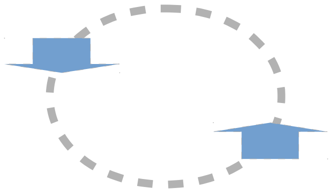
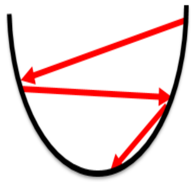
Searching Stateful Spaces

Optimizing Search Functions

- Optimization is a search
- Search is a function
- Function is a subject of optimization
- Optimization is a search…



The text is an attempt to think through this (seemingly circular) chain of declarations. Let's now take it one step at a time.



Any optimization problem can be expressed in terms of searching. A brute force searches meticulously traverse the entire search space while trying to visit each and every point only once. A blind search starts from an arbitrary point, and picks the next one at random. It'll stop when target is reached, based on a given definition of what the (optimization) target is, or what it may be. That is, unless it runs out of time first, of course...

Search logic, on the other hand, can always be interpreted as a function that takes the following *input*:

A) current position in the space, plus
B) possibly some kind of measure of a distance, or a direction that would lead us closer to the target, and
C) optionally, the previous search step, or the entire sequence of previous steps, or a part of thereof – translated into:
D) the next step, resulting in a new position

Note that A and C imply a definition of search-as-a-function problem domain – not to be confused with the domain of the function that is being optimized or *learned* in the first place. Functions (and that is the final argument in the **optimization ⇨ search ⇨ function** sequence) – do map domains. A function takes an input defined on its *input domain*, and maps it onto its *output* (the item D above).

Being a function, the search itself can be viewed as an optimization target in the separate problem space that contains all possible searches (i.e., search functions), including blind-random and brute-force. Important distinction is that the item **B** above – the measure of a distance, or a direction leading closer to the search target – is generally tough to figure out. But more on that later.

The rest of this paper is structured as follows:

Optimizing Search Functions....1
A Hybrid Example....2
Searching Stateful Spaces....3
NFL, and The Apparent Futility of Asking Big Questions....5
$\mathbb{R}^2$ ⇨ $\mathbb{R}^2$ illustration....6
Meta-learning....7
Super RNN....9

A Hybrid Example

There is, for instance, a popular hybrid-storage case comprised of a storage initiator connected to two or more storage targets with different capabilities. A conceptual SSD target (Fig. 1), although limited in capacity, features superb performance. HDD target, on the other hand, would deliver abundant, practically free, capacity, albeit coupled with below-average performance:

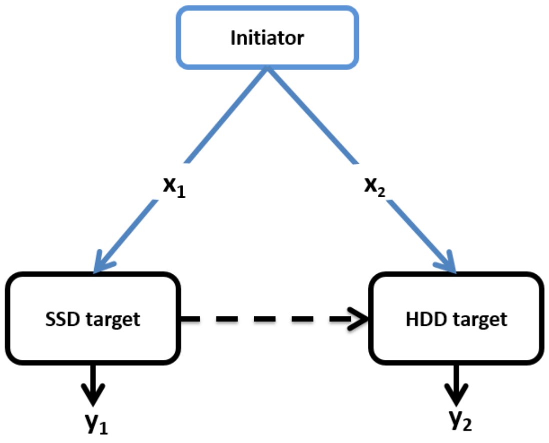


Fig. 1. Hybrid storage (pattern)

The Fig. 1 pattern is clearly motivated by the idea of combining best-of-both as far as $/IOPS and $/GB. Given comparable implementations and hardware resources, the resulting performance will depend on the numbers of unacknowledged IOs in flight between the initiator and the respective targets. But only in part.

The second, slightly less obvious part of the hybrid "performance equation" is defined by the distributed internal state of the [initiator + targets] system – a state that in turn is a function of the SSD target's remaining capacity at a given point in time, and the rate at which this capacity is getting consumed. And, possibly, the second derivative of this rate as well.

In its most distilled and basic realization, and in the presence of only two storage targets (Fig. 1), the system performance function would map $\mathbb{R}^2 \Rightarrow \mathbb{R}^2$, with two per-target queue depths on the input side of the mapping (denoted as x_1 and x_2 on the picture), and two performance numbers y_1 and y_2 on the output (for instance, SSD and HDD respective IOPS).

