GenAI for Software Engineering 2025

Transformers and fine-tuning

Jorge Melegati

Let’s recall RNNs’ limitations

- Limited memory
  - Problems with long sentences
- Bias to recent data
- Sequential nature
  - Each element of the input is provided at a time
  - Precludes parallelization

Transformers

- Proposed by Vaswani et al. (2017) in the paper “Attention is all you need”

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

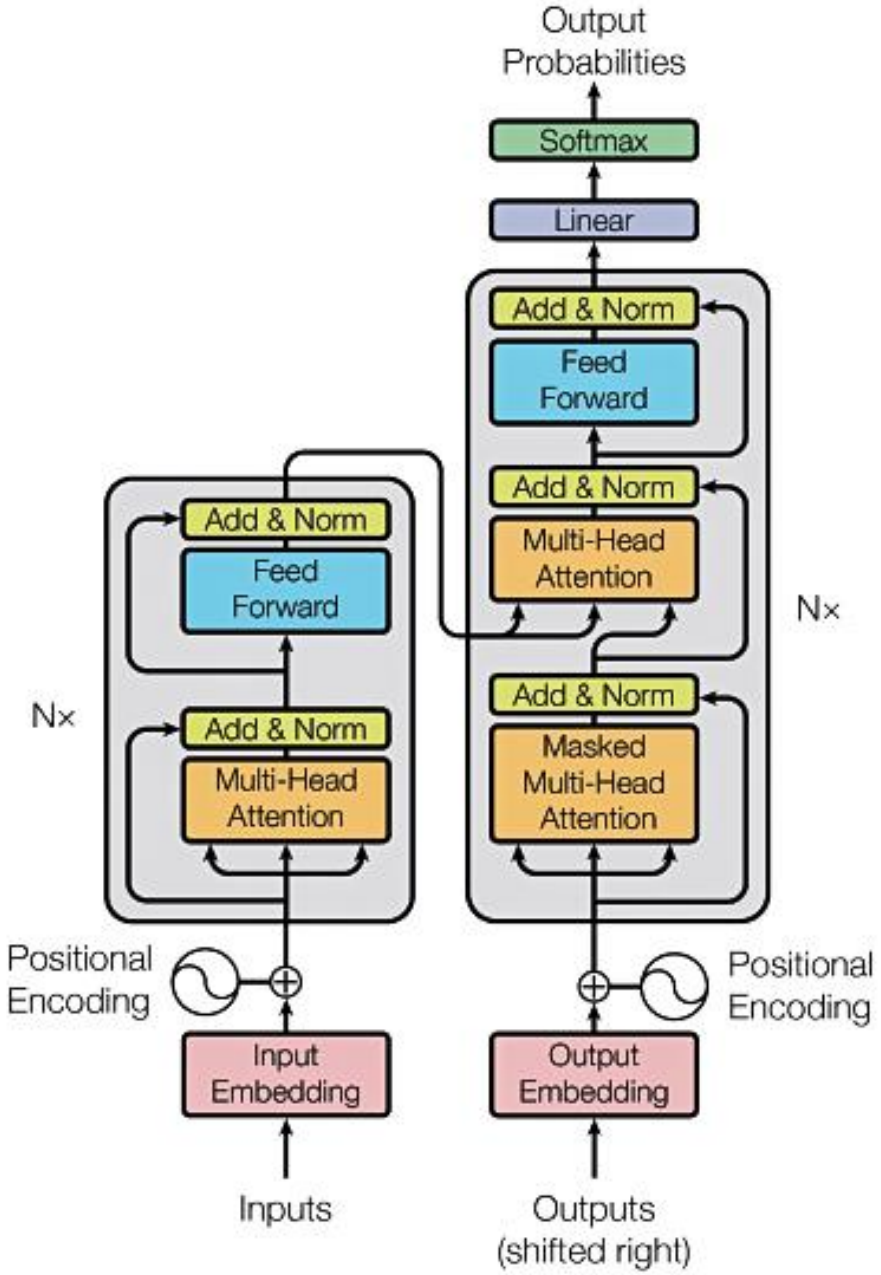
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

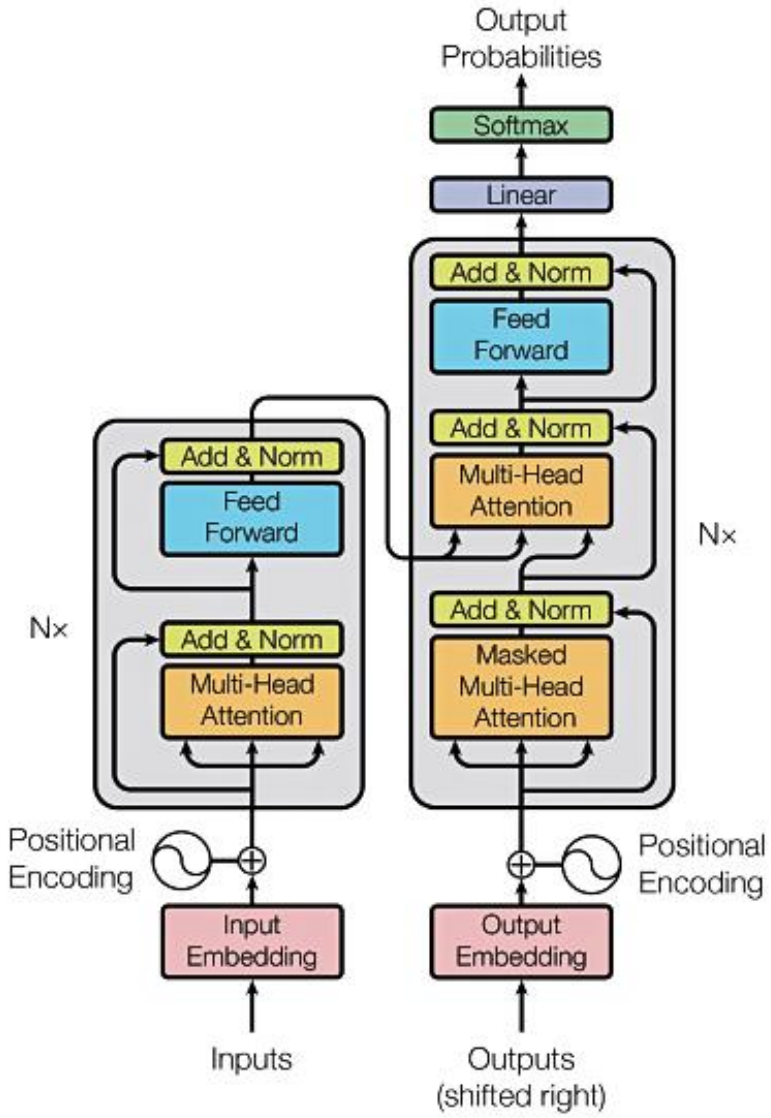
Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention



Transformers

- No recurrent units!
  - All the input is given at once! Increased parallelism!
- Added self-attention mechanisms
  - Identifies dependencies and relationships in the context

