Rheinisch-Westfälische Technische Hochschule Aachen

Master Thesis

Detecting Geological Structures in Seismic Volumes Using Deep Convolutional Neural Networks

Ying JIANG

Matriculation No.: 350785

22.02.2017

1st Examiner: Prof. Dr. Christian Bauckhage

2nd Examiner: Prof. Dr. Stefan Wrobel

Eidesstattliche Versicherung

Name, Vorname

Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum

Unterschrift

Contents

List of Figures

List of Tables

List of Abbreviations

Abbrevation	Meaning
DNN	deep neural networks
CNN	convolutional neural networks
FCN	fully convolutional networks
CRF	conditional random field
MRF	Markov random field
SVM	support vector machine
SGD	stochastic gradient descent
$ReLU$	rectified linear unit
FC	fully connected
$DBSCAN$	density-based spatial clustering of applications with noise
EM	electron microscopy

Abstract

Identifying the geological structures in seismic volumes is of great importance for oil and gas exploration. However, seismic data interpretation is a time consuming manual task even for experienced experts. In this thesis we propose an automatic method based on 2D and 3D convolutional neural networks (CNN) to detect geophysical structures such as channels and faults in 3D seismic volumes. We apply CNN as a local classifier to 2D and 3D patches around every voxel in the seismic volume in order to perform semantic segmentation. The models trained on patches from orthogonal views are then ensembled to improve the classification accuracy. We have conducted extensive experiments on the Parihaka and F3 volumes and presented detailed results. With the help of regularization techniques, our models generalize well to new data, despite the fact that only a small training set generated from weakly labeled data is provided. Our experiments show that convolutional neural networks trained on raw pixel intensities are capable of achieving high-quality segmentation results in the seismic interpretation field that requires specific domain knowledge.

1 Introduction

To locate the oil and gas reservoirs, seismic surveys need to be carried out and detailed interpretation need to be done by geophysicists. In this chapter, basic knowledge of seismic surveys are introduced in the first section. Our goal to automatically extract fault and channel structures is then specified as a semantic segmentation task, which is explained in the second section.

1.1 Seismic Survey

Petroleum exploration is the process of searching and localizing the oil and gas resources. Seismic surveys play a crucial role in petroleum exploration due to their high success rate in determine the location of oil and gas reservoirs.

Seismic surveys require large amounts of manpower and material resources. First, seismic data is acquired onshore or offshore with specialized vessels. Then the acquired raw seismic traces are processed in order to generate a data volume including a series of detailed images describing the underneath rock layers. Finally, the processed data are interpreted by geophysicists to provide a detailed analysis of underground geology. The workflow is illustrated in Figure 1.1:

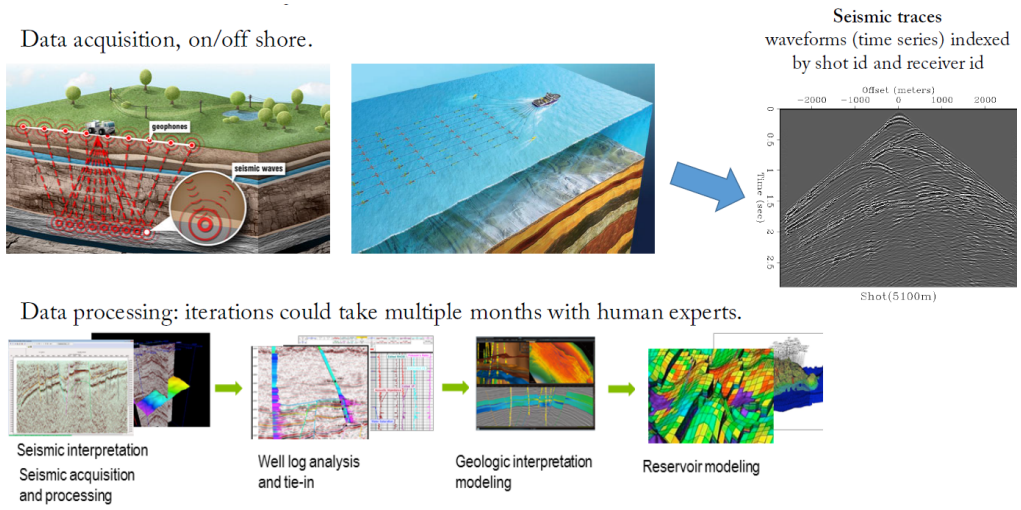

Figure 1.1: Seismic survey workflow, illustrated by Zhang et al. [2016].

1.1.1 Seismic Reflection

In seismic surveys, the seismic reflection method is the most common technique to acquire data from rocks beneath the earth's surface. It makes use of the different properties such

as density and refractive index of rock layers to visualize the underground structure. The fundamental mechanism of the seismic reflection method is as follows:

1. An energy source such as Vibroseis generates artificial shock waves. The waves propagate through the underground structures. On the rock interfaces, a part of the waves get reflected and travel back to the surface, the other part will refract and continue transmitting.
2. Receivers called geophones receive the reflected signal and record the waveform, this form of raw data is called seismic traces. When the receivers are arranged in a line, the acquired data presents seismic information of a 2D plane beneath that line. In order to get 3D seismic data, several lines of receivers are arranged densely in a certain region.
3. Based on the raw seismic traces, 2D or 3D seismic data can be reconstructed. A 3D seismic volume can be decomposed to a stack of 2D slices. Based on different views, they can be categorized as time slice (horizontal slice), inline slice (vertical slice that is parallel to the line of receivers) and crossline slice (vertical slice that is perpendicular to the line of receivers.) The arrangement of different views is depicted in Figure 1.2. Most of the interpretation procedures are carried out on the 2D slices and then gathered together. Namely, geophysicists mainly work on these images to interpret seismic data and analyze underground structures.

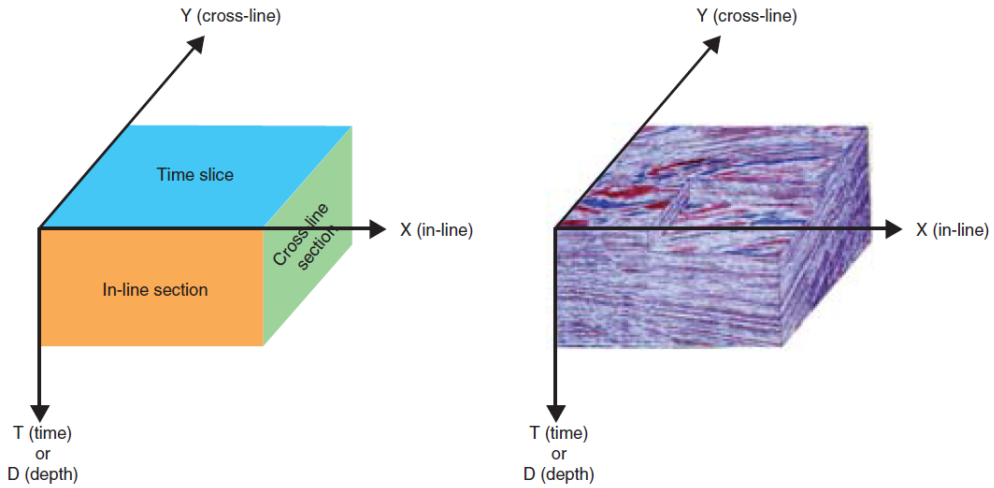

Figure 1.2: 3D seismic volume. Left: arrangements of time slice, inline and crossline section. Right: 3D seismic data volume. Figure by Chaouch and Mari [2006].

1.1.2 Seismic Interpretation

To analyze possible oil and gas deposits, seismic data need to be interpreted by geophysicists. Since the early 20th century, a lot of studies have been conducted in fields such as structural seismic interpretation and seismic attribute analysis.

The aim of structural seismic interpretation is to create structural maps for underground rock formations based on the acquired seismic volume. There are many 3D structure features that help with the search of oil and gas reservoirs such as fault, channel and salt domes. In

this thesis detecting faults and channels are our main focus.

- Fault Identifying faults is a crucial task for structural analysis, since faults are strong indicators for the movements of petroleum. From a commercial perspective, the locations of fault are important for planning the drill sites. Faults are caused by relative movement of adjacent rocks, therefore they appear as discontinuities in rock formations. In 3D seismic volumes they are normally seen as fault planes. The faults in real life and processed seismic volume are illustrated in Figure 1.3.

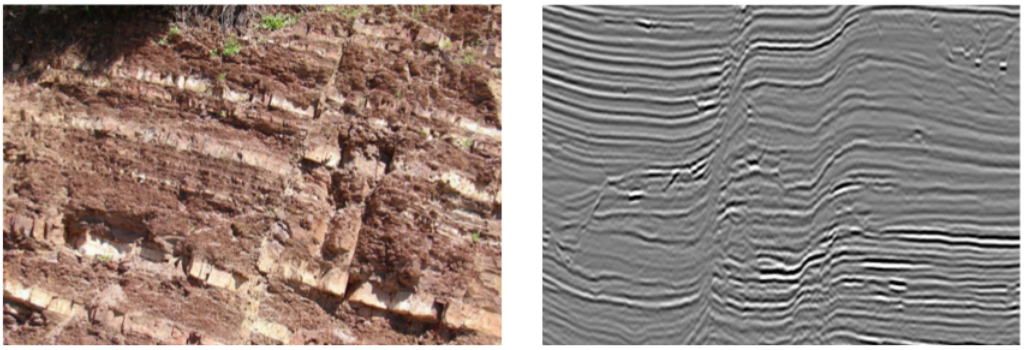

Figure 1.3: Left: a picture of faults in Morocco. Right: seismic inline-slice showing fault structures in underground rock layer.

Localizing faults used to be a time consuming manual task performed by geophysicists. With the development of image processing and machine learning techniques, automatic fault detection has been of particular interest to a growing amount of oil companies and researchers: Randen et al. [2001] developed a method to condition a series of chosen fault enhancing seismic attributes (which will be introduced later) to detect fault. They further extracted the fault plane by thinning the obtained result vertically and pick up the connected components. Pedersen et al. [2002] applied cooperative units called "artificial ants" for conditioning fault enhancing attributes. Gibson et al. [2003] used a coherency measure to generate semblance maps on top of vertical slices. The semblance map indicates where the discontinuities in the rock layers are significant. On top of the semblance map, points of interests are generated and merged in order to extract fault planes. Zhang et al. [2016] used a four-layer deep neural network to detect faults from the synthesized raw seismic traces. By assuming faults are straight lines in 2D images, they applied Wasserstein loss to predict the structured output.

- Channel The term channel in geography refers to the landform of rivers or straits. Normally a channel body is a narrow and shallow tubular passage where water or liquid used to flow through. Detecting channels is an important task in seismic interpretation because such structures are likely to be oil or gas reservoirs. In seismic data, a channels can be visualized with a horizontal time slice as Figure 1.4.

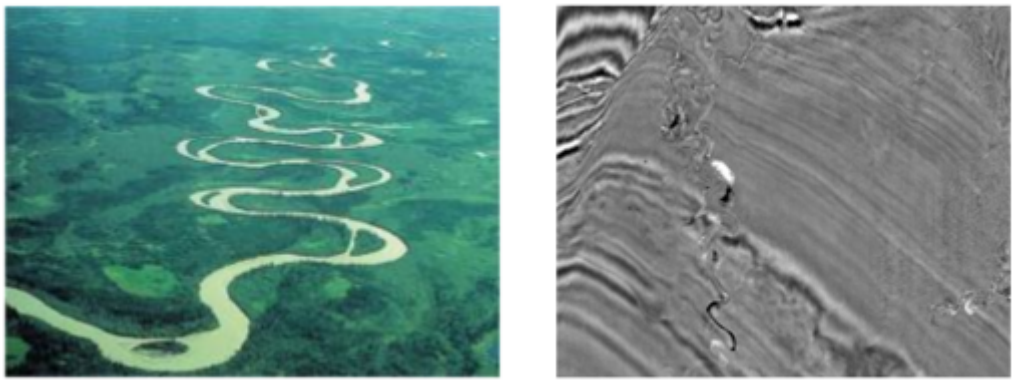

Figure 1.4: Left: A picture of natural channel. Right: Seismic time slice showing a channel structure in underground rock layers.

Compared to faults, channels are less evident in the interpretation process. So far there is not much literature providing methods for automatic channel extraction. Mohebian et al. [2013] used instantaneous spectral attributes to detect channels in an Iran oil field. Cao et al. [2015] introduced several color blending methods applied to seismic attributes data in order to better visualize channels. Hart [2008] proved the effectiveness of using a seismic attribute called sweetness to detect channels in deep-water and coastal-plain settings.

1.1.3 Seismic Attributes

Seismic attributes are hand-crafted filters which are commonly used to enhance the visibility of certain structures in seismic data. Based on properties of different structures, different seismic attributes are carefully designed accordingly.

There are many attributes with each different functions. For example, to reduce noise, a dip Steered median filter and frequency filter can be applied. For fault detection, similarity and semblance maps are often used.

The similarity attribute is able to highlight fault structures in seismic data, as shown in Figure 1.5. In this thesis our goal is to learn more generic attributes that can detects different structures using machine learning algorithms, in order to achieve similar or even better results.

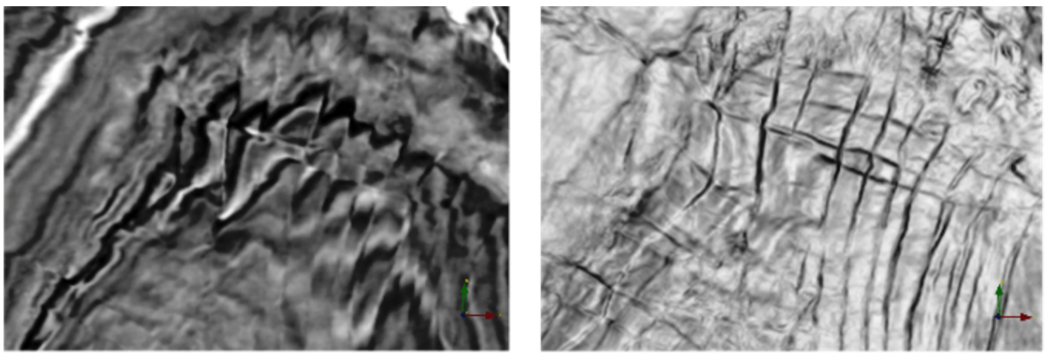

Figure 1.5: An example of the similarity attribute enhancing the fault structure. Left: input seismic trace (dip-steered filtered from raw trace) Right: effect of similarity attribute. Darker part shows high similarity, where faults are more likely to be located.

1.2 Image Segmentation

To extract channels and faults from seismic data, we want to partition the whole volume into multiple parts with category labels—channel, fault or background. More specifically, we want to assign a label to each voxel in the volume. In the image processing domain, this is called semantic segmentation or scene labeling. In this thesis we will use the term semantic segmentation since it is more commonly used in literature in recent years.

1.2.1 Segmentation

Research on image segmentation has been carried out for decades. Segmentation algorithms aim at partitioning the image based on features such as intensity, color or texture, in order to obtain disjoint sub-regions. A simple case of segmentation is shown in Figure 1.6.

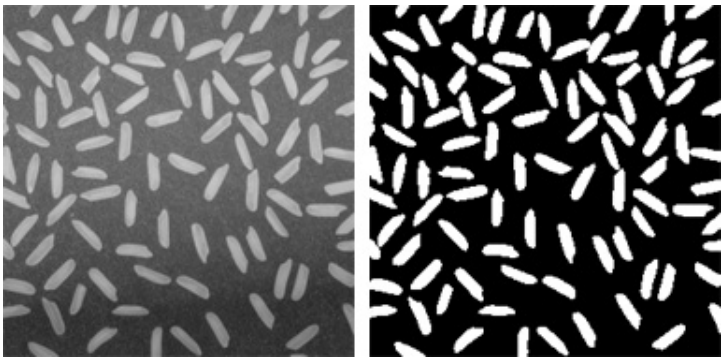

Figure 1.6: Example of segmenting an image into a binary mask showing foreground objects and background by thresholding.

Some basic methods for segmentation are:

- Threshold-based method is the simplest method for image segmentation. The algorithm calculates one or more thresholds according to a certain rule or function, and assigns the

pixels to different categories based on the thresholds.

-Edge based method is also widely used in segmentation tasks. The boundary of a region normally indicates a sharp change of intensity. Therefore, the gradients are important for detecting object boundaries or edges. There are already many well-developed edge detectors including the Canny operator and the Sobel operator. Based on the detected edges, further processing can follow to segment the images into regions with closed boundaries.

- Region-based method partitions the image based on certain similarity criterion. Different approaches to perform region-based segmentation are the following:

1. Seeded region growing: Starting from selected "seed" pixels in the image, the neighboring pixels of the seed can be merged into the seed region iteratively when a certain similarity criterion is fulfilled. The choice of "seed" is crucial for this algorithm.
2. Split and merge: the key idea of this method is to first split the image into several regions, iteratively, regions are either split into small regions or merged with neighboring regions according to certain rule.

- Clustering-based method partitions the image into several different clusters based on similarity of pixel color, intensity, location and other properties. One of the most widely adopted clustering-based methods is the K-means algorithm. The algorithm first picks k cluster centers, then repeatedly assign all pixels to the closest clusters and compute the new cluster centers until convergence. This algorithm is also used in the state-of-art super-pixel segmentation method, as will be discussed later.

There are many more studies on image segmentation which can not be all listed here. In this thesis the main focus is semantic segmentation, the bottom-up segmentation algorithms described above are helpful for preprocessing the image and post-processing the results.

1.2.2 Semantic Segmentation

As discussed above, image segmentation algorithms partition the image into separate regions without understanding the semantics of each region. Semantic segmentation aims to divide the images into regions which are labeled with one of the semantic classes. The ground truth as well as the target of a typical semantic segmentation task is shown in Figure 1.7. To extract faults and channels from the seismic data volume, we expect a similar formation of result.

Figure 1.7: Example from Pascal VOC segmentation challenge. The pixels belong to one object category are assigned the same color, otherwise it is categorized as background and shown in black.

Different approaches to perform semantic segmentation are the following:

1.2.2.1 Sliding window approach

One straightforward way to perform semantic segmentation is to apply pixel-wise classification with a sliding window. To be more precise, for each pixel in the image, a patch is extracted around the pixel, and machine learning algorithm is applied to classify the patch, as shown in Figure 1.8.

Figure 1.8: Simple pipeline of pixel-wise classification. The patch detector traverses the whole image and predict a label for each pixel.

Ciresan et al. [2012] used this approach for segmenting neuronal membranes in stacks of electron microscopy (EM) images. They applied a deep neural network (DNN) as the pixel classifier. Pixels belongs to two possible categories: membrane and non-membrane. The DNN model is trained with millions of patches extracted from ground truth images, and then tested with the same patch size on the test EM images. Their model won the ISBI 2012 segmentation of neuronal structures in EM stacks challenge with 0.6% pixel error.

1.2.2.2 Local classification approach

The sliding window approach provided pixel-level segmentation based on only short range information. The algorithm generates independent labels for each pixel across the image, regardless of their semantic surroundings, therefore the result usually lacks in consistency.

To address the above mentioned problem, many approaches have combined bottom-up segmentation and global features with the result of local classifier. A commonly used approach for semantic segmentation usually consists of three parts as depicted in Figure 1.9:

1. Over-segmentation: Partition the image into superpixels or segment candidates to insure visual consistency of labeling. Over-segmentation preserves the natural boundary of objects, while capturing redundancy of the image.
2. Classification:The features of every segment candidates are extracted and used to classify the segment. Each segment is assigned with one semantic label.
3. Post-processing: Normally a graphical model such as conditional random field (CRF) or Markov random field (MRF) is applied to the classification result, in order to constrain the local decision with global context.

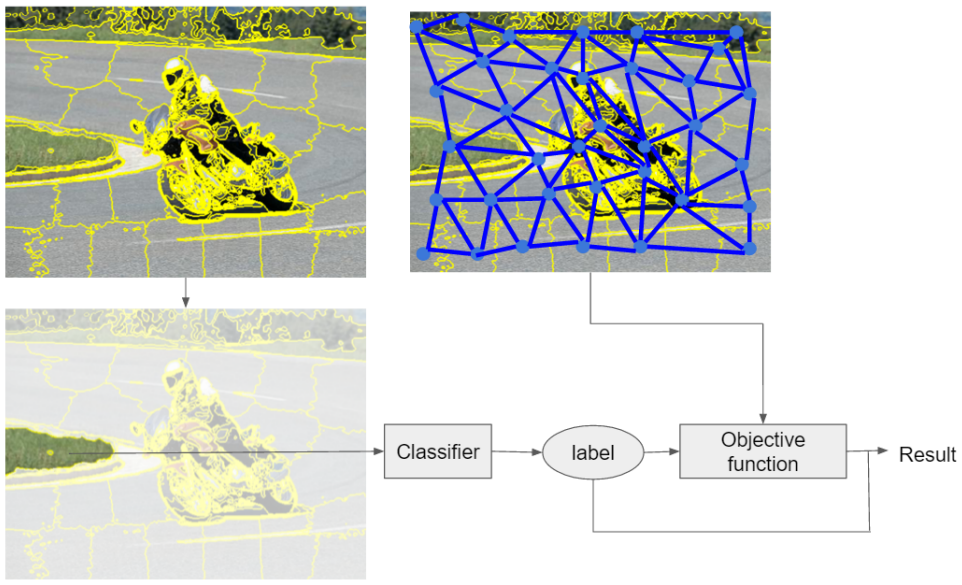

Figure 1.9: Superpixels are aggregated from consistent pixels. A local classifier is trained with features extracted from superpixels and assigns a label to each superpixel. The label is further constrained by the global context from the graph model.

