Deep Learning Project 2 - Abnormality Detection in bone X-Ray

Mavroforos Nikolaos - p3352014
Ntenezos Panagiotis - p3352025

April 3, 2022

The code can be found here: https://github.com/d-tchmnt/Deep-Learning

1 Introduction

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. More information about the task and the dataset can be found at https://stanfordmlgroup.github.io/competitions/mura/.

2 Dataset

The dataset contains predefined train and validation directories, where type of data comes into separate sub-directory. Training set contains 36812 train images from the following body parts:

WRIST
HAND
FINGER
FOREARM
SHOULDER
HUMERUS
ELBOW

Since our data are actual coloured images with size 256x256, it would require a lot of memory to load everything into the RAM and train our models. This is why we used **ImageDataGenerator**, which handles the batch-loading and augmentation of images into our NN.

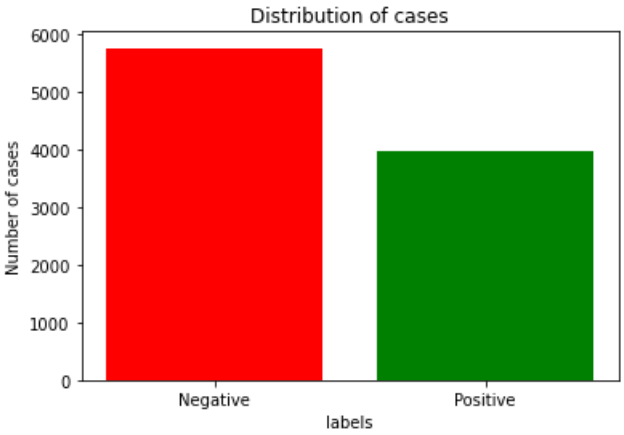
2.1 Data split

In order to create valid sub-splits of our data, we will split the training data containing of 36808 rows, and split it 80%-20% to create our validation data. The images that exist in the directory 'valid' will be used as test data.
Using the image_generator library and method 'flow_from_dataframe' we will create the appropriate generators for train, valid and test sets.
We chose to keep the original RGB colour, using batch size 64 and shuffling the images.

2.2 Data Exploratory Analysis

Firstly, we will present the cumulative class distribution for each label.

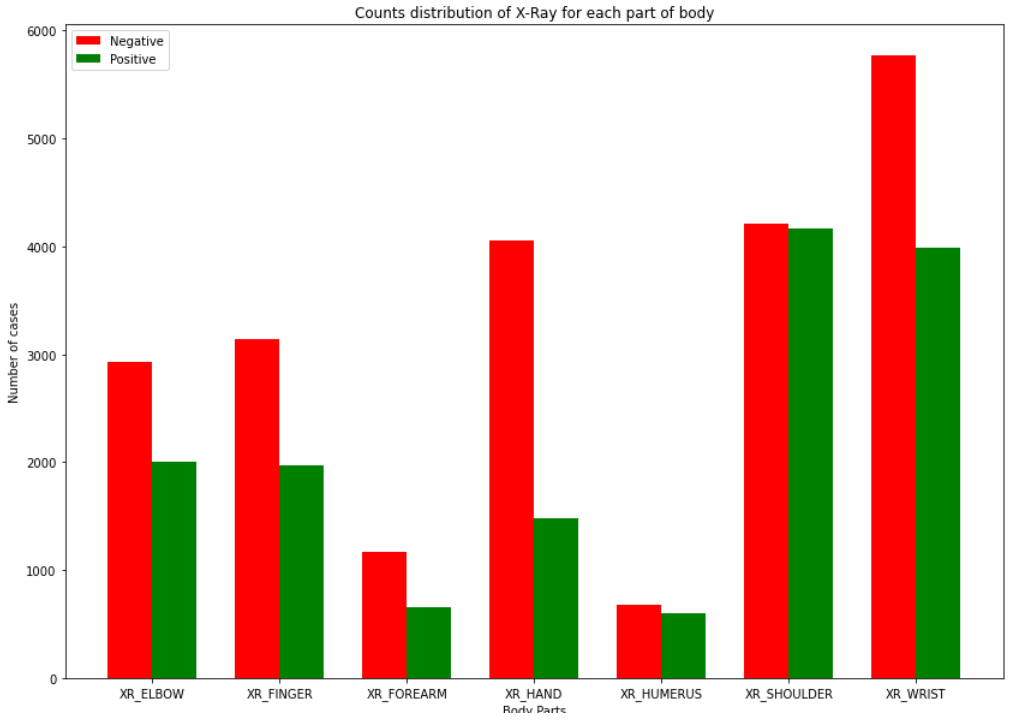
Cumulative Class Distribution for Training Data