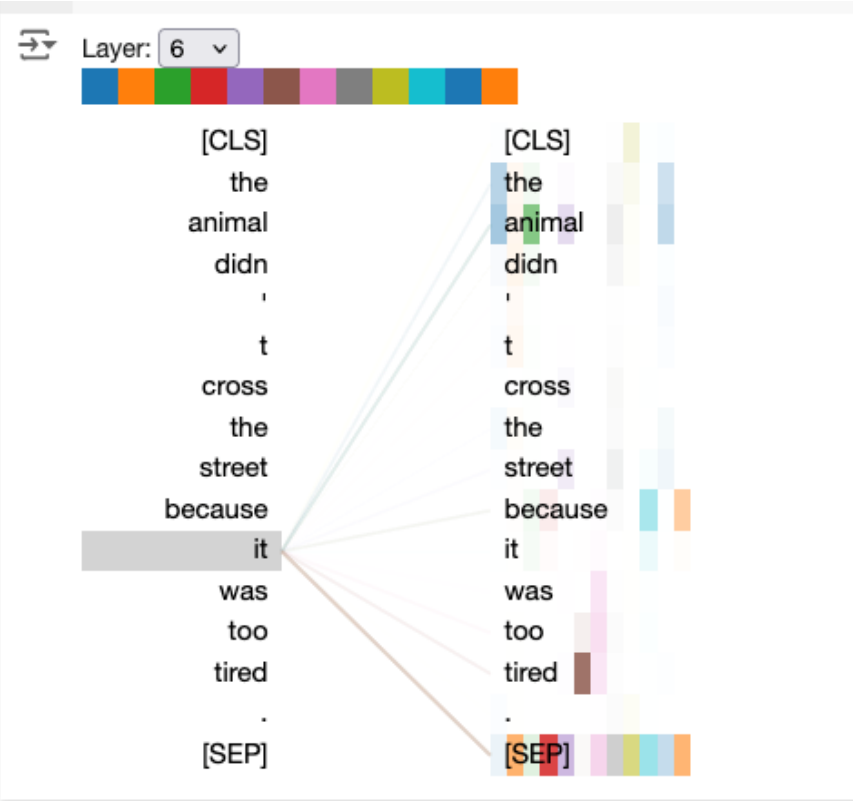


Attention example

- What does “it” refer to in the following sentence?

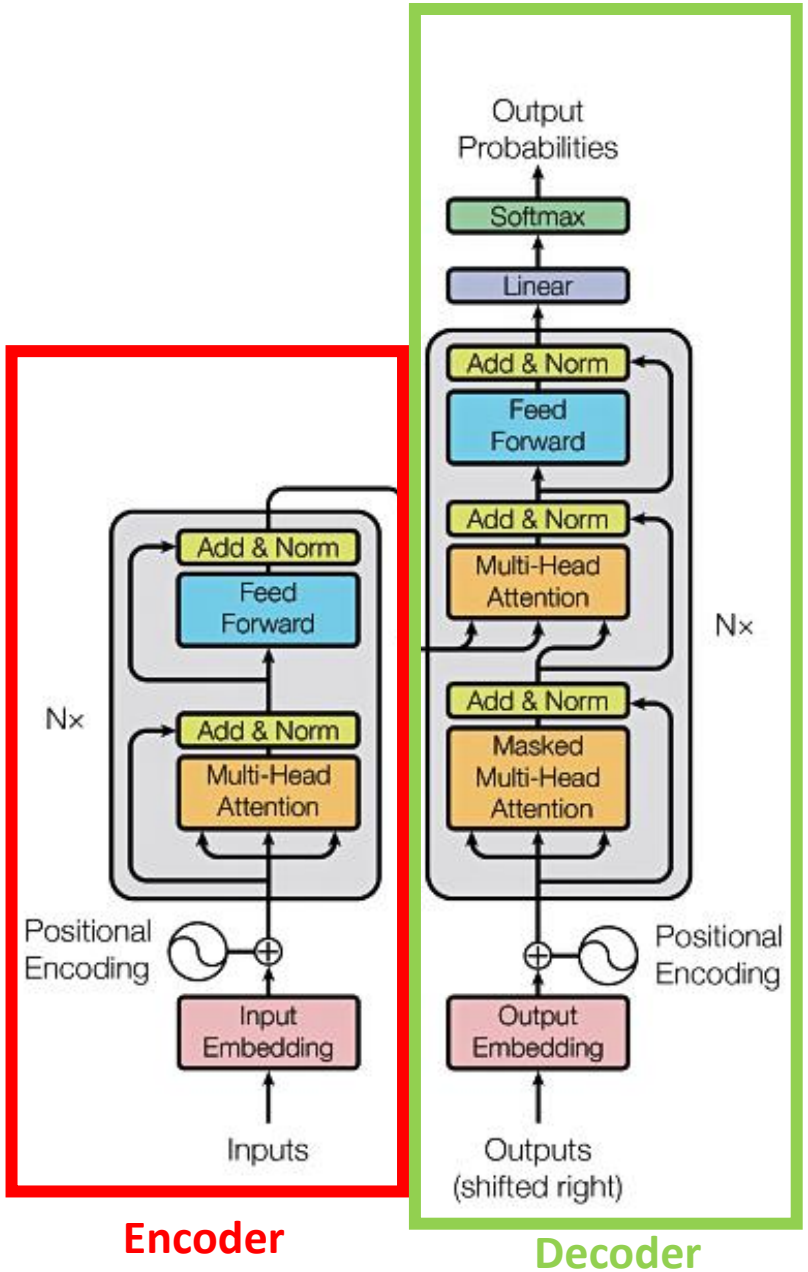
  “The animal didn’t cross the street because it was too tired.”

