Imperial College London

Department of Earth Science and Engineering

MSc Applied Computational Science and Engineering

Module ACSE9

MSc Individual Research Project

Neural Networks Applied to Signal Separation on Multi-Sensor Array Data

Internal Supervisors:
Prof. Olivier Dubrule
Dr. Lukas Mosser

Author:
Mattia Guerri

Author email, GitHub Alias:
mattia.guerri18@imperial.ac.uk
mattiaguerri

Company Supervisors:
Dr. Song Hou
Dr. Henning Hoeber

Company Name, Address:
CGG
Crompton Way, Crawley, UK

August 30, 2019

Abstract

In the oil industry, seismic surveys have been used for decades to investigate the structure of sedimentary basins and localise potential hydrocarbon reservoirs. In marine environments, seismic surveys can be conducted with ships towing airguns, generating mechanical waves, and receivers, recording the waves reflected by geological structures. In order to obtain better data quality and save on acquisition costs, the multi-shooting technique consists in firing multiple sources simultaneously in the same acquisition area. The data so obtained is characterised by interferences between signals coming from different sources. In order to be properly processed, the signals generated by the sources have to be untangled, an operation known as signal de-blending. This is conducted with algorithms requiring long computational time and massive hardware resources. The goal of the project is to develop and test artificial neural networks (ANNs) capable of performing good quality de-blending, while requiring a fraction of the computational time needed by standard algorithms. I developed a number of ANNs, featuring an encoding and decoding sides, and trained them to perform de-blending using the outputs of standard algorithms as training examples. Having been able to obtain networks capable of producing good quality results, I compared their outputs with those computed with standard algorithms, highlighting the limitations of the networks results and offering possible solutions. I thoroughly explored how the ANNs performance is influenced by various kinds of architectural modifications. In addition, I carried out an analysis of how the size and features of the training dataset impacts the performance of the networks. The results of the project show the feasibility of employing ANNs in performing signal de-blending, with a significant saving in terms of computational time, from dozens to only a couple of hours, for the processing of one seismic acquisition line.

Acknowledgements

I would like to thank all the teaching staff of the MSc in Applied Computational Science and Engineering. I thoroughly enjoyed my time at Imperial. All the Master modules were fascinating, with the perfect blending between theoretical background and applications. In particular, I would like to thank Olivier Dubrule and Lukas Mosser for the exciting course in Machine Learning and for supervising this research project. I also need to thank all the students in my cohort. Working with them, specially during the team projects, helped in consolidating the skills in mathematical modelling and scientific computing learned throughout the Master. Working at CGG have been a great experience. I thank all the CGG staff that helped me through the internship, in particular Song Hou for the constant support and stimulating discussions, and also Henning Hoeber, Ewa Kaszycka, Sharon Howe, Damiana Montanino, Krzysztof Cichy, Ula Maka, Alex Clowes, and Ali Hariri. In addition, I thank CGG MCNV for making available the data used in the project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Seismic surveys are a tool of primary importance in hydrocarbon exploration [SG95]. In marine environments, a common method adopted to acquire data consists in a ship towing a number of airguns and cables. Attached to each cable there is an array of instruments, the receivers, capable of detecting the passage of mechanical waves. The airguns (also known as sources) are fired to produce waves that propagates in the seawater and beyond the sea bottom into the subsurface. Downwards propagating waves are reflected upwards when they encounter discontinuities in physical properties, in particular density and velocity, of the medium they are propagating in. Reflected waves cause pressure variations that are recorded by the receivers towed by the ship. The receiver recording are gathered in datasets. These are processed in order to obtain physical representations of the subsurface. The models are interpreted from a geological point of view, identifying for example, the geological structures and, in favourable cases, rock lithologies and other parameters like porosity. The final goal is to identify geological systems, which in the industry are indicated as plays and prospects, suitable for generating and storing hydrocarbon resources. Further details about marine seismic data acquisition and processing will be given in the Methodology chapter.

1.2 Rationale and Goal

A marine data acquisition for commercial purposes can span an area of thousands of square kilometers. In order to acquire data, the ship swipes the area several times following rectilinear trajectories until the entire area has been covered. While moving along one of the trajectories, usually called simply lines or sequences, the airguns carried by the ship are fired and the reflected waves recorded. The acquired dataset is thus subdivided in subgroups, one for each line. The airguns are usually fired sequentially, with a certain waiting time between the fires. The waiting is necessary for the vanishing of all the energy generated by one explosion. In an alternative approach, known as multishooting, the waiting time between the fires is reduced. This results in a faster, hence cheaper, acquisition. Another example of multishooting consists in multiple ships firing their airguns simultaneously in the same acquisition area, aiming to obtain a dataset of improved quality. Combination of the two approaches, reduced waiting time and multiple sources fired contemporaneously, is also possible, combining the benefits, faster acquisition and better data quality, respectively. The downside of multishooting is that the recorded signals are generated by the interferences of waves coming from multiple sources. The blended datasets, characterized by these interferences, require additional processing procedures with respect to the not blended ones. Such procedures, aiming to isolate the signals generated by the single sources, a task known as signal deblending, require enormous computational time and resources. The goal of the project is to explore the possibility of using artificial neural networks to perform signal deblending. The idea is to train the networks on the results obtained with the standard deblending procedures already employed by CGG. Several different algorithms are involved in the deblending, for simplicity they are internally defined just as G2, and thus I will indicate them throughout this report. Training neural networks is also a computationally demanding task, however, once the training is completed, the inference stage, which consists in using the trained network to obtain new results, is a relatively fast procedure. The possibility of obtaining a signal separation algorithm, more efficient than those already available, is of great interest to CGG. It would allow indeed to save time and computational resources, which translates in more resources available for other tasks, and a faster delivery of the products to the clients.

