Applications of Deep Neural Networks in Seismology

Author:
Noah Grayson Luna

Supervisors:
Prof. Dr. Heiner Igel
Dr. Qingkai Kong

A thesis presented for the degree of
Master of Science

Department of Earth and Environmental Sciences
Ludwig-Maximilians-Universität München
& Technische Universität München
12 August 2019

Declaration of Authorship

With this statement I certify that this master thesis has been composed by myself. Unless otherwise acknowledged in the text, it describes my own work. All references have been quoted and all sources of information have been specifically acknowledged. This thesis has not been accepted in any previous application for a degree.

Hiermit versichere ich, diese Masterarbeit selbständig und lediglich unter Benutzung der angegebenen Quellen und Hilfsmittel verfasst zu haben. Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Signature	Date

Acknowledgements

It goes without saying that this thesis came to fruition thanks to several key players. Special thanks to my thesis advisor Dr. Heiner Igel, Professor in the Department of Earth and Environmental Sciences of the Ludwig-Maximilians-Universität, München. Heiner found the perfect balance of challenging me to think critically of our work but at the same time providing supporting advice. It was a pleasure to work with Heiner and I would be so lucky to have the opportunity to collaborate with him in the future.

Tremendous thanks goes to Dr. Qingkai Kong, Assistant Data Science Researcher Jointly at Berkeley Seismology Lab & Berkeley Divisions of Data Sciences. Throughout this thesis Qingkai provided guidance and support for all my machine learning questions. His work using Machine Learning algorithms for Seismic applications first ignited my interest in the subject. This spark of fascination inspired me to return to Academia and seek my own project in the same field. Thanks, Qingkai!

A heartfelt thank you to my dearest friends: Arthur, Charley, Jason, Jennifer, Dr. Graham (Steve), Rebeca, and Tommy. You all have (and continue) to bring absolute joy into my life.

My last words of gratitude go to my small, but incredible family. To my brother, Cesar and my parents, Miguel and Hortencia, thank you for being a continuous source of inspiration, love, and support.

Abstract

Only within the last decade has the potential of deep neural networks begun to surface. Improved computational resources, data storage, and breakthroughs in the theoretical framework driving machine-learning, particularly deep learning has allowed researchers across disciplines explore the usefulness of these algorithms. The geophysics community is in the early stages of exploring the applicability of deep neural networks in seismology. We participate in this endeavor by applying two different deep neural networks on two different challenges facing seismology today: improving the signal-to-noise ratio of seismic data and S-phase picking. In both cases we were able to develop and implement the entire machine learning workflow. This includes downloading, pre-processing, converting the data into a workable format, training, and validating the model via a test set. Both deep neural networks demonstrate promising results thus far; with recovered SNR in the case of our denoiser and accurate phase picks for our S-phase picker. The current models are baselines which will later be improved as we continue to implement various machine learning procedures and techniques. The pipeline for transfer learning has been developed and tried as well for both deep neural networks.

Thesis Structure

This thesis covers two independent machine learning projects. As such, it made sense to allocate separate and self-contained chapters for each project. Thus allowing the interested reader to learn about either project without having to refer to the other.

With that stated, the Thesis is structured as follows: Chapter I is devoted to the motivation of this thesis and current state of Machine Learning in the field of Seismology. Chapter II provides some theoretical background on topics critical to Machine Learning and Deep Learning. Chapter III goes over basic concepts of Seismology. This chapter is mostly geared to provide the reader with background knowledge of rotational measurements, as it is not a topic still not yet widely discussed in the Seismological community.

Chapters IV and Chapter V are devoted to the thesis projects. To mimic the flow of a scientific paper we refrained from restating (in depth) definitions and theory since said topics are covered in Chapter II. We cite which sections to refer to in order to obtain the technical details as they come along in Chapters IV and V.

Note: Chapter II provides sufficient information to understand what was done for this thesis. However, by no means does this cover enough theory to practice Machine Learning. The interested reader should understand the statistical framework that allows Machine Learning to be possible. This section was omitted for this Thesis in order to keep the thesis as short and 'to the point' as possible.

Contents

List of Figures

Chapter 1

Introduction

Applying machine learning algorithms (ML) in seismology is gaining rapid momentum. With readily available computing resources and troves of seismic data now out of human processing reach, seismologists are turning to artificial neural networks to gain new insights.

As geophysicists, we have the theory, intuition, and logic to analyze data, but Machine Learning can do something that we as humans cannot. Machine Learning can work beyond our geophysical intuition and facilitate the discovery of unconsidered patterns.

In a nutshell, machine learning is a set of techniques that use computational methods to learn information directly from data without relying on a predetermined equation. ML algorithms learn from data using probabilistic theory. The goal of these algorithms is to adaptively improve its performance as you give it more data to learn from.

Thus far machine learning algorithms have produced impressive results in a host of different seismological topics. [33] developed a machine learning algorithm which can detect and locate earthquakes given a recording from a single station. To do this they trained a convolutional neural network that can make predictions of which spatial region an event occurred.

At the University of California, Berkeley researchers have developed a smartphone application capable of detecting earthquakes given a user's built-in accelerometer on their smartphone. The goal is to one day incorporate this information to increase the seismic network density in California which is key for a reliable Earthquake Early Warning system [28].

In addition to earthquake detection and location, progress has been made in the field of Seismic Hazard Analysis. [18] used a hybrid method combining neural networks and simulated annealing in order to create a model capable of predicting the peak time-domain characteristics of strong ground-motions. Their combined model approach was successful in providing reliable estimations of PGA, PGV and PGD values. This allowed them to make precise estimations of the site ground-motion parameters for their case studies.

The above-mentioned applications are but a subset of how machine learning is currently being applied in the field of Seismology. In addition to successes in estimating ground motion prediction parameters, earthquake detection, and earthquake location, we have also seen impressive results using ML for subsurface tomography and aftershock pattern recognition.

With the above mentioned success of applying machine learning algorithms for seismic applications we set out to rethink how we view two topics in seismology: denoising seismic data and single-station location using 6 degree-of-freedom (Dof) data.

Dating since at least the start of Modern Seismology in the 1960's it has been understood that observations made about Earth's processes and underlying structure must be done in the presence of background noise. To this day there does not exist a 'best method' in removing noise. Given the developments using deep neural networks for speech-recognition (such as those used in smartphone applications), we set out to use a similar methodology but with seismic data. This topic is described in Chapter IV.

There is growing intrigue in six-degree-of-freedom (three translational and three rotational component measurements) in the seismological community. Among other advantages, 6Dof theoretically can lead to single-station location with higher accuracy and less computational efforts than current-day practices [22]. However, applying theory to true measurements has been non-trivial. Machine learning algorithms can potentially provide new insight into patterns and features we as humans may be oblivious to. As such, we were curious to investigate what machine learning might be able to tell us about rotational motions in conjunction with translational observations. We'll dive in greater detail into the theory and motivation behind this machine learning project in Chapter V.

Chapter 2

Fundamentals of Machine Learning

2.1 What is Machine Learning?

In 1959 Arthur Samuel, a pioneer in the field of Artificial Intelligence described Machine Learning as “the field of study that gives computers the ability to learn without being explicitly programmed.” Machine learning is a subset of the Artificial Intelligence field which focuses on algorithms that use computational methods to “learn” information directly from data without relying on a predetermined equation. The goal of said algorithm is to adaptively improve its performance as you give it more data to learn from. This definition leads to one of the fundamental differences between machine learning and statistical learning[1]; statistical learning requires explicit ‘rule-based’ instructions whereas machine learning does not. In addition, statistical learning is generally applied to smaller data sets and attributes (more on this later). That being said, one cannot stress enough the importance of the field of statistical learning; it is the theoretical basis for machine-learning algorithms.

2.1.1 Why Use Machine Learning?

There has been a lot of hype regarding machine learning in the last decade so it goes without saying that some justification on why one would use machine learning needs to be stated. Machine learning is an attractive option when: one’s dataset is both large and complicated, when implementing a solution requires an unreasonable amount of explicitly coding rules, and when traditional methods (such as statistical learning) fail to provide a reasonable solution or a solution at all. The reason for the above is as follows (in order by which they were stated): machine learning can reduce the amount of code needed to maintain the original non-machine learning approach, given the tendency of larger data sets (made possible by modern-day hard-drives, computing clouds, etc.) machine learning can help find patterns we as humans could not detect given the size of the data set, and finally machine learning can help find a solution to complex problems such speech recognition (manually doing this would

[1]A framework for studying the problem of inference, that is of gaining knowledge, making predictions, making decisions or constructing models from a set of data [24]

require writing code which could have every rule of pronunciation of the English language, syntax rules, regional dialect variations, etc.)[15].

2.1.2 Machine Learning in a Nutshell

In Section 2.1 machine learning was defined as a set of algorithms by which a computer can learn on its own without explicit programming. More specifically the machine learning algorithm is finding an approximation function g which approximates an unknown function f known as the *target function.* The target function maps a set of inputs(observations) $\mathbf{x}$ onto the set of possible outputs $\mathbf{y}$, $f : X \to Y$. X and Y are the input space (the set of all possible inputs) and output space (the set of all possible outputs), respectively [3].

To find this approximating function g the algorithm is given*training examples*, which are instances from the input space X. In addition we (normally) provide the associated output value associated with said input. In other words the data set D consists of input-output pairs $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_n, y_n)$ for $n = 1, 2, ..., N$.[3].

During the learning process, or *training*, the algorithm will search and test different approximation functions g from a set of candidate options known from the hypothesis set, H, using the training examples as inputs.

In other words, during training the approximation function, g will be given input-output pairs $(\mathbf{x}_n, y_n)$ from the data set D. For each iteration, it will fine-tune this function and return the appropriate output y for the given input $\mathbf{x}$. The approximation function g used for future predictions is the function which best minimizes the error between the true function f and g.

2.2 Can ML Algorithms Learn?

The question of whether or not a machine learning has learned boils down to answering one fundamental question: can the model generalize? In other words, can the machine learning algorithm make accurate predictions on data it has never seen before? During training, the machine learning is referring to the training data to update dates which will (hopefully) reduce the prediction error. At the end of each cycle, the ML algorithm refers to a validation set, which is a subset of data set aside to mimic an *out-of-sample error.* An out-of-sample error is the error found when the machine learning algorithm makes a prediction it has not seen during training.

During training the neural network is trying to find the best approximation to an unknown function f known as the target function using algorithms which are directly learning from the data set, D. The target function is a function of the entire input space χ and the output space Y. Our approximating function, g, is based only a subset of the entire input-output space, D. Thus, despite the outputs of the initial training, one can not asses whether or not the ML algorithm has learned the true target function since it is only looking at a subset of the entire input-output space χ during training [3].

To find out if the ML algorithm can infer outside the data set it was originally given, the pre-trained model is run on a data set it has never seen before. This data set is known as the test set. The test set consists of instances which are outside of our data set D. If the trained neural network can make accurate predictions on data it has never seen before, then one can interpret this as indicating the neural network has learned. Said another way, the approximating function has found weights that can fit the observations found during training but are not too specific such that it can not *generalize* to data it has not seen before.

2.2.1 Validation and Test Set

For the above reasons, part of the machine learning workflow involves splitting the data set into three pieces: a training set, a validation set, and a test set. Generally, 70% of the data is used for training, 20% is used for the validation set, and the remaining 10% is used for the test set.

2.3 Types of Learning

There are several Machine Learning architectures and categories, each of which are appropriate for different data sets and objectives. For example, Google Assistant, the A.I. - powered virtual assistant which can take in as input both text and speech, takes advantage of a subset of Recurrent Neural Networks (RNN). RNN are a type of Deep Neural Network (more on this later) which have been largely successful in handling data where the order of the input is important such as in speech and text comprehension. Because of the complex problems Deep Neural Networks (DNN) are designed to tackle, DNN requires significantly more data and computation power. Therefore, diving into the Deep Neural Network world might be overkill and, more importantly, a waste of computing resources and storage space if your objective could be solved with less-expensive Machine Learning approaches.

Broadly speaking one can categorize Machine Learning systems based on whether they are: supervised, unsupervised, or semi-supervised; whether they make predictions or only compares examples with what it has already seen (instance-based learning versus model-based); whether the Machine Learning algorithm learns by taking in batches of examples or one-by-one (batch learning versus online learning); and whether the desired output is a category or a continuous value such as an integer or floating-point number (regressor vs. classifier). Given the focus of this thesis (and for brevity's sake), attention will be spent on understanding the difference between unsupervised and supervised learning. Additionally, a definition of what is a regressor and classifier machine learning algorithm will be covered.

2.3.1 Supervised vs. Unsupervised

The objective of the Machine Learning algorithm is to find a model (our approximation function g) which can map an input $\mathbf{x}$, $x \in X$, to their responses $\mathbf{y}$, $y \in Y$. [24] In supervised learning the data set consists of the training data, x_n, and its associated output measurement, y_n.

This is a contrast to unsupervised learning where only the input $\mathbf{x}$ is available for our algorithm to learn from. Intuitively one can already imagine that the unsupervised approach is much more complicated. For example, imagine we wanted to train a Machine Learning algorithm to classify cat and dog images; If we give it an image of a cat it will return 'cat', and if it is a dog it will return 'dog'. To do this we could create an algorithm that takes in images of cats and dogs as inputs and also provides it with labels for each of these images (i.e. 'cat' for cat images and 'dog' for dog images). Thus, during training our algorithm will know what answer it should come up on its own. This is contrary to unsupervised learning where the associated output for a given input is not provided.

Both of the above mentioned have their pros and cons. Unsupervised learning generally requires more data and computing resources because the algorithm must figure out on its own what the answer should be (more on this in Section 2.3.1). Supervised learning generally requires fewer data and less computing power (since the answer is provided to begin with). However, in this case, the algorithm is more restricted in what it can output because of the bias it is being supervised on. In addition, there is the logistical headache of providing the answer/labels for each input, $\mathbf{x}_i$.

2.3.2 Regression vs. Classification

Under the Supervised Learning umbrella category[2], there are two categories of response variables (y) used: *regressors* and *classifiers.*

Regressors are real-valued response variables in the form of integers or floats, i.e. they are *quantitative.* Classifiers, on the other hand, use categories as response variables and are thus *qualitative* in nature.

Let's look at a couple of examples to make this more clear. Let's start with understanding classifier response variables. Imagine we want a Machine Learning algorithm to classify images. Specifically, it is desired to have an algorithm print "cat" when it is given an image of a cat and will print "dog" when it is given an image of a dog. To train an algorithm to perform said task, the *classifier* is given thousands of examples of images of cats and dogs. Each image is a training example, $\mathbf{x}_i$, and the entire collection of images is the *training set.* Since this is supervised learning, labels/response variables, y, are known and given to the machine learning algorithm to learn from. During training the algorithm will take in an image $\mathbf{x}_i$ and its associated response variable y_i and it will predict whether or not it should print "cat" or "dog". If it predicts correctly based on its associated label, it will strengthen the parameters (or weights) which helped contribute to the correct outcome. If it predicts incorrectly, it will adjust its weights to (hopefully) predict correctly the next time. Given enough examples (and correct algorithm set-up) our algorithm will learn how to appropriately label the images. This is a textbook example of a Supervised Machine Learning Classifier.

To understand regressor output variables, imagine instead that it is desirable to have an algorithm that predicts income ranges given a persons' education, gender, sex, and age. Each person is a single training example, $\mathbf{x}_i$, and the entire collection of individuals (whose information we gathered) defines are training set. The associated labels/response variables, y_i, (since this is again a supervised learning task) which in this case are yearly incomes in the form of integer values are provided. Unlike the classifier output variable before, the values here are quantitative and not qualitative. The same logic for training follows as the previous example: the algorithm will take in examples and compare its predicted yearly income value with is provided then it will iteratively attempt to find a mapping function which minimizes the error between the known true values and its predicted value.

[2]We focus our attention on supervised learning given that both applications fall under said form of learning.

2.4 Assessing Model Accuracy

In order to quantitatively describe how the Machine Learning Algorithm behaves on the training, validation, or a test set, a performance measure must be defined. The type of Machine Learning algorithm used will narrow the field of performance measures to choose from. Common evaluation measure for Classifiers are accuracy, ROC curves, and logarithmic losses. There are about four common error metrics used for regression problems such as: Mean Absolute Error, Mean Squared Error, and Mean Squared Logarithmic Error.

This then leads us to the question of whether or not the sweet spot of training our model based on the data set has been reached such that the approximation function does a decent mapping between the inputs and outputs, but at the same time we wish our function to be able to generalize. This question is explored in Section 2.4.2

2.4.1 Defining the Error Metric

Before diving deeper into different Machine Learning performance measures (i.e. error metrics), let us formally *errors* in the machine learning context. In Section 2.1.2 the notion of a Hypothesis Set, H, was introduced. It was defined as the set of possible options the approximation function g can have. In other words, $g \equiv h \in H$. The error quantifies how well the hypothesis, h approximates the target function, f [3]:

$$\text{Error} = E(h, f)$$

Careful consideration needs to be taken into account when listing possible error measures. Some metrics are appropriate for certain Machine Learning tasks. Additionally, some error metrics can influence the outcome of the learning process. Only a handful of error metrics are defined here for brevity sake. The error metrics defined below are the four most common error metrics used for Regression problems (see Section 2.3.2): Mean Absolute Error, Mean Squared Error, and the Mean Squared Logarithm Error.

The Mean Squared Error (MSE) is defined as the average,squared difference between the true response variable , y_i and the predicted value from the approximation function $g \equiv \hat{f}$ [24]:

$$MSLE = \frac{1}{n}\sum_{i=1}^{n}(\log y_i - \log \hat{f}(x_i))^2 \tag{2.1}$$

Where i stands for the ith observation. MSE is perhaps the most common error measure for regression. It works well when the distribution of the response variable y has a Gaussian distribution. Note from the equation that larger errors are penalized more than smaller ones (as a result from the squaring). Thus, using the Mean Square Error may not be appropriate if the training and test data set have large outliers.

The Mean Absolute Error Lost (MAE) can be defined as:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{f}(x_i)| \tag{2.2}$$

In other words, it is the absolute difference between the true response and the predicted value. MAE is useful to use when the distribution of our response variable is mostly Gaussian-like, but not quite. This allows us to have outliers in our data sets without causing huge penalties as is the case with MSE.

Finally, the Mean Squared Logarithmic Error, or MSLE, can be defined as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{f}(x_i))^2 \tag{2.3}$$

The MSLE first calculates the natural logarithm of the predicted values and then calculate the mean squared error. Taking the natural log first avoids punishing the model heavily in the event that the target values are spread out. In other words, MSLE can be a useful error metric to use when there are large differences in large predicted values.[19]

It should be stressed the initial error found during training describes how close the algorithm is to predicting the true target value. However, at the end of the day the error found from making predictions on never-before-seen data (e.g. the test set) tells us if the algorithm can generalize (2.2). The *training error* helps guide the choice of hyper-parameters, neural network architecture, activation functions, and so on. However, the training error will not ensure that the *test error* will be as low as possible.[24] This leads to the topic of *overfitting* and *underfitting.*

2.4.2 Avoiding Underfitting and Overfitting

The goal is to train an algorithm that learns features which will help it accurately predict on never-before-seen data. In other words, it can generalize (see Section 2.2). When using complex Machine Learning models (such as Deep Neural Networks which can consists of thousands of weights/parameters) there runs the risk of training the algorithm such that it uses an approximation function g which has a *low* training error but a *large* test error (see Section 2.4.1). In other words, the Machine Learning model is *overfitting.* Overfitting can occur when the training data set is too small (relative to the size of the model) and/or is too noisy (i.e. the data has outliers and/or errors) [15]. Common solutions to avoid overfitting include: using more data for training, removing data with errors (also known as noise reduction), and using simpler Machine Learning architectures.

One technique to avoid overfitting is to reduce a complex Machine Learning model to a simpler model. This is known as *regularization.* Regularization comes in many shapes in forms. For linear models regularization takes the form of constraining the weights both in how many are used and their contribution to the overall algorithm. Some common linear regularization approaches include: Ridge Regression, Lasso Regression, and Elastic Net [15]. The interested reader should check out Aurélien's book [15] to learn more on linear regularization approaches (or one of the many wonderful online resources available for free).

The scope of this Thesis deals with Deep Neural Networks, thus the regularization techniques of interest here are: Early Stopping, L_1 and L_2 Regularization, Dropout, and Max-Norm Regularization. Some of these methods, such as the L_1 and L_2 Regularization and the Max-Norm Regularization are similar to linear regularization approaches in that they attempt to constrain the number and size of the weights. Early Stopping, on the other hand, avoids overfitting by terminating the training process when overfitting begins. More specifically, the training is terminated at the point where the error on the validation set begins to increase (and continues to increases) while though the error on the training set is decreasing[4].

Dropout, perhaps the most popular regularization tool used for Deep Neural Networks, is designed to prevent overfitting from occurring by randomly setting a number of layer outputs to zero. Essentially, we introduce another hyperparameter which assigns a probability that the output layer is retained. The reason being that Neural Networks can develop a co-dependency amongst each other during training. Specifically, during training some layers may adjust in order to correct for the mistakes of previous layers. This co-dependency will make the model fit the training data but will the restricted in how it can generalize to unseen data [41].

Overfitting can be seen as an end member case of training going awry, the other end member being underfitting. In this scenario the Machine Learning algorithm fails to capture the underlying trend of the data. In other words, the error from the training and validation set will remain more or less static or possibly increase during training. One can attempt to avoid underfitting by using a more complex model (e.g. more layers in the case of Deep Neural Networks), reducing the amount of regularization used (if it is implemented), and/or being more selective on the training data used[15].

2.5 Deep Neural Networks

Overview

This would not be a proper overview of Machine Learning overview the earliest machine learning architectures, perceptrons were not discussed. Perceptrons can be viewed as building blocks which can be combined to make a shallow neural network.

