Big Data Report
Ollie Keers – adbr309

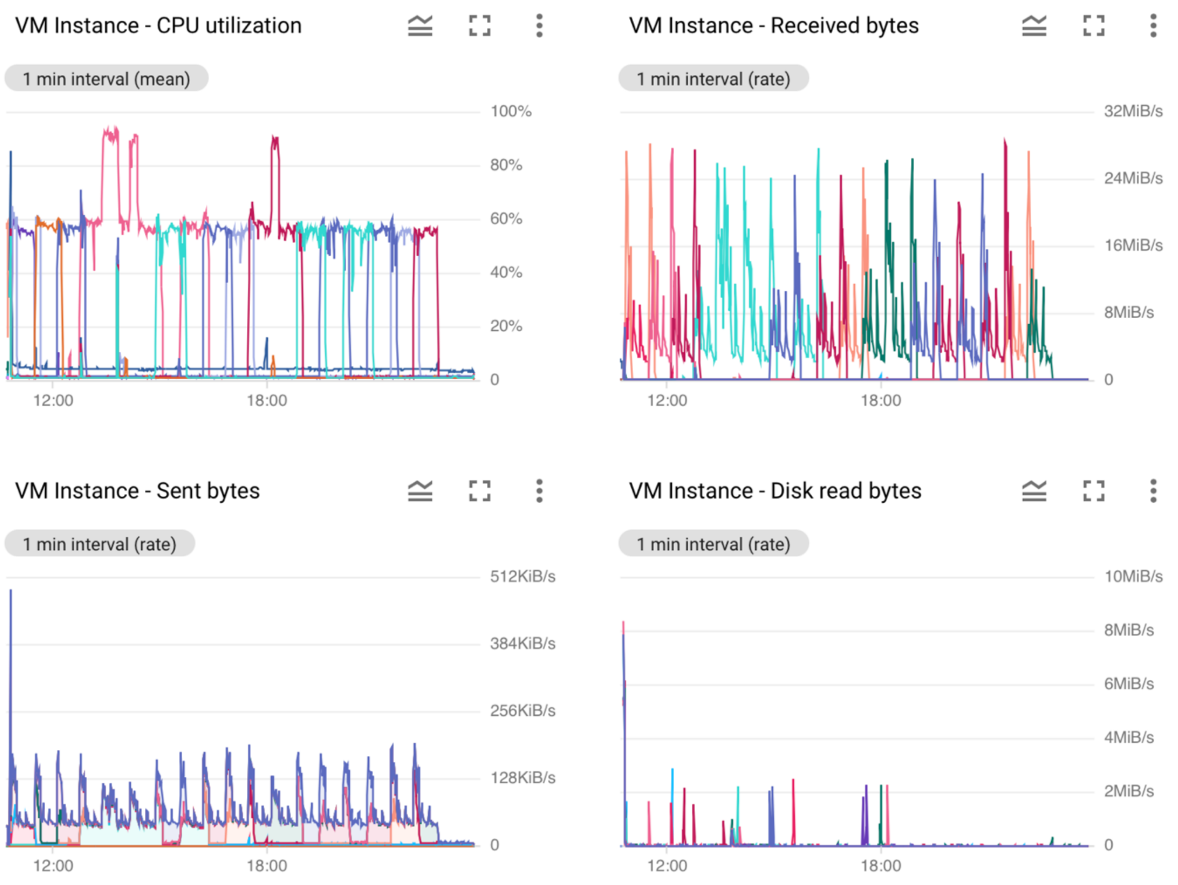
Workbook:
https://colab.research.google.com/drive/1xfmN2EJ9ElsblYgSiABXpvEMqGFMPEh6

https://drive.google.com/file/d/1xfmN2EJ9ElsblYgSiABXpvEMqGFMPEh6/view?usp=sharing

2c

RDDs are not stored by default to save on memory, but caching retains a copy of the data. Without this, Spark will recompute the RDD each time that an action is performed upon it, which can require the repetition of costly steps.

