Splitting datasets and metrics examples

Neural Networks for Health Technology Applications

22.2.2020

Sakari Lukkarinen

Helsinki Metropolia University of Applied Sciences

Contents

- Import libraries
- Read the original file names
- Create own dataset folders
  - Shuffle the data
- Copy images and make data generators
- Training and wathcing the elapsed time
- Finding the labels and predicted values
- Confusion matrix and classification results
  - ROC curve analysis
  - Decision threshold analysis

Import libraries

Import libraries

In[14]:

```
%pylab inline
import time # used for timing the training
import os # operating system commands
import shutil # High level operations for files
import random # Random generators
import pandas as pd # Pandas data read, maybe not needed in this case??
import tensorflow as tf # Tensorflow
from tensorflow.keras import layers, models, optimizers # Neural network stuff
from tensorflow.keras.preprocessing.image import ImageDataGenerator # For image processing with keras
from tensorflow.keras.metrics import Accuracy, FalseNegatives, FalsePositives # Specific metrics
from sklearn.metrics import classification_report, confusion_matrix, roc_curve # For final results analysis
print('Tensorflow version = ', tf.__version__) # Check the version
```

```
Populating the interactive namespace from numpy and matplotlib
Tensorflow version =  2.1.0
```

Read the original file names

In[7]:

```
# Original training files can be found here.
orig_dir = '/kaggle/input/chest-xray-pneumonia/chest_xray/chest_xray/train'

# List only all jpeg-files (images)
normal_images = [x for x in os.listdir(os.path.join(orig_dir,'NORMAL')) if x.endswith(".jpeg")]
pneumonia_images = [x for x in os.listdir(os.path.join(orig_dir,'PNEUMONIA')) if x.endswith(".jpeg")]

# Show the statistics
N_NORMAL = len(normal_images)
N_PNEUMONIA = len(pneumonia_images)
TOTAL = N_NORMAL + N_PNEUMONIA

print('Original training images:')
print(f'{N_NORMAL:5d} normal cases')
print(f'{N_PNEUMONIA:5d} pneumonia cases')
print(f'{TOTAL:5d} totally.')
```

```
Original training images:
 1341 normal cases
 3875 pneumonia cases
 5216 totally.
```

Create own directories

In[8]:

```
# Own libraries, will be deleted when restarted
train_dir = './train'
valid_dir = './validation'
test_dir = './test'
all_dirs = [train_dir, valid_dir, test_dir]
try:
    for d in all_dirs:
        os.mkdir(d)
        os.mkdir(os.path.join(d,'NORMAL'))
        os.mkdir(os.path.join(d,'PNEUMONIA'))
except:
    pass
print('Training directory = ', train_dir)
print(os.listdir(train_dir))
```

```
Training directory =  ./train
['NORMAL', 'PNEUMONIA']
```

Suffle the data

In[9]:

```
# Shuffle the lists in random order
random.shuffle(normal_images)
random.shuffle(pneumonia_images)
```

Copy original images to dataset folders

In[10]:

```
# Copy the original images into train, validation, and test directories

# Normal test images
for fname in normal_images[:500]:
    src = os.path.join(orig_dir, 'NORMAL', fname)
    dst = os.path.join(test_dir, 'NORMAL', fname)
    shutil.copyfile(src, dst)

# Normal validation images
for fname in normal_images[501:1001]:
    src = os.path.join(orig_dir, 'NORMAL', fname)
    dst = os.path.join(valid_dir, 'NORMAL', fname)
    shutil.copyfile(src, dst)

# Normal training images
for fname in normal_images[1002:]:
    src = os.path.join(orig_dir, 'NORMAL', fname)
    dst = os.path.join(train_dir, 'NORMAL', fname)
    shutil.copyfile(src, dst)
```

```
# Pneumonia test images
for fname in pneumonia_images[:500]:
    src = os.path.join(orig_dir, 'PNEUMONIA', fname)
    dst = os.path.join(test_dir, 'PNEUMONIA', fname)
    shutil.copyfile(src, dst)

# Pneumonia validation images
for fname in pneumonia_images[501:1001]:
    src = os.path.join(orig_dir, 'PNEUMONIA', fname)
    dst = os.path.join(valid_dir, 'PNEUMONIA', fname)
    shutil.copyfile(src, dst)

# Pneumonia training images
for fname in pneumonia_images[1002:]:
    src = os.path.join(orig_dir, 'PNEUMONIA', fname)
    dst = os.path.join(train_dir, 'PNEUMONIA', fname)
    shutil.copyfile(src, dst)
```

