Deblending using convolutional neural networks

Zhan Niu and Daniel O. Trad

ABSTRACT

Machine learning has been a booming subject in computer science and its applications have been made in various subjects including geophysics. Convolutional Neural Networks (CNNs) have great potential for solving image processing problems like denoising and interpolation. Deblending, considered as an underdetermined denoising problem, falls into this category. In this report, we use CNN to replace the deblending operator and its performance is analyzed. We use a 4-layer U-Net to perform deblending on synthetically blended shots from a wedge velocity model with point scatterers. We test out different hyper-parameters and the trained model could successfully remove the noise and preserve diffractions from the scatterers with some tolerance. The generality of the model is evaluated by testing the model on an easier 2-layer velocity model. The model can successfully identify and recover most part of the primaries but fails to deal with some interferences and leaves them muted.

INTRODUCTION

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

The deblending process essentially separates overlapped shots, which is an underdetermined problem to solve since it tries to produce several shots from each supershots. Therefore, additional constraints must be applied to get a unique solution. Pseudo-deblending is a technique that is commonly used. The method generally involves introducing known random delays to each shot. These time delays shift each shot differently so that the events become incoherent in other domains. We use this characteristic to separate each shot from the other ones simultaneously acquired. This converts deblending into a denoising process. The most challenging part is to solve the interference where different shots overlap. There are many choices for the denoising tool. For example, masks or mutes can be applied to F-K or hyperbolic Radon domains. Furthermore, inversion-based methods been developed as well, which involve creating a cost function with a regularization term. Results are highly depending on the definition of the regularization and the assumptions within can cause loss of signal (Stanton and Wilkinson, 2018).

On the other hand, machine learning methods can be applied instead of signal-processing / inversion types of deblending operators. The problem can be defined in two ways, a classification problem or a regression problem. The classification generates a mask that indicates the position of the desired shots but leaves the interferences unsolved, while the regression problem tries to produce individual shots but requires more parameters to be determined. For example, Baardman et al. (2019) used convolutional neural network (CNN, LeCun et al. 2015) for both problems. However, we think that CNNs may not be the best choice to capture the relationship between inputs and outputs due to the lack of skipping

connections. Richardson and Feller (2019) chose a U-Net model with ResNet34 encoder pre-trained on ImageNet and trained with random velocity models that are harder than the reality.

In this paper, we will look at an easier case and discuss the suitability of U-Net on solving deblending problems.

THEORY

Model definition

The neural network architecture to solve the problem is the U-Net (Ronneberger et al., 2015). The U-Net was designed based on the CNNs and bridge connections were added so that it performs fast and well especially for solving segmentation problems. Figure 1 shows a typical structure of U-Nets.

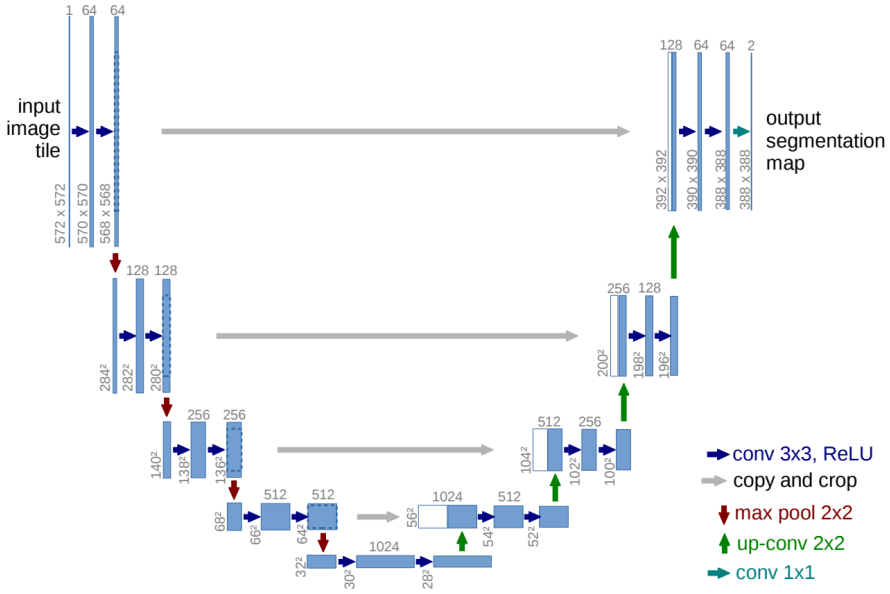

FIG. 1. Diagram of U-Net model modified from Ronneberger et al. (2015). The gray arrows refer to the bridge connections that directly pass the features from down-going layers to up-going layers.

