
Deep learning for GPU poor

Author: Anton Zhitomirsky

Contents

1	Calculating statistics of a model	2
1.1	Foundation model parameter count	2
1.2	Floating Point System	2
1.3	Floating Point Operations (FLOPs)	3
2	Deep Learning At Scale	4
3	Methods for reducing computational requirements	6
3.1	Gradient Accumulation	6
3.2	Gradient Checkpointing	7
4	Finetuning problem setting	7
4.1	Low Rank Adaptation (LoRA)	7
4.2	QLoRA	9
4.2.1	K-Bit Quantisation	9
4.2.2	BrainFloat	9
4.3	Quantising W_0 from bfloat16 to 4 bit	10
4.3.1	4-bit NormalFloat	10
4.3.2	Double Quantisation	10
4.3.3	Dequantisation	10

1 Calculating statistics of a model

1.1 Foundation model parameter count

Commonly when you look online for models, you will see their parameter counts. *Transformers are typically described by the #parameters* which impacts the computational requirements to store and run these models.

1.2 Floating Point System

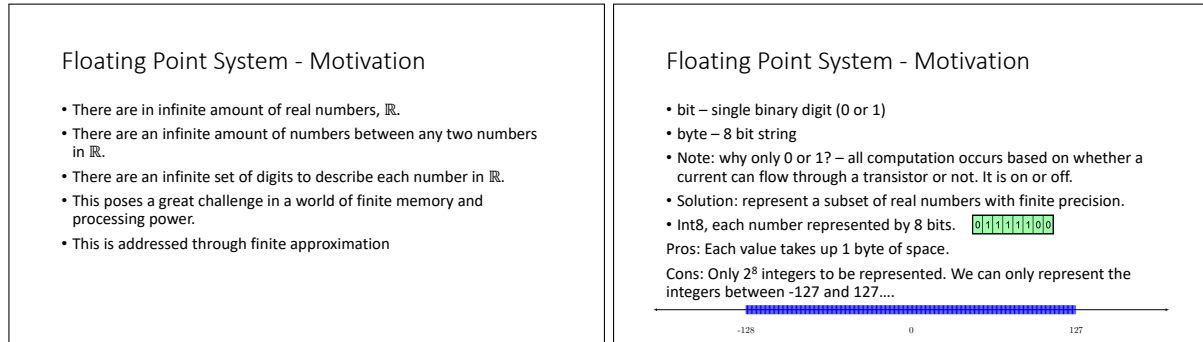
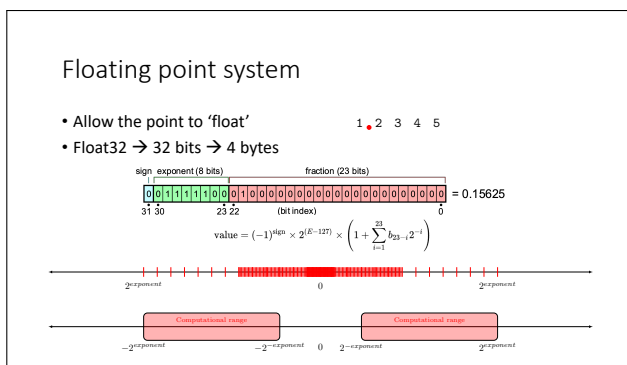


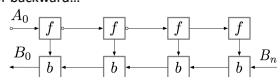
Figure 1: We store data in computers in bits with transistors. The more bits, the more precision.



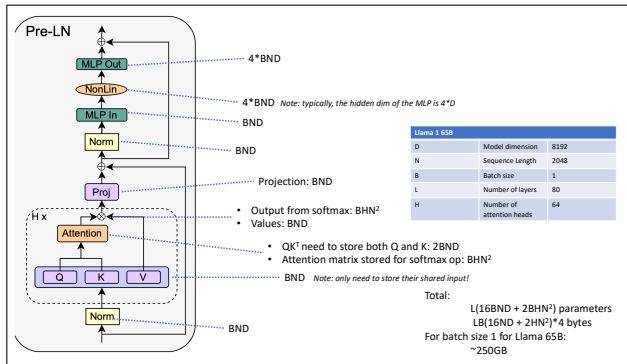
- **Floating point** is a way to represent numbers in a computer where the decimal point can float.
- A certain section is allocated to represent the precision (actual number) the exponent and the sign.
- still has the problem of underflow or overflow.

