Profiling GPU Memory with PyTorch

by

Izaak Niksan

Supervisor: Prof. Gennady Pekhimenko
Advisor: Hongyu Zhu
April 2020

B.A.Sc. Thesis

Division of Engineering Science
UNIVERSITY OF TORONTO

Profiling GPU Memory with PyTorch

by

Izaak Niksan

Supervisor: Prof. Gennady Pekhimenko
Advisor: Hongyu Zhu
April 2020

ABSTRACT

Machine learning is developing at an unprecedented pace, with applications to everything from self driving cars to medical imaging. Neural networks are the backbone of many of these machine learning implementations due to their inherent ability to learn from input data. Creating an effective machine learning model requires substantial training, which is typically done on Graphics Processing Units (GPUs). Training performance can be improved by loading large amounts of input data onto the GPU concurrently, which requires a large memory footprint. By developing tools to understand how a model is using GPU memory, researchers can gain insight into how to optimize their algorithms to work better on their hardware. This paper presents a novel memory profiler for PyTorch, a popular new machine learning framework. This new profiler's design is a departure from existing profilers, which were designed for frameworks with static computational graphs. The profiler is an easy-to-use, flexible, low-overhead, and effective solution which decomposes the memory consumption into feature maps, weights, and gradients. It also presents detailed layer-by-layer metrics for further insight into how a model uses memory. The profiler requires minimal setup and can be integrated with existing PyTorch programs using only a few lines of code. Experiments were conducted on existing models such as Neural Collaborative Filtering and ResNet-50, and show the profiler's ability to correctly measure their memory consumptions.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitute to my thesis supervisor, Prof. Gennady Pekhimenko, for his continued support and mentorship during my research. I deeply appreciate the opportunity he has given me in allowing me to work with the EcoSystem research group at UofT. I would also like to thank Hongyu Zhu for the many hours he has spent guiding me during this project. His expertise was invaluable, and he was always willing to lend a helping hand. I greatly appreciate the time that Bojian Zheng spent explaining to me his previous work on the MXNet memory profiler. And finally, I would like to thank the members of the PyTorch forums who continually volunteer their time to answer questions. When documentation was lacking, reading through countless of posts on the forums helped fill gaps in my knowledge.

Contents

List of Figures

List of Tables

Listings

1. Introduction

Deep neural networks (DNNs) have a history dating back over half a century, with Donald Hebb's 'Hebbian Learning Rule' laying the foundations for modern techniques [39]. In the following years this approach was developed and improved, with notable developments by Rosenblatt (the first perceptron), Werbos (backpropagation), Jordan (Recurrent Neural Network), Hochreiter & Schmidhuber (LSTM), and Hinton (Deep Belief Networks) [39]. While this theoretical progression was occurring, the computational machinery required to actually implement these models was non-existent. In the last decade, the development of high-performance computing hardware - especially GPUs - has enabled many of these once-theoretical DNNs to be implemented in practice. While GPUs were once used for graphics processing applications - improving performance by parallelizing computation across many cores - they have found use in machine learning applications. Still, however, there remain hardware bottlenecks which constrain the training performance of machine learning models, and therefore gaining deeper insight into the underlying physical resource usage is vital for further advancements in the field. GPU memory is one highly-constrained resource of particular importance to many researchers. For reference, the NVIDIA RTX 2080 Ti has 11 GB of device memory; even this modern, high-end device will run out of memory during training if the mini-batch size and the network topology are too large. Notable examples of memory optimization techniques that researchers have developed include Gist [15], vDNN [30], SCNN [26], and EIE [10]. Without profiling tools to gain deeper insight into performance, such advancements would be more difficult.

The EcoSystem research group at the University of Toronto has developed open-source memory profiling tools for several machine learning frameworks, including MXNet [40], TensorFlow [1], and CNTK [32]. These tools were created by directly logging all GPU memory allocations made by the framework. Most such allocations happen at the start of the program when the static computational graph (representing the training algorithm) is being created. The profiler then links these allocations to the part of the computational graph which they correspond to, making it possible to create a detailed memory breakdown of the model. The difficulty with creating a memory profiler for PyTorch lies in the fact that PyTorch *(a)* uses *dynamic* computational graphs, one of which is created every iteration of training, and *(b)* incrementally allocates large chunks of GPU memory to create a *cache*. Throughout training, memory is continually being taken and released from the cache as the forward and backward

passes progress.

To profile memory with PyTorch, the fastest-growing machine learning framework today, new strategies and insight are required. This paper first provides background on the topic, then describes the design and usage of a novel memory profiler for PyTorch, then presents results of it being used on real models.

2. Background in Computational Machine Learning

2.1 Neural Networks

While there are many types of artificial neural networks, they all involve the transformation of some input data (for example the pixels of an image) into meaningful output - in fact, a neural network is nothing more than a composition of mathematical functions. The function modeled by a neural network is special, however, because it can model *any* mathematical function with arbitrary accuracy (provided that the network has at least two layers) [39]. It is because of this property that neural networks have been labeled 'universal approximators'. Loosely, this means that if there are useful patterns in the world - whether they correspond to user preferences in movies based on past movies they have watched, or classification of images based on their pixel composition, or even cardiac arrest likelihood based on heart rate - a sufficiently-trained neural network can identify it. The effectiveness of a neural network in identifying these patterns depends primarily on two things: the amount of data it has been trained with, and its specific topology.

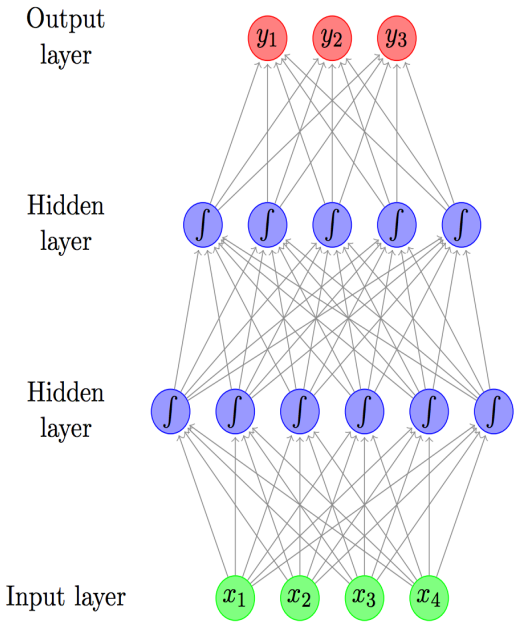


Figure 1: Topology of a fully-connected feedforward neural network [3]

The neural network in Figure 1 shows the basic building blocks from which modern neural networks emerge. It is a simple feedforward neural network - a more complete list of network types can be

found in Appendix B. The green nodes represent the input data to the network, which is what the network must use to make meaningful predictions. Perhaps, the input vector $[x_1, x_2, x_3, x_4]^T$ might represent a four-word sentence.

The gray lines in the diagram represent *weights*, or the scaling coefficients from one node to the next. These weights together create linear combinations of the outputs of one layer into the nodes of the next layer.

The purple nodes represent *neurons*, which take a weighted sum of input nodes and produce a scalar-valued output. Each neuron is a nonlinear function; the nonlinearity is important here because without it the universal approximator guarantees no longer apply. For example, each neuron might be the ReLU function $f(x) = max(x, 0)$ which clips all negative values. The intermediate outputs of each layer are known as *feature maps*. Notice how there are two layers of purple nodes here - this denotes that the model has two hidden layers. There can be an arbitrary number of hidden layers in a network, and while there is no standard for the number of layers required for a network to be considered a 'deep neural network', it should have at least three.

Finally, the red nodes are the *outputs*, which are the network's predictions based on the provided input data. In this example, each output node might be the one-hot-encoded representation of who likely wrote the input sentence. The result closest to 1 here indicates a network's predicted output node. If y_1 is associated with Alice, y_2 with Bob, and y_3 with Charlie, then the output vector $[y_1, y_2, y_3]^T = [0.983, 0.217, 0.015]^T$ indicates that it was likely Alice's sentence.

2.2 GPUs and Their Applications to DNNs

Graphics Processing Units (GPUs) are at the center of recent machine learning breakthroughs. It turns out that the computational workloads of neural networks are, in essence, sequences of matrix operations; as it was shown in the previous section, matrix multiplication is used to conveniently denote the application of scalar weights to the outputs of a neural network layer. GPUs can perform these computations orders of magnitude faster than CPUs due to the parallelism inherent in their design. Figure 2 shows the architecture of a modern GPU, which contains numerous computational cores (green).