Generally, when there's a distributed system that executes IO requests, there will be an $R^n \Rightarrow R^m$ function $Y(t) = F(X(t), S(t))$ "computing" the system's runtime performance, where:

- $X(t)$ is the *input* that carries various IO parameters (e.g., per-target queue depths, I/O sizes, spatial and temporal distributions, synchronicity, etc.)
- $S(t)$ is the internal *state* that depends on the system's resources and/or history of previous *inputs* and/or previous state transitions

Searching Stateful Spaces

When the time's quantized, the $Y_t = F(X_t, S_{t-1})$ notation indicates a time-stepped ($t = 0, 1, 2, \ldots$) progression of a system through its multi-dimensional space of states, with each state S_t being defined by the previous state(s) and the current input:

$$Y_t = F(X_t, S_{t-1}), \text{ where}$$

$$S_t = G(X_t, S_{t-1})$$

Generally, this type of nonlinear and not necessarily differentiable $F()$ and $G()$ behaviors can be *learned* (and therefore, **optimized**) through two separate machine learning (ML) techniques:

- Reinforcement Learning (RL)
- Recurrent Neural Network (RNN)

The two are ostensibly different – but also similar in so many ways. Reinforcement Learning (with its TD and Q-learning *model-free* algorithms) looks at the world of stateful dynamic systems through a prism of agents (that *act*), and an environment that is, effectively, everything else other than the agents:

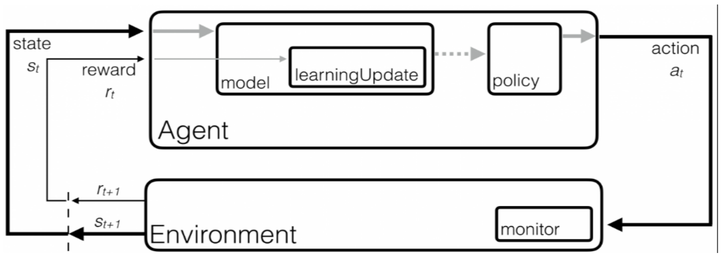


Fig. 2. RL agent⇔environment diagram

In particular, the environment responds to an agent's actions by generating *rewards* which in turn provide positive or negative feedback to the agent, to modify its own runtime behavior (a.k.a. *policy*) via a so-called *learning update* (Fig. 2).

Recurrent neural nets, on the other hand, are often designated as artificial neural networks with memory. This memory, denoted as h_{t-1}, gets conflated (⊗ sign on Fig. 2) with the *next* environment-generated input x_t, to produce a new output (not shown) and a new internal state h_t:

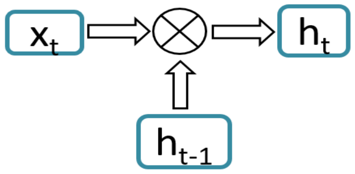


Fig. 3. Recurrent "superposition"

Unlike RL type methods (where *rewards* must be explicitly defined), neural networks get rewarded or, rather, penalized via changes in the values of associated cost functions and through the sci-fi inspired mechanism called back propagation through time (BPTT). The latter provides *learning updates* (Fig. 2) to the RNN's numerous internal parameters (aka, weights).

Here's a quick summary that puts two respective *terminologies* in perspective, side by side:

Reinforcement Learning	**Recurrent Neural Network**
Learning update	BPTT
State	Network memory (activations of the RNN's hidden layers at previous timestamps)
Action	Output
Reward	(-) Cost for instance, Mean Squared Error (MSE)

Table 1. RL vs RNN terminologies

What's important is the *statefulness* of both machine-learning techniques, as well as their respective algorithmic ability to translate environmental feedback into internal operation-improving changes – learning updates. As time goes by, the cumulative effect of those (very special type) searches translates into lesser cost (for RNN) and larger cumulative rewards (for RL), respectively.

Searching for usable RNN

Any existing (and future) machine learning technique that can be utilized to *learn* a given unknown function $F()$ will entail:

1) Using existing ML framework, or building a new one (done)
2) Optionally, modeling and simulating a stateful system that generates realistic performance numbers (done)
3) Benchmarking, logging, and otherwise collecting data
4) Writing scripts, parsing the logs, cleaning and normalizing the data, and more.

Some of these items are heavy-duty tasks, and investments. For some of these items we would probably want to have an assurance that the learning-model part (for instance, RNN's parameters and its architecture) was done right. Let's step back, therefore, and take a look at RNNs from a *configurability* perspective.

RNNs are famous for providing unbounded freedom, in terms of the size and numbers of hidden layers, the amount of past they can store, and the degree to which this past can influence the present – and therefore, the future.

Choosing an optimal machine-learning architecture, its hyper-parameters, and its other numerous tunables becomes, therefore, a non-trivial task in and of itself. Especially in multi-dimensional and stateful (translation: non-trivial real-life) cases.

