2021-10-18

Theory-guided machine learning in geophysics

Niu, Zhan

UNIVERSITY OF CALGARY

Theory-guided machine learning in geophysics

by

Zhan Niu

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN GEOLOGY AND GEOPHYSICS

CALGARY, ALBERTA

OCTOBER, 2021

Abstract

Machine learning has become a popular topic in the past decade thanks to the booming in computer hardware and the tools invented. Many successful applications have been made in various subjects in geophysics, including salt body detection, facies recognition and inversion etc. However, the fact that most geophysical theory is well-established sometimes contradicts the black box theory in machine learning when combining methods in the two fields. This thesis will discuss several ways of incorporating well-established knowledge into machine learning by giving a few applications and experiments in geophysics. We will also discuss the limitations and challenges machine learning is facing.

Acknowledgements

I would first like to thank the directors and sponsors of CREWES project, who have funded me during my master's research. Among those, I sincerely thank my supervisor Dr. Daniel Trad, who has given me guidance not only on my research but also on my career and life. Not to mention the efforts he has made editing this thesis and correcting the ugly grammar errors I have made.

Among all CREWES fellows, I would also like to express my special gratitude to Jian Sun, Marcelo Guarido and Lei Yang, who have given me valuable discussions not only as colleagues but also have been supporting me as big brothers.

In addition, I would like to thank my parents, who have been living in China but never made me feel apart. I thank both my grandmothers Huizhen Li and Shuzhen Wang, who have passed away during my master's program. They taught me to be honest, caring and humble with their live examples, and I will carry their philosophies with me for the rest of my life.

I would also like to thank the geoscientists I have met, including my father, Jiayu Niu. Their professional skills and styles make me always proud of being a geoscientist myself.

To anyone or anything that has helped me.

Table of Contents

List of Figures and Illustrations

List of Tables

List of Symbols, Abbreviations and Nomenclature

Symbol or abbreviation	Definition
ML	Machine learning
BP	Back-propagation
ReLU	Rectified Linear Unit
CNN	Convolutional neural networks
RNN	Recurrent neural networks
FC	Fully-connected neural networks
GAN	Generative adversarial networks
WGAN	Wasserstein GAN
ADAM	Adaptive moment estimation
RMSprop	Root mean square propagation
L-BFGS	Limited-memory Broyden-Fletcher-Goldfarb-Shanno algorithm
FR-CG	Fletcher-Reeves conjugate gradient
FWI	Full waveform inversion
LSRTM	Least squares reverse time migration

Epigraph

Plurality is never to be posited without necessity.

- William of Ockham,

Questions and the decisions of the Sentences of Peter Lombard

Chapter 1

Introduction to machine learning

1.1 What is machine learning used for?

Machine learning is a flexible tool capable of solving many different types of problems. In theory, machine learning can always give you a promising solution even without an understanding of the actual problem from a physical point of view (often referred to as domain knowledge). However, for the same reason, we need to be careful when we use it. The deployment of machine learning can be easy — one can throw all the data we have into a neural network and hope to get the right answer after training. However in this case, the solution is likely to have compromised generalization and be inconsistent with physics.

To avoid this side effect of machine learning flexibility, for every problem we have to ask ourselves — What part of a problem are we expecting for machine learning to help us? In my perspective, machine learning can be applied in two ways to make use of its advantages. The first is to train models for problems or subproblems that are difficult to formulate explicitly. Those type of problems are usually intangible and abstract. A cat classifier would be a great example that falls in this category. It is hard to come up with a accurate formulation for the probability of being a cat given a picture, but machine learning can find the relationship by treating it as a black box. The cost of treating it as a black box

is minimal, since the most robust theory to identify a cat is by observation and comparison with previous experience. Another example in geophysics would be facies recognition based on well logs. This process integrates information from various sources, including rock properties, the surrounding geology and seismic data. The quality of each source is also uncertain. Data from different sources sometimes contradict each other, and the main job is to evaluate the importance of different parameters, which can be highly based on the interpreter's experience. This process is thus very hard to formulate, and machine learning has played an important role in this type of task. We can apply machine learning to only the part where the method is vague (B. Sun & Alkhalifah, 2019; Zhou & Brown, 2020). The other category is that we want to use machine learning to speed up the calculation we have already had. An example is migration/inversion solved by machine learning (Biswas et al., 2019). Because sometimes the established physics model can be over-parameterized, the performance might be harmed by having additional parameters. A trained neural network can help skip unnecessary steps and hence speed up the process.

1.2 The limitation of data-driven methods

A key challenge of applying machine learning techniques to geophysics is that geophysical solutions are usually not unique. We are able to acquire large quantities of data, but unfortunately most of the data are often redundant because they contain a similar type of information. Therefore, in spite of data abundance, the problem can be ill-conditioned. In seismology, for example, although reflections can cover many different incident angles, the data still only sense the reflector from the surface down. As a comparison from a different field, magnetic resonance imaging (MRI) is able to look at the object from all angles. In this case, the extra views that different angles of incidence provide independent information and hence bring better imaging.

Another problem we face in Geophysics is the lack of labels or the existence of wrong

labels. Because the subsurface is inaccessible in general , we do not really know the ground truth for our problems and the labels we use in geoscience-related problems are often incorrect. The data are the best guesses from the collaboration of experienced seismic interpreters, geologists and reservoir engineers, but they are still estimates. Even for well logs, where we cut through the rocks and step close to the ground truth, errors can still be introduced during processing. Furthermore, the number of well logs is often very limited due to the drilling cost and well log information tends to be insufficient.

With all the above in mind, geophysical problems are often underdetermined. An underdetermined problem is where the number of unknowns is more than the number of data points. It requires constraints and robust data fitting to compensate for the non-uniqueness and data error, respectively. One type of constraint is regularisation, where we add an extra term to the cost function. For example, in inversion, models can be constrained to be sparse or smooth by minimizing the first and second derivatives of the model. The model can be constrained by the loss function itself. The effect of outliers is often attenuated by choosing ℓ_1 loss functions, while it can be emphasized by choosing ℓ_1 loss functions. We will discuss more on regularisation in Chapter 4.

Similarly, machine learning is also an under-determined problem because we usually have more trainable parameters than the number of data points. However, being under-determined is not a disadvantage if handled properly, and it enables the potential of neural networks to mimic all kinds of functions. The quality of data is usually better than in the geophysical case. For example, for a cat/dog classification problem, we can have a clear view to help us identify whether it is a dog/cat. Also, we can be certain with our answer in most cases. Those conditions can seldom be met in a geophysical context.

Therefore, although machine learning techniques can help to solve geophysical problems, we should be aware that additional constraints may be required for most problems. These new constraints could have similar formulations and traditional methods.

This discussion suggests that a reasonable approach to take advantage of the power

of ML is to use it for solving regression problems whose formulation is not well defined. For example, there is not much advantage of using a completely flexible formulation to simulate a wave equation unless we are uncertain that the existent formulation is accurate.

1.3 Physics-informed neural networks

In general, replacing complex physics with a black box ML tool is probably not bringing significant progress because this approach bypasses the understanding that the physical theories provide. For example, Weyn et al., 2019 trained deep learning models to predict the weather at hPa geopotential height, but none of these models could outperform a non-machine-learning model developed by using physical laws.

However, we certainly could use the flexibility of ML combined with our understanding of a problem. In other words, we could inject physics into the ML formulation. von Rueden et al., 2019 mentions three main ways for injecting information to neural networks:

1. **Introduce information into the training data.** Since machine learning is a data-driven method, we can use our knowledge about the problem to modify the data and make it easier for the network to learn from the data. Additional knowledge can be implicitly passed to the model by carefully defining the distributions of the data and selecting the features to be fed to the model, i.e. feature engineering.

2. **Introduce knowledge into the neural network architecture.** A big part of any type of regression, including ML, is choosing a proper model. One can observe the relationship between the input and output and find a function that would best reflect this relationship. The knowledge is injected during model selection. Convolutional neural networks for image recognition are a good example. By choosing a CNN as the model structure, we inject the knowledge and assumption that hidden features are only related to spatially nearby points and that they are space invariant. Similarly, by

choosing an RNN as the model, we inject knowledge of a causal relationship between the input and output, which means the output is related to prior time steps.

3. **Add extra constraints in the cost function to reflect prior knowledge about the model.**

   The most straightforward way of modifying the cost function is to add regularisations. For example, using an ℓ_1 norm to measure the model size implies we believe on a sparse model with many null coefficients. The ℓ_2 norm of the second derivatives of the variable helps to minimize the structure of the variables or a complex function that minimizes the change in energy (Karpatne, Watkins, et al., 2017). Aside from adding regularisation terms, similarly, we can also alter the main body of the cost function, i.e. the metrics that used to evaluate the error. One can choose to use ℓ_1 or ℓ_2 norms, just like solving problems in traditional ways. But also, we can have more complex loss functions thanks to auto differentiation. For example, in Biswas et al., 2019, instead of using a cost function that evaluates the error between neural network outputs with label data, the authors apply modelling on the output and evaluate the error with the output. Their approach is similar to an auto-encoder, that is a network that transforms the input to itself $x' = D(E(x))$, with the constraint that the decoder E is not trainable.

1.4 The continuum and thesis outline

Let us consider the following general equation to describe a medium:

$$\boldsymbol{y} = f(\boldsymbol{x}) \tag{1.1}$$

As shown in Equation (1.1), there are three elements. The independent variable $\boldsymbol{x}$, the dependent variable $\boldsymbol{y}$, and the mapping function f that describes the relationship between

them. A physics model is a series of equations or differential equations f_1 that best describes the mapping between x and y. This is often done in physics using derivations. For example, in Newton's 2nd law ($F = ma$), the input $\boldsymbol{x}$ refers to the mass of the object m and the acceleration a. The output could be the force applied to the object. $f_1()$ is the multiplication, which says the external force F is certain once given the mass and acceleration of an object. However, the function is a result of human understanding of the problem and as such it depends on the robustness of previous studies. For example, this law is held on assumptions that mass is not a function of velocity, which is not true when more complete theories are taken into account. The assumption indeed brings uncertainties and fails to describe the truth where it cannot be met. A more accurate form f_2 is that force equals the rate of change in momentum ($F = \frac{\mathrm{d}(mv)}{\mathrm{d}t} = m\frac{\mathrm{d}v}{\mathrm{d}t} + v\frac{\mathrm{d}m}{\mathrm{d}t}$). This form expands Newton's 2nd law by removing the assumption on constant mass.

Forward seismic modelling is another example. The choice of absorbing boundary conditions and the assumptions of wave complexity can greatly affect modelling results, most of which are far from the true representation of the real world. In all, improvements of a model are tied to better understandings of the real world, and the search for the ultimate true model is an non-ending journey for every problem.

On the other hand, Machine learning is also a study on finding a mapping function f. Instead of being based on a known theory for the problem at hand, machine learning trains a model with available data to find the best mapping of data. With no other information provided, the accuracy of the model is tied to the quality of data. In the era of big data that we live today, machine learning is booming in computer vision, classification and regression. This is because data samples in these areas are abundant. A shortcoming of data-driven models is also obvious — the need for a large amount of labelled data is usually difficult or too costly to fulfil in some other fields.

Despite different viewpoints, data-driven and theory-driven methods serve the same goal: finding the most accurate representation of f possible. Figure 1.1 is the 1D version

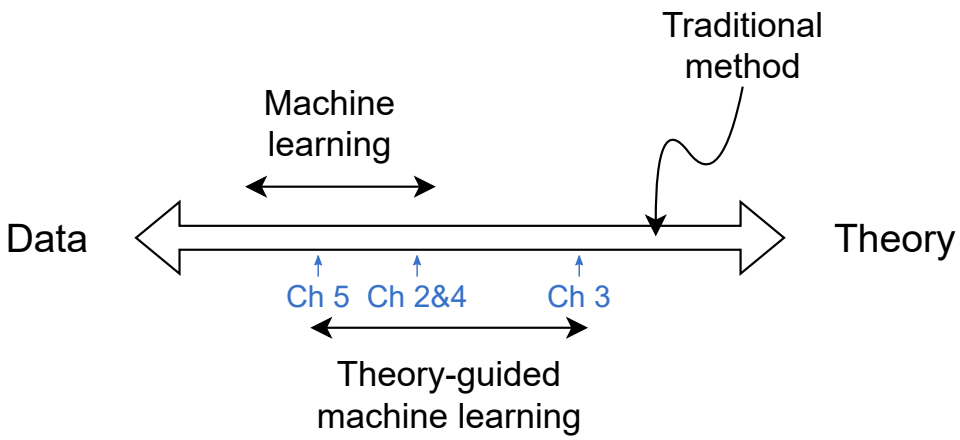

Figure 1.1: The continuum between theory and data-driven method.

modified from Karpatne, Atluri, et al., 2017. Note that machine learning in this figure refers to classic data science methods in general.

Each chapter can be placed qualitatively on this continuum. In Chapter 2, we trained a U-net for separating pseudo-deblended shots, which still fall in the category of classic machine learning. However, the domain knowledge injected by carefully selecting model architecture and data preparation shifts the point to the theory side. In Chapter 3, we proposed born inversion in an RNN framework. The architecture is customized by hand using low-level APIs. Coefficients and relationships are hardcoded to the network so that the method is very close to the theoretical methods. Chapter 4 and Chapter 2 are in the same scenario, where we trained a fully connected ResNet to perform velocity inversion from seismic images. In Chapter 5, we trained a generative adversarial neural network with Wasserstein loss and gradient penalty to generate 1D seismic shots. The training is unsupervised so that we can inject only a limited amount of knowledge, and hence it is closest to the data side. The knowledge injected includes the domain knowledge to select a suitable neural network type. Also, the discriminator itself can be treated as a constraint added to the loss function, despite it being trainable.

1.5 Training method

1.5.1 Defining dataset

An important aspect of ML is that the model has to be trained and evaluated on different data sets. We divide the dataset into different sets, each with a different purpose. The largest data set is used for training, that is for the model to learn the input/output connections.

Because the model sees these data during training, it also memorizes them. As a consequence, predictions obtained from this dataset are better than would be if the data were completely unknown. Therefore, a separate test dataset is required to fairly evaluate the model. This test dataset should come from the same distribution of the training set and should have known labels, that is the true answers when the network operates on it should be known. To be fully independent of the model, the test dataset must remain unseen to the neural network and should never be used on gradient calculation or model selection.

In addition, we need another dataset from the same distribution known as the validation set. During the training process, we want to converge to a model that generalizes well, i.e. a model that performs better on unseen data. This goal requires adjusting optimization parameters such that the prediction produces a minimum error on unseen data which is the validation dataset. This dataset is not directly used for the training process, that is to estimating mapping weights by back-propagation, but will be used to give some insights for tuning hyper-parameters and test different network architectures. Validation is important since it is the only method we have to evaluate whether a model is over-fitted or under-fitted. Machine learning problems can sometimes be an under-determined problem, that is having more parameters to solve for than the information provided by the data points can support. This is a serious difficulty in deep networks, especially when the sample size is small. If the loss from the validation set is similar to the loss from the training set, then we can say the model performs equally well on seen and unseen data. Hence we can have confidence that the model will do well on the test set.

On a typical machine learning problem, the validation loss curve will decrease similarly to the training loss during the early stage. However, the training loss will usually decrease faster since the gradient is optimized for the training dataset (network weights are calculated using the information provided by the training residuals). Then, as the model starts to over-fit the training set, the validation loss will start to increase because the model is over-fitting particular details that belong to the training data set only. Although the training loss will be smaller after this point, we should prefer the model where the validation loss is minimal. This follows from the fact that the validation dataset is less susceptible of being over-fit since it was not used to calculated gradients. This discussion shows that, although not directly, the validation set is used for training, which makes the model depend on the validation set. Therefore a separate test set is still necessary with the presence of the validation set,

There is no strict rule on how many fractions each set should have since it depends on the specific situation and the nature of the problem. Insufficient number of samples in any of the three sets will cause its own problems, and there is a trade-off between these problems that should be balanced. In the ideal case where the number of samples is adequate, the convention is to split the entire dataset by $60\,\%/20\,\%/20\,\%$ to form the training/validation/test set. However, for small datasets, we may give up the validation and use a ratio $80\,\%/20\,\%$ for training/test. The latter scenario is non-ideal, but with very limited amount of data the emphasis is to have enough data for training. Later we will discuss other ways to compensate for this issue by using data augmentation, which is also non-ideal, but it tends to work better than ignoring validation.

1.5.2 Training workflow

Algorithm 1 shows a template of a training workflow. $\mathcal{M}(\cdot)$ refers to the chosen model, X, Y are input and label of the dataset. The subscript of X and Y indicate its source, and prime means the corresponding prediction. $\mathcal{L}(\cdot)$ refers to the loss function and α is the

Algorithm 1 Training template.

Require: $\mathcal{L}(\cdot)$, $\mathcal{M}(\cdot)$, X, Y, α
  for each epoch **do**
    $Y'_{\text{train}} \leftarrow \mathcal{M}(X_{\text{train}})$ ▷ compute prediction
    $L \leftarrow \mathcal{L}(Y'_{\text{train}}, Y_{\text{train}})$ ▷ compute loss
    $g \leftarrow BP(L)$ ▷ back-propagate
    $\mathcal{M}(\cdot) \leftarrow \mathcal{M}(\cdot) + \alpha g$ ▷ update model parameters
    $Y'_{\text{val}} \leftarrow \mathcal{M}(X_{\text{val}})$
    $L_{\text{val}} \leftarrow \mathcal{L}(Y'_{\text{val}}, Y_{\text{val}})$ ▷ compute validation loss
    if L_{val} is the smallest **then**
      $M \leftarrow \mathcal{M}(\cdot)$ ▷ save the best model as M
  $L_{\text{test}} \leftarrow \mathcal{L}(M(X_{\text{test}}), Y)$ ▷ Evaluate test loss

learning rate or the step size. Chapter 2,3 and 4 each uses a slightly modified version of the training template. Details will be discussed when introduced in each chapter.

Chapter 2

Deblending with UNet

2.1 Introduction

Blending acquisition is a technique to reduce the cost of seismic acquisition. It enables us to fire several shots simultaneously, which not only reduces the time of recording but also reduces the cost of storing seismic data (Beasley et al., 1998). The reduction of recording time also reduces the cost of labour and mitigates the exposure to ambient noise.

To process blended data, we need to apply pre-processing to separate the simultaneous shots. This deblending process essentially separates coherent energy coming from different sources so they can be treated as regular seismic data. This is an under-determined inversion problem since it tries to produce several shots from each supershot (that is, it has more unknowns than equations). Therefore, additional constraints must be applied to get a unique solution.

There are many different blending approaches, but one commonly applied is shot dithering. This method involves introducing known random delays to the firing time for each shot. These time delays shift each shot differently, which has an effect on domains where we group together different shots (for example, common midpoint gathers or receiver gathers). The energy coming from different shots becomes incoherent in those domains. We

can use this characteristic to separate shots by applying corrections for these delays to the targeted shots. This process is called pseudo-deblending, and its effect is the conversion of the multi-shot gathers into noisy data sets. Thus deblending becomes similar to a denoising process, where the noise here is the blended energy.

The most challenging part is to solve the interference where different shots overlap. There are many choices for the denoising tool. For example, masks or mutes can be applied to F-K or hyperbolic Radon domains. Furthermore, inversion-based methods have been developed as well, which involve creating a cost function with a regularization term. Results are highly dependent on the definition of the regularization, and the assumptions within can cause loss of signal (Stanton & Wilkinson, 2018).

On the other hand, machine learning methods can be applied instead of signal-processing / inversion types of deblending operators. The problem can be defined in two ways based on previous studies: a classification problem or a regression problem. The classification method aims to generate a mask that indicates the position of the desired shots but leaves the interferences unsolved, while the regression problem tries to produce individual shots but requires more parameters to be determined. For example, Baardman, Tsingas, et al., 2019 used convolutional neural network (CNN, LeCun et al. 2015) for both problems. However, we think that CNNs may not be the best choice to capture the relationship between inputs and outputs due to the lack of skipping connections. Richardson and Feller, 2019 chose a U-Net model with ResNet34 encoder pre-trained on ImageNet and trained with random velocity models.

In this chapter, we will look at a wedge model with scatterers as a toy problem and discuss the suitability of U-Nets in general for solving deblending problems.

2.2 Theory

2.2.1 Model definition

Our chosen neural network architecture to solve the deblending problem is the U-Net (Ronneberger et al., 2015). The U-Net was designed based on an encoder-decoder backbone with added bridge connections, which communicate the encoder and decoder parts and facilitate the flow of information. This type of network is often used for solving segmentation problems. Figure 2.1 shows a typical structure of U-Nets.

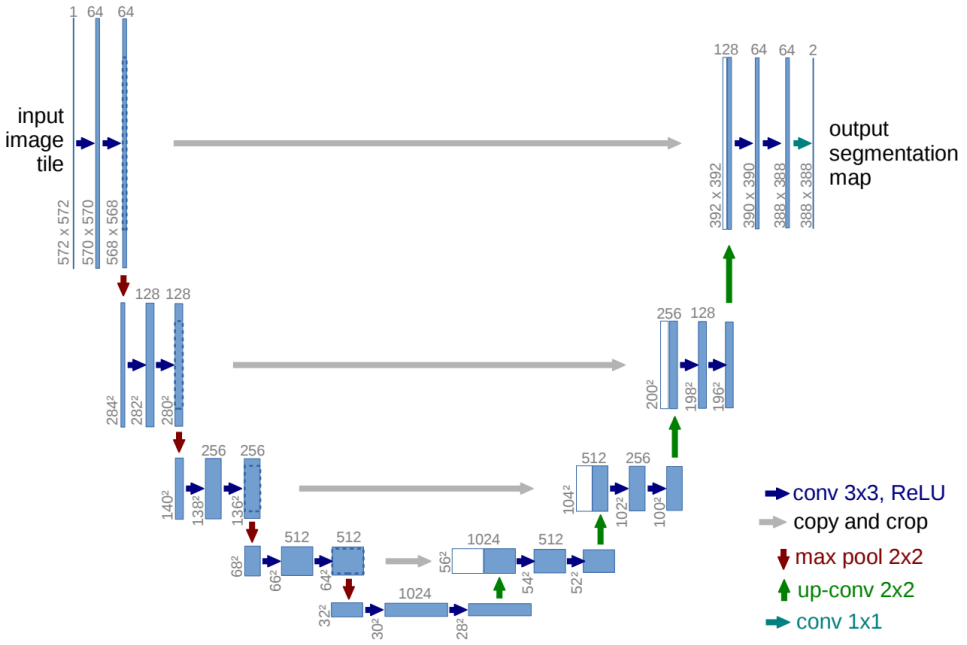

Figure 2.1: Diagram of U-Net model modified from Ronneberger et al., 2015. The gray arrows refer to the bridge connections that directly pass the features from down-going layers to up-going layers.

The U-Net contains three parts. The down-going/encoding part, the up-going/decoding part and the bridge connections. The down-going part refers to the left half of the figure. The U-Net has four tiers in total. At the first tier, the inputs went through two 3 by 3 convolutional layers with a predefined initial number of filters (64 filters in this case). The initial number of filters defines how many features are extracted from the inputs.

The cascading of convolutional layers essentially enlarges the extent of kernel coverage. After the convolution, the output was max-pooled by a 2 by 2 kernel with a stride of 2, and the result is used for the inputs fed to the next tier. At each time going down a tier, the number of filters used for convolution doubled while the image dimension is halved due to the max-pooling. On the other side for the up-going part, the inputs will go through the opposite process generally. In this part, the number of filters gets halved, and the image dimension gets doubled as the tier goes up. At the top tier, the image dimensions are restored to the original size and the number of channels is reduced to 1 by an additional outputting convolution layer. Each green arrow in the figure stands for a 2 by 2 up-convolution, which up-samples the image by two and then convolves with a 2 by 2 kernel. At this point, this structure is called an encoder-decoder convolutional network. This type of structure extracts high-order features from the inputs and reconstructs the output by decoding. The third part is the bridge connections, which are indicated by the gray horizontal arrows. Each grey arrow refers to the process where part or the whole outputs was forwarded as additional features to the same tier in the up-going part. These additional channels were concatenated in the channel dimension with the outputs from deeper tier after up-convolution. The introduction of bridge connections reduces the length of backpropagation and hence mitigates the risk of vanishing gradients.

