Deep Learning for SIFT Unit Source Inversions

Christopher Liu

06/09/21, Revised: 06/11/21

1 Introduction

The intention of this document is to provide instructions for future users and developers on how to modify the machine learning workflow. Furthermore, since deep learning oftentimes is heuristic and lacks hard-set rules, I would also like to leave behind my thoughts on future work and obstacles I faced while creating this workflow. Many of the machine learning concepts and terminology used in this document are briefly defined or not defined at all. Numerous resources on machine learning exist and some will be provided in the Additional Reading section.

Finally, the workflow, especially the neural network architecture itself, provided in this repository is by no means the best approach and should instead be considered as a basic framework for the problem at hand. I strongly believe that better results can be achieved with the current available dataset.

2 Training and Hyperparameter Tuning

While training a model is straightforward, tuning the hyperparameters can be overwhelming due to the vast amount of options and combinations of hyperparameters available. This section will give a brief overview of the model architecture, how to validate the model error, and what hyperparameters can be tuned.

2.1 Model Overview

The input to the model is assumed to be the following,

$$[i \times j \times k] = [\text{batch size} \times \text{n\_channels} \times \text{signal length}]$$

where batches size refers to the size of the batches in the training set (more on this later), n_channels is the number of input channels (e.g. $j = 3$ if we use DARTs 46404, 46407, and 46419), and signal length corresponds to the actual time series (e.g. $k = 45$ for an input time window of 45 minutes and a time series with an increment of 1 minute).

Table 1 shows the architecture of the neural network, how the dimension of the input data changes after each layer, and the values of some of the hyperparameters. Each of the 5 Conv1D layers is followed by a LeakyReLU activation function and a max pooling layer with kernel size 2 and stride 2. The squeeze layer only manipulates the dimensions of the tensor and can be thought of as taking the transpose of the output of Conv1D-5. In between Linear1 and Linear2 there is a Dropout layer which is only used during model training. Finally, a ReLU activation function is added after Linear2 to impose a non-negativity requirement on the final model output. For a more detailed description of the various layers and activation functions used, please refer to the PyTorch documentation (a link can be found under Additional Reading).

After each 1-D convolution layer, the number of channels, j, is either unchanged or increases and the signal length is unaltered. The subsequent LeakyReLU activation function does not alter the dimensions of the data. The final pooling layer in the sequence of 1-D convolution, LeakyReLU, and max pooling, halves the signal length (rounded down) and does not alter the number of channels.

Layer	Kernel Size	Stride	Padding	In Channels	Out Channels	In Signal	Out Signal
Conv1D-1	3	1	1	n_channels	20	45	22
Conv1D-2	3	1	1	20	20	22	11
Conv1D-3	3	1	1	20	40	11	5
Conv1D-4	3	1	1	40	40	5	2
Conv1D-5	3	1	1	40	80	2	1
Squeeze	-	-	-	80	1	1	80
Linear1	-	-	-	1	1	80	40
Linear2	-	-	-	1	1	40	n_sources

Table 1: 1-D Convolutional Neural Network Architecture for an input window of 45 minutes

2.2 Training, Validation, and, Testing Error

In order to train a neural network, we typically split the dataset into a training set, validation set, and testing set in order to minimize overfitting. The training set is the subset of data used to fit or optimize the model parameters (not to be confused with hyperparameters). The validation set is the subset of data used to provide an unbiased evaluation of the model fit while training the model. The test set is the subset of data used to provide an unbiased evaluation of the final model fit on the dataset. In this workflow, we split the dataset as follows, 70% for training, 15% for validating, and 15% for testing. The realizations for each set are also randomly selected using a set seed for reproducibility. It is also important to note that the initial model parameters are *randomly chosen* and that both the NumPy and PyTorch random seeds also need to be set in the script or notebook used to train the model if true reproducibility is desired.

