Building Machine Learning Models like Open-Source Software

Colin Raffel

Abstract

Transfer learning – i.e., using a machine learning (ML) model that has been pre-trained as a starting point for training on a different, but related task – has proven itself as an effective way to make models converge faster to a better solution with less labeled data. These benefits have led pre-trained models to see a staggering amount of reuse; for example, the pre-trained BERT model has been downloaded tens of millions of times.

Taking a step back, however, reveals a major issue with the development of pre-trained models: They are never updated! Instead, after being released, they are typically used as-is until a better pre-trained model comes along. There are many reasons to update a pre-trained model – for example, to improve its performance, address problematic behavior and biases, or make it applicable to new problems – but there is currently no effective approach for updating models. Furthermore, since pre-training can be computationally expensive (for example, training the GPT-3 model was estimated to cost millions of dollars), many of the most popular pre-trained models were released by small resource-rich teams. The majority of the ML research community is therefore excluded from the design and creation of these shared resources. Past community efforts like the SahajBERT and Big Science BLOOM models show that there is a desire for community-led model development, but there is no mature tooling to support their development or continual improvement. Contrast this with the development of open-source software. For example, if the open-source Python programming language had remained frozen after its first release, it would not support boolean variables, sets, Unicode, context managers, generators, keyword arguments, or many other widely-used features. It also would not include thousands of bugfixes that were contributed by members of the distributed community of Python developers over the past few decades. This kind of large-scale distributed collaboration is made possible through a mature set of tools and concepts, including version control, continuous integration, merging, and more.

This Viewpoint advocates for tools and research advances that will allow pre-trained models to be built in the same way that we build open-source software. Specifically, models should be developed by a large community of stakeholders who continually update and improve them. Realizing this goal will require porting many ideas from open-source software development to the building and training of pre-trained models, which motivates many new research problems and connections to existing fields.

Incremental and cheaply-communicable updates

Modern machine learning models are often trained through gradient descent. Crucially, gradient descent typically involves updating *all* of the parameters at *every* training iteration. As such, communicating any change made during training requires communicating every parameter's value. Unlike the incremental patches used to update code in version control, gradient descent makes it infeasible to keep track of the complete history of every parameter value. Fortunately, a variety of past work has shown that it is possible to effectively train models using small and cheaply-communicable parameter updates. These techniques could provide a helpful starting point for enabling community-developed models.

One motivation for communication-efficient training has been to reduce costs in distributed optimization, where individual workers train local copies of a model and must communicate their changes to a centralized server. This has led to schemes that either reduce the frequency of communication (i.e. having workers train for more iterations before they communicate their updates) or reduce the size of each update through quantization, sparsification, or other approximations [1, 2].

It has also been demonstrated that a model can be improved or changed by adding or removing parameters. One example is the Net2Net framework of Chen et al. [3], which describes ways of adding parameters to a neural network that does not change the function the model originally computed. Relatedly, adapters [4] are tiny trainable subnetworks that are added to a model. By only updating the adapter module parameters and leaving the rest of the model's parameters fixed, the model can be trained in a parameter-efficient manner.

Finally, certain "modular" model architectures may be more amenable to cheaply-communicable updates because their architecture allows selectively updating a subset of parameters. For example, models that involve conditional computation [5] among submodules (such as the sparsely-gated mixture-of-experts layer [6]) can have individual submodels added, removed, or updated while the remainder of the parameters remain fixed. Alternatively, models that make predictions by querying a collection of data (such as a k-nearest-neighbor classifier) can be incrementally modified by changing the data itself.

Merging models

In distributed open-source software development, "merge conflicts" occur when a contributor changes an out-of-date copy of the codebase or when multiple contributors introduce conflicting changes. A similar situation arises in distributed optimization of machine learning models: Since individual workers compute many local updates before communicating their changes, the individual models on each worker can become significantly different over time. A strong baseline for combining (or "merging") disparate updates from different workers is to average together the updates from each of the workers when aggregating changes [7, 8], though averaging can degrade performance when individual workers are training on differently-distributed data. Using specialized model architectures could also provide a more principled way of combining updates.

For example, updates to different submodels in a model using conditional computation would not conflict. Further investigation into modular model architectures will help clarify the situations where updates can be merged without a degradation in performance.

Testing proposed changes

Given the ability for contributors to propose changes to a model, a natural question arises as to how to decide when maintainers should accept a change. In open-source software development, this is typically done through the aid of automated testing, where a proposed change undergoes a battery of public tests to make sure the software would continue to work as intended. In contrast, the utility and validity of a machine learning model is typically measured in terms of whether it suits a given application and delivers satisfactory results, which is less straightforward to test.