To understand the value that GPUs bring in terms of computational parallelism, suppose that a

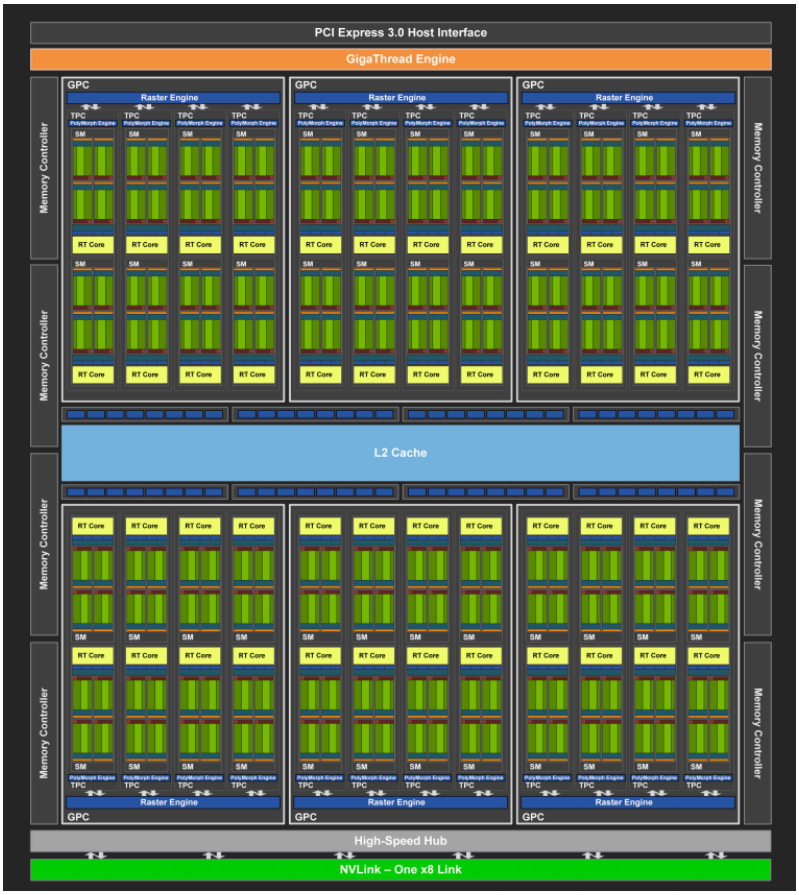


Figure 2: Diagram of the NVIDIA Turing TU104 architecture. Green sections are CUDA cores, which together compose Streaming Multiprocessors (SMs) [21]

simple program is scaling a matrix as follows:

$$3\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} 3a & 3b \\ 3c & 3d \end{bmatrix}$$

This computation can be distributed over 4 GPU cores, which would each scale one element of the matrix. The program can then accumulate the results of the four operations, and store the results either by overwriting the old matrix or by creating a new one in memory. While this is a small example, the same principles can be applied to matrices with thousands of elements - as the dimensions increase, so does the benefit of using GPUs.

NVIDIA conveniently provides the functionality to distribute a program's workload across GPU cores in the form of a rich C++ API [8]. This API can be integrated seamlessly into existing codebases

in such a way that the programmer is abstracted away from the low-level architecture of the GPU they are using. In recent years, highly-tuned primitives for machine learning-specific applications (e.g. convolution, softmax, batch normalization, neuron activations, etc.) have become publicly available [9], which further allow programmers to use GPUs to their benefit.

2.3 Inference and Training

Neural networks are used in two modes: *training* and *inference*. While these two modes are closely related, they have certain key differences which ultimately affect the way they each interact with GPUs. A useful neural network must first be trained - this process involves iteratively improving the network's trainable parameters (i.e. weights) which are either initialized randomly or all to zero. Training can further be broken down into two steps: a *forward pass* and a *backward pass*. As it turns out, the forward pass of training is very similar to inference. Thus, the training of a neural network can be thought of as an extension of the inference process.

Both inference and the forward pass of training accept some input data (in the Figure 1 example this would be $[x_1, x_2, x_3, x_4]^T$). These input nodes are passed through the network - whatever its specific topology may be - and transformed through compositions of weight scaling and nonlinear activation functions. Then, at the end of the network the outputs are produced. For inference, these final outputs are almost always the only thing that the user is interested in, since they are the meaningful interpretations of the input data. For the forward pass of training, however, more than just the final outputs must be kept; in fact, nearly all of the intermediate values that were used to propagate the data through the network must be kept for later stages of the training algorithm.

2.4 Backpropagation and Memory

Once the forward pass of training has finished, the first step is to evaluate the outputs of the network based on how accurate their predictions were. These evaluations are done via a *loss function*, which quantifies the difference between the network's outputs and the *ground truth* of the input data (in most cases this refers to the *correct* interpretation of the input data). Referring once again to the previous example, if it was known that Bob in fact wrote the sentence, then the network's output $[y_1, y_2, y_3]^T = [0.983, 0.217, 0.015]^T$ would be compared to the ground truth $[0, 1, 0]^T$ and produce a large loss function result, since y_2 was not close to 1 and y_1 was incorrectly close to 1. While loss functions are beyond the

scope of this paper, more information on them can be found in Appendix C.

Next comes perhaps the most important part of training - backpropagation. Originally gaining traction in the late 1980s [31], backpropagation involves calculating the partial derivatives of the loss function with respect to each weight in the network by using the chain rule. These calculations require the intermediate outputs of each layer, which are generally kept in memory during the forward pass and held until they are needed. Once all the gradients in the network have been calculated, the weights can then be adjusted appropriately. It is this extra space requirement that lies at the heart of why GPU memory is an essential resource. As for how the weights are updated, there are many different algorithms known as *optimizers* available [33] which can be employed. Gradient Descent, perhaps the most ubiquitous optimization algorithm, is known as a first order optimization algorithm. It involves updating each weight in the network by a global scaling factor after all the gradients have been computed. Another popular algorithm is Adam [22], which is known as a second order algorithm. Once again, the exact details of such algorithms are beyond the scope of this paper.

The training process is done with a training dataset, which should be an unbiased representation of what actual data looks like in the real world. For example, it is important that a speech-to-text model is not trained using only the speech of one person, otherwise the network would learn to recognize *their* speech very well but would likely under-perform when fed data from another person's speech. Training datasets are generally very large - for example, a commonly-used object detection dataset known as COCO [23] has 330 000 images and is 37.57 GB in size. This is far beyond the size of a typical GPU's memory. One implementation of Gradient Descent involves using mini-batches, which are uniform-size subsets which together comprise the entire dataset. Each example within a mini-batch has the training algorithm applied to it, and only once the entire mini-batch is completed are the weights updated. A breakdown of the entire training process is summarized in Figure 3.

2.5 Machine Learning Frameworks and Memory Profilers

In practice, DNNs are implemented via frameworks such as Caffe [17], Theano [2], MXNet [4], TensorFlow [1], CNTK [32], Chainer [37], Torch [5], Keras [19], and PyTorch [27]. These frameworks provide convenient interfaces which enable researchers to codify their DNN architecture in common programming languages including Python, C++, and Java. Despite the rise in popularity of these frameworks in recent years, comprehensive tools to understand their memory usages are not provided out of

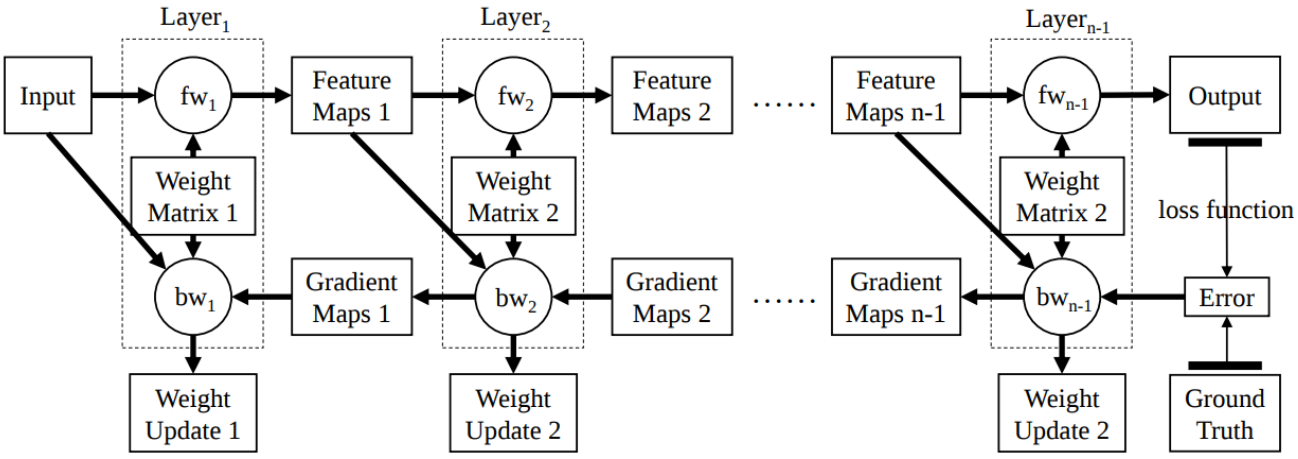


Figure 3: A breakdown of the training process, courtesy of TBD [41]

the box. The EcoSystem research team at UofT, led by Prof. Gennady Pekhimenko, has recently developed open-source memory profiling tools for MXNet [4], TensorFlow [1], and CNTK [32]; such a tool for PyTorch, however, remains to be created. PyTorch is increasingly becoming one of the most widely-used frameworks and thus a memory profiler would have a large impact to the community.

All of these machine learning frameworks employ directed acyclic *computational graphs*, which organize sequences of operations on data. The nodes of computational graphs are the program's variables (tensors, scalar values, etc.), while the edges represent data dependencies. Edges terminating at a node signify inputs to a certain mathematical function, whose output value is stored in the variable representing the node. For the purposes of backpropagation, each node must also know how to compute the derivative of its output with respect to its inputs. In Figure 4, the green edges represent the flow of data in a forward pass. Once the output e has been calculated, the backward pass can be initiated, during which the gradients are calculated by traversing the graph in the reverse direction (red edges).

