Journal on Software and Systems Modeling manuscript No.
(will be inserted by the editor)

A Generic LSTM Neural Network Architecture to Infer Heterogeneous Model Transformations

Loli Burgueño · Jordi Cabot · Shuai Li · Sébastien Gérard

Received: date / Accepted: date

Abstract Models capture relevant properties of systems. During the models' life-cycle, they are subjected to manipulations with different goals such as managing software evolution, performing analysis, increasing developers' productivity, and reducing human errors. Typically, these manipulation operations are implemented as model transformations. Examples of these transformations are *(i)* model-to-model transformations for model evolution, model refactoring, model merging, model migration, model refinement, etc., *(ii)* model-to-text transformations for code generation and *(iii)* text-to-model ones for reverse engineering.

These operations are usually manually implemented, using general-purpose languages such as Java, or domain-specific languages (DSLs) such as ATL or Acceleo. Even when using such DSLs, transformations are still time-consuming and error-prone.

We propose using the advances in artificial intelligence techniques to learn these manipulation operations on models and automate the process, freeing the developer from building specific pieces of code. In particular, our proposal is a generic neural network architecture suitable for heterogeneous model transformations. Our architecture comprises an encoder-decoder LSTM (Long Short-Term Memory) with an attention mechanism. It is fed with pairs of input-output examples and, once trained, given an input, automatically produces the expected output.

We present the architecture and illustrate the feasibility and potential of our approach through its application in two main operations on models: model-to-model transformations and code generation. The results confirm that neural networks are able to faithfully learn how to perform these tasks as long as enough data is provided and no contradictory examples are given.

Keywords model manipulation · code generation · model transformation · artificial intelligence · machine learning · neural networks

Loli Burgueño
Open University of Catalonia, IN3, Spain
Institut LIST, CEA, Université Paris-Saclay, France
E-mail: lburguenoc@uoc.edu

Jordi Cabot
ICREA, Spain
Open University of Catalonia, IN3, Spain
E-mail: jordi.cabot@icrea.cat

Shuai Li
Institut LIST, CEA, Université Paris-Saclay, France
E-mail: shuai.li@cea.fr

Sébastien Gérard
Institut LIST, CEA, Université Paris-Saclay, France
E-mail: sebastien.gerard@cea.fr

1 Introduction

In software development, data, structured as models, are manipulated on a daily basis through systematic model manipulation operations. Automating such operations, typically in the form of model transformations, can reduce the time-to-market of project development and improve its quality. Usually, a model-driven project involves several consecutive transformations. Intermediate steps are often implemented as model-to-model transformations, each one taking as input the output of the previous step as part of a continuous refinement process from high-level models to platform-specific ones. One last step consists in a model-to-text transformation that takes this low-level models and generates a textual output, i.e. the final code, as a result.

This automatic code generation step aims to limit tedious tasks, reduce the chances of programming er-

rors, and improve the quality of the code, hence minimizing the maintenance cost. We also have the reverse scenario where we want to generate the models corresponding to a (legacy) system. Starting with a text-to-model transformation we generate an initial set of models that are then abstracted out via additional model-to-model transformations. All these heterogeneous types of model transformations play a key role in any model-driven engineering activity.

Domain-specific languages and tools, such as ATL [1], Acceleo [2], and Xtext [3], aim to accelerate the writing of model transformations by offering facilities to handle model querying, model serialization, etc.

Nevertheless, creating model transformations remains a challenging task that requires a high-level expertise, competences in language engineering, and extensive domain knowledge [4]. Moreover, developers may be reluctant to adopt some automatic model manipulators, especially model-to-text ones because they do not trust them—usually considering that it cannot be as performant as the manually performed tasks or because the output artefacts of such automatic tools look artificial and foreign to them [5]. For instance, they do not follow the company's coding style.

As systematic model transformation is a key aspect of Model-Based Engineering (MBE), we argue that the lack of competences, the maintenance cost of model transformation operations, and the defiance from developers, hinder the adoption of MBE at enterprise scale [4, 6]. The work presented in this paper aims at fostering model-driven engineering at enterprise scale, providing a stepping stone towards a next generation of cognitive model-based engineering tools.

The proposal we present in this paper aims at overcoming aforementioned issues. We propose to automatically infer heterogeneous model transformations from only input-output examples. Not only are the inferred transformations able to automatically produce expected output artifacts, but they are able to oblige developers to comply to company or project standards, key to the solution's adoption and valuable to ensure quality [7].

Given the recent improvements in artificial intelligence, and especially in supervised machine learning, we believe such intelligent model manipulations can be "implemented" as an encoder-decoder [8] with an attention mechanism [9] trained with pairs of input-output data, where both the encoder and decoder are LSTM (Long Short-Term Memory) neural networks [10].

We rely on existing works on Machine Learning (ML) and neural machine translation [11, 12] to propose an ML-based generic architecture for heterogeneous model transformations. This architecture consolidates, generalizes and expands on the the technical details of our previous experience with the specific problem of applying machine learning for model-to-model transformations [13]. Beyond generalizing and validating this previous result to more types of model transformations, this paper adds a second case study (for model-to-text transformations) and includes several technical improvements in the configuration and evaluation of the network to optimize the learning process.

Our proposal is among the first ones to explore and bridge two different fields: model transformations and artificial neural networks. The feasibility of our approach is shown by applying it to realistic scenarios and projects, while discussing the limitations of neural networks in this domain.

The rest of the paper is organized as follows. Section 2 describes basic concepts related to neural networks, and the encoder-decoder LSTM network we use in this paper. Section 3 presents our neural network architecture that automatically infers model manipulation operations. In Section 4, the architecture is instantiated and evaluated on a model-to-model transformation and a code generation scenario. Section 5 presents the limitations of neural networks to deal with model manipulation activities. Related work is discussed in Section 6. Finally, we conclude in Section 7 with future work.

2 Background and Motivation

The approach presented in this paper uses the advances in Artificial Neural Networks (ANN) that we present in this section. In particular, we give some background information on the so-called neural networks and on the specific type we have used: Long Short-Term Memory (LSTM), and motivate why they are our choice.

2.1 Introduction to Neural Networks

An ANN can be seen as a structure composed by neurons (also called cells) with directed connections between them. Each neuron is a mathematical function that receives a set of values through its input connections and computes an output value that is transferred to another neuron through its output connection(s). Two specific types of neurons are the *input* and *output* neurons which do not have input predecessors or successors respectively and serve only as input and output interfaces of the ANN. Connections have associated weights (i.e., real numbers) that the neurons use and that are adjusted during the learning process with the purpose to increase or decrease the strength of the connection.

To illustrate all these concepts, Figure 1 shows an example of the simplest neural network: a perceptron neuron [14,15] (in green) that receives an input x, which is a vector with three components, through three input cells, and generates an output which is a single value.

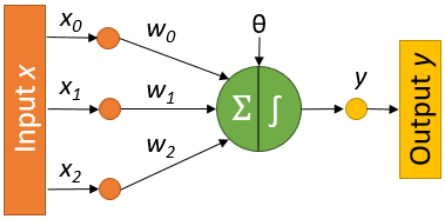


Fig. 1 Visual representation of a perceptron.

The perceptron propagation rule is given by the following equation (note that θ is a hyperparameter that needs to be provided):

$$y_i = \begin{cases} 1 & \text{if } \sum w_{ij}x_j > \theta_i \\ 0 & \text{if } \sum w_{ij}x_j <= \theta_i \end{cases}$$

Most ANNs are part of a supervised learning procedure where a set of example input-output pairs are used to derive a function that can generate or predict new outputs from completely new inputs. Supervised learning has two main phases: training and predicting (transforming in our case). During training, the input and output pairs from the dataset are used to "teach" the system until it learns the matching patterns of input/output. Once trained, we can give the ANN an input and it produces its corresponding output.

For instance, in the case of the perceptron, before training, the weights (w_{ij}) are randomly initialized. During training, for each input-output pair (x–y), the weights are updated according to the following equation:

$$\Delta w_i = \varepsilon(y'_i - y_i)x_i$$

where ε is a hyperparameter that fixes the learning rate, y'_i is the expected output and y_i is the output that the neural network produces.

These concepts can lead to more powerful neural networks containing more neurons and layers. For instance, using the same type of neurons, a multi-layer perceptron [16]—which is an extension and generalization of its predecessor—can be built as Fig. 2 shows.

Thus, from a mathematical point of view, ANNs are complex functions composed by other functions and equations.

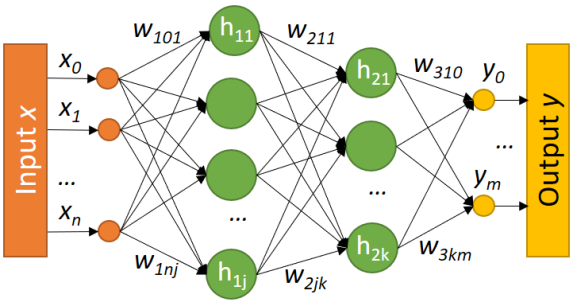


Fig. 2 Visual representation of a Multi-layer Perceptron with two hidden layers (h_1 and h_2).