The U-Net contains 3 parts. The down-going/encoding part, the up-going/decoding part and the bridge connections. The down-going part refers to the left half of the figure. The U-Net has 4 tiers in total. At the first tier, the inputs went through two 3 by 3 convolutional layers with a predefined initial number of filters (64 filters in this case). The initial number of filters defines how many features are extracted from the inputs. The cascading of convolutional layers essentially enlarges the extent of kernel coverage. After the convolution, the output was max-pooled by a 2 by 2 grid and used for the inputs fed to the next tier. At each time going down a tier, the number of filters used for convolution doubled while the number of image dimension is halved due to the max-pooling. On the other side for the up-going part, the inputs will go through the opposite process. In this part, the number of filters gets halved and the image dimensions get doubled as the tier goes up. At the top tier, the image dimensions are restored to the original size and the number of channels is reduced to 1 by an additional outputting convolution layer. Each green arrow in the figure stands for a

2 by 2 up-convolution, which up-samples the image by two and then convolves with a 2 by 2 kernel. For now, this structure is also called an encoder-decoder convolutional network. This type of structure extracts high order features from the inputs and reconstructs the output by decoding. The third part is the bridge connection, which are indicated by the gray horizontal arrows. Each grey arrow refers to the process where part or the whole outputs was forwarded as additional features to the same tier in the up-going part. These additional channels were concatenated with the outputs from deeper tier after up-convolution. The directivity of the connections reduces the number of backpropagation terms and hence mitigates the risk of vanishing gradients.

A segmentation prediction produces a mask of a given picture indicating an area of interests. For example, this has applications on a brain MRI for finding the damaged area, or in our case in seismic, for targeting events in a noisy shot record. Essentially, it predicts the probability of a given pixel to be true. The probability on each pixel then can be converted to a true or false by applying a judging threshold. The reason why U-Net is more suitable to solve segmentation problems over traditional variations of CNNs is that U-Net has bridge connections that directly link the features with the same tier as shown in the gray arrows in Figure 1. Since the inputs and outputs of the segmentation usually correlate and share spatial similarities, the connections will greatly reduce the efforts to learn this relation by skipping unnecessary transforms, which helps to reduce the chance of vanishing gradients.

Loss function

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

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

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

Back-propagation

The gradients with respect to each model parameters can be calculated by a back-propagation algorithm, which is a recursive estimation of error propagation by applying the chain rule (Goodfellow et al., 2016). Equation 2 shows a form of gradient calculated by back-propagations. Suppose that $L = \mathcal{L}(\mathbf{a})$ and $\mathbf{a} = f(\mathbf{h})$, then the gradient of L with respect to the hidden parameters h is

$$\frac{\partial L}{\partial h_i} = \sum_j \frac{\partial L}{\partial a_j} \cdot \frac{\partial a_j}{\partial h_i} \tag{2}$$

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

One can use the gradients directly to update the model or use gradient-based optimization methods to make updates in a more controlled manner. The first is the most intuitive way but may result in a zigzag path to the minimum. The second method trying to reduce the zigzag pattern and is faster in an ideal case. One popular method to perform minimization is ADAM (Kingma and Ba, 2014), which reduces the transverse oscillations by cumulatively summing all the previous gradients during the optimization. The pseudocode of ADAM update is shown in Algorithm 1. A more detailed explanation of ADAM and its characteristics can be found in Niu et al. (2018).

Algorithm 1 The ADAM optimization. i stands for the current iterations. $\mathbf{g}$ is the gradient calculated and $\mathbf{h}$ contains the parameters to be updated. $\mathbf{v}$ and $\mathbf{s}$ are the two vectors storing the cumulative sum of historical $\mathbf{g}$ and $\mathbf{g}^2$. α and β are two hyper parameters that control the portion of updates that is related to $\mathbf{v}$ and $\mathbf{s}$.

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

Training workflow

The training template can be summarized in the pseudocode described as Algorithm 2. During the training process, we want to find a model that has better generalization, i.e. a model that performs better on unseen data. This goal can best be achieved by finding the model with the least error on the validation set. A common way of creating a validation set is to take it from a portion of the training set. Validation is important since it is the only method we have to evaluate whether a model is over-fitted or under-fitted. Machine learning problems can sometimes be an underdetermined problem with more number of

Algorithm 2 Training workflow.

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

parameters to determine compared to the number of data points, especially when the sample size is small. As the training proceeds, the model will fit the data better but may become less general and invalid for other data sets. In other words, the model will learn to perform well only on the data provided, but do a poor job on unseen data. Therefore, we need a labelled dataset that is in the same distribution as the training set but also stays independent of the model gradient updates for checking the degrees of over-fit. If the loss from validation set is similar to the loss from the training set, then we can say the model performs equally well on seen and unseen data. Hence we can have confidence that the model will do well on the test set. On a typical machine learning problem, the validation loss curve will decrease with the training loss at the early stage but the training loss will decrease faster since the gradient is optimized for the train set only. Then the validation loss will start to increase where the model starts to over-fit the training set. Although the training loss will be smaller after this point, we should prefer the model where the validation loss is minimal as the validation set best represents the samples in the test set.

Based on the characteristics of PyTorch, we should make zero the gradients at each iteration, otherwise, the gradients will cumulate and cause long-wavelength oscillations in the loss curve. At each minibatch, it first loads a batch of inputs to the device. After each full cycle of epochs, the mini-batches will be shuffled again for stochastic gradient updates. In theory, the validation set should be evaluated after each model update, which should be in the inner loop. In practice, validation loss is usually calculated once at the end of each epoch.

