Erik Brorsson – My research

I'm working as an industrial PhD in Volvo Group Trucks Operations (the manufacturing organization of Volvo trucks). In our factories, we are dealing with increasing amounts of autonomous systems, constituting for example kitting/bin-picking robots and autonomous transport robots for internal logistics. Although these autonomous agents may replace some of the human workforce, many tasks cannot be handled by such autonomous systems and will remain for human workers. Therefore, these robots must be able to work side by side, and in some cases collaborate, with human operators. Since the robots operate in a continuously evolving environment, they must be equipped with the capability of perceiving the environment and acting accordingly. For example, this could include detecting the current pose of a human operator and anticipating his/her future actions to avoid a collision/accident. To this end, I am developing camera-based perception algorithms that aim to provide autonomous robots with sufficient perception capabilities to safely navigate such an environment. Specifically, I am working on the task of semantic segmentation, which entails classifying each pixel in an image according to a predetermined set of classes. Typically, this task is tackled by collecting vast amounts of labeled/annotated data and training a deep neural network (e.g., convolutional neural network) to perform this task. While satisfactory performance often can be achieved when the dataset is large enough, this is a limitation of the method since labeled data is often scarce and expensive to collect. As such, much recent research has focused on complementing the labeled dataset with unlabeled data (semi-supervised learning) or utilizing synthesized data (domain adaptation). How this is done most effectively is an open research question which I am currently working on. Additionally, handling uncertainty in safety-critical perception systems is also a hot research topic, which presumably requires even more emphasis if one attempts to develop a reliable perception system based on unlabeled and/or synthesized data. My aim for the coming year is to investigate practical methods for safety-critical perception systems that can leverage both labeled and unlabeled data.

Two Principles/ideas/concepts from Robert's lectures

Verification and Validation: When developing a piece of software we typically want to ensure that it meets a set of requirements that we have specified. The process of testing whether such requirements are met is called verification. While meeting the requirements is often necessary in order to produce a well-functioning software, it may not be sufficient to meet the end-users/customers expectations about the software. In fact, it is often the case that a software does not meet the end-users expectation although it satisfies every requirement. This is because requirements may be difficult to define. Especially in complex systems where it is not clear how a set of requirements on a specific module would affect the performance/requirements on other modules. Therefore, it is also important to do validation, which involves testing/checking whether the software meets the end-users needs. In the case of my research, which revolves around navigation of autonomous robots in a dynamic environment, the end user may have an expectation of the system on a wholistic level, e.g., that the transport robots should deliver material with a certain efficiency and uptime. Trying to boil down these expectations into requirements of the different submodules, e.g., perception, path planning and control, which are all connected and interdependent, is indeed very difficult. I believe that a good strategy is to simply start with some requirements (an initial guess), develop the code according to these requirements, observe how the system behaves/get feedback from the customer, and then update the requirements if necessary. This defines an iterative process that involves frequent feedback from the customer, derivation of requirements from this feedback (stated expectations), and development of code that meets the requirements. Although this may sound straight forward, various problems arise when trying to verify code, not the least with software based on AI/ML. For example, a requirement on the perception system may be that it detects humans with a certain accuracy, regardless of the clothes the human is wearing or the environment in which he/she is in. Unfortunately, there is no other way to test if a deep neural network meets such requirements than to “brute force”, e.g., gather data from every possible scenario and test that the network produces the desired output for each of these inputs. Since the scenarios are typically endless, this is not feasible, and more research is needed to be able to verify the functioning of deep neural networks, especially to identify the “safe” operating domain of such algorithms.

Static vs Dynamic techniques

Methods of code verification can roughly be divided into two groups, namely static and dynamic methods. Static methods are defined as techniques for verifying the functionality of the code without executing it, while dynamic methods revolve around executing the code. Examples of static methods are manual review of the code (peer review) and automated static tests. Automated static tests can ensure that the code lives up to certain standards without executing the code. For example, a program can do a linting check, checking that the code lives up to the documentation requirements (docstrings etc), and check for test coverage (what proportion of the code is covered by automatic (dynamic) testing). It is my understanding that static testing in the form of code review is also common practice in the industry to certify code, e.g., ensure that it follows ISO standards etc. While static testing is obviously a corner stone of any software development, it may be difficult to unveil all problems and risks with a software solely with this technique. Dynamic testing, which involves code execution, is used to further verify that the code does what is intended. It is common to include automated dynamic tests for large portions of the software to achieve a high test coverage (measured in percentage of code that is being executed by the tests).

Two Principles/ideas/concepts from one of the three guest lectures

Boundary Value Testing: Typically, it is the boundary values that one would like to use for automatic testing. This is because while it is often easy to create a program that behaves satisfactorily on the bulk of data, it may be very difficult to get all the boundary values correctly. I connect this to my own research which revolves around computer vision. Deep learning models for image analysis typically performs well on the bulk of data, provided enough training examples. However, it fails in edge cases / boundary cases. To improve a deep learning model on the edge cases one typically needs to identify such cases/scenarios and collect training data that resamples these cases. Indeed, evaluating a model’s performance in practice boils down to evaluating the performance on such edge cases/ boundary values.

