information

Article

Exploring Neural Dynamics in Source Code Processing Domain

Martina Saletta [1,2,*] and Claudio Ferretti [2,*]

[1] Department of Engineering and Architecture, University of Trieste, 34127 Trieste, Italy
[2] Department of Informatics, Systems and Communication, University of Milano-Bicocca, 20126 Milan, Italy
* Correspondence: martina.saletta@unimib.it (M.S.); claudio.ferretti@unimib.it (C.F.)

Abstract: Deep neural networks have proven to be able to learn rich internal representations, including for features that can also be used for different purposes than those the networks are originally developed for. In this paper, we are interested in exploring such ability and, to this aim, we propose a novel approach for investigating the internal behavior of networks trained for source code processing tasks. Using a simple autoencoder trained in the reconstruction of vectors representing programs (i.e., program embeddings), we first analyze the performance of the internal neurons in classifying programs according to different labeling policies inspired by real programming issues, showing that some neurons can actually detect different program properties. We then study the dynamics of the network from an information-theoretic standpoint, namely by considering the neurons as signaling systems and by computing the corresponding entropy. Further, we define a way to distinguish neurons according to their behavior, to consider them as formally associated with different abstract *concepts*, and through the application of nonparametric statistical tests to pairs of neurons, we look for neurons with unique (or almost unique) associated concepts, showing that the entropy value of a neuron is related to the rareness of its concept. Finally, we discuss how the proposed approaches for ranking the neurons can be generalized to different domains and applied to more sophisticated and specialized networks so as to help the research in the growing field of explainable artificial intelligence.

Keywords: explainable AI; artificial neural networks; knowledge representation; source code analysis

Citation: Saletta, M.; Ferretti, C. Exploring Neural Dynamics in Source Code Processing Domain. *Information* **2023**, *14*, 251. https://doi.org/10.3390/info14040251

Academic Editors: Gabriele Gianini and Pierre-Edouard Portier

Received: 28 February 2023
Revised: 5 April 2023
Accepted: 17 April 2023
Published: 21 April 2023

1. Introduction

Research results on the use of deep learning systems show how the internal representation developed by a system during its training is of value, even for tasks different than those it was trained for. Techniques which exploit this fact are, for example, the methods for pretraining [1] or semisupervised learning, transfer learning [2], or internal interpretability [3]. The research that aims at characterizing such internal representation is active for the domains of image processing [4] and of natural language processing (NLP) [5], while fewer results are, however, available for the domain of source code processing.

In the specific field of source code static analysis, many neural systems have been presented (we refer the reader to [6,7] for surveys), especially for tasks related to the software engineering domain, such as bug detection or code completion, but also for more *semantic* purposes, e.g., automatic tagging or classification according to the functionality of the code snippet being examined.

Given the importance of understanding these neural networks for source code processing, which are becoming more and more common, we focus on examining the internal neurons of some given learning system in order to look for those which exhibit interesting behaviors in terms of classification performance or activation patterns.

The main goal is to define a general approach for discovering and exploiting *all* the knowledge learned by a given model. For instance, one can assume to have a neural model (trained on a main task) embedded in a code editor or in a software repository that also

allows tapping into further parts of the internal representation developed while being trained. To this aim, this work provides the following contributions:

- A procedure for ranking neurons according to their ability in solving arbitrary binary classification tasks. With the experiments in this direction, we show how some neurons are able to autonomously build internal representations for different program properties.
- An information-theoretic approach for identifying neurons which exhibit interesting behaviours, with the aim to identify the most *informative* neurons in the network and to discriminate among neurons showing different activation patterns.
- A statistical measure for comparing the arbitrary binary tasks defined by single neurons (namely by simply establishing a threshold and by splitting the dataset according to the activation induced by each program instance) so as to identify neurons which recognize unique (or uncommon) *concepts*.

Related Work

Chasing the success achieved in a wide range of domains, such as those of images and NLP, systems based on machine learning (ML) are becoming popular also for dealing with source code (see, e.g., [6,7]) and, more in general, with software artifacts [8]. To this end, the recent literature features several examples of ML models trained in solving tasks related to the source code processing domain, including code completion [9], code summarization [10], and classification [11]. The choice of the input representation for feeding such models is, in general, a crucial aspect in this scope, since it is not always effective to use the pure textual representation as in classical NLP models. To this end, several works are focused in the design of program encodings or representations that are able to capture different properties so as to properly convey the seized information and to help in the solution of a specific task. An interesting approach in this direction is, for instance, the work described in [12], where a graph-based representation for programs derived from the abstract syntax tree (AST) is used for solving classical software engineering tasks such as predicting the name of a variable or if a variable has been misused. A similar idea is tackled in [13], where the authors propose a vector representation for programs, namely a source code *embedding*, for solving similar tasks, e.g., for predicting the name of a method.

Recently, besides the use of networks that need as input specific program representations, also the models commonly known as transformers [14], widely used for NLP applications, are becoming popular in the source code processing domain [15,16]. One of the advantages in using these kinds of networks is their flexibility: they can be trained once on generic and big corpora of data and then fine-tuned for solving several specific tasks.

The study of the internal behavior of neural models is becoming popular, and many research results in this direction show how the analysis of the activation patterns that a neuron exhibits is of interest, both in terms of the internal representations it develops and when considered only for its inherent dynamics. A recent work [17], for instance, proposes a study of these dynamics from an information-theoretic perspective, while in the area of image analysis an interesting approach for studying the internal representations developed by the neurons is proposed in [4], where each neuron of an unsupervised trained network was evaluated with respect to a given image classification task, with insightful results. More recently, results have been obtained for evaluating single neurons for sentiment analysis tasks [18] or in networks trained to model natural languages [5].

More recently, groups of neurons have been devised to explain the decision processes of neural networks. For instance, *concept activation vectors* (CAVs) [19] have proven to be effective to model human-understandable concepts in the internal states of a network and have been effectively applied also in many different domains, such as that of chess [20] and of source code analysis [21,22].

2. Approach Description

In this section, we provide a high-level description of the approach we devised for analyzing the internal behavior of neural networks trained on source code, with the aim to express and reason about its dynamics in a measurable way.

We first trained a simple autoencoder (i.e., an artificial neural network trained in the reconstruction of the input, see [23] for a complete reference) on two different source code embeddings, and then we performed three categories of experiments:

1. Binary classification experiments, for ranking neurons considering their ability in solving specific tasks.
2. Analysis of the relevance of the neurons for the network itself, regardless of a given task.
3. Pairwise comparison of the neurons' dynamics, through the adoption of statistical techniques.

