AUEB M.S.c in Data Science (part-time)

Deep Learning - Abnormality Detection in bone X-Rays

Mura Dataset

Manolis Proimakis (P3351815)

Introduction

Mura dataset is a collection of bone X-ray images. It contains **36808** images for **training** purposes and **3197** images for **validation** purposes.

Each X-ray is classified as:

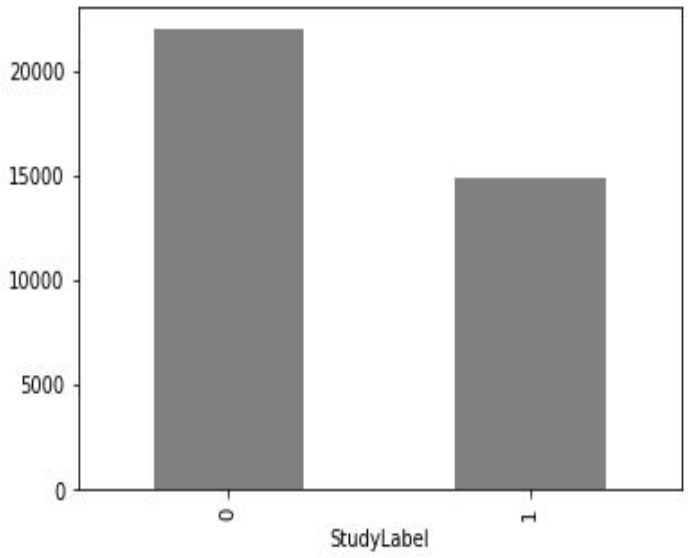
- (negative) 0: **normal**
- (positive) 1: **abnormal**

Moreover, the dataset has additional information about the body part that each X-ray refers too.

- XR_SHOULDER
- XR_HUMERUS
- XR_FINGER
- XR_ELBOW
- XR_WRIST
- XR_FOREARM
- XR_HAND

Inspection

We need to further examine the dataset to find out what is the information that we can use for our classification task.



In more analysis, the train set is containing

- 21935 normal (0) **(0.59%)**
- 14873 abnormal (1) **(0.41%)**

X-rays

Apparently, there is a small problem of class imbalance which we will handle but the issue is not very big from a first point of view. Of course, the results from further analysis and model predictions will provide further information

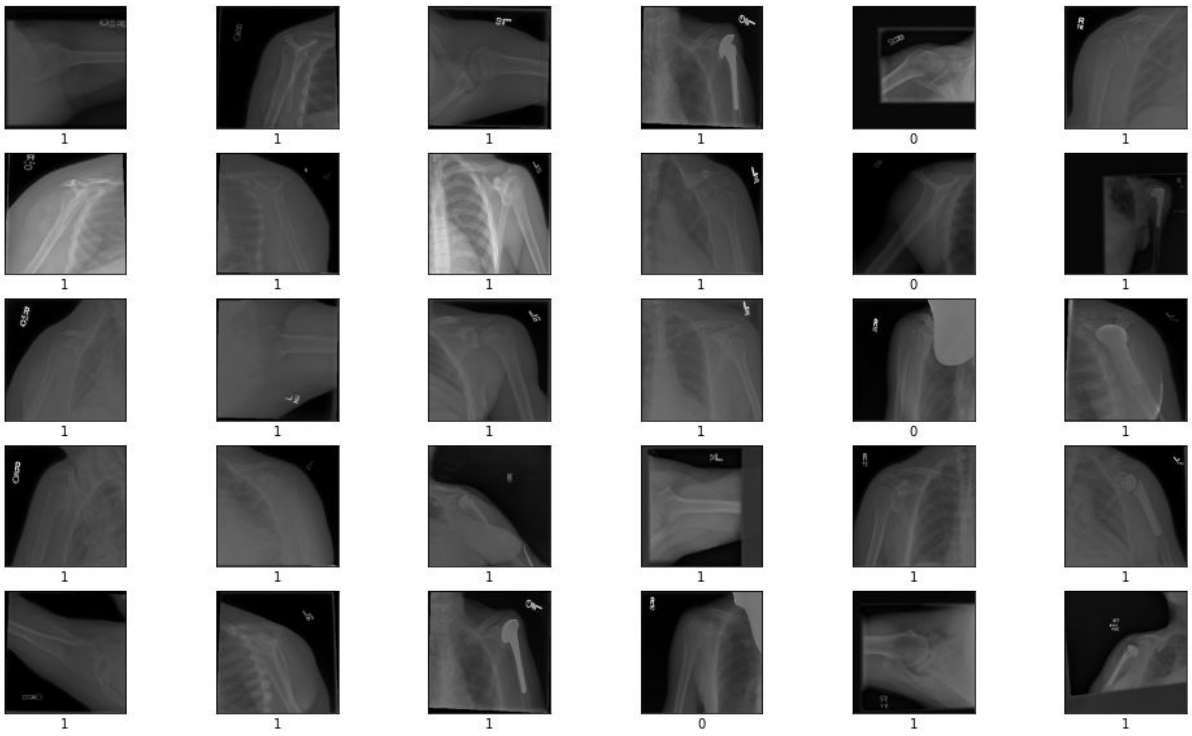
	BodyPart	StudyLabel	count	percent
0	XR_ELBOW	0	2925	59.318597
1	XR_ELBOW	1	2006	40.681403
2	XR_FINGER	0	3138	61.457109
3	XR_FINGER	1	1968	38.542891
4	XR_FOREARM	0	1164	63.780822
5	XR_FOREARM	1	661	36.219178
6	XR_HAND	0	4059	73.227494
7	XR_HAND	1	1484	26.772506
8	XR_HUMERUS	0	673	52.908805
9	XR_HUMERUS	1	599	47.091195
10	XR_SHOULDER	0	4211	50.256594
11	XR_SHOULDER	1	4168	49.743406
12	XR_WRIST	0	5765	59.116079
13	XR_WRIST	1	3987	40.883921

If we examine the dataset more closely we can see that the same analogy is common for **all** body parts but the biggest problem is on **HR_HAND** where we have only **26%** of abnormal examples.

