Lessons Learned Building the Neural GPU

What I learned building NNs to learn algorithms?

Łukasz Kaiser
2/24/17

What's That?

Can **you** recognize these patterns?

Should we expect a computer to do so?

- All patterns? What are patterns?
- Simple computer? Advanced?
- Does is require a search?
- Can the training data have noise?
- In what time should we do it?
- Can a neural network learn it?
- Just with gradient descent?

Pattern 1:

```
i: 10 i: 11 i: 10110   i: 1000  i: 1101
o: 01 o: 11 o: 01101   o: 0001  o: 1011
```

Pattern 2:

```
i: 1_ i: 10__ i: 101___  i: 1000____
o: 11 o: 1010 o: 101101  o: 10001000
```

Pattern 3:

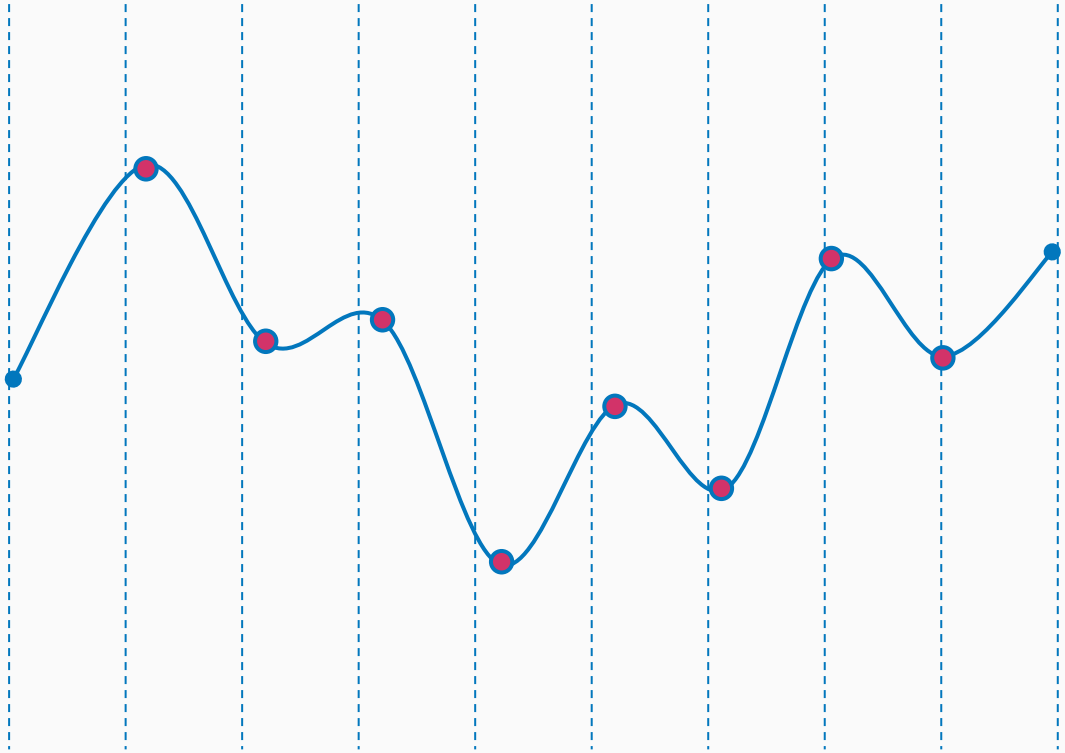
```
i: 121 i: 10210 i: 01210  i: 01201  i:11201
o: 1__ o: 1____ o: 01___  o: 001__  o:011__

i: 11211 i: 1012100 i: 1012010  i: 011020101
o: 101__ o: 101____ o: 0101___  o: 00111____
```

Gradient Descent?

Gradient descent is the standard way of training neural networks and many other forms of machine learning.

But it is a very basic method that only finds local minima. It is counter-intuitive to apply it to combinatorial problems. But very high-dimensional spaces are often counter-intuitive and proper representations can make gradient descent applicable to combinatorial problems too.

But how can gradient descent do combinatorics?



Computing with Neural Networks

Basic feed-forward neural networks operate on fixed-size vectors, so their power to generalize is limited.

Recurrent neural networks lift this major limitation, but their architecture restricts their power in other ways.

Feed-forward neural network:

- fixed-size input
- fixed-size output
- number of parameters depends on these sizes
- limited computational power

Recurrent neural network:

- variable-size input
- variable-size output
- number of parameters depends on **memory size**
- computational power limited by this memory size

Great success in speech recognition and language processing!

