Gradient Compression in Distributed Deep Learning

Daniel Harris (B165702)

Master of Science

Artificial Intelligence

School of Informatics

University of Edinburgh

2020

Abstract

Distributed deep learning is a technique used to reduce the large training times associated with large, complex models by training across multiple processors. However, as these distributed computations are scaled up across more and more processors, the communication of model training updates between these processors can start to become a bottleneck - potentially outweighing the savings in running costs gained by moving to a distributed environment in the first place. Therefore, the field of communication-efficient training has become an important research area which looks to satisfy the accelerating demand of distributed training, and its related costs.

This work looks at gradient compression, a technique used to reduce the memory volume of the training updates that are communicated. A comparative survey is carried about between three of the most prominent gradient compression techniques, Top-k, DGC and QSGD, focusing on providing a fair and quantitative evaluation across the spectrum of metrics that defines the field.

It is found that tuning models on each compression setting separately, as opposed to adopting baseline settings, can lead to a fairer evaluation of techniques - sometimes even consistently beating the baseline as model size and worker number is scaled. In addition, sparsification techniques maintain modest performance relative to the baseline just like quantisation, but with higher gradient compression. However, this higher compression can come at a cost of additional computation time or losses in convergence time which negate any positive benefits of the compression. With a range of experimental details and settings particular to a given situation, it is ultimately concluded that there is no objectively best compression technique of those surveyed, and consideration of the resources at hand is the most useful indicator for which technique is best to adopt.

Acknowledgements

I would like to first and foremost thank my supervisors, David Wardrope and Roberto Pellegrini, for their consistent support and advice throughout the dissertation process, alongside fellow cluster members Maysara Hammouda and Di Weng for their help and discussions.

Thank you to my family, especially for help with so suddenly moving across the country, and thank you Camille for being my shot of adrenaline every day, pushing me through with relentless care.

Thank you to Camille, Chris and Theo, for agreeing to give up your time and comb through this dissertation with me (sorry, also).

Finally, I could not graduate from this course without a special mention to the Seventh Floor Crew, all of whom inspire me in my transition to artificial intelligence everyday, and have been a source of constant entertainment and friendship all year.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Daniel Harris (B165702)*)

Table of Contents

Chapter 1

Introduction

Deep Learning has become a powerful method for solving a broad family of tasks, ranging between machine translation [62], speech recognition [24] and object detection [48]. Its progress has multiple attributions: namely increasing availability and quantity of multi-modal data for training of complex models [23], and rapid advances in machine learning algorithms and techniques (for example, convolutional neural networks and the 'ImageNet moment' [32]). One of the key reasons for this progress, however, lies in the increased access to cheaper and more powerful hardware.

Use of graphical processor units (GPUs) in deep learning have allowed for training of deeper and more complex models. This is because they can be used to perform highly-parallel computation in short amounts of time [11, 47]. This scale in model size and computational power matters and is observable through ever-improving results in multiple fields. A notable, recent example can be seen in advancements in parameter-hungry language models [9]). However, scale also matters as a variable in training costs for these models - on a single GPU, training of popular deep models such as ResNet-50 can take many days [21].

This problem of scale has encouraged the introduction of distributed deep learning systems as a method of parallelising model training [14]. Large-scale clusters of machines (also referred to as workers) are used to distribute batches of data, most commonly, into even workloads, such that computations can be run in parallel and results aggregated into one global model update. This reduces overall training times of complex models, however introduces further penalties to the cost of training. Networks of machines need to communicate individual updates, which can often be slowed by bandwidth limitations of the connections. Survey-

ing technological advances over the past decade, 5× greater relative speedups can be seen in GPUs compared to network bandwidth in distributed systems [19, 36].

Relative increase in computational power of GPUs has led to a situation where model parameter updates are being produced more frequently than they can be communicated. This bottleneck-effect can cause overwhelming of central parameter servers used in distributed deep learning architectures [13], and under-utilisation of expensive GPU computational power as they remain idle between communication bursts. Therefore, it has become an important field of research to combat this communication problem.

One popular extension of this research investigates methods of gradient compression of model updates [53]. This technique aims to reduce the size of the parameter updates that are communicated and used to perform steps of a model optimiser such as stochastic gradient descent (SGD). Reduction in gradient size results in lower volume of traffic through network communication lines per model training iteration, providing a form of solution to the communication bottleneck issue. Compression ratio of gradients can vary on orders of magnitude, and provide differing convergence guarantees and computational overhead - it is the mark of an effective gradient compression technique to be able to minimise loss of model accuracy, while operating with low computational overhead and yielding a high compression of the gradients.

1.1 Problem Statement

In the field of gradient compression, there is a large scope of factors that can be changed and experimented with, including model architecture, size, dataset, optimiser and hyperparameters. This scope makes it difficult to compare innovative gradient compression techniques, due to inconsistent test-beds and frameworks. One of the more notable examples of this is between two prominent techniques in gradient quantisation (a method of compression), where conclusions are drawn relative to each other despite the use of different architectures [3, 59]. Further, it can be the case that techniques are presented without consideration of important metrics (such as computation overhead time), which can lead to a misrepresentation in true effectiveness of a technique [63].

Few surveys exist comparing gradient compression techniques directly, and

of those that do, most are more qualitative [17, 53]. This means that, for current practitioners looking to use gradient compression to combat difficulties with their own distributed systems, the literature does not provide a useful reference for understanding which techniques are most applicable for a given situation. This work is left to the practitioner to determine, which is time-consuming.

The current work identifies a further gap in the literature for gradient compression: the focus of qualitative, or even quantitative, surveys of techniques is most frequently within the computer vision domain [63]. Compared to work in natural language processing (NLP), model network architectures in computer vision tend to have a relatively higher computation to communication ratio, due to weight-sharing structures and non-sequential, parallel computation [64]. Considering motivation in easing the communication bottleneck in distributed systems, focus on recurrent neural network (RNN) language modelling architectures, with an inherently lower computation to communication ratio, will therefore be valuable.

Under consideration of these points, the objectives of the current work are outlined below:

1. Compare a careful selection of gradient compression techniques from the literature, under an equivalent framework that allows for fair, relative comparisons and conclusions to be drawn.

2. Evaluate all metrics associated with training a distributed model using gradient compression - this includes testing accuracy, rates of convergence, computation time and compression ratio (each outlined in Section 4.3.1).

3. Approach the study through the lens of recurrent neural network architectures, which allows for evaluation of gradient compression techniques in a highly appropriate setting not substantially investigated in the literature.

1.2 Research Hypotheses

In this work, it is predicted that each compression technique will be able to successfully compress gradients while maintaining a test performance that is not greatly lower than the zero compression, baseline case. Reflecting research trends, it is expected that the Deep Gradient Compression (DGC) technique in

particular will perform best as compression is scaled to larger ratios. However, it is expected that this performance will trade off with running costs, especially as distributed environments scale in worker number and model size, making compression technique choice more ambiguous.

Further to this, more focused hypothesis discussion is deferred to Section 4.4, by which point the scope and details of the project are laid out. This allows for collection of informed expectations just before results are presented in Chapter 5.

1.3 Overview

This dissertation is set up to guide the reader first through the theory and motivations of the project, before detailing and discussing experimental results.

In Chapter 2, further required background for distributed deep learning and gradient compression is outlined. Chapter 3 details the methodology and approach adopted, and Chapter 4 then provides experimental details and scope of how the experiments are setup and evaluated. Finally, Chapter 5 presents the experimental results alongside their discussion, before conclusions are drawn in Chapter 6.

Chapter 2

Background

2.1 Distributed Deep Learning

Distributed deep learning defines an infrastructure consisting of multiple compute nodes, which are able to process data in parallel and work together to train a deep neural network model. These compute nodes typically consist of multiple GPUs, or workers, that are used to accelerate training and inference of deep learning models. Other hardware components exist such as FPGAs [44] and Google TPUs [28], depending on the application. Recent surveys of network architectures show a general dominance in the use of GPUs, and more importantly, a growing dominance of multiple-node architectures in research, reflecting the increased importance of distributed learning for accelerating experimentation [5].

For communicating information between compute nodes, the most common interconnection is the Infiniband adapter, introduced for use in distributed deep learning between GPU node clusters [12]. Other popular interconnects involve Ethernet, which likewise has a high bandwidth, however Infiniband sports a lower latency which is preferable for faster communication. Despite this, advances in technology for network communication remain behind hardware accelerator progress [36], and the expense of 100Gbps+ connection infrastructures is often foregone in favour of cheaper but slower alternatives, for example popular Amazon EC2 data centre instances, which run on up to 25Gbps [19].

Beyond these hardware infrastructure decisions that must be made for distributed systems, there are many additional decisions. Described below are three such decision areas that are important, which shape communication efficient dis-

tributed deep learning, and inform design choices in Chapter 4; further details on infrastructure design can be found in comprehensive surveys [5, 39].

2.1.1 Parallelism

Parallelism defines how splitting a task among computational resources is performed. Most commonly, this takes the form of data parallelism, where data is split into non-overlapping batches which are fed to identical models held on each worker in a distributed system. The majority of operations (excluding, for example, batch normalisation) in feed-forward and backpropagation steps of a deep network operate on a sample level, which allows mini-batch updates of parameters to be performed independently and then aggregated at the end of a full iteration. In the data parallel regime, the usual optimisation problem becomes

$$\min_{\mathbf{x},\mathbf{y}\sim p(\mathbf{x},\mathbf{y})} \quad \frac{1}{n}\sum_{i=1}^{n} L_i(\mathbf{x}) \tag{2.1}$$

for input sample $\mathbf{x}$, loss function between true and predicted values L_i and number of workers n. Full proof of the equivalence to the non-distributed case is outlined in Appendix A.1.

Model parallelism is an alternative technique used, which distributes work across neurons in each layer, effectively splitting the model. This is often used in scenarios where a model is too large to fit efficiently in memory on a GPU, however has two major drawbacks: firstly, the allocated split is non-trivial to determine (in fact NP-complete [40]); secondly, partitioning by structure incurs larger penalties in the volume of information to be communicated between compute nodes. Therefore data parallelism is more common in practice.

2.1.2 Centralisation

Centralisation represents the infrastructure setup of a distributed environment, and largely falls under two main categories. The first is a centralised regime where all updates from separate workers are communicated to one location, often a parameter server. Updates are then applied to a master model at once, and the parameter changes are communicated back to each worker. The second is a decentralised regime, which largely relies on message passing interface (MPI) protocols such as `allreduce` or `allgather` to communicate updates between nodes themselves. The choice of centralisation hinges on many factors,

including network bandwidth, topology and latency. Decentralised architectures are lightly considered a preferable setup, as they do not necessitate extra parameter server-type resources, or contain a single crucial failure point for the network. More applicable to this work, however, is a discussion on communication. Decentralisation results in a scaling of communication logarithmic in the number of workers [26], as well as avoiding the need for 'double compression', where gradient compression must be performed in two directions, which incurs additional overhead [57].

2.1.3 Synchronisation

Synchronisation describes the level of coordination between communicated updates. Global aggregation and model changes must all be made and communicated before the next iteration can proceed when using synchronous updates. This is algorithmically parallel to classic SGD [31], allowing for convergence properties to be well-reasoned, however also has a problem with only being as fast as the slowest worker in the network. Asynchronous updates instead send messages as and when they are ready, which eliminates this problem. Unfortunately, this can then lead to 'stale' parameters (the gradient update is less relevant than it was several iterations prior), which harm convergence properties of the model, making synchronous updates in general the more popular method.

2.2 Communication Efficient Training

Combating the issue of communication in distributed deep learning is an important one - with a larger acceleration of advances in computational power relative to network connection speeds in these distributed systems, this means that many more updates per unit time need to be communicated than is possible with current bandwidth capabilities. Two prominent methods exist to help ease this issue.

Communication scheduling. It is found that in popular deep learning frameworks, the timing of parameter update communication is random and leads to high variance in iteration, and therefore communication, time [37]. Therefore it becomes important to optimise the order and timing of updates to maximise

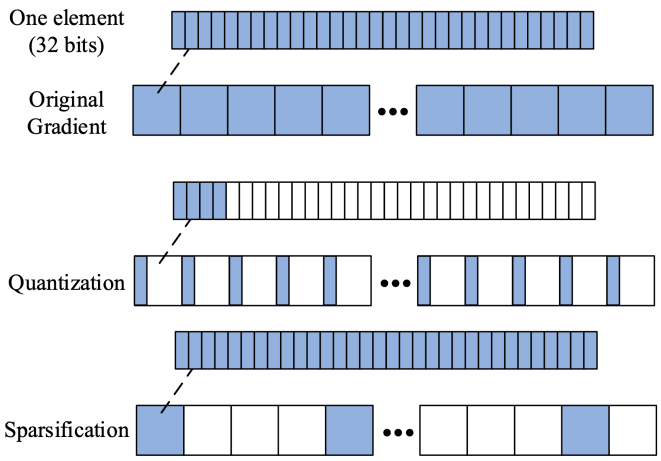

Figure 2.1: Gradient quantisation/sparsification of a gradient vector. Quantisation acts to reduce the number of bits of every element, whereas sparsification selects a few components, but at full bit representation. *Image Credit*: Tang et al. [53].

bandwidth usage. Further to this, methods have been employed that focus on sending high-priority updates when the queue becomes large [58].

Gradient compression. The second method of improving communication efficiency is the compressing of model parameters, or gradients. In the case of the former, popular approaches are to reduce the precision of model parameters from the 32-bit floating point accuracy typically used in deep learning [22], which can lead to more than double-savings in communication volume. This work focuses instead on gradient compression, a technique that reduces the volume of the gradient updates being communicated between workers, which are used to step an optimiser model. This method can be largely split into two categories: gradient quantisation, and gradient sparsification.

2.2.1 Gradient Quantisation

Gradient quantisation is a compression technique that looks to lower the bit-representation of the components of a gradient vector, by defining discrete buckets that the continuous data is assigned to. The first form of quantisation reduced values to 16-bits [22], which corresponds to a compression ratio of 2× since each component occupies half as much volume. Multiple quantisation algorithms have since been introduced, and it should be noted that due to natural limitations of the method, the maximum compression ratio that can be achieved is 32×.

