Active learning for program induction

Timothy Hanson & Justin Jung

May 10 2024

Abstract

This post goes into more detail on what is meant by active learning and how it relates to program induction. We discuss the use of a simulator for running a program ($\sim$ compressed model), and the use of a world model to encapsulate and operationalize accrued experience for efficient inference of programs. Desirable features of this world model are described, particularly features that enable *mechanistic planning and reasoning*, but the exact form is left as an open problem.

1 Introduction

A general motif in machine learning is (famously) function approximation, $y = f(x)$, where f is parameterized by θ,

$$y = f_\theta(x)$$

The form of $f_\theta()$ depends on the application & the data x, y. Two prominent examples are LLMs, where y is the next token, and x is a list of contextual tokens, and diffusion/flow models, where x is the current image (+ tokens), and y is an estimate of either the noise in that image, or the deltas to change/improve the image.

Deep learning overparameterizes f with very many θ's, which naturally means that the number of datapoints required to set or eliminate those weights is also large.[1] This works very well in domains where data is internet scale (including the two examples above). Internet scale data in turn requires internet scale compute to measure and change all those weights based on the data and the model, which limits accessibility to smaller companies and groups.

For some tasks like in-the-wild perception, it seems that there is no way around large data and compute; as Yann LeCunn points out, humans obtain $\sim 50x$ the data used to train an LLM through their optic nerves in four years of childhood[2].

We do know one mechanism that is highly data efficient: science, which takes minimal samples to yield models that closely and parsimoniously match the causal structure of the real world. Children, too, are amateur scientists in their own way, which arguably leads to sample efficiency. As mentioned in the previous post, Are Transformers all you need?, active learning is dependent on sample efficiency for learning and exploration: progress is governed by how much data is required to learn a new skill.

Scientific models usually can be described mathematically – or equivalently as *programs* which define and describe how elements in the model interact, how latent or unobserved quantities are set, and how the observed variables are generated. These programs are considered to mimic the causal, computational nature of the world, and often are as simple as possible.

Yet because these models are causal and computational, they are hard to fit; in comparison to the overparameterized functions $f_\theta()$, there is only one correct minimal solution, and a program either works or it doesn't. In comparison, the task of generating realistic image or video sequences tends to be tolerant of slight inaccuracies, such as an unrealistic smile in a generated image.

[1] Overparameterization also increases the absolute number of acceptable solutions, and sometimes the *fraction* of acceptable solutions to these function approximation problems.

[2] The human brain is obviously much more energy-efficient. This efficiency is primarily a difference in communication energy, as the per-float8 of a H100 and per-synapse energy are approximately the same, $\sim$1e6 times that of the Landauer limit.

Common approaches to this minimal version of modelling (also known as symbolic regression) frequently involves MCMC or sampling based algorithms [1] which work in spaces where the input-output mapping for from model $\rightarrow$ data is highly nonlinear and discontinuous[3].

Note that overparameterized models (e.g. large language models) *do* work well with discrete sequences, which are discontinuous and nonlinear, but LLMs are in nature autoregressive (predicting the next token based on other tokens) and so may not be able to model the causal dynamics of the world, as scientific models & programs require.

Humans work fine in these spaces, though. We understand causal and discontinuous relations in a sample-efficient manner. Can we replicate their behavior algorithmically?

[3] Again, compare this with overparameterized deep models, where all points are approximately saddles and so the space is navigable.

2 Learning with a simulator

Assume that we have a dataset $(\hat{x}, \hat{y})$ that is much too small for traditional function-approximation $y = f_\theta(x)$. Instead, assume we additionally have a simulator g which runs a program ϕ that transforms x into y:[4]

$$y = g(\phi, x)$$

As this simulator can be human-designed (e.g. python), the program ϕ can be human-interpretable, and (ideally) also concise.[5] We can sample as many times as we want from $g(\phi, x)$, which eliminates some sample-efficiency limitations – yet we still operate in a limited data regime in the sense that we don't have assume access to many examples of good programs ϕ that transform x, y – we have to learn to generate good code ϕ ourselves.[6]

The task of learning is to infer an optimal ϕ^* :

$$\phi^* = \arg\min_{\phi} L_{rec}(\hat{y} - g(\phi, \hat{x})) + L_{code}(\phi)$$

Where L_{rec} is a reconstruction loss and L_{code} measures program ϕ quality (e.g. length, syntactic, and type correctness).