A segmentation prediction produces a mask of a given picture indicating an area of interest. For example, this has applications on the brain MRI for finding the damaged area, or in our case in seismic, for targeting events in a noisy shot record. Essentially, it predicts the probability of a given pixel to be true. The probability on each pixel then can be converted to a true or false by applying a judging threshold. The reason why U-Net is more suitable to solve segmentation problems over traditional variations of CNNs is that U-Net has bridge connections that directly link the features with the same tier as shown in the gray arrows in Figure 2.1. Since the inputs and outputs of the segmentation usually correlate and share spatial similarities, the connections will greatly reduce the efforts to

learn this relation by skipping unnecessary transforms, which helps to reduce the chance of vanishing gradients.

2.2.2 Loss function

In machine learning, the optimization concept of a cost function is called loss, and it represents some measure of the proposed model undesired features (for example fitting error or complexity). Since the deblending problem can be thought of as a regression problem, a common loss to use is the mean square error (MSE), which is the squared L_2 norm averaged across each pixel. This type of loss function offers easy derivatives and provides a convex shape. The MSE loss is defined as

$$L = \text{mean}\left(\|Y - Y_{\text{pred}}\|_2^2\right) = \frac{1}{N_s}\sum_{\text{sample}}\frac{1}{N_p}\sum_{\text{pixel}}(Y - Y_{\text{pred}})^2, \tag{2.1}$$

where the loss is normalized by the number of samples (N_s) and the number of pixels in a shot record (N_p) so that the error reflects the mean error in each pixel. N_s represents the number of samples in one evaluation, which is not necessarily the total number of inputs since often the error is evaluated in each minibatch independently due to the memory limitations of the device.

2.2.3 Back-propagation

The gradients with respect to each model parameters can be calculated by a back-propagation algorithm, which is a recursive estimation of error propagation by applying the chain rule (I. Goodfellow et al., 2016). Equation 2.2 shows a form of gradient calculated by back-propagations. Suppose that $L = \mathcal{L}(\mathbf{a})$ and $\mathbf{a} = f(\mathbf{h})$, then the gradient of L with respect to the hidden parameters h is

$$\frac{\partial L}{\partial h_i} = \sum_j \frac{\partial L}{\partial a_j} \cdot \frac{\partial a_j}{\partial h_i} \tag{2.2}$$

In short, back-propagation is an algorithm that calculates the gradient of a scalar function (typically the cost function J) with respect to the hidden parameters (h) in the model. The back-propagation starts from $\frac{\partial J}{\partial J} = 1$ and then gets the gradient for the last hidden parameter by multiplying the Jacobian for the operations that produce the output. By a recursive process, each gradient for hidden parameters at each layer can be obtained and used to update the parameters in the calculation order.

One can use the gradients directly to update the model or use gradient-based optimization methods to make updates in a more controlled manner. The first is the most intuitive way but may result in a zigzag path to the minimum. The second method tries to reduce the zigzag pattern and is faster in an ideal case. One popular method to perform minimization is adaptive moment estimation (ADAM, Kingma & Ba, 2014), which reduces the transverse oscillations by cumulatively summing all the previous gradients during the optimization. The pseudocode of ADAM update is shown in Algorithm 2. A more detailed explanation

Algorithm 2 The ADAM optimization. i stands for the current iterations. $\mathbf{g}$ is the gradient calculated and $\mathbf{h}$ contains the parameters to be updated. $\mathbf{v}$ and $\mathbf{s}$ are the two vectors storing the cumulative sum of historical $\mathbf{g}$ and $\mathbf{g}^2$. α and β are two hyper parameters that control the portion of updates that is related to $\mathbf{v}$ and $\mathbf{s}$.

$\mathbf{v}_0 \leftarrow \mathbf{0}$
$\mathbf{s}_0 \leftarrow \mathbf{0}$
$i \leftarrow 0$
while $i <$ iterations **do**
  $i \leftarrow i + 1$
  Calculate $\mathbf{g}_i$
  $\mathbf{v}_i \leftarrow \beta_1 \mathbf{v}_{i-1} + (1 - \beta_1)\mathbf{g}_i$
  $\mathbf{s}_i \leftarrow \beta_2 \mathbf{s}_{i-1} + (1 - \beta_2){\mathbf{g}_i}^2$
  $\hat{\mathbf{v}}_i \leftarrow \frac{\mathbf{v}_i}{1-{\beta_1}^i}$
  $\hat{\mathbf{s}}_i \leftarrow \frac{\mathbf{s}_i}{1-{\beta_2}^i}$
  $\mathbf{h}_i = \mathbf{h}_{i-1} - \alpha \frac{\hat{\mathbf{v}}_i}{\sqrt{\hat{\mathbf{s}}_i} + \epsilon}$

of ADAM and its characteristics will be discussed in Chapter 3, where a more complicated

problem will be solved.

2.2.4 Training workflow

Algorithm 3 Training workflow for the UNet.

Require: $\mathcal{L}(\cdot)$, $\mathcal{M}(\cdot)$, X, Y, ADAM$(\cdot)$
  for each epoch **do**
    for each minibatch **do**
      zero the gradients
      load X_{train} and Y_{train}
      $Y'_{\text{train}} \leftarrow \mathcal{M}(X_{\text{train}})$ ▷ compute prediction
      $L \leftarrow \mathcal{L}(Y'_{\text{train}}, Y_{\text{train}})$ ▷ compute loss
      $g \leftarrow BP(L)$ ▷ back-propagate
      $\mathcal{M}(\cdot) \leftarrow \mathcal{M}(\cdot) + \text{ADAM}(g)$ ▷ update model parameters
    $Y'_{\text{val}} \leftarrow \mathcal{M}(X_{\text{val}})$
    $L_{\text{val}} \leftarrow \mathcal{L}(Y'_{\text{val}}, Y_{\text{val}})$ ▷ compute validation loss
    if L_{val} is the smallest **then**
      $M \leftarrow \mathcal{M}(\cdot)$ ▷ save the best model as M

The training workflow can be summarized in the pseudocode described as Algorithm 3, which is modified from Algorithm 1 in Chapter 1. The main modifications are explained below.

We implemented this algorithm using PyTorch (Paszke et al., 2017), which requires making zero the gradients at each iteration. Otherwise, the gradients will accumulate and cause long-wavelength oscillations in the loss curve and hence the model will fail to converge. To save memory usage, datasets are usually loaded in batches. In the inner loop, the model is trained with one batch at a time. After each full cycle of epochs, the mini-batches will be shuffled for stochastic gradient updates. Batching is not always necessary for the validation phase. This is because backpropagation is not involved, which will consume a large piece of memory for storing intermediate gradients. In practice, validation loss is usually calculated in bulk at the end of each epoch.

2.3 Synthetic data examples

2.3.1 Data preparation

All data used in this chapter were generated synthetically with a finite difference method. We applied forward modelling with the acoustic wave equation for simplicity. Second-order approximation was used in the time domain, while eighth-order approximation was used in the space domain. The blended data was created by injecting shots simultaneously with random delays and measuring the total wavefield in the receiver locations using the velocity model shown in Figure 2.2. This model contains three layers and a wedge on the left, with several point scatterers under the dipping layer. These scatterers are intended to test whether the deblending algorithm can honour data diffractions. The dipping layer of the wedge moves the apexes of reflections in the shot domain. In this model, 64 supershots were recorded with four shots blended in each and with 512 receivers. No artificial noise was added to the modelling process. However, reflections from boundaries are still present. The direct waves and source effects are removed by subtracting the shot record modelled with the velocity of the first layer. The data was resampled by increasing the time-step size and the number of time samples was reduced from 3600 to 512 to reduce the computation cost. Both sources and receivers are evenly distributed at the near-surface. Also, we created for the training a regular data set without blending or time delays, which we call here "true data".

First, the blended data were pseudo-deblended as follows: the supershots are repeated as many times as the number of blended shots per supershots, and the copies are concatenated in the shot axis with the time delays removed one shot at a time. After this pseudo-deblending, only those shots whose time delay were completely removed become coherent in the receiver domain (Figure 2.3). Since duplications of supershots were concatenated together, the blended data now has dimensions of $N_g \times N_t \times N_{\text{shot}}$, which refers to the number of receivers, timesteps, and shots, respectively. The number of shots here is the

product of supershots and the number of blended shots. We treat each receiver gather as a picture which we feed to the network for training. The time and source axis becomes the height and width of the picture. We also have to define an extra dimension in the 2$^{\text{nd}}$ place to represent the number of channels. Since the input is in grayscale, the number is 1. Therefore, the input tensors has the format of $N_{\text{sample}} \times 1 \times H \times W$ which is $512 \times 1 \times 512 \times 256$ where N_{sample} is the number of receiver gathers. The "true data", that is receiver gathers without blending, have the same dimensions as well (Figure 2.3). Both inputs and labels were normalized to be ranging from 0 to 1 for better generalization, as indicated by the scale bar.

The dataset was then separated into training and validation sets. In this problem, the randomly chosen $20\,\%$ of the entire dataset becomes the validation set and the rest becomes the training set that will be used for calculating the gradient. We did not split the data from test data set because the amount of data is limited and we want to make more data available for training. However, it is still worth having a validation set to give some confidence of not overtraining the model. We test the trained model's performance on a different velocity model, which is shown later in Figure 2.8. Furthermore, it is fair to assume all the data comes from the same distribution due to the fact that all data was synthesized from one model setup. Therefore, the effect of using the cross-validation method is marginal compared to a static validation set.

2.3.2 Training

We used PyTorch for the machine learning framework and adapted the U-Net implementation described in Buda et al., 2019, which was designed for brain MRI. The U-Net has four tiers in depth with two 3 by 3 convolutional layers in each block with zeros padding of 3 at each boundary, which guarantees the inputs and outputs having the same dimension. The original model was designed to take inputs with three channels as RGB images and has 32 filters in the initial layer. In this chapter, however, since the receiver gathers only have one

channel, the default 32 filters may be more than needed. We discuss later in the chapter our choice for the number of initial filters. The U-Net uses ReLU as inter-layer activation functions and uses batch normalization layers. The output activation is sigmoid, which regularizes the outputs to a $(0, 1)$ range.

After some testing and experimentations, we decided to train the model by following Algorithm 3. We used ADAM optimizer with a learning rate of 0.002, $\alpha = 0.9$ and $\beta = 0.999$. To mitigate large oscillations at later epochs, we decrease the step learning rate every 100 iterations. After each completion of 100 epochs, we reduce the learning rate to its $10\,\%$. This learning rate decay slows down the descent in later epochs but makes it tolerant to a bigger learning rate at the early stage (the default learning rate for the brain MRI problem was 0.0001). Figure 2.4 shows the loss curves. The validation loss reaches a plateau with small oscillations at 200$^{\text{th}}$ to 300$^{\text{th}}$ epochs. Models at those iterations can be considered to have the same confidence level. One could choose the model at the last epoch since it undergoes more training but we picked the model with the least L_{val} to avoid over-fitting (see the red cross in Figure 2.4).

As mentioned in the previous section, the initial number of filters defines how many features are extracted from the inputs. There is a trade-off between the complexity of the model and performance. The more features are extracted, the more information from the inputs are used for training but with more computational cost. Figure 2.5 shows three validation losses with different initial numbers of filters. The saved model is summarized in Table 2.1. Losses with 16 and 32 filters performed almost the same. The model with 16 filters descends faster than the model with 32 filters because of being simpler, but the later achieves lower validation loss and requires fewer iterations. The model with 8 initial filters shows poor results. Having too few parameters did not help with the model updates, which may be evidence that 8 filters are not enough to capture the important information in this problem. The model with 32 filters has better accuracy but also takes almost double time to calculate and memory to train. After balancing these trade-offs, we chose the model with

16 initial filters.

Table 2.1: Best epochs and the corresponding L_{val} for different initial filters.

# filters	$\min(L_{\text{val}})$	Best epoch
8	3.321×10^{-6}	295
16	1.318×10^{-6}	280
32	1.207×10^{-6}	241

Figure 2.6 shows an example for a prediction from the validation set. Most of the incoherent noise is removed. Furthermore, the diffractions from the point scatterers are mostly preserved although with some attenuation in their tail endings. The larger errors are concentrated in two regions. The first region is inside the major primary, where the reflections get complicated. Probably the identification of the reflections becomes difficult for the algorithm and the interference complicates this. It is important to keep in mind that the network does not know what reflections or diffractions are, but just see them as patterns. Furthermore, we can also see some meshed patterns at the bottom of the plot. These patterns could be multiples of the point scatterers' reflections or boundary artifacts. Likely the model behaves poorly for them because of their weak amplitude and complexity. The second region is around the tails of the reflection. Something to mention is that the input data have some missing samples due to the removal of time delays, but the "true" data do not. To get the right prediction, the model tries not only to remove the incoherent noise but also to interpolate the missing samples, which itself is a complex problem to solve. Therefore, the prediction contains relatively larger errors after training on these points.

Figure 2.7 shows nine predictions on all samples from the train and validation set. Some shadows of the blended shots are still present. Probably this is because in shots from the acquisition edges the blended reflections are not like a typical hyperbola and are different from the majority of the receiver gathers. Therefore, the model fails to resolve the

signal and noise in this case. One solution could be to use gradient boosting, which trains several models iteratively to adapt to different situations. However, additional models will introduce more model parameters and extra attention must be paid to avoid over-fitting. Another point to notice is that the error seems larger in the shot domain than in the receiver domain, in which the model was trained on. This may be solved by feeding the data in multiple receiver gathers or even the entire volume instead of single gathers. However, this cannot be done with one velocity model and indeed need exponentially larger computation resources for the training.

So far, we trained the network with data from the same model on which we want to perform deblending. It remains to see how the trained network will generalize to other models. Figure 2.8 shows the result from applying the model trained on the wedge to data from a two-layer model. This is a relatively easy test since the model on which we are applying the network is simpler than the wedge model (Figure 2.2) on which we trained. The results are not as good as in the first case, as expected. The primaries are resolved well but the model gets confused inside the primaries (see shot 2, 3, 7 and 8). Even if the two-layer model is an easier problem to solve, its data have different distributions than the training dataset. We expect that results will improve by training on several different models instead of one.

2.4 Conclusion

In this chapter, we trained a U-Net model to perform deblending, that is the separation of coherent and incoherent signal coming from blended shots. We tried several optimizations and network parameters and found the best combination for the current problem setup. When the training and test data come from the same velocity model, the network performed well by preserving small diffractions and correctly identifying primaries. It performed slightly worse for the shots at the edge of the model because of the lack of training pictures

representative of this case. We also tested the performance with data that comes from a different velocity distribution than the training data. Specifically, the velocity model was made to be simpler than the training model, so the input should not exceed the trained model's capability. In this case, however, the network performs okay but not as well as the first case. This observation indicates the model still memorizes part of the training data and provides a direction for improvement.

More work is required to fully understand how to generalize the network to new problems. The direction is to apply constraints and reduce flexibility. To address these issues, one can investigate generalizing the model by gradient boosting, which gives balanced outputs from several trained models. Another direction is to make the network predict the residual instead, so it can be trained and applied iteratively like the least-square methods.

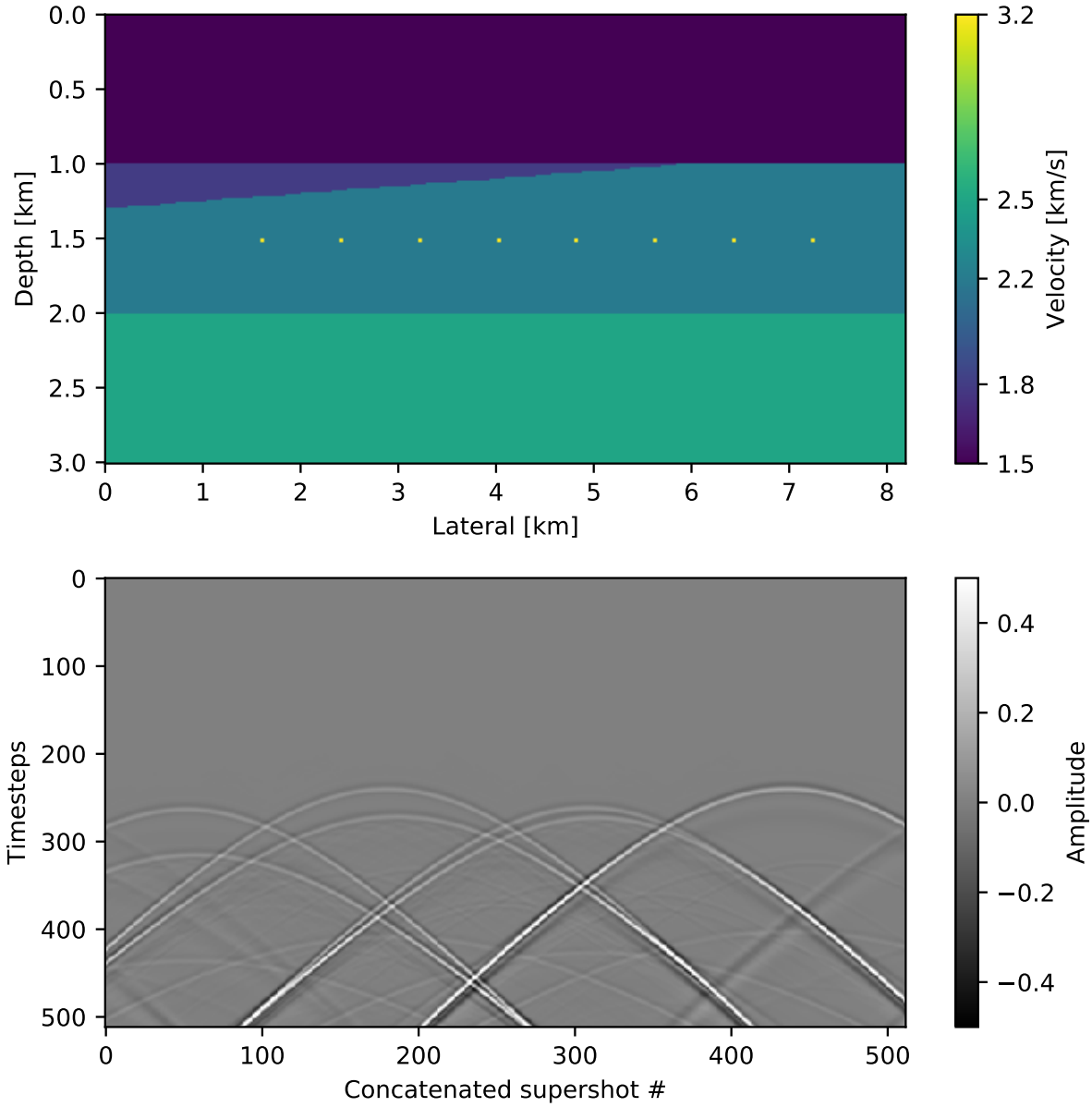

Figure 2.2: The supershot generated by the finite difference method. The figure below refers to an example supershot corresponding to the above velocity model. There are four shots in each supershot and random delays are applied.

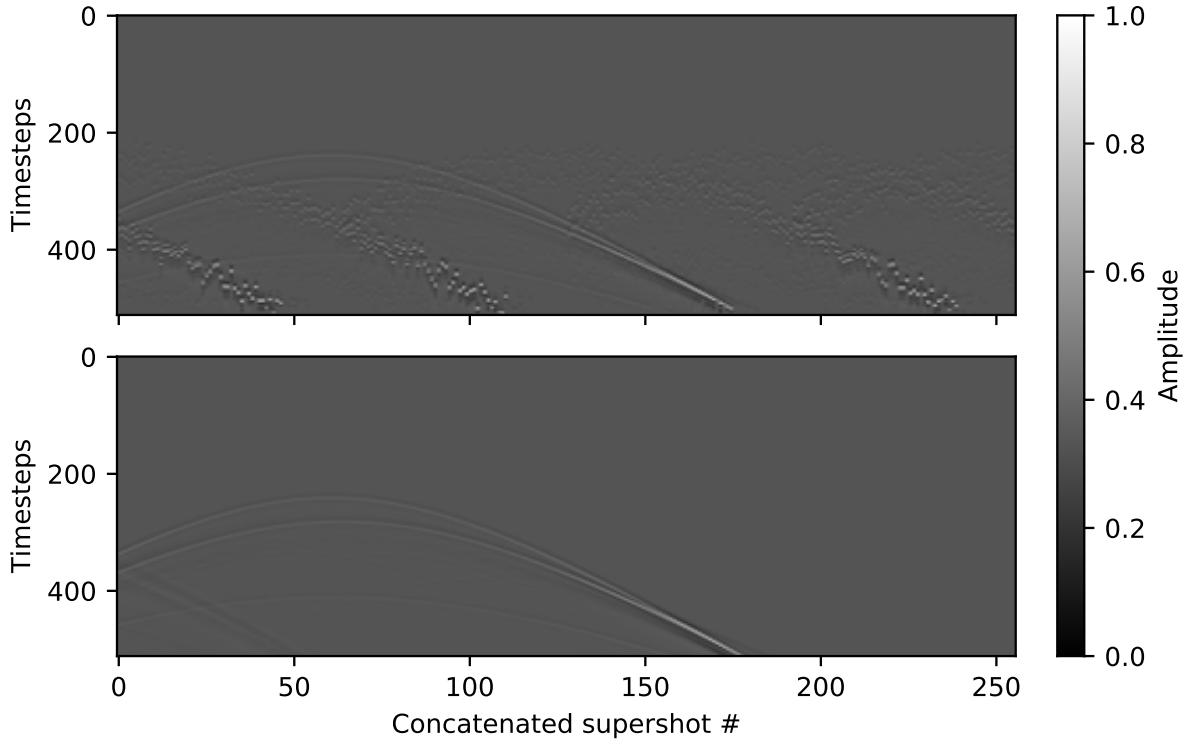

Figure 2.3: The inputs fed to the U-Net model. The plots show the corresponding input (above) and label (bellow) pair at the 120^{th} receiver, with 512 receiver slices in total.

