DEEP LEARNING - PROJECT

Detect targets in radar signals

https://www.kaggle.com/c/detect-targets-in-radar-signals

Henning Erik Raimar

Group 507

Artificial Intelligence,

Faculty of Mathematics & Informatics

University of Bucharest

Challenge description: "In autonomous driving, radar systems play an important role in detecting targets such as other vehicles on the road. Radars mounted on different cars can interfere with each other, degrading the detection performance. In this context, participants are asked to employ deep learning methods to detect (count) targets (vehicles, people, obstacles, etc.) under radar interference."

Kaggle public leaderboard score (25%) : 0.52872

Kaggle private leaderboard score (75%): 0.53478

Local validation dataset score: 0.53

Best solution:

At the data preprocessing part I got the best score by preprocessing the datas ImageDataGenerator from keras.preprocessing.image package. In the ImageDataGenerator I configured a scaler of 1/255, shear_range of 0.2, zoom_range of 0.2 along with horizontal_flip.

After I configured ImageDataGenerator I created the dataset for training and validation with function "flow_from_dataframe" with a batchSize of size 32, because I found it optimal, between other variants I tried, in a training time and precision point of view. I also tried with batch size of 8 and 16, but the training time was too high. I also kept the image sizes to their original of 128 x 55.

I chose to use the ImageDataGenerator along with flow_from_dataframe after I tried more preprocessing steps manually, like scaling, flipping, zoom, reshaping and converting to grayscale and I had to deal with a lot of errors until I managed to train the model and get a good accuracy. I used the generator because, from my point of view, it was the simplest version to implement those kinds of image preprocessing steps.

As expected, the image preprocessing steps, implemented improved the results by a good margin, helping the model to associate better the labels.

To bring pixels in the same range I divided every pixel by 255 and this contributed to the improvement in Kaggle Public Leaderboard from 0.3 to 0.53.

After scaling the training samples, I trained a Convolutional Neural Network with 4 Convolutional layers, each with MaxPooling and BatchNormalization, with neurons 64, 128, 256 and 512 neurons. After those

Convolutional layers I added 3 Dense layers of 256, 512 and 1024 neurons, preceded by a Dropout of 0.4 to prevent overfitting.

During multiple tests performed, I found out that on my neural network, BatchNormalization was a must have increasing all my previous results from scores below 0.25.

For the output layer I used softmax as activation function, after I compared the results with sigmoid and tanh. The summary of the model can be observed in Fig 1.

To validate the results on my local machine I used the K-Fold Cross Validation procedure from sklearn library, performing 5 splits and I found out that it was a good method because I didn't have very big differences between training accuracy and validation accuracy. (0.65 in the model training and 0.53 for validation datas, the same result I scored on Kaggle too).

Also to prevent overfitting I used early stopping callbacks for minimum and maximum, configured to stop the training process if there was not any significant improvement in the last 7 epochs (> 0.005).

To understand better the behaviour of the model I printed everytime the Confusion Matrix to analyse the predictions. (Fig 2)

```
In [7]: model.summary()
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_8 (Conv2D)           (None, 127, 54, 64)       832

 max_pooling2d_8 (MaxPooling  (None, 63, 27, 64)       0
 2D)

 batch_normalization_8 (Batc  (None, 63, 27, 64)       256
 hNormalization)

 conv2d_9 (Conv2D)           (None, 62, 26, 128)       32896

 max_pooling2d_9 (MaxPooling  (None, 31, 13, 128)      0
 2D)

 batch_normalization_9 (Batc  (None, 31, 13, 128)      512
 hNormalization)

 conv2d_10 (Conv2D)          (None, 29, 11, 256)       295168

 max_pooling2d_10 (MaxPoolin  (None, 14, 5, 256)       0
 g2D)

 batch_normalization_10 (Bat  (None, 14, 5, 256)       1024
 chNormalization)

 conv2d_11 (Conv2D)          (None, 12, 3, 512)        1180160

 max_pooling2d_11 (MaxPoolin  (None, 6, 1, 512)        0
 g2D)

 batch_normalization_11 (Bat  (None, 6, 1, 512)        2048
 chNormalization)

 flatten_2 (Flatten)         (None, 3072)              0

 dropout_6 (Dropout)         (None, 3072)              0

 dense_8 (Dense)             (None, 256)               786688

 dropout_7 (Dropout)         (None, 256)               0

 dense_9 (Dense)             (None, 512)               131584

 dropout_8 (Dropout)         (None, 512)               0

 dense_10 (Dense)            (None, 1024)              525312

 dense_11 (Dense)            (None, 5)                 5125

=================================================================
Total params: 2,961,605
Trainable params: 2,959,685
Non-trainable params: 1,920
_________________________________________________________________
```

