SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF COMPUTER SCIENCE

Training on Parameters Subspaces

DEEP LEARNING & APPLIED AI

Professor:
E. Rodolà

Assistants:
L. Moschella
D. Crisostomi

Student:
Minut R. Adrian,
1942806

Academic Year 2021/2022

1 Introduction

We can think of training a neural network as a walk through an objective landscape, which is fixed once we define a network architecture and a dataset[1].
Considering a network with parameters K, which lives in a D-dimensional space, we can define W in a d-dimensional subspace such that: $K = PW$, where P is a $D \times d$ projection matrix. We can now train a network using slices ($d < D$) of the objective landscape.
This method brings some benefits:

- More control over the effective number of parameters.
- By using a random projection P, we can compress the size of the trained network by storing the seed for generating P and the subspace parameters W.

Also, we can use this family of networks for:

- Comparing different model architectures on the same problem by using the same number of parameters d, to see which performs better. The better performing model makes a more efficient use of its parameters.
- As shown in [1], given a network architecture and a dataset, we can measure the intrinsic dimension of the objective landscape, which translates to the difficulty of the task given a model architecture and a dataset.

The experiments aim to show how these models perform in classic tasks such as classification on MNIST and CIFAR-10 and how we can explore subspaces to obtain some insights from the objective landscape (Section 3.1). It's also interesting to see how we can generate the projection matrix using multiple methods (Section 3.2 & Section 3.3), and how projecting from a subspace compares to simply considering a smaller network (Section 3.4).
Given the computational and memory limits of Google Colab, the architectures considered are the Multi Layer Perceptron (MLP), for which we only introduce a Fully Connected layer between input and output, and the Standard LeNet[1].

2 Method

One way of projecting the parameters K onto a subspace is through Random Projections[1].
Another key idea is that we can actually consider the parameters of each layer separately and use a projection matrix for each of them. Instead of:

$$K = PW \quad \text{where} \quad \begin{aligned} W &= [W_1 W_2 \dots W_N] \in \mathbb{R}^d \\ K &= [K_1 K_2 \dots K_N] \in \mathbb{R}^D \\ D &= D_1 + D_2 + \dots + D_N \\ d &= d_1 + d_2 + \dots + d_N \\ P &\in \mathbb{R}^{D \times d} \end{aligned}$$

We have a projection matrix P_i for each of the N layers:

$$K_i = P_i W_i \quad \text{where } K_i \in \mathbb{R}^{D_i} \quad \forall i$$

The reference paper[1] doesn't consider this idea, but it was the only way for fitting everything inside the 12GB GPU memory offered by Google Colab when using a larger number of parameters. The results are in line with what they obtained[1], so it doesn't seem to have any impact onto the training of the network. It also introduces more hyper-parameters which can be finetuned for better results, as shown in the Weights & Biases Report (Section 5).

The following Random Projections were considered:

- Random Projection using a Gaussian Distribution[1]
- Orthogonal Random Projection
- Sparse Random Projection[2]

2.1 Gaussian Random Projection

The first method consists of generating a $D \times d$ matrix P where each element is sampled independently using a Gaussian distribution $p_{ij} \sim \mathcal{N}(0, 1)$. The large number of values and the use of the Gaussian distribution means that we will have an approximate orthogonal matrix, once we scale each column to unit length. We can also approximate this operation using a scaling factor of $1/\sqrt{d}$.

2.2 Orthogonal Random Projection

We can generate a Gaussian Random Projection (Section 2.1), and then orthogonalize P.
This can be achieved in many ways, in my implementation I used QR Decomposition since it's the most efficient algorithm offered by PyTorch[3] in terms of memory.
By orthogonalizing the matrix we have some guarantee that the projection will introduce lower distortion, which also means that the optimization process will be smoother[1].