Training these shallow (and larger) networks is done via gradient descent optimization techniques, and back propagation. The architecture of Deep Neural Networks will be covered once the aforementioned topics are covered. The last sections of this chapter will be devoted to what Convolutional Neural Networks and Autoencoders are and how they work. The latter two models are the model architectures used for this thesis.

2.5.1 Shallow Neural Networks

One of the simplest machine learning architecture is the perceptron. Having a firm understanding of how perceptrons work will make the transition to shallow and then deep neural networks fairly straightforward.

The perceptron was invented by Frank Rosenblatt in 1957. It consists of a single input layer, where each input node is connected to a weight, and an output node [15]. For a given training instance $(\mathbf{x}_i, \mathbf{y}_i)$, the perceptron will compute a weighted sum of its inputs and then applies a step function to the computed sum.

$$\mathbf{h}(\mathbf{x}) = step[\sum(\mathbf{w}_i * \mathbf{x}_i)] \tag{2.4}$$

Our simple perceptron can be used as a linear binary classifier; given a training example with n many features, the perceptron will computed a linear combination of the inputs. If the threshold is exceeded, it will output a positive class else it will output a negative class.

geron step function

Training a perceptron is fairly straight forward: one at a time the Perceptron is given a training instance (input), the input is multiplied by the weight, the product is then fed through some threshold function. The output of which is a prediction. The difference between this prediction and the true observation is computed and used to reinforce connecting weights that contributed to the correct prediction [15]. The learning rule can generically be defined as:

$$\mathbf{w}_{ij} = \mathbf{w}_{ij} + \eta(\hat{\mathbf{y}}_j - \mathbf{y}_j)\mathbf{x}_i \tag{2.5}$$

The step function serves as an activation function. The activation function used determines which neuron should fire or not. There are a plethora of activation functions. The choice of activation function is influenced by what the objective of the machine learning model is [4]. Some popular activation functions including the: logistic function, hyperbolic tangent function, and rectified linear units (ReLU). Section 2.5.2 covers some of the more popular activation functions.

In many cases there is an invariant part of the part of our prediction called the *bias* [4]. The bias term allows the activation function to be shifted left or right in order to better fit the data. For example, suppose we have an identity activation function, $h(x) = x$. Multiplying the weight *theta* with the input $\mathbf{x}$, would gives us a linear function starting from the origin.

During training weight theta is updated. In other words, the gradient of the function becomes either flatter or steeper with each update. It may be the case, however, that the fit to the true prediction might improve if we could also translate (shift) our linear function by some constant. To do this a bias term is added to the weighted sum and fed through the activation function. And thus, a better fit to the observations can be made.

Multi-layer Perceptron

While a single perceptron might serve as a binary classifier, we can begin to build more complicated architectures by 'stacking' them. Doing so gives us a *Multi-Layer Perceptron.*[15] Multi-Layer Perceptrons (MLPs) consists of one or more input layers and several activation functions. The layer with our activation functions is known as a *hidden layer.* Three or more hidden layers define a *Deep Neural Network.*

The overall process of training an MLP is the similar as a single Perceptron. We begin by giving our neural network a training instance, this then moves forward to the next layer which is our hidden layer. Here we multiply by the weights, the output then goes to our final layer which then outputs a prediction, and we end *forward pass* by computing the error between our prediction and the true value.

This is where the similarities begin to diverge. To update the weights in our hidden layers, we go back one layer at a time (i.e. via the *reverse pass*) and compute how each connecting weight contributed to the overall error. With this information we make update the weights to reduce the error [16].

This process of moving backwards, one row at time, and computing how much each neuron in the last hidden layer contributed to each output neuron's error is known as *back-propagation.* Before the introduction of backpropagation, researchers struggled to train multi-layer perceptrons. Without it, and advances in computation power and storage capabilities, machine learning struggled for years to get its wings of the ground. Given the importance of backropagation we will describe the maths in some detail in Section 2.5.3.

2.5.2 Activation Functions

There are a handful of activation functions floating around. Some of the most popular activation functions inclduded: step functions, linear functions, sigmoid function, hyperbolic tangent, and rectified linear units (ReLU). Linear activation functions will be discussed first along with an overview of some of their shortcomings. Afterwards hyperbolic tangent and ReLU will be covered as they are the most widely used today.

Recall that the purpose of an activation function is to output the sum of the products of the incoming inputs and weights [4] in some hidden layer l for some node i:

$$\mathbf{y}_i = \sum_{i=1}^{N}(w_i * \mathbf{x}_i) + bias \tag{2.6}$$

For example, one could define a liner activation function (sometimes referred to as an identity activation function), which has the form:

$$a(\mathbf{x}_i) = c * \mathbf{x}_i \tag{2.7}$$

But there are two drawbacks using the linear activation function in our hidden layer. First, the derivative of a linear function is a constant, which means that it has no relationship with X. More importantly, this means we have a constant gradient. Thus, the weight updates made by back propagation (see Section 2.5.3) will be constant and independent of the input. Second, we state (without proof) an important theorem [4]:

Theorem 1 *A multi-layer network that uses only the identity activation function in all its layers reduces to a single-layer network performing linear regression.*

In other words, a combination of linear functions is still a linear function.

Why does this matter? Remember that in a neural network the output of one layer is now the input of the next. Instead of using a standalone approximation function g, as is the case non deep learning algorithms, deep neural networks use composite functions.

Composite function give machine learning algorithms the ability to model more complex functions than the individual functions of which it is made of. Thus, being left with a linear function at the end will give the machine learning algorithm flexibility to model only linear functions. (Which probably be of any use if the objective is to model complex process). For this reason it is desirable to use activation functions which are nonlinear and that, when combined, will give the neural network more expressive power [4].

Given the above knowledge, the hyperbolic tangent function is an attractive activation function because it is: a) differentiable, b) continuous, and it is c) non-linear (so they can be stacked and be used to model complex functions):

$$A(\mathbf{x}_i) = \tanh(\mathbf{x}_i) = \frac{2}{1+e^{-2\mathbf{x}_i}} - 1 \tag{2.8}$$

Note that the output is restricted to values between -1 and +1, giving a mean centered at 0. Which means that the output of each layer will be semi-normalized, and thus speeds up convergence [15].

Another activation function of interest is the rectified linear unit (ReLU) function. It has the form of:

$$A(\mathbf{x}_i) = max(0, \mathbf{x}_i) \tag{2.9}$$

This is also an attractive function because combining them forms a nonlinear function. It is not however continuously differentiable (note the kink at z = 0). And this abrupt change in slope can make Gradient Descent bounce around. But, there is a distinct advantage to ReLUs, their sparsity. Given that our ReLU will occasionally spit out a 0, the density of the neural network will be less. In other words, we have less neurons which are giving an output signal. Which means that computationally speaking, there are less computations for both the forward pass and backward pass. This sparsity does not exist when all of our neurons print out an output which would be the case if a sigmoid or hyperbolic tangent functions were used.

That being said, the ReLU does encounter the problem of vanishing gradients. There could be a scenario where too many of the neurons in the model go to 0. Which means that the neurons will be unresponsive to updates because their gradient is 0 (and thus we make no weight update).

With that said, both the hyperbolic tangent and ReLU activation functions are widely used and perform fairly well. As a rule of thumb one begins with ReLUs as they are 'general approximator'. Sigmoid activation functions (not discussed here) are more well-fitted for classification.

2.5.3 Backpropagation

Backpropagation looks intimidating, tut having a firm understanding of how it works allows the machine-learning practioner to avoid some costly mistakes when designing their model architecture.

Our data set can be defined as the set of pairs ($\mathbf{x}_i$,$\mathbf{y}_i$) where x, y are elements of real numbers where $n = 1$ to k. Given m training examples, the cost function at the lth layer can be defined as:

$$J(\mathbf{w}, b) = \frac{1}{m}\sum_{i=1}^{m} J(\mathbf{w}, b; \mathbf{x}_i, \mathbf{y}_i) + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(\mathbf{w}_{ji}^{l})^2 \tag{2.10}$$

using the mean squared differences (Section 2.4.1) gives:

$$J(\mathbf{w}, b) = \frac{1}{m}\sum_{i=1}^{n} ||\mathbf{y}_i - \hat{f}(x_i)||^2 + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(\mathbf{w}_{ji}^{l})^2 \tag{2.11}$$

where, b is the bias term, $J(\mathbf{x}, b)$ is the generic term for our mean sum squared error cost function, and lambda is the weight decay parameter. The second term of Equation 2.11 helps prevent over fitting.

When the model is first compiled the weights are initialized to have random values and the bias term is added to each layer (except the input and output layer) such that it is both random and near 0. During the forward pass (when the input is moving from one layer to the next) the input is multiplied by the weights, a prediction is made (the output of our final layer), before finally an error is computed between the true value and the prediction which can be defined as $\delta_i^{n_l}$.

An update rule for the ith in the lth layer weight and bias term can be defined as:

$$\mathbf{w}_{ij}^{l} = \mathbf{w}_{ij}^{l} - \alpha \frac{\partial}{\partial \mathbf{w}_{ij}^{l}} J(\mathbf{w}_{ij}^{l}, b) \tag{2.12}$$

$$b_i^l = b_i^l - \alpha \frac{\partial}{\partial b_i^l} \tag{2.13}$$

Putting this all together, the algorithm to implement backpropagation is [44]:

1. Perform the forward pass, computing the activation functions l_1,l_2, l_3 until the last layer.
2. $\delta_i^{n_l} = \frac{\partial}{\partial z_i^{n_l}} \frac{1}{2} ||\mathbf{y}_i - \hat{f}(x_i)||^2$
3. For $l = n_l - 1, n_l - 2, n_l - 3, ..., 2$
   For each neural node i in layer l, set

   $\delta_i^{n_l} = (\sum_{j=1}^{s_l+1} \mathbf{w}_{ij}^{l} \delta_j^{l+1}) f'(z_i^l)$
4. Compute partial derivatives:

   $\frac{\partial}{\partial \mathbf{w}_{ij}^{l}} J(\mathbf{w}, b; \mathbf{x}_i, \mathbf{y}_i) = A_j^l \delta_i^{l+1}$

   $\frac{\partial}{\partial b_i^l} J(\mathbf{w}, b; \mathbf{x}_i, \mathbf{y}_i) = \delta_i^{l+1}$

where $z_i^{n_l}$ is the weighted sum of the inputs for the ith neural node and lth layer. For example for the output of the first layer (not the input is):

$$z_i^2 = \sum_{j=1}^{N} \mathbf{w}_{ij}^{1} \mathbf{x}_j + b_i^1$$

2.5.4 Convolutional Neural Networks

Convolutional neural networks are a type of artificial neural network that are designed to handle grid-like structured input, such as time-series data and images. Unlike feed-forward networks, convolutional neural networks preserve the temporal and spatial dependencies in its input. In the broader context of Machine Learning, Convolutional Neural Networks (CNN) are one of the most popular learning architectures today given their effectiveness in image and video recognition, natural language processing, image analysis and classification, just to name a few.

Unlike standard densely connected neural networks (see Section 2.5.1) which operate on matrix or tensor multiplication, CNN operate on convolutions. A convolutional neural network is composed of one or more convolutional layers, pooling, and linear output layers followed by a fully connected neural network.

One of the great advantages, aside from preserving spatial and temporal dependencies, of CNN is their sparse interactions, parameter sharing, and equivariant representations in.[16] CNNs can learn spatial hierarchy of patterns, allowing them to learn increasingly more complex, visual concepts.[4]

The same principles apply when training a convolutional neural network compared to a feed-forward network. Perhaps the most notable difference is the implementation of back propagation through convolutions.

Convolutional neural networks using two-dimensional image data as input are described in detail here because it is easier apply the theory from the 2D case to the 1D case than it is vice versa. The subtle differences between the input set-up for training using 1D time-series data as opposed to 2D images is discussed at the end of this chapter.

Architecture of a Convolutional Neural Network

Just like a standard, feed-forward neural networks, convolutional neural networks (CNNs) are composed of three main components: an input layer, hidden layers, and an output layer. However, the 'hidden layers' of a CNN are instead replaced with one more convolutional layers and subsampling layers followed by fully connected layers.

An image can be interpreted as a two-dimensional grid of pixels with three channels, each referring to the intensity of the red, green, and blue color scheme (RGB). The input is thus an l x b x d image where l is the height, b is the breadth, and d is the depth. As an example, propose we have a 32 x 32 image with three channels for the colors. Thus, $l = 32$, $b = 32$, and $d = 3$ [4]. In the convolutional layer, the element that will carry out the convolutions is known as the kernel or filter. The kernel will have n x n x q dimension where n is smaller than the dimensions of the image and q is allowed to vary [44]. The kernels are the parameters that are learned during training. For our example, let's define a 5 x 5 x 1 kernel.

During convolution, the kernel will glide across the entire l x b image and performs a dot product, the number of alignments between the kernel and the image will determine the output dimension of the next layer called the feature map [4]. Feature maps will have dimension of l–n + 1. Giving feature maps of 28 x 28 in our example. The step size, or stride, of the kernels are specified prior to initiating training. In our example we'll leave this to 1, i.e. the kernel is moving one pixel at a time.

The number of features maps made by a convolutional layer determines the number of parameters that are fine-tuned during training. In other words, the number of feature maps/filters used determines the complexity of our convolutional neural network [4]. The more filters, the more complex.

It is common for the number of feature maps to increase as the input moves to deeper layers. The idea being that the first few layers can be considered as 'general feature extractors' that detect low-level features such as edges, color, gradient orientation, etc [1]. Moving toward the deeper layers, the convolutional neural network will learn larger patterns made from the previous layers, creating a *hierarchical* representation of the original input [10].

Subsampling with a mean or max pooling is typically performed on the feature maps. The *pooling layer* reduces the size of the feature map. Reducing the dimension of the convolved output is of importance because it is a) useful for extracting the dominate features and b) it reduces the computational resources during training.[1]

In order to give the CNN more flexibility, one generally uses nonlinear activation functions (see Section 2.5.2). As a starting point for training convolutional neural networks it is common to start off with ReLU activation functions. The ReLU layer does not change the dimensions of the layer because it is a simple one-to-one mapping of activation values [4].

To compute the convolution of the qth feature map in the kth to the $(k+1)$th row we use:

$$h_{ijq}^{(k+1)} = \sum_{r=1}^{l} \sum_{s=1}^{n} \sum_{d=1}^{n} w_{rsd}^{(q,d)} h_{i+r-1,j+s-1,d}^{k}$$

$$\forall i \in \{1, .., l-n+1\}, \, \forall j \in \{1, .., l-n+1\}, \, \forall q \in \{1, .., l-d_{d+1}\}$$

Where i, j, d represent the height, width, and depth of the feature map. The kth layer is represented by a 3-dimensional tensor, $h_{ijq}^{(k)}$. Parameters(i.e. weights) for the qth feature map in the kth row are defined by a 3-dimensional tensor $w_{rsd}^{(q,d)}$

Note that the above formulation is not a convolution, but rather, a *correlation.* The commutative properties of convolution allow us to flip the kernel and perform *correlation.* Most out-of-the-box machine learning APIs perform correlation instead, but the name convolution stuck. Hence, they are called convolutional neural networks and not correlation neural networks.[16]

Practical Example: 1D Case

Both from an application and practical point of view, 1D convolutional neural networks closely resemble their 2D counterparts. Instead of images, the input to a 1D CNN is time-series data or text. The main difference comes in how we handle the input and how the convolution slides across our data.

Given that the theory behind 2D Convolutional Neural Networks has been covered in detail above, we'll illustrate the 1D case via an example. We'll demonstrate in detail how the input of a 1D convolution is treated during training by using our s-phase picker (see Section 5.7) as this might serve most useful to readers.

Input for the 1D Case

Unlike a 2D convolutional neural network which takes in as input (*height*, *width*, *channels*) the input of a 1D convolutional neural network takes the form of:

(number of points, channels)

Taking as example the synthetic data generated for the s-phase picker (see Section 5.7), this meant our CNN had an input shape of:

(1800, 4)

Corresponding to the fact that our data set consisted of 60 second records, with a sampling rate of 20Hz which left us with 1800npts. We used four-component data; therefore the number of "channels" in our case is 4.

For our simple s-phase picker we used the following model architecture (see Figure 2.1):

Table 2.1: Summary of Convolutional Neural Network

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 1800, 8)	200
max pooling1d	(None, 900, 8)	0
conv1d (Conv1D)	(None, 900, 16)	784
max pooling1d	(None, 450, 16)	0
flatten (Flatten)	(None, 7200)	0
dense (Dense)	(None, 8)	57608
batch normalization	(None, 8)	32
dense (Dense)	(None, 1)	9
Total params: 58,633		
Trainable params: 58,617		
Non-trainable params: 16		

Breaking this apart, we have:
Layer 0: Our input layer with dimension (1800, 4)
Layer 1: Our first convolutional layer. (With Keras one can omit specifying the Input layer since it is only a placeholder). The tensor has dimension (1800, 8); 1800 referring to the number of sampling points and a kernel of length 8.
Layer 2: Maxpooling of size 2. Reducing the dimension of input from 1800 to 900. Leaving us with a (900, 8) matrix.
Layer 3: Another 1D Convolutional layer with filter size of 16. This filter is learning higher order features as opposed to edges in the previous convolutional layer. Our output is a matrix of shape (900, 16)
Layer 4: Another maxpooling layer of size 2. The output matrix is now half: (450, 16).

2.5.5 Autoencoders

Introduction

Autoencoders are a type of Deep Neural Network architecture which can be used for unsupervised learning. They can be used for: dimensionality reduction, principle component analysis, and matrix factorization [4]. The beauty behind these neural networks comes from a hidden layer known as the coding. The coding layer has a lower dimension than the input. Which means that, if the neural network is trained to reconstruct its input, this coding layer must contain the information needed to represent the input but at a lower dimension. This makes them powerful feature extractors.

The architecture of autoencoders can vary, but the traditional set-up defines three major components: the encoder, coding, and decoder. Typically, the encoder and the decoder are symmetric; i.e. for an M-layer autoencoder, the number of units in the kth layer is the same as the (M - k + 1)th layer. [4] Since autoencoders are feed-forward networks, one can use loss functions and output unit types, as traditional feed-forward networks [16].

To prevent the autoencoder from trivially learning its input, i.e. the identity function, it is common practice to corrupt the input; therefore forcing the neural network to learn useful features. Such autoencoders are known as *denoising autoencoders.*

Autoencoder Architecture

Machine learning algorithms attempt to capture aspects of the unknown distributions made by the observed data set. This is the driving force behind machine learning. One method to capture the essence of this unknown distribution is with unsupervised feature learning algorithms such as autoencoders [6].

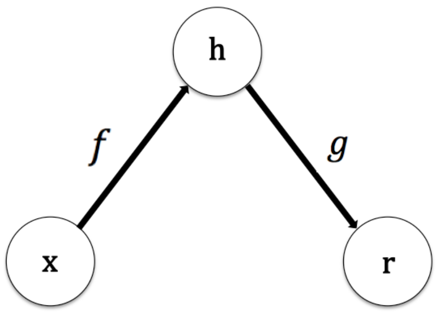

Figure 2.1: Autoencoder mapping. f and g represent the encoder and decoder mapping functions, respectively.

The autoencoder can be considered as being composed of two major pieces: an encoder and a decoder. Given an input $\mathbf{x}$, we define the encoder f as the function which maps the input into an internal representation $f(\mathbf{x})$ known as the coding. The coding, $f(\mathbf{x})$ is then mapped back to the input space with the decoding function, g. (See Figure 2.1). The composition of f and g is called the reconstruction function r , giving us $r(\mathbf{x}) = g(f(\mathbf{x}))$.

The approach of training an autoencoder is the same as the case of training a standard feed-forward network: the loss function is minimized using optimization techniques using some form of weight updating via back-propagation. The loss function can be defined as:

$$L(\mathbf{x}, g(f(\mathbf{x}))) \tag{2.14}$$

Where the loss function, L, penalizes $g(f(\mathbf{x}))$ for being dissimilar to, $\mathbf{x}$ [16].

To prevent the autoencoder from simply memorizing, or copying its input, one can corrupt the input by adding noise to it. This corruption process can be written as $C(\tilde{\mathbf{x}}|\mathbf{x})$. Doing so makes the autoencoder learn a *reconstruction distribution*, $p_{reconstruct}$ $(\mathbf{x}|\tilde{\mathbf{x}})$ from the training pairs. The algorithm to find the reconstruction distribution is thus [16]:

1. Sample a training example $\mathbf{x}_i$ from the training data
2. Sample the corrupted version $\tilde{\mathbf{x}_i}$ from $C(\tilde{\mathbf{x}}|\mathbf{x})$
3. Use the instances from steps 1 and 2 to estimate the autoencoder reconstruct distribution $p_{reconstruct}$ $(\mathbf{x}|\tilde{\mathbf{x}}) = p_{decoder}$ $(\mathbf{x}|\mathbf{h})$ where h is the output of the encoder $f(\tilde{\mathbf{x}})$ and $p_{decoder}$ defined by the decoder $g(\mathbf{h})$.

Denoising Autoencoder

In order to reduce the corruption process, the neural network learns/captures the probabilistic distribution of the input. We call these type of autoencoders, where the input has been corrupted with noise added to it, *Denoising Autoencoders.* [6] Denoising autoencoders are a type of regularized autoencoders. Regularized autoencoders use loss functions which increase the likelihood the neural network will not only trivially copy its input. In other words, regularization forces the autoencoder to be less sensitive to input, but minimizing the reconstruction error forces it to remain sensitive to similar inputs. *Sparse Autoencoders* are another type of regularized auotencoder, but they will not be discussed here.

Denoising autoencoders are trained minimizing the denoising loss function [6]:

$$\mathcal{L}_{DAE} = E[L(\mathbf{x}, r(N(\mathbf{x}))] \tag{2.15}$$

which describes the expectation of the training distribution and corruption noise source and $N(x)$ is a stochastic corruption of $\mathbf{x}$ (e.g. Gaussian noise corruption).

