

Lossless and Near-Lossless Compression for Foundation Models

Moshik Hershcovitch^{1,2}, Leshem Choshen^{1,3}, Andrew Wood⁴,
Ilias Enmouri¹, Peter Chin⁵, Swaminathan Sundararaman¹, Danny Harnik¹

¹ IBM Research ² Tel-Aviv University ³ MIT

⁴ Boston University ⁵ Dartmouth University

Abstract—With the growth of model sizes and scale of their deployment, their sheer size burdens the infrastructure requiring more network and more storage to accommodate these. While there is a vast literature about reducing model sizes, we investigate a more traditional type of compression – one that compresses the model to a smaller form and is coupled with a decompression algorithm that returns it to its original size – namely lossless compression.

Somewhat surprisingly, we show that specific lossless compression can gain significant network and storage reduction on popular models, at times reducing over 50% of the model size. We investigate the source of model compressibility and introduce specialized compression variants tailored for models that further increase the effectiveness of compression.

We also categorize models to compressibility groups and introduce a tunable lossy compression technique that can further reduce size even on the group of less compressible models with little to no effect on the model accuracy. Finally, we explore the usefulness of delta compression for checkpointing and model variations.

We estimate that these methods could save over an ExaByte per month of network traffic downloaded from a large model hub like Hugging Face.

I. INTRODUCTION

With scale, we have learned that models gain performance and with it, gain popularity. With scale, models also require more memory and with popularity more communication bandwidth. Taken together, we observe strains on communication bottlenecks that call for efficient solutions. Storage requirements, while often ignored, may accumulate to hundreds or thousands of times the size of a model if checkpoints [1] or distributed updates are to be saved (c.f., VI) [2]–[4].

Similarly, models are repeatedly moved around in multiple channels: from a storage hub to inference machines; from training/fine-tuning nodes to the storage backend; between GPU nodes during distributed training and so on. Network hubs epitomize the strains by model size. For instance, with over 14.5 GBs and 2.77 M downloads per month from Hugging Face [5] Mistral [6] alone requires 40 PBs of transferred information a month.

A large body of work has been aimed at reducing model sizes focusing on the number of computations in inference. Such methods transform the model into a smaller one in an irreversible fashion. For example, distillation [7], pruning [8] and quantization [9] either remove nodes from the network or reduce each parameter size. Since these methods main focus is on inference speed, they are bound to create a format of

an actual running model. As such, they don’t necessarily push the space saving to its limit, and are not stored in the minimal possible way.

In this work, on the other hand, we follow a more traditional definition of compression typically used for networking and storage. Compression that is also accompanied by a decompression process, returning a model to its original size and usability. This definition encompasses among other things all forms of lossless compression.

Surprisingly, we observe (§III) that even standard lossless compressors like zlib [10] or zstd [11] can achieve meaningful savings and these can be further amplified using specialized modifications to the compressors. While common rationale expects model parameters to have high entropy and therefore be non-compressible, we find that in reality there is ample redundancy in representation. We classify popular models from Hugging Face [5] into three categories with distinct compressibility traits helping to understand when and for what models it is beneficial to employ lossless compression.

After exploring the source of compressibility in models we introduce *byte grouping* – an adaptation that is tailored for the models use case (§III-B). The method rearranges the bytes in a model to compress the different bytes of the parameters together. This results in grouping of similar bytes which in turn yields better compression.

We make another key observation, that fine-tuning of models often degrades their compressibility (at times significantly). This high entropy in the parameters often stems from minuscule updates. To overcome this, we introduce a novel tunable lossy compression method that can significantly improve compression ratio with no measurable harm to model accuracy. In a nutshell, this technique allows for incurring controlled inaccuracies to parameters, under the assumption that a lot of the entropy in model weights is actually redundant, i.e., noise saved to disk. Surprisingly, we find ranges where those precision reductions can even slightly benefit the model, corroborating a few similar findings (see §VI). While the goal of this work is not to affect models, this unexpected phenomenon is worth mentioning.

Finally, we explore the benefits of delta compression and show that by compressing the delta between two similar models one can achieve compression far greater than compressing a standalone model. This is useful for checkpointing and management of model variations.

TABLE I
TOP RANKED DOWNLOADED MODELS FROM HUGGING FACE AND THE POTENTIAL TRAFFIC SAVINGS IF COMPRESSED.

MODEL NAME	MODEL SIZE	#MONTHLY DOWNLOADS	RANK	COMPRESSION RATIO	POTENTIAL SAVINGS
Wav2Vec	1.26 GB	63M	#1	85.2%	11.7 PB
BERT	0.4 GB	43.7M	#2	85.3%	2.6 PB
ROBERTA	0.5 GB	15M	#3	47.0%	4 PB
GPT2	0.5 GB	14.6M	#4	78.1%	1.6 PB
CLIP	1.7 GB	14.3M	#5	50.1%	12.2 PB
MISTRAL	14.5 GB	2.77M	#~60	71.0%	11.6 PB
BLOOM	328.2 GB	278K	#100+	71.4%	26.1 PB

II. BACKGROUND

A. Motivation - use cases

With small models weighing about a Gigabyte [12] and large ones Terrabytes [13] storage by itself is an issue for many purposes. However, common use cases require many model types or model versions and hence increased resources. We list some below as a motivation.

1) *Model Hubs*: Large model repositories or hubs like Model Zoo [14], PyTorch [15], Tensorflow [16], Adapter [17], Hugging Face [5], IBM watsonx.data [18] and Qualcomm® AI Hub [19] hold a large number of models and serve numerous download requests of popular models. As of 2024, Hugging Face, the largest model hub, transfers PetaBytes of data every day, primarily downloaded data. Table I shows some of the top ranked models¹ and their compression ratio (using the methods described in Section III-A). As seen, the potential traffic savings from compression is substantial. Note that the same trends also apply to models that are not downloaded as often, for example, the Bloom model offers significant savings due to its large initial size.

In this use case, there are three ways in which compression can be beneficial, the first and the most important is to reduce the amount of data transferred, the second is to reduce the amount of data stored and the third is to reduce the time to download and upload those models.

2) *Distributed Training*: transfer data between nodes during training to overcome the need to save the full model and computation on a single GPU/node. In some methods, only the model weights are transferred between nodes and in other methods, the optimizer weights and gradients are transferred as well [20]. Either way, distributed training is usually limited by data transfer between nodes, so compressing this data can help train larger models.

