Machine Learning Engineer Nanodegree

Capstone Project – Predicting earthquakes

Jan Nowakowski
March 20th, 2019

I. Definition

Project Overview

Forecasting natural disasters is a critical part of the field of earth science, due to the impact of early alarming on decreasing the toll of such a catastrophe, both with regards to human life, as well as economic loss. One of the most deadly, dangerous and omnipresent disasters are earthquakes. Therefore, I chose to participate in the Kaggle competition that strives to improve the forecasting of when an earthquake will occur, based on real-time seismic data.

The Kaggle competition can be viewed here:
https://www.kaggle.com/c/LANL-Earthquake-Prediction

The datasets and inputs of this Kaggle competition are described here:
https://www.kaggle.com/c/LANL-Earthquake-Prediction/data

To summarize, the train dataset is a single, continuous training segment of experimental data and the test dataset is a set of many small segments of a longer experiment.
The training data contains two fields: acoustic_data, which is the seismic signal, and time_to_failure, which is the time in seconds until the next failure (i.e. Earthquake). The test data naturally has only the first of the two fields. All of the data used in this competition has been generated by a laboratory setup described in [1].

Problem Statement

The goal of the competition is to use seismic signals in order to predict when an earthquake will occur. After the initial exploration of the data, I will design and train different neural networks in order to solve the problem. I plan to use two different neural network types: Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), the latter being a canonical way of treating time-series data. They has been shown to be successful in many different applications including, but not limited to Natural Language Processing [2], financial forecasting [3] and anomaly detection [4].

I will also use two approaches when it comes to input data. The first naive approach will be to feed the raw data (the seismic signal over time) the ML algorithms. This way, the algorithm will have to learn from scratch, what to look at when predicting the time to the next earthquake. It might therefore be better (or at least learning might be quicker) if the ML algorithm is fed processed data. For example, with decreasing time to next earthquake the seismic data becomes more noisy; therefore, it seems very reasonable to calculate the standard deviation of seismic data slices and feed that to the algorithm. I will do that in the second approach.

Metrics

In this project I used the metric required by the competition, i.e. the mean absolute error between the predicted and real time to the next earthquake. It is the average of the difference between the real and the predicted values, as shown in the equation below:

$$MAE = \frac{\sum_{i=1}^{n} |x_i - y_i|}{n},$$

where x_i and y_i are the real and predicted value of the ith point, and n is the sample size.

The choice of this metric is reasonable, as we would like to predict the time_to_failure as well as possible. Personally, however, I would choose a different metric – e.g. mean squared error or R^2, as I would like to have as few large discrepancies between the real and the predicted values, and not, for example, be extremely exact in half of the cases and very wrong in the other half.

II. Analysis

Data Exploration

The training data is a long measurement (~630 million datapoints) with 16 earthquakes in it. It is in csv format, with two columns, as the sample shown below:

acoustic_data	time_to_failure
4.0	1.469100
3.0	1.469100
5.0	1.469100
9.0	1.469100
4.0	1.469100
7.0	1.469100
9.0	1.469099
4.0	1.469099
7.0	1.469099
8.0	1.469099
6.0	1.469099
6.0	1.469099

The points are, however, not equally spaced. Sometimes there are ~1000 datapoints with exactly the same time_to_failure, and then it will suddenly jump by 0.001s. In end result, there are 4e6 datapoints per second. The values of time_to_failure vary between 0 and ~16 seconds, whereas the acoustic data values are oscillating between -25 and 25 during the most quiet time, increasing quite steadilly as an earthquake approached,and right before time_to_failure reaches 0 (i.e. the earthquake happens), the acoustic_data values can even reach even ±4000. Hence, choosing standard deviation or peak-to-peak value seems like a reasonable feature to feed to the ML algorithm. More statistics are provided in the next paragraph.

