The Computer **Nonsense** Guide

The Fall of Church-Turing 01.09.2018

Abstract

"My aim is: to teach you to pass from disguised nonsense to something that is patent nonsense." — Ludwig Wittgenstein

This guide is product of the efforts of many people too numerous to list here and of the unique environment of the Space Beam community.

The software environment and operating-system-like parts contain many things which are still in a state of flux. This work confines itself primarily to the stabler parts of the system, and does not address the window system, user interface or application programming interfaces at all.

We are an open-source research & development community that conducts multidisciplinary work on distributed systems, artificial intelligence and high-performance computing.

Our Mission: is provide tools inside a simple workspace for play, work and science through observation and action.

Our Goal: a distributed AI toolkit and workspace environment for machines of all ages!

We make a custom Debian workspace that anyone can use today, these things work together with native support for Python 3, LuaLang and BEAM ecosystems.

Core ideas

- Functions are a form of objects.
- Message passing and function calling are analogous.
- Asynchronous message passing is necessary for non-blocking systems.
- Selective receive allow to ignore messages uninteresting now.

Prerequisites

It is assumed that the reader has done some programming and is familiar with data types and programming language syntax.

Introduction

"An object is really a function that has no name and that gets its argument a message and then look at that message and decide what to do next." — Richard P. Gabriel

About 100 years ago there were two schools of thought a clash between two paradigms for how to make an intelligent system, one paradigm was mathematical logic if I give you some true premises and some valid rules of inference you can derive some truth conclusions and people who believe in logic thought that's the way the mind must work and somehow the mind is using some funny kind of logic that can cope with the paradox of the liar or the fact that sometimes you discover things you believed were false.

Classical logic has problems with that and the paradigm said we have these symbolic expressions in our head and we have rules from repairing them and the essence of intelligence is reasoning and it works by moving around symbols in symbolic expressions.

There was a completely different paradigm that wasn't called artificial intelligence it was called neural networks that said we known about an intelligent system it's the mammalian brain and the way that works is you have lots of little processes with lots of connections between them and you change the strengths of the connections and that's how you learn things so they thought the essence of intelligence was learning and in particular how you change the connection strengths so that your neural network will do new things and they would argue that everything you know comes from changing those connection strengths and those changes have to somehow be driven by data you're not programmed you somehow absorb information from data, well for 100 years this battle has gone on and fortunately today we can tell you recently it was won.

Lua in Erlang

"Scripting is a relevant technique for any programmer's toolbox." — Roberto Ierusalimschy

Luerl is an implementation of standard Lua 5.2 written in Erlang/OTP.

Lua is a powerful, efficient, lightweight, embeddable scripting language common in games, IoT devices, machine learning and scientific computing research.

It supports procedural, object-oriented, functional, data-driven, reactive, organizational programming and data description.

Being an extension language, Lua has no notion of a "main" program: it works as a library embedded in a host. The host program can invoke functions to execute a piece of Lua code, can write and read Lua variables, and call Erlang functions by Lua code.

Luerl is a library, written in clean Erlang/OTP. For more information, check out the get started tutorial. You may want to browse the examples source code.

Luerl goal

A proper implementation of the Lua language

- It SHOULD look and behave the same as Lua 5.2
- It SHOULD include the Lua standard libraries
- It MUST interface well with Erlang

Embedded language

Lua is an embeddable language implemented as a library that offers a clear API for applications inside a register-based virtual machine.

This ability to be used as a library to extend an application is what makes Lua an extension language.

At the same time, a program that uses Lua can register new functions in the Luerl environment; such functions are implemented in Erlang (or another language) and can add facilities that cannot be written directly in Lua. This is what makes any Lua implementation an extensible language.

These two views of Lua (as extension language and as extensible language) correspond of two kinds of interaction between Erlang and Lua. In the first kind, Erlang has the control and Lua is the library. The Erlang code in this kind of interaction is what we call application code.

In the second kind, Lua has the control and Erlang is the library. Here, the Erlang code is called library code. Both application code and library code use the same API to communicate with Lua, the so called Luerl API.

Modules, Object Oriented programming and iterators need no extra features in the Lua API. They are all done with standard mechanisms for tables and first-class functions with lexical scope.

Exception handling and code load go the opposite way: primitives in the API are exported to Lua from the base system C, JIT, BEAM.

Lua implementations are based on the idea of closures, a closure represents the code of a function plus the environment where the function was defined.

Like with tables, Luerl itself uses functions for several important constructs in the language. The use of constructors based on functions helps to make the API simple and general.

The result

Luerl is a native Erlang implementation of standard Lua 5.2 written for the BEAM ecosystem.

- Easy for Erlang to call
- Easy for Lua to call Erlang
- Erlang concurrency model and error handling

Through the use of the BEAM languages, Luerl can be augmented to cope with a wide range of different domains, creating a customized language sharing a syntactical framework.

Lisp Flavoured Erlang

"Lisp: Good News Bad News How to Win Big ™." — Richard P. Gabriel

LFE is a proper Lisp based on the features and limitations of Erlang's virtual machine, attuned to vanilla Erlang and OTP it coexists seamlessly with the rest of the BEAM ecosystem.

Some history: Robert Virding tried Lisp 1 but it didn't really work, Lisp 2 fits the BEAM better so LFE is Lisp 2+, or rather Lisp 3?

For all of us in general the bad news is that almost everything that what we have been using is WORNG! no one can denied the respect that Black Mesa Research deserves but even with it's limited concurrency implemented by a CSP model with monads or static types have yet it classical boundaries; the λ-calculus low concurrent ceiling. [T. Church]

We were not that into Lisp until reading some tweets from certain no-horn viking, the short story is that Scheme is Lisp 2, the goal of Scheme was to implement a Lisp following the actor model but they discover closures instead got hyped with them and forget about Hewitt.

Erlang was born to the world on Stockholm Sweden in 1998, Jane Walerud, Bjarne Däcker, Mike Williams, Joe Armstrong and Robert Virding open-source a language that implement this model of universal computation based in physics without even know or care much about it, just pure engineering powers and a great problem to solve.

It's a language out of a language out of Sweden that can be used to build web scale, asynchronous, non-blocking, event driven, message passing, NoSQL, reliable, highly available, high performance, real time, clusterable, bad ass, rock star, get the girls, get the boys, impress your mom, impress your cat, be the hero of your dog, AI applications.

It's Lisp, you can blast it in the face with a shotgun and it keeps on coming.

LFE goal

An efficient implementation of a "proper" Lisp on the BEAM with seamless integration for the Erlang/OTP ecosystem.

The result

A New Skin for the Old Ceremony where the thickness of the skin affects how efficiently the new language can be implemented and how seamlessly it can interact.

Why Lisp 3?

"Lisp is the greatest single programming language ever designed." — Alan Kay

A lot has changed since 1958, even for Lisp it now has even more to offer.

- It's a programmable programming language
- As such, it's excellent language for exploratory programming.
- Due to it's venerable age, there is an enormous corpus of code and ideas to draw from.

Overall, the evolution of Lisp has been, guided more by institutional rivalry, one-upmanship, and the glee born of technical cleverness characteristic of the hacker culture than by sober assessment of technical requirements.

Lisp 1

Early thoughts about a language that eventually became Lisp started in 1956 when John McCarty attended the Dartmouth Summer Research Project on Artificial Intelligence.

The original idea was to produce a compiler, but in the 50's this was considered a major undertaking, and McCarthy and his team needed some experimenting in order to get good conventions for subroutine linking, stack handling and erasure.

They started by hand-compiling various functions into assembly language and writing subroutines to provide a LISP environment.

They decided on garbage collection in which storage is abandoned until the free storage list is exhausted, the storage accessible from program variables and the stack is marked, so the unmarked storage is made into a new free storage list.

At the time was also decided to use SAVE and UNSAVE routines that use a single contiguous public stack array to save the values of variables and subroutine return addresses in the implementation of recursive subroutines.

Another decision was to give up the prefix and tag parts of the message, this left us with a single type an 15 bit address, so that the language didn't require declarations.

These simplifications made Lisp into a way of describing computable functions much neater than the Turing machines or the general recursive definitions used in recursive function theory.

The fact that Turing machines constitute an awkward programming language doesn't much bother recursive function theorists, because they almost never have any reason to write particular recursive definitions since the theory concerns recursive functions in general.

Another way to show that Lisp was neater than Turing machines was to write a universal LISP function and show that it is briefer and more comprehensible than the description of a universal Turing Machine.

This refers to the Lisp function `eval(e,a)` which computes the value of a Lisp expression `e`, the second argument `a` being a list of assignments of values to variables, `a` is needed to make the recursion work.

Lisp 2

The Lisp 2 project was a concerted language that represented a radical departure from Lisp 1.5.

In contrast to most languages in which the language is first designed and then implemented Lisp 2 was an implementation in search of a language, in retrospect we can point out that was searching from one out of Sweden.

The earliest known LISP 2 document is a one-page agenda for a Lisp 2 Specifications Conference held by the Artificial Intelligence Group at Standford. Section 2 of this agenda was:

Proposals for Lisp 2.0

- Linear Free Storage
- Numbers and other full words
- Auxiliary Storage
- Input language, infix notation.
- Arrays
- Freer output format
- Sequence of implementation
- Comments

- Documentation and maintenance
- Hash Coding
- Sobroutine linkage
- Storage conventions
- Effect of various I/O apparatus
- Interaction with programs in other languages
- Expressions having property lists

The Actor Model

Actors are the universal primitive of concurrent digital computation. In response to a message that it receives, an actor can make local decisions, create more Actors, send more messages, and designate how to respond to the next message received.

Unbounded nondeterminism is the property that the amount of delay in servicing a request can become unbounded as a result of arbitration of contention for shared resources while still guaranteeing that the request will eventually be serviced.

Arguments for unbounded nondeterminism include the following:

There is no bound that can be placed on how long it takes a computational circuit called an Arbiter to settle.

- Arbiters are used in computers to deal with the circumstance that computer clocks operate asychoronously with input from outside, "e.g, keyboard input, disk access, network input, etc..."
- So it could take an unbounded time for a message to sent to a computer to be received and in the meantime the computer could traverse an unbounded number of states.

The following were the main influences on the development of the actor model of computation:

- The suggestion by Alan Kay that procedural embedding be extended to cover data structures in the context of our previous attempts to generalize the work by Church, Landin, Evans, and Reynolds on "functional data structures."
- The context of our previous attempts to clean up and generalize the work on coroutine control structures of Landin, Mitchell, Krutar, Balzer, Reynolds, Bobrow-Wegbreit, and Sussman.
- The influence of Seymour Papert's "little man" metaphor for computation in LOGO.
- The limitations and complexities of capability-based protection schemes. Every actor transmission is in effect an inter-domain call efficiently providing an intrinsic protection on actor machines.
- The experience developing previous generations of PLANNER. Essentially the whole PLANNER-71 language (together with some extensions) was implemented by Julian Davies in POP-2 at the University of Edinburgh.

