Machine Learning for Programming Languages

An Overview of Machine Learning for a Software Engineer

Raphael Jenni
OST Eastern Switzerland University of Applied Sciences
MSE Seminar “Programming Languages”
Supervisor: Farhad Mehta
Semester: Autumn 2021

Abstract

Artificial Intelligence, or more precisely deep learning, has become a trending topic in the broad public and software engineering circles. Some exciting technologies have arisen from it, such as voice assistants or language translation services. Also, programmatically understanding source code and supporting the developer in writing better code have been a topic for a while. In recent times, a push toward combining these two fields has been made.

For a software engineer coming from the world of tackling a problem with the help of algorithms with a predictable outcome, deep learning can be rather challenging to grasp. This paper aims to bring a software engineer or a programming language researcher up to speed on the current state of deep learning and show the possibilities of such technologies. All this, in an easily digestible manner for someone without any profound knowledge about deep learning.

Keywords: Programming Languages, Software Engineering, Deep Learning, Machine Learning

1 Introduction

Deep learning is a field of computer science that uses artificial neural networks to learn representations of data. This technology has enabled the rapid development of methods used to classify, recognize, and understand objects, images, and other data. In recent times using deep learning has become a relatively common application of artificial intelligence. The advances in computational power and publicly available data were the leading enabler for that. Nevertheless, deep learning, or more generally, machine learning, is a topic most software engineers only know from hearing about its use cases but rarely really know how to use it.

This paper aims to provide a general overview of the deep learning field and introduce the use cases of deep learning in the context of programming languages.

1.1 Assumed Knowledge

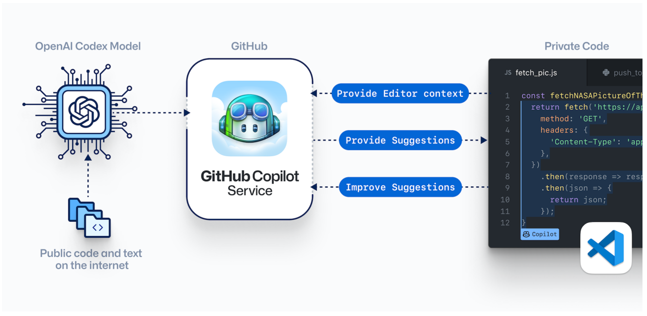
This paper assumes that the reader knows the basics of programming language constructs and is able to program.

1.2 Motivating Examples

Before diving into the main content of the paper, we will look at four examples. Those should help get a feel for the paper's general idea and some possible use cases of DL4PL.

1.2.1 GitHub Copilot. One of the most significant projects at the time of writing this paper is the *GitHub Copilot*[1]. The GitHub Copilot is an AI-powered code assistant powered by Codex[2], an AI system created by OpenAI [CTJ+21].

It supports dozens of programming languages and works (see Figure 1) by sending the code context to the GitHub Copilot Service, which the OpenAI Codex Model trains based on publicly available repositories. This service then sends back suggestions and receives back a report regarding the developer's choice to further improve the model.



```
const fetchNASAPictureOfTh
  return fetch('https://ap
    method: 'GET',
    headers: {
      'Content-Type': 'app
    },
  })
    .then(response => resp
    .then(json => {
      return json;
    });
}
```

Figure 1. General Workings of the GitHub Copilot [Git21]

An example would be writing a Fibonacci function in Kotlin. By just providing function name `fun fibonacci`, the copilot suggests the following code completion:

```kotlin
fun fibonacci(n: Int): Int {
    if (n == 0) return 0
    if (n == 1) return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
}
```

Listing 1. Autocompleted Fibonacci Function

This assistance by itself is already pretty nice, but it is just a relatively common code snippet. The following code demonstrates how the copilot leverages the context to give suggestions.

[1] https://copilot.github.com
[2] https://openai.com/blog/openai-codex/

```kotlin
// data class Person  -  gets completed to:
data class Person(var name: String, var age: Int)

// val attendees = listOf(Person(  -   results in:
val attendees = listOf(
    Person("John", 20),
    Person("Jane", 25),
    Person("Jack", 30)
)

// fun averageAgeOfPersons  -  gets completed to
fun averageAgeOfPersons(persons: List<Person>):
     Double {
    return persons.fold(0.0)
        { acc, person -> acc + person.age } /
             persons.size
}
```

Listing 2. Extended example of the Copilot's capabilities

By just having the definition of a person and providing the name of the function we want to write, the copilot could generate a working piece of code that does what was intended.

Such coding support is already a huge help when writing code. The GitHub Copilot website has a catchy sentence: "Skip the docs and stop searching for examples. GitHub Copilot helps you stay focused right in your editor [Git21]." By providing code completion for relatively simple but maybe not familiar code constructs, this aid can boost productivity quite a bit. Take the Fibonacci function from above (Listing 1) as an example. The function itself is straightforward and well known but still requires a bit of thinking to get it right on the first try. Many developers will not bother to write it by themselves anyway and turn to the internet for help. Moving the "internet" directly into the IDE results in fewer context switches and therefor less downtime in general.

The copilot has more features like converting a function description into code or suggesting tests for your code. Those will not be covered here but can be tested by yourself[3].

1.2.2 Tabnine. Moving on from the GitHub Copilot to a very similar product, Tabnine[4]. It completes code with the help of deep learning similar to the copilot but has a significant focus on privacy. Besides the public-code-trained AI, the service also offers to run locally on your or your team's code. Suggestion improvements will also not be reported back to the Tabnine services and are only available for you and your team. Tabnine also supports a wide range of IDEs. For a coding assistant, this is an important point worth considering.

I have been using Tabnine for about half a year at the time of writing and have found it very useful. When comparing it to the GitHub Copilot, the copilot seems to have the edge in completing large code constructs. In a matter of speed or accuracy, the Tabnine service feels more advanced. Currently, both services are running in parallel, delivering the best of both worlds and demonstrating that such tools do not have to be used exclusively.

Unfortunately, the Tabnine team has not released any specific information, papers or code, that shows how the internals work. This lack of insights means that the user has to trust what is released on their website.

