ΟΙΚΟΝΟΜΙΚΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΑΘΗΝΩΝ

ATHENS UNIVERSITY
OF ECONOMICS
AND BUSINESS

ASSIGNMENT 2:

CNN ARCHITECTURES AS SOLUTION FOR THE ABNORMALITY DETECTION IN BONE X-RAYS

INTZOGLOU EVANGELOS – MASTRODIMITRIS GOUNAROPOULOS SPYRIDON
2023

Folder Link: https://drive.google.com/drive/folders/1h01FzXeGb59ciE5XKw-ZQfk4Sj-qL8C1?usp=sharing

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/.

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

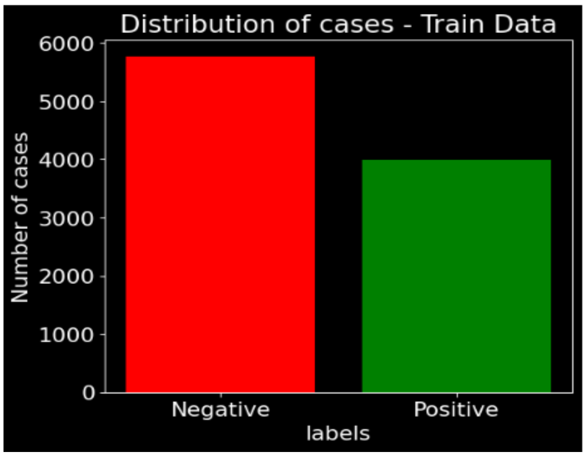
Train-Test-Validation split:

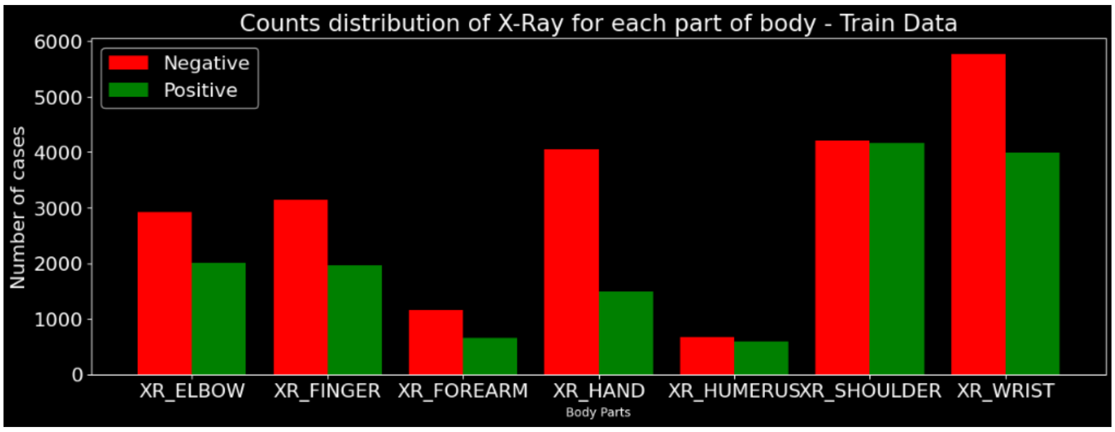
In order to create valid sub-splits of our data, we will split the training data containing of 36808 rows, and split it 80%-20% to create our validation data. The images that exist in the directory 'valid' will be used as test data.

We will generate the proper generators for train, valid, and test sets using the image generator package and method "flow from dataframe."

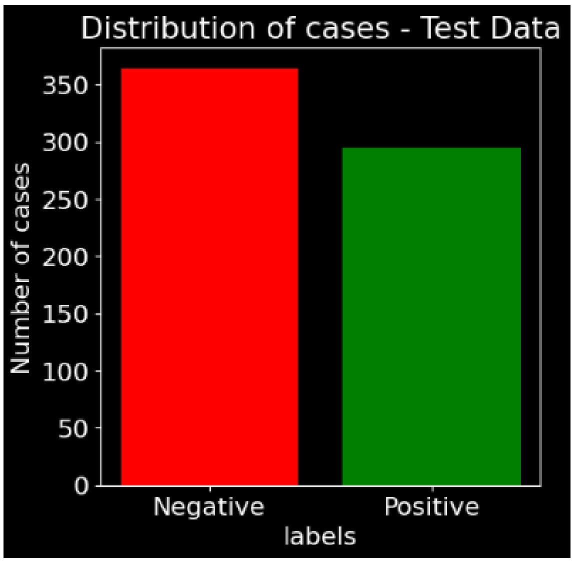
Exploratory Analysis:

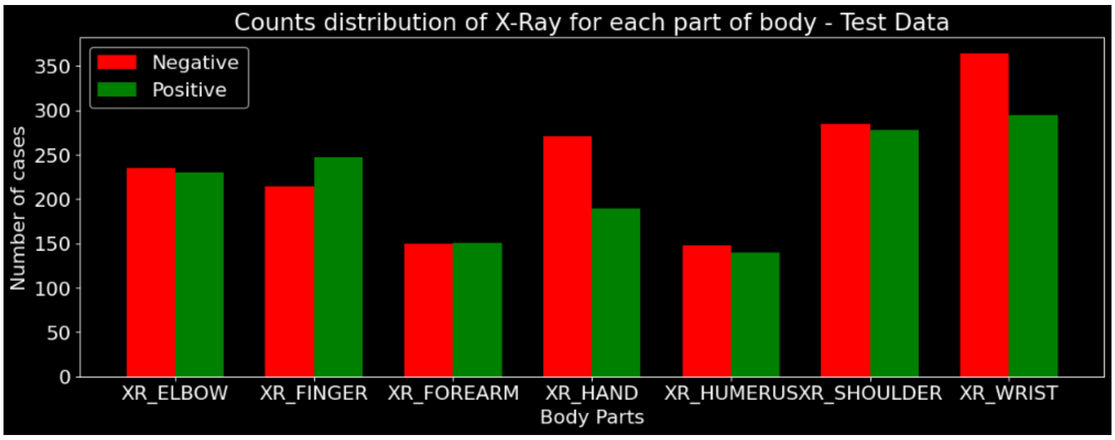
Below, we will present the plots with frequency distributions for the data as a whole and for each subset of radiographs. First for the Train Data,