SYNTHETIC DATA EXAMPLES

Data preparation

All data used in this paper were generated synthetically with a finite difference method. The blended data was created by injecting shots simultaneously with random delays and measuring the total wavefield in the receiver locations using the velocity model shown in Figure 2. This model contains 3 layers and a wedge on the left, with several point scatterers

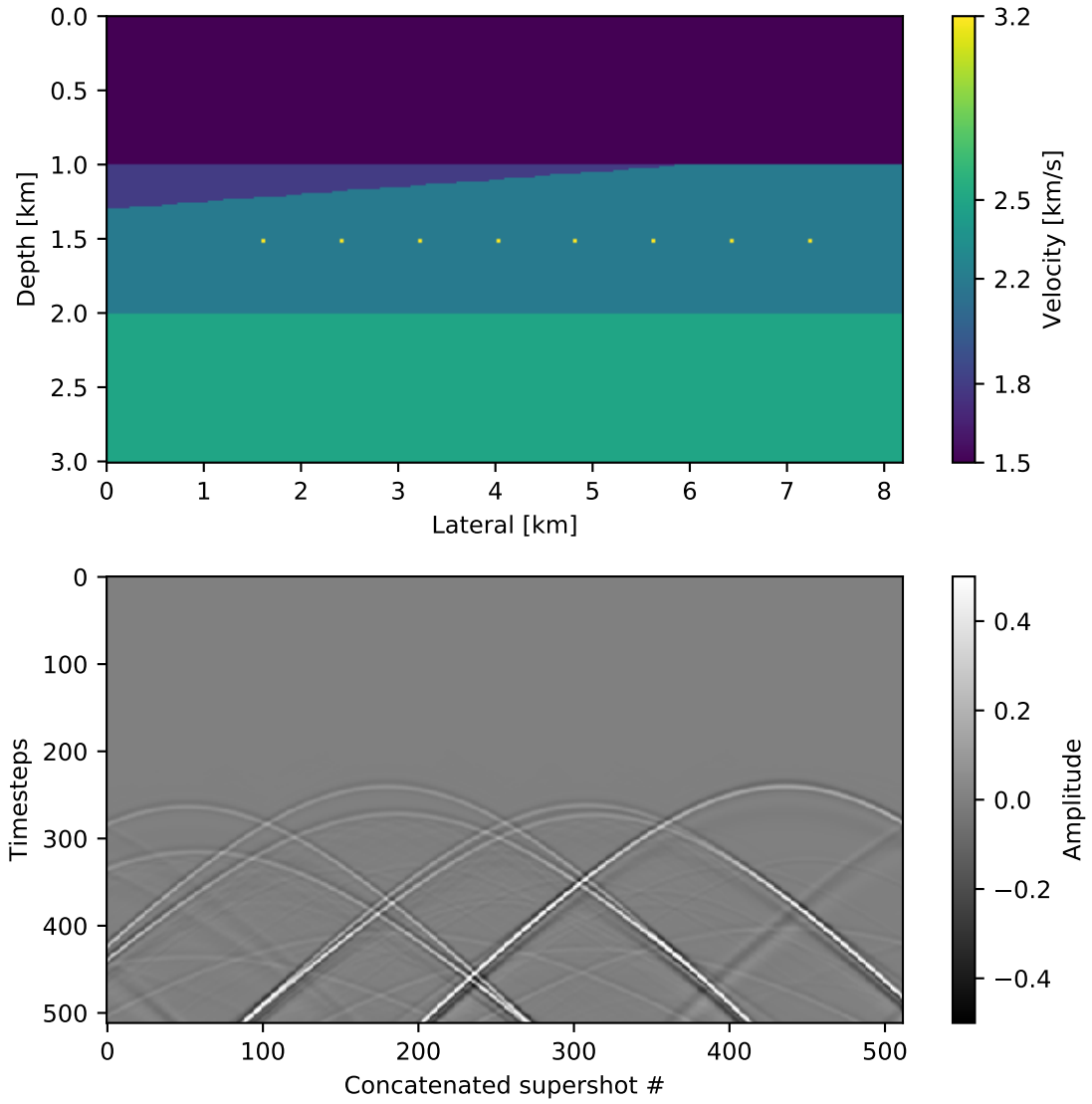

FIG. 2. The inputs fed to the U-Net model. The plots show the corresponding input (above) and label (bellow) pair at the 120th receiver, with 512 receiver slices in total.

under the dipping layer. These scatterers are intended to test whether the deblending algorithm can honour data diffractions. The dipping layer of the wedge moves the apexes of reflections in the shot domain. In this model, 64 supershots were recorded with 4 shots blended in each and with 512 receivers. The data were resampled by increasing the time-step size and the number of time samples was reduced from 3600 to 512 to reduce the computation cost. Both sources and receivers are evenly distributed at the near-surface. Also, we created for the training a regular data set without blending or time delays, which we call here "true data".

