AI

MDPI

Article

Automated Source Code Generation and Auto-Completion Using Deep Learning: Comparing and Discussing Current Language Model-Related Approaches

Juan Cruz-Benito [1,*], Sanjay Vishwakarma [2,†], Francisco Martin-Fernandez [1] and Ismael Faro [1]

[1] IBM Quantum, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA; paco@ibm.com (F.M.-F.); ismael.faro1@ibm.com (I.F.)
[2] Electrical and Computer Engineering, Carnegie Mellon University, Mountain View, CA 94035, USA; svishwak@andrew.cmu.edu
* Correspondence: juan.cruz@ibm.com
† Intern at IBM Quantum at the time of writing this paper.

Abstract: In recent years, the use of deep learning in language models has gained much attention. Some research projects claim that they can generate text that can be interpreted as human writing, enabling new possibilities in many application areas. Among the different areas related to language processing, one of the most notable in applying this type of modeling is programming languages. For years, the machine learning community has been researching this software engineering area, pursuing goals like applying different approaches to auto-complete, generate, fix, or evaluate code programmed by humans. Considering the increasing popularity of the deep learning-enabled language models approach, we found a lack of empirical papers that compare different deep learning architectures to create and use language models based on programming code. This paper compares different neural network architectures like Average Stochastic Gradient Descent (ASGD) Weight-Dropped LSTMs (AWD-LSTMs), AWD-Quasi-Recurrent Neural Networks (QRNNs), and Transformer while using transfer learning and different forms of tokenization to see how they behave in building language models using a Python dataset for code generation and filling mask tasks. Considering the results, we discuss each approach's different strengths and weaknesses and what gaps we found to evaluate the language models or to apply them in a real programming context.

Keywords: deep learning; language model; source code; software engineering; natural language processing

Citation: Cruz-Benito, J.; Vishwakarma, S.; Martin-Fernandez, F.; Faro, I. Automated Source Code Generation and Auto-Completion Using Deep Learning: Comparing and Discussing Current Language Model-Related Approaches. *AI* **2021**, *2*, 1–16. https://doi.org/10.3390/ai2010001

Received: 3 November 2020
Accepted: 14 January 2021
Published: 16 January 2021

1. Introduction

We are digitally surrounded by computational Language Models (LMs) that guide us while writing to reduce user effort, suggest different options for words/sentences to enhance our style, or accurately fix our grammatical errors [1–3]. Many of the keys we press while typing on a keyboard act as a part of the inputs to compose new datasets for those models that shape how we communicate with others. Nevertheless, does this happen in the same way when we write code? Succinctly, the answer is "yes". According to some recent surveys found in the literature [4,5], the Natural Language Processing (NLP) subfield related to programming languages includes examples of LMs used in several tasks and contexts. For example, the authors of [6–8] used different techniques such as graph-based statistical LMs, probabilistic LMs, or Deep Learning (DL) LMs to suggest code to programmers similarly to auto-complete features in integrated development environments (IDEs). LMs were used to generate automated source code based on sample code inputs or pseudo-code, and how this generated code performs was evaluated [9–11]. Another exciting application of NLP to source code languages is the automatic translation between different languages. The work reported in [12,13] explored different supervised and

unsupervised approaches to migrate code between different programming languages to improve interoperability or port codebases written in obsolete or deprecated languages (such as COBOL or Python2). Another example found is the use of Bayesian networks, attention mechanisms, and pointer networks [14–16] to fill in a given code portion with missing code.

There is a more general understanding of natural languages' different characteristics in the broad NLP field. Since there exist many research fields related to human languages, there is a richer background on existing language characteristics. For example, there is much knowledge on aspects like the minimal representation units of a word in a specific language, the most used words of a language, or if a word is a neologism or not. Programming languages share some syntax similarities with spoken languages. However, they do not have the same restrictions in the sense of common words or neologisms [17–19] or other syntax restrictions and features such as punctuation, format, or style. Every programming language has indeed reserved words and symbols to denote different actions, resources, or syntax. However, there is an essential part of the source code that is only limited by the programmer's imagination, the existing conventions, or the guidelines for good practices. As [18] claims,

> [...] traditional language models limit the vocabulary to a fixed set of common words. For code, this strong assumption has been shown to have a significant negative effect on predictive performance [...]

In that paper, Karampatsis and Sutton [18] presented how segmenting words into subword units can improve source code modeling. Similarly, other researchers [20–22] dug into representing the source code vocabulary with a similar emphasis on modeling words using sub-word units and envisioning their importance when using Neural Networks (NNs). Nevertheless, how does that word segmentation affect the accuracy or the appropriateness of the code generated or auto-completed in some modern LMs using deep learning approaches? That kind of question raises the main objective of this paper: to discover what kinds of associations between different modern neural network architectures and tokenization models produce the best results when creating LMs to generate and auto-complete source code.

To pursue that goal, this research aims to conduct experiments combining different Deep Neural Network (DNN) architectures with different tokenization and pre-trained models over an existing Python dataset. Using that experimentation, we want to investigate the combinations that improve code generation and auto-completion tasks (for example, filling in the blanks) while checking the outcomes from those tasks using metrics like accuracy and human assessment.

The rest of the paper is as follows: Section 2 presents the different approaches followed during the research, the DNNs used, the software methods, and the data employed. Section 3 describes the results achieved during the research according to different metrics and tests, while Section 4 discusses these findings and the implications of the results as appropriate. Finally, Section 5 presents some conclusions.

2. Materials and Methods

We trained a set of deep neural networks using different architectures, tokenization techniques, and software libraries to develop this research work. In the following, we introduce the different materials and methods employed for that purpose.

2.1. Deep Neural Networks and Tokenization Models Used

Regarding the DNN architectures employed, we chose to use the following ones: Average Stochastic Gradient Descent (ASGD) Weight-Dropped LSTM (AWD-LSTM) [23], Quasi-Recurrent Neural Networks (QRNNs) [24], and Transformer [25]. These DNN architectures have reportedly obtained some state-of-the-art results [23,26–38] recently in the NLP field in several groundbreaking digital products (https://openai.com/blog/openai-api/, https://blog.google/products/search/search-language-understanding-bert/) and in some of

the most known datasets like the Penn Tree Bank [39], WikiText-2 and WikiText-103 [40], the One-Billion Word benchmark [41], or The Hutter Prize Wikipedia dataset (http://prize.hutter1.net/).

The AWD-LSTM is a variation of the famous Long Short-Term Memory (LSTM) architecture [42]. The LSTM is a type of Recurrent Neural Network (RNN) especially capable of processing and predicting sequences. That ability with sequences is the reason why LSTMs have been employed largely in LMs [21]. The AWD-LSTM includes several optimizations compared to the regular LSTM. Two of the most important ones are the use of Average Stochastic Gradient Descent (ASGD) and the weight dropout. The ASGD is used as the NN's optimizer to consider the previous weights (not only the current one) during each training iteration. The weight dropout introduces the dropout technique [43] to avoid overfitting, but with the characteristic of returning zero, not with a subset of activations between layers, like in traditional dropout, but with a random subset of weights.