The followig theorom is stated without proof [6]

Theorem 2 *Let p be the probability density function of the data. If we train a denoising autoenocder (DAE) using the quadratic loss and corruption noise $N(x) = x + \epsilon$ with*

$$\epsilon \sim \mathcal{N}(0, \sigma^2 I)$$

then the optimal reconstruction function $r^*(x)$ will be given by

$$r^*(x) = \frac{E_\epsilon[p(x-\epsilon)(x-\epsilon)]}{E_\epsilon p(x-\epsilon)} \tag{2.16}$$

for values of x where p(x) $\neq=0$.
Moreover, if we consider how the optimal reconstruction function r^*_σ behaves asymptotically as $\sigma \to 0$, we get

$$r^*_\sigma = x + \sigma^2 \frac{\partial \log p(x)}{\partial x} + o(\sigma^2) as \sigma \to 0 \tag{2.17}$$

The takeaway is that learning the gradient of the $\log p_{data}$ is one method to learn the actual distribution of the input, p_{data} [16].

Chapter 3

Fundamentals of Seismology

3.1 Equation of Motion

The equation of motion of an infinitesimal cubic element in a medium undergoing internal motions can be defined as:

$$\rho \frac{\partial^2 u_i}{\partial t^2} = f_i + \frac{\partial \sigma_{ij}}{\partial x_j} \tag{3.1}$$

where σ_{ij} is the symmetric stress tensor

$$\sigma_{ij} = \begin{vmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_{22} & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_{33} \end{vmatrix} \tag{3.2}$$

u_i is the displacement field, ρ is density, and $\mathbf{f}$ is the body force per unit volume acting on the cube. If we ignore gravity or body forces from Eq. 3.1 we have the homogeneous equation of motion

$$\rho \frac{\partial^2 u_i}{\partial t^2} = \frac{\partial \sigma_{ij}}{\partial x_j} \tag{3.3}$$

which in Einstein notation can be rewritten as

$$\rho \ddot{u}_i = \sigma_{ij,j} \tag{3.4}$$

The constitutive laws provide a relationship between stress and strain. Using the elastic constitutive equation, we have

$$\sigma_{ij} = C_{ijkl} \epsilon_{kl} \tag{3.5}$$

where

$$\epsilon_{ij} = \frac{1}{2}(\delta_j u_i + \delta_i u_j) \tag{3.6}$$

is the infinitesimal strain tensor and C_{ijkl} is the third-order elastic moduli tensor. The elastic moduli contains 81 terms relating the nine elements of the stress tensor and the nine elements of the strain tensor. The symmetry of the stress and strain tensor reduces the 81 terms down to 36. A final symmetry relationship

$$C_{ijkl} = C_{klij} \tag{3.7}$$

reduces the 36 terms down to 21. The stress-strain behavior in the above case depends on direction. In the case of an isotropic elastic substance we are left with only two independent elastic moduli, λ and μ, which are the Lame constants. Inserting the Lame parameters into Eq. 3.5 gives

$$\sigma_{ij} = (\lambda\delta_{ij}\delta kl + \mu(\delta_{ik}\delta_{kl} + \delta_{il}\delta_{jk}))\epsilon_{kl} \tag{3.8}$$

δ_{ij} is the Kronecker delta. Taking the equation of motion Eq. 3.1, the infinitesimal strain tensor Eq. 3.6 and Eq. 3.8 and simplfying terms gives

$$\rho\ddot{\mathbf{u}} = \mathbf{f} + (\lambda + 2\mu)\nabla(\nabla \cdot \mathbf{u}) - \mu\nabla \times (\nabla \times \mathbf{u}) \tag{3.9}$$

which leaves us with partial differential equations for displacements in a homogeneous isotropic medium.

Helmholtz's theorem states that any sufficiently smooth and rapidly decaying vector field can be considered as the sum of a curl-free and divergence free vector field. Meaning that the vector field $\mathbf{u}$ can be represented in terms of a vector potential Ψ and scalar potential Φ

$$\mathbf{u} = \nabla\Phi + \nabla \times \Psi \tag{3.10}$$

if

$$\nabla \times \Phi = 0 \text{ and } \nabla \cdot \Psi = 0$$

$\nabla\Phi$ and $\nabla \times \Psi$ are the P-wave and S-wave components of $\mathbf{u}$, respectively. Since we have a curl free scalar potential and a divergence free vector field, we have

$$\nabla \times \mathbf{u} = \nabla \times \nabla \times \Psi \tag{3.11}$$

$$\nabla \cdot \mathbf{u} = \nabla^2\Phi \tag{3.12}$$

In other words, we can separate the P and S-wave components in an isotropic and homogeneous medium by measuring the divergence and curl of an elastic wavefield.

One can directly measure the curl of the wavefield with a rotational sensor or indirectly using the spatial seismic wavefield gradients [40]:

$$\boldsymbol{\omega} = \frac{1}{2}\nabla \times \mathbf{u} = \frac{1}{2}\begin{pmatrix}\partial_y u_z - \partial_z u_y \\ \partial_z u_x - \partial_x u_z \\ \partial_x u_y - \partial_y u_x\end{pmatrix} \tag{3.13}$$

where $\boldsymbol{\omega} = [\omega_x, \omega_y, \omega_z]^T$. We can also write Eq. 3.13 taking its time derivative:

$$\dot{\boldsymbol{\omega}} = \frac{1}{2}\nabla \times \mathbf{v} = \frac{1}{2}\begin{pmatrix}\partial_y v_z - \partial_z v_y \\ \partial_z v_x - \partial_x v_z \\ \partial_y v_y - \partial_y v_x\end{pmatrix} \tag{3.14}$$

where $\dot{\boldsymbol{\omega}}$ is the rotation rate and v is the velocity.

Free Surface Effects on Rotational Motion

The vertical components of the stress tensor vanish at the free surface, thus the strain tensor components $\epsilon_{iz} = 0$ for $i = x, y$ in Eq. 3.6. Which means that, at the free surface, the spatial gradient (Eq. 3.13) can be rewritten as[11]:

$$\boldsymbol{\omega} = \frac{1}{2}\nabla \times \mathbf{u} = \frac{1}{2}\begin{pmatrix} \partial_y u_z \\ -\partial_x u_z \\ \frac{1}{2}(\partial_x u_y - \partial_y u_x) \end{pmatrix} \tag{3.15}$$

Meaning that, at the Earth's free surface, horizontal rotation corresponds to tilt.

As a consequence of Eq.3.15 P-waves have no rotational component inside a homogeneous medium. However, do to the free surface boundary condition, there is a contribution to horizontal rotation from P-to-SV converted waves [11].

3.2 Comparing Rotations and Translations

Considering the case of a transversely polarized plane traveling in the x direction with a wave with displacement of:

$$\mathbf{u} = (0, u_y(t - \frac{x}{c}), 0) \tag{3.16}$$

where c is the horizontal velocity. The rotation is thus:

$$\boldsymbol{\omega} = \frac{1}{2}\nabla \times \mathbf{u} = (0, 0, -\dot{u}_y(t - \frac{x}{c})/(2c), 0) \tag{3.17}$$

leaving us with:

$$\omega_z = -\frac{\dot{u}_y}{2c} \tag{3.18}$$

Assuming plane wave propagation Eq. 3.18 has some profound results. Namely, we expect the vertical rotation and transverse velocity seismograms to: 1) have identical waveforms and 2) the amplitudes are proportional to twice the phases velocity. The same assumptions is true for the vertical rotation rate and transverse acceleration seismograms[22].

The vertical acceleration and radial rotation rate (assuming a plane wave propagation) in the x direction for P-, SV-, and Rayleigh waves, are related by [11] [6]:

$$c = -\frac{\ddot{u}_z}{\dot{\omega}_y} \tag{3.19}$$

As a result of the similarity of the transverse velocity and vertical rotation one can investigate the source direction of SH-type motions. [20] [30] [17] found that rotating the transverse component of a translational seismic record, until the resemblance with the vertical rotational measurement is at a maximum, corresponds to the back azimuth of the event.

In addition, Eq. 3.18 signifies one can estimate phase velocities using collocated point measurements of rotational and translational motions which has been done by [21] [20] [32] [11].

3.3 Array-Derived Rotations

In a linear elastic medium when can describe, assuming infinitesimal deformation, one can describe the displacement of a point $\mathbf{x}$ in relation to that of a neighboring point $\mathbf{x} + \delta\mathbf{x}$ as [5]:

$$\mathbf{u}(\mathbf{x}) + \boldsymbol{\epsilon}\delta\mathbf{x} + \boldsymbol{\omega} \times \delta\mathbf{x} \tag{3.20}$$

Eq.3.20 implies that, given then complete displacement field, one can theoretically compute rotational ground motions[11]. However, additional knowledge and assumptions of the Earth's structure must be imposed in order to approximate all components of the rotation vector if the displacement field is only measured at Earth's surface[22]. One approach is to use the surface arrays of translational seismometers to estimate the horizontal and vertical components of the rotation vector[**der**] [43] [22] [11]. This method can be applied when the P- and S-wave velocities beneath the array are known and deformation is linear over the array area [22].

When considering computing array-derived rotations one should also consider that translational sensors are contaminated by rotations. In particular, the horizontal components of strong motion and long-period observations are sensitive to horizontal components of the rotation vector. In theory one should be able to correct this effect by measuring the seismometer tilt, but that has yet to be fully realized [11].

Chapter 4

Denoising Seismic Signals with Deep Neural Networks

4.1 Introduction

Seismic signals provide the observations which drive discoveries of the Earth, allows for imaging the Earth's interior, drives oil exploration, and can be used in conjunction with assumptions of the speed of traveling waves to detect Earthquakes. It is therefore pivotal that as much of the signal generated from the source is preserved.

In reality, seismic data is contaminated with various sources of noise including, but not limited to: instrumental, tidal, pressure variations, and traffic. It is therefore common practice to apply filters. However, knowing which filter to apply is not intuitive. Additionally, applying a filter might remove relevant information from your signal if the noise is in the same frequency band as your signal.

Over the years there have been efforts to find an effective means of removing noise (denoising) seismic data. However, these methods usually require specifying some sort of threshold which, if selected incorrectly, can lead to false signals or loss in effective signals [25] [49].

Given the ongoing efforts of improving the signal-to-noise ratio of seismic data, our goal for this Thesis is to find a new approach to denoise seismic records with a deep learning approach.

Our approach of using deep neural networks for denoising was motivated by the new gains made by in the audio signal processing domain. Typical signal processing of an audio file include: removing ambient noise, taking into account unequal distribution microphone recording, enhancing certain frequency bands and suppressing others, and so on. In recent years deep neural networks have been introduced to combat some of the above-mentioned processing. Most famously are the applications of deep neural networks for speech recognition enhancement which are the driving force behind voice-recognition applications such as Amazon's Alexa and Apple's Siri.

This paper is structured as follows: first, we will go over current approaches of denoising seismic data. In the same section, we will ignite the motivation of why the use of Deep Neural Networks for this task is attractive. Next, we will go over our machine learning workflow which includes downloading, pre-processing, and defining a machine learning algorithm architecture. We will then discuss the results from training, testing, and transfer learning as they presented. We will end by going over the next steps in store for our neural network.

4.2 Background

4.2.1 Denoising Neural Networks

The driving force behind today's automated speech recognition (ASR) are deep neural networks. They operate by first breaking apart and analyzing the components (phonemes) of the incoming speech waveform, digitizing it, analyzing the meaning before feeding it to a neural network. One of the challenges behind speech recognition is the presence of background noise [25]. As a result, one of the routes currently being investigated for improving the signal-to-noise ratio of speech are deep neural networks. In particular the denoising autoencoder, which is a type of deep neural network built with convolutional bases (see Section 2.5.4), is demonstrating its ability to reconstruct a clean speech signal given its own noisy input [7] [45] [38] [25].

The challenge of improving the signal-to-noise ratio facing ASR parallels the challenge the Seismological community faces with seismic recordings. Instead of removing noise for the purpose of ASR, seismologists desire to remove ambient noise (or microseisms) to enhance an earthquake signal (for example). Given the promising results of denoising audio files, we were motivated to use denoising autoencoders.

4.2.2 Current Approaches in Seismology

There are groups within the geophysical community incorporating various types of machine learning architectures for improving the signal-to-noise ratio. [51] have approached this problem using Convolutional Neural Networks (see Section 2.5.4) which are a type of neural network designed to work with grid-like structured data such as images and time-series data (in the 1D case). Their approach involves first transforming their data into the frequency domain then using both the real and imaginary part as input to their CNN [51]. [50] developed a denoising method based on a parametric dictionary learning scheme. Dictionary learning is a branch of signal processing and machine learning which exploits underlying sparse structures of the input training data. With these sparse representations, they were able to capture the characterizations of seismic data which could then be used for denoising. [35] used a combination of a U-net (which are formed using CNN) and incorporating a pre-trained neural network known as ResNet34. In doing so they took advantage of a feature extracting capabilities of CNN and the benefits of using pre-trained neural networks. Using a pre-trained neural network for training is a machine learning technique known as transfer learning, and it is an approach we attempt here. At the moment geophysicists are still exploring different techniques in denoising using machine learning algorithms. It remains to be seen which method is most effective in denoising when considering the computation time, pre-processing, and memory requirements. As mentioned in the previous section our architecture of choice are denoising autoencoders.

4.3 Methodology

4.3.1 Software

Open-source software and free data available to the public were used for the execution of this Thesis. The list of open-source machine learning libraries for use is constantly expanding, as is the 'crowd favorite'. We used the open-source Keras Python Deep Learning library API [26]. It currently has one of the largest community support, fastest-growing, and most scholarly cited (among other criteria) [47]. Keras is user-friendly, modular, extensible, and there are a plethora of tutorials online to learn from. The heavy-lifting tensor multiplication and convolutional operations are done with TensorFlow. TensorFlow is another open-source software that performs its computations with an optimized C++ backend.

For querying, downloading, and pre-processing our seismic data set we also used Python. In particular, we heavily relied on NumPy, Pandas, and ObsPy Python libraries.

4.3.2 Data Set

Our original intent was to train our model on time-series seismic data recorded by the STS-2 broadband translational sensor in Wettzell, Germany (WET). However, the cataloged and archived events for WET are predominately teleseismic and have a magnitude of 5.0 and above. This limited the number of events we could use for training. At the time of our query, there were less than 1000 events we could use for our data set. With this limitation, we turned our focus to train our denoiser on local events with a minimum magnitude threshold of 3.0.

This left us with two options: 1) use data from other sensors which are part of the German Regional Seismic Network (GRSN) or 2) use seismic data recorded in California. We chose the latter because the seismic network in California is robust, high quality, and there are numerous local events to choose from given the seismicity of the state. In choosing to use data from California we committed ourselves to apply a machine learning technique known as transfer learning. In essence, we use a pre-trained neural network on a similar, but different, data set. We discuss how this was done in more detail in Section 4.6.

Our data set, therefore, consisted of STS-2 broadband data from Northern California. Given the rich history of earthquake monitoring and phase-arrival picking, we were able to download our data set along with picked arrival times from the Northern California Earthquake Data Center (NCEDC) for BHE, BHN, and BHZ channel data. We chose four stations with high signal to noise ratio: Columbia College (CMB), Hull Mountain (HUMO), McLaughlin Natural Reserve (MNRC), and Bear Valley Ranch (PKD). We chose events within a 300km distance of each station. Events had a minimum magnitude of 3.0. With this criterion we found approximately 4,300 unique station recordings within a 5-year time span.

4.3.3 Preprocessing

Each component of our records (Z, N, E) were treated as a 'unique' instance and thus artificially expanded our training set by a factor of three, giving us a total of 13,000 training instances to learn from (and thus satisfying our minimum training data set desires). Waveforms were detrended, high-passed filtered with a maximum frequency of 0.5Hz, and normalized. We then decimated our traces from 40Hz to 20Hz sampling rate; allowing us to reduce the input length size from 3600npts to 1800npts. This was done in order to reduce the number of parameters the machine learning algorithm would have to learn. Waveforms were then trimmed to 90-second records, starting 30 seconds before the picked P-arrival.

Our denoising autoencoder needs two inputs: an input $\mathbf{x}$ and a corrupted version $\tilde{\mathbf{x}}$. (see 2.5.5) To create our corrupted version $\tilde{\mathbf{x}}_i$ for each input $\mathbf{x}_i$ in our data set, we downloaded recorded seismic data between catalogued earthquake events. This is our noise and corruption process. We took this downloaded noise and added it to our pre-processed data. This left us with two versions: a clean version and a noisy/corrupted version. The noisy waveform and its associated original clean version are the input and label to our denoiser autoencoder, respectively.

4.3.4 Model

We used a denoising autoencoder (DAE) for our model architecture. Autoencoders are a type of Convolutional Neural Network which are used for unsupervised learning (see Section 2.5.5). If trained reliably, an autoencoder will take an input, convert it to a lower dimension, reconstruct the dimension, and then output something that looks similar to its original input. Assuming the model is trained to recreate the input, the hidden layer ‘bottleneck’ in the neural network forces the autoencoder to contain information needed to represent the input but at a lower dimension. This makes autoencoder power feature extractors.

To prevent the autoencoder from trivially learning its input, we corrupt the input; thereby forcing the autoencoder to learn useful features (Section 2.5.5). In order to reduce the corruption process, the neural network learns/captures the probabilistic distribution of the input. We call this type of autoencoder, where the input has been corrupted with noise added to it, denoising autoencoders.

The rule of thumb when designing a neural network is to start shallow and check how the algorithm performs. If there are indications that the model is not learning (e.g. the learning/loss curve does not improve as a function of epoch), then it is at this point one adds more layers.

With this in mind, we started off with a simple denoising autoencoder. The encoder consists of two convolutional and maxpooling layers. For each maxpooling layer applied, the dimension of the output of the previous layer (i.e. the input of the maxpooling layer) is reduced (see Figure 4.1). To bring back our reduced dimension input back to its original size, we mirrored our encoder but this time using two convolutional and upsampling layers.

There does not exist a predefined set of rules for knowing which hyperparameters to use for a given model. It is therefore common practice to read current literature about similar machine learning projects to get a ’ballpark’ idea of which to use.

The hyperparameters used for training were thus found via trial-and-error. We found that the best set-up giving our training data set consisted of a mean squared error loss/cost function (see Section 2.4.1) to assess model performance during training and validation. We used an Adam optimizer algorithm, which is an extension of stochastic gradient descent, to iteratively update the network weights during training ([27]). Some of the attractive features of using the Adam optimizer include,

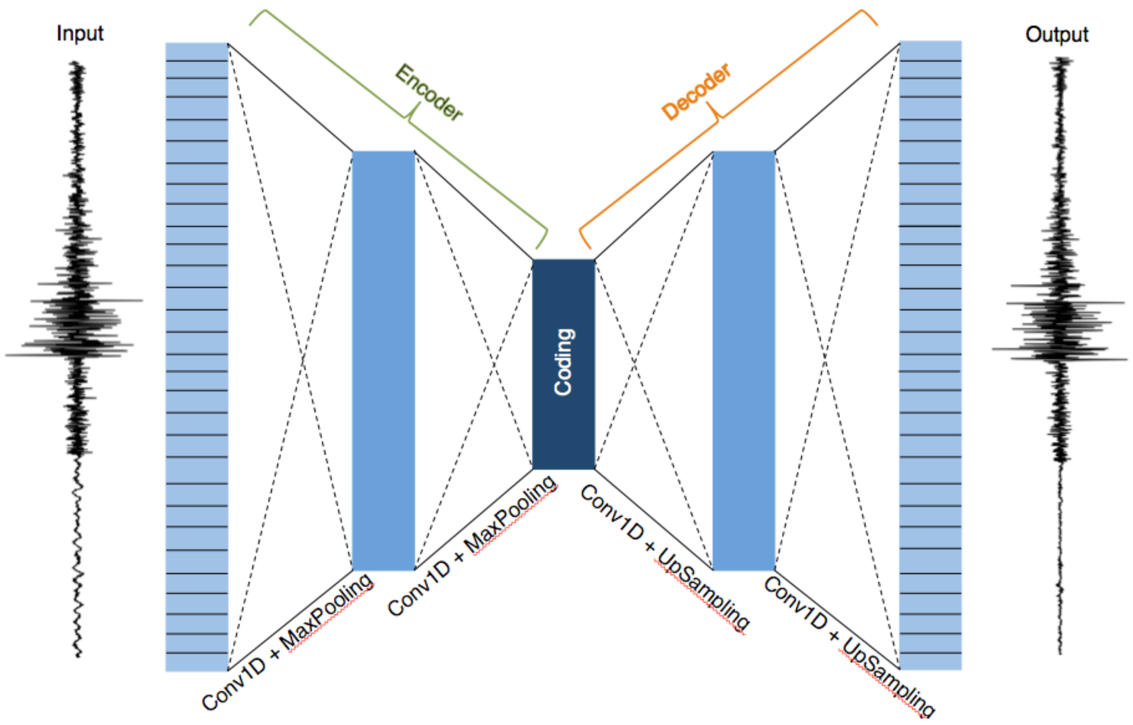

Figure 4.1: Denoising autoencoder model architecture used for training. The left portion (outlined in green on the top) depicts the encoder and the right portion (outlined in orange on top) is the decoder. If trained to reconstruct its input reliably, the coding (center dark blue block where the bottleneck occurs) is able to represent the input but at a lower dimension.

but are not limited to it being: computationally efficient, has little memory requirements, and it can handle well problems with large data sets and/or parameters [27]. An outline of our set up can be seen in Figure 4.1.

Table 4.1: Model architecture of the denoising autoencoder. Conv = 1D convolution, Max = maxpooling, UpS = upsampling.

