Llama2.pi: Running LLMs on the Bleeding Edge

Stanford CS224N Custom Project

Matthew Ding
Department of Computer Science
Stanford University
mattding@stanford.edu

Abstract

We attempt to run Llama2 inference on a bare-metal Raspberry Pi Zero with 512 MB of RAM. It doesn't work, but we get close and, in the process, benchmark memory usage, inference speed, and model performance for multiple software optimizations.[1]

1 Key Information to include

- Mentor: Tony Wang
- Sharing project: with CS140E

2 Introduction

The rapid advancement of large language models (LLMs) has revolutionized natural language processing; however, their immense size and computational requirements often restrict them to run on specialized hardware or cloud infrastructure. There is growing interest in running LLMs on cheap, commodity hardware, such as the Raspberry Pi line of microcontrollers. Achieving efficient inference on inexpensive processors like the Pi would democratize access to these powerful language models and allow individual researchers and developers to leverage their capabilities. In addition, it would enable a new suite of applications for these so-called "edge devices," with new enhancements to privacy and data protection.

This project takes a crack at this lofty goal. We attempt to perform inference with Meta's 7-billion parameter Llama2 model on a Raspberry Pi Zero, which runs on a 1GHz single-core CPU and contains just 512 MB of RAM. Quick internet searches show that there have been successful attempts at running Llama2-7B on Raspberry Pi's in the past, but these projects always use the more-powerful B Model with 4 GB or 8 GB of RAM. As far as we can tell, there have been no previous (public) attempts to run Llama2 on the Pi Zero. In our case, minimizing memory usage is critical, and our primary focus is on optimizing the memory requirements for the forward pass to fit within the strict confines of the hardware.

3 Related Work

Due to computational improvements in recent years, the primary bottleneck for LLM inference has shifted from compute — the number of arithmetic operations per second — to memory bandwidth — the speed at which parameters can be loaded and stored. As a result, there is much ongoing research aimed at reducing LLMs' memory requirements to get around this "memory barrier" (Kim et al., 2024). Current software methods for reducing memory can be broadly categorized into three areas: pruning, quantization, and matrix decomposition. We now discuss the existing literature on all three methods, although we only use the first two in our experiments.

[1] `https://github.com/mattyding/llama.pi`

3.1 Pruning

Weight pruning techniques induce sparsity in model weights by removing less-important connections, allowing for sparse matrix operations and storage. Classical pruning techniques remove small weights based on thresholding (Han et al., 2016; Narang et al., 2017), and these simple approaches have been shown to extend to LLMs (Sun et al., 2023). However, while pruning can achieve high compression ratios, it introduces irregularity (through the sparse matrix representations) that can limit speedups on modern hardware. Thus, pruning is generally less preferred to other approaches.

3.2 Quantization

Quantization refers to the process of representing model weights and activations with lower-precision numerical formats. Typically, 32-bit floating point numbers (FP32) are quantized to 8-bit or 4-bit integers (INT8, INT4), on which arithmetic operations can be performed more efficiently. This process can be applied either post-training (Frantar et al., 2023; Xiao et al., 2023) or during the training process itself (Jacob et al., 2017). While quantization generally trades off model size and efficiency for accuracy, recent work has shown that extremely low-precision quantization down to 1 or 2 bits is possible with careful techniques (Ma et al., 2024; Chee et al., 2024).

3.3 Matrix Decomposition

Matrix decomposition methods factorize large weight matrices into smaller components, reducing the number of multiplication operations required. These methods include classical algorithms such as Singular Value Decomposition (SVD) and tensor decomposition. Recently, the low-rank adaptation (LoRA) approach has emerged as an efficient method for fine-tuning LLMs while simultaneously reducing memory requirements (Hu et al., 2021; Dettmers et al., 2023). LoRA factorizes the weight updates during fine-tuning into two smaller matrices, significantly reducing the number of parameters.