Lucchi et al. [2010] proposed a superpixel based method to segment cellular structures in EM images. They first aggregate pixels to superpixels, which preserves the boundary of desired cellular structure. For each superpixel, they extract hand-crafted features to train a support vector machine (SVM). The class label of each superpixel are constrained by minimizing an objective function in the graph-cut algorithm.

Another problem that the sliding-window approach faces is a dilemma between "what" and "where": when a big contextual window is used, more surrounding information are provided to classify the center pixel, but at the same time the pixel's location is more uncertain only

obtain a coarse segmentation result can be obtained. Vice-versa, with a small contextual window, more pixel-level details can be shown in the segmentation result, but the lack of global information will sabotage the category-level classification performance.

Farabet et al. [2013] addressed this problem by combining information from multi-scale images. They downscale the image into three different sizes. For each scale in the image pyramid, they applied the same window size for extracting training patches. They train a convolutional neural network (CNN) on these training patches to generate feature maps. The feature maps were then upscaled to the same size and concatenated together in order to give the per-pixel prediction. They used several methods to combine the result with external bottom-up segmentation components such as superpixels and segmentation tree. One method is to apply majority-vote over the pixels in a component and decide a single class label per component.

1.2.2.3 End-to-end approach

In the past, most of the semantic segmentation tasks were performed as a combination of the classification model and external segmentation methods. Currently, more effort has been put in end-to-end systems. Such systems take the whole image as input and produce the segmentation directly as output.

The model that popularized end-to-end segmentation algorithm is called fully convolutional networks (FCN) by Long et al. [2015]. The basic architecture of FCN is shown in Figure 1.10.

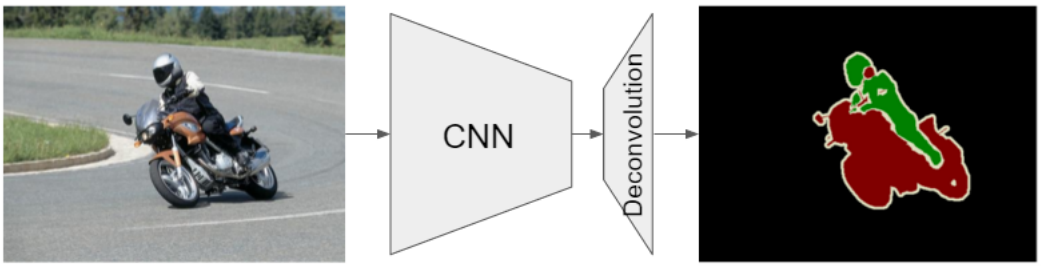

Figure 1.10: A simple demonstration of Fully convolutional networks architecture.

The first part of FCN is based on CNN, which we will discuss further in Chapter 2. The original image is normalized to size 224×224 and used as input to the neural network. Then the image is processed through a sequence of convolutional layers and pooling layers. In convolutional layers, a bank of filters (convolutional kernels) are trained to detects certain features. The pooling layer shrinks the feature maps spatially by subsampling. After different stages of layers in the CNN, hierarchical feature maps can be generated.

The second part of FCN is different from typical CNNs, which consist of fully connected layers at the end to classify the image to a single label or a bounding box coordinate. FCN aims at generating per-pixel labeling directly. Since the feature maps are spatially much smaller than the original image size, a learnable up-sampling or deconvolution layer is attached to the end of the network. This layer up-scale the feature maps and output the per-pixel label map instead of a single class label.

To combine the information of "what" and "where", the architecture takes the feature maps from different pooling layers, up-samples them to the image size and combines them together to make the final prediction for each pixel. It can be seen from the result in Figure 1.11, when features from only the last layer is used, the network yields a correct but coarse result showing a human and bicycle shape, and when more hierarchical features from earlier layers are added up, the segmentation result is more fine-grained with more details.

Figure 1.11: (a) Ground truth. (b) Segmentation result with feature map from the last layer. (c)(d) Segmentation results when feature maps from earlier layers are added.

2 Background

Deep learning is a new branch of machine learning based on artificial neural networks. By deploying neural networks with multiple layers with hundreds of neurons, deep neural networks are able to simulate highly complicated functions in order to solve tasks in computer vision, natural language processing and many other fields. Given enough computational power and massive amount of data, deep neural networks have achieved outstanding performance in many machine learning tasks such as image recognition and machine translation.

In this chapter, we will first discuss about the basic concepts and mechanism of artificial neural networks, and then introduce an important architecture—convolutional neural networks, which is the core technique in our method.

2.1 Neural Networks

Neural Networks in the machine learning domain are inspired by simulating biological neural networks. A typical artificial neural network is formed by neurons (also called perceptrons) organized in layers, including an input layer, several hidden layers and an output layer. In neural networks, a neuron is used as a logistic unit. A single neuron takes input $x = [x_1...x_n]$, computes the dot product between input vector and weight vector $w = [w_1...w_n]$, and adds bias b. To overcome the limitation of linearity, nonlinearity is introduced by passing the result to a nonlinear function called an activation function. The output is also called the activation of neuron. A single neuron functions as a logistic unit as shown in Figure 2.1.

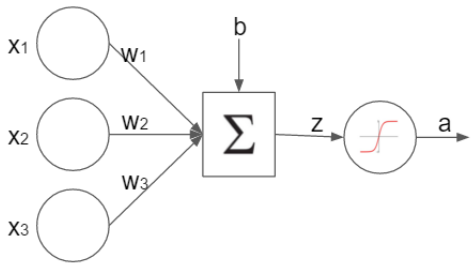

Figure 2.1: A single neuron as a logistic unit.

Formally, the function of a neuron can be described as follows:

$$z = wx + b \tag{2.1}$$

$$a = \sigma(z) \tag{2.2}$$

where $\sigma(\cdot)$ is a non-linear activation function, a is the activation of this neuron.

A neural network is a group of these neurons organized in layers, and interconnect to each other between adjacent layers. In such neural network architectures the information is flowing forward, therefore it is called a feed forward neural network. There are also neural networks with feedback connections such as recurrent neural networks, but they will not be discussed in this thesis.

In the past few decades, due to the lack of computational power, people were only able to train neural networks with one or two hidden layers. With the development of computer infrastructure and increasing use of GPUs, neural networks with more hidden layers and far bigger capacity, which we call deep neural networks, emerged as a new trend in the machine learning field. With multiple hidden layers, deep neural networks are able to approximate functions of very high complexity. A typical deep feed forward neural network is shown in Figure.2.2.

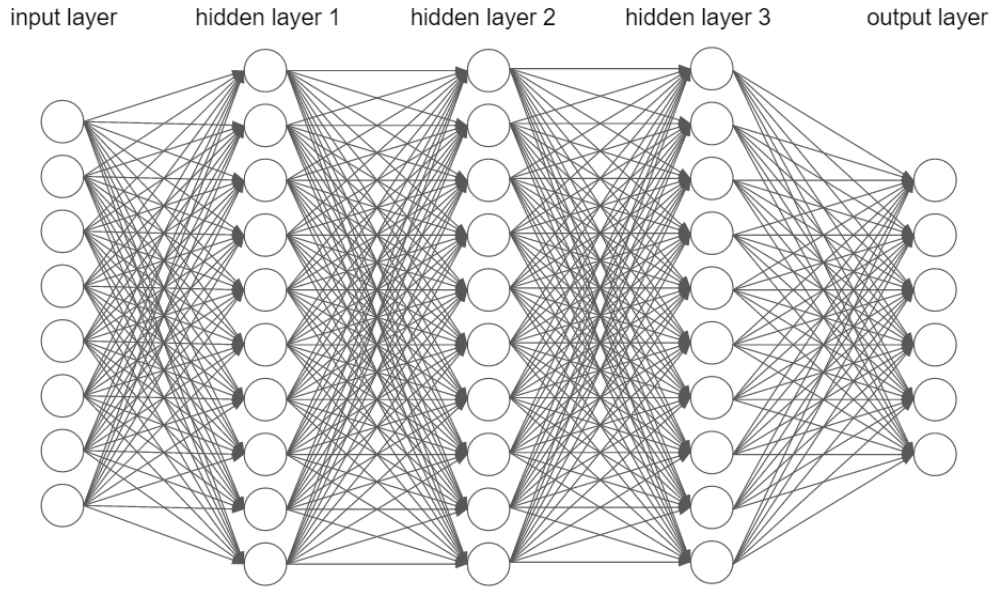

Figure 2.2: A typical fully connected deep neural network with four weight layers.

By applying deep neural networks, we can solve many tasks in computer vision. Nielsen [2016] gave an example of applying a simple three-layer neural network to recognize handwritten digits in the MNIST dataset. The input image size is 28×28, therefore the input layer has 784 neurons, each neuron is corresponding to one pixel in the original image. Labels (the digit shown on the image) are provided to the network as ground truth for calculating the loss. Instead of integers from 0 to 9, the label uses the one-hot representation (e.g.$[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]$ represents the digit 2), so that the output layer contains 10 neurons accordingly. Same as most of the machine learning algorithms, we expect such neural network to learn a set of parameters (weights and bias) that minimizes the loss function, so that when given an unseen image as input, it will predict the correct number shown in the image.

2.1.1 Gradient Descent

In neural networks, loss function is used to quantify the quality of the model. Given loss function $L(\theta)$, where θ is the collection of all trainable weights and biases, the problem becomes how to change the value of θ so that $L(\theta)$ is minimized over the whole training set X.

One of the most common ways to minimize the loss function is gradient descent. To illustrate the mechanism, we can imagine $L(\theta)$ as a plane with respect to two free parameters θ_1 and θ_2, the goal is then to find the lowest point of plane L, as shown in Figure 2.3.

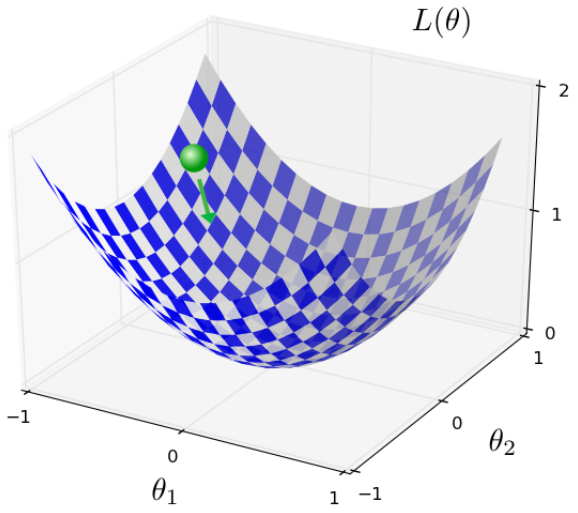

Figure 2.3: Loss function L visualized as a 2D plane, the ball follows the steepest gradient to reach the bottom of the valley. Figure from Nielsen [2016].

The gradient descent algorithm suggests that we always follow the steepest gradient to search for the lowest point, like a ball following the steepest direction to reach the bottom of a valley. We define the gradient of L as the partial derivative of its parameters:

$$\nabla L(\theta) = \frac{\partial L(\theta)}{\partial \theta} \tag{2.3}$$

Every time we find the steepest gradient, we will calculate the moving distance by multiplying it to a small number η and update θ (weights and biases) as follows:

$$\theta \leftarrow \theta - \eta \nabla L(\theta) \tag{2.4}$$

η decides how much change will be added to the parameters, so it is also called the learning rate.

The problem of this approach is, that when we have millions of samples for training, every update in the learning process will take a very long time to compute. Moreover, the whole dataset will not fit into computer memory most of the time. Therefore, in deep learning we often adopt another algorithm — stochastic gradient descent (SGD), also called mini-batch gradient descent — as optimizer. As the name suggests, for every update, the algorithm extracts a random batch of data from the whole dataset and approximates the real overall

gradient using the gradient based on this mini-batch.

Assume we have in total N samples in the dataset, and mini batch size is set to m, we can compute the average gradient over all data points in the batch, and update the parameters as follows:

$$\theta \leftarrow \theta - \frac{\eta}{m} \sum^{N} \nabla L(\theta) \tag{2.5}$$

Iterating through the whole dataset is called an epoch. In practice, stochastic gradient descent has been proved to be highly efficient and effective, therefore it is a widely used optimizer for neural networks.

2.1.2 Back Propagation

A neural network can have millions of parameters, to compute the gradients of those parameters, another important technique—back propagation is introduced. In other words, the goal of back propagation algorithm is to calculate $\frac{\partial L}{\partial b}$ and $\frac{\partial L}{\partial w}$.

Intuitively, the gradient of each neuron is a measurement for how much influence this neuron will donate to the final error. Therefore, we define the error at the j_{th} neuron on the l_{th} layer:

$$\delta_j^l = \frac{\partial L}{\partial z_j^l} \tag{2.6}$$

Apparently, in the output layer where $l = l_{max}$, the error is the difference between the activation of the neuron and the label y:

$$\delta_j^{l_{max}} = a_j^{l_{max}} - y_j \tag{2.7}$$

notice that $a^l = \sigma(z^l)$.

The error can be back propagated through each layer, therefore the l_{th} layer's error can be calculated from the $(l+1)_{th}$ layer as follows:

$$\delta^l = w^{l+1} \cdot \delta^{l+1} \odot \sigma'(z^l) \tag{2.8}$$

where $\odot$ is the element-wise product.

By combining eq.2.7 and eq.2.8, we can back propagate through all layers starting from the output layer, and calculate their error δ^l.

To calculate the gradients with respect to w and b, we can apply chain rule:

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial v} \cdot \frac{\partial v}{\partial t} \tag{2.9}$$

where $u = f(v)$ and $v = g(t)$.

Therefore:

$$\frac{\partial L}{\partial b_j^l} = \frac{\partial L}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \tag{2.10}$$

$$\frac{\partial L}{\partial w_{jk}^l} = \frac{\partial L}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial w_{jk}} = a_k^{l-1} \cdot \delta_j^l \tag{2.11}$$

where w^l_{jk} represents the weight of the connection between the k_{th} neuron in the $(l-1)_{th}$ layer and the j_{th} neuron in the l_{th} layer, b^l_j is the bias of the j_{th} neuron in the l_{th} layer.

In conclusion, back propagation algorithm recursively back propagates the error from the last layer to compute the error of each layer, then calculate the gradients of all intermediate parameters using chain rule.

In general, the training of a neural network includes two processes:

1. Information flow forward through the network and compute the output.
2. Error between the output and ground truth propagates backwards in order to update parameters to minimize the loss function.

2.2 Convolutional Neural Network

One drawback of applying the above mentioned neural network in the computer vision field is that such architecture ignores the natural 2D structure of the image by flattening the input image to a 1D array. Convolutional neural networks overcome this limitation and has made great success on many tasks in the computer vision field in the past few years. In this section, we will discuss the background and mechanism of convolutional neural networks.

2.2.1 Biological Inspiration

The concept of convolutional neural networks is inspired by the biological study on the visual cortex of cats by Hubel and Wiesel [1959]. They conducted experiments on the visual information processing procedure in cat's brain by showing the cat different visual stimuli and examining the firing of individual nerve cells. Stimuli were presented with different parameters such as shape, orientation and moving speed.

Their study showed that the firing of a cortical cell is influenced by a small region on the retina called receptive field. According to the complexity of receptive fields, the cells can also be categorized as simple cells and complex cells (in a later study they further defined a "hypercomplex cell" on top of complex cell). For simple cells, the most effective shape of stimuli is a long narrow rectangle or edge, the orientation of the rectangle is a critical factor to the response of simple cells. Complex cells, however, respond to variously-shaped stimuli with properties of more restrictive requirements. Compared to simple cells, complex cells respond to larger regions in receptive field. Their further study showed that complex cells are of higher order and simple cells serve as their afferents. In the sequential processing, complex cells, which receive projection from lower level simple cells, are less critical about basic information like orientation and position, and care more about generalized abstraction.

In general, cells with similar functionalities are spatially close to each other and are organized in hierarchical orders, as upper layers containing complex cells and lower layer containing simple cells. In this way, a larger receptive field can be formed from small receptive fields. This scheme for complex receptive field is illustrated by Hubel and Wiesel [1962] in Figure 2.4:

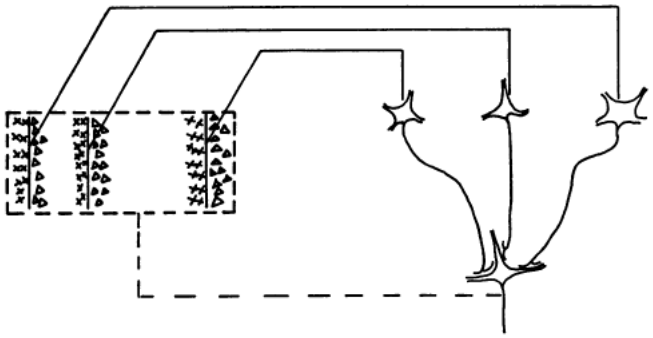

Figure 2.4: Scheme of complex receptive field. Three cells with simple receptive field project to complex cortical cell of higher order. Left shows the receptive field of these simple cells locating in different areas in the rectangle and detecting vertical edges over the whole area.

2.2.2 Architecture

Compared to fully connected neural networks, CNNs have three main features in their architecture: sparse connectivity/local respective field, weight sharing, and pooling.

In a fully connected neural network, the neurons in layer $l+1$ are connected to all the neurons in the previous layer l, which we call a fully connected layer as Figure 2.2 depicts. Each connection is assigned with a weight w. Assume that we have m neurons in layer l and n neurons in layer $l+1$, together with biases we will have $(m+1) \times n$ parameters.

However, in a convolutional neural network, the neurons in layer $l+1$ are only sparsely connected to a small number of neurons in layer l.

To better illustrate the functionality of CNN, we can perceive the neurons in layer l and $l+1$ as 2D matrices as shown in fig.2.5. As can be seen, each neuron in layer $l+1$ receives information from a patch of neighboring neurons in the previous layer l. Therefore, the sparse connectivity shown on the left is also called local receptive field shown on the right, with respect to the research of cortex cells.

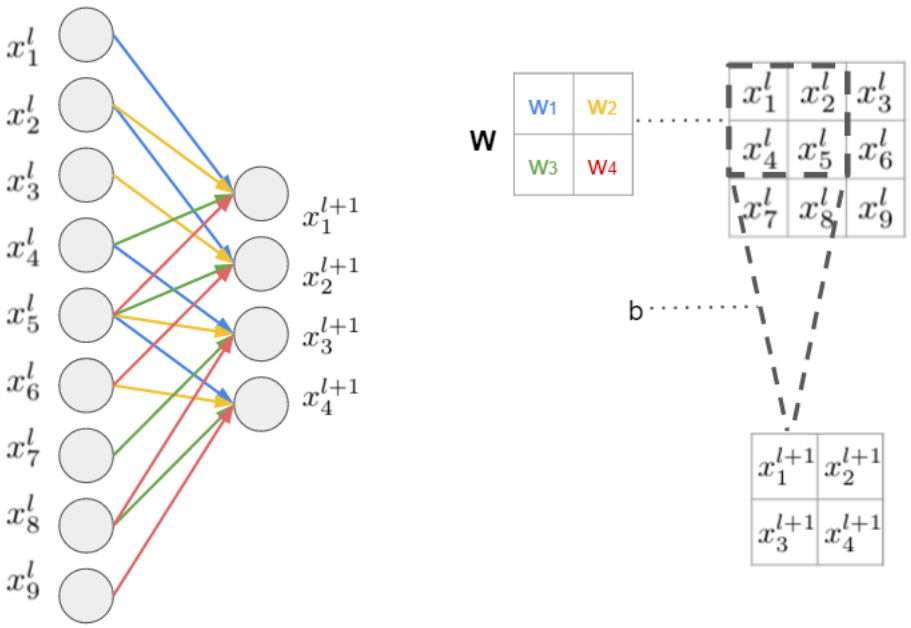

Figure 2.5: Left: Sparsely connected layer in CNN. Every neuron in $(l+1)_{th}$ layer is only connected to four neurons in l_{th} layer.

In the example, layer $l+1$ has a receptive field of size 2×2, since each neuron can "see" a 2×2 region in the previous layer. Layer $l+2$ will slightly expand the receptive field to size 3×3, since it gathers information of all nine pixels in layer l.

We illustrate the same weights by connections with the same color. It can be seen that the neurons in layer $l+1$ share a same set of weights $W = [w_1, w_2, w_3, w_4]$ and bias b despite there being 16 connections. By weight sharing, convolutional neural networks have largely reduced the amount of free parameters.

Assume the neurons represent pixels in an image, the input from layer l is a 3×3 image. The shared weights can be perceived as a 2D matrix, which is called a convolutional kernel or a filter. The filter traverses the whole image by a certain stride (in this example the stride is set to 1). At each position, the dot product is computed between the filter and the neurons within its receptive field, then the shared bias is added. In other words, the input image in layer l is filtered by a 2×2 convolutional kernel, which detects a certain feature on all positions of the image. Therefore the activations of layer $l+1$ is called feature map.

2.2.2.1 Convolutional layer

To detect different features, multiple convolutional kernels of size $P \times Q$ are applied to the input of size $W \times H$ in a convolutional layer. The corresponding result feature maps are stacked up as shown in Figure 2.6.

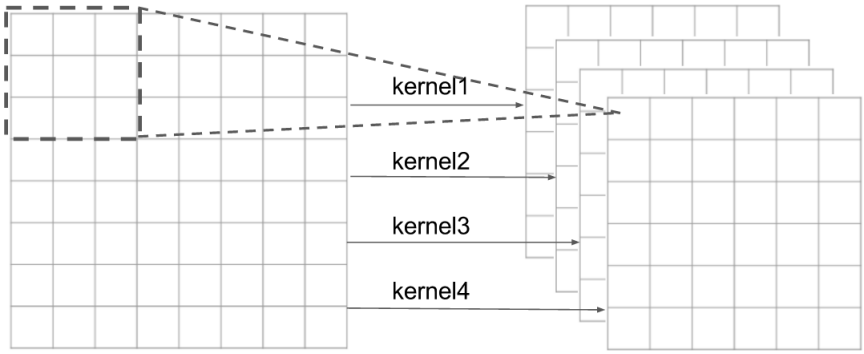

Figure 2.6: Assume $P = Q = 3$, $W = H = 8$, the figure shows an input image of size 8×8 filtered by four convolutional kernels of size 3×3. The convolution operation is performed with a stride of 1 and no padding. Each convolution kernel generates a feature map of size 6×6. These feature maps are stacked together, therefore the output of the convolutional layer is a feature map of size $6 \times 6 \times 4$.

