry_point("classify")
    workflow.add_edge("classify", "fetch_data")
    workflow.add_edge("fetch_data", "handle")
    workflow.add_conditional_edges(
        "handle",
        should_escalate_decision,
        {"escalate": "escalate", "respond": END}
    )
    workflow.add_edge("escalate", END)

    return workflow.compile()
```

Best Practices

1. State Design

- Keep state minimal and focused
- Use TypedDict for type safety
- Avoid deeply nested structures
- Make state serializable for checkpoints

2. Error Handling

```
def robust_node(state):
    try:
        # Your logic here
        result = process_data(state)
        return {"result": result, "error": None}
    except Exception as e:
        return {"result": None, "error": str(e)}

def error_recovery(state):
    if state.get("error"):
        # Implement recovery logic
        return {"error": None, "retry_count": state.get("retry_count", 0) + 1}
    return state
```

3. Testing Strategies

```
def test_graph():
    app = create_your_graph()

    # Test individual nodes
    test_state = {"messages": [HumanMessage("test")]}
    result = app.get_node("your_node").invoke(test_state)
    assert "expected_key" in result

    # Test full flow
    final_result = app.invoke(test_state)
    assert final_result["messages"][-1].content is not None
```

4. Performance Optimization

- Use streaming for long-running operations
- Implement proper caching strategies
- Consider parallel execution where possible
- Monitor state size and complexity

5. Monitoring and Debugging

```
from langgraph.prebuilt import ToolExecutor
import logging

logging.basicConfig(level=logging.INFO)

def debug_node(state):
    logging.info(f"Node input: {state}")
    result = your_processing_function(state)
    logging.info(f"Node output: {result}")
    return result
```

Conclusion

LangGraph provides a powerful framework for building complex, stateful AI applications. By understanding its core concepts and patterns, you can create sophisticated multi-agent systems that handle real-world complexity.

Next Steps:

1. Experiment with the examples provided
2. Build your own custom nodes and edges
3. Explore the LangGraph documentation for advanced features
4. Join the LangChain community for support and updates

Additional Resources:

- LangGraph Documentation (https://python.langchain.com/docs/langgraph)
- LangGraph Examples Repository (https://github.com/langchain-ai/langgraph/tree/main/examples)
- LangChain Community (https://discord.gg/cU2adEyC7w)

OpenAI API Tutorial: Complete Guide to GPT Models and Beyond

Table of Contents

Introduction to OpenAI API

The OpenAI API provides access to powerful AI models including GPT-4, GPT-3.5, DALL-E, Whisper, and more. This tutorial covers everything you need to know to integrate OpenAI's models into your applications.

Available Models:

- **GPT-4**: Most capable model for complex reasoning
- **GPT-3.5**: Fast and efficient for most tasks
- **GPT-4 Vision**: Understands images and text
- **DALL-E 3**: Generate images from text
- **Whisper**: Speech-to-text transcription
- **TTS**: Text-to-speech synthesis

Setup and Authentication

Installation

```
# Install the OpenAI Python library
pip install openai

# For async support
pip install openai[async]

# Latest version with all features
pip install openai>=1.0.0
```

API Key Setup

```python
import openai
import os
from openai import OpenAI

# Method 1: Environment variable (recommended)
os.environ["OPENAI_API_KEY"] = "your-api-key-here"
client = OpenAI()

# Method 2: Direct initialization
client = OpenAI(api_key="your-api-key-here")

# Method 3: Using Azure OpenAI
from openai import AzureOpenAI
azure_client = AzureOpenAI(
    api_key="your-azure-key",
    api_version="2023-12-01-preview",
    azure_endpoint="https://your-endpoint.openai.azure.com/"
)
```

Basic Configuration

```python
# Set default parameters
client = OpenAI(
    api_key="your-key",
    organization="your-org-id",  # Optional
    project="your-project-id",   # Optional
    base_url="https://api.openai.com/v1",  # Custom endpoint if needed
    default_headers={"Custom-Header": "value"}
)
```

Chat Completions

Basic Chat Completion