For the training phase, the dataset is split into three subsets: *training*, *validation* and *test* dataset. The training dataset contains most of the inputs-output pairs and is used to train the ANN (i.e., to adjust the weights of the ANN's connections). The test dataset is only used once the training has finished. Its goal is to check the quality of the ANN's predictions for inputs it has not seen before, and hence to study its accuracy. The accuracy is calculated as the percentage of predictions our model gets right out of the total number of predictions. Thus, it is a number in the range [0; 1]—the closer to 1, the better.

The validation dataset plays a similar role as the test dataset, during the training process. Its goal is to control that the learning process is correct and avoid overfitting[1]. More specifically, the validation dataset is used to check that during the training process (i.e., while neural networks' parameters are being adjusted), any accuracy increase over the training dataset also yields to an accuracy increase over the validation dataset (i.e., the model is not being overfitted to the data on the training dataset). Together with the accuracy, the *loss* is another metric in charge of monitoring the quality of the training process and the overfitting. The *loss* reports the sum of the error made for each example in the dataset, thus it should be as close as 0 as possible. It is expected that a decrease in *training loss* should trigger a decrease in *validation loss*.

A more in-depth introduction to neural networks and all the concepts presented in this section can be found in [17].

2.2 Long Short-Term Memory

Among all the artificial neural network family types and configurations [18] we have chosen Recurrent Neural Networks (RNN) as our subject of study. Unlike its

[1] In statistics, overfitting is the situation in which the ANN is so closely fitted to the training data that it is not able to generalize and make good predictions for new data.

predecessor (feed forward networks, FNN), in RNNs, the neurons are organized in layers with forward connections (i.e., connections to neurons in the next layers) as well as back propagation connections (i.e., to neurons in the same layer or previous layers). This back-propagation mechanism in which the outputs of neurons are fed back into the network again makes the ANN "remember" at a step some information received in the previous steps. This kind of neurons is called a *memory cell* and make the ANNs aware of their context.

Long Short-Term Memory (LSTM) neural networks [10] are a specific kind of RNN composed by LSTM cells that were specifically created to solve the problem of the *vanishing gradients* (used for the backpropagation) from which their predecessors suffer and which hamper the learning of long data sequences. Although, computationally speaking, LSTM neurons are more expensive due to the increased number of operations and complexity of its propagation function[2] they have a longer "memory" than their predecessors, which makes them be able to remember better their context throughout different inputs. For instance, if the operation is to transform lines of code (one at a time), each line of code would be an input for the network. At some point in time, traditional RNNs would be able to remember only parts of the current input (line of code), while LSTMs are able to remember previous lines of code too. Clearly, in our scenario, we may need this long-term memory to remember previous mappings as part of a more complex mapping pattern. Therefore, we have chosen LSTM neural networks as the most suitable networks to solve the problem of translation/transformation.

3 Generic Neural Architecture for Model Transformations

3.1 Encoder-Decoder Architecture

After exhaustively testing a transformation architecture based on single LSTMs, we realized that not even LSTMs alone were enough to generate good results. Instead, we adopted a more complex framework based on an encoder-decoder [19] architecture that has been proven to be the most successful for dealing with translation problems.

This architecture is composed of two LSTM neural networks: one that reads the input data (which is of variable size) and encodes it into a fixed-length numeric vector (called embedding), and a second one that receives this vector and predicts (transforms in our case) the output data, which is again of variable size.

In the literature, most works using this encoder-decoder architecture are applied to sequence-to-sequence transformations, for example, for natural language translation. In those cases, the raw input data that needs to be embedded is a sentence (i.e., a sequence of words).

In our initial experiments, we found out this representation to be too simplistic since we were losing many of the structural information in the models/code. Therefore, we have settled for a more advanced tree-to-tree architecture as depicted in Fig. 3. It is composed by an encoder-decoder enhanced with an attention mechanism. As we said above, both the encoder and decoder are LSTMs. As any supervised learning method, they need to go through a training phase before they are ready to use.

This architecture can be used to train neural networks for heterogeneous model transformations (i.e., model-to-model, model-to-text and text-to-model) with input-output pairs. Unlike in natural language translation, we can take advantage of the more restricted syntax rules of models and/or code to easily derive an abstract-syntax tree (AST) of our software artifacts to feed to our neural networks.

During training, we have to **embed** each AST from the training dataset into a tree of numeric vectors to enable their processing by the networks. In the ASTs, each node contains a word (called *token*). We use the one-hot encoding technique[3] to transform each token into a numeric vector. With this, we obtain a tree of numeric vectors which is the **embedding** with which we feed our architecture.

The **encoder** is a single-layer Tree-LSTM neural network [20] that takes model embeddings and converts them into fixed-sized vectors, as needed by the decoder. As an LSTM is a kind of recursive neural network, it computes its output recursively by reading the input token after token. In our work, our architecture does not deal with sequences of tokens but with trees. Therefore, the order in which the trees are traversed needs to be defined. Although trees with an arbitrary number of nodes can be transformed to embeddings [21], we transformed our trees to binary trees—as Chen [12] proved that they are more effective for translation purposes—using the *left-child right-sibling* representation. Then, given a binary tree with root n_{root} and children t_l and t_r, the encoding of the tree is recursively calculated as the concatenation of the encoding of t_l, t_r and the embedding of n_{root}. The base case is when a child does not exist and then the embedding is a vector of zeros.

[2] We refer the reader to [10] for details about LSMT cells.

[3] `https://www.sciencedirect.com/topics/computer-science/one-hot-encoding`

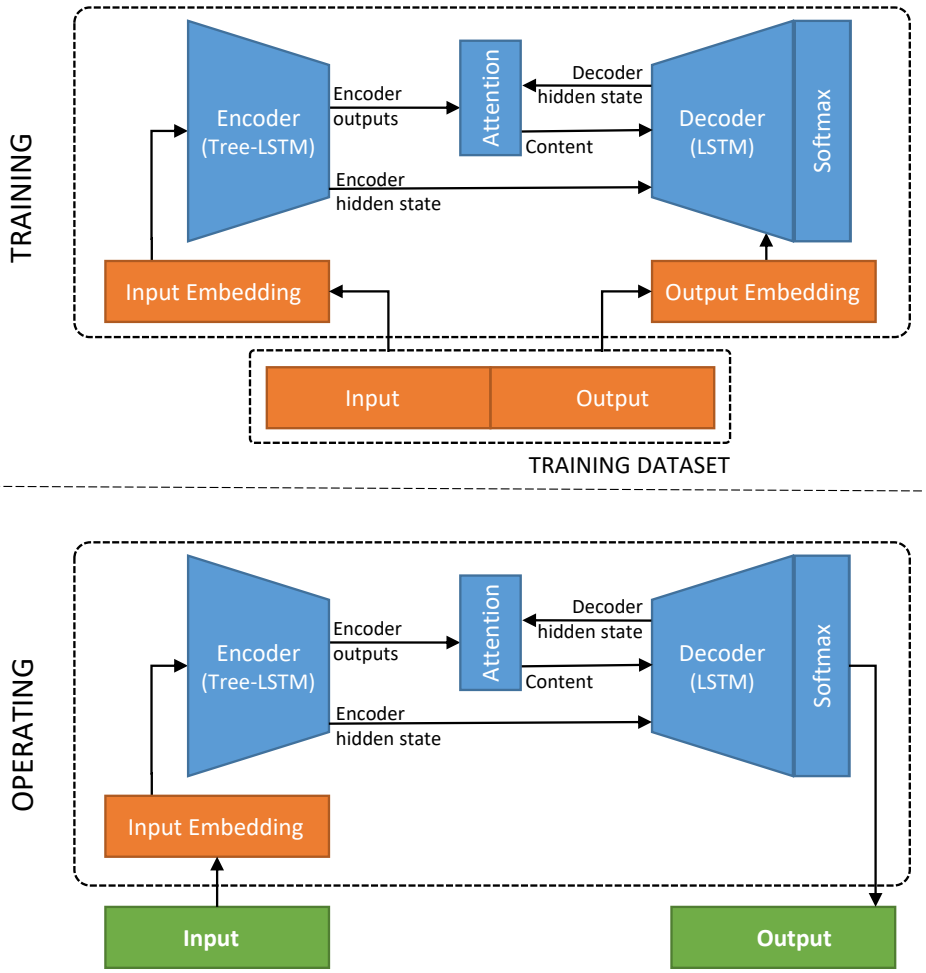


Fig. 3 Neural Network Architecture for Model Manipulation – Components.

The **attention** mechanism helps the decoder recognise the relevant information in the vectorial representation of the input AST at each step. This is, each time the decoder is working on the generation of a node of the output AST, the attention mechanism locates the sub-tree in the input AST with useful information to guide the expansion of the node in which it is working. For example, if the decoder is generating the type of an attribute, the attention mechanism helps it by pointing to the location in which the attribute is defined in the input AST.

The **decoder** is a single-layer LSTM network. During training, it takes the vectors generated by the encoder and attention mechanism, as well as the output embeddings[4] and generates the vectors that compose the output AST. These are vectors of real numbers. This is, the decoder is in charge of performing the actual model transformation. In technical terms, the decoder, as another RNN, generates the output token after token, which in our case is node after node. Our decoder starts building the target tree iteratively starting from the root node. It receives the encoding of the root node, computes its value using the softmax layer and recursively takes the next node as the left child and the following as the right child. It stops expanding each branch when it finds the end-of-sequence token (EOS).

