Ryan Herrin, Luke Stodgel
DS7333
03/24/2023

Particle Detection using a Dense Neural Network

Introduction

In this project we used a Dense Neural Network model to try to accurately detect the presence of a particle. This was a binary classification task.

Methods

Because our objective was to detect a particle, and the only descriptive column name in our dataset was “mass” we thought that our dataset was made up of features pertaining to chemical properties. The shape of the dataset was 7,000,000 rows by 29 columns. Our target variable, named “# label”, contained values of 0 and 1. In 3,500,879 rows, our target variable was labeled 1, and in 3,499,121 rows, it was labeled 0. The classes in our target variable were almost perfectly balanced.

Data Preprocessing

First, we loaded our data into a dataframe using pandas’ read_csv function. Next, with our dataframe, we created our X and y variables and converted them to float32 from float64 in order to save on processing time and to use less RAM. Next, we scaled our X data using StandardScaler’s fit_transform function and created our X/y train/test split using the train_test_split function.

Addressing the Missing Data

There was no missing data in this dataset.

Building our Dense Neural Network Structure

First, we built a baseline dense neural network model and from there we tried to find improvements by testing other hyper parameters. Our baseline model was a Sequential model, containing one input layer, three dense layers each with relu activation functions (sizes 32, 64 and 128 respectively), a dropout(0.3) layer, and finally an output layer of size 1 with a sigmoid activation function. We used the relu activation function for our three central dense layers because it is less computationally intensive and it uses thresholding operations unlike our other options such as sigmoid or tanh.

Our purpose for including a dropout layer was to try to prevent overfitting on the training data. The dropout layer randomly drops some of the neurons in a layer during training and makes the model focus on more robust and generalizable features rather than any one specific neuron.

We chose a sigmoid activation function for our output layer because our target variable was binary. We purposely did not use a softmax activation function for our output layer because it is better suited for multiclass classification tasks.

Next we compiled our model with optimizer= Adam(0.001) (learning rate = 0.001), loss='binary_crossentropy' and metrics=['accuracy']. We used binary_crossentropy because it was the most appropriate loss function given our binary target variable. Also, we did not test using loss functions mean_squared_error or mean_absolute_error because of our binary target variable which made them not applicable. For the metrics variable we chose 'accuracy' because the objective of our project was to achieve the highest accuracy with our model.

Next, we created an early stopping callback with monitor='val_loss' and patience=5. Our reasoning for including early stopping was to prevent overfitting on the training data and to save computing resources because early stopping will stop training after the model has stopped improving. We tested using monitor='val_accuracy' instead of monitor='val_loss' and it had no effect on our model's metrics, so we stuck with monitor='val_loss'.

Finally, we fit our model with 1000 epochs, batch_size=1000, we gave it validation data parameters, and set callbacks=['early_stopping']. Additionally we stored the output from our model's fit function in a variable so that we could use the model's history to plot the model's loss, validation loss, accuracy, and validation accuracy at each epoch, which we will show in the results section of this paper.

This model returned 88% accuracy when tested on a validation data set, but we still tested out other parameters to see if we could improve this score.

The different parameters we tested were different sized dense layers, different learning rates, different dropout sizes and we tried using an RMSprop optimizer instead of Adam. Here are all of the different combinations of parameters that we tested:

1. Dense layer 1 size: 32, dense layer 2 size: 64, dense layer 3 size: 128, dropout size: 0.3, activation function: sigmoid, optimizer: Adam, learning rate: .001, loss function: binary_cross_entropy, early stopping monitor: 'val_loss', patience: 5.
2. Keeping everything else constant, except: Dense layer 1 size: 128, dense layer 2 size: 128, dense layer 3 size: 128.
3. Keeping everything else constant, except: Dense layer 1 size: 256, dense layer 2 size: 256, dense layer 3 size: 256.
4. Keeping everything else constant, except: Dense layer 1 size: 512, dense layer 2 size: 512, dense layer 3 size: 512.

This is where our model performance peaked. We used the parameters in step four as our final model parameters. At this point, we would have tested larger dense layers past 512, but our GPUs did not have enough memory. *Also, when building and fitting this model, when using my GPU on overclocked settings, I received an error saying that I was trying to access an inaccessible memory location. To fix this, I ran with my GPU on the default settings (the GPU in this instance is a NVIDIA GeForce GTX 1660 Ti 6GB).

