Applying Product Line Engineering Concepts to Deep Neural Networks

Javad Ghofrani
Faculty of Informatics/Mathematics
HTW University of Applied Sciences
Dresden, Germany
javad.ghofrani@gmail.com

Ehsan Kozegar
Faculty of Engineering (Eastern Guilan)
University of Guilan
Guilan, Iran
kozegar@guilan.ac.ir

Anna Lena Fehlhaber
Leibniz University Hanover
Hanover, Germany
fehlhaberlena@gmail.com

Mohammad Divband Soorati
University of Lübeck
Lübeck, Germany
divband@iti.uni-luebeck.de

ABSTRACT

Deep Neural Networks (DNNs) are increasingly being used as a machine learning solution thanks to the complexity of their architecture and hyperparameters—weights. A drawback is the excessive demand for massive computational power during the training process. Not only as a whole but parts of neural networks can also be in charge of certain functionalities. We present a novel challenge in an intersection between machine learning and variability management communities to reuse modules of DNNs without further training. Let us assume that we are given a DNN for image processing that recognizes cats and dogs. By extracting a part of the network, without additional training a new DNN should be divisible with the functionality of recognizing only cats. Existing research in variability management can offer a foundation for a product line of DNNs composing the reusable functionalities. An ideal solution can be evaluated based on its speed, granularity of determined functionalities, and the support for adding variability to the network. The challenge is decomposed in three subchallenges: feature extraction, feature abstraction, and the implementation of a product line of DNNs.

CCS CONCEPTS

• **Software and its engineering → Software development methods**; **Software product lines**;

KEYWORDS

Software product lines, variability, deep neural networks, transfer learning, machine learning

1 INTRODUCTION

Software product lines (SPLs) [16] or software families [23] are software intensive systems which are developed using a managed set of reusable assets[1]. Software product line engineering methods enable software engineers to generate different software products while supporting the variety and commonalities in communication and behavior of software assets. Using customization in behavior and constraints on communications of software assets facilitate the organizations to develop their products faster with supporting a larger spectrum of requirements of their customers. Advances in reusing software assets have shown that SPLs optimize implementation while saving time and costs. These factors make the significant difference in the field of production and software industry. A software asset is a well-designed software system or subsystem that is engineered or refined to get easily integrated into the development of a SPL while providing a functionality within the software products. Software assets can have new behaviors and are able to communicate with other components. Feature-oriented software development [1] is one of the most efficient ways to realize SPLs in which the assets are described as features providing functionality to the system. Adding and removing features can affect the entire structure of the software product. Feature models are used to describe a SPL including the features and their relation to each other, i.e., constrains [4, 11].

Deep neural network (DNN) [18] is a buzz word in different fields of computer science which attracted the interest of many experts, e.g., image classification, activity recognition, object detection, automatic speech recognition, machine translation, etc. Neural networks cover a subset of the machine learning methods to perform classification and prediction tasks, based on the input

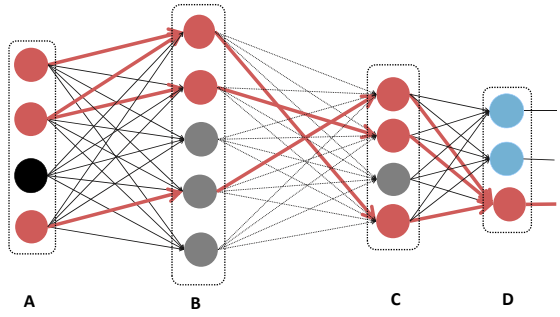

Figure 1: Example of DNN in which some parts are activated with a certain input value. Input (A), hidden (B&C), and output layers (D) are shown. The activated part of the neural network is highlighted in red.

