Differentiable Neural Computers and their abilities

Report by:
Denys Kovalenko
Gagandeep Singh

University of Tartu 2017

Introduction **3**

Theory behind Differential Neural Computers **4**

Turing machine 4

Artificial Neural network 5

Recurrent Neural Network 5

DNC - Differential Neural Computer 6

Explanation of how DNC works on “copy” example **7**

Training and testing on real problem 9

Conclusions **10**

Introduction

Machine learning have seen many breakthroughs in recent years due to reduced cost of computing power and enhanced algorithms. But in most cases foundation of modern machine learning approaches was developed at least a decade ago. This project aims to cast light to new concept in machine learning - enhance possibilities of neural networks with external memory. Maybe this has roots in fact that people have always wanted to be able to connect external devices to brain and learn new languages, skills instantly by “downloading” data into their brain. This is still far from reality, but DeepMind shows that they are working on similar problem in artificial neural networks world - “let’s connect external memory that will work like RAM to ANN and see what happens”. In this combination recurrent neural net performs like algorithm, which can be learned with gradient descent, and it can write to external memory and read from it, just like CPUs in computers work with RAM.

Goal of this project is to find out for what Differential Neural Computers can be used for, provide understanding how they can be trained and how they perform tasks. Also, we are briefly explaining idea that lays behind DNC.

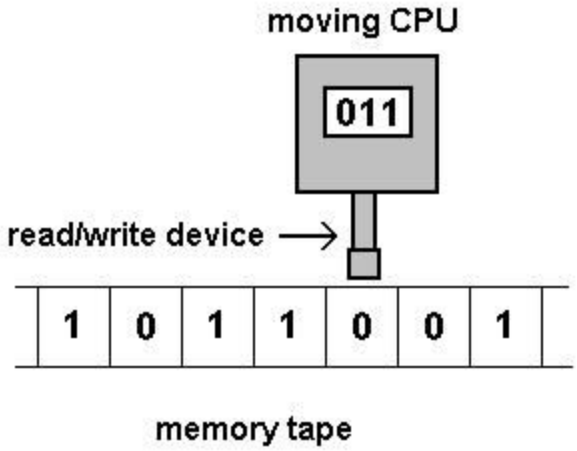
Finally, we will demonstrate how DNC performs on real data, and describe characteristics related to functioning of it’s memory, and describe limitations of DNCs.

Theory behind Differential Neural Computers

To get better understanding of DNCs we should provide brief overview of key components that forms foundation of what DNC is - Turing Machine and Recurrent Neural networks.

Turing machine

Turing machine is abstract machine, represented by moving head that is connected to infinite tape, and can perform read/write operation, and move along this tape. Set of those operations is defined in finita table with instructions.



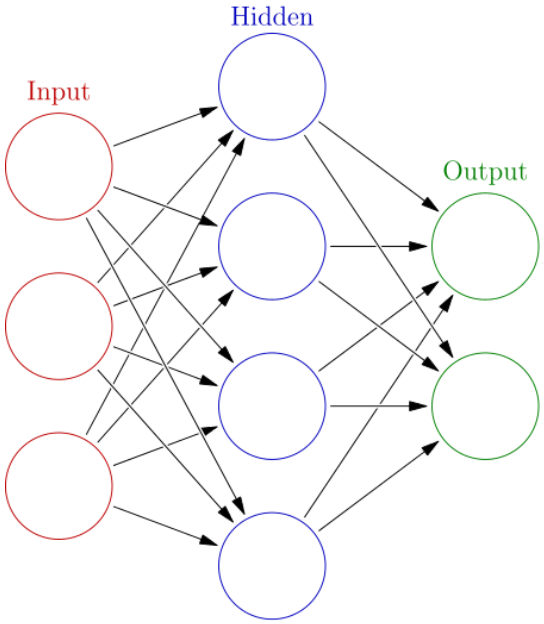
Representation of turing machine

Given any computer algorithm, such turing machine can be constructed that it will implement this algorithm. [1]

So, from another side, when we hear that programming language is Turing-complete it means that it can implement any algorithm that can be constructed in physical form or that can be represented by turing machine. And when we say some notations aren't turing complete (Petri nets or Context-free grammars[2], for instance) it means that they can't capture all those algorithms.