Here are more different combinations we tested before concluding that our best parameters were the ones from list number 4 (above):

5. Keeping everything else constant, except: Dense layer 1 size: 256, dense layer 2 size: 256, dense layer 3 size: 256, learning rate: 0.01.
6. Keeping everything else constant, except: Dense layer 1 size: 256, dense layer 2 size: 256, dense layer 3 size: 256, learning rate: 0.1.

7. Keeping everything else constant, except: Dense layer 1 size: 256, dense layer 2 size: 256, dense layer 3 size: 256, learning rate: 0.1.
8. Keeping everything else constant, except: Dense layer 1 size: 512, dense layer 2 size: 512, dense layer 3 size: 512, learning rate: 0.001, dropout size: 0.5.
9. Keeping everything else constant, except: Dense layer 1 size: 512, dense layer 2 size: 512, dense layer 3 size: 512, learning rate: 0.001, dropout size: 0.1.
10. Keeping everything else constant, except: Dense layer 1 size: 512, dense layer 2 size: 512, dense layer 3 size: 512, learning rate: 0.001, dropout size: 0.2.
11. Keeping everything else constant, except: Dense layer 1 size: 512, dense layer 2 size: 512, dense layer 3 size: 512, learning rate: 0.001, optimizer: RMSprop.
12. Keeping everything else constant, except: Dense layer 1 size: 512, dense layer 2 size: 512, dense layer 3 size: 512, learning rate: 0.01, optimizer: RMSprop.
13. Keeping everything else constant, except: Dense layer 1 size: 512, dense layer 2 size: 512, dense layer 3 size: 512, learning rate: 0.1, optimizer: RMSprop.
14. Keeping everything else constant, except: Dense layer 1 size: 512, dense layer 2 size: 512, dense layer 3 size: 512, learning rate: 0.0001, optimizer: RMSprop.

When we made a model using the RMSprop optimizer instead of the Adam optimizer, it achieved an accuracy that was slightly worse than our model that used the Adam optimizer (0.1% lower accuracy). So in the end we kept using the Adam optimizer.

Here is a visualization of our final model layer structure:

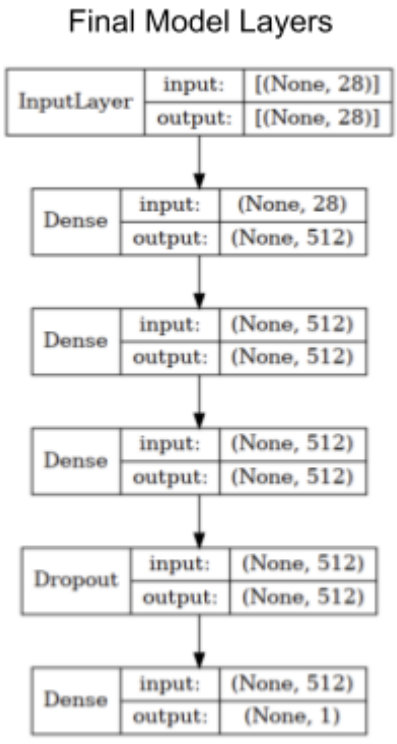

Figure 1 - Final Model Layers

Using an Optimal Threshold for Predicted Probabilities vs. the Default 0.5 Threshold

We thought that since our sigmoid function output layer outputs an array of predicted probabilities, we could use that to our advantage by finding an optimal prediction threshold and increase the performance of our model. This proved to not be useful in this project specifically because the mean optimal threshold we received from our model was 0.5 (the same as the default classification threshold used with sigmoid functions). Because of this, we decided to scrap the code for finding the optimal threshold and we just used a prediction threshold of 0.5.

This problem may not always occur given different data sets so we suggest that, if you are using a sigmoid activation function output layer in a dense neural network, you should attempt to find an optimal threshold and use that to build a model to see if your results improve or not from your baseline 0.5 threshold model.

