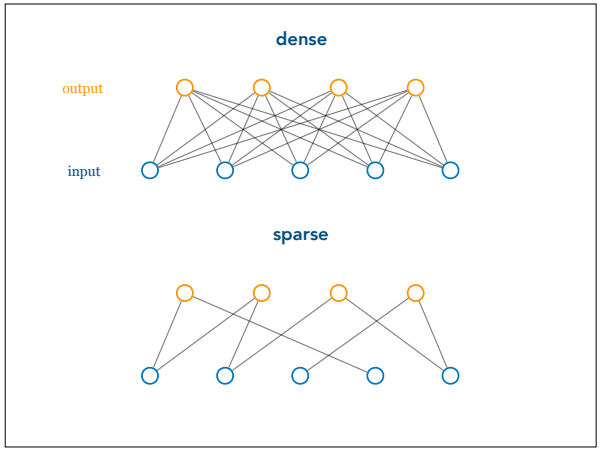
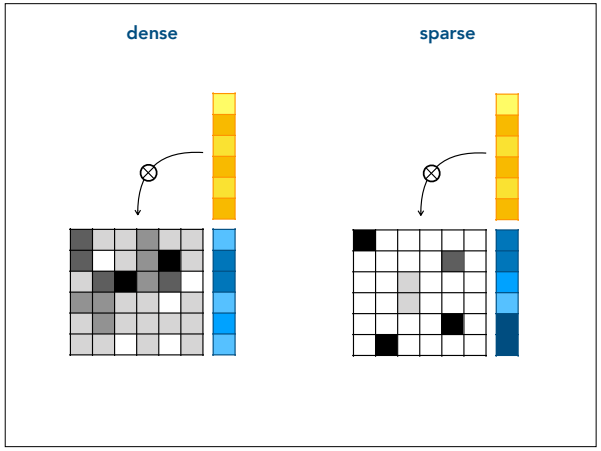
Learning sparse transformations through backpropagation

Peter Bloem
github.com/pbloem · twitter.com/pbloemesquire · peterbloem.nl · p@peterbloem.nl

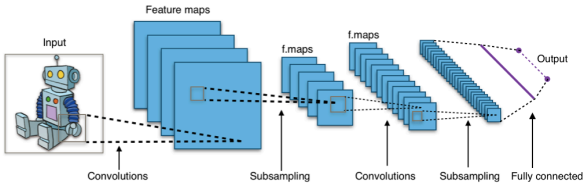


In neural networks, sparsity is often a good property to have. It can make learning easier and faster, and the results are often more interpretable. This image shows what we mean by sparsity: instead of connecting every input with every output, we connect the only *sparsely*.



If we think of a neural network as a matrix transformation, sparsity looks like this: most of the elements of the sparse matrix are zero. Such sparse matrices can be stored efficiently, by storing the nonzero elements by enumerating their indices, and assuming that any element not mentioned has value zero.

convolutions

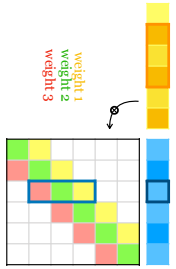


4

One example of a power sparse transformation is the famous convolution layer. Instead of connecting every input with every output, we assume that our input has a grid structure and connect only one patch of the input to a particular hidden node.

---

convolutions



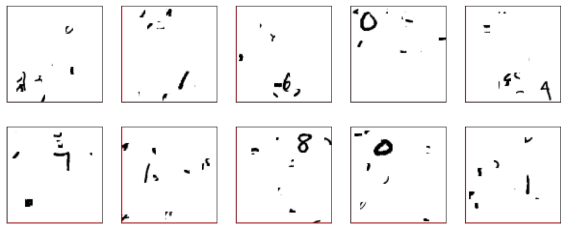
5

Here's what a 1D convolution looks like if we raw it as a sparse matrix.

This is fine when the transformation is the same for every instance in our data. But sometimes, the sparsity of our transformation changes between instances.