```python
def basic_chat_completion():
    response = client.chat.completions.create(
        model="gpt-4",
        messages=[
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": "Hello! How can you help me today?"}
        ],
        max_tokens=150,
        temperature=0.7
    )

    return response.choices[0].message.content

print(basic_chat_completion())
```

Advanced Parameters

```
def advanced_chat_completion():
    response = client.chat.completions.create(
        model="gpt-4",
        messages=[
            {"role": "system", "content": "You are an expert Python developer."},
            {"role": "user", "content": "Explain list comprehensions with examples."}
        ],
        max_tokens=500,
        temperature=0.3,           # Lower = more focused
        top_p=0.9,                # Nucleus sampling
        frequency_penalty=0.0,     # Reduce repetition
        presence_penalty=0.0,      # Encourage new topics
        stop=["\n\n", "###"],      # Stop sequences
        seed=42                    # For reproducible outputs
    )

    return response.choices[0].message.content
```

Streaming Responses

```
def streaming_chat():
    stream = client.chat.completions.create(
        model="gpt-4",
        messages=[{"role": "user", "content": "Tell me a long story about AI"}],
        stream=True
    )

    full_response = ""
    for chunk in stream:
        if chunk.choices[0].delta.content is not None:
            content = chunk.choices[0].delta.content
            print(content, end="", flush=True)
            full_response += content

    return full_response
```

Conversation Management

```
class ChatManager:
    def __init__(self, system_message="You are a helpful assistant."):
        self.messages = [{"role": "system", "content": system_message}]
        self.client = OpenAI()

    def add_user_message(self, content):
        self.messages.append({"role": "user", "content": content})

    def add_assistant_message(self, content):
        self.messages.append({"role": "assistant", "content": content})

    def get_response(self, user_input):
        self.add_user_message(user_input)

        response = self.client.chat.completions.create(
            model="gpt-4",
            messages=self.messages,
            max_tokens=500,
            temperature=0.7
        )

        assistant_message = response.choices[0].message.content
        self.add_assistant_message(assistant_message)

        return assistant_message

    def clear_history(self, keep_system=True):
        if keep_system and self.messages[0]["role"] == "system":
            self.messages = [self.messages[0]]
        else:
            self.messages = []

# Usage
chat = ChatManager()
response1 = chat.get_response("What is machine learning?")
response2 = chat.get_response("Can you give me an example?")
```

Text Generation

Legacy Completions (GPT-3.5-turbo-instruct)

```
def text_completion():
    response = client.completions.create(
        model="gpt-3.5-turbo-instruct",
        prompt="Complete this sentence: The future of AI is",
        max_tokens=100,
        temperature=0.8,
        stop=["\n"]
    )

    return response.choices[0].text.strip()
```

Creative Writing Assistant

```python
def creative_writer(prompt, style="narrative", length="medium"):
    length_map = {
        "short": 200,
        "medium": 500,
        "long": 1000
    }

    system_message = f"""You are a creative writing assistant specializing in {style} writing.
    Create engaging, well-structured content that captures the reader's attention."""

    response = client.chat.completions.create(
        model="gpt-4",
        messages=[
            {"role": "system", "content": system_message},
            {"role": "user", "content": f"Write a {style} piece based on: {prompt}"}
        ],
        max_tokens=length_map.get(length, 500),
        temperature=0.8,
        presence_penalty=0.1
    )

    return response.choices[0].message.content
```

Function Calling

Basic Function Calling

```python
import json

def get_weather(location, unit="celsius"):
    """Simulate weather API call"""
    return {
        "location": location,
        "temperature": "22",
        "unit": unit,
        "condition": "sunny"
    }

def function_calling_example():
    # Define the function schema
    tools = [
        {
            "type": "function",
            "function": {
                "name": "get_weather",
                "description": "Get current weather for a location",
                "parameters": {
                    "type": "object",
                    "properties": {
                        "location": {
                            "type": "string",
                            "description": "City name"
                        },
                        "unit": {
                            "type": "string",
                            "enum": ["celsius", "fahrenheit"]
                        }
                    },
                    "required": ["location"]
                }
            }
        }
    ]

    messages = [
        {"role": "user", "content": "What's the weather like in Paris?"}
    ]

