Automatic semblance velocity analysis using Convolutional Neural Networks

by

Min Jun Park

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Physics
University of Alberta

Abstract

Velocity analysis can be a time-consuming task when it is performed manually. Methods have been proposed to automate the process of velocity analysis, which, however, typically requires significant manual effort. We propose using the Convolutional Neural Network (CNN) to estimate stacking velocities directly from the semblance. Our CNN model uses two images as one input data for training. One is the entire semblance (guide image), and the other is a small patch (target image) extracted from the semblance at a specific time step. Labels for each input dataset are the root mean square (RMS) velocities. We generate the training dataset using synthetic data. After training the CNN model with synthetic data, we test the trained model with other synthetic data that was not used in the training step. The result shows that the model can predict a consistent velocity model. One also notices that when the input data is extremely different from the one used for the training, the CNN model will hardly pick the correct velocities. In this case, I propose to adopt transfer learning to update the trained model (base model) with a small portion of the target data. The latter improves the accuracy of the predicted velocity model. The Marmousi dataset and a marine dataset from the Gulf of Mexico are used for validating the proposed automatic velocity analysis algorithm.

Acknowledgements

First of all, I must thank my supervisor, Dr. Mauricio Sacchi, for his mentorship over the last two years. This experience and my career path would not exist if I did not have his advice and clarity of thoughts. I am truly grateful for all the knowledge, guidance, kindness, and arguments he had shared with me during the time I have been in the program. His willingness to explore new topics and commitment to understanding them have strongly motivated me.

A special thanks go out to everyone in Signal Analysis and Imaging Group (SAIG) for those enjoyable conversations and fruitful discussions. I would like to express my gratitude to SAIG member for the great assistance I have received. Also, I appreciate the sponsors of SAIG for their financial support and valuable feedback from the annual meetings.

Finally, I want to dedicate this piece of work to my wife, who has encouraged me with constant faith, understandings, and encouragements, which are valued beyond measure. Thanks to my parents for their love and guidance that push me onwards.

Contents

1 Introduction **1**

1.1 Background 1

1.2 Thesis overview 10

2 Convolutional Neural Networks **12**

2.1 Introduction 12

2.2 Basic CNN architecture 14

2.3 Convolutional layer 15

2.4 Pooling layer 19

2.5 Fully-connected layer 20

2.5.1 LeNet-5 22

2.5.2 AlexNet 25

2.5.3 VGG16 26

2.6 Hyperparameters 27

3 Preparation **29**

3.1 Introduction 29

3.2 Data definition 30
3.3 Training data construction 34
3.4 CNN architecture adoption 40
3.5 Hyperparameter tuning 41
3.5.1 Learning rate 42
3.5.2 Batch size 44
3.5.3 Activation Function 46
3.5.4 Final CNN architecture 48

4 Training and Testing **51**
4.1 Introduction 51
4.2 Base model training 52
4.3 Transfer Learning 53
4.4 Numerical examples 54
4.4.1 Custom dataset 54
4.4.2 The Marmousi dataset 59
4.5 Field data example 63

5 Conclusions **75**
5.1 Main contributions 75
5.2 Future work 76

Bibliography **78**

Appendices

A Reproducibility **86**

List of Tables

3.1 Evaluation results (Loss, Accuracy, Precision, Recall, and F1 score) for five models with different Learning rates. All the results are from the model after 100 epoch training process. Data used for accuracy is the same as the training data. `Learning rate` 0.001 shows the best performance in every evaluations. . 44

3.2 Evaluation results (Time per epoch, Loss, Accuracy, Precision, Recall and F1 score) for five models with different batch size. All the results are from the model after 100 epoch training process. Data used for accuracy is the same as the training data. 46

3.3 Evaluation results (Loss, Accuracy, Precision, Recall, and F1 score) for three models with different activation functions. All the results are from the model after 100 epoch training process. Data used for accuracy is the same as the training data. . . . 48

3.4 Evaluation results (Time per epoch, Loss, Accuracy, Precision, Recall and F1 score) for two different models (Custom and VGG16). All the results are from the model after 100 epoch training process. Data used for accuracy is the same as the training data. 50

4.1 Information of the model after Transfer Learning with 19 semblances (second row) and 38 semblances (third row). 60

4.2 Parameters information of the Gulf of Mexico dataset. 63

4.3 Information pertaining Transfer Learning for the Gulf of Mexico dataset. 64

List of Figures

1.1 (a) Common Shot Gather (CSG). (b) Common Midpoint (CMP) Gather. 5

1.2 (a) The CMP gather before NMO correction. (b) The CMP gather after NMO correction. After the NMO correction, all traveltimes are matched to the zero-offset time. After NMO correction traces can be stacked (summed) coherently. (c) Traces after stacking. 6

1.3 The CMP gather and its semblance (Equation 1.4) panel displaying velocity spectra. 6

1.4 (a) CMP gather. (b) Semblance panel with estimated velocity (red line). (c) NMO corrected gather. 8

2.1 Typical CNN architecture. It consists of two convolutional layers followed by `FC layer`. 14

2.2 $8 \times 8 \times 3$ Input image and the $3 \times 3 \times 3$ kernel are shown. Convolutional operator (`X`) between the local receptive field ($3 \times 3 \times 3$ shaded part in the Input image) and the kernel generates a value 9 as an output. 16

2.3 Four different non-linear activation functions: (a) `Sigmoid`, (b) `tanh`, (c) `Leaky ReLU`, and (d) `ReLU`. 17

2.4 Average-pooling operator. It takes the values' average in the patch extracted from the input image. This process repeats through the entire input image. 19

2.5 Max-pooling operator. It takes maximum value of the patch from an input image. This process repeats through the entire input image. 20

2.6 The fully connected layer uses the same principle as the conventional MLP where W is the length of the input vector and H is the desired output length. It performs a matrix multiplication between the declared weight matrix of size $W \times H$ and input vector to form an output of length H. 21

2.7 LeNet-5 architecture (LeCun et al., 1998). It contains two convolutional layers followed by `FC layer`. After each convolutional layer, there is a sub-sampling operator or average pooling. 22

2.8 The feature maps from each layer corresponding to each input (LeCun et al., 1998). The first stage of the feature maps (C1, C2) show relatively simple features such as edge detection and color reversal, while the final output (F6) shows more complicated features. 24

2.9 AlexNet architecture (Krizhevsky et al., 2012). This architecture consists of two models in parallel. Each model includes 5 convolutional layers and 3 fully connected layers. Also, there are three max-pooling layers to perform sub-sampling. 25

2.10 `VGG16` architecture. Two or three convolutional layers are stacked before sub-sampling. It consists of a total of 16 layers (13 convolutional layers and 3 fully connected layers). Max pooling is adopted for the sub-sampling. 26

3.1 Three patches (v,t) extracted from the same semblance. Red dots indicate the stacking velocities at the center of each patch. These are from the manual velocity analysis. The red dot in (a) the first patch matches with the high-condensed energy while other dots in (b) the second and (c) third patches have higher velocities than the high-condensed energies. 31

3.2 Input data definition. For each semblance, there is one Guide image (G) and 45 Target images (T). Then, Input data (I) pair each G_n and T_{n,t_m} to produce the input data (I_{n,t_m}), where n is the CMP number and the index $m = \{1, 2, 3..., 45\}$ corresponds to the intercept traveltime t_m. 32

3.3 (a) The P-wave velocity model. The snapshots of the forward modeling at the time (b) 0.3 sec, (c) 0.9 sec, (d) 1.5 sec, (e) 2.1 sec, and (f) 2.7 sec. I use Keys method for boundary condition. It clearly shows that the reflected wave is formed at the layer's boundary. 37

3.4 Five synthetic velocity models are used to generate the training data. All models have a different number of layers and the velocity distribution that increases with depth. 38

3.5 Workflow displaying the training step. I solve the acoustic wave equation via the Finite Difference method to compute synthetic shot gathers. I calculate the semblance panels for each CMP location after CMP sorting and automatic gain control. To compute the labels, the RMS velocities are computed analytically from the velocity models. 39

3.6 The CNN architecture I adopt. It is based on `LeNet-5`. It contains two convolutional layers (`Conv1`, `Conv2`) and two fully-connected layers (`FC1`, `FC2`) followed by the softmax classifier. The model outputs 40 numbers in one array for each input data. Each output represents the probability of each class. 40

3.7 Loss trends for five models with different learning rates. `Learning rate` 0.001 (green line) shows the best result of convergence. . 43

3.8 Loss curve for five models with different batch sizes $(16, 64, 128, 256, 512)$. I have not found any noticeable correlation between batch size and the performance of the model. 45

3.9 Loss curve for four models with different activation functions (`Sigmoid, tanh, ReLU, Leaky ReLU`). 47

3.10 Loss trends for two different models: custom architecture and VGG16. 49

4.1 Loss trend for the base model training. 53

4.2 The concept of Transfer Learning. 54

4.3 The velocity model used for testing. This model was not used in the training stage. 55

4.4 The label velocity model. 56

4.5 Velocity field predicted via CNN. 57

4.6 (a) Predicted velocity (red line) and true RMS velocity (blue line) are shown in conjunction with the semblance. (b) NMO correction corresponding to the CMP gather. (c) The results of NMO correction with the true velocity. (d) The predicted velocity are shown as well. 58

4.7 Stack sections. (a) Stack section obtained with the true velocity model. (b) Stack section obtained via the velocity model predicted by CNN. (c) Difference panel ((b) minus (a)). (d) The near offset traces. 59

4.8 The Marmousi velocity model. 59

4.9 Loss trend for Transfer Learning using the Marmousi dataset. 61

4.10 (a) Label (true) velocity model, (b) the predicted velocity model from Transfer Learning CNN with 5% of target data, (c) the predicted velocity model from Transfer Learning CNN with 10% of target data. 62

4.11 Near-offset traces of a dataset from the Gulf of Mexico. Data includes the salt body from position CMP 700 to CMP 1600. . 65

4.12 Loss trend for Transfer Learning using the Gulf of Mexico dataset. 66

4.13 The velocity field created using linear interpolation with velocity analysis at every 50 CMP gathers. Dashed lines are indicating the location where I have got the data for Transfer Learning (CMP: 202, 302, 402, 502, 602, 1002, 1152, 1302, 1452, 1599). 67

4.14 The velocity field predicted by CNN. I use a total of ten semblances and velocity profiles (CMP: 202, 302, 402, 502, 602, 1002, 1152, 1302, 1452, 1599) to perform Transfer Learning. Dashed lines are indicating the data's CMP location for Transfer Learning. 68

4.15 The difference in velocity field (Figure 4.13 minus Figure 4.14). 69

4.16 Semblance panels. CMP number: (a) 350, (b) 750 (c) 1250, (d) 1400. Manual analysis velocities (red lines) and the velocities predicted via CNN (white lines). These semblance panels are not used for Transfer Learning. 71

4.17 Stack section obtained by manual analysis velocity model. . . 72

4.18 Stack section obtained via the velocity model predicted by CNN. 73

4.19 (a) The zoomed part from CMP 650 to 1000, τ =2.4 to τ =3.6 s.(Right) The results from the manual analysis and (Left) the results from the CNN prediction. Red arrows are indicating clearer reflectors in the zoomed parts of the CNN prediction. 74

4.20 CMP 600 to 1000, τ =5 to τ =6.2 s. (Right) The results from the manual analysis and (Left) the results from the CNN prediction. Red arrows are indicating clearer reflectors in the zoomed parts of the CNN prediction. 74

CHAPTER 1

Introduction

1.1 Background

Reflection seismology is one of the geophysical methods that is used to estimate the images of the subsurface. It is often adopted by oil and gas exploration because it can offer high-resolution maps of subsurface interfaces and structures of geological interest (Yilmaz, 2001). Reflection seismology uses human-made sources such as explosives, mechanical vibrators, and airguns to create elastic waves that propagate downward into the subsurface. These waves are reflected by geological interfaces and generates an upward propagating wavefield that is recorded by arrays of sensors deployed on the surface of the earth. Seismic data acquired by sensors in the form of time series[1] are processed and then, utilized for imaging the interior of the earth.

Seismic data acquisition entails deploying a large array of receivers and sources

[1]Seismograms or seismic traces

on the surface of the earth. When one source is activated, all sensors record the reflected wavefield produced by the source. The product of this experiment is called a Common Shot Gather (CSG), which represents the group of seismic traces generated by a given source. A CSG is portrayed in Figure 1.1 (a). I denote x_r and x_s the receiver and source coordinates, respectively. Data from multiple CSGs are then sorted to create a Common Midpoint (CMP) gather. Figure 1.1 (b) illustrates a CMP gather. The CMP gather contains seismic traces generated by different sources and acquired by different receivers. These traces have a common midpoint x position

$$x = \frac{x_s + x_r}{2} \,.$$

Conversely, each source-receiver pair participating in a CMP gather has a different offset h (source-receiver distance) that is given by

$$h = x_r - x_s \,.$$

A simple trigonometric exercise shows that the two-way traveltime of a reflection as a function of offset is given by a hyperbola

$$t(h) = \sqrt{t_0^2 + \frac{h^2}{v_1^2}} \,, \tag{1.1}$$

where $t(h)$ is the two-way travel time of the trace of offset h, $t_0 = t(h = 0)$ is the two-way traveltime of the zero-offset trace, and v_1 is the velocity of the layer above the reflection point. Equation 1.1 holds for a single horizontal

interface and for a dipping interface. In the dipping interface case, the velocity v_1 in expression 1.1 must be replaced by $v_1/cos(\phi)$ where ϕ is the dip of the interface (Yilmaz, 2001).

Similarly, one can show that for a series of horizontal layers equation 1.1 is a good approximation to the reflection traveltime if I replace the velocity v_1 by a weighted average of the velocities above the reflection point. In this case, I can express

$$t(h) = \sqrt{t_0^2 + \frac{h^2}{V(t_0)^2}} \tag{1.2}$$

and consider $V(t_0)$ as the root-mean-squared velocity of the layers above the reflection point (Yilmaz, 2001). Similarly, t_0 indicates the two-way traveltime from the surface to the reflecting layer for a source-receiver pair at the same position ($h = 0$). In other words, t_0 is time a wave takes to travel from the source to an interface and back to the receiver if the source and the receiver were placed at the same spatial position.

The Normal Moveout (NMO) correction (Figure 1.2) is a procedure that corrects the CMP gather to make it behave like a group of traces acquired with source-receiver distance $h = 0$. In other words, it is a correction that permits to flatten reflection hyperbolas. The correction is given by the following expression

$$\Delta t(h) = t_0 - \sqrt{t_0^2 + \frac{h^2}{V(t_0)^2}}\,. \tag{1.3}$$

We can use the above equations to align travel-times of any arbitrary offset h

to that of the zero offset trace. The velocity in equation 1.3, $V(t_0)$, is the NMO velocity or stacking velocity which, in general, is close to the the hyperbola's RMS velocity. The stacking velocity is the velocity of the hyperbola that optimally stacks the reflections after NMO correction.

As it was already mentioned, the goal of the NMO correction is to flatten the seismograms that belong to a CMP gather. Once the traces are aligned, they can be stacked (summed) to enhance their signal-to-noise ratio and to generate the so called stacked trace. The latter is the ideal seismic trace one would have been able to obtain via a zero-offset seismic experiment. Figure 1.2 (a) shows a CMP gather prior to the NMO correction. This case corresponds to an earth model composed of two geological interfaces. Figure 1.2 (b) is the CMP gather after NMO correction. Finally, Figure 1.2 (c) is the CMP gather after NMO correction and stacking. The stacking process is a fundamental part of seismic imaging because adjacent CMPs are NMO corrected and stacked to produce a seismic section. A seismic section is the subsurface image that corresponds to a map of the seismic reflectivity of the earth, functioning as midpoint and time. Seismic sections are important in reflection seismology because they generate time domain images of the earth interior. These images are commonly used for geological structure interpretation (Yilmaz, 2001). In order to perform the NMO correction, one needs to know $V(t_0)$. The process estimating $V(t_0)$ directly from the CMP gather is often called Velocity Analysis.

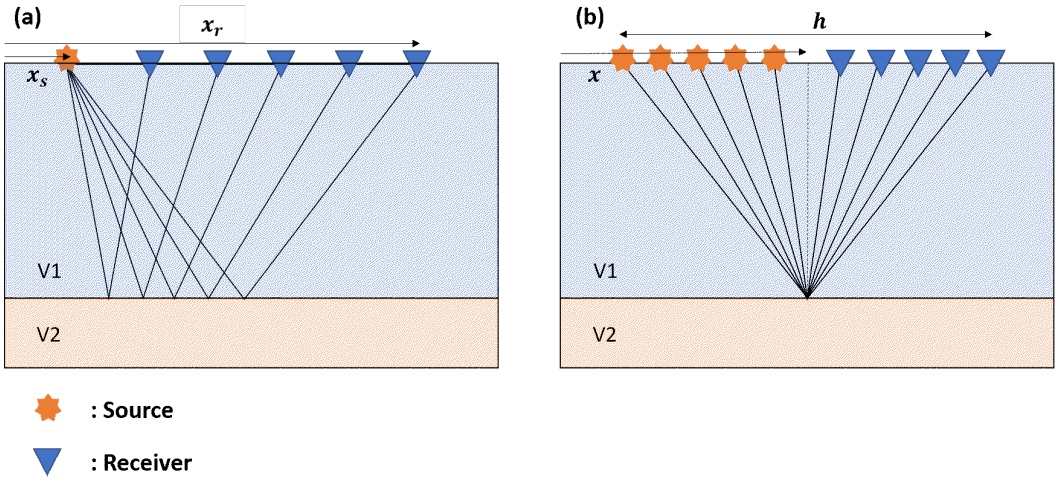

Figure 1.1: (a) Common Shot Gather (CSG). (b) Common Midpoint (CMP) Gather.

Subsurface velocity analysis is an essential processing step since it provides the stacking velocity $V(t_0)$ that is necessary for the NMO correction. It is also needed for extracting preliminary velocity information for advanced seismic imaging methods such as migration (Yilmaz, 2001). A popular method to estimate stacking velocities is via semblance velocity analysis (Taner and Koehler, 1969; Neidell and Taner, 1971). Semblance is a measure of energy that can be used to determine stacking velocities from CMP gathers optimally. The semblance can be interpreted as an energy spectra, and it can be displayed as an image indicating the reflection energy versus two-way traveltime t_0 and stacking velocity $V(t_0)$.

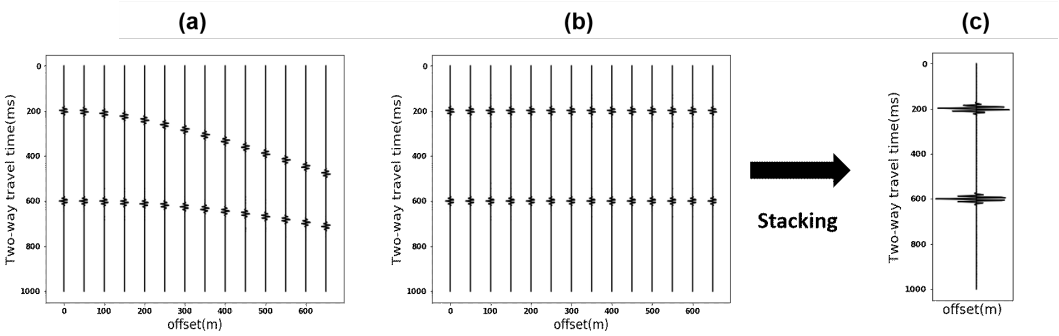

Figure 1.2: (a) The CMP gather before NMO correction. (b) The CMP gather after NMO correction. After the NMO correction, all traveltimes are matched to the zero-offset time. After NMO correction traces can be stacked (summed) coherently. (c) Traces after stacking.

