Case Study 6:
Detecting the Existence of a New Particle

Kebur Fantahun, Eli Kravez, Halle Purdom

March 28, 2022

Introduction

In this study, a large dataset is provided to train a model that will predict the existence of a new particle. For this case, it is a binary classification task predicting either detection of the particle or no detection of the particle. This dataset is very large in size, which introduces challenges that arise when the memory and computational resources cannot accommodate the size. A dense neural net will be built to predict particle detection with the goal of finding the model structure with the highest accuracy.

Methods

Data Preparation

In this dataset, there are 28 numeric attributes including f0, f1, …, f26 and a mass attribute. The target is a binary integer that is 0 if there is no detection of the new particle and 1 if the new particle was detected. The data file itself is 4.82 GB with 7,000,000 rows of data. To find the amount of rows in the dataset, the csv package in python was used to open the data file and count each row using a for loop.

To use this data in a neural network, the data has to be streamed in through **batches**, rather than finiloading the entire dataset into memory. To do this, tensorflow is used to dynamically train the model in batches so that only those batches are referenced at a time, rather than all 7 million rows of data. Google Colab was used in this project, with the data stored and referenced from Google Drive and a GPU used to run the notebook.

Because the dataset had to be streamed in batches to train the model, there were challenges in standardizing the data as well as splitting the data into training and testing sets. The data batch is drawn from the data then packed into a MapDataset type from tensorflow. Because the dataset could not be referenced all at once, the data could not be standardized all together. This introduced the question of whether or not standardizing the data in batches would introduce any

bias into the data that standardizing as a whole would not. In addition, there is a shuffling data option when drawing the batches from the data, however this is only within each batch and would not refer to the entire dataset.

Based on this Batch Normalization document [1], the **batch normalization** from the tensorflow package was used in order to attempt to solve this issue with standardization in batches. Following the document, batch normalization was used on each layer after the first input layer and in all following layers until the last output layer. This should also increase the training runtime of the neural network.

In terms of creating a **train test split** of the entire dataset, this then posed a problem because the data again could only be referenced in batches. Due to the batches, the split was made on each particular batch of the data with an 80/20 split using the take() and skip() functions on the packed dataset for the training and validation set respectively. While shuffling would be ideal for this split, any shuffling would interfere with the take() and skip() functions not dividing the data correctly since the entries changed in position.

Model Development

In the neural network, there were many different parts of the structure that had to be optimized including layer size, layer activation function, optimizer, loss function, batch size, number of epochs, and early stopping point. Because of the large dataset and time it takes to run each version of the model, not every possible combination of the parameters could be tested.
For the initial layers of the neural network, the relu activation function was used. In the final layer of the neural network, the sigmoid activation was used because of the binary classification output that was needed. For the optimizer, adam was chosen rather than SGD in order to minimize runtime as adam generally converges faster than SGD. The loss function for the network was chosen as binary cross entropy because it is the default loss function for binary classification tasks. Different amounts of layers, as well as different layer sizes were also tested to find the most accurate model.

To find the best model for this study, the validation loss was monitored to see when the model was finished learning. As each layer is added in the neural network, accuracy of the model improves. However, adding layers to the network increases runtime, so to get an accurate model while also being able to train the models with manageable runtimes, six dense layers were used.

Results

After model development, the final binary classification sequential neural network model had an input layer, five dense layers with batch normalizations after each layer, and a final output layer

as seen in the below table. The optimizer used was adam, and the loss function binary cross entropy loss, with a batch size of 32, and epochs equal to 1000. To know the model is finished training, the model's **early stopping** patience parameter was set at three, and the monitor was validation loss. Tensorboard was used to monitor the model performance.

