Problem Chosen	2022 ShuWei Cup Summary Sheet	Team Control Number
A		2022111313915

Summary

In this paper, we focus on algorithms for automatic seismic stratigraphic tracking. We first use a robust deep learning denoising algorithm to denoise the initial seismic data, and then build an automatic seismic stratigraphic tracking algorithm for the denoised seismic data. We mainly use the full convolutional network U-net in deep learning to achieve the automatic seismic stratigraphic tracking, and propose an improved seismic DNA algorithm to improve the continuity and accuracy of the automatic stratigraphic tracking by introducing the clustering method into the common seismic DNA algorithm. Finally, a correlation-based and waveform 3D level automatic tracking model is established using denoised seismic data, and the results are analyzed.

Question one: Seismic data denoising. Since the traditional modeling approach based on a priori cannot accurately portray the noise distribution, which affects the accuracy of model assumptions and parameter settings, a robust deep learning denoising algorithm is used in this paper to denoise the raw seismic data. First, the training data are put into the deep learning network model one by one for network model training, then the trained model parameters are saved, and finally the pre-trained parameters are used for testing, comparing the best test results, and the best test result pre-trained parameters are selected for the denoising process of the raw seismic data.

Question two: Automatic seismic layer tracking algorithm based on full convolutional neural network. Traditional layer conventional layer tracking techniques are mainly interpreted manually on two-dimensional seismic profiles, and interpreters track seismic layers based on their own experience and subjective judgment. The deep learning method is based on a large amount of data, and the data is predicted by constructing a suitable network model for multi-level feature extraction. In this paper, we use U-net network as the base network, and train the network by feeding a large amount of data to make the network have certain prediction ability of seismic layer tracking, and achieve the ability of automatic seismic layer tracking by effectively extracting the features in the seismic data.

Question three: Improved seismic DNA automatic tracking algorithm. The traditional seismic DNA algorithm needs to scan all points in the region when tracking, which leads to matching more seismic waves that meet the conditions and picking up poor continuity of the seismic layers. For this reason, we improve the traditional seismic DNA algorithm by introducing the clustering method into the seismic DNA algorithm. By classifying the seismic waves found by the seismic DNA algorithm, and then using the Euclidean distance to connect the clustered points, and using the coherence algorithm to find and mask out the fault areas, the seismic stratigraphy found by the seismic DNA algorithm has better continuity and accuracy.

Question four: Evaluate the results of two automatic tracking algorithms in Problem 2 and Problem 3. The full convolutional neural network-based seismic layer auto-tracking algorithm is affected by noise and is more subjective, some edges are not clear enough, and the identified seismic layers are cluttered. However, the improved seismic DNA auto-tracking algorithm can pick up multiple seismic layers at the same time, and the traced seismic layers are more continuous, and no cascading occurs at the fault location.

Question five: Correlation-based and waveform 3D horizontal automatic tracking model. The matching search-based full-layer tracking method is mainly based on the amplitude characteristics of the longitudinal distribution of multiple layers in the time direction and the characteristics of the gaps between layers, and designs a layer longitudinal distribution feature

extraction algorithm, a matching search-based data block generation algorithm, and an amplitude-oriented data block connection algorithm. The method uses the feature extraction algorithm of longitudinal distribution of layers and the data block generation algorithm based on matching search to connect the extreme points in the 3D seismic image into layer blocks in 3D space, and the data block connection algorithm based on amplitude orientation to connect the layer blocks to form large layers in the seismic image, and then the final tracking results are obtained by extending the layers for the gap problem of the layers.

Keywords: deep learning; fully convolutional neural network; improved seismic DNA algorithm; clustering algorithm; U-net network

Content

1. Introduction

1.1 Background

With the rapid development of the economy, the demand for oil as the main source of energy in industry is increasing. Over the years, seismic exploration technology is developing rapidly, and coupled with the continuous exploration and development of oil and gas, simple tectonic oil and gas reservoirs can no longer meet the needs of exploration and development, which urgently requires increased development of tectonic-rocky and lithologic oil and gas reservoirs. However, tectonic-rocky and lithologic reservoirs are extremely complex and difficult to explore, so it is necessary to do a good job in the interpretation of seismic data and accelerate the work of reservoir prediction and fine reservoir description. Accurate tracking of seismic levels can better serve the interpretation of seismic data, thus realizing the exploration and development of complex oil and gas reservoirs.

Initially, seismic level tracking was done manually, and interpreters compared the seismic wave kinematics and dynamics manually. However, with the increasing amount of seismic data, the time-consuming, labor-intensive, inefficient and inaccurate manual picking can no longer meet the production needs, so it becomes necessary to implement automatic seismic level tracking.

2. Problem analysis

2.1 Analysis of question one

The noise contained in seismic exploration data is complex, and traditional modeling approaches based on a priori cannot accurately portray the noise distribution. Deep learning automatically extracts the deep-level features of the data through multilayer convolutional neural networks and adaptively learns a complex denoising model by using nonlinear approximation capability, which brings a new idea for denoising seismic data. However, the current deep learning-based denoising methods do not have strong generalization ability of the learned model when the sample coverage is not sufficient, which greatly reduces the denoising effect. To this end, a robust deep learning denoising algorithm will be used to denoise the initial seismic data. The specific denoising process is shown in Figure 1：

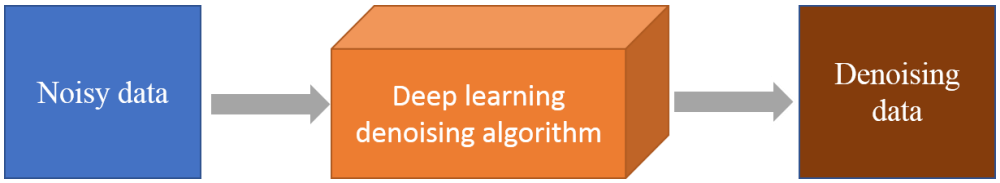


Figure 1 Seismic data denoising flow chart

2.2 Analysis of question two

Stratigraphic tracing is a very important part of seismic image interpretation and a fundamental task in seismic image interpretation. Seismic exploration is to excite seismic waves by means of explosions, and due to the variation of wave impedance in different geological formations, the sound waves are reflected by the rock layers when crossing different geological structures, and the reflected waves are transmitted upward to the ground to be accepted and recorded by the ground receiving sensors, which is the This is the most primitive seismic recording. The traditional layer tracking method is carried out manually on a two-dimensional seismic profile, relying mainly on the subjective judgment of the interpreter to complete the homogeneous axis tracking according to the wave peaks and troughs in the seismic profile, and in the process of homogeneous axis tracking, three criteria are mainly followed: continuity, homogeneity and waveform similarity.

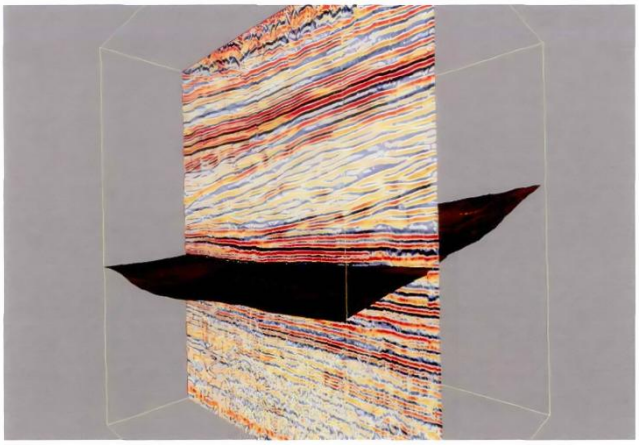


Figure 2 Layer plane

With the increase of exploration efforts and the maturity of exploration technology, more and more seismic data are available and the requirements for exploration accuracy are getting higher and higher. This time-consuming and laborious method of manual tracking, which is easily influenced by the subjectivity of the interpreters, has gradually failed to meet the actual production requirements, so the research of automatic seismic level tracking algorithms has gradually received the attention of researchers.

In recent years, deep learning has developed rapidly and has been widely used in different fields, and more and more new algorithms have been applied in the processing and interpretation of seismic exploration data, and the most representative

one is the neural network method. The structure of neural network is shown in Fig. 3, and various complex mappings can be constructed through network connections to achieve multi-level extraction of data features.

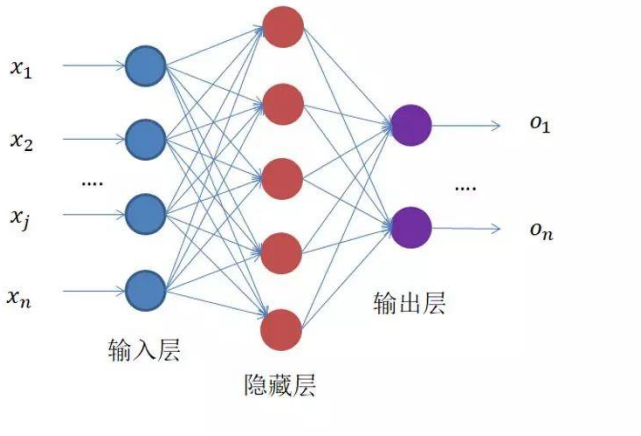


Figure 3 Neural network structure diagram

2.3 Analysis of question three

In the interpretation of seismic data, the tracing of the in-phase axis is very important. For decades, interpreters have conducted artificial comparison tracking based on the dynamics and kinematics of seismic waves, namely, three basic criteria: amplitude, in-phase or continuity and waveform similarity. Since 1970s, many scholars have studied the automatic tracking and picking method of seismic wave in-phase axis, using different methods and from different angles to quantitatively express three basic criteria. Artificial horizon picking is a method to obtain the horizon line (stratum interface) by manually tracing the continuous reflection in-phase axis of the stratum by using waveform similarity, and then all the horizon lines are interpolated to form the horizon plane. Manual pickup has disadvantages such as low efficiency and poor accuracy. In order to overcome the problems of low tracking time efficiency and poor reliability of results, researchers have paid more and more attention to automatic level tracking algorithm in recent years, and relevant researches on automatic level tracking have been developed rapidly.

Seismic DNA algorithm is a new seismic level tracking algorithm. Its core idea is to convert the numerical information of seismic data into the character information of text data through some kind of conversion, so that seismic data can be searched as text data. Based on the seismic DNA algorithm, an improved algorithm is proposed in this paper. The clustering method is introduced into the seismic DNA algorithm, which can solve the problem that the seismic DNA algorithm matches the seismic layer continuity is poor and the layer division is not obvious. The specific process is shown in Figure 4：

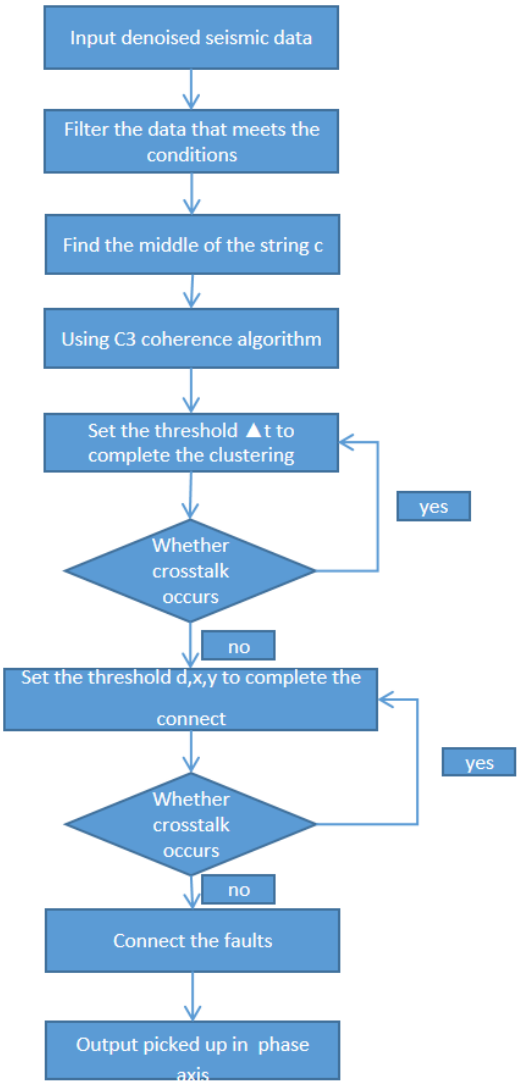


Figure 4. Flow of improved seismic DNA algorithm

2.4 Analysis of question four

The results of the deep learning-based image edge detection algorithm for Problem 2 and the improved seismic DNA-based algorithm for Problem 3 are analyzed in terms of seismic layer information, algorithm efficiency, model rationality, data error, and tracking accuracy. Finally, the results of both algorithms are evaluated. The specific flow of the model results evaluation is shown in Figure 5:

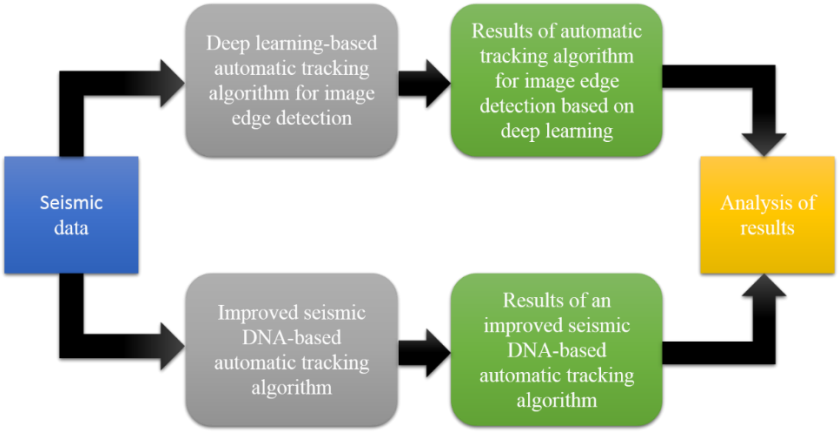


Figure 5 Model result evaluation process

2.5 Analysis of question five

There are three directions in the 3D seismic image, namely inline direction, crossline direction and time direction. The seismic record is presented in the 3D seismic image as a waveform distributed along the time direction, and the distribution of the 3D seismic image in 3D space is shown in Figure 6, which is called a data in the 3D seismic image. 3D seismic data is the distribution of seismic waveform data in the inline and The distribution of seismic waveform data in the inline and crossline profiles. This is reflected in the 3D seismic image where the waveform amplitude is smaller.