For all these experimental approaches, we first map the source code to feature vectors, namely via a neural embedding, and then we study the internal behavior of a neural network trained on such vectors by analyzing the activation values of the neurons on different program instances. In the first two cases, we are interested in assigning a score to each neuron, i.e., in ranking the neurons according to different criteria, while the aim of the third point is to possibly define a partition for the set of neurons in order to discriminate among different behaviors and to define an association among neurons sharing similar patterns in terms of statistical distribution of the activation values.

In the classification experiments, the score we assign to each neuron is represented by the accuracy obtained when used as a classifier for given binary problems, as will be detailed in Section 4. The basic concept is to consider, for each neuron, different activation thresholds and then to measure, for each threshold, the accuracy of the neuron in classifying program instances from a balanced labeled sample when predicting a program to be in class 0 if the activation yielded by that program is less than the threshold and to be in class 1 otherwise. The scoring mark for a neuron is the accuracy obtained while considering the threshold that leads to the highest accuracy.

In the second class of experiments, the score of each neuron is instead computed independently from any task. Similar studies, i.e., the definition of a scoring measure for evaluating the importance of single neurons in a network, have been already investigated in the literature [5,24]. While in the referred works the core idea is to use the correlation between activation values of neurons in distinct but isomorphic models (i.e., retraining on different training sets of the same model) for finding neurons that possibly capture properties that emerge in different models, in this paper we propose a ranking based on the concept of *entropy* used in information theory. The reason is that, while by means of the correlation analysis one is able to state which neurons are the most important with respect to the task the network is trained on, with the entropy-based measure we are proposing, the neurons are graded with regard to the importance they have according to their behavior in the network: since, by definition, the entropy in information theory is the average level of information emitted by a signaling system [25], computing the entropy of single neurons is equivalent to measuring how much each neuron is informative.

Finally, in the third class of experiments, we consider all the possible pairs of neurons and, for each pair, we perform a nonparametric statistical test for assessing which neurons share some activation patterns. This will also allow us, as it will be detailed and formalized in Section 5, to study if there exists a partition of the neurons based on their entropy levels such that (intuitively) activation patterns of the neurons belonging to the same part can be distinguished from those of neurons belonging to the other parts. Since, as it will be detailed in Section 5, we can associate with each neuron an arbitrary *concept*, being able to distinguish among different neurons' behaviors could be somehow comparable to detect different concepts that are autonomously learned by the network, regardless to the original task it is trained on.

3. Experimental Settings

This section describes the experimental setting we adopted in order to study both the ability of the hidden neurons of a generic neural network in building a high-level representation for some specific source code-related features, similar to what the authors of [4,18] did for images and natural language, and to characterize (and distinguish) the behavior of the neurons in terms of their entropy and their activation patterns.

3.1. Network Architecture

We chose to work in the context of a simple neural model, namely an autoencoder. Without any hyper-parameter optimization nor any in-depth study on the network design, we implemented a simple dense autoencoder having two hidden layers in the encoder, two symmetrical hidden layers in the decoder, and one code layer in the middle, as shown in Figure 1. We used the *ADADELTA* optimizer [26] and the mean squared error as a loss function. As the activation function for the hidden layers, we applied the Rectified Linear Unit (ReLU), defined for all $x \in \mathbb{R}$ as $\max(0, x)$. The model is implemented using the APIs provided by the Keras library [27].

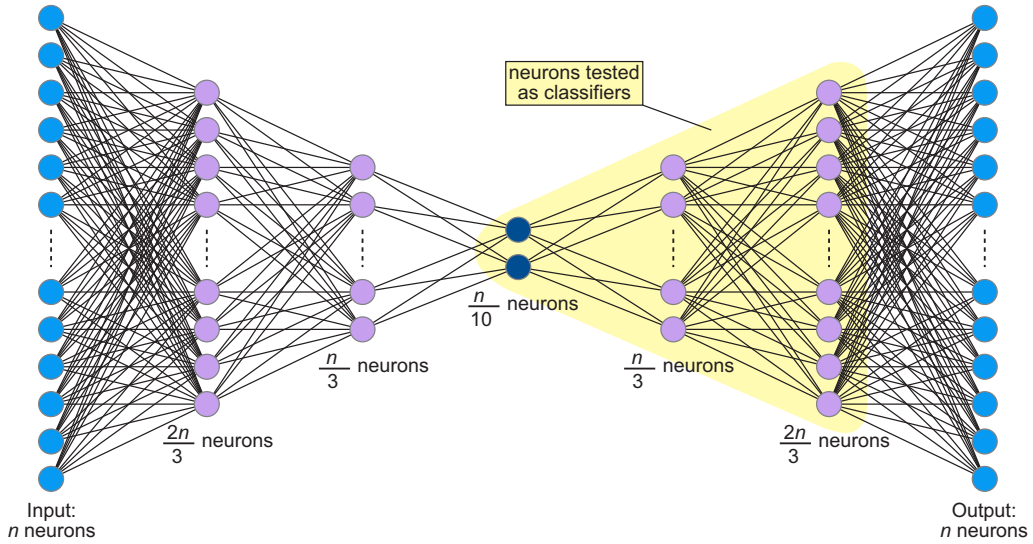

Figure 1. Network architecture. The sizes of the layers depend to the program vector dimensions. In all our experiments, we considered $n = 300$. Layers highlighted in yellow are those tested in the experiments.

3.2. Training

For the training phase, we used the dataset also adopted in [10], which is a collection of popular GitHub (https://github.com, accessed on 4 April 2023) Java projects that contains over 400,000 methods, while for all the experiments, we employed a subset of the Java-med dataset described in [28].

We performed two independent trainings of the autoencoder, using two different source code embedding algorithms for preprocessing all the methods in the dataset and computing the corresponding program vectors:

1. A 300-dimensional embedding obtained by simply applying the doc2vec model [29] to the methods in the dataset using the gensim framework [30]. To avoid inconsistencies related to formatting choices, such as the presence or absence of spaces between operands and operators, keywords and parentheses, we applied the doc2vec model to a pretty printed version of the methods obtained by using the Javaparser library [31].
2. The source code embedding proposed in [32], which we will refer to as ast2vec, consists of the application of the word2vec model [33] to words and sentences derived from the abstract syntax tree.

Notice that our approach can be applied to any neural network, from the most simple, to more sophisticated ones. In general, differently from the field of images, where the input of the network is exactly the object being studied, more often than with images, models for source code processing require a specific program representation as input, e.g., a numerical vector representation (embedding) or a stream of tokens, and each representation inherently preserves or emphasizes some program features to the detriment of others. Since in this work we are mostly interested in proposing and evaluating the approach rather than in studying the best input representation, the two sets of program vectors considered for our experiments have different underlying construction ideas: doc2vec embedding is built by applying a classical NLP technique to the pure source code, so it is not supposed to be particularly viable in this context, while ast2vec is developed by considering both structural (neighborhood of nodes in the AST) and lexical (identifiers chosen by the programmer) features, and thus it is assumed to be particularly suitable and flexible for general program comprehension applications.