The good part is that we have 2 easy ways to solve this problem.

1) We can fit our model by providing the distribution of classes so that we can give more attention to the classes with fewer examples
2) We can use image augmentation to create more examples, which will help us improve the accuracy of our model.

And an example of the images at hand is like below:



Preparation

Datasets

As we have seen the dataset provides training and validation examples. For the purpose of this task, I will need 3 sets

- **training_set** (for fitting my model)
- **validation_set** (for early stopping the training process when validation loss starts to drop)
- **test_set** (to evaluate my model after training completes)

So I will **stratify** split the **training examples** to keep class balance and create my **training set (0.8%), validation set (0.2%)** and I will use the **validation examples** as my **test set**

X-rays image preparation

Since the images in the MURA dataset do not have the same dimensions all images will be scaled to 256x256 and using 3 channels (RGB).

Metrics

The competition uses Kappa as the reference metric for the best model evaluation. Kappa is defined as ***K = (obeserved_accuracy - expected_accuracy)/(1-expected_accuracy)*** so this will be the key metric to evaluate the performance along with though I would like to have more metrics at hand to evaluate the performance of the network, so in general, I will take into account:

- **Loss**
- **ROC-AUC**
- **Cohen Kappa**
- **Accuracy** *(this will not be used much as a metric of how good my classifier is.)*

Optimizers

In respect to optimizers, I will use **Adam** (without initial LR) as a base for all my experiments.

Callbacks

- **EarlyStopping:** will be used for all experiments with patience of **10 epochs**. All models will be trained on **100 max epochs** and the best model with less **val_loss** will be saved for further evaluation.
- **`ReduceLROnPlateau`:** will be used for all experiments with a factor of 0.2 until 0 minimum LR.

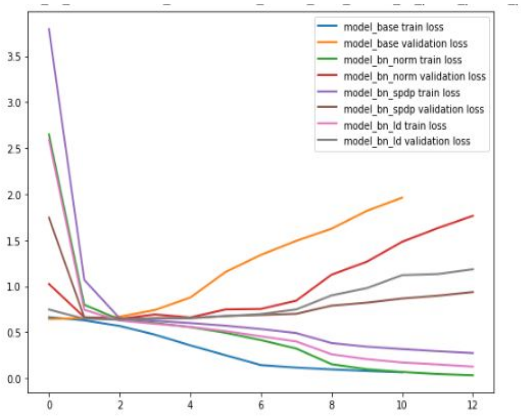
Convolutional Neural Network (CNN)

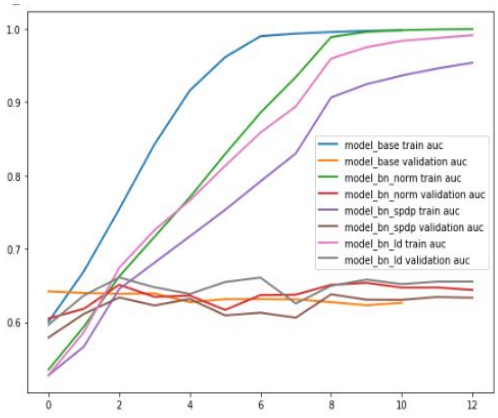
Fitting a good network from scratch for this classification task has many challenges and I will go through them in this report. First of all,I X-Rays will be scaled to (256, 256) pixels that have 3 channels (RGB) so we will need an Input layer of (256, 256, 3). And since this is a binary classification for Output I'll use a Dense layer with 1 neuron and **sigmoid activation** which will output the probability of an image belonging to each class,

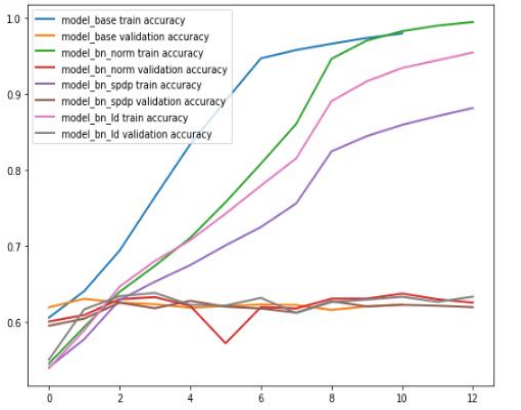
- **P <= 0.5 => 0** normal
- **P > 0.5 => 1** abnormal

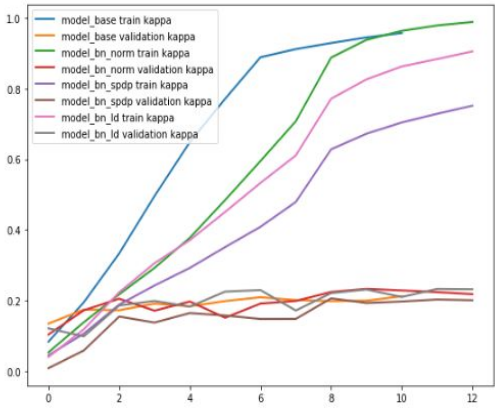
Initial network Approach

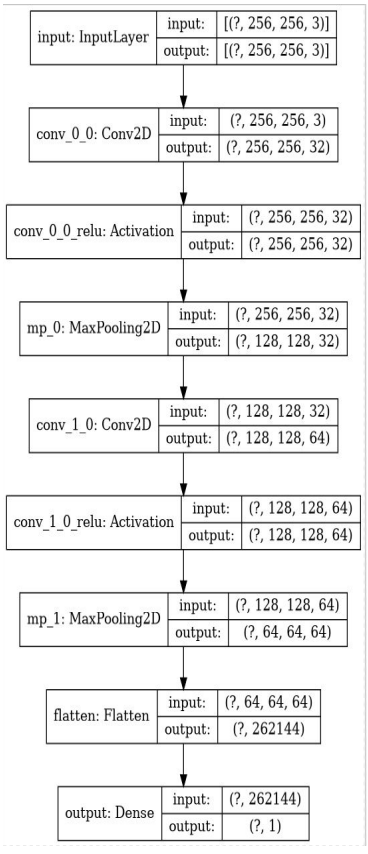
As a base network, I have used a 2 layer CNN to evaluate the performance before any optimizations.