Limited-bit variations on quantisation which simply truncate the bit-number continue the trend of increasing compression ratio, from 8-bit [15] all the way to 1-bit quantisation. This latter method is the most extreme and can take different forms, such as a threshold-based scheme (where values below a threshold are compressed to '0', and otherwise '1') [50], or a sign-based scheme which encodes values based on their signum [8]. In either case, due to the lossy nature of compression techniques, it becomes important to additionally define an error-feedback mechanism which helps to ensure the convergence properties of 1-bit quantisation [8]. This involves keeping the quantisation error of the compression in memory and adding it back to the respective gradient in the next iteration (defined in more detail in Section 3.1.1).

Other quantisation schemes instead use probabilistic techniques to assign gradients to buckets. TernGrad [59] uses three levels $\{-1,0,1\}$ to define values of gradient vector g, reached via the equation: $s_t \cdot sign(g) \circ b_t$; for maximum absolute value s_t, and b_t defined via a Bernoulli distribution. This achieves a compression ratio 16×. QSGD [3] incorporates a probabilistic, stochastic rounding function, which instead defines a family of algorithms which are able to trade off compression ratio for variance in convergence guarantees, through tuning of the bucket size used (Section 3.1.3). Further quantisation techniques exist including adaptive quantisation [34], which reacts to the gradients to provide a non-static compression ratio, and split-gradient quantisation [42], which compresses gradient differences to help guarantee model convergence properties.

2.2.2 Gradient Sparsification

The other prominent form of gradient compression used is sparsification. This technique works by selecting a proportion of the gradient components and sending just their values and indices (at full 32-bit precision). An illustration of this, in comparison with gradient quantisation, is given in Figure 2.1. The first algorithm for sparsification, Truncated Gradient [33], was employed primarily to address memory and computation constraints with model training, however this group of methods also has an advantage over quantisation in reducing communication volume of gradients, since they are not limited to only 32× compression ratios. Further study into the nature of gradient updates in deep neural networks finds a distribution of values that is narrowly dispersed and frequently

close to zero [2], making sparsification an ideal target.

Fundamental approaches to gradient sparsification mostly involve masking or thresholding of values, which identifies the components to send in communication with other workers. The most basic of these is a random selection of $k\%$ of the gradient values, otherwise known as Random-k [51]. This is a somewhat naive approach, however, and instead techniques such as Top-k are more popular [16, 51]. Here, the gradient values are first ordered (by absolute value) and then the top $k\%$ proportion taken, effectively deciding to only send the 'heavy-hitting' gradients that will make the biggest impact on optimiser updates. Fixed thresholds above which to send gradient components have also been proposed [52], however these can be harder to tune. Much like with quantisation, an error-feedback mechanism is often utilised to carry over sparsification error to further iterations, which helps with convergence.

Top-k selection of gradients is pushed further with Deep Gradient Compression [35], which additionally uses momentum (among other techniques) to both improve the error-feedback step, and combat the staleness that is observed in these correction steps (Section 3.1.2). Other methods of sparsification use an adaptive thresholding approach, such as AdaComp [10], which tunes the level of sparsification used according to the local activity of each layer, making compression more widely effective across layer types. Sketched-SGD [27] estimates the 'Top-k' gradients through use of a Count-Min sketch data structure, which provides additional benefits on the size and scalability of communicated updates.

Literature surrounding gradient compression techniques is rich with methods beyond standalone quantisation and sparsification. Hybrid compression techniques exist to combine the benefits of both [4, 43], which can help with compounding the compression effects. Further methods include low-rank decomposition of gradient update matrices [57], and second-order methods which communicate only unambiguous, low-variance gradients [54]. Details of more techniques are deferred to a full survey on the area [53].

2.3 Language Modelling

The current work focuses on gradient compression through the lens of language modelling. This task is chosen in line with research objective 3: to investigate more communication-heavy models, complementing previous work with a more

in-depth comparison of gradient compression techniques in language modelling.

Language modelling defines a task that sets out to estimate the joint probability distribution over a sequence of tokens:

$$P(\mathbf{s}) = P(w_1 w_2 ... w_N) = \prod_{n=1}^{N} P(w_n | w_{1:n-1}) \tag{2.2}$$

which follows from the chain rule, for word sequence $w_1 w_2 ... w_N$. In this expression there is a conditional dependence on a variable length context sequence of tokens. In early language modelling work, this context was set to a fixed size under a Markov assumption [29]. However, this leads to problems with long-range token dependencies, or computational complexity. More recently, continuous conditional language models, such as recurrent neural networks (RNNs), have solved this problem through use of co-learned word representations [41]. Here, tokens are embedded to be passed to a hidden layer that models the complete history, and then projected back to a complete vocabulary matrix to be softmaxed. This work focuses on the long-short term memory (LSTM) [25] variant of RNNs, which additionally utilises a 'memory cell' to help better encode long-term dependencies in token context.

Language models are most commonly evaluated using a metric called perplexity, which fundamentally represents the weighted-average branching factor of a language [29]. Through this definition, we understand it as the typical number of feasible words a language model will 'choose' between - the lower the better. Perplexity (PP) is calculated as:

$$PP(\mathbf{s}) = P(\mathbf{s})^{-\frac{1}{N}} = \sqrt[N]{\frac{1}{P(w_1 w_2 ... w_N)}} \tag{2.3}$$

This expression can be more intuitively reached through the cross-entropy loss function (see Appendix A.3 for a proof).

Chapter 3

Methodology

3.1 Compression Methods

This section walks through the gradient compression techniques investigated in this project in more detail. For each technique, a thorough background is provided, alongside motivations for their inclusion over other techniques, supporting research objective 1. The number of techniques to be chosen are limited due to the thorough random search experimentation (Section 4.2.1), on top of time and budget limitations (Section 4.1).

3.1.1 Random- and Top-k

Random-k is a gradient sparsification technique that fixes k as a proportion of the gradient update vector components (for example $k = 0.01$ represents 1%), and selects this many gradient elements to keep when communicating SGD updates. Top-k instead takes a more informed approach, ordering the gradient vector elements by largest absolute magnitude, and taking the top proportion of these. In either case, any elements that are not communicated are set to zero, such that the overall volume of communicated data is decreased.

To improve the convergence and generalisation properties of these techniques, error feedback (or residual memory, in this work) is used [30, 51]. This additional technique acts to store the difference between the compressed and uncompressed gradients before communication, thus keeping track of compression errors which can be added back on to gradients at the next iteration. This enables small-magnitude gradient elements to eventually update their respective parameters,

after they become large enough through repeated residual memory updates. For stochastic gradient $\mathbf{g}_t$ at timestep t, this correction takes the form:

$$\mathbf{p}_t = \gamma\mathbf{g}_t + \beta\mathbf{e}_t \quad \text{(Updated Gradient)} \tag{3.1a}$$

$$\mathbf{e}_{t+1} = \mathbf{p}_t - C(\mathbf{p}_t) \quad \text{(Update Compression Error)} \tag{3.1b}$$

where $\mathbf{p}_t$ is the error-updated gradient $\mathbf{g}_t$, and the residual error is updated for the next iteration as $\mathbf{e}_{t+1}$. Tuning parameters β and γ are set to unity for this work to reduce complexity, and $C(\cdot)$ represents an arbitrary gradient compression algorithm, serving in this case as Random-/Top-k.

When calculating the compression ratio for sparsification techniques, the values of both the gradient elements and their indices must be communicated, so that they can be decompressed at the other end. This cuts the perceived compression ratio in half; for example a Top-0.01 method represents a $100/2 = 50\times$ compression ratio. Further methods for packing sparse updates exist, such as described in Section 3.1.2 for Deep Gradient Compression, however the value-index communication method is adopted in this work, as it is most common.

Motivation. Both gradient compression techniques are simple and fundamental in the literature. Random-k serves as a useful naive baseline against which further methods can be tested. Top-k promises a high compression ratio, and is quick to implement, which makes for a useful method to analyse for practitioners looking for quick, effective solutions.

Notes. Each technique uses built-in library functions to perform tensor manipulation, with Random-k using the `torch.randperm()` function, and Top-k using the `torch.topk()` function. It is suggested that simply sorting the tensor can be faster on GPUs in practice[1], and so this approach is also tested in experimentation.

3.1.2 Deep Gradient Compression (DGC)

Deep Gradient Compression (DGC) [35] also approaches gradient compression using sparsification of gradient components, to achieve high compression ratios between $270\times$ and $600\times$ with negligible loss of test perplexity. DGC aims to improve the local accumulation of gradients (changing the error-feedback mechanism of Top-k). Furthermore, it attempts to combat the effect of staleness in

[1] https://github.com/dhroth/sketchedsgd/blob/master/sketchedsgd/sketched_optimizer.py

the residual errors that are fed back to gradients. It achieves this through four techniques:

- **Momentum correction** adapts the momentum SGD formula, such that momentum can be applied to residual memory gradient updates (as are used in methods such as Top-k). In this technique the gradient update vector $\nabla_{k,t}$ takes the form:

$$\mathbf{u}_{k,t} = m\mathbf{u}_{k,t-1} + \nabla_{k,t} \tag{3.2a}$$

$$\mathbf{v}_{k,t} = \mathbf{v}_{k,t-1} + \mathbf{u}_{k,t} \tag{3.2b}$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \sum_{k=1}^{N} \text{sparse}\left(\mathbf{v}_{k,t}\right) \tag{3.2c}$$

  where $\mathbf{u}_{k,t}$ is a 'velocity' term that can be considered as the corrected gradient, $\mathbf{v}_{k,t}$ is the accumulated gradient term, and sparse() is a sparsification function. This has the effect of correcting the trajectory for parallel SGD and helping with convergence of the model when using sparse updates. The momentum parameter m is treated as a hyperparameter to be tuned in this work.

- **Momentum factor masking** is used to combat momentum staleness in residual gradients. This is important because these correction updates, for any particular parameter, can take on-the-order of thousands of iterations to become large enough to be applied to the parameter before communication. This means that gradients are at risk of being carried in a stale direction which is no longer applicable. This momentum factor mask is calculated as $|\mathbf{v}_{k,t}| >$threshold, for accumulated gradients $\mathbf{v}_{k,t}$, and the threshold found as the minimum absolute value of a 1% sample of the gradients. Applying this mask to the gradients acts to set affected components' momentum to zero, preventing them being carried in a stale direction.

- **Local gradient clipping** at each node is used instead of being performed after gradient aggregation, since gradients that are accumulated on each node are independent. **Warm-up training** defines the use of a smaller learning rate during the initial stages of training, and is used to reduce the impact of aggressive, initially-delayed gradient updates.

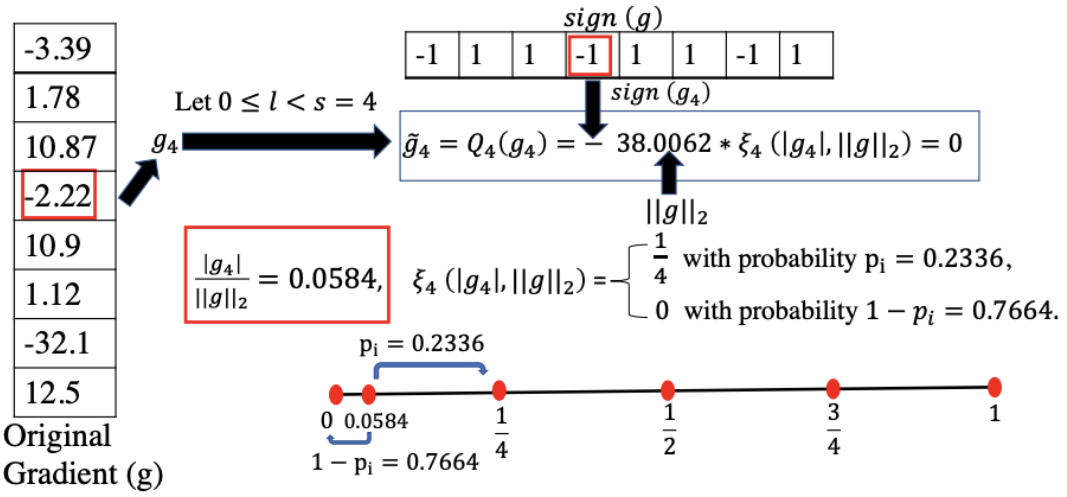

Figure 3.1: QSGD example with s = 4, l = 3. The possible code-words are $0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1$, and are represented by 3-bits. *Image Credit*: Tang et al. [63].

When packing the sparse gradient updates, the authors use an encoding scheme that retains knowledge of the gradient tensor shape, and only communicates values as 32-bits, and the length of zero-runs between values as 16-bits (instead of values and indices). Although more efficient, inconsistencies like this between compression technique implementations can hinder fair comparison. This is because additional benefits are not shared, which makes it harder to evaluate differences due to the methods themselves. In this work, naive communication of values and indices is assumed.

Motivation. Of all the sparsification, hybrid and proximal methods that could be selected from, DGC was chosen as it met two main criteria: it boasts one of the higher compression ratios in the literature, and it is a well-cited, promising technique. A sparsification technique was selected as it could be most directly compared with the existing Random-/Top-k results.

3.1.3 Quantised SGD (QSGD)

Quantised-SGD [3] defines a family of gradient compression techniques that focus on reduction of gradient values to a lower bit representation. This is achieved through the use of randomised rounding of stochastic gradient values to a set of discrete quantisation levels - through changing the number of levels available, the communication time of distributed gradient updates can be traded off with the variance introduced, and therefore the convergence guarantees of the model.

Considering a gradient $\mathbf{g} \in \mathbb{R}^n$, then component $\mathbf{g}_i$ is quantised across s quantisation levels according to:

$$Q_s(\mathbf{g}_i) = ||\mathbf{g}||_2 \cdot \text{sign}(\mathbf{g}_i) \cdot L_i(\mathbf{g}, s) \tag{3.3a}$$

$$L_i(\mathbf{g}, s) = \begin{cases} \frac{l+1}{s}, & \text{with probability } \frac{s|\mathbf{g}_i|}{||\mathbf{g}||_2} - l \\ \frac{l}{s}, & \text{otherwise} \end{cases} \tag{3.3b}$$

where L_i is defined for $0 < l < s$ such that $\frac{|\mathbf{g}_i|}{||\mathbf{g}||_2} \in [\frac{l}{s}, \frac{l+1}{s}]$. A visualisation of this quantisation process is illustrated in Figure 3.1. Changing the parameter s (the number of quantisation levels) acts to control the number of bits of compression. For example, $s = 16$ represents 2^4 unsigned integers (from $0 - 16$), and so incorporating the $\text{sign}(\mathbf{g}_i)$ corresponds to a 5-bit compression. Stochastic rounding as above results in an unbiased estimate of the original gradient component.

