M.Sc. in Data Science

- Course: Deep Learning
- Instructor: Prof. P Malakasiotis
- Author: Andreas Stavrou (f3352120)
- Assignment 2

Table of Contents

INTRODUCTION AND SCOPE 3

DATA INGESTION 3

EXPERIMENTAL SETUP 4

OPTIMIZER, LOSS FUNCTION AND METRICS 4

DEEP NEURAL ARCHITECTURES 5

CONVOLUTIONAL NEURAL NET (CNN) ARCHITECTURE 5

TRANSFER LEARNING 7

CHOOSING THE RIGHT ARCHITECTURE 7

ENSEMBLE LEARNING 8

DATA AUGMENTATION 9

EXPERIMENTAL RESULTS 10

CONCLUSIONS 10

Introduction and scope

Submit a report (approx 10 - 15 pages, PDF format) for the following machine learning project. Explain briefly in the report the architectures that you used, how they were trained, tuned, etc. Describe challenges and problems and how they were addressed. Present in the report your experimental results and demos (e.g., screenshots) showing how your code works. Do not include code in the report, but include a link to a shared folder or repository (e.g. in Dropbox, GitHub, Bitbucket) containing your code. The project will contribute 60% to the final grade.

Given a study containing X-Ray images, build a deep learning model that decides if the study is normal or abnormal. You must use at least 2 different architectures, one with your own CNN model (e.g., you can use a model similar to the CNN of the previous project) and one with a popular CNN pre-trained CNN model (e.g., VGG-19, ResNet, etc.). Use the MURA dataset to train and evaluate your models.

Data Ingestion

A rather non-trivial problem ingesting data, when dealing with large datasets such as *MURA v1.1*, stems from the size of the dataset itself. Initial efforts to load the dataset and split into *train* / *validation* / *test* sets by using recursive directory access and combining retrieved images with class labels proved futile.

Fortunately, *TensorFlow* provides users with the mechanics to stream entire datasets as *Python* generators by using methods of class "*ImageDataGenerator"*, from the "*tf.keras.preprocessing.image*" package, specifically method "*flow_from_dataframe"*. As a bonus, the class provides users with the ability to automatically generate augmentations of the dataset (e.g. random image rotation, random image flipping etc.).

Since input data is images, it is worth mentioning that the input shape of images was chosen at (*224*, *224*) with *3* channels, in "*channels_last"* format. Choosing the input image dimensions is a balance between performance (training speed, avoiding OOM errors etc.) and classification accuracy. Shape (*224*, *224, 3*) was initially chosen because of requirements imposed by the *ResNet* architecture, when the default classification head is chosen as output and we adhered to this shape thereafter.

Note: a major shortcoming of "*ImageDataGenerator"* in *TensorFlow* is that it does not take image dimensions ratio into account, when performing resizing. In effect, images produced by the generator are distorted. We researched the issue and the only solution was found in the "monkey-patch" solution mentioned in 2. Unfortunately, a significant amount of time was lost until we were able to diagnose, validate and correct the problem.

Experimental setup

The strategy to achieve the task at hand can be broken down in two approaches:

1. Classification of *normal* and *abnormal* cases per X-ray type (*XR_HAND*, *XR_WRIST* etc.)
2. Classification of *normal* and *abnormal* on the entire dataset.

Optimizer, loss function and metrics

Considering the task at hand and considering the *MURA* contest, we realize that the appropriate measure to be used is *Cohen's Kappa* metric. We were able to evaluate the trained models on *Cohen's Kappa* during the model evaluation stage.

Considering that we are dealing with a problem of classification with unbalanced classes, we chose *ROC-AUC* as the measure to use because the curve balances the class sizes, as it will only actually select models that achieve false positive and true positive rates that are significantly above random chance, which is not guaranteed for accuracy. *AUC* measures how *true positive rate* (*r ecall*) and *false positive rate* trade off. More importantly, *AUC* is the evaluation of the classifier as the threshold varies over all possible values. It is a broader metric, testing the quality of the internal value that the classifier generates and then compares to a threshold. It is easy to achieve *99%* accuracy on a data set where *99%* of examples are in the same class. Thus, a perfect predictor gives an *ROC-AUC* score of *1.0* while a predictor which makes random guesses has an *ROC-AUC* score of *0.5*.