Layer	Neurons	Activation
Input	28	relu
BatchNormalization		relu
Dense	500	relu
BatchNormalization		relu
Dense	400	relu
BatchNormalization		relu
Dense	300	relu
BatchNormalization		relu
Dense	200	relu
BatchNormalization		relu
Dense	100	relu
BatchNormalization		relu
Dense (Output)	1	sigmoid

After running the final neural network on the testing data, the following performance metrics were calculated:

Accuracy	Precision	Recall
0.8775	0.8613	0.8999

Conclusion

In conclusion, a neural network was built to complete the binary classification task of predicting the existence of a new particle. The large dataset size introduced many data preprocessing

challenges in terms of standardization and the train/test split. The neural network built was a dense neural network containing input, dense, normalization, and output layers. The model was trained by monitoring the validation loss, and the final accuracy was 0.8775. In future work, more models could be explored that may take more computing power and runtime like increasing the number of layers or changing the optimizer to SGD.

References

1. https://www.baeldung.com/cs/batch-normalization-cnn#:~:text=Batch%20Norm%20is%20a%20normalization,learning%20rates%2C%20making%20learning%20easier

In [ ]:
```
import tensorflow as tf

from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
```

In [ ]:
```
!pip install -U -q PyDrive
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials
# Authenticate and create the PyDrive client.
auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)
```

In [ ]:
```
link = 'https://drive.google.com/file/d/1hJGgFSvtRsNREGPVjkSTLZqOzNc0okgv/view?u

fluff, id = link.split('=')
print (id) # Verify that you have everything after '='
# https://drive.google.com/file/d/1hJGgFSvtRsNREGPVjkSTLZqOzNc0okgv/view?usp=sha
```

```
sharing
```

In [ ]:
```
import pandas as pd
```

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

New Section

In [ ]:
```
path = "/content/drive/MyDrive/all_train.csv"
#df_bonus = pd.read_csv(path)
```