3) *Decentralized algorithms*: Along the different ways to distribute the computation along nodes. Some methods propose to alternate training or the models themselves to accommodate different contributors training the same model. This stems from federated learning that contributes gradients [4], to contributing partially trained models [21], from changing the kinds of updates done [22] to relying on volunteer computing [23], or even relying on different objectives and expertise all merged into the same model [3]. All of those methods

inherently transfer and store a lot of model versions, which also drove dedicated version control frameworks [2],

4) *Checkpoints and Versions*: During model creation, multiple intermediate versions of the models are commonly saved. This often includes tests on the training regime such as hyperparameter tuning [24]. Even during the training of a single model, the current model is periodically checkpointed to recover after a crash, to select the best checkpoint from a few options [25], for analysis [1], improve performance [26], [27] etc. Even though saving during checkpointing rarely slows the training time, it does burden the networking and storage, limiting the frequency and amount of saved and shared checkpoints, which are encouraged by the community (e.g.; [1], [28]).

B. Models Structure and Types

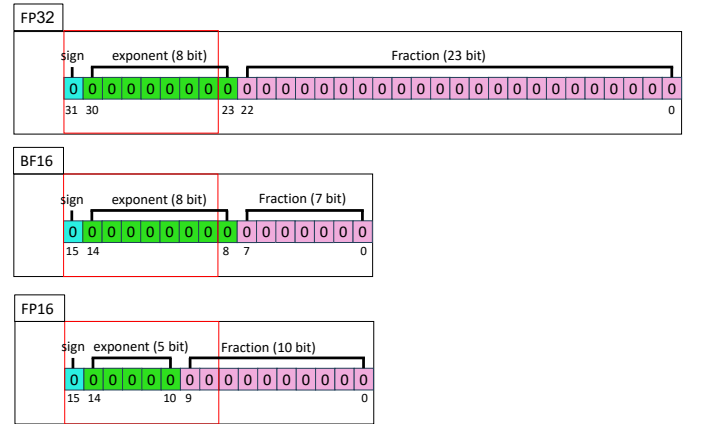


Fig. 1. FP32, a sign bit + 8 bits exponents + 23 bits of mantissa. BF16, a sign bit + 8 bits exponents + 7 bits of mantissa. FP16, a sign bit + 5 bits exponents + 10 bits of mantissa

a) *Models*: Regardless of the architecture, current models are mainly a function of many matrices or tensors of different sizes and a code that can read the parameters in the matrix and convert it to a function. While a layer may contain several such tensors, for brevity we call each tensor a layer. We note that the code is negligible in weight and hence the main focus of our compression is reduced to tensors, or even put more simply, long arrays of numeric parameters.

¹Rank as of August 2023 except for Mistral taken in March 2024.

One main variant in models that strongly affects compression tendencies is the type of parameters that make up a model. Parameters typically represent real numbers and as such the most straightforward (and popular) approach is to hold them using *floating point* numbers. Floating point is a way to represent real numbers using a fixed number of bits but a flexible scale that allows for larger numerical ranges. In a nutshell, floating point contains an exponent part - representing the range in which the real number lies, and a *mantissa* or *fraction* representing the actual number within this range. To this, a sign bit is added indicating whether a number is positive. For example, FP32 is a 32 bit floating point number with a sign bit, an 8 bit exponent and a 23 bit mantissa (see Figure 1). The real number is calculated by $(-1)^{sign} \cdot 2^{exponent-127} \cdot 1.fraction$. Another popular parameter type used for models is BF16 [29] which simply cuts the tail end of the fraction (hence reducing the precision level) but maintains the same exponent as shown in Figure 1.

III. COMPRESSION FOR MODELS

In this section, we introduce the basics of lossless compression variants that are relevant to model and present our variant of lossy compression. Note that in this paper, we evaluated only compression run in the CPU, since the GPU computations and GPU memory are more valuable resources, whereas CPU computations and CPU memory are typically abundant.

A. Lossless compression

Lossless compressors are the traditional form of compression and are widely used for reducing network and storage overheads in all fields of computing. They consist of two algorithms - compression and decompression where after applying both in sequence the output returns to the exact same state. There are countless compression techniques, those vary in the tradeoff between compressibility and compression/decompression time (see for example; [30]).

Throughout the paper, we measure the *compression ratio*. Namely, the percentage of the data that is left after compression - *lower is better*. For example, if the method compresses a GB into a quarter of a GB it has a compression ratio of 25%.

The main techniques employed in lossless compression are based on repetition removal (stemming from the seminal work of [31]) and entropy encoding (e.g. [32], [33]) which reduces the entropy seen at a byte level and represents such bytes at a bit level granularity.

We observed that compressors that employ solely repetition removal (such as LZ4; [34]) are not very beneficial when compressing models. This is expected since to remove a repetition a span of multiple parameters should repeat itself. However, model tensors are not structured and parameters typically do not have an affinity with their neighbours, making repetitions that span multiple parameters scarce. For the rest of our experiments, we choose compressors that also use entropy encoding such as Zlib [35] or Zstd [36]. In our experiments, we chose Zstd as the underlying compressor/decompressor due to its superior speed vs. compression tradeoff [30], [36]. Note that

once Byte Grouping (see §III-A1) is used, then LZ4 fares much better, but still Zstd is far superior in terms of compression ratio. For example, for RoBERTa we get 47.0% with Zstd vs. 56.7% with LZ4 and for Bloom we get 71.4% with Zstd as opposed to 80.5% with LZ4.

1) *Understanding Model Compressibility and Byte Grouping*: Initially, one may expect models to be non-compressible and show high entropy, as parameters may encode unpredictable information and differ from each other. This is correct to a certain degree, but in reality, the actual range in which parameters reside is typically limited, which reduces the entropy and opens the door for compression to be effective.

Take a model with FP32 parameters for example. It will have high entropy in its fraction (or mantissa), but relatively low entropy in the exponent, as the parameter scales are quite limited. Therefore it makes sense to compress exponents and mantissa bytes separately. Indeed, compressing the exponent bytes without the interference of the mantissa bytes yields higher compressibility.