The QRNN is another type of RNN that includes alternate convolutional and pooling layers in the architecture. This design makes the QRNN able to capture better long-term sequences and train much faster since the convolutional layers allow the computation of intermediate representations from the input in parallel. They can be up to 16 times faster at training and test time than LSTMs while having better predictive accuracy than stacked LSTMs of the same hidden size. We use a modified QRNN (AWD-QRNN) to include the same ASGD and weight dropout modifications to improve its stability and optimize its capabilities, as for the AWD-LSTM.

We utilize AWD-LSTM and AWD-QRNN to produce LMs capable of solving the task of generating source code based on the input as in the literature [23,26–30,34,34–36].

Transformer is probably the most popular current DNN architecture in NLP due to its performance and recent state-of-the-art results in many tasks. It is an encoder-decoder architecture in which each layer uses attention mechanisms. This use of (self-)attention mechanisms makes Transformer able to model the relationships between all words in a sentence regardless of their respective position. That implies a significant improvement over RNNs, enabling much more parallelization of data processing and unblocking the training over more massive datasets. The excellent results of the Transformer architecture empowered the NLP community to create new state-of-the-art transformer-based models [44] like those used in the current research: GPT-2 [45], BERT [38], and RoBERTa [46].

We chose to use GPT-2 since it is a causal transformer (unidirectional) that can predict the next token in a sequence. Therefore, it can generate source code based on the input, allowing us to compare the results with the AWD-LSTM and AWD-QRNN experiments. Regarding BERT and RoBERTa, we used them to study how a masked modeling approach can auto-complete the source code. In that case, we did not use them for text generation, as in the other experiments, since BERT and RoBERTa are not designed for text generation. However, they can generate text (more diverse, but slightly worse in quality) [47].

Considering the tokenization techniques, for every AWD-LSTM and AWD-QRNN, we chose the following types of tokens: word, unigram, char, and Byte-Pair Encoding (BPE) [48]—albeit, some studies showed that BPE is suboptimal for pre-training [49]. For the Transformer models, we used the default ones from the pre-defined models: the word piece method [50] for BERT and BPE over raw bytes instead of Unicode characters for GPT-2 and RoBERTa. The different techniques were selected because they produce different token granularities that can enrich our experimentation: full words, sub-words of specific sizes, character-sized tokens, or byte pairs. Furthermore, they enable us to compare the tokenization between the different types of models and tasks to solve.

2.2. Experimentation Details

The dataset used for the experimentation is the Python dataset included in the "GitHub CodeSearchNet Challenge dataset" [51]. It includes 2 million (comment, Python code) pairs from open-source libraries. As observed during the dataset preparation for our experiments, there are about 11 million Python code sentences. The reason for choosing

this dataset is that it has already been used in previous research related to NLP and source code. The full dataset includes several languages: Go, Java, JavaScript, PHP, Python, and Ruby. We chose to use only the Python part of the dataset because it enables us to compare the existing literature, which uses the Python language more than other programming languages. The software libraries and packages used primarily during the research were the following: FastAI [52], Google SentencePiece [53], and Hugging Face's Transformers [54]. The preprocessing applied to the dataset included removing most of the code comments and auto-formatting the code according to the PEP-8 Python style guide using the autopep8 (https://pypi.org/project/autopep8/) package. Regarding the AWD-LSTM networks, we used the FastAI-provided base models pre-trained using the Wikitext-103 dataset [40]. There are no default pre-trained models in FastAI's AWD-QRNN version of those networks, so we trained them from scratch. Regarding the Transformer architectures, we used three standard pre-trained models as a basis: GPT-2, BERT, and RoBERTa. In each case, the exact pre-trained model used were gpt2, bert-base-cased, and roberta-base. These pre-trained models are available from Hugging Face's model (https://huggingface.co/models).

As the reader could infer from the previous explanations about using pre-trained versions, we followed a transfer learning approach similar to other researchers in existing literature [35,36,55,56]. We employed the pre-trained models on English texts to later fine-tune the models for the selected tasks using the GitHub CodeSearchNet dataset. The deep neural network-related source code was coded using the FastAI library (Versions 1.0.61 and 2 dev 0.0.21). To apply the different tokenization techniques to the AWD-LSTMs and QRNNs, we replaced the default Spacy tokenizer [57] with Google SentencePiece [53], following a similar approach to [58]. In the development of the Transformer architectures to see how they perform in filling in the blanks and generating texts, we used Hugging Face's Transformer library combined with FastAI v2 (following the FastAI's example https://docs.fast.ai/tutorial.transformers), as included on the code repository that supports this paper. To train the neural networks, we used some techniques that are worth mentioning (all the details are in the code repository). To find the most appropriate learning rate to use automatically, we used the function *lr_find* provided by FastAI following the proposal of [59]. This function trains the DNN over the dataset for a few iterations while varying the learning rates from very low to very high at the beginning of each mini-batch of data to find which is the optimal one regarding the error (loss) metrics until the DNN diverges. To pursue a faster convergence, we scheduled the learning rate as described in [60] using the one cycle policy (*fit_one_cycle*) in FastAI. Considering the transfer learning technique used, we trained the first "one cycle" on the top of the existing pre-trained model to later unfreeze all the model layers and perform a more extended training (10–30 epochs) to improve the results. Regarding other training details, in general, we used the default parameters from FastAI, except for a fixed multiplier to control all the dropouts (*drop_mult*) in the AWD-LSTMs and AWD-QRNNs set to 0.3 because of some heuristics discovered during testing. Furthermore, we decided to train similar architectures using a fixed number of epochs to make the models comparable. For the AWD-LSTM and AWD-QRNN, we used 30 epochs for fine-tuning because we found during the experimentation that the most remarkable improvement for every model produced occurs during that range of iterations. Similarly, we fine-tuned the transformers for ten epochs since we did not find a significant improvement after that. For more information about the training setup and software details, please refer to the repository that supports this paper and the FastAI documentation.

Finally, the hardware used to run the different software and neural network training was a computer running Ubuntu Linux 18.04 LTS Bionic Beaver (64 bits). It has two Nvidia Tesla V100 GPU x16 gigabytes of RAM (Nvidia CUDA Version 10.1), a CPU with 16 cores Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz, 120 gigabytes of RAM, and 120 gigabytes for the primary disk (HDD).

All the supporting materials and software details related to this paper are publicly available in a GitHub repository [61]. The NN models produced are under the Zenodo record [62].

3. Results

This section presents the results achieved after the full training of the selected DNN architectures with the different tokenization models.

As outlined in the previous section, we trained the AWD-LSTM and AWD-QRNN DNN architectures using different tokenization model—word, unigram, BPE, and char—and Transformer using three different base models (GPT-2, BERT, and RoBERTa). We trained every AWD-LSTM and AWD-QRNN using one epoch to fit the model's head and fine-tuned for 30 epochs. Meanwhile, the Transformer networks were trained equally for one epoch to fit the head and fine-tune the models for ten epochs.

We followed a two way strategy to evaluate the NNs trained: using the NN training metrics and human evaluation of the models' output. The metrics used are some of the most common in the literature: accuracy for the validation set and loss for the training and validation sets. They help researchers understand how the NN acts over time, how the model is fitted to the dataset, and the performance and error scores while using training and validation datasets. In this case, the accuracy is the score concerning the LM's ability to predict the next word of filling in the missing ones accurately given a sequence of words from the validation set. The loss metrics reports the error after applying the DNN to the training or validation set, respectively. Every implementation detail related to the DNNs and the metrics is available in the GitHub repository [61]. Apart from those metrics, we assessed the models' quality by applying them in the proposed tasks—generate text and auto-complete—and observing how they performed.