Alongside use of statistical rounding, the authors of QSGD utilise Elias encoding to pack values (losslessly) to a lower bit-representation [18]. This is efficient since smaller magnitude numbers take fewer bits to encode, and larger values are argued to occur less frequently in data. Despite this, Elias coding is not applied in this work, much like in the work of other quantitative surveys [63]. Similarly to the exclusion of efficient packing encountered for DGC, this is so that compression techniques are kept on a level playing field for comparison.

Motivation. Choosing a gradient quantisation technique is important to perform a full review of compression techniques, however due to their relatively limited compression ratio capabilities (and project constraints), only one is chosen. QSGD is selected on the grounds of its variable compression ratio, which allows for investigation into trade-off between performance and compression ratio for a quantisation method.

Notes. Error feedback is not applied alongside QSGD, as is the case in the original work [3].

3.2 Distributed Environment Simulation

In light of the limited resources available for this project (Section 4.1), a novel approach is taken to simulate the effect of gradient compression on distributed systems, instead on a single machine (GPU). Since each worker during data-parallel distributed training calculates on its own split of the dataset mini-batch

at each iteration, then updates to the global loss are independent. Gradient updates from each worker to a central server (or each other in a decentralised setting) do not depend on the calculations of other workers, and so it is possible to simulate the distributed learning process using *gradient accumulation* [61].

Gradient accumulation involves running multiple backward passes through a neural network before any zeroing out (as is typically performed) of parameter gradients. Like with data parallel training, a full batch of data is processed in splits of mini-batches (n mini-batches for n simulated workers), however in the proposed simulation, these mini-batches are run one after the other in series on the same machine. Gradients are allowed to persist between mini-batch operations, cumulatively adding up to the final effective full-batch gradient. After all gradients are calculated on their respective mini-batch and added, an optimiser update step is carried out. This has the desired effect of performing a forward pass of each mini-batch through the *same* model at each iteration, simulating the process of having separate workers (with identical models) processing on splits of the data. Full pseudocode in the case of training an LSTM model can be found in Algorithm 1.

In the context of this project, a custom optimiser is developed to additionally handle compression of gradient updates from a mini-batch. This is because multiple backward passes of a model naturally accumulate the mini-batch gradients of a parameter additively in its `grad()` variable, however each gradient must be compressed separately before being accumulated in this way. This is handled by copying the parameter gradient, compressing it, and adding cumulatively to a local variable for use in stepping the model later. This allows for zeroing of parameter gradients between iterations as usual. This custom optimiser builds on the Horovod distributed deep learning framework[2].

Approaching distributed model training on a single GPU has the advantage of simulating performance of distributed models in a resource constrained environment, as with this project. However, the serial nature of batch processing (as opposed to parallel), is a drawback since computation time is higher relative to a distributed setting. Additionally, absence of practical steps, such as use of `allreduce` communication, reduce the accuracy of metrics such as communication time. These drawbacks are kept in mind when presenting results in Chapter 5.

[2]https://github.com/horovod/horovod

Algorithm 1 LSTM training epoch, simulating multiple workers on one GPU.

```
procedure RUN EPOCH(n_workers)
    h_list ← hidden states initialised for each worker
    epoch_loss ← 0                                          ▷ running loss
    for batch in epoch do
        for worker in n_workers do
            input, target ← worker split of the batch
            out, hidden = FORWARD(input, h_list[worker])
            h_list[worker] = hidden
            loss = CROSSENTROPYLOSS(out, target)
            epoch_loss += loss  / n_workers                 ▷ normalise loss update
            backward pass for gradients                     ▷ zero grads beforehand
            compress and accumulate gradients               ▷ accumulate in memory
        end for
        model gradients ← accumulated gradients
        step optimiser
    end for
end procedure
```

3.3 Dataset & Model

Following research objective 3 to focus on the language modelling sub-task, in this project the word-level language modelling Penn Treebank (PTB) dataset [38] is used. This is a dataset consisting of 923,000 training, 73,000 validation and 82,000 test tokens, all pre-processed which makes it a popular choice for use in comparative surveys in language modelling. In addition, the dataset size is not the largest in comparison to other popular datasets, making it ideal for experimentation under the resources of this project (Section 4.1). When processing the data, a vocabulary size of 10,000 is assumed, with all other less frequent tokens taking the `[UNK]` token.

The study of language models as opposed to convolutional neural networks, a more popular test-bed for compression techniques, is motivated by their relatively lower computation-to-compression ratio in distributed systems [35]. The model used in this work is a Long-Short Term Memory (LSTM) [25] model, which is a recurrent neural network based model (Section2.3, Appendix A.2). This family of models is chosen, despite rising popularity and momentum from the trans-

former family of language models [56], for two main reasons: firstly, the scope of experimentation for gradient on language modelling compression techniques is dominated by analysis of LSTM-based language models, better aligning the current work with literature surveys. Secondly, the scale of parameter number in transformer-based language modelling often reflects need for higher computational power beyond the resources of this project. Extending analysis of compression techniques towards other language models is a useful direction of future work.

The model choice for this project is a standard LSTM model, with additional features taken from literature, including the popular AWD-LSTM architecture[3]. The AWD-LSTM is not itself used for two reasons: firstly, this allows for gradient compression analysis and conclusions at a basic level, without additional regularisation techniques specific to one model type. Additionally, implementation and debugging of the novel simulated environment (Section 3.2) is made easier.

Weight tying from AWD-LSTM is utilised between the input and output embedding layers [46]. This acts to reduce the number of trainable parameters, which in turn allows for training of larger, more complex models for experimentation under the computation restrictions. In addition, the first and last layers of the LSTM have their input and output dimensions respectively fixed to the embedding dimension size, to further reduce intrinsic parameter number.

[3]https://github.com/salesforce/awd-lstm-lm

Chapter 4

Experiments

Description of project resources and scope as a result of any limitations, followed by experimental method and details of parameters used.
Project code available at `https://github.com/D-J-Harris/Gradient-Compression`.

4.1 Resources & Scope

- **Time**. This work is conducted on a timescale of ~ 80 days, beginning in June - a full breakdown of the planned use of time against the final use of time is provided in Appendix C. In line with planning, experimentation is allowed to take 30-40 days total, limiting the range of experiments that can be performed.

- **Computation**. To run enough experiments with models of a meaningful size (> 1 million parameters), access to a GPU is required for this work. One important factor when finding a suitable resource is that this should be a dedicated machine, since the computation time metric (Section 4.3.1) should be stable between runs. This eliminated the option of using a shared GPU cluster resource, or Google Colab[1], due to its habit in sharing GPU memory between users. In this work, a virtual machine instance is setup on the free trial Google Cloud services[2]. To run experiments, an NVIDIA T4 GPU[3] is used with 16GB memory, which provides ~ 30 days computation time using provided free credits. This is ideal for the project time frame.

[1] https://colab.research.google.com/
[2] https://cloud.google.com/
[3] https://www.nvidia.com/en-us/data-center/tesla-t4/

To run one model using the fair random search procedure outlined in Section 4.2.1, average run-time is approximately 4 hours, which restricts the number of experiments that can be run. It is therefore decided to test fewer compression methods (techniques and motivations outlined in Section 3.1), but with a broader number of variables under investigation, including the size of models, the number of workers, and the compression ratio adopted by compression methods.

4.2 Experimental Approach

4.2.1 Procedure

It is common practice in the literature for models to retain the hyperparameters tuned for the baseline when experimenting on further techniques [3, 1]. This implicitly has the effect of favouring the baseline test scores when comparing methods. To combat this, a new procedure for tuning on each method is implemented, to reinforce fairness in the comparison of results.

For each model type investigated (including variations in model size and compression technique used), a random search is performed over the space of hyperparameters for the given model on the same initial seed. Repeating runs for 60 iterations using this method of tuning has a 95% chance of finding a combination of parameters within 5% of the optima [7], giving each model type a fair chance at demonstrating relative optimal performance. Early stopping with model checkpointing is implemented on the validation set [20], which helps save on training time and allows for analysis of convergence properties for each experiment.

Plans to run repeat experiments on the best-case hyperparameters for each model type in theory allow for error bounds to be found on results. In practice, however, it is found that performance varies greatly even through only altering the random seed. This reinforces the above procedure of tuning under each model setting, and limits investigation to experiments run on one seed (for this project, 42).

Finally, in line with background research in Section 2.1, this work assumes synchronous, data-parallel training. Although an explicit architecture is not constructed, calculation of estimated communication times assumes a parameter server architecture with bi-directional communication.

4.2.2 Experiments Covered

Each individual experiment will focus on the variation of three experimental variables, to investigate the effect of changing and scaling common distributed settings on model performance. Each variable is changed within a pre-decided set of options, such that a landscape of results is created:

- **Number of workers** represents the (simulated) number of GPUs used in the training of the distributed model. The baseline case for this is a single worker, and scaling of this variable is investigated in the set $\{1, 2, 4\}$.
- **Model size** reflects the number of units in hidden layers of the LSTM and is varied between 100 (small), 300 (medium) and 500 (large). Through changing this parameter, the number of trainable parameters is changed between $1,161,600$; $4,444,800$; and $9,008,000$ respectively, which helps investigate scalability of techniques. These sizes are incomparable to much larger literature hidden sizes of ~ 1500 [35].
- **Compression ratio** is changed for each compression technique, to test the limits of their effectiveness. For sparsification techniques, Top-k and DGC, this ratio is varied on a logarithmic scale in the set $\{0.01, 0.001, 0.0001\}$. This corresponds to $50\times$, $500\times$ and $5000\times$ compression respectively, when considering communication of both gradient values and indices. For QSGD, compression ratio is masked by the quantum number variable. For this project, this is varied in the set $\{64, 16, 4\}$, which defines a $7-$, $5-$ and $3-$bit compression respectively. In terms of compression ratio this is $4.6\times, 6.4\times$ and $10.7\times$ respectively.

4.3 Experimental Details

4.3.1 Evaluation

To evaluate fairly each compression method, in line with research objective 2, it is important to define the metrics that span the literature. In conducting experiments that track all of the following metrics, this allows for full and fair comparison of compression methods on all important fronts.

- **Test perplexity**, as defined in Section 2.3, is the central metric consistently reported. This represents an accuracy-like metric - the lower the better. In the case of compression analysis, ideal techniques should result in little-to-no increase in perplexity evaluated on the test set. Perplexity is calculated through exponentiation of the cross-entropy loss on the test (or training/validation) set (see Appendix A.3 for an explanation).

- **Compression ratio** is a measure of the reduction in volume of communicated gradient updates, and is the metric all methods strive to maximise without sacrifice of test perplexity. Referred to also as 'bits per iteration', compression ratio is calculated as a ratio of pre- and post- compression gradient sizes (for example in megabytes, MB):

$$\text{Compression Ratio} = \frac{\text{Size of Gradient / MB}}{\text{Size of Compressed Gradient / MB}} \tag{4.1}$$

- **Computation time** represents the effect of implementing compression techniques on the computational overhead - sometimes this is considered negligible, but can become significant for more complex models. In the context of reducing communication time in a distributed environment, a high computation time can even outweigh time savings from a great compression ratio, and so is important to track. The current work takes a naive approach through measurement of the training time taken per epoch, using the `time.process_time()` library method. Due to simulating workers in series, Section 4.2.1, this time is also divided by number of workers.

  Communication time, on the other hand, is calculated given (compressed) gradient size, and estimated Ethernet speed, assuming a bi-directional (parameter server) communication.

- **Convergence rate** measures the time it takes for a model to converge, as per the number of epochs. This is subtly different to computation time, since two methods can have the same per-epoch time, yet converge (or early stop) in fewer epochs, for example. Understanding and outlining the difference is important in making fair technique comparisons, as noted for computationally-heavy sketching compression methods [63].

4.3.2 Settings

Random searches are performed through the Weights and Biases experiment tracking software[4], using the Sweeps feature. The following settings and ranges are found through an initial experimentation testing period.

When carrying out hyperparameter sweeps, learning rate is drawn from a uniform distribution $U \sim [1,50]$, with the optimal range of values found to be higher than expected due potentially to the use of vanilla SGD. Dropout is drawn from $U \sim [0,0.8]$, and for experiments running Deep Gradient Compression (DGC), the momentum hyperparameter is drawn from $U \sim [0.1,0.9]$. Batch size is set at 256 to utilise GPU memory, as observed using the `nvidia-smi` command, and split evenly among workers in a distributed setting. In all experiments, weights are tied between input and output embeddings. When performing early stopping in models, a patience parameter of 10 is used, and models are trained for up to 50 epochs, since fewer than this are needed for convergence of the baseline during initial testing. For all experiments, vanilla SGD is assumed, with alternative optimisers left for future work. It is decided to use no learning rate decay, which leads to better results.

Regarding the LSTM model parameters, through all experiments a sequence length (BPTT parameter) of 35 is used, along with a fixed 2 `torch.nn.lstm` layers. This is to minimise complexity in the number of variables explored. As in the `nn.Linear` module, the embedding layer is initialised according to $U[-\sqrt{k},\sqrt{k}]$, for $k = 1/(\text{num. embedding dimensions})$.

4.4 Detailed Hypotheses

In addition to the broader research hypotheses set out for this project (Section 1.2), below is a set of further hypotheses that focus on the details of the expected results, following the methodology and experiments laid out in Chapters 3 & 4. This will help guide discussion when presenting results in the next chapter.

1. **Compression ratio** increase is predicted to correlate with an increasing (worse) test perplexity compared to the baseline. In the sparsification

[4] https://www.wandb.com/

regime, DGC is expected to perform better than Top-k due to its momentum correcting residual memory.

2. **Computation time** for the sparsification techniques DGC and Top-k is predicted to be much greater than that for quantisation and no compression, as they require sorting tensor data. This could potentially outweigh the savings in communication time.

3. **Convergence** for QSGD, measured according to the early stopping epoch number, is expected to display greater variance as the compression ratio is increased. Similar behaviour is predicted with sparsification techniques too, which hasn't been explored previously.

4. **Learning rate** values found to be optimal for each compression technique are expected to differ from the baseline hyperparameter choices. This would motivate future work in the field to not carry baseline choices through to all experiments.