From the training, validation plots, I can see that a simple CNN is starting to overfit the training set from the 1st epoch, so I need some form of regularization. To fix this I considered 3 options:

- **Dropout after each MaxPooling layer**. Which I later **disregarded** as Dropout doesn't make sense to be used on Convolutional layers as due to the spatial relationships in features maps, activations can be highly correlated which makes dropout ineffective. I could use dropout though, after the fully connected Flatten layer. [1]
- **BatchNormalization after the convolution layer.** From the plots, it has a good effect on regularizing the model and also it makes it more stable during training. (red, green lines) [ref]
- **SpatialDropout** after the activation of each convolution layer which will drop entire feature maps before max pooling. (purple, brown lines)
- **Dropout** only after the fully connected layer before the sigmoid output

	model_base	model_bn_norm	model_bn_spdp	model_bn_ld
epoch	[0]	[2]	[2]	[2]
loss	[0.6640332937240601]	[0.6393173336982727]	[0.6531590223312378]	[0.627886950969696]
auc	[0.6006818413734436]	[0.6623112559318542]	[0.6451549530029297]	[0.6741585731506348]
accuracy	[0.6059906482696533]	[0.6398831605911255]	[0.6290497779846191]	[0.6468111276626587]
kappa	[0.08346885442733765]	[0.2173094749450684]	[0.18825709819793698]	[0.22310250997543332]
val_loss	[0.6446119546890259]	[0.6461774706840515]	[0.6477799415588379]	[0.6354892253875732]
val_auc	[0.642005980014801]	[0.6510496735572815]	[0.6336060762405396]	[0.6609166860580444]
val_accuracy	[0.6195327639579773]	[0.6302635073661804]	[0.6256452202796936]	[0.6342026591300964]
val_kappa	[0.13448554277420044]	[0.20502185821533206]	[0.15445417165756226]	[0.1857135891914368]

From these methods, I have chosen to use **always BatchNormalization** since it has a better training curve directly after the Convolutional layer and before the relu activation and **Dropout on the fully connected layer** because it has comparable results with respect to the validation curve, in relation to spatial dropout but it has better validation loss from an average of 5 runs as we can see.
Moreover I will use other ways to also improve the training curves like augmentation and i don't want to have so much dropout after each layer from this early step.

The performance is not optimal till now but I expect to have better results with bigger models and by augmenting the data which I will do later after I conclude to good architecture that can be fitted by my PC.

Bigger Models

For simplicity, I will define the term cell. Convolution layers, in general, provide better results by stacking many of them. By doubling the filters of each layer each subsequent convolution will be able to detect from simpler to more complex features. The hard part is to find out a good combination for each cell and a way to combine stacked layers:

- without losing information
- not overfit early

Subsequently, I will try to stack more of these cells pooling after each layer, for this I will use **MaxPolling** with stripes of **(2,2)** to downsample the input and reduce the parameters (because keeping the same size causes "out of memory" issues) but increase the filters after each layer to increase the feature maps and detect more complex patterns.

Moreover, to tackle class imbalance on our training set. I will use the class weights while fitting the model, to penalize the loss for the majority class and increase attention/optimization with respect to the minority.

Below we can see the results for more deep CNN models following the same pattern.

	epoch	loss	auc	accuracy	kappa	val_loss	val_auc	val_accuracy	val_kappa
model_3_layers	[7]	[0.5315427184104919]	[0.8076416850090027]	[0.7280785441398621]	[0.4454463720321655]	[0.6309834718704224]	[0.7043504118919373]	[0.6498234272003174]	[0.28890204429626465]
model_4_layers	[13]	[0.4196323454380035]	[0.8887190818786621]	[0.804625391960144]	[0.5993938446044922]	[0.5990333557128906]	[0.7561915516853333]	[0.6958706974983215]	[0.3767048716545105]
model_5_layers	[14]	[0.4224697947502136]	[0.8879661560058594]	[0.8071045279502869]	[0.6020364761352539]	[0.528891563415527 3]	[0.8073039054870605]	[0.742325484752655]	[0.46302330493927]
model_6_layers	[17]	[0.4020474553108215]	[0.8979669809341431]	[0.8173945546150208]	[0.6230074167251587]	[0.5235744118690491]	[0.8172265887260437]	[0.7553653717041016]	[0.4880296587944031]

Fig - Multilayer CNN training comparison

From the training results I see that adding more convolutional layers along with class weighing achieves better results while training the model and the bigger the models the better the **kappa score** it achieves with the best model being a 6 layer stacked CNN.

	model_3_layers	model_4_layers	model_5_layers	model_6_layers
loss	[0.6664800047874451]	[0.679706871509552]	[0.5886131525039673]	[0.5825543403625488]
auc	[0.6737177968025208]	[0.7067725658416748]	[0.7787353992462158]	[0.7931535840034485]
accuracy	[0.6212073564529419]	[0.6443541049957275]	[0.7037848234176636]	[0.7231779694557191]
kappa	[0.2394890785217285]	[0.28576910495758057]	[0.4037413001060486]	[0.44298869371414185]

