

(A big thanks to [Elodie Tordjman](https://il.linkedin.com/in/elodietordjman) from [Smart-content](http://smart-content.fr/smart-content/) for her great advice on this blog post)

Have you ever been given a task/project starting with: “Wouldn’t it be great if we had capability X? It would be a game-changer in so many aspects for our company!”. While this sounds like a very positive and energizing task, I’ve become more and more skeptical over the years of such tasks, because they lead you to build a “Solution In Search of a Problem”, a term popularized [by Y Combinator to describe one of the most common startup pitfalls](https://www.ycombinator.com/library/8g-how-to-get-startup-ideas):

*“The third mistake is to start with a solution instead of a problem. [...] We see this so commonly at YC that we have a term for it. It's called a "Solution In Search of a Problem," or a SISP. And these are usually not great, because usually, you never actually find the problem. You're much better off starting with a problem, and then looking for solutions.”*

I’ll argue that what is true for a startup is actually also very true for AI algorithms. Actually, even if a task is properly defined as a problem, **we often tend to jump on building a solution, rather than investigating the pain point**, simply because it is our comfort zone: after all, we’ve all been trained and hired for our “problem-solving” skills right? Who wants to have a “problem-digger”?

This post will be divided into 2 parts:

1. **How this SISP trend materializes in the AI world (both academic and industrial).**
2. **7 practical tips to make sure your AI solution stays focused on solving a real and meaningful problem.**

# A look at the industry - it happens more than you’d think!

## An example from the industry

In 2011, Microsoft released a consumer gaming accessory called [Kinect](https://en.wikipedia.org/wiki/Kinect) that enabled people to play by moving freely in front of their camera. In addition to being a huge commercial success, it was also a very hard technological achievement. The Kinect team not only managed to build a brilliant gesture detection algorithm, but also made a huge hardware achievement: in order to provide a truly seamless experience, Microsoft realized that they would need a special type of camera, called “depth cameras” (a camera able to estimate the 3D location of each pixel). They found a way to miniaturize them, make them much cheaper, and manufacture them at scale, partnering with an Israeli startup called Primesense, which was later [acquired by Apple](https://techcrunch.com/2013/11/24/apple-primesense-acquisition-confirmed/) in a surprising turn of events. Kinect is in my opinion the perfect example of a problem-driven technological breakthrough, and this is in my opinion why it was so successful.

However, as is very often the case, this technological achievement somehow gave birth to many solution-driven technological failures. Suddenly, other big companies realized that these cameras were able to achieve what regular cameras couldn’t and that these cameras would “open doors to many amazing new applications for smartphone and tablet cameras”. Google opened a project called [Tango](https://en.wikipedia.org/wiki/Tango_(platform)), Intel a project called [RealSense](https://www.intelrealsense.com/), and [Apple acquired Primesense](https://techcrunch.com/2013/11/24/apple-primesense-acquisition-confirmed/).

Fast-forward 10 years later, what happened to these projects? Well, the results are unfortunately not brilliant. Google managed to manufacture a few devices, but didn’t achieve any commercial success with Tango, and [shut down the program](https://www.theverge.com/2017/12/15/16782556/project-tango-google-shutting-down-arcore-augmented-reality). Intel manufactured a few RealSense products, built a lot of software, but finally didn’t reach any meaningful commercial success and recently announced that [they will be shutting down their RealSense department](https://www.therobotreport.com/intel-realsense-camera-business-being-shut-down/). Apple did achieve commercial success with iPhones and iPad that have depth cameras integrated, but never really found a “killer-app”. To my knowledge, the most important use-case of the iPhone depth camera is currently FaceID, which allows you to unlock your smartphone with your face. And yes, FaceID is cool, but will it be remembered as a game-changing technology? Is it so much better than fingerprint-based or regular-camera-based identification? My personal feeling is that Apple successfully sold depth-equipped smartphones not because of the depth cameras, but because of other features (and most importantly… Because it was the new iPhone!).

## An example from the academy

In 2017, some researchers from Google introduced [the Transformers architecture](https://en.wikipedia.org/wiki/Transformer_(machine_learning_model)#:~:text=in%20successive%20layers.-,Architecture,thing%20to%20the%20encoder's%20output.), and it was a huge breakthrough in the world of NLP/NLU (Natural Language Processing/Understanding) because it enabled a dramatic performance boost on almost every language-related AI task. But exactly like in my previous examples, after the [seminal paper introducing Transformers](https://arxiv.org/abs/1706.03762), many people tried to apply Transformers to other domains, and in particular computer vision, and even managed to show some [slight improvements](https://arxiv.org/abs/2010.11929) on ImageNet classification. However, in computer vision, we already have an architecture that works really well: Convolutional Neural Networks (CNN). Actually, an amazing recent paper called [“ResNet Strikes back”](http://arxiv.org/abs/2110.00476) proved that the most trivial CNN architecture can reach the same level of performance as Transformer-based architectures if they are trained correctly. Of course, this doesn’t mean that transformer will never be useful for images, but why not rather spend time to leverage transformers in much harder AI problems where CNNs do not work well, such as [audio processing](https://towardsdatascience.com/the-promise-of-ai-in-audio-processing-a7e4996eb2ca), [geometric deep learning](https://geometricdeeplearning.com/), few-shots classification, 3D mesh processing, etc…?