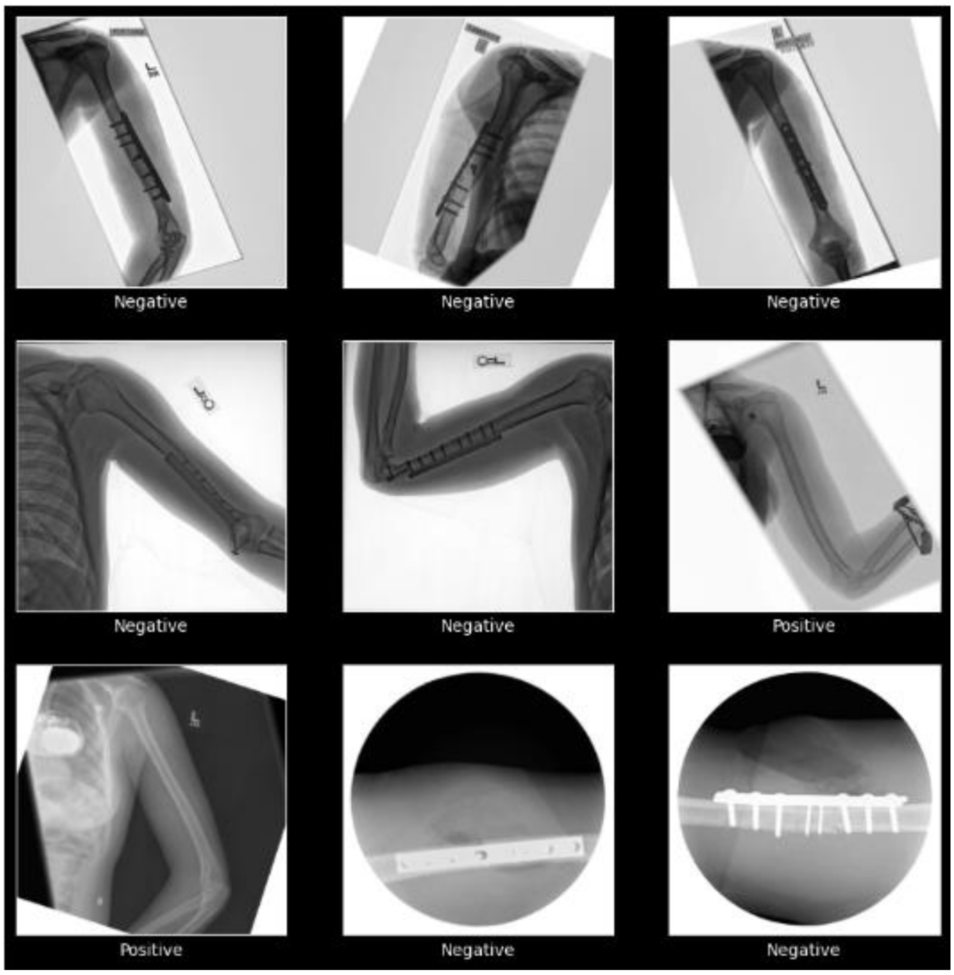
and then for Test Data,





The dataset for Train data is imbalanced with more negative images. Also, we observe that only the XR_HOULDER and XR_HUMERUS have balanced data and XR-Wrist has the most of data for negative X-Ray.

Continuing, we will present below some indicative X-ray images from the dataset,



Metrics Used:

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. 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.

NOTE:

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. In addition, due to lack of computing power to create the models, XR subclasses or a combination of them were taken.

CNN Model (Own build):

Our CNN model was built using the "cnn builder" function. We provide options for the number of convolutional layers, batch normalization, number of convolutional layers, strides, the default kernel matrix (3x3), and dropout in the final layer.

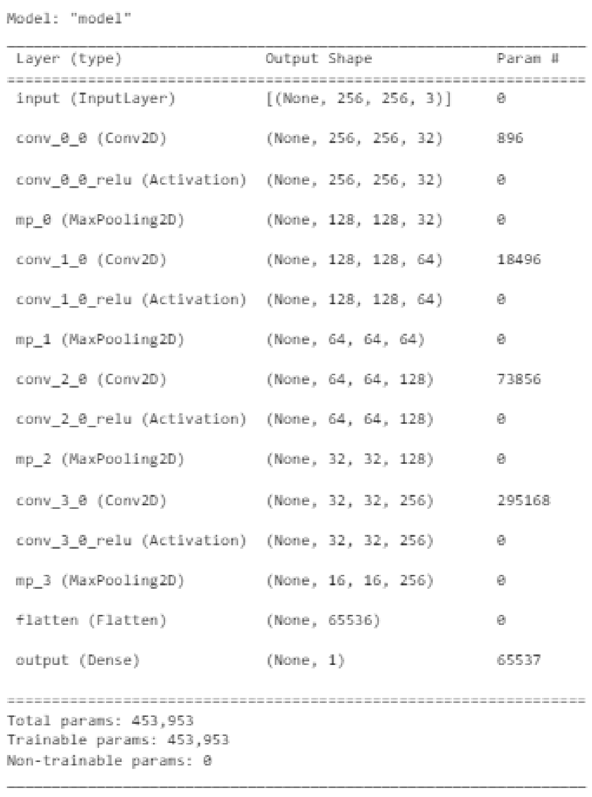
On top of each convolutional layer, pooling procedures (either maximum or average pooling) were added. Using pooling, feature maps can be condensed. Nevertheless, we discovered through testing that avg pooling performs better for our task. To achieve non-linearity in our predictions, we apply the relu activation function after each convolution.

To prevent overfitting to the training data, dropout was used. Thus, we used two different types of dropout, both with a rate of 0.2: a spatial dropout after each convolutional layer and a dropout layer on the parameters of the fully connected layer. The same reason is why we the number of filters will exponentially increase as the number of convolutional layers increases. 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.

The results our models are presented below:

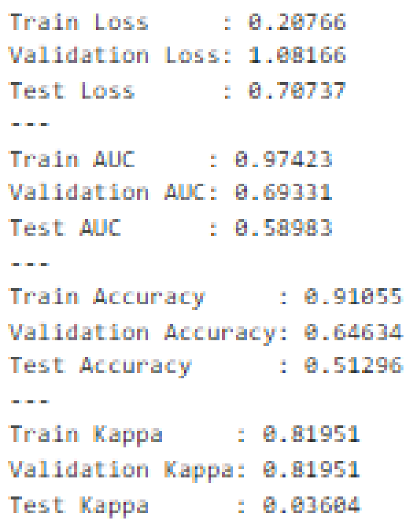
CNN – Baseline:

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. Below, we can see our baseline model architecture:



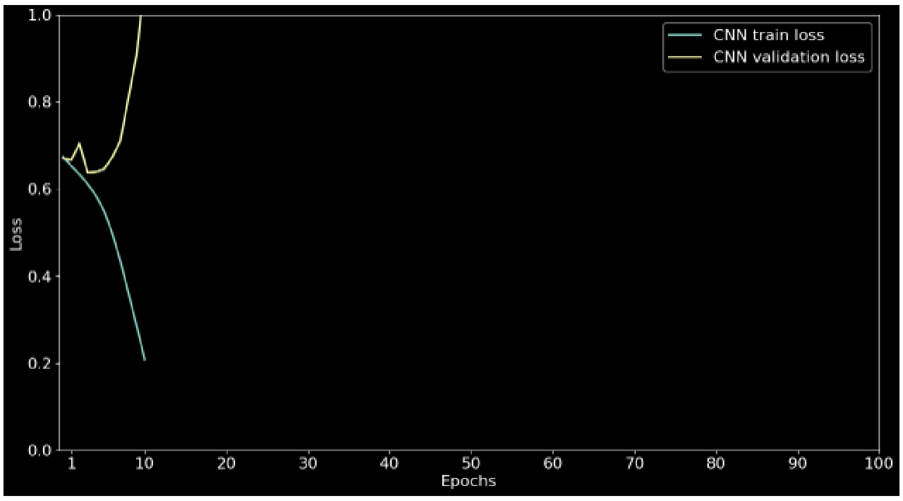
```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input (InputLayer)          [(None, 256, 256, 3)]     0

 conv_0_0 (Conv2D)           (None, 256, 256, 32)      896

 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
_________________________________________________________________
```