Neural networks are typically trained iteratively using some form of gradient descent. Figure 1 shows the batch averaged mean-squared error versus epoch (or iteration) number for 3000 epochs. We can observe that the training error on average is monotonically decreasing whereas the validation and test loss appear to be slowly increasing after around 500 epochs. We can then reasonably assume that the model is overfitting the training data after 500 epochs which is something we would like to avoid. For the current workflow, I found that around 300 epochs gave the best results. Note that in this case we are plotting the test loss to show how both the validation and test loss increases with the number of epochs after a certain point. It is however best practice to avoid using the test loss to tune the hyperparameters during training as it should be an *unbiased* evaluation of the *final* model fit.

When making substantial changes to the model architecture, input data format, or output data format, it is highly recommended that the model is first trained with a large number of epochs to determine the optimal number. Other than reducing the number of epochs, numerous other methods are available to minimizing overfitting such as adding dropout layers.

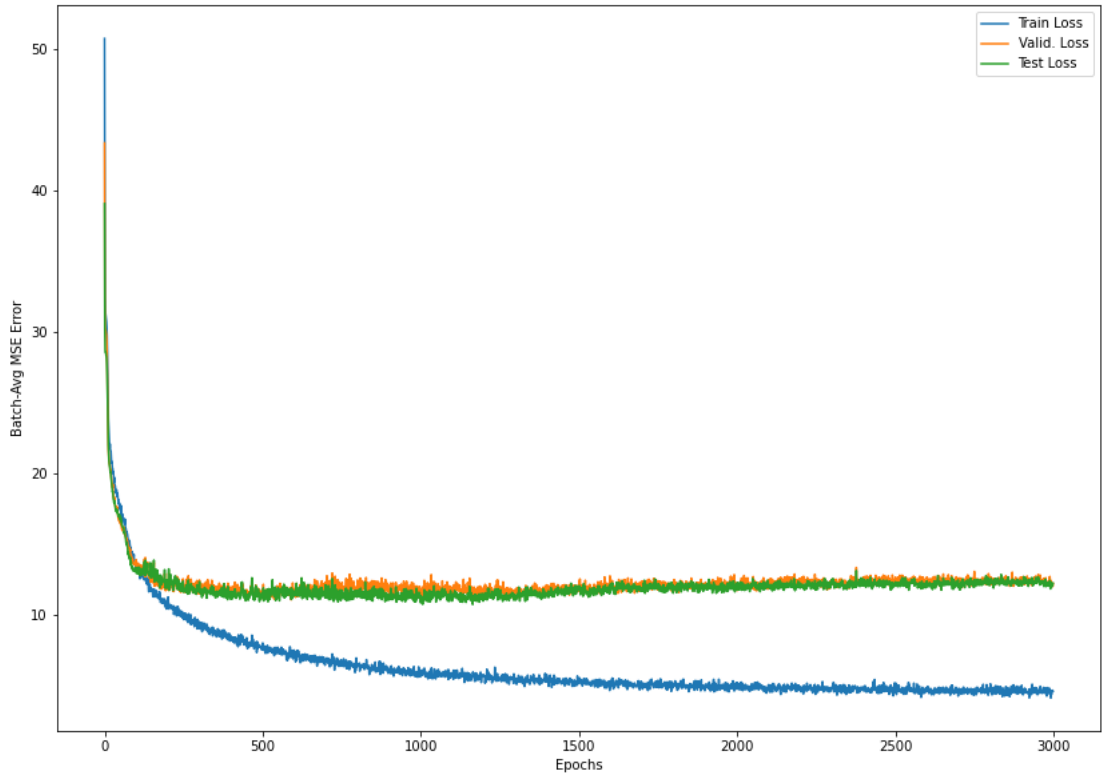

Figure 1: Batch Averaged MSE Error during training

2.3 Hyperparameters

A model's hyperparameters are parameters whose values are chosen by the user and cannot be inferred during model training (e.g. number of epochs from the previous section). In contrast, the model's parameters (or weights) are inferred through model training. Rather than providing an exhaustive laundry list of model hyperparameters that we can tune, this section will instead highlight the more significant ones specific to this model architecture. The hyperparameters referenced here can be adjusted in either `sift_conv1d_train.ipynb`, the notebook used to train the model, or `sift_conv1dnet.py`, which defines the model architecture.

