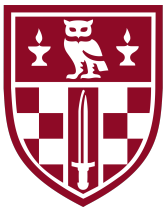
Birkbeck, University of London

Pneumonia Detection from Chest X-Ray Images

Author:
Baran Buluttekin

Supervisor:
Dr. George Magoulas



A project report submitted in fulfillment of the requirements for the degree of MSc Data Science

in the

Department of Computer Science

September 15, 2020

Declaration

I have read and understood the sections of plagiarism in the College Policy on assessment offences and confirm that the work is my own, with the work of others clearly acknowledged. I give my permission to submit my report to the plagiarism testing database that the College is using and test it using plagiarism detection software, search engines or meta- searching software.

The report may be freely copied and distributed provided the source is explicitly acknowledged.

Acknowledgement

Abstract

Placeholder, to be completed later.

Contents

1 Introduction

Medical diagnosis and specifically computer-aided diagnosis (CAD) is a hot topic in the field of technology. One of the main reasons for becoming a hot topic is the recent innovation and breakthroughs achieved by computer vision research. Combined with poor healthcare coverage around the globe, CAD systems offer a promising solution to mitigate the devastating impact of fatal diseases such as pneumonia. Achieving human-level accuracy in computer vision task in a wide array of classification task such as ImageNet large scale visual recognition challenge (ILSVRC) [4] sparked the debates about whether these CAD systems can reduce or altogether replace the jobs such as radiologist in the future. Controversial topics such as whether or not artificial intelligence will replace the radiologist in the future aside, these automated systems can offer answers for patient's questions in absence of medical help or to very least offer much needed second opinion in the face of unsatisfied diagnoses. Given all the mentioned possible benefits of the CAD systems, this project is focused on building classification CAD systems for diagnosing pneumonia from the chest X-ray images.

1.1 Aims and Objectives

The aim of this project is to build a fully functional chest X-ray image classification pipeline that implements CI/CD principals to experimentation and deployment. These pipelines also referred as MLOps where the part of the machine learning workflow is automated.

1.1.1 Objectives

Project will be implemented with execution of fallowing objectives:

1. **Data pre-processing and data exploration:** Preparing the data for model ready state and general data exploration.

2. **Building baseline model with well known neural network architectures:** This step involves setting additional benchmarks with out of the box models from section 4.

3. **Increasing model performance:** Using ensemble method and techniques such as transfer learning to increase model performance.

4. **Ensuring model interpretability with visualization:** For making sure model learning as intended and focusing on correct parts of the image.

5. **Applying different deployment options:** Implementation of model development. Based on best choice for project specification.

It's worth emphasizing that the objective of this project is not to achieve the state of the art result in pneumonia detection but to offers a preferred method for improving and enhancing the existing models. The intuition behind choosing the above objectives instead of attempting to build novel architecture from scratch is the process of choosing a novel architecture has a very large search space and requires a lot of iteration and experimentation. Due to the limited time frame of this project attempting to find new architecture would not be feasible. Additionally, objectives designed to serve the project goal with consistent aims. For example, item 1 and 2 will focus on reducing the model over-fitting while item 5 would serve as a tool to detect over-fitting. Objective 2 serves as a selection for a suitable model and setting benchmark while 6 is aimed at improving the model.

1.2 CI/CD Pipeline

In this section, I will give a brief introduction to the CI/CD pipeline to explain what CI/CD is and why it is chosen as a preferred way to build this project.

Continuous integration (CI) is a workflow strategy that helps ensure everyone's changes will integrate with the current version of the project in the typical software engineering team. This lets members of the team catch bugs, reduce merge conflicts, and increase overall confidence that your software is working. While the details may vary depending on the development environment, most CI systems feature the same basic tools and processes. In most scenarios, a team will practice CI in conjunction with automated testing using a dedicated server or CI service. Whenever a developer adds new work to a branch, the server will automatically build and test the code to determine whether it works and can be integrated with the code on the main development branch. The CI server will produce output containing the results of the build and an indication of whether or not the branch passes all the requirements for integration into the main development branch. By exposing build and test information for every commit on every branch, CI paves the way for what's known as continuous delivery, or CD, as well as a related process, called continuous deployment. The difference between continuous delivery and continuous deployment is that CD is the practice of developing software in such a way that you could release it at any time. When coupled with CI, continuous delivery lets you develop features with modular code in more manageable increments. Continuous development is an extension of continuous delivery. It's a process that allows you to actually deploy newly developed features into production with confidence, and experience little if any, downtime. Even though the benefits of using CI/CD pipelines are more prominent in the software teams, integration automated testing will help even individual projects such as this by reducing time for debugging.

In more granular detail, this system works with central version control services and in this project central version control service used is Github. GitHub uses a communication tool called *webhooks* to send messages to external systems about activities and events that occurred in the project. For each event type, subscribers will receive messages related to the event. Generally, events refer to action involving the software such as new commit push, pull (merge) request, or other software related action. In this case, whenever a new commit is pushed to any branch of the project, a message from Github will be sent out to a third party system called *travis.* [1] Travis is a hosted CI service that allows build and test software hosted in version control services. When travis receives the webhook call it will fetch the most recent version of the project and run the tests associated with it. When the test runs completed with the latest version of the software, test results will be sent back to relevant commits as status information using GitHub API. This information can either be used by developers for making decisions such as whether to accept the pull request versus reject it or if applicable can be used by service to initiate the deployment process for the software. In all cases, CI/CD will work as automation for software quality assurance process to speed up the development and improve the overall reliability of the software.