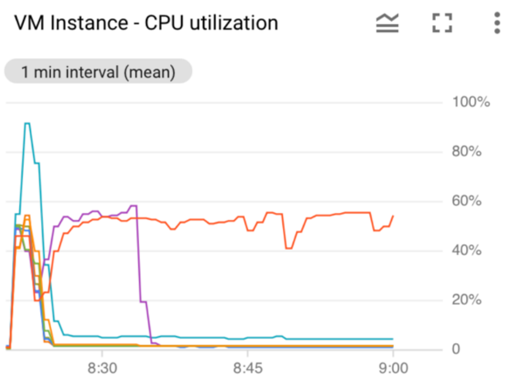
Unoptimized Code - 14 Hour Runtime

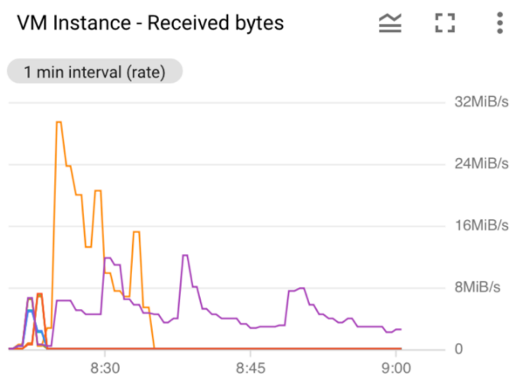


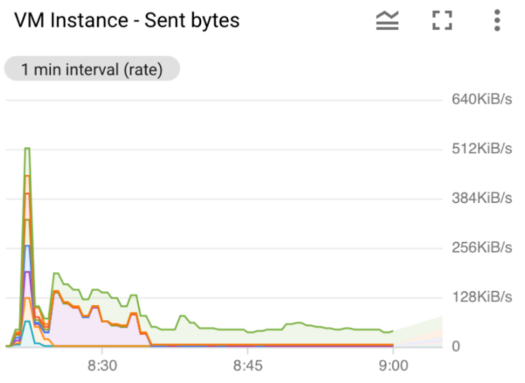
My code called seventeen collect operations after generating the RDD of throughputs, (one for all outputs, and for both datasets four for results by parameter, and four for averages by parameter). This gives rise to the 17 visible peaks in the CPU & network traffic diagrams, indicating the retransmission and reprocessing of data. Worker failure incidents are also visible at approx. 13:30 and 18:00.

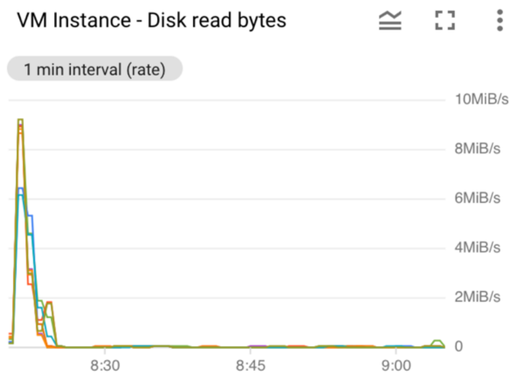
Caching the output rdd removes the need for repeatedly going through the most computationally expensive timing step and so significantly reduces wall clock time. The protracted sawtooth of the un-streamlined process becomes a single peak. Subsequent caching each of the parameter RDDs before computing averages, wouldn't be worth it because of the trivial computation.