While all of these methods work well individually, they can be utilized in tandem for better results. For instance, (Han et al., 2016) combines quantization, pruning, and Huffman coding to achieve a 49% compression ratio. Similarly, in this project, we will combine these methods with novel techniques to substantially reduce the model size.

4 Approach

We use the Llama2-7B model for our experiments, obtaining the pre-trained weights from Meta's website (Touvron et al., 2023). Our code is a modification of Karpathy's `llama2.c` repository, which reimplements the forward pass of the model in C and with INT8 quantization.[2] Like many other popular LLMs, the Llama2 model employs a decoder-only (unidirectional) Transformer architecture, which enables faster inference than the encoder-decoder (bidirectional) variant. A diagram of the forward pass is shown in Figure 1.

We port Karpathy's implementation to run bare-metal on a Raspberry Pi. This means that we do not have the benefits of a full operating system, such as virtual memory and thread support. We borrow the setup from Stanford's CS140E class to compile and bootload our program.[3] We prune and quantize our model weights to INT8, and additionally implement a novel optimization approach that iteratively loads layer weights into memory.

[2] `https://github.com/karpathy/llama2.c`
[3] `https://github.com/dddrrreee/cs140e-24win`

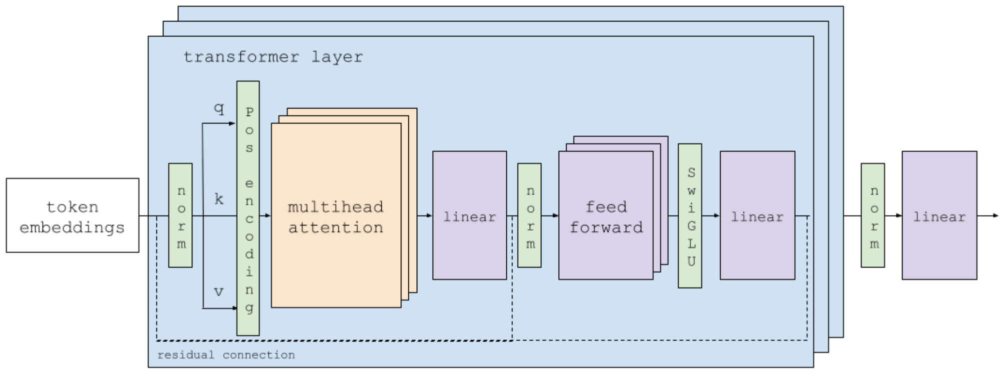

Figure 1: Llama2 forward pass.

4.1 Pruning

We implement a simple serialization format for sparse matrices that allows them to be represented with memory proportional to the number of nonzero elements. The format is as follows:

- Each file begins with a fixed-size header, containing a field `n_bytes` that stores the total number of entries (both zero and nonzero) in the matrix.
- The next $\lceil \log_2(\texttt{n\_bytes}) \rceil$ bytes of the file store a bitvector, where each bit corresponds to an entry of the matrix. A bit is set to 1 if the corresponding matrix entry is nonzero and 0 otherwise.
- Following the bitvector is an array of all nonzero matrix entries, stored in row-major order.

To look up the value of an entry in the matrix, we first check the corresponding bit in the bitvector. If the bit is 0, the entry is zero. Otherwise, we iterate through the bitvector, counting the number of nonzero entries preceding the desired entry, and then look up the corresponding value in the array of nonzero entries.

Note that, like almost all of our methods, we incur additional computational costs (namely, constant to linear-time lookup of nonzero matrix entries), but in return, are able to instantiate large sparse matrices in memory, provided that the number of nonzero elements is relatively small.

We evaluate our approach on a single feedforward layer. The unmodified weights in FP32 precision require 225.4 MB of memory, while 20% pruned weights require 189.4 MB, and 50% pruned weights require only 135.3 MB. We use L1-pruning, which removes the proportion of parameters with the smallest L1-norm. Samples of model output at varying pruning levels can be found in Appendix A.