[3]https://copilot.github.com
[4]https://www.tabnine.com

1.2.3 DeepCode by Snyk. DeepCode was an ETH Zurich spin-off that got acquired by snyk[5] in 2020. Their focus was on applying AI to assist developers in achieving better code quality and application security. They leverage commits and big code to learn how software can help fix specific issues and vulnerabilities. The DeepCode team has published several papers on those topics, although not with a special connection to deep learning [RZVY, ESR+, RBV, RVK, BRTV, BRV16, CBR+, PZT+18, RBVK, HZT+18, BRV].

Snyk combines the work of DeepCode with other tools regarding security and code quality with the help of machine learning.

1.2.4 ControlFlag by Intel. ControlFlag is a system developed by Intel. It does not use deep learning but shows another possibility of machine learning for programming languages. The goal of it is to detect violations of programming patterns in conditional statements like `if` or `for` and suggests possible fixes for it. An example of such a coding error would be:

```c
for (int i = 0; i < N; i++) {
    // Some program code
    i++;
}
```

Listing 3. Example of possible coding error [HG21]

This example shows an instance of code that compiles but is probably written this way on accident. Increasing the counter twice can be correct but usually is not the intended way to use a for-loop. Same goes for the statement `if ( x = 7 )y = x`, that does an assignment of x instead of a comparison. Most likely, the desired behavior is `if ( x == 7 )y = x`. This additional equal sign is a relatively small change but one that could have a tremendous impact.

ControlFlag uses big code to learn the bug patterns as well. The more a pattern gets used, the higher the certainty of correctness. Based on those learned patterns and their correctness, ControlFlag can suggest possible fixes.

1.2.5 Related Work. As utilizing deep learning for code is a relatively new field, not many finished products are available. One system that uses deep learning for detecting and correcting bugs, like shown with the example of ControlFlag, but written for JavaScript, is called Hoppity [DDB+]. Other than that, primarily ideas are available in the form of papers. For more papers covering different areas of deep learning for code or, more generally, for software engineering, check out the references in [WCP+20].

[5]https://snyk.io

2 Deep Learning for Software Engineers

Deep learning has developed into a trending topic. A software engineer certainly has heard of it and knows what it is but has a somewhat limited understanding of it most of the time. This section aims to shed some light on common confusions regarding the naming, explains the general idea of deep learning and its use cases, and discusses the most prominent models used and how they differ.

2.1 Clarifying the Confusion In the Naming

When talking casually with colleagues, *Artificial Intelligence* (AI), *Machine Learning* (ML), *Neural Networks* (NN), and *Deep Learning* (DL) are often used interchangeably. While it is not always wrong, it is usually used in a generalized way or in an uninformed and undistinguished manner. The easiest way to think about the four designations is to take them as specializations of each other or being nested inside each other (Figure 2).

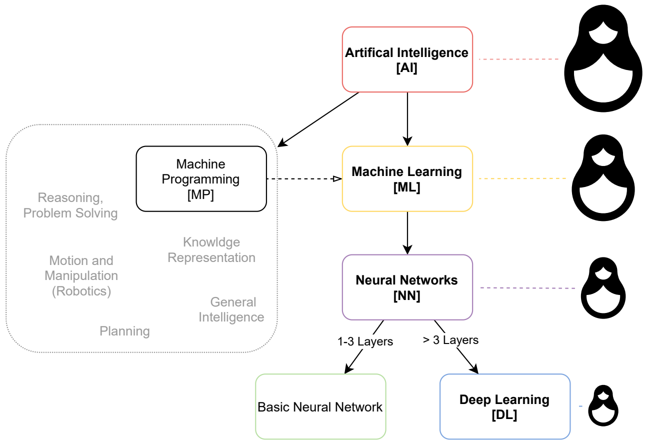


Figure 2. Visualization of the different AI designations and their relationships. The nesting visualized as Russian Dolls.

2.1.1 Artificial Intelligence (AI).

Artificial Intelligence is the most general category. It can be defined as "Artificial intelligence leverages computers and machines to mimic the problem-solving and decision-making capabilities of the human mind [IBM20d]." It has been around for a long time, dating back to 1950, where Alan Turing asked the question "can machines think [Tur50]?". Based on that question, the *Turing Test* was designed to define intelligence. A computer passes this test if a human interrogator cannot differentiate whether an answer to a question came from a human or a machine. This test already covers several AI disciplines used today, like natural language processing, automated reasoning, or machine learning [NRN10]. Since then, AI has developed enormously and in various directions.

2.1.2 Machine Learning (ML).

Going one step deeper in the direction of deep learning, there is the field of *Machine Learning*. ML is a branch of AI that uses data to learn and improve over time. In traditional ML, a human has to preprocess a lot of the data to be learned [IBM20e]. However, learning can happen in several different ways.

Supervised learning utilizes data that is labeled to categorize or predict something. For example, this approach is used for image recognition, where the computer tries to say whether it is an image of a cat or a dog.

Unsupervised learning does not require labels but instead tries to find common patterns in the data and build clusters. A possible use case would be finding common customer behavior patterns or automatically developing a concept of "good" and "bad" traffic days [NRN10]. Unsupervised learning can also be used as a preprocessing step for reducing the number of features (specific information) in a dataset before leveraging other machine learning techniques.

Reinforced learning is the method used, when the learning process tries to improve the model by maximizing or minimizing the resulting score based on some received feedback. This method is widely used in games where the computer tries to finish the race in the least amount of time. Another example would be maximizing the received tip after a taxi ride [NRN10].

2.1.3 Machine Programming (MP).

Machine Programming (MP) is a relatively new specialization of ML. It is all about programming with the help of machine learning. The goal is to have a system that can produce secure, correct, and efficient code. Further, it aims to enable non-programmers to solve problems correctly and efficiently without the need to be able to code. [GSLT+]

The GitHub Copilot, Tabnine, or ControlFlag introduced in subsection 1.2, are good MP examples.