We suggest **Byte Grouping** which groups together bytes from the same position in all the model's parameters. If each parameter in the model consists of several bytes (typically 2 or 4 bytes), then group together the first byte from all parameters, then the second byte, etc., as shown in Figure 2. Note that since typical models use the same parameter type throughout the model, then byte grouping can be done without real knowledge on the model structure (except parameter type) and for example can be executed even on models that are already stored as a binary file.

Our test shows that byte grouping the data before compressing it can improve the compression ratio between 7%-30% (see Section III-B).



Fig. 2. An example for Byte Grouping, each parameter has 4 bytes and we group them into 4 arrays.

2) *The Sign Bit*: Another observation is that the sign bit tends to hold high entropy and that compressing it together with the exponent byte interferes with compression effectiveness. To overcome this, we consider the following approach to deal with the sign bits: translate the stream into an unsigned stream (for example, using the *abs* function) and store the sign values separately. The unsigned values are then fed into the compressor. This method can further improve the compression ratio without further affecting the precision. However, with lossless compression our experiments showed that the compression benefit is relatively low - on the order of 1%, making it unappealing due to its computational overhead.

B. Model compressibility with Lossless Compression

While model compressibility has high variance, we observe that there are essentially three popular categories of models

TABLE II
COMPRESSION RATIO OF MODELS AFTER ZSTD COMPRESSION WITH BYTE GROUPING.

MODEL NAME	PARAM TYPE	MODEL SIZE	COMPRESSION RATIO	COMPRESSION RATIO PER BYTE GROUP
WAV2VEC	FP32	1.2 GB	85.2%	(42.9%, 99.0%, 99.0%, 98.6%)
BERT	FP32	0.4 GB	85.3%	(41.2%, 99.0%, 99.0%, 99.0%)
GPT2	FP32	0.5 GB	78.1%	(38.9%, 90.8%, 90.8%, 90.8%)
ROBERTA(1 EPOCH)	FP32	0.5GB	80.7%	(42.9%, 99.9%, 93.5%, 86.4%)
ROBERTA(9 EPOCHS)	FP32	0.5GB	82.5%	(42.9%, 99.9%, 96.9%, 90.2%)
STABLE-VIDEO-DIFFUSION	FP16	4.27GB	84.9%	(69.8%, 100%)
CAPYBARAHERMES-MISTRAL	FP16	14.5 GB	83.7%	(68.7%, 98.7%)
ROBERTA	FP32	0.5 GB	47.0%	(42.9%, 99.9%, 44.7%, 0.005%)
XLN-ROBERTA	FP32	1.1 GB	45.7%	(42.6%, 95.7%, 44.6%, 0.002%)
CLIP	FP32	1.7 GB	50.1%	(42.1%, 99.0%, 49.0%, 8.0%)
T5 BASE	FP32	0.8 GB	35.7%	(42.6%, 99.9%, 0.005%, 0.005%)
LLAMA-13B	FP16	26 GB	66.8%	(69%, 64%)
TULU-7B	FP16	13.5 GB	66.6%	(68.9%, 64%)
FALCON-7B	BF16	14.4 GB	71.3%	(42.7%, 100%)
BLOOM	BF16	328.2 GB	71.4%	(42.3%, 100%)
OPENLLAMA-3B	BF16	6.9 GB	71%	(42.1%, 100%)
MISTRAL	BF16	14.5 GB	71%	(42.0%, 100%)

from a compressibility standpoint. Table II shows examples of models from the three various categories, all compressed using Zstd with its default setting (level 3) and employing the Byte Grouping technique.

The first category of models are mainly compressible in the exponent and hence have more modest savings, on the order of 15-20% (namely with a compression ratio of ~ 80 -85%). Those models are saved in FP32 or FP16. The main source of compressibility for these models is the exponent byte which is highly compressible (around 40%), but the other bytes hardly compress at all.

The second category includes “clean” models, or base models. These have high compressibility stemming from both the exponent and the two lower bytes of the mantissa. The second byte, in all cases, is incompressible and holds most of the model’s entropy. Overall these models show very high compressibility (reducing the size by 50-65%) and prove very attractive for compression. Note that some of the most downloaded models from Hugging Face fall into this category. This leaves the two lower bytes zeroed. We call these clean models because after fine tuning they lose much of their compressibility. For example, We fine tuned the RoBERTa model (using the Rotten Tomatoes public dataset [37]) and see that even after a single epoch of fine-tuning the model falls into the first category and only saving 20%. After 9 epochs of fine-tuning the compressibility is even slightly worse. The source of compressibility in the clean models stems from using only parts of the available entropy during the training of these models. For example, the T5 model is trained in a 16-bit environment and then cast into an FP32 parameter for further fine-tuning [38]. In other models the lower bytes do have some entropy in them, but not as high as the random looking bytes.

The final category is of BF16 models that show ~ 30 % space savings. Like the first group, the exponent is very compressible, and the mantissa is not, but in these models,

the savings are more significant as the exponent makes up a larger part of the model.

In the evaluation above we used mainly highly downloaded models from the Hugging Face hub to check for realistic scenarios (see full model list in Appendix A). We include models from different modalities, sizes, architectures and saving float formats. We split them to the 3 groups mentioned above.

To account for how commonly each model was downloaded or what version was tested (models are rarely changed after upload, but technically one can commit an update) all numbers were gathered in August 2023 except for the Mistral model and the FP16 models which were updated in March 2024.

a) *The benefit of Byte Grouping:* Byte grouping benefits also vary between the categories. Byte grouping reduces the size of the compressed model by 7-8.2% on the first category, by 19-27% on the clean model category and by 8.5-10% on the BF16 category. We note that if LZ4 was used then the effect of byte grouping is massive and without byte grouping these compressors manage nearly no compression at all. For example, LZ4 on RoBERTa fails badly and achieves only 95% compression ratio, but with byte grouping, this jumps to 56%.

C. Tunable Lossy Compression

The observation that fine-tuning greatly diminishing the model compressibility suggests that tweaking of parameters, even if very minor, introduces a lot of entropy. This phenomena is amplified by the use of floating point arithmetic which, by design, invests a significant amount of bits even to very small numbers. However, in reality, parameters with very small numbers tend to have a very minor effect on the results of model inference, if at all. This suggests that removing some of the entropy dedicated to very small numbers could increase model compressibility without actually changing the model results or accuracy.