    # First call - model decides to use function
    response = client.chat.completions.create(
        model="gpt-4",
        messages=messages,
        tools=tools,
        tool_choice="auto"
    )

    # Check if model wants to call a function
    if response.choices[0].message.tool_calls:
        tool_call = response.choices[0].message.tool_calls[0]
        function_name = tool_call.function.name
        function_args = json.loads(tool_call.function.arguments)

        # Execute the function
        if function_name == "get_weather":
            function_result = get_weather(**function_args)

            # Add function result to conversation
            messages.append(response.choices[0].message)
            messages.append({
                "role": "tool",
                "content": json.dumps(function_result),
                "tool_call_id": tool_call.id
            })

            # Get final response
            final_response = client.chat.completions.create(
                model="gpt-4",
                messages=messages
            )

            return final_response.choices[0].message.content

    return response.choices[0].message.content
```

Multiple Function Agent

```python
class FunctionAgent:
    def __init__(self):
        self.client = OpenAI()
        self.functions = {
            "calculator": self.calculate,
            "search": self.search,
            "save_note": self.save_note
        }

        self.tools = [
            {
                "type": "function",
                "function": {
                    "name": "calculator",
                    "description": "Perform mathematical calculations",
                    "parameters": {
                        "type": "object",
                        "properties": {
                            "expression": {"type": "string", "description": "Math expression"}
                        },
                        "required": ["expression"]
                    }
                }
            },
            {
                "type": "function",
                "function": {
                    "name": "search",
                    "description": "Search for information",
                    "parameters": {
                        "type": "object",
                        "properties": {
                            "query": {"type": "string", "description": "Search query"}
                        },
                        "required": ["query"]
                    }
                }
            }
        ]

    def calculate(self, expression):
        try:
            result = eval(expression)  # Use safely in production!
            return f"Result: {result}"
        except:
            return "Error in calculation"

    def search(self, query):
        return f"Search results for '{query}': [Simulated results]"

    def save_note(self, content):
        # Simulate saving
        return f"Note saved: {content[:50]}..."

    def run(self, user_input):
        messages = [{"role": "user", "content": user_input}]

        response = self.client.chat.completions.create(
            model="gpt-4",
            messages=messages,
            tools=self.tools,
            tool_choice="auto"
        )

        # Handle tool calls
        if response.choices[0].message.tool_calls:
            messages.append(response.choices[0].message)

            for tool_call in response.choices[0].message.tool_calls:
                function_name = tool_call.function.name
                function_args = json.loads(tool_call.function.arguments)

                if function_name in self.functions:
                    result = self.functions[function_name](**function_args)
                    messages.append({
                        "role": "tool",
                        "content": result,
                        "tool_call_id": tool_call.id
                    })

            # Get final response
            final_response = self.client.chat.completions.create(
                model="gpt-4",
                messages=messages
            )
            return final_response.choices[0].message.content

        return response.choices[0].message.content

# Usage
agent = FunctionAgent()
result = agent.run("What's 25 * 47 + 123?")
```

Embeddings

Basic Embeddings

```
def get_embeddings(texts):
    if isinstance(texts, str):
        texts = [texts]

    response = client.embeddings.create(
        model="text-embedding-3-large",  # or text-embedding-3-small
        input=texts
    )

    return [embedding.embedding for embedding in response.data]

# Usage
text = "This is a sample sentence for embedding."
embeddings = get_embeddings(text)
print(f"Embedding dimension: {len(embeddings[0])}")
```

Semantic Search System

```python
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

class SemanticSearch:
    def __init__(self):
        self.client = OpenAI()
        self.documents = []
        self.embeddings = []

    def add_documents(self, docs):
        """Add documents to search index"""
        self.documents.extend(docs)

        # Get embeddings for new documents
        response = self.client.embeddings.create(
            model="text-embedding-3-large",
            input=docs
        )

        new_embeddings = [emb.embedding for emb in response.data]
        self.embeddings.extend(new_embeddings)

    def search(self, query, top_k=5):
        """Search for similar documents"""
        if not self.embeddings:
            return []