The dataset of training data is unbalance with more negative images. So, we may need to create augment data for the positive category to be balance, if the models are bias. Due to computational limits we did not apply this step, but it is highly recommended.

We can, also, have a more detailed analysis of the class distribution for all the body parts.

Class Distribution for all body parts

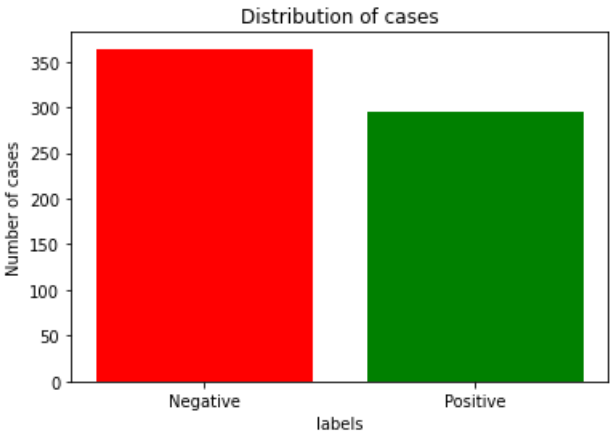


We observe that only the XR-Shoulder has balance data and XR-Wrist has the most of data for negative X-Ray.
Moreover, we understand that we do not have same number of data of each category, so we may be difficult to train the model in some categories like XR-Humerus. Thus, it is vital we use the feature category in our model.

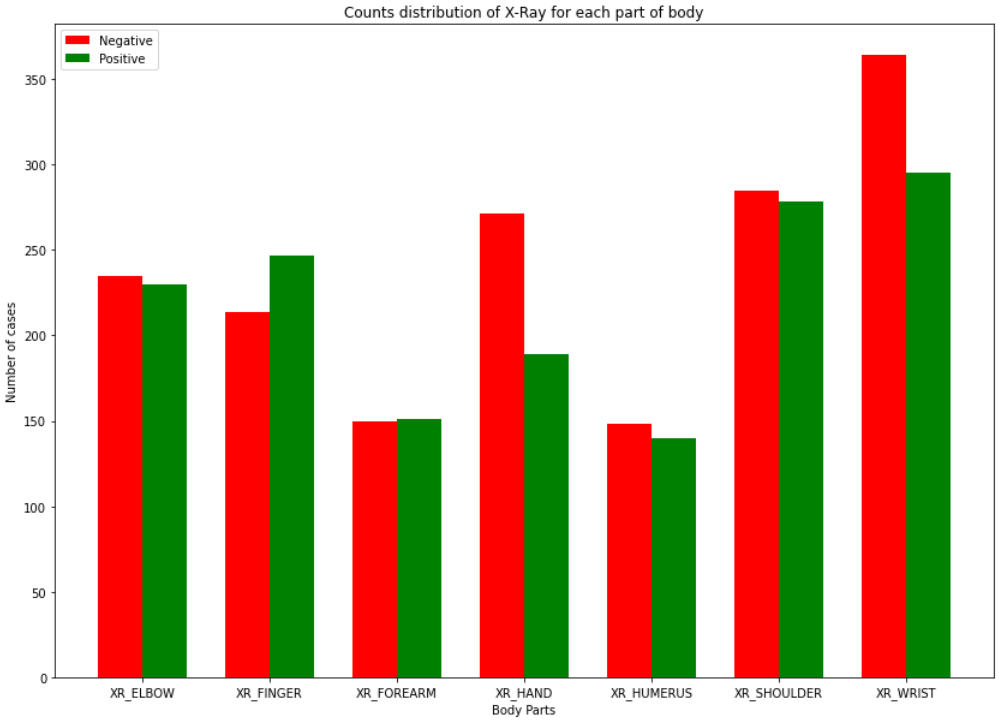
Furthermore, if we split train set to validation and develop, it is better we take a part of each part of body separately (or suffle the data), to maintain the distribution of each category.