[1] https://www.sei.cmu.edu/

values in a computational system. The advantage of neural networks compared to the algorithms are their relative tolerance to the noise in their input values. Furthermore, neural networks can be trained only with the data and do not require the development of an algorithm or mathematical solutions, which is the base of many functionalities in software systems. Recently, DNNs have grown dramatically thanks to cloud computing and analytic methods in big data. In DNNs, the process of finding any correlation between input and output will be performed in different levels of abstraction. Due to the necessity of finding structures and patterns in the world, the machine is forced to transform observations into data. Therefore, it is possible to identify different and more complex patterns. Various types of DNNs are developed based on their usage and input types. The commonly used type of DNNs are convolutional networks[2] [18] which are used in image processing tasks (e.g., NVIDIA's autonomous driving project[3]).

A DNN consists of an architecture (network layers) and its hyperparameters (weights of the edges). The basic unit of the architecture of a neural network is called "neuron". A set of neurons form a layer in the architecture of neural networks. A neural network consists of one or many layers with a matrix of neurons. "Edges" are the other basic units within the architecture of DNNs that connect the neurons in the neighboring layers. Figure 1 illustrates the structure of a neural network. DNNs receive the inputs from the first layer (input layer) and return the results via the last layer (output layer) of their architecture. A matrix of float values will be fed to the input layer to activate the network. The network maps the input values to a matrix in the output layer. In an ideal case, a node of output with maximum value reflects the decision of the network based on the input values. This mechanism will be used for prediction or classification tasks. Training a neural network initializes the neurons and sets the weights on edges. The weight values assigned to the edges are called hyperparameters. Using Keras library[4] one can define the architecture of a network in python and save or load the hyperparameters of neural network using a separate file with *h*5 extension.

Motivation: DNNs have shown that they can be applied to almost any domain if a set of training data is available. Instead of writing new programs, the data will be collected, and the neural network will be trained. Image processing, text summarization, text translation, and IoT (such as autonomous driving and smart homes) are making DNNs an alternative or even competitor to the software systems.

There is a trend for using DNNs in various fields such as natural language processing, image understanding, even android apps (See Figure 2). There is a potential for deep neural networks to act as a replacement for software systems in various fields. However, for each new task, the neural networks should be rebuilt, re-trained, justified or re-tuned.

Contribution: Transferring the knowledge of software engineering to the new emerging concept of DNNs is an opportunity that we try to provide. We aim for connecting the communities of machine learning and variability management to motivate both fields in performing interdisciplinary research for applying reuse in DNNs which leads to save of time, resources and cost of re-training DNNs for new tasks. From a software engineering perspective, we are interested in factors such as cost (here for providing data and computational power for training DNNs), time-to-market, and quality. For example, eliminating ineffective parts of the neural network (if possible and detected) reduces the memory and power consumption and makes the system more efficient. This paper is a way to investigate the feasibility and performance of these factors.

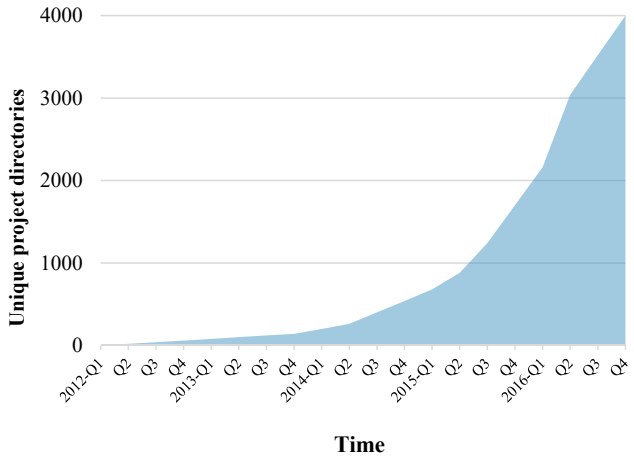

Figure 2: Growing use of deep neural networks at Google [5]

The remainder of this paper is structured as following. Section 2 explains the details of the challenge and evaluation methods for solutions. Section 3 introduces some related work, and Section 4 discusses the benefits of applying variability management methods to DNNs. Finally, we summarize our paper in Section 5.