But how to choose anything when we are facing myriad possibilities? The one answer that inevitably comes first is – trial and error (which would be a rather limited version of blind search). That is not a good answer, though.

> Other than a mind-blowing cardinality of the associated *search domain*, RNNs feature the so-called vanishing gradient as well as exploding gradient problems – two sides of the same "coin" rooted in the chain rule for computing derivatives of the composition of functions. This is because, having time dimension, RNNs are particularly predisposed to generate lengthy functional compositions.

In the space of all RNNs that can be used for a given stateful $\mathrm{F}()$ – what could be the search strategy that finds the optimal one? Notice that in presence of any real-time or close to real-time requirements (e.g., when trying to optimize performance of a hybrid-storage system, Fig. 1), the difference between fast and slow RNN is a difference between usable and unusable.

That is the question. Is there a methodology for searching the domain of all (or at least some) possible RNNs? Anything?

NFL, and The Apparent Futility of Asking Big Questions

Optimization is a search. Search is a function that must itself be optimizable.

Given nonlinear $\mathbb{R}^{n} \Rightarrow \mathbb{R}^{m}$ transformation that must be *learned*, and given defined limits on size/complexity of neural networks used for this *learning* (e.g., maximum depth of the network and maximum size of its hidden layers) – is there a way to predict that a network that is substantially below permitted maximums will consistently perform better? That it'll converge orders of magnitude faster, with better cost/loss results by orders of magnitude?

> Since artificial networks were inspired by natural networks, it is only natural that the language itself ("neurons", "synapses", etc.) motivates further inferences.
>
> For instance, the question is whether RNNs could possibly "rehabilitate" themselves when running with fewer memory-based neuron connections. How few would still be enough? And at which point during the training would the remaining part of RNN's "brain" sufficiently develop itself to, in effect, overcompensate?

Colorful connotations aside, at least some trade-offs are inevitable – trade-offs between the speed, the precision, and the applicability of learning models. In other words, there will be absolutely No Free Lunch (NFL). The 1997 NFL theorem states that for the problem of optimizing an arbitrary $\mathbb{R} \Rightarrow \mathbb{R}$ function, no algorithm performs better than a blind search. Meaning: a method of optimizing across the board simply does not exist. The same, of course, applies to higher dimensions.

Which is why it would be rather pointless to look for a methodology of searching through the space of applicable neural networks (for instance), hoping to find the best fit for an arbitrary $\mathrm{Y}_{\mathrm{t}} = \mathrm{F}(\mathrm{X}_{\mathrm{t}}, \mathrm{S}_{\mathrm{t}\text{-}1})$.

Which is also why it would make total sense to identify, and then resolve, non-trivial special cases – in particular, the case of all RNNs that are limited in depth (layer-wise and history-wise) and have no more than a given maximum number of neurons (aka units) per layer.

This is because the common wisdom tells us: when searching, the cardinality of the search domain is the first thing to try to reduce, if possible…

$\mathbb{R}^2 \Rightarrow \mathbb{R}^2$ illustration

This section is a pure illustration, motivated in part by the hybrid-storage example in Fig. 1. As stated, the runtime function in question would (minimally) map $\mathbb{R}^2 \Rightarrow \mathbb{R}^2$ and have a top-level requirement of being quite stateful and resolutely nonlinear (better yet, non-polynomial). Like, for instance, the following two examples:

```
Ys[i][0] = Xs[i][1] * math.Cosh(Ys[iprev][0]+Xs[iprev][0])
Ys[i][1] = Xs[i][0] * math.Sinh(Ys[iprev][1]+Xs[iprev][1])
```

and:

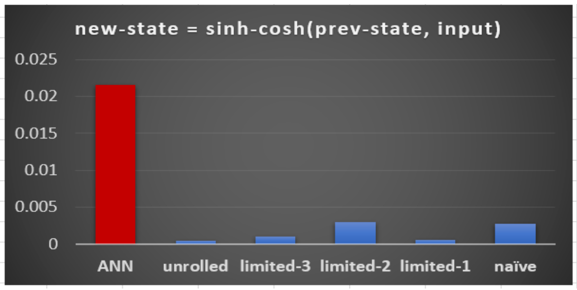
```
Ys[i][0] = Xs[i][1] * math.Cos(Ys[iprev][0]+Xs[iprev][0])
Ys[i][1] = Xs[i][0] * math.Sin(Ys[iprev][1]+Xs[iprev][1])
```

Fig. 4. Two stateful $\mathbb{R}^2 \Rightarrow \mathbb{R}^2$ examples

For the sake of brevity, the state "component" of a system-generated output above is folded into the Y_t output – via hyperbolic sine/cosine in one case, and conventional sine/cosine functions in another, with subscript $[\text{iprev}]$ corresponding to the previous instant of time $(t - 1)$.

