Machine Learning Engineer Nanodegree
Capstone Project

Irina Gruzinov

January 20, 2020

Definition

Project Overview

In this project I will apply a Denoising Convolutional Neural Network (DCNN) to a problem of reconstructing missing traces on 2D seismic sections. The DCNN's architecture I want to implement was successfully used for denoising images when the noise destribution is unknown as well as for up-sampling images. The ability of a neural net of this architecture to learn both content and noise is important for this project, because the noise and the useful signal have overlapping frequency spectra in seismic images. I will be using the DCNN setting suitable for an up-sampling task.

Problem Statement

Seismic methods are the most important methods of remote sensing in oil and gas exploration, because they allow to reconstruct 3D subsurface Earth structure by collecting sound (seismic) data on the surface only. Seismic data acquisition is a very expensive field operation and all the imperfections in data acquisition need to be mitigated during processing. A very important processing step is to regularize data, that is, to interpolate traces into regular grid and to fill gaps in data. This task is difficult on the seismic data because a seismic section is made of traces, and each trace is a time record of sound waves reflected from underground layers. Since these sound waves propagate at different directions, their signatures will be recorded at different times at different recorder locations. See Figure 1 upper panel for a representation of a seismic section. A proper interpolation should be along a waveform, i.e. along the direction in which the amplitude changes the least. That requires to delineate many interfering waveforms across many traces and this is a very difficult and ambiguous task. Deep Convolutional Neural Nets have an unsurpassed ability to learn patterns and statistics on data originated from the same distribution and thus can infer missing data better than any imperative algorithm. I will be using the gray image representation of the seismic section, see Figure 1, middle section. The values will be from -1 to 1.

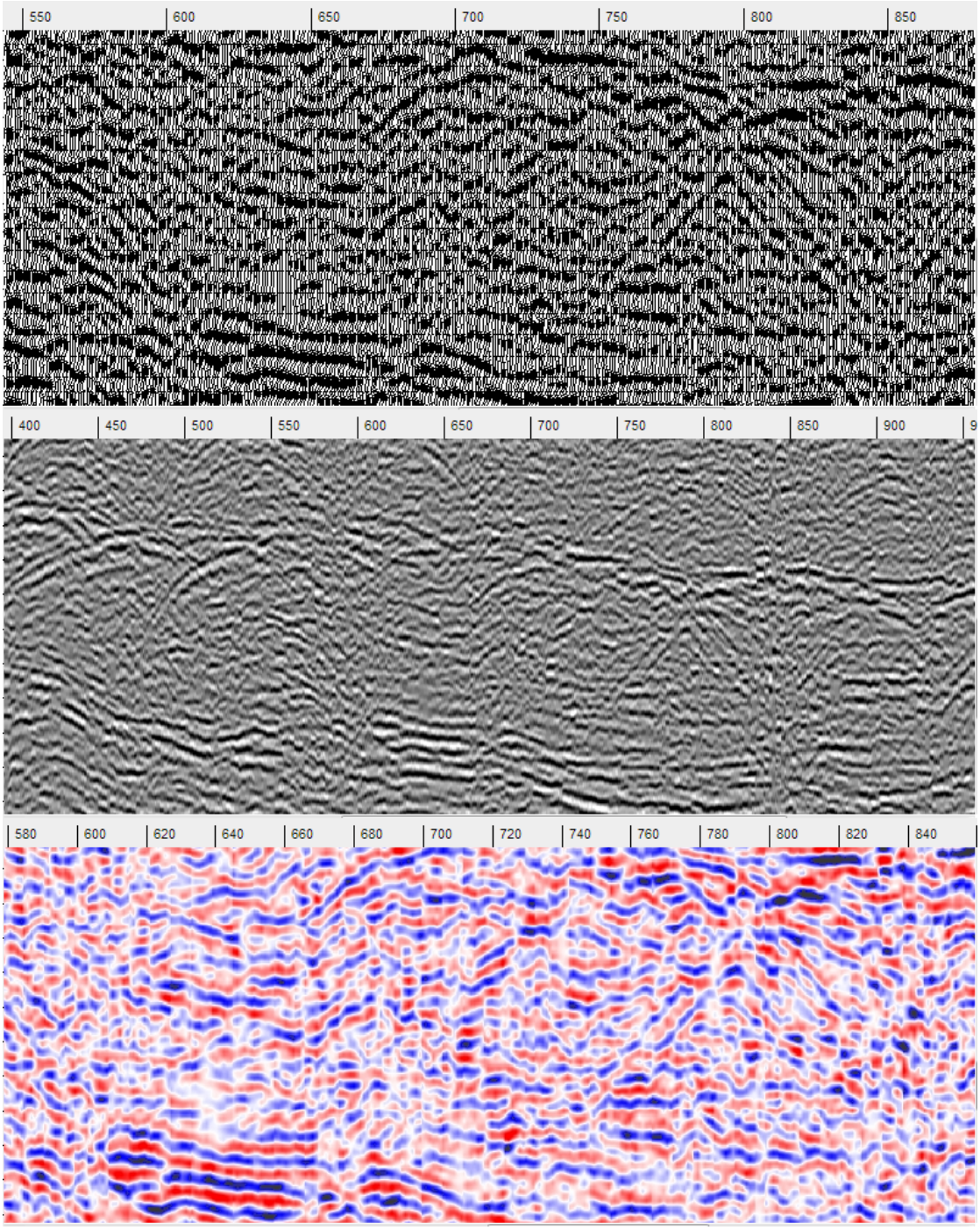
Metrics

High quality processed seismic sections, Figure 2 left column, will be downsampled in different ways to generate training data. Sometimes the downsampling will be regular, when traces are dropped at a regular interval, Figure 2 middle column. Sometimes the dowsampling will

be random, when a certain percentage of traces will be dropped randomly, see Figure 2 right column.

The model performance will be judged by the quality of infilling missing traces, i.e. by comparing the images modeled by the DCNN with images before downsamling. A good qualitative measure of the quality of data restoration will a root mean square error (RSME): $RMSE = (\frac{1}{N}\sum_{i=1}^{N}(X_i - \hat{Y}_i)^2)^{1/2}$, where X is a ground truth image and $\hat{Y}$ is a restored image. Subscript i indicates an element-wise operation. RMSE is a standard measure of deviation between two sets of values, but it's sensitive to spikes and fails on missing data. So infilling missing data and clipping data to a reasonable range of values prior to RMSE calculation will be implemented here.

Figure 1: Three common ways of displaying seismic sections. The upper section is a wiggle plot, in which separate traces and the recorded wavelets are distinguishable. The middle section is a gray plot, and the lower panel shows the seismic section in false colors: negative values are red, and positive values are blue.



