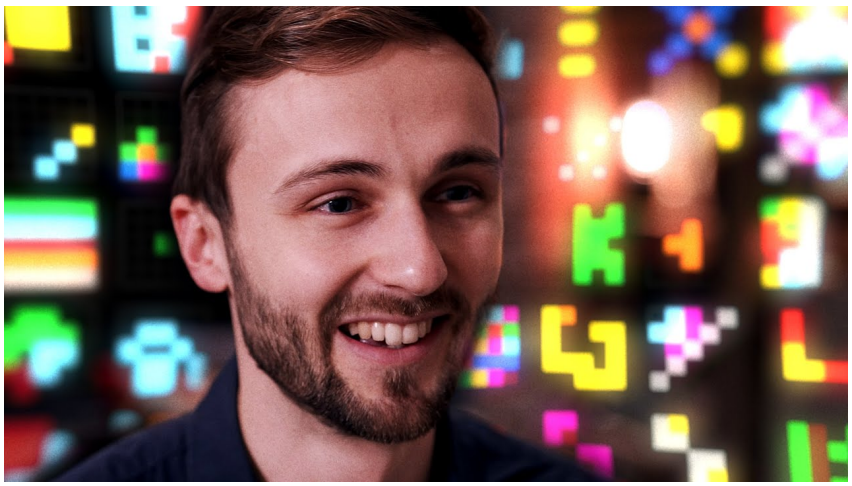
/mlst

Can Latent Program Networks Solve Abstract Reasoning? [Clement Bonnet]

March 23, 2026



Machine Learning Street Talk

Podcast Transcript

Revision: 989d31b6

Topic headings include retention indicators: • above average, • average, • below average (vs. similar YouTube videos)

Interested in licensing ReScript for your podcast? Get in touch with Tim Scarfe.

Contents

Overview 2

Introduction & LPN Overview 3

Why Neural Networks Fail at ARC 5

Induction vs Transduction Debate 7

LPN Architecture & VAE Training 10

Training Data & Scaling from Scratch 14

Compositionality & Multi-Thread Search 18

Search Methods & Architecture Scaling 22

Creativity, Scaling & Future Directions 25

References 29

Overview

Clement Bonnet presents Latent Program Networks (LPNs), a fundamentally different approach to the ARC-AGI benchmark that embeds programs into a continuous latent space rather than generating code or fine-tuning language models. The architecture uses a VAE-style encoder to map input-output pairs into distributional program representations, performs gradient-based search during both training and inference to refine these representations, and decodes solutions directly without an explicit program intermediary.

The conversation explores why ARC remains impervious to standard neural networks — Chollet designed it to resist memorization, with test tasks far enough from any training distribution that zero-shot generalization fails. Bonnet and Tim Scarfe dig into the induction vs. transduction distinction, the role of the kernel trick in blurring that boundary, and why parameter-space search (test-time training) may be less efficient than searching a compressed program manifold.

A key architectural insight: training the search procedure end-to-end forces the latent space to become smooth and searchable, similar to MAML-style meta-learning. The system achieves ~10% on ARC evaluation sets using only small transformers (40M parameters total) trained from scratch on Michael Hodel's re-ARC dataset — no pre-trained LLMs, no internet priors.

The discussion also probes the limits of continuous latent spaces for compositional reasoning. Bonnet acknowledges that single-thread search through a fixed latent space cannot achieve true program composition, but suggests multi-threaded search with external composition could bridge the gap. The episode closes with reflections on creativity, the efficiency gap between human hypothesis-testing (a handful of guesses) and LLM program sampling (millions of samples), and what scaling LPNs might reveal about the structure of program spaces.

Introduction & LPN Overview

Clement Bonnet 00:00:00

The ARC-AGI benchmark, Abstraction and Reasoning Corpus, is a program synthesis benchmark whose goal is to assess how AI systems can adapt to novelty at this time. But if we knew the kind of distribution of tasks that are in the test sets and you would, you know, fine tune an LLM on it, you would solve it. The the thing with transduction tasks is that there is no point in latent space that would correspond to your transduction task because by definition there's no, like, induction necessary there. You know, they're exponentially not creative, so, you would need more samples to make them creative. I solve no task. I have some intuition of what the task, you know, might look like in, like, you know, tenth of a second. I see the structures. I see regularities very, like, quickly. And I realized, oh, no. I mean, this is not as easy as this. The situation was happening in large language models that they have these very intricate, you know, very spurious, highly dimensional, you know, vector functions that can be recombined in some narrow way. I I'm a big believer in, like, trying to find, you know, small representations that could explain your your outputs. Now how do you how do you build these spaces? How do you search through them? So my real name is Clement.[1]

Tim Scarfe 00:01:16

Clement.

Clement Bonnet 00:01:17

Clement Bonnet. I'll go by Clem. Okay. So whatever, Clement, Clem, bonnet. Whatever is easier for you.

Tim Scarfe 00:01:25

Amazing. Clem, welcome to MLST. It's so nice to have you here.

Clement Bonnet 00:01:28

Thank you so much for having me.

[1] ARC-AGI Benchmark — paper

Tim Scarfe 00:01:29

So I've just been reading your paper, and Chollet told me yesterday that it's 1 of his favorite approaches to, you know, to the ARC challenge.

Clement Bonnet 00:01:38

Oh, is it?

Tim Scarfe 00:01:38

Can you tell the audience about the approach?

Clement Bonnet 00:01:41

The ARC-AGI benchmark, Abstraction and Reasoning Corpus, is a program synthesis benchmark whose goal is to assess how AI systems can adapt to novelty at this time. So it's a very interesting benchmark because, you know, pre trained LLMs perform actually perform very poorly on it because it's the tasks that are seen at test time are very different from from the training training sets. So the whole assumption we've had with this architecture that that I'm going to introduce is to have to built in search in architecture such that you're going to do test time test time search. And so the way we do it is that we embed programs into a latent space that we train such that it it is good to search inside.[2]

Sponsor 00:02:26

So Tufa Labs is a new AI research lab. I'm studying in Zurich. It is funded from past ventures involving AI as well. And so we are Swiss version of deep seek. So a small group of people, very, very motivated, very hardworking. And we try to do some AI research starting with LLM and Owen style models. What we're looking for now is chief scientist and also research engineers. You can check out positions at tufalabs.ai.