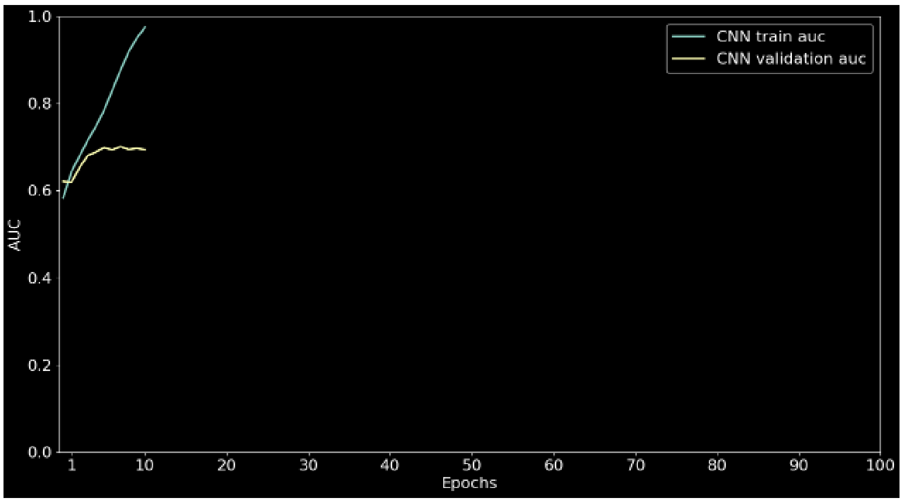
The results from our CNN – Baseline model:

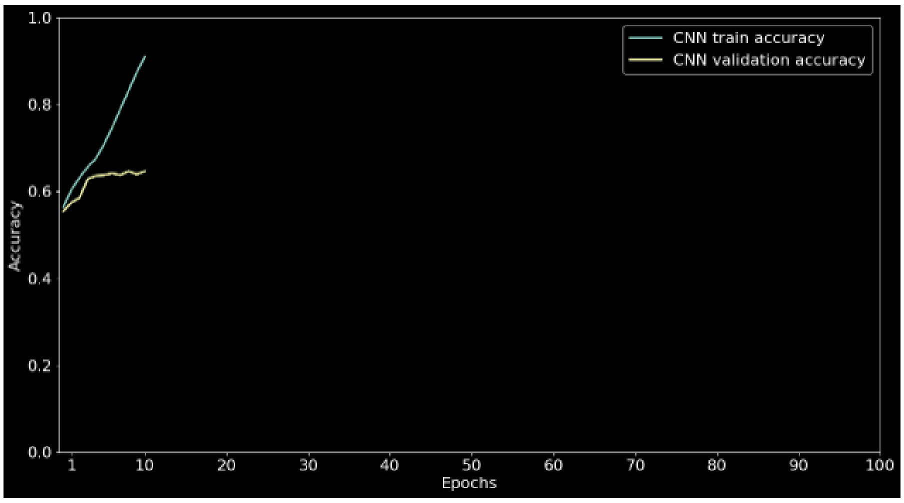


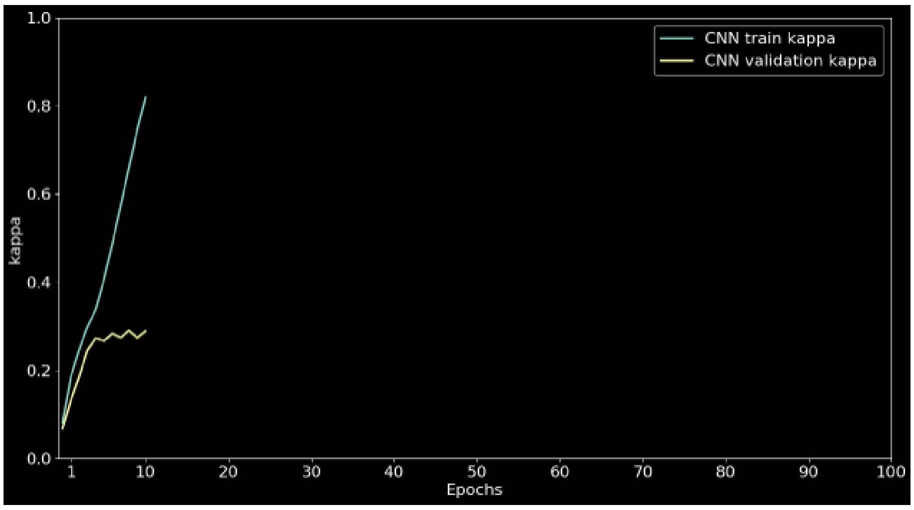
```
Train Loss      : 0.20766
Validation Loss: 1.08166
Test Loss       : 0.70737
---
Train AUC      : 0.97423
Validation AUC: 0.69331
Test AUC       : 0.58983
---
Train Accuracy      : 0.91055
Validation Accuracy: 0.64634
Test Accuracy       : 0.51296
---
Train Kappa      : 0.81951
Validation Kappa: 0.81951
Test Kappa       : 0.03604
```

The Learning Curves of the CNN – Baseline model:





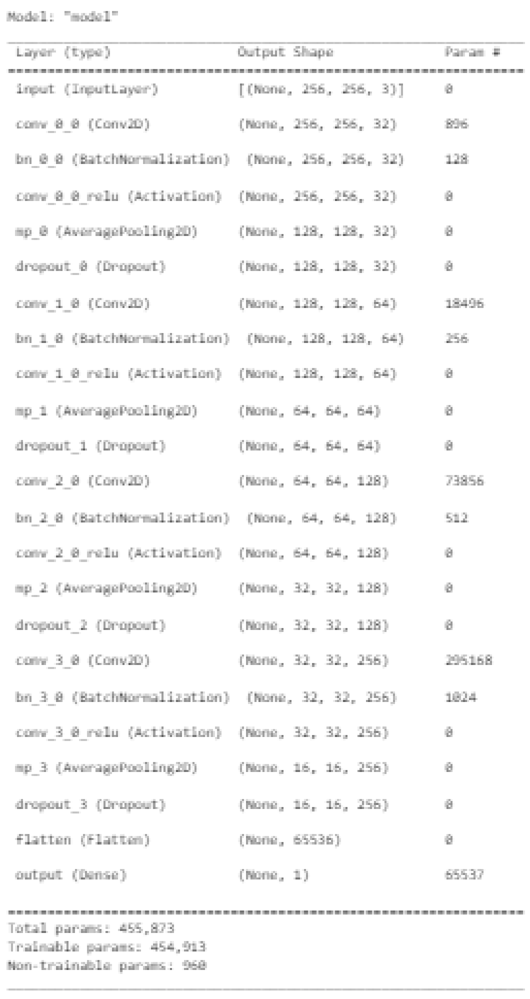




The CNN - Baseline model we trained for image classification achieved promising results on the training set, with low training loss and high AUC, accuracy, and kappa scores. However, the model's performance on the validation and test sets was not as strong, indicating potential overfitting. The validation and test losses were higher, suggesting the model struggled to generalize to unseen data. The AUC, accuracy, and kappa scores were also lower on the validation and test sets, indicating decreased performance compared to the training set.