4.2 INT8 Quantization

We quantize our weights to INT8 using per-channel absmax quantization, as described in Dettmers et al. (2022). For a FP32 weight matrix $\mathbf{X}_{F32} \in \mathbb{R}^{s \times h}$, we group the weights into batches ("channels") of size g, where g divides sh. Let $G = \frac{sh}{g}$ denote the number of groups and define the kth group to be $X_{F32_k} \in \mathbb{R}^{g \times G}$ for $k = 1, \ldots, g$.

For each group k, we quantize its weights into the 8-bit range $[-127, 127]$ by dividing by the scaling factor for that group, defined as $s_k = \frac{\max(|\mathbf{X}_{F32_k}|)}{127}$, where $\max(|\mathbf{X}_{F32_k}|)$ denotes the absolute maximum of the group. The INT8 quantization matrix for that group is:

$$\mathbf{X}_{I8_k} = \left\lfloor \frac{1}{s_k} \cdot \mathbf{X}_{F32_k} \right\rceil = \left\lfloor \frac{127 \cdot \mathbf{X}_{F32_k}}{\max(|\mathbf{X}_{F32_k}|)} \right\rceil$$

where $\lfloor \rceil$ indicates rounding to the nearest integer. We obtain the full quantized weight matrix $X_{I8} \in \mathbb{Z}^{s \times h}$ by combining the quantized groups back into the original tensor shape.

We store the INT8 representation of the weights, along with the scaling factor for each group, and can approximate the original weights by multiplying the quantized weights by the scaling factor:

$$\tilde{\mathbf{X}}_{F32} \approx s_k \cdot \mathbf{X}_{I8}$$

The error of our approximation is computed by the L1 norm of the difference between the two matrices $\|\mathbf{X}_{F32} - \tilde{\mathbf{X}}_{F32}\|_{L1}$. The reason for splitting the weights into batches is to reduce error caused by high-magnitude weights. Applying this algorithm to the pretrained Llama2 weights with a group size of $g = 64$, we obtain a maximum reconstruction error of 0.007 across all weights.

The memory size of each layer's weights in FP32 and INT8 is shown in Figure 3. Running an unoptimized forward pass of the FP32 model requires at least 27 GB of memory to hold the full weights. The INT8 version is more lenient, requiring just over 7.5 GB of memory. Of course, while loading all the weights into memory eliminates expensive disk reads, one can observe that most weights are not involved in computation at any particular moment. Our next optimization exploits this observation to free up unused memory space.

One final note about quantization is that ARM processors, including the one in our Raspberry Pi, do not have instructions that operate on sub-byte granularity. In our early testing, we found that running the model with INT4 quantization caused it to run slower due to the extra computation required for casting the weights to an 8-bit data type. Thus, we limit our experiments to INT8 quantization only.

Hyperparameters	
`dim`	4096
`hidden_dim`	11008
`n_layers`	32
`n_heads`	32
`n_kv_heads`	32
`vocab_size`	32000
`seq_len`	2048

Figure 2: Default Llama2-7B hyperparameters

Layer	**Dimension**	**FP32 (Bytes)**	**INT8 (Bytes)**
`token_embedding_table`	(vocab_size, dim)	524,288,000	139,264,000
`rms_att_weight`	(layer, dim)	524,288	139,264
`wq`	(layer, dim, n_heads × head_size)	2,147,483,648	671,088,640
`wk`	(layer, dim, n_kv_heads × head_size)	2,147,483,648	671,088,640
`wv`	(layer, dim, n_kv_heads × head_size)	2,147,483,648	671,088,640
`wo`	(layer, n_heads × head_size, dim)	2,147,483,648	671,088,640
`rms_ffn_weight`	(layer, dim)	524,288	139,264
`w1`	(layer, hidden_dim, dim)	5,771,362,304	1,533,018,112
`w2`	(layer, dim, hidden_dim)	5,771,362,304	1,533,018,112
`w3`	(layer, hidden_dim, dim)	5,771,362,304	1,533,018,112
`rms_final_weight`	(dim,)	16,384	4,352
`wcls`	(vocab_size, dim)	524,288,000	139,264,000
Total Size (Bytes):		**26,953,646,080**	**7,562,219,776**

Figure 3: Llama2 layer memory requirements. Note that the memory required for INT8 is slightly more than `fp32_memory /4` since we store scaling factors. The `rms` weights are used in normalization. The `wq`, `wk`, `wv`, and `wo` weights are used in multihead attention. The `w1`, `w2`, `w3` weights correspond to our three feedforward layers. `wcls` is the weight of the final classifier layer.