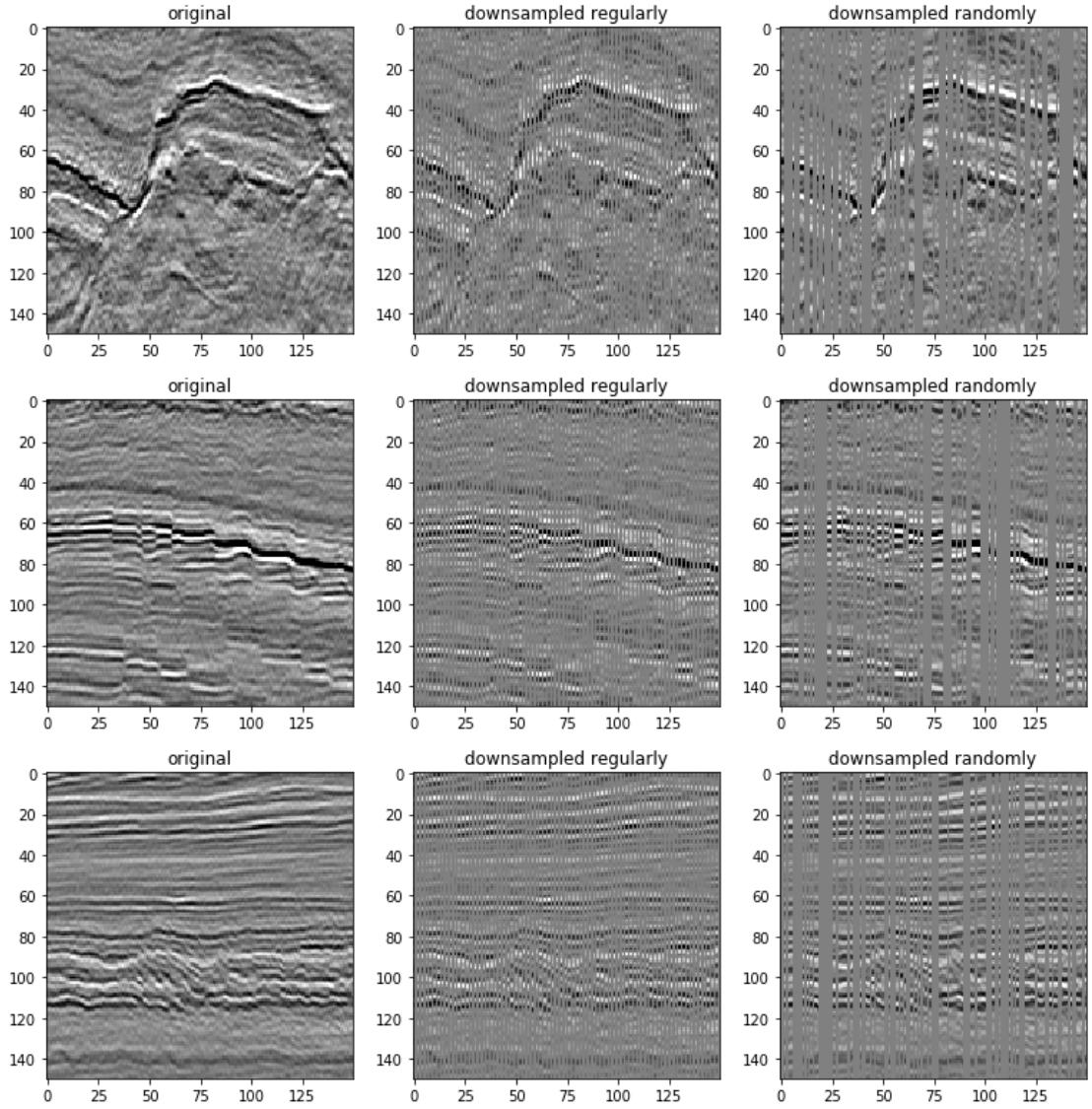


Figure 2: Examples of the training data preparation. The left column is the ground truth seismic images. The middle column is regularly downsampled images: in this case, every other trace is omitted. The right column is randomly downsampled images: randomly selected half of the traces were omitted. Random downsampling can result in large gaps in the data, which makes it the most difficult case for the interpolation.

Analysis

Data Exploration

A large processed 3D seismic volume provides the data that will be used in this study. Out of this 3D volume, 17666 2D seismic images of size 99×99 were sliced and labeled and specially prepared for Machine Learning studies of seismic facies classification. They represent four types of subsurface structures. We don't need to know the names of these structures, we just recognize the fact that there is a variety in the dataset. Because all 2D images originate from the same volume, they are also normalized and balanced in amplitude and have similar other characteristics, like frequency content and wavelet shape. In other words, the training dataset cames from the same distribution and is representative of its type of data.

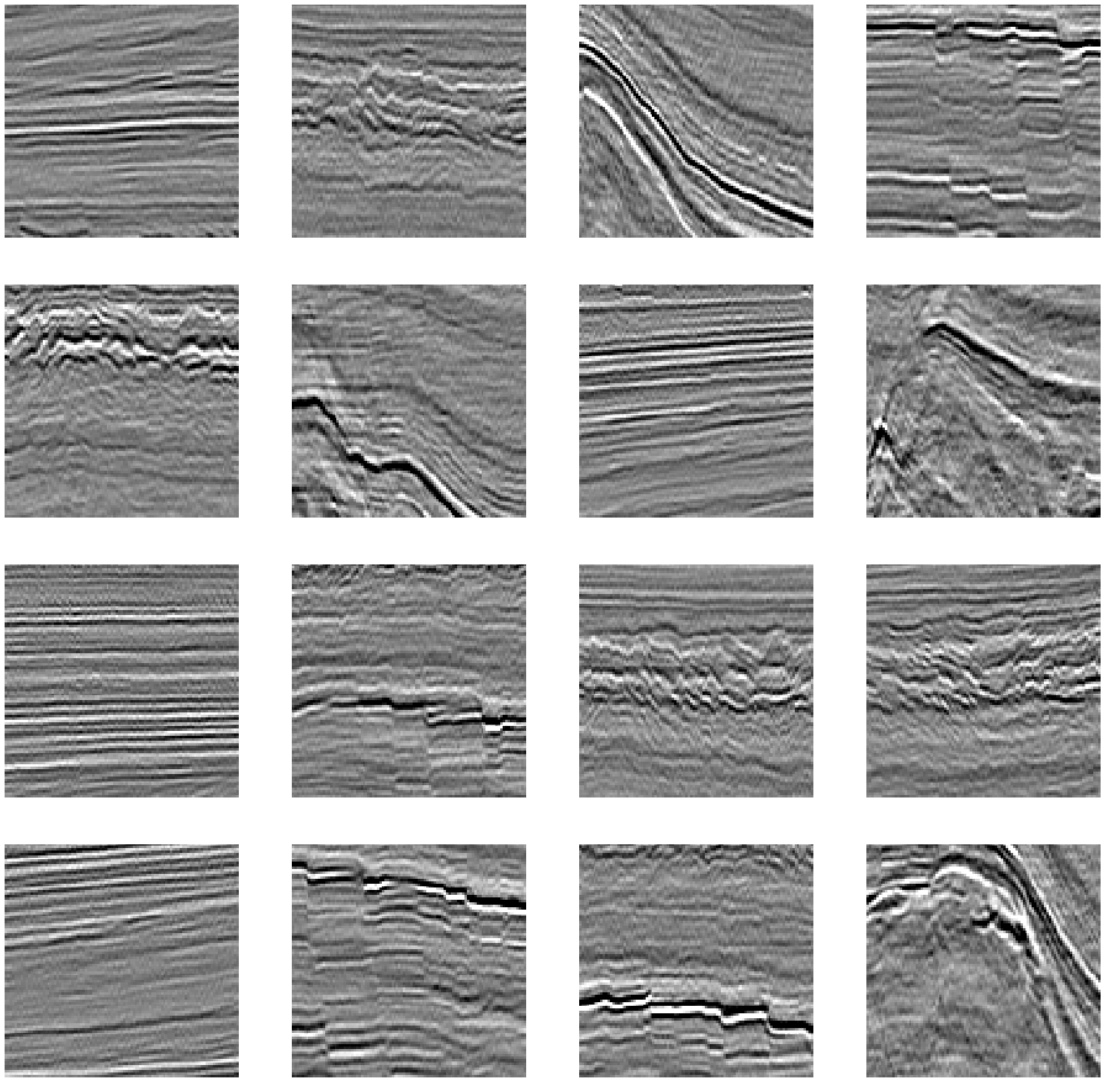


Figure 3: Random selection of the ground truth images. There are four types of geological structures in this dataset: salt dome, faults, flat layers and uncompacted reflectors.