Below we show the class distribution for the test data and the class distribution.

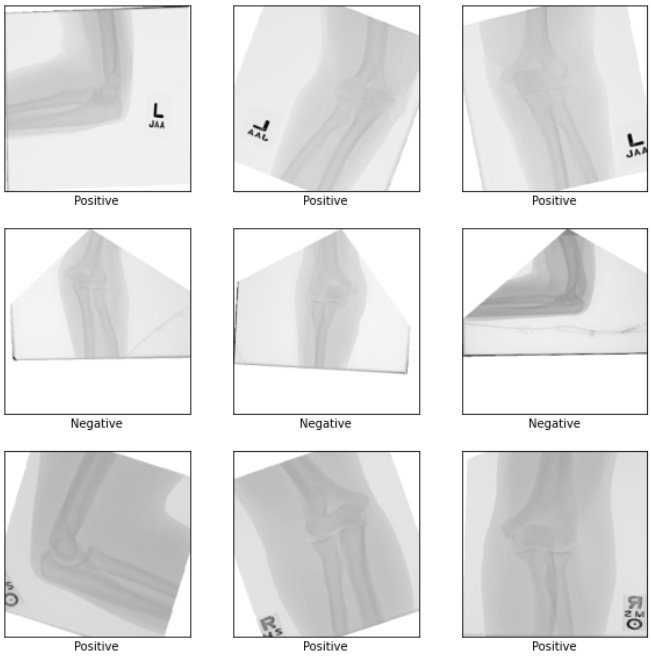
Class Distribution For Test Data



Class Distribution For Test Data

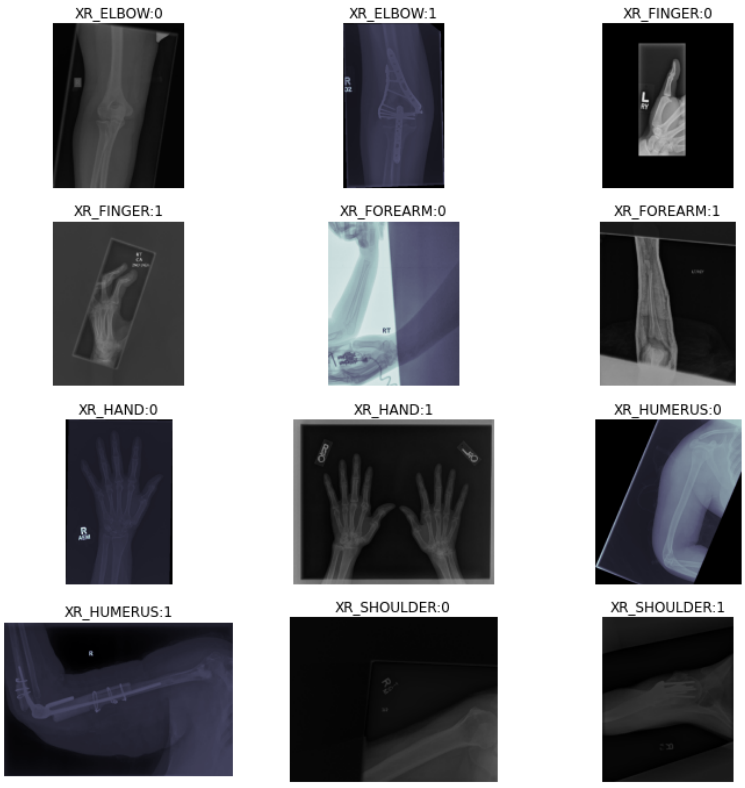


We will proceed by presenting some of the actual images of our dataset.



