applied sciences

MDPI

Article

Machine Learning-Based Code Auto-Completion Implementation for Firmware Developers

Junghyun Kim [1], Kyuman Lee [2,*] and Sanghyun Choi [3]

[1] School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA; andy.kim@gatech.edu
[2] Department of Robot and Smart System Engineering, Kyungpook National University, Daegu 41566, Korea
[3] Memory S/W Development Team, Samsung Electronics, Hwasung 18448, Korea; sh518.choi@samsung.com
* Correspondence: klee400@knu.ac.kr

Received: 29 October 2020; Accepted: 19 November 2020; Published: 28 November 2020

Abstract: With the advent of artificial intelligence, the research paradigm in natural language processing has been transitioned from statistical methods to machine learning-based approaches. One application is to develop a deep learning-based language model that helps software engineers write code faster. Although there have already been many attempts to develop code auto-completion functionality from different research groups, a need to establish an in-house code has been identified for the following reasons: (1) a security-sensitive company (e.g., Samsung Electronics) may not want to utilize commercial tools given that there is a risk of leaked source codes and (2) commercial tools may not be applicable to the specific domain (e.g., SSD firmware development) especially if one needs to predict unique code patterns and style. This research proposes a hybrid approach that harnesses the synergy between machine learning techniques and advanced design methods aiming to develop a code auto-completion framework that helps firmware developers write code in a more efficient manner. The sensitivity analysis results show that the deterministic design results in reducing prediction accuracy as it generates output in some unexpected ways, while the probabilistic design provides a list of reasonable next code elements in which one could select it manually to increase prediction accuracy.

Keywords: machine learning; code auto-completion; GPT-2 model; advanced design methods

1. Introduction

1.1. Research Motivation

Firmware software developers at a company are typically responsible for developing a software program that operates a product. A company always seeks to provide a streamlined work process for firmware software developers to increase productivity as the process leads to saving money for the company. One potential barrier for increasing productivity is to spend considerable time writing code that is particularly due to a repetitive task. Another potential problem is that firmware software developers may be generating similar codes simultaneously as they are separately involved in developing different hardware products. Figure 1 notionally illustrates the issue where we noticed that two firmware software developers separately worked on each code that was eventually similar to each other. This situation could prevent them from working efficiently if they would need to handle a large volume of source codes, resulting in decreasing productivity. Thus, a need to develop a framework that helps firmware software developers work efficiently has been identified in this research.

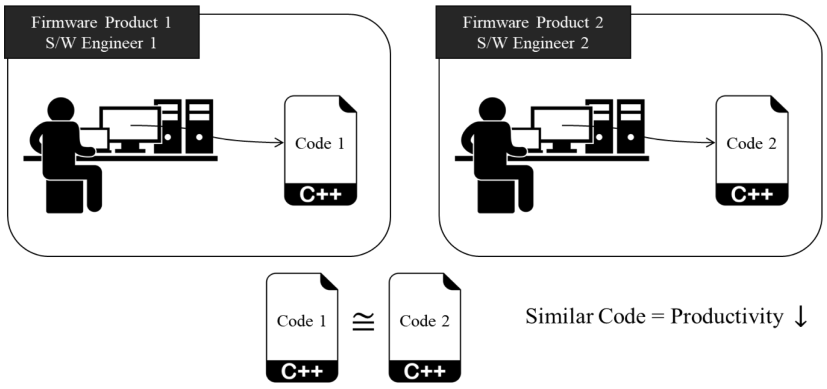

Figure 1. Notional sketch of necessity for developing a code auto-completion framework.

1.2. Background

With the advent of machine learning (ML) techniques, the research paradigm in various engineering areas has been recently transitioned from theory-based to data-driven approaches [1]. There have already been many studies asserting that ML techniques outperform traditional statistical methods. A language model is no exception to this paradigm shift. Many research groups have been committed to developing a language model using ML techniques. For example, Google developed the bidirectional encoder representations from transformers (BERT) [2] that mainly use transformer encoder blocks. OpenAI released the generative pre-trained transformer (GPT) models [3,4] such as the GPT-2 models.

The GPT models are transformer-based language models trained on a massive text dataset from the websites. Depending on the size of neural network weight parameters, the GPT-2 models are classified into small (i.e., 117 M), medium (i.e., 345 M), and large (i.e., 762 M) pre-trained models, as shown in Figure 2. Here, 117 M means that there are 117 million parameters of the neural network model. The obvious upside of the GPT models is that the pre-trained models can be easily tailored to various domain-specific language modeling tasks, given that the models are fine-tuned with domain-specific training datasets. For this reason, the GPT models have been widely used for a variety of domain-specific tasks such as speech recognition and language translation.

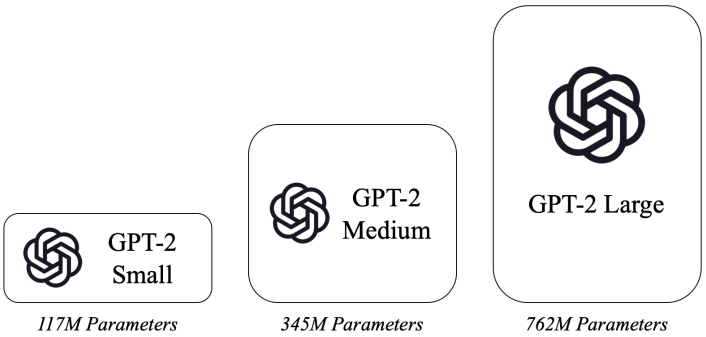

Figure 2. GPT-2 models (reproduced from [5]).

In fact, the GPT models have stunned the world by demonstrating the impressive capability that may exceed the current language models. One example is the Allen AI GPT-2 Explorer [6] as shown in Figure 3 , where it uses the GPT-2 345M model to predict the most likely next word alongside their probability score. In this example, it appears that the model generates a list of candidates for the new few words (e.g., Electronics Co., Ltd.) once some initial text (e.g., I am currently working at Samsung) is provided.

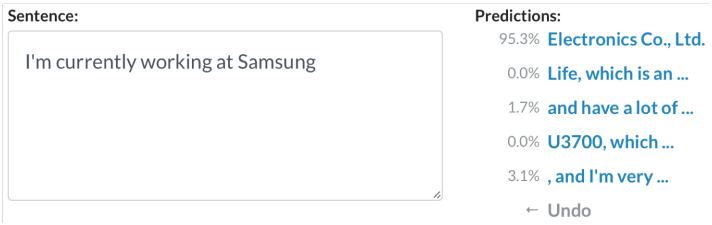

Figure 3. AllenNLP language modeling demonstration example.

