International Journal of Web of Multidisciplinary Studies
(Peer-Reviewed, Open Access, Fully Refereed International Journal)
website: https://ijwos.com
Volume 3 Issue 3 , March 2026
E-ISSN : 3049-2424
DOI: https://doi.org/10.71366/ijwos

SYNTAXFLEXI AI

Abhijeet Todmal[1], Rajvardhan Ughade[2], Yashraj Riakar[3], Vedang Kulkarni[4]
[1,2,3,4] *Department of Computer Engineering, JSPM's RSCOE Faculty of Engineering, tathawade*

Article Info

Article History:

Published:13 March 2026

Publication Issue:
Volume 3, Issue 3
March-2026

Page Number:
257-267

Corresponding Author:
Abhijit Todmal

Abstract:

Basically, a Code Converter is a Machine Learning– based model used to convert source code from one programming language to another. Through this model, we can generate multiple new codes using different source languages and target languages. This system changes the syntax and structure of the code while preserving its original logic and behavior. The Code Converter is a Machine Learning, Deep Learning , and Artificial Intelligence– based technique that learns programming patterns from datasets. The input code contains all logical statements, and the output code contains the converted syntax of the target language. This technique bridges the gap between human programming logic and automated code transformation by introducing neural network models.

Keywords: Code Converter, Machine Learning, Deep Learning, Artificial Intelligence, Neural Networks, Source Code, Target Code, Programming Languages, Syntax Conversion, Semantic Analysis, Code Translation, Automated Code Generation

1. Introduction

Transferring the logic from one programming language into another can be considered a problem of **code translation**. In code translation, the goal is to generate equivalent code from a source language while constraining the conversion process in order to preserve the original program logic, structure, and functionality. Algorithms used for code conversion aim to generate syntactically correct and logically equivalent programs by learning patterns from a given source code. Most previous code conversion approaches rely on rule-based or pattern-matching techniques while using different methods to preserve the logical structure of the target program. For instance, some approaches introduce an intermediate representation that includes features of the source code such as control flow and data dependencies to constrain the code translation process. Other methods use code analogies to transfer logic from an already converted program into a target language. These approaches focus on preserving high-level logical information while maintaining the overall structure of the target code. This process can be further enhanced by incorporating semantic context and programming constructs to improve the accuracy of code translationAlthough existing code conversion algorithms achieve promising results, they all suffer from a common fundamental limitation: they often rely only on surface-level syntactic features of the source code. Ideally, a code conversion algorithm should be able to extract the **semantic logic** of the source program (such as control flow, data dependencies, and functional intent) and then guide the code translation process to generate equivalent code in the target programming language. Therefore, a fundamental requirement is to develop code representations that independently model variations in **program logic** and **language-specific syntax**, which can then be combined to generate correct and efficient target code.

With the increasing use of intelligent code transformation technologies, automated code conversion has become more widespread, especially for migrating programs from one language to another while ensuring that the core logic remains unchanged without major functional differences. With the rise of deep learning, neural network–

based models have been applied to code conversion tasks, attracting significant attention from researchers. Code conversion has become an important research direction in recent years. Initially, code translation relied on rule-based systems, but later, neural network–based program representation and learning techniques introduced new possibilities for intelligent code conversion. Although these approaches show strong potential, several challenges remain unresolved. Improving the efficiency and accuracy of code conversion algorithms while maintaining high-quality translated code is a key research direction, particularly for real-time applications. This project provides an overview of recent developments in machine learning–based code conversion techniques.

Literature Survey:

In [1], progress in automatic code translation systems is limited by the accuracy and efficiency of semantic code analysis. Here, the authors present an intelligent code conversion framework for efficient semi-automated transformation of large-scale source codebases across programming languages. By combining learned program representations with neural network–based translation models, the system enables accurate code conversion with significantly reduced manual effort compared to traditional rule-based methods and existing conversion tools.

In [2], human programmers demonstrate remarkable ability to understand and rewrite code logic across different programming languages, even under variations in syntax, structure, and programming paradigms. This capability is mediated by an abstract representation of program semantics. In parallel, recent advances in machine learning have led to high-performing models for code understanding and translation using deep neural networks (DNNs).