1.3 Artificial Neural Networks

Artificial neural network algorithms are so called because the first researchers experimenting with them were loosely inspired by the functioning of the networks of neurons constituting our brains (see for example the McCullock-Pitts Neuron [MP88], originally published in 1943). The fundamental component of an artificial neural network is the neuron, which is essentially an element storing a value. Neurons are organized in layers and these are stacked up in sequences. The number of layers defines the depth of a network. The deeper the network, the more layers are involved in it. Each neuron has connections with either all or only a certain portion of the neurons in the previous and in the next layers. One connection has a parameter attached to it, a weight, which scales its importance in defining the output of the network. Artificial neural networks are the backbone of many machine learning algorithms. In machine learning we recognize two different kinds of task, defined as supervised and unsupervised learning [LBH15]. In supervised learning the algorithm is trained to perform a certain task through exposure to examples. In unsupervised learning instead, the algorithm is supposed to automatically extract insight from the data. In this project I will focus on supervised learning. Today, machine learning algorithms relying on neural networks are successfully applied in many fields, for example, image processing, autonomous driving, video games, trading, drugs engineering, manufacturing optimization, and many others [LBH15]. The seismic dataset involved in the project can be organised in bi-dimensional arrays of numbers, exactly like images. More interesting for the project are therefore existing applications of neural networks to the field of image analysis. What I want to accomplish shares similarities with a particular image analysis task known as image semantic segmentation [ZMCL15] [Tho16]. It consists in having an algorithm capable of classifying all the pixels of an image. For a example, given the picture of an urban environment, the algorithm has to be able to identify whether a set of pixels represents a car, or the sidewalk, a person, and so on. Such a task has many applications, some of them wielding strong economical impact. One example is represented by autonomous driving technologies. Image segmentation is therefore the focus of vigorous research efforts, both in academia and industry, and many neural networks have been developed to deal with it, either from scratch or adapting already established architecture. Some examples are VGG16 [SZ14], GoogLeNet [SLJ$^+$15], ParseNet [LRB15], and many others (see [GOO$^+$17] for a review about the topic). Most of the networks I developed during the project are based upon the UNet architecture presented in [RPB15]. They are formed by two parts, one producing a down-sample of the image, and the second one up-sampling the output of the first one. I will discuss in details the architecture and the components of the developed networks in the next chapter.

Chapter 2

Methodology

2.1 Neural Networks Development

2.1.1 Training Examples

In supervised learning a neural network is used to learn the function mapping an input into an output. This is achieved by exposing the network to input/output couples of data, with a couple indicated as a training example. The output in a training example is called the target, highlighting the fact that the neural network has to learn how to reproduce it. As described in the previous chapter, neural networks consist in sets of neurons structured in layers. When applied to image analysis, the first layer of the network, the input layer, is represented by the image to be processed. Mathematically, the image is a tensor of second or third order. In case of a one channel image we only need a second order tensor, with the two dimensions representing image width and height, respectively. If the image has more then one channel (like an RGB picture, having three channels), we require a third order tensor, the first dimension indicating the depth of the image, or the number of channels. In machine learning, the values that are fed into the network are known as features. Since images are formed by a set a bi-dimensional features, one for each channel, the channels are also known as feature maps. In the training examples I used in the project, the input is represented by a third order tensor, with dimensions of size 3X636x1251. As said above, the first dimension gives the number of feature maps. The first feature maps is given by the recordings of the receivers. The columns are essentially time series, with the values (1251 for each column) proportional to the amplitude of the recorded signal. The rows are therefore values recorded by different receivers at the same time. In the second feature map each column indicates time, starting from the firing of the source. The third feature maps is given by rows containing the location of the receiver on the cable. All the columns in the second feature map, and all the columns in the third, have the same values. The addition of these maps, which essentially give space-time coordinates, helps the network in identifying the signals generated by different sources. The targets and the outputs of the networks are second order tensors containing the deblended signals. Examples of input, target and network output are showed in the next chapter.

2.1.2 Forward and Backpropagation

Training a neural network consists in finding the optimal values for the weights scaling the connections between neurons. The goal is to have the output of the network as close as possible to the targets supplied in the training examples. The training involves two stages, the forward propagation and the backpropagation. In the forward propagation the input image is fed into the network, various mathematical operations (described in the following sections) process the input to generate the network output. The output is then compared with the target through a function indicated as loss function (also misfit, cost). The goal of the training is to minimise the discrepancy between the output and the target, hence the value of the loss function. Training a neural network is therefore an optimisation problem. To solve it, gradient based methods are primarily used, which in their simplest form are represented by gradient descent. Gradient methods require the computation of the first order partial derivatives of the loss function with respect to each network weight. The partial derivatives are obtained starting from the weights of the neurons in the layer closer to the output and then going backward through the network using the chain rule. This operation is known as backpropagation, first introduced by [RHW86]. Once the partial derivatives are computed, a single weight θ is updated with the gradient descent formula:

$$\theta = \theta - \alpha \frac{\partial L(\mathbf{X}, \mathbf{Y})}{\partial \theta}, \tag{2.1}$$

where $L(\mathbf{X}, \mathbf{Y})$ is the loss function, $\mathbf{X}$ and $\mathbf{Y}$, the network output and the target, are tensors of arbitrary shape. The parameter α scales the step taken in the (negative) direction of the gradient in order to modify the weight. Forward and backpropagation are repeated for many iterations until convergence is reached. As a convergence criterion, one can assume that convergence is reached if after a certain number of iterations the absolute value of the loss hasn't changed beyond a user set threshold. Alternatively, one can train the algorithm for a fixed number of iterations. After each iteration, the present value of the loss is compared with the previous one. If the present value is lower, the model obtained is saved, otherwise discarded.

2.1.3 Training and Validation

When training a neural network it is common practice to use two distinct sets of data, the training and the validation set. The training set is used in the actual training. The weights of the network are modified in order to minimise the difference between the network output and the targets contained in this set of data. In doing one step of gradient descent optimisation one can take into account the whole training set (batch gradient descent), a portion of it (stochastic gradient descent). The first approach is rarely used nowadays, given that the entire dataset would not fit in the memory of the machine. A portion of the training set is usually considered. The number of training examples used in one optimisation step is called the batch size. One complete iteration of the training procedure, with the entire dataset being used in the optimisation, is called epoch. In stocahstic gradient descent, one epoch is actually constituted by several optimisation steps, one for each portion of training examples, the number of which is given by the batch size. The performance of the network is monitored using the validation set. Since the targets contained in the validation set are not used in the training procedure, verifying how the network is able to reproduce them gives us an indication about its ability to be applied to a population of unseen data in the inference stage. Is is important to reduce as much as possible the loss with respect to the training set, while avoiding the model to become too specialised in reproducing the training set examples, an issue known as overfitting. In addition, the validation set is used to guide the choice of a set of networks features and parameters, known as hyperparameters, which affect the performance of the network but remain constant during training.

2.1.4 Architectures

In order to mimic the deblending performed by G2, the network needs to produce an output tensor that has the same width and height as the target, which are also the same of the input image. Previous networks tested at CGG consisted in a series of convolutional layers that kept unaltered the width and height of the input feature maps. Experiments adopting residual learning blocks (which will be explained in chapter 4) were also performed. These models were able to produce encouraging results, but the differences between their output and targets supplied by G2 were still significant. During the project, I developed architectures inspired by the UNet algorithm presented in [RPB15]. These networks are characterised by an encoder and a decoder joined together. The encoder performs a down-sample of the input, while increasing the number of feature maps. Concretely, while the input image has dimensions of 3x636x1251, the output of the encoder can be a 3700x1x2 tensor (in case of the deeper models developed). The decoder has the opposite effect, taking in the 3700x1x2 tensor and up-sampling it to the original size, generating an output directly comparable with the target. Both the encoder and the decoder are constituted by blocks of two or more convolutions. At the end of each block there is either an operation of pooling (down-sampling block) or a transposed convolution (up-sampling block). An explanation of all the numerical operation defining the network will follow in the next section. A number of structural variations of the UNet architecture have been explored, changing for example the number of down-sampling blocks and the numbers of convolutions involved in each block. The outcome of these experiments is presented in the next chapter.