Tim Scarfe 00:02:53

There there have been many approaches to the art challenge.

Clement Bonnet 00:02:56

Mhmm.

[2]Latent Program Networks — paper

Tim Scarfe 00:02:57

And even going back to the days of Dreamcoder, know, there's Kevin Ellis. Yeah. And he, had this DSL. Well, maybe even before that, you know, we could just search a DSL, which is exponential time. Ellis had a neurally guided search where we train a neural network. And then these days everyone is embracing this kind of green blot approach where you just you just make an LLM generate programs. And yours is completely different to all of the other types of test time adaptation. You're embedding the programs in a latent space, and then you're searching that latent space, and you're not actually creating programs, you're generating the solutions directly.

Clement Bonnet 00:03:38

Yes, exactly. So really the assumption we had was that we want to do search at this time. And we want to do search in a program space, and to do that efficiently we try to embed, you know, combinatorially hard programs into a continuous latent space that then we can search using any search method that we want. And so by learning the manifold of programs into a single coherent latent space, We allow efficient test time adaptation. So there has been a lot of methods. The the top performing methods are using test time training, which is this parameter efficient fine tuning of a test a newly a new test input. You search in your parameter space parameters that would better perform on your test input, and so that's a very effective way to do this kind of recombination synthesis at this time. We we argue that it's a very inefficient way to do so because the well, the parameter space is huge, and it is not obvious that you can recombine low level primitives in a compositional way. So, I mean, yeah, it is to be seen whether these approaches can can solve ARC-AGI, but we we believe that by embedding programs into a more coherent and compressed representation, we can do easier, more

Why Neural Networks Fail at ARC

Clement Bonnet 00:05:03

efficient search.

Tim Scarfe 00:05:03

It might be worth meditating just for a second on why the Arc challenge is so impervious to neural networks. I mean, Chollet wanted it to be that way. Why is that?

Clement Bonnet 00:05:13

Chollet designed it to be robust to memorization. And way it is robust to memorization is because it is the so the input output task are so different from whatever has been seen in the training distribution. It is quite guaranteed that they don't exist anywhere on the Internet. It's a private test set that is, you know, kept completely hidden. And therefore, it is argued that the task similarity is is is not high enough for LLMs to kind of generalize your shots.

Tim Scarfe 00:05:43

Let me just push on it just a tiny bit though. So many of the sentences that we've just spoken in this conversation don't exist on the internet either. So what's the difference between that and an ARC program?

Clement Bonnet 00:05:52

So I believe our discussion can be, you know, embedding into a rather small latent space that the LLMs have learned to actually emulate and kind of, you know, compose in a way. The ARC the ARC AGI task are truly novel, and so they're only based on the, you know, the core human knowledge priors, but they compose these priors in arbitrary ways that I believe are not anywhere on the Internet and that the, while the pre trained logic models cannot make CentML of, otherwise you would get 0 shot generalization, which This we

Tim Scarfe 00:06:29

is an interesting thing just to linger on those. So are you saying that the reason why neural networks can't learn ARC is simply because there isn't loads of ARC like data on the Internet? It's not because I think at some point people might have thought there's something intrinsically about the ARC, you know, because it's a discrete program, you know, distribution or whatever that, you know, neural networks can't learn it, but you're saying they can learn it, it's just because there's no data out there.

Clement Bonnet 00:06:55

So if we knew the kind of distribution of tasks that are in the test sets and you would, you know, fine tune an LLM on it, If you're touching any transactive, any neural network architecture on the distribution that includes the test set data, you would solve it. It's it's machine learning training. But the the reason why we can't we can't solve this test task is because the well, this the test distribution is is quite far away from from the training distribution. So it's it's not able to do, you know, extreme generalization and trying to come up with this really novel way of recombining knowledge to to perform the task.

Tim Scarfe 00:07:35

MLST is sponsored by CentML, which is the compute platform specifically optimized for AI workloads. They support all of the latest open source language models out of the box, like Lama, for example. You can pay on consumption essentially, or you can have a model which is always working or it can be freeze dried when you're not using it. All of the models that they deploy support the, OpenAI API specification out of the box, which means it's just a 1 line change in your application to switch over to CentML and start saving money and make your application go faster. This gets to the next entry, but then so so I guess the reason why perhaps a few years ago if you asked me was it possible in principle at all to learn ARC, I might have said no because, you know, many people, even even Kevin Ellis, for example, he loves the idea of generating programs, and that's what Greenblatt did. You know, like, we feel intuitively that we should generate a program because then when it runs, you know, on on a computer, programs can do this kind of compositionality and like various forms of generalization and so on. So intuitively we think that if we generate the actual answer without going via a program intermediary, then it won't be as good. But is that not[3]

Induction vs Transduction Debate ●

Tim Scarfe 00:08:43

the case?

[3]Kevin Ellis - Program Synthesis — person

Clement Bonnet 00:08:43

So I'm assuming you're referring to doing transduction versus induction. So going through a, you know, a compressed program representation, which would be, you know, induction based versus directly trying to predict the answer.[4]

Tim Scarfe 00:08:56

Yeah, maybe maybe we should we should define induction and transduction because I think even before you get to transduction, you could train a neural network just to generate the solution directly. Yes. And without even using transduction.

Clement Bonnet 00:09:07

That'd using transduction though? The if you predict the output data directly from like, reading out the inputs.

