# Improving Seismic Wave Simulation and Inversion Using Deep Learning

Lei Huang(✉), Edward Clee, and Nishath Ranasinghe

Department of Computer Science, Prairie View A&M University, Prairie View, TX 77446, USA
{lhuang,niranasinghe}@pvamu.edu, T_Clee@acm.org

**Abstract.** Accurate simulation of wave motion for the modeling and inversion of seismic wave propagation is a classical high-performance computing (HPC) application using the finite difference, the finite element methods and spectral element methods to solve the wave equations numerically. The paper presents a new method to improve the performance of the seismic wave simulation and inversion by integrating the deep learning software platform and deep learning models with the HPC application. The paper has three contributions: 1) Instead of using traditional HPC software, the authors implement the numerical solutions for the wave equation employing recently developed tensor processing capabilities widely used in the deep learning software platform of PyTorch. By using PyTorch, the classical HPC application is reformulated as a deep learning recurrent neural network (RNN) framework; 2) The authors customize the automatic differentiation of PyTorch to integrate the adjoint state method for an efficient gradient calculation; 3) The authors build a deep learning model to reduce the physical model dimensions to improve the accuracy and performance of seismic inversion. The authors use the automatic differentiation functionality and a variety of optimizers provided by PyTorch to enhance the performance of the classical HPC application. Additionally, methods developed in the paper can be extended into other physics-based scientific computing applications such as computational fluid dynamics, medical imaging, nondestructive testing, as well as the propagation of electromagnetic waves in the earth.

**Keywords:** Machine learning · Inverse problem · Wave propagation

## 1 Introduction

Physical simulation and inversion are classical scientific computing applications to discover the physical phenomenon and reveal the underlying properties. The simulation solves the partial differential equations (PDE) that governs the physical phenomenon using numerical approximation methods, while the inversion applies the gradient-based optimizations to find the underlying properties by minimizing the observed data and the simulated results. The entire process takes


J. Nichols et al. (Eds.): SMC 2020, CCIS 1315, pp. 3–19, 2020.
https://doi.org/10.1007/978-3-030-63393-6_1

significant computing resources to achieve the satisfied accuracy. However, the inverse problem is naturally challenging since it is ill-posed and nonlinear for most cases.

Recent advances in high-performance tensor processing hardware and software are providing new opportunities for accelerated linear algebra calculations as used in machine learning, especially for deep learning neural networks, that contributes significantly to the success of data science. Such calculations are also at the heart of many simulations of physical systems such as wave propagation. The use of tensor processing in neural networks, with its need for back-propagation through multi-layered networks, has led to capabilities for automatic differentiation [1] for gradient calculations in deep learning software.

*Motivations:* The motivations of the work have twofold. The first one is to understand the new deep learning software package such as PyTorch and TensorFlow, and their capacity of solving a scientific computational problem. Especially, we are interested in how to model the traditional partial differential equations (PDEs) used in the scientific computational problem with a deep learning model. The other is to study how to integrate the machine learning models that are data-driven into the scientific computational model that are physics-driven. The differentiable programming has the potential to smoothly integrate them together with a global optimization. The authors believe the study will lead to more interesting research findings in the topic of Scientific Machine Learning (SciML) and to find an efficient way to combine the power of these two different methods to facilitate scientific discovery.

In this paper, we study how to use the tensor-based machine learning software to formulate the physical simulation and to compute the gradients for optimizations to solve the inverse problem. We use the seismic wave propagation simulation and the Full Wave Inversion (FWI) as the physical case study. We have adapted the techniques of others in this area of wave propagation [2,3] to demonstrate how direct finite difference integration can be implemented via a deep learning software platform, allowing the gradients calculated by automatic differentiation to be used for the FWI of seismic reflection survey data as an augmentation to the well-known PySIT [4] seismic research platform.

We summarize the paper's contributions in the following:

i) We formulate the PDE solver in the seismic forward model using the Recurrent Neural Network (RNN) implemented with the deep learning software package PyTorch, which allows us to take advantages of the tensor processing software and its accelerator implementation.
ii) We apply the automatic differentiation implemented in PyTorch to solve the seismic inverse problem to uncover the earth's interior physical properties.
iii) We improve the automatic differentiation efficiency by creating a hybrid back propagation method with the adjoint-state method to calculate the gradients.
iv) We implement an AutoEncoder network to reduce the dimensions of the inverted parameters to argument the convergence process and get more accurate results for the ill-posed problem.

## 2 Wave Equations and RNN

### 2.1 Wave Equations

The wave motion is governed by physical rules that can be expressed in the following partial differential equation (PDE) (1) and the boundary conditions (2) and (3). We use the 1D scalar wave equation for simplicity purpose in this paper:

$$\frac{1}{c^2(x)}\frac{\partial^2 u(x,t)}{\partial t^2} - \frac{\partial^2 u(x,t)}{\partial x^2} = f(x,t) \tag{1}$$

$$\frac{1}{c(0)}\frac{\partial u(0,t)}{\partial t} - \frac{\partial u(0,t)}{\partial x} = 0 \tag{2}$$

$$\frac{1}{c(1)}\frac{\partial u(1,t)}{\partial t} - \frac{\partial u(1,t)}{\partial x} = 0 \tag{3}$$

where $c(x)$ is the spatial velocity distribution, $u(x,t)$ is the wave field distribution in space and time, and $f(x,t)$ is the energy source distribution in space and time.

