If I Only Had a Brain
...In Scala

Julie Pitt
Order of Magnitude Labs
@yakticus
quick poll
You’ll walk out of this room with...

- evidence that there is one algorithm for the brain
- a glimpse of how your brain does inference
- basic inference in ~180 lines of scala
Does the brain run on a single algorithm?
Echolocation
Sensory substitution devices

Sound is captured and processed on a smartphone. The data is then sent over Bluetooth and played in real-time using a series of vibration motors on a wearable vest.
Hypothesis:

brain function is a product of sensory input
Testing the hypothesis

- we will use a visual example
- same reasoning applies to other sensors
What do you see?
How did your brain find the dalmatian?
Three key ingredients

1. pooling
2. inhibition
3. hierarchy
1. pooling
Coincidence detection
Sparse Distributed Representations

- Sparse, binary vector
- Each “on” bit represents a firing neuron
- Information is distributed across “on” bits
- Similar inputs have high degree of overlap

Overlap
“Fire together, wire together”

- Each connection has permanence value
  - When pooler and input fire together, ++permanence
  - When pooler fires and input doesn’t, --permanence
- When computing overlap, only use connections where (permanence >= threshold)
Properties of poolers

- coincidence detectors
  - coincidence = occurs close in time
- resistant to noise
- tolerant to signal loss
  - can recognize incomplete patterns
2. inhibition
Layers

- Poolers are arranged in layers
  - can be 2D (matrix) or 1D (vector)
- Poolers in the same layer cross-inhibit one another

Spotted mammal layer
Resolving ambiguity
Most overlap wins

Inhibition works within a layer
Properties of inhibition

- **Self-organizing**
  - Poolers avoid learning the same thing
  - Similar concepts clump together

- **Sparsity emerges**
  - Compression algorithm
  - Prevents dense activations
3. hierarchy
Hierarchical layers

- Output of a layer is input to layer(s) above it
  - output = firing pattern = SDR
Properties of hierarchy

● Increasing abstraction
  ○ More complex patterns
  ○ More abstract concepts

● Increasing stability of activation
  ○ Slower moving firing patterns
shout out
On Intelligence & NuPIC

- **On Intelligence**, by Jeff Hawkins
- **NuPIC OSS**
let’s implement
Why Scala?

- **Efficient**
  - time spent working on algorithm, not writing code
  - rewrites in days, refactoring in hours
  - ~100k training iterations in a few seconds

- **Compact**
  - demo in ~180 lines of code

- **Type safe**
  - compiler does most of the debugging
Design

- entirely immutable
  - represent state as nested case classes
  - all state encapsulated in “top level” case class (Model)
- functional
  - largely a series of transformations on collections
- avoid circular references
  - refer to poolers by address
Approach

- initial prototype used akka
  - each pooler was an actor
  - it was going to run on 1000 machines!
  - it was going to be awesome!
  - but it wasn’t.

- reset: remove the frameworks
  - solve in simplest possible way first
  - only add frameworks when there is no other way
  - always ask whether you can simplify
Data Model

case class Model(layers: List[Layer], inhibition: Inhibition)

case class Layer(poolers: IndexedSeq[Pooler], active: Set[Int])

trait Inhibition {
  def compete(overlaps: Iterable[Overlap]): Set[Int]
}

case class Pooler(index: Int, connections: Map[Int, Double],
  permanenceThreshold: Double, updatePermanence: PermanenceFunction)
Macro data flow

// This is the stream of raw data coming from “the world”
data: Stream[T]

// Knows how to encode “real world” data into SDRs
trait Encoder[T] {
  def encode(item: T): Set[Int]
}

// The encoded SDR which is the sensory reading
input: Set[Int]

// Takes the sensory reading and does its magic
class Model … {
  def processInput(input: Set[Int]): Model
}
Model data flow

Layer

Model

layer.overlap(input)

Pooler

inhibition.compete(overlaps)

Inhibition

layer.learn(winners, input)
tips
1. case classes have a secret

.copy() = builder pattern++

```scala
// Original object acts as template for new object; only touch fields being modified
case class Pooler(..., connections: Map[Int, Double], ...) { ...
def learn(input: Set[Int]): Pooler = {
  this.copy(connections = connections.map {
    case (axonIndex, permanence) =>
      axonIndex -> updatePermanence(permanence, input.contains(axonIndex))
  })
}

// Perform a chain of fluent operations
val newModel = model.step1.step2.step3
```
2. beware of .mapValues

.mapValues creates a map view (see article)

case class SomethingWithMap(theMap: Map[Int, Layer]) {
  def doSomething(input: Set[Int]): SomethingWithMap = {
    // Retaining the map view --> GC out of control!
    this.copy(theMap = theMap.mapValues(_.learn(Set.empty, input)))
  }
}
3. Streams can hog memory

- streams are memoized
- holding onto references can cause memory leaks
  - see 3 part article
How to parallelize

- Each layer is self-contained
  - one layer processing one input is a task
- All layers operate in parallel
- Activation propagates up the hierarchy
  - One hop per iteration
demo time!
Word game

free
tree

f r t e
so what?
Goal: intelligent behavior
Contact & Code

@yakticus

julie@oomagnitude.com

Scala Code
Thanks for listening