In [ ]:
```
import tensorflow as tf
import numpy as np

#data_file ='G://all_train.csv/all_train.csv'

data_file = path
temp_data_set = tf.data.experimental.make_csv_dataset(
    data_file,
    batch_size=1000,
    num_epochs = 1,
    label_name='# label',
    ignore_errors=True,)

def pack(features, label): # why tf decided to return feature in individual colu
```

```
  return tf.stack(list(features.values()), axis=-1), tf.cast(label, tf.int32)

# how to normalize this ? !!!!! Standard Scaler
packed_dataset = temp_data_set.map(pack)

for features, labels in packed_dataset.take(1):
  print(features.shape) # we have 28 features
  #print(labels.unique())
  print(np.unique(labels.numpy()))
  print(len(features.numpy()))
  print()
  print(labels.numpy())
```

```
(1000, 28)
[0 1]
1000

[1 1 0 1 0 1 1 1 1 0 0 0 1 0 0 0 0 0 1 1 1 0 0 1 1 0 1 1 1 0 0 0 0 0 1 0 0
 0 0 1 0 0 1 1 1 1 0 1 0 0 0 1 1 1 0 0 1 1 1 1 0 1 0 0 0 0 1 0 1 0 0 1 0 1
 0 1 0 1 1 0 0 1 0 1 1 0 1 0 0 1 1 1 0 1 1 1 0 1 1 0 0 0 1 0 0 0 1 1 0 0 1
 1 0 0 0 1 1 1 1 0 0 0 1 1 0 1 0 1 0 0 1 1 1 1 0 0 1 1 1 1 1 0 1 0 1 1 1 1
 0 0 0 0 0 1 0 1 1 1 1 1 0 0 1 0 0 0 1 0 1 1 1 0 1 1 1 1 0 1 0 1 0 1 0 0 1
 1 1 0 0 0 0 1 0 0 0 0 1 1 1 1 1 1 0 0 0 0 1 1 1 0 0 1 1 1 0 1 1 0 1 1 1 1
 1 1 0 1 0 1 0 0 1 1 0 1 1 0 1 1 0 0 0 1 0 1 1 0 1 0 0 1 1 1 0 1 1 0 1 1 0
 1 1 1 1 0 0 1 0 0 1 0 1 0 0 0 1 1 0 1 0 0 1 1 0 1 0 1 0 0 0 0 0 0 0 1 0 0
 1 0 1 1 0 1 0 0 1 1 1 1 0 1 0 1 1 0 1 1 0 0 1 0 0 1 1 0 1 1 0 1 0 0 1 0 1
 0 1 0 0 1 0 0 0 1 0 0 1 1 0 0 0 1 1 0 1 0 0 0 0 1 1 0 0 1 1 0 0 0 0 0 0 1
 0 0 0 0 1 1 0 1 0 0 0 0 0 1 0 0 0 1 0 1 0 1 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1
 1 1 1 0 0 0 1 1 0 0 1 1 1 1 1 0 0 1 0 0 0 1 1 0 0 0 0 0 1 0 1 1 1 0 0 0 0
 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 1 0 0 0 0 1
 0 0 1 0 0 0 0 1 1 0 0 0 1 0 1 1 0 1 0 1 1 0 0 0 1 0 0 0 1 1 1 1 0 1 0 0 0
 0 0 1 1 1 0 1 0 1 0 1 0 1 1 0 1 0 0 0 1 1 1 1 0 0 0 1 1 0 0 0 0 0 0 0 1 1
 0 0 1 1 1 0 1 1 0 0 0 0 1 1 1 0 1 0 1 1 0 0 0 1 1 0 1 0 1 1 0 0 1 1 1 0 0
 0 1 1 1 1 1 0 0 1 1 0 1 0 0 0 1 0 1 1 0 1 0 1 1 1 1 0 0 1 0 1 1 0 0 1 1 0
 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 1 0 0 0 1 0 0 1 0 1 1 1 1 0 1 0 1 1 0 0 1
 1 1 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 1 0 0 1 0 0 1 0 1 0 0 0 1 1 0 0 1 0 0
 1 0 0 1 0 0 1 1 0 0 1 0 0 1 0 1 1 1 1 1 1 0 1 1 1 0 0 0 0 1 1 1 0 0 1 1 0
 1 0 1 1 1 0 1 0 1 0 0 1 1 1 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 1 1 1 0 0 1 1
 0 0 1 1 0 0 0 1 0 0 1 1 0 0 0 1 0 1 0 1 0 0 0 1 1 1 0 1 0 1 1 0 1 1 1 1 0
 1 1 0 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 1 0 1 0 1 1 0 1 1 0 1 1 0 1 0 0 1 0 0
 1 0 0 0 1 0 1 1 0 1 0 0 1 0 0 1 1 0 0 0 0 0 0 0 1 0 1 1 0 0 0 1 1 1 1 1 1
 0 1 0 0 0 0 0 1 1 0 0 1 0 1 0 0 1 1 0 0 1 0 1 1 1 1 0 0 0 0 0 1 0 1 0 1 0
 1 0 1 0 0 1 1 0 0 0 0 0 1 0 1 1 0 0 1 0 0 1 1 1 1 1 1 1 0 1 0 1 0 1 0 0 1
 1 0 1 0 0 0 0 0 0 0 1 0 1 0 0 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 0 1 1 1 0 1 0
 1]
```

In [ ]:
```
packed_dataset
```

Out[ ]:
```
<MapDataset element_spec=(TensorSpec(shape=(None, 28), dtype=tf.float32, name=No
ne), TensorSpec(shape=(None,), dtype=tf.int32, name=None))>
```

In [ ]:
```
from time import time
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.callbacks import TensorBoard
```