Layer	1	2	3	4	5	6	7	8	9
Stage	Conv	Max	Conv	Max	Conv	UpS	Conv	Ups	Conv
# Channel	8	-	16	-	16	-	8	-	1

4.4 Training

To train our regressor (Section 2.3.2), we gave as input our 'clean' version, $\mathbf{x}$, and its associated corrupted/noisy version, $\tilde{\mathbf{x}}$ (see Section 2.5.5). We split our data set into two pieces: a training and validation set. 80% of the data we set aside for training and the remaining 20% was set aside for validation. (Section 2.2.1) Thus of the 13,000 pre-processed training examples, 2,600 waveforms were set aside for validation and the remaining 10,400 were used for training. We used mini-batch gradient descent with batch sizes of 200. Unlike a typical regressor or classifier (see

Section 2.3.2) the output of our unsupervised will be continuous time-series arrays, i.e. our denoised waveforms.

At the time of the execution, we had access to multi-core processors. Thus, the training time was approximately 2.8 hours. We later obtained NVIDIA GPU access from LRZ but we did not run our model to see how the performance time might have improved.

4.4.1 Results from Training

A quick diagnostic tool to check if the machine learning algorithm is learning is to refer to its learning (sometimes called loss) curves. Learning curves describe the experience on the x-axis and learning (or improvement) on the y-axis (see Figure 4.2). A single epoch describes one full cycle of making predictions on the training set. As a function of epoch we expect the loss (sometimes called 'cost') to decrease for both the data it is directly learning from (i.e. the training set) and the data set used to validate it learning (i.e. the validation set). From these plots it is possible to identify warning signs of overfitting, underfitting, underrepresented training set, and underrepresented validation sets (see Section 2.2).

In Figure 4.2 we note that the loss curve decreases with epoch for both the training and validation set. This suggests our algorithm is improving its ability to reconstruct the waveforms corresponding to its corrupted input, $\tilde{\mathbf{x}}$.

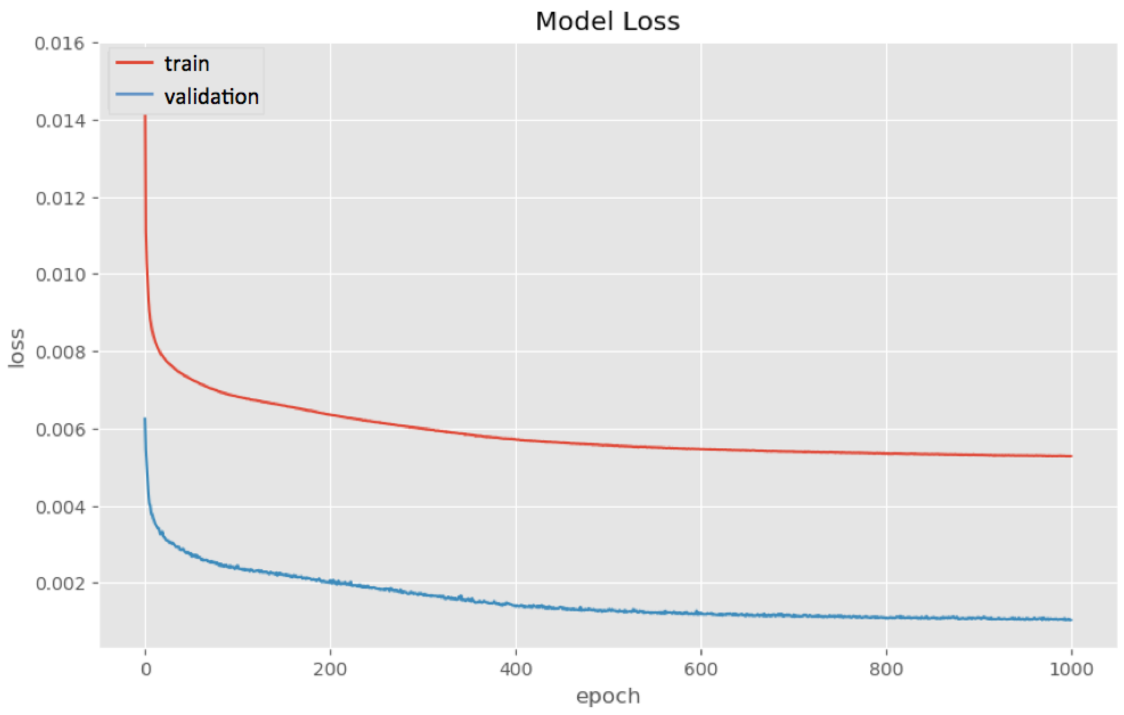

Figure 4.2: Loss curve computed during training from the denoising autoencoder. Training was terminated when the model was no longer to improve its predictions (i.e. loss value no longer decreases) around epoch 1,000.

The number of epochs chosen was done so as to minimize the likelihood of over-fitting (see Section 2.4.2) which can happen if the neural network is allowed to run for too long. There are a couple of visual indicators to determine if one is running into the risk of overfitting 2.4.2. In our case, the plateau of the curves at around 1,000 epochs was hint that the algorithm was no longer improving its predictions, and thus was time for us to terminate the algorithm.

In the future, to automate the process of terminating learning based on the performance of the learning curves, we will implement Early Stopping. Early Stopping is a method that stops training once it is no longer improving its performance (e.g. the loss curve no longer decreases) on the validation set. Doing so will help us reach the model's peak performance by helping us automatically avoid underfitting and overfitting (see Section 2.4.2).

Note that in our case our validation error is smaller than our training error. One would expect that the algorithm performs better at making predictions for the training set since it is learning directly from it. This could be the result of our data set not being shuffled. Currently there is an unequal distribution of CMB, PKD, HUMO, and MNRC in the training and validation set. More specifically, the neural network is being fed more stations from HUMO and MNRC, two stations that have a lower SNR than CMB and PKD. Meaning the algorithm is systematically training on data which has a less easily recoverable signal. If we then give it a data set with better SNR, we can expect that the performance of the neural network to be better. This is the current understanding of why the validation set loss is better than that of the training.

To investigate the output of our neural network we started off by creating time-frequency goodness-of-fit(GOF) comparisons as defined by [29]. The plots compare the goodness-of-fit between two waveforms based on the time-frequency representation of the seismograms obtained as the continuous wavelet transform with the Analyzing Morlet transform. More specifically, the criteria established by Kristekova includes quantitative analysis of: time-frequency envelope (TFEG) and phase (TFPG) goodness-of-fit, time-dependent envelope (TEG) and phase (TPG) goodness-of-fit, frequency-dependent envelope (FEG) and phase (FPG) goodness-to-fit, in addition to single-value and phase and envelope goodness-of-fit (PG and EG, respectively) [29].

A select number of waveforms from our training set are seen in Figure 4.3 and 4.4. The black waveforms are the original pre-processed waveforms and the red waveforms are the denoised waveform created by our denoiser autoencoder. Subplots above the trace comparisons break apart the time-frequency envelope goodness-to-fit as a function of frequency (labeled FEG, in the upper left of plots) and time (labeled TEG, directly above the waveform). Similarly, subplots below the traces describe the time-frequency phase goodness-to-fit as a function of frequency (labeled FPG, in the lower left of plots) and time (labeled TPG, directly below the waveform).

We hope to see that the output of our neural network (here shown in red) is a waveform that closely resembles the original clean (black waveforms). Said another way, the goodness of fit between the black and red traces should be high (6.5 and above for the single-valued one the GOF scale).

From Figure 4.4 we can see that the goodness-to-fit is exceptional; with single-valued EG and PG values above 9.5. The top trace has some deterioration in the TFEG starting around 50 seconds, with a dominant frequency of approximately 1Hz. The GOF values decrease to approximately 6.5. While the TPFG is nearly unchanged throughout the record.

The bottom trace TFEG sees a decrease in goodness-of-fit from approximately

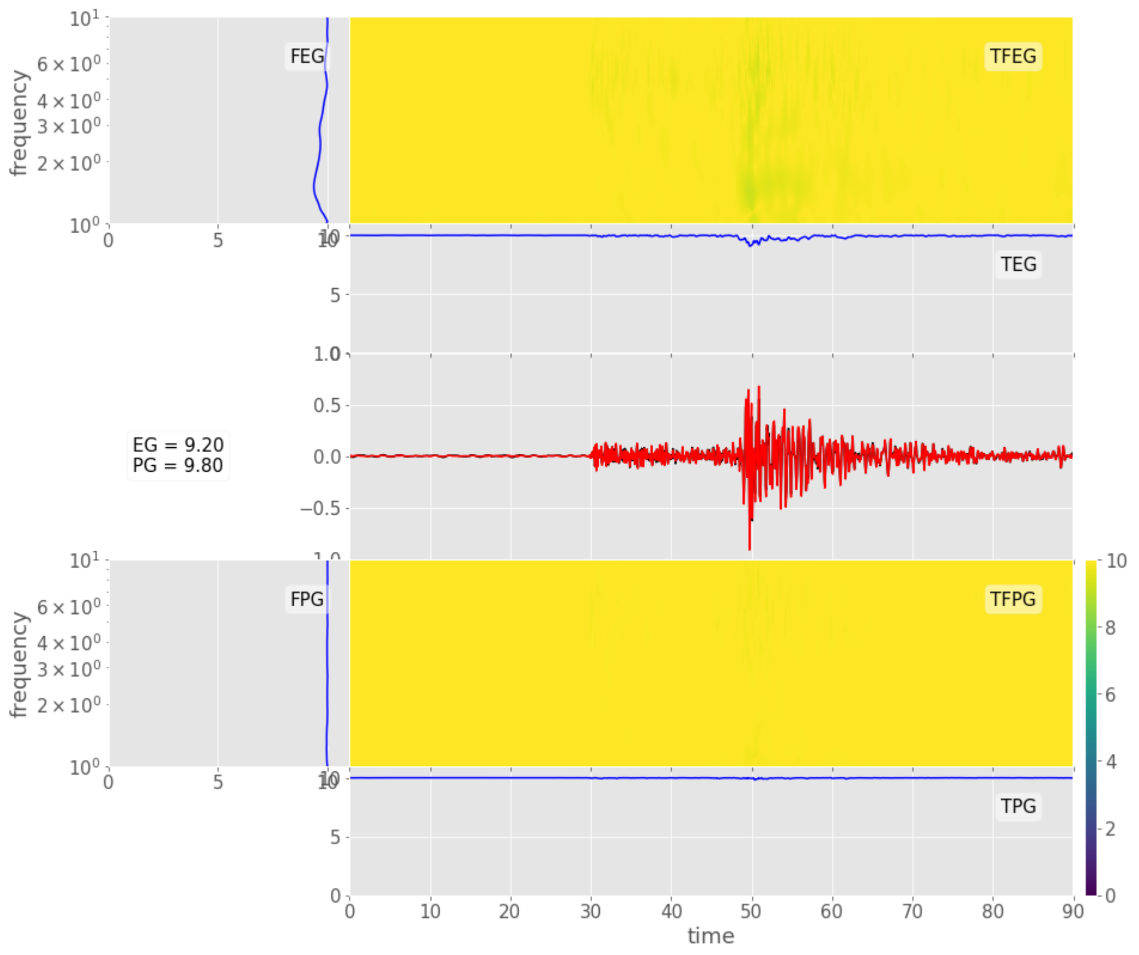

Figure 4.3: [First of two] goodness-of-fit results of randomly selected waveform from training set. Black waveform is the uncorrupted version (i.e. clean) version and the red waveform is the denoised waveform made by the DAE transfer learning model. Time axes is in seconds.

9.5 to 6 starting at the onset of the P-phase at around 30 seconds. The dominant frequency between 2 - 4Hz. Once again the TFPG looks unchanged.

In Figure 4.5 we demonstrate the signal-to-noise ratio (SNR) of the original traces $\mathbf{x}$ shown in pink, the corrupted (noisy) trace $\tilde{\mathbf{x}}$ in yellow, and the denoised waveform (the output of our DAE) in purple from the training set.

The overlap between the near overlap of SNR between the denoised and the original clean waveform, suggests that the denoiser is able to recover the original SNR. There was a loss in SNR for signals which originally had SNR values above 25Hz. However, there was an increase in the number of events with SNR values between 10-20Hz as can be seen with the higher denoised bin heights.

In the future, it would be insightful to investigate if our denoiser is still able to recover the original SNR when the difference between the clean and its associated version are larger (the yellow bins shifted to the left and the pink bins shifted to the right, for example). This is something we wish to explore in future iterations.

The envelope goodness-of-fit plots provide insight into how similar the denoised traces from our DAE are. However, we do not get a sense of what, if any, part

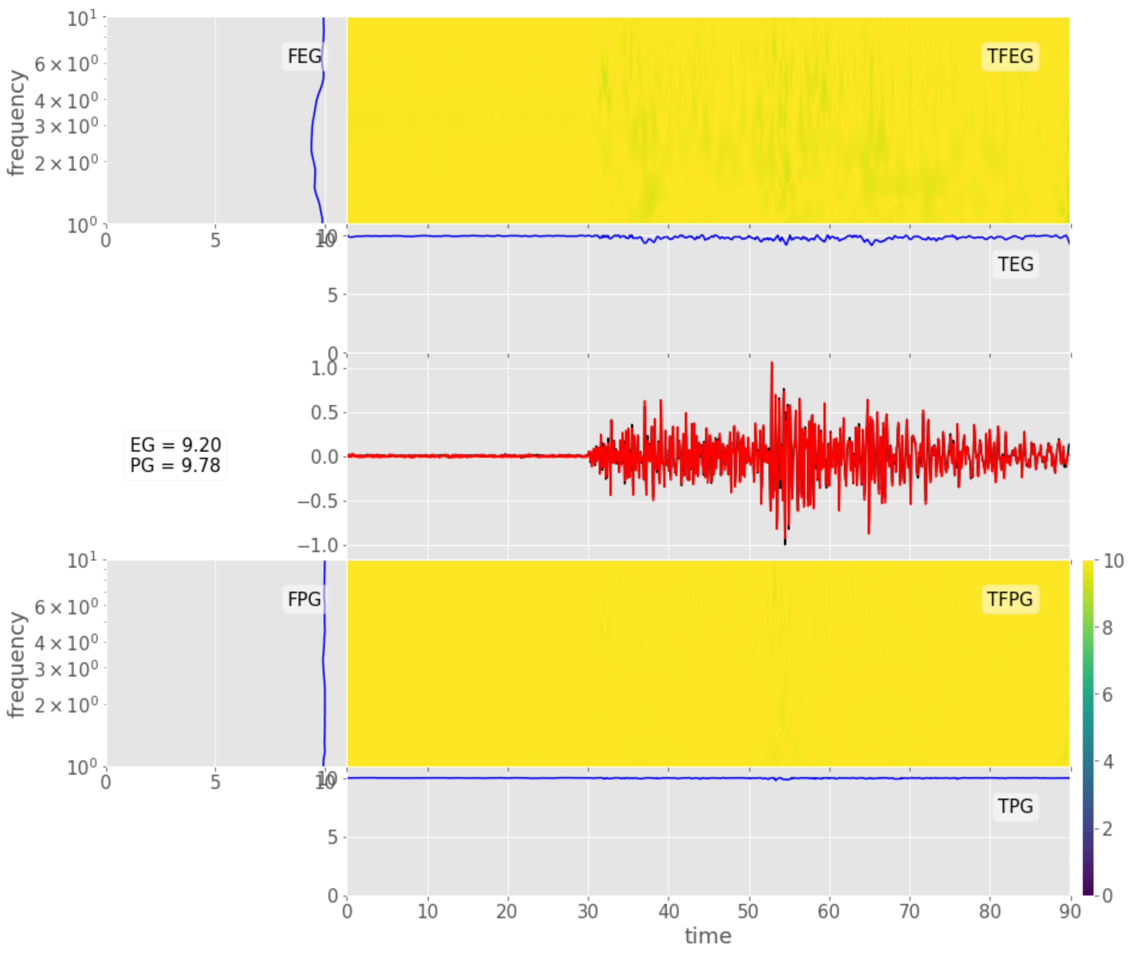

Figure 4.4: [Second of two] goodness-of-fit results of randomly selected waveform from training set. Black waveform is the uncorrupted version (i.e. clean) version and the red waveform is the denoised waveform made by the DAE transfer learning model. Time axes is in seconds.

of the physics which generated our waveforms is preserved. There are two metrics which can be used to investigate this: comparing changes in peak amplitude and phase arrival times. Unfortunately, we had to toss out looking into peak amplitude changes because during pre-processing we normalized our waveforms. This left us with changes in arrival times. We will do a quantitative analysis of how the DAE effects phase arrivals when we look at the results of the test set in the next section. The reason being, the test set tells us whether or not our deep neural network actually learned.

4.5 Test Set

The results from our model suggest the denoiser is recovering the SNR of the training data set and preserving arrival times. The results from training only tell us about the *insample* error (see Section 2.2). To determine if we have truly learned we need to look at the *out-of-sample* error.

If the entire input-output space χ and the entire output space Y are the set

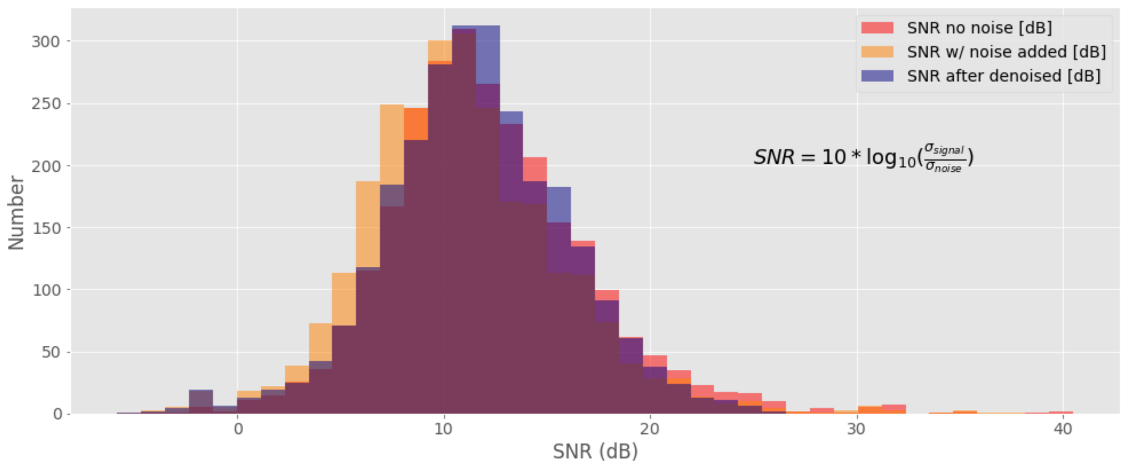

Figure 4.5: Histogram of the signal-to-noise ratio of the original/clean waveform (shown here in pink), the noisy/corrupted version (yellow), and the denoised version from our DAE (purple) from training.

of all possible inputs, $\mathbf{x}$, and the set of all possible outputs, $\mathbf{y}$, respectively, then we can define our data set as the space D which consists of input-output pairs $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_n, y_n)$ for $n = 1, 2, ..., N$[3].

During training the neural network is attempting to find the best approximation to an unknown function f known as the target function (see Section 2.1.2) using algorithms which are directly learning from the data set, D. The target function is a function of the entire input space χ and the output space Y. The approximating function, g, is based only a subset of the entire input-output space, D. Thus, despite the outputs of the initial training, one can not asses whether or not the ML algorithm has learned the true target function since it is only referring to a subset of the entire input-output space χ during training.

To find out if the ML algorithm can infer outside the data set it trained on, the algorithm is run on a data set it has never seen before. This data set is known as the test set. The test set consists of instances which are outside of our data set D. If our trained neural network is able to make accurate predictions on data it has never seen before, then we can interpret this as a means of determining the neural network has learned. Said another way, the approximating function has found weights that can fit the observations found during training but are not too specific such that it can not generalize to data it has not seen before.

For our test set we used one year of data from the NCEDC with the same querying criteria as the training data set (i.e. we searched for local events with minimum magnitude of 3.0). Data was collected from the same stations used for training: CMB, PKD, HUMO, and MNRC. With the above criteria, we were able to download approximately 1,000 waveforms.

The same pre-processing was done for the test set as was done for the training set. Each component of each record (Z, N, E) was treated as a separate training example, giving us ~3,000 waveforms. Waveforms were detrended, high-passed filtered with a maximum frequency of 0.5Hz, normalized, decimated to 20Hz sampling rate, and then cut into 90-second records. Running the model took less than 1 minute to run. Results are discussed in the next section.

4.5.1 Results from Test Set

From Figure 4.6 we note that the envelope and phase goodness-to-fit for the waveforms is exceptional, with single-valued GOF criteria above 8.0. Once again in black, we have the original uncorrupted input and in red we have its denoised version which is the output of our DAE. A decrease in the TFEG from approximately 9.5 to 7.0 around the onset of the surface waves at around 45 seconds can be seen as well. The phase content (bottom figures) show near-perfect fit with minimal degradation around 50 seconds.

There is a similar, though less pronounced, degradation for the waveform in Figure 4.7. Fluctuations between approximately 9.4 and 7.5 for the phase goodness-of-fit can be seen through the entire length of the TFEG and TFPG plots. There does not seem to be a general trend in how the phase fluctuates as a function of time. In this case, the largest contrast in GOF happens around the 70-second mark. The time-frequency phase GOF is nearly perfect. It would seem that in both cases that the goodness-of-fit for both waveforms is most noticeable during the arrival of the surface waves.

Based on the test set GOF plots it would seem that the neural network is not yet able to recover the peak amplitudes perfectly. It would be useful to quantify the changes in peak amplitude between the original waveform and the denoised waveform. However, as mentioned above, we normalized our data during pre-processing so such a comparison would not provide much insight.

In Figure 4.8 we plot the histogram of SNR for the clean, noisy, and denoised waveforms as we did with the training set. The histogram suggests the DAE was able to recover the original SNR of the non-corrupted waveforms (i.e. the clean version). The number of events with SNR between 10 and 20dB increased relative to the clean and corrupted waveforms. However, we did decrease in SNR for events which originally had SNR values between 25 to 35dB.

