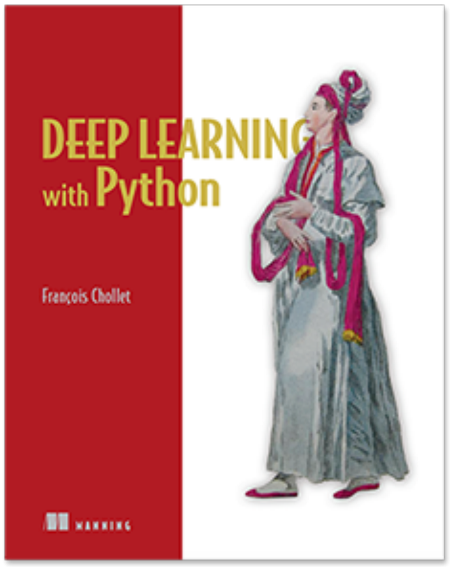
The future of deep learning

Tue 18 July 2017
By Francois Chollet
In Essays.

This post is adapted from Section 3 of Chapter 9 of my book, Deep Learning with Python (Manning Publications).



It is part of a series of two posts on the current limitations of deep learning, and its future. You can read the first part here: The Limitations of Deep Learning.

Given what we know of how deep nets work, of their limitations, and of the current state of the research landscape, can we predict where things are headed in the medium term? Here are some purely personal thoughts. Note that I don't have a crystal ball, so a lot of what I anticipate might fail to become reality. This is a completely speculative post. I am sharing these predictions not because I expect them to be proven completely right in the future, but because they are interesting and actionable in the present.

At a high-level, the main directions in which I see promise are:

- Models closer to general-purpose computer programs, built on top of far richer primitives than our current differentiable layers—this is how we will get to *reasoning* and *abstraction*,

the fundamental weakness of current models.
- New forms of learning that make the above possible—allowing models to move away from just differentiable transforms.
- Models that require less involvement from human engineers—it shouldn't be your job to tune knobs endlessly.
- Greater, systematic reuse of previously learned features and architectures; meta-learning systems based on reusable and modular program subroutines.

Additionally, do note that these considerations are not specific to the sort of supervised learning that has been the bread and butter of deep learning so far—rather, they are applicable to any form of machine learning, including unsupervised, self-supervised, and reinforcement learning. It is not fundamentally important where your labels come from or what your training loop looks like; these different branches of machine learning are just different facets of a same construct.

Let's dive in.

Models as programs

As we noted in our previous post, a necessary transformational development that we can expect in the field of machine learning is a move away from models that perform purely *pattern recognition* and can only achieve *local generalization*, towards models capable of *abstraction* and *reasoning*, that can achieve *extreme generalization*. Current AI programs that are capable of basic forms of reasoning are all hard-coded by human programmers: for instance, software that relies on search algorithms, graph manipulation, formal logic. In DeepMind's AlphaGo, for example, most of the "intelligence" on display is designed and hard-coded by expert programmers (e.g. Monte-Carlo tree search); learning from data only happens in specialized submodules (value networks and policy networks). But in the future, such AI systems may well be fully learned, with no human involvement.

What could be the path to make this happen? Consider a well-known type of network: RNNs. Importantly, RNNs have slightly less limitations than feedforward networks. That is because RNNs are a bit more than a mere geometric transformation: they are geometric transformations *repeatedly applied inside a `for` loop*. The temporal `for` loop is itself hard-coded by human developers: it is a built-in assumption of the network. Naturally, RNNs are still extremely limited in what they can represent, primarily because each step they perform is still just a differentiable geometric transformation, and the way they carry information from step to step is via points in a continuous geometric space (state vectors). Now, imagine neural networks that would be "augmented" in a similar way with programming primitives such as `for` loops—but not just a single hard-coded `for` loop with a hard-coded geometric memory, rather, a large set of programming primitives that the model would be free to manipulate to expand its processing function, such as `if` branches, `while` statements, variable creation, disk storage for long-term memory, sorting operators, advanced datastructures like lists, graphs, and hashtables, and many more. The space of programs that such a network could represent would be far broader than what can be represented with current deep learning models, and some of these programs could achieve superior generalization power.