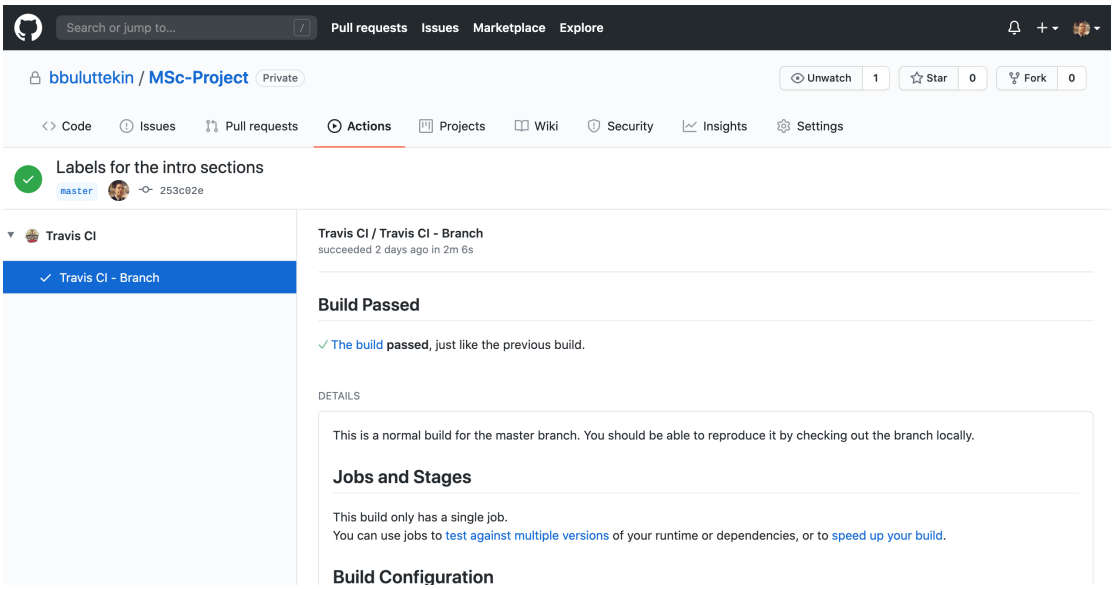


Figure 1.1: CI feedback received from Travis.

1.3 Project specification and design

This project I aimed to keep code and reporting together to provide easy reproduction. Codebase design to be extendable and modular. Therefore, I assign a sub-folder for all the project-specific code under the name *src*. Having a module in the same directory level with the other components allows the ability to use the code in notebook experiment as well as with the tests in the CI integration. Both

[1] https://travis-ci.org/

project proposal and report developed using the LaTeX typesetting system and documents kept in the version controlling to allow easy changes and rolling back to the desired version. Finally, root-level files such as *.travis.yml* and *requirements.txt* is instrumental in defining which steps to take in CI runs and constructing a near-identical environment for software dependencies. Below, I added a directory tree to serve as a guide for navigating and finding project files.

```
├── Proposal
│   └── Proposal files
├── Report
│   ├── chapters
│   │   └── Chapter files
│   ├── img
│   │   └── Images
│   └── Main project latex files
├── scripts
│   └── Utility scripts
├── notebooks
│   └── Experiment notebooks
├── src
│   └── Python library files
├── tests
│   └── Test files
├── .gitignore
├── .travis.yml
├── README.md
└── requirements.txt
```

1.4 Reproducibility Guidance

As a scientific project, it is very important that anyone can reproduce the experiments and findings in this project to verify the conclusions reached are accurate. Main components of reproducible research are open code, open data and repeatable software runtime specification. Open code component is the most straightforward among the other components as the source-code produced part of this project will be shared with the project reviewers and will be made public in GitHub [2] once the assessment of the project is completed. Dataset [15] used in this project is also available through the website URL cited and hosted in online data science community called Kaggle [3]. I have used Kaggle as the main source of accessing this data for two reasons, firstly for its functionality of allowing API calls to retrieve data and secondly for managing the data versioning for the user. Data versioning is an integral component of the reproduction of machine learning projects because the model produced by the training will heavily depend on the data it trained. The

[2]https://github.com/bbuluttekin/MSc-Project

[3]https://www.kaggle.com/paultimothymooney/chest-xray-pneumonia

current version of the dataset as of the writing of this project is version 2. To enable easier runtime replication and to leverage computational power I have chosen to use an online service called Google Colaboratory. [4] Colaboratory or *"Colab"* for short is a free service provided by Google Research. It will allow running Python code through the browser that connected to remote compute resources. Considering that Colab is a remote compute resource, I have created starter utility scripts to automate data acquisition. These files can be found inside the *scripts* folder. Please note that using these script will require obtaining API key from Kaggle platform and this API key file should be in the path specified in the scripts. However, reproducing in the Colab is optional and software dependencies required to produce local development environment is provided with the "requirements.txt" file. Lastly, custom software components for this project resides in the "src" folder and this folder must be placed in a location available to the scope of the python runtime.

[4] https://colab.research.google.com/

2 | Background

Generally, the first part of every machine learning project is to choosing the algorithm to tackle the problem in hand. As I stated in the proposal of this project, I choose to apply specific machine learning algorithms called Artificial Neural Networks (ANNs) to tackle the classification challenge of detecting pneumonia in X-ray images. The first part of this chapter I would like to provide some information to justify that decision.

The objective of the algorithm in this report is to classify X-ray images with or without pneumonia. Classification is a task of determining what is the defined class of an example given its data associated with it. In our case, we defined our classes for prediction to the person in the example image having pneumonia or not. In order to achieve that goal, machine learning algorithm must produce a function that outputs the class within defined finite possible classes such as $f : \mathbb{R}^n \rightarrow \{1, \ldots, k\}$ which in this case k is equal to 2. In essence all machine learning algorithms will map input representation of the data $\boldsymbol{x}$ to prediction output $\hat{y} = f(x)$. The only difference is how each algorithm is representing the data distribution with a model f. Which consequently leads to the question of which algorithm is the best algorithm for machine learning or which algorithm to choose to find the best model representation. According to **no free lunch theorem** [32] there is no such algorithm exist that will consistently achieve low error rate averaged over all possible distributions. In other words, no model is universally any better than any other machine learning model. Luckily, the objective in this project is not to find the universally best algorithm but rather to find the algorithm that will find the best representation for the data distribution of healthy and pneumonia X-ray images. Historically, traditional machine learning algorithms often performed poorly on tasks such as computer vision, detecting objects or speech recognition. Part of the reason is these task usually involve high-dimensional data. Because many traditional machine learning algorithms assume that any unseen data point should be similar to nearest training point, generalization in high-dimensional data such as image classification also suffers due to the fact that data points in these space are spread out and the notion of similarity weakens. Effect of this high-dimensionality also known as *curse of dimensionality*. Because of the weakness described earlier, Artificial Neural Networks emerges as a clear choice for image classification and object detection task which became evident with the performance of AlexNet [18], VGGNet [28] and ResNet [12] in the ILSVRC [4].

