Internship Report: Computer Vision in OCaml & Computation Graph Optimisation

Pierre Vandenhove
pierre.vdhove@gmail.com

11th December 2018

Supervisors: Dr Anil Madhavapeddy and Dr Liang Wang

OCaml Labs, Computer Laboratory, University of Cambridge

Contents

List of Figures

1 Introduction

This report describes what was produced during my internship at OCaml Labs from 9 August to 12 November 2018.

It consists of

- an implementation of the Mask R-CNN network using OCaml's numerical library Owl (see Section 2);
- a new memory allocation algorithm for Owl's *computation graph* module (see Section 3);
- some minor contributions to Owl (see Section 4);
- a technical report about Owl's *computation graph* module[1];
- a blog post about the internship[2].

I would like to deeply thank Christopher Troestler for the idea to apply for an internship at OCaml Labs, Anil Madhavapeddy and Gemma Gordon for the amazing internship opportunity, Liang Wang for building the great library Owl and for his continuous support throughout the internship, and Jianxin Zhao for his help and the frequent conversations about Owl.

[1] The technical report is available at `https://arxiv.org/abs/1812.03770`.
[2] The blog post is available at `http://ocaml.xyz/project/pierre_cgraph.html`.

2 Computer Vision with Owl

2.1 Motivation

My initial task was to find a way to automatically segment and categorise pictures and videos, using OCaml's numerical library Owl [Wan17], created by Liang Wang. Many machine learning tasks had already been successfully ported to Owl (for character recognition[3], image style transfer[4],...). There was even an implementation of Google's *Inception* neural network [SLJ+15], which can already label images very accurately.

Computer vision is a field dealing with many different automated tasks whose goal is to give high-level descriptions of images or videos. It has been applied to a wide variety of domains ranging from highly technical (automatic tagging of satellite images, analysis of medical images,...) to more mundane (categorise pictures in your phone, make your face into an emoji,...). It has seen tremendous progress since 2012, when A. Krizhevsky et al. [KSH12] used the first deep learning approach to computer vision, crushing all their opponents in the ImageNet challenge[5]. It has therefore evolved quite a lot since Inception was first described in 2014 and it was relevant to implement a more recent and involved network with Owl.

Inception performs *single-label image classification* — it works well when there is one large object in an image, but gets easily confused when there are lots of small ones. Other programs are meant to classify the pixels on an image in different categories (*semantic segmentation* such as in [LSD15]), or to detect the position of the objects on an image (*object detection*). In 2017, the *Mask R-CNN* (*Mask Region-based Convolutional Neural Network*) architecture [HGDG17] was published and with sufficient training, it can solve all these problems at once: it can detect objects on an image, label each of them and provide a binary mask to tell which pixels belong to the objects. This network has now been implemented in Owl. An example of output can be seen in Figure 1. A video processed with the network is also available[6].

2.2 The Mask R-CNN network

I will briefly outline the main parts of the architecture of Mask R-CNN and how it stands out from its predecessors. The Owl implementation of the inference mode is available at `https://github.com/pvdhove/owl-mask-rcnn`. The code was mostly ported from an existing Keras/TensorFlow implementation [Abd17]. At the beginning of each paragraph, I refer to the file in which each part is implemented in the `src/model/` directory of the GitHub repository. The file

[3] `https://github.com/owlbarn/owl/blob/master/examples/lazy_mnist.ml`
[4] `http://demo.ocaml.xyz/neuraltrans.html`
[5] `http://image-net.org/challenges/LSVRC/2012/results.html`
[6] `https://www.youtube.com/watch?v=ruM7S-cqk-k`

Figure 1: Image processed by the Owl implementation of Mask R-CNN.

linking all the parts together is `model.ml`. See [HGDG17] for more general details about Mask R-CNN.

2.2.1 Feature extractor

`resnet.ml` The picture is first fed to a convolutional neural network in order to extract features on the image. The first few layers detect low-level features (edges and basic shapes) but as you go deeper into the network, these features are assembled to detect higher level features (people, cars). Five of these layers (called *feature maps*) of various sizes, both high- and low-level, are then passed on to the next parts. This implementation chooses Microsoft's ResNet101 [HZRS16] as a feature extractor.