We observe that the XR, has label R(right) or L(left) and some metadata(Q, MS) on top of the image. We can see that most of the images are randomly rotated, randomly resized of images and randomly resized of labels. Moreover, some labels can not see easily or do not exist.

(We may need to transform the image as they need to extract labels)



We can, also, see that the images do not have the same colour or the same clearness.
(We may need to transform to gray-scale all images)

3 Metrics

Considering the task at hand and considering the MURA contest the appropriate measure to be used is **Cohen's Kappa** metric. 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 (recall) 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 **Binary 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.

Cohen's kappa

$$\kappa = \frac{2 \times (TP \times TN - FN \times FP)}{(TP + FP) \times (FP + TN) + (TP + FN) \times (FN + TN)}$$

4 CNN Model

Using the function '**cnn_builder**' we will create our CNN model. We want the option to have **many convolutions in each convolution layer**, **number of convolutional layers**, **batch normalization**, provide option for **strides**, the **default kernel matrix** (3x3), as well as **dropout in the final layer**.
Furtherly, pooling operations (either max / average pooling) were applied on top of each convolutional layer. Pooling provides a way to summarize feature maps. However, we found through experimentation that average pooling works best for the task. After each convolution we use the **relu** activation function to achieve non-linerarity in our predictions.
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 As the convolutional layers increase, the **numbers of the filters will also increase exponentially**. After the convolutional layer have completed their transformation, we include option for **last pooling** or **average pooling**, we will **flatten** the output, and have one final dense (output) layer with **sigmoid** activation function in order to make our prediction.
As a result, the defualt values, we have the following:

4.1 Initial Architecture

We will create a simple initial model for our comparison purposes. Our model will have 4 convolution layer, with 1 convolution per layer along with 1 max pooling, and it will run for 100 epochs, using Adam Optimizer, with early stopping with patience 10 epochs,trying to minimize the Binary Cross-Entropy.
First we can see our initial model's architecture

initial CNN Model Architecture

```
Model: "model_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input (InputLayer)           [(None, 256, 256, 3)]     0
_________________________________________________________________
conv_0_0 (Conv2D)            (None, 256, 256, 32)      896
_________________________________________________________________
conv_0_0_relu (Activation)   (None, 256, 256, 32)      0
_________________________________________________________________
mp_0 (MaxPooling2D)          (None, 128, 128, 32)      0
_________________________________________________________________
conv_1_0 (Conv2D)            (None, 128, 128, 64)      18496
_________________________________________________________________
conv_1_0_relu (Activation)   (None, 128, 128, 64)      0
_________________________________________________________________
mp_1 (MaxPooling2D)          (None, 64, 64, 64)        0
_________________________________________________________________
conv_2_0 (Conv2D)            (None, 64, 64, 128)       73856
_________________________________________________________________
conv_2_0_relu (Activation)   (None, 64, 64, 128)       0
_________________________________________________________________
mp_2 (MaxPooling2D)          (None, 32, 32, 128)       0
_________________________________________________________________
conv_3_0 (Conv2D)            (None, 32, 32, 256)       295168
_________________________________________________________________
conv_3_0_relu (Activation)   (None, 32, 32, 256)       0
_________________________________________________________________
mp_3 (MaxPooling2D)          (None, 16, 16, 256)       0
_________________________________________________________________
flatten (Flatten)            (None, 65536)             0
_________________________________________________________________
output (Dense)               (None, 1)                 65537
=================================================================
Total params: 453,953
Trainable params: 453,953
Non-trainable params: 0
_________________________________________________________________
```