In [3], recent methods have been proposed that perform automated code translation using neural network architectures such as sequence-to-sequence and transformer-based models. These approaches are promising as they can generate high-quality translated code in many cases. However, these methods still face challenges related to translation accuracy, logical consistency, parameter tuning, and limited user control. This work presents an improved machine learning–based code conversion pipeline that addresses these limitations and enhances translation reliability and efficiency.

Deep Code Representation

The results presented in this work are generated using a deep neural network trained for program understanding and code translation tasks. The network is designed to learn abstract representations of source code, capturing both syntactic structure and semantic behavior. The model is normalized by scaling internal parameters so that the average activation of each hidden unit remains stable across different code samples. Such normalization does not affect the output of the model because it uses non-linear activation functions and does not rely on explicit normalization across representation layers. Fully connected layers that are specific to classification tasks are not used, as the focus is on learning transferable code representations rather than prediction labels.

Content (Logic) Representation

In general, each layer of the network defines a non-linear transformation that captures increasingly complex programming features as the depth of the network increases. Therefore, a given input source code $x$is encoded at each layer of the neural network by the corresponding feature representation. These representations capture elements such as control flow, variable dependencies, and functional logic.

To ensure that the converted code preserves the original program logic, we define a content (logic) loss as the squared error between the feature representations of the source code and the generated target code at a selected layer.

Style(Syntax) Representation:

To obtain a representation of the **syntax/style** of an input program, we use a feature space designed to capture programming patterns and language-specific structures. This feature space can be constructed using the hidden representations learned at different layers of the neural network. It consists of the **correlations between feature representations**, where the expectation is taken over the entire code sequence rather than spatial dimensions.

These feature correlations are represented using a **correlation matrix** $G_l \in \mathbb{R}^{N_l \times N_l}$, where each element G_{ij}^lmeasures the similarity between the feature vectors corresponding to programming constructs $i$and $j$at layer l.

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l.$$

Style Transfer (Code Conversion)

To transfer the **syntax/style** of a target programming language $\sim a$onto a source code $\sim p$, we synthesize a new program that simultaneously matches the **content (logic) representation** of $\sim p$ and the **style (syntax) representation** of $\sim a$. This is achieved by jointly minimizing the distance between the feature representations of an initially random code representation and the stored content and style representations.

The generated code is optimized using the content representation extracted from the source code at a selected network layer and the syntax/style representations of the target language defined across multiple layers of the neural network. Several well-studied programming examples are used to validate the approach. The converted code is created by simultaneously preserving the logical structure of the source program and applying the syntactic rules of the target programming language.

Stability Improvements in Machine Learning–Based Code Conversion

Recent machine learning–based code conversion methods can be broadly classified into two categories. The first category follows an **optimization-based approach**, where the converted code is generated by solving an optimization problem that minimizes multiple loss functions. These methods often produce highly accurate results but require more computation time for converting each program.

The **combined loss function** is defined to balance different objectives of code conversion. Let the total loss be:

$$L(W, c_{1:T}, s) = \sum_{t=1}^{T} (\lambda_c L_C(p_t, c_t) + \lambda_s L_s(p_t, s) + \lambda_t L_t(p_{t-1}, p_t))$$

Where:

- $\lambda_c, \lambda_s, \lambda_t$are weighting parameters that control the importance of each loss term
- L_c= Content (Logic) Loss
- L_s= Syntax/Style Loss
- L_t= Translation or Structural Consistency Loss

2. Content (Logic) Loss

The **content loss** ensures that the logical behavior of the source code is preserved in the converted code and is defined as the squared difference between feature representationsNST Evaluation Metrics: Several evaluation metrics are used to assess the quality and stability of machine learning–based code conversion models:

- Fréchet Distance–based Metrics:
  Similar to the Fréchet Inception Distance (FID) used in image generation, feature distributions of real and generated code are approximated using multivariate Gaussian distributions. The Fréchet distance between these distributions is used to evaluate the realism and correctness of generated code.

- Intersection-over-Union (IoU):
  Some studies use IoU-like metrics to evaluate how accurately structural elements such as functions, classes, and control blocks are preserved during code conversion.