`model.ml` These feature maps are then transformed with a Feature Pyramid Network [LDG+17] to share information between all the maps, such that each map knows about both high- and low-level features at different resolutions. Later on, the size of the objects will determine which feature map is used to analyse them.

2.2.2 Proposal generation

`regionProposalNetwork.ml`, `proposalLayer.ml` To try to locate the objects, about 250K overlapping rectangular regions (*anchors*) are generated. A small convolutional network (a Region Proposal Network, introduced in 2015 by the predecessor of Mask R-CNN [RHGS15]) scans the feature maps and quickly associates to each of them a number that could be called the 'objectness' of that region. The 1000 best anchors are then selected according to their objectness (higher is better). Anchors that overlap too much with each other are eliminated, to avoid detecting the same object multiple times. Each selected anchor is also refined in case it was not perfectly centered around the object.

2.2.3 Classification

`featurePyramidNetwork.ml`, `detectionLayer.ml` All anchor proposals from the previous layer are resized to a fixed size and fed into a 10-layer neural network that assigns to each of them probabilities that it belongs to each class (the network is pre-trained on fixed classes; changing the set of classes requires to re-train the whole network). Note that this step does not take as much time for each anchor as a full-fledged image classifier (such as Inception) since it reuses the precomputed feature maps from the Feature Pyramid Network — there is no need to go back to the original picture. The class with the highest probability is chosen for each proposal and thanks to the class predictions, the anchor proposals are even more refined. Proposals classified in the 'background'

class are deleted. Eventually, only the proposals with an objectness over some threshold are kept, and these are the final detections.

`featurePyramidNetwork.ml` The only thing left to do is to generate a *binary mask* on each object. This is handled by a small convolutional neural network which outputs for each detected bounding box a small square of values between 0 and 1. This square is resized to the original size of the bounding box with bilinear interpolation, and pixels with a value over 0.5 are tagged as being part of the object.

2.3 State of the implementation

A demonstration of the network is available online[7]. If you want to apply it on large pictures, videos or experiment a bit more, see the code[8]. Pre-trained weights on 80 classes of common objects are provided, which have been converted from the aforementioned TensorFlow implementation [Abd17].

A few things can still be improved. First of all, to fully support training, some operations are still missing both in Owl and in my implementation of Mask R-CNN. Then to make it even faster, especially for videos, GPU support would be incredibly helpful. Owl's GPU support is already fully functional, but some work is still necessary to apply it to Mask R-CNN.

[7] `http://demo.ocaml.xyz/mrcnn.html`
[8] `https://github.com/pvdhove/owl-mask-rcnn`

3 Computation graph optimisation

3.1 Motivation

The first noticeable issue after porting Mask R-CNN to Owl is that the memory usage, in inference mode, is huge. The network has over 400 layers and to avoid reinitialising the network for every picture, it is good to keep its input size fixed and to resize instead all the images to that size — a larger size takes more time and memory but yields more accurate results. A reasonable input size for this network is a 1024-pixel-wide square. Unfortunately, obtaining detections for one picture with this size required over 11 GB of RAM. As a comparison, the TensorFlow implementation only uses 1 GB.

What was not used yet is the *computation graph* module of Owl. The following definition is inspired by [LINK18].

Definition 3.1 (Computation graph)**.** Let O be a set of operations such that $\texttt{Var} \in O$.

A *computation graph* (or *dataflow graph*) over O is a graph $G = (V, E, \lambda, U)$, where V is the set of vertices of G, $E \subseteq V \times V$ is the set of directed edges, $\lambda : V \to O$ is a function mapping each vertex to an operation $o \in O$, $U \subseteq V \times V$ is the set of *update* edges. We require that the graph (V, E) is acyclic and that for all $(u, v) \in U$, $\lambda(v) = \texttt{Var}$.

Each vertex of the computation graph represents an operation. An edge $(u, v) \in E$ means that the output of vertex u will be used as an input by the operation of vertex v — it defines the dependencies between the operations. With a distributed system, different vertices of the graph might be computed on different machines. The graph is then also a description of the network linking the vertices together. You can see an example of a program and its computation graph in Figure 2. In this example, to evaluate the output vertex x_5, we need to specify values for the variables x_1 and x_3. If after evaluating it once, we only modify the value of x_3, there is no need to re-evaluate x_2.