In terms of the actor model of computation, control structure is simply a pattern of passing messages.

We have quoted Hewitt at length because the passage illustrates the many connections among different ideas floating around in the AI, Lisp, and other programming language communities; and because this particular point in the evolution of ideas represented a distillation that soon fed back quickly and powerfully into the evolution of Lisp itself.

Logic and λ-calculus

Logic programming is the proposal to implement systems using mathematical logic.

Perhaps the first published proposal to use mathematical logic for programming was John McCarthy's Advice Taker paper.

Planner was the first language to feature "procedural plans" that were called by "pattern-directed invocation" using "goals" and "assertions". A subset called Micro Planner was implemented by Gerry Sussman, Eugene Chariak and Terry Winograd and was used in Winograd's natural language understanding program SHRDLU, and some other projects.

This generated a great deal of excitement in the field of AI. It also generated controversy because it proposed an alternative to the logic approach one of the mainstay paradigms for AI.

The upshot is that the procedural approach has a different mathematical semantics based on the denotation semantics of the Actor model from the semantics of mathematical logic.

There were some surprising results from this research including that mathematical logic is incapable of implementing general concurrent computation even though it can implement sequential computation and some kinds of parallel computation including the lambda calculus.

Classical logic blows up in the face of inconsistent information that is kind of ubiquitous with the growth of the internet.

This change enables a new generation of systems that incorporate ideas from mathematical logic in their implementation, resulting on some reincarnation of logic programming. But something is often transformed when reincarnated!

A limitation of logic programming

In his 1988 paper on early history of Prolog, Bob Kowalski published the thesis that "computation is controlled deduction" which he attributed to Pat Hayes.

Contrary to Kowalski and Hayes, Hewitt's thesis was that logical deduction was incapable of carrying out concurrent computation in open systems because of indeterminacy in the arrival order of messages.

Indeterminacy in concurrent computation

Hewitt and Agha [1991] argued that: The Actor model makes use of arbitration for determining which message is next in the arrival ordering of an Actor that is sent multiple messages concurrently.

For example Arbiters can be used in the implementation of the arrival ordering of an Actor which is subject to physical indeterminacy in the arrival order.

In concrete terms for Actor systems typically we cannot observe the details by which the arrival order of messages for an Actor is determined. Attempting to do so affects the results and can even push the indeterminacy elsewhere.

Instead of observing the internals of arbitration processes of Actor computations, we await outcomes.

Physical indeterminacy in arbiters produces indeterminacy in Actors. The reason that we await outcomes is that we have no alternative because of indeterminacy.

According to Chris Fuchs [2004], quantum physics is a theory whose terms refer predominately to our interface with the world. It is a theory not about observables, not beables, but about 'dingables' we tap a bell with our gentle touch and listen for its beautiful ring.

It is important to distinguish between indeterminacy in which factors outside the control of an information system are making decision and *choice* in which the information system has some control.

It is not sufficient to say that indeterminacy in Actor systems is due to unknown/unmodeled properties of the network infrastructure. The whole point of the appeal to quantum indeterminacy is to show that aspects of Actor systems can be unknowable and the participants can be entangled.

The concept that quantum mechanics forces us to give up is: the description of a system independent from the observer providing such a description; that is the concept of the absolute state of a system. I.e, there is no observer independent data at all.

According to Zurek [1982], "Properties of quantum systems have no absolute meaning. Rather they must always characterized with respect to other physical systems."

Does this mean that there is no relation whatsoever between views of different observers? Certainly not. According to Rovelli [1996] "It is possible to compare different views, but the process of comparison is always a physical interaction (and all physical interactions are quantum mechanical in nature)".

Lisp 3

Lisp Flavored Erlang (LFE) is a functional, concurrent, general-purpose programming language and Lisp dialect built on top of Core Erlang and the Erlang Virtual Machine (BEAM).

What isn't

- It isn't an implementation of Maclisp
- It isn't an implementation of Scheme
- It isn't an implementation of Common Lisp
- It isn't an implementation of Clojure

What is

- LFE is a proper Lisp based on the features and limitations of the Erlang VM (BEAM).
- LFE coexists seamlessly with vanilla Erlang/OTP and the rest of the BEAM ecosystem.

- LFE runs on the standard Erlang Virtual Machine (BEAM).

The object-oriented programming style used in the Smalltalk and Actor families of languages is available in LFE and used by the Monteverde HPC package system. Its purpose is to perform generic operations on objects. Part of its implementation is simply a convention in procedural-calling style: part is a powerful language feature, called flavors, for defining abstract objects.

Lisp Machine flavors

When writing a program, it is often convenient to model what the program does in term of objects, conceptual entities that can be likened to real-world things.

Choosing what objects to provide in a program is very important to the proper organization of the program.

In an object-oriented design, specifying what objects exist is the first task in designing the system.

In an electrical design system, the objects might be "resistors", "capacitors", "transistors", "wires", and "display windows".

After specifying what objects there are, the next task of the design is to figure out what operations can be performed on each object.

In this model, we think of the program as being built around a set of objects, each of which has a set of operations that can be performed on it.

More rigorously, the program defines several types of object, and it can create many instances of each type.

The program defines a set of types of object and, for each type, a set of operations that can be performed on any object of that type.

The new types may exist only in the programmer's mind. For example, it is possible to think of a disembodied property list as an abstract data type on which certain operations such as `get` and `put` are defined.

This type can be instantiated by evaluating this form you can create a new disembodied property lists are really implemented as lists, indistinguishable from any other lists, does not invalidate this point of view.

However, such conceptual data types cannot be distinguished automatically be the system; one cannot ask "is this object a disembodied property list, as opposed to an ordinary list".

We represent our conceptual object by one structure.

The LFE flavors we use for the representation has structure and refers to other Lisp objects.

The object keeps track of an internal state which can be examined and altered by the operations available for that type of object, get examines the state of a property list, and put alters it.

We have seen the essence of object-oriented programming. A conceptual object is modeled by a single Lisp object, which bundles up some state information. For every type there is a set of operations that can be performed to examine or alter the object state.

Application containers

"Awaken my child, and embrace the glory that is your birthright." — The Overmind

Singularity: Application containers for Linux enables full control of the environment on whatever host you are on. This include distributed systems, your favorite blockchain, HPC centers, microservices, GPU's, IoT devices, docker containers and the whole computing enchilada.

Containers are used to package entire scientific workflows, software libraries, and datasets.

Did you already invest in Docker? The Singularity software can import your Docker images without having Docker installed or being a superuser.

As the user, you are in control of the extent to which your container interacts with its host. There can be seamless integration, or little to no communication at all.

- Reproducible software stacks: These must be easible verifiable via cheksum or cryptographic signature in such a manner that does not change formats. By default Singularity uses a container image file which can be checksummed, signed and easily verified.
- Mobility of compute: Singularity must be able to transfer (and store) containers in a manner that works with stadandard data mobility tools and protocols.
- Compatibility with complicated architectures: The runtime must be compatible with existing HPC, scientific, compute farm and enterprise architectures maybe running legacy vintage systems which do not support advanced namespace features.

Linux containers

A Unix operating system is broken into two primary components, the kernel space, and the user space. The kernel supports the user space by interfacing with the hardware, providing core system features and creating the software compatible layers for the user space. The user space on the other hand is the environment that most people are most familiar with interfacing with. It is where applications, libraries and system services run.

Containers shift the emphasis away from the run-time environment by commoditizing the user space into swappable components. This means that the entire user space portion of a Linux operating system, including programs, custom configuration, and environment can be interchangeable at run-time.

Software developers can now build their stack onto whatever operating system base fits their needs bets, and create distributable run-time encapsulated environments and the users never have to worry about dependencies, requirements, or anything else from the user space.

Singularity provides the functionality of a virtual machine, without the heavyweight implementation and performance costs of emulation and redundancy!

Container instances

Singularity "container instances" allow you to run services (e.g. Nginx, Riak, PostgreSQL, etc...) a container instance, simply put, is a persistent and isolated version of the container image that runs in the background.

Important notes

The instances are linked with your user. So if you start an instance with sudo, that is going to go under root, you will need to call `sudo singularity instance.list` in order to see it.

Live for the swarm!

"Send colonies to one or two places, which may be as keys to that state, for it is necessary either to do this or else to keep there a great number of lings and hydras." — Sarah Kerrigan

We present Blueberry a TorchCraft bot system build for online competition and AI research on the real-time strategy game of StarCraft; ours is a message-passing, asynchronous system that exploits the hot swap loading and parallelism of Luerl and the properties of the BEAM ecosystem.

StarCraft serve as an interesting domain for Artificial Intelligence (AI), since represent a well defined complex adversarial environment which pose a number of interesting challenges in areas of information gathering, planning, dealing with uncertainty, domain knowledge exploitation, task decomposition, spatial reasoning, and machine learning research.

Unlike synchronous turn-based games like chess and go, StarCraft games are played in real-time, the state continue to progress even if no action is taken, actions must decided in fractions of a second, game frames issue simultaneous actions to hundreds of units at any given time, players only get the information about what they units observe, there is a fog of information present in the environment and hidden units that require additional detection.

Core ideas

- StarCraft is about information, gas and minerals.
- Strong units vs mobile units.
- Defense units are powerful but immobile, offense units are mobile but weak.
- Efficiency is not the number one goal.

Stages of a game

- Early, Make/defend a play & send colonies to one or two bases.
- Middle, Core units, make/defend pressure & take a base.
- Late, Matured core units, multi-pronged tactics & take many bases.
- Final, The watcher observes, the fog collapses an event resolves.

Information, colonies, improved economy for better tools.

What is an organization?

"ORGs is a paradigm in which people are tightly integrated with information technology that enables them to function with an organizationally relevant task or problem." — Carl Hewitt

A monkey, a building, a drone: each is a concrete object and can be easily identified. One difficulty attending the study of organizations is that an organization is not as readily visible or describable.

Exactly what is an organization such as a business concern? It is a building? A collection of machinery? A legal document containing a statement of incorporation? It is hardly likely to be any of these by itself. Rather, to describe an organization requires the consideration of a number of properties it possesses, thus gradually making clear, or at least clearer, that it is.

The purposes of the organization, whether it is formal or informal, are accomplished by a collection of members whose efforts or to use a term to be employed throughout this work, behavior are so directed that they become coordinated and integrated in order to attain sub-goals and objectives.

Perception and behavior

All of us at some point or another have had the experience of watching another person do something or behave in a certain way, saying to ourselves, "She/he acts as if she/he thought, ... " and then filling in some supposition about the way the other person looked at things.

Simple as the statement "He acts as if he thought ... " may be, it illustrates two important points.