We trained each autoencoder for 50 epochs using the hyper-parameters described in Section 3.1; due to the input vector dimensions, the layers in the encoder have, respectively, 300, 200, and 100 neurons, those in the decoder symmetrically have 100, 200, and 300 neurons, and the code layer has 30 neurons.

4. Experiments on Classification Tasks

For assessing the ability of the internal neurons in our networks to build internal representations for different program properties, we first tested each neuron in being used as a classifier for distinct binary classification problems, following the same approach of previous works [4,18].

4.1. Problems Definition

When dealing with images and product reviews as in the referred papers, the properties according to which to classify the input objects can be easily defined could be, for instance, the presence of particular patterns (e.g., cats or faces [4]) or positive and negative review sentiments [18], as in classical image recognition and sentiment analysis tasks. In the program comprehension context, however, such kinds of properties do not directly arise from the source code, or at least they are not immediately evident for a human being reading it. Therefore, we first defined different labeling policies for classification so as to capture properties having different natures:

- The first one, designed using the control flow graph (CFG) [34], addresses the syntactical structure of a method in terms of its **structural complexity**.
- The second one relies on the method's **identifiers** chosen by the programmers in order to target a task related to the functionality of a method.
- The third one is related to its **I/O relationship**, that is the relation between the input parameters and the returned object of a method.
- The last one is a **random** labeling strategy used as a baseline.

In the following, we formally describe these labeling policies with full details:

Structural Labeling Policy

We consider the *cyclomatic complexity* [35] of a program, defined starting from its CFG $\mathcal{G}$ having n vertices, e edges, and p connected components as:

$$V(\mathcal{G}) = e - n + p \tag{1}$$

Dealing with Java methods, such metric can be easily calculated by counting 1 point for the beginning of the method, 1 point for each conditional construct and for each case or default block in a switch-case statement, 1 point for each iterative structure, and 1 point for each Boolean condition. Starting from this software metric, we define the problem as follows, for a given parameter c:

Problem 1. *Let M be a set of Java methods, let $c \in \mathbb{N}$, and let $h_c : M \to \{0,1\}$ be a binary classification rule for the methods. We define, for each $m \in M$:*

$$h_c(m) = \begin{cases} 0 & \text{if } V(\mathcal{G}_m) < c \\ 1 & \text{otherwise} \end{cases} \tag{2}$$

Identifiers-Based Labeling Policy

For the definition of the semantic labeling strategy, we adopted the same assumption that Allamanis et al. [10] and Alon et al. [36] made in their works, namely that the name a programmer gives to a method can be somehow considered as a summary of the method's operations, meaning that the name of a method shall provide some semantic information on the method itself. Starting from this premise, we define this semantic labeling considering the presence or absence of specific patterns, from a given set T, in the method name:

Problem 2. *Let M be a set of methods, let $N = \{lab_m : m \in M\}$ be the set of the names of the methods in M, and let T be a set of patterns. We write $r \leq s$ if r and s are strings and r is a substring of s. Let $h_T : M \to \{0,1\}$ be a binary classification rule for the methods. We define, for each $m \in M$:*

$$h_T(m) = \begin{cases} 1 & \text{if } \exists t \in T : t \leq lab_m \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

I/O-Based Labeling Policy

The idea beyond this kind of labeling is that the relation between the input and the output of a program can suggest something about the functionality of a program. For example, a program that takes an array of integers as input, and that returns another array of integers, could possibly be a program that fulfills some kind of sorting or filtering operations, while a program that requires as input an array, no matter its type, and that returns an object of the same type, can possibly represent some kind of search operation.

For the definition of this class of binary problems, we only consider a subset of all the possible I/O relations, namely the presence or the absence of an array among the input arguments and whether the returned object is an array or a single element. For easing the discussion, we adopt the following binary notation to describe such possible relations:

00: many to many
01: many to one
10: one to many
11: one to one

Following this notation, this labeling strategy can be formalized as follows:

Problem 3. *Let M be a set of non-void methods, each having at least one input argument. Let $L = \{00, 01, 10, 11\}$ be the set of possible labels for each $m \in M$. We remark that it exists a function $l : M \to L$ that assigns a label to each method. Let $\mathcal{P}(L)$ be the power set of L. Let $h_P : M \to \{0,1\}$ be a binary classification rule. We define, for each $m \in M$ and for a given $P \in \mathcal{P}(L)$:*

$$h_P(m) = \begin{cases} 1 & \text{if } l(m) \in P \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Random Labeling Policy

We finally define a baseline labeling strategy for assessing our results by comparing them with the results obtained while solving an arbitrary task whose results should be only noise. We simply consider a random split of the methods:

Problem 4. *Let L be a randomly shuffled list of methods, and let $m_i \in L$ be the method having index i. Given a threshold $n \in \mathbb{N}$ and let $h_n : L \to \{0,1\}$ be a binary classification function, we define, for each $m_i \in L$:*

$$h_n(m_i) = \begin{cases} 1 & \text{if } i \leq n \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

4.2. Classification

We tested the performance of the hidden neurons in classifying methods according to different instances of the classes of problems described in the previous section. To this aim, we considered all the neurons in the code layer and in the two hidden layers in the decoder, as shown in Figure 1, and we tested their classification accuracy by considering the activation produced by a neuron for each method given as input. The reason why we tested only the decoding neurons lies in the nature of an autoencoder: in the encoder layers a progressive dimensionality reduction (and thus a compression of information) is performed, and this (likely) means that in the middle code layer only relevant features are encoded. Since in the decoder layers the dimension is symmetrically increased for reconstructing the input, we decided to test only those neurons since they are expected to hold more relevant features. We remark that the same approach have been proposed, with promising results, for images [4] and for natural language [18], but it is new, to the best of our knowledge, for source code processing applications.

In detail, for each of the selected neurons, we considered as possible thresholds 10 equally spaced values among the minimum and the maximum activation value of that neuron for methods in the training set. For each activation threshold, we computed the classification accuracy of the neuron on a given problem instance by considering, in a precomputed balanced sample of the test set, the activation value of the neuron for that method and by predicting the method to be in class 0 or in class 1 if the activation value is less or greater than the threshold, respectively.

This process, formally described in the algorithm outlined in Algorithm 1, gives us a procedure for ranking neurons according to a task: we assign to each neuron its highest accuracy score. Table 1 shows the accuracies obtained by the best neuron for the considered problem instances, while the complete results obtained with the classification experiments will be discussed with further details in Section 6.

Table 1. Best accuracy score for each of the problems defined in Section 4.