Boundary Value Exploration: Revolves around finding boundary values for a specific function/program. This is crucial for testing the functionality of a program. For some function with a low-dimensional input space it may be possible to find the boundary values relatively easy by doing an exhaustive search. However, when the input dimension is high, smarter search algorithms may be required in order to effectively cover the input space. In the case of computer vision, the input space is an array of pixel values, which is typically extremely high-dimensional considering images of reasonable resolution. This makes it difficult to perform boundary value exploration. Furthermore, we are also often interested in mining boundary values for input data that can occur during test time of the software, I.e. it may not be relevant to test the software on input which could never occur during test time. As such, we may also have some constraints on the input space, possibly making the search even harder. One way to find the boundary values of a computer vision algorithm could be to generate synthetic images and test the model against these. Obviously, it is nontrivial to generate synthetic images that resample real world data, so performing testing on such images to prove the efficacy of the model on real data is challenging.

Automated Software Testing

Software testing is the process of verifying that the code does what is intended/meets the specification. Traditionally, software testing requires extensive manual work in the form of skillfully designing test cases that covers the entire specification. In complex software, however, it is often difficult to understand what test cases are needed to ensure that the code meets the specification. Therefore, manual software testing may become too labor intensive to be practical, and instead one needs to turn to automated software testing, which comes with its own challenges. Automated software testing typically revolves around designing algorithms for automatically executing the piece of code that is being tested and checking whether the output fulfills the specifications, I.e., whether the output is faulty or correct. Depending on the software that is being tested, the use of automated testing may be more or less suitable. For some software/functions, we may be able to specify what the output should be for any input, I.e., we have access to an oracle. This definitely simplifies the use of automated testing, since regardless of what input is fed to the function we can always test whether the output is equivalent to the output of the oracle (I.e. meets the requirements). However, in this case it may still be difficult to generate valid inputs to the function that should be tested. This could for example be due to a too large input space, making I.e. a random search too time consuming to cover all relevant test cases. Or it could also be that the input itself must meet some requirements, e.g., that it should be “realistic”, which may be difficult to define. Indeed, generating valid inputs that constitute relevant test cases is an active research topic. While the existence of an oracle simplifies the automated testing, it may be possible to design automated tests even without an oracle. One idea is to for example check for inconsistencies in the behavior of the software/function when exposed to small perturbations in the input, for which the output is expected to be consistent. When testing deep learning models for e.g., image recognition, an oracle is typically not available, which makes these types of methods relevant.

In Audee: Automated Testing for Deep Learning Frameworks [1], Guo et al. developed a method for automated testing of four deep learning frameworks, TensorFlow, PyTorch, CNTK, and Theano. They show that their automated testing framework is efficient in detecting bugs in the frameworks. They use a genetic algorithm to generate inputs that produce inconsistent behavior of the deep neural network. Specifically, they search for inputs that result in crashes, NaNs or inconsistent behavior between the different frameworks (e.g., the result is not the same for Tensorflow and PyTorch). In their case, the input is RGB images, which makes the input space huge, and doing an exhaustive or random search on this space would not be efficient in finding bugs. This is the reason that use a genetic algorithm to find relevant test cases. Furthermore, like in many applications of deep neural network, they don’t have access to an oracle, so they cannot directly check whether the output is desirble or not for a specific input. This is the reason why they instead check for consistency between the frameworks, building on the assumption that the frameworks should produce similar outputs for the same inputs if the software is free from bugs. Although I am more interested in testing the deep learning models in my research, rather than the deep learning frameworks, I think that some of the ideas can be used also for this task. Specifically, I am interested in testing whether a deep neural network for semantic segmentation produces satisfactory output for all relevant inputs. Checking for consistency in the network's output for different perturbations of the same input could be a reasonable test. The input that result in inconsistency may be of particular interest as they are likely to cause incorrect predictions from the network. When these inputs are found, it may be checked manually if the network produces erroneous predictions for these inputs. If so, then such inputs may be manually labeled and added to the training set to avoid making this same mistake in the future. Some research challenges in this area are (I) generating input images that are realistic and (ii) finding the inputs that cause malfunction. Regarding (I) one could imagine perturbing real images with different transformations, such as rotating the image or changing the brightness. Such transformations would keep the input realistic and thus constituting valid test cases for the neural network. However, in practice, such transformations would only cover a very small subspace of the entire "valid" input space. To be able to cover larger portions of the input space, one could perhaps turn to new techniques, such as diffusion models, to generate synthetic images from a wide distribution. Regarding (ii), a genetic algorithm may be suitable if the perturbations can be parameterized by a reasonable number of chromosomes. However, in the case of generating input images through a diffusion model, this is hardly the case, and I believe that future research will also be directed to this issue.