The blended data are first pseudo-deblended as follows: the supershots are repeated as many times as the number of blended shots per supershots, and the copies are concatenated in the shot axis with the time delays removed one shot at a time. After this pseudo-deblending, only those shots whose time delay were completely removed become coherent in the receiver

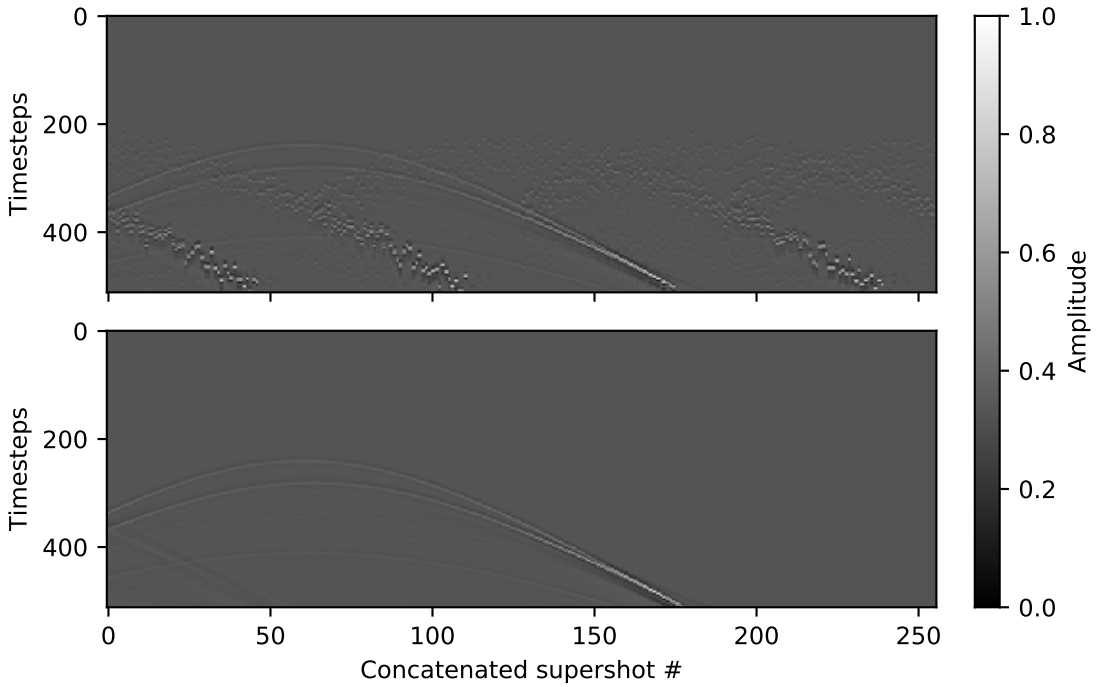

FIG. 3. The inputs fed to the U-Net model. The plots show the corresponding input (above) and label (bellow) pair at the 120th receiver, with 512 receiver slices in total.

domain (Figure 3). Since duplications of supershots were concatenated together, the blended data now has dimensions of $N_g \times N_t \times N_{shot}$, which refers to the number of receivers, timesteps, and shots, respectively. The number of shots here is the product of supershots and the number of blended shots. We treat each receiver gather as a picture which we feed to the network for training. The time and source axis becomes the height and width of the picture. We also have to define an extra dimension in the 2nd place to represent the number of channels. Since the input is in grayscale, the number is 1. Therefore, the input tensors has the format of $N_{\text{sample}} \times 1 \times H \times W$ which is $512 \times 1 \times 512 \times 256$ where N_{sample} is the number of receiver gathers. The "true data", that is receiver gathers without blending, have the same dimensions as well (Figure 3). Both inputs and labels are normalized to be ranging from 0 to 1 for better generalization, as indicated by the scale bar.

The dataset was then separated into training and validation sets. In this project, the randomly chosen 20% of the entire dataset becomes the validation set and the rest becomes the training set which will be used for calculating the gradient.

Training

We used PyTorch (Paszke et al., 2017) for the machine learning framework and adapted the U-Net implementation described in Buda et al. (2019), which was designed for brain MRI. The U-Net has 4 tiers in depth with two 3 by 3 convolutional layers in each block with zeros padding of 3 at each boundary, which guarantees the inputs and outputs having the same dimension. The original model was designed to take inputs with 3 channels as RGB

images and has 32 filters in the initial layer. In this paper, however, since the receiver gathers only have one channel, the default 32 filters may be more than needed. We discuss later in the report our choice for the number of initial filters. The U-Net uses ReLU as inter-layer activation functions and uses batch normalization layers. The output activation is sigmoid, which regularizes the outputs to a $(0, 1)$ range.