Given the aforementioned observations, it can be hypothesized if the pre-trained GPT-2 models are fine-tuned with SSD firmware source codes, the fine-tuned model predicts the most likely next code element. To that end, this research aims to develop a framework that deploys the GPT-2 models to help SSD firmware developers write code in a more efficient manner. The remainder of this paper consists of the following: Literature Review, Proposed Methodology, Results and Discussion, and Conclusion.

2. Literature Review

In relation to the research objective, there have already been many attempts to develop similar capabilities. This section is aimed at reviewing the advances and limitations of the previous efforts about code auto-completion functionality, which helps identify research gaps that need to be bridged.

2.1. Related Work

Code auto-completion functionality has been considered as one of the most essential functions for software engineers. The Integrated Development Environment (IDE) has provided a set of effective features that include code auto-completion capability [7]. The code auto-completion feature in the IDEs basically suggests next probable code elements; however, there are some potential issues that have been identified [8]: (1) the feature requires an exhaustive set of rules, (2) predictions do not consider the category of code, (3) predictions do not consider context such as class definition, and (4) recommendations are often lexicographical and alphabetical, which may not be very useful.

Many research groups have adopted statistical methods to resolve the potential issues of the IDEs. For example, Sebastian Proksch et al. [9] replaced an existing code auto-completion engine by an approach using Bayesian networks named pattern-based Bayesian network (PBN). Raychev et al. [10] proposed the state-of-the-art probabilistic model for code auto-completion functionality, which is mainly equipped with the n-gram model that computes the probability of the next code elements given previous n elements. The statistical approach, however, examines only a limited number of elements in the source codes when completing the code; thus, the effectiveness of this approach may not scale well to large programs [11].

With the advent of deep learning, many research groups have been committed to developing deep learning-based code auto-completion functionality. The most common technique is to use a recurrent neural network (RNN). In fact, Karampatsis et al. [12] showed that the RNN-based language models would be much better than the traditional statistical methods. Moreover, Martin White et al. [13] illustrated how to use the RNN-based language model to facilitate the code auto-completion task. It seemed that RNN-based language models gained the most popularity at the time; however, it was identified that the models were limited by the so-called hidden state bottleneck: all the information about the current sequence is compressed into a fixed-size vector. This limitation made it hard for the RNN-based models to handle long-range dependencies [14].

A transformer-based language model has been introduced to overcome a major drawback of an RNN-based language model by relying on the attention mechanism. For example, Alexey Svyatkovskiy introduced IntelliCode Compose [15], which is capable of predicting sequences of code tokens of arbitrary types. It leveraged the state-of-the-art generative transformer model trained on 1.2 billion lines of source codes. In addition to the IntelliCode Compose, a variety of transformer-based language models have recently achieved excellent work [2–4,16] for various natural language processing

(NLP) tasks such as language modeling. There are numerous practical applications that deploy a transformer-based language model for code auto-completion functionality. TabNine [17] published a blog post mentioning the use of GPT-2 model in their code auto-completion feature. However, they never revealed technical details about the modeling process. Kite-Pro [18], which also employs the transformer-based language model, reports on average 18 percent more efficiency by using the code auto-completion feature. Table 1 summarizes four different approaches with the advantages and limitations of the previous efforts about code-completion functionality.

Table 1. Comparative table of the related works.

Approach	Example Method	Advantage	Limitation
Integrated Development Environment (IDE)	Eclipse's content assist feature	It provides a list of type-compatible names for the next tokens immediately	It is generally organized alphabetically, which is not very effective
Statistical Language Models	n-gram	It solves a drawback of IDE's ineffectiveness by optimizing/ranking the list	It is difficult to be scaled to large programs
Machine learning-based Language Models	Recurrent Neural Network (RNN)	It is typically better than statistical language models in predictions	It is not effective to capture long-term dependencies
Transformer-based Language Models	Generative Pre-trained Transformer (GPT)	It overcomes the issue related to long-term dependencies by introducing attention mechanisms	It is computationally expensive for those who use a personal computer

2.2. Research Gap

Although many research groups have deployed a transformer-based language model for code auto-completion functionality, it is important to note that they have trained the model, with open source codes mostly coming from GitHub. For example, the Deep TabNine [17] is trained on around 2 million files from GitHub. This indicates that the software may not be applicable if one needs to predict very unique code patterns and style. Therefore, a need to establish a domain-specific language model has been identified based on the following reasons: (1) firmware codes implement very specific sets of features for the hardware and (2) firmware codes typically comply with unique coding styles and patterns optimized for embedded environments.

In fact, some companies (e.g., TabNine) advertise that they offer a GPU-based cloud service that enables users to create a custom model by fine-tuning the model with their input data. However, a security-sensitive company such as Samsung Electronics may not want to utilize the commercial tools given that there is a risk of leaked source codes. In addition to the security issue, a company has to pay for the license fee because the tools are not free to use the service. Thus, a need to develop in-house codes for code auto-completion functionality is identified.

As we seek to develop a domain-specific (i.e., solid state drive (SSD) firmware development) language model by using the GPT-2 model, it naturally leads us to consider how to determine diversity parameters (i.e., Top_k, Top_p, and Boltzmann temperature) of the model. Given that there has not been any analysis done on the optimal diversity parameter values especially on the SSD firmware development domain, the following research question can be constructed: "How can we determine the GPT-2 diversity parameter values properly for the SSD firmware development domain"?

To answer the question, in this paper, we propose a hybrid approach that harnesses the synergy between ML techniques and advanced design methods (e.g., design of experiment, surrogate modeling, and Monte Carlo simulation) [19] to enhance the level of understanding of the relationship between

the GPT-2 model diversity parameters and code auto-completion functionality in the SSD firmware development domain. Figure 4 notionally illustrates the process of the hybrid approach used for this research.

