Knowledge Distillation: Finance Domain

Author : Hari Venkata

Understanding Proprietary LLMs: Capabilities and Constraints

Proprietary Large Language Models (LLMs) like GPT-3.5, GPT-4, Gemini, and Claude have dramatically advanced the field of natural language processing, enabling human-like text generation and powerful problem-solving capabilities. These massive models demonstrate emergent abilities — skills that go beyond their initial training — making them highly versatile across a wide range of tasks. Their influence spans creative content generation, complex reasoning, and more, signaling a transformative shift in how we interact with technology. However, despite their impressive performance, proprietary LLMs come with notable limitations. High costs, restricted access, and data privacy concerns make them less practical for individuals or smaller organizations. Additionally, their general-purpose design might not suit specific, niche applications, highlighting the growing need for more accessible, customizable, and secure alternatives.

Open-Source Models Like LLaMA and Mistral: Powerful, But Not Without Limits

Open-source LLMs like LLaMA and Mistral offer significant advantages over proprietary models, primarily due to their accessibility, adaptability, and community-driven development. Free from licensing restrictions and high usage costs, these models promote collaboration, innovation, and customized solutions, making them ideal for research and niche applications. However, their relatively smaller size and limited pretraining investment often result in lower performance on complex, real-world tasks compared to models like GPT-4. They may also lack the depth and fine-tuning necessary to excel in specialized domains.

Bridging the Gap: Knowledge Distillation for Smarter Open-Source LLMs

To address this performance gap, **knowledge distillation (KD)** has emerged as a powerful technique. KD allows open-source models (students) to learn from the strengths of proprietary LLMs (teachers) by mimicking their behavior. This not only boosts the efficiency of open-source models but also enhances their capabilities. Newer approaches like data augmentation-based KD and self-distillation — where open-source models improve by learning from their own stronger versions — are pushing this idea further. These strategies help compress and refine LLMs, making them lighter and more effective without heavily compromising performance.

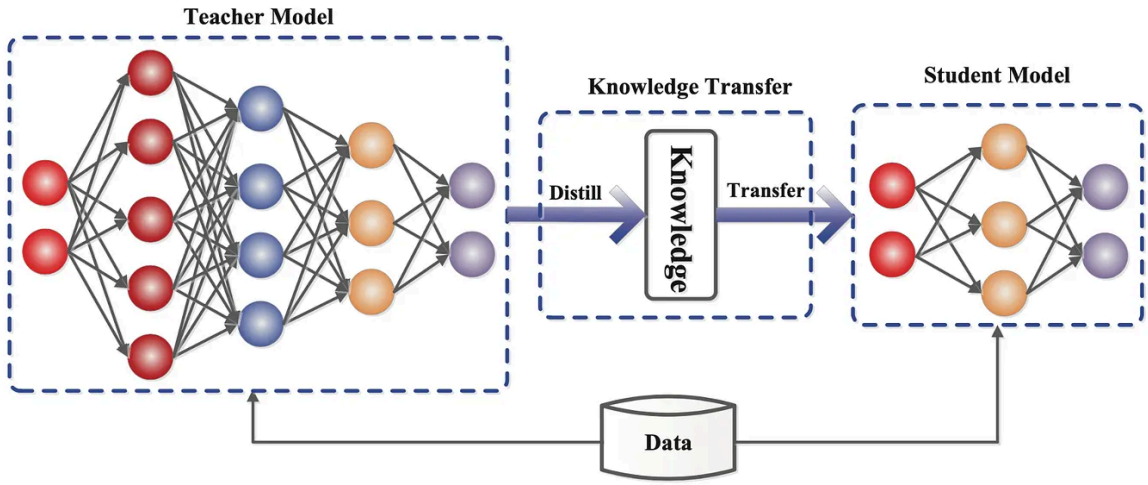
LLMs in the Financial Landscape

The integration of Large Language Models (LLMs) into the financial domain is revolutionizing how institutions analyze data, make strategic decisions, and engage with customers. LLMs excel at understanding and extracting insights from complex financial documents, predicting market dynamics, and streamlining risk assessments. By leveraging their ability to process massive volumes of unstructured data — including news articles, earnings reports, and live market feeds — these models unlock patterns and trends that were previously difficult or even impossible to detect. This leads to faster, more accurate decision-making and opens the door to smarter, data-driven financial planning.

Beyond analytics, LLMs are also transforming customer experience in finance. They enable personalized financial advice, automate customer support, and power intelligent chatbots capable of handling sophisticated queries. These capabilities not only enhance service quality but also improve operational efficiency, reduce costs, and support better compliance and risk management. While much of the current work in financial AI focuses on fine-tuning or continually pretraining general-purpose LLMs on financial data, the use of **knowledge distillation** from proprietary LLMs remains relatively unexplored. This presents an exciting opportunity to create efficient, domain-specialized models without compromising performance — a crucial step toward scaling LLMs across the financial industry.

