Project Report: X-Ray Machine Learning

Team 6

May 2019

Our task, set by TPP, was to produce an algorithm capable of detecting abnormalities in X-ray samples. We took the natural machine learning approach, and made use of convolutional neural networks. We experimented with more complex structures involving residual learning [1, 2] before deciding on a simpler architecture. To interpret our models we made use of Class Activation Mapping [3]. Throughout the process, we implemented more memory efficient versions of standard libraries in order to train and run our models with our limited hardware resources.

1 Libraries and hardware

All of our work was written in the Python programming language (Python Software Foundation, `https://www.python.org/`). To process the data we made extensive use of the NumPy scientific computing library (NumPy Developers, `https://www.numpy.org`). Neural networks were written in Tensorflow [4], making extensive use of its Keras implementation. Our code can be found on GitHub (`https://github.com/Cicero17/MURA`).

The models were trained using two NVIDIA GeForce GTX 1070 GPUs, making use of Tensorflow's CUDA support. These provided reasonable training times for the most part, although we found that 8GB of memory was sometimes insufficient for the complexity of the models we were attempting to use.

2 Dataset

To help with our task, we were given access to the MURA dataset [5], containing 14,863 labelled musculoskeletal studies.

The data set is partitioned into training data (8,280 normal and 5,177 abnormal studies) and validation data (661 normal and 538 abnormal studies). Images are of seven types, corresponding to different parts of the body: elbow, finger, forearm, hand, humerus, shoulder and wrist. Images are given as PNG files of varying dimension. CSV files are provided containing the path to each image and a corresponding label.

3 Data extraction and preprocessing

Our first task was to create separate CSV files for each category of X-ray, split by training and validation. This was accomplished with `generate_paths.py` in our codebase.

The next task was to create functions to extract the images from their folders and convert them to NumPy arrays in order to use them for our models. We decided to convert all images to grayscale, as the extra dimensionality of RGB images was unnecessary. Our extract functions provide great flexibility: first of all, one can extract the training and testing data as NumPy arrays for any of the seven categories or all at once; secondly, they allow the user to choose to reshape the images. Given that originally images had different dimensions, this step is necessary as neural networks have a fixed input size. We also created functions for shuffling data, and for extracting data partitioned by patient.

The user also has the option to standardize data. Our initial approach to making standardization possible was to use built-in functions from NumPy. However, we soon discovered that these functions are far too memory inefficient to run on such large datasets. Thus, we decided to write our own functions for standardization, trading speed for memory efficiency. The way we standardize is by computing the ‘mean’ and the ‘deviation’ image of the train data given. The way we calculate the ‘mean’ image is by creating an auxiliary array, that is recycled, containing the pixels values the images have at a given position. The mean of this auxiliary array gives the entry of our ‘mean’ at the corresponding pixel. Computing the ‘deviation’ image consists of a similar approach, computing the standard deviation instead of the mean. We improved this process by making use of the pre-computed mean. Computing standard deviation requires summing many very small numbers, so to reduce the impact of floating point errors, we implemented the Kahan summation technique [6]. With the mean and standard deviation images computed, from each training or testing image we subtract the mean image and divide by the standard deviation image pixel-wise.

4 Models related functions

In order to improve and evaluate our models and manipulate them more efficiently we designed specific functions for the following purposes: validation, computing confusion matrices, saving, copying and loading models.

We produced two validation functions. The first one is for classic validation. In classic validation, we split the shuffled training data, basing on a given proportion, into two groups: one for training and one for validation. The default value for the proportion is 0.8, but the user has the ability to modify it. The validation score of the model is given by its accuracy on the validation data. The second validation function is k-fold cross validation. In this type of validation, we split the shuffled training data into k folds. We then choose each fold separately as a validation fold and train the model on the other $k-1$ folds. We evaluate our model on the validation fold and thus obtain a score. The average of all these k scores gives us the result of our k fold cross validation. Note that when training and testing, we are had to use ’fresh’ (not trained) copies of our model. Because of that we created a `copy_model` function which is based on the `clone_model` function in Keras. Both validation functions give user the option of selecting the batch sizes, number of epochs and whether or not to include class weights when training.

We observed that in MURA there is a bias towards the number of normal, negative images. This is the reason we considered necessary to implement a class weight function that would force models to assign the same level of importance to both classes. Given the medical nature of the classification, we gave user the option to include a false positive bias b with default value 0.0, although we did not consider it necessary and never used it ourselves. For the positive class, the formula is $1 + \frac{N+b}{P}$ and for the negative class it is $1 + \frac{P}{N}$, where P and N are the number of positive and negative examples in the training data, respectively.