- Let’s check in BertViz



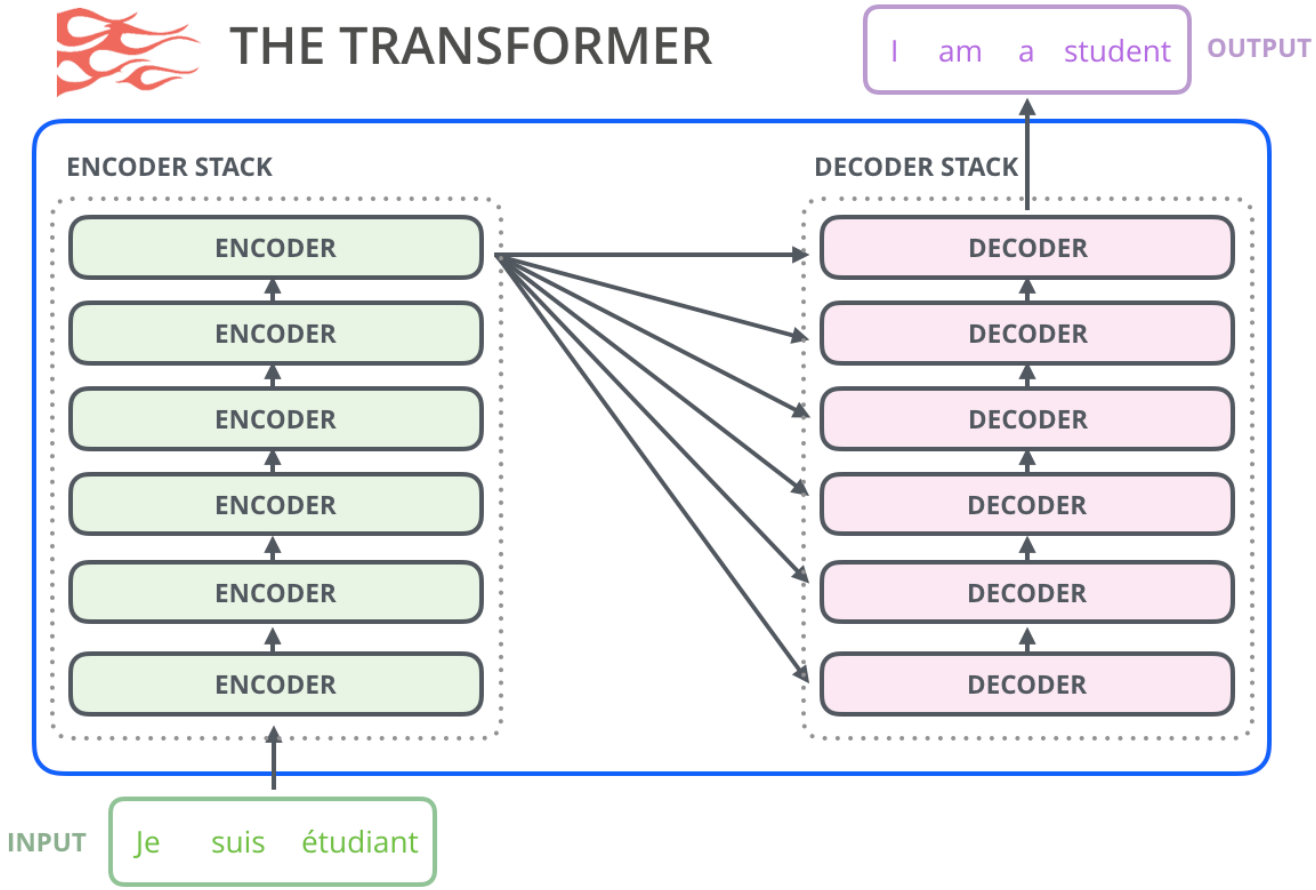
Transformers

- Divide the problem into:
  - Encoder
  - Decoder
- Encoder: represents each item in the input as a vector (the context)
- Decoder: produces the output based on the context



Transformers stack

- Transformers blocks are then stacked
- Goal: capture more complex dependencies



Source: https://jalammar.github.io/illustrated-gpt2/

Encoder-only models

- Only use encoders
- The model has access to all words in the input

- Best suited for tasks requiring the understanding of the full sentence
  - Example: sentence classification

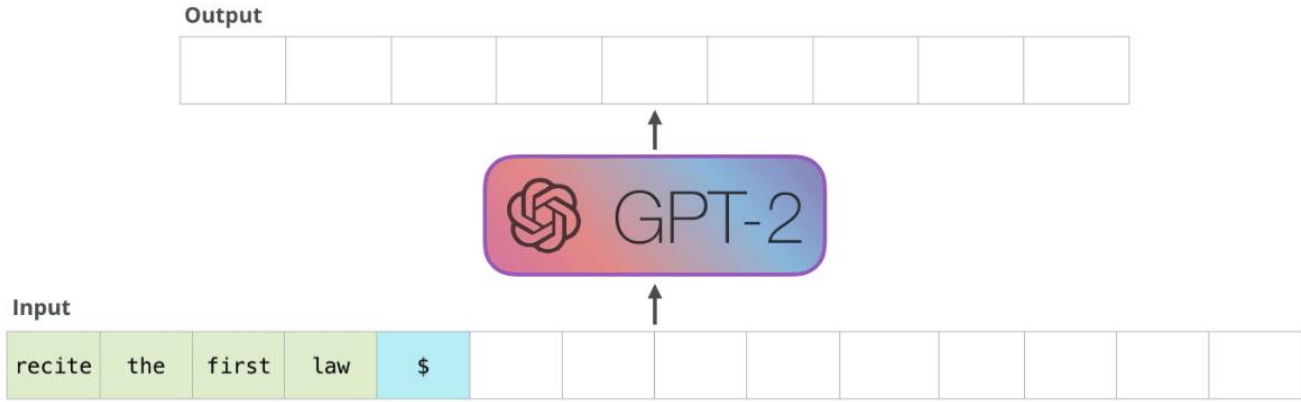
- Example: BERT

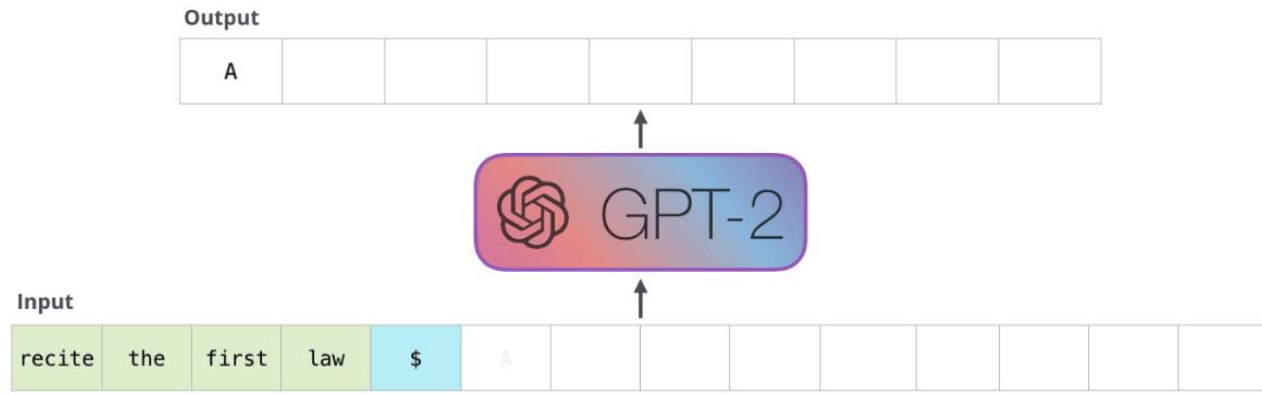
Decoder-only models

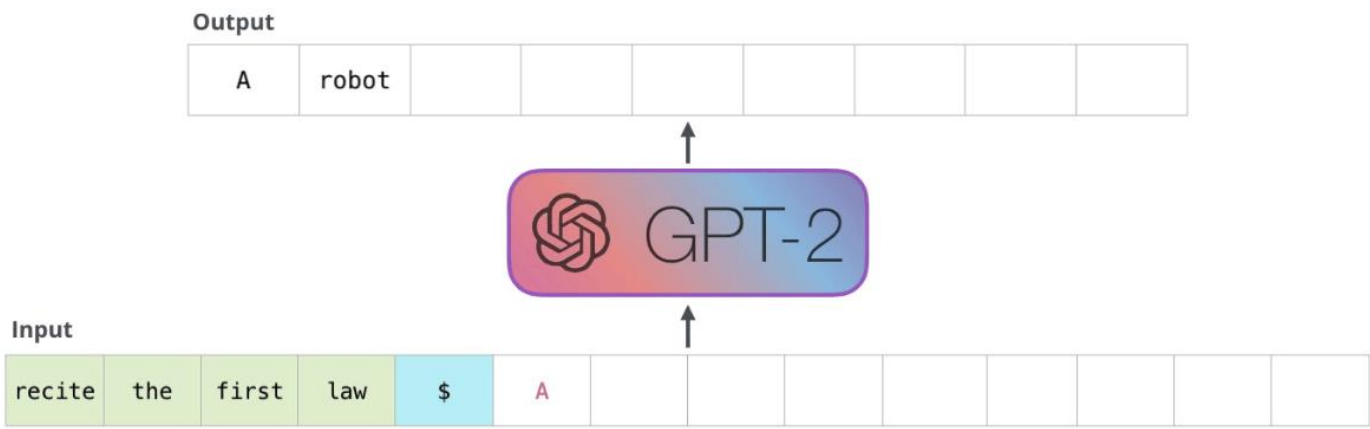
- Only use decoders
- The model has access only to the words before the current word in the sentence