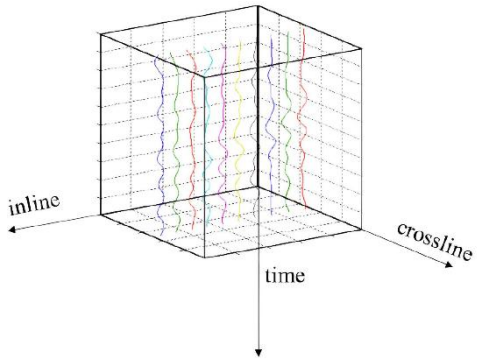


Figure 6 Schematic diagram of 3D seismic image

In order to facilitate the observation and use by seismic interpreters, the profile data can be extracted from the 3D seismic data body. The profile data are divided into vertical and horizontal profiles, and the profiles along the vertical direction are usually called inline profiles; and crossline profiles perpendicular to the main measurement line. The profile data are distributed in the 3D seismic image as shown in Figure 7. Different inline profiles or crossline profiles are distributed horizontally in the 3D seismic image, while inline profiles and crossline profiles are orthogonal vertically in the 3D seismic image.

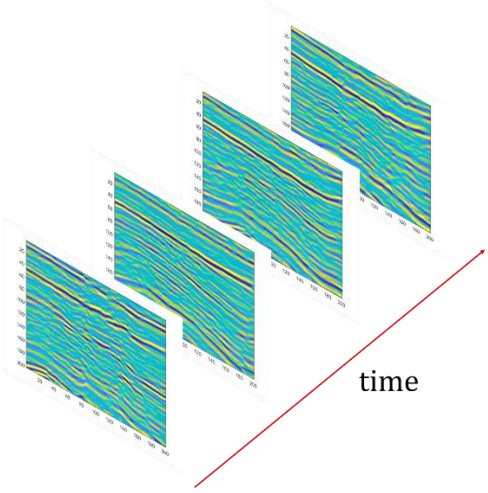


Figure 7 Schematic diagram of the distribution of profile data in 3D seismic images

With the emergence and development of high-precision full data acquisition systems, the workload of manual tracking becomes very huge, which seriously affects the efficiency of seismic interpretation. 3D full-layer tracking is generated to address the shortage of manual interpretation. In this paper, we extract layer longitudinal distribution features for multiple vertically adjacent layers, and realize parallel tracking of layers in 3D seismic images by matching and searching the layer longitudinal distribution features.

3. Symbol and Assumptions

3.1 Symbol Description

Category	Meaning
F_{ES}	the function of noise estimation subnet
$\boldsymbol{\theta}_{\mathrm{ES}}$	Parameter set
w_i	the weight parameter of layer i
R	ReLU activates the function
$\hat{\boldsymbol{\eta}}$	noise estimation
F_{DS}	the function of denoising the subnet
$\boldsymbol{\theta}_{\mathrm{DS}}$	parameters in a denoising subnet
$\boldsymbol{y}_{\mathrm{out}\ i}$	the output of layer i
M	total number of slice time samples
N	total number of slice seismic track
T	samples
S	time sampling serial number
U	serial number of seismic track records
l	the mean of the seismic data scale factor of noise intensity

3.2 Fundamental assumptions

In order to simplify the given problem and modify it to a more suitable simulation. In reality, we make the following basic assumptions. Every assumption has a valid reason.

(1) It is assumed that the individual missing data in the data set will not have a significant impact on the establishment and solution of the model.

(2) It is assumed that the selected seismic data are representative and accurate.

(3) It is assumed that the time scales of all influencing factors in the tracking model are the same.

(4) Let's assume the actual situation is the same as the model.

4. Model

4.1 Modeling of question one

Random noise seriously affects the processing of seismic data, and how to efficiently extract effective information from noise-laden data is an important research direction in the field of seismic data processing. To solve this problem, scholars have proposed a variety of random noise suppression methods. According to whether a specific a priori model is assumed, these methods can be divided into two categories: traditional denoising methods based on a priori models and denoising methods based on deep learning. The traditional denoising methods based on a priori models start from the distribution of data a priori and then establish mathematical models, but based on the data a priori knowledge, the manually established models can only extract shallow features with weak expressiveness and cannot describe complex noise distributions, which affects the accuracy of model assumptions and parameter settings. However, the noise suppression method based on deep learning has a powerful deep feature extraction capability, and the main features of seismic data are extracted by multi-layer convolution to construct the denoising model adaptively, which solves the problems of inaccurate models and uncertainty of parameter settings due to limited a priori knowledge in the traditional denoising methods. However, the current deep learning-based denoising methods do not have strong generalization ability of the learned models with insufficient sample coverage, which greatly reduces the denoising effect. To this end, a robust deep learning denoising algorithm is used in this paper to denoise the seismic data.

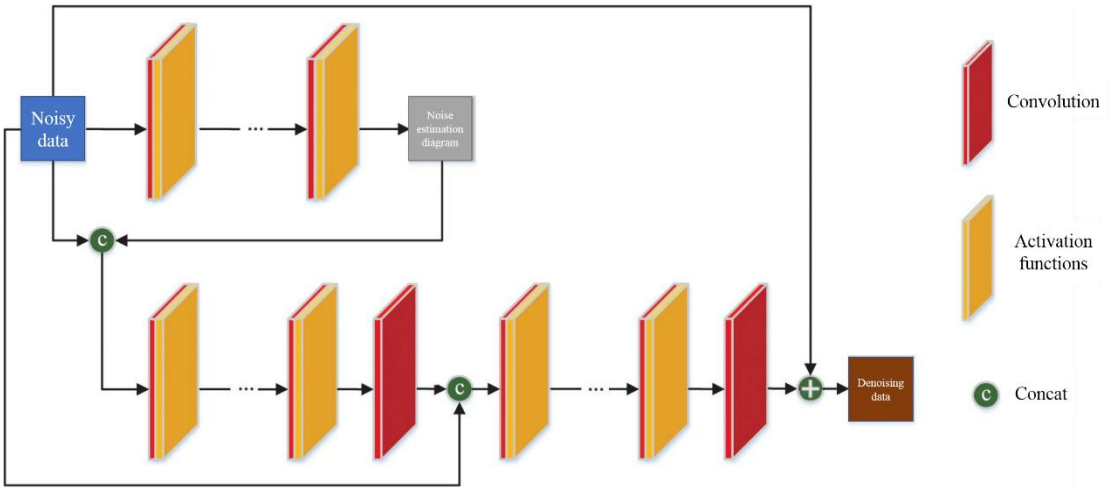


Figure 8 Structure of deep learning denoising network model

The network model structure of the algorithm in this paper is shown in Figure 8, which contains two parts of molecular networks, the noise distribution estimation sub-network and the denoising sub-network, respectively. Among them, the noise distribution estimation subnet is divided into 5 layers, each layer consists of convolution and modified linear units (ReLU), which are used to learn the distribution of random noise in seismic data. The convolution operation is used to extract the noise distribution features, and the first 4 layers and the 5th layer are convolved to obtain 64 and 1 feature mapping, respectively. The ReLU activation function is used to better approximate the distribution of the real noise estimation map. The denoising sub-network consists of two stages. Among them, the first stage consists of 5-layer network, the first 4 layers consist of Conv and ReLU respectively, and 64 feature mappings are obtained after convolution processing in each layer. The fifth layer consists of Conv only, and 1 feature mapping is obtained after the convolution operation, which is the deep seismic data feature. Then, the deep features output from the first stage are fused with the noisy seismic data features and then passed to the second stage. The feature fusion is performed by vector splicing, i.e., the shallow and deep seismic data features are spliced and fused to increase the accuracy of seismic data feature extraction. The second stage includes 12 layers of the network, corresponding to layers 6 to 17 of the denoising sub-network. Among them, layers 6 to 16 consist of Conv and ReLU, and 64 feature mappings are obtained after convolutional processing; layer 17 consists of Conv, and 1 feature mapping is obtained after convolutional operation, which is the denoised seismic data. Finally, the obtained denoised seismic data is subtracted from the noise-containing seismic data to obtain the residual learned noise.

All the above convolution operations are performed before expanding the boundaries with 0 to ensure that the input and output sizes are the same, and the convolution kernel size is 3 × 3. All the convolution operations in the whole network model have a step size of 1, and no BatchNormalization (BN) layer is used.

In summary, the denoising principle of the algorithm in this paper can be described as follows.

First, the noise-bearing seismic data $\boldsymbol{y}$ is input and the predicted noise estimate

is output after the noise distribution estimation subnet.

$$\begin{cases} \hat{\boldsymbol{\eta}} = F_{\mathrm{ES}}(\boldsymbol{y};\boldsymbol{\theta}_{\mathrm{ES}}) = R\left(w_5\boldsymbol{y}_{\mathrm{out}_4} + b_5\right) \\ \boldsymbol{y}_{\mathrm{out}_{i+1}} = R\left(w_i\boldsymbol{y}_{\mathrm{out}_i} + b_i\right)\ i = 1,2,3 \end{cases} \tag{1}$$

Where F_{ES} is the function of the noise estimation subnetwork; $\boldsymbol{\theta}_{\mathrm{ES}}$ is the set of parameters in the noise distribution estimation subnetwork; w_i is the weight parameter of layer i in the noise distribution estimation subnetwork; $\boldsymbol{y}_{\mathrm{out}_i}$ is the output of layer i; b_i is the bias parameter of layer i; R is the ReLU activation function.

Then, the noise estimate $\hat{\boldsymbol{\eta}}$ and the noisy seismic data $\boldsymbol{y}$ are simultaneously input into the denoising subnet, and the final denoised seismic data are obtained after output.

$$\begin{cases} \hat{\boldsymbol{x}} = F_{\mathrm{DS}}(\boldsymbol{y},\hat{\boldsymbol{\eta}};\boldsymbol{\theta}_{\mathrm{DS}}) = R\left(w_{17}\boldsymbol{y}_{\mathrm{out}_{16}} + b_{17}\right) \\ \boldsymbol{y}_{\mathrm{out}_{i+1}} = R\left(w_i\boldsymbol{y}_{\mathrm{out}_i} + b_i\right) \\ \boldsymbol{y}_{\mathrm{out}_5} = w_5 C\left(\boldsymbol{y},\boldsymbol{y}_{\mathrm{out}_4}\right) + b_5 \\ \boldsymbol{y}_{\mathrm{out}_1} = w_1 C(\boldsymbol{y},\hat{\boldsymbol{\eta}}) + b_1 \end{cases} \tag{2}$$

Where F_{DS} is the function of denoising subnet; $\boldsymbol{\theta}_{\mathrm{DS}}$ is the set of parameters in the denoising subnet; C is the vector stitching operation; $i = 1,2,\cdots,15$ and $i \neq 4$。

The data selected in this paper are 10,000 original noise-free data x of size 300 samples and 207 channels obtained after cropping, and the data set is divided into training set, validation set and test set according to the ratio of 80%, 10% and 10%, respectively. Gaussian random noise simulation with 0 mean positive distribution is added, and the noise standard deviation is defined as:

$$\sigma = l\sqrt{\frac{1}{MN}\sum_{t=1}^{M}\sum_{s=1}^{N}\left(\boldsymbol{x}_{t,s} - u\right)^2} \tag{3}$$

Where M is the total number of sliced time samples; N is the total number of sliced seismic channel samples; t is the time sampling sequence number; s is the seismic channel record sequence number; u is the mean value of seismic data; l is the scale factor of noise intensity, and l is set in the range of $0.01 \sim 0.03$ during the training process.