As we said before, these vectors are passed through a neural network for multi-class classification which uses a ***softmax*** activation function [22]. The softmax layer maps each real number to a number in the $(0, 1)$ range in such a way that the sum of all of them is up to 1 and can be interpreted as probabilities. The softmax output is a vector with as many components as tokens there are in the output vocabulary. Then, in each iteration, the component with higher probability is selected and its corresponding token passes to be part of the output of the neural network architecture. This way we obtain the

[4] Note that these output embeddings are available and used only during training. Once the training phase is done, they are neither available nor needed.

actual output representation for a specific input. Note that this softmax neural network is trained, and works together, with the decoder.

Once the training phase finishes, the neural networks are ready to receive inputs and perform the model transformation it has been trained for, generating as output the corresponding expected artifact.

This generic neural architecture must be configured with a number of hyperparameters to optimize the prediction. Data also needs to be pre-processed and post-processed to conform to what is expected by the neural network input layers, and to increase the network's performance. In the next sections, we discuss the optimal hyperparameters for the model transformation scenario, and how data is pre and post processed.

3.2 Hyperparameters

In machine learning, hyperparameters are those parameters that are not learned during training. Instead, they are adjusted by experts to improve the learning process. Choosing the right values has a critical impact on the success and performance of the network. Since there is no rule to choose the best hyperparameters for a specific task [23], using our knowledge of the problem and some experimentation we provide a hyperparametrization by default, which we have found out to be the best for our two cases presented in this paper. For this experimentation, apart from using these two cases, we artificially created datasets covering further model transformations with both simple and complex mappings, small and large input-output examples, small and large datasets, etc. In the following, we provide a brief description of each hyperparameter—note that more detailed explanations can be consulted in [17]—and detail our default hyperparametrization[5].

Neural networks learn by repetition, so the same input-output pairs from the training dataset must be given to the ANNs several times during training. The training phase is divided in **epoch**s. An epoch is the number of times the complete training dataset is passed through the ANNs. In each epoch, the training dataset is randomly shuffled and split into batches of input-output pairs that are passed through the ANN. Each time a **batch** is passed, an iteration is completed. The number of epochs and batch size can be adjusted depending on the size of the dataset to improve the learning process. Nevertheless, we provide a configuration by default which trains with 30 epochs (i.e., the training dataset is passed 30 times through the networks) and batches of 64 units, (i.e., 64 input-output pairs are processed by the neural networks before their internal parameters are updated).

As we already stated, we have used a shallow LSTM network with one single layer for the encoder and a single-layer LSTM network for the decoder. Unlike one may think, increasing the number of layers more than needed (i.e., using deep learning) may lead to overfitting to the training data, resulting in a very poor quality of results after training.

For the **embedding and hidden vector size**, we have selected 256 units. Although always chosen empirically, note that this value needs to always be higher than the vocabulary size. Thus, it should be increased if our approach is used in scenarios with larger vocabularies than those that we present in this paper. There is also a tight relation between the dimensionality of the vectors and the performance since the higher dimensionality, the more operations need to be computed during training and prediction.

One of the hyperparameters to prevent overfitting is the **dropout**. Its mission is to select which weights are updated and which are not in each iteration. If all the weights were adjusted in each iteration, overfitting would be more likely to happen. We have set this parameter at 0.75, which means that each neuron is updated with a probability of 0.25 or ignored with a probability of 0.75.

Finally, the **learning rate** is a hyperparameter in the range $[0, 1]$ that controls how much the weights are adjusted, i.e., how much the neural networks learn from each input-output pair. Empirically, we have chosen a learning rate of 0.005.

3.3 Data preprocessing and postprocessing

One key issue of training neural networks is the quality and quantity of data. To meet this requirement, we preprocess the raw data before feeding it to our networks to *(i)* fulfil their input requirements, and *(ii)* improve the performance of the system.

First of all, we take advantage of the more restricted syntax of models and code and represent these artifacts as abstract-syntax-trees (AST) before feeding them to our networks. To do this, we create a tree representation of the models and/or code in JSON format.

For each object in the model, we create a branch hanging from the root node. Each object has two children: one that is always present and captures its identifier and type, and one to keep track of its features (e.g., attributes) and its values. When the values are of primitive types, we store the actual value. On the

[5] Different values can be provided as parameters when executing our python implementation

contrary, when values are not an instance of a primitive type—i.e., they are references to other objects—we store the identifier of the referenced object). We consider associations as first class citizens, too, hence each link present in the model is represented in the tree as a branch hanging from the root node. These branches contain information about: the source object, the target object and the role name. Bidirectional associations are represented as two branches hanging from the root node, each one representing one association role.
A concrete example illustrating how this representation looks in practice is presented in Section 4.

Our dataset is composed of three files, one containing the training data, other for the validation data and other for test data. Each of these files contains an array of JSON objects. Each object contains the input and output model/code of our architecture. Listing 1 shows the structure that a JSON file must follow. The keywords that our architecture uses to parse the JSON files and build the tree for its internal representation are *source_ast*, *target_ast*, *root*, and *children*. We present in Section 4 the structure of these files in more detail.

As part of this work, we have built a simple Java program that receives UML models as input, parses them using the EMF API and and generates their corresponding tree representation. Seamlessly, other formats can be supported as long as a driver is created for them.

Listing 1 JSON file structure.

```
[
 {"source_ast": { ... },
  "target_ast": { ... }
 },
 ...
 {"source_ast": { ... },
  "target_ast": { ... }
 }
]
```

During training, neural networks build a dictionary with each token in the training dataset. Any word not present during training will be interpreted as the token *UNK* when transforming models. The networks will not be able to understand its meaning and thus they might fail performing the transformation. Unlike natural languages in which the vocabulary is closed, in our case there is an infinite dictionary because of all the names a developer can give to the entities. This is known as the unlimited vocabulary problem. To solve this problem, and avoid the presence of the *UNK* token, during the preprocessing phase, we rename all the tokens that are not keywords to a closed set of words (for instance, classes' names are A, B, C, ..., attribute names are x, y, z, ..., etc.) and keep track of this renaming to "undo" it in a postprocessing phase.

Another advantage of this renaming phase is the reduction of the scarcity of data, which improves the accuracy of the output artifact. For the case studies presented in Section 4, we measured the quality of the result with and without the renaming and observed that for the same datasets, the renaming of non-keywords raised the accuracy between 0.25 and 0.4.

In the next section, we show the results given by our architecture applied on two typical model transformations.

4 Case Studies

In this section, we show the generality and usefulness of our architecture and study its feasibility by applying it on two case studies covering two of the main operations that are performed on models: model-to-model transformations and code generation[6].

For each case study, we show how the generic architecture has been successfully instantiated and its evaluation in terms of the quality and performance of the results in each case study.

The experiments that we present in this section use the PyTorch library[7] and have been executed on a machine with Ubuntu 16.04, an Intel Xeon E5 (2.50GHz) processor, 32Gb of RAM memory, and no support for Nvidia CUDA3. Although using the GPU that an Nvidia Graphic Card provides with CUDA usually speeds up the training process and improves its scalability, we felt that it was more realistic to assume that our target user would not have such specific equipment even if it is becoming more and more common. We plan to study the performance improvements in the near future.

4.1 Model-to-Model Transformations

The model-to-model transformation case study is the well-known model transformation example Class-to-Relational [24]. This transformation takes as input a model conform to the Class metamodel and transform it into a Relational model. Figure 4 presents the input and output metamodels.

Translation rules. Some examples of translation rules[8] that the neural networks have to learn are:

[6] The code and data required to reproduce our experiments are available at: `https://github.com/modelia/ai-for-model-manipulation`

[7] `https://pytorch.org/`

[8] The ATL implementation of the transformation contains 1 helper and 6 rules – 94 LoC in total (without counting the ATL file headers). For full transformation details, please, check [24].

– Each Class is transformed into a Table;
– Each DataType is transformed into a Type;
– Each single-valued Attribute of type DataType is transformed into a Column;
– Each multi-valued Attribute instance of the type DataType is transformed into a Table.

Listing 2 Input model of Figure 5 as JSON tree.

```
{
"source_ast":
{ "root": "<MODEL>",
  "children": [
   { "root": "<OBJ>",
     "children": [
    { "root": "D",
          "children": [
      { "root": "Datatype",
      "children": [ ] },
      { "root": "<ATTR>",
    "children": [
        { "root": "name",
      "children": [
          { "root": "Integer",
        "children": []}]}]}]}]},
   { "root": "<OBJ>",
     "children": [
     { "root": "A",
       "children": [
       { "root": "Class",
         "children": [ ]},
       { "root": "<ATTR>",
         "children": [
         { "root": "name",
           "children": [
           { "root": "x",
             "children": []}]}]}]}]},
   { "root": "<OBJ>",
     "children": [
     { "root": "B",
       "children": [
       { "root": "Attribute",
         "children": [ ]},
       { "root": "<ATTR>",
         "children": [
         { "root": "name",
           "children": [
           { "root": "y",
             "children": []}]},
         { "root": "multivalued",
           "children": [
           { "root": "false",
             "children": []}]}]}]}]},
   { "root": "<ASSOC>",
     "children": [
     { "root": "att",
       "children": []},
     { "root": "A",
       "children": []},
     { "root": "B",
       "children": []}]},
   { "root": "<ASSOC>",
     "children": [
     { "root": "type",
       "children": []},
     { "root": "B",
       "children": []},
     { "root": "D",
       "children": []}]}]},
 "target_ast":
 { ... }
}
```

Training dataset. For this example we created a synthetic dataset in which each example contained an input model with up to 30 model elements (classes, attributes, datatypes) and its corresponding output model. Figure 5 shows one of the examples that are part of our dataset.