Tim Scarfe 00:09:13

To me, the define transduction as generating a prediction directly from the data rather than being a statement about using the solution space. For example, you know, Jonas Hobbitur, his stuff is transduction because, you know, given some input examples, he does some retrieval and then he fine tunes the model and then he does a prediction. So he's like he's building a new model for every single data that is received. But we could build a massive neural network which directly predicts the solution space, which is still an inductive model.

Clement Bonnet 00:09:44

Yes, so I mean that's where the fine line between induction and transaction I think comes in. Also you have the kernel trick which turns any induction method into a transaction method. And I believe so I mean, I I don't know if the distinction between induction and transcription actually actually matters. I think what matters to me is is how much you is whether you compress report stations. So obviously, when you have an inductive method that comes up with a you know, whether it's a Python program or maybe, like, some kind of latent program, a small repartations a repartation that is supposed to emulate your outputs, I would call this an induction method because there is this kind of repartation compression bottleneck. And if you're not trying to do that, if you're trying to do some kind of shallow recombination of, you know, just transitive test time training, then I would, you know, put this in the group of transitive methods. And, yeah, for me, it's all about the the compression because compres-

[4] Induction vs Transduction in Abstract Reasoning — paper

sion allows you for for more efficient search. In architecture, we we build this very compressed representation of programs in which we can we can effectively search. And I believe when you when you have some methods about that generate Python programs, I believe these are very similar methods of generating a a very compressed LBPs program in Python that can can generate an output.

Tim Scarfe 00:11:00

I was gonna ask you why a kernel method is not induction.

Clement Bonnet 00:11:05

Call of linear regression is to come up with this linear function that, you know, best explains your your data. And so you can and then once you have this function, you can then use it for inference using a new a new, you know, input output pair a new input. When you solve this, you know, linear regression of the of the shape y equal a x, you have, you know, a solution of the form x t x minus 1, x x t y or something. And and so you can actually write the outputs as a pure function of the training data and your new input, you know, by just multiplying matrices. So I think I think that's, like, 1 repetition of the Kalman trick. It's just although you have this, like, very low rotation function, which is here a linear function, you can actually rewrite the whole thing as like 1 computation.

Tim Scarfe 00:11:50

You could, but isn't the kernel matrix is like the training data, it's the inner product of the data basically, giving you like this positive semi definite matrix. Is that not the inductive function?

Clement Bonnet 00:12:03

Yes, so essentially it's like saying y equal effects, and then you know, you write it out, like kind of unroll whatever computational graph you have, then the whole thing is like a so I think that that's where the equivalence between transduction and induction comes in to me, at least how I see it. So that's why I don't I don't think it's a very useful distinction here.

Tim Scarfe 00:12:23

I don't know. I I I really like the distinction. Mean, to me, it's about creating a model from the data rather than reusing an existing model that I'd So for example, we could make your model transductive if we fine tuned the encoder based on some augmentations of the data. But I would argue your system is not inductive now because we're doing some kind of gradient optimization through the model, but we're not changing the model, therefore it's inductive.

Clement Bonnet 00:12:50

Yeah, we're changing the model, that's the whole thing. Although I do think that so our, you know, latent program network, LPN search, is actually, I think, fits in a spectrum of test time training methods. So if if you see the latent space as, you know, an input conditioned parameter space in a way, we are turning this space. We are searching through this space using 0 order or first order optimization methods to find a better explanation for your data. And it's quite similar to searching through your parameter space, doing parameter fine tuning at this time. So I do I do think they fit in the same spectrum. I'm curious to explore methods where this adaptation at this time would be as efficient as possible if, you know, we want to create systems that, you know, adapt online very efficiently. And so I really like the same training. I I do think that they're not it's probably a very suboptimal method searching in this vast parameter

LPN Architecture & VAE Training

Clement Bonnet 00:13:48

space.

Tim Scarfe 00:13:48

So can you talk me through the architecture?

Clement Bonnet 00:13:50

So as I said, we we want to embed programs into a latent space. So to do so, we have first an encoder that takes input output pairs independently and then embeds them into a program similar to a VAE architecture, actually, a version autoencoder. But if you think about it, an input output actually can be explained by an infinity of programs, right, of varying description length or complexity. So we would, you know, we would want to find a minimum description length program. But so because you can explain input output pair with an infinimum

of programs, we use a functional framework, so we encode these programs into a distribution actually of written programs. So we embed this input output pairs into distribution of programs, and then we we have this, like, search component in the middle, which I'm going to explain right after. And then once which is all about refining this latent to be a bit better. And then once you have a better latent, you try decode it with your decoder. And so your decoder is actually going to take a new a new input. It's going to take this latent program and then generate an output for it. So it's literally applying a program. So so I'm saying programs here, but I but I mean actually latent vectors. We all in the geometric space here. And and so did this so the the true novelty, I think, of this architecture is actually this middle stage here. And how it works is that the we expect the encoder to actually output a first guess of what the program looks like, you know, bit as a human intuition when solving an an org task of, oh, I do believe there's something about shapes here, and, like, we should move them around. So that, you know, it's like a first guess. And then we we use optimization methods to refine this latent, to find actually a different point in your latent space, in your latent program space, that would better explain your data. So you have this kind of inner loop with that uses the decoder and the input output pair, so your specification, to refine, find a better latency in your latent space that would actually, for each of the input output pairs, would generate the right output. And once you found this, you know, a further away point in latent space that better explains these input output pairs, you're rather confident that you can apply it to your new test inputs. So I think that's where the the induction part here comes in because we're really trying to find this, you know, small robot stations, small latent program that explains your whole task.

Tim Scarfe 00:16:17

So ARC, let's say, typically comes with about 3 specific and when we say specification, we just mean like that you get a few input example pairs. And you can place all of them through the encoder and the encoder will give you 3 different points in the latent space. You average those together and then you perform some gradient steps to improve the latent to a point where you get a good solution for all of the 3 examples, something like that.