Fig - Multilayer CNN evaluation comparison

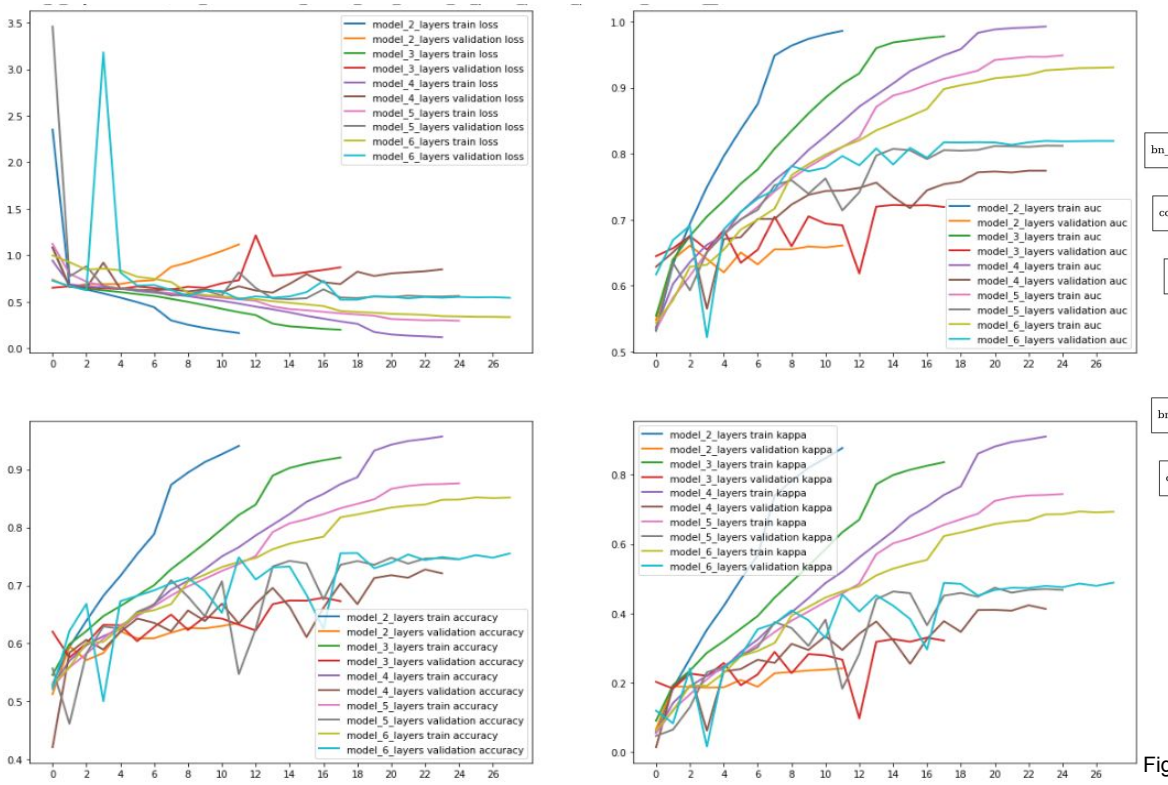


Fig- Multilayer CNN training curves

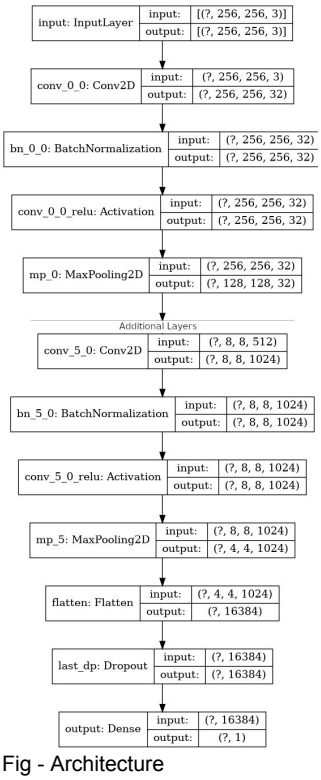


Fig - Architecture

From the training curves I can see that by adding more layers, Generalization of the model has also improved as a 2 layer CNN achieves the best val_loss on **2nd epoch** while the

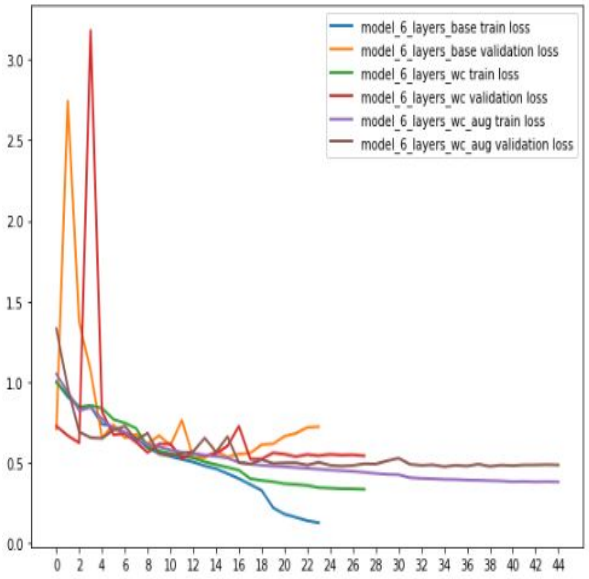
one with 6 layers on **epoch 17** and until now our best model achieves **0.44 kappa score**, with a models that has **6,308,609** trainable parameters. The improvement comes from letting the model detect more complex patterns in each image because we have more convolutional layers with more filters and not rely on simple patterns to make predictions.

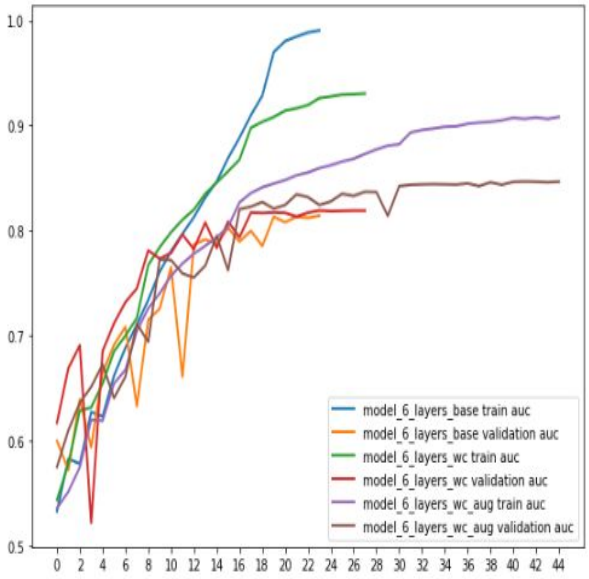
Data augmentation

With all the above ways we were able to achieve a more stable model and a good architecture. All these networks have 32 starting filters that are doubled after each convolutional block which was mostly done to save training time and prevent memory issues.

