Of Forms and Structures

How different neural architectures fit different RE problems

VINCENZO GERVASI
Dipartimento di Informatica - Università di Pisa

Summary

What's going on

As the existence of AIRE itself proves, Artificial Intelligence is gaining an increasing role in RE

Not surprisingly: RE is a discipline of knowledge, beliefs, wishes and understanding

Yet, so far most applications fall into two classes of problems

- classification
- NLP-related

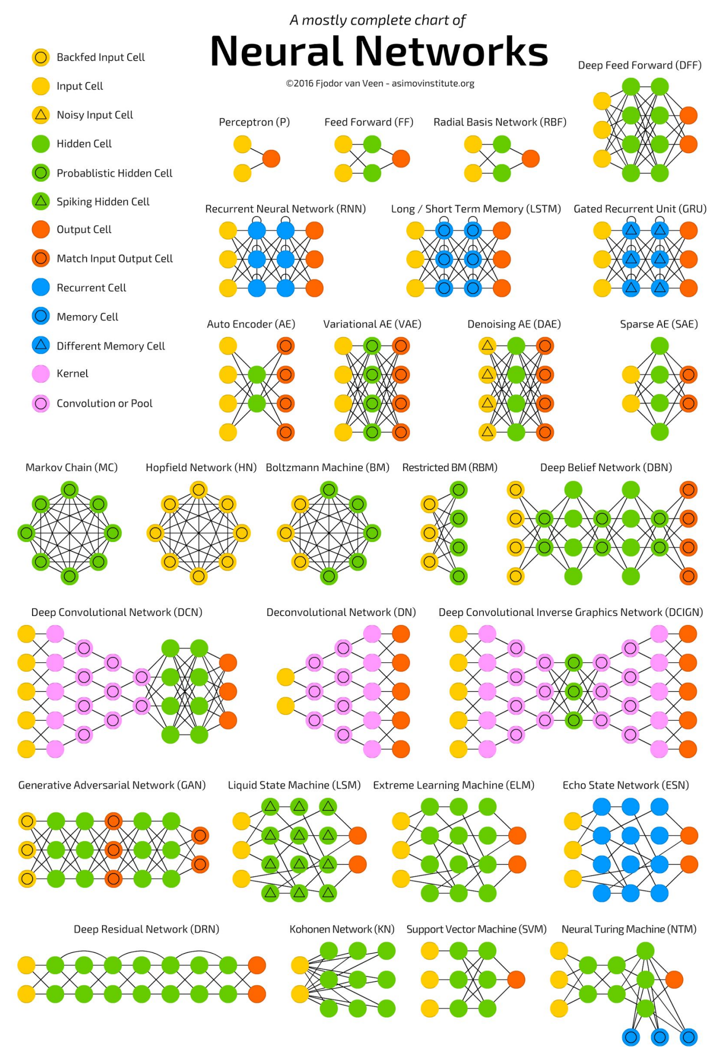
Can we do more?

A crowded zoo

Inspiration for this talk came from a delightful small post at the Asimov Institute*

If you can't tell your SAEs from your VAEs and DAEs by heart, this is a good post to read

Our question: how the diverse beasts in the zoo can help tackle different RE problems?



* http://www.asimovinstitute.org/

Basics

One of the most basic task is **supervised learning**

We have a number of cases of (*input*,*output*) observations

often, the training set requires manual tagging or historical data

Want to train a NN to learn the relation

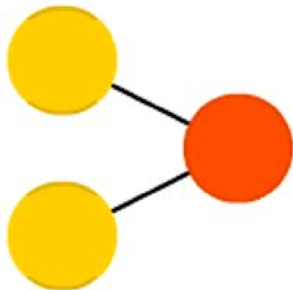
Then, we feed a previously-unseen *input*, and expect the NN to divine the *output*

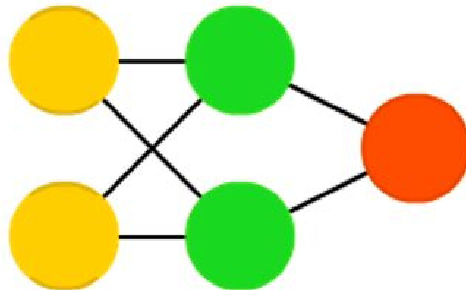
When the *output* is interpreted as a class to which the *input* belong, we have the most classic task of **classification**

Basics

Perceptrons (P) and feed-forward neural networks (FF) provide the basic machinery for supervised learning

On each learning step, a measure of the difference between the expected output and the actual output is back-propagated, thus optimizing the weights along the various connections





- Backfed Input Cell
- Input Cell
- Noisy Input Cell
- Hidden Cell
- Probablistic Hidden Cell
- Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- Different Memory Cell
- Kernel
- Convolution or Pool

Basics

In theory, given a sufficient number of nodes / hidden layers, and a sufficient number of training samples, a FFNN can learn most relations

In practice, performance is terrible, and capabilities are limited (however, they make useful components of more complex architectures)

A major problem for RE: how do you get enough samples for training?

- We all know: RE is human-intense, no huge annotated datasets are standardised or even available, etc.
- Student-based annotation may not provide representative samples for real-world learning

An example

Automatically Classifying Requirements from App Stores: A Preliminary Study
by R. Deocadez, R. Harrison, D. Rodriguez

2 classes: functional or non-functional (requirement)

300 manually-classified app store reviews used for training (in a semi-supervised learning context); forced 50/50 representation of F/NF

1 million reviews for 40 top-scoring apps

k-NN (*k* **n**earest **n**eighbour, not **n**eural **n**etwork!), C4.5, Naive Bayes, SVM

Beyond classification

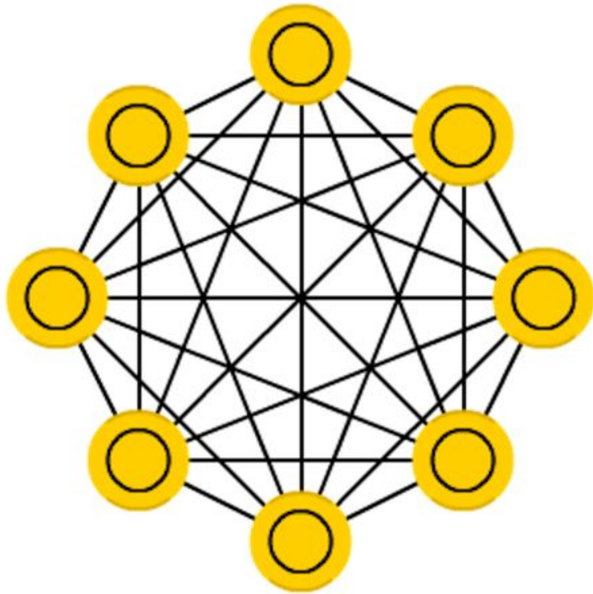
Although interesting, separating (mostly piecewise-linearly separable) instances into classes cover just a small fraction of RE problems

Thought experiment: imagine a day in the life of a requirement engineer. How much time would s/he spend on the job looking at instances and classifying them?