Data generators

In[11]:

```
TS = (150, 150) # Image size
BS = 128 # Batch size

print('Training:')
tg = ImageDataGenerator(rescale=1./255)
train_generator = tg.flow_from_directory(
    train_dir,
    target_size = TS,
    batch_size = BS,
    class_mode = 'binary')

print('Validation:')
devg = ImageDataGenerator(rescale=1./255)
dev_generator = devg.flow_from_directory(
    valid_dir,
    target_size = TS,
    batch_size = BS,
    shuffle = False,
    class_mode = 'binary')

print('Testing:')
testg = ImageDataGenerator(rescale=1./255)
test_generator = devg.flow_from_directory(
    test_dir,
    target_size = TS,
    batch_size = BS,
    class_mode = 'binary')
```

```
Training:
Found 3212 images belonging to 2 classes.
Validation:
Found 1000 images belonging to 2 classes.
Testing:
Found 1000 images belonging to 2 classes.
```

In[12]:

```
# Check the indices
train_generator.class_indices
```

Out[12]:

```
{'NORMAL': 0, 'PNEUMONIA': 1}
```

Model

- Your own CNN model comes here ….

Training and watching the elapsed time

```
E = 3 #Number of epochs

# Start the clock
t_start = time.time()

print('Training ...', end='')
h = model.fit_generator(
      train_generator,
      steps_per_epoch = None,
      verbose = 0,
      epochs = E,
      validation_data = dev_generator,
      validation_steps = None
      )

# Check the time and calculate the elapsed time and time per epoch
t_end = time.time()
t_elapsed = t_end - t_start
t_per_epoch = t_elapsed/E

print('Done.')
print(f'Time elapsed = {t_elapsed:.0f} seconds.')
print(f'Time per epoch = {t_per_epoch:.2f} seconds.')

# Save the model
model.save('case_2_run_007.h5')
print('Model saved.')
```

```
Training ...Done.
Time elapsed = 131 seconds.
Time per epoch = 43.57 seconds.
Model saved.
```

Check the keys and use them

```
h.history
```

Out[23]:

```
{'loss': [0.40175620756885627, 0.37634724265388947, 0.23620304895813113],
 'accuracy': [0.0, 0.0003113325, 0.0],
 'false_negatives_2': [116.0, 118.0, 51.0],
 'false_positives_2': [327.0, 309.0, 209.0],
 'val_loss': [1.0419244868680835, 0.6296937242150307, 1.3800700112478808],
 'val_accuracy': [0.0, 0.0, 0.0],
 'val_false_negatives_2': [0.0, 3.0, 0.0],
 'val_false_positives_2': [500.0, 395.0, 466.0]}
```

Seems that the index for false_positives and false_negatives is increasing whenever you rerun the model code (!?)

```
# Extract the metrics and loss
hh = h.history
acc = hh['accuracy']
acc_v = hh['val_accuracy']
fn = hh['false_negatives_2']
fn_v = hh['val_false_negatives_2']
fp = hh['false_positives_2']
fp_v = hh['val_false_positives_2']
loss = hh['loss']
loss_v = hh['val_loss']
epochs = arange(len(loss)) + 1
```

How to avoid increasing indexing in metrics

Create a list of metrics beforehand

ถ้าจะให้ดีใช้ “acc”

