THARUN NAIK RAMAVATH

Gen Ai Fundamentals

11 December 2024 20:32

What is Generative Ai:

Generative AI is an exciting field of artificial intelligence that opens the door to creating new and original content, spanning from written text to stunning visuals and even computer-generated music. It showcases the innovative side of AI by going beyond simple analytical tasks to engage in creative processes.

Technical Terms Explained:
Text Generation: This involves making computers write text that makes sense and is relevant to the topic, akin to an automatic storyteller.

Image Generation: This allows computers to make new pictures or change existing ones, like a digital artist using a virtual paintbrush.

Code Generation: This is Gen AI for programming, where the computer helps write new code.

Audio Generation: Computers can also create sounds or music, a bit like a robot composer coming up with its own tunes.

Chat GPT: A language model developed by OpenAI that can generate responses similar to those a human would give in a conversation by predicting the next word in a sequence based on context.

DALL-E: An AI program by OpenAI that produces images from textual descriptions, mimicking creativity in visual art.

GitHub Copilot: A coding assistant tool that suggests code snippets and completes code lines to help developers write more efficiently and with fewer errors.

Contextual Suggestions: Recommendations provided by AI tools, like Copilot, which are relevant to the current task or context within which a user is working.

Perceptron: An early type of neural network component that can decide whether or not an input, represented by numerical values, belongs to a specific class.

Neural Networks: Computer systems modeled after the human brain that can learn from data by adjusting the connections between artificial neurons.

Back Propagation: A method used in artificial neural networks to improve the model by adjusting the weights by calculating the gradient of the loss function.

Statistical Machine Learning: A method of data analysis that automates analytical model building using algorithms that learn from data without being explicitly programmed.

Deep Learning: A subset of machine learning composed of algorithms that permit software to train itself to perform tasks by exposing multilayered neural networks to vast amounts of data.

Generative Adversarial Networks (GANs): A class of machine learning models where two networks, a generator and a discriminator, are trained simultaneously in a zero-sum game framework.

Transformer: A type of deep learning model that handles sequential data and is particularly noted for its high performance in natural language processing tasks.

Large Language Models (LLMs): These are AI models specifically designed to understand and generate human language by being trained on a vast amount of text data.

Variational Autoencoders (VAEs): A type of AI model that can be used to create new images. It has two main parts: the encoder reduces data to a simpler form, and the decoder expands it back to generate new content.

Latent Space: A compressed representation of data that the autoencoder creates in a simpler, smaller form, which captures the most important features needed to reconstruct or generate new data.

Parameters: Parameters are the variables that the model learns during training. They are internal to the model and are adjusted through the learning process. In the context of neural networks, parameters typically include weights and biases.

Weights: Weights are coefficients for the input data. They are used in calculations to determine the importance or influence of input variables on the model's output. In a neural network, each connection between neurons has an associated weight.

Biases (not mentioned in the video): Biases are additional constants attached to neurons and are added to the weighted input before the activation function is applied. Biases ensure that even when all the inputs are zero, there can still be a non-zero output.

Hyperparameters: Hyperparameters, unlike parameters, are not learned from the data. They are more like settings or configurations for the learning process. They are set prior to the training process and remain constant during training. They are external to the model and are used to control the learning process.

Autoregressive text generation: Autoregressive text generation is like a game where the computer guesses the next word in a sentence based on the words that came before it. It keeps doing this to make full sentences.

Latent space decoding: Imagine if you had a map of all the possible images you could create, with each point on the map being a different image. Latent space decoding is like picking a point on that map and bringing the image at that point to life.

Diffusion models: Diffusion models start with a picture that's full of random dots like TV static, and then they slowly clean it up, adding bits of the actual picture until it looks just like a real photo or painting.

Perceptron: A basic computational model in machine learning that makes decisions by weighing input data. It's like a mini-decision maker that labels data as one thing or another.

Binary Classifier: A type of system that categorizes data into one of two groups. Picture a light switch that can be flipped to either on or off.

Vector of Numbers: A sequence of numbers arranged in order, which together represent one piece of data.

Activation Function: A mathematical equation that decides whether the perceptron's calculated sum from the inputs is enough to trigger a positive or negative output.it basically decide whether a neuron should be activated or not.

Multi-Layer Perceptron (MLP): A type of artificial neural network that has multiple layers of nodes, each layer learning to recognize increasingly complex features of the input data.

Input Layer: The first layer in an MLP where the raw data is initially received.

Output Layer: The last layer in an MLP that produces the final result or prediction of the network.

Hidden Layers: Layers between the input and output that perform complex data transformations.

Labeled Dataset: This is a collection of data where each piece of information comes with a correct answer or label. It's like a quiz with the questions and answers already provided.

Gradient Descent: This method helps find the best settings for a neural network by slowly tweaking them to reduce errors, similar to finding the lowest point in a valley.

Cost Function: Imagine it as a score that tells you how wrong your network's predictions are. The goal is to make this score as low as possible.