Thus we introduce a *tunable lossy compression* technique that may significantly improve compression savings at the expense of small measurable changes to the original model. In a nutshell, the proposed method casts every parameter into an integer representation with a chosen level of fixed precision, in essence trimming some of the least bits. Then compression follows as before using byte grouping and a standard lossless compressor. The full decompression of our tunable lossy technique returns the model to its original format albeit zeroing some information that resided in the least bits.

Formally, given a parameter θ in floating point representation, and precision $B = 2^b$ the casting is done as follows. First, multiply the parameter by the precision factor and then cast it into an integer, effectively rounding it to $\lfloor \theta \cdot B \rfloor$. The transformed parameters are then fed into a standard lossless compressor. During decompression, the stream first undergoes standard decompression and then the resulting integers are transformed into floating point after division by the precision factor. Note that floating point parameters typically lie in the range $[-1, 1]$. The meaning of multiplying by the precision factor and rounding is that anything smaller than 1 after the multiplication is discarded. Thus, the greater the precision factor, the more of the parameter’s original entropy is kept. Choosing a precision factor of 2^b essentially means that we only discard quantities that are smaller than 2^{-b} . Maintaining a higher precision implies preserving more information or less “lossy” compression.

A small tweak we add deals with outlier values such as parameters that are outside of the range $[-1, 1]$. These are problematic because when multiplied by the precision factor they may overflow the integer range. In such a case we forgo compression of the layer and mark this layer as non-compressible. Note also that compressing unsigned integers and adding the signed bits separately (as described in Section III-A) turns out to be more significant with tunable lossy compression, so our tests include this optimization as well. The improvement from separating the sign bit grows as the precision factor drops – from $\sim 3\%$ in the case of $B = 2^{27}$ to more than 16% for $B = 2^{15}$.

D. Compression vs. Accuracy with Tunable Lossy Compression

One desirable trait of this method is that the amount of information lost is tunable, and determined by the precision factor chosen. The main question is then, how to choose this precision factor?

We offer some rules of thumb that help identify levels that we can push our precision factor to without harming the accuracy of general models. We expect that there is an accuracy level of weight information beyond which information is actually redundant for the computation. This is basically true by construction, as training mechanisms introduce such errors during training, considering it beyond their precision. For example, in order to avoid zero values, Adam’s epsilon defaults in PyTorch [39] and Tensorflow [40] are set to 10^{-8} and to 10^{-7} in Keras [41] deeming anything below this precision irrelevant. Under this assumption rounding anything below

10^{-8} or 10^{-7} is a safe choice and choosing a precision factor of $B = 2^{27}$ or $B = 2^{24}$ (respectively) will not insert more noise than what is done naturally in the training process.

A further argument postulates that Floating Point 32 arithmetic inserts precision error all the time - for example, when adding very small numbers to relatively large numbers, the small numbers may lose their accuracy. For FP32 this happens naturally during computations (such as inference and training) introducing errors of up to 2^{-23} . Under this rationale, using a precision factor of $B = 2^{23}$ is not expected to change overall outcomes more than what arbitrary FP32 operations might do.

Using $B = 2^{23}$ offers significant savings and improves compression ratio significantly for the FP32 models, reducing the compressed model by an additional 20%, for example reducing wav2vec compression ratio from $\sim 85\%$ to $\sim 68\%$.

To test this rationale we checked two base models with measurable fine-tuned performance. The first set of fine-tuned models are trained variants of T5 on CNN-DM [42], XSUM [43], SQUAD [44], asQA [45], wikiAns [46] WMT22En-Ru [47]. Those were evaluated using exact match, Rouge-L [48] and sacreBleu [49], but due to the heavy cost of fine-tuning to evaluate, we only evaluated the model under two precision values $B = 2^{24}$ and $B = 2^{19}$. The results were very encouraging and we did not observe a significant change in model performance. The compression for these two configurations was 70% and 56% compared to 85% with only lossless compressors.

We ran a more detailed test on RoBERTa fine-tuned on the RottenTomatos dataset [37] and the results are shown in Figure 3. We see that model accuracy remains high (near 90%) until $B = 2^6$ and drops dramatically beyond that. The compression on the other hand improves at an almost linear rate culminating at around 20% before the drop-off. Note that there is even a slight rise in accuracy just before the drop, a phenomenon that we saw also in other tests. Note that running this method on the “clean” version of RoBETra yields no benefits before reaching $B = 2^{18}$ which aligns with our understanding.

These large cuts in precision with no change in accuracy have empirical and theoretical implications. Empirically, they suggest that much higher compression gains could be achieved with tuning. Theoretically, it may mean there are other training factors stronger than the computation errors above. Possibly, those are not calculation errors but traits of the network such as noise in gradient updates [50].

This technique is similar in nature to quantization techniques but is more tunable than quantization with the flexible precision factor, whereas quantization is limited to sizes that can naturally run the actual models like 16 or 8 bits. We elaborate on previous works suggesting low sensitivity to precision in §VI.

IV. EVALUATION

A. Integration in PyTorch

In order to make our compression applicable and easy to use we integrated these methods into PyTorch functions. `pytorch.save()` and `pytorch.load()`. These functions are used during uploading and downloading models from Hugging Face.

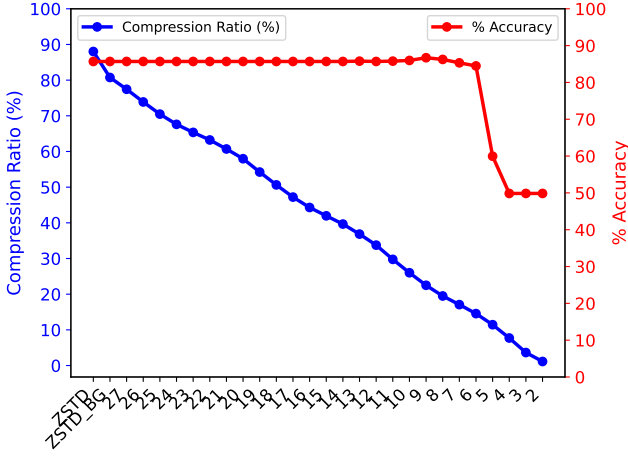


Fig. 3. Fine tuned RoBERTa compression and accuracy as a function of the precision factor parameter b (i.e., for $b = 27$ the factor is $B = 2^{27}$). The first two values are lossless compression without and with byte grouping.

This integration is the base for our timing evaluation below. Our intention is to contribute this code upstream into Pytorch in order to make compression a standard easy to use component for the community.

