Course/Programme	BSc (Hons) Computing
Module Name and Code	Machine Learning (SWE6204)
Student ID	2011184
Tutor	Shivang Shukla
Assessment Number	2
Assessment Type	Portfolio
Assessment Title	Mini-Project and Report
Weighting	50% of overall module grades (Task 1, 2 and 4 - 80%, Task 3 - 20%)
Issue Date	01/04/2023
Submission Deadline	11/05/2023 at 23:59
Learning Outcomes Assessed: LO3: Demonstrate the application of machine learning algorithms to solve real-world problems. LO4: Critically evaluate the performance of machine learning solutions and identify the scope of improvements and optimisations.	

Student ID#: 2011184

Identification of Bone Fractures from Hand X-ray Images via Machine Learning Models

Contents

Table of Figures

1. Introduction

Deep learning approaches have shown impressive performance in various domains, but they rely heavily on labelled data, posing challenges in healthcare due to the laborious process of data collection and the need for trained medical staff. Additionally, the complexity of the diagnostic process requires consideration of multiple factors, making individualised and transparent decision-making crucial. This project aims to leverage machine learning models to assist medical professionals in accurate X-ray predictions, reducing the likelihood of missing important details and improving diagnosis speed and accuracy. The literature review will explore an existing application, followed by the implementation of another model for the same purpose (Davletshina *et al.*, 2020).

2. Literature Review

In the work of Davletshina et al. (2020), the authors are experimenting with different unsupervised approaches to achieve the desired outcome.

The main algorithm groups that they are utilising for the purpose are:

- Autoencoders
- Generative Adversarial Networks(Davletshina *et al.*, 2020a)

2.1. MURA Dataset

The dataset that is being used is a subset of the MURA dataset (Rajpurkar *et al.*, 2017), where only X-rays of hands were extracted. It comprises of 5543 images gathered within 2018 studies of 1945 individuals. Each study is labelled as negative or positive. The meaning of the positive label is that during the diagnostic process no abnormalities were identified. The positive results in the dataset are 521, that are backed up by 1484 pictures(Davletshina *et al.*, 2020a).

Since this is a subset with data that is gathered from real-world studies, there is the issue with noise in the images that needs to be addressed. Hence, the authors have proposed the developed by them complex pipeline in order to ensure a good quality training data for the unsupervised models, which are being applied(Davletshina *et al.*, 2020).

2.2. Pre-processing Pipeline

As mentioned above the problem of noise levels in the data is present and would impact the performance of the unsupervised models, while trying to identify abnormalities in the images. Thus, it is important to reduce it and in the same time ensure that the noise would not be considered as an abnormality (Davletshina *et al.*, 2020).

Hence, the authors are proposing their comprehensive pipeline. It is divided into two segments (Figure 1):

- Offline Pre-processing – completed one time. Data is saved on the hard drive to reduce processing times.
- Online Pre-processing – the procedure is being conducted while simultaneously loading data (Davletshina *et al.*, 2020).

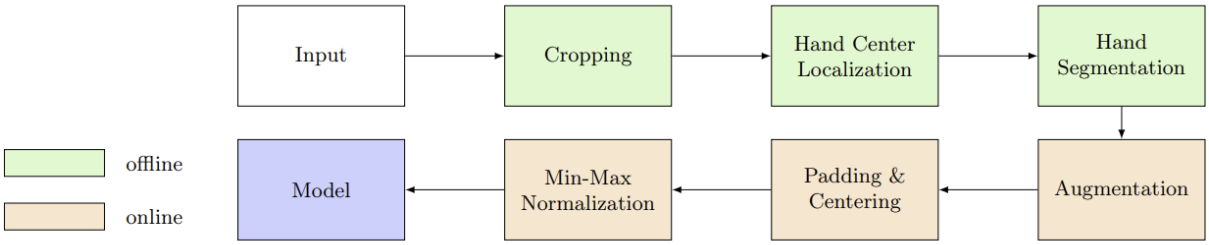

Figure 1 - Pre-processing Pipeline (Davletshina *et al.*, 2020)

As it can be observed in Figure 1, there are several steps that are involved in the pre-processing of the training data:

- Cropping – It is conducted through OpenCV and its ability to detect lineaments in conjunction with Otsu binarization. Its aim is to identify the image carrier and fetch the minimum size of the bounding box, keeping in mind it is not necessary to be axis-aligned. It is important to note that this technique may not work, if the X-ray is very tilted or if the image carrier goes outside of the picture margins.

- Hand Localisation – After the picture is cropped a pre-trained single-shot multibox detector is applied together with MobileNet from TensorFlow.
- Foreground Segmentation – a pixel-wise mask is developed that surrounds the hand in the image through batch processing mode and the “select subject” approach in Photoshop.
- Data Augmentation – Specimen for the BiGAN and α-GAN have their size readjusted to 128 pixels on the longer side of the picture, before the augmentation procedure begins. There is no need for such actions with Auto Encoders. Then the data is processed via the imgaug4 library, where it is flipped, multiplied in channels, rotated and scaled, which is followed by re-sizing the images to 512x512 (AE + DCGAN) or 128x128 (BiGAN + αGAN) pixels(Otsu, 1979; Liu *et al.*, 2016; Davletshina *et al.*, 2020a; Jung, 2020).

2.3. Models Used

The models utilised by the authors are trained on the pre-processed data, which partially includes only pictures of individuals without an abnormality detected. The two main groups of models used, as mentioned earlier, are GANs and AEs(Davletshina *et al.*, 2020a).

2.3.1. Autoencoders

In their work Davletshina et al. (2020) have researched various kinds of AEs that would be suitable for the application. A feature that these approaches commonly possess is the reconstruction loss. To reproduce the picture with good enough quality and excerpt the features needed for that, the model has to compress the data. This is needed, because there is an informational bottleneck through which the input has to go. This is why the network includes an encoder (E). The data (x) is then altered to a latent space representation in a

non-linear manner. On the other hand, a decoder (D) is also applied, which reconstructs the input (z) from the above space and places it into xˆ to the input area(Davletshina *et al.*, 2020). The general loss is depicted via a vector input ($x \in R^n$). The reconstruction loss is provided as the mean over pixel-wise squared differences (x, xˆ = D(E(x)) ∈ R^n).

$$\mathcal{L}(\mathbf{x}, \hat{\mathbf{x}}) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{x}_i)^2 = \frac{1}{n}\|\mathbf{x} - \hat{\mathbf{x}}\|_2^2$$

Figure 2 - Autoencoders Reconstruction Loss (Davletshina *et al.*, 2020)

Since, the scope of this project focuses on identifying abnormalities in the X-rays of hands, the authors have applied a variation of the loss, which is called “masked reconstruction loss”, because here the pixels that are taken into consideration are the ones that reside within the mask(Davletshina *et al.*, 2020).

Hence, if $m \in \{0, 1\}^n$ is the mask, m_i will be equal to “1”, only if the position “i” is associated with the hand. The Hadamard product is also denoted by the “^{O}” symbol (Figure 3).

$$\mathcal{L}_M(\mathbf{x}, \hat{\mathbf{x}}, \mathbf{m}) = \frac{1}{\|\mathbf{m}\|_1}\|\mathbf{m} \odot (\mathbf{x} - \hat{\mathbf{x}})\|_2$$

Figure 3 - Masked Reconstruction Loss (Davletshina *et al.*, 2020)

2.3.1.1. Convolutional Autoencoder

This form of AEs is used as a fully connected CNN with the application of an encoder and decoder. The encoder consists of a series of re-casted convolutional blocks. Batch Normalisation is also utilised between each convolution and its activation, for which “ReLU” is applied(Glorot *et al.*, 2011; Ioffe and Szegedy, 2015; Davletshina *et al.*, 2020).

Thus, the decoder is built of repeated blocks of transposed convolutions. Batch normalisation is again used before every activation. The bottleneck size is a spatial resolution of 16 × 16 and 512 channels(Davletshina *et al.*, 2020).