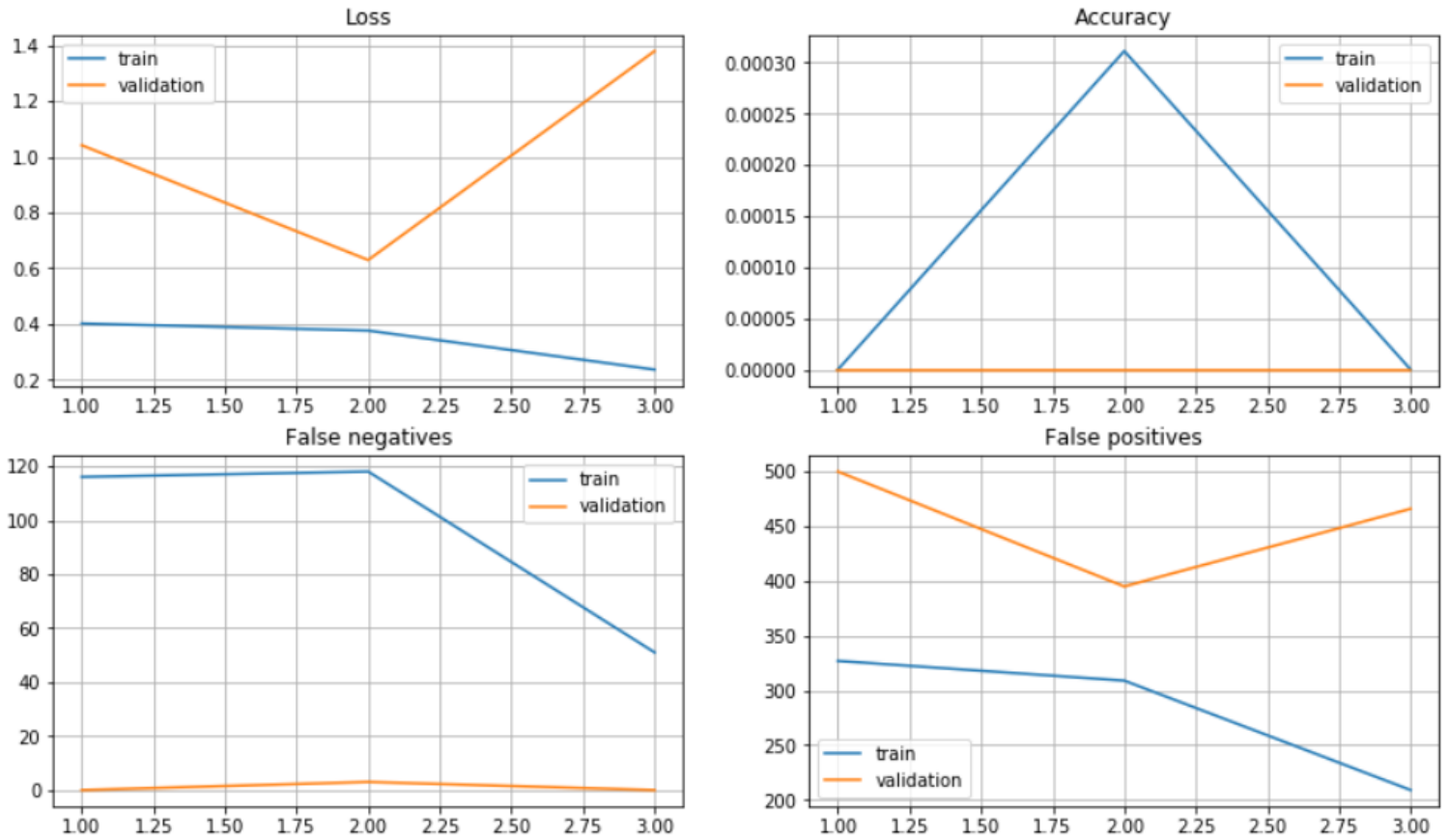
```
my_metrics = [Accuracy(), FalseNegatives(), FalsePositives()]
```

Then use the list in model compilation step:

```
# Use normal RMSprop optimizer, but calculate extra metrics
model.compile(loss = 'binary_crossentropy',
              optimizer = optimizers.RMSprop(),
              metrics = my_metrics)

# Don't show the summary. It can be read above.
# model.summary()
```

Training results (bad)



Finding the labels and predictions

```
devg = ImageDataGenerator(rescale=1./255)
dev_generator = devg.flow_from_directory(
    valid_dir,
    target_size = TS,
    batch_size = BS,
    shuffle = False,
    class_mode = 'binary')
```

Note: You need to have shuffle = False for the validation generator!