        # Get query embedding
        query_response = self.client.embeddings.create(
            model="text-embedding-3-large",
            input=[query]
        )
        query_embedding = query_response.data[0].embedding

        # Calculate similarities
        similarities = cosine_similarity(
            [query_embedding],
            self.embeddings
        )[0]

        # Get top results
        top_indices = np.argsort(similarities)[::-1][:top_k]

        results = []
        for idx in top_indices:
            results.append({
                "document": self.documents[idx],
                "similarity": similarities[idx]
            })

        return results

# Usage
search_engine = SemanticSearch()
search_engine.add_documents([
    "Python is a programming language",
    "Machine learning is a subset of AI",
    "Deep learning uses neural networks",
    "Natural language processing handles text"
])

results = search_engine.search("What is AI?", top_k=2)
```

```python
for result in results:
    print(f"Similarity: {result['similarity']:.3f}")
    print(f"Document: {result['document']}\n")
```

Vision Models

Image Analysis

```python
import base64
import requests

def encode_image(image_path):
    """Encode image to base64"""
    with open(image_path, "rb") as image_file:
        return base64.b64encode(image_file.read()).decode('utf-8')

def analyze_image(image_path, prompt="What's in this image?"):
    base64_image = encode_image(image_path)

    response = client.chat.completions.create(
        model="gpt-4-vision-preview",
        messages=[
            {
                "role": "user",
                "content": [
                    {"type": "text", "text": prompt},
                    {
                        "type": "image_url",
                        "image_url": {
                            "url": f"data:image/jpeg;base64,{base64_image}",
                            "detail": "high"  # or "low" for faster processing
                        }
                    }
                ]
            }
        ],
        max_tokens=300
    )

    return response.choices[0].message.content

# Usage with URL
def analyze_image_url(image_url, prompt="Describe this image"):
    response = client.chat.completions.create(
        model="gpt-4-vision-preview",
        messages=[
            {
                "role": "user",
                "content": [
                    {"type": "text", "text": prompt},
                    {"type": "image_url", "image_url": {"url": image_url}}
                ]
            }
        ],
        max_tokens=300
    )

    return response.choices[0].message.content
```

Document OCR and Analysis

```
def document_analyzer(image_path):
    """Extract and analyze text from documents"""
    base64_image = encode_image(image_path)

    response = client.chat.completions.create(
        model="gpt-4-vision-preview",
        messages=[
            {
                "role": "user",
                "content": [
                    {
                        "type": "text",
                        "text": "Extract all text from this document and provide a summary of its key points."
                    },
                    {
                        "type": "image_url",
                        "image_url": {"url": f"data:image/jpeg;base64,{base64_image}"}
                    }
                ]
            }
        ],
        max_tokens=1000
    )

    return response.choices[0].message.content
```

Audio and Speech

Speech-to-Text (Whisper)

```python
def transcribe_audio(audio_file_path):
    """Transcribe audio file using Whisper"""
    with open(audio_file_path, "rb") as audio_file:
        transcription = client.audio.transcriptions.create(
            model="whisper-1",
            file=audio_file,
            response_format="text"
        )

    return transcription

def transcribe_with_timestamps(audio_file_path):
    """Get transcription with timestamps"""
    with open(audio_file_path, "rb") as audio_file:
        transcription = client.audio.transcriptions.create(
            model="whisper-1",
            file=audio_file,
            response_format="verbose_json",
            timestamp_granularities=["word"]
        )

    return transcription

# Translation
def translate_audio(audio_file_path):
    """Translate foreign language audio to English"""
    with open(audio_file_path, "rb") as audio_file:
        translation = client.audio.translations.create(
            model="whisper-1",
            file=audio_file
        )

    return translation.text
```

Text-to-Speech

```python
def text_to_speech(text, voice="alloy", output_file="speech.mp3"):
    """Convert text to speech"""
    response = client.audio.speech.create(
        model="tts-1",  # or "tts-1-hd" for higher quality
        voice=voice,    # alloy, echo, fable, onyx, nova, shimmer
        input=text,
        speed=1.0       # 0.25 to 4.0
    )

    with open(output_file, "wb") as f:
        for chunk in response.iter_bytes():
            f.write(chunk)

    return output_file

# Real-time streaming
def streaming_text_to_speech(text, voice="alloy"):
    """Stream audio in real-time"""
    response = client.audio.speech.create(
        model="tts-1",
        voice=voice,
        input=text,
        response_format="opus"  # Better for streaming
    )