5. **Model size** increase is predicted to improve relative performance of high compression techniques, due to an increased number of more 'redundant' model parameters i.e. high compression becomes more effective as model size is increased.

6. **Worker number** increase is predicted to have a greatly positive effect on computation time regardless of compression technique, however to reduce final test performance. This is due to the compression being performed on smaller mini-batches.

Chapter 5

Results & Discussion

Results are grouped in terms of the evaluated metrics (Section 4.3.1). Within each subsection, comparisons are made between each of the variables explored in this project. Helpful supplementary (but non-essential) data and plots are deferred to Appendix B where space does not permit in the main body, and pointed to where relevant.

Results are assumed to be the best-case run from the random search methodology, unless a data range is displayed, in which case this represents the top 8 runs. For reference, model size (small, medium and large) definitions can be found alongside other variables in Section 4.2.2.

5.1 Baseline

Particular attention is paid to the results from the baseline experiments, which encompass training of the distributed system with zero compression applied. This serves as a reference point for each metric. Additionally, the Random-k compression technique is applied to demonstrate the effects of a naive compression relative to the baseline. Figure 5.1 reflects the clear drop in performance when using Random-k both with some form of residual error correction ($\sim 76\%$), and without ($\sim 271\%$), acting as a worst-case technique to compare against.

When evaluating the performance of the baseline as the number of workers is increased, it is expected that the test perplexity does not change due to the equivalence of distributed SGD to the single machine case with zero compression (Appendix A.1). However, it is observed that there is a slight discrepancy in the performance. This is concluded to be due to the floating point precision

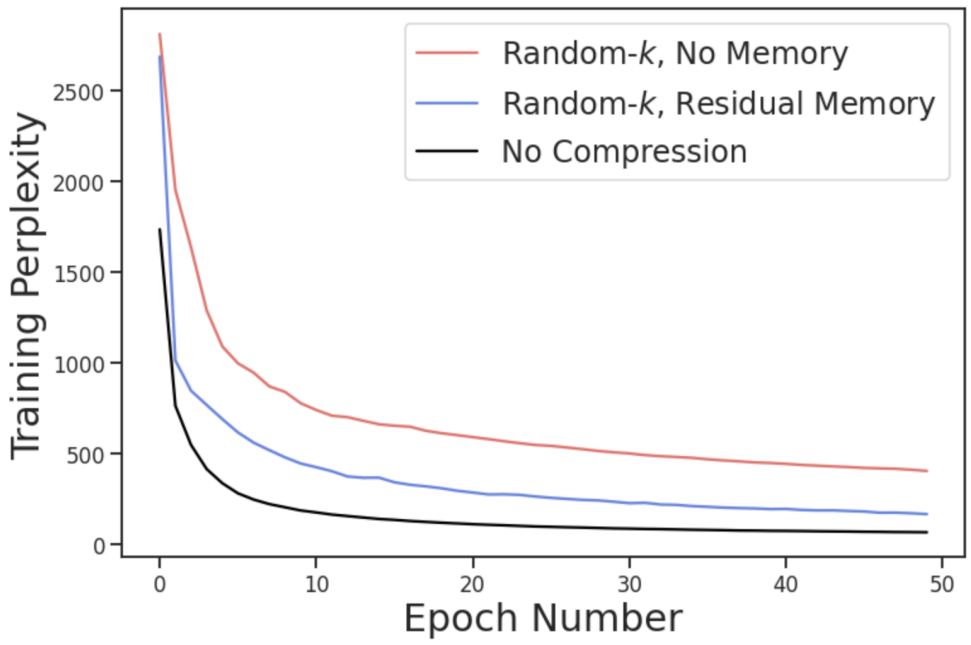

Figure 5.1: Training perplexity tracked over the course of training for the medium sized model, 1 worker. Compression techniques used are zero compression, and Random-k with and without residual error correction.

errors carried forwards through training of models in PyTorch [45]. Exact test perplexity could in fact be achieved through casting of data to the 64-bit `double` and `long` data types and truncating out imprecision, however this greatly impacted training time and decreased performance. When incorporating compression techniques too, this floating point error is found to be negligible.

5.2 Compression vs. Performance

For the medium sized model, the experiment is run for each compression technique across 1, 2 and 4 workers (Figure 5.2). These results show a general trend in decreasing the performance of the model, in line with hypothesis 1. Interestingly, for the 50× compression of Top-k and DGC, improved performance relative to the baseline is observed across all worker numbers. This is the smallest sparsification tested, compared to 500× and 5000×. This result suggests that gradient sparsification, when used more conservatively, can potentially provide a regularising effect to the model when trained (distributed or otherwise). It is important to contrast this with even lower compression ratio achieved by QSGD, which instead can lose model performance dependent on compressed bit-rate. Therefore,

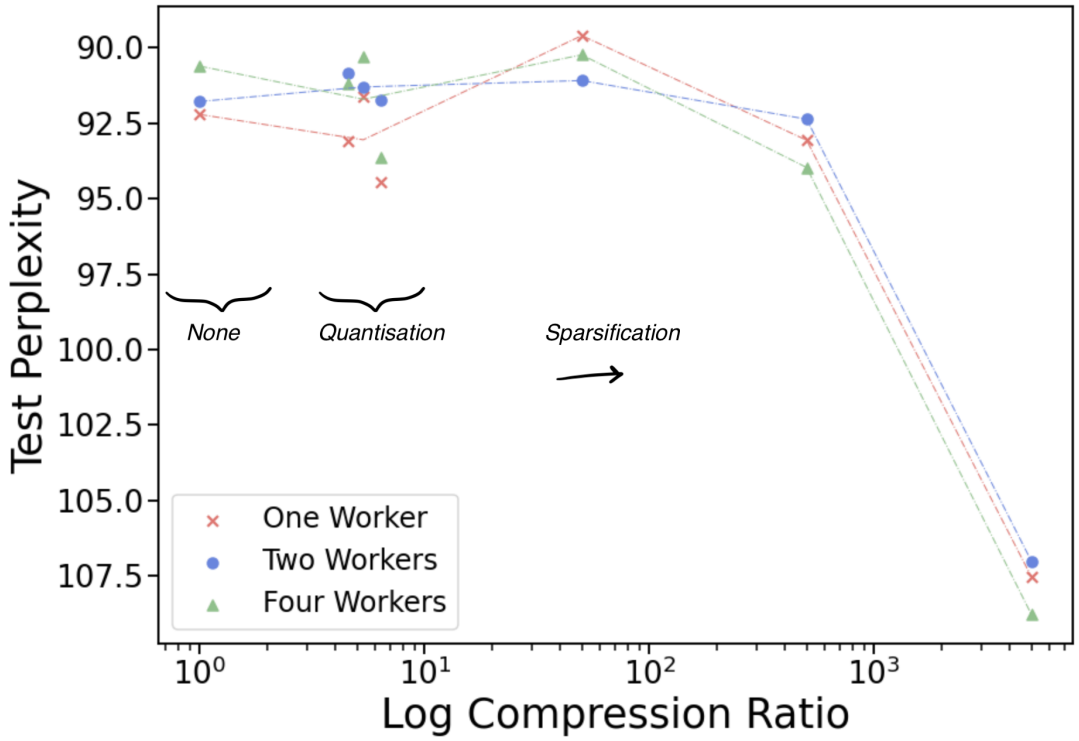

Figure 5.2: Test perplexity vs. compression ratio, on the medium size model, for 1, 2 and 4 workers. Results only show Top-k for sparsification for clarity - analysis between both Top-k and DGC found in Figure 5.4. Equivalent for DGC and small model size found in Appendix B.2.1.

this potential regularisation is not an effect of compression ratio magnitude, but the technique that is used.

This result highlights a key finding in this project, covered further in Section 5.5 - tuning of models on each compression method separately can yield performance which is often better than that quoted in the literature. In the case of DGC, for 462× compression a relative drop of only 0.06 perplexity is reported [35]. However, the researchers retain baseline hyperparameter settings here [1], which suggests promising future work into separate model tuning.

In contrast to hypothesis 6, no strong trend is observed in test performance as worker number is increased (observed additionally for varying model sizes, Appendix B.2.1). The hypothesis is only supported for the case of zero compression as model size changes. This means that scaling of worker number should only be used as an indicator for training run-times, not used for improving model performance.

Figure 5.3 focuses on the sparsification regime of these results. It can be seen

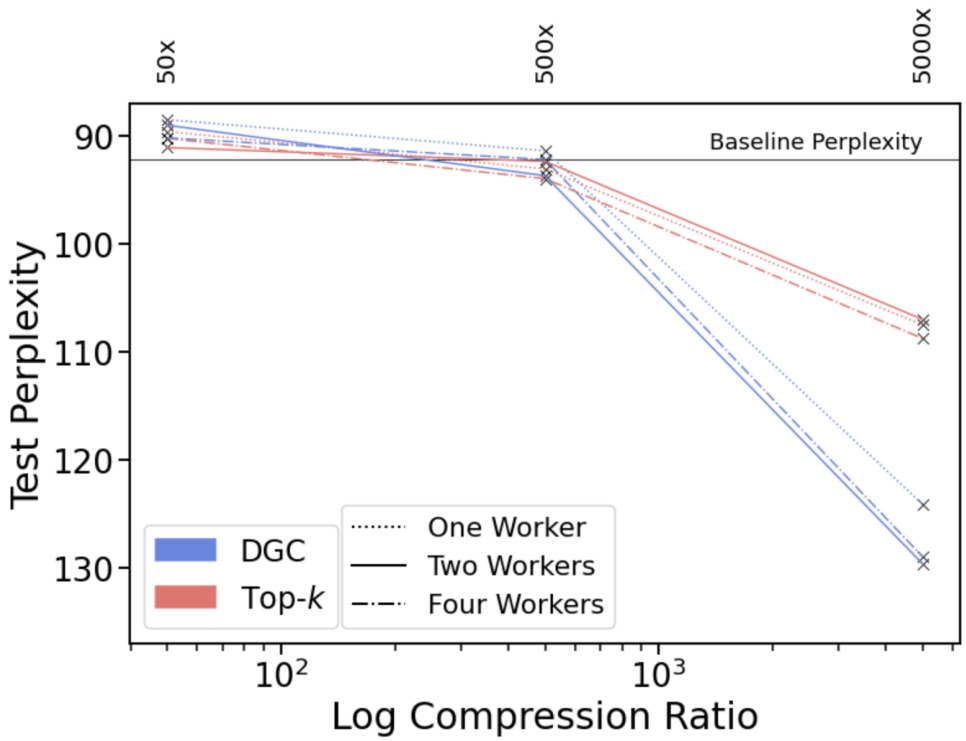

Figure 5.3: Sparsification regime of Figure 5.2, for Top-k and DGC. 1, 2 and 4 worker experiments are plotted for each.

that, for smaller levels of compression, DGC appears to provide better performance, however this performance greatly diverges in favour of Top-k for very high (5000×) compression ratios. This is beyond the quoted literature compression ratio for DGC, at 270-600×. This shows that, while the more advanced DGC algorithm performs great in a lower sparsification regime, performance suffers compared to naive techniques for very high compression. This is despite the additional momentum factor masking to alleviate staleness, suggesting that too many updates were masked in this regime.

Figure 5.4 offers an alternative view on test perplexity and compression trade-off, as the model sizes are varied in a single worker setting. Again, in each case the apparent regularisation is observed for 50× compression. However, as model size is increased, the average improvement above the baseline decreases. This finding suggests that this 'low-sparsification regularisation' effect is most prominent in smaller models. In this respect, performance improvement findings above the baseline are not as helpful when considering the use case of gradient compression is mainly for larger models. On the other hand, this auxiliary finding could open exciting research into the effect of gradient sparsification on improv-

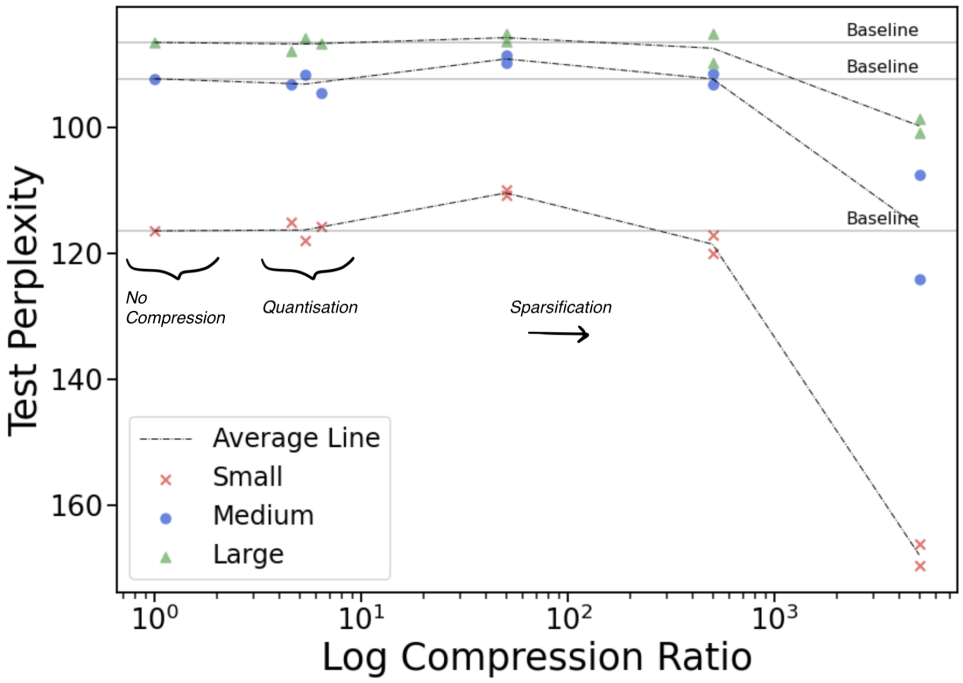

Figure 5.4: Test perplexity vs. compression ratio, for a single worker setting. Both Top-k and DGC plotted (not distinguished). The model sizes range from small to large, at ~ 1, 4 and 9 million parameters respectively.

ing small-model sequence modelling, in a language modelling setting or otherwise.

Further to this, the expected drop-off in performance is observed with increasing compression. Notably, this drop-off is lighter for larger models, supporting hypothesis 5, which could suggest that compression technique performance claims in the literature are also a function of the chosen model size. Specifically, one can achieve negligible losses under higher compression for larger models, and claim a superior technique.