The Eq. (1) can be solved numerically using a finite difference approximation:

$$\begin{aligned} f(x,t) = &-\frac{u(x-\Delta x,t) - 2u(x,t), + u(x+\Delta x,t)}{\Delta x^2} \\ &+\frac{1}{c^2}\frac{u(x,t-\Delta t) - 2u(x,t) + u(x,t+\Delta t)}{\Delta t^2}. \end{aligned} \tag{4}$$

After factoring, the Eq. (4) can be expressed as

$$\begin{aligned} u(x,t+\Delta t) = &f(x,t)c^2\Delta t^2 + (2u(x,t) - u(x,t-\Delta t)) \\ &+c^2\frac{\Delta t^2}{\Delta x^2}(u(x-\Delta x,t) - 2u(x,t) + u(x+\Delta x,t)) \end{aligned} \tag{5}$$

which shows that the next wave field in time $u(x,t+\Delta t)$ can be calculated based on the current and prior wave fields, as well as spatial neighbors in the current wave field. The wave motion simulation follows the time sequence to produce the next state based on the prior ones, which is similar to the Recurrent Neural Network (RNN) in deep learning to model a time sequence function.

### 2.2 Recurrent Neural Network

Recurrent Neural Network (RNN) is used to model the pattern in a sequence of data, mostly in time sequence. In recent years, RNN and its variants have been applied successfully to problems such as speech recognition, machine translation, and text-to-speech rendering. It has an internal cell that repeatedly processes an input, carries a hidden state, and produces an output at each step. The RNN cell

can be designed to be simple or complex to model a problem with a forgettable memory mechanism (Long Short-Term Memory (LSTM) [5]) or/and a gating mechanism (Gated Recurrent Unit (GRU) [6]).

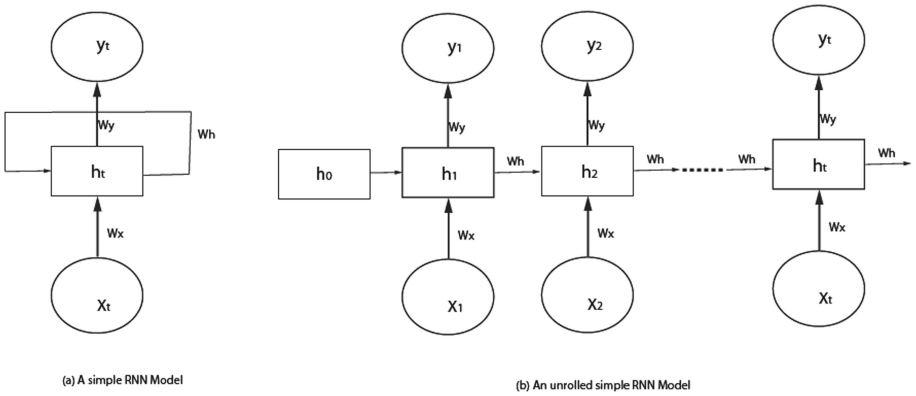



**Fig. 1.** A Simple RNN Model (a) with feedback loop, and (b) with loop unfolded

Figure 1(a) shows a typical RNN structure that repeatedly takes an input, updates its hidden state, and produces an output at every step. The RNN model can be unfolded as shown in Fig. 1(b) that learns the recurrence relationship from a sequence of data. The hidden state $h_i$ remembers the prior state of the process and is updated at each step. The hidden state enables RNN to learn the temporal relationships among the inputs since most of the time sequence data do contain temporal patterns. LSTM allows RNN to forget long-term relationships built up in the hidden state and emphasizes the short-term relationships, which can be useful for many cases.

A simple RNN can be expressed in the Eq. (6):

$$\begin{aligned} h_t &= \sigma_h(W_h x_t + W_h h_{t-1} + b_h) \\ y_h &= \sigma_y(W_y h_t + b_y) \end{aligned} \tag{6}$$

where $x_t$ is the input, $h_t$ is the hidden state, $W$ is the weights, $b$ is the bias, and $\sigma$ is the activation function.

Looking back to the Eq. (5), there are two hidden states $u(x,t)$ and $u(x,t-\Delta t)$ if we can restructure the finite difference method using an RNN. There is also a spatial stencil relationship of neighboring velocity distribution. We define a new function $F$ with input of $f(x,t)$, two hidden states $u(x,t)$ and $u(x,t-1)$, and the constant velocity distribution $c$:

$$
\begin{aligned}
&F(f(x,t),u(x,t),u(x,t-1),c)\\
&\quad = f(x,t)c^2\Delta t^2 + (2u(x,t) - u(x,t-1))\\
&\quad + c^2\frac{\Delta t^2}{\Delta x^2}(u(x-1,t) - 2u(x,t) + u(x+1,t)).
\end{aligned}
\tag{7}
$$

Then, the Eq. (5) can be restructured as an RNN format:

$$
\begin{aligned}
h_{t+1} &= \sigma(F(f(t),h(t),h(t-1),c))\\
y_{t+1} &= P(h_{t+1})
\end{aligned}
\tag{8}
$$

where P is the projection function to get the sample of a trace from a receiver. The Eq. (8) is then a non-learnable, deterministic physical solution represented as the deep learning RNN model. Figure 2 shows the RNN model we designed that solves the wave equation with four inputs $f(x,t)$, $h(t)$, $h(t-1)$, and $c$, the velocity distribution which is constant in the equation. The output $y_t$ is the trace sample of a receiver at each time step.