Probably we could add more CNN layers and more filters and achieve better scores but this requires a different training process with better hardware. Here comes data augmentation, augmenting the images with random (non destructing) transformation could improve the generalization ability of our 6 layer model and improve score as we would feed it with different images of the same content. This would prevent from overfitting on the training data and be able to recognize more generic patterns. For this purpose we will fit the same model with augmented X-Rays by applying these 3 transformations

- Horizontal Flip
- Vertical Flip
- Random rotations in the range of (-30, 30) degrees





	loss	auc	accuracy	kappa
model_6_layers_base	[0.5894367694854736]	[0.7727674245834351]	[0.7006568908691406]	[0.3919324278831482]
model_6_layers_wc	[0.5825543403625488]	[0.7931535840034485]	[0.7231779694557191]	[0.44298869371414185]
model_6_layers_wc_aug	[0.5159420967102051]	[0.8317021131515503]	[0.7657178640365601]	[0.5274899601936339]

And as expected by training the same model with **augmented X-rays**, the model has **improved** much over the previous networks. It is also possible that more evolved transformations would prevent more early overfitting and improve our **kappa score** on the test set which as of now increased to **0.527.**

Ensemble

One addition that could potentially enhance our model prediction capability is to create a different model per body part and then ensemble these models by averaging the probabilities they give for every image. The intuition behind that is that each classifier would be specialized to predict abnormalities on specific body parts and the ensemble of them would have better accuracy. To evaluate that I will train 7 different models of the same type by filtering the images per each body part and applying the same augmentation pattern as in the “all parts” model.

Unfortunately the results seem disappoint after the evaluation:

	epoch	loss	auc	accuracy	kappa	val_loss	val_auc	val_accuracy	val_kappa
model_6_layers_ELBOW	[5]	[1.0459190607070925]	[0.5947659611701965]	[0.5707403421401978]	[0.14383035898208618]	[0.6708067655563354]	[0.6558191776275635]	[0.6160081028938293]	[0.2324486970901489]
model_6_layers_FINGER	[33]	[0.5121141672134399]	[0.8307381272315979]	[0.7372673749923706]	[0.4711514711380005]	[0.5346609354019165]	[0.8014505505561829]	[0.7211350202560425]	[0.4315729141235352]
model_6_layers_FOREARM	[35]	[0.5863669514656067]	[0.7839041948318481]	[0.7260273694992065]	[0.4258512258529663]	[0.4935268759727478]	[0.829626739025116]	[0.7671232819557191]	[0.4947973489761353]
model_6_layers_HAND	[5]	[1.0819077491760254]	[0.5562950372695923]	[0.5365358591079712]	[0.06779724359512329]	[0.5744850039482117]	[0.6105284690856934]	[0.7321911454200745]	[0.0]
model_6_layers_HUMERU	[4]	[0.8551684617996216]	[0.5580573081970215]	[0.5358898639678955]	[0.07562899589538574]	[0.6926431655883789]	[0.5728395581245422]	[0.5215686559677124]	[0.003842473030090332]
model_6_layers_SHOULDER	[42]	[0.5663805603981018]	[0.7907875776290894]	[0.72847980260849]	[0.456645131111145]	[0.5806487202644348]	[0.7687239646911621]	[0.7165871262550354]	[0.4329449534416199]
model_6_layers_WRIST	[29]	[0.4226312935352325]	[0.8871933221817017]	[0.8092552423477173]	[0.6060657501220703]	[0.441970556974411]	[0.8600398898124695]	[0.7985648512840271]	[0.5728696584701538]

It is pretty clear that training different classifiers for each body part does not have good results as seen above. The reason for that is not related to the actual model but to the amount of X-rays we have for each body part. As we can see for the classes that we have problems, i.e. **HUMERUS** and **FOREARMS,** we don’t have many examples to train a good model and for **HANDS** there is a huge class imbalance which does not allow the model learning to predict correctly the abnormal classes.

	BodyPart	Examples
0	XR_ELBOW	4931
1	XR_FINGER	5106
2	XR_FOREARM	1825
3	XR_HAND	5543
4	XR_HUMERUS	1272
5	XR_SHOULDER	8379
6	XR_WRIST	9752

Transfer Learning

As we have seen, training big models on personal computers is very hard and takes much time. To solve this issue we can use transfer learning and take advantage of pre-trained large networks like DenseNet and Resnet. We will try both of these networks but put more emphasis on DenseNet since a Denset with 169 layers was used in the initial Mura paper.

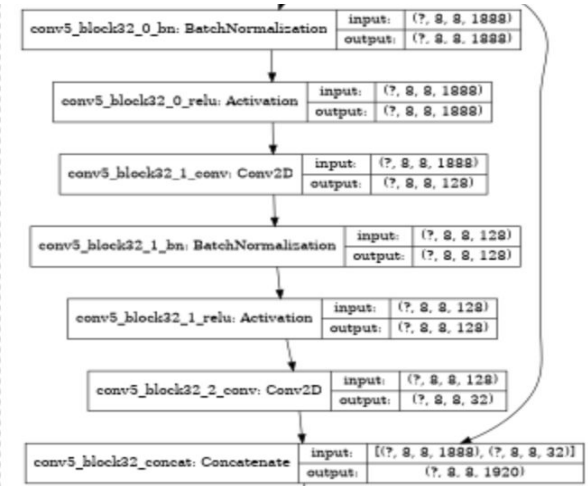
DenseNet 201

For this experiment I have chosen to use DensetNet 201 that is provided by tensorflow with pre-trained weights from ImageNet. The input shape will be the same as my custom networks and the last layer will be a **GlobalAveragePooiling,** the reason behind that is that the output shape of DenseNet is a (8, 8, 1920) which cannot be Flattened due to OOM issues. So Average pooling over the output will provide a 1920 dense vector with the average of the most prominent features. Moreover, data will be augment here too, after seeing the good results in our custom model.

