Introduction to Large Language Models

Dr Nikos Bakas nibas@grnet.gr

Co-funded by the European Union

EuroHPC Joint Undertaking

EURO Greece EuroCC@Greece

https://eurocc-greece.gr/

1

Generative AI

1.1 Turing's Test

The development of digital computers marked the beginning of artificial intelligence being recognized as a viable field.

The Turing Test evaluates a machine's ability to exhibit human-like intelligence.
It is passed if the machine's responses are indistinguishable from a human's.

"The Mathematical Objection"[1]

- based on Gödel's incompleteness theorem[2] about logical systems,
- "describing logical systems in terms of machines"
- assuming a machine with "infinite capacity"
- "there are certain things that such a machine cannot do"

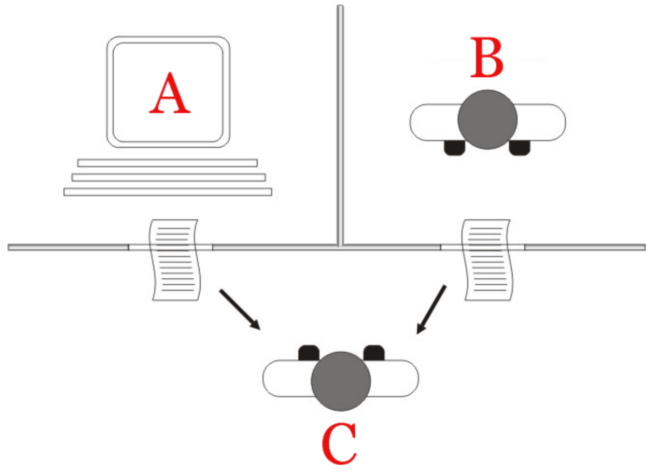


Figure 1.1: Turing's Test[3]

This sounds disappointing (along with the Incompleteness Theorem). However, does it matter?

[1] A. M. TURING (Oct. 1950). "I.—COMPUTING MACHINERY AND INTELLIGENCE". in: *Mind* LIX.236, pp. 433–460. ISSN: 0026-4423. DOI: 10.1093/mind/LIX.236.433
[2] Kurt Gödel (1931). "On Formally Undecidable Propositions of Principia Mathematica and Related Systems I". in: *Monatshefte für Mathematik und Physik* 38.1, pp. 173–198
[3] Juan Alberto Sánchez Margallo https://creativecommons.org/licenses/by/2.5/ https://en.wikipedia.org/wiki/Turing_test#/media/File:Turing_test_diagram.png

1.2 "Shakespeare's" poem on AI and HPC

Prompt: Write a poem for AI and HPC in the style of William Shakespeare

. . .

High Performance Computing, noble steed,

On which we ride to meet our greatest need,

With speed unbridled, across data plains,

To solve the puzzles locked within our brains.

And AI, fair muse, who with gentle hand,

Guides us through the unknowns of digital land,

With algorithms fine and patterns clear,

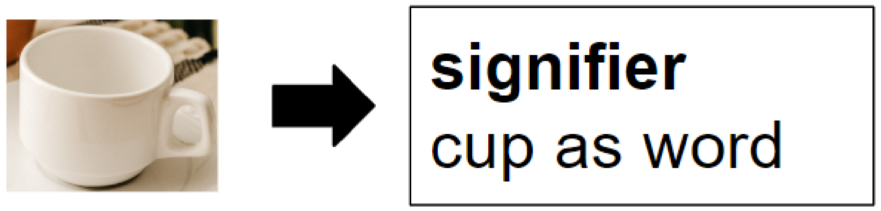
Brings to light the answers we hold dear.

