10

Privacy Preserving in Large Language Models

Large Language Models (LLMs) have emerged as a transformative technology in the field of artificial intelligence, enabling advanced natural language processing tasks and generative capabilities. These models, such as OpenAI's GPT-3.5, Meta’s Llama-2 have shown remarkable proficiency in generating human-like text and demonstrating a deep understanding of language patterns. In this chapter, you will learn about closed source large language model, open source large langue models at high level and privacy issues with these LLM’s and state of the art research in privacy preserving technologies for LLM’s.

We will cover the following main topics:

* Introduction to Large Language Models
  + High Level Overview of LLM’s
  + Key concepts/terms used in LLMs.’
  + Prompt Engineering
    - Example using ChatGPT as well using open source LLM’s.
  + Comparison of Open Source LLM’s vs Closed Source LLMs
* Privacy Attacks on LLM’s
  + Real world incidents on Privacy Leaks in LLM’s
  + Membership inference attacks against generative models
  + Extracting training data from large language models
  + Prompts Injection attacks.
* Privacy Preserving Technologies for LLM’s
  + Text Attacks on ML and Gen AI
  + Training LLM’s using Differential Privacy with Private Transformer
  + State of the art research (STOA) on Privacy Preserving LLM’s

Introduction to Large Language Models

Large Language Models (LLMs) are a category of artificial intelligence models that have gained immense prominence in recent years due to their exceptional ability to understand and generate human-like text. These models are often based on deep learning architectures, particularly transformers.

High Level overview of Large Language Models:

Transformer Architecture:

Most Large Language Models are built upon the Transformer architecture. Transformers are neural networks that excel at handling sequential data, making them well-suited for natural language understanding and generation tasks.

Transformers introduced the concept of self-attention mechanisms, which allows the model to weigh the importance of different words in a sentence when processing it. This innovation significantly improved the model's ability to understand context and dependencies within text.

Scale and Size:

What sets Large Language Models apart is their scale and size. These models have a massive number of parameters, often ranging from hundreds of millions to billions. The more parameters a model has, the better it typically performs on a wide range of language tasks.

Examples of well-known LLMs include GPT-3, GPT-4, BERT (Bidirectional Encoder Representations from Transformers), FLAN T5 (Text-to-Text Transfer Transformer), and many more.

Pretraining and Fine-Tuning:

Large Language Models are pretrained on vast corpora of text data, which means they learn from a massive amount of text from the internet. This initial pretraining helps the model grasp the nuances of language, grammar, context, and even some world knowledge.

After pretraining, LLMs can be fine-tuned on specific tasks, such as text classification, translation, summarization, and more. Fine-tuning makes them adaptable and capable of performing various natural language processing tasks with minimal task-specific data.

Versatility:

LLMs are versatile and can be used for a wide range of applications, including:

Text Generation: They can generate human-like text, which can be used for chatbots, content generation, and creative writing.

* Language Translation: LLMs excel at translating text from one language to another.
* Sentiment Analysis: They can determine the sentiment (positive, negative, neutral) of a piece of text.
* Question Answering: LLMs can answer questions based on a given context.
* Summarization: They can generate concise summaries of longer text documents.
* Language Understanding: They can understand and analyze the meaning of text.

Ethical Considerations:

The sheer power of LLMs raises ethical concerns, including the potential for misuse. Large Language Models can generate text that is indistinguishable from human-written content, making them susceptible to producing fake news, disinformation, or biased content and sensitive data leaks as well.

Research Efforts are being made to address these ethical concerns through responsible AI research, including content moderation, bias detection, and fact-checking.

Computational Resources:

Training and running Large Language Models require massive computational resources, including high-end GPUs and TPUs (Tensor Processing Units). This poses challenges for smaller organizations and researchers with limited resources.

Ongoing Advancements:

The field of Large Language Models continues to evolve rapidly. Researchers are continually pushing the boundaries of model size, efficiency, and performance, seeking to build even larger and more capable models.

Large Language Models are a transformative force in natural language processing and understanding. Their scale, versatility, and potential for real-world applications make them a focal point of research and development in the field of artificial intelligence. However, they also come with ethical and computational challenges that need to be carefully addressed as the technology continues to advance.

Key Concepts/terms in Large Language Models

Large Language Models (LLMs) are a complex field of natural language processing (NLP), and there are several terms associated with them.

Some of the key terms and concepts used in the context of LLMs:

Transformer Architecture: The foundational architecture for most LLMs, known for its self-attention mechanism, which allows the model to weigh the importance of different words in a sentence.

Pretraining: The initial phase in which the LLM is trained on a massive corpus of text data from the internet to learn language patterns and context. This pretrained model is often referred to as the "base model."

Fine-Tuning: The subsequent phase where the pretrained model is adapted to perform specific NLP tasks, such as text classification, translation, summarization, or question answering. Fine-tuning helps the model specialize in these tasks.

Parameters: These are the trainable components of the LLM, represented by numerical values. The number of parameters is a key factor in determining the size and capability of an LLM.

Attention Mechanism: A core component of the Transformer architecture, it enables the model to focus on different parts of the input sequence when processing it, improving contextual understanding.

Self-Attention: A specific type of attention where the model assigns weights to different words in a sentence based on their relevance to each other, allowing it to capture dependencies between words. Most of the transformers are built based on the research paper from Google “ Attention Is All You Need”. <https://arxiv.org/abs/1706.03762>

Embeddings: Word embeddings or token embeddings are vector representations of words or tokens in a continuous space. These embeddings capture semantic relationships between words.

Contextual Embeddings: Unlike static word embeddings, these embeddings change based on the context of the sentence, allowing LLMs to understand the meaning of words in different contexts. Positional Embeddings and rotary position embeddings are comes under contextual embeddings.

Tokenization: The process of breaking down text into individual tokens (words or subwords) for input into the model. LLMs use tokenizers to perform this task.

Decoding: The process of converting model-generated representations (usually logits or token IDs) into human-readable text. Decoding is necessary to produce the final output.

Transfer Learning: The concept of transferring knowledge gained from one task or domain to another. LLMs often benefit from transfer learning, as they are pretrained on a broad range of text before fine-tuning for specific tasks.

Prompt Engineering

Prompt Engineering: The process of designing input prompts or instructions that guide the LLM to generate desired output. Crafting effective prompts is crucial in controlling the model's behaviour.

Zero-shot Learning: A type of transfer learning where a model is asked to perform a task for which it was not explicitly fine-tuned. LLMs are capable of zero-shot learning to some extent.