Learning Rate: This hyperparameter specifies how big the steps are when adjusting the neural network's settings during training. Too big, and you might skip over the best setting; too small, and it'll take a very long time to get there.

Backpropagation: Short for backward propagation of errors. This is like a feedback system that tells each part of the neural network how much it contributed to any mistakes, so it can learn and do better next time.

PyTorch: it is a dynamic and powerful tool for building and training machine learning models. It simplifies the process with its fundamental building blocks like tensors and neural networks and offers effective ways to define objectives and improve models using loss functions and optimizers. By leveraging PyTorch, anyone can gain the skills to work with large amounts of data and develop cutting-edge AI applications.

Tensors: Generalized versions of vectors and matrices that can have any number of dimensions (i.e. multidimensional arrays). They hold data for processing with operations like addition or multiplication.

Matrix operations: Calculations involving matrices, which are two-dimensional arrays, like adding two matrices together or multiplying them.

Scalar values: Single numbers or quantities that only have magnitude, not direction (for example, the number 7 or 3.14).

Linear algebra: An area of mathematics focusing on vector spaces and operations that can be performed on vectors and matrices.

Loss functions: They measure how well a model is performing by calculating the difference between the model's predictions and the actual results.

Cross entropy loss: This is a measure used when a model needs to choose between categories (like whether an image shows a cat or a dog), and it shows how well the model's predictions align with the actual categories.

Mean squared error: This shows the average of the squares of the differences between predicted numbers (like a predicted price) and the actual numbers. It's often used for predicting continuous values rather than categories.

PyTorch optimizers are important tools that help improve how a neural network learns from data by adjusting the model's parameters. By using these optimizers, like stochastic gradient descent (SGD) with momentum or Adam,

Gradients: Directions and amounts by which a function increases most. The parameters can be changed in a direction opposite to the gradient of the loss function in order to reduce the loss.

Learning Rate: This hyperparameter specifies how big the steps are when adjusting the neural network's settings during training. Too big, and you might skip over the best setting; too small, and it'll take a very long time to get there.

Momentum: A technique that helps accelerate the optimizer in the right direction and dampens oscillations.

PyTorch Dataset class: This is like a recipe that tells your computer how to get the data it needs to learn from, including where to find it and how to parse it, if necessary.

PyTorch Data Loader: Think of this as a delivery truck that brings the data to your AI in small, manageable loads called batches; this makes it easier for the AI to process and learn from the data.

Batches: Batches are small, evenly divided parts of data that the AI looks at and learns from each step of the way.

Generative Adverserial Netwoks(GANS)

Generative Adversarial Networks (GANs) are a type of machine learning framework that use two neural networks to create new data that resembles training data:
Generator: Learns to produce plausible data
Discriminator: Learns to distinguish the generator's fake data from real data
GANs are based on game theory

Recurrent Neural Networks:

Recurrent Neural Networks introduce a mechanism where the output from one step is fed back as input to the next, allowing them to retain information from previous inputs. This design makes RNNs well-suited for tasks where context from earlier steps is essential, such as predicting the next word in a sentence.

The defining feature of RNNs is their hidden state—also called the memory state—which preserves essential information from previous inputs in the sequence. By using the same parameters across all steps, RNNs perform consistently across inputs, reducing parameter complexity compared to traditional neural networks. This capability makes RNNs highly effective for sequential tasks.

Transformer Models:

Transformer models are a type of deep learning model that use self-attention mechanisms to process sequential data. They are commonly used for natural language processing (NLP) tasks, such as: machine translation, text summarization, and question answering.
Eg: text generation, translations,learn long range dependencies, generate new sequential data.

The benefit with transformer model is they read the entire input sequence parallally, rather than one word ata a time like rnn

Perceptron:

A perceptron is an essential component in the world of AI, acting as a binary classifier capable of deciding whether data, like an image, belongs to one class or another. It works by adjusting its weighted inputs—think of these like dials fine-tuning a radio signal—until it becomes better at predicting the right class for the data. This process is known as learning, and it shows us that even complex tasks start with small, simple steps.

A Multilayer Perceptron (MLP) makes accurate predictions by iteratively adjusting its internal weights and biases through a process called backpropagation, allowing it to learn complex non-linear relationships within data, effectively mapping input features to the desired output, even on unseen data, by progressively refining its understanding of the patterns in the training data; essentially, the more layers it has, the more intricate patterns it can capture, leading to better prediction accuracy.

Hugging Face is a company making waves in the technology world with its amazing tools for understanding and using human language in computers. Hugging Face offers everything from tokenizers, which help computers make sense of text, to a huge variety of ready-to-go language models, and even a treasure trove of data suited for language tasks.

A foundation model is a powerful AI tool that can do many different things after being trained on lots of diverse data. These models are incredibly versatile and provide a solid base for creating various AI applications, like a strong foundation holds up different kind of buildings. By using a foundation model, we have a strong starting point for building specialized AI tasks.

