Deep Neural Network for Oil Spill Detection

GitHub Repository: https://github.com/CoryKjar/OilNet

Cory Kjar
Southeast Missouri State University
Cape Girardeau, Missouri
cokjar1s@semo.edu

Nirmal Kandel
Southeast Missouri State University
Cape Girardeau, Missouri
nkandel1s@semo.edu

Abstract—In this paper, we will introduce a deep neural network framework developed for detecting oil spills based on satellite imaging. The data used to train our model underwent extensive preprocessing via computer vision algorithms that will be detailed later. However, the main concept still applies and the idea of detecting oil spills remains the same.

1. Introduction

While most are aware of the disastrous effects of oceanic oil spills, far fewer are aware of how common they are. In just U.S. waters, thousands of oil spills occur annually. While most of these spills are minor, they can still have harmful effects. However, large oil spills, especially those in oceanic waters, can be massive and have devastating outcomes for both the environment and humans.

While it may not be possible to prevent every oil spill, there are measures that can be taken to detect when an oil spill occurs so that it can be monitored and responded to promptly, hopefully before oil spreads uncontrollably.

One proposed method of detecting oil spills involves employing the use of synthetic-aperture radar (SAR) [1], which can be used in satellites, for imaging earth in extreme resolution. At a high-level SAR satellites can take clear images of the earth's surface even when there are obstructions such as clouds or weather. Essentially, these satellites are able to "see through" atmospheric obstructions. This is useful as these satellites can capture high-resolution images of ocean patches. These images of ocean patches have many use cases. However, these image files can be massive and have limited accessibility. Our workaround for this obstacle will be discussed in section 3 of this paper.

There are many other applications for SAR satellites and they all have unique and highly interesting uses. These satellites, along with the SAR technology, can monitor shipping ports, airports, military bases, and more. There are, however, ethical questions that arise when discussing such an invasive imaging process. However, that is outside the scope of this paper and rather we will focus on the benefits of this technology, like rapidly detecting and responding to crisis situations and natural disasters. Other applications can be read about at [2], where it discusses border monitoring, forest monitoring (for detecting forest fires, monitoring deforestation, etc.), and sea ice monitoring.

This paper aims to develop a semi-novel method for detecting oil spills. While not entirely a new idea, the concept of machine/deep learning approaches to oil spill detection is relatively new, and we hope to make a meaningful contribution to the cause

2. Data

A. Issues Gathering Data

As stated in the introduction, data from SAR Satellites is not highly accessible. There are several approaches to gathering imaging data from these satellites. Our first and most immediate discovery was contacting the European Space Agency, however, data requests through this organization can take up to two months, as we did not have the time for that for the scope of this project. Our next finding was an organization known as ICEYE. ICEYE is an organization that owns and operates a collection of SAR Satellites. They use these satellites for a variety of research,

monitoring, and detection. See [2] for more on ICEYE. ICEYE had several attributes that interested us. First, they do specific work on detecting oil spills. They even have an oil spill SAR image dataset. However, this dataset was only accessible through a special GIS program. That is when we came upon a new finding. While it was not optimal, it worked and worked well for the timeframe of this project. The dataset we ended up working with was downloaded from the online data science community, Kaggle [3]. The next thing to do was to begin studying the dataset.

B. ***Dataset and Dataset Analysis***

As mentioned, the dataset we ended up working with was from Kaggle. Although it was not the ideal data we had envisioned, it actually came from the same source. This dataset was not unlike the dataset we had in mind. It was derived from the SAR images we hoped to work with. The difference is that while the SAR images were image files and we could train a convolutional neural network on them, this was a tabular dataset. This tabular dataset was the result of processing the SAR images in a computer vision (CV) algorithm. This CV algorithm returned a vector of features that detail the content of each ocean patch image.

Once downloaded and imported into a Jupyter Notebook, we began an initial analysis of the dataset. Figure 1 shows a brief overview of the dataset, however, this is only part of the entire dataset, as it had a shape of (937, 50).

f_44	f_45	f_46	f_47	f_48	f_49	target
135.46	3.73	0	33243.19	65.74	7.95	1
1648.80	0.60	0	51572.04	65.73	6.26	0
45.13	9.33	1	31692.84	65.81	7.84	1
144.97	13.33	1	37696.21	65.67	8.07	1
109.16	2.58	0	29038.17	65.66	7.35	0

Figure 1: Preview of Dataset

This dataset has 50 columns, including the target variable. Each feature represents some description of the relative ocean patch. The next thing we noted about the data was the data types, all of which were float64 or int64. We also noted that there were 0 null values in the entire data, which was beneficial as it assisted preprocessing steps. Finally, we checked the distribution of the target class. This is where we ran into another issue. The target class was extremely imbalanced.