In a word, we will move away from having on one hand "hard-coded algorithmic intelligence" (handcrafted software) and on the other hand "learned geometric intelligence" (deep learning). We will have instead a blend of formal algorithmic modules that provide *reasoning and abstraction* capabilities, and geometric modules that provide *informal intuition and pattern recognition* capabilities. The whole system would be learned with little or no human involvement.

A related subfield of AI that I think may be about to take off in a big way is that of *program synthesis*, in particular neural program synthesis. Program synthesis consists in automatically generating simple programs, by using a search algorithm (possibly genetic search, as in genetic programming) to explore a large space of possible programs. The search stops when a program is found that matches the required specifications, often provided as a set of input-output pairs. As you can see, is it highly reminiscent of machine learning: given "training data" provided as input-output pairs, we find a "program" that matches inputs to outputs and can generalize to new inputs. The difference is that instead of learning parameter values in a hard-coded program (a neural network), we generate *source code* via a discrete search process.

I would definitely expect this subfield to see a wave of renewed interest in the next few years. In particular, I would expect the emergence of a crossover subfield in-between deep learning and program synthesis, where we would not quite be generating programs in a general-purpose language, but rather, where we would be generating neural networks (geometric data processing flows) *augmented* with a rich set of algorithmic primitives, such as `for` loops—and many others. This should be far more tractable and useful than directly generating source code, and it would dramatically expand the scope of problems that can be solved with machine learning—the space of programs that we can generate automatically given appropriate training data. A blend of symbolic AI and geometric AI. Contemporary RNNs can be seen as a prehistoric ancestor to such hybrid algorithmic-geometric models.

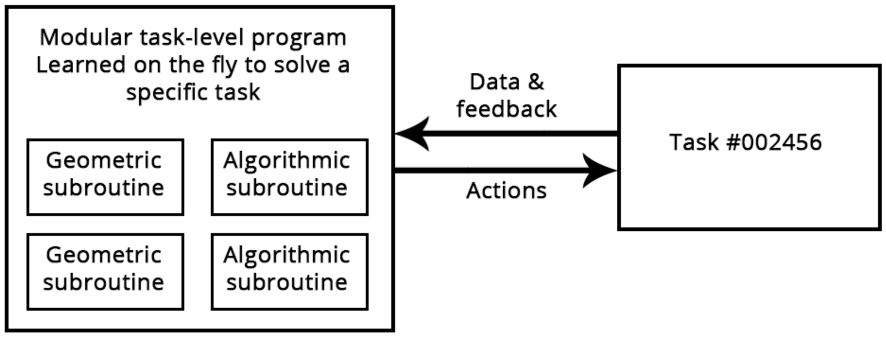


Figure: *A learned program relying on both geometric primitives (pattern recognition, intuition) and algorithmic primitives (reasoning, search, memory).*

Beyond backpropagation and differentiable layers

If machine learning models become more like programs, then they will mostly no longer be differentiable—certainly, these programs will still leverage continuous geometric layers as subroutines, which will be differentiable, but the model as a whole would not be. As a result, using backpropagation to adjust weight values in a fixed, hard-coded network, cannot be the method of choice for training models in the future—at least, it cannot be the whole story. We need to figure out to train non-differentiable systems efficiently. Current approaches include genetic algorithms, "evolution strategies", certain reinforcement learning methods, and ADMM (alternating direction method of multipliers). Naturally, gradient descent is not going anywhere —gradient information will always be useful for optimizing differentiable parametric functions. But our models will certainly become increasingly more ambitious than mere differentiable parametric functions, and thus their automatic development (the "learning" in "machine learning") will require more than backpropagation.

Besides, backpropagation is end-to-end, which is a great thing for learning good chained transformations, but is rather computationally inefficient since it doesn't fully leverage the modularity of deep networks. To make something more efficient, there is one universal recipe: introduce modularity and hierarchy. So we can make backprop itself more efficient by introducing decoupled training modules with some synchronization mechanism between them, organized in a hierarchical fashion. This strategy is somewhat reflected in DeepMind's recent work on "synthetic gradients". I would expect more more work along these lines in the near future.