The input and output of our architecture, as stated in Section 3.3, are trees in JSON format. For readability reasons, Figure 6 presents with a graphical notation the tree derived from the input model in Figure 5, and Listing 2 its JSON representation.

We have used this synthetic case study to analyse how our neural networks perform in terms of correctness (accuracy and loss) and performance, and which are the factors (number of models, size of models, etc.) that impact these two properties and how they are related.

4.1.1 Correctness of the results

As explained in Section 2.1, the correctness of ANNs is studied through its accuracy and overfitting (the latter being measured through the validation loss). The accuracy should be as close as 1 as possible while the validation loss as close to 0 as possible.

The accuracy is calculated comparing for each input model in the test dataset whether the output model transformed by the networks corresponds with the expected expected output model. The formula is:

$$\text{accuracy} = \frac{\#\text{ of correctly transformed models}}{\#\text{ of input-output models in the test dataset}}$$

In Fig. 7, we show how the accuracy grows and the loss decreases with the size of the dataset, i.e., the more input-output pairs we provide for training, the better our software learns and predicts (transforms). In this concrete case, with a dataset of 1,000 models, we reached an accuracy of 1 and zero loss (meaning that no overfitting was taking place), which means that the ANNs are perfectly trained and ready to use. Note that we show the size of the complete dataset but, we have split it using 64% of the pairs for training, 16% for validation, and 20% for testing. It is worth emphasizing that these results (as little as 750 models to reach a perfect training) are specific for this concrete example

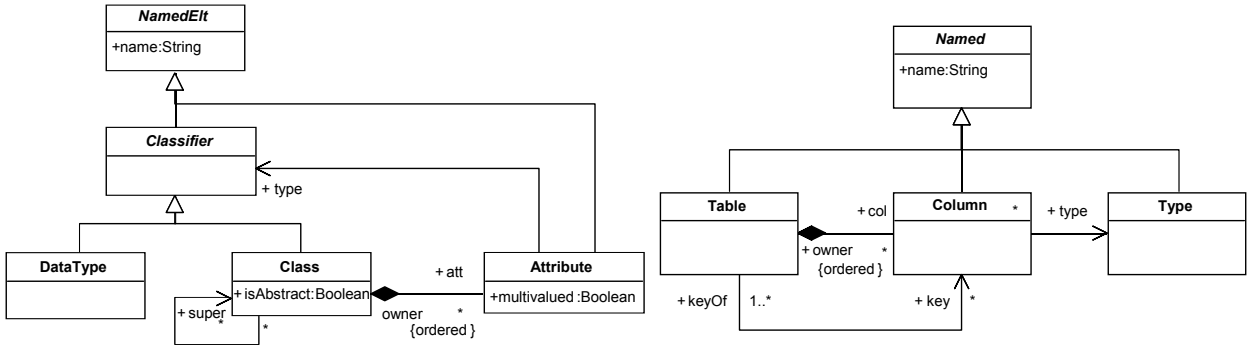


Fig. 4 Class (left) and Relational (right) metamodels from [24].

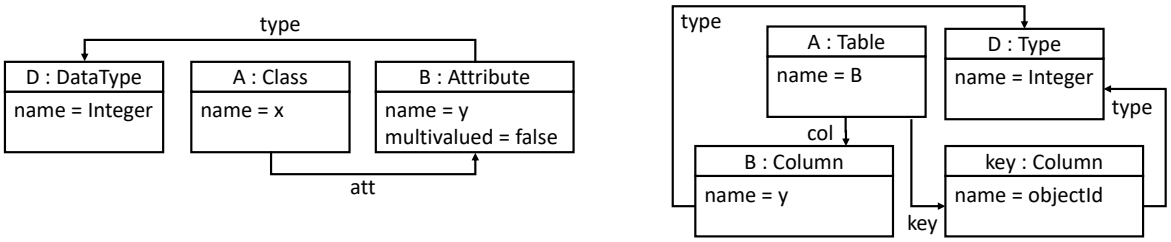


Fig. 5 Input-output example for the Class2Relational case study.

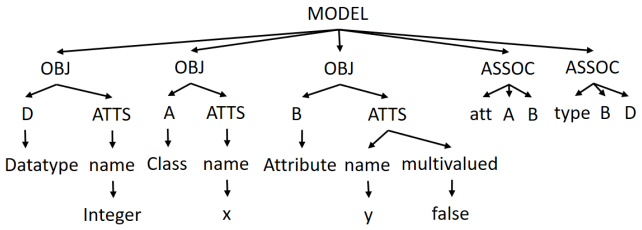


Fig. 6 Input tree corresponding to the input model of Figure 5.

since the number of models needed depends on different factors (which cannot be predicted even with heuristics but discovered empirically) as we will show next[9].

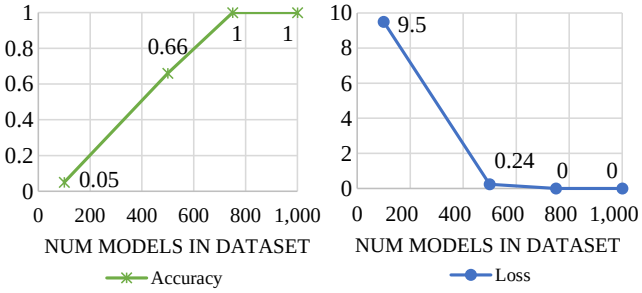


Fig. 7 Variation of accuracy and loss during training.

[9] For the sake of comparison, in this particular case, we are inferring the same model-to-model transformation that we presented in [13]. This time the transformation is learned with a dataset of only 750 models instead of 1,000.

4.1.2 Performance

There are two performance dimensions we need to consider: how long it takes for the training phase to complete and, once the network has been trained, how long it takes to transform an input model with it. Note that the training needs to be performed only once per each transformation scenario.

Training performance: The two main factors that impact the training time are the size of the training dataset (i.e., the number of models that compose the dataset) and the average size of models in it.

Figure 8 shows the performance of the training phase for the Class2Relational example depending on (1) the size of the dataset and (2) the size of the models that compose it. To avoid as much noise as possible due to the influence of the transformation or different types of models, we have reduced the transformation that we want the neural network architecture to learn to only one rule: DataType2Type. To do this, we have created datasets whose input-output pairs only contain DataTypes and their corresponding Types.

To study the first case (1), each input-output pair only has one DataType and Type, respectively. Note that due to the simplicity of the transformation and the fact that all input models are basically the same, we are able to reach maximum accuracy by using a dataset with a very few models (less than 10), nevertheless we built datasets of increasing sizes to test the performance of our architecture when the complexity of

other transformations require larger datasets. The top of Fig. 8 shows how the training time grows linearly when increasing the size of the training dataset.

To study the second case (2), we have kept constant the number of input-output examples to 100 pairs of models and have varied the number of elements in each model, we have added models with a different number of DataTypes and Types. On the right-hand side we study the impact of growing the average size of the models (ranging from 1 to 30 DataType-Type pairs). As shown at the bottom of Fig. 8, there is a quadratic growth when increasing the size of the models.

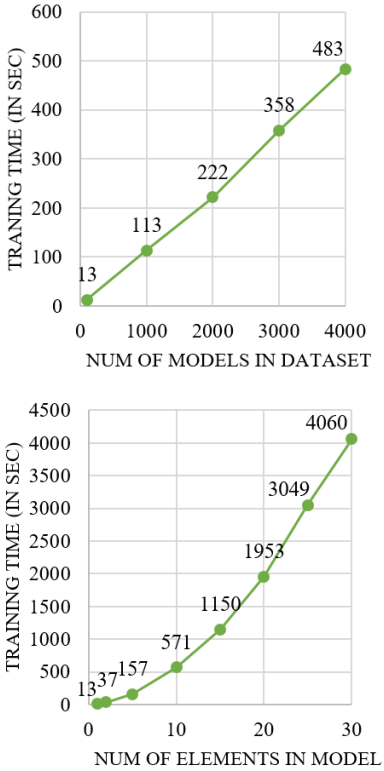


Fig. 8 Impact of the size of the training dataset (top) and of the size of the models (bottom) when training.

Transformation performance: After the training, we have evaluated the transformation time of the network on a set of input models. Figure 9 shows that the transformation time grows linearly with the number of model elements.