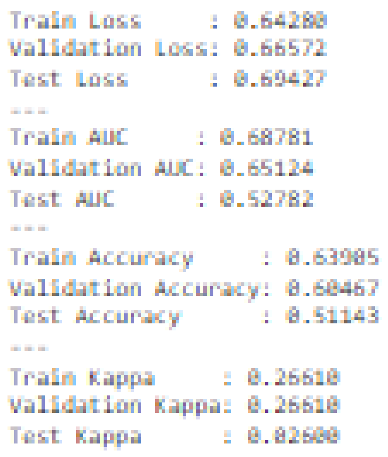
CNN – Complex:

The second CNN model had a more complex architecture compared to the previous model. It consisted of four convolutional layers with one convolutional operation per layer. The pooling strategy used was average pooling, and batch normalization was applied. A dropout rate of 0.2 was employed to reduce overfitting. The model was trained using the Adam optimizer and Binary Crossentropy loss and pooling stride of (2,2). Below, we can see our complex model architecture:



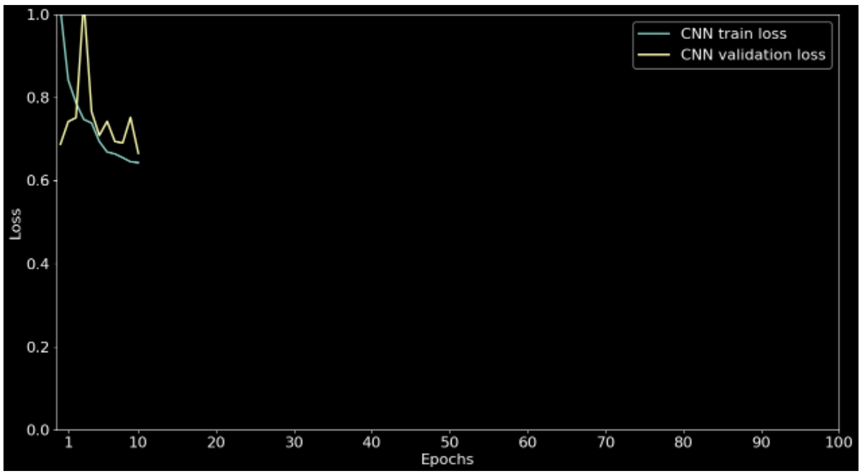
```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input (InputLayer)          [(None, 256, 256, 3)]     0

 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
_________________________________________________________________
```

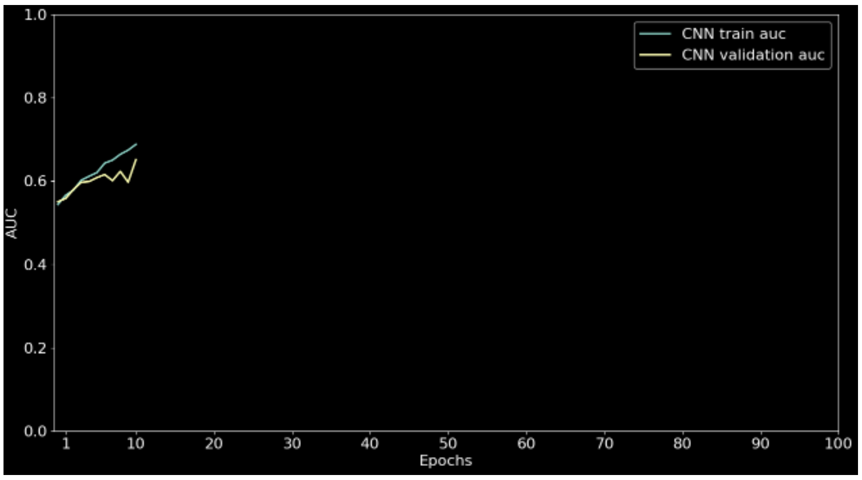
The results from our CNN – Complex model:

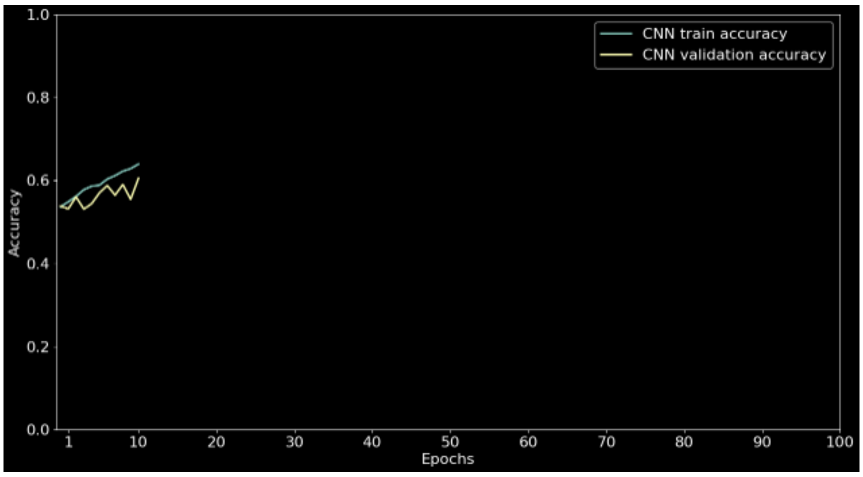


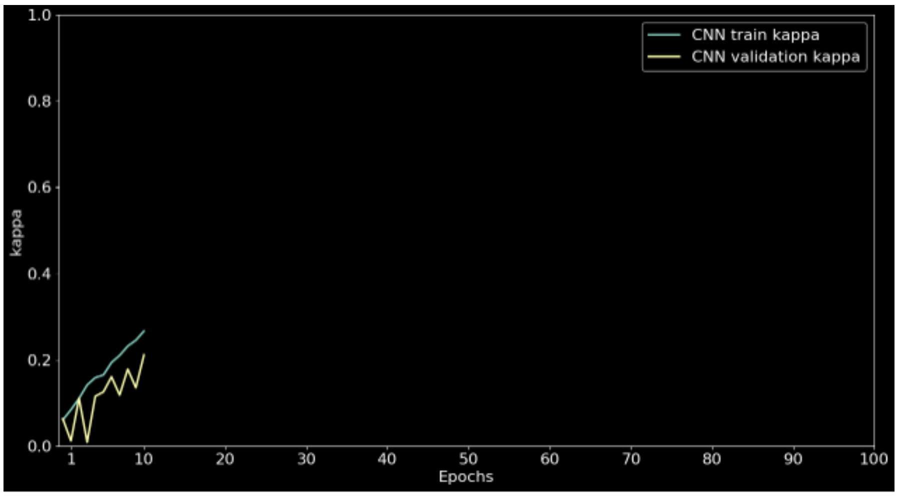
```
Train Loss     : 0.64280
Validation Loss: 0.66572
Test Loss      : 0.69427
---
Train AUC     : 0.68781
Validation AUC: 0.65124
Test AUC      : 0.52782
---
Train Accuracy     : 0.63985
Validation Accuracy: 0.60467
Test Accuracy      : 0.51143
---
Train Kappa     : 0.26610
Validation Kappa: 0.26610
Test Kappa      : 0.02600
```

The Learning Curves of the CNN – Complex model:


CNN train loss
CNN validation loss
Loss
Epochs


CNN train auc
CNN validation auc
AUC
Epochs


CNN train accuracy
CNN validation accuracy
Accuracy
Epochs



After evaluating the second CNN model on the test data, the following results were obtained:

- Train Loss: 0.64280
- Validation Loss: 0.66572
- Test Loss: 0.69427
- Train AUC: 0.68781
- Validation AUC: 0.65124
- Test AUC: 0.52782
- Train Accuracy: 0.63905
- Validation Accuracy: 0.60467
- Test Accuracy: 0.51143
- Train Kappa: 0.26610
- Validation Kappa: 0.26610
- Test Kappa: 0.02600

These results indicate that the model's performance on the test data is similar to its performance on the validation data. However, the model still struggles to achieve high accuracy and AUC scores, suggesting that it may not effectively generalize to unseen data.