The set of *update edges* U is a mechanism to allow reusing the value of some vertices at the end of an evaluation as variables for the next evaluation. This is necessary to express recurrent neural networks or neural network training (for which at the end of each iteration, weights are updated). Note that the graph $(V, E \cup U)$ can contain cycles. Notice also that the graph is not necessarily *simple* in the graph-theoretical meaning: two vertices can be linked by more than one edge (for instance with the computation $x \leftarrow y * y$).

Representing the structure of a program as a computation graph has several advantages, especially for computationally-intensive code dealing with large multi-dimensional arrays. A really useful one is that prior to evaluating the nodes, you can optimise the structure of the graph: for instance, useless calculations such as adding an array with nothing but zeros can be removed, common patterns can be merged into one node and executed more efficiently, etc. Thanks

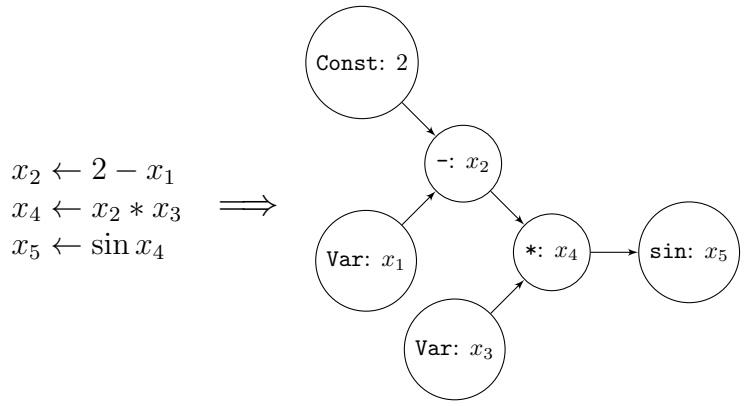

Figure 2: A simple program and its computation graph.

to these optimisations, the number of nodes of Mask R-CNN drops from 4095 to 3765. Another really important feature in this case is the ability to preallocate a memory space to each node, to decrease the overall memory consumption and reduce the garbage collector overhead. This is especially relevant to reduce the memory consumption of Mask R-CNN. Changing the memory allocation algorithm is my main contribution to Owl.

The following standard definition will be used later in the report.

Definition 3.2 (Topological ordering)**.** Let $G = (V, E, \lambda, U)$ be a computation graph with n vertices. A *topological ordering* of G is a bijection $\gamma : V \to \{0, 1, \ldots, n-1\}$ such that for all $(u, v) \in E$, $\gamma(u) < \gamma(v)$.

3.2 Memory allocation problem

Allocating memory to each vertex of a computation graph is a problem similar to one that was first described in 1973 [Set73] to look for an efficient algorithm for register allocation, using an abstraction called the *pebble game*. We recall its definition.

Definition 3.3 (Pebble game)**.** The *pebble game* is played on a directed acyclic graph (DAG). Each vertex can store at most one pebble. The game begins with no pebble on any vertex. At each step, the player can perform one of the following moves:

1. if a vertex v has no predecessor (*input vertex*), the player can place a pebble on v.
2. if all predecessors of a vertex v are pebbled, the player can place a pebble on v or *slide* a pebble from one of its predecessors to v.

3. the player can remove any pebble from a vertex (and reuse that pebble later).

The goal of the game is to place a pebble at least once on some fixed *output vertices* of the graph. A *pebbling strategy* is a sequence of moves following the rules of the game and reaching the goal. The *space* used by a pebbling strategy is the maximum number of pebbles used simultaneously during the execution of the strategy. The *time* of a strategy is the number of times a pebble is placed on a vertex (without counting the removals of pebbles).

This relates to the memory allocation of the computation graph if we see pebbles as memory blocks used to store the output value of a vertex. We assume that the values of the *input vertices* are known and can be loaded into memory anytime (move 1). We can only compute the value of a vertex if all its predecessors are simultaneously stored in memory (move 2). The *sliding* move means that the memory of a vertex can be overwritten by its successor during its computation (*inplace reuse*). We can always reuse a memory block from another vertex (move 3).