As mentioned in Section 4.4.1 when we were analyzing the results from training, the goodness-of-fit plots and the SNR histograms provide an insight on how well the denoising autoencoder is at recreating the waveforms (GOF) and how well it can recover the SNR. However, such analysis does not tell us if the underlying physics which generated a given waveform were preserved. Information about changes in peak amplitude must be tossed out as waveforms were normalized. This leaves us with analyzing the influence the DAE had on phase arrivals.

To investigate this, we computed short-term average/long-term average (STA/LTA) trigger algorithm to pick P-phases for all three versions of our data: the original clean version ($\mathbf{x}$), the denoised version ($\tilde{\mathbf{x}}$), and the denoised version. Using the STA/LTA derived P-phases, we computed the time differences between each combination of traces: clean vs. noisy, noisy vs. denoised, and clean vs. denoised. The results, as a function of the SNR before using the DAE and after are plotted in Figure 4.9. We used a least-square polynomial of degree equal to one to describe the linear trend of the differential times. The figure demonstrates that the difference in phase picks errors between each combination of waveform is preserved as a function of the original SNR and the SNR after denoising. This can be seen by observing that the overall linear trend looks the same for both figures. We do note that the SNR after denoising (left figure) has a systematic increase in slope for all three curves. However, the change is less than a one second difference for all three curves.

Results from the DAE on the test set suggest it can recover the SNR of the

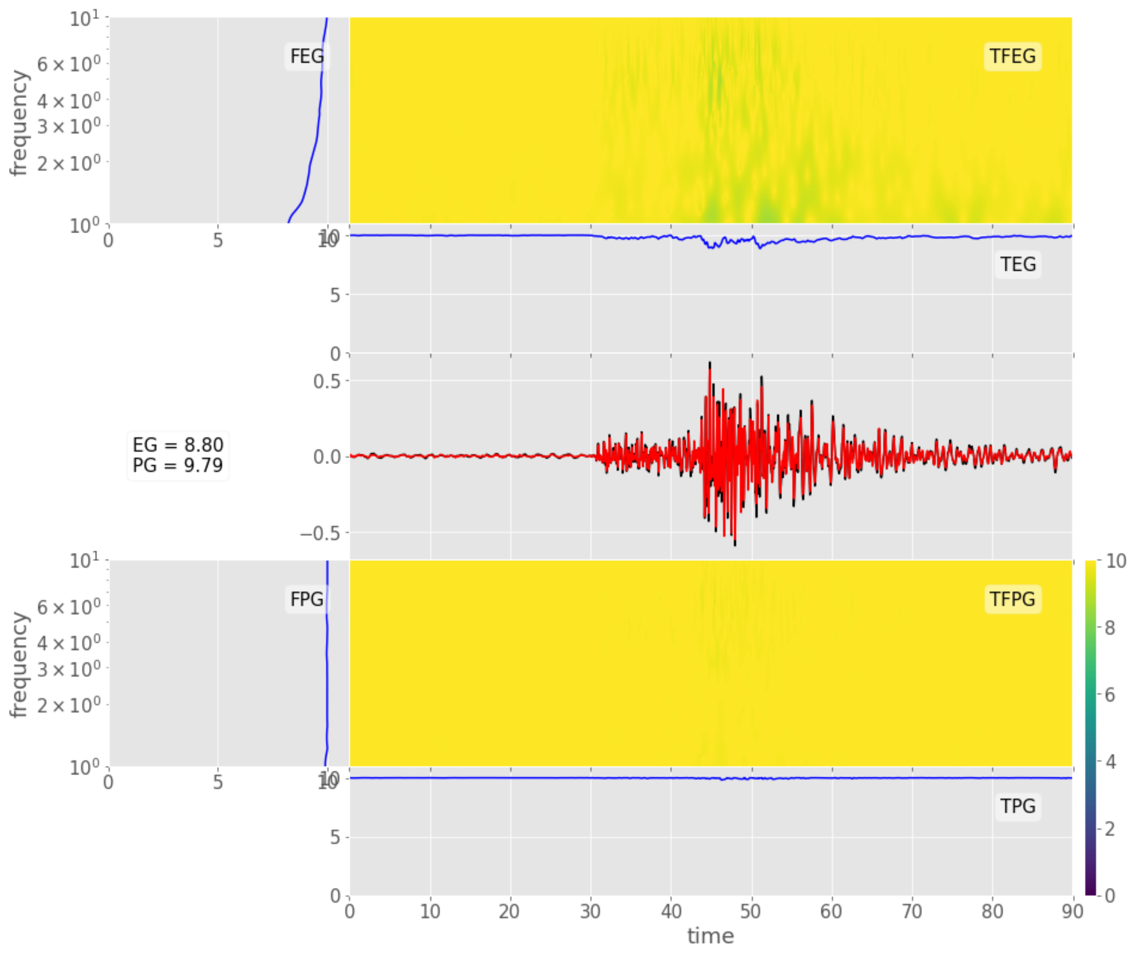

Figure 4.6: Goodness-of-fit plot of denoised (red) and original (black) waveform from data set the neural network has never seen before. Time axes is in seconds.

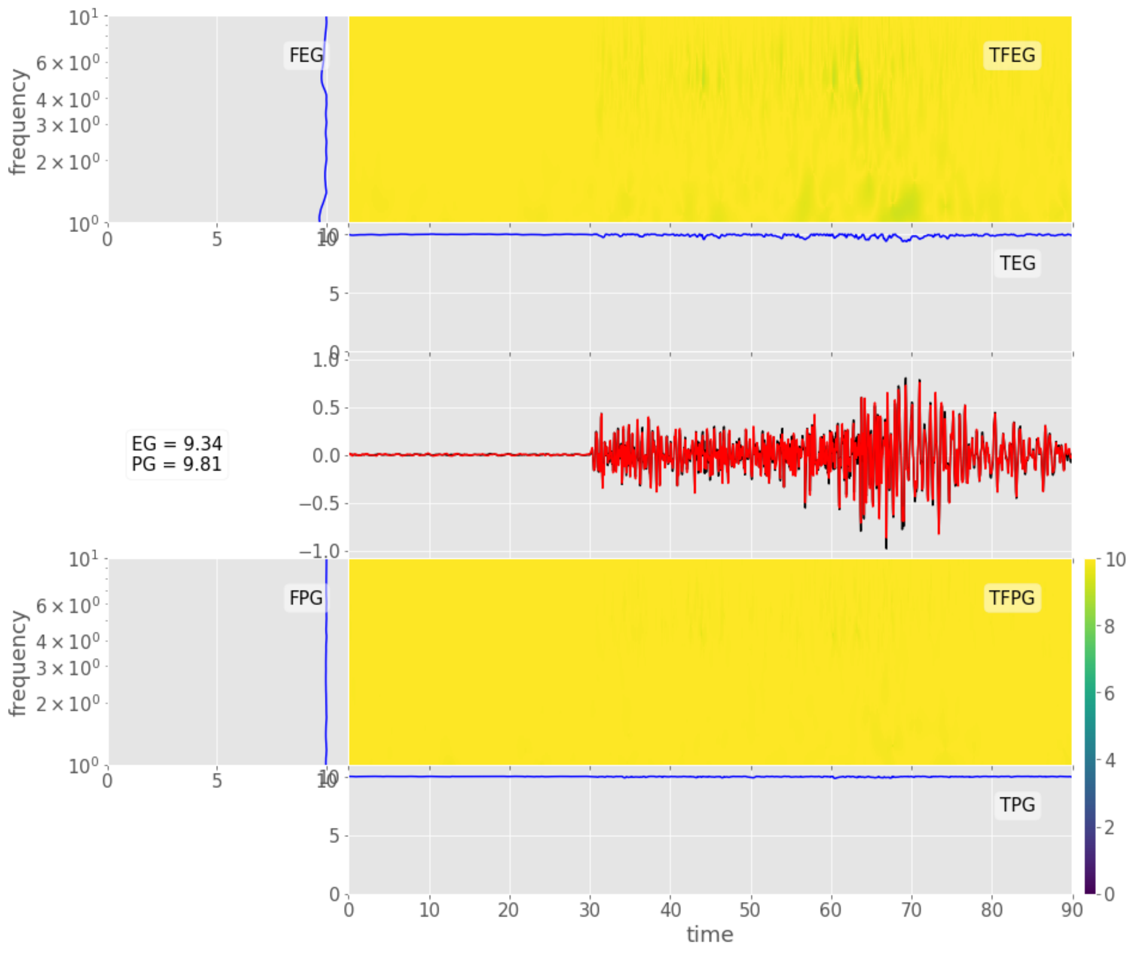

Figure 4.7: Same as the previous GOF plot but for a different randomly chosen waveform from the data set the DAE has never seen before. The denoised output of the neural network is in red and the original waveform is black. Time axes is in seconds.

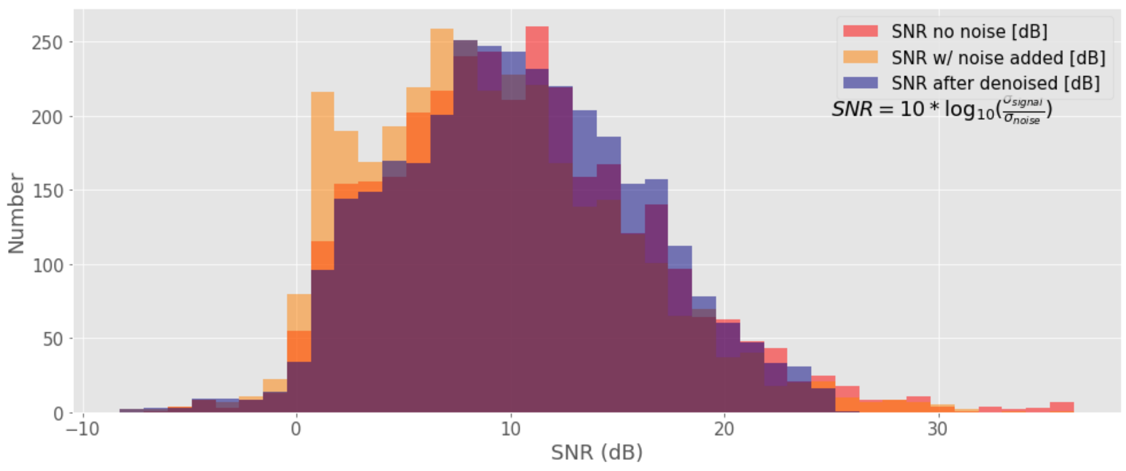

Figure 4.8: Histogram of the signal-to-noise ratio of the original/clean waveform (shown here in pink), the noisy/corrupted version (yellow), and the denoised version from our DAE (purple) using the pre-trained model on the test set.

original clean waveforms. Given that the model was trained on less than 15,000 waveforms (which is considered small data set in the context of deep learning) and considering a simple DAE (only a couple of alternating convolutional and maxpooling layers were used), gives us reason to believe we will be able to improve our model such that it could one day improve the SNR of data sets.

4.6 Transfer Learning

The denoising autoencoder demonstrated it can: recreate the original waveform with GOF above 8.0 (out of a scale of 10), recover the SNR, and it preserves phase arrivals using an automated trigger algorithm. There are several routes that will be explored improve the DAE's performance. However, given the promising results thus far, we were curious to see how the current model performs on data recorded in Wettzell, Germany. This method of using a trained neural network on a different, but similar data set, is known as transfer learning. Transfer learning is a technique that can be used in the case where this not enough data to initially train with.

There are two strategies to apply the transfer learning technique using deep neural networks: via *feature extraction* or *fine-tuning*. In our case we will use feature extraction. Because we want to exploit the weighted layers in our pre-trained neural network model to extract features in new samples [2].

4.6.1 Model

As of writing this thesis, there does not seem to be a 'best practice' of how to reuse a pre-trained autoencoder within the Machine Learning community. Generally speaking, transfer learning requires 'freezing' part of a neural network during training. In the case of convolutional neural networks (See Section 2.5.4) the shallow portion of the network is kept constant [10]. The first few layers of a traditional convolutional neural network can be regarded as 'general feature extractors' which can be reused on similar, but different data sets [16]. During transfer learning we want to prevent

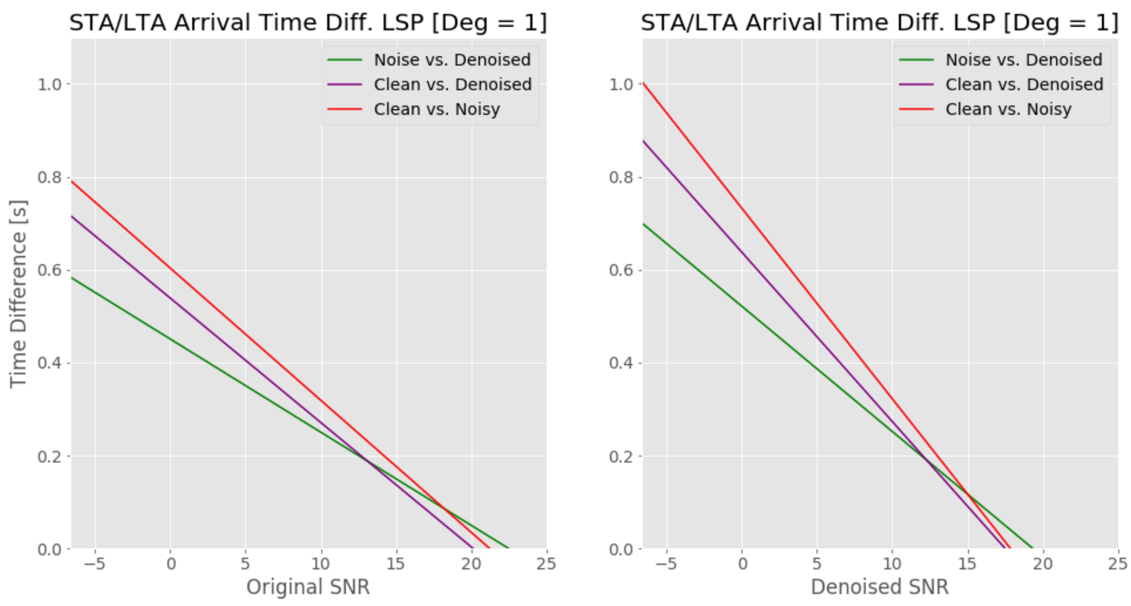

Figure 4.9: Arrival time differences (in seconds) of P-phase arrivals computed by the STA/LTA trigger algorithm. A degree one least-squares polynomial was used to fit the data.

these weights from being updated, else we will lose the features they learned and which we want to exploit for transfer learning.

Table 4.2: Model architecture of the tranfer learning denoising autoencoder. Conv = 1D convolution, Max = maxpooling, UpS = upsampling, * = new layer.

Layer	1	2	3	4	5	6	7	8	9
Stage	Conv	Max	Conv	Max	Conv*	UpS*	Conv*	Ups*	Conv*
# Channel	8	-	16	-	16	-	8	-	1

The denoising autoencoder can be viewed as a convolutional neural network with symmetry around the coding. Thus, the most intuitive approach to applying transfer learning was to keep the encoder (the two first convolutional layers and the maxpooling layers) fixed during and give it a new decoder (see Figure 4.2).

4.6.2 Data Set

We queried teleseismic events with magnitude 6 and above. From this search we were able to find approximately 780 events. Each channel (Z, N, E) recording was treated as a unique training example, giving us approximately 2,300 events to work with. Events were low-passed filtered at 0.1Hz, normalized, and cut into 90-second records starting 30 seconds before the P-arrival. Archived Wettzell data does not have manually picked phase arrivals, so we computed theoretical P and S phase arrivals using the IASP91 1D velocity model using ObsPy's built-in TauPy instance class. TauPy is based on Java TauP Toolkit by [12] and can be used to calculate theoretical arrival times based on a 1D spherically symmetric background model.

We once again downloaded data between archived events and considered this as pure noise. We added the noise to our pre-processed data set giving us two versions: a noisy and clean version. The noisy waveform, $\tilde{\mathbf{x}_T}$, and its associated original clean version, $\mathbf{x}_T$, are the input and label to our denoiser autoencoder, respectively.

4.6.3 Training

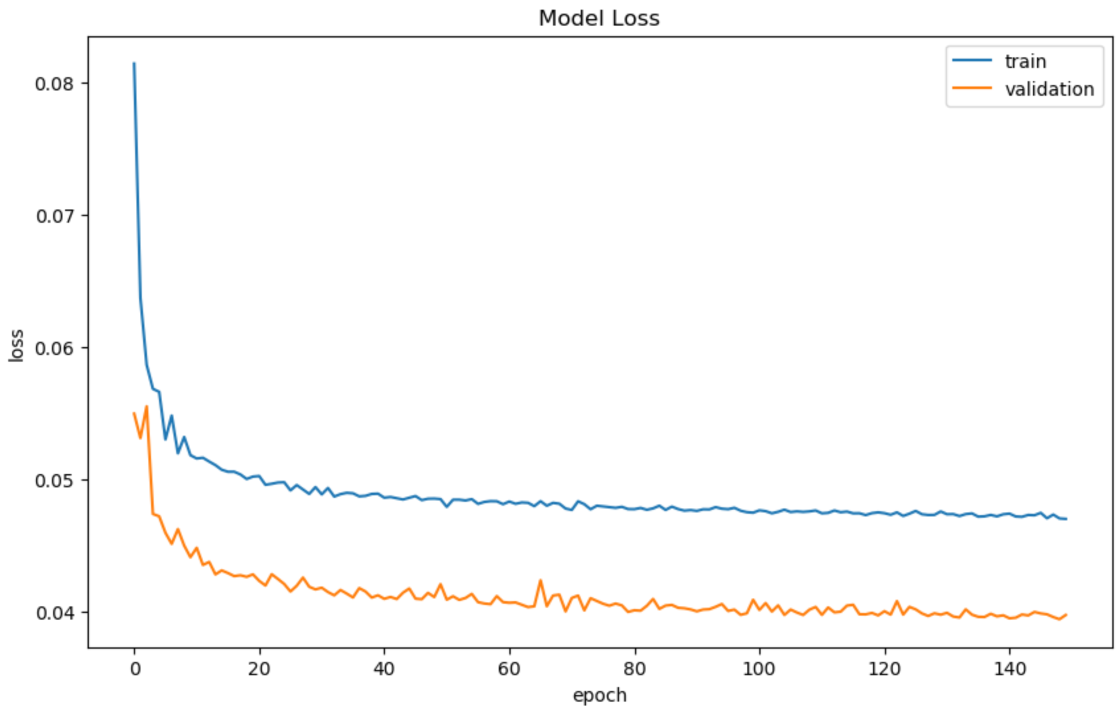

Figure 4.10: Learning curve of transfer learning model using teleseismic events from Wettzell. The algorithm is unable to improve its prediction around epoch 140.

Of the 2,300 events, approximately 1,870 events were kept for training sin460 events were set aside as our validation set. With the exception of the number of epochs, the same hyperparameters used for the original model were kept the same for our transfer learning model.

4.6.4 Results from Transfer Learning Training

Detail analysis of the results from transfer learning still needs to be investigated before conclusions can be drawn. As such, we will provide early results from this model. Starting with the loss curves computed during training, Figure 4.10 demonstrates that that machine learning algorithm is improving its predictions with experience. Train-ing was terminated after 150 epochs when both the training and loss curves were no longer improving its learning performance as a function of epoch. At this point, the transfer learning model is reaching a loss score (value) of approximately 0.04. The loss score of the transfer learning model at its time of termination is higher than the final loss score for the training model which had a value of 0.002. Suggesting the transfer learning model is unable to denoise the waveforms as effectively as the original model.

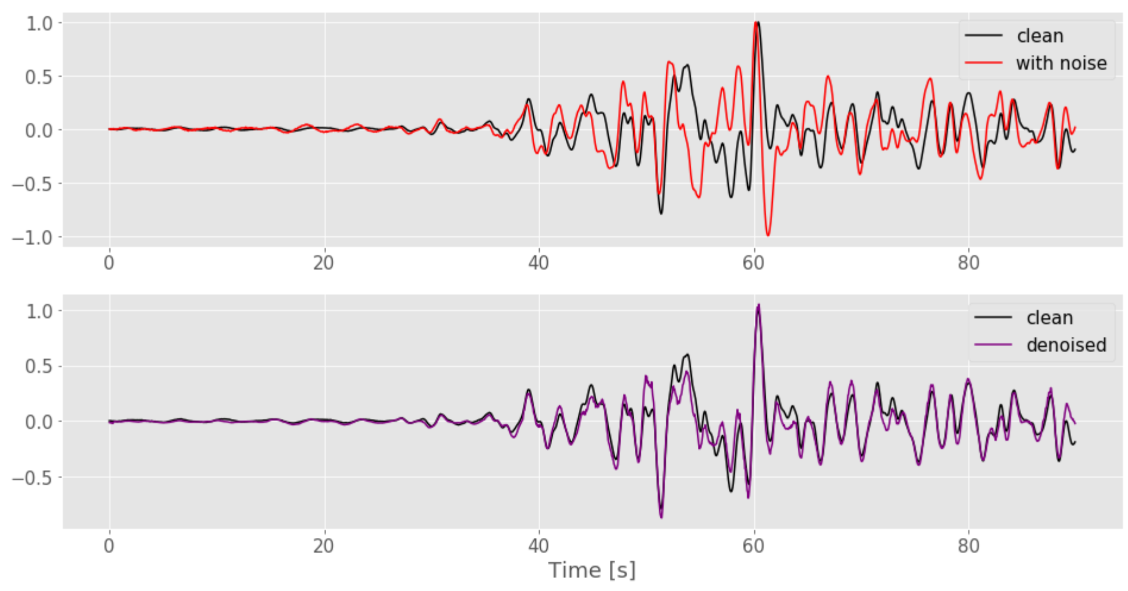

Figure 4.11: Comparison of the original waveform (black), its corrupted version (red), and the output of our denoiser deep neural network (purple) from the data set used to traing the transfer learning model.