2.3 Sparse Random Projection

With the method introduced by Achlioptas[4], we can build a Random Projection P which is both Sparse and an Approximate Orthogonal matrix. Sparse representations can make the projection operation more efficient by using proper matrix-vector multiplication algorithms.
Considering the density of the projection $s = 1/\sqrt{d}$, the original algorithm for building the projection matrix P is to sample its *i.i.d.* elements through:

$$p_{ij} = \begin{cases} \sqrt{s} & \text{with prob.} \quad \frac{1}{2s} \\ 0 & \text{with prob.} \quad 1 - \frac{1}{s} \\ -\sqrt{s} & \text{with prob.} \quad \frac{1}{2s} \end{cases} \tag{1}$$

We can introduce another optimization by using Very Sparse Random Projections [2], instead of fixing $d = 3$, we can use the actual number of dimensions of the subspace. By doing this, the matrix is even more sparse, resulting in more efficient computations, keeping the distortion introduced by the projection to a minimum. The p_{ij} values in this case are:

$$p_{ij} = \begin{cases} \frac{\sqrt{\frac{1}{s}}}{\sqrt{D}} & \text{with prob.} \quad \frac{1}{2s} \\ 0 & \text{with prob.} \quad 1 - \frac{1}{s} \\ -\frac{\sqrt{\frac{1}{s}}}{\sqrt{D}} & \text{with prob.} \quad \frac{1}{2s} \end{cases} \tag{2}$$

But the experiments (sparse_original, Fig. 5) show that such a projection matrix P leads to bad performance. The problem is that P is designed to project from the D-dimensional space to the d-dimensional subspace, while during training in the forward pass what we actually do is project from the subspace to the full space. Keeping this in mind, we can adjust for it by swapping D and d in these formulas.
Other experiments (Fig. 5) were performed to test how (sparse, (2)) compares to (sparse2full, (1)) when using $s = 1/\sqrt{D}$, and what happens if we only consider the positive values in our projection matrix (sparse2).

3 Experiments

3.1 Exploring Subspaces

The following experiments were performed using the Gaussian Random Projection method. The aim is to confirm the reference paper[1] results and to get some insights on the architectures and difficulty of the respective tasks.

In the plots there is a horizontal dotted line which represents the 90% accuracy baseline of the models trained without projection.

3.1.1 MNIST MLP

Considering the MNIST dataset and an MLP with a single FC layer, we have: 784 dimensions for our input, 200 hidden units, 10 output dimensions, so $159,010$ parameters in total.

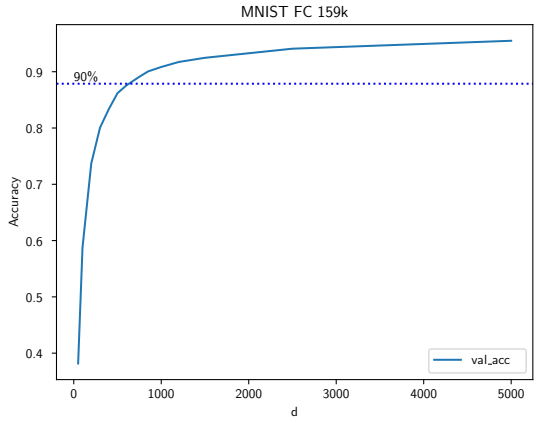

Figure 1: MLP MNIST validation accuracy depending on d.

By incrementing the number of dimensions of the subspaces and measuring the accuracy we get a smooth line which shows us that by using around 750 parameters we can obtain 90% accuracy of our baseline model, and by adding more parameters we quickly get close to the original performance (0.9763).

3.1.2 CIFAR MLP

Considering the CIFAR-10 dataset and the MLP, we have: 784×3 dimensions for our input, 200 hidden units, 10 output dimensions, so $472,610$ parameters in total.

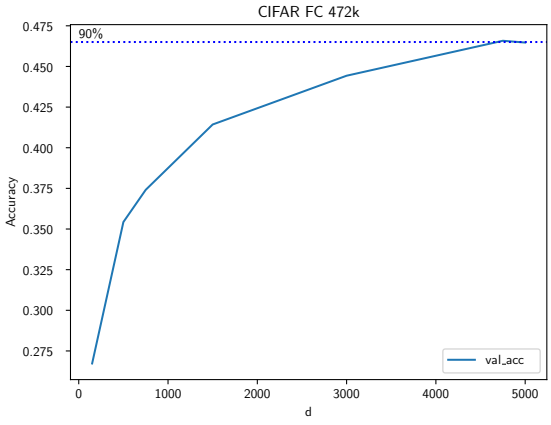

Figure 2: MLP CIFAR validation accuracy depending on d.

In this case we can observe lower performance, which is what we expect since CIFAR-10 is a more difficult classification task. The models only get to 90% baseline accuracy with 4.5k parameters, but the performance increase is lower this time. Further testing of parameters wasn't possible due to GPU memory limitations.