In [ ]:
```

```

In [ ]:
```
#model.fit(packed_dataset)  # no validation!!! VERY BAD!!!
```

In [ ]:
```
from tensorflow.keras.layers import  BatchNormalization
my_model = Sequential()
my_model.add(tf.keras.Input(shape=(28,)))
my_model.add(BatchNormalization())
my_model.add(tf.keras.layers.Dense(500, activation='relu'))
my_model.add(BatchNormalization())
my_model.add(tf.keras.layers.Dense(400, activation='relu'))
my_model.add(BatchNormalization())
my_model.add(tf.keras.layers.Dense(300, activation='relu'))
my_model.add(BatchNormalization())
my_model.add(tf.keras.layers.Dense(200, activation='relu'))
my_model.add(BatchNormalization())
my_model.add(tf.keras.layers.Dense(100, activation='relu'))
my_model.add(BatchNormalization())
my_model.add(tf.keras.layers.Dense(1, activation='sigmoid')) # softmax  sigmoid
```

In [ ]:
```
train_size = int(0.8 * 1000)
val_size = int(0.2 * 1000)
#test_size = int(0.15 * 1000)

full_dataset = packed_dataset.shuffle(buffer_size = 1000)
train_dataset = full_dataset.take(train_size)
#test_dataset = full_dataset.skip(train_size)
val_dataset = full_dataset.skip(val_size)
#test_dataset = test_dataset.take(test_size)
```

In [ ]:
```
train_dataset
```

Out[ ]:
```
<TakeDataset element_spec=(TensorSpec(shape=(None, 28), dtype=tf.float32, name=N
one), TensorSpec(shape=(None,), dtype=tf.int32, name=None))>
```

In [ ]:
```
from tensorflow.keras.optimizers import SGD
opt = SGD()
my_model.compile(optimizer='adam', loss = tf.keras.losses.BinaryCrossentropy() ,
```

In [ ]:
```
from tensorflow.keras.callbacks import TensorBoard
from time import time
import datetime

log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
#f"\\content\\drive\\MyDrive\\logs\\{time()}"

tb = TensorBoard(log_dir=log_dir, histogram_freq=1)
%load_ext tensorboard
%tensorboard --logdir logs
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

In [ ]:
```
from tensorflow.keras.callbacks import EarlyStopping
```

```
safety = EarlyStopping(patience = 3, monitor='val_loss')
```

In [ ]:

```
#my_model.fit(packed_dataset.shuffle(buffer_size = 1000).take(train_size), valid
 # add validation set validation_data = packed_dataset.skip(train_size),