In the regime of quantisation, modest results are observed, with performance on average in-line with baseline results. In the case of medium and larger sized models, it would appear that the 5-bit compression performs best, however this is a weak conclusion to draw, and differences in performance are likely due to the error tolerance provided through the random search procedure.

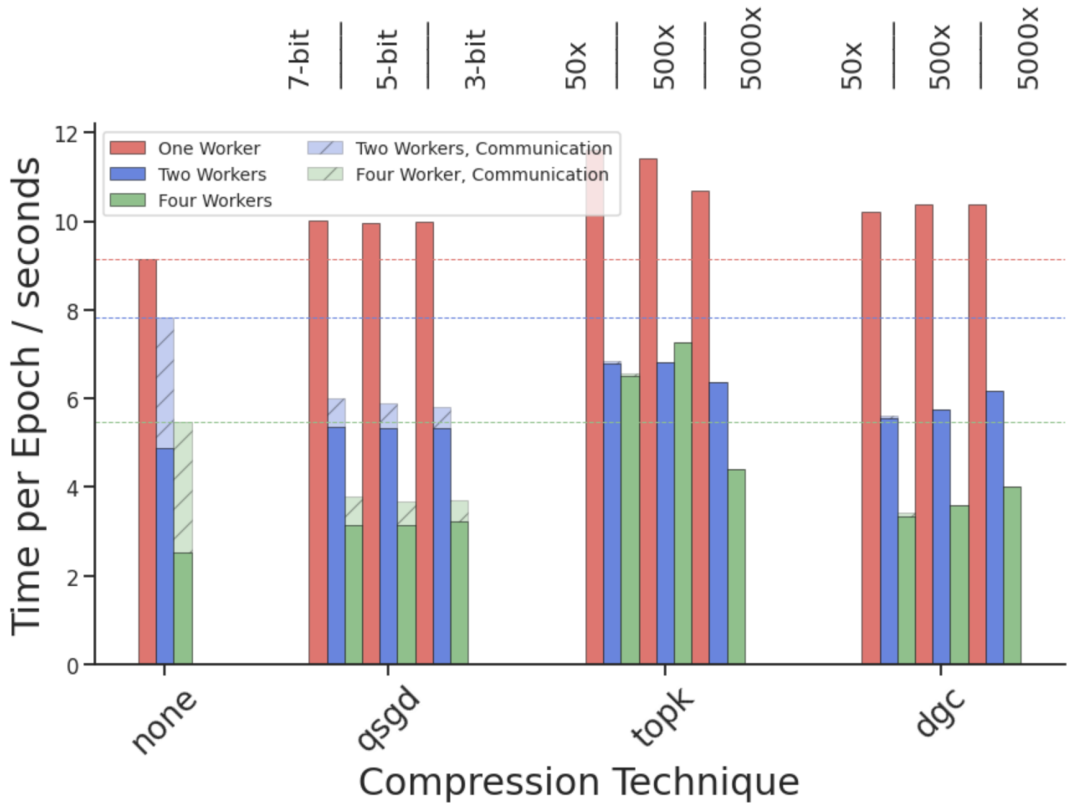

Figure 5.5: Computation and communication time during training of the medium sized model, for each compression technique and ratio analysed. 1, 2 and 4 workers plotted.

5.3 Computation vs. Communication

Running times of the medium size model are compared for a range of worker numbers in Figure 5.5. For each compression technique, the length of computation and communication time is measured and stacked to give an overall running time. Due to the nature of simulating a distributed environment on one machine, communication time is approximated constant for $n > 1$ workers. This ignores practical fluctuations and extra time due to physical communication. Despite this, it can be observed for the baseline case that as the number of workers is increased, the communication time forms a larger proportion of the overall runtime of the model. This reflects the motivation behind communication efficient training (through gradient compression).

It is seen that QSGD adds time to the training process, however this is not a great change, at $\sim 9\%$ average across workers. For the sparsification techniques, hypothesis 2 predicts a much greater increase in computation time. However, this is supported only by results for Top-k; DGC only sees an average increase

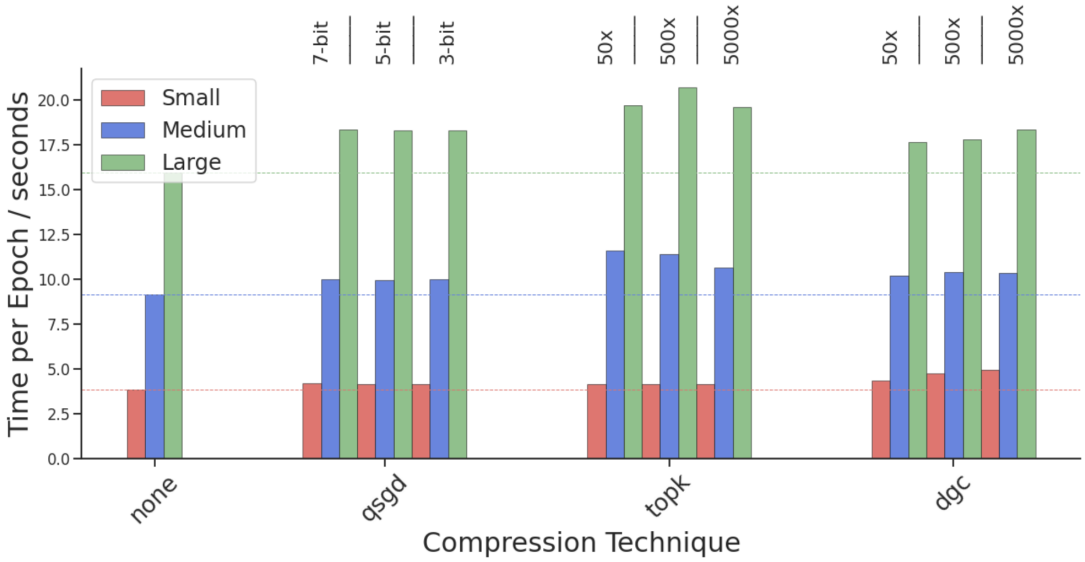

Figure 5.6: Computation time during training of the small, medium and large sized models, for each compression technique and ratio analysed. Data plotted for the single worker case.

of $\sim 13\%$ relative to baseline computation (comparable to QSGD), whereas Top-k has an average $\sim 26\%$ increase. This puts Top-k above total baseline running times in four worker case - the only instance for any of the compression techniques. This can be attributed to the necessary sorting of entire tensor elements during compression, compared to the DGC approach of sampling only $\sim 1\%$ of the tensor before this sorting.

For Top-k specifically, run-times are quoted for the `torch.sort()` method. This approach was found in literature to be faster than usual `torch.topk()` method, and is confirmed to indeed be $\sim 20\%$ faster during testing and experimentation. This highlights implementation details alone greatly impact perceived effectiveness of a compression method.

The lower opacity, hatched bars in Figure 5.5 reflect the simulated communication time in each case, calculated as a bi-directional communication of the model parameters, on a 10Gbps Ethernet connection. Due to its nature of being an estimation, additional material such as headers and error corrections are ignored from the communication time. This speed was chosen to reflect the experimental setting of QSGD, where Amazon EC2 pc2.16xlarge instances are utilised [3]. Each compression method does an effective job of reducing communication relative to the baseline, as can be seen from Figure 5.5. At the scale of this

project, the sparsification methods Top-k and DGC reduce this communication to a negligible proportion of the overall run-time, even at compression ratios of 50×. This demonstrates the effectiveness of each technique in eliminating this run cost.

The effect of increasing model size on the computation time of distributed models using different compression techniques is also investigated, in Figure 5.6. It is observed that, relative to the baseline, QSGD and DGC techniques both scale better than Top-k in terms of additional computation time. Quantitatively, for the large model case, QSGD and DGC add on ~ 3.3 seconds per epoch, whereas Top-k requires up to an additional ~ 5.8 seconds. Compounded with previous findings of poorer worker scaling of Top-k computation relative to other techniques, this makes it a clearly poorer choice when looking to save on model training run-times. Little variation is noted as compression ratio is changed.

Performing distributed training on a simulated, single-GPU system has its drawbacks for run-time analysis. Firstly, the choice of Ethernet speed for communication calculations is both arbitrary, and unable to capture the fluctuations experienced by physical systems. Secondly, the RAM of the T4 model used for computation is 16GB, compared to the additional 732GB of host memory[1] used at the larger scale QSGD was researched under [3]. This additionally has a large impact on the absolute computation time. Due to this, the presented computation-to-communication stacked bar charts are at an arbitrary ratio, chosen to reflect physical settings as closely as possible. In reality, communication times are likely to be slightly higher, and computation times even lower with better utilised memory. This subsection therefore serves to demonstrate scaling trends and compression technique comparisons, not draw absolute conclusions.

5.4 Convergence

Figure 5.7 plots the perplexity evaluated on the model during the course of training, for each compression method under investigation (medium sized model, one worker). It is observed that, relative to the baseline case of zero compression, sparsification techniques Top-k and DGC appear to have a faster rate of convergence initially. On the other hand, QSGD groups up with zero compression. Similar trends are observed in Appendix B.2.2. This result suggests that, un-

[1] https://aws.amazon.com/ec2/instance-types/p2/

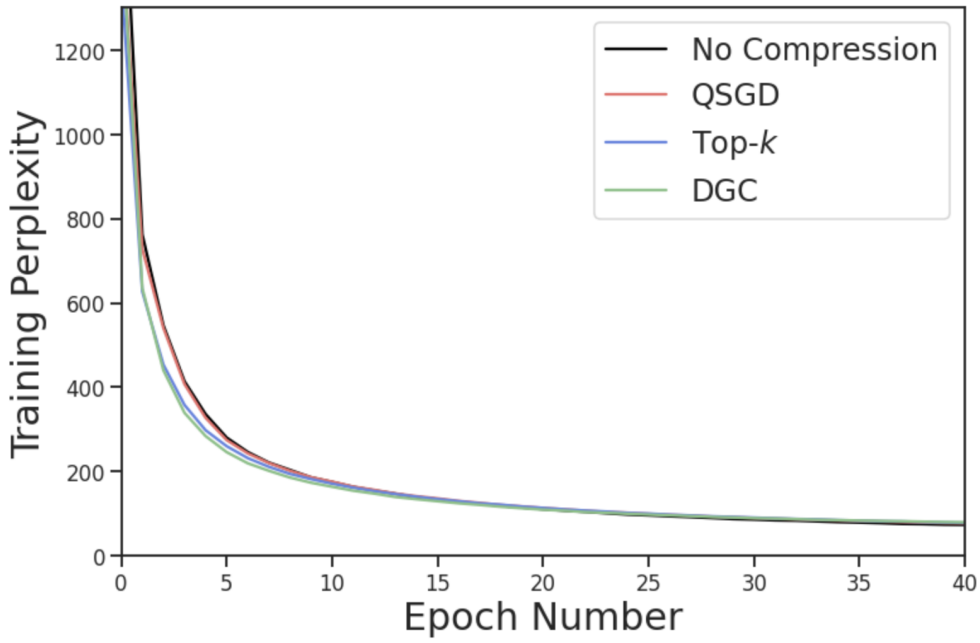

Figure 5.7: Training perplexity tracked over the course of training for the medium sized model, 1 worker. Compression techniques used are zero compression, QSGD, Top-k and DGC.

der a sparsification regime, a distributed model can be trained to convergence (or as close as is desired) in a smaller number of epochs than with no compression. This has direct implications on the compute time as analysed in Section 5.3, since the time savings for a model that converges in fewer epochs are likely to be greater than per-epoch savings. This result could be attributed to the finding of a generally lower learning rate found for optimal sparsification-based distributed models (Section 5.5), which opens up interesting avenues for future work into accelerated training with early stopping on sparsified distributed models.

To further analyse this property of convergence, a chart of the epochs for early-stopping is presented in Figure 5.8, for the fine-grained compression ratio options of each technique (medium sized model, one and two workers). This metric is hindered by a late bug discovered in the project codebase, where early stopping was implemented incorrectly to increment for any non-improvement in validation perplexity during training (as opposed to a streak of non-improvements). This is not expected to greatly impact performance results, since the best-case epoch hovering around 45 would suggest a patience parameter of 5, which is not uncommon. However, conclusions on convergence variability for hypothesis 3 are tainted by this.

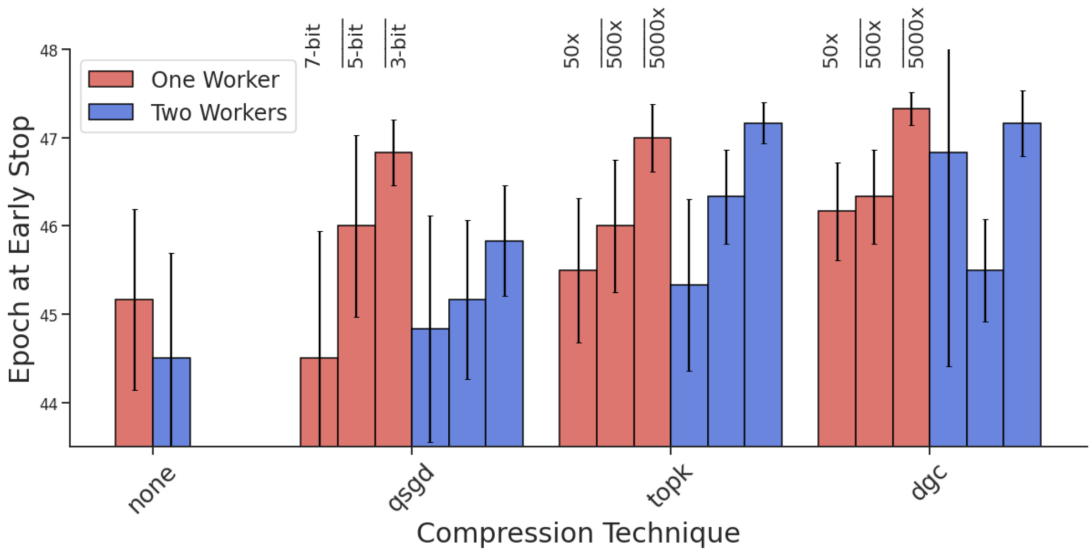

Figure 5.8: Grouped bar chart of the average epoch that the medium size model converges at using early stopping, for both the 1 and 2 worker settings. Trends further grouped according to the compression technique used.

