Andrej Karpathy Follow
Director of AI at Tesla. Previously Research Scientist at OpenAI and PhD student at Stanford. I like to train deep neural nets on large datasets.
Nov 12 · 7 min read

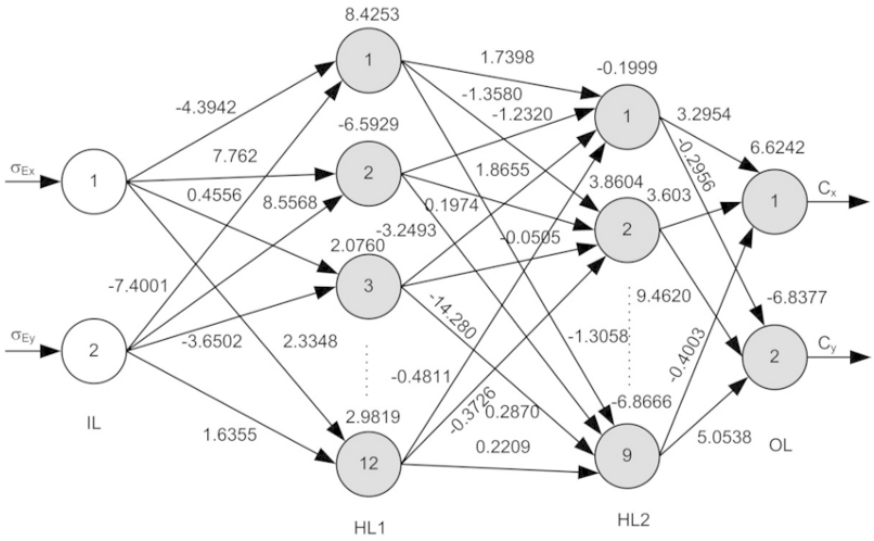
Software 2.0

I sometimes see people refer to neural networks as just “another tool in your machine learning toolbox”. They have some pros and cons, they work here or there, and sometimes you can use them to win Kaggle competitions. Unfortunately, this interpretation completely misses the forest for the trees. Neural networks are not just another classifier, they represent the beginning of a fundamental shift in how we write software. They are Software 2.0.

The “classical stack” of **Software 1.0** is what we’re all familiar with—it is written in languages such as Python, C++, etc. It consists of explicit instructions to the computer written by a programmer. By writing each line of code, the programmer is identifying a specific point in program space with some desirable behavior.

In contrast, **Software 2.0** is written in neural network weights. No human is involved in writing this code because there are a lot of weights (typical networks might have millions), and coding directly in weights is kind of hard (I tried). Instead, we specify some constraints on the behavior of a desirable program (e.g., a dataset of input output pairs of examples) and use the computational resources at our disposal to search the program space for a program that satisfies the constraints. In the case of neural networks, we restrict the search to a continuous subset of the program space where the search process can be made (somewhat surprisingly) efficient with backpropagation and stochastic gradient descent.

It turns out that a large portion of real-world problems have the property that it is significantly easier to collect the data than to explicitly write the program. A large portion of programmers of tomorrow do not maintain complex software repositories, write intricate programs, or analyze their running times. They collect, clean, manipulate, label, analyze and visualize data that feeds neural networks.



Software 2.0 is not going to replace 1.0 (indeed, a large amount of 1.0 infrastructure is needed for training and inference to "compile" 2.0 code), but it is going to take over increasingly large portions of what Software 1.0 is responsible for today. Let's examine some examples of the ongoing transition to make this more concrete:

Visual Recognition used to consist of engineered features with a bit of machine learning sprinkled on top at the end (e.g., SVM). Since then, we developed the machinery to discover much more powerful image analysis programs (in the family of ConvNet architectures), and more recently we've begun searching over architectures.

Speech recognition used to involve a lot of preprocessing, gaussian mixture models and hidden markov models, but today consist almost entirely of neural net stuff.

Speech synthesis has historically been approached with various stitching mechanisms, but today the state of the art models are large

convnets (e.g. WaveNet) that produce raw audio signal outputs.

Machine Translation has usually been approaches with phrase-based statistical techniques, but neural networks are quickly becoming dominant. My favorite architectures are trained in the multilingual setting, where a single model translates from any source language to any target language, and in weakly supervised (or entirely unsupervised) settings.

Robotics has a long tradition of breaking down the problem into blocks of sensing, pose estimation, planning, control, uncertainty modeling etc., using explicit representations and algorithms over intermediate representations. We're not quite there yet, but research at UC Berkeley and Google hint at the fact that Software 2.0 may be able to do a much better job of representing all of this code.