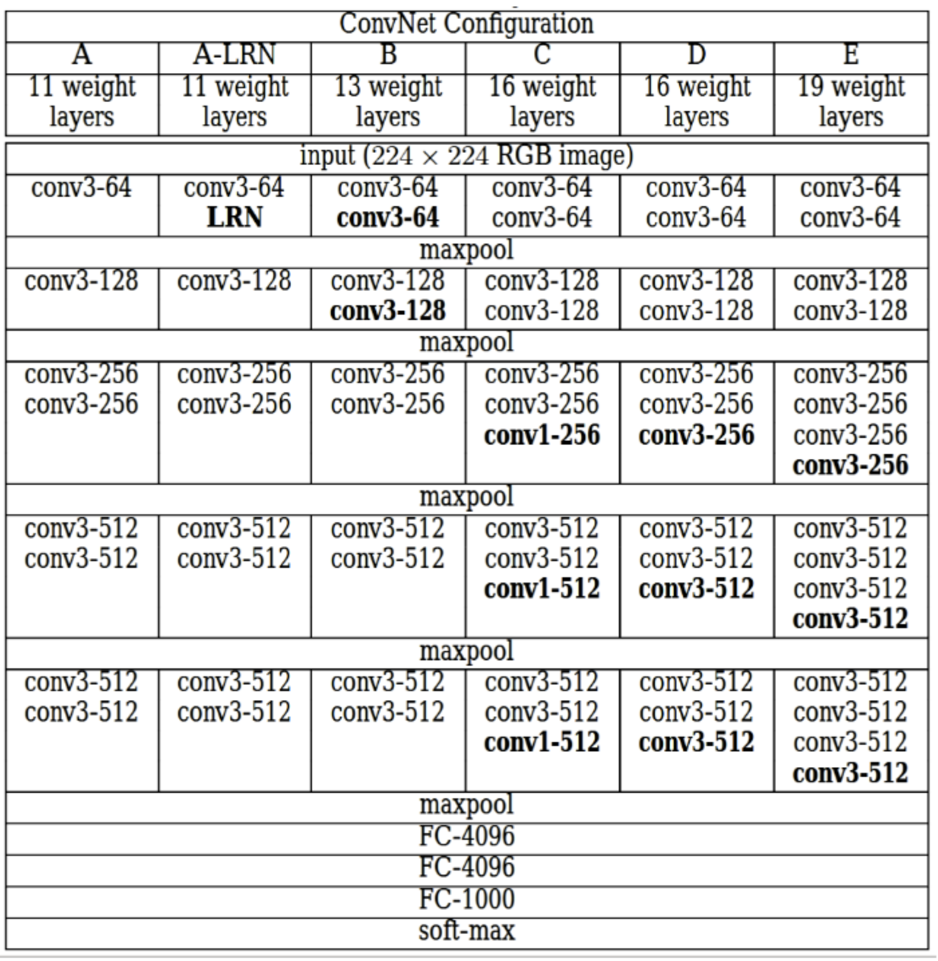
Use of Pretrained CNNs for the MURA Classification:

VGG19:

VGG19 is a popular pretrained convolutional neural network (CNN) architecture that was introduced by the Visual Geometry Group (VGG) at the University of Oxford. It is an extension of the VGG16 network and is named after the number of layers it has, specifically 19 layers.

One of the notable aspects of VGG19 is its simplicity and uniformity in architecture. The network has 16 convolutional layers and 3 fully connected layers. The convolutional layers are divided into five groups, and each group contains multiple convolutional layers with ReLU activation followed by a max-pooling layer.

The output of VGG19 is a probability distribution over the classes in the dataset it was trained on. The pretrained weights of VGG19 can be used as a starting point for transfer learning in various computer vision tasks. By fine-tuning the network on a specific dataset, it is possible to leverage the learned features and achieve good performance even with limited training data.



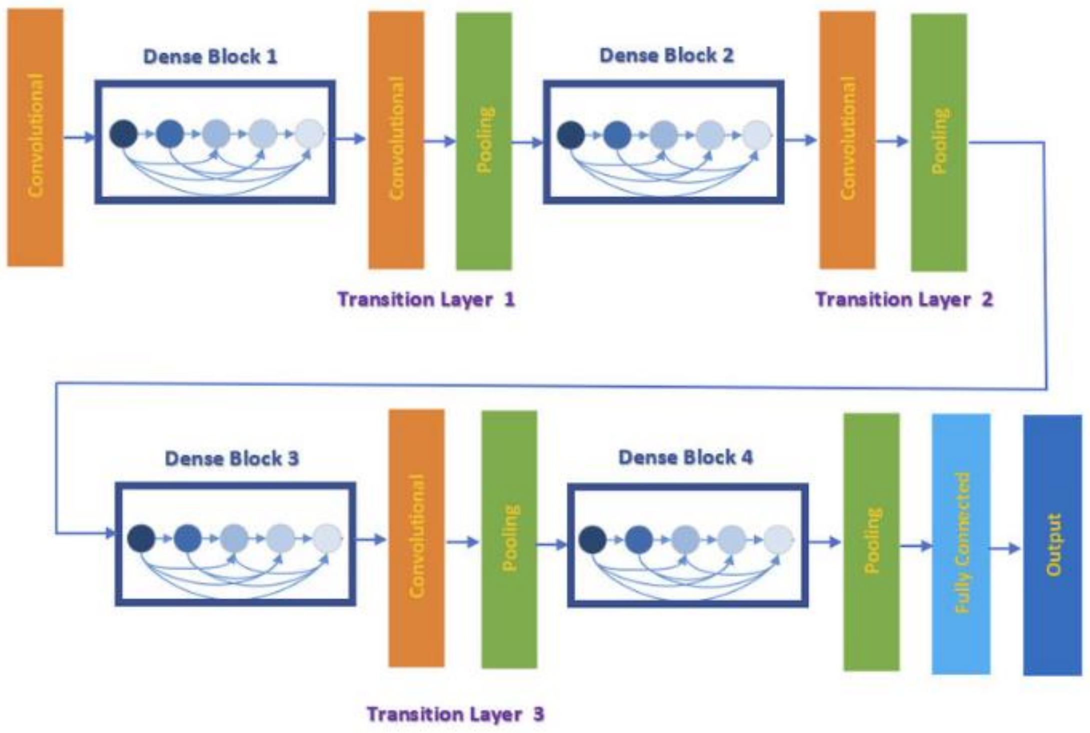
ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

<u>DenseNet:</u>

DenseNet-201 is a pretrained convolutional neural network (CNN) architecture that belongs to the DenseNet family, introduced by researchers at Cornell University. It is designed to address the challenges of information flow and parameter efficiency in deep networks. The main idea behind DenseNet-201 is the concept of dense connectivity. Unlike traditional CNN architectures where layers are connected sequentially, DenseNet-201 connects each layer to every other layer in a feed-forward manner. This dense connectivity promotes feature reuse and enhances information flow throughout the network and reduces the problem of "vanishing gradient". It enables gradients to flow directly from the output layer to the early layers, addressing the vanishing gradient problem and allowing the network to learn more effectively.