Nonetheless, across each technique a trend is uncovered which demonstrates that, for increased compression, the average epoch for early stopping gets later. This result goes hand in hand with the finding of accelerated convergence for sparsification-based distributed models above, to further emphasise the trade-off one is paying in computation time for a higher compression ratio.

Additionally graphed are the standard-mean error bars on 'early-stopped epoch'. Due to constraining the number of training epochs and the aforementioned training bug with early stopping, no clear trend in variance is seen here. With increased resources, future work in addressing this metric would be interesting, to analyse the claims on performance-convergence variance trade-off in QSGD, and to see if this theory applies for sparsification techniques.

5.5 Hyperparameter Choices

In the range of research literature on gradient compression techniques (and in particular for those surveyed in this project), the learning rate chosen for training is always kept the same as the default baseline (zero compression) setting [35, 3]. Figure 5.9 shows the spread of learning rates observed for each compression technique, for each of one, two and four workers when the learning rate is

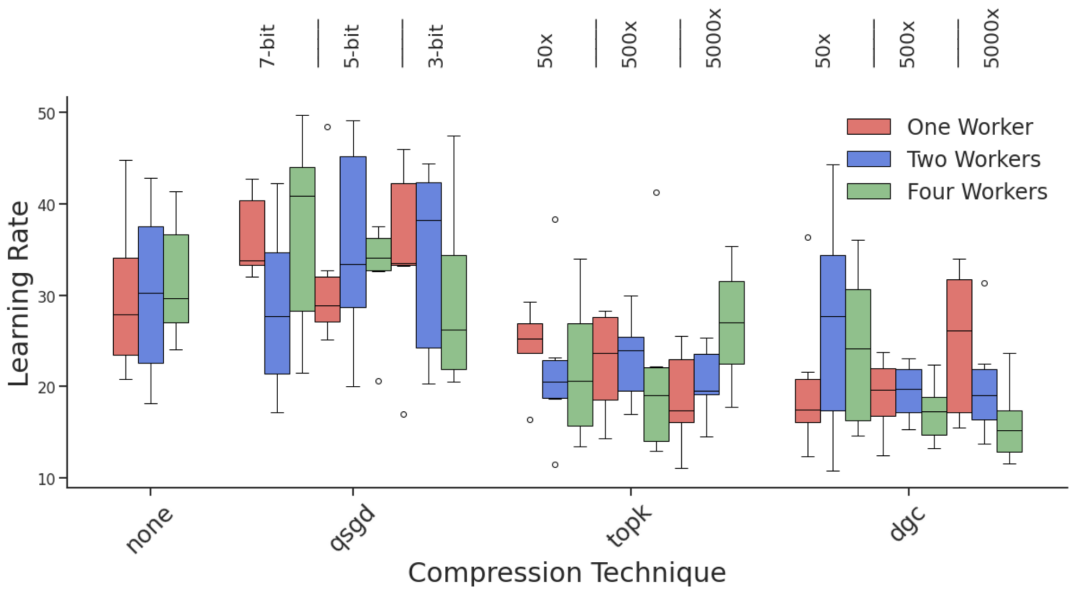

Figure 5.9: Box plot representing the range of learning rate values adopted for each compression method, across 1, 2 and 4 workers. Model size visualised is the medium model.

instead tuned for. It is observed that, especially when comparing quantisation to sparsification techniques, there is a noticeable difference in magnitude across the average spread of best-performing learning rates. Top-k and DGC methods tend to adopt lower learning rates for optimal performance, whereas the zero compression and QSGD methods adopt higher values. No clear trend is observed for changing worker number. As referenced in Section 5.4, a lower learning rate could also reflect the trend in accelerated convergence for these sparsification techniques - a property which has not been explored previously and opens up interesting avenues into compression technique optimisation.

A further hyperparameter that is analysed is the dropout values, visualised for one, two and four workers on the medium size model in Figure 5.10. Although no clear trend appears when increasing compression ratio in QSGD, a steady decrease appears in optimal dropout value in the sparsification regime, particularly for the two worker case. This would make sense, since there are fewer gradient elements being selected as the compression ratio is increased, and so a lower dropout is preferable to avoid culling too many important 'heavy-hitting' gradient elements. It would also appear that dropout rate is less sensitive when using no compression or QSGD, perhaps making these approaches more stable for tuning.

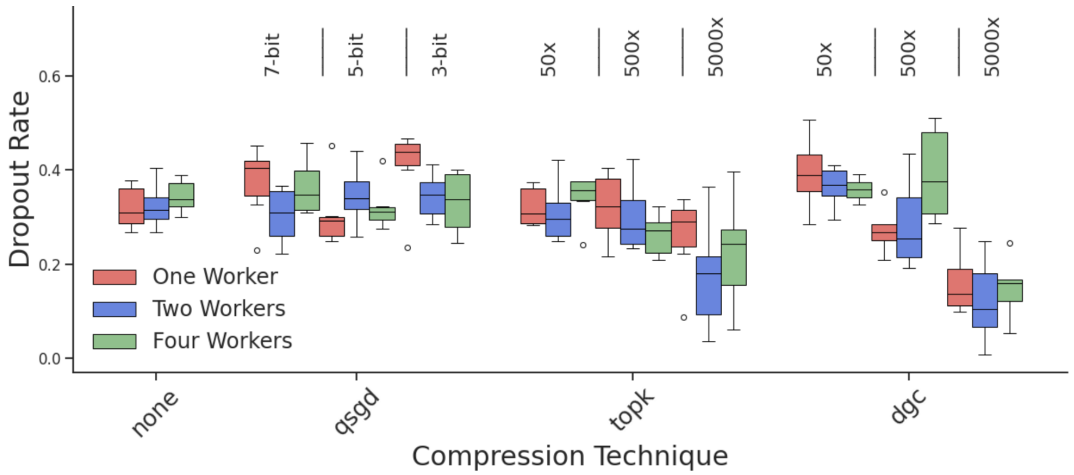

Figure 5.10: Box plot representing the range of dropout values adopted for each compression method, across 1, 2 and 4 workers. Model size visualised is the medium model.

These findings again reflect the need for more careful tuning of distributed models when performing a fair analysis of compression techniques relative to a baseline. In practice, the purpose of compression techniques is to make the training of complex models feasible in the first place, and so tuning would often be performed on a model when gradient compression is involved - by encouraging research that incorporates hyperparameter tuning additionally on non-baseline models, reported results are both more true to practice, and give fairer opportunity for a technique to perform optimally.

Chapter 6

Conclusions

In this work, the effect of gradient compression in distributed deep learning is investigated. This is motivated by the wealth of existing research into reducing the communication time between workers in a distributed environment for training of large-scale models, with few resources effectively comparing techniques in a fair and equal environment.

Under the constraints of the project, three prominent compression techniques from the literature are selected and surveyed, which cover a representative range of top performing methods; Top-k and Deep Gradient Compression (DGC), which are sparsification techniques, and Quantised SGD (QSGD), a quantisation technique. This investigation is performed through the lens of language modelling, due to its low computation-to-communication ratio (making the problem of communication bottlenecks more pronounced), and the relative lack of experimentation in gradient compression for distributed deep learning in this domain.

Given the range of metrics and experimental settings possible in this field, findings are split according to each metric recorded, which provides a holistic view on the merits and drawbacks of each compression technique in a localised sense. Although ideally an objectively best compression technique would be determined, this conclusion instead aims to draw together the localised findings of the project work, providing pros and cons that help inform the practitioner reading this in their own work.

Contrary to the hypothesis of generally decreased performance of distributed training (as per test perplexity) as compression ratio is increased, it is found that across different numbers of workers (GPUs), even improved performance relative to a baseline is possible at modest compression rates. This is found for

sparsification techniques Top-k and DGC, however modest improvements can even be found under certain settings for QSGD. It is believed this is because of a different (and more fair) adaption of the methodology compared to literature, in tuning each individual model's hyperparameters. Motivation for doing this is reflected in the generally lower spread found in optimal learning rates for sparsification techniques. Despite this, these improvements are observed to generally diminish in higher model size regimes, perhaps reflecting likewise-diminishing returns for tuning models at scale in research.

Compression rates are, by design, much higher for Top-k and DGC than QSGD, however, investigation into overall running costs of distributed models trained using each method shows this is not always necessarily positive. Particularly for Top-k, the additional computation time associated with running the compression can be found to outweigh savings in communication time, as both the worker number and model size is scaled. On the other hand, QSGD achieves more limited compression ratios, but has the advantage of more modest additional computation times.

In tandem with trends displaying earlier convergence both for sparsification methods, and for smaller compression ratios for a given compression method, it is concluded that there is no one right choice of method to use. Choice of model size and worker number affect running costs, but so do many other factors, including hardware-dependent GPU power and communication speed (such as Ethernet). Picking the right compression technique to use is dependent both on resources available, and the experimental setting of the model - for larger scale models, for example, the gain in compression may have to be foregone in favour of a less computationally heavy method. One finding that is important from this work, however, is that it is beneficial to tune for each model type, and not assume baseline settings.

6.1 Drawbacks and Future Work

Limitations with regards to project resources, and resulting design decisions, create some disadvantages which reduce the quality of the work on a larger, research scale. Firstly, usage of free cloud services restrict the scale at which models can be trained, meaning results fail to reflect those in the literature in an absolute sense. Secondly, restrictions in budget lead to the production of a novel,

single-machine, multi-worker simulation of a distributed training environment, which fails to faithfully capture the minutia (especially in running costs and times) of training a truly distributed system.

Despite these drawbacks, results remain accurate in a relative sense, and there is merit to analysing trends at a smaller scale. The methodology that allows for this smaller scale distributed training abstracts away the difficult details of practical distributed model training, and opens up options for investigating performance of these methods to a wider research audience.

It is suggested that, in future work, the scope of investigation is scaled down to allow for more targeted conclusions, for example with regards to only a changing number of workers. This will prevent thinning of resources and help focus results, even on a wider range of compression techniques.

With access to more resources, a number of exciting avenues can be extended from the findings of this paper. Firstly, full convergence analysis of compression techniques relative to each other appears promising, even in the space of how this changes with varying compression ratio. This trade-off is observed for just QSGD in the literature, and not fully analysed in the current work due to a bug with early stopping, and restricted number of epochs.

Secondly, analysis covering a wider range of language model types, including the ever-increasingly popular family of transformer models, would provide a comparative survey not currently carried out in the field of gradient compression. As research trends point towards growing language model sizes in NLP, researching effective methods for reducing training costs of these models will become vital.

Bibliography

[1] Openreview page: Deep gradient compression: Reducing the communication bandwidth for distributed training, February 2018. Available at https://openreview.net/forum?id=SkhQHMW0W, accessed: 2020-08-04.

[2] Alham Fikri Aji and Kenneth Heafield. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021*, 2017.

[3] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, pages 1709–1720, 2017.

[4] Debraj Basu, Deepesh Data, Can Karakus, and Suhas Diggavi. Qsparse-local-sgd: Distributed sgd with quantization, sparsification and local computations. In *Advances in Neural Information Processing Systems*, pages 14695–14706, 2019.

[5] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

[6] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[7] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012.

[8] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Anima Anandkumar. signsgd: Compressed optimisation for non-convex problems. *arXiv preprint arXiv:1802.04434*, 2018.

[9] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[10] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. Adacomp: Adaptive residual gradient compression for data-parallel distributed training. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[11] Dan Claudiu Cireşan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, 22(12):3207–3220, 2010.

[12] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In *International conference on machine learning*, pages 1337–1345, 2013.

[13] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[14] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[15] Tim Dettmers. 8-bit approximations for parallelism in deep learning. *arXiv preprint arXiv:1511.04561*, 2015.

[16] Nikoli Dryden, Tim Moon, Sam Ade Jacobs, and Brian Van Essen. Communication quantization for data-parallel training of deep neural networks. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, pages 1–8. IEEE, 2016.

[17] Aritra Dutta, El Houcine Bergou, Ahmed M Abdelmoniem, Chen-Yu Ho, Atal Narayan Sahu, Marco Canini, and Panos Kalnis. On the discrepancy between the theoretical analysis and practical implementations of compressed communication for distributed deep learning. *arXiv preprint arXiv:1911.08250*, 2019.

[18] Peter Elias. Universal codeword sets and representations of the integers. *IEEE transactions on information theory*, 21(2):194–203, 1975.

[19] Jiarui Fang, Haohuan Fu, Guangwen Yang, and Cho-Jui Hsieh. Redsync: Reducing synchronization traffic for distributed deep learning. *arXiv preprint arXiv:1808.04357*, 2018.

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[21] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[22] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015.

[23] Alon Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, 2009.

[24] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[25] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[26] Torsten Hoefler, William Gropp, Rajeev Thakur, and Jesper Larsson Träff. Toward performance models of mpi implementations for understanding ap-

plication scaling issues. In *European MPI Users' Group Meeting*, pages 21–30. Springer, 2010.

[27] Nikita Ivkin, Daniel Rothchild, Enayat Ullah, Ion Stoica, Raman Arora, et al. Communication-efficient distributed sgd with sketching. In *Advances in Neural Information Processing Systems*, pages 13144–13154, 2019.

[28] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[29] Daniel Jurasfky and James H Martin. An introduction to natural language processing, computational linguistics, and speech recognition, 2000.

[30] Sai Praneeth Karimireddy, Quentin Rebjock, Sebastian U Stich, and Martin Jaggi. Error feedback fixes signsgd and other gradient compression schemes. *arXiv preprint arXiv:1901.09847*, 2019.

[31] Jack Kiefer, Jacob Wolfowitz, et al. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3):462–466, 1952.

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

[33] John Langford, Lihong Li, and Tong Zhang. Sparse online learning via truncated gradient. In *Advances in neural information processing systems*, pages 905–912, 2009.

[34] Youjie Li, Jongse Park, Mohammad Alian, Yifan Yuan, Zheng Qu, Peitian Pan, Ren Wang, Alexander Schwing, Hadi Esmaeilzadeh, and Nam Sung Kim. A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–188. IEEE, 2018.

[35] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[36] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: a rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 41–54, 2018.

[37] Luo Mai, Chuntao Hong, and Paolo Costa. Optimizing network performance in distributed machine learning. In *7th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

[38] Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. 1993.

[39] Ruben Mayer and Hans-Arno Jacobsen. Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools. *ACM Computing Surveys (CSUR)*, 53(1):1–37, 2020.