A possible means of testing a pre-trained model would be to measure its performance after fine-tuning on a downstream task. However, fine-tuning a model on a task typically requires many iterations of training and the utility of a pre-trained model is typically measured on dozens of tasks. Testing each change to a community-developed model through fine-tuning and evaluation on downstream tasks could therefore become excessively expensive. One way to mitigate this cost would be to make fine-tuning dramatically cheaper. For example, if we fine-tune a pre-trained model, perform additional pre-training, and then fine-tune it again, we might hope to re-use the updates from the first fine-tuning run. This idea bears some similarity to “merging models” in the sense that we want to merge updates from the first fine-tuning run into the updated pre-trained model. We also might hope to use some of the advantages of modular architectures - for example, adapter modules created for a given pre-trained model might be reusable after the model undergoes additional pre-training. Alternatively, multi-task learning (where a model is trained to be able to perform many tasks) provides an alternative to the pre-train-then-fine-tune paradigm which could make testing as simple as evaluating the model on all of the tasks it targets. Recent results have shown that pre-trained language models can perform well on many tasks at once [9], though multi-task performance often lags behind the performance of models trained separately on each individual task.

Even with cheaper fine-tuning and evaluation, determining how to rigorously test a contribution will require clarifying what it means to test machine learning models. As Christian Kästner noted [10], "the challenge is how to evaluate whether the model is 'acceptable' in some form, not whether it is 'correct'." Developing ways to specify the intended uses and limitations of models will help maintainers articulate what kinds of contributions are welcome.

Versioning, backward compatibility, and modularity

The semantic versioning system provides a standardized way of communicating the significance of a new version of a piece of software. A similar system could likely be devised for community-developed models. For example, patch releases could correspond to changes that

only modify (a small subset) of a model's parameters while guaranteeing no significant degradation of performance on supported downstream tasks, minor releases could possibly change the architecture of the model while retaining the majority of its parameters, and major releases could completely replace the existing model and its parameters while targeting the same use-cases. "Backwards compatibility" of a pre-trained model most naturally maps to the notion of maintaining intended performance on targeted downstream tasks and that the model is applicable to the same types of inputs and outputs.

The ability to incorporate open-source packages into a piece of software allows developers to easily add new functionality. This kind of modular re-use of subcomponents is currently rare in machine learning models. In a future where continuously-improved and backward-compatible models are commonplace, it might be possible to greatly improve the modularity of machine learning models. For example, a core "natural language understanding" model could be augmented with an input module that allows it to process a text in a new language, a "retrieval" module that allows it to look up information in Wikipedia, or a "generation" output module that allows it to conditionally generate text. Including a shared library in a software project is made significantly easier by package managers, which allow a piece of software to specify which libraries it relies on. Modularized machine learning models could also benefit from a system for specifying that a model relies on specific subcomponents. If a well-defined semantic versioning system is carefully followed, models could further specify which version of their dependencies they are compatible with.

Conclusion

The use of pre-trained models for transfer learning has ushered in a golden era in many applications of machine learning. However, the development of these models is still in the dark ages compared to best practices in software development. Well-established concepts from open-source software development provide inspiration for methods that allow building continually improved and collaboratively developed pre-trained models. These connections motivate research related to existing topics like continual learning, multitask learning, distributed optimization, federated learning, modular architectures, and more. Building on advances in these fields will provide a jump-start towards this new mode of model development. In the longer term, the ability for a distributed community of researchers to build pre-trained models will help democratize machine learning by shifting power away from large corporate entities working in isolation.

If you are interested in discussing and contributing to this research direction, please join our community at https://bit.ly/cccml-community.

References

[1] Konecný, Jakub, et al. "Federated Learning: Strategies for Improving Communication Efficiency." *Private Multi-Party Machine Learning* (2016).

[2] Sung, Yi-Lin, Varun Nair, and Colin Raffel. "Training Neural Networks with Fixed Sparse Masks." *Advances in Neural Information Processing Systems* (2021).
[3] Chen, Tianqi, Ian Goodfellow, and Jonathon Shlens. "Net2net: Accelerating Learning via Knowledge Transfer." *International Conference on Learning Representations* (2016).
[4] Rebuffi, Sylvestre-Alvise, Hakan Bilen, and Andrea Vedaldi. "Learning Multiple Visual Domains with Residual Adapters." *Advances in Neural Information Processing Systems* (2017).
[5] Bengio, Yoshua, Nicholas Léonard, and Aaron Courville. "Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation." *arXiv preprint arXiv:1308.3432* (2013).
[6] Shazeer, Noam, et al. "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer." *International Conference on Learning Representations* (2017).
[7] McMahan, Brendan, et al. "Communication-Efficient Learning of Deep Networks from Decentralized Data." *Artificial Intelligence and Statistics* (2017).
[8] Matena, Michael, and Colin Raffel. "Merging Models with Fisher-Weighted Averaging." *arXiv preprint arXiv:2111.09832* (2021).
[9] Sanh, Victor, et al. "Multitask Prompted Training Enables Zero-Shot Task Generalization." International Conference on Learning Representations (2022).
[10] Kästner, Christian. "Machine Learning is Requirements Engineering — On the Role of Bugs, Verification, and Validation in Machine Learning." *Analytics Vidhya* (2022).