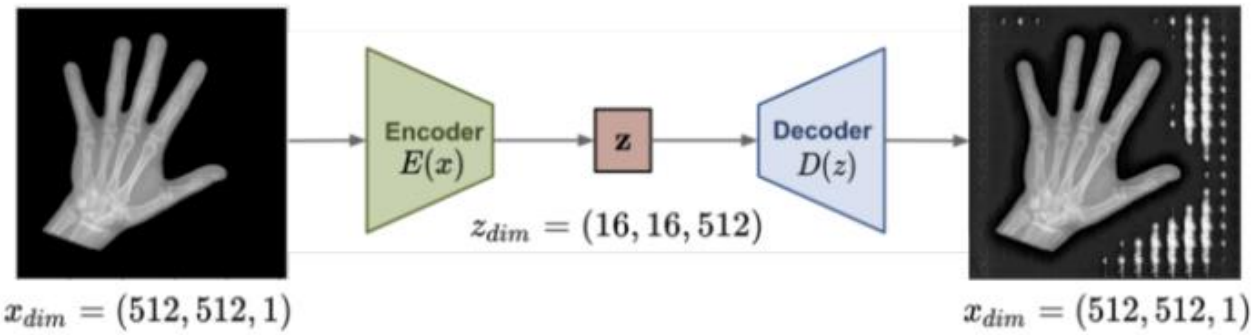

Figure 4 - Convolutional Autoencoder - Schematic Overview (Davletshina *et al.*, 2020)

CAE	
encoder	
kernel size	output filters
(3, 3)	(512, 512, 16)
(4, 4)	(256, 256, 32)
(3, 3)	(256, 256, 32)
(4, 4)	(128, 128, 64)
(3, 3)	(128, 128, 64)
(4, 4)	(64, 64, 128)
(3, 3)	(64, 64, 128)
(4, 4)	(32, 32, 256)
(3, 3)	(32, 32, 256)
(4, 4)	(16, 16, 512)

Figure 5 -Convolutional Autoencoder - Architecture Details(Davletshina *et al.*, 2020)

2.3.1.2. Variational Autoencoder

This is a productive approach, that can map an input to a Gaussian distribution in latent space. It is described by its mean and covariance (μ(x), Σ(x)) and is not mapped to a static latent representation. In most cases a diagonal matrix is applied. For reconstruction, a sample z ~ N (μ(x), Σ(x)) is drawn, and processed via the decoder sub-network. One more loss function is implemented - Kullback-Leibler divergence (KLD). This is needed in order to prevent values in Σ that are insignificant. The KLD is inserted between N (μ(x), Σ(x)) and the standard normal distribution N (0, I)(Davletshina *et al.*, 2020).

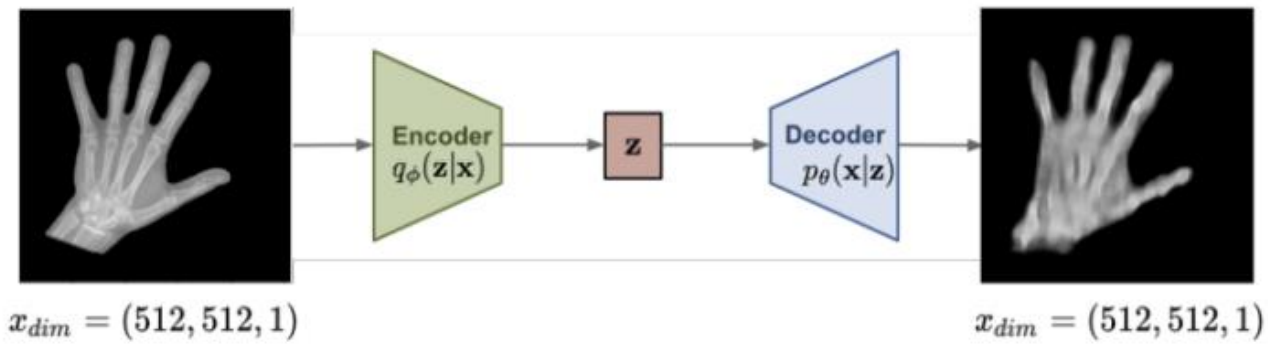

Figure 6 - Variational Autoencoder - Schematic Overview (Davletshina et al., 2020)

VAE	
encoder	
kernel size	output filters
(4, 4)	(255, 255, 8)
(4, 4)	(126, 126, 16)
(4, 4)	(62, 62, 32)
(4, 4)	(30, 30, 64)
(4, 4)	(14, 14, 128)
(4, 4)	(6, 6, 256)
(4, 4)	(2, 2, 512)
bottleneck	
reshape: z	$(2 \cdot 2 \cdot 512)$
$\mu = \mathrm{FC}(z)$	(1024,)
$\sigma = \mathrm{FC}(z)$	(1024,)
$z' = \sigma\epsilon + \mu$	(1024,)
$z'' = FC(z')$	$(2 \cdot 2 \cdot 512)$
reshape:	(2, 2, 512)
decoder	
kernel size	output filters
(4, 4)	(6, 6, 256)
(4, 4)	(14, 14, 128)
(4, 4)	(30, 30, 64)
(4, 4)	(62, 62, 32)
(4, 4)	(126, 126, 16)
(4, 4)	(254, 254, 8)
(6, 6)	(512, 512, 1)

Figure 7 - Variational Autoencoder - Architecture Details (Davletshina et al., 2020)

2.3.2. GAN

This model type can be characterised by the presence of two sub-networks. One of them is annotated as the generator (G) and the other one as the discriminator (D). This is needed, so these networks can act as adversaries to each other. The generator's function is to gather random noise, which works as the input, and produce specimen in the target domain. On the other side, the discriminator receives both real and generated data points and its main task

is to make the difference between the two types of data. The networks are being trained in a cyclical manner(Goodfellow *et al.*, 2020; Davletshina *et al.*, 2020).

2.3.2.1. Deep Convolutional GAN (DCGAN)

DCGAN is a continuation model and is leveraging this type of GAN to a CNNs. This architecture also comprises of a discriminator and generator. The difference with the normal GAN model is that here, we do not have fully connected layers(Radford *et al.*, 2016; Davletshina *et al.*, 2020).

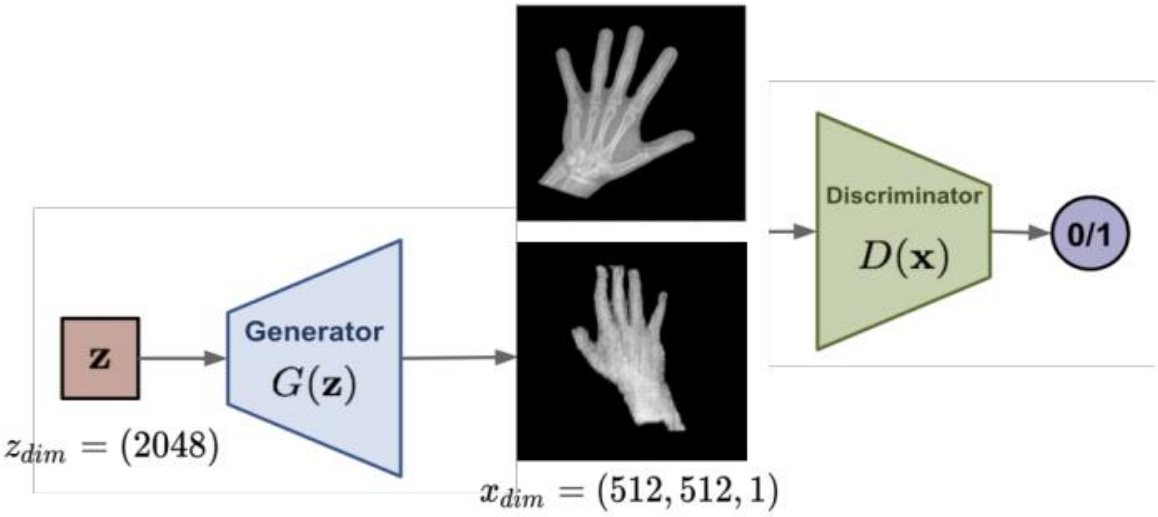

Figure 8 - Schematic Overview of DCGAN (Davletshina et al.*, 2020)*

DCGAN	
generator	
kernel size	output filters
(4, 4)	(4, 4, 1024)
(4, 4)	(8, 8, 512)
(4, 4)	(16, 16, 256)
(4, 4)	(32, 32, 128)
(4, 4)	(64, 64, 64)
(4, 4)	(128, 128, 32)
(4, 4)	(256, 256, 16)
(4, 4)	(512, 512, 1)
discriminator	
kernel size	output filters
(4, 4)	(256, 256, 4)
(4, 4)	(128, 128, 8)
(4, 4)	(64, 64, 16)
(4, 4)	(32, 32, 32)
(4, 4)	(16, 16, 64)
(4, 4)	(8, 8, 128)
(4, 4)	(4, 4, 256)
(4, 4)	(1, 1, 512)
minibatch discrimination	(1, 1, 528)
FC	(1,)

Figure 9 - DCGAN - Architecture Details (Davletshina et al., 2020)

2.3.2.2. BiGAN / ALI

This model is an extension to DCGAN. The difference is that the real picture would be encoded into latent space by the encoder (E). Thus, the discriminator is obtaining the real, fake pictures and the latent codes behind them(Donahue *et al.*, 2017; Dumoulin *et al.*, 2017; Davletshina *et al.*, 2020).