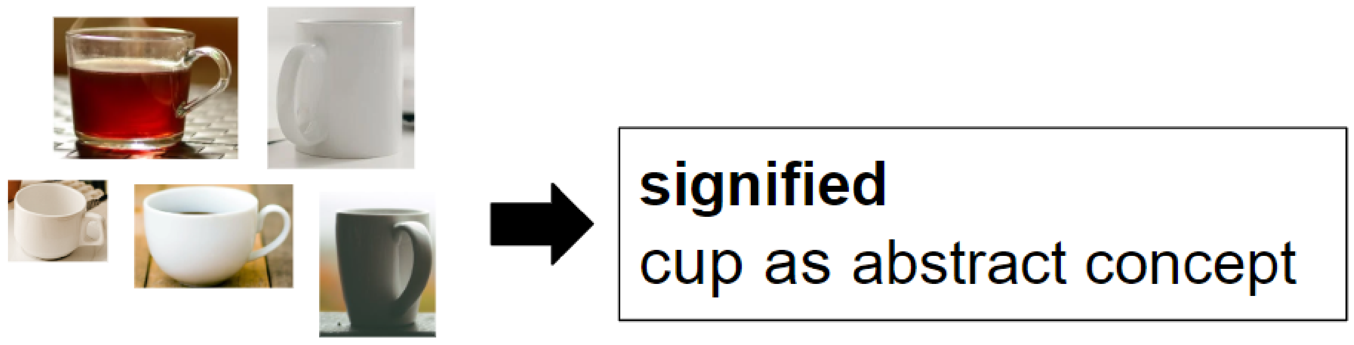
. . .

1.3 Language issues

"this is not a supercomputer"

1.4 Semiotics





We need to represent words in a manner that captures both:

- specific things like the physical form of a cup
- abstract ideas and concepts

1.5 Word representation

In language, words function as signs (signifiers) with specific meanings.

Vocabulary with one word

- If exists in the text:

$$word_1 = [1]$$

- If does not exist in the text:

$$word_1 = [0]$$

- One number (0-1) is enough to represent the existence of a single word.
- This is a common concept of representing categorical variables in machine learning.

To represent 2 words with numbers, we cannot use a single number because if e.g., $word_1 = 1$ and $word_2 = 2$ then we would imply that $word_2$ is somehow twice $word_1$. We need vectors with 2 dimensions.

1.6 One-hot encoding

Vocabulary with two words

$$word_1 = [1, 0]$$

$$word_2 = [0, 1]$$

1.7 Representation of sentences

We need to preserve word sequencing.

For example if

$$\text{``this''} = [1, 0]$$

and

$$\text{``is''} = [0, 1]$$

then

$$\text{``this is''} = [1, 0, 0, 1]$$

and

$$\text{``is this''} = [0, 1, 1, 0]$$

which provide different vectors.

1.8 Vocabulary with N words

One-hot Encoding for N words

In a vocabulary with N words, we use a vector of length N to represent each word. This is called one-hot encoding, where each word is represented by a vector that has a 1 in its corresponding position and 0s elsewhere.

- For a vocabulary of N words: $word_1, word_2, ..., word_N$
- The vector representation for $word_1$ would be:

$$word_1 = [1, 0, 0, ..., 0]$$

- The vector representation for $word_2$ would be:

$$word_2 = [0, 1, 0, ..., 0]$$

- And so on, until:
- The vector representation for $word_N$ would be:

$$word_N = [0, 0, 0, ..., 1]$$

This approach ensures that each word is uniquely represented, avoiding any implication of numerical relationships between different words.

The vector length increases with the number of words, making this method efficient for smaller vocabularies but highly demanding for very large datasets.

1.9 Semantic Limitations of One-hot Encoding

Dot Product of Related Words

In one-hot encoding, the representation of each word is **orthogonal** to every other word in the same vocabulary. This means that the **dot product** between the vectors of any two different words is always **zero, regardless of their semantic relationship**.

- Consider two semantically related words, say *car* and *automobile*. In a vocabulary where each word is represented as a unique one-hot encoded vector:

$$car = [1, 0, 0, ..., 0]$$

$$automobile = [0, 1, 0, ..., 0]$$

- The dot product between *car* and *automobile* would be:

$$car \cdot automobile^T = 1 \times 0 + 0 \times 1 + 0 \times 0 + ... + 0 \times 0 = 0$$

This calculation illustrates a fundamental limitation:

Even though *car* and *automobile* are synonyms, their one-hot vectors are orthogonal, and their dot product does not reflect any semantic closeness.