The frameworks that these memory profilers were created for all use static computational graphs. MXNet, for example, allows users to define computational symbols in their front end language which are then compiled under the hood. Since this graph is defined at the start, the program does not need to wait until the forward pass to create the graph. This compiled graph contains all placeholders for input and output tensors. When it comes time to use the graph - i.e. pass some input data through the network in the forward pass - MXNet internally calls its *executor* [34]. The executor will then allocate the memory for the tensors required for the graph. This point is important because specific memory regions of the GPU

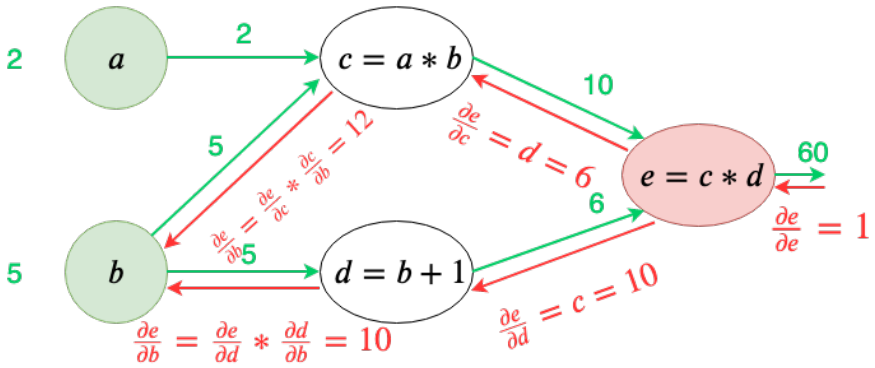


Figure 4: A simple computational graph [20]

are linked to parts of the computational graph, and remain linked for the duration of the program. To derive meaning from what each part of the computational graph does - for example the part of the graph representing a Long Short-Term Memory (LSTM) cell [13] within a speech recognition model - the naming employed by the programmer is used. Listing 1 shows an example of how a programmer might define an LSTM cell in their code. Notice the name assignments that are inputted as function parameters - i2h meaning "input to hidden" and h2h meaning "hidden to hidden" - which are meaningful descriptors of the neural network's layers.

```
i2h = symbol.FullyConnected(data=inputs, weight=self._iW, bias=self._iB,
                            num_hidden=self._num_hidden*4,
                            name='%si2h'%name)
h2h = symbol.FullyConnected(data=states[0], weight=self._hW, bias=self._hB,
                            num_hidden=self._num_hidden*4,
                            name='%sh2h'%name)
```

Listing 1: Symbolic naming of an LSTM cell [40]

Thus, the picture so far is that a user defines their network and training/inference procedures in a front end language such as Python, using existing machine learning framework APIs. This framework interprets the user's code, and distributes the computational work to one or more GPUs (CPUs can be used as well, but for reasons outlined earlier GPUs are preferred). There are APIs used by the framework to interact with the GPU, which may vary as well. For NVIDIA GPUs, CUDA APIs are used,

which interact with C++, C, and Fortran. For this reason, along with the fact that interpreted languages such as Python are generally less efficient than compiled languages, machine learning frameworks are themselves primarily written in C or C++.

For the purposes of this paper, one CUDA function is of particular importance: `cudaMalloc()`. This function allows the framework to request any number of bytes in memory on the device[1]. Each call to `cudaMalloc()` will return a pointer to where the requested memory lies; if space for a tensor was requested, then this pointer might be used for the remainder of the program to access and modify this tensor. However, it is ultimately up to the framework to manage its memory, and different frameworks have different strategies.

The MXNet memory profiler's main challenge is linking symbol names to specific `cudaMalloc()` calls. This linkage allows the profiler to establish how much memory was needed for each part of the computational graph, and therefore can determine the memory breakdown of the models' weights, feature maps, and gradients. However, there is no direct way to do this in the existing MXNet codebase, and thus it was modified to make these connections. Function signatures were changed so that the strings representing symbol names could be passed down to the underlying allocations. A C++ helper class was created to output entries into a log file with information about the memory breakdown. The log file can then be parsed with provided scripts to create visualizations of the results.

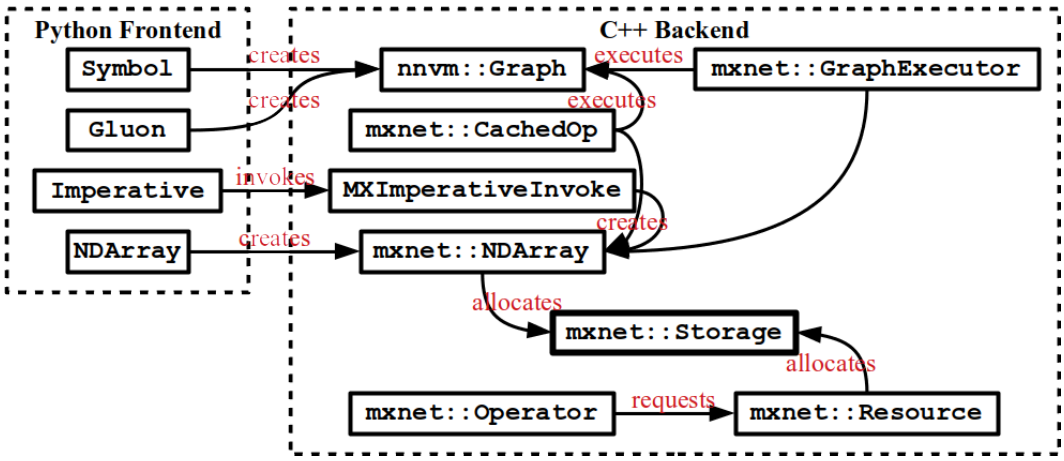


Figure 5: MXNet GPU memory profiler system design [40]

[1]Typically, *device* refers to the GPU and *host* refers to the CPU

2.6 Memory Caching in PyTorch

There are many underlying differences between PyTorch and the frameworks for which these existing memory profilers were created. The first difference - and perhaps most important one in the context of this paper - is how PyTorch handles its device memory. Unlike other frameworks, PyTorch uses *caching* as its primary memory management strategy. The reason for this lies in the fact that PyTorch does not create a static computational graph with pre-allocated room for variables and tensors. This new paradigm, which is discussed in section 2.9, is known as *dynamic computational graphs*. For now it suffices to say that PyTorch does not know how the forward and backward pass will be executed until runtime, i.e. until training or inference begins. Caching is best explained by the creators of PyTorch [27]:

> " PyTorch implements a custom allocator which incrementally builds up a cache of CUDA memory and reassigns it to later allocations without further use of CUDA APIs. The incremental allocation is also crucial for better interoperability, because taking up all GPU memory ahead of time would prevent the user from utilizing other GPU-enabled Python packages. To further improve its effectiveness, this allocator was tuned for the specific memory usage patterns of deep learning. For example, it rounds up allocations to multiples of 512 bytes to avoid fragmentation issues. Moreover, it maintains a distinct pool of memory for every CUDA stream (work queue). "

By incrementally building up a cache of memory as-needed during runtime, PyTorch does not need to make frequent `cudaMalloc()` calls to reserve space for each tensor as the forward or backward pass progress. Instead, it reserves chunks of space which can be assigned and unassigned freely by PyTorch. The allocation strategy is relatively straightforward: when a tensor is needed by the program, it can request space for it using an allocator. If there is enough space in the existing cache, it takes memory from there. If not, the cache size is increased, and then the tensor is assigned memory from the cache.

In general, deallocation strategies are more nuanced than their allocation counterparts. There is no reason for a tensor to occupy space in device memory if the tensor is no longer being used by the program. Other machine learning frameworks generally use *garbage collection* techniques to free

memory once it is no longer needed. With this technique, the program will periodically analyze the state of all objects in its memory. All objects currently in use - i.e. tensors that are needed for future computations - are marked. The garbage collector will then free all objects in the address space that are not marked; these objects are of no use to the program. The issue with garbage collection is the timing of the collection - for the duration between a tensor becoming unneeded and garbage collection occurring there is wasted device memory.

PyTorch does not use garbage collection. Instead, to determine whether a certain tensor is needed or not, a technique called *reference counting* is employed. The reference counter of a tensor is modified dynamically at runtime based on how many variables reference it. This system tracks both the back end references (PyTorch's internal references to the tensor) as well as the front end references (user-created references in their Python code) to ensure that a tensor is only freed when there is truly no use for it anymore. When a reference counter reaches zero, the memory occupied by the tensor is immediately released back to the cache. This means that at no point will there be objects with zero references which still occupy device memory.

By gradually increasing the cache size as the program progresses, the majority of the CUDA memory API calls will occur near the start of the program, particularly in the first few iterations[2] of training. This is illustrated in Figure 6, where the utilization of the GPU (density of CUDA kernel execution in blue) is low in the first iteration compared to subsequent iterations. This decreased computational efficiency can be attributed to the costly memory management APIs.

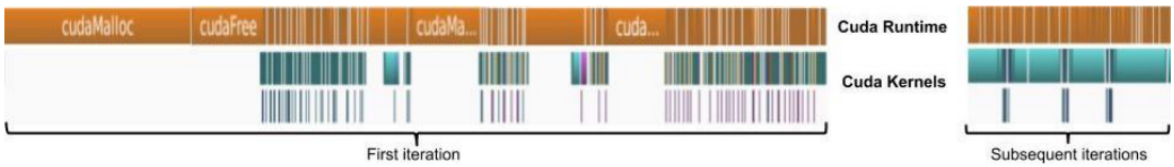


Figure 6: ResNet-50 GPU execution traces in PyTorch [27]

2.7 Tensors in PyTorch

Thus far, tensors have been mentioned but not yet explained in detail. Tensors are n-dimensional data structures which contain a certain type of data, for example integers or floats. The actual data represented by a tensor - for example a matrix representing the scalar weights from the first to the second

[2] An iteration refers to the combined forward and backward pass of one mini-batch through the network

hidden layer of a network - are accompanied by some metadata which describes various attributes about the tensor. The metadata of a tensor includes its dimensions (depth, height, width), strides, element data type, and the device it is stored on. Among these, strides may be an unfamiliar one - in fact, striding is one of the defining features of PyTorch. To explain striding, it must first be understood that every tensor is associated with a **storage** object. A tensor can be thought of as the logical view of data, while a storage can be thought of as a structure which manages the raw data. It is entirely possible in PyTorch for two tensors to point to the same storage object, with each of the two tensors having different perspectives about how to view the underlying raw data. For example, suppose a PyTorch program created the following two tensors:

$$X = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}, Y = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Both X and Y contain the integers from 1 to 4; however, they each provide different logical representations of the data. This information is captured by the stride of a tensor, which are offsets to map element indices into specific memory addresses. Storages do not understand the data they encapsulate, it is up to the tensors to provide the interpretation. The reference counters explained in section 2.6 are in fact handled by storage objects. When no tensors point to a certain storage anymore, the memory occupied by the storage is release to the cache. Given that this paper is concerned with the memory breakdown of PyTorch, two questions arise:

(i) How can a tensor's link to underlying device memory be ascertained, and how can a memory profiler safely handle two or more tensors which point to the same underlying storage?