Since we are loading our target classes as *categoricals,* loss is computed based on the *categorical cross-entropy* loss function.

For our optimizer, we chose *Adam* since it has been demonstrated that it is the best for the task at hand.

During the model evaluation stage, we acquired the *loss*, *accuracy* and *AUC metrics*, measured on the model's performance on the test set. Furthermore, we measured each model's predictive power by acquiring their respective *accuracy* (for completeness, this measure is not very informative for our task, as mentioned previously)*, precision, recall, F1-score and Cohen Kappa* metrics.

Deep neural architectures

Convolutional Neural Net (CNN) architecture

Convolutional layers apply learned filters to input images in order to create feature maps that summarize the presence of input features. For my *CNN* architecture, I experimented with a variety of stacked *convolutional layers* as well as experimented with the following modes of building the progression of kernel and filter sizes:

- Filter: 32,64,128,256,512
- Conv. Layers: 2-6
- Convolutions per layer: 1-2
- Pooling: Avg, Max
- Batch normalization: True, False
- Dropout: 0.2-0.3

For each layer, the non-linearity applied is *ReLU* which has been shown to work well for image classification tasks.

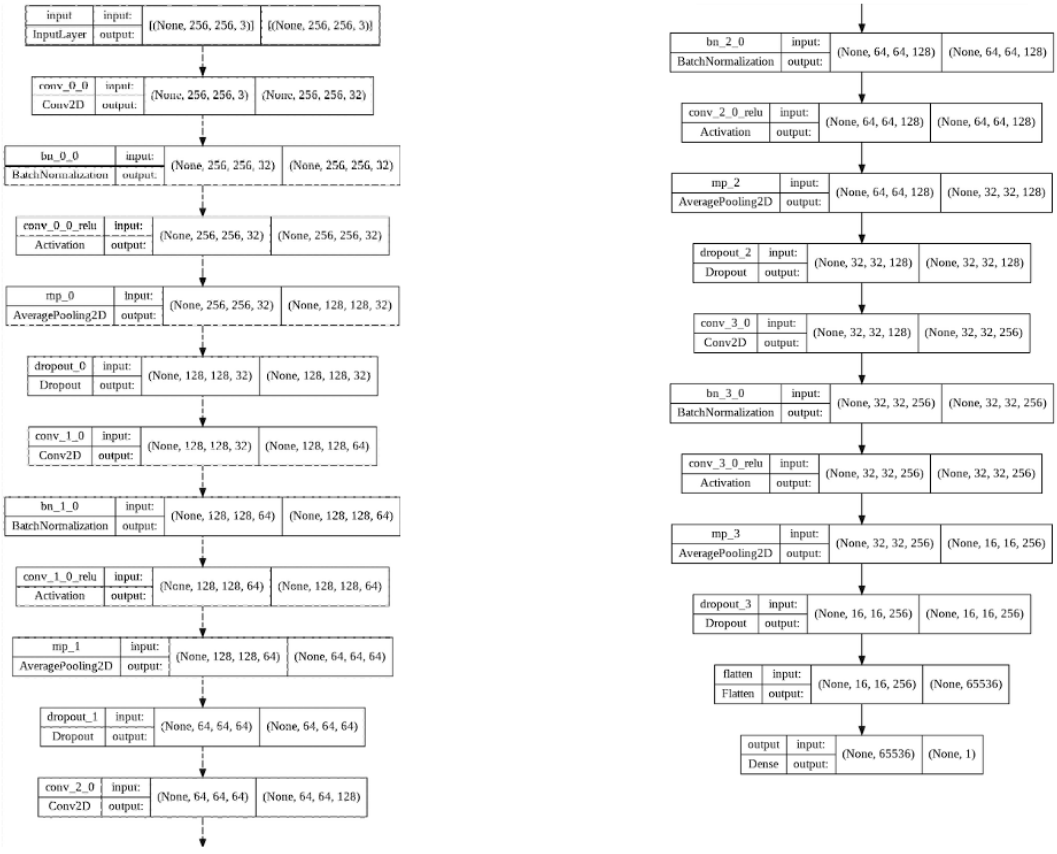
Further to this, *pooling operations* (either *max* / *average* pooling) were applied on top of each *convolutional layer. Pooling* provides a way to summarize feature maps. There is debate on which operation to employ as empirical findings postulate that the operator selector is input domain specific. For example, using the *MNIST* dataset as input benefits from choosing a min pooling operation, as the background of images in the dataset is white (max pixel value is *255*) and the actual feature values (digits) are darker (min pixel value is 0). For our *CNN*, we parameterised the type of pooling operation to be employed as well as kept the parameters *pool size* to *(2, 2)*, strides to *(1, 1)* and padding to be "*same*". Due to lack of a *min pooling* layer in *Keras* we were unable to test with a min pooling strategy. However, we found through experimentation that *average pooling* works best for the task.

