Model Compression for Uncertainty Estimation using Deep Ensembles

Decomposers Team

Gleb Bazhenov, Saydash Miftakhov, Lina Bashaeva

Vladislav Trifonov, Alexander Volkov

Problem Statement

What?

- Investigate how **compression methods** based on tensor decomposition for linear and convolutional layers influence the uncertainty estimation of neural networks in classification task.

Why?

- The **problem of uncertainty estimation** is critical for such domains as medical diagnostics and self-driving cars.
- The standard approach for estimating the calibration of models and their capability to detect the out-of-distribution (OOD) samples is **Deep Ensembles** [Lakshminarayanan et al., 2017], which requires to construct many independent models.
- However, in **memory-constrained applications** the number of parameters needs to be reduced without decrease in performance.

Problem Statement

Hypothesis

- We assume that the compressed models with proper choice of approximation rank **do not lose their performance** and still provide an opportunity for estimating the uncertainty in OOD cases.

Quality Measurement

- In order to measure the performance of compressed models in terms of uncertainty we track various properties of predictive distribution of Deep Ensembles, such as **negative log-likelihood** (NLL), **Brier score** (BS) and **entropy**.
- Generally, we compare Deep Ensembles with **Monte Carlo Dropout** [Gal & Ghahramani, 2016], providing the uncertainty metrics and the results in **classification accuracy** for both original and compressed models.
- All experiments are conducted using MNIST and notMNIST datasets.

Singular Value Decomposition

To compute low-rank approximation, we need to compute **singular value decomposition** (SVD).

Theorem. Any matrix $A \in \mathbb{C}^{n \times m}$ can be written as a product of three matrices:

$$A = U \Sigma V^*,$$

where

- U is an $n \times n$ unitary matrix,
- V is an $m \times m$ unitary matrix,
- Σ is a diagonal matrix with non-negative elements $\sigma_1 \geqslant \ldots \geqslant \sigma_k$ on the diagonal, where $k = \min(m, n)$.
- Moreover, if $\operatorname{rank}(A) = r$, then $\sigma_{r+1} = \ldots = \sigma_k = 0$.

What is a Tensor

Tensor = multidimensional array:

$$A = [A(i_1, \ldots, i_d)], i_k \in 1, \ldots, n_k$$

Terminology:

- *dimensionality* = d (number of indices).
- *size* = $n_1 \times \cdots \times n_d$ (number of nodes along each axis).

Case $d = 1$ is for vector, $d = 2$ is for matrix.

Curse of Dimensionality

Number of elements = n^d(exponential in d)

When $n = 2, d = 100$

$$2^{100} > 10^{30} (\approx 1018 \text{ PB of memory}).$$

Cannot work with tensors using standard methods.

Tensor Rank Decomposition [Hitchcock, 1927]

Recall the rank decomposition for matrices:

$$A(i_1, i_2) = \sum_{\alpha=1}^{r} U(i_1, \alpha) V(i_2, \alpha).$$

This can be generalized to tensors.

Tensor rank decomposition (**canonical decomposition**):

$$A(i_1, \ldots, i_d) = \sum_{\alpha=1}^{R} U_1(i_1, \alpha) \ldots U_d(i_d, \alpha).$$

The minimal possible R is called the (canonical) rank of the tensor A.

Pros

- No curse of dimensionality.

Cons

Unfolding Matrices: Definition

Every tensor A has $d-1$ **unfolding matrices**:

$$A_k := [A(i_1 \dots i_k; i_{k+1} \dots i_d)],$$

where

$$A(i_1 \dots i_k; i_{k+1} \dots i_d) := A(i_1, \dots, i_d).$$

Here $i_1 \dots i_k$ and $i_{k+1} \dots i_d$ are row and column (multi)indices; A_k are matrices of size $M_k \times N_k$ with

$$M_k = \prod_{s=1}^{k} n_s, N_k = \prod_{s=k+1}^{d} n_s.$$

This is just a reshape.

Unfolding Matrices: Example

Consider $A = [A(i,j,k)]$ given by its elements:

$$\begin{aligned} A(1,1,1) &= 111, \quad A(2,1,1) = 211, \\ A(1,2,1) &= 121, \quad A(2,2,1) = 221, \\ A(1,1,2) &= 112, \quad A(2,1,2) = 212, \\ A(1,2,2) &= 122, \quad A(2,2,2) = 222. \end{aligned}$$

Then

$$A_1 = [A(i;jk)] = \begin{bmatrix} 111 & 121 & 112 & 122 \\ 211 & 221 & 212 & 222 \end{bmatrix},$$

$$A_2 = [A(ij;k)] = \begin{bmatrix} 111 & 112 \\ 211 & 212 \\ 121 & 122 \\ 221 & 222 \end{bmatrix}.$$

Tensor Train Decomposition: Motivation

Main idea — **variable splitting**.

Consider a rank decomposition of an unfolding matrix:

$$A(i_1 i_2; i_3 i_4 i_5 i_6) = \sum_{\alpha_2} U(i_1 i_2; \alpha_2) V(i_3 i_4 i_5 i_6; \alpha_2).$$

On the left: 6-dimensional tensor; on the right: 3- and 5-dimensional. The dimension has reduced! Proceed recursively.