2 THE CHALLENGE

In this paper, we ask the community of SPL researchers to apply concepts of SPLs to DNNs with well-defined commonalities and variabilities. By using the term "feature" we refer to the concept of feature-oriented software product lines [1]. In this domain, the features are the functionalities of software systems. We consider the functionalities of DNNs in analogy to features in software systems. As mentioned before, one of the use cases of the neural networks is in image processing as classifiers. Let us assume a neural network that identifies and classifies animals such as dogs and cats. In this example, the input layer of DNN receives a matrix of a 2-D image. In the elementary layers, the primary components including the edges and the corners are extracted. Two neurons in the output layer indicate the probabilities of having a cat or a dog in the input image.

A functionality of a DNN can be the prediction of the class of a certain object in an image that is given as the input. A DNN capable of recognizing (predicting or classifying) three types of animals in a picture has then three functionalities. We explain our notion of functionalities with an example of VGG 16 DNN. This network is pre-trained on the ImageNet dataset capable of recognizing 1000 different object categories (e.g., tiger cat, blue tick, sea lion, sorrel,

[2]http://cs231n.github.io/convolutional-networks/
[3]https://www.nvidia.com/en-us/self-driving-cars/
[4]https://keras.io/

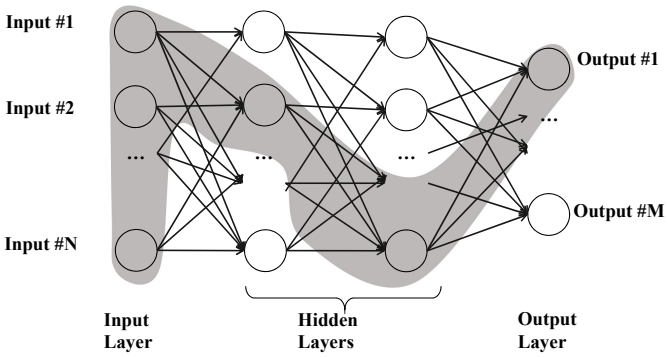

Figure 3: An example of a part of a DNN as a subset of its architecture

...). For instance, the capability to recognize a Persian cat and a zebra are two different functionalities. A list of all 1000 categories can be found online[5].

The term "feature" appears in both software engineering and for feature mapping in deep learning domains. We avoid the confusion by using features only in the context of software engineering and the term functionality when we discuss the neural networks. Both terms—feature and functionality—refer to the same concept but in different domains.

The challenge is in three levels of increasing complexities. The participants of the challenge should report their solutions, results and faced challenges during the adoption process. We welcome suggestions from the participants regarding software configuration methods that allow DNNs to be configured for reusability without additional re-training.

2.1 First Challenge: Feature Extraction

Since the extraction and specification of features is a challenge in product line engineering, the participants are asked to propose solutions for extraction of functionalities from DNNs. We use the term "parts" in this paper as subsets of a neural network's architecture [7]. We also consider a subset of the hyperparameters—weights of the edges—as parts. Figure 3 highlights a part in a neural network. We explain this challenge with an example of a simple pre-trained network capable of recognizing cats and dogs in an input image. The part in charge of recognizing a cat consists of the subsets of the network's architecture and hyperparameters that are activated if a given image to the network contains a picture of a cat.

Challenge: We ask participants to take a sample pre-trained network which can be found online in imageNet[6] with 1000 classification functionalities. Examples of such models are available in Keras documentation[7] [2]. Some of the pre-trained models are Xception [3], VGG16 and VGG19 [20], ResNet50 [8], InceptionV3 [21], and MobileNet [9].