Clement Bonnet 00:16:45

Yes, yes, it's exactly that. So we have a so we must find a way to recombine these different latent distributions for these different input output pairs. Ideally, each input output pair should actually through the encoder should generate roughly similar latent distribution because they actually correspond to the same inherent task. And so the vanilla way of doing so is to actually compute a mean and latent space and then use that as a starting point for your search. I do believe there are better ways to do so. We could see the whole aggregation as a, you know, a mixture of distributions and probably, like, I don't know, maybe sample with a small temperature or something. I do believe there are better ways to aggregate these kind of distributions. You know, a first proof of concept we've done was this mean aggregation, which ends up working pretty well. And so the the nice thing about this architecture is that it's trained end to end using a virtual VA loss, so the the elbow loss. So this means the you know, it is decomposed into 2 terms, the reconstruction loss. So, you know, given the latent program that is refined, try to reproduce your input output pair, given all the other input output pairs in your specification. So you have a reconstruction loss, and then you have a prior loss. Make sure your latent space looks roughly Gaussian. So this is the the classic prior loss from from the VAE, and it actually encourages the the latent space to be very structured. We we did try without the VAE framework, so the pure to encoding framework. So no prior on what the latent space should look like. And we end up with very unstructured spiky spaces where most programs tend to be as far away as well, programs tend to be as far away as possible from each other, making search impossible and very and making the whole latent space rather useless. So I think the version aspects of it is very important, trying to keep a Gaussian compressed representation of your program space.[5]

Tim Scarfe 00:18:45

Yeah, we should double click on that a little bit. So we want to have a well formed space that doesn't degenerate and doesn't memorize solutions, and I think you also used the KL divergence. Just reading from your paper a little bit here. So you said that you want to prevent encoding the output directly in the latent space. So you encode an input output pair to the latent space, but train this representation to decode the output of a different input output pair to prevent memorization? So that was 1 of the tricks that you used.

Clement Bonnet 00:19:12

Yes, actually it's well it's not really a trick, it's a so with most of these latent space ideas, usually when they don't perform them on program synthesis benchmarks

[5]Variational Autoencoders — paper

like ARC, there's usually this problem of you want to learn a compressed rotation of whatever your input output space is and then try to decode it. But the obvious thing is that you you may actually end up just compressing your outputs in your space. You would kind of leak your output in that space, and so it's easier to reconstruct. It it would be a shortcut that CentML training would would lead, and it would be very useless because you you wouldn't learn anything inputs that connects input to the output. You would just learn to map the outputs to latent space. And so the way we deal with this is just we consider during training, use a very similar setup as we would have during tests, which is we have access to CentML input output pairs, and we want to predict an CentML plus 1 output. So actually during training, if we have if we assume CentML input output pairs for each input output pair, we are going to embed into the latent space the CentML minus 1 other pairs to predict that pair and kind of do that for all the pairs in parallel.

Tim Scarfe 00:20:26

Very cool. Now CeCeCeCentMLtMLtMLtMLow the other great trick you used is you actually do, like, gradient steps, so search steps during the training process because it actually makes the the final kind of latent space more amenable to search at inference time. Was that something that you thought to do intuitively or you did it to help it because it wasn't working?

Clement Bonnet 00:20:48

Yeah, so we've always wanted to train using search, and the reason why we want that is that if you think about it, during training your encoder will even at the end of training, your encoder is not perfect. The whole assumption is that it provides a, you know, a best guess of what the task is. And so if you're if you're not allowed to kind of refine this guess, well, decoder can be as good as what the guess from the encoder is. So you end up with a latent space well, you would end up with a latent space that doesn't cannot really encode many programs. It would just encode kind of the, you know, regression to the encoder's best guess distribution. So what we end up doing is that we actually activate this search method during training. It is it it brings significant overheads at training time, so we actually propose some kind of pre training without it and then kind of fine tuning with it, but it it doesn't matter. The the whole point is that you by switching it on during training using some kind of random local search or gradient ascent first order optimization search, we train the latent space to be good at being searched. So the best guess ends up being actually quite bad, but very close to a very good guess. It's the whole point behind these meta learning methods like MAML and so on.

Tim Scarfe 00:22:08

Qualitatively, tell me about the latent space. So you know you've looked at it, do you find that different directions correspond to different types of ARC problems or whatever, what do you see?

Clement Bonnet 00:22:19

So we haven't analyzed the latent space that much by comparing different ARC tasks yet, I think it's a very interesting thing to do. We I think in the paper, we we do, you know, bring a bring up a plot of kind of a TSNE partition of the different programs that are learned. And we so during training, we definitely see this kind of blob of, like, all the so all the input output pairs that are not really well understood, neither by the decoder or or by the the the encoder, end up being in this kind of bag of, you know, unlearn programs. And programs that start being learned start to, like, you know, cluster and and occupy the space. So we we we kind of scaled this architecture for ARC, you know, very very recently. We we got some nonzero yet non Sota results, And I think it's interesting because we although we don't get very, very high performance, we do get it without any priors, so without using any pretrained LLMs, without looking at the variation set or whatever by just training on, like, the on the 400 train test distributions. So getting something like 10% on the on the evaluation sets, which is in itself actually a a harder distribution of tasks, was quite impressive, I think, from from the the first initial scaling curves we've we've had.

Training Data & Scaling from Scratch

Tim Scarfe 00:23:39

Can you explain that bit in bit more detail? So you've got an encoder and a decoder and their transformer models, and you've got the is it the re arc tasks from, you know, Michael Huddle? And what you mean, what kind of transformers were they, and and what kind of training did you do?