2.3.1 Data Format/Structure

In a previous section, we outlined the dimensions and format of the input data expected by the model but neglected to describe one of the dimensions, batch size, which also happens to be a hyperparameter. Before training the model, the 3 data sets are further split into batches of equal size (typically much smaller than the size of the full dataset). The batch size refers to the number of samples (or in this case fakequake realizations) propagated through the model before updating the model parameters using gradient descent. Note that if the size of the dataset is not divisible by the batch size, the remaining samples can either be discarded or the last batch will contain less samples. The former is done in the current workflow as the PyTorch DataLoader does not support the latter. Implementing a custom function that can handle uneven batch sizes so that samples are not discarded could be beneficial.

Using batches has the advantage of requiring less memory and reducing the number of epochs needed to train the model. Unfortunately, I am not familiar on how changing the batch size affects model performance. All I know is that we don't want to make it too large or small relative to the full dataset. I am currently using a batch size of 20 but 32, 64, and 128 are common sizes depending on how large our dataset is. Also note that the number of channels and signal length can have a significant impact on model performance but are not considered to be hyperparameters.

2.3.2 Model Architecture

The hyperparameters with the most room for tuning are found within the model architecture. The 5 sequences of convolution, LeakyReLU, and max pooling layers make up the bulk of the neural network The structure of one sequence is defined below,

```
nn.Conv1d(in_channels, out_channels, kernel_size, stride=1, padding=1)
nn.LeakyReLU(negative_slope=0.01)
nn.MaxPool1d(kernel_size = 2, stride = 2)
```

The kernel size, stride and padding for the Conv1D layer should not be changed in order to preserve the signal length. Similarly, the kernel size and stride of the pooling layer should also be left as is in order to reduce the signal length by a factor of 2. The negative slope of the LeakyReLU is usually left at its default value of 0.01 and I'm not sure how much changing this will affect the results.

This leaves us with the number of input and output channels as defined in Table 1. This is loosely based on the 'encoder' part of a convolutional autoencoder as well as other examples of convolutional neural networks found in literature. It is not clear if this pattern gives the best results. Different channel increments with the same pattern (e.g. $3 \rightarrow 15 \rightarrow 15 \rightarrow 30 \rightarrow 30 \rightarrow 60$) were tested and had a significant impact on model performance. I don't have much intuition for what the right step sizes should be and the current selection was made through trial and error. I found that the output from the final convolution sequence should be relatively large compared to the total number of unit sources and the number of out channels in the first sequence should not be too large compared to the number of input channels.

The 2 linear layers after the convolution layers, defined below, can also be tuned to a lesser degree. The in_features of the first layer must be consistent with the out_channels in the final convolution sequence and the out_features of the last linear layer must be equal to the number of unit sources used in the inversion. This leaves us with the out_features of the first linear layer or in_features of the 2nd linear layer which must also be equal. The hyperparameter p in the dropout layer controls the probability of an element to be zeroed out during training. Both hyperparameters will require trial and error to determine their optimal values.

```
nn.Linear(in_features = 80, out_features = 40, bias=True)
nn.Dropout(p=0.3)
nn.Linear(40, n_sources, bias=True)
```

2.3.3 Optimizer

For this workflow we use the Adam algorithm as our optimizer. The main hyperparameter to adjust is the learning rate (lr) which controls the step size at each iteration (in this case after a batch is propagated through the model) in the descent direction. A too small learning rate could result in the optimizer getting stuck in a local minima or take too long to converge but a too high learning rate could result in jumping over minima. The optimizer in the training notebook is defined as,

```
optim.Adam(model.parameters(), lr=0.0005)
```

See the PyTorch documentation for a full list of parameters for Adam.