Formally, the activation of neuron x at position (i, j) on the m_{th} feature map in the $l_t h$ layer is given by:

$$x_{ij}^{lm} = \sigma(b^{lm} + \sum_{m} \sum_{p=0}^{P-1} \sum_{q=0}^{Q-1} w_p q^l m x_{(i+p)(j+q)}^{(l-1)m}) \tag{2.12}$$

Similarly, if the input image/feature map is a volume of size $W \times H \times D$, the 2D convolutional kernel will have size $P \times Q \times D$ accordingly, as shown in Figure 2.7.

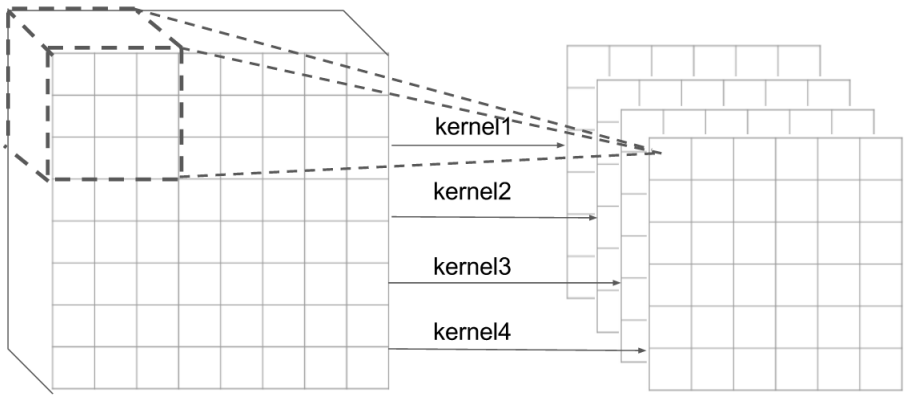

Figure 2.7: Assume $P = Q = 3$, $W = H = 8$, $D = 5$, the input image of size $8 \times 8 \times 5$ filtered by four convolutional kernels of size $3 \times 3 \times 5$. The convolution operation is performed spatially with a stride of 1 and no padding. Each convolution kernel generates a feature map of size 6×6, these feature maps are stacked together therefore the output feature map remains $6 \times 6 \times 4$.

Without padding, the feature map is shrinked spatially. Normally in convolutional layers, zero padding is applied to keep the spatial size of the feature map for convenience.

One advantage of the convolution operation is that it can detect features regardless of its position in the image. Besides, due to the sparse connection and shared weights, convolutional layers have tremendously reduced the number of free parameters, and therefore makes

the network easier to train and to generalize.

2.2.2.2 ReLU Layer

As discussed in the former section, to introduce nonlinearity into the network, a nonlinear activation function is necessary. In CNNs, after every convolutional layer, the result feature map is passed to an nonlinear layer.

One of the most commonly used activation functions in neural networks is *tanh* function:

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.13}$$

One disadvantage of *tanh* activation is that the neuron can get saturated. Namely, when z is a relatively large number, the output will hardly change when we adjust the value of z, as we can see from Figure 2.8. It brings the problem that during back-propagation, the gradient will be "killed". Another disadvantage of *tanh* is that computing e^z is expensive.

Rectified linear unit (ReLU) has solved the above mentioned issues by simply performing a thresholding at 0, as described below:

$$f(z) = max(0, z) \tag{2.14}$$

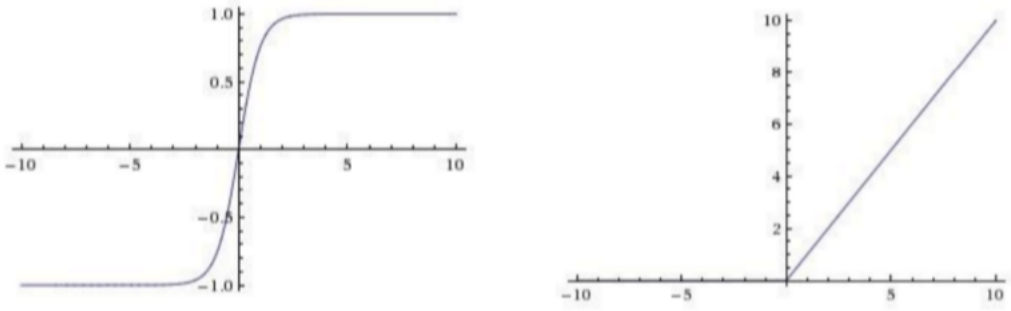

Figure 2.8: Left: *tanh* activation function. Right: ReLU activation function

In recent years, most CNNs adopted ReLU as the activation function. One important reason to use ReLU is the training time. According to Krizhevsky et al. [2012], by applying the ReLU activation function to CNNs, the training speed can be several times faster than applying *tanh* activation without sacrificing performance. Recently, some other activation functions such as pre-ReLU and leaky ReLU have been proved to be effective, but so far ReLU is the most commonly used activation function in deep CNNs.

For brevity, in this thesis we use the ReLU as an activation function instead of depicting it as an extra layer.

2.2.2.3 Pooling layer

Another important architecture idea of convolutional neural network is pooling. A pooling layer performs a down-sampling operation spatially, as illustrated in Figure 2.9.

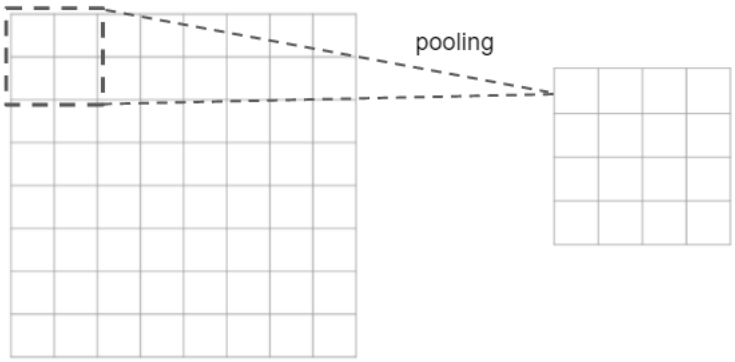

Figure 2.9: An example of pooling operation. The input of a pooling layer is a 8×8 feature map, activation of each neuron in the pooling layer is calculated from a 2×2 region from the input. The kernel performs pooling function with stride 2, thus generating a feature map of 4×4.

Pooling layer operates on each depth slice independently using pooling functions, including max-pooling, average pooling and L2 pooling. The most common pooling function is max-pooling. It takes the maximum value of the region in the input as the corresponding neuron's activation.

The goal of the pooling layer is to reduce the resolution of the input feature map and concentrate the information. Once the convolutional layer has detected certain features, the exact position of the feature becomes less important to the final decision but rather harm the generalization ability of the model. Therefore, a pooling layer is applied to reduce the precision of the position information of features.

Meanwhile, with the same receptive field size, latter layers can receive information from a larger region in the original image, so that global information instead of details in the image can be valued more in latter layers to help with the final decision.

2.2.2.4 Architecture

In general, a convolutional neural network is constructed by a sequence of alternating convolutional layers and pooling layers, as shown in Figure 2.10.

The image is first passed to a convolutional layer, where convolutional kernels detect feautures and generate feature maps as output. In some models they also use two or more consecutive convolutional layers as will be discussed later. After convolution, the resolution of the feature map is reduced spatially by the following pooling layer. The pattern of CONV + POOL is repeated multiple times in a sequence. Finally, to produce scores for different classes, the output of the last pooling layer is flattened and passed to several fully-connected layers.

Commonly, the last fully connected layer uses a $softmax$ activation to normalize the computed scores to the range $[0, 1]$ smoothly. Moreover, the sum of all output after $softmax$ is 1 so that the output of the network can be interpreted as probabilities assigned to each class.

Given the input vector $z = [z_1 ... z_K]$ which represents the scores to K classes computed by

the network, the output of the $softmax$ function for the j_{th} neuron is given as below:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{2.15}$$

The overall architecture of a convolutional neural network is depicted as below.

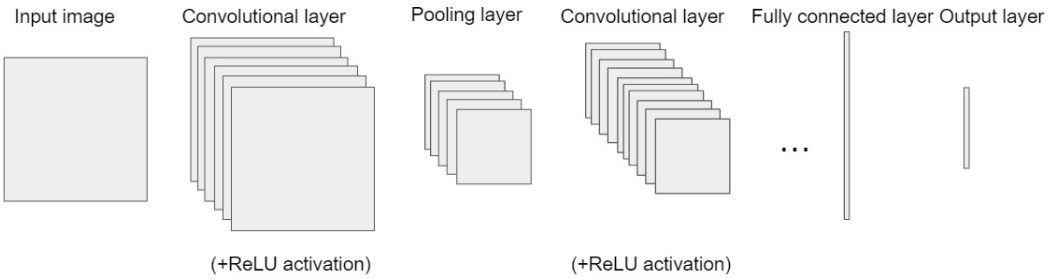

Figure 2.10: Common architecture of a convolutional neural network. The image is passed to a convolutional layer, then a pooling layer shrinks the feature map spatially. After that, more convolutional layers and pooling layers are arranged in the same pattern, which have been omitted in this figure. At the end of CNN, several fully connected layers are arranged. The output layer is commonly a fully connected layer with softmax activation, which gives the prediction of probabilities to each class for the input image.

Similar to the findings of the cortex cells studies, lower convolutional layers detect basic features such as edges and corners, while upper convolutional layers receive the output feature map of lower layers and detect more complicated features such as a swirl or an eye. With the sequential processing of signal, the detected features gets more complex and global. A visualization of features in different layers are given by Zeiler and Fergus [2014] as below:

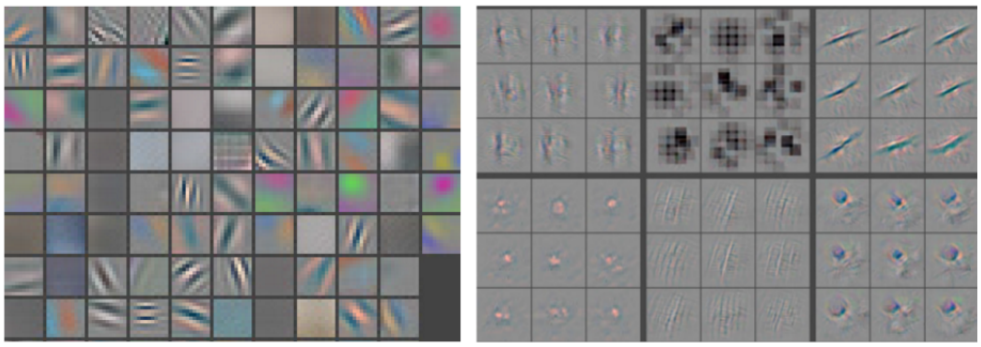

Figure 2.11: Left: First layer features from AlexNet. Right: Visualizations of second layer features from AlexNet by Zeiler and Fergus [2014].

2.3 Image Classification

In the computer vision domain, image classification is a typical task and has been studied for years. The goal of image classification is to identify the objects presented in the image.

In this section, we will introduce three popular CNN architectures which perform image classification.

2.3.1 LeNet

LeCun et al. [1998] first proposed a convolutional neural network called LeNet and successfully applied it to recognize isolated handwritten digits in the MNIST dataset. The architecture of LeNet-5 is shown in Figure 2.12. For brevity we use the format <type> + <number> such as CONV1 and POOL2 to specify each layer.

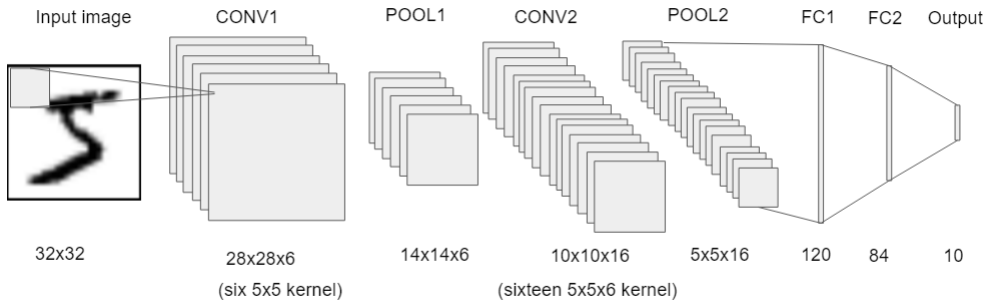

Figure 2.12: Architecture of LeNet-5

It takes 32×32 grayscale image as input. The first convolutional layer CONV1 applies six filters of size 5×5. The number of trainable parameters in this layer is $(5 \times 5 + 1) \times 6 = 156$. Without any padding, the result feature map size becomes $28 \times 28 \times 6$.

The second layer is a pooling layer POOL1, it downsamples the feature map by 2×2 kernel. In downsampling operation, they add up the four values in the 2×2 kernel and multiply the sum by a trainable coefficient and then add a trainable bias. Therefore there are 12 trainable parameters in this layer.

CONV2 layer applies sixteen filters of size 5×5. In the original paper, they applied different filters on different subsets of the feature map in order to reduce the number of connections. The number of trainable parameters in this layer is 1,516. However, due to the development of computational power, nowadays we would simply apply a filter of size $5 \times 5 \times 6$ to the feature map without too much concern about the training time.

Similarly, POOL2 layer uses a 2×2 kernel to shrink the feature map and obtains 32 trainable parameters. LeNet-5 has used a sigmoid function as the activation function.

Fully connected layer FC1 can also be interpreted as a convolutional layer with 5×5 kernel size. FC1 has $120 \times (5 \times 5 \times 16 + 1) = 48,120$ trainable parameters. FC2 has $120 \times 84 = 10,080$ trainable parameters and the output layer has $84 \times 10 = 840$ trainable parameters. Therefore in total this five layer architecture has 60,840 free parameters, and most of the parameters come from fully connected layers.

They presented in their paper that with this architecture, the test result based on 60,000 training samples of MNIST has reached an error rate of 0.95%. After they distort the original training images and generated a 540,000 dataset, the test error decreased to 0.8%.

They also showed that with the growth of the training set size, the network performance has increased considerably.

LeNet-5 is an important milestone in the history of CNN. Not only did it build an early prototype for convolutional neural networks for academia research, but it has also been widely applied in US banks and other industries until recent years.

2.3.2 AlexNet

In 2012, ImageNet database was released and it has been responsible for accelerating the development of deep learning to a large extent. Before ImageNet, most datasets of labeled images were relatively small (on the scale of 10,000) and contained only a limited number of categories of objects. ImageNet provided 14 million labelled images from over 22,000 categories. Most images are real life pictures with natural surroundings, some examples can be seen in Figure 2.13.

Figure 2.13: Some examples from ImageNet dataset for image classification. Each image has at least one label indicating which category it belongs to.

Since then, image classification on a large scale has been a popular research topic and many powerful models have been invented to compete in the ImageNet challenge.

AlexNet by Krizhevsky et al. [2012] won the ImageNet 2010 challenge to classify 1.2 million images into 1000 categories. AlexNet has a similar architecture compared to LeNet. However, it is considerably bigger in both breadth and depth. AlexNet has 11 layers and in total 60 million parameters. The architecture is shown in the Table 2.1.

Table 2.1: AlexNet Architecture

Layer	Size	Parameters
Input	$224 \times 224 \times 3$	$224 \times 224 \times 3$ RGB image
CONV1	$55 \times 55 \times 96$	kernel:96@$11 \times 11 \times 3$; stride:4; padding:0
POOL2	$27 \times 27 \times 96$	kernel:3×3; stride:2 (overlapping pooling)
CONV3	$27 \times 27 \times 256$	kernel:256@$5 \times 5 \times 96$; stride:1; padding:2
POOL4	$13 \times 13 \times 256$	kernel:3×3; stride:2
CONV5	$13 \times 13 \times 384$	kernel:384@$3 \times 3 \times 256$; stride:1; padding:1
CONV6	$13 \times 13 \times 384$	kernel:384@$3 \times 3 \times 384$; stride:1; padding:1
CONV7	$13 \times 13 \times 256$	kernel:384@$3 \times 3 \times 384$; stride:1; padding:1
POOL8	$6 \times 6 \times 256$	kernel:3×3; stride:2
FC9	4096	$6 \times 6 \times 256 \times 4096$ connections
FC10	4096	4096×4096 connections
FC11	1000	4096×1000 connections

Note that in the original paper, they split the above architecture into two parts in order to train on seperate GPUs. For simplicity, we merged their distributed architecture to an equivalent architecture as shown above.

Instead of *tanh*, AlexNet used ReLU as activation function of the convolutional layers. CNNs with the ReLU nonlinearity are easier to train, and also converge considerably faster. In Krizhevsky et al. [2012] it is shown that to reach the same error rate of 25%, networks with *tanh* nonlinearity take 35 epochs while the same network with ReLU nonlinearity takes only seven epochs.

To reduce overfitting, AlexNet applies data augmentation on the original training dataset. By randomly extracting 224×224 patches from the normalized 256×256 image, and flipping the images horizontally, they enlarged the original dataset by a factor of 2048. Another anti-overfitting method they deployed is the dropout regularizer in the fully connected layers. These two techniques are widely used in CNNs, as will be dicussed in Chapter 4.

AlexNet achieved 37.5% and 17.0% on the top-1 and top-5 error rates, which at that time both beat the state-of-art system by more than 10%. After the success of AlexNet on large scale image recognition task, CNN has become one of the major trends in the computer vision field.

2.3.3 VGG Net

On the basis of AlexNet, many new architectures have been built to achieve better performance. VGGNet by Simonyan and Zisserman [2014] became one of the most popular CNN architectures. Compared to other architectures, VGG Net has two main features:

1. Use of 3×3 receptive fields with stride 1 for all the convolutional layers, rather than large receptive fields like 11×11 in AlexNet. It is demonstrated in their paper that by stacking up two of these convolutional layers, their actual receptive field would be 5×5, as discussed before. Similarly, while three of such convolutional layers are stacked in a sequence, their actual receptive field grows to 7×7. On the one hand, such configuration can reduce the number of parameters, as 7×7 receptive field will have 49 parameters while three 3×3 receptive field will have only 27 free parameters. On the

other hand, more convolutional layers introduces more nonlinearity to the network.

2. VGG Net is considerably deeper than previous networks. With the small receptive field and stacked up convolutional layers, VGG Net managed to train CNNs with more weight layers. They experimented on six different settings from 11 weight layers to 19 weight layers. As the network goes deeper, they also introduced convolutional layers with 1×1 kernel size in order to increase the nonlinearity of the decision function.

The most commonly used VGG architecture is VGG-16, which contains 16 weight layers and in total 21 layers. Below the setting of VGG-16 is listed.

Table 2.2: VGG-16 Architecture

Layer	Size	Parameters
Input	$224 \times 224 \times 3$	$224 \times 224 \times 3$ RGB image
CONV1	$224 \times 224 \times 64$	kernel:64@$3 \times 3 \times 3$; stride:1; padding:1
CONV2	$224 \times 224 \times 64$	kernel:64@$3 \times 3 \times 64$; stride:1; padding:1
POOL3	$112 \times 112 \times 64$	kernel:2×2; stride:2
CONV4	$112 \times 112 \times 128$	kernel:128@$3 \times 3 \times 64$; stride:1; padding:1
CONV5	$112 \times 112 \times 128$	kernel:128@$3 \times 3 \times 128$; stride:1; padding:1
POOL6	$56 \times 56 \times 128$	kernel:2×2; stride:2
CONV7	$56 \times 56 \times 256$	kernel:256@$3 \times 3 \times 128$; stride:1; padding:1
CONV8	$56 \times 56 \times 256$	kernel:256@$3 \times 3 \times 256$; stride:1; padding:1
CONV9	$56 \times 56 \times 256$	kernel:256@$3 \times 3 \times 256$; stride:1; padding:1
POOL10	$28 \times 28 \times 256$	kernel:2×2; stride:2
CONV11	$28 \times 28 \times 512$	kernel:512@$3 \times 3 \times 256$; stride:1; padding:1
CONV12	$28 \times 28 \times 512$	kernel:512@$3 \times 3 \times 512$; stride:1; padding:1
CONV13	$28 \times 28 \times 512$	kernel:512@$3 \times 3 \times 512$; stride:1; padding:1
POOL14	$14 \times 14 \times 512$	kernel:2×2; stride:2
CONV15	$14 \times 14 \times 512$	kernel:512@$3 \times 3 \times 512$; stride:1; padding:1
CONV16	$14 \times 14 \times 512$	kernel:512@$3 \times 3 \times 512$; stride:1; padding:1
CONV17	$14 \times 14 \times 512$	kernel:512@$3 \times 3 \times 512$; stride:1; padding:1
POOL18	$7 \times 7 \times 512$	kernel:2×2; stride:2
FC19	4096	$7 \times 7 \times 512 \times 4096$ connections
FC20	4096	4096×4096 connections
FC21	1000	4096×1000 connections

In total, VGG-16 has 133 million parameters.

Through experiments, they found out that with the increasing number of layers, the testing error decreased. VGG-16 achieved 26.3% and 8.2% for top-1 and top-5 error. VGG-19 performs the best of all six architectures presented in the paper and achieves 24.4% and 7.1% respectively.

Due to the standardized formation, VGG Net can save most efforts in tuning hyper-parameters in CNNs such as kernel size and number of kernels. Since then, researchers such as He et al. [2016] and Long et al. [2015] use the architecture of VGGNet as foundations. In this thesis, the network we build is also based on the design idea of VGGNet, as will be discussed in Chapter 3.