One approach to this problem is to generate a dataset Φ by sampling n (x_i, y_i, ϕ_i) tuples through random enumeration of ϕ, x and transformation via the simulator $y = g(\phi, x)$

$$\Phi = \bigcup_{i=1}^{n} (x_i, y_i, \phi_i)$$

Then train a function h_θ to predict the program given (x, y). The parameters θ are optimized to minimize the loss over Φ:[7]

$$\sum_{i=1}^{n} L(\phi_i, h_\theta(x_i, y_i))$$

Then h can be used to approximate the optimal ϕ^*:

$$\phi^* \approx \tilde{\phi} = h_\theta(x, y)$$

Thus h is a *policy* for generating the program, and it requires that the initialization of Φ includes the the true data $(\hat{x}, \hat{y})$ in its domain and range. This is true in LLMs, which are trained on human-devised solutions to problems, and usually queried with problems within their domains. The approach has also proven successful in modelling numerical sequences[2].

Yet: knowing a good initialization is tantamount to being able to solve the problem! If you can reasonably enumerate ϕ_i and evaluate whether $\hat{y} \stackrel{?}{=} g(\phi_i, \hat{x})$, then you can simply search for ϕ^*. In the spaces we're interested in, ϕ^* is factorizable but infinite-dimensioned, and cannot be readily enumerated – there are just too many programs.[8]

[4] Because ϕ is a program - hence represented as a graph and not set of parameters - we adopt this notation instead of $g_\phi()$.

[5] The simulator additionally emits intermediate 'disambiguating' states z: $y, z = g(\phi, x)$, which will be discussed later.

[6] Although we assume access to a forward simulator, we are also interested in sample efficient methods that require less real environment data. In many settings, gathering real world data is expensive!

[7] If the loss is taken over the whole dataset, as indicated here, then this is gradient descent, not SGD!

[8] Local enumeration is and must be possible, as mentioned below in the reinforcement learning section.

In this document we define **active learning** as the exploratory process that recovers $\tilde{\phi}$ when the only knowledge is supplied by the specification $(\hat{x}, \hat{y})$ and the function $g(\phi, x)$. [9] We assume you don't know a good initialization to Φ - active learning should work when Φ is unstructured, e.g. the result of random enumeration.[10]

[9] Note the loss terms are optional.

[10] This form of 'active learning' is thus a subset of 'active inference' as proposed in [3].

2.1 Incremental & recursive generation

– Before approaching the modeling and action-generation problem, an interlude on factoring the problem –

Learning to produce a complete $\tilde{\phi}$ length j is very hard: the probability of getting all j choices correct becomes vanishingly small as j becomes large; even one small mistake can ruin a program. Humans almost never do this; we construct programs (and writ large, software) incrementally, by editing an existing program & observing intermediate output. [11]

$$\tilde{\phi}_{j+1} = h_\theta(x, y, \tilde{\phi}_j)$$

Alternately, h can explicitly create edits:

$$\Delta\tilde{\phi} = h_\theta(x, y, \phi) \tag{1}$$

$$\tilde{\phi} = \sum_0^j \Delta\tilde{\phi} \tag{2}$$

(This is very loose nomenclature – $\Delta\tilde{\phi}$ is discrete, and you can't really sum over it, only recursively apply the edits.) Edits are sampled from Φ, which is good as it approximately squares the size of the dataset.

Incremental learning and editing makes intuitive & natural sense. Experimentally, a transformer *does* approximate h_θ in eq.1 with high fidelity – but it does not generalize well. As mentioned in the previous post, this transformer is evincing a *policy* over programs ϕ or program edits $\Delta\phi$, and solutions to the original problem defined by $(\hat{x}, \hat{y})$ are by definition out of distribution and do not match the real data distribution of "good code/programs" that we care to replicate.

[11] This metaphorically is similar to how a diffusion model creates an image: images are generated recursively (by applying an incremental transition to the previous output); code can be written by adding a smaller piece to the previous work-in-progress. They are different in that in image generation the transition we learn has a parameterized structure in a continuous space, whereas incremental pieces of code are discrete and don't readily come from typical structured distributions.