(ii) Given (i), how can this link be translated into a specific size in bytes, and how can a memory profiler take into account PyTorch's unique caching memory management strategy, where CUDA memory allocations are no longer directly tied to a specific object within the framework?

These questions are answered in later sections, where the proposed PyTorch memory profiler is described.

2.8 Models in PyTorch

Tensors can only get a user so far. Ultimately, researchers need an interface to codify their machine learning models, which can then be trained and used. The main interface used to do this in PyTorch is the `nn.Module` class. For example, Listing 2 defines a simple two-layer network:

```
class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred
```

Listing 2: Definition of a simple two-layer neural network from the PyTorch documentation [18]

Here, a network called `TwoLayerNet` is being defined. Notice how it inherits from `nn.Module`. The `__init__()` method of this network is invoked whenever an instance of it is created and a user generally instantiates one model which persists for the duration of their PyTorch program. This method is where the *structure* of the model is defined - how many and what types of layers are present, which layers connect to what, and so on. The code here can be as complex or as simple as the user wishes; this modularity is what enables PyTorch to implement arbitrary neural networks. This example creates two linear layers using the PyTorch-provided `torch.nn.Linear` [3] implementations [6]. The input parameters `D_in` and `D_out` are used to specify the dimensions of these two layers.

One interesting feature of the `__init__()` method is that all *learnable* layers of the network - i.e. ones that must be trained, which must have their gradients computed - are automatically registered and saved to the model's list of `parameters`. Using the `torch.nn.Linear` API handles this by default, but if one were to define a learnable tensor within a model themselves, they would have to wrap them with `torch.nn.Parameter`. For example, observe the following two tensor definitions in Listing 3:

[3] The `torch.nn.Linear` layer applies the linear transformation $y = xA^T + b$

```
import torch
class MyNetwork(torch.nn.Module):
    def __init__(self):
        super(MyNetwork, self).__init__()
        self.wont_be_registered=torch.tensor([1.0])
        self.will_be_registered=torch.nn.Parameter(torch.tensor([1.0]))
```

Listing 3: Example of how to create a tensor which will be saved in the `state_dict` of a model

Here, one tensor is defined correctly and thus will forever be saved in a special data structure by the PyTorch back end code. This data structure is called the `state_dict`. The other is not, and will not be added to the `state_dict`. Wrapping a tensor in this way does not change any of its functionality. To verify that tensors are being saved as expected, an instance of the model is instantiated and then a built-in class function called `named_parameters()` is invoked to reveal the named items within the model's `state_dict`:

```
model=MyNetwork()
for name,param in model.named_parameters():
    print(f"Tensor name={name}, Tensor data={param}")
```

The output is as follows:

```
>>> Tensor name=will_be_registered, Tensor data=Parameter containing:
    tensor([1.], requires_grad=True)
```

This convenient saving of layer names has many similarities to the one previously discussed in MXNet in section 2.5, and will serve to be useful in creating a new memory profiler for PyTorch given the mapping it provides between layer names and their data objects.

2.9 Dynamic Computational Graphs and Autograd

The network definition in Listing 2 has one other important definition, namely its `forward()` method. This method describes the exact sequence of calculations that will be undertaken during the forward pass of the network, both during training and inference. This example is trivial, and the backward pass procedure could be defined manually with some basic calculus and a few lines of code. However, modern neural networks are very complex and have detailed computational procedures that define their forward passes. Manually defining a backward pass each time would be an arduous and error-prone

process, and thus a more scalable system is required.

PyTorch uses **automatic differentiation** [24], implemented within the `torch.autograd` package, to take care of this problem. Distinct from other algorithmic methods such as symbolic and numerical differentiation, automatic differentiation is founded on the observation that any arbitrary computer program can be decomposed into arithmetic operations ($+, -, *, \div$) and elementary functions ($log, sine, cosine, exp$, etc.). Recursively applying the chain rule allows the program to calculate derivatives (subject of course to floating point precision errors) using only an order of $O(n)$ extra computation. In practice, the linear factor is typically small in magnitude, leading to only a small decrease in performance.

During each iteration, PyTorch will gradually create a computational graph *dynamically* as the program executes - this is a large departure from other frameworks, such as MXNet, which generally create a *static* graph at the start of the program. When PyTorch starts the backward pass with a $.backward()$ call, the graph will be walked back and gradients will be calculated. Returning to Figure 4, the gradient of the output node e with respect to the input node b could not be calculated directly; because of the nature of the chain rule, the gradients of *intermediate* nodes were required for the calculation. These occupy space in memory and must not be neglected. Notice also that derivatives with respect to a were *not* calculated; for this reason, it cannot be assumed that every node will participate in PyTorch's backward pass procedure. In fact, if the gradient of a with respect to the output e were not required, and there were other intermediate nodes in the graph which were not needed for computing the gradient of e, these nodes would not have their gradients calculated either. Thus, it cannot be assumed that every tensor involved in a PyTorch computational graph will necessarily have its gradients computed and stored in memory. More generally, if a tensor has `requires_grad=True`, then PyTorch must calculate gradients for it. The exact behaviour is outlined in the PyTorch documentation [7].

3. PyTorch Memory Profiler

This section describes the design and usage of the proposed PyTorch memory profiler. The profiler program is available for use on GitHub [4], with detailed setup and usage instructions.

3.1 Designing the Profiler

In the face of all the internal complexities that have been outlined thus far, it should be clear that new mechanisms must be created to profile PyTorch's device memory usage. A proposed strategy to attain a layer-by-layer breakdown into weights, feature maps, and gradients is developed here using knowledge and methods that have been developed in previous sections. It is noted that this strategy departs from the strategies used with existing profilers in that it only uses the front end Python API available in PyTorch. This means that the profiler is more resilient to future changes in the underlying C++ PyTorch codebase. Given the particularly fast-changing nature of PyTorch, with several recent large-scale design changes such as the variable-tensor merger [29], having a less intrusive profiler that does not have to be re-coded every update is a significant benefit.

First, let us return to the two questions posed in section 2.7. Question (i) requires linking tensors to their device memory. To solve this, there is a fundamental structure in PyTorch known as a **DataPtr**. Defined in `c10/core/Allocator.h` in the source code, these structures are described as follows:

> " A DataPtr is a unique pointer (with an attached deleter and some context for the deleter) to some memory, which also records what device is for its data."

The DataPtr of a tensor is saved in the `storage` object that the tensor points to, and can be accessed by calling `tensor.storage().data_ptr()`. This also provides a way to determine if two tensors with different names actually share the same underlying memory, and thus should not be counted twice when diagnostics are reported. A hash table which maps unique DataPtrs to the model's layers (i.e. tensors) is used by the profiler to hold relevant information about naming and memory usage. These DataPtrs are also used to keep track of gradient and feature map tensors. It is important that the memory profiler **does not store a direct reference to the tensors themselves**, since this would prevent the reference counting system from freeing them as expected, and thus incorrectly increase the memory usage of the program.

[4] https://github.com/izaakniksan/engsci_thesis/tree/master/pytorch_mem_profiler

Question (ii) involves measuring the amount of memory used by a tensor. Unfortunately there is no direct way to do this, but a convenient calculation can be used instead. This calculation requires multiplying the number of elements by the size per element of a tensor. These values are spread out between the tensor and storage objects, and the total memory can be computed as follows:

```
total_memory = tensor.storage().size() * tensor.element_size()
```

This calculation is used by the profiler to calculate how much memory a given tensor is using at a specific time during execution. It is noted that, as mentioned in section 2.6, allocations are rounded to 512 bytes and this must be taken into account by the profiler.

Feature maps are the scalar outputs of neurons, and gradients are the derivatives necessary for backpropagation. To create a memory profiler, the tensors which represent these data structures must be properly accounted for. During the forward pass of training, feature maps are gradually computed and stored in tensors. Then, during the backward pass, gradients are computed using these feature maps. Immediately after the backward pass, all of these intermediate tensors are no longer needed, and are cleared. At this point, only the gradients of **leaf tensors** [7] of the computational graph - i.e. the ones ones which must be improved during the iteration - are kept in memory. These gradients can then be used by an optimizer (e.g. Adam) to take a step in the direction which reduces the loss function of the network.

This complicated set of procedures - where the total allocated memory routinely fluctuates throughout training, and where tensor memory is not even linked directly to `cudaMalloc()` calls but rather is simply taken and returned to a global cache - provides new challenges not faced when tracking the memory of a static graph framework. Luckily, however, PyTorch provides some hidden tools which help shed light on the forward and backward pass: **hooks**.

There are several variants of hooks in PyTorch, but unfortunately they have very little documentation. Still, the insight they bring is valuable, despite the fact that most information about them is only found spread across the PyTorch forums [28]. Hooks are *functions* which can be registered on either a `nn.Module` or a `Tensor` object. The three methods used to register hooks are: `register_hook()`, `register_forward_hook()`, and `register_backward_hook()`. Depending on the type of hook, during the backward or forward pass these functions are called. Hooks allow direct access to the intermediate feature maps and gradients at the time of their creation - exactly what is needed for a

memory profiler.

