Application of Machine Learning for Earthquake Detection

Matthew Gonzalgo, Ethan Jaszewski, Su Min Lee, Serena Yan

FA 20-21

1 Abstract

Until recent years, earthquake signal detection and seismic phase picking has required an expert in the field of seismology to examine wave signals and manually label the arrival times. Despite having decades worth of seismic wave data readily available, the application of machine learning for these tasks is still largely unexplored. We re-implemented the state-of-the-art EQTransformer earthquake detection model originally implemented by *Mousavi et al.* [1] in PyTorch, then expanded upon EQTransformer by incorporating several machine learning techniques, namely normalizing flows and squeeze and excite networks. In our final model, we placed a layer of linear normalizing flow as the pre-processing layer, and one squeeze and excite block in between each convolutional layer in the neural networks. Our modified final model achieved the same accuracy measures as the EQTransformer model while significantly reducing the runtime. We concluded that the transformer layer has the biggest impact on improving model performance while the LSTM and attention layers of the original model have the least impact. Of the two structures that we added, the squeeze and excite block has a larger impact on improving model performance.

2 EQTransformer

Our baseline model utilizes the EQTransformer model implemented by *Mousavi et al.* [1]. The neural network has a multi-task structure consisting of one encoder and three separate decoders composed of 1D convolutions, and a transformer layer in between the encoder and decoder layers. The encoder consumes the seismic signals in the time domain and generates a high-level representation and contextual information on their temporal dependencies. Decoders then use this information to map the high-level features to three sequences of probabilities associated with: existence of an earthquake signal, P-phase, and S-phase, for each time point. The original EQTransformer also includes a bi-directional and a uni-directional long-short-term memories (LSTM) layer, but we excluded

LSTM in our final model as our experiments show that they do not contribute to model performance overall.

3 Improvements: Accuracy

3.1 Normalizing Flows

3.1.1 Background

Normalizing flows is a type of generative model that specifically relates to the likelihood distribution of the data present. Breaking up the two words, the "normalizing" part of normalizing flow implies that the density produced after this model is normalized. The "flow" part of normalizing flow implies that this model is composed of one or multiple invertible transformations. The main objective of utilizing normalizing flow is to construct more complex distributions from the existing simple distributions with the key point that the transformations involved are invertible.

$$p_X(\mathrm{x};\theta) = p_Z(f_\theta^{-1}(\mathrm{x}))\left|\det\left(\frac{\partial f_\theta^{-1}(\mathrm{x})}{\partial \mathrm{x}}\right)\right|$$

Figure 1: The change of variable formula involved in normalizing flow.

One advantage of normalizing flow [4] is the broad possibilities of specific implementations to fine tune to the objective. Since normalizing flows broadly covers all normalizing invertible transformations, there are many types such as planar, radial, autoregressive, inverse autoregressive, etc. Another advantage of choosing to add normalizing flows was the compatibility and flexibility with what happens in later parts of the model and the fast calculation speed. By preprocessing through normalizing flows in the very beginning of the model, we aimed to transform our initial data into a form that can be more easily learned by the neural nets in the later parts of the model.

3.1.2 Implementation

For our final model, an ActNorm [5] type normalizing flow was added to the very beginning of the network. ActNorm uses transformation in the form $f(x) = Wx + b$, where W is a diagonal matrix and b is a constant. As prompt prototyping was important in the time constraint of the project, ActNorm was the simplest form to implement and debug. Implementing more complicated normalizing flows are immediate next steps that can possibly improve the performance of the model. This will be further discussed in the next steps portion of the report.

3.2 Squeeze and Excite Network

3.2.1 Background

An inherent problem of Convolutional Neural Networks is that each convolutional layer operates within a local receptive field. As a result, each unit of the transformation output is unable to exploit information outside of its local region. A proposed solution to this problem is the Squeeze and Excite network/block, which "squeezes" global spatial information into a channel descriptor using global average pooling and captures channel-wise dependencies using an activation function (excitation). Another appeal of the Squeeze and Excite block is that they are computationally inexpensive to add into an existing network. [2].

Additionally, since the development of the original channel-wise SE model (SE), a Time-Frequency-Channel Squeeze and Excite (tf-SE) network [3] has been developed specifically for improved performance with time-series events.