DenseNet-201 has 201 layers, which makes it suitable for complex computer vision tasks such as image classification, object detection, and segmentation. The architecture of DenseNet-201 comprises multiple dense blocks, transition layers, and a final global average pooling layer followed by a softmax classifier. Dense blocks consist of multiple convolutional layers with batch normalization and ReLU activation, where each layer takes inputs from all preceding layers in the block. Transition layers are used to downsample the spatial dimensions of the feature maps and control the number of channels to maintain an efficient network structure.

DenseNet-201 is known for its parameter efficiency, as it requires fewer parameters compared to other architectures while still achieving competitive performance. This property makes it suitable for scenarios with limited computational resources or when working with smaller datasets.

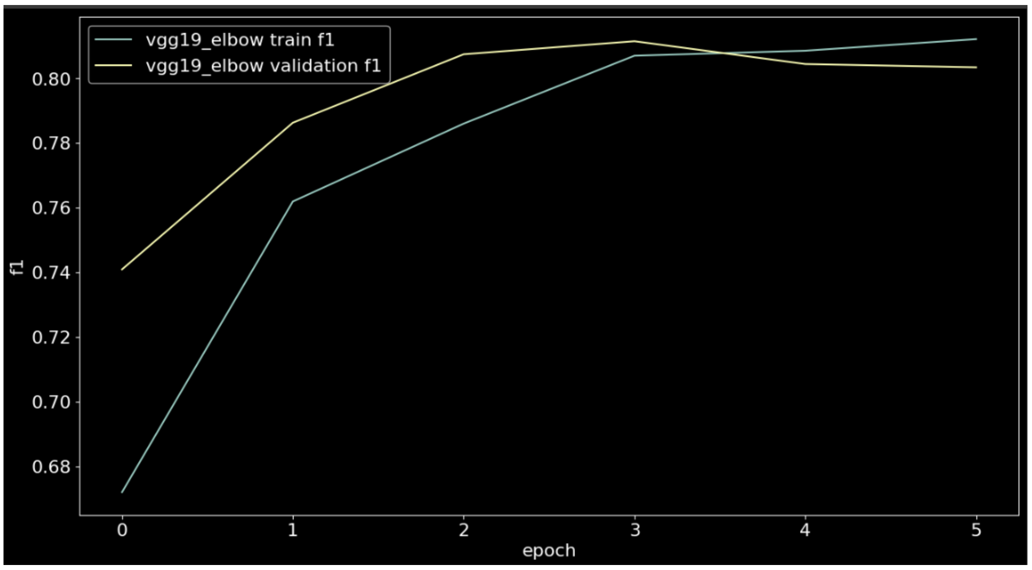


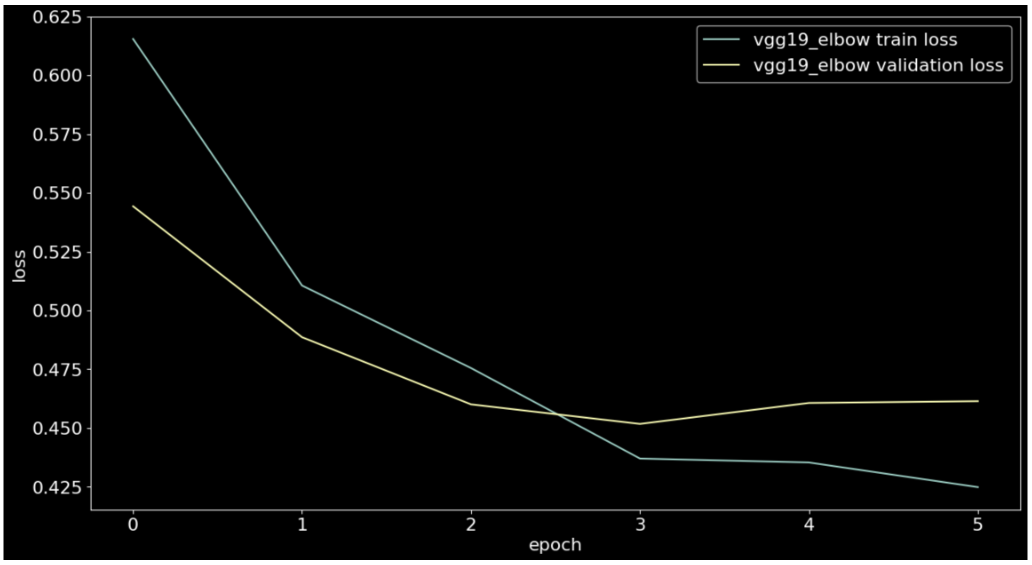
<u>VGG19 Implementation</u>:

The model had 20,2 million parameters and even using the GPU feature of the google collab, each epoch took forever. Meaning that each mistake and solution implementation was agonizing slow. To bypass this challenge, we implemented the VGG independently for 3 human parts (Elbow, Finger and Wrist) and from there we hypothesize the effectiveness of the Pretrained CNN model.

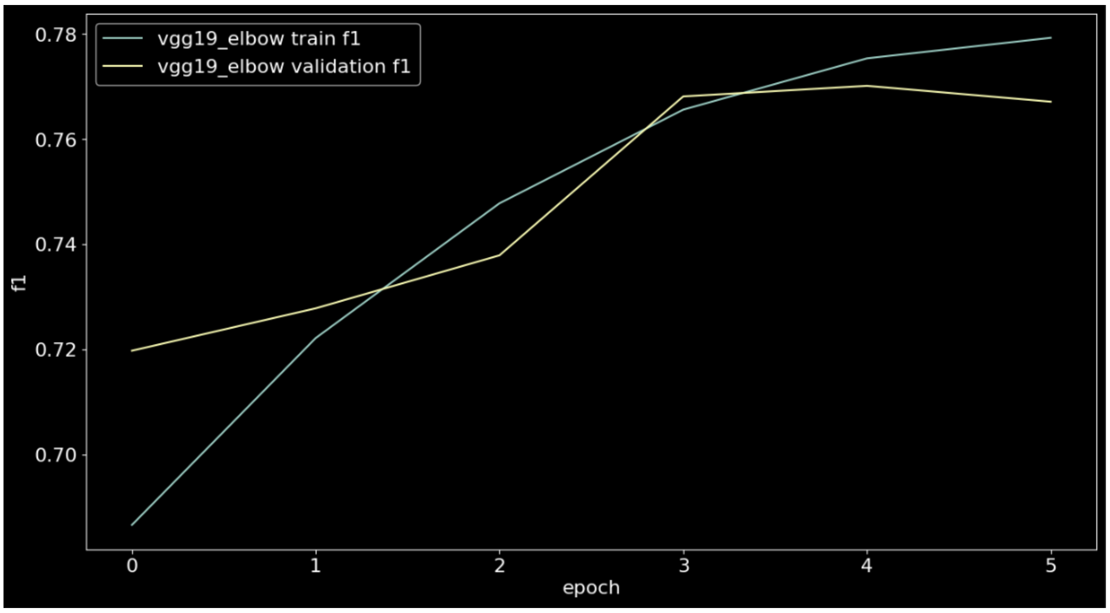
For the pretraining random zooming, rotation and shuffling was used. For the model was used the pretrained VGG19 with dropout set at 20%, batch size 32, Adam optimizer. It was run for 15 epochs with early stopping and for the monitoring was used f1 on the validation data.