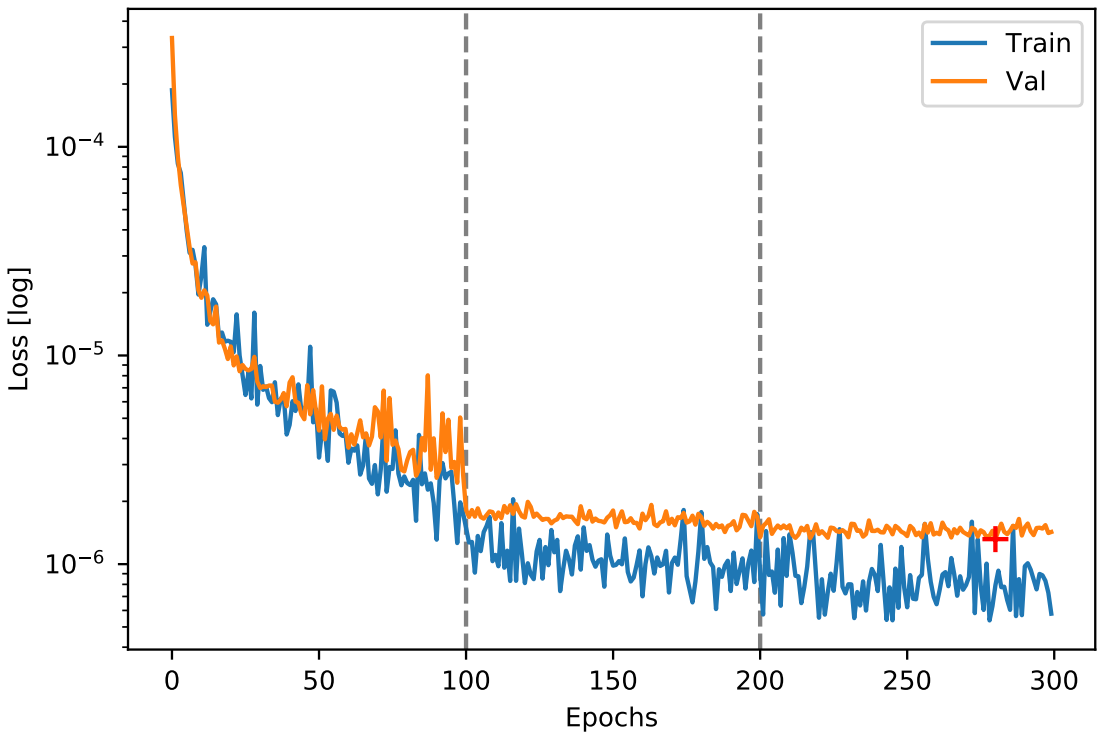

Figure 2.4: The loss curve when initial filters is 16. The blue line refers to the training loss while the orange line is the validation loss (L_{val}). The red cross indicates the least L_{val}, which is 1.318×10^{-6} at epoch 280. The gray dashed lines separate regions with different learning rates.

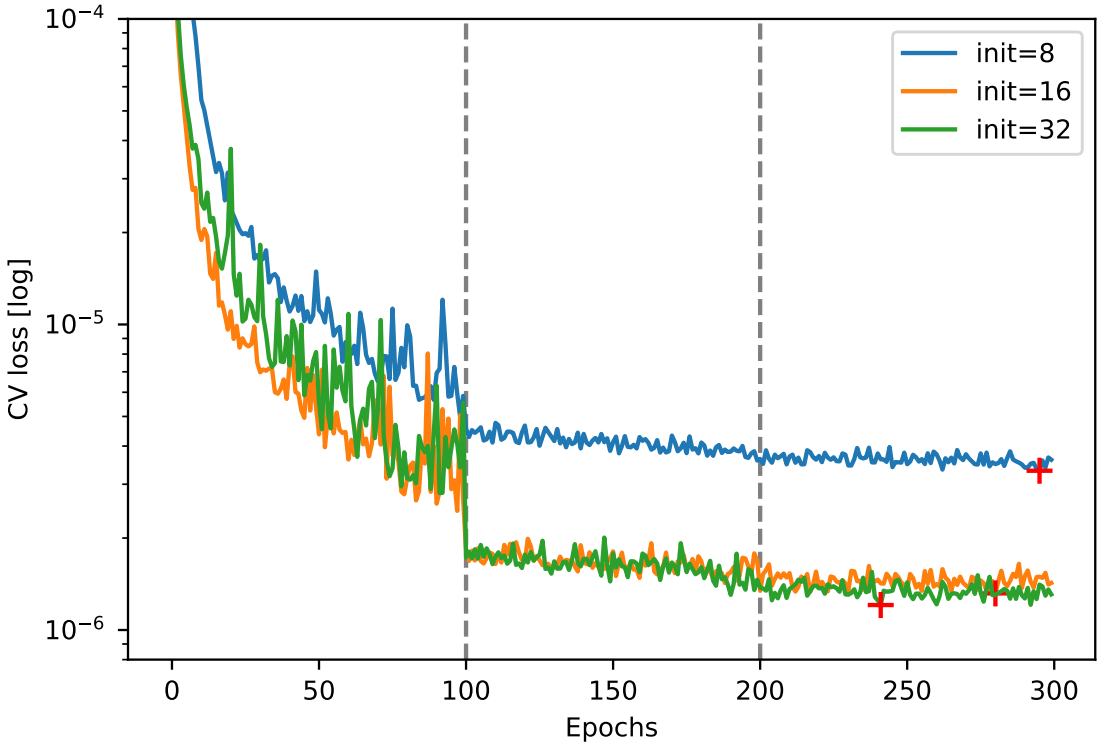

Figure 2.5: The cross comparison of L_{val} with varying initial filters. The blue, orange and green line refer to the case with 8, 16 and 32 intial filters, respectively. Red crosses stand for the least L_{val} on each line. The results are summarized in Table 2.1.

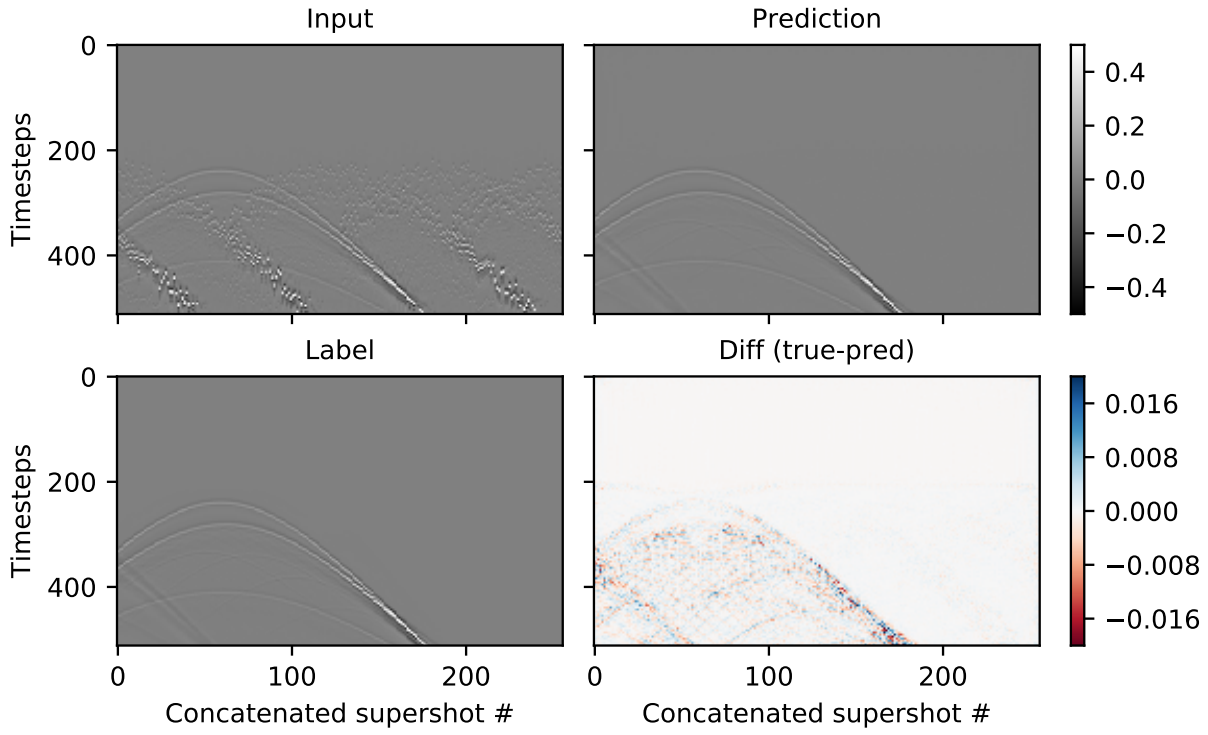

Figure 2.6: The prediction and label for a sample in the validation set. All three grayscale images have the same scale. The picture in the bottom right shots the difference of the prediction and the label, with a smaller color scale.

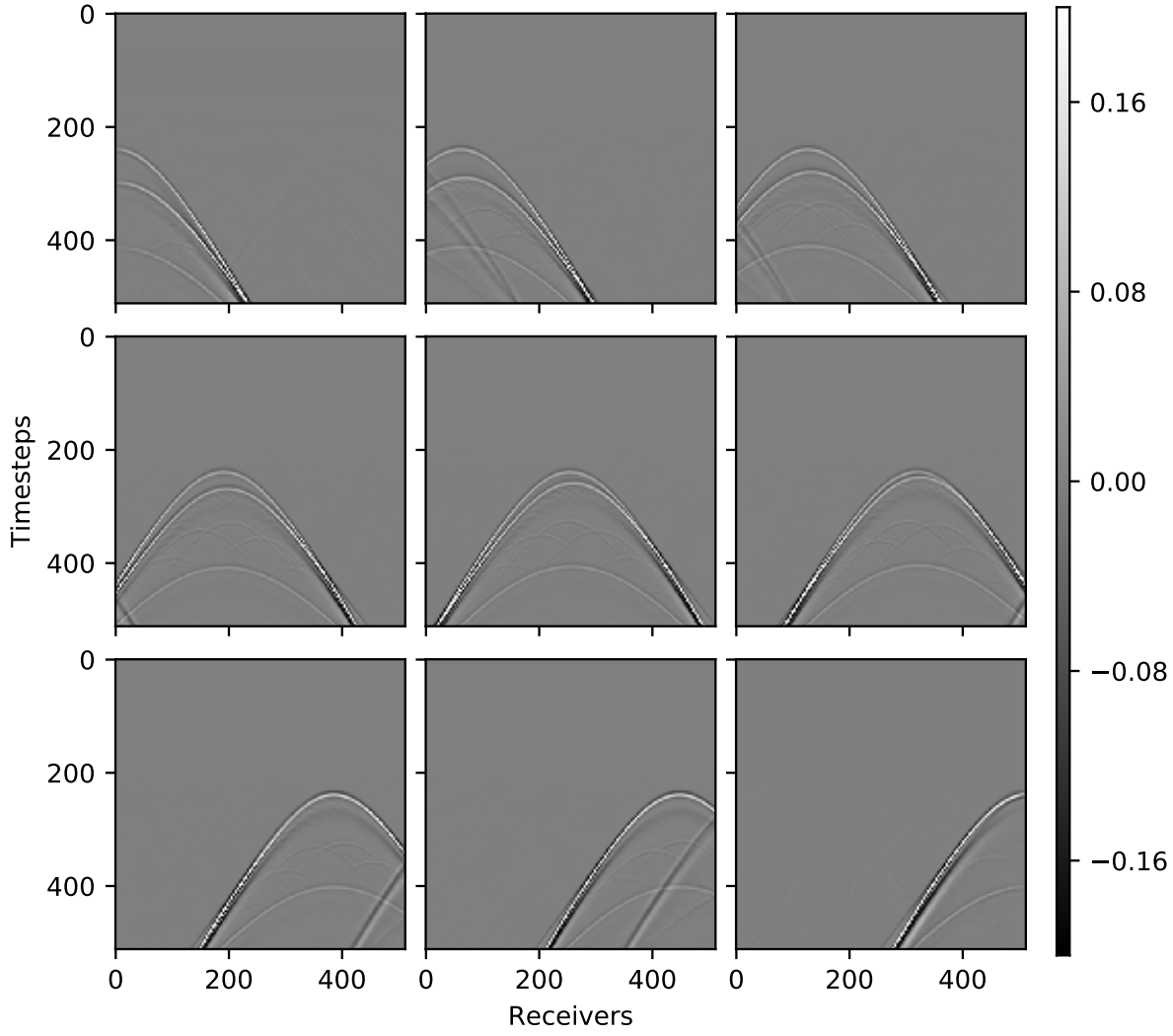

Figure 2.7: The prediction on the whole dataset containing both the training and validation set (transposed to the shot domain). Note the preservations of the diffractions.

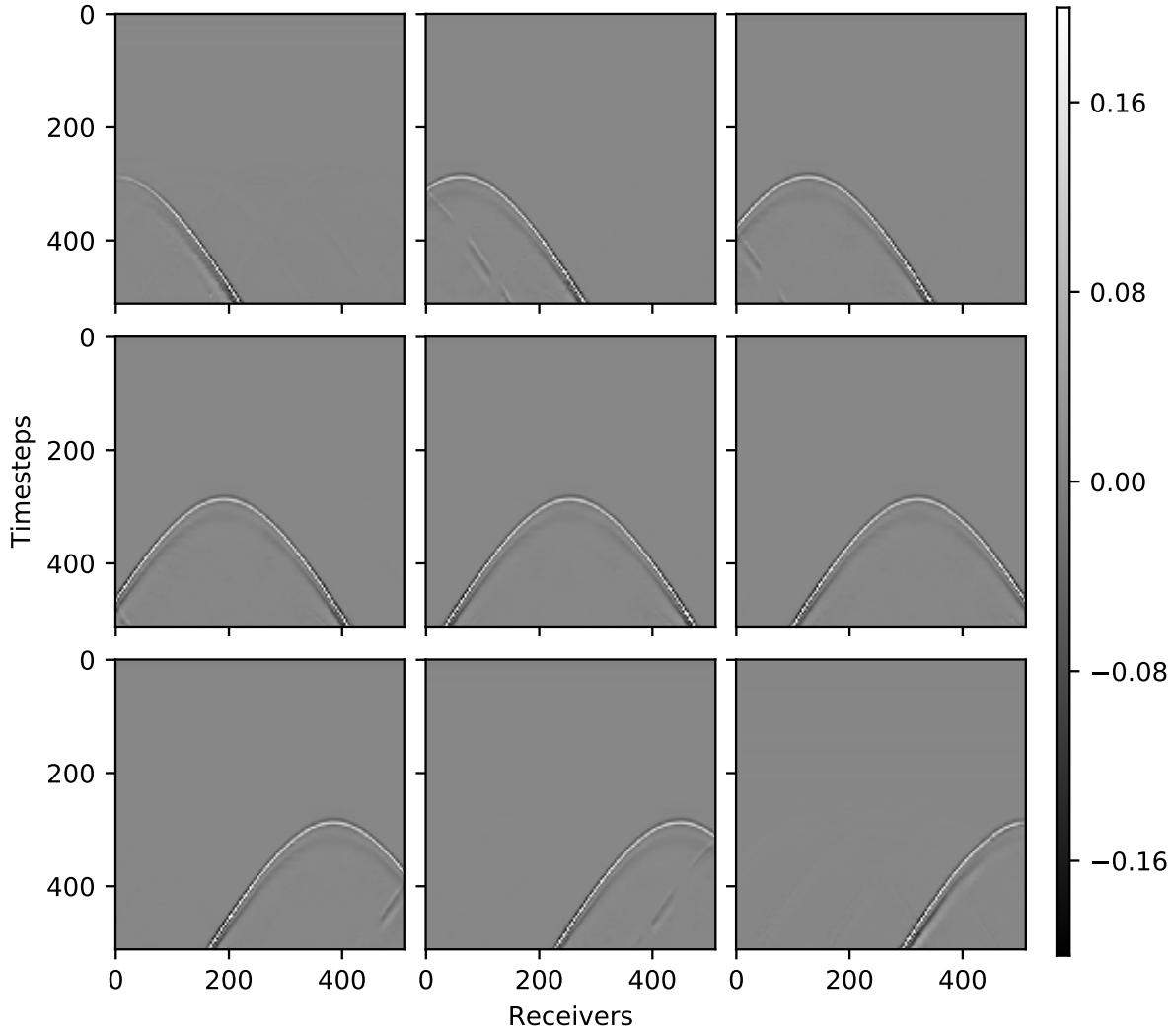

Figure 2.8: Predictions for blended data from a two-layer model.

Chapter 3

Born inversion with recurrent neural networks

In the previous chapter, we showed an example of tackling the deblending problem using a convolution-based neural network. We let the theory to guide the machine learning solution by carefully generating the input data and choosing a suitable architecture. This is like most machine learning algorithms treat problems from a statistical perspective. In this chapter, we will lean more to the theoretical side of the continuum mentioned in Chapter 1.

Linear regression, as a straightforward example, extracts features (model parameters) from the given data and learns how to correlate the selected features to the data (I. Goodfellow et al., 2016). By fitting a line through a cluster of points, we assume that there is a linear trend between the independent and dependant variables. By using this limiting assumption, we greatly reduced the number of unknowns to just two, the slope and the intercept coefficients. This is an example of how prior-knowledge can be injected into a statistical problem to solve it with less effort.

Even the most fundamental fully connected neural network contains injected knowledge—it requires the number of layers and the number of nodes to be predefined, which is related to how many orders of non-linearity need to be simulated. The Convolutional Neural Net-

works (CNNs), a more complex model that is popular in image recognition/segmentation, injects the idea that a feature at a given point only depends on its nearby points (Fukushima, 1979; LeCun et al., 2015). This assumption significantly reduces the burden on the learning process compared to fully connected neural networks. The recurrent neural network (RNN, Lipton et al., 2015) feeds in the knowledge that the output at a current state depends on the features at the current state and the previous states. This causal characteristic makes the network time- or sequence-relevant and makes RNN a perfect match for a complex job like word recognition and natural language processing. As a general rule, the more knowledge is fed into the neural network/structure, the less flexible is the model. The return of yielding flexibility is easier training and the reduced risk of diverging.

Most studies applying machine learning to wave simulation in geophysics treat the forward problem as a black box, selecting the velocity as a feature (Moseley et al., 2018). The black box idea solves the problem statistically, which follows the classical machine learning philosophy, but it ignores theoretical knowledge that has been well studied in geophysics (e.g. the wave equation and scattering theory). With the neglect of those crucial theories, the neural network will spend too much energy finding approximations to the theories by itself. The approximations are usually poor and highly dependent on the problem trying to solve. Some recent works in geophysics proposed that we shall inject our knowledge to the machine learning algorithm and only treat part of it to be a numerical problem (Karpatne, Atluri, et al., 2017).

In this chapter, we attempted to add wave propagation knowledge to an RNN. By following a similar idea from Richardson, 2018 and J. Sun et al., 2020, we incorporated the Born approximation into the structure of the RNN. The RNN takes a background velocity and the source as input features. We designed the network structure from scratch to make the velocity perturbation to be the hidden parameter, and the output is a shot record. We performed an inversion of the model by training the RNN with generated data. The inversion process can be proven to be the same as least squares reverse time migration

(LSRTM). We implemented the RNN based on the APIs in TensorFlow (Abadi et al., 2016), tested popular machine learning optimizers and discuss their performances. As a test result, we concluded that the ADAM optimizer is the most stable and time-efficient for this method.

3.1 Theory

3.1.1 Forward modelling with the Born approximation

Consider a wavefield $\boldsymbol{p}_0(\boldsymbol{x}, t)$ propagating in a background velocity $\boldsymbol{v}_0(\boldsymbol{x})$ with a source $\boldsymbol{f}(\boldsymbol{x}, t)$. This wavefield obeys the wave equation

$$\left(\frac{1}{\boldsymbol{v}_0^2}\frac{\partial^2}{\partial t^2} - \nabla^2\right)\boldsymbol{p}_0 = \boldsymbol{f} \tag{3.1}$$

Let us consider another wavefield $\boldsymbol{p}(\boldsymbol{x}, t) = \boldsymbol{p}_0(\boldsymbol{x}, t) + \delta\boldsymbol{p}(\boldsymbol{x}, t)$ propagating in a velocity media $\boldsymbol{v}(\boldsymbol{x}) = \boldsymbol{v}_0(\boldsymbol{x}) + \delta\boldsymbol{v}(\boldsymbol{x})$ with the same source, where $v(\boldsymbol{x})$ is a velocity that differs from $\boldsymbol{v}_0(\boldsymbol{x})$ by $\delta\boldsymbol{v}(\boldsymbol{x})$ and $p(\boldsymbol{x}, t)$ is the corresponding wavefield with respect to $\boldsymbol{v}(\boldsymbol{x})$. The wavefield is hence governed by

$$\left(\frac{1}{(\boldsymbol{v}_0 + \delta\boldsymbol{v})^2}\frac{\partial^2}{\partial t^2} - \nabla^2\right)(\boldsymbol{p}_0 + \delta\boldsymbol{p}) = \boldsymbol{f} \tag{3.2}$$

If it is assumed that $\frac{\delta\boldsymbol{v}}{\boldsymbol{v}_0} \to 0$, then

$$\frac{1}{(\boldsymbol{v}_0 + \delta\boldsymbol{v})^2} = \frac{1}{\boldsymbol{v}_0^2}\left(1 + \frac{\delta\boldsymbol{v}}{\boldsymbol{v}_0}\right)^{-2} \approx \frac{1}{\boldsymbol{v}_0^2}\left(1 - 2\frac{\delta\boldsymbol{v}}{\boldsymbol{v}_0}\right) \tag{3.3}$$

Replacing Equation 3.1 in Equation 3.2

$$\left(\frac{1}{\boldsymbol{v}_0^2}\frac{\partial^2}{\partial t^2} - \nabla^2\right)\delta\boldsymbol{p} = \frac{2\delta\boldsymbol{v}}{\boldsymbol{v}_0^3}\frac{\partial^2}{\partial t^2}(\boldsymbol{p}_0 + \delta\boldsymbol{p}) \tag{3.4}$$

If we apply the Born approximation assuming the second time derivative of $\delta \boldsymbol{p}$ is negligible, we can write two wave equations for the background and perturbed wavefield as follows:

$$\left(\frac{1}{\boldsymbol{v}_0^2}\frac{\partial^2}{\partial t^2} - \nabla^2\right)\boldsymbol{p}_0 = \boldsymbol{f} \tag{3.5a}$$

$$\left(\frac{1}{\boldsymbol{v}_0^2}\frac{\partial^2}{\partial t^2} - \nabla^2\right)\delta\boldsymbol{p} = \frac{1}{\boldsymbol{v}_0^2}\boldsymbol{m}\frac{\partial^2}{\partial t^2}\boldsymbol{p}_0 \tag{3.5b}$$

where $\boldsymbol{m}$ is the model defined as velocity perturbation ($\frac{2\delta \boldsymbol{v}}{\boldsymbol{v}_0}$). The perturbation wavefield can be considered as the wavefield of a source that is the zero-lag cross-correlation of the model and the second time derivative of the background wavefield. With Equation 3.5a and 3.5b, we can calculate the perturbation wavefield ($\delta \boldsymbol{p}$) given a velocity perturbation $\boldsymbol{m}$ from $\boldsymbol{v}_0$.

3.1.2 The implementation using TensorFlow

Based on the fact that the wave propagation is a function of time, the most suitable neural network framework for simulating the wavefield is the RNN, which already contains time structures. The RNN can be built with the help of TensorFlow, which brings the power of parallel computing with GPUs. Tensorflow provides a flexible python Application Programmer Interface (API) that calls the CUDA library "cudnn". Moreover, as one of the popular machine learning packages, TensorFlow has an API to compare different optimization methods systematically. Figure 3.1 shows the RNN architecture used for this chapter.