2.3.2.3. α-GAN

There are four sub-networks that this model consists of:

- The real picture is revamped into a latent representation by an encoder E(x).
- The difference between the arbitrary noise, that would be utilised as the input for the generator, and the latent representations deriving from the encoder would be made by a code-discriminator CO(z).
- From either the randomly sampled X-ray (z) or the encoded picture (E(x)), there will be a new one produced by the generator G(z).
- A difference will also be made between reconstructed real-world pictures G(E(x)) and produced ones G(z). This task will be completed by a discriminator D(x)(Rosca *et al.*, 2017; Davletshina *et al.*, 2020).

The encoder-generator pair in the auto-encoder model includes a code-discriminator that imposes a reconstruction loss along with classification losses for the discriminators. This encourages the encoder to transform the inputs in a way that aligns with the random distribution, similar to how the KL-divergence works in the VAE method. Additionally, the discriminator promotes matching the image domain's data distribution(Rosca *et al.*, 2017; Davletshina *et al.*, 2020).

2.4. Anomaly Detection Scores

2.4.1. Anomaly Detection Scores for AE Models

Due to the fact that Autoencoders are trained on a dataset that contains regular data, where abnormalities are not introduced, the models would not be able to reconstruct the images. This is why AEs are being used for identification of abnormalities. This failure is most likely to happen in zones surrounding the anomaly.

There would be two purposes that the reconstruction can be used for:

- As a **pixel-wise heatmap** that would be denser in the areas where the abnormality is observed.
- Image-wise score – aggregating over all pixels under the mask(Davletshina *et al.*, 2020).

2.4.2. Anomaly Detection Scores for GAN Models

Here, the authors have utilised the output coming from the discriminator to produce the scores for the identification of abnormalities. Once assembled, the discriminator is supposed to differentiate between X-rays with and without issues on them(Davletshina *et al.*, 2020).

For αGAN we use the mean over code discriminator and discriminator probability(Davletshina *et al.*, 2020).

2.5. Conducted Experiments

To conduct their experiments the authors saved 3062 pictures in single-channel PNG image and 2481 as three RGB channels. Due to the look of the second batch of images, which resembled grey-scale ones, the authors took a decision to transform all of them to single-channel ones. Most of the pictures had between 350 and 450 pixels as their shorter side, however they were varying between 160 and 512. The longer side was kept to 512 for all of the X-rays(Davletshina *et al.*, 2020).

Since this project focuses on unsupervised models, they were trained with data that consists of pictures without any abnormalities on them. They were divided by patient in order to ensure

that there would not be any X-rays belonging to the same person in the training, validation or test datasets(Davletshina *et al.*, 2020).

Thus, if P is denoting the collection of all people involved in the studies and P+ the one of individuals where an anomaly was observed, the remaining negative studies would be incorporated as P− := P \ P +(Davletshina *et al.*, 2020).

The authors were focusing to achieve a fair spread across the test and validation datasets, hence the positive studies were randomly and equally split. Subsequently, an equal number of patients were selected, where no detected abnormalities were present. This was done, randomly for both validation and testing purposes, while the remaining patients are utilised for training (Figure 10). These actions produced 2554 training samples, 1494 to validate and 1495 testing ones(Davletshina *et al.*, 2020).

The experiments were conducted on a system with one NVIDIA Tesla V100 GPU with 16GiB of VRAM, 20 cores and 360GB of RAM(Davletshina *et al.*, 2020).

All models were trained from zero and no transfer learning was applied. Additionally, a manual search for hyper-parameters was conducted using the validation set. The most effective models of each type were identified based on their Area-under-Curve for the Receiver-Operator-Curve (ROC-AUC). Ultimately, the performance of the selected models was evaluated on the test set, using ROC-AUC as our metric for assessment conducted(Davletshina *et al.*, 2020).

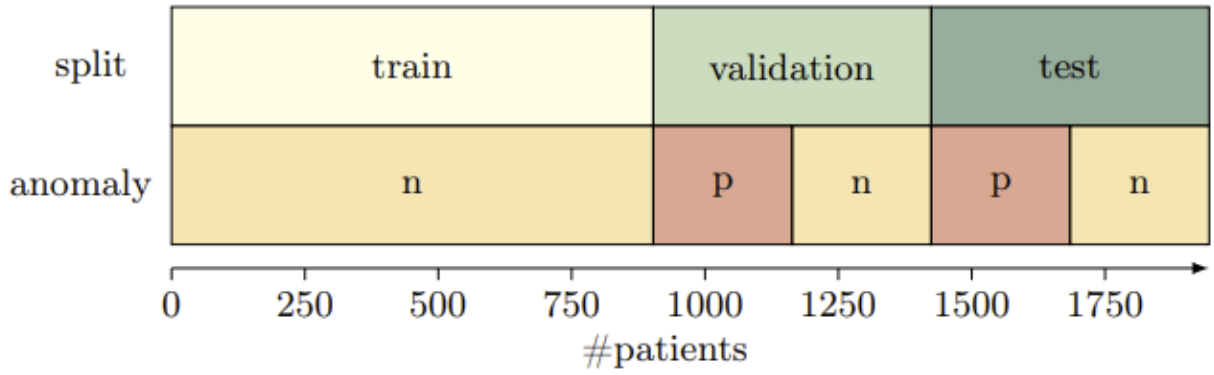

Figure 10 - Data Split (Davletshina et al.*, 2020)*

2.6. Reported Results

2.6.1. Quantitative Results

In addition to evaluating the performance of individual models, the significance of pre-processing techniques was also assessed. A comparison between the results of models trained on raw data, data with cropped hand regions, and fully pre-processed data was delivered. The effect of histogram equalisation before augmentation was also evaluated. The produced findings, presented in Figure 11, demonstrate that fully pre-processed data consistently produces the best results, emphasising the importance of the pre-processing pipeline for noisy datasets. Interestingly, cropping only the hands without foreground segmentation results in poorer performance than using raw data. Furthermore, histogram equalisation consistently improves the results for AE-based models. Experiments on BiGAN and αGAN were not finished by the deadline. For raw and cropped data, ROC-AUC values often fall below 45%, suggesting that flipping the anomaly decision could improve the score. However, this partly attributes to the instability of the results for these models. It was also observed that using only the top-k loss values across all pixels does not improve the result for aggregation of reconstruction error. This may be due to using a small value of k (=200) for all models, which could fail to detect some anomalies. Overall, α-GAN achieved the highest ROC-AUC score of 60.7% using discriminator probability, while CAE also achieved a respectable 57% ROC-AUC and provides pixel-level anomaly scores for better interpretability(Davletshina *et al.*, 2020).

	raw		crop		full	
	w/o HE	w/ HE	w/o HE	w/ HE	w/o HE	w/ HE
CAE						
MSE	.460 ± .033	.504 ± .034	.466 ± .022	.510 ± .021	.501 ± .013	**.570 ± .019**
MSE (top-200)	.466 ± .013	.448 ± .025	.486 ± .015	.473 ± .018	.506 ± .039	.553 ± .023
VAE						
KLD	.488 ± .031	.491 ± .013	.470 ± .046	.496 ± .045	.520 ± .026	**.533 ± .014**
L1	.432 ± .033	.446 ± .016	.438 ± .033	.438 ± .016	.435 ± .014	.483 ± .009
L1 + KLD	.432 ± .033	.446 ± .016	.438 ± .034	.437 ± .016	.438 ± .011	.488 ± .011
L1 (top-200)	.438 ± .017	.472 ± .010	.440 ± .025	.471 ± .013	.428 ± .013	.481 ± .010
MSE	.432 ± .033	.446 ± .016	.438 ± .033	.438 ± .016	.435 ± .014	.483 ± .009
MSE + KLD	.432 ± .033	.446 ± .016	.438 ± .033	.438 ± .016	.436 ± .013	.486 ± .010
MSE (top-200)	.438 ± .017	.472 ± .010	.440 ± .025	.471 ± .013	.428 ± .013	.481 ± .010
DCGAN						
Disc. (D)	.497 ± .018	.491 ± .041	.493 ± .015	.493 ± .025	**.530 ± .027**	.527 ± .022
BiGAN						
MSE	.471 ± .021	-	.438 ± .039	-	.491 ± .042	.522 ± .017
MSE (top-200)	.471 ± .011	-	.459 ± .030	-	.475 ± .033	.508 ± .026
Disc. (D)	.508 ± .007	-	.534 ± .016	-	**.549 ± .006**	.522 ± .019
αGAN						
Code-Disc. (C)	.500 ± .000	-	.500 ± .001	-	.500 ± .000	.500 ± .000
MSE	.476 ± .029	-	.466 ± .022	-	.442 ± .013	.528 ± .018
MSE (top-200)	.465 ± .031	-	.446 ± .018	-	.422 ± .016	.533 ± .013
Disc. (D)	.503 ± .022	-	.534 ± .022	-	**.607 ± .016**	.584 ± .012
C + D	.503 ± .022	-	.534 ± .022	-	**.607 ± .016**	.584 ± .012

Figure 11 - Quantitative Results (Davletshina et al., 2020)

2.5.2. Qualitative Results

Heatmaps were created using all methods with reconstruction loss, including AE and α-GAN, to visualise the pixel-wise losses and highlight regions that were not reconstructed well. Based on their hypothesis, these regions could potentially be anomalous. Figure 12 depicts typical samples produced by CAE. The top picture displays a hand in a study that was labelled as normal. The reconstruction error is spread widely across the hand instead of being concentrated in one area. The maxima occur around joints. The bottom image shows an abnormality, with a clear highlighting at the middle finger. Metal parts in the X-ray at the same location are visible. The highlighted regions correspond largely to the anomalous regions(Davletshina *et al.*, 2020).