We implemented two approaches to compression, one compressing the entire model as a bin file and the other compressing each layer separately. The latter approach turned out faster for downloads (and slightly slower for uploads), and we use it in the model hub use-case that we evaluate next, which is downloads dominant.

B. Setup

In Sections III-B and III-D we present the compressibility traits of various models. In this section, we focus on time aspects and end-2-end timing of our first use-case - that of model hubs. We measured the time it takes to upload and download from Hugging Face to a virtual machine that runs on one of the cloud providers and is Located in the Milan region. We also measured upload and download performance on a home laptop with a 500Mbps network.

Unlike storage benefits, communication speeds depend heavily on the medium. We first characterized the general behavior of the communication with the Hugging Face hub. The upload bandwidth observed in the cloud remained mostly constant (at around 20 MBps). On the downloads, we observed 2 types of data transfer speeds.

- **First Download** - The speed in the first download showed large variance was between 20-40 MBps on the cloud VM. The home machine got approximately 10MBps.
- **Cached Download** - From the second read on the data is likely downloaded from a cloud cache and exhibits speed of 120-130 MBps in the cloud and approximately 40MBps at the home location.

We measured timing with lossless compression on 3 models, one from each of the model groups presented in Section III-B. Specifically, we used wav2vec, XLM-RoBERTa and Openllama. We used ZSTD in default setting (level 3) and byte

grouping. We also measured the timing of lossy compression on wav2vec.

C. Results

The end-2-end timing behavior is dictated by the time to compress/decompress the model and the time to upload/download it respectively. There is additional work done that is mostly constant and does not change with compression. For the overall time to be better than vanilla torch.save and torch.load, the benefits of uploading/downloading less data need to overcome the overhead of the actual compression/decompression. Figure 4 shows the timing of upload and download of three models. Each test was run 10 times for the cached reads and 5 times for the 1st timers. The variance was almost entirely due to the network time and this standard deviation is depicted in the graph. The actual compression and decompression time had very little variance. For example, in the xlm-RoBERTa the average time for load part (which includes the decompression) was 3.92 seconds with a standard deviation of 0.017.

As expected, highly compressible models show significant time improvements whereas the less compressible model category struggles to maintain the same non-compressed timing (but for the most part manages to do so). Naturally, the time saving is more significant when the network is slower. Therefore the cached reads in the cloud hardly save time even for the clean model and add a small overhead for the less compressible model. The upload time shows significant time improvement since the bandwidth for uploads is low. On the other hand, the upload savings are lower than download with similar bandwidth reflecting the fact that compression is slower than decompression.

For the first group of models that are hardly compressed, it is worthwhile to also test the tunable lossy compression. Figure 5 shows a breakdown of the download time of the wav2vec model on a 30MBps network and includes lossy compression with precision parameter 2^{23} . As expected for this model category the time savings with lossless is marginal, yet with the lossy compression it manages to reduce download time by almost 20% and maintains a slight edge over vanilla Pytorch even with the cloud cache network of 120MBps.

The breakdown of upload time is presented in Figure 6. The lossless technique was just 1% faster than vanilla PyTorch while the lossy version managed to save 16% of the upload time. Note that the tested implementation of tunable lossy did not include the sign bit optimization, as its time overhead did not justify the improved compression ratio. The lossy compression ended up with a 72% compression ratio (versus 85% with the lossless compression). Interestingly, the compression time of lossy compression was slightly faster than in the lossless case. This is likely due to the improved compression ratio which typically translates to faster compression speeds.

V. BEYOND FULL MODEL COMPRESSION

A. Delta compression.

When models have high similarity, one strategy to optimize storage and network transfer is to save a base model and for

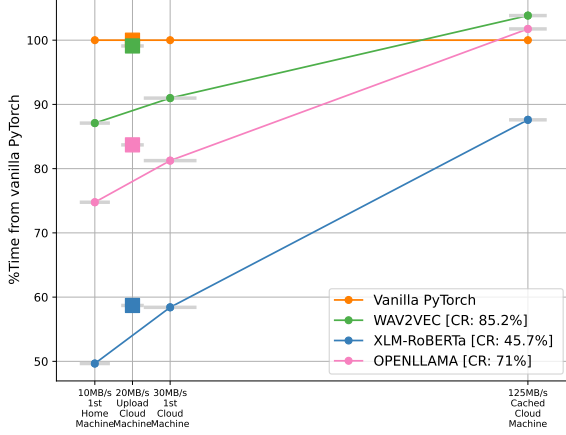


Fig. 4. Download and upload times of 3 models using full model compression vs. the non-compressed version.

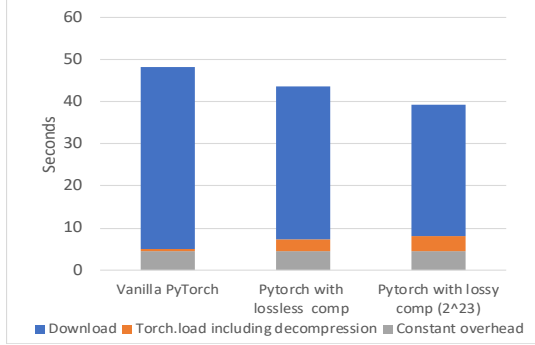


Fig. 5. Breakdown of download time for the wav2vec with a 30MBps network.

the rest of the models only store the differences from this base model [2]. We refer to compressing those differences as *delta compression*. To reconstruct a model, one only needs to apply the delta to the base model. A straightforward approach to delta compression is to compute the difference between the two models (e.g. using XOR or subtraction) and compress this delta using a standard compressor.