[5]http://image-net.org/challenges/LSVRC/2014/browse-synsets
[6]http://www.image-net.org/
[7]https://keras.io/applications/

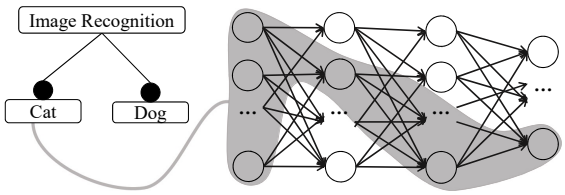

Figure 4: Example of binding the parts of a DNN to a feature model

We see two possibilities for solving this challenge; first, trying to identify the common parts between the members of a set of trained neural networks which are activated if a certain input (e.g. *persian_cat*) is given to these networks. For example, taking ResNet50, VGG19 and VGG16 as a set and trying to feed all of them with the same group of input images and identifying the parts of network which will be activated or used to take a decision. The commonalities between the identified parts of the networks have to be found next. The second possibility is to take one network to specify the boundaries of functionalities within the network. For example, we can take VGG19 and try to determine responsible parts for recognition of each category of objects, e.g., responsible parts for recognizing Persian cat.

Solutions could be manual, semi-automated or fully automated to extract the corresponding parts which are responsible for each functionality.

Evaluation: Solutions for the raised challenge can be evaluated based on several criteria. We favor the solutions with least time taken during the extraction. Another metric to minimize is the level of human interventions. The solution with the least time consumption and human intervention is considered the highest quality solution.

2.2 Second Challenge: Feature Abstraction

The principle of uniformity [1] denotes that, not only the code fragments are considered as reusable assets in SPLs but the other artifacts such as documentation—including models and specifications—and test cases are reusable assets as well. Feature models [10] and orthogonal variability model [16] provide a level of abstraction to hide the complexity of the implementation of the features. These models handle the dependencies between features of product lines and the reusable assets required to implement them (including every required artifacts). In our notion of reusable assets of a DNN, each subset of a DNN that is responsible for a certain functionality should be bound to a feature in a feature model. This enables the application engineers to include or exclude some of the functionalities of a DNN to create a new DNN with reduced number of functionalities. Figure 4 shows an example of features of a product line bound to their implementations which are the parts of a DNN. In this example, the corresponding part for cat recognition in the input image is bound to the cat feature in feature diagram.

Various mechanisms are developed to facilitate the binding of feature in feature models to their corresponding reusable assets with less implementation efforts. A set of well-established methods are presented in feature oriented software product lines by Apel et al. [1]. These methods support the implementation of denoted variabilities and commonalities by a feature model. However, handling the parts of a DNN as reusable assets is not considered in existing methods.

Challenge: In this level of the challenge, we are expecting the participants to introduce a method to assign the extracted parts of DNN to a feature model. Selecting or deselecting a feature in feature model should include or exclude the corresponding part of the DNN that are responsible for mentioned functionality by that feature. For this purpose, an implementation of a feature model using existing feature modeling tools (e.g., FeatureIDE [12]) is required.

Evaluation: We expect the participants to apply known methods for implementing commonalities and variabilities from SPLs to DNN. Apel et al. introduced and explained these methods in their book on "*Feature-Oriented Software Product Lines*" [1]. These methods are Parameters, Design Patterns, Frameworks, Components and Services, Version-Control Systems, Build Systems, Preprocessors, Feature-Oriented Programming, Aspect-Oriented Programming, Delta-Oriented Programming, and Context-Oriented Programming. Referring to each of these methods to implement the variability in DNN will be counted as a positive point for evaluation. Furthermore, introducing a new method for implementing the variability in DNNs is considered as a plus as well. For example, a possible solution, that is implemented with Parameters method, takes a pre-trained neural network (e.g. VGG16) and a list of parameters to enable or disable the functionalities of that network (e.g., Cat=$true$, Dog=$false$, ...), and generates a new network with desired functionalities. This type of solution takes a positive point in the evaluation. Reengineering a network to apply a design pattern for adding or removing some observer for certain functionalities is also a plus. Granularity of the extracted parts is the next metric for efficiency of a proposed solution. The number of the functionalities included in an extracted part of the network defines the granularity. We prefer the solutions with higher number of parts that explicitly perform a set of few functionalities. Assume A is a solution that extracts two parts from the given network, the first part performs one functionality and the second part performs two (i.e., $P_A = \{1, 2\}$). Similarly, there are two other solutions (B and C) with $P_B = \{1, 1, 1\}$ and $P_C = \{1, 2, 1\}$. Based on our evaluation method, C is the best solution followed by B. Equation 1 is a formal definition for our quantitative approach to evaluate the solutions.