As shown in Figure 3.1, each cell refers to a series of calculations to compute the perturbed wavefield. We decided to use the 3-point-centred finite difference expansion for the second order time derivative since it is accurate enough for small time steps. In the space domain, we also chose a second order finite difference for prototype development, which can be improved to a more accurate approximation in the future. The second order

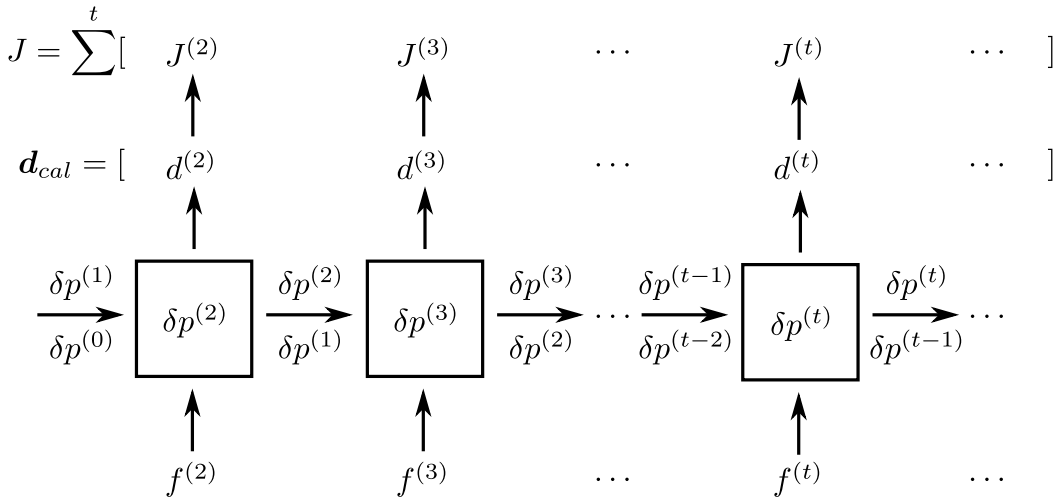

Figure 3.1: The diagram of the RNN structure. The black boxes are neural cells that take the source and two previous perturbation wavefields to compute the next perturbation wavefields. The output of each cell is the shot record at a given time, and the most recent two wavefields will be passed to the next cell.

approximation in the space is usually not adequate but is fast and stable. After discretization, Equation 3.5a and 3.5b can be rearranged to

$$\boldsymbol{p}_0^{(t+1)} - 2\boldsymbol{p}_0^{(t)} + \boldsymbol{p}_0^{(t-1)} = \boldsymbol{v}_0^2 \Delta t^2 \left(\nabla^2 \boldsymbol{p}_0^{(t)} + \boldsymbol{f} \right) \tag{3.6a}$$

$$\delta\boldsymbol{p}^{(t+1)} - 2\delta\boldsymbol{p}^{(t)} + \delta\boldsymbol{p}^{(t-1)} = \boldsymbol{v}_0^2 \Delta t^2 \left(\nabla^2 \delta\boldsymbol{p}^{(t)} + \frac{m}{{\boldsymbol{v}_0}^2} \left(\nabla^2 \boldsymbol{p}_0^{(t)} + \boldsymbol{f} \right) \right) \tag{3.6b}$$

where the superscript (t) denotes a variable at the t^{th} time step. Note that the time derivative ($\frac{\partial^2}{\partial t^2}\boldsymbol{p}_0$) on the right hand side of Equation 3.5b is replaced by the equality in Equation 3.5a. Instead, in Equations 3.6a and 3.6b, the wavefields at the current time step can be calculated if the previous two wavefields are known or given. With the wavefield at the current cell, $d^{(t)}$ can be extracted from the wavefields. Then the wavefields at (t) and $(t-1)$ will be moved forward to the next cell to go through a similar process until it reaches the maximum time step. The shot record ($\boldsymbol{d}_{\text{cal}}$) can be formed by concatenation of $d^{(t)}$ from each cell to produce a synthetic wavefield. This wavefield can be compared with the observed data for

either full waveform inversion, to estimate velocity errors, or for least squares migration, to estimate a reflectivity model.

3.1.3 The gradient update and optimization

In the previous section, we introduce the procedure to use an RNN to perform forward modelling. Neural networks problems are essentially optimization problems. If we let a model parameter to be unknown and feed the generated data by the forward modelling, the model can be solved by "training" of the neural network.

The derivation of the gradient follows a similar idea as described by Richardson, 2018. A cost function penalizing modelling errors can be defined as

$$J = \frac{1}{2n_s}\sum_{\boldsymbol{x}_s}\|\boldsymbol{D}-\boldsymbol{d}_{\text{cal}}\|^2 = \frac{1}{2n_s}\sum_{\boldsymbol{x}_s}\sum_{\boldsymbol{x}_r}\sum_{t}(\boldsymbol{D}-\boldsymbol{d}_{\text{cal}})^2 = \frac{1}{2n_s}\sum_{\boldsymbol{x}_s}\sum_{\boldsymbol{x}_r}\sum_{t}\boldsymbol{r}^2, \tag{3.7}$$

where $\boldsymbol{D}$ stands for the observed data and $\boldsymbol{d}_{\text{cal}}$ is the prediction calculated by the Born modelling. $\boldsymbol{r}$ refers to the data residual. A gradient of this cost function with respect to the unknown parameters, the model, will tell us how the error changes as the parameter changes. This gradient will provide us with information on how to change the model to decrease the prediction error. Since the error was summed through out the time, the gradient is the sum of all gradients at each time step as well:

$$\boldsymbol{g}_i = \sum_{t=0}^{T}\boldsymbol{g}_i^{(t)} = \sum_{t=0}^{T}\frac{\partial J^{(t)}}{\partial \boldsymbol{m}}. \tag{3.8}$$

The gradients at each time step can be proved to be the dot product of the time-reverse-propagated wavefield of the data residual ($\boldsymbol{r}$) and the 2nd time derivative of the time-forward-propagation of the background wavefield ($\boldsymbol{p}_0$), i.e.

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{m}} = \frac{1}{{\boldsymbol{v}_0}^2}\boldsymbol{B}(\boldsymbol{r},\boldsymbol{x},T-t)\frac{\partial^2}{\partial t^2}\boldsymbol{p}_0(f,\boldsymbol{x},t) \tag{3.9}$$

(proof is developed in the Appendix). Therefore, the optimization problem for the inverse of Born modelling can be treated in the same manner as a LSRTM inversion.

Optimization by adaptive moment estimation (ADAM)

For each iteration of the training step, one can update the model using gradient descent as following:

$$\boldsymbol{m}_i = \boldsymbol{m}_{i-1} - \alpha \boldsymbol{g}_i \tag{3.10}$$

where α is the learning rate that is usually smaller than 1 ($\alpha = 1$ means the full step along the gradient). However, the negative direction of the gradient is not necessarily the direction towards the local minimum. The traditional gradient descent method is not costly to calculate at each iteration but usually takes a zigzag path to the optimal solution. Adaptive moment estimation (ADAM) is a first-order optimization method that can suppress the oscillations that commonly appear in the gradient descent method. Similar to conjugate gradients, ADAM chooses a more optimal path based on previous updates. This optimization method combines the advantages of momentum and root mean square propagation (RMSprop) (Kingma & Ba, 2014).

ADAM needs three (hyper-) parameters to be set manually: 1) α refers to the step length of the gradient update, which is similar to what is used in other gradient methods; 2) β_1 and 3) β_2 are extra parameters to control how much the new gradient is related to previous gradients. These parameters are used to calculate two momentum terms, which are the accumulative sum of the first order and second order of the gradient in previous iterations. These momentums are defined as

$$\boldsymbol{v} \leftarrow \beta_1 \boldsymbol{v} + (1 - \beta_1)\boldsymbol{g} \tag{3.11a}$$

$$\boldsymbol{s} \leftarrow \beta_2 \boldsymbol{s} + (1 - \beta_2)\boldsymbol{g}^2. \tag{3.11b}$$

Instead of the model update described in Equation 3.10, ADAM optimizer updates the

model in the following way:

$$\boldsymbol{m}_i = \boldsymbol{m}_{i-1} - \alpha \frac{\hat{\boldsymbol{v}}}{\sqrt{\hat{\boldsymbol{s}}} + \epsilon} \tag{3.12}$$

where $\hat{\boldsymbol{v}}$ and $\hat{\boldsymbol{s}}$ are $\boldsymbol{v}$ and $\boldsymbol{s}$ normalized by $(1 - {\beta_1}^i)$ and $(1 - {\beta_2}^i)$, respectively. The division is point-wise, and ϵ is a regularization coefficient to avoid the division by zero. α refers to the learning rate. The pseudo-code provided by Kingma and Ba, 2014 is shown below as Algorithm 4.

Algorithm 4 The ADAM optimization

$\boldsymbol{v}_0 \leftarrow 0$
$\boldsymbol{s}_0 \leftarrow 0$
$i \leftarrow 0$
while $i <$ #iter **do**
  $i \leftarrow i + 1$
  Calculate $\boldsymbol{g}_i$
  $\boldsymbol{v}_i \leftarrow \beta_1 \boldsymbol{v}_{i-1} + (1 - \beta_1)\boldsymbol{g}_i$
  $\boldsymbol{s}_i \leftarrow \beta_2 \boldsymbol{s}_{i-1} + (1 - \beta_2){\boldsymbol{g}_i}^2$
  $\hat{\boldsymbol{v}}_i \leftarrow \frac{\boldsymbol{v}_i}{1-{\beta_1}^i}$
  $\hat{\boldsymbol{s}}_i \leftarrow \frac{\boldsymbol{s}_i}{1-{\beta_2}^i}$
  $\boldsymbol{m}_i = \boldsymbol{m}_{i-1} - \alpha \frac{\hat{\boldsymbol{v}}_i}{\sqrt{\hat{\boldsymbol{s}}_i} + \epsilon}$

Generally speaking, the numerator of the update $\hat{v}$ can be interpreted as the weighted sum of gradients through iterations. Therefore the oscillation will be cancelled in terms of the vector sum. The denominator $\sqrt{\hat{s}}$ can be thought of as the weighted root-sum-square of the gradients. For the very first iteration of ADAM, the update will be a scale of alpha and the information of the gradient will be added later on. From the flow of the ADAM optimization, one can notice that the absolute value of the parameter update will never be greater than the step size α. This means the ADAM optimizer will still perform small updates even if the optimal update should be large. This is because $\hat{\boldsymbol{v}} < \sqrt{\hat{\boldsymbol{s}}}$ holds as long as gradients at each iteration are not always in the same direction. This characteristic prevents overshooting of the gradient. In our problem, the parameter to be solved is usually smaller than 1 (unless the initial guess for the velocity is very inaccurate), so the updates should be smaller than 1 as well. This is similar to classical machine learning problems, where the

hidden parameters are usually small numbers. Therefore, we can start by testing different α values from 0.001 to 0.1, which works well in most machine learning algorithms. However, for problems that need greater updates, one may need a step size that is greater than the values recommended by machine learning.

3.1.4 The Fletcher-Reeves method

The Fletcher-Reeves method (FR) is a non-linear adaptation of the traditional linear conjugate gradient method (Wright & Nocedal, 1999). The traditional linear conjugate gradient method is designed for a quadratic cost function with respect to the model parameters. Although the Born modelling is a linear method and applying non-linearity seems to be unnecessary, the cost function may not be perfectly quadratic with respect to the model. Therefore, we decided to test this method by using the FR optimizer in the SciPy package of Python (Virtanen et al., 2020), which was implemented following Wright and Nocedal, 1999. The pseudo-code is shown as Algorithm 5.

Algorithm 5 The Fletcher-Reeves method

Require: The initial guess of model $\boldsymbol{m}_0$ (zeros)
  Compute $J_0 = f(\boldsymbol{m}_0), \nabla J_0 = \frac{\partial J}{\partial \boldsymbol{m}}(\boldsymbol{m}_0)$
  Set $\boldsymbol{p}_0 = -\nabla J_0$
  $i \leftarrow 0$
  while $i < niter$ and $\nabla J_i \neq 0$ **do**
    Compute α_i and set $\boldsymbol{m}_{i+1} = \boldsymbol{m}_i + \alpha_i \boldsymbol{p}_i$
    Evaluate ∇J_{i+1}
    $\beta_{k+1} \leftarrow \frac{\langle \nabla J_{i+1}, \nabla J_{i+1} \rangle}{\langle \nabla J_i, \nabla J_i \rangle}$ ($\langle \cdot, \cdot \rangle$ refers to inner product)
    $\boldsymbol{p}_{i+1} \leftarrow -\nabla J_{i+1} + \beta_{k+1} \boldsymbol{p}_k$
    $i \leftarrow i + 1$

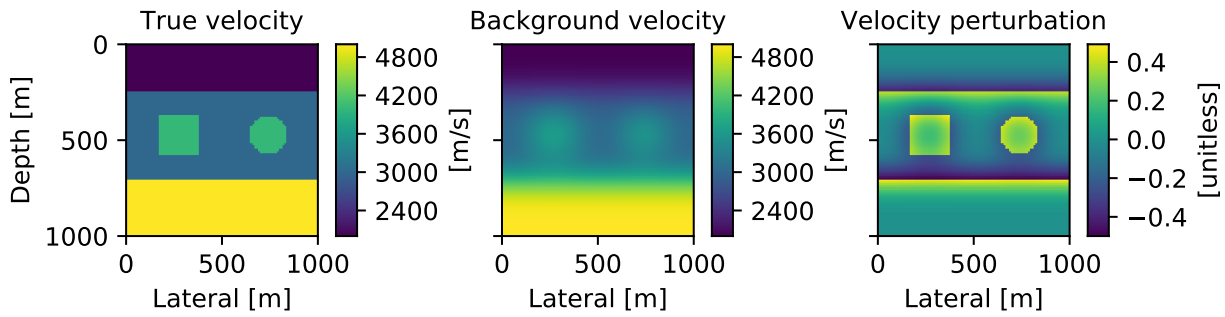

Figure 3.2: The scattering model. The true model was used for the finite difference modelling (Figure 3.3) while the background and perturbed model were used for the Born modelling method (Figure 3.4).

3.2 Synthetic data examples

3.2.1 The modelling results

We designed a simple scattering model to test the modelling accuracy using the Born approximation. The model is shown in Figure 3.2, which has a dimension of 101×101 cells with a grid spacing of $10\,\mathrm{m}$. For both the finite difference modelling and the Born modelling, 11 shots are fired at the indexes 1, 11, 21, 31, 41, 51, 61, 71, 81, 91 and 101 at the surface and the receivers are placed at each cell with the same depth. The receivers record the first $1\,\mathrm{s}$ with time step of $1\,\mathrm{ms}$. The injected source was a $15\,\mathrm{Hz}$ Ricker wavelet. Figures 3.3 and 3.4 are the corresponding synthetic wavefields, showing that finite difference and Born modelling produce similar results for this case.

3.2.2 The inversion results

We tested whether the RNN is capable of calculating model corrections from the prediction errors, that is we test the RNN back propagation algorithm. The shot record shown in Figure 3.4 was fed to the RNN as the desired output (label). At each iteration, the training process automatically finds the inverse of the prediction error and returns an estimate of the model. Then we repeated the same test on the more structured Marmousi model (Figure

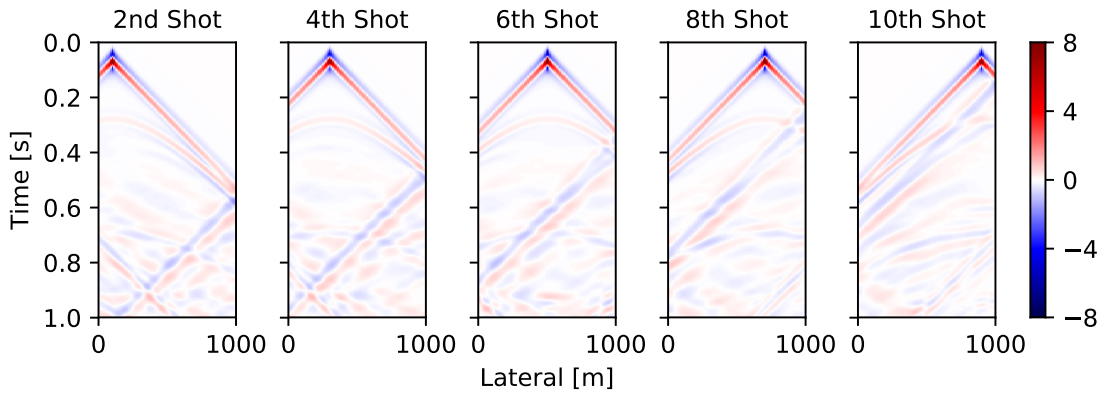

Figure 3.3: A shot record calculated by the finite difference modelling method.

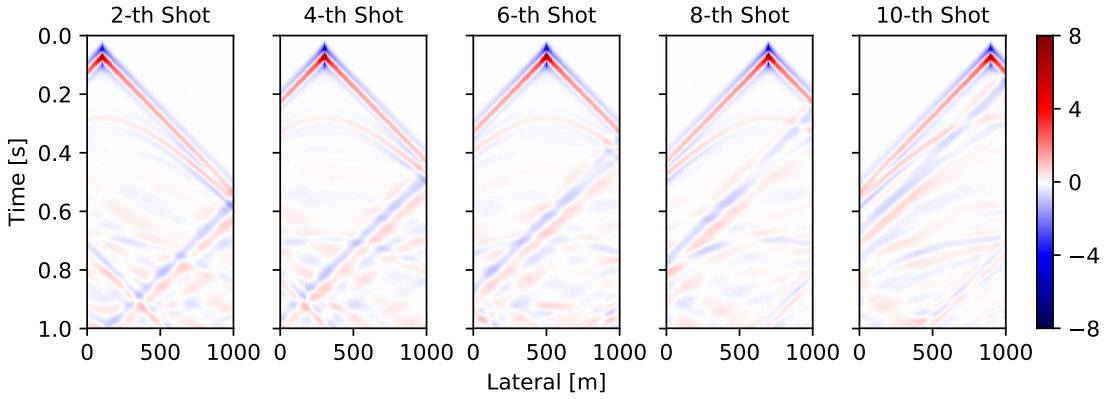

Figure 3.4: The shot record calculated by the Born modelling method.

3.7). In our tests, we found that the method does not seem to reach a stopping criterion. This is because the high-frequency component of the model is difficult to recover, which leads to the convergence rate decreasing with iterations. Therefore, even after a reasonable period of time, the cost function does not reach the accuracy threshold. To circumvent this issue, we have to set a maximum for the number of iterations. Unfortunately, the maximum number of iterations depends on the chosen optimization method. The following subsection shows the results of using the ADAM optimizer.

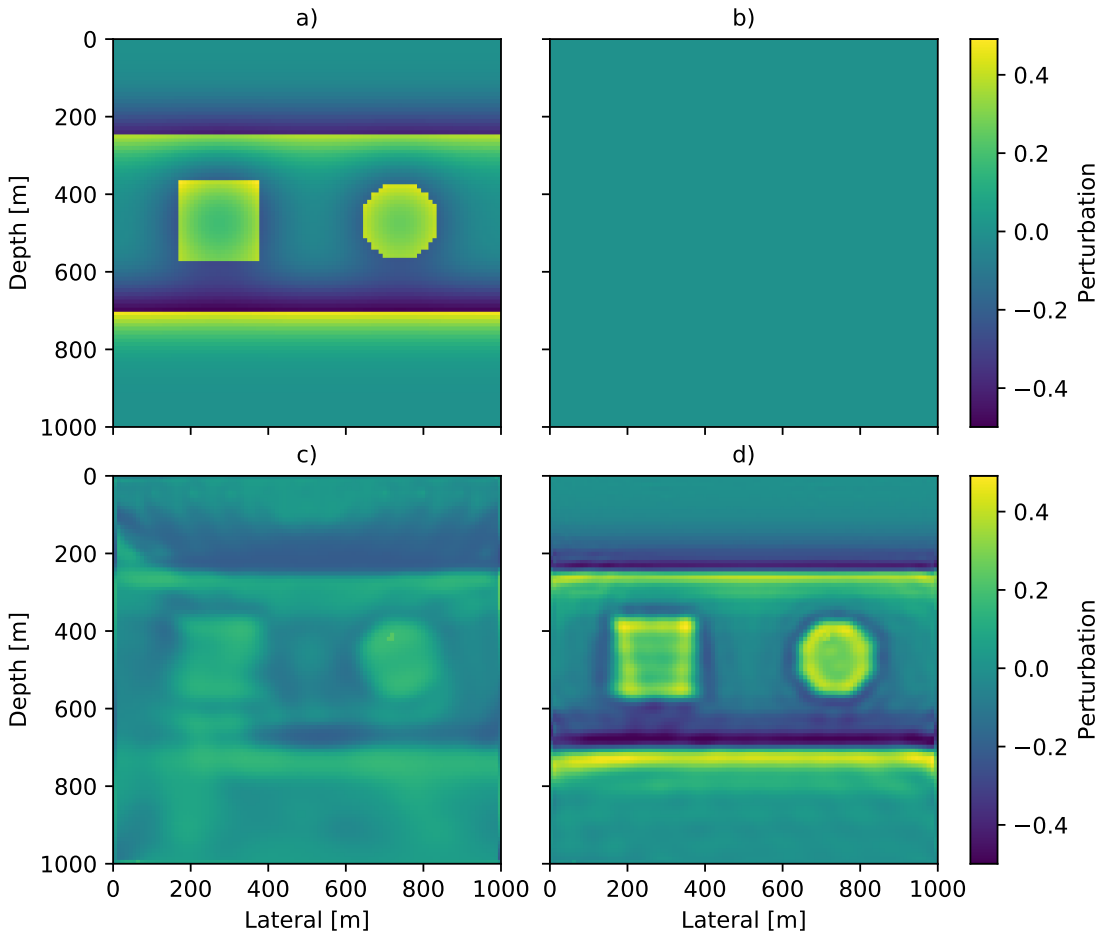

Figure 3.5: The updated model at specific iterations. a) The true model; b) The initial zero model; c) The estimated model at the 10th iteration with ADAM optimizer using a learning rate of 0.3; d) The model at the 50^{th} iteration.

The scattering model

Figure 3.5 shows the inversion result by the ADAM optimizer applied to the scattering model shown in Figure 3.2. The initial model was all zeros and the learning rate was set to a value of 0.03. We can see that the main skeleton of the model was partially recovered after 10^{th} iterations. The model kept improving and gained more high frequencies through iterations. Figure 3.6 shows the cost function response to different choices of the learning rate α. As mentioned in the theory section, β_1 and β_2 are the weighing factors that determine how much the update is related to previous gradients, while the step size is controlled by α.

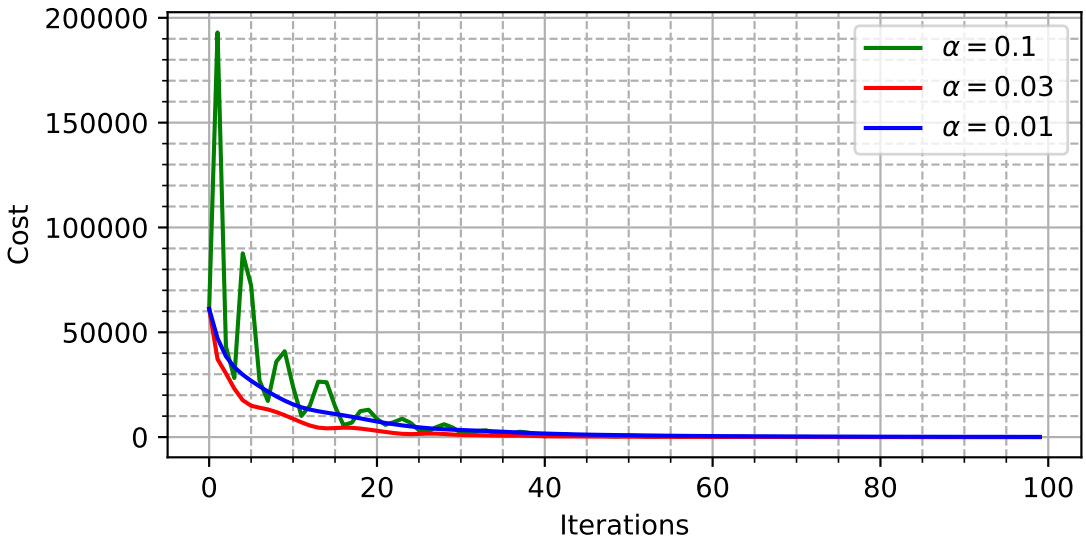

Figure 3.6: The cost functions for different value of α with $\beta_1 = 0.9$ and $\beta_2 = 0.999$ in all cases.