To evaluate a model, computing its accuracy on the test set is not enough, because we needed to be sure that the model is truly learning from our training data. With the MURA data, a program that would simply classify all images as negative would have an accuracy of 0.61 on training data, which is a good validation score. In light of this, we decided to create functions that would provide us the accuracy and the confusion matrices our models produce. We made two versions of such functions: one when data is represented by images separately and one when images are grouped in patient cases. The way we predict a patient case is by taking the average of the scores of the images that belong to that particular patient. If this mean is over or equal to 0.5, then the model classifies it as positive, otherwise, as negative.

We also required some tools to quickly load trained or partially trained models and save them, so we made use of the corresponding functions from the Keras library. Our models are saved in `.h5` format.

5 Model design and hyperparameter tuning

5.1 Simple dense architectures

The begin, we tried simple dense neural networks. All of them featured a flatten layer, followed by blocks of layers containing different number of neurons, with ReLU activations and dropout layers. The last block always consisting of a Dense layer with 1 neuron and ‘sigmoid’ activation. We use a cross-entropy loss function and the Adam optimizer [7]. Initially we reshaped our images to 512×512. The last architectures we tried for these kind of models consisted of a flatten layer, followed by a fense layer with 512 neurons, one with 64 neurons and one with 32 neurons, before the final block.

	T	F
P	73	41
N	107	67

	T	F
P	26	12
N	54	40

Figure 1: Confusion matrices for Conv3 on humerus samples. On individual images (left) and grouped by patient (right).

The confusion matrices revealed that these kind of models were simply predicting all images to be negative, even when imposing class weights. From this we concluded that simple dense networks were simply not complex enough for this task.

5.2 Convolutional neural networks

The next step was to use convolutional neural networks (CNNs). For our first two models, we chose keep reshaping images to 512×512. In general, all trained on batches of size 8 and for about 20 to 30 epochs. One exception is our last model, Conv4, which ran for only 10 epochs to prevent overfitting. Our first model was inspired by the VGG16 model [8]. Confusion matrices revealed that the model was not just predicting one class to each image. However, the accuracy of the model remained low even after tuning. For example, the accuracy on the humerus test images was just 0.57.

5.2.1 Conv2

Our second model (and first CNN), Conv2, consisted of three blocks before the Flatten layer. Each block consisted of a convolution layer, a pooling layer and finally a dropout. While tuning this model we found out that it is beneficial to start with fewer filters and a small kernel size in the first layers, and then gradually increase these features. After the flatten layer, we added a dense layer with 32 neurons, followed by a dropout and the standard dense layer with just one neuron and 'sigmoid' activation. The idea of this last block was taken from first experiments with simple dense neural networks.

The overall accuracy of Conv2 did not represent a drastic increase over our first model, but it did have an accuracy of 0.63 on forearms.

5.2.2 Conv3

We then moved to build a new CNN, Conv3. For this, we decided to reshape our images to 224×224. This allowed us to use more complex models on the hardware we had, without running into performance or memory issues. Conv3 is broadly similar in design to Conv2. We used five blocks before the flatten layer. After flattening, first dense layer featured 64 neurons, instead of 32. Using 5-fold cross validation, we tuned and improved Conv3. We concluded with the following architecture: the first 2 blocks were identical: convolution layers used two filters and kernel sizes of 3; pool sizes in pooling layers are 2. The next two blocks increased the number of filters to 4 and the kernel sizes to 5. Pooling layers remained the same. The last block featured 8 filters, kernels of size 7 and pool sizes of 4. Conv3 provided satisfactory results. When tested on individual humerus images (Figure 1), it produced an accuracy of 0.625. When grouping images by patient, the accuracy dropped slightly to 0.606.

5.2.3 Conv4

We kept experimenting with convolutional neural networks and created a new one, Conv4. We decided to keep our images' sizes as 224×224. Conv4 is broadly similar in design to Conv3. We again used 5 blocks, a flatten layer, a hidden dense layer and the 1-neuron output layer with sigmoid activation. The major difference is inside the blocks. We used batch normalization layers instead of blockout layers. The aim of this is to stabilize the distributions of hidden layer inputs, so that the training is more efficient [9]. There are longer explanations on why batch normalization is effective, but suffice to say there's plenty of empirical evidence that it improves the training performance [9]. This time, we kept the kernel size the same for all of the blocks and increased the number of filters from one block to another with a steeper curve. Batch

Category	Accuracy (%)	True positives	False positives	True negatives	False negatives
Elbow	72.0	157	57	178	73
Finger	66.6	150	57	157	97
Forearm	63.8	71	29	121	80
Wrist	73.0	169	52	312	126
Shoulder	69.8	179	71	214	99
Humerus	69.1	97	46	102	43

Figure 2: The results of the Conv4 network on each type of image.