The peak signal to noise ratio (PSNR) is used as a measure of the denoising effect, and the corresponding expression for a single sample is:

$$\mathrm{PSNR} = 20\lg\frac{\mathrm{m}\ (|\boldsymbol{x}|)}{|\boldsymbol{x} - \hat{\boldsymbol{x}}|} \tag{4}$$

4.2 Modeling of question two

4.2.1 Automatic seismic horizon tracking based on full convolutional neural network

(1) Deep learning and neural networks

Convolutional neural network is the most common and practical neural network in deep learning. Its network structure generally consists of a different number of convolutional layers, pooling layers and fully connected layers. In the process of network training, appropriate activation functions, loss functions and optimization algorithms should be selected according to different target tasks. In this section, the basic structure of neural network and other basic operations of network construction will be introduced.

(2) Basic structure of neural network

Inspired by biological neural networks, neural networks establish network models with a certain learning ability through a large number of connections to achieve complex operations. As shown in Figure 9， The input data $x_1, x_2, \cdots, x_n$ are the inputs of the neural network, which simulates the received signals in the biological neural network. The connection weights $w_1, w_2, \cdots, w_n$ are the weights corresponding to each input data, and the synapses of different receiving strengths in the neurons of biological neural network are simulated. The sum $\sum$ simulates the integration and summary of the received information by neurons in the biological neural network. Bias b simulates neurons with different sensitivities in biological neural networks; The activation function φ simulates the cumulative triggering of action potentials in the biological neural network, and the output Y simulates the new number released by neurons in the neural network. Through such a large number of linear and nonlinear connections to simulate the connection mechanism between neurons, so as to further simulate the brain's information transmission.

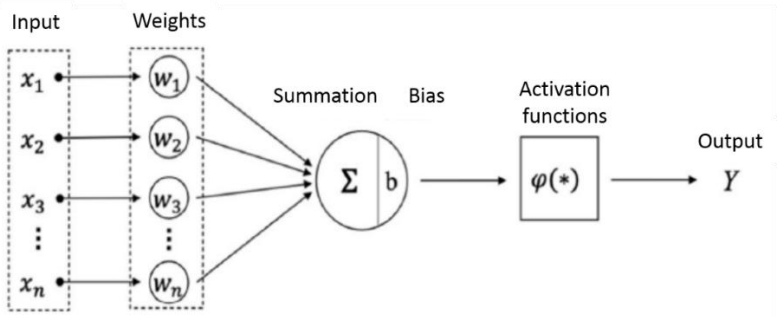


Figure 9 A neural network model corresponding to a single neuron

The basic structure of neural network is shown in Figure 10,Generally including input layer, output layer and a number of hidden layers in the middle. Each circle in the hidden layer represents a neuron. In the process of forward propagation, each neuron will go through the action of activation function, and the output of each layer will be used as the input of the next layer until the output layer is reached. Then the error between the predicted value and the real value is measured by the specified loss function calculation formula, and the back propagation algorithm is used to guide the updating of each parameter in the neural network under the goal of decreasing the error of the loss function calculation. When the loss function drops to the target value or the number of iterations reaches the specified value, the parameter update will be stopped, the network training will be completed, and the prediction can be made according to the trained network.

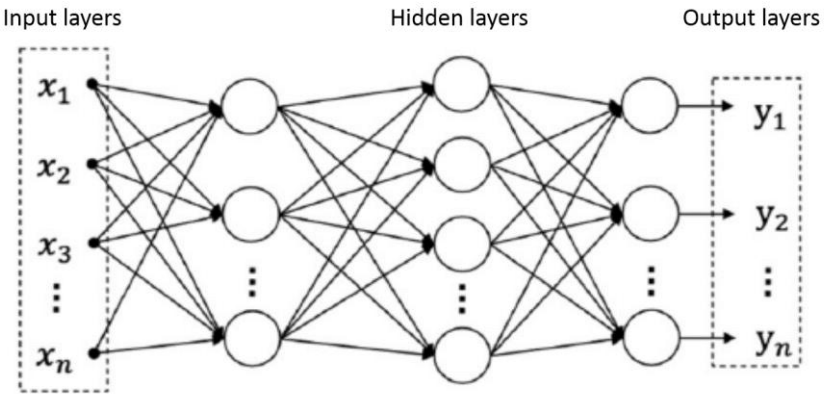


Figure 10 Neural network structure

The neural network can fit and approximate complex nonlinear functions through the connection of multiple neurons. By increasing the number and depth of neurons in the hidden layer, the results of the network will become more complex and the approximable functions will also become more complex, but the corresponding calculation amount will also increase.

(3) Convolution and pooling

The convolution operation is an important physical concept in many diverse scientific fields. Mathematically, the convolution operation of a function f and a function g is defined as follows.

If the functions f and g are continuous functions, the expression for the convolution operation is：

$$f(t) * g(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{5}$$

If the functions f and g are discrete functions, the expression of the convolution operation is：

$$f(t) * g(t) = \sum_{\tau=-\infty}^{\infty} f(\tau)g(t-\tau) \tag{6}$$

In the field of computer vision, usually the digital image is a relatively large two-dimensional or multidimensional matrix, and the convolution kernel is a small matrix, such as 3×3, 5×5, etc. The convolution operation of an image is to slide the convolution kernel on the image, multiply the elements of the corresponding positions in the two matrices and sum them, and use the summation result as the new value corresponding to the original several pixels.

The convolution operation can extract features from the data, and its main parameters include the size of convolution kernel $f \times f$, the number of feature maps obtained after convolution of different convolution kernels O. For the original image of size $x \times x$, the convolution operation is performed after expanding (padding) k zero pixels to obtain a new image of $C \times N \times N$, where $N = (x + 2P - f)/S + 1$, and S is the convolution kernel move step The specific calculation example is shown in Figure 11.

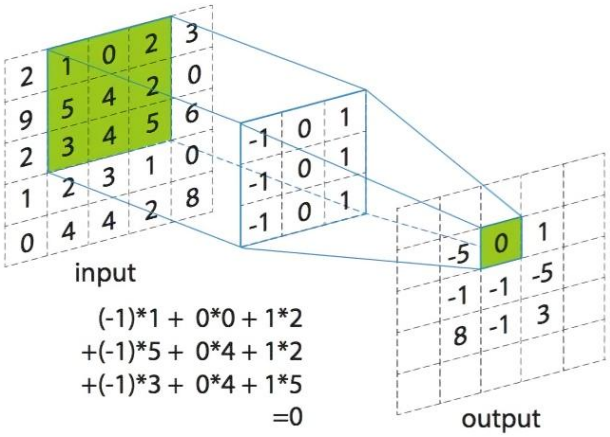
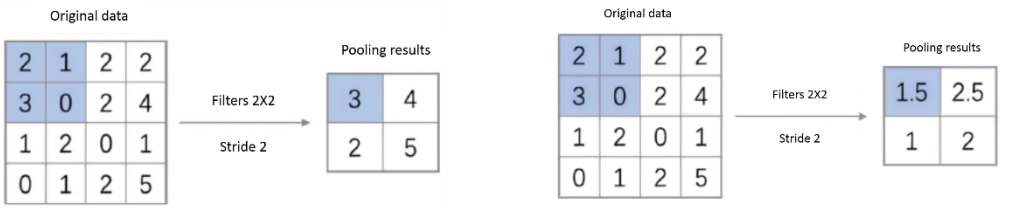


Figure 11 Convolution calculation example

In the feature extraction process, the convolution operation is usually used in conjunction with the pooling operation, which reduces the dimensionality of the data by simulating the human visual system and makes the extracted features more focused by reducing redundant information. Pooling, also known as subsampling or downsampling, is usually used after the convolutional layer in the neural network structure. The pooling operation reduces the dimensionality of the data and prevents overfitting while reducing the network parameters.

The common pooling operations are average pooling and maximum pooling, and the convolution feature graph is further processed by pooling function. Pooling operation will statistically summarize the values of a position on the feature graph and its adjacent positions within a certain range in a certain way, and the summarized value will be the new value of the position. The statistical method of maximum pooling is to take the maximum value in a certain area of the position as the value of the position, while the average pooling is to take the average value in a certain area of the position as the value of the position. Pooling reduces dimension only by reducing the height and width of the data matrix without changing the depth of the data matrix.

Similar to the convolution operation, the size of the feature map after the pooling operation is also related to the size of the filter and the filter's moving step. For the original feature image with the size $x \times x$, k zero pixels are expanded and the pooling operation is carried out to obtain a new $N \times N$ feature image, where $N = (x + 2P - f)/S + 1$, f is the size of the pooling filter, and S is the movement of the pooling filter Step size. The specific calculation example is shown in Figure 12.



a) Example of maximum pooling calculation b) Example of average pooling calculation

Figure 12 Pooling operation calculation example

(4) Activation function

Convolution and pooling operations in neural networks are essentially linear operations on the input of the upper layer, and the operation process only includes addition and multiplication. If there are only convolution, pooling and full connection layers in a network, the network can only approximate some linear functions with simple operations. However, problems in real life usually consist of complex nonlinear functions, so researchers introduce nonlinear mechanisms into neural networks by activating functions. The activation function is usually located after the linear operation of each part, and the expression and generalization ability of the network are enhanced by nonlinear mapping, so that it can approximate more complex functions.

Relu function proposed by researchers is a linear and unsaturated activation function, which is also the most widely used activation function in recent years. Its mathematical expression is as follows:

$$\mathrm{Relu}(x) = \mathrm{m}\ \{0, x\} \tag{7}$$

The image of Relu function and its derivative function is shown in Figure 13. Although the Relu function is linear respectively in the positive and negative intervals, it is a nonlinear function on the whole.

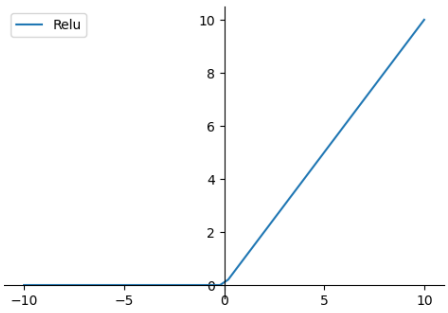


Figure 13 Image of Relu function

(5) Loss function

The essence of neural network is to give a series of training samples (x_i, y_i), and learn the mapping relationship between $x \to y$ by building a network structure. The ultimate goal is to give an x that is not in the training sample, and the output y after network calculation $\hat{y}$ .As close as possible to the true value y. In this process, the difference between the output $\hat{y}$ of the network model and the true value y is measured by the loss function, which is used to guide the optimization direction of the model.

The output value of the loss function reflects the prediction ability of the network model, and its output value is usually positive, and the closer the predicted value of the neural network is to the real value, the closer the calculated value of the loss function is to zero. For different model tasks, the emphasis of the loss function should be different. Selecting the appropriate loss function can also help the neural network achieve better results in the training process.

Cross entropy loss function. For an event x, probability $p(x)$ refers to the

probability of its occurrence. In information theory, information refers to the uncertainty of an event. The greater the probability, the greater the uncertainty, and the smaller the information. For event x, the amount of information to the $I(x) = -\log(p(x))$. The information entropy is expected for the information of the whole system, information is the information of a single sample, the information entropy is the average information of the whole system. For a system $X = [x_0, x_1, \cdots, x_{N-1}]$, the information entropy is:

$$H(p) = -\sum_{i=0}^{N-1} p(x_i)\log\big(p(x_i)\big) \tag{8}$$

For the binary classification problem, let the given input data be x_i, the probability that the model output is judged to be positive is $\hat{y}_i$, and the true category be y_i (y_i scalar is 0 or 1), then the expression of the cross entropy loss function is as follows:

$$L_{CE} = -\sum_{i=1}^{N} \big(y_i\log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\big) \tag{9}$$

If there are multiple classification problems, the general form is:

$$L_{CB} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{i,k}\log\hat{y}_{i,k} \tag{10}$$

Among them, N is the total number of sample points, K is the total number of categories, $y_{i,k}$ is the true label of the ith sample for the kth category, $\hat{y}_{i,k}$ is the prediction value of the ith sample for the kth category.

(6) Optimization algorithm

The training process of neural network is to constantly update the weight parameters in the network model in a certain way, so that the predicted value of the network is constantly approaching the true value, and the loss function is gradually approaching zero, among which the parameter update method is the optimization algorithm.

Adam algorithm，The gradient descent method introduces an adaptive learning rate. The learning rate is adaptively adjusted through each training result. With the same amount of data, the memory consumption is less. At the same time, the gradient descent method is more stable in the case of sparse gradient and noise. It is applicable to scenes with large amount of data. Its parameter update formula is：

$$\begin{cases} m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta L(\theta_t; x_i; y_i) \\ v_t = \beta_2 v_{t-1} + (1-\beta_2)\nabla_\theta L(\theta_t; x_i; y_i)^2 \\ \hat{m}_t = \dfrac{m_t}{1-\beta_1^t} \\ \hat{v}_t = \dfrac{v_t}{1-\beta_2^t} \\ \theta_{t+1} = \theta_t - \dfrac{\alpha}{\sqrt{\hat{v}_t + \epsilon}}\hat{m}_t \end{cases} \tag{11}$$

where β_1 and β_2 are the decay rate constants close to 1, m_t and v_t are the first-order moment estimates and second-order moment estimates of the gradient,

respectively, $\hat{m}_t$ and $\hat{v}_t$ are the bias-corrected first-order distance estimates and second-order moment estimates, respectively, and α is the iteration step size.