Games. Go playing programs have existed for a long while, but AlphaGo Zero (a ConvNet that looks at the raw state of the board and plays a move) has now become by far the strongest player of the game. I expect we're going to see very similar results in other areas, e.g. DOTA 2, or StarCraft.

You'll notice that many of my links above involve work done at Google. This is because Google is currently at the forefront of re-writing large chunks of itself into Software 2.0 code. "One model to rule them all" provides an early sketch of what this might look like, where the statistical strength of the individual domains is amalgamated into one consistent understanding of the world.

The benefits of Software 2.0

Why should we prefer to port complex programs into Software 2.0? Clearly, one easy answer is that they work better in practice. However, there are a lot of other convenient reasons to prefer this stack. Let's take a look at some of the benefits of Software 2.0 (think: a ConvNet) compared to Software 1.0 (think: a production-level C++ code base). Software 2.0 is:

Computationally homogeneous. A typical neural network is, to the first order, made up of a sandwich of only two operations: matrix

multiplication and thresholding at zero (ReLU). Compare that with the instruction set of classical software, which is significantly more heterogenous and complex. Because you only have to provide Software 1.0 implementation for a small number of the core computational primitives (e.g. matrix multiply), it is much easier to make various correctness/performance guarantees.

Simple to bake into silicon. As a corollary, since the instruction set of a neural network is relatively small, it is significantly easier to implement these networks much closer to silicon, e.g. with custom ASICs, neuromorphic chips, and so on. The world will change when low-powered intelligence becomes pervasive around us. E.g., small, inexpensive chips could come with a pretrained ConvNet, a speech recognizer, and a WaveNet speech synthesis network all integrated in a small protobrain that you can attach to anything.

Constant running time. Every iteration of a typical neural net forward pass takes exactly the same amount of FLOPS. There is zero variability based on the different execution paths your code could take through some sprawling C++ code base. Of course, you could have dynamic compute graphs but the execution flow is normally still significantly constrained. This way we are also almost guaranteed to never find ourselves in unintended infinite loops.

Constant memory use. Related to the above, there is no dynamically allocated memory anywhere so there is also little possibility of swapping to disk, or memory leaks that you have to hunt down in your code.

It is highly portable. A sequence of matrix multiplies is significantly easier to run on arbitrary computational configurations compared to classical binaries or scripts.

It is very agile. If you had a C++ code and someone wanted you to make it twice as fast (at cost of performance if needed), it would be highly non-trivial to tune the system for the new spec. However, in Software 2.0 we can take our network, remove half of the channels, retrain, and there—it runs exactly at twice the speed and works a bit worse. It's magic. Conversely, if you happen to get more data/compute, you can immediately make your program work better just by adding more channels and retraining.

Modules can meld into an optimal whole. Our software is often decomposed into modules that communicate through public functions, APIs, or endpoints. However, if two Software 2.0 modules that were originally trained separately interact, we can easily backpropagate through the whole. Think about how amazing it could be if your web browser could automatically re-design the low-level system instructions 10 stacks down to achieve a higher efficiency in loading web pages. With 2.0, this is the default behavior.

It is easy to pick up. I like to joke that deep learning is shallow. This isn't nuclear physics where you need a PhD before you can do anything useful. The underlying concepts require basic linear algebra, calculus, Python and some lectures from CS231n. Of course, there is a good amount of expertise and intuition that one can acquire over time, so a more precise statement is that the Software 2.0 stack is easy to pick up but non-trivial to master.

It is better than you. Finally, and most importantly, a neural network is a better piece of code than anything you or I can come up with in a large fraction of valuable verticals, which currently at the very least involve anything to do with images/video, sound/speech, and text.

The limitations of Software 2.0

The 2.0 stack also has some of its own disadvantages. At the end of the optimization we're left with large networks that work well, but it's very hard to tell how. Across many applications areas, we'll be left with a choice of using a 90% accurate model we understand, or 99% accurate model we don't.

The 2.0 stack can fail in unintuitive and embarrassing ways ,or worse, they can "silently fail", e.g., by silently adopting biases in their training data, which are very difficult to properly analyze and examine when their sizes are easily in the millions in most cases.

Finally, we're still discovering some of the peculiar properties of this stack. For instance, the existence of adversarial examples and attacks highlights the unintuitive nature of this stack.

Last few thoughts

If you think of neural networks as a software stack and not just a pretty good classifier, it becomes quickly apparent that they have a huge number of advantages and a lot of potential for transforming software in general.

In the long term, the future of Software 2.0 is bright because it is increasingly clear to many that when we develop AGI, it will certainly be written in Software 2.0.

And Software 3.0? That will be entirely up to the AGI.