2.1 Building Blocks of ANN

The idea of Artificial Neural Networks inspired by the neural cells of the human brain. Earliest known research for ANNs dates back to 1943 as a multidisciplinary work of phycology and mathematics by Warren McCulloch and Walter Pitts [21]. Their work covered how computational logic could model complex neural cells activities. However, first development that reshaped the way for current generally accepted practices of ANNs was the work of Frank Rosenblatt's *"The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain"* [26]. Perceptron is the simplest linear ANN that always converges to hyper-plane when there are two sets of classes that can be linearly separable. Like the current single neurons generally used in modern ANNs, for producing output it calculates the sum of weighted inputs and applies a *step function* to that sum. For example, let the input $\boldsymbol{x}$ be n-dimensional input vector, Perceptron first calculates $z = w_1x_1 + w_2x_2 + \ldots + w_nx_n = x^Tw$ then pass this weighted sum of inputs to step function $h(z)$ below to calculate final output.

$$h(z) = \begin{cases} 0, & \text{if } z < 0 \\ 1, & \text{if } z \geq 0 \end{cases} \tag{2.1}$$

Current ANN units also have same properties with the exception of the use of step function. Instead current ANN units uses set of non-linear functions that generally called *activation functions*.

Despite its robust nature Perceptron is ultimately a linear model which implies that it can only be effective for the data distributions that can be linearly separable. The popularity of the algorithm is faded as the limitations such as not being able to separate logical operation exclusive OR (also know as XOR) is discovered [20]. However, later on, it is discovered that these limitations can be eliminated by just using layers of many Perceptrons together as a single model and the resulting model is called *Multilayer Perceptron* (MLP). (*Feedforward Network* is another term that usually used interchangeably with Multilayer Perceptron.) MLPs uses a concept called layer which is useful for calculation and managing the architecture of the model. In a nutshell, a layer is the combination of neurons with bias neuron stack together (with exception of output layer) that have the same input source. Concept of a layer is used in many different architectures, for example, a layer with each neuron that connected to each unit in previous and next later is known as *Fully connected layer* or *Dense layer* whereas layer with pre-defined convolution operation is called *Convolutional layer*.

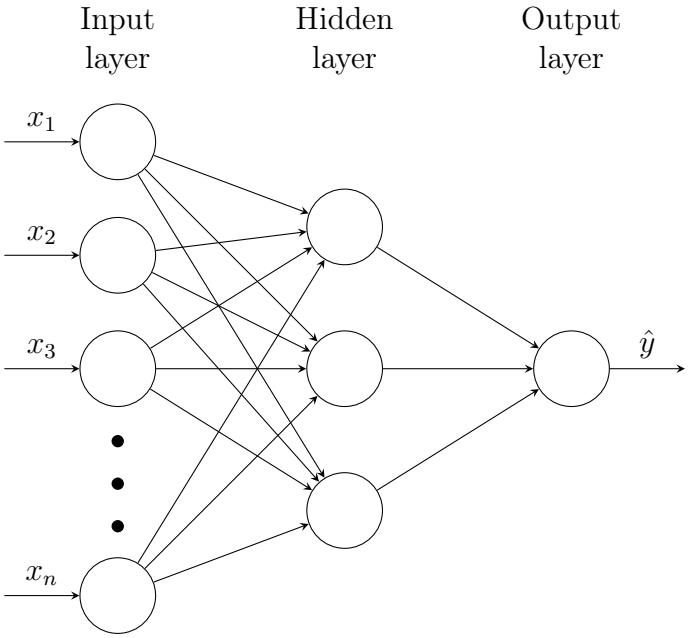


Figure 2.1: Illustration of a MLP model.

2.2 Exploding and Vanishing Gradients

Having the sequential architecture described in section 2.1 MLPs faces an additional implementation challenge that is not common in traditional machine learning, the difficulty of training. The typical training process for the machine learning algorithm is at a very high level is a standard procedure. The first step is to feeding input data to model and produce an output, then based on desired output for the input loss value can be calculated. Loss value is a calculation of how much the output is further away from the desired output. Using this loss value we can approximate weight updates with the help of the optimization algorithm until the model weights converge. Even though the training process is also the same for the MLPs there is more complexity involved in MLPs giving that instead of dealing with one model we have layers of neurons chained together that each has its own weights. For those reasons training MLPs was a challenge until the influential *backpropagation* paper [27] is published. In this paper efficient technique of calculating the gradients using forward and backwards passes demonstrated. Utilizing the chain rule of calculus, the backpropagation algorithm was able to calculate the update for each weight in every neuron.

Notwithstanding help of the backpropagation algorithm, as the need for training deeper networks increased, the difficulty of training such networks remained a problem. Part of the problem was, as the gradients get smaller and smaller as the gradient flowed down to lower layers (layers close to the input layer) of the network. When the gradient updates get close to very small values some of the lower layer weights do not updated enough and consequently not converging model to appropriate representation. This phenomenon is usually referred to as *vanishing gradients* problem. Similarly, some network such as recurrent neural networks

can have an opposite problem that resulting in gradients getting larger and larger which pushes weights to be a very large number. Similarly, this phenomenon referred to as the *exploding gradients* problem. Later on, this unstable gradient flow problems are shown to be the combination of choice for weight initialization and the characteristic of the activation functions that widely used [8]. Previously, activation function often get used was the sigmoid (also known as logistic function) $\sigma(x) = \frac{1}{1+e^{-x}}$ and the hyperbolic tangent function $(tanh(x) = 2\sigma(2x) - 1)$ have a flatten tails where $\boldsymbol{x}$ get very large or very small. Giving that the chain rule states that the derivative of a variable is equal to the product of the partial derivatives of the composite function with respect to output, neurons that produce an output very large and very small will not likely to be updated. For example lets suppose $\hat{y} = \sigma(z(w, b))$ and $z(w, b) = wx + b$. Then chain rule states that

$$\frac{\partial \hat{y}}{\partial w} = \frac{\partial \hat{y}}{\partial \sigma} \frac{\partial \sigma}{\partial w} \tag{2.2}$$

It is clear that the gradient update of $\boldsymbol{w}$ is predicated on the gradient of the σ is not being zero. While the sigmoid and hyperbolic tangent functions derivatives get close to zero when the functions are on the saturating region.