3.1. Training Results

Table 1 displays the final metrics for the different NNs at the end of the training. Similarly, Figures 1 and 2 show the evolution of each model's accuracy during the training. Figures 3–6 show the *training_loss* and *validation_loss* evolution along the training epochs.

Table 1. Results after full training of each NN architecture. AWD, Average Stochastic Gradient Descent Weight-Dropped; BPE, Byte-Pair Encoding; QRNN, Quasi-Recurrent Neural Network.

DNN Architecture	Epochs	Accuracy	Train Loss	Validation Loss	Pre-Trained?
AWD-LSTM *word*	31	0.494893	2.308937	2.341698	Yes
AWD-LSTM *unigram*	31	0.557226	1.639875	1.826841	Yes
AWD-LSTM *BPE*	31	0.580373	1.561393	1.703536	Yes
AWD-LSTM *char*	31	0.779633	0.757956	0.742808	Yes
AWD-QRNN *word*	31	0.515747	1.972508	2.144126	No
AWD-QRNN *unigram*	31	0.539951	1.790150	1.894901	No
AWD-QRNN *BPE*	31	0.538290	1.824709	1.896698	No
AWD-QRNN *char*	31	0.736358	0.944526	0.897850	No
GPT-2	11	0.743738	1.407818	1.268246	Yes
BERT	11	0.999238	0.014755	0.004155	Yes
RoBERTa	11	0.999468	0.010569	0.002920	Yes

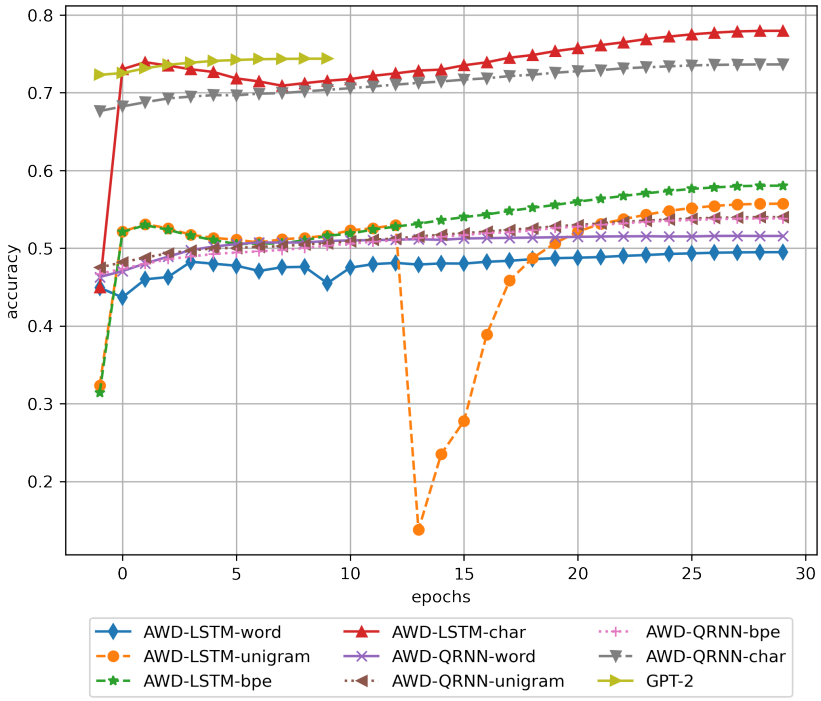

Figure 1. Evolution of the accuracy of neural networks devoted to source code generation during the training epochs.

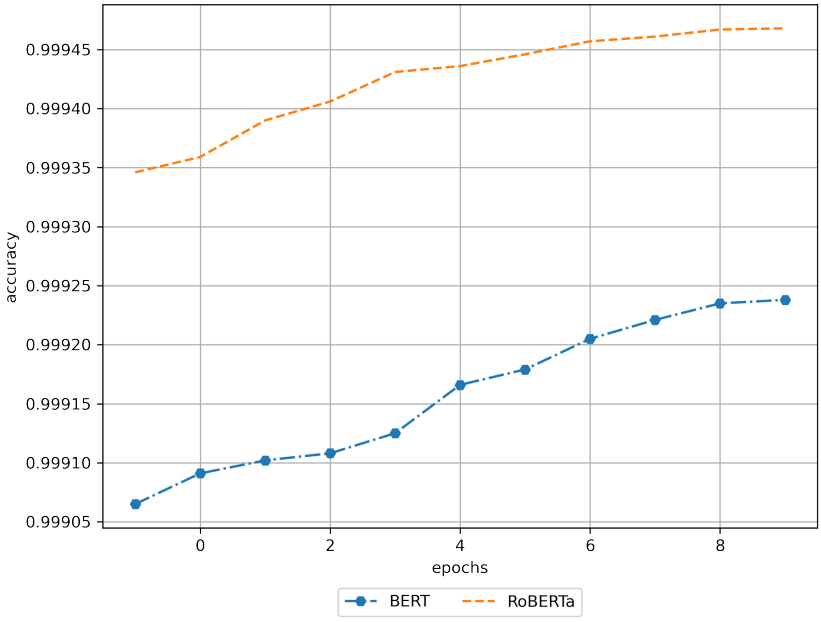

Figure 2. Evolution of the accuracy of neural networks devoted to filling in the blanks during the training epochs.

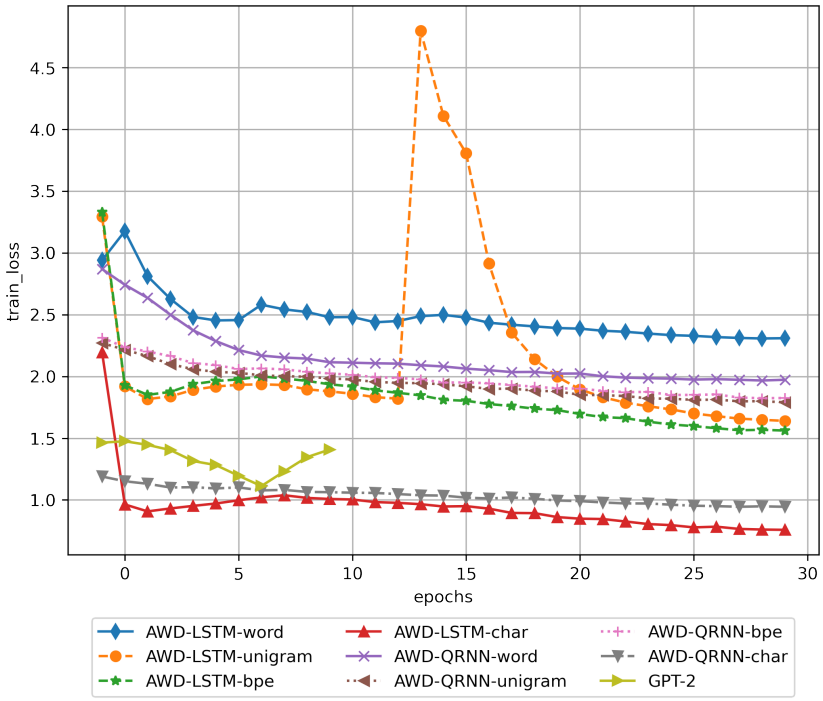

Figure 3. Evolution of the *training_loss* of DNNs devoted to generating source code during the training epochs.