Figure 12 - Samples of Heatmaps that were produced by CAE (Davletshina et al.*, 2020).*

2.7. Advantages and Disadvantages of the Approach

Advantages:

- Unsupervised models used - no requirement for labelled data during training.
- The approach can detect anomalous regions in X-ray images without prior knowledge about the specific anomalies.
- A pre-processing pipeline is utilised that can improve the performance of the models on noisy datasets.
- Some of the models include a visualisation technique that highlights the regions of the X-ray images that could not be reconstructed well, which could help medical experts in identifying anomalous regions(Davletshina *et al.*, 2020).

Disadvantages:

- The models require a large number of computational resources and time for training and testing.
- The approach has limitations in detecting anomalies in highly complex or rare cases, where the model may not have seen similar anomalies during training.
- The work relies on the assumption that anomalous regions have higher reconstruction error, which may not hold true in all cases.
- The approach may not work well with low-contrast images or images with low resolution(Davletshina *et al.*, 2020).

3. Methodology

3.1. Dataset

For the purpose of this mini-project, a subset of the MURA v.1.1. dataset was used. It comprises of 6003 X-ray images that are stored into separate folders by patients' number. Since, each individual can have multiple studies, each one and its respective images are stored separately according to the study they belong to. Each study is labelled by medical

professionals as “positive” or “negative” according to the diagnosis that was deducted by them and the presence of a fracture. The subset has been manually trimmed to its current state, so it can hold pictures of hands only. The data is going to be analysed in further detail in the following sections(Rajpurkar *et al.*, 2017).

For the ease of use and accessibility purposes it has been uploaded to the link below:

https://www.kaggle.com/datasets/bogomililiev2308/mura-hand-xray

3.2. Choice of Model

For the purpose of this project, a slightly customised version of the original DenseNet model presented in the MURA paper is going to be applied. The architecture consists of 169 layers, where layers are linked with each other. They way in which they are working is by each one of them being fed in a forward motion, which betters the feature dissemination, reduces the issue of vanishing gradients, promotes the re-utilisation of features and diminishes the count of parameters. The model will also use a weighted binary cross-entropy loss function for its training. It will also use Adam as an optimiser (Rajpurkar *et al.*, 2017; Agrahari, 2018).

3.3. Data Pipeline

A pipeline is also going to be applied, which will include different data augmentation techniques and will be described in further detail in the **Implementation** section(Rajpurkar *et al.*, 2017; Agrahari, 2018).

4. Implementation

4.1. Data Analysis

As mentioned earlier in the **Methodology** section we would be using a subset of MURA that contains only images of hands. The dataset originally is split into train, valid and test sections, which a holding the respective data. Also, there are **csv** files that hold the labels for each study and the paths for each x-ray. All images are labelled with 1 (positive or anomalous) or 0 (negative or without anomalies) (Rajpurkar *et al.*, 2017; Agrahari, 2018).

Elements of the validation and training datasets are:

- The set that is used for training comprises of one study type – hands.
- Subfolders are created for each patient containing their studies. The name is produced from “patient” plus their consecutive identification number (Figure 14).
- These folders may have a single or multiple pictures or studies inside. Each study is named by using the word “study”, its consecutive number and its outcome (Figure 14).
- The train set contains 2018 studies and the validation one 167(Figure 13).
- Pictures in both sets are variable in pixel size, tilt and quality (Figures 15 & 16).
- The training set has 1945 patients and the validation – 159 (Figure 17).
- There are patients with variable number of studies (Figure 18).
- XR_HAND - a maximum of 5 images in 5 studies, 4 views in 60 studies, 2 pictures in 517, 1404 studies with 3 images and 32 studies with 1 x-ray in it (Figure 19).

Checking the DFs

```
[ ]  1  train_df.shape, valid_df.shape
```

```
((2018, 2), (167, 2))
```

Figure 13 - Data Frames Content

```
[ ]  1  train_df.head(3), valid_df.head(3)
```

```
(                                                  Path  Label
 0  MURA-v1.1/train/XR_HAND/patient09734/study1_positive/      1
 1  MURA-v1.1/train/XR_HAND/patient09735/study1_positive/      1
 2  MURA-v1.1/train/XR_HAND/patient07365/study1_positive/      1,
                                                    Path  Label
 0  MURA-v1.1/valid/XR_HAND/patient11497/study1_positive/      1
 1  MURA-v1.1/valid/XR_HAND/patient11498/study1_positive/      1
 2  MURA-v1.1/valid/XR_HAND/patient11499/study1_positive/      1)
```

Figure 14 - Review of the top 3 rows of the train and validation datasets labels and paths.

```
1  from IPython.display import Image
2  Image("/content/drive/My Drive/ML/DenseNet-MURA-PyTorch/MURA-v1.1/train/XR_HAND/patient00218/study1_positive/image1.png")
```

Figure 15 - Plotting of random x-rays for analysis 1

```
Image("/content/drive/My Drive/ML/DenseNet-MURA-PyTorch/MURA-v1.1/train/XR_HAND/patient00565/study1_negative/image2.png")
```

Figure 16 -Plotting of random x-rays for analysis 2

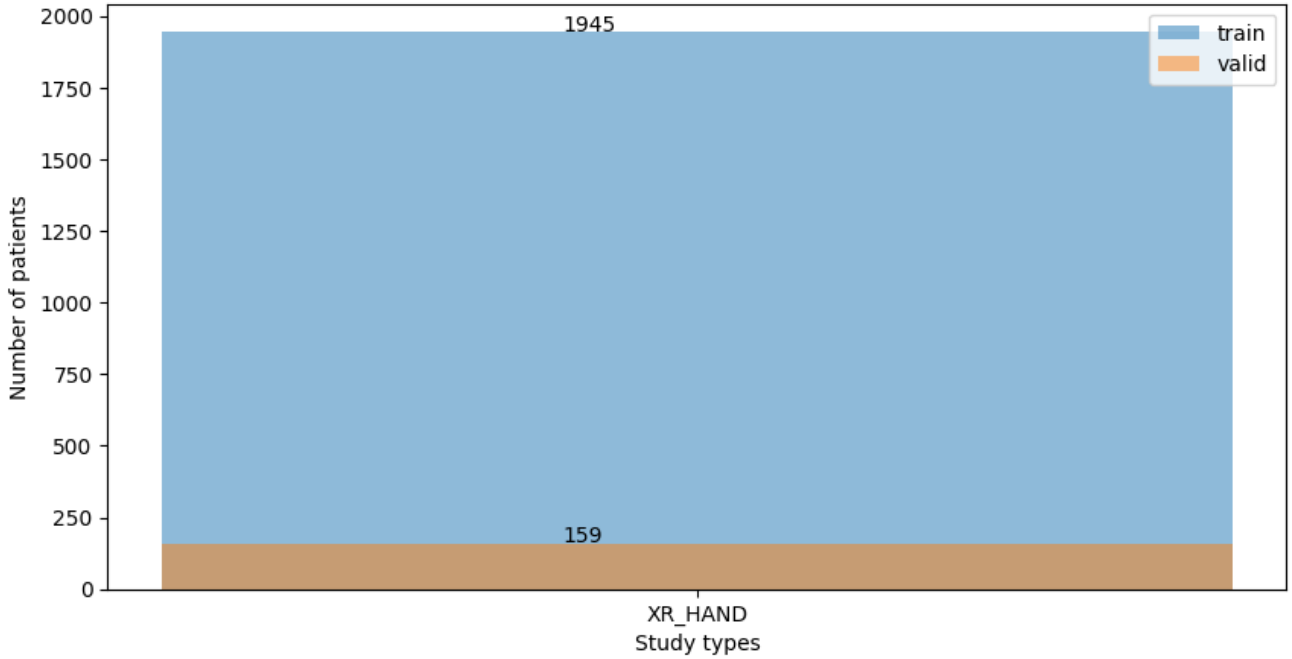

Figure 17 - Count of number of patients per set.