Foundation Models and Traditional Models are two distinct approaches in the field of artificial intelligence with different strengths. Foundation Models, which are built on large, diverse datasets, have the incredible ability to adapt and perform well on many different tasks. In contrast, Traditional Models specialize in specific tasks by learning from smaller, focused datasets, making them more straightforward and efficient for targeted applications.

Benchmarks matter because they are the standards that help us measure and accelerate progress in AI. They offer a common ground for comparing different AI models and encouraging innovation, providing important stepping stones on the path to more advanced AI technologies.

The GLUE benchmarks serve as an essential tool to assess an AI's grasp of human language, covering diverse tasks, from grammar checking to complex sentence relationship analysis. By putting AI models through these varied linguistic challenges, we can gauge their readiness for real-world tasks and uncover any potential weaknesses.

SuperGlue is designed as a successor to the original GLUE benchmark. It's a more advanced benchmark aimed at presenting even more challenging language understanding tasks for AI models. Created to push the boundaries of what AI can understand and process in natural language, SuperGlue emerged as models began to achieve human parity on the GLUE benchmark. It also features a public leaderboard, facilitating the direct comparison of models and enabling the tracking of progress over time.

Generative AI, specifically Large Language Models (LLMs), rely on a rich mosaic of data sources to fine-tune their linguistic skills. These sources include web content, academic writings, literary works, and multilingual texts, among others. By engaging with a variety of data types, such as scientific papers, social media posts, legal documents, and even conversational dialogues, LLMs become adept at comprehending and generating language across many contexts, enhancing their ability to provide relevant and accurate information.

The scale of data for Large Language Models (LLMs) is tremendously vast, involving datasets that could equate to millions of books. The sheer size is pivotal for the model's understanding and mastery of language through exposure to diverse words and structures.

Biases in training data deeply influence the outcomes of AI models, reflecting societal issues that require attention. Ways to approach this challenge include promoting diversity in development teams, seeking diverse data sources, and ensuring continued vigilance through bias detection and model monitoring.

In today's digital landscape, disinformation and misinformation pose significant risks, as foundation models like AI language generators have the potential to create and propagate false content. It's crucial to educate people about AI's capabilities and limitations to help them critically assess AI-generated material, fostering a community that is well-informed and resilient against these risks.

Foundation models have both environmental and human impacts that are shaping our world. While the environmental footprint includes high energy use, resource depletion, and electronic waste, we're also facing human challenges in the realms of economic shifts, bias and fairness, privacy concerns, and security risks.

Foundation models leverage advanced architectures and vast computational power to process unprecedented volumes of data, pushing machine learning into an exciting new territory. These models serve as robust platforms for developing a variety of applications, from language processing to image generation.

While foundation models present vast opportunities, they also introduce increased risks of bias and misinformation, making the discussion around mitigating these risks essential.

Adaptation in AI is a crucial step to enhance the capabilities of foundation models, allowing them to cater to specific tasks and domains. This process is about tailoring pre-trained AI systems with new data, ensuring they perform optimally in specialized applications and respect privacy constraints. Reaping the benefits of adaptation leads to AI models that are not only versatile but also more aligned with the unique needs of organizations and industries.

Adapting foundation models is essential due to their limitations in specific areas despite their extensive training on large datasets. Although they excel at many tasks, these models can sometimes misconstrue questions or lack up-to-date information, which highlights the need for fine-tuning. By addressing these weaknesses through additional training or other techniques, the performance of foundation models can be significantly improved.

Retrieval-Augmented Generation (RAG) is a powerful approach for keeping Generative AI models informed with the most recent data, particularly when dealing with domain-specific questions. It cleverly combines the comprehensive understanding capacity of a large language model (LLM) with the most up-to-date information pulled from a database of relevant text snippets. The beauty of this system is in its ability to ensure that responses remain accurate and reflective of the latest developments.

Prompt Design Techniques are innovative strategies for tailoring AI foundation models to specific tasks, fostering better performance in

Code Examples
Images as PyTorch Tensors
```
import torch

# Create a 3-dimensional tensor
images = torch.rand((4, 28, 28))

# Get the second image
second_image = images[1]
```
Displaying Images
```
import matplotlib.pyplot as plt

plt.imshow(second_image, cmap='gray')
plt.axis('off') # disable axes
plt.show()
```
Matrix Multiplication
```
a = torch.tensor([[1, 1], [1, 0]])

print(a)
# tensor([[1, 1],
#        [1, 0]])

print(torch.matrix_power(a, 2))
# tensor([[2, 1],
#        [1, 1]])

print(torch.matrix_power(a, 3))
# tensor([[3, 2],
#        [2, 1]])

print(torch.matrix_power(a, 4))
# tensor([[5, 3],
#        [3, 2]])
```