2.1.4 Neural Networks (NN).

Next in line, we have *neural networks*, sometimes also called *artificial neural networks* (ANN). They work by mimicking the behavior of the human brain to enable the computer to recognize patterns. The name "neural" comes from the concept of the brain cells called neurons. As visualized in Figure 3, a neuron receives input data

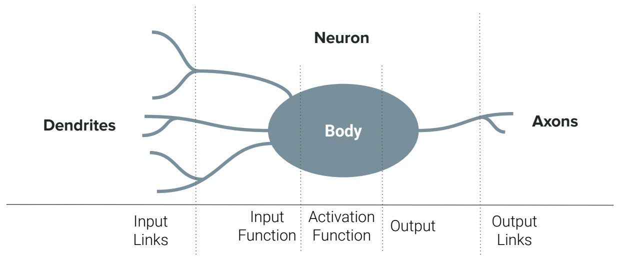


Figure 3. Visualization of a Single Neuron

together with a weight for each connection through the dendrites, which gets fed into an input function. The function outputs a value based on the received information, which gets fed into an activation function. There are many different activation functions, but all have a common goal of reducing the value into a specific space. For example, the frequently used sigmoid function takes any value and outputs a value between 0 and 1. The output gets forwarded to the output links, which can be other neurons or a final state. [NRN10]

A neural network consists of different artificial neuron layers: An input layer, one or more hidden layers, and an output layer. Information gets fed into the input layer, passed, if it triggers the activation function, to the successive layers until it reaches the output layer. A neural network with one to three layers is called a *basic neural network*. A neural network with more layers can be considered a *deep neural network* (see Figure 4), which brings us to the last level, Deep Learning. [IBM20b]

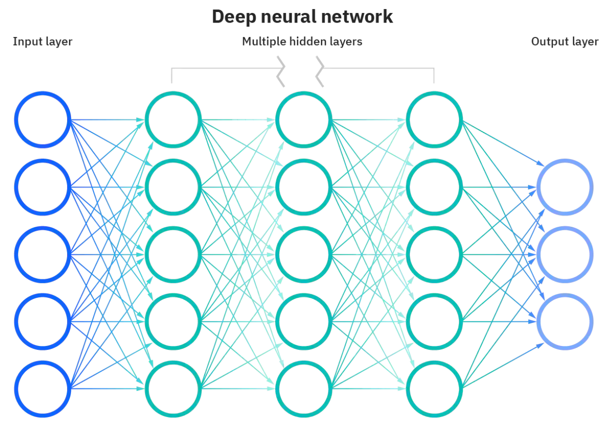


Figure 4. Visualization of Deep Neural Network [IBM20b]

2.1.5 Deep Learning (DL).

Deep Learning (DL), sometimes also referred to as *Deep Neural Networks* (DNN), is only a particular form of an NN. So everything mentioned above also holds here. The difference between DL and ML is that DL automates a lot of preprocessing work and therefore eliminates the need for some of the human input. The more hidden layers a DNN has, the more information it can learn. However, having more layers demands more computation power and space and represents the limiting factor.

2.2 The Process for Building a Deep Learning Model

The process of training and later using a deep learning model always starts with not using one. Always ask yourself first what problem you are facing. Only after that, ask whether ML would be suitable to solve this problem [Goob]. A relatively simple heuristic performs better than a sophisticated ML/DL model in many cases. And even if it does not, it provides a baseline against which your model can be compared. “Never launch a fancy ML model that can’t beat a heuristic. [...] Non-ML solutions can sometimes be simpler to maintain than ML solutions. [Gooa]”

If you have decided on using an ML/DL model, you, first of all, need to have a lot of data. The cleaner the data, meaning no noise and useless data entries, the easier it will be to train the model. Further, the more data you have, the better the model will become. Most of the time required to build a model often gets used in preparing, cleaning, and preprocessing the data.

2.2.1 Preprocessing.

Before we can train a DL model, we first need to put the data into a form, the model can learn. This step is called preprocessing. Preprocessing also includes altering the data if possible, so the learning process has an easier task handling the data. There are many different variants of preprocessing. We will only cover three variants that often get used when working with programming languages.

Tokenization. Tokenization is the process of breaking up a text into a list of tokens. This process is also often done as a first step when parsing a program. Tokens can be words, characters, or any other sequence that can be somehow separated. The tokenization process can also eliminate certain parts that are not needed, for example, punctuations or spaces [MRS08]. `if (a == b)` could be tokenized into `if`, `(`, `a`, `==`, `b`, `)`.

One Hot Encoding. One-hot encoding is a technique that converts a categorical variable into a binary variable. A categorical variable is a variable that can take on only a limited number of values. It does that by creating a new variable for each possible value of the original variable. For example, if we want to encode a mail carrier’s shift into a single binary value, we can do that as follows: We create a new variable for each weekday, the weekday being the category. If mail carrier Alice works only on Monday, we set the variable’s value for Monday to 1 and for all other days to 0. For mail carrier Bob working on Tuesday, Wednesday, and Thursday, we set the value for each of those days to 1 and the other to 0. This process can be displayed in a matrix, like the following (Table 1). As a result Alice gets the shift encoding “`10000`” and Bob “`01110`”.

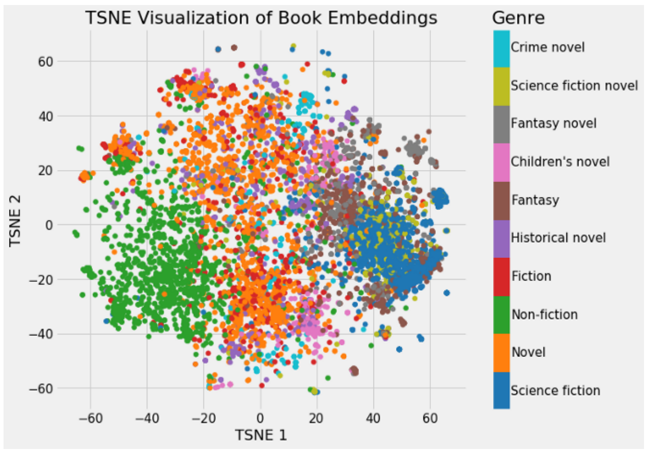
Table 1. One Hot Encoding for Shifts of Mail Carriers

	Mon	**Tue**	**Wed**	**Thu**	**Fr**
Alice	1	0	0	0	0
Bob	0	1	1	1	0

This encoding is a straightforward, understandable form but gets unmanageable for cases with many unique categories. Furthermore, categories are all distributed uniformly, meaning similar categories do not get put closer together than categories that have nothing to do with each other. [Koe18]

(Neural) Embedding. An embedding is a technique that converts a categorical variable into a vector of real numbers. Embedding can drastically reduce the number of categories and can be used to represent a categorical variable more compactly. *Neural* embeddings are vector representations that already involve a neural network for learning how to build the vector. Those embeddings can act as an input for machine learning models and provide the possibility to search for similar embeddings.