Since the absolute value of the update is never greater than α, the choice of α is crucial and problem-specific. For the scattering model shown in Figure 3.2, velocity perturbations range from -0.4 to 0.4. If setting the number of iterations to 400 and letting the step size be equal to α, then α should be around $0.4/400 = 0.01$. In Figure 3.6, we can see that $\alpha = 0.01$ is too conservative—the curve is smooth but it converges slower then $\alpha = 0.03$. In contrast, $\alpha = 0.1$ seems to be too aggressive—the oscillation on the curve indicates the step size used is too large. Although it is still approaching the local minima with satisfying speed, the progress is at high risk as the function may get trap into other local minima at any oscillation peaks. $\alpha = 0.03$, despite some minor imperfection, converges the fastest to the local minimum. Furthermore, the training process does not need 400 iterations to converge, and it gives good results after 50 iterations. Selected iterations of model was shown in Figure 3.5b, 3.5c and 3.5d.

The Marmousi model

One may find that even a non-linear method like FWI has difficulty estimating flat layers since the high-frequency part of the model takes a long time to converge. Therefore, we test a different model. Marmousi (see Figure 3.7) is more complex and representative of the kind of problems we would like to solve in the real world. The model grid has a dimension of 94×288 with a cell size of $10\,\mathrm{m}$. In our tests, we reduced the depth and lateral distance by a constant scale factor to reduce memory usage. Otherwise, the model would need a lower dominant frequency to satisfy the dispersion condition and would result in images with poor resolution. Figure 3.8 shows the model updates obtained at selected iterations using the ADAM optimizer with a learning rate of $\alpha = 0.03$. The red line in Figure 3.9 refers to the cost function corresponding to it. The values of the cost function are, at the initial state equal to $1\,933\,903.87$, at the 200$^{\text{th}}$ iteration equal to 247.67, and at the 300$^{\text{th}}$ iteration equal to 101.88.

3.2.3 Non-linear optimizers

Two non-linear optimization methods were used in this chapter, the FR-CG and the L-BFGS-B method. Both methods use line search, which means the cost will be computed more than once at some iterations. The green and blue lines in Figure 3.9 show cost functions for both methods, respectively.

For the FR-CG method, it is noticeable that some severe line search oscillation happens in early iterations and causes it to be slow from the beginning. The cost is $33\,246.41$ at the 200$^{\text{th}}$ function evaluation and converged extremely slow after the cost is minimized to around 2000.

For the L-BFGS-B method, there are only minor line search oscillations. The method is much faster than the FR-CG method and has also converged to a lower cost. The cost is 4044.99 at the 200$^{\text{th}}$ function evaluation and converges to around 300 eventually. However,

neither of the methods outperforms the ADAM optimizer. This is because the step length is problem-specific and gives a boost to the optimization process.

3.2.4 Limitations

Although RNN is convenient to take advantage of parallel CPUs and GPU acceleration, it also has many drawbacks. On the one hand, RNN uses small number of time steps (usually less than 100), because of the kind of problems it was designed for (voice recognition and word processing). On the other hand, it needs to use small time steps to avoid aliasing and distortion and to fully cover realistic models. TensorFlow saves all the functions to cache for each layer before starting the back-propagation, but in the kind of problems discussed in this chapter there are too many variables. For the scatter model as shown in Figure 3.2, though the actual training progress is fast, it takes $10\,\mathrm{gigabytes}$ of memory and usually needs $15\,\mathrm{min}$ to $20\,\mathrm{min}$ just for the initial setup. This disadvantage may be avoided by designing a more suitable neural network structure or by transforming the time domain into the frequency domain to reduce the number of layers in RNN.

The other notable drawback is from the use of the Born approximation. Like LSRTM, that also uses the Born approximation, the method also highly depends on the quality of the background velocity estimation. A poor/wrong background velocity will result in position shifts for the imaged reflector. A solution for this would be adding more non-linearity. For example, the forward modelling can be replaced by the finite difference method, or we can update the background velocity during the iteration to add more non-linearity.

3.3 Conclusion

The Born modelling can be successfully implemented using RNN with TensorFlow. Then, by feeding a theoretical data to the RNN built, the model can be inverted by back-propagation of the RNN. This operation can be proven to be the same as the LSRTM formulation. We

found the ADAM method seems to be the most efficient optimizer but it requires to be extra careful when choosing the hyper-parameters. The second efficient optimizer is L-BFGS-B, which does not take extra hyper-parameters. The least efficient optimizer in our tests is the FR-CG, which spends much time in line searching and hence causes too many perturbations to the loss curve. The overall computing performance is good but TensorFlow takes too much time and memory to build the network before the back-propagation. In the future, we are interested in bringing this method to the frequency domain and looking for a more suitable neural network structure for wave propagation.

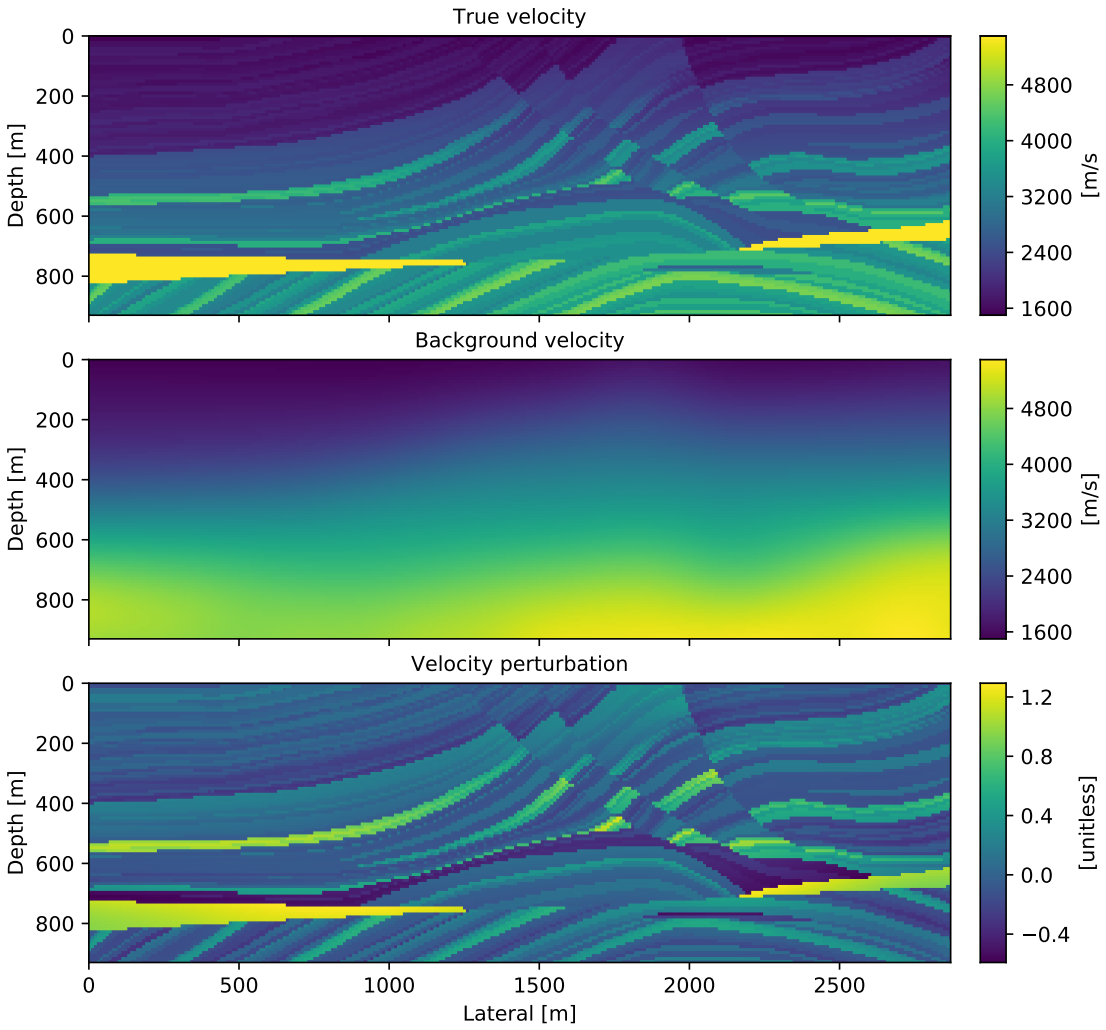

Figure 3.7: The Marmousi model. The background velocity model is obtained by gaussian filtering of the true model. The velocity perturbation is $2\delta \boldsymbol{v}/\boldsymbol{v}_0$, as defined in the theory section.

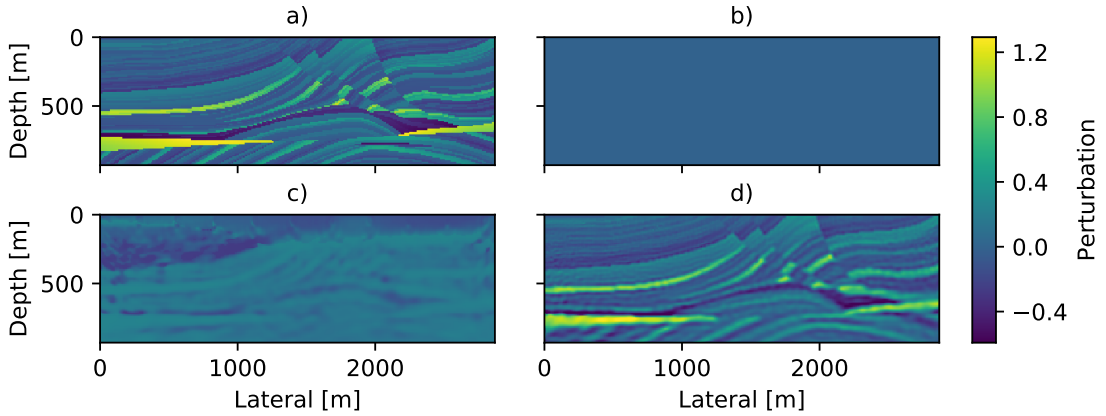

Figure 3.8: The inversion results of Marmousi model by RNN. a) The true model; b) The initial zero model; c) The estimated model at the 10th iteration with ADAM optimizer using learning rate 0.3; d) The model at the 50th iteration.

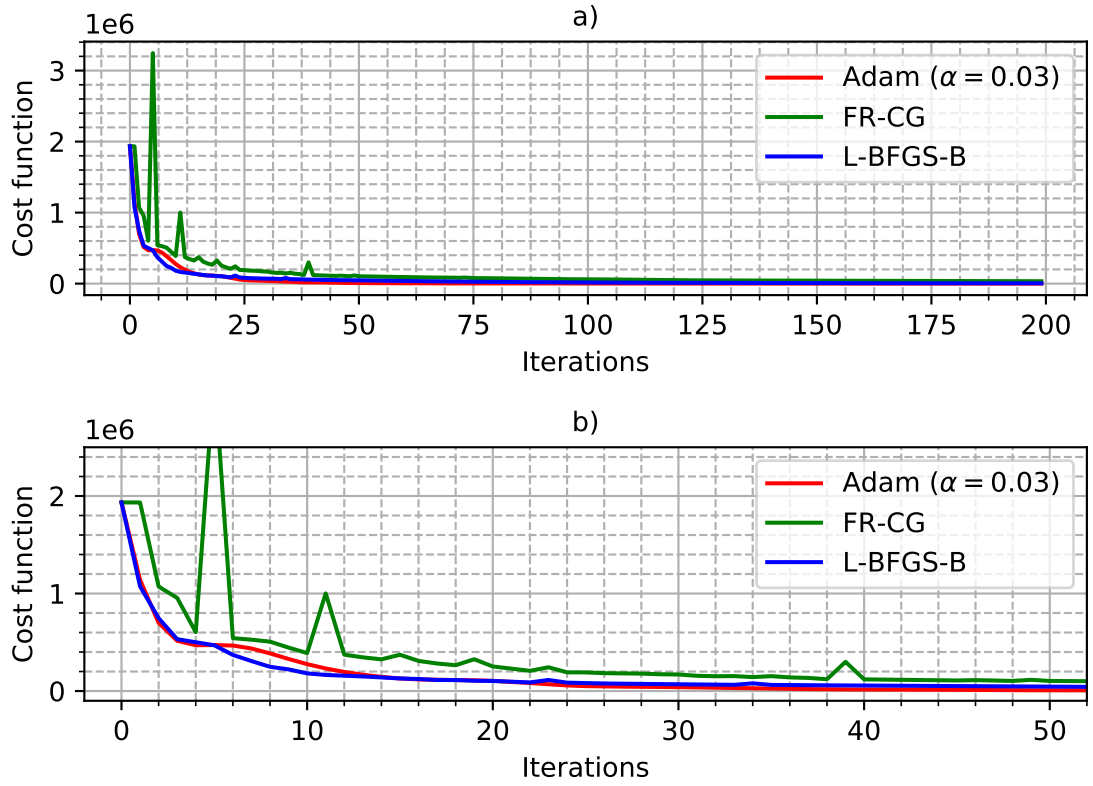

Figure 3.9: Cost function curves comparison of the three used methods. a) The first 200 times of loss calculations; b) a zoomed version of the figure a.

Chapter 4

Velocity extraction from migration images

4.1 Introduction

Machine learning has become an important tool in the field of geophysics. It can overcome some drawbacks of the theory and facilitates the combination of CPUs and GPUs because of the maturity of high-level APIs. In the previous chapters, we discussed that machine learning has great potential in numerical analysis, but it also has limitations. The most crucial limitation is generality. Since machine learning is a data-driven technique, the learned knowledge can be specific to the dataset used for training. Therefore, the predictions from a trained network can often be invalid for other data, for example in seismic, data observed in a different type of geological environment or survey conditions. Especially in seismic inversion and forward modelling, a small change in the acquisition can result in great changes in the data. Since there are many possible types of surveys, training a network can be very challenging, because a dataset cannot cover countless acquisition setups. Most researchers on solving modelling/inversion problems often assume a fixed acquisition geometry and train networks with data based on random velocity models. This

generalizes the network for a variety of velocity models but not for different acquisition geometries.

In this chapter, we will explore a way to extract velocity from reverse time migration images. We use migration images because that helps to remove dependencies on the acquisition geometry. Shot gathers are a function of time, acquisition parameters and the geology that lies beneath. Not all of them are of our interests — we care more about the structure and the rock properties. Migration, however, reduces the number of parameters by mapping seismic events to reflector boundaries, which are of interests for seismic interpretation. Reverse time migration (Claerbout, 1971) uses the correlation of source wavefields and receiver wavefields to show the location of the reflector. This method requires a good background velocity model and does not deal with multiples (which is related to acquisition geometry), but it has become very popular because of its good balance between accuracy and efficiency. Different imaging conditions have been proposed, and they all aim to achieve better accuracy for estimating the reflectivity, from which we can extract more information about the subsurface. It is crucial to recover the correct reflectors' amplitude, and it is beneficial for model updates in inversion methods such as FWI (Lailly & Bednar, 1983; Tarantola, 1984). Starting from the inaccurate amplitude from migration images, we trained a neural network to convert them into velocities at low costs.

4.2 Theory

4.2.1 Reverse time migration

Reverse time migration (RTM) uses an imaging condition to combine the shot and receiver wavefields and obtain the location of the reflectors. The simplest form of this condition is the dot product of the up-going wavefield and down-going wavefield. Following Claerbout, 1971, the reverse propagation of the shot record will coincide with the source wavefield at the position of the reflector, providing a maximum of the cross-correlation where the

reflector is located. A typical 2D cross-correlation imaging condition can be defined as

$$I(\boldsymbol{x}) = \sum_t S(\boldsymbol{x}, t) R(\boldsymbol{x}, T - t) \tag{4.1a}$$

$$I'(\boldsymbol{x}) = \frac{2}{v_0(\boldsymbol{x})^3} \sum_t \frac{\partial^2 S(\boldsymbol{x}, t)}{\partial t^2} R(\boldsymbol{x}, T - t) \tag{4.1b}$$

where S refers to the source wavefield while R refers to the receiver wavefield. The dot in between refers to element-wise production. The indexes t and $T - t$ indicate the forward and reverse propagation of the wavefields. Each time t produces an image. When all these images are added, constructive interference produces a stable image of the reflectors. Equation 4.1a does not yield the true reflectivity but is a good approximation. In this chapter, we define the imaging condition by Equation 4.1b as it is closer to the FWI gradient. The source wavefield is replaced by its second derivative with respect to time, and the amplitude is normalized by the background velocity model $v_o(\boldsymbol{x})$. This replacement corresponds to using Born modelling instead of the full wavefield, as is common in least-squares migration since it casts imaging as a linear problem.

4.2.2 ℓ_1 norm and ℓ_2 norm

ℓ_1 norm and ℓ_2 norm (or their squares) are the most common loss functions used in optimization problems. The choice between them lead to different results in inversion problems (Taylor et al., 1979). Suppose the observation/true data is $\boldsymbol{y}$ and prediction is $\boldsymbol{y}_{\text{pred}}$, then

$$L_1(\boldsymbol{y}, \boldsymbol{y}_{\text{pred}}) = \frac{1}{n} \sum_i^n \left| y^{(i)} - y_{\text{pred}}^{(i)} \right| \tag{4.2a}$$

$$L_2^2(\boldsymbol{y}, \boldsymbol{y}_{\text{pred}}) = \frac{1}{n} \sum_i^n \left(y^{(i)} - y_{\text{pred}}^{(i)} \right)^2 \tag{4.2b}$$

Equations 4.2a and 4.2b are both metrics that positively relate to the error magnitude and will be zero if there is no difference between $\boldsymbol{y}$ and $\boldsymbol{y}_{\text{pred}}$. Note that Equation 4.2b is ℓ_2 norm square to be precise, but it is called ℓ_2 in this chapter for simplicity. Another interesting aspect appears when we consider their gradients. Let $\nabla_{\boldsymbol{y}} L$ denote the gradient of L with respect to $\boldsymbol{y}$, then

$$\nabla_{\boldsymbol{y}} L_1 = \frac{1}{n} \cdot \text{sign}\left(\boldsymbol{y} - \boldsymbol{y}_{\text{pred}}\right) \tag{4.3a}$$

$$\nabla_{\boldsymbol{y}} L_2^2 = \frac{2}{n} \left(\boldsymbol{y} - \boldsymbol{y}_{\text{pred}}\right) \tag{4.3b}$$

We can see that the gradient of ℓ_2 is related to the error at that point while the gradient of ℓ_1 is essentially a direction which could only be either 0, 1 or -1. These characteristics can be both advantages of disadvantages. Optimizations based on ℓ_2 gradient produce updates proportional to the error, which will be strong when the error is large but weaken as the solution gets closer to the truth. On the other hand, optimizations based on ℓ_1 have a more stable model update but will oscillate when the learning rate is bigger than the error.

Another way of understanding the difference between ℓ_1 and ℓ_2 losses is the priority when dealing with outliers, that is data points with large observational error. From Equation 4.2b we can see that ℓ_2 weights large errors much more than small errors since the error is squared and the gradient scales with the distance. Using an ℓ_2 norm as a loss function will focus more on solving these large errors first, which can be difficult if the initialization is bad. On the other hand, ℓ_1 norm weights large and small errors in a more similar manner.

4.2.3 Chain rule and back-propagation

The chain rule from calculus can be summarized with the following equation. Assuming $L = f(\boldsymbol{y})$ where $\boldsymbol{y} = g(\boldsymbol{x})$, then

$$\nabla_{\boldsymbol{x}} L = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\intercal} \nabla_{\boldsymbol{y}} L \tag{4.4}$$

L is a scaler function that could represent the loss while $\boldsymbol{x}$ and $\boldsymbol{y}$ are two vectors that do not necessarily have the same dimension. ∇ refers to the gradient with respect to the subscript. We can see that the gradient with respect to x, $\partial \boldsymbol{y}/\partial \boldsymbol{x}$, refers to the Jacobian matrix, which links the two gradients. One can calculate the gradient with respect to later variables ($\boldsymbol{y}$), then convert it to the gradient of previous variables ($\boldsymbol{x}$). That is why this scheme is named "back-propagation". The back-propagation starts from $\partial J/\partial J = 1$ and then recursively multiplies the Jacobian matrices down to each trainable parameter, which yields the gradient for parameter updates. Then the gradient will keep propagating until it reaches the beginning of the computation graph.

One can infer that if the chain becomes too long and any of the gradients in this chain has become a small number, the resulting gradient will end up with an even smaller number and make the parameters difficult to update. This phenomenon is called vanishing gradient in numerical differentiation. There are several ways to mitigate gradient vanishing. The first cure is to avoid too deep networks. This is usually not the case because more layers are needed for adding enough non-linearity. Another popular method is to add shortcuts to the network, which allows the neural network to learn some features first and then deal with the other. U-Net (Ronneberger et al., 2015) and ResNet (He et al., 2016) are two of the most popular frameworks that use shortcuts.

4.2.4 Residual network (ResNet)

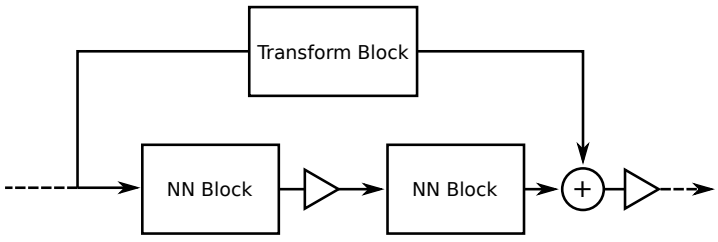

Figure 4.1: A ResNet building block modified from He et al., 2016. Regular triangles refer to activation functions. Dashed arrows are connected to other blocks.

Figure 4.1 shows a building block of a typical ResNet. The building block contains two parts: the backbone which contains the main structure of the neural networks, and a shortcut to skip neural network blocks. Each of the neural blocks can contain several neural layers which can be either convolutional or fully connected. The shortcut contains a transformation that will fix the dimensional mismatch of its inputs and outputs. Usually, the transformation is linear with no activation function applied, or it can also be the identity operator if the input and output are in the same dimensions. Then the result of the transformation is added back to the backbone's output, and the sum is then fed to an activation function. The dashed line on either side may be connected to other ResNet building blocks.

The introduction of shortcuts enables the neural network to skip unnecessary steps. Since there are fewer terms when applying the chain rule, this process prevents vanishing gradient to some degree. Furthermore, the shortcuts do more than skipping some layers because of the existence of the addition node. This could be understood in another way. Suppose that the transformation block is identity. Instead of fitting a function that maps from the block input $\boldsymbol{x}$ to the block output $\boldsymbol{y}$, the ResNet block is trying to fit a function that maps from $\boldsymbol{x}$ to $(\boldsymbol{y} - \boldsymbol{x})$. In other words, the ResNet is forced to focus on learning features that are non-linear to $\boldsymbol{x}$.

4.3 Example

4.3.1 Images from RTM

In this example we will use a neural network to convert reflectivity traces from 1D RTM migration to velocities. The velocity models are defined as 4-layer 1D models. Each generated velocity model contains 1000 points with spacings of $8\,\mathrm{m}$. We fixed the shallowest layer to have a velocity of $3000\,\mathrm{m/s}$ and assume it is the minimum of the entire model. The other layers have random velocities within $3000\,\mathrm{m/s}$ to $5500\,\mathrm{m/s}$. The range was designed

to match the global range for the acoustic Marmousi model. The positions of reflectors are randomized with uniform distribution. However, we reject the models with layers being too thin to avoid severe overlapping of primaries. There is no need to apply such restriction to the deeper layers, since velocity model can be treated as having fewer layers if any adjacent velocities are close.

The RTM images are generated by following Equation 4.1b. The forward modelling used second order finite difference method for calculating both temporal and spatial derivatives. The source wavelet is a Gaussian source. The source is non-negative and symmetric, which helps to identify reflectors. The source and receiver are both placed at $8\,\mathrm{m}$ below the surface. We use absorbing boundary conditions on both sides of the wavefield, and direct waves are removed when calculating the receiver wavefield.