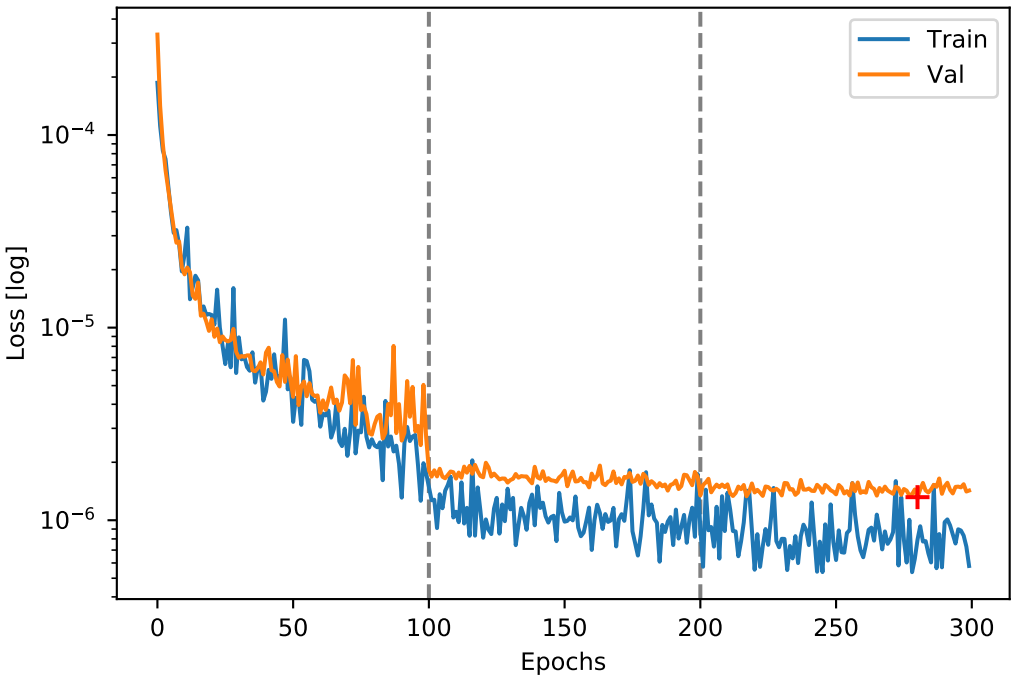

FIG. 4. The loss curve when initial filters is 16. The blue line refers to the training loss while the orange line is the validation loss (L_{val}). The red cross indicates the least L_{val}, which is 1.318×10^{-6} at epoch 280. The gray dashed lines separate regions with different learning rates.

After some testing and experimentation, we decided to train the model by following Algorithm 2, with an ADAM optimizer, a learning rate of 0.002, $\alpha = 0.9$ and $\beta = 0.999$. To mitigate large oscillations at later epochs, we decrease the step learning rate every 100 iterations. After each completion of 100 epochs, we reduce the learning rate to its 10%. This learning rate decay slows down the descent in later epochs but makes it tolerant to a bigger learning rate at the early stage (the default learning rate for the brain MRI problem was 0.0001). Figure 4 shows the loss curves. The validation loss reaches a plateau with small oscillations at 200^{th} to 300^{th} epochs. Models at those iterations can be considered to have the same confidence level. One could choose the model at the last epoch since it undergoes more training but we picked the model with the least L_{val} to avoid over-fitting (see the red cross in Figure 4).

As mentioned in the previous section, the initial number of filters defines how many features are extracted from the inputs. There is a trade-off between the complexity of the model and performance. The more features are extracted, the more information from the inputs are used for training but with more computational cost. Figure 5 shows three validation losses with different initial numbers of filters. The saved model is summarized