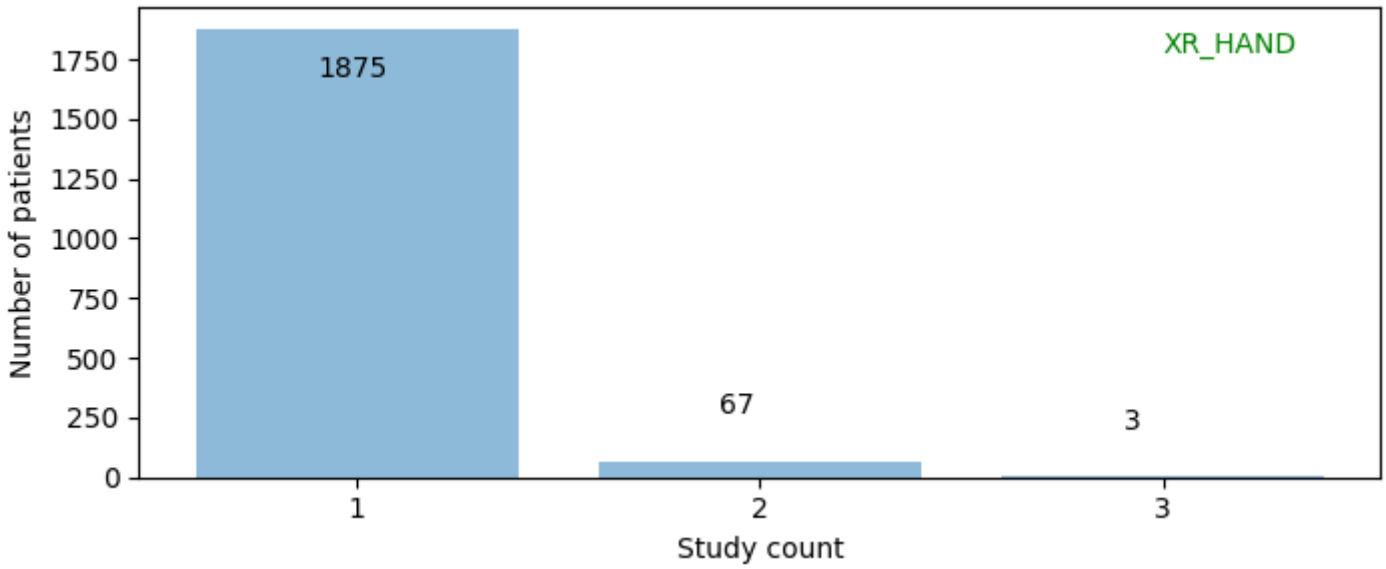

Figure 18 - Count of studies per person.

```
# let's find out number of studies per study_type
view_count = {} # to store study counts for each study type, study count = number of patients which have similar number of studies
for study_type in study_types:
    BASE_DIR = '/content/drive/My Drive/ML/DenseNet-MURA-PyTorch/MURA-v1.1/train/%s/' % study_type
    view_count[study_type] = defaultdict(lambda:0) # to store study count for current study_type, initialized to 0 by default
    patients = list(os.walk(BASE_DIR))[0][1] # patient folder names
    for patient in patients:
        studies = os.listdir(BASE_DIR + patient)
        for study in studies:
            views = os.listdir(BASE_DIR + patient + '/' + study)
            view_count[study_type][len(views)] += 1
```

```
[ ]  view_count
```

```
{'XR_HAND': defaultdict(<function __main__.<lambda>()>,
            {3: 1404, 2: 517, 1: 32, 4: 60, 5: 5})}
```

Figure 19 - Count of images per individual.

4.2. The Pipeline

As the model accepts a single or multiple x-rays as input for a study of the upper limbs, the **DenseNet** generates the predictions for possible anomalies. Then by using the arithmetic mean of the abnormality probabilities output by the network for each image the overall probability is calculated. The model makes the binary prediction of abnormal if the probability of abnormality for the study is greater than 0.5(Rajpurkar *et al.*, 2017; Agrahari, 2018).

Hence, the possibility of anomalies needs to be guessed at study level. Thus, the pipeline has to be at the latter level. It should bring back all x-rays of a study and pass them to the model with the label that belongs to it(Rajpurkar *et al.*, 2017; Agrahari, 2018).

To adhere to the data augmentation needed according to the original MURA paper, **PyTorch** was utilised with its data pipeline, augmentation modules, Dataset and **Dataloader**(Rajpurkar *et al.*, 2017; Agrahari, 2018).

ImageDataSet is organising the data for the training. On every repetition of the pipeline the __**getitem**__ function is summoned. All the x-rays in a study are piled in a tensor and then a dictionary is created, which holds the respective label as well. For the augmentation applied to the pictures a transform module provided by **PyTorch** is being used and the views are scaled to 224x224 pixels. Other techniques utilised are:

- Random horizontal flips
- Picture turning (<10)
- Normalisation to the mean and standard deviation of **ImageNet** dataset (Rajpurkar *et al.*, 2017; Agrahari, 2018).

After these steps are completed, the function **get_dataloaders** provides **dataloaders** as a dictionary. These are used for the datasets utilised for training and validation(Rajpurkar *et al.*, 2017; Agrahari, 2018).

```python
import os
import pandas as pd
from tqdm import tqdm
import torch
from torchvision import transforms
from torch.utils.data import DataLoader, Dataset
from torchvision.datasets.folder import pil_loader

data_cat = ['train', 'valid'] # data categories

def get_study_level_data(study_type):
    """
    Returns a dict, with keys 'train' and 'valid' and respective values as study level dataframes,
    these dataframes contain three columns 'Path', 'Count', 'Label'
    Args:
        study_type (string): one of the seven study type folder names in 'train/valid/test' dataset
    """
    study_data = {}
    study_label = {'positive': 1, 'negative': 0}
    for phase in data_cat:
        BASE_DIR = '/content/drive/My Drive/ML/DenseNet-MURA-PyTorch/MURA-v1.1/%s/%s/' % (phase, study_type)
        patients = list(os.walk(BASE_DIR))[0][1] # list of patient folder names
        study_data[phase] = pd.DataFrame(columns=['Path', 'Count', 'Label'])
        i = 0
        for patient in tqdm(patients): # for each patient folder
            for study in os.listdir(BASE_DIR + patient): # for each study in that patient folder
                label = study_label[study.split('_')[1]] # get label 0 or 1
                path = BASE_DIR + patient + '/' + study + '/' # path to this study
                study_data[phase].loc[i] = [path, len(os.listdir(path)), label] # add new row
                i+=1
    return study_data

```

Figure 20 - Pipeline Code 1(Rajpurkar et al.*, 2017; Agrahari, 2018)*

```
class ImageDataset(Dataset):
    """training dataset."""

    def __init__(self, df, transform=None):
        """
        Args:
            df (pd.DataFrame): a pandas DataFrame with image path and labels.
            transform (callable, optional): Optional transform to be applied
                on a sample.
        """
        self.df = df
        self.transform = transform

    def __len__(self):
        return len(self.df)

    def __getitem__(self, idx):
        study_path = self.df.iloc[idx, 0]
        count = self.df.iloc[idx, 1]
        images = []
        for i in range(count):
            image = pil_loader(study_path + 'image%s.png' % (i+1))
            images.append(self.transform(image))
        images = torch.stack(images)
        label = self.df.iloc[idx, 2]
        sample = {'images': images, 'label': label}
        return sample

def get_dataloaders(data, batch_size=8, study_level=False):
    '''
    Returns dataloader pipeline with data augmentation
    '''
    data_transforms = {
        'train': transforms.Compose([
                transforms.Resize((224, 224)),
                transforms.RandomHorizontalFlip(),
                transforms.RandomRotation(10),
                transforms.ToTensor(),
                transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]),
        'valid': transforms.Compose([
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]),
    }
    image_datasets = {x: ImageDataset(data[x], transform=data_transforms[x]) for x in data_cat}
    dataloaders = {x: DataLoader(image_datasets[x], batch_size=batch_size, shuffle=True, num_workers=4) for x in data_cat}
    return dataloaders

if __name__=='main':
    pass
```

Figure 21 - Pipeline Code 2 (Rajpurkar et al.*, 2017; Agrahari, 2018)*

4.3. The DenseNet Model

It is a 169-layer CNN. DenseNet calculates for every view in a study the possibility of anomalies. By establishing connections between every layer in a forward direction, the architecture enables the effective optimisation of deep networks, ensuring their manageability and feasibility. The last fully connected layer was exchanged for a single output one. Then sigmoid nonlinearity was put in place. The network's weight parameters were initially assigned values derived from a model that underwent pretraining on the ImageNet dataset (Deng *et al.*, 2010; Huang *et al.*, 2017; Rajpurkar *et al.*, 2017; Agrahari, 2018).