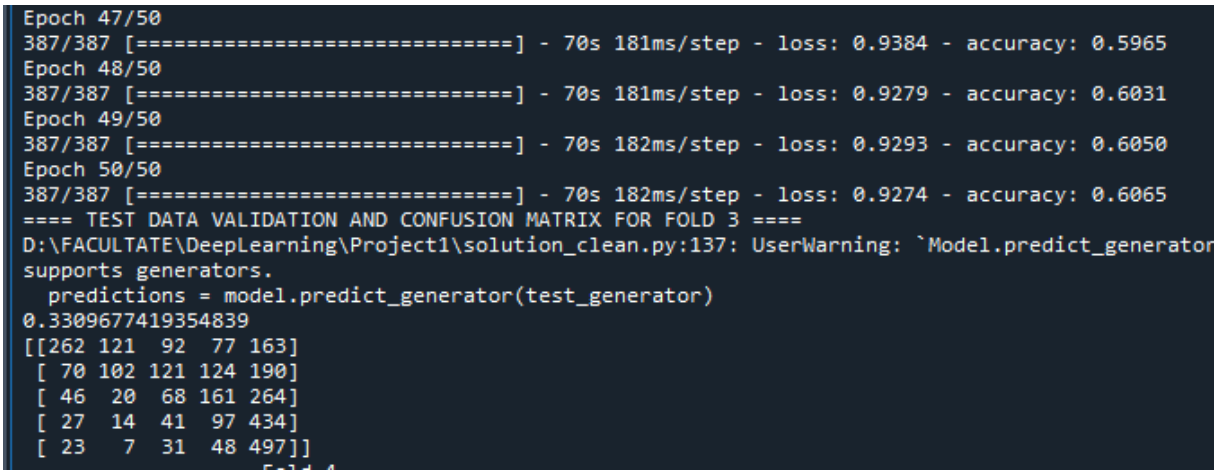
Fig 1

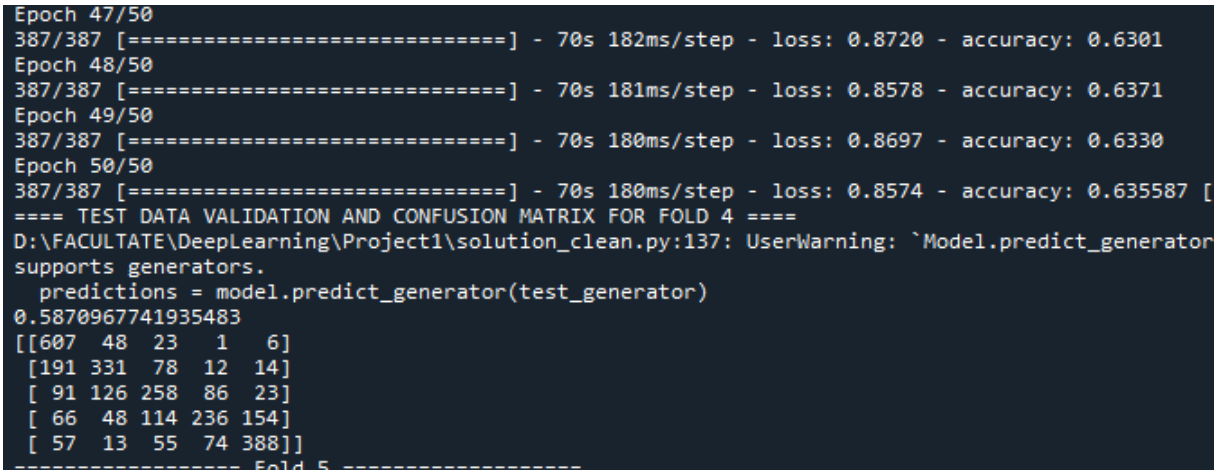
```
Epoch 47/50
387/387 [==============================] - 71s 183ms/step - loss: 0.7858 - accuracy: 0.6673
Epoch 48/50
387/387 [==============================] - 72s 185ms/step - loss: 0.8049 - accuracy: 0.6617
Epoch 49/50
387/387 [==============================] - 71s 185ms/step - loss: 0.8555 - accuracy: 0.6358
Epoch 50/50
387/387 [==============================] - 71s 183ms/step - loss: 0.8272 - accuracy: 0.6531
==== TEST DATA VALIDATION AND CONFUSION MATRIX FOR FOLD 5 ====
D:\FACULTATE\DeepLearning\Project1\solution_clean.py:137: UserWarning: `Model.predict_generator` is deprecated and will be
removed in a future version. Please use `Model.predict`, which supports generators.
  predictions = model.predict_generator(test_generator)
0.5325806451612903
[[542  88  54   6   9]
 [153 292 175   7  15]
 [ 76  88 333  38  18]
 [ 25  40 262 211  69]
 [ 13  25 139 149 273]]

In [6]:
```