Results

Figure 2 (below) is a graph showing our final model's training loss and validation loss at each epoch. Looking at the training loss, it consistently decreased which is a good sign. This shows that our model did not overfit on the training data. We ended at about a 0.245 training loss. For our validation loss, it decreased up until about epoch 17, with its lowest value being at about 0.257. We used validation loss for our early stopping metric and this can be seen in this graph. For 5 epochs, epochs 17 to 22, our validation loss did not decrease, so our model stopped training there.

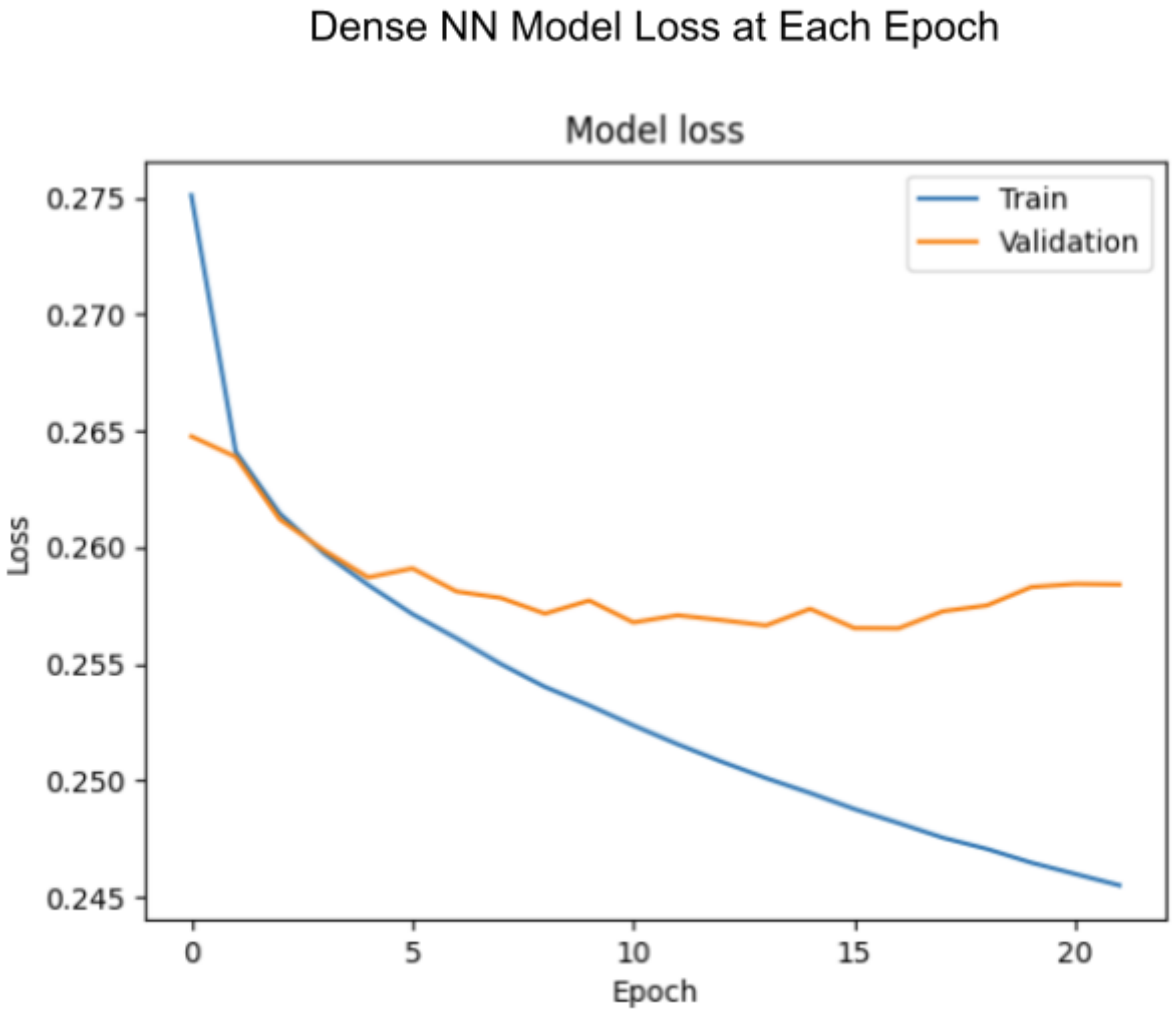

Figure 2 - Loss at Each Epoch

Figure 3 (below) shows our final model's accuracy at each epoch. The best validation accuracy our model achieved was 0.885. This occurred at about the same time that the validation loss stopped decreasing in the previous chart (at about epoch 17). After about epoch 17, our validation accuracy did not increase anymore while our training accuracy continued to increase. This shows that we were overfitting on the training data, and our early stopping did its job and stopped us before we overfit too much.