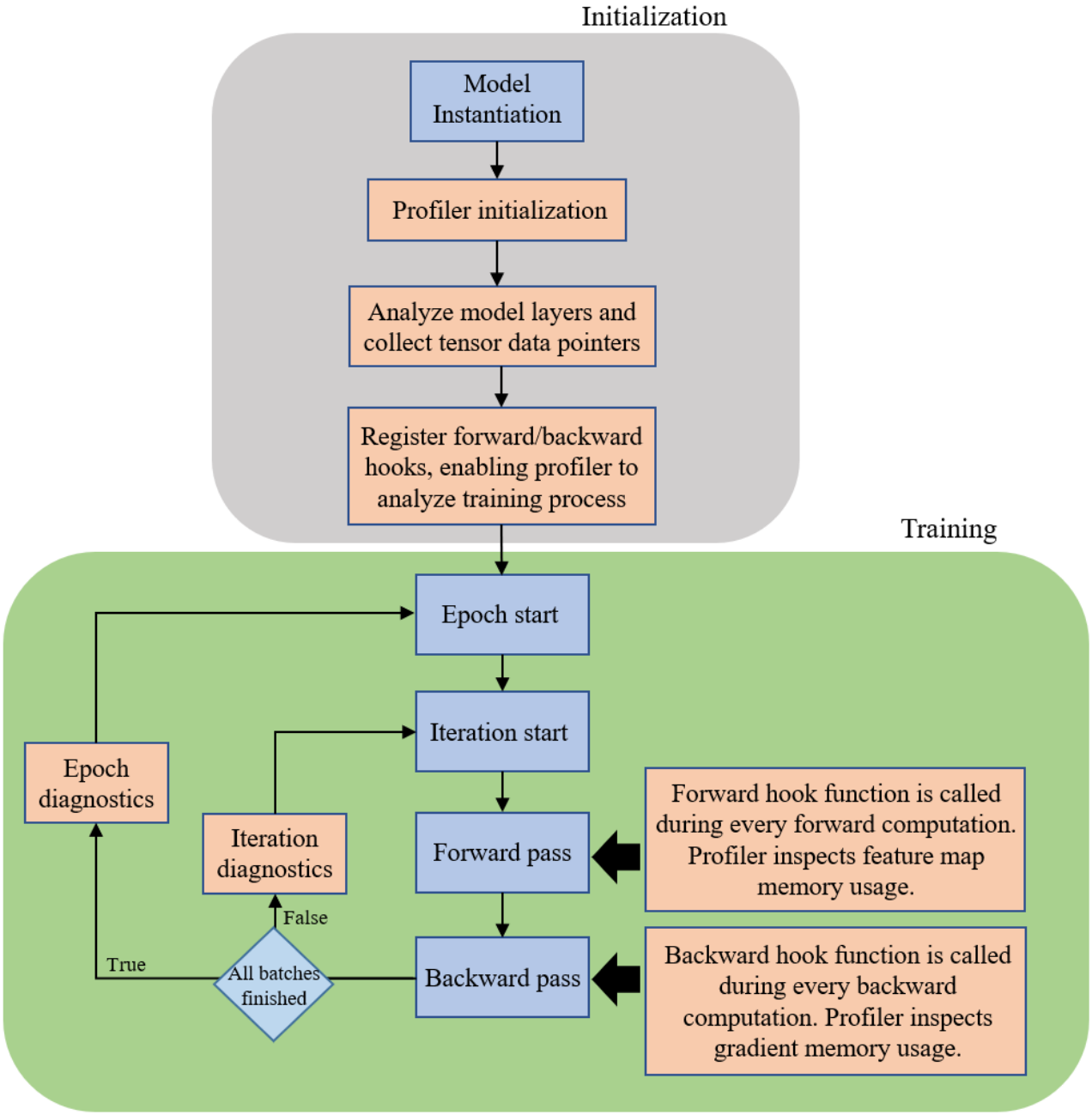


Figure 7: System diagram of the proposed PyTorch memory profiler

Figure 7 shows a system diagram of the PyTorch memory profiler. Blue steps comprise the normal training workflow, and orange steps are the modifications to the flow done autonomously by the profiler. After the model has been has been instantiated by the training program, a `memory_profiler` instance is created. This object contains all the functionality necessary for the memory usage to be profiled. When

it is initialized, it will analyze the layers of the model and save their data pointers in a hash table. The layer names are extracted as well, which provide meaningful descriptions of their use.

Next, the profiler registers forward and backward hooks. Since the PyTorch Autograd engine accumulates values necessary to walk back the computational graph (i.e. backpropagation) internally in C++ buffers, hooks provide otherwise-unavailable access to intermediate tensors. Hooks allow the profiler to inspect feature maps and gradients as PyTorch creates them. The hooks that are used are specialized functions defined within the profiler to gather memory consumption metrics. Given that a PyTorch model may have many nested submodules, the profiler recursively registers hooks on all such submodules.

During the training algorithm, the forward and backward passes execute as normal. However, the hooks registered earlier are automatically run after every intermediate computation. The profiler keeps track of the memory usage by each part of the model, and reports relevant diagnostics after each iteration. As the profiler inspects tensors, it continually stores their DataPtrs and maps them to their meaning; for example, if an input tensor to a forward hook function has never been inspected before, the profiler will understand that the tensor's purpose is a feature map, and will increase the feature map memory usage by the computed memory usage of that tensor. Given that DataPtrs are unique, this ensures that no intermediate tensor is mistakenly counted twice by the memory profiler. The cache is also monitored continually by the profiler, and statistics about its size are gathered and reported at the end of each iteration.

3.2 Profiler Setup

The prerequisites to use the memory profiler are:

- Python 3.6 or later
- PyTorch (https://pytorch.org/)
- CUDA (any version, as long as the training code uses it)

One of the largest benefits of this memory profiler is that there is no lengthy installation process. Other existing profilers require downloading a specific version of the machine learning framework, then patching the source code, then building the library from source; this PyTorch profiler is not version-dependent and no re-installation of the library is required.

Once the GitHub repo has been cloned, add the path to the profiler to your path in your training code as follows:

```
import sys
sys.path.append('<path to pytorch_mem_profiler/ directory>')
from pytorch_mem_profiler import *
```

Listing 4: How to import the PyTorch memory profiler into existing training code

3.3 Profiler Usage

There are only three steps (and only four lines of code) required to start using the profiler:

- Create a profiler instance
- Call `.record_stats()` after every iteration
- Call `.epoch_end()` after every epoch

The usage of the profiler will be demonstrated using the Neural Collaborative Filtering [12] model as an example. Listing 5 shows an example from training code [36] which instantiates a model. On line 14, the model is sent to the GPU. This code is unchanged by the profiler - it is simply shown here as an example of a typical PyTorch program.

```
...
# Create model
model = NeuMF(nb_users, nb_items,
              mf_dim=args.factors, mf_reg=0.,
              mlp_layer_sizes=args.layers,
              mlp_layer_regs=[0. for i in args.layers])

# Add optimizer and loss to graph
optimizer = torch.optim.Adam(model.parameters(), lr=args.learning_rate)
criterion = nn.BCEWithLogitsLoss()

if use_cuda:
    # Move model and loss to GPU
    model = model.cuda()
    criterion = criterion.cuda()
```

Listing 5: Initialization of an NCF model

Next, right before the training loop, an instance of the `memory_profiler` class must be instantiated. The model is passed in as an input argument, along with two other optional arguments:

- **print_period** (postive integer) determines the number of iterations between memory reporting (default value is 1).
- **csv** (boolean) allows profiling data to also be exported into a .csv file located in `./memory_csv_data/` (default value is False).

The profiler initialized in Listing 6 will report statistics every 5 iterations to the terminal and to a csv file. The `global` keyword ensures that the profiler is accessible anywhere within the main training program.

```
global profiler
profiler = memory_profiler(model, print_period=5, csv=True)
```

Listing 6: How to create an instance of the PyTorch memory profiler

The final step is to call `.record_stats()` at the end of each iteration and `.epoch_end()` at the end of each epoch. In the training loop shown in Listing 7, Lines 33 and 35 were added:

```
# Epoch loop
for epoch in range(args.epochs):

    model.train()
    losses = utils.AverageMeter()

    begin = time.time()
    loader = tqdm.tqdm(train_dataloader)
    length = len(loader)

    # Iteration loop
    for batch_index, (user, item, label) in enumerate(loader):
        user = torch.autograd.Variable(user, requires_grad=False)
        item = torch.autograd.Variable(item, requires_grad=False)
        label = torch.autograd.Variable(label, requires_grad=False)
        if use_cuda:
            user = user.cuda(async=True)
            item = item.cuda(async=True)
```

```
19            label = label.cuda(async=True)
20
21        outputs = model(user, item)
22        loss = criterion(outputs, label)
23        losses.update(loss.data.item(), user.size(0))
24
25        optimizer.zero_grad()
26        loss.backward()
27        optimizer.step()
28
29        # Save stats to file
30        description = ('Epoch {} Loss {loss.val:.4f} ({loss.avg:.4f})'
31                       .format(epoch, loss=losses))
32        loader.set_description(description)
33        profiler.record_stats()
34
35    profiler.epoch_end()
36    hits, ndcgs = val_epoch(model, test_ratings, test_negs, args.topk,
37                            use_cuda=use_cuda, output=valid_results_file,
38                            epoch=epoch, processes=args.processes)
```

Listing 7: Example NCF training code with lines 33 and 35 added to enable memory profiling

The results will be printed as the training progresses. Since PyTorch uses a memory caching strategy, tensors dynamically take and release from the GPU memory cache. The profiler will give you insight into the cache size, as well as a detailed layer-by-layer breakdown of what the memory is being used for. As the training is running, the csv output file located in `./memory_csv_data/` will become populated with memory diagnostics. Additionally, the following table will be printed to the screen:

```
*****************************************
Memory Usage for Iteration 85 of Epoch 1
*****************************************
Peak cached.........................857 MB
Current cached......................857 MB
Total feature map usage.............301 MB
```

```
Total layer weight usage.............127 MB
  mf_user_embed.weight................35 MB
  mf_item_embed.weight.................7 MB
  mlp_user_embed.weight...............71 MB
  mlp_item_embed.weight...............14 MB
  mlp.0.weight.........................0 MB
  mlp.0.bias...........................0 MB
  mlp.1.weight.........................0 MB
  mlp.1.bias...........................0 MB
  mlp.2.weight.........................0 MB
  mlp.2.bias...........................0 MB
  final.weight.........................0 MB
  final.bias...........................0 MB

Total layer weight gradient usage....127 MB
  mf_user_embed.weight grad...........35 MB
  mf_item_embed.weight grad............7 MB
  mlp_user_embed.weight grad..........71 MB
  mlp_item_embed.weight grad..........14 MB
  mlp.0.weight grad....................0 MB
  mlp.0.bias grad......................0 MB
  mlp.1.weight grad....................0 MB
  mlp.1.bias grad......................0 MB
  mlp.2.weight grad....................0 MB
  mlp.2.bias grad......................0 MB
  final.weight grad....................0 MB
  final.bias grad......................0 MB
Intermediate gradients...............34 MB
```

4. Profiling Results

This section presents memory profiling results for the training of modern deep neural networks. All models presented here have open-source implementations and training code available on TBD Suite [36]. The system used for training benchmarks is described in Table 1.

Type	Component Used
CPU	AMD Ryzen 7 2700X, 4.3 GHz Max Boost, 8 Cores
GPU	NVIDIA GeForce GTX 1080 Ti, 11 GB GDDR5X
OS	Ubuntu 16.04 LTS
Host Memory	16 GB DDR4, 3200 MHz

Table 1: Specification of system used for profiling

4.1 Neural Collaborative Filtering (NCF) Training Benchmarks

This section presents the memory breakdown for the training of a Neural Collaborative Filtering [12] model with PyTorch. The MovieLens 20M dataset [25] was used for input training data. The hyperparameters used for all training experiments are listed in Table 2.

Hyperparameter	Value
Learning rate	0.0002
Number of predictive factors	64
Sizes of hidden layers for MLP	256, 256, 128, 64
Number of negative examples per interaction	8
Threshold	0.635
Number of processes	1
Number of dataloader workers	0

Table 2: Hyperparameters used for NCF memory profiling experiments

Figure 8 shows the memory usages of the feature maps, layer weights, and layer weight gradients for different mini-batch sizes. Feature maps consume the majority of the GPU memory, which is especially evident as the mini-batch size increases. The memory usage of layer weights and the gradients of these layer weights are not dependent on the mini-batch size. Values were recorded once the training memory usage remained constant.

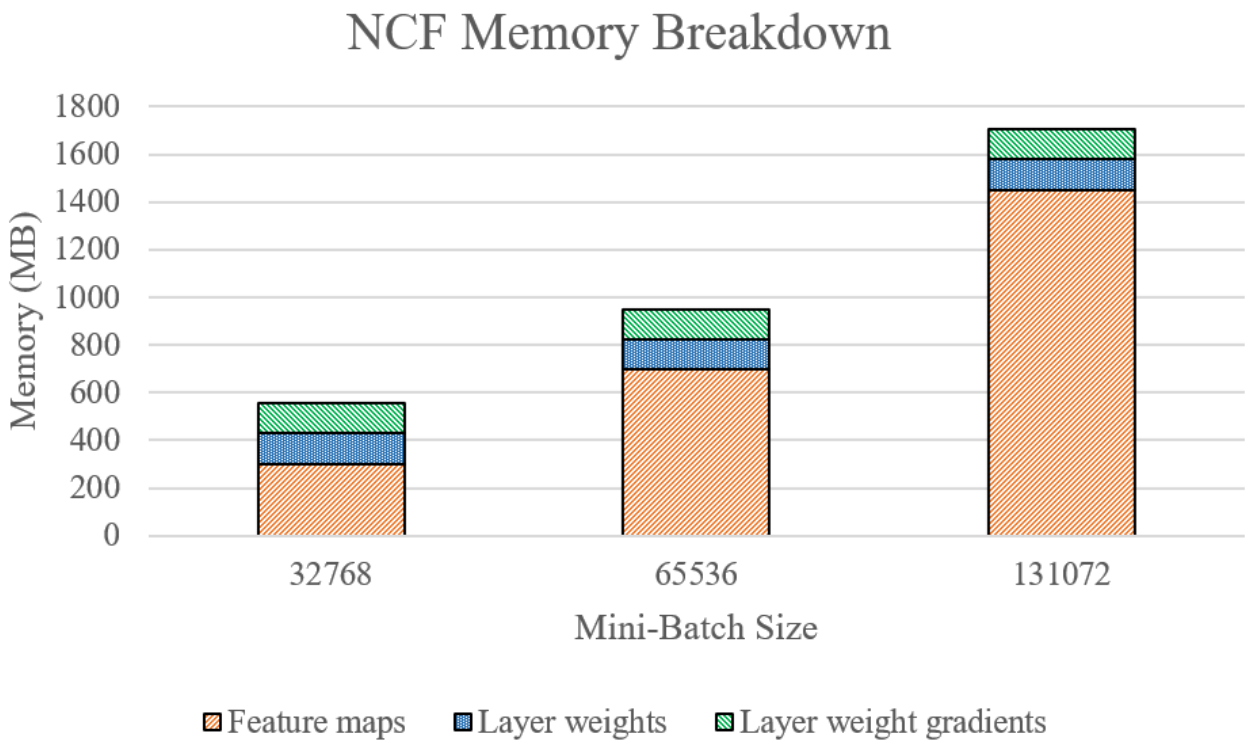


Figure 8: Feature map, layer weight, and layer weight gradient memory usage of NCF training for different batch sizes

Table 3 shows a more detailed memory breakdown. The intermediate gradients listed in the last row are tensors which were seen by the profiler's hooks during the backward pass, but whose memory was not attributable to the named layer weight gradients. The memory consumed by these intermediate tensors did increase with mini-batch size. Values were recorded once the training memory usage remained constant.

Type \ Batch Size	32768 Ratings	65536 Ratings	131072 Ratings
peak cached	**857**	**1107**	**1714**
total feature map usage	**301**	**697**	**1451**
total layer weight usage	**127**	**127**	**127**
mf_user_embed.weight	35	35	35
mf_item_embed.weight	7	7	7
mlp_user_embed.weight	71	71	71
mlp_item_embed.weight	14	14	14
mlp.0.weight	0	0	0
mlp.0.bias	0	0	0
mlp.1.weight	0	0	0
mlp.1.bias	0	0	0
mlp.2.weight	0	0	0
mlp.2.bias	0	0	0
final.weight	0	0	0
final.bias	0	0	0
total layer weight gradient usage	**127**	**127**	**127**
mf_user_embed.weight grad	35	35	35
mf_item_embed.weight grad	7	7	7
mlp_user_embed.weight grad	71	71	71
mlp_item_embed.weight grad	14	14	14
mlp.0.weight grad	0	0	0
mlp.0.bias grad	0	0	0
mlp.1.weight grad	0	0	0
mlp.1.bias grad	0	0	0
mlp.2.weight grad	0	0	0
mlp.2.bias grad	0	0	0
final.weight grad	0	0	0
final.bias grad	0	0	0
intermediate gradients	**34**	**101**	**203**

Table 3: Neural Collaborative Filtering memory breakdowns for varying batch sizes. All values rounded to the nearest MB.

4.2 ResNet-50 Training Benchmarks

This section presents the memory breakdown for the training of a ResNet-50 v2 [11] model in PyTorch. The ImageNet LSVRC 2012 dataset [14] was used for input training data. The hyperparameters used for all training experiments are listed in Table 4.

Hyperparameter	Value
Learning rate	0.1
Number of dataloader workers	4

Table 4: Hyperparameters used for ResNet-50 v2 memory profiling experiments

Figure 9 shows the memory usage of the feature maps, layer weights, and layer weight gradients for different mini-batch sizes. As was the case with the NCF model, again the feature maps consume the majority of the GPU memory. Given the large number of layers, a full layer-by-layer memory breakdown is presented in Appendix A. Values were recorded once the training memory usage remained constant.

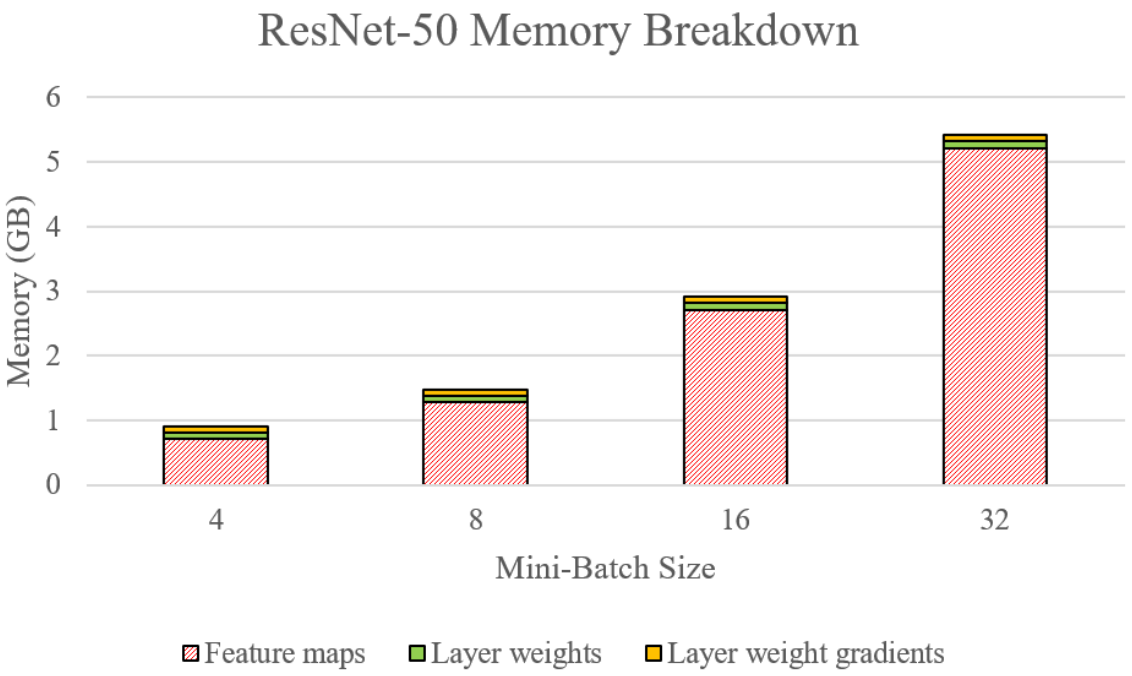


Figure 9: Feature map, layer weight, and layer weight gradient memory usage of ResNet-50 v2 training for different batch sizes

Using existing memory profilers, the EcoSystem research group has gathered memory breakdowns for ResNet-50 implementations in other frameworks. While the system specifications of those experiments are not identical to those used for the PyTorch experiments (notably, a Quadro P4000 GPU was used), the breakdowns are similar. This further demonstrates the accuracy of this proposed PyTorch memory profiler and its ability to correctly characterize training memory usage. The results for those breakdowns are shown in Figure 10.

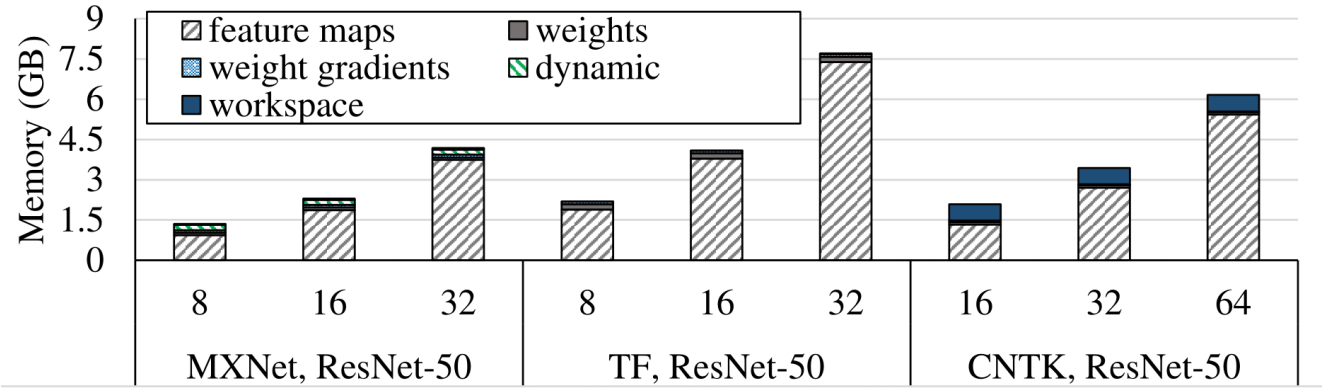


Figure 10: Memory breakdowns of ResNet-50 training for other frameworks [35]

5. Conclusion

In this paper, a novel memory profiling tool for PyTorch was proposed. Given the differences between how PyTorch handles GPU memory compared to other frameworks - namely its use of caching and dynamic computational graphs - new methods were employed to design and implement this profiler. The profiler is easily integrated into existing training code, and has proven to be effective in capturing the training memory usage of the Neural Collaborative Filtering and ResNet-50 models. Given the importance of GPU memory in the training of large and complex modern deep neural networks, and with the increasing popularity of PyTorch, a tool which provides insight into how memory is being used will benefit the community of researchers. Due to the recent rise of multi-GPU training, extending this profiler to provide a device-based breakdown of memory usage is an area for future work. As new models are created and implemented, it is hoped that this PyTorch memory profiler will shed light on new optimization techniques that can increase training efficiency and performance.

References

[1] Martın Abadi et al. “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems”. In: *arXiv e-prints*, arXiv:1603.04467 (Mar. 2016), arXiv:1603.04467. arXiv: 1603.04467 [cs.DC].

[2] James Bergstra et al. “Theano: A CPU and GPU Math Compiler in Python”. In: 2010.

[3] Jason Brownlee. *Primer on Neural Network Models for Natural Language Processing*. Aug. 2019. URL: https://machinelearningmastery.com/primer-neural-network-models-natural-language-processing/.

[4] Tianqi Chen et al. “MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems”. In: *arXiv e-prints*, arXiv:1512.01274 (Dec. 2015), arXiv:1512.01274. arXiv: 1512.01274 [cs.DC].

[5] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. “Torch7: A Matlab-like Environment for Machine Learning”. In: (2011). URL: http://infoscience.epfl.ch/record/192376.

[6] Torch Contributors. *Documentation for PyTorch*. 2019. URL: https://pytorch.org/docs/stable/nn.html.

[7] Torch Contributors. $is_leaf conventions in PyTorch$. 2019. URL: https://pytorch.org/docs/stable/autograd.html#torch.Tensor.is_leaf.

[8] *CUDA C++ Programming Guide*. Nov. 2019. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[9] *cudnn-developer-guide*. Nov. 2019. URL: https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html.

[10] Song Han et al. “EIE: Efficient Inference Engine on Compressed Deep Neural Network”. In: *arXiv e-prints*, arXiv:1602.01528 (Feb. 2016), arXiv:1602.01528. arXiv: 1602.01528 [cs.CV].

[11] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *arXiv e-prints*, arXiv:1512.03385 (Dec. 2015), arXiv:1512.03385. arXiv: 1512.03385 [cs.CV].

[12] Xiangnan He et al. “Neural Collaborative Filtering”. In: *arXiv e-prints*, arXiv:1708.05031 (Aug. 2017), arXiv:1708.05031. arXiv: 1708.05031 [cs.IR].

[13] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[14] *ImageNet LSVRC 2012 Dataset*. URL: http://image-net.org/challenges/LSVRC/2012/.

[15] Animesh Jain et al. “Gist: Efficient Data Encoding for Deep Neural Network Training”. In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)* (2018), pp. 776–789.

[16] Katarzyna Janocha and Wojciech Czarnecki. “On Loss Functions for Deep Neural Networks in Classification”. In: *ArXiv* abs/1702.05659 (2017).

[17] Yangqing Jia et al. “Caffe: Convolutional Architecture for Fast Feature Embedding”. In: *arXiv e-prints*, arXiv:1408.5093 (June 2014), arXiv:1408.5093. arXiv: 1408.5093 [cs.CV].

[18] Justin Johnson. *LEARNING PYTORCH WITH EXAMPLES*. 2017. URL: https://pytorch.org/tutorials/beginner/pytorch_with_examples.html.

[19] *keras*. URL: https://github.com/keras-team/keras.

[20] Rafay Khan. *Nothing but NumPy: Understanding Creating Neural Networks with Computational Graphs from Scratch*. June 2019. URL: https://www.kdnuggets.com/2019/08/numpy-neural-networks-computational-graphs.html.

[21] Emmett Kilgariff et al. *NVIDIA Turing Architecture In-Depth*. Sept. 2018. URL: https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth.

[22] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *arXiv e-prints*, arXiv:1412.6980 (Dec. 2014), arXiv:1412.6980. arXiv: 1412.6980 [cs.LG].

[23] Tsung-Yi Lin et al. “Microsoft COCO: Common Objects in Context”. In: *arXiv e-prints*, arXiv:1405.0312 (May 2014), arXiv:1405.0312. arXiv: 1405.0312 [cs.CV].

[24] Charles C. Margossian. “A Review of automatic differentiation and its efficient implementation”. In: *arXiv e-prints*, arXiv:1811.05031 (Nov. 2018), arXiv:1811.05031. arXiv: 1811.05031 [cs.MS].

[25] *MovieLens 20M Dataset*. URL: https://grouplens.org/datasets/movielens/20m/.

[26] Angshuman Parashar et al. “SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks”. In: *arXiv e-prints*, arXiv:1708.04485 (May 2017), arXiv:1708.04485. arXiv: 1708.04485 [cs.NE].

[27] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *arXiv e-prints*, arXiv:1912.01703 (Dec. 2019), arXiv:1912.01703. arXiv: 1912.01703 [cs.LG].

[28] *PyTorch Forum*. URL: https://discuss.pytorch.org/.

[29] *PyTorch Variable Tensor Merger*. URL: https://pytorch.org/blog/pytorch-0_4_0-migration-guide/.

[30] Minsoo Rhu et al. “vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design”. In: *arXiv e-prints*, arXiv:1602.08124 (Feb. 2016), arXiv:1602.08124. arXiv: 1602.08124 [cs.DC].

[31] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning Representations by Back-Propagating Errors”. In: *Neurocomputing: Foundations of Research*. Cambridge, MA, USA: MIT Press, 1988, pp. 696–699. ISBN: 0262010976.

[32] Frank Seide and Amit Agarwal. “CNTK: Microsoft’s Open-Source Deep-Learning Toolkit”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’16. San Francisco, California, USA: Association for Computing Machinery, 2016, p. 2135. ISBN: 9781450342322. DOI: 10.1145/2939672.2945397. URL: https://doi.org/10.1145/2939672.2945397.

[33] Anish Singh Walia. *Types of Optimization Algorithms used in Neural Networks and Ways to Optimize Gradient Descent*. June 2017. URL: https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f.

[34] *Symbolic Executor component of MXNet.* URL: https://mxnet.apache.org/api/python/docs/api/mxnet/executor/index.html.

[35] *TBD Benchmarks*. URL: http://www.cs.toronto.edu/~pekhimenko/tbd/.

[36] *TBD Suite*. URL: https://github.com/tbd-ai/tbd-suite.

[37] Seiya Tokui and Kenta Oono. "Chainer : a Next-Generation Open Source Framework for Deep Learning". In: 2015.

[38] Fjodor Van Veen. *The Neural Network Zoo*. Sept. 2016. URL: https://www.asimovinstitute.org/neural-network-zoo/.

[39] Haohan Wang and Bhiksha Raj. "On the Origin of Deep Learning". In: *arXiv e-prints*, arXiv:1702.07800 (Feb. 2017), arXiv:1702.07800. arXiv: 1702.07800 [cs.LG].

[40] Bojian Zheng. *MXNet GPU Profiler*. May 2019. URL: https://github.com/UofT-EcoSystem/incubator-mxnet/tree/bojian/GPU_Memory_Profiler/example/gpu_memory_profiler.

[41] Hongyu Zhu et al. "TBD: Benchmarking and Analyzing Deep Neural Network Training". In: *arXiv e-prints*, arXiv:1803.06905 (Mar. 2018), arXiv:1803.06905. arXiv: 1803.06905 [cs.LG].