GPU Memory footprint for Llama 65B @32float

- Inference time:
32 (bits) * 65 * 10⁹ (params) = 2.08 * 10¹² (bits) / 8 = 260GB
- Training time
 - Model parameters = 260GB (same at inference time)
 - Gradients! 4 * 65 * 10⁹ = 260GB
 - Optimiser: 2 * 4 * 65 * 10⁹ (Adam coefficients momentum + variance) = 520GB
 - Activations for backward...



- At inference time, just to store the data it is already 260GB
- At training time, we also require gradients and optimiser values (momentum, etc) which can be 10x the size of the model and the activations.



So if we pass through the network, we need to store the inputs to every single layer that there's a change in the computational graph. Here:

- The first is the batch times the number of tokens or the number of patches times by that dimensionality.
- For the query key value, they all get the same value.
- The attention matrix requires query with key-values
- We also need to store the n^2 complexity of the attention matrix.
- Also we store the result of the soft max which is the batch times the number of heads times the sequence dimension.
- With multi-head attention we usually have another layer that learns how to combine these values together.
- All in all, we have 250GB for a batch size of 1.

GPU Memory footprint for Llama 65B @32float

- Inference time:
 $32 \text{ (bits)} \times 65 \times 10^9 \text{ (params)} = 2.08 \times 10^{12} \text{ (bits)} / 8 = 260 \text{ GB}$
- Training time
 - Model parameters = 260GB (same at inference time)
 - Gradients! $4 \times 65 \times 10^9 = 260 \text{ GB}$
 - Optimiser: $2 \times 4 \times 65 \times 10^9$ (Adam coefficients momentum + variance) = 520GB
 - Activations for backward assuming batch size of 1 = 250GB
- Activations for backward pass are strongly dependent on sequence length and batch size
- Model params + Gradients + optimizer + activations = 1290GB
- 25 RTX 6000s! – just for a batch size of 1....

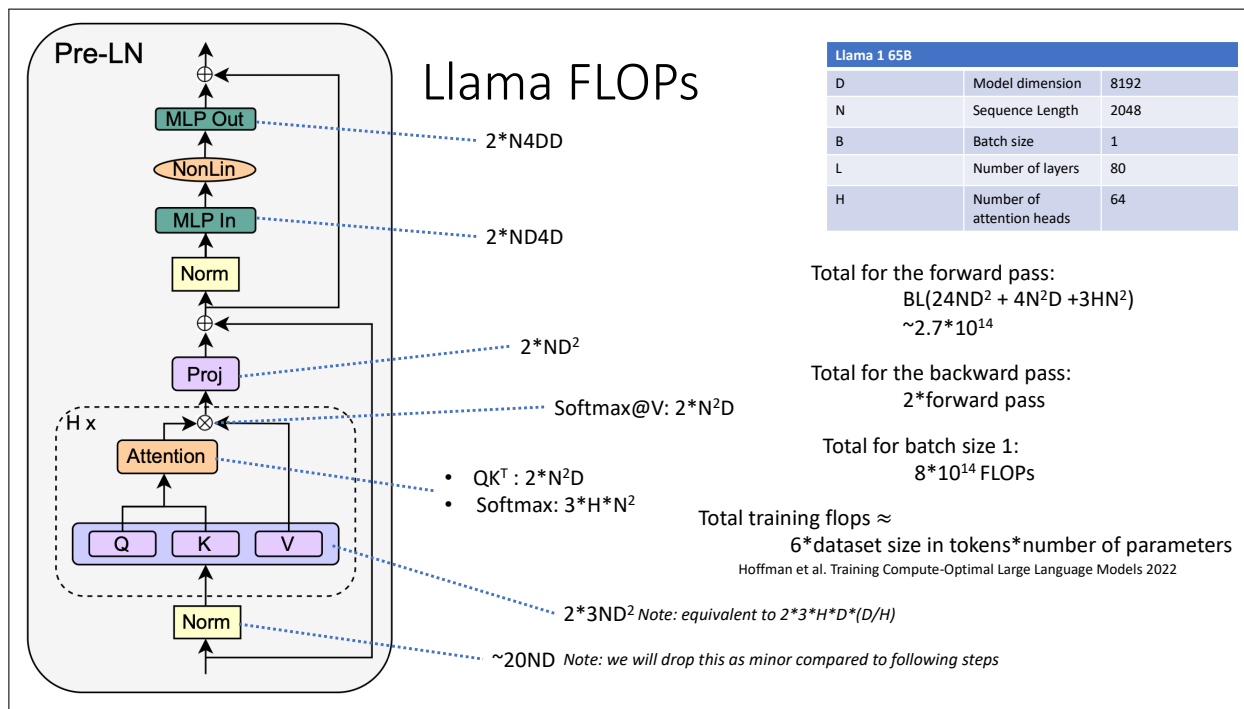
1.3 Floating Point Operations (FLOPs)

Floating point operations (FLOPs)

- Basic unit of computation
- e.g. addition, subtraction, multiplication, divide of 2 floating point numbers
- Examples
- Given $\mathbf{w}, \mathbf{x} \in \mathbb{R}^n$
- $\mathbf{w} + \mathbf{x}$:
 n FLOPs
- $\mathbf{w}^T \mathbf{x}$:
 $2n$ FLOPs (multiplication + $(n-1)$ addition)
- Given $\mathbf{W} \in \mathbb{R}^{m \times p}$, $\mathbf{X} \in \mathbb{R}^{p \times n}$
- $\mathbf{W}\mathbf{X}$:
 $2nmp$

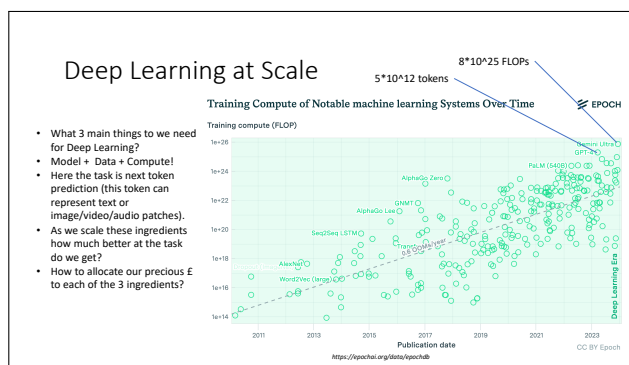
Computational requirements:

- given two vectors, if we add this will require n operations.
- For the dot products, there is $2n$ flops.
- For the matrix multiplication, there is $2nmp$ multiplications

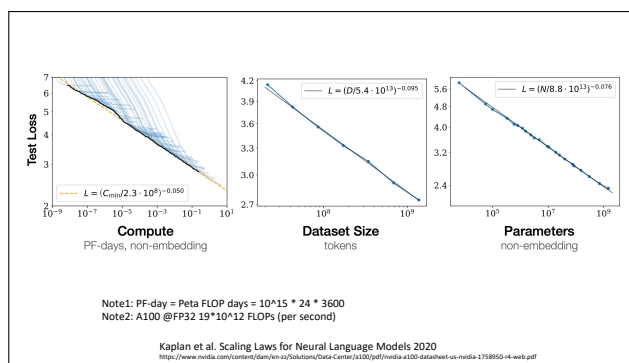


1. the layer norm is roughly 20 operations for every feature and token
2. for the query key values we have it 3 times, and 2 because it's a dot product.
3. The softmax: for every single query we dot product it for every single value.
4. projectino is same as query key values (but only 1)

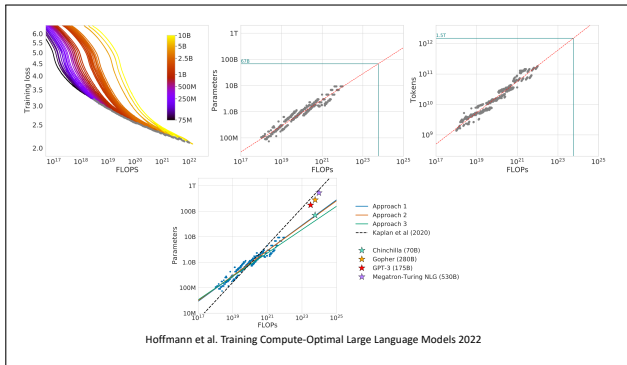
2 Deep Learning At Scale



- For deeplearning we need model data and compute.
- At scale, you can't train the model many times for hyper-parameter tuning. A single model may take many dollars to train just once.



“When they have limitless compute and limitless datasets, they saw this really nice power law trend where it sticks very closely to the trend. This says that generally you can predict quite well, if you have limited compute and limited dataset size what loss you would get at the end.” The values here were proven to be wrong because the smaller models were not fully optimized; they used the same learning rate schedule as they did for the larger models. Therefore, the line was too steep.

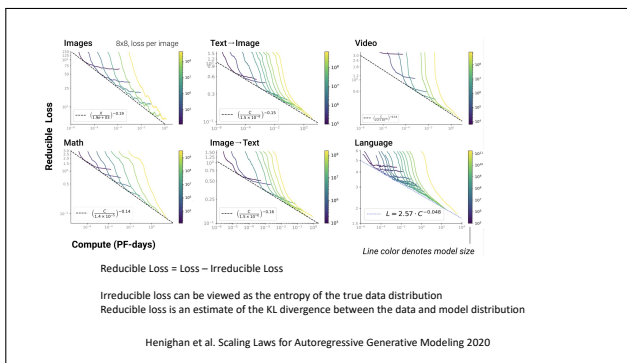


“openAI published findings about colour representing the size of the model, and they fit for given flops and compute budgets which model gets the lowest loss.” Here, we see that chinchilla is favoured since it prefers the number of data points over the model size.

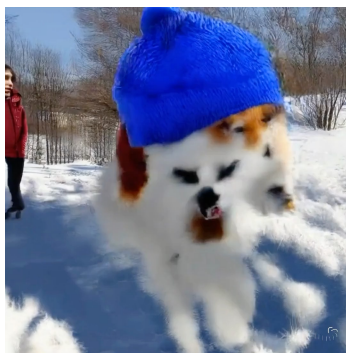
Here we see that ‘deep learning has been solved’. “This is showing that this happens in all modalities; these losses can just keep going until we get to super human performance.”

Definition 2.1 (Reducible Loss). It is the actual loss - the irreducible loss, where irreducible loss is viewed as the true entropy of the data i.e. if I had a perfect fit of the data, what would be my loss even if I had a perfect fit?

It can be viewed as the KL divergence between the manifold of the dataspace and the manifold of the learnt model.



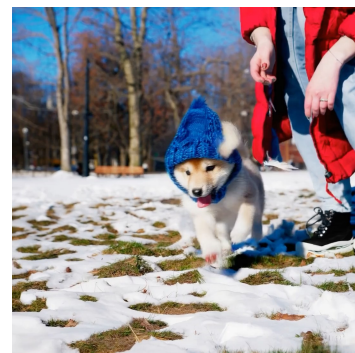
SORA



Base compute



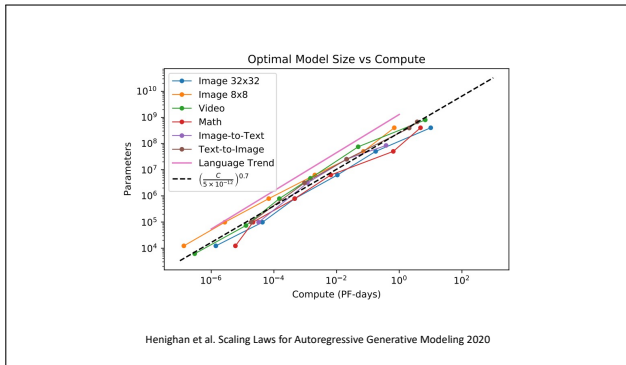
4*compute



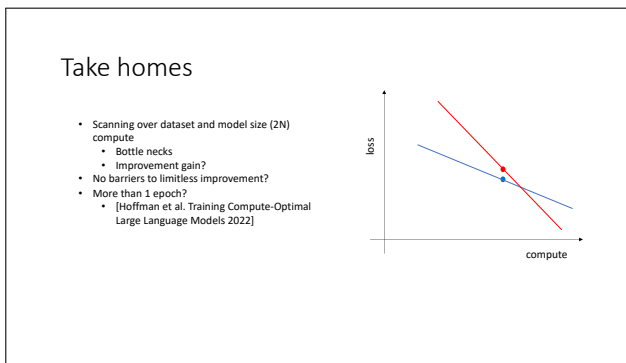
16*compute

<https://openai.com/research/video-generation-models-as-world-simulators>

Figure 2: SORA AI just used more compute!



“When we plot all modalities the number of compute days vs the number of parameters for the optimal setting. There are correlation between what it is like to compute potimal size model for videa and for language model”



- if blue is state of the art and red is my implementation,
- then we can do a scaling analysis and conclude that since the curve is steeper then it may perform better at high levels of compute.

3 Methods for reducing computational requirements

3.1 Gradient Accumulation

How to get the most out of your resources Part 1: Gradient Accumulation

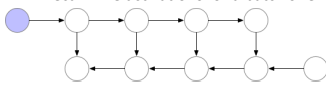
- Limitations of a very small batch size?
 - Significant noise in parameter updates
 - Many parameter updates
 - Cannot replicate lab's work
- Iteratively compute forward + backward passes and sum up gradients.
- Only run the optimiser step once (your minibatch * number gradient accumulation steps = target batch)

- As we increase the batch size our computation requirements increase
- If we use 1 batch, this is just Stochastic Gradient Descent so you're approximating the true gradient with one sample; your loss will be jumping around without convergence
- accumulate the gradients without updating the optimizer, then when the mini-batch size times the number of cumulative gradient accumulation steps equals the target batch then we do the step in the loss surface.*
- Useful if you want a larger batch size

3.2 Gradient Checkpointing

How to get the most out of your resources Part 1: Gradient Checkpointing

- Recall: The activations for a batch size 1 for Llama 65B is 172GB



Memory: O(1)
Compute: O(L²)
#forward per layer: 1-L

- We could forget each activation during the forward pass and then recompute them during the backward pass
 - This adds significant more compute
- Compromise! Strategically select activations throughout computational graph, so only a fraction of the graphs need to be recomputed.

Chen et al. Training Deep Nets with Sublinear Memory Cost 2016
<https://medium.com/tensorflow/finetuning-larger-networks-into-memory-583e3c758899>

- idea is to initially throw out the activation in the forward pass after we've done it, but in the backpropagation step we redo the entire forward computation step.
- This is computationally expensive, but the memory requirements stay low; it doesn't scale with the number of layers.
- The forward pass is much faster than the backwards pass
- Therefore, we select checkpoints from which we continue the forward pass.

4 Finetuning problem setting

Finetuning problem setting Examinable

- After pretraining we may want to adapt our model to a specific downstream task.
- The pretrained weights Φ , are updated $\Phi + \Delta\Phi$ according to

$$\max_{\Phi} \sum_{(x,y) \sim \mathcal{Z}} \sum_{t=1}^Y \log(P_{\Phi}(y_t | x, y_{<t}))$$

Context-target tokens Autoregressive foundation model

- Here, every parameter in Φ is updated therefore need to store gradients + optimisers of every weight.
- For every downstream task a new model of size Φ is required + we need to relearn $\Delta\Phi$ parameters.

(e) “For every downstream task” we need to fine tune (lots of flops of updating) and store each separate model for each downstream task. This is computationally burdensome

- We're talking about autoregressive models; we have a pre-trained auto-regression model, and for our fine tuning dataset, we select some context target pairs (the tokens before the token you're trying to predict).
- We're trying to min/max this over each of the pre-trained parameters.
- This requires us to store the gradients and optimisations for every single one of these parameters.
- We also require more leaning signal to provide enough signal to update every single one of these parameters (scalability issue as shown above with scaling laws).


4.1 Low Rank Adaptation (LoRA)

Low Rank Adaption (LoRA) Examinable

- Motivation
 - Li et al. demonstrated that the model-loss overparameterised space has a low intrinsic dimension/rank.
 - What if $\Delta\Phi$ also has low intrinsic rank we can approximate $\Delta\Phi$ with Θ where $|\Theta| \ll |\Phi|$

$$\max_{\Theta} \sum_{(x,y) \sim \mathcal{Z}} \sum_{t=1}^Y \log(P_{\Phi+\Theta}(y_t | x, y_{<t}))$$

Only optimise Θ



Li et al. Measuring the intrinsic dimension of objective landscapes 2018
Goodfellow et al. Qualitatively Characterizing Neural Network Optimization Problems 2015
Aghajanyan et al. Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning 2020

(f) Diagram: “The intrinsic low rank of the over-parameterised space decreases as the number of parameters of our pre-trained model increases; the larger the model, the simpler the loss parameter space is”.

- During fine-tuning, researchers constrained the number of parameters that are available for fine-tuning and saw that the training performance actually didn't drop.
- If we increase the number of parameters, the likelihood of getting stuck in a local minima 'saddle' increases.
- The authors: What happens if $\Delta\Phi$'s rank is intrinsically low? We can instead update Θ that is significantly less than Φ .
- TLDR: update only a subset of parameters.

Low Rank Adaption (LoRA)

- For a weight matrix in a pretrained model: $W_0 \in \mathbb{R}^{D \times D}$
- Following convention that at fine tuning we are learning: $\Delta W \in \mathbb{R}^{D \times D}$
- For a pretrained set up of $y = W_0 x$
- $y = XW_0 + X\Delta W \approx XW_0 + XL_1L_2$
- Where, $L_1 \in \mathbb{R}^{D \times r}$ and $L_2 \in \mathbb{R}^{r \times D}$ and $r \ll D$
(note: matrix multiplication is distributive!)
(note: $L_1L_2 \in \mathbb{R}^{D \times D}$)
- This can be done for every weight layer in the network

Hu et al. LoRA: Low-Rank Adaption of Large Language Models 2021

Low Rank Adaption (LoRA)

- Empirically show that $r = 1$ or $r = 2$ yields good results for GPT-3 which has a full rank of 12,288!
- r will depend on the degree of shift between the pretraining dataset and the finetuning dataset.
- Here, the number of trainable parameters, $|\Theta|$ can be 0.01% of the pretrained parameters, $|\Phi|$.

Weight Type	$r = 1$	$r = 2$	$r = 4$	$r = 8$	$r = 64$
WikiSQL ($\pm 0.5\%$)					
W_q	68.8	69.6	70.5	70.4	70.0
W_k	73.4	73.3	73.7	73.8	73.5
W_v	74.1	73.7	74.0	74.0	73.9
MultiNLI ($\pm 0.1\%$)					
W_q	90.7	90.9	91.1	90.7	90.7
W_k	91.3	91.4	91.3	91.6	91.4
W_v	91.2	91.7	91.7	91.5	91.4

Hu et al. LoRA: Low-Rank Adaption of Large Language Models 2021

Low Rank Adaption (LoRA)

- At inference time we combine: $W_0 + L_1L_2$
 - No additional inference time (no extra steps to compute both XW_0 and XL_1L_2)
 - We can always subtract L_1L_2 when we are done and combine with a different set of parameters L_1L_2
- How much GPU memory does this save?
 - No longer need to train 99.99% of parameters (number from Llama 65B in part1):
 - Gradients: $4 \times 65 \times 10^9 = 260\text{GB}$
 - Optimizer: $2 \times 4 \times 65 \times 10^9 = 520\text{GB}$
- Still need to store:
 - Model parameters = 260GB
 - Activations for backward = 172GB
- Assuming batch size of 1 we need 3 times less GPU resources.

