
Factorize and conquer: Genghis-GAN

Jimmy Leroux^{*1} Nicolas Laliberte^{*1} Tobi Carvalho^{*1}

Abstract

Generative Adversarial Network (GAN) achieve state-of-the-art performance within the class of generative models. In order to capture complex distribution in high dimensional setting, the learning procedure of such models require a large number of parameters and thus is demanding memory wise when it comes to inference. This difficulty arises when deploying effective GAN on platforms with restricted computational resources such as smartphones. In this article, we tackle this problem by presenting a GAN where layers are compressed in the Tensor Train format (TT-format). The objective is to drastically reduce the number of parameters while maintaining a high generative performance. Experiments show that our fully tensorized model, the Genghis-GAN, can achieve a 40x parameter compression rate when compared to a baseline DCGAN, while maintaining comparable generative power. However, even if the number of parameters has been greatly reduced, we found out that it didn't take less memory because of the massive input/output it creates. While tensorizing convolutional layer can, in theory, lighten a model, we found out that in practice it's rarely the case.

1. Introduction

Generative Adversarial Networks (GAN) are a class of generative models which achieves state-of-the-art performance. These models are used to learn very high dimensional distribution, usually with algorithms that are very computationally expensive. The high computational cost prevents us from implementing those models in low power devices

^{*}Equal contribution ¹Mila, University of Montreal. Correspondence to: Jimmy Leroux <jim.leroux1@gmail.com>, Nicolas Laliberte <n.laliberte01@gmail.com>, Tobi Carvalho <jim.leroux1@gmail.com>.

like smartphones. This is one of the many motivations to find new techniques for compressing these models. This work proposes the use of tensor-train decomposition in order to compress a DCGAN (deep-convolutional GAN). In (Garipov et al., 2016) they used this technique to achieve a compression rate of 80× on a CNN with only 1.1% accuracy drop on CIFAR-10. More precisely, the proposed approach in (Garipov et al., 2016) consists of compressing both convolutional and fully-connected layers. Following this work, we will also use tensor-train decomposition on both convolution and fully-connected layers. For a reminder of GANs, see (Goodfellow et al., 2014).

1.1. Related work

Previous work has been done on the subject. In (Cao et al., 2018), they propose a GAN composed of an MLP where the layers are compressed using the Tucker Decomposition. The proposed architecture here consists of convolutional layers in the TT-format. Although applying the TT format to layer demonstrates a large compression rate, it remains a hard problem to find an appropriate number of cores and TT-ranks. In this project, we will barely address this problem. The number of cores and TT-ranks were arbitrarily fixed.

2. Framework

2.1. Tensor Train

A convenient way to represent tensor operations is through tensor networks. Describing operation with high order tensors can be cumbersome with a large number of indices. Tensor network consists of a non oriented graph with weight to edges. Nodes represent tensor and the number of edges associated with this node represent the order of the tensor. Weights to edges gives the dimension of the tensor. In the figure 1, the network represents a tensor of order n where $T \in \mathbb{R}^{d_1 \times \dots \times d_n}$.

Using this notation, we present the tensor train decomposition.

Given a tensor $T \in \mathbb{R}^{d_1 \times d_2 \times \dots \times d_n}$ and a collection of cores $\{G^s\}_{s=1}^n$ where $G^1 \in \mathbb{R}^{d_1 \times R_1}$, $G^s \in \mathbb{R}^{R_{s-1} \times d_s \times R_s}$ and $G^n \in \mathbb{R}^{R_{n-1} \times d_n}$. We can decompose T using the tensor-

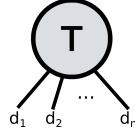


Figure 1. Diagrammatic way of representing tensors.