Otherwise, the images are shuffled and you don't know which is the right label.

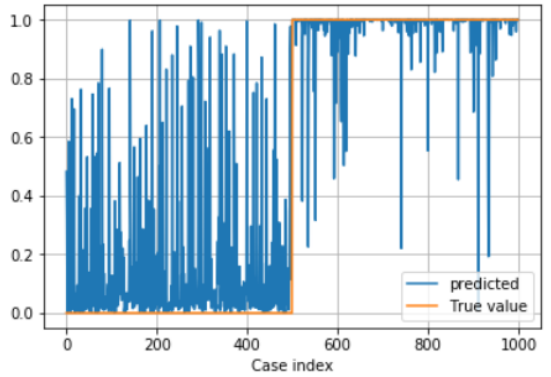
In[17]:

```
# Find the labels
labels = dev_generator.classes

# Predict the results
predicted = model.predict_generator(dev_generator).flatten()
```

+ Code + Markdown

```
# Plot the predicted and true labels as lists
plot(predicted, label = 'predicted')
plot(labels, label = 'True value')
legend()
xlabel('Case index')
grid()
```



Aim is the predicted values are as close as possible for true values (=labels).

Confusion matrix and classification report

In[28]:

```
print('Confusion matrix (machine learning way):')
cm = confusion_matrix(labels, predicted > 0.5)
print(cm)
tn, fp, fn, tp = confusion_matrix(labels, predicted > 0.5).ravel()
print('Confusion matrix (medicine way):')
print(array([[tp, fn], [fp, tn]]))
```

```
Confusion matrix (machine learning way):
[[446  54]
 [ 10 490]]
Confusion matrix (medicine way):
[[490  10]
 [ 54 446]]
```

+ Code + Markdown

```
# Calculate the classification report
cr = classification_report(labels, predicted > 0.5, target_names = ['Normal (0)', 'Pneumonia (1)'])
print(cr)
```

```
              precision    recall  f1-score   support

   Normal (0)       0.98      0.89      0.93       500
Pneumonia (1)       0.90      0.98      0.94       500

     accuracy                           0.94      1000
    macro avg       0.94      0.94      0.94      1000
 weighted avg       0.94      0.94      0.94      1000
```

This is how you can convert the confusion matrix into format usually shown in medicine literature.

Remember: recall = *sensitivity,* recall for Pneumonia (1) = Sensitivity.

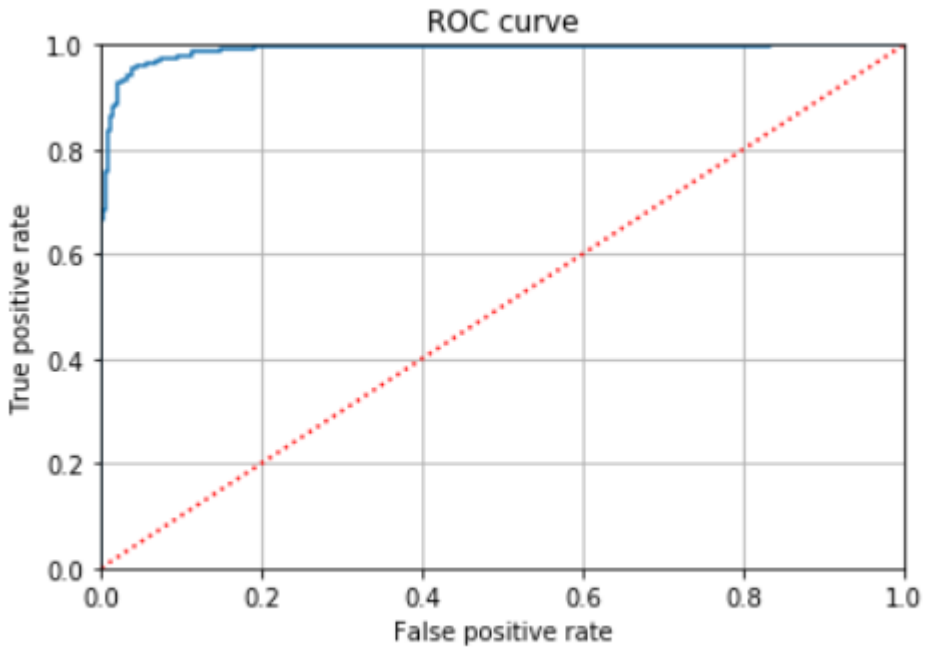
Q: **How can you get the *specificity* from the classification report?**