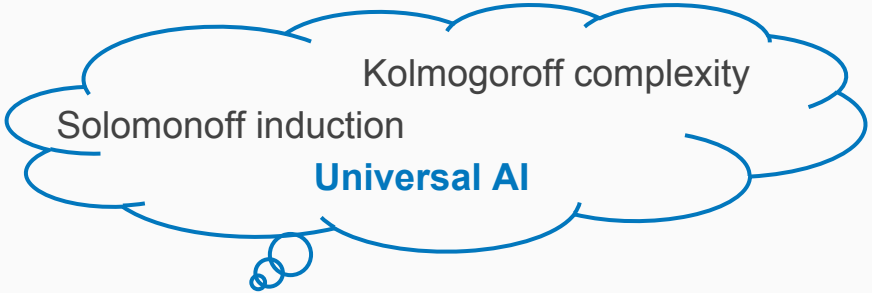
Why Learn Algorithms?

Algorithms are universal patterns, so if we can learn them, then in principle we can learn everything. If we can't learn basic ones, then we can't hope to build universal machine intelligence.

Can't recurrent neural networks do it?

- Only if testing length ~ training.
- But with higher length we need to increase network capacity (number of parameters).
- Fail to capture algorithmic patterns, even in principle.

The big dream of universal learning.



Previous work on neural algorithm learning.

- RNN and LSTM-based, generalizes a bit with attention.
- Data-structure based (stack RNN, memory nets).
- Neural Turing Machines.

Problems with Neural Turing Machines.

- Look close, your computer does not have a tape!
- Very sequential in nature, hard to train on hard problems.
- Example problem: **long multiplication.**

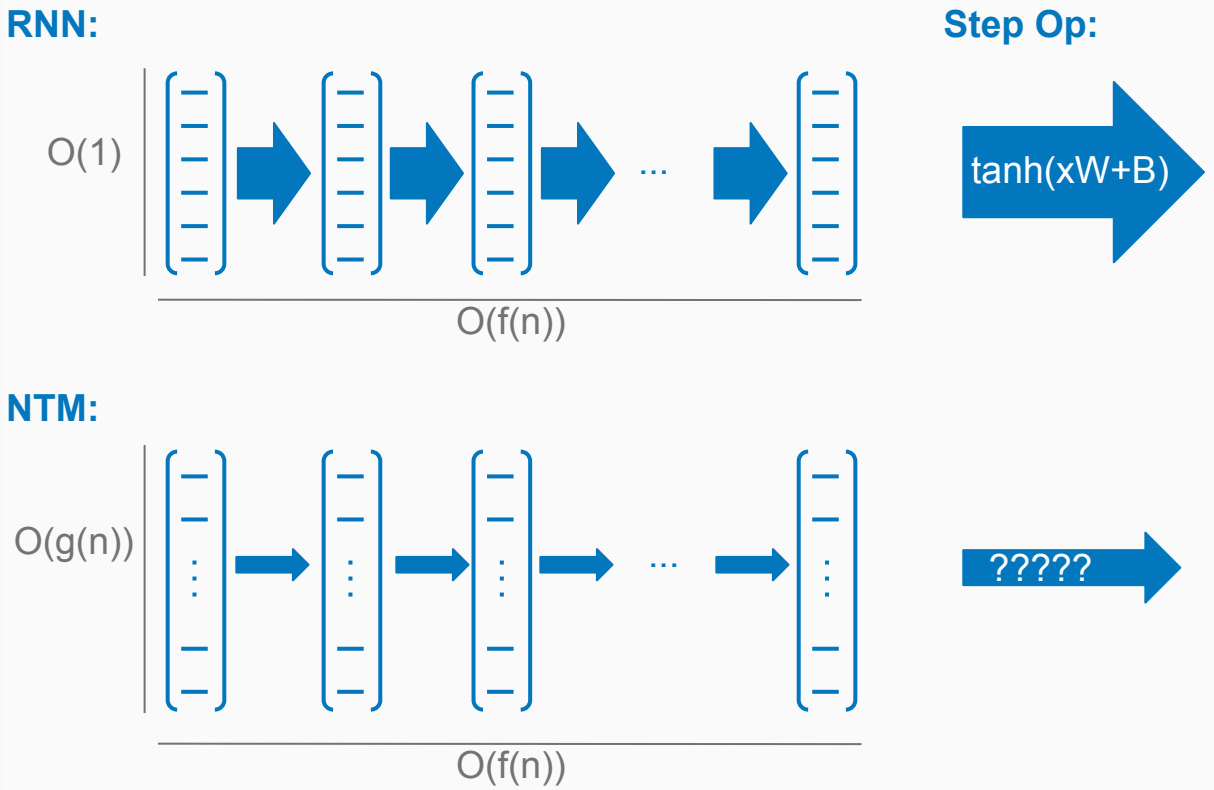
Neural Computation

Recurrent Neural Networks share weights across time steps but use fixed-size memory vectors.

Time: O(f(n)) Space: O(1)

Neural Turing Machines work with variable-size memory and so can do general computation, but are sequential.

Time: O(f(n)) Space: O(g(n))

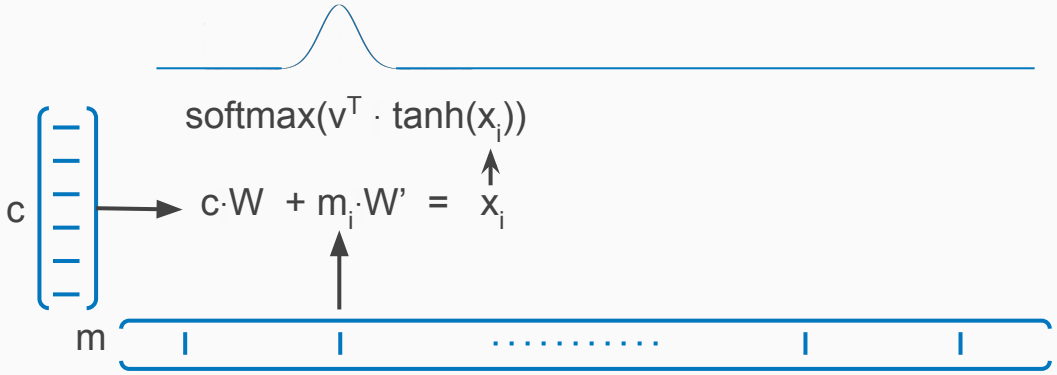


Arbitrary Size Ops

Neural computation requires operators that use a **fixed number of parameters** but operate on **memory of arbitrary size**. There are a few candidates:

1) Attention (Neural Turing Machine)
2) Stack, (De)Queue, Lists
3) … other memory structures ...
4) Convolutions!

Attention mechanism:



Why convolutions? They act like a **neural GPU**.

- Attention is a softmax, effectively ~1 memory item / step.
- Similarly a stack and other task-specific memory structures.
- Convolutions affect all memory items in each step.
- Convolutions are already implemented and optimized.
- To train well we need to use LSTM-style gates: **CGRN**.

Neural GPU

Convolutional Gated Recurrent Networks (CGRNs) perform many parallel operations in each step, akin to a **neural GPU** and in contrast to the sequential nature of Neural Turing Machines.

The definition of a CGRN is also very **simple and elegant**, see on the right!

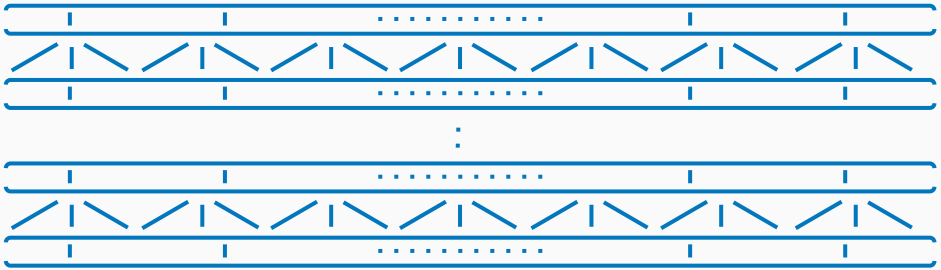
CGRU definition: (similar to GRU replacing linear by convolution)

$$s_{t+1} = g \cdot s_t + (1 - g) \cdot c \quad \text{where:}$$

$$g = \text{sigmoid}(\text{conv}(s_t, K_g))$$

$$c = \tanh(\text{conv}(s_t \cdot r, K_c))$$

$$r = \text{sigmoid}(\text{conv}(s_t, K_r))$$



Computational power:

- Small number of parameters (3 kernels: K_g, K_c, K_r).
- Can simulate computation of cellular automata.
- With memory of size n can do n local operations / step.
- E.g., can do long multiplication in O(n) steps.

Neural GPU

Convolutional Gated Recurrent Networks (CGRNs) perform many parallel operations in each step, akin to a **neural GPU** or a cellular automaton.

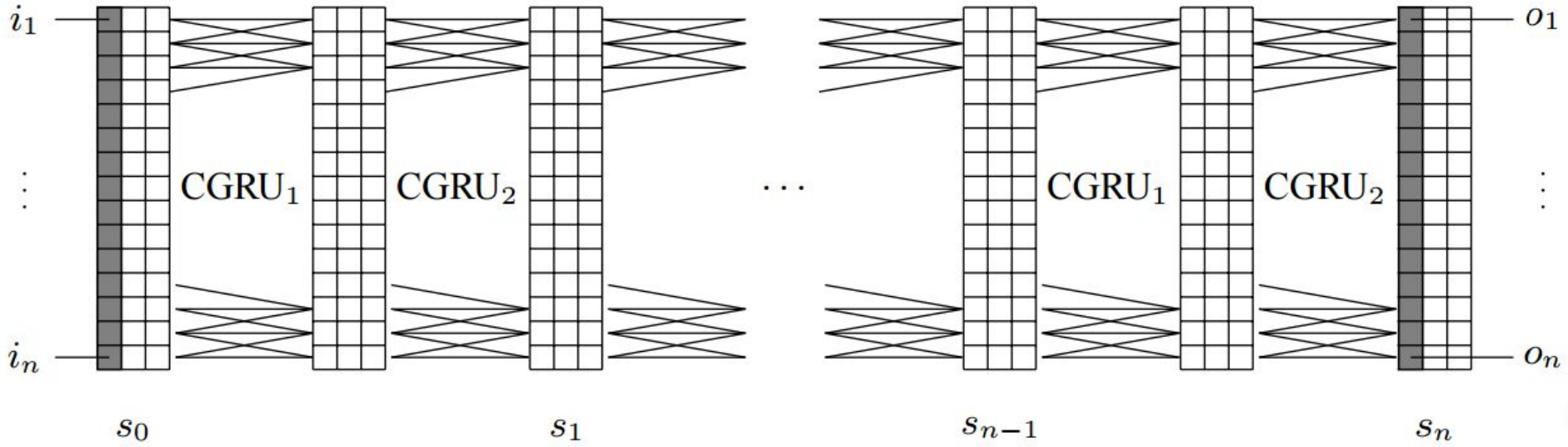
CGRU definition: (similar to GRU replacing linear by convolution)

$$m_{t+1} = g \cdot m_t + (1 - g) \cdot c \quad \text{where:}$$

$$g = \text{sigmoid}(\text{conv}(m_t, K_g))$$

$$c = \tanh(\text{conv}(m_t \cdot r, K_c))$$

$$r = \text{sigmoid}(\text{conv}(m_t, K_r))$$



Neural GPU
Results

A n-step n-size memory CGRN achieves **100x generalization** on a number of tasks such as reversing sequences, long addition, or even non-linear-time **long multiplication**.

We say an **output is correct only if all its digits are right**. So an error of 40% means that there are still 60% of fully correct sequences on output.

Reversing:

i: 3 0 8 4
o: 4 8 0 3
(all examples for length=4)

Length	Neural GPU Error	LSTM+A Error
10*	0%	0%
20*	0%	0%
60	0%	7%

Long decimal addition:

i: 4 5 3 0 OP_+ 0 9 9 5
o: 4 4 3 6
(OP_+, OP_* are input characters)

10*	0%	34%
20*	0%	100%
25	0%	100%
2000	0%	100%

Long binary multiplication:

i: 0 1 1 0 OP_* 0 1 0 1
o: 0 0 1 1 1
(6 * 10 = 60 = 32+16+8+4)

10*	0%	91%
20*	0%	100%
25	0%	100%
2000	0%	100%

* Training upto length 20.

Tricks of the Trade

A number of techniques are needed to make the training of a Neural GPU work well, and some are required for the generalization to work or to be stable.

Curriculum learning.

- Start training on small lengths, increase when learned.

Parameter sharing relaxation.

- Allow to do different things in different time-steps first.
- Not needed now with bigger models and orthogonal init.

Dropout on recurrent connections.

- Randomly set 10% of state vectors to 0 in each step.
- Interestingly, this is key for good length generalization.

Noise added to gradients.

- Add small gaussian noise to gradients in each training step.

Gate cutoff (saturation).

- Instead of sigmoid(x) use $[1.2\text{sigmoid}(x) - 0.1]_{[0,1]}$.

Tune parameters. Tune, tune, tune.

- A lot of GPUs running for a few months; or: better method!

How to code this?

A few issues that come up.

Illustrate each one with code.

Is the graph static or dynamic?

- Dynamic ops were not exposed in first TF release, tricky.
- Static graph requires bucketing and takes long to build.
- **Focus:** why conditionals are tricky: **batches and lambdas.**

How do we do bucketing?

- Sparse or dense buckets? Masks are bug-prone.
- How do we bucket training data? Feed or queues?
- If queues, how to do curriculum? How to not starve?

How do we write layers?

- Is there a canonical way to define new functions?
- Frameworks: Keras vs Slim (OO vs functional)
- Unification in tf.layers: callable objects save scope.
- **Example:** weight-normalization through **custom_getter.**

How do we organize experiments?

- Use tf.learn, Estimator, Experiment.
- How to registering models and problems? Save runs?
- Hyper-parameters manual or tuned with ranges?

Thank You

Joint work with Ilya Sutskever, Samy Bengio, friends in Brain.

Exercises:

- Write your trainer framework.
- Focus on design first, not any particular performance metric!
- Train some interesting model, maybe the Neural GPU.