With Caching - 45 Minute Runtime

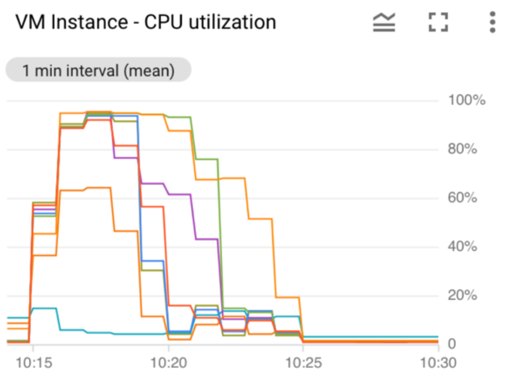


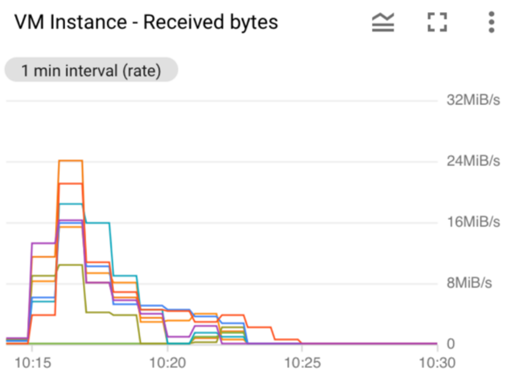


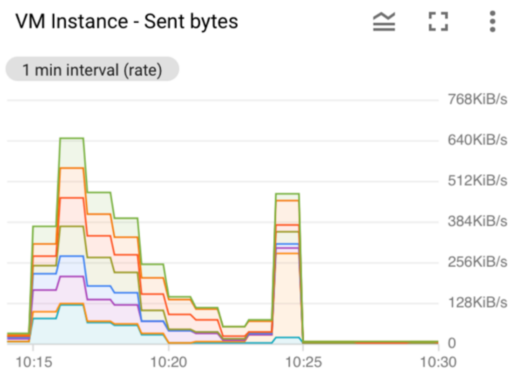


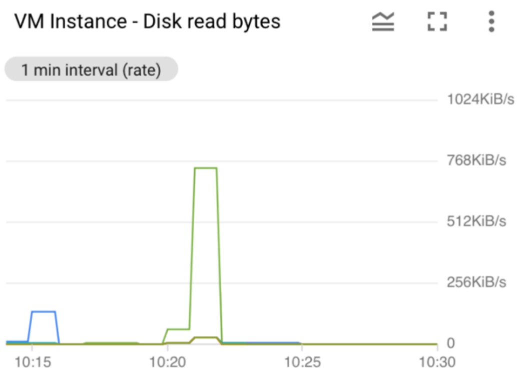


With Caching & Partitioning - 10 Minute Runtime





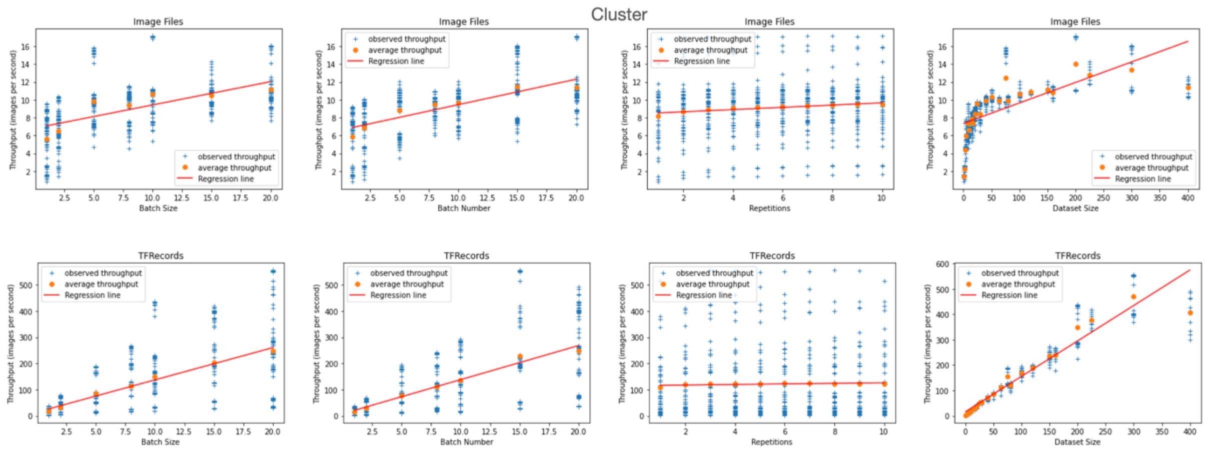




So far, only two workers were active for the processing, as visible by only the purple and red lines on the CPU chart remaining raised after the blue master line falls. There was no explicit call indicating how many partitions should be created when parallelizing. Experimentation with 7 (number of workers), 14 (number of worker CPUs), 16 (number of shards), and 49 (number of parameter combinations) partitions gave the shortest runtime with 49 partitions. With this, all workers are utilised, as seen in the multiple continually raised lines for CPU and network usage graphs. There is no longer a plateau where one worker is active at the end of the operation