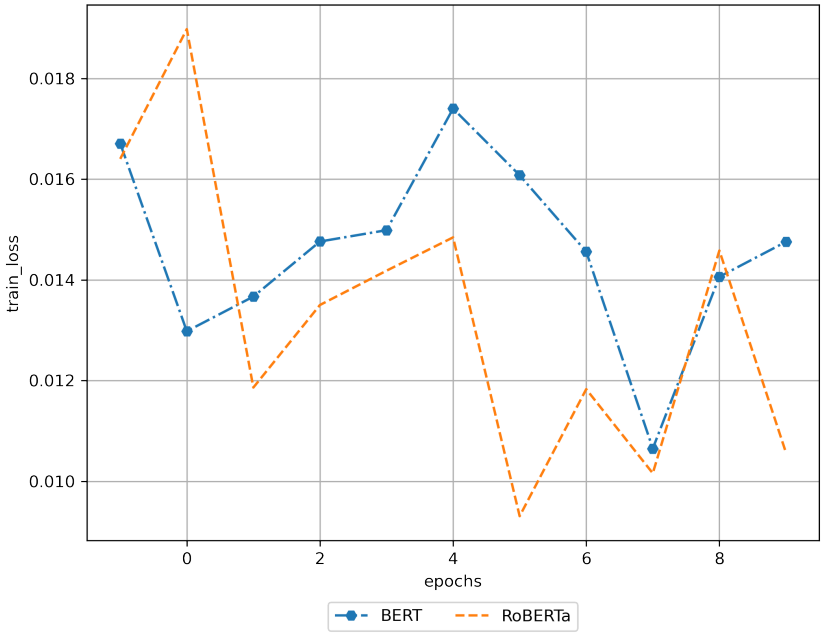

Figure 4. Evolution of the *training_loss* of neural networks devoted to filling in the blanks during the training epochs.

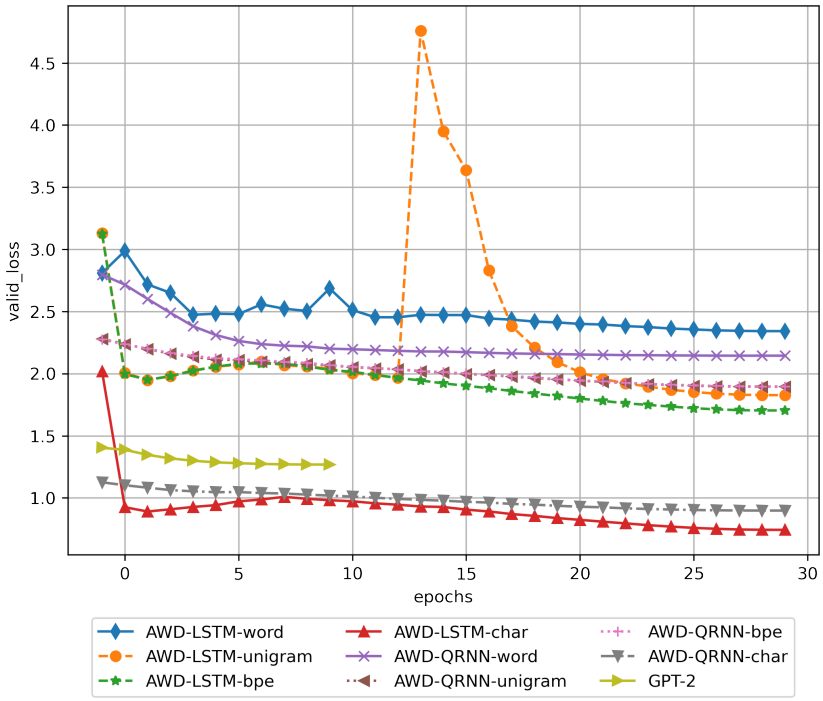

Figure 5. Evolution of the *validation_loss* of DNNs devoted to generating source code during the training epochs.

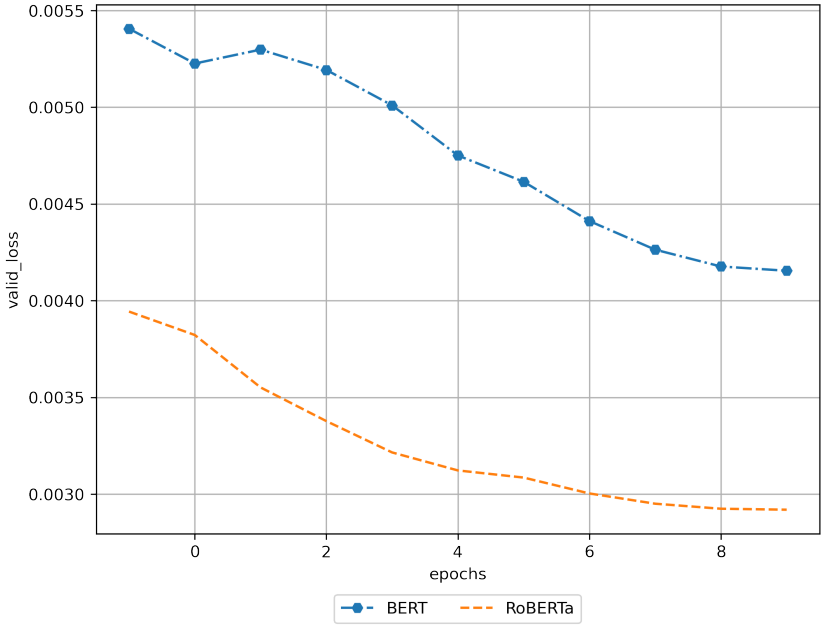

Figure 6. Evolution of the *validation_loss* of neural networks devoted to filling in the blanks during the training epochs.

On the one hand, according to the results displayed in Table 1 and Figure 1, for neural networks intended for automated source code generation—AWD-LSTM, AWD-QRNN, and Transformer GPT-2—the overall NN-tokenization model combination that performed better in the case of accuracy metrics was the AWD-LSTM with char tokenization (accuracy 0.779633). The second one was the GPT-2 transformer model—BPE over raw bytes tokenization—(0.743738) and the third one the AWD-QRNN with char tokenization (0.736358). Related to the AWD-LSTM and AWD-QRNN architectures' combination with other tokenization techniques, we obtained poor results on accuracy: between 0.494893 and

0.580373. On the other hand, according to the results shown in Table 1 and Figure 2, both models (BERT and RoBERTa) had excellent accuracy results in the Transformer models intended for auto-completion, 0.999238 and 0.999468, respectively.

Regarding how the pre-training and transfer learning affected the results, the two top results regarding the accuracy were the pre-trained models in the English language (0.779633 and 0.743738), yet the third best result was from a non-pre-trained network (0.736358). Comparing the similar networks, the average (mean) accuracy of the AWD-LSTM pre-trained versions was 0.603031 (standard deviation (std) of 0.123144), while the average accuracy of the AWD-QRNN non-pre-trained versions was 0.582587 (std of 0.103107). The only combination NN-tokenization model that worked worse when it was pre-trained was the one with the word tokenization.

Regarding the observed losses, it is worth commenting that the AWD-LSTM *char*, AWD-QRNN *char*, and the three transformer models (GPT-2, BERT, RoBERTa) could be trained for more epochs or with a higher learning rate. The model may have been underfitting since the training loss was higher than the validation loss (Table 1, Figures 3–6).

To put the accuracy achieved during the experimentation into context, we compare the results with the existing literature. The papers [10,15,63] presented models trained to generate Python code that were considered the state-of-the-art when they were published. Our three best models trained to generate source code outperformed the approaches based on the following architectures: vanilla LSTM (accuracy: 67.3%) [15], attention LSTM (69.8%) [15], pointer mixture network (70.1%) [15], or probabilistic model (69.2%) [63]. Our models performed worse than the other approaches based on the Abstract Syntax Tree (AST) instead of using the textual code: AST-LSTM (accuracy 90.3%) [10] or AST-MLP (90.1%) [10]. Therefore, considering this, our results are good in terms of accuracy compared to the existing peer-reviewed literature. As a side note, we did not find models available publicly to compare with, neither in these, nor in other peer-reviewed papers. For the models devoted to auto-complete code (BERT and RoBERTa), as of now, we did not find peer-reviewed papers trying to solve this task using Transformer architectures.

3.2. Results of Using the Trained DNN Models

In the following, we introduce some examples of how the different neural networks generated source code or auto-completed it. These examples are used to assess the model's output quality, avoiding relying only on the NN metrics. This validation using more methods than the simple metrics is common in the AI and NLP fields. Many papers employed human evaluation or specialized metrics to better grasp the model's performance and capture any other particular details or quality indicators for the related task proposed. In this case, we show only those with the best accuracy results. To generate the source code, we show the output from AWD-LSTM *char*, AWD-QRNN *char*, and GPT-2. For auto-completion, we show the results from BERT and RoBERTa. For the source code generation, every model faced the same challenges [61]. These are the results when any of them tried to generate five different sentences with the input *from pandas import*:

AWD-LSTM *char*

```
"from pandas import defaults as _rp pandas ="
"from pandas import DataFrameEncoder if self"
"from pandas import OrderedDict else: passed"
"from pandas import sequence if type(seq1) i"
"from pandas import conditional_insertion as"
```

AWD-QRNN *char*

```
"from pandas import webbrowser self.login fr"
"from pandas import __http_args or [ 'Pipeli"
"from pandas import iterations is not None p"
"from pandas import service params to servic"
"from pandas import difference if not isinst"
```

GPT-2

```
"from pandas import time, np\n
"
"from pandas import pandas from time.time.datetime import Date\n
with n"
"from pandas import gtk, os\n
from pandas_utils import pandas_utils\n
import pylint"
"from pandas import wcpy\n
import cpy_context as cpy_context\n
"
"from pandas import gkpy\n
"""\n
... pass\n
kwargs = cg"
```

To assess the generation, we did not focus on the semantics of the imports used or whether they were part of the Pandas library or not, but on the language's correctness. In general, from a Python language perspective, the outputs from GPT-2 were better. The outputs included line breaks, indentation, and fair use of multiple inputs in one sentence (except in one of the outputs). The AWD-LSTM and AWD-QRNN failed to auto-generate an import sentence appropriately, or at least they failed regarding the usual manner used by regular users. As a final comment on this, the other models trained failed on similar issues, and they did not get enough semantic context related to the Pandas library.

Similarly, concerning the source code auto-completion, both BERT and RoBERTa tried to auto-fill the *mask* token in the sentence *from pandas import [MASK]*. These are the results:

BERT

```
[{'sequence': '[CLS] from pandas import [SEP] [SEP]',
'score': 0.9969683289527893,
'token': 102,
'token_str': '[SEP]'},
{'sequence': '[CLS] from pandas import [CLS] [SEP]',
'score': 0.0010887219104915857,
'token': 101,
'token_str': '[CLS]'},
{'sequence': '[CLS] from pandas import. [SEP]',
'score': 0.0004200416151434183,
'token': 119,
'token_str': '.'},
{'sequence': '[CLS] from pandas import ; [SEP]',
'score': 0.00027348980074748397,
'token': 132,
'token_str': ';'},
{'sequence': '[CLS] from pandas import def [SEP]',
'score': 8.858884393703192e-05,
'token': 19353,
'token_str': 'def'}]
```

RoBERTa

```
[{'sequence': '<s>from pandas import\n</s>',
'score': 0.6224209666252136,
'token': 50118,
'token_str': 'Ċ'},
{'sequence': '<s>from pandas import.</s>',
'score': 0.22222988307476044,
'token': 4,
'token_str': '.'},
{'sequence': '<s>from pandas import </s>',
'score': 0.038354743272066116,
'token': 1437,
'token_str': 'Ġ'},
{'sequence': '<s>from pandas import\n\n</s>',
'score': 0.028566861525177956,
'token': 50140,
'token_str': 'ĊĊ'},
{'sequence': '<s>from pandas import.</s>',
'score': 0.021909384056925774,
'token': 479,
'token_str': 'Ġ.'}]
```

For auto-completion, we observed that almost no auto-completion was right. The more accurate sentences from the Python language point of view were the ones in which

the mask token was replaced by a white space or by a dot. Nevertheless, they were not correct, but closer to being right compared to the other ones. One interesting thing is that BERT assigns a very high score to a predicted mask, which is not correct, and shallow scores to the other possible solutions (also incorrect). In the case of RoBERTa, it gives lower scores to all the solutions, yet also fails on the correctness: the second sentence predicted (score 0.222) can be closer to being right compared to the first one (score 0.622).

4. Discussion

Considering the results obtained, one could convincingly assert that the tokenization model used profoundly affects the results when generating automated source code. Although that may be accurate, we must discuss it carefully.

4.1. Discussing the Outcomes from the Resulting Models

First, our overall results are consistent with the existing literature [18–22]. Sub-word tokenization works better in the case of modeling source code, as [18,19] stated. Every result obtained is consistent in that sense. Even more, as [22] envisioned, char tokenization probably should the best option to try by default when dealing with LMs and source code. Furthermore, according to the results achieved, models such as GPT-2—using a tokenization model based on BPE over raw bytes—can outperform LSTM/QRNN models like those we tested to grasp better the internal aspects of a programming language. As showcased during the results, even if GPT-2 was not the best model in terms of accuracy, it gave better code outputs than the other ones selected for the comparison.

As future work, it would be great to check if the better textual output in the case of GPT-2 is because of (a) it being a much bigger and better pre-trained model (163,037,184 parameters against 36,491 for AWD-LSTM and AWD-QRNN models), (b) it being related to the dataset's size or quality, (c) if this is related to both causes, or (d) if it this related to other issues.