Code Example
```
import torch.nn as nn

class MLP(nn.Module):
   def __init__(self, input_size):
      super(MLP, self).__init__()
      self.hidden_layer = nn.Linear(input_size, 64)
      self.output_layer = nn.Linear(64, 2)
      self.activation = nn.ReLU()

   def forward(self, x):
      x = self.activation(self.hidden_layer(x))
      return self.output_layer(x)

model = MLP(input_size=10)
print(model)
# MLP(
#   (hidden_layer): Linear(in_features=10, out_features=64, bias=True)
#   (output_layer): Linear(in_features=64, out_features=2, bias=True)
#   (activation): ReLU()
# )

model.forward(torch.rand(10))
# tensor([0.2294, 0.2650], grad_fn=<AddBackward0>)
```

Code Examples
Cross-Entropy Loss
```
import torch
import torch.nn as nn

loss_function = nn.CrossEntropyLoss()

# Our dataset contains a single image of a dog, where
# cat = 0 and dog = 1 (corresponding to index 0 and 1)
target_tensor = torch.tensor([1])
target_tensor
# tensor([1])
Prediction: Most likely a dog (index 1 is higher)

# Note that the values do not need to sum to 1
predicted_tensor = torch.tensor([[2.0, 5.0]])
loss_value = loss_function(predicted_tensor, target_tensor)
loss_value
# tensor(0.0181)
Prediction: Slightly more likely a cat (index 0 is higher)

predicted_tensor = torch.tensor([[1.5, 1.1]])
loss_value = loss_function(predicted_tensor, target_tensor)
loss_value
# tensor(0.9130)
```
Mean Squared Error Loss
```
# Define the loss function
loss_function = nn.MSELoss()

# Define the predicted and actual values as tensors
predicted_tensor = torch.tensor([320000.0])
actual_tensor = torch.tensor([300000.0])

# Compute the MSE loss
loss_value = loss_function(predicted_tensor, actual_tensor)
print(loss_value.item()) # Loss value: 20000 * 20000 / 1 = ...
# 400000000.0
```

Code Examples
Assuming model is your defined neural network.

lr=0.01 sets the learning rate to 0.01 for either optimizer.

```
import torch.optim as optim
```
Stochastic Gradient Descent
```
# momentum=0.9 smoothes out updates and can help training
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```
Adam
```
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

Datasets
```
from torch.utils.data import Dataset

# Create a toy dataset
class NumberProductDataset(Dataset):
   def __init__(self, data_range=(1, 10)):
      self.numbers = list(range(data_range[0], data_range[1]))

   def __getitem__(self, index):
      number1 = self.numbers[index]
      number2 = self.numbers[index] + 1
      return (number1, number2), number1 * number2

   def __len__(self):
      return len(self.numbers)

# Instantiate the dataset
dataset = NumberProductDataset(
   data_range=(0, 11)
)

# Access a data sample
data_sample = dataset[3]
print(data_sample)
# ((3, 4), 12)
```
Data Loaders
```
from torch.utils.data import DataLoader

# Instantiate the dataset
dataset = NumberProductDataset(data_range=(0, 5))

# Create a DataLoader instance
dataloader = DataLoader(dataset, batch_size=3, shuffle=True)

# Iterating over batches
for (num_pairs, products) in dataloader:
   print(num_pairs, products)
# [tensor([4, 3, 1]), tensor([5, 4, 2])] tensor([20, 12, 2])
# [tensor([2, 0]), tensor([3, 1])] tensor([6, 0])
```

Code Examples
Create a Number Sum Dataset
This dataset has two features—a pair of numbers—and a target value—the sum of those two numbers.

Note that this is not actually a good use of deep learning. At the end of our training loop, the model still doesn't know how to add 3 + 7! The idea here is to use a simple example so it's easy to evaluate the model's performance.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader

class NumberSumDataset(Dataset):
   def __init__(self, data_range=(1, 10)):
```

Applications Of Gen Ai

1.Creative Content Generation
- Artwork Synthesis
- Music composition
- Literary creation

2.Product Development
- Design Optimization
- Rapid Prototyping
- Material Exploration

3.Scientific Research
- Experiment Simulation
- Data Analysis and Prediction
- Molecular Discovery

4.Data Augmentation
- Image Enhancement
- Test Augmentation
- Synthetic Data Creation

5.Personalization
- Content Recommendation
- Bespoke Product Creation
- Experience Customization

Challenges of Gen Ai:

Misinformation campaign
Effects on working people liveli hood
Concerns about originality in art and design
Environmental impacts

GLUE Tasks / Benchmarks

Short Name	Full Name	Description
CoLA	Corpus of Linguistic Acceptability	Measures the ability to determine if an English sentence is linguistically acceptable.
SST-2	Stanford Sentiment Treebank	Consists of sentences from movie reviews and human annotations about their sentiment.
MRPC	Microsoft Research Paraphrase Corpus	Focuses on identifying whether two sentences are paraphrases of each other.
STS-B	Semantic Textual Similarity Benchmark	Involves determining how similar two sentences are in terms of semantic content.
QQP	Quora Question Pairs	Aims to identify whether two questions asked on Quora are semantically equivalent.
MNLI	Multi-Genre Natural Language Inference	Consists of sentence pairs labeled for textual entailment across multiple genres of text.
QNLI	Question Natural Language Inference	Involves determining whether the content of a paragraph contains the answer to a question.
RTE	Recognizing Textual Entailment	Requires understanding whether one sentence entails another.
WNLI	Winograd Natural Language Inference	Tests a system's reading comprehension by having it determine the correct referent of a pronoun in a sentence, where understanding depends on contextual information provided by specific words or phrases.

SuperGLUE Tasks / Benchmarks:

Short Name	Full Name	Description
BoolQ	Boolean Questions	Involves answering a yes/no question based on a short passage.
CB	CommitmentBank	Tests understanding of entailment and contradiction in a three-sentence format.
COPA	Choice of Plausible Alternatives	Measures causal reasoning by asking for the cause/effect of a given sentence.
MultiRC	Multi-Sentence Reading Comprehension	Involves answering questions about a paragraph where each question may have multiple correct answers.
ReCoRD	Reading Comprehension with Commonsense Reasoning	Requires selecting the correct named entity from a passage to fill in the blank of a question.
RTE	Recognizing Textual Entailment	Involves identifying whether a sentence entails, contradicts, or is neutral towards another sentence.
WiC	Words in Context	Tests understanding of word sense disambiguation in different contexts.
WSC	Winograd Schema Challenge	Focuses on resolving coreference resolution within a sentence, often requiring commonsense reasoning.
AX-b	Broad Coverage Diagnostic	A diagnostic set to evaluate model performance on a broad range of linguistic phenomena.
AX-g	Winogender Schema Diagnostics	Tests for the presence of gender bias in automated coreference resolution systems.

Prompting Techniques Covered:

Prompt Tuning: Customizing templates or prompts to guide predictions in specific tasks. The placement and choice of words in prompts are essential.
Few-shot Prompting: Providing a few examples in the prompt to help the model understand the desired output.
Zero-shot Prompting: Allowing the model to handle tasks without any specific examples, relying on general knowledge.
Chain of Thought: Including logical reasoning steps in the prompt to aid the model in solving complex problems.
In-context Learning: Allowing the model to learn from the context provided in the prompt, either through instructions or examples.
Combining Techniques: These techniques can be used together to enhance the model's performance, and learners may need to develop their own methods based on their specific needs.

PyTorch Dataset class: This is like a recipe that tells your computer how to get the data it needs to learn from, including where to find it and how to parse it, if necessary.

PyTorch Data Loader: Think of this as a delivery truck that brings the data to your AI in small, manageable loads called batches; this makes it easier for the AI to process and learn from the data.

Batches: Batches are small, evenly divided parts of data that the AI looks at and learns from each step of the way.

Shuffle: It means mixing up the data so that it's not in the same order every time, which helps the AI learn better.

Training Loop: The cycle that a neural network goes through many times to learn from the data by making predictions, checking errors, and improving itself.

Batches: Batches are small, evenly divided parts of data that the AI looks at and learns from each step of the way.

Epochs: A complete pass through the entire training dataset. The more epochs, the more the computer goes over the material to learn.

Loss functions: They measure how well a model is performing by calculating the difference between the model's predictions and the actual results.

Optimizer: Part of the neural network's brain that makes decisions on how to change the network to get better at its job.

Tokenizers: These work like a translator, converting the words we use into smaller parts and creating a secret code that computers can understand and work with.

Models: These are like the brain for computers, allowing them to learn and make decisions based on information they've been fed.

Datasets: Think of datasets as textbooks for computer models. They are collections of information that models study to learn and improve.

Trainers: Trainers are the coaches for computer models. They help these models get better at their tasks by practicing and providing guidance. HuggingFace Trainers implement the PyTorch training loop for you, so you can focus instead on other aspects of working on the model.

Tokenization: It's like cutting a sentence into individual pieces, such as words or characters, to make it easier to analyze.

Tokens: These are the pieces you get after cutting up text during tokenization, kind of like individual Lego blocks that can be words, parts of words, or even single letters. These tokens are converted to numerical values for models to understand.

Pre-trained Model: This is a ready-made model that has been previously taught with a lot of data.

Uncased: This means that the model treats uppercase and lowercase letters as the same.

IMDb dataset: A dataset of movie reviews that can be used to train a machine learning model to understand human sentiments.

Apache Arrow: A software framework that allows for fast data processing

Truncating: This refers to shortening longer pieces of text to fit a certain size limit.

Padding: Adding extra data to shorter texts to reach a uniform length for processing.

Batches: Batches are small, evenly divided parts of data that the AI looks at and learns from each step of the way.

Batch Size: The number of data samples that the machine considers in one go during training.

Epochs: A complete pass through the entire training dataset. The more epochs, the more the computer goes over the material to learn.

Dataset Splits: Dividing the dataset into parts for different uses, such as training the model and testing how well it works.

Transfer learning: The process where knowledge from a pre-trained model is applied to a new, but related task.

Foundation Model: A large AI model trained on a wide variety of data, which can do many tasks without much extra training.

Adapted: Modified or adjusted to suit new conditions or a new purpose, i.e. in the context of foundation models.

Generalize: The ability of a model to apply what it has learned from its training data to new, unseen data.

Sequential data: Information that is arranged in a specific order, such as words in a sentence or events in time.

Self-attention mechanism: The self-attention mechanism in a transformer is a process where each element in a sequence computes its representation by attending to and weighing the importance of all elements in the sequence, allowing the model to capture complex relationships and dependencies.