Fig 2

```
Epoch 47/50
387/387 [==============================] - 70s 181ms/step - loss: 1.0827 - accuracy: 0.5288
Epoch 48/50
387/387 [==============================] - 70s 182ms/step - loss: 1.0830 - accuracy: 0.5314
Epoch 49/50
387/387 [==============================] - 70s 182ms/step - loss: 1.0919 - accuracy: 0.5255
Epoch 50/50
387/387 [==============================] - 71s 183ms/step - loss: 1.0879 - accuracy: 0.5288
==== TEST DATA VALIDATION AND CONFUSION MATRIX FOR FOLD 1 ====
D:\FACULTATE\DeepLearning\Project1\solution_clean.py:137: UserWarning: `Model.predict_generator
supports generators.
  predictions = model.predict_generator(test_generator)
0.49032258064516127
[[717  16   4   3   3]
 [286 252  35   4   5]
 [185 159 169  61  11]
 [140  92 114 172  65]
 [106  44  84 163 210]]
```

```
Epoch 47/50
387/387 [==============================] - 69s 179ms/step - loss: 0.9910 - accuracy: 0.5726
Epoch 48/50
387/387 [==============================] - 70s 180ms/step - loss: 0.9899 - accuracy: 0.5745
Epoch 49/50
387/387 [==============================] - 70s 180ms/step - loss: 0.9983 - accuracy: 0.5691
Epoch 50/50
387/387 [==============================] - 70s 180ms/step - loss: 0.9758 - accuracy: 0.5856
==== TEST DATA VALIDATION AND CONFUSION MATRIX FOR FOLD 2 ====
D:\FACULTATE\DeepLearning\Project1\solution_clean.py:137: UserWarning: `Model.predict_generator
supports generators.
  predictions = model.predict_generator(test_generator)
0.48193548387096774
[[573 145  21   7   2]
 [162 213 172  38   7]
 [105  70 153 191  55]
 [ 71  42  75 189 201]
 [ 53  29  52 108 366]]
================== Fold 3 ==================
```





Other solutions:

- **CNN (2 Dense layers) + Just grayscale preprocessing: (0.30)**

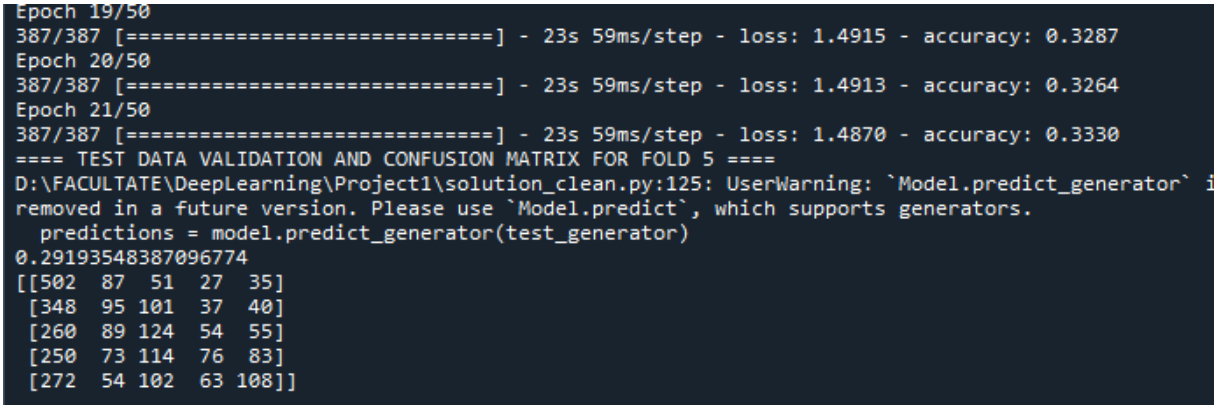
  The second best solution I implemented and submitted was with the same algorithms as above, with the difference that I used built the neural network with 2 Dense Layers of (256, 512) instead of 3 Dense Layers of (256, 512, 1024). Also on this solution I didn't implement any other preprocessing method, except the grayscale conversion.

- **ANN (3 Dense layers) + MaxPooling and BatchNormalization + Keras Generator Preprocessing: (0.29)**

  I implemented an ANN model which performed worse than the Convolutional Neural Network, with 3 Dense layers of 32, 64 and 128 neurons, with a Dropout of 0.2 between the layers. After the first layer I

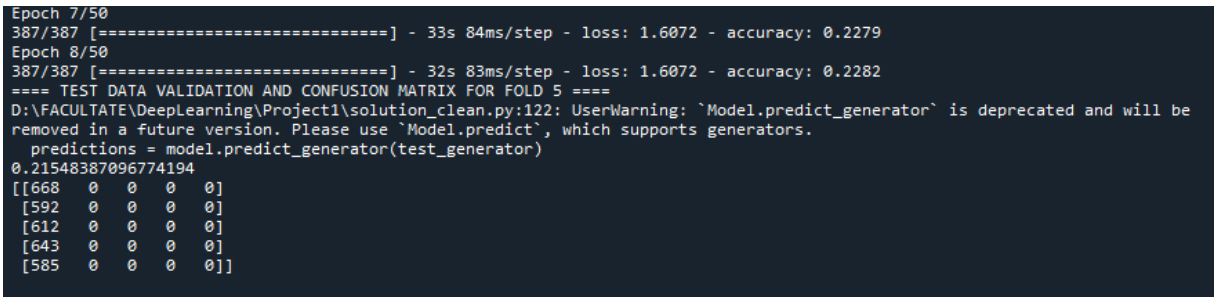
added MaxPooling and I also added BatchNormalization between after the other layers. I implemented the ANN model with a lot less neurons because the training time was too big for a configuration similar to the CNN described above. Although the result was not a significant improvement (0.33 on training, 0.29 on validation), compared to the below model, describing the ANN without MaxPooling and BatchNormalization, by analyzing the confusion matrix I concluded that this model was a step forward being one of the first that predicted different labels.

In the below image I attached the confusion matrix along with the accuracy validation metrics.



- **ANN (3 Dense layers) + Keras Generator Preprocessing: (0.23)**

  I tried to implement an ANN model which performed poorly with 3 Dense layers of 32, 64 and 128 neurons, with a Droput of 0.2 between the layers. With this solution I managed to get just a 0.25 score on the training and 0.23 on the validation set, the model being early stopped on epoch 8. Also, after analysing the confusion matrix I realised that the model predicted just one class.



- **CNN without BatchNormalization + Grayscale preprocessing: (0.20)**

Another solution I tried was to implement the CNN described above, but without the BatchNormalization layer. During this experiment I found out by analyzing the confusion matrix that without this layer, my model predicted a single label in most of the cases, with some small exceptions.

- **MultiLayered Perceptron - No preprocessing: (0.18)**

To have a starting point I tried to implement an MLP Classifier model without any preprocessing steps that scored just 0.17-0.18 on Kaggle and 0.23 on my local environment.

LINK GITHUB: https://github.com/Katzuno/DetectTargetsInRadarSignals