Continuing with the comments about the accuracy, one may note that the textual outputs generated by the AWD-LSTM *char*, AWD-QRNN *char*, and GPT-2 could be polished to be more accurate. The final training loss is higher than the validation loss for the three selected DNN architectures, which can be a sign of underfitting. We found the same issue (*train_loss* « *valid_loss*) for the BERT- and RoBERTa-based models. While the purpose of this paper is not to produce state-of-the-art results per se, we continued the training for over five more epochs to verify it. The improvement obtained from extending the training for the best approaches was residual in general, so we decided to report the results for 1 + 30 epochs for the AWD-LSTMs and AWD-QRNNs and 1 + 10 epochs for Transformer.

4.2. Examining the Effect of Pre-Training and Transfer Learning

Regarding pre-training and transfer learning, every pre-trained model (on English-based datasets) obtained better accuracy than its non-pre-trained counterpart, except for the word tokenization models. This seems to be strongly related to the statements we introduced at the beginning of this paper, citing [17,18] about the source code not having the same restrictions in the sense of common words or neologisms. In this sense, the conclusion comes into our mind rapidly: if we consider source code words in "word" units, they probably will not fit in the fixed set of words used in a human language like English. Therefore, the LM's knowledge acquired during the pre-training is not entirely valid when we get out of that fixed set of words that compose a language. Most words in the programming language are neologisms for the LMs pre-trained in English, and thus, it needs to incorporate them and their relationships into the learned knowledge. For the sub-word units, the LM can be less sensitive to the neologisms. Potentially, it could be more robust the more divided a word is since the set of bytes or chars is more straightforward than the chunks present in richer constructions or information units.

Going deeper into this research, concerning the pre-training effect over the LMs' modeling source code, it could be worth researching the relationship between the pre-

training in different human-spoken languages and the LMs' ability to work with existing source code-specific programming languages.

4.3. Reviewing the Textual Assessment of the Resulting LMs

Regarding the tests done generating source code or filling in the blanks using the trained LMs, we think that, in general, the textual results obtained are not so good, yet they are informative of how LMs are working and how they can be improved. One of the things that can explain these results is the dataset used. In this case, we used a public dataset that other researchers can use to make results and experiments comparable and replicable. In the literature, we did not find a standard dataset for these tasks against which we can compare easily. Other papers [10,15,63] used custom datasets, but we found a lack in the literature of well-recognized code datasets to use. Comparing with other recent papers in the NLP field used as the basis for this research [37,38,45,46], the dataset may be relatively small to train a big LM to accomplish appropriately challenging tasks like generating source code or auto-completing it. Future work may test these or new approaches in bigger datasets to train big LMs focused on modeling the Python language and checking whether the results are better. Recent examples of LMs, such as GPT-3 [37], claim to produce accurate textual outputs even in contexts in which they were not trained. Part of the explanation given for that ability is the use of gargantuan datasets combined with Transformer and other attention-based architectures. Therefore, those approaches can also be relevant to other contexts like ours. Another line for future research can be using datasets focused on specific libraries or Python aspects and verifying if these approaches specialize positively for those contexts the DNN models used in this paper.

Related to evaluating the code generated or filled, we observed in the literature different approaches [13,64]. In the context of LMs' modeling source code, many papers and software libraries devoted to translating between programming languages typically evaluate text generation using methods and metrics like BLEU [65] or variants like SacreBLEU [66]. Other papers like [10] rely on the accuracy to assess an LM's performance based on deep learning. Some models can even solve different tasks that are part of existing benchmarks [67] or are evaluated, checking their perplexity (similarly to those that evaluate the model using the accuracy). The current tendency in large models is to evaluate them using human intervention to evaluate the output's quality [37,45]. We assessed the models using accuracy during our experiments and evaluated the models' textual outputs based on our prior human knowledge. It would be interesting for the future to plan new evaluation processes involving larger cohorts of source code experts to evaluate the models such as [37,45] did. One of the potential new assessments can be usability tests conducted with programmers. They can compare the code they would write against the code proposed by any of the DNNs presented here and the result from other common code auto-completion tools included in IDEs. As we outlined in the Results Section, relying only on metrics like accuracy should not be enough. As in our case, accuracy and the other metrics can be a good indicator of the model's performance, yet we need to verify the LMs' behavior and quality using complementary methods like specialized metrics or human evaluation. For tasks like auto-completion or source code generation, there are no existing specialized metrics (like BLEU in translation, for example), so one of the future research lines is improving the evaluation of LMs for source code. Based on some existing ideas in broad NLP, there are many opportunities to explore in that sense; from new test suites for language models used in source code contexts [13] to behavioral testing [68] or human-centric evaluation of the models [64] with particular emphasis on reproducible and unbiased assessments or combinations of automatic testing and human-centric assessments.

5. Conclusions

This paper compares how different approaches to tokenization models, deep neural network architectures, pre-trained models, and transfer learning affect the results from language models used to generate source code or auto-complete software pieces. We

studied different DNN architectures like AWD-LSTM, AWD-QRNN, and Transformer to seek which kind works better with different tokenization models (word, unigram, BPE, and char). Furthermore, we compared the pre-training effect on the results given by LMs after training them and fine-tuning them via transfer learning to work with other languages (English language to Python programming language). As a result of this work, we find that in small LMs (like our AWD-LSTM and AWD-QRNN models), the tokenization using char-sized chunks works better than using any other tokenization models. In larger models like Transformer GPT-2, the accuracy was slightly worse than the other architectures. However, GPT-2 gave better results on the source code generation tests (even using another tokenization approach like BPE over raw bytes). For source code auto-completion, we tested some Transformer models like BERT and RoBERTa. While their accuracy was above any other models, they did not perform very well when performing the tasks proposed in our tests. In general, we find that pre-trained models work better, even if they are not trained initially for a programming language like Python (our models were pre-trained using the English language). Finally, related to evaluating tasks like automating source code generation and source code auto-completion, we raise concerns about the literature gaps and propose some research lines to work on in the future.

Author Contributions: Conceptualization, J.C.-B., S.V., and I.F.; methodology, J.C.-B.; software, J.C.-B. and S.V.; validation, F.M.-F. and I.F.; writing and original draft preparation, J.C.-B.; supervision, F.M.-F. and I.F. All authors read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The source code used is available in 10.5281/zenodo.4011767. The resulting neural network models are available in 10.5281/zenodo.4293857.

Acknowledgments: We thank the IBM Quantum team and the IBM Research ETX team for the insightful discussions about this research and the support received during the development of this research.