- Perceptual Distance (Embedding Distance):
  This metric measures the difference between consecutive generated code representations in the latent space. It helps determine whether the conversion process follows a smooth transformation path without abrupt logical changes.

- Structural Consistency Error:
  This metric measures the difference between expected and generated control-flow or data-flow structures, helping to evaluate the stability and correctness of converted code across large programs.

Research Gaps

The research gaps in machine learning–based code conversion can be grouped into three major categories: architecture-related, platform-related, and dataset-related.

Platform-Related Research Gaps

a. Native Mobile Code Conversion:
Implementing real-time code conversion directly on mobile devices remains a challenge due to limited computational resources, memory constraints, and power efficiency issues. Most current solutions rely on cloud-based processing rather than on-device execution.

b. Use of Federated Learning:
Federated learning presents a promising research direction for code conversion systems, particularly for mobile and low-power devices. By training models across distributed devices without sharing raw code data, federated learning can improve privacy, reduce data transfer costs, and overcome hardware limitations. However, its application to code conversion tasks is still in the early stages and requires further exploration.

Models

Convolutional Neural Networks (CNNs) / Deep Neural Networks for Code Conversion

Neural networks form the core building blocks used in machine learning–based code conversion systems. These networks are used to extract meaningful features from both the source code (content/logic) and the target programming language (syntax/style). Popular deep learning architectures such as encoder–decoder networks, CNN-based models, and transformer models are commonly used for this purpose.

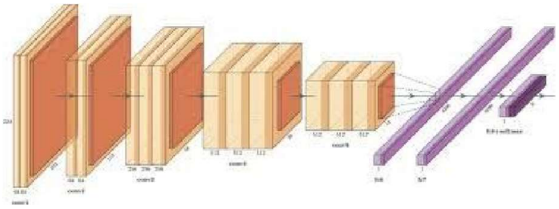


Fig: Convolutional Neural Networks (CNNs):

Content and Style Loss Functions: Neural style transfer uses loss functions to quantify the difference between the content and style of the input images and the generated image.

Content Loss Function: The content loss function quantifies how well the content of the generated image matches the content of the reference content image. It is

x= source code

- $\hat{x}$= generated (converted) code

F_{ij}= feature representation at a selected network layer

A neural network consists of multiple layers, where each layer learns increasingly complex representations of the input code. In the lower layers, the network learns simple programming constructs such as keywords, operators, and syntax patterns. As the depth of the network increases, higher layers capture more complex structures such as loops, conditional logic, function calls, and program flow.

Each layer applies learned filters to the input representation, producing feature maps that progressively encode richer semantic and syntactic information. These extracted features are later used to guide the code conversion process.

Optimization Using Gradient Descent

To generate accurate converted code, an optimization method (usually gradient descent) is employed to minimize the combined loss function, which includes content (logic) loss and syntax/style loss.

The gradient of the loss function with respect to the generated code is computed, and the code representation is updated iteratively in order to reduce the total loss and improve conversion quality.

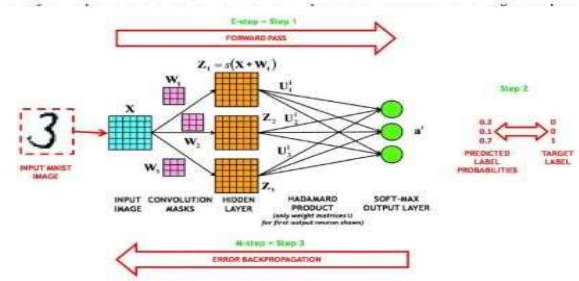
Backpropagation: Backpropagation, short for **backward propagation of errors**, is a supervised learning algorithm used to train neural networks. It is a critical part of the training process because it allows the model to adjust its internal parameters (weights and biases) to minimize prediction errors during code conversion.

Working of Backpropagation:

1. The source code is passed forward through the network to generate converted code.
2. The loss is calculated by comparing the generated code with the expected target code.
3. The error is propagated backward through the network layers.
4. Gradients are computed for each parameter using the chain rule.
5. Model parameters are updated to reduce the error.

This process is repeated iteratively until the model achieves stable and accurate code conversion.