New job of the year:

Requirement **E**ntomologist

Identifying patterns



Something we could want to do is **identifying patterns** in things

A **Hopfield Network (HN)** connects every node to every other; each node is both input, hidden and output at different stages

Once trained with “perfect instances” of patterns, the network will converge to the (right?) “perfect instance” when presented with imperfect instances

Identifying patterns

HNs can work as **classifiers**, if classes coincide with patterns

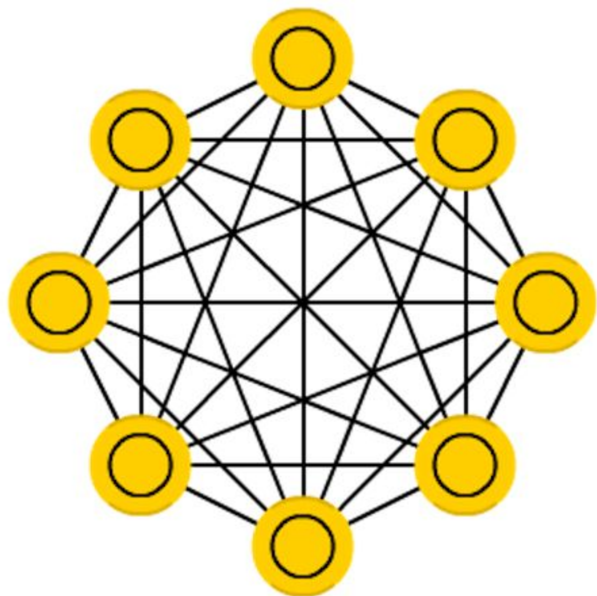
HNs can work as **de-noisers**, if the difference between the imperfect instance and the pattern can be interpreted as noise

HNs can work as **associative memories**, if we present just part of a pattern and want it complete.

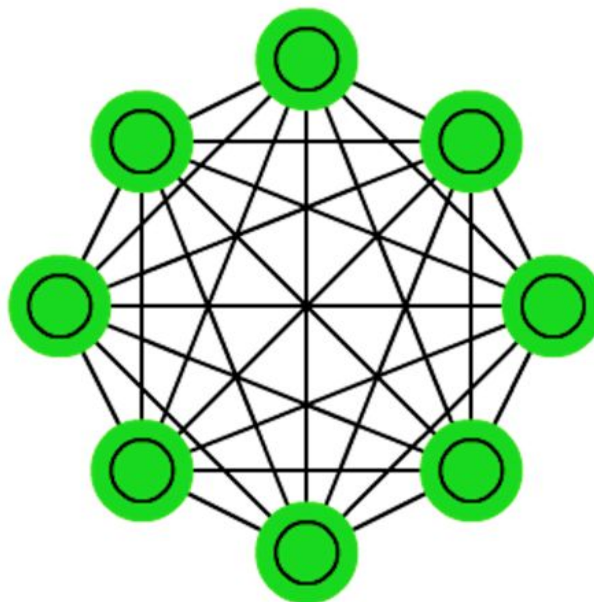
All these uses correspond to different RE problems, for example:

- autocompletion in a “soft” editing tool
- refactoring of requirements / templates

HN vs MC



Hopefield Network



Markov Chain

- Backfed Input Cell
- Input Cell
- Noisy Input Cell
- Hidden Cell
- Probablistic Hidden Cell
- Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- Different Memory Cell
- Kernel
- Convolution or Pool

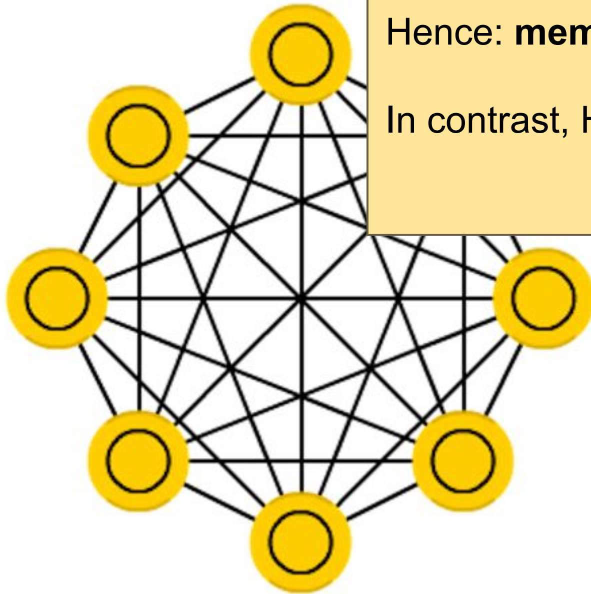
HN vs MC

Structurally similar, but used differently.

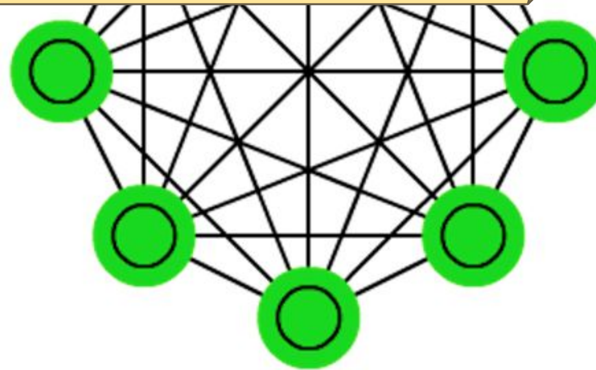
In MC, a link (*a*,*b*) with weight *w* means that if the present state is *a*, then the probability that the next state is *b* is *w*.

Hence: **memoryless** -- only the current state counts.

In contrast, HN are more global (values propagation)



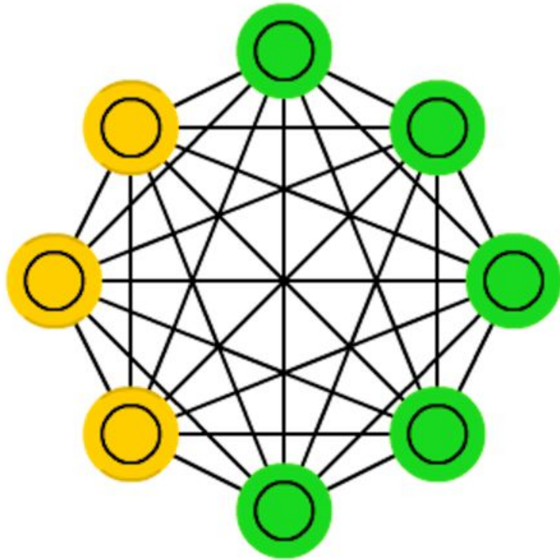
Hopefield Network



Markov Chain

Backfed Input Cell
Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Different Memory Cell
Kernel
Convolution or Pool

Completing patterns



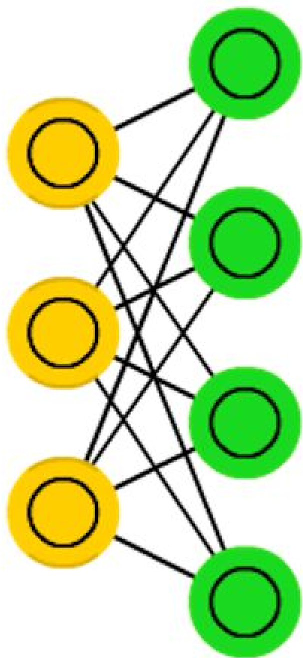
Boltzmann Machines (BM) combine HN and MC

Certain nodes are designated as input, while the hidden layer is probabilistic

Run by going back and forth until the network stabilizes

- Backfed Input Cell
- Input Cell
- Noisy Input Cell
- Hidden Cell
- Probablistic Hidden Cell
- Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- Different Memory Cell
- Kernel
- Convolution or Pool

Completing patterns



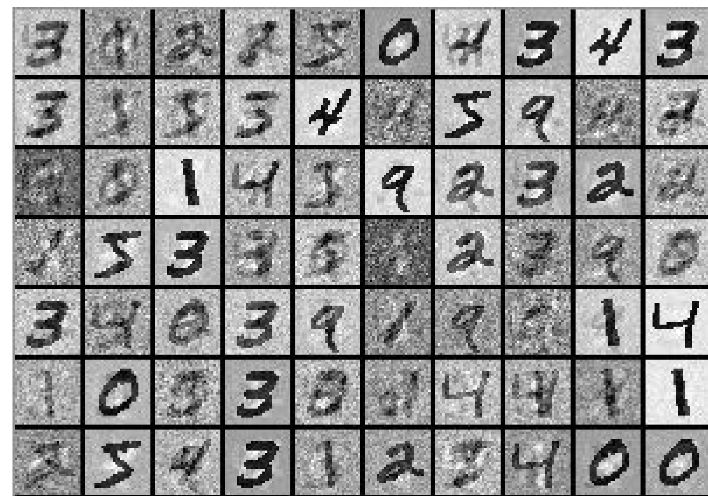
Restricted Boltzmann Machines (RBM) are similar, but connect by layers instead of a complete graph

This makes them more easily usable and more efficient to run

Importantly, these are the building blocks for **deep belief networks** (later)

- Backfed Input Cell
- Input Cell
- Noisy Input Cell
- Hidden Cell
- Probablistic Hidden Cell
- Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- Different Memory Cell
- Kernel
- Convolution or Pool

Generating data



(Stacked) RBMs can be used to **generate** ideal forms of data.