Initially I will try 4 models:

1) The first will have all layers freezed, and the only trainable parameters will be 1920 for the weights from the GlobalAveragePooling to the output layer, to establish a baseline

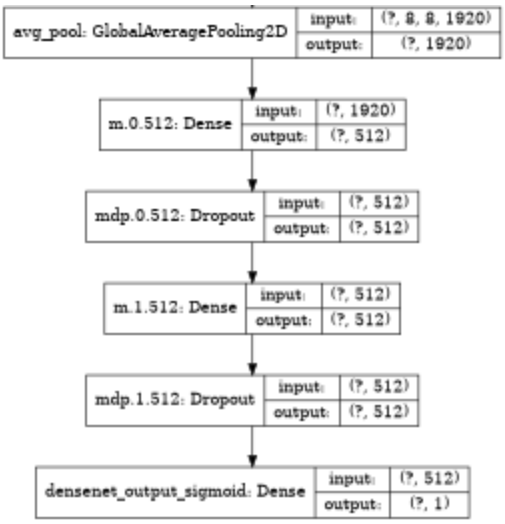
2) For the second one I will unfreeze and retrain some of the top layers of DenseNet to let it predict better our X-Rays. Of course, we can't and should not retrain the full DenseNet since it contains more than 18million trainable parameters and it is already pretrained on a huge dataset which is the point of transfer learning. On the other hand, the bottom convolutional layers learn more simple features that generalize well but the top layers if retrained can move the pretrained weights closer to our dataset and we will let them specialize to new features.



The last 2 dense blocks (31 and 32) each contain 2 Conv2D layers and their output is then concatenated in a feedforward manner. So we will make trainable the layers after **690** that contains

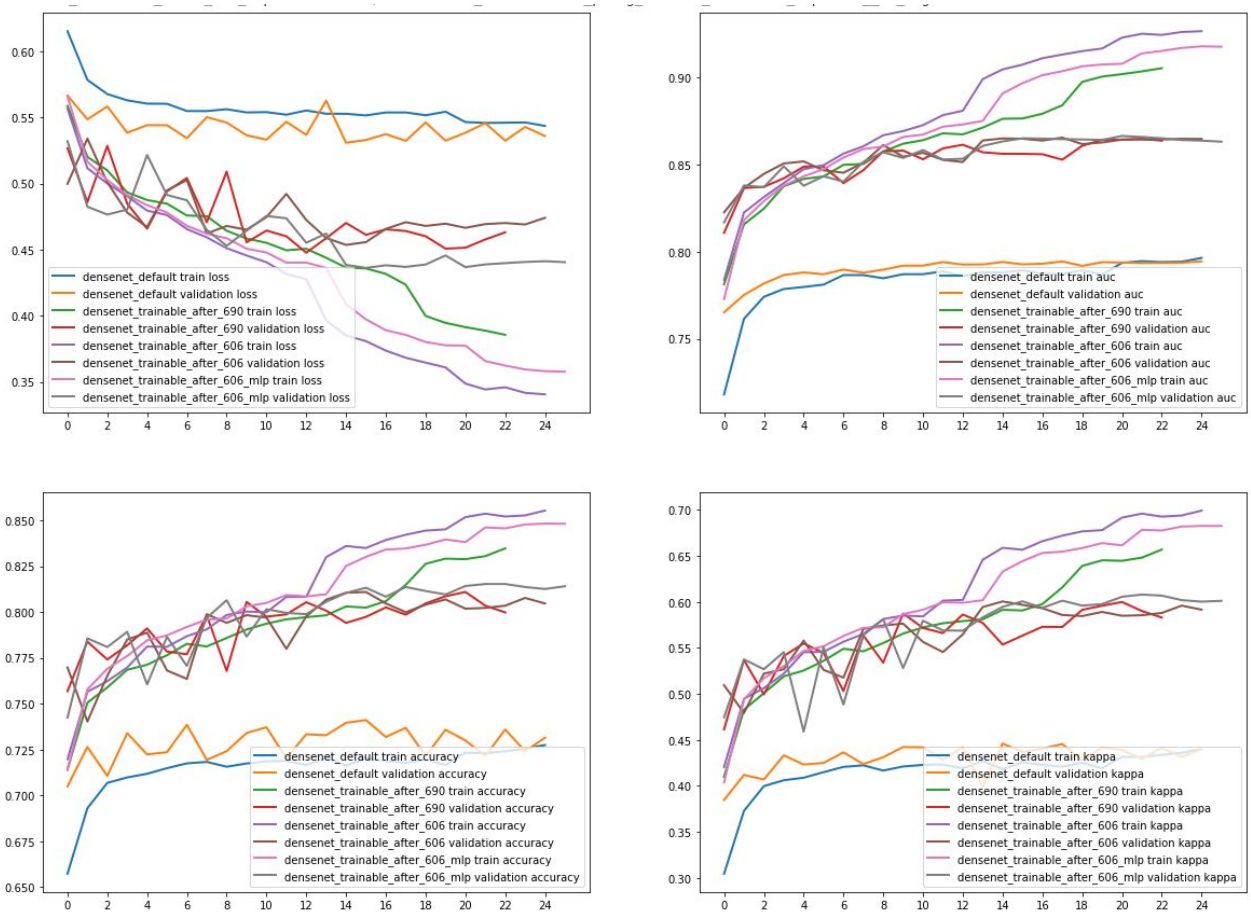
- **693** (conv5_block31_1_conv)
- **698** (conv5_block31_2_conv)
- **700** (conv5_block32_1_conv)
- **703 (**conv5_block32_2_conv)

3) I will make trainable the blocks from 19 and onwards to check if it helps the increase the validation loss



4) I will add a 2 layer MLP as the classification head. (as seen on the right) with dropout of 0.25 to prevent early overfitting.