Category	Accuracy (%)	True positives	False positives	True negatives	False negatives
Elbow	77.0	42	12	75	23
Finger	71.1	55	20	63	28
Wrist	73.4	49	10	103	45
Shoulder	68.8	54	19	65	35
Humerus	75.8	47	13	53	19

Figure 3: The results of the Conv4 network on each type of image, with patients grouped together (forearm is eliminated due to the small amount of data).

normalization combined with these changes performed somewhat better when trained on the whole data set with 0.61 accuracy, but performed significantly better in some categories such as wrist (Figure 2).

5.2.4 Residual neural networks

Our final Conv4 model had relatively few layers. For a task as complex as medical diagnosis, it would be useful to have more layers to allow for more subtle feature extraction. Until recently, attempting to create networks with a very high number of layers would lead to the *vanishing gradient problem* [10]. In short, the gradients used in gradient descent end up being so small as to elicit essentially no change in the parameters of the net.

Much progress has been made in recent years to tackle the vanishing gradient problem. *Residual neural networks* implement *skip connections* which connect non-adjacent layers. Particularly successful implementations include ResNet [1] and DenseNet [2]. In particular, the baseline model described in [5] makes use of a 169-layer DenseNet architecture.

We spent some time attempting to implement versions of ResNet and DenseNet. We ran into some bugs with the Keras implementation of batch normalization, which delayed our progress somewhat. We eventually found, once our implementations were ready, that they were simply too large to run reasonably on the GPUs we had available. In particular, the GTX 1070 simply does not have enough memory to hold such a network when dealing with so many large images. We did attempt to create smaller versions, but found them to be less accurate than our own Conv4 network. Had we had more time, we would have experimented further with residual networks, but we decided after some time that the amount of effort required to get them running wasn't worth the expected performance benefit.

6 Interpretation

In order to make sure our model is truly learning, we made use of Class Activation Mapping [3]. We created a heatmap function that generates the relevant heatmap for an image and a given model, saving it as a PNG. We also provided the flexibility of reshaping the image and standardizing it, if the mean and standard deviation images are given. We implemented the Grad-CAM algorithm [11, 12]. The algorithm consists of taking the model's last convolutional layer and the output layer and finding the areas from the input image with higher activation value. In this way we generate a heatmap which we then apply, by pixel-wise addition to the original X-ray, converted to RGB. Thus we obtain the applied heatmap of the image. The red areas represent the areas with higher activation and the blue areas lower activation.

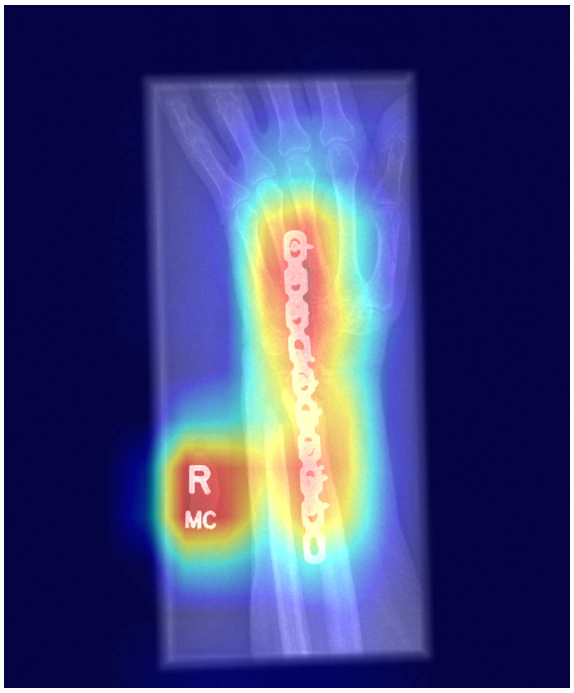

Figure 4: A heatmap for a positive wrist sample. The network has correctly identified the abnormality, but was significantly distracted by the text label to the left.

We see that the images classified as negative tend to have blank heatmaps. More interestingly, we see that the model identifies lettering present on images as abnormalities (Figure 4). The letters have the potential to confuse our model and classify a healthy case as positive because of their presence. Significant improvements could be made by cleaning the data set, by removing the letters from X-rays. This could be made by possibly building another program that would be able to identify them, or by a very tedious, manual image processing.