Few-shot Learning: Similar to zero-shot learning, but the model is provided with a limited number of examples for a new task during fine-tuning.

**Question**

Prompt

**Answer**

Result

Example using ChatGPT

Let’s try an example using ChatGPT ( <https://chat.openai.com/> ) and ask question translate a sentence from English to German.

In this case the question is called as Prompt and the response from ChatGPT ( LLM’s) called as completion/result.

A screenshot of a phone

Description automatically generated

Example using Open source LLMS

Let’s try an example using the open source LLMs programmatically using python.

This code snippet demonstrates how to use the Google FLAN-T5 model for translation from English to German. It utilizes the Hugging Face Transformers library, specifically the T5Tokenizer and T5ForConditionalGeneration classes.

Steps followed to implement this example:

Ensure that you have the Hugging Face Transformers library installed and that you have access to the "google/flan-t5-large" model for this code to run successfully.

* install the library using
  + pip install transformers
* Additionally, you might need to download the model using transformers.AutoModel.from\_pretrained("google/flan-t5-large") if you haven't already.
* Import the necessary classes from the Transformers library, namely T5Tokenizer and T5ForConditionalGeneration.
* Initialize the T5 tokenizer and model with the pre-trained "google/flan-t5-large" model. This model is designed for translation tasks.
* Define the input text you want to translate from English to German, which is "translate English to German: How old are you?"
* Tokenize the input text using the tokenizer, and convert it into PyTorch tensors. This step prepares the text for input to the model.
* Generate the translation using the T5 model by passing the tokenized input to the model's generate method. The translation output is stored in the outputs variable.
* Decode the generated output using the tokenizer's decode method, and print the translated text, which will be the German translation of the input text.

Source code : Ex\_LLM\_Opensource.ipynb

!pip install transformers

from transformers import T5Tokenizer,

T5ForConditionalGeneration

tokenizer = T5Tokenizer.from\_pretrained("google/flan-t5-large")

model =

T5ForConditionalGeneration.from\_pretrained("google/flan-t5-large")

input\_text = "translate English to German: How old are you?"

input\_ids = tokenizer(input\_text, return\_tensors="pt").input\_ids

outputs = model.generate(input\_ids)

print(tokenizer.decode(outputs[0]))

A green and pink bar chart

Description automatically generated

input\_text = "Who is the prime minister of India?"

input\_ids = tokenizer(input\_text, return\_tensors="pt").input\_ids

outputs = model.generate(input\_ids)

print(tokenizer.decode(outputs[0]))

<pad> narendra modi</s>

Comparison of Open Source LLM’s vs Closed Source LLMs

Open source and closed source LLMs represent two different approaches to the development and availability of large language models:

**Open Source LLMs:**

Accessibility: Open source LLMs are publicly accessible, and their architecture and parameters can be examined, modified, and shared by the community. This transparency fosters collaboration and innovation.

Community Contributions: They often benefit from contributions and enhancements from a diverse community of researchers and developers, leading to rapid improvements and addressing potential biases.

Customization: Users have the freedom to fine-tune and adapt open source LLMs for specific tasks, languages, or domains, making them highly flexible and versatile.

Cost-Efficiency: Typically, open source LLMs are free to use, which can be particularly advantageous for researchers, startups, and developers.

Examples: Google FLAN-T5, Meta’s Llama models , GPT-3, Hugging Face Transformers are open source and widely accessible.

**Closed Source LLMs:**

Proprietary: Closed source LLMs are developed and owned by organizations or companies, and their architecture and parameters are not publicly disclosed.

Control: Developers of closed source LLMs retain control over their models, algorithms, and intellectual property, allowing them to protect their innovations.

Limited Customization: Users of closed source LLMs may have limited options for fine-tuning or adapting the model to specific needs, as the source code is not openly available.

Costs: Closed source LLMs often come with licensing fees or usage costs, which can be a significant factor for some users or organizations.

Examples: Commercial language models like GPT 3.5 or GPT 4 models and proprietary model used by tech companies may be closed source.

The choice between open source and closed source LLMs depends on factors such as budget, data privacy concerns, customization needs, and the level of control required.

Open source LLMs offer accessibility, collaboration, and cost savings but may require more technical expertise for customization. Closed source LLMs provide intellectual property protection and may come with specialized support and features, but at the cost of limited transparency and potential licensing fees.

Organizations and developers should carefully consider their specific requirements when choosing between these two approaches.

Privacy Attacks on LLM’s

In recent years, large language models have revolutionized natural language understanding and generation, powering a wide range of applications from chatbots and virtual assistants to content recommendation systems and language translation services. However, the rapid advancement of these models has raised significant concerns about privacy and security. As large language models become increasingly prevalent in our digital landscape, there is a growing need for effective strategies to protect sensitive information and uphold user privacy.

As discussed in the earlier chapters, machine learning models are susceptible to privacy attacks and no exception for Generative AI models (Large Language models) as well.

The following two recent articles provides the privacy issues in enterprises with respective to Generative AI.

Cyberhaven’s Survey :

As per the article from Cyberhaven ( https://www.cyberhaven.com/blog/4-2-of-workers-have-pasted-company-data-into-chatgpt/ ) the potential risks of data leaks when employees paste company data into chatbots like OpenAI's GPT-3. The company conducted a survey of 2,000 workers in the US and the UK, and found that 4.2% of them have pasted company data into chatbots. While chatbots like GPT-3 are designed to forget information after the conversation ends, the risk lies in the fact that these chatbots could potentially remember and replicate sensitive information during the conversation. The article also mentions that if a hacker gains control of the chatbot during the conversation, they could access sensitive data. The article emphasizes the need for companies to have clear policies about what data can be shared with chatbots and to educate employees about potential risks. It also suggests that companies should implement data loss prevention (DLP) solutions to automatically block sensitive data from being shared with chatbots. It concludes by stating that while AI chatbots have many benefits, companies need to be aware of potential security and privacy risks and take appropriate measures to protect sensitive data.

Samsung’s IP Leak

As per the article ( <https://techcrunch.com/2023/05/02/samsung-bans-use-of-generative-ai-tools-like-chatgpt-after-april-internal-data-leak/>)

Samsung employees have inadvertently shared confidential information while using ChatGPT for work-related tasks, shedding light on potential privacy and security concerns. Samsung's semiconductor division had allowed engineers to utilize ChatGPT for source code checks and other tasks. However, three separate incidents were reported by The Economist Korea, where sensitive data was unintentionally exposed to ChatGPT.