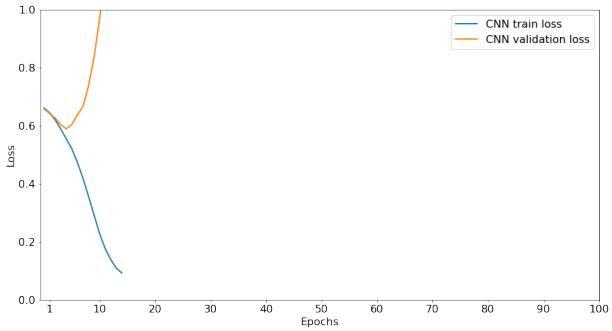
The results from the initial model are shown below

initial CNN Model Results

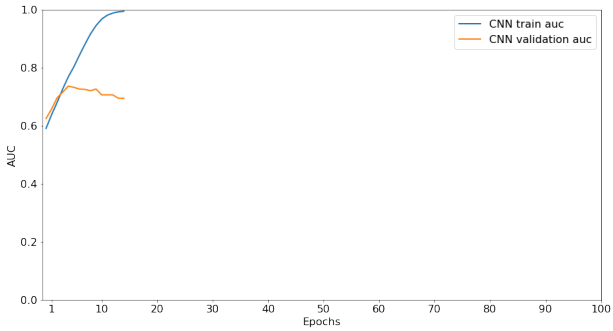
```
Train Loss     : 0.09284
Validation Loss: 1.75171
Test Loss      : 0.63218
---
Train AUC     : 0.99450
Validation AUC: 0.69432
Test AUC      : 0.70425
---
Train Accuracy     : 0.96465
Validation Accuracy: 0.65838
Test Accuracy      : 0.64967
---
Train Kappa     : 0.92640
Validation Kappa: 0.92640
Test Kappa      : 0.29082
```

Now let's see how well our model performed

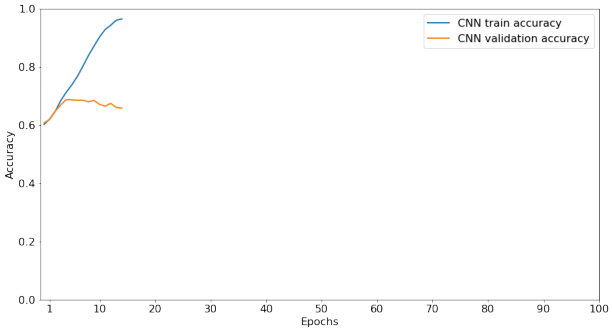
initial CNN Model Plot: Loss



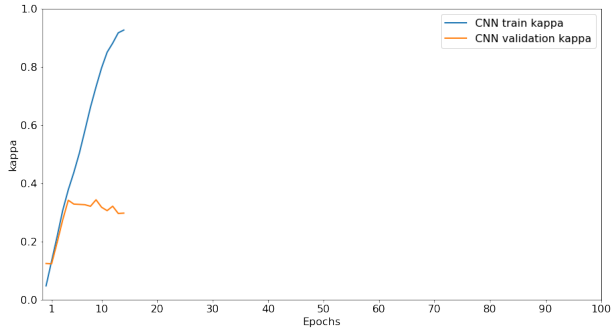
initial CNN Model Plot: AUC



initial CNN Model Plot: Accuracy



initial CNN Model Plot: kappa



We can see that our model overfits, while not achieving great results. We should also mention that the loss increases while accuracy remains the same. This could be explained by the cross-entropy which we are trying to minimize. When we use cross-entropy loss for classification false predictions are penalized much more strongly than correct predictions are rewarded. For a normal image, the loss is log(1prediction), so even if many normal images are correctly predicted (low loss), a single misclassified normal image will have a high loss, hence "blowing up" the mean loss.

4.2 Deep Model

We will carry on by creating a more complex CNN model, again with 4 convolutional layers and 1 convolution per layer, with average pooling, dropout of 0.2, batch normalization and pooling stride of (2,2).

First we can see our initial model's architecture

Deep CNN Model Architecture

Model: "model"