[40] Ruben Mayer, Christian Mayer, and Larissa Laich. The tensorflow partitioning and scheduling problem: it's the critical path! In *Proceedings of the 1st Workshop on Distributed Infrastructures for Deep Learning*, pages 1–6, 2017.

[41] Gábor Melis, Chris Dyer, and Phil Blunsom. On the state of the art of evaluation in neural language models. *arXiv preprint arXiv:1707.05589*, 2017.

[42] Konstantin Mishchenko, Eduard Gorbunov, Martin Takáč, and Peter Richtárik. Distributed learning with compressed gradient differences. *arXiv preprint arXiv:1901.09269*, 2019.

[43] Goncalo Mordido, Matthijs Van Keirsbilck, and Alexander Keller. Monte carlo gradient quantization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 718–719, 2020.

[44] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2(11):1–4, 2015.

[45] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[46] Ofir Press and Lior Wolf. Using the output embedding to improve language models. *arXiv preprint arXiv:1608.05859*, 2016.

[47] Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873–880, 2009.

[48] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[49] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[50] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[51] Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. Sparsified sgd with memory. In *Advances in Neural Information Processing Systems*, pages 4447–4458, 2018.

[52] Nikko Strom. Scalable distributed dnn training using commodity gpu cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[53] Zhenheng Tang, Shaohuai Shi, Xiaowen Chu, Wei Wang, and Bo Li. Communication-efficient distributed deep learning: A comprehensive survey. *arXiv preprint arXiv:2003.06307*, 2020.

[54] Yusuke Tsuzuku, Hiroto Imachi, and Takuya Akiba. Variance-based gradient compression for efficient distributed deep learning. *arXiv preprint arXiv:1802.06058*, 2018.

[55] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[56] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[57] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. Powersgd: Practical low-rank gradient compression for distributed optimization. In *Advances in Neural Information Processing Systems*, pages 14259–14268, 2019.

[58] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 381–394, 2015.

[59] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*, pages 1509–1519, 2017.

[60] Paul J Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural networks*, 1(4):339–356, 1988.

[61] Thomas Wolf. Medium article: Training neural nets on larger batches: Practical tips for 1-gpu, multi-gpu distributed setups, October 2018. Available at `https://medium.com/huggingface/training-larger-batches-practical-tips-on-1-gpu-multi-gpu-distributed-setups-ec88c3e51255`, accessed: 2020-04-14.

[62] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey,

et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[63] Hang Xu, Chen-Yu Ho, Ahmed M Abdelmoniem, Aritra Dutta, El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini, and Panos Kalnis. Compressed communication for distributed deep learning: Survey and quantitative evaluation. Technical report, 2020.

[64] Jun Yang, Yan Chen, Siyu Wang, Lanbo Li, Chen Meng, Minghui Qiu, and Wei Chu. Practical lessons of distributed deep learning. In *Workshop on Principled Approaches to Deep Learning, at ICML*, 2017.

Appendix A

Theoretical Background

A.1 Distributed Optimisation Problem

The following provides an outline for the theoretical step from non-distributed deep learning stochastic gradient descent (SGD) to parallel distributed SGD, taking notation and cues from literature reviews [53, 63].

Optimisation in deep learning takes the form of the minimisation of a cost function $J(\mathbf{x})$, and is formulated as:

$$\min_{\mathbf{x},y\sim D}\Big[J(\mathbf{x}) := \mathbb{E}_{\boldsymbol{x},y}[L(h(\boldsymbol{x}),y)]\Big] \quad = \quad \min_{\mathbf{x},y\sim D}\frac{1}{B}\sum_{i=1}^{B}L(h(\boldsymbol{x}_i),y_i) \tag{A.1}$$

For B (batch size) samples drawn drawn i.i.d from distribution D (the dataset), and model feed-forward function $h(\cdot)$. $\mathbb{E}_{\mathbf{x},y}$ is an expectation under these samples, here approximated by a sample average, and $L(\cdot)$ a loss function, such as mean squared error or cross-entropy loss.

Stochastic gradient descent [31], a first-order gradient descent method for solving this optimisation problem, has a simple update rule which uses the gradient of the loss function, $\nabla L(\cdot)$, to push model parameters $\mathbf{p}$ in a direction towards a local optimum, scaled by a learning rate γ:

$$\mathbf{p}_{k+1} = \mathbf{p}_k - \gamma\frac{1}{b}\sum_{i=1}^{b}\nabla L(h(\boldsymbol{x}_i),y_i) \tag{A.2}$$

For mini-batch size b.

When using multiple workers distributed in a data-parallel environment, the optimisation problem A.1 changes to the minimisation of the mean cost function

of all n workers, where worker i has loss function L_i, and access to a partition D_i of the dataset:

$$\min_{\mathbf{x},y\sim D_i} \frac{1}{n}\sum_{i=1}^{n} \mathbb{E}_{\boldsymbol{x},y}[L_i(h(\boldsymbol{x}),y)] \tag{A.3}$$

The most common adaption to the SGD algorithm of A.2 is bulk synchronous parallel SGD, or BSP-SGD [55]:

$$\mathbf{p}_{k+1} = \mathbf{p}_k - \gamma \frac{1}{n}\sum_{i=1}^{n} \nabla L_i\left(h\left(\boldsymbol{x}\right),y\right) \tag{A.4}$$

This adaption simply takes an average of the gradient updates for a parameter from each worker. In a practical setting, this is equivalent to normalising each gradient update by $\frac{1}{n}$. The reason this works is because common loss functions are reduced by an average across the mini-batch (and additionally sequence length for language models, Section 2.3), and in moving to a data-parallel setting the batch size is split and so the calculated loss requires re-normalisation. Under the hood, the update term in A.4 becomes:

$$\frac{1}{n}\frac{1}{(\frac{b}{n})}\sum_{i=1}^{n}\sum_{j=1}^{b} \nabla L_i\left(h\left(\boldsymbol{x}_j\right),y_j\right) = \frac{1}{b}\sum_{j=1}^{b} \nabla L\left(h\left(\boldsymbol{x}_j\right),y_j\right) \tag{A.5}$$

Recovering the update in the original SGD equation A.2, for each worker possessing identical loss function L_i.

A.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) are a family of neural networks specialised for processing sequential data $\mathbf{x}^{(t)}$, indexed by timestep t. To do this, parameter sharing across timesteps is employed, to allow for arbitrary length inputs to a model and generalisation across them [20]. This parameter sharing is achieved through a recurrence relation, where the output of one timestep is a function of the output of a previous timestep, which results in the maintenance of an internal state over time able to model history observed in the sequence. This tackles the fixed context length barrier observed in statistical language modelling approaches such as n-grams.

Recurrent units typically learn an intermediate hidden state $\mathbf{h}^{(t)}$, which models the summary of sequence inputs up to time t, and is computed as:

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \theta) = f\left(W_{hh}\mathbf{h}^{(t-1)} + W_{hx}\mathbf{x}^{(t)} + b_h\right) \tag{A.6}$$

for current input $\mathbf{x}^{(t)}$, output at previous timestep $\mathbf{h}^{(t-1)}$, and feedforward and recurrent learned weight matrices W_{hh} and W_{hx} respectively. f represents a non-linear activation function such as $\tanh(\cdot)$. To learn from these systems, unrolling of a recurrent computation over its timesteps can be performed to form a feedforward computational graph. This allows for the standard learning procedure via gradient backpropagation, and is called backpropagation through time (BPTT) [49, 60].

Although in theory this method allows for learning of long-range dependencies in sequence/language modelling, the recursive use of a non-linear activation function (and repeated multiplication of many Jacobians) results in vanishing or exploding gradients over extended timesteps [6]. To combat this, gated RNNs were introduced with the idea of creating paths where gradients would neither vanish nor explode. The long-short term memory (LSTM) cell [25] uses gated units with internal recursion to control the flow of information that is passed through the network.

A.3 Calculating the Perplexity

Derivation of perplexity from the cross entropy loss between target and predicted sequence tokens, as is commonly performed in practice. For a given sequence $w_1 w_2 ... w_N$, the target distribution over $\tilde{P}(t_n)$ defines the probability of token w_n in the test set, and the prediction distribution $P(t_n)$ represents the probability of token w_n given the context of tokens before it $w_1 w_1 ... w_{n-1}$, modelled through a language model.

Starting with the definition of perplexity:

$$PP(\mathbf{s}) = \sqrt[N]{\frac{1}{P(w_1 w_2 ... w_N)}} = \prod_{n=1}^{N} P(w_n | w_{1:n-1})^{-\frac{1}{N}} = \prod_{n=1}^{N} P(t_n)^{-\frac{1}{N}} \tag{A.7}$$

Through exponentiation this becomes:

$$PP(\mathbf{s}) = e^{-\sum_{n=1}^{N} \frac{1}{N} log(P(t_n))} \tag{A.8}$$

Despite the strong assumption [29], the target distribution $\tilde{P}(t_n)$ is now assumed to follow an empirical distribution, such that the cross-entropy is defined in the limit of large sequence lengths as:

$$H(\mathbf{s}) = -\sum_{n=1}^{N} \frac{1}{N} log(P(t_n)) \tag{A.9}$$

arriving at the perplexity, defined and calculated as:

$$PP(\mathbf{s}) = e^{H(\mathbf{s})} \tag{A.10}$$

Appendix B

Auxiliary Results

B.1 Data

On the following two pages are data tables for two of the primary collected data metrics: test perplexity and computation time. Due to the cross-variable nature of the table, no results are highlighted in particular to show comparisons - these tables serve simply as reference points.

B.1.1 Test Perplexity

Table B.1: Table of data for the test perplexity, across the range of variables investigated. For each run type, the result quoted is the best run (measured by lowest test perplexity) of the 60 run random search procedure (Section 4.2.1).

Model Size / Number of Workers		**Compression Technique**								
		No Compression	Top-k			DGC		QSGD		
		1x	50x	500x	5000x	50x	500x	5000x	7-bit	5-bit	3-bit
Small	1	116.398	110.814	120.008	169.623	109.912	117.002	166.213	115.023	117.987	115.717
	2	115.113	114.365	117.742	151.729	111.001	120.002	160.590	118.521	114.993	115.133*
Medium	1	92.213	89.597	93.066	107.530	88.504	91.387	124.156	93.109	91.608	94.436
	2	91.781	91.088	92.367	107.013	89.020	93.694	129.669	90.848	91.325	91.727
	4	90.611	90.236	93.969	108.778	90.193	92.195	129.006	91.178	90.313	93.634
Large	1	86.427	86.302	89.600	98.583	85.045	85.008	100.809	87.766	85.760	86.539

B.1.2 Computation Time

Table B.2: Table of data for the average computation time, per epoch, of model training, across the range of variables investigated. For each run type, the result quoted is the average of the top 8 runs (measured by lowest test perplexity) of the 60 run random search procedure (Section 4.2.1).

Model Size / Number of Workers		**Compression Technique**									
		No Compression	Top-k			DGC			QSGD		
		1x	50x	500x	5000x	50x	500x	5000x	7-bit	5-bit	3-bit
Small	1	3.862	4.158	4.158	4.144	4.375	4.733	4.931	4.213	4.141	4.141
	2	2.197	2.612	2.612	2.557	2.864	3.271	3.310	2.449	2.426	2.449*
Medium	1	9.159	11.620	11.417	10.685	10.204	10.392	10.382	10.028	9.973	9.998
	2	4.875	6.799	6.813	6.374	5.544	5.752	6.163	5.347	5.330	5.344
	4	2.521	6.501	7.268	4.400	3.352	3.593	4.004	3.137	3.133	3.23
Large	1	15.968	19.7228	20.724	19.608	17.656	17.801	18.363	18.361	18.332	18.333

B.2 Plots

On the remaining pages of Appendix B are auxiliary plots which support results and conclusions drawn in the main text. These are for the interested reader only, and analysis does not go beyond that of Chapter 5.

B.2.1 Compression vs. Performance

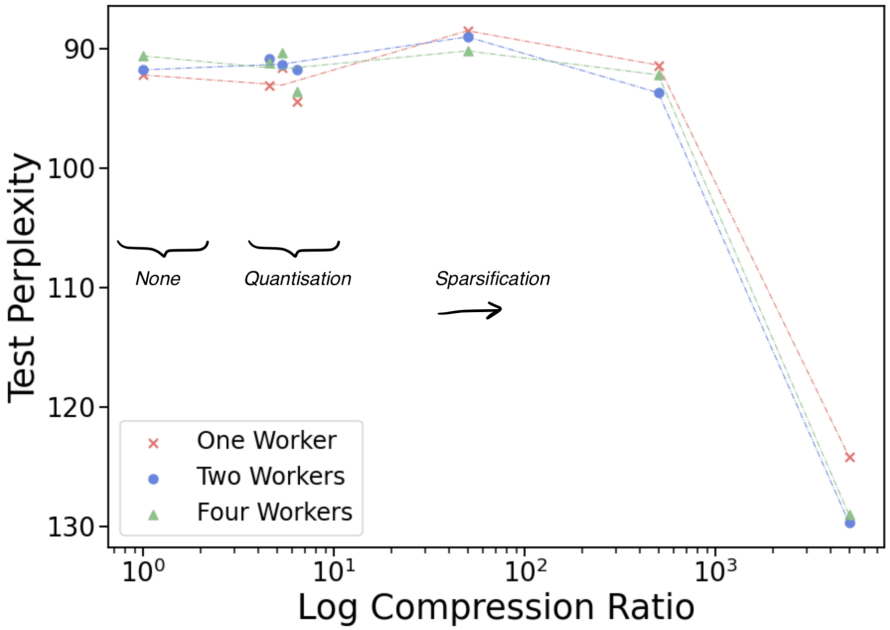

Figure B.1: Test perplexity vs. compression ratio, on the medium size model, for 1, 2 and 4 workers. Results only show DGC for sparsification for clarity. Much like Figures 5.2 and B.3, no clear trend is observed for increasing worker number, however a regularising effect is seen at $50\times$ compression.