history = my_model.fit(packed_dataset.take(train_size), validation_data = packed
```

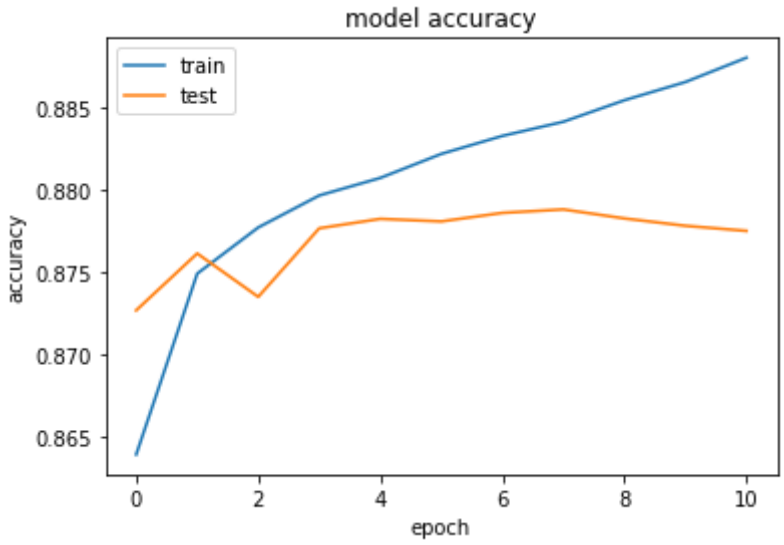
```
Epoch 1/1000
800/800 [==============================] - 391s 485ms/step - loss: 0.2929 - accu
racy: 0.8639 - false_negatives_1: 44448.0000 - false_positives_1: 64429.0000 - a
uc_1: 0.9455 - precision_1: 0.8468 - recall_1: 0.8890 - val_loss: 0.2805 - val_a
ccuracy: 0.8727 - val_false_negatives_1: 392335.0000 - val_false_positives_1: 39
7087.0000 - val_auc_1: 0.9514 - val_precision_1: 0.8721 - val_recall_1: 0.8735
Epoch 2/1000
800/800 [==============================] - 351s 439ms/step - loss: 0.2742 - accu
racy: 0.8749 - false_negatives_1: 40809.0000 - false_positives_1: 59267.0000 - a
uc_1: 0.9525 - precision_1: 0.8585 - recall_1: 0.8981 - val_loss: 0.2729 - val_a
ccuracy: 0.8761 - val_false_negatives_1: 342526.0000 - val_false_positives_1: 42
5461.0000 - val_auc_1: 0.9532 - val_precision_1: 0.8663 - val_recall_1: 0.8895
Epoch 3/1000
800/800 [==============================] - 360s 449ms/step - loss: 0.2692 - accu
racy: 0.8777 - false_negatives_1: 40114.0000 - false_positives_1: 57715.0000 - a
uc_1: 0.9542 - precision_1: 0.8620 - recall_1: 0.8998 - val_loss: 0.2757 - val_a
ccuracy: 0.8735 - val_false_negatives_1: 420111.0000 - val_false_positives_1: 36
4231.0000 - val_auc_1: 0.9530 - val_precision_1: 0.8804 - val_recall_1: 0.8645
Epoch 4/1000
800/800 [==============================] - 424s 529ms/step - loss: 0.2658 - accu
racy: 0.8797 - false_negatives_1: 39460.0000 - false_positives_1: 56812.0000 - a
uc_1: 0.9554 - precision_1: 0.8641 - recall_1: 0.9015 - val_loss: 0.2698 - val_a
ccuracy: 0.8777 - val_false_negatives_1: 348840.0000 - val_false_positives_1: 40
9650.0000 - val_auc_1: 0.9543 - val_precision_1: 0.8704 - val_recall_1: 0.8875
Epoch 5/1000
800/800 [==============================] - 365s 455ms/step - loss: 0.2636 - accu
racy: 0.8807 - false_negatives_1: 39074.0000 - false_positives_1: 56345.0000 - a
uc_1: 0.9561 - precision_1: 0.8651 - recall_1: 0.9024 - val_loss: 0.2697 - val_a
ccuracy: 0.8782 - val_false_negatives_1: 311550.0000 - val_false_positives_1: 44
3379.0000 - val_auc_1: 0.9541 - val_precision_1: 0.8628 - val_recall_1: 0.8995
Epoch 6/1000
800/800 [==============================] - 347s 434ms/step - loss: 0.2610 - accu
racy: 0.8822 - false_negatives_1: 38509.0000 - false_positives_1: 55740.0000 - a
uc_1: 0.9570 - precision_1: 0.8666 - recall_1: 0.9038 - val_loss: 0.2700 - val_a
ccuracy: 0.8781 - val_false_negatives_1: 341224.0000 - val_false_positives_1: 41