3 Methods

In this thesis, we adopt the local classifier approach while combining local patch prediction and bottom up segmentation. More global information is introduced to the system according to domain knowledge. The reason to use this approach is simple: unlike cat and dog pictures, seismic data are fairly hard to collect and annotate, and it is almost impossible to provide millions of fully annotated images to train an end-to-end system. The advantage of the local classifier approach is that from one annotated image we can generate hundreds of thousands of training patches, therefore training with deep neural networks is possible. In recent years, most of medical image segmentation systems, such as Ciresan et al. [2012], adopt this approach due to the same reason.

The workflow of our approach is shown in Figure 3.1. We first annotated the public available seismic data. 2D and 3D training patches are extracted based on the annotations. The patches are used to train the deep neural networks. A sliding window with corresponding patch size then goes through every pixel on the test slices and performs pixel-wise classification using the trained model. The local classification results are combined with long-range global information from bottom-up segmentation methods. The 2D test results are stacked up to form the final 3D result showing the desired structures.

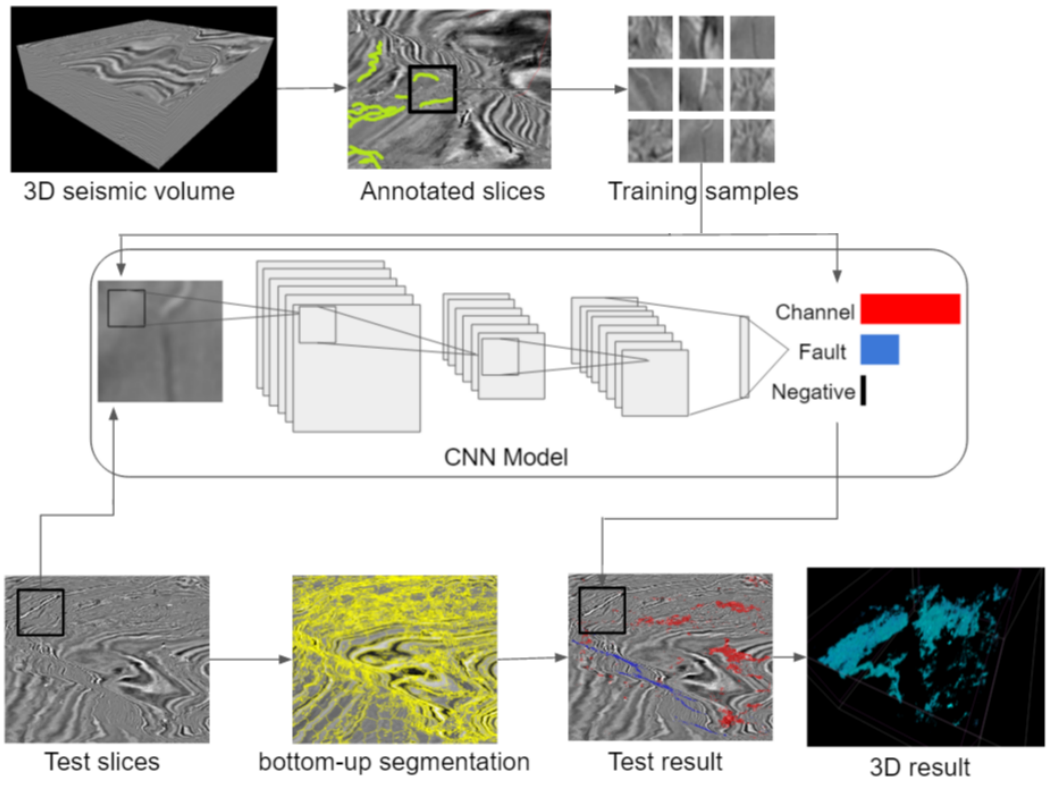

Figure 3.1: An overview of our approach described in this thesis.

3.1 Data

Since there is no similar study or available dataset, we need to build the system from scratch. In this section, we introduced the process of generating training data for the CNNs.

3.1.1 Data Annotation

There is hardly any annotated seismic data available since the data are of high importance for oil companies. Therefore, annotations are done by ourselves on the free public seismic data.

Parihaka dataset is a public seismic volume provided by New Zealand Crown Minerals. It is a 3D volume with a resolution of $268 \times 923 \times 1126$. A 3D overview of Parihaka dataset is visualized in Figure 3.2. The 2D images from three orthogonal views— time slice, inline slice and crossline-slice — can be seen in Figure 3.3.

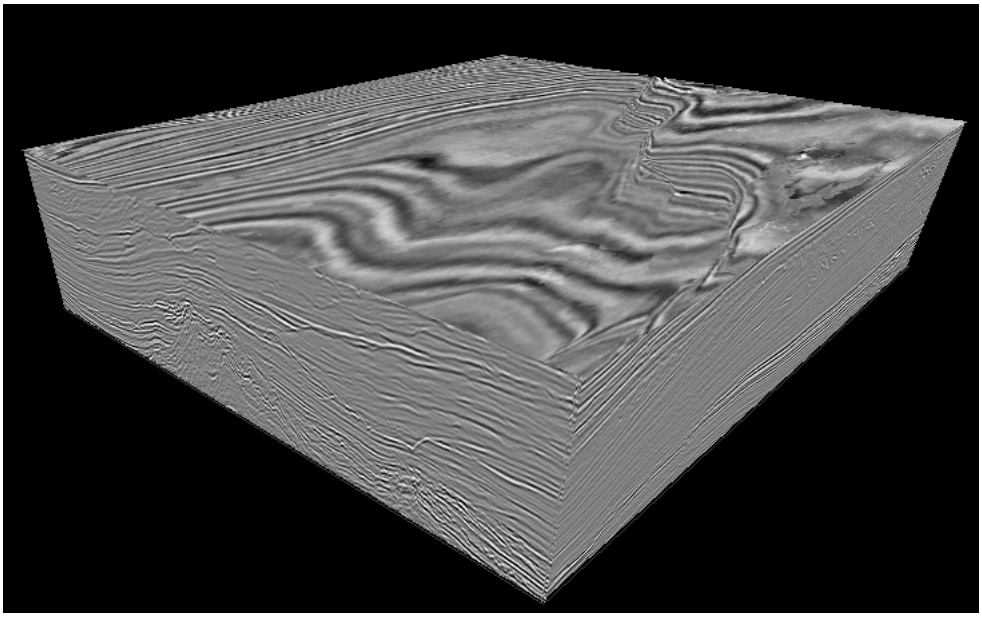

Figure 3.2: 3D view of Parihaka seismic data volume

We annotated the desired structures on 2D slices. Channels are annotated on time slices, and faults are annotated on both inline and crossline slices. First, we need to identify the structure on the VRGeoDemonstrator, with the help of different transfer functions for coloring the raw data. A decision can be made by closely examining the local features from different angles. Then we turn to the corresponding 2D slice, locate the matching structure and draw a one-pixel-wide line within the channel structure. We used a combination of straight lines to fit the underlying faults in order to improve the annotation efficiency. Due to the lack of domain knowledge and experience, "precision over recall" is our annotating principle. Namely, only the dominant faults in which we have confidence are marked in vertical slices. Possible small faults are left unmarked because of uncertainty.

108 timeslices with annotated channels are provided at the beginning. Faults are annotated on 172 inline and 83 crossline slices by the author. Roughly 12 hours are required to annotate faults from 255 slices in one dataset. Examples of channel and fault annotations are shown in Figure 3.3.

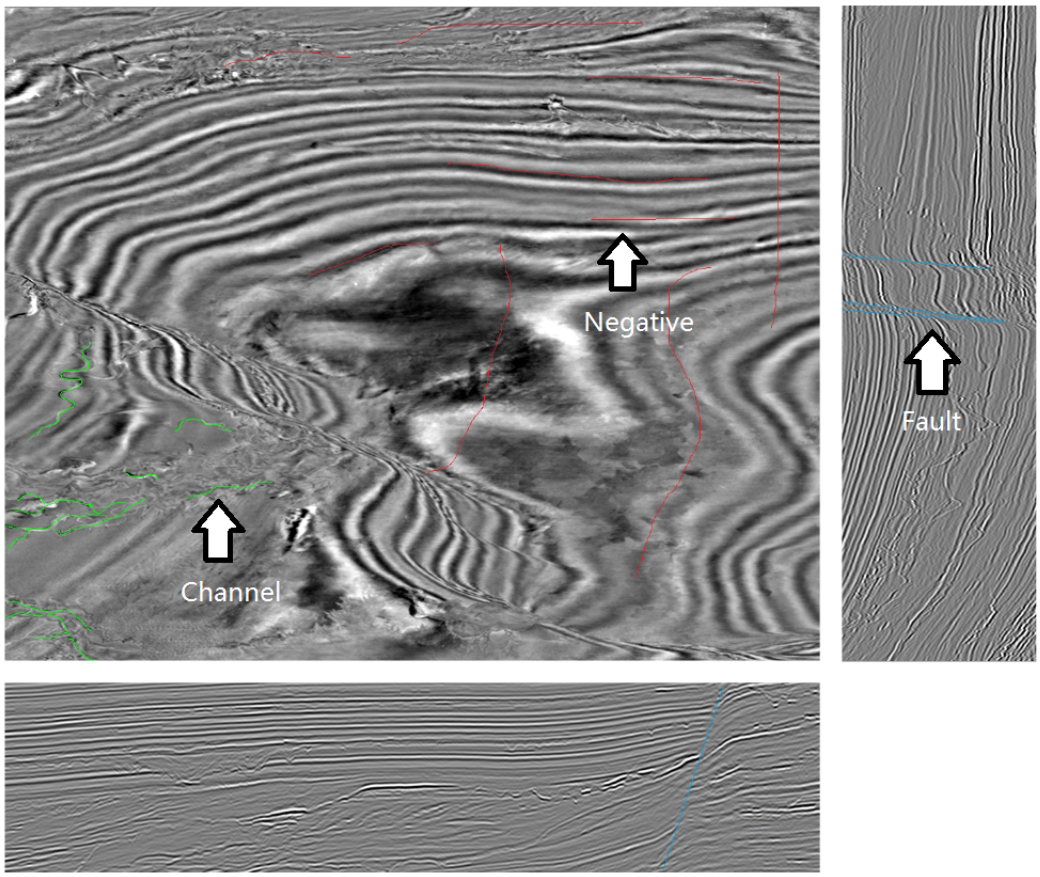

Figure 3.3: Example of our annotation. Channels are annotated on time slices with green pixels, faults are annotated on inline and crossline slices with blue pixels. Red pixels are the original annotation of negative samples.

3.1.2 Dataset Construction

As channels and faults are manually annotated and can be used directly for extracting training patches, we still need negative/background samples to train the model. At the beginning we annotate the negative samples manually by drawing several random lines on the background where no structures are lying, as shown in Figure 3.3. The disadvantage of this annotation is that the background samples are interdependent with very low variance since they are extracted along a line of consecutive pixels. Therefore we chose to generate negative samples by random sampling the area that contains no structures.

Since our annotation for structures is not per-pixel and has fairly weak constraints, simply excluding the annotated pixels might cover the desired structures and cause inaccuracy in the dataset. On the other hand, annotated pixels are scattered around the slice so an overall bounding box might exclude a large part of the background. Therefore we group the annotated pixels using density-based clustering method DBSCAN, and draw bounding boxes for each cluster. The negative pixels are then randomly sampled from areas which are not covered by those bounding boxes.

Each annotated pixel can be treated as a voxel in a 3D data volume. Therefore, for each annotated voxel, we can extract three orthogonal patches based on different views in 3D

space. To implement it, we first combine the 2D coordinates of the annotated pixel and its slice number to generate the 3D coordinates, then we project the 3D coordinates into different views in order to extract patches from 2D slices, the process is described as follows:

$$Coordinate2D(x, y), View(z) \rightarrow Coordinate3D(x, y, z) \rightarrow Coordinate2D(y, z), View(x) \tag{3.1}$$

We construct datasets in accordance with the views, e.g. for the time slice dataset, we project the voxel coordinates to time view and extract patches.

To sum up, given a seismic volume and a predefined patch size, three datasets from different views can be generated. In each dataset, there are three classes of training patches— channel, fault and negative. Figure 3.4 present examples of training patches from three views with size 32×32.

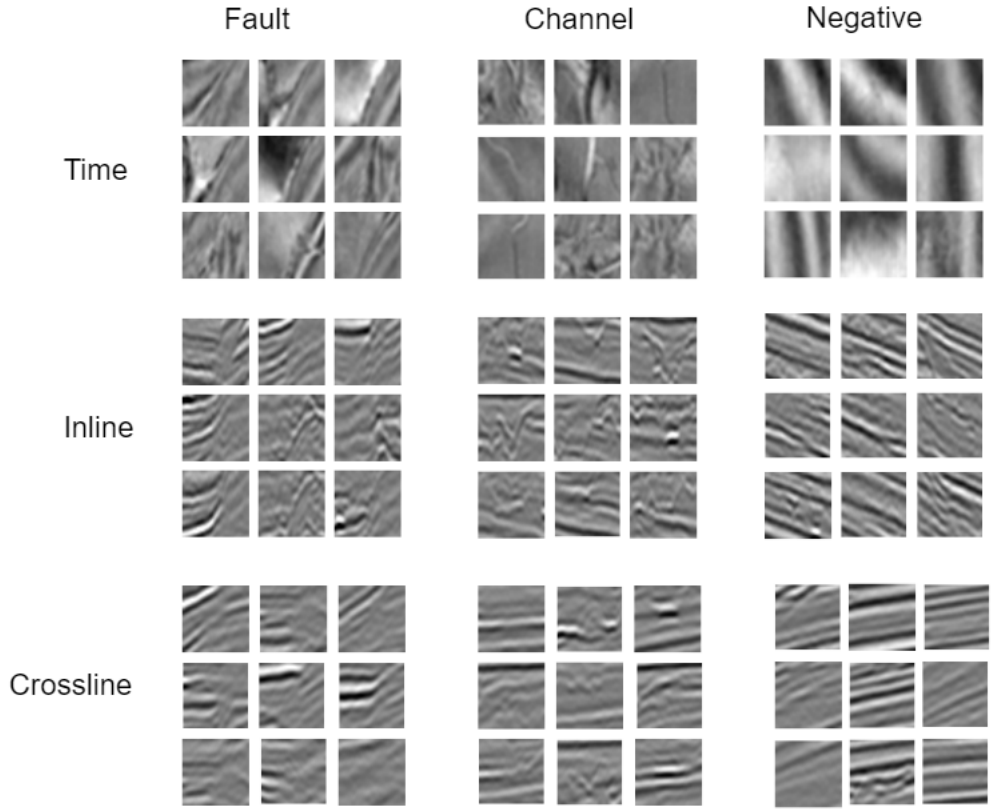

Figure 3.4: Each row shows the 32×32 training patches from different views. Each column shows the category of the patches.

3.2 Architecture

We have experimented with the existed architectures including LeNet, AlexNet, VGGNet and Residual Network (ResNet), which is the state-of-art model in 2016 proposed by He et al. [2016]. First the input size was kept as the in original papers and all output layers have three neurons instead of 1000. The comparison of these CNNs are listed in Table 3.1.

The main problem of AlexNet and VGGNet is that the input patch of 224×224 is way too

large for our segmentation task. Such big patch size not only introduces much redundancy to the dataset (due to consecutive annotated pixels, we need to make a compromise between less samples or more overlapping between patches, and this problem is even more severe when the patch size is big), but also lead to a very coarse segmentation result, which is hardly interpretable. Another problem of these two architectures is that they contains way too many free parameters to train, considering our dataset size of around 300,000 samples.

A ResNet using 32×32 patches as input shows good results during the training session. However in practice, it takes around three hours to train a single epoch so in this thesis we did not use ResNet as the basic architecture for training.

Table 3.1: Comparison of different convolutional neural networks

Year	Model	Input	Layers	Kernel	number of parameters
1998	LeNet	32×32	6	5×5	0.43 million
2012	AlexNet	224×224	11	11×11	60 million
2014	VGG Net	224×224	21	3×3	133 million
2015	ResNet	32×32	32	3×3	0.46 million

After experimenting on different architectures and various hyper-parameters, we decided on the following architectures for further experimentation.

3.2.1 2D Convolutional Neural Networks

The configuration of our 2D CNN including CNN7 and CNN10 as specified in Table 3.2 and Table 3.3 is mainly based on the idea of VGG Net. The input patch is first filtered by convolutional layer CONV1 with 32 3×3 (spatially) kernels. CONV3 and CONV4 are stacked together in order to have bigger receptive field and more non-linearity. Both layers contains 64 3×3 convolution kernels, so equivalently they form a receptive field of 5×5 as discussed in Chapter 2. CONV6 and CONV7 have the same configuration, only with double the amount of convolution kernels—each layer contains 128 kernels.

The stride of all convolution kernels is set to 1 with 1 pixel zero-padding, so that the patch remains unchanged spatially after convolution. The non-linearity is introduced by ReLU functions at the end of every CONV layer.

Each convolution block is followed by a pooling layer. All POOL layers perform spatial down-sampling by 2×2 max-pooling kernel with stride of 2, therefore halving the resolution of the patch each time after pooling.

After the last POOL layer, the output feature map is flattened and passed to the fully connected layers. We performed experiments on two different configurations with two FC layers and three FC layers. The number of neurons in each FC layer are decided later in the experiments (except for last FC layer which has three neurons represents three classes). In the last FC layer, a $softmax$ function is applied so that the output result represents probabilities of each class.

Table 3.2: CNN7 Model

Layer	Kernel	Size 1	Size 2
Input		32×32	64×64
CONV1	32@$3 \times 3 \times 1$; stride:1; padding:1	$32 \times 32 \times 32$	$64 \times 64 \times 32$
POOL2	2×2; stride:2	$16 \times 16 \times 32$	$32 \times 32 \times 32$
CONV3	64@$3 \times 3 \times 32$; stride:1; padding:1	$16 \times 16 \times 64$	$32 \times 32 \times 64$
CONV4	64@$3 \times 3 \times 64$; stride:1;padding:1	$16 \times 16 \times 64$	$32 \times 32 \times 64$
POOL5	2×2; stride:2	$8 \times 8 \times 64$	$16 \times 16 \times 64$
FC6		512	512
FC7		3	3
Number of parameters		2.1 million	8.4 million

Table 3.3: CNN10 Model

Layer	Kernel	Size 1	Size 2
Input		32×32	64×64
CONV1	32@$3 \times 3 \times 1$; stride:1; padding:1	$32 \times 32 \times 32$	$64 \times 64 \times 32$
POOL2	2×2; stride:2	$16 \times 16 \times 32$	$32 \times 32 \times 32$
CONV3	64@$3 \times 3 \times 32$; stride:1; padding:1	$16 \times 16 \times 64$	$32 \times 32 \times 64$
CONV4	64@$3 \times 3 \times 64$; stride:1;padding:1	$16 \times 16 \times 64$	$32 \times 32 \times 64$
POOL5	2×2; stride:2	$8 \times 8 \times 64$	$16 \times 16 \times 64$
CONV6	128@$3 \times 3 \times 64$; stride:1; padding:1	$8 \times 8 \times 128$	$16 \times 16 \times 128$
CONV7	128@$3 \times 3 \times 128$; stride:1;padding:1	$8 \times 8 \times 128$	$16 \times 16 \times 128$
POOL8	2×2; stride:2	$4 \times 4 \times 128$	$8 \times 8 \times 128$
FC9		512	512
FC10		3	3
Number of parameters		1.05 million	4.2 million

3.2.2 3D Convolutional Neural Networks

The most commonly used CNNs are limited to learn features from 2D images. For 3D inputs such as 3D shapes rendered as point clouds and videos considered as continuous frames of 2D images, 3D CNNs are developed so that spatial and temporal features can be learned compactly. Ji et al. [2013] and Tran et al. [2015] successfully applied 3D CNN to perform human action and dynamic scene recognition from video streams.

3D CNNs consist of 3D convolution layers and 3D pooling layers. In convolutional layers, each convolution kernel of size $P \times Q \times R$ performs covolution operation in both spatial dimension and depth dimension as shown in Figure 3.5. The output of convolution layer is a 4D tensor formed by stacking up M 3D feature maps generated by M kernels.

Formally, the activation of neuron x at position (i, j, k) on the m_{th} feature map in the l_{th} layer is given by:

$$x_{ijk}^{lm} = \sigma(b^{lm} + \sum_{m} \sum_{p=0}^{P-1} \sum_{q=0}^{Q-1} \sum_{r=0}^{R-1} w_{pqr}^{lm} x_{(i+p)(j+q)(k+r)}^{(l-1)m}) \tag{3.2}$$

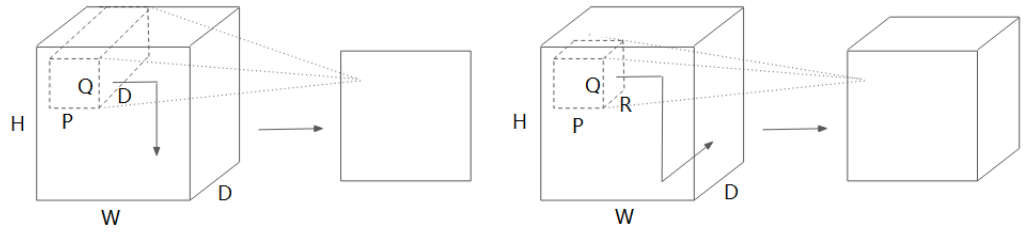

Figure 3.5: Comparison of 2D and 3D convolution. Left: 2D convolution with $W \times H \times D$ input and a single convolution kernel of size $P \times Q \times D$. The convolution is performed spatially and generates a 2D feature map. Right: 3D convolution with $W \times H \times D$ input and a single convolution kernel of size $P \times Q \times R$, where $R < D$. The 3D convolution is performed along spatial dimension and depth dimension and generate a 3D feature map.