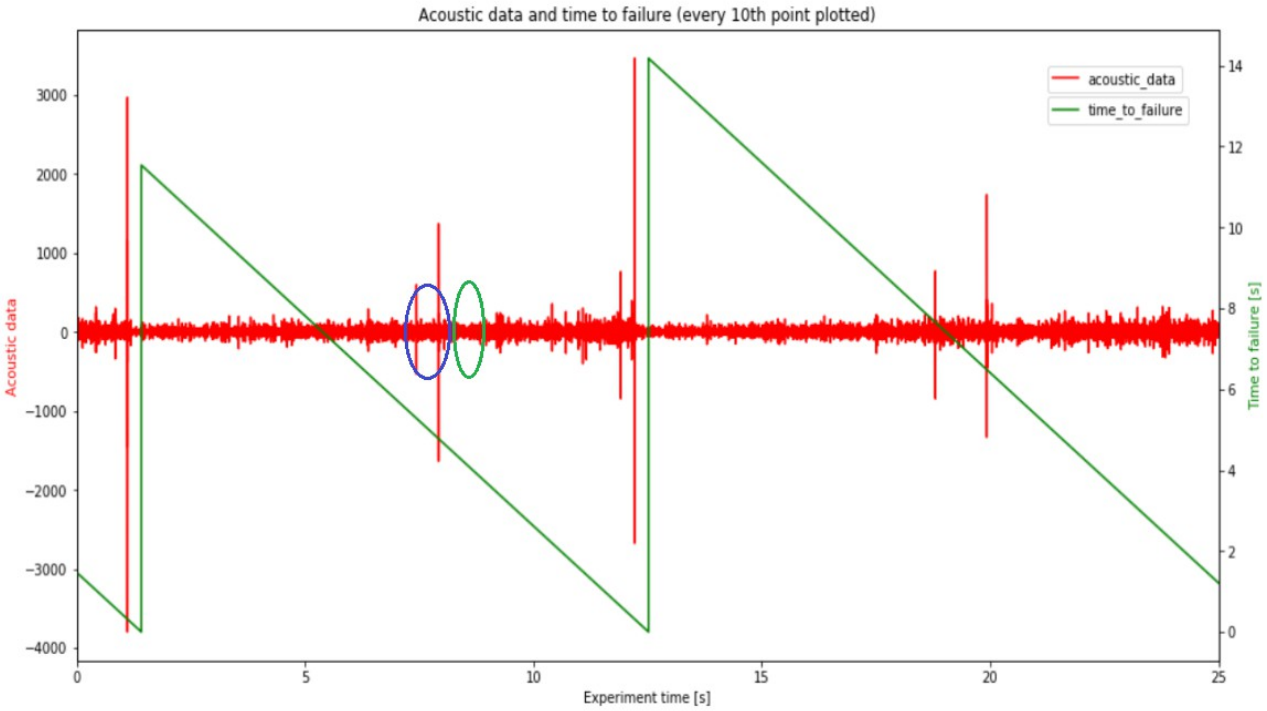
The testing data, on the other hand, is a set of shorter, equal-length (150000 datapoints) measurements, with only the acoustic_data column present. The task of the algorithm will be to predict what the time_to_failure value next to the last datapoint in the slice would be (no need to predict it for all the datapoints).

Due to the discrepancy in how the training and testing data is structured, the simplest way to obtain a large training set is to split the training file into many slices of the same size as the test data. Each input file is just a time-series seismic data – so one of the most important tasks will be to figure out what are the features that are most important for predicting the correct time to the next earthquake.

Seismic data describes how significantly the setup shakes, i.e. it is a measure of the displacement of the experimental setup (it could also be the velocity or acceleration, it is not clearly stated in the description of the Kaggle competition). Right after an earthquake (the most quiet time) the intensity varies between -100 and 100. During an earthquake, however, it varies between – 4000 and 4000. Time to failure (naturally only given in the training data) is the time in seconds to the next earthquake. Its values vary between 0 and 16 seconds. This is what we are trying to predict.