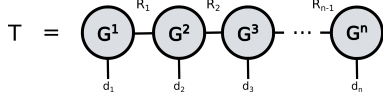


Figure 2. Tensor in the TT-format.

train (TT) decomposition in figure 2, where the components are given by

$$\begin{aligned} T_{i_1, i_2, \dots, i_n} &= \sum_{r_1=1}^{R_1} \sum_{r_2=1}^{R_2} \dots \sum_{r_n=1}^{R_n} G_{i_1, r_1}^1 G_{r_1, i_2, r_2}^2 \dots G_{r_{n-1}, i_n}^n \\ &= G^{(1)}[i_1] \cdot G^{(2)}[i_2] \cdot \dots \cdot G^{(n)}[i_n] \end{aligned}$$

This form is also called Matrix product state. Many questions arise from this decomposition, such as how we choose the ranks R_i . We won't dive into this, for more detailed information see (Oseledets, 2011).

2.2. Generative adversarial model

Generative adversarial network consists of 2 components: discriminator and a generator. Both are in general neural nets. In this case, both are deep convolutional neural network. The discriminator's goal is to distinguish real data from fake data. More specifically, if we consider P_{real} and P_{gen} as the distribution of the real and the generated data respectively, the task of the discriminator is to determine whether the input belongs to P_{real} or P_{gen} . On the other hand, the generator tries to reproduce samples that are indistinguishable from the real data to deceive the discriminator. There are several possible training procedures to produce this adversarial process. Our procedure is described in section 4.1. See (Goodfellow et al., 2014) for more details about GAN.

3. Methods

3.1. Compression of linear layers

We now briefly describe the *TT-layer* following (Novikov et al., 2015). In short, the TT-layer is a fully-connected layer with the weight stored in the TT-format. Usual FC layers apply a linear transformation on an input x :

$$y = Wx + b$$

where W denotes the weight matrix and b the bias. Storing W in a TT-format allows the use of a large number of hidden units while maintaining a moderate number of parameters. Using the same step as in 2.1 the linear transformation of a TT-layer can be expressed as

$$\begin{aligned} \mathcal{Y}(i_1, \dots, i_d) &= \sum_{j_1, \dots, j_d} \mathbf{G}_1[i_1, j_1] \dots \mathbf{G}_d[i_d, j_d] \mathcal{X}(j_1, \dots, j_d) + \mathcal{B}(i_1, \dots, i_d) \end{aligned}$$

where \mathcal{Y} , \mathcal{X} and \mathcal{B} are the tensorized output, input and bias respectively.

3.2. Compression of convolution layers

To do so (Novikov et al., 2015) reshaped the convolutional tensor into higher dimensional tensors (then taking full advantage of the TT-decomposition technique) instead of simply directly compressing them. Indeed the naive approach, as they call it, would result in a rather small gain in compression since the tensors are only of order 3 or 4. To do so they start by transforming the tensor convolution in a matrix to matrix multiplication, which they then transform again to a high dimensional tensor convolution.

Starting with the 3 dimensional input tensor $\mathcal{X} \in \mathbb{R}^{W \times H \times C}$, the 3 dimensional output tensor $\mathcal{Y} \in \mathbb{R}^{W-l+1 \times H-l+1 \times S}$ and the 4 dimensional kernel tensor $\mathcal{K} \in \mathbb{R}^{l \times l \times C \times S}$, where H is the height of the image input, W the width, K the size of the kernels, C the number of channels (ex:RGB) and S the output dimension, they are all transformed into matrices.

$$\begin{aligned} \mathcal{Y}(x, y, s) &= Y(x + (W - l + 1)(y - 1), s) \\ \mathcal{X}(x + i - 1, y + j - 1, c) &= X(x + (W - l + 1)(y - 1), i + l(j - 1) + l^2(c - 1)) \\ \mathcal{K}(i, j, c, s) &= K(i + l(j - 1) + l^2(c - 1), s). \end{aligned}$$