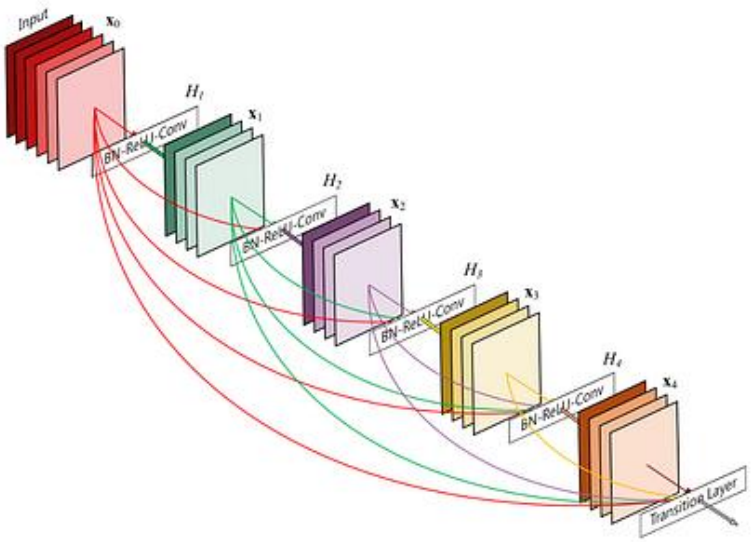

Figure 22 - DenseNet Architecture (Rajpurkar et al.*, 2017; Agrahari, 2018)*

It is important to mention that PyTorch comes as a library with a pre-trained application of DenseNet. However, in order to achieve the latter customisation points, the base model needs to be tweaked (Figures 23, 24 & 25).

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.model_zoo as model_zoo
from collections import OrderedDict

__all__ = ['DenseNet', 'densenet169']


model_urls = {
    'densenet169': 'https://download.pytorch.org/models/densenet169-b2777c0a.pth',
}

def densenet169(pretrained=False, **kwargs):
    r"""Densenet-169 model from
    `"Densely Connected Convolutional Networks" <https://arxiv.org/pdf/1608.06993.pdf>`_

    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
    """
    model = DenseNet(num_init_features=64, growth_rate=32, block_config=(6, 12, 32, 32),
                     **kwargs)
    if pretrained:
        model.load_state_dict(model_zoo.load_url(model_urls['densenet169']), strict=False)
    return model

class _DenseLayer(nn.Sequential):
    def __init__(self, num_input_features, growth_rate, bn_size, drop_rate):
        super(_DenseLayer, self).__init__()
        self.add_module('norm1', nn.BatchNorm2d(num_input_features)),
        self.add_module('relu1', nn.ReLU(inplace=True)),
        self.add_module('conv1', nn.Conv2d(num_input_features, bn_size *
                        growth_rate, kernel_size=1, stride=1, bias=False)),
        self.add_module('norm2', nn.BatchNorm2d(bn_size * growth_rate)),
        self.add_module('relu2', nn.ReLU(inplace=True)),
        self.add_module('conv2', nn.Conv2d(bn_size * growth_rate, growth_rate,
                        kernel_size=3, stride=1, padding=1, bias=False)),
        self.drop_rate = drop_rate

    def forward(self, x):
        new_features = super(_DenseLayer, self).forward(x)
        if self.drop_rate > 0:
            new_features = F.dropout(new_features, p=self.drop_rate, training=self.training)
        return torch.cat([x, new_features], 1)


class _DenseBlock(nn.Sequential):
    def __init__(self, num_layers, num_input_features, bn_size, growth_rate, drop_rate):
        super(_DenseBlock, self).__init__()
        for i in range(num_layers):
            layer = _DenseLayer(num_input_features + i * growth_rate, growth_rate, bn_size, drop_rate)
            self.add_module('denselayer%d' % (i + 1), layer)

```

Figure 23 - DenseNet.py Code 1 (Rajpurkar et al.*, 2017; Agrahari, 2018)*

```
class _Transition(nn.Sequential):
    def __init__(self, num_input_features, num_output_features):
        super(_Transition, self).__init__()
        self.add_module('norm', nn.BatchNorm2d(num_input_features))
        self.add_module('relu', nn.ReLU(inplace=True))
        self.add_module('conv', nn.Conv2d(num_input_features, num_output_features,
                                          kernel_size=1, stride=1, bias=False))
        self.add_module('pool', nn.AvgPool2d(kernel_size=2, stride=2))


class DenseNet(nn.Module):
    r"""Densenet-BC model class, based on
    `"Densely Connected Convolutional Networks" <https://arxiv.org/pdf/1608.06993.pdf>`_

    Args:
        growth_rate (int) - how many filters to add each layer (`k` in paper)
        block_config (list of 4 ints) - how many layers in each pooling block
        num_init_features (int) - the number of filters to learn in the first convolution layer
        bn_size (int) - multiplicative factor for number of bottle neck layers
          (i.e. bn_size * k features in the bottleneck layer)
        drop_rate (float) - dropout rate after each dense layer
        num_classes (int) - number of classification classes
    """
    def __init__(self, growth_rate=32, block_config=(6, 12, 24, 16),
                 num_init_features=64, bn_size=4, drop_rate=0, num_classes=1000):

        super(DenseNet, self).__init__()

        # First convolution
        self.features = nn.Sequential(OrderedDict([
            ('conv0', nn.Conv2d(3, num_init_features, kernel_size=7, stride=2, padding=3, bias=False)),
            ('norm0', nn.BatchNorm2d(num_init_features)),
            ('relu0', nn.ReLU(inplace=True)),
            ('pool0', nn.MaxPool2d(kernel_size=3, stride=2, padding=1)),
        ]))

        # Each denseblock
        num_features = num_init_features
        for i, num_layers in enumerate(block_config):
            block = _DenseBlock(num_layers=num_layers, num_input_features=num_features,
                                bn_size=bn_size, growth_rate=growth_rate, drop_rate=drop_rate)
            self.features.add_module('denseblock%d' % (i + 1), block)
            num_features = num_features + num_layers * growth_rate
            if i != len(block_config) - 1:
                trans = _Transition(num_input_features=num_features, num_output_features=num_features // 2)
                self.features.add_module('transition%d' % (i + 1), trans)
                num_features = num_features // 2
```

Figure 24 - DenseNet.py Code 2 (Rajpurkar et al., 2017; Agrahari, 2018)

```
        # Final batch norm
        self.features.add_module('norm5', nn.BatchNorm2d(num_features))

        # Linear layer
        # self.classifier = nn.Linear(num_features, 1000)
        # self.fc = nn.Linear(1000, 1)

        self.fc = nn.Linear(num_features, 1)

        # Official init from torch repo.
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal(m.weight.data)
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()
            elif isinstance(m, nn.Linear):
                m.bias.data.zero_()

    def forward(self, x):
        features = self.features(x)
        out = F.relu(features, inplace=True)
        out = F.avg_pool2d(out, kernel_size=7, stride=1).view(features.size(0), -1)
        # out = F.relu(self.classifier(out))
        out = F.sigmoid(self.fc(out))
        return out
```

Figure 25 - DenseNet.py Code 3 (Rajpurkar et al., 2017; Agrahari, 2018)

The Loss Function

The binary cross entropy loss that has weights was enhanced for every x-ray **X** that originates from the kind of study **T** in the training dataset. Here, **y** serves as the study's label, the allocation of probability that the model is doing to the label **i** is annotated by **p(Y = i|X).** The amount of anomalous views and non-problematic ones are denoted with **|AT |** & **|NT|** in **i, wT ,1 = |NT |/(|AT | + |NT |)** and **wT ,0 = |AT |/(|AT | +|NT |)** (Rajpurkar et al., 2017; Agrahari, 2018)**.**

Following these requirements an adapted class was produced to better suit the algorithm's purpose (Figure 26 & 27).

$$L(X, y) = - wT,1 \cdot y \log p(Y = 1|X) - wT,0 \cdot (1 - y) \log p(Y = 0|X),$$

Figure 26 - The Loss Function (Rajpurkar et al., 2017; Agrahari, 2018)

```
# tai = total abnormal images, tni = total normal images
# study_data[x] is a study level dataframe
tai = {x: get_count(study_data[x], 'positive') for x in data_cat}
tni = {x: get_count(study_data[x], 'negative') for x in data_cat}
Wt1 = {x: n_p(tni[x] / (tni[x] + tai[x])) for x in data_cat}
Wt0 = {x: n_p(tai[x] / (tni[x] + tai[x])) for x in data_cat}

class Loss(torch.nn.modules.Module):
    def __init__(self, Wt1, Wt0):
        super(Loss, self).__init__()
        self.Wt1 = Wt1
        self.Wt0 = Wt0

    def forward(self, inputs, targets, phase):
        loss = - (self.Wt1[phase] * targets * inputs.log() + self.Wt0[phase] * (1 - targets) * (1 - inputs).log())
        return loss
```

Figure 27 - Code for the Loss Function (Rajpurkar et al., 2017; Agrahari, 2018)

4.4. Model Training

To start the model's training and validation phase the code to initialise the process has been implemented and run from the Google Colab notebook called **DataAnalysisTraining.ipynb** in the project folder**.** The process is not very complicated. The data loader is being summoned, which produces a dictionary that stores all x-rays belonging to a study and its label. Then the views are being passed one by one like vectors. Afterwards, the predictions for anomalies are being generated for every picture individually. The mean of all predictions is getting calculated alongside the loss that is computed. The model is then getting refined and the process is repeated (Rajpurkar et al., 2017; Agrahari, 2018).

An Adam optimiser was utilised during the training phase with parameters by default of β1 = 0.9 and β2 = 0.999. The batch size was 8 and the starting learning rate is assigned to 0.0001. It is later diminished by 10, after each epoch passes and the loss in the validation flattens. Afterwards, the model that produces the smallest validation loss is selected (Kingma and Ba, 2015; Rajpurkar *et al.*, 2017; Agrahari, 2018).

The function used for training is located in the train.py python script in the project's folder (Figures 28, 29 & 30).