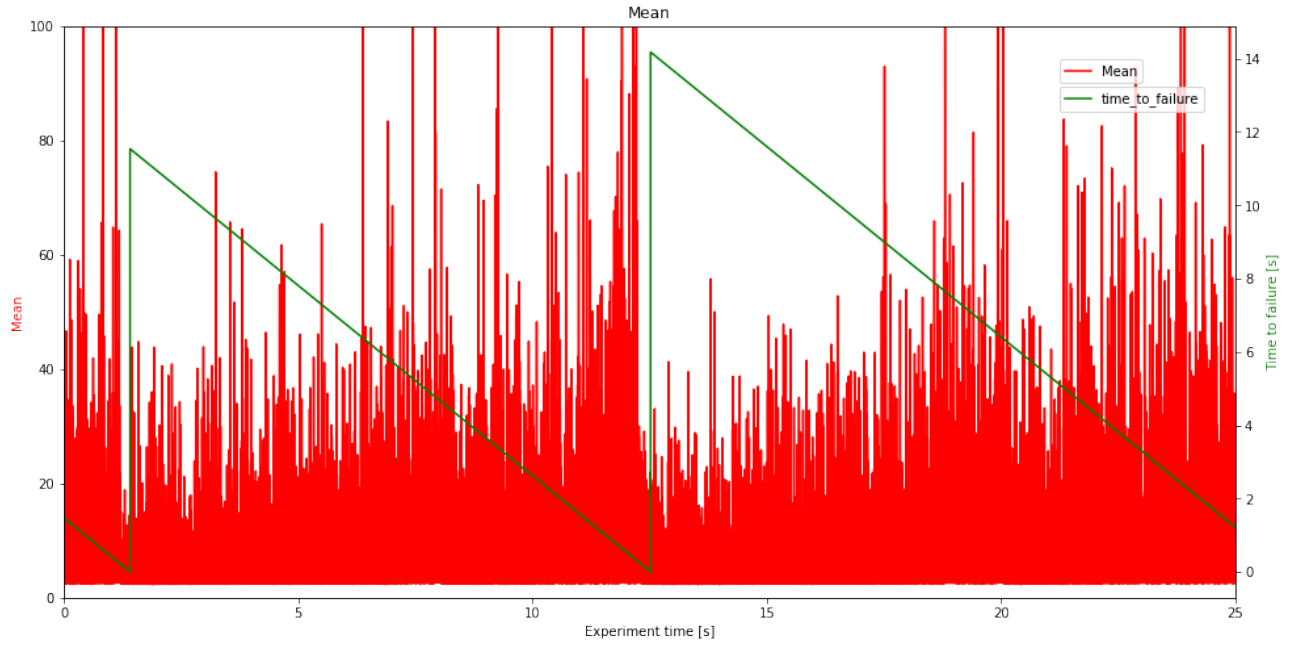
Exploratory Visualization

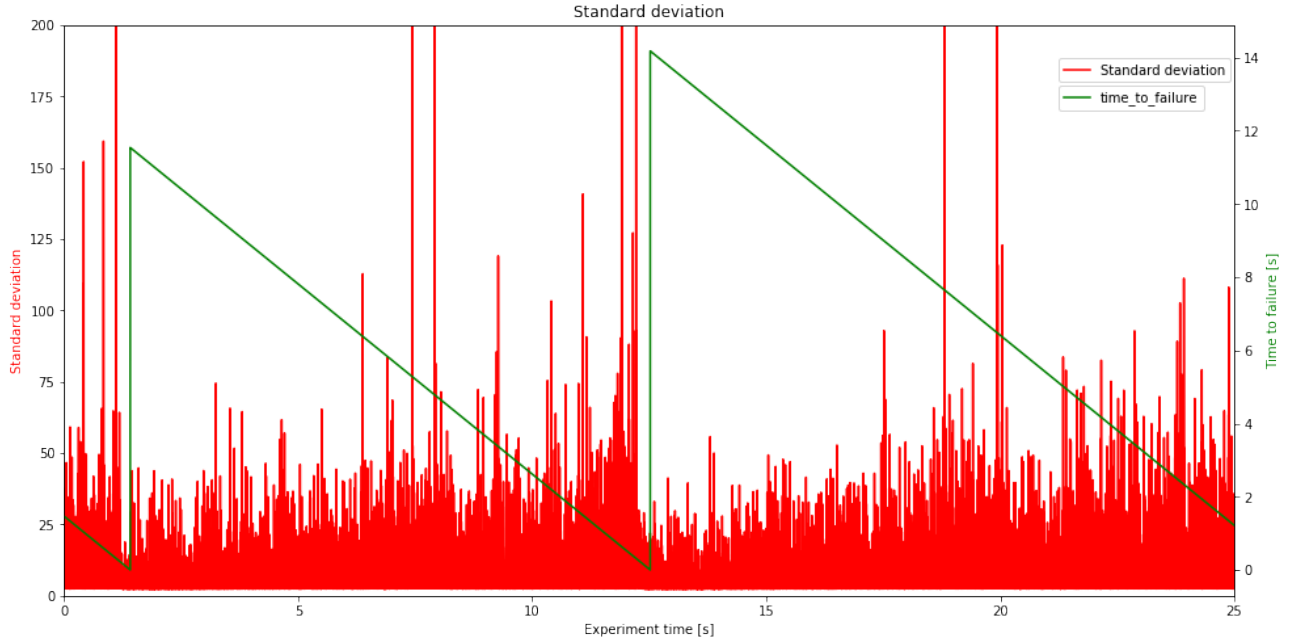
Below roughly 1/6th of the training data is shown - first 1e8 rows corresponding to 25 seconds of the experiment (data recorded at 4MHz). The green line shows the time_to_failure that starts at ~12 seconds and linearly goes to 0, and then immediately to 14 seconds, indicating that an earthquake has just occurred. The red line shows the seismic data. It becomes in general more noisy, the closer it is to the next earthquake. Unfortunately, this is not a very steady increase, as the data in the green ellipse is much less noisy than the data in the blue ellipse, even though it is closer to an earthquake by around a second. What adds to the difficulty of this project is the fact, that test data slices are very short, equivalent to 0.0375 seconds of experiment (that means, the data shown below is equivalent to ~667 training data slices.