Quality Assurance

While testing of code discussed in the previous section is surely necessary to achieve a high-quality software, it is not sufficient as it does not consider other aspects of software development. Quality assurance basically aims to monitor all processes of software development to ensure a high-quality output. This includes for example requirements engineering (which serves as the foundation for any subsequent testing), software design, release management, etc. In "Quality Assurance for Machine Learning – an approach to function and system safeguarding" by Poth et. Al [2], they describe a method (EvAIa) for quality assurance in the context of machine learning. Their approach consists of four steps: (I) identifying potential quality issues and defining a mitigation scope, (ii) answering a questionaire that comes with the method to systematically identify weaknesses with the machine learning component, (iii) deciding on mitigation actions, which could include both quality by design and specific validation/verification tests, and (iv) the defined actions are documented through the entire lifecycle of the product. The questionnaire consists of three separate set of questions, each considering a separate stage in the lifecycle of the machine learning model. The first questionnaire considers pre-processing, revolving around data collection, inherent biases, completeness of the training set (what are the bounderies of the dataset, can they change over time etc), training validation splits, and regulation and compliences. The second questionnaire considers the actual implementation of the machine learning model: I.e., is the software for training developed with quality assurance guidelines, how are hyperparameters tuned, how can the robustness of the model be measured etc. The third questionnaires concernsdeployment/serving of the machine learning model: how is the model's output monitored, how are peak load resources made available, does the code for serving undergo sufficient testing etc.

While we can see that (automated) testing also may play an important role in quality assurance, quality assurance is a much broader term and also includes other methods/tools. In EvAIa, proposed by Poth et. Al, they put much emphasis on the data collection part of the machine learning model life-cycle. Personally, I believe that it is good to focus on this since much else of the machine learning life-cycle is already managed very easily with packages such as pytorch. Model training and testing typically don't constitute more than a few hundred lines of code, making it easy to verify the correctness of it. However, what is difficult to quality assure is the data collection, e.g., that the data is not biased, has sufficient coverage etc. Indeed, also the third part of the questionnaire proposed by Poth et al, which focuses on model serving, also includes relevant points. I believe that in practice, although the code that constitutes the model is relatively simple, the environment in which it is deployed may be very complex and requires rigorous quality assurance measures. For example, when deploying a model in a distributed environment that must scale up and down with demand in e.g. a kubernetes cluster. Therefore, I believe that further research on quality assurance of machine learning models will mainly go into the data collection and deployment aspects. Although Poth et al mentions the boundaries of the data an important aspect, they do not propose a method as to how investigate this boundary. I believe that boundary value exploration and testing is a fundamental problem/risk when it comes to machine learning models and there are great opportunities for research on this topic in the area of quality assurance.

Future trends and directions of Software Engineering in relation to my topic

I believe that software engineering will fundamentally go from being a very human labor-intensive work to more and more automated. For example, I believe that in a few years' time there will be AI text/code analysis tools that help with testing, for example based on a few lines of documentation provided by the software engineer. Such tools could perhaps reduce the risk of malfunctioning code and reduce the workload of the software engineers by freeing up time that traditionally would be spent on code review and designing dynamic tests. On the other hand, it also seems likely that AI models will in greater extent even write the code for us in the future. In this case, the verification of the code will perhaps be left to the software engineers, which in some sense contradicts the above statement. I believe that it will very much depend on the use case as to what extent the AI is allowed to write code, perform the verification, or do both. With respect to AI/ML engineering I believe that there will be a shift in focus from designing neural networks to data cleaning and model verification. Specifically, I believe that the models will continue to grow in size (and perhaps complexity), which will make it virtually impossible (and indeed unnecessary) for a solo practitioner and even smaller companies to develop their own model (I.e. unique architecture) that outperforms the best publicly available models. This means that there will be no need to think about the optimal architecture design for the specific use case, but rather a successful implementation of an AI model will primarily revolve around collecting sufficient data and verifying/validating that the AI serves its purpose. Indeed, it is already the case that off-the-shelf models with impressive performance are readily available for many common tasks. However, they are typically very niche and solve a specific task, so for any tasks that these models cannot handle there are still opportunities for AI/ML researchers to develop new models.

References

[1] Guo, Qianyu & Xie, Xiaofei & Li, Yi & Zhang, Xiaoyu & Liu, Yang & Li, Xiaohong & Shen, Chao. (2020). Audee: automated testing for deep learning frameworks. 486-498. 10.1145/3324884.3416571.

[2] A. Poth, B. Meyer, P. Schlicht and A. Riel, "Quality Assurance for Machine Learning – an approach to function and system safeguarding," *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, Macau, China, 2020, pp. 22-29, doi: 10.1109/QRS51102.2020.00016