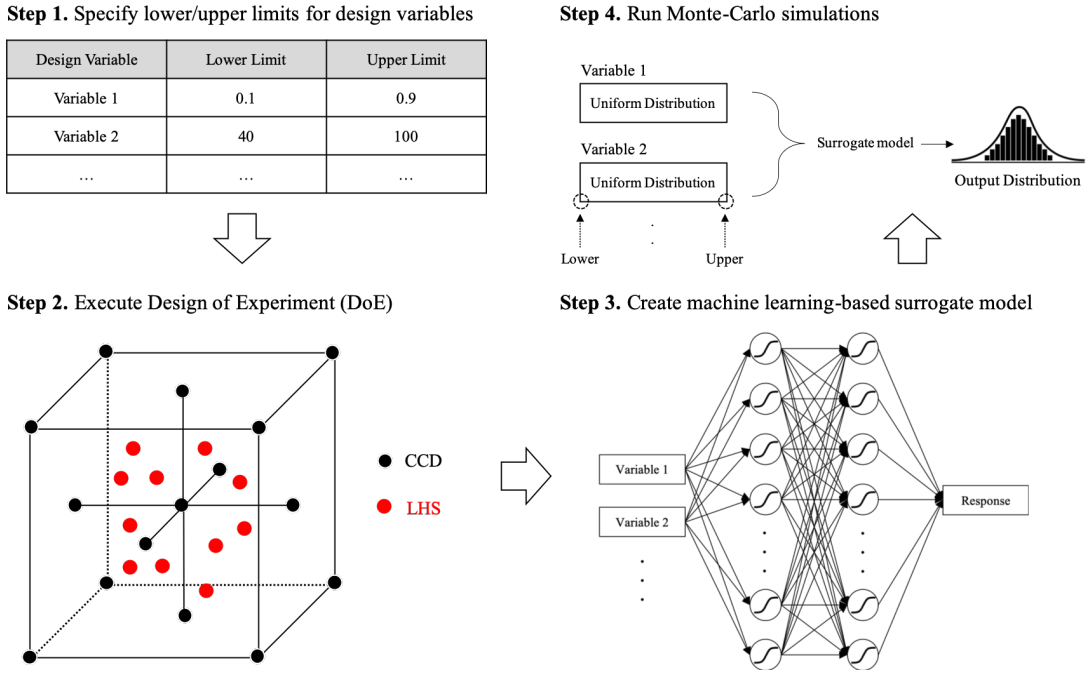

Figure 4. Notional sketch of the process of the hybrid approach used for this research.

3. Proposed Methodology

3.1. Overview of the Methodology

This research aims to not only develop a framework that deploys GPT-2 117M model to help firmware developers write code in a more efficient manner, but also unravel the hidden relationships between the GPT-2 model diversity parameters and code auto-completion capability. Figure 5 depicts an overview of the proposed methodology.

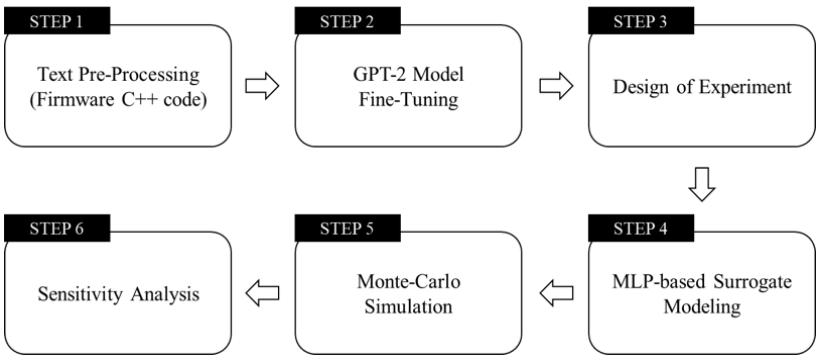

Figure 5. Overview of the proposed methodology.

The framework is a Python-based program that consists of several modules with its primary data sources. There are three different modules: (1) the first module is designed to automate data pre-processing, such as removing all unnecessary C++ code comments, (2) the second module performs fine-tuning for the GPT-2 model with optimized hyper-parameters (i.e., batch size and learning rate), and (3) the third module employs advanced design methods for diversity parameter sensitivity analysis.

3.2. Text Pre-Processing

To customize the original GPT-2 117M model to the SSD firmware development domain, it is imperative to prepare input data properly for the fine-tuning process, because the process may require understanding input data in its own way. Since the source codes include unnecessary information (e.g., C++ code comments) that may deteriorate training data quality, we develop a Python code that automatically removes all unnecessary code through the pattern analysis. Once the Python code completes the removal process, it also removes white spaces as well as empty lines. It then combines all the source codes into one single text file with the delimiter in order to allow the model to learn the formatting of the training data.

3.3. GPT-2 Model Fine-Tuning

The GPT-2 model is a transformer-based language model trained with a massive 40GB text data that mainly includes web pages [18]. Users can fine-tune the GPT-2 model with new input data. During the fine-tuning process, users can either increase or decrease two hyper-parameters, namely batch size and learning rate, to optimize the model's predictive capability. We employ the grid search method as shown in Figure 6 and test all candidate cases on the NVIDIA DGX-1 machine (i.e., Volta 32GB version) to isolate the hyper-parameters. The effective model is finally determined by minimizing the log-loss value. The choice of hyper-parameters is tabulated in Table 2. Figure 7 shows the plot of the loss curve describing that the loss value is actually converged with the chosen hyper-parameters.

Table 2. Grid search results for the hyper-parameters.

Hyper-Parameter	Final Choice
Learning rate	0.0001
Batch size	Stochastic

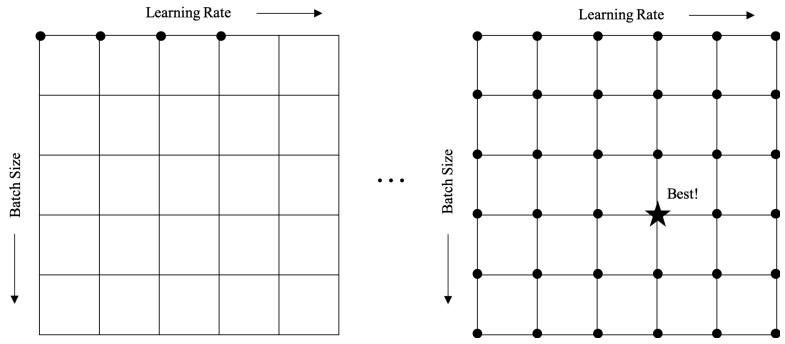

Figure 6. Notional sketch of the grid search method for isolating hyper-parameters.

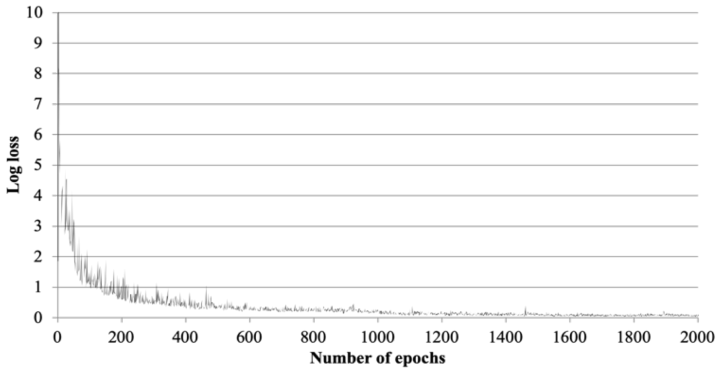

Figure 7. Plot of the loss history of the tailored GPT-2 model with isolated parameters.