ROC curve analysis

In[29]:

```
# Calculate the ROC curve analysis
fpr, tpr, thresholds = roc_curve(labels, predicted, pos_label = 1)
```

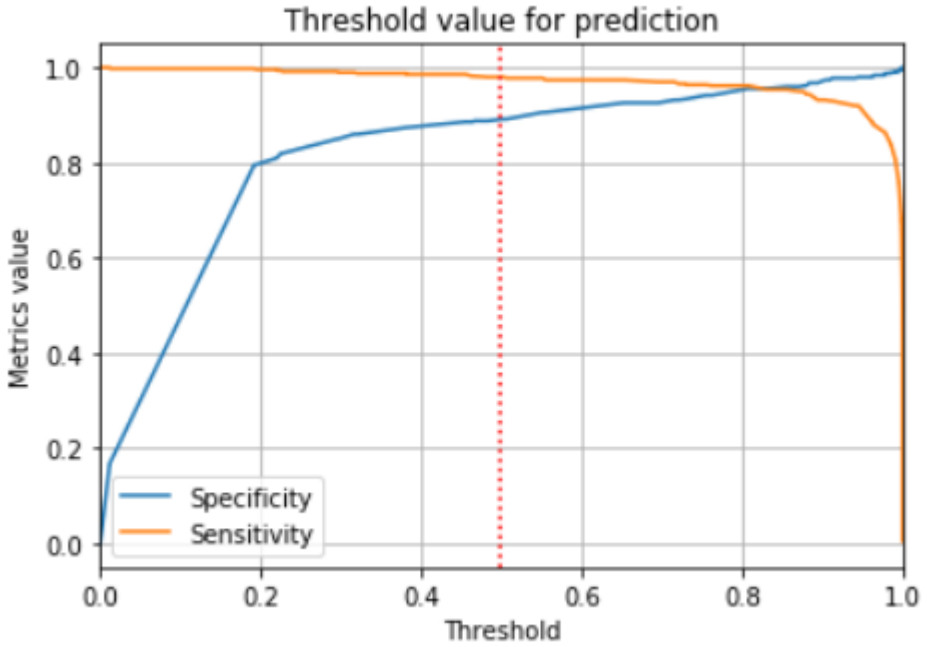
```
# Show the ROC curve
plot(fpr, tpr)
plot([0, 1], [0, 1], 'r:')
xlabel('False positive rate')
ylabel('True positive rate')
title('ROC curve')
xlim([0, 1])
ylim([0, 1])
grid()
```



Q: What does the ROC curve tell about the classifier?

Threshold analysis

```
# Find the best threshold
plot(thresholds, 1 - fpr, label = 'Specificity')
plot(thresholds, tpr, label = 'Sensitivity')
axvline(0.5, color = 'red', linestyle = ':')
xlim([0, 1])
title('Threshold value for prediction')
xlabel('Threshold')
ylabel('Metrics value')
legend()
grid()
```



Q: How would you select the theshold for the prediction?

Q: Is 0.5 (see classification report and confusion matrix code, predicted > 0.5) the optimal value for making decisions?

Q: How sensitivity and specificity are usually tuned for *diagnostic screening purposes*?