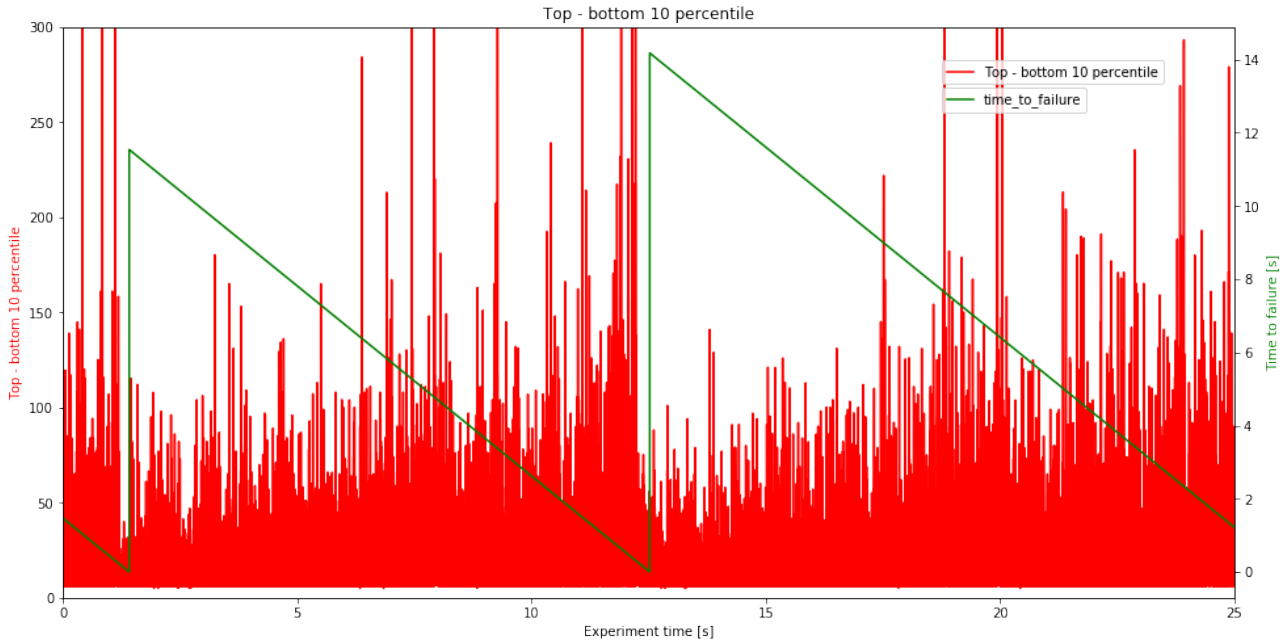
My first approach will be to just feed the ML algorithm the raw data, and make it figure out that noise is important in this case – so no feature engineering required in this case.

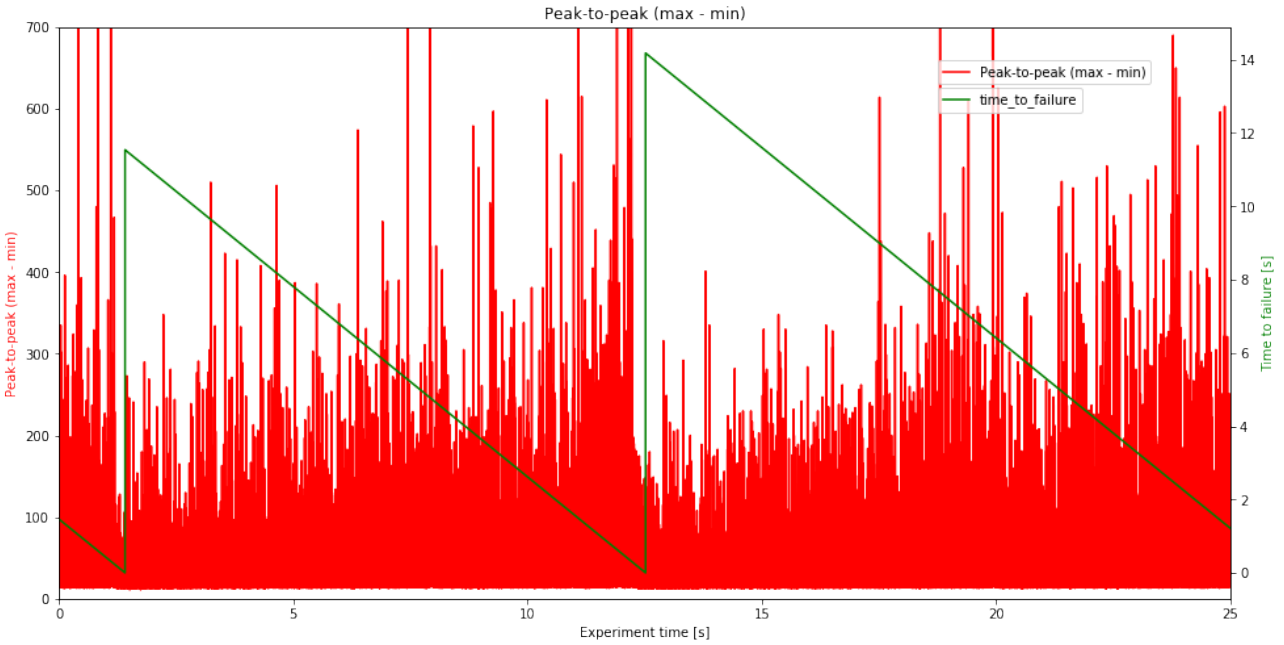
In the second approach I will try to feed statistical data based on one or more functions to the ML algorithm in the hope that it will make it learn faster. Out of all the statistical function I tried, the most promising one were the mean, standard deviation, top-bottom 1th percentile as well as peak-to-peak distance. Their graphs, based on 1000 datapoints long slices, can be found below. The choice of 1000 is not accidental, it gave the best results during training (I tried many different values between 100 and 5000). The statistics plotted below were calculated on the exact same data as shown on the previous page. I left the green line indicating the time_to_failure as a reference.