Class	Instance	doc2vec	ast2vec
Random	none	54%	52%
Structural	$c = 10$	81%	84%
Semantic	$T = \{\text{test}\}$	63%	71%
Semantic	$T = \{\text{daemon}\}$	70%	68%
I/O	$\{00\}$ vs. $\{01, 10, 11\}$	64%	65%
I/O	$\{00, 10\}$ vs. $\{01, 11\}$	59%	64%

Algorithm 1 Algorithm for finding the best neuron in classifying programs on binary problems. The accuracy is computed by considering two balanced classes, each having at least 300 examples.

```
bestAcc ← 0                                              ▷ accuracy of the best neuron
for all neuron N do
    A ← activation values of N
    T ← activation thresholds          ▷ 10 evenly spaced thresholds between 0 and max a ∈ A
    best_N ← 0                                           ▷ best accuracy for N
    for all t ∈ T do
        pred ← empty list                                ▷ list of predictions
        for all a ∈ A do
            if a ≤ t then
                append 0 to pred
            else
                append 1 to pred
            end if
        end for
        if ACCURACY(pred) ≥ best_N then
            best_N ← ACCURACY(pred)                      ▷ update best accuracy of N
        end if
    end for
    if best_N ≥ bestAcc then
        bestAcc ← best_N                                 ▷ update best neuron
        bestNeuron ← N
    end if
end for
```

5. Scoring Neurons Independently of any Task

In the previous section, we described our experiments for evaluating the ability of individual neurons in solving specific classification problems or, in other words, in recognizing predetermined program properties. In the following, we propose a scoring measure for neurons based on the concept of *entropy* used in information theory. Further, we use the Mann–Whitney U statistical test for comparing the behavior of two neurons, assessing whether they share similar activation patterns and thus whether they are able to approximately detect the same concept.

5.1. Entropy and Single Neurons

The method we propose for evaluating the importance of each neuron in the network is based on the information-theoretic concept of entropy [25]. As we will discuss in Section 6, the experiments performed for assessing the results obtained with this scoring approach proved the effectiveness of this ranking, since it can discriminate among neurons which exhibit very simple activation patterns (i.e., active on only very few instances and therefore with low entropy), from more elaborate ones (i.e., those having varied activation values on many instances, corresponding to medium or high entropy).

The baseline idea behind this approach is that each neuron can be seen as a signaling system whose symbols are its activation values. Formally, in information theory, the entropy is defined as the average information obtained from a signaling system S which can output q different symbols $s_1, \ldots, s_q$ with probability $p_i = P(s_i)$:

$$H(S) = \sum_{i=0}^{i \leq q} p_i \log \frac{1}{p_i} \tag{6}$$

Dealing with activation values, whose domain is continuous over $\mathbb{R}_0^+$, we constructed a discretization of that space by considering a set $R = \{r_1, \ldots, r_{1000}\}$ of 1000 evenly spaced intervals between 0 and the maximum activation value reached by a neuron for the vectors in the training set, and we considered those intervals as the possible symbols of the neurons' alphabet. More precisely, for each neuron N we computed the activation values yielded on

a random sample of 10,000 vectors, and we determined the number of occurrences of each symbol by counting the activation values yielded in each interval r_i. We then considered the set $R_N \subseteq R$ of the occurring symbols in the neuron N and for each $r_i \in R_N$ we derived its occurring probability p_i using a softmax function over the set of countings. Then, we assigned to each neuron a score defined as the entropy computed over the set $P_N = \bigcup p_i$, where each p_i is the probability associated to the symbol r_i in R_N, as described by the algorithm reported in Algorithm 2.

Algorithm 2 Algorithm for computing the entropy of each neuron.

```
R ← list of intervals
for all neuron N do
    M ← random sample of 10000 methods
    V ← activations of N for each m ∈ M
    SCORE_NEURON(N, R, V)
end for

procedure SCORE_NEURON(N, R, V)
    C ← empty list
    for all r ∈ R do
        c ← number of v ∈ V such that v ∈ r
        append c to C
    end for
    remove all the 0s from C
    P ← SOFTMAX(C)
    return −∑_{p_i∈P} p_i log p_i
end procedure
```

5.2. Pairwise Neuron Comparison

We now provide further considerations on how entropy distinguishes neurons. The main insight is that it is always is possible to associate, to each neuron, a *concept* by simply looking at the instances that produces the highest activation values for that neuron. In other words, the concept corresponds to the binary classification task obtained by fixing an activation threshold and then by predicting the instances as satisfying that concept if the yielded activation value is higher than the threshold. Given this premise, we can define an heuristic procedure for measuring the similarity of the concepts defined by two neurons, by applying the Mann-Whitney U test in the following way:

1. Choose two neurons N_{ref} and N_{cf}, representing the neuron that defines the concept and the neuron to compare it to, respectively.
2. Considering the neuron N_{ref}, for each program instance $m_i \in M = \{m_1, \ldots, m_n\}$, compute the set of activation values $A = \{a_1, \ldots, a_n\}$ and create the list $L = \langle m_1, a_1 \rangle, \ldots, \langle m_n, a_n \rangle$, sorted according to a_i.
3. After splitting the sorted list L in three equally sized parts, generate the sets M_0 and M_1 by grouping the instances from the first and last of those parts, respectively.
4. Select two equally sized random samples of instances from M_0 and M_1 and compute the corresponding two sets C_0 and C_1 of activation values for N_{cf}.
5. Perform the Mann-Whitney U test on the sets of values C_0 and C_1, with alternative hypothesis that the distribution underlying the first set is stochastically *less* than the distribution underlying the second one.

Notice that, in our procedure, the threshold is represented by a range of values instead of a single point. The reason is to make the definition of the binary classification problem more robust, since we are removing the points in the middle that could likely give rise to confusion.

As it will be discussed in Section 6, by applying this procedure to all the possible pairs of neurons we are able to assess that different entropy values correspond to different

behaviors in terms of recognized concepts. Further, we show how the replicability of the concepts defined by the neurons varies when comparing neurons belonging to different entropy ranges.

6. Results Discussion

In this section, we introduce the results obtained with our experiments. We first analyze the performance of the neurons while solving different instances of the classification tasks described in Section 4, in the second part we look at the neurons from the information theoretic standpoint discussed in Section 5, while in the third part we compare pairs of neurons and we show how different entropy intervals clearly characterize specific behaviors.

6.1. Task-Based Experiments

A summary of the results obtained in the classification experiments is reported in Table 1. We considered different instances for the classification problems described in Section 4, and we evaluated the classification accuracy of each neuron. As can be seen in Figure 2, where we reported the accuracy distributions for some of the considered problem instances, for each problem most of the neurons reach an accuracy level between 0.5 and 0.55, while only a few neurons are indeed able to reach higher accuracies. This evidence is already interesting by itself since it means that single neurons perform differently when tested on a given task and also that some neurons are actually able to detect source-code-related properties. The accuracy varies a lot when considering different problem and different embeddings, but this is probably due to the features that are naturally seized by the vectors. Indeed, in our experiments this is confirmed by the good results obtained for the structural task with the ast2vec embedding (first diagram in Figure 2). Finally, the experiments on the baseline random problem confirm the validity of our results showing how the performance of the neurons on a randomly defined problem is far from being comparable to the one obtained on all the other tasks, hence good performances are not emerging by chance.