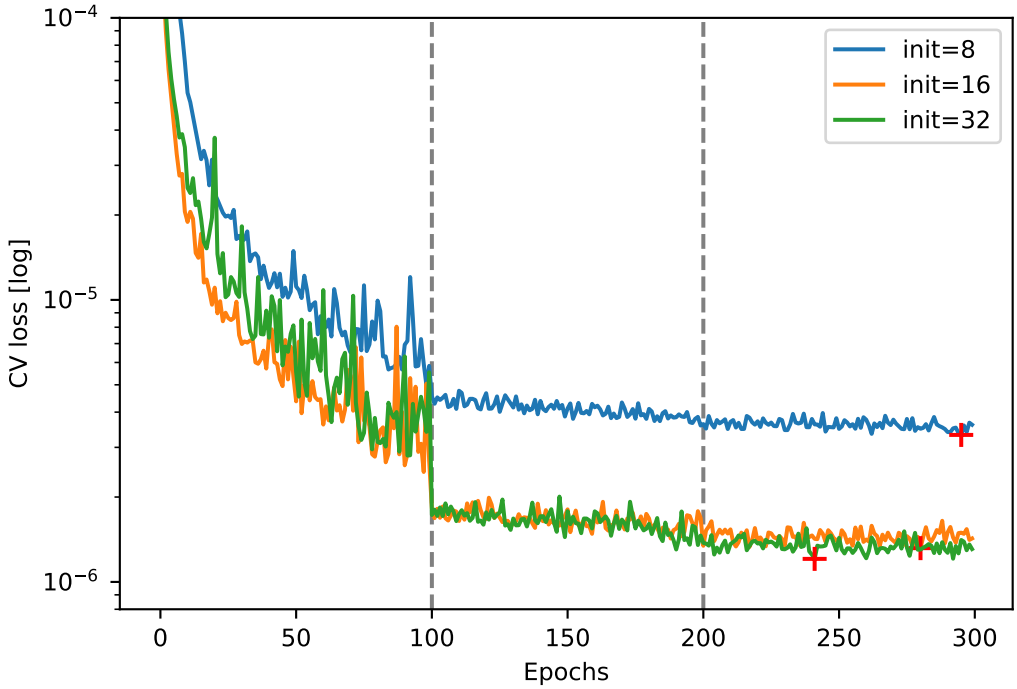

FIG. 5. The cross comparison of L_{val} with varying initial filters. The blue, orange and green lines refers to the cases with 8, 16 and 32 filters, respectively. Red crosses stand for the least L_{val} on each line. The results are summarized in Table 1.

in Table 1. Losses with 16 and 32 filters performed almost the same. The model with 16 filters descends faster than the model with 32 filters because of being simpler, but the later achieves lower validation loss and requires fewer iterations. The model with 8 initial filters shows poor results. Having too few parameters did not help with the model updates, which may be evidence that 8 filters are not enough to capture the important information in this problem. The model with 32 filters has better accuracy but also takes almost double time to calculate and memory to train. After balancing these trade-offs, we chose the model with 16 initial filters.

Table 1. Best epochs and best L_{val} for different initial filter setups.

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

Figure 6 shows an example for a prediction from the validation set. Most of the incoherent noise is removed. Furthermore, the diffractions from the point scatterers are mostly preserved although with some attenuation in their tail endings. The larger errors are concentrated in two regions. The first region is inside the major primary, where the reflections get complicated. Probably the identification of the reflections becomes difficult

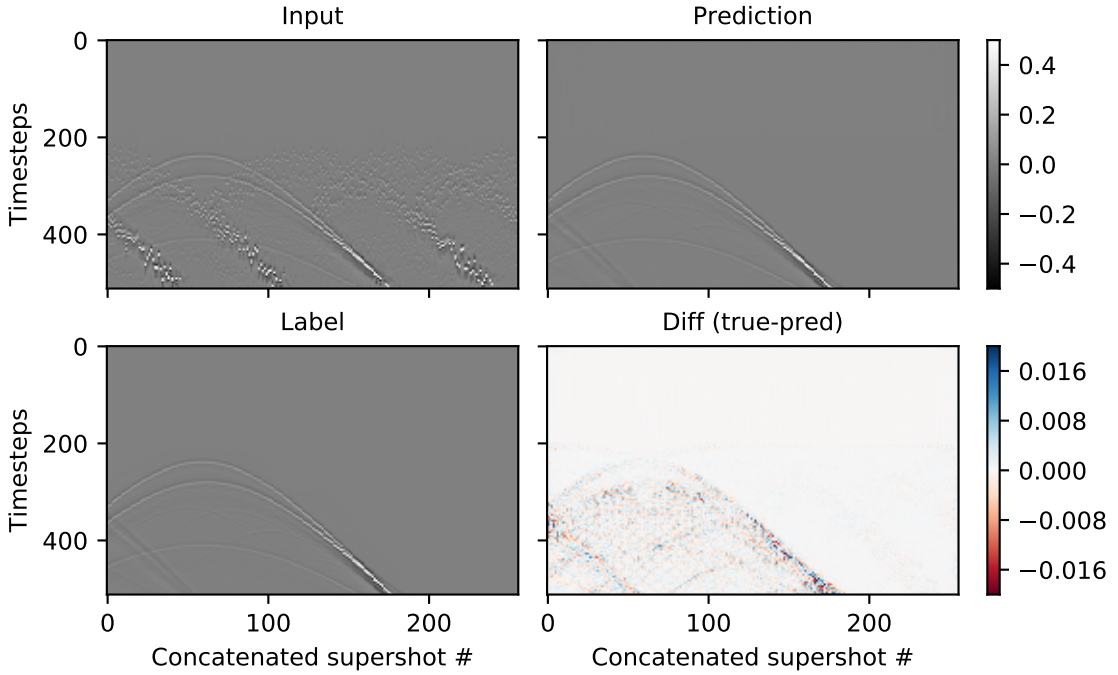

FIG. 6. The prediction and label for a sample in the validation set. All three grayscale images have the same scale. The picture in the bottom right shots the difference of the prediction and the label, with a smaller color scale.

for the algorithm and the interference complicates this. It is important to keep in mind that the network does not know what reflections or diffractions are, but just see them as patterns. Furthermore, we can also see some meshed patterns at the bottom of the plot. These patterns could be multiples of the point scatterers' reflections or boundary artifacts. Likely the model behaves poorly for them because of their week amplitude and complexity. The second region is around the tails of the reflection. Something to mention is that the input data have some missing samples due to the removal of time delays, but the "true" data do not. To get the right prediction, the model tries not only to remove the incoherent noise but also to interpolate the missing samples, which itself is a complex problem to solve. Therefore, the prediction contains relatively larger errors after training on these points.