	Caching & Parallelisation	Caching Only	Unoptimized
Runtime (mins)	10	45	840

2d



Working with TFRecords is an order of magnitude faster than working with image files, roughly equal to the difference in file sizes. For both file types, the throughput increases with increasing dataset size, but after reaching a maximum begins to decline. This suggests the increased efficiency for processing more data in one go has a limit, which could be when the data no longer fits in memory and needs to be read from disk. This is reinforced by the threshold being reached at a much greater dataset size for the smaller TFRecords than for images. The relatively small increase in throughput with repetition indicates that initial dataset creation is not a significant limiting factor, despite high latency for network traffic (~150ms for transatlantic packet return), suggesting the data is stored close to the processing. The ceiling of 30 images and 600 TFRecords per second is likely due to disk seeking and reading operations (~10ms & ~1ms, respectively) becoming the limiting factor. This could be helped by storing datasets in memory (0.25x the speed of SSD), but this isn't possible for very large datasets. For large scale ML, it is therefore important to compress data, and read or transfer it as few times as possible and parallelise operations so that this time can be distributed.

For a single machine, we would expect to see fairly similar behaviour. Once distributed, each parameter combination's operations are all taking place on a single virtual machine. As long as the single machine and each worker have the same configuration, processing a particular parameter combination would take place at the same rate. The advantage in overall time is because many workers run concurrently.

Processing is limited by the read/write speed of disks. The data that is produced needs to be stored immediately, and so cannot be produced faster than it can be written to disk, or it would be lost. It therefore makes sense to tie throughput to the capacity of disk resources.

Default object reading rate (5000/s) is 5 times greater than writing (1000/s), which we perform for each file. It takes several minutes for this to scale up, longer than our operations. The peak throughput seen (~600 files/s) seems well below the reading limit, but as there are 7 VMs making simultaneous requests this is actually at the limit, and likely to be the bottleneck observed.

In theory, data reading should scale linearly with size, O(n), and we do see this for smaller dataset sizes. In practice this doesn't apply to larger sizes because of limits elsewhere in the process such as disk reading. The point that linear modelling ceases to fit the data indicates the point that another factor is limiting throughput, and this can be used to improve efficiency.

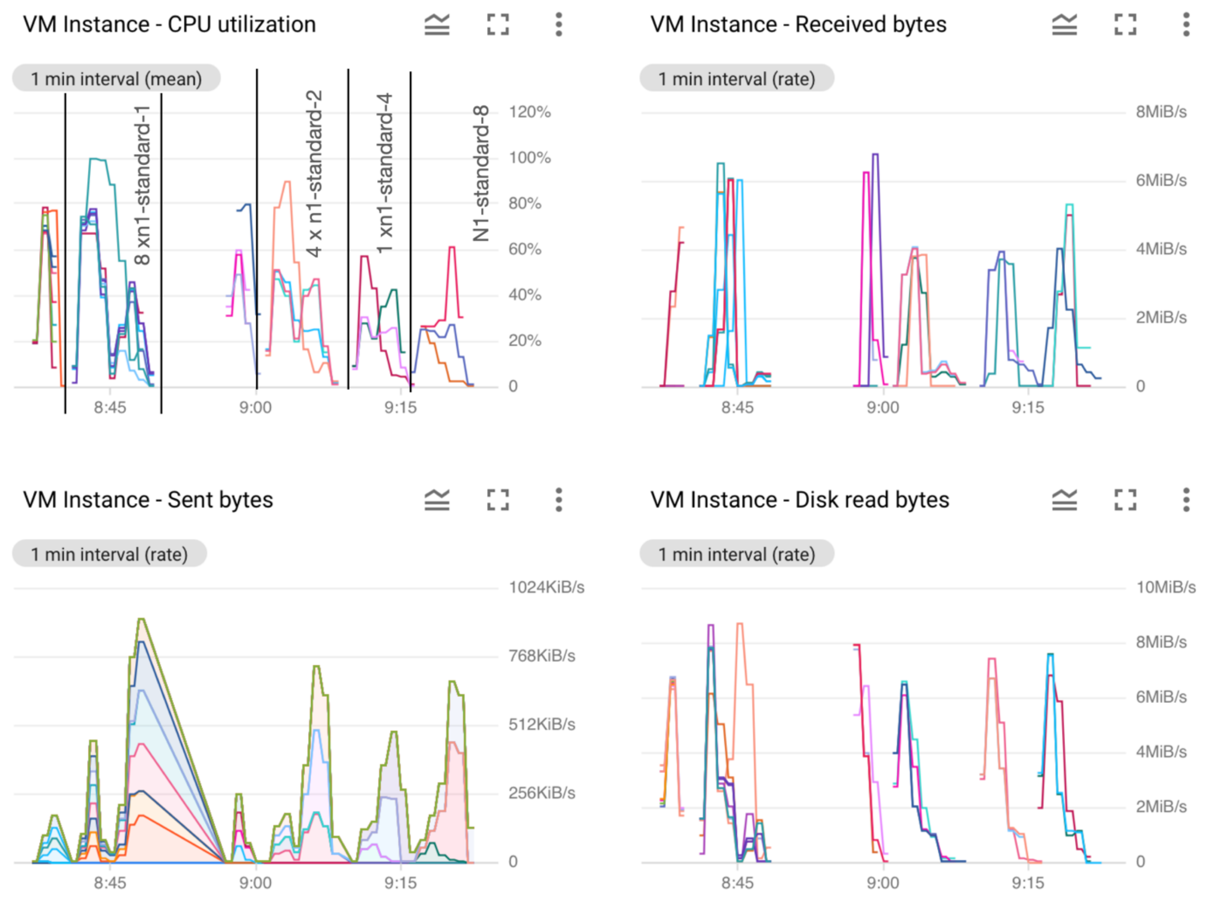
3c i

For most applications, Spark is used for parallelising processing before the output is collected for subsequent steps. Map transformations are lazy, and only executed when an action is called on the entire RDD. In this instance, mapping tfr_rdd_writer includes a write operation that causes each branch to do so in parallel, whereas usually this would happen after a reduce operation collected output. Here, the output remains parallelised.

3c ii

There is no significant difference in performance when going from 8 single-resourced machines to 4 double-resourced machines. However, there is an improvement when going to a single machine with quadruple the resources. The disk of the master remains at 100GB throughout, and as network capacity is tied to throughput this suggests that a bottleneck exists at the master distributing the files to the workers, negating the efficiency of performing the task in parallel. There is visibly less network traffic for the single machine, a stage with comparatively high latency. The slight increase in time for a single 8-CPU machine underlines that processing capacity is not the limiting factor.

	8 x n1-standard-1	**4 x n1-standard-2**	**1 x n1-standard-4**	**1 x n1-standard-8**
Time (s)	169	168	136	149

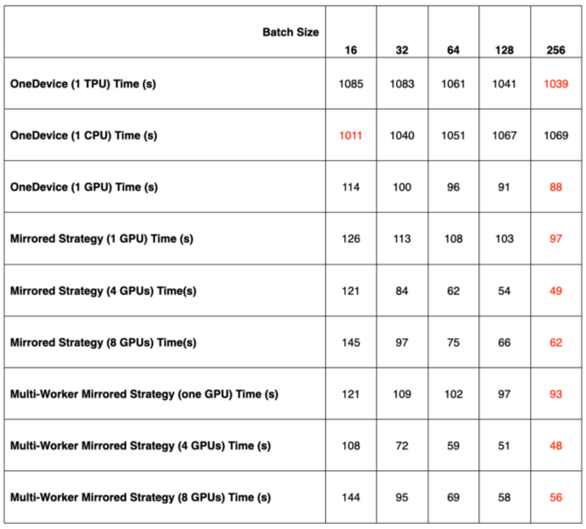


The incoming bandwidth for the bucket is likely to be a further limiting factor here, especially for the parallelised machines. The relatively slow decline in the 'sent-bytes' for the 8-machine cluster after processing shows that the transmission of outgoing data has taken far longer than the processing. This has reached a higher level than for smaller configurations, indicating that the outgoing bandwidth from the cluster is not the problem. It is therefore likely that there is a further bottleneck at the bucket, either through its network bandwidth or disk write speed.