By pebbling a graph in topological order and removing pebbles when they are not needed anymore, it is always possible to pebble each vertex exactly once (time is minimal). However, such a strategy may not always yield a minimal space value. If we consider the example from Figure 2, we notice that we can pebble it with a space of 2 and a time of 6 (see Figure 3).

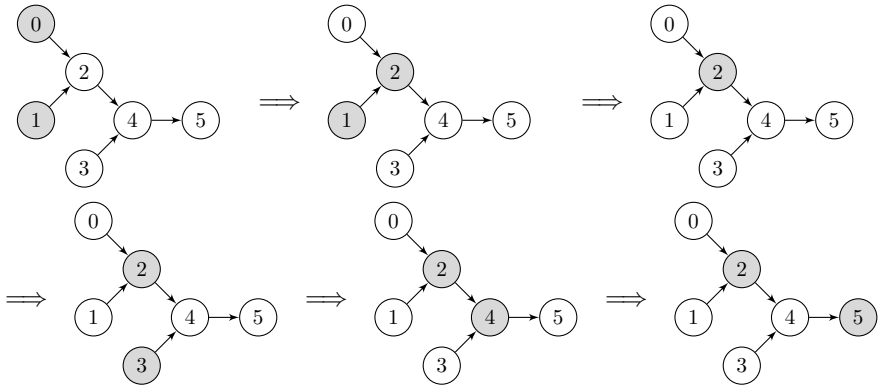

Figure 3: Optimal pebbling of a computation graph, using moves $1 \to 2 \to 3 \to 1 \to 2 \to 2$.

The values of space and time of this strategy are both minimal. We now consider another example in Figure 4. It is possible to pebble this graph with a time of 6 and a space of 3 (by leaving a pebble on vertex 1 until the end). It is also possible to pebble it with a time of 8 and a space of 2 (by repebbling 0 and 1 once 4 is pebbled). However, one single strategy can not be minimal for both time and space.

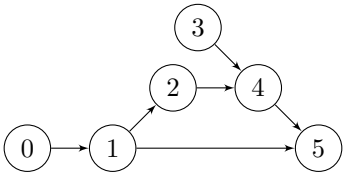

Figure 4: One pebbling strategy cannot reach the minimal values of both space and time for this computation graph.

We see that there is a time-space tradeoff and that depending on the space constraints, it might be worth computing the same vertex more than once. If a graph has certain properties, some algorithms can obtain interesting bounds on space complexity without sacrificing much of the time complexity. For instance, [CXZG16] uses the specific shape of the computation graph when training a neural network to devise a memory allocation strategy with a square root bound on space complexity by means of two computations of the forward pass. You can find more examples in [Sav97, Chapter 10]: there even exists a family of graph G_k such that using k pebbles takes exponential time at best, but using $k+1$ pebbles takes minimal time. This illustrates that excessive memory minimisation is not always desired in practice.

The problem of whether there exists a pebbling strategy using less than k pebbles has been proven to be PSPACE-complete [GLT79]. If we do not allow any repebbling of vertex, the same problem is NP-complete [Set73], and even hard to approximate within any constant factor [APW12]. Since computation graphs can have a few thousand vertices, we will not consider exact algorithms to allocate memory.

For more information on pebbling games, see [Sav97, Nor13].

3.3 Allocation algorithm

The initially implemented strategy to allocate memory to a node u in Owl's computation graph module was simply to reuse the memory of a direct predecessor with same output shape as u when that is possible. This optimisation allows to decrease the memory consumption of Mask R-CNN from 11 GB to 7 GB, which is much better but still quite far from the 1 GB of the TensorFlow implementation.

We can actually make it much more performant by sharing memory between nodes

- that are not necessarily a parent/child pair;
- that do not have the same output size (by allocating a large block of memory once, without necessarily using all of it all the time).

Compared to the pebble game (Definition 3.3), we need to be careful of a few more caveats, listed below.