(7) Automatic 2D seismic layer tracking based on U-net

The U-net network is a modified network of the full convolutional network. The whole network model can be viewed as consisting of two parts: the encoder extracts features by downsampling and the decoder recovers the extracted feature details by upsampling. The network consists of convolution, pooling, and upsampling, and does not contain a fully connected layer.

The structure of the U-net network is shown in Figure 14. The entire network model consists of a systolic path on the left and an extended path on the right, in a U-shaped structure. The original U-net network has 23 convolutional layers with a network depth of 5, and contains 4 downsamples as well as 4 upsamples. The input is 256×256 seismic data and the output is 256×256 seismic layer labels.

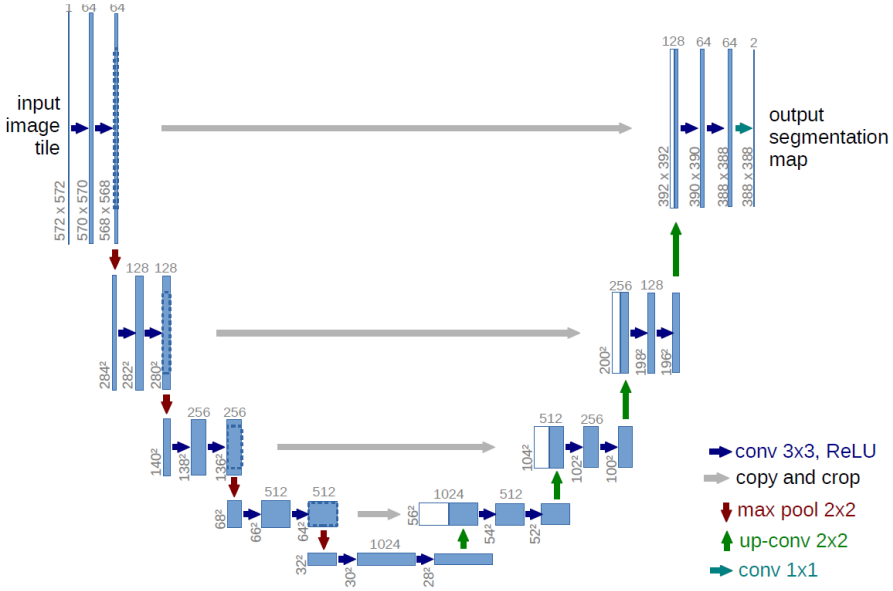


Figure 14 U-net network structure diagram

The contraction path on the left side of the network model is the encoder, follows the classical convolutional neural network architecture, consisting of two 3 × 3 convolutions repeatedly applied, and the Same model is used in this paper, i.e., the data size does not change before and after convolution. After each convolution is a nonlinear mapping with Relu function, and after both convolutions are completed a 2×2 maximum pooling operation is performed to downsample in steps of 2. After pooling, the data size is reduced to half of the original size, thus obtaining more sparse and important features, and being able to reduce the model parameters and make the network structure simpler. The number of feature channels varies from 64, 128, 256, 512, and 512 for each depth convolution layer.

The extended path on the right side of the network model is the decoder, and each part is upsampled from the features of the previous step, mainly by expanding the feature size to twice the original size through deconvolution, and the number of feature channels should be halved correspondingly. Then the features obtained by downsampling at the corresponding depth are fused with the upsampling results in a

stitching manner by jump-join, and then the deep features of the data are extracted by two convolutional operations with an activation function of Relu of 3×3. Since the shallow features of the data are acquired by downsampling, this jump-join approach enables better fusion of the deep and shallow features of the data and recovery of feature details by upsampling. The probability that each sampled point is a layer is then returned by a 1 × 1 convolution at the last layer. After the downsampling operation, the feature size of the data is usually reduced to compared with the original data, while upsampling refers to the mapping operation that restores the features of the data extracted by downsampling to the same size as the original data, realizing the resolution from small to large, and its common ways include interpolation, deconvolution and inverse pooling. In this paper, we use the inverse convolution upsampling which can learn the parameters through network training.

For the deconvolution, the low-dimensional data O is known, and to get the high-dimensional result X, it is sufficient to multiply O by a matrix of the same size as the transposed dimension of F. This is where the deconvolution is called transposed convolution.

As shown in Figure 15, for the original 2×2 data, the data size can be converted to 4×4 by zeroing padding=2, moving step stride=1, and convolution kernel of 3×3, which completes the resolution improvement.

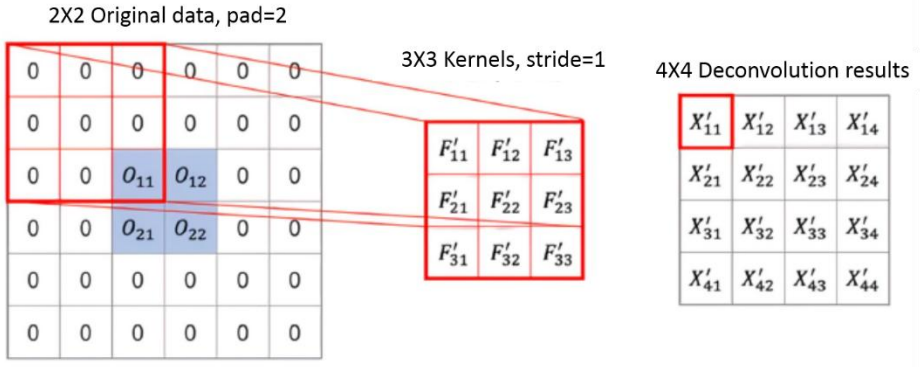


Figure 15 Schematic diagram of the deconvolution operation

Regarding the loss function, the cross-entropy loss function mentioned in the original U-net network is chosen:

$$L_{CE} = \sum_{K} \left(y_k \log(\hat{y}_k) \right) \tag{12}$$

K is the total number of categories, y_k is the true category of the indicated sample,if the input sample is category k,then y_k takes 1,otherwise it takes 0; $\hat{y}_k$ indicates the probability that the input sample is predicted to belong to sample k.

The gradient of each parameter can be obtained by the chain rule and the derivative of the loss function, so that the gradient can be updated according to the optimization algorithm. In this paper, the Adam optimization algorithm is used for training until the network training reaches the set number of iterations or the degree of decline of the loss function reaches the threshold ε, then the training is stopped, and the trained network model parameters can be used to predict the layers based on the seismic data.

4.3 Modeling of question three

In view of the problems such as poor continuity of seismic horizon and easy cross-bedding of fault location traced by seismic DNA algorithm, this paper introduces the clustering method to improve the seismic DNA algorithm. Firstly, the clustering method is used to classify the peaks found by seismic DNA algorithm, and then the Euclidean distance is used to connect the good clustering points, and the coherence algorithm is used to find and shield the fault area, so as to improve the continuity and accuracy of seismic DNA algorithm to pick up seismic layers.

4.3.1 Cluster

The clustering method is used to classify the scattered points found by seismic DNA algorithm. First, a search is made for scattered points on channel 1. If n points are found, they are temporarily divided into n categories (A1,...,An), and then go through the second pass. When traversing the second channel, first judge whether the found scatter belongs to one of the n classes of the first channel. Because it is a match between adjacent channels, time difference Δt can be used to judge whether it belongs to the same class. If the time difference is less than the threshold, it indicates that the time difference belongs to the same class and is placed in the corresponding class. If more than one number is less than the threshold, it is placed in the category with the smallest difference. If there is no number less than the threshold, a new class An+1 is added and placed in An+1. Go all the way to the last seismic track, so that you can connect successive and adjacent points in each seismic track. As shown in Figure 16 (a).

4.3.2 Connectivity

The clustering method has been used to connect the scattered points with close distances. In order to further improve the continuity, the points that are far away and on the same seismic horizon are connected. If there is no fault, generally speaking, the seismic horizon is continuous, so the Euclidean distance between seismic waves is an important basis for determining whether the seismic wave is on the same seismic horizon. Therefore, the Euclidean distance is selected as the constraint condition for connecting distant points.

When connecting, first of all, the coordinates of the first point and the last point of each class are recorded, and then the coordinates of the last point of each class and the coordinates of the first point of other classes are calculated Euclidean distance. The coordinates of points in two-dimensional time domain seismic data are represented by time and channel number, and the European distance formula is:

$$d(J,T) = \sqrt{(J_r - J_l)^2 - (T_r - T_l)^2} \tag{13}$$

In the formula, J_r and T_r are the track numbers and time of the rightmost seismic track points of the class, while J and T are the track numbers and time of

the leftmost seismic track points of other classes. After calculating all the Euclide distance, threshold values need to be set. In order to avoid cross-layer, the scope of time is generally set smaller, and the scope of Tao is set larger. The shortest distance principle is used to find the value with the smallest distance within the threshold range and connect it. The connection effect is shown in Figure 16(b). It can be seen from Figures 16(a) and 16(b) that the cluster connection method improves the continuity of layers.

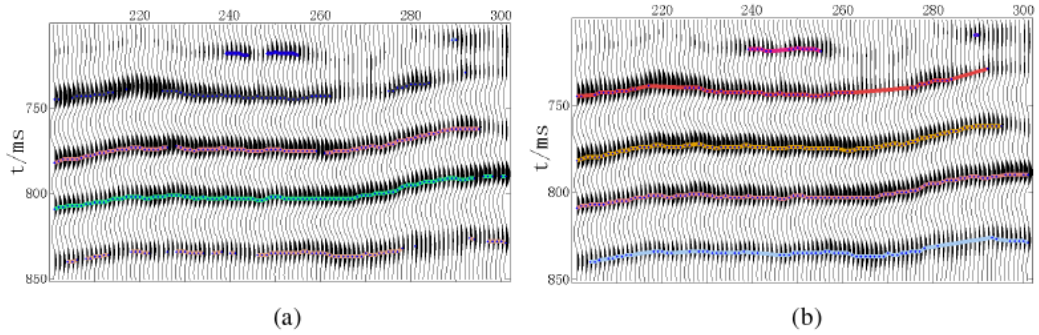


Figure 16. Clustering and connection effect

4.3.3 Coherent algorithm based on eigenstructure

C3 coherence algorithm is also known as the coherence algorithm based on the proof structure. This algorithm calculates the coherence value through the eigenvalue of the covariance matrix in the time window. This method converts the high-dimensional data matrix to the low-dimensional space by using the linear transformation, so as to extract the information that best represents the original data.

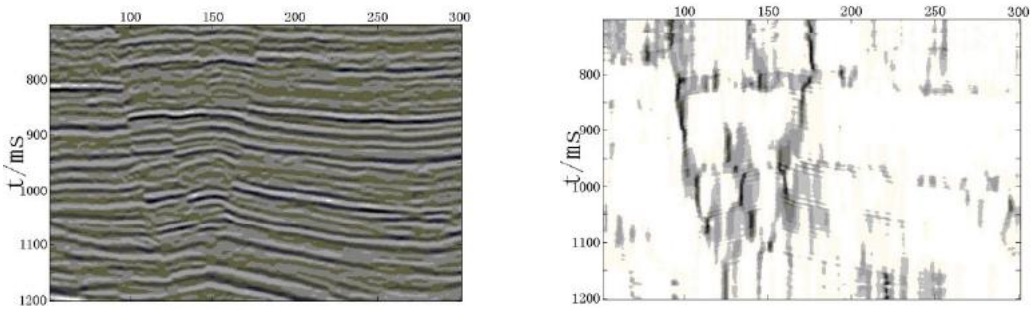
Firstly, an analysis window containing J seismic tracks is also set, and N sampling points are selected for each seismic track. The obtained matrix is:

$$D = \begin{bmatrix} d_{11} & d_{12} & \cdots & d_{1J} \\ d_{21} & d_{22} & \cdots & d_{2J} \\ \vdots & \vdots & \vdots & \vdots \\ d_{N1} & d_{N2} & \cdots & d_{NJ} \end{bmatrix} \tag{14}$$

The covariance of the matrix is calculated. If the data of the covariance matrix is basically zero except the diagonal data, it means that the data of the J seismic tracks are basically irrelevant. It can be seen that the part with low energy value can be removed after the matrix is diagonalized, so the noise has little influence on the method. The data on the diagonal are the eigenvalues of the matrix. If (j=1,2... J) is the JTH eigenvalue of the covariance matrix, where 2 is its maximum eigenvalue. Then the formula of the C3 coherence algorithm is as follows:

$$C_3 = \frac{\lambda_1}{\sum_{j=1}^{J} \lambda_j} \tag{15}$$

The C3 coherence algorithm uses the covariance matrix to complete the coherence value calculation. This method compresses the matrix and compresses the part with low energy value, leaving only the part with high energy value, so the algorithm can effectively suppress the noise. The processing results of C3 coherence algorithm are shown in Figure 17.



a) Data to be detected b) C3 detection results

Figure 17. Processing result diagram of C3 coherence algorithm

4.4 Modeling of question five

The layers in the 3D seismic image are distributed in a laminar pattern. Figure 18 shows the extraction of a 2D profile data from the 3D seismic image.