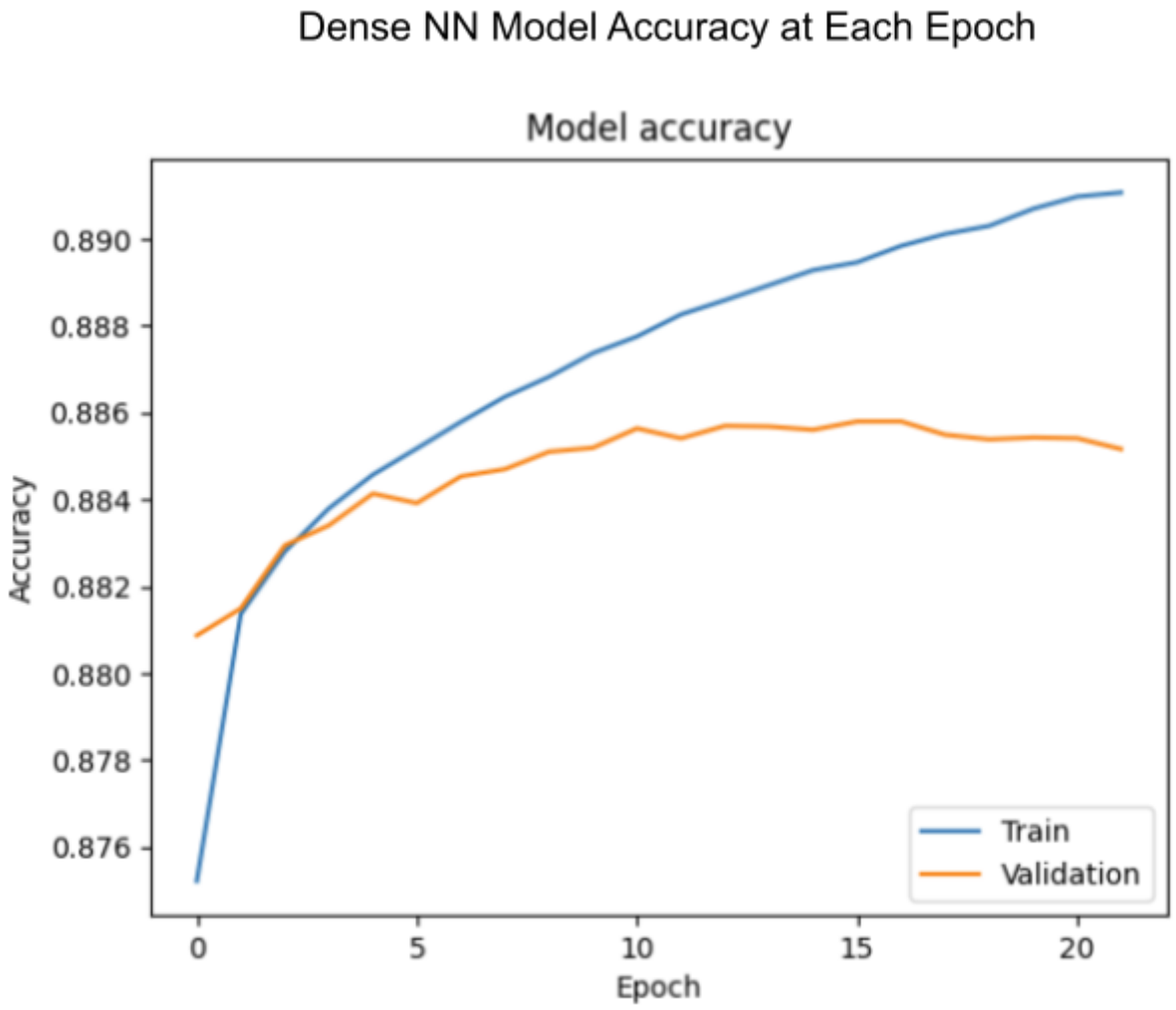

Figure 3 - Accuracy at Each Epoch

Figure 4 (below) shows the metrics for our final model. Just to be clear these metrics were obtained using validation data created during our train test split. Our model's final validation accuracy was 0.885, precision: 0.866, recall: 0.912, and f1_score: 0.888.

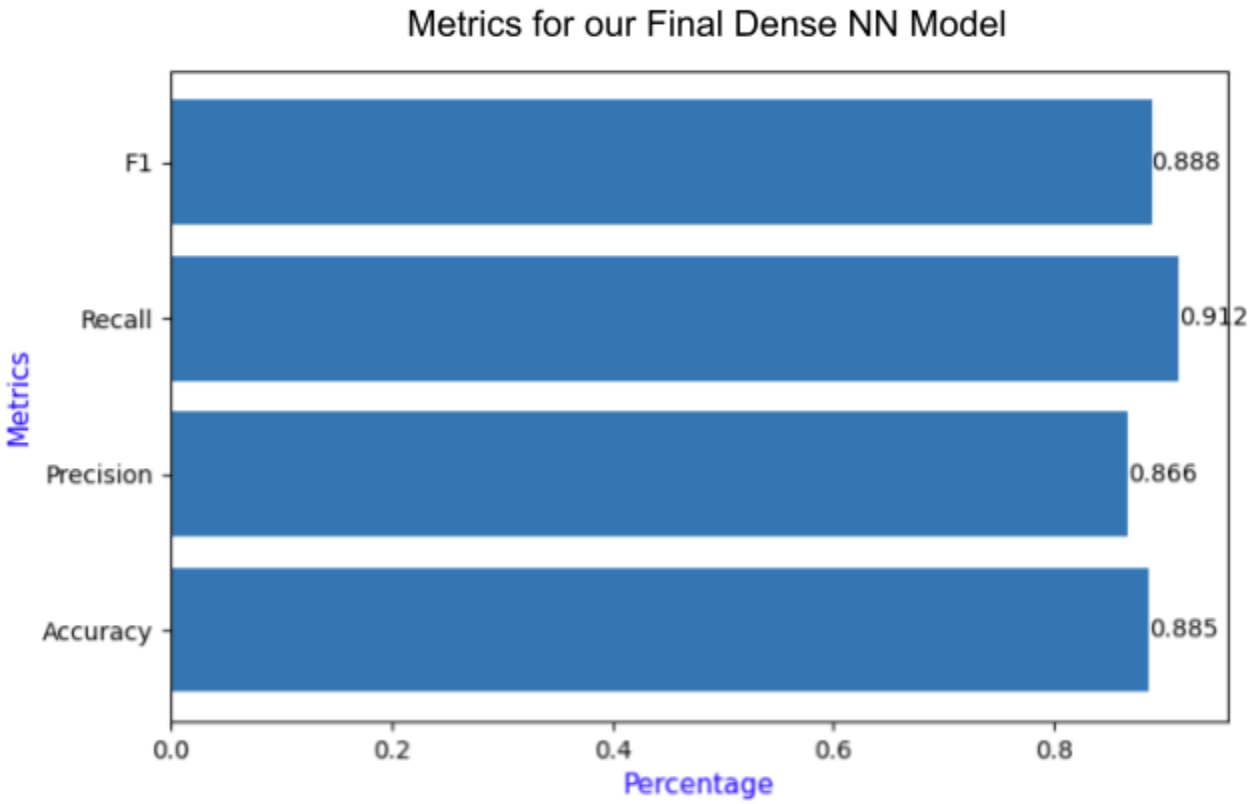

Figure 4 - Final Model Metrics

Figure 5 (below) is a confusion matrix for our final model. Our model accurately predicted label 0 (598,118 / 699,368) times and accurately predicted label 1 (641,478 / 700,632) times. We had 101,250 false positive classifications of label 0 compared to the 59,154 false negative classifications of label 1. That explains why our precision is on the lower side compared to our other metrics (see the metrics in Figure 4). In the end, our recall (0.912) was higher than our precision (0.866) because we had fewer false negative classifications than false positive classifications.