C. ***Target Class Imbalance***

As stated, the target class in this dataset was highly imbalanced with the negative case being the large

Figure 2: Target Class Distribution

majority. Imbalance essentially means that the target variable has a lot more of one outcome than another. In our case, less than 5% of the data represented a positive case (oil spill). The distribution of the target class can be seen in Figure 2.

First, let us understand why this imbalance is an issue. In our situation, the imbalance was so extreme, that when we originally trained our model, it was highly overfitting. We discovered the problem was the target distribution because the model was only predicting 0. Essentially, the model saw that nearly every case was 0 (negative) and learned that it would be right most of the time by predicting 0. While this is not necessarily wrong. Should there be a positive case, the model would not detect it. So to solve this, we had to synthesize new data. The method we chose for this was oversampling. According to [4], oversampling is the process of randomly duplicating values from the minority class to use in the training data. To implement this, we used the imblearn package [5] which has a convenient random oversampling method [6]. Figure 3 shows our code for implementing imblearn's random oversampler.

```
# Use oversampling to create new data
oversample = RandomOverSampler(sampling_strategy='minority')
# fit and apply the transform
X_over, y_over = oversample.fit_resample(X, y)
# summarize class distribution
print(Counter(y_over))
```

```
Counter({0: 896, 1: 896})
```

Figure 3: Implementing Oversampling

In the above code snippet, we defined the oversampler and set it to a variable oversample. We used the minority sampling strategy, which adds samples based on the minority class (or in our case, negative cases). The next line sets the two numpy arrays to the variables X_over and y_over, where X_over will be the data and y_over will be the target variables. We passed X and y (the data and targets of the original data) to the oversampler's fit method, which returned two respective numpy arrays. Finally, we used the python

collections counter method on y_over and as can be seen by the output in Figure 3, there are now equal positive and negative cases.

D. ***Preprocessing***

The final phase of working with our data before building a model was some simple preprocessing. The first thing we did was split our data into a training set, a test set, and a validation set. We decided on using 80% of our data for training and then evenly split the remaining 20% into the test and validation sets. Figure 4 shows the shapes of these different data sets.

Note: We realized that one feature was all 0 values and thus, we dropped it. So the new data shape is (x, 48).

```
X_train shape: (1433, 49)
y_train shape: (1433,)

X_test shape: (180, 49)
y_test shape: (180,)

X_val shape: (179, 49)
y_val shape: (179,)
```

Figure 4: Split Data Shapes

Following this we simply One-Hot Encoded the target variables. Lastly, we scaled the data using the StandardScaler function in sklearn [7]. Standard scaling is a scaling technique that works by subtracting the mean and dividing by standard deviation. This equation is shown below in Figure 4.

$$z = \frac{x - \mu}{\sigma}$$

μ = Mean

σ = Standard Deviation

Figure 4: Standard Scaling

3. Method

A. ***Model Design***

The next step in the project was developing and implementing a model. As we have covered neural networks through this course, naturally we will be using some form of a neural network. First, we started with what we knew. We knew that our model would need to intake tabular data. Because of this, we will go with a deep neural network. Much of the model development was trial and error. We began with a basic fully connected model consisting of 5 layers (1 input layer, 1 output, and 3 hidden layers). As we knew the data shape, we set our input layer to intake 48 features. Likewise, we knew this was a binary classification problem, so we initialized our output layer with 2 neurons, and applied a sigmoid activation function as is commonly used for binary classification. To give a brief explanation, as seen in Figure 5, the sigmoid function exists between the values of 0 and 1, which makes it useful for predicting probabilities.

Next, we implemented 3 hidden layers, all using the Rectified Linear Unit (ReLU) activation function, which has become one of the most common activation functions for hidden layers. See Figure 5 below for the equation and graph of the ReLU function. We set the number of neurons to 64 in layer 1 and 32 in layers 2 and 3. We also used a learning rate of 1e-4 or 0.0001.

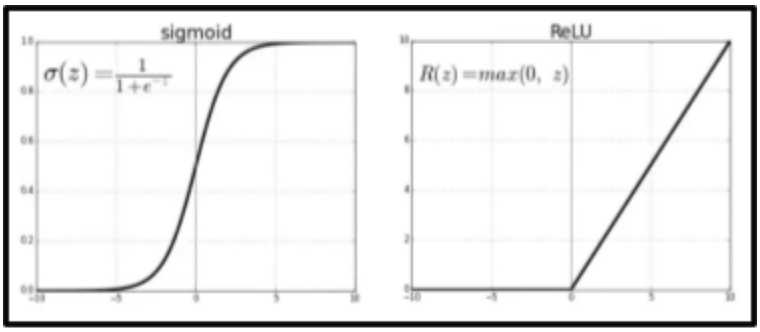

Figure 5: Sigmoid & ReLU Activation Functions [8]

As for our model's optimizer, we stuck with the standard Stochastic Gradient Descent (SGD). Finally, in compiling our model, we used binary_crossentropy for our loss function as is standard for binary classification and accuracy for our model's performance metrics (We will evaluate the model more precisely later on).