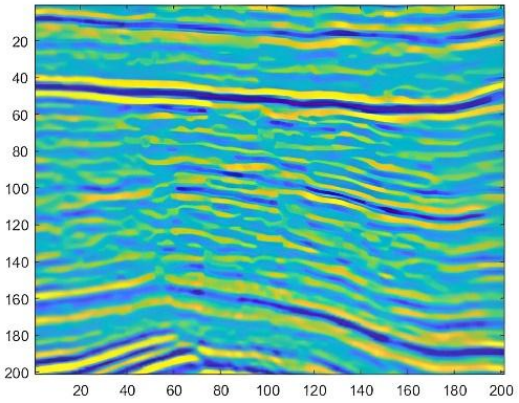


Figure 18 2D profile of 3D seismic image

The color highlighted in Figure 22 is the area of large amplitude in the seismic waveform data, which is the location of the layers. The amplitude of the layers is not exactly the same, and the overall amplitude of some layers is larger than that of other layers. The matching search-based full-layer tracking method is based on this longitudinal distribution of layers for layer tracking, and the specific flow of the algorithm is shown in Figure 19.

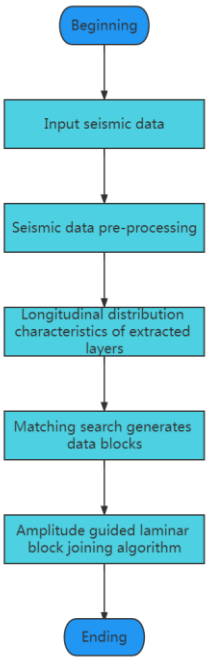


Fig. 19 Flow chart of the full layer tracking method based on matching search

The full-layer tracking method based on matching search makes full use of the longitudinal distribution characteristics of layers and the amplitude gap between different layers.

amplitude gap between different layers, and consists of four main steps.：

(1) Data pre-processing: The first step of the algorithm is to pre-process the data of the input 3D seismic amplitude data. The data processing process is consistent with the noise reduction process of Problem 1 data, which effectively reduces the noise on the basis of retaining the original structural and detail information.

(2) Layer longitudinal distribution feature extraction algorithm: The 3D seismic amplitude data is not convenient for feature extraction directly, so the amplitude data is firstly discretized into several specific values because the seismic waveform can be seen as a continuous curve, and these specific values of discretization will generally appear continuously in a certain interval. The algorithm extracts the distribution of these values. The flow of the layer longitudinal distribution feature extraction algorithm is shown in Figure 20.

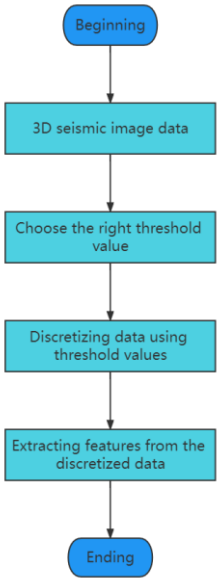


Figure 20 Flow chart of the feature extraction algorithm for longitudinal distribution of layers

(3) Matching search-based data block generation algorithm: Based on the layer longitudinal distribution feature extraction algorithm can extract the layer longitudinal distribution features of a specific region, match search around the region to search all the data areas that meet this feature, and then continue to feature extraction and match search in the searched region, so that the iterative process will continue to divide the entire data body into several data blocks according to the layer longitudinal distribution The features are divided into several data blocks. The specific flow chart of the matching search-based data block generation algorithm is shown in Figure 21.

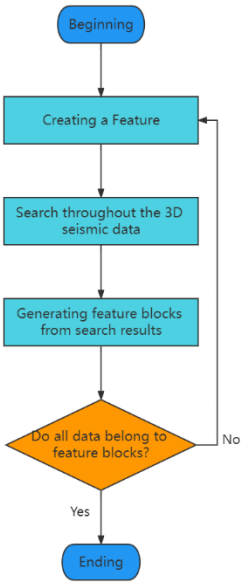


Figure 21 Flow chart of data block generation algorithm based on matching search

(4) Amplitude oriented data block connection algorithm: Different layers may have certain differences in amplitude, and the amplitude oriented data block connection algorithm is mainly based on the difference of amplitude distribution between different layers for layer block connection, connecting the data belonging to

the same layer in different layer blocks to form the layers in 3D space, so that the layers may have some missing data areas, and then expand the layers based on the location and amplitude of the layers to cover the missing data areas after the data block connection is completed.

5. Analysis of model results

5.1 Analysis of question one results

In order to verify the robustness of the algorithm in this paper, the algorithm in this paper was trained synchronously in comparison with the algorithm without noise subnets, which was implemented using the deep learning framework Pytorch and accelerated using CUDA for training, and the model was trained for 100 iterations on Windows 10 operating system, resulting in the change curve of PSNR as shown in Figure 22.

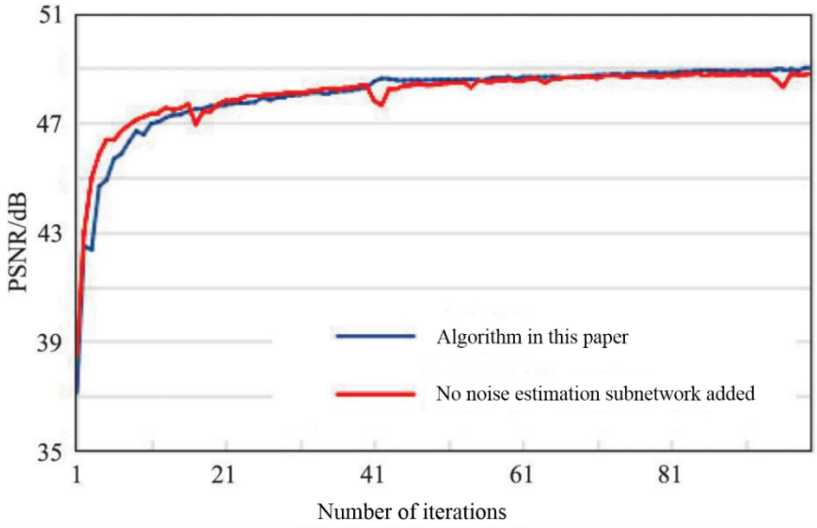
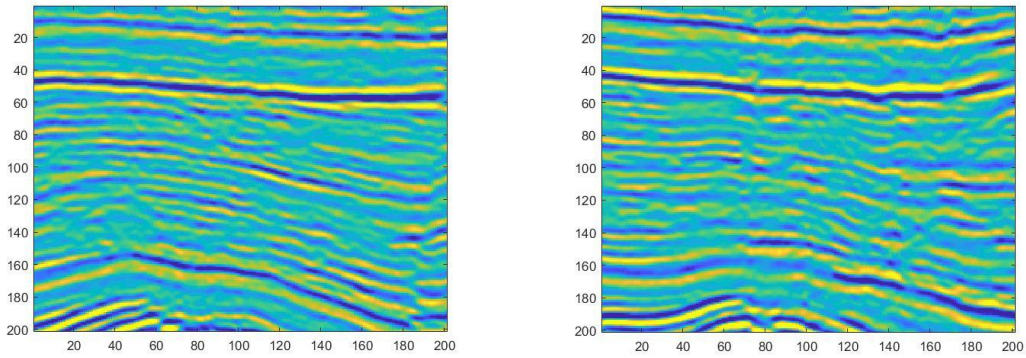


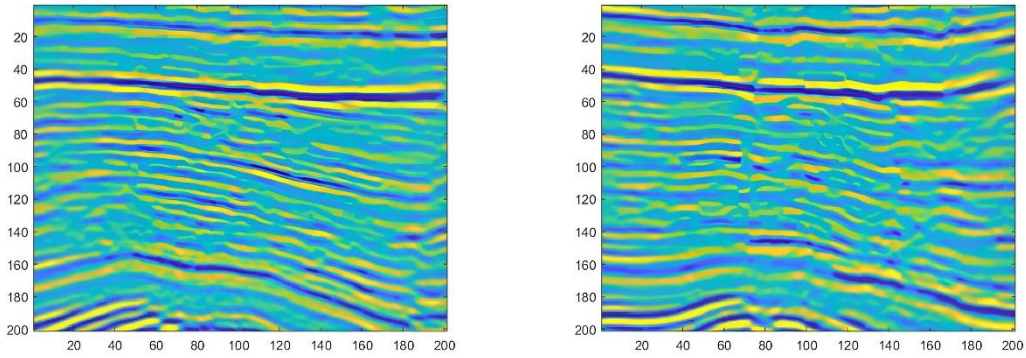
Figure 22 Model convergence diagram

As seen in Fig. 22, the PSNR curves of the noise-free estimation subnetwork have multiple fluctuations, and the PSNR curves of the algorithm in this paper converge relatively steadily, so the algorithm in this paper has the advantages of good robustness and strong denoising ability for seismic data.

The noisy seismic data of the topic are input into the deep learning denoising algorithm model of this paper, and the pre-trained parameters are imported into the model for seismic data denoising, resulting in a comparison graph of denoising results as in Figure 23.



(a)Raw seismic data with noise



(b)Denoised seismic data

Figure 23 Comparison of denoising results

As shown in Figure 23, the original seismic data in (a) contains a large amount of noise; the seismic data in (b) is the output of the denoising algorithm in this paper; comparing the two data, we can see that the denoising algorithm proposed in this paper can remove most of the original seismic data noise and produce clear seismic layer edges, which greatly improves the visual effect of the seismic data.

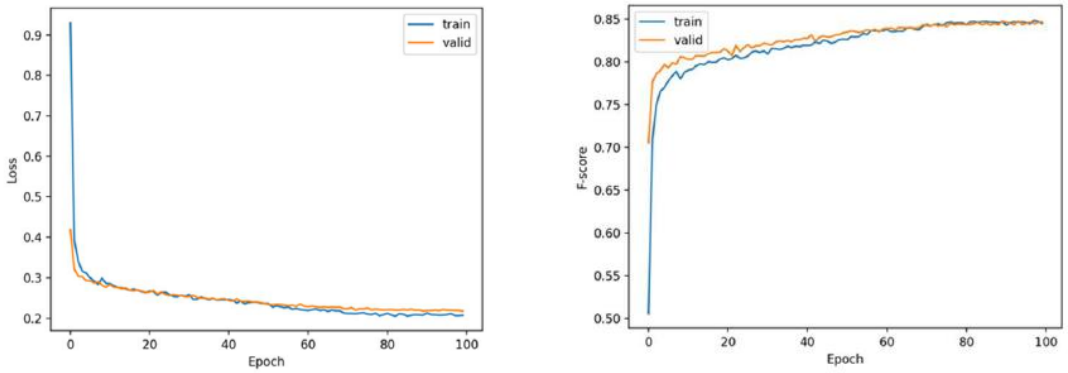
5.2 Analysis of question two results

The experimental data in this experiment are all from the seismic data described in Seismic Data. Firstly, a total of 201 sets of 2D data were used in the training phase, and they were divided into training and validation sets in a ratio of 9:1 to complete the whole training process. In order to expand the data, the training data were cut, rotated, and added random noise with certain probability. The test phase selected 20 sets of data for the participating training as the test data, and the average MIOU and ACC values of all test data were output. The seismic data were used as the input and the real layers were used as the corresponding labels for supervised learning training, and this experiment was trained 100 times with a batch size of 5.

Since the last layer of the network is a softmax activation function, its output is a value between 0 and 1, which indicates the probability value that the sampled point is a layer loci. And the final result should only wrap the two cases of layer loci or

non-loci, so a threshold needs to be set for the division of the final result. In this paper, we observe its effect on the accuracy of the validation set by setting different thresholds, and finally measure that the validation set accuracy is highest in the U-net network when the threshold value is set to 0.39. Therefore, the following test is performed by setting the softmax threshold to 0.4 and using the trained network.