We then have $\mathbf{Y} = \mathbf{XK}$. We can then reshape all these components again into these high dimensional components

$$\begin{aligned} Y(x + (W - l + 1)(y - 1), s) &= \tilde{\mathcal{Y}}(x, y, s_1, \dots, s_d) \\ X(x + (W - l + 1)(y - 1), i + l(j - 1) + l^2(c - 1)) &= \tilde{\mathcal{X}}((x, y, c_1, \dots, c_d)) \\ K(i + l(j - 1) + l^2(c - 1), s) &= \tilde{\mathcal{K}}((x + l(y - 1), 1), (c_1, s_1), \dots, (c_d, s_d)) \\ &= \tilde{\mathbf{G}}_0[x + l(y - 1), 1] \mathbf{G}_1[c_1, s_1] \dots \mathbf{G}_d[c_d, s_d]. \end{aligned}$$

where $c = c_1 + \sum_{i=2}^d (c_i - 1) \prod_{j=1}^{i-1} C_j$ and $s = s_1 + \sum_{i=2}^d (s_i - 1) \prod_{j=1}^{i-1} S_j$, under the restriction that the dimensions factorized as $C = \prod_{i=1}^d C_i$ and $S = \prod_{i=1}^d S_i$ (these are not real restrictions since we can always add some dummy channels filled with zeros to increase the values

of C and S). The $\mathbf{G}_k[c_k, s_k]$ are the core in the TT-format (note that in this notation there is also 1 or 2 additional dimensional which corresponds to the connection between the core tensors).

The tensorization of each element is defined as follows: consider a vector $x \in \mathbb{R}^N$ where $N = \prod_{k=1}^d n_k$. You can at least always choose the prime decomposition of N as the n_k . Then, we can establish a bijection between this vector x and a d -dimensional tensor \mathcal{X} where each fiber represents a portion of the vector x . Let $l \in \{1, \dots, N\}$ and the bijection $\mu(l) = \{\mu_1(l), \dots, \mu_d(l)\}$ where $\mu_k(l) \in \{1, \dots, n_k\}$, then the corresponding tensor \mathcal{X} is defined by the vector elements $\mathcal{X}(\mu(l)) = x_l$ (this can be generalized to matrix by using another mapping $v(p)$ for the second coordinate).

3.3. Inference

When using the kernel in the tensor-train format (TT-kernel), there is two way to proceed for the convolution, see figure 3. The first method consists of making the convolution with the first core of the TT-kernel (the first core being the one containing the x-y dimension used to make the convolution) and then to contract successively with the following cores. The second one starts by fully re-contracting the TT-kernel to then using it to make the convolution (the later being equivalent to using a regular kernel without any TT decomposition). Both techniques yield the same outputs but have different algorithmic complexity. While only the first method can yield an improvement over the normal method, it's not always better in term of memory and calculation time even though it always reduces the number of parameters needed to express the kernel.

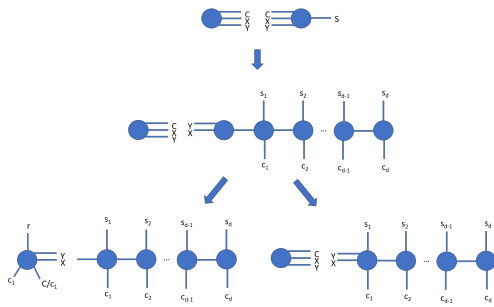


Figure 3. Feed forward through TT-convolutional layer

Indeed, to evaluate the total memory taken while doing inference, one has to consider not only the space taken by the kernel, but also the maximum size of the input/output at each step of the computation. Even if in the first strategy the kernel is smaller, the output of the convolution using only the first core is often bigger than the kernel was. The

maximum size of the input/output between the operation is of size $\in \mathcal{O}(\max(C, S) * r * h * w)$ while the kernel was initially of size $\in \mathcal{O}(C * S * k_h * k_w)$, where C is the number of input maps, S the number of output maps, r the ranks between the cores, h the height of the input, w the width of the input, k_h the height of the kernels and k_w the width of the kernels. As an approximating criteria to consider using this decomposition we can use $\min(C, S) * k_h * k_w \geq r * h * w$ (the exact criteria depend on several other parameters and can be cumbersome to express). Since h and w are often significantly bigger than k_h and k_w it's rarely the case.