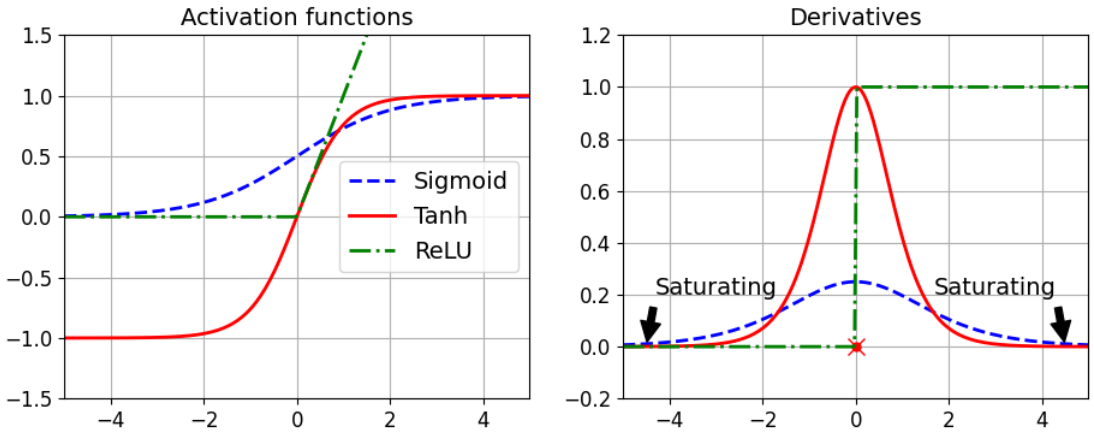


Figure 2.2: Saturation points in activation functions.

Due to these properties, *Rectified Linear Unit* (ReLU) activation function emerges as a good alternative to train deeper neural networks. ReLU behaves like a linear function for the non-negative values $(f(x) = max(0, x))$ and outputs zero for the negative input values. With that characteristic gradient of the ReLU will be either zero for the input values less than or equal to zero and one for any other values. Because of consistent properties of the non-saturating functions, it is always better to use such activation functions for hidden layers to ensure the gradient flow is faster and Network will converge faster as a result.

2.3 Optimization

Most machine learning algorithms utilize some sort of optimization. The main objective of the optimization algorithms is to find the maximum or the mini-

mum point for the function in hand with respect to one or more variable. For the machine learning domain, we would like to optimize the loss function (also known as a cost function) to the global minimum for having a model that is most representative of the data. The task of searching minimum or maximum can be used interchangeably as finding the minimum point of the function f may also be found by getting maximum of the inverse function of the same function (f^{-1}). In essence, the process is achieved by taking derivative of the function which will give us a slope for the given point, the moving opposite to the slope with the small increments until we reach to the minimum point where the gradient is zero. The process also known as *gradient descent*. Gradient descent used in machine learning field for a long time, however depending on the complexity of the function space this process could take a significantly long time. For mitigating this problem new set of algorithms called *momentum optimizers* created. The difference between these algorithms and gradient descent is simply, gradient descent will take small consistent steps regularly throughout the optimization process. How these momentum-based algorithms works are they generally pay attention to the previous gradients and accelerate the updates based on gradients and the slope at the point of the gradient. This will help the optimization to converge to global minimum faster when there is a lot of plateau areas in the function surface. Later on, momentum algorithms expanded to include taking the steepest dimension of the gradient to point updates more toward to the global minimum, a new family of optimizers named as *adaptive* optimizers. Most well know algorithms of this category is AdaGrad [6], RMSProp [30], Adam [16] and Nadam [5]. Despite the fact, these adaptive optimizers usually converge to good solutions faster research by Ashia C. Wilson et al [31]. showed that in some cases these optimizers can lead to poor generalization result depending on the dataset. It might be good practice to try adaptive and momentum-based optimizers in the training job to observe their effect in generalization.

2.4 Regularization and Over-fitting

Generally, the fundamental challenge in machine learning is that our model should perform well on previously unseen data points which is also referred to as generalization. If the scope of the model is sufficiently large (model with a large number of parameters) optimization techniques described previously could enable the model to capture noise that specific to training records. In other words *overfitting* is where the model will focus on individual nuances of data records instead of focusing on general pattern overall. However, there are numerous regularization techniques available to mitigate the overfitting in ANNs. Just like traditional machine learning l regularization (norm) can be applied to the artificial neural networks. In general form l_p norm is given by

$$\|x\|_p = \left(\sum_i |x_i|^p \right)^{\frac{1}{p}} \tag{2.3}$$

for vector $\boldsymbol{x}$ and $p \in \mathbb{R}$, $p \geq 1$.

Most frequently used l norms are l_1 and l_2 regularization. The l_1 norm usually used when the difference between elements are important. This regularization seldom referred to as *Manhatten distance* in the literature. Similarly, another norm get used often is l_2 norm, also known as *Euclidean distance*, is the geometric distance between points and could be calculated simply as $x^T x$

In addition to l_p norms *dropout* [29] emerges as a popular regularization technique specific to the ANNs. Dropout is a process of removing a fraction of the units from a neural network layer by setting the weight of the unit to zero during training. This technique reduces the reliance of each unit for the prediction. In each training iteration fraction of the neurons dropped randomly so the remain neurons are forced to train in the absence of the neurons picked. Although dropout has proven to improve generalization in many machine learning tasks, due to its unstable nature time it takes for the network to converge usually increases.

2.5 Convolutional Networks

In the proposal of this project, I used the definition from Deep Learning book [9] for the definition of Convolutional Neural Networks (CNNs) that I believe summarize the concept very well. To quote that definition again CNNs are:

> "Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers."

In CNN's neurons replaced by units usually called kernel or filter. These units are rectangular matrix span over the layers of the input data. The way this unit calculates its output by positioning each kernel item to pixels on the image and calculate the element-wise multiplication between them. The kernel then slides over the rest of the image until all image covered. The area that processed by the image is called *receptive field* and operating this way by calculating each area in image CNNs are able to capture information about the locality of the image as opposed to flattening. The output of the operation depends on the design of the convolution operation. There are two main concept defines the operation, one is stride or how many pixels the kernel will be moved to complete the operation, and the size of the padding. Padding is zero value pixels that get placed around the image to centralize the items at the edge of the image in kernels.