$$Score = \sum_{x \in P} x \times \sum_{x \in P} \frac{1}{x} \quad , \tag{1}$$

where P is the set of functionalities performed by the parts in a solution. The scores of the solutions in our previous example will be $Score(A) = (1 + 2) \times (1/1 + 1/2) = 4.5$, $Score(B) = 3 \times 3 = 9$, and $Score(C) = 4 \times 2.5 = 10$.

2.3 Third Challenge: Product Line of DNNs

This challenge corresponds to the promising capability of SPLs to generate software products from existing well managed software assets.

Challenge: At this stage of the challenge, participants are asked to create a product line from DNNs. To solve this challenge, the results of the previous steps should be put together in order to produce a composed neural network. The functionalities of the generated DNN is the same as the functionalities of the neural networks extracted, abstracted, and decomposed in the previous stages of the challenge. Suppose M is a DNN that classifies images of dogs and cats and N is another DNN that classifies crows and eagles. There has to be the possibility of generating a network that supports a combination of these functionalities. From M and N a new network O can be built that identifies crows and dogs or another network P that recognizes crows, eagles, and cats but dogs.

Evaluation: The size of the generated neural network should be minimal. Any redundancy increases the size of the generated neural network. The accuracy of the final neural networks should not be much lower than the base neural networks for the same input. The solutions should have a feature model to bind the variable and common assets or parts of the network.

3 RELATED WORK

Kruger et al. [13] proposes the challenge of generating a product line from APO-Game clones. They support extracting features in a set of cloned software products which is different to the extraction of features in the application domain of machine learning techniques. The state-of-the-art lacks the opportunities emerging from deep learning for the field of SPLs.

Deep transfer learning [15] is a method to reinforce or induce a trained DNN to reuse it for a different task [22]. DNNs trained with a general dataset for a general task are refined or re-tuned for a special task. For example, a DNN trained for a general image processing task is then refined for face recognition (as a subset of image processing domain). This method reduces the time and costs for training DNNs from scratch for each new task. Companies like Google train the models and provide them so that everyone can reuse the pre-trained neural networks in their own domains such as image processing[8] or natural language processing[9].

Transferred learning is an effective approach if the available dataset for training the network is not big enough or the infrastructure does not provide the required computational power to perform the training. In transferred learning a neural network can be reused as a subset of another network only after being re-training with the most recent data. In our challenge, we want to apply the reuse possibilities for neural networks without the need for re-training.

Catastrophal forgetting in the neural networks is a known issue [14, 17] where the networks forget the old skills while training for new skills due to the change in the weights of the edges of the networks. Ellefsen et al. [6] apply the modular neural networks [7] to solve the issue by reinforcing them with various solutions. The

[8] https://image-net.org
[9] https://wordnet.princeton.edu

possibility to apply these methods for the DNNs is not investigated yet.

4 DISCUSSION

Sculley et al. [19] investigated the drawbacks of using machine learning, specifically the DNNs, as a part of a software system. They address the risk factors raised by any change in the requirements or specifications of system that is based on machine learning techniques. Knowing this issue, here we explain how our notion of applying SPL concepts to DNNs help to overcome this problem. Separating and abstracting the functionalities of a DNN allow a new network to form from a combination, composition, or subtraction of the network functionalities without re-training. In case of an erosion of boundaries, only the affected parts need to be replaced. It takes less time, cost, and effort to train and test the new neural network. The same applies to the problem of Entanglement. Despite of CACE (change anything, change everything) concept, any accuracy improvement in certain parts of the network can be performed without affecting or re-training the whole network. Correction cascades can be prevented by our notion of features.