Clement Bonnet 00:23:53

Yeah. So the the LPN architecture, you know, assumes this encoder and decoder modules. And to to train this architecture in ARC, we we reused vanilla transformers from scratch to record to encode the the input output grids. So really, you could use any generative model, any sequence model. We decide to use self attention transformers that kind of flatten the input output grids into sequences of, you know, 900 values because these are 30 by 30 grids. We, you know, we

use some kind of roughly smart 2 d positional encoding to, you know, make CentML of spatial structure, but that's that's roughly it. And then we train rather small transformers. I think they're about 20,000,000 parameter each, roughly 40,000,000 in total, on encoding and then generating auto reciduous input output pairs. We didn't tune the architecture that much. We had to do something, you know, around the positional encoding to to make CentML, but but otherwise, it's a very vanilla transformer architecture. We really focused on trying to get a good latent space of programs and how to, you know, efficiently search it during training and at this time. And we, yeah, we kind of we experimented with architecture, the transformer architectures at first, and we're like, okay. We're good to go. Let's just focus on what we I think what we think is is interesting.

Tim Scarfe 00:25:20

This blows my mind because most people that use transformers fine tune them. They take ones that have been trained on internet data and they fine tune them. I guess, like, the intuition for that is we're told it's very difficult to train transformers, and they only start doing good things when you train them with loads of data for a very long time. So you started from scratch. I think you said you didn't even train it to convergence just with, like, Michael Hoddle's rearch dataset, so hardly anything, and it still worked quite well.

Clement Bonnet 00:25:47

Yes. So the so the whole point during for our training was to to learn to embed these 400 trained tasks in your latent space. And if we if we do so, you know, in a rather structured way, then we hope that by interpolating this latent space, you know, during search at this time, for instance, giving a new input output pair, giving you tasks, sorry, then you would find programs that are not seen during training but at you know, represent kind of interpolation, probably superposition of programs seen during training. Finding a transformer on single input output pairs from a a single program is a is a very bad idea. There is no there's not enough data to for the transformer to make CentML of, you know, position relative positions and so on. So we use Michael Huddle's very useful library of real transformations. So the the the data Michael Huddle created is just a distribution of the for each of the 400 tasks, it's a distribution of input output pairs following that program. So we the program is constant, so there's not not a lot more information that we extract. Really, the program is constant. It's just the input distribution is now, you know, a a rather broad distribution instead of being a a single in in single input. And so this allows, you know, training transformers to make CentML of 2 d grids in a way that, you know, they would not when they're initialized from scratch. We do show some kind of, like, some training curves on the on the on ReArc. We actually didn't have enough compute to train

to convergence, but we showed, you know, kind of some increase of accuracy on the train sets. We do believe there are some bottlenecks in the architecture that make learning very slow not that wouldn't converge to a 100%. So, yeah, we we figure we're figuring this out.

Tim Scarfe 00:27:34

And in this rearch dataset, so 400 tasks blown up with this dataset generation that Michael does, how many, like, data you know, like, many program examples do you have?

Clement Bonnet 00:27:44

It's in the order of 100,000,000 data points.

Tim Scarfe 00:27:47

Oh, it's a lot,

Clement Bonnet 00:27:48

It's quite a lot, yes. We do sample a lot, these programs. I think transformers are really bad to make CentML of random sequences of pixels, like when they train from scratch, of course. Yeah. So that's why we really have to do this intense training of, you know, making CentML of this input output grids, which is something, you know, a bit silly, but we we believe RKGI is actually, you know, as said, developer aware bench generalization benchmark. And so we really wanted to have no priors from anyone on the Internet, so to not use any pretrained LLMs, and really to train architecture on the, you know, very little data that, you know, is that matters to the task.[6]

Tim Scarfe 00:28:27

Isn't the the ReArc dataset generator a prior, though? Like a a pretty big I mean, to me, that would invalidate the developer aware generalization.

Clement Bonnet 00:28:36

The priors contained in ReArc are all about, you know, when you actually have an input output there with 2 objects, while you could have the same with like 3 objects, you know. So you have the priors of counting, for instance, I do believe in the task, you also have this other priors of objectiveness and so on. For us, it's it's very much the the priors used contained in ReOCC are very much to teach

[6] re-ARC Framework — tool

transformers how to make CentML of 2 d grids because the all the 2 d grid space is huge. It's, you know, 10 power 900 for each grid. And obviously, most of these, you know, 10 power 900 grids don't make any CentML. They kind of white nose. So we would use this distribution to kind of feed in the human some of the human priors into architecture, which I believe is in line with the developer word generalization assumption.

Tim Scarfe 00:29:22

Yeah, we can we can debate that. How did it generalize to different grid sizes? I mean, course, you're you're using transformers, so so you you just you turn it into a sequence, right? But but did you notice any weird like overfitting on sequence size?

Clement Bonnet 00:29:36

No. The way we deal with the the shapes is very naive. We actually predict the shape first. So let's say you have a 10 by 10, you know, 10 by 11 grids, we actually predict, you know, it's going to be 10 rows, 11 columns, and then we have the 900 tokens corresponding to the padded 30 by 30 grids, most of which is actually padded and ignored by transformer. But we end up with actually sequences of 9 0 2, you know, tokens values per grit. Yeah. That's how we, you know, predict shapes. Yeah. I think a lot of things to say about how to optimize this. We don't think it's an optimal way to do so, but it's, you know, it's it's probably enough for for ARC or enough given that you have a lot of data to train on.

Tim Scarfe 00:30:21

Yeah. I suppose this might be an example of why generating an explicit Python program might might be better because it could generalize to different, you know, grid sizes.

Clement Bonnet 00:30:29

So Python programs, grounded programs, you know, executive by end interpreter would would, you know, would make CentML of all these priors that, you know, are not contained into a random initialized transformer. So, yeah, I mean, I do believe, you know, pro like, real programs are also a very smart way to to do induction to compress your rotations. 1 of the problems, of course, with program search is the combinatorial expansion, combinatorial explosion of of the, you know, the search through your program space. And so we really try to try to compress that by learning the manifold of programs that, you know, we hope would be easier to search. But, obviously, it comes with some caveats and, you know, potentially, you you can't represent all programs into this latent space. So, yeah, I think there's interesting ways to maybe try and think about, you know, combining both ideas. But there are pros and cons for, like, program search and latent program search,

Compositionality & Multi-Thread Search

Clement Bonnet 00:31:28

I believe.