The output of the convolutional network can be calculated easily to ensure no bug is introduced to the model during implementation. Given the volume size of the input image ($\boldsymbol{I}$), kernel size of CNN ($\boldsymbol{K}$), stride which is applied ($\boldsymbol{S}$), and the amount of zero padding applied is ($\boldsymbol{P}$). We can calculate the output as:

$$output = \frac{(I - K + 2P)}{S} + 1 \tag{2.4}$$

The generally accepted way of padding in convolution operations are, the SAME and the VALID padding. SAME padding is using zero paddings around the image to output *same* size output from the convolution when chosen a stride

is one. VALID padding, however, is a practice of not applying any zero paddings and only uses *valid* pixels from the image.

Conventionally, convolutional layers are used together with the pooling layer. The main contribution of the pooling layer is to under-sample the output data. Doing so they will reduce the memory usage and the computational complexity of the network. Additionally reducing the size of the output will reduce the chance of overfitting. Most frequently used versions of the pooling layer are max pooling and average pooling. Regardless of the operation pooling layer works by choosing the pool size and applying the desired operation to that field in the input. For example, if it is a max-pooling layer, the maximum pixel value in the polling size is passed on to the next layer.

2.6 Software Challenges Specific to Machine Learning Systems

It is clear from some of the problems mentioned earlier, training deep neural networks is a difficult task. One particular reason for difficulty when applying ANNs to new fields is identifying the problem when getting bad scores from the model. Because unlike tradition software development it is difficult to know whether there is a bug in the implementation of the software or the model itself is not suitable to represent the underlying data distribution. For overcoming this challenge I am employing a diagnosis and debugging check list that will help identify and eliminate any bugs that can occur in the implementation phase. Where applicable checklist points are:

- Overfit to sample data to ensure no optimization mistakes made.
- Monitor training loss during training and make sure it is declining throughout the training.
- Check the input and output shapes are correct.
- Make minor changes to models and only add addition changes after ensuring there is no bug in the system.

Another challenge is the computationally expensive nature of the training. Due to the iterative nature of the machine learning, it is vital that the experimentation for the task will run sufficiently quick that the improvement for the model can be identified quickly and progressed to further stages. However, standard ANNs usually have millions of parameters that need to be updated while training with hundreds of thousands if not millions of training examples to train on. Considering the average training process will need multiple passes on the entire dataset, the need for high computation power becomes evident. That's why most training is done with the help of hardware accelerators such as GPUs or TPUs. Choice of computation environment for this project luckily includes such hardware accelerators but these special hardwares must be detected and enabled so the calculation

can take advantage of the compute power. To enable the hardware related capabilities and to reduce training management overhead I have written a wrapper module in the custom code library. Some of the other features that made possible with this module are:

- Ability to detect and set the hardware accelerators.
- Choosing training strategy suitable for the hardware accelerator.
- Logging performance metrics persistently.
- Saving model persistently in case of computing failure and continue where the training left previously.
- Extract final model with runtime data.

Possibly one of the most important features of the helper library is providing resilient training. Some large models take hours or days to train and my choice of computation environment allows only twelve-hour training before terminating. Not to mention virtual machines power that environment might disconnect or terminate much earlier. In the case of such an event, if all the previous training is lost, time spent on training will be wasted. If similar events happen many more times it could even risk not completing all the experimentations on time for this project deadline. To eliminate that risk, I have implemented a process that will save model weight to storage outside the compute environment regularly. If such failure happens, the module will pick up the last saved weights and continue to training where it left of previously. In addition to starting the training with the previous model, by using naming conventions, the module can initialize the epoch number for the training accordingly and allow accurate comparison between other models. In other words, if the first training attempt is failed in epoch number 23 restarting training will start with the epoch number 23.

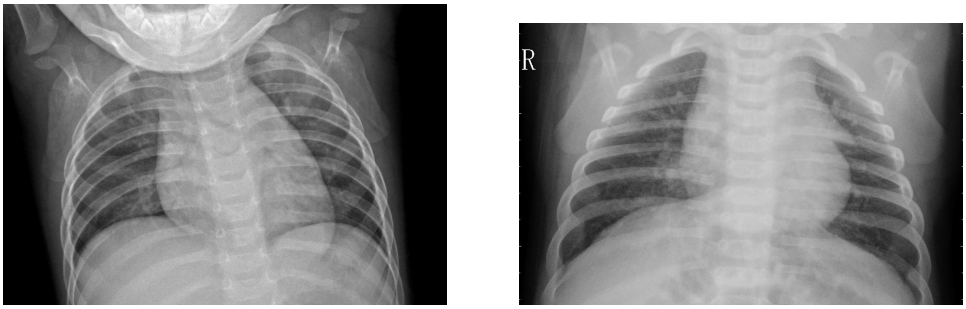
2.7 Literature Review

Applying machine learning to medical imaging have seen rise of popularity with the emergence of the popularity of neural networks. Such algorithms applied to many medical diagnosis problems such as Gulshan et al [10] diabetic retinopathy detection or Estava et al [7] cancer classification etc. More specifically performance of convolutional neural networks and localization studied in Islam et al [14] by using OpenI [23] dataset. However, because of the limited number of examples in the dataset for this project perhaps the most impactful part of the literature was the transfer learning in the medical imaging field. Use of transfer learning become ubiquitous when it comes to eliminate the learning problems in the small datasets. But use of transfer learnings effectiveness is an open question in this problem because of the dissimilarity between the source dataset of learned features such as imagenet versus the dataset of this project. Performance of the pre-trained networks have shown to be lower in such cases by Yosinski et al [33]. Actual benefit

from transfer learning is not trying to reuse the feature dissimilar to pneumonia detection but to utilize the pre-trained weight as a good initialization step to start the training and research has shown this method is better than just using trained features out of the box [24, 17, 11]. Evidence of effectiveness of this method is also proven by the results of CheXNet [25] which have fallowed the initialization of the Dense convolution networks this way to surpass practitioner level accuracy in X-ray images.