A scenario for **unsupervised learning**:

- the network is trained on a number of untagged instances
  - this sets weights on the links
- then, random values are set on the nodes, the network is run until stabilization, and output is read on the “input” nodes
- This is sometimes referred as **machines dreaming**

Use cases for RBM

In general:

- dimensionality reduction
- classification
- regression
- collaborative filtering
- feature learning
- topic modeling

In RE applications:

- What if I reverse the FR/NFR classifier and ask to generate a requirement (given the class) instead?
  - Creativity-enhancing techniques
- Once we have trained a network in an unsupervised fashion, what can we learn from the synthesized stable states?
- Let's feed requirements at a RBM. It will implicitly classify them according to "invented" classes
  - Will these classes mirror the IEEE Stds?

Compressing and diluting data

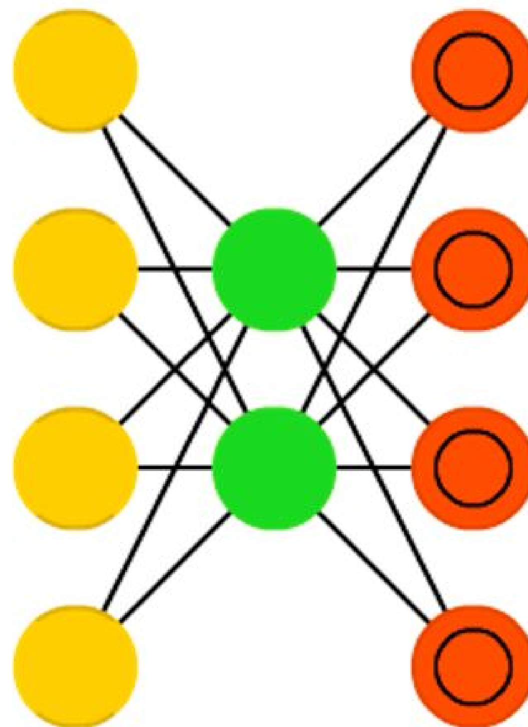
Auto-encoders (AE) are a nifty way to compress data

Values are presented to the input cells, and the same values are expected in the output cells

During training, the AE is forced to invent compact representations in the hidden cells

- simply because there are less green nodes than yellow/red nodes

In use, compressed form is read from "hidden" cells



- Backfed Input Cell
- Input Cell
- Noisy Input Cell
- Hidden Cell
- Probablistic Hidden Cell
- Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- Different Memory Cell
- Kernel
- Convolution or Pool

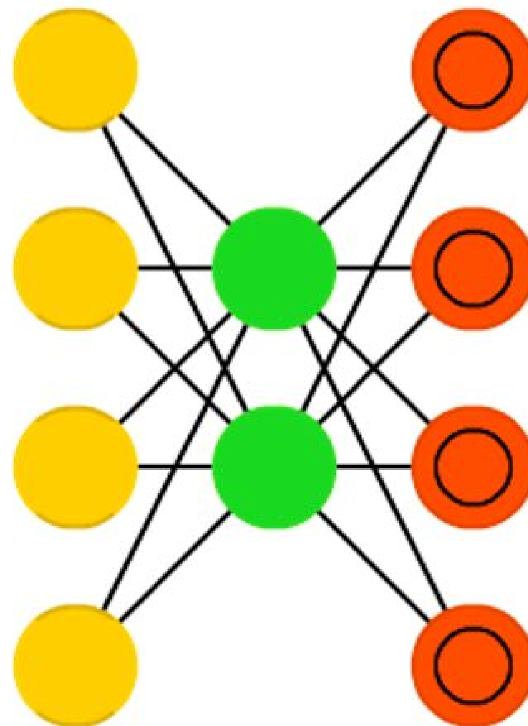
Compressing and diluting data

This idea could be applied, for example, to identifying **redundancy** in input data

Example: maybe separate features are not that independent?

- All requirements coming from a certain person are robustness requirements?
- All requirements after a certain time are scheduled for the next release?

There is knowledge to be extracted both at the code (hidden cells) and at the weights

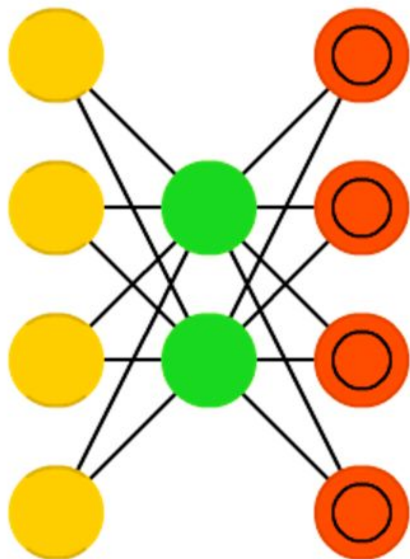


Backfed Input Cell
Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Different Memory Cell
Kernel
Convolution or Pool

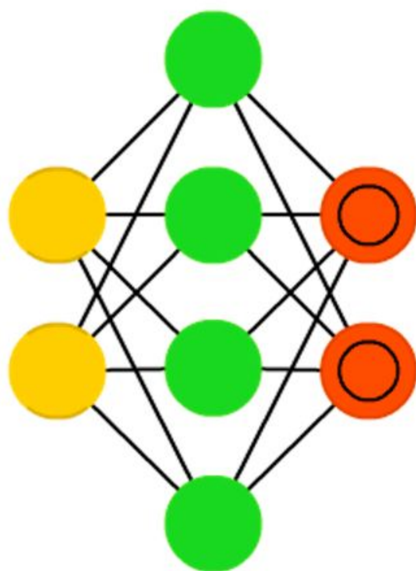
Compressing and diluting data

Variants of AEs are used to learn more sparse representations that are resistant to noise, or to handle probability distributions instead of values

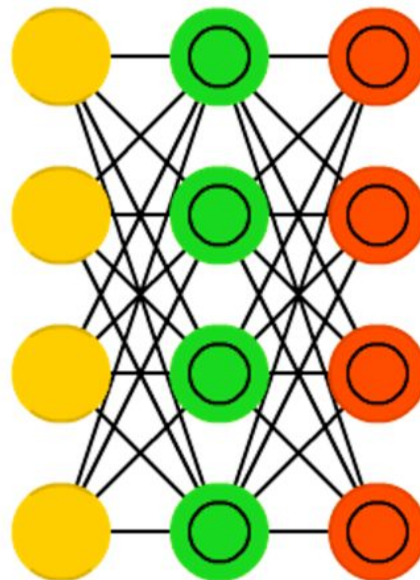
Auto-encoder (AE)



Sparse AE (SAE)



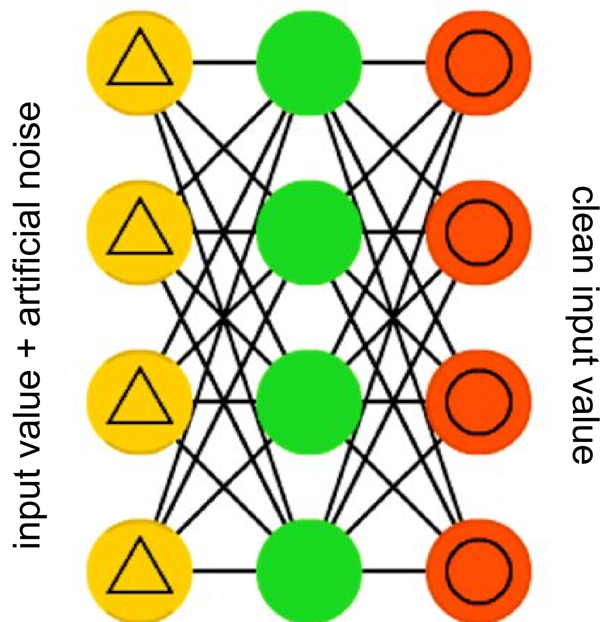
Variational AE (VAE)



- Backfed Input Cell
- Input Cell
- Noisy Input Cell
- Hidden Cell
- Probablistic Hidden Cell
- Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- Different Memory Cell
- Kernel
- Convolution or Pool

Compressing and diluting data

Variants of AEs are used to learn more sparse representations that are resistant to noise, or to handle probability distributions instead of values



Denoising auto-encoders (DAE) are particularly useful in training networks that will react graciously to “dirty” never-seen before input

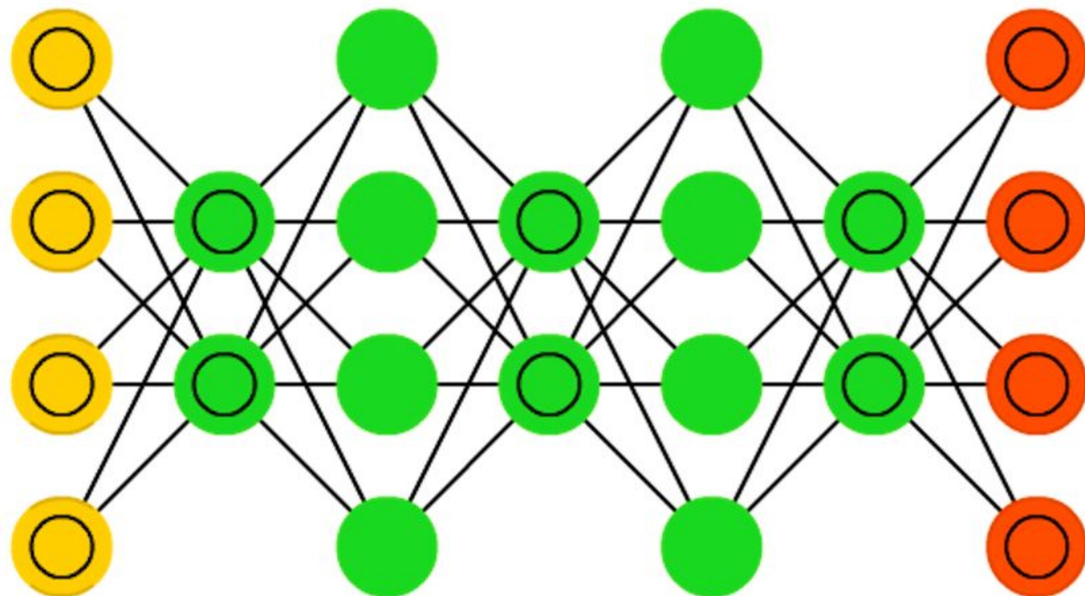
They learn to remove noise, and produce clean data

Useful for subsequent processing

Backfed Input Cell
Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Different Memory Cell
Kernel
Convolution or Pool

Deep belief networks

DBN are stacked architectures of RBMs and AEs. Each layer can be trained independently, and find a good encoding for the information provided by the previous layer, and for the needs of the subsequent layer.



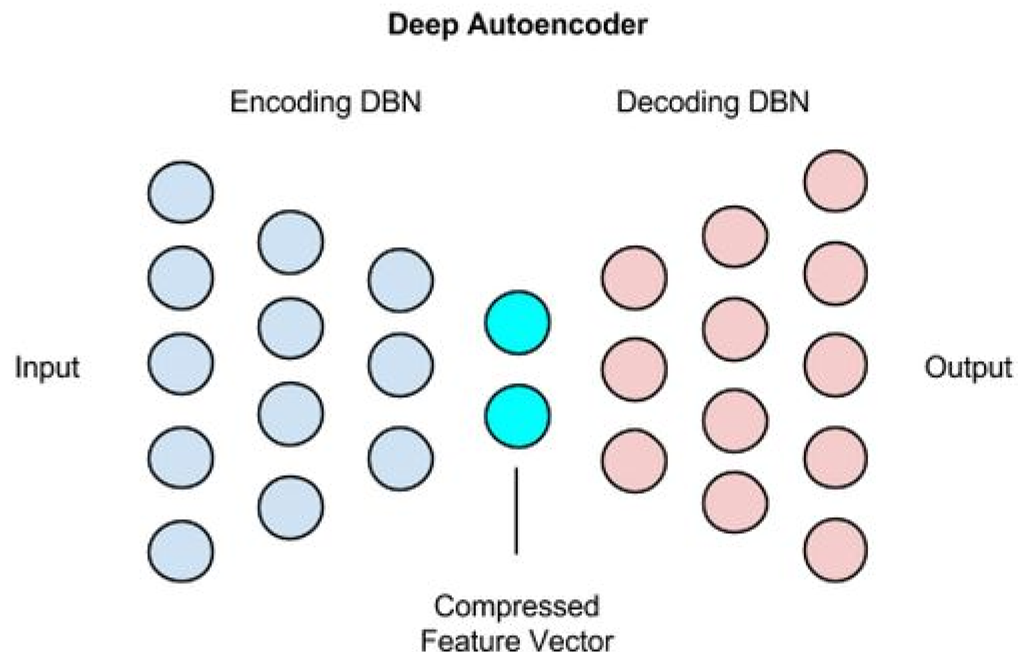
- Backfed Input Cell
- Input Cell
- Noisy Input Cell
- Hidden Cell
- Probablistic Hidden Cell
- Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- Different Memory Cell
- Kernel
- Convolution or Pool

Semantic hashing

Deep AEs can be used to provide a compact representation for very long and complex data

- Example: a whole image to a 20-numbers vector

The compressed representation can then be used for searching and matching



The compressed code is semantic in nature

Figure from deeplearning4j user manual

Applications

An old favourite of mine: **find similar requirement**

A new incoming requirement *r* is auto-encoded via AEs or DAEs to a small feature vector (FV)

The FV for *r* is compared, on some distance, with the FVs of all previously entered requirements

The closest FVs lead to the most similar requirements

- Old approaches based on lexical similarity
- Here, based on semantic hashing

Applications

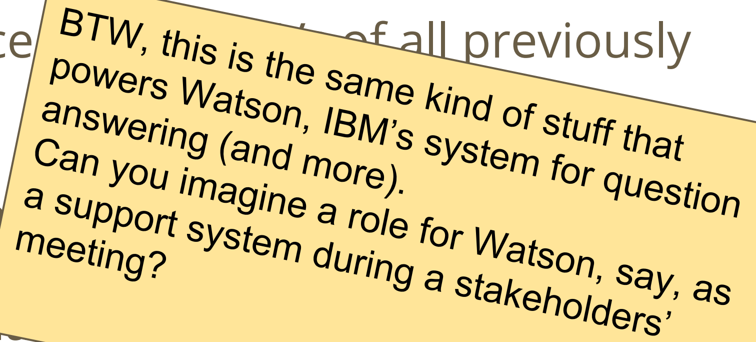
An old favourite of mine: **find similar requirement**

A new incoming requirement *r* is auto-encoded via AEs or DAEs to a small feature vector (FV)

The FV for *r* is compared, on some distance [...] of all previously entered requirements

The closest FVs lead to the most similar re[...]

- Old approaches based on lexical simi[...]
- Here, based on semantic hashing



Finding related stuff

Searching for semantically-related artifacts is the basic idea for **traceability**

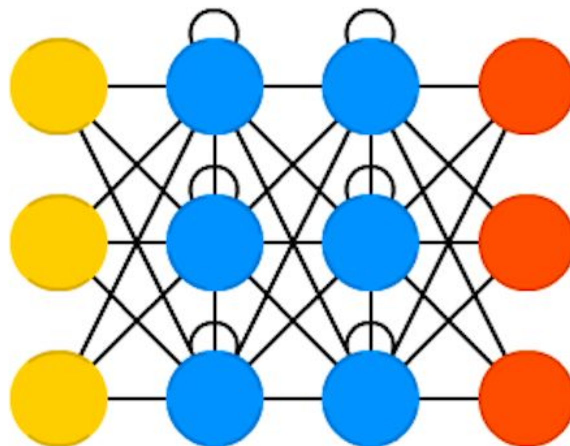
We can ask traceability questions in many ways, tho:

1. Given a pair of artifacts (a,b), tell me if they should be linked or not
   - → a **classification** problem (but usually the answer is “no” in 98% of the cases, bias)
2. Given an artifact a and a set of artifacts B, tell me which $b \in B$ should be linked to a
   - → a **search** problem
3. Given that I just linked a to b, tell me which a' (or which (a',b')) I should consider next for linking
   - a **prioritization** problem

Handling sequences

When problems have a sequential structure (as in our priority problem), basic NNs are no longer fit for service

- They have only a “stored memory” that is created at training time
- Each new instance presented to the NN treated as an individual case



Instead, we have to resort to **recurrent neural networks (RNN)**

- Backfed Input Cell
- Input Cell
- Noisy Input Cell
- Hidden Cell
- Probablistic Hidden Cell
- Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- Different Memory Cell
- Kernel
- Convolution or Pool

Handling sequences

In RNN, special recurrent cells provide a form of memory

They get as input both the (weighted) values from the input cells, and the (weighted) value from their own previous state

Hence, the results of a step are influenced by the partial results from all preceding steps

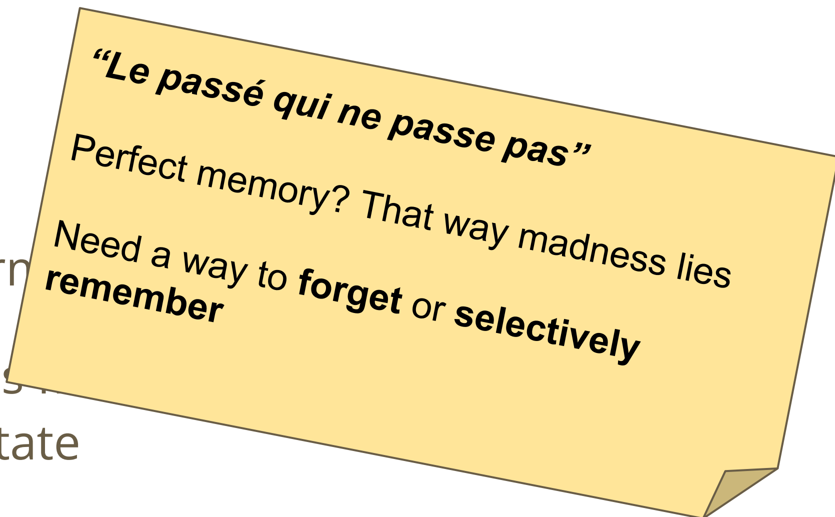
Problem: if **each sequence** becomes unique, the NN cannot learn

- “vanishing gradient” / “exploding gradient” problem

Handling sequences

In RNN, special recurrent cells provide a form

They get as input both the (weighted) values
(weighted) value from their own previous state



Hence, the results of a step are influenced by the partial results from all preceding steps

Problem: if **each sequence** becomes unique, the NN cannot learn

- "vanishing gradient" / "exploding gradient" problem

“Le passé qui ne passe pas”

In a tool that is assisting me in visiting a requirements base, while I establish links for traceability, it is perfectly sensible to “remember” what I have done 5 or 10 minutes ago. Less so, what I have done 10 days ago.

A simple temporal decay is not sufficient: certain things should be remembered forever, other are irrelevant after 10 minutes.

Solution: **Long Short-Term Memory (LSTM)** or **Gated Recurrent Units (GRU)**

- LSTM store value, but also have gates that decide when the value should be updated, propagated, or reset
- when to open or close these gates is learned by the RNN as part of training

LSTM RNN

LSTM RNN have been shown to be able to extract *incredible* sense from linear structures

Most typically: **text** (a sequence of symbols), which accidentally includes most requirements documents

My favourite piece on the subject:

- Andrej Karpathy, *The Unreasonable Effectiveness of Recurrent Neural Networks*, May 2015 (http://karpathy.github.io/2015/05/21/rnn-effectiveness/)

Example: Cybernetic Humour

Corpus: 11.000 jokes (around 1.5Mb of ASCII text, **small**!)

Network: LSTM RNN with 3 (4) hidden layers, 512 cells per layer, mostly standard hyperparameters

No embedded or encoded knowledge about lexicon, syntax, tokenization, lemmatization, POS, ontology... just a long sequence of 1.500.000 8-bit values.

What did the network learn?

D. Bacciu, V. Gervasi, G. Prencipe, ☺*: An investigation into cybernetic humour, or: Can Machines Laugh?*, FUN with Algorithms, 2016

Language learning

Our NN learned enough of English (lexicon and syntax) and enough about the rethorical structure of jokes that it could **generate** instances such as:

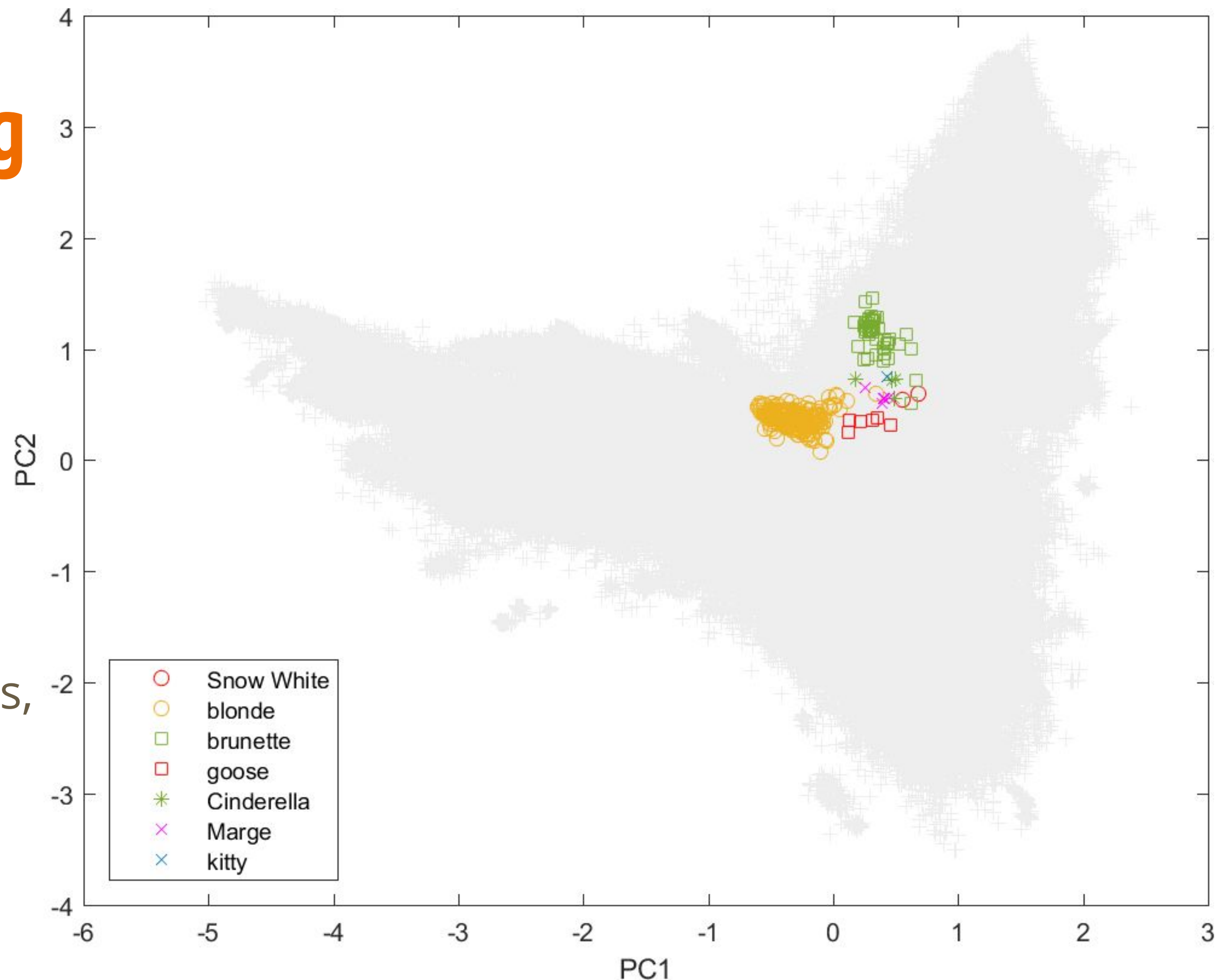
What do you get if you cross a famous california little boy with an elephant for players?
Market holes.

(Puctuation, spaces, and newlines generated as well)

Concept Learning

It also learned enough about the way we use sequences of symbols (words) in dirty jokes

Q: what can be (and cannot be) learned from requirements, user stories, use cases, etc.?



Learning programming

A basic LSTM RNN as the one we used, trained on 474 MB of C code (the Linux source tree) can generate pretty plausible-looking code.

Notice in particular how comments and coding style are learned from the samples and generated accordingly

```
/*
 * Increment the size file of the new incorrect UI_FILTER group information
 * of the size generatively.
 */
static int indicate_policy(void)
{
  int error;
  if (fd == MARN_EPT) {
    /*
     * The kernel blank will coeld it to userspace.
     */
    if (ss->segment < mem_total)
      unblock_graph_and_set_blocked();
    else
      ret = 1;
    goto bail;
  }
  segaddr = in_SB(in.addr);
  selector = seg / 16;
  setup_works = true;
  for (i = 0; i < blocks; i++) {
    seq = buf[i++];
    bpf = bd->bd.next + i * search;
    if (fd) {
      current = blocked;
    }
  }
  rw->name = "Getjbbregs";
  bprm_self_clearl(&iv->version);
  regs->new = blocks[(BPF_STATS << info->historidac)] | PFMR_CLOBATHINC_SECONDS
<< 12;
  return segtable;
}
```

More complex structures

It is not uncommon for a problem to have a more complex structure which cannot easily be reduced to a sequence

Example: a traceability matrix or graph could well be considered in its entirety, rather than as a set of unrelated (*a*,*b*) links.

Indeed, a traceability matrix could appear as a B/W bitmap (or a color bitmap, if linking information is more nuanced)

(Deep) Convolutional NN - (D)CNN

In a **CNN**, each element of the input structure (the focus) is fed to the network together with its immediate neighbours (the window).

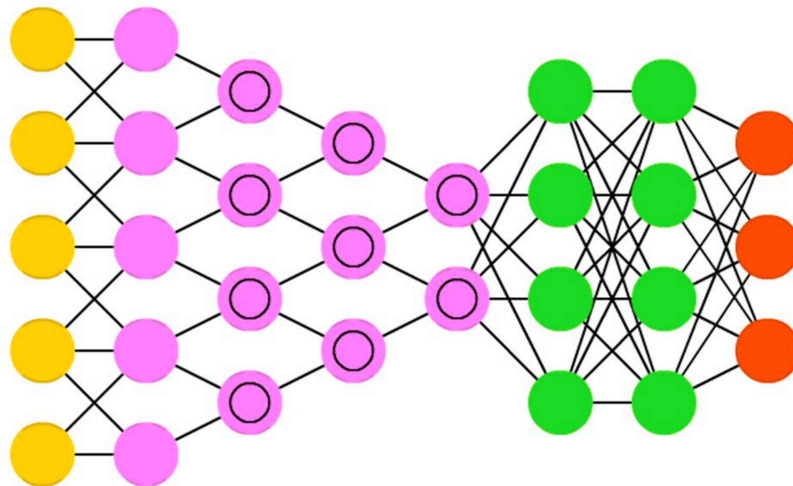
The focus is then shifted to the next element, thus inducing another window (where the previous focus would appear), and the process is repeated

Backfed Input Cell
Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Different Memory Cell
Kernel
Convolution or Pool

(Deep) Convolutional NN - (D)CNN

In a **Deep CNN**, the process is iterated multiple times; each layer perform a convolution on the output of the layer immediately preceding.

In other words, the first layer may look at 4x4 blocks, and the next up would look at 4x4 blocks of results, hence identify features that appear at the 16x16 scale.

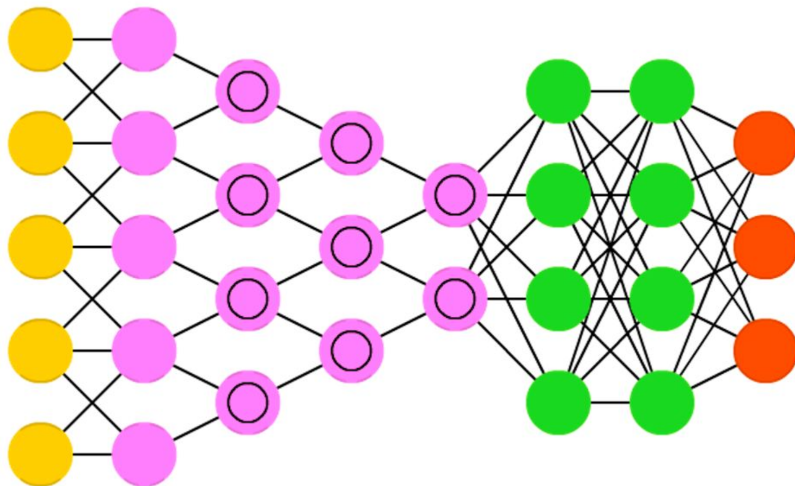


- Backfed Input Cell
- Input Cell
- Noisy Input Cell
- Hidden Cell
- Probablistic Hidden Cell
- Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- Different Memory Cell
- Kernel
- Convolution or Pool

(Deep) Convolutional NN - (D)CNN

In object recognition tasks, the final outputs are then fed to a classifier, tasked with identifying what is represented in an image.

Q: Can we imagine using a DCNN to identify “bad smells” in traceability structures?



Backfed Input Cell
Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Different Memory Cell
Kernel
Convolution or Pool

Understanding and re-generating

Another common problem in RE has to do with **refactoring** RE artifacts.

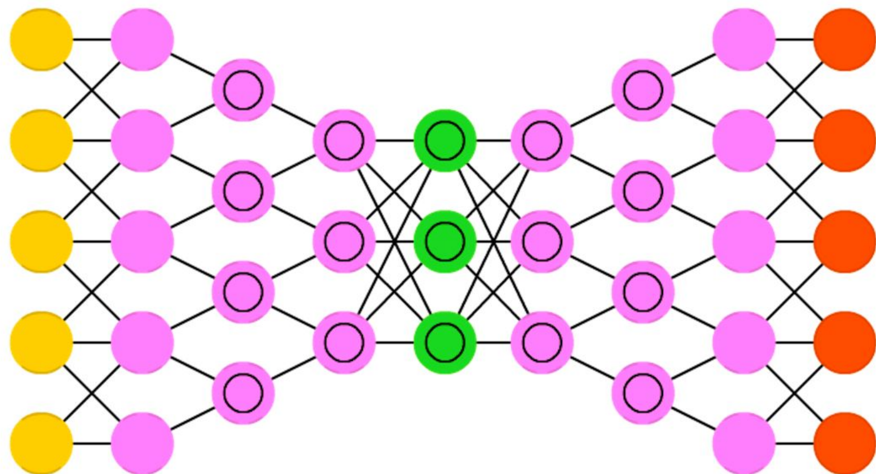
For example: I have a proposed requirement, written in customer-speak. I want to rewrite it in technical language.

Sounds like magic!

Deep convolutional inverse graphics networks (DCIGN)

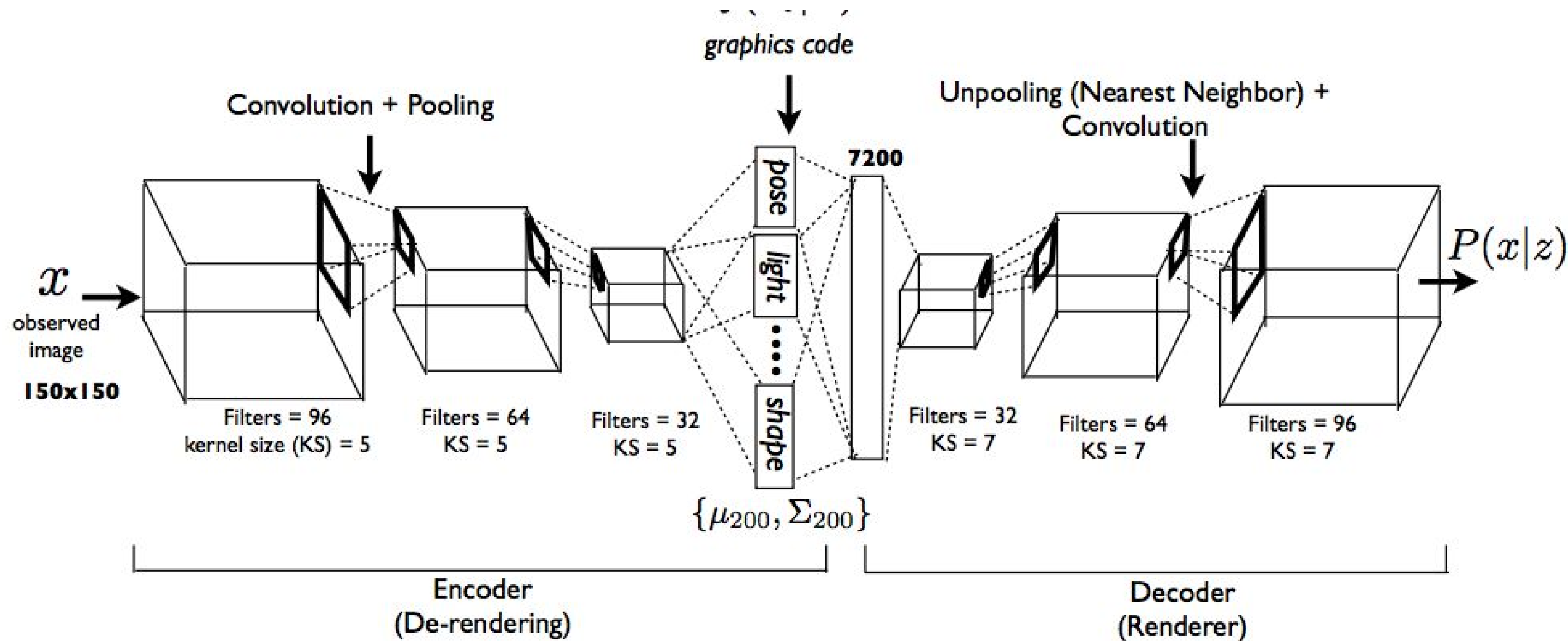
DCIGNs are built to “understand” the input (by encoding it in a FV, via convolution), and then generating an output from the FV (via deconvolution).

Similar idea to AEs, but now we can change the FV in a controlled way, and see what gets generated...



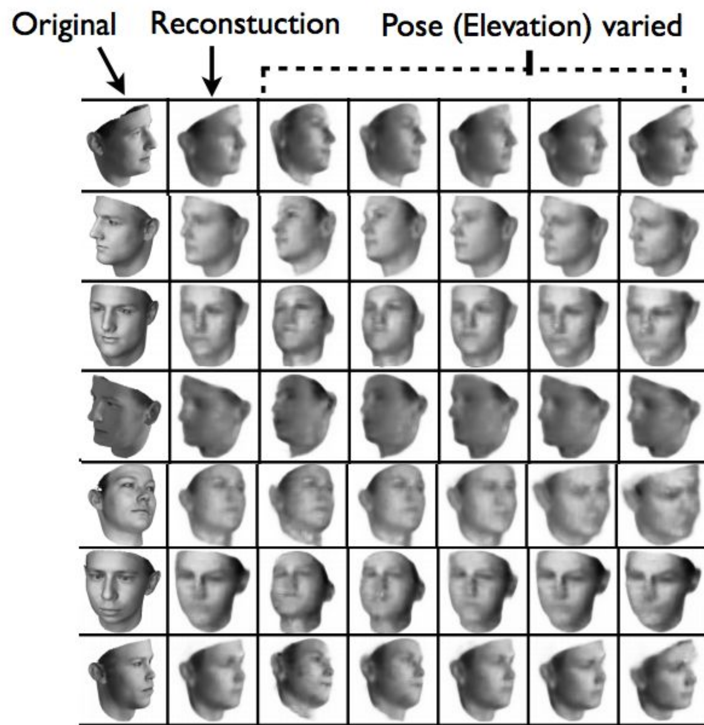
- Backfed Input Cell
- Input Cell
- Noisy Input Cell
- Hidden Cell
- Probablistic Hidden Cell
- Spiking Hidden Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- Different Memory Cell
- Kernel
- Convolution or Pool

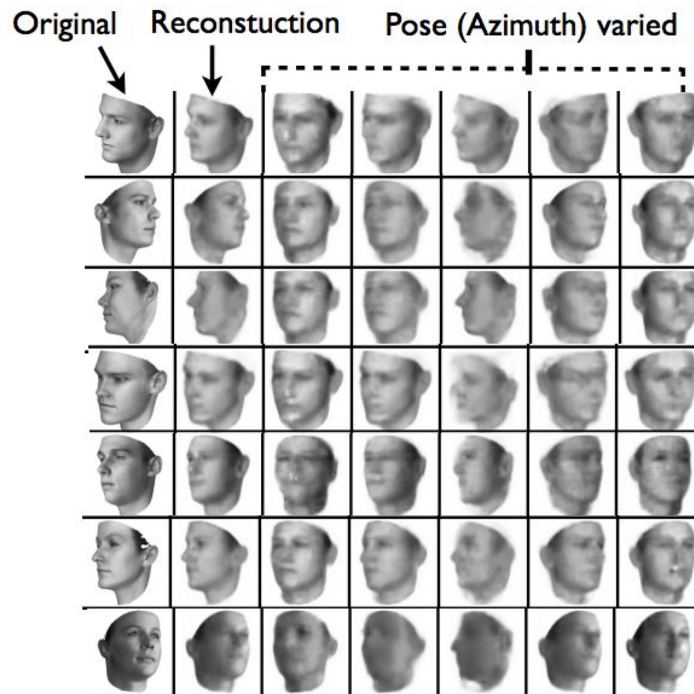
DCIGN in graphics



T.D. Kulkarni, W. Whitney, P. Kohli, J.B. Tenenbaum, *Deep Convolutional Inverse Graphics Network*

