Can LLMs Reason?

Introduction

- LLMs are great at creating text that makes sense, but they mostly follow patterns in the data they were trained on instead of actually understanding logic or ideas.
- Their weakness in real reasoning shows when they need to solve problems step by step, handle complex ideas over time, or think beyond what they were trained on.

Datasets for Analysis:

Winograd Schema Challenge (WSC)

Winograd Schema Challenge

What is WSC?

A test to see if a model can use common sense to understand pronouns in a sentence.

Example:

“The trophy doesn’t fit in the brown suitcase because it is too small. What is too small?"

- Human Answer: "The suitcase"
- LLM Answer: "The trophy"

LLM Challenges:

- Relies on statistical correlations, not logical deduction.
- Prone to making contextually inconsistent choices.
- This leads to answers that don't always make sense in context.

Common sense QA

What is Common Sense QA?

A multiple-choice question dataset requiring commonsense knowledge.

Example:

"A farmer has 10 apples and gives 4 to his neighbor. How many apples does he have left?"

Options: (a) 4 (b) 6 (c) 10

- Human answer : 6
- LLM Often selects (c) '10', reflecting confusion between initial and remaining quantities.

LLM Challenges:

- Misinterprets nuanced questions.
- Struggles with implicit knowledge.

Why LLMs Fail at Reasoning

- Lack of true understanding.
- Dependence on statistical patterns over logical structures.
- Absence of symbolic reasoning.
- Inability to perform multi-step reasoning reliably.

How to Improve

- Hybrid Models: Combining LLMs with symbolic reasoning frameworks.
- Reinforcement Learning (RL): Training LLMs on reasoning-specific tasks with feedback.
- Tool Integration: Use of external knowledge bases and logic solvers.
- Future Frameworks: Development of neuro-symbolic models to bridge the gap.

Can LLMs Reason?

No, not in the true sense.

Evidence:
Failures in WSC and Common sense QA demonstrate lack of logical reasoning.

Path Forward:
Hybrid approaches, ReInforcement Learning and neuro-symbolic models offer hope for future improvements.

Conclusion

- LLMs excel in generating natural and coherent text, but they fall short when it comes to genuine reasoning.
- Their capabilities are driven by patterns in training data, which limits their ability to perform logical problem-solving or adapt to unfamiliar scenarios.

Design and Procedure to Deploy and Serve LLM on AWS SageMaker and EKS

Introduction

Objective:

- Deploy and serve Large Language Models (LLMs) for production use using AWS SageMaker and Amazon EKS.

- Two primary deployment approaches:
  - SageMaker Deployment
  - FastAPI Deployment on EKS

Approach 1 : SageMaker Deployment

Overview of SageMaker Deployment Workflow

- Objective: Deploy a fine-tuned LLM using AWS SageMaker for scalable and secure AI model serving.

- Key Steps:
    - Fine-tune the LLM
    - SageMaker Deployment Pipeline
    - Model Hosting
    - Monitoring and Scaling

Step 1: Fine-Tune the LLM

Model Selection

- Browse AWS Bedrock to select an appropriate pre-trained model that aligns with your use case (e.g., Gpt-2 ,Gpt-3).

Fine-Tuning Process

- Utilize Bedrock's fine-tuning APIs to customize the model for your specific task.
- Incorporate domain-specific data and configure hyperparameters (e.g., learning rate, batch size).

Output Artifacts

- Save the fine-tuned model's artifacts (e.g., .tar.gz file) in an S3 bucket for subsequent deployment steps.

Visual Elements:

- **Diagram: AWS Bedrock → Fine-Tuning API → S3 Storage**

Step 2 : SageMaker Deployment Pipeline

1. Set Up SageMaker Project

Model Building Pipelines: WE can Use SageMaker Model Building Pipelines or external CI/CD tools like AWS CodePipeline to automate the pipeline.

- Pipeline Steps:
  - Data Preprocessing: Prepare the data for training.
  - Training: Run the fine-tuned model as a training job.
  - Model Evaluation: Evaluate and refine the model.
  - Model Packaging: Package the model for deployment.

2. Model Registry for Versioning

- Register the fine-tuned model in SageMaker Model Registry.
- Create a Model Group for version management.
- Automate the approval and promotion of models to deployment pipelines.

3. Model Hosting

- Deploy the model to a SageMaker Endpoint:
  - we can a real- time endpoint for reducing latency of the predictions
  - Use asynchronous or batch transform endpoints for large-scale batch jobs.

- Specify model.tar.gz from S3 in the deployment script.

4. Monitoring and Scaling

- Set up **Amazon CloudWatch** to keep an eye on key metrics such as model performance, latency, and the number of requests. You can create custom alerts to notify you of any issues with the endpoint, helping you catch problems before they impact users.
- Implement Auto Scaling to automatically adjust the number of instances based on traffic and resource usage.
- This ensures your model can handle spikes in demand without manual intervention, keeping things running smoothly.

Approach 2: FastAPI Deployment on EKS

Overview of FastAPI Deployment on EKS Workflow

- Key Steps:
  - Fine-tune the LLM
  - FastAPI Application
  - Containerization
  - Deploy to EKS
  - API Gateway & Lambda Integration
  - Monitoring and Logging

1. Fine-Tune the LLM

- Similar to the first approach, leverage AWS Bedrock for fine-tuning.
- Save model artifacts to S3 or an artifact repository for EKS consumption.

2. FastAPI Application

1. Build FastAPI
    - Create a Python FastAPI app to serve predictions.
    - Load the model in the app during startup to avoid overhead per request.
    - Expose endpoints like `/predict` or `/health`.

2. Containerization

- Create a Dockerfile for the FastAPI app
- Build and push the container image to Amazon Elastic Container Registry (ECR).

3. Deploy to EKS

1. **Set up Kubernetes (EKS) Cluster:**

   Start by creating an EKS cluster using either eksctl or the AWS Console, depending on your preference and setup requirements.

2. **Deploy the FastAPI Container to EKS:**

   Deploy your FastAPI application container to EKS by creating the necessary Kubernetes Deployment and Service YAML manifests.

3. **External Access via LoadBalancer or Ingress:**

   For external access, use a LoadBalancer or Ingress to route traffic to your FastAPI app.

4. Integrate API Gateway and Lambda

1. Set up API Gateway to expose your FastAPI application and route traffic to the EKS service.
2. Configure caching and throttling in API Gateway as needed to manage traffic flow and improve performance.
3. Optionally, use a Node.js Lambda Function to handle routing and middleware tasks, with express or nginx, and proxy requests to the EKS service or FastAPI app.

5. Monitoring and Logging

1. Use CloudWatch for monitoring API Gateway and Lambda.
2. Set up Kubernetes monitoring with Prometheus and Grafana.

Reference Links

1. “Understanding the Limitations of Mathematical Reasoning in Large Language Models” 7 Oct 2024 by Iman Mirzadeh,Keivan Alizadeh, Hooman Shahrokhi ,Oncel Tuzel, Samy Bengio,   Mehrdad Farajtabar

2. https://huggingface.co/datasets/ErnestSDavis/winograd_wsc from hugging face '

3. https://huggingface.co/datasets/Sadanto3933/commonsense_qa