Tensor Train Decomposition [Oseledets, 2011]

- TT-format for a tensor A:

$$A(i_1, \ldots, i_d) = \sum_{\alpha_0, \ldots, \alpha_d} G_1(\alpha_0, i_1, \alpha_1) G_2(\alpha_1, i_2, \alpha_2) \ldots G_d(\alpha_d - 1, i_d, \alpha_d).$$

- This can be written compactly as a matrix product:

$$A(i_1, \ldots, i_d) = \underbrace{G_1[i_1]}_{1 \times r_1} \underbrace{G_2[i_2]}_{r_1 \times r_2} \ldots \underbrace{G_d[i_d]}_{r_{d-1} \times 1}$$

Finding a TT-representation of a Tensor

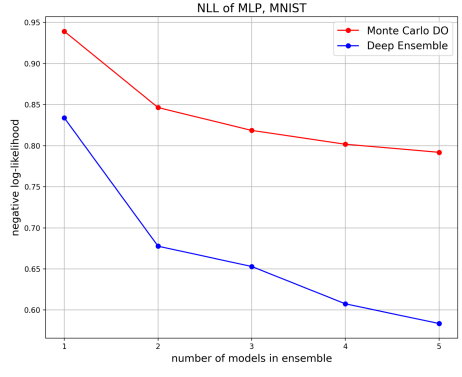
General ways for building a TT-decomposition of a tensor:

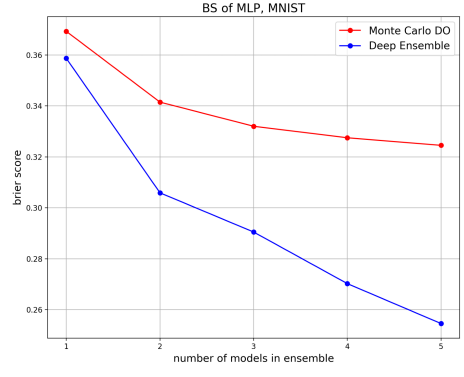
- Analytical formulas for the TT-cores.
- TT-SVD algorithm [Oseledets, 2011]:
    - Exact quasi-optimal method.
    - Suitable only for small tensors (which fit into memory).
- Interpolation algorithms: AMEn-cross [Dolgov & Savostyanov, 2013], DMRG [Khoromskij & Oseledets, 2010], TT-cross [Oseledets, 2010]
    - Approximate heuristically-based methods.
    - Can be applied for large tensors.
    - No strong guarantees but work well in practice.
- Operations between other tensors in the TT-format: addition, element-wise product etc.

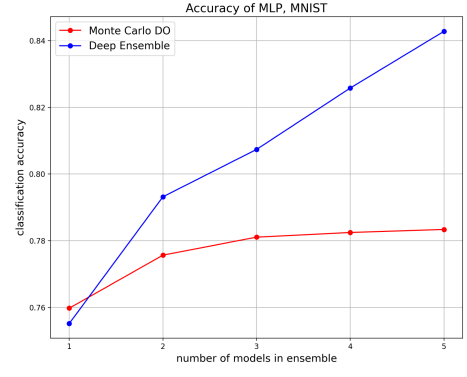
Results: Ensemble of MLPs

```
MLP(
    (feature_extractor): Sequential(
        (0): BatchNorm1d(784, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (1): Linear(in_features=784, out_features=200, bias=True)
        (2): ReLU()
        (3): Dropout(p=0.1, inplace=False)
        (4): BatchNorm1d(200, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (5): Linear(in_features=200, out_features=200, bias=True)
        (6): ReLU()
        (7): Dropout(p=0.1, inplace=False)
        (8): BatchNorm1d(200, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (9): Linear(in_features=200, out_features=200, bias=True)
        (10): ReLU()
        (11): Dropout(p=0.1, inplace=False)
    )
    (classifier): Sequential(
        (0): BatchNorm1d(200, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (1): Linear(in_features=200, out_features=10, bias=True)
    )
)
```