In order to improve the DAE's performance given its current architecture (Figure 4.2) we might need to either: a) use a more complex model during the original training on the California data set or b) use more and diverse data to learn from.

In Figure 4.11 we compare a randomly selected waveform of the data set used for the transfer learning model. In black is the original uncorrupted waveform, x, in red is its associated corrupted version, $\tilde{\mathbf{x}}$, and in purple is the denoised version. Visually inspecting the two waveforms we can see the phase and amplitude of the noisy version in the top plot begins to deviate beginning at around 40 seconds. The denoised version, though it has phase and amplitude characteristics closer to that of the clean version, is not perfect. We can confirm this with the goodness-fit-plot in Figure 4.12. Both the TFEG and TFPG witnesses a degradation from GOF values of approximately 9.5 to 6.8 starting around 40 seconds. The time-dependent envelope (TEG) highlights this and we can see the largest contrast around 60 seconds. This suggests that, once again, our DAE struggles most at preserving the peak amplitude relative to the rest of the waveform.

With much left to explore with the original model, the results presented here provide only a glimpse of what transfer learning can do for the Wettzell. It remains to be seen if, once we have a robust model, if we can effectively apply transfer learning. We discuss the next steps in store for the transfer learning model in the next section.

4.7 Discussion & Further Work

There are several items we would like to investigate in order to improve our neural network. As a start, we will increase the training data set to include all data retrieved from 1999. At the moment we limited ourselves to using only five years worth of

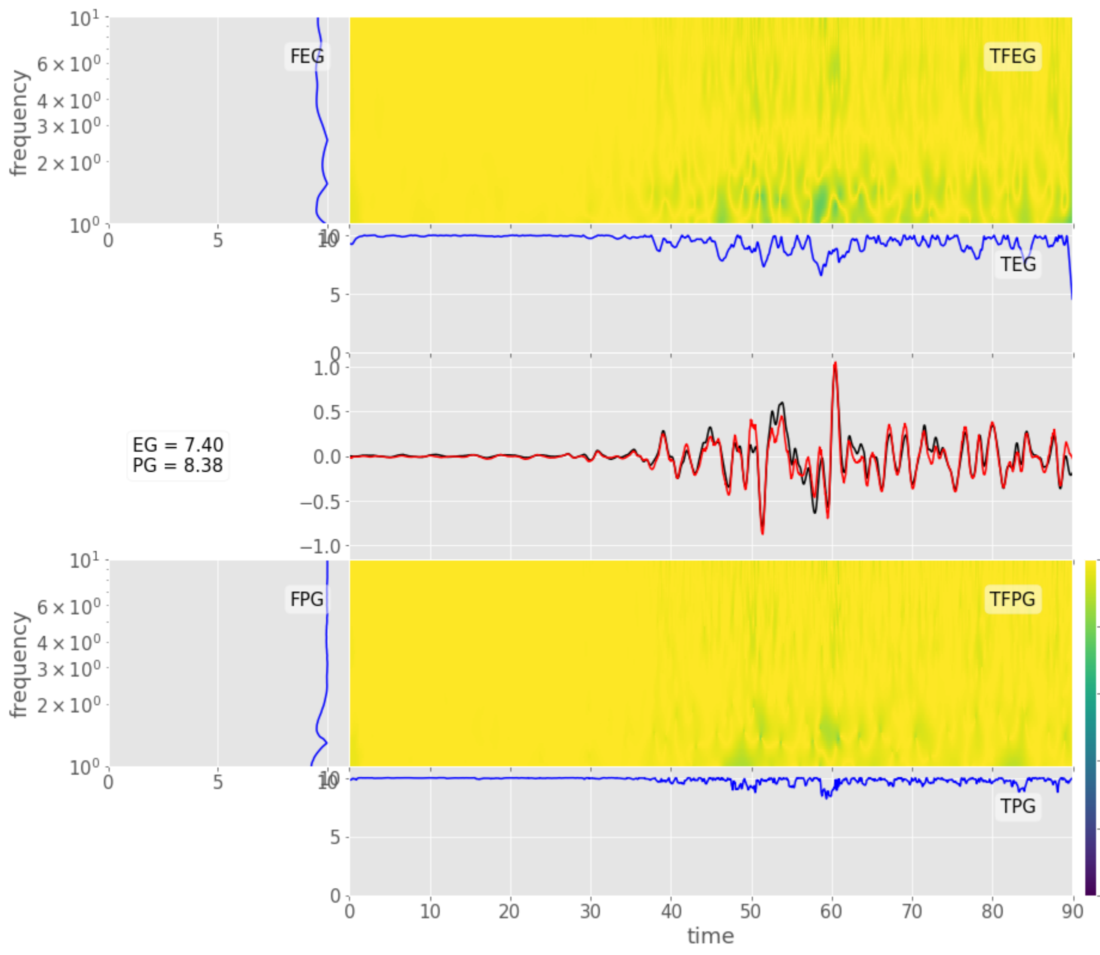

Figure 4.12: Goodness-of-fit of the waveform shown above. The black waveform is the uncorrupted version (i.e. clean) version and the red waveform is the denoised waveform made by the DAE transfer learning model. Time axes is in seconds.

data. This was done to minimize computational time and storage space while we were in the development stage of building and testing different model architectures. Including more data will increase the diversity of the domain space from which our machine learning algorithm can learn from. Thus, improving its ability to denoise future waveforms it has never seen before.

To expand our data set, we plan to vary the noise level added during pre-processing. This is a form of *data augmentation* and it is a strategy used by practitioners to increase the diversity of the data available for training without collecting new data [8].

As discussed in Section (4.4.1), we need to shuffle our data set. Shuffling data sets is a pre-processing technique done to ensure that a bias isn't introduced during training. In other words, for each training epoch the machine learning algorithm is grabbing a distribution that is representative of the entire data set, D. This is currently not the case for our data set. Currently, we have data being read in where stations CMB and PKD make up most of the validation set and stations MNRC and HUMO make up the training set. CMB and PKD typically have higher SNR thatn MNRC and HUMO, thus it makes intuitive sense that the machine learning algorithm is able to make recreated these less noisy stations in the validation set than in the training set.

Once the above steps have been satisfied we plan to apply transfer learning on teleseismic events recorded by the same California stations used for training (PKD, CMB, HUMO, and MNRC). Given that we are using four stations (instead of just the one in Wettzell) there will be more teleseismic data to train with. In addition, the corruption process, in this case the noise characteristics of the CA stations, will be the same during training. Right now our machine learning algorithm is confronted with both: new seismogram characteristics (teleseismic as opposed to local) and different corruption process (noise in CA versus noise in Germany). Having been exposed to teleseismic events during training, our neural network might perform better on the Wettzell data set.

We are interested in exploring how our denoiser performs if, instead of using 1D time-series data, that we instead use 2D time-frequency domain images. The inputs of our neural network would therefore consist of both the real and imaginary parts of the time-frequency coefficients. Its associated label would then consist of a corrupted version of these time-frequency coefficients. Using 2D images and two inputs per training example, $\mathbf{x}_i$, might allow the model to make more complex outputs. This would require using 2D convolutional layers, 2D kernels, and perhaps more fully-connected feed-forward networks. In doing so we could compare which model is better in denoising our data.

Given that there are other groups [52] [35] [50] looking into using machine learning algorithms for denoising, it might be worthwhile to conduct a comprehensive study comparison. In particular, it might be of interest to compare computational resources used for pre-processing and training. For example, Zhu et. al used convolutional neural networks (the basic of our autoencoder) for their DeepNoiser. Their pre-processing included converting time-series data into the frequency domain. Where they then gave the neural network the real and imaginary parts of time-frequency coefficients.[52] In our current set-up did minimal pre-processing by leaving our data set in the time-domain and directly feeding it the neural network.

Finally, given that our current machine learning workflow removes amplitude information during pre-processing, it is of interest to have a model that works with unnormalized data. We chose to work with normalized data because we wanted to start off with a simple neural network; we wanted to create a neural network with minimal features to learn. The range of peak amplitude values in our data set might have required we use more hidden layers and thus required more data to learn from.

4.8 Conclusion

Multiple disciplines depend on seismic data to explore open questions and gain insight about Earth. Having a high signal-to-noise ratio is critical given the critical role seismic data plays in the geophysical community. There have, and will continue, to be different approaches to improving the SNR in seismic data. The goal of this Master Thesis was to develop a new approach to denoising seismic data using Deep Neural Networks.

To investigate this we trained a denoising autoencoder, a type of deep neural network architecture which uses a combination of convolutional neural networks, maxpooling, and upsampling layers to first reduce then rebuild the dimension of a given input. In doing so it forces the neural network to learn which features are critical in representing the input (but at a lower dimension). The given process inherently denoises corrupted inputs when its associated input-pair is uncorrupted.

We were originally set out to create a denoiser for the inertial sensor in Wettzell, Germany. However, given the limited number of events we could use for training we decided to first train on local seismic events recorded by stations part of the Northern California Earthquake Data Center (NCEDC).

Several model architectures and hyperparameters where tested before we settled on a configuration that looked most promising. With this model we were able to train a DAE to denoise waveforms.

With results looking promising, we then tested waters and applied transfer learning on teleseismic data from Wettzell. Results at this point are preliminary, but the current outcome looks like our denoising autoencoder is on track to denoising Wettzell data.

The current model is capable of recovering the SNR of waveforms it has never seen before. Given the current results and the list of tasks which we have in store for this model, we are optimistic we will be able to reach a point where our DAE will be able to improve the SNR of waveforms.

ACKNOWLEDGMENTS

Waveform data, metadata, or data products for this study were accessed through the Northern California Earthquake Data Center (NCEDC), doi:10.7932/NCEDC.

Chapter 5

S-Wave Phase Detection on Rotational Seismic Data

5.1 Introduction

The growing excitement behind single station 6 degree-of-freedom (three translational and three rotational) observations is well justified. Improved instrument sensitivity within the last two decades has fueled efforts in exploiting the theoretical framework that allows the reconstruction of the complete motion of a measurement [5].

Adding a single rotational measurement has been shown to improve the determination of earthquake locations [20]. In part because rotational measurements have been shown to narrow the degree of azimuth uncertainty and can separate S-phases [13].

In theory it is possible to locate earthquake sources using a single translational seismometer. To do so one uses the P-wave polarization to infer the azimuth to the epicenter. To do so, one can use the P-wave ratio of the two horizontal components of a translational record to find the vector projection of the P-wave along the azimuth to the source. The source distance itself is found taking the travel time difference between the P- and S-wave. With the distance known, one can then approximate the P travel time and thus find an origin time. However, this simplistic approach is not accurate for distances greater than 20 degrees. A distance greater than 20 degrees make it difficult to identify the P-wave arrival on the horizontal components.

Because of the limitation the above method introduces, several groups have proposed alternative methodologies [46] [39] [34]. Most of the proposed approaches are computationally heavy and require a heavy amount of pre-processing because of the statistical framework from which they operate on [9].

The above-mentioned limitations of using a single three-component translational sensor in locating earthquakes, interest in 6 Dof and 4 Dof (three translational and one rotational component) has grown. Additionally, the necessity to exploit rotational data has gained new traction as with the successful deployment of

While the theoretical and instrumental framework to take full advantage of 4Dof and 6Dof is developing, there are still hurdles to face. Noise, contaminated translational data by the rotational component, features unique to rotational not yet well-understood (among other things) complicate identifying phases and the back azimuth.

This is where Machine Learning can help. Machine Learning is a branch of artificial intelligence based on the idea that systems can learn directly from data, identify patterns, and make decisions with minimal human intervention. It is a method of data analysis that could bring new insight to the difficulties facing 6Dof data.

To break the larger list of challenges currently facing 6Dof data, we will first focus on developing a machine learning algorithm capable of accurately picking S-Phases using 4Dof data. To do so we will train an Artificial Neural Network to pick S-Phases.

This paper is structured as follows: first, we will invoke inspiration by describing observations using 4Dof has had on improving earthquake location. In the same section, we will take a look into the current state of machine algorithms used for phase picking and detection. Next, we will look into our machine-learning workflow. We will then discuss the results from training and our test set (see Section 2.2.1) before taking a quick preview into the results from transfer learning (Section 5.5). We will end by going over the next steps in store for our algorithm which will allow us to fully explore what role Machine Learning can bring to 4Dof and 6Dof observations.

5.2 Background

5.2.1 Earthquake Location using Rotational Data

In order to determine the source location of an event one must accurately identify seismic phases, measure their arrival times, and have knowledge of the velocity structure between the seismic station and the hypocenter [31]. Knowledge of the back azimuth determines the longitudinal and transverse direction of the incoming array (relative to the station).

The classical approach of identifying S-phase arrivals measured by inertial sensors involves a careful analysis of the radial and transverse (rotated horizontal N-S and E-W) components of the record. This is done because the particle motion of SKS and SKKS waves are primarily SV in nature and thus most present on the radial component. The Sdiff is characterized as a SH wave and can be most present on the transverse component [42]. In practice, identifying the S-phase is challenging because, depending on the source distance, the signal may be convoluted by: the P-coda, reflections from previous-arriving phases, and microseisms (i.e. noise).

Taking advantage of rotational data can aid in this endeavor. Heiner et. al (2007) showed that there exists a high cross-correlation (near 1.0) between transverse translation and vertical rotation for the S-wave onset (as well as Love waves). With this, the ambiguity of S-Phase picking can be reduced and thus subsequently aid in more accurate in determination of locating sources.

Assuming plane wave propagation (Eq. 3.18) we expect the vertical rotation and transverse velocity seismograms to: have identical waveforms and that their amplitudes are proportional to twice the phases velocity. As a result of the similarity of the transverse velocity and vertical rotation one can investigate the source direction of SH-type motions. [17] [11] [21] [22] found that rotating the transverse component of a translational seismic record, until it reaches a maximum resemblance to the rotational measurement, corresponds to the back azimuth of the event. [13] showed that computing back azimuths with rotations can reduce the back azimuth error to less than 10 degrees. This is in contrast to [9] which had errors on the order of 30 degrees in addition to the 180 degree ambiguity [13]. The combined improvement of S-Phase detection (and thus arrival times) and back azimuth estimations using a single rotational measurement, can thus lead to improved source locations.

Applying theory to all real observations has proven nontrivial. First, translational observations are contaminated by rotations. There is a geometric effect that comes into play with horizontal components of strong motion and long-period observations. In particular, these observations are sensitive to tilt. It is theoretical possible to correct this effect by using the measured seismometer tilt, however, this has yet to be fully successful [20] [21].

In addition to contamination by the translational components, rotational measurements (like standard inertial sensors) are susceptible to noise. Rotation sensors still can still not achieve similar signal-to-noise ratios as standard inertial (translational) broadband sesimometers [22].

Given the challenges that face 6Dof, and thus 4Dof observations, we are looking into whether machine learning can provide new insight. The focus of this master thesis is to develop a machine learning algorithm which can accurately pick S-Phases using 4Dof data.

5.2.2 Deep Neural Networks & Phase Picking

There are numerous neural network architectures to choose from. Of interest to us are Convolutional neural networks 2.5.4. Convolutional neural networks (CNN and sometimes refers as ConvNets) are powerful feature detectors which are immensely popular in the field of computer vision, natural language processing, and image analysis and classification. The design of a CNN was inspired to mimic that of the hierarchical organization of human eye-sight. The first features detected by our neurons are simple patterns such as circular shapes. On top of these cells are neurons which can then detect edges. Moving deeper into more complicated cells, the neurons begin to combine detected lower feature levels and combine them into one complicated image.

As of writing this paper, there are several working groups exploring Convolutional Neural Networks for phase picking and detection. [36] used three-component translational seismic waveforms to train their neural network to generate probability distributions of P arrivals, S arrivals, and noise as output. [51] used three-component translational seismic waveforms to train their neural network to generate probability distributions of P arrivals, S arrivals, and noise as output. Their CNN, named 'PhaseNet', is demonstrating it can achieve higher picking accuracy and recall rate than current statistical and automated phase picking method [51]. Using the vast, hand-labeled data archives of the Southern California Seismic Network were able to train a ConvNet to detect and classify seismic body wave phases over a broad range of waveform types (e.g. local vs. teleseismic events). Their generalized phase detector framework could improve seismicity catalogs with detection sensitivities comparable to template matching catalogs, but without their inherent biases [37].

Given the promise of convolutional neural networks in phase picking and detection, we chose this as our model architecture of choice.

5.3 Methodology

5.3.1 Software

Open-source software and free data available to the public were used for the execution of this Thesis. The list of open-source machine learning libraries for use is constantly expanding, as is the 'crowd favorite'. We used the open-source Keras Python Deep Learning library API [26]. It currently has one of the largest community support, fastest-growing, and most scholarly cited (among other criteria)[47]. Keras is user-friendly, modular, extensible, and there are a plethora of tutorials online to learn from. The heavy-lifting tensor multiplication and convolutional operations are done with TensorFlow. TensorFlow is another open-source software that performs its computations with an optimized C++ backend.

5.3.2 Data Set

Our goal was to train our CNN using four-component data from Wettzell, Germany; the vertical rotation rate from the G-Ring and the translational three-component collocated STS-2 broadband sensor. In order to have a clear separation of the P- and S-arrival, we queried the German Regional Seismic Network (GRSN) for teleseismic event with minimum magnitude of 6.0 and above.

However, at the time of our search we were only able to retrieve 250 teleseismic events from 2009 - present day. Events were chosen from 2009 and later because the signal-to-noise ratio was improved by a factor of three [22].

Even with data augmentation, a technique that artificially expands the diversity of one's data set without collecting new data, would not create enough data to train with. There is no set precedence of how much data one should have in order to train a machine learning algorithm. Some postulated that one would have about ten times as many training examples as one would have parameters to train[3]. Based on the work done by [36] [51] [37] we wish to eventually work with +40,000 unique station recordings.

Instead, we turned our attention to using synthetic waveforms generated using InstaSeis [23]. Instaseis calculates broadband seismograms from a database of Green's functions. The databases are generated by AxiSEM (axisymmetric spectral element method) and given by a Python interface for convenient extraction of seismograms [14]. For our synthetics we used the IASP91 model which has a 2-100 second resolution.

5.3.3 Preprocessing

Each four-component record was low-passed filtered at 0.1Hz and normalized. Keeping in mind we want to use data collected from Wettzell in the future, our synthetics were resampled to 20Hz thus mimicking the same sampling rate. The one-hour records were trimmed to 60-second records, centered around the computed S-arrival which were computed using theoretical arrival times from the IASP91 1D velocity model using ObsPy's built-in TauPy instance class. TauPy is based on Java TauP Toolkit by Crotwell et. al. [12] and can be used to calculate theoretical arrival times based on a 1D spherically symmetric background model. Records were trimmed for two reasons: we wanted to restrict the input length of our training data and reduce the storage space of our data set.

By limiting the window length we reduced the number of parameters (see Section 2.5.4) our algorithm needed to learn. In doing so we reduced the size, complexity, and thus the number of training examples our algorithm needs. In our case, limiting the window size to a one-minute record reduced our input size by a factor of four, from a 7200npts to 1800npts input array. The window size of 60 seconds was chosen to give a buffer from theoretical arrival times.

To more closely resemble real seismic data we added noise to the vertical rotational component in our four-component records. The noise was found by searching through the Wettzell earthquake catalog with a minimum magnitude of 4.5 within the last decade. Data between cataloged events were downloaded and then added to the rotational component of our synthetic records. In the future we plan to add real noise to all components (see Section 5.6).

5.3.4 Model

Our model is based on a scaled-down version of the Generalized Phase Picker by [36] (Figure 5.1). For their model they used a convolutional neural network. CNN are a type of neural network architecture designed to handle grid-like structured input, such as time-series data (see Section 2.5.4). As the name suggests, convolutional neural networks perform convolutions instead of standard tensor multiplication.

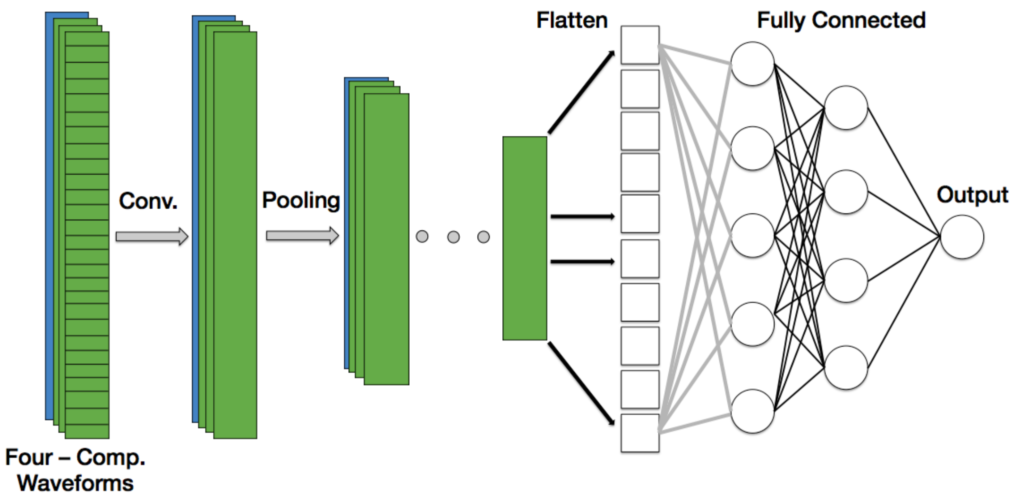

Figure 5.1: Convolutional neural network set-up used for training on synthetic 4Dof data. The highlighted portion in green depicts the convolutional base. The non-highlighted portion (i.e. white) describe a general fully-connected neural network.