3.4. Design of Experiment

The design of experiment (DoE) is a procedure that selects samples in the design space to maximize the amount of information with a limited set of experiments. To generate a non-linear surrogate model that represents the design space of the GPT-2 model's diversity parameters, we employ two representative DoE methods: (1) the Latin hypercube sampling (LHS) method is used to capture inner points of the design space and (2) the full factorial design with three factors is utilized to capture corner points of the design space. Figure 8 shows how samples are distributed in the design space of the GPT-2 model's diversity parameters.

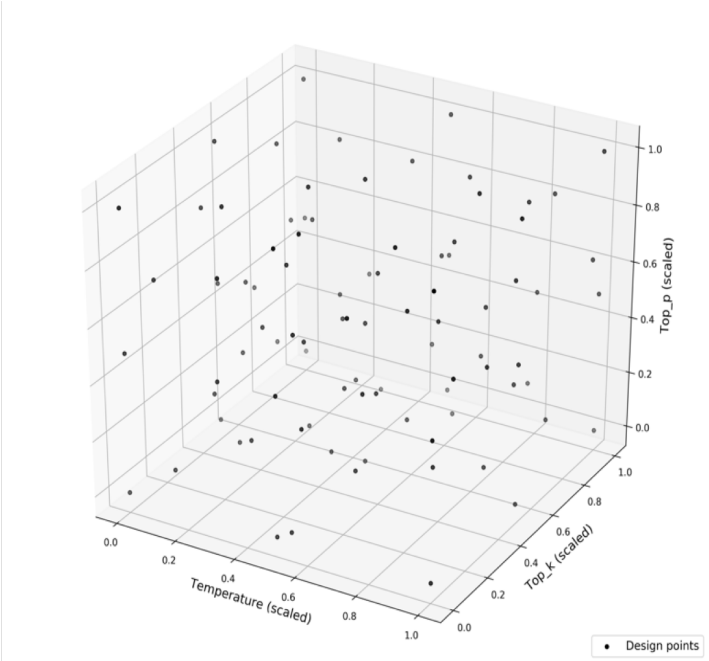

Figure 8. Design space of the GPT-2 model's diversity parameters.

3.5. Surrogate Modeling

The multi-layer perceptron (MLP), which is one of the most representative non-linear regression methods, is deployed as a surrogate model with respect to the GPT-2 model's diversity parameters. The MLP model used for this research entails the following fully-connected layers: (1) an input layer to receive diversity parameter values, (2) an output layer that makes a prediction in terms of the score function illustrated in Table 3, and (3) two hidden layers that are the true computational engine for the regression. Figure 9 shows a diagram of the MLP model structure used for this research.

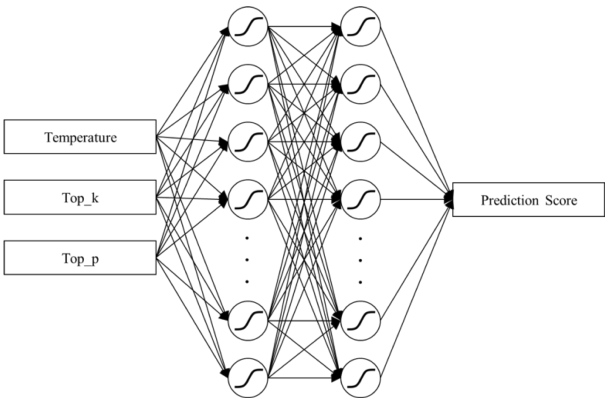

Figure 9. Diagram of the MLP model structure.

Table 3. Score metric for the MLP-based surrogate modeling.

Case	Score
An option with the highest probability matches the actual code	100
There is an option among the possible candidates, which matches to the actual code	50
There is no available option that matches the actual code	1

To evaluate the accuracy of the MLP-based surrogate model, the model representation error (MRE) is calculated with respect to additional random DoE cases. As a result, R-square, which describes how well the model predictions adhere to reality, is equal to 0.98 and root mean square error (RMSE), which describes how to spread out the residuals, is approximately 3.12.

3.6. Monte Carlo Simulation

We utilize the Monte Carlo simulation (MCS) technique to see the trend of resulting outcomes generated from the MLP-based surrogate model. Uniform distribution with min/max values is used for the GPT-2 model's diversity parameters. The MCS is then performed with 1,000,000 sample points generated by the uniform distribution of the GPT-2 model's diversity parameters, which are eventually incorporated into the MLP-based surrogate model to yield statistical distributions. Figure 10 notionally depicts the MCS process flow diagram especially used in this research (i.e., input and output mapping).

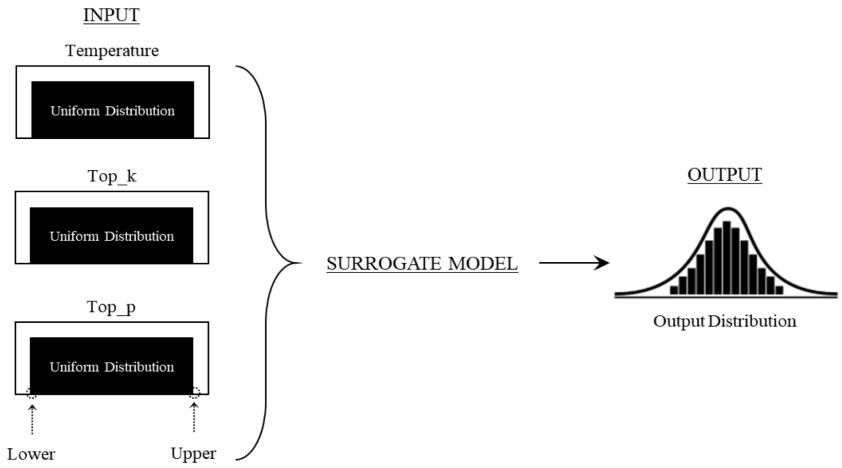

Figure 10. Notional sketch of the MCS process flow diagram.

4. Results and Discussion

4.1. Sensitivity Analysis

Sensitivity analysis with respect to the GPT-2 diversity parameters is performed to enhance the level of understanding of the relationship between prediction accuracy and the diversity parameters. The GPT-2 model has three different diversity parameters implemented in the sampling process.