Tim Scarfe 00:31:28

Yeah, I'm just thinking back to when I first interviewed Trolley years ago, know, always said there are type 1 and type 2 problems and type 2 problems are not interpretive. But we're now we're taking it as granted that this is actually interpretive after all. So we have this latent space and it's got all of these modes in and we do this test time computation and the more test time computation we do the better the results. And at the moment if I understand you're using like a first order kind of like gradient search method. So first of all, talk me through that. So you do more search at inference time, you get better results.

Clement Bonnet 00:32:00

Yes, but just before that, I do believe type 2 problems will not work with our method. So I do believe 1 of the big limitations of learning a continuous latent space is actually you can't fit arbitrary complex combinatorial problems sorry, programs in it. I think you can do some things like superposition of programs, but I don't think you can do composition of programs, which is obviously what we're trying to look into as a as a as a follow-up work. We so, yeah, I I I do believe that to you would not be able to get composition by searching through this latent

space. I think you can get some composition, like, you know, maybe just a couple steps of composition. But at the end of the day, it's a it's a fixed continuous latent space, so it's unclear whether you can really compose objects with it. Right. Now going back to the search

Tim Scarfe 00:32:53

Well, just before we get there, this is the is the mode this is like you've you've really touched on an interesting point. So this gets to the core of it. Right? So people like Gary Marcus and what Charley was was intuiting is is the lack of compositionality. So so you're you're saying, okay, there is no compositionality in in the latent space, but we could we could actually come up with some kind of a a compositional inference time method that gives us compositionality. So if you think about it, we could have multiple search threads through the latent space, and we could have some system 2 program that like composes the things together, and we could get the compositionality but still use your method. Is that kind of what you're saying?[7]

Clement Bonnet 00:33:30

I think that's a promising approach, a promising way to think about it. I I do believe that single threads searching this single latent space would not would never bring you a composition. But something like having, you know, multiple threads, as you said, or some kind of, like, multiple things happening at the same time being kind of synthesized thereafter is is could be a way to get composition. So I think what what what we wanted to do is kind of a POC of how to try fit as many programs as as we can in this phase and space. I think now it opens up, you know, lots of possibilities to extend this further.

Tim Scarfe 00:34:07

Then we have the matter of if we were generating, like, symbolic programs, it feels like we could compose them together, like, for example, we could we could learn some iterative, like, you know, steps of functions that just get composed together. Whereas in this case, we have solution predictions directly. So let's say right now I've got 3 solutions that I found that represent some kind of, you know, 3 different types of transformation that need to be composed together. Now we need to have a function that can somehow take those 3 solutions and combine them together.

[7] Critique of Deep Learning — paper

Clement Bonnet 00:34:43

Yeah, so I do think you would need some kind of enrolled computational graph. So I don't think you can easily look at these 3 solutions and kind of synthesize them in 1 shot. Intuitively, would it would be equivalent to actually having only 1, like, better thread or something. I I so if you want to, you know, try to explore different regions of your space and then try to combine them, maybe you would need to kind of do multiple forwards or something. So maybe, like, you know, 1 thread gives you, you know, some kind of primitive program, and then you apply the second thread on the output of the first primitive and so on. This way you would get, like, a recursive unrolled, like, convolutional graph. I mean, yeah, it's it's I think it's up to it's it's a research question how to to think about these things.

Tim Scarfe 00:35:35

Yeah. Or how to train that would be difficult. Even maybe like you could have an intermediate stage where the decoder could take in multiple inputs, and it could it could learn to compose them together to a fixed depth because, you know, like language, it's not actually that compositional. Right? You know, like we only use a depth, some languages don't use depth, but you know, English, we have a depth of up to like 3 or 4 or 5 or something, so you might find that most of the, you know, the colloquial ARC problems that we'd ever want to solve would have a fixed depth, and we could somehow encode that in a neural network.

Clement Bonnet 00:36:03

Yes, and actually the ARC solutions using brute force program search, you know, have a depth varying from 1 to, you know, usually something like 12, 15, and sometimes up to, like, 50. But, like, from those tasks, you actually you can solve them using a rather small length. So, yeah, very

Tim Scarfe 00:36:22

Yeah. And we're pragmatic people, you know, like, I love I love the theoretical arguments about, you know, compositionality and and, you know, symbolic AI and so on. But, you know, in in the real world, we don't actually need to have this infinite, you know, composition.

Clement Bonnet 00:36:37

Going back to the search methods, so our assumption is, you know, that there exists a different program, latent program in your latent space that is better than, you know, your first guess. So the question is how do you find that program? And so it's just a matter of it's a question of search now, and we we actually tried 2 things. We tried some kind of 0 order random search, you know, local search around your your first program, And then we actually realized that the the latent space itself that, you know, was learned during training ended up being very smooth. And so when I say smooth, it's smooth with respect to the decoder likelihood of decoding the right input outputs. So you want to kind of maximize the likelihood of decoding your right output. So this latest first ended up being very smooth, which allowed us to use first order methods. But I I do believe that, you know, potentially with different tasks or if we train for longer, you could have some some more discrepancy and and, you know, kind of local optima in this space, which would, you know, maybe bring to the table methods like evolutionary strategies, you know, 0 other methods that or maybe a combination of the 2, but for for our use case, it worked very well to use gradient descent. I don't think it's the best method, but yeah.

Tim Scarfe 00:37:53

Okay. So so at the moment, it's quite smooth, it's quite amenable to simple, like, you know, gradient ascent methods or whatever. That's all cool. What about its resilience to I don't know whether you tried putting noisy inputs in and or another thing is like you could almost under train the architecture but you know do more more searching like is is there a relationship that that you've explored there?