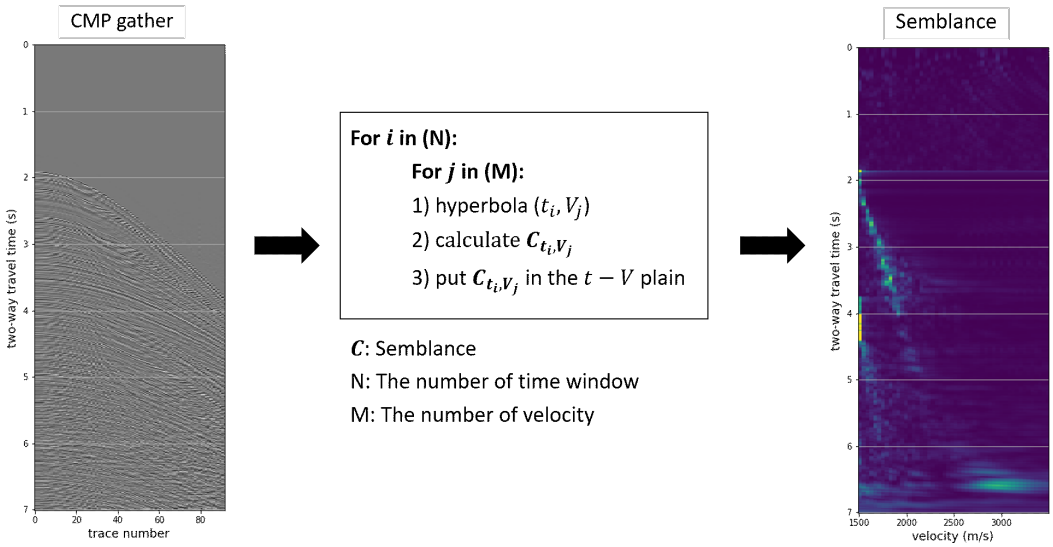

Figure 1.3: The CMP gather and its semblance (Equation 1.4) panel displaying velocity spectra.

Figure 1.3 illustrates the process of velocity analysis for one CMP gather. For different times, one extracts windows representing hyperbolas with a trial

velocity. The semblance (Equation 1.4) is measured in the window of analysis and it is placed in the $t_0 - V(t_0)$ plane. K indicates the number of trace in the window and a_k represents the kth trace's amplitude. Thus, the energetic peaks in the semblance represent the $t_0 - V(t_0)$ pairs that flatten the reflections

$$\boldsymbol{c} = \frac{\left(\sum_{k=1}^{K} a_k\right)^2}{K\left(\sum_{k=1}^{K} a_k^2\right)}. \tag{1.4}$$

Once the semblance panel is obtained, the stacking velocity can be inferred by extracting the energy peaks manually. The estimated velocity is then used for the NMO correction (Figure 1.4 (c)). In general, velocity analysis is a time-consuming task because it requires a visual examination of a large number of semblance panels by a processor. Many studies have been conducted to automate velocity analysis (Toldi, 1985; Abbad et al., 2009; Fomel, 2009; Choi et al., 2010; Chen, 2018). Non-convolutional Neural Networks (NN) have also been suggested for velocity analysis (Schmidt and Hadsell, 1992; Fish and Kusuma, 1994; Calderón-Macías et al., 1998).

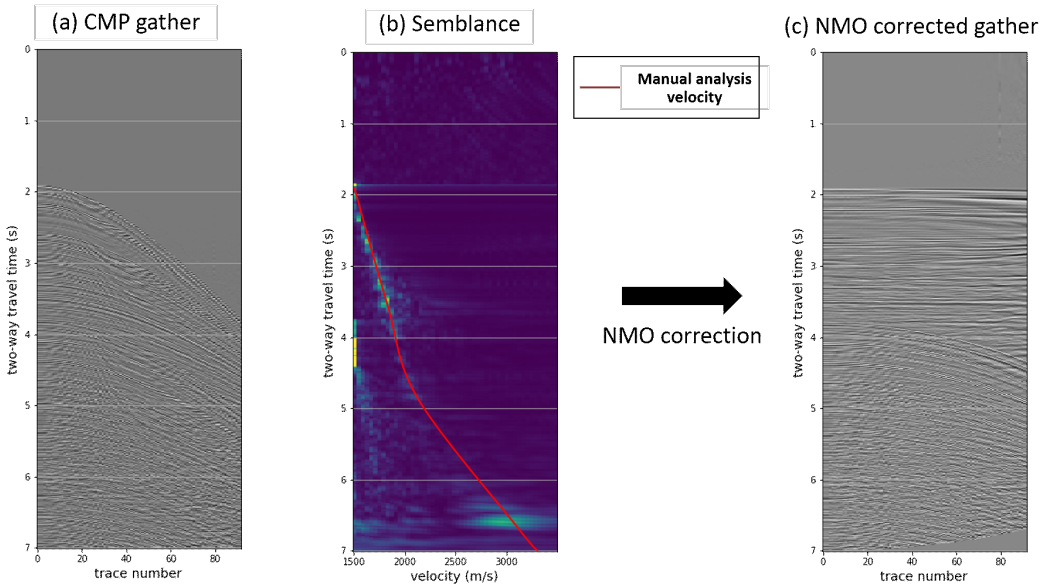

Figure 1.4: (a) CMP gather. (b) Semblance panel with estimated velocity (red line). (c) NMO corrected gather.

Recently, there has been a significant improvement in deep learning itself due to the advancement of computer performance (LeCun et al., 2015a). A Convolutional Neural Network (CNN) is one of the most powerful algorithms to classify images among various approaches in deep learning (LeCun et al., 1989). This is because CNN can consider the local connectivity of the input image by adopting convolutional filters that provide invariance of geometric shifts or distortions (LeCun et al., 1995). Given its superiority in extracting features from images, CNN has been extensively employed in image classification or recognition (Lawrence et al., 1997; Simard et al., 2003; Krizhevsky et al., 2012; Karpathy et al., 2014; Maturana and Scherer, 2015; He et al., 2016). Studies of the application of deep learning to seismic data processing have also been carried out. For instance, Araya-Polo et al. (2018) has proposed to

adopt deep neural network for tomographic inversion. Also, CNN applications in problems such as fault classification (Xiong et al., 2018; Wu et al., 2018, 2019), first-break picking (Yuan et al., 2018) and denoising (Liu et al., 2018) have been explored.

In this thesis, I propose to adopt CNN for automatic velocity analysis. I consider the velocity picking process as an image classification problem by assuming that small patches extracted from the semblance represent the stacking velocity corresponding to a specific time. For training, I use synthetic custom models to build the training dataset. I first compute synthetic shot gathers from those models by using the finite difference method (Taflove and Hagness, 2005) and calculate semblance panels after performing CMP sorting. Then, I obtain the input data from the semblance panels and calculate the RMS velocity from the velocity model to generate the labels. After training, the trained model can predict a consistent velocity field. I assume that the CNN model trained with the RMS velocity can predict the stacking velocity because the synthetic models correspond to quasi-horizontal structures (Yilmaz, 2001). However, if the target data are significantly different from the training data, the base model can barely output the correct labels. To overcome this limitation, I use transfer learning (Pan et al., 2010; Yosinski et al., 2014), which is an additional training process with a small portion of target data. In my research, I have found that although the new input data has different features from those of the training dataset, the CNN model obtained after transfer learning can predict reasonable stacking velocities. The methodology developed for automatic velocity analysis was adopted to process a marine dataset

from the Gulf of Mexico. My work also addresses the importance of transfer learning to save computing time. For instance, I have found that training the CNN for automatic velocity analysis can be considerably reduced by adopting transfer learning.

1.2 Thesis overview

In **Chapter 2**, I describe the principal operators utilized for setting the CNN. I first explain the basic CNN architecture. Then, I describe the purpose and operational principles behind the convolutional layer, the pooling layer, and the fully connected layer that make up the CNN. After that, I introduce three representative CNN models named LeNet-5, AlexNet, and VGG16 (LeCun et al., 1998; Krizhevsky et al., 2012; Simonyan and Zisserman, 2014). Finally, I discuss the problem of understanding and selecting parameters (hyperparameters) needed for CNN training. I also discuss the hyperparameter tuning process.

Chapter 3 introduces the conventional semblance velocity analysis and explain data preconditioning prior to CNN training. I describe how one can define the training dataset that includes the input data and labels.

Chapter 4 explains the CNN model training process. In this chapter, I have adopted synthetic and field data examples to train and test the CNN model for automatic velocity analysis. I use three datasets: a custom velocity model, the Marmousi model (Versteeg, 1994), and a marine dataset from the Gulf

of Mexico[2] (Guitton, 2003). In this chapter, I also introduce and discuss Transfer Learning, an alternative training method, that permits to improve the predictability power of the CNN model.

Chapter 5 includes the conclusions of my project and recommendations for future work.

[2]Mississippi Canyon dataset

CHAPTER 2

Convolutional Neural Networks

2.1 Introduction

As a subset of Artificial Neural Networks (Rosenblatt, 1958), CNN is gaining a tremendous amount of popularity because of its powerfulness of extracting features from images. Unlike CNN, Artificial Neural Network (ANN) has a limitation in handling images because it can use only one-dimensional data as an input. Thus, it is necessary to flatten the image data into one dimension before adopting ANN. A loss of spatial information during this process reduces the ANN's feature extracting ability. CNN, however, is designed to process multidimensional data (LeCun et al., 1998, 2015a). It uses a kernel with a convolution operator to extract features preserving data spatial information. Thus, CNN not only maintains the shape of input data but also effectively extracts features of the images. With multiple kernels, it can produce various features which can represent the image. Each kernel shares parameters so that

the computational cost for training decreases a lot (LeCun et al., 1998, 2015a).

In this chapter, I describe the CNN that I propose to automate semblance velocity analysis. As mentioned before, CNN has excellent performance in extracting features from images by considering the local connectivity of the input data. I will treat the semblance extracted from CMP gathers as images that will constitute the input to our CNN algorithm for automatic velocity analysis. Before moving into my particular seismic application, in this chapter, I describe the building blocks of CNN algorithm. In particular, I will briefly explain CNN architecture itself as well as the main three components of CNN, namely the Convolutional Layer, the Pooling Layer, and the Fully-Connected Layer. I will also introduce three popular CNN architectures: LeNet-5, AlexNet, and VGG16 (LeCun et al., 1998; Krizhevsky et al., 2012; Simonyan and Zisserman, 2014).

2.2 Basic CNN architecture

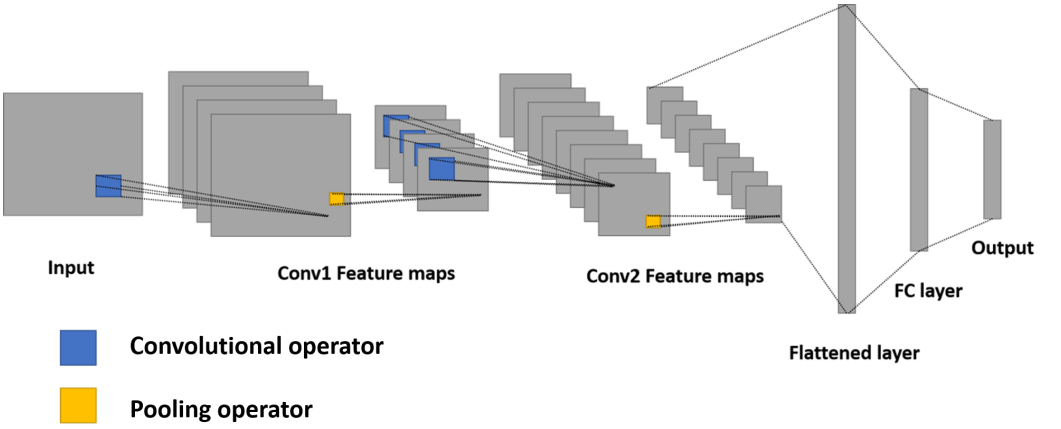

Figure 2.1: Typical CNN architecture. It consists of two convolutional layers followed by `FC layer`.

The typical CNN architecture for classification includes two parts. The first part is a feature extraction part, which consists of two types of layers: convolutional layers and pooling layers. In convolutional layers, each layer produces feature maps from the output of the previous layer by using convolutional operator. Also, an activation function such as a `ReLU` (Nair and Hinton, 2010) gives non-linearity to the feature maps in this stage. Then, pooling layers decrease the size of feature maps to reduce computational cost. Generally, at least two pairs of convolutional layer and pooling layer exist in a CNN architecture so that CNN can be trained with more complex features. The second part of a CNN is the classification part, which is known as the fully connected layers. It firstly flatten the final output (feature maps) of the feature extrac-

tion part. Then, it performs matrix multiplication between the flattened array and the declared weight matrix. The length of the final output is the same as the number of classes to perform classification. Figure 2.1 shows the typical CNN architecture, which consists of two convolutional layers, followed by a fully connected layer.

2.3 Convolutional layer

The convolutional layer consists of two parts. The first is the convolutional kernel (or filter) K, which extracts different features from each input image I. The convolutional kernel performs a simple dot product between a patch extracted from the input data and the kernel itself. In signal processing, the operation is basically the traditional two-dimensional convolution between two matrices (Simard et al., 1999). The output of this operation is called the `feature map` and denoted by O. If I assume that I have kernel (filter) of size $H \times W$ samples, the convolution between the input data and the kernel is given by

$$O_{i,j} = \sum_{h=0}^{H} \sum_{w=0}^{W} I_{i+h,j+w} K_{h,w} \, . \tag{2.1}$$

The size of the kernel K depends on the features' complexity. Figure 2.2 shows how the convolution kernel works. It generates a value, 9, after the convolutional operator (Equation 2.1) between the local receptive field and the kernel. After this process, the kernel slides and repeats the same process throughout

the entire input image to generate the full two-dimensional feature map. It is important to point out that the same kernel applies equally to all parts of the image because it considers a spatially stationary operator. Considering a kernel that changes for each feature point $O_{i,j}$ will entail adopting a non-stationary two-dimensional convolution with a different Kernel for each out i, j. Clearly, the latter will create a large number of unknown filter coefficients and probably lead to overfitting.

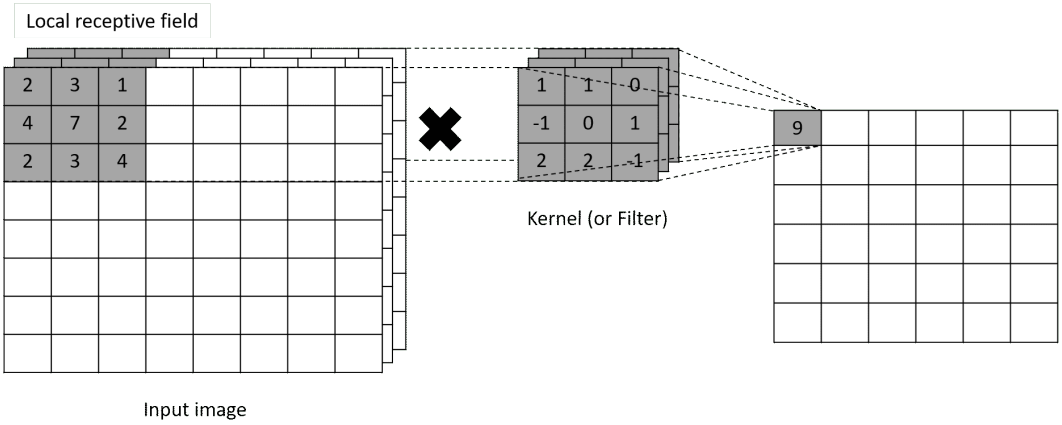

Figure 2.2: $8 \times 8 \times 3$ Input image and the $3 \times 3 \times 3$ kernel are shown. Convolutional operator (`X`) between the local receptive field ($3 \times 3 \times 3$ shaded part in the Input image) and the kernel generates a value 9 as an output.

The second part of the convolutional layer is the activation function. The latter serves to adjust the value that is passed to the next neuron. The activation function was firstly used in a perceptron, which is the basic template for an ANN (Rosenblatt, 1958). It receives multiple signals and combines them to determine whether to propagate the signal using a step function. The multi

layers perceptron (MLP) that was developed from the perceptron, adopted a different activation function called `Sigmoid` function (Equation 2.2). It determines the strength of the signal to be sent to the next neuron when multiple signals arrive (Rumelhart et al., 1988). The activation function of an ANN (or a CNN) is always non-linear. This is because the output of the multi-layer ANN (or CNN) with linear activation function is just a linear combination of the various inputs. In other words, it becomes meaningless to stack multiple layers with a linear activation function (LeCun et al., 1998).

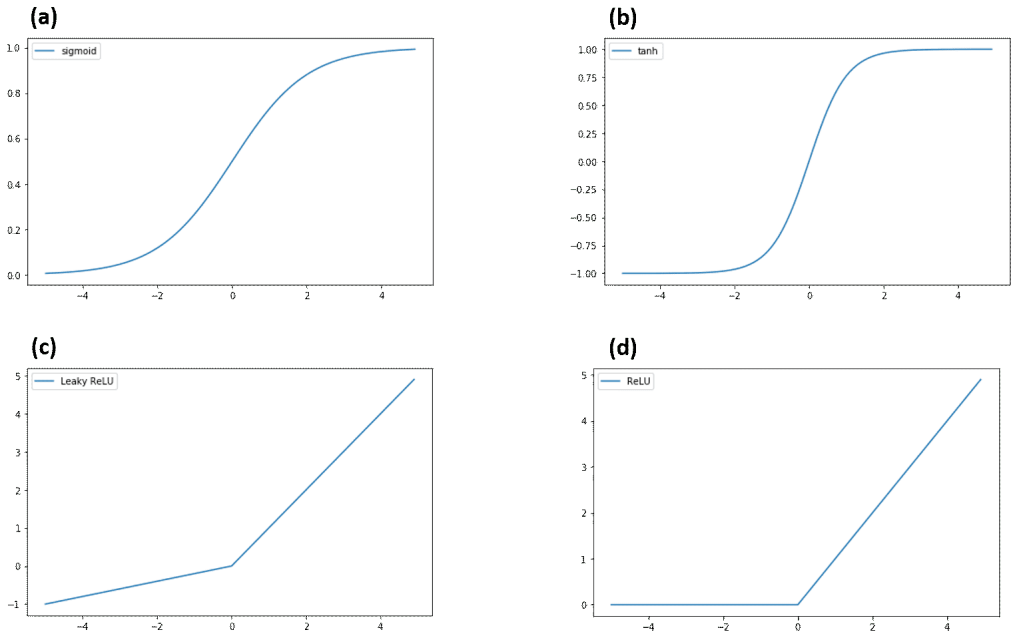

Figure 2.3: Four different non-linear activation functions: (a) `Sigmoid`, (b) `tanh`, (c) `Leaky ReLU`, and (d) `ReLU`.

Figure 2.3 shows four different non-linear activation functions. I would like to point out that both the `Sigmoid` (Equation 2.2) and the `tanh` (Equation

2.3) activation functions are barely used nowadays since they are affected by the so called vanishing gradient problem. When the input value is close to 0 or 1, the corresponding partial value is very close to zero, which causes the gradient to converge to zero during the back-propagation training step. Thus, the vanishing gradient problem becomes more serious as the depth of the CNN or the number of epoch increases. Nair and Hinton (2010) suggests to adopt a `ReLU` (rectified linear unit) activation function (Equation 2.4) to solve this problem. However, there is also a disadvantage with the `ReLU` activation function because some of the neurons can die when the slope becomes zero for negative input values. To circumvent this problem, Maas et al. (2013) suggests adopting `Leaky ReLU` (Equation 2.5) activation function. Analytical expressions for the aforementioned activation functions are given by

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}\,, \tag{2.2}$$

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}\,, \tag{2.3}$$

$$ReLu(x) = \max(0, x)\,, \tag{2.4}$$

and

$$LeakyReLu(x) = \max(\alpha x, x)\,. \tag{2.5}$$

2.4 Pooling layer

The purpose of the pooling layer is to conduct a sub-sampling of the input in order to decrease the computational cost of training. This sub-sampling can also mitigate the overfitting problem that occurs when too many neurons are trained. There are two popular pooling methods named `average pooling` (Figure 2.4) and `max pooling` (Figure 2.5). The `average pooling` takes the average of all the values in the selected patch while `max pooling` just takes the maximum value of the patch.