3 | Data

Dataset [15] for the project is X-ray images collected at Guangzhou Women and Children's Medical Center, Guangzhou from pediatric patients aged between one to five. All X-ray images are in JPEG format and organized into a folder-based structure. Main sub-structures in the dataset are train, validation and the test folders. All main folders also are broken into sub-folders that named *Normal* and *Pneumonia* to indicate their classification. All X-rays have graded by expert physicians and check individually, damaged or un-readable X-rays discarded. Below are the illustrative examples from both classes of the X-ray.



(a) X-ray without Pnuemonia (b) X-ray with Pnuemonia

Figure 3.1: Two sample X-ray Chest images with and without pneumonia.

For a more detailed understanding of the data, I have checked the content of each train, validation and test datasets. There is a significant imbalance in between Pneumonia and Normal classes for train and the test datasets. A detailed breakdown of each dataset is given in table 3.1.

Dataset Name	No of images	Pneumonia	Normal
Train	5216	3875	1341
Validation	16	8	8
Test	624	390	234

Table 3.1: Breakdown of images for each classification folder

3.1 Data Augmentation

Data augmentation is a process of producing additional training instances by modifying existing training data points. Process of data argumentation is very common within the computer vision field. Generally, because it is well understood that the images could be modified some way without having to lose the classification of the image. For example image of a cat on the table can be cropped such a way that will still contain the cat on the table but without the background that is not relevant. The resulting image from this cropping will also be an image classified as a cat image.

There are similar sets of transformation that could be applied to images to create labelled data instances. It is import to choose which augmentation to apply and which one to abandon. Example to highlight some argumentation that is not appropriate is flipping the image upside down. Even though flipping upside down is acceptable some cases like the cat image example we considered earlier. It would not be useful in the setting of Pneumonia classification because considering upside down X-rays is not a natural occurrence in the field of medical diagnosis. Perhaps the most striking example of an area that this argumentation technique should not be used is handwritten digits classification. Reason for that is although upside number one is still an image of number one, in the case for number six it will result in images labelled as six but in content, they will appear as number nine. Train machine learning model with samples of number nine labelled as six will indeed hinder the accuracy of the model instead of helping it.

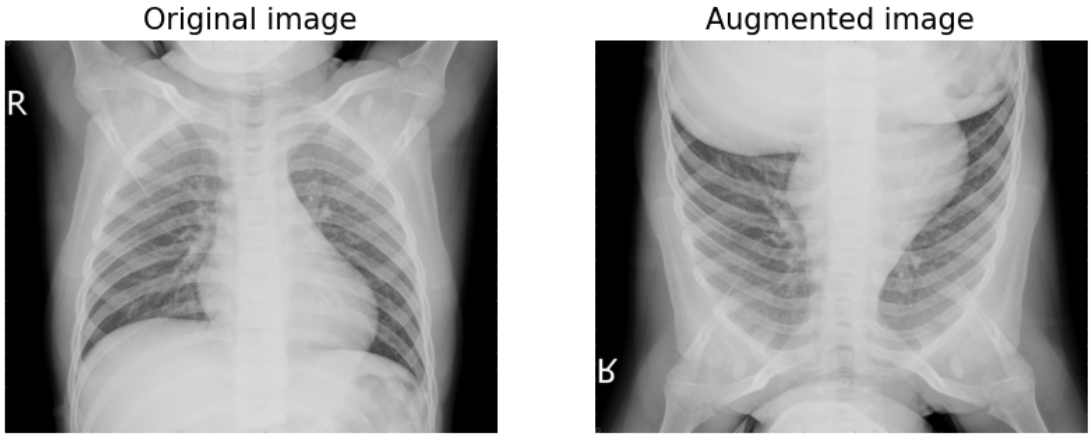


Figure 3.2: Flipping an image upside down is not applicable in CAD context

3.1.1 Horizontal flip

This data augmentation is achieved by reversing the pixel values horizontally. The resulting image would look like a mirror flip from left to right of the same image.

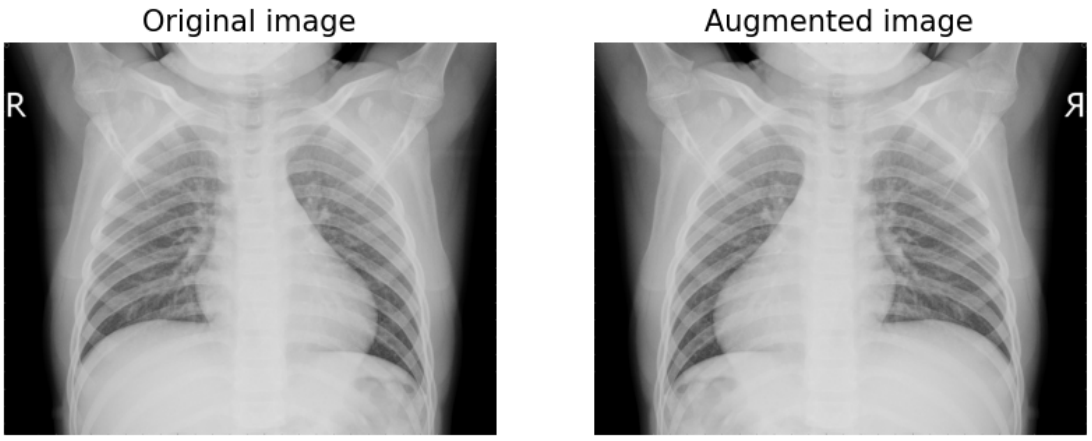


Figure 3.3: Horizontal flip augmentation

3.1.2 Random zoom augmentation

Also knows as random cropping augmentation. This argumentation applied by randomly sampling a certain percentage of the original image and either adding padding to a sampled area or rescale the sampled area with interpolation to produce an image of the same shape as the original image. It is important to choose an appropriate percentage for this argumentation. The percentage should be chosen in a way that it would not omit the area of interest in the image. For the purpose of my project, this argumentation must be applied such a way that it would not leave any part of the lungs outside of the image. As an example figure 3.4, is applied with 90% of random zoom and it is a considerable level for this dataset.