---

attention

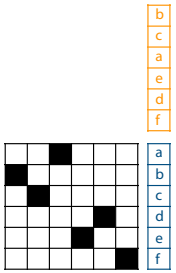


mnist-cluttered

6

On example is *visual attention*. Here, we have a dataset (called mnist-cluttered) of 100 by 100 pixel images, with an MNIST digit placed at a random location, with added noise. A basic ConvNet scores little higher than chance. To do better, we need to first learn to focus on the part of the image that's relevant, and then classify that.

sorting



- Pointer networks, Sinkhorn-Gumbel networks
- All parametrize the *sparse* permutation matrix with a *dense* parameter matrix.

7

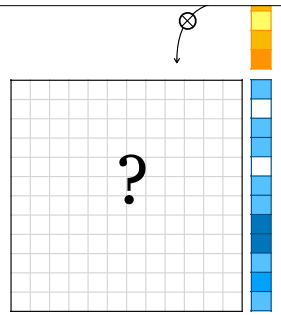
A final example is sorting. Sorting an input vector is a highly complex, nonlinear transformation. But if we can condition the transformation on the input, it becomes a simple, high sparse, permutation matrix.

---

problem statement:

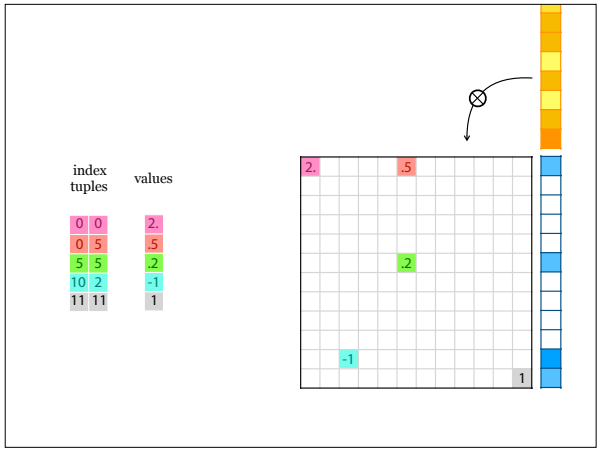
Can we learn sparse transformations using a sparse parametrization?

Ideally, without using REINFORCE

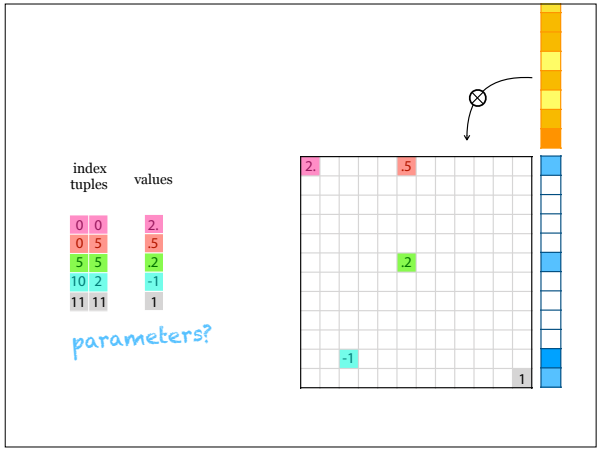


$$y = W \cdot x$$

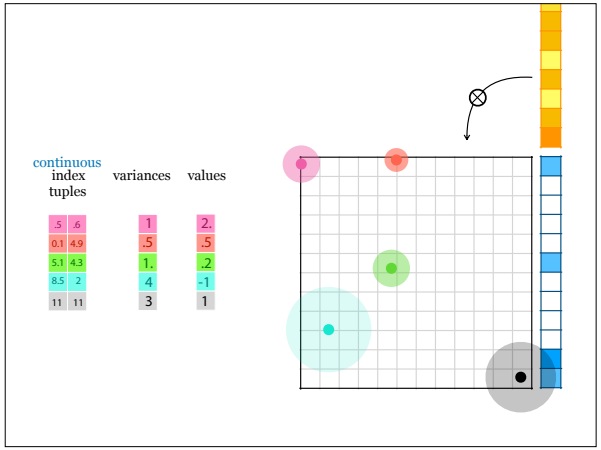
Or visually: can we learn, through backpropagation, the parameters of the sparse matrix W, **without embedding it in a dense matrix during training**. We know that most elements of this matrix will be zero, but we don't know beforehand which ones.