Interestingly, most of the graphs shown show very similar shapes, and unfortunately they, as expected from the seismic data, show only a general trend of increasing with decreasing time to the next earthquake.





The one I found by visual inspection to be the most promising is the peak-to-peak plotted right above, as in my opinion it shows the most clear trend. I will therefore focus mainly on that feature when preparing data for the algorithm. But naturally I will also try with different functions, as well as combining many to check if it provides better results.

Algorithms and Techniques

I will use deep learning to tackle this problem. After an online research I concluded that the two best candidates to achieve the best result are CNNs and RNNs – the latter being the most common way of treating time-series data. See literature [2-4] for examples where RNNs were successful.

Convolutional layers in CNNs have so-called filters – that is, matrices that are applied to the input data in a clearly defined way. The size of the filter is usually much smaller than the size of the input data, so the same filter is applied over multiple times over different parts of the input. The role of input layers is to spot distinct features – for example, in image recognition there would be one filter responsible for detecting horizontal edges, one for vertical edges etc. Here, my hope is that the convolutional layers will be able to recognize that many very steep peaks (in case of raw data input) or high and quickly increasing variance (in case of giving variance calculated over the raw data as input) are a sign of an incoming earthquake.

RNNs on the other hand, work in a completely different manner. They do not only take the current input into account – with the use of their internal state (one can think of it as 'memory') they can take into account what they have seen before. In other words, the RNNs have to different inputs – the current one, and the recent past. These two pieces of information are then used to determine their decision. In this specific case the desired conclusion of the RNN (for raw data input) would be that if the current and the past input do not differ by much, we are not in danger of an earthquake happening soon. If they do, however, oscillate very rapidly, it would understand that the disaster is iminent.

I will try both 1D and 2D (after some data wrangling) convolutional neural networks in the hopes that the 2D version will be able to learn faster. I plan to try 2D CNNs also because if I have enough time, I will try to use transfer learning to hopefully get better results (I did not find any pre-trained 1D CNNs). It might be interesting, despite most pre-trained networks being designed for image recognition and not for noisy spectral data. I will also try combining both RNNs and CNNs.

Benchmark

A basic feature benchmark has been given in the Kaggle competition, and can be viewed here: https://www.kaggle.com/inversion/basic-feature-benchmark

To summarize, it uses the mean, standard deviation, min and max of the data to a Support Vector Machines algorithm to predict the time_to_failure.

I will judge the quality of each model based on the training score, i.e. the Mean Absolute Error (mae) after submitting the predictions for the test data to Kaggle. As a reference, the basic feature benchmark receives the score of 1.881 (it is the error of the prediction, so lower means better). It is worth noting, that this benchmark is already very good, as random guessing would lead to results of ~8 (time_to failure that we predict is in the range of 0 to 16 seconds). This is supported by the fact, that at the time of writing this report, the 1st place contestant of the Kaggle competition has could only reach a ~25% lower error (1.362).

III. Methodology

Data Preprocessing

The training data and the testing data was provided in different sizes – the training data a one long measurement with >600'000'000 datapoints, whereas the test data was in the form of 2624 segments with 150'000 datapoints each. Therefore, in order to easily apply the trained algorithm to the testing data, I needed to cut the training data to be the same size as the testing data slices. As a side note, I also tried overlapping the cuts of the training data in order to re-use the data get more out of the dataset provided, but did not see any improvement with respect the cutting to non-overlapping slices.

Then, I employed two different approaches in the next step:

1. Feed the ML algorithm raw data. I tried with a 1D vector of 150000 datapoints, but quickly realised the algorithm learns much quicker if I use 2D CNNs. In order to be able to do that, I reshaped the data into a 375x400 2D array.
2. Feed the ML algorithm with the result of calculating a statistical function (e.g. mean, standard deviation, peak-to-peak, interquantile range or variance) over the dataset cut into even smaller segments. I tried segment-sizes between 100 and 5000, and the ones giving the best result were 1000. When it comes to the statistical functions, all of them performed similarly, but the one I had the best results with was the peak-to-peak calculation. I also tried using the result of many statistical functions at the same time, but it did not improve the results. It is not very surprising, as it is visible from the graphs plotted in the „Data Exploration" part, that they carry similar information.

