Modelo de red neuronal artificial para la generación automática de código en aplicaciones de interfaces gráficas

Artificial neural network model for automatic code generation in graphical interface applications

DOI: http://doi.org/10.17981/ingecuc.19.1.2023.04

Artículo de Investigación Científica. Fecha de Recepción: 13/09/2022, Fecha de Aceptación: 14/12/2022

Daniel Esteban Arenas-Varela
Corporación Universitaria Comfacauca – Unicomfacauca - Research group MIND. Popayán, (Colombia)
danielarenas@unicomfacauca.edu.co

Julián Fernando Muñoz-Ordóñez
Corporación Universitaria Comfacauca – Unicomfacauca - Research group MIND. Popayán, (Colombia)
jfmunoz@unicomfacauca.edu.co

To cite this paper
D. Arenas Varela, J. Muñoz Ordóñez "Artificial neural network model for automatic code generation in graphical interface applications" DOI: http://doi.org/10.17981/ingecuc.19.1.2023.04

Resumen

Introducción: En la actualidad, la industria del desarrollo de software vive su época dorada debido al avance en áreas relacionadas con el aprendizaje máquina el cual es parte de las técnicas de IA, estos avances han permitido que tareas consideradas exclusivamente del ser humano sean resueltas utilizando un equipo de cómputo. Sin embargo, la complejidad y la extensa área que están abarcando los nuevos proyectos que deben ser desarrollados utilizando lenguajes de programación han generado que los tiempos de entrega de los proyectos se vean ralentizados y la productividad de la empresa afectada.
Objetivo: Esta investigación presenta la metodología que se llevó a cabo para la construcción de un modelo de red neuronal recurrente para la generación automática de código fuente relacionado con interfaces gráficas de usuario utilizando lenguaje de programación Python.
Metodología: Mediante la construcción de un conjunto de datos relacionado con el lenguaje natural para la descripción de interfaces gráficas programadas en Python se construye un modelo de red neuronal profunda para generar código fuente automático.
Resultados: El modelo entrenado logra alcanzar valores de pérdida y perplejidad de 1.57 y 4.82 respectivamente en la etapa de validación evitando el sobreajuste en el entrenamiento del modelo.
Conclusiones: Un modelo de red neuronal es entrenado logrando procesar el lenguaje natural relacionado con la petición de creación de interfaces gráficas utilizando el lenguaje de programación Python para generar automáticamente código fuente que puede ser ejecutado a través del intérprete de Python.

Palabras clave

Aprendizaje automático; procesamiento de lenguaje natural; interfaz gráfica; transformadores; Tkinter; aprendizaje profundo; generación automática de código.

Abstract

Introduction: Currently, the software development industry is living in its golden age due to the progress in areas related to machine learning, which is part of AI techniques. These advances have allowed tasks considered exclusively human to be solved using a computer. However, the complexity and the extensive area covered by new projects that must be developed using programming languages have slowed down project delivery times and affected the company's productivity.
Objective: This research presents the methodology carried out for constructing a recurrent neural network model for the automatic generation of source code related to graphical user interfaces using Python programming language.
Method: By constructing a natural language-related dataset for describing graphical interfaces programmed in Python, a deep neural network model is built to generate automatic source code.
Results: The trained model achieves loss and perplexity values of 1.57 and 4.82, respectively, in the validation stage, avoiding overfitting in the model's training.
Conclusions: A neural network model is trained to process the natural language related to the request to create graphical interfaces using the Python programming language to automatically generate source code that can be executed through the Python interpreter.

Key Words

Machine learning; Natural language processing; Graphical interface; Transfotmers; Tkinter; Deep learning; Automatic code generation; Automatic code generation.

I. INTRODUCCIÓN

Trends in software development have shown that its growth and change are compared to technological advances. Areas such as artificial intelligence, the internet of things, advanced web applications, and languages for application development, among many others [1] have a high application in different areas of computing and are currently generating that the software development industry is living a golden age in the resolution of essential tasks for the world [2]. Artificial intelligence has shown that the dream of human beings to replicate their way of learning in a computer is getting closer and closer; tasks that were exclusive to humans are already being performed by computational algorithms that obtain similar or superior results. Nowadays, improving code programming tasks is a necessity [3], Every day, the tasks to be solved by developers are bigger and more complex. Usually, each project has repetitive tasks that revolve around the problem's solution; these tasks tend to slow delivery times since they entail a more significant effort in the development stage, affecting productivity, simplification, portability, and consistency [4]. Having these repetitive code tasks generate that developers lose the focus of the application and the context of the logic producing unnecessary delays in achieving the objective [5].

Currently, the solutions that seek to help developers are mainly focused on the creation of development editors, which allow them to add code fragments that speed up the creation process. [5], However, this tool does not have any associated machine learning model. Automatic code generation is the branch of computer science that seeks to assist programmers in the automatic and intelligent creation of code based on input and output variables [6] allowing to detection or building of executable source code using a programming language and natural language processing, thus speeding up application implementation processes. In this work, we intend to deepen the Big Code research area. [6] Which joins machine learning models together with deep neural network architectures to create software applications that are robust and scalable, speeding up the development processes that are currently in such high demand. Therefore, this project seeks to answer the following question: Is it possible to create a model based on artificial intelligence that automatically generates source code related to graphic interface generation applications?

Automatic code generation represents the future in software engineering and development, creating applications that favor productivity: referred to the generation of code only once and its reuse in other projects; simplification: which seeks to create source code of some abstract description; portability: refers to the adaptation of the generated source code to multiple development platforms, specifically to programming languages; and finally consistency: generating repeatability in the results obtained by the code generator, are considered key aspects that all development based on programming must contain. On the other hand, advances in automatic code generation will have a direct impact on the performance and productivity of companies because their objectives are focused on predicting explicit code or program structure by receiving as input variables multimodal data sources such as incomplete program codes, codes in different programming languages and descriptions based on natural language [6]. With respect to software developers, the benefit of automatic source code generation will be to assist professionals in the generation of repetitive code structures that solve tasks based on the description of the input requirements and the corresponding output, maximizing productivity in development and ensuring that the professional does not lose focus of the application and the initial logic to have more robust programs that meet quality standards: productivity, reusability, simplification, and portability.

II. STATE OF THE ART

The main applications for generating source code in programming language-related topics have been studied in various types of research. In 2017, Ling et at. [7], present a novel approach based on a neural network architecture that generates an output sequence dependent on several input functions; this is done to address the issue related to the dependency of language generation tasks mediated by the production of structured and unstructured text; its development allows to generate a mixture between natural language and a structured specification. Rabinovich et al. [8], proposes that code generation tasks comprise processes that receive unstructured or partially structured inputs and produce executable outputs; in their work, they obtain a model with 79.2 in BLEU on the Hearthstone dataset; their model consists of abstract syntax networks whose output is represented by abstract syntax trees (ASTs). Yin et al. [9] consider the transformation of natural language descriptions to code written in Python; they propose a neural network architecture driven by a grammatical model to shape the input data to the model and consider it as prior knowledge; their results far exceed the approaches that existed to date related to semantic analysis and code generation. In 2018, Stehnii continued in the line of generating code focused on the Python programming language; its development focuses on the premise that software development requires multiple skills and that it is unlikely to know all existing solutions by the development community; this allows him to consider the idea of creating a sequence by sequence model with a recursive coder, also presents as a novel point working with syntactic trees instead of a simple sequence in the model input data. [10]. Yin et al. present TRANX [11], a neural semantic parser based on changing natural language expressions into formal representations of meaning; their results show that TRANX is highly accurate, and its generalization capability with different tasks such as semantic analysis and code generation is reliable and significant. Finally, in 2020, Zhu et al. [3], state that with the progress in the field of deep learning, code generation from natural language is relevant to research; in their research, they state that although there are many code generators, most of them use BLEU as evaluation metric, in addition, the data sets used lack diversity, for such reasons, the research focuses its efforts on tracking a more robust data set and evaluate it with five new aspects: lexical similarity, tree similarity, syntactic legality, semantic legality, and functional correctness. With the latter, they seek to give a greater analysis capacity to the methods or algorithms in the state of the art regarding automatic code generation. The previously analyzed works can be framed in 4 traditional approaches to handle the syntactic and semantic part of the source code, namely: domain-specific language-guided models, probabilistic grammars, simple probabilistic language models and simple neural language models [6]. Concerning domain-specific language-guided models (DSLs), research such as [12][13][14] have studied such algorithms and have succeeded in reducing the search space to suggest expressions that allow source code completion from succinct type computations and higher-order functions. On the probabilistic grammar side, some authors [15][16] argue that the production of rules can generate all possible source code instructions and that these models can find patterns that generate more complex code structures. The n-gram language models are another approach that enables automatic code construction, from Hindle et al. [17] who built a language model for source code using n-grams in a novel way to [18] who used n-grams for language mining tasks. Finally, simple neural program models consist of neural networks that embed models that use the one-hot encoding of a word to transform it into an intermediate vector of words with a shorter length than the initial word. These algorithms are known as distributed word representations [6]. In this research, an algorithm will be developed for the automatic generation of source code based on the Python programming language, with datasets recognized as state of the art and using the metrics proposed in [3], the innovative part of the project will be the construction of simple graphical user interface programs using a case study related to a web component.

III. MATERIALS AND METHODS

The main task in the development of the project was the creation of 7 graphical interfaces using the Python Tkinter library, which were used to implement the data capture tool, as follows, the Python code of each GUI and its respective representation (See Figures 1-2).

Graphic Interface Code 1
from tkinter import *
root = Tk()

```
root.columnconfigure(0, weight=0)
root.columnconfigure(1, weight=1)
root.rowconfigure(2, weight=1)
Label(root, text= "Nombre").grid(row=0, columna=0)
Label(root, text= "Apellido").grid(row=1, columna=0)
Entry(root).grid(row=0, column=1, sticky=E+W)
Entry(root).grid(row=1, column=1, sticky=E+W)
Button(root, text="Aceptar”).grid(pady=10, padx=10,
row=3, columna=0, columnspan=2,sticky=S+N+E+W)
root.mainloop()
```

Table I Source code of graphical interface related to Figure 1. **Source:** self-made.

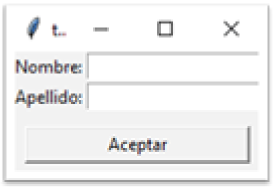


Fig. 1 Graphical interface 1: two labels, two text boxes, and one button. Source: self-made.

Graphic Interface Code 2

```
import tkinter as tk
class Application(tk.Frame):
        def _init_(self, master=None):
        super()._init_(master)
        self.master = master
        self.pack()
        self.create_widgets()
def create_widgets(self):
        self.hi_there = tk.Button(self)
        self.hi_there = ["text"] = "haz clic"
        self.hi_there = ["command"] = self.say_hi
        self.hi_there.pack(side="top")
        self.quit      =      tk.Button(self,      text="Salir",      fg="red"),
command=self.master.destroy
        self.quit.pack(side="bottom")
def say_hi(self):
     print("Hola mundo")
root = tk.Tk()
app = Application(master=root)
app,mainloop()
```

Table II Source code of graphical interface related to Figure 2. Source: self-made.

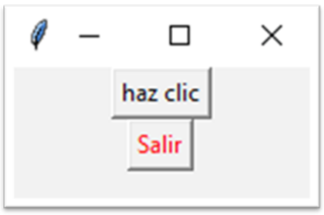
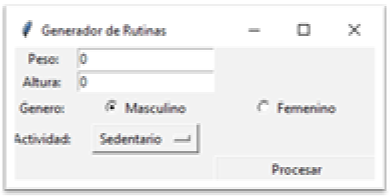


Fig 2 Graphical interface 2: two buttons.

In addition, other graphical interfaces built to feed the dataset are shown; the GUIs can be seen in Figures 3-7.



Peso: 0
Altura: 0
Genero: [✓ Masculino] Femenino
Actividad: Sedentario

Fig 3 Graphical interface with widgets: labels, text boxes, radio buttons, and buttons. Source: self-made.

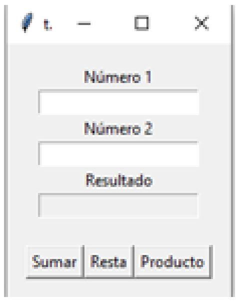


Fig 4 Graphical interface with widgets: labels, text boxes, buttons. Source: self-made.

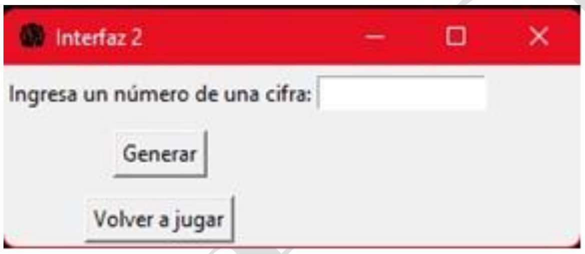
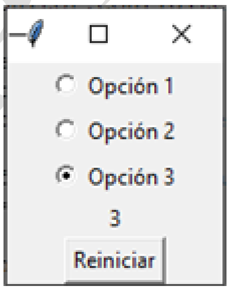
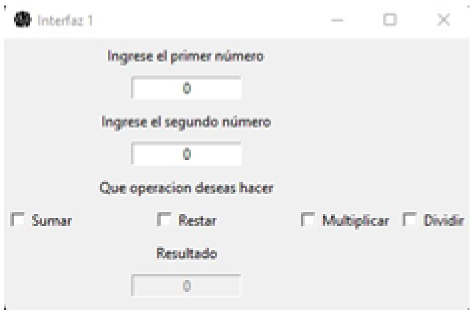


Fig 5 Graphical interface with widgets: text labels, buttons, and text box. Source: self-made.



Opción 1
Opción 2
[✓ Opción 3]
3

Fig 6 Graphical interface with widgets: radio buttons and buttons. Source: self-made.



Ingrese el primer número: 0
Ingrese el segundo número: 0
Que operacion deseas hacer
Sumar Restar Multiplicar Dividir
Resultado: 0

Fig 7 Graphical interface with widgets: labels, textboxes, check buttons. Source: self-made.

The dataset was constructed by applying a data capture tool to a group of Systems Engineering students from different semesters; the dataset is composed of two essential bases: a natural language text and a python code text. The first version of the dataset comprises 722 records describing each of the interfaces created (see Figure 8).

[illegible]

Fig 8 Dataset Version 1, result provided by the data capture tool. Source: self-made.

The final version of the dataset is composed of a sentence in Spanish describing the graphic interface to be created; this process will be expanded in the chapter on results and analysis. Once the data for training and its natural language processing had been constructed, an algorithm for the automatic generation of code for the generation of graphical interfaces was built by applying artificial intelligence techniques related to deep neural networks.

Transformers are the predominant architecture in most state-of-the-art NLP applications today; the following explains the transformer architecture that was applied to a sequence-to-sequence learning problem. The transformer can be understood in terms of its three components:

1. An encoder that encodes an input sequence into state representation vectors.

2. An attention mechanism that allows our transformer model to focus on the correct aspects of the sequential input stream. This is used repeatedly in both the encoder and decoder to help them contextualize the input data.

3. A decoder decodes the state representation vector to generate the target output sequence.

The encoder in this research accepts a batch of source sequences and sequence masks as input. The source mask contains one at locations where the input sequence has valid values and 0 where the input sequence has <pad> values. This ensures that the attention mechanism within the encoder does not pay attention to the <pad> values; the complete transformer is shown in Figure 9.

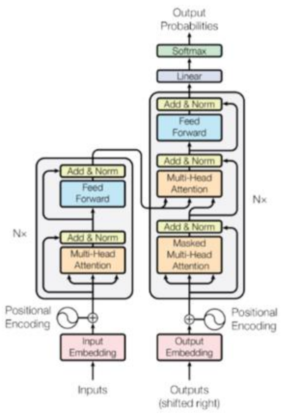
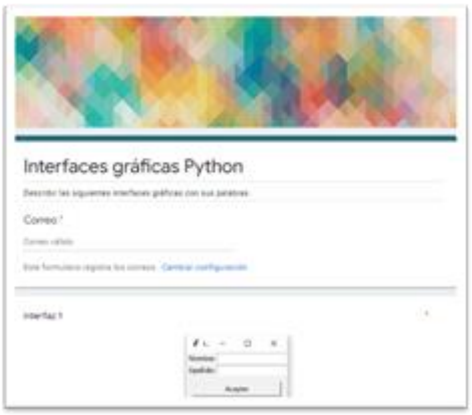


Fig 9 The complete architecture of a Transformer. Source: Wolfram Computation Meets Knowledge. Source: self-made.

IV. ANALYSIS AND RESULTS

A Google Forms form was built where the graphical interfaces created were assembled, and a simple textual description of each one was requested using natural language, which was applied in the Artificial Intelligence & Big Data

(IA&BD) seedbed of the Corporación Universitaria Comfacauca – Unicomfacauca, which resulted in 722 records (See Figure 10).



Interfaces gráficas Python

Describe las siguientes interfaces gráficas con sus palabras

Correo *

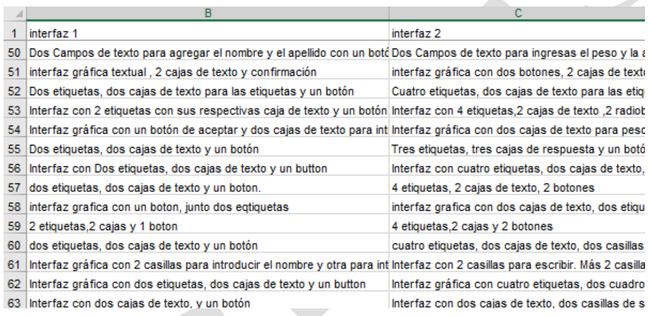
Correo válido

Este formulario registra los correos. Cambiar configuración

Interfaz 1

Fig 10 Data capture tool applied to Systems Engineering students from different semesters. Source: self-made.

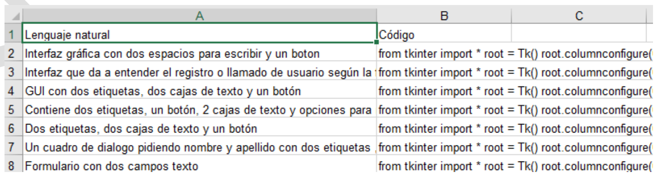
The captured information is recorded as follows: in a .xlsx file containing 722 records (See Figure 11).



	B	C
1	interfaz 1	interfaz 2
50	Dos Campos de texto para agregar el nombre y el apellido con un botó	Dos Campos de texto para ingresas el peso y la a
51	interfaz gráfica textual , 2 cajas de texto y confirmación	interfaz gráfica con dos botones, 2 cajas de text
52	Dos etiquetas, dos cajas de texto para las etiquetas y un botón	Cuatro etiquetas, dos cajas de texto para las etiq
53	Interfaz con 2 etiquetas con sus respectivas caja de texto y un botón	Interfaz con 4 etiquetas,2 cajas de texto ,2 radiob
54	Interfaz gráfica con un botón de aceptar y dos cajas de texto para int	Interfaz gráfica con dos cajas de texto para peso
55	Dos etiquetas, dos cajas de texto y un botón	Tres etiquetas, tres cajas de respuesta y un botó
56	Interfaz con Dos etiquetas, dos cajas de texto y un button	Interfaz con cuatro etiquetas, dos cajas de texto,
57	dos etiquetas, dos cajas de texto y un boton.	4 etiquetas, 2 cajas de texto, 2 botones
58	interfaz grafica con un boton, junto dos eqtiquetas	interfaz grafica con dos cajas de texto, dos etiqu
59	2 etiquetas,2 cajas y 1 boton	4 etiquetas,2 cajas y 2 botones
60	dos etiquetas, dos cajas de texto y un botón	cuatro etiquetas, dos cajas de texto, dos casillas
61	Interfaz gráfica con 2 casillas para introducir el nombre y otra para int	Interfaz con 2 casillas para escribir. Más 2 casilla
62	Interfaz gráfica con dos etiquetas, dos cajas de texto y un button	Interfaz gráfica con cuatro etiquetas, dos cuadro
63	Interfaz con dos cajas de texto, y un botón	Interfaz con dos cajas de texto, dos casillas de s

Fig 11 Captured records. Source: self-made.

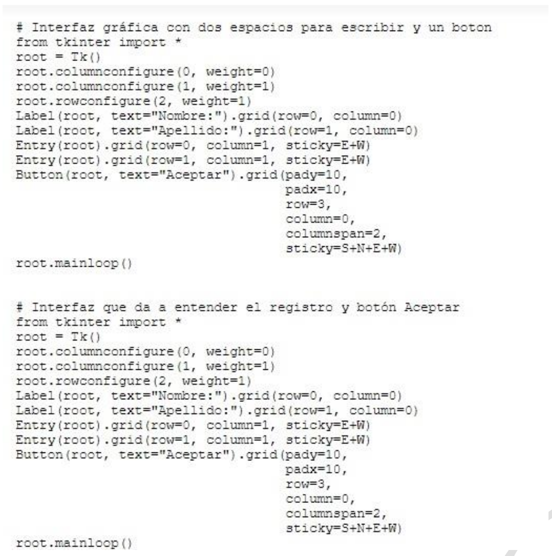
People described the graphical interfaces using natural language processing; it was applied to 115 students, and 722 answers were recorded, then a filtering process was performed to eliminate erroneous answers, and finally, the final version of the dataset was built, which is composed of a sentence in Spanish with the description of the graphical interface and Python code (See Figure 12).



	A	B	C
1	Lenguaje natural	Código	
2	Interfaz gráfica con dos espacios para escribir y un boton	from tkinter import * root = Tk() root.columnconfigure(	
3	Interfaz que da a entender el registro o llamado de usuario según la	from tkinter import * root = Tk() root.columnconfigure(	
4	GUI con dos etiquetas, dos cajas de texto y un botón	from tkinter import * root = Tk() root.columnconfigure(	
5	Contiene dos etiquetas, un botón, 2 cajas de texto y opciones para	from tkinter import * root = Tk() root.columnconfigure(	
6	Dos etiquetas, dos cajas de texto y un botón	from tkinter import * root = Tk() root.columnconfigure(	
7	Un cuadro de dialogo pidiendo nombre y apellido con dos etiquetas	from tkinter import * root = Tk() root.columnconfigure(	
8	Formulario con dos campos texto	from tkinter import * root = Tk() root.columnconfigure(	

Fig 12 Dataset final version. Source: self-made.

Figure 12 shows the dataset filtered and stored in a spreadsheet; however, the format required for training the dataset using recurrent neural network architectures must be modified; Figure 13 shows the result obtained by making the change:

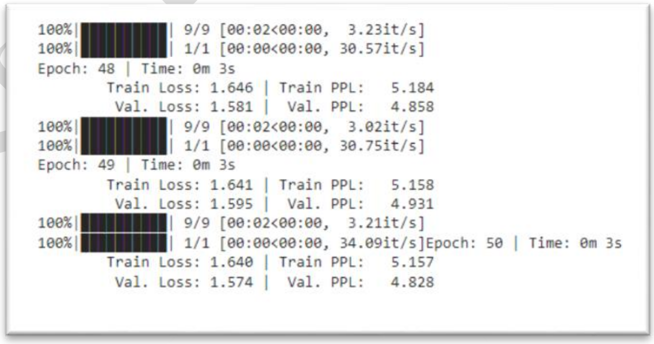


```
# Interfaz gráfica con dos espacios para escribir y un boton
from tkinter import *
root = Tk()
root.columnconfigure(0, weight=0)
root.columnconfigure(1, weight=1)
root.rowconfigure(2, weight=1)
Label(root, text="Nombre:").grid(row=0, column=0)
Label(root, text="Apellido:").grid(row=1, column=0)
Entry(root).grid(row=0, column=1, sticky=E+W)
Entry(root).grid(row=1, column=1, sticky=E+W)
Button(root, text="Aceptar").grid(pady=10,
                                  padx=10,
                                  row=3,
                                  column=0,
                                  columnspan=2,
                                  sticky=S+N+E+W)
root.mainloop()

# Interfaz que da a entender el registro y botón Aceptar
from tkinter import *
root = Tk()
root.columnconfigure(0, weight=0)
root.columnconfigure(1, weight=1)
root.rowconfigure(2, weight=1)
Label(root, text="Nombre:").grid(row=0, column=0)
Label(root, text="Apellido:").grid(row=1, column=0)
Entry(root).grid(row=0, column=1, sticky=E+W)
Entry(root).grid(row=1, column=1, sticky=E+W)
Button(root, text="Aceptar").grid(pady=10,
                                  padx=10,
                                  row=3,
                                  column=0,
                                  columnspan=2,
                                  sticky=S+N+E+W)
root.mainloop()
```

Fig 13 Two records compose the training dataset for the automatic generation of source code for graphical interfaces.