- Best suited for text generation

- Example: Open AI’s GPT


Output
GPT-2
Input
recite
the
first
law
$


Output
A
GPT-2
Input
recite
the
first
law
$
A


Output
A
robot
GPT-2
Input
recite
the
first
law
$
A

Encoder-decoder models

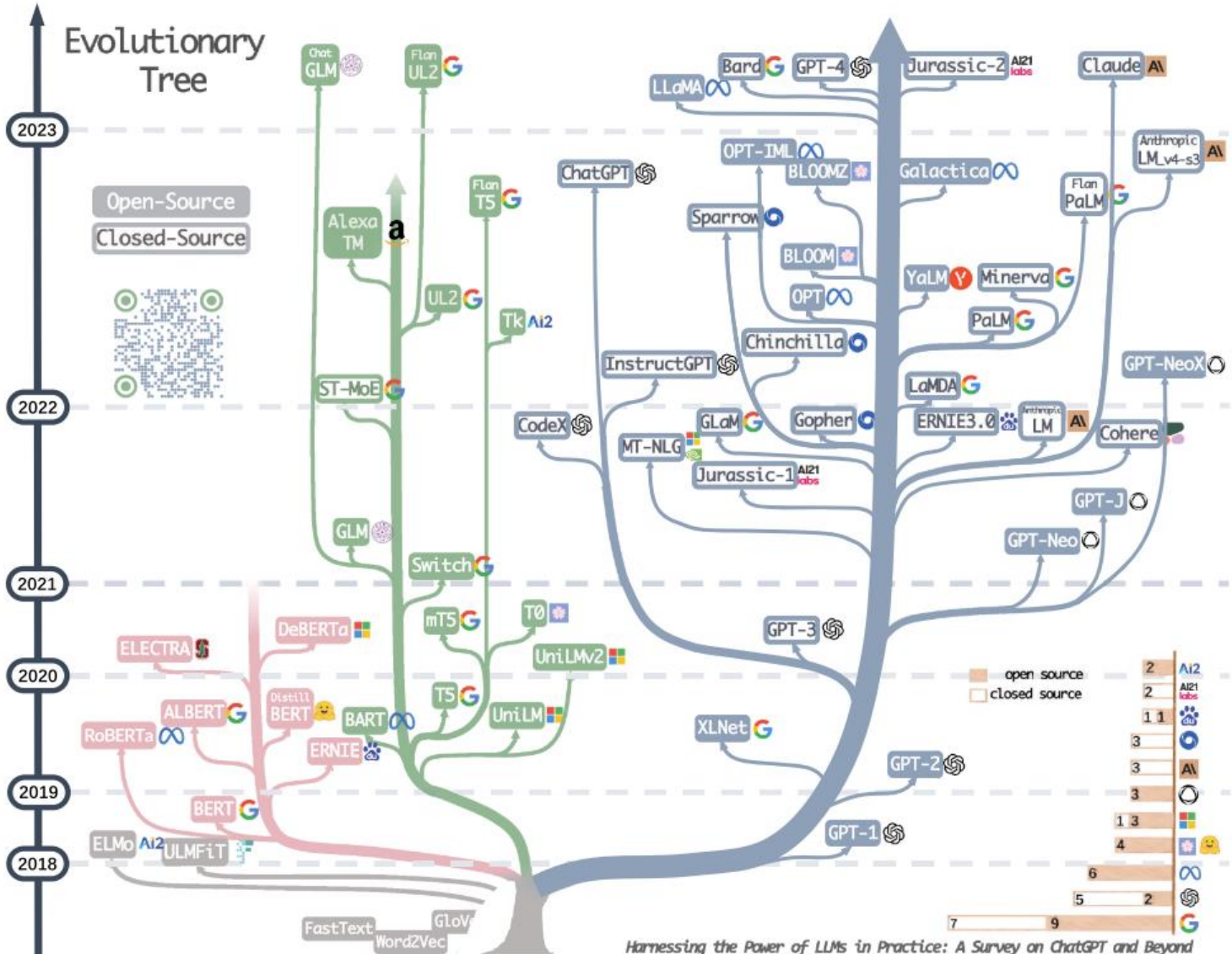
- Also called sequence-to-sequence models

- Use both parts of the transformer
- The encoder has access to all the words of the input
- The decoder has access only to the words before the current one

- Best suited for generation based on a given input
  - Example: summarization and translation

- Examples: T5, BART

Encoder-only and decoder-only models



Foundational models (FMs)

- Foundational models are models trained on broad data at scale such that they **can be adapted** to a wide range of **downstream tasks**.
  - Examples: BERT, GPT-3, Codex

Source: https://hai.stanford.edu/news/introducing-center-research-foundation-models-crfm

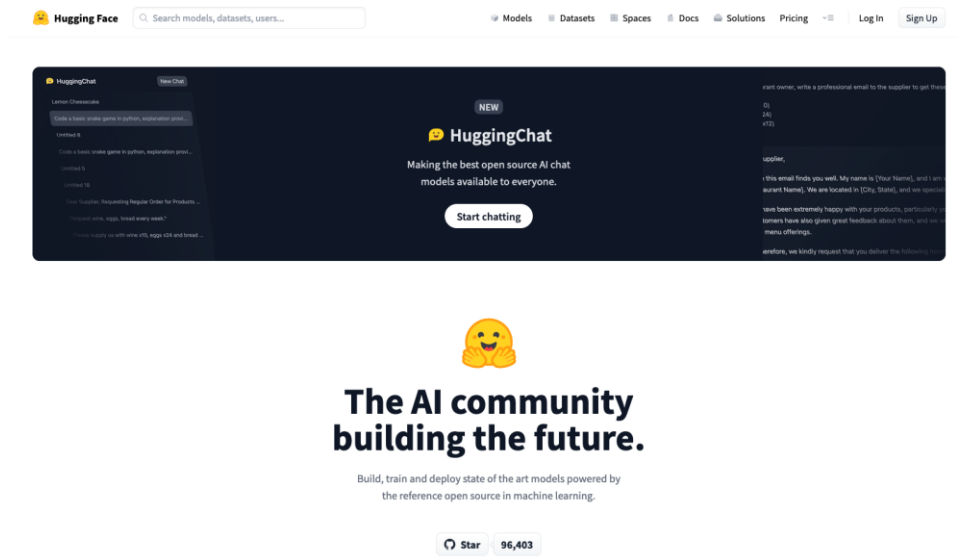
LLMs vs FMs

- Not all foundational models are LLMs (but generally they are)
  - Example: Meta’s SAM (Segment Anything Model)

Pre-trained models

- HuggingFace: a catalog of pre-trained models
  - https://huggingface.co/docs/transformers/index

- It also provides the transformers library (https://github.com/huggingface/transformers)



How can FMs solve specific tasks?