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

References

1. Kannan, A.; Kurach, K.; Ravi, S.; Kaufmann, T.; Tomkins, A.; Miklos, B.; Corrado, G.; Lukacs, L.; Ganea, M.; Young, P.; et al. Smart reply: Automated response suggestion for email. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016; pp. 955–964.
2. Bryant, C.; Briscoe, T. Language model based grammatical error correction without annotated training data. In Proceedings of the Thirteenth Workshop on Innovative Use of NLP for Building Educational Applications, New Orleans, LA, USA, 5 June 2018; pp. 247–253.
3. Ghosh, S.; Kristensson, P.O. Neural networks for text correction and completion in keyboard decoding. *arXiv* **2017**, arXiv:1709.06429.
4. Allamanis, M.; Barr, E.T.; Devanbu, P.; Sutton, C. A survey of machine learning for big code and naturalness. *ACM Comput. Surv. (CSUR)* **2018**, *51*, 1–37. [CrossRef]
5. Chen, H.; Le, T.H.M.; Babar, M.A. Deep Learning for Source Code Modeling and Generation: Models, Applications and Challenges. *ACM Comput. Surv. (CSUR)* **2020**, *53*, 1–38.
6. Nguyen, A.T.; Nguyen, T.N. Graph-based statistical language model for code. In Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 24 May 2015; Volume 1, pp. 858–868.
7. Bielik, P.; Raychev, V.; Vechev, M. PHOG: Probabilistic model for code. In Proceedings of the 33rd International Conference on International Conference on Machine Learning, New York, NY, USA, 19–24 June 2016; pp. 2933–2942.
8. Cruz-Benito, J.; Faro, I.; Martín-Fernández, F.; Therón, R.; García-Peñalvo, F.J. A Deep-Learning-based proposal to aid users in Quantum Computing programming. In *International Conference on Learning and Collaboration Technologies*; Springer: Berlin, Germany, 2018; pp. 421–430.
9. Oda, Y.; Fudaba, H.; Neubig, G.; Hata, H.; Sakti, S.; Toda, T.; Nakamura, S. Learning to generate pseudo-code from source code using statistical machine translation (t). In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 9–13 November 2015; pp. 574–584.
10. Tiwang, R.; Oladunni, T.; Xu, W. A Deep Learning Model for Source Code Generation. In Proceedings of the 2019 SoutheastCon, Huntsville, AL, USA, 11–14 April 2019; pp. 1–7.

11. Fedus, W.; Goodfellow, I.; Dai, A.M. MaskGAN: Better Text Generation via Filling in the _____. In Proceedings of the 6th International Conference on Learning Representations, Vancouver, BC, Canada, 30 April–3 May 2018.
12. Nguyen, A.T.; Nguyen, T.T.; Nguyen, T.N. Lexical statistical machine translation for language migration. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, Saint Petersburg, Russia, 18 August 2013; pp. 651–654.
13. Roziere, B.; Lachaux, M.A.; Chanussot, L.; Lample, G. Unsupervised Translation of Programming Languages. *Adv. Neural Inf. Process. Syst.* **2020**, *33*.
14. Proksch, S.; Lerch, J.; Mezini, M. Intelligent code completion with Bayesian networks. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **2015**, *25*, 1–31. [CrossRef]
15. Li, J.; Wang, Y.; Lyu, M.R.; King, I. Code completion with neural attention and pointer networks. In Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18), Melbourne, Australia, 19–25 August 2017; pp. 4159–4165.
16. Donahue, C.; Lee, M.; Liang, P. Enabling Language Models to Fill in the Blanks. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, Online, 5–10 July 2020; pp. 2492–2501. [CrossRef]
17. Allamanis, M.; Barr, E.T.; Bird, C.; Sutton, C. Suggesting accurate method and class names. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Bergamo, Italy, 30 August–4 September 2015; pp. 38–49.
18. Karampatsis, R.M.; Sutton, C. Maybe deep neural networks are the best choice for modeling source code. *arXiv* **2019**, arXiv:1903.05734.
19. Karampatsis, R.M.; Babii, H.; Robbes, R.; Sutton, C.; Janes, A. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, New York, NY, USA, 24 June–16 July 2020; pp. 1073–1085. [CrossRef]
20. Ganin, Y.; Ustinova, E.; Ajakan, H.; Germain, P.; Larochelle, H.; Laviolette, F.; Marchand, M.; Lempitsky, V. Domain-adversarial training of neural networks. *J. Mach. Learn. Res.* **2016**, *17*, 1–35.
21. Kim, Y.; Jernite, Y.; Sontag, D.; Rush, A.M. Character-Aware Neural Language Models. In Proceedings of the AAAI'16: Thirtieth AAAI Conference on Artificial Intelligence, Phoenix, AZ, USA, 12–17 February 2016; AAAI Press: Menlo Park, CA, USA, 2016; pp. 2741–2749.
22. Karpathy, A. The unreasonable effectiveness of recurrent neural networks. *Andrej Karpathy Blog* **2016**, *21*, 23.
23. Merity, S.; Keskar, N.S.; Socher, R. Regularizing and Optimizing LSTM Language Models. In Proceedings of the International Conference on Learning Representations, 2018, Vancouver, BC, Canada, 30 April–3 May 2018.
24. Bradbury, J.; Merity, S.; Xiong, C.; Socher, R. Quasi-recurrent neural networks. In Proceedings of the 5th International Conference on Learning Representations (ICLR 2017), Toulon, France, 24–26 April 2017.
25. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems*; The MIT Press: Cambridge, MA, USA, 2017; pp. 5998–6008.
26. Wang, D.; Gong, C.; Liu, Q. Improving Neural Language Modeling via Adversarial Training. In Proceedings of the 36th International Conference on Machine Learning, Long Beach, CA, USA, 9–15 June 2019; Chaudhuri, K., Salakhutdinov, R., Eds.; PMLR: Long Beach, CA, USA, 2019; Volume 97, pp. 6555–6565.
27. Gong, C.; He, D.; Tan, X.; Qin, T.; Wang, L.; Liu, T.Y. Frage: Frequency-agnostic word representation. *Adv. Neural Inf. Process. Syst.* **2018**, *31*, 1334–1345.
28. Takase, S.; Suzuki, J.; Nagata, M. Direct Output Connection for a High-Rank Language Model. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, 31 October–4 November 2018; pp. 4599–4609.
29. Yang, Z.; Dai, Z.; Salakhutdinov, R.; Cohen, W.W. Breaking the Softmax Bottleneck: A High-Rank RNN Language Model. In Proceedings of the International Conference on Learning Representations, 2018, Vancouver, BC, Canada, 30 April–3 May 2018.
30. Krause, B.; Kahembwe, E.; Murray, I.; Renals, S. Dynamic Evaluation of Neural Sequence Models. In Proceedings of the 35th International Conference on Machine Learning, Stockholm, Sweden, 10–15 July 2018; Dy, J., Krause, A., Eds.; PMLR: Stockholmsmässan, Stockholm Sweden, 2018; Volume 80, pp. 2766–2775.
31. Rae, J.W.; Potapenko, A.; Jayakumar, S.M.; Hillier, C.; Lillicrap, T.P. Compressive Transformers for Long-Range Sequence Modelling. In Proceedings of the International Conference on Learning Representations, 2019, New Orleans, LA, USA, 6–9 May 2019.
32. Dai, Z.; Yang, Z.; Yang, Y.; Carbonell, J.G.; Le, Q.; Salakhutdinov, R. Transformer-XL: Attentive Language Models beyond a Fixed-Length Context. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, Florence, Italy, 28 July–2 August 2019; pp. 2978–2988.
33. Baevski, A.; Auli, M. Adaptive Input Representations for Neural Language Modeling. In Proceedings of the International Conference on Learning Representations, 2018, Vancouver, BC, Canada, 30 April–3 May 2018.
34. Merity, S.; Keskar, N.S.; Socher, R. An analysis of neural language modeling at multiple scales. *arXiv* **2018**, arXiv:1803.08240.
35. Howard, J.; Ruder, S. Universal Language Model Fine-tuning for Text Classification. In Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Melbourne, Australia, 15–20 July 2018; pp. 328–339.