A second approach to factoring $\tilde{\phi}$ is to make the process recursive – some actions generate intermediate states, which serve as sub-goals that can generate their own actions.[12]

[12] Most real-world action generation can be expressed in tree form, however our knowledge of this literature is lacking.

2.2 Reinforcement learning

One way of formalizing the problem of creating edits to a program is to treat it as a reinforcement learning problem. The RL objective is to select edits that maximize reward:

$$\Delta\phi^* = \arg\max_{a_t} \sum_{a_t} r_t \tag{3}$$

The action is the program edit $a_t = \Delta\phi_t$ and our reward r_t is sparse reward, i.e the negative of the total loss function $L = L_{rec} + L_{code}$. Selecting actions or edits is thereby done via search, which creates a supervised dataset to train h_θ. Search can be tree-search to select multiple actions in a sequence, which is how AlphaZero works – the upper-confidence bound (UCB, or in this case, UCT) on the Monte-Carlo tree search (MCTS) over actions is used as a supervised target for a policy network (that, in turn, hopefully generalizes beyond the MCTS policy).[13]

[13] This seems like a safe bet with recent architectures.

You certainly can search over possible actions or edits, but this operation is:

1. Linear in the program length – locations to edit = l,
2. Linear in the number of options – atoms + extant variables + new variables = o,
3. Geometric in the edit depth = d

Resulting in at least $O((lo)^d)$

That's not how humans program! We don't start with a random policy of editing code, and update the likelihood of selecting an action based on whether it worked or not. As mentioned above, the probability of obtaining a working program in that way is astronomically small. Moreover, we assume here that we *don't* have access to any structured data or sample trajectories: all information must come from interaction. Instead we **plan and reason with a world model** that encapsulates the dynamics & structure of the world.[14]

Planning can avoid some of the curse-of-dimensionality problems that plague pure RL approaches to open-ended domains like programming. Instead, human programmers iteratively select intermediate and end-states based on features of the data $\hat{x}, \hat{y}$, features of the current program $\tilde{\phi}$ and its intermediate data z, and accrued knowledge of past efforts.[15]

[14] MuZero & EfficientZero make good use of world models, but they rely on MCTS to propagate the utility of future states back to present action selection; they do not plan or reason *per se*. Their action spaces are also smaller.

[15] For example, z can be code-flow and data-flow. Programming is notorious for having vast unobserved state.

3 World models

Knowledge of past efforts can be represented either as a database of (*context,action,outcome*) tuples or more usefully as a world model:

$$p(x, y, \phi) \propto w_\theta(x, y, \phi)$$

Ideally a world model encapsulates full knowledge distilled from observations; from this full joint probability you can calculate the conditionals:

- The forward transform, $p(y|x, \phi)$, which is equivalent to running the simulator. It can also be incremental: $p(y'|x, y, \phi, \Delta\phi)$
- The reverse transform, $p(x|y, \phi)$. (This tends to be intractable due to computational irreducibility, but it can be locally approximated.)
- The 'policy' or posterior likelihood, $p(\phi|x, y)$. This too tends to be globally intractable. We would like this to also be incremental: $p(\Delta\phi'|x, y, \phi)$

The full world model also implicitly defines a topology – Jacobians of $\partial\phi/\partial x$, $\partial y/\partial x$, and $\partial\phi/\partial y$ can be estimated when conditioned on intermediate data z. Likewise, the idea that the data lies on (at least a locally) smooth manifold permits the computation of a similarity metric $M(y, \hat{y})$: similar ϕ should produce similar y from fixed x.[16] The similarity metric can be calculated via the KL divergence:

$$M(y, \hat{y}) = D_{KL}(p(\phi|y, x)||p(\phi|\hat{y}, x))$$

or the mutual information between the two distributions.

A similarity metric that reflects the topology created by the simulator seems essential: you need to direct actions toward a goal (the specification) in the *absence* of explicit reward signals. Hypothetically, a learned similarity metric derived directly from the world model should do this.