First, what the person thinks he sees may not actually exist. They could act as if changes in methods as an attempt by management to exploit them.

As long as they had this attitude or belief, any action by management to change any work method would be met, at the very least, with suspicion and probably with hostility.

The second point is that people act on the basis of what they see. In understanding behavior, we must recognize that facts people do not perceive as meaningful usually will not influence their behavior, whereas the things they believe to be real, even though factually incorrect or nonexistent, will influence it.

Organizations are intended to bring about integrated behavior. Similar, or at least compatible, perceptions on the part of organizational members are therefore a matter of prime consideration.

Clues

One of the first things we must recognize is that in learning about things we not only learn what they are, that is, that the round white object is for football, but we also learn what these things mean, that is, football is a sport that the USA men's team don't get and their woman counterpart have master perfectly.

Upon receiving a signal (sight of football) we perform an interpretative step by which a meaning is attached to it.

Many of these "meanings" are so common and fundamental in our understanding of the world that we fail to note them except under unusual circumstances.

One way these meanings are brought home to us is by meeting people from countries different from our own; many of the meanings which things have come from our culture, they are things all people within the culture share.

These common interpretations of things help enormously in communicating, but they sometimes make it difficult to set factors in perspective so that we can really understand the reasons for behavior.

Threshold of perception

We all, have certain things (stimuli) to which we are sensitized and that when these appear we are instantly alert and eager to examine them.

There are other stimuli of relative unimportant to us to which we do not pay as much attention and may, in effect, actually block out.

One way of viewing this subject is to suggest that we have thresholds or barriers which regulate what information from the outside world reaches our consciousness.

On some matters the barriers are high and we remain oblivious to them, but on others which are quite important to us we are sensitized and, in effect, we lower the barrier, permitting all the information possible concerning these matters to reach our consciousness.

Resonance

Related to this idea of sensitivity and selectivity is a phenomenon that might be called resonance.

Through experience and what we see ourselves to be, the understanding of a particular item of information may be very similar to that of others.

It is explained this way: since all the people inside a group look upon themselves as peers, they know what a change on the individual means in annoyance and inconvenience.

They can easily put themselves into his shows and, once having done so, probably feel almost as disturbed as he might be.

Internal consistency

One property of the images formed of the world around us is that they are reasonable, or internally consistent.

For instance, we may look at some draw on a page and see a rabbit. One portion along these lines might suggest a duck, but we do not have an image of something half rabbit and half duck.

In fact, if our first impression is of a duck, we may never notice that a portion looks like a rabbit.

We seem to tune out the elements that do not fit.

Dealing with conflict

Organizations that posses the capacity to deal adequately with conflict have been described as follows:

1. They posses the machinery to deal constructively with conflict. They have an structure which facilitates constructive interaction between individuals and work groups.
2. The personnel of the organization is skilled in the process of effective interaction and mutual influence (skills in group ledership and membership roles in group building and maintenance functions).
3. There is a high confidence and trust in one another among members of the organization, loyalty to the work group and to the organization, and high motivation to achieve the organization's objectives.

Confidence, loyalty, and cooperative motivation produce earnest, sincere, and determined efforts to find solutions to conflict. There is greater motivation to find constructive solution that to maintain an irreconcilable conflict. The solutions reached are often highly creative and represent a far better solution than any initially proposed by the conflicting interests.

The essence here is that out of conflict will come a new synthesis superior to what existed before and perhaps superior to any individual point of view existent in conflict.

Conflict, resting in part on different perspectives of what "ought" to be, is one of the avenues for opening new directions for the organization or one of the ways of moving in new directions. This is not only useful but also vital for organizational survival. The question, therefore, as we view conflict is not, "How to eliminate it?" but, "Is it conflict of such a type and within circumstances where it will contribute to rather than detract from organizational interest?"

Whether a conflict is good or bad for an organization, whether a conflict can be made useful for an organization, depends not so much on manipulating the conflict itself as on the underlying conditions of the overall organization. In this sense, conflict can be seen as;

1. a symptom of more basic problems which requires attention
2. an intervening variable in the overall organization to be considered, used, and maintained within certain useful boundaries.

Organizational adaptation frequently proceeds through a new arrangement developing informally, which, after proving its worth and becoming accepted, is formally adopted. The first informal development, however, may be contrary to previously established procedures and in a sense a violation or a subversion of them; or the informal procedures may be an extension of a function for internal political purposes.

Programmed links

If the process had given instruction to report immediately on completion of the task, this instruction facilitates linking the completed act with the next one. Through an information transfer, we call this a programmed link.

The supervisor node, of course may detect that something is wrong through another control cycle. It can then take corrective action by including or adding into this programmed link or perhaps by attacking on the more difficult problem of human apathetic attitudes and motivation his units could display.

Progression of goals

Organizations have progression of goals which result from a division of work.

A subdivide goal becomes the task of a process contained within a specialized organizational unit.

This nesting of goals is contained as part of the core organizational means-ends chain.

Needless to say, the hierarchy of control loops which are connected with the progression of goals may be handle in number of ways, regardless of how the elements are allocated, the important factor is that all elements must be provided for in some way. Hence, our model supplies an extremely useful tool in analyzing complex control situations by telling us what basic functions bust occur and in what sequence, even though initially we have no idea as to where or how they are executed in the organization.

Goals and feedback

The feedback loop containing information about organizational performance and conditions leads to definition of subunit goals or standards. It's important to show how a situation in one area could lead to modifications in a number of organizational units at higher levels.

This even result in reformulating the basic goals of organizations. Feedback is essential to adequate goal formation.

Iterative programming

"Programming is an iterative process, *iterative* is another name for intelligent trial and error."
— Michael C Williams

In cybernetics, the word "trial" usually implies random-or-arbitrary, without any deliberate choice.

Programming is an iterative process with a large amount of trial and error to find out:

- What needs to be implemented,
- Why does it need to be implemented,
- How should be implemented.

Trial and error is also a heuristic method of problem solving, repair, tuning, or obtaining knowledge.

In the field of computer science, the method is called generate and test. In elementary algebra, when solving equations, it is "guess and check".

Cumulative adaptation

The existence of different available strategies allows us to consider a separate superior domain of processing, a "meta-level" above the mechanics of switch handling from where the various available strategies can be randomly chosen.

Suppose N events each have a probability *p* of success, and the probabilities are independent. An example would occur if N wheels bore letters A and B on the rim, with A's occupying the spun and allowed to come to rest; those that stop at an A count as successes. Let us compare three ways of compounding these minor successes to a Grand Success, which we assume, occurs only when every wheel is stopped at an A.

Case 1: All N wheels are spun; if all show an A, Success is recorded and the trials ended; otherwise all are spun again, and so on till 'all A's' come up at one spin.

Case 2: The first wheel is spun; if it stops at an A it is left there; otherwise it is spun again. When it eventually stops at an A the second wheel is spun similarly; and so on down the line of N wheels, one at a time, till all show A's.

Case 3: All N wheels are spun; those that show an A are left to continue showing it, and those that show a B are spun again. When further A's occur they also are left alone. So the number spun gets fewer and fewer, until all are at A's.

The conclusion that Case 1 is very different from Cases 2 and 3, does not depend closely on the particular values of p and N.

Comparison of the three Cases soon shows why Cases 2 and 3 can arrive at Success so much sooner than Case 1: they can benefit by partial successes, which 1 cannot. Suppose, for instance, that, under Case 1, a spin gave 999 A's and 1 B. This is very near complete Success; yet it counts for nothing, and all the A's have to be thrown back into the melting-pot. In Case 3, however, only one wheel would remain to be spun; while Case 2 would perhaps get a good run of A's at the left-hand end and could thus benefit from it.

The examples show the great, the very great, reduction in time taken that occurs when the final Success can be reached by stages, in which partial successes can be conserved and accumulated.

We can draw, then, the following conclusion. A compound event that is impossible if the components have to occur simultaneously may be readily achievable if they can occur in sequence or independently.

Dynamic optimization

Dynamic optimization (also known as dynamic programming) is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed result, saving computation time at the expense of (it is hoped) a modest expenditure in storage space.

Dynamic programming is both a mathematical optimization method and a computer programming method. In both contexts it refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner.

If sub-problems can be nested recursively inside larger problems, so that dynamic programming methods are applicable, then there is a relation between the value of the larger problem and the values of the sub-problems. In the optimization literature this is called the Bellman equation.

Overlapping subproblems

Like Divide and Conquer, Dynamic optimization combines solutions to sub-problems mainly used when they are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to recomputed. So Dynamic Programming is not useful when there are no common subproblems because there is no point storing the solutions if they are not needed again.

Optimal substructure

A problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems. This property is used to determine the usefulness of dynamic programming.

The application of dynamic programming is based on the idea that in order to solve a dynamic optimization problem from some starting period t to some ending period T, one implicitly has to solve subproblems starting from later dates s, where $t<s<T$. This is an example of optimal substructure. The Principle of Optimality is used to derive the Bellman equation, which shows how the value of the problem starting from t is related to the value of the problem starting from s.

Bellman's principle

The dynamic programming method breaks this decision problem into smaller subproblems.

In the context of dynamic game theory, this principle is analogous to the concept of subgame perfect equilibrium, although what constitutes an optimal policy in this case is conditioned on the decision-maker's opponents choosing similarly optimal policies from their points of view.

Analytical concepts

To understand the Bellman's principle, several underlying concepts must be understood. First, any optimization problem has some objective: minimizing travel time, minimizing cost, maximizing profits, maximizing utility, etcetera.

The mathematical function that describes this objective is called the objective function.

Dynamic programming breaks a multi-period planning problem into simpler steps at different points in time. Therefore, it requires keeping track of how the decision situation is evolving over time. The information about the current event that is needed to make a correct decision is called the "state".

Process of abstraction

In trying to understand what is happening around us we are faced with a fundamental problem. In approaching any situation, the system trying to understand it, does not attempt to gather all information. Instead it selects certain facts and searchers for others.

This selection of some items and ignoring of others is a process of abstraction.

It is the abstracting form a real or if you will empirical situation the things seemingly most important to deal with.

In this process of abstraction and model building we deliberately select a few items, ignore may others, and then place the items chosen in a particular relationship to one another.

In doing so we are intentionally ignoring facts or relationships that can influence the type of situation under study.

The problem it to select the most meaningful elements and relationships and dropout the rest.

Those who use abstraction skillfully know well that they neither have all the facts nor have considered all the relationships bearing on the outcome of what they are analysing.

We do not use the abstractions from one situation in another setting without carefully examining the fit. Neither do we expect a model to handle all aspects of a situation.

We shall be dealing with many abstractions and models, not with the intention of exactly mirroring the real world but with the objective of clarifying our perception of its most essential features.

Abstractions and models are mechanisms for economizing both time and effort, but like any tool they must be used within their limits.