That being said, I have to admit that I’m being a bit unfair with the AI academic world. Maybe academic research SHOULD be solution-driven and not problem-driven. Some problems in the world are so hard to fix that the only way to solve them is by trying lots of things and “connecting the dots backward” (as [Steve Jobs famously said](https://www.brainyquote.com/quotes/steve_jobs_416875)). The world obviously needs some solution-driven research to accomplish great things, but people engaging in this path should be aware that their impact will at best be in the long run, at worst be null.

# 7 tips to make sure you are building a problem-driven AI

## Get business insights about the problem

Whenever you receive a task, it is very likely that the person giving you the task (e.g. an Algorithm Team Lead) is not the real source of the task. More likely, they heard from other people in the company about a painful problem that needs to be addressed and thought it could be a good fit for your skills.

Therefore, your first goal should be to **become the most knowledgeable person about this problem**: go and talk to the people that really FEEL the pain. Very often, this should involve talking to customers, but if it’s difficult for any reason, try to at least ask as much about it to your Product Manager, or customer-facing people. In particular, I recommend this [user interview framework from Startup School](https://www.youtube.com/watch?v=MT4Ig2uqjTc) when trying to understand a problem:

* What's the hardest part about the problem?
* Tell me about the last time you found this problem?
* Why was that hard?
* What, if anything, have you done to try to solve this problem?
* What are the issues with the solutions you've tried?

Another famous and in my opinion very efficient framework to understand the root cause of a problem is to ask the [5 whys](https://en.wikipedia.org/wiki/Five_whys). This will really help you get to the core reasons behind your problem, and I found that it can be particularly helpful later on when you’ll need to find creative solutions to solve the problem.

## Check that the problem is correctly prioritized

Once you understand the problem, try to get a very rough sense of its worth. How often does the company lose a deal because of this? How much engineering time would be saved if it were solved? How important is this problem compared to others that you could work on?

This is actually a very tricky part. First, because prioritization requires having the big picture, and this is something you don’t always have. In addition, as an AI scientist, you usually don’t have the luxury to decide whether a task should or should not be prioritized. However, you should always have the ability (it’s even a duty!) to discuss with the person that gave you this task, and try to change their mind if you don’t agree with them. **Making correct prioritization is probably one of the toughest job leaders have**, so help them be better at their job: they will appreciate it, and you will avoid working in vain. I’ve seen too many AI scientists making amazing solutions that never made it to production because they turned out not to be so important at the end, and hence the deployment task never got prioritized.

## Prove that this problem really requires you

Especially if you’ve done the 5 whys exercise, you should now see the deep root causes of the problem you are trying to solve. At that point, you may realize that the main problem might be solved by something very different from what you’ve been originally told. It could potentially be anything: a different marketing strategy, a better internal process, or simply the integration of some external software.

The most frequent example is when you find a third-party company that already offers a product that solves your problem. Of course, it will cost money, but unless the third-party company has a VERY bad pricing strategy, this solution should be a better deal than building it in-house from scratch. So before you start writing a single line of code, **make sure that you’ve done serious market research**, and that a commercial product suiting your needs does not already exist.

## Build a good evaluation pipeline

If you’ve done your research correctly on the problem understanding part, you should already have a good idea of what an ideal solution looks like. You may even have a few brilliant ideas on how to tackle the problem. However, it is very important that you **refrain from coding your brilliant solution**, otherwise, you’ll have no idea how good it is, and unfortunately, no one will be convinced that it works just because it comes from your brilliant mind, or because it is based on the latest CVPR award-winning paper.

