Deep Learning for SIFT Unit Source Inversions

Christopher Liu

June 9, 2021, Revised:

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.

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 parameters 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. for an input time window of 45 minutes and a time series with an increment of 1 minute, $k = 45$).

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.

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

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.

2.2 Training, Validation, and, Testing Error

In order to train a neural network, one typically splits their 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 tuning the hyperparameters. 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.

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.

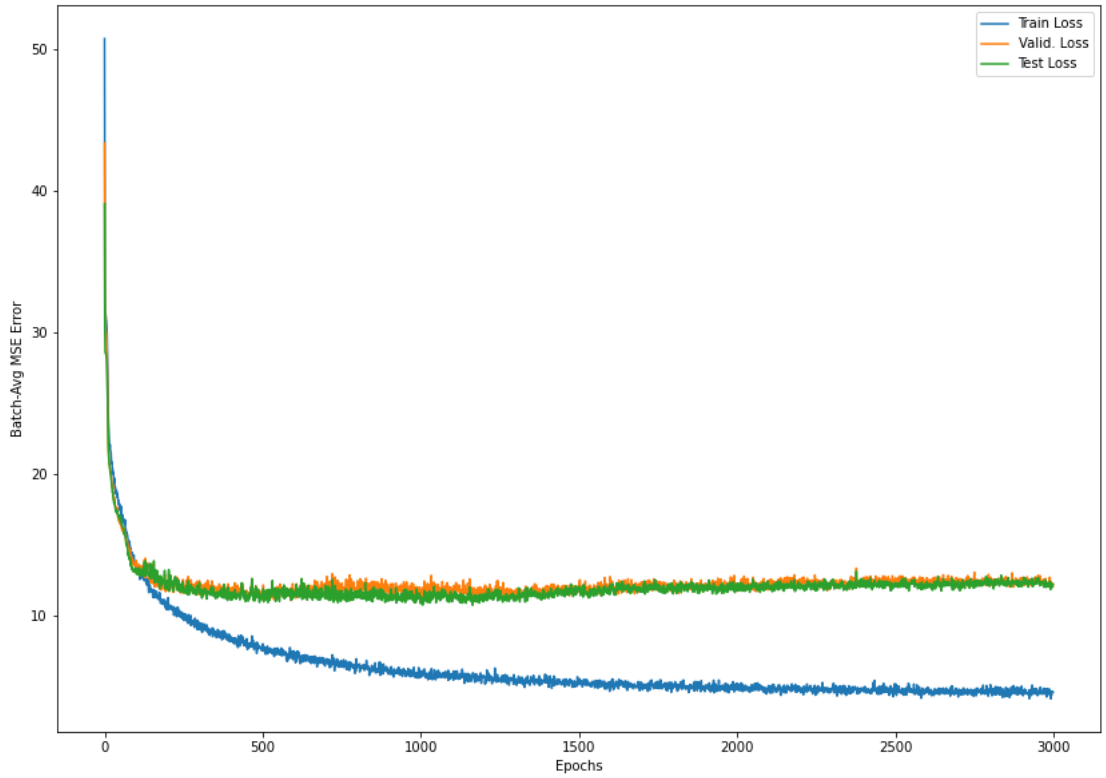

Figure 1: Batch Averaged MSE Error during training

2.3 Hyperparameters

A model's hyperparameters are parameters whose values are chosen by the user and cannot be inferred during model training. In contrast, the model's parameters (or weights) are inferred through model training (e.g. number of epochs from the previous section). 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.

This has the advantage of requiring less memory and reducing the number of epochs needed. 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. 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 and are 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, such as the input data and the model architecture itself, in PyTorch 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 neural network itself. We just need to 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 the end of all convolution sequences to be equal to 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.