Similarly, pooling is also performed in a 3D window, therefore shrinks the size of input both spatially and temporally, as shown in Figure 3.6. For example, a pooling kernel of size $2 \times 2 \times 2$ with a stride of 2 will shrink the $W \times H \times D$ input to $\frac{W}{2} \times \frac{H}{2} \times \frac{D}{2}$.

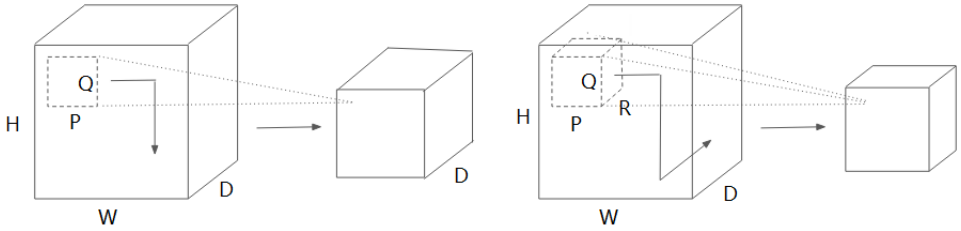

Figure 3.6: Comparison of 2D and 3D pooling. Left: 2D spatial pooling with $W \times H \times D$ input. A 2D pooling kernel of size $P \times Q$ is applied to each depth-slice independently, therefore the output preserves the depth D. Right: 3D pooling with $W \times H \times D$ input. A 3D pooling kernel of size $P \times Q \times R$, where $R < D$, is applied to the input volume. The resolution of the output is reduced both along the spatial dimension and depth dimension.

In the last section, 2D CNNs are used to extract features from 2D patches or 3D patches which stack up a few continuous 2D patches. However, for the task of extracting 3D structures from seismic data volume, it is desirable to capture 3D features directly. Therefore in this section, we proposed a 3D CNN architecture to recognize geophysical structures from an input seismic cube.

To control the number of parameters, three pooling layers are used in the model in order to reduce the size of each feature map from $32 \times 32 \times 32$ to $4 \times 4 \times 4$. The configuration of the 3D CNN is specified in Table 3.4.

Table 3.4: 3D CNN Model

Layer	Kernel	Size
Input		$32 \times 32 \times 32$
CONV1	$32@3 \times 3 \times 3 \times 1$; stride:1; padding:1	$32 \times 32 \times 32 \times 32$
POOL2	$2 \times 2 \times 2$; stride:2	$16 \times 16 \times 16 \times 32$
CONV3	$64@3 \times 3 \times 3 \times 32$; stride:1; padding:1	$16 \times 16 \times 16 \times 64$
CONV4	$64@3 \times 3 \times 3 \times 64$; stride:1;padding:1	$16 \times 16 \times 16 \times 64$
POOL5	$2 \times 2 \times 2$; stride:2	$8 \times 8 \times 8 \times 64$
CONV6	$128@3 \times 3 \times 3 \times 64$; stride:1; padding:1	$8 \times 8 \times 8 \times 128$
CONV7	$128@3 \times 3 \times 3 \times 128$; stride:1;padding:1	$8 \times 8 \times 8 \times 128$
POOL8	$2 \times 2 \times 2$; stride:2	$4 \times 4 \times 4 \times 128$
FC9		500
FC10		500
FC11		3
Number of parameters		5.17 million

3.3 Training

We experimented with the above mentioned networks with slightly different settings. The patch size was set to 32×32 and 64×64 for the seismic volume of resolution $293 \times 1126 \times 976$. We chose such patch sizes that are big enough to capture short-range features, and small enough to preserve the pixel location information. Comparison of different patch sizes is shown in Figure 3.7.

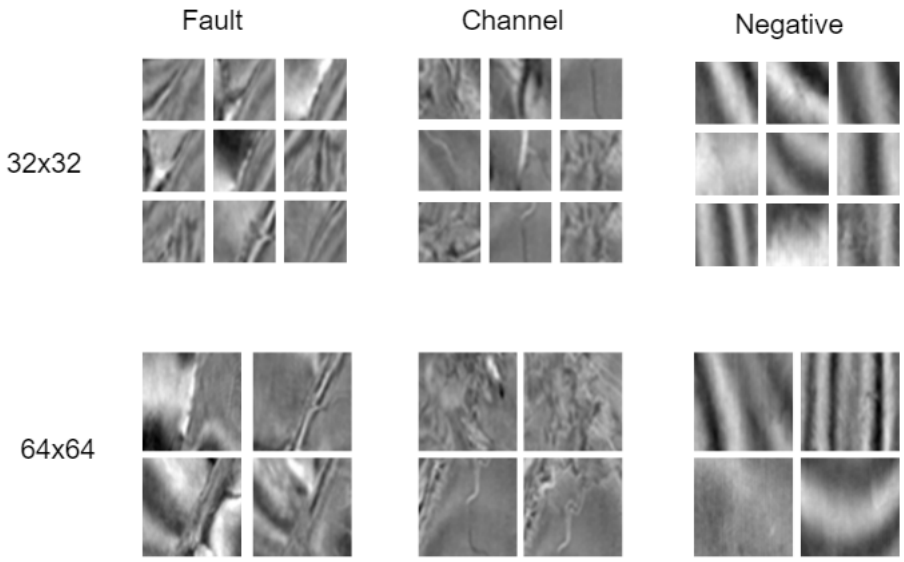

Figure 3.7: Comparison of patch size 32×32 and 64×64. Sizes smaller than 32 are hardly recognizable since not enough surrounding information is provided. Sizes bigger than 64 contains more long-range information which bring only little help for identifying long and narrow structures like channels and faults, but make the object location less obvious.

We also trained the 2D CNNs with 3D volumes with small depth. Namely, we extract a

data volume centering at the annotated pixel by stacking up a small number of continuous 2D patches. We mainly experimented with $32 \times 32 \times 5$ volume size. The idea of 3D volumes is inspired by the manual annotation process. While identifying underground structures, we examine a few consecutive slices, since a structure generally shows continuity of itself in neighboring slices, unlike noise. A Depth of 5 is set according to our annotating experience.

With 3D volume as input, the 2D CNNs need to be slightly adjusted. The only difference is that in the first convolutional layer, the kernel size needs to be changed from $3 \times 3 \times 1$ to $3 \times 3 \times 5$. Note that although the input is a 3D volume, the kernel is only performing convolution and pooling operations along spatial dimensions in 2D CNNs.

Experiments on 3D input volumes with larger depth are also carried out according to the advice of geophysicists. To encourage symmetry, we used the volume size of $32 \times 32 \times 32$ to train the proposed 3D CNNs.

Training samples and validation samples are drawn from different slices in Parihaka dataset instead of randomly splitting the whole dataset, in order to reduce the relevance between training and validation set. The ratio of training set to validation set is approximately $4 : 1$.

3.3.1 Data Preprocessing

Before passing the data to the neural network, preprocessing is an essential step. In experiments the following pre-processing methods are used:

- Mean subtraction Feature-wise mean subtraction is a common data pre-processing method. For images we consider each pixel as an individual feature. For simplicity reason, it is common to subtract a single value from all pixels. In our experiments, the mean matrix/ tensor are calculated over all training patches, then subtracted from each patch.

Mean subtraction showed great importance in the training process. Without this step, the training accuracy reached only 60% after slow convergance. After subtracting the mean, the network converges fast and could achieve more than 95% accuracy.

- Normalization After mean-subtraction, it is also common to perform normalization in which we simply scale the pixel intensities between $[-1, 1]$.

3.3.2 Initialization

Before the training starts, all parameters need to be properly initialized. By initializing all the weights with the same number, we will face a symmetry problem. Namely, when two neurons are connected to the same input, the back-propagation algorithm will always update their parameters in the same way. To break the symmetry, we need to randomly initialize the parameters.

Typically the weights are initialized by randomly drawing small values from gaussian or uniform distribution, and the biases are initialized with zero. Since with the increasing number of input neurons n, the variance of the output will also grow. The factor $\frac{1}{\sqrt{n}}$ is introduced so that the output's variance can be scaled to 1. In experiments, we initialized weights in fully connected layers with random number divided by $\sqrt{n}$, as recommended by

Karpathy et al. [2016]. Weights in convolutional layers are initialized from normal truncated distribution, in which the values two standard deviations away from mean are discarded.

3.3.3 Batch Normalization

To reduce the problem of carefully initialize the network, we experimented with a recently developed technique called batch normalization by Ioffe and Szegedy [2015]. The batch normalization method proposed to add normalization layers in the model architecture and normalize the activations for each training mini-batch. For the input $x = [x^1...x^d]$, each feature dimension $x^d = [x_1...x_m]$ across the mini-batch of size m is forced by the algorithm explicitly to take a unit gaussian distribution, which has a mean of 0 and a variance of 1:

$$\widehat{x}_m = \frac{x_m - \mu[x_m]}{\sqrt{Var[x_m]}} \tag{3.3}$$

where $\mu[x^k]$ and $Var[x^k]$ are the mean and variance for each feature across the mini-batch.

To make sure that the normalization can represent an identity transformation so that the following non-linear function is not constrained by the normalized output, two trainable parameters γ_m and β_m are introduced to scale and shift the output as follows:

$$y_m = \gamma_m \dot{\widehat{x}}_m + \beta_m \tag{3.4}$$

Batch normalization allows higher learning rate and less careful initialization. Ioffe and Szegedy [2015] also proved that batch normalization makes the network converge faster and achieve higher accuracy.

In our experiments, we add a batch normalization layer after every convolution layer and fully connected layer for the 3D CNN architecture.

3.3.4 Optimization Algorithm

Except for stochastic gradient descent, there are many other optimization algorithms. Different optimization algorithms have different dynamics while performing parameter updates, as shown in Figure 3.8.

Optimizers including SGD, Momentum, RMSprop and Adam were compared in our experiments, where Adam appears to be the most effective optimizer.

We started searching for a proper learning rate from 0.1 and every time decrease it by the factor of 10. The learning rate is finally set to 10^{-4} which gives us the best training loss curve and result.

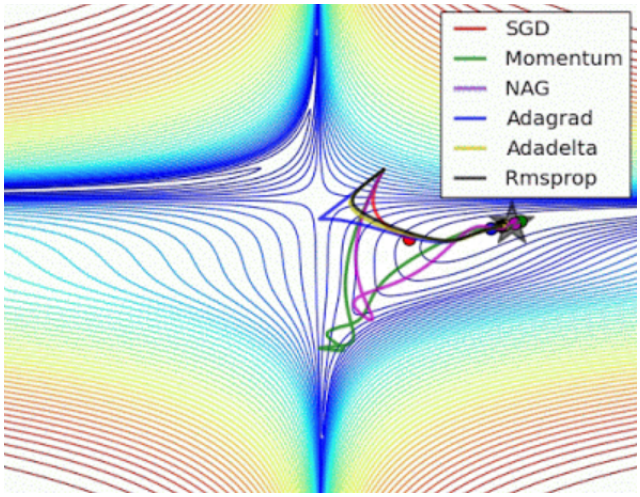

Figure 3.8: Animation from Alec Radford visualizing optimization process on low dimenstional problems. Algorithms including SGD, Momentum, NAG, Adagrad, Adadelta and RMSprop are shown by curves in different colors, and the destination is shown by the grey star. From this figure we can see, SGD converges very slow, Momentum and NAG shoot off and have unstable dynamics. RMSprop and Adadelta works as accelerated SGD and are more stable than other algorithms.

3.3.5 Infrastructure Details

All experiments are performed on a NVIDIA deep learning DevBox with four GeForce Titan X GPU with 12 GB memory. All implementation are based on TensorFlow and TensorFlow based APIs such as Tensorlayer and TFLearn.

For the 2D CNN architecture, the implementation is based on TFLearn. The model loads all training samples into memory. Data processing is performed on the CPU and tensor computation are performed on the GPU. It takes approximately three to four hours to finish 30 training epochs with the above mentioned 2D CNN configurations.

For the 3D CNN architecture, the training dataset is too big to be loaded into memory as a whole. Therefore we used TFRecords to load input data in mini-batches. The implementation is based on Tensorlayer due to its better support for TFRecord. 3D CNNs are harder to train compared to 2D CNNs. Every epoch takes roughly three hours to train on a single GPU. Therefore in the experiment section we are only able to provide one graph showing the performance of 3D CNN due to the limited amount of time.

3.4 Testing

At test time, a trained model is applied to a given test slice from the corresponding view. Since it is common in seismic data volumes that some of the slices are corrupted with blank spaces as seen in Figure 5.11, we first pre-process the image with a small window calculating the variance of neighboring pixels, and eliminate the pixels whose surrounding has variance less than 0.01. Afterwards, patches are drawn for all remaining pixels in the test slice and passed to the trained CNN model. The model calculate the probability of each class and

the class with the highest probability is assigned to the pixel. The testing process is fast compared to the training since it only needs to compute the forward pass. For a slice with one million pixels, the testing process takes around 10 minutes on a single Intel CPU.

3.4.1 Bottom-up Segmentation

With domain knowledge, we know that the desired structure normally presents discontinuity in the rock layers. In other words, large uniform regions are less likely to be the structures we are searching for.

By generating bottom-up segmentation such as super-pixels on top of the original image, we can observe that the desired structures, especially faults are mainly captured by the super-pixel boundaries, while the big blobs within the super-pixel are mostly noise. Therefore, we combined this segmentation with the prediction of CNN in order to refine the result.

There are many different super-pixel generating algorithm. We applied SLIC (Simple Linear Iterative Clustering) by Achanta et al. [2012]. The algorithm segments the image by k-means clustering the pixels. The distance measure consists of normalized color proximity and space proximity. A segmentation on seismic time-slice using SLIC is given by Figure 5.17.

3.4.2 Ensemble Different Models

Normally different models have different output for the same input data. Averaging their output is a common and effective method to improve the performance. Most tasks ensemble models that are trained with different architectures but the same training data. In this task, ensembling is especially powerful because for each pixel, we can ensemble models trained from different datasets, namely datasets from three views as mentioned before, so that we can make full use of the 3D information around the pixel.

This process also correlates with manual seismic data interpretation, as human experts will examine the slices from all three views to decide whether it is a desired structure at the slice intersection.

We have experimented on datasets with different patch sizes including $32 \times 32 \times 5$, 64×64 and 32×32. For each size we have trained three models from time view, inline view and crossline view. Since inline and crossline views are both vertical slices with different view angles, we consider these two view to be equivalent. Therefore all the models and the test data they applied to are listed in Table 3.5:

Table 3.5: Matching views of training and testing data

	View of training data	View of testing data
1	time	time
2	inline	inline
3	inline	crossline
4	crossline	inline
5	crossline	crossline

3.4.3 Post-processing

CNNs classify all the pixels independently regardless of the semantics of its neighboring pixels. This usually leads to an unsmooth result. For post-processing, we applied a density-based clustering method to group the predicted channel and fault structures separately, while eliminating outliers whose surrounding has low density of pixels that belongs to the same class.

In practice we used the DBSCAN (Density-based spatial clustering of applications with noise) algorithm by Ester et al. [1996]. The algorithm iteratively assigns points to be core points when at least $minPts$ points are within a radius ϵ surround it. The points that are neither core points nor density-reachable from core points are considered to be noise. For example, if a pixel is labeled as channel by the CNN but none of its 15 neighbors are classified as channel but as background, then the algorithm will decide that this pixel belongs to noise and discard its label. Furthermore, to encourage continuity within detected structures, we assigned all pixels within the density reachable range of core points to have the same label with its core point.

By applying density based clustering, the pixel-wise classification result can be smoothed and a big part of false positives can be removed.

4 Regularization

The success of deep learning is driven by massive amounts of data. Deep neural networks contain millions of free parameters, therefore enough annotated training data has to be provided in order to achieve good performance and avoid overfitting.

Overfitting is caused by excessive model capacity relative to the amount of training samples. An overfitted model memorizes the training data including random error or noise, instead of learning useful features for prediction. Therefore, the model can achieve good accuracy on the training set, but has comparatively worse performance on unseen data.

Due to the limit amount of data we obtained, overfitting becomes a major issue in our task. To overcome the overfitting problem, we considered solutions from mainly two aspects:

1. To generate more data based on the current available dataset. Here we applied data augmentation techniques with respect to the properties of seismic data.
2. To deploy regularizers such as dropout, L2 normalization and early stopping in our CNN architectures.

4.1 Data Augmentation

Data augmentation is a simple but highly effective technique to enhance the performance of deep CNNs, especially for small datasets which suffer from overfitting. The basic idea of data augmentation is to apply certain transformations to the original training image and pass the transformed image to the network. Although the dataset size remains unchanged in a single epoch, throughout the whole training process, the network is trained on s different transformed dataset in each epoch. Therefore, data augmentation techniques can effectively enlarge the training set and is able to reduce overfitting.

There are several operations to perform data augmentation. In this thesis we mainly experimented with flipping, rotation and scale augmentation which are chosen based on seismic data properties.

- Flipping Flipping is a common image transformation method. For data augmentation especially left-right flipping, since in most datasets the natural up-down relationship is expected to be preserved for object recognition.

In our experiment we applied random left-right flipping when training on vertical slices. On time slices we added both left-right and up-down flipping since on the horizontal slices there is no up-down relationship. Examples of flipping training patches are shown in Figure 4.1.

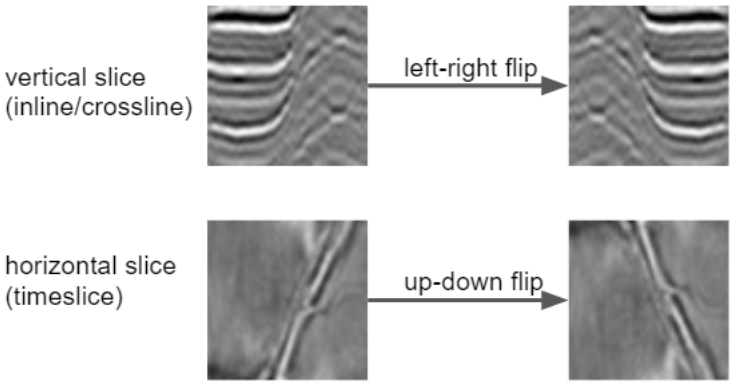

Figure 4.1: For vertical slices, only random left-right flipping is applied. For horizontal slices, both left-right and up-down flipping is performed.

- Rotation Rotation is another basic transformation method. We applied random rotation with maximum 360 degree to the timeslice dataset, since the features on the horizontal slice are rotation-invariant. On vertical slices no rotation is performed.

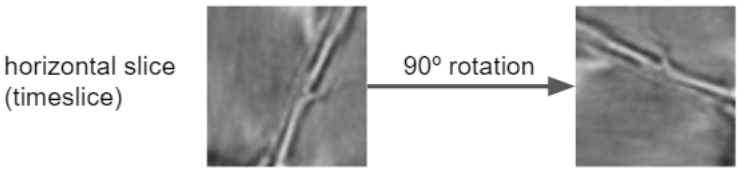

Figure 4.2: Rotation applied on horizontal slices. In our experiments random angles from 0 to 360 degrees are used for each image.

- Scale augmentation The motivation of scale augmentation is to train the network to recognize objects at different scales. Simonyan and Zisserman [2014] applied scale augmentation in their training process, which they call multi-scale training. For each training image, they randomly sampled a scale S from range $[S_{min}, S_{max}]$ and resized the original image to let its smallest side be S. On the resized image, they then cropped 224×224 patches according to some pattern (e.g. four crops on the corners and one in the center) and used these patches as input to the network.

In our experiments, we perform scale augmentation slightly differently. For each annotated pixel P, we randomly sampled width s and height t from $[s_{min}, s_{max}]$ and $[t_{min}, t_{max}]$ and extract the training patch with size $s \times t$ around pixel P. We then resize the patch to our training set scale and use it as input of the network. If a training set scale of T is assumed, the final patches will have the dimension $T \times T$.

Figure 4.3: Examples of scale augmentation. Patches are extracted with random width and height between $[64, 128]$ and rescaled to size 64×64.

When we train 2D convolutional neural networks with 3D volumes of size $T \times T \times d$, where d is the depth of the volume, we only apply scale jittering spatially and keep d the same.

When we train 3D convolutional neural networks with 3D volumes of size $T \times T \times T$, we also randomly sample d from range $[d_{min}, d_{max}]$ and extract volumes of size $w \times h \times d$ then resize to scale $T \times T \times T$.

4.2 L2 Regularization

L2 regularization, also known as weight decay, is one of the most common regularization strategies for a lot of machine learning algorithms. L2 regularization a kind of parameter norm penalty, in which a regularization term $\frac{\lambda}{2} \|w\|^2$ is added to the loss function $L(\theta)$:

$$\tilde{L}(\theta) = L(\theta) + \frac{\lambda}{2} \|w\|^2 \tag{4.1}$$

where $\lambda \in \Re$ and $\lambda > 0$.

With the new loss function, the gradient over bias b remains unchanged, the gradient over w is given by:

$$\frac{\partial \tilde{L}}{\partial w} = \frac{\partial L}{\partial w} + \lambda w \tag{4.2}$$

Therefore we update the weights as follows:

$$w \leftarrow w - \eta(\frac{\partial L}{\partial w} + \lambda w) \tag{4.3}$$

$$w \leftarrow (1 - \eta\lambda)w - \eta \frac{\partial L}{\partial w} \tag{4.4}$$

By comparing this to Eq.2.4 we can find that with the addition of the regularization term, the weight is forced to shrink by a factor λ.

One intuition is that when the model overfits, it tends to become more complicated and learns larger weights, therefore the regularization strategy restrains the model to learn small and diffuse weight vector instead of a peaky one, in order to prevent overfitting.