4.3 Iterative Loading of Weights

We implement a novel memory optimization that iteratively loads weights for each transformer layer in the forward pass. Since the Pi Zero only has a single-core CPU, there is no speedup achieved by parallelizing computation. Thus, storing the weights for all transformer layers in memory is unnecessary since we only use one layer at a time. Since our Pi does not have virtual memory, this modification also allows us to load more weights into memory than we otherwise would be able to.

We shard each layers' weights into a separate file and implement a segmented forward pass that loads layer weights one at a time. This approach incurs substantial performance costs from the frequent disk reads, but in return allows us to scale our memory usage by a factor of `num_layers`. The updated requirements to hold a single transformer layer in memory is shown in Figure 4.

Layer	FP32 (Bytes)	INT8 (Bytes)
rms_att_weight	16,384	4,352
wq	67,108,864	17,825,792
wk	67,108,864	17,825,792
wv	67,108,864	17,825,792
wo	67,108,864	17,825,792
rms_ffn_weight	16,384	4,352
w1	180,355,072	47,906,816
w2	180,355,072	47,906,816
w3	180,355,072	47,906,816
Total Size (Bytes):	**809,533,440**	**215,032,320**

Figure 4: Memory requirements for a single Transformer layer.

Theoretically one could extend this approach to only ever store one set of weights in memory at a time. In that case, the lower bound for the amount of memory necessary to run a forward pass of the model would be the size of the largest weight layer, which from Figure 4, we see is one of the feedforward layer at 45 MB. However, if our method of loading individual layers wasn't already prohibitively slow, this approach would certainly render our model completely useless.

5 Experiments and Analysis

Despite our best efforts, we were unable to run a forward pass of Llama on the Raspberry Pi Zero. We implemented all previously-described memory optimizations — pruning the feedforward weights by 50%, quantizing to INT8, sharding weight files and loading individual transformer layers — and additional, more-aggressive optimizations — eliminating multi-head attention and just using a single, randomly-selected attention head; allocating space for just one (pruned) feedforward layer and iteratively loading feedforward weights; removing the tokenizer, embedding table, and final classification layer — but were still unsuccessful. We estimate that this extremely stripped-down version of the model still requires about 700 MB of memory, the bulk of which ($\sim$569 MB) is needed to hold the state of the model. We didn't try more-aggressive optimizations that save and load unused portions of the model state to disk, although that will likely work.

Despite this disappointing outcome, we still wanted to evaluate the effectiveness of our strategies. To this end, we modified our code to run on an Intel-processor Mac and profiled its performance. For each model, we performed multipled inference runs for up to 256 steps with the prompt "hello world," a fixed random seed of 0, and all other hyperparameters set to their default values.

5.1 FP32 vs. INT8

We first measured the speedup from quantizing the model to INT8. Figure 5 shows the measured wall time for forward passes of the FP32 and INT8 models. The FP32 model reports an average of 210.10 sec/token, while the INT8 model reports an average of 7.52 sec/token. This gives us a substantial 28x speedup for the quantized INT8 model.

By tracing heap calls, we measured the total amount of heap-allocated memory for the FP32 forward pass to be 44.3 GB, while the INT8 pass allocated 9.4 GB. This memory usage includes the weights, additional variables holding the run state, the SentencePiece tokenizer, and a sampler that samples tokens from the model's final logits.