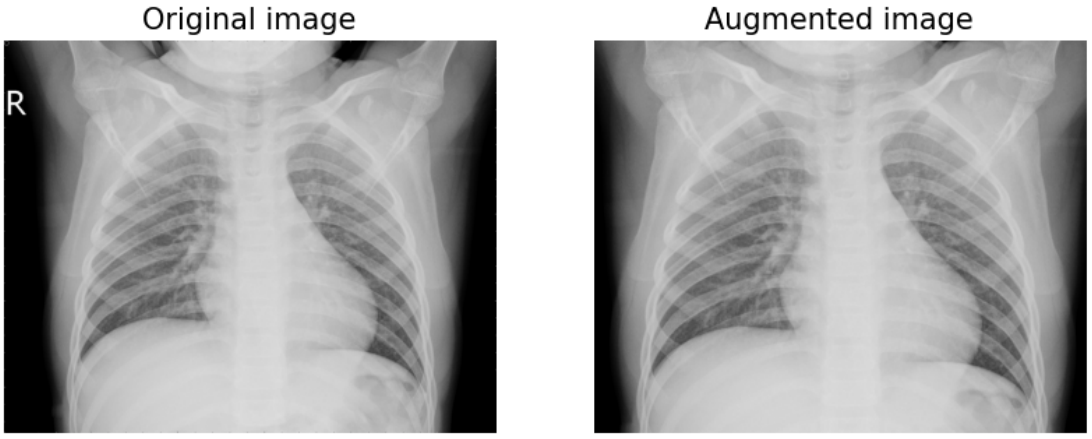


Figure 3.4: Random zoom augmentation

3.1.3 Changing brightness or saturation

These argumentations either change the brightness or saturation of the image to create an augmented image. Saturation is done by converting RGB images to

HSV (Hue Saturation Value) then multiplying saturation channel with saturation factor chosen for the augmentation.

Brightness augmentation also works a similar way. RGB values of the image are converted to float representation then applied brightness factor to the values.

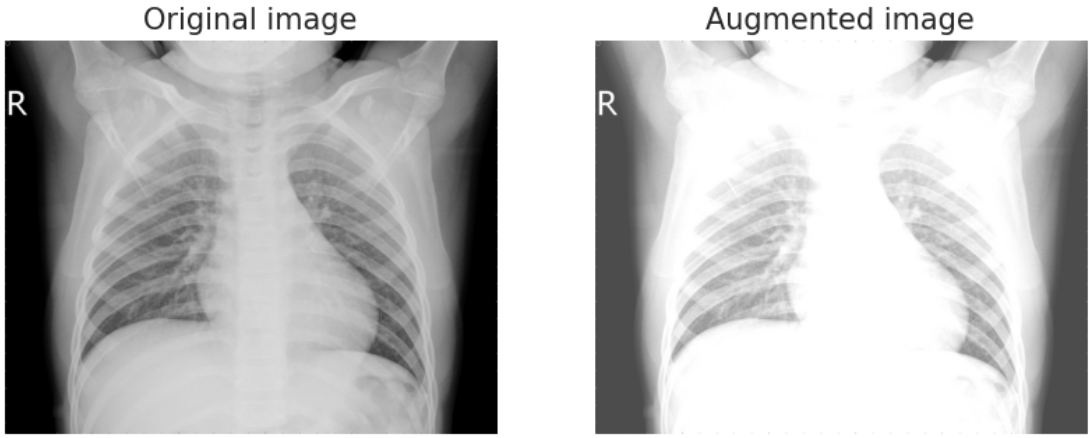


Figure 3.5: Brightness augmentation example

3.2 Data Representation

Original and augmented images need to be turned into data representations that machine learning algorithms can process. Normally, colored images represented with pixel values for red, green and blue (RGB) color channels and values for each pixel in any given channel ranges between 0 to 255. For the traditional machine learning techniques, each data point should be represented in one vector. Additionally pixel values for each color channel needs to be turned into a vector by flattening each row of pixel values for particular color channel. After applying the same step to each channel flattened, channel vectors should be appended one after another to get the final vector for the image. Given that the model size cannot change during training each image should be resized to the same shape before representational vector is created (e.g. $\boldsymbol{x} = [x_1, x_2, \ldots, x_n]$).

Representation for CNNs, however, require different processing because the convolution operation requires locality of the data points to be taken into consideration. Therefore the most common approach is to represent each colour channel of the image as an array of vectors which commonly know as *matrix*. Because the coloured image has three colour channels each matrix for the channels should be combined as a multi-dimension matrix. This multi-dimensional matrix representation usually get referred to as *tensor*.

Much like input data, the label for each image should also be represented in appropriate mathematical entity to be able to process in training. Because this problem has only two classifications most used method is to encode each class in binary (e.g. 1 for pneumonia and 0 for normal). However it can also be represented in one-hot encoding, which will return a vector for each label includes binary values for each class (e.g. $[0, 1]$ for pneumonia and $[1, 0]$ for normal).

3.3 Limitations of the Dataset

Although dataset I have choose is well designed and validated, some aspects of data has undesired properties. First of such property is the size of the validation data. As I mentioned earlier dataset has a folder where it hosts validation files. Within this folder there is sixteen images where each class have eight images each. This is significantly small validation dataset and would reduce the bins for each available accuracy step significantly. For example making one error in validation data will result accuracy declining from 100% to 93.75%.

Second undesired property is the high variance of the image resolutions. Having such a high variance in image resolution makes it difficult to choose right resolution to resize all the images. To show the variance within the data I have added a density plot that illustrates the height and width of the images and their distribution in Figure 3.6.