- Traditional way: fine-tuning
  - Requires task-specific datasets of thousands or tens of thousands of examples
  - Need of dedicated hardware
  - Know-how on training and running the model

Language model meta-learning

- At training time, the model develop a broad set of pattern recognition abilities
- At inference time, use the abilities to rapidly adapt to or recognize the task
  - In-context learning: use the text input as a form of task specification

Fine-tuning

- Use a model trained for other tasks
- Obtain a dataset for your specific task
- Use this dataset to further train the model (fine-tuning)
- Advantages:
  - Faster and cheaper than training from scratch
  - Greater variability from the original dataset

BERT

- **B**idirectional **E**ncoder **R**epresentations from **T**ransformers

- A language representation model
  - Useful for many NLP tasks

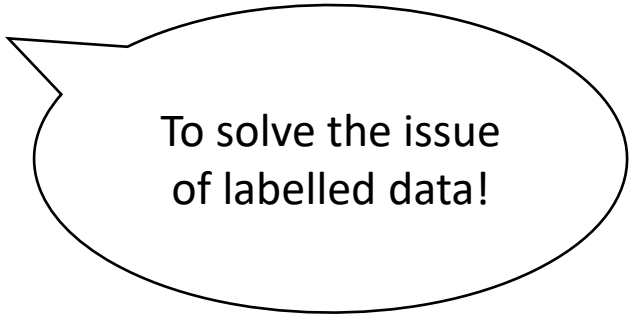
- WordPiece embeddings

- Encoder-only
- Two sizes: base and large
  - Base: 12 transformer blocks
  - Large: 24 transformer blocks

Pre-training BERT

- Masked Language Modelling (MLM)
- Next Sentence Prediction (NSP)

- [CLS], [SEP], [MASK] special characters



Masked language model (MLM)

- Masking: replacing a token with a special token ([MASK])
- Similar to “fill in the blanks” or cloze
- Approach: mask some percentage of the input then predict those masked tokens
- For BERT, 15% of input tokens masked at random

Pre-training BERT - Data

- Data:
  - BooksCorpus (800M words)
  - English Wikipedia (2,500M words)

CodeBERT

- A model based on the BERT (RoBERTa) architecture pre-trained on natural language and programming language tasks:
  - Masked Language Modelling (MLM)
  - Replaced Token Detection (RTD)

- Fine-tuning for downstream tasks

CodeBERT – Pre-training

- Two segments separated with [SEP] special token
  - The first is natural language
  - The second is programming language

- Pre-trained with bimodal and unimodal (code only) data

```
def _parse_memory(s):
    """
    Parse a memory string in the format supported by Java (e.g. 1g, 200m) and
    return the value in MiB

    >>> _parse_memory("256m")
    256
    >>> _parse_memory("2g")
    2048
    """
    units = {'g': 1024, 'm': 1, 't': 1 << 20, 'k': 1.0 / 1024}
    if s[-1].lower() not in units:
        raise ValueError("invalid format: " + s)
    return int(float(s[:-1]) * units[s[-1].lower()])
```

Figure 1: An example of the NL-PL pair, where NL is the first paragraph (filled in red) from the documentation (dashed line in black) of a function.

CodeBERT – Pre-training

- MLM: only bimodal data
- RTD: bimodal and unimodal data
  - Replace some tokens in the original input data with wrong ones
  - The model should predict if the tokens are correct or not

CodeBERT – Pre-training data

- Codesearchnet dataset
- 2.1M bimodal and 6.4M unimodal datapoints
- Six programming languages
  - Python
  - Java
  - JavaScript
  - PHP
  - Ruby
  - Go

CodeBERT example: Vulnerability detection

- Give a line (or block), tell if it contains a security vulnerability or not

LineVul: A Transformer-based Line-Level Vulnerability Prediction

Michael Fu
michael.fu@monash.edu
Monash University
Australia

Chakkrit Tantithamthavorn*
chakkrit@monash.edu
Monash University
Australia

ABSTRACT

Software vulnerabilities are prevalent in software systems, causing a variety of problems including deadlock, information loss, or system failures. Thus, early predictions of software vulnerabilities are critically important in safety-critical software systems. Various ML/DL-based approaches have been proposed to predict vulnerabilities at the file/function/method level. Recently, IVDetect (a graph-based neural network) is proposed to predict vulnerabilities at the function level. Yet, the IVDetect approach is still inaccurate and coarse-grained. In this paper, we propose LineVul, a Transformer-based line-level vulnerability prediction approach in order to address several limitations of the state-of-the-art IVDetect approach. Through an empirical evaluation of a large-scale real-world dataset with 188k+ C/C++ functions, we show that LineVul achieves (1) 160%-379% higher F1-measure for function-level predictions; (2) 12%-25% higher Top-10 Accuracy for line-level predictions; and (3) 29%-53% less Effort@20%Recall than the baseline approaches, highlighting the significant advancement of LineVul towards more accurate and more cost-effective line-level vulnerability predictions. Our additional analysis also shows that our LineVul is also very accurate (75%-100%) for predicting vulnerable functions affected by the Top-25 most dangerous CWEs, highlighting the potential impact of our LineVul in real-world usage scenarios.

ACM Reference Format:
Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3524842.3528452

Server, that allowed an attacker to bypass the authentication and impersonating. Therefore, an unauthenticated attacker could execute arbitrary commands on the server. Cybersecurity Ventures expects global cybercrime costs to reach $10.5 trillion USD by 2025, up from $3 trillion USD in 2015 [4]. As cyberattacks become the main contributing factors to revenue loss of some businesses [10], ensuring the safety of software systems becomes one of the critical challenges of private and public sectors.

To mitigate this challenge, program analysis (PA) tools [1, 2, 5, 9] have been introduced to analyze source code using predefined vulnerability patterns. For instance, Gupta *et al.* [19] found that there are static analysis approaches that were used to detect SQL injection and cross-site scripting vulnerabilities. Both PA-based and ML/DL-based approaches fall short of the capability to detect fine-grained vulnerabilities." – PA tools, including ones cited in the paper, highlight the specific lines in code that contain the detected vulnerabilities.

On the other hand, Machine Learning (ML) / Deep Learning (DL) approaches have been proposed. Specifically, these ML/DL-based approaches [12, 32, 33, 43, 65] first generate a representation of source code in order to learn vulnerability patterns. Finally, such approaches will learn the relationship between the representation of source code and the ground-truth (i.e., whether a given piece of code is vulnerable). Despite the advantages of dynamically learning the vulnerability patterns without manual predefined vulnerability patterns, previous ML/DL-based approaches still focus on coarse-grained vulnerability prediction where models only point out vulnerabilities at the file level or the function level—which is still coarse-grained.