Here's how backpropagation works:



3. Algorithms

Here are the key algorithms and techniques used in a typical Machine Learning–based Code Converter:

Pre-trained Neural Networks

Pre-trained deep learning models such as encoder–decoder networks, transformer models, or neural language models are commonly used as the backbone for code conversion. These models are trained on large-scale code datasets and act as feature extractors for both program logic (content) and language syntax (style).

Content (Logic) Loss Function

The content loss measures the difference between the logical representations of the generated (converted) code and the source code. It is typically computed using mean squared error (MSE) between feature representations extracted from selected layers of the neural network.

Style (Syntax) Loss Function

The style loss evaluates how well the generated code follows the syntax and structural patterns of the target programming language. This loss is calculated by comparing correlation statistics of feature representations between the generated code and target-language examples.

Structural Regularization

To ensure syntactic correctness and structural consistency, regularization techniques are applied. These techniques penalize invalid token sequences, broken control structures, and abrupt changes in code structure, helping to produce stable and readable converted code.

Optimization Algorithm

Optimization algorithms such as Gradient Descent, Adam, or L-BFGS are used to iteratively update the model parameters and generated code representations. The goal is to minimize the combined content and style loss while maintaining correctness.

Hyperparameters

Tuning hyperparameters such as the weights assigned to content and style loss, learning rate, batch size, and number of training iterations is critical to achieving an optimal balance between logic preservation and syntax accuracy.

Initialization

The initialization of the generated code representation affects the final output. Common initialization strategies include starting from a random code embedding, a partially translated version, or a template-based initialization.

Multi-Language Code Conversion

Advanced code converters support multi-language translation, enabling conversion between multiple programming languages using a single trained model. This allows combining features learned from various languages and improves generalization.

4. Tools

TensorFlow

TensorFlow is a widely used open-source deep learning framework developed by Google. It provides powerful APIs for building, training, and deploying machine learning models for code conversion tasks.

PyTorch

PyTorch is a flexible deep learning framework developed by Meta AI. Its dynamic computation graph makes it especially suitable for research-oriented projects such as machine learning–based code translation.

Keras

Keras is a high-level neural network API that runs on top of TensorFlow. It simplifies model development and experimentation, making it suitable for implementing code conversion architectures.

Prebuilt Code Translation Tools

Several prebuilt tools and research implementations exist for automated code translation, enabling users to perform code conversion with minimal manual configuration.

IDEs and Code Editors

Integrated Development Environments (IDEs) and code editors are often used alongside ML models to visualize, test, and validate the converted code output.

Advantages

Automation:
Machine learning–based code converters automate the process of translating source code from one programming language to another, reducing manual effort and saving significant development time compared to rewriting code manually.

Improved Productivity:
By automatically converting code, developers can focus more on logic optimization and feature development rather than repetitive syntax rewriting. This makes code migration faster and more efficient.

Learning and Educational Support:
Code converters can be used as educational tools to help students and beginners understand how programming concepts and logic are implemented across different programming languages.

Consistency and Accuracy:
Machine learning models maintain logical consistency during conversion, reducing human errors that commonly occur in manual code translation.

Scalability:
The system can be applied to large codebases and multiple programming languages, making it suitable for software migration, legacy system modernization, and cross-platform development.

Implementation:
Input:

Output:

Technology Used:

- JavaScript – Used for frontend development and user interaction

  Python – Used for implementing machine learning models and backend logic

  Artificial Intelligence – Enables intelligent code understanding and transformation

  Deep Learning – Used to learn complex patterns in source and target programming languages

  Neural Network Algorithms – Used for feature extraction and code translation

  Source Code – Input code provided by the user

  Target Language Syntax – Represents the programming style of the target language

Generated Code – Output code produced after conversion

Applications

Automated Code Translation:
Machine learning–based code converters allow developers to automatically convert source code from one programming language to another while preserving the original logic. This is useful for migrating legacy systems to modern programming languages.

Software Migration and Modernization:
Code converters help organizations upgrade old applications by translating them into newer languages or frameworks, reducing redevelopment time and cost.