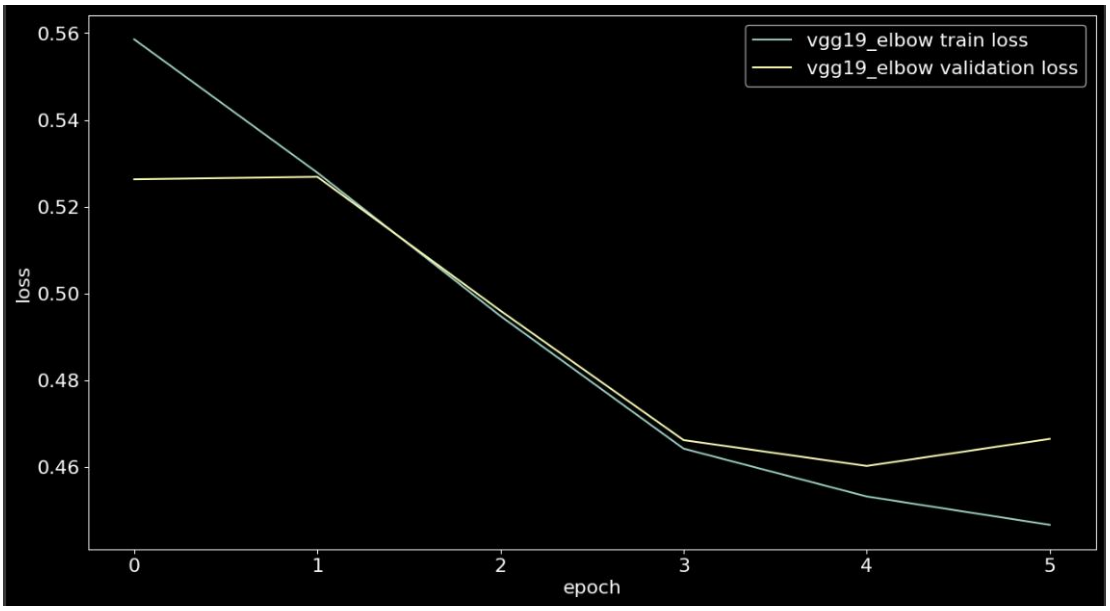
Results of the Normal Abnormal Classification for Elbow X-Rays.





Results of the Normal Abnormal Classification for Finger X-Rays.

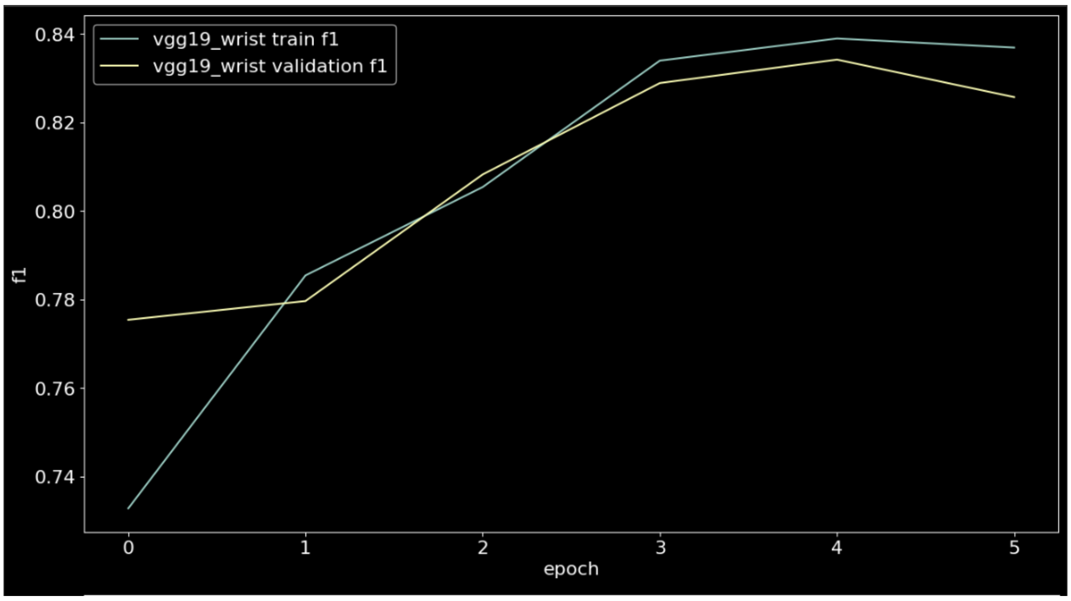


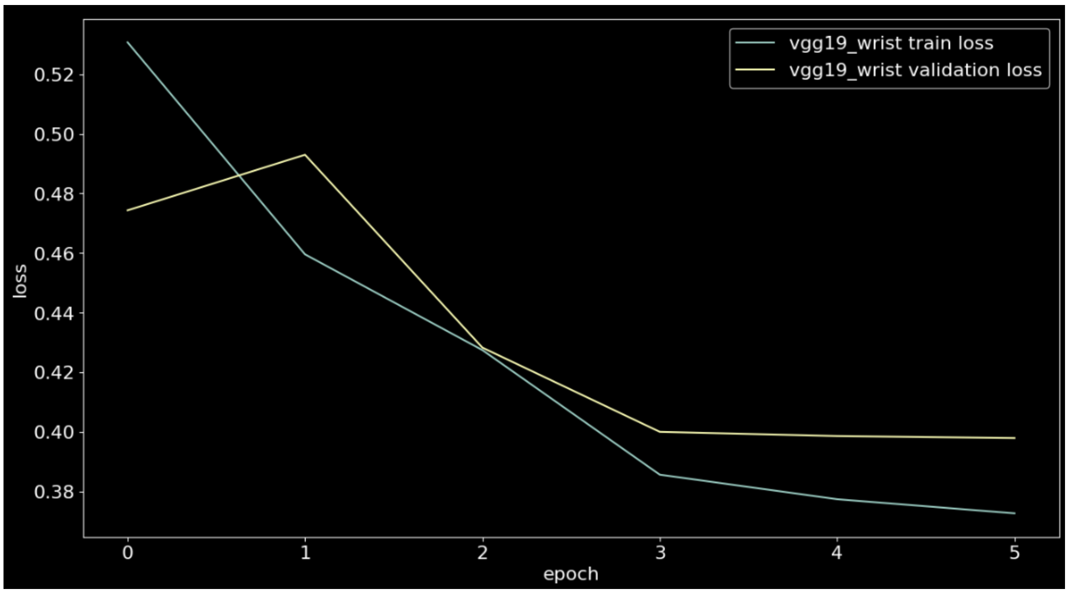


This is a great opportunity to voice the second biggest problem after the long training time. As I have already mentioned, the google collab's GPUs were used to decrease the execution time, unfortunately not long after the beginning of any attempt to solve the MURA classification the time of GPU use had already been used. As a result, even when we could use this feature we were under the sword of Damocles as many times without warning the environment would any existing runtime.