Appendix A. Detailed ResNet-50 Memory Statistics

In this section the layer-by-layer PyTorch memory breakdown of ResNet-50 training for a batch size of 32 is presented. Training was done using system specs listed in Table 1. All values are rounded to the nearest MB and were taken directly from the profiler's csv output after the memory usages remained constant. In the interest of space, only non-zero values are shown.

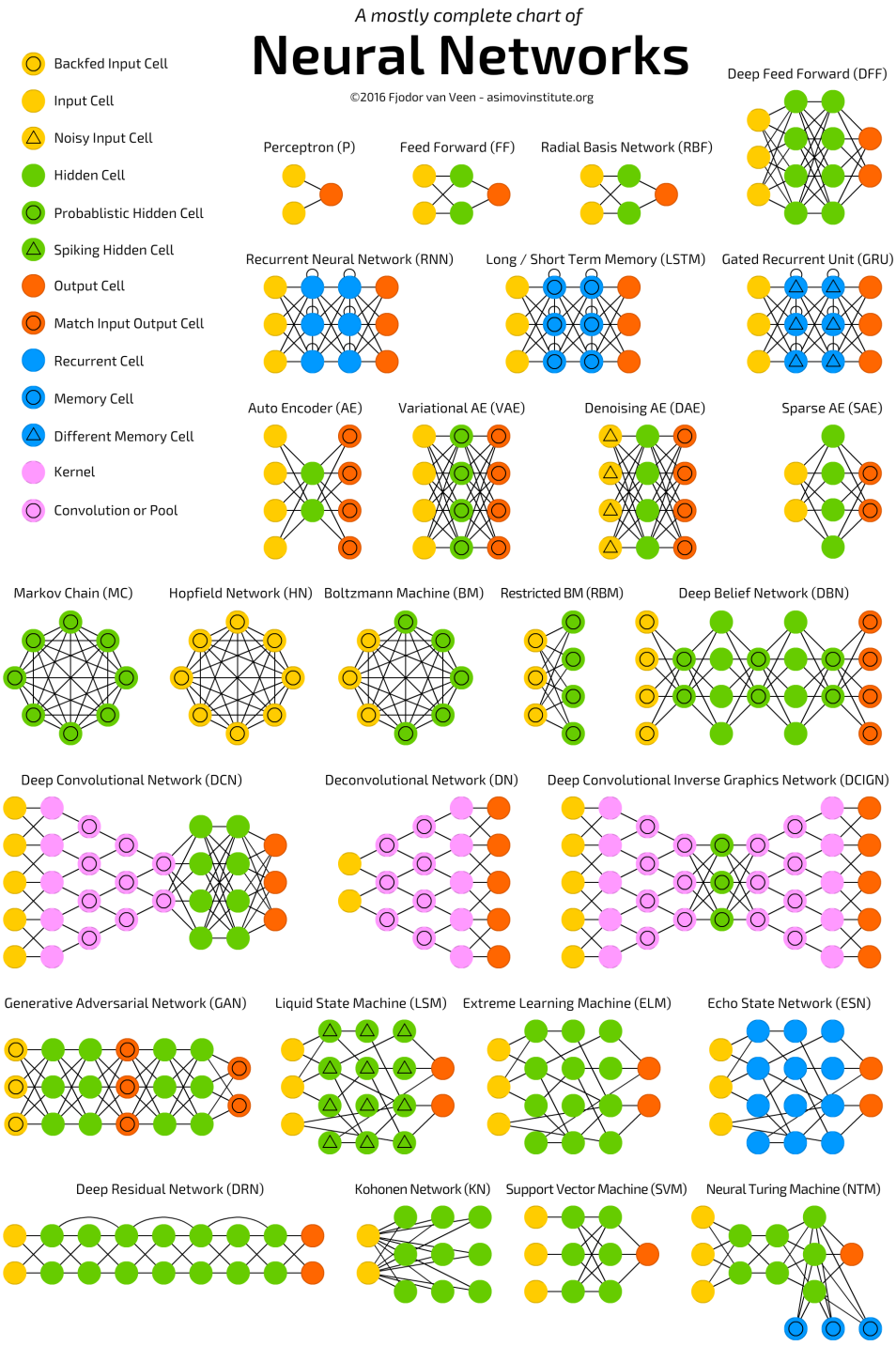
peak cached	**5475**
total feature map usage	**5219**
total layer weight usage	**102**
module.layer2.0.conv2.weight	1
module.layer2.0.downsample.0.weight	1
module.layer2.1.conv2.weight	1
module.layer2.2.conv2.weight	1
module.layer2.3.conv2.weight	1
module.layer3.0.conv1.weight	1
module.layer3.0.conv2.weight	2
module.layer3.0.conv3.weight	1
module.layer3.0.downsample.0.weight	2
module.layer3.1.conv1.weight	1
module.layer3.1.conv2.weight	2
module.layer3.1.conv3.weight	1
module.layer3.2.conv1.weight	1
module.layer3.2.conv2.weight	2
module.layer3.2.conv3.weight	1

module.layer3.3.conv1.weight	1
module.layer3.3.conv2.weight	2
module.layer3.3.conv3.weight	1
module.layer3.4.conv1.weight	1
module.layer3.4.conv2.weight	2
module.layer3.4.conv3.weight	1
module.layer3.5.conv1.weight	1
module.layer3.5.conv2.weight	2
module.layer3.5.conv3.weight	1
module.layer4.0.conv1.weight	2
module.layer4.0.conv2.weight	9
module.layer4.0.conv3.weight	4
module.layer4.0.downsample.0.weight	8
module.layer4.1.conv1.weight	4
module.layer4.1.conv2.weight	9
module.layer4.1.conv3.weight	4
module.layer4.2.conv1.weight	4
module.layer4.2.conv2.weight	9
module.layer4.2.conv3.weight	4
module.fc.weight	8
total layer weight gradient usage	**94**
module.layer2.0.conv2.weight_grad	1

module.layer2.0.downsample.0.weight_grad	1
module.layer2.1.conv2.weight_grad	1
module.layer2.2.conv2.weight_grad	1
module.layer2.3.conv2.weight_grad	1
module.layer3.0.conv1.weight_grad	1
module.layer3.0.conv2.weight_grad	2
module.layer3.0.conv3.weight_grad	1
module.layer3.0.downsample.0.weight_grad	2
module.layer3.1.conv1.weight_grad	1
module.layer3.1.conv2.weight_grad	2
module.layer3.1.conv3.weight_grad	1
module.layer3.2.conv1.weight_grad	1
module.layer3.2.conv2.weight_grad	2
module.layer3.2.conv3.weight_grad	1
module.layer3.3.conv1.weight_grad	1
module.layer3.3.conv2.weight_grad	2
module.layer3.3.conv3.weight_grad	1
module.layer3.4.conv1.weight_grad	1
module.layer3.4.conv2.weight_grad	2
module.layer3.4.conv3.weight_grad	1
module.layer3.5.conv1.weight_grad	1
module.layer3.5.conv2.weight_grad	2
module.layer3.5.conv3.weight_grad	1

module.layer4.0.conv1.weight_grad	2
module.layer4.0.conv2.weight_grad	9
module.layer4.0.conv3.weight_grad	4
module.layer4.0.downsample.0.weight_grad	8
module.layer4.1.conv1.weight_grad	4
module.layer4.1.conv2.weight_grad	9
module.layer4.1.conv3.weight_grad	4
module.layer4.2.conv1.weight_grad	4
module.layer4.2.conv2.weight_grad	9
module.layer4.2.conv3.weight_grad	4
intermediate gradients	26

Appendix B. Types of Neural Networks



A guide to common types of neural networks, created by van Vleen [38].

Appendix C. Types of Loss Functions

$\mathcal{L}_1$	L_1 loss	$\|\mathbf{y} - \mathbf{o}\|_1$
$\mathcal{L}_2$	L_2 loss	$\|\mathbf{y} - \mathbf{o}\|_2^2$
$\mathcal{L}_1 \circ \sigma$	expectation loss	$\|\mathbf{y} - \sigma(\mathbf{o})\|_1$
$\mathcal{L}_2 \circ \sigma$	regularised expectation loss[1]	$\|\mathbf{y} - \sigma(\mathbf{o})\|_2^2$
$\mathcal{L}_\infty \circ \sigma$	Chebyshev loss	$\max_j \lvert\sigma(\mathbf{o})^{(j)} - \mathbf{y}^{(j)}\rvert$
hinge	hinge [13] (margin) loss	$\sum_j \max(0, \frac{1}{2} - \hat{\mathbf{y}}^{(j)}\mathbf{o}^{(j)})$
hinge^2	squared hinge (margin) loss	$\sum_j \max(0, \frac{1}{2} - \hat{\mathbf{y}}^{(j)}\mathbf{o}^{(j)})^2$
hinge^3	cubed hinge (margin) loss	$\sum_j \max(0, \frac{1}{2} - \hat{\mathbf{y}}^{(j)}\mathbf{o}^{(j)})^3$
log	log (cross entropy) loss	$-\sum_j \mathbf{y}^{(j)} \log \sigma(\mathbf{o})^{(j)}$
$\log^2$	squared log loss	$-\sum_j [\mathbf{y}^{(j)} \log \sigma(\mathbf{o})^{(j)}]^2$
tan	Tanimoto loss	$\frac{-\sum_j \sigma(\mathbf{o})^{(j)}\mathbf{y}^{(j)}}{\|\sigma(\mathbf{o})\|_2^2 + \|\mathbf{y}\|_2^2 - \sum_j \sigma(\mathbf{o})^{(j)}\mathbf{y}^{(j)}}$
D_{CS}	Cauchy-Schwarz Divergence [3]	$-\log \frac{\sum_j \sigma(\mathbf{o})^{(j)}\mathbf{y}^{(j)}}{\|\sigma(\mathbf{o})\|_2 \|\mathbf{y}\|_2}$

A guide to loss functions, created by Janocha and Czarnecki [16].