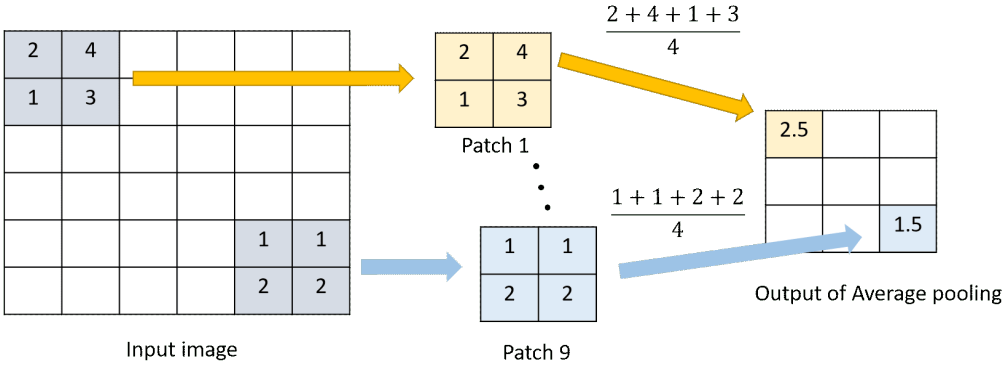

Figure 2.4: Average-pooling operator. It takes the values' average in the patch extracted from the input image. This process repeats through the entire input image.

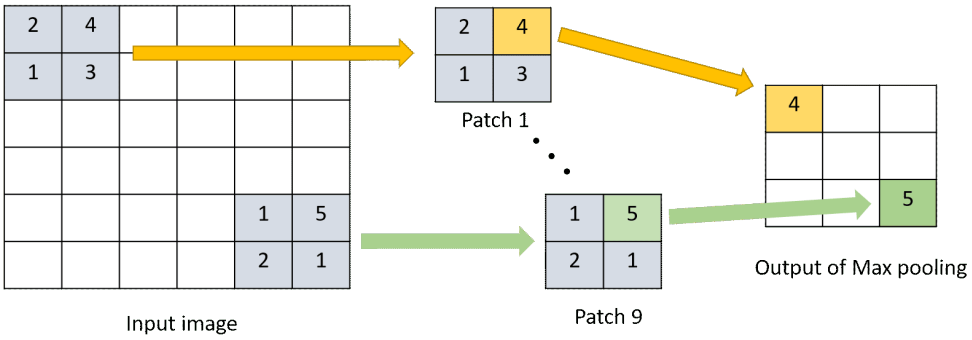

Figure 2.5: Max-pooling operator. It takes maximum value of the patch from an input image. This process repeats through the entire input image.

2.5 Fully-connected layer

The fully-connected layer (`FC layer`) follows the same principle as the conventional MLP. To solve the classification problem with the CNN architecture, the matrix (or tensor) has to be flattened into a one-dimensional vector. Then, through the `FC layer`, the final output must be the same length as the number of classes. Figure 2.6 shows how the output of the `FC layer` produces the vector with the specific length. The fully-connected layer performs a matrix multiplication between the declared weight matrix of size $W \times H$ and input vector to yield an output of length H. Thus, each element of the final output of the `FC layer` correlates with a corresponding class. In my research, I have adopted a `Softmax` classifier (Equation 2.6) to convert each element of the output vector to a probability for each class

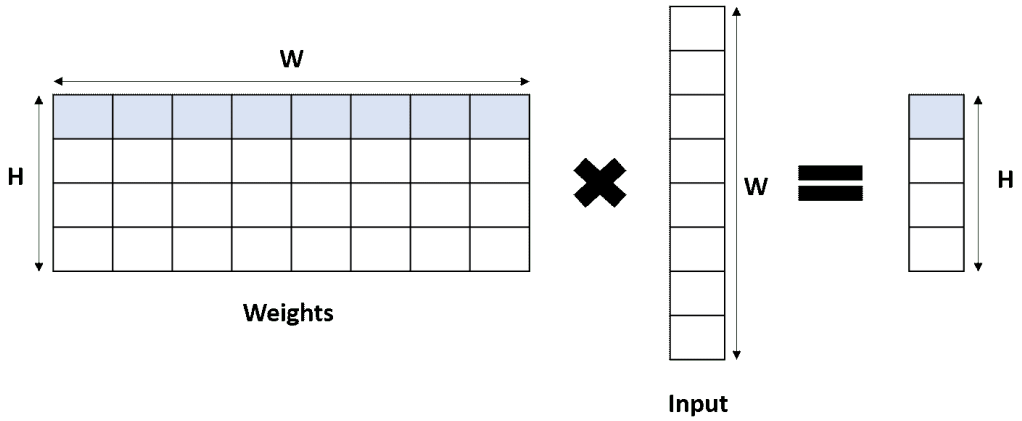

Figure 2.6: The fully connected layer uses the same principle as the conventional MLP where W is the length of the input vector and H is the desired output length. It performs a matrix multiplication between the declared weight matrix of size $W \times H$ and input vector to form an output of length H.

$$Softmax(\mathbf{x}_j) = \frac{e^{x_j}}{\sum_{i=1}^{k} e^{x_i}} \quad \text{for } j = 1, \ldots, k. \tag{2.6}$$

For training, one needs to define an objective function that allows comparing the predicted probability to the true probability (label). Equation 2.7 shows one popular objective functions named Cross-Entropy objective function. Recently, the Cross-Entropy objective function has been reported to yield a higher training process (Nielsen, 2015)

$$J\left(\phi_1, \phi_2\right) = -\sum_{i=1}^{N} \phi_1(i) \log\left(\phi_2(i)\right), \tag{2.7}$$

where N is the number of classes, ϕ_1 and ϕ_2 are the true and predicted probability for the i-th class. Since every ϕ_1 is zero except when i is k, I can simplify

Equation 2.7 as follows

$$J(\phi_1, \phi_2) = -\log(\phi_2(k)) \ , \tag{2.8}$$

where k is the target number. All the weights are then repeatedly optimized using the back-propagation algorithm.

2.5.1 LeNet-5

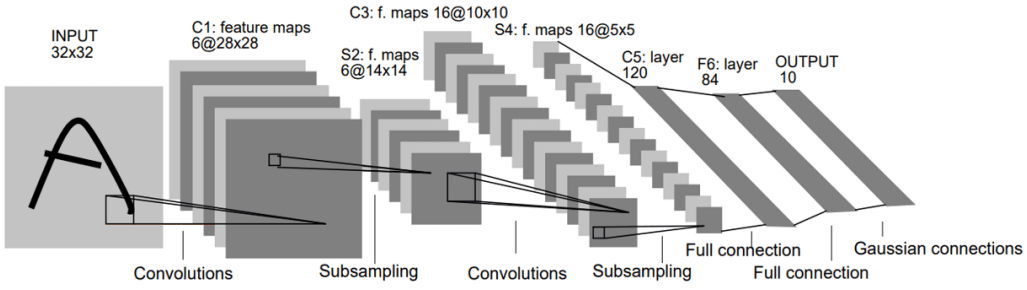

Figure 2.7: LeNet-5 architecture (LeCun et al., 1998). It contains two convolutional layers followed by `FC layer`. After each convolutional layer, there is a sub-sampling operator or average pooling.

LeCun et al. (1998) first suggested a CNN model called `LeNet`, which is designed to recognize handwriting and `LeNet-5` is the latest version of it (LeCun et al., 2015b). `LeNet-5` has a quite simple architecture compared to the modern architectures. Figure 2.7 shows this architecture. The image is taken from LeCun et al. (1998). This architecture has 4 layers, 2 convolutional layers (C1 and C3) and 2 fully connected layer (C5 and F6) followed by an output layer.

The sub-sampling layers are given by 2×2 average pooling layers. In terms of the activation function, this architecture uses the *tanh* activation function which is used through every layer except for the output layer. At the output, a *sigmoid* function is used as the activation function. Even if `LeNet` has a relatively simple architecture and a small number of parameters, the error rate it achieved is below 1% when working with the MNIST database. The latter is a huge dataset of handwritten digits often adopted to test machine learning algorithms (Deng, 2012). Figure 2.8 shows the feature maps extracted from each layer corresponding to each input. The first stage of the feature maps (C1, C2) show relatively simple features such as edge detection and color reversal. The final output layer (F6) shows more complicated features. The output ($F6$) always represents the labels.

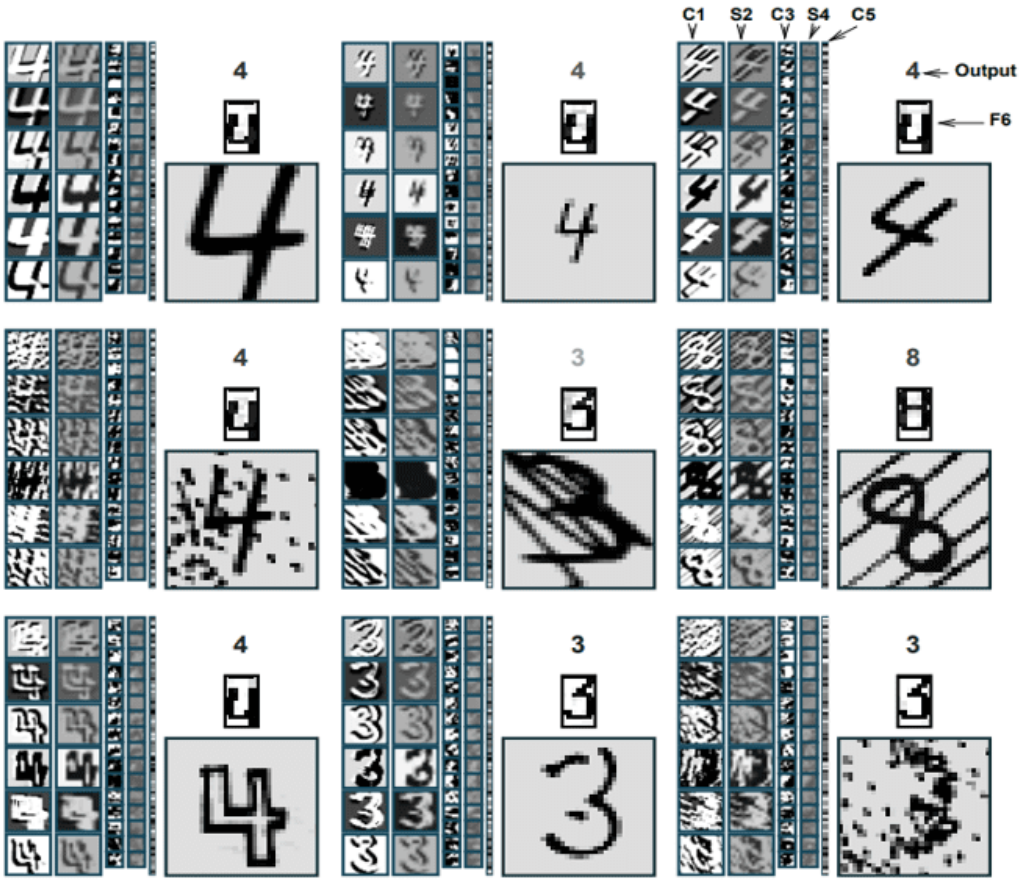

Figure 2.8: The feature maps from each layer corresponding to each input (LeCun et al., 1998). The first stage of the feature maps (C1, C2) show relatively simple features such as edge detection and color reversal, while the final output (F6) shows more complicated features.

2.5.2 AlexNet

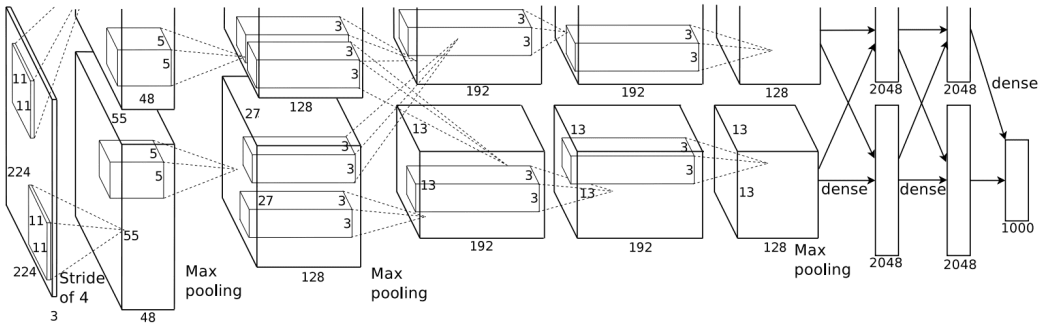

Figure 2.9: AlexNet architecture (Krizhevsky et al., 2012). This architecture consists of two models in parallel. Each model includes 5 convolutional layers and 3 fully connected layers. Also, there are three max-pooling layers to perform sub-sampling.

`AlexNet` architecture was the top in the ImageNet challenge (Deng et al., 2009) in 2012 (Krizhevsky et al., 2012). Top-5 error rate was 16.4%. To classify a large size image ($224 \times 224 \times 3$) into a thousand classes, it was necessary to make the network deeper than `LeNet-5` architecture. `AlexNet` is the foundation of the current deep convolutional neural network using several methods such as `ReLU` activation function, max pooling and the `softmax` classifier. Figure 2.9 shows the AlexNet architecture. One can observe that the two CNN models are joined in parallel. Each model consists of a total of eight layers (5 convolutional layers and 3 fully connected layers). The two sub-sampling layers are 2×2 max-pooling layers. As mentioned before, another great feature of `AlexNet` is that it uses `ReLU` activation function. The training speed with `ReLU` activation function is much higher than training with `tanh` or `sigmoid`

activation functions (Krizhevsky et al., 2012).

2.5.3 VGG16

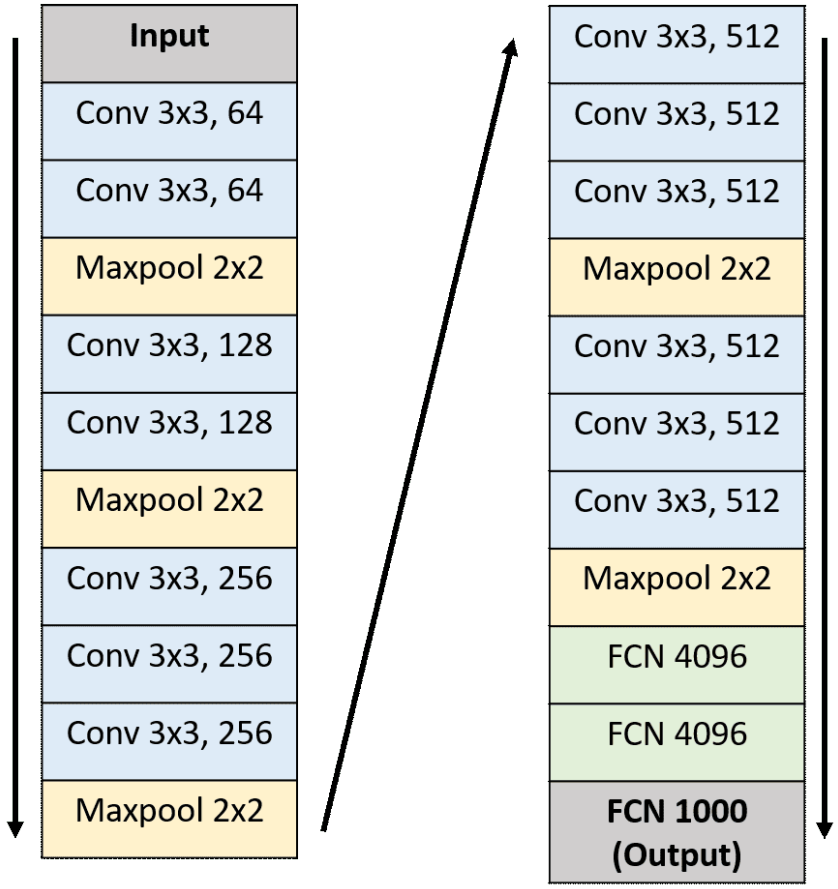

Figure 2.10: `VGG16` architecture. Two or three convolutional layers are stacked before sub-sampling. It consists of a total of 16 layers (13 convolutional layers and 3 fully connected layers). Max pooling is adopted for the sub-sampling.

The `VGGNet` (Simonyan and Zisserman, 2014) architecture was the second (Top-5 error rate was 7.3%) in the ImageNet challenge in 2014, with a slight difference in `GoogleNet` (Szegedy et al., 2015). The error rate of `GoogleNet` was 6.67%. However, the `VGG` model have been more popular because it is much simpler and more intuitive than `GoogleNet`. The figure 2.10 shows the architecture of one (`VGG16`) of the six `VGGNet` architectures. It consists of 16 layers (13 convolutional layers and 3 fully connected layers). Max pooling is used for the sub-sampling. As the networks become deeper, vanishing gradient problems arise even if `ReLU` is adopted. `VGG16` mitigates this problem by stacking several convolutional layers before sub-sampling.

2.6 Hyperparameters

When defining a CNN model to use for data learning, the architecture such as the size and the size of the kernel is the first thing one must consider. If the size of the kernel is too large, it will not properly extract the input's local feature. Thus, kernels of size $3 \times 3 \times n$ (where n is the number of channels of the input) are usually adopted. On the other hand, the number of kernels in each layer is often defined according to the available computing power used in the calculation. The initial convolutional layers do things like edge or color detection. These low-level features are sufficiently expressible with a relatively small number of kernels. As the layers go deeper, however, the features become increasingly difficult to distinguish, and the way the kernels represent each feature acdquires more variability.

In addition to this, there are hyperparameters to consider before the training step. The learning rate is one of the key hyperparameters which can affect CNN's performance. It is a variable that can adjusts the training speed. In classical optimization theory, the learning rate is the step size of the steepest descent method (Ruder, 2016). The batch size is also a critical hyperparameter. As it grows, it increases the amount of computation in each iteration, but it can estimate the gradient more accurately. The appropriate batch size can be a solution to this problem. Also, a different optimizer, loss function, or weight initialization method could be considered before the training step. Unfortunately, as it is impossible to calculate the optimal hyperparameters analytically, it is necessary to determine them empirically.

CHAPTER 3

Preparation

3.1 Introduction

This chapter explains the four preparatory steps required before CNN training for automatic velocity analysis. The first step is to define the input data. I define the semblance as an input data, allowing CNN to take the overall pattern of the semblance into account and return a single velocity value. The second step is the construction of training dataset. I first created common shot gather (CSG) using a custom velocity model and a finite difference method (FDM). Then, I perform a CMP sorting followed by the estimation of the semblance of each CMP and finally I construct the input data as a set of images that are paired to labels. For the label, I use the RMS velocity under an assumption that the RMS velocity and the stacking velocity are equivalent because the velocity model I use for training is composed of quasi-horizontal layers. The third step is to define the CNN architecture for training and

testing. I construct the CNN model based on the `LeNet-5` architecture. A more complex architecture could have been adopted. However, considering the relatively small number of classes (40) in which I have divided RMS velocity field, it is reasonable to adopt a non-overwhelmingly complex architecture like `LeNet-5`. I also add methods that boost the performance of the CNN model, such as ReLU and dropout. The last step of my study involve hyperparmeter tuning. In particular, I have optimized values of the `learning rate`, `batch size`, and `activation function` through the hyperparameter tuning.

3.2 Data definition

To use CNN, I define the semblance velocity analysis task as an image classification problem. One semblance panel contains one-dimensional velocity profile for two-way travel time axis. Therefore, I extract small patches so that each patch has one velocity value. In this case, however, it is impossible to distinguish a multiple reflection (Yilmaz, 2001) from a primary reflection because there is no information about the semblance overall pattern. Figure 3.1 shows three patches extracted from one semblance with the stacking velocity (red dots) at the center of each patch. These are obtained via manual velocity analysis. However, Figures 3.1 (b) and (c) display energy peaks at lower velocity than the optimal stacking velocity. The low veloicity event is a multiple. In other words, a reflection that has undergone more than one reflection in the subsurface. It is important to avoid selecting the multiple as it corresponds to a coherent noise (Yilmaz, 2001). Thus, when I train CNN using small patches,

the method might not be able to recognize a primary from a multiple. I solve this problem by adding the image of the entire semblance to images of small extracted patches. In other words, our training data consists of two images per time step; one capturing local features and the other the general trend of the velocity in the semblance panel.