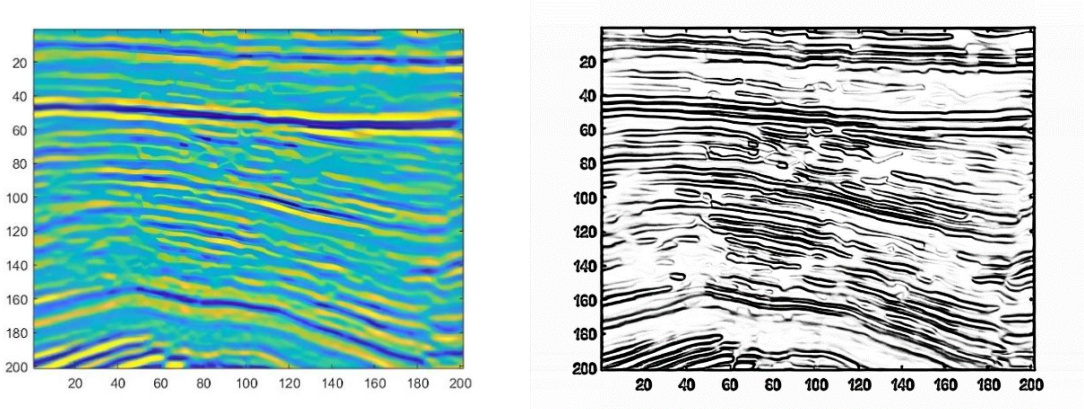
uring the testing phase, the U-net network was tested on 20 sets of seismic data, and the average MIOU value of the output was 61.58 and the average ACC value was 94.72. Figure 24 shows the network convergence during the training of one of the seismic data. It can be seen from Fig. 28 in Fig. a) and Fig. b) that as the network is trained, the loss function value of the network keeps decreasing and the F1 score keeps increasing, and the two loss values and F1 scores gradually level off at the completion of the network training without continuing to decrease or increase significantly, indicating that the network has converged at the end of the training.



a) Loss function change b) F1 fractional transformation

Figure 24 Convergence of U-net 2-D layer training

Figure 25 shows the prediction results compared with the real picture. Compared with the original data in Figure (a), the layer labels obtained by U-net network prediction (b) show that most of the layer information is predicted accurately, but there are still discontinuities and there are layer points that are not predicted correctly.



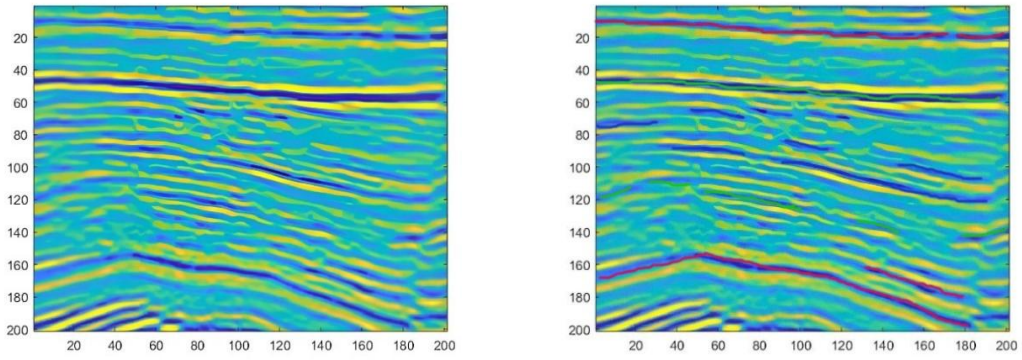
(a) Raw data after noise reduction (b) U-net prediction results

Figure 25 U-net 2D layer prediction results

5.3 Analysis of question three results

Some of the denoising seismic data in the topic are input into the improved seismic DNA algorithm for the test. First, the median filtering algorithm and Laplace algorithm are used to process the seismic data, and then the seismic amplitude information is normalized to ensure that the amplitude range is between [-1.0,1.0]. Then the amplitude space according to the transformation rules [1.0, 0.2) b [0.2, 0.2] c (0.2, 1.0] into space characters.

Set the regular expression for searching to a{0,9}b{1,3}c{4,10}b{2,5}a{0,11}, and search for a string sequence that meets the conditions. Next, the data found by seismic DNA algorithm is processed by the method of cluster connection. Firstly, the data space is divided into a thousand 49x49 matrices, and C3 coherent operation is performed on them. Then, the threshold of Δt is set as (0,5) to complete the clustering, and the threshold of d, x and y is set as (0,6), (0,15) and (0,5) to complete the connection. Finally, the seismic horizon is obtained, as shown in Figure 26.



a) Seismic data after denoising

b) Improved seismic DNA algorithm prediction results

Figure 26. Seismic horizon processed by improved seismic DNA algorithm

5.4 Comparison of model results for questions two and three

From the comparison of the results of the algorithm models of Problem 2 and Problem 3, it can be seen that the advantage of the deep learning-based image edge detection algorithm is that it can identify multiple seismic layers at one time with high efficiency. However, the disadvantages of this method are that it is influenced by noise, subjective, some edges are not clear enough, and the identified seismic layers are cluttered. The reason for this problem is that the deep learning method requires a large amount of data for training in order to train an accurate model, and the problem that some of the edge parts are not clear appears because the training accuracy of the model is limited due to the limited data.

However, the advantages of the improved seismic DNA algorithm are that multiple seismic layers can be picked up simultaneously, the traced seismic layers are

more continuous, and no cascading occurs at fault locations. The search can be performed for many different alignment patterns using regular expressions, which is beneficial for capturing complex features of seismic data, more flexible, and does not limit the input data type, i.e., it can be processed using any attribute that is helpful for identification, and can also use multiple attributes for simultaneous identification. It is characterized by high convenience and fast accuracy rate. Relative to deep learning methods, this algorithm does not require a large amount of data for model training, and it is more objective, more credible, and more applicable to the problem of automatic seismic layer tracking.

5.5 Analysis of question five results

In order to verify the layer tracking effect of the full layer tracking method based on matching search in the 3D seismic data body, Seismic Data data is selected for testing in this paper. Figure 27 shows the automatic tracking effect of this algorithm. It can be seen that this algorithm has outstanding effect in 3D seismic data layer tracking.

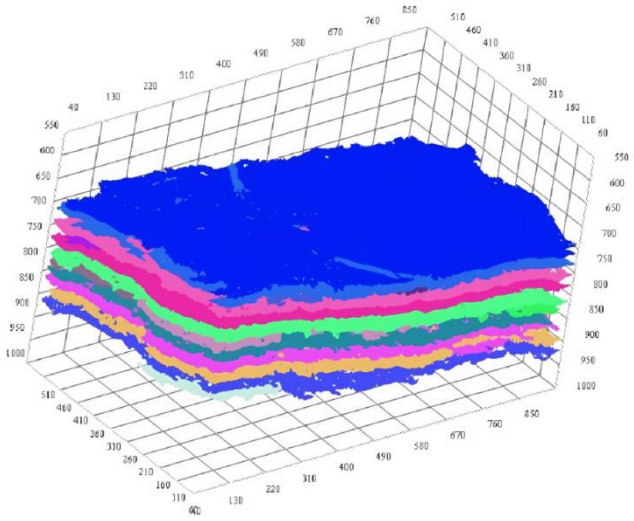


Figure 27 Automatic tracking effect of full layer position

6. Conclusion

In this paper, we focus on algorithms for automatic seismic stratigraphic tracking. We first use a robust deep learning denoising algorithm to denoise the initial seismic data, and then build an automatic seismic stratigraphic tracking algorithm for the denoised seismic data. For problem 2, we mainly use the full convolutional network in deep learning to achieve automatic seismic stratigraphic tracking. This method can identify multiple seismic strata at one time with high efficiency, but it is more affected by noise, more subjective, some edges are not clear enough, and the identified seismic strata are cluttered, so in problem 3 we propose an improved seismic DNA algorithm

that introduces the clustering method into the The improved algorithm has the ability of capturing complex features of seismic data, flexibility and convenience with high accuracy rate. Finally, the denoised seismic data are used to build an automatic tracking model based on correlation and waveform 3D levels, and the results are analyzed.

References

[1] Zheng Gongying , Zeng Tingting . Research on automatic seismic stratigraphic tracking technology [J]. Computational Technology of Physical and Chemical Exploration ,2013,35(6): 711-716.

[2] Yang L. Q. Research on automatic 3D seismic layer tracking technology in oil and gas exploration[J]. Henan Science and Technology:First Half of the Month,2014(3): 22-23.

[3] Xu Jingxin,Liu Xiande. A 3D seismic reflection surface automatic tracking algorithm and its application[J]. Journal of Daqing Petroleum Institute,2007,31(4):73-75,94

[4] Yang W, Chen K-Y. Detection of seismic reflection homogeneous axis using image refinement algorithm[J]. Complex Oil and Gas Reservoirs,2011,4(2):31-34

[5] Zhaoge Su. Research on full-layer tracking method of 3D seismic images based on multi-level framework [D]. University of Electronic Science and Technology, 2015.

[6] Li Xuefeng, Yan Jianguo, Zhao Zhou, Yao Shuang. Stratigraphic interpretation using coherent attribute profile features[J]. Computational Techniques in Physical and Chemical Exploration,2011,33(02):134-139+106.

[7] Zhang Lin,Yuan Feiniu,Zhang Wenrui,et al. A review of research on fully convolutional neural networks[J]. Computer Engineering and Applications,2020,56(1):25-37.

[8] Cai Yufei. Intelligent Identification and Reconstruction of Three-Dimensional Earthquake Fault [D]. University of Electronic Science and Technology. 2020.

[9] Q. Wei,J. Tang. CUDA-based algorithm for automatic seismic data layer level tracking[J]. Computer Technology and Development,2012,22(9):1-4.