> Source code and running instructions for these examples, as well as ANN/RNN implementation, can be found on github.
>
> Note also that all recurrent networks in this paper are variations of Elman RNN.

The Fig. 5 graphics compares (2, 4, 4, 2) neural networks used to *learn* the formulas above – RNNs with 2-dimensional inputs/outputs and 2 hidden layers, each comprised of 4 units. Height of the bars on Fig. 5 reflects MSE of separate testing examples after running each of the respected networks through 5 million training iterations. Notice that in both cases the memoryless Artificial Neural Network (ANN) does not converge.



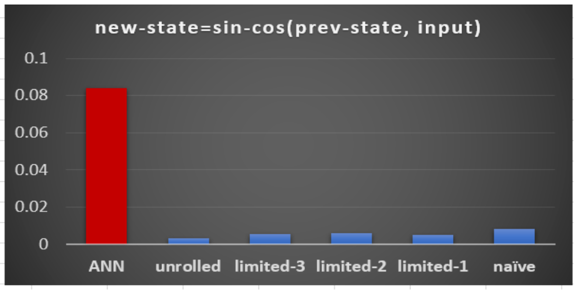


Fig. 5. Neural networks: MSE in comparison

All the rest of the X-axis labels denote recurrent networks of different degrees of *connectedness* between the past and the present. For instance, "limited-1" RNN restricts feed-forward pass (and therefore, back propagation pass) to a single neuron connection, while "unrolled" implies full (past ⇨ present) connectivity between all hidden layers (Fig. 6).

Finally, the configuration labeled "naïve" comes with a single $(t - 1)$ layer that backs up the very first hidden layer in the neural network.

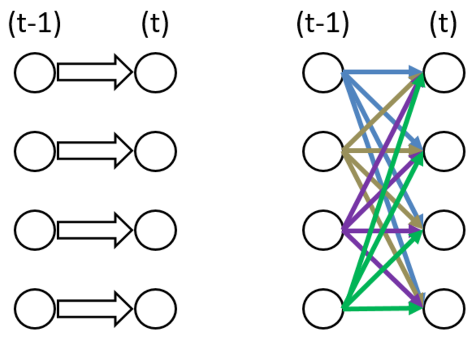


Fig. 6. RNNs: "limited-1" (left) and "unrolled" (right)

This section shows just one experiment in a (totally non-representative) series that included networks of up to 8 hidden layers, up to 128 hidden units in a layer, and a variety of stateful time-series $F(X_t, S_{t-1})$, such as, for instance, those shown on Fig. 4.

> One quick observation: my attempts to identify (or somehow guess) best-performing RNN prior to running benchmarks have been off the mark most of the time...

Meta-learning

When under-training, an RNN undergoes a sequence of state transitions:

$$W^h_{(t-1)} \Rightarrow W^h_{(t)} \text{ and } W^x_{(t-1)} \Rightarrow W^x_{(t)}$$

where W^h is combination of all "recurrent" weights (associated with arrows shown on Fig. 6), and W^x – the weights of all non-recurrent hidden units/neurons, or, more precisely, their connections. The Fig. 3, therefore, can be slightly modified as follows (Fig. 7):

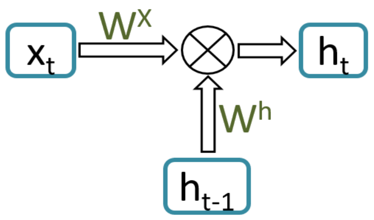


Fig. 7. Recurrent superposition: two sets of weights

Given a fixed network architecture, fixed hyper-parameters and selected optimization algorithm, the quest for an optimal RNN can be narrowed down to finding the optimal (W^h, W^x) pair that would yield the best cost/loss for the original $Y_t = F(X_t, S_{t-1})$, for all possible inputs (X_t, S_{t-1}). Since any search process is a function that can in turn be *learned* and optimized, the very first (and likely very dumb) idea that comes to mind is – using (W_{t-1}, W_t) as training examples – for both W^h and W^x, or just one of those sets of weights, or any subset of thereof.

In other words, we are talking *meta-learning*.