What is **Knowledge Distillation?**

Knowledge Distillation (KD) is a model compression technique that transfers knowledge from a large, complex model (or ensemble) to a smaller, more efficient model suitable for real-world deployment. Originally demonstrated in 2006 and later formalized by Hinton et al., KD has become especially important with the rise of deep learning, enabling smaller models to retain much of the performance of larger ones. It's particularly useful for deploying neural networks on resource-constrained devices like mobile or edge systems.



Knowledge Distillation in NLP

Knowledge distillation plays a crucial role in Natural Language Processing (NLP), especially given the scale of modern deep learning models. For instance, cutting-edge language models like GPT-3 have around **175 billion parameters**, which is vastly larger than earlier models such as BERT-base, which has **110 million parameters**.

Due to the computational demands of such large models, knowledge distillation is widely used in NLP to create **smaller, faster, and more efficient models**. These compact models are significantly easier to train and deploy, making them ideal for practical applications with limited resources. Besides language modeling, KD is used across a range of NLP tasks including:

- Neural Machine Translation
- Text Generation
- Question Answering

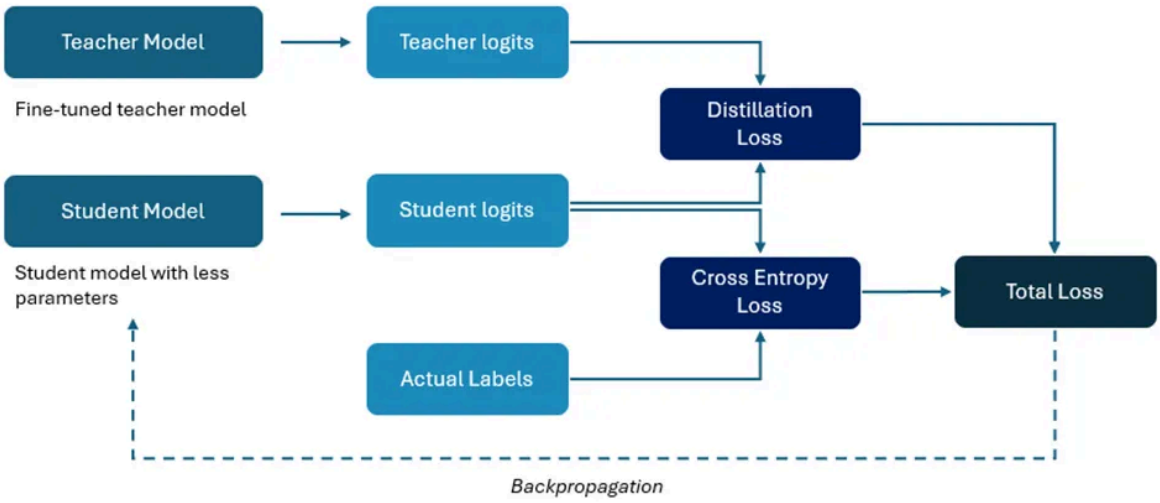
- Document Retrieval
- Text Recognition

Distillation techniques also enable effective multilingual NLP solutions by allowing student models to learn and share knowledge from powerful multilingual teacher models.

Case Study: DistilBERT

DistilBERT, developed by Hugging Face, is a leaner version of BERT — 40% smaller (66M vs. 110M parameters), 60% faster, and retains about 97% of BERT's performance on benchmarks like GLUE. It achieves this via a customized training process that uses a triplet loss function, combining traditional language modeling loss, distillation loss, and cosine-distance loss. Notably, DistilBERT shares the same architecture as BERT but is more efficient for deployment on limited-resource environments

KD : Training Strategy



Knowledge distillation from teacher models can be performed in several ways: through the responses (or logits) of the teacher model, known as **response-based knowledge distillation**; by leveraging the weights and activations of the teacher model, referred to as **feature-based knowledge distillation**; or by utilizing the

relationships between model parameters, which is known as **relationship-based knowledge distillation**. This blog will focus on applying response-based knowledge distillation in large language models. The image below illustrates knowledge distillation using response-based knowledge from a larger teacher model.

KD Implementation with PyTorch:

1. Install Required Libraries

Before diving into the code, we first need to install the required libraries. These include `transformers`, `datasets`, and `torch`. This setup allows us to work with pre-trained models and datasets seamlessly.

```
pip install transformers datasets torch
```

2. Import Required Libraries

In this section, we import the necessary libraries for our task.

```
import torch
from transformers import AutoModelForSequenceClassification,
from datasets import load_dataset
```

- `torch`: For working with PyTorch models and tensors.
- `transformers`: To load pre-trained models like BERT and DistilBERT from Hugging Face.
- `datasets`: To easily load datasets like GLUE or others for fine-tuning.
- `Trainer` and `TrainingArguments`: These are used to train the models with Hugging Face's training loop.

3. Load Teacher and Student Models

Here, we load the teacher (larger) and student (smaller) models from Hugging Face's model hub.

```
teacher_model_name = "bert-large-uncased"
```

```
student_model_name = "distilbert-base-uncased"
```

```
# Load teacher and student models
teacher_model = AutoModelForSequenceClassification.from_pretr
student_model = AutoModelForSequenceClassification.from_pretr
```

- **Teacher Model**: We use a large pre-trained model (`bert-large-uncased`) as the teacher model.
- **Student Model**: We use a smaller pre-trained model (`distilbert-base-uncased`) as the student model.