Exploratory Visualization

The training data will be prepared on the fly from the ground truth images. The ground truth images will be downsampled with one of six downsampling algorithms listed below selected at random:

1) Every fourth trace is dropped, RMSE = 0.16
2) Every third trace is dropped, RMSE = 0.18
3) Every other trace is dropped, RMSE = 0.22
4) 20% of traces selected randomly are dropped, RMSE = 0.14
5) 30% of traces selected randomly are dropped, RMSE = 0.17
6) 40% of traces selected randomly are dropped, RMSE = 0.20

The first algorithm is the easiest case for the interpolation, the sixth algorithm is the most difficult, because it can result in big gaps which are impossible to fill just because the information needed to guess the missing data might not be there anymore. See also Figure 4 for illustration.

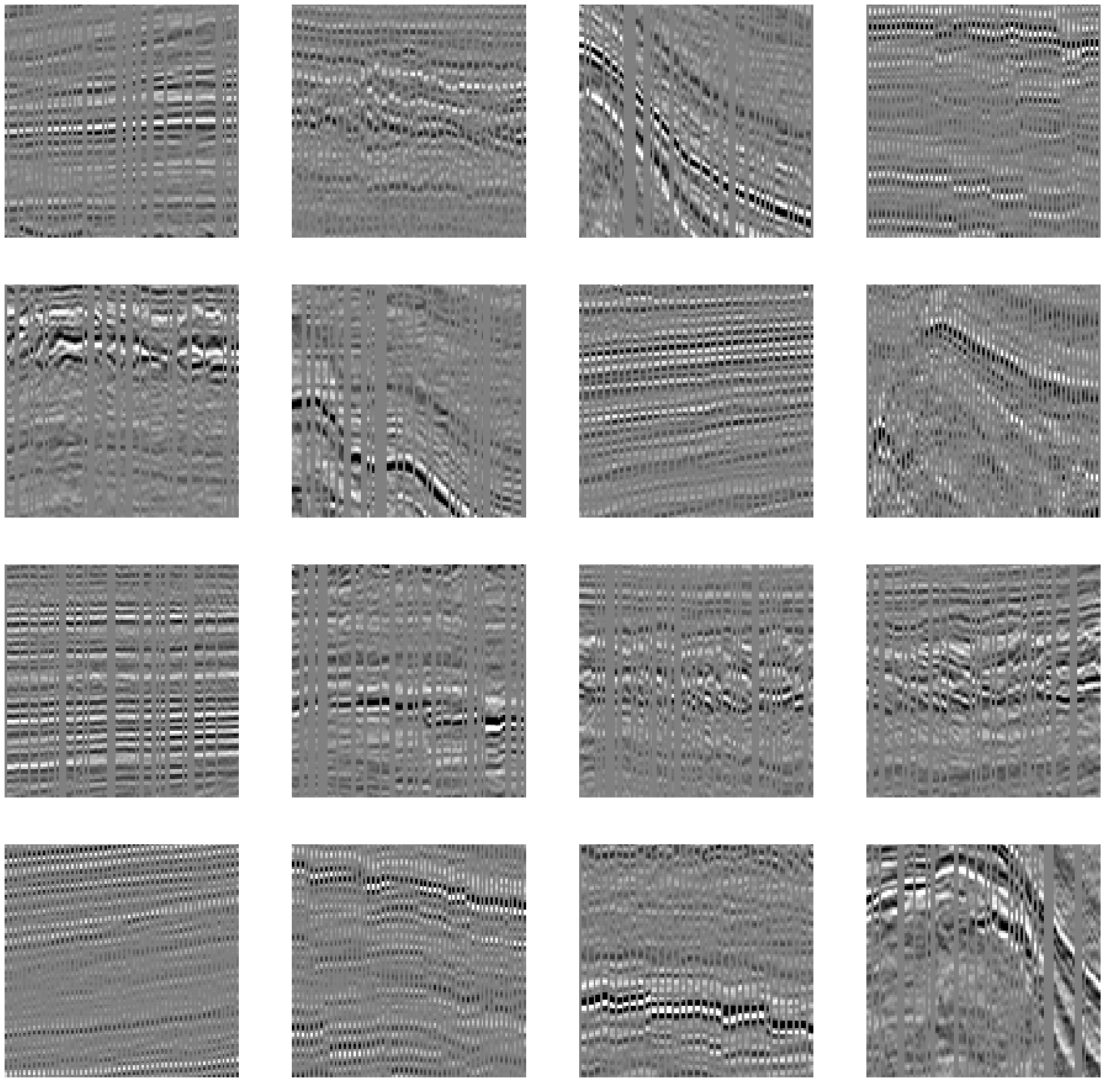


Figure 4: These types of downsampled images will be fed to DCNN. Some downsampling is random, some is regular

Benchmark

As a benchmark, I will apply SciPy griddata 2D interpolation to downsampled images. See Figure 5 for illustration. It's apparent that the benchmark interpolation is not very good restoring dipping events in data. It also doesn't attempt to restore missing data at the edge, i.e. doesn't attempt to extrapolate. In order to be able to apply our metrics, i.e. calculate the RMSE, the NaN values at the edge on the data after SciPy griddata 2D interpolation will be repared by repeating the last valid trace.