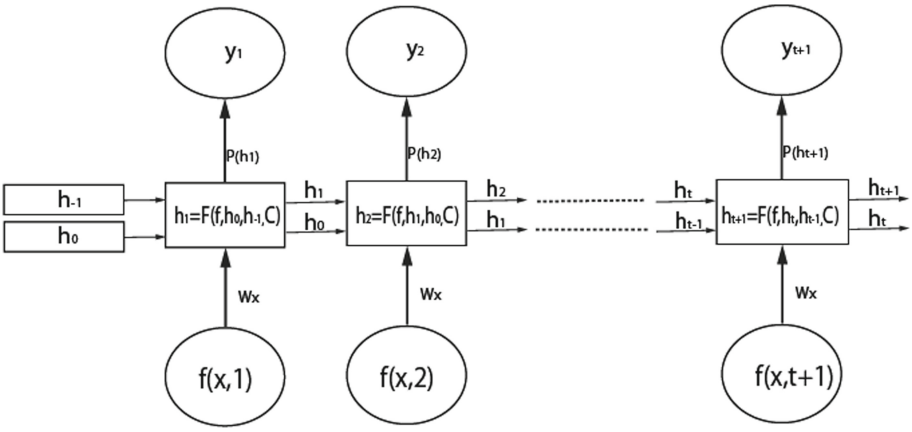



**Fig. 2.** A RNN model for wave equation

### 2.3 PyTorch RNN Implementation

The wave equation RNN model we designed in Fig. 2 enables us to utilize the deep learning software platform to solve the wave equations. The benefits of using a deep learning model to represent an HPC application include: (1) we will be able to leverage the HPC implementation of the deep learning model exploiting the advantages of GPUs/multicores and vectorization for better performance; (2) have an automatic gradients calculation using the built-in automatic differentiation package in deep learning; (3) utilize the variety of built-in optimizers to apply the gradients to find the global/local optimums; (4) use the data- and

model- parallelism framework implemented in deep learning package to run the application on a HPC cluster.

The following shows a code snippet of our RNN-similar implementation of wave equation using PyTorch. There are two classes derived from torch.nn.Module for RNN cell and RNN driver respectively. We called them Wave_PGNNcell and Wave_Propagator in our code. The Wave_PGNNcell implemented a cell function in RNN that computes the wavefield at a time step. The Wave_Propagator iterates over all time steps and takes the Ricker source waveform sample as the input at each time step. The hidden state (self.H) contains the next and current wavefields, which are fed into the cell for the next iteration. The trace is collected by projecting the current wavefield based on the receiver location. The program returns the simulated wavefield and sampled trace at the end.

```
class Wave_PGNNcell(torch.nn.Module):
  def forward(self, H, src ):
      uC,uP = [ H[0], H[1] ]
      ...
      return [uN,uC]

class Wave_Propagator(torch.nn.Module):
  self.cell = Wave_PGNNcell(C, config)

  def forward(self):
    us = []      # list of output wavefields
    traces = []
    rcv = self.rcvrs
    for it in range(self.nt):
      self.H = self.cell.forward(self.H, self.ws[it])
      us.append( self.H[0].detach().numpy() )
      # Extract wavefield sample at each receiver
      samps = rcv.sample( self.H[0].clone() )
      traces.append( samps )
    trc = torch.stack(traces,dim=1)
    return us, trc
```

### 2.4 Seismic Wave Simulation

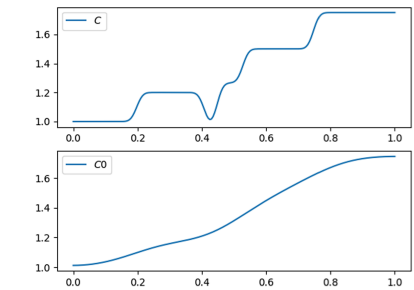
For seismic wave simulation, we use our RNN model to simulate the acoustic wave propagation for the scalar wave equation. We create a "true" synthetic model and an initial model, which can be a smoothed version of the true model or some other separately chosen function. We use the Ricker wavelet as a waveform for one or more energy sources (shots) and create an array of receivers for collecting traces. We assume the constant density in these models.

As we stated earlier, one benefit of using deep learning software is to take advantage of its multiple CPUs and GPUs implementation. We only need to specify which devices the code will operate on and define tensors to these devices. All remaining device-specific implementation and optimizations are done internally by PyTorch. We do not need to use CUDA or OpenACC to port the code to these devices.

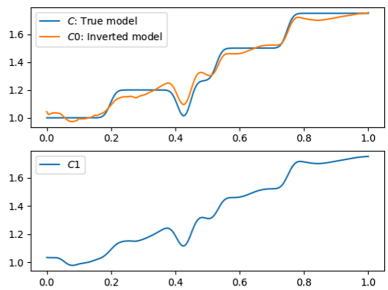
Another benefit is to use the data-parallelism implemented in PyTorch. We can parallelize the code by the number of the sources/shots to run the code on multiple GPUs and distributed clusters.