3.1.3 MNIST CNN

Considering the MNIST dataset and LeNet, we have: 784 input dims and 1 channel, 6 Kernels (5×5) – Max Pooling (2×2) – 16 Kernels (5×5) – Max Pooling (2×2) – 120 FC – 84 FC – 10 FC. So $61,706$ parameters in total.

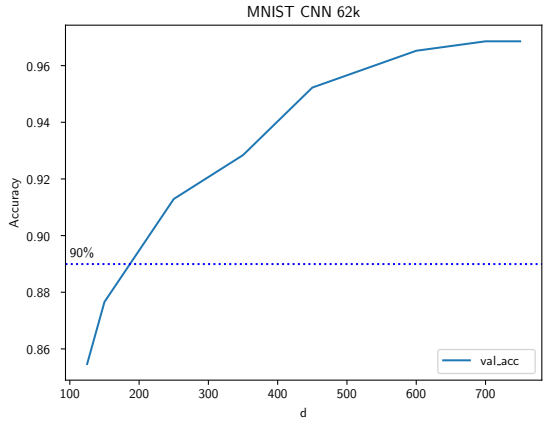

Figure 3: LeNet MNIST validation accuracy depending on d.

We get very good results even with few parameters (150), and by 200 we already reach the 90% baseline, but the models can't reach the original performance(0.9888) and it takes many parameters to get to that level.

3.1.4 CIFAR CNN

Considering the CIFAR-10 dataset and LeNet, we have: 784 input dims and 3 channels, 6 Kernels (5×5) – Max Pooling (2×2) – 16 Kernels (5×5) – Max Pooling (2×2) – 120 FC – 84 FC – 10 FC. So $62,006$ parameters in total.

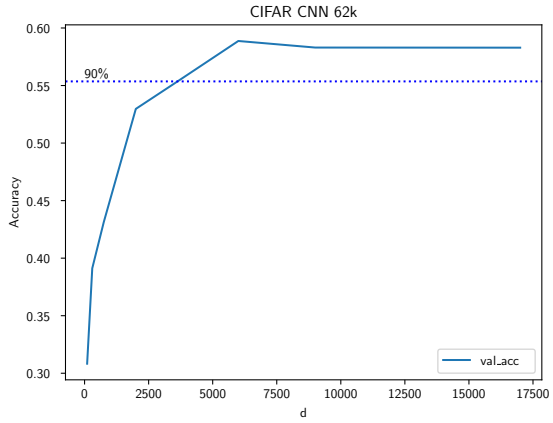

Figure 4: LeNet CIFAR validation accuracy depending on d.

We get better results compared to the MLP models, but we saturate the performance at around 6k parameters with an accuracy of 0.5829, not reaching the original performance (0.6141).

3.2 Sparse Matrix Implementation

We can make an efficient use of sparse projection matrices through algorithms designed for sparse representations. PyTorch offers very limited support[5] for sparse matrices, especially for the gradient computations required by the backward pass. Still, for basic operations like addition and matrix-vector multiplication it was possible to build an implementation.
Multiple representations were considered:

- Dense (sparse): used as a baseline.
- Coordinate Format (sparse_coo): lower memory usage but high computational cost.
- Compressed Sparse Row (sparse_csr): low memory usage but high computational cost.
- Compressed Sparse Column (sparse_csc): low memory usage and lower computational cost compared to sparse_csr, the best performing sparse representation for our use case.

In Fig. 5 we can see the benchmarks for multiple random projection types, all of which were tested using the model with most parameters (Section 3.1.2), and $d = 4.5$k.

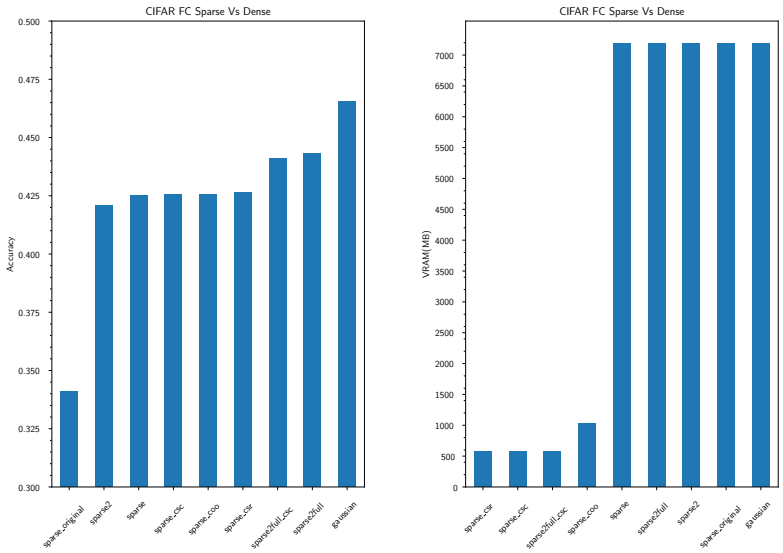

Figure 5: Accuracy and VRAM usage for MLP CIFAR

The best performing in terms of accuracy is the simpler Gaussian Random Projection (Section 2.1).