Instead of using the old models as a basis for new ones, the parts of the system that are not relevant anymore will be replaced. Dependencies to the old network can be prevented and the undeclared consumers can be covered partly with our method. When a functionality changes, the undeclared consumers are affected as well, if they use the altered functionalities. Hidden feedback loops and data dependencies will be easier to detect and remove if the corresponding parts of the network for each feature or functionality is already determined.

Changes in the external world is a general problem of the software systems. In DNNs, any change in the requirements of the system leads to re-training the whole system even when a change is in one or some functionalities of the network. This problem could be handled if the relation between network parts and their functionality is clearly explained. Under these circumstances, only the parts of the network will be changed or updated that their corresponding functionalities are in correlation with changes in outside world. The configuration issues and variety of system-level anti-patterns mentioned by Sculley et al. [19] are not supported directly by our challenge proposal. One can implement a DNN for each functionality, but it would suffer from the similar shortages in the methods for service composition [1] that leads to a redundancy and waste of memory and energy.

5 SUMMARY

In this paper, we proposed the challenge of applying techniques and concepts of SPLs to the DNNs. We challenge the researchers in both fields to find a solution to specify, extract, and combine the functionalities from the DNNs to enable the reuse in building a product line. Solving this challenge can improve the preparation of data, time, and costs of the resources for training a DNN for every given task. The proposed solutions will be evaluated based on the promised benefits of the product line engineering—time, cost, and quality. The solutions will be collected in a repository to provide a benchmark for evolution and evaluation of the new methods of applying concepts of the SPLs to the DNNs.

REFERENCES

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2016. *Feature-oriented software product lines*. Springer.

[2] François Chollet. 2017. Keras Documentation. 2016. (2017).

[3] François Chollet. 2017. Xception: Deep learning with depthwise separable convolutions. *arXiv preprint* (2017), 1610–02357.

[4] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool features and tough decisions: a comparison of variability modeling approaches. In *Proceedings of the sixth international workshop on variability modeling of software-intensive systems*. ACM, 173–182.

[5] Jeff Dean. 2017. AIFrontiers:Trends and Developments in Deep Learning Research. (2017). https://www.slideshare.net/AIFrontiers/jeff-dean-trends-and-developments-in-deep-learning-research

[6] Kai Olav Ellefsen, Jean-Baptiste Mouret, and Jeff Clune. 2015. Neural modularity helps organisms evolve to learn new skills without forgetting old skills. *PLoS computational biology* 11, 4 (2015), e1004128.

[7] Frédéric Gruau. 1994. Automatic definition of modular neural networks. *Adaptive behavior* 3, 2 (1994), 151–183.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[9] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[10] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

[11] Kyo C Kang, Jaejoon Lee, and Patrick Donohoe. 2002. Feature-oriented product line engineering. *IEEE software* 19, 4 (2002), 58–65.

[12] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. 2009. FeatureIDE: A tool framework for feature-oriented software development. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 611–614.

[13] Jacob Krüger, Wolfram Fenske, Thomas Thüm, Dirk Aporius, Gunter Saake, and Thomas Leich. 2018. Apo-games: a case study for reverse engineering variability from cloned Java variants. In *Proceeedings of the 22nd International Conference on Systems and Software Product Line-Volume 1*. ACM, 251–256.

[14] Michael McCloskey and Neal J Cohen. 1989. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*. Vol. 24. Elsevier, 109–165.

[15] Sinno Jialin Pan, Qiang Yang, et al. 2010. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering* 22, 10 (2010), 1345–1359.

[16] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.

[17] Roger Ratcliff. 1990. Connectionist models of recognition memory: constraints imposed by learning and forgetting functions. *Psychological review* 97, 2 (1990), 285.

[18] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural networks* 61 (2015), 85–117.

[19] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. In *Advances in neural information processing systems*. 2503–2511.

[20] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[21] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.

[22] Lisa Torrey and Jude Shavlik. 2010. Transfer learning. In *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*. IGI Global, 242–264.

[23] David M Weiss and Chi Tau Robert Lai. 1999. *Software product-line engineering: a family-based software development process*. Vol. 12. Addison-Wesley Reading.