The rule of thumb when designing a neural network is to first start shallow, i.e. with a minimal number of layers. If there are indications that the model is not learning (e.g. the learning/loss curve does not improve as a function of epoch), then it is at this point one adds more layers. For our setup we used four alternating convolutional and maxpooling layers (see Section 2.5.4). During training the convolution layer performs a dot product using a *kernel* (the element which is performing the convolution) and the incoming input layer. The output is then a *feature map*, which are equivalent to weights in a standard feed-forward fully-connected network (see Section 2.5.1). During training, the neural network is updating these feature maps. For each maxpooling layer applied, the dimension of the incoming input (i.e. the output of the previous CNN layer) is reduced by the chosen filter size.

The output of our convolutional and maxpooling layers are then fed to a fully-connected feed-forward network with batch normalization (see Table 5.1 and Section 2.5.4 for model summary). The feed-forward takes our extracted features learned from the CNN and uses this to output an S-pick pick.

The hyperparameters (parameters not updated or changed during training and are specified at the time of compilation) chosen were found via trial and error. This is done because there does not exist an algorithm for knowing which hyperparameters will work best during training. However, there are some hyperparameters (e.g. which cost function used) which should be avoided depending on the objective of our model (see Section 2.4.1). It is also common practice to read current literature about similar successful machine learning projects to help narrow the field of options.

The hidden layers used rectified linear unit (ReLU) activation functions (see Section 2.5.2). ReLU activation functions are a popular choice for deep neural networks because they can be combined to create nonlinear functions, thus allowing us to model more complicated target functions (see Section 2.5.2). In addition, because ReLUs will can print out 0 (2.9), they can lead to sparse neural networks and thus reducing computation time for both the forward and backward pass. The final layer (i.e. the output) was defined by a linear activation function. A mean squared error loss/cost function (see Section 2.4.1) was used to assess model performance during training and validation. We used an Adam optimizer algorithm, which is an extension of stochastic gradient descent, to iteratively update our network weights during training. Some of the attractive features of using the Adam optimizer include, but are not limited to it being: computationally efficient, minimal memory requirements, and it can handle well problems with large data sets and/or parameters [27].

Table 5.1: Model architecture of the S-phase picker. Conv = 1D convolution, Max = maxpooling layer, FB = fully-connect layer and batch normalization, F = fully-connected layer

Layer	1	2	3	4	5	6	7
Stage	Conv	Max	Conv	Max	FL	FB	F
# Channel	8	2	16	2	-	8	1

5.4 Training

To train the CNN we gave our pre-processed four-component data as input, **x**, and its associated s-phase pick as its label, **y** (See Section 2.3.1 on supervised vs. unsupervised training). Data was fed into the neural network in batches of 200 training instances.

The data set was split into a training and validation set. 80% of the data was set aside for training and the remaining 20% was used for validation (Section 2.2.1.

The training time for our simple neural network took on average between 8 to 20 minutes. (Depending on the number of epochs used, cost function used, etc.).

The output of our regressor (Section 2.3.2) for a given input, $\mathbf{x}_i$, is an s-phase pick prediction made by the machine learning algorithm. For each epoch (i.e. a single cycle through the entire training set) the machine learning is trying to find the relevant features in the four-component waveforms necessary to identify the S-pick.

5.4.1 Results from Training

A quick diagnostic tool to evaluate whether or not the machine learning algorithm is learning is to look at learning curves. Learning curves describe the experience on the x-axis and learning (or improvement) on the y-axis (Figure 5.2). An epoch describes one full cycle of making predictions on the training set. As a function of epoch we expect the loss (sometimes called 'cost') to decrease for both the data it is directly learning from (i.e. the training set) and the data set used to validate it learning (i.e. the validation set). From these plots it is possible to identify warning signs of overfitting, underfitting, an underrepresented training set and underrepresented validation sets (see Section 2.2).

Our training results demonstrate that indeed, our machine learning algorithm is learning given that the loss (i.e. the error in prediction) is decreasing with epoch. The red curve describes the model's performance on the training data while the blue curve shows the model's performance on the hold-out set, i.e. the validation set. During training the machine learning algorithm is checking how good/bad it is at making predictions (in our case, S-phase picks) compared to the true value (the S-phase it was given).

We terminated training once the loss value of our algorithm stopped decreasing. At this point the machine learning algorithm is no longer able to improve its prediction capabilities. Letting the neural network to continue running could lead to overfitting (see Section 2.4.2).

The fact that the loss and validation curves are nearly identical suggests that the machine learning algorithm is performing equally as bad (or good, depending on your point of view) in making predictions on both data sets. Which makes sense given that our synthetic waveforms probably do not exhibit much diversity given that events were generated using a pseudo-random Python function. Thus, there is a chance we had repeated seed values in our code which could have lead to repeated events in our data set.

In Figure 5.3 we compare the theoretical (blue vertical line) pick and the pick made by the convolutional neural network (purple vertical line). Note that the picks are virtually identical and difficult to differentiate when plotted together. Quantifying the arrival time differences (Figure 5.4) we can see that our machine learning algorithm is accurate in picking the S-phase arrival to within less than a second.

The results from training suggest that our convolutional neural network is learning how to pick S-phases from our 4Dof synthetic data. And it is doing with less than 0.5-second error. To confirm the neural network has truly learned we need to check the accuracy of the model after it has been given data it has never seen before, the test set. Doing so will help confirm that the weights found during training are fitted well to model the target function describing the data set it was given (training data set), but general enough that it can make predictions on data it has never seen before.

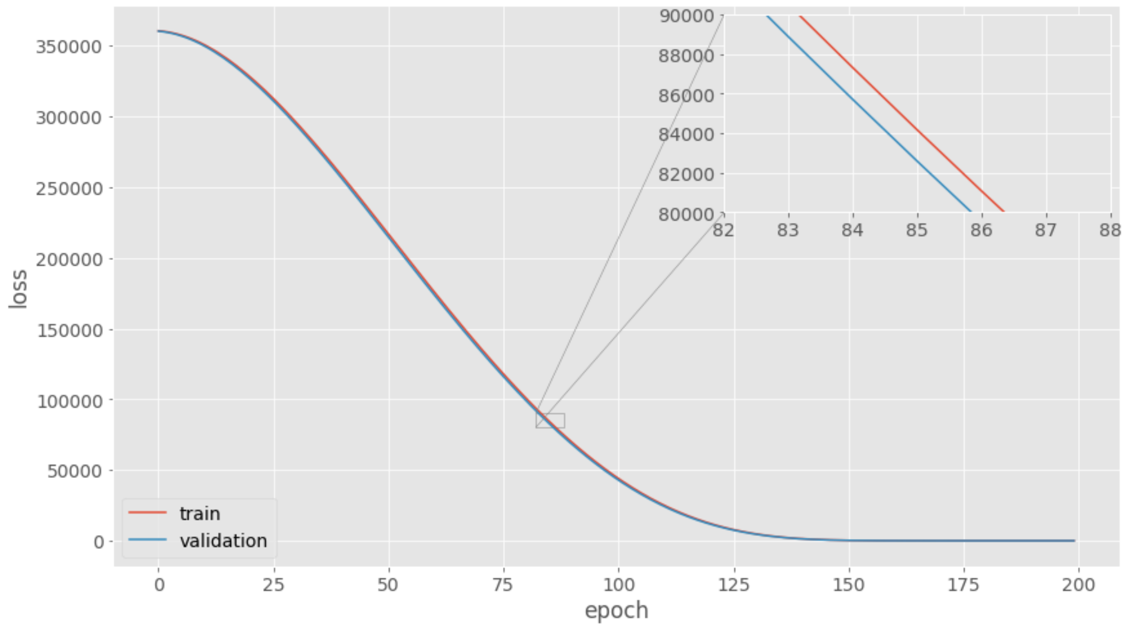

Figure 5.2: Loss (learning) curve using a mean squared error loss function during training on 4Dof synthetic waveforms. Zoomed-in plot displayed (upper right) to show the nearly identical behavior of both curves.

The fact that the validation and training curve looks nearly identical, this would suggest that our data set might be too similar. As mentioned above it could be that we have repeated events. With this in mind, we did not formally use a test set to validate the results from training. This will be done after we have created a larger and more unique data set.

Aside from confirming our training data set can generalize, training our model on a larger and more diverse data will provide an opportunity to model more complex approximation functions (see Section 2.1.2). This might prove useful if we one day want to migrate our model from making predictions on purely synthetic data to real data recorded from Wettzell.

5.5 Transfer Learning on Wettzell Recordings

Given the above results, we were encouraged to use our convolutional neural network, which was trained using synthetic data, on teleseismic events recording by the G-Ring laser and the collocated STS-2 broadband seismomter at the Wettzell Observatory in South-East Germany.

To use our pre-trained model we applied a technique known as Transfer Learning. The idea is to use a trained neural network (the fine-tuned weights, features extracted, etc.) on a statistically different, but similar data set. We want to exploit what has been learning on using the larger synthetic data set to help generalization in a different data set (see Section 2.2).

Transfer Learning is a technique that can be used in the case where this not enough data to initially train with. For example, one could use a neural network to classify images of bears first, then reuse part of this pre-trained neural network on classifying images of cars.

There are two strategies to apply the transfer learning technique using deep neural networks: via *feature extraction* or *fine-tuning.* In our case we will use feature extraction because we want to exploit the weighted layers in our pre-trained neural network model to extract features in new samples [2].

5.5.1 Model

To reuse our pre-trained neural network's weighted layers, we 'froze' the convolutional base (Figure 5.2) and introduced a new, untrained, fully-connected feed-forward network.

In the case of convolutional neural networks the shallow portion of the network (i.e. convolutional base) are kept constant [10]. These first few layers can be regarded as 'general feature extractors' [16]. In other words, we prevent the weights in part of our model from being updated during training to ensure the features/representations learned during the original training are not lost or 'erased' during training [4].

With exception to the number of epochs, the same hyperparameters used for the initial model trained on synthetics were used for the transfer learning model. During training the feed-forward, densely connected neural network (i.e. the portion of the network not including the input layer and convolutional base) uses the features extracted from the previous model to make predictions on the new data set. The hope is that the features in the synthetic waveforms which allowed the neural network to make accurate S-phase picks, are similar enough to Wettzell data thus allowing the transfer learning model to make accurate S-phase picks.

Table 5.2: Transfer learning model architecture of the S-phase picker. Conv = 1D convolution, Max = maxpooling, FB = fully-connect and batch normalization, F = fully-connected, * = new layer.

Layer	1	2	3	4	5	6	7
Stage	Conv	Max	Conv	Max*	FL*	FB*	F*
# Channel	8	2	16	2	-	8	1

5.5.2 Data Set

We downloaded events with a minimum magnitude of 6.0 from the rotational G-Ring laser and the collocated STS-broadband sensor at Wettzell (thus, 4-component data consisting of the vertical rotational component and the translational three-component data). From this we were able to retrieve roughly 830 events.

The same pre-processing done on the synthetics was done for Wettzell Data. Each four-component was first low-passed filtered at 0.1Hz and normalized. We then trimmed our one hour records to 60 second records. Records were once again centered around the computed S-arrival.

At the present moment we avoided manually picking S-Phase arrivals by calculating theoretical arrival times based on a 1D spherically symmetric background and the IASP91 1D velocity model. This is only an approximation, however, so it is prone to errors. Thus, it might be worthwhile in the future to manually pick this data set.

The data set was once again split 80:20; 80% of the data was reserved for training and 20% of the data was set aside as the validation set. This left us with approximately 670 and 160 events for the training and validation set, respectively. Just as before our input, **x**, consists of 4Dof data and its associated label, **y**, are the S-phase arrival times.

5.5.3 Results from Transfer Learning

Looking at the learning curves (Figure 5.5) suggests that the transfer learning model is learning with experience. The red curve corresponds to the loss from looking at the training set, the blue curve is the loss from the validation set. The learning curve is demonstrating that the convolutional neural network, which is using a convolutional base originally trained on synthetic data, is improving (as a function of epoch) making S-Phase predictions.

Note that we used several more (orders of magnitude) epochs to train the model. The fact that we need more epochs makes sense considering that characteristics of real time-series data is more complex than the idealistic synthetic waveforms created using oversimplifications of the real Earth. In addition, aside from the vertical rotation component, the synthetics do not possess characteristic background noise(ambient, instrumental, etc.) that would be picked up from a true sensor. All this combined makes the task of learning from real Wettzell data more difficult.

To minimize the chance of overfitting, we stopped the machine learning algorithm from continuing to train once the loss curve reached a plateau. In this case, we stopped training around the 2400th epoch. To automate termination we will implement Early Stopping in the future. Early Stopping is a method that causes the model to terminate training once it stops improving its performance (e.g. the loss curve no longer decreases) on the validation set. Doing so will help us reach the model's peak performance by helping us automatically avoid underfitting and overfitting (see Section 2.4.2).

Note that the overall behavior of training and validation loss curves are nearly identical. This is a characteristic currently not well understood. It could be an artifact of the cost function used (in this case mean squared error). When we trained the model using a mean square logarithmic loss function (not shown here) we noticed oscillatory behavior of the validation for the first 400 epochs before it converged to the asymptotic declining behaviour of the training curve. This erratic behaviour could be explained by the fact that the logarithmic mean squared error is less sensitive to outliers. In other words, it doesn't punish predictions with large errors as severely as say, the mean squared error. With that said, using a logarithmic loss function displays a more typical loss curve behavior in that there is a larger error when making a prediction on the validation set than on the training data set. We will revisit cost functions again once we have a larger and unique data set.

The preliminary findings of applying transfer learning on data collected from the Wettzell Observatory hint there is potential in using Machine Learning to accurately pick S-Phases. There are still several tests to perform for both the model trained on synthetics and on the transfer learning model. We will go into these details in the next section.

5.6 Discussion & Further Work

The behaviour of the learning curves and the near 0 differential S-phase arrival times between the true S-phase arrival time and those predicted by the machine learning algorithm for both the training and transfer model are encouraging results.

To further push our neural network forward we will need to generate a larger and more diverse data set. As mentioned in Section 5.4.1 we believe our synthetic data set might have repeated events. Currently, events are defined using Numpy's Random function, which is not really random but rather, pseudo-random. Pseudo-random generators work by starting off with a seed value which is then fed back to the function the next time it is called[48]. To avoid repeated integers one can specify different seed values. When we generated our data we did not do this. Meaning there is a chance we had repeated synthetic waveforms. Thus, the diversity between our training and validation set could have been minimal, thus explaining why the machine learning algorithm had nearly identical training and validation loss curves.

In addition to the possibility of lacking diversity in our synthetic data set we also ran into long computation times when generating our synthetic waveforms. The inability to generate a large enough data set for training was the biggest setback we encountered. Our current method of generating synthetic data is not efficient. It can take as long as two days in order to make 1,000 synthetic events. Thus, as a first step, we will optimize our code to generate synthetics with shorter computation times. A machine learning algorithm is only as good as the data you feed it, and our case we were limited to a data set which was hindered by a lack of real teleseismic events.

Once the above criteria has been met we will be able to explore the limitations of our neural network. To begin, we would like to include varying levels of noise measured by the G-Ring and the collocated STS-2 to the synthetic waveforms. In doing so we can explore how accurate the neural network is at picking S-Phases as a function of the signal-to-noise ratio. We would hope to see the S-phase picker is robust (at least to a certain degree) to varying levels of noise. Therefore demonstrating its capability to make accurate predictions on station recordings with poor SNR.

Along those same lines we would like to investigate how the neural network performs on events with varying epicentral distance from the station. Given that the separation of the P-Phase and the S-Phase are more discernible as a function of distance, it may be that our neural network has an optimal minimum distance to which it can accurately pick an S-Phase. In which case, it might be more beneficial to have a probabilistic output describing the confidence level of the machine-learning pick.

Varying where the window is centered could make our neural network more robust for two reasons: first, shifting where the window is centered would artificially expand our data set thus giving us more data to work with; second the task of identifying S-Phases more difficult would be less trival. Currently the algorithm is given waveforms where the computed S-Phase is directly in the middle of time window. In doing so this could potentially bias how the neural network is learning to pick S-Phases.

As a quality control check it would be instructive to convert the algorithm from picking S-phases to picking P-phases. From the human point of view it is easier to identify changes in the phase and amplitude of the P-phase than the S-phase. Even if we used a zoomed-in 60-second window it will still be easier to identify said changes because we do not have remnants of the P-coda or reflected phases. This observation was made when we used data from Wettzell (see Figure 5.6 and Figure 5.7. It could be that the SNR of some of the events (including the ones shown here) was poor and not optimal for phase-picking. The data set used for the transfer learning model was a first iteration. We were more concerned about keeping as many waveforms for training than we were about being selective of which waveforms were suited for training.

Assuming we are able to train a robust neural network with the above mentioned updates, there are is room for improvement worthy of exploration with regards to the Wettzell data. Aside from having a larger data set to work with, it would be beneficial to manually sift through the collected data and remove outliers. Given that the size of the training data set is rather small in the context of Deep Learning, the presence of signals with poor SNR and/or events that are inconsistent with plane wave assumptions should be kept at a minimum so as to not influence the outcome of training.

If we are unable to generate a larger and unique data set we could alternatively train on teleseismic events recorded by one of the data centers part of the California Integrated Seismic Network (CISN). We would have orders of magnitude more teleseismic data to work given the number of stations in California to choose from and the recording duration (some stations span more than two decades). However, as of writing this thesis, there does not exist a network of stations with inertial seismic instrumentation and a collocated rotational sensor. Since we want to exploit the insight rotational data can provide, we would have to compute theoretical rotation rates for each event. The feasibility of this route is worth investigating given that the trade-off would be a significantly larger data set to work with. Which in turn would remove the issue of our inefficient code scheme and all the pre-processing required to make the synthetic data behave characteristically like real data (e.g. downloading and adding noise from the Wettzell Observatory and adding this to the synthetics).

5.7 Conclusion

The purpose of this Master Thesis project was to train a machine learning model that can accurately pick S-Phases using 4Dof data. This was motivated from both the theoretical and observational findings that including rotational observations (in this case the vertical rotation rate) in conjunction with a collocated inertial sensor (e.g. broadband seismometer) can improve back azimuth estimate and help separate different S-phases using a single station. Improving either of the aforementioned source properties can lead to more accurate and computationally less expensive, single station source location approximations. However, there are several challenges confronting 4Dof before we can exploit improved azimuth and S-Phase detection from the extra observational component. Challenges include (but are not limited to): noise, rotational measurements contaminating translational measurements that can not be corrected given the current sensitivity of collocated instrumentation, and features unique to rotational data that are not yet well understood.

To break this problem down, we focused on developing a neural network which can accurately pick S-Phases using 4Dof data. Given the limited size of the data to train with from Wettzell Observatory, we shifted our focus to training on synthetic data generated by a simple 1D velocity model. Several model architectures and hyperparameters where tested before we settled on a configuration that looked most promising. With this model we were able to achieve S-Phase picks with less than 0.5 second accuracy between the theoretical S-Phase arrival time and the pick made by the neural network. Satisfied with our results we then used this pre-trained model on teleseismic events recorded from the Wettzell Observatory via a technique known as transfer learning. Early tests suggest there is a possibility that training on synthetic data first, then applying transfer learning, might be sufficient to accurately pick S-Phase arrival times on 4Dof data recorded at the Wettzell Observatory.

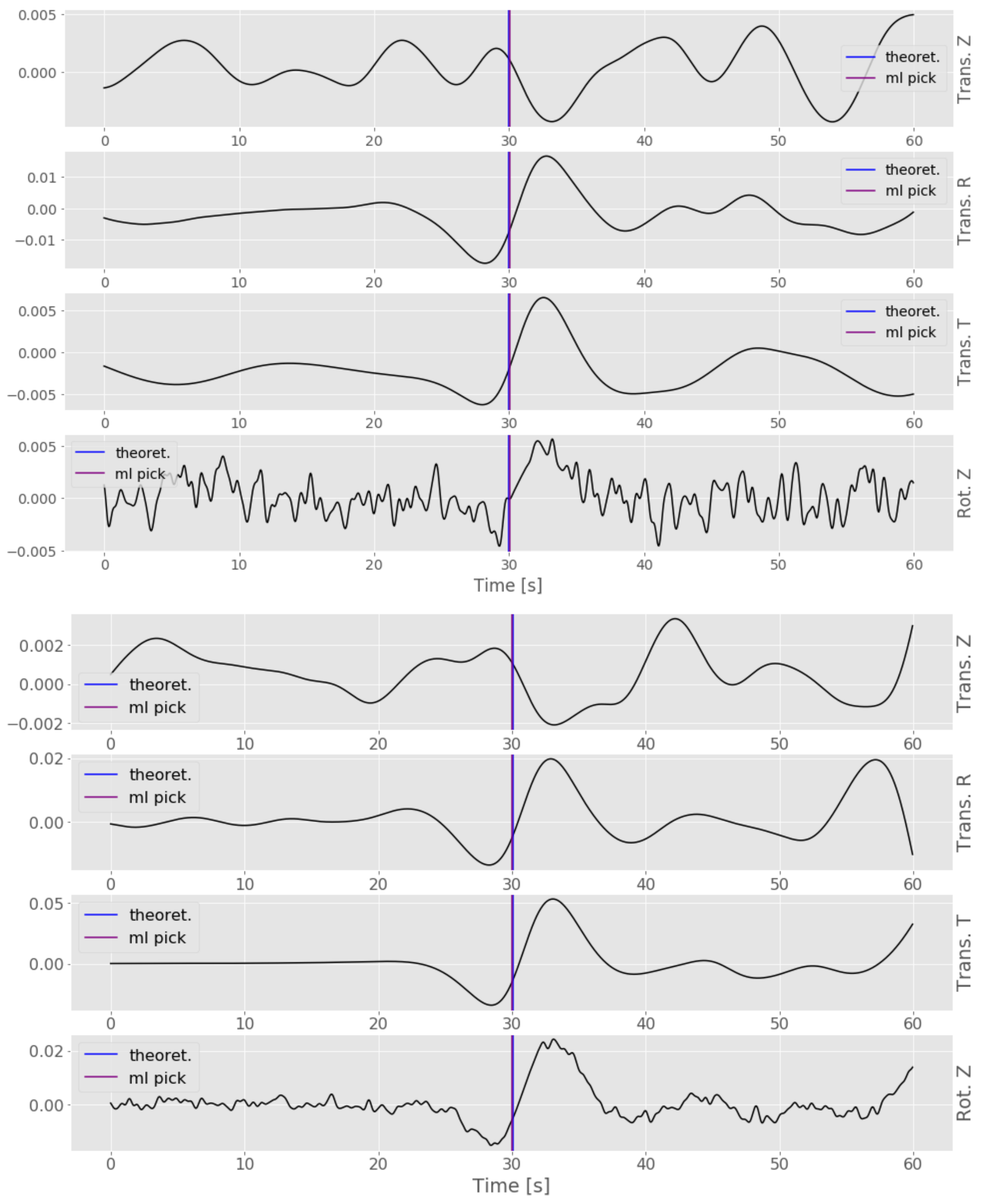

Figure 5.3: Input waveforms and their associated S-phase picks made by the machine learning algorithm (purple) and the true theoretical (blue) picks using 4Dof synthetic waveforms. Translational components were generated using Instaseis. The vertical rotation rate was derived using array-derived rotations. Noise from the Wettzell sensor was added to the rotational vertical component. 60 second window inputs are currently centered around the S-wave.