The Boltzmann temperature is one of the GPT-2 model diversity parameters that control randomness in the sampling process. Figure 11 shows the MCS results with respect to the Boltzmann temperature. Lower and upper bounds are specified with 0.1 and 0.9, respectively. A black dot represents one experiment case generated by the MLP-based surrogate model with three different input variables randomly sampled from the uniform distributions with respect to the GPT-2 model's diversity parameters. As can be seen from Figure 11, it seems that decreasing Boltzmann temperature (i.e., x-axis) keeps the model to generate a high prediction score (i.e., y-axis), while increasing Boltzmann temperature causes the model to tend to frequently have a low prediction score. Based on these observations, one may claim that the model with lower Boltzmann temperature value, named deterministic design in this paper, would be the best option to predict the most likely

next code element as the deterministic design strives to minimize the degree of surprise in model output. However, it is too early to draw such a conclusion, because the model with a higher Boltzmann temperature value, named probabilistic design in this paper, would provide a list of reasonable next code elements in which one could select it manually to increase prediction accuracy. Details will be discussed in the section of model evaluation.

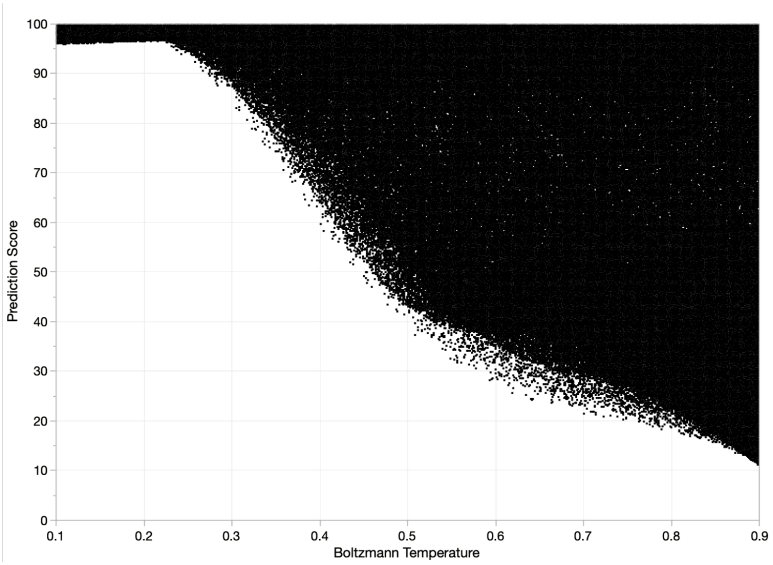

Figure 11. MCS results for diversity parameter 1 (i.e., Boltzmann temperature).

The *Top_k* is another GPT-2 model diversity parameter that controls the number of sampling words to be considered. For example, the most likely word is only considered if the *Top_k* is equal to onem thus resulting in deterministic design. The deterministic design can successfully eliminate rather weird candidates; however, one may claim that better results would be achieved if the algorithm considers sampling words more than one. Figure 12 shows the MCS results with respect to the *Top_k* parameter. Lower and upper bounds are specified with 40 and 100, respectively. As can be seen, it appears that the *Top_k* parameter value does not have a significant impact on the prediction score, indicating that the *Top_k* parameter may not entirely contribute to the model output diversity.

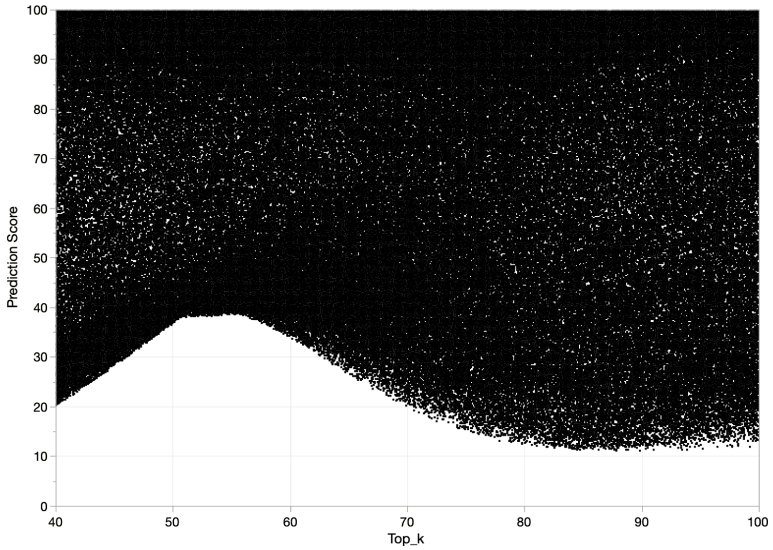

Figure 12. MCS results for diversity parameter 2 (i.e., *Top_k*).

The *Top_p*, one of the GPT-2 model diversity parameters, considers sampling words from the largest possible set of words whose cumulative probability exceeds a user-defined number. Instead of sampling only from the most likely *K* words, the *Top_p* parameter provides an option that dynamically controls the size of the set of sampling words to be considered. For example, if the *Top_p* is equal to 0.9, the algorithm computes cumulative probability distribution (CDF) and cuts off the words as soon as the CDF exceeds 90 percents. Figure 13 shows the MCS results with respect to the *Top_p* parameter. Lower and upper bounds are specified with 0.1 and 0.9, respectively. As the *Top_p* parameter value increases, it results in more randomness in terms of the prediction score. On the other hand, as the *Top_p* parameter value decreases, it leads to less randomness with regard to the prediction score. This implies that the model with lower *Top_p* parameter value, which is approximately 0.25 in this case, becomes deterministic and repetitive; while, the model with a higher *Top_p* parameter value becomes a probabilistic design that may relatively improve code suggestion quality compared to a deterministic design. Details about the difference between deterministic and probabilistic design will be discussed in the section of model evaluation.

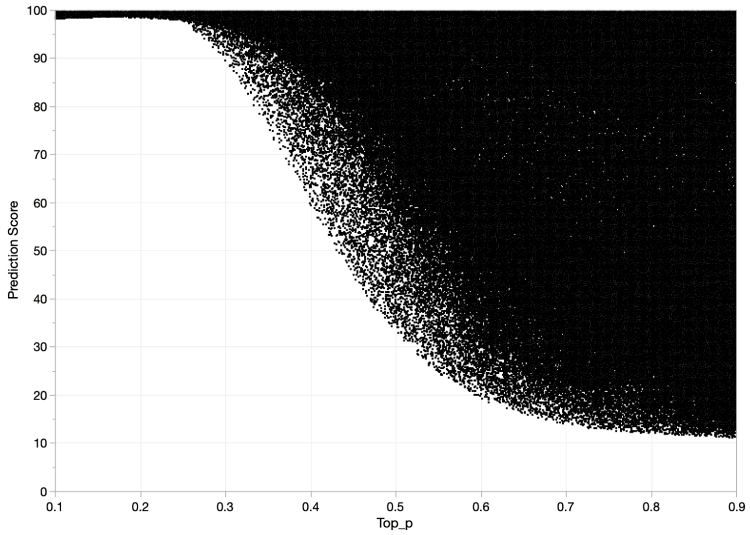

Figure 13. MCS results for diversity parameter 3 (i.e., *Top_p*).