[16] If the data does not lie on some approximate manifold, then it's effectively chaotic and there is no sense doing anything other than enumeration of ϕ.

3.1 Planning and Reasoning

Planning refers to the iterative generation of sequences of actions to obtain a goal. It can be forward, in which case the forward transform $p(y|x, \phi)$ (or the simulator $g()$) is used to estimate intermediate states; it can also be backward (from an end goal), in which case the reverse transform $p(x|y, \phi)$ is used. Planning typically involves search or dynamic programming like A* which, like MCTS, is external to the world model.

Reasoning can be narrowly defined as the propagation of information from one modality (e.g. end state or goal) to another (the action). For example, you can mechanistically reason over a world model by asking "What must I do to generate this output / obtain this state?" Reasoning is effectively Bayesian inference over unknowns.

This indicates the use of an undirected graphical model as the world model – the problem is that deep learning tooling & implementations are

much more mature and scalable than Bayesian inference on graphs. Variational methods & variational free energy[17] over factor graphs are another approach; see the recent dissertation by Koudahl [4] for progress on this front.

Backprop is also a form of mechanistic reasoning: it propagates information from the output to the weighs, answering the questions "How must I change this weight to reduce the error?". If the model is differentiable or emits derivatives / Jacobians, then backprop can similarly be used for propagating information / inference. (More discussion on reasoning below.)

[17] As employed more broadly by active inference

4 Problem statement

Our task is to determine the structure(s) of $w_\theta(x, y, \phi)$ such that useful quantities (forward / reverse transform, policy, partial derivatives or finite-differences, and similarity metric) can be calculated.[18] Additionally, the model should support incremental and recursive $\Delta\phi$ generation.

[18] They do not all need to be the same function! But it could be nice from a parameter re-use & generalization standpoint.

4.1 Hypothetical approaches

4.1.1 Energy-based world model

Rather than treating the function w_θ as a probability density, we can think about it as an energy:

$$\ell = E_\theta(x, y, \phi)$$

If we can define a E such that programs ϕ compatible with x, y have lower energy $E_\theta(x, y, \phi)$, then we can infer good programs by minimizing E given x, y:

$$\tilde{\phi} = \arg\min_\phi E_\theta(x, y, \phi)$$

Alternately, if it's difficult to explicitly construct a similarity metric for the output space y (or for a latent space), we might instead define an energy which can output the similarity between two outputs $E_\theta(y, y')$. By measuring how close the output state y' is to the goal state $\hat{y}$ you can thereby assess progress of the program ϕ.

If the function w_θ is differentiable and information can be propagated backwards – via the Jacobian of the energy function, $\nabla_\phi[w_\theta(x, y, \phi)]$, MCMC sampling, or flow-based methods – then $\tilde{\phi}$ can be estimated by holding x, y constant while iteratively improving ϕ.

One problem with this is that w_θ is a causal & usually information-destroying process, and ϕ carries a lot of information, so naive backpropagation does not work well - or equivalently, MCMC takes a long time to sample dense regions.

4.1.2 Likelihood-approaches

Another option is likelihood-based sampling,

$$p(\phi|x, y) = \frac{p(y, x, \phi)}{p(x, y)} = \frac{p(y|x, \phi)p(x, \phi)}{p(x, y)} \sim p(y|g(x, \phi))p(x, \phi)$$

(Assuming that you can't easily evaluate the joint probability $p(x, y)$). By sequentially evaluating the likelihood of the data (perhaps with a noise estimate, e.g. $p(y|g(x, \phi)) = N(\mu, \sigma^2)$), and by observing features of the input data $p(x, \phi) = p(x|\phi)p(\phi)$ (e.g. with a type system) you can do coarse-to-fine program synthesis [ref]. Many symbolic regression approaches adopt this; though ϕ is discrete / not directly optimizable, the process is not necessarily inefficient.

4.1.3 Flow-based modelling

This seems presently like a decent approach:

- Model $p(\Delta\phi|x, y, \phi)$ via normalizing flow & train the model via supervised learning.
- Model the forward and reverse transforms via flow (or just use the simulator for the forward transform).
- Use backprop to estimate the Jacobians for pointwise inference (i.e. without propagating probabilities).
- Train the similarity metric as an independent function through contrastive learning.