As a reference, we have also compared the execution time of our ML-based transformation with the execution time of the ATL version of the same transformation for our the synthetic models and the model provided in [24], which contains 2 objects of type *Class* with 2 and 3 object of type *Attribute* each and 2 objects of type *Datatype*. Though ours was a little bit slower for the models we tested (for instance for the model in [24] ATL takes 0.033 seconds while our approach takes 0.722 seconds), time is within the same order of magnitude: less than a second for models up to 30 elements to be transformed. Therefore, we do not see this as a negative aspect for ML-based transformations. Although a bit slower, the time in all cases is quite reasonable and the advantages of our approach may pay off.

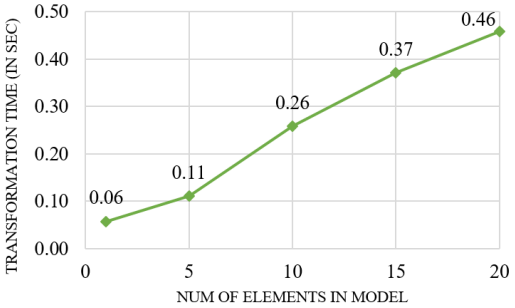


Fig. 9 Impact of the size of the models when transforming.

4.2 Code Generation

The goal of this case study is to train our networks to take as input a UML class model and generate its corresponding Java code (focusing on the structural part). The generated code should follow the coding standards imposed by the project, which we present below.

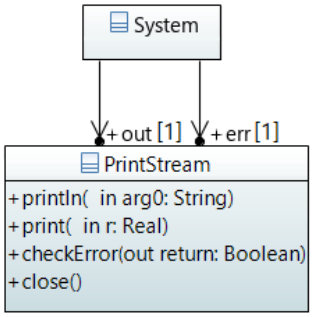
Translation rules. Due to the abstraction gap between UML and Java, there are many variability points in the translation.

Some translation rules can be considered as the de facto standard because of share object-oriented programming concepts that exist both in the UML model and the Java code. Examples of such translation rules are:

- UML classes are transformed into Java classes;
- UML attributes into Java attributes and;
- UML associations into Java attributes referencing the class in the other end of the association.

Other translation rules vary depending on the project or company's style, as for example:

- Primitive data type conversions, e.g., a UML attribute whose type is *Real* could be mapped to a Java attribute with type *double*, *Double* or *float*;
- Visibility management. Public UML attributes could be transformed into Java public attributes or as a private attributes plus the corresponding getter and setter public methods;
- UML internal model structure in packages can be replicated in Java optionally.




```
public class System {

  private PrintStream out;
  private PrintStream err;

  public PrintStream getOut() {
      return out;
  }
  public void setOut(PrintStream out){
      this.out = out;
  }
  public PrintStream getErr() {
      return err;
  }
  public void setErr(PrintStream err){
      this.err = err;
  }
}
```

```
public class PrintStream {

  public void println(String arg0){

  }
  public void print(double d){

  }
  public boolean checkError(){

  }
  public void close() {

  }
}
```

Fig. 10 UML model - Java code example.

This means that there is no universal UML-to-Java translation but it varies depending on the company or project. Our network architecture needs to learn the translation of both standard and project/company-specific aspects as part of its training in order to obtain a generator that reflects the specific company choices. Note that we would face a similar discussion when dealing with the opposite scenario, a Java-to-UML transformation.

In our particular case, given our training data (which we present below), our neural network architecture learns to translate each UML class into a Java class which contains the same attributes and the same operations, and that associations are mapped to Java attributes. It also learns that all these elements must have the same name. It learns that the UML visibilities *private (-)*, *public (+)*, and *protected (#)* are mapped to the Java visibility keywords *private*, *public*, and *protected*; and that the UML visibility *package (∼)* has no keyword associated in Java. It also learns that the primitive types *Real*, *Integer*, *String*, *Boolean* and *UnlimitedNatural* are transformed into the Java keywords *double*, *int*, *String*, *boolean* and *int*, respectively. Finally, our architecture learns that for each private attribute or association, it has to generate its getter and setter. Figure 10 shows an example of an input-output pair that our neural network architecture has learned.

Training dataset. To the best of our knowledge, there are no public repositories with enough model-code examples from a single company or project to train our neural network architecture. Therefore, we created our own training dataset. We have downloaded the source code of the Eclipse IDE, which is written in Java. To obtain the corresponding UML model, we have reverse-engineered it using MoDisco [25], obtaining a Java model of the code. We transformed this Java model to a pure high-level UML model by removing all the "low-level" details such as method implementations.

From these artifacts, we derived the training dataset (training, validation and test files), which is the set of pairs *(UML-class, Java-class)*, preprocessed as stated in section 3.3. For this case study, we have to transform into JSON trees both models and code. In the previous subsection, we presented how class diagrams are transformed into trees and how these trees are mapped to JSON files to be readable by our architecture. For the code part, which is the value of *target_ast*, we have taken the code, built its abstract-syntax tree (AST) and then transformed those trees to JSON format.

The dataset (*D1*) contains 25,375 pairs.

We discarded[10] some pairs because due to the size of their classes they fell into one of these known issues:

1. They did not fit into RAM memory. Pytorch, which is the ML library used in this work, displayed the message "not enough memory: Buy new RAM!" when stopping the execution due to an excessive size of some of the input-output examples that it was unable to process. This is a current constraint imposed by the technology (Pytorch or any other library) dealing with neural networks.

[10] Another solution could have been to slice those classes in chunks

Dataset	Size	Training time	Acc	Loss
D2	1,000	21' 30" (0.36h)	0.43	9.85
	5,000	54' 30" (0.91h)	0.56	3.83
	10,000	2h 57' 15" (2.95h)	0.67	2.62
	15,000	4h 28' (4.47h)	0.69	2.44
	20,480	5h 28' (5.47h)	0.69	2.44
D3	1,000	23' 30" (0.39h)	0.71	5.08
	3,000	1h 1' 28" (1.02h)	0.88	1.98
	5,000	1h 24' (1.40h)	0.92	1.08
	7,000	2h 23' 41" (2.39h)	0.94	1.37
	8,937	3h 8' (3.13h)	0.94	1.26

Table 1 Training results.

2. Their size led to the problem of long-term dependencies [26], which imposes the constraint that each input-output cannot be of an arbitrary size because RNNs cannot deal with examples with hundreds of elements yet.

The curated dataset (*D2*) contains 20,840 examples. A second issue we faced when training the network with this dataset was that:

3. There were inconsistencies in the training data coming from the original Java code. For instance, when there is inheritance between classes, the getters and setters of an attribute are placed arbitrarily in the class in which the attribute is defined or in any subclass. When there is no exact rule saying where getters/setters should be placed, neural networks are confused and might fail. Typically, if for the same input, neural networks receive different outputs (which is usually the case when writing code [27]), they follow the "rule" which they have seen more often.

In order to measure the impact of this "human factor" in the accuracy of the translations, we created a curated version of *D2* for comparison purposes. Instead of accounting for all possible inconsistencies, we decided to remove all inherited classes as a means to remove many of them in one single step. As a result, we obtained the dataset *D3*, which contains 8,937 examples.

4.2.1 Correctness of the results

For each experiment, we took the complete dataset and split it into three: 64% for training, 16% for validation and 20% for testing, which is a popular ratio that, empirically, has worked well in our case.

The results on accuracy and loss, after training our architecture with the dataset D2 and D3 of increasing size, are shown in Table 1.

The top chart of Fig. 11 shows the results for D2 (and several subsets of different sizes). Despite feeding the network with all the data available, the accuracy

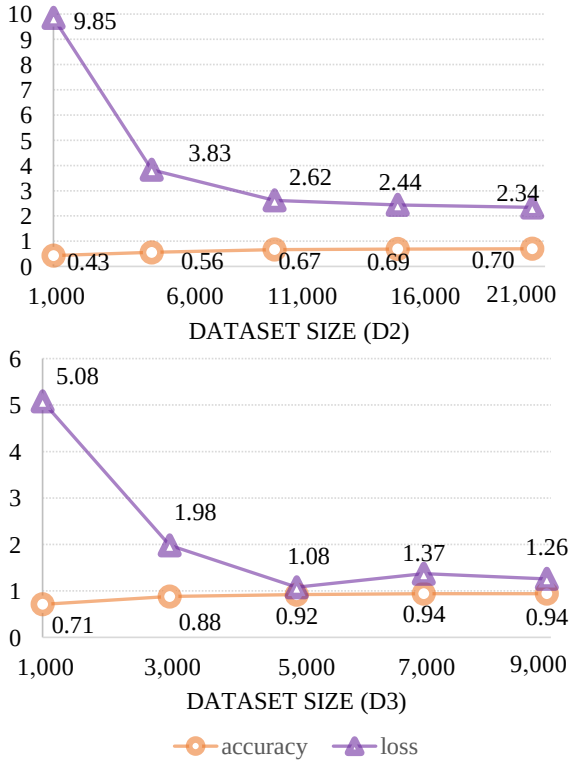


Fig. 11 Training phase – Correctness.