The idea is to transfer knowledge from the teacher to the student using knowledge distillation.

4. Load Tokenizer and Dataset

Here, we load a tokenizer to convert text data into tokens, and then load the dataset we will use for training. We use the SST-2 dataset for sentiment analysis from the GLUE benchmark.

```
tokenizer = AutoTokenizer.from_pretrained(student_model_name)

# Load dataset for training (e.g., SST-2 for sentiment analys
dataset = load_dataset("glue", "sst2")

# Tokenize dataset

def tokenize_function(examples):
    return tokenizer(examples['sentence'], padding=True, trun

tokenized_datasets = dataset.map(tokenize_function, batched=T
```

- **Tokenizer**: We use the student model's tokenizer to process the text.
- **Dataset**: We load the **SST-2** dataset, which contains sentences labeled as positive or negative sentiment.

- **Tokenization**: We define a function to tokenize the sentences and apply padding and truncation to ensure consistent input size.

5. Define the Distillation Loss Function

Here, we define the loss function used for knowledge distillation. The distillation loss compares the logits (predictions) of the student and teacher models.

```
def distillation_loss(student_logits, teacher_logits, tempera
"""
Knowledge distillation loss using the logits (response-based)
"""
    # Softmax temperature scaling
    student_probs = torch.nn.functional.softmax(student_logit
    teacher_probs = torch.nn.functional.softmax(teacher_logit
    # Cross-entropy loss between student and teacher's softma
    loss = torch.nn.functional.kl_div(student_probs.log(), te
    return loss
```

- **Temperature Scaling**: We scale the logits using a temperature factor to soften the probabilities. This helps the student model learn from the teacher model's softer predictions.
- **KL Divergence**: We compute the Kullback-Leibler divergence between the student's and teacher's predicted probabilities. This is the distillation loss.

6. Custom Trainer Class for Knowledge Distillation

In this section, we define a custom `Trainer` class that computes the distillation loss instead of the default cross-entropy loss. This allows the student model to learn from the teacher model using the distillation approach.

```
class KnowledgeDistillationTrainer(Trainer):
def compute_loss(self, model, inputs, return_outputs=False):
"""
Override compute_loss to use teacher model for distillation l
"""
        # Get teacher's logits
        with torch.no_grad():
        teacher_logits = teacher_model(**inputs).logits
```

```
        # Get student model's logits
    student_logits = model(**inputs).logits

    # Compute distillation loss
    loss = distillation_loss(student_logits, teacher_logits)

    return (loss, None) if not return_outputs else (loss, stu
```

- **Override** `compute_loss`: We override the `compute_loss` method in the `Trainer` class to calculate the distillation loss instead of the default loss.
- **Teacher's Logits**: The teacher model is used to generate logits (predictions), which are then compared with the student model's logits.

7. Set Training Arguments

Here, we define the training parameters such as learning rate, batch size, and the number of epochs for training.

```
training_args = TrainingArguments(
output_dir="./results",          # output directory
evaluation_strategy="epoch",     # evaluation strategy to ado
learning_rate=2e-5,              # learning rate
per_device_train_batch_size=8,   # batch size for training
per_device_eval_batch_size=16,   # batch size for evaluation
num_train_epochs=3,              # number of training epochs
weight_decay=0.01,               # strength of weight decay
)
```

- `output_dir`: Directory to save the trained model and logs.
- `evaluation_strategy`: Defines how often to evaluate the model during training. Here, it's set to evaluate every epoch.
- `learning_rate`: The learning rate for the optimizer.
- `batch_size`: Batch size for training and evaluation.
- `num_train_epochs`: The number of times the model will see the entire training dataset.
- `weight_decay`: A regularization term to prevent overfitting.

8. Initialize and Train the Model

Finally, we initialize the custom trainer and start training the student model using the distillation process.

```
trainer = KnowledgeDistillationTrainer(
model=student_model,
args=training_args,
train_dataset=tokenized_datasets["train"],
eval_dataset=tokenized_datasets["validation"],
tokenizer=tokenizer,
)

# Train the student model using the KD approach

trainer.train()
```

- **Custom Trainer**: We pass our custom `KnowledgeDistillationTrainer` with the required arguments.
- **Training**: The `trainer.train()` method starts the training process. During training, the student model will learn from the teacher model's logits using the distillation loss.

Conclusion

By splitting the code into smaller, manageable chunks, we can see how knowledge distillation helps in transferring knowledge from a larger model (teacher) to a smaller model (student). This process allows the student model to perform similarly to the teacher model, with reduced size and computational cost, making it suitable for deployment in resource-constrained environments.

In this example, we used **DistilBERT** as a student model and a larger **BERT** model as the teacher, fine-tuning them. The knowledge distillation technique used here is response-based, where the logits (predictions) of the teacher model are used to guide the student model's learning process.