At this stage, you should design a robust and informative way **to evaluate any** **solution to the problem you’re solving**:

* **Quantitative performance tests:** metrics that really characterize how good your solution is. In general, one needs to define several metrics that are anti-correlated, such that it is very hard to increase one without hurting the other.
* **Qualitative performance tests:** even though it’s important to define the algorithm’s quality through clear metrics, these metrics need in general to be complemented with human-based judgment. For example, if you are building an AI that generates images, and want to evaluate how good-looking these images are, you can use metrics such as [FID](https://en.wikipedia.org/wiki/Fr%C3%A9chet_inception_distance) or [PPL](https://towardsdatascience.com/explained-a-style-based-generator-architecture-for-gans-generating-and-tuning-realistic-6cb2be0f431), but at some point, someone will have to take a look at them (spoiler: most of the time, YOU will have to!!).
* **Divide the performance tests** above into test sets that
  + Represents as faithfully as possible the data you’ll have in production
  + Help you find your solution’s strengths and weaknesses **before being deployed**

Needless to say that the process to test a given solution and visualize its results should be as straightforward and convenient as possible (one push button, ideally). It will save you a lot of precious time later when you’ll need to iterate. On this topic, I highly recommend tools such as [Tensorboard](https://www.tensorflow.org/tensorboard) and [Weights & Biases](https://wandb.ai/site) for tracking machine learning results, or [QA-Board](https://samsung.github.io/qaboard/) for more generic algo projects (QA-Board has in particular an amazing image comparison tech). You may also want to use tools such as [Kubeflow](https://www.kubeflow.org/) that enable you to manage and run many machine learning experiments in parallel. I’ll probably make a separate blog post on MLOps, so that we can dive more into those tools and the pain points they solve.

Interestingly, this idea that one should write problem-defining tests before coding the solution that will pass those tests is very famous and widely adopted in the world of software development. It’s called [test-driven development](https://en.wikipedia.org/wiki/Test-driven_development), and Robert Martin (Uncle Bob) [has a wonderful presentation](https://www.youtube.com/watch?v=qkblc5WRn-U) on this topic. For some reason, its adoption in the world of AI is still relatively low.

## Think big, start small

The previous company I worked for (Via Transportation) had an amazing catchphrase for the R&D teams, that I try to remember whenever I envision a new challenge: “**think big, start small**”. Now that you’ve set up a great testing environment, your next step is to build a baseline, i.e. the **SIMPLEST algorithm you can write in your test environment that will get you ANY performance score** (yes, this score does need to be great). Don’t try to be too sophisticated there, the goal is to have something to compare to, and to get some convincing evidence that the complicated algorithm you will build next **really had to be complicated**.

## Iterate fast

Once you have a baseline and a good evaluation environment, the algo fun can finally begin! **This is the place where your research talent comes most into play**: read papers, get interesting insights about the data and your problem, implement things, etc… However, keep in mind that it’s not just about being smart, but **also being able to iterate fast**. Here again, the resemblance with startups is stunning: because of the very uncertain environment of algorithm research, it is often VERY worth spending time on the iteration speed process rather than figuring out the next thing you want to try. Typically, do not train a network for 7 days if you have no idea whether it’s going to work at the end.

## Move to prod as fast as possible

Ok, so you’ve finally got your first good-enough algorithm, but at this stage, chances are you’re still not satisfied… There are so many things you haven’t optimized yet! So many other ideas you’d like to experiment! But this is also where **you need the discipline to focus on your problem** (the one you fell in love with, remember :)?). Even though your algorithm seems pretty basic and very improvable, you should try to push it to production as soon as possible, since it is probably already better than what exists today.

The reasons for this are:

1. The process to move things to prod is slow and painful. You’ll probably have plenty of time to improve your algorithm during the deployment process, and in most cases, you will be able to update your model/code without problems.
2. The process of deploying will likely teach you a lot about the constraints your solution needs to fit in, and you might suddenly realize that some key performance aspect of your algorithm was not properly taken into account by your evaluation metrics/pipeline: in other words, you’re back to step 4 ;-).
3. Once you’ve made the full cycle of moving to prod, you should be able to iterate and deploy improvements on a regular basis. This will make your work much more visible, and if you care about the problem you’re solving, you’ll feel great that your work actually makes someone else’s life better.

# Conclusion - Wait, is this really my job?!!

You may have noticed that most of the tips in this article don’t require reading lots of papers, implementing super-fancy-math-based AI algorithms, etc… So I think that it’s important for me to clarify a few things.

1. I didn’t write a lot about step #6 (will probably explain more about fast iteration in a future post), but it is probably the hardest, longest, and most uncertain part of an AI development process, so in practice, you can expect to spend a significant time on researching a good solution for your problem. But don’t forget the other steps!
2. In some organizations, other people (e.g. product managers) will perform steps 1 to 3, so that you can focus on others that require more coding. Therefore, if you only want to code, some organizations may help you do that.
3. That being said, let’s face the ugly truth: real-life impactful AI is not academic research, and it’s not even a Kaggle competition. If you are intrinsically motivated by the value you’re creating for the world, and the cause you are helping move forward, you’ll eventually thrive, be recognized for your work, **and find huge satisfaction in seeing the impact of your work**. On the other hand, if you only want to do the fun and interesting parts of AI, write a paper or add a new technical skill to your resume, then you are probably better off doing academic (Vs “applied”) research, which is actually also a great path that I highly respect, but it’s a very different one!