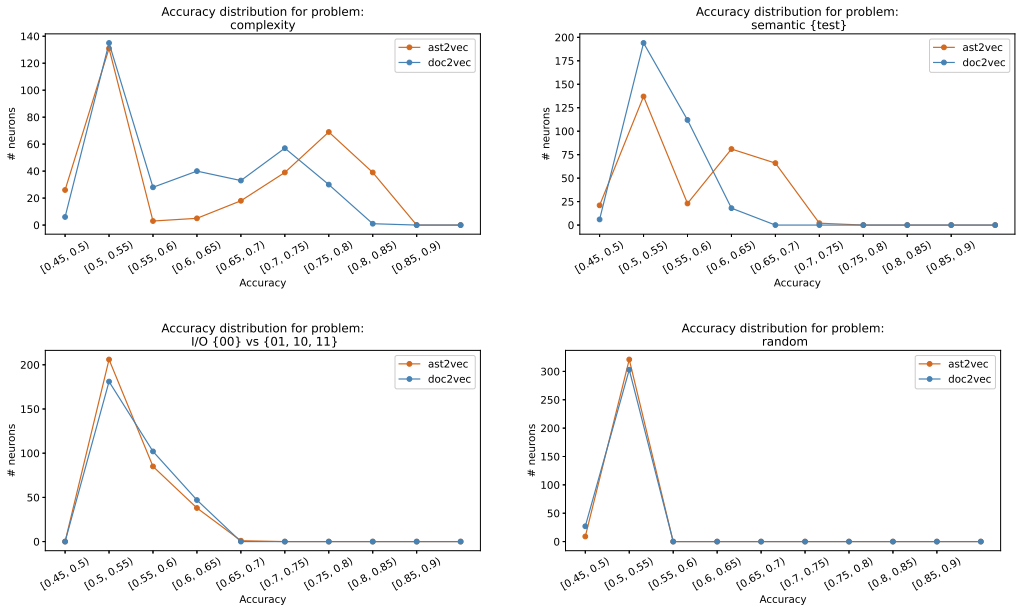

Figure 2. Classification accuracies reached by the neurons on different problem instances (see Section 4).

6.2. Task-Independent Experiments

We can start from the original goal of information theoretic entropy: it measures the average level of information of a source given its outcome. In our setting, each neuron can be considered a source of symbols, when we interpret its possible activation values as explained in Section 5.1.

Our measure of entropy allows us to distinguish (internal) neurons with respect to the variety of the activation values they output for each instance presented to the network. Eventually, each neuron's behavior in terms of output activation values is defined by the training process and, when considering internal neurons, also by its connectivity to the rest of the network.

Here we perform experiments aimed at showing how such a measure can be used to identify neurons that can be used to perform some interesting classifications of input instances. Previously, we identified interesting neurons by first specifying some classification problem, and then by measuring the performance of each neuron on it. Differently, the interest here is to specify what can characterize interesting classifications so as to look for corresponding behaviors among the internal neurons.

In this work, we firstly chose to understand how the behavior of neurons varies with different entropy values, and when operating on two different ways of embedding source code input instances.

To this aim, we first plotted the distribution of entropy values among the neurons of the autoencoder's section highlighted in Figure 1. The resulting distributions, in Figure 3, are qualitatively similar under both doc2vec and ast2vec embeddings, with a bimodal profile characterized by a peak of occurrences for very low values of entropy and an area of normally distributed frequencies for higher entropy values.

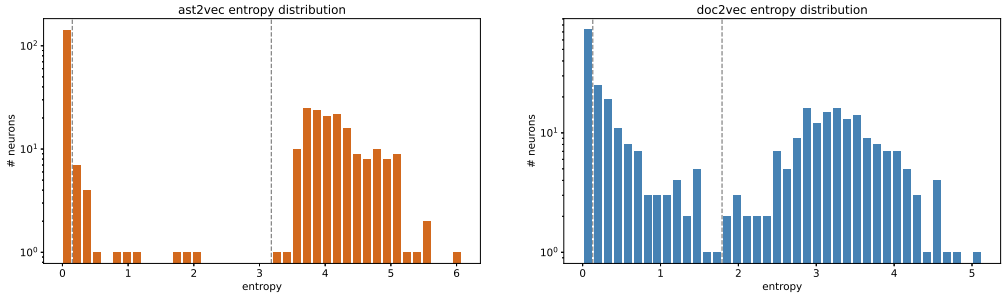

Figure 3. Distribution of the neurons' entropies in the ast2vec (**left**) and doc2vec (**right**) models. Vertical dashed lines separate the three classes of neurons described in Section 6.2.

Therefore, three classes of neurons having different entropy values can be roughly distinguished:

1. A big number of neurons (notice that the figure is in a logarithmic scale) having an entropy equals or very close to 0. These neurons are of no interest in this context since they are neurons that (almost) never activate. They could only be used for pruning the network in order to optimize the architecture, but it is out of the focus of this work.
2. Another big class of neurons having normally distributed high entropy values. Those neurons reach an high score since their activation values are distributed over a wide range. In addition, the probabilities of the occurring activation values to be in distinct intervals are relatively similar: this leads to an high score in terms of information theory.
3. A smaller set of neurons whose values are higher than 0 but that are out of the normal distribution of the majority of the values. The corresponding activation values are those between the dashed bars plotted in Figure 3. As it will be clear by the discussion in Section 6, those neurons are peculiar since they produce an activation higher than 0

only for a significant amount of vectors (i.e., >10%), while in all the other cases their activation is equal to 0.

For two neurons, one with low entropy and one with high entropy, we plotted their activation values for a set of input instances in Figure 4 (left). More related to our entropy measure, we show in the same Figure 4 (right) the distribution of symbols, defined on intervals of activation values, for the same two neurons.