Model your goals

Taking the abstracted elements, a character with the flat tire begins to connect them into a pattern.

Better yet, he weaves them into a model of the confronting situation, which we can use both to understand his plight and figure out what to do about it.

The parts of this model would probably include, among other things, the flat tire, the image of the spare in the trunk, the telephone, the service station, a forthcoming business meting, etc.

A second model would contain the telephone, the service station, and the repairman there.

Finally, it concludes that it will call a cab and leave his wife to deal with the flat tire as best as she can.

These are extraordinarily elementary models, but they serve a very practical purpose.

With them the main character in our illustration can see the likely consequences of various courses of action.

We can find out these things by doing them directly by actually handling the tire and observing that we get dirty, or by calling the repairmen and waiting for him and learning that it takes too long.

In the age of big data; big models are good.

- For any given size of data, the bigger the model, the better it generalizes, provided you regularize well.
- This is obviously true if your model is an ensemble of smaller models.
- Adding extra models to the ensemble always helps.
- It is a good idea to try to make the data look small by using a big model.

By using the model, however, we can make some reasonable predictions about what will occur and therby accept or reject the choices open to us.

Predictive modeling

Our emphasis is on people in organizations where managers have to make decisions even through their knowledge and concepts may be wrong or inadequate.

As generalizations are made about what takes place in organizations, we shall often be talking about the decision processes that organizational members engage in or the sequence of factors which lead to a particular behavior.

We talk about the possible actions a person may take on recepipt of an order from his superior, which he thinks is improper and unwarranted.

We talk about choossing amount the courses of accepting the order competly, protesting, protesting and raiisng an alternative, or leavng the organization.

It sometimes appers as if there were a basic assumption that people consicosly bring our all these possibilities and rationally weigh the pros and cons of each.

Does the individual at times carry on the same process unconsicusly that on other occasions he perhaps conducts consciously? We hardly know...

We are faced with the fact that people sometimes do things and later say, I never thougth that I would act like that under those circumstances.

In saying this the individual indicates that he saw other opportunities and that, through some process unobservable to him he decided among them and chose on that came from elsewhere that his own conscious thought.

The distinction between conscious and unconscious thought are by no means easy to determine, and for our purposes, it is not usually necessary to make them.

Considering the development of the field of Artificial Intelligence at the moment, it seems reasonable to conduct our analysis at the level at which both machines and humans do make decisions, without taking into account whether the choices and decisions processes are conscious or not.

Several references have been made with the intent of this guide to provide conceptual tools for analysis. As with any other tool models, abstractions and generalizations are useful only when within their limitations.

Why Erlang helps?

"Any sufficiently complicated concurrent program in another language contains an ad hoc informally-specified bug-ridden slow implementation of half of Erlang." — Robert Virding

Erlang suits iterative development perfectly, the ecosystem offers a variety of languages with different focus all build in top of the BEAM and the OTP framework.

Let it crash!

Robust systems must always be aware of errors but avoid the need of error checking code everywhere.
We want to be able to handle processes crashes among cooperative processes.

- If one process crashes all cooperating processes should crash
- Cooperating processes are linked together
- Process crashes propagate along links

System processes can monitor them and rest them when necessary but sometimes we do need to handle errors locally.

Pattern matching

Functions use pattern matching to select clauses, this is a BIG WIN™

Supervision trees

Too often, developers try to implement their own error-handling and recovery strategies in their code, with the result that they increase the complexity of the code and the cost of maintaining it, how many times have you seen catch statements with nothing more than TODO comments to remind some future, better smarted developer to finish the job on the error handling.

This is where the supervisor process makes it entrance. It takes over the responsibility for the unexpected-error-handling and recovery strategies from the developer.

The behavior, in a deterministic and consistent manner, handles monitoring, restart strategies, race conditions, and borderline cases most developers would not think of.

A supervisor has a standard set of interface functions and include functionality for tracing and error reporting. Supervisors are used to build a hierarchical process structure called a supervision tree, a nice way to structure a fault-tolerant application.

- Supervisors will monitor their processes through links and trapping exists.
- Supervisors can restart the workers when they terminate.

On production, this usually means a fairly straight-forward combination of external process management, overload monitoring and proxying.

A supervisor is responsible for starting, stopping, and monitoring external processes. The basic idea of a supervisor is that it is to keep its processes alive by restarting them when necessary.

Fault-tolerance

Fault-tolerance is achieved by creating supervision trees, where the supervisors are the nodes and the workers are the leaves of this analogy. Supervisors on a particular level monitor and handle children in the subtrees they have started.

If any worker terminates abnormally, the simple supervisor immediately restart it. If the process instead terminate normally, they are removed from the supervision tree and no further action is taken.

Stopping the supervisor results in all the processes in the tree being unconditionally terminated. When the supervisor terminates, the run-time ensures that all processes linked to it receive an EXIT signal.

It is a valid assumption that nothing abnormal should happen when starting your system. If a supervisor is unable to correctly start a process, it terminates all of its processes and aborts the startup procedure. While we are all for a resilient system that tries to recover from errors, startup failures is where we draw the line.

BEAM virtual machine

"Links were invented by Mike Williams and based on the idea of a *C-wire* a form of electrical circuit breaker." — Joe Armstrong

The virtual machine runs as one OS process. By default it runs one OS thread per core to achieve maximum utilization of the machine. The number of threads and on which cores they run can be set when the BEAM is started.

Erlang processes are implemented entirely by the VM and have no connection to either OS processes or OS threads. So even if you are running a BEAM system of over one million processes it is still only one OS process and one thread per core, in this sense the BEAM is a "process virtual machine" while the Erlang system itself very much behaves like an OS and Erlang processes have very similar properties to OS processes.

- Process isolation
- Asynchronous communication
- Error handling, introspection and monitoring
- Predefined set of datatypes
- Immutable data
- Pattern matching
- Functional, soft real-time, reactive, message-passing system
- Modules as function containers and the only way of handle code

Inside the BEAM ecosystem, we just worry about receiving messages.

Load balancing

The goal is to not overload any scheduler while using as little CPU as possible.

Compacting the load to fewer schedulers is usually better for memory locality, specially on hyperthreads, the primary process is in charge of balance the rest of the schedulers.

Process stealing

Process stealing is used by artists of all types and computers alike, on the BEAM is the primary mechanism to load balance and spread processes.

- A scheduler with nothing runnable will try to "steal processes" from adjacent schedulers, then next beyond that.
- We only steal from run-queues, never running or suspended processes.
- Schedulers changes on other schedulers run-queues.
- Each scheduler has its own run-queue.
- Processes suspend when waiting for messages, this is NOT a busy wait.
- Suspended processes become runnable when a message arrives.

By this mechanism the BEAM suspend unneeded schedulers. Once every period of 20k function calls is reach a new primary process inside a node scheduler is chosen.

Functions and modules

Modules contain functions, its a flat module space with just functions they only exist in modules there are no dependencies between running modules they can come and go as they please.

Functions

Functions cannot have a variable number of arguments! Erlang/OTP assumes functions with same name but different arities, each function has only a fixed number of arguments.

Modules

Modules can have functions with the same name and different number of arguments (arity), inside the virtual machine they are different functions.

Modules can consist of

- Declarations
- Function definitions
- Macro definitions
- Compile time function definitions

Macros can be defined anywhere, but must be defined before used.

The system only has compile code there is no build-in interpreter just compile code in modules. Everything is in modules the module is the unit of code handling, you compile modules, load modules, delete modules, update modules, everything run though modules there are no living functions outside modules.

We can have multiple versions of modules in the system at the same time, all functions belong to a module, this handle of modules means there is no inter-module dependency of modules at all, they just come and go when the system is running.

In this sense a running BEAM instance has no notion of a system, and can be described more like a collection of running modules.

McIlroy garden hose

"Streams means something different when shouted." — Dennis Ritchie

The pipe location in your home is important for proper maintenance and water flow. Many pipes are located in walls, floors and ceilings and are hard to locate.

One of the most widely admired contributions of Unix to the culture of operating systems and command languages is the pipe, as used in a pipeline of commands.

The fundamental idea was by no means new; the pipeline is merely a specific form of coroutine.

Pipes appeared in Unix in 1972, well after the PDP-11 version of the system was in operation, at the insistence of M.D McIlroy, a long advocate of the non-hierarchical control flow that characterizes coroutines.

Some years before pipes, were implemented, he suggested that commands should be thought of as binary operators, whose left and right operand specified the input and output files. Thus a 'copy' utility would be commanded by `inputfile copy outputfile`.

Multics provided a mechanism by which I/O Streams could be directed through processing modules on the way to (or from) the device or file serving as source or sink.

Thus it might seem that stream-splicing in Multics was the direct precursor of UNIX pipes.

We don't think this is true, or is true only in a weak sense. Not only were coroutines well-known already, but their embodiment as Multics I/O modules required to be specially coded in such a way that they could be used for no other purpose.

The genius of the Unix pipeline is precisely that it is constructed from the very same commands used constantly in simplex fashion.

The mental leap needed to see this possibility and to invent the notation is large indeed.

Coroutines

Are computer-program components that generalize subroutines for non-preemptive multitasking by allowing multiple entry points for suspending and resuming execution at certain locations.

Subroutines

At the same time that assembly languages were being developed, programmers were gaining experience with subroutines.

Subroutines are short programs that perform functions of a general nature that can occur in various types of computation.

A branch sequence of instructions is executed, which jumps the program to the subroutine, the set of instructions in the subroutine is executed using the specified number, and, at completion, the computer goes back to the problem program for its continuation.

A sequence of program instructions that perform a specific task, packaged as a unit. This unit can then be used in programs wherever that particular task should be performed.

The experience with assemblers and subroutines helped to generate the ideas for the next step, that of a higher level language that would require the programmer to understand the problem he wishes to solve and not the machine that will be used to solve it.

Subprograms may be defined within programs, or separately in libraries that can be used by multiple programs.

In different programming languages, a subroutine may be called a procedure, a function, a routine, a method, or a subprogram.

Difference with processes

Processes are independent units of execution instead of a subroutine that lives inside a process.

Cooperative multitasking

Also known as non-preemptive multitasking, is a style of computer multitasking in which the operating system never initiates a context switch from a running process to another process.

Instead, processes voluntarily yield control periodically or when idle in order to enable multiple applications to be run concurrently.

Streams

A stream is a full-duplex connection between a process and a device or another process. It consists of several linearly connected processing modules, and is analogous to a Shell pipeline, except that data flows in both directions.

In essence, the stream I/O provides a framework for making file descriptors act in the standard way most programs already expect, while providing a richer underlying behavior, for handling network protocols, or processing the appropriate messages.