5.2 Full Weights vs. Segmented

We measured the difference in inference speed between storing all model weights in memory and loading in layers one-by-one. Figure 6 shows a comparison between the two approaches. The INT8 segmented approach generates at an average of 69.8 sec/token, while the INT8 model with fully-loaded weights generates at 7.52 sec/token. This means that the segmented pass experiences a 9x slowdown due to the additional disk reads. While this is a considerable decrease in performance, the segmented implementation only allocates 5.5 GB of memory, corresponding to roughly a 40% decrease. Note that this 5.5 GB improvement is only with our optimization to load individual transformer layers.

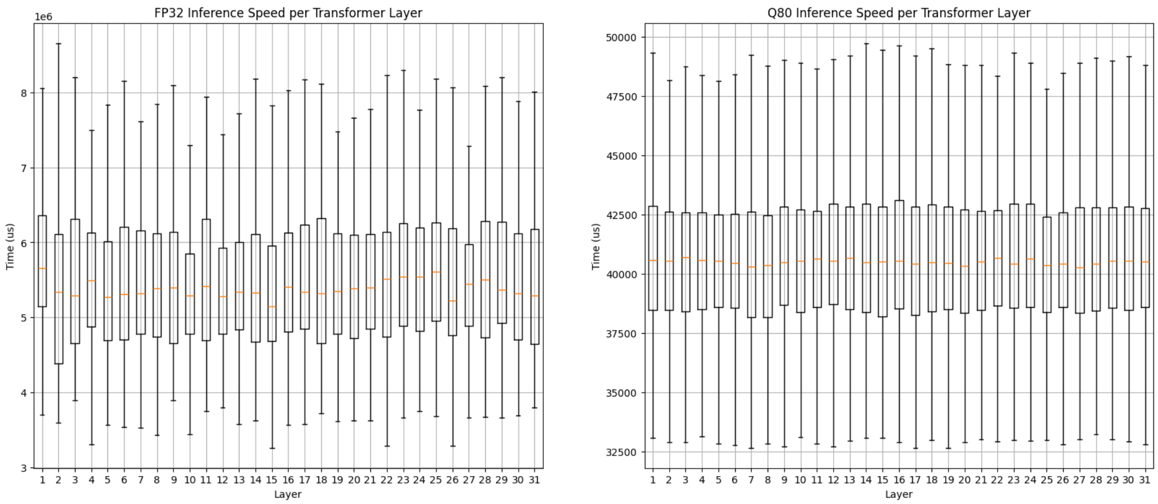

Figure 5: Per-layer performance comparison between the FP32 and INT8 forward passes.

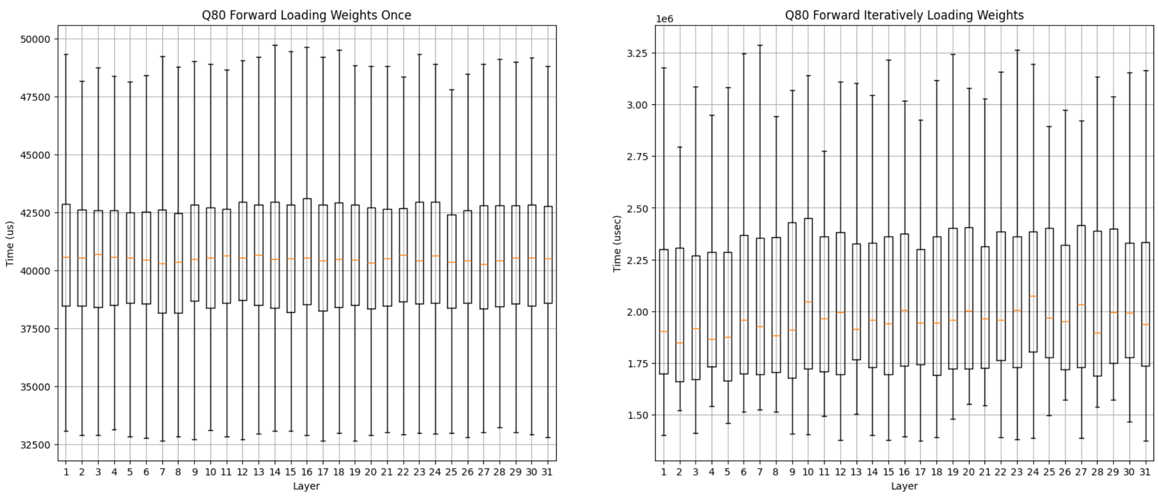

Figure 6: Per-layer performance comparison between full-weight and segmented forward passes