This outcome is a clear indicator that one-hot encoding is purely syntactic and lacks the capability to capture and represent the semantic relationships between words.

1.10 Word Embeddings and Distributional Semantics

Word embeddings are a type of word representation that allows words with similar meaning to have a similar representation. Based on the principle articulated by J.R. Firth in 1957,

"You shall know a word by the company it keeps"

embeddings encode words into high-dimensional space based on their contexts.

- Similar words cluster together in the vector space, making it possible to perform arithmetic operations like **"king" - "man"** + **"woman"** = **"queen"**.
- Word embeddings **significantly reduce the dimensionality** compared to one-hot encoding while providing a much richer semantic representation.

Examples of Embedding Models:

- **Word2Vec** and **GloVe** utilize the context of words to learn dense embeddings, where the dot product reflects semantic similarity to some extent.
- **BERT** and other transformer-based models further enhance this by considering the entire context of a word's usage in language, leading to even richer semantic representations.

1.11 Optimizing Word Embeddings

Optimizing word embeddings involves identifying the best vector representations that accurately capture semantic relationships in a dataset.

1.11.1 Objective Function

The objective function, denoted by $\mathcal{L}$, evaluates the performance of word vectors in their ability to predict context words. In the context of the Word2Vec model, the function $\mathcal{L}$ to be maximized is:

$$\mathcal{L} = \sum_{(w,c)\in D} \log P(c|w)$$

Here, w is a target word, c is a context word, and D consists of all pairs of target and context words in the dataset. The probability $P(c|w)$ is computed as follows:

$$P(c|w) = \frac{\exp(\mathbf{v}_w \cdot \mathbf{u}_c)}{\sum_{c'\in\text{Vocabulary}} \exp(\mathbf{v}_w \cdot \mathbf{u}_{c'})}$$

where $\mathbf{v}_w$ and $\mathbf{u}_c$ are the vector representations of the target word w and the context word c respectively.

- **Numerator** $(\exp(\mathbf{v}_w \cdot \mathbf{u}_c))$: This term is the exponential of the dot product between the vector of the target word w and the vector of the context word c. It quantifies the compatibility or similarity between w and c, with a **higher dot product** resulting in a larger value, indicating a **stronger relationship**.
- **Denominator** $(\sum_{c'\in\text{Vocabulary}} \exp(\mathbf{v}_w \cdot \mathbf{u}_{c'}))$: This sum involves the exponential of the dot products between the vector of the target word w and every possible context word vector in the vocabulary. It serves as a normalization factor that ensures **the probabilities across all possible context words sum to one**, thus forming a valid probability distribution.

This formulation uses the softmax function to convert the dot products into probabilities that sum to one over all possible context words. The optimization goal is to maximize $\mathcal{L}$, which aggregates the log probabilities of predicting the correct context words given target words, effectively enhancing the semantic accuracy of the embeddings.

1.11.2 Optimization Algorithm

Gradient descent is the algorithm of choice for maximizing $\mathcal{L}$. This iterative method adjusts word vectors to improve the value of $\mathcal{L}$. The update formula is:

$$\mathbf{v} = \mathbf{v} - \eta \nabla \mathcal{L}(\mathbf{v})$$

In this formula, $\mathbf{v}$ represents a word vector, η is the learning rate, and $\nabla \mathcal{L}(\mathbf{v})$ is the gradient of $\mathcal{L}$ at $\mathbf{v}$, directing the update towards the highest increase of the likelihood function.

By effectively maximizing $\mathcal{L}$, this optimization process refines word vectors, enhancing their ability to predict context and thereby improving both their syntactic and semantic accuracies.