4.3 Early Stopping

During training, we noticed that although the training accuracy has been rising all the time, the validation accuracy starts to slowly decrease at some point, as seen in Figure 4.4. This

is a sign that the model is overfitting the data. Generally, this situation is observed reliably during the training process of neural networks.

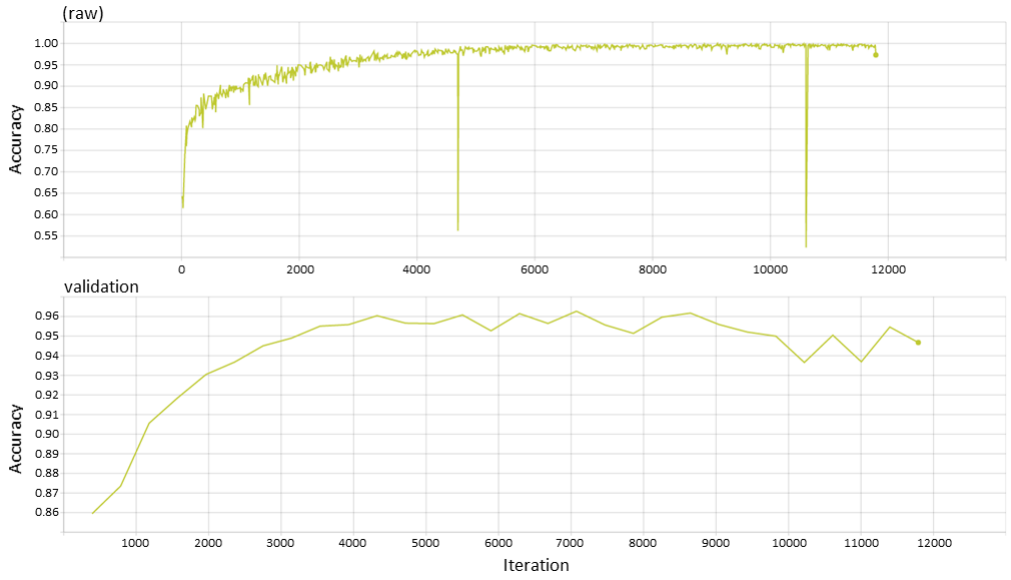

Figure 4.4: Top: Training accuracy keeps increasing throughout the whole training process. Down: Validation accuracy start to drop from around 9000_{th} iteration.

By stopping the training process at the point where validation accuracy stops increasing for some time, we can obtain a better generalized model. Even though the validation accuracy might rise again and finally reach a higher value, it could most probably end up in a local minimum, which would result in worse performance on test data.

Early stopping is a very straight-forward and simple strategy since only one hyper-parameter — the number of training epochs — is involved. It is also a commonly used regularization strategy due to its effectiveness in experiments.

In our experiment, we save the model parameters after every training epoch. If we observe that the validation accuracy is dropping continuously for certain epochs, we stop training and take the model with the best validation performance as the final model.

4.4 Dropout

Compared to our limited number of training samples, we have a large amount of free parameters to train, especially in the fully connected layers. Therefore, we experimented with a new regularization method designed for deep neural networks — dropout.

The key idea of dropout is to disable a random set of neurons during the training process. As explained by Srivastava et al. [2014], this can prevent the neurons from co-adapting too much. Eventually this technique can be seen as an ensemble of multiple sub-networks, as illustrated in Figure 4.5.

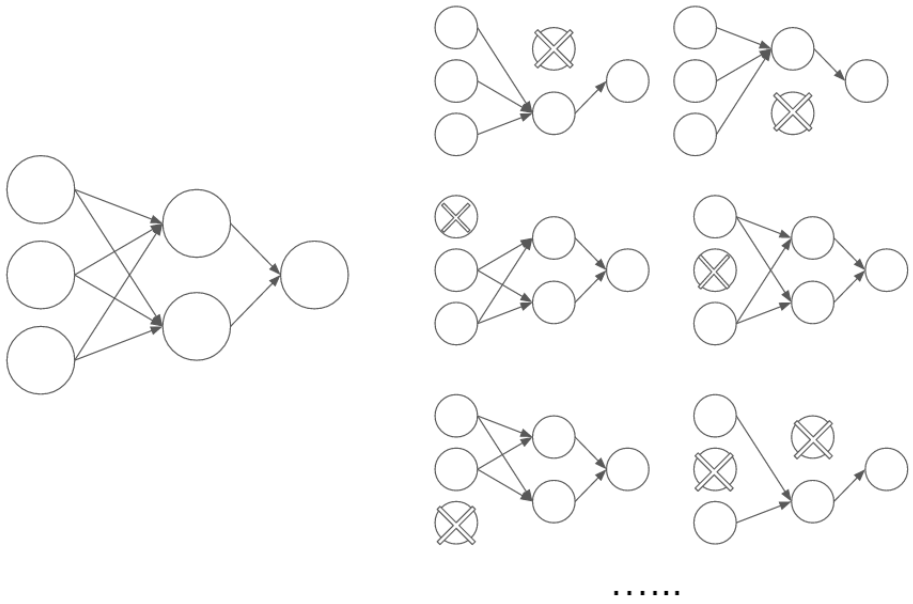

Figure 4.5: Left: the original underlaying network. Right: subnetworks of the original network where random neurons along with their connections are disabled. In total 2^n subnetworks are available.

For a dropped neuron, its connection weights are frozen during forward and backward pass. And the optimization algorithm only updates weights of the kept neurons. The only hyperparameter dropout introduces is a keep probability p between 0 and 1 indicating the probability of a neuron not being dropped.

Typically, p can be chosen by validation set or simply setting the value to 0.5. When $p = 0.5$, roughly half of the neurons are dropped during each training step. For a fully connected network, a subnetwork only contains half the parameters in each single training step. Intuitively, such subnetworks with smaller capacity are less likely to overfit.

Another advantage of dropout is its high efficiency. As is well known, ensemble of several models will increase the performance most of the time. However, training several independent deep neural networks requires multiple times of runtime or memory, applying ensemble to the test set also costs more time and computation. Dropout addresses this issue by sharing parameters. As illustrated in Figure 4.5, a network with n units can be seen as a collection of 2^n sub-networks with shared weights. Therefore with the same number of parameters, dropout can approximate the effect of training an ensemble of exponentially many networks.

At test time, all the neurons and connections are enabled which approximates the averaging of results from multiple models. An important trick is that the weights have to be scaled by the factor p at test time, as illustrated in Figure 4.6. According to Goodfellow et al. [2016] this approximation has not been proved theoretically, but empirically it works very well.

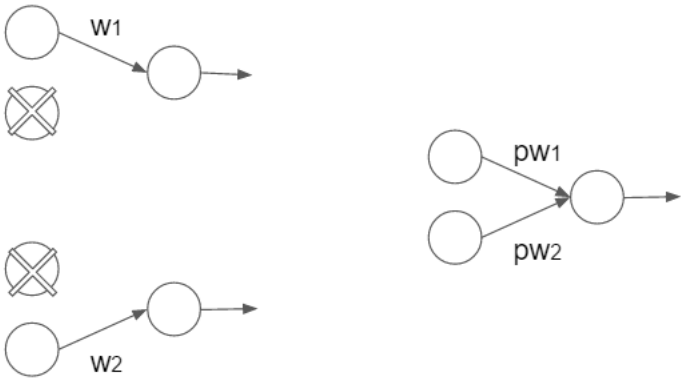

Figure 4.6: An example of weight scaling inference rule. Left: During training, two neurons got dropped at different times. The training forward and backward passes completely ignore the dropped neuron and its connections. The upper subnet only calculates and updates w_1, and keep w_2 the same, while the lower subnet only updates w_2. Right: During testing time, all neurons are enabled, but the weights need to be multiplied by keep probability p in order to give the correct approximation of averaging over all the subnets.

We applied dropout to the fully connected layers in the CNNs with keep probability is set to 0.5. During testing, we observed that applying dropout normally increases the validation accuracy by 1% to 2%.

5 Experiments

We evaluate our method using two different seismic volumes: Parihaka and F3. All models are trained on patches extracted from slices in the Parihaka volume. Testing is performed using one time slice from Parihaka that is far apart from the training slices, and one time slice from F3 volume.

5.1 Parihaka Dataset

All patches are generated from around 400 labeled slices from Parihaka volume. CNN models are trained on around 400k patches, including 163k channel patches, 63k fault patches and 160k negative patches. The labeled data are randomly split into training and validation sets with ratio 4:1. Training batch size is set to 256, the training accuracy is recorded after each batch. After every training epoch is finished, the validation accuracy is computed using the validation set.

5.1.1 Architecture and Patch Size

Experiments are first carried out on different architectures. In Table 5.1 we list the best performance achieved by four different architectures trained on patch size $32 \times 32 \times 5$ and 64×64. The architecture CNN7 and CNN10 were specified in Chapter 3.

Table 5.1: Training and validation results with different architectures

Model	Size	Parameters	Training accuracy(%)	Validation Accuracy (%)
CNN7	$32 \times 32 \times 5$	2.1 million	99.20	**98.83**
CNN7	$64 \times 64 \times 1$	8.4 million	98.85	98.44
CNN10	$32 \times 32 \times 5$	1.1 million	97.26	97.06
CNN10	$64 \times 64 \times 1$	4.2 million	98.50	97.58

In general, CNN7 has the best performance on both scales, while CNN10 is slightly worse although the model contains less parameters. Therefore we decide that CNN7 has the proper model capacity for our task, and later experiments are mainly conducted with CNN7 as specified in Chapter 3.

Figure 5.1 compares the model performances trained on different patch sizes including 32×32, 64×64 and $32 \times 32 \times 5$.

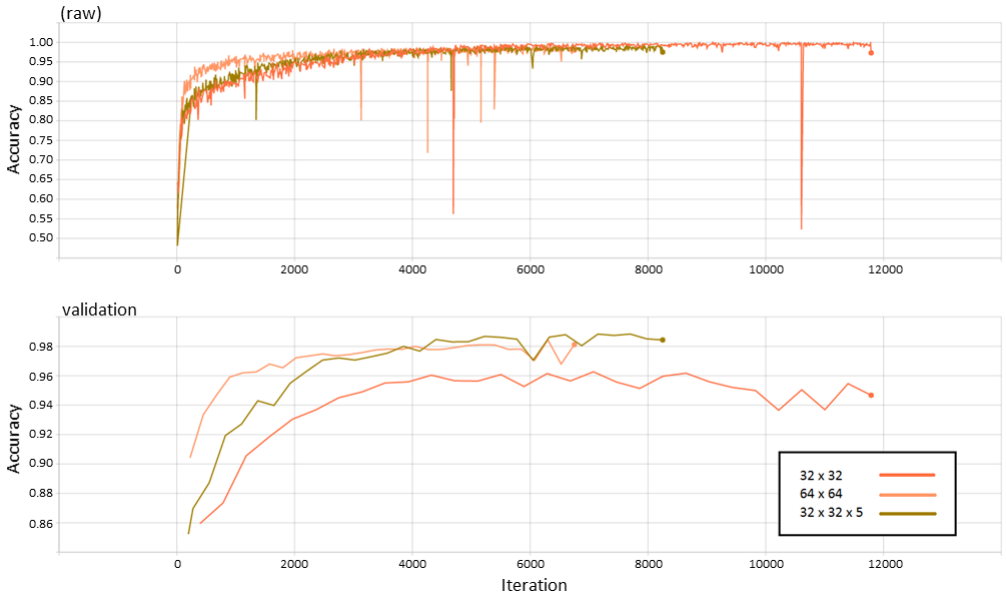

Figure 5.1: Training and validation curves of 2D CNN7 architecture trained on 32×32, 64×64 and $32 \times 32 \times 5$ input patches. All accuracy curves are automatically generated by Tensorboard which is integrated in Tensorflow and TFLearn API, where x axis indicates number of iterations and y axis indicates training and validation accuracy.

Table 5.2: Training and validation results on different scales and depth

Layer	Training accuracy(%)	Validation Accuracy (%)
32×32	99.60	96.27
64×64	98.50	98.44
$32 \times 32 \times 5$	99.20	**98.83**

We observe that during training time, patches with same spatial size 32×32 and $32 \times 32 \times 5$ have similar performance, while CNNs trained on 64×64 patches converge faster than the other two models.

For validation accuracy, models trained on 64×64 and $32 \times 32 \times 5$ patches outperform the model trained on 32×32 patches from the beginning by around 2%. The best validation accuracy is achieved by the model trained on $32 \times 32 \times 5$ with 98.83% as shown in Table 5.7. For further experiments, we chose $32 \times 32 \times 5$ patch size instead of 64×64, since they achieved similar accuracy in classification, and the former has smaller spatial scale which helps with more accurate localization of the desired structure.

5.1.2 Optimizer

Next we experimented with several popular optimization algorithms including SGD, RMSProp, Momentum and Adam using the CNN7 architecture on the $32 \times 32 \times 5$ time slice dataset. The training curves are given below:

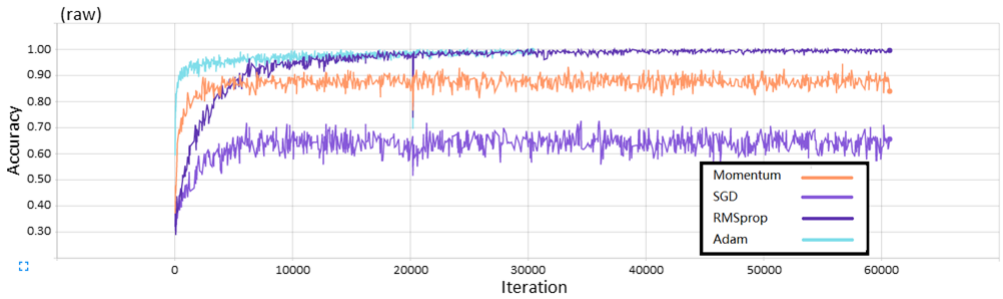

Figure 5.2: Training accuracy of CNN7 architecture with different optimizers including SGD, Momentum, RMSprop and Adam.

Table 5.3: Training accuracy from different optimizers

Optimizer	Training Accuracy(%)
Adam	**98.52**
Momentum	85.58
RMSprop	98.21
SGD	64.52

As seen in Figure 5.2 and Table 5.3, stochastic gradient descent(SGD) converged to a local minimum of only 64.52% accuracy. Momentum has relatively better performance than SGD but accuracy 85.58% is still not high. Adam and RMSprop have reached similar accuracies at the end, while Adam converges faster in the first few epochs. Therefore we mainly use Adam as the optimizer in later experiments.

5.1.3 Regularization

Furthermore, we experimented with different regularization methods as described in Chapter 4. The comparison of plain CNN and CNN with different regularizers are shown in Figure 5.3.

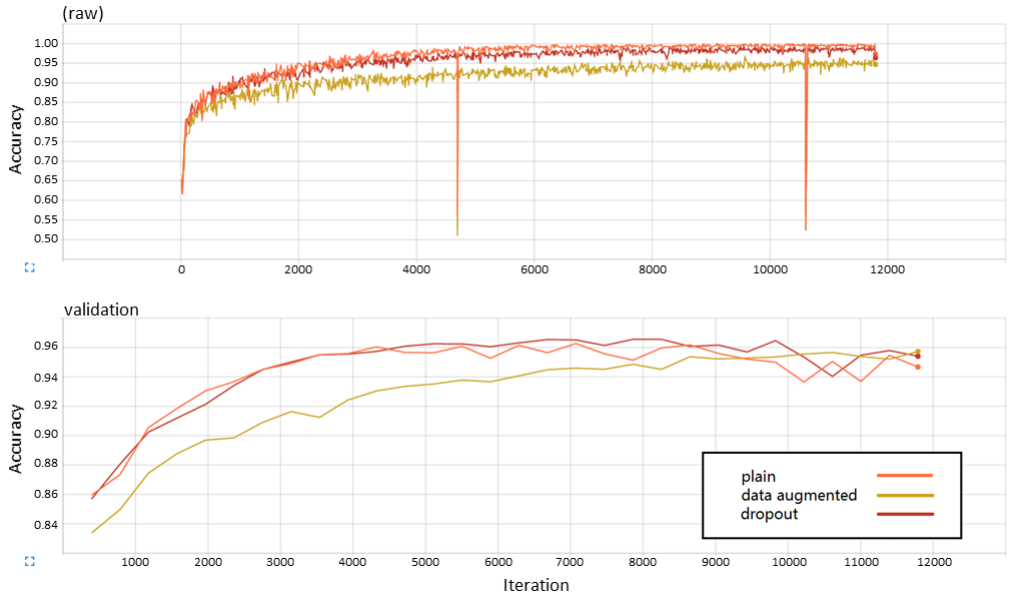

Figure 5.3: Training and validation accuracy of plain CNN7 architecture, CNN7 trained on augmented dataset (including left-right flipping, bottom-up flipping and maximum 360 degree rotation) and CNN7 with dropout in fully connected layers.) All three models are trained on the 32×32 patches for efficiency.

Table 5.4: Training and validation results with different regularizers

Regularizer	Training accuracy(%)	Validation Accuracy (%)
plain	99.60	96.27
dropout	98.50	**96.55**
data augmentation	95.20	95.66

When the input is augmented by random flipping and rotation, training accuracy is reduced by around 3%, which is most probably caused by more variance added to the training data. The validation accuracy is lower than the plain network at the beginning, but surpassed it by the end with constant increasing tendency, while the plain network already shows signs of overfitting via decreasing validation curve. A possible explanation is that when data is augmented by many different transformations, the network's capacity needs to be raised to some extent in accordance with the growing number of training data, and meanwhile more epochs shall be trained to achieve better accuracy. In later experiments we found out that not all data augmentation methods are beneficial for this task. Random-flipping is more helpful than random-rotation and blur, even on horizontal slices where we assumed that random-rotation would not harm the semantics of training samples.

CNNs with dropout applied to the fully connected layers achieved the best validation accuracy. In practice, we normally observe a 1% to 2% increase in validation accuracy.

Applying regularization techniques were found to be useful against overfitting. In later experiments, we combine the regularization techniques including data augmentation, dropout and L2 normalization.

5.1.4 View of training data

For a determined patch size, three training datasets can be generated from time, inline and crossline view, in which inline and crossline are considered equivalent, since they are both vertical views.

We trained models separately on datasets from inline and time views. The model performances are shown in Figure 5.4.

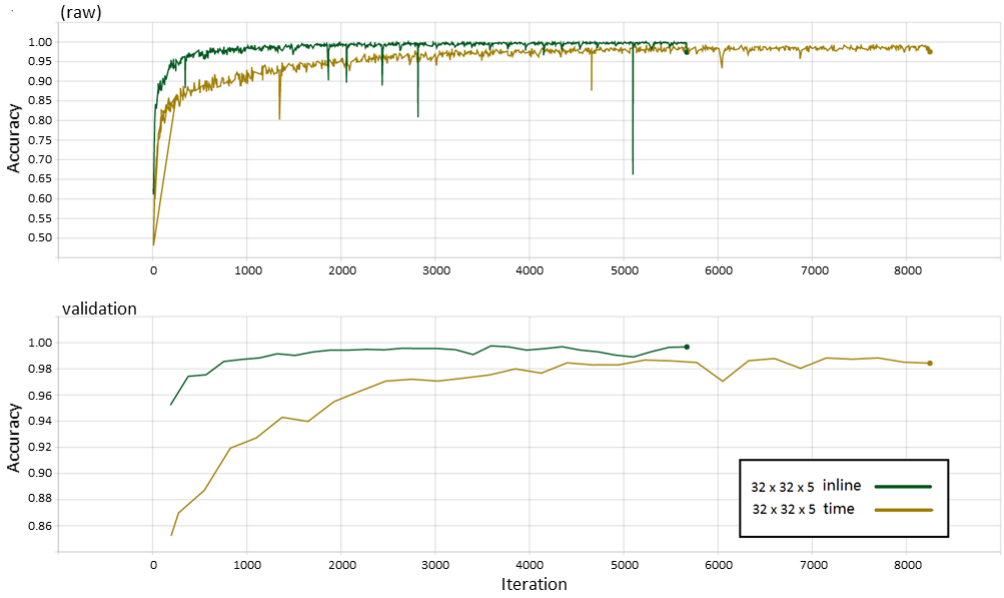

Figure 5.4: Training and validation accuracy curves of 2D CNN7 architecture trained on $32 \times 32 \times 5$ input patches from inline and time view.

Table 5.5: Training and validation results with different views

View	Training accuracy(%)	Validation Accuracy (%)
time	99.20	98.83
inline	99.80	**99.68**

Figure 5.4 and Table 5.5 show that model trained on inline dataset converge faster than the time slice dataset, and beats the latter in both training and validation accuracy. An example test result on an inline slice is given below:

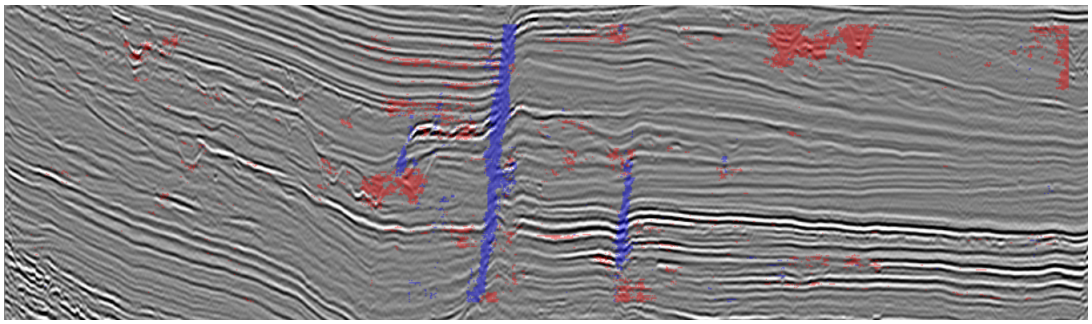

Figure 5.5: Test result on an inline slice from the Parihaka dataset.