- Some operations, dependently on how they are implemented, can not overwrite some of their parents while they are being carried out. In the pebble game, this corresponds to forbidding the sliding of one pebble for some parents of some operators. To take this into account, we partition the parents of each operator in two sets, the ones that can be overwritten during the computation of the operation and the ones that cannot.
- We need to take into account that some vertices have different output sizes. In the pebble game, this means that we want to assign to each pebble a size that must be larger than the output size of the vertices the pebble is placed on. What we actually want to minimise is the sum of the sizes of the pebbles rather than the number of pebbles (this is still equivalent to the original problem in the specific case where all the output sizes are equal).
- We want to always keep the value of some vertices in memory for practical purposes (for example, vertices v such that $\lambda(v) = \texttt{Const}$, output vertices, vertices u such that $(u, v) \in U$, neural network weights). These vertices cannot share their block of memory.

We explain in more detail the new memory allocation performed in Owl when running the code on a CPU device. For practical reasons, since the algorithm should be efficient for arbitrary graphs, we assume that each vertex can only be computed once (*i.e.* keeping a minimal time value). The chosen algorithm is inspired by the one used by MXNet [CLL+15].

A first challenge is to find an efficient way to share memory between tensors which do not have the same output size. Since Owl's multi-dimensional array module is based on OCaml's `Bigarray`, we can use the `reshape` and `sub_left` functions in the following way:

```
module N = Owl.Dense.Ndarray.S
let block = N.empty [|block_size|] in
let memory = N.reshape (N.sub_left block 0 node_numel) shape
```

The `block` variable is a one-dimensional array of sufficient size. We reuse the `node_numel` first elements of `block` and reshape that one-dimensional array to the right `shape`. Multiple nodes can thus base their memory on the same block, as long as their size is smaller than the size of the block, with no loss of performance.

The topological ordering γ we use for allocation and evaluation is given by traversing the graph from the outputs using a post-order DFS. By going through γ in order, we can notice when the memory of a vertex becomes useless by keeping a counter of the number of times its value has been used by one of its

successors. When all of its successors have been evaluated, the memory block of the vertex can be tagged as reusable.

We follow the rules below to allocate a block of memory to a vertex v:

- we only allocate a new block when no block is available;
- when multiple blocks are available, we pick the smallest block that is big enough to contain the value of v, so that as little memory as possible stays unused. If the available blocks are all smaller that v, we increase the size of the current biggest block to fit v;
- when possible, we always favour an *inplace reuse* (*i.e.* reusing the memory of a direct predecessor of v) to reduce memory access overhead.

You can find the pseudo-code in Algorithm 1. It is important to realise that the block allocation depends on γ. It is primordial to use the same order when initialising and evaluating a graph. It is likely that some other topological order gives a slightly better allocation for some graphs, but no easy way to generate such an order appears to be constantly better. The post-order DFS approach has been chosen for its ease of implementation.

The overall time complexity of the allocation is $\mathcal{O}(n * \log(b))$, where n is the number of nodes in the graph and b is the number of distinct memory blocks at the end of the algorithm (of course, $b \leq n$). The $\log(b)$ factor comes from the FindBestBlock function. Implementing this effectively reduced the memory consumption of Mask R-CNN from 7 GB to 1 GB for a 1024x1024 picture. A summary of the changes can be found at `https://github.com/owlbarn/owl/pull/318`.

For instance, when evaluated in the right order, the computation graph in Figure 5, which can be used to recognise hand-written digits, needs only two different blocks of memory for 18 shareable nodes. You can find more statistics illustrating what the computation graph with this new algorithm achieves in Figure 6.

Using a computation graph has many other advantages that were not mentioned in this report. To learn more about it, see `http://ocaml.xyz/chapter/cgraph_intro.html`. It is important to emphasise that this mechanism can be used for any scientific computation using multi-dimensional arrays, not only neural networks.