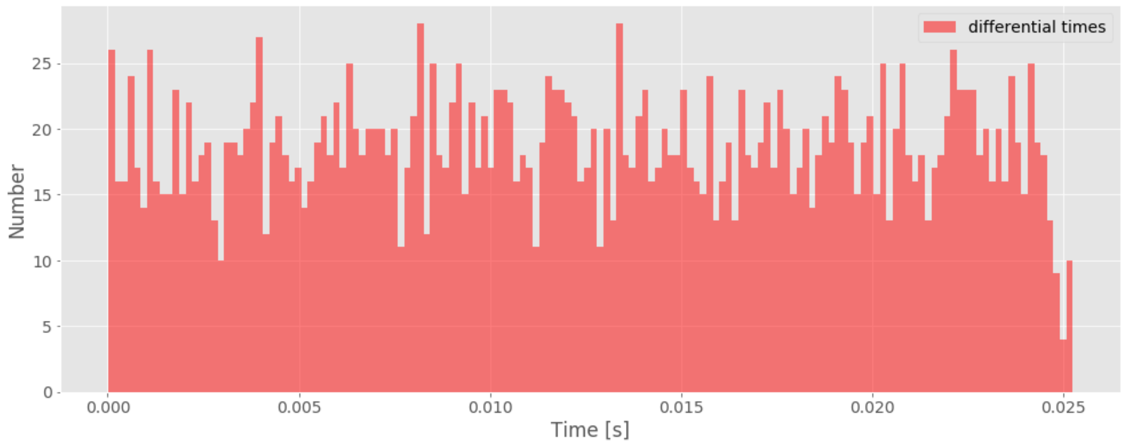

Figure 5.4: Absolute arrival time differences between machine learning and computed S-phase picks.

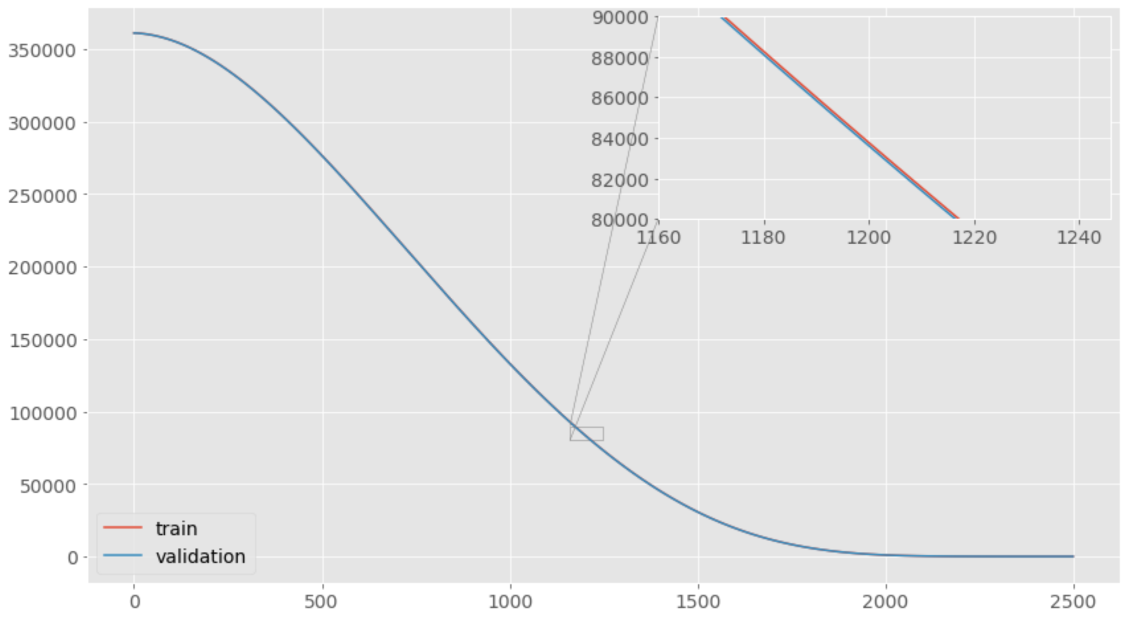

Figure 5.5: Learning curve using a mean squared error loss function using transfer model on 4Dof waveforms from the Wettzell Observatory. Zoomed-in plot displayed (upper right) to show the nearly identical behavior of both curves.

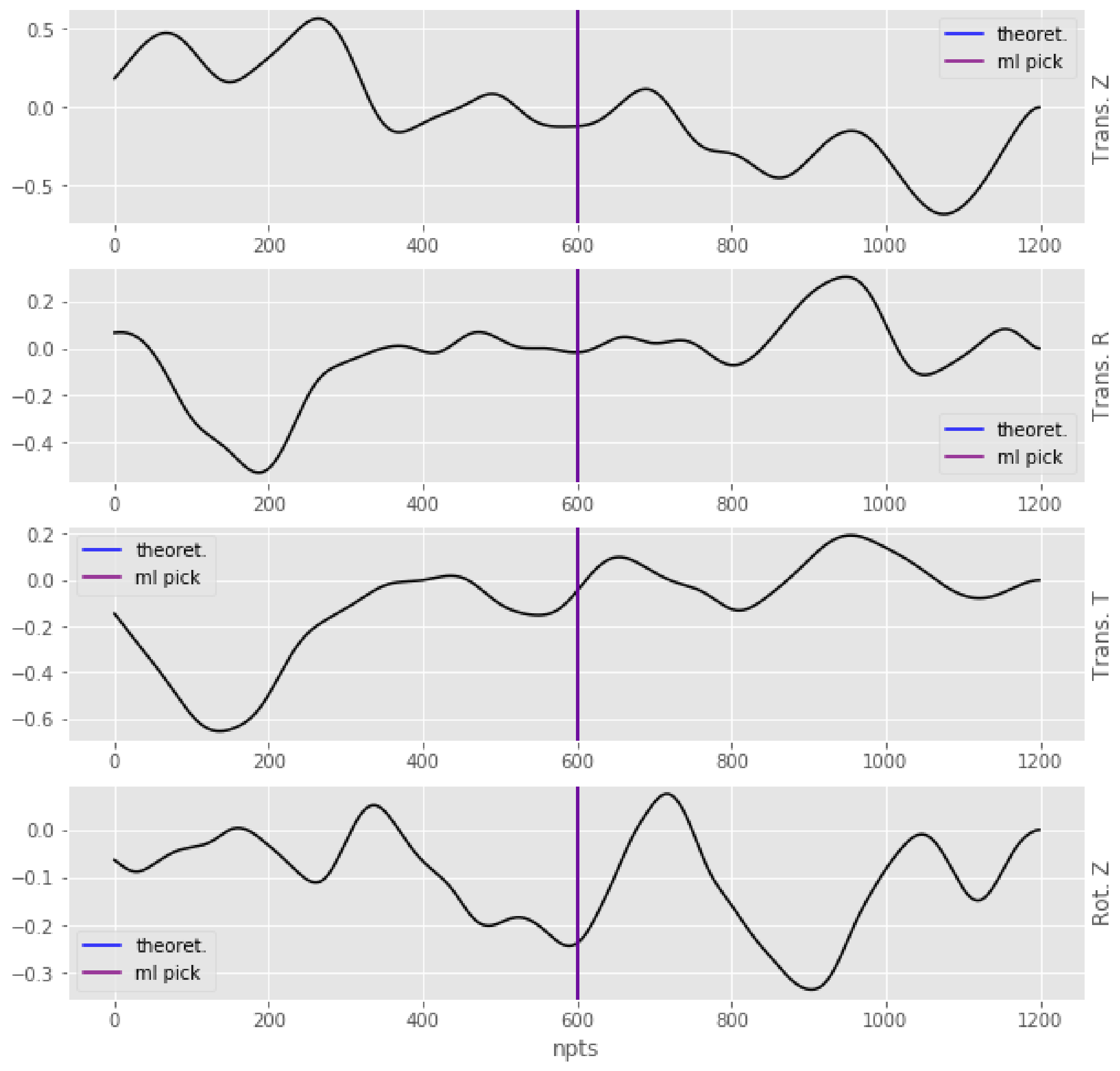

Figure 5.6: Input waveforms and their associated S-phase picks made by the machine learning algorithm (purple) and the true theoretical (blue) picks using 4Dof teleseismic events from the G-Ring and collocated STS-2 seismometer. 60 second window inputs are currently centered around the S-wave.

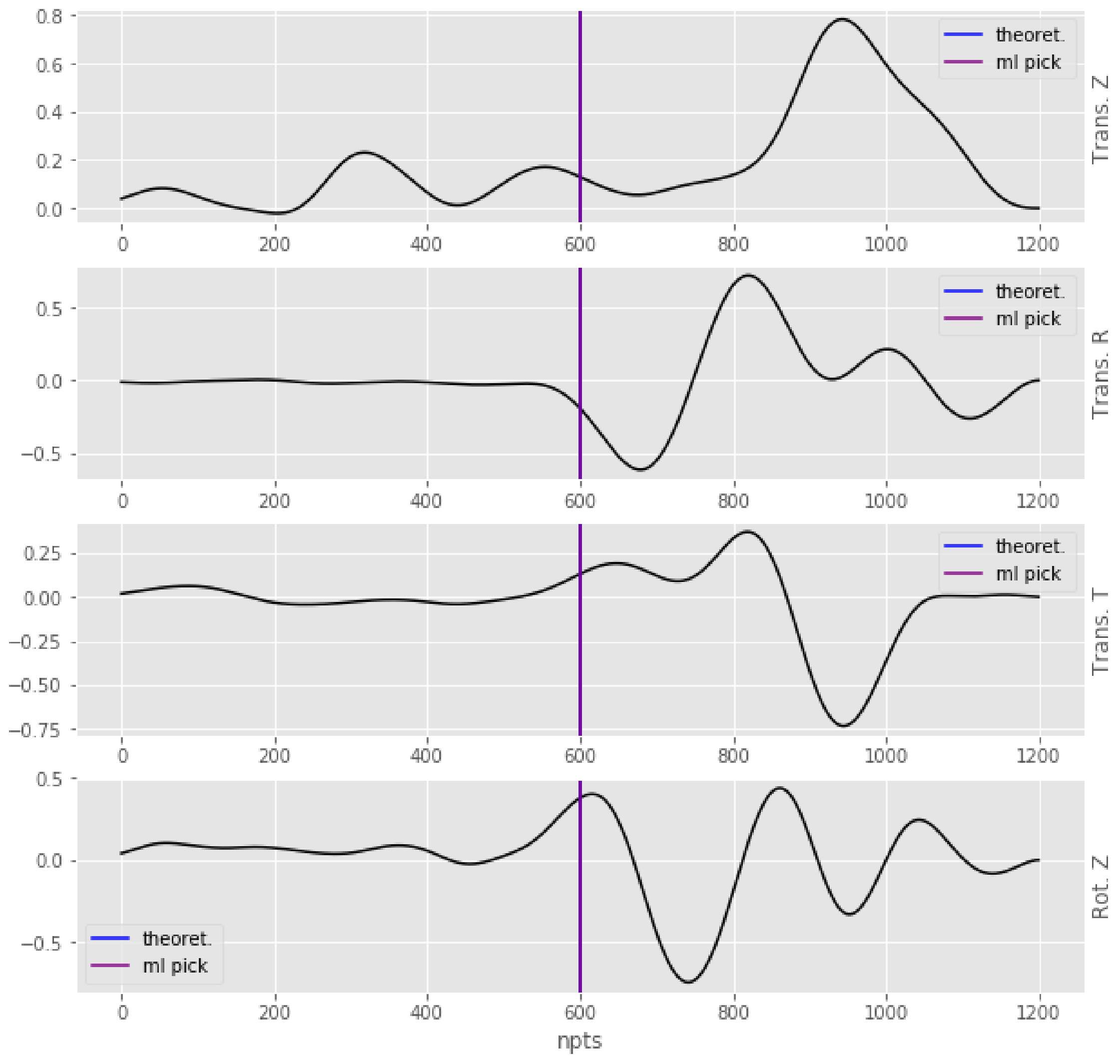

Figure 5.7: Input waveforms and their associated S-phase picks made by the machine learning algorithm (purple) and the true theoretical (blue) picks using 4Dof teleseismic events from the G-Ring and collocated STS-2 seismometer. 60 second window inputs are currently centered around the S-wave.

Bibliography

[1] *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way.* `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`. Accessed: 07.27.2019.

[2] *A Comprehensive Hands-on Guide to Transfer Learning with Real-World Applications in Deep Learning.* `https://towardsdatascience.com/a-comprehensive-hands-on-guide-to-transfer-learning-with-real-world-applications-in-deep-learning-212bf3b2f27a`. Accessed: 08.04.2019.

[3] Yaser S. Abu-Nostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learn From Data: A Short Course.* AMLbook, 2012.

[4] Charu C Aggarwal. *Neural Networks and Deep Learning.* Springer, 2018.

[5] Keiiti Aki and Paul G. Richards. *Quantitative Seismology.* 2nd ed. University Science Books, 2002.

[6] Guillaume Alain and Yoshua Bengio. "What Regularized Auto-Encoders Learn from the Data Generating Distribution". In: *1st International Conference on Learning Representations, ICLR 2013* (2013).

[7] Shoko Araki et al. "EXPLORING MULTI-CHANNEL FEATURES FOR DENOISING-AUTOENCODER-BASED SPEECH ENHANCEMENT". In: *IEEE* (2015), pp. 116–120.

[8] *Berkeley Artificial Intelligence Research: 1000x Faster Data Augmentation.* `https://bair.berkeley.edu/blog/2019/06/07/data_aug/`. Accessed: 07.28.2019.

[9] M. Böse et al. "A probabilistic framework for single-station location of seismicity on Earth and Mars". In: *Physics of the Earth and Planetary Interiors* 262 (2017), pp. 48–65.

[10] Francois Chollet. *Deep Learning with Python.* Manning Publications Co., 2018.

[11] Alain Cochard et al. "Rotational motions in seismology: theory, observation, simulation". In: *Earthquake source asymmetry, structural media and rotation effects* (2006), pp. 391–411.

[12] H. Philip Crotwell, Thomas J. Owens, and Jeroen Ritsema. "The TauP Toolkit: Flexible Seismic Travel-time and Ray-path Utilities". In: *Seismological Research Letters* 70 (1999), pp. 154–160. DOI: `https://doi.org/10.1785/gssrl.70.2.154`.

[13] Stefanie Donner et al. "The Case for 6C Ground Motion Observations in Planetary Seismology". unpublished. 2019.

[14] Martin van Driel et al. “Instaseis: instant global seismograms based on a broadband waveform database”. In: *Solid Earth* (2015).

[15] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques for Building Intelligent Systems.* O’Reilly UK Ltd, 2017.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* The MIT Press, 2016.

[17] Celine Hadziioannou and Peter Gaebler abd Ulrich Schreiber abd Joachim Wassermann abd Heiner Igel. “Examining ambient noise using colocated measurements of rotational and translational motion”. In: *Journal of Seismology* 16 (2012), pp. 787–796.

[18] Amir Hossein and Alavia Amir Hossein Gandomib. “Prediction of principal ground-motion parameters using a hybrid method coupling artificial neural networks and simulated annealing”. In: *Computers and Structures* 89 (2011), pp. 2176–2194.

[19] *How to Use Metrics for Deep Learning with Keras in Python.* `https://machinelearningmastery.com/custom-metrics-deep-learning-keras-python/`. Accessed: 06.29.2019.

[20] Heiner Igel et al. “Broad-band observations of earthquake-induced rotational ground motions”. In: *Geophysical Journal International* 168 (2007), pp. 182–196.

[21] Heiner Igel et al. “Rotational motions induced by the M8.1 Tokachi-Oki earthquake”. In: *Geophysical Research Letters* 32 (2005), pp. 3553–3556.

[22] Heiner Igel et al. “Seismology, Rotational, Complexity”. In: *Earthquake source asymmetry, structural media and rotation effects* (2015).

[23] *Instaseis: Instant Global Seismograms Based on a Broadband Waveform Database.* `http://instaseis.net/`. Accessed: 06.01.2019.

[24] Gareth James et al. *An Introduction to Statistical Learning with Applications in R.* Springer, 2017.

[25] Mike Kayser and Victor Zhong. “Denoising Convolutional Autoencoders for Noisy Speech Recognition”. In: ().

[26] *Keras.* `https://keras.io/`. Accessed: 07.29.2019.

[27] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *3rd International Conference for Learning Representations* (2018).

[28] Qingkai Kong, Richard Allen, and Louis Schreier. “MyShake: Initial observations from a global smartphone seismic network”. In: *Geophysical Research Letters* 43 (2016), pp. 9588–9594.

[29] Miriam Kristekova et al. “Misfit Criteria for Quantitative Comparison of Seismograms”. In: *Bulletin of the Seismological Society of America* (2006).

[30] Dieter Kurrle et al. “Can we estimate local Love wave dispersion properties from collocated amplitude measurements of translations and rotations”. In: *Geophysical Research Letters* 37 (2010).

[31] Thorne Lay and Terry C. Wallace. *Modern Global Seismology.* Academic Press, 1995.

[32] Aasha Pancha et al. “Ring laser detection of rotations from teleseismic waves”. In: *Geophysical Research Letters* 27 (2000), pp. 3553–3556.

[33] Thibaut Perol, Michaël Gharbi, and Marine Denolle. “Convolutional neural network for earthquake detection and location”. In: *Science Advances* 4 (2018).

[34] K. B. Richards-Dinger and P. M. Shearer. “Earthquake locations in southern California obtained using source-specific station terms”. In: *JGR Solid Earth* 105 (), pp. 10939–10960.

[35] Alan Richardson and Caelen Feller. “Seismic data denoising and deblending using deep learning”. In: (2019).

[36] Zachary E. Ross, Men-Andrin Meier, and Egill Hauksson. “P-wave arrival picking and first-motion polarity determination with deep learning”. In: *JGR Solid Earth* 123 (2018), pp. 5120–5129.

[37] Zachary E. Ross et al. “Generalized Seismic Phase Detection with Deep Learning”. In: *Bulletin of the Seismological Society of America* 15 (2014), pp. 1929–1958.

[38] Michael L. Seltzer, Dong Yu, and Yongqiang Wang. “An investigation of deep neural networks for noise robust speech recognition”. In: *IEEE* (2013), pp. 7398–7402.

[39] Peter M. Shearer. “Improving local earthquake locations using the L1 norm and waveform cross correlation: Application to the Whittier Narrows, California, aftershock sequence”. In: *JGR Solid Earth* 102 (), pp. 8269–8283.

[40] David Sollberger et al. “Single-component elastic wavefield separation at the free surface using source and receiver-side gradients”. In: *SEG International Exposition and 87th Annual Meeting* (2016), pp. 2268–2273.

[41] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958.

[42] Seth Stein and Michael Wysession. *An Introduction to Seismology, Earthquakes, and Earth Structure.* Blackwell Publishing, 2003.

[43] Wiwit Suryanto Suryanto et al. “First Comparison of Array-Derived Rotational Ground Motions with Direct Ring Laser Measurements”. In: *Bulletin of the Seismological Society of America* (2006).

[44] *UFLDL Tutorial: Convolutional Neural Network.* `http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/`. Accessed: 07.27.2019.

[45] Pascal Vincent et al. “Extracting and Composing Robust Features with Denoising Autoencoders”. In: *Proceedings of International Conference on Machine Learning* (2008), pp. 1096–1103.

[46] Aaron G. Wech and Kenneth C. Creager. “Automated detection and location of Cascadia tremor”. In: *Geophysical Research Letters* 35 (2008).

[47] *Which Deep Learning Framework is Growing Fastest?* `https://towardsdatascience.com/which-deep-learning-framework-is-growing-fastest-3f77f14aa318`. Accessed: 07.30.2019.

[48] John Zelle. *Python Programming: an Introduction to Computer Science.* Franklin, Beedle & Associates INC., 2009.

[49] Li-ping Zhang et al. "Seismic Signal Denoising and Decomposition Using Deep Neural Networks". In: *International Journal of Petrochemical Science & Engineering* (2017).

[50] Lingchen Zhu, Entao Liu, and James H. McClellan. "Seismic data denoising through multiscale and sparsity-promoting dictionary learning". In: *Geophysics* 80 (2015).

[51] Weiqiang Zhu and Gregory C. Beroza. "PhaseNet: A Deep-Neural-Network-Based Seismic Arrival Time Picking Method". In: *Geophysics Journal International* 216 (2018), pp. 261–273.

[52] Weiqiang Zhu, S. Mostafa Mousavi, and Gregory C. Beroza. "Seismic Signal Denoising and Decomposition Using Deep Neural Networks". In: (2018).