Implementation

In this project I used Keras with Tensorflow as backend. I did not have to write any overly complicated functions in this project; the ones that are probably the most difficult to understand from reading them are the data-wrangling function in order to reshape the data appropriately. An example of such a function can be found below. It slices the >6e8 datapoints of the training data into 150000 long slices, and then re-shapes them into 375x400 2D array.

```
step = 150000 # size of train data
columns = 400 # 150000 = 375*400, this number of colums makes the
sets = math.floor(len(float_data)/step)
x_train = np.zeros(shape=(sets, int(step/columns), columns))
y_train = np.zeros(shape=(sets))
for i, each in enumerate(range(0, len(input_data), step)):
    if i == sets:
        break
        print(f'Reshaping set number {i+1}/{sets}', end='\r')
        x_train[i] = np.reshape([e[0] for e in float_data[each:each+step]], (-1, columns))
        y_train[i] = float_data[each+step-1][1]
        x_train = x_train.reshape(*x_train.shape, 1)
        assert x_train.shape == (sets, int(step/columns), columns, 1)
```

I used four different NN architechtures. A simple example of each of NN architecture is shown below:

1. Pure CNN. This architecture combines Conv2D and Pooling layers. It was trained using raw data (150000 points), reshaped into a 375x400 2D array.

```
model = Sequential()
input_shape=(None, 375, 400)
model.add(Conv2D(filters=8, kernel_size=4, strides=2, padding='same',
          activation='relu', input_shape=input_shape))
model.add(Dropout(0.1))
model.add(MaxPooling2D(pool_size=2, strides=2))
model.add(GlobalAveragePooling2D())
model.add(Dense(1, activation = 'relu'))
```

2. Mixing CNNs and RNNs, trained also with raw data reshaped into a 375x400 2D array. This implementation required some reshaping of the data in the model itself, in order to be able to feed an output of convolutional layers into recurrent layers (done with a „Reshape" layer)

```
model = Sequential()
filters = 32
input_shape=(None, *x_train.shape[1:])[1:]
model.add(Conv2D(filters=filters, kernel_size=8, strides=2, padding='same',
    activation='relu', input_shape=input_shape, name = 'conv'))
conv_shape = model.get_layer('conv').output_shape[1:]
model.add(Reshape((*conv_shape, 1)))
model.add(CuDNNGRU(int(filters/8), kernel_size=8))
model.add(GlobalAveragePooling2D())
model.add(Dense(int(filters/2), activation = 'relu'))
model.add(Dense(1, activation = 'relu'))
```

3. Mixing CNNs and RNNs, but this time fed with peak-to-peak data and not the raw data. This NN also required reshaping the data within the model.

```
model = Sequential()
cells = 64
input_shape=(None, 1)
model.add(CuDNNGRU(cells, input_shape=input_shape, name = 'rnn'))
rnn_shape = model.get_layer('rnn').output_shape[1:]
model.add(Reshape((*rnn_shape, 1)))
model.add(Conv1D(filters=int(cells/2), kernel_size=8, strides=2, padding='same',
    activation='relu'))
model.add(GlobalAveragePooling1D())
model.add(Dense(int(cells/4), activation = 'relu'))
model.add(Dense(1))
```

4. Pure RNN. I used the RNN layer optimized for GPU usage (CuDNNGRU). Notably, I also tried the same NN architecture with Long Short-Term Memory (LSTM) network, with very similar results. I chose the GRU network mainly because training it was much faster.

```
model = Sequential()
model.add(CuDNNGRU(128, input_shape=(None, 1)))
model.add(Dense(64, activation='relu'))
model.add(Dense(1))
```

The biggest trouble I had while implementing these neural networks was reshaping the data into the exact shape the layers require. That was especially the case for models 2 and 3 – where apart from the initial reshaping of the input data, there was an additional reshaping done within the NN. It had to be performed in order to be able to use both convolutional and recurrent layers in one network – as they both require significantly different input sizes.

Luckilly, the best performing model number 4 did not cause any problems with the data reshaping. The only data preparation steps required was to take the 150000 datapoints long data segment, calculate the peak-to-peak difference of 1000 datapoints long slices within them (resulting in 150 peak-to-peak values) and yield them to the RNN layer one-by-one using a python generator.

Refinement

Apart from modifying the algorithm's input data, as described before, during the optimization process I was modifying the number of layers, adding and removing dropout layers as well as changing number of cells (filters) in each layer, optimizers, learning rate and activations.

When it comes to activations, it is clear that the last layer needs to be able to return a double-digit decimal number. Hence, the last layer should have a linear one ('relu', to be more specific - it works great, especially since time_to_failure cannot be negative). I tried different ones for the previous layers (sigmoid, softmax), but they did not provide good results, and in the end I chose relu for all layers.

Instead of manually fiddling with the learning rate, I chose to use LearningRateScheduler callback provided by keras. After trying different scheduling functions, I settled on using the epoch number divided by 1000.

I tried many different optimizers (RMSprop, Adam, Nadam, Adadelta), and the one that worked best for me was Nadam.

With regards to the size of the model, most of the time I used a very simple design, in order to keep the time to train for one epoch small and try many different hyperparameters and data processing methods. From time to time I tried a more elaborate NN design and trained it for several hours, but the result was usually very similar, or sometimes even worse than using a simple design.