Transformers are a type of deep learning architecture designed to process sequential data, such as text, by capturing contextual relationships between elements in the sequence. They use a mechanism called selfattention to weigh the importance of each word relative to others in a sentence, allowing them to model longrange dependencies effectively. Transformers are the foundation for many modern NLP models like BERT, GPT, and T5.

Robustness: The strength of an AI model to maintain its performance despite challenges or changes in data.

Open Access: Making data sets freely available to the public, so that anyone can use them for research and develop AI technologies.

Semantic Equivalence: When different phrases or sentences convey the same meaning or idea.

Textual Entailment: The relationship between text fragments where one fragment follows logically from the other.

Coreference Resolution: This is figuring out when different words or phrases in a text, like the pronoun she and the president, refer to the same person or thing.

Preprocessing: This is the process of preparing and cleaning data before it is used to train a machine learning model. It might involve removing errors, irrelevant information, or formatting the data in a way that the model can easily learn from it.

Fine-tuning: After a model has been pre-trained on a large dataset, fine-tuning is an additional training step where the model is further refined with specific data to improve its performance on a particular type of task.

Gigabytes/Terabytes: Units of digital information storage. One gigabyte (GB) is about 1 billion bytes, and one terabyte (TB) is about 1,000 gigabytes. In terms of text, a single gigabyte can hold roughly 1,000 books.

Common Crawl: An open repository of web crawl data. Essentially, it is a large collection of content from the internet that is gathered by automatically scraping the web from 25 billion webpages.

Selection Bias: When the data used to train an AI model does not accurately represent the whole population or situation by virtue of the selection process, e.g. those choosing the data will tend to choose dataset their are aware of

Historical Bias: Prejudices and societal inequalities of the past that are reflected in the data, influencing the AI in a way that perpetuates these outdated beliefs.

Confirmation Bias: The tendency to favor information that confirms pre-existing beliefs, which can affect what data is selected for AI training.

Discriminatory Outcomes: Unfair results produced by AI that disadvantage certain groups, often due to biases in the training data or malicious actors.

Echo Chambers: Situations where biased AI reinforces and amplifies existing biases, leading to a narrow and distorted sphere of information.

Bias Detection and Correction: Processes and algorithms designed to identify and remove biases from data before it's used to train AI models.

Transparency and Accountability: Openness about how AI models are trained and the nature of their data, ensuring that developers are answerable for their AI's performance and impact.

Synthetic Voices: These are computer-generated voices that are often indistinguishable from real human voices. AI models have been trained on samples of speech to produce these realistic voice outputs.

Content Provenance Tools: Tools designed to track the origin and history of digital content. They help verify the authenticity of the content by providing information about its creation, modification, and distribution history.

Fine-tuning: This is a technique in machine learning where an already trained model is further trained (or tuned) on a new, typically smaller, dataset for better performance on a specific task.

Semantic-embedding: A representation of text in a high-dimensional space where distances between points correspond to semantic similarity. Phrases with similar meanings are closer together.

Cosine similarity: A metric used to measure how similar two vectors are, typically used in the context of semantic embeddings to assess similarity of meanings.

Vector databases: Specialized databases designed to store and handle vector data, often employed for facilitating fast and efficient similarity searches.

Domain-Specific Task: A task that is specialized or relevant to a particular area of knowledge or industry, often requiring tailored AI responses.

Prompt: In AI, a prompt is an input given to the model to generate a specific response or output.

Prompt Tuning: This is a method to improve AI models by optimizing prompts so that the model produces better results for specific tasks.

Hard Prompt: A manually created template used to guide an AI model's predictions. It requires human ingenuity to craft effective prompts.

Soft Prompt: A series of tokens or embeddings optimized through deep learning to help guide model predictions, without necessarily making sense to humans.

One-shot prompting: Giving an AI model a single example to learn from before it attempts a similar task.

Few-shot prompting: Providing an AI model with a small set of examples, such as five or fewer, from which it can learn to generalize and perform tasks.

combines the comprehensive understanding capacity of a large language model (LLM) with the most up-to-date information pulled from a database of relevant text snippets. The beauty of this system is in its ability to ensure that responses remain accurate and reflective of the latest developments.

Prompt Design Techniques are innovative strategies for tailoring AI foundation models to specific tasks, fostering better performance in various domains. These methods enable us to guide the AI's output by carefully constructing the prompts we provide, enhancing the model's relevance and efficiency in generating responses.

Prompt tuning is a technique in generative AI which allows models to target specific tasks effectively. By crafting prompts, whether through a hands-on approach with hard prompts or through an automated process with soft prompts, we enhance the model's predictive capabilities.

When performing few-shot, one-shot, or zero-shot learning, we can pass information to the model within the prompt in the form of examples, descriptions, or other data. When we rely on a model using information from within the prompt itself instead of relying on what is stored within its own parameters we are using in-context learning.

As these AI models grow in size, their ability to absorb and use in-context information significantly improves, showcasing their potential to adapt to various tasks effectively. The progress in this field is inspiring, as these advances hint at an exciting future where such models could be even more intuitive and useful.

Using probing technique to train a classifier is a powerful approach to tailor generative AI foundation models, like BERT, for specific applications. By adding a modestly-sized neural network, known as a classification head, to a foundation model, one can specialize in particular tasks such as sentiment analysis. This technique involves freezing the original model's parameters and only adjusting the classification head through training with labeled data. Ultimately, this process simplifies adapting sophisticated AI systems to our needs, providing a practical tool for developing efficient and targeted machine learning solutions.

Parameter-efficient fine-tuning (PEFT) is a technique crucial for adapting large language models more efficiently, with the bonus of not requiring heavy computational power. This approach includes various strategies to update only a small set of parameters, thereby maintaining a balance between model adaptability and resource consumption. The techniques ensure that models can be swiftly deployed in different industrial contexts, considering both time constraints and the necessity for scaling operations efficiently

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader

class NumberSumDataset(Dataset):
   def __init__(self, data_range=(1, 10)):
      self.numbers = list(range(data_range[0], data_range[1]))

   def __getitem__(self, index):
      number1 = float(self.numbers[index // len(self.numbers)])
      number2 = float(self.numbers[index % len(self.numbers)])
      return torch.tensor([number1, number2]), torch.tensor([number1 +
number2])

   def __len__(self):
      return len(self.numbers) ** 2
```