Layer (type)	Output Shape	Param #
input (InputLayer)	[(None, 256, 256, 3)]	0
conv_0_0 (Conv2D)	(None, 256, 256, 32)	896
bn_0_0 (BatchNormalization)	(None, 256, 256, 32)	128
conv_0_0_relu (Activation)	(None, 256, 256, 32)	0
mp_0 (AveragePooling2D)	(None, 128, 128, 32)	0
dropout_0 (Dropout)	(None, 128, 128, 32)	0
conv_1_0 (Conv2D)	(None, 128, 128, 64)	18496
bn_1_0 (BatchNormalization)	(None, 128, 128, 64)	256
conv_1_0_relu (Activation)	(None, 128, 128, 64)	0
mp_1 (AveragePooling2D)	(None, 64, 64, 64)	0
dropout_1 (Dropout)	(None, 64, 64, 64)	0
conv_2_0 (Conv2D)	(None, 64, 64, 128)	73856
bn_2_0 (BatchNormalization)	(None, 64, 64, 128)	512
conv_2_0_relu (Activation)	(None, 64, 64, 128)	0
mp_2 (AveragePooling2D)	(None, 32, 32, 128)	0
dropout_2 (Dropout)	(None, 32, 32, 128)	0
conv_3_0 (Conv2D)	(None, 32, 32, 256)	295168
bn_3_0 (BatchNormalization)	(None, 32, 32, 256)	1024
conv_3_0_relu (Activation)	(None, 32, 32, 256)	0
mp_3 (AveragePooling2D)	(None, 16, 16, 256)	0
dropout_3 (Dropout)	(None, 16, 16, 256)	0
flatten (Flatten)	(None, 65536)	0
output (Dense)	(None, 1)	65537

Total params: 455,873
Trainable params: 454,913
Non-trainable params: 960

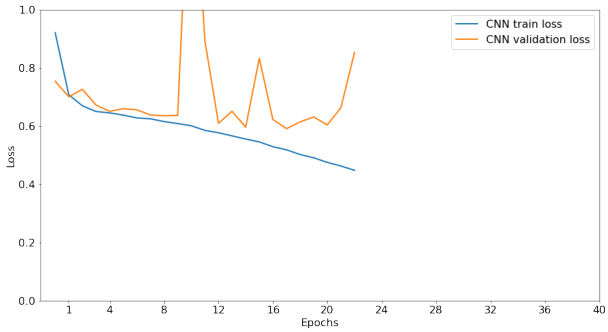
The results from the initial model are shown below

Deep CNN Model Results

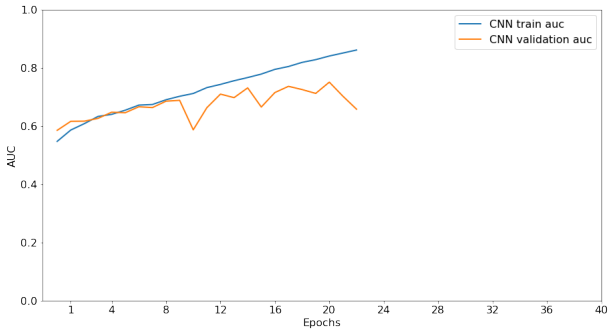
```
Train Loss     : 0.44815
Validation Loss: 0.85330
Test Loss      : 0.63419
---
Train AUC      : 0.86130
Validation AUC: 0.65808
Test AUC       : 0.71669
---
Train Accuracy     : 0.78839
Validation Accuracy: 0.63923
Test Accuracy      : 0.65812
---
Train Kappa     : 0.55070
Validation Kappa: 0.55070
Test Kappa      : 0.30771
```