From the inline view, faults can be better visualized as a result of relative movements of rock layers. In the above test result, the predicted fault pixels form a clear line along the underlying fault. The channel structures are not as evident from the inline view as can be seen from the time view, but they can still be identified through the sunken part of rock layers. As we can see in the result, the predicted channel pixels are rather noisy compared to the faults.

5.1.5 Test result

We tested the trained CNNs on a time slice from the Parihaka dataset, which is chosen to be as far apart from the training slices as possible. For every single pixel in the slice, a corresponding patch is extracted. The test patches are passed to the trained model and computed through the whole forward pass. The output gives three normalized scores between $[0, 1]$ indicating the probability of the underlying pixel to belong to corresponding classes. Finally, a segmentation map is generated by choosing the class with the highest probability for every pixel.

5.1.5.1 Patch Size

Since we do not have any fully annotated test slice to evaluate the pixel error, we present the following test results in Figure 5.6 showing the performance of models trained on the 32×32 and $32 \times 32 \times 5$ datasets.

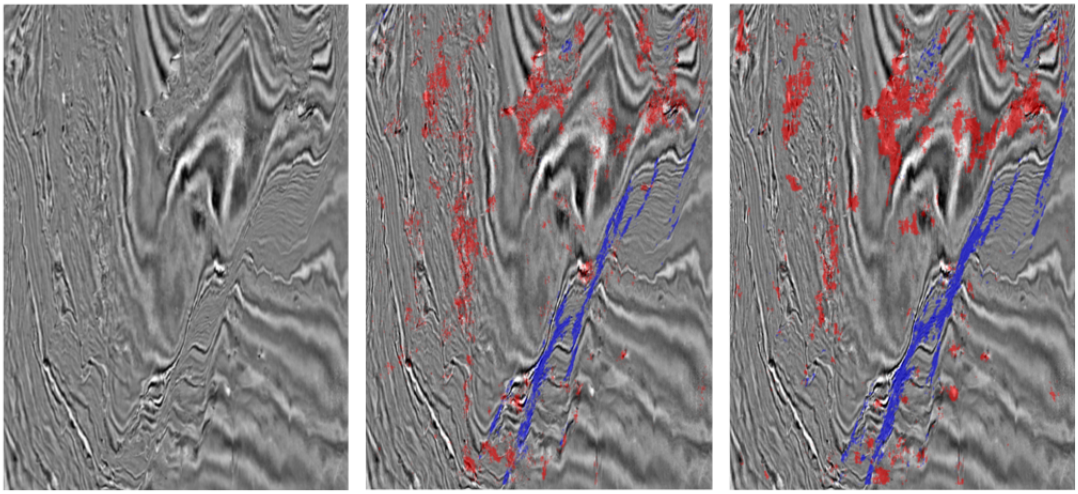

Figure 5.6: Test results from models trained on different patch sizes. Pixels which are predicted to be channel are marked in red, pixels predicted to be fault are marked in blue. (a) Test image, raw time slice 130 from Parihaka dataset. (b) Test result from model trained on 32×32 dataset. (c) Test result from model trained on $32 \times 32 \times 5$ dataset.

According to Figure 5.6, test result (c) computed from $32 \times 32 \times 5$ patches has less noise/false positives than test result (b) computed from 32×32 patches. More true positives can also be seen from the density increment of correctly marked pixels in channel and fault segments.

The test result is consistent with the training performance we obtained. We can infer that the $32 \times 32 \times 5$ model outperforms the 32×32 model because with basically the same number of parameters and number of training data, $32 \times 32 \times 5$ provides more valid surrounding information for classification.

5.1.5.2 View

Centered on each testing pixel, we can generate three patches from different views. Each patch is passed to the model with corresponding view and patch size, the test result obtained from three separate views are shown in Figure 5.7. The patch size here is set to $32 \times 32 \times 5$.

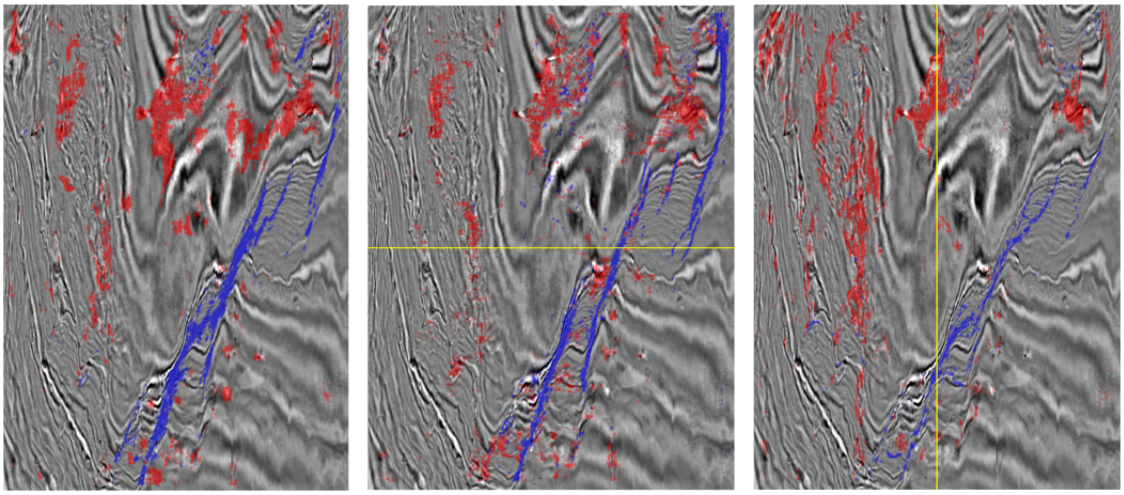

Figure 5.7: Test result from three orthogonal views.

As can be seen from the above figure, the model from horizontal time view yields more consistent yet coarser segmentation on the test time slice, while models from the two vertical views gives more refined result for faults, and noisy but more detailed results for channel. We can also observe that, the result of the vertical view is affected by its own orientation and the structure's trend. Apparently, when the trend of a structure (e.g. fault plane) has bigger intersection angle with one vertical view, then the structure can be more easily recognized from this view instead of the other. Here the orientation of different vertical views are illustrated by the yellow line.

All three views show the basic shape of the desired structure with slightly different details. Note that different views might yield false positives in different locations. Therefore, we can take advantage of it by averaging the results. An example of the averaged result is shown in Figure 5.8

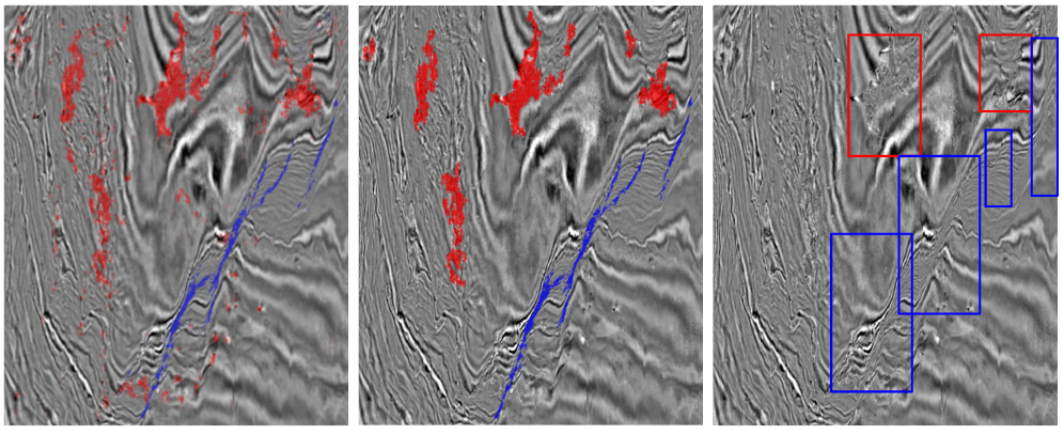

Figure 5.8: Left: Averaged result from three views presented in Figure 5.7. Middle: Result post-processed by using density-based clustering method. Right: Ground truth annotated using bounding box. Red boxes indicate channels, blue boxes indicate faults.

As shown in the averaged result, many big chunks of false positives are removed since three models have different classification results over them. Furthermore, a post-processing step is performed by using density-based clustering method as described in Chapter 3. It can be observed that most small noises with a low density of points around it are eliminated. At last, a clean and consistent segmentation can be obtained through the whole procedure.

Note that the result also shows some extra channel structures which are not labeled by the ground truth bounding box. Since our annotation strategy is "precision over recall", there could be some subtle structures missing from the ground truth. Therefore we can only have more accurate evaluation on the segmentation performance by further consulting the experts.

5.1.6 3D Visualization

For each voxel in the whole 3D seismic volume, we ensemble the result from three models of different views, which are trained on $32 \times 32 \times 5$ patches. The result is then filtered in the post-processing step. The resolution of the 3D volume is $268 \times 923 \times 1126$. Therefore in total 278 million voxels are classified, in which 0.4 million voxels are labeled training data.

We illustrate the detected fault and channel in 3D space separately using VRGeoDemonstrator, as shown in Figure5.9 and Figure 5.10.

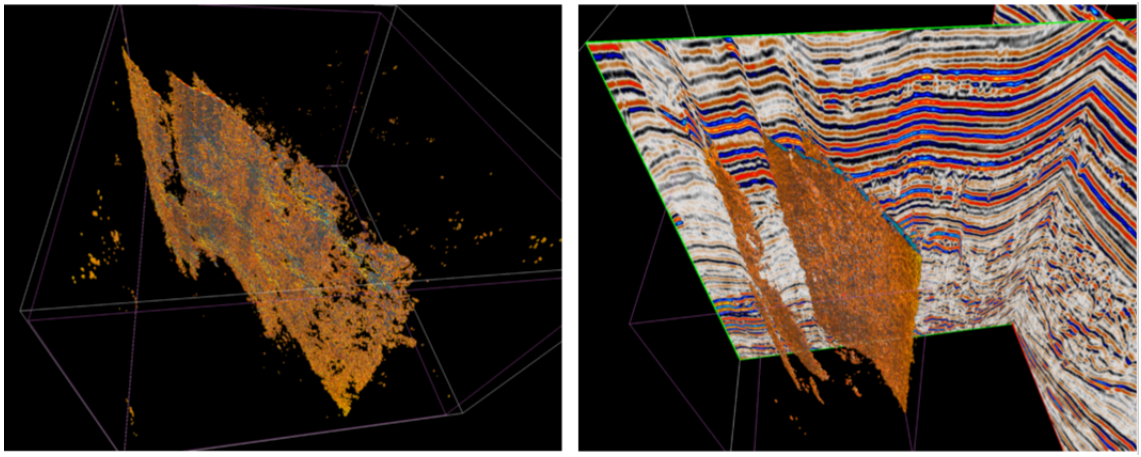

Figure 5.9: Left: Overall faults detected in the whole dataset in 3D space. Right: Part of the detected fault with original slices as reference.

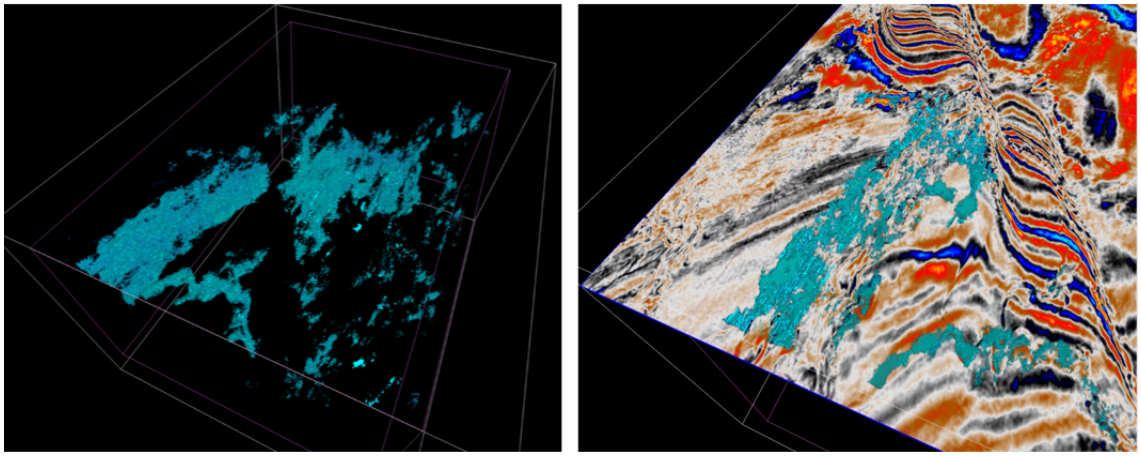

Figure 5.10: Left: Overall channels detected in the whole dataset in 3D space. Right: Part of the detected channel structure with original slices as reference.

As shown in the 3D visualization, the fault voxels form several thin and clear fault planes going through the seismic volume. Detected channel structures are bit noisy compared to the faults, but when aligned with raw slices we can tell that the location of channels are correct despite the outline details. One possible reason suggested by the geophysicists is that there are many sub categories of channels which have different forms. Therefore in further studies it is recommended to divide the annotation of channels to sub-classes.

5.2 F3 Dataset

In the last section we mainly tested the trained model on the Parihaka seismic volume. To further evaluate and improve the generalizability of our models, we conducted more experiments on the F3 volume.

Due to different geographic locations, data acquisition techniques and data processing methods, the visual difference between seismic volumes can be considerably large. Comparison

of the two volumes can be seen in Figure 5.11.

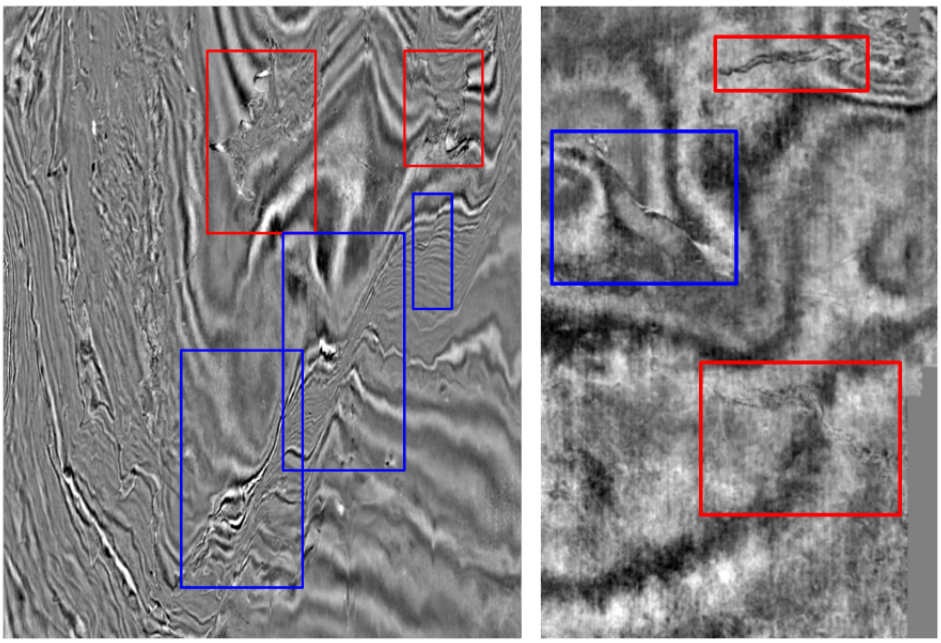

Figure 5.11: Left: Time slice from the Parihaka dataset. Right: Time slice from the F3 dataset. Channels are included in red bounding boxes, while faults are included in blue bounding boxes.

Since we only have a few slices from one Parihaka seismic volume for training, it is foreseeable that the model will overfit to the Parihaka dataset. Moreover, the labeled pixels in the Parihaka dataset are mainly presenting the network with one or two types of channel and fault structures. Therefore when given other types of faults and channels from a more noisy data volume, generalization performance of the network certainly becomes a major issue.

When directly apply the best model trained from the Parihaka inline $32 \times 32 \times 5$ dataset to test slice from the F3 dataset, we observe that seldom any pixel is classified as fault, instead, fault structures are marked as channel as shown in Figure 5.12. Also, the segmentation result is inaccurate and noisy. Compared to the test results we obtain from Parihaka volume with high-quality, it is obvious that our model is overfitted to the training data.

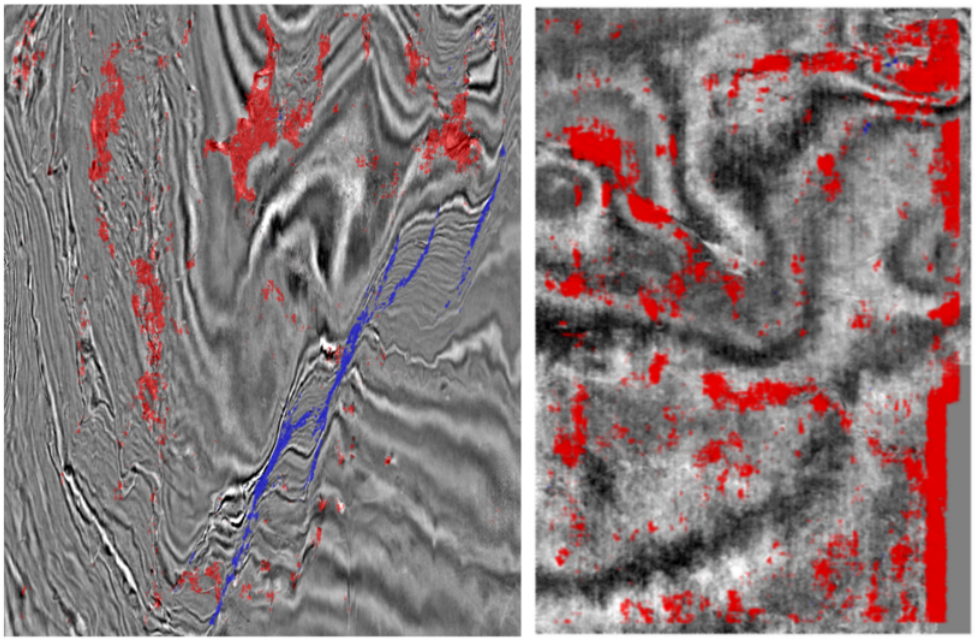

Figure 5.12: Left: Test result on time slice of Parihaka volume. Right: Test result on time slice of F3 volume using the same model.

5.2.1 Architecture

In order to improve the validation performance, we first experimented with a 3D CNN architecture as specified in Chapter 3. The 3D CNN takes $32 \times 32 \times 32$ patch as input and performs 3D convolution in order to detect 3D features in the input volume. Our 3D CNN architecture contains 5.17 million parameters in total. It is trained on patches extracted from Parihaka dataset, and validated on patches extracted from F3 dataset. The validation accuracy curve is given in Figure.. A baseline of validation accuracy 61.67% is given by the best model trained on $32 \times 32 \times 5$ inline Parihaka dataset and validated on the F3 dataset.

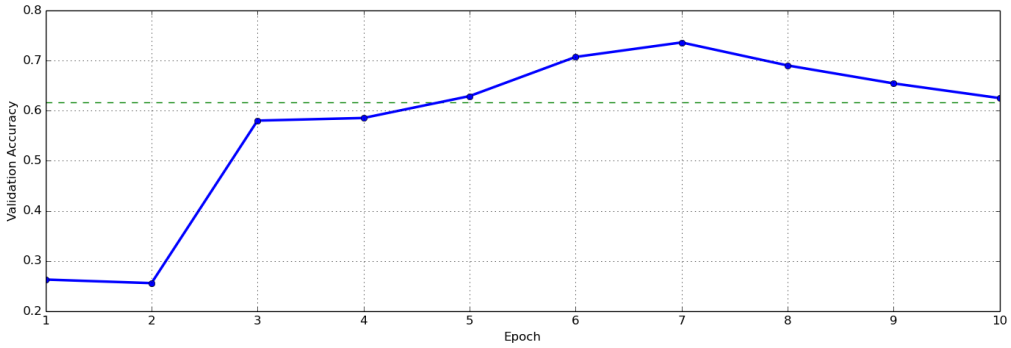

Figure 5.13: Validation accuracy curve of 3D CNN in comparison with baseline. The figure is plotted using matplotlib since 3D CNN implementation is based on Tensorlayer API which do not support automatic Tensorboard logs.

The 3D model showed big advantage in generalizability compared to 2D models. The best validation accuracy is achieved by 3D model at epoch 7 with 73.60%, which beats the 2D model by over 10%.

Although 3D models shows great potential in our task both theoretically and empirically, it is fairly hard to train. A single epoch takes more than three hours, which makes it hard to tune the hyper-parameters and trying new dataset. In the future we will conduct more experiments on the 3D DNN when enough time is granted.

5.2.2 Dataset

After experimenting with different architectures and various combinations of hyper-parameters, we noticed that the all models still have low quality fault detection performance. Excluding the influence of architecture and hyper-parameters, we infer that the pitfall lays in the dataset.

One assumption is that the former dataset is unbalanced. After extracting patches from all annotated pixels, we obtained a dataset that contains 164k channel patches and 63k fault patches. When we simply train the network using the unbalanced dataset, the network learned to assign higher probability to the channel class and lower probability to faults. This problem has been amplified when applied to a new dataset which has different distribution of samples for classes.

The other assumption is that different data volumes scale their seismic data differently, especially in the vertical direction where faults can be better detected. Therefore when the same window size is used, the local properties may look quite different. To prove this assumption, we simply scaled an inline test slice from the F3 volume vertically to make it more similar to the training slice and applied the same model to both the original test slice and the scaled test slice. The comparison of results is shown in Figure 5.14. As shown in the comparison, faults can be better recognized in the scaled test slice.