3 Modifying the ML Workflow

Modifying the ML workflow in PyTorch, such as the input data and the model architecture itself, requires minimal effort. This section will give a brief overview on how to adjust certain aspects of the workflow for future work such as the dimensions of the input data.

3.1 Adding/Removing Input Channels and Unit Sources

Changing the number of input channels and unit sources used is easy and does not require any changes to be made to the model architecture itself. Simply pass the correct parameters when defining the model in the training notebook as shown below,

```
model = sconv.Conv1DNN(in_channels, n_sources).to(device)
```

where in_channels refers to the number of input channels, j, and n_sources refers to the number of unit sources. Adjusting the input and output data can be done in `proc_fakequake.py` or in `sift_conv1d_train.ipynb` after the data is loaded but before it is split into batches and converted to tensors. Note that the neural network assumes the order of the input channels and unit sources are consistent for every realization and will not output an error if they are not. Also note that the hyperparameters may need to be re-tuned for optimal model performance when the input and output dimensions are changed.

3.2 Adjusting the Time Window

Adjusting the time window can be more of a challenge as it may require changes to the model architecture. Recall that the model expects the signal length at after all 5 convolution sequences to equal 1 and each max pooling layer reduces the signal length by a factor of 2. Therefore if a shorter time window is desired, we would have to reduce the number of convolution sequences since a max pooling layer with kernel size 2 and stride 2 applied to a signal length of 1 will result in an error. Similarly, if a longer time window is desired, we will need to increase the number of convolution sequences to ensure the signal length is reduced to 1. Once the appropriate changes have been made to the architecture, the window length is set in `sift_conv1d_train.ipynb` by adjusting the `twin` variable. The model parameters may need to be re-tuned after adjusting the time window. An example of a model architecture for a 30 minute time window can be found in `sift_conv1dnet.py`.

3.3 Modifying Model Architecture

The model architecture provided in this workflow is by no means guaranteed to be the best for the problem at hand. Layers can be removed or added and can have a significant impact on model performance. The choice of activation function, of which there are many to choose from, is also easily changed and similarly has a large impact on performance. See the PyTorch documentation for an exhaustive list of predefined layers and activation functions. Custom layers and activation functions can also be implemented. Unfortunately, this area will also require a great deal of trial and error given the uniqueness of this problem.

4 Future Work and Thoughts

Due to time constraints, there were a number of improvements to streamline the current workflow that I had planned to make but was not able to complete. These include making it more user-friendly to change the input and output data used to train the model, increased automation in the data ingestion and processing scripts, and defining a better error metric. A description of these improvements will make up the majority of this section. I will also briefly comment on a few issues I encountered while creating this workflow and how to potentially address them.

Finally, breakthroughs in deep learning techniques and neural network architectures are being made every day and could lead to significant increases in model performance. I unfortunately was unable to do a deep dive into the current deep learning literature but will briefly discuss some of my thoughts.

4.1 Streamlining Workflow

Optimizing model performance by tuning the hyperparameters, modifying the model architecture, and adjusting the model inputs and outputs involves significant amounts of trial and error. The goal in streamlining the workflow is to reduce the amount of time each trial takes. Again, this is not an exhaustive list of improvements to the workflow but just a few that stuck out to me during development.

One of the first major improvements to make is to store the DART buoy and/or forecast point information in a separate file that the scripts can load when needed. This includes the names and coordinates of the buoys and/or points as well as the list of unit sources used in the inversion. As the order of this information is important, having this information stored in a single file would ensure consistency among all scripts. Furthermore, this information is currently hardcoded into the scripts and modifying them would require the user to edit each individual script. To a lesser extent, the filepaths the scripts output to are also hardcoded and should be passed as a parameter when executing the script, particularly the 2 plotting scripts.