In one instance, an employee pasted confidential source code into a chat with the intention of checking for errors. Another employee shared code and requested code optimization. A third employee shared a meeting recording for conversion into presentation notes. This information is now accessible to ChatGPT.

In response to the leaks, Samsung has taken immediate action by limiting ChatGPT's upload capacity to 1024 bytes per person. They are also conducting investigations into the individuals involved in the data leaks. Additionally, Samsung is contemplating the development of an in-house AI chatbot to enhance data security and privacy in the future. However, it's important to note that it's unlikely that Samsung will be able to recall the leaked data due to ChatGPT's data policy, which utilizes data for model training unless users explicitly opt out. The usage guide for ChatGPT explicitly warns users against sharing sensitive information during conversations.

These incidents exemplify real-world scenarios that privacy experts have long been concerned about. Such scenarios include sharing confidential legal documents or medical information for text summarization or analysis, which might be used to improve the model. Privacy experts caution that this could potentially violate GDPR compliance, leading to regulatory issues.

Membership inference attacks against generative models

We have learnt about membership inference attack on machine learning models in Chapter 4. Generative AI Modes also are susceptible to Membership inference attacks in a similar lines.

* Generative models estimate the underlying distribution of a dataset to generate realistic samples according to that distribution.
* Membership inference attacks against generative models: given a data point, the adversary determines whether or not it was used to train the model.
* These attacks based on both white-box and black-box access to the target model, against several state-of-the-art generative models

Example:

Source code : MemberShipInference\_LLM.ipynb

This example provides a basic membership inference attack against a generative model using PyTorch. The attack aims to determine if a specific data point was part of the generative model's training dataset. It includes the following components:

Sample Generative AI Model using Variational Autoencoder (VAE):

Steps :

* A simple VAE is used as the generative model. The VAE is capable of encoding and decoding binary data points.
* Adversary Model: An adversary model is implemented as a two-layer feedforward neural network. This model is trained to predict whether a given data point was a member of the training dataset.
* Synthetic Data: Synthetic binary data is generated for demonstration purposes. In practice, you would replace this with your actual dataset.
* Training Process: The VAE and the adversary model are trained independently. The VAE learns to encode and decode data, while the adversary learns to predict membership.
* Membership Inference Attack: The membership inference attack function takes a target data point, encodes it using the VAE, and then uses the adversary model to predict whether the target data point is a member or non-member of the training dataset.

Code Components:

* SampleGenModel Class: Defines the architecture of the Variational Autoencoder.
* Adversary Class: Defines the architecture of the adversary model.
* Data Generation: Generates synthetic binary data for training and testing.
* Training: Training loops for the VAE based SampleGenModel and the adversary.
* Membership Inference Attack: The function for conducting the membership inference attack.
* Main Execution: Initializes the VAE, the adversary, and performs the attack on a target data point.

import torch

import torch.nn as nn

import torch.optim as optim

import numpy as np

# Define a simple generative model

class SampleGenModel(nn.Module):

def \_\_init\_\_(self, input\_dim, hidden\_dim, latent\_dim):

super(SampleGenModel, self).\_\_init\_\_()

self.encoder = nn.Sequential(

nn.Linear(input\_dim, hidden\_dim),

nn.ReLU(),

nn.Linear(hidden\_dim, latent\_dim \* 2) # Two times latent\_dim for mean and log-variance

)

self.decoder = nn.Sequential(

nn.Linear(latent\_dim, hidden\_dim),

nn.ReLU(),

nn.Linear(hidden\_dim, input\_dim),

nn.Sigmoid()

)

def reparameterize(self, mu, log\_var):

std = torch.exp(0.5 \* log\_var)

eps = torch.randn\_like(std)

return mu + eps \* std

def forward(self, x):

x = self.encoder(x)

mu, log\_var = x[:, :latent\_dim], x[:, latent\_dim:]

z = self.reparameterize(mu, log\_var)

reconstructed = self.decoder(z)

return reconstructed, mu, log\_var

# Generate synthetic data for demonstration

num\_samples = 1000

data\_dim = 20

data = torch.tensor(np.random.randint(2, size=(num\_samples, data\_dim)), dtype=torch.float32)

print(data)

# Initialize the SampleGenModel

input\_dim = data\_dim

hidden\_dim = 64

latent\_dim = 16

vae = SampleGenModel(input\_dim, hidden\_dim, latent\_dim)

# Define an adversary model (a simple feedforward neural network)

class Adversary(nn.Module):

def \_\_init\_\_(self, input\_dim):

super(Adversary, self).\_\_init\_\_()

self.fc = nn.Sequential(

nn.Linear(input\_dim, 32),

nn.ReLU(),

nn.Linear(32, 1),

nn.Sigmoid()

)

def forward(self, x):

return self.fc(x)

# Train the SampleGenModel

# Train the adversary model

adversary = Adversary(latent\_dim)

optimizer = optim.Adam(adversary.parameters(), lr=0.001)

criterion = nn.BCELoss()

# Prepare target data for the membership inference attack

target\_data\_point = torch.tensor(np.random.randint(2, size=data\_dim), dtype=torch.float32)

# Membership inference attack function

def membership\_inference\_attack(vae, adversary, target\_data\_point):

# Encode the target data point using the VAE

with torch.no\_grad():

target\_data\_point = target\_data\_point.unsqueeze(0) # Add batch dimension

reconstructed, mu, log\_var = vae(target\_data\_point)

# Use the adversary to predict membership

prediction = adversary(mu)

# If the prediction is close to 1, the target data point is likely a member

if prediction.item() > 0.5:

return "Member"

else:

return "Non-Member"

# Perform the membership inference attack

result = membership\_inference\_attack(vae, adversary, target\_data\_point)

# Output the result

print("Membership Inference Result:", result)

tensor([[0., 0., 1., ..., 1., 0., 1.],

[0., 1., 1., ..., 0., 0., 1.],

[1., 0., 1., ..., 1., 0., 1.],

...,

[0., 0., 0., ..., 1., 0., 0.],

[0., 1., 0., ..., 0., 1., 1.],

[1., 0., 1., ..., 0., 0., 0.]])

Membership Inference Result: Member

Membership inference attacks can be more complex in practice, and this code serves as a

basic demonstration. Implement privacy and security measures when deploying generative models to protect against such attacks. We will cover in detail how to protect generative AI models in privacy preserving manner in the next section.