In our implementation, we use PyTorch[1] 1.5 to build the RNN model. PyTorch is an open source machine learning framework developed by Facebook by merging Torch and Caffe2, which supports a variety of hardware platforms including multiple CPUs, GPUs, distributed systems, and mobile devices. Besides the machine learning and deep learning functions, one unique feature of PyTorch is that it contains a just-in-time compiler to optimize the code if it complies with TorchScript, which is a subset of Python. It has a built-in automatic differentiation package for calculating derivatives, as well as a distributed training module to train a model on a HPC cluster. PyTorch has both Python and C++ frontends.

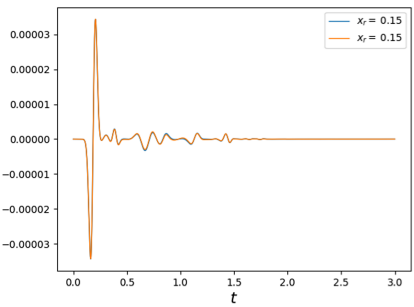
Figure 3 shows a 1D seismic Velocity Inversion case applying our physics-ruled RNN implementation. The Fig. 3(a) shows a true synthetic velocity model and an initial model; Fig. 3(b) shows the inverted model comparing with the true model (up) and a slightly smoothed final inverted model (down); Fig. 3(c) shows the comparison of the true traces and the inverted traces; and Fig. 3(d) shows the wavefield on how the seismic wave propagates with respect to space and time.



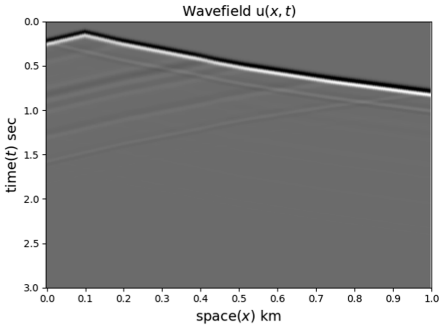

(a) a True Model and an Initial Model




(b) The Inverted Model Comparison




(c) Trace Comparison




(d) Wavefield

**Fig. 3.** Applying RNN for 1D seismic velocity inversion

[1] https://pytorch.org/.

The 1D inversion experiment finds a close-to the true model solution after 100 iterations. We use Adam optimizer [7] with L2 regularization. We are currently working on 2D cases by revising PySIT package. We continue performing more testing cases to evaluate the performance with both data and model parallelism provided by PyTorch on a CPU cluster and multiple GPUs.

## 3 Differentiable Programming

### 3.1 Automatic Differentiation and Adjoint-State Method

The automatic differentiation (AD) is also called algorithmic differentiation that calculates the derivatives of any arbitrary differentiable program. Unlike using the numerical differentiation of the adjoint state method that is an approximation to calculate the derivatives, the automatic differentiation returns the exact answer of the derivatives, though subject to the intrinsic rounding error. Machine learning software such as TensorFlow and Pytorch all have the built-in implementation of AD as the core functionality of backpropagation to optimize machine learning models. Accurate gradients are critical to the gradient-based optimizations used in both scientific computing and machine learning.

In order to calculate the derivatives of any differentiable programs, AD needs to store all operations on the execution path along with the intermediate results. It then propagates derivatives backward from the final output for every single operation connected with the chain rule. For large scale application, AD faces the challenge of meeting the demands of fast-growing storage in proportion to the executed operations. Furthermore, the individual derivative function for each operation also slows down the computation with intrinsic sequential execution. More work needs to be done if AD can be directly applied to a real scientific application.

Computationally expensive scientific applications typically use the adjoint state method to calculate the gradient of a function with much better computation efficiency, although it is a numerical approximation. In FWI, the adjoint state method calculates the derivative of a forward function *J(m)* that depends on *u(m)*. The forward function $J$ can be defined using $h$, as following [8]:

$$J(m) = h(u(m), m) \tag{9}$$

where $m$ is the model parameter, which belongs to the model parameter space $\mathbf{M}$ and $u$ belongs to the state variable space, $\mathbf{U}$. The state variables, $u$ follow the state equations outlined with the mapping function, $F$, which is also known as the forward problem or forward equation [8]:

$$F(u(m), m) = 0. \tag{10}$$

The mapping function $F$ is mapping from $\mathbf{U} * \mathbf{M}$ to $\mathbf{U}$ and is satisfied by the state variable $u$. If the condition *F(u, m) = 0* is satisfied, the state variable $u$ becomes a physical realization. Then, the adjoint state equation can be given as following, where $\lambda$ is the adjoint state variable and $\tilde{u}$ is any element of $\mathbf{U}$ [8]:

$$[\frac{\delta F(u,m)}{\delta \tilde{u}}]^* \lambda = \frac{\delta h(u,m)}{\delta \tilde{u}}. \tag{11}$$

This adjoint-state gradient calculation involves computing the reverse-time propagated residual wavefield, combining with the saved forward-propagated wavefield snapshots at specified time intervals to provide adjustments to the medium properties (the gradient) at each spatial mesh point. In summary, the forward propagation computes data observations representing the response of the model, and the residual between the model response and actual observed data is backward propagated and combined with the forward model response to compute adjustments to the current model estimate.