As can be seen in the results below:



	epoch	loss	auc	accuracy	kappa	val_loss	val_auc	val_accuracy	val_kappa
densenet_default	[14]	[0.5528280735015869]	[0.788080096244812]	[0.7163282036781311]	[0.4182202219963074]	[0.5309038162231445]	[0.7940923571586609]	[0.7396088242530823]	[0.445992648601532 04]
densenet_trainable_after_690	[12]	[0.4507141411304474]	[0.8673561811447144]	[0.7971880435943604]	[0.5790766477584839]	[0.4477538168430328]	[0.8614115118980408]	[0.8053517937660217]	[0.5862799882888794]
densenet_trainable_after_606	[14]	[0.3850914239883423]	[0.9046438932418824]	[0.836004912853241]	[0.6587384939193726]	[0.453654408454895]	[0.8650296926498413]	[0.8105134367942809]	[0.600480854511261]
densenet_trainable_after_606_mlp	[15]	[0.3973292112350464]	[0.8967631459236145]	[0.8300617933273315]	[0.6443120241165161]	[0.4362366795539856]	[0.8651068806648254]	[0.8132300972938538]	[0.600592315196991]

- The simple network is learning very little from training which is expected as it has only 1920 trainable parameters and we have done that just to establish a baseline, but the loss achieved is comparable to our biggest CNN, and the kappa of 0.445 is not so bad for such a simple model. (blue and orange lines)

- In addition, retraining the **top 2** (green and red lines) and **top 10** (purple and brown lines) blocks of densenet both make the model overfit early but at the same time we have better results on the validation set, as the pretrained weights are adjusted closer to my dataset. Unfortunately retraining more blocks made the networks overfit very early and the training curves were not good as expected.

- Last but not least the **addition of an MLP** after the DenseNet output and our Classification head seems to have the best results. Currently we have added only 2 layers with 512 neurons each and a dropout of 0.25 after each layer, but still we could tune these MLP layers with more Dense units and even bigger dropouts to improve stability and have better results.

So after evaluation on the test set the scores of the above are:

	loss	auc	accuracy	kappa
densenet_default	[0.5565700531005859]	[0.7884572148323059]	[0.7228651642799377]	[0.4397755861282349]
densenet_trainable_after_690	[0.4704350531101227]	[0.8547149300575256]	[0.7851110696792603]	[0.5663684606552124]
densenet_trainable_after_606	[0.48848319053649897]	[0.8556095957756042]	[0.7888645529747009]	[0.5746790766716003]
densenet_trainable_after_606_mlp	[0.4570291638374329]	[0.8596268892288208]	[0.800750732421875]	[0.597819447517395]

So especially after retraining the **top 12 blocks** of densenet we achieved a **0.5746 Kappa score** with a pretty good **auc of 0.855** and with the same model and the addition of MLP head the test_loss is even higher with a Kappa of **0.597** and test_loss of **0.45.**

Best Architecture

From the results the best model is the one using

- **The pretrained DenseNet weights**

  Because DenseNet was pretrained using a very large size of examples from imagenet which improves generalization as the weights of the convolutional layers are already adjusted to detect patterns from many images. Of course if we had enough examples to fully retrain such a large network we could achieve very good results without many more changes.

- **Retraining the last 12 blocks of DenseNet**

  As expected though the problem is that our X-Rays are not similar to ImageNet, so even though the initial results (without retraining any layers) as seen previously are very promising we can't expect that we will have a good model with the pretrained weights are they were calculated based on different images that contain different patterns. For this purpose we can retrain some of the top layers of DenseNet. The intuition on that is that in all CNN networks bottom convolutions are not very specialized to the dataset and detect simple generic patterns while top layers that have more filters are able to detect combinations of patterns which can be specialized to our X-Rays.

- **Finally the addition of MLP on the classification head** helps to create a correlation on the feature output of the DensNet base Model and effectively weigh these outputs to our categorization.

The custom CNN is not so bad though, maybe it cannot be used for the competition but provided that we had more training examples it would provide better results. In any case transfer learning with pretrained networks was introduced to tackle this problem from the start.

Code explanation

I have broken my code in parts:

src/MuraLoader: is the data loader to get the images from the ./data folder

- It contains logic to parse the names of the images
- It handles the train validation and test split part with static random seed
- It creates the train validation and test generators
    - Train generator is always shuffled and augmentation can be enabled conditionally
    - Validation and test generator are never shuffled and never augmented.
- It filters the initial dataset by body part which was used for the ensemble

For example to get the sets and generators for HANDS I am using:

[17]:
```
train_set, validation_set, test_set = muraLoader.get_sets(body_part='XR_HAND')

train_generator_temp, _, _ = muraLoader.get_generators(train_set, validation_set, test_set)
batch = train_generator_temp.next()
images = batch[0]
categories = batch[1]

# print(images.shape)
# print(categories)
```

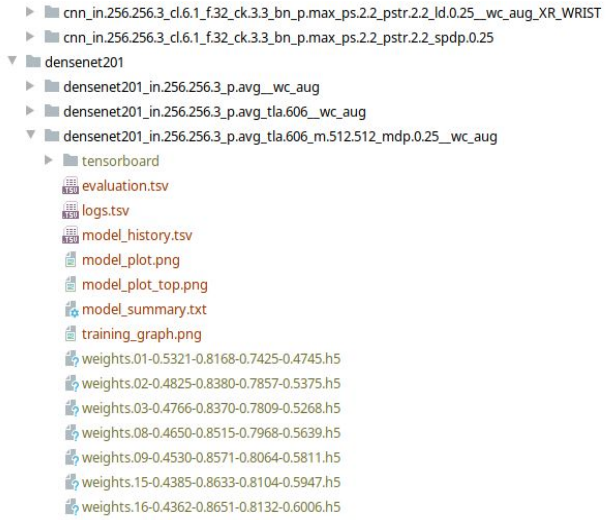
```
Get sets only for XR_HAND
Creating train generator
augment:  False - train:  True
Found 4434 validated image filenames.
Creating validation generator
augment:  False - train:  False
Found 1109 validated image filenames.
Creating test generator
augment:  False - train:  False
Found 460 validated image filenames.
```

src/models/ModelRunner: This is a class that I used to wrap common functionality for compiling/fitting all models and save the results

- It provides functionality to conditionally enable class weighting
- It provides functionality to conditionally enable data augmentation
- Add the same callbacks and metrics to all trainings
- Saving each model's related outputs to its respective folder.
- Model reload based on the model name.