Figure 7 shows 9 predictions on all samples from the train and validation set. Some shadows of the blended shots are still present. Probably this is because in shots from the acquisition edges the blended reflections are not like a typical hyperbola and are different from the majority of the receiver gathers. Therefore, the model fails to resolve the signal and noise in this case. One solution could be to use gradient boosting, which trains several models iteratively to adapt to different situations. However, additional models will introduce more model parameters and extra attention must be paid to avoid over-fitting. Another point to notice is that the error seems larger in the shot domain than in the receiver domain, in which the model was trained on. This may be solved by feeding the data in multiple receiver gathers or even the entire volume instead of single gathers. However, this cannot be done with one velocity model and indeed need exponentially larger computation resources for the training.

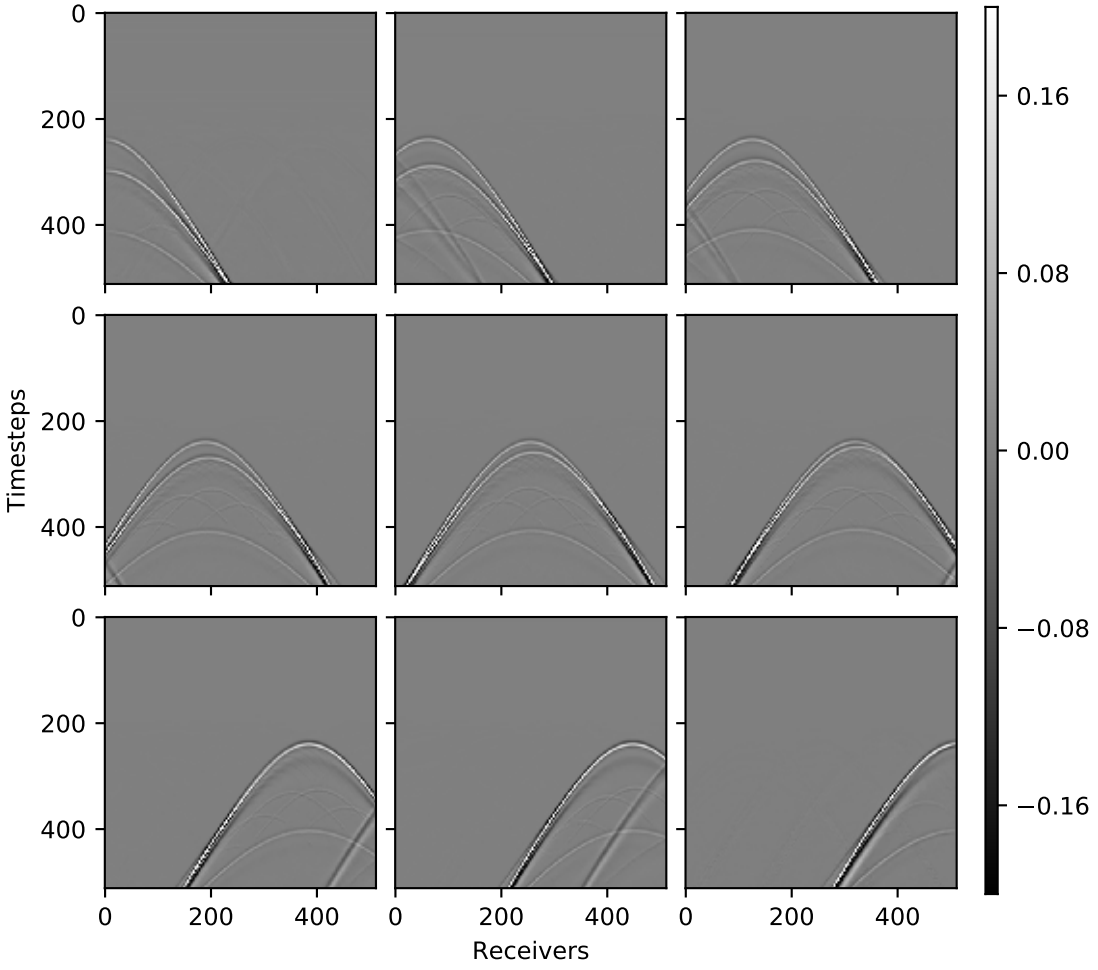

FIG. 7. The prediction on the whole dataset containing both the training and validation set (transposed to the shot domain). Note the preservations of the diffractions.