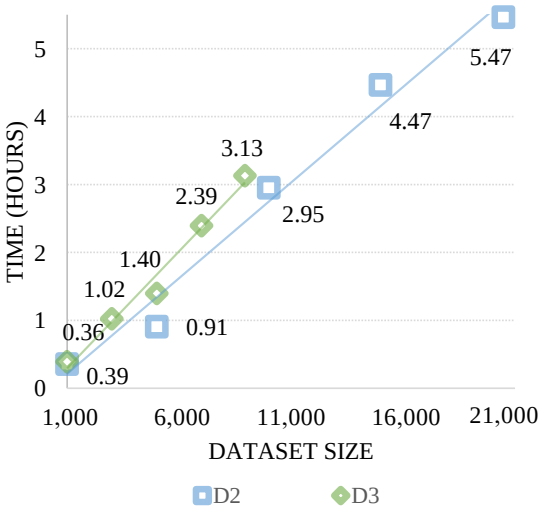


Fig. 12 Training phase – Performance.

stabilizes around 0.7. In comparison with the results for D3 (shown in the bottom chart of Fig. 11), we observe how, even with less data, the accuracy increases up to 0.94. This highlights the importance of the quality (i.e., consistency in this case) of the training data.

The accuracy has not reached 1 due to other small inconsistencies and the presence in the test dataset of situations not covered during training. A simple example of the former issue occurs when the pattern that

most input-output pairs follow is that attributes of type *Real* are transformed into attributes of type *float*. Nevertheless, there are a few cases in which *Real* attributes are transformed into *double* attributes. Since our neural networks learn the general pattern during the training phase, they transform *Real* into *float*. Then, during the testing phase, when it happens that an input-output example contains in its output a *double*, the expected output (i.e., *double*) and the generated output (i.e., *float*) do not coincide and our differencing algorithm marks it as mistaken, resulting in a decrease in the accuracy. An example of the latter case occurs when the training dataset does not contain any example with *enumerations*. When during the testing phase our architecture finds an input that contains an enumeration, it fails because it is a case for which it was not trained and did not learn how to handle.

Note that, as explained in Section 2.1, the accuracy is calculated as:

$$\text{accuracy} = \frac{\#\text{ of correct outputs}}{\#\text{ pairs in the test dataset}}$$

and that we use a full tree differencing algorithm to compare the output produced by the networks (i.e., the generated code) and the expected output (i.e., expected code) for each pair input-output pair in the test dataset. This means that, if a single token fails, we mark the code generation for that pair as a failure. This can be regarded as a worst-case scenario as we are sometimes discarding generated code that it is semantically equivalent to the expected output even if it presents slight syntactic differences. We could relax this comparison by evaluating our results with metrics such as *word error rates* (WER), BLEU [28] and NIST [29].

4.2.2 Performance

Figure 12 shows the training time for our two datasets (and subsets). It shows how the training time grows linearly with the size of the dataset.

Once the networks are trained, the code generation for a model takes an average time of 18 milliseconds with a standard deviation of 3 milliseconds. This shows that the networks are efficient enough to be part of any continuous software development process.

4.3 Threats to Validity

In this subsection, we elaborate on several factors that may hinder the validity of our results. We follow the guidelines proposed in [30].

Internal validity – are there factors which might affect the results in the context the case studies?. Concerning the measurement approach we used in our case study, ongoing threads within the OS could affect our performance measurements. To address this issue, we stopped all possible tasks, including those that are automatically started. Furthermore, part of the training data we have used is synthetic due to the shortage of public data—either artificially created or derived from real data (i.e., reversed-engineered models). Nevertheless, we created it in such a way that it faithfully represent reality and does not benefit our approach in any single way. We have also discussed how the characteristics of the training data (quality, quantity and coverage) impact the results.

External validity – to what extent is it possible to generalize the findings for other type of model manipulation operations? So far, we cannot claim any correctness and/or performance results outside the context of the presented case study. Nevertheless, the evaluation method used in the case study can indeed be applied on other model manipulation tasks. Since each one of these manipulations are reduced to pairs of input-output trees the results will depend on the quality of the training data (amount of data, coverage, size of the input-output pairs, etc.) a lot more than on the kind of manipulation operation itself.

5 Current Limitations and Mitigation Plans

As shown in the previous section, our architecture is a prominent new way to address model transformations. The instantiation of the architecture only requires to provide the input and output models/code encoded as trees in a textual format. For code, this is straight forward using the AST of the program. For models, there is not one single way to do it, but all the possible generated trees would be equally valid, i.e., our architecture does not impose any restriction on the structure or content of the trees.

There are some technical limitations related to the use of neural networks that need to be considered before choosing our architecture over traditional approaches. These are:

Size and quality of the training dataset. ANNs require a considerably amount of data for training. Unfortunately, there is no rule that states how much data is needed. It fully depends on the concrete mapping to learn, as complex mappings will require more data than simple mappings.

If the models available are not enough[11] we face the risk of getting poor results.

Strategies to mitigate this issue would include data augmentation techniques (e.g., reusing model mutation procedures or employing Generative Adversarial Networks (GAN) [31] to generate further examples for a core set) or apply transfer learning[12] to avoid starting the learning process from scratch.

The quality of the training is also tied to the diversity of the dataset. An ANN can only predict scenarios that follow a pattern that it has seen before. Coverage metrics for the input-output metamodels [33] and the use of graph kernel techniques [34] could give feedback to the user regarding the need for adding more samples to cover for corner cases.

Operations with values and evaluation of expressions. ANNs are still a field under heavy development. Although the state of the art presents improvement day after day, they are some aspects for which no solution is available yet. For which it affects our approach, ANNs are unable to perform operations with values and evaluation of expressions. For now, our neural network architecture deals with the mapping from elements between the input and output domains, but it is not able to deal with the evaluation of expressions. For instance, our architecture is not be able to compute mathematical operations, operations with strings, etc.

Fortunately, given the importance of such challenge in many domains, this is an active research topic and we are confident that, at the current innovation pace, neural networks obstacles will be surpassed in a not-so-distant future.

Large inputs/outputs. We have reported that neural networks are not able to deal with large inputs and outputs. This problem can be addressed by slicing models and code in order to avoid the problem of running out of memory and having large models for which neural network-based approaches performs poorly. This slicing, manipulating and putting the pieces back together may raise other problems due to the dependencies of the different chunks. Nevertheless, this is not a new problem and there are available solutions to deal with it such as [35].

Apart from these technical limitations, there is another important limitation which is the social acceptance. Social factors may also hamper the adoption of a "gray-box" ML-based approach[13]. Users may be reluctant to trust a piece of software that they are not able to understand. As done in other AI applications, adding explanation capabilities to the system will be a must.

6 Related Work

The typical solution to tackle model manipulations is to write a transformation program using a specific transformation language [36]. In the model transformation field, we have plenty of well-known examples of model transformation languages such as ATL [1], QVT [37] and ETL [38] that could be use to write such transformations. Still, the adoption of these languages in the industry is limited. Model transformation languages are not very intuitive to non-expert users and their IDEs usually lack the advanced facilities (e.g., nice debugging tools) required to develop transformations.

Model Transformation By-Example (MTBE) is an attempt to simplify the writing of exogenous model transformations [39–43]. In an MTBE approach, users have to provide source models, their corresponding target models as well as the correspondence between them—for which a correspondence language has to be used. From this, the MTBE approach generates partial mappings that form the basis of the transformation. Although these approaches free users from learning a full transformation language, they still have to learn a correspondence language and manually build/complete the generated transformation mappings.

Kessentini et al. [44,45] use search-based techniques to generate target models even if there is a limited number of examples available. The basic idea is to find among the examples the ones that are probably the closest match to the source model the user is trying to transform. Nevertheless, similar to the previous MTBE approaches they require the existence of transformation traces for the available examples so that they can generate the optimal solution.

Lano et al. [46] propose a method to infer model transformation mappings from natural language model transformation requirements. The method starts by applying natural language processing (NLP) [47] to obtain clauses that are either mappings or constraints. Missing mappings are inferred by using metamodel matching like Data Structure Similarity [48]. While this approach requires less available data (mappings are further enhanced by validating them with MTBEs and correcting them with inductive logic programming [49],

[11] The challenge of collecting and curating model repositories is well-known in our community

[12] Transfer learning is the improvement of learning in a new task through the transfer of knowledge from a related task that has already been learned [32].

[13] Unlike black-box approaches, our contribution is grey-box since details such as its architecture, (hyper-)parameter values, training method and training data are available.

and thus, optimal results still need some examples to rely on) than ours, it does depend on the availability of explicit transformation requirements. Moreover, in our work, by inferring model transformations with neural networks instead, transformations learn the company/project style in the process, which is not typically collected in natural language requirements.

Baki et al. [50] are able to discover more complex transformations by splitting the transformations traces in pools and applying genetic algorithms. Again, transformation traces are needed for the discovery phase.

In contrast to previous works, our approach does not require any kind of correspondence or tracing information to be provided by the user or domain expert and learns purely from the couples of input/output models. This enables non-expert users to employ our approach.

Model transformation is only an instantiation of the more general problem of data transformation, which shows up across many fields of computer science (e.g., databases, XML, modeling, and big data). This topic has been largely addressed by the (relational) database community, especially dealing with the heterogeneity of the data sources and the impedance mismatch problems [51–53]. Nevertheless, to the best of our knowledge, machine learning based approaches have not been attempted in said community to address the problem of schema manipulation. We believe our results can be replicated in this context as well.