Even when the first strategy is worth using for inference, the second one can be preferred for training since the batch size add up to the size of the input/output.

On a side note, the reason why the dimension of h and w are not also expended to higher dimensions, is because that doing so, the connectivity between the pixels would probably be lost (which is the primary reason why convolutions work so well with images).

3.4. Upper bond on the first rank

We think that the ranks between the cores of the TT-kernel should be treated as hyperparameters and that the optimal values should be determined by testing and evaluating the trade-off between the rate of compression and the quality of the images produced. That being said, we think an argument can be made to fix an upper-bound on the first rank of the TT-kernel.

Since only the first core of the TT-kernel is used to make the convolution, we think the first rank can be bounded by the product of the kernel size. The first rank could be seen as the number of convolutional matrices from which each map is then constructed. Using the fact that the product of the kernel size corresponds to the minimum number of kernels from which any kernel can be constructed (since a convolution is a linear operation), we think that using a bigger rank would only lead to redundancy and shouldn't be needed.

4. Experiments

We performed experiments on tensorized/partially tensorized generative models and showed the effectiveness of pretraining the model's weights using an autoencoder. The main experiments are done on the MNIST dataset, an open source dataset of 60000 hand written digits. We also report sampled images of our TT-DCGAN model as well as the benchmark models. All the codes are available at <https://github.com/jimlroux/TT-DCGAN>.

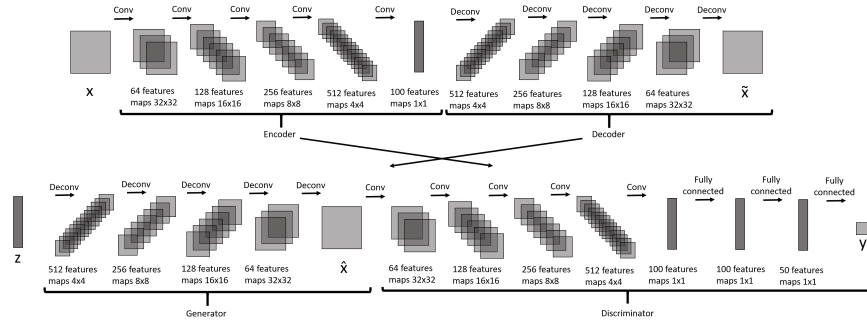


Figure 4. Architecture of the DCGAN and the autoencoder. We take the discriminant of the DCGAN without the fully connected layers to form the encoder and we take the generator as is to form the decoder.

4.1. Training procedure

For the training of our models, we kept the same training procedures across all models, so we could more easily identify the impact of tensorization on the training. At first, the data were resized to 64x64 using bilinear interpolation and renormalized to the range $[-1, 1]$. All the experiments used almost the same set of hyperparameters. The minibatch size was set to 64 with a learning rate of 0.0005 for the generator as well as for the discriminator. We kept the ratio of generator updates vs discriminator updates to 1 by minibatch. The optimization algorithm that was used was ADAM with betas of 0.5 and 0.999. The architecture was also fixed, except when tensorizing different layers. When training the autoencoder, every hyperparameters were the same except that the learning rate was set to 0.003.

4.2. Trained Models

Multiple different architectures were tried, ranging from non-tensorized DCGAN to fully tensorized DCGAN. For each of the trained model, we had the possibility to tune a lot of hyperparameters. We started by finding an hyperparameter setting that was generating decent sample images in the hardest conditions (the fully tensorized model). Next, we fixed these hyperparameters across all our models and tried various architectures to explore the effect of tensorizing different layers of the model. The different architectures are shown in the table 1.