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

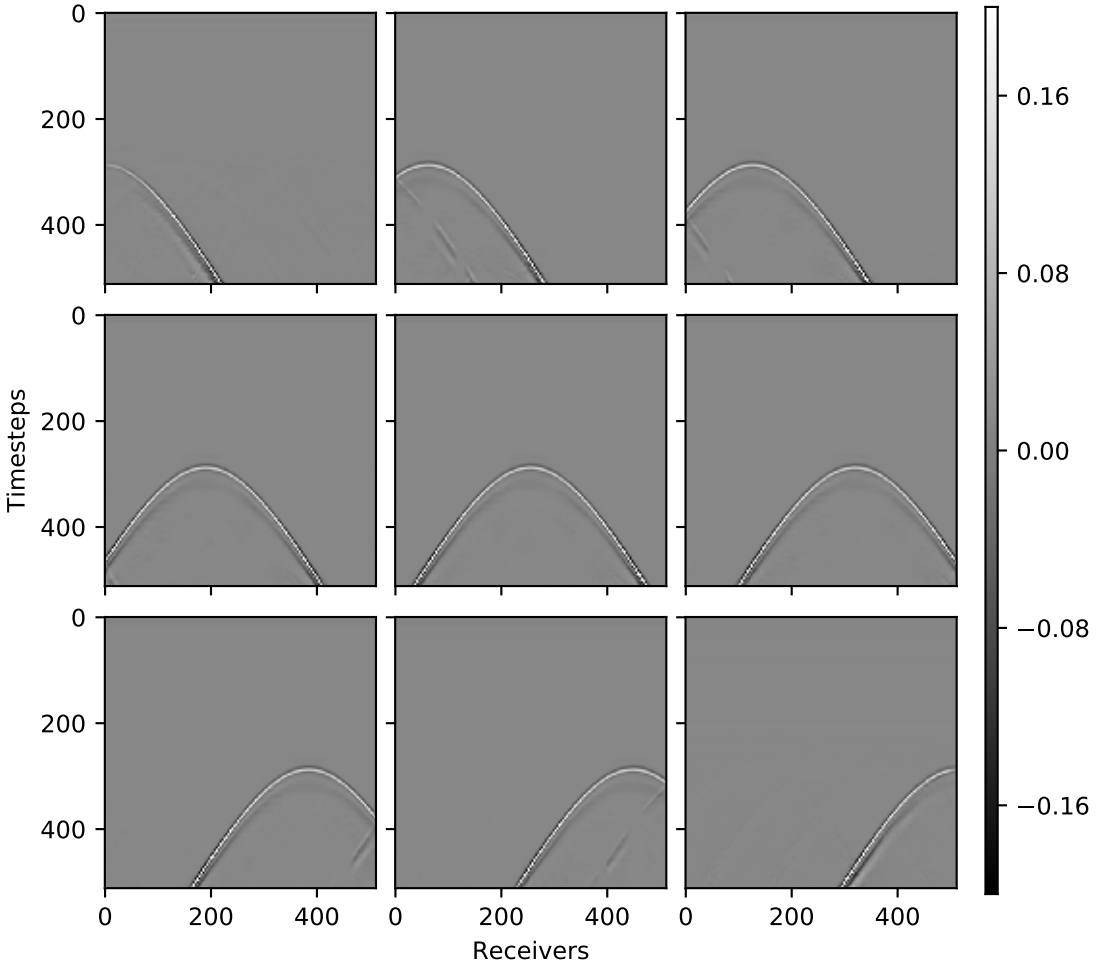

FIG. 8. Predictions for blended data from a two-layer model.

CONCLUSION

In this report, we trained a U-Net model to perform deblending, that is the separation of coherent and incoherent signal coming from blended shots. We tried several optimization and network parameters and found the best combination. In the case where the training and test data come from the same velocity model, the network performs well by preserving small diffractions and correctly identifying primaries. It performs a bit worse for the shots at the edge of the model because of the lack of training pictures representative of this case. For the case where the test data comes from a different model than the training data, the network performs okay but not as well as the first case. In this case, the test model was simpler than the training model, so the test is not conclusive and more work is required to fully understand how to generalize the network to new problems. To address these issues we plan to investigate in generalizing the model by gradient boosting, and provide several

models for training.

ACKNOWLEDGEMENTS

We thank the sponsors of CREWES for continued support. This work was funded by CREWES industrial sponsors and NSERC (Natural Science and Engineering Research Council of Canada) through the grant CRDPJ 461179–13. Special thanks to Jian Sun at Penn State University and Marcelo Guarido for valuable discussions.

REFERENCES

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

Beasley, C. J., Chambers, R. E., and Jiang, Z., 1998, A new look at simultaneous sources, *in* SEG Technical Program Expanded Abstracts 1998, Society of Exploration Geophysicists, 133–135.

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

Goodfellow, I., Bengio, Y., and Courville, A., 2016, Deep Learning: MIT Press, `http://www.deeplearningbook.org`.

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

LeCun, Y., Bengio, Y., and Hinton, G., 2015, Deep learning: nature, **521**, No. 7553, 436.

Niu, Z., Sun, J., and Trad, D., 2018, Inversion with the born approximation in a deep learning framework: CREWES Research Report.

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

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

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

Stanton, A., and Wilkinson, K., 2018, Robust deblending of simultaneous source seismic data: arXiv preprint arXiv:1812.06040.