Stream Mechanisms

When things wish to communicate, they must first establish communication. The stream mechanism provide a flexible way for processes to conduct an already-begun conversation with devices and with each other: an existing stream connection is named by a file descriptor, and the usual read, write, and I/O control request apply. Processing modules may be inserted dynamically into a stream connection, so network protocols, terminal processing, and device drivers are independent and separate cleanly.

However, these mechanisms, by themselves, do not provide a general way to create channels between them.

Simple extensions provide new ways of establishing communication. In our system, the traditional UNIX pipe is a cross-connected stream. A generalization of file-system mounting associates a stream with a named file. When the file is opened, operations on the file are operations on the stream. Open files may be passed from one process to another over a pipe.

These low-level mechanisms allow construction of flexible and general routines for connecting local and remote processes.

The work reported on STREAMS describes convenient ways for programs to establish communication with unrelated processes, on the same or different machines.

Unix System V

STREAMS in this framework, a stream is a chain of coroutines that pass messages between a program and a device driver (or between a pair of programs).

An important concept is the ability to push custom code modules which can modify the functionality of a network interface or other device – together to form a stack. Several of these drivers can be chained together in order.

Research Unix 8th

A modified 4.1cBSD with a System V shell and sockets replaced by Streams. Added a network file system that allowed accessing remote computers' files as /n/hostname/path, and a regular expression library that introduced an API later mimicked by Henry Spencer's reimplementation. First version with no assembly in the documentation.

Starting with the 8th Edition, versions of Research Unix had a close relationship to BSD. This began by using 4.1cBSD as the basis for the 8th Edition.

Research Unix 8th Edition started from BSD 4.1c, but with enormous amounts scooped out and replaced by our own stuff. This continued with 9th and 10th. The ordinary user command-set was, a bit more BSD-flavored than SysVish, but it was pretty eclectic.

Why ZeroMQ helps?

"Diversity is not a political slogan. It's the basis for collective intelligence." — Pieter Hintjens

ZeroMQ is a community of projects focused on decentralized message passing. They agree on protocols (RFCs) for connecting to each other and exchanging messages. Messages are blobs of useful data of any reasonable size.

You can use this power to queue, route, and filter messages according to various "patterns."

ZeroMQ (also known as ØMQ, 0MQ, or zmq) looks like an embeddable networking library but acts like a concurrency framework. It gives you sockets that carry atomic messages across various transports like inter-process and TCP.

Multilingual Distributed Messaging thanks to the ZeroMQ Community.

- Carries messages across inproc, IPC, TCP, multicast.
- Smart patterns like pub-sub, push-pull, and request-reply.
- Backed by a large and active open source community.

Its asynchronous I/O model gives you scalable multicore applications, built as asynchronous message-processing subroutines. Read the guide and learn the basics.

Protocols

The ZeroMQ protocols live as RFCs on http://rfc.zeromq.org. The main one would be RFC 23, the ZeroMQ Message Transport Protocol (ZMTP), which describes how two pieces talk, over TCP or IPC. RFC 23 provides backwards compatibility to all older stable releases of ZeroMQ.

ZMTP defines rules for backward interoperability, extensible security mechanisms, command and message framing, connection metadata, and other transport-level functionality.

Message patterns

Let's recap briefly what ZeroMQ does for you. It delivers blobs of data (messages) to nodes, quickly and efficiently. You can map nodes to threads, processes, or nodes. ZeroMQ gives your applications a single socket API to work with, no matter what the actual transport (like in-process, inter-process, TCP, or multicast). It automatically reconnects to peers as they come and go. It queues messages at both sender and receiver, as needed. It limits these queues to guard processes against running out of memory. It handles socket errors. It does all I/O in background threads. It uses lock-free techniques for talking between nodes, so there are never locks, waits, semaphores, or deadlocks.

But cutting through that, it routes and queues messages according to precise recipes called patterns.

The built-in core ZeroMQ patterns are:

- Request-reply, which connects a set of clients to a set of services. This is a remote procedure call and task distribution pattern.
- Pub-sub, which connects a set of publishers to a set of subscribers. This is a data distribution pattern.
- Pipeline, which connects nodes in a fan-out/fan-in pattern that can have multiple steps and loops. This is a parallel task distribution and collection pattern.

There is a wire-level protocol called ZMTP that defines how ZeroMQ reads and writes frames on a TCP connection. If you're interested in how this works, the spec is quite short.

Device patterns

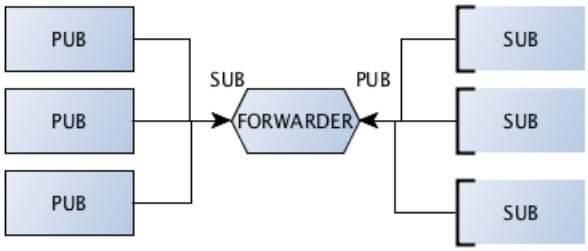
"To cope with blind spots we have to make use of another vocabulary or system outside the one in which the doubt exists." — Joseph. A Litterer

You must have noticed that you can bind a port to any of the ZeroMQ Socket types. In theory, most stable part of the network (the server) will **bind** on a specific port and have the more dynamic parts (the client) **connect** to that.

ZMQ provides certain basic proxy processes to build a more complex topology with basic device patterns our work this guide focus on *Forwarder* and *Streamer*.

Forwarder

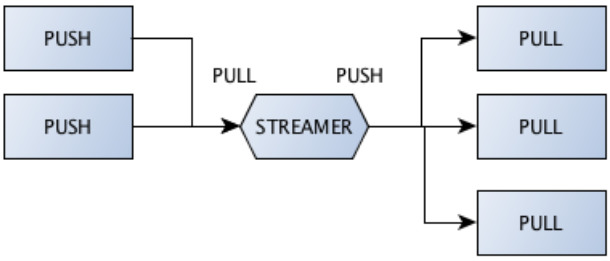
Forwarder device is like the pub-sub proxy server. It allows both publishers and subscribers to be moving parts and it self becomes the stable hub for interconnecting them.



This device collects messages from a set of publishers and forwards these to a set of subscribers.

Streamer

Streamer is a device for parallelized pipeline messaging. Acts as a broker that collects tasks from task feeders and supplies them to task workers.



Community

The ZeroMQ community uses a collaboration contract, C4.1. This is an RFC (of course), at http://rfc.zeromq.org/spec:22. It defines how the community works together and has been the main factor for the happy growth and stability of the community.

"Man is evidently the most intelligent animal but also, it seems, the most emotional." — D.O Hebb

We have produced this primarily to guide our own explorations in computational nonsense.

There are some gaps, but we have found that once mammals have gained some hands-on experience with a subject of models in the field, there is considerable possible transfer, and they are able to assimilate material about other models better once they have had this experience.

D.O Hebb propose that the human capacity for recognizing patterns without eye movement is possible only as the result of an intensive and prolonged visual training that goes on from the movement of birth, during every moment that the eye are open, with an increase in skill evident over a period of 12 to 16 years at least.

During the continuous, intensive, and prolonged visual training of infancy and childhood, we learn to recognize the direction of line and the distance between points, separately for each grossly separate part of the visual field.

Growth of the assembly

Let us assume then that the persistence or repetition of a trace tends to induce lasting cellular changes that add to its stability.

The assumption can be precisely stated as follows: When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.

The general idea is an old one, that any two cells, or systems of cells that are repeatedly active at the same time will tend to become

"associated" so that activity in one facilitates activity in the other.

One cannot guess how great the changes of growth would be; but it is conceivable, even probable, that if one knew where to look for the evidence on would find marked differences of identity in the perceptions of child and adult.

To get psychological theory out of a difficult impasse, one must find a way of reconciling tree things without recourse to animism: perceptual generalization, the stability of memory, and the instabilities of attention.

Our problem essentially is to see how a particular sensory event can have the same central effects on different occasions despite spontaneous central activity.

Considering the association areas as made up of a population of transmission units, two factors must affect the length of time needed to bring all these units under control.

One is the number of controlling fibers leading from sensory areas into association areas. The second is the number of transmission units in the association areas themselves.

We can then regard the stage of primary learning as the period of establishing a first environmental control over the association areas, and so, indirectly over behavior.

The learning occurs when the events to be associated ca already command organized trains of cortical activity; in other words, when the environment has a control of association areas that can be repeated, so that the central activity is not at random and the stimulation can impinge on the same central pattern when the training situation is repeated.

The characteristic adult learning is learning that takes place in a few trials, or in one only.

So adult learning is typically an interaction of two or perhaps three organized activities; being organized, they are capable of a continued existence after cessation of the stimulation that set them off, which gives time for the structural changes of permanent learning to take place.

This organized activity of the association areas is subject to environmental control. To the extent that the control is effective, and re-establishes the same central pattern of activity on successive trials, cumulative learning is possible.

Adult learning is thus a changed relationship between the central effects of separate stimulations, and does not directly concern the precipitating stimulus or, primary, the motor response whose control is embedded in the central activity.

The facts already discussed have indicated that one-trial learning occurs only as the association of concepts with "meaning" having, that is, a large number of associations with other concepts.

But more: the perception of an actual object that can be seen from more than one aspect, and touched, heart, smelled and tasted involves more that one phase cycle.

It must be a hierarchy: of phases, phase cycles, and a cycle of series of cycles.

"Cycle" is of course temporal: referring not to a closed anatomical pathway but to the tendency of a series of activities to recur, irregularly.

The two ideas or concepts to be associated might have not only phases, but one or more subsystems in common.

Two concepts may acquire a latent "association" without even having occurred together in the subject's past experience.

Cognitive processes

Mind is the central psychological problem, although it is no longer fashionable to say so, psychologists prefer to talk about "cognitive processes" instead.

They also, most of them, abstain from discussion of what those processes consist of and how their effects are achieved.

It is inaccurate-worse, it is misleading, to call psychology the study of behavior:

It is the study of the underlying processes, just as chemistry is the study of the atom rather than pH values and test tubes; but behavior is the primary source of data in modern psychology.

All science, from physics to physiology, is a function of its philosophic presuppositions, but psychology is more vulnerable than others to the effect of misconception in fundamental matter's because the object of its study is after all the human mind and the nature of human thought.

There is a well-developed specially called social psychology, which certainly sounds like social science; but social behavior can be considered from a biological point of view.

The idea that mind is a spirit is a theory of demonic possession, a form of the vitalism that biology got rid of a century ago.

Monism, the idea that mind and matter are not fundamentally different but different forms of the same thing: in practice in psychology, the idea that mental processes are brain processes.

Mind is the capacity for thought, consciousness, a variable state, is a present activity of thought processes in some form; and though itself is an activity of the brain.

Parallelism

The theory that mental events and brain events run side by side, perfectly, correlated but not causally related: in the old analogy, like two clocks that stay perfectly in step but not because either influences the other.

It has been highly regarded as a way of avoiding commitment to an interaction of mind and body, or even worse, identifying them, while recognizing how closely influences the other.