```python
import time
import copy
import torch
from torchnet import meter
from torch.autograd import Variable
from utils import plot_training

data_cat = ['train', 'valid'] # data categories

def train_model(model, criterion, optimizer, dataloaders, scheduler,
                dataset_sizes, num_epochs):
    since = time.time()
    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0
    costs = {x:[] for x in data_cat} # for storing costs per epoch
    accs = {x:[] for x in data_cat} # for storing accuracies per epoch
    print('Train batches:', len(dataloaders['train']))
    print('Valid batches:', len(dataloaders['valid']), '\n')
    for epoch in range(num_epochs):
        confusion_matrix = {x: meter.ConfusionMeter(2, normalized=True)
                            for x in data_cat}
        print('Epoch {}/{}'.format(epoch+1, num_epochs))
        print('-' * 10)
        # Each epoch has a training and validation phase
        for phase in data_cat:
            model.train(phase=='train')
            running_loss = 0.0
            running_corrects = 0
            # Iterate over data.
            for i, data in enumerate(dataloaders[phase]):
                # get the inputs
                print(i, end='\r')
                inputs = data['images'][0]
                labels = data['label'].type(torch.FloatTensor)
                # wrap them in Variable
                inputs = Variable(inputs.cuda())
                labels = Variable(labels.cuda())
                # zero the parameter gradients
                optimizer.zero_grad()
                # forward
                outputs = model(inputs)
                outputs = torch.mean(outputs)
                loss = criterion(outputs, labels, phase)
                running_loss += loss.data[0]
                # backward + optimize only if in training phase
                if phase == 'train':
                    loss.backward()
```

Figure 28 -Code for Training Function 1 (Kingma and Ba, 2015; Rajpurkar et al.*, 2017; Agrahari, 2018)*

```
                    optimizer.step()
                # statistics

                preds = (outputs.data > 0.5).type(torch.cuda.FloatTensor)
                preds = preds.view(1)
                running_corrects += torch.sum(preds == labels.data)
                confusion_matrix[phase].add(preds, labels.data)
            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects / dataset_sizes[phase]
            costs[phase].append(epoch_loss)
            accs[phase].append(epoch_acc)
            print('{} Loss: {:.4f} Acc: {:.4f}'.format(
                phase, epoch_loss, epoch_acc))
            print('Confusion Meter:\n', confusion_matrix[phase].value())
            # deep copy the model
            if phase == 'valid':
                scheduler.step(epoch_loss)
                if epoch_acc > best_acc:
                    best_acc = epoch_acc
                    best_model_wts = copy.deepcopy(model.state_dict())
        time_elapsed = time.time() - since
        print('Time elapsed: {:.0f}m {:.0f}s'.format(
                time_elapsed // 60, time_elapsed % 60))
        print()
    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    print('Best valid Acc: {:4f}'.format(best_acc))
    plot_training(costs, accs)
    # load best model weights
    model.load_state_dict(best_model_wts)
    return model


def get_metrics(model, criterion, dataloaders, dataset_sizes, phase='valid'):
    '''
    Loops over phase (train or valid) set to determine acc, loss and
    confusion meter of the model.
    '''
    confusion_matrix = meter.ConfusionMeter(2, normalized=True)
    running_loss = 0.0
    running_corrects = 0
    for i, data in enumerate(dataloaders[phase]):
        print(i, end='\r')
        labels = data['label'].type(torch.FloatTensor)
        inputs = data['images'][0]
```

Figure 29 - Code for Training Function 2 (Kingma and Ba, 2015; Rajpurkar et al., 2017; Agrahari, 2018)

```
            # wrap them in Variable
            inputs = Variable(inputs.cuda())
            labels = Variable(labels.cuda())
            # forward
            outputs = model(inputs)
            outputs = torch.mean(outputs)
            loss = criterion(outputs, labels, phase)
            # statistics
            running_loss += loss.data[0] * inputs.size(0)
            preds = (outputs.data > 0.5).type(torch.cuda.FloatTensor)
            preds = preds.view(1)
            running_corrects += torch.sum(preds == labels.data)
            confusion_matrix.add(preds, labels.data)