Upon compiling this model and returning a summary, we could see that the model had 6,338 trainable parameters. However, upon initial training, the model did not perform very well, so we went back to tune the model.

B. ***Model Tuning***

Tuning the model was a lot of trial and error as we changed minor details and retrained the model each time to see how it impacted its performance. The following is a list of model-tuning steps we took to achieve the final model:

- Decreased learning rate to 1e-6
- Changed the number of neurons
    - Hidden layer 1: 64
    - Hidden layer 2: 48
    - Hidden layer 3: 32
- Between hidden layers 2 and 3, we added a BatchNormalization layer. This replaced a dropout layer we had implemented in testing.

After performing these steps, our model had satisfactory results. The final model's architecture can be seen below in Figure 6. As shown in the figure, the final model has a total of 8,082 parameters. As mentioned above, we replaced the Dropout layer with a Batch Normalization layer which has a regularizing effect, so it deactivates some (in this case 96) neurons. This leaves us with a total of 7,986 trainable parameters.

```
Model: "OilNet"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 Input (InputLayer)          [(None, 48)]              0

 Hidden1 (Dense)             (None, 64)                3136

 Hidden2 (Dense)             (None, 48)                3120

 Normalization (BatchNormali  (None, 48)               192
 zation)

 Hidden3 (Dense)             (None, 32)                1568

 Output (Dense)              (None, 2)                 66

=================================================================
Total params: 8,082
Trainable params: 7,986
Non-trainable params: 96
```

Figure 6: OilNet Architecture

C. ***Model Training***

As with designing our model, training took some testing to find parameters that produced satisfactory results. First, we implemented Early Stopping, which will stop training when and if a condition is met. In our case, we set it up to monitor when validation loss reaches a minimum and does not decrease for 20 epochs.

Next, we defined the model history. We played around with the batch size a lot and ended up settling on a batch size of 64. We originally set the number of epochs to 1,000 as we had early stopping set up and it would be unlikely to need so many epochs. The early stopping callback stopped training around 600 epochs, but increases in accuracy were minimal after about 120, so we settled with 100 epochs in the end. Figure 7 below shows the last few lines of output by the training function. As seen in the figure, preliminary results look good, but this high of accuracy led to worries about overfitting. That is where model evaluation comes into play.

Figure 7: Training Log

4. Evaluation

Our model appeared to achieve outstanding results, however, we were cautious to accept these results at face value. The standard TensorFlow evaluation method returned results of:

- Cross-Entropy: 0.04009
- Accuracy: 0.9888

We also plotted these values by epoch. Figure 8 shows these plots.

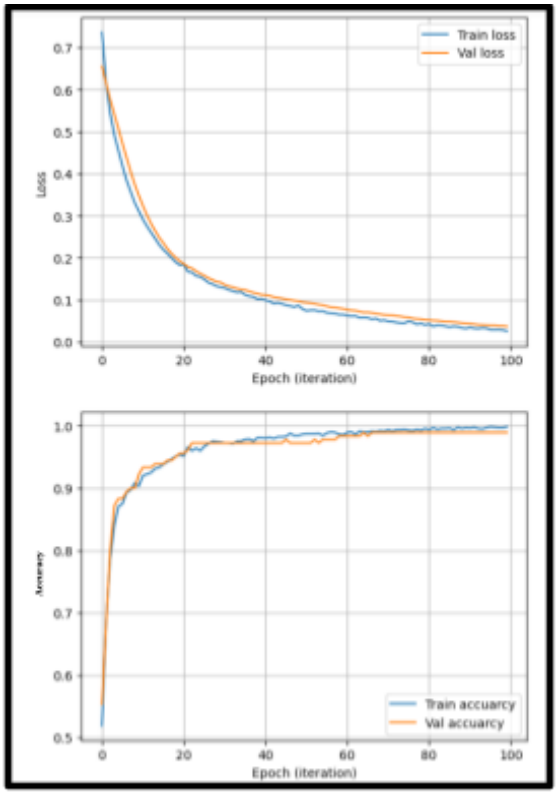

Figure 8: Loss & Accuracy by Epoch

These plots tell us a couple of things. As for overfitting, it does not appear to be the case as train and validation accuracy and loss follow the same trend and do not diverge. However, recall that this is an imbalanced dataset, so to raise our confidence in the model, we will perform some more evaluations. The other thing these plots show is that most of the learning the model did took place in the first 25 epochs, so realistically we could stop training there, but we were able to squeeze out a little more performance.

The next evaluation method we used was a confusion matrix. A confusion matrix is a classification evaluation method that, given a model's predictions and correct values, returns a matrix of true positives, false positives, true negatives, and false negatives. Figure 9 shows how to read a confusion matrix and Figure 10 is the confusion matrix of our model's predictions.