2.1.5 Forward Propagation

Here is a list and description of the operations involved in the networks forward propagation.

Convolution

This operation is routinely used in neural networks designed to process images. It consists in processing an input with a set of filters. The input is, in case of the first layer, one of the images in the training set. For the successive layers, the input is represented by the tensor generated by the previous layer. Each filter, also called kernel, is defined by a certain height and width. Its depth equals the depth of the input tensor. Height and width of the filter are usually smaller than those of the input to be processed. The filter is thus applied on the entire input tensor in successive stages, after each stage the filter is moved vertically and laterally by an amount defined as stride. The convolution consist in element-wise multiplication of the elements of the input tensor with the element of the filters, followed by summation. It therefore results in a scalar. The elements of the filters are the weights that are modified during the network training. It is useful to notice, also for the understanding of transposed convolution (explained in the following), that a convolution can be obtained as a matrix vector inner product (see Figure 2.1). Let us for example assume that the input tensor only has one channel (depth equal to 1). In this case, the elements of the matrix would be zeros and the weights of the filter, properly arranged. The vector would be instead obtained by unrolling the input. The inner product of the matrix and the vector so obtained, a vector, after being properly reshaped, is the output of the convolution.

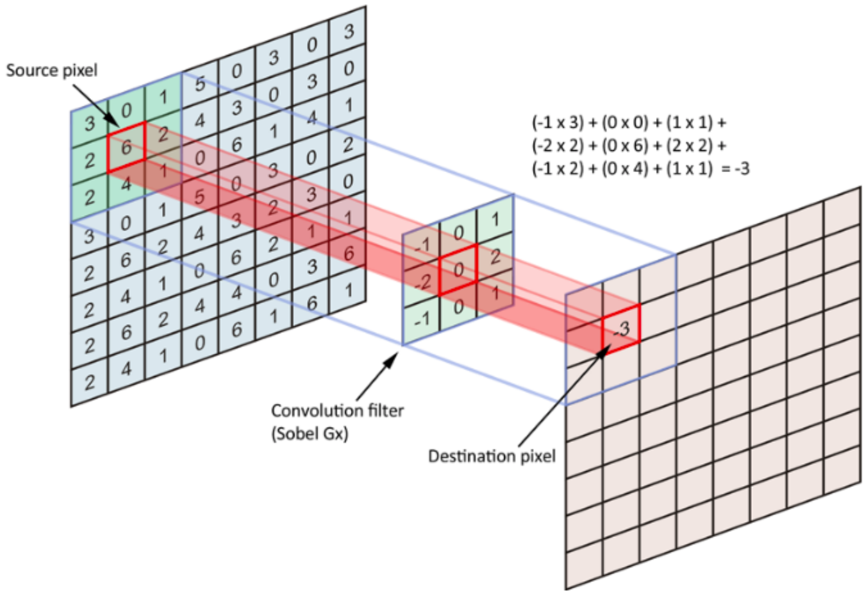
Source pixel
(-1 x 3) + (0 x 0) + (1 x 1) +
(-2 x 2) + (0 x 6) + (2 x 2) +
(-1 x 2) + (0 x 4) + (1 x 1) = -3
Convolution filter
(Sobel Gx)
Destination pixel

Figure 2.1: The convolution operation. The kernel (in this case a Sobel filter), is applied to the input image. The convolution consists in an element-wise multiplication between input image pixels values and kernel weights. Source: www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/

ReLU activation function

The output of the convolution is not directly used as an input feature for the next layer. It is instead fed into the so-called activation function, which is non-linear. There are many kinds of activation function available. A standard choice is represented by the ReLU (Rectified Linear Unit), which is simply defined by $x = max(0, x)$.

Max Pooling

Max pooling is an operation allowing to down-sample an input tensor. It is therefore placed at the end of each down-sampling block (with the exception of the last one). It consists in processing the input tensor with a bi-dimensional filter of arbitrary size. The filter size is usually smaller then the input size. So the filter is applied in various stages until all the input has been processed. When the filter is applied to a set of elements of the input, the output is represented by only one value, the maximum among those in the input set.

Transposed convolution

Transpose convolution is the inverse operation of the convolution described above. It can be used to obtain an output with a height and width larger then those of the input. As for convolution, we can see this operation as a matrix-vector multiplication. Going back to our previous example used in explaining the convolution, if we wanted to use transpose convolution to to get an output of the same size of the matrix used as an input in the convolution, we could just use the transposed of the matrix used in the convolution, and multiply it to the output of the convolution (see Figure 2.1).

2.1.6 Loss Function and Optimisation Algorithm

For the majority of the experiments I adopted a loss function consisting in the MSE (Mean Squared Error) between the network output, obtained starting from a certain input, and the target coupled to the input in the training example. The MSE loss function is given by:

$$MSE = \frac{1}{B}\sum_{b=1}^{B}\sum_{n=1}^{N}\frac{(\mathbf{x}_n^b - \mathbf{y}_n^b)^2}{N}, \tag{2.2}$$

where $\mathbf{x}$ and $\mathbf{y}$ are two tensors of arbitrary shape, the output and the target, containing a total of N elements each. B indicates the batch size. The optimisation technique I adopted the most throughout the project is the Adam optimiser [KB14]. It is based on gradient descent with momentum and RMSProp. The three algorithms, gradient descent with momentum, RMSProp and Adam are routinely used in machine learning. The Adam optimizer allows to have a dynamic learning rate.

2.1.7 Hyperparameters Tuning

In addition to its weights, the performance of a network is influenced by many other parameters that are kept constant during the training. The validation set is also used to explore the impact of these parameters on the model performance, and to tune them in order to achieve the best possible accuracy. This operation is known as hyperparameters tuning. Some examples of hyperparameters are: i) learning rate, which scales how much the weights are changed after each optimisation step; ii) optimisation method, many are available, all of them essentially based on gradient descent; iii) batch size, how many training examples are used in each optimisation step; iv) depth of the network; iv) number of kernels involved in each convolutions and size of the kernels.

2.2 Software Implementation

2.2.1 Hardware Specifications

Before describing how the networks have been implemented in code, it is useful to give some details about the hardware employed in the experiments. The CGG datacenter is constituted by several hundreds of computer nodes. Each node has its own CPUs (Central Processing Units, usually two per node), memory, and a number of GPUs (Graphic Processing Units, four per node) cards. In training neural networks, the possibility to leverage on GPUs is particularly important. The mathematical operations that are performed during training mostly consist of matrix/matrix multiplications. These operations are easily parallelised. This means that the workload necessary to perform the operations can be split in several threads and run in parallel by several different computational units (cores). Training a neural network on a GPU is not a necessity, it can be accomplished on a CPU as well. The advantage of using a GPU lies on the fact the this consists of thousands of cores. A CPU instead only has at most few dozens of cores. Even if the CPU cores are capable of more operations per seconds with respect to GPU cores, spreading the workload on thousands of parallel threads makes training neural networks on a GPU order of magnitudes faster then training on a CPU.

2.2.2 Pipeline Components

The developed code is intended as an extension of a software pipeline developed by CGG engineers. Data input and ouptut (I/O) is handled by a set of routines interacting with CGG datasets. As the training progresses, a number of training examples, defined by the batch size, is loaded into the node memory and the copied into the GPU memory. The whole training set is not directly loaded into the memory node for two reasons: i) the images in the training examples are large, hence the training set might not fit entirely in the node memory (usually 128 GB); ii) in a production environment, a parsimonious usage of the node memory is favoured, this to allow multiple processes, launched by one and/or more users, to run on the same node. Network training and validation is implemented by a series of scripts interacting with the data I/O routines, loading the model defining the network to be trained, and performing the necessary number of training iterations. An additional group of scripts is used in the inference stage, which consist in deploying the trained model to perform signal deblending.

2.2.3 Implementation of Networks Constituents

The code allowing the user to build a UNet architecture is contained in a Python script named UNet.py. To build the network I use a number of classes imported from the PyTorch package [PGC+17]. The classes allow to set up the structural elements of the networks and perform the mathematical operations described in section 2.1.5. To build the network architecture I developed three classes:

- DownBlock: this class implements one block constituting the encoder of the network. The block consists in two convolutions (for our reference architectures, blocks formed by more convolutions have also been tested) and a maxpooling operation. Each convolution is followed by the activation function. After the two convolutions there is a pooling operation responsible for the down-sampling of the tensor.
- UpBlock: this class is used to create the blocks forming the decoder. There are two convolutions (as above, also more convolutions have been tested), each one followed by the activation function. At the end of the block a transposed convolution up-samples the tensor.
- UNet: this class makes use of the previous two classes to build the network. As input it requires the number of channels of the input tensor, the number of channels of the output tensor, a list of integers indicating the number of kernels used by the convolutions in each block, the input tensor width and height, and a boolean variable indicating whether to use residual learning blocks or not. The list of numbers relative to the convolutions kernels also determines the number of blocks constituting the encoder and, consequently, the number of blocks forming the decoder. For example, if nine integers are listed, the encoder will consist of nine blocks and the decoder of eight. The user is therefore free to change the depth and/or the number of parameters constituting the network. The number of blocks that is possible to use is limited. This is because at the end of each block there is a pooling operation halving the width and height of the tensor, which have to be integers equals or larger the one. The maximum number of blocks is therefore defined by the size of the input image. This can be increased using different kinds of padding. With an input image width and height of 636x1251, the maximum number of encoding blocks is 11. This is the depth of the deeper models tested in the project.

The network is implemented in modules in order to allow the user to choose the model depth. As I will show in the next next chapter, the model depth impacts its accuracy the amount of time necessary for training. Other UNet implementations, featuring additional structural blocks, placed in front of the enocder and/or after the decpoder have been tested. Since the effects of these variations on the accuracy reached by the network was not particularly strong, I decided to not include them in the UNet.py file keep them in separated scripts. Another reason to keep the implementation of the networks separated is that, in a production stage, the machine learning pipeline is also used by workers not trained in machine learning, thus the input required from the user is kept simple and to a minimum.

2.2.4 Software Parallelisation

As described above, parallelisation is fundamental in machine learning workflows in order to train the large models that are usually adopted today. The code runs on a single thread on the CPU but the workload is then spread in multiple threads in the GPU, in a shared memory paradigm, adopting the CUDA platform. The code can also run on multiple GPUs leveraging on the PyTorch class DataParallel. The model is still copied on each GPU, but the batch is spread among them. This is useful because it allows to test the effects of larger batch sizes, and it speeds up the training procedure.

2.2.5 Code Metadata

The entire machine learning pipeline (except data I/O routines), including the codes I developed during the project, is written in Python 3.6. As detailed above, I made extensive use of PyTorch classes. PyTorch is an open source platform available at https://github.com/pytorch/pytorch. Detailed documentation of the PyTorch classes and methods is available at
https://pytorch.org/docs/stable/index.html. The PyTorch version used is 1.0.1.post2. The code has been developed and run in a Linux environment. Concerning the hardware, the requirements are strongly affected by the depth of the models, number of convolutional kernels, batch-size and dimension of single input. Having at least one GPU with 12 GB of memory is recommended. The machine learning pipeline is part of CGG proprietary software and cannot be shared. The codes I developed are available at https://github.com/msc-acse/acse-9-independent-research-project-mattiaguerri.

Chapter 3

Results

3.1 Signal De-blending with Neural Networks

I first defined the reference training and validation sets. For the training set, I collected data from two seismic lines, which are positioned close to each other in southern portion of the acquisition area. The validation set contains data from a third line, which sits roughly in between the two used for training. A schematic representation of the location of these lines and others discussed in the report is showed in Figure 3.1. Training data are form lines 10 and 12, validation data are from line 11.

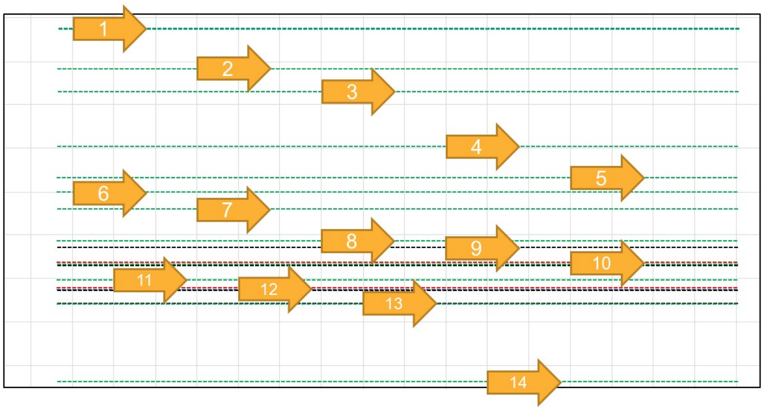

Figure 3.1: Schematic representation of a portion of the acquisition with the lines used for network training and validation. Data from red lines was used to create the reference dataset. Data from black lines was used for Dat4x. Data from green and blue lines was used to create datsets as large as the reference one. See text for more details.

The total amount of training examples (input/target couples) in the reference dataset is 2064. The reference validation set contains instead 344 examples. I also defined a reference model, UNet9, characterised by a U-shaped architecture and 9 down-sampling blocks (8 up-sampling blocks). In Figure 3.2 I show an example of de-blending obtained with G2 and the reference network. For comparison, I also show the result produced by a network previously tested at CGG, (ResNet3) consisting in same padding convolutions (height and width of input and output tensors are kept constant) and residual learning blocks (residual learning will be discussed in the next chapter). The first panel on the left shows an example of the input image. The signal generated by two different sources is evident. One source is responsible for the signal developing from the left edge of the image. The second source generates the signal appearing as a dome-like structure in the center of the image. This signal, usually identified as cross-talk, is the one that has to be removed. In the second panel we have the de-blended image as produced by G2. The cross-talk has been indeed removed. This is the kind of image that is used as target in training the neural networks. The algorithm has to learn the function mapping the first panel into the second one. The third panel is the output generated by ResNet3. The major features of the G2 output are reproduced, but some cross-talk is still visible. The last panel on the right shows the output generated by the reference model (UNet9). In this image the cross-talk appears to have been properly removed.

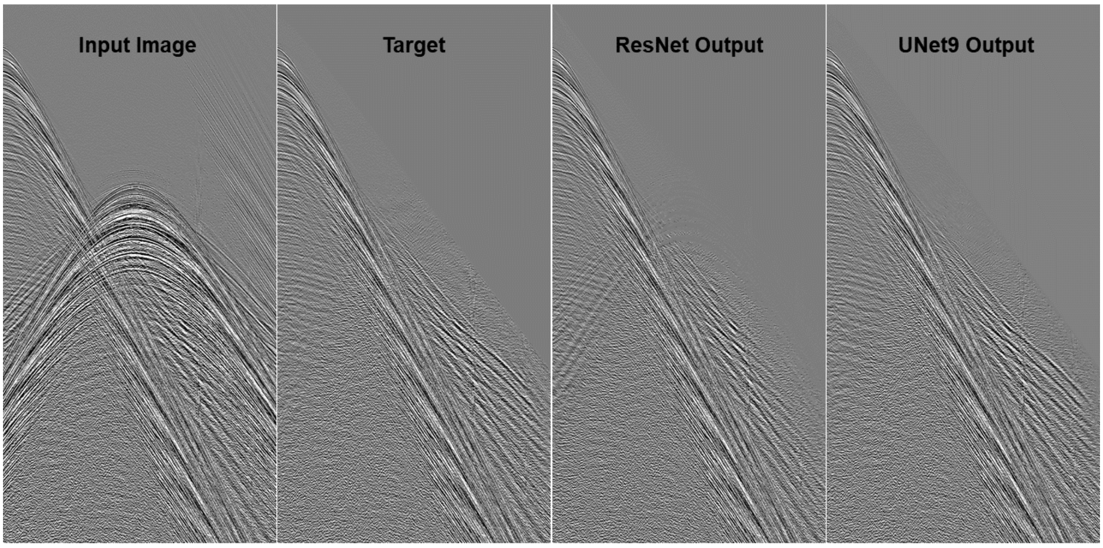

Figure 3.2: First panel shows an example of input image fed into the neural network. Second panel is the deblended signal obtained with G2. This image is used as target to train the neural networks. Third and fourth panels are the outputs of ResNet3 and UNet9, respectively.

Given that the results of UNet9 are of good quality, I used the model to de-blend the signal acquired on various seismic lines. The same model has also been trained on other datasets, again containing 2064 training examples, but constituted by images coming from different lines respect than those used for the reference dataset. In addition, I trained the model on a dataset (Dat4x) that contains roughly four times the number of images contained in the reference dataset, and on another one (DatHalf) containing half the number of training examples. These tests have two motivations. First, evaluating what is the minimum amount of data necessary to obtain a model capable of generating good quality de-blending. This is an important parameter to evaluate because more data needed for training implies longer training time. Second, exploring how the performance of the model changes when applied to seismic lines far from those that have been used for training. Across the acquisition the geology can vary significantly. For example, the depth of the water bottom can range from hundreds to thousands of meters. Also the geometry of subsurface features can vary abruptly, for example in correspondence of faults. The recorded signal is affected by these variations as are the patterns contained in the images utilised for training. It is therefore important to evaluate how a model, trained on data coming from only few lines, is able to properly de-blend signals from seismic lines across the entire acquisition.
The results of these experiments are summarised in Table 3.1 and 3.2. Table 3.1 reports the mean of the differences between the root mean square of the signals contained in the output of G2 and the outputs of UNet9 trained on different datasets. UNet9Ref was trained on the reference dataset, with training examples coming form lines 10 and 12 (red lines on Figure 3.1), which are located close to each other. UNet9_1 was trained on two lines 1 and 12 (blue lines in Figure 3.1), located in the north and the south of the acquisition, respectively. UNet9_2 was trained on data coming from twelve different lines across the acquisitions (green lines in Figure 3.1). All these datasets have the same amount of training examples. In Table 3.2 we can see the validation loss (computed on the reference validation set) reached by a number of models trained on the reference dataset and on the one four times larger (Dat4x, data coming from black lines in Figure 3.1).

Table 3.1: Mean of the differences between G2 output and the output of the reference network trained on various datasets, referring to Figure 3.1, UNetRef was trained on the red lines, UNet_1 om the blue lines, and UNet_2 on the green ones. see text for details.

Model	Acquisition Lines		
	Line 1	Line 11	Line 14
UNet9Ref	2.840	3.343	3.271
UNet9_1	2.241	3.594	3.367
UNet9_2	2.301	3.524	3.147

3.2 Hyperparameters Tuning

3.2.1 Network Architecture

Among the various parameters that remain constant during training, some are related to the network architecture. I explored how the performance of the network responds to various types of structural modifications: i) variation of the number of blocks constituting the encoder and, consequently, the decoder; ii) different number of kernels used in the convolution operations involved in the blocks; iii) increasing the number of convolutional layers in each block (the reference model has two). The results of experiments performed using the reference training set and Dat4x are presented in Table 3.2. The validation error is always computed on the reference validation set. When the reference dataset is used, I trained the models for 400 epochs. In case of Dat4x instead, the models have been trained for 200 epochs. Pushing the training any further would not have been useful because of overfitting. For example, the validation loss for the reference model starts increasing after roughly 140 epochs. In the reference model, in the first block, the number of kernel for convolution is 64. The number of kernels then increase by 50% in each block. Thus, in the second block convolutions have 96 kernels, in the third 144 and so on. The model UNet9Half features convolution with half the number of kernels (32, 48, 72, etc.). Models even smaller then UNet9Half have been tested, but the further reduction of the number of kernels severely impact the the quality of the de-blending. The model UNetDeep3 has three convolutional layers per block. A model with four convolutional has been tested, but it does not obtain an accuracy comparable to the one displayed by the reference model, while requiring longer time for training. The models UNet9Skip and UNet11Skip implement the residual learning block, and UNet9InPlus has an additional block of convolutions characterised by large kernel size. These models will be discussed in the next chapter. The training times in Table 3.2 (format is minutes:seconds) are obtained averaging the time necessary to perform four epochs. I consider taking into account only four epochs sufficient since the time per epochs only varies by few (<10) seconds. All the models have been trained using four NVIDIA Tesla P100 (12GB), with the exception of the deeper ones, UNet9Deep3, UNet11 and UNet11. These were trained on two NVIDIA Quadro RTX 6000 (24GB).

Table 3.2: Minimum validation loss for networks trained on the reference dataset (achieved within 400 epochs) and on a dataset four times larger (achieved within 200 epochs). The training times (format is minutes:seconds) refer to the average time necessary to complete four epochs.

Network	Reference Dataset			Dat4x		
	Train. loss	Val. loss	Train. time	Train. loss	Val. loss	Train. time
UNet7	8.795	10.40	3:37	7.684	10.04	14:04
UNet9	8.989	10.28	6:05	7.299	9.948	18:39
UNet9Half	8.958	10.90	2:10			
UNet9Skip	8.455	10.28	6:53	6.901	9.771	24:23
UNet9InPlus	8.015	10.36	11:17			
UNet9Deep3	9.232	10.42	10:51			
UNet11	8.340	10.32	10:23			
UNet11Skip	6.736	10.18	14:16	6.754	9.788	

3.2.2 Loss Function

As explained in the previous chapter, the training of a neural network is an optimisation problem involving a loss function. All the results presented so far have been obtained with the MSE loss. I coded and tested an additional loss function. It consists in computing the Fourier transform of the time series contained in the G2 and neural network outputs. Each column of the output tensors is a time series, being the recordings of the receivers. I then computed the MSE between the transforms of the signals and sum the results. The value so obtained is the output of the loss function, and it is the one that is minimised in the training process. I trained the reference network employing this loss function. If I compute a simple MSE between this network and G2 outputs, the training and validation error are 8.099 and 10.33, respectively. Note from Table 3.2 that the validation error is the second lowest obtained considering all the models. In Figure 3.3 I show one of the outputs of the model trained on the modified loss function.

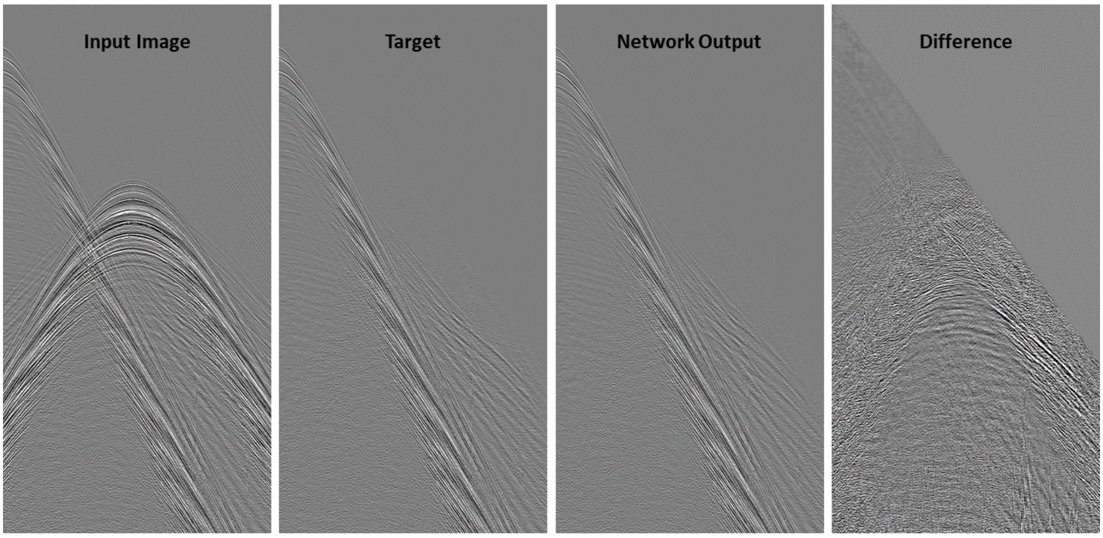

Figure 3.3: Input image, G2 output (target), output of the network trained with the modified loss function and difference between the two. Note that for the difference image the contrast has been strongly enhanced.

Chapter 4

Discussion

4.1 Network Architecture and Performance

I explored how architectural variations impact the performance of U-shaped networks, both in terms of accuracy and computational time. Results are summarised in Table 3.2. Among the simple UNet architectures (UNet7, UNet9, UNet11), the lowest training error is reached by the one featuring the higher number of down-blocks (11). The validation errors achieved by these models are quite similar, within about 1% of UNet9 validation loss. The training time per epoch varies instead significantly, with UNet7 requiring almost half the time required by UNet9. This shows how similar accuracy can be achieved by using simpler models, with less trainable parameters (UNet9 has $\approx 144M$, UNet7 $\approx 28M$). Also decreasing the number of kernels per convolution does not seem to have a strong negative impact on the model accuracy, while significantly affecting the time required for training. In UNet9Half I halved the number of kernels per convolution with respect to UNet9. The validation loss decrease by about 6%, but the training time is almost three times smaller is only. If UNet7 training is pushed further than 400 epochs, the network reaches a minimum validation loss of 10.36 before starting to overfit, at around epoch 430. Such an outcome is interesting considering that, when using U-shaped architectures, it is common practice to double the convolutional kernels in each down-sampling blocks, following what was done in [RPB15]. Here I show how the number of kernels can be reduced, without a significant loss in terms of model accuracy. Is is important to notice that a network constituted by a smaller number of trainable parameters is not only faster to train, but it is also faster in the inference stage. The same portion of data, equal to roughly 3% of an entire acquisition line, is de-blended in 15:53 (minutes:seconds) and 12:40 by UNet9 and UNet7, respectively. The advantage in a production environment is obvious. Adding more convolutional layers per block (three in UNetDeep3), increases the training time without having a positive impact on the validation loss. Adding layers to a network is one of the simpler things that can be attempted in order to improve its accuracy. More layers implies more trainable parameters, and this in turn should lead to better accuracy. However, as I showed above, the effect of such an approach is limited. Increasing the parameters of a networks leads at first to a better accuracy. The effects though saturates and, after a certain threshold is reached, the opposite is observed, the accuracy decreases as the parameters are increased further. This process has been already discussed in various studies, for example [HS14, SGS15, HZRS16]. To overcome the issue, [HZRS16] introduced the concept of residual learning. The assumption is that, if we want a stack of layers to learn a function $F(x)$, it would be easier to learn the difference, the residual, between $F(x)$ and the identity function $I(x) = x$ rather than the function $F(x)$ itself. The stack of layers is brought to learn $g(x) = F(x) - I(x)$. This is achieved by adding the input tensor to the one output by the stack of layers, through a connection that skips the layers in between the input layer and the final one. I have implemented residual learning blocks into the UNet architectures. The accuracy achieved by the models so obtained, UNet9Skip and UNet11Skip, are listed in Table 3.2. The network UNet9Skip performs slightly better then UNet9. If the larger dataset is used for training, the validation loss decreases by almost 2% respect than the value reached by UNet9. Because of an additional convolution per block, these models require more time for training, with respect to their UNet counterparts, and also occupy more space in memory. The count of trainable weights increases from $\approx 144M$ to $\approx 151M$, for UNet9 and UNet9Skip, respectively. UNet11Skip achieves the lowest training and validation errors among all the networks trained on the reference dataset. The model overfits the both the reference and the Dat4x datasets. This indicates that its capacity is not fully exploited. As a further test I could train it on a larger dataset, aiming to obtained even better validation accuracy. Another technique that can be used to favour convergence of deep models is batch normalization [IS15]. This essentially consists in the normalisation and standardisation (setting to zero mean and standard deviation equal to one) of the input to each layer. Implementing batch normalisation to UNet9 leads to a strong increase of its validation loss, from 10.28 to 42.53. In my opinion, this is due to the small batch size used to train the model. Since the input images are quite large, the batch size is kept small (never more than 8) in order to fit the batch of images in the GPU card memory. Batch normalisation is performed taking into account statistics relative to the single batch. Since the batch size is small, these values might not be representative of the entire dataset. One possible way to solve this issue (in addition to using more powerful hardware) would be to use shallower models and train them on smaller input images, while increasing the batch size.

4.2 Impact of Variations on the Training Dataset

Table 3.2 shows that, increasing the dataset size, leads to a lower validation error. All the tested models trained on Dat4x overfit the validation set after roughly 120-140 epochs (see an example if Figure 4.1). This implies that the capacity of the networks is not fully exploited. The algorithms could be therefore trained on even larger datasets, which are available at CGG, hoping to reach an even lower validation error.

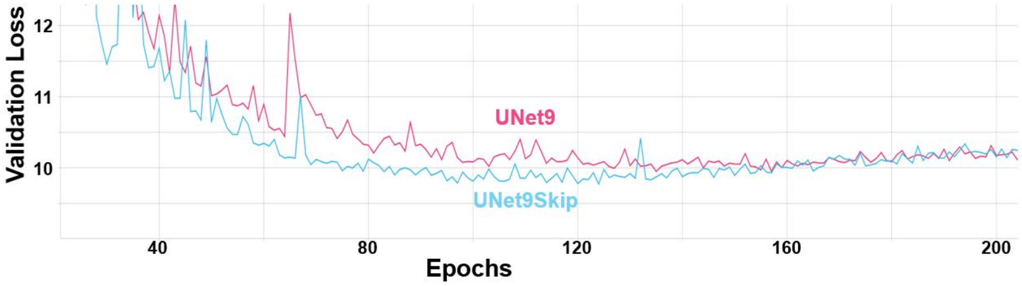

Figure 4.1: Evolution of the validation loss for the models UNet9 (purlple) and UNet9Skip (blue). The loss decreases until epoch 120 approximately, then starts increasing, showing a clear sign that the models are overfitting the training set.

In addition to the size, another important factor to take into account when collecting data for the training set, is how well it represents the entire acquisition region. In Table 3.1 we can see that, on line 1, the best agreement with G2 results is reached by UNet9_1. However, 50% of the data used to train the model came from the same line. Similarly, the best performing model one line 11 is UNet_Ref. The model though was trained on two lines sitting very close to line 11. Overall, the model producing results more in agreement with G2 is UNet9_2, which was trained on an ensemble of training examples extracted from twelve different lines spanning various portions of the acquisition area. This shows the importance of selecting training example that are representative of the different geological settings encountered over the entire region probed by the acquisition.

4.3 Quality Control of the Networks Deblending

As shown in Figure 3.2, the reference model is able to produce good quality outputs, comparable with G2. A careful analysis though reveals that some patterns, characterised by low amplitude and long wave-length, are not properly reproduced by the network. This is highlighted in Figure 4.2 (left panel), showing the difference between G2 and reference model outputs. The features underlined by blue dashed lines do not belong to the cross-talk. The neural network is not able to fully reproduce them, thus they appear when differences are plotted. I worked on two strategies to tackle the issue. First, I added two convolutional layers at the front of the UNet9 architecture. These are characterised by large kernel size (1x101), aiming to extract long wave-length features, with a vertical development, from the input images. The resulting network, UNet9InPlus, is the one achieving the lowest training error, 8.015 (down almost 11% with respect to UNet9), on the reference set. The validation error is close to the UNet9 model (10.28 and 10.36, for UNet9 and UNet9InPlus respectively). Plotting the differences between G2 and UNet9InPlus outputs, showed in 4.2 (central panel), we can see how the interested patterns appear weaker and with less horizontal continuity. This is an improvement with respec to UNet9 results. Adding convolutional layers with larger kernel size thus appear to be a suitable way to cope with the issue. More experimenting is required, modifying the kernel sizes of the convolutions in the down-sampling and up-sampling blocks. As a second approach, I trained the reference network with a different loss function, involving the Fourier transform of the de-blended signals, which was introduced in the previous chapter. The resulting model, UNet9_FTMSE achieved a training accuracy of 8.099, almost 10% better than the reference one, while displaying similar validation loss (10.33). Differences between G2 and UNet9_FTMSE outputs 4.2 (left panel) look similar to those between G2 and reference model. The fact that the training accuracy is significantly improved though, suggest that also this is a valuable strategy in improving the quality of the result. The MSE loss and the one here introduced could be adopted in combination in future tests.

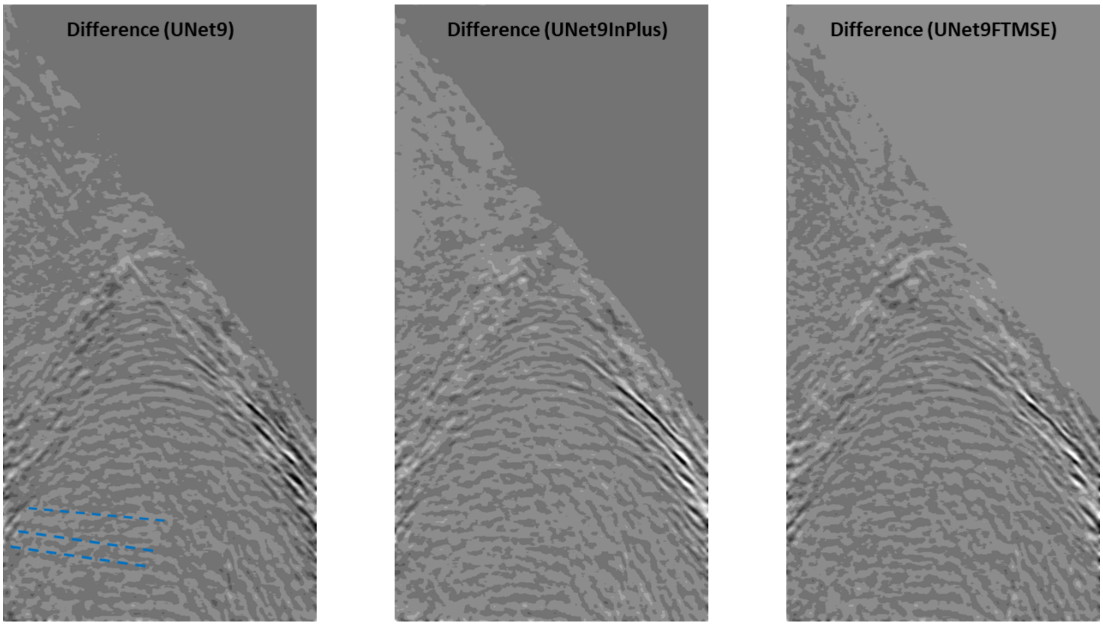

Figure 4.2: Examples of the Differences between the G2 output and those produced by UNe9 (left panel), UNet9InPlus (central panel) and UNet9FTMSE (right panel). Contrast and sharpness of the images have been modified (the same way for all three) in order to aid visualization. Blue dashed lines underline the patterns discussed in Section 4.3.

Chapter 5

Conclusion

I developed artificial neural networks with the aim to use them in signal de-blending. Applications of these techniques to seismic signal processing is in its early stage and no industrial standard is available. I obtained a number of networks capable of producing good quality results. The networks are characterised by two parts, one down-sampling the input data, and another one up-sampling the output of the first part, producing an output directly comparable with the training target. I explored how the accuracy and training time of the networks is affected by a number of architectural modifications, highlighting how simpler models, with less trainable parameters, can achieve an accuracy comparable with the one displayed by larger models, at a fraction of the computational time required for training. To the best of my knowledge, such an analysis has never been attempted before. It offers insights in the performance of U-shaped architecture that are particularly useful in production stage, where a parsimonious usage of hardware resources has a direct economic impact. I investigated the effects that different training datatsets have on the quality of the de-blending performed by the networks. In particular, datasets of different sizes, and built extracting data from various locations across the acquisition have been tested. The findings show how better accuracy can be obtained by not only increasing the size of the dataset, but also by using a sample of training examples that is representative of the various geological settings encountered across the acquisition, which can vary significantly. I conducted a careful comparison of the de-blending performed by the reference network and the one obtained with G2. The neural networks results are qualitatively good, with the major features of G2 de-blending properly reproduced. However, one particular issue has been revealed, the network is not able to reproduce long wave-wavelength low-amplitude features that might be related to actual geological structures. I proposed two strategies to tackle the issue. First, I modified the reference model adding, at its front, a set of two convolutions characterised by large kernel size. Even though this approach does not seem to completely solve the long wave-length patterns issue, its effect is promising, with the addition of only two large kernel convolutional layers reducing the training error of almost 11% (from 8.989 for UNet9, to 8.015 for UNet9InPlus). More experiments are needed in this directions, for example, increasing the kernel sizes of the convolutions constituting the up-sampling and down-sampling blocks. Increasing the kernel sizes leads to network with higher number of trainable parameters. More GPU card memory and computational time are required to train them. Therefore, careful crafting of the network architecture will be needed, in order to obtain algorithms trainable on the available hardware in a reasonable amount of time. Second, I trained the reference model adopting a loss function involving the Fourier transform of the time signals contained in the G2 and the network outputs. This in order to specifically target the difference in the frequency content between the two de-blended signals. Also this approach does not have a definitive effect and more experimentation is required in order to properly assess its impact on the quality of the de-blending. In particular, a combination of the two loss functions, MSE between output/target and MSE between their Fourier transform, each scaled by an appropriate weight, can be tested.

Bibliography

[GOO$^+$17] Alberto Garcia-Garcia, Sergio Orts-Escolano, Sergiu Oprea, Victor Villena-Martinez, and José García Rodríguez. A review on deep learning techniques applied to semantic segmentation. *CoRR*, abs/1704.06857, 2017.

[HS14] Kaiming He and Jian Sun. Convolutional neural networks at constrained time cost. *CoRR*, abs/1412.1710, 2014.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. pages 770–778, 06 2016.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. pages 448–456, 2015.

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 5 2015.

[LRB15] Wei Liu, Andrew Rabinovich, and Alexander C. Berg. Parsenet: Looking wider to see better. *CoRR*, 2015.

[MP88] Warren S. McCulloch and Walter Pitts. Neurocomputing: Foundations of research. chapter A Logical Calculus of the Ideas Immanent in Nervous Activity, pages 15–27. MIT Press, Cambridge, MA, USA, 1988.

[PGC$^+$17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986.

[RPB15] O. Ronneberger, P.Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, volume 9351 of *LNCS*, pages 234–241. Springer, 2015. (available on arXiv:1505.04597 [cs.CV]).

[SG95] R.E. Sheriff and L.P. Geldart. *Exploration Seismology*. Cambridge University Press, 1995.

[SGS15] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *CoRR*, abs/1505.00387, 2015.

[SLJ$^+$15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[Tho16] Martin Thoma. A survey of semantic segmentation. *CoRR*, abs/1602.06541, 2016.

[ZMCL15] Hongyuan Zhu, Fanman Meng, Jianfei Cai, and Shijian Lu. Beyond pixels: A comprehensive survey from bottom-up to semantic image segmentation and cosegmentation. *arXiv:1502.00717 [cs.CV]*, 34, 01 2015.