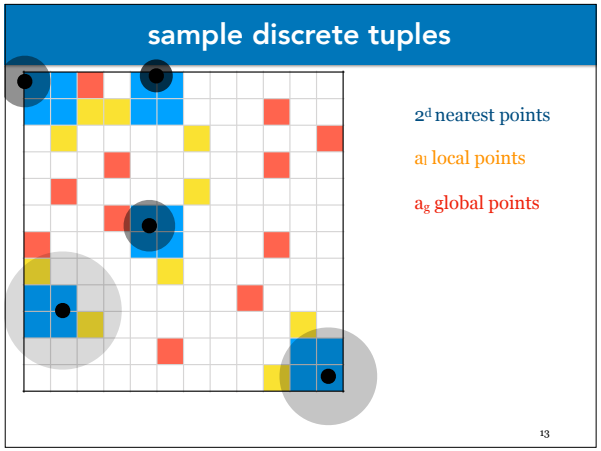
This is how a sparse matrix is normally represented.



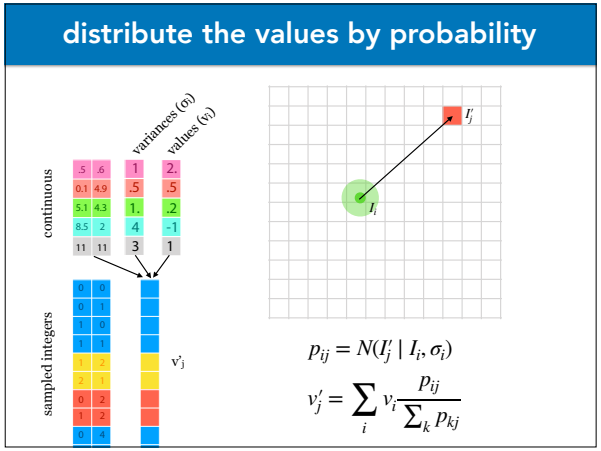
We could imagine just using this parametrization and trying to learn these three matrices through backpropagation. Unfortunately, the index tuples are integers, we can't make a small change in the index tuples and observe a small change in the output or in the loss. The smallest change we can make is to add or subtract 1 to one to or from one of the numbers, and it we do that the output will likely change a lot. In other words, we won't get a meaningful *gradient* over the index tuples.



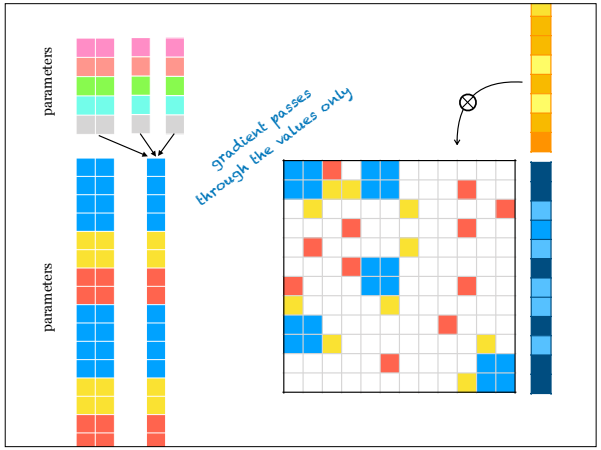
to achieve this, we parametrize the index tuples with Gaussian distributions over the continuous space containing the index tuples. The idea is that as we learn, the index tuples with converge to integer values and the variance will get closer and closer to zero.



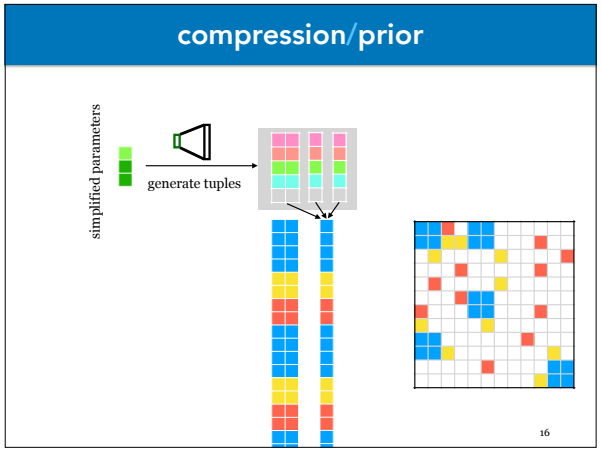
To run this continuously parametrized sparse matrix into something that (a) we can actually use to perform sparse matrix multiplication and (b) will give us a gradient, we sample integer index tuples and distribute the existing values according to probability under the continuous parameters.