An attempted solution was the use of 4 different mail accounts, it worked for a couple of days but at the end google noticed and for this reason I decided not to run it again to print the correct legend for fear of stopping midway.

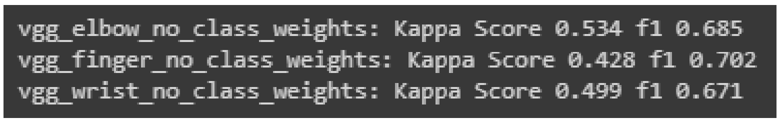
Results of the Normal Abnormal Classification for Finger X-Rays.





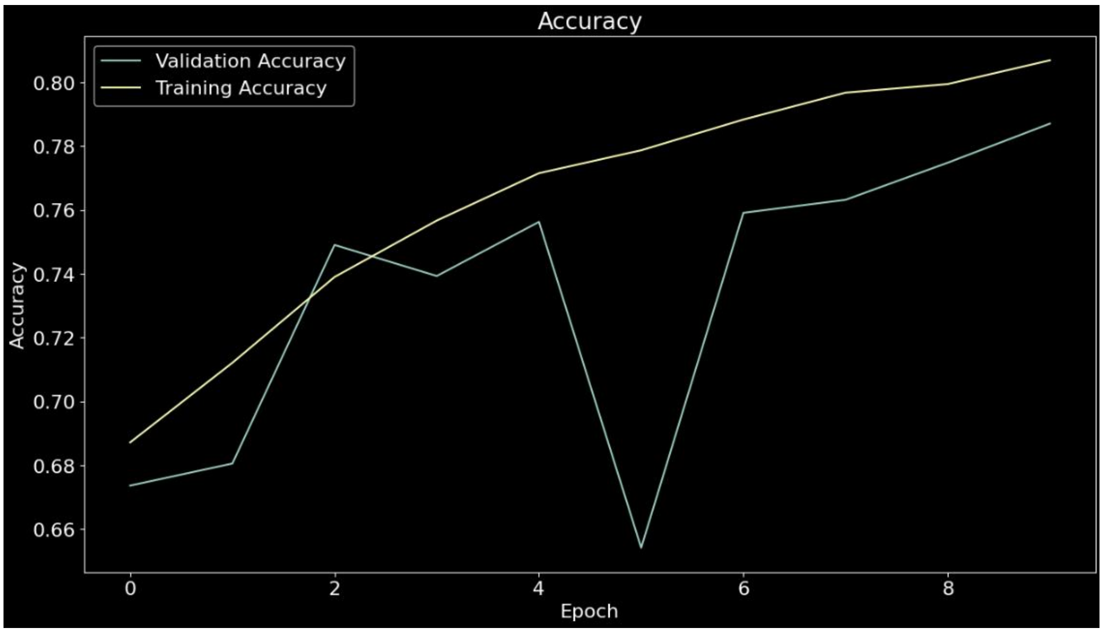
The problem of the nonavailability of the GPU continued as any change was very costly and risky to implement so careful consideration, and thought was taken before any changes. This also made difficult any tuning, that was kept to the most basic level.

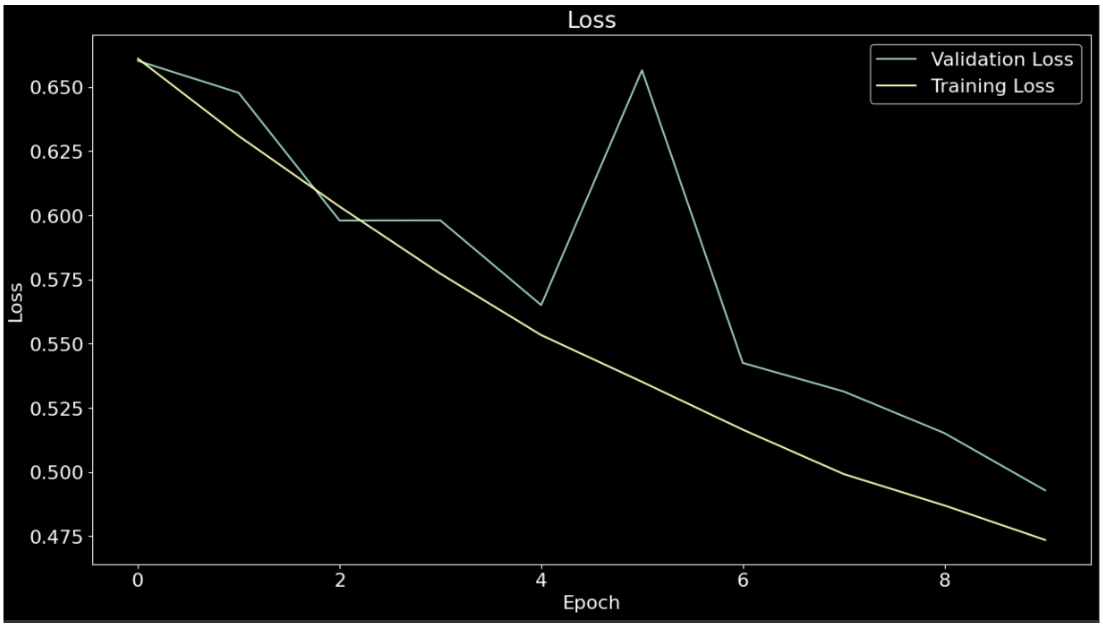
The overall result for the test set showed a significantly lower accuracy compared to the validation data. Although we can assume that with the necessary change the F1 could climb to ¾ accuracy. Especially if we take into consideration that the Cohens Cappa of the top contenders of the MURA Competition was above 75%.


```
vgg_elbow_no_class_weights: Kappa Score 0.534 f1 0.685
vgg_finger_no_class_weights: Kappa Score 0.428 f1 0.702
vgg_wrist_no_class_weights: Kappa Score 0.499 f1 0.671
```

VGG19 Implementation

Our initial idea was to implement both the VGG and DenseNet architectures and compare the result. Unfortunately, due to time restrictions, skill inadequacy and lack of the necessary resources we fall short of our goals. For the DenseNet implementation we were able to run it for all body parts. We also used shuffling and rotation. We changed the last layer to a Sigmoid for binary classification, used Adam optimizer and trained it for 10 epochs.





From our results we assume a similar quality in our results the only difference is that the DensNet has a lesser trend to overfitting and if we wait longer with more epochs of training perhaps it will achieve even lower loss. Unfortunately, by failing to have data for all body parts in VGG and successfully use the same metric for both any conclusion lacks ground.