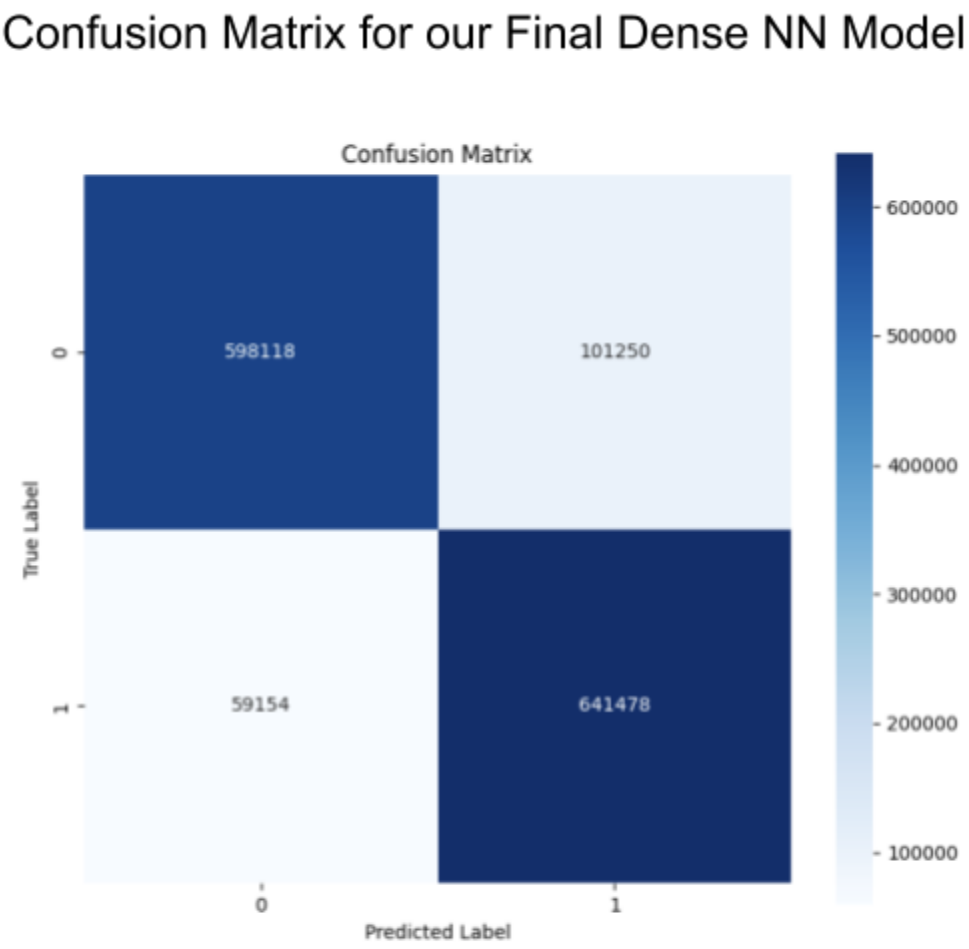

Figure 5 - Final Model Confusion Matrix

Conclusion

In conclusion, our final model was a Sequential model, containing one input layer of size 28, three dense layers each sized 512 with relu activation functions, a dropout(0.3) layer, and an output layer of size 1 with a sigmoid activation function. We used an Adam optimizer with a learning rate of 0.001, loss function of binary_cross_entropy, early stopping monitor of 'val_loss' and patience of 5.

Our final model obtained a validation accuracy of 88.5%, precision of 86.6%, recall of 91.2%, and an f1_score of 88.8%.

Appendix A

Link to main script:
https://github.com/Abillelatus/QTW-Case-Studies/blob/main/CaseStudy_6/MSDS-7333-CaseStudy-6.py

Link to full CaseStudy_6 full contents:
https://github.com/Abillelatus/QTW-Case-Studies/tree/main/CaseStudy_6