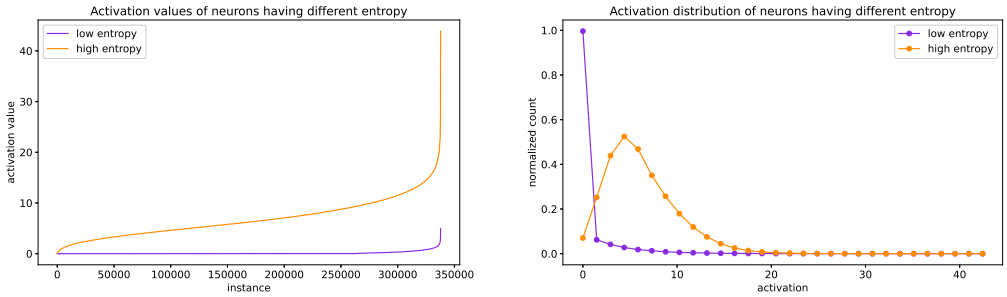

Figure 4. Activations of neurons having different entropies. In both the images, the purple line refers to a neuron having a low entropy score $H = 1.06$, while the orange line represents a neuron with an high entropy $H = 5.19$. The left figure is a simple plot of the sorted activation values for the two neurons, while the left figure reports the distribution of the activation values.

Moreover, even if in this analysis we consider the two embeddings systems as black boxes, the outcomes of these experiments, as presented by the data in Figure 3, are revealing something about how the ensembles of neurons of the two autoencoders are trained, with the two different vector datasets. The autoencoders eventually achieve their goals in different ways, with respect to the set of behaviors learned by their neurons. For ast2vec, Figure 3 tells us that only about half of the neurons have high entropy (above 3), while the others have a null or very small entropy value. It appears that the autoencoder is able to reconstruct the input with little contribution from many of its neurons. Instead, the autoencoder operating on the doc2vec instances eventually computes its output from the collective working of neurons with more diverse behavior in terms of their entropy, and with a smaller percentage of neurons with entropy of 0 or close to 0. The different training outcomes associated with the different datasets suggest further analysis of the datasets and of their distributional structures.

6.3. Pairwise Neuron Comparison Experiments

When exploring what neurons are representing, we experimented with the comparing measure introduced in Section 5. We take two neurons, N_{ref} and N_{cf}, and we assign to the first a reference role, and the second one will be compared to it in terms of how they can classify input instances. Preliminary findings show that experiments with ast2vec and doc2vec produce very similar results, and for this reason, all the following discussion is made by considering only the neurons in the ast2vec autoencoder.

As stated in Section 5.2, activations of a neuron on instances define its classifying behavior, and thus we will compare two neurons in terms of how a neuron can approximate the classification of another. For instance, looking at Figure 5 (right), we can see how a chosen neuron N_{cf} activates on negative instances (set M_0, bars in red) or positive (set M_1, bars in blue) for neuron N_{ref}. In this case, it appears that we can find a threshold on activation values good to classify most of the instances in the same way as the chosen N_{ref}.

Finally, we can see that in general such a threshold could be found when the medians of the two distributions, one from the activation values of N_{cf} on M_0 and the other from its activations on M_1, are separated, and we assess this with Mann–Whitney U test on those pairs of distributions.

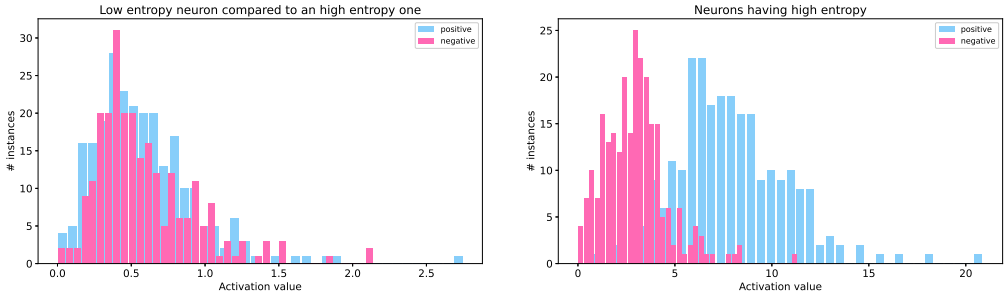

Figure 5. Activations of two high-entropy reference neurons, one compared against a low-entropy neuron (**left**) and one against a high-entropy neuron (**right**), both in the role of N_{cf}. Colors of bars correspond to the classes M_0 and M_1 defined by the reference neuron N_{ref}.

We remark that the results we will outline shortly still stay valid when considering the alternative hypothesis of having the first distribution stochastically *greater* than the second one. In fact, this would mean that instances producing a high activation level on the first neuron tend to produce a low activation level on the second one and vice versa. Therefore, in this case the binary classification produced by the first neuron could be approximated by the opposite of the classification produced by the second one.

Overall, we explored the reproducibility of a given classification, the one from a neuron N_{ref} by a second neuron N_{cf}, on every pair of neurons from the autoencoder section previously considered. Now we present the results and the insights of these experiments.

The first evidence is that there is a correlation between entropy of the two neurons we compare as described above. In Figure 6, we picture the success ratio of the test according to the entropy of the neurons being compared. When both the reference and the comparison neurons have high entropy, the Mann–Whitney U test favors the *less* hypothesis, with $p < \alpha$ where $\alpha = 0.01$ is the considered significance level.

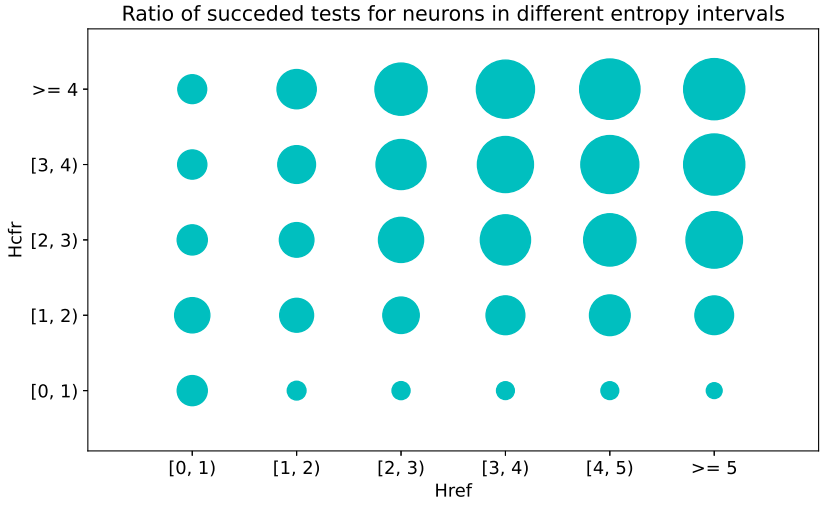

Figure 6. Ratio of successful Mann–Whitney U tests while comparing neurons belonging to different entropy ranges. When tests succeed the classification behavior of neuron N_{ref} can be approximated by defining a threshold on the activation levels of neuron N_{cf}.