One can imagine a future where models that would be globally non-differentiable (but would feature differentiable parts) would be trained—grown—using an efficient search process that would not leverage gradients, while the differentiable parts would be trained even faster by taking advantage of gradients using some more efficient version of backpropagation.

Automated machine learning

In the future, model architectures will be learned, rather than handcrafted by engineer-artisans. Learning architectures automatically goes hand in hand with the use of richer sets of primitives and program-like machine learning models.

Currently, most of the job of a deep learning engineer consists in munging data with Python scripts, then lengthily tuning the architecture and hyperparameters of a deep network to get a working model—or even, to get to a state-of-the-art model, if the engineer is so ambitious. Needless to say, that is not an optimal setup. But AI can help there too. Unfortunately, the data munging part is tough to automate, since it often requires domain knowledge as well as a clear high-level understanding of what the engineer wants to achieve. Hyperparameter tuning, however, is a simple search procedure, and we already know what the engineer wants to achieve in this case: it is defined by the loss function of the network being tuned. It is already common practice to set up basic "AutoML" systems that will take care of most of the model knob tuning. I even set up my own years ago to win Kaggle competitions.

At the most basic level, such a system would simply tune the number of layers in a stack, their order, and the number of units or filters in each layer. This is commonly done with libraries such as Hyperopt, which we discussed in Chapter 7 (Note: of Deep Learning with Python). But we can also be far more ambitious, and attempt to learn an appropriate architecture from scratch, with as few constraints as possible. This is possible via reinforcement learning, for instance, or genetic algorithms.

Another important AutoML direction is to learn model architecture jointly with model weights. Because training a new model from scratch every time we try a slightly different architecture is tremendously inefficient, a truly powerful AutoML system would manage to evolve architectures at the same time as the features of the model are being tuned via backprop on the training data, thus eliminating all computational redundancy. Such approaches are already starting to emerge as I am writing these lines.

When this starts happening, the jobs of machine learning engineers will not disappear—rather, engineers will move higher up the value creation chain. They will start putting a lot more effort into crafting complex loss functions that truly reflect business goals, and understanding deeply how their models impact the digital ecosystems in which they are deployed (e.g. the users that consume the model's predictions and generate the model's training data) —problems that currently only the largest company can afford to consider.

Lifelong learning and modular subroutine reuse

If models get more complex and are built on top of richer algorithmic primitives, then this increased complexity will require higher reuse between tasks, rather than training a new model from scratch every time we have a new task or a new dataset. Indeed, a lot datasets would not contain enough information to develop a new complex model from scratch, and it will become necessary to leverage information coming from previously encountered datasets. Much like you don't learn English from scratch every time you open a new book—that would be impossible. Besides, training models from scratch on every new task is very inefficient due to the large overlap between the current tasks and previously encountered tasks.

Additionally, a remarkable observation that has been made repeatedly in recent years is that training a *same* model to do several loosely connected tasks at the same time results in a model that is *better at each task*. For instance, training a same neural machine translation model to cover both English-to-German translation and French-to-Italian translation will result in a model that is better at each language pair. Training an image classification model jointly with an image segmentation model, sharing the same convolutional base, results in a model that is better at both tasks. And so on. This is fairly intuitive: there is always *some* information overlap between these seemingly disconnected tasks, and the joint model has thus access to a greater amount of information about each individual task than a model trained on that specific task only.

What we currently do along the lines of model reuse across tasks is to leverage pre-trained weights for models that perform common functions, like visual feature extraction. You saw this in action in Chapter 5. In the future, I would expect a generalized version of this to be commonplace: we would not only leverage previously learned features (submodel weights), but also model architectures and training procedures. As models become more like programs, we would start reusing *program subroutines*, like the functions and classes found in human programming languages.

Think of the process of software development today: once an engineer solves a specific problem

(HTTP queries in Python, for instance), they will package it as an abstract and reusable library. Engineers that face a similar problem in the future can simply search for existing libraries, download one and use it in their own project. In a similar way, in the future, meta-learning systems will be able to assemble new programs by sifting through a global library of high-level reusable blocks. When the system would find itself developing similar program subroutines for several different tasks, if would come up with an "abstract", reusable version of the subroutine and would store it in the global library. Such a process would implement the capability for *abstraction*, a necessary component for achieving "extreme generalization": a subroutine that is found to be useful across different tasks and domains can be said to "abstract" some aspect of problem-solving. This definition of "abstraction" is similar to the notion of abstraction in software engineering. These subroutines could be either geometric (deep learning modules with pre-trained representations) or algorithmic (closer to the libraries that contemporary software engineers manipulate).