Intervening in the calculation of the gradient in this manner allows for management of the required computational resources by saving the forward wavefields only as often as numerically required, explicitly managing data resources through staging to disk or check-pointing as needed, implementing shot-level parallelism, and other specially tailored techniques.

### 3.2 Extended Automatic Differentiation

A difficulty with the auto-differentiation (AD) procedure is that memory requirements for the back-propagation graph can become excessive, as noted by Richardson [2]. Applying chain-rule differentiation on elemental network nodes over thousands of RNN time steps for a large mesh of physical parameter values is a reasonably-sized task for 1D problems, but the graph quickly becomes intractable for 2D and 3D models. This issue renders impractical the use of pure AD for such model inversion problems.

In order to solve the problem, we extended the AD backward process using PyTorch AD workflow to integrate the adjoint-state method for the more efficient gradient calculation. In PyTorch, we can customize the AD workflow by providing a backward function to calculate the gradients of any function. We need to pass the required parameters of the forward function, the model parameters and loss function to allow the backward function to pick up these parameters for the adjoint-state calculation.

Control over this auto-differentiation process is available through use of a PyTorch extension to the Autograd feature pictured conceptually in Fig. 4, wherein the RNN layer of the network can be replaced by a forward propagation loop and corresponding adjoint back-propagation loop for an equivalent gradient calculation provided by the user. This alternative gradient calculation can take advantage of well-known techniques in seismic inversion processing, enabling existing performance enhancements to be applied using the extended PyTorch capability for specially designed back-propagation.

In the present case, the physical medium properties to be optimized are provided to the "forward" wave propagation problem implemented using the publicly available PySIT seismic inversion toolkit [4], creating a simulated seismic response. The corresponding "backward" propagation consists in using the residual wavefield represented by the difference between the simulated data and the

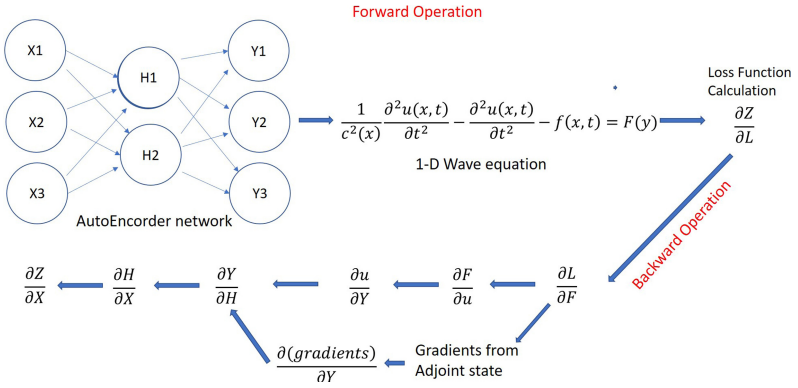



**Fig. 4.** Adjoint gradient: Automatic differentiation vs. Adjoint gradient calculation. Differentiation respect to model parameters are replaced by gradients from adjoint state in the backward automatic differentiation.

observed seismic trace data from the corresponding actual field data recording (or recordings from a "true" model in our synthetic studies), and implementing the "adjoint-state" solution to provide the required gradient of the model parameters. Other implementations of wave propagation solutions may also be used in this framework, such as spectral-element methods [9] for 2D, 3D and spherical 3D wave propagation.

The beneficial end result is that traditional adjoint-state solution methods are incorporated into the AD workflow, so that seismic inversion calculations can be integrated within the broader deep learning process with efficient calculation.

## 4 Seismic Inversion

### 4.1 Seismic Inversion

Seismic Inversion [10] is the method to reconstruct the earth subsurface image by inverting seismic data observed via the multiple distributed sensors on the surface. It is typically implemented using the adjoint state method [8] to calculate the gradients. As described in Sect. 2 and Sect. 3, by reconstructing the forward problem using deep learning software, the seismic inversion problem can be solved by the automatic differentiation package, a variety of optimizers provided by PyTorch, and a customized loss function. The automatic differentiation package in PyTorch implements the methodology of automatic differentiation by recording all the forward operations in sequence and performing backward derivative computation based on the chain rule.

Figure 5 shows the workflow of seismic inversion. The initial model $M0$ is a guess of the true model $M$ that needs to be inverted. In these early experiments using several shots of a synthetic seismic reflection survey over a small 2D Earth model, we used for convenience an initial model guess that is a smoothed version

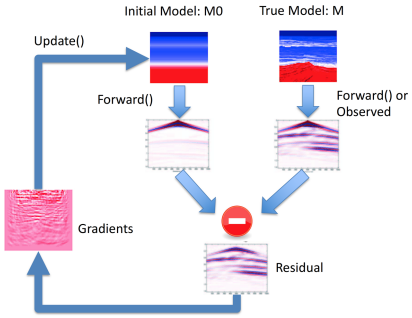



**Fig. 5.** The full waveform inversion workflow

of the true model. The seismic traces are either observed via distributed sensors on top of the earth surface in the real-world application or are simulated using the seismic wave forward function in this paper. The residual is obtained by comparing the synthetic data and observed data. The gradient $\frac{\partial u}{\partial M}$ is calculated based on the residual with respect to the initial model. The gradients are used by a gradient-based optimizer to update the initial model to get a step close to the real model. The entire process ends when the initial model and the true model are converged or exceeded the specified number of iterations.