We were also curious about whether the model's performance varied depending on the tokens it had generated. Figure 7 plots this measurement, comparing the full-weight and segmented INT8 models across 92 steps. Interestingly, the segmented pass runs consistently quickly for the first few tokens before gradually slowing down. This result persisted across multiple trials, but we are uncertain of its underlying cause.

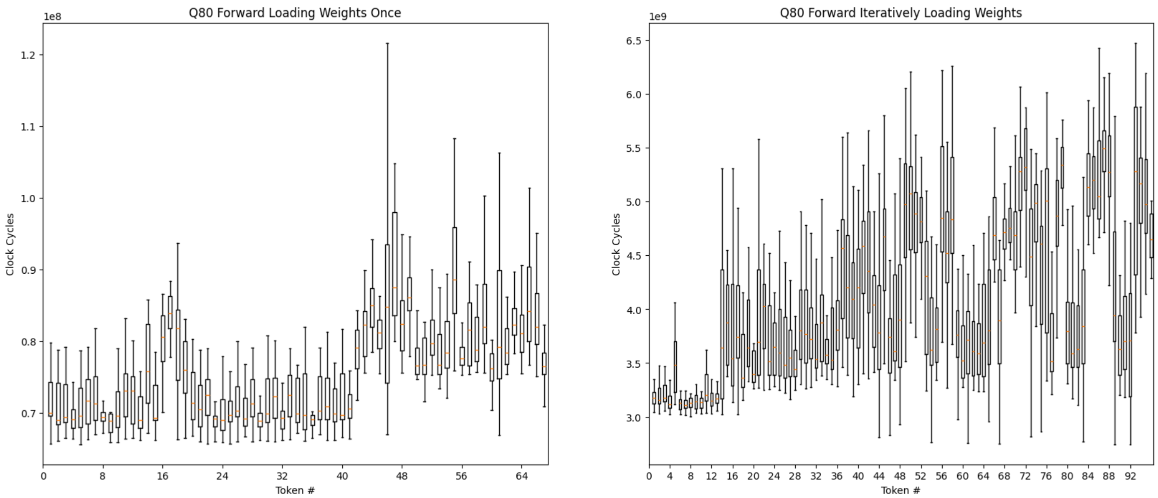

Figure 7: Per-token performance comparison between full-weight and segmented forward passes

6 Conclusion

In this paper, we explored various memory optimizations to run a forward pass of Llama2 on a Raspberry Pi Zero. Although we did not achieve our initial goal of performing inference on the Pi, the process allowed us to evaluate the performance and memory usage of these optimizations on real-world hardware. Our findings reveal promising memory improvements. Our methods are generalizable and can be used to improve memory efficiency in other applications.

We attempted to get hold of a Raspberry Pi with more RAM for further testing, but were unsuccessful before the paper deadline. Based on our current findings, we believe that it may be possible to run Llama2 inference with just 2GB of RAM. Future work could potentially test this hypothesis and refine our methods to push this hypothetical bound even lower.

In our investigation, we benchmarked individual matrix operations on the Pi and found that a single matrix multiplication (of which there are a few hundred in a single forward pass) could take upwards of 30 minutes. It is then likely that even if we could get Llama to run on the Pi without running out of memory, our single-core processor would be too slow for any practical use. However, this whole paper eschews practically anyways — any computer bootloading the inference code to the Pi could easily run it itself — so we encourage further development. Hack away.

References

Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher De Sa. 2024. Quip: 2-bit quantization of large language models with guarantees.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Llm.int8(): 8-bit matrix multiplication for transformers at scale.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2023. Gptq: Accurate post-training quantization for generative pre-trained transformers.