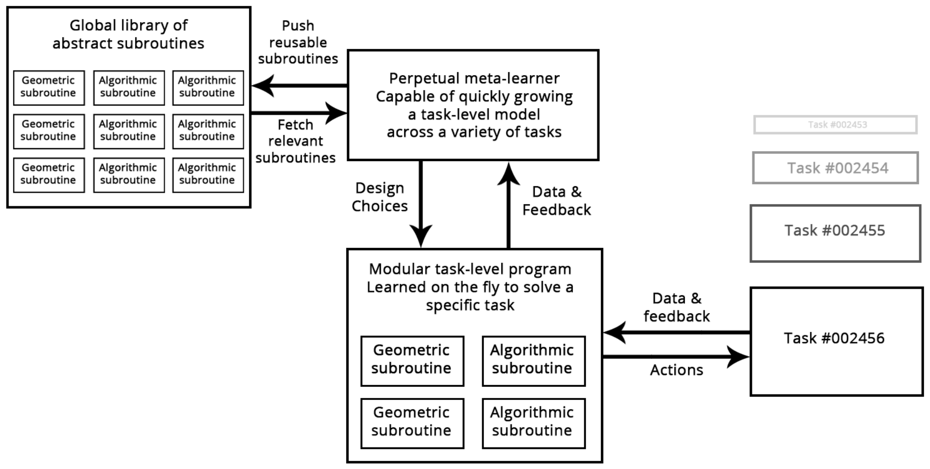

Global library of abstract subroutines
Geometric subroutine
Algorithmic subroutine
Push reusable subroutines
Fetch relevant subroutines
Perpetual meta-learner Capable of quickly growing a task-level model across a variety of tasks
Design Choices
Data & Feedback
Modular task-level program Learned on the fly to solve a specific task
Data & feedback
Actions
Task #002453
Task #002454
Task #002455
Task #002456

Figure: *A meta-learner capable of quickly developing task-specific models using reusable primitives (both algorithmic and geometric), thus achieving "extreme generalization".*

In summary: the long-term vision

In short, here is my long-term vision for machine learning:

- Models will be more like programs, and will have capabilities that go far beyond the continuous geometric transformations of the input data that we currently work with. These programs will arguably be much closer to the abstract mental models that humans maintain about their surroundings and themselves, and they will be capable of stronger generalization due to their rich algorithmic nature.
- In particular, models will blend *algorithmic modules* providing formal reasoning, search, and abstraction capabilities, with *geometric modules* providing informal intuition and pattern recognition capabilities. AlphaGo (a system that required a lot of manual software engineering and human-made design decisions) provides an early example of what such a blend between symbolic and geometric AI could look like.
- They will be *grown* automatically rather than handcrafted by human engineers, using

modular parts stored in a global library of reusable subroutines—a library evolved by learning high-performing models on thousands of previous tasks and datasets. As common problem-solving patterns are identified by the meta-learning system, they would be turned into a reusable subroutine—much like functions and classes in contemporary software engineering—and added to the global library. This achieves the capability for *abstraction*.
- This global library and associated model-growing system will be able to achieve some form of human-like "extreme generalization": given a new task, a new situation, the system would be able to assemble a new working model appropriate for the task using very little data, thanks to 1) rich program-like primitives that generalize well and 2) extensive experience with similar tasks. In the same way that humans can learn to play a complex new video game using very little play time because they have experience with many previous games, and because the models derived from this previous experience are abstract and program-like, rather than a basic mapping between stimuli and action.
- As such, this perpetually-learning model-growing system could be interpreted as an AGI—an Artificial General Intelligence. But don't expect any singularitarian robot apocalypse to ensue: that's a pure fantasy, coming from a long series of profound misunderstandings of both intelligence and technology. This critique, however, does not belong here.

@fchollet, May 2017