Meta-learning is defined as state of "being aware of and taking control of one's own learning" (Wikipedia).

The related machine-learning interpretation must sound like *co-training* of both a neural network, and another neural network (the meta-learner) that is being trained by "watching" and "observing" the former. This *other* neural network will, effectively, execute a search in the RNNs domain, by mapping a given *input* RNN to a more and more optimized *output*, as far as the original stateful $F()$ is concerned:

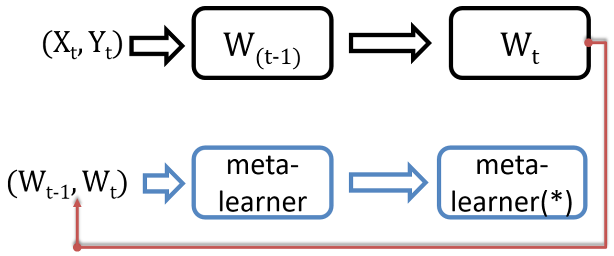


Fig. 6. Meta-learning

In this figure, the original RNN processes training mini-batches of (X_t, Y_t) to generate W^h and W^x updates. In parallel, the meta-learner (Fig. 6) is being fed pre- and post-update (W_{t-1}, W_t) pairs, to utilize them as a separate training sequence in its own right.

After a few (or a few million) iterations the meta-learner could be, supposedly, short-circuited to start generating RNN ⇨ RNN* in total autonomy (Fig. 7):

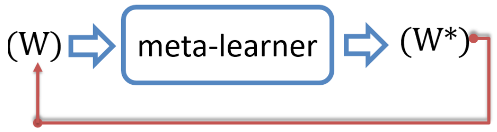


Fig. 7. Meta-learning short-circuited

A pair of fully connected hidden layers of size N adds precisely N^2 new variables (aka, weights) into the RNN-as-a-function. The $O(N^2)$ part gets further multiplied by the numbers of hidden layers as well as the number of stored instances of their (the layers) past activations – a fact that strongly indicates that the complexity of meta-learning can easily surpass the complexity of the original neural network.

Secondly – and this is much bigger problem – during the Fig. 6 process the meta-learner will be inevitably "observing" and being trained upon the sub-optimal *trajectory* of the original trainee. It is intuitively clear therefore, that the combined *co-training* results cannot include a better or version of this *trajectory*.

Super RNN

- Optimization is a search
- Search is a function
- Function is a subject of optimization...

Again, there must exist a way to search the domain of RNNs inversely defined by a given stateful $Y_t = F(X_t, S_{t-1})$ transformation. A member of such domain would have an input layer, an output layer, a bunch of hidden layers, and a *recurrent* state comprising previous hidden activations (Fig. 8):

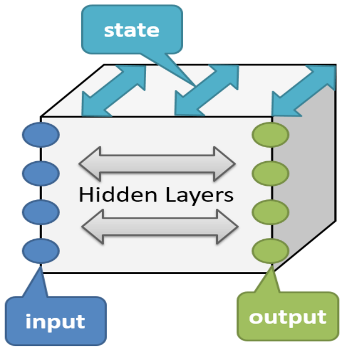


Fig. 8. RNN in 3D

Instead of, as in the previous section, "fixing" RNN's architecture, this time we enumerate the latter across its possible *dimensions*:

- number of hidden layers, say: 2, 4, 8, …, 64
- size of a hidden layer: N, 2N, 4N (where N would be the size of the input layer)
- number of the recurrent layers
- type of the recurrent layer and number of its connections: "unrolled", "limited-X", "naive" (Fig. 5, 6)

Once enumerated, RNN architectures define a new *input* domain with a total size that will likely fall in a range anywhere between 100 and 10 thousand (which must still be quite manageable). Fig. 9 shows a new RNN (dubbed "super RNN"), with an **input that consists of the** (vectored) **outputs** of the RNNs enumerated above (due to the space constraints, the picture shows only three of those):

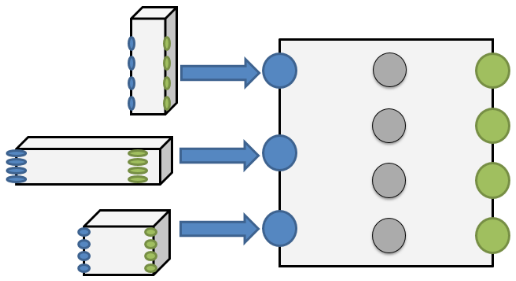


Fig. 9. Super RNN

Question is, which super architecture would be optimal to perform the search?

To be continued...