Hu et al. LoRA: Low-Rank Adaption of Large Language Models 2021

- If before, we learn that the pre-trained projection and also the finetuned projection Δ can we decompose ΔW into a low rank version.
- in blue: the original pre-trained model where we take the input and multiply it by the weight matrix. We freeze these, so we don't learn anything about them.
- In green, in parallel we have an approximation of the weight matrix that is low rank. We only learn this.
- intuitively it works because we project $L_1 \times L_2$ into a $D \times D$ matrix.
- they found that the low rank approximation of the weight matrix was able to capture the majority of the information in the original weight matrix even at $r = 1$.
- This is susceptible to initialisation bias. They set L_2 to be 0s; they don't add any new information in the first couple of iterations – its only the pre-trained model that is doing the work that is itself frozen. Then the model slowly introduces the new information.
- At inference time we have a lot of flops from multiplying the matrix together. What they do is they fuse the kernels together.
- We can then swap in the very small memory footprints in easily.
- We can then apply some methods from Gradient Accumulation/Checkpointing.

Therefore the big drawback with LORA is that we still need to store the W_0 matrix.

4.2 QLoRA

4.2.1 K-Bit Quantisation

QLoRA Background: k-bit quantisation

Examinable

- Teaser: QLoRA enables finetuning of a 65B parameter transformer model on a single 48GB GPU + in 12 hours leads to SOTA opensource model performance.

- K-bit quantisation. Example float32 \rightarrow Int8

$$X^{\text{int8}} = \text{round}\left(\frac{127}{\text{absmax}(X^{\text{FP32}})} X^{\text{FP32}}\right) = \text{round}(c^{\text{FP32}} \cdot X^{\text{FP32}})$$

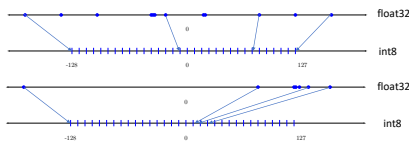
$$\text{dequant}(c^{\text{FP32}}, X^{\text{int8}}) = \frac{X^{\text{int8}}}{c^{\text{FP32}}} = X^{\text{FP32}}$$

- To ensure full range of input is captured in Int8, input is rescaled to target range here [-127, 127]
- Where c^x is the quantisation constant

- K-bit quantisation: this is going from a data type of higher precision or a higher bit rate to a lower bit rate
- we accomplish this by scaling by the absolute max of the input data

QLoRA: Background: k-bit quantisation

Examinable



Outliers result suboptimal allocation of quantized bins
 Chunking! Slice $W \in \mathbb{R}^{D \times D}$ into n chunks of size C where $n = \frac{D \times D}{C}$
 Quantise each block separately with unique c^x

- We have an example with datapoints in float in blue. We then quantise it down to int8
- We scale the absolute max to the lowest possible value and scale everything else equally by using the quantisation constant.
- This is a lossy process. We can't go back to the original data, but we can go back to the original data type.
- This has the setback that if we have outliers, we scale things to a bin, and the representation becomes very sparse.
- To solve this issue you perform chunks, into small pieces so that the chances of there being an outlier is very small.
- Each chunk therefore has a unique quantisation constant.

4.2.2 BrainFloat

QLoRA: BrainFloat

Examinable



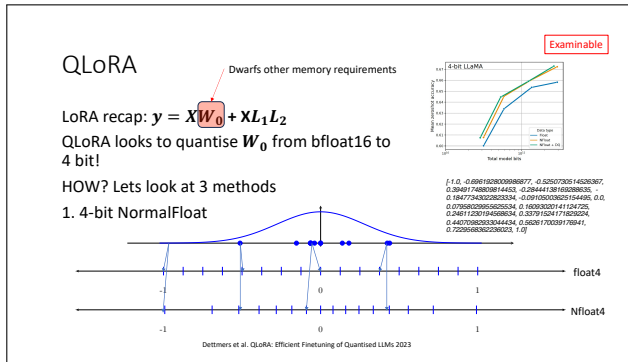
- Deep learning models are more sensitive to underflow + overflow than to precision
- Hardware requirements scale quadratically with fraction but not exponent

[developed at google] <https://cloud.google.com/tpu/docs/bfloat16>

- This type favours the exponent significantly.
- We found that the models were more robust to the actual precision of the bodies than they were to a thing called under or overflow. (where gradients are so small that they become 0)
- “models performed significantly better when we worried more about underflow than overflow”.