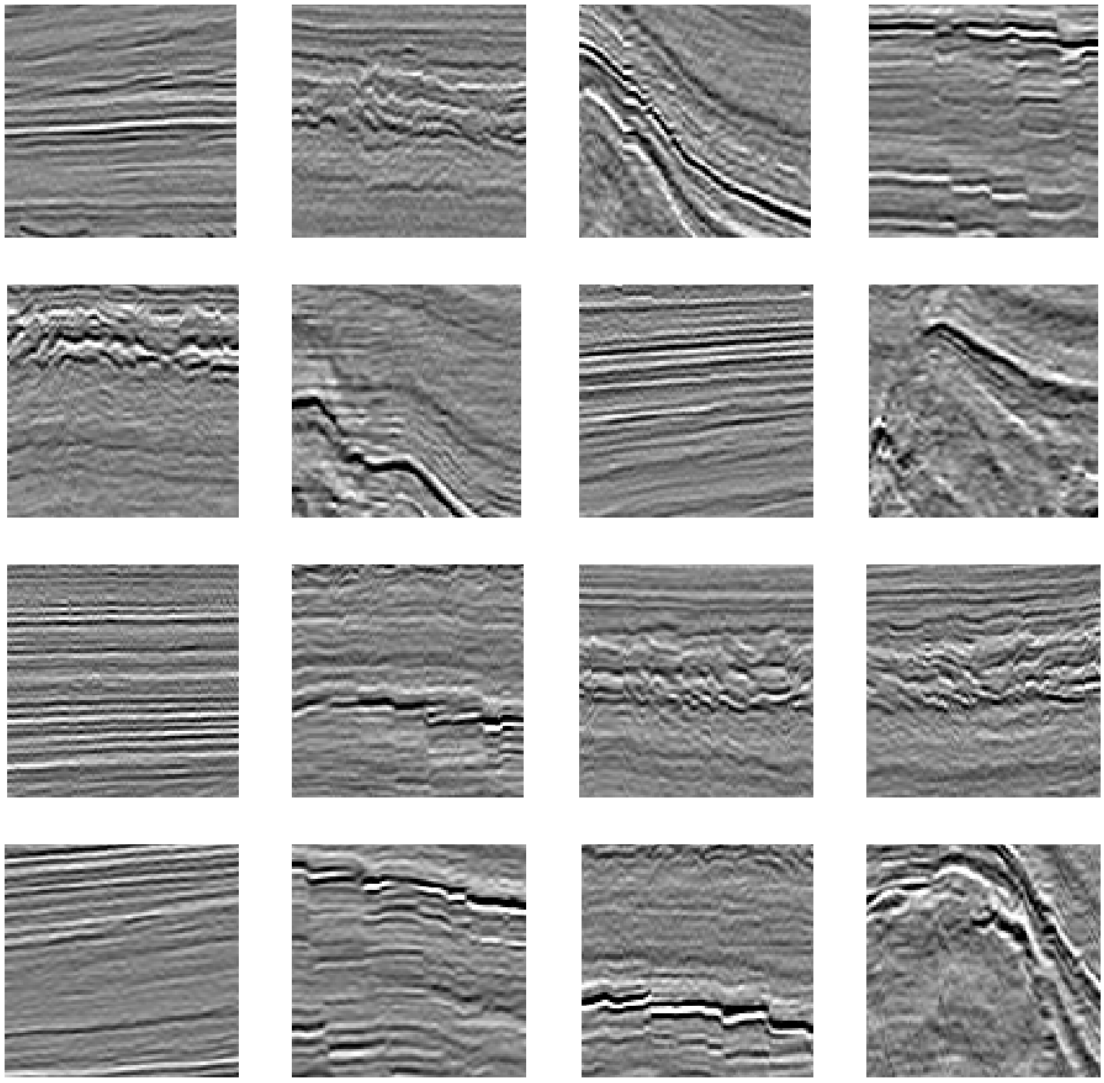


Figure 5: SciPy griddata 2D interpolation applied to downsampled images. This will be a benchmark for this study.

Algorithms and Techniques

To train a DCNN, downsampled seismic images will be provided as input, and a loss function will be calculated as a mean squared error between the output and the ground truth image. The desirable final outcome is that the trained DCNN would produce a restored image practically indistinguishable from the ground truth.

The structure of DCNN is very simple, which makes it also robust and stable. See Figure 6 for illustration. The first layer is a convlutional layer with $ReLU$ activation. It has 64 filters of the size 3×3. Then there is a number of layers each consisting of a convolutional layer with a batch normalization and $ReLU$. The last layer is a convolutional layer. The convolutional layers in the middle and in the last layer consist of 64 filters of the size $3 \times 3 \times 64$. There are no pooling layers.

The number of layers in the DCNN determines a tradeoff between performance and efficiency and will be a hyperparameter in this study. I start with 17 layers as in the referenced paper.

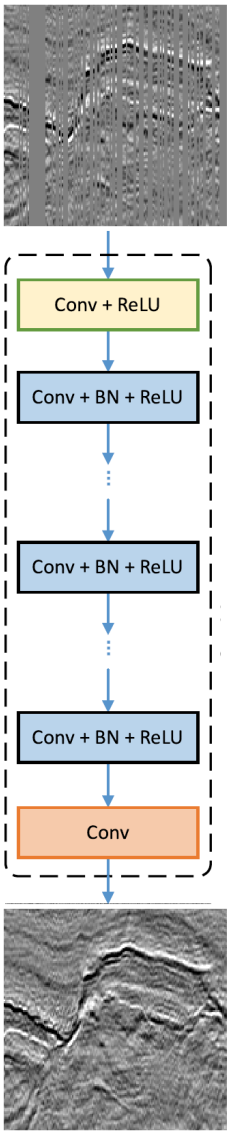


Figure 6: Trained DCNN should restore a downsampled image to a state indistinguishable from the ground truth.

Methodology

Data Preprocessing

The training data will be generated on the fly during training from a homogeneous dataset of 17666 2D seismic sections. To generate a training example, a ground truth image will be downsampled. The downsampling can be random or can be regular, and to a different degree in terms of number of dropped traces, as explained earlier. The randomization of the training examples is important for training, so that there were no repeated patterns for NN to memorize. Even if the downsampling is regular, the first dropped trace will be random. The data will be shuffled every epoch and one of six downsampling algorithms will be selected at random every time a new batch is generated. In other words, there are three stages of randomization in generating training examples: shuffling the images, random selection of the downsampling algorithms, and randomization of which traces to drop. This randomization has a regularization effect similar to a dropout method.

Implementation

A Convolutional Neural Network was implemented in Pytorch. The batch size was 32 of 99×99 mages, and the number of epochs was 40. The training took 34 hours, at 50 minutes per epoch on a modest Nvidia GeForce GTX 1070 with 8GB of VRAM. The hyperparameters were assessed by training on a smaller subset, and then re-evaluated after training for 10 epochs.

```
(0): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(1): ReLU(inplace=True)
(2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(3): BatchNorm2d(64, eps=0.0001, momentum=0.95, affine=True,
    track_running_stats=True)
(4): ReLU(inplace=True)
(5): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(6): BatchNorm2d(64, eps=0.0001, momentum=0.95, affine=True,
    track_running_stats=True)
(7): ReLU(inplace=True)
(8): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(9): BatchNorm2d(64, eps=0.0001, momentum=0.95, affine=True,
    track_running_stats=True)
(10): ReLU(inplace=True)
(11): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(12): BatchNorm2d(64, eps=0.0001, momentum=0.95, affine=True,
    track_running_stats=True)
(13): ReLU(inplace=True)
(14): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(15): BatchNorm2d(64, eps=0.0001, momentum=0.95, affine=True,
    track_running_stats=True)
(16): ReLU(inplace=True)
(17): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(18): BatchNorm2d(64, eps=0.0001, momentum=0.95, affine=True,
    track_running_stats=True)
(19): ReLU(inplace=True)
(20): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(21): BatchNorm2d(64, eps=0.0001, momentum=0.95, affine=True,
    track_running_stats=True)
```

```
(22): ReLU(inplace=True)
(23): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(24): BatchNorm2d(64, eps=0.0001, momentum=0.95, affine=True,
    track_running_stats=True)
(25): ReLU(inplace=True)
(26): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(27): BatchNorm2d(64, eps=0.0001, momentum=0.95, affine=True,
    track_running_stats=True)
(28): ReLU(inplace=True)
(29): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(30): BatchNorm2d(64, eps=0.0001, momentum=0.95, affine=True,
    track_running_stats=True)
(31): ReLU(inplace=True)
(32): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(33): BatchNorm2d(64, eps=0.0001, momentum=0.95, affine=True,
    track_running_stats=True)
(34): ReLU(inplace=True)
(35): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(36): BatchNorm2d(64, eps=0.0001, momentum=0.95, affine=True,
    track_running_stats=True)
(37): ReLU(inplace=True)
(38): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(39): BatchNorm2d(64, eps=0.0001, momentum=0.95, affine=True,
    track_running_stats=True)
(40): ReLU(inplace=True)
(41): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(42): BatchNorm2d(64, eps=0.0001, momentum=0.95, affine=True,
    track_running_stats=True)
(43): ReLU(inplace=True)
(44): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(45): BatchNorm2d(64, eps=0.0001, momentum=0.95, affine=True,
    track_running_stats=True)
(46): ReLU(inplace=True)
(47): Conv2d(64, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
```