Now let's see how well our model performed

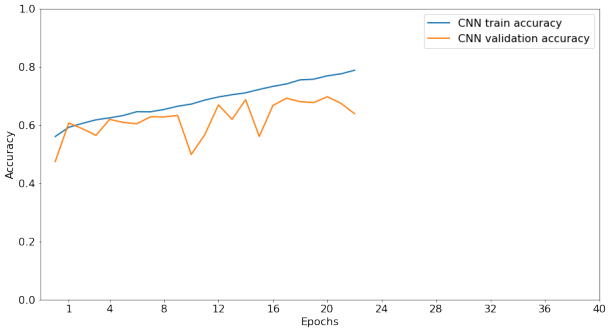
Deep CNN Model Plot: Loss



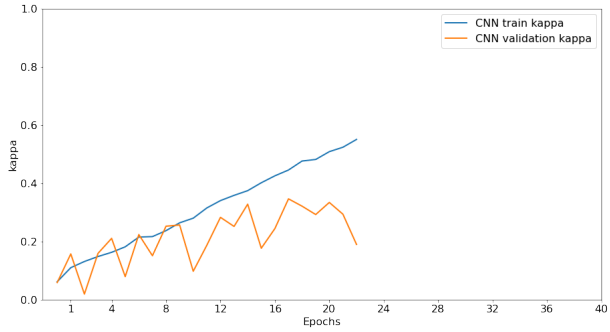
Deep CNN Model Plot: AUC



Deep CNN Model Plot: Accuracy



Deep CNN Model Plot: kappa



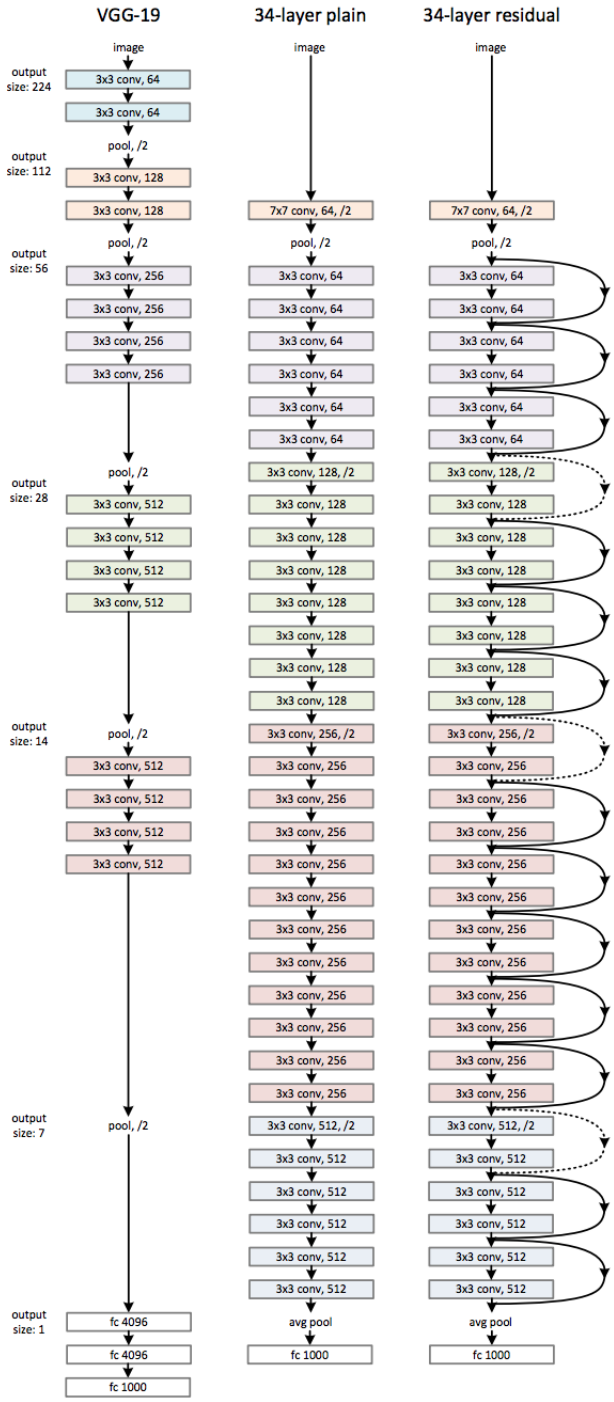
We can see that our loss does not go up, however it is fluctuating. Intuitively that some portion of examples is classified randomly, which produces fluctuations, as the number of correct random guesses always fluctuate.