When comparing embeddings with an encoding, like the previously mentioned one-hot encoding, the embeddings are usually more compact. Instead of needing to represent each category with a binary value, the embeddings can represent the categories in a very compacted way.



TSNE (t-distributed stochastic neighbor embedding) is a visualization technique, popular for visualizing high-dimensional data.

Figure 5. Visualization of Embeddings of 37,000 Books on Wikipedia [Koe]

As an example, we move from the simple mail carrier example to a more complex one. Assuming we want to give book recommendations to someone based on their genre interests. We first have to represent the books in the library (in the following example, all books on Wikipedia) in a vector space. We can do that by creating a new category for each book and setting the value for each book with a specific category to 1. This approach is the one-hot encoding. For the 37,000 books on Wikipedia, this approach would result in a 37,000-dimensional vector. Doing the same with a neural embedding results in a vector of only 50 numbers. Creating the embedding seems to be a bit of a black box, as the neural network takes the books' information, crushes its data, and outputs a vector of numbers. The learning is a supervised learning process, and those resulting numbers are the parameters used in the neural network. The benefit of using this technique is that using embeddings increases the efficiency of the later learning process and reduces the storage capacity needed to persist those vectors. Moreover, finding similar books is much easier, as the embedding includes the closeness in its representation (Visualized in Figure 5). Embedding is not quite as simple as the one-hot encoding, but it is still not overly complicated and can be well interpreted. [Koe18]

2.2.2 Training and Validation Process.

When having the preprocessed data, the training process can start. The training process consists of two steps: A) Training the model and B) Validating the model.

In a *unsupervised learning* setting, the machine learning algorithm uses unlabeled data and analyses and clusters the data on its own. The verification process needs to be performed by a human, most likely a data analyst.

In a *supervised learning* setting, the model is trained by feeding in labeled input data and then changing the model's weights until the expected output data is returned. This process of altering the weights based on the result is called backpropagation. Many different models use different kinds of backpropagation or other mechanisms to train. We will cover some of the most used ones in subsection 2.3. A training- and a verification set are used because training for too long on the same data will result in overfitting. Overfitting means that the model will learn the training data patterns "by heart" and cannot generalize well to new data. Feeding the training data will result in a perfect result, but the result may be awful for unseen data. An example of such a case can be seen in Figure 6. Instead of generalizing the data (the straight

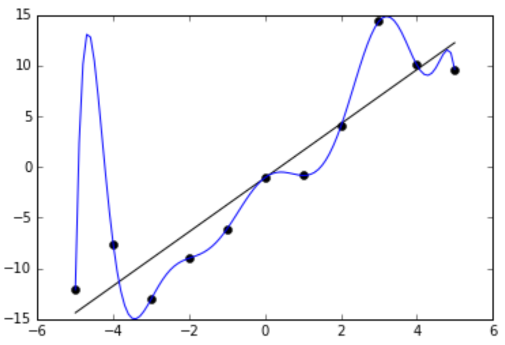


Figure 6. Example of Overfitting [Ghi]

graph), the model creates a graph (the curvy graph) that matches all data points exactly. For this reason, a validation set consisting of only unseen data is used to check the quality of the model. In the example's case, this would be a set of points that behave the same but are not at the same position. Therefore, the peak at position -4.5 would deliver a terrible result and indicates that the model cannot generalize well.

2.3 What different models are used?

As already mentioned, many different models can be used for machine learning.

2.3.1 Feed-Forward Neural Networks (FNNs).

Feed-Forward Neural Networks (FNNs) are the simplest versions of neural networks. They only work in one direction. The output of a node from one layer can only be passed to the nodes in the next layer. Therefore, the graph of connections is a directed acyclic graph (DAG).

2.3.2 Recurrent Neural Networks (RNNs).

Compared to the FNN, a *Recurrent Neural Network* (RNN) is a more complex neural network version. Instead of passing the node's output only to the next layer, in an RNN, the output can also be passed back to the same layer. This "back-passing" is called *recurrent* and gives the model a "memory". The output of a node heavily relies on the elements earlier in the sequence. Furthermore, do share all layers their weights across nodes. In contrast, in a feed-forward network, each weight is independent in each layer. The memory's effect is that an RNN can handle sequences or time-series data like used for processing natural language and speech recognition. For that reason, RNNs are currently the most used neural network models. [IBM20c, Phi18b]

Variants. RNNs are just the base concept. Many variants build upon that idea.

The *Bidirectional Recurrent Neural Network* (BRNN) is an RNN that uses not only previous inputs for prediction but also uses future data to enhance the accuracy of predictions. For example, in neural language processing, knowing the end of the sentence might help to understand better what a word, in the beginning, is referring to.

The *Long Short Term Memory* (LSTM) is a variant of an RNN where the memory gets slightly altered. In the normal RNN, the memory is relatively short-termed. The information further away in the sequence has less impact and gets slowly forgotten. LSTM tries to tackle this issue by having so-called "gates" that decide what remains in the memory and gets discarded. [Ola15]

Very similar to LSTM, the *Gated Recurrent Unit* (GRU) aims to improve the memory of the RNN. It is built in a slightly different way but has a similar outcome. We will not go into details here but [Phi18a] has a good explanation of the two variants.