4.2. Model Evaluation

The simplest way to evaluate the fine-tuned GPT-2 model is to allow the model to ramble on its own, which is called generating unconditional samples, but we are determined to use the option, called generating interactive conditional samples, for a model evaluation purpose, as it is easy to steer customized samples. The interactive conditional sample refers to generating samples based on a user-defined input code. For example, it generates the most likely next code element once the user provides an initial code. After users select the code element, the element is then added to the sequence of the input code. Then, a new sequence becomes the input for the next step. This process is repeated until it fills the rest of the sequence. In this paper, we use open-source SSD firmware codes, namely SimpleSSD [20], to evaluate the framework developed by this research, because Samsung Electronics SSD firmware source codes are strictly confidential. Figure 14 shows one sample code element [21] tested by the framework.

```
case POLICY_LEAST_RECENTLY_USED:
    evictFunction = [this](unit32_t setIdx, unit64_t & tick) -> unit32_t {
        unit32_t wayIdx = 0;
        unit64_t min = std::numeric_limits<unit64_t>::max();
```

Figure 14. Sample code element to be tested.

Based on the sensitivity analysis results, we specify the parameter values tabulated in Table 4 for the deterministic design. It should be noted that we specify 40 for the *Top_k* parameter (i.e., rule of thumb) [22], as the parameter does not affect randomness in the sampling process. Regarding the probabilistic design, we specify the maximum value (i.e., upper limit) for the GPT-2 diversity parameters except for the *Top_k* parameter.

Table 4. Diversity parameter values for deterministic design.

Diversity Parameter	Value
Boltzmann temperature	0.22
Top_k	40
Top_p	0.25

Figure 15 shows the results of predictions by the deterministic and probabilistic design for the sample code element from Figure 14. Incorrect code elements are underlined by solid straight lines. This result indicates that the probabilistic design is better than the deterministic design with respect to similarity, especially for the sample code element. Here, it must be noted that the probabilistic design is 100% correct as the users could select the correct code element after the framework suggest a list of the most likely next code elements. Furthermore, it is worth mentioning that the deterministic design is capable of predicting the correct next code elements in most cases; however, it sometimes produces the model output in some unexpected ways.

Sample code elements	Predictions by Deterministic design	Predictions by Probabilistic design

case POLICY_LEAST_RECENTLY_USED:
evictFunction = [this](uint32_t setIdx,
uint64_t & tick) -> uint32_t {
uint32_t wayIdx = 0;
uint64_t min =
std:numeric_limits<unit64_t>::max();
	case POLICY_LEAST_RECENTLY_USED:
	evictFunction = [this]<u>(uint64_t now, void
	*context) {</u>
	uint32_t wayIdx = 0;
	uint64_t min =
	std:numeric_limits<unit64_t>::max();
		case POLICY_LEAST_RECENTLY_USED:
		evictFunction = [this](uint32_t setIdx,
		uint64_t & tick) -> uint32_t {
		uint32_t wayIdx = 0;
		uint64_t min =
		std:numeric_limits<unit64_t>::max();

NOTE:

- ✓ *The probabilistic design is 100% correct given that users manually select the correct option from a list of the next code elements*
- ✓ *Incorrect code elements are underlined by solid straight lines*

Figure 15. Sample code element predictions by deterministic and probabilistic design.

Figure 16 shows how deterministic and probabilistic design predicts the next code element differently based on the same user-defined input. As can be seen, the deterministic design generated the only one output that would be the most likely (i.e., 100% probability) next code element in the prediction process. Unfortunately, this prediction was incorrect for this particular case even though the deterministic design was typically able to predict correctly in most cases. In other words, the reason why the deterministic design would not be able to predict the code element correctly for this case was that it would automatically select the option with the highest prediction probability. On the other hand, given the same input code element, the probabilistic design generated a list of the most likely next code elements with a certain probability. This approach enabled the users to select the option manually from the list that included the correct code element based on previous input. For example, although the option with the highest probability (e.g., [this](unit_64t) was incorrect for this particular case, the probabilistic design was able to increase prediction accuracy by allowing the users to select an option (e.g., [this](unit_32t) manually. Thus, there was no discrepancy with the probabilistic design given that the user could manually select the candidate from the list of options (e.g., choosing an option with 20% probability instead of an option with 80% probability).

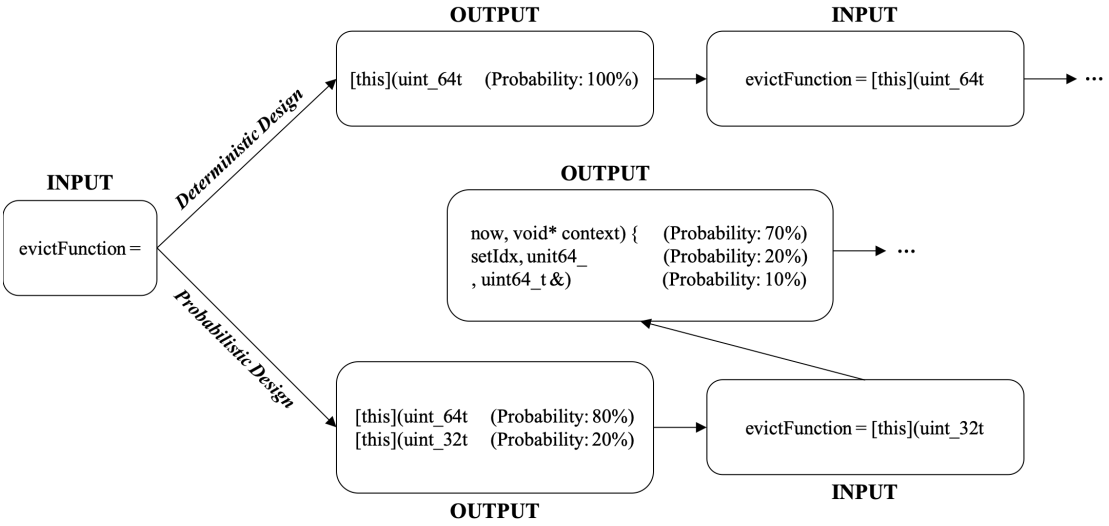

Figure 16. Deterministic design vs. Probabilistic design.

This example shows the impact of the GPT-2 model diversity parameters on the model output prediction accuracy. In addition to this particular example, Figures 17–19 support the argument that the deterministic design sometimes generates the model output in some unexpected ways, while there is no discrepancy with the probabilistic design, given that the user could manually select an option from a list of candidates.