We can see that by not swapping the D and d dimensions (sparse_original), or by only having positive values in our projection matrix (sparse2), the model performs worse.
When we use as matrix elements $\pm\sqrt{D}$ (sparse2full) we unexpectedly get better performance, but by looking at the loss curves (Section 5) we can see that it's noisier with respect to the others, which is a sign of additional distortion introduced by the projection.

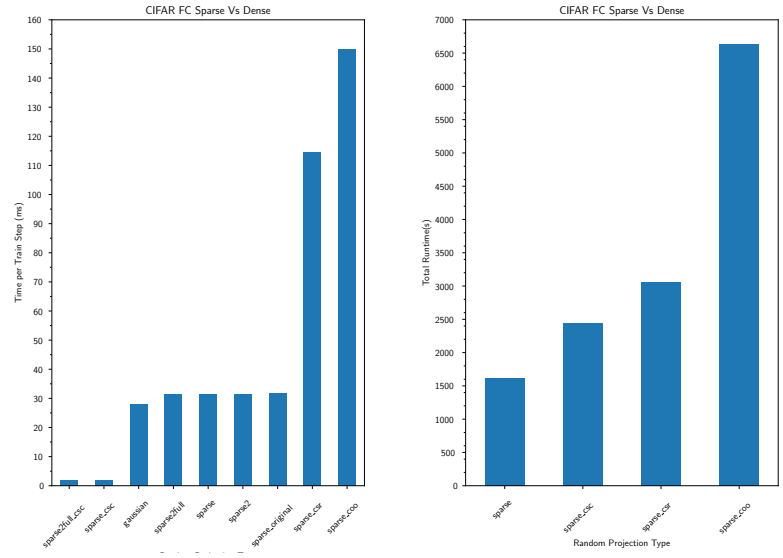

Figure 6: Train Time Step and Total Runtime for MLP CIFAR

Training step times were measured using CUDA Events[6] for the best precision possible, but it seems like the PyTorch CSC implementation brings some unexpected behaviour (Fig. 6).
As you can see, the train step time is very low, but the actual runtime isn't lower compared to the runs of the dense representations, even though the number of epochs is the same (Section 5).
Another peculiar result is that by using the CSC implementation for sparse2full we get a different but similar performance measures compared to its dense representation counterpart, this may also indicate that some unexpected behaviour is going on.

3.3 Orthogonal Random Projection

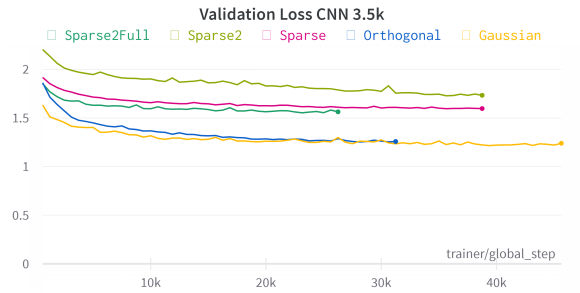

Figure 7: Validation Loss for MLP CIFAR

By orthogonalizing the projection matrix we don't gain much in terms of performance, and this method requires a lot of VRAM for the QR decomposition process. For these reasons, mainly because of memory issues, this method wasn't considered in the other experiments.

3.4 Smaller models

3.4.1 MLP

Even the smallest MLP model has 7.9k parameters for MNIST and 23.5k for CIFAR. In these two cases, the accuracy obtained for MNIST is 0.916 and for CIFAR is 0.395 (Section 5).
It wasn't possible to test models with the same amount of parameters using projection because of the large amount of VRAM required, but the same performance metrics are reached at $d = 1.2$k parameters for MNIST(Section 3.1.1) and $d = 6.5$k for CIFAR(Section 3.1.2).
As expected, the Fully Connected layers are not efficient in the number of used parameters, we have a very large gap between the models using projections and the simpler models.