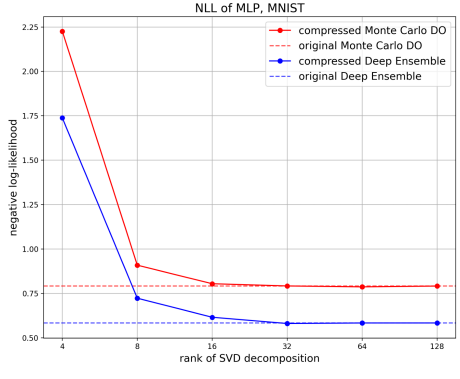
Results: SVD for Ensemble of MLPs

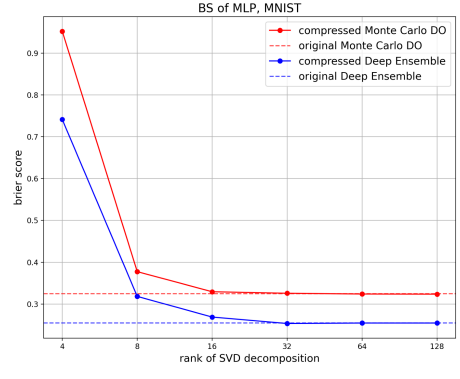


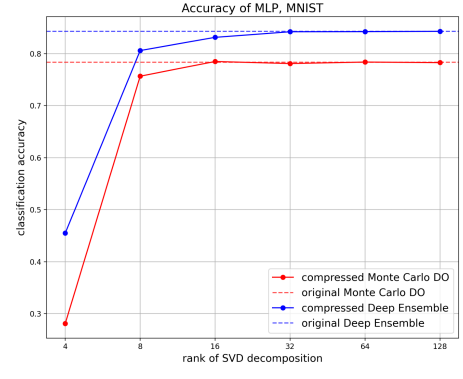




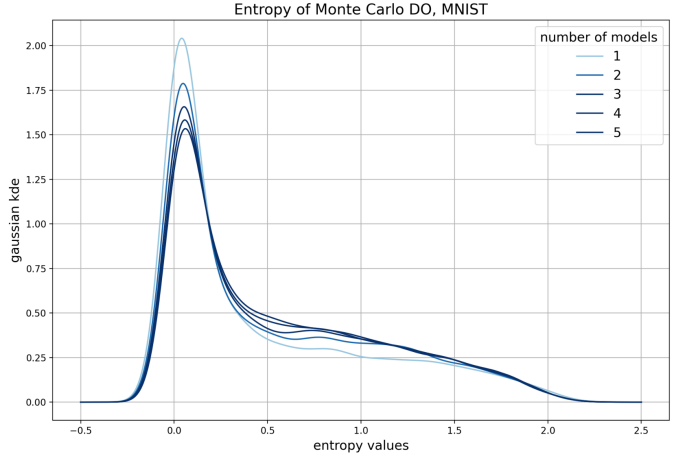
Results: SVD for Ensemble of MLPs

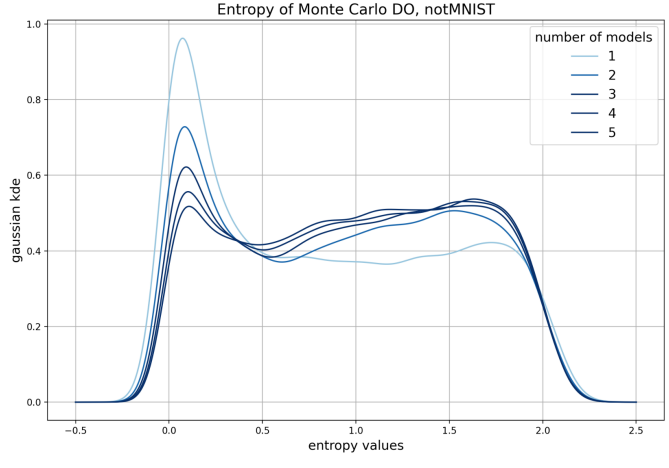




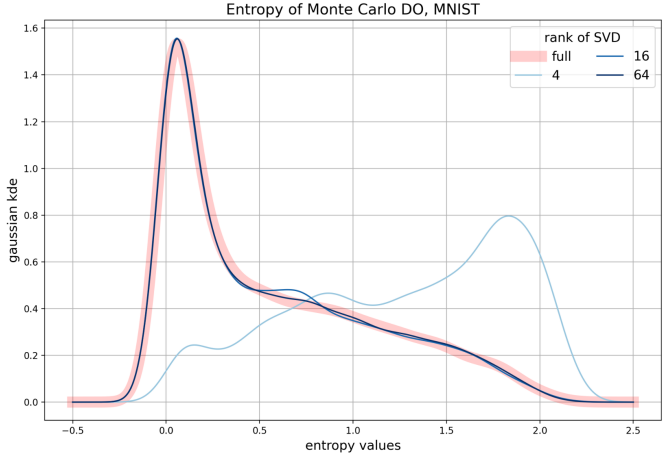


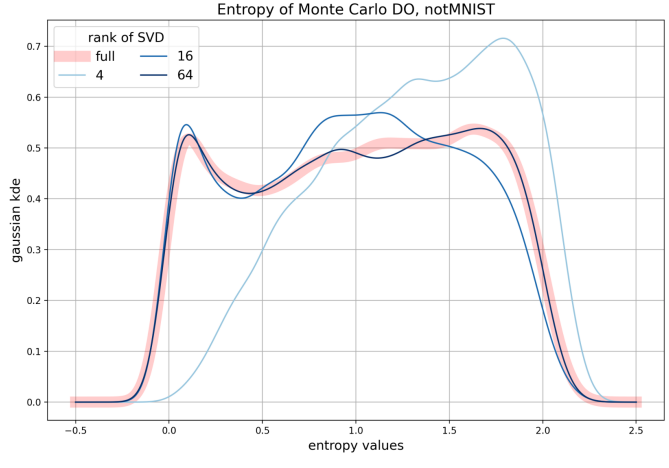
Results: SVD for Ensemble of MLPs



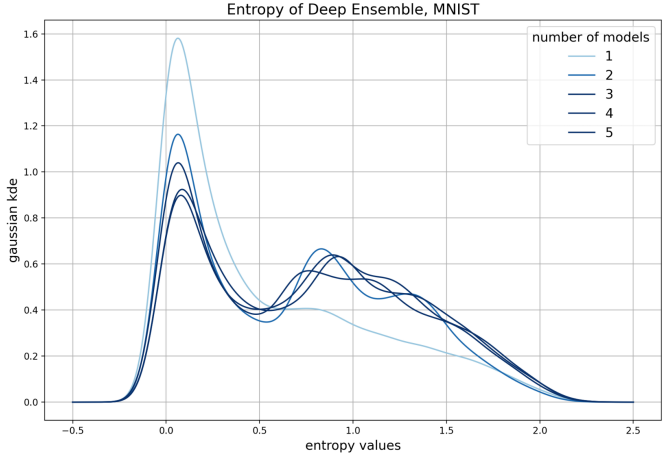


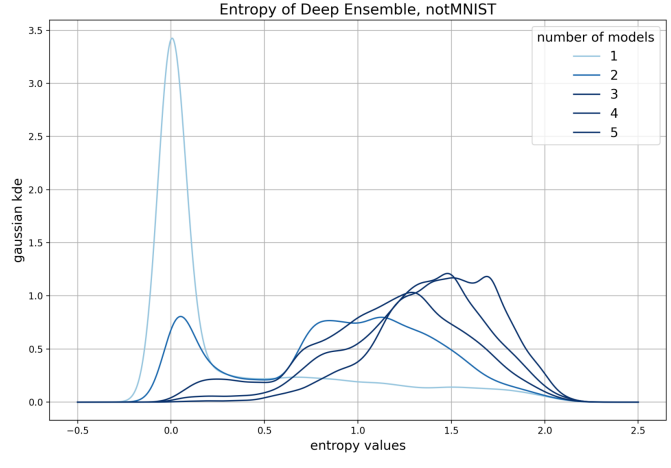
Results: SVD for Ensemble of MLPs



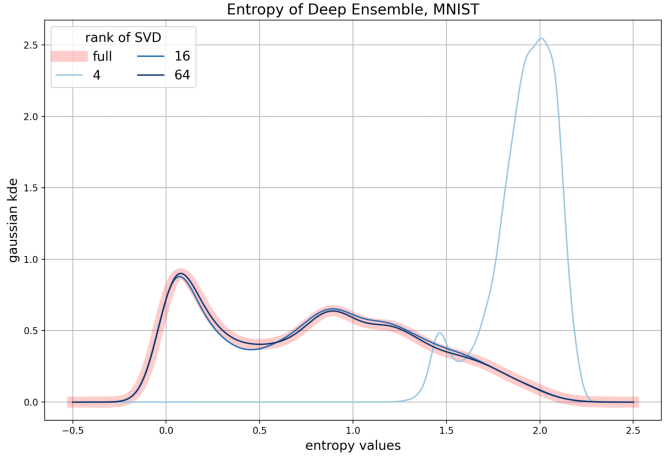


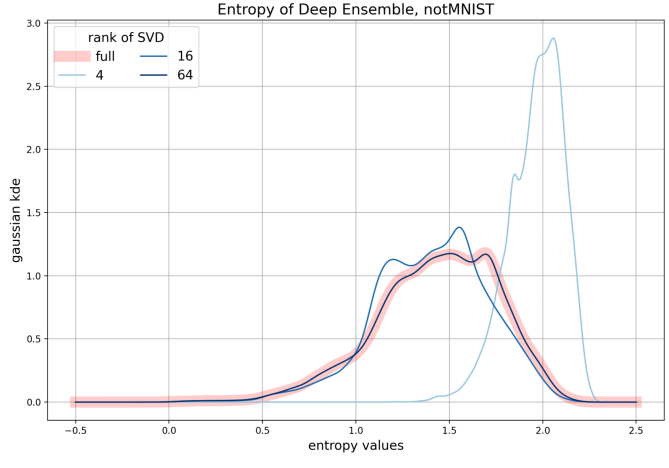
Results: SVD for Ensemble of MLPs



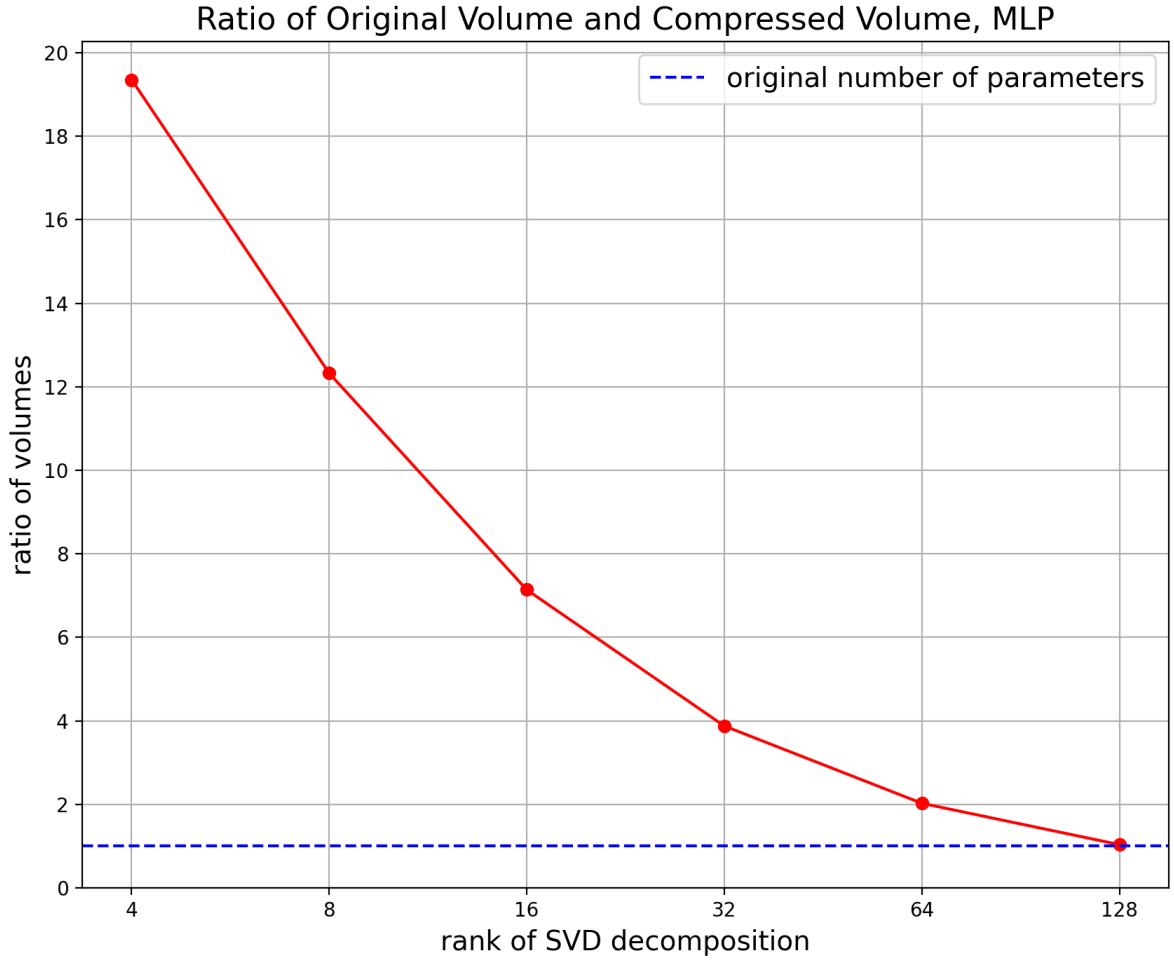


Results: SVD for Ensemble of MLPs





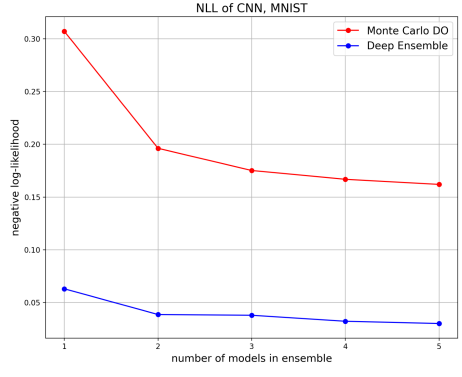
Results: SVD for Ensemble of MLPs