3.2.2 Channel-Wise

It is relatively straightforward to add a SE block onto any structure. For example, in our final model, we have inserted a SE block after each convolutional layer to recalibrate the feature representations.

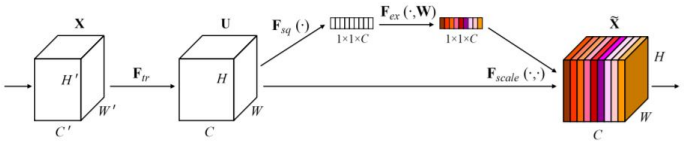

Figure 2: Visual representation of channel SE block used in the final model.

3.2.3 Time-Frequency-Channel

The tf-SE network is an add-on to the standard SE block that was designed specifically for time-series. There are two approaches for implementing the tf-SE block: concurrent and sequential.

As the name implies, the concurrent implementation aggregates the SE and tf-SE blocks concurrently, which can be done either by element-wise addition, element-wise multiplication, element-wise maximization activation, or by concatenation of the two feature maps (note that concatenation results in a feature map twice as large).

The sequential implementation is a more straightforward approach that feeds the output of the SE block directly into the tf-SE block. Between the two, the sequential implementation is both simpler and performs better according to the initial findings of *Xia et al.* [3]. While the tf-SE did not make it into our final model, it shows promise as a potential next step.

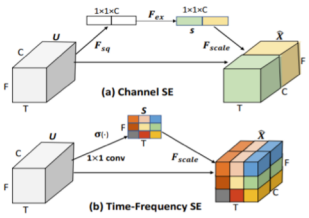

Figure 1: *Channel SE and Time-Frequency SE blocks.*

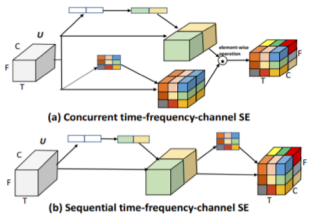

Figure 2: *Concurrent time-frequency-channel SE and sequential time-frequency-channel SE blocks.*

Figure 3: In Figure 3.1, (3.1a) reillustrates the visual representation of the channel SE block from Figure 2 to compare with the time-frequency SE block in (3.1b). Figure 3.2 shows how the two SE blocks are aggregated by the concurrent implementation (3.2a) and the sequential implementation (3.2b).

4 Improvements: Speed

4.1 Removing Layers

During our analysis of the EQTransformer, we investigated removing many of the different layers in order to improve performance. In order to do this, we trained the network repeatedly, removing one layer or set of layers at a time, and comparing the performance of the full model with the performance of the model with layers removed. We investigated the removal of the ResCNN layers, the Attention layers, the Transformer layer, and the LSTM layer. Ultimately, the ResCNN and Transformer layers proved too important to the performance of the layer to be removed, and the Attention layer, while not critical to network performance, had little impact on training and inference speed, so it was left in. The LSTM layers, however, appeared to be either unnecessary or redundant. Removing the BiLSTM layers and the LSTM layers at the start of the decoders did not impact performance but rather yielded nearly a 20% reduction in training time.

4.2 Other Optimizations

4.2.1 Data Loading

The STEAD dataset used, as provided, is a large HDF5 file with a hierarchical format where each trace is contained in a separate dataset containing the waveforms and corresponding metatada. This HDF5 file is coupled with a CSV which contains that metadata. Although good for sampling traces one at a time, this hierarchical format is extremely slow with bulk loading, taking several seconds to load only a handful of waveforms. In order to improve training time, we decided to convert the data from the HDF5 into a non-hierarchical format so that it could be bulk-loaded more easily.

We saved the waveforms without metadata as a dense NumPy array on-disk, and built a CSV which stored the metadata of a waveform along with the index of the waveform in the on-disk array. Once the CSV file is loaded, the NumPy array can either be loaded into memory or memory-mapped, and traces accessed either one-by-one or in bulk. Loading from this flattened format is nearly an order of magnitude faster than reading from the original STEAD HDF5 file.

We then wrote a custom PyTorch Dataset which uses the flattened format. It reads the whole CSV and memory-maps the array to reduce loading time. This Dataset is then used with provided PyTorch DataLoaders which multi-thread data loading operations. Theoretically, there is a small performance improvement if the array is loaded completely into memory; however, it is not important during training because the computation during a batch exceeds the time spent loading the data (so there is no waiting for the data at each iteration, even with the mmap). Moreover, if the data were to be loaded into memory, it could not be multithreaded, as the memory required would far exceed the memory available to us.