[1] https://en.wikipedia.org/wiki/Turing_machine
[2] https://en.wikipedia.org/wiki/Context-free_grammar

Artificial Neural network



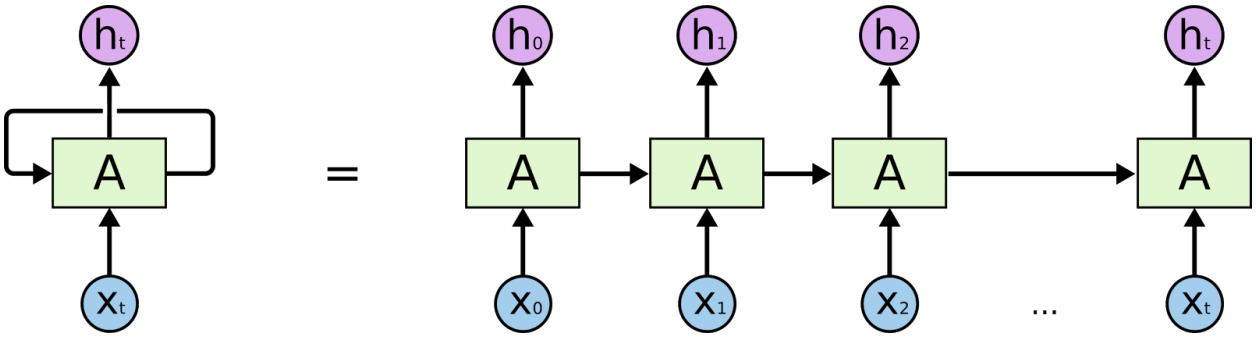
Feed-forward artificial neural network with 1 hidden layer

Artificial neural network is a computational model that mimics how neurons work in brain of humans and animals. It's heavily used in machine learning for image processing, speech recognition and others tasks for which no strict algorithm can be defined so it should be learned from datasets. "Knowledge" in ANN is represented in weights of connections between neurons.

Recurrent Neural Network[3]

One of the major shortcomings of ANN is that they do not have any way to store the previously learned knowledge, RNN address that problem, by creating directed cycles between neurons. This allows RNN to have "memory" and so to capture time. Each time RNN process information it leave trace inside this memory, that will affect output, possibly changing outputs produced by same inputs, but in different time. And RNN can use their internal memory to process arbitrary input sequences.

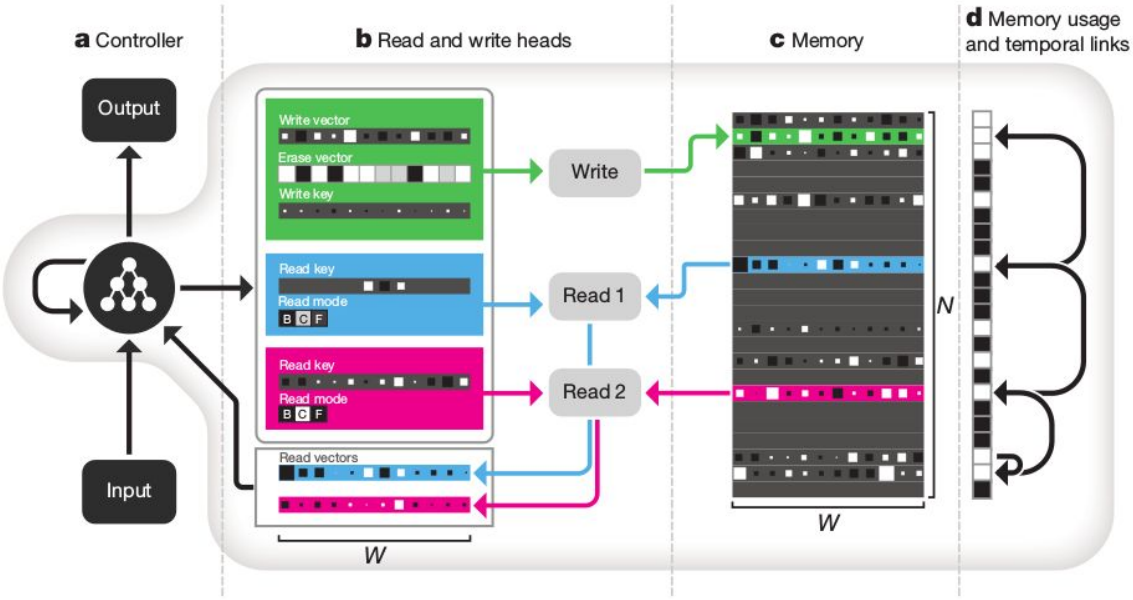
[3] http://colah.github.io/posts/2015-08-Understanding-LSTMs/