4.3. GAN pretraining

Since the training of GANs and all its variations can be very cumbersome, we initialized the weight by making the discriminator and the generator work together instead of against each other. This way they could quickly learn weight that works well with the data set right away.

To do so, we formed an autoencoder with the generator and discriminator of our (TT-)DCGAN models. First, we

Table 1. Layers that are in the TT-format are listed. Both encoder and decoder have 5 convolutional layers. The 4th column indicate if the architecture has a fully-connected layer in the TT format (TFC).

ID	ENCODER	DECODER	TFC	#PARAMS
BASELINE	NONE	NONE	NO	1 346 373
FTT	ALL	ALL	YES	34 080
ATT	[2, 3, 4, 5]	[1, 2, 3]	NO	111 329
ATT-TTFC	[2, 3, 4, 5]	[1, 2, 3]	YES	98 672
TTENC	[2, 3, 4, 5]	NONE	NO	697 025
TTENC-TTFC	[2, 3, 4, 5]	NONE	YES	684 368
TTDEC	NONE	[1, 2, 3]	NO	760 677
TTDEC-TTFC	NONE	[1, 2, 3]	YES	748 020

removed the 3 FC layers of the discriminator and used it as the encoder. Next, we took the generator as it was and used it as the decoder, see figure 4. We then trained the autoencoder for 20 epochs for each model by minimizing the Mean-Square Error (MSE) of the reconstruction, see figure 5.

You can see that almost all models are able to reach a pretty low reconstruction error on the validation set, except the fully tensorized one which stays a bit higher. Despite the huge compression rate of the model, it is still able to learn. We will see in the sections below how this pretraining improves generative power of the tensorized models.

4.4. Genghis-GAN

In the section 4.3, we saw that our models were all able reach a relatively low reconstruction error, meaning that they had enough capacity to recreate input images. After the pretraining, we trained the GANs corresponding to each of the autoencoders. Generated samples at several steps in the training process are shown in the table 2. Looking at the first two rows, you can notice that the pretraining not

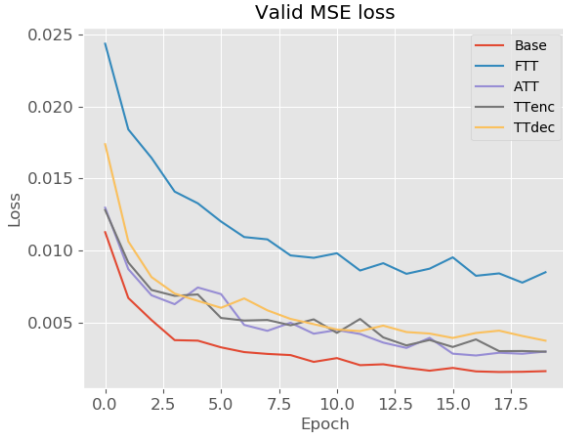


Figure 5. Mean-Square Error loss on the reconstruction of input images by the autoencoder for the Baseline model (Base), Full Tensor-Train (FTT), Almost All Tensor-Train (ATT), only the encoder in Tensor-Train (TTenc) and only the decoder in Tensor-Train (TTdec).













only helped training the GAN but is almost necessary. In fact, the unpretrained ATT seems to have mode collapsed, and even after 20 epochs, the generated samples are still bad. Conversely, when we pretrain the model (top row), we observe good quality samples even after only 5 epochs. We can compare the pretrained ATT with the pretrained Genghis-GAN (FTT). Even if the evaluation of the generated samples is hard to do, we can argue that the samples are similar in quality. Now comparing the two tensorized models with the baseline model (where we adjusted the number of parameters), we can see that the samples are a little bit better for the baseline model, the edges are smoother and are better defined. Still, we can then say that our model, the Genghis-GAN, even with a lot less parameters, can generate samples comparable to the conventional DCGAN.