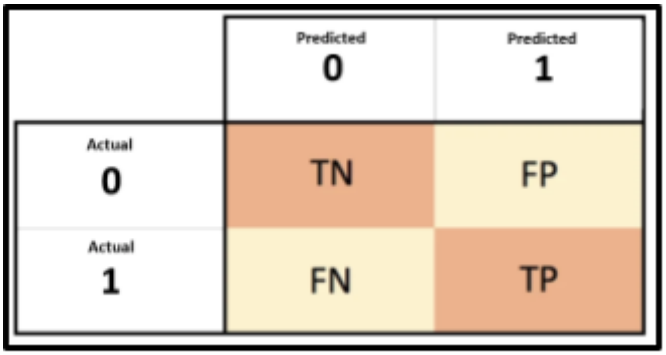

Figure 9: Confusion Matrix Example

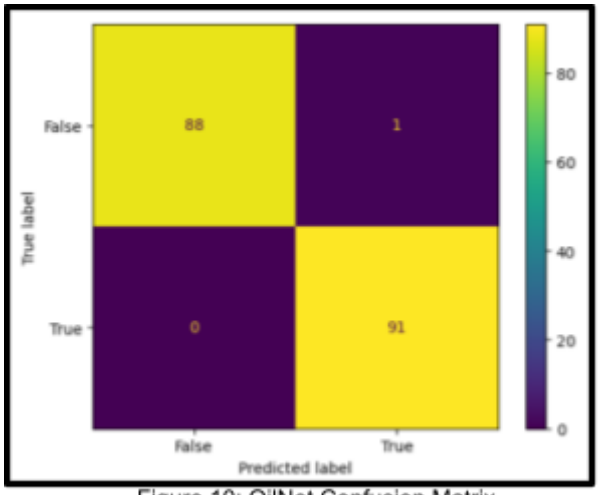

Figure 10: OilNet Confusion Matrix

As the above confusion matrix shows, our model's predictions were:

- True Positive: 91
- False Positive: 1
- True Negative: 88
- False Negative: 0

These results helped us be confident that our model was making predictions accurately. Thus, we state that our final accuracy for OilNet is 99.44% as per Keras' default accuracy method

5. Conclusion

Throughout this study, we have addressed the challenge of identifying oil spills from SAR satellite imagery that was processed in a computer vision algorithm. To do so, we proposed a deep neural network to classify whether an oil spill is present or not based on data derived from the features of the original image. The data, the model was trained on, was highly imbalanced, so we used over-sampling to add additional data to the minority class. We also had the challenge of designing and tuning a neural network. In the end, we managed to achieve a highly accurate model, with a prediction accuracy 99.44%.

References

[1] "The Value of SAR | Data | ICEYE," *www.iceye.com*. https://www.iceye.com/the-value-of-sar (accessed Dec. 06, 2022).

[2] "Sar Data Applications | Data | ICEYE," *www.iceye.com*. https://www.iceye.com/sar-data-applications (accessed Dec. 06, 2022).

[3] "ICEYE SAR Data Applications," *ICEYE*. [Online]. Available: https://www.iceye.com/sar-data-applications. [Accessed: 08-Dec-2022].

[4] "Why SAR?," *Capella Space*, 08-Mar-2021. [Online]. Available: https://www.capellaspace.com/data/why-sar/. [Accessed: 08-Dec-2022].

[5] "Institute of Electrical and Electronics Engineers," *IEEE*. [Online]. Available: https://www.ieee.org/. [Accessed: 08-Dec-2022].

[6] S. Rastogi, "Oil spill Classification," *Kaggle*, 12-Nov-2022. [Online]. Available: https://www.kaggle.com/datasets/sudhanshu2198/oil-spill-detection. [Accessed: 06-Dec-2022].

[7] J. Brownlee, "Random Oversampling and Undersampling for Imbalanced Classification," *MachineLearningMastery.com*, 04-Jan-2021. [Online]. Available: https://machinelearningmastery.com/random-oversampling-and-undersampling-for-imbalanced-classification/. [Accessed: 06-Dec-2022].

[8] "imbalanced-learn documentation," *imbalanced learn*. [Online]. Available: https://imbalanced-learn.org/stable/#. [Accessed: 06-Dec-2022].

[9] "RandomOverSampler," *RandomOverSampler - Version 0.9.1*. [Online]. Available: https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.RandomOverSampler.html. [Accessed: 06-Dec-2022].

[10] "Sklearn.preprocessing.StandardScaler," *scikit learn*. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html. [Accessed: 07-Dec-2022].

[11] S. Sharma, "Activation Functions in Neural Networks," *Medium*, 20-Nov-2022. [Online]. Available: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6. [Accessed: 08-Dec-2022].