### 4.2 AutoEncoder for Dimensionality Reduction

The seismic inversion process needs to uncover the physical properties at every point represented in the geological space, which quickly leads to a large number of model parameters to optimize in the traditional FWI process. The nature of the nonlinear and ill-posed inverse problem often falls into the local minimum traps. It is a sound solution to apply the dimensionality-reduction technique to reduce the optimization parameters to improve the optimization accuracy by engaging with machine learning models.

Since we have customized the automatic differentiation workflow by integrating the adjoint state method for the FWI gradients (described in Sect. 3), it is now feasible to integrate the machine learning models into the FWI workflow and keep the program differentiable. Since the AutoEncoder $A(x)$ is differentiable and the forward model $F(x)$ is differentiable, the composition of the $F(A(x))$ is differentiable. We choose the AutoEncoder as the dimensionality-reduction method and apply it before the forward model as shown in Fig. 6.

The AutoEncoder contains 743,938 parameters as shown in Fig. 7a and b. The AutoEncoder is an unsupervised learning model that compresses the information representation of the input data to a sparse latent variable with less dimensions at the middle of the encoded layer. It then reconstructs the data from the encoded latent variable to the original or enhanced data. The compression process is called encoder and the reconstruction is called decoder. The encoder learns how to compress the input data and describes it with the latent variable, while the decoder learns how to reconstruct the data from the latent variable.

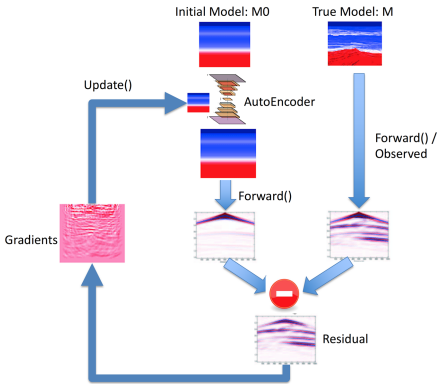
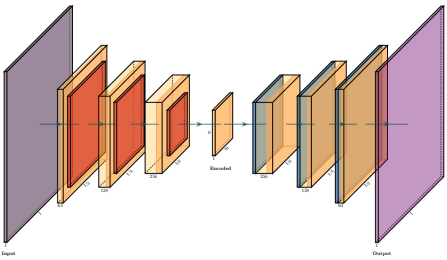



**Fig. 6.** The full waveform inversion workflow



(a) The AutoEncoder Network Structure

| Layer (type) | Output Shape | Param # |
|---|---|---|
| Conv2d-1 | [-1, 64, 30, 40] | 640 |
| ReLU-2 | [-1, 64, 30, 40] | 0 |
| Conv2d-3 | [-1, 128, 15, 20] | 73,856 |
| ReLU-4 | [-1, 128, 15, 20] | 0 |
| Conv2d-5 | [-1, 256, 8, 10] | 295,168 |
| Tanh-6 | [-1, 256, 8, 10] | 0 |
| Conv2d-7 | [-1, 1, 8, 10] | 2,305 |
| ConvTranspose2d-8 | [-1, 256, 15, 20] | 2,560 |
| ReLU-9 | [-1, 256, 15, 20] | 0 |
| ConvTranspose2d-10 | [-1, 128, 30, 40] | 295,040 |
| ReLU-11 | [-1, 128, 30, 40] | 0 |
| ConvTranspose2d-12 | [-1, 64, 60, 80] | 73,792 |
| ReLU-13 | [-1, 64, 60, 80] | 0 |
| ConvTranspose2d-14 | [-1, 1, 60, 80] | 577 |

Total params: 743,938
Trainable params: 743,938
Non-trainable params: 0

(b) The AutoEncoder Model Parameters

**Fig. 7.** Traditional seismic velocity inversion

We start the AutoEncoder training by generating a large number of random seismic velocity models. In this work, we are using some simple and flat velocity layers representing the velocities of different earth interiors including water and rocks. Specifically, these models contain one or more low velocity layers in the middle or bottom of these layers that is challenging for the low velocity inversion. All of these models have the fixed dimensions of $60 \times 80$. As indicated in Fig. 7a, the AutoEncoder has two components: a encoder and a decoder. The encoder compresses the input model with dimension of $60 \times 80$ to an encoded latent variable with dimension of $8 \times 10$, which is 1/60 of the original dimension. The latent variable is then decompressed by the decoder to restore to its original dimension.

The loss function we used to train the AutoEncoder is the mean-square-error (MSE) loss and the optimizer is Adam with learning rate of 0.001. The batch size used is 128. The loss values during the training process is shown in Fig. 8.