Sample code (*ftl.cc*) elements	**Predictions by Deterministic design**	**Predictions by Probabilistic design**

switch (conf.readInt(CONFIG_FTL, FTL_MAPPING_MODE)) {
case PAGE_MAPPING:
pFTL = new PageMapping(conf, param, pPAL, pDRAM);
	switch (conf.readFloat(CONFIG_FTL, FTL_OVERPROVISION_RATIO));
	case PAGE_MAPPING:
	pFTL -> initialize();
		switch (conf.readInt(CONFIG_FTL, FTL_MAPPING_MODE)) {
		case PAGE_MAPPING:
		pFTL = new PageMapping(conf, param, pPAL, pDRAM);

void FTL::read(Request &req, unit64_t &tick) {
debugprint(LOG_FTL, "READ | LPN %" PRIu64, req.lpn);
pFTL->read(req, tick);
tick += applyLatency(CPU::FTL, CPU::READ);
	void FTL::trim(Request &req, unit64_t &tick) {
	debugprint(LOG_FTL, "READ | LPN %" PRIu64, req.lpn);
	pFTL->initialize();
	tick += applyLatency(CPU::FTL, CPU::READ);
		void FTL::read(Request &req, unit64_t &tick) {
		debugprint(LOG_FTL, "READ | LPN %" PRIu64, req.lpn);
		pFTL->read(req, tick);
		tick += applyLatency(CPU::FTL, CPU::READ);

void FTL::resetStatValues() {
pFTL->resetStatValues();
pPAL->resetStatValues();
	void FTL::resetStatValues() {
	pFTL->resetStatValues();
	pPAL->resetStatValues();
		void FTL::resetStatValues() {
		pFTL->resetStatValues();
		pPAL->resetStatValues();

NOTE:

- ✓ *The probabilistic design is 100% correct given that users manually select the correct option from a list of the next code elements*
- ✓ *Incorrect code elements are underlined by solid straight lines*

Figure 17. Sample code (*ftl.cc*) element predictions by deterministic and probabilistic design.

The main gist of these case studies is as follows: First, the deterministic design is recommended for those who would like to reduce latency time for the model output prediction, but users should recognize that the prediction accuracy may not be guaranteed. Second, the probabilistic design is recommended for those who want to guarantee the model prediction accuracy; however, users may have to address potential issues related to high computational costs about inference. Regarding the potential issues, we recognize the trend that transformer-based language models are going to get bigger (e.g., GPT-3 model), so they may require a lot of computing power and memory to run them, which potentially leads to challenging to run on a personal computer with reasonable latency time. In this case, it is imperative to implement the code auto-completion engine developed by this

research into a cloud computing resource. The good news is that Samsung Electronics already has an environment internally that includes many cloud computing systems. With the internal systems, the company does not have to account for security concerns. However, one potential issue is that the cloud systems may experience a bottleneck because of an increase in traffic from user requests. This paper does not address the potential limitation about how to operate the framework developed by this research with the cloud computing systems, but focus on proving the concept of the machine learning-based code auto-completion functionality.

Sample code (*hil.cc*) elements	**Predictions by Deterministic design**	**Predictions by Probabilistic design**

void HIL::read(Request &req) {
DMAFunction doRead = [this](uint64_t beginAt, void *context) {
auto pReq = (Request *)context;
uint64_t tick = beginAt;
	void HIL::read(Request &req) {
	DMAFunction doFlush =[this](uint64_t tick, void *context) {
	auto pReq = (Request *)context;
	uint64_t tick = beginAt;
		void HIL::read(Request &req) {
		DMAFunction doRead = [this](uint64_t beginAt, void *context) {
		auto pReq = (Request *)context;
		uint64_t tick = beginAt;

void HIL::getLPNInfo(uint64_t &totalLogicalPages, uint32_t &logicalPageSize) { pICL->getLPNInfo(totalLogicalPages, logicalPageSize);	void HIL::getUsedPageCount(uint64_t lcaBegin, uint64_t, lcaEnd) { pICL->getLPNInfo(totalLogicalPages, logicalPageSize);	void HIL::getLPNInfo(uint64_t &totalLogicalPages, uint32_t &logicalPageSize) { pICL->getLPNInfo(totalLogicalPages, logicalPageSize);

void HIL::updateCompletion() {
if (completionQueue.size() > 0) {
if (lastScheduled != completionQueue.top().finishedAt) {
lastScheduled = completionQueue.top().finishedAt;
	void HIL::updateCompletion() {
	if (completionQueue.size() > 0) {
	if (lastScheduled != completionQueue.top().finishedAt) {
	lastScheduled = completionQueue.top().finishedAt;
		void HIL::updateCompletion() {
		if (completionQueue.size() > 0) {
		if (lastScheduled != completionQueue.top().finishedAt) {
		lastScheduled = completionQueue.top().finishedAt;

NOTE:

- ✓ *The probabilistic design is 100% correct given that users manually select the correct option from a list of the next code elements*
- ✓ *Incorrect code elements are underlined by solid straight lines*

Figure 18. Sample code (*hil.cc*) element predictions by deterministic and probabilistic design.

Sample code (*fifo.cc*) elements	**Predictions by Deterministic design**	**Predictions by Probabilistic design**

void FIFO::transferRead() {
auto iter = readQueue.waitQueue.begin();
bool smallerThanUnit = false;
uint64_t size = calcSize(iter->size, smallerThanUnit);
	void FIFO::insertRead() {
	auto iter = readCompletion.begin();
	bool unused;
	uint64_t size = calcSize(iter->size, smallerThanUnit);
		void FIFO::transferRead() {
		auto iter = readQueue.waitQueue.begin();
		bool smallerThanUnit = false;
		uint64_t size = calcSize(iter->size, smallerThanUnit);

copy.last = true;
copy.addr = iter->addr + param.transferUnit;
copy.size = iter->size – param.transferUnit;
copy.buffer = iter->buffer ? iter->buffer + param.transferUnit : nullptr;
	copy.last = true;
	copy.addr = iter->size – param.transferUnit;
	copy.size = iter->size – param.transferUnit;
	copy.buffer = iter->buffer ? iter->buffer + param.transferUnit : nullptr;
		copy.last = true;
		copy.addr = iter->addr + param.transferUnit;
		copy.size = iter->size – param.transferUnit;
		copy.buffer = iter->buffer ? iter->buffer + param.transferUnit : nullptr;

void FIFO::insertReadDoneMerge(std::list<ReadEntry>::iterator comp) {
uint64_t now = getTick();
if (now >= comp->dmaEndAt + comp->latency) {
insertReadDoneNext();
	void FIFO::insertReadDoneNext() {
	uint64_t now = getTick();
	if (now >= iter.insertEndAt + letency) {
	insertReadDoneNext();
		void FIFO::insertReadDoneMerge(std::list<ReadEntry>::iterator comp) {
		uint64_t now = getTick();
		if (now >= comp->dmaEndAt + comp->latency) {
		insertReadDoneNext();

NOTE:

- ✓ *The probabilistic design is 100% correct given that users manually select the correct option from a list of the next code elements*
- ✓ *Incorrect code elements are underlined by solid straight lines*

Figure 19. Sample code (*fifo.cc*) element predictions by deterministic and probabilistic design.