Extracting Training Data attack from generative models

Extracting training data from large language models (LLMs) can be a challenging task

because the training data is not typically available directly from the model. Instead, LLMs

are pretrained on vast datasets from the internet. If we have a specific LLM in mind and

want to extract training data related to it, we may need access to the original data sources used for pretraining, which may not be publicly available.

Sample Python code snippet that demonstrates how we can extract text data from a

pretrained Hugging Face Transformers model, like GPT-2. Keep in mind that this code is for

illustrative purposes and won't retrieve the actual training data but rather generates text

samples from the model:

In this code:

* We load a pretrained GPT-2 model and tokenizer from the Hugging Face Transformers

library. You can choose other models based on your requirements.

* We define a prompt, which serves as the starting point for generating text. You can change the prompt to suit your needs.
* We specify the number of text samples (num\_samples) to generate from the model.
* Inside the loop, we encode the prompt using the tokenizer and generate text sequences using the model. We decode the output to obtain human-readable text.

Please note that the generated text is not actual training data used for the model but rather synthetic text produced by the model based on the provided prompt. To access the actual

training data used to train LLMs, you would need access to the original data sources, which are typically large and diverse web corpora.

Source code: Training\_Data\_Extraction\_Gen\_AI.ipynb

!pip install torch

!pip install transformers

import torch

from transformers import GPT2LMHeadModel, GPT2Tokenizer

# Load a pretrained GPT-2 model and tokenizer

model\_name = "gpt2" # You can choose other pretrained models as well

model = GPT2LMHeadModel.from\_pretrained(model\_name)

tokenizer = GPT2Tokenizer.from\_pretrained(model\_name)

# Generate text samples from the model

prompt = "Once upon a time"

num\_samples = 5

for \_ in range(num\_samples):

input\_ids = tokenizer.encode(prompt, return\_tensors="pt")

output = model.generate(input\_ids, max\_length=100, num\_return\_sequences=1, no\_repeat\_ngram\_size=2)

generated\_text = tokenizer.decode(output[0], skip\_special\_tokens=True)

print("Generated Text:\n", generated\_text)

print("="\*80)

A screenshot of a computer

Description automatically generated

Researchers from Google, Apple, Open AI, Harvard, Sandford team) are demonstrated attack on GPT-2, a language model trained on scrapes of the public Internet, and are able to

extract hundreds of verbatim text sequences from the model's training data. These extracted examples include (public) personally identifiable information (names, phone numbers, and email addresses). URL: <https://arxiv.org/pdf/2012.07805.pdf>

.

Prompt Injection attacks

A prompt injection attack, also known as data or command injection, is a type of security vulnerability that happens when an attacker can influence the prompts or commands sent to a data processing system, such as a large language model (LLM). These attacks potentially allow attackers to manipulate the actions of the system or extract sensitive or private data.

In the context of an LLM, prompt injection attacks could involve an attacker providing a crafted input designed to trick the model into providing information it has been trained on, which could potentially include sensitive or confidential information if the training data was not properly anonymized or scrubbed.

Moreover, an attacker can inject malicious prompts to make the model produce outputs that inflict harm, such as generating offensive, defamatory, or illegal content. This could be used for spear phishing, spreading of disinformation, defaming individuals or entities, and many other nefarious purposes.

Currently, the extent to which LLMs are vulnerable to these attacks isn't fully understood. It's also worth mentioning that these models are designed not to directly recall any specifics about their training data, including documents or sources they were trained on, and they generally don't have the ability to access or retrieve personal data unless they've been explicitly programmed to do so, or they've been trained on data which contains sensitive personal information.

Nonetheless, it's always crucial to approach the use of LLMs, or any AI system, with robust security measures and an understanding of potential risks.

Example: PromptInjection.ipynb

class SimpleModel:

def \_\_init\_\_(self):

self.data={

'SSN':'123-45-6789',

'email':'example@example.com',

'password':'mypassword'

}

def generate\_text(self,prompt):

return self.data.get(prompt,'Sorry,I don\'t have the data')

model=SimpleModel()

## Normal Request

print(model.generate\_text('favorite\_color'))

## Malicious request , simulating an attempt to a prompt injection attack

print(model.generate\_text('SSN'))

Sorry,I don't have the data

123-45-6789

Privacy Preserving Technologies for LLM’s

Differential Privacy is one of the privacy preserving technology which can be used for Large Language Models as well.

Text Attacks on Machine Learning and LLM Models

TextAttack stands as a Python framework designed for conducting adversarial attacks, adversarial training, and data augmentation within the field of Natural Language Processing (NLP). This versatile tool streamlines the process of exploring NLP model robustness, offering a seamless, rapid, and user-friendly experience. Furthermore, it proves invaluable for NLP model training, adversarial training, and data augmentation purposes. TextAttack offers various components tailored for typical NLP tasks, including sentence encoding, grammar checking, and word replacement, which can also be utilized independently

Installation of TextAttack package.

Github Location : <https://github.com/QData/TextAttack>

!pip install textattack

TextAttack provides various recipes to attack on NLP modules.

In the below example, utilizes various libraries and components to perform adversarial attacks on Natural Language Processing (NLP) models using the TextAttack framework.

Source code : Privacy\_attacks\_LLMs.ipynb

High Level steps in the implementation of this example:

Step 0 : Importing Libraries: import the necessary libraries, including transformers from Hugging Face, torch for PyTorch, math, textattack, and random.

Step 1: Environment Setup: It sets the CUDA\_VISIBLE\_DEVICES environment variable to an empty string, essentially disabling GPU usage. It specifies the device to be used as "cpu" for PyTorch operations.

Step 2: Model Definition: Defines a custom PyTorch model called Model. This model uses BERT (Bidirectional Encoder Representations from Transformers) for NLP tasks. The model loads the pre-trained 'bert-base-uncased' BERT model from Hugging Face's Transformers library. It includes a dropout layer and a linear layer for classification.

Step3: Initialization of Model and Tokenization

Model Initialization: An instance of the Model class is created and moved to the CPU for evaluation. The model is set to evaluation mode using model.eval().

Tokenizer Initialization:

Initializes a BERT tokenizer (BertTokenizer) for tokenizing text.

Step 4: Custom Model Wrapper:

Defines a custom model wrapper class called CustomWrapper that wraps the PyTorch model. This wrapper allows the model to be used with the TextAttack library.

Attack Configuration:

Step 5: Utilizes the TextAttack library to build an attack using the TextFoolerJin2019 recipe.

The CustomWrapper instance is passed to the attack.

Dataset: Defines a list called dataset, containing text samples and corresponding labels.

