Rizq Khateeb

10 April 2025

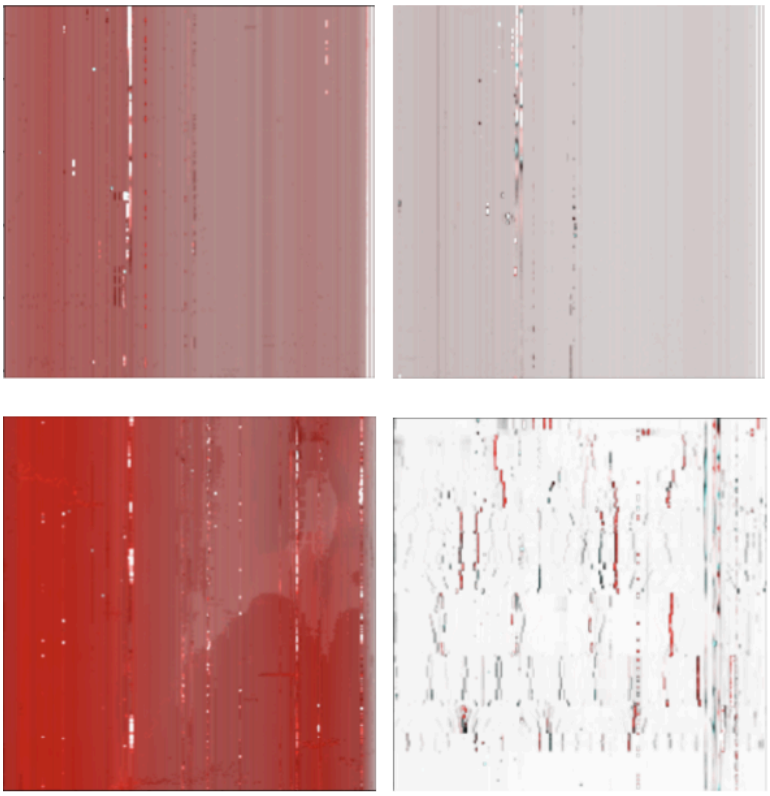
Anomaly Detection NN on the ROAD Dataset

The purpose of this project was to design a successful anomaly detection system identifying anomalies and the types of anomalies within the ROAD dataset.

Data Processing:

The first step involved learning about the data and how it was structured. The ROAD_dataset file is separated into 3 groups: training data, testing data, and anomaly data. The anomaly data is further split into 9 different groups for anomalies. Each entry contained the data itself, the frequency, the label of the data's category, the unique id, and the source. I chose to mainly focus on the data and labels for each entry. The source was not useful, and based on what I read in the paper, the frequency is represented by one of the data's dimensions. The data itself had dimensions corresponding to time, frequency, polarization, and station. Because of how the data was organized, the biggest problem I was initially running into was the extraction of data. This was my first time working with an .h5 file, and this was an extremely large one. I was working primarily on colab, which meant limitations in storage and RAM usage. I initially tried to upload all the data to a .csv just to see how much data I was truly working with, but this proved to be futile. Next, I decided to see if I could separate the data into separate h5 files for training and testing. I tried this out but soon ran into a problem. Because of the way the data was organized, I first had to create h5 files for the training and testing (which was simple because these were already split), and then create a file for anomaly data before splitting that file into separate training and testing files for anomalies. This was a cumbersome task, not to mention one that required a large amount of disk space compared to how much Google Drive traditionally gives

you. After much thought and experimentation, I settled upon accessing the data through a torch dataset directly from the original h5p file on the fly, which would allow for resource-efficient lazy loading. To separate the data into training and testing, I first focused on splitting the anomaly data (since the normal train and test data were already split in a near 80% split). For each anomaly group, I placed the path and indices in a tuple, extracted them, and then shuffled such that 80% were placed in a list for training, and 20% were placed in a list for testing. This was important because I wanted to ensure an 80/20 split for each anomaly category independently. Next, I joined the normal training paths and indices with the anomaly training paths indices, and did the same for testing. From here, I shuffled both resulting lists to get the full set of paths and indices for training and testing. After this, it was simply a matter of using the path and the index from the tuple and extracting data on the fly as part of the get_item function



Here we have 4 visualizations of the data. The top 2 are classified as normal, and the bottom 2 are classified as anomalies (left = galactic plane, right = oscillating tile).