Recently, Li *et al.* [30] proposed an IVDetect approach to address

CodeBERT examples

while the PyTorch library supports the computation during the training process (e.g., back-propagation and parameter optimization). We download the CodeBERT tokenizer and CodeBERT model pre-trained by Feng *et al.* [17]. We use our training dataset to fine-tune the pre-trained model to get suitable weights for our vulnerability prediction task. The model was fine-tuned on an NVIDIA RTX 3090 graphic card and the training time was around 7 hours and 10 minutes. As shown in Equation 1, the Cross Entropy Loss

LineVul (Fu & Tantithamthavorn, 2022)

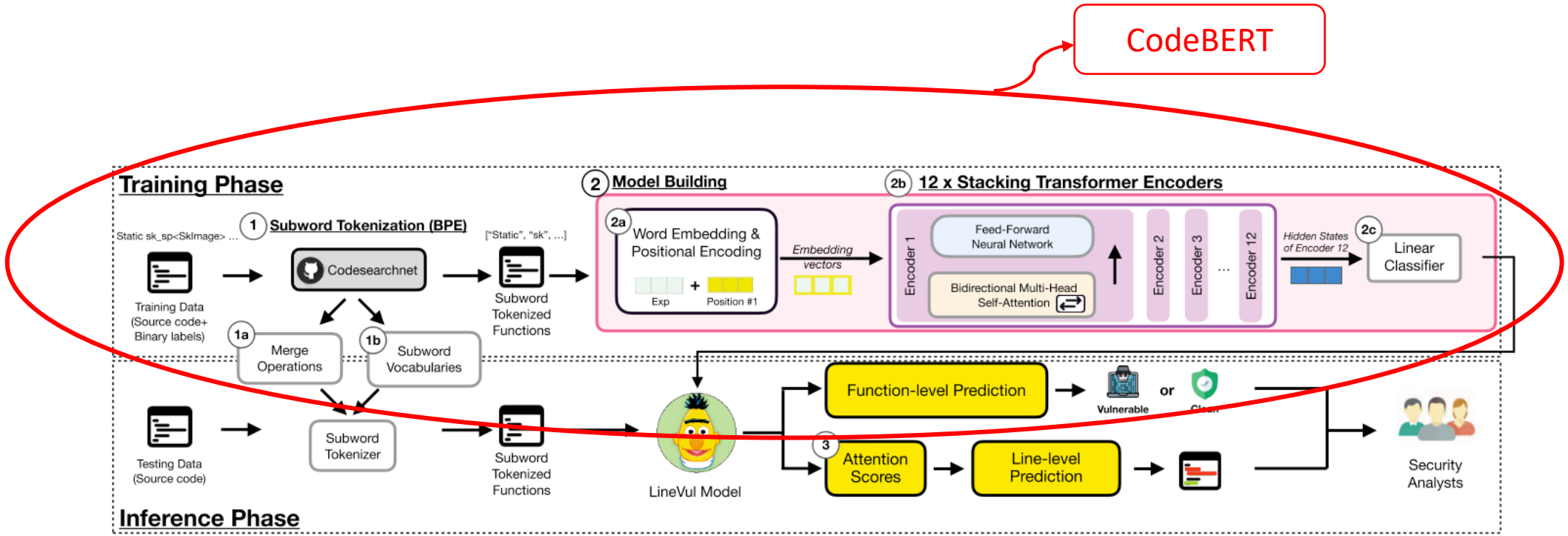


Figure 2: An overview architecture of our LineVul.

Other example

Applying CodeBERT for Automated Program Repair of Java Simple Bugs

Ehsan Mashhadi
Schulich School of Engineering
University of Calgary
Calgary, Canada
ehsan.mashhadi@ucalgary.ca

Hadi Hemmati
Schulich School of Engineering
University of Calgary
Calgary, Canada
hadi.hemmati@ucalgary.ca

***Abstract*—Software debugging, and program repair are among the most time-consuming and labor-intensive tasks in software engineering that would benefit a lot from automation. In this paper, we propose a novel automated program repair approach based on CodeBERT, which is a transformer-based neural architecture pre-trained on large corpus of source code. We fine-tune our model on the ManySStuBs4J small and large datasets to automatically generate the fix codes. The results show that our technique accurately predicts the fixed codes implemented by the developers in 19-72% of the cases, depending on the type of datasets, in less than a second per bug. We also observe that our method can generate varied-length fixes (short and long) and can fix different types of bugs, even if only a few instances of those types of bugs exists in the training dataset.**

***Index Terms*—Program repair, CodeBERT, Sequence to sequence learning, Transformers, Deep learning.**

Given the promising results on similar tasks, in this paper, we leveraged CodeBERT to automatically generate fixes for bugs reported on the ManySStuBs4J dataset [9]. The ManySStuBs4J dataset focuses on Java simple bugs that appear on a single statement and the corresponding fix is within that statement. To guide our investigation, we target answering the following research questions:

RQ1. Can CodeBERT be used to fix Java simple bugs, and what are the pros and cons? We found that our approach has an accuracy of 72% and 68.8% for the large and small versions of the dataset, respectively. Also, we observed that the accuracy is reduced to 23.7% and 19.65%, respectively, for unique datasets (after removing duplicate fixes from the datasets). Our approach does not require any special tokens for

Difficulties of fine-tuning

- Need for specific labelled **data**
- Need for specific and expensive **hardware**
- Need for specific **expertise**

Cost/infrastructure needed to run

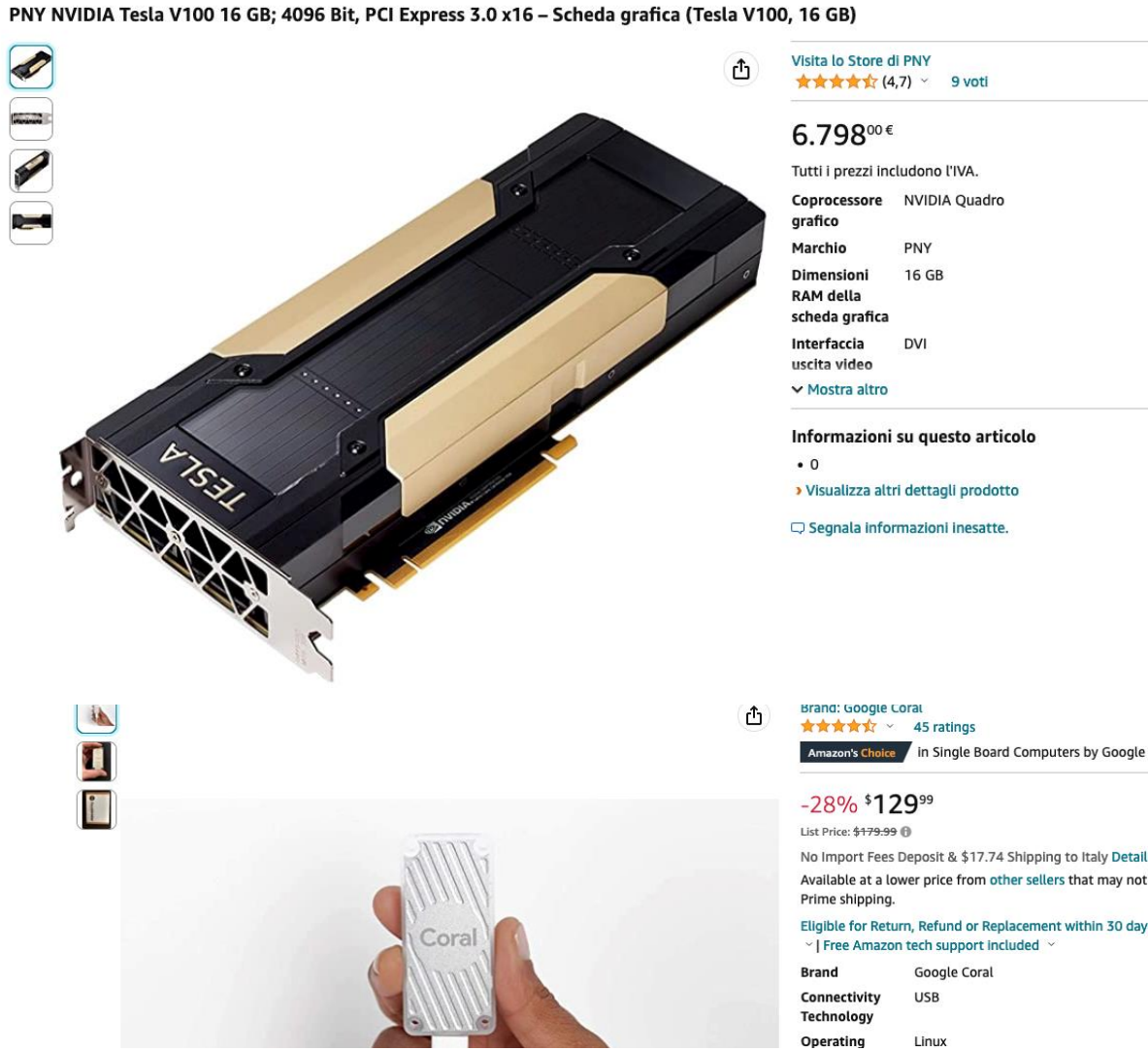
- Deep learning models are implemented using matrices and tensor multiplications