Refinement

I monitored the performance of the DCNN as it trained by using the last five saved checkpoints for inference for different scenarios of downsampling. Figure 7 shows a bar plot for comparison of the RMSE for the input, the RMSE for the benchmark interpolation, and the RMSE for the DCNN inference averaged over the last five epochs 35-39. The performance of the DCNN is better than the benchmark in all cases, except the case of 50% regular downsampling. The DCNN performance is especially better on the irregular downsampling cases. This observation will be further illustrated in the next section.

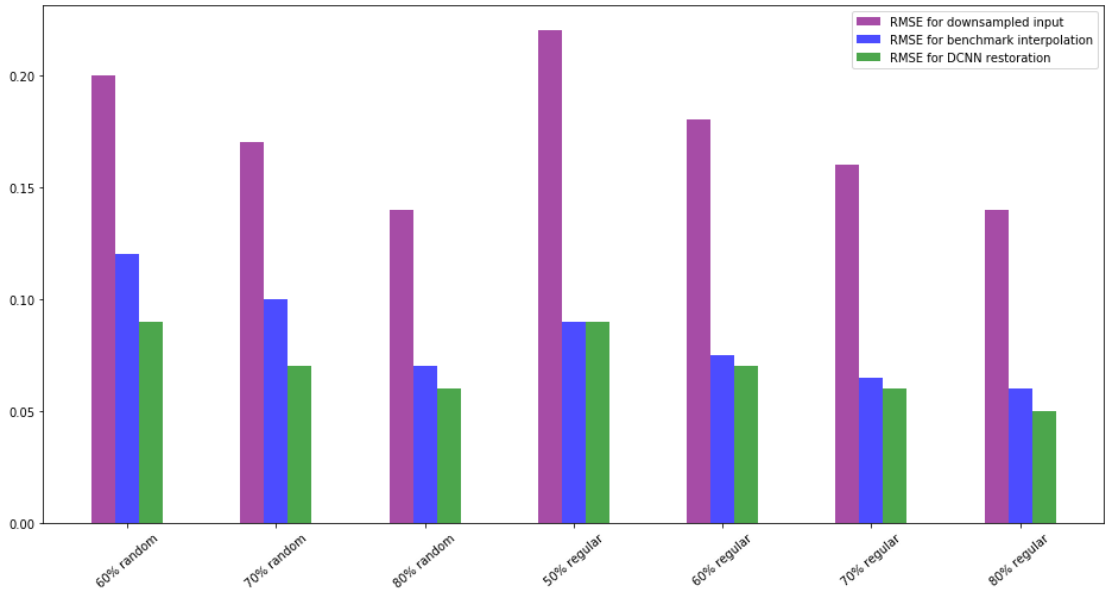


Figure 7: Comparison of RMSE of the inference at the end of the training with the RMSE of benchmark interpolation. The lower the bar, the closer the image to its ground truth.

Results

Model Validation

To validate the trained model, I used a different dataset from the same distribution, this time the 2D slices were of size 150×300. I selected the most difficult to interpolate images with dipping events and I applied the harshest random downsampling, 50%, which makes interpolation very ambiguous. During training, the harshest downsampling was random 40%, i.e. 60% of traces were kept. For benchmark, I used SciPy 2D griddata interpolation module. Figures 8-13 present examples of inference with the trained model; in all cases, the DCNN model restores data better than the benchmark.

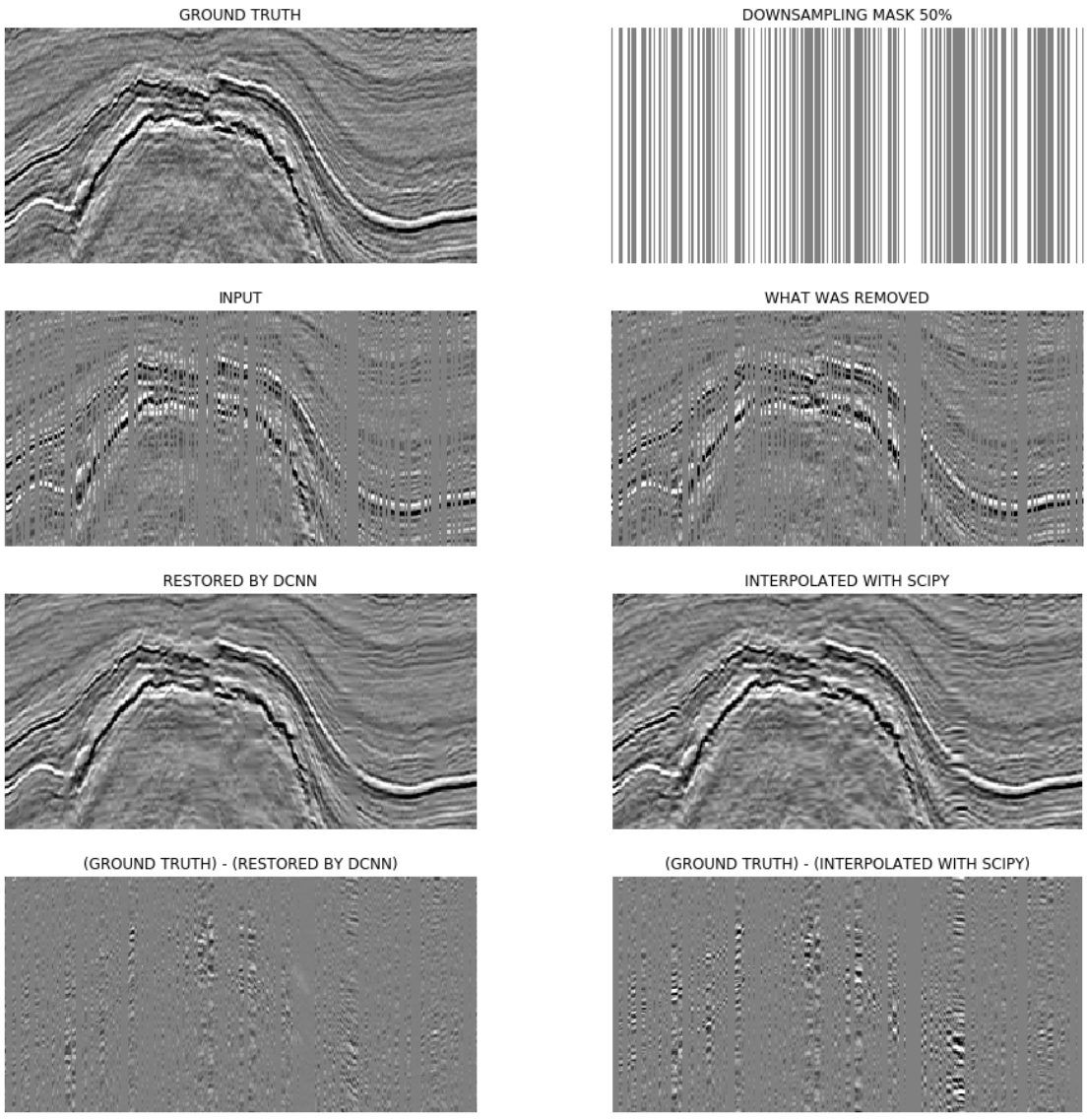


Figure 8: Example 1 of the inference of missing data by the model and a benchmark interpolation