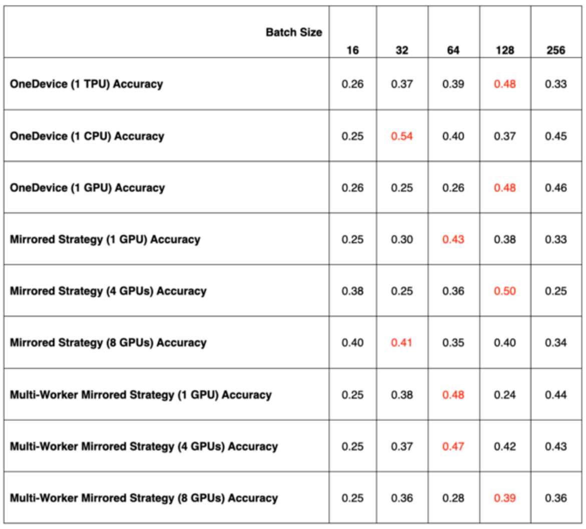
4c

Nine combinations of strategies and five different batch sizes were investigated. GPUs were, unsurprisingly, faster than TPUs or CPUs for one-device. For neural computing, GPUs are known to process data very quickly, and for the other machines the processing step was the bottleneck. For all other configurations, the most time-consuming stage was cluster startup (roughly ten minutes), which was longer than the combined total time to run all experiments on that machine.
Larger batch sizes led to faster convergence in all cases, with this having a greater effect as a percentage of the runtime for the faster configurations. There were no consistent findings for the most accurate batch size across machines.
For both multi-worker mirrored and mirrored strategy, the configuration using four GPUs was both faster and more accurate than using either one or eight. The speed gains from one GPU to four suggests that the speed increase through parallelisation of the processing has saved time, however the drop in performance for 8 GPUs

suggests that another factor is limiting performance. This could be the network bandwidth, as there would be more traffic at a given moment, or disk I/O speed.



Batch Size	16	32	64	128	256
OneDevice (1 TPU) Time (s)	1085	1083	1061	1041	1039
OneDevice (1 CPU) Time (s)	1011	1040	1051	1067	1069
OneDevice (1 GPU) Time (s)	114	100	96	91	88
Mirrored Strategy (1 GPU) Time (s)	126	113	108	103	97
Mirrored Strategy (4 GPUs) Time(s)	121	84	62	54	49
Mirrored Strategy (8 GPUs) Time(s)	145	97	75	66	62
Multi-Worker Mirrored Strategy (one GPU) Time (s)	121	109	102	97	93
Multi-Worker Mirrored Strategy (4 GPUs) Time (s)	108	72	59	51	48
Multi-Worker Mirrored Strategy (8 GPUs) Time (s)	144	95	69	58	56



Batch Size	16	32	64	128	256
OneDevice (1 TPU) Accuracy	0.26	0.37	0.39	0.48	0.33
OneDevice (1 CPU) Accuracy	0.25	0.54	0.40	0.37	0.45
OneDevice (1 GPU) Accuracy	0.26	0.25	0.26	0.48	0.46
Mirrored Strategy (1 GPU) Accuracy	0.25	0.30	0.43	0.38	0.33
Mirrored Strategy (4 GPUs) Accuracy	0.38	0.25	0.36	0.50	0.25
Mirrored Strategy (8 GPUs) Accuracy	0.40	0.41	0.35	0.40	0.34
Multi-Worker Mirrored Strategy (1 GPU) Accuracy	0.25	0.38	0.48	0.24	0.44
Multi-Worker Mirrored Strategy (4 GPUs) Accuracy	0.25	0.37	0.47	0.42	0.43
Multi-Worker Mirrored Strategy (8 GPUs) Accuracy	0.25	0.36	0.28	0.39	0.36