Next, we applied two forms of *dropout*: *spatial dropout* after each *convolutional layer* and a *dropout layer* on the parameters of the fully connected layer, both with a *rate* of *0.2*. We applied *dropout* so

as to avoid overfitting to the training data. For the same reason, we applied a *L2* penalty on the *kernel* and *activation parameters* of the *dense layer* by utilizing functions parameters *kernel_regularizer = tf.keras.regularizers.l2(0.01)* and *activity_regularizer = tf.keras.regularizers.l2(0.01)*, respectively.

Finally, the output layer of the *CNN* is *2* - unit with a *softmax* function.

After heavy manual and iterative experimentation, we concluded that the best architecture, based on the ROC-AUC and Kappa metrics on the test set and also on its balance between performance and training time, was the model which is depicted below:



As we will see in the sequel, the performance of this architecture is weak when compared to a *DenseNet*. A further comment is that increasing the number of *convolutional layers* does not seem to provide the gains expected, probably due to a vanishing gradients problem which the *DenseNet* architecture alleviates by employing redundant connections between *convolutional layers*, thus facilitating gradient propagation and allowing for greater depth.

Transfer Learning

Choosing the right architecture

Choosing the right architecture was a task of trial and error. Initially I tried 15 models with different architectures. In this section I am going to elaborate the results for the 5 most stable architectures that also yield the best Cohen Kappa scores.

DenseNet with pre-trained images on ImageNet

I experimented a with a couple of classification heads on top of the frozen weights of *DenseNet*, specifically,

- A moderately deep *MLP*, comprising *2* fully connected hidden layers of size *1024* and *512* respectively, with dropout layers in between and a *s oftmax* output layer.
- A single *softmax* activation layer.

Early experiments showed that the latter achieved better performance, so we devoted our resources and effort to training the latter. Note that we implemented code so as to avoid re-training all *DenseNet* layers, which were kept "frozen". Training was performed only on the classification head that we chose.

Resnet152v2

The next architecture that was implemented was the Resnet152V2 architecture. ResNet50V2 is a modified version of ResNet50 that performs better than ResNet50 and ResNet101 on the ImageNet dataset. In ResNet50V2, a modification was made in the propagation formulation of the connections between blocks. On top of the Resnet body I placed just one dense layer with a sigmoid activation function.
All Resnet layers were kept frozen, but the last 5. The above number was a product of trial and error. As callbacks, I used early stopping and reduce learning rate on plateau as the paper suggested with *patience* equal to 1 and *factor* of 0.1.

VGG16

The next architecture that was implemented was the VGG16 architecture which is a 16 layers deep convolutional neural network. I loaded the ImageNet weights on it and let is train for 5 epochs to save time. On top of the VGG16 body I placed a dense layer with a sigmoid activation function.