1.12 Code implementation

```python
import numpy as np

# Define the corpus
corpus = [
    "The apple is a sweet fruit that many enjoy fresh",
    "The vast ocean is far from the quiet orchard"
]

# Tokenize and create a vocabulary index
words = set(word for sentence in corpus for word in sentence.lower().split())
word_to_index = {word: i for i, word in enumerate(words)}
vocab_size = len(word_to_index)

# Initialize word vectors with small random values for simplicity
np.random.seed(42)
embed_size = 50  # Size of each word vector
word_vectors = np.random.rand(vocab_size, embed_size) * 0.1

# Define a simple training routine to simulate relatedness
def train_adjusted(word_vectors, word_to_index, learning_rate=0.01, iterations=100):
    for _ in range(iterations):
        apple_idx = word_to_index['apple']
        fruit_idx = word_to_index['fruit']
        ocean_idx = word_to_index['ocean']

        # Normalize vectors initially
        word_vectors[apple_idx] /= np.linalg.norm(word_vectors[apple_idx])
        word_vectors[fruit_idx] /= np.linalg.norm(word_vectors[fruit_idx])
        word_vectors[ocean_idx] /= np.linalg.norm(word_vectors[ocean_idx])

        # Adjust vectors
        word_vectors[apple_idx] += learning_rate * (word_vectors[fruit_idx] - word_vectors[apple_idx
         ↪ ])
        word_vectors[fruit_idx] += learning_rate * (word_vectors[apple_idx] - word_vectors[fruit_idx
        ↪ ])
        word_vectors[ocean_idx] -= learning_rate * (word_vectors[apple_idx] - word_vectors[ocean_idx
        ↪ ])

        # Re-normalize vectors after adjustment
        word_vectors[apple_idx] /= np.linalg.norm(word_vectors[apple_idx])
        word_vectors[fruit_idx] /= np.linalg.norm(word_vectors[fruit_idx])
        word_vectors[ocean_idx] /= np.linalg.norm(word_vectors[ocean_idx])

# Train with adjusted function
train_adjusted(word_vectors, word_to_index)

# Get the vectors
apple_vec = word_vectors[word_to_index['apple']]
fruit_vec = word_vectors[word_to_index['fruit']]
ocean_vec = word_vectors[word_to_index['ocean']]

# Compute and print Dot products
print("Dot product between 'apple' and 'fruit':", np.dot(apple_vec, fruit_vec))
print("Dot product between 'apple' and 'ocean':", np.dot(apple_vec, ocean_vec))
```

Output:

```
>>> Dot product between 'apple' and 'fruit': 0.995439598890918
>>> Dot product between 'apple' and 'ocean': -0.002559936403922
```

1.13 Training Transformers with Embedded Word Representations

Transformers, a class of models introduced by Vaswani et al. in 2017, utilize word embeddings to understand and generate human language. These models leverage the power of self-attention mechanisms to consider the context of each word in a sentence, regardless of their position.

Transformers do not process words sequentially like older RNN-based models. Instead, they handle all words in parallel, significantly improving speed and efficiency.

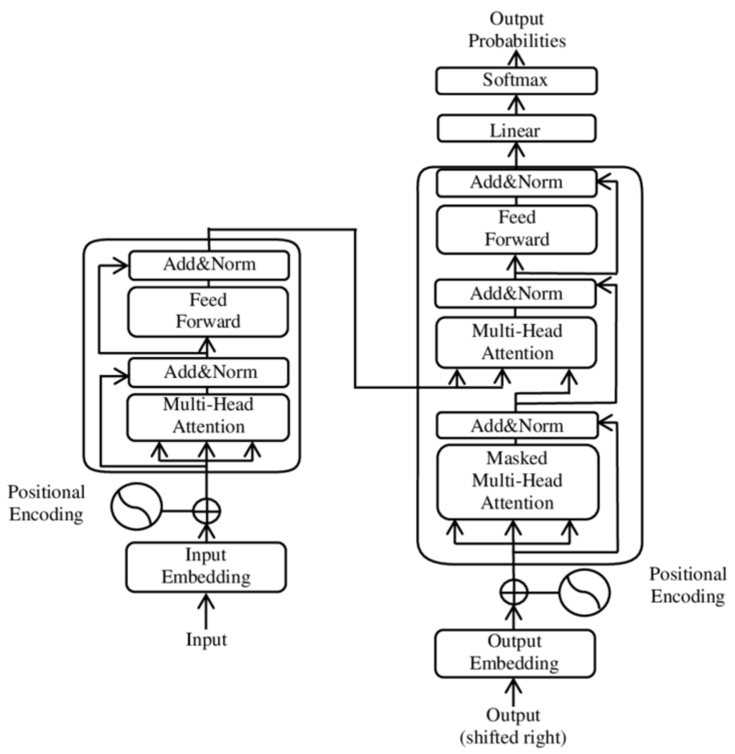


Figure 1.2: The Transformer model architecture[4]