Next, when processing the fakequake inversions calculated by NCTR in `proc_fakequake.py`, the list of unit sources to be extracted is hardcoded into the script. The script will only extract unit sources with names explicitly provided by the user regardless of what the auto-inversion algorithm selects. This was done because the auto-inversion selected sources 44a and 44b for a handful of realizations which needed to be excluded. Below is a section of code that can be used to determine the range of unit sources selected by the auto-inversion for all 1300 fakequake realizations.

```
names = [ ]
runs = np.arange(1300)

for run in runs:
        fname = 'fq%s_autoinv_slip.mat' % str(run).zfill(6)
        fpath = os.path.join(matdir,fname)

        mat = loadmat(fpath)
        faults = mat['faults']
        slips = mat['slip']

        for i in range(len(slips)):
                name = faults[0][i][0]
                wt = slips[i][0]

                if name not in names:
                        names.append(name)
```

Incorporating this into `proc_fakequake.py` should be straightforward if you wish to automate the unit source selection but the flexibility of excluding specific sources could also be desirable.

Finally, a more meaningful quantitative metric needs to be developed in order to quickly evaluate the model performance as it is difficult to interpret the current metrics. This should ideally be a single or small list of numbers or even a handful of plots. Comparisons of model performance are currently done by looking at the final mean-squared error, plots of the time series at the DART buoys and unit source weights, and time series comparisons and scatter plots of the maximum amplitude at the additional forecast points.

The mean-squared error is difficult to interpret for many reasons. For example, a prediction which places a large slip on a predominantly on-land subfault would have high error but the resulting time series and far-field forecasts may be satisfactory. The individual time series plots for each realization at the DARTs and forecast points take a non-trivial amount of time to generate and it is incredibly difficult to objectively compare performance between different models by examining hundreds if not thousands of plots. The scatter plots of the predicted vs. observed maximum amplitudes at the forecast points are easier to interpret but may not be the best metric to judge model performance.

Training the model for the current data set and architecture takes significantly less time than generating all of the plots. Furthermore, examining each plot one-by-one is a time-intensive and difficult process. As I've mentioned countless times already, the process of training a model is one of trial and error. The current workflow is incredibly inefficient for fine-tuning the hyperparameters due to the current available model performance metrics. In my opinion, this should be one of the *highest* priorities for any future work on unit source inversion.

4.2 Temporary Workarounds

There are 2 minor details in the current workflow that I felt should be documented. The first is that the forecast point chosen in the Strait of Juan de Fuca does not exist in the NCTR propagation database and hence is excluded in the plotting scripts. Among the forecast points, the majority are farfield so selecting another suitable nearfield point may be of interest.

Secondly, realization 551 was excluded from the dataset as it is truncated. We believe that the auto-inversion script may have run out of memory and stopped outputting the time series partway through however this has not been confirmed. Rather than trying to figure out whether the data is still valid since the model does not require the full 24 hours, I decided to exclude it for now (see `proc_fakequake.py`).

4.3 Breakthroughs in Deep Learning

As mentioned already, breakthroughs in deep learning are constantly made and it is oftentimes difficult to keep up with the rate at which new machine learning papers are published. In fact, convolutional neural networks are considered by some to be dated although they still perform well with time series data. Some ideas for future development include recurrent neural networks and transformers. Unfortunately due to time constraints, I was not able to spend much time exploring the current literature. A few Wikipedia articles which cite relevant papers can be found below.

5 Additional Reading

(1) PyTorch Documentation: https://pytorch.org/docs/stable/index.html

(2) Official PyTorch Tutorials: https://pytorch.org/tutorials/

(3) Recurrent Neural Networks: https://en.wikipedia.org/wiki/Recurrent_neural_network

(4) Transformers: https://en.wikipedia.org/wiki/Transformer_(machine_learning_model)

(5) Effect of Batch Size: Keskar, Nitish Shirish, et al. "On large-batch training for deep learning: Generalization gap and sharp minima." arXiv preprint arXiv:1609.04836 (2016).

(6) Using dropout to prevent overfitting: Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." The journal of machine learning research 15.1 (2014): 1929-1958.

(7) Introductory book on deep learning: Goodfellow, Ian, et al. Deep learning. Vol. 1. No. 2. Cambridge: MIT press, 2016. (can be found online here https://www.deeplearningbook.org/)