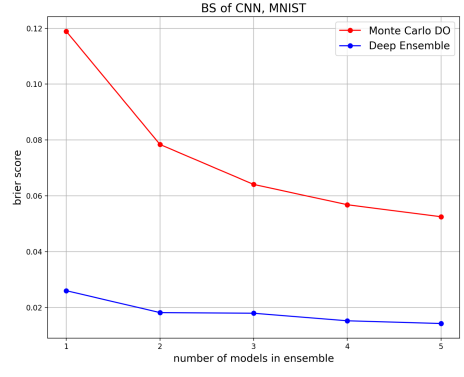


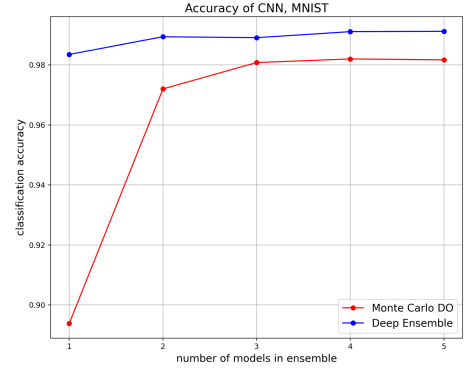
Results: TT for Ensemble of CNNs

```
CNN(
    (feature_extractor): Sequential(
        (0): BatchNorm2d(1, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (1): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (2): ReLU()
        (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        (4): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (5): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (6): ReLU()
        (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        (8): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (9): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (10): ReLU()
        (11): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        (12): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (13): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (14): ReLU()
        (15): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        (16): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (17): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (18): ReLU()
        (19): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
    (classifier): Sequential(
        (0): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (1): Linear(in_features=256, out_features=64, bias=True)
        (2): ReLU()
        (3): Dropout(p=0.1, inplace=False)
        (4): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (5): Linear(in_features=64, out_features=10, bias=True)
        (6): ReLU()
        (7): Dropout(p=0.1, inplace=False)
    )
)
```