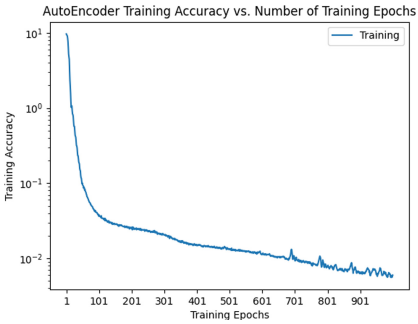



**Fig. 8.** The autoEncoder training loss

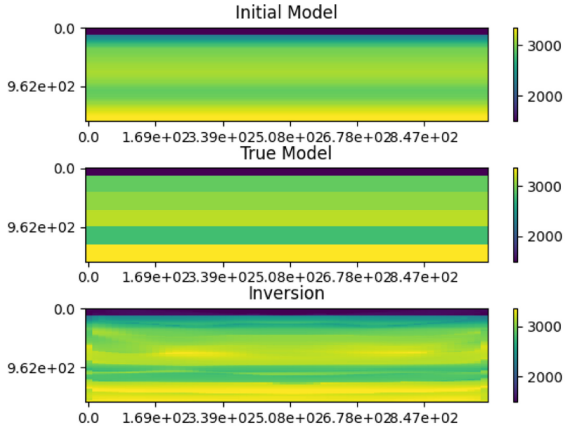
Figure 6 shows the AutoEncoder enhanced FWI process, where the AutoEncoder is inserted before the forward function simulation starts. Note that the encoder is only applied to the first iteration to get the encoded latent variable. For the rest of optimization iterations, the decoder is applied to decompress the encoded latent variable to get a new velocity model with the original dimension. During the gradient-based optimization process, the gradients are calculated with respected to the encoded latent variable, instead of the original model, which reduced the dimensionality of the optimization search space to 1/60. We use the MSE loss and Adam optimizer during the process.

### 4.3 Results

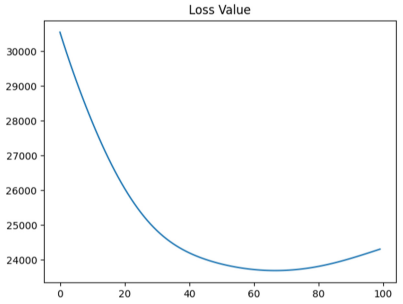
PyTorch has a list of optimizers including Adam [7], RMSprop [11], stochastic gradient descent (SGD), Adadelta [12], Adagrad [13], LBFGS, and their variants. The learning rate, scheduler and regularizations can be specified to fit different optimization problems. There are also multiple regression and classification loss functions implemented in PyTorch. All of these packages provide a rich environment to solve inverse problems.

In our implementation, we have demonstrated how to invoke the extended automatic gradient calculation for the velocity model. We choose the Adam optimizer and the MSE loss function to compare the misfit of the simulated traces and observed traces after each iteration of the forward model. The partial derivative (the gradient) of the loss function with respect to the initial model and the encoded latent variable is calculated by the automatic differentiation process, which is applied by the optimizer to minimize the misfit. These iterations gradually find an approximation of the true velocity distribution.

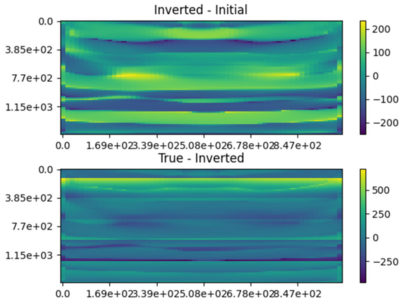
Figure 9 and Fig. 10 show the differences of the traditional FWI and the AutoEncoder enhanced FWI results. Fig. 9(a) shows the initial model, the true model, and the inverted model; the loss graph Fig. 9(b) shows the loss values (at different scales) after each optimization iteration, and Fig. 9(c) shows the difference between the inverted model and the initial model (top), as well as the difference between the inverted model and the true model. It appears that




(a) The Initial, True and Inverted Model Comparison




(b) Loss Function Value

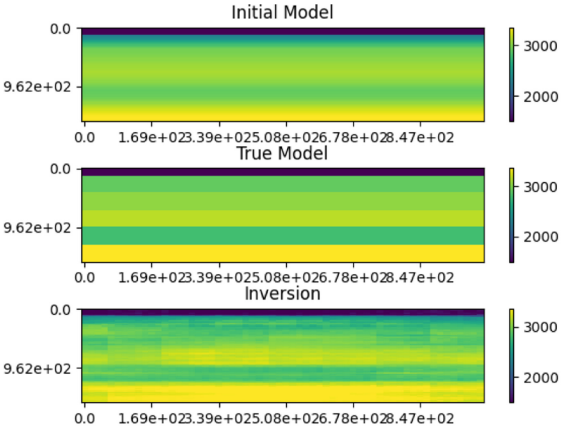



(c) Differences

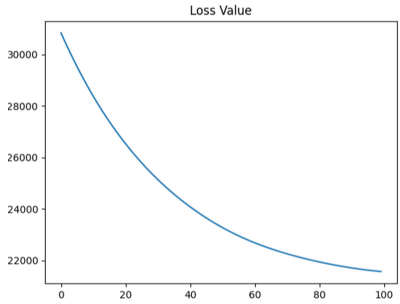
**Fig. 9.** Traditional seismic velocity inversion

the traditional FWI does not optimize well in the low velocity layer case after 40 iterations ended with a high loss value, which falls into a local trap. The AutoEncoder-enhanced FWI discovers the low velocity layer very well and continues to optimize the misfit for all 100 iterations. The difference graphs also confirm that the AutoEncoder case identifies all layers well showing less structured misfits. Noticeably, there are also less artifacts introduced in the AutoEncoder enhanced FWI compared with the traditional FWI.