Learning and Education:
Code conversion tools are valuable in educational environments, helping students understand how the same programming logic is implemented across different languages

Cross-Platform Development:
Developers can use code converters to adapt applications for different platforms by converting code between languages used on different systems.

Developer Productivity Tools:
Integrated code conversion tools assist programmers by reducing manual rewriting effort, improving development speed, and minimizing syntax-related errors.

5. Conclusion

We conclude that a machine learning–based Code Converter is a powerful and versatile technique that offers significant advantages for a wide range of practical and educational applications. It enables the automatic conversion of source code from one programming language to another while preserving the original program logic and functionality. By reducing manual effort and minimizing human errors, code conversion systems improve development efficiency and productivity.

This approach is useful for developers, learners, researchers, and organizations involved in software migration and modernization. Furthermore, the integration of advanced deep learning techniques and generative models, such as GAN-inspired architectures, enhances code translation quality by providing better control over syntax, structure, and logical consistency. With continued research and improvements, machine learning–based code converters have the potential to become highly reliable tools for cross-language software development.

References

[1] M. Berning, K. M. Boergens, and M. Helmstaedter, "SegEM: Efficient Image Analysis for High-Resolution Connectomics," *Neuron*, vol. 87, no. 6, pp. 1193–1206, Sept. 2015.

[2] C. F. Cadieu, H. Hong, D. L. K. Yamins, N. Pinto, D. Ardila, E. A. Solomon, N. J. Majaj, and J. J. DiCarlo, "Deep Neural Networks Rival the Representation of Primate IT Cortex for Core Visual Object Recognition," *PLoS Computational Biology*, vol. 10, no. 12, e1003963, Dec. 2014.

[3] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs," *arXiv preprint arXiv:1412.7062*, Dec. 2014.

[4] M. Cimpoi, S. Maji, and A. Vedaldi, "Deep Filter Banks for Texture Recognition and Segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3828–3836, 2015.

[5] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition," *arXiv preprint arXiv:1310.1531*, Oct. 2013.

[6] L. Gatys, A. Ecker, and M. Bethge, "A Neural Algorithm of Artistic Style," *Journal of Vision*, vol. 16, no. 12, 2016.

A. Mordvintsev, C. Olah, and M. Tyka, "Inceptionism: Going Deeper into Neural Networks," 2015.

[7] G. Berger and R. Memisevic, "Incorporating Long-Range Consistency in CNN-Based Texture Generation," *arXiv preprint arXiv:1606.01286*, 2016.

[8] E. Risser, P. Wilmot, and C. Barnes, "Stable and Controllable Neural Texture Synthesis and Style Transfer Using Histogram Losses," *arXiv preprint arXiv:1701.08893*, 2017.

[9] C. Castillo et al., "Son of Zorn's Lemma: Targeted Style Transfer Using Instance-Aware Semantic Segmentation," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017.

[10] Y. Li et al., "Demystifying Neural Style Transfer," *arXiv preprint arXiv:1701.01036*, 2017.

[11] C. Li and M. Wand, "Combining Markov Random Fields and Convolutional Neural Networks for Image Synthesis," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[12] J. Johnson, A. Alahi, and L. Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution," in *European Conference on Computer Vision (ECCV)*, Springer, 2016.

[13] D. Ulyanov et al., "Texture Networks: Feedforward Synthesis of Textures and Stylized Images," in *International Conference on Machine Learning (ICML)*, 2016.

[14] C. Li and M. Wand, "Precomputed Real-Time Texture Synthesis with Markovian Generative Adversarial Networks," in *European Conference on Computer Vision (ECCV)*, Springer, 2016.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems (NIPS)*, pp. 1097–1105, 2012.

[16] M. Kümmerer, L. Theis, and M. Bethge, "Deep Gaze I: Boosting Saliency Prediction with Feature Maps Trained on ImageNet," in *ICLR Workshop*, 2015.

[17] J. Long, E. Shelhamer, and T. Darrell, "Fully Convolutional Networks for Semantic Segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3431–3440, 2015.

[18] X. Liu, "Research on Neural Style Transfer," *Journal of Physics: Conference Series*, vol. 2079, 012029, 2021.

[19] A. Singh, "Neural Style Transfer: A Critical Review," 2021.