Song Han, Huizi Mao, and William J. Dally. 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models.

Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and training of neural networks for efficient integer-arithmetic-only inference.

Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W. Mahoney, and Kurt Keutzer. 2024. Squeezellm: Dense-and-sparse quantization.

Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. 2024. The era of 1-bit llms: All large language models are in 1.58 bits.

Sharan Narang, Erich Elsen, Gregory Diamos, and Shubho Sengupta. 2017. Exploring sparsity in recurrent neural networks.

Mingjie Sun, Zhuang Liu, Anna Bair, and J. Zico Kolter. 2023. A simple and effective pruning approach for large language models.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. Smoothquant: Accurate and efficient post-training quantization for large language models.

A Appendix

Here we include some generated text from Llama2 that we thought was amusing or interesting. First, we have some samples generated with varying levels of L1-pruning to all three of the feedforward weights. The seed for these generations is set to 0. As the percentage of weights pruned increases, our model transforms from a 20 year-old Dutch girl to a German FIFA fan.

%	Output
0.001	[same output as no pruning] hello world! I'm a 20 year old girl from the Netherlands. I'm a student and I'm studying to become a teacher. I'm a very open minded person and I love to meet new people. I'm a very social person and I love to have fun. I'm a very outgoing person and I love to go out with my friends. I'm a very active person and I love to do sports. I'm a very creative person and I love to do art. I'm a very passionate person and I love to do music. I'm a very romantic person and I love to do love. I'm a very caring person and I love to do care. I'm a very honest person and I love to do honesty. I'm a very loyal person and I love to do loyalty. I'm a very trustworthy person and I love to do trust. I'm a very reliable person and I love to do reliability. I'm a very responsible person and I love to do responsibility. I'm a very respectful person and I love to do respect. I'm a very supportive person and I love
1	hello world! I'm a 20 year old girl from the Netherlands. I'm a student and I'm studying to become a teacher. I love to read, write and draw. I'm a huge fan of Harry Potter and I'm a huge fan of the Hunger Games. I'm a huge fan of the Twilight Saga and I'm a huge fan of the Mortal Instruments. I'm a huge fan of the Percy Jackson and the Olympians. I'm a huge fan of the Divergent Series. I'm a huge fan of the Maze Runner. I'm a huge fan of the Vampire Academy. I'm a huge fan of the Infernal Devices. I'm a huge fan of the Delirium. I'm a huge fan of the Delirium. I'm a huge fan of the Delirium. I'm a huge fan of the Delirium. I'm a huge fan of the Delirium. I'm a huge fan of the Delirium. I'm a huge fan of the Delirium. I'm a huge fan of the Delirium
5	hello world, i'm back! nobody's perfect, but i'm trying to be. i'm a 20 year old girl who loves to write. i'm a writer, a reader, a dreamer, a thinker, a believer, a lover, a fighter, a friend, a sister, a daughter, a student, a teacher, a leader, a follower, a listener, a talker, a thinker, a doer, a dreamer, a believer, a lover, a fighter, a friend, a sister, a daughter, a student, a teacher, a leader, a follower, a listener, a talker, a thinker, a doer, a dreamer, a believer, a lover, a fighter, a friend, a sister, a daughter, a student, a teacher, a leader, a follower, a listener, a talker, a thinker, a doer, a dreamer, a believer, a lover, a fighter, a friend, a sister, a daughter, a student, a teacher, a leader, a follower, a listener
10	nobody's perfect, but i'm trying to be. i'm a 20 year old girl who loves to write. i'm a writer, a reader, a dreamer, a thinker, a believer, a lover, a fighter, a friend, a daughter, a sister, a student, a teacher, a leader, a follower, a believer, a lover, a fighter, a friend, a daughter, a sister, a student, a teacher, a leader, a follower, a believer, a lover, a fighter, a friend, a daughter, a sister, a student, a teacher, a leader, a follower, a believer, a lover, a fighter, a friend, a daughter, a sister, a student, a teacher, a leader, a follower, a believer, a lover, a fighter, a friend, a daughter, a sister, a student, a teacher, a leader, a follower, a believer, a lover, a fighter, a friend, a daughter, a sister, a student,
20	hello world, I'm a newbie here. Unterscheidung der verschiedenen Marken genau jenes Mittel zu kaufen, das zu Hause zum besten Lieferanten für. The 2018 FIFA World Cup was the 21st FIFA World Cup, an international football tournament contested by the men's national teams of the member associations of FIFA once every four years. It took place in Russia from 14 June to 15 July It was the first World Cup to be held in Eastern Europe, and the 11th time that it had been held in Europe. At an estimated cost of over $ billion, it. The 2018 FIFA World Cup was the 21st FIFA World Cup, an international football tournament contested by the men's national teams of the member associations of FIFA once every four years. It took place in Russia from 14 June to 15 July It was the first World Cup to be held in Eastern Europe, and the 11th time that it had been held in Europe. At an estimated cost of over $ billion, it. The 2018 FIFA World Cup was the 21st FIFA World Cup, an international football
30	hello world, 2017 edition Unterscheidung, 2017 edition 1000000, 2017 edition 1000000, 2017 edition 1000000, 2017 edition 1000000, 2017 edition 1000000, 2017 edition 1000000, 2017 edition 1000000, 2017 edition 1000000, 2017 edition 1000000, 2017 edition 1000000, 2017 edition 1000000, 2017 edition 1000000, 2017 edition 1000000, 2017 edition 1000000, 2017 edition 1000000, 201