7 Ethical considerations

Broadly speaking, we did not see any ethical issues with the goals of the project; classifying X-ray images for the purpose of diagnosis of musculoskeletal conditions can (practically) only be used for medical applications. These medical applications would generally increase public utility, assuming our project would be used only to assist diagnosis, as increasing the speed and accuracy of patient diagnosis would reduce the workload of healthcare professionals and contribute to better healthcare outcomes. Given the increasingly tight resource and manpower constraints the NHS is forced to operate under, having a way to make radiology work more efficient is highly desirable. There is a small possibility that, even if our project is only used for medical purposes, it could be used to decrease utility: say, if diagnoses are used to deny coverage to patients with conditions. However, these kinds of problems would reflect deep systemic issues within a healthcare system, which are beyond our power to address.

The most obvious area where ethical concerns could arise is in the data itself. There must be patient consent for the collection and distribution of patient X-rays, for fairly obvious reasons. Further, maintaining patient anonymity is important - our model does not require any identifying patient information, so there should be none. Having any such information in testing (or training) data would violate basic expectations of patient privacy, and could possibly represent security breaches in whichever health system the data is sourced from. It almost goes without saying that this kind of personal data could be used for a variety of non-medical or even malicious purposes, so ensuring that such data is not exposed is paramount. With the MURA dataset we checked that there are no such issues, so we are sure that the building and training of our model was not affected by these ethical concerns. With future users, however, we will have to ensure that their test data is carefully examined and do not have these problems.

On the implementation side, there is very little to be concerned about with the networks themselves, but it is important to be careful with representing results. Our solution is not quite as accurate at diagnosis as expert radiologists are, and since it lacks knowledge about radiology, even if our model can make an accurate diagnosis, it (thus far) cannot specify which condition is present, or provide any other information beyond a very simple 'healthy or not' diagnosis. The fairly high potential for misdiagnosis means that a second opinion, preferably and expert one, should always be sought before making official diagnoses. In no circumstances, then, should our model be used to replace actual radiologists - this would both take away jobs and worsen any medical outcomes that might benefit from more contextual information. We must communicate to any potential users that it must be used in conjunction with, rather than in place of, actual radiology professionals.

Another potential area of concern is that our model has varying accuracy with different body parts, so the overall accuracy rate is misleading, and if it is the only such figure emphasized, users may have incorrect notions of how accurate our model is when applied to a specific body part. Thus, the accuracy for each body part should be presented alongside any overall results, as these more specific rates would help users determine how to use our model. We believe this concern, as well as most of the others, can be addressed by clearly communicating the limitations of our model with users and using discretion with distribution of our work.

8 Summary

We trained a convolutional network to identify musculoskeletal problems from X-ray samples with accuracy from 69 to 77%, depending on the type of X-ray. We used Class Activation Mapping to identify the areas the model found relevant in its diagnosis.

Future work could focus on working with more complex architectures, which would require more powerful hardware. Work could also be done to clean up features such as text labels from images, which confuse the model and lead to false positives, as our heatmaps show.

References

[1] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016, pp. 770–778.

[2] Gao Huang et al. "Densely connected convolutional networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2017, pp. 4700–4708.

[3] Bolei Zhou et al. "Learning deep features for discriminative localization". In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016, pp. 2921–2929.

[4] Martín Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems". Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[5] Pranav Rajpurkar et al. "MURA: Large dataset for abnormality detection in musculoskeletal radiographs". 2017. arXiv: `1712.06957`.

[6] William Kahan. "Pracniques: further remarks on reducing truncation errors". In: *Communications of the ACM* 8.1 (1965), p. 40.

[7] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". 2014. arXiv: `1412.6980`.

[8] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". 2014. arXiv: `1409.1556`.

[9] Shibani Santurkar et al. "How Does Batch Normalization Help Optimization?" In: *arXiv e-prints*, arXiv:1805.11604 (May 2018), arXiv:1805.11604. arXiv: `1805.11604 [stat.ML]`.

[10] Sepp Hochreiter et al. “Gradient flow in recurrent nets: the difficulty of learning long-term dependencies”. 2001.

[11] Ramprasaath R Selvaraju et al. “Grad-cam: Visual explanations from deep networks via gradient-based localization”. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 618–626.

[12] Stepan Ulyanin. “Implementing Grad-CAM in PyTorch”. 2019. URL: `https://medium.com/@stepanulyanin/implementing-grad-cam-in-pytorch-ea0937c31e82`.