For instance, we show in Figure 5 the details of two comparisons, one (left) where we compare a neuron with low entropy to a neuron with high entropy and another (right) where we compare two neurons having high entropy. The bars represent the distribution of the activation values for the instances in the two classes M_0 and M_1 defined by N_{ref}. Specifically, the colors of the bars are associated with the classification of instances of N_{ref}, while their position and height represent the activation levels produced by N_{cf} on the same

instances. The two distributions on which the Mann–Whitney U test are performed are the activation values of the N_{cf} for instances associated to the two sets M_0 and M_1, respectively. In the first plot (left), we report the comparison of two neurons over which the test fails; that is, it is not possible to distinguish among the two distributions, and thus it is probable that the two medians are not one less than the other. We read this as evidence that no threshold is good enough to separate red and blue bars, while in the other one (right), this can be accomplished with some approximation.

Putting Things Together

The two measures we introduced, the entropy of each neuron and the similarity check between two neurons based on Mann–Whitney U test, are aimed at looking for (internal) neurons which can recognize interesting concepts. Therefore, we finally check how those measures apply on neurons which performed well in one of the specific tasks we presented in the previous section.

We took the neurons having accuracy above 0.8 on the task of estimating the cyclomatic complexity of source code. The evidence we gathered show that:

- Over the total 330 neurons, there are 39 under ast2vec embedding and only 1 for doc2vec,
- They all have entropy greater than 3.
- Some of the other neurons having high entropy perform badly on the task.
- Choosing a neuron performing well on the task, its comparison to low entropy neurons always fails (see Figure 7 left), and the outcome of its comparison to medium or high entropy neurons can be related to how good their accuracy is on the task (see Figure 7 right).

With our entropy measure and our comparison based on Mann–Whithey U test, we could first select and then group neurons which could be considered to be representative of specific learned concepts. In general, neurons having high entropy exhibit behaviors that appear to be the most interesting, and they can be compared, with respect to the concepts they can recognize, to neurons toward which our test succeeds.

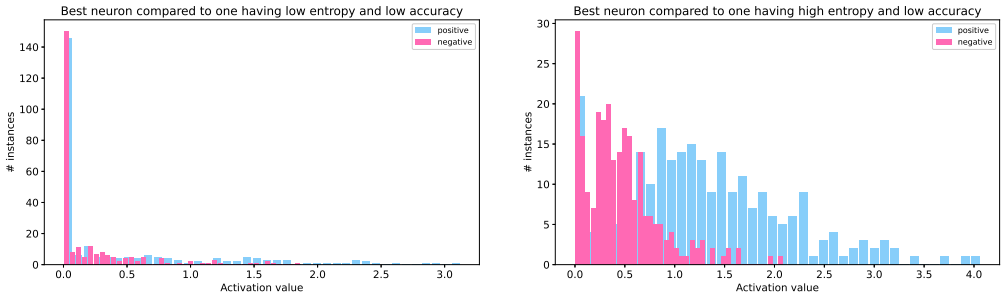

Figure 7. For the cyclomatic complexity classification, we show the outcome of our comparison between the best neuron in the task and two neurons not performing the same, with the first having low entropy (**left**) and the second one having high entropy (**right**).

7. Conclusions and Further Directions

This work aims at analyzing the dynamics of the internal parts of a neural network and at extracting knowledge from there. We approached this for a specific application domain, that of source code analysis, where what can be considered interesting knowledge cannot be defined as easily as when, for instance, recognizing objects in the natural domain of images.

Our approach is twofold. In the first part, we evaluate the performance of neurons belonging to an independently trained network when using them as classifiers for tasks related to properties of source code snippets. We then define when, in the same network, a neuron has a dynamics deemed interesting according to information-theoretic or statistical

measures. We applied our methods to a simple autoencoder whose input was obtained from source code by using two different embeddings, one that just considers the linear sequence of words in the program lines (i.e., as if dealing with natural language) and another that also takes into account the formal parsing of the given programming language. We are able to show that:

- Several internal neurons perform well on tasks related to syntactic properties of code, such as its cyclomatic complexity, a common software engineering metric.
- The two embeddings we used performed differently when considering different tasks.
- Neurons can be algorithmically selected based on the richness of their activation dynamics.
- All the neurons that perform well on known tasks also reach high scores with our entropy measure.
- By choosing appropriate thresholds on activation values, in order to classify instances, neurons with high entropy are able to approximate each other's behavior.

Notice that the autoencoder operates on a "transformed" version of the source code (namely the program vectors), and thus the results obtained, which result from the information given by the data after the embedding process, are affected not only by the neural model under analysis but also by the chosen neural embedder.

Further work will be to apply our methods to explore the behavior of internal neurons of more sophisticated networks. For instance, recent works show how neural transformer models can be fruitfully used for source code [15,16]. The techniques we introduced could be employed to look for neurons which perform well on known tasks, even if belonging to a network trained while keeping in mind other goals. However, it would also be interesting to analyze how neurons classify with respect to the richness of their activation patterns and to group them by similarity, as allowed by our information-theoretic measures. Finally, we could study how to associate human-understandable concepts with the discovered internal activation patterns, similarly to what other authors did in the field of image understanding [19]. We expect that the more the neural model is structured and powerful, the more the measures we are introducing can prove their effectiveness in studying the internal representations and the developed knowledge of neural systems.

Author Contributions: Conceptualization, M.S. and C.F.; Software, M.S.; Investigation, M.S. and C.F.; Writing—original draft, M.S. and C.F.; Supervision, C.F. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data is contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Erhan, D.; Courville, A.; Bengio, Y.; Vincent, P. Why does unsupervised pre-training help deep learning? In Proceedings of the 13th International Conference on Artificial Intelligence and Statistics, JMLR Workshop and Conference Proceedings, Sardinia, Italy, 13–15 May 2010; pp. 201–208.
2. Zhuang, F.; Qi, Z.; Duan, K.; Xi, D.; Zhu, Y.; Zhu, H.; Xiong, H.; He, Q. A comprehensive survey on transfer learning. *Proc. IEEE* **2020**, *109*, 43–76. [CrossRef]
3. Gilpin, L.H.; Bau, D.; Yuan, B.Z.; Bajwa, A.; Specter, M.; Kagal, L. Explaining explanations: An overview of interpretability of machine learning. In Proceedings of the IEEE 5th International Conference on data science and advanced analytics (DSAA), Turin, Italy, 1–3 October 2018; pp. 80–89.
4. Le, Q.V.; Ranzato, M.; Monga, R.; Devin, M.; Corrado, G.; Chen, K.; Dean, J.; Ng, A.Y. Building high-level features using large scale unsupervised learning. In Proceedings of the 29th International Conference on Machine Learning, ICML, PMLR, Edinburgh, Scotland, 26 June–1 July 2012; pp. 507–514.
5. Dalvi, F.; Durrani, N.; Sajjad, H.; Belinkov, Y.; Bau, A.; Glass, J.R. What Is One Grain of Sand in the Desert? Analyzing Individual Neurons in Deep NLP Models. In Proceedings of the 33rd AAAI Conference on Artificial Intelligence, Honolulu, HI, USA, 27 January–1 February 2019; pp. 6309–6317.