Clement Bonnet 00:38:16

So we haven't tried, you know, putting noise and shooting ourselves in the foot. Maybe we should. But we we have tried scanning architecture size, So obviously, if your transformers are, you know, too small, well, for sure, they can't represent your they can't decode your output very well. So, yeah, there is a there's kind of a trade off. Well, at least it goes you know, architecture size and, like, latent space size go, like, hand in hand. So, yeah, we we we did see that when the architecture size is too small, well, you can't even make CentML of the sequences very well, and, hence, you can't, like, decode your program very well no matter how big your search base is.

Tim Scarfe 00:38:51

Mhmm.

Clement Bonnet 00:38:51

So we need to actually when we train this architecture, it's there there's not enough, like, tuning to do, but we need to make sure that the decoder, you know, decoder's capacity is is high enough such that the they will and your latent space is big enough such that there exists a point in latent space that, you know, correspond to the right program. And given that point, you are you have enough capacity to execute that program, which is not the case when you use, you know, Python programs, you assure your interpreter will execute it well. You just need to, like, find it. So, yeah, you have this, like, second constraint here of, like, making sure the decoder can learn that program. And we actually have preliminary results showing they they were not in the paper actually, but showing that the it's it's not obvious that the decoder can actually learn, like, all the tasks. There's a there's also a paper mentioning this issue. So and it, you know, it it boils down to what priors you feed into your architecture in terms of, like, 2 d spatial representations and so on. So so, yeah, it's so try to have how much how much time you wanna spend on, like, optimizing, you know, your architecture to solve a specific benchmark versus how much you want to, like, improve the, you know, the method itself, I. CentML, like, the searching latent programs.

Search Methods & Architecture Scaling

Tim Scarfe 00:40:08

Very cool. So Ellis and Wendig, they just released this paper to, you know, sort of like comparing induction and transduction, and they had this ensemble approach where they kind of did both at the same time. They've got this beautiful Venn diagram that shows that, you know, sometimes induction with Python programs works really well, sometimes solution space works really well. Have have you kind of, like, done analysis on what types of ARC, you know, tasks work well and don't work well on your system?

Clement Bonnet 00:40:36

Yes. So we don't have very strong analysis on this, but from what we've seen, most of the, you know, bag of programs that are not learned often correspond to transductive task, transaction task. So I think it's very much aligned with their paper. You know you know, it's it was a it was a great paper highlighting this. We

actually, when when it came out, you know, maybe a week a week or so before the the odd price deadline, we we thought, oh, it's actually what's happening. We, you know, we struggle learning this transduction task. And, yeah, I think there are obvious ways to kind of use this assembling technique also for architecture or for other architectures. The well, the the thing with task is that there is no point in latent space that would correspond to your task because by definition, there's no, like, induction necessary there. So it's all in the decoder, and I think there are ways to, like, help the decoder with that, essentially. We just try to, you know, remain as benchmark agnostic as possible, trying to not overfit to ARC. Mhmm. But I I think there are obvious ways to improve.

Tim Scarfe 00:41:44

Is there a story that you're kind of like, you're not a symbolist?

Clement Bonnet 00:41:48

I've always been a connectionist, and I'm I'm kind of new to, like, trying to improve connectionist architectures to tackle this, like, maybe 1%, 10% task that people are really into these days of, like, reasoning, search. I do believe in a future where, you know, maybe probably most of your task can be tackled by, you know, kind of 1 shot deep learning based architectures like LLMs, but actually some of the, you know, hard, last mile, hard, like, 1% task need something a bit different. So, yeah, I'm a strong believer in, like, deep learning and program synthesis kind of being merged to achieve that, like, last percent of task.

Tim Scarfe 00:42:26

So you're not amenable to this school of thought that these things are just smarter than us, we just let it do the thing?

Clement Bonnet 00:42:35

I think we can't comprehend, you know, arbitrarily combinatorial sequences of symbols. And so that's where, you know, connectionism, like, comes in as very useful because we we try to learn, you know, manifolds of of complexities. I think it can bring you, you know, very far and can only bring you so far, and that's what I mean by I think that deep learning architectures can solve, you know, something like 90%, 99% of the task, but there will all be some some tasks that are, you know, probably, like, discrete inherently or tasks that require, like, long horizon planning and reasoning, things that we're trying to tackle with NLMs and other deep learning architectures nowadays, very recently, actually. And I do believe that other architectures could do this much more efficiently. I I think it's connected to creativity as well. Mhmm. I believe deep learning models are not highly creative. I I will not say they're not creative because I well, as you can see with going back to to ARC again, the like, solutions like Ryan Greenblatt's solution of sampling a lot of programs, and you kind of see this logarithmic scaling curves as you sample more and more programs. Well, this means that there is if you, like, plot the distribution of possible programs sampled by an LLM doing outputting codes, while you have some nonzero mass on like, you know, with long tails on, you know, very creative and good solutions. Right? And you only get these solutions if you sample like 1000000 times. So this way, I think I can't say that they're not creative because, well, if you sample high enough, like, they will create these things. But I do believe that, you know, they're exponentially not creative, so, like, you would need exponentially more samples to make them creative. I think that's why symbols can actually maybe help by doing, you know, symbolic programming synth program synthesis or what what we call programs, maybe just sequence of symbols. We we could fight this creativity issue by, you know, very easily going into a communal space that is completely novel and that, you know, you can't you can't you can't really keep track of, you can't amortize, you know, know, fighting things like well, going into things like computational irreducibility. So, yeah, I I do think program synthesis could help, and Symbols could help deep learning architectures to become more creative. Does No, that make

Creativity, Scaling & Future Directions

Tim Scarfe 00:45:09

it does. I'm very interested in creativity. I must admit, I think I might have actually changed my position on it because I took a very almost human chauvinist approach to creativity that, you know, there's creativity gap in LLMs that we have access to a source of creativity that LLMs don't. And so many people I've interviewed in Europe are just telling me the opposite. In fact, Sabaro Kambahati said the opposite in June at ICML. You know, he said, no, no, no. These things are very creative. And part of that is because creativity isn't just about novelty, it's also about interestingness. It's about things that we find interesting given our cultural biases and and interests and and so on. So they seem to capture a lot of that. And then, okay, you can say, we can greenblat it. And, we have to sample 30,000 times to get something interesting. But is that so different to us? Like, we're this huge collective intelligence, and we're all coming up with ideas, and most of them are rubbish. And every now and then we have this flash of inspiration. And, you know, perhaps we're embedded and situated, we have access to all of this entropy and whatnot, but is it really so different?