2.3.3 Convolutional Neural Networks (CNNs).

Convolutional Neural Networks (CNNs) are variants of a neural network mainly used for computer vision and image classification. They consist of different layers that are connected. The convolutional layer converts data into numerical values that the network can then interpret. The pooling layer reduces the input space by applying algebraic aggregation functions. And the fully-connected layer then classifies the features extracted by the other layers. The first two layers can be applied multiple times after each other, but the last layer is only used once in the end. Together, those layers can very accurately classify, for example, objects in images. [IBM20a]

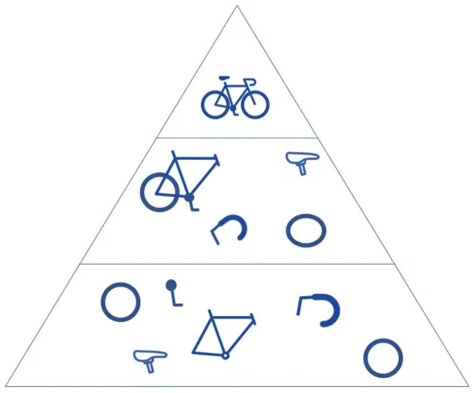


Figure 7. Illustration of classification process of a CNN employing pattern matching, reduction, and classification [IBM20a]. Process goes from bottom to top.

Figure 7 illustrates that for recognizing an item in a picture. In this example, a bicycle. It first converts the picture into a numerical representation. Then it reduces the input space by applying a pattern matching algorithm, which results in different parts, like a saddle or a wheel. It further reduces the input space by applying a reduction algorithm, which results in the end in a complete bicycle. The network does not know what it has put together at this stage. It is still just a numerical representation of an item. Finally, it classifies the feature by applying a classification algorithm and yields the result of the item in the picture as being a bicycle.

2.3.4 Encoder-Decoder Models.

The Encode-Decoder model uses two RNNs. One RNN encodes an input sequence, and one decodes the encoded sequence and outputs the result. It is generally used for *sequence-to-sequence* (S2S) models. The main advantage of having two independent RNNs work together is that the input and output sequence can differ in dimension. For example, the length of a sentence in German will most likely be longer than the length of an English sentence. By having the encoder encode the German input into a vector that abstractly contains its meaning, the decoder can then decode this abstract vector into another language without knowing anything about the original sentence.

For example, Google Translate has used Encoder-Decoder models since 2016, and more advanced S2S models are built upon those kinds of models. [Mos21]

3 Deep Learning for Programming Languages (DL4PL)

This section will move from deep learning in general to deep learning in the area of programming languages. It discusses what DL4PL can be used for and what not, and how it is used. It is all rounded up with an extensive example of one application of DL4PL, namely for the DeepBugs project, and a non-DL approach as a comparison.

3.1 What can I do with it?

In *programming languages* (PL), DL is mainly used for program synthesis, code comprehension, and code generation. Program synthesis means constructing a program that satisfies a given specification. The specification can be stated in many different forms, such as defining pseudo-code, writing prose text, or a formal specification. Code comprehension is the study of understanding the ways engineers maintain existing code. Code generation covers the general idea of generating code based on prior acquired knowledge, for example, from other codebases. [WCP+20]

In subsection 1.2 some examples regarding what is possible with DL were shown. In subsection 3.4 an extended example will be discussed.

3.2 What and how does it "learn"?

Deep Learning contains the word "learning", but what is learned? To leverage the power of DL, a lot of o data is needed. Since more and more projects are being developed and increasingly more open-sourced, tapping into those resources is the most straightforward way, and the most commonly used at the moment, for gathering data. GitHub, GitLab, BitBucket, to name the three most significant code collaboration and version control tools, all provide APIs over which one can farm the repositories based on several criteria. Mining code can be done in different ways. The source code can be used at the binary level, code-snipped level, method level, class level, or project level. Besides code, additional artifacts can be mined and utilized. For example, we can leverage code review comments and suggestions to understand code better or use bug reports to identify possible errors. Nevertheless, in general, the main source one tries to learn from is code.

Some methods utilize the code as a single version, visited at a certain point in time, to learn patterns of correct code. Other methods use the version control system to gather information regarding code changes to find possible errors that got fixed. Examples for both approaches are discussed later in section subsection 3.4 and subsection 3.5.

Also, how code is interpreted varies from application to application. Code can be parsed, put into an *abstract syntax tree* (AST), and interpreted by encoding it into some learnable sequence (for example, One Hot Encoding, Neural Embedding, or Execution Trace Vectorization). In subsection 3.4 this approach will be discussed. Another option is to interpret the code as it is, as a sequence of characters. This option would be similar to processing natural language. The GitHub Copilot, discussed in subsubsection 1.2.1, is an example of a code interpreter that uses this approach. [WCP+20]

3.3 What are its limitations?

DL is often thought to be kind of a silver bullet that can solve all problems. DL tries to mimic the human mind but is miles away from coming even close. The strength of DL is to find patterns in a blob of un- or semi-structured data and make some sense out of it. However, writing code is much more than just applying common patterns. Going back to the initial examples shown with the GitHub Copilot, code can be generated based on some description of the problem that is wanted to be solved. But this only works on a small scope. Defining a general problem, like "We need a system for managing our warehouse" will never be fully automated. To specify all the requirements and stitch together the parts, the need for humans will always persist. DL can assist in this but will most certainly never be able to do that independently.

Coming back to the current state of DL, many limitations originate from other sources than mere complexity. Often, there is a general lack of "proven-to-be-clean" code, meaning code that has no code smells, bugs, or "bad code" in it. Providing such a collection is a tremendous task and is also very likely not to be open-sourced. Further, the problems we want to solve are often not well-defined or not well-understood, such that a DL model would struggle to work effectively. Furthermore, even if they would be well-defined, the currently available architectures are not suitable for applying them to the available data [WCP+20]. Much more needs to be open-sourced and well documented to bring DL to the next level, so one would not need to reinvent the wheel each time something new wants to be tried. The lack of open-source material is a widespread problem in research or software engineering in general but is much more severe for machine/deep learning.

3.4 DeepBugs

We now look at a practical example of using deep learning to find bugs in a program. For this, we will use the DeepBugs[6] project by Michael Pradel and Koushik Sen [PSD17]. The general idea of the project is to use correct code, automatically generate a falsy/buggy version of it, and train the system to detect such bugs. Then, newly, previously unseen code gets fed into the bug detector, which results in a prediction of whether the code is buggy or not.