A natural use case in which delta compression proves very useful is checkpointing. In checkpointing, we repeatedly store models that have limited change between them. As mentioned in discussing tunable lossy compression (§III-C), fine tuning often changes models by small quantities. Hence the delta between the results of consecutive training epochs is highly compressible. This is true for lossless compression (especially with byte grouping) as well as tunable lossy compression. Figure 8 shows this for consecutive fine tuned epochs of the RoBERTa model. We see that lossless compression is as low as 55% (with byte grouping), down from nearly 83% of this model standalone. Figure 7 shows the compressibility of the 10th epoch vs. the base RoBERTa which is useful as it avoids maintaining long chains of deltas. In this case, the delta is less compressible but still achieves a 65% compression ratio. Using tunable lossy compression proves to be very beneficial also in delta compression. For example, taking $B = 2^{23}$ achieves a compression ratio of 37% for consecutive models and 49%

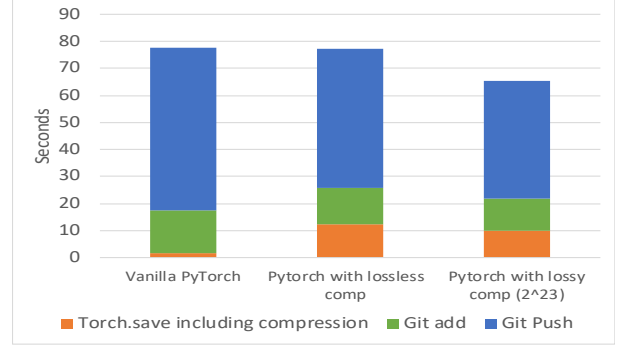


Fig. 6. Breakdown of upload time for the wav2vec model with a 20MBps network.

vs. the base model without affecting model accuracy. It is worth noting that an aggressive choice of the precision factor can achieve below 10% (over 90% savings!) without harming accuracy and in fact even achieving a slight improvement to the accuracy.

Another use-case is when a hub or user stores multiple models with high similarity (regardless of checkpointing). One source for such occurrence is when multiple models are trained or fine tuned from the same base model. For example, we found in Hugging Face 3 variations of RoBERTa trained on tweets and fine tuned for different purposes (For the exact model names, see Appendix C) - detecting irony, detecting offensive language and detecting abuse. As standalone models, their compression ratio (lossless with byte grouping) is 85.7% on average. However, when compressing the delta of each of the pairs achieves a ratio of 56% on average.

Another example is a set of models that are fine tuned on the twitter data set once every 3 months. The results, shown in Figure 9, show that delta compression is most beneficial in consecutive versions and its effectiveness slowly deteriorates over time.

B. Compressing Gradients and Optimizers

So far, we have mostly focused on compressing the actual models, but in various cases, such as distributed training or checkpointing, derivatives of the model also take up resources. Specifically, Gradients and Optimizers usually occupy a substantial amount of the communication, often of equal in size to the models [51]. We find that these components can also benefit from compression.

We investigate a BF16 version of RoBERTa. Figure 10 shows the break down of compressibility of the gradients broken down according to different layers. This is contrasted with the breakdown of the actual model in Figure 11. Not only that we find that most of the weights in the gradient compress similarly to regular weights. We find that token embeddings are extremely compressible, despite not showing a different behavior in regular models.

The optimizers show similar compressibility traits to the gradients. Namely, the embedding layer is extremely compressible and the general layers compress to around 67%, slightly better than these layers in the model itself as shown in Figure 12.

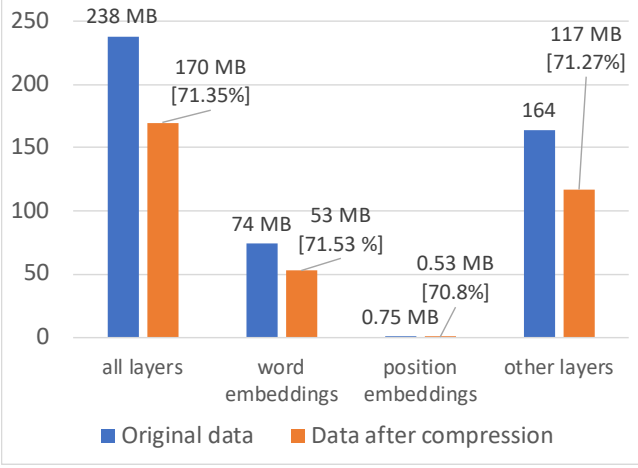


Fig. 11. Compressibility of layers in the **model**

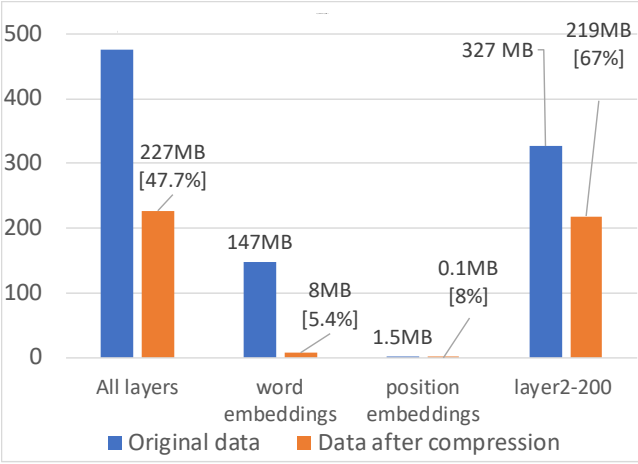


Fig. 12. Compressibility of layers in the **Optimizer**

reducing their size [52]. Under such conditions, a method is allowed to reduce the accuracy, and is judged on its tradeoff between size and performance. This differs from lossless compression which is supposed to return the model to its original state after decompression.

There are four main methods to reduce model size in that manner [52]. Pruning [53]–[55] (sometimes referred to as sparsification; [8]) where parts of the model are removed, dedicated training or network architecture [56], distillation [7] or otherwise training a smaller model from a better model [57] and quantization [9]. There are also methods combining several of those [58], including the only work we have found to propose compression, which it applies after two other model-compression steps [59].

B. Quantization

Out of the model-compression techniques, quantization is the most similar to the tunably lossy compression we discuss in this paper (§III-C). Quantization [60] is a method that bins weight values to a more coarse granularity. Since the model would be used as is, quantization is limited in the granularity to which it can compress models. For example, it cannot reduce to 23 bits (typically 16 is the first viable choice). Moreover,

quantization is not optimized for the smallest representation of the model and in fact quantized models can potentially be further compressed. We examine off-the-shelf quantized models (For the exact model names, See Appendix B) that have been quantized with GTPQ [61] and AWQ [62] and compress them using lossless compression. We see in Table III that they are still compressible, with a compression ratio between 85-91%, where byte grouping contributes to the compression 1-2%.

In a sense, quantization is complementary to compression. Quantization is able to improve inference speed and to reduce model size drastically, at costs to inference accuracy or the ability to further train the model. In contrast, compression cannot speed inference but can compress models further, or compress a model without affecting its behaviour at all. Similar to quantization, our Tunable lossy compression drops some of the information, but only ones that don't reduce accuracy in a measurable way. Unlike quantization, it changes the representation which is expected to further reduce the size as seen by the results above.