36. Eisenschlos, J.; Ruder, S.; Czapla, P.; Kadras, M.; Gugger, S.; Howard, J. MultiFiT: Efficient Multi-lingual Language Model Fine-tuning. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), Hong Kong, China, 3–7 November 2019; Association for Computational Linguistics: Hong Kong, China, 2019; pp. 5702–5707. [CrossRef]
37. 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. *Adv. Neural Inf. Process. Syst.* **2020**, *33*.
38. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the NAACL-HLT (1), Minneapolis, MN, USA, 2–7 June 2019.
39. Mikolov, T.; Deoras, A.; Kombrink, S.; Burget, L.; Černockỳ, J. Empirical evaluation and combination of advanced language modeling techniques. In Proceedings of the Twelfth Annual Conference of the International Speech Communication Association, Florence, Italy, 27–31 August 2011.
40. Merity, S.; Xiong, C.; Bradbury, J.; Socher, R. Pointer sentinel mixture models. In Proceedings of the 5th International Conference on Learning Representations (ICLR 2017), Toulon, France, 24–26 April 2017.
41. Chelba, C.; Mikolov, T.; Schuster, M.; Ge, Q.; Brants, T.; Koehn, P.; Robinson, T. One Billion Word Benchmark for Measuring Progress in Statistical Language Modeling. In Proceedings of the Fifteenth Annual Conference of the International Speech Communication Association, Singapore, 14–18 September 2014.
42. Hochreiter, S.; Schmidhuber, J. Long short-term memory. *Neural Comput.* **1997**, *9*, 1735–1780. [CrossRef] [PubMed]
43. Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I.; Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* **2014**, *15*, 1929–1958.
44. Young, T.; Hazarika, D.; Poria, S.; Cambria, E. Recent trends in deep learning based natural language processing. *IEEE Comput. Intell. Mag.* **2018**, *13*, 55–75. [CrossRef]
45. Radford, A.; Wu, J.; Child, R.; Luan, D.; Amodei, D.; Sutskever, I. Language Models are Unsupervised Multitask Learners. *OpenAI Blog* **2019**, *1*, 9.
46. Liu, Y.; Ott, M.; Goyal, N.; Du, J.; Joshi, M.; Chen, D.; Levy, O.; Lewis, M.; Zettlemoyer, L.; Stoyanov, V. Roberta: A robustly optimized bert pretraining approach. *arXiv* **2019**, arXiv:1907.11692.
47. Wang, A.; Cho, K. BERT has a Mouth, and It Must Speak: BERT as a Markov Random Field Language Model. In Proceedings of the Workshop on Methods for Optimizing and Evaluating Neural Language Generation (NAACL HLT 2019), 2019, Minneapolis, MN, USA, 6 June 2019; pp. 30–36.
48. Sennrich, R.; Haddow, B.; Birch, A. Neural Machine Translation of Rare Words with Subword Units. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Berlin, Germany, 7–12 August 2016; pp. 1715–1725.
49. Bostrom, K.; Durrett, G. Byte Pair Encoding is Suboptimal for Language Model Pretraining. In Proceedings of the Findings of the Association for Computational Linguistics: EMNLP 2020, Association for Computational Linguistics, Online, 16–20 November, 2020; pp. 4617–4624.
50. Schuster, M.; Nakajima, K. Japanese and korean voice search. In Proceedings of the 2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Kyoto, Japan, 25–30 March 2012, pp. 5149–5152.
51. Husain, H.; Wu, H.H.; Gazit, T.; Allamanis, M.; Brockschmidt, M. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv* **2019**, arXiv:1909.09436.
52. Howard, J.; Gugger, S. Fastai: A layered API for deep learning. *Information* **2020**, *11*, 108. [CrossRef]
53. Kudo, T.; Richardson, J. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, Brussels, Belgium, 31 October–4 November 2018; pp. 66–71.
54. Wolf, T.; Debut, L.; Sanh, V.; Chaumond, J.; Delangue, C.; Moi, A.; Cistac, P.; Rault, T.; Louf, R.; Funtowicz, M.; et al. Hugging Face's Transformers: State-of-the-Art Natural Language Processing. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, Association for Computational Linguistics, Online, 16–20 November 2020; pp. 38–45.
55. Ruder, S.; Peters, M.E.; Swayamdipta, S.; Wolf, T. Transfer learning in natural language processing. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Tutorials, Minneapolis, MN, USA, 2 June 2019; pp. 15–18.
56. Chronopoulou, A.; Baziotis, C.; Potamianos, A. An Embarrassingly Simple Approach for Transfer Learning from Pretrained Language Models. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), Minneapolis, MN, USA, 2–7 June 2019; pp. 2089–2095.
57. Honnibal, M.; Montani, I.; Van Landeghem, S.; Boyd, A. spaCy: Industrial-strength Natural Language Processing in Python. *Zenodo* **2020**. [CrossRef]
58. Czapla, P.; Howard, J.; Kardas, M. Universal language model fine-tuning with subword tokenization for polish. *arXiv* **2018**, arXiv:1810.10222.
59. Smith, L.N. Cyclical learning rates for training neural networks. In Proceedings of the 2017 IEEE Winter Conference on Applications of Computer Vision (WACV), Santa Rosa, CA, USA, 24–31 March 2017; pp. 464–472.

60. Smith, L.N.; Topin, N. Super-convergence: Very fast training of neural networks using large learning rates. In Proceedings of the Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications. International Society for Optics and Photonics, Baltimore, MD, USA, 15–17 April 2019; Volume 11006, p. 1100612.
61. Cruz-Benito, J.; Vishwakarma, S. cbjuan/tokenizers-neural-nets-2020- paper: v1.0. *Zenodo* **2020**. [CrossRef]
62. Cruz-Benito, J.; Vishwakarma, S. NN models produced by cbjuan/tokenizers-neural-nets-2020-paper: v1.0. *Zenodo* **2020**. [CrossRef]
63. Raychev, V.; Bielik, P.; Vechev, M. Probabilistic model for code with decision trees. *ACM SIGPLAN Not.* **2016**, *51*, 731–747. [CrossRef]
64. Celikyilmaz, A.; Clark, E.; Gao, J. Evaluation of Text Generation: A Survey. *arXiv* **2020**, arXiv:2006.14799.
65. Papineni, K.; Roukos, S.; Ward, T.; Zhu, W.J. BLEU: A method for automatic evaluation of machine translation. In Proceedings of the 40th annual meeting of the Association for Computational Linguistics, Philadelphia, PA, USA, 7–12 July 2002; pp. 311–318.
66. Post, M. A Call for Clarity in Reporting BLEU Scores. In Proceedings of the Third Conference on Machine Translation: Research Papers, Belgium, Brussels, 31 October–1 November 2018; Association for Computational Linguistics: Brussels, Belgium, 2018; pp. 186–191. [CrossRef]
67. Weston, J.; Bordes, A.; Chopra, S.; Rush, A.M.; van Merriënboer, B.; Joulin, A.; Mikolov, T. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv* **2015**, arXiv:1502.05698.
68. Ribeiro, M.T.; Wu, T.; Guestrin, C.; Singh, S. Beyond Accuracy: Behavioral Testing of NLP Models with CheckList. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics, Online, 5–10 July 2020; pp. 4902–4912. [CrossRef]