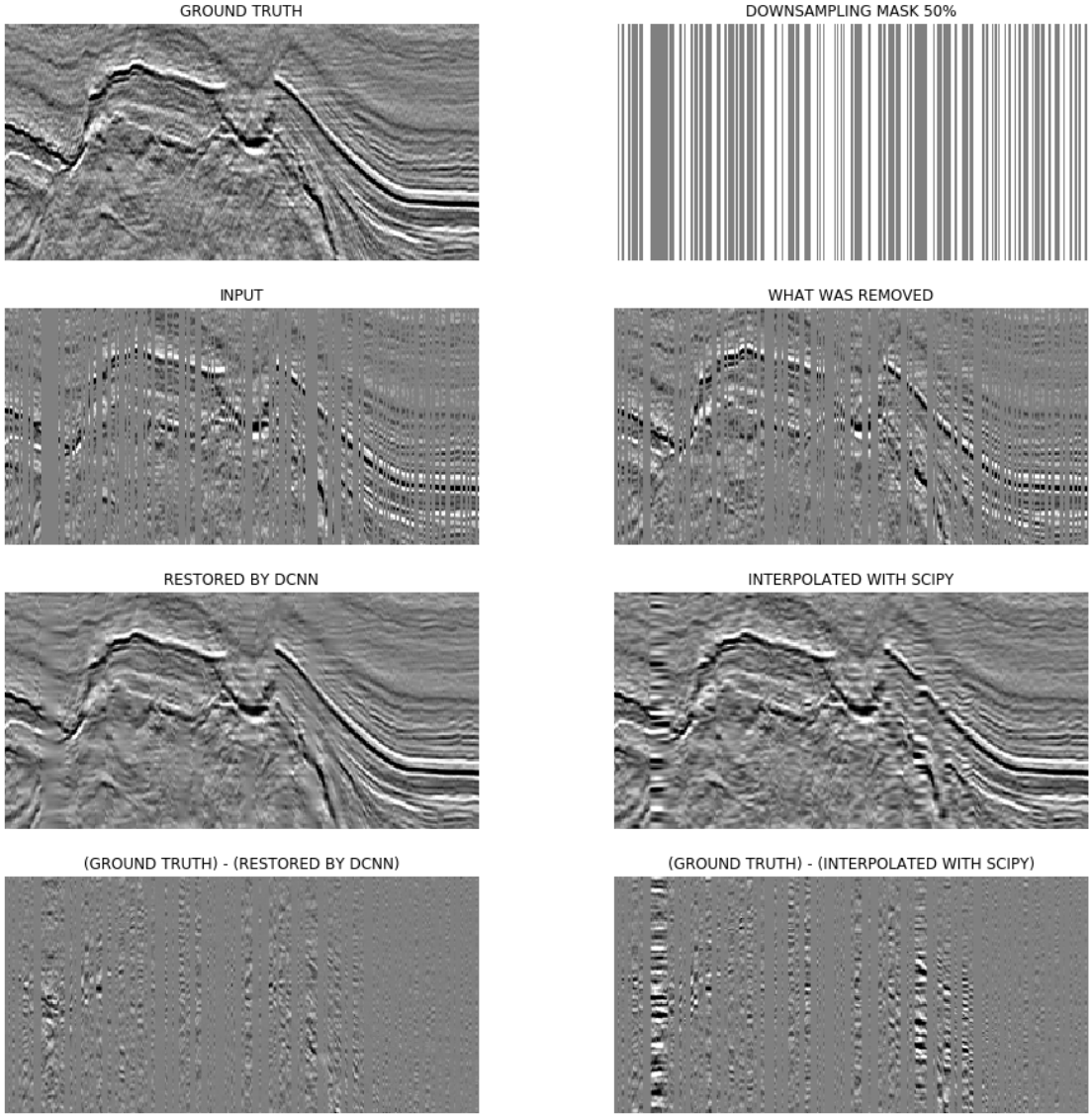


Figure 9: Example 2 of the inference of missing data by the model and a benchmark interpolation