IV. Results

Free-Form Visualization

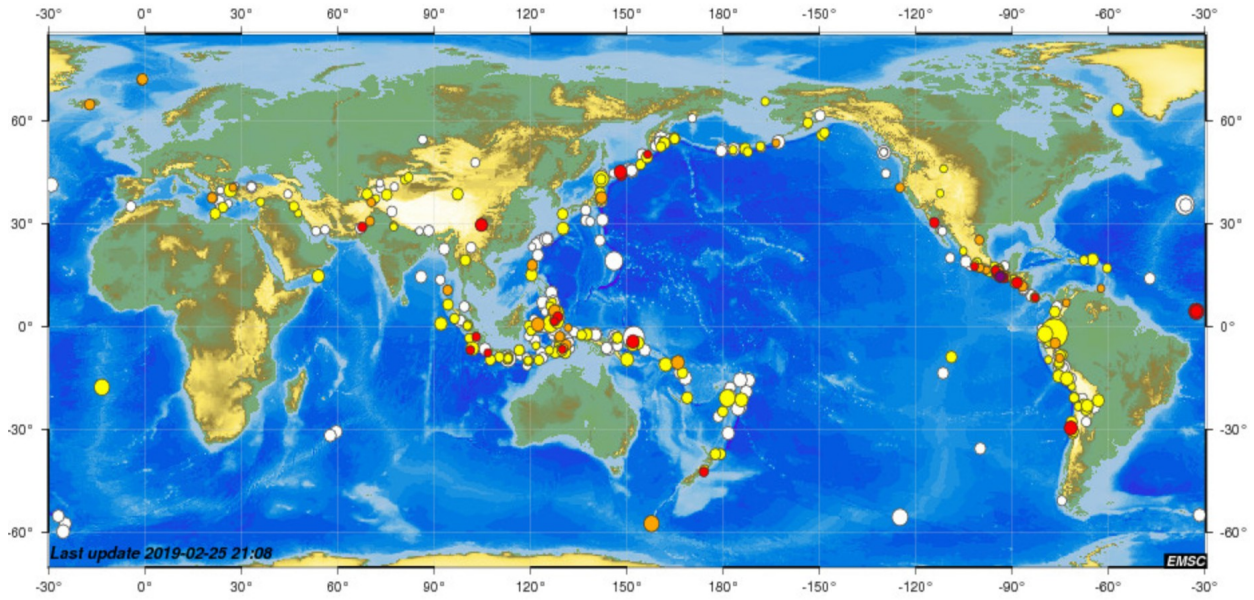


Image source: https://www.emsc-csem.org

The image above shows the locations where an earthquake of magnitude 4 or higher happened in the last two weeks (period 11.02-25.02.2019, circle size represents the magnitude). There are 474 dots on this map [5]. Magnitude 4 means an earthquake that is clearly felt by humans and can cause damage [6]. Even though not all of them happened in an inhabited area, it is still scary that this is such a common occurence.

The even scarier fact is, that in 15 year between 2000 and 2015, nearly 1 million people died due to earthquakes – most of which due to two earthquakes: the 2004 Indian Ocean earthquake and tsunami and the 2010 Haiti earthquake [7]. The death toll could have been significantly reduced, if only they were predicted a few days earlier. Therefore, the importance of early forecasting of not only earthquakes, but all natural disasters cannot be overstated – and this is the reason why I decided to participate in this Kaggle competition.

Model Evaluation and Validation

The neural network design that achieved the best results was the architecture number 3 – pure RNNs, that were given the peak-to-peak function calculated over 1000 data-points long slices of the data. It reached the test error of 1.518 – that places me, at the time of writing this report, in the top 35% of contestants of the Kaggle competition. (My Kaggle account can be found here: https://www.kaggle.com/lambda00). The feature benchmark model had an error of 1.881 and the best result on Kaggle at the time of writing this report is 1.362.

The best result of the other models was not very far behind. To my surprize, the simple approach of feeding the raw data to the CNN achieved an error of 1.723, CNN+RNN given the raw data achieved the result of 1.708, whereas the mix of CNN+RNN given the statistical data as input reached 1.605.

It is worth noting, that using those scores is not a very fair comparison, as I've spend by far the largest amount of time refining the pure RNN solution. It is very well possible, that I would have been able to achieve an even better result with the other two architectures if I've put more effort into

that. I realised, however, that the pure-RNN architecture provides the best results and learns the fastest out of the four and decided to focus more on fine-tuning that solution.

I also tested the robustness of the solution, by modifying the size of the network, statistical function used on the data, modifying batch size, changing the learning rate and also restarting the same model multiple times to check if it gives the same result. In each case, the resulting testing errors were within 0.1 from each other.

Justification

The error of the final model is 0.37 lower than the error of the already very good feature benchmark. It is also not significantly worse than the best solution currently on Kaggle (0.15 difference. And looking at the data, I would be very surprised if anyone was able to achieve a much better result than that. It is so because the data is very irregular, the increase in the variance of the seismic_data is not monotonous with respect to the time_to_failure and the testing data slices are incredibly short. Therefore, I am very happy with the result my model achieved.