6. Le, T.H.M.; Chen, H.; Babar, M.A. Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges. *ACM Comput. Surv.* **2020**, *53*, 62:1–62:38. [CrossRef]
7. Allamanis, M.; Barr, E.T.; Devanbu, P.T.; Sutton, C. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* **2018**, *51*, 81:1–81:37. [CrossRef]
8. Del Carpio, A.F.; Angarita, L.B. Trends in Software Engineering Processes using Deep Learning: A Systematic Literature Review. In Proceedings of the 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Portoroz, Slovenia, 26–28 August 2020; pp. 445–454. [CrossRef]
9. Liu, F.; Li, G.; Wei, B.; Xia, X.; Fu, Z.; Jin, Z. A Self-Attentional Neural Architecture for Code Completion with Multi-Task Learning. In Proceedings of the 28th International Conference on Program Comprehension, ICPC, ACM, Seoul, Republic of Korea, 13–15 July 2020; pp. 37–47.
10. Allamanis, M.; Peng, H.; Sutton, C.A. A Convolutional Attention Network for Extreme Summarization of Source Code. In Proceedings of the 33nd International Conference on Machine Learning, ICML, New York, NY, USA, 19–24 June 2016; pp. 2091–2100.
11. Mou, L.; Li, G.; Zhang, L.; Wang, T.; Jin, Z. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, Phoenix, AZ, USA, 12–17 February 2016; pp. 1287–1293.
12. Allamanis, M.; Brockschmidt, M.; Khademi, M. Learning to Represent Programs with Graphs. In Proceedings of the 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, 30 April–3 May 2018.
13. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.* **2019**, *3*, 40:1–40:29. [CrossRef]
14. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, L.; Polosukhin, I. Attention is All you Need. In Proceedings of the Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems (NIPS), Long Beach, CA, USA, 4–9 December 2017; pp. 5998–6008.
15. Kanade, A.; Maniatis, P.; Balakrishnan, G.; Shi, K. Learning and evaluating contextual embedding of source code. In Proceedings of the 37th International Conference on Machine Learning, ICML 2020, Virtual, 12–18 July 2020.
16. Ahmad, W.; Chakraborty, S.; Ray, B.; Chang, K.W. Unified Pre-training for Program Understanding and Generation. In Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Online, 6–11 June 2021; pp. 2655–2668.
17. Yu, S.; Príncipe, J.C. Understanding autoencoders with information theoretic concepts. *Neural Netw.* **2019**, *117*, 104–123. [CrossRef] [PubMed]
18. Radford, A.; Jozefowicz, R.; Sutskever, I. Learning to Generate Reviews and Discovering Sentiment. *arXiv* **2017**, arXiv:1704.01444.
19. Kim, B.; Wattenberg, M.; Gilmer, J.; Cai, C.J.; Wexler, J.; Viégas, F.B.; Sayres, R. Interpretability beyond Feature Attribution: Quantitative Testing with Concept Activation Vectors (TCAV). In Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholm, Sweden, 10–15 July 2018; Volume 80, pp. 2673–2682.
20. McGrath, T.; Kapishnikov, A.; Tomašev, N.; Pearce, A.; Wattenberg, M.; Hassabis, D.; Kim, B.; Paquet, U.; Kramnik, V. Acquisition of chess knowledge in alphazero. *Proc. Natl. Acad. Sci. USA* **2022**, *119*, e2206625119. [CrossRef] [PubMed]
21. Saletta, M.; Ferretti, C. Towards the evolutionary assessment of neural transformers trained on source code. In Proceedings of the GECCO'22: Genetic and Evolutionary Computation Conference, Companion Volume, Boston, MA, USA, 9–13 July 2022; Fieldsend, J.E., Wagner, M., Eds.; ACM: 2022; pp. 1770–1778. [CrossRef]
22. Ferretti, C.; Saletta, M. Do Neural Transformers Learn Human-Defined Concepts? An Extensive Study in Source Code Processing Domain. *Algorithms* **2022**, *15*, 449. [CrossRef]
23. Goodfellow, I.J.; Bengio, Y.; Courville, A.C. *Deep Learning*; Adaptive Computation and Machine Learning; MIT Press: Cambridge, MA, USA, 2016.
24. Bau, A.; Belinkov, Y.; Sajjad, H.; Durrani, N.; Dalvi, F.; Glass, J.R. Identifying and Controlling Important Neurons in Neural Machine Translation. In Proceedings of the 7th International Conference on Learning Representations, ICLR, New Orleans, LA, USA, 6–9 May 2019.
25. Shannon, C.E. A mathematical theory of communication. *Bell Syst. Tech. J.* **1948**, *27*, 623–656. [CrossRef]
26. Zeiler, M.D. ADADELTA: An Adaptive Learning Rate Method. *arXiv* **2012**, arXiv:1212.5701.
27. Chollet, F. Keras. Available online: https://keras.io (accessed on 4 April 2023).
28. Alon, U.; Brody, S.; Levy, O.; Yahav, E. code2seq: Generating Sequences from Structured Representations of Code. In Proceedings of the 7th International Conference on Learning Representations, ICLR, New Orleans, LA, USA, 6–9 May 2019.
29. Le, Q.V.; Mikolov, T. Distributed Representations of Sentences and Documents. In Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21–26 June 2014; pp. 1188–1196.
30. Řehůřek, R.; Sojka, P. Software Framework for Topic Modelling with Large Corpora. In Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks, Valletta, Malta, 22 May 2010; pp. 45–50.
31. Smith, N.; van Bruggen, D.; Tomassetti, F. *JavaParser: Visited*, Version Dated 5 February 2021; GitHub, Inc.: San Francisco, CA, USA, 2021.

32. Saletta, M.; Ferretti, C. A Neural Embedding for Source Code: Security Analysis and CWE Lists. In Proceedings of the 18th IEEE International Conference on Dependable, Autonomic and Secure Computing, DASC/PiCom/CBDCom/CyberSciTech, Calgary, AB, Canada, 17–22 August 2020; pp. 523–530.
33. Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.S.; Dean, J. Distributed Representations of Words and Phrases and their Compositionality. In Proceedings of the Advances in Neural Information Processing Systems 26, Proceedings of NIPS, Lake Tahoe, Nevada, 5–10 December 2013; pp. 3111–3119.
34. Allen, F.E. Control Flow Analysis. *ACM Sigplan Not.* **1970**, *5*, 1–19. [CrossRef]
35. McCabe, T.J. A Complexity Measure. *IEEE Trans. Softw. Eng.* **1976**, *2*, 308–320. [CrossRef]
36. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. A general path-based representation for predicting program properties. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, Philadelphia, PA, USA, 18–22 June 2018; pp. 404–419.