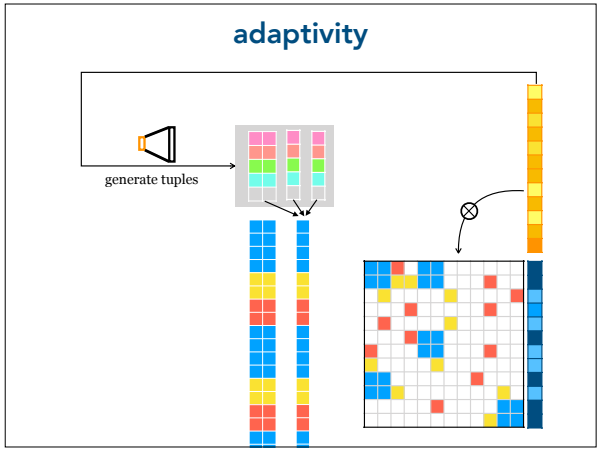
Each continuous tuple donates some of it value to each integer tuple, proportional to the probability of the integer tuple under the distribution described by the continuous tuple.



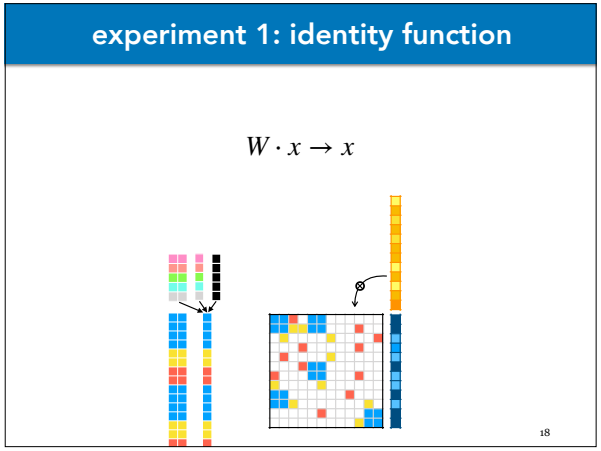
This gives us a sparse matrix with integer tuples, which we can use to perform a sparse-times-dense matrix multiplication. The gradient passes only through the values of this matrix, but since it's constructed from all three parameters (tuples, variances and values) we still get a gradient over our entire model.



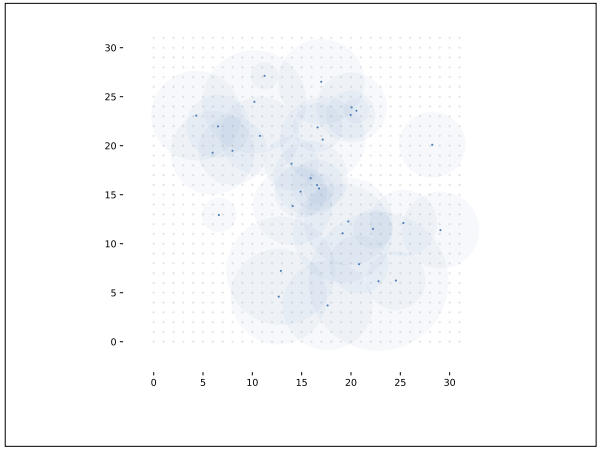
We can make the parameters free, but they can also be derived from a source network, and pass their gradient upstream during backpropagation. For instance, if we know we want to learn a convolution, layer, but we don't know with which dilation, we can make the dilation learnable parameter and make design a source network that constructs the index tuples of a convolution with the given dilation.