Feature location has some overlapping with model transformations. Feature location has been thoroughly explored for source code [54]. For artifacts resembling models, authors in various studies [55–57] propose feature location algorithms based on retrieval techniques (e.g., latent semantic indexing in NLP), rather than machine learning. In the modeling community, Ho-Quang et al. [58] encode models to classify UML class diagrams, and Marcén et al. [59] propose an ontology-based model fragment encoding to enable requirement traceability link retrieval. Although our problem is to infer any generic model transformation, we share with the model information retrieval problem the need to encode models in a computer-exploitable format. For such a task, encodings used in information retrieval are domain-specific (e.g., based on an existing ontology). We use instead model embedding that are specific to the particular input-output dataset, not the domain knowledge.

So far we have focused on works in the that apply machine learning to model transformations. There are other works that apply it to other model-driven engineering problems. As we present in the following, such works are complementary to ours. For instance, Barriga et al. [60] uses reinforcement learning [61] for model repair. In our work, we have used neural networks—which a form of supervised learning—while reinforcement learning is a different branch of machine learning in which an agent interacts with an environment and learns by receiving rewards (i.e., trial and error). Although a promising field, it remains unclear whether reinforcement learning would be appropriate to address our problems. Nguyen et al. [62] propose an approach to automate the classification of metamodel repositories with machine learning. The architecture is a feed forward neural network [63]. Besides the obvious difference in the task performed (classification), and the architecture (feed-forward neural network), the models are encoded as features vectors instead of dedicated embeddings for model transformation.

Although in the broader software engineering community machine learning techniques have also been applied to infer knowledge as models from unstructured project data (e.g., [64–67]), these tasks are out of the scope of our work since our approach deals only with systematic model manipulations whose input and outputs are (semi-)structured data that can be represented as trees.

The programming research community has been much more active in the area of mixing machine learning and (code) transformation. In [68], Chen et al. use LL machines, neural programs and a two-phase reinforcement learning-based search technique to infer programming language parsers from input-output pairs. In [69], the authors propose an algorithm that learns the style of a codebase, and suggests revisions to improve stylistic consistency. The Aicodoo tool [70] uses machine learning to automatically write code.

The main difference between these approaches and our work resides in the fact that they focus on code, while we operate on models, which demanded several adaptations. Although we were inspired by [68] in several points such as the encoding of the inputs and outputs of the neural networks and the need of pre- and post-processing steps to better fit model transformation problems, not only the application domain differs but also the pre- and post-processing steps are different as well as the architecture used to solve the problem of inferring mappings.

7 Conclusions and perspectives

We have explored a novel approach for inferring heterogeneous model transformations based on neural networks. Our results show the potential of this approach. The approach has been validated in a classic model-to-model transformation scenario and on realistic code-generation entreprise-scale scenario.

Throughout our experiments, we have identified important limitations related to both neural network technologies and methodology aspects as reported in Section 5. Given the current pace of innovation in the ML community, we are confident that some of them will be unblocked soon. Nevertheless, we hope that companies, open source communities and other interested parties appreciate the value in this approach and decide to take the necessary steps to integrate it in their development processes, especially when facing a number of repetitive projects.

Beyond the directions pointed out in the previous section, we also plan to extend our approach in other several directions to make it more enticing.

First of all, we would like to take advantage of the functioning of the *softmax* layer and the fact that its output can be seen as probabilities associated to each node of the output tree to provide a confidence value associated to each model/code that our architecture generates as well as to each element composing it.

In new projects with evolving practices or where the styling guidelines may be slightly different from existing projects, our neural network architecture has to be retrained from scratch. We plan to study how transfer learning (i.e., the reuse or refinement of pre-trained software artifacts that were trained for a specific task as starting point for a second task) can decrease the amount of data needed to train networks and improve the performance of our architecture.

We will extend the network capabilities to cover the more complex scenario of the generation of basic behavioral code, too. The general architecture covers already this case (behavioural code can also be expressed as trees) but we may need to explore the best hyperparameters for this scenario.

The use of Transformers [71] has been proven beneficial for tasks involving natural language processing. Despite its admirers [72] and detractors [73], we would like to study whether an architecture based on Transformers could impact our results.

Additionally, we would like to continue the validation of our approach by exploring how it fares when faced with transformation chains e.g. round-trip synchronization model-code-model [74].

Finally, at the tool level, we plan to provide connectors for models other than EMF, and pretrained networks for a variety of simple cases to help companies kickstart the adoption of our architecture.

Acknowledgements This work is supported by Spanish Research project and TIN2016-75944-R and CEA in the context of the Modelia initiative.

References

1. F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: A model transformation tool, Science of Computer Programming 72 (1-2) (2008) 31–39.
2. Eclipse, Acceleo (2006).
URL `http://www.eclipse.org/acceleo/`
3. Eclipse, Xtext (2009).
URL `https://www.eclipse.org/Xtext/`
4. L. Burgueño, J. Cabot, S. Gérard, The future of model transformation languages: An open community discussion, Journal of Object Technology 18 (3) (2019) 7:1–11. `doi:10.5381/jot.2019.18.3.a7`.
5. A. Forward, T. Lethbridge, Problems and opportunities for model-centric versus code-centric software development: A survey of software professionals, Proc. of the International Workshop on Models in Software Engineering (2008) 27–32.
6. B. Selic, What will it take? A view on adoption of model-based methods in practice, Software and Systems Modeling 11 (4) (2012) 513–526. `doi:10.1007/s10270-012-0261-0`.
URL `https://doi.org/10.1007/s10270-012-0261-0`
7. Y. Wang, B. Zheng, H. Huang, Complying with coding standards or retaining programming style: A quality outlook at source code level, Journal of Software Engineering and Applications 1 (1) (2008) 88–91.
8. Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Łukasz Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, J. Dean, Google's neural machine translation system: Bridging the gap between human and machine translation (2016). `arXiv:1609.08144`.
9. D. Bahdanau, K. Cho, Y. Bengio, Neural machine translation by jointly learning to align and translate (2016). `arXiv:1409.0473`.
10. S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (8) (1997) 1735–1780.
11. I. Sutskever, O. Vinyals, Q. Le, Sequence to sequence learning with neural networks, Advances in Neural Information Processing Systems 4 (2014) 3104–3112.
12. X. Chen, C. Liu, D. Song, Tree-to-tree neural networks for program translation, in: Proc. of the Annual Conference on Advances in Neural Information Processing Systems (NeurIPS'18), 2018, pp. 2552–2562.
13. L. Burgueño, J. Cabot, S. Gérard, An LSTM-based neural network architecture for model transformations, in: Proc. of the 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS'19), 2019, pp. 294–299.
14. M. Minsky, S. Papert, Perceptrons (1969).
15. F. Rosenblatt, The perceptron: A probabilistic model for information storage and organization in the brain, Psychological Review 65 (6) (1958) 386—-408. `doi:10.1037/h0042519`.
16. D. Rumelhart, G. Hinton, R. Williams, Learning internal representations by error propagation., Parallel Distributed Processing: Explorations in the microstrucrures of cognition 1 (1986).
17. S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Pearson, 2016.
18. J. Schmidhuber, Deep learning in neural networks: An overview, Neural Networks 61 (2015) 85 – 117.