Inspect the Dataset

```
dataset = NumberSumDataset(data_range=(1, 100))

for i in range(5):
   print(dataset[i])
# (tensor([1., 1.]), tensor([2.]))
# (tensor([1., 2.]), tensor([3.]))
# (tensor([1., 3.]), tensor([4.]))
# (tensor([1., 4.]), tensor([5.]))
# (tensor([1., 5.]), tensor([6.]))
```

Define a Simple Model

```
class MLP(nn.Module):
   def __init__(self, input_size):
      super(MLP, self).__init__()
      self.hidden_layer = nn.Linear(input_size, 128)
      self.output_layer = nn.Linear(128, 1)
      self.activation = nn.ReLU()

   def forward(self, x):
      x = self.activation(self.hidden_layer(x))
      return self.output_layer(x)
```

Instantiate Components Needed for Training

```
dataset = NumberSumDataset(data_range=(0, 100))
dataloader = DataLoader(dataset, batch_size=100, shuffle=True)
model = MLP(input_size=2)
loss_function = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Create a Training Loop

```
for epoch in range(10):
   total_loss = 0.0
   for number_pairs, sums in dataloader:  # Iterate over the batches
      predictions = model(number_pairs)  # Compute the model output
      loss = loss_function(predictions, sums)  # Compute the loss
      loss.backward()  # Perform backpropagation
      optimizer.step()  # Update the parameters
      optimizer.zero_grad()  # Zero the gradients

      total_loss += loss.item()  # Add the loss for all batches

   # Print the loss for this epoch
   print("Epoch {}: Sum of Batch Losses = {:.5f}".format(epoch, total_loss))
   # Epoch 0: Sum of Batch Losses = 118.82360
   # Epoch 1: Sum of Batch Losses = 39.75702
   # Epoch 2: Sum of Batch Losses = 2.16352
   # Epoch 3: Sum of Batch Losses = 0.25178
   # Epoch 4: Sum of Batch Losses = 0.22843
   # Epoch 5: Sum of Batch Losses = 0.19182
   # Epoch 6: Sum of Batch Losses = 0.15507
   # Epoch 7: Sum of Batch Losses = 0.07789
   # Epoch 8: Sum of Batch Losses = 0.06329
   # Epoch 9: Sum of Batch Losses = 0.04936
```

Try the Model Out

```
# Test the model on 3 + 7
model(torch.tensor([3.0, 7.0]))
# tensor([10.1067], grad_fn=<AddBackward0>)
```

Code Example

```
from transformers import BertTokenizer

# Initialize the tokenizer
tokenizer = BertTokenizer.from_pretrained('bert -base-uncased')

# See how many tokens are in the vocabulary
tokenizer.vocab_size
# 30522
# Tokenize the sentence
tokens = tokenizer.tokenize("I heart Generative AI")