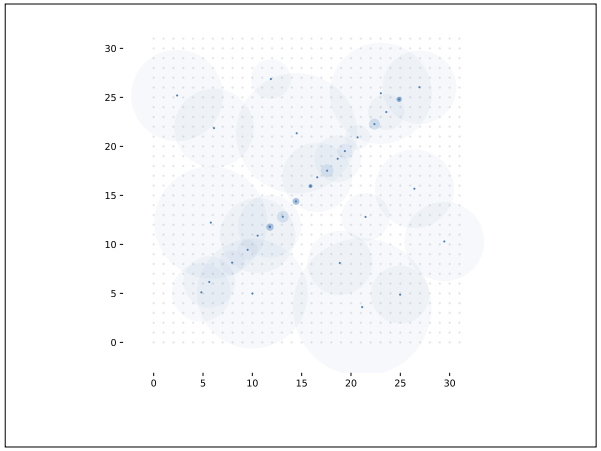
We can also make the sparse layer *adaptive*: let it learn its parameters as a function of the input.



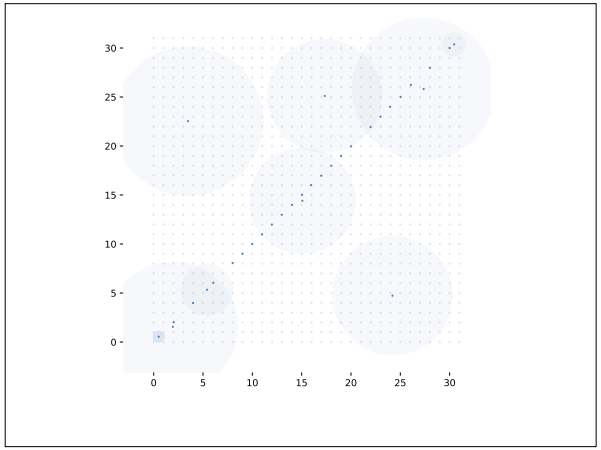
For our first experiment, we simply learn the identity function. We do not use a source network, and fix all values to 1, learning only the index tuples and the variances.



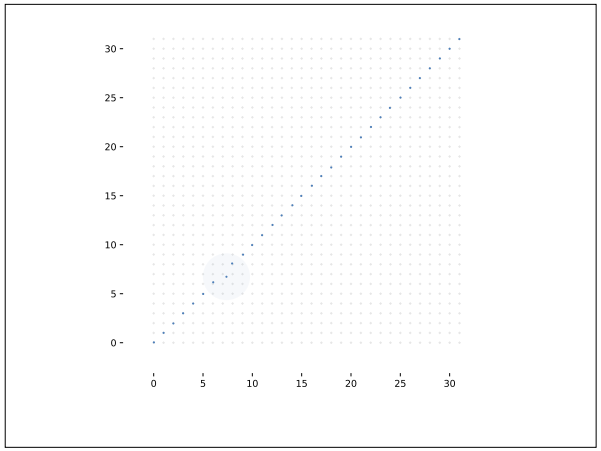
After 0 iterations.



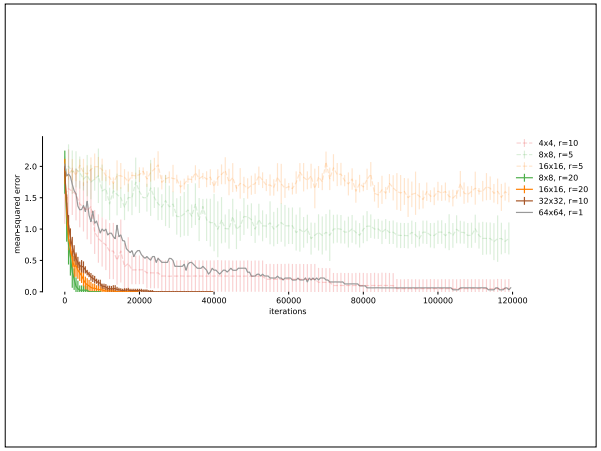
After 1000 iterations.



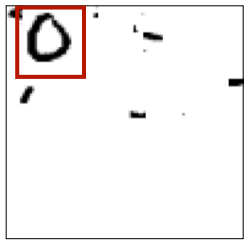
After 5000 iterations.