Algorithm 1 Memory Allocation for CPU devices

Variables
$refs$, associating to each node its number of successors
reusable, used to store the available blocks
block_size, associating to each block a size
block, associating to each node a block
end Variables

function FindBestBlock(s)
if there is a block in *reusable* with size $\geq s$ **then**
return smallest block in *reusable* of size $\geq s$
else if *reusable* is not empty **then**
$b \leftarrow$ largest block in *block*
$block\_size(b) \leftarrow s$
return b
else
$b \leftarrow$ new block
$block\_size[b] \leftarrow s$
return b

function Initialise(x)
Input: a node x of the computation graph
Effect: allocates a block of memory to x and its ancestors.
if x is not initialised **then**
for all predecessor p of x **do** Initialise(p)
for all predecessor p of x that can be overwritten **do**
$ref[p] \leftarrow ref[p] - 1$
if $ref[p] = 0$ **then** $reusable.add(p)$
$block[x] \leftarrow$ FindBestBlock($size(x)$)
for all predecessor p of x that cannot be overwritten **do**
$ref[p] \leftarrow ref[p] - 1$
if $ref[p] = 0$ **then** $reusable.add(p)$

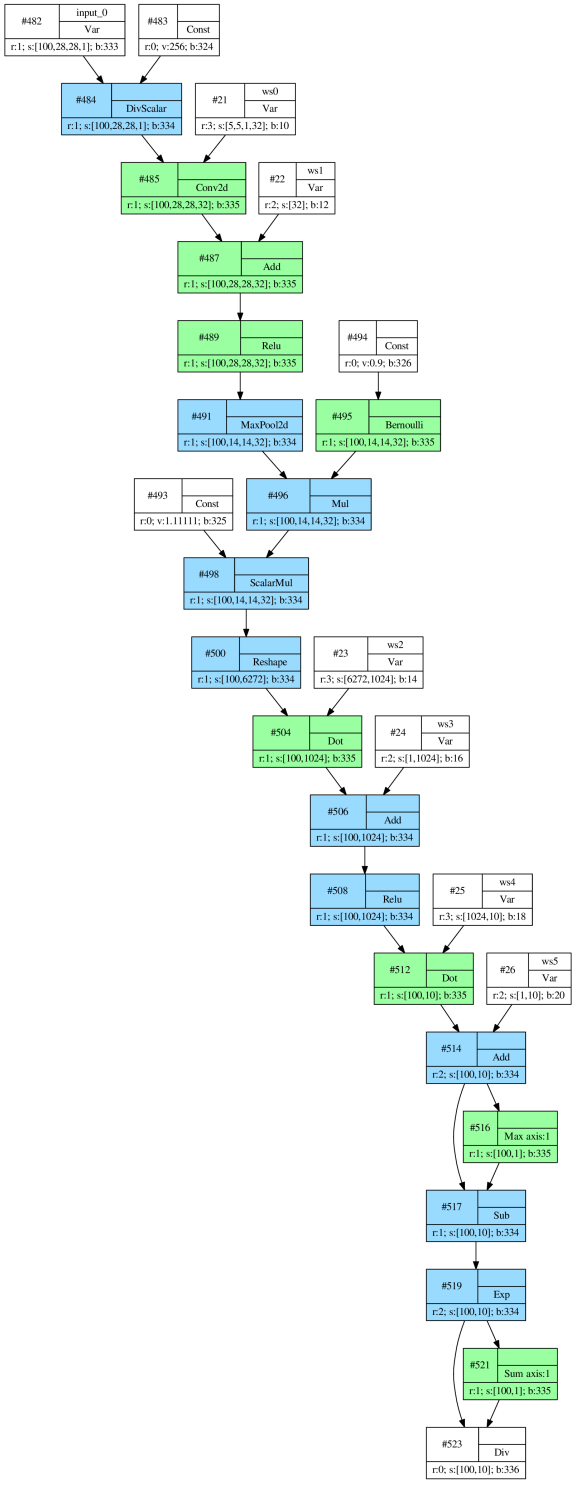

Figure 5: Computation graph of the inference mode of a neural network for character recognition (available at `https://github.com/owlbarn/owl/blob/master/examples/lazy_mnist.ml`). Each colour corresponds to a memory block, white nodes always need to be kept in memory.

Architecture	Time w/o CG (s)	Time w/ CG (s)		Memory w/o CG (MB)	Memory w/ CG (MB)
		Building	Evaluating		
InceptionV3	0.5649	0.1066	0.2275	625.76	230.10
ResNet50	0.7933	0.1395	0.6090	1309.9	397.07
MNIST (training)	20.422	0.1436	10.920	3685.3	895.32
Mask R-CNN	11.538	0.3630	8.379	6483.4	870.48

Figure 6: InceptionV3 and ResNet50 are tested with a 299x299 image; Mask R-CNN is tested with a 768x768 image. The MNIST line refers to the training mode of the network in Figure 5. The time is the average over 30 evaluations, without reusing precomputed nodes when a computation graph is used. The graph building phase includes graph construction, optimisation and memory initialisation. The memory is the maximum resident set size of the program. This was evaluated on a laptop with an Intel i5-6300HQ CPU and 8 GB of RAM.