C. Other Related Work

Interestingly, two recent works also found improved results by reducing some of the information in the network (see §III-B). They saw it during pruning [63] or pruning and extreme quantization [64]. We observe this same phenomena in the tunable lossy compression. Future work may find a theory to connect those findings.

Similar to delta compression some works analyze dimensionality [65], [66] or save the deltas for actions such as compositionally [67] and merging multiple deltas [68]–[70]. Few works also apply on such deltas the above methods, like pruning [71], trained sparsity [72] or quantization [73] or discuss deltas.

Another line of work worth mentioning is computation graph optimization [74], [75]. Such works reduce the computation graph and perform optimization there. It is mostly noteworthy to contrast it to our work, such work compresses the size of the computation graph, which speeds computation, but does not change the model weights, and it is hence orthogonal to our work. To validate that, we compressed pyTorch [75] models before and after compilation, finding compression works similarly well.

Last, related to the optimization process (see §V-B), a few recent works offer to reduce the information passed in gradient updates hence making them faster, faster overcoming the reduce in information per example by seeing more examples per second [76], [77].

VII. CONCLUSION

We are in an era where models and system requirements grow larger, overparametrization seems to be beneficial for better learning. As our compression findings hint, this overparametrization is not fully used for inference or for the weights themselves and there is redundancy. Hence the wide attention and progress made to reducing model sizes is not without merit. That being said, the reality is that commonly

used models are not kept or run in reduced form and there is great inefficiency in the way models are stored and communicated today. Some of this inefficiency can be mitigated using the compression techniques outlined in this paper.

Given the reduction in network bandwidth, storage and time, we think that lossless compression should be the default in communication with model hubs such as Hugging Face. Moreover, we believe that communication compression has multiple other use-cases in the realm of training, versioning and serving models.

REFERENCES

- [1] S. Biderman, H. Schoelkopf, Q. G. Anthony, H. Bradley, K. O’Brien, E. Hallahan, M. A. Khan, S. Purohit, U. S. Prashanth, E. Raff, A. Skowron, L. Sutawika, and O. van der Wal, “Pythia: A suite for analyzing large language models across training and scaling,” *ArXiv*, vol. abs/2304.01373, 2023.
- [2] N. Kandpal, B. Lester, M. Muqeeth, A. Mascarenhas, M. Evans, V. Baskaran, T. Huang, H. Liu, and C. Raffel, “Git-theta: A git extension for collaborative development of machine learning models,” *arXiv preprint arXiv:2306.04529*, 2023.
- [3] S. Don-Yehiya, E. Venezian, C. Raffel, N. Slonim, and L. Choshen, “ColD fusion: Collaborative descent for distributed multitask finetuning,” in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (A. Rogers, J. Boyd-Graber, and N. Okazaki, eds.), (Toronto, Canada), pp. 788–806, Association for Computational Linguistics, July 2023.
- [4] C. Zhang, Y. Xie, H. Bai, B. Yu, W. Li, and Y. Gao, “A survey on federated learning,” *Knowledge-Based Systems*, vol. 216, p. 106775, 2021.
- [5] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew, “Huggingface’s transformers: State-of-the-art natural language processing,” *ArXiv*, vol. abs/1910.03771, 2019.
- [6] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, et al., “Mistral 7b,” *arXiv preprint arXiv:2310.06825*, 2023.
- [7] J. Gou, B. Yu, S. J. Maybank, and D. Tao, “Knowledge distillation: A survey,” *International Journal of Computer Vision*, vol. 129, pp. 1789–1819, 2021.
- [8] X. Ma, M. Qin, F. Sun, Z. Hou, K. Yuan, Y. Xu, Y. Wang, Y.-K. Chen, R. Jin, and Y. Xie, “Effective model sparsification by scheduled grow-and-prune methods,” *arXiv preprint arXiv:2106.09857*, 2021.
- [9] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, “A survey of quantization methods for efficient neural network inference,” 2021.
- [10] P. Deutsch and J.-L. Gailly, “Zlib compressed data format specification version 3.3,” tech. rep., 1996.
- [11] Y. Collet and M. Kuchera, “Zstandard compression and the application/zstd media type,” tech. rep., 2018.
- [12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” in *North American Chapter of the Association for Computational Linguistics*, 2019.
- [13] W. Fedus, B. Zoph, and N. M. Shazeer, “Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity,” *J. Mach. Learn. Res.*, vol. 23, pp. 120:1–120:39, 2021.
- [14] J. Yu Koh, “Model zoo (hub),” 2018.
- [15] Pytorch, “Pytorch hub,” 2019.
- [16] Google, “Tensorflow hub,” 2018.
- [17] J. Pfeiffer, A. Rücklé, C. Poth, A. Kamath, I. Vulić, S. Ruder, K. Cho, and I. Gurevych, “AdapterHub: A framework for adapting transformers,” in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (Q. Liu and D. Schlangen, eds.), (Online), pp. 46–54, Association for Computational Linguistics, Oct. 2020.
- [18] “Ibm watsonx.data,” <https://www.ibm.com/products/watsonx-data>.
- [19] “Qualcomm® ai hub,” <https://aihub.qualcomm.com/>.
- [20] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer, et al., “Pytorch fsdp: experiences on scaling fully sharded data parallel,” *arXiv preprint arXiv:2304.11277*, 2023.
- [21] M. Li, S. Gururangan, T. Dettmers, M. Lewis, T. Althoff, N. A. Smith, and L. Zettlemoyer, “Branch-train-merge: Embarrassingly parallel training of expert language models,” *arXiv preprint arXiv:2208.03306*, 2022.
- [22] V. Lialin, S. Muckatira, N. Shivagunde, and A. Rumshisky, “Relora: High-rank training through low-rank updates,” in *Workshop on Advancing Neural Network Training: Computational Efficiency, Scalability, and Resource Optimization (WANT@ NeurIPS 2023)*, 2023.
- [23] M. Diskin, A. Bukhtiyarov, M. Ryabinin, L. Saulnier, q. lhoest, A. Sinitsin, D. Popov, D. V. Pyrkín, M. Kashirin, A. Borzunov, A. Villanova del Moral, D. Mazur, I. Kobelev, Y. Jernite, T. Wolf, and G. Pekhimenko, “Distributed deep learning in open collaborations,” in *Advances in Neural Information Processing Systems* (M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, eds.), vol. 34, pp. 7879–7897, Curran Associates, Inc., 2021.
- [24] R. Turner, D. Eriksson, M. J. McCourt, J. Kili, E. Laaksonen, Z. Xu, and I. M. Guyon, “Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020,” in *Neural Information Processing Systems*, 2021.
- [25] J. Dodge, G. Ilharco, R. Schwartz, A. Farhadi, H. Hajishirzi, and N. A. Smith, “Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping,” *ArXiv*, vol. abs/2002.06305, 2020.
- [26] M. Junczys-Dowmunt, R. Grundkiewicz, T. Dwojak, H. T. Hoang, K. Heafield, T. Neckermann, F. Seide, U. Germann, A. F. Aji, N. Bogoychev, A. F. T. Martins, and A. Birch, “Marian: Fast neural machine translation in c++,” in *Annual Meeting of the Association for Computational Linguistics*, 2018.
- [27] M. Sandler, A. Zhmoginov, M. Vladymyrov, and N. Miller, “Training trajectories, mini-batch losses and the curious role of the learning rate,” *ArXiv*, vol. abs/2301.02312, 2023.
- [28] Z. Liu, A. Qiao, W. Neiswanger, H. Wang, B. Tan, T. Tao, J. Li, Y. Wang, S. Sun, O. Pangarkar, R. Fan, Y. Gu, V. Miller, Y. Zhuang, G. He, H. Li, F. Koto, L. Tang, N. Ranjan, Z. Shen, X. Ren, R. Iriondo, C. Mu, Z. Hu, M. Schulze, P. Nakov, T. Baldwin, and E. P. Xing, “Llm360: Towards fully transparent open-source llms,” *arXiv*, 2023.
- [29] S. Wang and P. Kanwar, “Bfloat16: The secret to high performance on cloud tpus,” *Google Cloud Blog*, vol. 4, 2019.
- [30] Squash, “Squash compression benchmark,” 2016.
- [31] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.
- [32] D. A. Huffman, “A method for the construction of minimum-redundancy codes,” *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.
- [33] J. J. Rissanen, “Generalized kraft inequality and arithmetic coding,” *IBM Journal of Research and Development*, vol. 20, no. 3, pp. 198–203, 1976.
- [34] Y. Collet, “Lz4 - extremely fast compression,” 2024.
- [35] M. Adler and J.-L. Gailly, “Zlib,” 2024.
- [36] Y. Collet, “Zstandard,” 2024.
- [37] B. Pang and L. Lee, “Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales,” in *Proceedings of the ACL*, 2005.
- [38] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “Exploring the limits of transfer learning with a unified text-to-text transformer,” *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.
- [39] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, Curran Associates, Inc., 2019.
- [40] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattemberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015. Software available from tensorflow.org.
- [41] F. Chollet et al., “Keras,” <https://keras.io>, 2015.
- [42] R. Nallapati, B. Zhou, C. Gulcehre, B. Xiang, et al., “Abstractive text summarization using sequence-to-sequence rnns and beyond,” *arXiv preprint arXiv:1602.06023*, 2016.
- [43] S. Narayan, S. B. Cohen, and M. Lapata, “Don’t give me the details, just the summary! topic-aware convolutional neural networks for extreme