    # Play audio chunks as they arrive
    for chunk in response.iter_bytes(chunk_size=1024):
        # Send to audio player
        yield chunk
```

Fine-tuning

Prepare Training Data

```python
import json

def prepare_training_data(examples):
    """Prepare data for fine-tuning"""
    training_data = []

    for example in examples:
        training_data.append({
            "messages": [
                {"role": "system", "content": "You are a helpful assistant."},
                {"role": "user", "content": example["input"]},
                {"role": "assistant", "content": example["output"]}
            ]
        })

    # Save to JSONL file
    with open("training_data.jsonl", "w") as f:
        for item in training_data:
            f.write(json.dumps(item) + "\n")

    return "training_data.jsonl"

# Example data
examples = [
    {"input": "What is Python?", "output": "Python is a programming language..."},
    {"input": "How do lists work?", "output": "Lists in Python are ordered collections..."}
]

training_file = prepare_training_data(examples)
```

Fine-tuning Process

```
def create_fine_tuning_job(training_file, model="gpt-3.5-turbo"):
    """Create a fine-tuning job"""
    # Upload training file
    with open(training_file, "rb") as f:
        file_response = client.files.create(
            file=f,
            purpose="fine-tune"
        )

    # Create fine-tuning job
    job = client.fine_tuning.jobs.create(
        training_file=file_response.id,
        model=model,
        hyperparameters={
            "n_epochs": 3,
            "batch_size": 1,
            "learning_rate_multiplier": 2
        }
    )

    return job

def monitor_fine_tuning(job_id):
    """Monitor fine-tuning progress"""
    job = client.fine_tuning.jobs.retrieve(job_id)

    print(f"Job ID: {job.id}")
    print(f"Status: {job.status}")
    print(f"Model: {job.fine_tuned_model}")

    # Get events
    events = client.fine_tuning.jobs.list_events(job_id)
    for event in events.data[:5]:  # Show last 5 events
        print(f"{event.created_at}: {event.message}")

    return job

def use_fine_tuned_model(model_id, prompt):
    """Use your fine-tuned model"""
    response = client.chat.completions.create(
        model=model_id,
        messages=[
            {"role": "user", "content": prompt}
        ]
    )

    return response.choices[0].message.content
```

Best Practices

Error Handling

```
from openai import RateLimitError, APIError
import time

def robust_api_call(func, max_retries=3, backoff_factor=2):
    """Robust API call with retry logic"""
    for attempt in range(max_retries):
        try:
            return func()
        except RateLimitError:
            if attempt == max_retries - 1:
                raise
            wait_time = backoff_factor ** attempt
            print(f"Rate limit hit, waiting {wait_time} seconds...")
            time.sleep(wait_time)
        except APIError as e:
            print(f"API Error: {e}")
            if attempt == max_retries - 1:
                raise
            time.sleep(backoff_factor ** attempt)

# Usage
def safe_chat_completion(message):
    return robust_api_call(
        lambda: client.chat.completions.create(
            model="gpt-4",
            messages=[{"role": "user", "content": message}]
        )
    )
```

Token Management