5 Resnet Model

ResNet network uses a 34-layer plain network architecture inspired by VGG-19 in which then the shortcut connection is added. It solves the problem of vanishing gradiend and training degredation. Using the function '**resnet_builder**' we create our resnet(**ResNet152V2**) based classifier, using the weights from imagenet. We choose to remove the top layer, in order to add a dense layer with a sigmoid activation function to create our classifier. Below we present the architecture of our resnet:

First we can see our initial model's architecture

ResNet Model Architecture



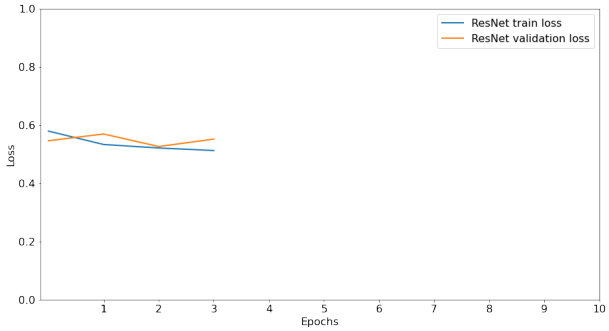
The results from the initial model are shown below

Resnet Model Results

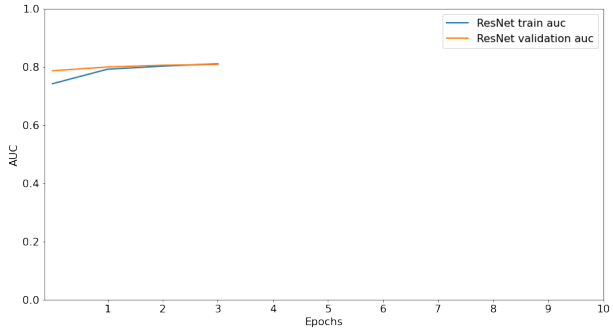
```
Train Loss      : 0.51262
Validation Loss: 0.55185
Test Loss       : 0.55354
---
Train AUC      : 0.81060
Validation AUC: 0.80821
Test AUC       : 0.78835
---
Train Accuracy      : 0.75511
Validation Accuracy: 0.73255
Test Accuracy       : 0.72099
---
Train Kappa      : 0.47469
Validation Kappa: 0.47469
Test Kappa       : 0.43671
```

Now let's see how well our model performed

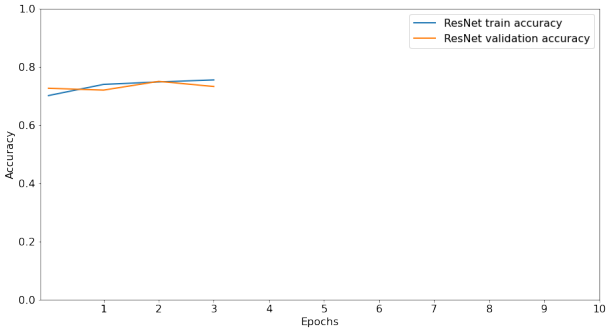
Resnet Model Plot: Loss



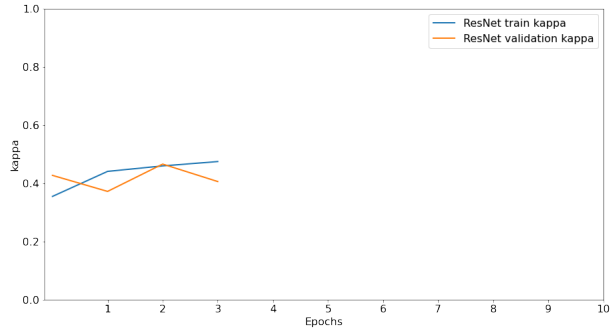
Resnet Model Plot: AUC



Resnet Model Plot: Accuracy



Resnet Model Plot: kappa



We can see that our model maintains a high enough AUC and achieves the best kappa from our previously created models. This makes sense, since it was pre-trained using images specifically

6 References

1. https://matplotlib.org/3.1.1/gallery/lines_bars_and_markers/barchart.html#sphx-glr-gallery-lines-b

2. https://www.kaggle.com/hasnaatawfik/mura-classification?fbclid=IwAR0uW8oNPMdcBMUe6keuRUmCNPp57rOCbI

3.https://github.com/vraul92/Humerus-Bone-Fracture-Detection/blob/master/train__humerus_fracture_detection_keras_model.ipynb

4. https://github.com/ag-piyush/Bone-Fracture-Detection---MURA/blob/cfd68b53dd39bd81aa893b4b7cc38c296data_loader.py#L52

5. https://www.kaggle.com/kmader/mura-data-overview