RNN element unfolded in time

DNC - Differential Neural Computer

DNC is a machine learning model, that have combined to approaches and took best from 2 different worlds - it can learn from data and neural networks, thus it can be trained via supervised learning with gradient descent to solve different task, and it has ability that all computers have - store information in variables and form complex structures, and store data over long time periods without interference.



Structure of differential neural computer[4]

DNC has Recurrent Neural Network which is called controller - it receives input sequence, produce output sequence and communicate with memory, which is fixed-size two dimensional matrix using so called read and write heads.

[4] Original article in Nature - http://www.nature.com/nature/journal/v538/n7626/full/nature20101.html

Differential Neural Computer is upgraded version of Neural Turing Machine, and it mainly differs by methods used to access external memory, and those mechanisms provide significant improvements in DNC.
In our practical part of research, we used implementation of NTM by Snips AI called NTM-Lasagna[5]. Main reason for that was that they had quite good explanation in their blog, had examples of copy, repeated copy, dyck-word tasks, and those tasks included basic visualisation. Unfortunately, choise of DNC libraries is not as wide as desired[6], so NTM-Lasagna does not capture all cool properties of DNC, but is good from getting hands on memory-augmented networks point of view.

So, how that works?
In DNC controller (RNN) receives input sequence, produce output and vectors parametrizing write head and read heads. Those parameters define "attention" of DNC - which cells will be updated and from which cells information will be read.

Those attention mechanisms as fascinating as hard to understand, however, authors of DNC article proudly imply that there are similarities between DNC memory mechanisms and mammalian hippocampus. Usage-based method of memory access is similar to how human recall things - mostly in the order they have learned it.

To highlight few differences, research article showed that in NTM memory access was similar to linked list, but it doesn't have mechanism to ensure that allocated block doesn't intersect with already used one, DNC, however can allocate single cell; NTM doesn't have mechanism to erase data so matrix can't be really reused. Also, when write head switch location read head couldn't find place of previous read. In DNC history of write head moves is stored.

DNC looks very promising for tasks when algorithm should be inferred from data, and when it's useful to store some data over long period of time without interference.

---

[5] NTM Lasagna description https://medium.com/snips-ai/ntm-lasagne-a-library-for-neural-turing-machines-in-lasagne-2cdce6837315#.dsng6vxcl

[6] Github repositiories mathcing DNC https://github.com/search?utf8=%E2%9C%93&q=dnc and Diffrerential Neural Computer - https://github.com/search?utf8=%E2%9C%93&q=Differentiable+Neural+Computer&type=Repositories&ref=searchresults

In our project we used much easier tasks to get feeling of how it works.