4 Minor contributions to Owl

In order to make Mask R-CNN work with a clean code and as efficiently as possible, I had to perform the following changes in Owl.

Implementation of ResNet[9]. ResNet is implemented in Owl as a part of Mask R-CNN, but has also been made available as a standalone repository.

Complexity of `top` and `bottom` functions[10]. The time complexity of the `top` (resp. `bottom`) function in the `Ndarray` module, used to return the `k` greatest (resp. smallest) elements in tensor `t`, has been lowered from $\mathcal{O}(n * k)$ to $\mathcal{O}(n * \log(k))$, where n is the number of elements in `t`, by implementing and using a binary heap.

Multi-input/output support in neural network module[11]. New functions to let a user define neural networks with multiple inputs or outputs. Also, new neural layer `LambdaArray` to let users define arbitrary functions taking multiple tensors as arguments inside their neural networks.

Save and load weights of `Normalisation` layer[12]. A bug caused some parameters of the `Normalisation` layer of the neural network module not to be properly saveable and loadable. This has been fixed.

Graph module improvements[13]. BFS traversal was implemented, as well as an optional parameter to perform a pre-order or post-order DFS. The function to generate a topological sort had a complexity bug which was fixed.

Computation graph additions[14,15,16].

1. Operator `Delay` allowing user-defined functions to be wrapped in a node of the computation graph.
2. Operator `LazyPrint` to print a node of a computation graph, in a similar fashion to TensorFlow's `tf.Print()` function.

[9] `https://github.com/pvdhove/owl-resnet`
[10] `https://github.com/owlbarn/owl/pull/306`
[11] `https://github.com/owlbarn/owl/pull/308`
[12] `https://github.com/owlbarn/owl/pull/317`
[13] `https://github.com/owlbarn/owl/pull/314`
[14] `https://github.com/owlbarn/owl/pull/311`
[15] `https://github.com/owlbarn/owl/pull/325`
[16] `https://github.com/owlbarn/owl/pull/315`

3. Function to automatically compile a neural network into an optimised computation graph for the inference mode, with a way to perform inference by batch.

References

[Abd17] Waleed Abdulla. Mask R-CNN for object detection and instance segmentation on Keras and TensorFlow. `https://github.com/matterport/Mask_RCNN`, 2017.

[APW12] Per Austrin, Toniann Pitassi, and Yu Wu. Inapproximability of treewidth, one-shot pebbling, and related layout problems. In Anupam Gupta, Klaus Jansen, José Rolim, and Rocco Servedio, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 13–24, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[CLL+15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[CXZG16] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016.

[GLT79] John R. Gilbert, Thomas Lengauer, and Robert Endre Tarjan. The pebbling problem is complete in polynomial space. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, STOC '79, pages 237–248, New York, NY, USA, 1979. ACM.

[HGDG17] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. Mask R-CNN. *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2980–2988, 2017.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[LDG+17] Tsung-Yi Lin, Piotr Dollár, Ross B Girshick, Kaiming He, Bharath Hariharan, and Serge J Belongie. Feature pyramid networks for object detection. In *CVPR*, volume 1, page 4, 2017.

[LINK18] Tung D. Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. TFLMS: Large model support in TensorFlow by graph rewriting, 2018.

[LSD15] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[Nor13] Jakob Nordstrom. Pebble games, proof complexity, and time-space trade-offs. *arXiv preprint arXiv:1307.3913*, 2013.

[RHGS15] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 91–99. Curran Associates, Inc., 2015.

[Sav97] John E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997.

[Set73] Ravi Sethi. Complete register allocation problems. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, pages 182–195, New York, NY, USA, 1973. ACM.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[Wan17] Liang Wang. Owl: A general-purpose numerical library in OCaml. *CoRR*, abs/1707.09616, 2017.