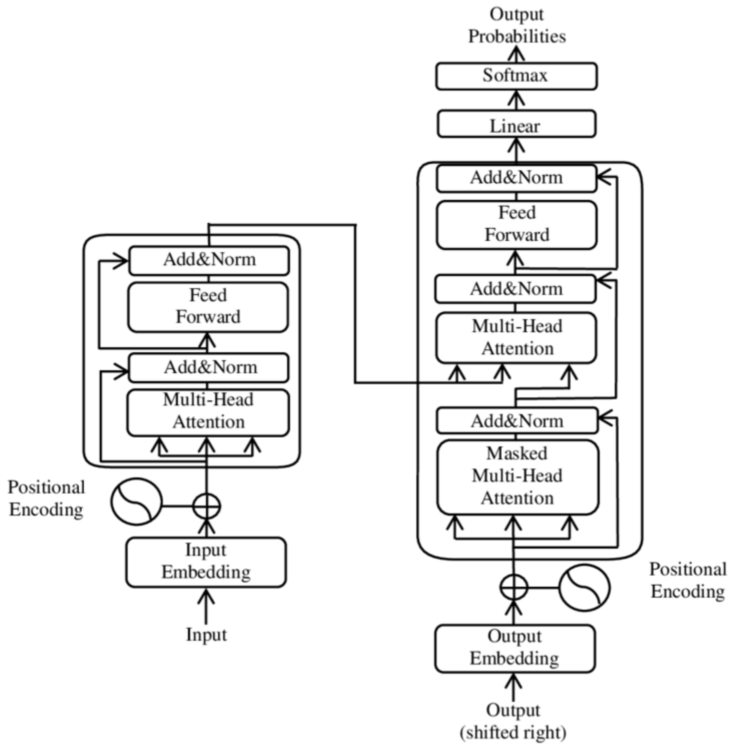
- **Input Embedding**: The input words are converted into vectors.
- **Positional Encoding**: This step adds information about the position of each word in the sequence to the embeddings.

[4]Yuening Jia - DOI:10.1088/1742-6596/1314/1/012186
https://en.wikipedia.org/wiki/Transformer_(deep_learning_architecture)#/media/File:The-Transformer-model-architecture.png
https://creativecommons.org/licenses/by-sa/3.0

- **Multi-Head Attention:** This layer allows the model to focus on different positions of the input sequence, essentially enabling it to **understand context and relationships** between words.

- **Add & Norm:** Also known as residual connections followed by layer normalization. These steps help in stabilizing the learning process and **allow gradients to flow** without diminishing or excessively amplifying through the network during backpropagation.

- **Feed Forward:** This is a simple neural network applied to each position separately and identically. It **transforms the output of the attention layer to help in predicting the next word** in the sequence.



- **Output Embedding** (shifted right): This is often used in the context of decoder parts of the transformer in sequence-to-sequence models, where **the output at each time step becomes the input to the next step** after being shifted right.

- **Softmax:** The final layer in the transformer output that **converts the logits** from the last linear layer **into probabilities**, which are used to predict the next word in the sequence.

1.14 Big Data

Large language models can be trained on datasets containing trillions of words, which translates to roughly **tens of trillions of tokens**.

- **100 tokens are approximately equal to 75 words**. Tokens can be a whole word, but they can also be smaller parts of words or even punctuation marks.
- **Wikipedia** comprises approximately **3 billion** tokens.
- **The British Library** has around 14 million books. Assuming an average book has 50,000 tokens (words x 1.3 tokens/word), we get a very rough estimate of **700 billion tokens** (14 million books * 50,000 tokens/book). **It spans 9.6 kilometres** (6mi) of shelf space. https://en.wikipedia.org/wiki/British_Library

1.15 This is a small amount of books!

1.16 Model and Token Size

Assuming **a model with 1.3 billion parameters** and calculations done in **mixed precision (float16)**, we obtain:

Model Size:

$$\text{Number of parameters} = 1.3 \times 10^9 \tag{1.1}$$

$$\text{Memory per parameter (float16)} = 2\,\text{bytes} \tag{1.2}$$

$$\text{Total model memory} \approx 2.42\,\text{GB} \tag{1.3}$$

Token Capacity

$$\text{Memory per token } \approx 95.37\,\text{KB} \tag{1.4}$$

$$\text{(including embeddings and activations' calculations, back-propagation, etc.)}$$

$$\text{Total available memory for tokens} \approx 29.58\,\text{GB} \tag{1.5}$$

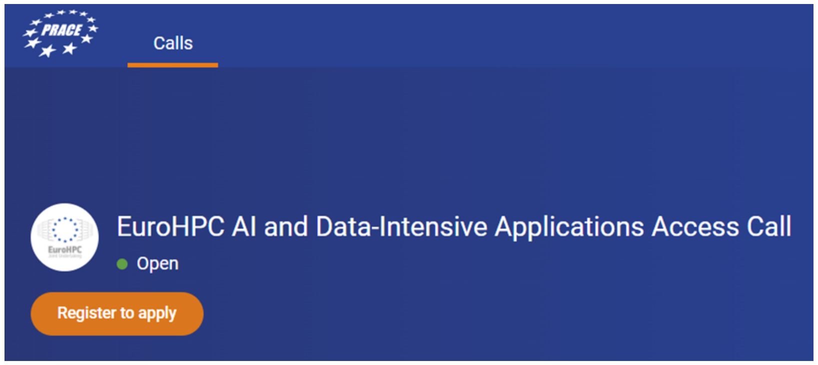
$$\text{Estimated maximum number of tokens} \approx 310{,}154 \tag{1.6}$$

Hence, a 32 GB GPU can roughly process the 1/10,000,000 of an LLM's dataset for a model with 1,3 billion parameters (only).

The previous calculations regard the processing of a single batch, not the entire dataset. The estimation for the number of tokens (approximately 310,154 tokens) that can fit into the GPU memory is based on processing these tokens in a single batch on a 32 GB GPU. Advanced subword tokenization (like BPE or SentencePiece), help handle the vocabulary size more efficiently than traditional tokenization methods.

.