- Dependence on GPU (Graphics Processing Unit) or TPU (Tensor Processing Units)

Cost/infrastructure needed to run

- CodeBERT was trained on one NVIDIA DGX-2 machine combining 16 interconnected NVIDIA Tesla V100 with 32GB memory.

- TPU price on cloud
  - https://cloud.google.com/tpu/pricing



Time/difficulty to train/run

- Scarcity of labelled datasets
- Trained on a NVIDIA RTX 3090 for around 7 hours
- LineVul was fine-tuned using Big-Vul
  - ~189k functions with around ~11k vulnerable

The process of fine-tuning

- Similar to the training process…
- … but the model's weights start with the values obtained in the pre-training

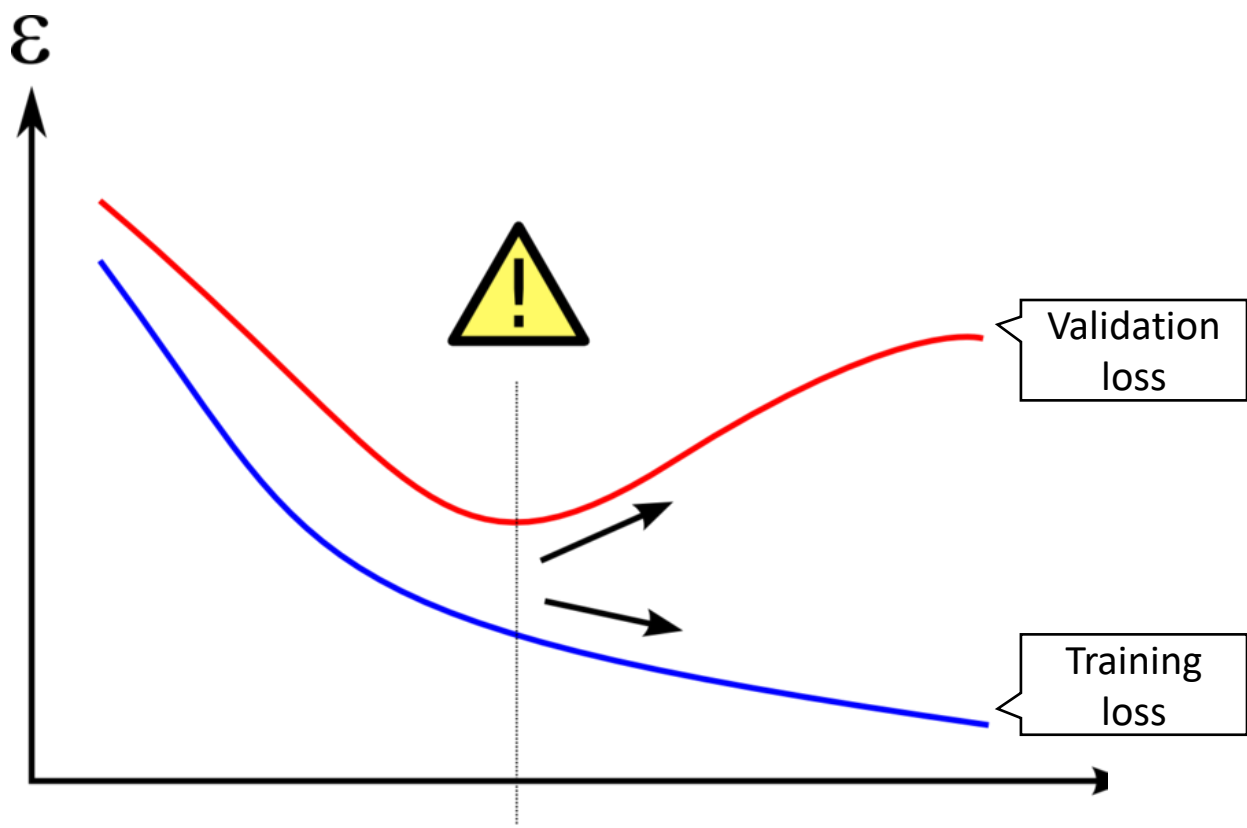
- Use the model to predict the output for the inputs
- Calculate the loss function (error) between the correct outputs and the predict ones
- Update the weights based on the loss

Dataset splitting

- Train, validation and test datasets
- Train dataset: examples given to the model used to update its parameters
- Validation dataset: examples not in the train dataset used to evaluate the model fit. Used, for example, for tuning the hyperparameters or early stopping
- Test dataset: examples not in the previous datasets used to evaluate the final model

Overfitting

- A validation dataset can be useful to see if the model is overfitting
- The model is getting better on the training, but it is not “learning” more



Source: https://commons.wikimedia.org/wiki/File:Overfitting.png
By Dake~commonswiki License: CC-BY-SA 3.0

Dataset splitting – how?

- The train dataset is larger -> giving more examples for the model learning
- In some cases, the validation is not used
- A common proportion is: 80%:10%:10%

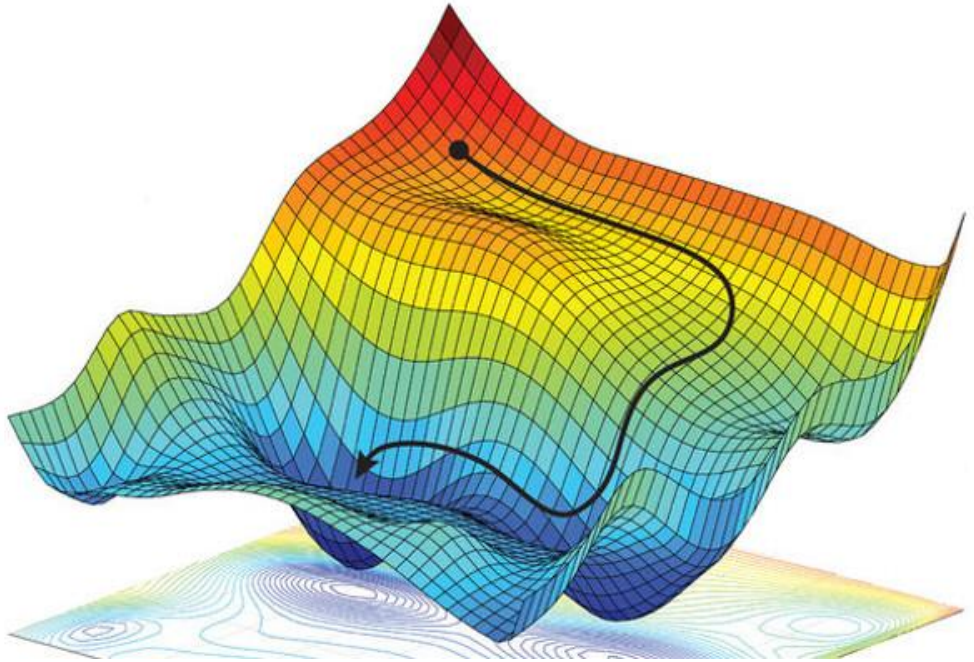
Hyperparameters

- Parameters refer to the weights inside the model
  - Example: CodeLlama with 7 billion parameters