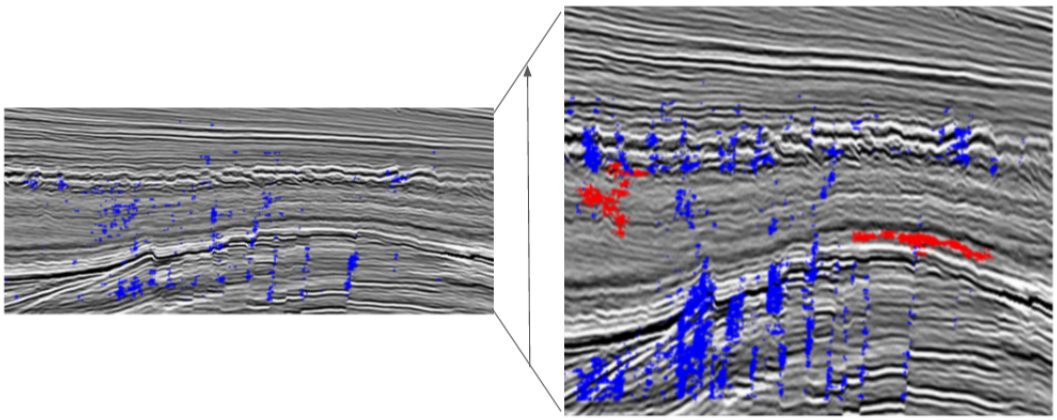

Figure 5.14: Left: test result on the original F3 slice. Right: test result on the same slice which has been vertically scaled so that it looks more similar to the density of rock layers in the Parihaka dataset.

Considering the above mentioned two factors — balanced classes and scale of patches, we can build several different datasets for training. We evaluate the performance of the trained models on patches extracted from an annotated F3 time slice. The training and validation curves of different datasets and architectures are shown in Figure 5.15.

Figure 5.15: Training and validation accuracy curves based on different dataset. The suffix "balanced" means that we flipped all fault samples horizontally to acquire double the amount of fault so that the dataset is roughly balanced. Suffix "scaled" suggests that the dataset consists of patches with random scale in a certain range. (e.g. For $32 \times 32 \times 5$ dataset, we extract patch size of $[32, 64] \times [32, 64] \times 5$, and resize them to $32 \times 32 \times 5$ spatially.)

Table 5.6: Training and validation results with different datasets

Model	Size	Dataset	Training(%)	Validation(%)
cnn7-time	$32 \times 32 \times 5$	unbalanced	99.20	60.56
cnn3d	$32 \times 32 \times 32$	unbalanced	97.37	**73.60**
cnn7-time	$32 \times 32 \times 5$	balanced	93.36	63.96
cnn7-time	$32 \times 32 \times 5$	scaled	97.20	55.00
cnn7-time	$32 \times 32 \times 5$	balanced + scaled	91.40	**69.58**
cnn7-time	$64 \times 64 \times 1$	balanced	95.14	**70.73**
cnn10-time	$64 \times 64 \times 1$	balanced	98.39	61.21

In the second part of the table it can be observed that, balancing the number of samples in classes has improved the validation accuracy by around 3%, while using scaled input patches didn't bring any advantage in validation.

The best validation accuracy on the $32 \times 32 \times 5$ patches was achieved by doing both balancing and scaling, where the dataset was balanced by flipping the faults horizontally and adding scaled fault samples. No scaling was used for channels and negative samples.

Therefore, we argue that a balanced dataset is necessary in training time, while scaled input patches only provide better result under certain condition. There are many possible reasons why scaling did not work in our experiments as expected, we infer that when all input patches are scaled, the variance of the dataset is much higher therefore the model capacity should be adjusted accordingly.

A comparison of the best test result achieved on unbalanced dataset and adjusted dataset are illustrated in Figure 5.18.

5.2.3 Regularization

Next we experimented with different regularization techniques including dropout, L2 norm and data augmentation in order to improve the generalizability of the model.

We noticed that since we initialized the network by random initialization, different runs using the same architecture and dataset can have very different performance on the validation set. This phenomena makes it hard in evaluating the performance of different settings. Therefore we added a batch normalization layer after the last convolution layer, which is expected to relieve the issue of proper initialization.

The best runs of each setting are illustrated below. Experiments are carried out with CNN7 architecture on $32 \times 32 \times 5$ inline dataset. The training dataset is balanced by flipping and adding scaled fault samples.

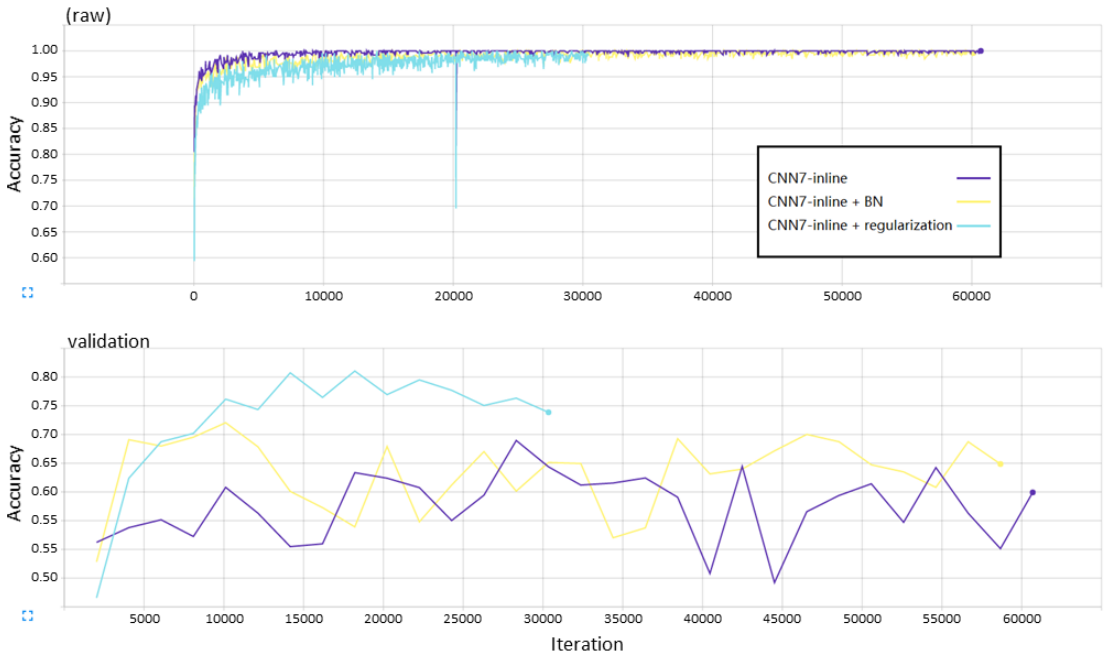

Figure 5.16: Comparison of training and validation accuracy curves with and without regularizers. The added term "BN" indicates that we used batch normalization after the fully connected layers. The added term "regularizers" indicates that dropout, data augmentation, L2 normalization are all applied.

Table 5.7: Training and validation results with and without regularizers

Model	Dataset	Training Acc.(%)	Validation Acc. (%)
cnn7-inline	plain	99.96	68.96
cnn7-inline	Batch Normalization	99.62	72.05
cnn7-inline	Regularizers	98.46	**81.04**

As can be seen from the figure above, the validation curve is not as smooth as presented in

the last section. One reason could be that the size of the validation set is too small. Since we only have one annotated time slice from the F3 dataset, the validation dataset contains merely 3,000 samples. In general, we see that both batch normalization and regularizers have slightly reduced the training accuracy, yet achieved better validation result compared to plain network. The best validation accuracy was achieved by the model with all regularizers applied with 81.04%.

5.2.4 Over-Segmentation

The super-pixel method partitions the image into segments based on similarity measures. A super-pixel shows a group of pixels that are visually consistent, which normally belong to the same semantic class.

A super-pixel segmentation example is given in Figure 5.17. As can be seen, all desired structures are covered by the boundaries of the super-pixels, while blobs of uniform regions where no structure exists are within super-pixels. Therefore, points of interest are picked as pixels on boundaries. The classification will be performed on points of interest instead of pixels over the whole image.

The original image has $951 \times 651 = 619,101$ pixels, while the boundaries marked in the second image covers only 144,169 pixels. Generally, by selecting points of interest using bottom-up segmentation method, we could reduce around 75% of testing time without sacrificing the performance.

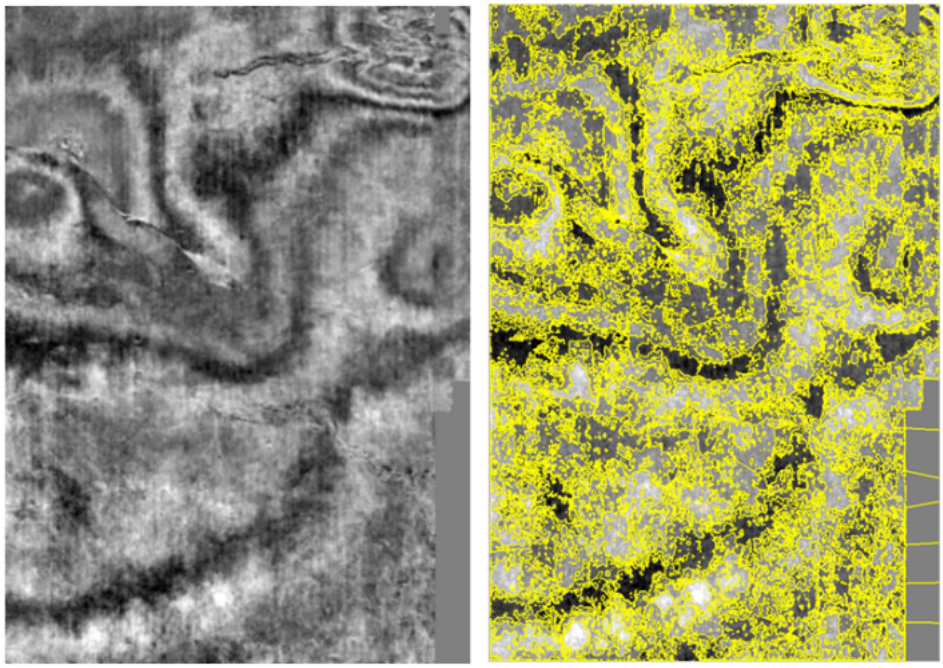

Figure 5.17: Left: Original time slice from the F3 volume. Middle: Boundaries of super-pixels generated using SLIC algorithm by Achanta et al. [2012].

With points of interest selected, it is also beneficial for combining results of different scales. As we know, large patch size such as 64×64 yields very coarse result over the whole image. By performing testing on boundaries, the disadvantage of localization can be reduced.

5.2.5 Test result

We evaluate the models on a time slice from F3 volume. The testing process follows the convention described in Section 5.1. We mainly tested with $32 \times 32 \times 5$ patch size and CNN7 model. The training dataset is balanced by flipping and adding scaled fault samples. Regularizers including dropout, L2 norm and data augmentation are used at training time. Finally, results are post-processed by DBSCAN clustering.

The comparison of unbalanced and balanced datasets is given in Figure 5.18. It can be seen that when the dataset contains a balanced number of channels and faults, more fault structures can be detected correctly.

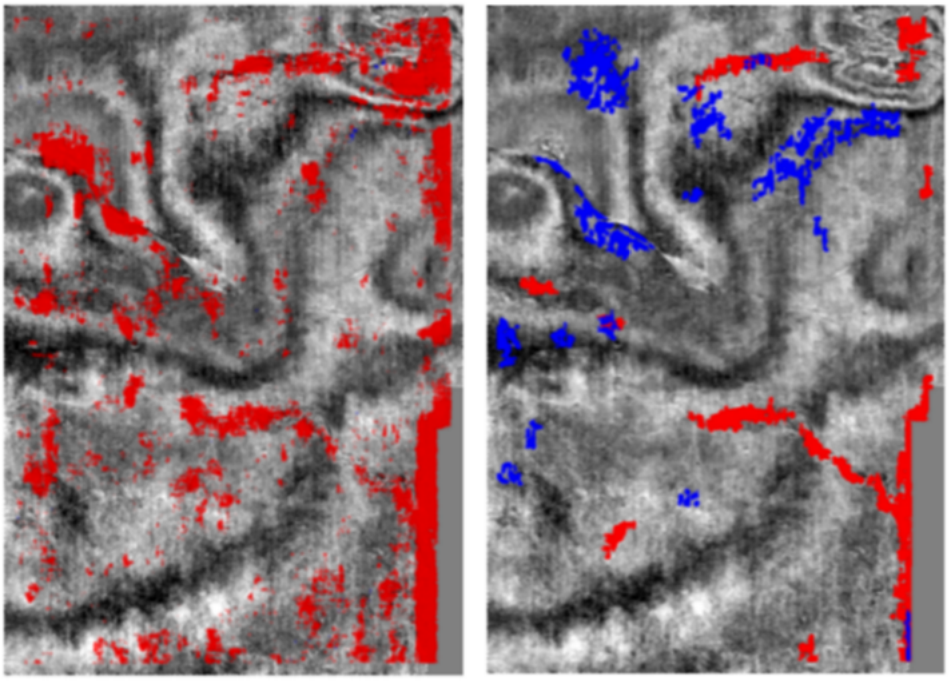

Figure 5.18: Left: Best result from models trained on unbalanced Parihaka dataset. Right: Result from models trained on balanced and scaled dataset, averaged from models of three views.

Points of interest are selected through over-segmentation and tested using inline, crossline and timeslice models. The final averaged result is given in Figure 5.19 in comparison to ground truth bounding box:

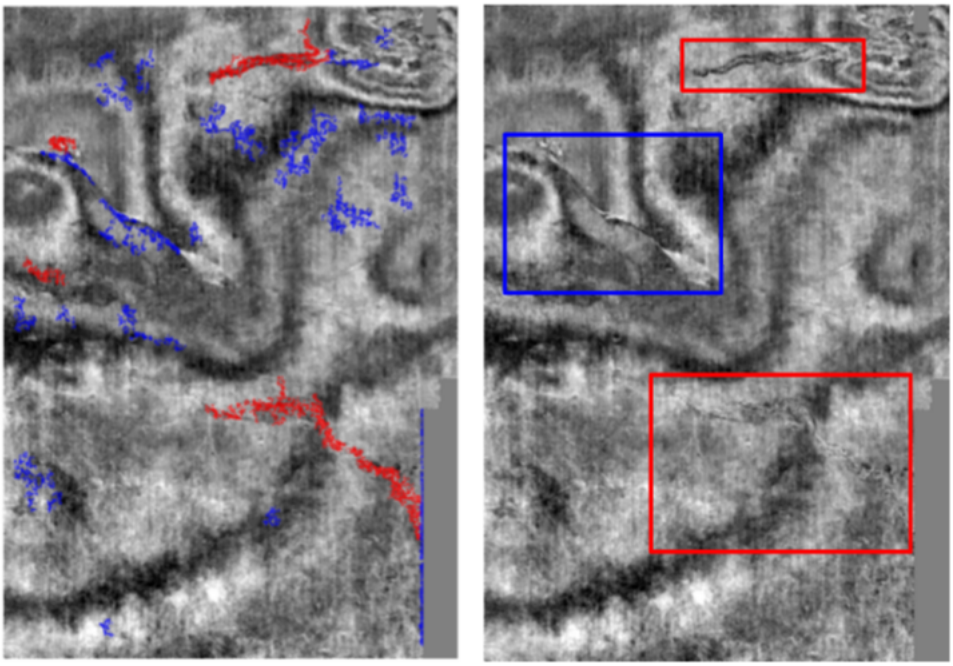

Figure 5.19: Left: Test result on points of interest. Results are averaged from crossline and inline view and post-processed. Right: Original image with ground truth bounding boxes.

It can be seen from above test results that the models perform better in detecting channels than faults. We consider the main reason to be that the number of original fault samples is less than a half the number of channels. Even though the dataset is balanced in numbers by jittering the fault samples, the generalizability of fault class is still worse than the channel class. Another possible reason could be the quality of fault annotation since it is performed by the author with little domain knowledge and experience.

6 Conclusion

In this thesis we proposed a method based on convolutional neural networks for detecting underground geophysical structures including channels and faults in seismic data volumes. We introduced a workflow to build the system from scratch. First, raw seismic data needs to be annotated on 2D slices by simply drawing lines inside the structure. Compared to typical per-pixel annotation, our annotation is considerably weaker but it also requires far less work. From the weakly annotated data, a training set can be constructed by extracting patches of a certain spatial size, depth, and view. We then conducted extensive experiments training different CNN architectures including CNN7, CNN10 and a 11-layer 3D CNN on different training sets. To validate our method, we applied the trained CNN models as local window classifier to unseen slices in Parihaka and F3 seismic volume. To achieve higher accuracy, we ensemble models trained on patches from different views by averaging their test results. The pixel-wise classification result is further refined by combining bottom-up segmentation results and performing post-processing using density-based clustering method.

Since the training set is fairly small and all samples are extracted from one volume, overfitting became the major challenge that needed to be dealt with. We addressed this issue by deploying regularization techniques, manipulating the datasets, and applying new architecture, namely 3D CNN. The best classification accuracy on the F3 dataset achieved by our models is 81%. (Since no similar study is conducted before and no standard dataset is available, there is no baseline to compare with.) Our experiment results proved that with the above mentioned methods, models trained on the Parihaka dataset generalize well to the test slice from the F3 dataset. To further quantify the performance of the segmentation task, measures such as intersection over union, pixel error are commonly used. However, note that we have no accurate per-pixel ground-truth, the segmentation results were only assessed by human experts.

It is notable that our CNN models can achieve high-quality segmentation results despite the small training set. As is well known, deep learning is driven by big data. Therefore, we believe that our models have high potential for further improvement when more training data from various volumes can be provided.

The work performed in this thesis laid the foundation of further researches. Future works can be conducted in several directions as follows:

1. To Deploy multi-view convolutional neural network, in which the network takes three orthogonal patches centered at the same voxel as inputs and learn their features jointly.
2. To conduct further experiments on the 3D CNN architecture. In our experiments, 3D CNN showed big improvement compared to 2D CNN. Therefore more time can be put into studies on the 3D CNN architecture.
3. To learn hierarchical features. Since channels and faults are both narrow structures, a

local classifier approach is sufficient to detect them. However, when more structures such as salt dome are added to our target group, long-range contextual information is needed. Therefore we should consider to learn hierarchical features in order to detect structures in different scales.

4. To combine domain knowledge with the CNN. We have briefly experimented with training our CNN models on patches extracted from data volume that has been filtered by some seismic attributes. Although it did not show any performance improvement compared to models trained on raw data, it is still worth further experimentation with different attributes and different CNN architectures.

.

References

Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Süsstrunk. Slic superpixels compared to state-of-the-art superpixel methods. *IEEE transactions on pattern analysis and machine intelligence*, 34(11):2274–2282, 2012.

Mike Bahorich and Steve Farmer. 3-d seismic discontinuity for faults and stratigraphic features: The coherence cube. *The leading edge*, 14(10):1053–1058, 1995.

Jianhua Cao, Yang Yue, Kunyu Zhang, Jucheng Yang, and Xiankun Zhang. Subsurface channel detection using color blending of seismic attribute volumes. *International Journal of Signal Processing, Image Processing and Pattern Recognition*, 8(12):157–170, 2015.

A Chaouch and JL Mari. 3-d land seismic surveys: Definition of geophysical parameter. *Oil & Gas Science and Technology-Revue de l'IFP*, 61(5):611–630, 2006.

Dan Ciresan, Alessandro Giusti, Luca M Gambardella, and Jürgen Schmidhuber. Deep neural networks segment neuronal membranes in electron microscopy images. In *Advances in neural information processing systems*, pages 2843–2851, 2012.

Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

Clement Farabet, Camille Couprie, Laurent Najman, and Yann LeCun. Learning hierarchical features for scene labeling. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1915–1929, 2013.

David Gibson, Michael Spann, and Jonathan Turner. Automatic fault detection for 3d seismic data. In *DICTA*, pages 821–830, 2003.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Dave Hale. Methods to compute fault images, extract fault surfaces, and estimate fault throws from 3d seismic images. *Geophysics*, 78(2):O33–O43, 2013.

Bruce S Hart. Channel detection in 3-d seismic data using sweetness. *AAPG bulletin*, 92 (6):733–742, 2008.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

David H Hubel and Torsten N Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology*, 148(3):574–591, 1959.

David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154, 1962.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Shuiwang Ji, Wei Xu, Ming Yang, and Kai Yu. 3d convolutional neural networks for human action recognition. *IEEE transactions on pattern analysis and machine intelligence*, 35 (1):221–231, 2013.

Andrej Karpathy, Justin Johnson, and Li Feifei. Lecture notes in convolutional neural networks for visual recognition, January 2016.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015.

Aurélien Lucchi, Kevin Smith, Radhakrishna Achanta, Vincent Lepetit, and Pascal Fua. A fully automated approach to segmentation of irregularly shaped cellular structures in em images. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 463–471. Springer, 2010.

R Mohebian, M Yari, MA Riahi, and R Ghanati. Channel detection using instantaneous spectral attributes in one of the sw iran oil fields. *Bollettino di Geofisica Teorica ed Applicata*, 54(3):271–282, 2013.

Michael Nielsen. *Neural Networks and Deep Learning*. 2016.

Stein Inge Pedersen, Trygve Randen, Lars Sonneland, and Øyvind Steen. Automatic fault extraction using artificial ants. In *SEG Technical Program Expanded Abstracts 2002*, pages 512–515. Society of Exploration Geophysicists, 2002.

Trygve Randen, Stein Inge Pedersen, and Lars Sønneland. Automatic extraction of fault surfaces from three-dimensional seismic data. In *SEG Technical Program Expanded Abstracts 2001*, pages 551–554. Society of Exploration Geophysicists, 2001.

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

Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. Learning spatiotemporal features with 3d convolutional networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4489–4497, 2015.

Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

Ciyuan Zhang, Charlie Frogner, and Tomaso Poggio. Automated geophysical feature detection with deep learning, April 2016.