These samples are examples for performing adversarial attacks.

Attack Execution: Creates an Attacker instance, specifying the attack, dataset, and other attack parameters. Finally, the attack\_dataset() method is called on the attacker to perform the adversarial attacks on the dataset.

Overall, this code sets up a PyTorch model, initializes an attack using the TextAttack library, and then applies this attack to a dataset of text samples for the purpose of evaluating the robustness of the NLP model.

import pandas as pd

import os

from transformers import BertTokenizer, BertModel

from torch import nn

import torch

import math

import textattack

import random

#from train\_bert import Model

os.environ["CUDA\_VISIBLE\_DEVICES"] = ""

#torch.cuda.is\_available = lambda : False

textattack.shared.utils.device = "cpu"

class Model(torch.nn.Module):

def \_\_init\_\_(self):

super(Model, self).\_\_init\_\_()

self.bert\_model = BertModel.from\_pretrained('bert-base-uncased')

#self.bert\_model.parallelize()

self.drop = torch.nn.Dropout(p=0.1)

self.l1 = torch.nn.Linear(768,2)

def forward(self, text):

tokenized\_text = tokenizer(text , max\_length=512, truncation=True, return\_tensors='pt').input\_ids#.to('cuda:3')

text\_rep = self.drop(self.bert\_model(tokenized\_text).pooler\_output)

out = self.l1(text\_rep)

#print(out)

return out.squeeze().tolist()

model = Model()

#model.load\_state\_dict(torch.load('bert-base-uncased'))

model = model.to('cpu')

model.eval()

tokenizer = BertTokenizer.from\_pretrained('bert-base-uncased')

class CustomWrapper(textattack.models.wrappers.ModelWrapper):

def \_\_init\_\_(self, model):

self.model = model#.to('cuda:3')

self.model.eval()

def \_\_call\_\_(self, list\_of\_texts):

results = []

self.model.requires\_grad = False

for text in list\_of\_texts:

results.append(self.model(text))

return results

class\_model = CustomWrapper(model)

from textattack.datasets import Dataset

from textattack.attack\_recipes.textfooler\_jin\_2019 import TextFoolerJin2019

from textattack import Attacker, AttackArgs

attack = TextFoolerJin2019.build(class\_model)

attack#.cuda\_()

dataset = [

["This film is a masterpiece! The story is incredibly moving, and the performances are outstanding. It's a true classic.", 1],

["The Godfather is a cinematic gem. The storytelling and performances are top-notch. A true classic in every sense.", 1],

["The Emoji Movie is a complete disappointment. The plot is weak, and it feels like one big advertisement. A waste of time.", 0],

["Mind-bending and visually stunning! Inception keeps you guessing from start to finish. Christopher Nolan at his best.", 1],

["Twilight is a guilty pleasure for some, but the acting and dialogue are cringe-worthy. Not a cinematic masterpiece.", 0],

["Forrest Gump is a heartwarming journey through history. Tom Hanks delivers an unforgettable performance.", 1],

["Explosions and CGI can't make up for the lackluster story in Transformers: The Last Knight. Disappointing.", 0],

["The Dark Knight is a dark and gripping superhero film. Heath Ledger's Joker is iconic. A must-see.", 1],

["Avatar is visually breathtaking, but the story is somewhat predictable. Still, it's a cinematic experience.", 1],

["The Room is so bad that it's almost good. The unintentional humor makes it a cult classic.", 1]

]

random.shuffle(dataset)

attacker = Attacker(attack, textattack.datasets.Dataset(dataset[:10]), AttackArgs(num\_examples=10))

attacker.attack\_dataset()

Output:

textattack: Unknown if model of class <class '\_\_main\_\_.Model'> compatible with goal function <class 'textattack.goal\_functions.classification.untargeted\_classification.UntargetedClassification'>.

Attack(

(search\_method): GreedyWordSwapWIR(

(wir\_method): delete

)

(goal\_function): UntargetedClassification

(transformation): WordSwapEmbedding(

(max\_candidates): 50

(embedding): WordEmbedding

)

(constraints):

(0): WordEmbeddingDistance(

(embedding): WordEmbedding

(min\_cos\_sim): 0.5

(cased): False

(include\_unknown\_words): True

(compare\_against\_original): True

)

(1): PartOfSpeech(

(tagger\_type): nltk

(tagset): universal

(allow\_verb\_noun\_swap): True

(compare\_against\_original): True

)

(2): UniversalSentenceEncoder(

(metric): angular

(threshold): 0.840845057

(window\_size): 15

(skip\_text\_shorter\_than\_window): True

(compare\_against\_original): False

)

(3): RepeatModification

(4): StopwordModification

(5): InputColumnModification(

(matching\_column\_labels): ['premise', 'hypothesis']

(columns\_to\_ignore): {'premise'}

)

(is\_black\_box): True

)

0%| | 0/10 [00:00<?, ?it/s]

10%|█ | 1/10 [00:48<07:13, 48.12s/it]

[Succeeded / Failed / Skipped / Total] 0 / 1 / 0 / 1: 10%|█ | 1/10 [00:48<07:13, 48.12s/it]

[Succeeded / Failed / Skipped / Total] 0 / 1 / 0 / 1: 20%|██ | 2/10 [00:48<03:12, 24.11s/it]

[Succeeded / Failed / Skipped / Total] 0 / 1 / 1 / 2: 20%|██ | 2/10 [00:48<03:12, 24.12s/it]

--------------------------------------------- Result 1 ---------------------------------------------

[[0 (56%)]] --> [[[FAILED]]]

Explosions and CGI can't make up for the lackluster story in Transformers: The Last Knight. Disappointing.

--------------------------------------------- Result 2 ---------------------------------------------

[[0 (55%)]] --> [[[SKIPPED]]]

Forrest Gump is a heartwarming journey through history. Tom Hanks delivers an unforgettable performance.

[Succeeded / Failed / Skipped / Total] 0 / 1 / 1 / 2: 30%|███ | 3/10 [00:48<01:52, 16.11s/it]

[Succeeded / Failed / Skipped / Total] 0 / 1 / 2 / 3: 30%|███ | 3/10 [00:48<01:52, 16.11s/it]

--------------------------------------------- Result 3 ---------------------------------------------

[[0 (57%)]] --> [[[SKIPPED]]]

The Room is so bad that it's almost good. The unintentional humor makes it a cult classic.