An example of the process is shown in Figure 8 with the code construct `setSize(width, height)`. The buggy version that gets generated is `setSize(height, width)`, where the order of the parameters is swapped. Those two versions get embedded into a vector representation. The vector representations then get fed into the deep learning network for training it to detect the buggy versions. For new code, in the example `setDim(y_dim , x_dim)`, the vector representation gets generated as well and classified by the bug detector. The result is a prediction that

[6]https://github.com/michaelpradel/DeepBugs

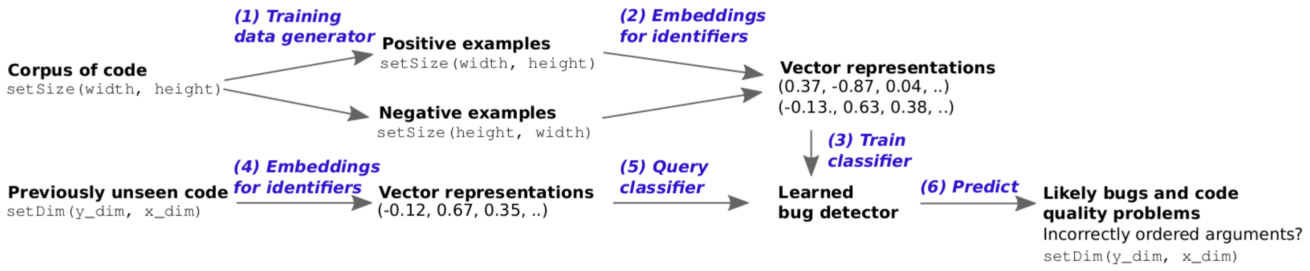


Figure 8. Overview of DeepBugs' aproach [PSD17]

the arguments are incorrectly ordered and therefore is buggy code.

As previously mentioned, training on the code first requires the code to be converted to a vector representation. To do that, the code gets parsed into an AST. Based on this, a context gets built for each statement. A context contains information regarding the parent, grandparent, siblings, uncles, cousins, nephews, and the statement's position in the code. These attributes each get encoded with the one-hot encoding and then concatenated, resulting in a single vector.

```
// Math.min(max, 1)
{
  'base': 'ID:Math',
  'callee': 'ID:min',
  'calleeLocation': 'location_identifier',
  'arguments': ['ID:max', 'LIT:1'],
  'argumentLocations': [location_identifier, ...],
  'argumentTypes': ['unknown', 'number'],
  'parameters': ['', ''],
  'src': 'fileanme : line_number',
  'filename': 'fileanme'
}
```

Listing 4. Sample data structure

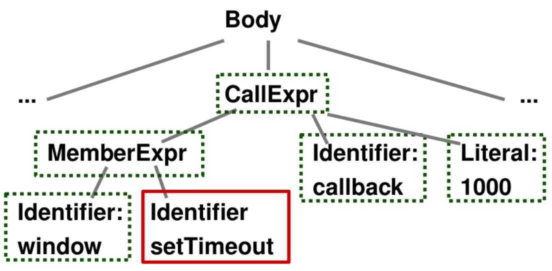


Figure 9. Simplified AST of the function call `window.setTimeout(callback, 1000);` [PSD17]

For example (AST shown in Figure 9), `setTimeout` is the identifier of the *member expression* of the *call expression*. The sibling is `window`, and the uncles are `callback` and `1000`. That information gets encoded and together builds the AST context vector.

When the context vector is built, the embeddings for the identifiers are created and forwarded to the bug detector. Examples for such embeddings are shown in Figure 8.

In the example, the training data generator built code versions with the arguments swapped. Swapped arguments are only one type of bug. The paper also covers bug detectors for wrong assignments, wrong binary operators, and wrong binary operands. The repository additionally contains bug detectors for incorrect assignments and missing arguments.

3.4.1 Practical Example. We now look at a code example provided by DeepBugs. Given a data set of parsed method invocations (Listing 4) and a map or function that maps each token to a vector, we can produce a correct and a buggy version for each method (Listing 5). Note how the AST nodes are identified. Identifiers are prefixed with "ID:", literals with "LIT:". This is important for later. After generating all wrong

```
# Inputs given to the model: Each element is
# the vector representation of a function call.
xs = []

# Outputs expected from the model: For each
# call, predict the probability that it's buggy.
ys = []

for call in calls:
    if (call["callee"] in token2vec and
            call["arguments"][0] in token2vec and
            call["arguments"][1] in token2vec):
        callee_vec = token2vec[call["callee"]]
        arg1_vec = token2vec[call["arguments"][0]]
        arg2_vec = token2vec[call["arguments"][1]]

        # Positive, i.e., correct example
        x_correct = callee_vec + arg1_vec + arg2_vec
        # Negative, i.e., buggy example
        x_buggy = callee_vec + arg2_vec + arg1_vec

        xs.append(x_correct)
        ys.append(0)  # Probability that buggy is 0
        xs.append(x_buggy)
        ys.append(1)  # Probability that buggy is 1
```

Listing 5. Traing Data Generation

versions, we can divide our two sets into training and validation sets. We put the sets into a nine-to-one ratio, the larger set being used for training. In total, this results in

```
# Split into training and validation data
nb_training = int(0.9*len(xs))
xs_training = np.array(xs[:nb_training])
ys_training = np.array(ys[:nb_training])
xs_validation = np.array(xs[nb_training:])
ys_validation = np.array(ys[nb_training:])
```

Listing 6. Creation of Training and Validation Sets

training sets with 21592 samples and validation sets with 2400 samples. With all the preparation work done, we can now train the model. The training in the example is done by Keras. Keras[7] is a human-friendly API for creating and training machine learning models. It is built on top of TensorFlow[8], an open-source machine learning framework by Google. We will not cover the building of such a model, as it requires some machine learning knowledge and does not yield any significant additional information. We use pseudo python code as a way to express it. After just five training

```
model = Model()
model.train(xs_training, ys_training)

# Epoch 1/5 - loss: 0.5015 - accuracy: 0.7396
# Epoch 2/5 - loss: 0.3581 - accuracy: 0.8323
# Epoch 3/5 - loss: 0.2963 - accuracy: 0.8662
# Epoch 4/5 - loss: 0.2554 - accuracy: 0.8875
# Epoch 5/5 - loss: 0.2236 - accuracy: 0.9006
```