APPENDIX

A. Models name Downloaded from Hugging Face

The main models used are: For FP32 Models: Bert [12], GPT2 [78], wav2Vec [79], RoBERTa [80] and fine tuned RoBERTa on the Rotten Tomato dataset [37], XLM-Roberta, CLIP [81], and T5-base [38].

For BF16 Models: Falcon-7B [82], Bloom [83] and OpenBuddy/OPENLLAMA (3B-BF16, [84]) and Mistral [6]. For FP16 Models: llama2-13B [85] Tulu-7B [86] and argilla/CapybaraHermes-2.5-Mistral-7B [6] and Stable-Video-Diffusion.

We provide a full list of the exact model names used as they appear in Hugging Face hub for complete reproducibility.

- jonatasgrosman/wav2vec2-large-xlsr-53-english
- google-bert/bert-base-uncased
- openai-community/gpt2
- runwayml/stable-diffusion-v1-5
- becausecurious/stable-video-diffusion-img2vid-fp16
- argilla/CapybaraHermes-2.5-Mistral-7B
- FacebookAI/roberta-base
- FacebookAI/xlm-roberta-base
- openai/clip-vit-large-patch14
- google-t5/t5-base
- TheBloke/Llama-2-13B-Chat-fp16
- TheBloke/tulu-7B-fp16
- tiuuue/falcon-7b
- bigscience/bloom
- OpenBuddy/openbuddy-openllama-3b-v10-bf16
- mistralai/Mistral-7B-v0.1

B. Quantized Models

We provide the names of the Quantized Models used as they appear in Hugging Face hub for complete reproducibility. We discuss compressing those in section VI.

- TheBloke/CapybaraHermes-2.5-Mistral-7B-GPTQ
- TheBloke/CapybaraHermes-2.5-Mistral-7B-GPTQ
- TheBloke/CapybaraHermes-2.5-Mistral-7B-GPTQ
- TheBloke/CapybaraHermes-2.5-Mistral-7B-GPTQ
- TheBloke/CapybaraHermes-2.5-Mistral-7B-GPTQ
- TheBloke/CapybaraHermes-2.5-Mistral-7B-AWQ

C. RoBERTa-base model trained on tweets

We provide the names of the models finetuned on twitter dataset, as they appear on Hugging Face hub for complete reproducibility. We discuss compressing those in §V-A.

- cardiffnlp/twitter-roberta-base-irony
- cardiffnlp/twitter-roberta-base-offensive
- cardiffnlp/twitter-roberta-base-hate
- cardiffnlp/twitter-roberta-base-mar2020
- cardiffnlp/twitter-roberta-base-jun2020
- cardiffnlp/twitter-roberta-base-sep2020
- cardiffnlp/twitter-roberta-base-dec2020
- cardiffnlp/twitter-roberta-base-mar2021
- cardiffnlp/twitter-roberta-base-jun2021
- cardiffnlp/twitter-roberta-base-sep2021
- cardiffnlp/twitter-roberta-base-dec2021