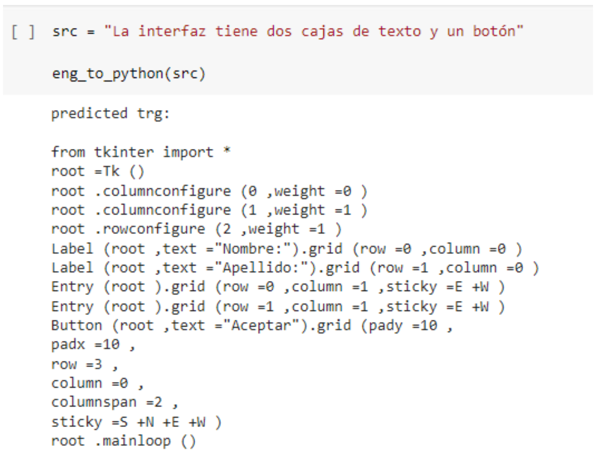
In Figure 13, the first line with the pound sign shows the description of the GUI by the user using natural language, and the following lines show the source code that meets the requirements requested by the user. Once the construction of the dataset is finished, deep neural network training is carried out to obtain a model that processes the natural language related to the GUI description to be converted to Python code using the Tkinter module. The training was tested with a recurrent neural network architecture on the dataset allowing for evaluation accuracy metrics both in training and validation: loss metric (function loss metric) yielding a value of 1.64 in the training stage and a value of 1.57 in the validation stage. Another metric that was evaluated is the perplexity generating values of 5.15 in the training stage and a value of 4.82 in the validation stage; this metric allows for measuring how well a probability distribution or probability model predicts a sample. The values achieved by the metrics in both the training and validation processes allow us to affirm that the model avoided overfitting since a decrease in the two metrics is observed in these stages. Other hyperparameters of interest that were applied in the training of the neural network were: 50 epochs, three encoding layers, and three decoding layers. The training took 10 minutes and was executed through the Google Colab tool (See Figure 14).



```
100%|██████████| 9/9 [00:02<00:00,  3.23it/s]
100%|██████████| 1/1 [00:00<00:00, 30.57it/s]
Epoch: 48 | Time: 0m 3s
	Train Loss: 1.646 | Train PPL:   5.184
	 Val. Loss: 1.581 |  Val. PPL:   4.858
100%|██████████| 9/9 [00:02<00:00,  3.02it/s]
100%|██████████| 1/1 [00:00<00:00, 30.75it/s]
Epoch: 49 | Time: 0m 3s
	Train Loss: 1.641 | Train PPL:   5.158
	 Val. Loss: 1.595 |  Val. PPL:   4.931
100%|██████████| 9/9 [00:02<00:00,  3.21it/s]
100%|██████████| 1/1 [00:00<00:00, 34.09it/s]Epoch: 50 | Time: 0m 3s
	Train Loss: 1.640 | Train PPL:   5.157
	 Val. Loss: 1.574 |  Val. PPL:   4.828
```

Fig 14 Results of training on Jupyter notebooks in Google Colab.

Finally, Figures 15 and 16 show the operation and testing of the trained model for the automatic generation of source code for the generation of graphical interfaces in Python using the Tkinter module. Figure 15 shows the sentence sent to the neural network, which constitutes the request to create a graphical interface: "The interface has two text boxes and a button" when executing the model, it can be observed how the neural network responds with a Python code that when executed on an interpreter its result is visualized in Figure 15.



```
[ ]  src = "La interfaz tiene dos cajas de texto y un botón"

     eng_to_python(src)

     predicted trg:

     from tkinter import *
     root =Tk ()
     root .columnconfigure (0 ,weight =0 )
     root .columnconfigure (1 ,weight =1 )
     root .rowconfigure (2 ,weight =1 )
     Label (root ,text ="Nombre:").grid (row =0 ,column =0 )
     Label (root ,text ="Apellido:").grid (row =1 ,column =0 )
     Entry (root ).grid (row =0 ,column =1 ,sticky =E +W )
     Entry (root ).grid (row =1 ,column =1 ,sticky =E +W )
     Button (root ,text ="Aceptar").grid (pady =10 ,
     padx =10 ,
     row =3 ,
     column =0 ,
     columnspan =2 ,
     sticky =S +N +E +W )
     root .mainloop ()
```

Fig 15 Testing of the trained model on a user request for the creation of a graphical user interface.

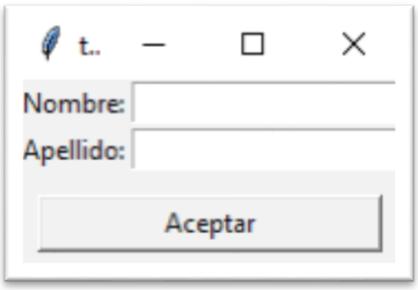


Fig 16 Result of running the code generated by the neural network model on a Python interpreter.