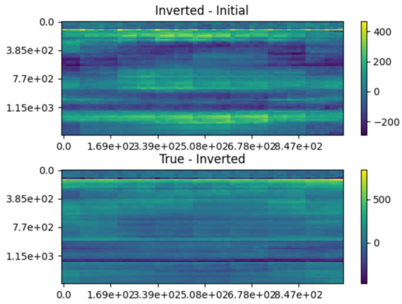
As described in Sect. 3, the automatic differentiation provided by the PyTorch software does not provide sufficient efficiency to solve the FWI 2D problem. The gradients calculated for the whole program takes too long and too much space to store them. We use the hybrid method describe in Sect. 3.2 to overcome the problem by incorporating the adjoint state method. As the result, the gradient calculation using the hybrid approach achieves both accuracy and efficiency,




(a) The Initial, True and Inverted Model Comparison




(b) Loss Function Value




(c) Differences

**Fig. 10.** The AutoEncoder enhanced seismic velocity inversion

which is feasible to be used for a large scale scientific computation problem integrating with machine learning models.

## 5 Discussion

There are a few of points that worth noting for the work. The first is that the automatic differentiation is key for differentiable programming, which can bridge the physics-based scientific computing with the machine learning (ML)/artificial intelligence (AI) technologies. ML/AI methods do not have physics principles built in that may create an infeasible solution given the fact that most of the scientific inverse problems may be ill-posed. In our prior work [14], the convergence of ML with a scientific application without differentiable programming may not find a generalized solution since optimizations of the two different methods are disconnected.

The second point we would like to make is that the automatic differentiation needs additional improvements to make it feasible to other applications. In our method, we integrate the adjoint-state method to make it feasible to solve a large case, however the solution is an approximation. If the automatic differentiation method can be more memory-efficient and parallelizable, it can be much more useful to compute the exact gradients for the large complex problems.

The last point is the deep learning model AutoEncoder requires a revisit to reduce the loss during decoding. Although it reduces the dimension by compressing the input data into a sparse latent variable, the reconstruction is not lossless. There are some errors introduced during the reconstruction process that may hinder the optimization process. There is a trade-off to take into the consideration when designing the convergence of ML/AI with scientific computing. The good news is that there are many options to integrate them waiting for us to explore.

## 6 Conclusion and Future Work

We have successfully demonstrated two case studies of restructuring the wave equation using finite difference method in a deep learning RNN model framework and an AutoEncoder enhanced FWI process. The benefits of the work include fully utilizing the high-performance tensor processing and optimization capabilities implemented in the deep learning package PyTorch, as well as the deep integration of machine learning models with the inverse problem. By integrating an HPC application with a deep learning framework with differential programming, we can explore a large number of combinations of machine learning models with physical numerical solutions to achieve better accuracy and efficiency.

**Acknowledgment.** This research work is supported by the US National Science Foundation (NSF) awards ##1649788, #1832034 and by the Office of the Assistant Secretary of Defense for Research and Engineering (OASD(R&E)) under agreement number FA8750-15-2-0119. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the US NSF, or the Office of the Assistant Secretary of Defense for Research and Engineering (OASD(R&E)) or the U.S. Government. The authors would also like to thank the XSEDE for providing the computing resources.

## References

1. Baydin, A.G., Pearlmutter, B.A., Radul, A.A., Siskind, J.M.: Automatic differentiation in machine learning: a survey. J. Mach. Learn. Res. **18**(1), 5595–5637 (2017)
2. Richardson, A.: Seismic full-waveform inversion using deep learning tools and techniques (2018). https://arxiv.org/pdf/1801.07232v2.pdf

3. Hughes, T.W., Williamson, I.A.D., Minkov, M., Fan, S.: Wave physics as an analog recurrent neural network (2019). https://arxiv.org/pdf/1904.12831v1.pdf
4. Hewett, R.J., Demanet, L., The PySIT Team: PySIT: Python seismic imaging toolbox (January 2020). https://doi.org/10.5281/zenodo.3603367
5. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997). https://doi.org/10.1162/neco.1997.9.8.1735
6. Chung, J., Gulcehre, C., Cho, K., Bengio, Y.: Empirical evaluation of gated recurrent neural networks on sequence modeling (2014)
7. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization (2014)
8. Plessix, R.-E.: A review of the adjoint-state method for computing the gradient of a functional with geophysical applications. Geophys. J. Int. **167**(2), 495–503 (2006). https://doi.org/10.1111/j.1365-246X.2006.02978.x
9. Tromp, J., Komatitsch, D., Liu, Q.: Spectral-element and adjoint methods in seismology. Commun. Comput. Phys. **3**(1), 1–32 (2008)
10. Schuster, G.: Seismic Inversion. Society of Exploration Geophysicists (2017). https://library.seg.org/doi/abs/10.1190/1.9781560803423
11. Ruder, S.: An overview of gradient descent optimization algorithms (2016)
12. Zeiler, M.D.: ADADELTA: an adaptive learning rate method (2012)
13. Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. J. Mach. Learn. Res. **12**, 2121–2159 (2011)
14. Huang, L., Polanco, M., Clee, T.E.: Initial experiments on improving seismic data inversion with deep learning. In: 2018 New York Scientific Data Summit (NYSDS), August 2018, pp. 1–3 (2018)