19. K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio, Learning phrase representations using RNN encoder-decoder for statistical machine translation, in: Proc. of the Conference on Empirical Methods in Natural Language Processing (EMNLP'14), 2014, pp. 1724–1734.
20. K. S. Tai, R. Socher, C. D. Manning, Improved semantic representations from tree-structured long short-term memory networks, in: Proc. of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, 2015, pp. 1556–1566.
21. K. S. Tai, R. Socher, C. D. Manning, Improved semantic representations from tree-structured long short-term memory networks (2015). arXiv:1503.00075.
22. I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016.
23. L. N. Smith, A disciplined approach to neural network hyper-parameters: Part 1 - learning rate, batch size, momentum, and weight decay, CoRR abs/1803.09820 (2018). arXiv:1803.09820.
URL http://arxiv.org/abs/1803.09820
24. AtlanMod (Inria), Class to relational transformation example, https://www.eclipse.org/atl/atlTransformations/#Class2Relational.
25. H. Bruneliere, J. Cabot, F. Jouault, F. Madiot, MoDisco: A generic and extensible framework for model driven reverse engineering, in: Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE'10), 2010, p. 173–174.
URL https://www.eclipse.org/MoDisco/
26. P. Koehn, R. Knowles, Six challenges for neural machine translation, in: Proc. of the 1st Workshop on Neural Machine Translation (NMT @ ACL), 2017, pp. 28–39.
27. M. Dorin, S. Montenegro, Coding standards and human nature, International Journal of Performability Engineering 14 (2018) 1308–1313.
28. K. Papineni, S. Roukos, T. Ward, W. Zhu, Bleu: a method for automatic evaluation of machine translation, in: Proc. of the 40th Annual Meeting of the Association for Computational Linguistics, 2002, pp. 311–318.
29. G. Doddington, Automatic evaluation of machine translation quality using n-gram co-occurrence statistics, in: Proc. of the 2nd International Conference on Human Language Technology Research (HLT'02), Morgan Kaufmann Publishers Inc., 2002, p. 138–145.
30. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, Experimentation in Software Engineering, Springer, 2012.
31. C. Bowles, L. Chen, R. Guerrero, P. Bentley, R. N. Gunn, A. Hammers, D. A. Dickie, M. del C. Valdés Hernández, J. M. Wardlaw, D. Rueckert, GAN augmentation: Augmenting training data using generative adversarial networks, CoRR abs/1810.10863 (2018). arXiv:1810.10863.
32. E. S. Olivas, J. D. M. Guerrero, M. M. Sober, J. R. M. Benedito, A. J. S. Lopez, Handbook Of Research On Machine Learning Applications and Trends: Algorithms, Methods and Techniques, Information Science Reference - Imprint of: IGI Publishing, Hershey, PA, 2009.
33. O. Semeráth, D. Varró, Iterative generation of diverse models for testing specifications of dsl tools, in: Proc. of the 21rd International Conference on Fundamental Approaches to Software Engineering (FASE'18)", year="2018", x-publisher="Springer International Publishing", x-address="Cham", pages="227–245", x-isbn="978-3-319-89363-1".
34. R. Clarisó, J. Cabot, Applying graph kernels to model-driven engineering problems, in: Proc. of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis (MASES@ASE'18), 2018, pp. 1–5.
35. L. Burgueño, M. Wimmer, A. Vallecillo, A Linda-based platform for the parallel execution of out-place model transformations, Information & Software Technology 79 (2016) 17–35.
36. L. Burgueño, F. Ciccozzi, M. Famelis, G. Kappel, L. Lambers, S. Mosser, R. F. Paige, A. Pierantonio, A. Rensink, R. Salay, G. Taentzer, A. Vallecillo, M. Wimmer, Contents for a model-based software engineering body of knowledge, Softw. Syst. Model. 18 (6) (2019) 3193–3205. doi:10.1007/s10270-019-00746-9.
37. OMG, MOF QVT Final Adopted Specification, Object Management Group, OMG doc. ptc/05-11-01 (2005).
38. D. S. Kolovos, R. F. Paige, F. A. C. Polack, The Epsilon Transformation Language, in: Proc. of the 11th International Conference on Model Transformations (ICMT'2008), Vol. 5063 of LNCS, Springer, 2008, pp. 46–60.
39. G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, M. Wimmer, Model transformation by-example: A survey of the first wave, in: Conceptual Modelling and Its Theoretical Foundations: Essays Dedicated to Bernhard Thalheim on the Occasion of His 60th Birthday, 2012, pp. 197–215.
40. D. Varró, Model transformation by example, in: Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MODELS'06), 2006, pp. 410–424.
41. M. Wimmer, M. Strommer, H. Kargl, G. Kramler, Towards model transformation generation by-example, in: Proc. of the 40th Hawaii International International Conference on Systems Science (HICSS'07), 2007, p. 285.
42. I. García-Magariño, J. J. Gómez-Sanz, R. Fuentes-Fernández, Model transformation by-example: An algorithm for generating many-to-many transformation rules in several model transformation languages, in: Proc. of the 2nd International Conference on Model Transformations (ICMT'09), 2009, pp. 52–66.
43. Z. Balogh, D. Varró, Model transformation by example using inductive logic programming, Software and System Modeling 8 (3) (2009) 347–364.
44. M. Kessentini, H. A. Sahraoui, M. Boukadoum, O. Benomar, Search-based model transformation by example, Software and System Modeling 11 (2) (2012) 209–226.
45. M. W. Mkaouer, M. Kessentini, Model transformation using multiobjective optimization, Advances in Computers 92 (2014) 161–202.
46. K. Lano, S. Fang, M. A. Umar, S. Yassipour-Tehrani, Enhancing model transformation synthesis using natural language processing, in: Proc. of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C'20), Virtual Event, Canada, 2020.
47. G. G. Chowdhury, Natural language processing, Annual review of information science and technology 37 (1) (2003) 51–89.
48. D. Buttler, A short survey of document structure similarity algorithms, in: Proc. of ICOMP 2004, Las Vegas, NV, USA, 2004.
49. N. Lavrac, S. Dzeroski, Inductive logic programming, in: Proc. of WLP 1994, Zurich, Switzerland, 1994.
50. I. Baki, H. A. Sahraoui, Multi-step learning and adaptive search for learning complex model transformations

from examples, ACM Trans. Softw. Eng. Methodol. 25 (3) (2016) 20:1–20:37.
51. P. A. Bernstein, S. Melnik, Model management 2.0: Manipulating richer mappings, in: Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD'07), 2007, pp. 1–12.
52. J. F. Terwilliger, P. A. Bernstein, A. Unnithan, Automated co-evolution of conceptual models, physical databases, and mappings, in: Proc. of the 29th International Conference on Conceptual Modeling (ER'10), 2010, pp. 146–159.
53. P. A. Bernstein, J. Madhavan, E. Rahm, Generic schema matching, ten years later, in: Proc. of 37th International Conference on Very Large Data Bases (VLDB'11), Vol. 4, 2011, pp. 695–701.
54. B. Dit, M. Revelle, M. Gethers, D. Poshyvanyk, Feature location in source code: a taxonomy and survey, Journal of software: Evolution and Process 25 (1) (2013) 53–95.
55. J. Font, L. Arcega, Ø. Haugen, C. Cetina, Feature location in model-based software product lines through a genetic algorithm, in: Proc. of the 2016 International Conference on Software Reuse, 2016, pp. 39–54.
56. S. Holthusen, D. Wille, C. Legat, S. Beddig, I. Schaefer, B. Vogel-Heuser, Family model mining for function block diagrams in automation software, in: Proc. of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools-Volume 2, 2014.
57. X. Zhang, Ø. Haugen, B. Moller-Pedersen, Model comparison to synthesize a model-driven software product line, in: Proc. of the 15th International Software Product Line Conference, 2011.
58. T. Ho-Quang, M. R. Chaudron, I. Samúelsson, J. Hjaltason, B. Karasneh, H. Osman, Automatic classification of uml class diagrams from images, in: Proceedings of the 21st Asia-Pacific Software Engineering Conference, IEEE, 2014.
59. A. C. Marcén, R. Lapeña, Ó. Pastor, C. Cetina, Traceability link recovery between requirements and models using an evolutionary algorithm guided by a learning to rank algorithm: Train control and management case, Journal of Systems and Software 163 (2020).
60. A. Barriga, A. Rutle, R. Heldal, Personalized and automatic model repairing using reinforcement learning, in: Proc. of the 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C'19), Munich, Germany, 2019.
61. R. S. Sutton, A. G. Barto, Reinforcement learning: An introduction, MIT press, 2018.
62. P. T. Nguyen, J. Di Rocco, D. Di Ruscio, A. Pierantonio, L. Iovino, Automated classification of metamodel repositories: A machine learning approach, in: Proc. of the 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS'19), Munich, Germany, 2019.
63. D. Svozil, V. Kvasnicka, J. Pospichal, Introduction to multi-layer feed-forward neural networks, Chemometrics and intelligent laboratory systems 39 (1) (1997) 43–62.
64. F. Gilson, C. Irwin, From user stories to use case scenarios towards a generative approach, in: Proc. of the 25th Australasian Software Engineering Conference (ASWEC'18), 2018, pp. 61–65.
65. M. Elallaoui, K. Nafil, R. Touahni, Automatic transformation of user stories into uml use case diagrams using nlp techniques, in: Proc. of the 9th International Conference on Ambient Systems, Networks and Technologies (ANT 2018) and the 8th International Conference on Sustainable Energy Information Technology (SEIT-2018) - Affiliated Workshops ANT/SEIT, 2018.
66. C. R. Narawita, K. Vidanage, Uml generator-an automated system for model driven development, in: Proc. of the 16th International Conference on Advances in ICT for Emerging Regions (ICTer'16), 2016.
67. M. Elallaoui, K. Nafil, R. Touahni, Automatic generation of uml sequence diagrams from user stories in scrum process, in: Proc. of the 10th International Conference on Intelligent Systems: Theories and Applications (SITA'15), 2015.
68. X. Chen, C. Liu, D. Song, Learning neural programs to parse programs, CoRR abs/1706.01284 (2017).
69. M. Allamanis, E. T. Barr, C. Bird, C. Sutton, Learning natural coding conventions, in: Proc. of the 22nd International Symposium on Foundations of Software Engineering (FSE'14), 2014, p. 281–293. doi:10.1145/2635868.2635883.
70. A. Derksen, aicoodoo, http://aicodoo.com/ (2020).
71. T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, A. M. Rush, Huggingface's transformers: State-of-the-art natural language processing (2019). arXiv:1910.03771.
72. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need (2017). arXiv:1706.03762.
73. S. Merity, Single headed attention rnn: Stop thinking with your head (2019). arXiv:1911.11423.
74. V. C. Pham, S. Li, A. Radermacher, S. Gerard, C. Mraidha, Fostering software architect and programmer collaboration, in: Proc. of the 21st International Conference on Engineering of Complex Computer Systems, (ICECCS'16), 2016, pp. 3–12.