[Succeeded / Failed / Skipped / Total] 0 / 1 / 2 / 3: 40%|████ | 4/10 [01:01<01:32, 15.47s/it]

[Succeeded / Failed / Skipped / Total] 0 / 2 / 2 / 4: 40%|████ | 4/10 [01:01<01:32, 15.47s/it]

[Succeeded / Failed / Skipped / Total] 0 / 2 / 2 / 4: 50%|█████ | 5/10 [01:01<01:01, 12.40s/it]

[Succeeded / Failed / Skipped / Total] 0 / 2 / 3 / 5: 50%|█████ | 5/10 [01:01<01:01, 12.40s/it]

--------------------------------------------- Result 4 ---------------------------------------------

[[0 (58%)]] --> [[[FAILED]]]

Twilight is a guilty pleasure for some, but the acting and dialogue are cringe-worthy. Not a cinematic masterpiece.

--------------------------------------------- Result 5 ---------------------------------------------

[[0 (54%)]] --> [[[SKIPPED]]]

The Godfather is a cinematic gem. The storytelling and performances are top-notch. A true classic in every sense.

[Succeeded / Failed / Skipped / Total] 0 / 2 / 3 / 5: 60%|██████ | 6/10 [01:27<00:58, 14.51s/it]

--------------------------------------------- Result 6 ---------------------------------------------

[Succeeded / Failed / Skipped / Total] 1 / 2 / 3 / 6: 60%|██████ | 6/10 [01:27<00:58, 14.60s/it]

[Succeeded / Failed / Skipped / Total] 1 / 2 / 3 / 6: 70%|███████ | 7/10 [01:27<00:37, 12.54s/it]

[Succeeded / Failed / Skipped / Total] 1 / 2 / 4 / 7: 70%|███████ | 7/10 [01:27<00:37, 12.54s/it]

[[0 (59%)]] --> [[1 (51%)]]

[[The]] Emoji Movie is a [[complete]] [[disappointment]]. The [[plot]] is weak, and it [[feels]] like one [[big]] [[advertisement]]. [[A]] [[waste]] of time.

[[Both]] Emoji Movie is a [[fulfills]] [[dissatisfaction]]. The [[plots]] is weak, and it [[reckon]] like one [[largest]] [[commercials]]. [[para]] [[vandalize]] of time.

--------------------------------------------- Result 7 ---------------------------------------------

[[0 (56%)]] --> [[[SKIPPED]]]

Avatar is visually breathtaking, but the story is somewhat predictable. Still, it's a cinematic experience.

[Succeeded / Failed / Skipped / Total] 1 / 2 / 4 / 7: 80%|████████ | 8/10 [01:27<00:21, 10.99s/it]

[Succeeded / Failed / Skipped / Total] 1 / 2 / 5 / 8: 80%|████████ | 8/10 [01:27<00:21, 10.99s/it]

[Succeeded / Failed / Skipped / Total] 1 / 2 / 5 / 8: 90%|█████████ | 9/10 [01:28<00:09, 9.78s/it]

[Succeeded / Failed / Skipped / Total] 1 / 2 / 6 / 9: 90%|█████████ | 9/10 [01:28<00:09, 9.78s/it]

--------------------------------------------- Result 8 ---------------------------------------------

[[0 (54%)]] --> [[[SKIPPED]]]

Mind-bending and visually stunning! Inception keeps you guessing from start to finish. Christopher Nolan at his best.

--------------------------------------------- Result 9 ---------------------------------------------

[[0 (55%)]] --> [[[SKIPPED]]]

The Dark Knight is a dark and gripping superhero film. Heath Ledger's Joker is iconic. A must-see.

[Succeeded / Failed / Skipped / Total] 1 / 2 / 6 / 9: 100%|██████████| 10/10 [01:28<00:00, 8.82s/it]

[Succeeded / Failed / Skipped / Total] 1 / 2 / 7 / 10: 100%|██████████| 10/10 [01:28<00:00, 8.82s/it]

--------------------------------------------- Result 10 ---------------------------------------------

[[0 (55%)]] --> [[[SKIPPED]]]

This film is a masterpiece! The story is incredibly moving, and the performances are outstanding. It's a true classic.

+-------------------------------+--------+

| Attack Results | |

+-------------------------------+--------+

| Number of successful attacks: | 1 |

| Number of failed attacks: | 2 |

| Number of skipped attacks: | 7 |

| Original accuracy: | 30.0% |

| Accuracy under attack: | 20.0% |

| Attack success rate: | 33.33% |

| Average perturbed word %: | 40.91% |

| Average num. words per input: | 17.3 |

| Avg num queries: | 213.33 |

+-------------------------------+--------+

In the similar way, GPT2 model also can be explored for NLP attacks. In the source code, the second model does the same with GPT2 module and the following are the results.

*-------------------------------+-------+*

*| Attack Results | |*

*+-------------------------------+-------+*

*| Number of successful attacks: | 0 |*

*| Number of failed attacks: | 3 |*

*| Number of skipped attacks: | 7 |*

*| Original accuracy: | 30.0% |*

*| Accuracy under attack: | 30.0% |*

*| Attack success rate: | 0.0% |*

*| Average perturbed word %: | nan% |*

*| Average num. words per input: | 17.3 |*

*| Avg num queries: | 250.0 |*

*+-------------------------------+-------+*

Private Transformers : Training Large Language Models using Differential Privacy

Github Location : <https://github.com/lxuechen/private-transformers>

Xuechen Li and Florian Tramer and Percy Liang and Tatsunori Hashimoto et all provided private transformers to train large launge models using differential privacy

They have modified the Opacus framework and integrated with Hugging Face transformers library and provided Privacy Engine to train the LLM’s in privacy preserving manner. Using this codebase, they successfully fine-tuned exceptionally large pretrained models, achieving some of the most impressive differentially private Natural Language Processing (NLP) results to date. In fact, certain models have exhibited performance comparable to robust non-private baseline approaches. This provides compelling empirical support for the notion that highly effective differentially private NLP models can be constructed even with relatively modest dataset. Furthermore, support for the ghost clipping technique which enables the private training of large transformers with significantly reduced memory requirements. In many instances, the memory footprint is nearly as lightweight as non-private training, with only a modest increase in runtime overhead. Private Transformers currently supports the following LLM’s only.