Explanation of how DNC works on “copy” example

Steps:

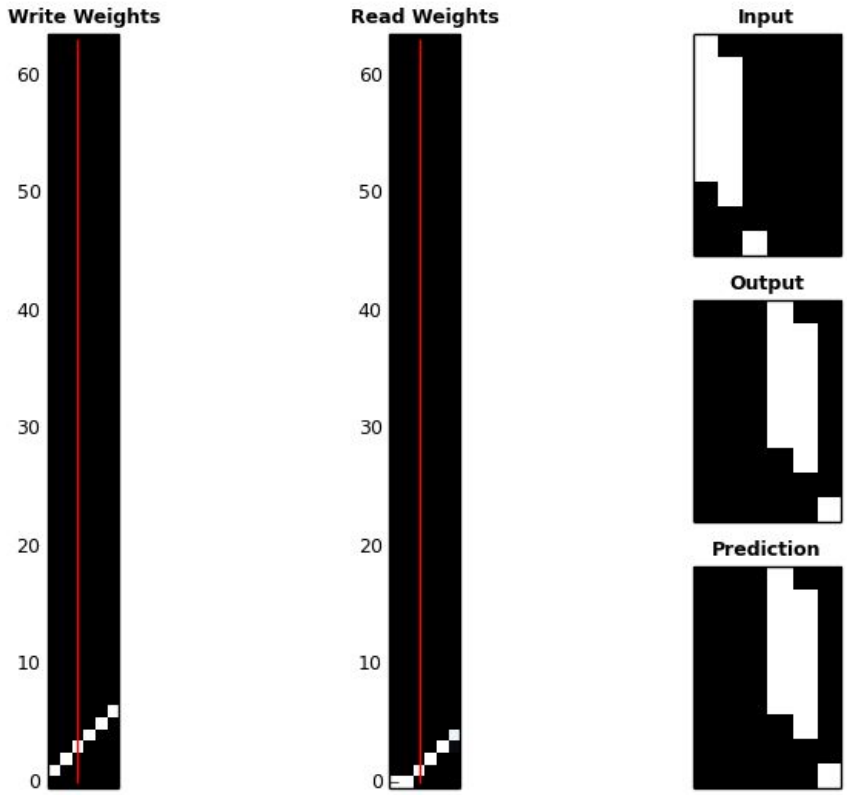
- We use a generator to create the training data(example_input & example_output).
- We create the DNC layer.
- We train the DNC on the training data.
- It learns from the data, how to copy.
- DNC performs the copy task on sample test.

So, at first let’s understand how data looks like. We tested NTM-Lasagna library, which is not DNC implementation, but for our purposes it behaves similarly. Data on outputs is represented as 2d binary matrix consisting of ones and zeros. Number of rows represents size of one portion of input data, and each column represents element of input sequence that was fed to NTM.
For copy task additional column and one row are used to represent “finish” flag.
If we take number of rows in one input element as N and number of elements in sequence as M (size and length in NTM Lasagna terminology), than dimensions of real matrixes of input and output data have size of N*2 + 1 and M+1. In input data matrix only first half rows (which are actually columns in graphical representation below) are filled with data, and similarly in output training data only second half of rows are filled.
For each training input-output pair random size of input sequence is chosen from given range, that ensures that DNC will be able to learn how to generalize algorithm for various datasets sizes.

As in our case we now algorithm that binds input and output data, we don’t need to collect data and we can generate it on the fly. This is an interesting concept implemented in NTM Lasagna Task interface. Dream of any data science person - finally possible to get as much data as you want.

Here is an example how NTM-Lasagna setup looks like:

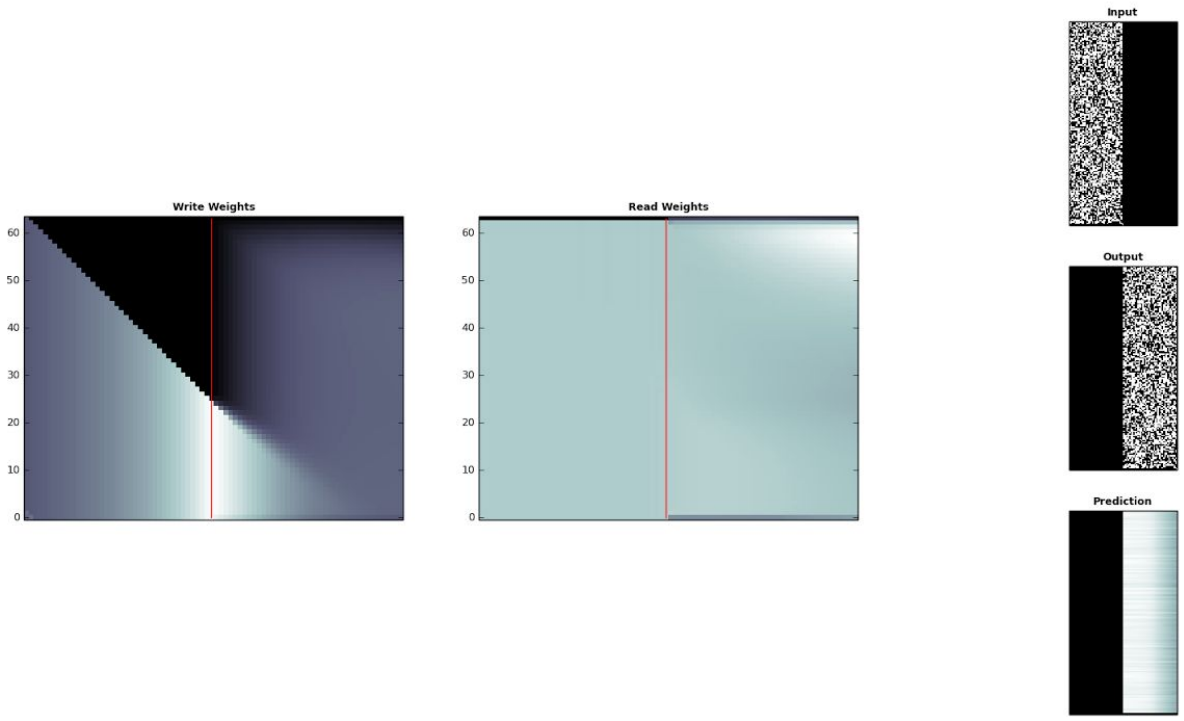
```
//define memory matrix properties
memory = Memory((128, 20), memory_init=lasagne.init.Constant(1e-6),
   learn_init=False, name='memory')
//RNN that works as controller
controller = DenseController(l_input, memory_shape=(128, 20),
   num_units=100, num_reads=1,
   nonlinearity=lasagne.nonlinearities.rectify,
   name='controller')
//define 1 read and 1 write had. NTM can have more read heads.
heads = [
   WriteHead(controller, num_shifts=3, memory_shape=(128, 20),
      nonlinearity_key=lasagne.nonlinearities.rectify,
      nonlinearity_add=lasagne.nonlinearities.rectify,
      learn_init=False, name='write'),

   ReadHead(controller, num_shifts=3, memory_shape=(128, 20),
      nonlinearity_key=lasagne.nonlinearities.rectify,
      learn_init=False, name='read')
]

l_ntm = NTMLayer(l_input, memory=memory, controller=controller,
heads=heads)
```