For example for every model trained i am saving everything in clean way under, **./models**

- The model_training_history.tsv
- The model_plot.png has the model schematics
- The evaluation.tsv
- The training graphs i have presented above
- The weights of the best model
- And the tensorboard logs for each model which I used to evaluate if the models were actually good.



And an explanation of how the code is working.

[38]:
```
custom_cnn_model = custom_cnn_builder(starting_filters=32, conv_layers=6, convs_per_layer=1, pooling='max', batch_norm=True, last_dropout=0.25)
modelStats = runner.run(model=custom_cnn_model, overwrite=False, verbose=1, weight_classes=True, augment=True, body_part='XR_WRIST')
display(modelStats.getBestEpoch())
display(modelStats.getEvaluation())
```

```
custom/cnn_in.256.256.3_cl.6.1_f.32_ck.3.3_bn_p.max_ps.2.2_pstr.2.2_ld.0.25__wc_aug_XR_WRIST
/home/manos/servers/jupyterhub/user_dirs/manos/AUEB Data Science/6 - Deep Learning (INF401) - Prodromos Malakasiotis/Assignments/Assignment 2/models/custom/cnn_in.256.256.3_cl.6.1_f.32_ck.3.3_bn_p.max_ps.2.2_pstr.2.2_ld.0.25__wc_aug_XR_WRIST
```

	loss	auc	accuracy	kappa	val_loss	val_auc	val_accuracy	val_kappa	lr
29	0.422631	0.887193	0.809255	0.606066	0.441971	0.86004	0.798565	0.57287	0.0002

	loss	auc	accuracy	kappa
0	0.491796	0.82975	0.769347	0.524725

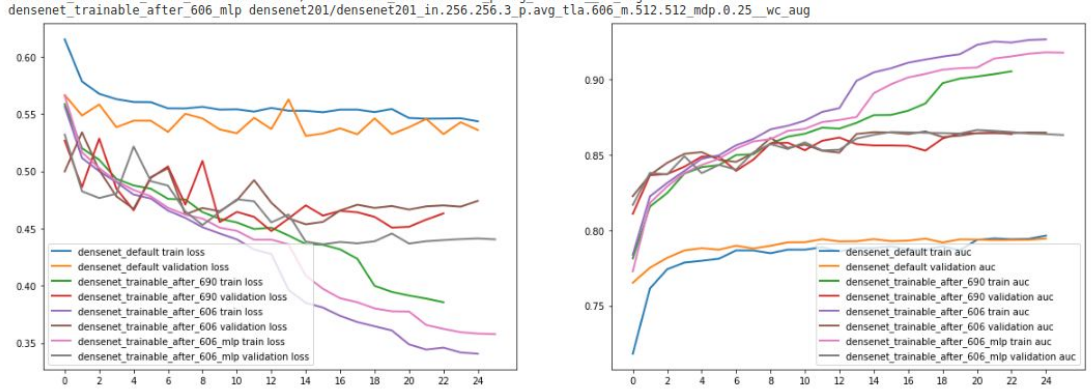
More functionality can also be seen on **Assignment 2 - 2.2 - Custom CNN.ipynb**

Moreover in this place i have added functionality to combine metrics from different models which i have used on the report.

2. DenseNet

```
ModelRunner.plot_histories({
    "densenet_default":              "densenet201/densenet201_in.256.256.3_p.avg__wc_aug",
    "densenet_trainable_after_690": "densenet201/densenet201_in.256.256.3_p.avg_tla.690__wc_aug",
    "densenet_trainable_after_606": "densenet201/densenet201_in.256.256.3_p.avg_tla.606__wc_aug",
    "densenet_trainable_after_606_mlp": "densenet201/densenet201_in.256.256.3_p.avg_tla.606_m.512.512_mdp.0.25__wc_aug",
})
```

```
densenet_default densenet201/densenet201_in.256.256.3_p.avg__wc_aug
densenet_trainable_after_690 densenet201/densenet201_in.256.256.3_p.avg_tla.690__wc_aug
densenet_trainable_after_606 densenet201/densenet201_in.256.256.3_p.avg_tla.606__wc_aug
densenet_trainable_after_606_mlp densenet201/densenet201_in.256.256.3_p.avg_tla.606_m.512.512_mdp.0.25__wc_aug
```



More functionality can also be found on **Assignment 2 - 0 - Report.ipynb**

src/models/builders: contains 4 functions

1) Custom cnn builder (i used it to move out some common tasks when building the model architecture) it basically creates the name of the model based on the parameters which will later be saved by ModelRunner.

```
custom_cnn_model = custom_cnn_builder(starting_filters=32, conv_layers=6, convs_per_layer=1, pooling='max', batch_norm=True, last_dropout=0.25)
```

2) DenseNet Builder (it downloads the pre-trained densenet application from tensorflow.
    a) Conditionally make last layers trainable or not
    b) Conditionally include MLP head.

```
densenet201 = densenet201_builder(pooling='avg', trainable_layers_after=606, mlp=[512, 512], mlp_dropout=0.25)
```

3) Resnet Builder: Same as above
4) FeedForward CNN builder: An attempt that I have made on creating a CNN that after each block combines the weights of the n-1 Conv block with the n-2 Conv to improve information Flow after reading DenseNet paper

More functionality can also be found on

- **Assignment 2 - 2.2 - Custom CNN.ipynb**
- **Assignment 2 - 3 - Transfer Learning.ipynb**

Additional Resources

Everything related to this project can be found on

Google Drive: here
Github: here

Mura Dataset Inspection: here
Reporting Graphs and metrics: here
Custom CNN experiments: here
Transfer Learning (DenseNet): here

References:

Dropout in convolutional networks

[1] Article: Dropout in CNN 1
[2] Article: Dropout in CNN 2
[3] Analysis of the Dropout Effect in Convolutional Neural Networks

BatchNormalization

[1] Batch Normalization for Training Deep neural networks
[2] Deep Residual Learning for Image Recognition

DenseNet

[1] Densely connected Convolutional Networks Paper
[2] Densely Connected convolutional Networks Paper Review