It has been highly regarded as a way of avoiding commitment to an interaction of mind and body, or even worse, identifying them, while recognizing how closely they are related.

Parallelism says that the actors in the theater, representing anger and fear did so with no guidance from their conscious minds; whatever thought there may have been in those entirely separated minds, the bodies functioned on the stage as self-programmed robots.

In the two-clock comparison with parallelism, the two clocks are separate entities by virtue of their reparation in space; if in addition to being identical in function.

They also occupied the same space, as mental activity and brain activity appear to do visual, auditory, verbal fluency, verbal comprehension, and so on each relating to particular parts of the brain, they would be on clock.

The objective evidence tells me that something complex goes on inside my head, I conclude therefore that something else is active also.

Reductionism

Reductionism is not a means of abolishing psychological entities and processes but a way of learning more about them.

Connectionism

Understanding through the interplay of multiple sources of knowledge. It is clear that we know a good deal about a large number of different standard situations. Several theories have suggested that we store this knowledge in terms of structures called variously: scripts, (Schank, 1976), frames (Minsky, 1975), or schemata (Norman & Bobrow, 1976; Rumelhart, 1975). Such knowledge structures are assumed to be the basis of comprehension. A great dial of progress has been made within the context of this view.

However, it is important to bear in mind that mostly everyday situations cannot be rigidly assigned to just a single script. They generally involve an interplay between a number of different sources of information.

Representations like scripts, and schemata are useful structures for encoding knowledge, although we believe they only approximate the underlying structure of knowledge representation that emerges from the class of models we consider in this bits.

Parallel distributed processing

A number of different pieces of information must be kept in mind at once. Each plays a part, constraining others and being constrained by them. What kinds of mechanisms seem well suited to these task demands? Intuitively, these tasks seem to require mechanisms in which each aspect of the information in the situation can act on other aspects. simultaneously influencing other aspects and being influenced by them. To articulate these intuitions, we and others have turned to a class of models we call Parallel Distributed Processing (PDP) models. These models assume that information processing takes place through the interactions of a large number of simple processing elements called units, each sending excitatory and inhibitory signals to other units.

How a pattern learns

So far, we have seen how we as model builders can construct the right set of weights to allow one pattern to cause another/ The interesting thing, though. is that we do not need to build these interconnection strenghts in by hand. Instead, the patterns associator can teach itself the right set of interconnections through experience processing the patterns in conjuction with each other.

A number of different rules for adjusting connection strengths have been proposed. One of the first, and definitely the best known is due to D. O. Hebb (1949) Hebb's actual proposal was not sufficiently quantitative to build into an explicit model. However, a number of different variants can trace their ancestry back to Hebb. Perhaps the simplest version is:

When unit A and unit B are simultaneously excited, increase the strength of the connection between them.

A natural extension of this rule to cover the positive and negative activation values allowed in our example is:

Adjust the strength of the connection between units A and B in proportion to the product of their simultaneous activation.

In this formulation, if the product is positive, the change makes the connection more excitatory, and if the product is negative, the change makes the connection more inhibitory. For simplicity of reference, we will call this the Hebb rule, although it is not exactly Hebb's original formulation.

It turns out that Hebb rule as stated here has some serious limitations, and, to our knowledge, no theorists continue to use it in this simple form. More sophisticated connection modulation schemes have been proposed; All these learning rules have the property that they adjust the strengths of connections between units on the basis of information that can be assumed to be locally available to the unit. Learning, then, in all of these cases, each connection without the need for any overall supervision. Thus models which incorporate these learning rules train themselves to have the right interconnections in the course of processing the members of an ensemble of patterns.

We already have noted Hebb's contribution of the Hebb rule of synaptic modification; he also introduced the concept of cell assemblies, a concrete example of a limited form of distributed processing, and discussed the idea of trace of activation within neural networks. Hebb's ideas were cast more in the form of speculations about neural functioning than in the form of concrete processing models, but his thinking captures some of the flavor of parallel distributed processing mechanisms.

Cognitive Science or Neuroscience?

One reason for the appeal of PDP models is their obvious "physiological" flavor: They seem so much more closely tied to the physiology of the brain than are other kinds of information-processing models. The brain consists of a large number of highly interconnected elements which apparently send very simple excitatory and inhibitory messages to each other and update their excitations on the basis of these simple messages.

The connectionism framework

It is useful to begin with an analysis of the various components of our models and then describe the various specific assumptions we can make about these components. These are eight major aspects of parallel distributed processing model:

- I. A set of processing units
- II. A state of activation
- III. An output function for each unit
- IV. A pattern of connectivity among units
- V. A propagation rule for propagating patterns of activities through the network of connectivities
- VI. An activation rule for combining the inputs impinging on a unit with the current state of that unit to produce a new level of activation for the unit
- VII. A learning rule whereby patterns of connectivity are modified by experience
- VIII. An environment within which the system must operate

I. A set of processing units

Any parallel activation model begins with a set of processing units. In some models these units may represent particular conceptual objects; in others they are simply abstract elements over which meaningful patterns can be defined. When we speak of distributed representation, we mean one in which the units represent small, feature-like entities.

A unit's job is simply to receive input from its neighbors and, as a function of the input it receives, to compute an output value which sends to its neighbors.

The system is inherently parallel in that many units can carry out their computations at the same time.

Within any system we are modeling, its useful to characterize three types of units: input, output, hidden.

Input units receive inputs from sources external to the system under study.

The output units send signals out of the system.

The hidden units are those whose only inputs and outputs are within the system we are modeling.

II. The state of activation

We need a representation of the state of the system at time T. This is primary specified by a vector of N numbers, a(t), representing the pattern of activation over the set of processing units.

It is the pattern of activation over the set of units that capture what the system is representing at any time.

It is useful to see processing in the system as the evolution, through time, of a pattern of activation over a set of units.

Different models make different assumptions about the activation values a unit is allowed to take on.

Activation values may be continuous or discrete. If they are continuous, they may be unbounded or bounded.

If they are discrete, they may take binary values or any of a small set of values.

III. Output of the units

Units interact.

They do so by transmitting signals to their neighbors. The strength of their signal, and therefore they degree to which they affect their neighbors, is determined by their degree of activation. In vector notation, we represent the current set of output values by a vector, o(t).

In some of our models the output level is , equal to the activation level of the unit.

IV. The pattern of connectivity

Units are connected to one another. It is this pattern of connectivity that constitutes what the system, knows and determines how it will respond to any arbitrary input.

Specifying the processing system and the knowledge encoded therein is, in a parallel distributed processing model, a matter of specifying this pattern of connectivity among the processing units.

We assume that each unit provides an additive contribution to the input of the units to which it is connected.

In such cases, the total input to the unit is simply the weighted sum of the separate inputs from each of the individual units.

That is, the input from all of the incoming units are t multiplied by a weigh and summed to get the overall input to that unit.

A positive weight represents an excitatory input a negative weight represents n inhibitory input.

The pattern of connectivity is very t. It is this pattern which determines what each unit represents.

One important issue that may determine both how much information can be stored and how much serial processing the network must perform is the fan-in and fan-out of a unit.

The fan-in is the number of elements that either excite or inhibit a given unit.

The fan-out of a unit is the number of units affected directly by a unit.

V. The rule of propagation

We need a rule which takes the output vector, o(t), representing the output values of the units and combines it with the connectivity matrices to produce a network input for each type of input into the unit.

In vector notation we can write network i(t) to represent the network input vector for inputs of type i.

VI. Activation rule

We need a rule whereby the network inputs of each type impinging on a particular unit are combined with one another and with the current state of the unit to produce a new state of activation.

We need a function, F, which takes a(t) and the vectors **net**j for each different type of connection and produces a new state of activation.

In the simplest cases, when F is the identity function and when all connections are of the same type, we can write a(t+1)=Wo(t)=net(t).

Sometimes F is a threshold function so that the net input must exceed some value before contributing to the new state of activation.

VII. Modifying patterns of connectivity as a function of experience.

Changing the processing or knowledge structure in a parallel distributed processing model involves modifying the pattern of inter-connectivity.

In principle this can involve three kinds of modifications:

1. The development of new connections.
2. The loss of existing connections.
3. the modifications of the strengths of connections that already exists.

VIII. Representation of the environment

It is crucial in the development of any model to have a clear model of the environment in which this model is to exist. In parallel distributed processing models, we represent the environment as a time-varying stochastic function over the space of input patterns.

We image that at any point in time, there is some probability that any of the possible set of input patterns is impinging n the input units, this probability function may in general depend on the history of inputs to the system as well as outputs of the system.

Each entity is represented by a pattern of activity distributed over many computing elements, and each computing element is involved in representing many different entities.

Sequential symbol processing

The obvious way to allocate the hardware is to use a group of units for each possible role within a structure and to make the pattern of activity in this group represent the identity of the constituent that is currently plating that role.

This implies that only one structure can be represented at a time unless we are willing to postulate multiple copies of the entire arrangement.

One way of doing this, using units with programmable rather than fixed connections is required.

The recursive ability to expand parts of a structure for indefinitely many levels and the inverse ability to package up whole structures

into a reduced form that allows them to be used as constituents of larger structures is the essence of symbol processing.

It allows a system to build structures out of things that refer to other whole structures without requiring that these other structures be represented in all their cumbersome detail.

This is exactly what is provided by sub-patterns that stand for identity / role combinations.

They allow the full identity of the part to be accessed form a representation of the whole and a representation of the role that the system wishes to focus on, and they also allow explicit representation, of an identity and a role to be combined into a less cumbersome representation, so that several identity / role combinations can be represented simultaneously in order to form the representation of a larger structure.

By encoding each piece of knowledge as a large set of interactions, it is possible to achieve useful properties like content-addressable memory and automatic generalization, and new items can be created without having to create new connections at the hardware level.

Another hard problem is to clarify the relationship between distributed representations and techniques used in artificial intelligence like schemas, or hierarchical structural descriptions.

Existing artificial intelligence programs have great difficulty in rapidly finding the schema that best fits the current situation.

Parallel networks offer the potential of rapidly applying a lot of knowledge to this best-fit search, but this potential will only be realized when there is a good way of implementing schemas in parallel networks.

Feedforward networks

An approach to speeding up learning is to exploit parallel computation. In particular, methods for training networks through asynchronous gradient updates have been developed for use on both single machines and distributed systems. By keeping a canonical set of parameters that are read by and updated in an asynchronous fashion by multiple copies of a single network, computation can be efficiently distributed over both processing cores in a single CPU, and across CPUs in a cluster of machines.

To avoid recomputations, we an think of back-propagation as a table-filling algorithm that takes advantage of storing intermediate results. Each node in the graph has a corresponding slot in a table to store the gradient for that node. By filling in these table entries in order, back-propagation avoids repeating many common subexpressions. This table-filling strategy is sometimes called dynamic programming.