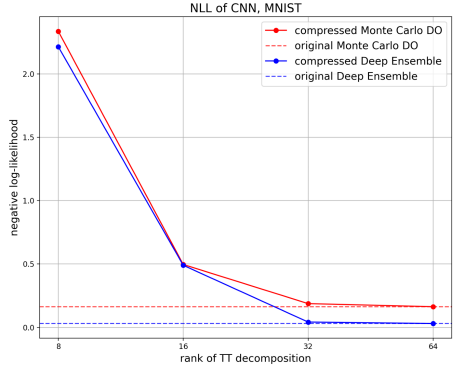
Results: TT for Ensemble of CNNs

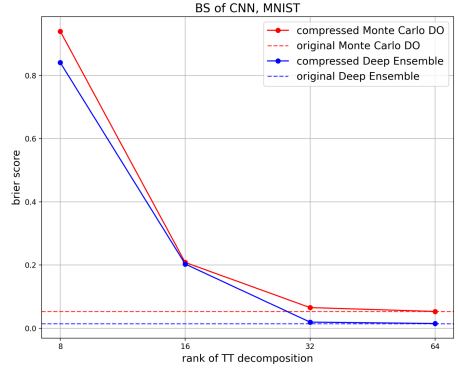


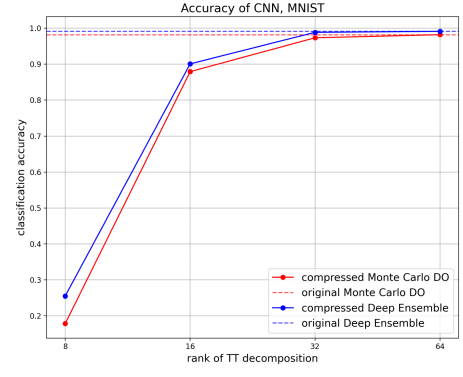




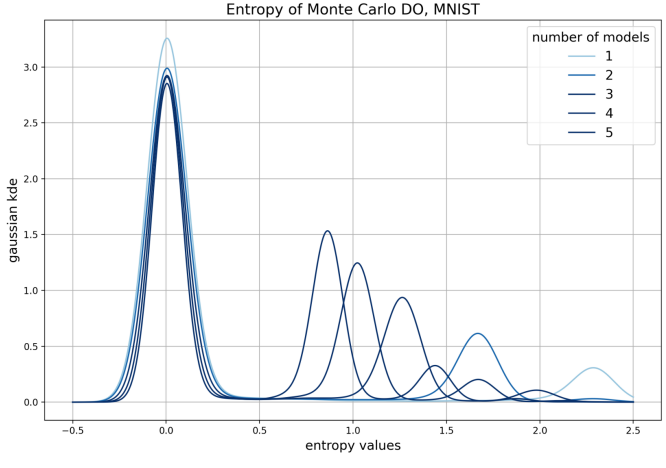
Results: TT for Ensemble of CNNs

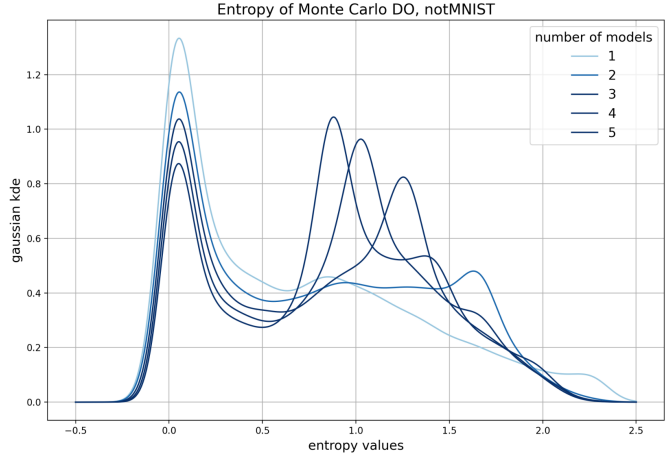




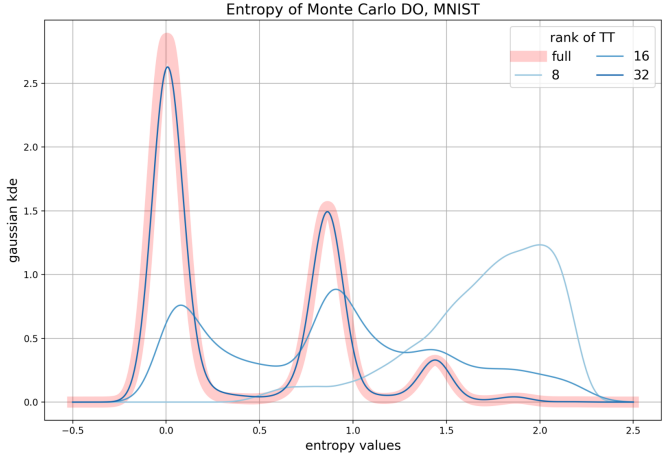


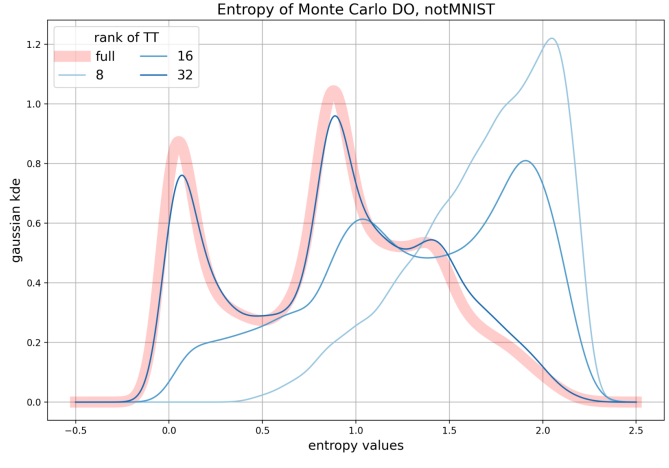
Results: TT for Ensemble of CNNs



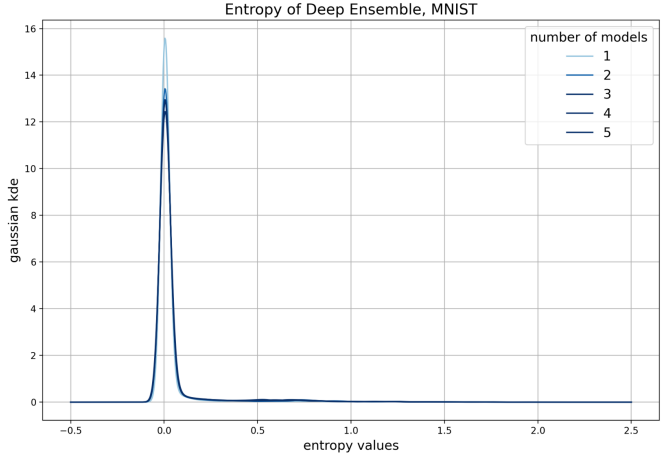


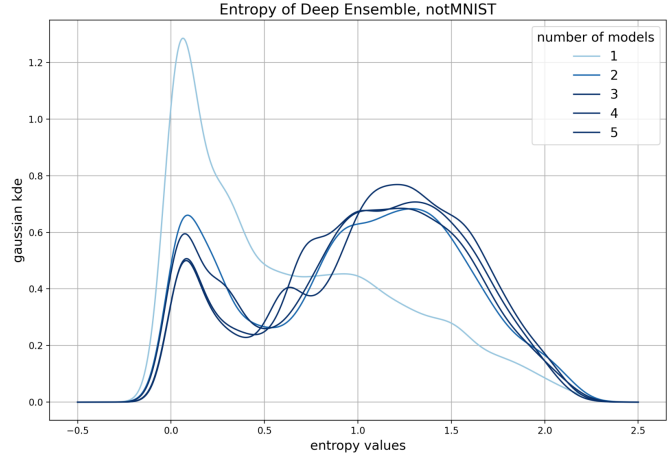
Results: TT for Ensemble of CNNs



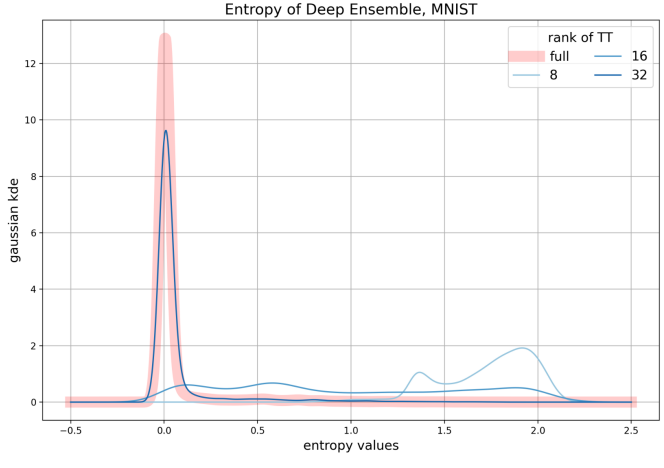


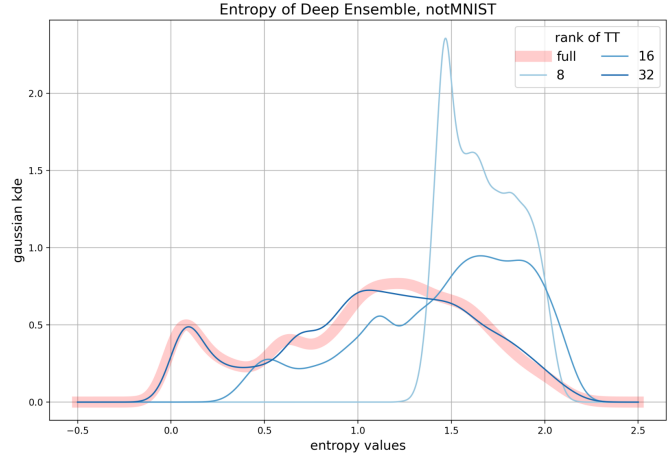
Results: TT for Ensemble of CNNs



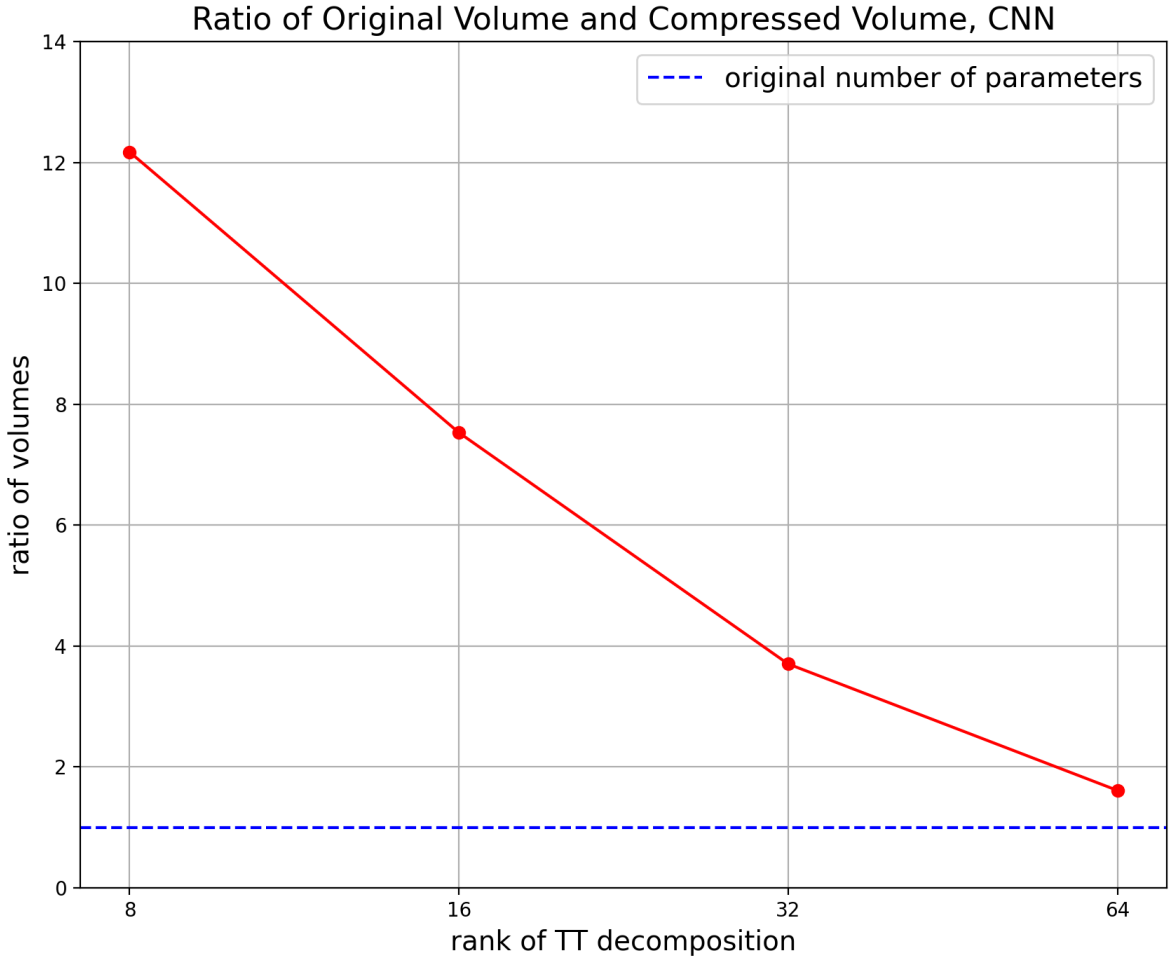