3.4.2 CNN

In the following experiments the models using projection are LeNet with a fixed architecture, same as (Section 3.1), while the smaller models are LeNets with different hyper-parameters: kernel size, output channels, stride, hidden size.

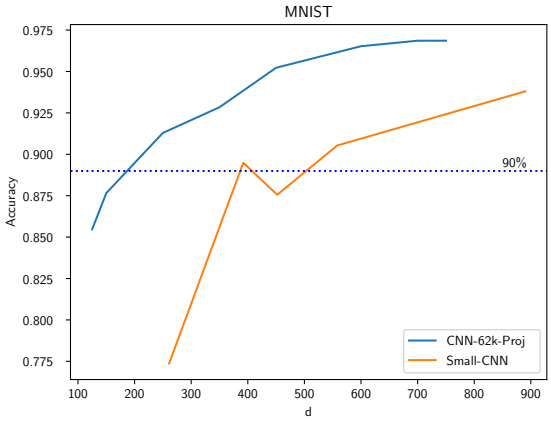

Figure 8: MNIST LeNets Small vs Projection.

The gap in the case of CNNs is lower, but it's still there. As we consider less parameters, we can see that the model using projection performs significantly better.

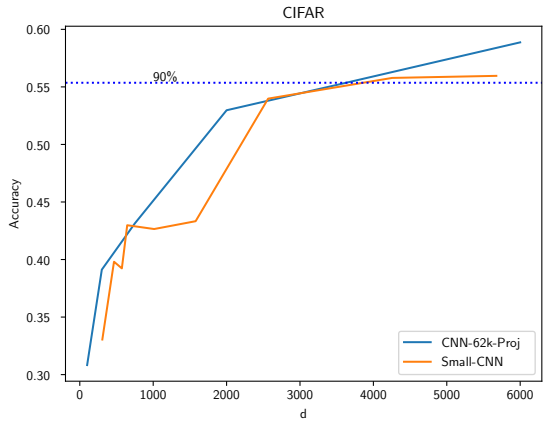

Figure 9: CIFAR LeNets Small vs Projection.

Interestingly, if we consider a more difficult task and dataset, in this case CIFAR-10, the gap pretty much closes. My hypothesis is that in this case the Fully Connected layers of the LeNet, which are used for predicting the output based on the high-level features of the images, extracted through the Convolutional layers, are actually fully exploited for this task.

4 Conclusions

Through these experiments a few limitations of this family of models were uncovered: high VRAM usage, high computational cost, lower performance.
They have major implications in terms of the applicability of these models, since the only practical advantage is that by storing the seed of the random generation of P and the W parameters we can compress the model in storage.
We would still be able to efficiently perform the inference operations by calculating once and for all $K = PW$, but the resulting model would have lower performance than a model which was originally trained without projection. Also, training a model without projection is more cost-effective in terms of computational power.

Although it's not the kind of model that we could deploy in production, the theoretical results are interesting. We can explore subspaces to infer the difficulty of a task based on the dataset and the model architecture, and make comparisons by seing how efficient they are in the use of their parameters.
We can also test different values of d_i for each layer of the network and see the impact on the training process (Section 5). Depending on the amount of parameters involved, we would need to change the projection method by sacrificing some model performance, but since it may not be the focus of the study it may still be worth it for the investigation of the model architectures.

5 Additional material

All model optimizations were logged on Weights & Biases, and the data can be accessed at the following link: Weights And Biases Report. There is also an additional experiment about how changing the hyper-parameters of the projections may bring better predictions.

References

[1] Chunyuan Li, Heerad Farkhoor, Rosanne Liu, and Jason Yosinski. Measuring the intrinsic dimension of objective landscapes. In *International Conference on Learning Representations*, 2018.

[2] Ping Li, Trevor J. Hastie, and Kenneth W. Church. Very sparse random projections. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, page 287–296, New York, NY, USA, 2006. Association for Computing Machinery.

[3] Pytorch qr decomposition. `https://pytorch.org/docs/stable/generated/torch.linalg.qr.html#torch.linalg.qr/`.

[4] Dimitris Achlioptas. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *Journal of Computer and System Sciences*, 66(4):671–687, 2003. Special Issue on PODS 2001.

[5] Pytorch sparse api. `https://pytorch.org/docs/stable/sparse.html`.

[6] How to measure execution time in pytorch. `https://discuss.pytorch.org/t/how-to-measure-execution-time-in-pytorch/111458`.