5a

The work of Alipourfard et. al (2017) introduces a procedure for finding the optimal configuration of a cloud for a particular task. In this work, we have experimented with different cluster and batch sizes during machine learning. As stated in the paper, finding the best setup is a very involved process, due to cluster startup time and the training time for each trial configuration, especially when jobs fail. While some choices may be logically deducible, such as faster processing on GPUs, others are not so evident. There are multiple bottlenecks in the process, as seen in tasks 2 and 3, so simply adding more processing units would not continuously speed up processing. We found that bandwidth and disk read & write speeds significantly limit operations and saw that processer capacity was not fully utilised as a result. Finding the sweet spot for a configuration is a process of trial and error, and Bayesian optimisation as outlined can aid with this. However, in this task we were working with a dataset of only a few thousand images. Unless the cluster going to be used for significant further work on similar data, the time invested in finding the optimal configuration is likely to be greater than the time saved in deploying it. Alipourfard et. al (2017) state that performance from an optimal cluster can be up to 12x faster. In task 2 caching and parallelisation were found to reduce processing times by 20x, suggesting that it in some cases it is wiser to invest time in optimising code rather than the cluster.

The work of Smith et al (2018) suggests increasing the batch size during model training to reduce training times. The work in Task 4 indicated that training times did reduce with larger batch sizes; while training was for the same number of epochs, there were fewer parameter updates because there were fewer batches of training examples. In this case the dataset was held in memory, which was possible because of the relatively small TFRecords, however in theory the performance benefits would be even greater if this was not the case. If the data needed to be retrieved live, this would involve slow network requests and disk operations for each batch, which were seen to limit operation speeds in Task 2 and 3. The total amount of data transferred would not change with a larger batch size, and it would not increase the training cycles needed. Using a decayed learning rate would increase the required training cycles, and thus require many additional network and disk operations to converge which would considerably increase cost. The benefits in this instance would be slight, especially given the number of training epochs, but be far more noticeable at scale.

5b
Working with batched data, it is worth using the largest batch size possible that fits in memory and cache them when appropriate. This minimizes disk I/O operations and network traffic, which are the slowest steps, and can also improve the efficiency of model training. It is beneficial to pre-process the data locally to reduce the file sizes to enable faster subsequent steps. Distributing this preprocessing is unlikely to decrease the time required due to bandwidth throttling. When parallelising, ensure that this is done so as to allow the master to distribute tasks effectively to all workers. As it is possible to define cluster architecture as well as partitioning, it is possible to distribute your data evenly across the workers, allowing for the most efficient processing. For repeated batch processing, Bayesian optimisation of the cluster architecture would prove effective over time as the nature of each batch is likely to remain consistent. Time spent in finding the optimal architecture during initial work would be rewarded down the line. Adapting the batch rate during training can be used to improve convergence wall clock times.

When working with online data, the rate of output must be at least as fast as the rate of input to avoid accumulation of data in the pipeline. The flow of data is limited by the slowest part of the system, and efforts should be focussed on optimising this bottleneck. This is likely to be related to disk operations or network capacity, rather than processing, so local preprocessing to minimize data size is vital. As the system is likely to be online for a prolonged period with relatively consistent throughput, investment in finding the optimal cluster configuration will yield significant benefits over time. Data is processed as soon as it arrives, hence the batch size is always one and cannot be increased, so decaying the learning rate is a more appropriate approach. Accumulating this data into minibatches, stream processing, is possible for some applications. Here increasing minibatch size may be a better strategy, as long as batch accumulation doesn't compromise the output rate required, however this change may mean the current cloud configuration is no longer optimal, requiring re-optimization.

Question	Wordcounts
2c	271
2d	457
3ci	67
3cii	199
4c	191
5a	454
5b	361
TOTAL	**2000**