# Print the tokens
print(tokens)
# ['i', 'heart', 'genera', '##tive', 'ai']

# Show the token ids assigned to each token
print(tokenizer.convert_tokens_to_ids(tokens))
# [1045, 2540, 11416, 6024, 9932]
```

Code Example

```
from transformers import BertForSequenceClassification, BertTokenizer

# Load a pre-trained sentiment analysis model
model_name = "textattack/bert-base-uncased-imdb"
model = BertForSequenceClassification.from_pretrained(model_name,
num_labels=2)

# Tokenize the input sequence
tokenizer = BertTokenizer.from_pretrained(model_name)
inputs = tokenizer("I love Generative AI", return_tensors="pt")

# Make prediction
with torch.no_grad():
   outputs = model(**inputs).logits
   probabilities = torch.nn.functional.softmax(outputs, dim=1)
   predicted_class = torch.argmax(probabilities)

# Display sentiment result
if predicted_class == 1:
   print(f"Sentiment: Positive ({probabilities[0][1] * 100:.2f}%)")
else:
   print(f"Sentiment: Negative ({probabilities[0][0] * 100:.2f}%)")
# Sentiment: Positive (88.68%)
```

Code Example

Note that this code uses IPython functions (display and HTML) so it should be run in an IPython environment (e.g. Jupyter Notebook).

```
from datasets import load_dataset
from IPython.display import HTML, display

# Load the IMDB dataset, which contains movie reviews
# and sentiment labels (positive or negative)
dataset = load_dataset("imdb")

# Fetch a revie from the training set
review_number = 42
sample_review = dataset["train"][review_number]

display(HTML(sample_review["text"][:450] + "..."))
# WARNING: This review contains SPOILERS. Do not read if you don't want some
points revealed to you before you watch the
# film.
#
# With a cast like this, you wonder whether or not the actors and actresses
knew exactly what they were getting into. Did they
# see the script and say, `Hey, Close Encounters of the Third Kind was such a hit
that this one can't fail.' Unfortunately, it does.
# Did they even think to check on the director's credentials...

if sample_review["label"] == 1:
   print("Sentiment: Positive")
else:
   print("Sentiment: Negative")
# Sentiment: Negative

from transformers import (DistilBertForSequenceClassification,
   DistilBertTokenizer,
   TrainingArguments,
   Trainer
)
from datasets import load_dataset

model = DistilBertForSequenceClassification.from_pretrained(
   "distilbert-base-uncased", num_labels=2
)
tokenizer = DistilBertTokenizer.from_pretrained("distilbert -base-uncased")

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


dataset = load_dataset("imdb")
tokenized_datasets = dataset.map(tokenize_function, batched=True)

training_args = TrainingArguments(
```

Soft Prompt: A series of tokens or embeddings optimized through deep learning to help guide model predictions, without necessarily making sense to humans.

One-shot prompting: Giving an AI model a single example to learn from before it attempts a similar task.

Few-shot prompting: Providing an AI model with a small set of examples, such as five or fewer, from which it can learn to generalize and perform tasks.

Zero-shot prompting: This refers to the capability of an AI model to correctly respond to a prompt or question it hasn't explicitly been trained to answer, relying solely on its prior knowledge and training.

Chain-of-Thought Prompting: A method of guiding a language model through a step-by-step reasoning process to help it solve complex tasks by explicitly detailing the logic needed to reach a conclusion.

Probing: This is a method of examining what information is contained in different parts of a machine learning model.

Linear Probing: A simple form of probing that involves attaching a linear classifier to a pretrained model to adapt it to a new task without modifying the original model.

Classification Head: It is the part of a neural network that is tailored to classify input data into defined categories.

Fine-tuning: This is the process of adjusting a pre-trained model so it performs better on a new, similar task. It's like teaching an experienced doctor a new medical procedure; they're already a doctor, but they're improving their skills in a particular area.

Catastrophic Forgetting: This happens when a model learns something new but forgets what it learned before. Imagine if you crammed for a history test and did great, but then forgot most of what you learned when you started studying for a math test.

Parameter-efficient fine-tuning: A method of updating a predefined subset of a model's parameters to tailor it to specific tasks, without the need to modify the entire model, thus saving computational resources.

Frozen Parameters: In the context of machine learning, this refers to model parameters that are not changed or updated during the process of training or fine-tuning.

Low-Rank Adaptation (LoRA): A technique where a large matrix is approximated using two smaller matrices, greatly reducing the number of parameters that need to be trained during fine-tuning.

Adapters: Additional model components inserted at various layers; only the parameters of these adapters are trained, not of the entire model.

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


dataset = load_dataset("imdb")
tokenized_datasets = dataset.map(tokenize_function, batched=True)

training_args = TrainingArguments(
    per_device_train_batch_size=64,
    output_dir="./results",
    learning_rate=2e-5,
    num_train_epochs=3,
)
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["test"],
)
trainer.train()
```

Thank you

THARUN NAIK RAMAVATH