1.17 EuroHPC JU Access Call for AI and Data-Intensive Applications



EuroHPC AI and Data-Intensive Applications Access Call

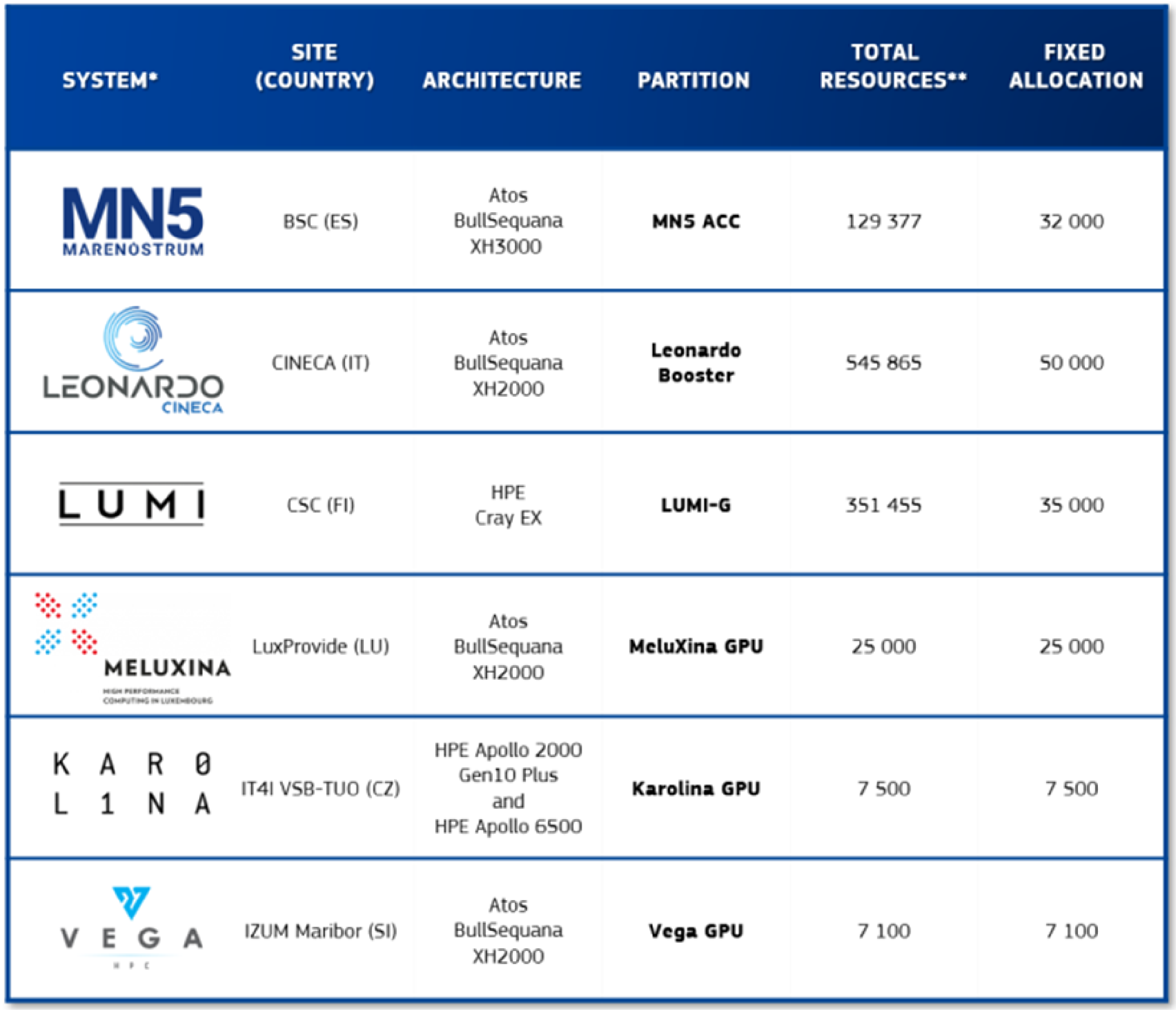
Open

Call Details:

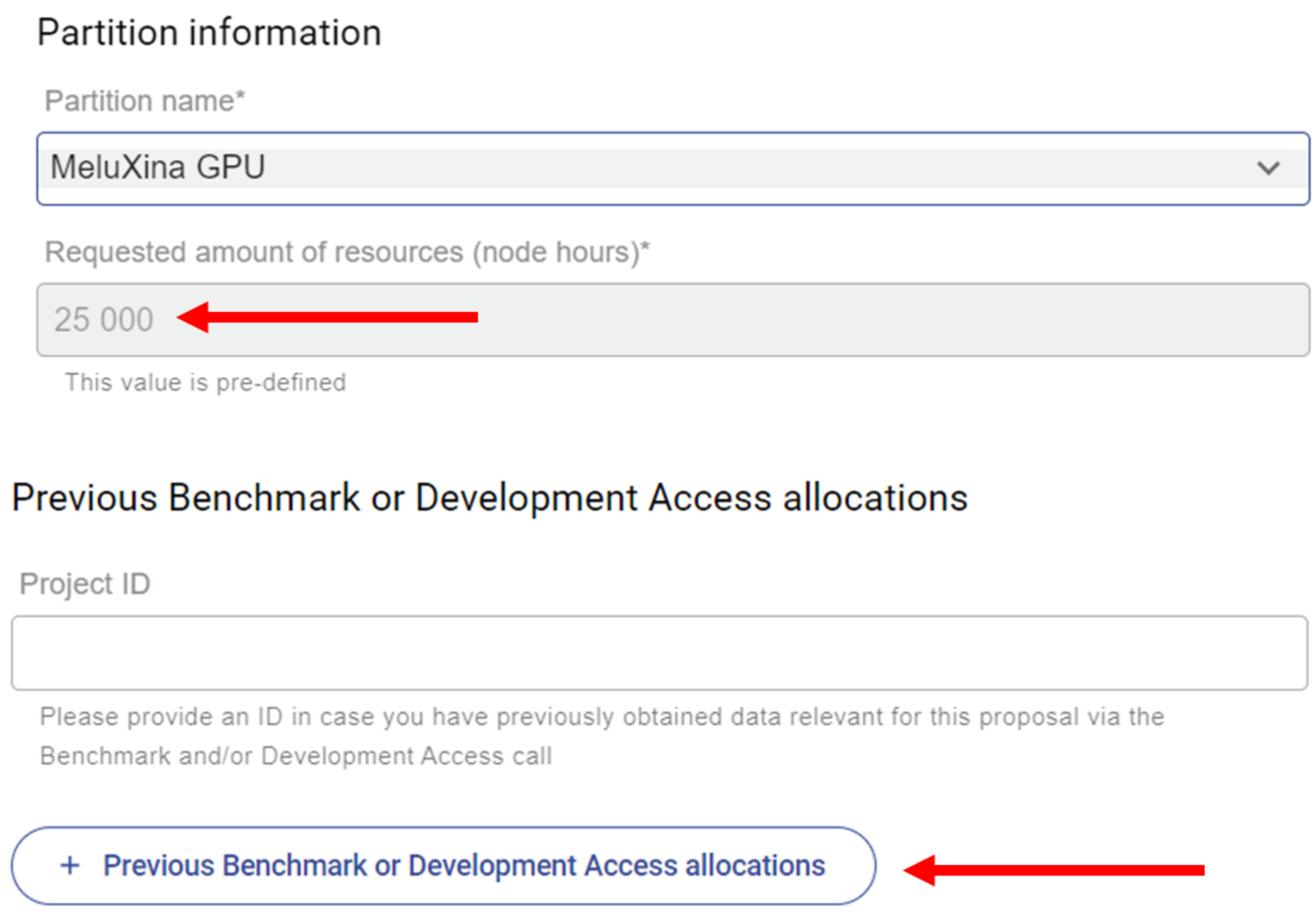
The EuroHPC JU AI and Data-Intensive Applications Access call aims to support **ethical artificial intelligence, machine learning, and in general, data-intensive applications**, with a particular focus on **foundation models and generative AI** (e.g. large language models).

The call is intended to serve **industry organizations, small to medium enterprises (SMEs), startups, as well as public sector entities**, requiring access to supercomputing resources to perform artificial intelligence and data-intensive activities.

https://eurohpc-ju.europa.eu/eurohpc-ju-access-call-ai-and-data-intensive-applications_en



SYSTEM*	SITE (COUNTRY)	ARCHITECTURE	PARTITION	TOTAL RESOURCES**	FIXED ALLOCATION
MN5 MARENOSTRUM	BSC (ES)	Atos BullSequana XH3000	**MN5 ACC**	129 377	32 000
LEONARDO CINECA	CINECA (IT)	Atos BullSequana XH2000	**Leonardo Booster**	545 865	50 000
LUMI	CSC (FI)	HPE Cray EX	**LUMI-G**	351 455	35 000
MELUXINA HIGH PERFORMANCE COMPUTING IN LUXEMBOURG	LuxProvide (LU)	Atos BullSequana XH2000	**MeluXina GPU**	25 000	25 000
KAR0L1NA	IT4I VSB-TUO (CZ)	HPE Apollo 2000 Gen10 Plus and HPE Apollo 6500	**Karolina GPU**	7 500	7 500
VEGA HPC	IZUM Maribor (SI)	Atos BullSequana XH2000	**Vega GPU**	7 100	7 100



Partition information

Partition name*

MeluXina GPU

Requested amount of resources (node hours)*

25 000

This value is pre-defined

Previous Benchmark or Development Access allocations

Project ID

Please provide an ID in case you have previously obtained data relevant for this proposal via the Benchmark and/or Development Access call

Thank you!

Introduction to Large Language Models
Dr Nikos Bakas

Co-funded by the European Union

EuroHPC Joint Undertaking

EURO Greece

EuroCC@Greece

https://eurocc-greece.gr/