4.2.2 Batch Size

Another area we explored to improve performance was the batch size used during training. The EQTransformer paper makes no mention of batch size, so we tried several different batch sizes between 32 and 1024. Ultimately, smaller batch sizes both trained slower and caused the network to generalize worse, so a batch size of 1024 was used for all subsequent testing.

5 Final Model

5.1 Structure

Our final model retains the same general structure as the EQTransformer model, but incorporates the SE layers and Normalizing Flows. We modified the EQ-Transformer by first adding an ActNorm layer at the very beginning of the network. We then added SE layers after each of the CNNs and ResCNNs in both the encoder and the decoder. Finally, we removed the BiLSTM layers from the encoder and removed the LSTM layers from both the P and S decoders.

5.2 Evaluation

The final model achieves performance on-par with the original EQTransformer model while being approximately 15% faster to train. We performed a final comparative run between the EQTransformer and our model by training each for 45 epochs (at which point the validation loss had not improved for three consecutive epochs), with the results outlined below.

		EQTransformer	Final Model
P Arrival	F1	0.991	0.990
	Precision	0.982	0.981
	Recall	1.000	1.000
	MAE	5.172	5.396
S Arrival	F1	0.895	0.895
	Precision	0.810	0.810
	Recall	1.000	1.000
	MAE	11.717	11.471
Detection	F1	1.000	1.000
	Precision	1.000	1.000
	Recall	1.000	1.000

Figure 4: Comparison of performance across four different metrics between EQTransformer and our Final Model.

For each waveform, the P and S arrival predictions are the locations of the maximum value in the predicted output waveforms if the value was above the detection threshold, otherwise a prediction is not made. A detection is counted if the detection waveform rises above the detection threshold at any point in the output. P and S detections are treated as true positives if the detection made falls within 15 samples of the actual arrival time. Ideal detection thresholds were determined by testing several different thresholds on a holdout set and selecting the threshold for each of P, S, and detection that yielded the best F1 score.

In the final comparison, our final model achieves effectively equal performance to the original EQTransformer in all categories. Notably, both networks achieve very good performance in the P Arrival prediction, and have nearly perfect performance on the Detection prediction, yet struggle with S Arrival prediction. Four example traces are included below that outline the performance of the network.