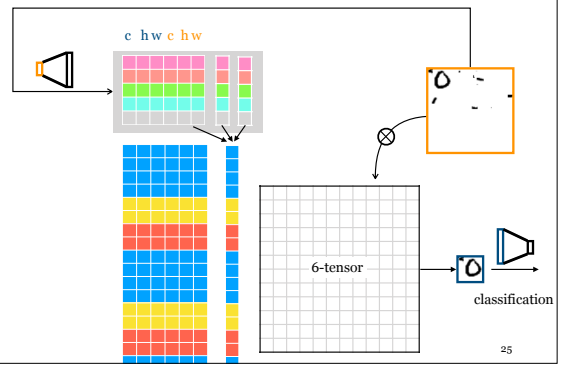
After 13000 iterations.



attention



attention

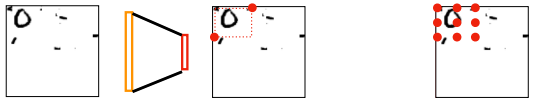


25

To tackle the mnist-cluttered task, we need a higher dimensional tensor. Three dimensions (channel, height, width) come in and three dimensions (channel, height, width) go out. Since we know we want to extract a rectangle, we can fix the output indices, and learn only the height and width in the input.

---

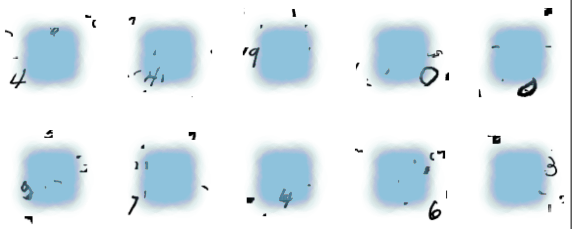
attention



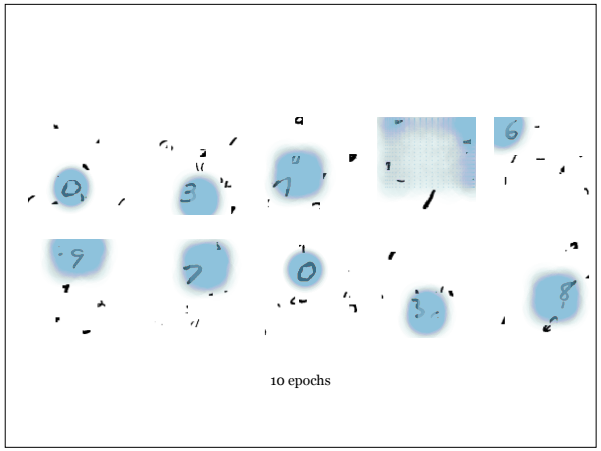
- Only the h and w dimensions of the input are learned, the rest are fixed.
- The input is 100 × 100, the output is k × k.
- Each input channel has one connection to each output channel.
- Values are shared between tuples.

26

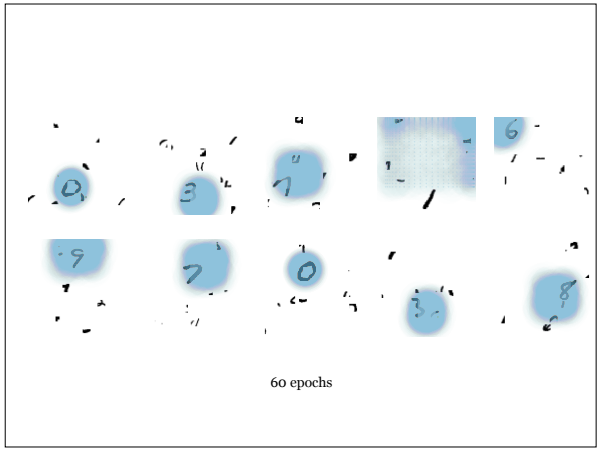
To constrain the network even further. The source network generate only four values, which together define a bounding box. Within this bounding box we generate k × k evenly spaced points, each point in this rectangle is connected to a node in the k × k output layer.