```
import tiktoken

def count_tokens(text, model="gpt-4"):
    """Count tokens in text"""
    encoding = tiktoken.encoding_for_model(model)
    return len(encoding.encode(text))

def truncate_text(text, max_tokens, model="gpt-4"):
    """Truncate text to fit within token limit"""
    encoding = tiktoken.encoding_for_model(model)
    tokens = encoding.encode(text)

    if len(tokens) <= max_tokens:
        return text

    truncated_tokens = tokens[:max_tokens]
    return encoding.decode(truncated_tokens)

def smart_chunking(text, chunk_size=1000, model="gpt-4"):
    """Split text into chunks based on token count"""
    encoding = tiktoken.encoding_for_model(model)
    tokens = encoding.encode(text)

    chunks = []
    for i in range(0, len(tokens), chunk_size):
        chunk_tokens = tokens[i:i + chunk_size]
        chunk_text = encoding.decode(chunk_tokens)
        chunks.append(chunk_text)

    return chunks
```

Cost Optimization

```
class CostTracker:
    def __init__(self):
        self.costs = {
            "gpt-4": {"input": 0.03, "output": 0.06},  # per 1K tokens
            "gpt-3.5-turbo": {"input": 0.001, "output": 0.002},
            "text-embedding-3-large": {"input": 0.00013, "output": 0}
        }
        self.total_cost = 0

    def calculate_cost(self, model, input_tokens, output_tokens):
        if model in self.costs:
            cost = (
                (input_tokens / 1000) * self.costs[model]["input"] +
                (output_tokens / 1000) * self.costs[model]["output"]
            )
            self.total_cost += cost
            return cost
        return 0

    def tracked_completion(self, **kwargs):
        response = client.chat.completions.create(**kwargs)

        usage = response.usage
        cost = self.calculate_cost(
            kwargs["model"],
            usage.prompt_tokens,
            usage.completion_tokens
        )

        print(f"Cost: ${cost:.4f} | Total: ${self.total_cost:.4f}")
        return response

# Usage
tracker = CostTracker()
response = tracker.tracked_completion(
    model="gpt-4",
    messages=[{"role": "user", "content": "Hello!"}]
)
```

Async Operations

```python
import asyncio
from openai import AsyncOpenAI

async_client = AsyncOpenAI()

async def async_chat_completion(message):
    """Async chat completion"""
    response = await async_client.chat.completions.create(
        model="gpt-4",
        messages=[{"role": "user", "content": message}]
    )
    return response.choices[0].message.content

async def batch_completions(messages):
    """Process multiple completions concurrently"""
    tasks = [async_chat_completion(msg) for msg in messages]
    results = await asyncio.gather(*tasks)
    return results

# Usage
async def main():
    messages = [
        "What is Python?",
        "What is JavaScript?",
        "What is Rust?"
    ]

    results = await batch_completions(messages)
    for i, result in enumerate(results):
        print(f"Question {i+1}: {result[:100]}...")

# Run
# asyncio.run(main())
```

Production Configuration

```
class ProductionOpenAI:
    def __init__(self, api_key=None):
        self.client = OpenAI(
            api_key=api_key or os.getenv("OPENAI_API_KEY"),
            timeout=30,
            max_retries=3
        )
        self.default_params = {
            "temperature": 0.7,
            "max_tokens": 1000,
            "top_p": 0.9
        }

    def chat(self, messages, **kwargs):
        params = {**self.default_params, **kwargs}

        try:
            response = self.client.chat.completions.create(
                messages=messages,
                **params
            )
            return {
                "success": True,
                "content": response.choices[0].message.content,
                "usage": response.usage,
                "model": response.model
            }
        except Exception as e:
            return {
                "success": False,
                "error": str(e),
                "content": None
            }
```

Conclusion

The OpenAI API provides powerful capabilities for building AI-powered applications. This tutorial covered the essential patterns and best practices for:

- Chat completions and conversation management
- Function calling for tool integration
- Embeddings for semantic search
- Vision capabilities for image analysis
- Audio processing with Whisper and TTS
- Fine-tuning for specialized models
- Production-ready error handling and optimization

Next Steps:

1. Experiment with different models and parameters
2. Build a complete application using multiple API features
3. Implement proper monitoring and cost tracking
4. Explore advanced techniques like RAG and agent frameworks

Additional Resources:

- OpenAI API Documentation (https://platform.openai.com/docs)
- OpenAI Cookbook (https://github.com/openai/openai-cookbook)
- Best Practices Guide (https://platform.openai.com/docs/guides/production-best-practices)