* [OpenAIGPTLMHeadModel](https://huggingface.co/docs/transformers/model_doc/openai-gpt#transformers.OpenAIGPTLMHeadModel)
* [OpenAIGPTDoubleHeadsModel](https://huggingface.co/docs/transformers/model_doc/openai-gpt#transformers.OpenAIGPTDoubleHeadsModel)
* [GPT2LMHeadModel](https://huggingface.co/docs/transformers/model_doc/gpt2#transformers.GPT2LMHeadModel)
* [GPT2DoubleHeadsModel](https://huggingface.co/docs/transformers/model_doc/gpt2#transformers.GPT2DoubleHeadsModel)
* [BertForSequenceClassification](https://huggingface.co/docs/transformers/model_doc/bert#transformers.BertForSequenceClassification)
* [RobertaForSequenceClassification](https://huggingface.co/docs/transformers/model_doc/roberta#transformers.RobertaForSequenceClassification)
* [AlbertForSequenceClassification](https://huggingface.co/docs/transformers/model_doc/albert#transformers.AlbertForSequenceClassification)
* [BartForConditionalGeneration](https://huggingface.co/docs/transformers/model_doc/bart#transformers.BartForConditionalGeneration)
* [T5ForConditionalGeneration](https://huggingface.co/docs/transformers/v4.20.1/en/model_doc/t5#transformers.T5ForConditionalGeneration)
* [OPTForCausalLM](https://huggingface.co/docs/transformers/model_doc/opt#transformers.OPTForCausalLM)
* [ViTForImageClassification](https://huggingface.co/docs/transformers/v4.20.1/en/model_doc/vit#transformers.ViTForImageClassification)
* [DeiTForImageClassification](https://huggingface.co/docs/transformers/model_doc/deit#transformers.DeiTForImageClassification)
* [BeitForImageClassification](https://huggingface.co/docs/transformers/model_doc/beit#transformers.BeitForImageClassification)

Privately training Hugging Face transformers simply consists of 4 steps:

1. Create your favourite transformer model and optimizer; attach this optimizer to a PrivacyEngine
2. Compute a per-example loss (1-D tensor) for a mini-batch of data
3. Pass the loss to optimizer.step or optimizer.virtual\_step as a keyword argument
4. Repeat from step 2

Example :

The code is designed for training a Language Model with privacy-preserving features. It utilizes the Hugging Face Transformers library and PyTorch. Below is a detailed steps to implement.

Steps

* Libraries and Imports
* Dataset Class
* Loading Data from a Text File
* Forward Step
* Training Function
* Running the Training

Step 0: importing the necessary libraries and modules. These include:

* tqdm: A library for displaying progress bars during training.
* transformers: A library for working with transformer-based models.
* torch: The PyTorch library for deep learning.
* GPT2Tokenizer and GPT2LMHeadModel from transformers: These classes provide access to the GPT-2 model and tokenizer.
* PrivacyEngine from private\_transformers: A custom privacy engine for training the model with privacy constraints.

Step 1: Dataset Class

A custom Dataset class is defined to handle the training data. This class has the following methods:

* \_\_init\_\_(self, texts, labels, eos\_token): Initializes the dataset with texts, labels, and an end-of-sequence token (eos\_token).
* \_\_len\_\_(self): Returns the length of the dataset.
* \_\_getitem\_\_(self, index): Retrieves a specific text and its corresponding label at the given index.

Step 2: Loading Data from a Text File

The get\_data\_from\_txt(path) function is used to load text data and labels from a text file. Each line in the file contains a label followed by a text. This function reads the file, extracts the labels and texts, and returns them as lists.

Step 3: Forward Step

The forward\_step(correct\_texts, wrong\_texts, tokenizer, model, mismatch\_loss, mismatch\_weight) function performs a forward step during training. It takes a list of correct and incorrect texts, a tokenizer, the model, and parameters for mismatch loss and mismatch weight. It tokenizes the texts, calculates the language modeling loss, and applies mismatch loss if specified. The result is a loss tensor.

Step 4: Training Function

The train\_llm(args\_model\_out, return\_results, train\_data, train\_loader) function trains the Language Model. It initializes the GPT-2 model, tokenizer, optimizer, and privacy engine. Privacy budget (epsilon) value as 0.5 used in this example, but it can be changed to the desired privacy budget. It then iterates over training epochs, processing data in batches and calculating losses. The model is saved at the end of each epoch.

Step 4: Running the Training

At the end of the code, a sample dataset is loaded from a text file, and the training process is initiated using the train\_llm function. The function takes parameters such as the output path for saving the model, whether to return results, the training data, and the data loader.

Source Code : Privacy\_attacks\_LLMs.ipynb

!pip install transformers

!pip install git+https://github.com/lxuechen/private-transformers.git

!pip install tqdm

from tqdm import tqdm

import transformers

import torch

from transformers import GPT2Tokenizer, GPT2LMHeadModel

**from private\_transformers import PrivacyEngine**

class Dataset(torch.utils.data.Dataset):

def \_\_init\_\_(self, texts, labels, eos\_token):

self.texts = texts

self.y = labels

self.eos\_token = eos\_token

def \_\_len\_\_(self):

return len(self.texts)

def \_\_getitem\_\_(self, index):

text = self.texts[index] + ' ' + self.eos\_token

label = self.y[index]

return text, label

def get\_data\_from\_txt(path: str):

texts = []

labels = []

with open(path, 'r') as f:

for line in f:

texts.append(' '.join(line.split(' ')[1:]).replace('\n', ''))

labels.append(int(line.split(' ')[0]))

return texts, labels

def forward\_step(texts,tokenizer, model):

tokenized\_texts = tokenizer(texts, truncation=True, max\_length=500, return\_tensors='pt', padding=True).input\_ids.to('cpu')

lm\_loss = model(tokenized\_texts, labels=tokenized\_texts).loss.unsqueeze(dim=0)

return lm\_loss

def train\_llm(train\_data, train\_loader,

):

model = GPT2LMHeadModel.from\_pretrained("gpt2")

#model.parallelize()

model.train()

tokenizer = GPT2Tokenizer.from\_pretrained("gpt2")

tokenizer.pad\_token = tokenizer.eos\_token

optimizer = torch.optim.Adam(model.parameters(), lr = 8e-6)

args\_epochs=2

print(args\_epochs)

epsilon=0.5

print(epsilon)

**privacy\_engine = PrivacyEngine(**

**model,**

**batch\_size=1,**

**sample\_size=10,**

**epochs=args\_epochs,**

**max\_grad\_norm=0.1,**

**target\_epsilon=epsilon,**

**)**

**privacy\_engine.attach(optimizer)**

for epoch in range(args\_epochs):

print(f'training epoch {epoch}')

total\_loss = 0

for texts, labels in tqdm(train\_loader):

lm\_loss = forward\_step(texts,tokenizer, model)

optimizer.step(loss=lm\_loss)

total\_loss += lm\_loss.item()

print('total language modelling loss', total\_loss/len(train\_data)

print()

print('model training done!')