There may be some features detectable to the human eye within these images, but disparities within the data are what causes differences to be evident to the model and for classification to be accurate and precise as compared to a human manually completing this task.

implementation inside the custom dataset. Doing so meant we did not have to load in all the data at once, thus saving time, memory, and disk space.

The Model:

Before even beginning the processing of data, I first wanted to look at the data and develop an idea of what I was working with. To do so, I read through the original provided paper, and visualized samples of the data myself. The data itself was present as a 256x256x4 numPy array, and visualizing as an image led me to decide on a CNN as the most appropriate neural network to develop from scratch (normally I would have started with a pretrained model like ResNet). Using a CNN would allow for the most important features to be identified and used when determining whether the entry is normal or an anomaly. I decided to use PyTorch for the development of my CNN, and chose to keep it fairly simple in terms of the number of convolutional layers. The structure is as follows:

1. First Convolutional Block: Consists of a convolutional layer, batch normalization layer, and pooling layer. Applies convolution, normalizes the output and activates with ReLU, and then reduces the spatial size with max pooling.
2. Second Convolutional Block: Consists of a convolutional layer and batch normalization layer. Adds more filters to capture more complex features, and reapplies batch normalization and ReLU activation. The spatial size is reduced once again with max pooling.
3. Third Convolutional Block: Consists of a convolutional layer and batch normalization layer. Same as previous, adding even more filters to capture more complex features, and reapplies batch normalization and ReLU activation. This allows for extraction of even higher-level features. A final max pooling layer is added as well.

4. Dropout Layer: Deactivates a portion of neurons to prevent reliance on them and consequently prevent overfitting.
5. GAP Layer: Global average pooling layer that collapses each feature map to a single value by averaging over its spatial dimensions, helping to reduce the number of parameters.
6. Fully Connected Layer: Flattens the pooled features and maps them to final output classes to be classified.

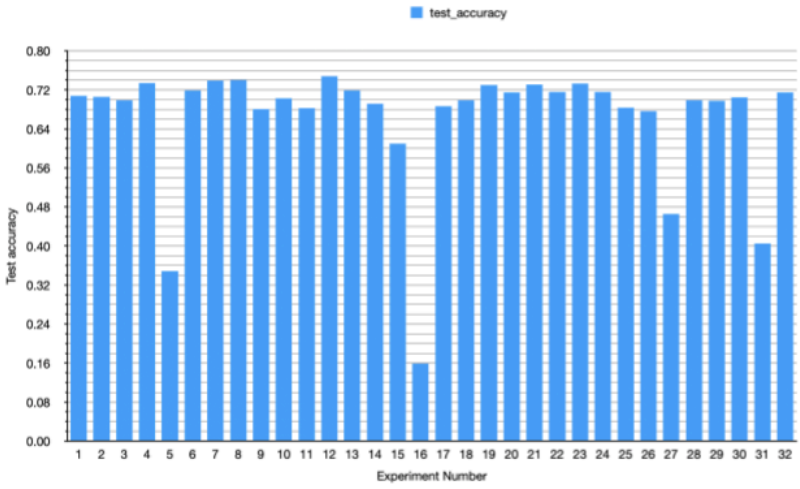
Hyperparameter Tuning:

To make sure I was developing the optimal model, I carried out hyperparameter tuning for a variety of combinations of multiple hyperparameters. I focused on the hyperparameters of dropout, kernel size, filters, batch size, and learning rate to determine which combination would be best in providing a high testing accuracy.