Differentiation

The deep learning community has been somewhat isolated from the broader computer science community and has largely developed its own cultural attitudes concerning how to perform differentiation. More generally, the field of automatic differentiation is concerned with how to compute derivatives algorithmically.

The back-propagation algorithm described here is only one approach to automatic differentiation. It is a special case of a broader class of techniques called reverse mode accumulation. Back-propagation is therefore not the only way or the optimal way of computing the gradient, but it is a practical method that continues to server the deep learning community well. In the future, differentiation technology for neural networks may improve as deep learning practitioners become more aware of advances in the broader field of automatic differentiation.

Feedforward networks

A feedforward neural network is an artificial neural network wherein connections between the nodes do not form a cycle. As such, it is different from recurrent neural networks. In this network, the information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) and to the output nodes. There are no cycles or loops in the network.

Feedforward networks can be seen as efficient nonlinear function approximators based on using gradient descent to minimize the error in a function approximation.

The chain rule that underlies back-propagation algorithm was invented in the seventeenth century. Calculus and algebra have long been used to solve optimization problems in closed form, but gradient descent was not introduced as a technique for iteratively approximating the solution to optimization problems until the nineteenth century.

Inspired by Hebb in the 1940's these function approximation techniques were used to motivate machine learning models such as the perceptron. However, the earliest models wee based on linear models.

Learning nonlinear functions required the development of a multilayer perceptron and a means of computing the gradient through such a model. Efficient applications of the chain rule based on dynamic programming began to appear in the 1960s and 1970s, mostly for control applications but also for sensitivity analysis. Werbos 1981 proposed applying these techniques to training artificial neural networks. The idea was finally developed in practice after being independently rediscovered in different ways (LeCun, 1985; Parker, 1985; Rumelhart at al., 1986a). The book Parallel Distributed Processing presented the results of some of the first successful experiments with back-propagation in a chapter (Rumelhart et al., 1986b) contributed greatly to the popularization of back-propagation and initiated a very active period of research in multilayer neural networks.

The core ideas behind modern feedforward networks have not changed substantially since the 1980s. The same back-propagation algorithm and the same approaches to gradient descent are still in use.

A small number of algorithmic changes have also improved the performance of neural networks noticeably. One of these algorithmic changes was the replacement of mean squared error with the cross-entropy family of loss functions. Mean squared error was popular in the 1980s and 1990s but gradually replaced by cross-entropy losses and the principle of maximum likelihood as ideas spread between the statistics community and the machine learning community. The use of cross-entropy losses greatly improved the performance of models with sigmoid and softmax outputs, which had previously suffered from saturation and slow learning when using the mean squared error loss.

The other major algorithmic change that has greatly improved the performance of feedforward networks was the replacement of sigmoid hidden units with piece wise leaner hidden units, such as rectified linear units. As of the early 2000s, rectified linear units were avoided because of a somewhat superstitious belief that activation funtions with nondifferentiable points must be avoided. This began to change in about 2009. Jarrett at al. (2009) observed that "using a rectifying nonlinearity is the single most important factor in improving the performance of a recognition system," among several different factors of neural networks architecture design.

Today, gradient-based learning in feedforward networks is used as a tool to develop probabilistic models, rather than being viewed as an unreliable technology that must be supported by other techniques, gradient-based learning in feedforward networks has been viewed since 2012 as a powerful technology that can be applied to many other machine learning tasks.

In 2006, the community used unsupervised learning to support supervised learning, and now, ironically, it is more common to use supervised learning to support unsupervised learning.

Dropout!

A powerful method of regulating a broad family of models. To a first approximation, dropout can be thought of as a method of making nagging practical for ensembles of very many large neural networks.

Nagging involves training multiple models and evaluating multiple models on each test example.

Dropout provides an inexpensive approximation to training and evaluating a bagged ensemble of exponentially many neural networks.

Specifically, to train with dropout we use a minivan-based learning algorithm that makes small steps, such as stochastic gradient descent.

Each time we load an example into a minibatch, we randomly sample a different binary mask to apply to all the input and hidden units in the network.

So far we have described dropout purely as a means of performing efficient, approximation bagging.

We like another view of dropout that goes further than this.

Dropout!!

Dropout thus regularizes each hidden unit to be merely a good feature but a feature that is good in many contexts.

Wards-Farley et al (2014) compared dropout training to training of large ensembles and concluded that dropout offers additional improvements to generalization error beyond those obtained by ensembles of independent models.

Machine learning rules

One answer that has occurred to many people over the years is the idea of using the difference between the desired target activation and the obtained activation to drive learning.

The idea is to adjust the strengths of the connections so that they will tend to reduce this difference or error measure. Because the rule is driven by differences was historically called the delta rule.

Others have called it the Widrow-Hoff learning rule of the least mean square (LMS) rule it is related to the perceptron convergence procedure of Rosenblatt(1959).

The correct set of weights is approached asymptotically if the training procedure is continued for several more sweeps through the set of patterns.

Each of these sweeps, or training epochs, as we will call them henforth, results in a set of weights that is closer to a perfect solution.

To get a measure of the closeness of the approximation to a perfect solution, to get a measure of the closeness of the approximation to be a perfect soution, we can calculate an error measure for each pattern as that pattern is being processed. For each pattern, the error measure is the value of the error (t-a) squared.

This measure is then summed over all patterns to get a total sum of squares or tss measure.

The error-correcting learning rule, then, is much more powerful than the Hebb rule.

In fact, it can be proven rather easily that error-correcting rule will find a set of weights that drives the error as close to 0 as we want for each and every pattern in the training set, provided such a set of weights exists.

The Linear Predictability Constraint

We have just noted that the delta rule will find a set of weights that solves a network learning problem, provided such a set of weights exists.

What are the conditions under which such a set of actually does exist?

Such a set of weights exists only if for each input-pattern-target-pair the target can be predicted from a weighted sum, or linear combination of the activation of the input units.

There is a constraint which we called the linear-predictability constraint that can be overcome by the use of hidden units, but hidden units can not be trained using the delta rule as we have described it here because by definition there is no teacher for them.

The Linear Predictability Constraint Again

Earlier we considered the linear predictability constraint for training a single output unit. Since the pattern associator can be viewed as a collection of several different output units, the constraint applies to each unit in the pattern associator.

The pattern associator

The algorithm described previously has been known since the late 1950's, when variants of what we have called the delta rule were first proposed.

In one version, in which output units were linear threshold units, it was known as the perceptron (cf. Rosenblalt, 1959,1962).

In another version, in which the output units were purely linear, it was known as the LMS or least mean square associator (cf. Widrow&Hoff, 1960).

in the more general case of multilayer networks, we categorize units into three classes: input, units which receive the input patterns directly; output units, which have associated teaching or target inputs; and hidden units, which neither receive inputs directly nor are given direct feedback.

This is the stock of units from which new features and new internal representations can be created. The problem is to know which new features are required to solve the problem at hand.

In short, we must be able to learn intermediate layers. *The Question is how?*

The original perceptron learning procedure does not apply to more than one layer.

Minsky and Papert believed that no such general procedure could be found.

To examine *How* such a procedure can be developed it is useful to consider the other major one-layer learning system of the 1950s and early 1960's namely, the least mean-square (LMS) learning procedure of Widrow and Hoff (1960).

The LMS procedure finds the values of all of the weights that minimize this function using a method called gradient descent.

That is, after each pattern has been presented, the error on that pattern is computed and each weight is moved "down" the error gradient toward its minimum value for that pattern.

Since we cannot map out the entire error function on each pattern presentation, we must find a simple procedure for determining, for each weight, how much to increase or decrease each weight.

The idea of gradient descent is to make a change in the wight proportional to the negative of the derivative of the error, as measured on the current pattern, with respect to each weight.

It should be clear why we want the negation of the derivative.

If the weight is above the minimum value, the slope at that point is positive and we want to decrease the weight; thus when the slope is positive we add a negative amount to the weight. On the other hand, if the weight is too small, the error curve has a negative slo[e at that point, so we want to add a positive amount to the weight.

In this case, the LMS procedure makes changes to the weights proportional to the effect they will have on the summed squared error.

The resulting total change to the weights is a vector that points in the direction in which the error drops mostly steeply.

The Back Propagation Rule!

The basic idea of the back propagation method of learning is to combine a nonlinear perceptron-like system capable of making decisions with the objective error function of LMS and gradient descent.

We will not bother with the mathematics here, since it is presented elsehere in detail.

Suffice to say, that with an appropriate choice of non-linear function we can perform the differentioation and derive the back propagation learning rule.

The application of the bac propagation rule, then, involves two phases:

During the first phase the input is presented and progagated forward through the network to compute the output value for each unit.

This output is then compared with the target, resulting in a term for each output unit.

The second phase involves a backward pass through the network (analogous to the initial forward pass) during which the term is computed for each unit in the network. This second, backwards pass allows the recursive computation of the term as indicated above.

Once these two phases are complete, we can compute, for each weight, the product of the term associated with the unit it projects to times the activation of the uit it projects from.

Henceforth we will call this product the weight error derivative since it is proportional to (minus) the derivative of the error with respect to the weight. These weight error derivatives can then be used to compute actual weight changes on a pattern-by-pattern basis, or they may be accumulated over the ensemble of patterns.

Convolutional networks

Convolutional networks (LeCun, 1989), also known as convolutional neural networks, or CNNs, are a specialized kind of neural network for processing data that has a known grid-like topology. Examples include time-series data, which can be thought of as a 1-D grid taking samples at regular time intervals, and image data, which can be thought of as a 2-D grid of pixels.

The name "convolutional neural networks" indicates that the network employs a mathematical operation called convolution. Convolution is a specialized kind of linear operation. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

Usually, the operation used in a convolutional neural network does not correspond precisely to the definition of convolution as used in other fields, such as engineering or pure mathematics.

Convolutional networks stand out as an example of neuroscientific principles influencing deep learning.

In convolutional network terminology, the first argument (in this example let's say the function x) to the convolution is often referred to as the *input*, and the second argument (the function w) as the *kernel*. The output is sometimes referred to as the *feature map*.

In machine learning applications, the input is usually a multidimensional array of data, and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. We will refer to these multidimensional arrays as tensors.

It is rare for convolution to be used alone in machine learning; instead convolution is used simultaneously with other functions, and the combination of these functions does not commute regardless of whether the convolution operation flips its kernel or not.

Discrete convolution can be viewed as multiplication by a matrix, but the matrix has several entries constrained to be equal to other entries.

In two dimensions, a doubly block circulant matrix corresponds to convolution. In addition to these constraints, convolution usually corresponds to a very sparse matrix (a matrix whose entries are mostly equal to zero).

Convolution leverages three important ideas that can help improve a machine learning system: sparse interactions, parameter sharing and equivariant representations.