print()

return model

train\_texts, train\_labels = get\_data\_from\_txt('imdb\_train.txt')

train\_texts = train\_texts[0:100]

train\_labels =train\_labels[0:100]

train\_data = Dataset(train\_texts, train\_labels, '<|endoftext|>')

train\_loader = torch.utils.data.DataLoader(train\_data, shuffle=False, batch\_size=1)

pmodel = train\_llm(train\_data,train\_loader)

print(pmodel)

2

0.5

training epoch 0

0%| | 0/100 [00:00<?, ?it/s]/opt/conda/lib/python3.10/site-packages/torch/nn/modules/module.py:1344: UserWarning: Using a non-full backward hook when the forward contains multiple autograd Nodes is deprecated and will be removed in future versions. This hook will be missing some grad\_input. Please use register\_full\_backward\_hook to get the documented behavior.

warnings.warn("Using a non-full backward hook when the forward contains multiple autograd Nodes "

100%|██████████| 100/100 [03:27<00:00, 2.07s/it]

total language modeling loss 4.433933084011078

training epoch 1

100%|██████████| 100/100 [03:25<00:00, 2.06s/it]

total language modelling loss 3.9711666679382325

model training done!

GPT2LMHeadModel(

(transformer): GPT2Model(

(wte): Embedding(50257, 768)

(wpe): Embedding(1024, 768)

(drop): Dropout(p=0.1, inplace=False)

(h): ModuleList(

(0-11): 12 x GPT2Block(

(ln\_1): LayerNorm((768,), eps=1e-05, elementwise\_affine=True)

(attn): GPT2Attention(

(c\_attn): Conv1D()

(c\_proj): Conv1D()

(attn\_dropout): Dropout(p=0.1, inplace=False)

(resid\_dropout): Dropout(p=0.1, inplace=False)

)

(ln\_2): LayerNorm((768,), eps=1e-05, elementwise\_affine=True)

(mlp): GPT2MLP(

(c\_fc): Conv1D()

(c\_proj): Conv1D()

(act): NewGELUActivation()

(dropout): Dropout(p=0.1, inplace=False)

)

)

)

(ln\_f): LayerNorm((768,), eps=1e-05, elementwise\_affine=True)

)

(lm\_head): Linear(in\_features=768, out\_features=50257, bias=False)

)

STOA - Privacy Preserving Technologies for LLM’s

The following section provides the high level state of the art research work on privacy preserving technologies for Large Language Models. This is not exhaustive list but the current trends in the research.

Prompts – Privacy - Differentially Private Prompt Learning for LLMs

Research Article : <https://arxiv.org/pdf/2305.15594.pdf>

Large language models (LLMs) excel at learning in-context information, but concerns arise regarding the privacy implications of the data within prompts. This research validates these concerns by demonstrating a straightforward yet highly effective membership inference attack on the data used for LLM prompts. To mitigate this vulnerability, one option is to abandon prompting and opt for fine-tuning LLMs using established algorithms for private gradient descent. However, this sacrifices the practicality and efficiency provided by the prompting approach. Therefore, we propose a novel solution: private prompt learning. They initially demonstrate the feasibility of obtaining soft prompts privately through gradient descent on downstream data. However, the challenge lies in dealing with discrete prompts. To address this, we orchestrate a process wherein an ensemble of LLMs is engaged with various prompts, akin to a flock of stochastic parrots. A noisy vote among these LLMs privately transfers the collective knowledge of the flock into a single public prompt. Their results illustrate that LLMs prompted using our private algorithms closely approach the performance of their non-private counterparts. For instance, when employing GPT-3 as the base model, we attain a downstream accuracy of 92.7% on the sst2 dataset with (ε = 0.147, δ = 10-6)-differential privacy, compared to 95.2% for the non-private baseline.

Prompts – Privacy – Encrypted Prompt

Research Article : <https://arxiv.org/abs/2305.18396>

In this paper, researchers demonstrated that by substituting the computation- and communication-intensive operators within the transformer architecture with privacy-computing-friendly approximations, significantly decrease the costs associated with private inference while only minimally affecting model performance. In comparison to the state-of-the-art Iron framework (NeurIPS 2022), our privacy-computing-friendly model inference pipeline achieves a 5× acceleration in computation and an 80% reduction in communication overhead, all while maintaining nearly identical accuracy.

Differentially Private Attention Computation

Research Article : <https://arxiv.org/abs/2305.04701>

The attention mechanism plays a crucial role in Large Language Models (LLMs), enabling them to selectively focus on various segments of input text. Computing the attention matrix is a well-recognized and substantial task in the LLM computation process. Consequently, determining how to offer verifiable privacy guarantees for the computation of the attention matrix is a significant research avenue. One natural mathematical concept for quantifying privacy, as found in theoretical computer science graduate textbooks, is differential privacy.

In this study, inspired by the work of [Vyas, Kakade, and Barak 2023], researchers present a provable outcome that demonstrates how to differentially privately approximate the attention matrix. From a technical perspective, the results draw upon pioneering research in the realm of differential privacy as established by [Alabi, Kothari, Tankala, Venkat, and Zhang 2022]."

**Differentially Private Decoding in Large Language Models**

Research Article : <https://arxiv.org/abs/2205.13621>

Researchers presented a straightforward, easily interpretable, and computationally efficient perturbation technique designed for implementation during the decoding phase of a pre-trained model. This perturbation mechanism is model-agnostic, compatible with any Large Language Model (LLM).Their work includes a theoretical analysis demonstrating the differential privacy properties of the proposed mechanism, along with experimental results illustrating the trade-off between privacy and utility."

**Differentially Private Model Compression**

Research Article : <https://arxiv.org/abs/2206.01838>

Large pre-trained language models (LLMs) have demonstrated the ability to undergo fine-tuning on private data, achieving performance levels comparable to non-private models across numerous downstream Natural Language Processing (NLP) tasks while ensuring differential privacy. However, these models, comprising hundreds of millions of parameters, often incur prohibitively high inference costs. Therefore, in practical applications, LLMs are frequently subjected to compression before deployment. Researchers embark on the exploration of differentially private model compression and propose frameworks capable of achieving 50% sparsity levels while retaining nearly full performance. Their study includes practical demonstrations on standard GLUE benchmarks using BERT models, thus establishing benchmarks for future research in this field.