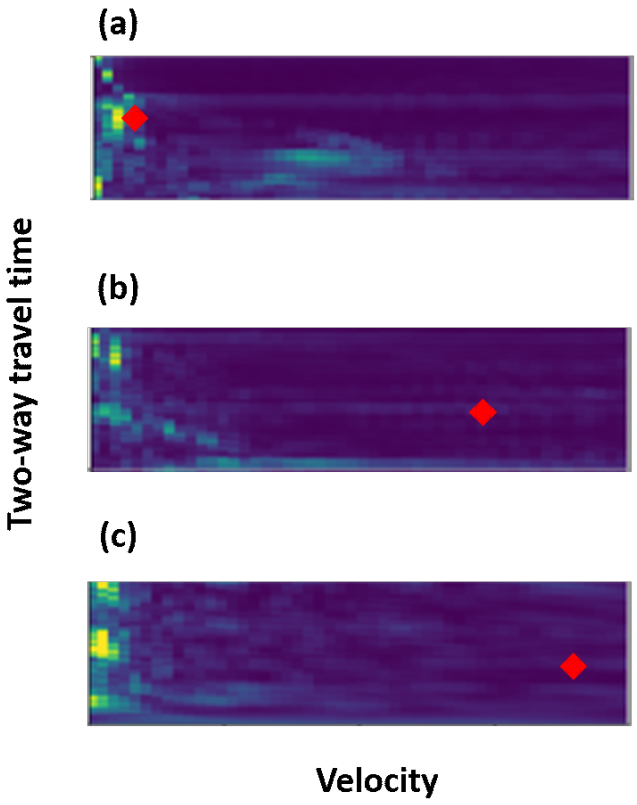

Figure 3.1: Three patches (v,t) extracted from the same semblance. Red dots indicate the stacking velocities at the center of each patch. These are from the manual velocity analysis. The red dot in (a) the first patch matches with the high-condensed energy while other dots in (b) the second and (c) third patches have higher velocities than the high-condensed energies.

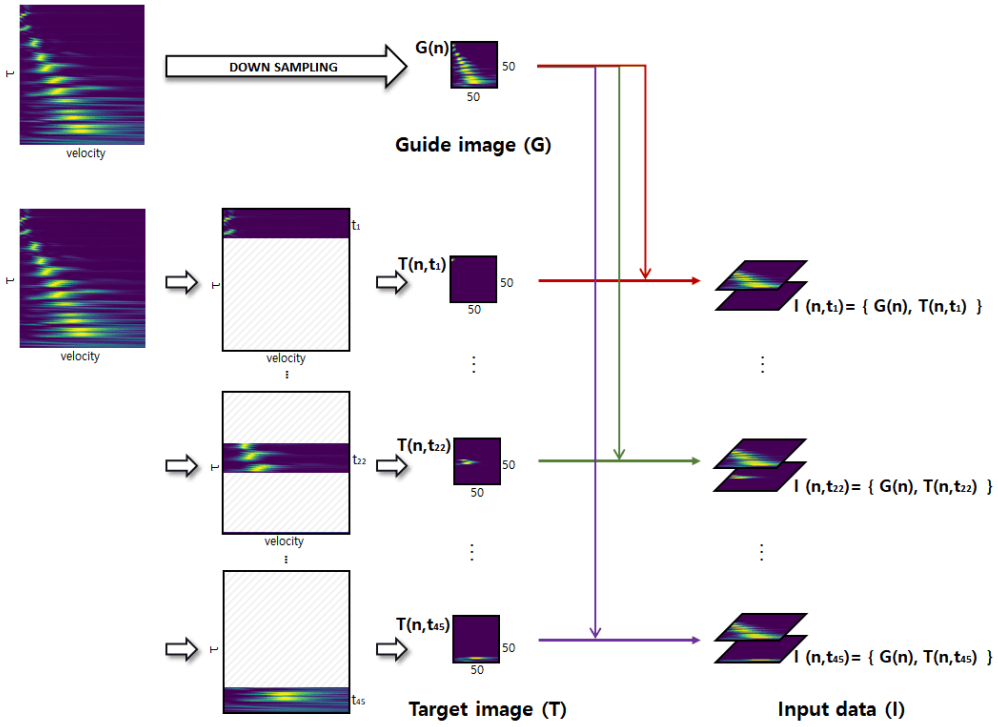

Figure 3.2: Input data definition. For each semblance, there is one Guide image (G) and 45 Target images (T). Then, Input data (I) pair each G_n and T_{n,t_m} to produce the input data (I_{n,t_m}), where n is the CMP number and the index $m = \{1, 2, 3..., 45\}$ corresponds to the intercept traveltime t_m.

Figure 3.2 shows the definition of the input data. The guide image (G) is representing the whole semblance. The target image (T) includes only the values in the specific t range where t is the zero offsets two-way travel time of the reflection hyperbola. Each image T represents the velocity at the middle of the t interval. I substitute all the values in the semblance by zero except for the values in the t interval of analysis. The number of target images T is 45 for each semblance. Thus, the input data are represented by pairs of

images denoted by G_n and T_{n,t_m}, where n indicates the CMP number and the index $m = \{1, 2, 3..., 45\}$ corresponds to the intercept traveltime t_m. I unify the size of all input data by compressing them into images of 50×50 pixels to reduce computational cost. The model trained with inputs G and T can pick a reasonable stacking velocity for each t because it allows the trained model to focus on a specific t time while it also considers the semblance's overall trend.

Regarding the label definition, I divide the semblance velocity axis into $K = 40$ compartments and define each compartment as one class. That is, one class represents a specific velocity bin. For example, if the velocity axis of the semblance ranges from 2000m/s to 4000m/s and if I assign classes with an increment of 50m/s, all value of velocity between 2000m/s and 2050m/s belong to the class 1. Similarly, velocities in the bin 2050m/s to 2100m/s belong to class 2 and so on. Labels can be represented in binary form. For instance, for the class 2 the label is given by the vector of length K, $\mathbf{p}_{n,m} = [0, 1, 0, 0, 0, \ldots, 0]$ where n is CMP number and m is intercept time. I match all input data with the corresponding label to construct the dataset for training. Our model outputs the vector $\hat{\mathbf{q}}[j]_{n,m}$, $j = 1 \ldots 40$. I adopt a Softmax classifier to convert $\hat{\mathbf{q}}_{n,m}$ to the probability of each class j being correctly identified

$$q[j]_{n,m} = \frac{\exp \hat{q}[j]_{n,m}}{\sum_{i=1}^{K} \exp \hat{q}[i]_{n,m}}. \tag{3.1}$$

Thus, each $q[j]_{n,m}$ represents a predicted probability for each velocity class j where each element of the vector, $q[j]_{n,m}$, is a number in the interval $[0, 1]$. The weights of the CNN are computed by minimizing the cost function that

measures the similarity of $\mathbf{p}_{n,m}$ and $\mathbf{q}_{n,m}$ which is equivalent to saying that CNN model should be able to predict the stacking velocities provided in the training step. I have adopted the cross-entropy cost function, which is usually applied to build CNN models because it leads to an algorithm with a high training speed (Nielsen, 2015). The cost function is expressed as:

$$J = -\sum_{n}\sum_{m}\sum_{j=1}^{K} p_{n,m}(j)\log(q_{n,m}(j))\,. \tag{3.2}$$

3.3 Training data construction

As mentioned earlier, I define the input data used for CNN training as the two images (G, T) taken from the semblance. I construct the CSG by performing forward modeling using the custom synthetic velocity models and the finite difference method (FDM) to build the training dataset.

The time domain acoustic wave equation for a two-dimensional isotropic media is given by

$$\frac{1}{v^2}\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + f\,, \tag{3.3}$$

where v is a P-wave velocity, u is the pressure, and f is a source term. I assume that exploration reflection seismology using single component data can model the wave propagation process at typical exploration scales (Kelly et al., 1976). Then, if I approximate the second order differential term respect to the time by the finite difference method

$$\frac{1}{v^2}\frac{u^{k+1}-2u^k+u^{k-1}}{(\Delta t)^2}=\frac{\partial^2 u^k}{\partial x^2}+\frac{\partial^2 u^k}{\partial y^2}+f^k \tag{3.4}$$

and

$$u^{k+1}=2u^k-u^{k-1}+v^2(\Delta t)^2\left(\frac{\partial^2 u^k}{\partial x^2}+\frac{\partial^2 u^k}{\partial y^2}+f^k\right), \tag{3.5}$$

where k is a time index and Δt is a time interval. In Equation 3.5, the second order differential terms respect to the two-dimensional space (x,y) are also approximated by the finite difference method as follows

$$u_{i,j}^{k+1}=2u_{i,j}^k-u_{i,j}^{k-1}+v_{i,j}^2(\Delta t)^2\left(\frac{u_{i-1,j}^k-2u_{i,j}^k+u_{i+1,j}^k-2u_{i,j}^k+u_{i,j+1}^k}{(\Delta x)^2}+\frac{u_{i,j-1}^k-2u_{i,j+1}^k}{(\Delta y)^2}+f_{i,j}^k\right), \tag{3.6}$$

where i and j are two-dimensional space indices. I also consider the discretization of the subsurface assuming $\Delta x=\Delta y=h$. Thus, the final form of the wave equation in discrete coordinates leads to the following time marching algorithm

$$u_{i,j}^{k+1}=2u_{i,j}^k-u_{i,j}^{k-1}+v_{i,j}^2(\Delta t)^2\left(\frac{u_{i-1,j}^k+u_{i,j-1}^k-4u_{i,j}^k+u_{i,j+1}^k+u_{i+1,j}^k}{h^2}+f_{i,j}^k\right). \tag{3.7}$$

Figure 3.3 shows an example of forward modeling with two layers velocity model using Equation 3.7. I use Keys method for imposing boundary conditions (Keys, 1985). Figure 3.3 clearly shows that the reflected wave is formed at the layer's boundary. By locating virtual receivers on the surface, one can

estimate the synthetic seismic data in the form of Common Shot Gathers (CSG). For an effective CNN model training, I create five velocity model that mimics a sedimentary geological structure (Figures 3.4). All models have a different number of layers and have a velocity distribution that increases with depth. As described above, I obtain CSGs for each velocity model using FDM and then generate the CMP gathers by conducting classical CMP sorting.

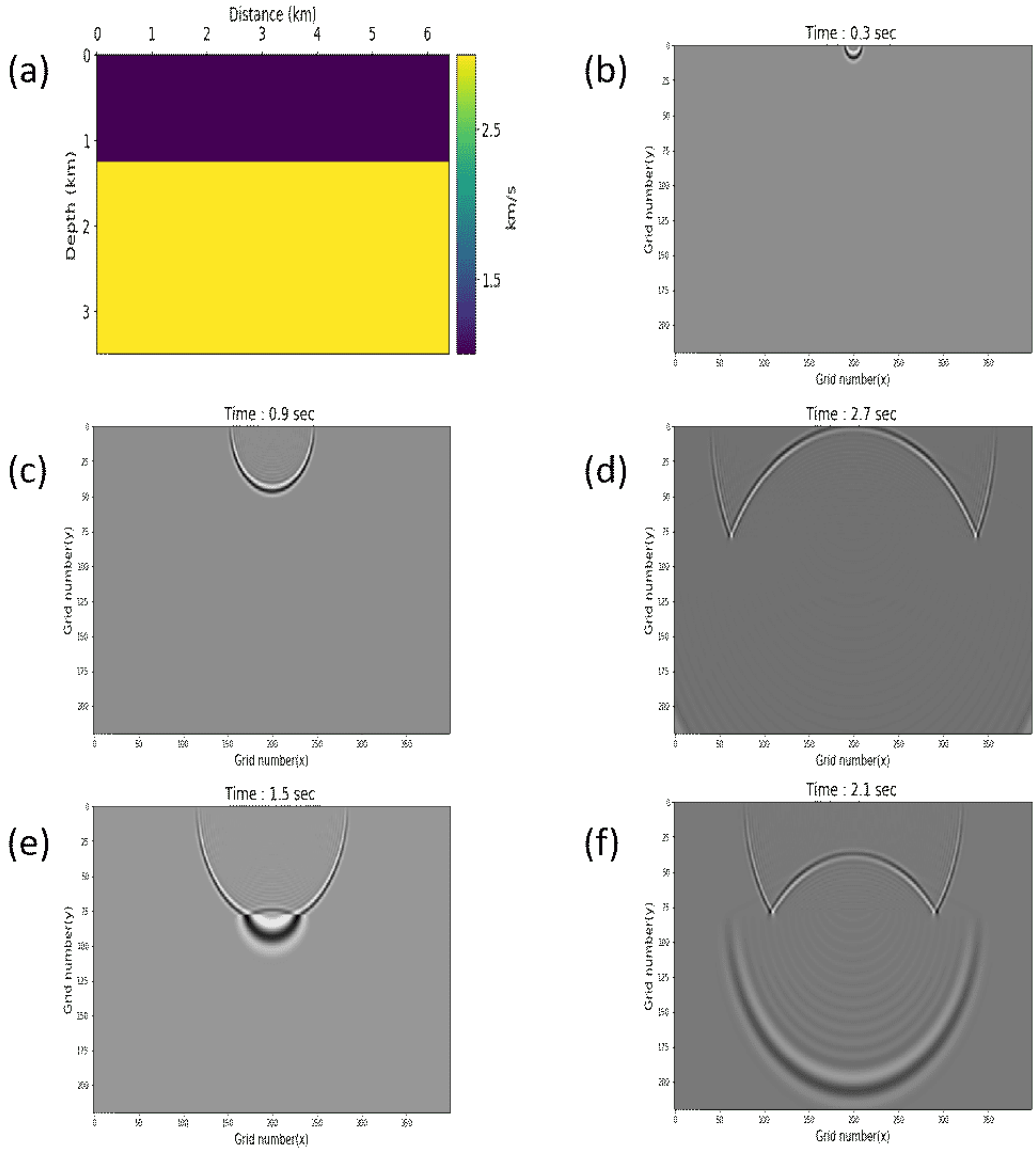

Figure 3.3: (a) The P-wave velocity model. The snapshots of the forward modeling at the time (b) 0.3 sec, (c) 0.9 sec, (d) 1.5 sec, (e) 2.1 sec, and (f) 2.7 sec. I use Keys method for boundary condition. It clearly shows that the reflected wave is formed at the layer's boundary.

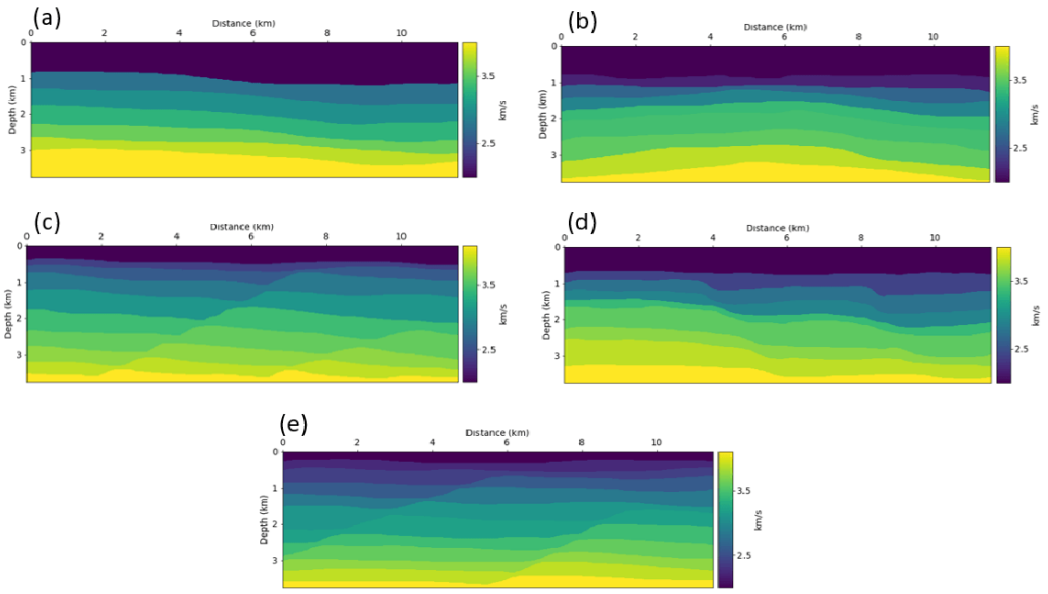

Figure 3.4: Five synthetic velocity models are used to generate the training data. All models have a different number of layers and the velocity distribution that increases with depth.

Figure 3.5 shows the workflow that I use to create the CNN training data. First, I simulate data from the velocity models by solving the acoustic wave equation via the Finite Difference Method. I also conduct CMP sorting and apply automatic gain control to the data to equalize amplitudes. Then, I compute the semblance panels for each model. I also normalize the semblance panels to a common predefined scale. For all custom models, the number of time samples of each record is 2850. The number of semblance panels is 70. The velocity for each semblance ranges from 2000 m/s to 4000 m/s. Finally, I construct the entire input dataset using the method mentioned in the last section. The total size of the input data is $15750 \times 50 \times 50 \times 2$,

where $15750 = 5 \times 70 \times 45$ is the number of input data and $50 \times 50 \times 2$ is the size of each input data. I also calculate the RMS velocity profiles at the CMP locations where I compute the semblance. Equation 3.8 shows how to calculate the RMS velocity at some two-way traveltime step τ.

$$V_{\mathrm{RMS}(\tau)} = \sqrt{\frac{\sum_{i=1}^{\tau} V_i^2 \Delta\tau_i}{\sum_{i=1}^{N} \Delta\tau_i}} \tag{3.8}$$

where V is an interval velocity. As mentioned earlier, I assume that the RMS velocity and the stacking velocity are equivalent because the five synthetic models used for training are composed of quasi-horizontal layers. All the RMS velocities are mapped into $K = 40$ classes. The total size of the label is 15750×40. I match all input datasets with all labels to create the entire training dataset.

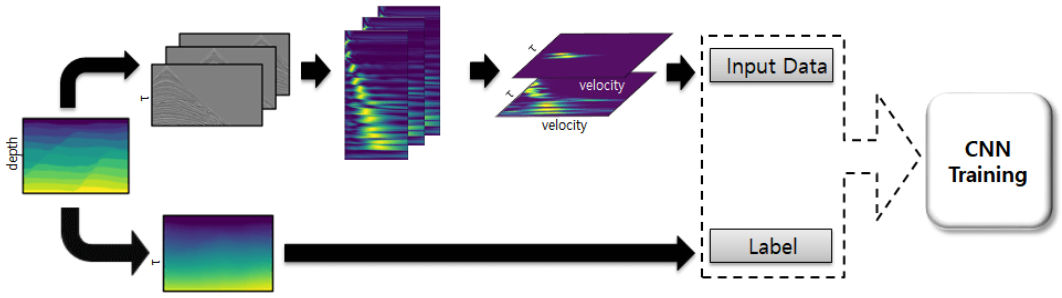

Figure 3.5: Workflow displaying the training step. I solve the acoustic wave equation via the Finite Difference method to compute synthetic shot gathers. I calculate the semblance panels for each CMP location after CMP sorting and automatic gain control. To compute the labels, the RMS velocities are computed analytically from the velocity models.

3.4 CNN architecture adoption

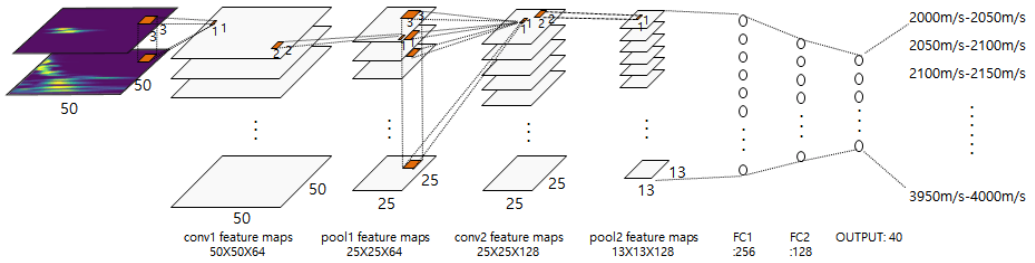

Figure 3.6: The CNN architecture I adopt. It is based on `LeNet-5`. It contains two convolutional layers (`Conv1`, `Conv2`) and two fully-connected layers (`FC1`, `FC2`) followed by the softmax classifier. The model outputs 40 numbers in one array for each input data. Each output represents the probability of each class.