All VGG16 layers were kept frozen, but the last one. The above number was a product of trial and error.
As callbacks, I used early stopping and reduce learning rate on plateau as the paper suggested with *patience* equal to 1 and *factor* of 0.1.

Inception V3

The next architecture that was implemented was the inception V3 model. The model itself is made up of symmetric and asymmetric building blocks, including convolutions, average pooling, max pooling, concatenations, dropouts, and fully connected layers. Batch normalization is used extensively throughout the model and applied to activation inputs.
I trained all the layers after 270 which was a product of trial and error.
As callbacks, I used early stopping and reduce learning rate on plateau as the paper suggested with *patience* equal to 1 and *factor* of 0.1.

Xception

The last architecture that was implemented was the Xception model which is deep convolutional neural network architecture inspired by Inception. Is used same number of parameters but more efficient use of those and in general has better results than the first.
I trained all the layers after 120 which was a product of trial and error.
As callbacks, I used early stopping and reduce learning rate on plateau as the paper suggested with *patience* equal to 1 and *factor* of 0.1.

Ensemble learning

Ensemble learning is a process by which models trained on a subset of the problem space are combined in order to strengthen their predictive power. Essentially, one employs the "*divide-and-conquer*" technique on the problems space, picking classifiers that behave best on the subset of data, according to their selected metric, and applying cooperative schemes on the set of acquired models for making predictions. The flavors of such schemes, such as being more appropriate for the task at hand, include the following:

Mixture of experts: a set of classifiers trained on specific classes is (e.g. *XR_HAND*, *XR_WRIST* etc.) are linearly combined to produce the output predicted class, effectively giving an average of the groups of model's predictions.

Voting: the majority output class of each model is selected as the actual prediction.

Stacking: stacking works by layering models in a way like collecting the features from models and feeding them into other models to get better results.

For this task, I used voting in which the 5 best architectures were called in to decide on the outcome. Those architectures were Densenet201, Resnet152V2, VGG16, Inception V3 and Xception and were selected based on the test loss. In every body part, the ensemble model managed to outperform the best scored model by 2-5%.

Data augmentation

Data augmentation involves the "*artificial variations* (of input data) *to mimic the appearance of future test samples that deviate from the training manifold*" .

A very nice feature of *TensorFlow*'s *ImageDataGenerator* is the ability to create altered versions of the input images during model training. Exploiting this capability, we chose to create random augmentations of the input dataset so as to investigate potential performance enhancements in our best model.

For this task, we chose to perform *data augmentation* on our best model only and verify whether this technique would augment our model's predictive capabilities. We chose to randomly vary the data augmentation parameters in the set (*rotation*: *0-30* degrees, *flip horizontally*: *True* / *False*, *flip vertically*: *True* / *False*) .

All my experiments with data augmentation managed to achieve a significant increase in accuracy (2%-5% increase) and decrease in loss when compared to the same experiments without data augmentation.

Experimental results

As far as the results goes, I compiled the list below. The results below depict our models when trained on a subset of the data, and most specific on the forearm:

Architecture	Test loss	Test accuracy	Test Kappa
Densenet201	0.53257	0.78738	0.57493
Densenet169	0.67100	0.77741	0.55504
VGG16	0.50226	0.76744	0.53527
InceptionV3	0.51300	0.78738	**0.57514**
Xception	0.55279	0.73090	0.46237

Implementation

You can find the code for the above implementation here.

Conclusions and future work

Transfer learning took the world of ML by storm a few years ago by allowing the re-use of feature weights of pre-trained models, trained in large scale and on massive datasets, to be used effectively for downstream ML tasks. Our own research attests to the vast improvement of key metrics when transfer learning is employed, specifically the use of *Inception and Xception* architecture for the image classification task.

Regarding future work, I would like to assess how the model works under the hood and compile a more complete error analysis by analyzing what the model learns and how it makes its decisions in order to serve better data representations to help the model achieve better results in the task.