Result of running copy task with 64 by 10 memory matrix and 8 by <=5 input data

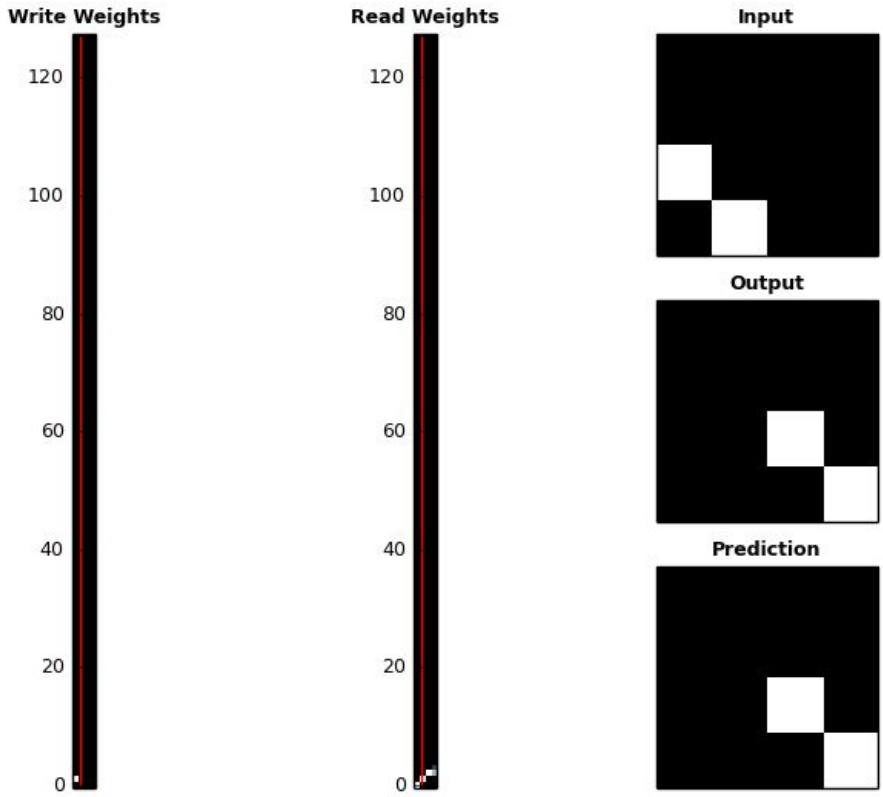
NTM works fine when we have much more memory that input sequence size. It copied matrix perfectly.

Note: to train model to to this it takes 7-10 minutes on core-i5 & 16gb ram.



Attempt to copy 100 by 50 matrix using 64 by 10 memory. Prediction looks strange

When we used same memory for large sequence (input element size - 100, 50 elements in sequence) prediction was somehow close to full mess.
Our observations showed that it takes at least 1.5 as much memory as input_size*input_length to work properly.
Also, training is really slow when it comes to long sequences and memory bigger that 100 by 100.



Perfect copy of small matrix

Training and testing on real problem

To test some of NTM properties we decided to pick sorting task, because it has few interesting properties:

- First, it's task simple enough so most humans including kids can do that intuitively, so that means that it can be learned with supervised learning
- Second, sorting is pure algorithm that doesn't need to be changed if input array grows (assuming that we have enough memory)
- Third, sorting is well-studied, there are many sophisticated algorithms including quick-sort, heap-sort, merge sort etc…, and we know properties for each of them ($O(n)$[7], required space)

[7] https://en.wikipedia.org/wiki/Big_O_notation

- Fourth, it's quite easy to interpret result - in our form of memory representation sorted set will look like non-decreasing sequence.

To do that we had to do following:
Define new python script to run task, in which we will use new generator. Generator - class implementing Task interface, and main method is **generate_sample(length)** which gives **input_data, output_data** pair. In our case, we needed to represent integer array using 2d binary matrix.
There was 2 ways:

1) Each columns is number in binary representation. But it would require more coding and will be definitely harder to train, as NTM will have to capture the idea that element on 1 position higher is twice as more "important" than it's neighbour.
2) In each column there will be only one non-zero element equal to one. So, number of rows will represent value range. 15 means we can capture values 0..14. And each column then will represent 1 element in array to be sorted.

We went with second approach.
Input data looked like this:
[1 0 0 0]
[0 1 0 1]
[0 0 1 0]
Which is equal to following array - [2;1;0;1]
And sorted array will look this way:
[0 0 0 1]
[0 1 1 0]
[1 0 0 0]
Corresponding to [0;1;1;2]