Clement Bonnet 00:46:10

Yeah, I think there's something to say about collective intelligence. I haven't thought that much about collective intelligence in general, so not sure I can comment very well on that, but if you focus for a second on individual intelligence, it seems obvious that you don't sample 1000000 programs in your mind when trying to solve an arc task or when trying to come up with something creative. There is, you know, some kind of synthesis that is much more efficient happening in your brain. And I think I'm I'm interested in trying to get closer to this kind of synthesis. Now I do believe there's something, you know, very interesting things to say about collective intelligence, which brings much more, you know, entropy and a lot more of the sampling that we talked about. Yeah, so I maybe both schools, you know, like, could come up with interesting breakthrough.

Tim Scarfe 00:46:57

When when you solve an arc challenge, you being creative?

Clement Bonnet 00:47:00

Well, the way I think about it, when I solve when I try to solve an ARC task, I so, again, I have some intuition of what the task, you know, might look like in, like, you know, tenth of a second. I see the structures. I see regularities very, like quickly, and I realize, oh no, I mean this is not as easy as this. And then I try to, you know, figure I propose a few hypotheses, maybe 2, 3, 4, and I refine them, and then the fifth is actually correct, checking matches all specification, and I just apply it, and I and I get it. So this is maybe on a half on the task. On the other half, it takes me a bit more time, but that's the idea. I do believe we have this kind of, like, few introspection, few hypotheses that we propose, and this this hypothesis seemed to be quite creative in a way that they they, you know, they could be very different from each other. And and, yeah, I felt like they would recombine, you know, priors in a way that is quite, you know, combinatorial and compositional. I mean, I need to stay humble in how I solve the task because many of them actually struggle with. But I think that's it's this idea of, like, coming up with a few hypotheses, testing them, but not coming up with 1000000 hypotheses. So it's how do you, you know, cut down this search space in an efficient way. There's probably much more than that in human cognition. I think getting this would be a good approximation at least.

Tim Scarfe 00:48:23

We scaled your solution up ridiculously, right? So, you know, much more training data, much bigger model and so on, and we trained it for longer, what do you think would happen?

Clement Bonnet 00:48:33

So I think it could scale quite a bit, especially if we scale the latent space. I do believe it would be very so I think 1 limitation is how much you can actually search in that space. I would expect this space to be actually not that linear, not that interpolative, not that smooth. And so if if, you know, you end up scaling task and and problems, yeah, I would end up this space to actually be harder to search. So it it's going to be about how much search you can do during training. You would need to scale this up probably, you know, keeping everything everything the same in this architecture. And so it would actually be quite costly, I think, to scale, like, both the kind of encoder decoder and the latent space as well and the sorry, and the search in this latent space. So I think there are better ways we could iterate on, like, this idea. I do believe coming up with, you know, a small representation of whatever task, whatever query you have can be a very useful way to to synthesis, to, you know, adapt, to to deal with epistemic uncertainty as well. I do not believe we can estimate epistemic uncertainty currently, but I think searching through this space could help resolve some uncertainty.

Tim Scarfe 00:49:47

That's very insightful, you're saying when we actually scale this up, the latent space might become less interpretive. Could we do meta optimization or could we have a different type of latent space or just sort of spitballing here, but you know, kind of topological space or some kind of graph representation, or you know, what else could we do?

Clement Bonnet 00:50:06

I think it's not just about scaling the latent dimension of your space, which has been the case with, you know, large language models. I think what's from, you know, my intuition was happening in large language models, that they have these very intricate, you know, very spurious, highly dimensional, you know, vector functions that can be recombined in some narrow way. And it's very useful, but it's very kind of it's very local. Like, you're trying to find, like, the obstruction locally that, like, is a little bit better. And I think I think it doesn't give you all these things around composition and auto distribution generalization. So, yeah, I I'm a big believer in, like, trying to find, you know, small representations that could explain your your outputs So for, you know, like, natural language would be a query and then the answer to a query. Now how do you how do you build these spaces? How do you search through them? I think these are all open questions. I've seen lots of interest in the direction lately, so it's very exciting.

Tim Scarfe 00:51:05

Yeah. Amazing. Well, congratulations on the work and your recognition from Charlez. Completely blank slate, unlike any of the other approaches. Very interesting indeed. So yeah, keep up the good work and thank you for coming on.

Clement Bonnet 00:51:18

Thank you so much, team. Thank you for having me.

References

[1] **ARC-AGI Benchmark**
https://arxiv.org/abs/2412.04604
paper

[2] **Latent Program Networks**
https://arxiv.org/abs/2411.08706
paper

[3] **Kevin Ellis - Program Synthesis**
https://www.cs.cornell.edu/~ellisk/
person

[4] **Induction vs Transduction in Abstract Reasoning**
https://arxiv.org/abs/2411.02272
paper

[5] **Variational Autoencoders**
https://arxiv.org/abs/1312.6114
paper

[6] **re-ARC Framework**
https://github.com/michaelhodel/re-arc
tool

[7] **Critique of Deep Learning**
https://arxiv.org/abs/2002.06177
paper