        loss = running_loss / dataset_sizes[phase]
        acc = running_corrects / dataset_sizes[phase]
        print('{} Loss: {:.4f} Acc: {:.4f}'.format(phase, loss, acc))
        print('Confusion Meter:\n', confusion_matrix.value())
```

Figure 30 - Code for Training Function 3 (Kingma and Ba, 2015; Rajpurkar et al., 2017; Agrahari, 2018)

The model was trained on Nvidia T4 with 16 GB GDDR6 in Google Colab. The 5 epochs that were completed were done in approximately 30 minutes. The trained model was saved in the pretrained folder (Rajpurkar et al., 2017; Agrahari, 2018).

5. Results

To evaluate the model a confusion matrix was generated after each epoch and then a “Loss & Cost” graphs for the training and validation of the best performing one through the epochs are being generated and saved in the EDA folder (Figures 31 & 32).

The performance of the model can be analysed based on various metrics, including true negatives, false positives, false negatives, true positives, training loss, training accuracy, validation loss, and validation accuracy.

From the training, it can be seen that the model scored a gradually improving performance as the number of epochs increased. The true negatives improved from 0.5498 in the first epoch to a best result of 0.9406 in the fifth epoch. On the other hand, the false positives showed a decreasing trend, starting from 0.4502 and reaching a minimum of 0.0594. The false negatives got reduced from 0.6008 to 0.5547, while the true positives were elevated

from 0.3992 to 0.4453. This suggests that the model progressively improved its ability to accurately classify instances within the training set.

Similarly, in the validation set the results are good, highlighting the generalisation capability of the model. The true negatives improved from 0.4653 to 0.9406, while the false positives went down from 0.5347 to 0.0594. The false negatives decreased from 0.3485 to 0.2121, and the true positives went up from 0.6515 to 0.7879. These results in the validation set demonstrate that the model can effectively classify unseen data.

Looking at the metrics of training and validation loss, it is visible that both reduced with consistency during the training process. The training loss went down from 0.2753 in the first epoch to a minimum of 0.2675 in the fifth epoch. Respectively, the validation loss was reduced from 0.4052 to 0.6828, indicating that the model learned to minimise the errors.

Regarding training and validation accuracy, the model achieved a progress in both metrics over the epochs. The training accuracy started at 0.5109 and reached 0.5743, while the validation accuracy was progressively improving from 0.5389 to 0.5749. These results point out that the model learned the underlying patterns in the data and became more accurate in its predictions.

Overall, the custom DenseNet model showed promising performance in both the training and validation sets. It successfully learned to classify data, where the accuracy and diminishing loss through the epochs were continuously getting better. The results indicate that the model has the potential to effectively generalise and make accurate predictions on unseen data.

Number of Epoch	True Negative (Training)	False Positive (Training)	False Negative (Training)	True Positive (Training)	True Negative (Validation)	False Positive (Validation)	False Negative (Validation)	True Positive (Validation)	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
1	0.5497662	0.4502338	0.60076773	0.39923224	0.46534654	0.5346535	0.34848484	0.6515151	0.2753	0.5109	0.4052	0.5389
2	0.6359385	0.36406144	0.6257198	0.37428024	0.34653464	0.65346533	0.21212122	0.7878788	0.2712	0.5684	0.6281	0.521
3	0.58650637	0.41349366	0.54702497	0.45297503	0.9207921	0.07920792	0.8787879	0.12121212	0.2719	0.552	0.588	0.6048
4	0.5390782	0.46092185	0.47408828	0.5259117	0.9405941	0.05940594	0.8939394	0.10606061	0.2678	0.5357	0.6831	0.6108
5	0.6192385	0.38076153	0.5547025	0.4452975	0.9405941	0.05940594	0.9848485	0.01515152	0.2675	0.5743	0.6828	0.5749
Best Results	**0.9405941**	**0.05940594**	**0.8939394**	**0.10606061**	-	-	-	-	-	-	**1.7742**	**0.610778**

Figure 31 - Results after Each Epoch

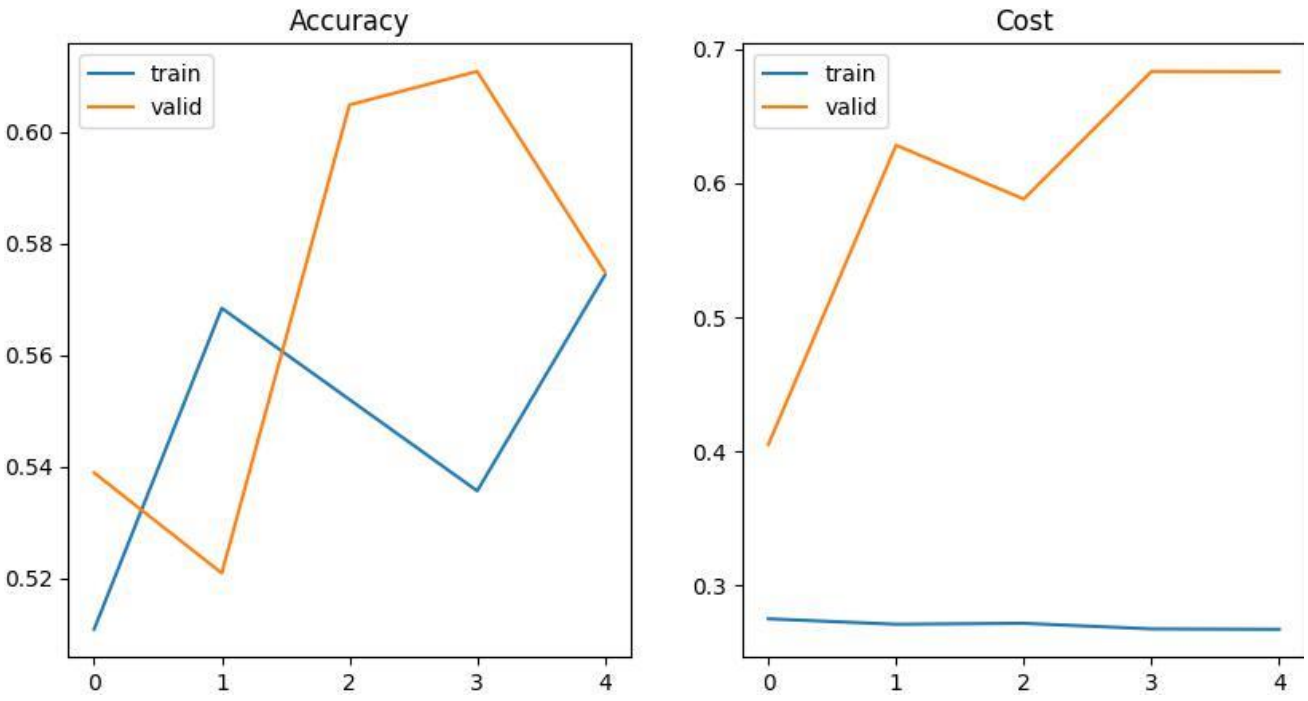

Figure 32 - Cost and Loss graphs of the best performing model.

6. Conclusion

As we observed through the above work different machine learning models are able to achieve high results on predicting fractures of hand from X-ray images. However, human accuracy is still a bit higher - 87.3% (Shelmerdine *et al.*, 2022). Further improvements that can be suggested to the DenseNet model are:

- More data augmentation techniques – rotation, zooming, etc.
- Different optimisers, batch sizes and learning rates can be applied.
- Addition of more and different training data.

7. Definitions and Abbreviations

Autoencoders (AE)

Convolutional Autoencoder (CAE)

Convolutional Neural Network (CNN)

Deep Convolutional GAN (DCGAN)

Dense Convolutional Neural Network (DenseNet)

Generative Adversarial Networks (GANs)

Reconstruction Loss – The evaluation of the reconstruction of the input, which is also the target.

Variational Autoencoder (VAE)

8. Acknowledgement

The author of this work would like to thank the Mura Team and Stanford University for providing their work and supplying the dataset used in it in an open-source form.

Also, an acknowledgement is needed for Rishabh Agrahari for their consecutive implementation of the DenseNet model that was used in this project as a re-implementation base.

9. Word Count

3979

10. Bibliography

Agrahari, R. (2018) *Implementing DenseNet on MURA using PyTorch | by Rishabh Agrahari | Medium*. Available at: https://pyaf.medium.com/implementing-densenet-on-mura-using-pytorch-f39e92566815 [Accessed: 17 May 2023].

Davletshina, D. et al. (2020a) *Unsupervised Anomaly Detection for X-Ray Images*. [Accessed: 3 May 2023].

Davletshina, D. et al. (2020b) *Unsupervised Anomaly Detection for X-Ray Images*. [Accessed: 7 May 2023].

Deng, J. et al. (2010) *ImageNet: A large-scale hierarchical image database*. In: Institute of Electrical and Electronics Engineers (IEEE), Mar 1, 2010. Institute of Electrical and Electronics Engineers (IEEE). Available at: doi:10.1109/cvpr.2009.5206848

Donahue, J. et al. (2017) *Adversarial feature learning*. In: International Conference on

Learning Representations, ICLR, May 31, 2017. International Conference on Learning Representations, ICLR. Available at: https://arxiv.org/abs/1605.09782v7 [Accessed: 9 May 2023].

Dumoulin, V. et al. (2017) *Adversarially learned inference*. In: International Conference on Learning Representations, ICLR, Jun 2, 2017. International Conference on Learning Representations, ICLR. Available at: https://arxiv.org/abs/1606.00704v3 [Accessed: 9 May 2023].

Glorot, X. et al. (2011) *Deep sparse rectifier neural networks*. In: JMLR Workshop and Conference Proceedings, Jun 14, 2011. JMLR Workshop and Conference Proceedings. Available at: https://proceedings.mlr.press/v15/glorot11a.html [Accessed: 8 May 2023].

Goodfellow, I. et al. (2020) Generative adversarial networks. *Communications of the ACM*, 63(11), 139–144. Association for Computing Machinery. [Accessed: 9 May 2023].

Huang, G. et al. (2017) *Densely connected convolutional networks*. In: Institute of Electrical and Electronics Engineers Inc., Aug 25, 2017. Institute of Electrical and Electronics Engineers Inc. Available at: doi:10.1109/CVPR.2017.243 [Accessed: 17 May 2023].

Ioffe, S. & Szegedy, C. (2015) *Batch normalization: Accelerating deep network training by reducing internal covariate shift*. In: International Machine Learning Society (IMLS), Feb 11, 2015. International Machine Learning Society (IMLS). Available at: https://arxiv.org/abs/1502.03167v3 [Accessed: 8 May 2023].

Jung, A. (2020) *aleju/imgaug: Image augmentation for machine learning experiments.* Available at: https://github.com/aleju/imgaug [Accessed: 8 May 2023].

Kingma, D.P. & Ba, J.L. (2015) *Adam: A method for stochastic optimization*. In: International Conference on Learning Representations, ICLR, Dec 22, 2015. International Conference on Learning Representations, ICLR. Available at: https://arxiv.org/abs/1412.6980v9 [Accessed: 18 May 2023].

Liu, W. et al. (2016) *SSD: Single shot multibox detector*. In: Springer Verlag, Dec 7, 2016.

Springer Verlag. Available at: doi:10.1007/978-3-319-46448-0_2 [Accessed: 8 May 2023].

Otsu, N. (1979) THRESHOLD SELECTION METHOD FROM GRAY-LEVEL HISTOGRAMS. *IEEE Trans Syst Man Cybern*, SMC-9(1), 62–66.

Radford, A. et al. (2016) *Unsupervised representation learning with deep convolutional generative adversarial networks*. In: International Conference on Learning Representations, ICLR, Nov 19, 2016. International Conference on Learning Representations, ICLR. Available at: https://arxiv.org/abs/1511.06434v2 [Accessed: 9 May 2023].

Rajpurkar, P. et al. (2017) *MURA: Large Dataset for Abnormality Detection in Musculoskeletal Radiographs*. [Accessed: 8 May 2023].

Rosca, M. et al. (2017) *Variational Approaches for Auto-Encoding Generative Adversarial Networks*. [Accessed: 9 May 2023].

Shelmerdine, S.C. et al. (2022) Can artificial intelligence pass the Fellowship of the Royal College of Radiologists examination? Multi-reader diagnostic accuracy study. BMJ, 379. British Medical Journal Publishing Group. [Accessed: 18 May 2023].