It is worth noting that my model was in the end very simple, just calculating the peak-to-peak of 1000 datapoint slices (150 of them, as the testing data consisted of 150000 datapoints) and feeding them to a RNN that consisted of just three layers (1RNN and 2 Dense layers). I tried many much more complex solutions, but provided the same or even worse results with more time required to train.

It is difficult to tell if this solution „significant enough to have solved the problem". It is not clear if the results from the laboratory setup that were analyzed here translate well when applied to the real-world data. First of all, it is unclear to me if real measured seismic data is of similar form to the lab data. In addition, the timescales of a real earthquake are on a completely different timescale – here we had to predict number of seconds to the next earthquake. That would naturally be useless in a real-world application, as we would need forecasting at least hours if not days before the disaster occurs in order to be able to react and move people as far as possible from the epicenter of the earthquake.

In addition, apart from predicting when the earthquake will happen, the model should also predict the magnitude of the earthquake, as well as which regions are in danger. This project is a one small step in solving a much bigger problem.

V. Conclusion

Reflection

I started the project by looking at the data and trying to understand it, finding out what can be successfully used as input data for the algorithm and how to deal with such a huge dataset (CSV file with the training data is 9GBs!) Then I researched, using both Kaggle forums and other resources, the possible techniques and ML algorithm designs. Then I started training the four different NN designs mentioned earlier using similar settings and hyperparameters, and found out that pure RNNs worked the best for me. After that, I improved the results achieved by the chosen algorithm with steps described in the „Refinement“ paragraph.

The biggest difficulty in this project for me was that I could not get comparable mean absolute error between the training, validation and testingdatasets. A very good score on the training/validation set seemed not very well connected with how good the score will be on the test dataset. And unfortunately, because of how the Kaggle competition is structured (allows only 2 submissions per day to check the score on the test dataset), it is difficult to pinpoint the clear reason for it, apart from differences in the content and structure of the training and testing datasets. Browsing through the forums, it seems that all of the Kagglers have a similar problem. Perhaps the error depends too much on where during the earthquake cycle the data was taken (i.e. right before the earthquake or long before one), the testing data slice is too short, or the mean absolute error is just not a good scoring method for this project.

In general, I found the testing error was roughly 20-25% lower(!) than the training and validation dataset for most models. This, again, is probably caused by how the testing data was chosen (i.e. right before the earthquake or long before one).

Despite those problems, I am very happy I chose this Kaggle competition as my capstone project. I've learned a lot about data processing and about a completely new to me type of NNs. It was also exciting to work on a project that might increase the security of millions if not billions of people by allowing for earlier forecasting of earthquakes. I am also very happy that I was able to noticeably decrease the error of the model as compared to the, already very good, feature benchmark. I think my solution is a good approach to solving problems like this.

Improvement

One thing that could definitely be improved is the fact that my model was trained only with very specific input data length in mind. I do not know how well it would generalize to data of different lengths. In general, a good forecasting algorithm for a real-world application should be able to continuously predict if an earthquake is incoming based on live data. For my algorithm the live data would need to be cut into chunks or predict based on a running window of measurements. A better way, in my opinion, would be to feed the data into the algorithm point-by-point, and allow it to modify its prediction based on the new datapoint just provided. It would allow the algorithm to react faster – and the speed if very important in such applications.

In addition, I think some more advanced processing of the data, apart from simple statistical functions, could be very beneficial to the project. There, for example, might be some information hidden in the frequency of the seismic_data. Using something like a Fourier transform-based processing of the data might improve the quality of the predictions.

I am also worried that all models based on the data provided would not generalize well to times when there is absolutely no risk of an earthquake – I tested my algorithm on self-made data with extremely low noise to check what will happen. My algorithm predicted close to 0 time_to_failure. It is clear as to why that is when one looks at the first graph in this report – the 0 time to failure is actually slightly shifted with respect to the very large spike in the seismic data, which causes the 0

time_to_failure to actually be right after the lowest seismic_activity in the whole dataset.

Literature:

[1] J. R. Leeman et al., Laboratory observations of slow earthquakes and the spectrum of tectonic fault slip modes, Nat. Commun. 7, 11104 (2016). (https://www.nature.com/articles/ncomms11104)
[2] https://medium.com/explore-artificial-intelligence/an-introduction-to-recurrent-neural-networks-72c97bf0912
[3] https://www.math.kth.se/matstat/seminarier/reports/M-exjobb18/180608m.pdf
[4] https://blog.statsbot.co/time-series-anomaly-detection-algorithms-1cef5519aef2
[5] https://www.emsc-csem.org
[6] C. F. Richter, Elementary Seismology, 1958
[7] https://www.statista.com/statistics/263108/global-death-toll-due-to-earthquakes-since-2000/