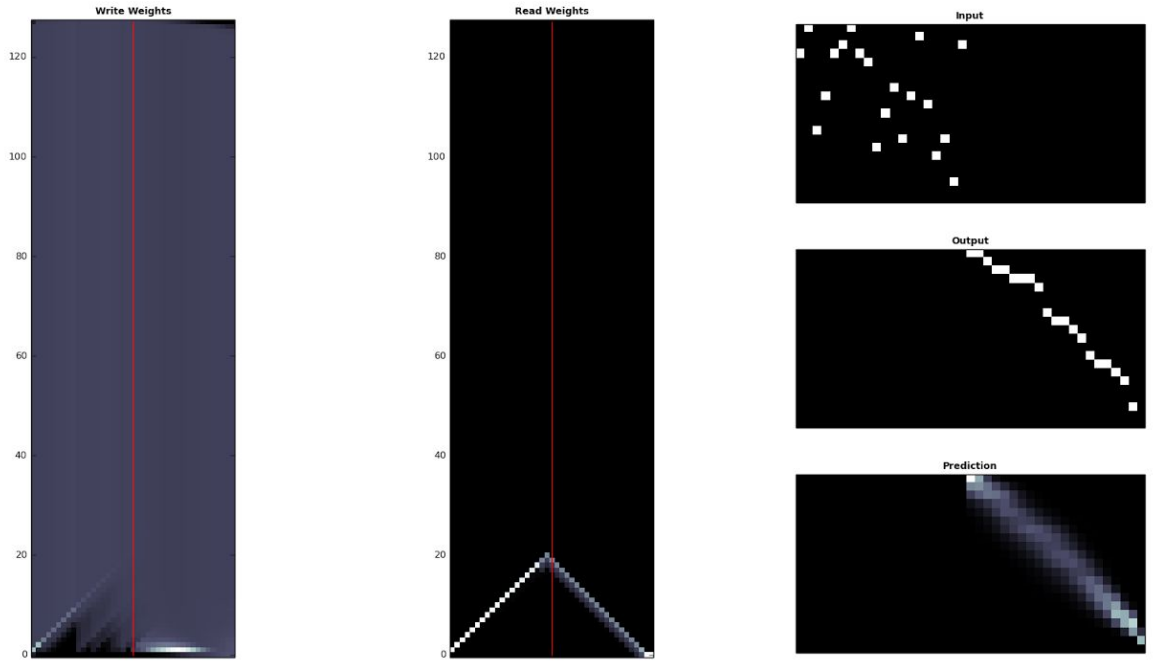
So we implemented generator that can create inputs and outputs that follows that rules for matrixes with given length and trained NTM on them.
We were using EENet grid as described in this course page[8] to train models. I don't know the reason, but training model in GPU just wouldn't work, when we ran it with following command: PYTHONPATH=. srun --partition=gpu --gres=gpu:1 --constraint=K20 python examples/copy-task.py

[8] https://courses.cs.ut.ee/2015/dmseminar/fall/Main/Keras

```
  File "/home/hpc_denis/venv_keras/src/theano/theano/gof/fg.py", line 481, in re
place
    str(reason))
TypeError: ('The type of the replacement must be compatible with the type of the
 original Variable.', Reshape{3}.0, Reshape{3}.0, TensorType(float32, (False, Tr
ue, True)), TensorType(float32, 3D), 'local_reshape_dimshuffle')

<<!! BUG IN FGRAPH.REPLACE OR A LISTENER !!>> <type 'exceptions.TypeError'> ('Th
e type of the replacement must be compatible with the type of the original Varia
ble.', Reshape{3}.0, Reshape{3}.0, TensorType(float32, (False, True, True)), Ten
sorType(float32, 3D), 'local_reshape_dimshuffle') local_reshape_dimshuffle
ERROR (theano.gof.opt): Optimization failure due to: local_reshape_dimshuffle
ERROR (theano.gof.opt): node: Reshape{3}(InplaceDimShuffle{0,x}.0, <TensorType(i
nt64, vector)>)
ERROR (theano.gof.opt): TRACEBACK:
ERROR (theano.gof.opt): Traceback (most recent call last):
  File "/home/hpc_denis/venv_keras/src/theano/theano/gof/opt.py", line 2004, in
process_node
    remove=remove)
  File "/home/hpc_denis/venv_keras/src/theano/theano/gof/toolbox.py", line 391,
in replace_all_validate_remove
    chk = fgraph.replace_all_validate(replacements, reason)
  File "/home/hpc_denis/venv_keras/src/theano/theano/gof/toolbox.py", line 340,
in replace_all_validate
    fgraph.replace(r, new_r, reason=reason, verbose=False)
  File "/home/hpc_denis/venv_keras/src/theano/theano/gof/fg.py", line 481, in re
place
    str(reason))
TypeError: ('The type of the replacement must be compatible with the type of the
 original Variable.', Reshape{3}.0, Reshape{3}.0, TensorType(float32, (False, Tr
ue, True)), TensorType(float32, 3D), 'local_reshape_dimshuffle')
```