4.3.2 The definition of input/output

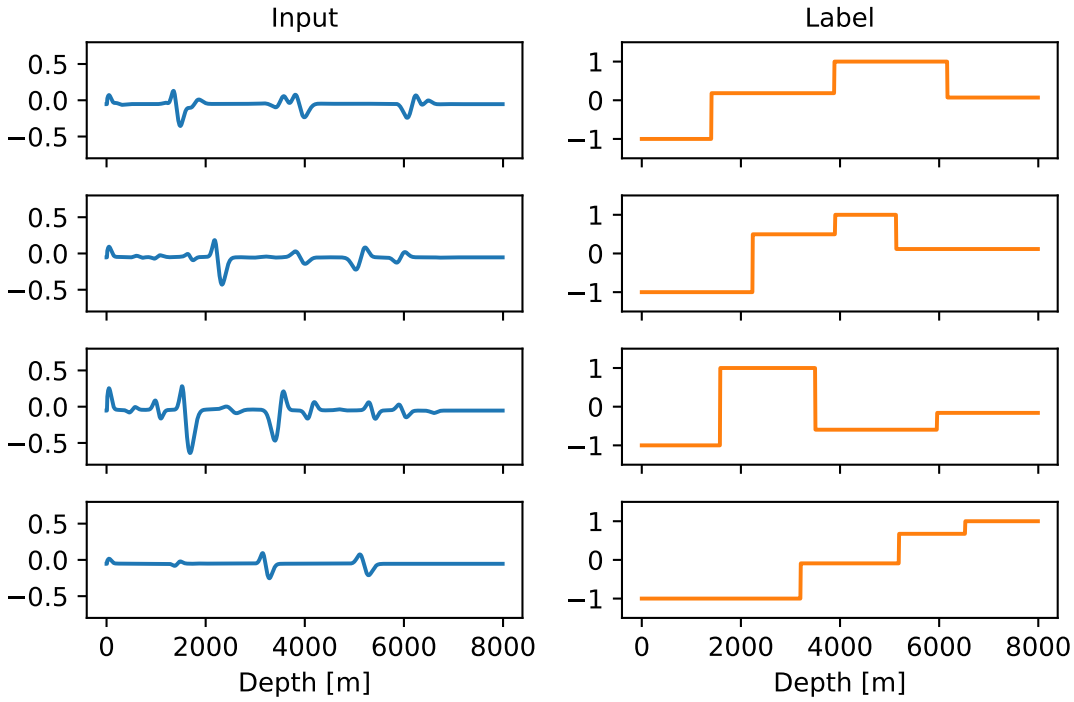

Figure 4.2: Four random examples of input and label pairs.

Here we define the input fed to the neural networks to be $50\,000$ RTM images from random 1D velocity models as described in the above section. In machine learning, the theoretical values used to calculate the loss of predictions are called true labels. In this case, the true labels are the velocity model corresponding to each image. There is an infinite number of choices in the forms of inputs and labels, which will affect the focus of the network. For example, the labels can be vectors that store the depth and velocity of each layer since we have flat velocity models. The representation is efficient in terms of telling information, and the model would spend no effort in learning each layer has constant velocity. The problem is that this representation makes the problem hard to generalize because we have to know the number of layers in advance and train models for different situations. Another representation is to use the true reflectivity directly as labels. This may also be problematic since the entire information is concentrated on the "spikes" in the reflectivity. Useful information will be flooded by less significant information and make the network less intuitive to what type of information should be learned.

Different input-label pairs are called *samples* in machine learning. Both input and labels have the dimensions of $n_{\text{sample}} \times n_{\text{z}}$, which is $50\,000$ by 1000 in this case. Figure 4.2 shows 4 example input-label pairs from the dataset. The left column refers to the inputs, which are normalized to be ranging from $(-1, 1)$. The right column shows the labels, which are the corresponding velocity models. The magnitudes have been normalized to the same range. Note that the "velocities" of the first layer are all -1. This is because they are equal to the minimum of the entire set of velocity models, i.e. $3000\,\mathrm{m/s}$. For the same reason, the maximums of the labels are all equal to 1. The entire dataset is then separated randomly into training and validation sets. The training set takes up $80\,\%$ from the whole, and the rest forms the validation set. The training set contains the data used directly for calculating the gradients at each iteration. The gradients would be directly related to the misfit between the predictions of the model and labels in the training set. On the other hand, the validation set is used indirectly as a metric of the optimization process. We used the performance on

the validation set to help determine hyper-parameters, the degrees of over-fitting or the timing for early stopping. The trained network is then still a function of the validation set and this is how a validation set differs from the test set. In an ideal case, the dataset should be divided into training/validation/test sets. The test set, which never gets involved in the training process from the start to finish, is the only reliable metric for judging a model. However, we ignore the difference and use validation error to estimate test error in this case. This may be unfair in some sense but will allow us to have more data reserved for the training set. The Adaptive moment estimation optimizer (ADAM, Kingma & Ba, 2014) is adopted. Despite different neural network types, the workflow can follow Algorithm 3. The algorithm saves the model that has the lowest validation error in a training process. More details on the workflow are discussed in Chapter 3.

4.3.3 Fully connected neural network

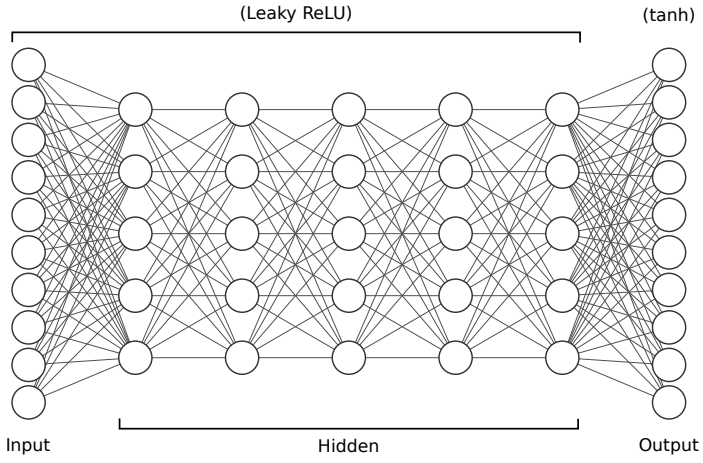

Figure 4.3: A 7-layer fully connected neural network. Each circle represents 100 nodes.

All neural networks in this chapter are implemented with the machine learning package PyTorch (Paszke et al., 2019). A fully connected model is used for solving the original problem (see Figure 4.3). The model contains seven layers. The input layer and output

layer have 1000 nodes, the same as the length of the model. There are five hidden layers in between, with 500 nodes in each layer. The activation function for each hidden layer is leaky ReLU (leaky rectified linear unit) with a slope of 0.2 on the negative side. The leaky ReLu will capture features on the negative half but still add non-linearity. The activation function for the outputting layer is $\tanh$. The connections between two adjacent layers in a fully connected network can be represented by a matrix containing trainable parameters. In this network structure, the trainable parameters are, 1000 by 500 for the connections between the inputs and first layer; four 500 by 500 matrices for interconnection between hidden layers; and a 500 by 1000 between the last hidden layer and the output layer. In addition, there is one extra parameter for the bias of each neuron. The total number of trainable parameters is quite large, about 2×10^6. Although the number of parameters is smaller than the number of data points, the problem is under-determined because data points are not fully independent of each other. This is typical in Neural Networks, which are designed to find non-linear patterns from data and therefore given more flexibility than formulations based on physical laws.

4.3.4 Choosing the right loss function

Figure 4.4 shows predictions from different models trained with ℓ_1 and ℓ_2 loss. The bottom figure refers to the inputting RTM image and the corresponding velocity model is shown as the blue line in the top figure. We used the same network structure and hyper-parameters for both tests. We can see that the ℓ_1 prediction (green) is visually better than the ℓ_2 prediction (orange). Especially, ℓ_1 loss reacts faster when there is an abrupt change in the label, which eventually helps to update other velocities. We can also notice the ℓ_2 prediction is affected by another layer at around $1000\,\mathrm{m}$ to $1500\,\mathrm{m}$. Similar observations can be made in the last layer ($6000\,\mathrm{m}$ to $6500\,\mathrm{m}$), where the prediction is affected by greater velocities above it. One interesting part is that although ℓ_1 is overall more stable than ℓ_2, they have similar behaviours handling different layers. We define the model to have a fixed min and

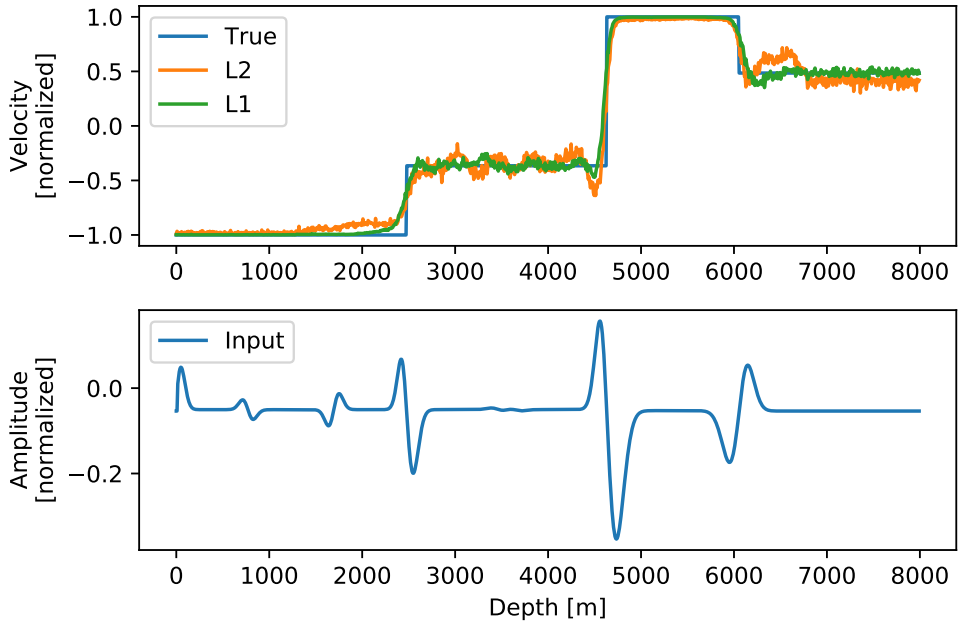

Figure 4.4: Predictions made by models with ℓ_1 and ℓ_2 loss function, respectively.

max velocity, which are -1 and 1 after normalization. This is because we use $\tanh$ as the outputting activation, which suppresses all prediction that is too large or too small. The raw output before the $\tanh$ activation function may have great oscillations and more drastic limits.

Figure 4.5 shows the comparison between ℓ_1 and ℓ_2 loss on the training set. The ℓ_2 loss in Equation 4.2b is square rooted to be comparable with ℓ_1 loss. The figure shows that the optimization curve with ℓ_1 not only converges faster but also achieves lower error in the late stage. Since the label contains abrupt changes, ℓ_2 loss focuses on dealing with those changes from the beginning but gets confused as iterations proceed and causes fluctuations eventually. On the other hand, ℓ_1 loss is less affected by this issue.

Although the ℓ_1 loss curve shows that the network probably needs more iterations or a larger learning rate to reach a plateau, it proves that ℓ_1 is a more suitable loss function for this type of problem. Therefore we adopt ℓ_1 norm as loss function. However, there may be better choices of the loss function, like total variation (Anagaw & Sacchi, 2012) which is

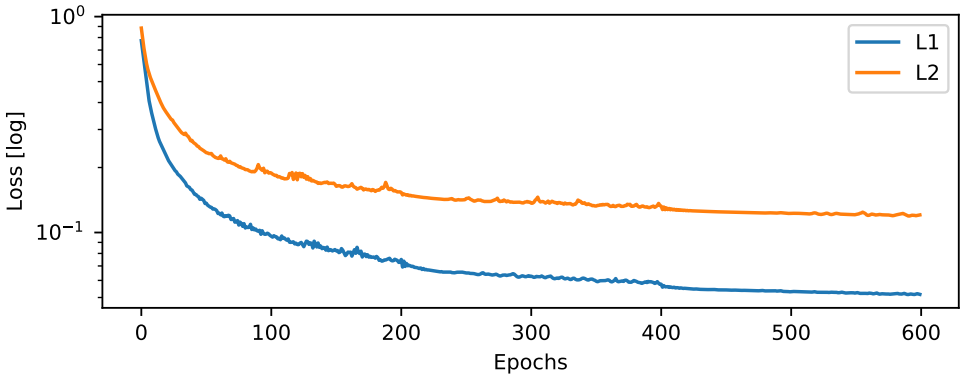

Figure 4.5: ℓ_1 and ℓ_2 loss comparison.

better in preserving blocky discontinuities rather than spiky models.

4.3.5 ResNet

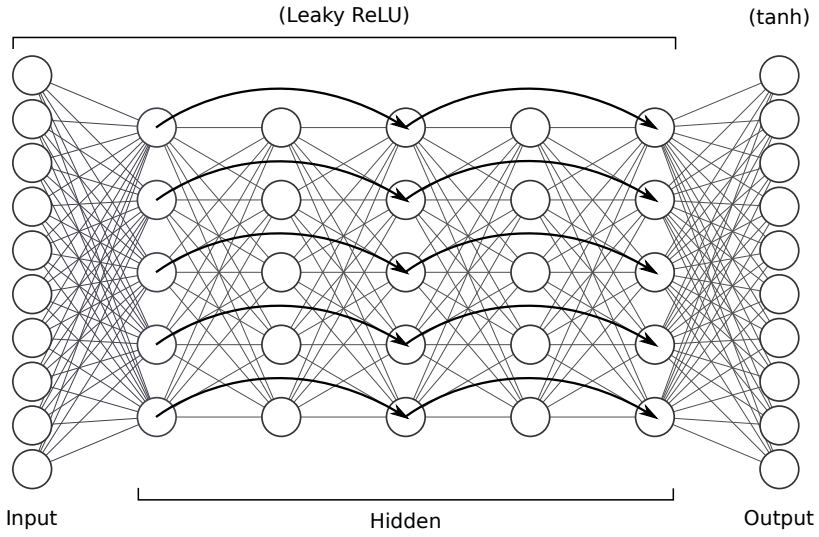

Figure 4.6: A ResNet based on the fully connected network in Figure 4.3.

Figure 4.6 shows the ResNet tested in this chapter. In addition to the network in Figure 4.3, skipping connections are added to the hidden layers. As shown by the black arrows, each connection skips two trainable fully connected layers and adds the input

directly to the output of the ResNet building block. Since the input and output of each building block are the same, the transformation block is the identity operator. There are no additional trainable parameters introduced. Therefore the networks should have comparable training burdens.

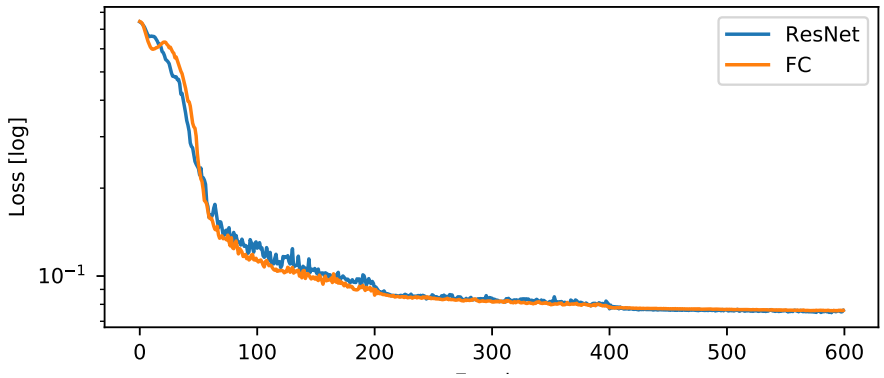

Figure 4.7: Loss curves of the fully connected (FC) and the ResNet model.

We train the ResNet with the same setup and hyper-parameters as before. The loss curve on the validation set is very similar to and almost overlapping the loss curve of the fully connected case (see Figure 4.7). Both models achieve small ℓ_1 errors and do a good job of identifying reflection interfaces. However, the two models make predictions differently.

As shown in Figure 4.8, although the two networks have similar errors, the ResNet predictions (green in the top figure) have fewer fluctuations than the fully connected (FC) predictions (orange in the top figure). This characteristic is common in different samples. The only difference between the ResNet and the fully connected models is the shortcuts that fed back to the backbone. The direct input from several layers before makes the network easier to find relationships between points and hence reduces the fluctuations. However, the shortcuts do not help much on the convergence in this case. This is because we are using a relatively shallow network that may not suffer much from vanishing gradient.

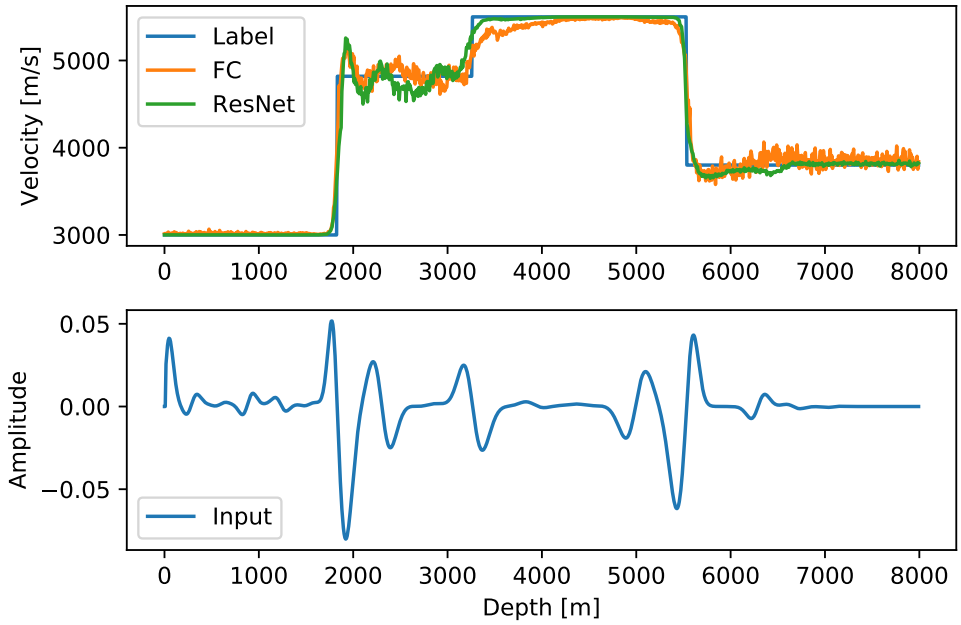

Figure 4.8: A comparison between predictions from the fully connected (FC) model and the ResNet.

4.3.6 Problematic cases

For testing, one thing to keep in mind is that the validation/test dataset must be normalized in exactly the same way as the training dataset. In this chapter, the training data are normalized by linearly stretching the min and max value to $(-1, 1)$ and the test dataset must be stretched with the min and max of the training dataset but not its own.

As shown in Figure 4.9, we test the model with inputs and output pairs generated by using a different wavelet (Ricker wavelet). Although we can easily recognize the positions of the reflectors on the bottom image, the predictions are bad for both models. The predictions somewhat react to reflections at the first two interfaces but failed to detect the deepest interface. Also, the velocity is correct only for the first 150 points and this is partly because the velocity of the first layer is fixed. Similar results can be observed if we change the imaging condition. This is certain because we have broken the fundamental rule of machine learning: the test set must come from the same distribution for the training set. In order

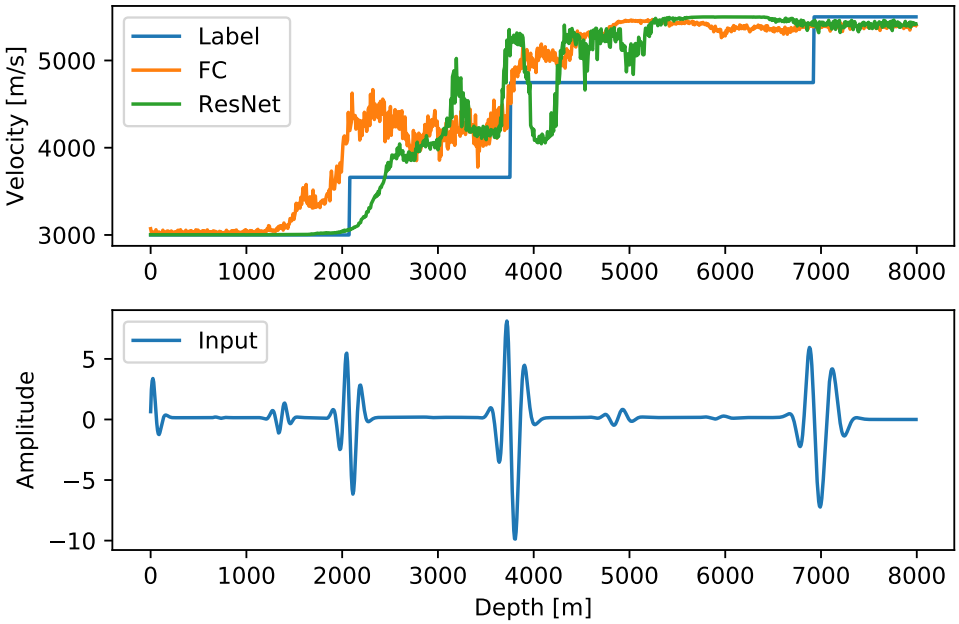

Figure 4.9: A typical prediction on data with Ricker wavelet.

to make the model work with different wavelets, we should either remove the wavelet effect by some methods (as we remove dependencies on acquisitions by migrations) or provide enough data for the network to learn about the change. The former will make the whole problem less meaningful as if the wavelet effect is fully removed, the results will be the true reflectivity and we can get velocity by integration. The latter would require the dataset to be several times larger and perhaps need a network with more complex structure, more trainable parameters and more advanced technique for the training (such as gradient boosting, which takes a lot more power to perform). This is hard to do, either more data is not available or computation cost is too high even for today's computation power.

4.4 Conclusion

In this chapter, we use fully connected networks to recover the reflectivity from random 4-layer velocity models. We investigate different behaviours when using ℓ_1/ℓ_2 norms as

loss function, and we conclude ℓ_1 is more suitable for this type of problem. We test ResNet shortcuts to the network and they reduce fluctuations. The model performs poorly on data from different distributions of the training set. Future works may include applying more advanced training techniques like gradient boosting or seeking better representations of the input and outputs. Also, we may test the effects of total variation as an additional regularisation to the loss function.

Chapter 5

Constructing seismic using generative adversarial network

5.1 Introduction

Machine learning has become a popular topic in most sciences and geophysical applications are not an exception. In Geophysics, many successful applications of supervised machine learning have been published, in particular in the area of image segmentation/detection, for example, facies recognition, salt body segmentation (Lomask et al., 2007; Shi et al., 2018), relative geological time picking (Bi et al., 2020), etc.

Although applications have become more robust in the image recognition field (Chen et al., 2017; Girshick, 2015; Girshick et al., 2014), we still face a significant challenge that does not exist in the broader machine learning society: the lack of abundant public labelled data. The abundance of data is perhaps more crucial and needed to solve geophysical problems than in other areas like image classification because seismic data interpretation, for example, relies on subtle details with complex relations between physics and geology. Therefore, in order to solve problems involving a complex theory by using machine learning, geophysical research injects theoretical knowledge through the use of complex network architectures

or applies physics-guided regularization to compensate for the gaps in information during the learning process. As a partial solution, researchers have tried to generate synthetic data with satisfying quality for use in training and improve model convergence and model generality. For example, Wu et al., 2019 successfully trained a relatively ordinary U-Net with synthetic fault images. The images are very carefully generated, so the trained model can provide accurate results on real data and easily adapt to other scenarios without harm in accuracy using transfer learning. However, we do not always have abundant or precise knowledge to model the data. Generative adversarial networks (GANs, I. J. Goodfellow et al., 2014) is capable of this kind of tasks. A successfully trained generator can produce artificial data in a given data distribution.