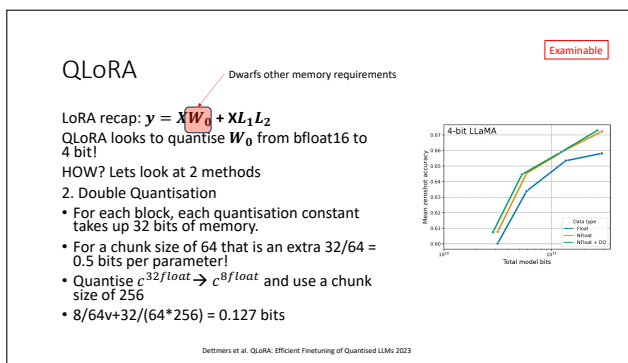
4.3 Quantising W_0 from bfloat16 to 4 bit

4.3.1 4-bit NormalFloat



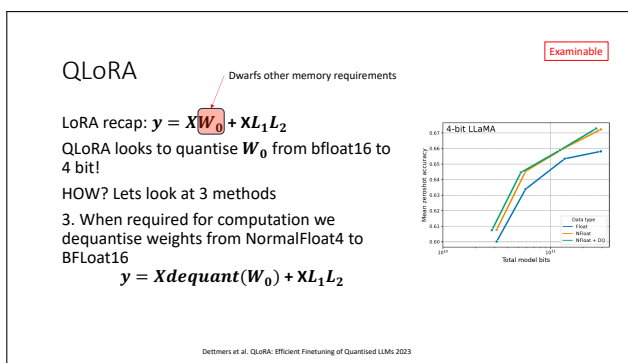
- A big drawback with LORA is that we still need to store the W_0 matrix.
- We can use BrainFloat to reduce the memory footprint of the model to 4 bits!
- Here, we rely on the idea that most data in deep learning is a normal distribution.
- Majority of values map to the same central point normally
- Therefore, they introduce a new type where values are normally distributed in the new type also. The idea is that they are wider at the edges
- The numbers are all the numbers that an Nfloat4 can represent, and in the graph we see Nfloat performs better.

4.3.2 Double Quantisation

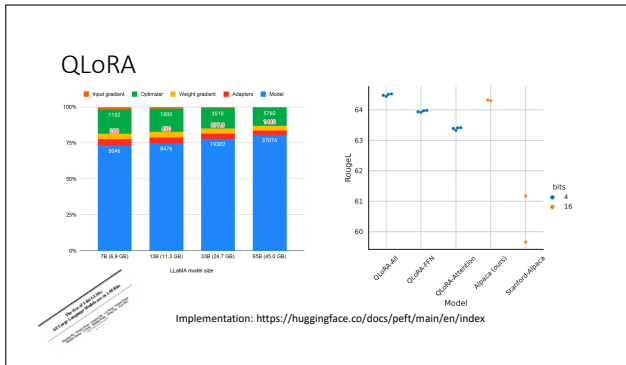


- In K-bit quantisation, we store quantisation constants that is stored in 32 bit precision.
- If we quantise 65 billion parameters, then we have a lot of quantisation constants.
- Therefore, we quantise the quantisation constant.
- Quantisation and de-quantisation is a cheap operation and fits nicely into consumer-grade hardware.

4.3.3 Dequantisation



- The normal float is a static memory type.
- When we train the model, the vast majority of the pre-trained model is stored in 4 bit.
- When we matrix multiply we de-quantise it into 16 bit brainfloat space and perform the matrix multiplication.
- (L_1 and L_2 are still in 16 bit representation)



- PEFT good for project.