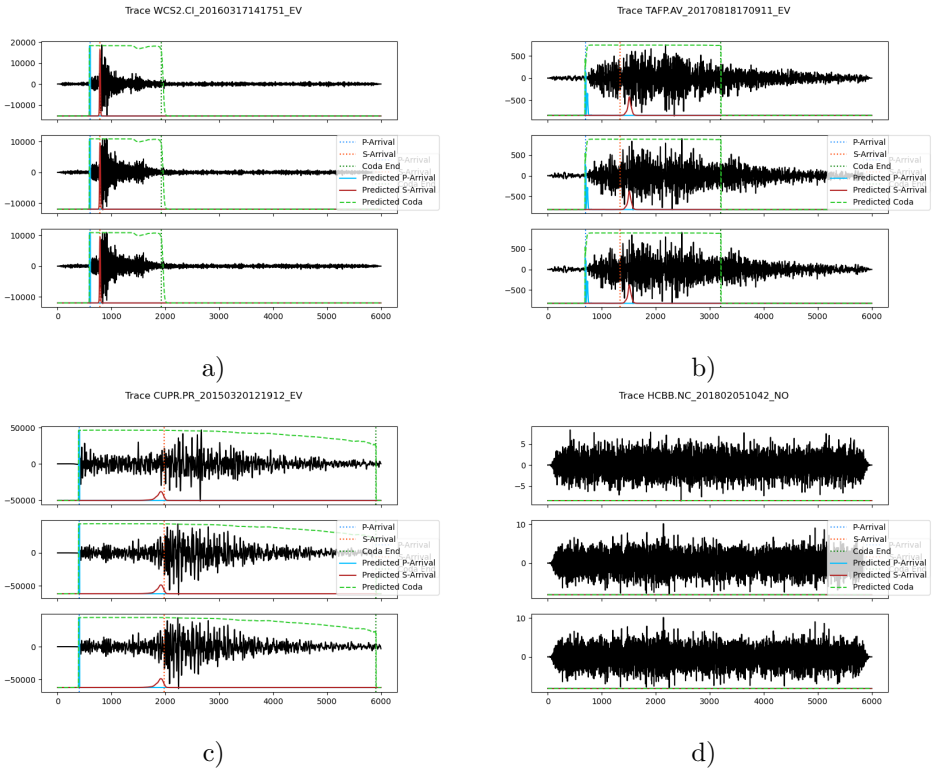

Figure 5: (a) is an example of proper prediction of the P arrival, S arrival, and detection. Note the high confidence level on all three predictions. (b) represents a trace where the S arrival prediction is shifted beyond the accepted range, with the potential inaccuracy reflected by a relatively low confidence. (c) is an example of the S arrival being predicted within the allowed tolerance, but with a very low confidence value, meaning the detection may not be accepted. (d) is an example of a noise trace, where all three of the predicted waveforms are nearly zero for the entire trace.

5.3 Next Steps

5.3.1 Normalizing Flows

There exists more complex normalizing flows developed specifically for the purpose of sound event detection and anomaly detection. For instance, AdaFlow is a unified model of normalizing flow and adaptive batch-normalization developed specifically for anomaly detection. Autoregressive flows and inverse autoregressive flows are stochastic type of normalizing flows often utilized in sound detection. These complex normalizing flows can better accomplish preprocessing the data into a form that is easier to learn for neural networks, leading to

improvement in accuracy.

5.3.2 Time-Frequency-Channel Squeeze and Excite

The final model incorporated standard SE blocks which showed positive results, so converting each SE network into a tf-SE network could be a simple yet effective means to further improve performance. With a modular tf-SE block properly implemented, in theory we would simply have to insert a tf-SE block after every SE instance in the SENetwork for the sequential implementation.

6 Conclusion

To conclude, we revisit the three initial goals and their respective outcomes. Firstly, the project re-implements EQTransformer in Pytorch with similar level of performance. Secondly, the project attempts to explore additional methods to better understand the space. Although the final model does not significantly outperform EQTransformer, Normalizing Flows and Squeeze and Excite models were explored and implemented with insight for further enhancements. Lastly, the project simplifies EQTransformer to reduce training time without a significant impact on performance. This was accomplished by removing layers we deemed non-critical to the performance of the model i.e. the Bi-LSTM layers and all but one LSTM layer. At the same time, the final model was able to incorporate Normalizing Flows and Squeeze and Excite models at a computationally low cost. An extensive exploration of EQTransformer and a final model with initial implementations of new approaches, this project provides solid base work for even further enhancement of EQTransformer and application of machine learning to Earthquake Detections.

7 Notes

Our source code and model can be found at the following GitHub repository:
`https://github.com/ejaszewski/cs101`

8 Acknowledgements

We would like to thank Katie Bouman, Zach Ross, Jeremy Bernstein, Joe Marino, and Jonny Smith for their expertise and mentorship throughout our time working on the project and for making this project possible. We also thank Zach Ross & Scott Dungan for allowing us access to the Arius server which significantly increased our workflow.

References

[1] S. Mostafa Mousavi, William L. Ellsworth, Weiqiang Zhu, Lindsay Y. Chuang & Gregory C. Beroza. Earthquake transformer—an attentive deep-learning model for simultaneous earthquake detection and phase picking. Nat Commun 11, 3952 (2020). https://doi.org/10.1038/s41467-020-17591-w

[2] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, Enhua Wu. Squeeze-and-Excitation Networks. *ArXiv*: 1709.01507v4, 2019

[3] Wei Xia, Kazuhito Koishida. Sound Event Detection in Multichannel Audio using Convolutional Time-Frequency-Channel Squeeze and Excitation. *ArXiv*: 1908.01399v1, 2019

[4] Rezende, D. J. and Mohamed, S. Variational Inference with Normalizing Flows. in Proceedings of the 32nd International Conference on Machine Learning - Volume 37 - Volume 37 1530–1538 (JMLR.org, 2015) *ArXiv*:1505.05770v6, 2015

[5] Kingma, D.P., and Dhariwal, P. (2018). Glow: Generative Flow with Invertible 1x1 Convolutions. In Advances in Neural Information Processing Systems 31, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds. (Curran Associates, Inc.), pp. 10215–10224. *ArXiv*:1807.03039v2, 2018