Sparse interactions

Sparse interactions (also referred to as sparse connectivity or sparse weights). This is accomplished by making the kernel smaller than the input when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. We need to store fewer parameters, which both reduces the memory that computing the output requires fewer operations.

Parameter sharing

As a synonym for parameter sharing, one can say that a network has tied weights, because the value of the weight applied to one input is tied to the value of a weight applied elsewhere.

The parameter sharing used by the convolution operation means that rather thn learning a spearte set of parmeters for every location, we learn only one set.

Convolutions is thus dramatically more efficient than dense matrix multiplication in terms of the memory requirements and statistical efficiency.

Equivariant representations

Convolution is an extremely efficient way of describing transformations that apply the same linear transformation of a small local region across the entire input.

In the case of convolution, the particular form of parameter sharing caused the layer to have a property called equivariance to translation. To say a function is equivariant means that if the input changes, the output changes in the same way.

Implementing convolution

Other operations besides convolutions are usually necessary to implement a convolution network. To perform learning, one must able to compute the gradient with respect to the kernel, given the gradient with respect to the outputs.

Recall that convolution is a linear operation and can this be described as a matrix multiplication. The matrix is sparse, and each

element of the kernel is copied to several elements of the matrix. This view help us to derive some of the other operations needed to implement a convolution network.

Multiplication by the transpose of the matrix defined by convolution is one such operation. This is the operation needed to back-propagate error derivatives through a conolution layer, so it is needed to train convolutional networks that have more than one hidden layer.

These three operations, convolution, backprop from output to weights, and backprop from output to inputs are sufficient to compute all the gradients needed to train any depth of feedforward convolutional network.

It is also worth mentioning that neuroscience has told us relatively little about how to train convolutional networks. Model structures with parameter sharing across multiple spatial locations date back to early connectionist models of vision (Marr and Poggio, 1976), but these models did not use the modern back-propagation algorithm and gradient descent.

Convolutional nets were some of the first working deep networks train with back-propagaion. It is not entirely clear why convolutional networks succeeded when general back-propagation networks were considered to have failed. It may simple be that convolutional networks were more computational efficient than fully connected networks, so it was easier to run multiple experiments with them, and tune their implementation and hyperparameters.

It may be that primary barriers to the success of neural networks were psychological (practitioners did not expect neural networks to work, so they did not make serious effort to use neural networks).

Convolutional networks provide a way to specialize neural networks to work with data that has a clear grid-structured topology and to scale such models to very large size. This approach has been the most successful on a two-dimensional image topology.

Capsule networks

Capsules are vector things that got a factor activity, capsules in one layer send information to capsules in the next layer the capsule in the next layer gets active if it sees a bunch of incoming vectors that agree, now of course the capsule in the next layer doesn't see just the output but it see the output multiplied by a weight matrix.

If those products agree, if it gets good agreement even if there is some outliers they will say hey I doung something and of course high-dimensional coincidence if you get six dimensional things to agree even if only dimension, each dmension only agreeto within ten percent the chance of a six dimensonal thing agreein is like one in a million, I mean if its sort of attempt of the normal long to disparity on each dimension then it's a millionth of the disparity of two random things so a high dimensional agreement is a really, really significant thing and is a much better filter than what we do at present which is you apply some weights see if you get above the threshold...

Obviously we know that if you stack that up and you train it by stochastic gradient decent it can do anything , but if you go back to basics its not in its nature to automatically be looking ot covariances between vectors and we want units where that's part of their nature.

The edge of a capsule

The edge of a capsule is a vector that represent the different properties of a thing, of an entity, now the entity might be a little fragment it might be something much bigger and if we won't doing vision it might be something else all together but the other aspect of capsules is we're going to try inside the network to get entities, a common neural network is not really committed to found multi-dimensional entities, where if you look at how people deal with the world they deal with the world in terms of objects that have properties.

We're going to understand the world in terms of entities and these entities are going to have properties and what a capsule is going to do, is its going to make a fundamental commitment.

If I'm a capsule I got a bunch of units and some of them are active at once then just the fact that were active together means they apply to the same thing, if you go sequential that is only do one thing at a time then you can bind together arbitrary things just by simultaneity, We want to do that at a low level two.

Factor analysis

Factor analysis is a statistical method used to describe variability among observed, correlated variables in terms of a potentially lower number of unobserved latent variables called factors.

For example, it is possible that variations in six observed variables mainly reflect the variations in two unobserved (underlying) variables. Factor analysis searches for such joint variations in response to unobserved latent variables. The observed variables are modelled as linear combinations of the potential factors, plus "error" terms.

The theory behind factor analytic methods is that the information gained about the interdependencies between observed variables can be used later to reduce the set of variables in a dataset.

Factor analysis performs a maximum likelihood estimate of the so-called loading matrix, the transformation of the latent variables to the observed ones, using expectation-maximization (EM).

Advantages

Both objective and subjective attributes can be used provided the subjective attributes can be converted into scores.

Factor analysis can identify latent dimensions or constructs that direct analysis may not.

Disadvantages

Usefulness depends on the researchers' ability to collect a sufficient set of product attributes. If important attributes are excluded or neglected, the value of the procedure is reduced.

If sets of observed variables are highly similar to each other and distinct from other items, factor analysis will assign a single factor to them. This may obscure factors that represent more interesting relationships.

Naming factors may require knowledge of theory because seemingly dissimilar attributes can correlate strongly for unknown reasons.

Maximum likelihood

Another view of learning is that the weights in the network constitute a generative model of the environment as would like to find a set of weights so that when the network is running freely, the patterns of activity that occur over the visible units are the same as they would be if the environment was clamping them.

Expectation Maximization

The EM algorithm is used to find (local) maximum likelihood parameters of a statistical model in cases where the equations cannot be solved directly. Typically these models involve latent variables in addition to unknown parameters and known data observations. That is, either missing values exist among the data, or the model can be formulated more simply by assuming the existence of further unobserved data points.

Finding a maximum likelihood solution typically requires taking the derivatives of the likelihood function with respect to all the unknown values, the parameters and the latent variables, and simultaneously solving the resulting equations.

The EM algorithm proceeds from the observation that there is a way to solve these two sets of equations numerically. One can simply pick arbitrary values for one of the two sets of unknowns, use them to estimate the second set, then use these new values to find a better estimate of the first set, and then keep alternating between the two until the resulting values both converge to fixed points. It's not obvious that this will work, but it can be proven that in this context it does, and that the derivative of the likelihood is (arbitrarily close to) zero at that point, which in turn means that the point is either a maximum or a saddle point.

The EM iteration alternates between performing an expectation (E) step, which creates a function for the expectation of the log-likelihood evaluated using the current estimate for the parameters, and a maximization (M) step, which computes parameters maximizing the expected log-likelihood found on the E step. These parameter-estimates are then used to determine the distribution of the latent variables in the next E step.

In general, multiple maxima may occur, with no guarantee that the global maximum will be found. Some likelihoods also have singularities in them, i.e., nonsensical maxima.

Constraint satisfaction

We being with a general discussion of constraint satisfaction and then turn to the schema model.

Consider a program whose solution involves the parallel processing of a very large number of processes. To make the problem more difficult, suppose that there may be no perfect solution in which all of the constraints are completely satisfied.

In such a case, the solution would involve processes of as many constraints as possible.

Image that some constraints may be more important than others.

In particular, suppose that each constraint has an important value associated with it and that the solution to the problem involves the simultaneous processing of as many of the most important of these constraints as possible.

It happens to be one of the kinds of problems that parallel distributed processing systems solve in a very natural way.

To the connectionist, Hinton was the first to sketch the basic idea of using parallel networks to solve constraint satisfaction problems (Hinton 1977).

Each unit represents a hypothesis and each connection a constraint among hypothesis.

The goal is to find a solution in which as many of the most important constraints are satisfied as possible.

Finally, different hypotheses may have different a priori probabilities. An appropriate solution to a constraint satisfaction problem must be able to reflect such prior information as well.

This is done by assuming that each unit has a bias, which acts to turn the unit on in the absence of other evidence. If a particular unit has a positive bias, then it is better to have the unit on; if it has a negative bias, there is a preference for it to be turned off.

A simple way of expressing this is to let the product of the activation of two units times the weight connecting them be the degree to which the constraint is satisfied.

Schema model

The schema model is one of the simplest of the constraint satisfaction models but, nevertheless, it offers useful insights into the operation of all the constraint satisfaction models.

The basic idea is that our knowledge is in the form of a constraint satisfaction network.

Conventionally (cf. Rumelhart & Ortony, 1977), a schema is a higher-level conceptual structure for representing the complex relationships implicit in our knowledge base.

Schemata are data structures for representing generic concepts stored in memory. They are like models of the outside world.

Information is processed by first finding the schema that best fits the current situation and then using that model to fill in aspects of the situation not specified by the current input.

Within the connectionist framework there is no explicit unit or other representational entity corresponding to a schema. Rather, schemata are implicit in our knowledge and arise, while processing information, form the interactions of a large set of constraints.

We argue that talk at the level of units and activations of units is the preferable way of describing human thought.

There are many important concepts from modern cognitive science which must be explicated in our adventures. Perhaps the most important, however, is the concept of the *schema* or related concepts such as scripts, frames, and so on.

These large scale data structures have been posited as playing critical roles in the interpretation of input data, the guiding of action, and the storage of knowledge in memory.

Instead, we use building blocks at a much more microlevel, at the level of units, activations and similar "low-level" concepts.

Perhaps the first thought that comes to mind is to map the notion of the schema onto the notion of the unit. However, such an identification misses much of what makes the scheme a powerful conceptual tool, the scale is wrong.

Schema theorists talk of schemata for rooms, stories, restaurants, birthday parties and many other high-level concepts.

In our parallel distributed processing models, units do not tend to represent such complex concepts. Instead, uits correspond to relatively simple features or as Hinton (1981) calls them microfeatures.

Doing justice to the concept of the schema, we are going to have to look beyond the individual unit.

We are going to have to look for schemata as properties of entire networks rather than single units or small circuits.

To summarize, there is a large subset of parallel distributed processing models which can be considered constraint satisfaction models.

These networks can be described as carrying out their information processing by climbing into states of maximal satisfaction of the constraints implicit in the network.

A very useful concept that arises from this way of viewing these networks is that we can describe the behavior of these networks, not only in terms of the behavior of individual units, but in terms of properties of the network itself.

We recounted a perspective on parallel distributed processing systems we address the nature of the schema and relate it to constraint satisfaction systems in parallel distributed processing models.

We will proceed here by recounting some of the history of the concept of schemata, then by offering an interpretation of the schema in terms of neural network models, by giving a simple example, and finally showing how the various properties attributed to schemata are, in fact, properties of Parallel Distributed Processing networks.

The schema, throughout its history, has been a concept shrouded in mystery.

Piagetian schemas