Listing 7. Pseudo-Train the Model

cycles, we can see that the model has an accuracy of 90% on the training set. We need to verify the actual performance with the verification set to see the actual performance. The

```
model.evaluate(xs_validation, ys_validation)

# loss: 0.3072 - accuracy: 0.8562
```

Listing 8. Verify the Model

accuracy on previously unseen data is slightly lower than before, but with 85% still an acceptable result. The model trained and verified is now ready to be used in a "real-world example". For that, we craft a piece of code. First, a correct example: The JavaScript method `setTimeout(func, time)` is used to execute a function after a certain time. If we feed this piece of code into our model, it should be classified as correct. If we feed the incorrect version into our model, the result should be classified as buggy. We can see that the model accurately predicts the buggy piece of code with a certainty of 96%. The correct piece of code is classified as being correct with a 71%

```
callee = token2vec["ID:setTimeout"]
arg1 = token2vec["ID:fn"]
arg2 = token2vec["ID:delay"]

x = callee + arg1 + arg2 # correct order
model.predict(x)

# Call is buggy with probability 0.2932
```

Listing 9. Setup Function and Predict Bugginess of Correct Piece of Code

```
x = callee + arg2 + arg1 # wrong order
model.predict(x)

# Call is buggy with probability 0.9697
```

Listing 10. Predict Bugginess of Wrong Piece of Code

certainty. The certainty for the correct code is not as high as for the buggy code but is still high enough not to be counted as a false positive. In the end, the threshold for marking a method call as being buggy or not lies in our hands. The way the model is trained (details omitted in the code above) also plays an important role. It needs some trying-out work to find the best possible combinations of parameters and methods used. With some tweaking on the parameters (epochs and batch sizes - not shown in the paper), we quickly can get a prediction certainty of 98% for the buggy code and 93% for the correct code.

The significant achievement of this approach is that vast amounts of training data can be automatically generated. Furthermore, although only shown for JavaScript, this approach works for all kinds of programming languages. The paper only covered elementary types of bug creators and detectors, but some more advanced versions could be possible in future versions. Combining this method with other methods like, for example, ControlFlag (covered in subsubsection 1.2.4) could yield some significant advances for coding assistants.

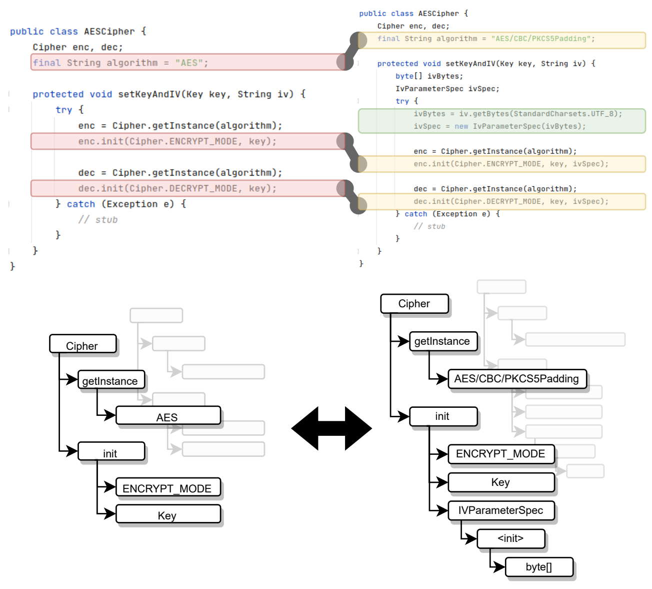
3.5 DiffCode

Another approach for detecting code bugs, code name "DiffCode", is described in the paper "Inferring Crypto API Rules from Code Changes [PZT+18]". This approach does not use deep learning but shows another option to understand and represent code. Adding deep learning to this could be a possibility for future projects.

DiffCode uses version control systems (VCS) commits to detect possible bugs. For that, it builds a DAG derived from the AST for every single instantiated object. The DAG contains the interactions with the object, starting from its initialization. In the example (Figure 10), the object `enc` of type `Cipher` is shown. The changes in the before (left) and after (right) versions are visualized in color.

With this representation, a system is constructed that matches the changes and filters out the irrelevant ones, such

[7] https://keras.io
[8] https://www.tensorflow.org



```java
public class AESCipher {
    Cipher enc, dec;
    final String algorithm = "AES";

    protected void setKeyAndIV(Key key, String iv) {
        try {
            enc = Cipher.getInstance(algorithm);
            enc.init(Cipher.ENCRYPT_MODE, key);

            dec = Cipher.getInstance(algorithm);
            dec.init(Cipher.DECRYPT_MODE, key);
        } catch (Exception e) {
            // stub
        }
    }
}
```

```java
public class AESCipher {
    Cipher enc, dec;
    final String algorithm = "AES/CBC/PKCS5Padding";

    protected void setKeyAndIV(Key key, String iv) {
        byte[] ivBytes;
        IvParameterSpec ivSpec;
        try {
            ivBytes = iv.getBytes(StandardCharsets.UTF_8);
            ivSpec = new IvParameterSpec(ivBytes);

            enc = Cipher.getInstance(algorithm);
            enc.init(Cipher.ENCRYPT_MODE, key, ivSpec);

            dec = Cipher.getInstance(algorithm);
            dec.init(Cipher.DECRYPT_MODE, key, ivSpec);
        } catch (Exception e) {
            // stub
        }
    }
}
```

Figure 10. Conversion of code usages of the field `enc` of type `Cipher` into an abstract DAG [PZT+18].

as refactorings. The system then clusters the changes and outputs a list of changes that are very similar and changed something in the behavior.

The benefit of this approach opposed to the one of DeepBugs (subsection 3.4), is that bad versions are created with the help of real-world code changes. Furthermore, the code representation also contains some information regarding the object's lifecycle. On the other hand, DiffCode's representation is not as contextually detailed as DeepBugs' one and has no compact way of representing it. Using statistical tools to analyze the data, instead of using something like deep learning, results in a system that will never be able to cope with the amount of data required to make a system that runs more or less in real-time and is fully automatic. The statistical approach is way too slow. A more detailed analysis of this paper can be found in [Jen21]. Nevertheless, a combination of both approaches, DeepBugs and DiffCode, could potentially yield some interesting results.