V. CONCLUSIONS

A neural network model is trained by managing to process the natural language related to the request for creating graphical interfaces using the Python programming language to automatically generate source code that can be executed through the Python interpreter. The model achieves loss and perplexity values in the validation stage of 1.57 and 4.82, respectively; furthermore, these values are lower than those achieved in the training stage, thus corroborating that the model avoided overfitting.

A dataset was constructed using the help of 115 Systems Engineering students from different semesters, who described graphical interfaces using natural language. Subsequently, the dataset feeds a recurrent neural network model to create a model for automatic source code generation. In future work, the authors propose to enrich the dataset with more robust graphical interfaces and to take the trained neural network architecture to generate source code in web components. To strengthen the research results, it is recommended to enrich the dataset with more robust graphical interfaces; in the project, GUIs use a maximum of four widgets.

ACKNOWLEDGEMENTS

We thank the Corporación Universitaria Comfacauca – Unicomfacauca, Faculty of Engineering, MIND Research Group - Popayán.

REFERENCES

[1] “10 Breakthrough Software Development Trends in Coming Years | Computools.” https://computools.com/software-development-trends/ (accessed Nov. 28, 2020).
[2] “7 Software Development Trends 2020 Revealed | Hacker Noon.” https://hackernoon.com/software-development-trends-2020-revealed-spi305m (accessed Nov. 28, 2020).

[3] J. Zhu and M. Shen, “Research on Deep Learning Based Code Generation from Natural Language Description,” in 2020 IEEE 5th International Conference on Cloud Computing and Big Data Analytics, ICCCBDA 2020, Apr. 2020, pp. 188–193, doi: 10.1109/ICCCBDA49378.2020.9095560.

[4] “A Guide to Code Generation | Java Code Geeks - 2020.” 2018, Accessed: Nov. 28, 2020. [Online]. Available: https://tomassetti.me/code-generation/.

[5] S. Shim, P. Patil, R. R. Yadav, A. Shinde, and V. Devale, “DeeperCoder: Code Generation Using Machine Learning,” in 2020 10th Annual Computing and Communication Workshop and Conference, CCWC 2020, Jan. 2020, pp. 194–199, doi: 10.1109/CCWC47524.2020.9031149.

[6] T. H. M. Le, H. Chen, and M. A. Babar, “Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges,” ACM Comput. Surv., vol. 53, no. 3, Feb. 2020, doi: 10.1145/3383458.

[7] L. Wang et al., “Latent predictor networks for code generation,” in 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016 - Long Papers, Mar. 2016, vol. 1, pp. 599–609, doi: 10.18653/v1/p16-1057.

[8] M. Rabinovich, M. Stern, and D. Klein, “Abstract syntax networks for code generation and semantic parsing,” in ACL 2017 - 55th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference (Long Papers), Apr. 2017, vol. 1, pp. 1139–1149, doi: 10.18653/v1/P17-1105.

[9] P. Yin and G. Neubig, “A syntactic neural model for general-purpose code generation,” in ACL 2017 - 55th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference (Long Papers), 2017, vol. 1, pp. 440–450, doi: 10.18653/v1/P17-1041.

[10] A. Stehnii and R. Hryniv, “Generation of code from text description with syntactic parsing and Tree2Tree model,” 2018.

[11] P. Yin and G. Neubig, “TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation,” in EMNLP 2018 - Conference on Empirical Methods in Natural Language Processing: System Demonstrations, Proceedings, Oct. 2018, pp. 7–12, doi: 10.18653/v1/d18-2002.

[12] S. Gulwani, “Dimensions in Program Synthesis,” in PPDP’10 - Proceedings of the 2010 Symposium on Principles and Practice of Declarative Programming, 2010, pp. 13–24, doi: 10.1145/1836089.1836091.

[13] M. Jaderberg, K. Simonyan, A. Zisserman, and K. Kavukcuoglu, “Spatial transformer networks,” in Advances in Neural Information Processing Systems, Jun. 2015, vol. 2015-Janua, pp.

Daniel Esteban Arenas-Varela, Systems Engineer. Member of the Artificial Intelligence & Big Data Research Group. https://orcid.org/0000-0002-8576-5381

Julián Fernando Muñoz-Ordóñez Physical Engineer, Master in Computer Science, PhD student at University of Cauca. Auxiliar Professor Corporación Universitaria Comfacauca – Unicomfacauca. Popayán, Colombia. Director of the Artificial Intelligence & Big Data Research Group, Junior Researcher. https://orcid.org/0000-0001-9393-6139