All require aggressive OOD generalization for the active learning to work, which will depend on the parametrization of each of these networks. Single-pass function approximation (e.g. transformers) can be used in place of the flow networks above, of course.

5 Appendix & thoughts

5.1 Reasoning

Three forms of mechanistic reasoning:

1. Gradient-based: If you have a forward causal model of the world, by taking the partial derivative of the output relative to the input, you can assign 'credit' for effects, and then directly minimize over them. This could be useful for figuring out where to edit given intermediate z.
2. Memory-mapping: If you record past experience - via a database or a structured model - then you can reverse-associate observed effects with past causes. I suspect the brain does a fair bit of bidirectionalization of this type.
3. Prior beliefs: Irrespective of paired or linked data, if you have recorded past experiences (again via a database or model), then this forms a prior over expected values of any given variable.

The question that we're facing is how to reconcile, at the PyTorch level, these three different sources of information? Each by itself can provide (linked, structural) information on input-output, but they provide *redundant information*: internal activation can be (say) dependent on the feedforward causal inputs, or it can be dependent on priors (as is used in the hidden-layer DAE), or it can be dependent on a associative inverse model (e.g. UDRL [5]).

Ideally, you have a fully direction-agnostic Bayesian network which factorizes the joint distributions per above, and combines information in a principled statistical manner. If each path of information provides a mean and variance ($\sigma^2 = \infty$ if no information is known), then you can combine the estimates by weighting based on their precision $= 1/\sigma^2$ But ... we don't propagate precision through typical neural networks, and so don't have a good way of weighting? Can do MCMC sampling to estimate variance / standard deviation, but this is slow - at least linear in the model dimension. Variational methods do estimate mean and variance; we could train the network to do this (at what cost?), or add extra mixing parameters that are also learned (and predicted – they must be a function of the data, of course.) Can this be fully modularized, so that each module takes multiple inputs and ... provides estimates to those absent, via suitable transformations? Isn't this the dream of EBM?

In the case of a MLP, since the transformation is linear, it seems that a MAP estimate is exactly a combination of backprop (asymptotically equivalent to the W^{-1}) and the prior, with iteration to combine the two.[19] Assume also that you can iteratively approximate the inverse of the forward weight matrix ala RLS, and this is tolerant of overdetermined / underdetermined W.

[19] Can you make such a system loop-stable, such that you don't need the DAE?

In the case of a transformer layer = conditional-gather MLP, it should also be it's own inverse? With a denoising prior in there too? Perhaps the solution is to simply train each layer, forward and inverse, normally (using SGD); to use a flow-based method, it would be to train an ODE to transform between the distributions. Yet we have the problem that frequently information is *not* provided to the model.

The crux of the problem seems to be: You clearly can train individual models to represent the forward / reverse / policy / similarity functions using standard supervised learning. This seems terribly redundant, though; it does not re-use

intermediate representations and activations. Why can't you bidirectionalize our familiar and well-loved layers so that they look more Bayesian, and you can do flexible inference on inputs, outputs, and actions - each even incrementally.

References

[1] R. Guimerà, I. Reichardt, A. Aguilar-Mogas, F. A. Massucci, M. Miranda, J. Pallarès, and M. Sales-Pardo, "A Bayesian machine scientist to aid in the solution of challenging scientific problems," vol. 6, no. 5, p. eaav6971. https://www.science.org/doi/10.1126/sciadv.aav6971

[2] S. dAscoli, P.-A. Kamienny, G. Lample, and F. Charton. Deep Symbolic Regression for Recurrent Sequences. http://arxiv.org/abs/2201.04600

[3] K. J. Friston, M. Lin, C. D. Frith, G. Pezzulo, J. A. Hobson, and S. Ondobaka, "Active Inference, Curiosity and Insight," vol. 29, no. 10, pp. 2633–2683. https://direct.mit.edu/neco/article/29/10/2633-2683/8300

[4] M. T. Koudahl, "A Factor Graph Approach to Active Inference."

[5] J. Schmidhuber. Reinforcement Learning Upside Down: Don't Predict Rewards – Just Map Them to Actions. http://arxiv.org/abs/1912.02875