4 Conclusion

This paper addresses the current state of machine learning in general and the current state of deep learning for programming languages. The critical insight is that the field is very quickly evolving and yields new results rather quickly. The technology is still in its early years, and many advancements are going to happen. Nevertheless, some fantastic results, like Google Assistant, Alexa, etc., for deep learning in general, or Github's Copilot and Tabnine explicitly in the context of programming languages, have been achieved already. With more and more code being open-sourced, many more possibilities are going to open.

However, there are still things for what you are always going to need a human. Deep learning is not a silver bullet that solves all our problems. It can assist a human but not replace it. In the long term, this assistance will grow and grow; the question is just by what speed this will happen.

References

[BRTV] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical Deobfuscation of Android Applications.

[BRV] Pavol Bielik, Veselin Raychev, and Martin Vechev. Learning a Static Analyzer from Data.

[BRV16] Pavol Bielik, Veselin Raychev, and Martin Vechev MARTIN-VECHEV. PHOG: Probabilistic Model for Code. 2016.

[CBR+] Victor Chibotaru, Benjamin Bichsel ETH Zurich, Veselin Raychev, Martin Vechev ETH Zurich, and Benjamin Bichsel. Scalable Taint Specification Inference with Big Code.

[CTJ+21] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code. jul 2021.

[DDB+] Elizabeth Dinella, Hanjun Dai, Google Brain, Ziyang Li, Mayur Naik, Le Song, Georgia Tech, and Ke Wang. HOPPITY: LEARNING GRAPH TRANSFORMATIONS TO DETECT AND FIX BUGS IN PROGRAMS.

[ESR+] Jan Eberhardt, Samuel Steffen ETH Zurich, Veselin Raychev, Martin Vechev ETH Zurich, Samuel Steffen, and Martin Vechev. Unsupervised Learning of API Aliasing Specifications.

[Ghi] Ghiles. Overfitted data. https://commons.wikimedia.org/w/index.php?curid=47471056#/media/File:Overfitted_Data.png.

[Git21] GitHub Copilot · Your AI pair programmer. https://copilot.github.com, 2021.

[Gooa] Google. Deciding on ML | Introduction to Machine Learning Problem Framing. https://developers.google.com/machine-learning/problem-framing/framing.

[Goob] Google. Identifying Good Problems for ML. https://developers.google.com/machine-learning/problem-framing/good.

[GSLT+] Justin Gottschlich, Armando Solar-Lezama, Nesime Tatbul, Michael Carbin, Martin Rinard, Regina Barzilay, Saman Amarasinghe, Joshua B Tenenbaum, and Tim Mattson. The Three Pillars of Machine Programming.

[HG21] Niranjan Hasabnis and Justin Gottschlich. ControlFlag: A Self-Supervised Idiosyncratic Pattern Detection System for Software Control Structures; ControlFlag: A Self-Supervised Idiosyncratic Pattern Detection System for Software Control Structures. 2021.

[HZT+18] Jingxuan He ETH Zurich, Eth Zurich, Petar Tsankov ETH Zurich, Veselin Raychev, and Martin Vechev ETH Zurich. Debin:

Predicting Debug Information in Stripped Binaries. *CCS*, 14, 2018.

[IBM20a] IBM. What are Convolutional Neural Networks? https://www.ibm.com/cloud/learn/convolutional-neural-networks, 2020.

[IBM20b] IBM. What are Neural Networks? https://www.ibm.com/cloud/learn/neural-networks, 2020.

[IBM20c] IBM. What are Recurrent Neural Networks? https://www.ibm.com/cloud/learn/recurrent-neural-networks, 2020.

[IBM20d] IBM. What is Artificial Intelligence (AI)? https://www.ibm.com/cloud/learn/what-is-artificial-intelligence, 2020.

[IBM20e] IBM. What is Machine Learning? https://www.ibm.com/cloud/learn/machine-learning, 2020.

[Jen21] Raphael Jenni. Reproducing: Inferring Crypto API Rules from Code Changes. 2021.

[Koe] Will Koehrsen. Book Recommendation System.ipynb. https://github.com/WillKoehrsen/wikipedia-data-science/blob/master/notebooks/Book%20Recommendation%20System.ipynb.

[Koe18] Will Koehrsen. Neural Network Embeddings Explained. https://towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526, 2018.

[Mos21] Kriz Moses. Encoder-Decoder Seq2Seq Models, Clearly Explained!! https://medium.com/analytics-vidhya/encoder-decoder-seq2seq-models-clearly-explained-c34186fbf49b, 2021.

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. Introduction to Information Retrieval. *Cambridge University Press*, jan 2008.

[NRN10] Nils J. Nilsson, Stuart J. Russell, and Peter Norvig. *Artificial intelligence: A modern approach (Third Edition)*, volume 82. 2010.

[Ola15] Christopher Olah. Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015.

[Phi18a] Michael Phi. Illustrated Guide to LSTM's and GRU's: A step by step explanation. https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21, 2018.

[Phi18b] Michael Phi. Illustrated Guide to Recurrent Neural Networks. https://towardsdatascience.com/illustrated-guide-to-recurrent-neural-networks-79e5eb8049c9, 2018.

[PSD17] Michael Pradel, Koushik Sen, and T U Darmstadt. Deep learning to find bugs, 2017.

[PZT+18] Rumen Paletov, Eth Zurich, Petar Tsankov ETH Zurich, Veselin Raychev, Martin Vechev ETH Zurich, Petar Tsankov, and Martin Vechev. Inferring Crypto API Rules from Code Changes. 2018.

[RBV] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic Model for Code with Decision Trees.

[RBVK] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning Programs from Noisy Data.

[RVK] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting Program Properties from "Big Code".

[RZVY] Veselin Raychev, Eth Zürich, Martin Vechev, and Eran Yahav. Code Completion with Statistical Language Models.

[Tur50] A M Turing. COMPUTING MACHINERY AND INTELLIGENCE. *Computing Machinery and Intelligence.*, 49:433–460, 1950.

[WCP+20] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. A systematic literature review on the use of deep learning in software engineering research. 9 2020.