- Hyperparameters: configuration variables to manage the training process

Common hyperparameters

- Learning rate
  - “How much” the weights will be changed in each training step
  - Higher learning rate => faster learning but higher chance of not converging



Source: Amini, A., Soleimany, A., Karaman, S., & Rus, D. (2018). Spatial uncertainty sampling for end-to-end control. *arXiv preprint arXiv:1805.04829*.

Common hyperparameters

- Number of epochs
  - Number of times the training data will be shown to the model
- Batch size
  - During training, data is fed into batches to the model rather than one by one
  - Common values are powers of 2: 16, 32, 64,128

Example (Jupyter notebook)

- Fine-tuning CodeBERT for detecting vulnerabilities
- Dataset: Devign
- Open in Google Colab the file VulnerabilityDetection.ipynb from the repository melegati/gen4ai-course
  - If you prefer (you have a powerful machine), you can also run locally!

Performance metrics

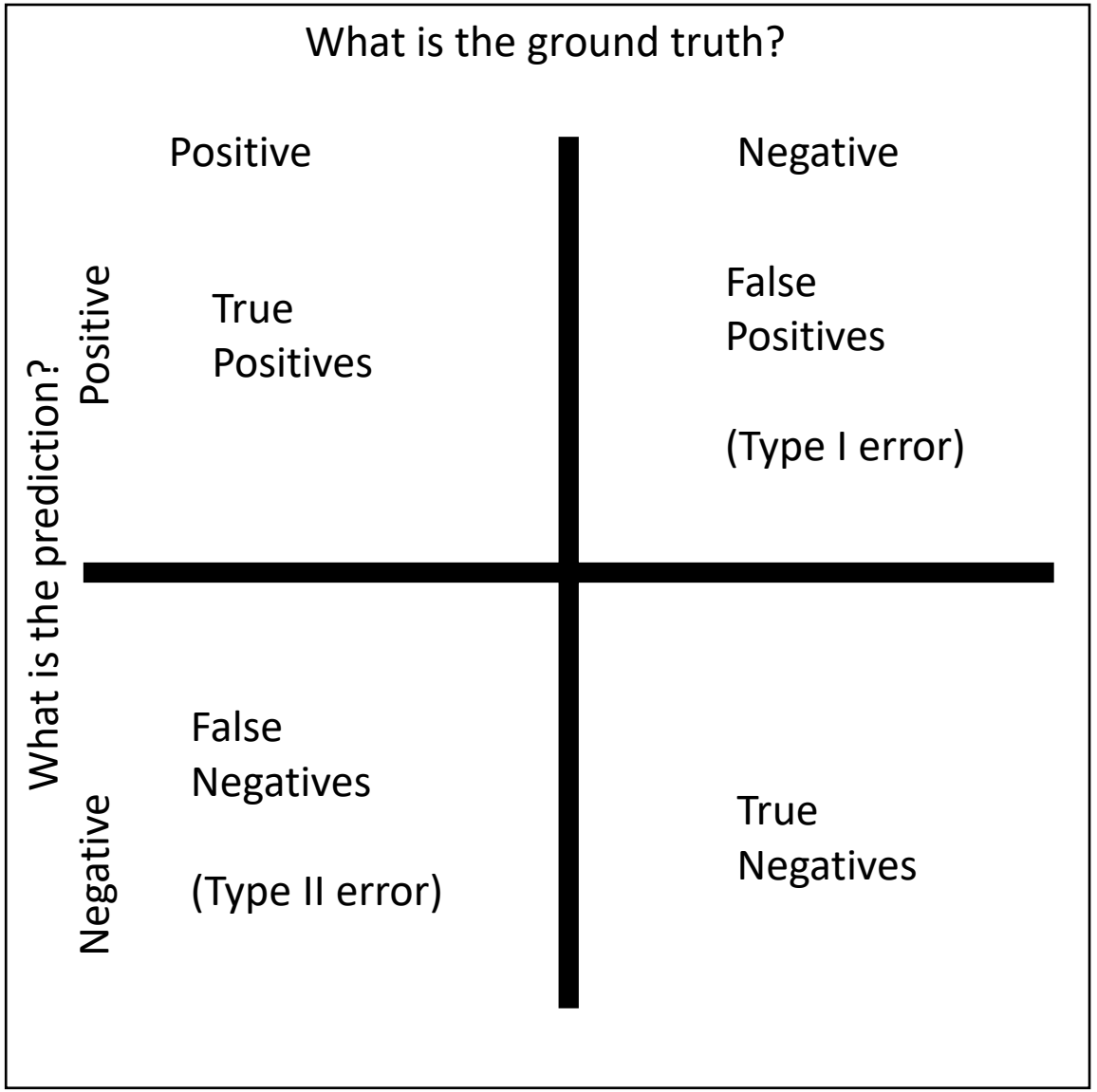
- Metrics to judge the performance of a model
- Depend on the problem
- Let’s focus on metrics for the classification problem

Type of errors on classification

- Type I error
  - False positive
  - Truth is negative but the prediction is positive
  - Examples of bad situations:
    - Criminal conviction
- Type II error
  - False negative
  - Truth is positive but the prediction is negative
  - Example of bad situations:
    - Detection of a disease

Generally, one needs to choose which error type to optimize!

Confusion matrix



Performance metrics

- Precision
  - Regards Type I errors
  - How much can we trust the predictions of positive?

$$Precision = \frac{TP}{TP + FP}$$

- Recall
  - Regards Type II errors
  - How much can we trust that all the positives were predicted?
  - Did we miss anything?

$$Recall = \frac{TP}{TP + FN}$$

Performance metrics for bug prediction

- What is better to have type I error or type II?
  - What are the consequences?
- Which metric should we check? Precision or recall?

Other metrics

- Accuracy
  - Fraction of correct answers among all

$$Accuracy = \frac{TP + TN}{TP + TN + FN + FP}$$

- F1 score
  - Harmonic mean of precision and recall
  - A way to represent both

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall}$$

Exercise

- You have to evaluate a bug detection tool
- You have an internal benchmark of 1000 snippets of code
  - 400 contains bugs
  - 600 are bug-free
- When you run the tool on this benchmark
  - Out of the 400 bugs, it detects 380
  - Out of the 600 bug-free, 120 are erroneously considered bugs

- Present the confusion matrix, calculate precision and recall and discuss where it should be used

$$Precision = \frac{TP}{TP + FP} \qquad Recall = \frac{TP}{TP + FN} \qquad Accuracy = \frac{TP + TN}{TP + TN + FN + FP} \qquad F1 = \frac{2 * Precision * Recall}{Precision + Recall}$$