5. Conclusions

In this research, we established a machine learning-based code auto-completion framework, especially for SSD firmware developers at Samsung Electronics. The hybrid approach that harnesses the synergy between machine learning techniques and advanced design methods was presented

to enhance the level of the understanding of the relationship between the GPT-2 model diversity parameters and prediction accuracy. The sensitivity analysis results showed that the probabilistic design outperformed the deterministic design with respect to the prediction accuracy as we observed a few cases of failure showing that the deterministic design generated output in some unexpected ways. It was found that there must be a balance between model prediction accuracy and prediction latency time given that users utilize the framework with either laptops or desktops. The accomplishment of this research can be implemented in any firmware development environment at a company as needed. In conclusion, it is expected that the framework developed by this research can save numerous hours of productivity by eliminating tedious parts of writing code and helping SSD firmware developers write code in a more efficient manner. Future research will extend the framework by implementing a new functionality accounting for potential issues related to the order of suggestions given that users may select accidentally the first entry (i.e., unwanted selections), which may not always be the correct option, among the recommended code elements.

Author Contributions: Conceptualization, J.K. and S.C.; Methodology, J.K.; Software, J.K. and S.C.; Validation, J.K.; Investigation, J.K. and K.L.; Writing—original draft preparation, J.K.; Writing—review and editing, K.L. and S.C.; All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by Kyungpook National University Research Fund, 2020.

Acknowledgments: This paper is an extension of a PhD summer internship project that was done at Samsung Electronics in 2020. We would like to thank Hankyu Lee for his feedback on this research. We would also like to thank Jinbaek Song for his support on the NVIDIA DGX-1 setup process.

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

References

1. Kim, J.; Briceno, S.; Justin, C.; Mavris, D. A Data-Driven Approach Using Machine Learning to Enable Real-Time Flight Path Planning. In Proceedings of the AIAA Aviation 2020 Forum, USA, 10 November 2020; p. 2873. Available online: https://arc.aiaa.org/doi/abs/10.2514/6.2020-2873 (accessed on 12 November 2020).
2. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv* **2018**, arXiv:1810.04805.
3. Radford, A.; Wu, J.; Child, R.; Luan, D.; Amodei, D.; Sutskever, I. Language models are unsupervised multitask learners. *OpenAI Blog* **2019**, *1*, 9.
4. Brown, T.B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language models are few-shot learners. *arXiv* **2020**, arXiv:2005.14165.
5. The illustrated GPT-2 (Visualizing Transformer Language Models. Available online: http://jalammar.github.io/illustrated-gpt2/ (accessed on 12 November 2020).
6. AllenNLP Language Modeling. Available online: https://demo.allennlp.org/next-token-lm?text=AllenNLP%20is (accessed on 12 November 2020).
7. Working with Content Assist. Available online: https://www.eclipse.org/pdt/help/html/working_with_code_assist.htm (accessed on 19 October 2020).
8. Das, S.; Shah, C. *Contextual Code Completion Using Machine Learning*; Technical Report; Stanford University: Stanford, CA, USA, 2015.
9. Proksch, S.; Lerch, J.; Mezini, M. Intelligent code completion with Bayesian networks. *ACM Trans. Softw. Eng. Methodol.* **2015**, *25*, 1–31.
10. Raychev, V.; Vechev, M.; Yahav, E. Code completion with statistical language models. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, Edinburgh, UK, 9–11 June 2014; pp. 419–428.
11. Neural Code Completion. Available online: https://openreview.net/pdf?id=rJbPBt9lg (accessed on 27 November 2020).
12. Karampatsis, R.M.; Babii, H.; Robbes, R.; Sutton, C.; Janes, A. Big Code!= Big Vocabulary: Open-Vocabulary Models for Source Code. *arXiv* **2020**, arXiv:2003.07914.

13. White, M.; Vendome, C.; Linares-Vásquez, M.; Poshyvanyk, D. Toward deep learning software repositories. In Proceedings of the 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, Florence, Italy, 16–17 May 2015; pp. 334–345.
14. Li, J.; Wang, Y.; Lyu, M.R.; King, I. Code completion with neural attention and pointer networks. *arXiv* **2017**, arXiv:1711.09573.
15. Svyatkovskiy, A.; Deng, S.K.; Fu, S.; Sundaresan, N. IntelliCode Compose: Code Generation Using Transformer. *arXiv* **2020**, arXiv:2005.08025.
16. Dong, L.; Yang, N.; Wang, W.; Wei, F.; Liu, X.; Wang, Y.; Gao, J.; Zhou, M.; Hon, H.W. Unified language model pre-training for natural language understanding and generation. *arXiv* **2019**, arXiv:1905.03197.
17. Autocompletion with Deep Learning. Available online: https://www.tabnine.com/blog/deep/ (accessed on 19 October 2020).
18. Better Language Models and Their Implications. Available online: https://openai.com/blog/better-language-models/ (accessed on 19 October 2020).
19. Kim, J.; Lim, D.; Monteiro, D.J.; Kirby, M.; Mavris, D. Multi-objective Optimization of Departure Procedures at Gimpo International Airport. *Int. J. Aeronaut. Space Sci.* **2018**, *19*, 534–541. [CrossRef]
20. Jung, M.; Zhang, J.; Abulila, A.; Kwon, M.; Shahidi, N.; Shalf, J.; Kim, N.S.; Kandemir, M. Simplessd: Modeling solid state drives for holistic system simulation. *IEEE Comput. Archit. Lett.* **2017**, *17*, 37–41. [CrossRef]
21. SimpleSSD Version 2.0 GitHub. Available online: https://github.com/SimpleSSD/SimpleSSD (accessed on 19 October 2020).
22. Beginner's Guide to Retrain GPT-2 (117M) to Generate Custom Text Content. Available online: https://medium.com/@ngwaifoong92/beginners-guide-to-retrain-gpt-2-117m-to-generate-custom-text-content-8bb5363d8b7f (accessed on 19 October 2020).