Here are some other outputs from testing that we particularly enjoyed reading. It's a bit concerning that these may be partially excerpted from real people's online posts...

hello world meme is there in python? Heh. My first post! :) What if you could make a Wikipedia post that everyone who clicks on it would find amazing? Or, how about a post that makes everyone who clicks on it wonder "why the hell do they think anyone would want to read this?" You can do that! You don't need to be good or smart, just log on and find out how. Congratulations, you are now officially the lowest of the low: a wikipedian. Just to encourage you to keep trying, here are some things that the editors of Wikipedia have said. Hopefully, you'll find one of them motivating to you: Members of Wikipedia's community can do anything. The answer to all problems is not 'the admins'. They are supposed to be people who are unwilling to accept the 'the admins' answer for all problems. -Geoffrey Chaucer, from this article. Do not take on more than you can chew, dear wikipedian. There is no way that you can know how to contribute effectively, but you can find out what you do want to contribute,vanil

hello world! (July 2008)

Notes
- We are on Facebook

Content
- Today's progress on the C programmers guide

36

</s>

hello world by Ladycrimson
A/N: 22 11/18/07
It's funny how things change in the blink of an eye. One moment you're fighting your life away, looking death in the face with your arms outstretched, begging for him to just take you and set you free from all the torture that you've been through. And the next moment you're finding yourself in the strangest place you've ever seen, surrounded by angels, your god himself, and your mate who's scaring you to death. I mean, how often does your mate actually betray you, even if he was a fae, after all? I didn't think I was ever going to see him again, or well, any of the others for that matter.

hello world, now what? get a little love for your company 45+years as a Service Professional Jumping into entrepreneurship Join forces with 2 other business owners in business and life Started his first "brand" under the "Mercedes" label (blue bird) Mercedes Benz representitive for Northern NJ and started delivering cars in the area Sells 11 cars in his 1st year Covers NY, NJ, CT, PA, and DE Sells over 200 cars Moving into franchises Expands to Volvo and Mazda Sells his dealership to the Bennett Family Takes a job with Direct Tire in NJ Moving into real estate Becomes a top producer in NJ Started Mercedes Benz of Larksville Started 2 real estate offices in NJ Started Greenwood Realty in Florida Mercedes Benz dealership failed 2001 move to Hudson Valley 2003 move to Florida Stuart took a real estate sales position to learn the market while pursuing his sales career Found Success Magazine