Appendix

```
""" Parts of the U-Net model """

import torch
import torch.nn as nn
import torch.nn.functional as F

class DoubleConv(nn.Module):
    """(convolution => [BN] => ReLU) * 2"""

    def __init__(self, in_channels, out_channels, mid_channels=None):
        super().__init__()
        if not mid_channels:
            mid_channels = out_channels
        self.double_conv = nn.Sequential(
            nn.Conv2d(in_channels, mid_channels, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(mid_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(mid_channels, out_channels, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.double_conv(x)

class Down(nn.Module):
    """Downscaling with maxpool then double conv"""

    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.maxpool_conv = nn.Sequential(
            nn.MaxPool2d(2),
            DoubleConv(in_channels, out_channels)
        )

    def forward(self, x):
        return self.maxpool_conv(x)

class Up(nn.Module):
    """Upscaling then double conv"""
```

```
    def __init__(self, in_channels, out_channels, bilinear=True):
        super().__init__()

        # if bilinear, use the normal convolutions to reduce the number of channels
        if bilinear:
            self.up = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)
            self.conv = DoubleConv(in_channels, out_channels, in_channels // 2)
        else:
            self.up = nn.ConvTranspose2d(in_channels, in_channels // 2, kernel_size=2, stride=2)
            self.conv = DoubleConv(in_channels, out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        # input is CHW
        diffY = x2.size()[2] - x1.size()[2]
        diffX = x2.size()[3] - x1.size()[3]

        x1 = F.pad(x1, [diffX // 2, diffX - diffX // 2,
                        diffY // 2, diffY - diffY // 2])

        x = torch.cat([x2, x1], dim=1)
        return self.conv(x)

class OutConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(OutConv, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=1)

    def forward(self, x):
        return self.conv(x)

""" Full assembly of the parts to form the complete network """

from .unet_parts import *

class UNet(nn.Module):
    def __init__(self, n_channels, n_classes, bilinear=False):
        super(UNet, self).__init__()
        self.n_channels = n_channels
        self.n_classes = n_classes
        self.bilinear = bilinear

        self.inc = DoubleConv(n_channels, 64)
        self.down1 = Down(64, 128)
```

```
        self.down2 = Down(128, 256)
        self.down3 = Down(256, 512)
        factor = 2 if bilinear else 1
        self.down4 = Down(512, 1024 // factor)
        self.up1 = Up(1024, 512 // factor, bilinear)
        self.up2 = Up(512, 256 // factor, bilinear)
        self.up3 = Up(256, 128 // factor, bilinear)
        self.up4 = Up(128, 64, bilinear)
        self.outc = OutConv(64, n_classes)

    def forward(self, x):
        x1 = self.inc(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
        x = self.up1(x5, x4)
        x = self.up2(x, x3)
        x = self.up3(x, x2)
        x = self.up4(x, x1)
        logits = self.outc(x)
        return logits

import torch
from torch import Tensor

def dice_coeff(input: Tensor, target: Tensor, reduce_batch_first: bool = False, epsilon=1e-6):
    # Average of Dice coefficient for all batches, or for a single mask
    assert input.size() == target.size()
    if input.dim() == 2 and reduce_batch_first:
        raise ValueError(f'Dice: asked to reduce batch but got tensor without batch dimension
(shape {input.shape})')

    if input.dim() == 2 or reduce_batch_first:
        inter = torch.dot(input.reshape(-1), target.reshape(-1))
        sets_sum = torch.sum(input) + torch.sum(target)
        if sets_sum.item() == 0:
            sets_sum = 2 * inter

        return (2 * inter + epsilon) / (sets_sum + epsilon)
    else:
        # compute and average metric for each batch element
        dice = 0
        for i in range(input.shape[0]):
```

```
            dice += dice_coeff(input[i, ...], target[i, ...])
        return dice / input.shape[0]

def multiclass_dice_coeff(input: Tensor, target: Tensor, reduce_batch_first: bool = False, epsilon=1e-6):
    # Average of Dice coefficient for all classes
    assert input.size() == target.size()
    dice = 0
    for channel in range(input.shape[1]):
        dice += dice_coeff(input[:, channel, ...], target[:, channel, ...], reduce_batch_first, epsilon)

    return dice / input.shape[1]

def dice_loss(input: Tensor, target: Tensor, multiclass: bool = False):
    # Dice loss (objective to minimize) between 0 and 1
    assert input.size() == target.size()
    fn = multiclass_dice_coeff if multiclass else dice_coeff
    return 1 - fn(input, target, reduce_batch_first=True)

import matplotlib.pyplot as plt

def plot_img_and_mask(img, mask):
    classes = mask.shape[0] if len(mask.shape) > 2 else 1
    fig, ax = plt.subplots(1, classes + 1)
    ax[0].set_title('Input image')
    ax[0].imshow(img)
    if classes > 1:
        for i in range(classes):
            ax[i + 1].set_title(f'Output mask (class {i + 1})')
            ax[i + 1].imshow(mask[i, :, :])
    else:
        ax[1].set_title(f'Output mask')
        ax[1].imshow(mask)
    plt.xticks([]), plt.yticks([])
    plt.show()

import torch
import torch.nn.functional as F
from tqdm import tqdm

from utils.dice_score import multiclass_dice_coeff, dice_coeff

def evaluate(net, dataloader, device):
    net.eval()
```

```
    num_val_batches = len(dataloader)
    dice_score = 0

    # iterate over the validation set
    for batch in tqdm(dataloader, total=num_val_batches, desc='Validation round', unit='batch', leave=False):
        image, mask_true = batch['image'], batch['mask']
        # move images and labels to correct device and type
        image = image.to(device=device, dtype=torch.float32)
        mask_true = mask_true.to(device=device, dtype=torch.long)
        mask_true = F.one_hot(mask_true, net.n_classes).permute(0, 3, 1, 2).float()

        with torch.no_grad():
            # predict the mask
            mask_pred = net(image)

            # convert to one-hot format
            if net.n_classes == 1:
                mask_pred = (F.sigmoid(mask_pred) > 0.5).float()
                # compute the Dice score
                dice_score += dice_coeff(mask_pred, mask_true, reduce_batch_first=False)
            else:
                mask_pred = F.one_hot(mask_pred.argmax(dim=1), net.n_classes).permute(0, 3, 1, 2).float()
                # compute the Dice score, ignoring background
                dice_score += multiclass_dice_coeff(mask_pred[:, 1:, ...], mask_true[:, 1:, ...], reduce_batch_first=False)



    net.train()

    # Fixes a potential division by zero error
    if num_val_batches == 0:
        return dice_score
    return dice_score / num_val_batches

import argparse
import logging
import sys
from pathlib import Path

import torch
import torch.nn as nn
```

```
import torch.nn.functional as F
import wandb
from torch import optim
from torch.utils.data import DataLoader, random_split
from tqdm import tqdm

from utils.data_loading import BasicDataset, CarvanaDataset
from utils.dice_score import dice_loss
from evaluate import evaluate
from unet import UNet

dir_img = Path('./data/imgs/')
dir_mask = Path('./data/masks/')
dir_checkpoint = Path('./checkpoints/')

def train_net(net,
              device,
              epochs: int = 5,
              batch_size: int = 1,
              learning_rate: float = 1e-5,
              val_percent: float = 0.1,
              save_checkpoint: bool = True,
              img_scale: float = 0.5,
              amp: bool = False):
    # 1. Create dataset
    try:
        dataset = CarvanaDataset(dir_img, dir_mask, img_scale)
    except (AssertionError, RuntimeError):
        dataset = BasicDataset(dir_img, dir_mask, img_scale)

    # 2. Split into train / validation partitions
    n_val = int(len(dataset) * val_percent)
    n_train = len(dataset) - n_val
              train_set, val_set = random_split(dataset, [n_train, n_val],
generator=torch.Generator().manual_seed(0))

    # 3. Create data loaders
    loader_args = dict(batch_size=batch_size, num_workers=4, pin_memory=True)
    train_loader = DataLoader(train_set, shuffle=True, **loader_args)
    val_loader = DataLoader(val_set, shuffle=False, drop_last=True, **loader_args)

    # (Initialize logging)
    experiment = wandb.init(project='U-Net', resume='allow', anonymous='must')
                    experiment.config.update(dict(epochs=epochs, batch_size=batch_size,
```

```
learning_rate=learning_rate,
                                  val_percent=val_percent,  save_checkpoint=save_checkpoint,
img_scale=img_scale,
                                  amp=amp))

    logging.info(f'''Starting training:
        Epochs:          {epochs}
        Batch size:      {batch_size}
        Learning rate:   {learning_rate}
        Training size:   {n_train}
        Validation size: {n_val}
        Checkpoints:     {save_checkpoint}
        Device:          {device.type}
        Images scaling:  {img_scale}
        Mixed Precision: {amp}
    ''')

    # 4. Set up the optimizer, the loss, the learning rate scheduler and the loss scaling for AMP
      optimizer = optim.RMSprop(net.parameters(), lr=learning_rate, weight_decay=1e-8,
momentum=0.9)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'max', patience=2) # goal:
maximize Dice score
    grad_scaler = torch.cuda.amp.GradScaler(enabled=amp)
    criterion = nn.CrossEntropyLoss()
    global_step = 0

    # 5. Begin training
    for epoch in range(1, epochs+1):
        net.train()
        epoch_loss = 0
        with tqdm(total=n_train, desc=f'Epoch {epoch}/{epochs}', unit='img') as pbar:
            for batch in train_loader:
                images = batch['image']
                true_masks = batch['mask']

                assert images.shape[1] == net.n_channels, \
                    f'Network has been defined with {net.n_channels} input channels, ' \
                    f'but loaded images have {images.shape[1]} channels. Please check that ' \
                    'the images are loaded correctly.'

                images = images.to(device=device, dtype=torch.float32)
                true_masks = true_masks.to(device=device, dtype=torch.long)

                with torch.cuda.amp.autocast(enabled=amp):
```

```
                masks_pred = net(images)
                loss = criterion(masks_pred, true_masks) \
                    + dice_loss(F.softmax(masks_pred, dim=1).float(),
                                F.one_hot(true_masks, net.n_classes).permute(0, 3, 1, 2).float(),
                                multiclass=True)

            optimizer.zero_grad(set_to_none=True)
            grad_scaler.scale(loss).backward()
            grad_scaler.step(optimizer)
            grad_scaler.update()

            pbar.update(images.shape[0])
            global_step += 1
            epoch_loss += loss.item()
            experiment.log({
                'train loss': loss.item(),
                'step': global_step,
                'epoch': epoch
            })
            pbar.set_postfix(**{'loss (batch)': loss.item()})

            # Evaluation round
            division_step = (n_train // (10 * batch_size))
            if division_step > 0:
                if global_step % division_step == 0:
                    histograms = {}
                    for tag, value in net.named_parameters():
                        tag = tag.replace('/', '.')
                        if not torch.isinf(value).any():
                            histograms['Weights/' + tag] = wandb.Histogram(value.data.cpu())
                        if not torch.isinf(value.grad).any():
                            histograms['Gradients/' + tag] = wandb.Histogram(value.grad.data.cpu())

                    val_score = evaluate(net, val_loader, device)
                    scheduler.step(val_score)

                    logging.info('Validation Dice score: {}'.format(val_score))
                    experiment.log({
                        'learning rate': optimizer.param_groups[0]['lr'],
                        'validation Dice': val_score,
                        'images': wandb.Image(images[0].cpu()),
                        'masks': {
```

```
                            'true': wandb.Image(true_masks[0].float().cpu()),
                            'pred': wandb.Image(masks_pred.argmax(dim=1)[0].float().cpu()),
                        },
                        'step': global_step,
                        'epoch': epoch,
                        **histograms
                    })

        if save_checkpoint:
            Path(dir_checkpoint).mkdir(parents=True, exist_ok=True)
                                        torch.save(net.state_dict(),        str(dir_checkpoint        /
'checkpoint_epoch{}.pth'.format(epoch)))
            logging.info(f'Checkpoint {epoch} saved!')

def get_args():
    parser = argparse.ArgumentParser(description='Train the UNet on images and target masks')
     parser.add_argument('--epochs', '-e', metavar='E', type=int, default=5, help='Number of
epochs')
    parser.add_argument('--batch-size', '-b', dest='batch_size', metavar='B', type=int, default=1,
help='Batch size')
    parser.add_argument('--learning-rate', '-l', metavar='LR', type=float, default=1e-5,
                        help='Learning rate', dest='lr')
    parser.add_argument('--load', '-f', type=str, default=False, help='Load model from a .pth file')
    parser.add_argument('--scale', '-s', type=float, default=0.5, help='Downscaling factor of the
images')
    parser.add_argument('--validation', '-v', dest='val', type=float, default=10.0,
                        help='Percent of the data that is used as validation (0-100)')
    parser.add_argument('--amp', action='store_true', default=False, help='Use mixed precision')
      parser.add_argument('--bilinear', action='store_true', default=False, help='Use bilinear
upsampling')
    parser.add_argument('--classes', '-c', type=int, default=2, help='Number of classes')

    return parser.parse_args()

if __name__ == '__main__':
    args = get_args()

    logging.basicConfig(level=logging.INFO, format='%(levelname)s: %(message)s')
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    logging.info(f'Using device {device}')

    # Change here to adapt to your data
    # n_channels=3 for RGB images
    # n_classes is the number of probabilities you want to get per pixel
```

```
    net = UNet(n_channels=3, n_classes=args.classes, bilinear=args.bilinear)

    logging.info(f'Network:\n'
                 f'\t{net.n_channels} input channels\n'
                 f'\t{net.n_classes} output channels (classes)\n'
                 f'\t{"Bilinear" if net.bilinear else "Transposed conv"} upscaling')

    if args.load:
        net.load_state_dict(torch.load(args.load, map_location=device))
        logging.info(f'Model loaded from {args.load}')

    net.to(device=device)
    try:
        train_net(net=net,
                  epochs=args.epochs,
                  batch_size=args.batch_size,
                  learning_rate=args.lr,
                  device=device,
                  img_scale=args.scale,
                  val_percent=args.val / 100,
                  amp=args.amp)
    except KeyboardInterrupt:
        torch.save(net.state_dict(), 'INTERRUPTED.pth')
        logging.info('Saved interrupt')
        raise

import argparse
import logging
import os

import numpy as np
import torch
import torch.nn.functional as F
from PIL import Image
from torchvision import transforms

from utils.data_loading import BasicDataset
from unet import UNet
from utils.utils import plot_img_and_mask