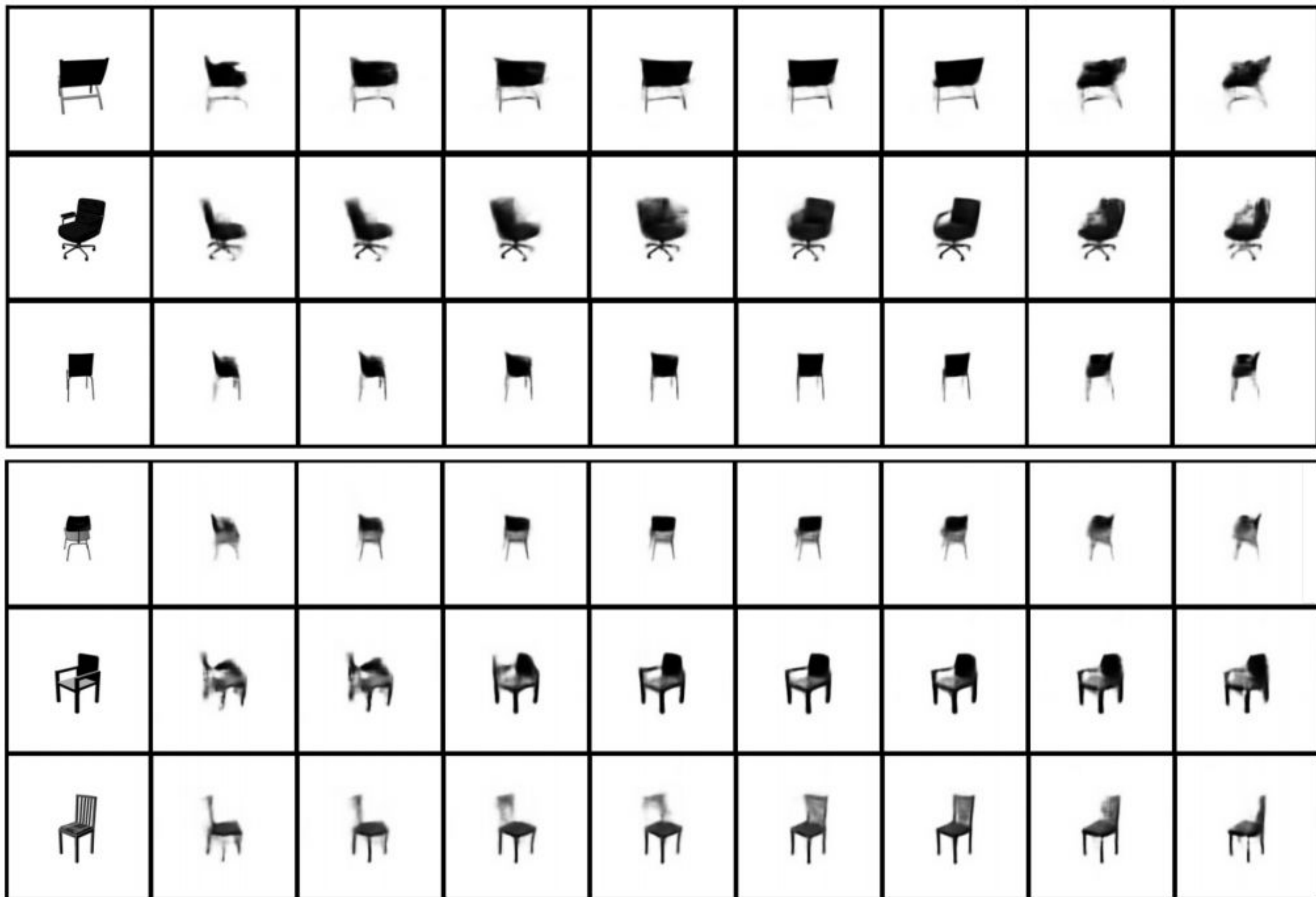
DCIGN in graphics





T.D. Kulkarni, W. Whitney, P. Kohli, J.B. Tenenbaum, *Deep Convolutional Inverse Graphics Network*

DCIGN in graphics



T.D. Kulkarni, W. Whitney, P. Kohli, J.B. Tenenbaum, *Deep Convolutional Inverse Graphics Network*

DCIGN in RE

In the experiments above, the NN had learned features such as *elevation*, *azimuth*, *pose*, *shape*

Q: what kind of **interpretable features** would such a network learn on RE artifacts of various kinds?

Q: what use could we imagine for **generated artifacts** obtained by manipulating specific features?

Q: is really RE so **ill-conditioned** that fuzziness in reconstruction would make any use impossible? (bonus point: even as creativity prompt?)

Attention!

RNNs work on linear sequences, whereas CNNs work on fixed-shape neighbourhoods…

… but this is not how the mind of a Requirements Engineer works!

Rather, we tend to follow **threads of interest** in the richly interwoven fabric of conflicting constraints and desires that make up the bulk of RE.

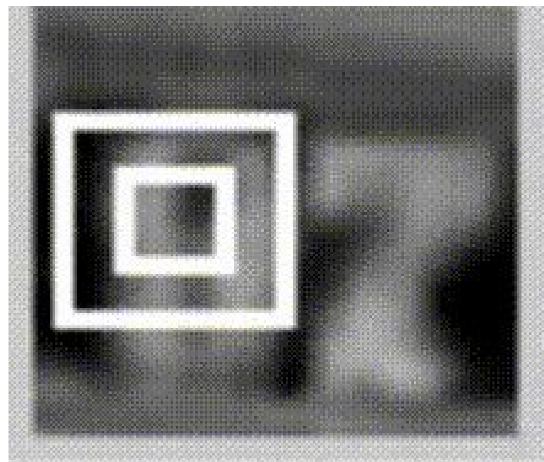
Can a NN be taught to identify and follow such threads in a bulk of complex material?

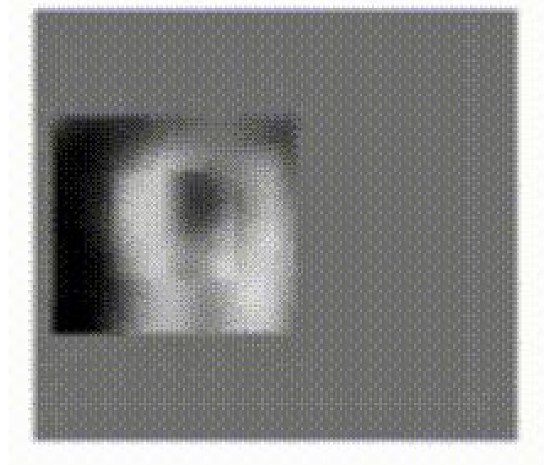
Attention!

Indeed, by buiding a NN architecture that includes a RNN, which directs a CNN to where to look next, we can simulate such a thread of interest.

Again, this works in graphics.

Q: would that work in RE artifacts?





Conclusions

There are many **forms** of NN, each apt to help with certain problems based on their underlying **structures**

Patterns, features, sequences, codes, graphs, ...

Researchers in AIRE have to familiarize themselves with the tools available

- which is not that easy: formal subtleties and optimization pitfalls abund
- on the other hand, we can ask ourselves questions that border on sci-fi

Looking for some answer in AIRE'18!