For efficiency, I decide to use the architecture, which is based on an existing well-known architecture, rather than constructing the CNN architecture from scratch. However, architectures with a deeper neural network such as `AlexNet` or `VGG16` are designed to solve relatively complex problems such as the ImageNet challenge (Deng et al., 2009). Because our classification problem has only 40 classes and the image size of the input is relatively small, I decided to use the LeNet-5 architecture as the base architecture for the automatic velocity analysis. Figure 3.6 shows the CNN architecture adopted for this thesis. The architecture consists of two convolution layers named (`Conv1`, `Conv2`) and two fully-connected layers (`FC1`, `FC2`) followed by a softmax-classification layer. `Conv1` has 64 filters of size $3 \times 3 \times 2$ size and `Conv2` has 128 filters with

$3 \times 3 \times 64$ size. After both `conv1` and `conv2`, I applied an activation function which increases the non-linearity of the model (Nair and Hinton, 2010; Krizhevsky et al., 2012), and a max-pooling of both size and stride 2×2. In the `FC1` layer, the feature maps are flattened to 256 elements. The `FC2` layer has 128 elements. I also apply an activation function to both `FC1` and `FC2` layers and also use `dropout` to both layers to prevent overfitting (Srivastava et al., 2014). Through the softmax classification layer, I predict the stacking velocities. For the optimization, I adopt the Adaptive Moment Estimation (Adam) optimizer (Kingma and Ba, 2014) implemented in `Tensorflow` (Abadi et al., 2016).

3.5 Hyperparameter tuning

As described in **Chapter 2**, I need to obtain optimal hyperparameters to obtain a better performance of the CNN architecture. Based on the model I described (Figure 3.6), I try to find the optimal values of three hyperparameters: `the learning rate, batch size, and activation function`. The evaluation method for the model is also important to find the optimal parameters. I evaluated the performance of the model using three methods. The first is the trend of the loss as the training progresses. The loss of good models is fast to converge and stable in changing directions. The second method is to measure the accuracy of the predicted value by the trained model. The final method is to check the F1 score based on the concept of precision and recall (Powers, 2011).

3.5.1 Learning rate

The first parameter I try to optimize is the `learning rate`. For tuning process, I unify every conditions except for the learning rate. Activation function is `ReLU`, and the `batch size` is 256. Every experiment uses the same subset of the training data I build for, and the same total number of epoch (number of epoch is 100). I test total five learning rates $(0.0001, 0.0005, 0.001, 0.005, 0.01)$. Figure 3.7 shows the loss trends of the five models that use five different `learning rates`. As we can see, `learning rate` 0.001 (gree line in Figure 3.7) shows the best result of convergence. Table 3.1 shows other measurements (Loss, Accuracy, Precision, Recall, and F1 score) of the five cases. As with the above results, every evaluation result tells that 0.001 is the best learning rate for our model and the dataset.

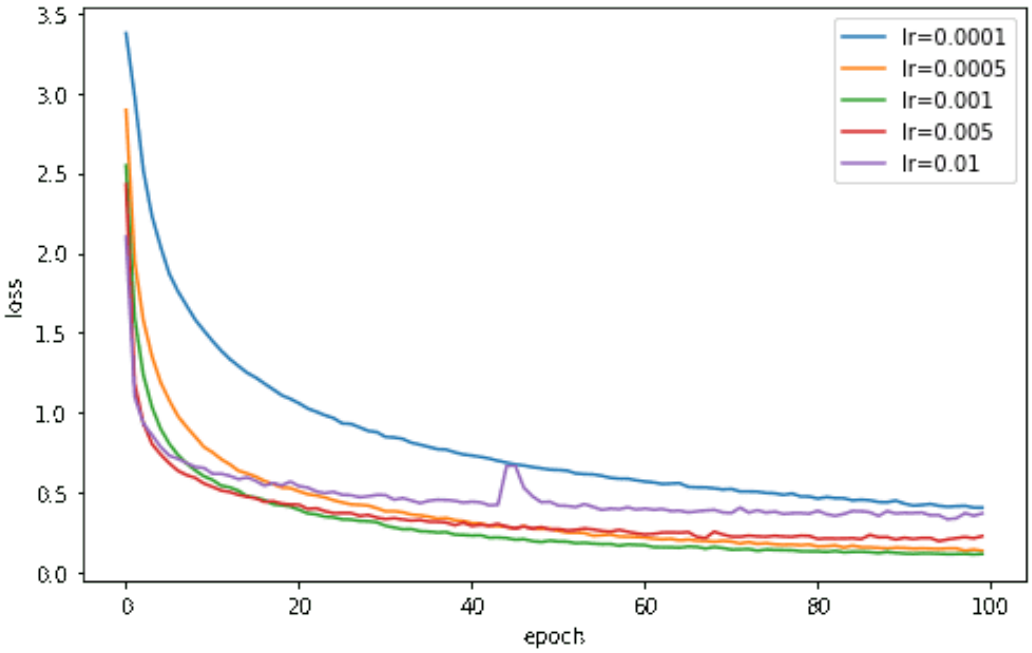

Figure 3.7: Loss trends for five models with different learning rates. `Learning rate` 0.001 (green line) shows the best result of convergence.

Learning rate	Loss	Accuracy	Precision	Recall	F1 score
0.0001	0.4035	0.8363	0.8555	0.8105	0.8324
0.0005	0.1324	0.9478	0.9512	0.9458	0.9485
0.001	0.1135	0.9566	0.9584	0.9549	0.9566
0.005	0.2247	0.9203	0.9240	0.9149	0.9194
0.01	0.3679	0.8716	0.8856	0.8575	0.8713

Table 3.1: Evaluation results (Loss, Accuracy, Precision, Recall, and F1 score) for five models with different Learning rates. All the results are from the model after 100 epoch training process. Data used for accuracy is the same as the training data. `Learning rate` 0.001 shows the best performance in every evaluations.

3.5.2 Batch size

The second parameter I tune is the `batch size`. In the starting model, `ReLU` is used as the activation function, and the `learning rate` is 0.001. Also, for every case I use the same dataset for model training. The total epoch number is 100 for every test. I test five batch sizes $(16, 64, 128, 256, 512)$, and Figure 3.8 and Table 3.2 shows the results. Unfortunately, I have not found any noticeable correlation between batch size and the performance of the model. The model with 512 batch size has the lowest final loss, and the other indicators

are also higher than average, but there is no significant difference.

In this test, as the batch size is related to the training time, I also measure the average time per epoch. As one increases the batch size, the time per epoch decreases (See the second column of Table 3.2). Thus, it is better to use a larger batch size to the extent that is allowed by the GPU (or CPU) memory. In this project, I choose a batch size of 256 because of both the insufficient GPU memory and the relatively small difference of time per epoch between using a 256 or a 512 batch size.

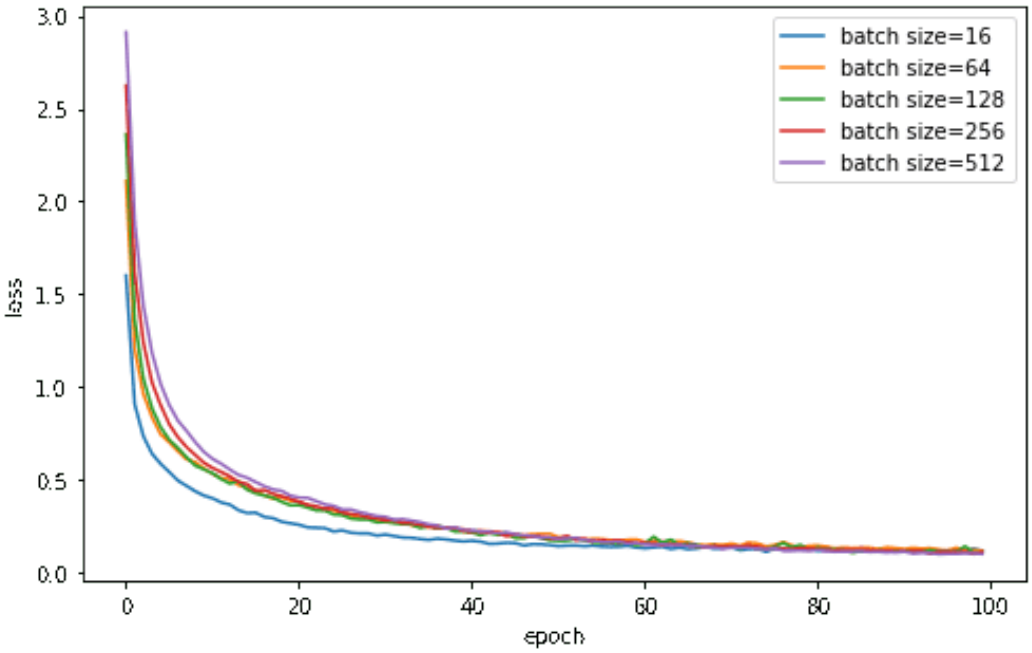

Figure 3.8: Loss curve for five models with different batch sizes $(16, 64, 128, 256, 512)$. I have not found any noticeable correlation between batch size and the performance of the model.

Batch size	Time per epoch	Loss	Accuracy	Precision	Recall	F1 score
16	16.5 sec	0.1158	0.9655	0.9678	0.9629	0.9653
64	4.3 sec	0.1164	0.9573	0.9599	0.9556	0.9577
128	2.1 sec	0.1077	0.9610	0.9631	0.9594	0.9612
256	1.9 sec	0.1108	0.9587	0.9610	0.9569	0.9590
512	1.7 sec	0.0992	0.9625	0.9637	0.9612	0.9625

Table 3.2: Evaluation results (Time per epoch, Loss, Accuracy, Precision, Recall and F1 score) for five models with different batch size. All the results are from the model after 100 epoch training process. Data used for accuracy is the same as the training data.

3.5.3 Activation Function

The final parameter I tune is the `activation function`. In the starting model, I use a `batch size` of 256 and a `learning rate` of 0.001. Also, every case adopts the same dataset and the same epoch number. I test four activation functions (`Sigmoid, tanh, ReLU, Leaky ReLU`), and the results are shown in Figure 3.9 and Table 3.3. As the training process of the model with the `Sigmoid` activation function does not work properly (blue line in Figure 2.3), I exclude it when I evaluate the models. I find that the `Leaky ReLU` shows

the best performance among three different activation functions. It shows the lowest loss and the highest F1 score. The loss curve (Figure 3.9) also shows that the `Leaky ReLU` is the optimal activation function for the dataset and the model I used. However, as it requires a higher computational cost than the `ReLU`, and the difference in performance is not significant, I adopt `ReLU` as the activation function.

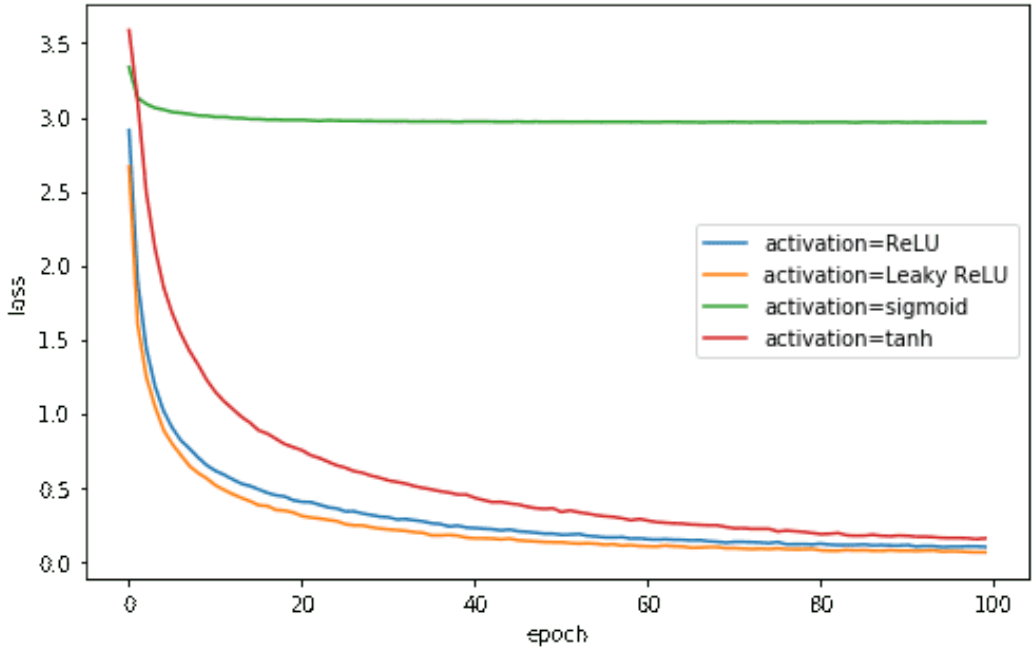

Figure 3.9: Loss curve for four models with different activation functions (`Sigmoid, tanh, ReLU, Leaky ReLU`).

Activation function	Loss	Accuracy	Precision	Recall	F1 score
tanh	0.1568	0.9451	0.9473	0.9428	0.9450
ReLU	0.0992	0.9625	0.9637	0.9612	0.9625
Leaky ReLu	0.0631	0.9766	0.9769	0.9763	0.9766

Table 3.3: Evaluation results (Loss, Accuracy, Precision, Recall, and F1 score) for three models with different activation functions. All the results are from the model after 100 epoch training process. Data used for accuracy is the same as the training data.

3.5.4 Final CNN architecture

The final CNN architecture is constructed after conducting the hyperparameter tuning for the `learning rate, batch size, and activation function`. In our final architecture, I choose a learning rate 0.001, batch size 256 and `ReLU` activation function. I conduct the last comparison with the other popular model (`VGG16`) to find out if the performance of our custom architecture is reasonable for our dataset. For the test, I change the output layer length of `VGG16` from $1,000$ to 40. Also, I use the same dataset and the same epoch number for the test. Figure 3.10 and Table 3.4 show the results. Every evaluation result represents `VGG16` is better than our custom architecture. This result is reasonable because `VGG16` consists of total 16 layers and $22,693,064$

parameters (our custom architecture has $4,831,976$ parameters). As a result, the `VGG16` model takes about four times as much training time per epoch as compared to the custom model. I conclude that the use of the custom model is reasonable because the difference in performance is not significant compared to the amount of computation required.

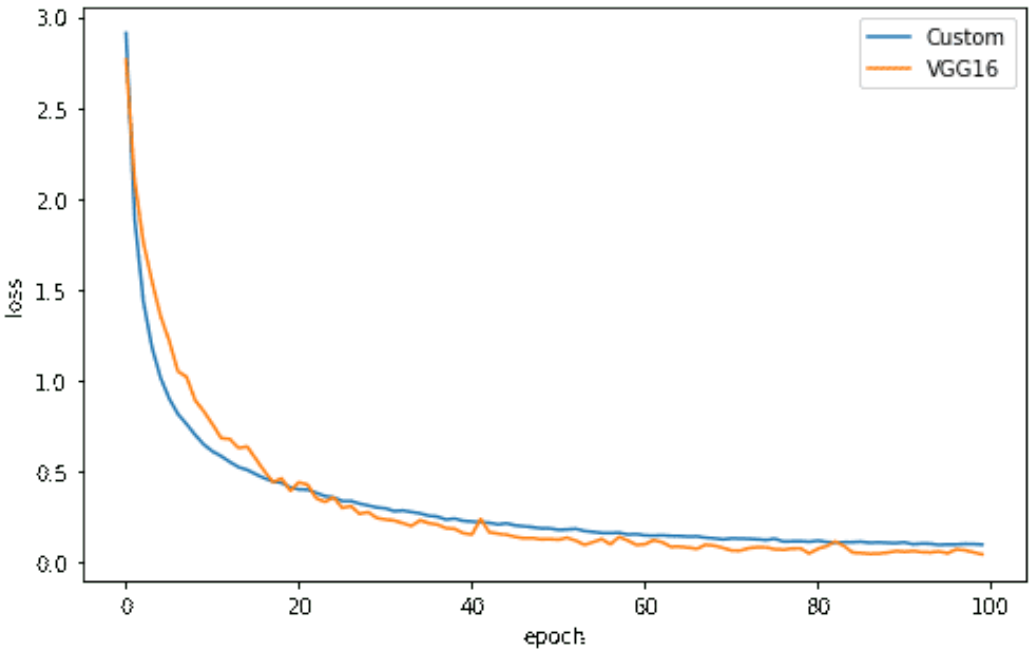

Figure 3.10: Loss trends for two different models: custom architecture and VGG16.

Architecture	Time per epoch	Loss	Accuracy	Precision	Recall	F1 score
Custom	1.7 sec	0.9451	0.9451	0.9473	0.9428	0.9450
VGG16	6.4 sec	0.9915	0.9919	0.9637	0.9911	0.9915

Table 3.4: Evaluation results (Time per epoch, Loss, Accuracy, Precision, Recall and F1 score) for two different models (Custom and VGG16). All the results are from the model after 100 epoch training process. Data used for accuracy is the same as the training data.

CHAPTER 4

Training and Testing

4.1 Introduction

This chapter firstly describes the actual training process with the training dataset obtained earlier. Also, I describe Transfer Learning that can be used as an alternative when training data is insufficient. Once one obtains the trained model, it is tested with a total of three datasets that are not included in the training dataset. The first dataset is a custom velocity model with features similar to the training dataset, and the trained model predicts reasonable velocities without Transfer Learning. Secondly, I conduct tests with the Marmousi dataset. In this case, however, since the features of the semblance obtained from the Marmousi dataset are much different from those of the training data I used, the trained model does not predict the correct velocities. In this case, I adopt Transfer Learning to solve the aforementioned problem. Finally, I test the proposed automatic velocity analysis with a ma-

rine dataset obtained from the Gulf of Mexico. I also use Transfer Learning, and compare the results to those obtain via manual velocity analysis. For completeness, I also estimate stack images of the Gulf of Mexico dataset with velocities that were manually derived and estimated by the proposed CNN scheme. The main results of this chapter are that a velocity field derived by CNN can deliver similar quality image to those estimated via manual velocity analysis.

4.2 Base model training

As I mentioned in **Chapter 4**, the total number of the input for training is 15750 and each input size is $50 \times 50 \times 2$. I randomly initialize all the weights of the CNN and train the model with randomly shuffled datasets. The total number of the epoch is 300. Figure 4.1 shows the loss value plot for the epoch. I use a computer equipped with a GTX 1060 graphic card to train the model. Training takes about 10 hours. I call the trained model as the 'Base model'.

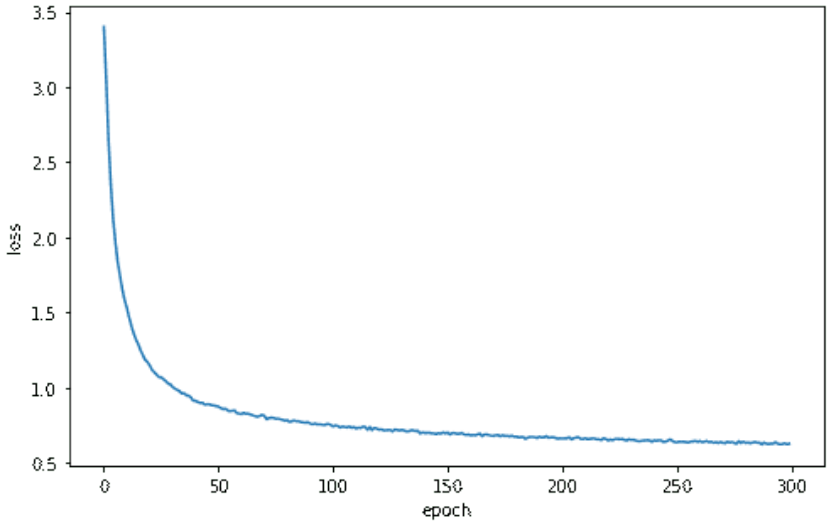

Figure 4.1: Loss trend for the base model training.

4.3 Transfer Learning

Even though our model is trained as expected, it is common to have insufficient data for training. Let us assume a task X and a trained CNN model through labeled data A. We expect that the trained model can predict the correct label of the new data B in the task Y, which is similar to X. However, this scenario is only possible if the data A is large enough to reflect both task X and Y or the data B has features that are similar to those of the data A. Unfortunately, there are not enough labeled data for training in many of our real seismic data processing applications. This data-hungry problem also occurs in our semblance analysis problem, which requires manual picking of velocities to obtain the labeled data. Thus, I use Transfer Learning when I

have an unsatisfying result with the original model trained with the synthetic data (base model). Figure 4.2 shows the concept of Transfer Learning. The base model (trained with labeled data A) can be updated with a small portion of the target data B to improve the predictability of Y.

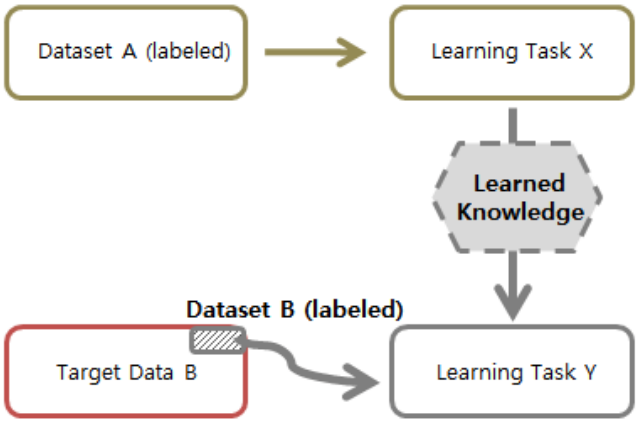

Figure 4.2: The concept of Transfer Learning.

4.4 Numerical examples

4.4.1 Custom dataset

The base model (without Transfer Learning) is first evaluated with a simple synthetic case. I use the velocity model (Figure 4.3), which includes similar features to the training data for testing. In this case, the base model can

predict the velocity properly. The testing data are obtained in the same way as the base model training data. The test result shows 42% of the accuracy and most predicted values of the remaining 58% outputs have a similar value to the label. This result is acceptable considering the number of classes is 40.

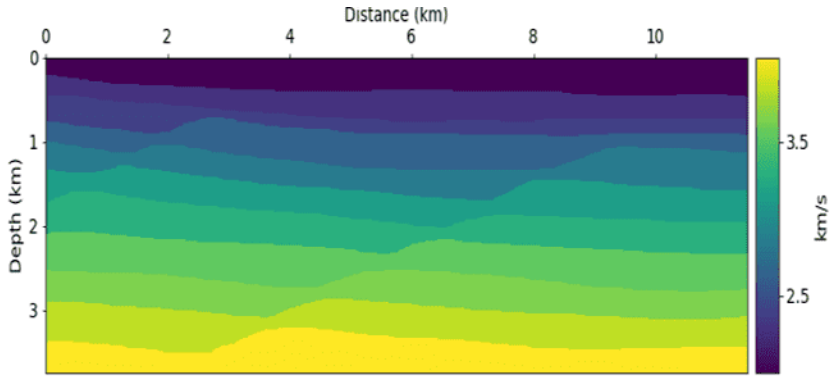

Figure 4.3: The velocity model used for testing. This model was not used in the training stage.

Figure 4.4 shows the label model and Figure 4.5 shows the predicted velocity model. Although the predicted velocities do not match exactly with the label model, it can be seen that they are similar. I performed NMO correction and stacking using both true RMS velocities and the predicted velocities. Figure 4.6 (a) shows the predicted velocity (red line) and the true RMS velocity (blue line) superimposed on the semblance panel. The NMO corrected result (Figure 4.6 (d)) using the predicted velocity is almost identical to the result (Figure 4.6 (c)) obtained by the true RMS velocity. There are also some parts with a relatively high error, especially in the shallow part. In those areas, the predicted velocity is slightly higher than the true RMS velocity. The NMO

correction with the velocity higher than the true RMS velocity causes under-correction, as shown in Figure 4.6 (d). Finally, I apply the NMO correction to all CMP gathers and stack them. Figure 4.7 shows the stack sections for both the true velocity model (Figure 4.7 (a)) and the predicted velocity model (Figure 4.7 (b)). As mentioned above, the trained model predicts a shallow velocity field that is higher than the true one. Thus, I can also observe the relatively large error in the shallow part of the stack (Figure 4.7 (c)). I confirm that the trained model can output a consistent velocity by comparing the stack section to the near-offset section (Figure 4.7 (d)).

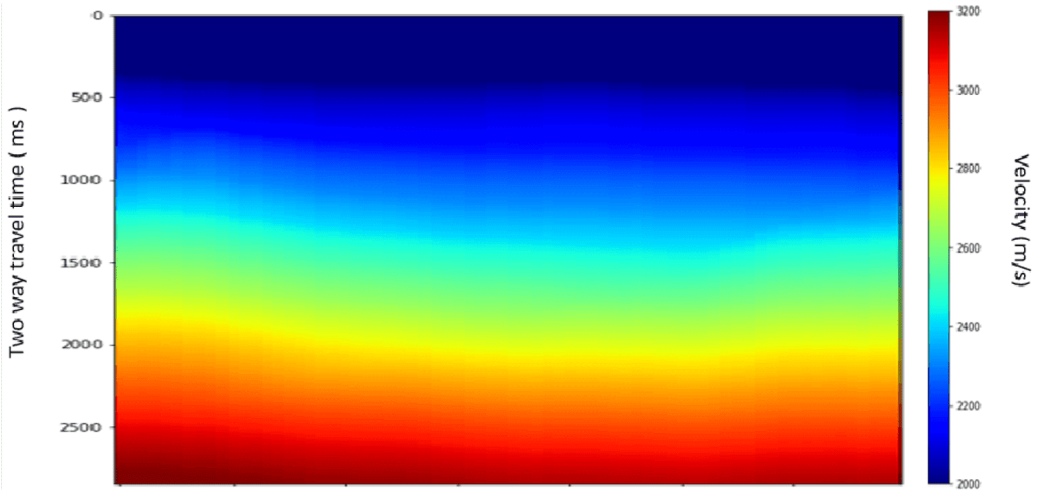

Figure 4.4: The label velocity model.

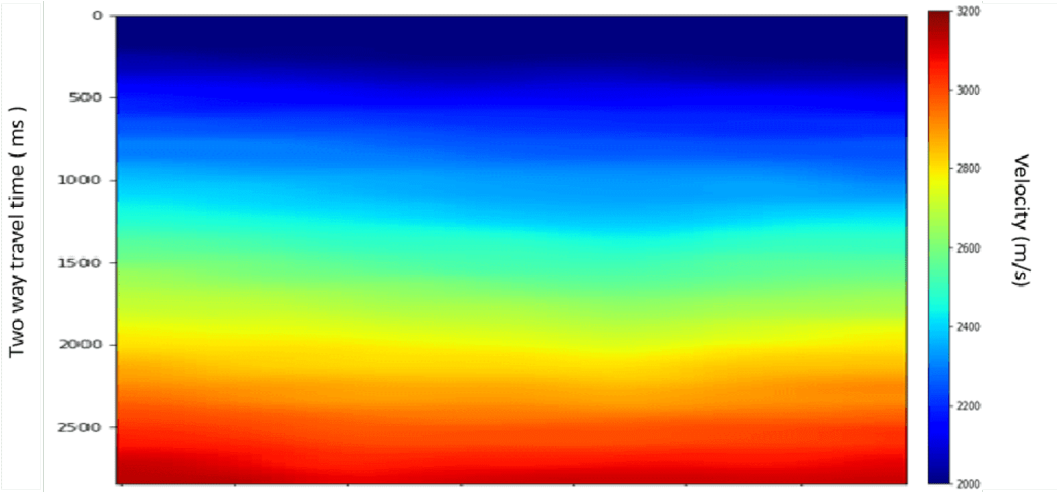

Figure 4.5: Velocity field predicted via CNN.

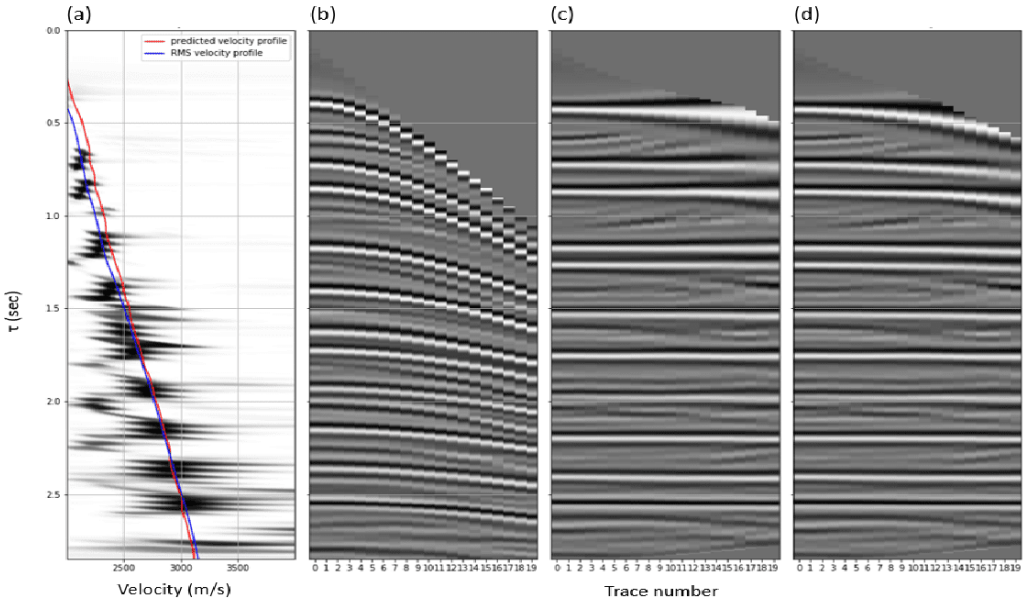

Figure 4.6: (a) Predicted velocity (red line) and true RMS velocity (blue line) are shown in conjunction with the semblance. (b) NMO correction corresponding to the CMP gather. (c) The results of NMO correction with the true velocity. (d) The predicted velocity are shown as well.

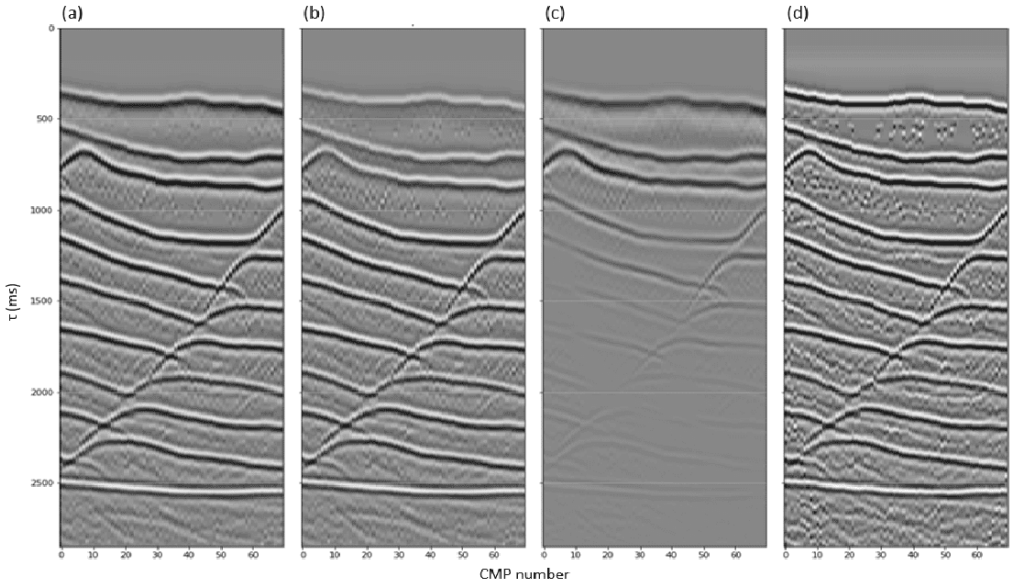

Figure 4.7: Stack sections. (a) Stack section obtained with the true velocity model. (b) Stack section obtained via the velocity model predicted by CNN. (c) Difference panel ((b) minus (a)). (d) The near offset traces.

4.4.2 The Marmousi dataset

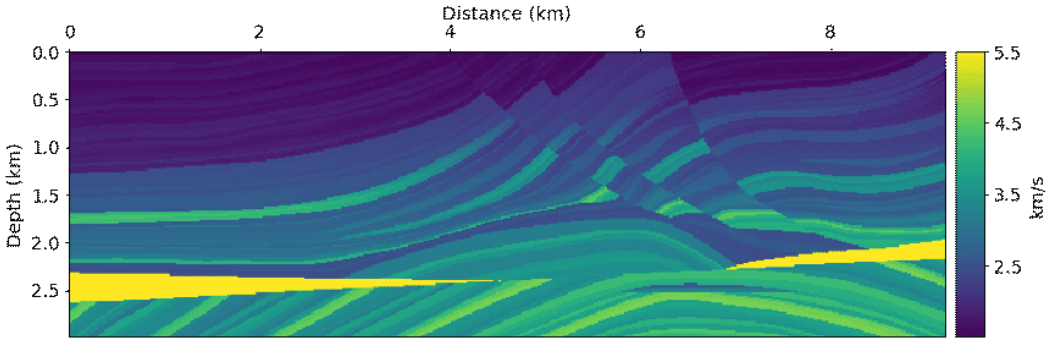

Figure 4.8: The Marmousi velocity model.

For the next synthetic example, I use the Marmousi model (Figure 4.8) to test the performance of automatic velocity analysis method. In this experiment, the total number of obtained CMP gather is 385. Thus, the total number of input data is 17325(385×45), and each data size is $50 \times 50 \times 2$. I first test this dataset with the base model. However, the result is unsatisfactory because the dataset obtained from the Marmousi model is much different from the training dataset I adopt. Thus, I use Transfer Learning. As I mentioned in chapter 'Transfer Learning', I take a small portion of the target data (the Marmosusi dataset) to update the base model. I test two cases by selecting the different number of Transfer Learning data, 19, and 38 semblances out of the entire dataset (385 semblances). All data are selected at regular intervals. Thus, the total numbers of the datasets for each case are 855(45×19) and 1710(45×38), respectively.

Number of the semblances	Data size	Time for Transfer Learning
19 (out of 385)	$855 \times 50 \times 50 \times 2$	83 sec
38 (out of 385)	$1710 \times 50 \times 50 \times 2$	171 sec

Table 4.1: Information of the model after Transfer Learning with 19 semblances (second row) and 38 semblances (third row).

Information about Transfer Learning is shown in Table 4.1. I perform the

Transfer Learning until the loss value is less than 0.05 or until the epoch reached 500. Transfer Learning takes 83 seconds and 171 seconds, respectively. This is a time that can be ignored when compared with the base model training, which takes about 10 hours. Figure 4.9 shows the loss trends of each case for the epoch. The red line indicates the case which uses 19 semblances for Transfer Learning while a blue line represents 38 semblances case. I would like to mention that the convergence speed of 38 is faster than that of 19 at the beginning of transition learning. Since the Marmousi model has a severe lateral variations, the more data used for Transfer Learning, the more effective learning can be. Figure 4.10 shows the label model (Figure 4.10 (a)), the predicted results of both 19 (Figure 4.10 (b)) and 38 (Figure 4.10 (c)) cases. It is clear that the predictive model of 38 case is closer to the label model than the model of 19 case.

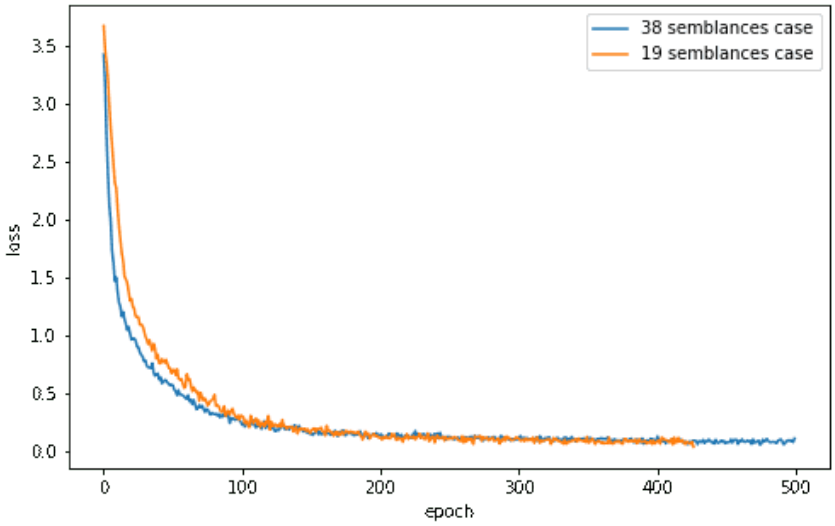

Figure 4.9: Loss trend for Transfer Learning using the Marmousi dataset.

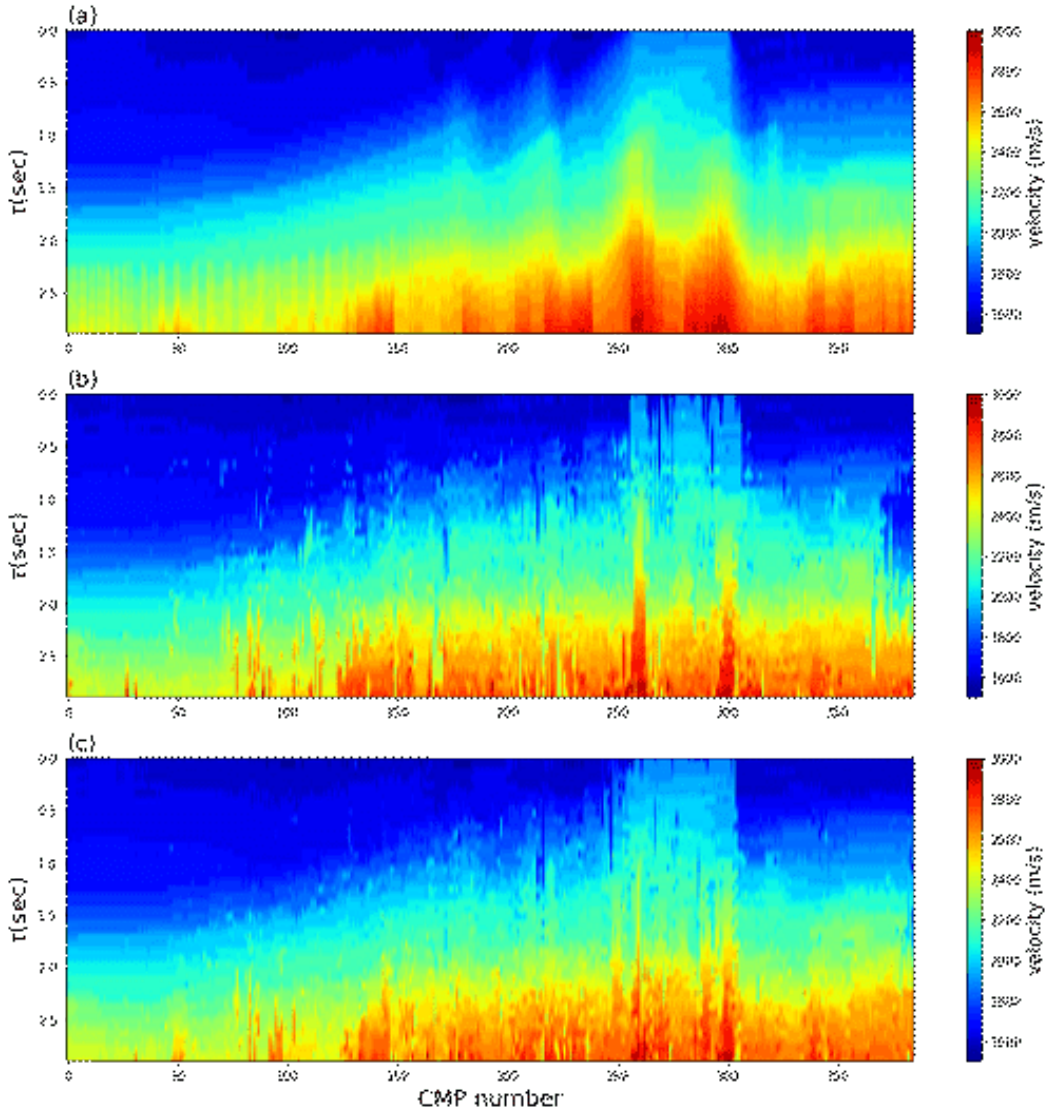

Figure 4.10: (a) Label (true) velocity model, (b) the predicted velocity model from Transfer Learning CNN with 5% of target data, (c) the predicted velocity model from Transfer Learning CNN with 10% of target data.

4.5 Field data example

Finally, I test our automatic velocity analysis method with a marine dataset from the Gulf of Mexico. These datasets have been provided by Western Geophysical, and Table 4.2 shows information about the data. The Gulf of Mexico dataset includes a salt body that causes multiples to be problematic. This is because the primaries below the salt body are normally weak, and the primary-multiple ratio is low.

CMP number	200-1599
Shot interval	87.5 ft
Number of samples per trace	1751
Time sampling interval	4 ms

Table 4.2: Parameters information of the Gulf of Mexico dataset.

Figure 4.11 shows the near-offset traces of a dataset from the Gulf of Mexico (Verschuur and Prein, 1999). The dataset is contaminated with free-surface multiples. Given that the base model was trained with synthetic data and that our real marine dataset is quite different from the synthetic data, I have to adopt Transfer Learning. In other words, I extracted ten semblance panels

(out of 1400) and velocity profiles (dashed lines in Figure 4.13 and Figure 4.14) which were obtained via manual velocity analysis and used them for Transfer Learning.

The manual analysis velocity field (Figure 4.13) is created using linear interpolation with velocity analysis of every 50 CMP gathers. Transfer Learning, in this case, takes only 45 seconds. As mentioned earlier, the time required for Transfer Learning is negligible compared to the time it takes to train the base model.

Number of semblance panels	Data size	Time for Transfer Learning
10 (out of 1400)	$450 \times 50 \times 50 \times 2$	45 sec

Table 4.3: Information pertaining Transfer Learning for the Gulf of Mexico dataset.

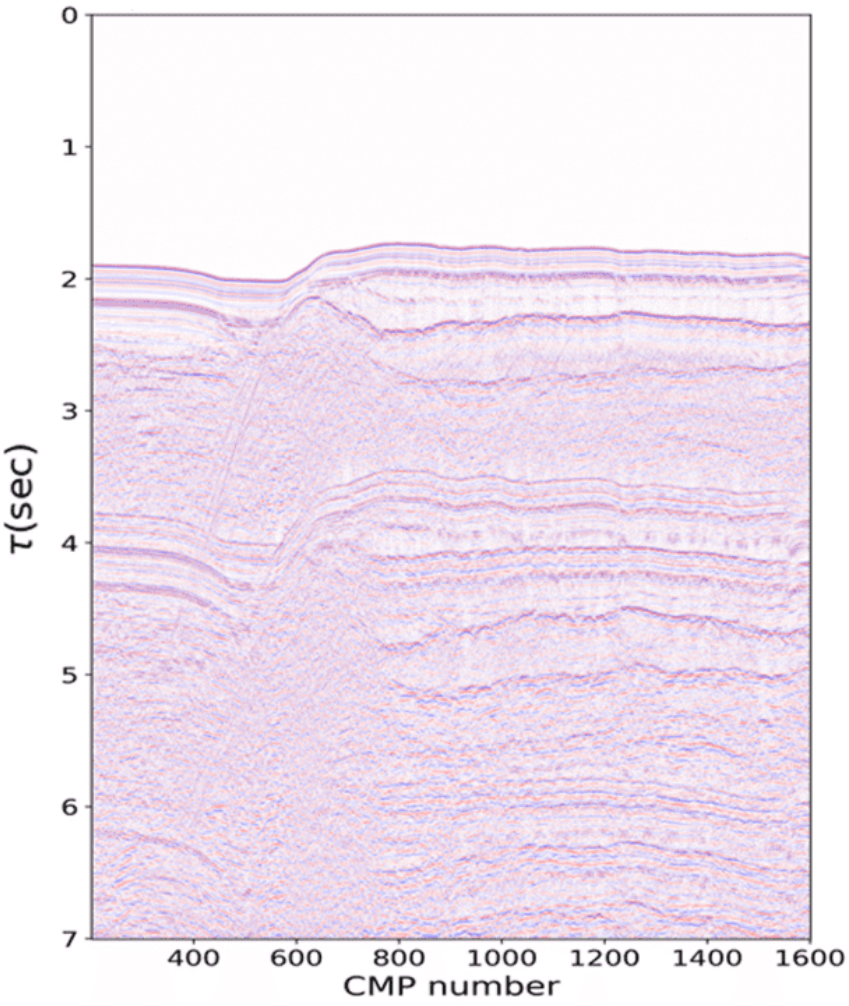

Figure 4.11: Near-offset traces of a dataset from the Gulf of Mexico. Data includes the salt body from position CMP 700 to CMP 1600.

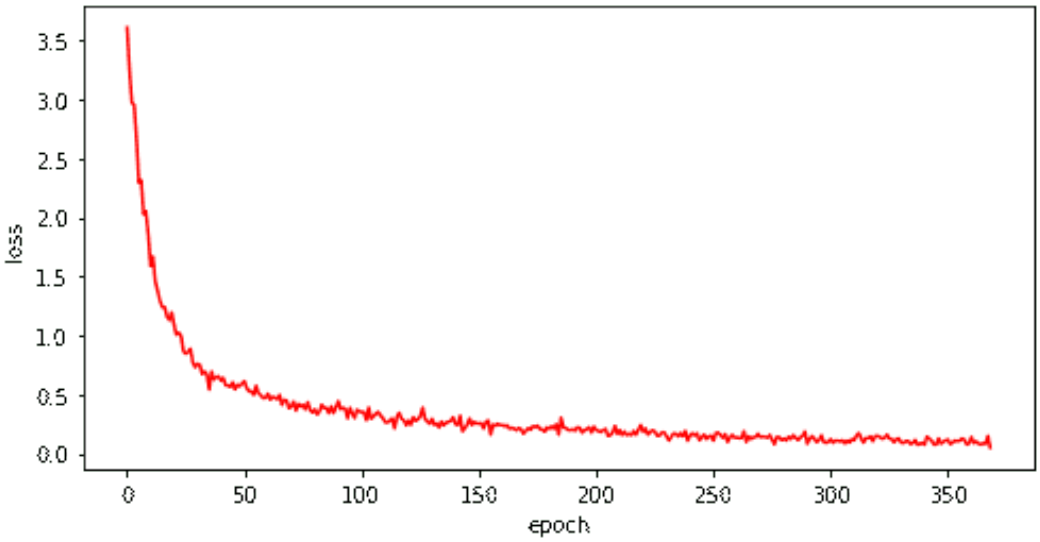

Figure 4.12: Loss trend for Transfer Learning using the Gulf of Mexico dataset.

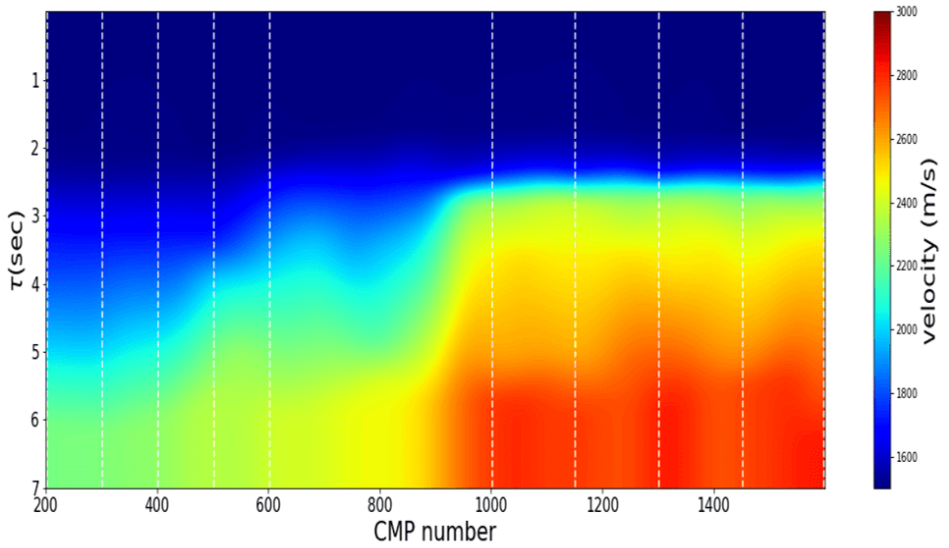

Figure 4.13: The velocity field created using linear interpolation with velocity analysis at every 50 CMP gathers. Dashed lines are indicating the location where I have got the data for Transfer Learning (CMP: 202, 302, 402, 502, 602, 1002, 1152, 1302, 1452, 1599).

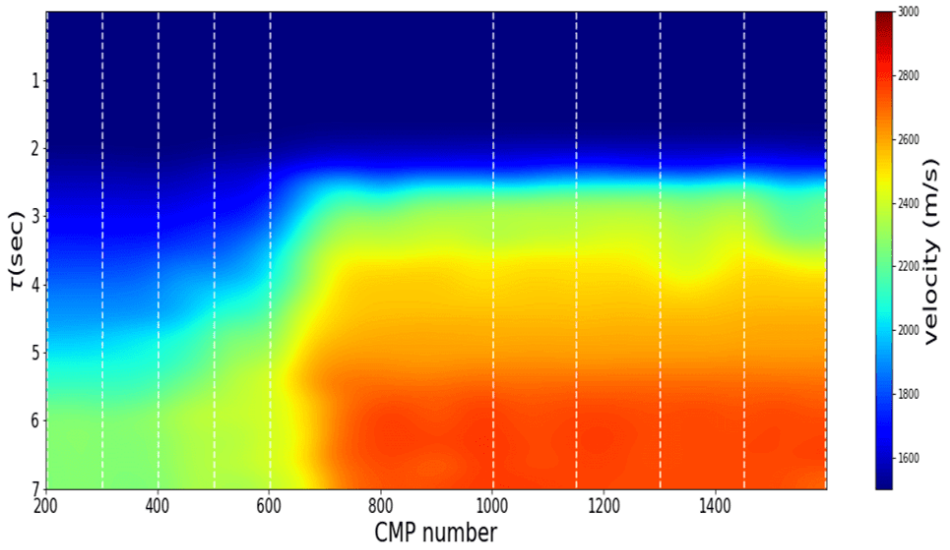

Figure 4.14: The velocity field predicted by CNN. I use a total of ten semblances and velocity profiles (CMP: 202, 302, 402, 502, 602, 1002, 1152, 1302, 1452, 1599) to perform Transfer Learning. Dashed lines are indicating the data's CMP location for Transfer Learning.

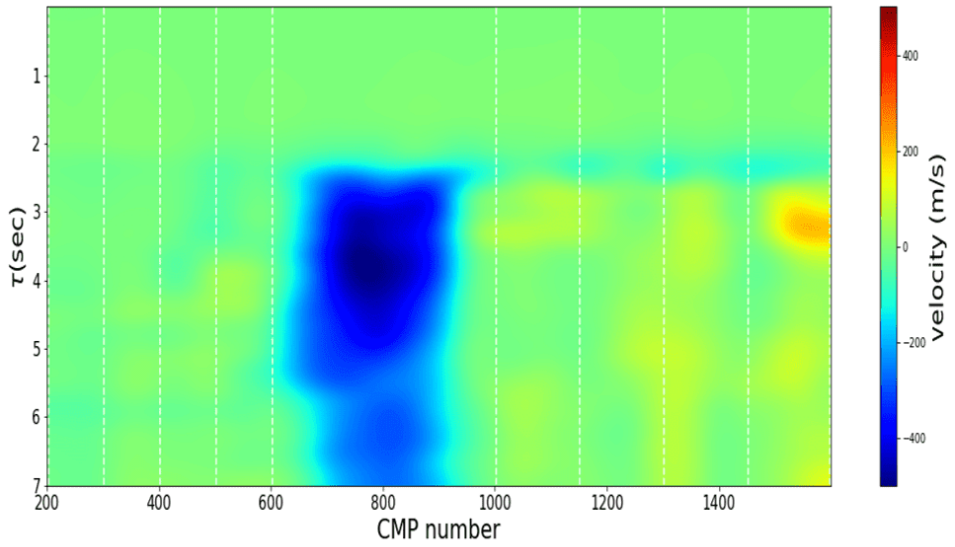

Figure 4.15: The difference in velocity field (Figure 4.13 minus Figure 4.14).

As we can see in Figure 4.11, there is a severe lateral structural variation between CMP 600 and CMP 1000, which can make the manual velocity analysis more difficult. Thus, when I choose the locations of the dataset for Transfer Learning, I exclude this area to check if the CNN model can provide a satisfactory solution. Five velocity profiles are taken from areas outside the tabular salt body (CMP: 202,302,402,502,602), and another five velocity profiles are taken from CMPs above the tabular salt body (CMP: 1002,1152,1302,1452,1599). I apply Transfer Learning to the base CNN model to predict the stacking velocity for all CMPs. After Transfer Learning, I compute the semblance of each CMP (the data contained 1400 CMPs) and then I use the estimated semblance panels as input to our network to obtain the

stacking velocities for each CMP. Prediction takes a few seconds, and Figure 4.14 shows the predicted velocity field. From the CNN model, I have 45 output points for each semblance panel. Then, I interpolate the 45 points via a spline method to estimate the whole velocity profile. I also apply a two-dimensional Gaussian filter to smooth the velocity field. White lines (dashed) indicate the positions of the CMP locations where I adopt Transfer Learning. Figure 4.15 shows the difference between the manual analysis velocity field and the predicted velocity field (Figure 4.13 minus Figure 4.14). The prediction result is quite similar to the velocity model estimated by manual analysis except for the area around CMP 800. As mentioned earlier, I exclude this area when I perform Transfer Learning. In this CMP range, the CNN model predicts velocities that are higher than those estimated by manual analysis.

Figures 4.16 show the semblance panels at CMP 350, 750, 1250 and 1400. It is important to mention that these CMPs were not used for Transfer Learning. Red lines indicate the manual velocity analysis and white lines represent the CNN predicted velocity. I point out a significant discrepancy between automatic velocities and manually estimated velocities at CMP 750. The discrepancy occurs in an area dominated by non-horizontal structures and diffracted energy. When analyzing the pattern of semblances from both areas outside the tabular salt body (Figure 4.16 (a)) and the area above the tabular salt body (Figures 4.16 (c) and (d)), it can be said that the CNN model has identified an acceptable solution.

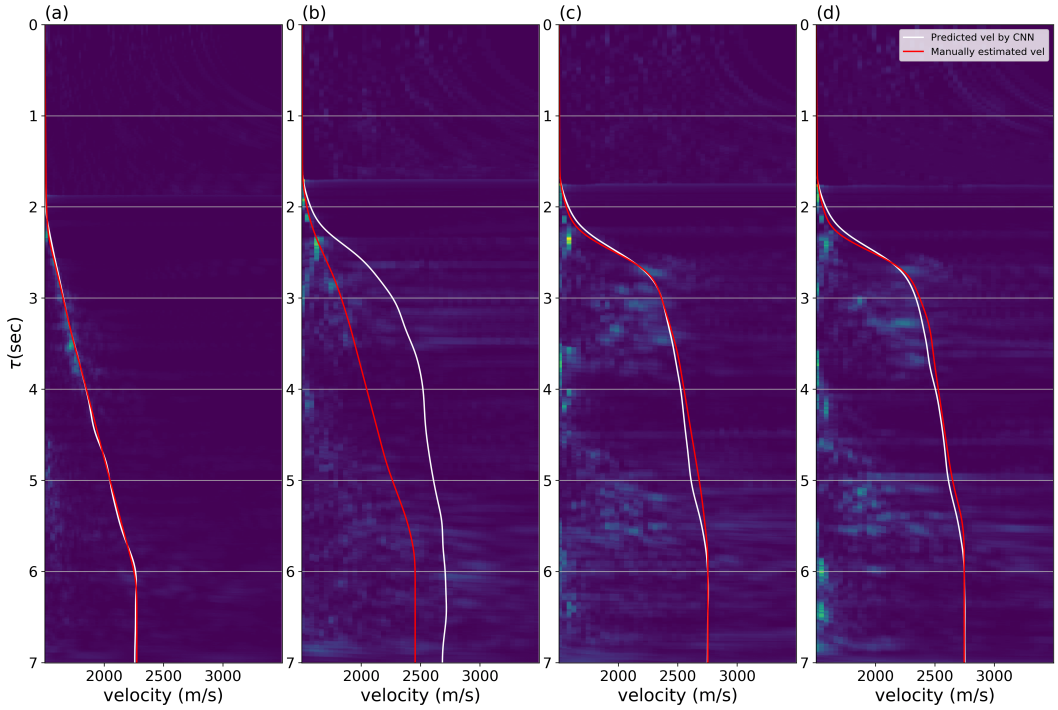

Figure 4.16: Semblance panels. CMP number: (a) 350, (b) 750 (c) 1250, (d) 1400. Manual analysis velocities (red lines) and the velocities predicted via CNN (white lines). These semblance panels are not used for Transfer Learning.

Finally, I compare stack sections for both the manual analysis (Figure 4.17) and the CNN prediction (Figure 4.18). Both stack sections have similar results in most areas. To verify the CNN prediction, especially in the area, including strong lateral variations, I zoomed two parts in this area (between CMP 600 and 1000). One is from τ =2.4 s to τ =3.6 s (Figure 4.19), and the other is between τ =5 s and τ =6.2 s (Figure 4.20). The CNN prediction results have clearer reflectors than the manual analysis; these results where indicated with red arrows.

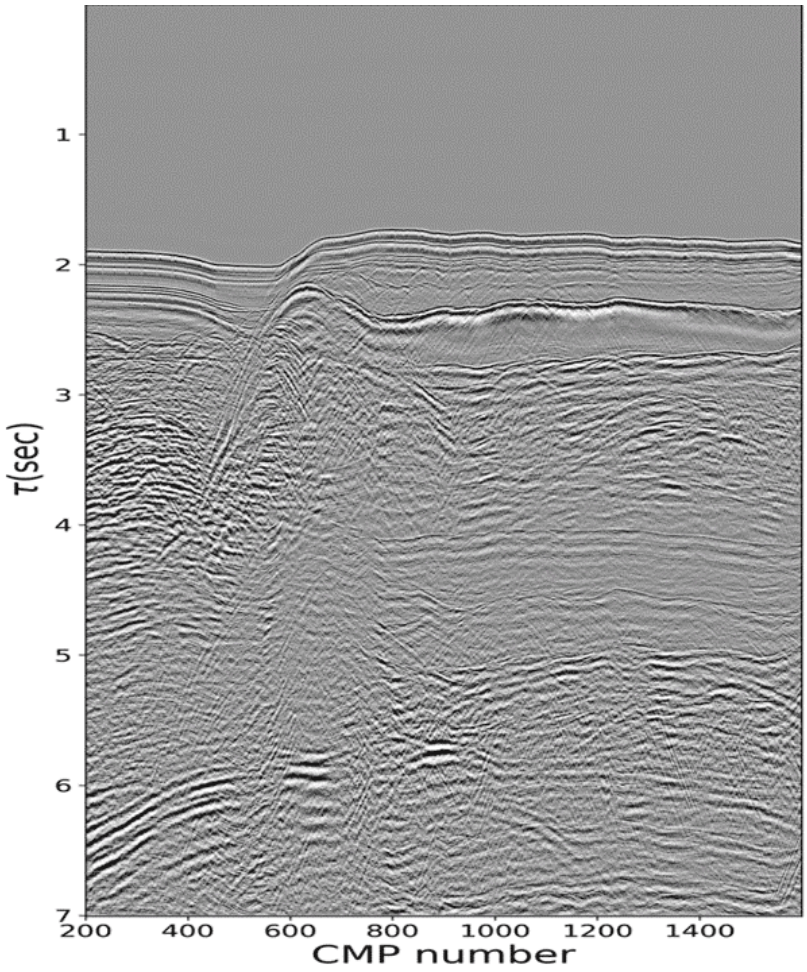

Figure 4.17: Stack section obtained by manual analysis velocity model.

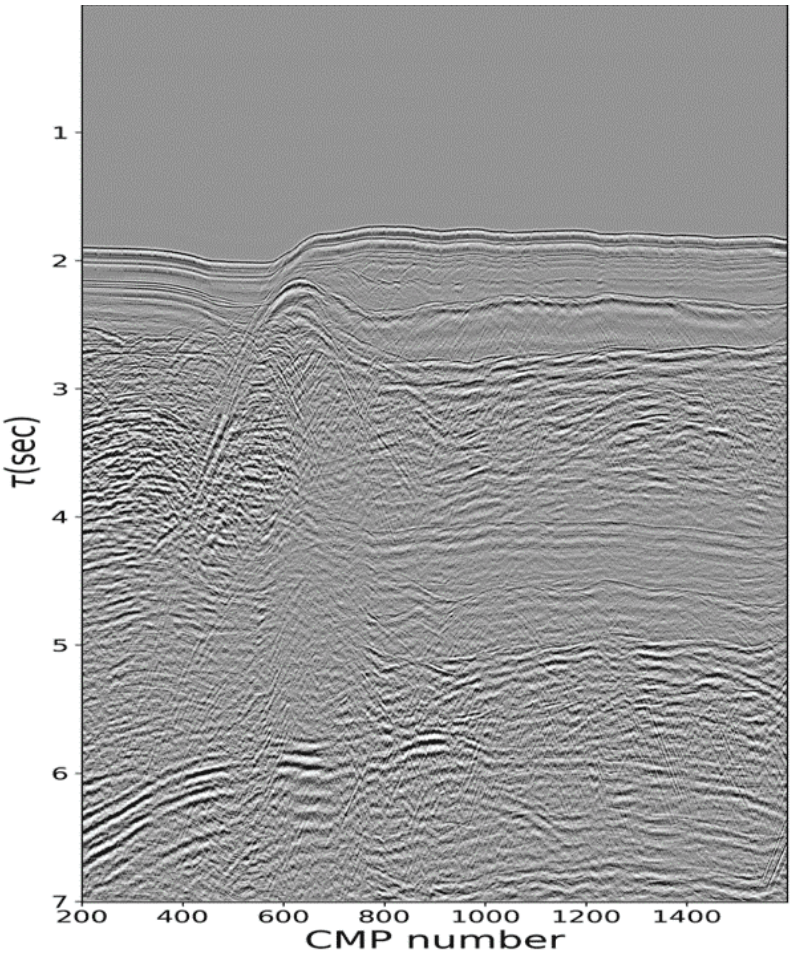

Figure 4.18: Stack section obtained via the velocity model predicted by CNN.

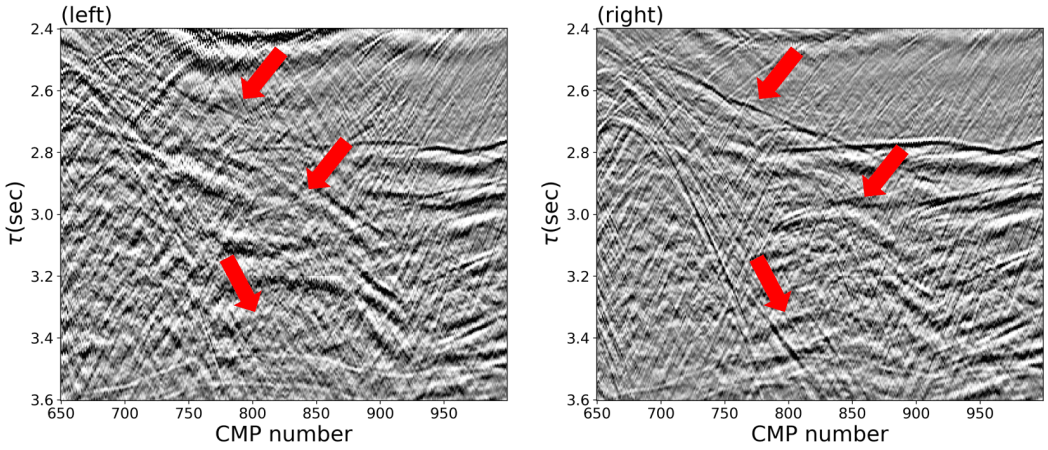

Figure 4.19: (a) The zoomed part from CMP 650 to 1000, τ =2.4 to τ =3.6 s.(Right) The results from the manual analysis and (Left) the results from the CNN prediction. Red arrows are indicating clearer reflectors in the zoomed parts of the CNN prediction.

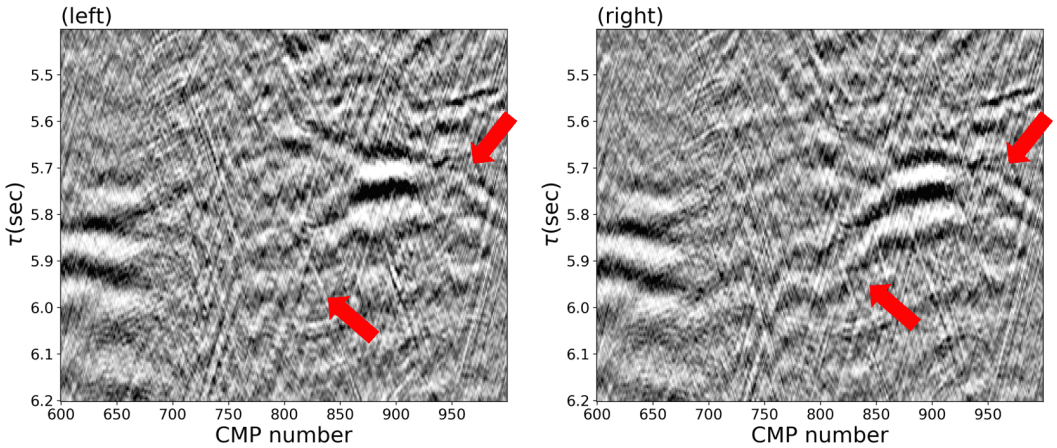

Figure 4.20: CMP 600 to 1000, τ =5 to τ =6.2 s. (Right) The results from the manual analysis and (Left) the results from the CNN prediction. Red arrows are indicating clearer reflectors in the zoomed parts of the CNN prediction.

CHAPTER 5

Conclusions

5.1 Main contributions

In this thesis, I have developed a method to perform semblance analysis using CNN automatically. In particular, I have used both the entire semblance and extracted patches to estimate the velocity at the specific traveltime. With this definition, I expect the trained CNN model to act like a professional human processor who can consider the entire trend of the semblance to avoid picking the multiples. The method is promising because the traditional way to conduct the velocity analysis requires a manual picking process which needs tremendous time and effort. I defined the velocity analysis task as the image classification problem by dividing the velocity axis of the semblance into segments of a specific size. The latter permits a direct CNN application to solve the velocity estimation problem. The propose methodology has the potential to yield a more accurate prediction using a deep layer CNN structure and smaller size of

compartments. Increasing the size of the input also causes a better prediction. Both synthetic and real examples are tested with our method. The base model predicts well in simple synthetic cases. In our approach, the base model can be updated continuously through new data. When the base model does not work correctly, I adopt the transfer learning, which can update the base model with a small portion of the target data. Even if I use only a few semblance panels for transfer learning, the updated model works well with both the Marmousi dataset and a real marine dataset. I want to point out that our method estimates reasonable velocities even if the data is contaminated by free-surface multiples as demonstrated with the Gulf of Mexico marine dataset.

5.2 Future work

The CNN model proposed in this paper does not consider surrounding semblance analysis and its lateral continuity as a good processor would do. This is the essential part we must take into account in the future. To overcome this limitation, one needs to expand our input data from a 2D semblance to a 3D semblance cube to include geological information for training (Araya-Polo et al., 2018). In this case, I have to use a different CNN approach. Our current CNN model structure has limited ability to extract features from a 3D input dataset. Semantic segmentation is one of the CNN approach that can be used to solve this limitation (Long et al., 2015; Garcia-Garcia et al., 2017). Using semantic segmentation, each pixel of the image can be defined as its label. Even if this method needs more computational resources and more training

datasets, it has shown enormous potential in various geophysical applications such as salt classification (Shi et al., 2018), channel detection (Pham et al., 2018), seismic facies classification (Zhao, 2018), and velocity estimation (Wang et al., 2018).

Another interesting future work is to adopt unsupervised learning such as generative adversarial nets (Goodfellow et al., 2014) to conduct data augmentation (Antoniou et al., 2017). In this thesis, I still have to perform manual velocity analysis to handle the real dataset. I expect that a data augmentation process could help to us to obtain better training performance.

Lastly, one must investigate ways to construct meaningful connections between deep learning and the physics of wave propagation. The performance of the trained CNN model only depends on the training data and does not consider constraints arising from physical considerations that involve solutions of the wave equation . In this thesis, for example, the base model needs to be updated to handle the target semblance that is slightly different from the semblance in training data. To overcome this, CNN consider typical physical considerations either arising from ray theory or approximated solutions to the wave equation.

Bibliography

Abadi, M., P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., 2016, Tensorflow: a system for large-scale machine learning.: OSDI, 265–283.

Abbad, B., B. Ursin, and D. Rappin, 2009, Automatic nonhyperbolic velocity analysis: Geophysics, **74**, U1–U12.

Antoniou, A., A. Storkey, and H. Edwards, 2017, Data augmentation generative adversarial networks: arXiv preprint arXiv:1711.04340.

Araya-Polo, M., J. Jennings, A. Adler, and T. Dahlke, 2018, Deep-learning tomography: The Leading Edge, **37**, 58–66.

Calderón-Macías, C., M. K. Sen, and P. L. Stoffa, 1998, Automatic nmo correction and velocity estimation by a feedforward neural network: Geophysics, **63**, 1696–1707.

Chen, Y., 2018, Automatic velocity analysis using high-resolution hyperbolic Radon transform: Geophysics, **83**, 1–21.

Chetlur, S., C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, 2014, cudnn: Efficient primitives for deep learning: arXiv preprint arXiv:1410.0759.

Choi, H., J. Byun, and S. J. Seol, 2010, Automatic velocity analysis using bootstrapped differential semblance and global search methods: Exploration Geophysics, **41**, 31–39.

Deng, J., W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, 2009, Imagenet: A large-scale hierarchical image database: 2009 IEEE conference on computer vision and pattern recognition, Ieee, 248–255.

Deng, L., 2012, The mnist database of handwritten digit images for machine learning research [best of the web]: IEEE Signal Processing Magazine, **29**, 141–142.

Fish, B. C., and T. Kusuma, 1994, A neural network approach to automate velocity picking, *in* SEG Technical Program Expanded Abstracts 1994: Society of Exploration Geophysicists, 185–188.

Fomel, S., 2009, Velocity analysis using AB semblance: Geophysical Prospecting, **57**, 311–321.

Garcia-Garcia, A., S. Orts-Escolano, S. Oprea, V. Villena-Martinez, and J. Garcia-Rodriguez, 2017, A review on deep learning techniques applied to semantic segmentation: arXiv preprint arXiv:1704.06857.

Goodfellow, I., J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, 2014, Generative adversarial nets: Advances in neural information processing systems, 2672–2680.

Guitton, A., 2003, Multiple attenuation with multidimensional prediction-error filters, *in* SEG Technical Program Expanded Abstracts 2003: Society of Exploration Geophysicists, 1945–1948.

Gulli, A., and S. Pal, 2017, Deep learning with Keras: Packt Publishing Ltd.

He, K., X. Zhang, S. Ren, and J. Sun, 2016, Deep residual learning for image recognition: Proceedings of the IEEE conference on computer vision and pattern recognition, 770–778.

Karpathy, A., G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, 2014, Large-scale video classification with convolutional neural networks: Proceedings of the IEEE conference on Computer Vision and Pattern Recognition, 1725–1732.

Kelly, K., R. Ward, S. Treitel, and R. Alford, 1976, Synthetic seismograms: A finite-difference approach: Geophysics, **41**, 2–27.

Keys, R. G., 1985, Absorbing boundary conditions for acoustic media: Geophysics, **50**, 892–902.

Kingma, D. P., and J. Ba, 2014, Adam: A method for stochastic optimization: arXiv preprint arXiv:1412.6980.

Kirk, D., et al., 2007, Nvidia CUDA software and GPU parallel computing architecture: ISMM, 103–104.

Krizhevsky, A., I. Sutskever, and G. E. Hinton, 2012, Imagenet classification with deep convolutional neural networks: Advances in neural information processing systems, 1097–1105.

Lawrence, S., C. L. Giles, A. C. Tsoi, and A. D. Back, 1997, Face recognition: A convolutional neural-network approach: IEEE transactions on neural networks, **8**, 98–113.

LeCun, Y., Y. Bengio, et al., 1995, Convolutional networks for images, speech, and time series: The handbook of brain theory and neural networks, **3361**, 1995.

LeCun, Y., Y. Bengio, and G. Hinton, 2015a, Deep learning: nature, **521**, 436.

LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, 1989, Backpropagation applied to handwritten zip code recognition: Neural computation, **1**, 541–551.

LeCun, Y., L. Bottou, Y. Bengio, P. Haffner, et al., 1998, Gradient-based learning applied to document recognition: Proceedings of the IEEE, **86**, 2278–2324.

LeCun, Y., et al., 2015b, Lenet-5, convolutional neural networks: URL: http://yann. lecun. com/exdb/lenet, **20**.

Liu, D., W. Wang, W. Chen, X. Wang, Y. Zhou, and Z. Shi, 2018, Random noise suppression in seismic data: What can deep learning do?, *in* SEG Technical Program Expanded Abstracts 2018: Society of Exploration Geophysicists, 2016–2020.

Long, J., E. Shelhamer, and T. Darrell, 2015, Fully convolutional networks for semantic segmentation: Proceedings of the IEEE conference on computer vision and pattern recognition, 3431–3440.

Maas, A. L., A. Y. Hannun, and A. Y. Ng, 2013, Rectifier nonlinearities improve neural network acoustic models: Proc. icml, 3.

Maturana, D., and S. Scherer, 2015, Voxnet: A 3D convolutional neural network for real-time object recognition: Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on, IEEE, 922–928.

Nair, V., and G. E. Hinton, 2010, Rectified linear units improve restricted boltzmann machines: Proceedings of the 27th international conference on machine learning (ICML-10), 807–814.

Neidell, N. S., and M. T. Taner, 1971, Semblance and other coherency measures for multichannel data: Geophysics, **36**, 482–497.

Nielsen, M. A., 2015, Neural networks and deep learning: Determination press USA, **25**.

Pan, S. J., Q. Yang, et al., 2010, A survey on transfer learning: IEEE Transactions on knowledge and data engineering, **22**, 1345–1359.

Pham, N., S. Fomel, and D. Dunlap, 2018, Automatic channel detection using deep learning, *in* SEG Technical Program Expanded Abstracts 2018: Society of Exploration Geophysicists, 2026–2030.

Powers, D. M., 2011, Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation.

Rosenblatt, F., 1958, The perceptron: a probabilistic model for information storage and organization in the brain.: Psychological review, **65**, 386.

Ruder, S., 2016, An overview of gradient descent optimization algorithms: arXiv preprint arXiv:1609.04747.

Rumelhart, D. E., G. E. Hinton, R. J. Williams, et al., 1988, Learning representations by back-propagating errors: Cognitive modeling, **5**, 1.

Sanders, J., and E. Kandrot, 2010, Cuda by example: an introduction to general-purpose gpu programming: Addison-Wesley Professional.

Schmidt, J., and F. A. Hadsell, 1992, Neural network stacking velocity picking, *in* SEG Technical Program Expanded Abstracts 1992: Society of Exploration Geophysicists, 18–21.

Shi, Y., X. Wu, and S. Fomel, 2018, Automatic salt-body classification using a deep convolutional neural network, *in* SEG Technical Program Expanded

Abstracts 2018: Society of Exploration Geophysicists, 1971–1975.

Simard, P., L. Bottou, P. Haffner, and Y. LeCun, 1999, Boxlets: a fast convolution algorithm for signal processing and neural networks: Advances in Neural Information Processing Systems, 571–577.

Simard, P. Y., D. Steinkraus, and J. C. Platt, 2003, Best practices for convolutional neural networks applied to visual document analysis: null, IEEE, 958.

Simonyan, K., and A. Zisserman, 2014, Very deep convolutional networks for large-scale image recognition: arXiv preprint arXiv:1409.1556.

Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, 2014, Dropout: a simple way to prevent neural networks from overfitting: The Journal of Machine Learning Research, **15**, 1929–1958.

Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, 2015, Going deeper with convolutions: Proceedings of the IEEE conference on computer vision and pattern recognition, 1–9.

Taflove, A., and S. C. Hagness, 2005, Computational electrodynamics: the finite-difference time-domain method: Artech house.

Taner, M. T., and F. Koehler, 1969, Velocity spectra digital computer derivation applications of velocity functions: Geophysics, **34**, 859–881.

Toldi, J., 1985, Velocity analysis without picking, *in* SEG Technical Program Expanded Abstracts 1985: Society of Exploration Geophysicists, 575–578.

Verschuur, D., and R. Prein, 1999, Multiple removal results from Delft university: The Leading Edge, **18**, 86–91.

Versteeg, R., 1994, The marmousi experience: Velocity model determination on a synthetic complex data set: The Leading Edge, **13**, 927–936.

Wang, W., F. Yang, and J. Ma, 2018, Velocity model building with a modified fully convolutional network, *in* SEG Technical Program Expanded Abstracts 2018: Society of Exploration Geophysicists, 2086–2090.

Wu, X., L. Liang, Y. Shi, and S. Fomel, 2019, Faultseg3d: using synthetic datasets to train an end-to-end convolutional neural network for 3d seismic fault segmentation: Geophysics, **84**, 1–36.

Wu, X., Y. Shi, S. Fomel, and L. Liang, 2018, Convolutional neural networks for fault interpretation in seismic images, *in* SEG Technical Program Expanded Abstracts 2018: Society of Exploration Geophysicists, 1946–1950.

Xiong, W., X. Ji, Y. Ma, Y. Wang, N. M. BenHassan, M. N. Ali, and Y. Luo, 2018, Seismic fault detection with convolutional neural network: Geophysics, **83**, 1–28.

Yilmaz, Ö., 2001, Seismic data analysis: Processing, inversion, and interpretation of seismic data: Society of exploration geophysicists.

Yosinski, J., J. Clune, Y. Bengio, and H. Lipson, 2014, How transferable are features in deep neural networks?: Advances in neural information processing systems, 3320–3328.

Yuan, S., J. Liu, S. Wang, T. Wang, and P. Shi, 2018, Seismic waveform classification and first-break picking using convolution neural networks: IEEE Geoscience and Remote Sensing Letters, **15**, 272–276.

Zhao, T., 2018, Seismic facies classification using different deep convolutional neural networks, *in* SEG Technical Program Expanded Abstracts 2018: So-

ciety of Exploration Geophysicists, 2046–2050.

APPENDIX A

Reproducibility

As part of my research, and in order to provide an opportunity to reproduce my results, I also provide the source codes produced during my research. Codes were written by Keras, which is the high-level API for Tensorflow (Gulli and Pal, 2017; Abadi et al., 2016). The source code can be dowload from the following site https://github.com/mjmr0128/SVACNN. The list below indicates prerequisites to run the codes. Note that CUDA and Cudnn are only needed for the GPU version of Tensorflow (Kirk et al., 2007; Sanders and Kandrot, 2010; Chetlur et al., 2014):

- Python 3.6+
- Jupyter Notebook 4.4+
- Tensorflow 1.13+ (GPU version)
- Keras 2.2.4+

- CUDA 10.1+
- Cudnn 7.5+

This distribution consists of two main parts. First one (`1_base_model_training.ipynb`) is for the Base model training with synthetic data. In terms of model architecture, there are three possible options: LeNet-5, AlexNet, and VGG16 (LeCun et al., 1998; Krizhevsky et al., 2012; Simonyan and Zisserman, 2014). After training, it tests the trained model with another synthetic dataset, which is not included in the training dataset. The second one (`2_transfer_learning.ipynb`) performs transfer learning to the Base model with the Marmousi dataset. It updates the base model with a small portion of the Marmousi dataset and tests the updated model with entire Marmousi dataset.