- Learning Rate: Learning rate controls the step size taken during optimization. A high learning rate could lead to divergence or overshooting, while a low learning rate could lead to slow convergence or getting stuck in local minima. It is essential to find a medium that mitigates both of these issues
- Batch Size: Batch size determines how many samples are processed before the weights are updated. Smaller batch sizes are noisier and take longer to train, but help escape local minima and improve generalization. Larger batch sizes speed up training and are more stable, but can poorly generalize as a result of overfitting.
- Filters: Choosing the right number of filters will determine how many feature maps the layer will produce. More filters allows for greater feature extraction, but also increases

the number of parameters, computational complexity, and risk of overfitting. Regularization and sufficient data can help mitigate these issues.

- Kernel Size: Kernel size determines how much of the image is considered in each convolutional layer. Smaller kernels capture finer details and are more efficient as opposed to larger kernels which can capture broader contextual information.
- Dropout: Dropout is a technique that aids in the prevention of overfitting. It randomly deactivates a fraction of neurons, thus helping the network generalize better



Here is the graph of hyperparameter tuning results. The x-axis represents test-accuracy, while the y- axis represents the experiment number.

After testing multiple combinations, I came to the conclusion that the best hyperparameter choices were the following:

- Learning Rate: 0.001
- Batch Size: 32
- # of Filters: 16
- Kernel Size: 5
- Dropout: 0.5

Coming up with these optimal hyperparameters did come with some caveats. I was not able to test as many combinations as I would have liked. Timing constraints caused by the due date and limitations of my hardware meant that I only really was able to try out around 32 combinations. In addition, I only analyzed the results of each of these combinations after 3 epochs (since for each combination it took 2 hours to run on average on my hardware). Thus, I may not have ultimately chosen the most optimal combination of hyperparameters. I tried to work around by running multiple notebooks in parallel and breaking down the combinations (for example I noticed that dropout of 0.5 was performing better than 0, so I started a new set of epochs all with dropout rates of 0.5). This did help me speed up the process and determine a suitable combination upon which to continue and run 10 epochs of training for, but limited my thoroughness in determining the best hyperparameter combination.

Training and Testing Results:

After determining the best combination of hyperparameters, I ran 10 epochs of training with the optimal combination of hyperparameters, and then validated on my testing data. With the optimal hyperparameter configuration, our model achieved an accuracy of approximately 76%. This means that it correctly classified the data as either normal or one of the specific anomalies 76% of the time, surpassing the 10% accuracy expected from random guessing. These results clearly demonstrate the effectiveness of my approach in identifying distinct data conditions, validating the success of our CNN classifier in the anomaly detection task.

Limitations:

The provided paper and the study associated with it were able to achieve higher accuracy rates in classification compared to my experimentation, but I was limited by a combination of factors that prevented me from finding the most optimal anomaly detection model possible. The biggest

limitation was time. With 4 business days, I had to act fast to explore the data, come up with a strategy from scratch, and both train and test a CNN to achieve sufficient performance. Working with a large dataset and limitations in hardware (Google Colab would often crash after long periods of runtime) further added to these problems. I also ran locally on my desktop in parallel to save resources and time, but even with this strategy it was still taking several hours to train on one set of hyperparameters. Were I to work on this project again for a longer period, I would primarily focus on first analyzing other models that might provide better performance for anomaly detection (the original paper credits ResNet architecture as being particularly useful). I would also likely build off a pretrained model like ResNet, similar to what the authors of the paper chose. After selecting the optimal model, I would dedicate more time and resources to ensuring I tested all relevant hyperparameters sufficiently (including different model architectures with more layers, for example) to achieve the optimal accuracy. Although my accuracy was not bad (it was significantly better than the 10% accuracy that would be associated with random classification), there is still room for improvement as is seen by the high accuracy rate achieved by the paper.