This chapter is our first attempt to test GANs in synthetic data modelling. We will use very simple cases to generate seismic data. At this stage, the main goal is to understand the characteristics of GAN and its behaviour during training and opening the door for further research. We explore the methodology of generating 1D data with a generative adversarial model. Both the generator and discriminator are convolutional, and the noise vectors are fed along the channel dimension to the generator. The networks are successfully trained via Wasserstein loss with gradient penalty and careful hypermeter tuning. We evaluate the trained networks quantitatively and qualitatively. We attempt to find the optimal stopping point for the training. However, the conclusion cannot be made during the training and part of it remains subjective.

5.2 Theory

5.2.1 Generative adversarial network

Figure 5.1 shows a typical structure of a GAN. There are two sub-networks in a GAN, the generator (G) and the discriminator (D is sometimes called the critic, depending on the form of the output). G generates some samples in the form of random vectors $\boldsymbol{n}$, while

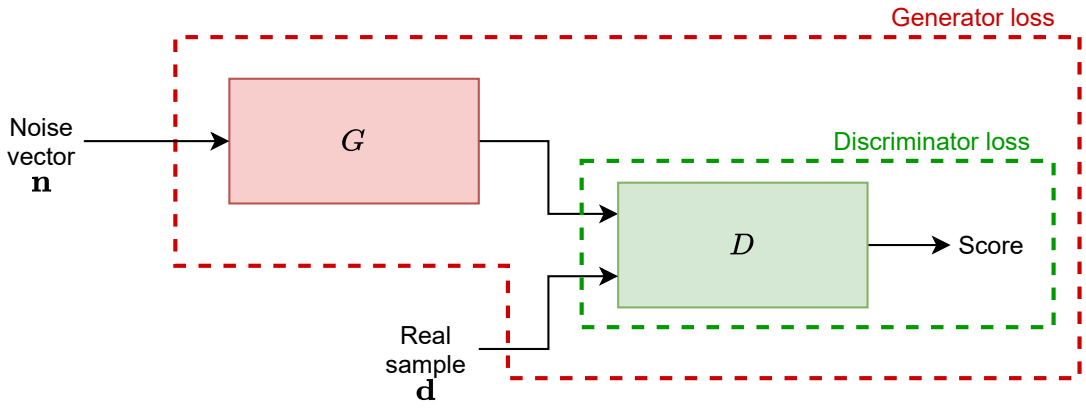

Figure 5.1: A typical structure of unconditional GAN

D distinguishes the generated samples from real samples $\boldsymbol{d}$. The two networks compete against each other and try to improve themselves during the training. G will try to learn how to trick D by generating more realistic predictions, while D will try to become more aware of the differences between the generated and real samples, which will be changing as G improves.

The main goal of a GAN is to find a transform from a randomly distributed variable to a given data distribution. The input of a GAN is usually a uniformly randomized noise vector $\boldsymbol{n}$. The output is not treated as the "ground truth" as we do in supervised learning because no individual input is trained to be bonded to a corresponding label. Instead the distribution of the input labels and the predictions are compared to each other to assess convergence.

GANs are notoriously famous for their difficulty in being properly trained. There are two critical aspects of designing a GAN. First, the accuracy of the network mostly depends on the robustness of the discriminator due to the way the value function is defined. The discriminator must be capable of doing its job of separating truth from fake, while being trained with general approaches. Second, the success is based on balancing the training of the generator and the discriminator. If the discriminator is too strong or learns too fast compared to the generator, it cannot provide useful feedback for the generator to continue

the learning. In this case, the discriminator will always reject the model no matter how the generator modifies its parameters, so the generator is likely to get trapped in local minima and fail to escape from it because of unhelpful gradients. On the other hand, if we have a generator that is much superior to the discriminator, the generated example will always fool the discriminator. Because the "same thing" is labelled to be both right and wrong from the perspective of the discriminator, there may not be a clear path to improvement and be unable to adapt its weights to decrease the loss function. Using a personification of the discriminator, we could say that it may get confused and refuse to improve itself.

Because of the reasons above, training a GAN is where science forgets its modern role and becomes alchemy (Chollet, 2018). Many empirical tricks need to be applied to the model, and they may not be suitable in other cases.

5.2.2 Wasserstein GAN with gradient penalty

Here we use the value function from WGAN (Arjovsky et al., 2017) for more stable training. The value function is defined as

$$V_W = \min_G \max_D \mathbb{E}[D(\boldsymbol{d})] - \mathbb{E}\left[D(G(\boldsymbol{n}))\right], \tag{5.1}$$

where $\boldsymbol{n}$ refers to the random latent vector and $\boldsymbol{d}$ refers to real data. In practice, the expectations are replaced by the mean value of the current mini-batch. Equation 5.1 defines a min-max game, in which we want to find a $D(\cdot)$ that maximizes its expected score on real examples while minimizes its expected score on the generated ones. Meanwhile, we find a $G(\cdot)$ that maximizes its expected score from the discriminator. Gulrajani et al., 2017 propose a gradient penalty as a regularization term in addition to Equation 5.1 to enforce 1-Lipschitz constraint. The term is defined as

$$V_P = \left(\left\| \frac{\partial D(\boldsymbol{m})}{\partial \boldsymbol{m}} \right\|_2 - 1 \right)^2, \text{where } \boldsymbol{m} = \epsilon \boldsymbol{d} + (1-\epsilon) G(\boldsymbol{n}). \tag{5.2}$$

Here $\boldsymbol{m}$ refers to the mixing of real and generated samples, which is controlled by a random scalar ratio ϵ. The value ϵ is drawn from $U(0, 1)$ at each discriminator update to lower the chance of being stuck in some local minima by introducing more stochasticity. The partial derivative can be calculated with auto differentiation. Minimizing Equation 5.2 will favour the discriminator gradients with a unitary norm, therefore clipping larger unstable gradients and guiding the model to avoid small updates.

By combining Equation 5.1 and 5.2, we obtain the following loss functions required for updating the generator and discriminator parameters:

$$L_G = -\mathbb{E}\left[D(G(\boldsymbol{n}))\right], \tag{5.3}$$

$$L_D = \mathbb{E}\left[D(G(\boldsymbol{n}))\right] - \mathbb{E}[D(\boldsymbol{d})] + \lambda V_P. \tag{5.4}$$

Note the negative sign in L_G since the two losses are opposing each other. The min-max problem then becomes two optimization problems where the two networks are updated according to the losses in an alternating fashion. The trainable parameters in the discriminator are temporarily frozen when updating the generator using Equation 5.3, and the parameters in the generator are frozen when updating with Equation 5.4.

5.3 Method

5.3.1 Architecture

Since the problem is relatively straightforward, we designed two small networks from scratch. We use PyTorch (Paszke et al., 2019) as the machine learning framework. Both the generator and the discriminator are constructed using sequential 1D convolutional layers. The noise vectors $\boldsymbol{n}$ with a length of 100 are fed into the generator via the channel dimension with a size of 1 in the spatial dimension, which is gradually increased by undergoing a sequence of 1D transposed convolutional layers with proper kernel sizes and strides. The

first layer in the sequence has 256 filters. The number of filters is halved multiple times in the successive layers. At the last layer, the number of channels is reduced to 1, and the spatial dimension is expanded to 499 to match the length of traces $\boldsymbol{d}$ from the forward modelling.

Table 5.1 shows the details on the output dimension after each transposed convolutional layer in the generator. Without zero-padding and dilation, the output dimension of the ith layer can be calculated as

$$l_i = l_{i-1}s_i + k_i \tag{5.5}$$

where s_i and k_i refers to the stride and kernel size of the ith layer, respectively.

Table 5.1: The detailed structure of the generator

layer	channel	length	kernel size	stride	# filters
1	100	1	3	2	256
2	256	3	4	1	256
3	256	6	4	2	128
4	128	14	4	2	128
5	128	30	3	2	32
6	32	61	4	2	32
7	32	124	3	2	16
8	16	249	3	2	1
output	1	499			

The discriminator is fully convolutional. It takes input that has one channel with a length of 499 and makes it through four convolutional layers with a kernel size of 3. Then the length and channel dimensions of the output are switched. Finally, the output goes

through two 1×1 convolutional layer to be packed to a scaler score for each sample. The details are summarized in Table 5.2.

Table 5.2: The detailed structure of the discriminator

layer	channel	length	kernel size	# filters
1	1	499	3	64
2	64	499	3	64
3	64	499	3	64
4	64	499	3	1
5	499	1	1	250
6	250	1	1	1
output	1	1		

We use leaky ReLU (Xu et al., 2015) instead of ReLU as inter-layer activation function in both generator and discriminator. We also introduce batch normalizations before each convolutional layers in the generator only, since the discriminator remains more stable during training compared to the generator.

5.4 The dataset

The data is generated by 1D forward modelling with the direct arrival removed by subtraction. We use simple velocity models with four horizontal layers with random interval velocity and thickness. As a source wavelet, we use a Gaussian function at shallow locations. The model has a free-surface boundary condition and absorbing boundary at depth. Since the source position is shallow, the primary wave overlaps with the ghost wave from the surface boundary and forms a unique waveform (see orange traces in Figure 5.2) $10\,000$ traces

are generated in total to ensure continuous distribution. Each trace has 2000 timesteps and is later resampled and trimmed to 499 to make the generator training-friendly.

The data are divided by ten folds of the global mean for normalization. No bias is removed from the data to avoid shifting the origin. The magic number 10 was obtained empirically by experiments. This number is bounded to the initialization of trainable parameters in both networks. Three traces after normalization are shown in Figure 5.2 as orange lines.

5.5 Training details and workflow

Since the GAN consists of two networks, we have to define two separate optimizers, one for each of them. Both networks use an Adam optimizer (Kingma & Ba, 2014) with a learning rate of 1×10^{-4} and a $\beta_1 = 0.5$ lower than the default value. This ensures that the model updated is more influenced by the current gradient than by the momentum part. Based on experiments, it is crucial to use additional methods to stabilize the training since the value function mentioned in the previous section will react more wildly than a common loss function like binary cross-entropy or mean square error. The convolutional kernels in both networks are initialized with a standard deviation of 0.2, which is smaller than PyTorch's default, to avoid huge predictions at early stages. The λ in Equation 5.4 is set to 10.

We train the GAN for 300 epochs. We load the data with a batch size of 64 on each $16\,\mathrm{GB}$ graphic card. Moreover, we trained the generator once but the discriminator twice at each iteration to balance the power of the two networks during training. The training workflow is shown as Algorithm 6.

Algorithm 6 Training workflow for GAN.

Require: $G(\cdot)$, $D(\cdot)$, $\boldsymbol{d}$
  for each epoch **do**
    for each mini-batch **do**
      for counting 2 **do**
        generate noise vectors $\boldsymbol{n}$
        $\hat{\boldsymbol{d}} \leftarrow G(\boldsymbol{n})$ ▷ generate fake data
        $S_{\text{fake}} \leftarrow D(\hat{\boldsymbol{d}})$ ▷ get the score of fake data
        $S_{\text{real}} \leftarrow D(\boldsymbol{d})$ ▷ get the score of real data
        $\boldsymbol{m} \leftarrow \epsilon \boldsymbol{d} + (1-\epsilon)\hat{\boldsymbol{d}}$
        $V_p \leftarrow \dfrac{\partial D(\boldsymbol{m})}{\partial \boldsymbol{m}}$ ▷ gradient penalty
        update $D(\cdot)$ based on Equation 5.4 ▷ back-propagate and apply Adam
      generate another noise vectors $\boldsymbol{n}$
      $\hat{\boldsymbol{d}} \leftarrow G(\boldsymbol{n})$ ▷ generate another fake data
      $S_{\text{fake}} \leftarrow D(\hat{\boldsymbol{d}})$ ▷ get score on fake data
      update $G(\cdot)$ based on Equation 5.3
    visualize $\hat{d}$ and save $D(\cdot)$ and $G(\cdot)$ regularly

5.6 Results and discussions

5.6.1 Manual inspection

One major issue of evaluating results from GAN is the lack of proper metrics. There are quantitative measurements that check if the generated examples are in the same distribution as the provided data. However, there are still no clear metrics that directly show us when to stop the training. One intuitive and still efficient way of verifying is to check the generated samples manually. From inspection, the generator stops improving efficiently after the 100^{th} epoch, despite oscillations continuing on the lost functions. Figure 5.2 shows the result after training for 100 epochs.

In Figure 5.2, we can see that the generated traces look like the real data. The number on the upper-left refers to the scores from the discriminator. Note that the negative signs do not have physical meaning since the score is not bounded and only the relative difference matters (higher is better). We can see that the generated traces achieved similar scores as the real data, which means the discriminator treats them as the same. Specifically, the zero

response of the discriminator is -10.824, which means the trained discriminator still cannot distinguish zero traces and traces with reflections. However, the discriminator responses to white noises range from -45 to -38, which can be safely considered as "different".

One common problem of a GAN is mode collapse, where the generator learns only one style presented by the data. In our case, the generator may end up producing similar traces all the time. The model collapse is less likely to have happened in our case since the result in Figure 5.2 shows great divergence. Besides, most of the examples can reproduce the unique waveform mentioned earlier in both normal and reversed polarity.

Figure 5.3 shows the loss curves for the first 100 epochs and Figure 5.4 shows the evolution of generated examples during the process. Since the two losses are competing with each other, we can see the loss curves are not guaranteed to drop all the time. In general, the curves are in a mirror relationship. Most severe competition happens during the first epochs. Before the 10^{th} epoch, the discriminator loss decreases drastically. This is because the discriminator's job at the early stage is to distinguish white noise generated by the generator (Figure 5.4a) and physically meaningful real data (Figure 5.4f), which is relatively easy. In the meanwhile, this is also a corresponding steep increase in the generator loss. Although the generator loss is increasing, the generator is much improved (Figure 5.4b) because the discriminator's feedback is useful. As the generator generates more reasonable results, the discriminator's job becomes harder. After the 20^{th} epoch, the generator loss starts to decrease, which indicates the discriminator learns slower compared to the generator. By comparing Figure 5.4d, 5.4e and 5.4f, we conclude that the two networks reach equilibrium and can hardly be improved.

5.6.2 Quantitative analysis

Figure 5.5 shows the distribution of the scores on the trained generator from the trained discriminator after 100 epochs. The real data score distribution is shown in blue, which can be assumed to be Gaussian. The mean discriminator score of the real data is -10.940,

and that of the generated samples is -10.932. The scores are close, and the score from generated samples is slightly higher than the real data distribution. This indicates the discriminator may get confused and stops improving itself. Figure 5.6 shows the mean generated score using the discriminator at the 100^{th} epoch. Note that the generator stops improving from the perspective of the 100^{th} discriminator, even though its gradients come from the discriminator at later epochs. We can infer that both the generator and the discriminator stops improving at around the 100^{th} epoch, which roughly agrees with our observations using manual inspection. Therefore, we chose the models at the 100^{th} epochs to be the best model. However, this conclusion is subjective and made after the training process. There are no clear metrics indicating the stopping point during the training, and further study is needed on this topic.

5.7 Conclusion

In this chapter, we have explored a way of generating 1D seismic traces using WGAN. The trained generator is able to transform uniformly distributed noise vectors to data distribution generated by the forward modelling. The two models reach equilibrium at around 100 epochs and hardly improve each other afterwards. The generated samples from the trained model preserve the unique waveform of real data, despite the discriminator still lacks the ability to distinguish empty traces from real examples. The future work will be expanding the same model architecture to 2D and applying conditions to the noise vector to gain more control over the generation process. Although the context of this chapter is generating 1D shots, GAN is not limited to the same purpose. For example, one can generate 2D fault images using the same architecture.

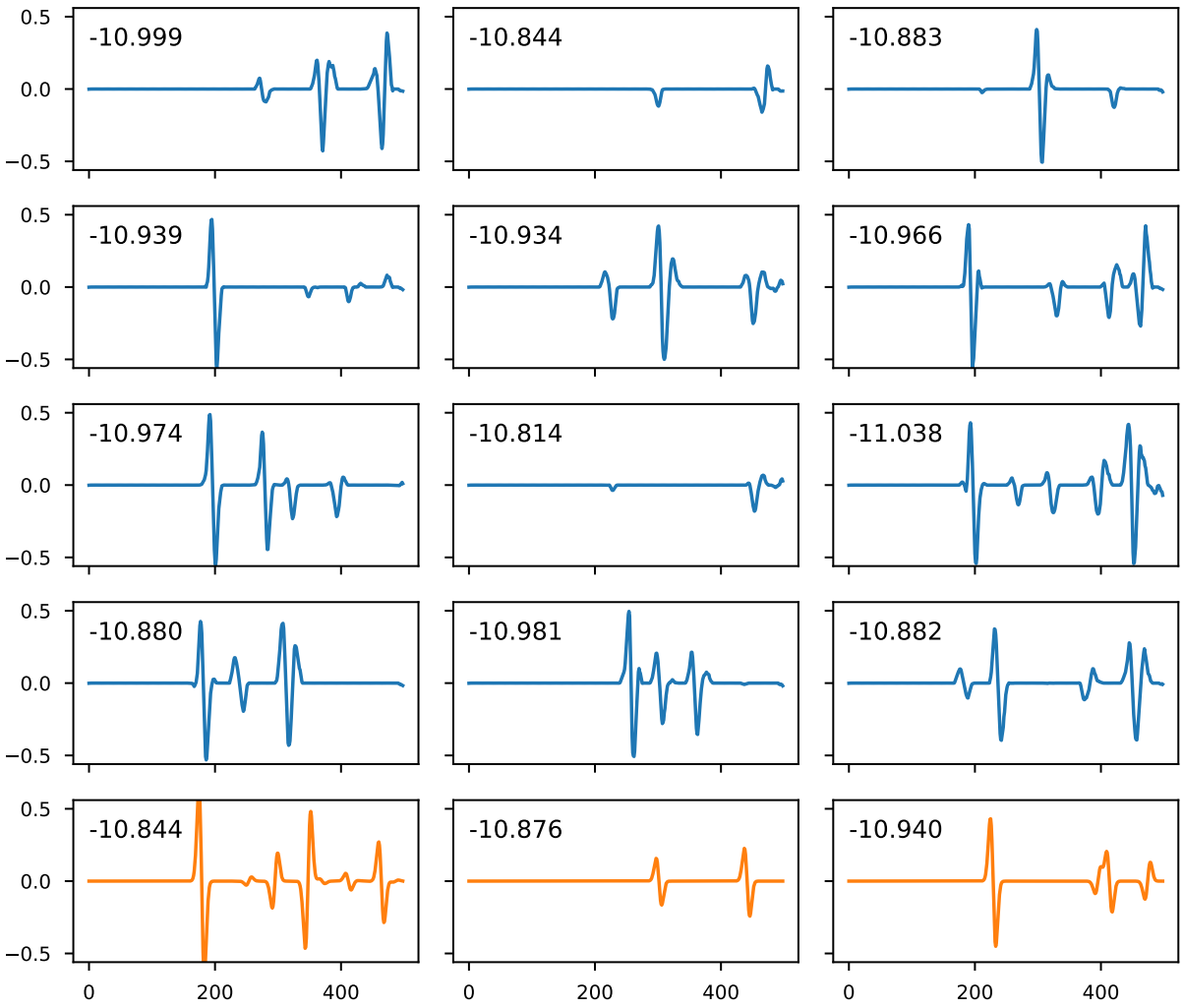

Figure 5.2: The results from the trained generator. The blue curves refer to the generated traces while the orange traces are from the data. The number on the upper left in each subplot refers to the scores obtained from the discriminator. The higher the score, the better it looks from the perspective of the discriminator.

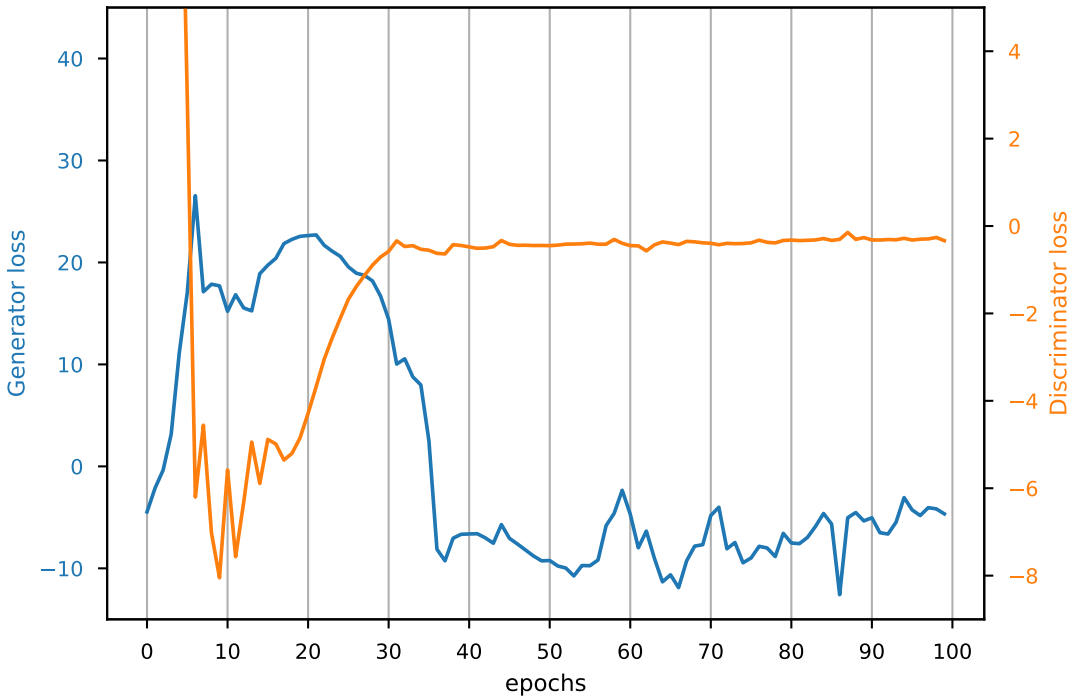

Figure 5.3: GAN loss curves. The blue and the orange lines refer to the losses of generator and discriminator, respectively. The losses are defined by Equation 5.3 and 5.4.

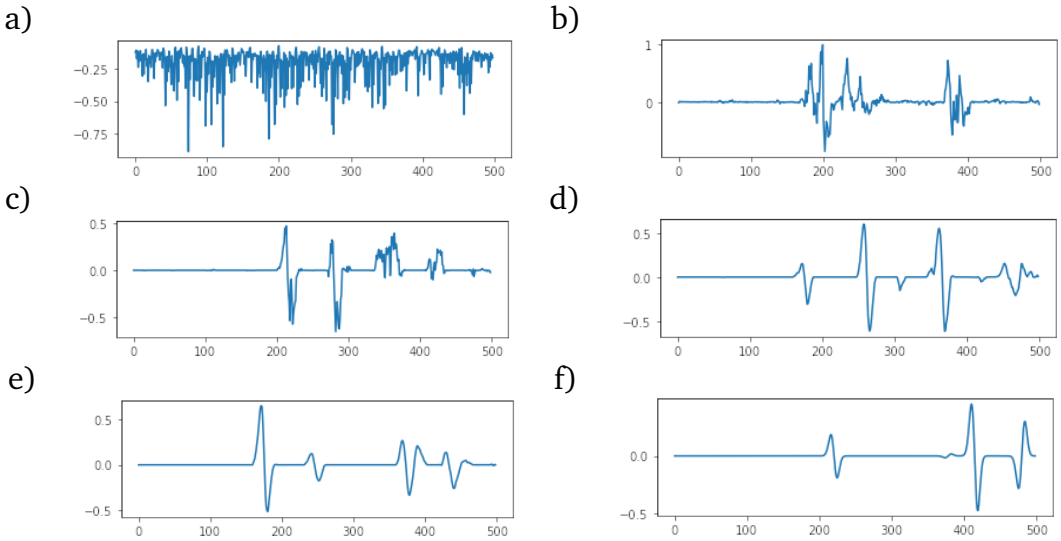

Figure 5.4: Predictions from the generator at **a)** 1st epoch; **b)** 5th epoch; **c)** 20th epoch; **d)** 47th epoch; **e)** 100th epoch and **f)** refers to a sample from real data for comparison.