def predict_img(net,
                full_img,
                device,
                scale_factor=1,
```

```
                out_threshold=0.5):
    net.eval()
    img = torch.from_numpy(BasicDataset.preprocess(full_img, scale_factor, is_mask=False))
    img = img.unsqueeze(0)
    img = img.to(device=device, dtype=torch.float32)

    with torch.no_grad():
        output = net(img)

        if net.n_classes > 1:
            probs = F.softmax(output, dim=1)[0]
        else:
            probs = torch.sigmoid(output)[0]

        tf = transforms.Compose([
            transforms.ToPILImage(),
            transforms.Resize((full_img.size[1], full_img.size[0])),
            transforms.ToTensor()
        ])

        full_mask = tf(probs.cpu()).squeeze()

    if net.n_classes == 1:
        return (full_mask > out_threshold).numpy()
    else:
        return F.one_hot(full_mask.argmax(dim=0), net.n_classes).permute(2, 0, 1).numpy()

def get_args():
    parser = argparse.ArgumentParser(description='Predict masks from input images')
    parser.add_argument('--model', '-m', default='MODEL.pth', metavar='FILE',
                        help='Specify the file in which the model is stored')
     parser.add_argument('--input', '-i', metavar='INPUT', nargs='+', help='Filenames of input images', required=True)
    parser.add_argument('--output', '-o', metavar='OUTPUT', nargs='+', help='Filenames of output images')
    parser.add_argument('--viz', '-v', action='store_true',
                        help='Visualize the images as they are processed')
    parser.add_argument('--no-save', '-n', action='store_true', help='Do not save the output masks')
    parser.add_argument('--mask-threshold', '-t', type=float, default=0.5,
                        help='Minimum probability value to consider a mask pixel white')
    parser.add_argument('--scale', '-s', type=float, default=0.5,
                        help='Scale factor for the input images')
      parser.add_argument('--bilinear', action='store_true', default=False, help='Use bilinear upsampling')
```

```
    return parser.parse_args()

def get_output_filenames(args):
    def _generate_name(fn):
        return f'{os.path.splitext(fn)[0]}_OUT.png'

    return args.output or list(map(_generate_name, args.input))

def mask_to_image(mask: np.ndarray):
    if mask.ndim == 2:
        return Image.fromarray((mask * 255).astype(np.uint8))
    elif mask.ndim == 3:
                        return Image.fromarray((np.argmax(mask, axis=0) * 255 /
mask.shape[0]).astype(np.uint8))

if __name__ == '__main__':
    args = get_args()
    in_files = args.input
    out_files = get_output_filenames(args)

    net = UNet(n_channels=3, n_classes=2, bilinear=args.bilinear)

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    logging.info(f'Loading model {args.model}')
    logging.info(f'Using device {device}')

    net.to(device=device)
    net.load_state_dict(torch.load(args.model, map_location=device))

    logging.info('Model loaded!')

    for i, filename in enumerate(in_files):
        logging.info(f'\nPredicting image {filename} ...')
        img = Image.open(filename)

        mask = predict_img(net=net,
                           full_img=img,
                           scale_factor=args.scale,
                           out_threshold=args.mask_threshold,
                           device=device)

        if not args.no_save:
            out_filename = out_files[i]
```

```
            result = mask_to_image(mask)
            result.save(out_filename)
            logging.info(f'Mask saved to {out_filename}')

        if args.viz:
            logging.info(f'Visualizing results for image {filename}, close to continue...')
            plot_img_and_mask(img, mask)
```

```
import logging
from collections import OrderedDict

import torch
import torch.nn as nn
from torch.nn.parallel import DataParallel, DistributedDataParallel
import models.networks as networks
import models.lr_scheduler as lr_scheduler
from .base_model import BaseModel
from models.modules.loss import ReconstructionLoss, Gradient_Loss, SSIM_Loss

logger = logging.getLogger('base')

class InvDN_Model(BaseModel):
    def __init__(self, opt):
        super(InvDN_Model, self).__init__(opt)

        if opt['dist']:
            self.rank = torch.distributed.get_rank()
        else:
            self.rank = -1    # non dist training
        train_opt = opt['train']
        test_opt = opt['test']
        self.train_opt = train_opt
        self.test_opt = test_opt

        self.netG = networks.define_G(opt).to(self.device)
        if opt['dist']:
            self.netG = DistributedDataParallel(self.netG,
device_ids=[torch.cuda.current_device()])
        else:
            self.netG = DataParallel(self.netG)
        # print network
```

```
        self.print_network()
        self.load()

        if self.is_train:
            self.netG.train()

            # loss
            self.Reconstruction_forw =
ReconstructionLoss(losstype=self.train_opt['pixel_criterion_forw'])
            self.Reconstruction_back =
ReconstructionLoss(losstype=self.train_opt['pixel_criterion_back'])
            self.Rec_Forw_grad = Gradient_Loss()
            self.Rec_back_grad = Gradient_Loss()
            self.Rec_forw_SSIM = SSIM_Loss()
            self.Rec_back_SSIM = SSIM_Loss()

            # optimizers
            wd_G = train_opt['weight_decay_G'] if train_opt['weight_decay_G'] else 0
            optim_params = []
            for k, v in self.netG.named_parameters():
                if v.requires_grad:
                    optim_params.append(v)
                else:
                    if self.rank <= 0:
                        logger.warning('Params [{:s}] will not optimize.'.format(k))
            self.optimizer_G = torch.optim.Adam(optim_params, lr=train_opt['lr_G'],
                                                weight_decay=wd_G,
                                                betas=(train_opt['beta1'],
train_opt['beta2']))
            self.optimizers.append(self.optimizer_G)

            # schedulers
            if train_opt['lr_scheme'] == 'MultiStepLR':
                for optimizer in self.optimizers:
                    self.schedulers.append(
                        lr_scheduler.MultiStepLR_Restart(optimizer, train_opt['lr_steps'],

restarts=train_opt['restarts'],

weights=train_opt['restart_weights'],

gamma=train_opt['lr_gamma'],

clear_state=train_opt['clear_state']))
```

```
            elif train_opt['lr_scheme'] == 'CosineAnnealingLR_Restart':
                for optimizer in self.optimizers:
                    self.schedulers.append(
                        lr_scheduler.CosineAnnealingLR_Restart(
                            optimizer, train_opt['T_period'], eta_min=train_opt['eta_min'],
                            restarts=train_opt['restarts'],
weights=train_opt['restart_weights']))
            else:
                raise NotImplementedError('MultiStepLR learning rate scheme is enough.')

            self.log_dict = OrderedDict()

    def feed_data(self, data):
        self.ref_L = data['LQ'].to(self.device)  # LQ
        self.real_H = data['GT'].to(self.device)  # GT
        self.noisy_H = data['Noisy'].to(self.device)  # Noisy

    def feed_test_data(self, data):
        self.noisy_H = data.to(self.device)  # Noisy

    def gaussian_batch(self, dims):
        return torch.randn(tuple(dims)).to(self.device)

    def loss_forward(self, out, y):
        l_forw_fit = self.train_opt['lambda_fit_forw'] * self.Reconstruction_forw(out, y)
        # l_forw_grad = 0.1* self.train_opt['lambda_fit_forw'] * self.Rec_Forw_grad(out, y)
        # l_forw_SSIM = self.train_opt['lambda_fit_forw'] * self.Rec_forw_SSIM(out,
y).mean()

        return l_forw_fit # + l_forw_grad + l_forw_SSIM

    def loss_backward(self, x, y):
        x_samples = self.netG(x=y, rev=True)
        x_samples_image = x_samples[:, :3, :, :]
        l_back_rec = self.train_opt['lambda_rec_back'] * self.Reconstruction_back(x,
x_samples_image)
        l_grad_back_rec = 0.1*self.train_opt['lambda_rec_back'] * self.Rec_back_grad(x,
x_samples_image)
        l_back_SSIM = self.train_opt['lambda_rec_back'] * self.Rec_back_SSIM(x,
x_samples_image).mean()
        return l_back_rec + l_grad_back_rec + l_back_SSIM


    def optimize_parameters(self, step):
```

```
            self.optimizer_G.zero_grad()

            # forward
            self.output = self.netG(x=self.noisy_H)

            LR_ref = self.ref_L.detach()

            l_forw_ce = 0
            l_forw_fit = self.loss_forward(self.output[:, :3, :, :], LR_ref)

            # backward
            gaussian_scale = self.train_opt['gaussian_scale'] if self.train_opt['gaussian_scale'] != None else 1
            y_ = torch.cat((self.output[:, :3, :, :], gaussian_scale * self.gaussian_batch(self.output[:, 3:, :, :].shape)), dim=1)

            l_back_rec = self.loss_backward(self.real_H, y_)

            # total loss
            loss = l_forw_fit + l_back_rec + l_forw_ce
            loss.backward()

            # gradient clipping
            if self.train_opt['gradient_clipping']:
                nn.utils.clip_grad_norm_(self.netG.parameters(), self.train_opt['gradient_clipping'])

            self.optimizer_G.step()

            # set log
            self.log_dict['l_forw_fit'] = l_forw_fit.item()
            self.log_dict['l_forw_ce'] = l_forw_ce
            self.log_dict['l_back_rec'] = l_back_rec.item()

        def test(self, self_ensemble=False):
            self.input = self.noisy_H

            gaussian_scale = 1
            if self.test_opt and self.test_opt['gaussian_scale'] != None:
                gaussian_scale = self.test_opt['gaussian_scale']

            self.netG.eval()
            with torch.no_grad():
                if self_ensemble:
```

```
                forward_function = self.netG.forward
                self.fake_H = self.forward_x8(self.input, forward_function, gaussian_scale)
            else:
                output = self.netG(x=self.input)
                self.forw_L = output[:, :3, :, :]
                y_forw = torch.cat((output[:, :3, :, :], gaussian_scale * self.gaussian_batch(output[:, 3:, :, :].shape)), dim=1)
                self.fake_H = self.netG(x=y_forw, rev=True)[:, :3, :, :]

        self.netG.train()

    def MC_test(self, sample_num=16, self_ensemble=False):
        self.input = self.noisy_H

        gaussian_scale = 1
        if self.test_opt and self.test_opt['gaussian_scale'] != None:
            gaussian_scale = self.test_opt['gaussian_scale']

        self.netG.eval()
        with torch.no_grad():
            if self_ensemble:
                forward_function = self.netG.forward
                self.fake_H = self.Multi_forward_x8(self.input, forward_function, gaussian_scale, sample_num)
            else:
                output = self.netG(x=self.input)
                self.forw_L = output[:, :3, :, :]
                fake_Hs = []
                for i in range(sample_num):
                    y_forw = torch.cat((output[:, :3, :, :], gaussian_scale * self.gaussian_batch(output[:, 3:, :, :].shape)), dim=1)
                    fake_Hs.append(self.netG(x=y_forw, rev=True)[:, :3, :, :])
                fake_H = torch.cat(fake_Hs, dim=0)
                self.fake_H = fake_H.mean(dim=0, keepdim=True)

        self.netG.train()

    def get_current_log(self):
        return self.log_dict

    def get_current_visuals(self):
        out_dict = OrderedDict()
        out_dict['LR_ref'] = self.ref_L.detach()[0].float().cpu()
        out_dict['Denoised'] = self.fake_H.detach()[0].float().cpu()
```

```
        out_dict['LR'] = self.forw_L.detach()[0].float().cpu()
        out_dict['GT'] = self.real_H.detach()[0].float().cpu()
        out_dict['Noisy'] = self.noisy_H.detach()[0].float().cpu()
        return out_dict

    def print_network(self):
        s, n = self.get_network_description(self.netG)
        if isinstance(self.netG, nn.DataParallel) or isinstance(self.netG,
DistributedDataParallel):
            net_struc_str = '{} - {}'.format(self.netG.__class__.__name__,
                                             self.netG.module.__class__.__name__)
        else:
            net_struc_str = '{}'.format(self.netG.__class__.__name__)
        if self.rank <= 0:
            logger.info('Network G structure: {}, with parameters: {:,d}'.format(net_struc_str,
n))
            logger.info(s)

    def load(self):
        load_path_G = self.opt['path']['pretrain_model_G']
        if load_path_G is not None:
            logger.info('Loading model for G [{:s}] ...'.format(load_path_G))
            self.load_network(load_path_G, self.netG, self.opt['path']['strict_load'])

    def save(self, iter_label):
        self.save_network(self.netG, 'G', iter_label)

    def forward_x8(self, x, forward_function, gaussian_scale):
        def _transform(v, op):
            v2np = v.data.cpu().numpy()
            if op == 'v':
                tfnp = v2np[:, :, :, ::-1].copy()
            elif op == 'h':
                tfnp = v2np[:, :, ::-1, :].copy()
            elif op == 't':
                tfnp = v2np.transpose((0, 1, 3, 2)).copy()

            ret = torch.Tensor(tfnp).to(self.device)
            return ret

        noise_list = [x]
        for tf in 'v', 'h', 't':
            noise_list.extend([_transform(t, tf) for t in noise_list])
```

```
        lr_list = [forward_function(aug) for aug in noise_list]
        back_list = []
        for data in lr_list:
            y_forw = torch.cat((data[:, :3, :, :], gaussian_scale * self.gaussian_batch(data[:,
3:, :, :].shape)), dim=1)
            back_list.append(y_forw)
        sr_list = [forward_function(data, rev=True) for data in back_list]

        for i in range(len(sr_list)):
            if i > 3:
                sr_list[i] = _transform(sr_list[i], 't')
            if i % 4 > 1:
                sr_list[i] = _transform(sr_list[i], 'h')
            if (i % 4) % 2 == 1:
                sr_list[i] = _transform(sr_list[i], 'v')

        output_cat = torch.cat(sr_list, dim=0)
        output = output_cat.mean(dim=0, keepdim=True)

        return output

    def Multi_forward_x8(self, x, forward_function, gaussian_scale, sample_num=16):
        def _transform(v, op):
            v2np = v.data.cpu().numpy()
            if op == 'v':
                tfnp = v2np[:, :, :, ::-1].copy()
            elif op == 'h':
                tfnp = v2np[:, :, ::-1, :].copy()
            elif op == 't':
                tfnp = v2np.transpose((0, 1, 3, 2)).copy()

            ret = torch.Tensor(tfnp).to(self.device)
            return ret

        noise_list = [x]
        for tf in 'v', 'h', 't':
            noise_list.extend([_transform(t, tf) for t in noise_list])

        lr_list = [forward_function(aug) for aug in noise_list]
        sr_list = []
        for data in lr_list:
            fake_Hs = []
            for i in range(sample_num):
                y_forw = torch.cat((data[:, :3, :, :], gaussian_scale * self.gaussian_batch(data[:,
```

```
3:, :, :].shape)), dim=1)
                    fake_Hs.append(self.netG(x=y_forw, rev=True)[:, :3, :, :])
                fake_H = torch.cat(fake_Hs, dim=0)
                fake_H = fake_H.mean(dim=0, keepdim=True)
                sr_list.append(fake_H)

            for i in range(len(sr_list)):
                if i > 3:
                    sr_list[i] = _transform(sr_list[i], 't')
                if i % 4 > 1:
                    sr_list[i] = _transform(sr_list[i], 'h')
                if (i % 4) % 2 == 1:
                    sr_list[i] = _transform(sr_list[i], 'v')

            output_cat = torch.cat(sr_list, dim=0)
            output = output_cat.mean(dim=0, keepdim=True)

            return output
```