Gets stuck in process of initializing theano objects

But we managed to train with only CPU.



Here we tried to copy and trained it for more than 1 hour on laptop, accuracy was around 95%.

Running model on arrays of different size showed that there is linear correlation between size of array and time to sort. Here is one of printouts:
1,0.00445103645325

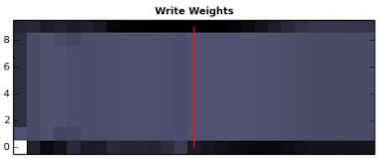
Count | time in seconds to sort
5, 0.0127348899841
25 , 0.0593011379242
125, 0.234660148621
625, 0.983366012573
3125, 4.65422677994
15625, 21.8376741409

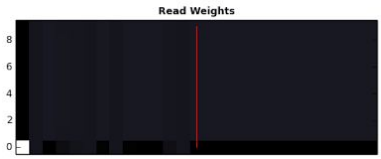
It gives a clue that NTM learned some kind of pocket-sort, when it stores how many of which elements were in array (4 elements with value ‘0’, 3 with value ‘1’...) and then build output like this [0;0;0;0;1;1;1;.....].
There are few reasons to think like that:

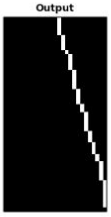
1) Pocket sort IS linear.
2) Pocket sort requires additional memory
3) Pocket sort works well on integer arrays with relatively small range of values (say 1..100 for array of 1000 elements and not -2456454..232424 for array of 100 elements). And we have exactly same situation - size was set around 15-30.

P.S. While doing this task we at first have implemented intuitive way with one ‘1’ in column, and runned it. However, as it turned out, data showed in visualisation is transposed, so we have developed our generator vice versa. It actually worked, but length - value which variates from 1 to say, 30 was our value range and not number of elements. And when we tried to apply model to arrays of different sizes we were punished by theano. So we had to reimplement everything. Here are few samples of that:

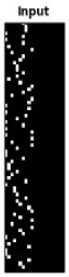

Input

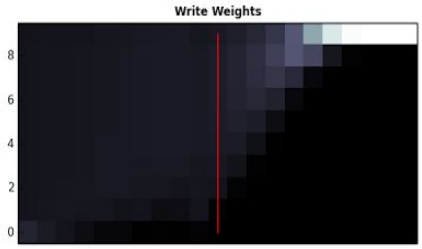

Write Weights
8
6
4
2
0

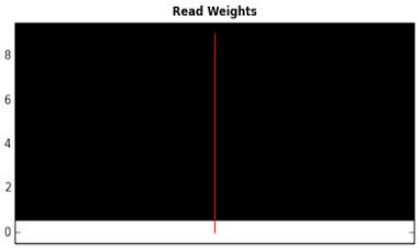

Read Weights
8
6
4
2
0


Output


Prediction


Input


Write Weights
8
6
4
2
0


Read Weights
8
6
4
2
0


Output


Prediction

Conclusion

This project was very interesting from point of learning cutting-edge machine learning technique and was challenging from understanding, running and training point of view.

We are not sure about future of DNCs, because they are really hard to train and sometimes it’s easier to find algorithm by means of logic, but as a part of big machine learning toolset it may find it’s place in data scientist’s lifes.