Results: TT for Ensemble of CNNs





Results: TT for Ensemble of CNNs



Summary

Main Results

- **Reproduce the comparison** of Deep Ensembles and Monte Carlo Dropout
- Show the relation between **rank of SVD and TT decompositions** and classification performance
- **Experimentally prove** that proper approximation rank still provides an opportunity for estimating the uncertainty in OOD cases

In particular, we show that too intensive compression of model leads to additional uncertainty and does not allow to detect the OOD samples.

Summary

Potentian Improvements

- Perform **fine-tuning** of compressed models
- Compare other methods of tensor decomposition (for instance, CP)

Contribution

- **Gleb Bazhenov** - uncertainty estimation insight and idea, building architecture of the models, conducting experiments
- **Lina Bashaeva** - working with theory, conducting experiments
- **Vladislav Trifonov** - preparing data, building architecture of the models
- **Saydash Miftakhov** - conducting experiments, building architecture of the models
- **Alexander Volkov** - conducting experiments, creating the presentation

Appendix: More Advanced Methods for UE

- Prior Networks [Malinin & Gales, 2018]
- Evidential Deep Learning [Sensoy et al., 2018]
- Ensemble Distribution Distillation [Malinin, Mlodozeniec et al., 2019]
- Posterior Network [Charpentier et al., 2020]
- and others...