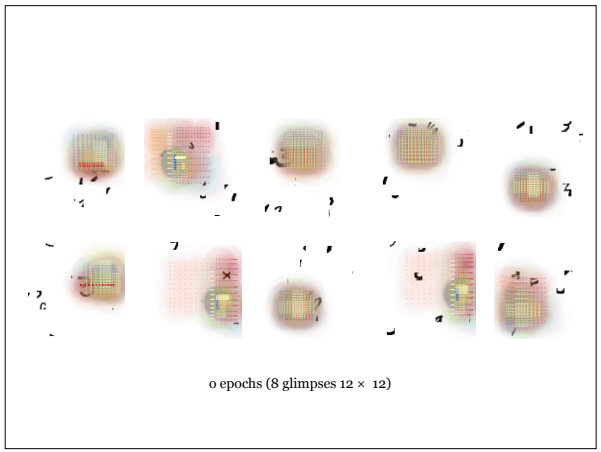
0 epochs (single 32 × 32 glimpse)



10 epochs



60 epochs



0 epochs (8 glimpses 12 × 12)

versus recurrent attention

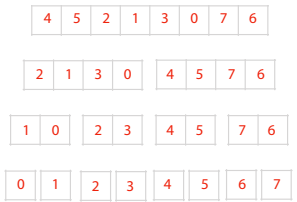
	error
RAM (best) Mnih et al, 2014	8.11 %
ours: 1 glimpse 32 × 32	0.85 %
ours: 8 glimpses 12 × 12	1.01 %

sorting

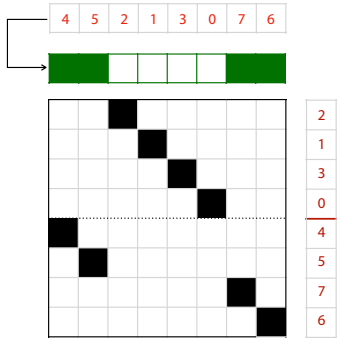
- Don't learn to sort. Sort by a key, and implement a sorting algorithm so that you get a gradient over the keys.

quick sort refresher

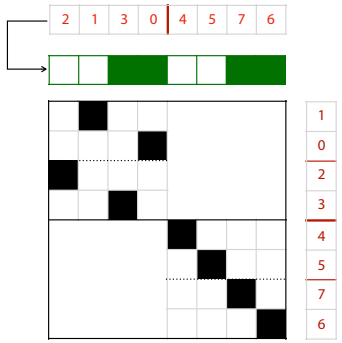
- split input into buckets, by pivot, recursively



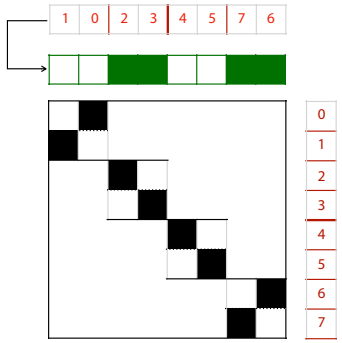
differentiable quick sort



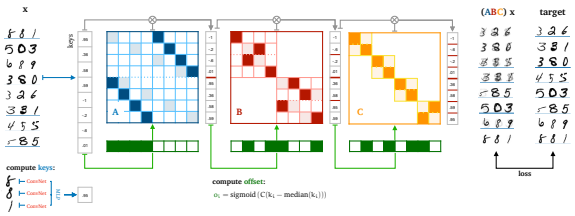
differentiable quick sort



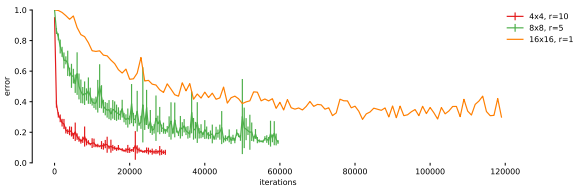
differentiable quick sort



experiment: 3-digit MNIST sorting



Note that the sorting network has no parameters of its own. It just allows a gradient to backpropagate to the keys.



NB the legend should not say n x n, just n.

conclusions

We can learn sparse transformations by sampling random paths from a larger, unseen, computation graph.

Sparse matrices are just one approach

Applications so far: attention, permutation

Future applications (?): sparse embeddings, implicit graph structure, interpretable AI

- Arxiv preprint: https://arxiv.org/abs/1810.09184
- code: https://github.com/MaestroGraph/sparse-hyper

  Not for the faint of heart, currently. If you're interested, feel free to ask for help.