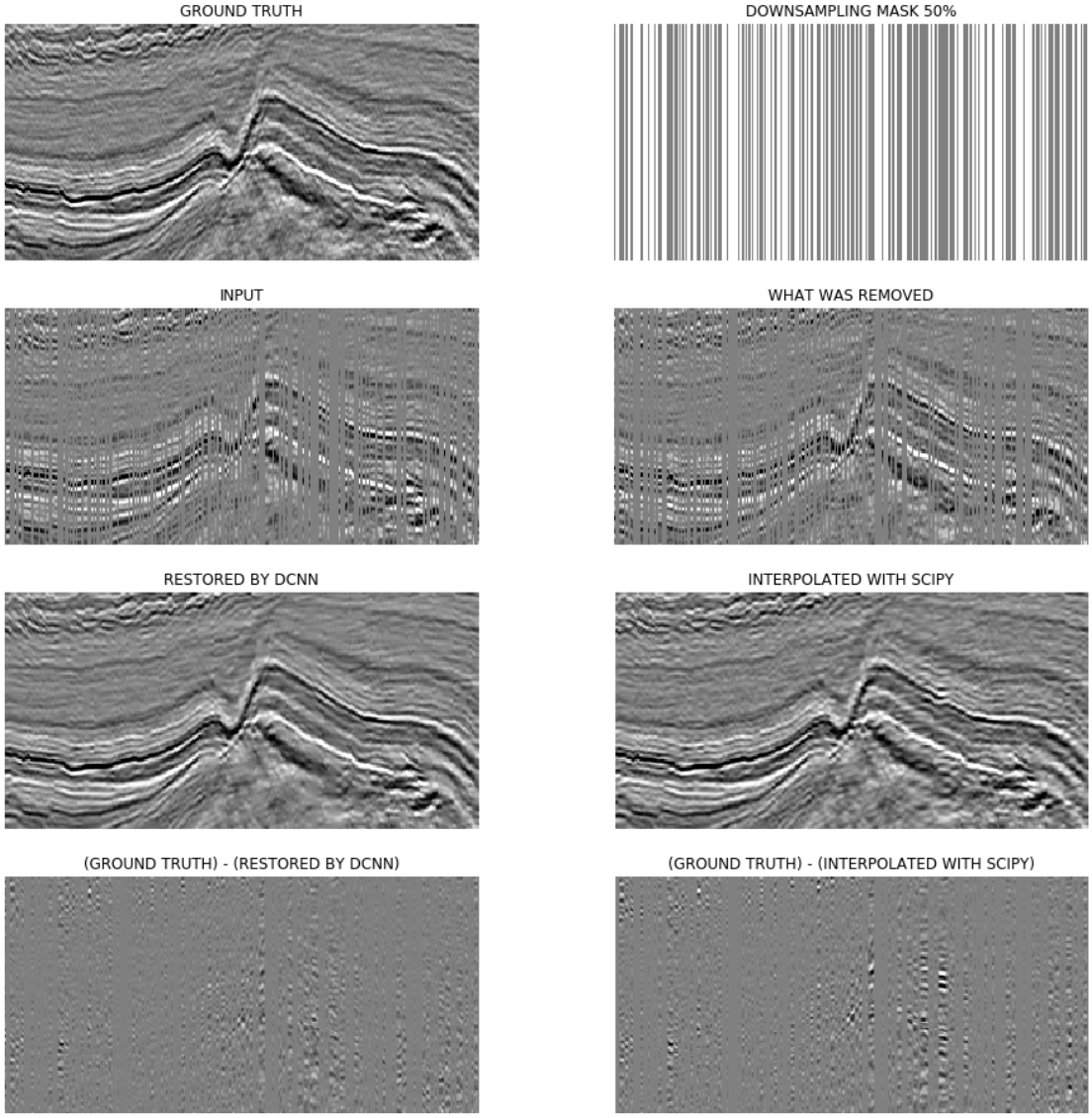


Figure 10: Example 3 of the inference of missing data by the model and a benchmark interpolation

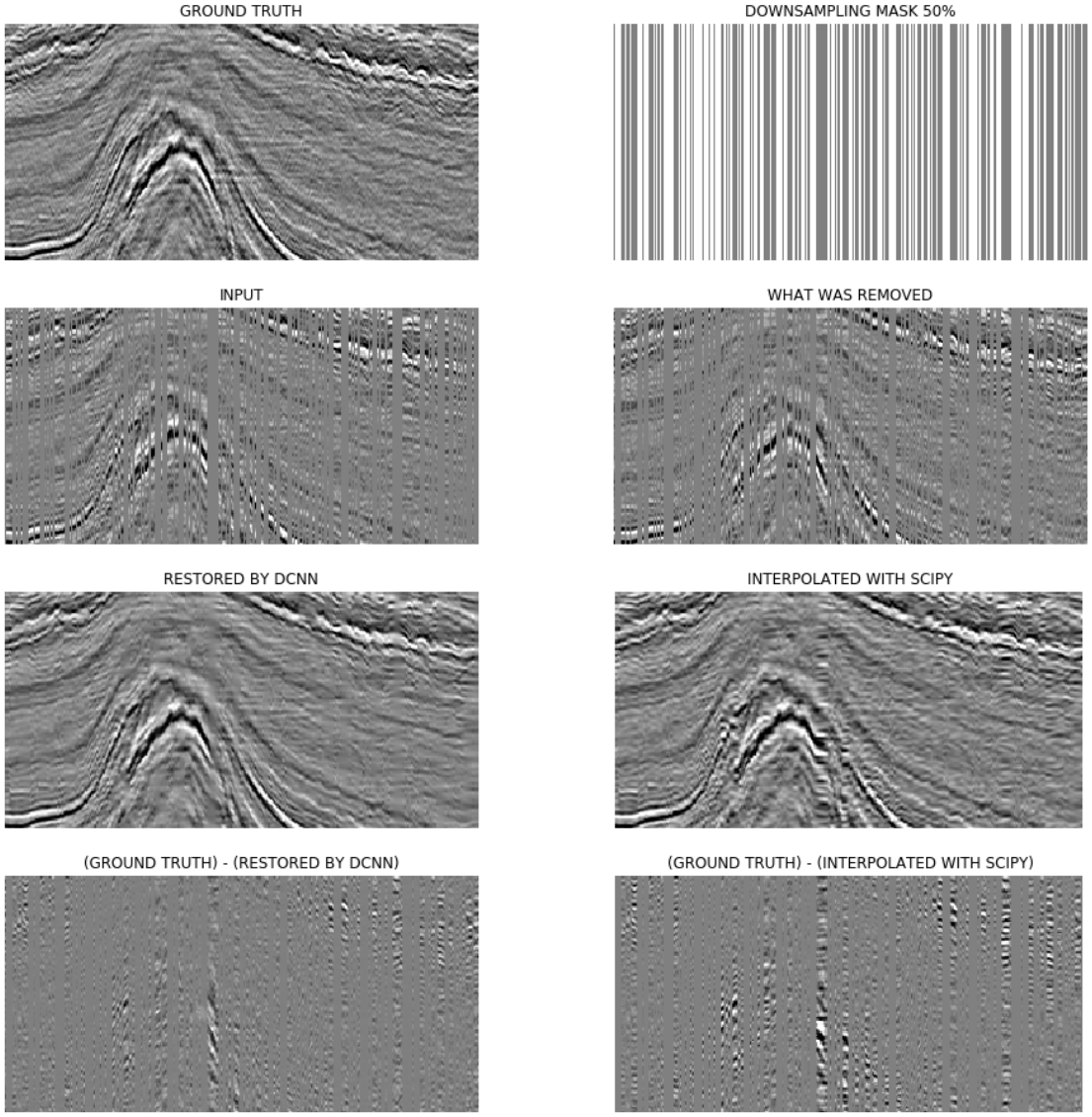


Figure 11: Example 4 of the inference of missing data by the model and a benchmark interpolation