4.5. TT-decomposition of the input/output

As described in section 3.2, because the size of the kernels is often small compared to the size of the inputs, the number of maps in the inputs and the outputs must be big for the TT-decomposition of the kernel to be worthwhile. In order to increase the number of situations where the decomposition can be used effectively and to improve the technique used, we considered using the TT-decomposition on both, the inputs/outputs and the kernels.

As seen in figure 6 by decomposing both the input and the kernel with the TT-decomposition and by contracting only along the input map axis, the output is already in a TT format and could already be used for the next layer. Furthermore, using the decomposition on the input prevents the output of

Table 2. Table showing generated samples at the epoch 1,5 and 20 of the training process. The top row is the pretrained ATT and below is the not pretrained version. The third row is the pretrained FTT (Genghis-GAN) and the last row is the baseline model.

MODEL	EPOCH 1	EPOCH 5	EPOCH 20	#PARAMS
PRETRAINED ATT				190241
ATT				
PRETRAINED FTT (GENGHIS)				34080
BASELINE				

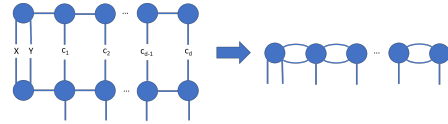


Figure 6. TT-decomposition of the input/output

the convolution to be too big since the output would only be $\in \mathcal{O}(r * r' * h * w + n * c_i * s_i * r * r')$ where n is the number of cores (c_i and s_i decrease with n). However, the size of the output would increase exponentially with the number of convolution layer, but this effect could either be controlled by limiting the number of layers or by using TT-rounding between the layers. Sadly, this technique doesn't work since it seems impossible to apply an activation function correctly. Since the cores are the same whatever the input, it doesn't make sense to use an activation function on them. Nothing would be gained from doing so since the results would only correspond to a different set of non-activated cores. The activation could then only be usefully applied on the result from the convolution with the first core. Doing so would greatly reduce the capacity of the model since activating a combination of maps can generate other maps that aren't part of the linear combination (because of the non-linearity of the activation function), while the same cannot be said about a linear combination of activated maps.

5. Conclusion and future works

In this paper, we proposed to decompose the convolutional layers and the linear layers into a tensor-train format to form a DCGAN. This way, we compressed the network by a factor of $40\times$, but found out that despite of this, the computation cost wasn't reduced because of the inputs/outputs produced by the TT-convolutional layers. We showed the importance of pretraining the GAN as an autoencoder, because otherwise the network doesn't learn at all. The fully tensorized DCGAN, the Genghis-GAN, achieved comparable performance to the baseline model. We also argued that the tensorization of the input/output tensor couldn't be used effectively since there is no way to apply an activation function while staying in this format.

In the future work, we would like to come up with a way to choose the number of cores in the decomposition as well as assigning their ranks. This would allow us to better optimize the training as well as the compression of the models. We would also like to use the TT-decomposition on different type of GAN since it was only ineffective for the convolutional layers.

References

- Cao, X., Zhao, X., and Zhao, Q. Tensorizing Generative Adversarial Nets. *2018 IEEE International Conference on Consumer Electronics - Asia, ICCE-Asia 2018*, 2018. doi: 10.1109/ICCE-ASIA.2018.8552122.
- Garipov, T., Podoprikin, D., Novikov, A., and Vetrov, D. Ultimate tensorization: compressing convolutional and FC layers alike. *arXiv e-prints*, art. arXiv:1611.03214, Nov 2016.
- Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative Adversarial Networks. *arXiv e-prints*, art. arXiv:1406.2661, Jun 2014.
- Novikov, A., Podoprikin, D., Osokin, A., and Vetrov, D. Tensorizing Neural Networks. *arXiv e-prints*, art. arXiv:1509.06569, Sep 2015.
- Oseledets, I. Tensor-train decomposition. *SIAM J. Scientific Computing*, 33:2295–2317, 01 2011. doi: 10.1137/090752286.