4672.0000 - val_auc_1: 0.9543 - val_precision_1: 0.8693 - val_recall_1: 0.8899
Epoch 7/1000
800/800 [==============================] - 364s 454ms/step - loss: 0.2590 - accu
racy: 0.8833 - false_negatives_1: 38264.0000 - false_positives_1: 55110.0000 - a
uc_1: 0.9577 - precision_1: 0.8679 - recall_1: 0.9045 - val_loss: 0.2693 - val_a
ccuracy: 0.8786 - val_false_negatives_1: 302573.0000 - val_false_positives_1: 45
0105.0000 - val_auc_1: 0.9543 - val_precision_1: 0.8614 - val_recall_1: 0.9024
Epoch 8/1000
800/800 [==============================] - 343s 428ms/step - loss: 0.2568 - accu
racy: 0.8841 - false_negatives_1: 38062.0000 - false_positives_1: 54622.0000 - a
uc_1: 0.9584 - precision_1: 0.8691 - recall_1: 0.9050 - val_loss: 0.2690 - val_a
ccuracy: 0.8788 - val_false_negatives_1: 301344.0000 - val_false_positives_1: 45
0029.0000 - val_auc_1: 0.9545 - val_precision_1: 0.8615 - val_recall_1: 0.9028
Epoch 9/1000
800/800 [==============================] - 363s 454ms/step - loss: 0.2544 - accu
racy: 0.8855 - false_negatives_1: 37541.0000 - false_positives_1: 54096.0000 - a
uc_1: 0.9592 - precision_1: 0.8703 - recall_1: 0.9063 - val_loss: 0.2709 - val_a
ccuracy: 0.8783 - val_false_negatives_1: 330114.0000 - val_false_positives_1: 42
4657.0000 - val_auc_1: 0.9542 - val_precision_1: 0.8671 - val_recall_1: 0.8935
Epoch 10/1000
```

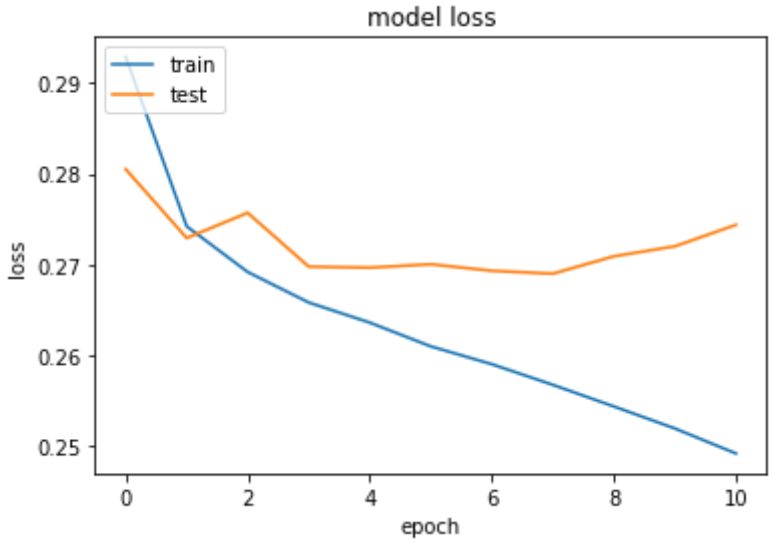
```
800/800 [==============================] - 343s 428ms/step - loss: 0.2520 - accu
racy: 0.8866 - false_negatives_1: 37395.0000 - false_positives_1: 53355.0000 - a
uc_1: 0.9600 - precision_1: 0.8719 - recall_1: 0.9066 - val_loss: 0.2720 - val_a
ccuracy: 0.8778 - val_false_negatives_1: 330011.0000 - val_false_positives_1: 42
7537.0000 - val_auc_1: 0.9538 - val_precision_1: 0.8663 - val_recall_1: 0.8936
Epoch 11/1000
800/800 [==============================] - 363s 453ms/step - loss: 0.2492 - accu
racy: 0.8880 - false_negatives_1: 36761.0000 - false_positives_1: 52807.0000 - a
uc_1: 0.9608 - precision_1: 0.8732 - recall_1: 0.9082 - val_loss: 0.2744 - val_a
ccuracy: 0.8775 - val_false_negatives_1: 310267.0000 - val_false_positives_1: 44
9201.0000 - val_auc_1: 0.9533 - val_precision_1: 0.8613 - val_recall_1: 0.8999
```

In [ ]:

```
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

```
dict_keys(['loss', 'accuracy', 'false_negatives_1', 'false_positives_1', 'auc_
1', 'precision_1', 'recall_1', 'val_loss', 'val_accuracy', 'val_false_negatives_
1', 'val_false_positives_1', 'val_auc_1', 'val_precision_1', 'val_recall_1'])
```





In [ ]: