Imperial College London

MENG INDIVIDUAL PROJECT

DEPARTMENT OF COMPUTING

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

Precise Optimal Checkpointing for Memory-Efficient Deep Learning

Author:
Shiraz Butt

Supervisor:
Prof. Peter Pietzuch

Septermber 10, 2019

Submitted to Imperial College London in partial fulfillment of the requirements for the degree of MEng Computer Science.

Abstract

Training a deep network is very memory intensive as the forward tensors must be kept alive for the backward pass. Checkpointing has been proposed to alleviate this, by dropping some forwards and recomputing them in the backwards pass. $\sqrt{N}$ checkpointing [1] is currently the most common technique. However, it is limited to only one recomputation, inhibiting the possible memory savings; assumes uniform per-layer compute and memory costs; and picks an arbitrary point on the compute-memory trade-off; that is, it minimises memory of an N layer network rather than minimising compute given M memory. A dynamic programming technique for RNNs has been proposed that achieves this [2]. However, it also assumes uniform costs as it is for RNNs, so fails to find the truly optimal checkpointing policy for general deep networks. Taking into account the precise per-layer costs would allow it to more judiciously determine the policy according to the exact nature of the layers, and thus truly minimise compute whilst satisfying the memory budget. Furthermore, to my knowledge, the latter technique is not imlpemented in any of the popular deep learning frameworks. Thus, in this thesis, I generalise the dynamic programming technique to use precise per-layer costs obtained from profiling, and implement this in PyTorch. To provide the groundwork for this, I meticulously derive the exact computational graph and memory requirements of one training step. I show that the precise solver finds better policies than what a uniform-cost solver could possibly find, leading to reduced overhead and the ability to satisfy lower memory budgets. However, I unfortunately could not remedy some implementation issues around imposing a policy on the execution of a network in PyTorch, so only have a simulator. I discuss how these issues arise. I also show that the ML community has rediscovered checkpointing from the Automatic Differentiation community, and that there is more work from them that we can leverage.

Contents

List of Figures

List of Listings

List of Algorithms

Chapter 1

Introduction

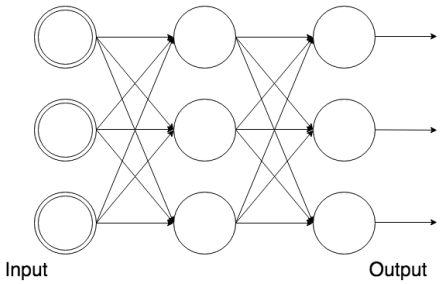
Deep Learning

Machine learning (ML) has become wildly popular among scientists, engineers, and even in the mainstream media, due to the unprecedented results it repeatedly provides in a multitude of areas. To name a few, we now have self-driving cars [9]; computers that can beat the best human players at complex games like StarCraft [10]; messaging apps can that understand what you're saying [11, 12]; many advances in computer vision such as automatic captioning [13]; automatic fraud detection in finance [14]; and automatic lesion detection in the fight against cancer [15].

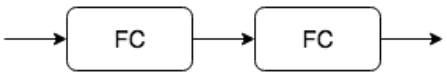
Artificial Neural Networks (ANNs), or Deep Learning, is the key technique behind many of these advances. They are a form of supervised statistical model. That is, given a set of $\{x \in \mathbb{R}^n,\ y \in \mathbb{R}^m\}$ labelled data points (e.g. the pixels of an image and the classification of the image), they try to learn the underlying function from input x to output y. ANNs use a particular kind of function representation made from the composition of elementary functions, or neurons, and try to incrementally optimise the parameters of the neurons by comparing the output of the network against the training data. The optimisation technique applied is almost always some variant of gradient descent, and the algorithm for applying it to an ANN is known as *backpropagation*. An example neural network is given in Figure 1.1.

The High Memory Requirement of Training

Training a neural network has a large memory requirement. To see how this arises, we can examine the computational (or data-flow) graph of backpropagation, as given in Figure 1.2. Backpropagation has two steps, (i) *the forward pass*: compute the outputs at each layer given the input, and (ii) *the backward pass*: compute the gradients of the parameters at each layer given the targets and the outputs from the forward pass. As



(a) At the individual neuron level.



(b) At the layer-wise level (FC means 'fully connected').

Figure 1.1: Example ANN. The double circles denote variables rather than functions. This network would be described as a feedforward/sequential/linear network of two fully-connected/dense layers. This is because there are two layers of neurons in sequence, with each layer feeding the outputs of the previous layer to all of its neurons.

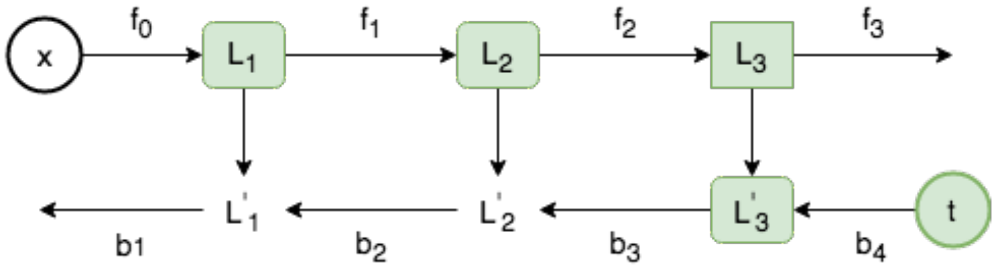


Figure 1.2: Demonstrating when peak memory occurs for the computational graph of backpropagation on a three-layer feedforward neural network. Layers in green have their output allocated in memory.

a result, the forward tensors cannot be computed in-place and must instead be kept around in memory for when the backward pass requires them. Specifically, only once b_i has been computed can f_i and b_{i+1} be freed. Therefore, as shown in Figure 1.2, peak memory will occur when all the forward tensors are in memory and the first backward tensor is being computed; a linear space complexity with respect to the number of layers.

This is not good enough for large networks - as most of the layers are performing operations like multiplication on large tensors, Deep Learning can be massively sped up by exploiting the parallelism of GPUs [16, 17, 18]; however, state-of-the-art models have far surpassed the memory available on even top-tier GPUs and continue to get larger [19, 20, 21].

This forces researchers to employ highly cumbersome and highly expensive alternatives, such as distributing the workload or developing custom hardware accelerators. Ideally, as the networks of tomorrow grow unabatedly, researchers should be able to focus on their experiments, rather than the complex systems problems this causes. Thus, finding a solution that can be readily implemented in existing ML software and that is transparent to the user is highly desirable, as well as orthogonal to the other approaches (which have other benefits too).

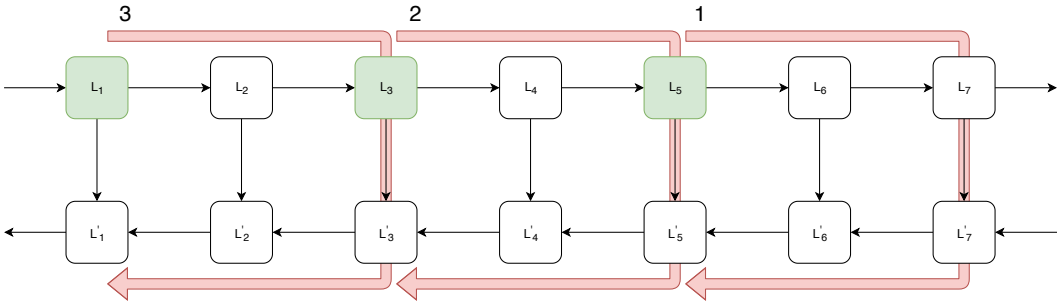


Figure 1.3: Segmented backpropagation with checkpointing: Say we are computing the backward pass. The forward pass has already been computed, and only the outputs of L_1, L_3, and L_5 were stored. Then, to perform the backward pass, we perform backproagation (both forward and backward) for segments 1, then 2, then 3, as shown.

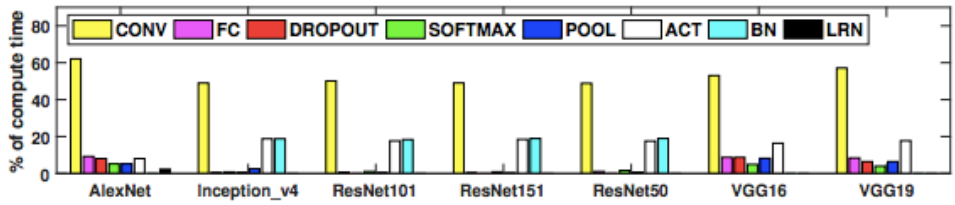
To this end, much has been done, but every memory optimisation proposed has its pitfalls. For example, quantization decreases the model's accuracy [22]; and GPU-CPU data swapping may require expert human intervention, is difficult to tractably solve for the optimal swapping strategy, and is hard to implement as you must delve deep into the ML framework's memory manager and have it precisely pipeline data transfer and computation [7, 23, 24, 3].

Checkpointing and its Limitations

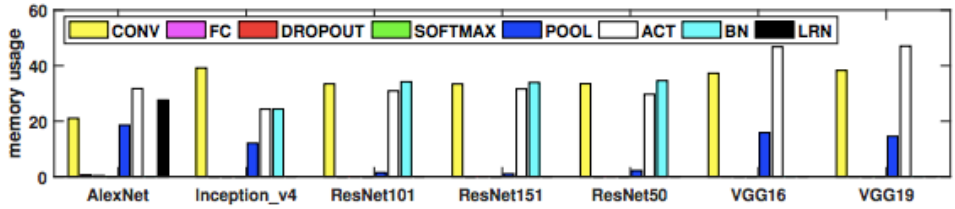
In this thesis, I focus on a technique known as checkpointing [25, 26, 27, 1, 2, 3], which results in large memory savings, is much simpler to implement, does not affect model accuracy and requires minimal effort from the user; though it is hard to solve tractably and, compared to a theoretically optimal swapping implementation, the computational overhead may be poor, though still very acceptable.

Checkpointing trades computation for memory by only storing, or *checkpointing*, some of the intermediate forwards and recomputing them later when required in the backwards pass. This causes backpropagation to become segmented, demonstrated in Figure 1.3. Martens and Sutskever [26], and later Chen [1], have shown that for k segments and n layers setting $k = \sqrt{n}$ yields a sublinear memory cost of $\Theta(2\sqrt{n})$, at the expense of an additional forward pass. Pages 7-8 of Chen's paper proposes recursively applying this technique to the segments to further trade compute for memory: during the recomputation of a segment, only checkpoint some tensors and recompute the rest in the backward, resulting in multiple recomputations. These results were actually first discovered by Griewank in 1992 [28] in the more general context of Automatic Differentiation, and have now been rediscovered in the ML community.

The question now becomes of solving for the optimal checkpointing policy: which tensors to checkpoint, how many to checkpoint, and how best to exploit multiple recomputations, such as to satisfy the memory budget with the least computational



(a) breakdown of execution time by layer types



(b) breakdown of memory usages by layer types

Figure 1.4: Breakdown of compute and memory costs by layer type in a number of popular neural networks [3, Figure 8].

cost. The solution depends on the precise memory budget and per-layer compute and memory costs. Gruslys et al. form DeepMind solve this for Recurrent Neural Networks (RNNs) using dynamic programming [2]. However, RNNs comprise a sequence of solely the same layer repeated many times, giving uniform per-layer compute and memory costs - a very simplifying assumption that does not hold for almost all neural networks. Thus, when the technique is applied to general networks, the lack of information about the precise per-layer costs causes the policy solver to make suboptimal decisions.

For example, consider Figure 1.4, which shows just how *not* uniform the layers are in some of the most common network architectures. Convolutional layers are shown to be at times more compute intensive than any of the other layers. However, a uniform cost policy solver does not know this, so will always choose to recompute a convolutional layer once rather than recomputing another layer twice.

In addition to the above limitation of checkpointing, popular ML frameworks like TensorFlow [29], MxNet [30], and PyTorch [31] have very limited support for checkpointing. PyTorch, MxNet, and a third-party TensorFlow library by OpenAI [32] have similar functionality. They force the user to specify the number of segments and will split the sequence evenly, or ask the user to choose the checkpoints. They do not support multiple recomputations. TensorFlow's internal graph optimiser Grappler employs a non-optimal, user-guided, greedy, heuristic-based approach that finds the backward nodes in the computational graph, selects their inputs and the user-specified inputs as candidates for dropping, then does so until the memory budget is satisfied. Moreover, Grappler, in my opinion, requires non-trivial effort on the user's behalf to manually configure.

Clearly, the checkpointing support in these frameworks is behind research. I have chosen to improve PyTorch as it is one of the most popular frameworks, and because it already has a simple, clean implementation of performing a single recomputation.

Contributions

Specifically, to improve upon the above discussed limitations of checkpointing, I will make the following contributions in this thesis:

- Derive the exact computational graph of backpropagation and precisely describe its memory requirements so that precise checkpointing can be applied to it.
- Generalise the dynamic programming technique proposed by DeepMind to solve for the optimal checkpointing policy of a feedforward neural network given *arbitrary* per-layer compute and memory costs and the memory budget.
- Provide an implementation of this technique in PyTorch that allows the user to overcome out-of-memory errors with minimal effort. Given only the user's existing sequential network, it will:
  1. Profile the precise per-layer costs;
  2. Solve for the optimal checkpointing policy;
  3. Present a helper function that can be called in the user's training loop to execute the sequence according to the policy, satisfying the memory budget.
     - Unfortunately, I have so far only been able to implement a simulator.

Chapter 2

Background

In this section, I present the relevant background material and motivate my thesis. First, Deep Learning and backpropagation are introduced insofar as deriving its computational graph, which the memory optimisations are based on. I then show how backpropagation is a specific instance of *reverse-mode automatic differentiation* [6]. I introduce PyTorch, the ML framework I will use, and show how its Autograd module performs reverse-mode automatic differentiation. This will be relevant to how my implementation works. I will next introduce liveness analysis as a technique for reducing the memory cost of a computational graph; formulate the exact computational graph that will be used to describe precise checkpointing; and analyse the precise memory requirements of that graph when liveness analysis is applied. Finally, I introduce the checkpointing memory optimisation and appraise the current approaches, motivating my contributions.

2.1 Deep Learning

2.1.1 Supervised Learning with Gradient Descent

Machine learning is about solving problems we do not know how to solve directly. Rather, we perform some kind of statistical inference. In supervised machine learning, we are trying to learn a mapping from inputs to outputs, for example between the pixels of an image and the classification of that image. Neural networks are a type of mapping known as a *parametric* model. I will outline here the general procedure for training such a model before detailing the specifics of neural networks themselves.

To first step to learning this mapping is to obtain some $\{x_i \in \mathbb{R}^n,\ y_i \in \mathbb{R}^m\}_{i=1}^N$ data points to learn from, called the training data. Then, we pick some functional representation, for example $y = xw + b$, where $w \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m$; meaning we approximate the true mapping as a linear function. We will attempt to learn from the data the parameters w, b that best model the training data. Note, θ is often used as a single

variable to denote all the parameters. We will call our linear function given those parameters $f_\theta : R^n \to R^m$. It is often referred to as the hypothesis, model, or the network in the case of neural networks. The space of all possible θ is known as the hypothesis space, weight space, or parameter space.

To conduct learning, we need a performance metric for our function and a method to optimise the parameters with respect to that metric. The performance metric is often referred to as a loss function, cost function, or error function; and evaluates how 'good' the parameters are using the training data. We denote this loss function, $L_\mathrm{D} : \Theta \to \mathbb{R}^+$, where Θ is the set of all possible parameters and D is the training data. Typically, the loss averages over data points as follows:

$$L_\mathrm{D}(\theta) = \frac{1}{N} \sum_{i=1}^{N} l(f_\theta(x_i), y_i)$$

where $l : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^+$ measures the 'loss' between the training data outputs, known as the targets, and what f_θ produced, for that specific data point. This could be the square-error loss $l(o, t) = (o - t)^2$, for example.

Thus, the original problem has been reduced into an optimisation problem. Specifically, the objective is to minimise L_D with respect to θ.

We can do this using gradient descent [33]. This is an iterative method that follows the line of steepest descent (the gradient) down the slope in small increments, often likened to placing a ball on the loss surface and watching it roll down to the minimum point. Formally, the procedure is to initialise our parameters to some θ_0 before iteratively applying the following update rule until we are satisfied with our parameters:

$$\theta_{k+1} = \theta_k + \eta((\nabla L_\mathrm{D})(\theta_k))^\top$$

Where ∇L_D is the Jacobian matrix of the loss with respect to the parameters, and $\eta \in \mathbb{R}$ is a small, positive number known as the learning rate. Given sufficiently small η, the sequence $L_\mathrm{D}(\theta_0) \geq L_\mathrm{D}(\theta_1) \geq L_\mathrm{D}(\theta_2) \geq \ldots$ must converge to a local minimum of L_D.

2.1.2 Mini-Batch Stochastic Gradient Descent

Attempting to use gradient descent exactly as described above leads to some issues in the real-world. Firstly, every single weight update step requires iterating over the entire dataset to evaluate the loss. Real world datasets are very large, for example the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) Object Localisation dataset has over 1.2 million images [34]. Iterating over this entire dataset for a single weight update would be prohibitively slow. Secondly, recall gradient descent is only

guaranteed to converge to the *local* minima. For our simple example, the loss function is convex, so this is not a problem. However, to model complex real world problems, we need more expressive functional representations that can model highly complex non-linearities, leading to a non-convex loss. As an example, the loss surface of the VGG-56 [1] network [35] is visualised in Figure 2.1. Note it has many narrow local minima for gradient descent to get stuck in.

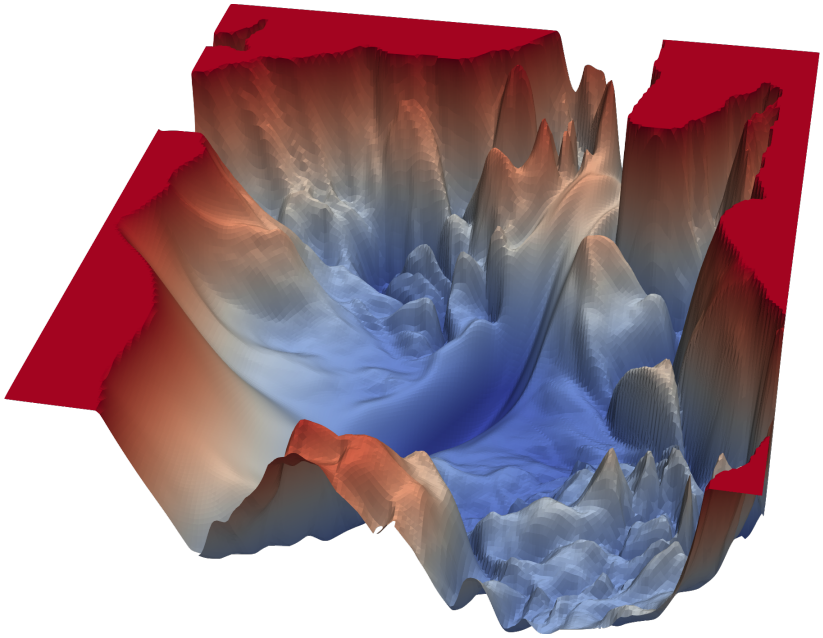


Figure 2.1: Visualisation of the loss function of VGG-56 [4].

To work around these problems, stochastic gradient descent (SGD) is used [36]. Rather than updating the parameters by computing the loss over the entire dataset, only a small 'mini-batch' is used for one update step. This approximates true gradient descent and takes more steps to converge, but each step is now computationally tractable. Moreover, it means that on every update we are actually descending a different loss curve, L_B, defined by the mini-batch $B \subseteq D$ we are working on, as opposed to always using the same loss curve L_D that sums over all of the data. To see why this is beneficial, consider when after an update we have reached the local minima for that loss, but not the true global minima. On the next update, we are descending a different loss curve that likely does not have that exact local minima too, so we can fall through it and continue towards the global minima.

A Note on Stochastic Gradient Descent in Practice

I have outlined the SGD process used to train models in machine learning, which will be applied to neural networks in the next section and made memory-efficient in the

[1] VGGNets are a popular neural network architecture for computer vision tasks. A VGGNet won the object localisation task of the ILSVRC in 2014.

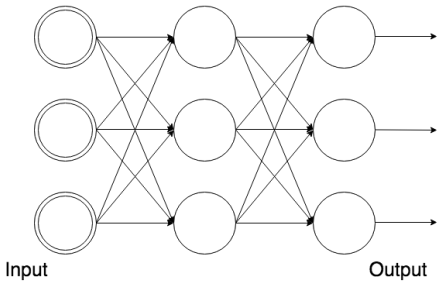
remainder of this thesis.

It should be noted, however, that even SGD is not actually good enough by itself. Much research goes into improving the optimisation methods of machine learning. Commonly used techniques include dropout [37], L1/L2 regularisation [38, 39], momentum [40], and Adam Optimisation [41].

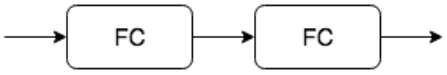
None of these methods significantly affect the nature of the memory optimisations presented later, so I will not cover them here.

2.1.3 Artificial Neural Networks

The problems being solved today, such as computer vision tasks, are extremely difficult. A simple linear function will not be able to sufficiently express the intricacies of the real underlying mapping being approximated. We need a model that (i) can express complex non-linearities, (ii) has a continuous hypothesis space, and (iii) the hypothesis space is differentiable with respect to its parameters. Neural networks satisfy these conditions and have been found to work very well. They are the composition of many elementary non-linear operators, called *neurons*. An example of such a network is given in Figure 2.2, repeated from the introduction. This network would be



(a) At the individual neuron level.



(b) At the layer-wise level (FC means 'fully connected').

Figure 2.2: Example ANN. The double circles denote variables rather than functions.

described as a feedforward/sequential/linear network of two fully-connected/dense layers. This is because there are two layers of neurons in sequence, with each layer feeding the outputs of the previous layer to all of its neurons.

The function performed by an individual neuron is normally of the form:

$$y = \sigma(w^\top x + b)$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is called the *activation function* or a *threshold* and applies a non-linearity to the affine transform inside. Examples include the Rectified Linear Unit (ReLU), $\sigma(y) = \max(0, y)$; and the sigmoid/logistic function, $\sigma(y) = 1/(1 + e^{-y})$.

This is all it takes to describe a neural network, but they are incredibly powerful - the Universal Approximation Theorom [42, 43][44, p. 105] states that any bounded

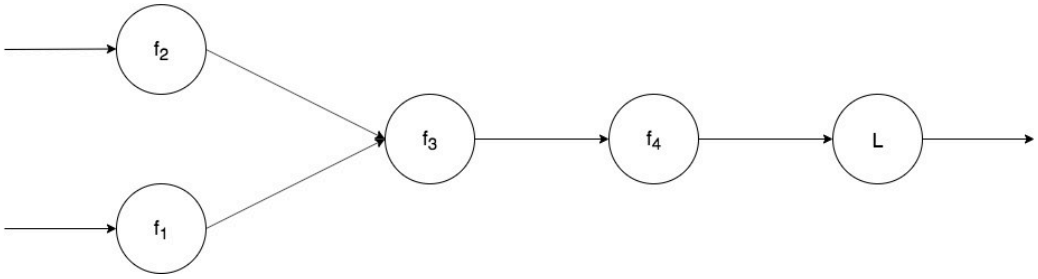


Figure 2.3: An example network.

continuous function can be approximated with arbitrarily small error (under a finite norm) by a network of just *two* layers [2].

2.1.4 Backpropagation

How, then, do we train such a network? To recap, we must minimise the loss with respect to the parameters of the model using gradient descent, which requires finding the partial derivatives of the loss with respect to each parameter. This was a trivial matter for a simple linear function, but, for a neural network, we need to get the partial derivatives for the parameters *at each neuron*. This is easy too; as the neurons are just functions composed together, we can use the chain rule to find the derivatives for the parameters at each neuron.

Consider the example network in Figure 2.3. Then, we use the recursive formulas to obtain the desired derivatives:

$$\begin{aligned}
\frac{\partial L}{\partial \theta_4} &= \frac{\partial L}{\partial f_4}\frac{\partial f_4}{\partial \theta_4} && = \delta_4 \frac{\partial f_4}{\partial \theta_4} \\
\frac{\partial L}{\partial \theta_3} &= \frac{\partial L}{\partial f_4}\frac{\partial f_4}{\partial f_3}\frac{\partial f_3}{\partial \theta_3} && = \delta_4 \frac{\partial f_4}{\partial f_3}\frac{\partial f_3}{\partial \theta_3} && = \delta_3 \frac{\partial f_3}{\partial \theta_3} \\
\frac{\partial L}{\partial \theta_2} &= \frac{\partial L}{\partial f_4}\frac{\partial f_4}{\partial f_3}\frac{\partial f_3}{\partial f_2}\frac{\partial f_2}{\partial \theta_2} && = \delta_3 \frac{\partial f_3}{\partial f_2}\frac{\partial f_2}{\partial \theta_2} && = \delta_2 \frac{\partial f_2}{\partial \theta_2} \\
\frac{\partial L}{\partial \theta_1} &= \frac{\partial L}{\partial f_4}\frac{\partial f_4}{\partial f_3}\frac{\partial f_3}{\partial f_1}\frac{\partial f_1}{\partial \theta_1} && = \delta_3 \frac{\partial f_3}{\partial f_1}\frac{\partial f_1}{\partial \theta_1} && = \delta_1 \frac{\partial f_1}{\partial \theta_1}
\end{aligned}$$

where

$$\delta_l = \frac{\partial L}{\partial f_l}$$

[2] This does not say anything about the *learnability* of the parameters of such networks though.

That is, we start at the end of the network and go backwards: we use δ_{l+1} to work out δ_l and $\frac{\partial L}{\partial \theta_l}$; then use δ_l to work out δ_{l-1} and $\frac{\partial L}{\partial \theta_{l-1}}$; and so on. Thus, the *deltas* are being propagated backwards, and so the process is known as *backpropagation*. Figure 2.4 shows the computational graph for this. The 'forward graph' is shown with the corresponding 'backward graph' for computing the derivatives using the chain rule.

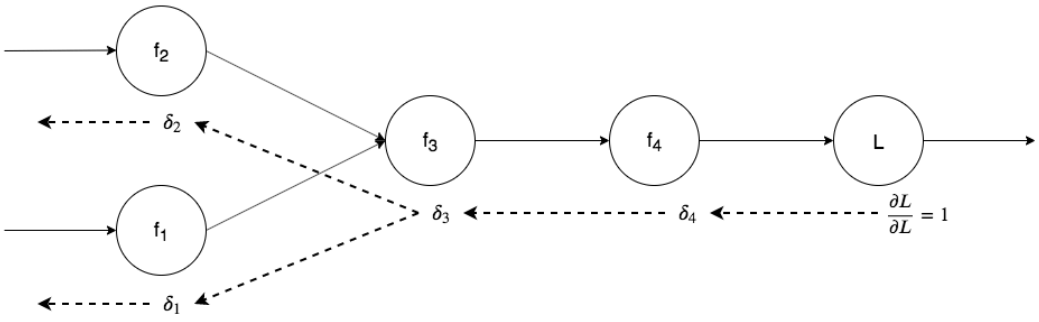


Figure 2.4: Backpropagation on a simple network.

2.1.5 The Computational Graph for Training a Network

Figure 2.4 merely shows how to backpropagate the deltas. It does not fully describe one step of training a neural network, for example it does not show the weight updates.

To derive the full graph, we must first fully specify the update rules for backpropagation and gradient descent. I have borrowed from Erik Hallström's derivation of the update rules, which can be found online [45].

Consider an N layer network with any number of neurons per layer. Let $\boldsymbol{y}^l$ be the activations of the neurons in layer l. Let $\boldsymbol{W}^l$ be the weights from layer $l-1$ to l, where w^l_{kj} is the connection from the jth neuron in layer $l-1$ to the kth neuron in layer l. Let $\boldsymbol{b}^l$ be the bias for each neuron in layer l, where b^l_k is the bias added to the kth neuron in layer l. The layer l is computed,

$$\boldsymbol{y}^l = \sigma(\boldsymbol{W}^l \boldsymbol{y}^{l-1} + \boldsymbol{b}^l) \tag{2.1}$$
$$= \sigma(\boldsymbol{z}^l), \tag{2.2}$$

where we have defined $\boldsymbol{z}^l = \boldsymbol{W}^l \boldsymbol{y}^{l-1} + \boldsymbol{b}^l$, or, the *input sum* for layer l.

We compute the input sum of a neuron m in layer $l+1$ computed from the activations of the layer $l-1$ as follows,

$$z_k^l = \sum_j w_{kj}^l \cdot y_j^{l-1} + b_k^l, \tag{2.3}$$

$$y_k^l = \sigma(z_k^l), \tag{2.4}$$

$$z_m^{l+1} = \sum_k w_{mk}^{l+1} \cdot y_k^l + b_m^l. \tag{2.5}$$

We wish to compute the derivative of the loss L with respect to each weight w_{kj}^l. Using the above equations and the chain rule, we write this as,

$$\frac{\partial L}{\partial w_{kj}^l} = \frac{\partial L}{\partial z_k^l}\frac{\partial z_k^l}{\partial w_{kj}^l} \tag{2.6}$$

$$= \frac{\partial L}{\partial y_k^l}\frac{\partial y_k^l}{\partial z_k^l}\frac{\partial z_k^l}{\partial w_{kj}^l} \tag{2.7}$$

$$= \left(\sum_m \frac{\partial L}{\partial z_m^{l+1}}\frac{\partial z_m^{l+1}}{\partial y_k^l}\right)\frac{\partial y_k^l}{\partial z_k^l}\frac{\partial z_k^l}{\partial w_{kj}^l}. \tag{2.8}$$

Substituting in the derivatives of Equations 2.3-2.5, we arrive at,

$$\left(\sum_m \frac{\partial L}{\partial z_m^{l+1}} w_{mk}^{l+1}\right) \cdot \sigma'(z_k^l) \cdot y_j^{l-1}. \tag{2.9}$$

Next, define the deltas δ_k^l for a specific neuron,

$$\delta_k^l = \frac{\partial L}{\partial z_k^l}. \tag{2.10}$$

Note that in contrast with the previous section, here we use notation δ_k^l, where the layer l now appears in superscript, and the subscript k refers to a specific neuron in that layer. Furthermore, the delta refers to the derivative of the loss L with respect to the input sum (unthresholded output) of a layer.

We can expand Equation 2.10 leading to a recursive formula for computing the delta for every weight,

$$\delta_k^l = \frac{\partial L}{\partial z_k^l} = \left(\sum_m \frac{\partial L}{\partial z_m^{l+1}} w_{mk}^{l+1}\right) \cdot \sigma'(z_k^l) \tag{2.11}$$

$$= \left(\delta_m^{l+1} w_{mk}^{l+1}\right) \cdot \sigma'(z_k^l). \tag{2.12}$$

The gradient of the loss with respect to biases can also be written,

$$\frac{\partial L}{\partial b_k^l} = \frac{\partial L}{\partial z_k^l}\frac{\partial z_k^l}{\partial b_k^l} = \frac{\partial L}{\partial z_k^l} = \delta_k^l, \tag{2.13}$$

where we used that $\frac{\partial z_k^l}{\partial b_k^l} = 1$, which can be seen easily from Equation 2.3. We now have recursive formulas for the deltas.

To compute the gradient of the loss with respect to the weights and biases, we first compute δ_j^N,

$$\delta_j^N = \frac{\partial L}{\partial z_j^N} = \frac{\partial L}{\partial y_j^N}\frac{\partial y_j^N}{\partial z_j^N} = \sigma'(z_j^N), \tag{2.14}$$

for all j neurons in the final layer. Then, we can apply the recursive formulas to compute δ_k^{N-1} for all k layers in the $(N-1)$-th layer, and so on. Note we need to know $\frac{\partial L}{\partial a_j^N}$, the derivative of the loss with respect to the final outputs, which can be found analytically (else it cannot be used).

Rather than working with individual neurons, we can generalise the equations to deal with delta vectors δ^l and weight matrices $\boldsymbol{W}^l$ for an entire layer. This gives the following recursive backpropagation formulae,

$$\delta^N = \nabla_{\boldsymbol{y}^N} L \odot \sigma'(\boldsymbol{z}^l), \tag{2.15}$$

$$\delta^l = (\boldsymbol{W}^{l+1})^\top \delta^{l+1} \odot \sigma'(\boldsymbol{z}^l), \tag{2.16}$$

$$\frac{\partial L}{\partial b_j^l} = \delta_j^l, \tag{2.17}$$

$$\frac{\partial L}{\partial w_{kj}^l} = \delta_k^l y_j^{l-1}, \tag{2.18}$$

where $\odot$ denotes Hadamard (element-wise) product.

From this, the *backpropagation algorithm* for training a neural network emerges. It can be seen from the rules that, in order to perform one step of gradient descent, we need to evaluate the network to get the intermediate forwards, then plug them in to the above rules.

By further inspection, we can derive the full computational graph for this algorithm. For simplicity, collect the weights and biases for a layer l into an object $\boldsymbol{\theta}_l$. To find the weight update for $\boldsymbol{\theta}_l$, we must:

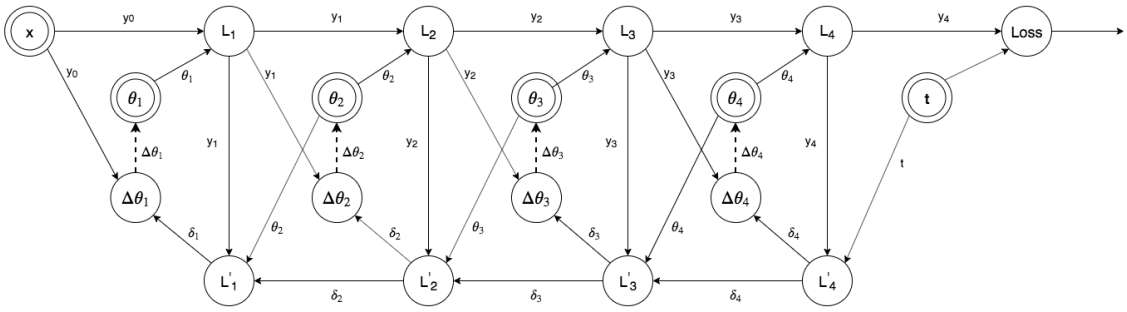


Figure 2.5: The computational graph for one step of gradient descent on a neural network of four layers.

1. Calculate δ^l using the layer's output $\boldsymbol{y}^l$, the upstream deltas δ^{l+1} and the upstream weights $\boldsymbol{\theta}_l$.
2. Calculate the parameter update $\Delta\boldsymbol{\theta}_l$ using the layer's inputs $\boldsymbol{y}^{l-1}$ and δ^l from the previous step.

Also, as the forward passes for each input in a batch are independent, we collate the inputs into one tensor and perform the forward pass at once element-wise. Similarly, once the loss has been computed, we collate and backpropagate the deltas. At each layer, we sum over all the derivatives from the batch to compute one weight update. However, the weight updates cannot be performed immediately in-place, as the old weights are required to compute the next layer's delta, so control dependencies are required.

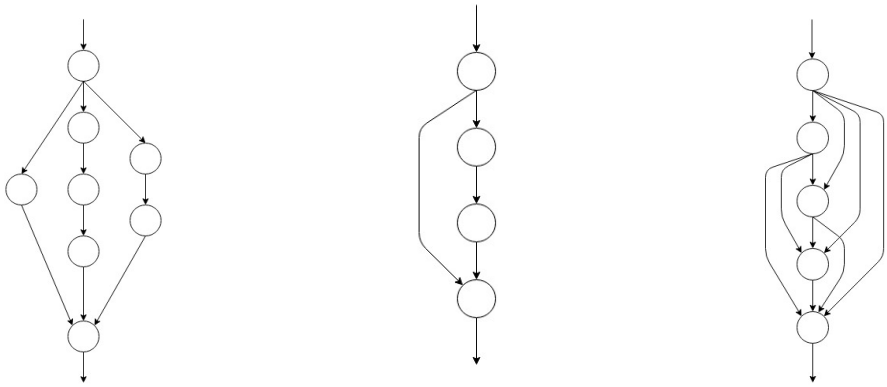
Finally, we arrive at the computational graph given in Figure 2.5. I omit the control dependencies for clarity. The nodes are operators and the arcs are the data they output. The double-circled nodes are nullary operators representing variables. The dashed arcs represent updates to variables.

This graph underlies the memory optimisations presented in this thesis. In Section 2.4, I will analyse the graph to determine how much memory is required to perform one step of training, then in Section 2.5 show how to reduce this requirement using checkpointing.

2.1.6 Beyond Feedforwad Networks

In recent years, thanks to a number of breakthrough papers, certain non-linear architectures have become extremely popular, especially in vision-related tasks. In particular, Inception modules use a split-merge architecture [20, 46]; and residual blocks [21, 47, 46] and dense blocks [19] use skip connections. These non-linearities are visualised in Figure 2.6.

Recurrent Neural Networks (RNNs) also see wide use. RNNs are networks whose output feed back into their input. The part of the output that that feeds back to the



(a) A split-merge unit [20, 46]. **(b)** A residual unit [21, 47, 46]. **(c)** A dense unit [19].

Figure 2.6: Common non-linearities in neural networks.

cell is known as the *hidden state*. The cell can then be unrolled over many time steps, creating a feedforward network of RNN cells, where each cell propagates a hidden state to the next. Such networks are trained through the Backpropagation Through Time (BPTT) algorithm [48, 49, 50]. Unfolding this for t time steps can be visualised as in Figure 2.7. Note that the same cell f_W with the same parameters W are used at each time step.

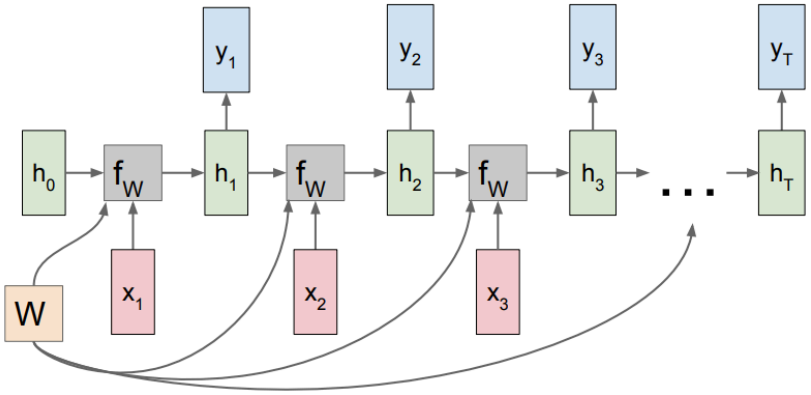


Figure 2.7: An RNN cell unrolled for T time steps [5, p. 29].

In this example, we are using a `seq2seq` model. These map a variable-sized input sequence to a variable-sized output sequence. An example use case is natural language translation [51], where the input and output are variable-sized phrases.

2.2 Automatic Differentiation

Evidently, training a neural network involves taking derivatives of a computation - applying the network to a batch of data to get the loss - but how do we actually

evaluate derivatives on a computer? There are three main approaches: numerical differentiation, symbolic differentiation, and automatic differentiation (AD). In this section, I will show how automatic differentiation is used in machine learning.

Baydin et al. [6] have an excellent survey paper on the intersection of AD and ML; covering in more depth the pitfalls of the other techniques, both forward and reverse mode AD, how to implement AD, and the varying adoption of AD in ML libraries. I will merely give a brief motivation of AD over the other techniques and show how backpropagation is an instance of *reverse mode* AD. I will borrow heavily from their paper.

2.2.1 Motivation and Overview

Numerical differentiation refers to finite-difference methods, which evaluate the original function at two sample points and plug that into the limit definition of a derivative. This can be highly inaccurate due to round-off and truncation errors [52]. It also requires sampling the function for each partial derivative required, so does not scale to the millions of parameters in a neural network.

Symbolic differentiation refers to the mechanistic application of symbolic differentiation rules to a symbolic expression tree. This is slow and suffers from 'expression swell' [53] - the rote application of rules causing the size of the derivative expression to become large and complex.

Moreover, both these techniques require the model to be defined in a closed-form expression. This is extremely difficult for neural networks, especially in the case where the model contains control flow, common in RNNs for example.

Automatic differentiation, also known as algorithmic differentiation or just autodiff, is a powerful family of techniques that gives evaluated derivatives, not a symbolic expression, by using pre-programmed derivatives of the individual operators applied during a computation, and the chain rule to propagate derivatives between them to get the derivative of the entire computation. As the user code applies operators, the AD library will transparently be augmenting the values to store the derivatives too, and augmenting the operators to calculate their derivatives and propagate them to the next operators. That is, both forward and backward functions of operators are implemented in the AD library manually. The user runs a computation using the forward functions as usual, and the derivatives will automatically be evaluated by the AD library using the backwards functions. The chain rule is used to compute the derivatives of a composition of operators with respect to the data at an individual operator.

This allows the evaluation of derivatives at machine precision with only a small constant factor of overhead. Furthermore, the user can take derivatives of arbitrary, imperatively defined computations, including the use of control flow, with zero extra effort; rather than deriving complicated closed-form expressions. This flexibility has

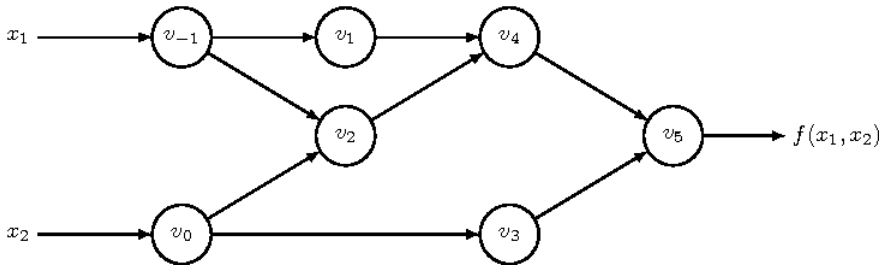


Figure 2.8: Computational graph of the example $f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$. See Figure 2.8 for the definitions of the intermediate variables $v_{-1} \dots v5$. [6, Figure 3]

made AD a popular tool in many fields, from machine learning to structural mechanics [54].

AD should sound a lot like backpropagation. We run a series of forward operations, and need to compute derivative of the output of the entire computation with respect to the data at each operator. We know the derivatives functions of each operator, and take the overall derivative by using the chain rule. Backpropagation, then, is a specific instance of AD.

Specifically, backpropagation works best with what is known as *reverse mode* AD, as opposed to *forward mode*. Though I will not cover the latter here, suffice it to say that forward mode allows you to take the derivative of any number of outputs with respect to a single input in only one pass. However, it requires a separate pass for each input. Reverse mode is the other way around - it can take the derivative of a single output with respect to any number of inputs in one pass. This is what happens in backpropagation - we take the derivative of the loss with respect to a large number of parameters. Thus, backpropagation is best implemented using reverse mode.

2.2.2 Reverse Mode AD

In reverse mode automatic differentiation, a forward and backward pass occurs. In the forward pass, as each operator is applied, the reverse dependencies are recorded. Then, in the backward pass, the derivatives of the final output with respect to each operator output are calculated by propagating derivatives back from the output to the input using the chain rule.

In the AD literature, the forward outputs are known as the primals, and the backwards as the adjoints. A specific forward pass on some data is a primal trace, and the backward an adjoint trace.

Consider $f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$. The computational graph for this is given in Figure 2.8. The v_i denote the intermediate outputs (primals). As the forward pass occurs, the primals are stored for use in the backward pass, and the reverse dependencies are stored. The backward pass is initialised by setting the partial derivative of the output with itself to 1. Then, we can work backwards using the chain rule. For

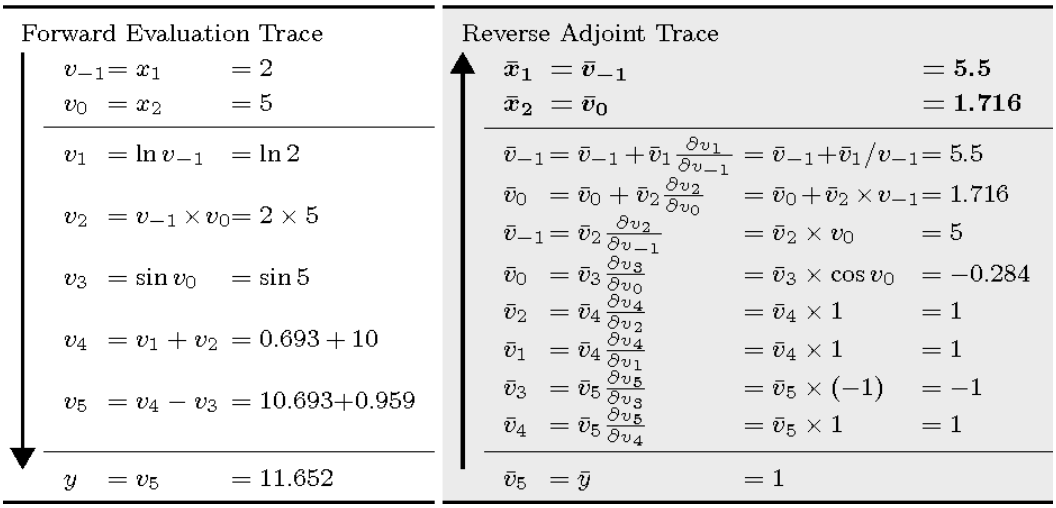


Figure 2.9: Reverse mode AD example, with $f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ at $x_1, x_2 = (2, 5)$. After running the original forward run on the left, the augmented AD operations on the right are run in reverse. [6, Table 3]

example, the subset of the trace computing the derivative for v_0 will use the following rules:

$$
\begin{aligned}
\bar{v}_5 &= \frac{\partial v_5}{\partial v_5} && = 1 \\
\bar{v}_4 &= \frac{\partial v_5}{\partial v_4} && = \bar{v}_5 \frac{\partial v_5}{\partial v_4} \\
\bar{v}_3 &= \frac{\partial v_5}{\partial v_3} && = \bar{v}_5 \frac{\partial v_5}{\partial v_3} \\
\bar{v}_2 &= \frac{\partial v_5}{\partial v_2} && = \bar{v}_4 \frac{\partial v_4}{\partial v_2} \\
\bar{v}_0 &= \frac{\partial v_5}{\partial v_0} && = \bar{v}_2 \frac{\partial v_2}{\partial v_0} + \bar{v}_3 \frac{\partial v_3}{\partial v_0}
\end{aligned}
$$

An example primal and adjoint trace is given in Figure 2.9.

This shows how backpropagation is a specific instance of the more generic reverse mode AD. It also shows how, in a single pass, the derivative of a single output with respect to *every* intermediate variable can be calculated.

Note that the intermediate outputs were all stored during the forward pass, so they could be reused in the backward pass. This linear space complexity (with respect to the number of operators) causes the high peak memory requirement of backpropagation. This will be analysed in more depth in Section 2.4.

2.3 PyTorch

2.3.1 Overview

PyTorch [31] is a popular ML framework with a Python frontend (as well as others). Its key features include:

- Easily execute tensor computations in Python, with an API similar to the Numpy library [55].
- Easily transfer between PyTorch tensors and Numpy nd-arrays.
- Easily accelerate these computations on a GPU.
- Easily evaluate derivatives of the computation with respect to the tensors involved. The user simply tells PyTorch to track the gradients when constructing a tensor.
- *Define-by-run* execution: The user does not need to statically define a computation graph like in TensorFlow [29], even when taking derivatives. They can execute any computation using native Python constructs such as control flow.
- Provides a rich API for constructing common ML operators, such as convolutions or ReLU
- Allows the user to extend the API with their own operators. Can easily interleave pre-defined and custom operators in one computation.
- Provides distribution strategies for further scaling out ML training and inference.

An example PyTorch program that demonstrates some of these features is given in Listing 2.1.

```python
import numpy as np
import torch

# Numpy like API with derivatives and GPU acceleration.
a = torch.Tensor([[1., 2.], [-1., 3.]], device='cuda')
a.requires_grad = True

b = torch.randn(2, 2, requires_grad=True).cuda()

c = a + b
d = torch.full_like(a, 15., requires_grad=True)

# From numpy.
e_np = np.ones((2, 2), dtype=np.single)
e = torch.from_numpy(e_np).to('cuda')

# Broadcasting.
f = ((e + 2) ** 1.3) * d

# ML functions.
g = torch.nn.functional.relu(c)
h = torch.nn.functional.linear(f, g).sum()

# Perform backward pass.
h.backward()

a.grad
  # ----> The gradient tensor.
b.grad
  # ----> The gradient tensor.
c.grad
  # ----> None, c is not a leaf.
d.grad
  # ----> The gradient tensor.
e.grad
  # ----> None, did not require gradient.
```

Listing 2.1: Example PyTorch program

2.3.2 Autograd

PyTorch's Autograd module is responsible for the evaluation of derivatives of dynamically defined computations. It uses true reverse-mode automatic differentation. That

is, the user runs their forward computation normally and it is automatically augmented with a backwards graph. As stated above, this means there is no need to define a static graph, and native Python control flow can be used.

Consider, in Figure 2.10, some PyTorch code and the backward graph generated by Autograd. It is very important to point out that this graph, nor the following explanation of how it is constructed, do not exactly correspond to how Autograd is implemented; only to help convey the ideas at a high level.

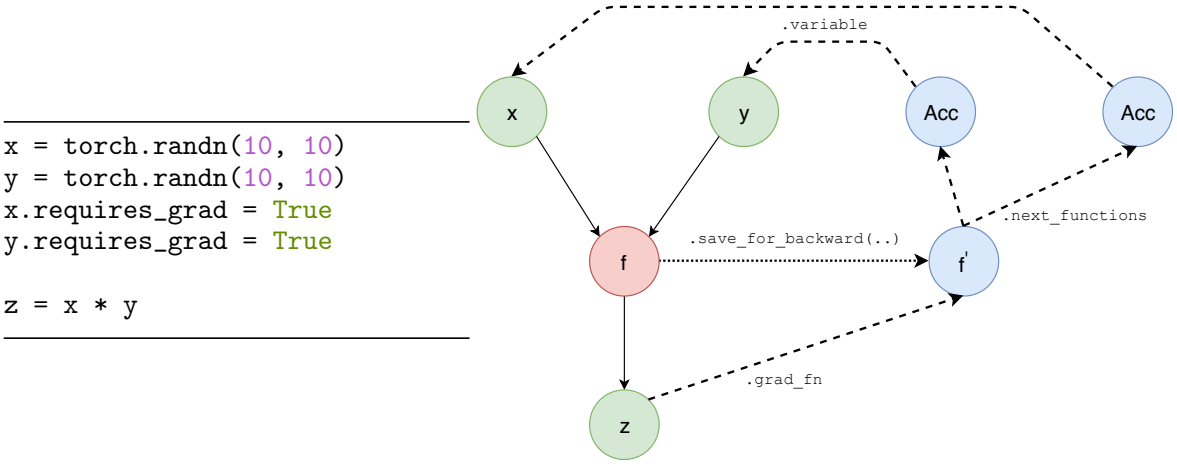


Figure 2.10: Forward PyTorch code and its autograd backwards graph. Green nodes are tensors. Red nodes are forward functions. Blue nodes are backward functions. Dashed edges are object composition (fields of the object). Dotted edges are function calls. Straight edges represent the forward computation.

First, `x` and `y` are constructed with `requires_grad=True`, telling Autograd to build a backward graph on computations that use them. They will also have `is_leaf=True` as they were explicitly set to a value by the user, rather than being constructed from existing tensors. When `z` is constructed by applying the operator `f` to `x` and `y`, Autograd sees that `x` and `y` require gradient. This results in a number of steps occuring:

1. `z` will require gradient.
2. `z.grad_fn` will be set to the corresponding backward operator of `f`, which will be denoted `f'`.
3. `f` will save handles to any tensors `f'` requires for the backwards pass, such as its inputs in the case of multiplication.
4. `f'.next_functions` will be set to point to the `.grad_fns` of `x` and `y`.
5. However, because they are leaves, actually `AccumulateGrad` backward functions will be constructed for each of them instead. These accumulate all incoming upstream gradients into `.grad` of their tensor so the user can access it after the backwards pass is complete.

Because `z` requires gradient and has its `.grad_fn` set, when subsequent operators are applied to it, the exact same process will occur, extending the backward graph. Thus, any time an operation is performed, the backward graph is being dynamically constructed alongside. This is why no static graph is required beforehand.

There is no forward graph during this process. The straight edges of Figure 2.10 represent the computation that occurs; there are no pointers between the objects.

Finally, when the user calls `.backward()` on the output tensor, the backward graph already in place will be traversed. An Autograd backward function takes upstream gradients for each of its outputs and produces downstream gradients for each of its inputs that require gradient, possibly using outputs saved from the forward pass. The gradients are then propagated downstream using the backward function's `.next_functions`. The process bottoms-out when this field is empty, usually when an `AccumulateGrad` for a leaf tensor has been reached. These accumulate the incoming upstream gradients into the tensor in their `.variable` field.

One thing to consider is mutations to the tensors. If a tensor is mutated after a computation is run and the backward graph defined, the backward function no longer has access to the tensor originally involved in the forward, because it only stored a handle to the tensor object, it did not make a copy of the data. Thus, though tensors are mutable in PyTorch, they will invalidate existing backward graphs involving that tensor, causing an error to be thrown if the user attempts to traverse that graph.

It can also be useful to detach tensors from the backward graph. Calling `.detach()` on a tensor will create a new tensor object (aliasing the same data) that is not part of the Autograd graph. The new tensor will be a leaf and does not require gradient. When the original backward graph is traversed, the new tensor will not have a `.grad_fn` that is invoked.

Detaching is often used to ensure the backward graph is cleared from memory when the backward pass has been done and the gradients are no longer required, but we would like to keep a forward tensor. I will use detaching in my implementation of checkpointing with multiple recomputations to manually control how execution moves forward and backward along a sequence.

For a more in-depth look on Autograd internals, I recommend starting with Elliot Waite's excellent tutorial on Youtube [3].

[3] https://youtu.be/MswxJw-8PvE, accessed 26/08/19

2.4 Memory Requirements of Training

In this section, I will show how to perform liveness analysis on an arbitrary computational graph to find the memory allocation strategy that yields the least peak memory. I will then apply this to the computational graph of training a neural network. The optimal policy solver for checkpointing presented later will assume these optimisations are in place.

Tim Salimans and Yaroslav Bulatov of OpenAI have a great article online [8] that, with animations, shows how the peak memory of a neural network arises; how to reduce this with checkpointing; and the limitation of chekckpointing to sequences. I highly recommend reading it. I borrow from their work in a few sections throughout this thesis, starting here. I will give a similar analysis of peak memory, but make it more precise for arbtirary costs, and eventually define the exact computation graph that will be used by my policy solver. In Section 2.5, I will give a more in-depth discussion of the existing checkpointing techniques, including those already studied in the automatic differentiation literature. In section 5.2, I will also show how checkpointing is limited to sequences, as well as reference some recent approaches for tackling this in neural networks.

2.4.1 Pebbling

Sethi introduced the 'pebble game' as a framework for understanding how much memory must be allocated to a computation [56]. In the game, 'pebbles' are placed on the nodes, representing their allocation in memory. The following rules govern the placement of the pebbles:

1. For any $c = f(a, b)$; c cannot be computed until both a and b have been allocated.
2. Once no computation relies on b being in memory, it can be deallocated.

However, these rules allow for many allocation strategies - the order in which we place pebbles, remove them, and reuse them for other nodes.

Consider the graph in Figure 2.11.

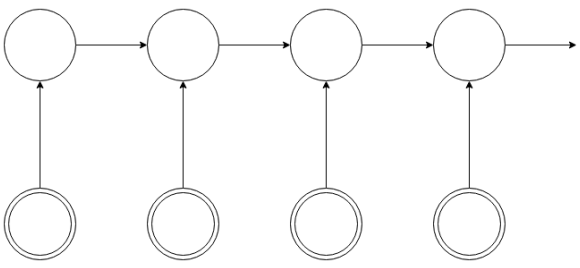


Figure 2.11: Example graph for pebbling.

One strategy for this graph would be to allocate all the inputs immediately. This is shown in Figure 2.12. Coloured nodes are 'pebbled' - allocated in memory. By inspection, we can see the most pebbles (5) are allocated in the second step. Generalising, for a sequence of size n, $\Theta(n)$ memory is required. However, if we instead place pebbles as late as possible, like in Figure 2.13, we get a $\Theta(1)$ strategy.

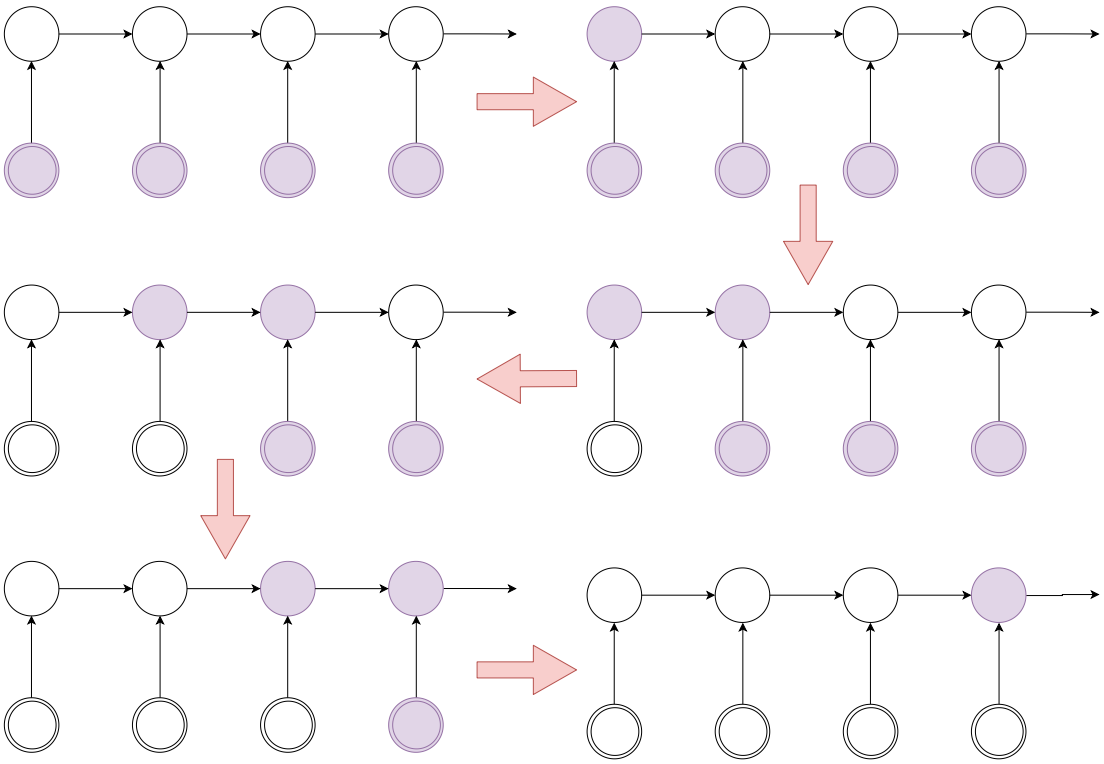


Figure 2.12: A poor pebbling strategy that allocates all leaves immediately. Peak memory is 5 pebbles, occuring in step 2.

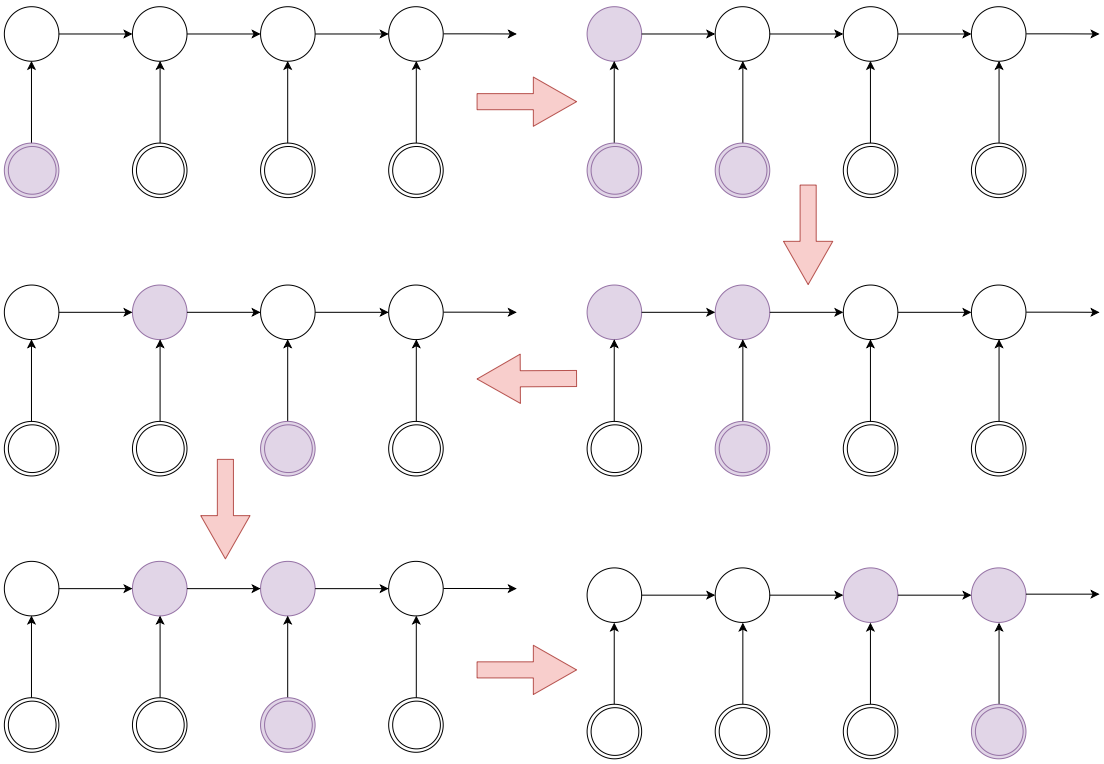


Figure 2.13: A good pebbling strategy that allocates leaves lazily. Can handle any chain length with 3 pebbles.

2.4.2 Liveness Analysis

In general, finding an optimal strategy is hard [57], even in this simplified game where we assume all the nodes have a uniform memory cost. An example use case is register allocation in compilers [58]. The first step of register allocation is to determine the lifetimes of each node relative to each other, and thus how their lifetimes overlap. Variables with overlapping lifetimes cannot be allocated to the same register. Next, from the liveness analysis, an *interference graph* is constructed. The nodes of this graph are the variables and they have arcs between them if their lifetimes overlap. Then, graph colouring is performed. The colours represent the registers, so this gives an assigment of registers to variables such that no overlapping variables have the same register, using the least amount of registers.

However, this approach takes $\Theta(n^2)$ to solve, which will be too slow for the size of large neural networks. We can instead use some reference counting heuristics that are linear time [1]. Essentially, we simulate the computation whilst keeping track of, for each node, how many nodes that use it are still yet to be computed. From this, we can perform two optimisations:

- **Memory Sharing:** Once a node's count has reached zero, it is no longer needed, so its memory can be reused for a different node. This simply means we reuse pebbles as soon as they become available.
- **In-place Operation:** When a node's count is one, only one computation depending on it is left. That computation can be performed in-place on the already allocated memory.

An example of applying these optimisations is shown in Figure 2.14.

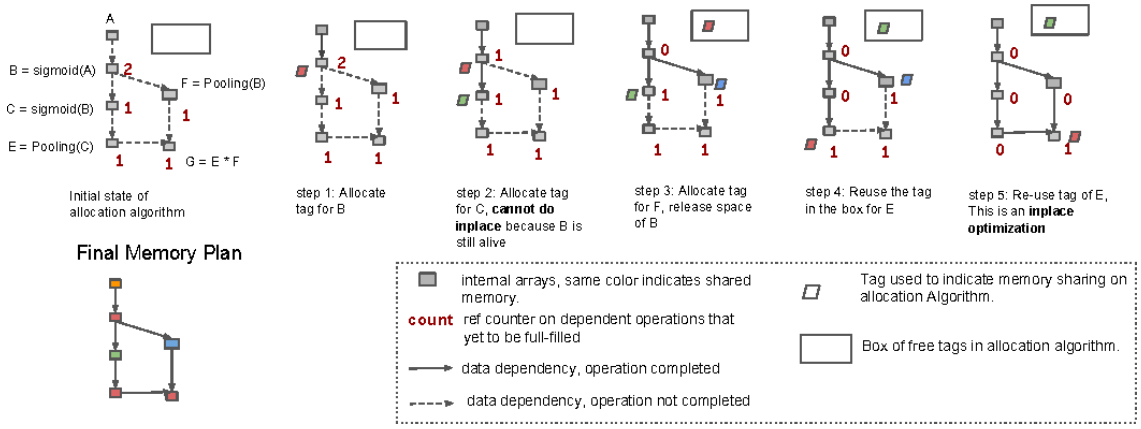


Figure 2.14: Memory allocation algorithm on a computation graph. Each node is associated with a liveness counter to count on operations to be fullfilled. A temporal tag is used to indicate memory sharing. In-place operation can be carried out when the current operations is the only one left (input of counter equals 1). The tag of a node can be recycled when the nodes counter goes to zero. [1, Figure 2]

For the examples shown previously, this will find the $\Theta(1)$ strategy from Figure 2.13,

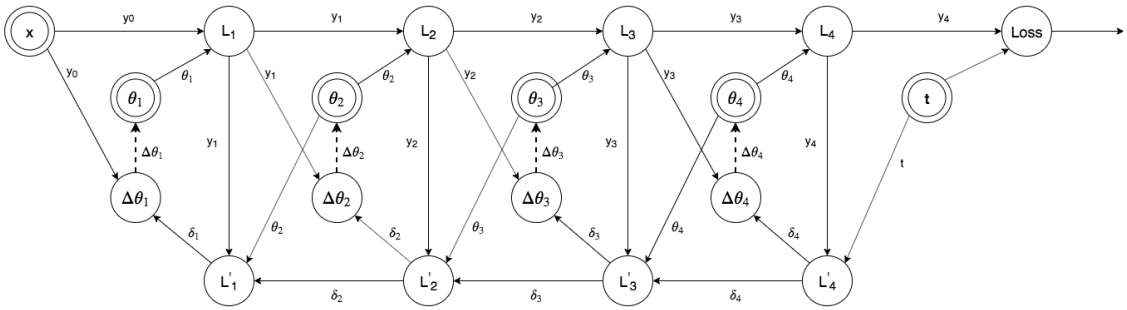


Figure 2.15: The computational graph for one step of gradient descent on a neural network of four layers.

in only $\Theta(n)$ time.

As stated in the whitepaper, PyTorch performs these optimisations aggressively during the backward pass [31, p. 3].

2.4.3 Peak Memory Usage During Training

Recall the computational graph for one step of training a neural network, repeated in Figure 2.15.

We need to analyse what is actually consuming memory and what is the bottleneck. From the graph, we can see what needs to be allocated:

- The forwards;
- The backwards;
- The weights;
- The targets.

However, there is some less obvious memory consumption too.

- **Workspaces**: Different algorithms for an operator can have different memory requirements. The memory required for an algorithm is known as the workspace.
- **Additional Optimiser Memory**: As mentioned before, in practice vanilla SGD is not used much. Many of the other optimisers require extra memory than just the $\Delta\theta$. For example they may smooth the update of a parameter by taking some average of the gradient and the previous gradients, in which case the previous gradient must be stored for every parameter.
- **Miscallenous**: The implementation used will likely allocate some miscallenous memory. Between versions of a framework, the exact configuration used, etc., it would be infeasible to account for this statically in a cost model.

In my implementation, I will account for the workspace memory as part of the memory required for the forwards and backwards operations.

I will not apply checkpointing to the weights or additional optimiser memory because they only use a small proportion of the total memory reqiurement. Figure 2.16 shows a break down of memory consumption of popular network architectures. As can be seen, the proportion of memory allocated to the weights is very small. The reason is that modern deep networks hardly use fully-connected layers, which cause an exponential growth in the number of weights. Instead, we mostly see layers such as the convolutional layer. These typically apply a small filter, such as of size 3x3, to many large images. The image is the layer output and the filter weights are the learnable parameters. Thus the parameters are very small compared to the forward and backward tensors. As the optimiser memory is usually equivalent to one or two extra values *per parameter*, they are omitted from the optimisations too.

Therefore, we can exclude the parameter memory from our analysis. When checkpointing is applied, it will be assumed that the parameters and their gradients persist in memory throughout the forwards and backwards passes. However, performing the backwards operators will still involve computing their updates, so the cost of this must be included. I disucss the modelling of the computational costs because solving for the optimal checkpointing policy requires knowing the per-layer computational costs as well as the memory-costs, since it trades compute for memory.

Omitting the parameters from the computational graph in Figure 2.15 results in the graph typically found in the literature, with some slight tweaks, shown in Figure 2.17.

The loss node has also been omitted for brevity.

The forward operators L_i represent the same forward functions as before. f_{i-1} is their input and f_i is their output. The memory cost of f_i is modelled as all the (non-parameter) memory required to compute y_i in the original graph, including the workspace and y_i itself. The cost of computing f_i is just the cost of performing L_i to get y_i

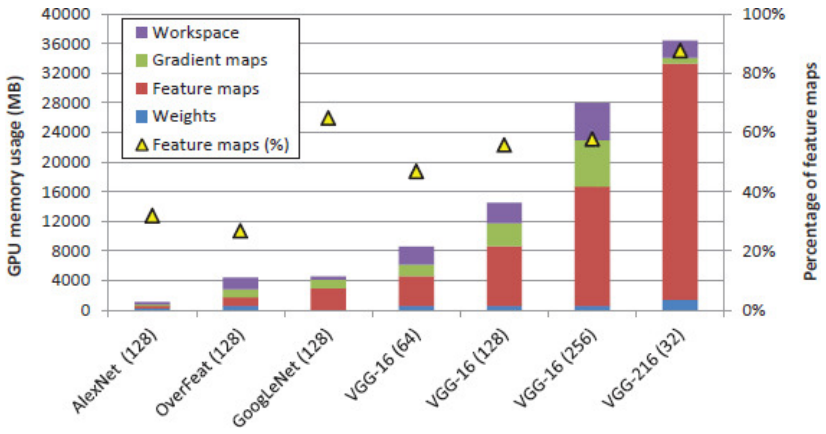


Figure 2.16: Breakdown of memory usage of popular networks. [7, Figure 4]

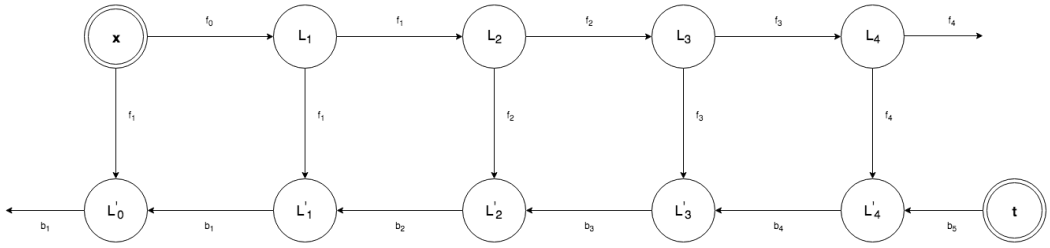


Figure 2.17: Computational graph for one step of training, not showing the parameters or their gradients.

The backward operators L'_i have been defined differently to further simplify the graph. They perform two operations: (i) using the output of layer y_i and the upstream backward δ^{i+1}, compute the upstream weight gradients $\Delta\theta_{i+1}$; and (ii) again using the output of layer y_i and the upstream delta δ^{i+1}, compute the downstream delta δ^i. This means L'_{i-1} will compute $\Delta\theta_i$. Consequently, an L'_0 must be added to the graph to represent the computation of $\Delta\theta_0$.

The memory cost of b_i is thus all the memory required for δ^i, but not that for $\Delta\theta_{i+1}$ as we have excluded parameter memory. However, the computational cost will include the cost to compute $\Delta\theta_{i+1}$.

f_0 is the inputs. For a network of N layers, f_N is the output of the network. b_{N+1} is the targets. b_0 is not really a tensor that exists; it represents the output of L'_0, which is there to set $\Delta\theta_0$, but does not actually output anything. However, as this incurs some computational cost, which will affect what strategy best optimises the network, we still model this 'tensor'.

From here, we can clearly see how peak memory occurs, visualised in Figure 2.18. As the forwards are required again in the backwards pass, they must be kept in memory. If there were no backwards pass, we could perform all the forwards in-place in $\Theta(1)$ memory, but instead we must keep them in memory until the backwards pass has used them, resulting in $\Theta(N)$ memory. Only once b_i has been computed, can we finally free f_i and b_{i+1}. As a result, peak memory occurs at the end of the forward pass, just before we start backpropagating.

This is the result found in the literature, where per-layer uniform costs are assumed. However, I am going to be more precise than this; because the sizes of the tensors vary, peak memory could occur in say the third layer instead of the fourth, if the size of the newly allocated backward is greater than the size of the freed forward and backward together.

Thus, if $\beta : \{f, b\} \times [0, N] \to \mathbb{Z}^+$ denotes the memory requirement of the i^{th} forward or backward tensor, the peak memory of an N layer network becomes:

$$\max_{0 \le i \le N} \sum_{l=0}^{i} \beta_l^f + \beta_{i+1}^b + \beta_i^b \tag{2.19}$$

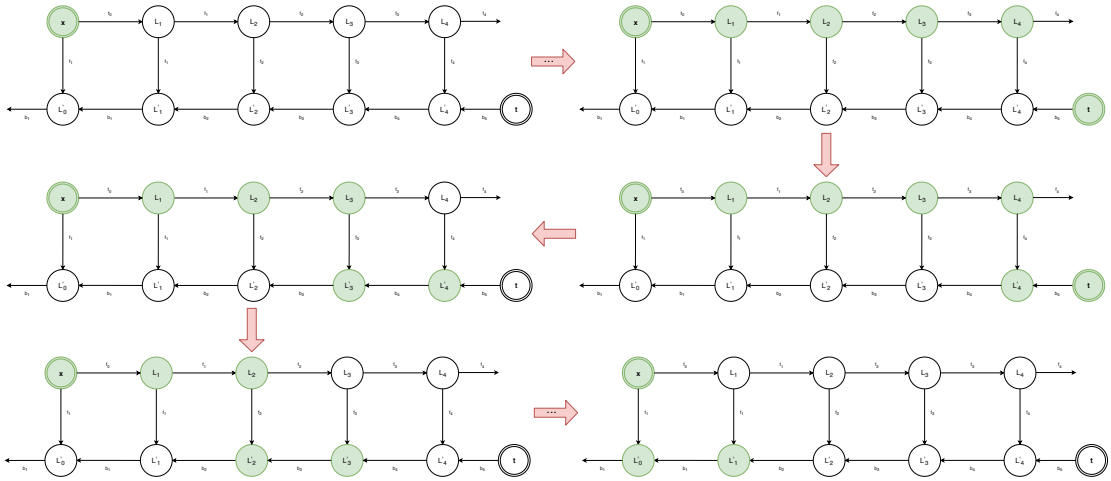


Figure 2.18: Peak memory during backpropagation.

2.5 Checkpointing

In this section, I will show how to reduce the peak memory established in the previous section using checkpointing and motivate my extension to Gruslys et al.'s technique [2]. I will also show that checkpointing has been rediscovered from the AD literature.

2.5.1 $\Theta(\sqrt{N})$ Checkpointing

Consider the simplified computational graph of backpropagation that was derived in the previous Section 2.4, or more generically, the graph for computing the adjoint of a sequence of operations. I have shown this to have linear space complexity in Section 2.4, due to the forwards being stored for use in the backwards pass. We can elide this restriction by trading off compute for memory, using what is known as *checkpointing*.

In checkpointing, only some of the forwards are stored in memory, known as the checkpoints, or snapshots. Then, during the backwards pass, forwards are recomputed from the last checkpoint in the chain, this time keeping them in memory so the backwards pass can proceed. Thus, the checkpoints have split the sequence into segments, and to *compute the adjoint of the entire sequence*, we first run the forward pass only storing the checkpoints (which demaracte the segments), then proceed backwards by *computing the adjoints segment-by-segment*, where computing the adjoint of a segment involves recomputing the forwards from the last checkpoint and this time storing them in memory so the adjoint can proceed. This segmented backwards pass is visualised in Figure 2.19, repeated from the introduction.

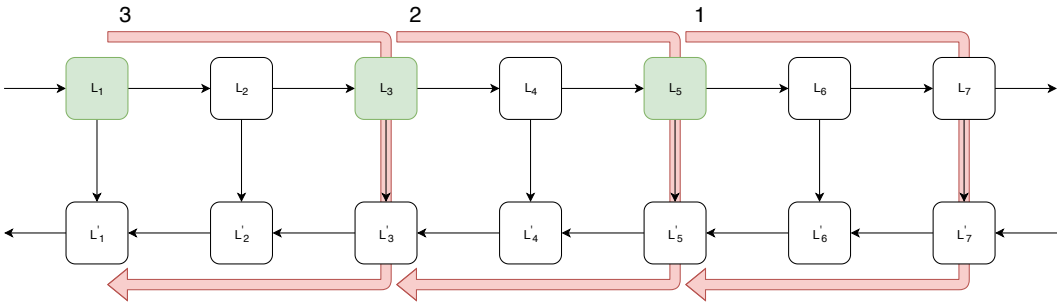


Figure 2.19: Segmented backpropagation with checkpointing: Say we are computing the backward pass. The forward pass has already been computed, and only the outputs of L_1, L_3, and L_5 were stored. Then, to perform the backward pass, we perform backproagation (both forward and backward) for segments 1, then 2, then 3, as shown.

Assuming uniform per-operator costs, the new peak memory requirement still occurs at the end of the chain, but is now lowered to only the cost of the checkpoints, rather than the entire forward sequence, plus the cost of computing the adjoint of the final segment. Assuming $k-1$ checkpoints that split the sequence of length n into k evenly-sized segments, the peak memory cost is approximately:

$$g(n) = \Theta(n/k) + \Theta(k)$$

With basic calculus, it can be found that $k = \sqrt{n}$ minimises this, with a sublinear cost of $2\sqrt{n}$. As every forward is recomputed at most once, the computational overhead is no more than the cost of a single extra forward pass. According to Chen [1], the forward pass is about twice as fast as the backward pass, resulting in the actual overhead being around 30%.

2.5.2 Multiple Recomputations

The peak memory cost can be reduced even further by introducing *multiple recomputations*. The above technique split the adjoint computation of a sequence into the adjoint computations of segments of this sequence. There is no reason that, during the adjoint computation of a segment, we cannot recursively apply the technique by splitting the segment into sub-segments.

That is, consider the scenario where the forward pass of the sequence has been computed and we are now processing a segment. We compute the forwards of the segment (their first recomputation), storing only some checkpoints. Then we perform the adjoint computation on each sub-segment, resulting in the dropped forwards being recomputed a second time. We could keep recursively applying the technique on the sub-segments resulting in many recomputations.

A visualisation of how we move forwards and backwards along a sequential computation during multiple recomputations is shown in Figure 2.20.

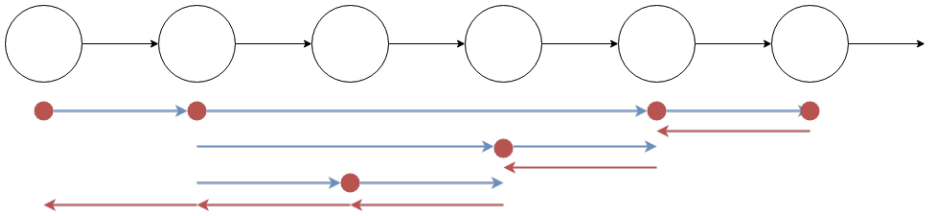


Figure 2.20: Execution of multiple recomputations on a sequence. Blue arrows represent forward computation. Red arrows represent backward computation. Red circles denote a forward being checkpointed. Moving vertically down represents the next recursion depth - blue arrows on the i^{th} line are being computed for the i^{th} time, or recomputed for the $i-1^{\text{th}}$ time.

When applying this recursive scheme, always storing k evenly-spaced checkpoints per segment, the peak memory cost becomes the cost of the k checkpoints plus the (recursive) cost of one segment:

$$g(n) = k + g(n/k+1)$$

Solving this gives:

$$g(n) = k \log_{k+1}(n)$$

Setting $k = 1$ minimises this to $\log_2(n)$. Setting up a similar recurrence for the computational cost shows it has increased to $n \log_k(n)$. This demonstrates how, through multiple recomputations, we have further traded compute for memory.

Going beyond this recursive formulation, the extreme case of simply 'recomputing everything' gives $\Theta(1)$ memory at $\Theta(n^2)$ computational cost: Say we have the forward at the start of the chain, f_0, and the backward at the end, b_n; we compute the forwards in-place, from f_0 to f_{n-1}, then use that and b_n to compute b_{n-1}. Thus, we have done one step of the backwards pass in $\Theta(1)$ space. To perform the next step, we simply repeat this process, computing in-place f_0 to f_{n-2} and using b_{n-1} to get b_{n-2}, and so on until we have b_0, still only using $\Theta(1)$ space. The computational overhead is quadratic because the first forward pass is over n steps, then the second $n-1$ steps, then $n-2$ steps, and so on. Summing these gives $\Theta(n^2)$.

2.5.3 Limitations of Checkpointing Techniques and their Implementations

Griewank first proposed all of these techniques for sequential adjoint computations in 1992 [28], later developed into the `REVOLVE` algorithm with Walther [59]. Given a fixed number of checkpoints per recomputation pass and the sequence length, it solves for a strategy that uses the least recursion depth (the max number of recomputation passes). For a recursion depth of r and number of checkpoints s, a sequence of length $\binom{s+r}{r}$ can be executed in logarithmic space, where $\binom{\cdot}{\cdot}$ is the binomial coefficient. However, their work assumes uniform per-operator costs. A lot of AD literatature is in the domain of differential equations on time, where such assumptions may hold, but it is not the case for most neural networks.

Martens and Sutskever [26] briefly outlined the (uniform costs) $\sqrt{(}n)$ technique for neural networks in 2012. Chen made the technique more widely known with his 2016 paper [1] and corresponding implementation in MxNet [60]. It has since been implemented in PyTorch [61] and TensorFlow [32]. Chen also presented the asymptotic analysis for the recursive scheme and gave an algorithm that, given the original (arbitrary) computational graph and the number of times to recompute each individual forward, will construct a new graph that performs the adjoint computation according to this policy, using 'node mirroring'.

Though these techniques greatly reduce the memory cost by trading off computation, they pick an arbitrary point on the compute-memory trade-off curve. That is, given infinite memory, we could of course do any computation with no overhead, as we would simply not use checkpointing. However, as the memory budget is tightened, we are forced to checkpoint, incurring compute overhead. Ideally, the user should specify their memory budget and the system should find the checkpointing strategy that satisfies this budget with the least computational cost, rather than having the user specify arbitrary parameters like the number of checkpoints. They should not even have to know what checkpointing is.

Chen proposed a heuristic algorithm that, given the memory budget and per-operator memory costs, traverses the sequence and greedily chooses to drop nodes until the cost of that segment exceeds the budget, at which point the current node is instead

checkpointed, starting a new segment. To try to match the memory savings of $\sqrt{n}$ checkpointing for sequences with abitrary per-operator costs, another heuristic is used for guiding a grid search over the memory budget to find the plan returned by the algorithm that gives the least memory cost. Though this does not give the provably optimal memory cost, they do report good results. However, the more important limitations are that this is restricted to one recomputation and that it makes no attempt to optimise for computational cost, only to minimise the memory cost.

Again, ideally, we want to solve for the optimal checkpointing policy that (i) minimises computational cost whilst satisfying a memory budget, and (ii) does so according to the precise per-operator compute and memory costs.

Before detailing Gruslys et al.'s [2] work on memory-efficient RNN training, which achieves the former goal but not the latter, I will briefly discuss some related work on checkpointing.

2.5.4 Related Work

`REVOLVE` gives the optimal recursion depth for sequences of known length. Work has also been done for sequences of unknown length. `REVOLVE`'s optimality has been matched for sequences that do not exceed a length that has a certain bound with respect to the number of checkpoints per segment. Some deep learning models, like `seq2seq` RNNs, do have unknown sequence length, so there could be some opportunity here. However, I only consider sequential neural networks where the sequence length is known.

Dauvergne and Hascoët reframe checkpointing in terms of the data-flow equations used in compilers [25]. They also formalise what other data-flow techniques from compilers look like for automatic differentiation, such as liveness analysis and *to be recorderd* analysis, and seek to combine them with checkpointing in their `TAPENADE` platform [62].

Siskind and Pearlmutter [27] generalised `REVOLVE` to arbitrary computation *trees*, rather than just sequences. However, their implementation does not take into account precise per-operator costs and does not solve for the least computational cost given a memory budget, but instead gives `REVOLVE`-like optimality by finding the strategy of least recursion depth. Also, though non-linear architectures are now prevalent in neural networks, they tend to not be trees either.

2.5.5 Optimal Compute Cost Checkpointing Through Dynamic Programming

Gruslys et al. solve for the checkpointing policy that gives the least computational cost, subject to a memory budget [2]. They formulate this as a dynamic programming

(DP) problem where the subproblems are over the sub-sequences and the size of the memory budget. The solution is defined for RNNs, where every layer in the sequence is an identitical RNN *cell*, leading to uniform per-layer costs. The first implication of this is that they can talk about memory in terms of 'memory slots' instead of precisely-sized buffers, where one slot is a workspace for everything needed to compute the forwards and backwards of one cell. Similarly, computational cost can be described using arbitrary units. Secondly, every sub-sequence of length t is identitical, so there is no need to solve subproblems for all possible subsequences, but only over sequences of length $1 \leq t \leq n$.

The problem is formulated similarly to how Markov Decision Processes are optimised in Reinforcement Learning [63, 64]. There is a cost function for states $C(t, m)$, where a state is a pair of the sequence length t and the number of memory slots m. This returns the least computational cost achievable for that pair. $C(t, m)$ is defined recursively on the subproblems for smaller sequences and fewer memory slots. There is a policy $D(t, m)$ that says for each state what action should be chosen - that is, what layer should next be checkpointed. $1 \leq y < t$ denotes the action and is defined as the offset to the next layer to be checkpointed. The cost $C(t, m)$ of a state depends on what the policy says we should do in that state. The cost-action function $Q(t, m, y)$ is the cost of the state (t, m) if we choose action y.

Thus, to find the optimal $C(t, m)$ we must know the optimal costs for all the subproblems, so we can choose the action y that gives the optimal cost for this problem. We therefore need to co-optimise these mutually recursive functions, which are defined as follows:

$$
\begin{aligned}
C(t,\, m) &= Q(t,\, m,\, D(t,\, m)) \\
Q(t,\, m,\, y) &= \text{some function of } y \text{ and relevant subproblems } C(t',\, m') \\
D(t,\, m) &= \operatorname*{argmin}_{1 \leq y < t} Q(t,\, m,\, y)
\end{aligned}
$$

This can be solved using standard DP techniques. Gruslys et al. use bottom-up value iteration. I will give the algorithm after fully describing the solution.

That is, the base cases of C and the recursive case Q that actually solve our original problem: finding the least computational cost within which we can perform back-propagation on a sequence of length t whilst satisyfing the memory budget m, using checkpointing.

First, the base cases. For $C(1, m)$, we simply perform the forwards and backwards for that single cell, giving a cost of 1. For $C(t, 1)$, we use the $\Theta(1)$ space technique discussed above that recomputes everything, resulting in a quadratic cost. For $C(t, m)$ where $m \geq t$, there is enough memory to not do any checkpointing, resulting in linear computational cost.

The recursive case can be formulated as shown in Figure 2.21, taken from the original

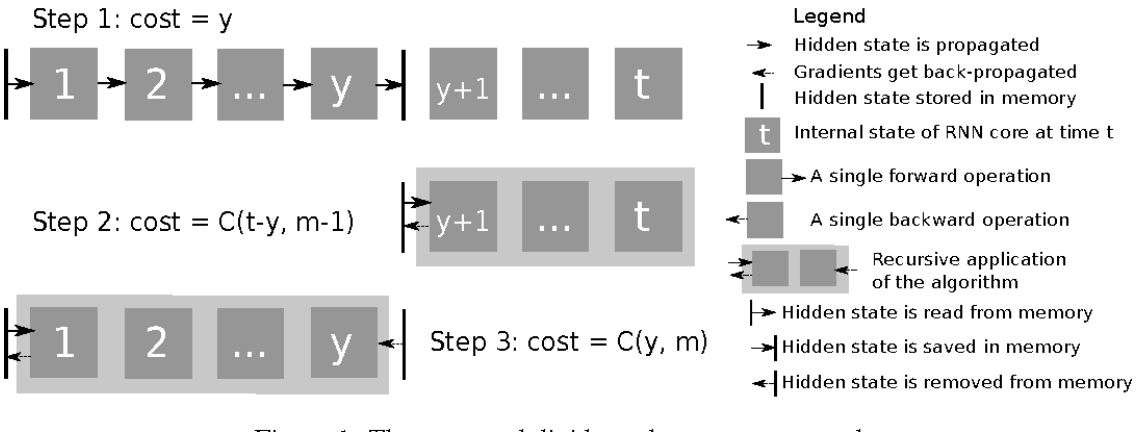


Figure 2.21: The recursive case $Q(t, m, y)$: computing the forwards to y, recursing on the right whilst storing y, and recursing on the left with all the slots [2, Figure 1]. 'hidden states' refer to the forwards.

paper[4]. For $Q(t, m, y)$, we first compute the forwards up to y without storing them, incurring a compute cost of y. Then, we recurse on the right hand side holding y in a memory slot. That means we are recursing on a sequence of length $t-y$ with $m-1$ slots. Once this returns, we can recurse on the left hand side. The left sequence is of length y and we can use all y slots, because the forward of y is not required to backpropagate on the left hand side. Thus, the cost-action function is defined as:

$$Q(t, m, y) = y + C(t - y, m - 1) + C(y, m)$$

The algorithm for solving these equations is given in Algorithm 2.1. I do not explicity define the linear cost base case as it depends on whether you are storing the full internal state of an RNN cell or just the hidden state and recomputing the internal state. I do not dwell on this further as I will be extending the technique for general feedforward networks where this is not a consideration.

The algorithm is for solving these equations is bottom-up: It first sets the base cases, before moving 'up' through the search space, solving for longer sequences and more memory slots recursively. The recursive case chooses the y that gives the least $Q(t, m, y)$. It sets $C(t, m)$ to that cost and $D(t, m)$ to y.

Once we have the optimal policy, we can 'traverse' it to perform backpropagation according to the policy. Say we are given f_i and b_j, initialised to the inputs and targets, to perform backpropagation on this segment to compute b_i within m memory slots, we do the following:

1. If the sequence length $t = j-i$ is 1, we simply perform the backward operation.
2. If m is 1, we use the constant-space, quadratic-cost strategy.

[4]This is spcifically for BPTT-HSM, where the hidden states are checkpointed. See the paper for more details [2].

3. Else, letting $k = D[t][m]$, we run the forwards to f_k, checkpoint it, recurse on the right to get b_k, release f_k, then recurse on the left to get b_i.

Algorithm 2.1: Optimal policy solver using dynamic programming [2, Algorithm 1]

Input: t_{max} the maximum sequence length
Input: m_{max} the maximum memory capacity

Let C, D be 2D arrays of size $t_{max} \times m_{max}$
for $t \in \{1, \ldots, t_{max}\}$ **do**
    $C[t][1] \leftarrow \frac{t(t+1)}{2}$
    for $m \in \{t, \ldots, m_{max}\}$ **do**
        $C[t][m] \leftarrow$ *linear cost*
        $D[t][m] \leftarrow 1$
for $m \in \{2, \ldots, m_{max}\}$ **do**
    for $t \in \{m+1, \ldots, m_{max}\}$ **do**
        $C_{min} \leftarrow \infty$
        for $y \in \{1, \ldots, t-1\}$ **do**
            $c \leftarrow y + C[y][m] + C[t-y][m-1]$
            if $c < C_{min}$ **then**
                $C_{min} \leftarrow c$
                $D[t][m] \leftarrow y$
        $C[t][m] \leftarrow C_{min}$
return (C, D)

Chapter 3

Implementation

In this chapter, I will detail the contributions I make in this thesis. First, I will show how to generalise the uniform-cost assumptions in Gruslys et al.'s policy solver. At least to me, this seemed fairly trivial at first thought, but it turns out there are a lot of gritty details to get through. My explanation will gradually amend their algorithm as I consider the effects of relaxing the assumptions.

Secondly, I will present my implementation in PyTorch, which takes a sequence of layers, profiles them, solves for the optimal policy, and performs backpropagation on the sequence according to the checkpointing policy. I will give the algorithm for executing backpropagation according to the policy, essentially the same as the one used by Gruslys et al., but with some PyTorch-specific considerations. As I describe the implementation, I will motivate and evaluate some of the design decisions, such as the API it presents to the user.

3.1 Optimal Policy Solver with Precise Per-Layer Costs

It is important to have a firm grasp of Algorithm 2.1 before reading this section.

Let us restate the exact problem it is solving, which is for a sequence of uniform layers:

Find the optimal cost $C(t, m)$ for performing backpropagation on a sequence of length t within m memory slots, using checkpointing.

How do we generalise this to take into account precise per-layer compute and memory costs? In this section, I will explain this by deriving the shortcomings of the original policy solver when tasked with handling per-layer costs, and then showing how to overcome them.

3.1.1 Introducing Per-Layer Costs

Let $\alpha : \{f, b\} \times [0, N+1] \to \mathbb{R}^+$ and $\beta : \{f, b\} \times [0, N+1] \to \mathbb{Z}^+$ be the per-layer forward and backward compute costs and memory costs, respectively, denoted α_i^f for the computational cost of the i^{th} forward, for example.

We can no longer talk about arbitrary sequences of length t, as each individual subsequence of length t will have different costs as defined by α and β. Thus the cost function now becomes $C(i, j, m; \alpha, \beta)$, and similarly the policy $D(i, j, m; \alpha, \beta)$. The parameters α and β will be omitted for brevity when it is clear from the context what they are.

The exact definition of $C(i, j, m; \alpha, \beta)$ becomes:

Given f_i and b_j, the optimal computational cost according to α and β to perform backpropagation on the segment, that is to compute b_i, within m remaining memory, using checkpointing.

To recap what these tensors are, consider the simplified computational graph for backpropagation not including the weights given in Figure 3.1, as derived in Section 2.4.3.

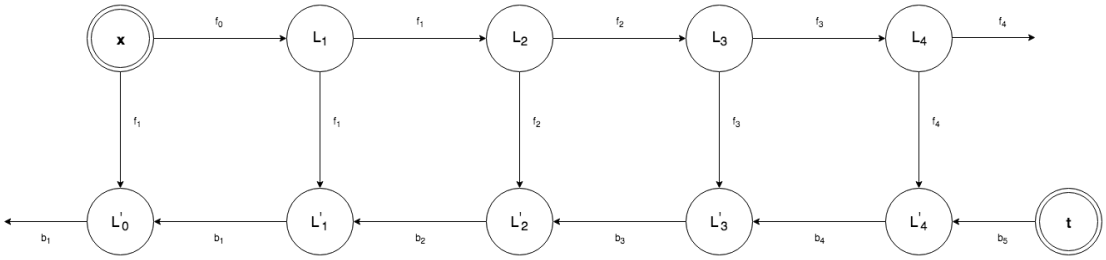


Figure 3.1: Computational graph for backpropagation on a sequence, not showing the parameters.

Layer i's forward operator takes input f_{i-1} and produces f_i. Its backward takes f_i and b_{i+1} and produces b_i. The network inputs are f_0 and the targets are b_{N+1}, for an N layer network.

Now that we have this all defined, we can think about how it affects the solver algorithm.

3.1.2 Traversing the Subproblems Bottom-Up

Iterating over the search space bottom-up no longer means iterating over increasing sequence length t, but over all possible subsequences in 'subproblem order' - when we reach the subsequence (i, j), we should have already solved all *its* subproblems. We can break this down recursively by saying that for any $i < k < j$, we have already encountered all subsequences of (i, k) and (k, j). Note, given my formulation of the

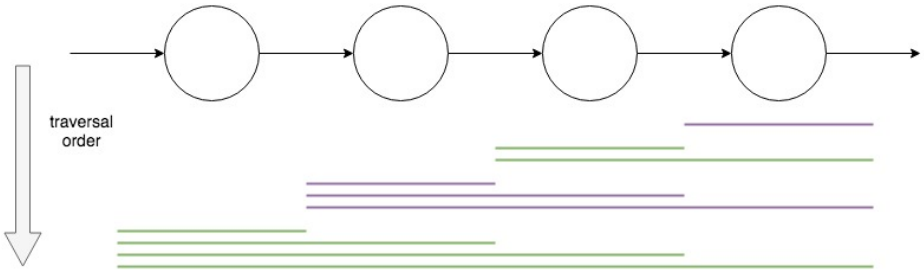


Figure 3.2: Visualisation of traversing all subsequences in 'subproblem order'. The horizontal bars represent a subproblem (i, j). Traversal order goes from top to bottom. Notice that, whenever an (i, j) is encountered, all of its subproblems have already been done.

computational graph, only subsequences where $i < j$ are defined, so (i, i) is not a valid subsequence.

The way in which I traverse the subsequences is visualised in Figure 3.2. For some sequence (s, e), we will traverse all subsequences using iterators i and j. We start at the end, that is $i = e - 1$ and $j = e$. We cannot move j further right. We have now solved for all subsequences that start with $i = e - 1$. We move i left one to $e - 2$ and initialise j to the first one to the right of i, $e - 1$. Now, as we move j further right, we have already done (memoised) all the subproblems (i', j), where $i' > i$. In this case, that means we have done all subsequences starting with $i = e - 1$, which is true. This means that, whenever we are considering a problem (i, j), we have already solved all subproblems of (i, i') and (i', j), so never need to go 'down' more than one level, but can use the memoised results. Once we have moved j all the way right, we have now done all subproblems for $i \geq e - 2$. Again, we move i left one to $e - 3$ and initialise j to $e - 2$. Once we have solved this case, we move j right, knowing that we have already solved all (i, i') and (i', j), for all $i' \geq i$. We keep traversing like this until eventually we have moved i left to i, and then j right to e.

3.1.3 Redefining $Q(\cdot)$ with Per-Layer Costs

To recap, so far we have introduced per-layer costs, redefined the problem to be on all possible subsequences, and seen how to set up the solver to traverse the subsequences bottom-up. Next, we give the new definition of the cost-action function that takes into account the per-layer costs:

$$Q(i, j, k, m) = \sum_{l=i+1}^{k} \alpha_l^f + C(k, j, m - \beta_k^f) + C(i, k, m) \tag{3.1}$$

This is the same as before, except we now incur the *precise* costs of executing the forwards to f_k, and recurse on the right with the *precise* reduction in memory from checkpointing f_k. Note f_i is given by the definition of the problem so we start the sum of the forwards at f_{i+1}.

It would seem we have finished describing the new solver. However, all is not so simple; the fact that we are now being precise about memory causes a lot of issues.

3.1.4 Handling the Continuous Memory State by Tracking the Peak Memory Cost of Subproblems, $B(\cdot)$

Previously, for RNNs, we divided memory into coarse-grained slots, and every layer always used exactly one slot - no more, no less. Now, layers use a precise amount of memory, so when we checkpoint a tensor and recurse to the right, the new memory budget could be any arbitrary number. This means memory is now a *continuous* state. As mentioned, this causes a lot of problems. Primarily, we now have a much larger state space to iterate over in the policy solver, as, no matter what the arbitrary new memory budget is, we need to have solved a subproblem for it. Thus, a 10GB GPU would result in us having to iterate from $m = 1$ all the way to 10 billion. I will show in Section 4.2 that this can be dealt with by bucketing memory, and that it only has a reasonable affect on optimality.

Secondly, because layers use a specific amount of memory, the optimal solution to some $C(i, j, m)$ may not use all of the m permitted memory. We want to take advantage of that - if the optimal solutions for the left and right subproblems use less memory than required, we have less pressure at this 'level' to trade-off compute for memory, resulting in a lower computational cost. This means, as well as $C(i, j, m)$, we need to track a $B(i, j, m)$ that represents the peak memory required for the corresponding solution to $C(i, j, m)$. To be clear, $B(i, j, m)$ means f_i and b_j have already been computed and are in memory, so we have m memory remaining in which to compute b_i.

Recall computing $C(i, j, m)$ requires computing $Q(i, j, k, m)$ for all possible actions k, then picking the optimal one. Likewise, to find the corresponding $B(i, j, m)$ we must know how to calculate the peak memory of each $Q(i, j, k, m)$, then set $B(\cdot)$ (and $C(\cdot)$) to the peak memory (and compute cost) of the optimal $Q(\cdot)$.

What, then, is the peak memory required for $Q(i, j, k, m)$? We consider the peak memory of the three stages of computing it.

1. **Compute the forwards from f_{i+1} to f_k**: The forwards are done in place, so peak memory for this is the maximum pairwise $f_{l-1} + f_l$ across the layers. For the computation of f_{i+1}, though, we exclude the memory of f_i as the definition of $B(\cdot)$ says it is already in memory, so not required to be within the m bound.

2. **Recurse on the right**: We hold f_k in memory and recurse on the right, so the peak memory cost becomes β_k^f plus the cost of the right. We assume by induction that we have $B(k, j, m - \beta_k^f)$.

3. **Recurse on the left**: We assume by induction we have $B(i, k, m)$.

The peak memory is the max of the three stages:

$$
\begin{aligned}
&\max(\\
&\quad \beta^f_{i+1}, \\
&\quad \max_{i+1<l\leq k} \beta^f_{l-1} + \beta^f_l, \\
&\quad \beta^f_k + B(k, j, m - \beta^f_k), \\
&\quad B(i, k, m) \\
&)
\end{aligned}
\tag{3.2}
$$

All would seem well then, except it does not quite work due to the definition of $C(i, j, m)$ stating b_j is already in memory elsewhere, not within the m memory, but I will explain this after first addressing the case of failure.

3.1.5 Handling Failure When Memory Is Insufficient

Again, because we are using precise memory costs, not coarse-grained memory slots, it is not guaranteed that $C(i, j, m)$ can actually be done in m memory. Previously, we could always fall back to the 'recompute everything' case that could compute any sequence within one slot. Now, it is possible, for example, to have $m = 50$ but a sequence with a tensor or size 100. In this case, not even the 'recompute everything' strategy can save us; we have failed. This means, in $Q(i, j, k, m)$, we must check for failure.

First, we see if the peak memory cost of running the forwards to the checkpoint exceeds m. Then, before we can even look at the right subproblem, we must check that $m - \beta^f_k > 0$. Once we have done that, we can check that neither the right or left subproblems resulted in failure. Finally, we compute the peak memory of our cost-action $Q(\cdot)$ according to the above Equation 3.2, and check if that is under m.

If any one of these checks fails, this action k fails. If every action k fails, we have run out of options and must set the subproblem for (i, j, m) to failure. I choose to propagate failure through $B(\cdot)$, rather than $C(\cdot)$. All this means is that we check $B(\cdot)$ for failure of a subproblem and set failure of our current subproblem in $B(\cdot)$, rather than $C(\cdot)$.

I will elaborate on how failure propagates 'up' through the search space later, after I give the algorithm as derived so far, as it will be more clear then. Before that, as promised, we must talk about how our expression for $Q(i, j, k, m)$ in Equation 3.1 is not quite right.

3.1.6 Delegating Ownership of b_j to the Subproblems

The definition of $Q(i, j, k, m)$ imposes that b_j is already computed and in memory elsewhere, not in the m we have left to compute b_i. I will show how this definition is not consistent with our memory analysis in Section 2.4.3 that allows us to free b_j as soon as b_{j-1} is computed, but instead forces us to wait until b_i has been computed.

Consider Figure 3.3, which visualises the unrolling of a couple recursive calls of $C(i, j, m)$.

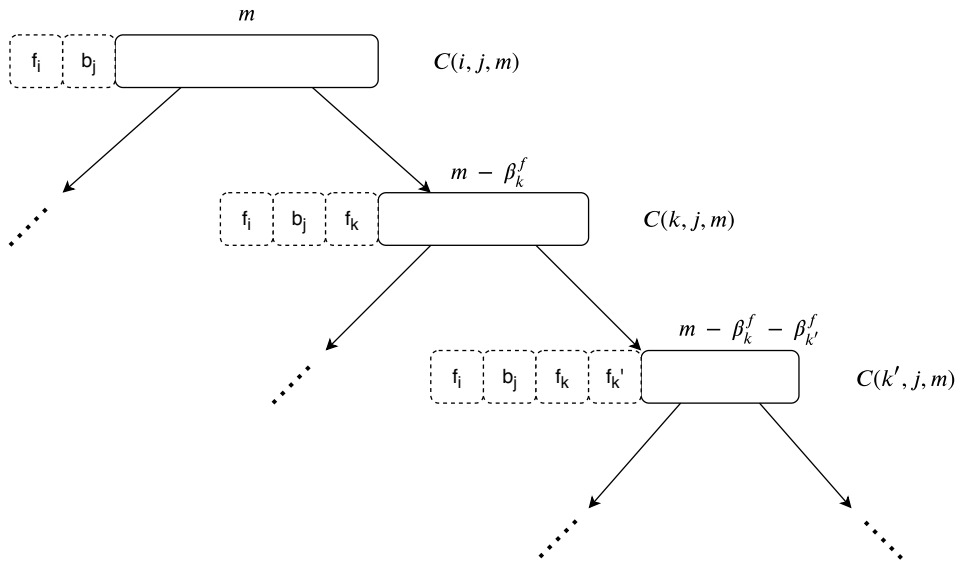


Figure 3.3: The unrolling of two recursive calls into the right subproblems of $C(i, j, m)$.

As per the definition of $C(\cdot)$, we start with f_i and b_j already allocated in memory, and m memory left to work with to solve the problem. We pick some checkpoint f_k and recurse on the right. At the level of the subcall, f_k is defined as 'already' in memory, separate from the $m - \beta_k^f$ we have left to work with. Once again, we pick some $f_{k'}$ and recurse on the right. Now, $f_{k'}$ is already in memory, and we have $m - \beta_k^f - \beta_{k'}^f$ to work with.

As we recurse further right, shown in Figure 3.4, this memory will get used up as we place more checkpoints, until eventually we place f_{j-1}, at which point we can start the backwards pass. We compute b_{j-1} from f_j and b_j. As shown in our analysis in Section 2.4.3, we can now free f_{j-1} and b_j. As we proceed backwards freeing memory, eventually our subcall $C(k', j, m - \beta_k^f - \beta_{k'}^f)$ finishes with only the computed $b_{k'}$ sitting in the $m - \beta_k^f - \beta_{k'}^f$ memory.

However, notice that b_j was considered already allocated in memory elsewhere, separate to our workspace of size $m - \beta_k^f - \beta_{k'}^f$; as we proceeded backwards, we had no control over whether b_j was freed so could not reclaim its memory and reuse it. The problem is that a much further up supercall had fixed the allocation of b_j, and

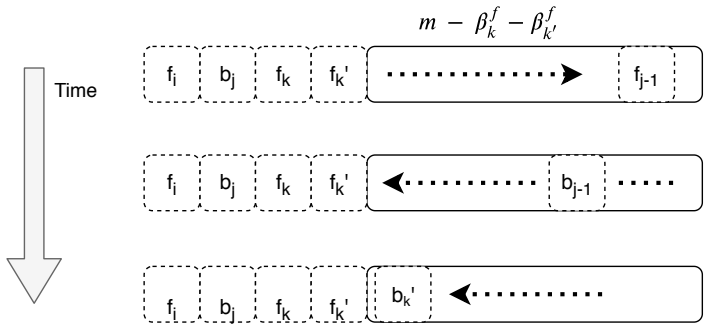


Figure 3.4: Moving 'right' until we finally reach f_{j-1}, at which point we can move left, freeing tensors until only the ouput $b_{k'}$ remains.

only when it returns is b_j freed. This is caused by defining $C(\cdot)$ to say that b_j is already allocated outside of the m memory remaining, fixing its placement 'early' in the supercall, and so it is not until the supercall returns that eventually it will be freed.

Instead, we want to delegate responsibility of placing and freeing b_j down the subcalls, until eventually a subcall performs the backward step and can free it, reclaiming the memory. Otherwise, when we start with $C(0, N+1, m)$, the targets b_{N+1} will be kept allocated until the entire call has finished and we have gone all the way left to b_0, rather than being the first backward at the end to get freed. Each time we recurse on the left, the b_j for that call will also not be finished until we have proceeded all the way left to the b_i for that call. Clearly, this definition of $C(\cdot)$ has totally broken our memory analysis.

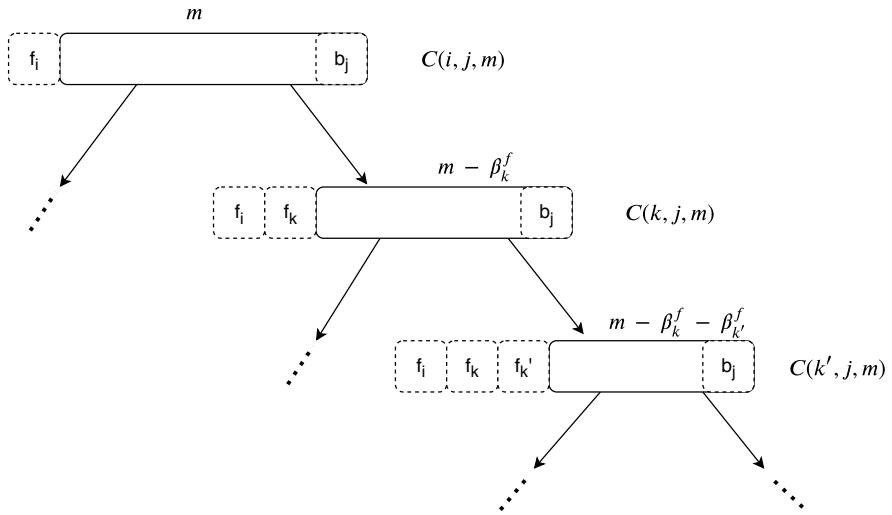


Figure 3.5: How b_j is now placed within m, so the subcalls take ownership of placing it.

Therefore, we redefine $C(i, j, m)$ to mean that f_i is computed and already in memory, and b_j is computed, *but we are responsible for placing b_j within our m memory and freeing it*. This way, as we recurse down the subcalls, say to the right, we pass 'ownership' of b_j to that subcall, which will place b_j within its own $m - \beta_k^f$ memory

and then free it as soon as possible. This is visualised in Figure 3.5.

Now, as we keep recursing further right placing checkpoints, when we eventually reach f_{j-1}, the subcall we are in can compute the backward step for b_{j-1} and free b_j. As the backward pass proceeds left, the backwards will now actually be freed, until we have gone back up to the $C(k', j, m - \beta_k^f - \beta_{k'}^f)$ call and $b_{k'}$ is the only tensor left. This is shown in Figure 3.6.

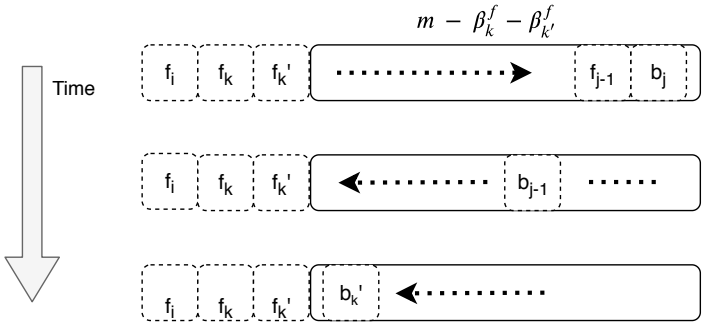


Figure 3.6: Demonstrating how the backwards will now actually be freed during the backwards pass

This new definition of $C(\cdot)$ of course affects our analysis of peak memory for $Q(i, j, k, m)$ in Equation 3.2. We must take into account the fact the b_j is using part of the m memory. Thus, the first stage where we compute the forwards to f_k must be done within $m - \beta_j^b$ memory. Then, we recurse on the right with the same $m - \beta_k^f$ memory as before, but now the definition of the subcall is that it must also place b_j within that memory. By the time the subcall has finished, b_j will be long gone and b_k will be all that remains in its $m - \beta_k^f$ memory. We then recurse on the left with the same m memory as before; the left subcall is now responsible for placing b_k within that m memory (and then freeing it), rather than saying b_k is in some separate memory to m whilst the left subcall proceeds.

The peak memory across the three stages is visualised Figure 3.7. Specifically, we update the peak memory of $Q(i, j, k, m)$ as established in Equation 3.2 to the following:

$$
\begin{aligned}
&\max(\\
&\quad \beta_j^b + \beta_{i+1}^f, \\
&\quad \beta_j^b + \max_{i+1<l\leq k} \beta_{l-1}^f + \beta_l^f, \\
&\quad \beta_k^f + B(k, j, m - \beta_k^f), \\
&\quad B(i, k, m) \\
&)
\end{aligned}
\tag{3.3}
$$

As before, we will have to check each of these stages for failure before evaluating $Q(\cdot)$.

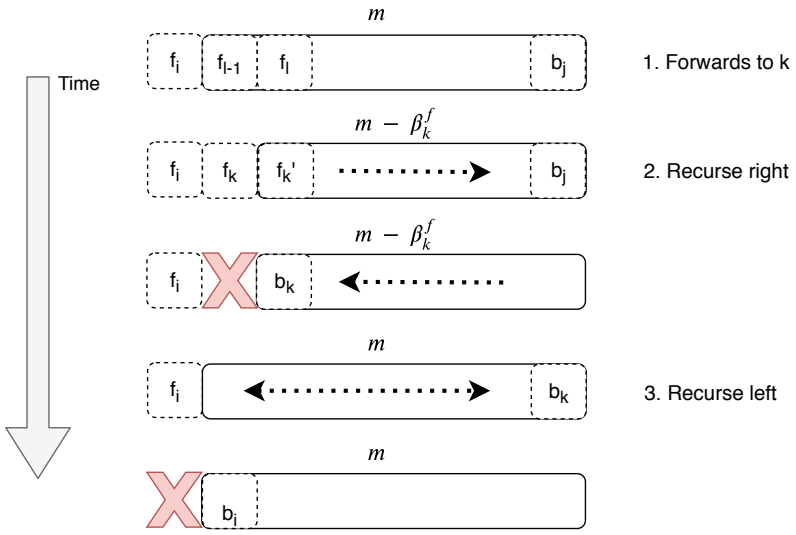


Figure 3.7: How peak memory occurs across the three stages of $Q(\cdot)$. The red cross denotes the memory being freed by the supercall before recursing left.

3.1.7 The New Algorithm So Far

In the above sections, I have accounted for per-layer costs; how to traverse the subproblem space; introduced $B(\cdot)$; accounted for failure; and changed how the problem is modelled so the backwards actually get freed during the backward pass.

Let us take a breather from analysing the naunces of the problem, and instead consolidate the above changes into the new algorithm *thus far*, shown in Algorithm 3.1.

There are a still a few minor details to note. First, whereas Gruslys et al. defined $y = D(i, j, m)$ as the *offset* to the next layer to checkpoint; I use $k = D(i, j, m)$ as the *index* of the forward tensor.

Second, according to our definition of $C(i, j, m)$, the m memory budget does not include f_i but does include b_j. This means the largest subproblem on $(0, N+1)$ must be solved within $M - \beta^f_0$ memory, not M, as there must be enough space remaining in the original M to store f_0 and separately compute $C(0, N+1, m-\beta^f_0)$.

Also, α^f_{N+1} and β^f_{N+1} are not actually defined costs as f_{N+1} does not exist, but since α^b_{N+1} and β^b_{N+1} are defined, I make the second dimension of their arrays $N+2$. I make a similar simplification for the DP tables B, C, D.

Lastly, as arrays start at 0, but memory starts at 1, I will index B, C, D with $m-1$ when referring to m memory.

Algorithm 3.1: The new policy solver thus far, incorporating the changes from the above sections.

Input: N the sequence length
Input: M the memory budget
Input: α a $2 \times (N+2)$ `float` array of the per-operator compute costs
Input: β a $2 \times (N+2)$ `int` array of the per-operator memory costs

$M \leftarrow M - \beta_0^f$
Let B, C, D be 3D arrays of size $(N+1) \times (N+1) \times M$
Base cases will go here.
# Traverse the subproblems bottom-up.
for $m \leftarrow 2$ **to** M **do**
  for $i \leftarrow N$ **to** 0 **do**
    # Base case has already handled $(i, i+1)$, so start at $i+2$.
    for $j \leftarrow i+2$ **to** $N+1$ **do**
      $C_{min} \leftarrow \infty$
      failed $\leftarrow \top$
      for $k \leftarrow i+1$ **to** $j-1$ **do**
        # Pre-check the stages of this strategy for failure.
        # 1: Forwards to f_k in-place whilst holding b_j.
        $\text{peak}_\text{F} \leftarrow \beta_j^b + \max(f_{i+1}, \max_{i<l<k} \beta_l^f + \beta_{l+1}^f)$
        if $\text{peak}_\text{F} > m$ **then**
          continue
        # 2, 3: Right and left subproblems
        if $m - \beta_k^f < 0$ **then**
          continue
        $\text{peak}_\text{R} \leftarrow B[k][j][m-\beta_k^f-1]$
        $\text{peak}_\text{L} \leftarrow B[i][j][m-1]$
        if peak_R *or* peak_L *failed* **then**
          continue
        $\text{peak} \leftarrow \max(\text{peak}_\text{F}, \beta_k^f + \text{peak}_\text{R}, \text{peak}_\text{L})$
        if $\text{peak} > m$ **then**
          continue
        # Success, a strategy works!
        failed $\leftarrow \bot$
        $c \leftarrow \sum_{l=i+1}^{k} \alpha_l^f + C[k][j][m-\beta_k^f-1] + C[i][k][m-1]$
        if $c < C_{min}$ **then**
          $C_{min} \leftarrow c$
          $B[i][j][m-1] \leftarrow \text{peak}$
          $C[i][j][m-1] \leftarrow c$
          $D[i][j][m-1] \leftarrow k$
      if failed **then**
        $B[i][j][m-1] \leftarrow$ *failure*

3.1.8 More Memoisation of Costs

Now that we have the algorithm concretely written down, we can see further opportunity for memoisation.

First, as we go over the k loop, we need the cost of computing $f_{i+1} \to f_k$. Obviously, we can accumulate this over the loop, rather than summing from f_{i+1} each time.

We could memoise this fully over all possible subsequences, so when we move j right we already have the costs to $j-1$, and when we move i left, we already have the costs from $i+1$. However, I do not think the extra effort is worth it. The extra code to memoise over just the k loop is so little compared to the entire loop body that it hardly adds any cost, so further savings from full memoisation will not have much effect.

Likewise, across the k loop, we can memoise the peak memory of computing the forwards to f_k; and likewise, I do not think it is worth 'fully' memoising this.

3.1.9 Failure Propagation and Short-Circuiting

In this section, I consider how failure propagates 'up' through the search space. If some i, j, m fails - as in even when using the constant memory strategy we cannot run the subsequence - then we definitely cannot do any sequence of which this is a subsequence. That is, all i', j' where $i' \leq i$ and $j' \geq j$ must fail too.

We can propagate this failure right through j easily, as it is the inner loop. When we encounter the failed subsequence, we short-circuit and set all the i, j' to failure, then break from the j loop.

Propagating left through i is more complex, as it is the outer loop. If the failure occurred on $(i_\mathrm{F}, j_\mathrm{F})$ We need to remove from the (i, j) space iterated over by the loops all (i', j_F) for $i' \leq i_\mathrm{F}$. To clarify, I do not mean storing the j_F at which we failed, then for each $i' \leq i$ we iterate to, doing the js up to j_F, setting the rest to failure, and breaking from the loop. I mean actually removing those elements from the search space altogether, so we never iterate to them, like we elide the j by breaking out of the j loop.

I am sure any solution would greatly increase the code complexity and so even make it slower. I do not even bother with the stated approach of storing j_F and immediately short-circuiting when we hit it, as the algorithm will only have to run that one extra iteration for j_F before realising failure and setting all $j' \geq j_F$ to failure.

3.1.10 Skipping the Small m

If m_{min} is the absolute minimum memory required to run the network (by using the 'recompute everything' strategy), then every (i, j, m) subproblem for $m \leq m_{min}$ is guaranteed to fail. Considering m_{min} could be in the hundreds or thousands, and that this is $N^2 m_{min}$ subproblems, this is a considerable amount of unnecessary work. We can easily elide it by pre-calculating m_{min} and only starting the m loop from there, rather than 1. In the section on base cases, I explain how m_{min} is calculated. It corresponds to the peak memory cost of the 'recompute everything' strategy on the entire sequence.

3.1.11 Base Cases

Finally, we consider the base cases of the algorithm. Previously, in Algorithm 2.1, these were when the sequence length t was 1; when the number of memory slots m was 1, so we employed the constant space strategy; and when $m \geq t$, so we employed the no recomputation strategy. In this section, I show how they generalise under arbitrary-costs.

Unit Sequence Length

We shall first address the unit sequence length case. It corresponds to $j = i+1$. For the problem $C(i, i+1, m)$, we are given f_i and b_{i+1}, and simply need to run the backward operator to get b_i. Thus, the computational cost is $C(i, i+1, m) = \alpha_i^b$. Recall the definition of the subproblem means it is responsible for placing b_{i+1} within its memory, as well as of course b_i. This makes the peak memory cost $B(i, i+1, m) = \beta_i^b + \beta_{i+1}^b$. Also, we need to check for failure (that this memory cost is less than m) before actually setting $C(\cdot)$ and $B(\cdot)$. If failure did occur, then, like explained above, we set all (i, j') where $j' \geq i+1$ to failure, then skip over the rest of the j loop. Finally, we do not need to explcitly set the policy $D(\cdot)$ as the executor function will have a base case for $j = i+1$ that performs the backward step.

Given that we have already set up the loops to traverse *all* subproblems bottom-up, we do not need a separate loop beforehand for initialising $C(\cdot)$ with this base case, like in the original Algorithm 2.1 for RNNs. In fact, looking at our new Algorithm 3.1, we can see that, before the j loop starts, initialised to $j = i+2$, we can handle the $j = i+1$ base case first.

The "No Recomputation Strategy"

When there is enough memory, we do not need to trade-off any compute for memory. Recall from Equation 2.19 the peak memory of running backpropagation on a

sequence:

$$\max_{0 \leq i \leq N} \sum_{l=0}^{i} \beta_l^f + \beta_{i+1}^b + \beta_i^b$$

Figure 3.8 visualises some possibilities of what this looks like.

The base case would be triggered for any subsequence we encounter where this expression is within m. That would be quite hard to memoise over and efficiently detect. However, I have carefully crafted the $j = i+1$ base case and the recursive case $Q(\cdot)$ to have already accounted for this base case themselves. The following outlines a proof of this.

First, consider a sequence of size one. The $j = i+1$ base case for this simply computes the single backward operator; it is vacuously true that no recomputation occurs.

Second, consider again the sequence of length greater than one in Figure 3.8. Let us say that its peak memory with no recomputation, represented by either of the coloured lines, is within the memory budget m. To find the optimal $C(i, j, m)$, the algorithm will consider $Q(i, j, k, m)$, for all $i < k < j$. That is, we break the problem down into the left subproblem (i, k, m) and the right subproblem $(k, j, m-\beta_k^f)$. For the optimal strategy of recomputing nothing to arise, we need the algorithm to choose to checkpoint the first forward, $k = i+1$, and for the right subproblem to also checkpoint the first forward, and so on to f_{j-1}; meaning everything was checkpointed and nothing was recomputed. The left subproblem for each of these is the unit length base case so it is vacuously true that they involve no recomputation.

Consider the $k = i+1$ case, where the right subproblem becomes $(i+1, j, m-\beta_{i+1}^f)$. Let c be the optimal computational cost when choosing $k = i+1$. By induction, let us say the optimal solution to this subproblem has correctly been found to be the no recomputation case. Note it does not matter which backward step, shown as the coloured lines in Figure 3.8, actually incurred the peak memory, as either way it is less than the memory budget by the definition of the problem, and has no bearing on whether we can say by induction that the right subproblem will have involved no recomputation.

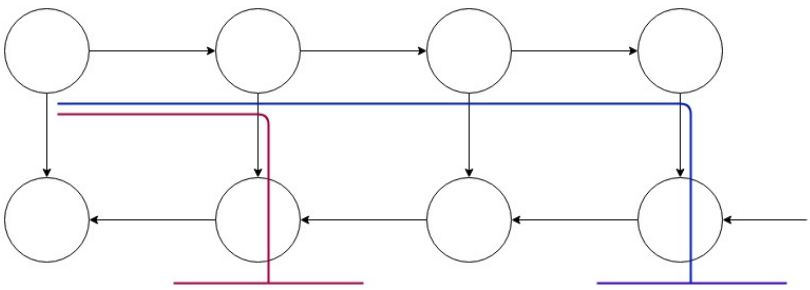


Figure 3.8: Two possibilities for when peak memory occurs. The blue line indicates peak memory occurs at the beginning of the backward pass. The red line indicates when we are computing the backward of the second layer.

So, in the $k = i+1$ case, the optimal cost c becomes the cost to compute $f_i \to f_{i+1}$, plus the cost of the right subproblem, which involved no recomputation, plus the cost of the left subproblem, which will be over a single layer and so involves no recomputation. Thus, choosing this strategy involves no recomputation, so it must be the optimal strategy that is picked by the algorithm, as it is impossible to beat the computational cost of doing no recomputation.

To show this, consider any other k' with optimal cost c'. Say its right and left subproblem involve no recomputation; we get the same optimal cost $c' = c$ and so, regardless of whether the algorithm chooses k or k', it will have found the optimal cost. On the other hand, if either subproblem involved recomputation, then the cost cannot possibly be better than c, so we choose k and get the no recomputation strategy.

Thus, we have (loosely) shown by induction that the algorithm already takes care of the "no recomputation" base case itself, so no further work is required to encode it in.

The "Recompute Everything" Strategy

Let us recap how this strategy works. To perform backpropagation on the sequence (i, j), we start with f_i and b_j, then run the forwards in-place to f_{j-1}. The peak memory so far is the max of the forwards pair-wise across the layers we just ran, whilst holding b_j in memory. Then, we compute $f_{j-1}, b_j \to b_{j-1}$. The peak memory for the entire first step then, is (i) the max of the peak memory of each invididual forward step, whilst holding the upstream backward and (ii) the backward step. Next, we start again at f_i, run the forwards in-place to f_{j-2} whilst holding b_{j-1}, and compute $f_{j-2}, b_{j-1} \to b_{j-2}$. We repeat this until we have computed all the backward steps to b_i.

Thus, the peak memory becomes the max of (i) the max of all the individual backward steps and (ii) for each of those steps, the max of the forward steps to that layer whilst holding the layer's upstream backward. It is essentially the size of the single largest step required to perform backpropagation on the sequence. The precise definition given below makes this more clear. As explained in the previous sections, we will exclude the memory of f_i but include b_j. This cost is $\Theta(1)$ with respect to the number of layers.

$$
\begin{aligned}
&\max(\\
&\quad \max_{i<j'<j} \left(\beta^b_{j'+1} + \max_{i+1<l<j'} \beta^f_{l-1} + \beta^f_l \right), \\
&\quad \max_{i \le l < j} \beta^f_l + \beta^b_{l+1} + \beta^b_l, \\
&)
\end{aligned}
\tag{3.4}
$$

The computational cost of this strategy is $\Theta(N^2)$; first we do $j-i$ forward steps, then $j-i-1$ forward steps, down to a single step. In total, the first forward f_{i+1} will be

computed $j-i-1$ times, the second $j-i-2$ times, and so on, with the last forward f_{j-1} only being computed once on the first step. We do each of the $j-i$ backward steps once. This gives the precise computational cost below. In accordance to the definition of $C(\cdot)$, we do not count the cost of computing f_i or b_j.

$$\sum_{l=1}^{j-i-1} (j-i-l) \cdot \alpha_{i+l}^{f} + \sum_{l=i}^{j-1} \alpha_l^b \tag{3.5}$$

How do we calculate these costs efficiently? We should be able to memoise them across subsequences, rather than computing the sums every time.

However, memoising over i is actually quite hard for this problem. Consider we know the peak memory for some (i, j), then move left to $(i-1, j)$. For the first problem, we considered f_i as already given, so did not take into account its memory cost. For the second problem, we do now consider the memory cost of f_i. Normally, if f_i is part of the step incurring the peak memory cost for $(i-1, j)$, then we would expect this to be reflected in the memoised peak memory cost of (i, j). This will not be the case as the memory of f_i was not included in the (i, j) subproblem. Thus, we cannot easily 'induct' over the subproblems. Instead, similar to before, we will trade off a little bit of speed for reduced code complexity by only memoising over the j loop. Also as before, the extra cost of performing this memoisation is very little compared to the size of the j loop, so there is not much scope for improvement anyway.

We shall first consider the peak memory cost from Equation 3.4. Consider separately the peaks of the backward and forward steps. The backward steps do not sum over any layers; we can trivially memoise the peak across j by memoising the max $\beta_j^f + \beta_{j-1}^b + \beta_j^b$ so far.

The forward steps are a bit more complicated. Let us break down the max expression. The outer expression ranges over j'. Each j' all share the same inner expression - the peak memory of the forward pass - but only up to layer $j'-1$. Thus, as the j loop moves right, we can can memoise the peak of the forward layers separately and use that result in the calculation of the overall peak. That is, the new overall peak becomes the max of (i) the peak of the forwards so far plus this layer's backward and (ii) the backward step for this layer. This is shown in Algorithm 3.2. The conditional on $j-2=i$ is because we do not count the memory of f_i.

Now we look at the compute costs. Equation 3.5 sums over the layers j. There are two separate sums for the forwards and backwards costs. Though it was easier to explain it in that way, we can rethink it to be more suited to memoisation.

Consider we are at some (i, j) subsequence. Say c is the memoised cost of executing the quadratic (no recomputation) strategy on that subsequence. As we move right to $(i, j+1)$, we want to add the marginal cost of performing quadratic on this longer subsequence. The strategy on $(i, j+1)$ does everything the strategy on i, j does, except it has an extra step at the beginning where all the forwards from f_{i+1} to f_{j-1} are calculated, and then b_{j-1} is calculated. This cost can be trivially memoised by

Algorithm 3.2: Memoising the peak memory cost of the "recompute everything" strategy across the j loop.

```
1 ...
2 fpeak_quad ← 0
3 peak_quad ← max(fpeak_quad , β^b_{i+1} + β^b_i)
4 for j ← i+2 to N+1 do
5     fpeak_quad ← max(fpeak_quad , (j−2 = i ? 0 : β^f_{j−2}) + β^f_{j−1})
6     peak_quad ← max(peak_quad , β^b_j + fpeak_quad , β^f_{j−1} + β^b_j + β^b_{j−1}
7     ...
8 ...
```

Algorithm 3.3: Memoising the computational cost of the "recompute everything" strategy across the j loop.

```
1 ...
2 fcompute_quad ← 0
3 compute_quad ← fcompute_quad + α^b_i
4 for j ← i+2 to N+1 do
5     fcompute_quad ← fcompute_quad + α^f_{j−1}
6     compute_quad ← fcompute_quad + α^b_{j−1}
7     ...
8 ...
```

accumulated the marginal cost as we move right over the layers. However, it involves summing the forward costs from f_{i+1} to f_{j-1} every time. To avoid this, we separately memoise the cost of a single pass over the forwards. The memoisation is demonstrated in Algorithmm 3.3.

Removing the Separate Base Case Loop

Lastly, I will show that, unlike the original policy solver in Algorithm 2.1, there is no need to separately initialise $C(\cdot)$ with the base cases prior to the main loop.

Recall when we discussed the $j = i+1$ base case. We concluded that it can be done inside the main loop, as it traverses the subproblems in order, including that base case. It is the only 'real' base case.

The second one we looked at, when no recomputation is required, turned out to no longer be a separate base case.

The last base case of recomputing everything is not something that for any (i, j, m) subsequence we trivially know how to set. It requires knowing the costs across the layers, which can be memoised as above. We do this as part of the main loop over j. Then, *only if the recursive case failed to find a viable strategy* - that is, every action k failed - do we resort to this 'base case'.

Therefore, there is no longer a separate loop at the beginning to set the trivial base cases; everything is done within the main loop body.

3.1.12 The Final Policy Solver Algorithm

Finally, we are done. The new policy solver algorithm, with all of the above rigmarole, is given in Algorithm 3.4.

Algorithm 3.4: The final policy solver algorithm for abritrary per-layer compute and memory costs.

Input: N the sequence length
Input: M the memory budget
Input: α a $2 \times (N+2)$ `float` array of the per-operator compute costs
Input: β a $2 \times (N+2)$ `int` array of the per-operator memory costs

$M \leftarrow M - \beta_0^f$
Let B, C, D be 3D arrays of size $(N+1) \times (N+1) \times M$
# Traverse the subproblems bottom-up.
for $m \leftarrow 2$ **to** M **do**
  for $i \leftarrow N$ **to** 0 **do**
    # Base Case: $[i, i+1, , m-1]$
    $b \leftarrow \beta_i^b + \beta_{i+1}^b$
    $c \leftarrow \beta_i^b$
    if $b > m$ **then**
      $B[i][j][m-1] \leftarrow$ failure, $\quad \forall j \in [i+1, N+1]$
      continue
    $B[i][i+1][m-1] \leftarrow b$
    $C[i][i+1][m-1] \leftarrow c$
    # Recursive Case for each j to the right.
    # Set up memoisation of quadratic strategy costs.
    $\mathsf{fcompute}_{\mathsf{quad}} \leftarrow 0$
    $\mathsf{compute}_{\mathsf{quad}} \leftarrow \mathsf{fcompute}_{\mathsf{quad}} + c$
    $\mathsf{fpeak}_{\mathsf{quad}} \leftarrow 0$
    $\mathsf{peak}_{\mathsf{quad}} \leftarrow \max(\mathsf{fpeak}_{\mathsf{quad}}, b)$
    for $j \leftarrow i+2$ **to** $N+1$ **do**
      # Update quadratic costs.
      $\mathsf{fcompute}_{\mathsf{quad}} \leftarrow \mathsf{fcompute}_{\mathsf{quad}} + \alpha_{j-1}^f$
      $\mathsf{compute}_{\mathsf{quad}} \leftarrow \mathsf{fcompute}_{\mathsf{quad}} + \alpha_{j-1}^b$
      $\mathsf{fpeak}_{\mathsf{quad}} \leftarrow \max(\mathsf{fpeak}_{\mathsf{quad}}, (j-2 = i \,?\, 0 : \beta_{j-2}^f) + \beta_{j-1}^f)$
      $\mathsf{peak}_{\mathsf{quad}} \leftarrow \max(\mathsf{peak}_{\mathsf{quad}}, \beta_j^b + \mathsf{fpeak}_{\mathsf{quad}}, \beta_{j-1}^f + \beta_j^b + \beta_{j-1}^b$
      # Choose optimal checkpoint k. Initialise variables for its loop.
      $\mathsf{fpeak}_{\mathsf{k}} \leftarrow -\infty$
      $\mathsf{fcompute}_{\mathsf{k}} \leftarrow 0$
      $C_{min} \leftarrow \infty$
      $\mathsf{failed} \leftarrow \top$

```
31
32
33
34      for k ← i + 1 to j − 1 do
35          # Update costs over k.
36          fcompute_k ← fcompute_k + α_k^f
37          fpeak_k ← max(fpeak_k , (k−1 = i ? 0 : β_{k−1}^f) + β_k^b)
38          # Pre-check the stages of this strategy for failure.
39          # 1: Forwards to f_k in-place whilst holding b_j.
40          peak_F ← β_j^b + max(f_{i+1}, max_{i<l<k} β_l^f + β_{l+1}^f)
41          if peak_F > m then
42              continue
43          # 2, 3: Right and left subproblems
44          if m − β_k^f < 0 then
45              continue
46          peak_R ← B[k][j][m − β_k^f − 1]
47          peak_L ← B[i][j][m − 1]
48          if peak_R or peak_L failed then
49              continue
50          peak ← max(peak_F , β_k^f + peak_R , peak_L )
51          if peak > m then
52              continue
53          # Success, a strategy works!
54          failed ← ⊥
55          c ← Σ_{l=i+1}^{k} α_l^f + C[k][j][m − β_k^f − 1] + C[i][k][m − 1]
56          if c < C_min then
57              C_min ← c
58              B[i][j][m − 1] ← peak
59              C[i][j][m − 1] ← c
60              D[i][j][m − 1] ← k
61      # All possible checkpointing strategies failed.
62      # Resort to quadratic. If even that fails, fail this subproblem.
63      if failed then
64          if peak_quad > m then
65              B[i][j′][m − 1] ← failure,   ∀j′ ∈ [j, N + 1]
66              break
67          B[i][j][m − 1] ← peak_quad
68          C[i][j][m − 1] ← compute_quad
69          D[i][j][m − 1] ← quadratic strategy
70  return (C, D)
```

3.2 Implementation in PyTorch

In this section, I present my implementation in PyTorch of dynamic programming checkpointing with precise per-layer costs. It is a standard Python 3 package that can be installed with `pip`. However, it is not published on PyPI, so you must download the source and install it locally. This is done as in Listing 3.1.

```
git clone https://github.com/shirazb/pytorch-autograd-checkpointing.git
pip install pytorch-autograd-checkpointing
```

Listing 3.1: Installing the library.

3.2.1 Overview of API

The library presents a helper class `SequentialCheckpointer`. An overview of the API is given in Listing 3.2. The class takes a sequence, profiles it, solves for the optimal policy, and executes the sequence according to the policy.

```
sequence = # ... a torch.nn.Sequential or a list of layers

chkpter = SequentialCheckpointer(sequence)

chkpter.profile_sequence(dummy_input, dummy_upstream_grad)
policy = chkpter.solve_optimal_policy(memory_budget)

# Inside training loop
downstream_grad = chkpter.backprop_sequence(policy, x, upstream_grad)
```

Listing 3.2: Overview of the `SequentialCheckpointer` API.

Noe that, if the checkpointed sequence is not a sub-model of some larger model, meaning there are no more layers after it, then the 'upstream gradient' is simply 1 (recall that reverse-mode automatic differentiation is initialised with the gradient of the output with itself, which is 1).

For greater flexibility, the profiling stage can be done separately to the policy solver stage. Alternatively, you can tell the solver to use uniform costs or user-defined costs, shown in Listing 3.3. This can be done for both compute and memory, or only one of them. Similarly, for enhanced flexibility, I have chosen to make the budget and policy not a part of the class' state. This way, the user can solve for many policies using the same profiling results.

```
# Will profile then solve
policy = chkpter.solve_optimal_policy(M)

# Will reuse the profiling results
policy2 = chkpter.solve_optimal_policy(another_M)

# Will use uniform costs
policy3 = chkpter.solve_optimal_policy(M, profile_mem=False, profile_comp=False)

# Will profile compute and use given memory costs
policy4 = chkpter.solve_optimal_policy(M, profile_mem=False, memory_costs=costs)
```

Listing 3.3: Solving with uniform or given costs.

The API is very specific about where it expects input tensors to be in memory. This is because the solver is very specific about what is in memory, in order to best utilise it.

If the last layer of an N layer sequence is the loss, then recall that (b_{N+1} are the targets. The solver expects this to be freed as the backwards pass proceeds. Usually though, the user has a handle to the GPU copy of the tensors in the training loop, meaning the targets will not be freed. Thus, the user must make a custom loss layer that moves the targets to the device only when needed. This is trivial to do. An example is given in Listing 3.4.

```
class LossLayer(torch.nn.Module):
    def __init__(self, loss, targets_cpu, device):
        self.loss = loss
        self.targets_cpu = targets_cpu
        self.device = device

    def forward(self, x):
        return self.loss(x, self.targets_cpu.to(self.device))
```

Listing 3.4: A loss layer that moves the targets to the device.

Secondly, the profiler expects the model to be on the device, but the inputs and upstream gradient to be on the CPU, so it can individually move layers to the device and profile them. The model itself (which includes the parameter memory) will always be in memory during execution. I choose to assume the user has already put the model on the device as it is common in PyTorch for users to move the model in-line as it is created, for example `create_model().to(device)`.

On the other hand, the executor expects everything to already be on the device. This is as it naturally should be though, as during the training loop everything will be on

the device already. Expecting them on the CPU would require the user to move them back to the CPU only for us to move it back.

Therefore, given the above, I do not think it is particularly ugly or paritcularly restrictive for the user to adhere to these assumptions.

3.2.2 Translating our Graph to PyTorch

The computational graph of backpropagation derived in Section 2.4.3, and the policy solver for it in Section 3.1 do not perfectly translate to PyTorch. I provided them as they are to give a more generic approach that can be specialised to any framework, not just PyTorch. The original computational graph is repeated here in Figure 2.17.

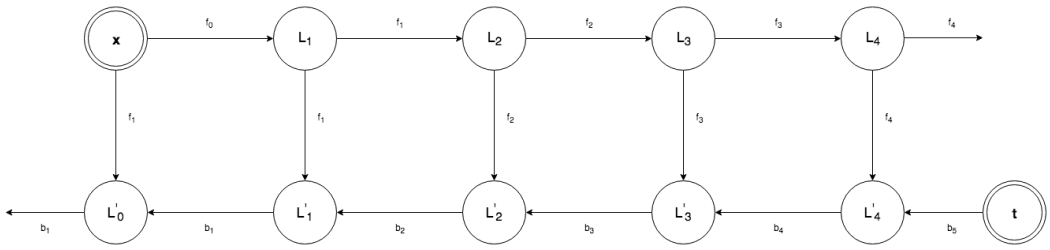


Figure 3.9: The computational graph of backpropagation that we have defined checkpointing on.

Consider my definition of the backward operator L'_k. It performs two operations: (i) computes the gradient of the parameters of the upstream layer using b_{k+1} and f_k; and (ii) computes its downstream gradient using b_{k+1} and f_k. This means a backward operation computes its own downstream gradient, but the gradient with respect to its parameters is computed by the downstream layer.

This definition is not compatible with PyTorch. A layer is known as a module, which contains parameters. Running its forward function invokes an Autograd function on the parameters and the input to produce some output. Thus, when the layer's backward function is called, it will call the backward function of the parameters too (as they require gradient). It must be the case, then, that the layer's forward function saved its input as well as its output, as both are required to compute the gradient of the parameters. This of course breaks the original graph, in which only the output of a layer goes to its backward function, meaning the assumptions baked into the policy solver about what memory is currently allocated do not hold. Specifically, in PyTorch, the output of a layer f_k, which feeds its backward function L'_k to produce b_k, will actually include the input f_{k-1} too.

To work around this, I simply redefine the forward tensors to include both the input and output, and the backward operators to compute the gradients of their own parameters. The psuedo-tensor b_0 is no longer required to represent the calculation of the first layer's parameters. Instead, we now have $N+1$ backward tensors, from b_1 to b_{N+1}. As before, there are $N+1$ forward tensors, from f_0 to f_N.

As this is what PyTorch is actually doing, redefining the forwards like this should not cause any major issues. However, I have to amend the solver to be compatible with this. The memory cost of computing the output of layer k is now just β_k^f, not $\beta_{k-1}^f + \beta_k^f$. When we checkpoint f_k and recurse on the right, we are modelling that both the input and output have been checkpointed. According to our original computational graph, this is suboptimal, but this is what PyTorch is doing, so I model it as such. Changing this would require a fundamental change to Autograd. Additionally, as the solver expects a b_0, rather than change the solver, I simply pass in a dummy value with zero costs.

3.2.3 Imposing the Policy on a Network Through Autograd

We need to be able to execute a sequence according to a given policy. We can impose the dropping and recomputing of forwards in PyTorch through Autograd.

There is already a `torch.utils.CheckpointFunction` class in PyTorch. It is an Autograd function that wraps the layer whose output should be recomputed. In the forward function, it will save the input for the backward pass and run the layer with no grad, returning the output. As the output was computed with no grad and was not saved for the backward pass, it will thus be freed. In the backward function, the output is recomputed from the saved input, before running the backward function of the layer using the output.

We can use this function to encode 'drops' into a list of layers. That is, we create a `Drop` module that takes a child module and, in the forward, applies `torch.utils.CheckpointFunction(x)` to implement recomputation. We can then implement multiple recomputations by nesting `Drop` modules. A `Drop` nested r times will have its forward called $r+1$ times due to the r outer `Drop`s being recomputed, which invoke this `Drop`s forward again. That is, the outer most `Drop`'s backward will recompute its child `Drop` and call its backward, which will recompute its child `Drop` and call its backward, and so on. We would also need a `recursion_depth` parameter. We initialise it to r, then every time the forward of `Drop` is called, it decrements `recursion_depth` before invoking the checkpoint function. We amend `torch.utils.CheckpointFunction` to also take the `recursion_depth` parameter and only save the input when this is 0. This avoids the input being checkpointined repeatedly.

This solution allows the user to define a sequential model with `Drop`s encoded, run it normally, and `output.backward()` will handle all the recomputations naturally, as we have implemented the recomputation in Autograd. The policy found by the solver can then be traversed to automatically encode the drops in the sequence for the user. However, despite the elegance of this solution, it is actually fundamentally incompatible with our formulation of checkpointing. The reason is that it checkpoints *inputs* of layers and recomputes outputs, rather than checkpointing outputs, which is how the checkpointing policy was defined.

Consider, for a sequence (i, j), the policy tells us to checkpoint k. We wrap the k^{th} layer in a `Drop`. We want to checkpoint its output f_k. Wrapping the k^{th} layer means we actually checkpoint its input, f_{k-1}. This is not what was intended. Say we instead wrap the $k+1^{\text{th}}$ layer, we would checkpoint f_k correctly, but force f_{k+1} to be recomputed. The problem is that by default forwards are kept in memory and `Drop` marks forwards to be dropped; whereas the semantics of the policy solver is that all outputs are considered dropped unless the solver chooses to checkpoint them.

This fundamental difference could not be reconciled and meant that this approach had to be abandoned.

3.2.4 Manually Imposing the Policy on a Network

Instead, we must create an executor function similar to the one used in Grusyls et al. [2, Algorithm 2] that manually checkpoints tensors and manually controls the backward pass, rather than leaving it to Autograd with `.backward()`. The (adapted) psuedocode is given in Algorithm 3.5.

The main PyTorch-specific consideration here is that we run the forwards to f_k with gradient tracking disabled, meaning the intermediate forwards will not be stored for the backwards pass. We then re-enable gradients afterwards so following computations will be tracked. We detach f_k because, otherwise, in the base case of the right subcall, when `.backward()` is called, it would attempt to propagate further back than layer k. We want to 'interrupt' the backward pass at k so we can run the left hand side according to our policy.

However, this algorithm does not work. There are a number of issues.

First, the stack frame of the current function holds a strong reference to b_j whilst the subcall proceeds right. By the end of the subcall, we expect everything to the right of b_k to be freed, including b_j. However, the strong reference on our stack frame will prevent this. The same problem happens when we recurse on the left. I experimented with using Python's `weakref` library to hold only weak references from the stack frame of these recursive calls, so the only strong reference is from f_j`.grad`. However, this did not work as, for reasons as yet unknown, `weakref` did not count that reference so would deallocate the gradient tensor.

Also, the base case as written does not work. Because of the above, the invariant is that f_i will be detached. This means it will not have a backward graph though, so we cannot compute its backward. Instead, we can remove that base case and instead have a $j = i+2$ base case. It computes the forward f_{i+1} before running the backward. The recursive case now needs to manually check for the unit length base, and, if so, compute the forward and run the backward there, rather than recursing.

However, I unfortunately could not resolve the `weakref` problem. I do have a simulator though. Given a policy and the precise costs, it will simulate execution according

Algorithm 3.5: Executes a network according to the given policy. Adapted from Grusyls et al. [2, Algorithm 2]. `Execute_Strategy_Quad` is not shown for brevity.

```
1  Function Execute_Strategy(i, j, m, f_i, b_j, β, D) is
       // Base Case
2      if j = i + 1 then
3          run the backward operator of layer i
4          b_i ← f_i.grad
5          return b_i
6      k ← D[i][j][m − 1]
7      if k = quadratic strategy then
8          b_i ← Execute_Strategy_Quad(i, j, m, f_i, b_j, β)
9      with torch.no_grad() do
           // Run the forwards to k without gradient
10         f_k ← f_i
11         for l ∈ [i + 1, k] do
12             f_k ← Layer_i(f_k)
13     f_k._detach()
14     f_k.requires_grad ← true
       // Recurse on the right and left
15     m_R ← m − β_k^f
16     b_k ← Execute_Strategy(k, j, m_R, f_k, b_j, β, D)
17     b_i ← Execute_Strategy(i, k, m, f_i, b_k, β, D)
18     return b_i
```

to the above algorithm and report back the total execution time and peak memory. As we obtain the costs from profiling, these should be quite accurate.

Chapter 4

Experimental Results

In this chapter, I will present and evaluate the results of my experiments. I will show the compute-memory tradeoff that dynamic programming checkpointing provides; profile the execution time of the solver to evaluate its practicality; and demonstrate the efficacy of generalising to arbitrary per-operator costs by measuring the effect of checkpointing when assuming uniform costs, profiling only memory, profiling only compute, and when profiling both memory and compute.

All experiments were run on a `n1-highmem-4` Google Compute Engine instance using the Deep Learning image with the following specs:

- Intel Xeon @ 2.3GHz (specifically family 6, model 63, stepping 0)
- 15GB memory
- NVIDIA Tesla K80
- PyTorch 1.2.0
- CUDA 10.0
- cuDNN 7.6.2

4.1 The Compute-Memory Trade-Off

I demonstrate the compute-memory trade-off by showing for varying memory budget m, what is the optimal computational cost the solver predicts. 'Simulated' in the y axis label refers to the fact that I am measuring what the solver 'thinks', not that I am using the simulator instead of a real implementation. I plot this for sequence lengths $N = 20, 40, 100$, and vary m up to 60 for each. I use a uniform network: the compute and memory costs of every forward and backward tensor is 1.

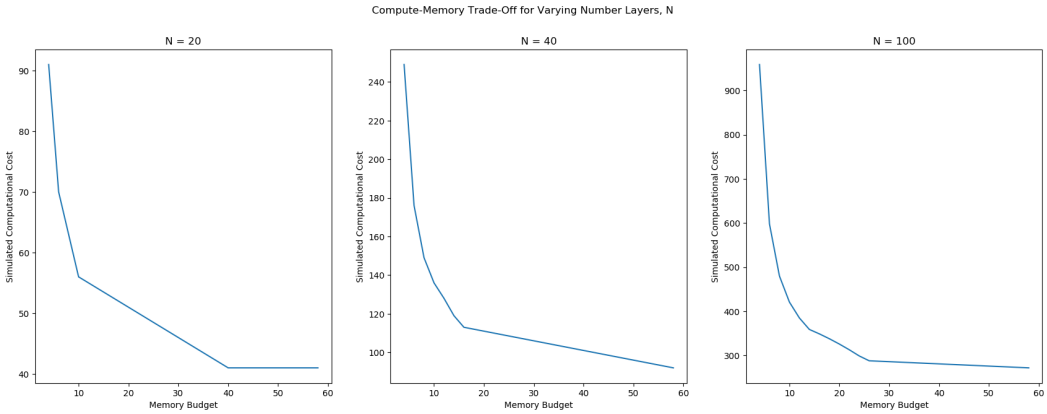


Figure 4.1: The compute-memory trade-off for networks of different sequence length.

Starting from the right of the curve, we can see that compute cost degrades very slowly until m gets too low, then there is a visible 'corner' and it degrades very fast. However, this m is really quite low. The following numbers are all read approximately from the graph.

We can see that for $N = 40$, the cost at the right of the curve, where $m = 60$, starts at just under 100. m decreases all the way to just over 10 before we even see a 30% overhead. I postulate that, for a uniform network like this, the solver is keeping to at most one recomputation until the 'corner'. The costs are consistent with Chen's $\Theta(\sqrt{N})$ analysis for one recomputation.

For $N = 20$, the theoretical optimal memory consumption for uniform layers is $2\sqrt{20} \approx 9$. We can see that it is at about $m = 9$ where the corner is encountered and the cost starts to quickly degrade. This shows, primarly, how powerful even one recomputation is, as that is a signficant cost saving. It also shows that, when given enough memory to not need to do more than one recomputation, the solver finds Chen's theoretical optimal policy. However, Chen's solution simply picks a point on that curve, my solver is much more flexible. It can make use of any extra memory to do less recomputation than the $\Theta(\sqrt{N})$ approach, and, if memory gets too tight for that approach, it will find a policy that still satisfies the memory budget. All of this is done automatically for the user. In the extreme cases, we can see that for $N = 20$, at about $m = 40$ the curve flattens out, because there is now enough memory for *no* recomputation to be required; and on the other end of the graph, though at high cost, we see the solver is also able to satisfy *very* small m. This will be due to it employing the quadratic-compute constant-memory 'recompute everything' strategy.

In addition, the results across the three graphs show how well checkpointing scales to very deep networks. For $N = 20$, reducing the memory budget from about 40 to 10, which is a 60% reduction, gives a 38% overhead. For $N = 100$. reducing from about 60 to 24, which also is a 60% reduction, gives hardly any visible effect. Reducing all the way down to $m = 10$, a huge 83% reduction, is required to observe a similar overhead of 40%.

4.2 Investigating the Affects of Bucketing on the Solver

The policy solver has a continuous memory state, so it must solve the entire problem for $m = 1$ all the way up to the actual memory budget. Given that typical budgets would be in the range of gigabytes, so billions of bytes, that will obviously be intractable. Thus, we must apply some kind of bucketing to memory. Say we choose buckets of size 2MB, we divide the memory budget and the memory costs by 2MB. Each of the tensors must then have their size rounded *up* to the nearest 2MB 'page', causing internal fragmentation. As the atomic unit of memory considered by the solver is now these 2MB pages, it cannot make use of the unused memory. Thus, the solver has lost its true optimality - if that wasted memory could be reclaimed, maybe the solver would have found a faster strategy that uses less recomputation.

In this section, I investigate the repercussions of this in practice. I profiled the solver on a large, state-of-the-art network, DensetNet-121 [19]. Figure 4.2 shows how execution time of the solver and the optimal cost it predicts vary with bucket size.

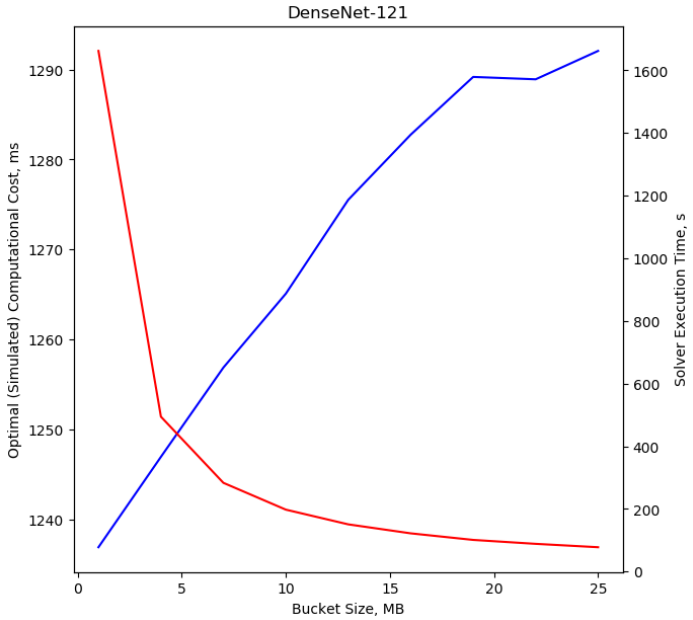


Figure 4.2: Optimal cost (blue) and solver execution time (red) against bucket size, for DenseNet-121.

At the left side of the graph, with a bucket size of just under 1MB, we see that the solver takes 26 minutes to complete. I think this is acceptable for networks like this that take a large number of hours to train. As we increase the bucket size, the execution time drops drastically, before starting to level out. This will be due to the large sequence size starting to dominate the time complexity instead of the memory budget. At a reasonable bucket size of 5MB, the execution time has dropped 75% to 7 minutes. Thus, this increased bucket size is already more than practical, and the corresponding increase in computational cost is little over 1%. I think it is therefore reasonable to conclude that the large memory budget and resulting bucketing has not

greatly inhibited the solver's practicality.

During experimentation, it is quite feasible that researchers may want to run a large model for a short period of time. In that case, even 7 minutes of waiting for the solver could be annoying. Thus, I explored bucketing further. We can see from the graph that, for the very large bucket size of 25MB, the execution time has dropped all the way to under a minute, which should be satisfactory in such a use case. The optimal computational cost has risen from the origin 1235ms to 1900ms, an overhead of 53%. This is still reasonable, especially considering the stated use case was for models that will only be run for a short period of time. Clearly, enabling checkpointing will be far easier, cheaper and quicker for the researcher than implementing distribution to overcome the memory requirement.

4.3 The Efficacy of Using Precise Costs

The initial aim of this section was to investigate the efficacy of precise costs over uniform costs. This would be done by solving for a policy with precise costs and with uniform costs, then imposing both on the network and comparing the resulting compute and memory. To recreate the same problem for the solver but with uniform memory costs, the real costs would be averaged and the same memory budget used. Specifically, I would have showed the difference between (i) profiled compute and memory; (ii) profiled compute and uniform memory; (iii) uniform compute and profiled memory; and (iv) uniform compute and memory. I would individually profile compute and memory so I can more deeply analyse the effects of profiling them on the solver.

However, I unfortunately could not resolve some bugs in the test cases. Thus, I can only show the results of when profiling is turned on for both compute and memory. However, we will see this is enough to make such deductions about profiling versus uniform costs, as the solver behaves in a way that would be impossible with uniform costs.

Figure 4.3 shows the effect of batch size on the compute cost and memory cost of training ResNet-50 [21]. I explored batch size until failure.

We can see that the computational cost and the memory cost increase at a steady linear rate until just under batch size 100. This is because the solver is exploiting the plentiful memory and performing no (or very little) recomputation. Note memory does not stay steady at the max capacity during this phase because the entire capacity is not being used, so memory will scale with batch size despite the same 'recompute nothing' policy being applied that keeps all the network in memory.

After this point, the compute cost increases at a faster rate due to the increased recomputations. From the log messages, I know it is even starting to employ the quadratic strategy. This is an important observation. With uniform costs, the solver would only

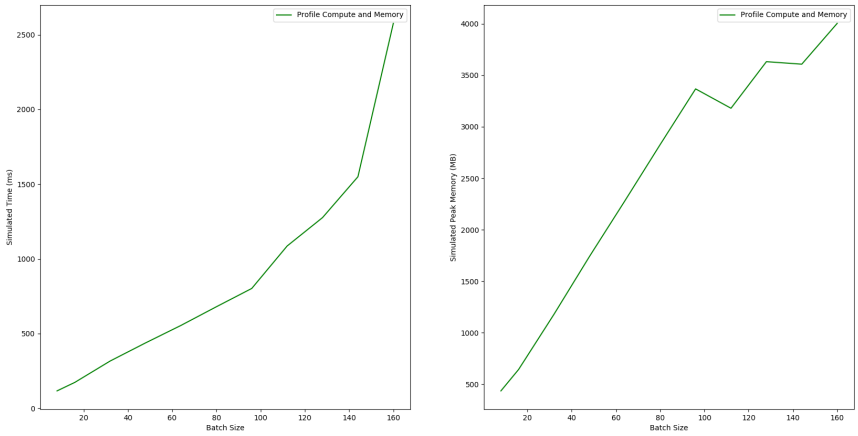


Figure 4.3: Left: The (simulated) computational cost of executing one training step on ResNet-50 according to the precise optimal checkpointing policy. Right: The (simulated) peak memory step of the same operation.

employ this strategy in the absolute extreme case when there is no other way to satisfy the budget. However, given that failure does not happen til a much higher batch size, we can deduce that this is not the case. Thus, the solver has used the exact nature of the per-layer costs to judge that performing quadratic on some subsequences, rather than using fewer recomputations on other subsequences, will actually give a lower compute cost whilst still satisfying the budget. It is impossible for the uniform-cost solver to ever find such a strategy as it simply thinks more recomputations is always more compute. The fact that memory dropped at this point despite the increased batch size shows the extent to which the solver has 'rearranged' its policy. Furthermore, this also demonstrates the flexibility of dynamic programming checkpointing, as for the previous batch size, the solver was making use of extra memory to keep compute down.

Further right from this point, the computational cost and memory cost both increase. The solver is employing quadratic more aggresively, causing the high increase in compute cost. The point where memory slightly decreases signifies that the solver has again 'rearranged' its policy according to the exact costs to further supress the compute cost, and it did it so signficantly that memory actually went down.

After this point, just after batch size 140, the burdern of memory is too high - the solver is forced to perform quadratic on almost the entire sequence and eventually fails. This corresponds to the even steeper increase in compute cost.

4.4 Summary of Results

To summarise, the experiments in this section have showed the following:

- Dynamic programming checkpointing is flexible: it successfully uses as little recomputation as possible to satisfy the budget and, only as the budget decreases, does it gradually incur more compute cost.
- The compute cost degradation is very good until the budget becomes very small, at which point the cost starts to increase drastically due to the many recomputations.
- However, pertinently, this shows the solver is still capable of finding a strategy to satisfy such small budgets.
- The solver is still practical when using bucketing to overcome the large memory budget. That is, the execution time of the solver is made sufficiently quick at little expense to the compute cost of the found optimal strategy.
- The solver can judiciously exploit the varying, precise costs of the layers to find better policies than what a uniform-cost solver can find.

I regret that I was not able to give a proper quantitative evaluation of the precise-cost solver versus a uniform-cost solver. In the future, as well as this, I would like to run experiments on a greater varitey of networks to give more evidence as to the efficacy of the solver.

I would also like to investigate quantitatively how the variations in per-layer costs causes the theoretical benefits of checkpointing to degrade *as compared to if the network was actually of uniform costs*. This analysis could guide users, or future heuristics, as to when checkpointing may be more or less useful versus other optimisations.

Chapter 5

Qualitative Evaluation and Future Work

In this chapter, I will give a qualitative evaluation of my work and suggest directions for further work. I will describe the limitations of my technique and, where possible, suggest ideas for remedying them. I will assess the role of checkpointing in relation to other optimisation techniques and discuss how they can be combined based on their various merits and drawbacks.

5.1 Implementation in PyTorch

I think the API of my PyTorch implementation is not ideal. As described in Section 3.2, it presents a helper function to the user that they must call in the training loop to explictly perform the forwards *and* backwards opertaions. To clarify, I mean the actual Autograd backward functions of the intermediate variables, not `.backward()` on the output tensor. This is completely against the spirit of PyTorch and automatic differentiation; the user should run a forward computation and the backwards should be automatically tracked. To achieve this, the API should be an Autograd function that wraps the user's existing sequence to be checkpointed. This way, they can call the forward functions as usual, interleaved with other non-checkpointed functions, and when they call `.backward()` on the output, the recomputation will automatically take place. I described this in Section 3.2.3.

However, I also described in that section that avoiding this was, to my knowledge, impossible with PyTorch. Implementing multiple recomputations requires manually moving forwards and backwards along the sequence in a way that is not naturally compatible with Autograd.

Another limitation of the proposed approach is that, when the sequence is a sub-model of a larger model, which is very likely the case, some ugly fudging is required

from the user to be compatible with my API. In the forward pass, they must manually detach the output of the sequence to be checkpointed before propagating further forward. This is so the backward propagation can be interrupted and the upstream gradient manually fed by the user into the checkpointing function.

5.2 Limitation of Checkpointing to Sequences

In this thesis, I have worked exclusively on checkpointing *sequential* neural networks; but what about more general graphs? The difficulties of handling these with checkpoinitng is possibly its biggest limitation. In this section, I will show how this limitation arises and evaluate the attempts to tackle it. I will evaluate the role of my contributions with respect to this work and suggest future work to combine their advantages.

In Section 2.1.6, I introduced some of the non-linearities that are prevalent today. For example, consider a ResNet-like architecture [21], made from a sequence of residual units, which contain skip connections, shown in Figure 5.1.

This is almost a sequence. Can we apply checkpointing to this? Consider we checkpoint a square node. Every node to the right of it can indeed be recomputed just from that checkpoint. In effect, it encapsulates all prior computation. However, consider checkpointing one of the circular nodes and dropping everything to the left. The nodes to the right could not be recomputed just from the circular node, as they also require the square node to the left of it; the circular nodes do not encapsulate all prior computation.

To formalise this notion, what we need is a node that is a *graph separator*. Every node on the right of this separator will not depend on any of the nodes on the left, only the separator. That is, the node separates the graph into two *disjoint* subgraphs.

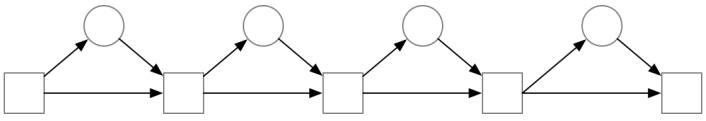


Figure 5.1: The ResNet architecture at a high-level, showing the skip connections. Taken from [8].

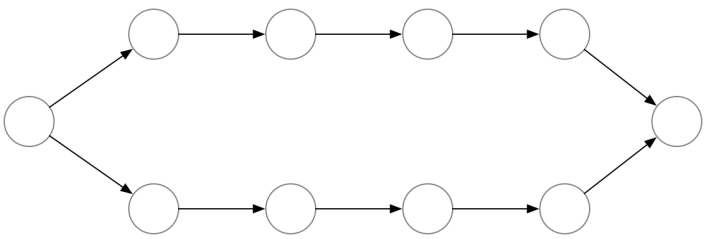


Figure 5.2: A split-merge architecture. Taken from [8].

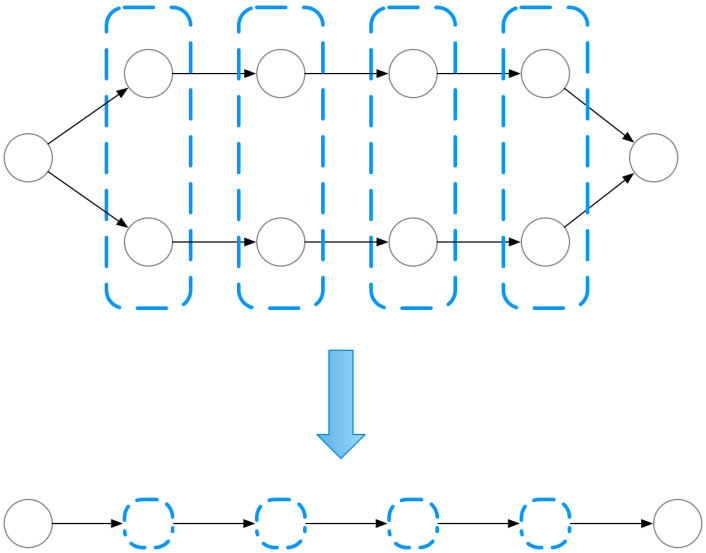


Figure 5.3: Separating sets of a split-merge architecture. The dashed lines indicated the merging of nodes into a bag. Taken from [8].

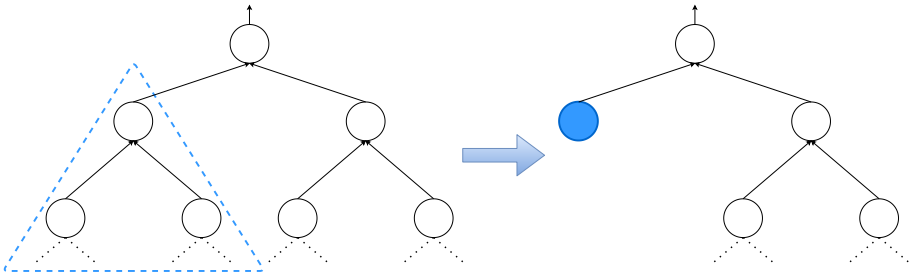


Figure 5.4: Checkpointing on a tree. The blue node on the right 'subsumes' all the computation within the dashed lines on the left.

Consider the split-merge architecture shown in Figure 5.2, such as in GoogleLeNet [20]. Again, not all nodes are viable checkpoints; only the 'split' and 'merge' nodes. The nodes of the parallel branches cannot be checkpointed invidually. However, they can be checkpointed *together*. That is, if we take one node from each branch, this forms a *separating set*, or *bag*, which does split the graph into two disjoint subgraphs. Equivalently, we have merged nodes into bags such that the network at the bag-level is a sequence. This is shown in Figure 5.3

More generally, we can checkpoint any node of a computation *tree*, because any node of the tree is a graph separator - it 'subsumes' all the computation of its subtrees. This is shown in Figure 5.4.

Before we can apply checkpointing to trees, we must solve how to merge nodes of our neural network graphs to form a tree of bags that can be checkpointed at the bag-level. We need to do this such that we minimise *treewidth* - the maximum number of nodes merged into one bag - in order to be able to perform checkpointing as granularly as possible. Ideally, we also want to do this without user annotation.

This is known as a *tree decomposition*. Bodlaender discovered an algorithm that can find the optimal tree decomposition in k-polynomial time, where k bounds the treewidth of the graph and is given by the user [65]. Looking at the above examples of skip connections and split-merges, we can see that this k is small, so the algorithm is efficient. Specifically, for skip connections, the treewidth is three; and for the split-merges it is two. However, given an abitrary network, we cannot just know this - either the user must tell us or we must compute it ourselves, for which no exact polynomial time algorithms exist.

Say that we have found a tree decomposition for the network, we still need to generalise checkpointing to trees. First, it would be useful to perform liveness analysis on the tree, as we did in Section 2.4, which allowed us to free tensors as we proceeded backwards. Liu shows how to generalise Sethi's pebble game to trees [66]. However, once again each node has a uniform cost.

In 2018, Siskind and Pearlmutter [27] generalised Griewank and Walther's `REVOLVE` checkpointing algorithm to trees, using 'divide-and-conquer' checkpointing. However, as far as I can tell, this also does not take into account precise costs. Furthermore, like `REVOLVE` iteself, it is optimal in the sense that it will find the best logarithmic space strategy using multiple recomputations; it will not find the fastest strategy that satisfies a memory budget.

In 2019, there has been some promising work in the Machine Learning community that generalises checkpointing to arbitrary graphs. I have only become aware of this work very recently so did not have time to reformulate my solution to build on top of theirs.

The approach of Feng and Huang [67] can find a checkpointing strategy given an arbitrary graph, without user annotation. They can even handle arbitrary memory costs. However, their algorithm minimises the peak memory cost using one recomputation at most; rather than minimising the compute cost given a memory bound with any number of allowed recopmutations.

Kusumoto, Inoue et al. [68] also generalise checkpointing to arbtirary graphs. They do indeed minimise computational cost given a memory budget. However, all is not perfect. Firstly, like Feng and Huang's approach, they only consider at most one recomputation. Furthermore, the per-layer costs are not truly precise, profiled costs. For example, they assume backward tensors are always twice as large as the forwards, something I found not to be the case in my experiments (sometimes they were even ten times as large). They also model compute costs by saying that convolutional layers are ten times more expensive than all the other layers. Lastly, they report that their exact dynamic programming solution is too inefficient for large networks; but they also give an approximate solution that gives very near-optimal results.

Nonetheless, both of the above papers represent huge progress, as they generalise to arbitrary graphs. However, it is clear that a perfect solution is yet to exist; none of our approaches tick all the boxes. In the future, we need to find a way to combine the merits of the many approaches. In particular for me, I would like to look into

combining the latter approach, that uses dynamic programming on arbtirary graphs to minimise compute given a memory bound, with mine, that does the same for sequences only, but allows multiple recomputations and takes into account the exact per-operator costs.

5.3 Making the Solver More Practical with Reinforcement Learning Techniques

As we saw in Section 3.1, generalising to precise memory costs, rather than using discrete memory slots, caused a lot of headache for the policy solver. We can see why when we view the algorithm in a similar light to Markov Decision Processes, as described in Section 2.5.5. The states of our MDP are all the possible combinations of subsequences and memory budgets. The memory budget, however, is now akin to a continuous state, unlike the discrete memory slots. This makes finding the optimal policy very impractical, as there are far too many states to iterate over. For example, the memory budget on NVIDIA's T4 GPU, designed specifically for deep learning performance, has 16GB of memory [69], which is 16e9 possible states. This is totally intractable, unless we apply some bucketing on memory; but this detracts from the true optimality of the solution and eventually becomes too crude.

The second core issue is that our algorithm attempts to be 'too perfect' - it finds *the* perfect solution, but requires perfect information and a lot of time to do this. I have found that getting this perfect information is not always easy to do in practice. Also, we do not need to spend inordinate amounts of time converging to the true optimum, only something that is close enough.

These issues are at the core of MDPs and Reinforcement Learning, if not the entire point of Reinforcement Learning: when it is too impractical to solve MDPs perfectly using dynamic programming, statistically infer a solution using data. Though the notion of statistical inference does not translate here, I am sure there must be a wealth of techniques we can leverage to cope with the above issues and make the solver more practical. I am not aware of any literature that has attempted to do this. It would definitely be the next step for this project.

An example for dealing with the first issue is to use a constrained MDP [70]. This was suggested to me by Sanket Kamthe, a PhD student at Imperial College London. Rather than modelling memory as a continuous state, we constrain the set of possible actions taken to satisfy the memory budget. That is, we still minimise $Q(i, j, k)$, but only choose k that give a peak memory within the memory constraint m. This is distinctly different from m being a part of the state space, as we no longer have to iterate over every possible m.

5.4 The Role of Checkpointing in Relation to Other Memory Optimisations

In this thesis, I have shown that checkpointing can greatly decrease memory costs at a reasonable overhead. However, other memory optimisations for deep learning exist that are certainly very powerful too. It is therefore important to appraise my work, and checkpointing more generally, in light of these other techniques; evaluating the trade-offs between them and how they can or cannot be combined. I will cover swapping and the choice of convolutional algorithm.

5.4.1 Swapping

Swapping is where forward tensors are swapped out to CPU memory and swapped back in when they are required in the backward pass. In the optimal case, this incurs no compute overhead and constant memory; you can keep everything swapped out except the currently in-use tensors, as long as you can swap the other tensors back in on time. That is, communication has to be hidden under computation. Whether or not we can do this depends on how big a tensor is and how long they take to compute, just like in checkpointing. However, it is actually even more difficult than this. Choosing to swap a tensor consumes the bandwidth of the CPU-GPU link, usually PCIe, for a certain amount of time. This means it is not just about the properties of individual tensors, but that swapping a tensor delays all future swaps, making this a scheduling problem. For example, this could be modelled as a Resource Constrained Scheduling Problem. To quote one textbook [71, p. 23], these problems are "one of the most intractable combinatorial optimization problems" and are NP-hard in the strong sense. Scheduling problems are an entire field of research in their own right, and will surely have a great deal of literature we can leverage in ML.

Many approximate approaches in ML exist [24, 3, 7, 72, 73, 74, 75, 76, 77, 78, 79, 80], which have varying merits and limitations. I will not give a fully survey here. Instead, I will discuss how the technique in general compares to checkpointing.

Swapping tends to give better overheads than checkpointing when the memory budget is quite tight. We can see why from the Compute-Memory trade-off curves for checkpointing in Figure 4.1 - compute cost degrades quite drastically for very low M. This does not mean checkpointing is not needed and that swapping is without difficulty. Firstly, as shown in Section 4, checkpointing gives good results for most reasonable M. Secondly, in order to precisely hide communication under computation, we need to know the precise per-layer costs, just like when finding the optimal checkpointing policy, which can be hard to do. Moreover, as described above, it is very hard to solve for an optimal swapping schedule given these costs. Zhang et al. [24] have proposed a powerful approximation that gets excellent results. Similar to my work, they swap enough tensors to satisfy the memory budget whilst trying to minimise overhead, rather than picking an arbitrary point on the compute-memory

trade-off.

However, it is still not perfect. It is fairly often the case that the overhead of recomputing a tensor is less than the overhead caused by swapping, especially given the ever-increasing performance of ML libraries and GPUs. Tensors towards the end of a network are especially difficult to swap without incurring overhead, as they are the first to be reused in the backward. Recomputing them instead is not affected by their position in the network. Most importantly though, choosing to recompute a tensor rather than swapping it will free up PCIe over the time period it would have been swapped, *bringing forward all future swaps*. Thus, I believe that any approach to combining the techniques should have this idea at their core.

I did try to extend my policy solver to do this, given an already worked out swapping schedule. When it measures the cost of checkpointing a node that has been swapped, it considers the possible savings from being able to swap the other tensors earlier. It also takes into account the fact that swapping a checkpointed node means all of the dropped forwards would have to be recomputed from the last checkpoint. Fully deriving the theory for this and implementing it was outside of the feasible scope of this project.

Also, I have recently come across a whole set of literature in the automatic differentiation community that describes swapping as a generalisation of checkpointing to two levels of memory, called heirarchical or multistage checkpointing [77, 78, 79, 80]. Swapping becomes merely checkpointing to the second layer. These approaches model this is as a scheduling problem that takes into account memory costs, compute costs, transfer costs, and even allocation/deallocation costs. However, they are surely *really* intractable to solve perfectly. Due to time constraints, exacerbated by how recent some of the work is, I am unsure of the practicality of these algorithms when applied to neural networks or in what sense they claim to be optimal. I do know each has their limitations, for example not all permit multiple recomputations or asynchronous swapping (which greatly reduces overhead). However, this seems like a powerful technique that we should look into in ML.

5.4.2 Choice of Convolutional Algorithm

Next, I consider the choice of convolutional algorithm. The convolutional layer is a type of layer in a neural network. It is one of the most compute and memory intensive. Various algorithms exist for computing a convolution [81, 82], with varying time and space complexities. This adds another dimension to our optimisation problem, akin to how database optimisers search over the logical operators *and* their physical implementations.

Recall Figure 2.16 which shows the memory breakdown of neural networks. The space required for the workspace, that is the specific implementation of a convolutional layer, is quite significant. Therefore, what implementation we choose can have

a large affect on how we can apply the other optimisations. It is not just that more memory can be saved, but that compute can be saved when memory is being over-aggressively reduced by other heuristics. For example, Rhu et al [7] developed `vDNN`, which achieves most of its memory savings from swapping. However, it is not based on real costs; they have heuristics to either swap all layers or all convolutional layers. Often, this is too much, causing unnecessary overhead from communication. To combat this, they profile the convolutional layers using the different implementations, and then greedily choose faster, less memory-efficient implementations of the layers whilst the memory budget is still satisfied.

It should be possible to extend my policy solver algorithm with checkpointing. Rather than choosing just the tensor to checkpoint, the algorithm will choose both a tensor and an implementation of computing that tensor. One drawback is that clearly it would make the algorithm less efficient, as it increases the dimension of the search space, though that dimension is quite small as there are only a handful of algorithms. However, I think it could significantly improve the technique, due to the large variation in space and time of the implementaions, unlocking new strategies for the algorithm to discover. For example, the poor compute degradation of checkpointing when the memory budget is very small could be alleviated by choosing more memory-efficient implementations, rather than excessively recomputing.

Chapter 6

Conclusion

In this thesis, I have described how deep learning is hindered by its large memory requirement, and how checkpointing is used to make it more memory-efficient. I have outlined the work that the automatic differentiation community has already done in this field, which the machine learning community is now rediscovering.

My contributions center on improving the dynamic programming checkpointing technique proposed by Gruslys et al. of Google DeepMind. I have demonstrated that this technique is very *powerful*, due to the low memory cost it can achieve. Even with one recomputation, large memory savings can be obtained at little overhead. Through multiple recomputations, it can even achieve constant-space complexity with respect to network depth, at quadratic-time overhead. However, unlike other work, this technique is also *flexible*. It will automatically find for the user the most computationally efficient checkpointing strategy that recomputes as little as possible, rather than picking a single point on the compute-memory trade-off curve.

I have generalised the existing technique to take into account the precise compute and memory costs of each tensor; allowing the policy solver to be significantly more judicious in finding the *truly* optimal policy that gives the least computational cost, or to be able to satisfy even lower memory budgets. I regret that I do not have a working implementation and can only use simulated results to demonstrate the efficacy of profiling over uniform costs.

I have explained the limitations of checkpointing to sequential networks, but described recent literature that has made significant headway in overcoming this. However, they either cannot flexibly find the optimal compute cost policy given a memory budget, or they do not implement multiple recomputations. In the future, the technique studied in this thesis should be combined with their techniques so we can solve for the optimal policy of more general networks.

I have showed how the continuous memory state of the solver requires bucketing to avoid an intolerably long execution time. Although I showed reasonable amounts of bucketing to only have a modest effect on optimality, I have also suggested how this

limitation could be elided altogether by leveraging existing reinforcement learning techniques.

Lastly, I have appraised checkpointing with respect to other memory optimisations for deep learning and suggested how they could be combined. I argued that checkpointing should be used to supplement swapping, which can already achieve very low compute overhead. I aso outlined how the policy solver can be extended to additionally solve for the optimal implementation of each layer, and argued the signficance of this.

More generally, throughout my research on this project, I have found that deep learning systems are actually quite immature, and could certainly be improved when it comes to memory. I think the implementation of precise dynamic programming checkpointing would be a signficant improvement for the end-user, as it can automatically give the user great memory savings at reasonable overhead (depending on just how low the memory budget is); and because, to my knowledge, no sophisticated, precise checkpointing or swapping exists in the popular frameworks. In the long run, I think there is surely a wealth of literature from more mature systems, such as compilers, databases, automatic differentiation or DAG execution engines; that could introduce major improvements to machine learning frameworks.

I look forward to seeing what happens.

Bibliography

[1] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training Deep Nets with Sublinear Memory Cost. apr 2016. URL http://arxiv.org/abs/1604.06174. pages i, vii, 3, 26, 32, 33

[2] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-efficient backpropagation through time. In *Advances in Neural Information Processing Systems*, pages 4132–4140, jun 2016. URL http://arxiv.org/abs/1606.03401. pages i, viii, xi, 3, 4, 31, 34, 36, 37, 61, 62

[3] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. SuperNeurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, pages 41–53. Association for Computing Machinery, feb 2018. ISBN 9781450349826. doi: 10.1145/3178487.3178491. pages vi, 3, 4, 74

[4] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. In *Advances in Neural Information Processing Systems*, volume 2018-Decem, pages 6389–6399, dec 2018. URL http://arxiv.org/abs/1712.09913. pages vi, 8

[5] Fei-Fei Li, Justin Johnson, and Serena Yeung. cs231n: CNNs for Image Recognition, Lecture 10: Recurrent Neural Networks, 2018. URL http://cs231n.stanford.edu/slides/2018/cs231n{_}2018{_}lecture10.pdf. pages vi, 15

[6] Atilim Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: A survey. *J. Mach. Learn. Res.*, 18(1):5595–5637, January 2017. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=3122009.3242010. pages vi, vii, 6, 16, 17, 18

[7] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, volume 2016-Decem, 2016.

ISBN 9781509035083. doi: 10.1109/MICRO.2016.7783721. URL http://arxiv.org/abs/1602.08124. pages vii, 3, 28, 74, 76

[8] Yaroslav Bulatov and Tim Salimans. Fitting larger networks into memory., 2018. URL https://medium.com/tensorflow/fitting-larger-networks-into-memory-583e3c758ff9. pages viii, ix, 23, 70, 71

[9] Lex Fridman, Daniel E. Brown, Michael Glazer, William Angell, Spencer Dodd, Benedikt Jenik, Jack Terwilliger, Aleksandr Patsekin, Julia Kindelsberger, Li Ding, Sean Seaman, Alea Mehler, Andrew Sipperley, Anthony Pettinato, Bobbie D. Seppelt, Linda Angell, Bruce Mehler, and Bryan Reimer. MIT Advanced Vehicle Technology Study: Large-Scale Naturalistic Driving Study of Driver Behavior and Interaction With Automation. *IEEE Access*, 7:102021–102038, jul 2019. doi: 10.1109/access.2019.2926040. URL https://arxiv.org/abs/1711.06976. pages 1

[10] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Kttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. StarCraft II: A New Challenge for Reinforcement Learning. Technical report, Google DeepMind, 2017. URL http://arxiv.org/abs/1708.04782. pages 1

[11] Ahmad Abdulkader, Aparna Lakshmiratan, and Joy Zhang. Introducing DeepText: Facebook's text understanding engine - Facebook Engineering, 2016. URL https://engineering.fb.com/core-data/introducing-deeptext-facebook-s-text-understanding-engine. pages 1

[12] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level Convolutional Networks for Text Classification. In C Cortes, N D Lawrence, D D Lee, M Sugiyama, and R Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 649–657. Curran Associates, Inc., 2015. URL http://papers.nips.cc/paper/5782-character-level-convolutional-networks-for-text-classification.pdf. pages 1

[13] Andrej Karpathy and Li Fei-Fei. Deep Visual-Semantic Alignments for Generating Image Descriptions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(4):664–676, 2017. ISSN 01628828. doi: 10.1109/TPAMI.2016.2598339. pages 1

[14] Andrea Dal Pozzolo. *Adaptive Machine Learning for Credit Card Fraud Detection*. PhD thesis, Université Libre de Bruxelles, 2015. URL http://di.ulb.ac.be/map/adalpozz/pdf/Dalpozzolo2015PhD.pdf. pages 1

[15] Martin Zlocha and Ben Glocker. Universal Lesion Detector: Deep Learning for Analysing Medical Scans. Master's thesis, Imperial College London, 2019. URL https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/1819-ug-projects/ZlochaM-Universal-Lesion-Detector-Deep-Learning-for-Analysing-Medical-Sca pdf. pages 1

[16] Amr Kayid, Yasmeen Khaled, and Mohamed Elmahdy. Performance of CPUs/GPUs for Deep Learning workloads, 2018. pages 2

[17] Stefano Scanzio, Sandro Cumani, Roberto Gemello, Franco Mana, and P. Laface. Parallel implementation of artificial neural network training. In *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, pages 4902–4905, 2010. ISBN 9781424442966. doi: 10.1109/ICASSP.2010.5495108. pages 2

[18] Radu Dogaru and Ioana Dogaru. Optimization of GPU and CPU acceleration for neural networks layers implemented in python. In *Proceedings - 2017 5th International Symposium on Electrical and Electronics Engineering, ISEEE 2017*, volume 2017-December, pages 1–6. Institute of Electrical and Electronics Engineers Inc., dec 2017. ISBN 9781538620595. doi: 10.1109/ISEEE.2017.8170680. pages 2

[19] G. Huang, Z. Liu, L. v. d. Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269, July 2017. doi: 10.1109/CVPR.2017.243. pages 2, 14, 15, 65

[20] 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. URL http://arxiv.org/abs/1409.4842. pages 2, 14, 15, 71

[21] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016. doi: 10.1109/CVPR.2016.90. pages 2, 14, 15, 66, 70

[22] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. jun 2016. URL http://arxiv.org/abs/1606.06160. pages 3

[23] Shriram S B, Anshuj Garg, and Purushottam Kulkarni. Dynamic Memory Management for GPU-based training of Deep Neural Networks. Technical report. URL https://www.cse.iitb.ac.in/{~}shriramsb/submissions/GPU{_}mem{_}ML.pdf. pages 3

[24] Junzhe Zhang, Sai Ho Yeung, Yao Shu, Bingsheng He, and Wei Wang. Efficient Memory Management for GPU-based Deep Learning Systems. feb 2019. URL http://arxiv.org/abs/1903.06631. pages 3, 74

[25] Benjamin Dauvergne and Laurent Hascoët. The data-flow equations of checkpointing in reverse Automatic Differentiation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3994 LNCS - IV, pages 566–573. Springer Verlag, 2006. ISBN 3540343857. doi: 10.1007/11758549_78. pages 3, 34

[26] James Martens and Ilya Sutskever. *Training Deep and Recurrent Networks with Hessian-Free Optimization*, pages 479–535. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-35289-8. doi: 10.1007/978-3-642-35289-8_27. URL https://doi.org/10.1007/978-3-642-35289-8{_}27. pages 3, 33

[27] Jeffrey Mark Siskind and Barak A. Pearlmutter. Divide-and-conquer checkpointing for arbitrary programs with no user annotation. *Optimization Methods and Software*, 33(4-6):1288–1330, nov 2018. ISSN 10294937. doi: 10.1080/10556788.2018.1459621. pages 3, 34, 72

[28] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1(1):35–54, 1992. doi: 10.1080/10556789208805505. URL https://doi.org/10.1080/10556789208805505. pages 3, 33

[29] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org. pages 4, 19

[30] 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. dec 2015. URL http://arxiv.org/abs/1512.01274. pages 4

[31] Adam Paszke, Gregory Chanan, Zeming Lin, Sam Gross, Edward Yang, Luca Antiga, and Zachary Devito. Automatic differentiation in PyTorch. In *31st Conference on Neural Information Processing Systems*, number Nips, pages 1–4, 2017. ISBN 9788578110796. doi: 10.1017/CBO9781107707221.009. URL https://openreview.net/pdf?id=BJJsrmfCZ. pages 4, 19, 27

[32] Tim Salimans and Yaroslav Bulatov. Gradient checkpointing in tensorflow, 2018. URL https://github.com/cybertronai/gradient-checkpointing. pages 4, 33

[33] Augustin-Louis Cauchy. Méthode Générale pour la Résolution des Systèmes d'Équations Simultanées. *Comptes Rendus a l'Académie des Sciences, Series A*, (25):536–538, 1847. pages 7

[34] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y. Object Localization Dataset. pages 7

[35] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. sep 2014. URL http://arxiv.org/abs/1409.1556. pages 8

[36] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning, jun 2018. ISSN 00361445. URL http://arxiv.org/abs/1606.04838. pages 8

[37] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. URL http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf. pages 9

[38] Anders Krogh and John A Hertz. A Simple Weight Decay Can Improve Generalization. In J E Moody, S J Hanson, and R P Lippmann, editors, *Advances in Neural Information Processing Systems 4*, pages 950–957. Morgan-Kaufmann, 1992. URL http://papers.nips.cc/paper/563-a-simple-weight-decay-can-improve-generalization.pdf. pages 9

[39] Andrew Y Ng. Feature selection, L1 vs. L2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*, pages 379–387, 2004. URL https://icml.cc/Conferences/2004/proceedings/papers/354.pdf. pages 9

[40] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *30th International Conference on Machine Learning, ICML 2013*, number PART 3, pages 2176–2184, 2013. URL https://www.cs.toronto.edu/{~}fritz/absps/momentum.pdf. pages 9

[41] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on*

Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015. URL http://arxiv.org/abs/1412.6980. pages 9

[42] G Cybenkot. Mathematics of Control, Signals, and Systems Approximation by Superpositions of a Sigmoidal Function. *Math. Control Signals Systems*, 2:303–314, 1989. URL https://pdfs.semanticscholar.org/05ce/b32839c26c8d2cb38d5529cf7720a68c3fab.pdf. pages 9

[43] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989. ISSN 08936080. doi: 10.1016/0893-6080(89)90020-8. URL https://www.sciencedirect.com/science/article/pii/0893608089900208. pages 9

[44] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997. ISBN 0070428077, 9780070428072. pages 9

[45] Erik Hallström. Backpropagation from the beginning, 2016. URL https://medium.com/{@}erikhallstrm/backpropagation-from-the-beginning-77356edf427d. pages 11

[46] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *AAAI Conference on Artificial Intelligence*, 02 2016. pages 14, 15

[47] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, volume 2017-Janua, pages 5987–5995, 2017. ISBN 9781538604571. doi: 10.1109/CVPR.2017.634. pages 14, 15

[48] A. J. Robinson and Frank Fallside. The utility driven dynamic error propagation network. Technical Report CUED/F-INFENG/TR.1, Engineering Department, Cambridge University, Cambridge, UK, 1987. pages 15

[49] Michael C. Mozer. A focused backpropagation algorithm for temporal pattern recognition. *Complex Systems*, 3, 1989. pages 15

[50] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, Oct 1990. ISSN 0018-9219. doi: 10.1109/5.58337. pages 15

[51] Mia Xu Chen, Orhan Firat, Ankur Bapna, Melvin Johnson, Wolfgang Macherey, George Foster, Llion Jones, Niki Parmar, Mike Schuster, Zhifeng Chen, Yonghui Wu, and Macduff Hughes. The best of both worlds: Combining recent advances in neural machine translation. *CoRR*, abs/1804.09849, 2018. URL http://arxiv.org/abs/1804.09849. pages 15

[52] Max E. Jerrell. Automatic Differentiation and Interval Arithmetic for Estimation of Disequilibrium Models. *Computational Economics*, 10(3):295–316, 1997. ISSN 09277099. doi: 10.1023/A:1008633613243. pages 16

[53] Andreas Griewank and George F. Corliss. *A taxonomy of automatic differentiation tools*, pages 315–329. Kluwer Academic Publishers, Philadelphia, USA, 1991. doi: 10.1023/A:1008633613243. pages 16

[54] Gundolf Haase, Ulrich Langer, Ewald Lindner, and Wolfram Mühlhuber. Optimal Sizing of Industrial Structural Mechanics Problems Using AD. In *Automatic Differentiation of Algorithms*, pages 181–188. Springer New York, 2002. doi: 10.1007/978-1-4613-0075-5_21. pages 17

[55] Stéfan Van Der Walt, S. Chris Colbert, and Gaël Varoquaux. The NumPy array: A structure for efficient numerical computation. *Computing in Science and Engineering*, 13(2):22–30, mar 2011. ISSN 15219615. doi: 10.1109/MCSE.2011.37. pages 19

[56] 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. doi: 10.1145/800125.804049. URL http://doi.acm.org/10.1145/800125.804049. pages 23

[57] Per Austrin, Toniann Pitassi, and Yu Wu. Inapproximability of Treewidth, One-Shot Pebbling, and Related Layout Problems. sep 2011. URL http://arxiv.org/abs/1109.4910. pages 26

[58] G J Chaitin. Register Allocation & Spilling via Graph Coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 98–105, New York, NY, USA, 1982. ACM. ISBN 0-89791-074-5. doi: 10.1145/800230.806984. URL http://doi.acm.org/10.1145/800230.806984. pages 26

[59] Andreas Griewank and Andrea Walther. Algorithm 799: Revolve: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation. *ACM Trans. Math. Softw.*, 26(1):19–45, mar 2000. ISSN 0098-3500. doi: 10.1145/347837.347846. URL http://doi.acm.org/10.1145/347837.347846. pages 33

[60] Tianqi Chen. Mxnet memory monger, 2016. URL https://github.com/dmlc/mxnet-memonger. pages 33

[61] Priya Goyal and other PyTorch contributors (see PyTorch GitHub). torch.utils.checkpoint module of pytorch v1.2.0, 2018. URL https://github.com/pytorch/pytorch/blob/v1.2.0/torch/utils/checkpoint.py. pages 33

[62] Laurent Hascoet and Valérie Pascual. The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification. *ACM Trans. Math. Softw.*, 39(3):

20:1—-20:43, may 2013. ISSN 0098-3500. doi: 10.1145/2450153.2450158. URL http://doi.acm.org/10.1145/2450153.2450158. pages 34

[63] Richard Bellman. The theory of dynamic programming. *Bull. Amer. Math. Soc.*, 60(6):503–515, 11 1954. URL https://projecteuclid.org:443/euclid.bams/1183519147. pages 35

[64] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 0262193981. pages 35

[65] Hans L Bodlaender. Discovering Treewidth. In Peter Vojtáš, Mária Bieliková, Bernadette Charron-Bost, and Ondrej Sýkora, editors, *SOFSEM 2005: Theory and Practice of Computer Science*, pages 1–16, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-30577-4. pages 72

[66] Joseph W H Liu. An Application of Generalized Tree Pebbling to Sparse Matrix Factorization. *SIAM J. Algebraic Discrete Methods*, 8(3):375–395, jul 1987. ISSN 0196-5212. doi: 10.1137/0608031. URL http://dx.doi.org/10.1137/0608031. pages 72

[67] Jianwei Feng and Dong Huang. Cutting Down Training Memory by Refowarding. jul 2019. URL http://arxiv.org/abs/1808.00079. pages 72

[68] Mitsuru Kusumoto, Takuya Inoue, Gentaro Watanabe, Takuya Akiba, and Masanori Koyama. A Graph Theoretic Framework of Recomputation Algorithms for Memory-Efficient Backpropagation. may 2019. URL http://arxiv.org/abs/1905.11722. pages 72

[69] NVIDIA. NVIDIA T4 GPU Specifications. Technical report, 2019. URL https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-datasheet-951643.pdf. pages 73

[70] E Altman. *Constrained Markov Decision Processes*. Stochastic Modeling Series. Taylor & Francis, 1999. ISBN 9780849303821. URL https://books.google.co.uk/books?id=3X9S1NM2iOgC. pages 73

[71] Christian Artigues, Sophie Demassey, and Emmanuel Neron. *Resource-Constrained Project Scheduling: Models, Algorithms, Extensions and Applications*. ISTE, 2007. ISBN 190520972X. pages 74

[72] B Shriram, Anshuj Garg, and Purushottam Kulkarni. Dynamic Memory Management for GPU-Based Training of Deep Neural Networks. *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 200–209, 2019. pages 74

[73] Shijie Li, Xiaolong Shen, Yong Dou, Shice Ni, Jinwei Xu, Ke Yang, Qiang Wang, and Xin Niu. A novel memory-scheduling strategy for large convolutional neural network on memory-limited devices. *Computational Intelligence and Neuroscience*, 2019:1–12, 04 2019. doi: 10.1155/2019/4328653. pages 74

[74] Tung Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. Tflms: Large model support in tensorflow by graph rewriting, 07 2018. pages 74

[75] X. Chen, D. Z. Chen, Y. Han, and X. S. Hu. moDNN: Memory Optimal Deep Neural Network Training on Graphics Processing Units. *IEEE Transactions on Parallel and Distributed Systems*, 30(3):646–661, March 2019. ISSN 1045-9219. pages 74

[76] Y Ito, R Matsumiya, and T Endo. ooc_cuDNN: Accommodating convolutional neural networks over GPU memory capacity. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 183–192, dec 2017. doi: 10.1109/BigData.2017.8257926. pages 74

[77] Guillaume Aupy, Julien Herrmann, Paul Hovland, and Yves Robert. Optimal multistage algorithm for adjoint computation. *SIAM J. Scientific Computing*, 38, 2016. URL https://www.researchgate.net/publication/278381104{_}Optimal{_}Multistage{_}Algorithm{_}for{_}Adjoint{_}Computation pages 74, 75

[78] Michel Schanen, Oana Marin, Hong Zhang, and Mihai Anitescu. Asynchronous two-level checkpointing scheme for large-scale adjoints in the spectral-element solver nek5000. *Procedia Computer Science*, 80:1147–1158, 12 2016. doi: 10.1016/j.procs.2016.05.444. URL https://www.sciencedirect.com/science/article/pii/S1877050916309267. pages 74, 75

[79] Navjot Kukreja, Jan Hückelheim, and Gerard J. Gorman. Backpropagation for long sequences: beyond memory constraints with constant overheads. 2018. URL http://arxiv.org/abs/1806.01117. pages 74, 75

[80] H-Revolve: A Framework for Adjoint Computation on Synchrone Hierarchical Platforms. working paper or preprint, March 2019. pages 74, 75

[81] Rui Xu, Sheng Ma, and Yang Guo. Performance analysis of different convolution algorithms in GPU environment. In *2018 IEEE International Conference on Networking, Architecture and Storage, NAS 2018 - Proceedings*. Institute of Electrical and Electronics Engineers Inc., oct 2018. ISBN 9781538683675. doi: 10.1109/NAS.2018.8515695. pages 75

[82] Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang Zhou. Optimizing memory efficiency for deep convolutional neural networks on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 54:1–54:12, Piscataway, NJ, USA, 2016. IEEE Press. ISBN 978-1-4673-8815-3. URL http://dl.acm.org/citation.cfm?id=3014904.3014977. pages 75