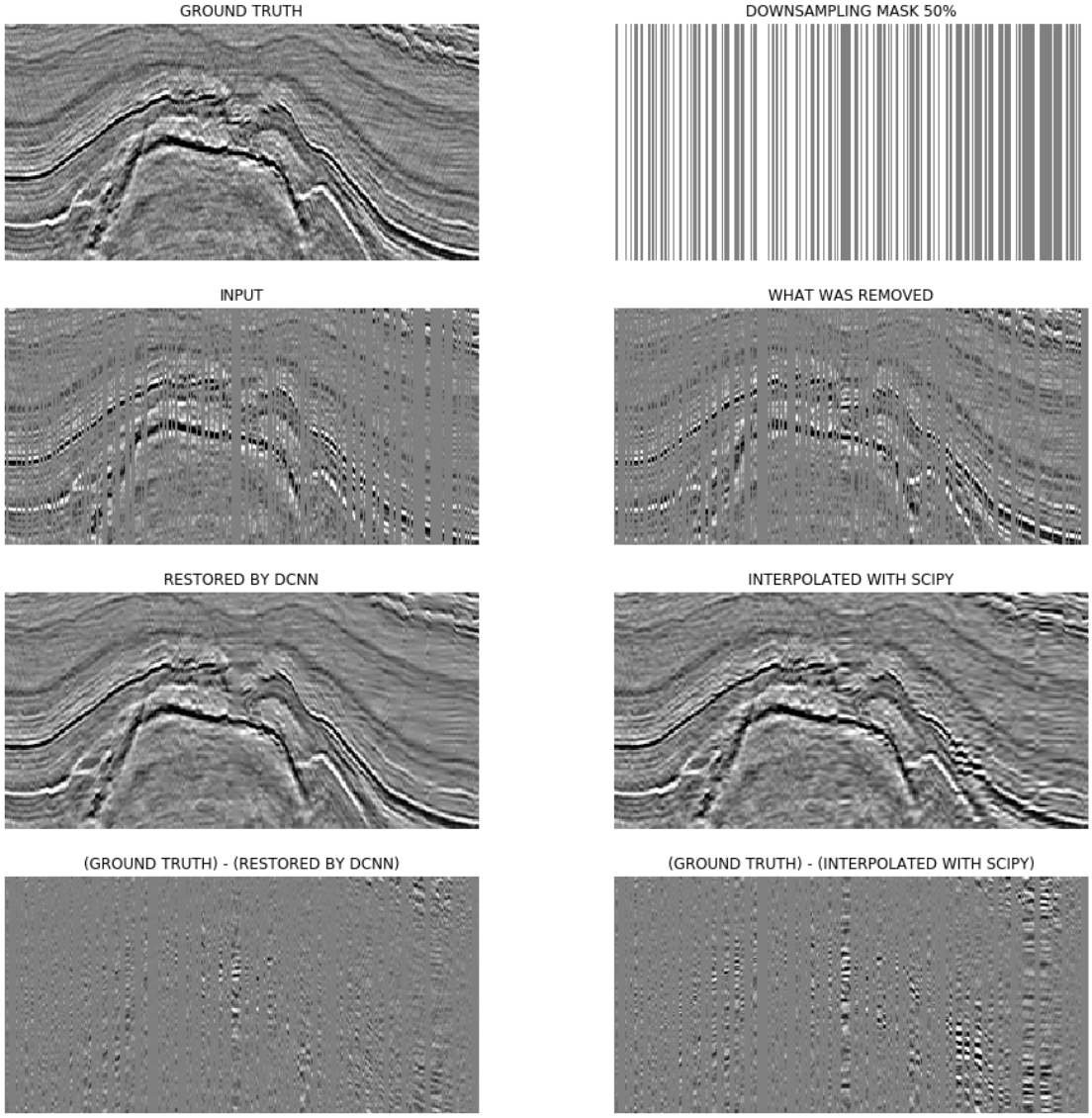


Figure 12: Example 5 of the inference of missing data by the model and a benchmark interpolation

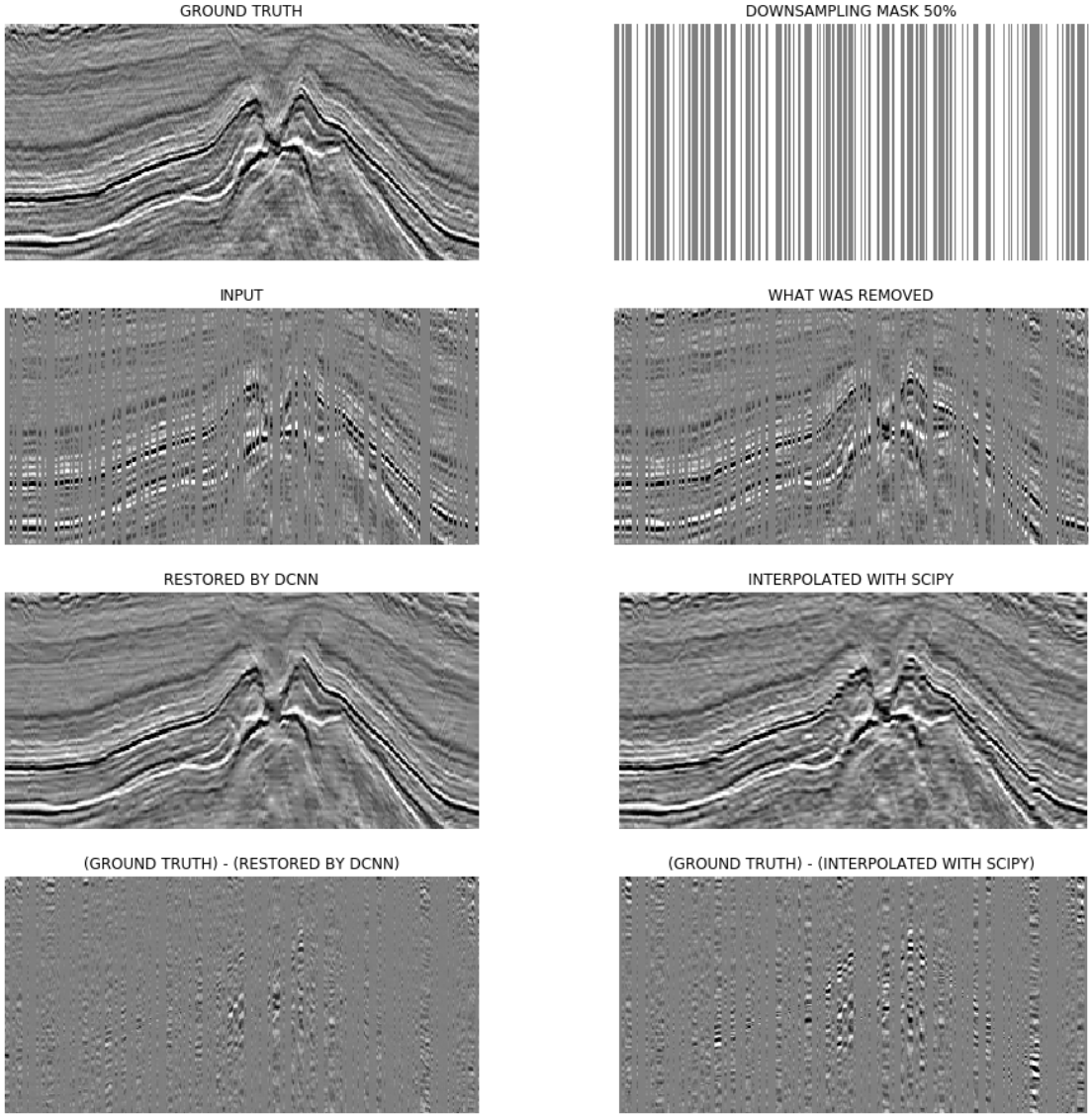


Figure 13: Example 6 of the inference of missing data by the model and a benchmark interpolation

Justification

In this study, I've applied a Deep Convolutional Neural Network that is developed for denoising and upsampling images to data that are not images. More over, by simply zeroing values of dropped traces, I've probably selected not the best representation for missing data, because zero is a valid value for seismic data. In fact, for the SciPy interpolation, I had to supply a mask to indicate invalid data. I am surprised that my DCNN managed to figure out by itself which zeros were valid, and which had to be filled, and did the interpolation better than the SciPy benchmark. The trained DCNN is also robust with respect to a bigger input with a substantial amount of missing traces and big gaps. I also like the simplicity of this DCNN architecture. It's quite obvious how to adapt it to volumetric data: increase a third dimension of the first layer's

filters.

References

K. Zhang, W. Zuo, Y. Chen et al., "Beyond a Gaussian denoiser: residual learning of deep Cnn for image denoising", IEEE Trans. Image Process., vol. 26, no. 7, pp. 3142-3155, 2017.

S. Ioffe, C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", Proc. Int. Conf. Mach. Learn., pp. 448-456, 2015.

LaTeX