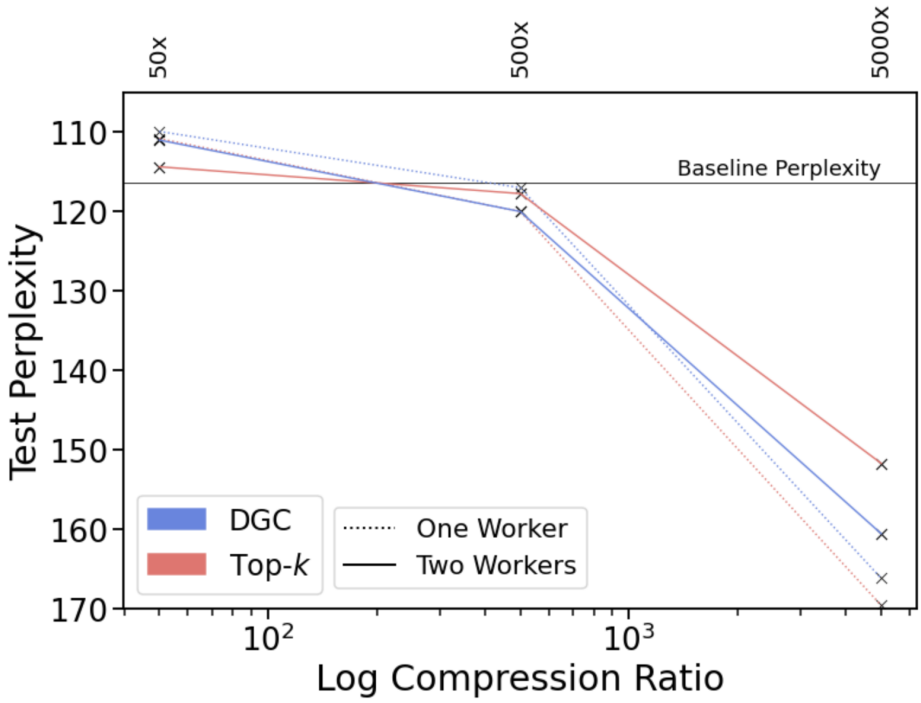

Figure B.2: Sparsification regime of Figure B.3, for Top-k and DGC. 1, 2 and 4 worker experiments are plotted for each.

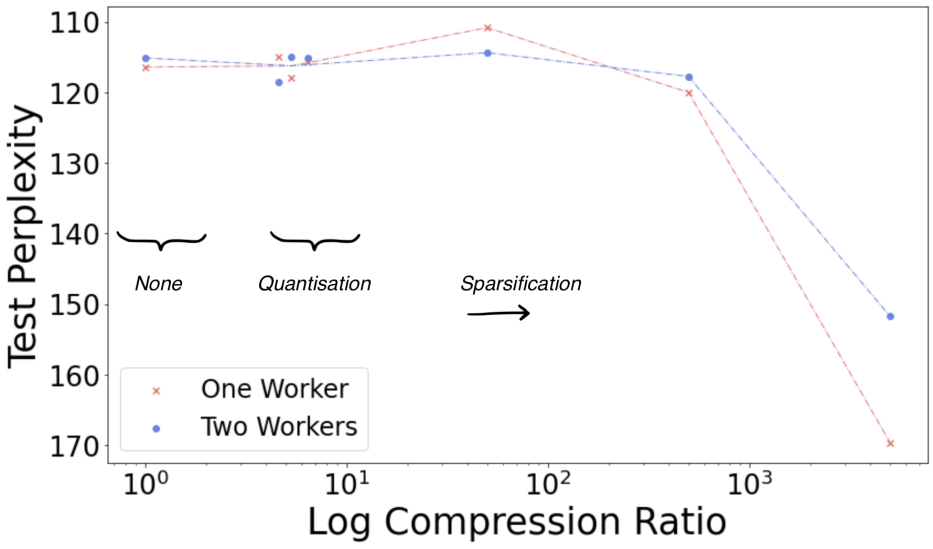

Figure B.3: Test perplexity vs. compression ratio, on the small size model, for 1 and 2 workers. Results only show Top-k for sparsification for clarity. Much like Figures 5.2 and B.1, no clear trend is observed for increasing worker number, however a regularising effect is seen at $50\times$ compression.

B.2.2 Convergence

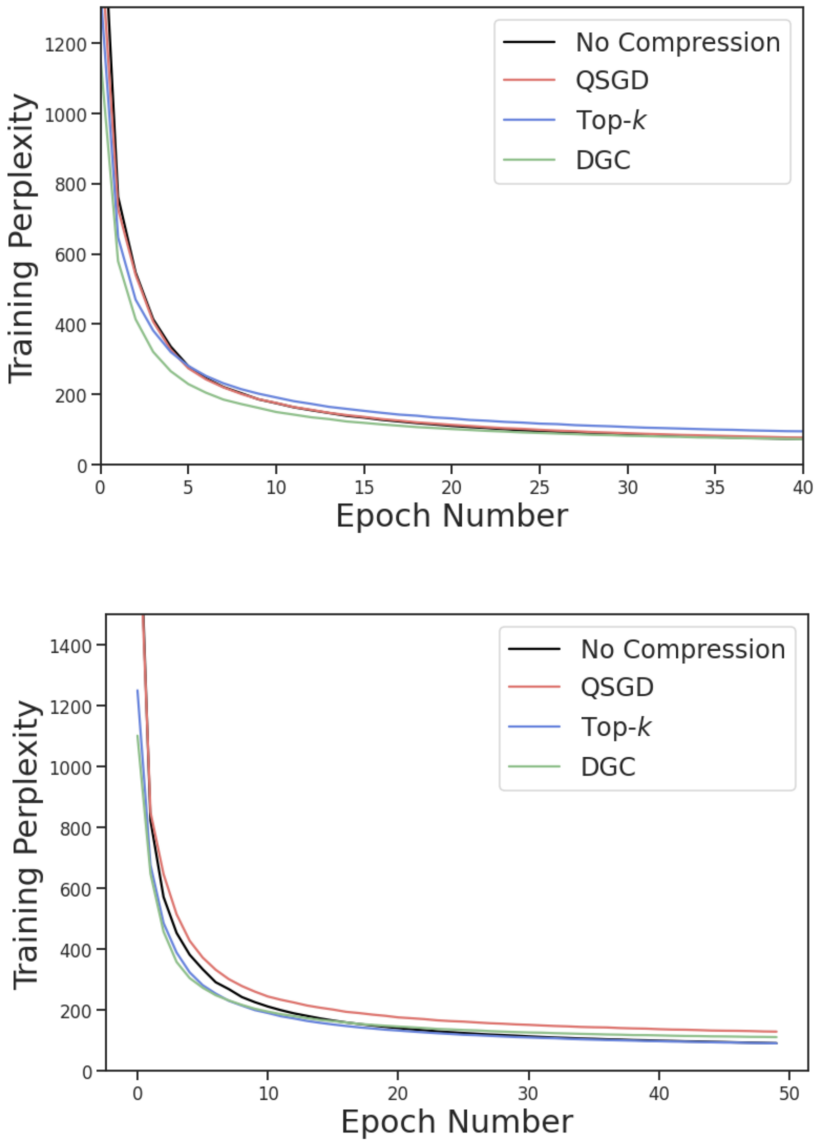

Figure B.4: Training perplexity tracked over the course of training for the medium size model, 2 workers (*top*); and small size model, 1 worker (*bottom*). Compression techniques used are zero compression, QSGD, Top-k and DGC. It is observed that the two sparsification techniques appear to convergence at a faster rate, with regards to epoch number.

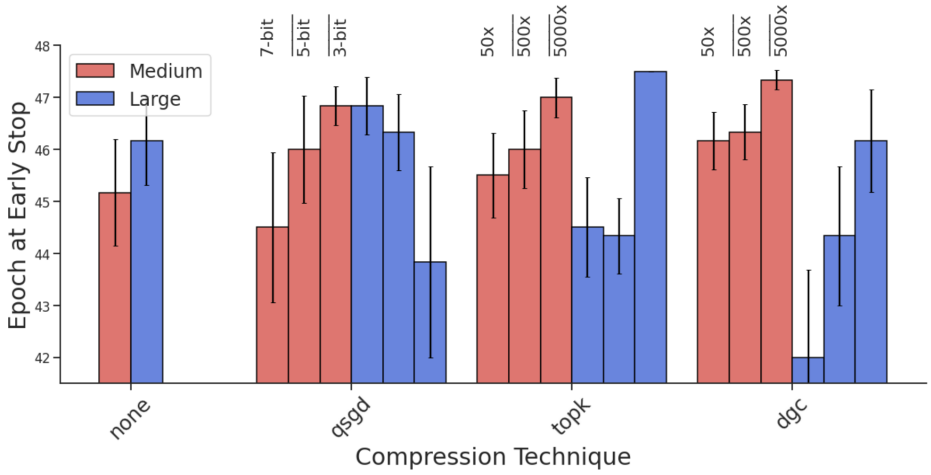

Figure B.5: Grouped bar chart of the average epoch that the medium and large models converge at using early stopping, for the single worker setting. Trends further grouped according to the compression technique used. The general trend of earlier convergence for lower compression ratios is still observed, besides some differences for the case of larger models.

B.2.3 Hyperparameter Choices

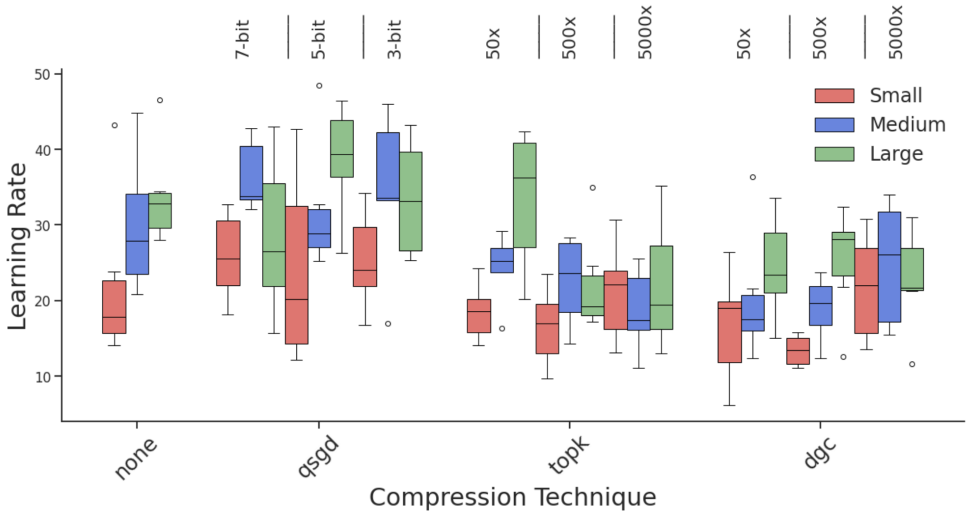

Figure B.6: Box plot representing the range of learning rate values adopted for each compression method, across the small, medium and large model sizes. Visualised for the single worker case. Weaker trend towards sparsification having a smaller learning rate is observed in the case of growing model size.

Appendix C

Project Planning

On the following page is an updated version of the Gantt chart[1] from this project's original Informatics Project Proposal document. Original planned timeline has boxes with no outline, and the final time spent on the respective task sits just below with a red outline. The work has been split into five sections, which define the work packages set out for successful completion of the project.

As a result of initial troubles with setting up the (simulated) distributed training environment and baseline, this project initially fell about one week behind schedule. This was necessary to ensure that the expected behaviour was observed in the case of no compression, as worker number was scaled. In the future, steps will be taken to further split this initial task into smaller steps, for example working with a smaller dataset to make iteration more quick and effective.

In line with the project plan, experiment running and data gathering took between $1-1.5$ months. Due to the nature of deep learning experimentation taking time, greater overlap with starting to write the current dissertation document was therefore found. This allowed for catching up on previous lost time.

[1] constructed using https://www.lucidchart.com/

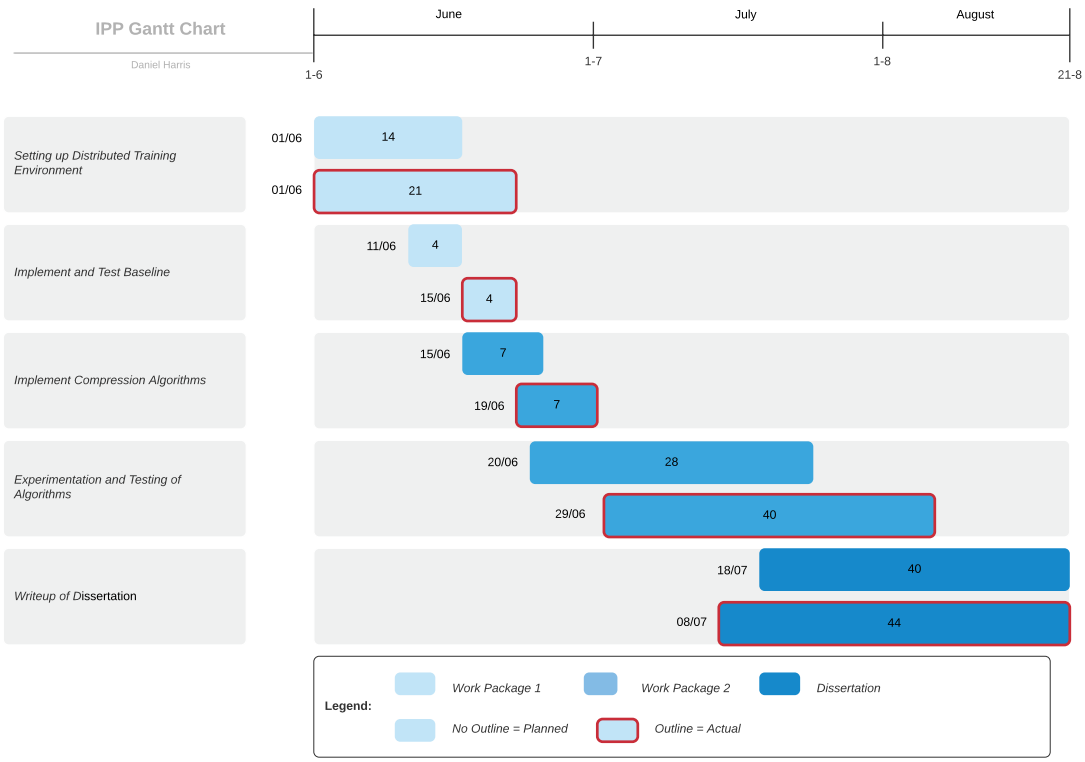

Figure C.1: Gantt chart reflecting time allocation both in project planning (no outline), and in time used during the project (red outline). The start date of each respective work package is indicated to its left in the chart.