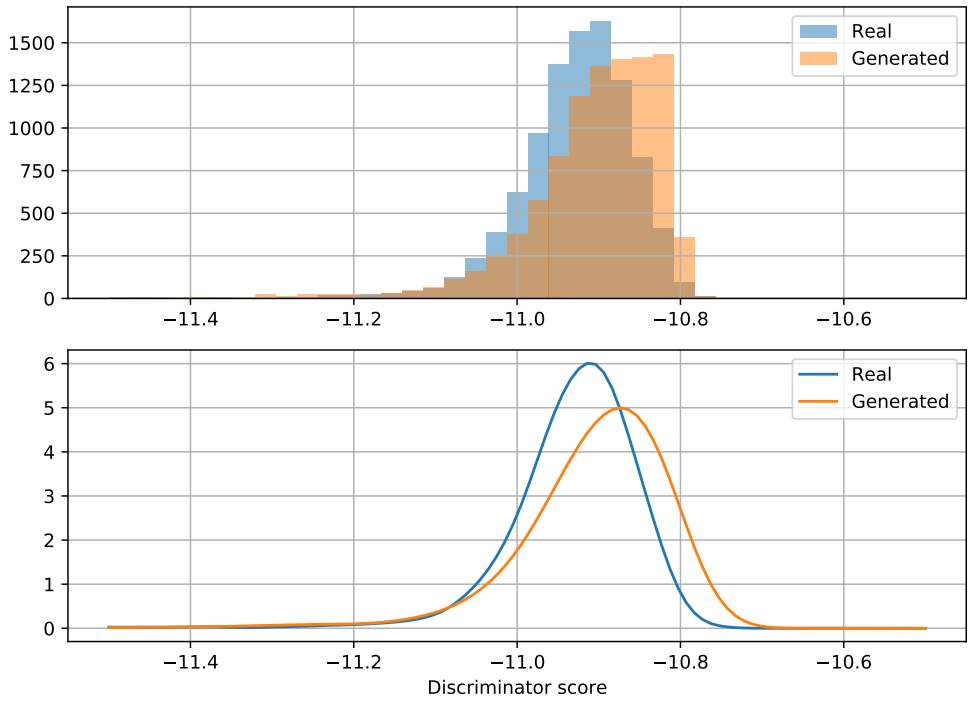

Figure 5.5: Histogram and kernel density estimation of real data and generated samples. The parts in blue represent the results from real data distribution, while the orange parts represent the results from generated examples.

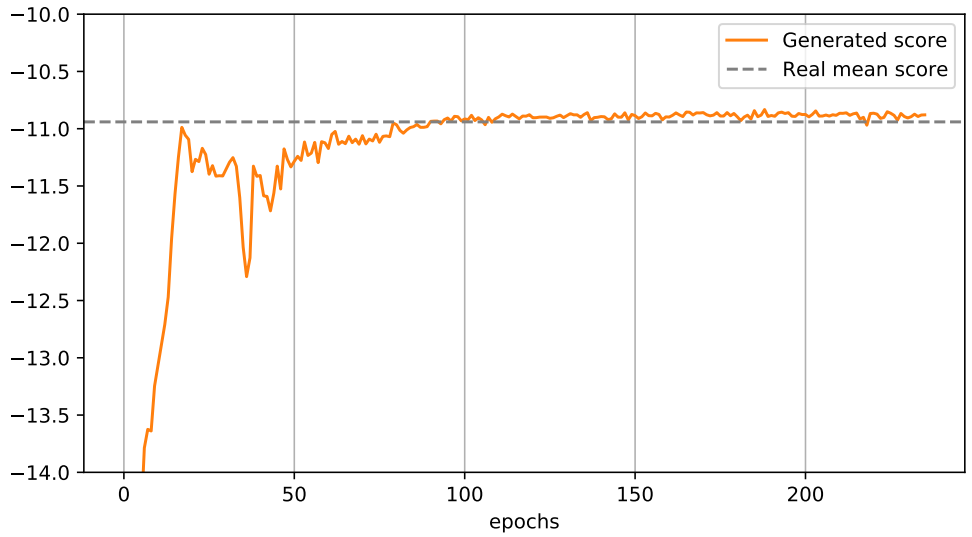

Figure 5.6: Mean generator score using the discriminator from the 100$^{\text{th}}$ epoch. The grey dashed line refers to the mean real data score, which is -10.940.

Chapter 6

Conclusions

In this thesis, we have investigated several applications of machine learning to the solution of geophysical problems. The emphasis of this research has been on introducing physical principles into the algorithms to facilitate achieving a meaningful solution. These principles act as constraints that limit the type of solutions that the networks will produce.

In Chapter 2, we trained a U-Net to separate pseudo-deblended shots. The trained network can successfully remove noise with the same amplitude of the signal. We found the best optimizer and training parameters through experiments for the current problem setup. The trained model preserved small diffractions and left minor residuals on the test dataset. The model performed slightly worse for the shots at the edge of the model because of the lack of training pictures representative of this case.

We also tested the performance with data that comes from a different velocity distribution. Specifically, the velocity model was made to be simpler than the training model. Therefore the corresponding pseudo-blended shots are simpler than the training set and they should not exceed the trained model's capability. In this case, however, the network performs okay but not as well as the first case. This observation indicates the model still memorizes part of the training data and provides a direction for improvement.

More work is required to generalize the network to new problems. The direction is to

apply constraints and reduce flexibility. Another direction for future work is to make the network predict the residual instead. In this way, the model can be trained and applied iteratively on the residual to reduce the error.

In Chapter 3, we implemented the Born modelling using TensorFlow API. The physics of wave propagation was incorporated into the problem by matching the finite difference method with the architecture of RNN so that the velocity model can be calculated via the back-propagation method. Back-propagation finds the inverse of the Born modelling automatically and can be proven to have a close connection to the LSRTM formulation. After trying different optimizers, we found that the Adam optimizer is the most efficient in speed, but requires careful hyperparameter tuning. The other two non-linear optimizers tested, FR-CG and L-BFGS-B, could not outperform Adam. Back-propagation and the optimizers are very powerful in the sense that they can find the step length automatically, but the price to pay for this advantage is the need for more computation power and memory requirements. The preparation time before computation requires additional memory that scales with the number of time steps due to the design of Tensorflow. In the future, we are interested in bringing this method to the frequency domain and looking for a more suitable neural network structure for wave propagation.

In Chapter 4, we made attempts to recover the reflectivity from randomly generated 4-layer models using fully connected networks. Besides the main problem to tackle, we also investigated different behaviours when using ℓ_1/ℓ_2 norms as loss function. We conclude ℓ_1 is more suitable for this type of problem. We found ResNet shortcuts can reduce fluctuations and converge to a lower loss. The model performs poorly on data from different distributions of the training set, and hence it needs more generalization. Future work can include gradient boosting or seeking better representations of the input and outputs. Also, we can study the effects of total variation as an additional regularisation to the loss function.

In Chapter 5, we have explored a way of generating 1D seismic traces using WGAN.

The WGAN was built with a 1D convolutional generator and discriminator. The network was trained with the Wasserstein loss plus a gradient penalty added to achieve stability. The trained generator successfully transforms uniformly distributed 1D noise vectors to 1D synthetic seismic traces from the forward modelling. The trained generator and discriminator competing against each other and reach equilibrium at early epochs. The generated samples preserve the unique waveform of the synthetic data, despite the discriminator still lacking the ability to distinguish empty traces from real examples. The future work will be expanding the same model architecture to 2D seismic shots or images. Also, additional conditions can be applied to the noise vector to gain more control over the generation process.

As a final conclusion, except for the RNN application, which is highly constrained by physics, all the other applications show a common problem of over-parametrization. Each trained network is "overfitted" by memorizing specifics of the problems that are trying to solve and, as a consequence, needs extra attention to adapt to new problems. To improve generalization, we can try in the future to include transfer learning and regularization techniques that penalize or disfavour overfitting. These techniques include gradient boosting, which can combine the strengths of different methods and even out the weaknesses. Furthermore, we can apply more constraints to the problem. The constraints can be introduced by defining the problem better and add extra terms for the formulation. However, it requires understandings of the task to choose a proper constraint and usually requires trials.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

Anagaw, A. Y., & Sacchi, M. D. (2012). Edge-preserving seismic imaging using the total variation method. *Journal of Geophysics and Engineering*, *9*(2), 138–146.

Arjovsky, M., Chintala, S., & Bottou, L. (2017). Wasserstein generative adversarial networks. *International conference on machine learning*, 214–223.

Baardman, R., Tsingas, C. et al. (2019). Classification and suppression of blending noise using convolutional neural networks. *SPE Middle East Oil and Gas Show and Conference*.

Beasley, C. J., Chambers, R. E., & Jiang, Z. (1998). A new look at simultaneous sources. *Seg technical program expanded abstracts 1998* (pp. 133–135). Society of Exploration Geophysicists.

Bi, Z., Geng, Z., Gao, H., Wu, X., & Li, H. (2020). 3d relative geologic time estimation with deep learning. *Seg technical program expanded abstracts 2020* (pp. 1465–1470). Society of Exploration Geophysicists.

Biswas, R., Sen, M. K., Das, V., & Mukerji, T. (2019). Pre-stack inversion using a physics-guided convolutional neural network. *Seg technical program expanded abstracts 2019* (pp. 4967–4971). Society of Exploration Geophysicists.

Buda, M., Saha, A., & Mazurowski, M. A. (2019). Association of genomic subtypes of lower-grade gliomas with shape features automatically extracted by a deep learning algorithm. *Computers in biology and medicine, 109*, 218–225.

Chen, L.-C., Papandreou, G., Schroff, F., & Adam, H. (2017). Rethinking atrous convolution for semantic image segmentation. *arXiv preprint arXiv:1706.05587*.

Chollet, F. (2018). *Deep learning mit python und keras: Das praxis-handbuch vom entwickler der keras-bibliothek*. MITP-Verlags GmbH & Co. KG.

Claerbout, J. F. (1971). Toward a unified theory of reflector mapping. *Geophysics*, *36*(3), 467–481.

Fukushima, K. (1979). Neural network model for a mechanism of pattern recognition unaffected by shift in position-neocognitron. *IEICE Technical Report, A*, *62*(10), 658–665.

Girshick, R. (2015). Fast r-cnn. *Proceedings of the IEEE international conference on computer vision*, 1440–1448.

Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 580–587.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning* [http://www.deeplearningbook.org]. MIT Press.

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). Generative adversarial networks.

Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., & Courville, A. (2017). Improved training of wasserstein gans.

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

Karpatne, A., Atluri, G., Faghmous, J. H., Steinbach, M., Banerjee, A., Ganguly, A., Shekhar, S., Samatova, N., & Kumar, V. (2017). Theory-guided data science: A new paradigm for scientific discovery from data. *IEEE Transactions on Knowledge and Data Engineering*, *29*(10), 2318–2331.

Karpatne, A., Watkins, W., Read, J., & Kumar, V. (2017). Physics-guided neural networks (pgnn): An application in lake temperature modeling. *arXiv preprint arXiv:1710.11431*.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Lailly, P., & Bednar, J. (1983). The seismic inverse problem as a sequence of before stack migrations. *Conference on inverse scattering: theory and application*, 206–220.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *nature*, *521*(7553), 436.

Lipton, Z. C., Berkowitz, J., & Elkan, C. (2015). A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*.

Lomask, J., Clapp, R. G., & Biondi, B. (2007). Application of image segmentation to tracking 3d salt boundaries. *Geophysics*, *72*(4), P47–P56.

Moseley, B., Markham, A., & Nissen-Meyer, T. (2018). Fast approximate simulation of seismic waves with deep learning. *arXiv preprint arXiv:1807.06873*.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., & Lerer, A. (2017). Automatic differentiation in PyTorch. *NIPS Autodiff Workshop*.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., ... Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems 32* (pp. 8024–8035). Curran Associates,

Inc. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Richardson, A. (2018). Seismic full-waveform inversion using deep learning tools and techniques. *arXiv preprint arXiv:1801.07232*.

Richardson, A., & Feller, C. (2019). Seismic data denoising and deblending using deep learning. *arXiv preprint arXiv:1907.01497*.

Ronneberger, O., Fischer, P., & Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. *International Conference on Medical image computing and computer-assisted intervention*, 234–241.

Shi, Y., Wu, X., & Fomel, S. (2018). Automatic salt-body classification using a deep convolutional neural network. *Seg technical program expanded abstracts 2018* (pp. 1971–1975). Society of Exploration Geophysicists.

Stanton, A., & Wilkinson, K. (2018). Robust deblending of simultaneous source seismic data. *arXiv preprint arXiv:1812.06040*.

Sun, B., & Alkhalifah, T. (2019). Ml-descent: An optimization algorithm for fwi using machine learning. *SEG International Exposition and Annual Meeting*.

Sun, J., Niu, Z., Innanen, K. A., Li, J., & Trad, D. O. (2020). A theory-guided deep-learning formulation and optimization of seismic waveform inversion. *Geophysics*, *85*(2), R87–R99.

Tarantola, A. (1984). Inversion of seismic reflection data in the acoustic approximation. *Geophysics*, *49*(8), 1259–1266.

Taylor, H. L., Banks, S. C., & McCoy, J. F. (1979). Deconvolution with the l1 norm. *Geophysics*, *44*(1), 39–52.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., ... SciPy 1.0 Contributors. (2020). SciPy 1.0: Fundamental Algorithms for Scientific

Computing in Python. *Nature Methods*, *17*, 261–272. https://doi.org/10.1038/s41592-019-0686-2

von Rueden, L., Mayer, S., Beckh, K., Georgiev, B., Giesselbach, S., Heese, R., Kirsch, B., Pfrommer, J., Pick, A., Ramamurthy, R., et al. (2019). Informed machine learning–a taxonomy and survey of integrating knowledge into learning systems. *arXiv preprint arXiv:1903.12394*.

Weyn, J. A., Durran, D. R., & Caruana, R. (2019). Can machines learn to predict weather? using deep learning to predict gridded 500-hpa geopotential height from historical weather data. *Journal of Advances in Modeling Earth Systems*, *11*(8), 2680–2693.

Wright, S., & Nocedal, J. (1999). Numerical optimization. *Springer Science*, *35*(67-68), 7.

Wu, X., Liang, L., Shi, Y., & Fomel, S. (2019). Faultseg3d: Using synthetic data sets to train an end-to-end convolutional neural network for 3d seismic fault segmentation. *Geophysics*, *84*(3), IM35–IM45.

Xu, B., Wang, N., Chen, T., & Li, M. (2015). Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*.

Zhou, C., & Brown, S. (2020). Automatic velocity model building with machine learning. *Seg technical program expanded abstracts 2020* (pp. 1596–1600). Society of Exploration Geophysicists.

Appendix A

The derivation of the Born gradient

First, let us express the current perturbation wavefield with the two previous perturbation wavefields at time step t, $t+1$ and $t+2$, respectively. We get

$$\delta \boldsymbol{p}^{(t)} = \left(2 + \Delta t^2 {\boldsymbol{v}_0}^2 \nabla^2\right) \delta \boldsymbol{p}^{(t-1)} - \delta \boldsymbol{p}^{(t-2)} + \Delta t^2 \boldsymbol{m} \frac{\partial^2 \boldsymbol{p}_0^{(t-1)}}{\partial t^2} \tag{A.1a}$$

$$\delta \boldsymbol{p}^{(t+1)} = \left(2 + \Delta t^2 {\boldsymbol{v}_0}^2 \nabla^2\right) \delta \boldsymbol{p}^{(t)} - \delta \boldsymbol{p}^{(t-1)} + \Delta t^2 \boldsymbol{m} \frac{\partial^2 \boldsymbol{p}_0^{(t)}}{\partial t^2} \tag{A.1b}$$

$$\delta \boldsymbol{p}^{(t+2)} = \left(2 + \Delta t^2 {\boldsymbol{v}_0}^2 \nabla^2\right) \delta \boldsymbol{p}^{(t+1)} - \delta \boldsymbol{p}^{(t)} + \Delta t^2 \boldsymbol{m} \frac{\partial^2 \boldsymbol{p}_0^{(t+1)}}{\partial t^2} \tag{A.1c}$$

where $\delta \boldsymbol{p}$ refers to the perturbation wavefield and the superscript refers to the corresponding time step. $\boldsymbol{p}_0$ is the background wavefield. $\boldsymbol{m}$ refers to the velocity perturbation ($2\delta \boldsymbol{v}/\boldsymbol{v}_0$). Note that the last term in (A.1a) is the source term scaled by $\Delta t^2 {\boldsymbol{v}_0}^2$. This characteristic is used later in the proof.

Differentiate (A.1a) with respect to $\boldsymbol{m}$ and differentiate (A.1b) and (A.1c) with respect

to $\delta \boldsymbol{p}^{(t)}$. Then we get

$$\frac{\partial \delta \boldsymbol{p}^{(t)}}{\partial \boldsymbol{m}} = \Delta t^2 \frac{\partial^2 \boldsymbol{p}_0^{(t-1)}}{\partial t^2} \tag{A.2a}$$

$$\frac{\partial \delta \boldsymbol{p}^{(t+1)}}{\partial \delta \boldsymbol{p}^{(t)}} = 2 + \Delta t^2 {\boldsymbol{v}_0}^2 \nabla^2 \tag{A.2b}$$

$$\frac{\partial \delta \boldsymbol{p}^{(t+2)}}{\partial \delta \boldsymbol{p}^{(t)}} = -1 \tag{A.2c}$$

From Equation 3.7, the cost of each shot at a specific time slice is

$$J^{(t)} = \frac{1}{2}\Big(\boldsymbol{D}^{(t)} - {\boldsymbol{d}_{cal}}^{(t)}\Big)^2 = \frac{1}{2}\Big({\boldsymbol{d}_{obs}}^{(t)} - \mathcal{S}_{\boldsymbol{x}_r}\delta \boldsymbol{p}^{(t)}\Big)^2 \tag{A.3}$$

where $\mathcal{S}_{\boldsymbol{x}_r}$ is the sampling operator that extract the data from the wavefield at the receiver positions to form the shot record $\boldsymbol{d}_{cal}$. By taking the derivative with respect to the current perturbation wavefield $\delta \boldsymbol{p}^{(t)}$ on both side, we can get

$$\frac{\partial J^{(t)}}{\partial \delta \boldsymbol{p}^{(t)}} = -\mathcal{S}_{\boldsymbol{x}_r}\big(\boldsymbol{D}^{(t)} - \mathcal{S}_{\boldsymbol{x}_r}\delta \boldsymbol{p}^{(t)}\big) \tag{A.4a}$$

$$= -\mathcal{S}_{\boldsymbol{x}_r}\boldsymbol{r}^{(t)} \tag{A.4b}$$

$$= -\boldsymbol{r}^{(t)} \tag{A.4c}$$

Since $\mathcal{S}_{\boldsymbol{x}_r}$ is the sampling operator, $\mathcal{S}_{\boldsymbol{x}_r}\boldsymbol{r}^{(t)}$ will simply be $\boldsymbol{r}^{(t)}$. Similar to what is discussed by Richardson, 2018. The gradient of the const function with respect to tone wave field at specific time step t can be express as

$$\begin{aligned}\left.\frac{\partial J}{\partial \delta \boldsymbol{p}}\right|_t = &\left.\frac{\partial J}{\partial \delta \boldsymbol{p}}\right|_{t+2} \frac{\partial \delta \boldsymbol{p}^{(t+2)}}{\partial \delta \boldsymbol{p}^{(t)}} \\ &+ \left.\frac{\partial J}{\partial \delta \boldsymbol{p}}\right|_{t+1} \frac{\partial \delta \boldsymbol{p}^{(t+1)}}{\partial \delta \boldsymbol{p}^{(t)}} \\ &+ \frac{\partial J^{(t)}}{\partial \delta \boldsymbol{p}^{(t)}}\end{aligned} \tag{A.5}$$

According to the chain rule, the gradient of the cost J with respect to $\boldsymbol{m}$ at a specific

time step t can be express as

$$\left.\frac{\partial J}{\partial \boldsymbol{m}}\right|_t = \left.\frac{\partial J}{\partial \delta \boldsymbol{p}}\right|_t \frac{\partial \delta \boldsymbol{p}^{(t)}}{\partial \boldsymbol{m}} \tag{A.6}$$

By Substituting (A.5), (A.2b), (A.2c) and (A.2a) into (A.6), the equation becomes

$$\left.\frac{\partial J}{\partial \boldsymbol{m}}\right|_t = \left[\left(2 + \Delta t^2 {\boldsymbol{v}_0}^2 \nabla^2\right) \left.\frac{\partial J}{\partial \delta \boldsymbol{p}}\right|_{t+1} - \left.\frac{\partial J}{\partial \delta \boldsymbol{p}}\right|_{t+2} + \frac{\partial J}{\partial \delta \boldsymbol{p}^{(t)}}\right] \left[\Delta t^2 \frac{\partial^2 \boldsymbol{p}_0^{(t-1)}}{\partial t^2}\right] \tag{A.7}$$

where the last term in the first bracket equals to the residual time slice (Equation A.4c). If we compare the terms in the first bracket with terms in Equation A.1a, we can notice that $\left.\frac{\partial J}{\partial \delta \boldsymbol{p}}\right|_t$ is actually a wavefield at time step $t+1$ which uses the residual slice scaled by $1/\Delta t^2 {\boldsymbol{v}_0}^2$ as source, denoted by $\boldsymbol{B}(\boldsymbol{r}^{(t+1)}/\Delta t^2 {\boldsymbol{v}_0}^2, \boldsymbol{x}, t+1)$. $\boldsymbol{B}$ refers to a time-reverse-propagation of wavefield since the wavefield at time step t is calculated by the wavefields at future time steps $t+1$ and $t+2$. The term in the 2nd time derivative of the background wavefield ($\boldsymbol{p}_0$) scaled by Δt^2 at time step $t-1$, denoted by $\Delta t^2 \boldsymbol{F}(f, \boldsymbol{x}, t-1)$. $\boldsymbol{F}$ is the time-forward-propagation of wavefield since $\boldsymbol{p}_0$ is propagating in the positive time direction. Then, Equation A.7 becomes

$$\left.\frac{\partial J}{\partial \boldsymbol{m}}\right|_t = -\boldsymbol{B}(\boldsymbol{r}^{(t+1)}/\Delta t^2 {\boldsymbol{v}_0}^2, \boldsymbol{x}, t+1) \Delta t^2 \boldsymbol{F}(f, \boldsymbol{x}, t-1) \tag{A.8a}$$

$$\approx -\frac{1}{{\boldsymbol{v}_0}^2} \boldsymbol{B}(\boldsymbol{r}^{(t+1)}/\Delta t^2 {\boldsymbol{v}_0}^2, \boldsymbol{x}, t+1) \boldsymbol{F}(f, \boldsymbol{x}, t-1) \tag{A.8b}$$

With Equation A.8b, we can infer the gradient in the global scale instead of at individual time step

$$\left.\frac{\partial J}{\partial \boldsymbol{m}}\right|_t \approx -\frac{1}{{\boldsymbol{v}_0}^2} \boldsymbol{B}(\boldsymbol{r}, \boldsymbol{x}, T-t) \frac{\partial^2}{\partial t^2} \boldsymbol{p}_0(f, \boldsymbol{x}, t) \tag{A.9}$$

which is similar to the form of LSRTM.