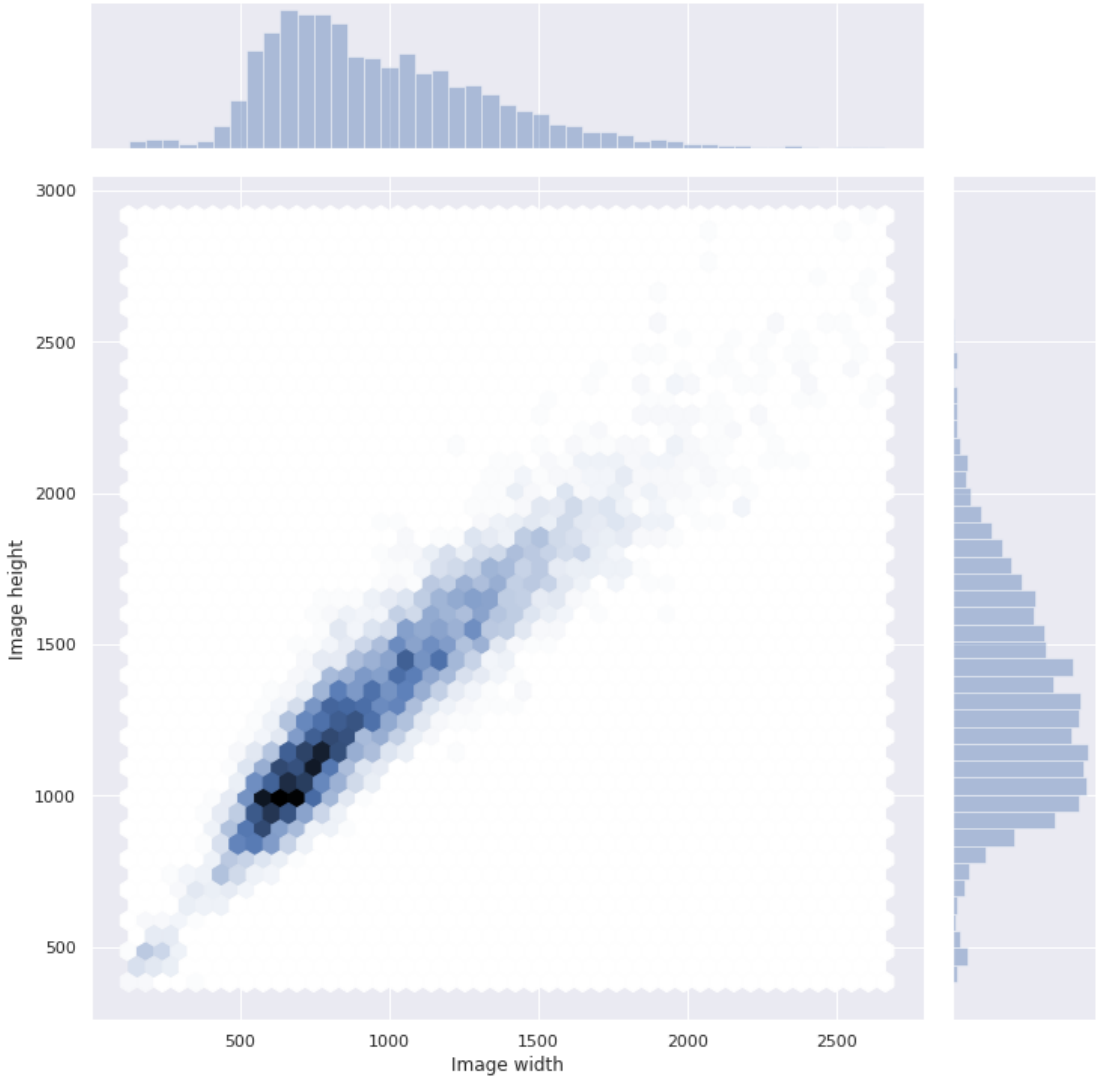


Figure 3.6: Density plot of image dimensions

This graph gives an insight for what resolution for the image should be chosen to have minimal informal loss and distortion in the final image.

3.4 Data Processing

Inline with the information provided in section 3.2, I have designed two separate data processing pipeline. First pipeline created for data processing for so-called traditional machine learning algorithms such as Random forest or Support vector machine (SVM) that will be used in performance benchmarking. This pipeline does the flattening of the X-ray images discussed and depends on open-source software libraries keras [3] and numpy [22].

The second pipeline is created for pre-processing of CNN models which would be used in the majority of the time in training and testing. The module is designed to be lightweight with minimal dependencies. Only third party open source libraries used are, Numpy and the TensorFlow [1]. Although TensorFlow has high-level Keras module that allows easier implementation for data pipeline, I have decided on building data processing pipeline in tf.data module of the TensorFlow library. Building my pipeline in tf.data meant that I have to build all of the implementations from scratch myself but it also allowed me to build more flexible and more performant pipeline I could have had than the alternative. The goal of having faster training runs and ultimately having shorter prototyping time only achievable if the training is fast enough to feed data into powerful hardware accelerators. Otherwise, the process will become a bottleneck in the training because the processing unit whether it is GPU or any other device will remain idle until the data fed to the system. Tf.data designed with parallelization in mind that carries out the process of fetching next data batch while the training for the current batch in progress so the next batch can be ready as soon as the training is finished. Providing that all the augmentations and pre-processing will be done in CPU while GPU is training on the previous batch in effect means all the augmentations are computationally free.

At the beginning of this chapter, I point out that the dataset for this project has an imbalance problem. Building the data pipeline with the tf.data also meant that I am able to eliminate this issue by making a custom pipeline operation that creates a balanced training set. With the flexibility provided, I was able to apply the augmentations to only minority class of the dataset and add this new training example to training set until I have the same number of examples for both classes.

4 | Methodology

This chapter lays out the methodical steps I took while implementing the project. Main steps for implementing this project fall into four general categories. These four categories are:

- Establishing a baseline benchmark
- Improving metrics beyond baseline benchmark
- Ensuring model interpretability is maintained
- Deploying the final model to production

Further sections will provide detail for these four main categories.

4.1 Establishing a benchmark

The first step to every machine learning project is that model produced is to perform better than any random or naive solution. Although random guess in any binary classification suggests that the minimum accuracy should be greater or equal than 50%, because of the imbalance in the test dataset actually requires higher accuracy for this project. Mainly because we have 390 pneumonia images in the 624 total images in the test dataset dummy classifier that predict pneumonia for any given image will achieve 62.5% accuracy. However, given the imbalance of the test dataset accuracy would not be the choice of a good metric for assessing the performance. For that reason on any experiment conducted, collecting precision, recall as well as calculating f1 score will be utilized. If I introduce these metrics in more detail. Precision is calculated by dividing true positives (tp) in predictions to true positives and false positives (fp).

$$Precision = \frac{tp}{tp + fp} \tag{4.1}$$

The recall is calculated by dividing true positives to true positives and false negatives (fn).

$$Recall = \frac{tp}{tp + fn} \tag{4.2}$$

To capture the correct performance of the classifier we need to consider both precision and recall. F1 score provides the ability to consider both precision and

recall for the same classification problem because it is calculated by getting a harmonic mean of the precision and recall.

$$F_1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} = 2 \times \frac{precision \times recall}{precision + recall} \tag{4.3}$$

Next part for the benchmarking is choosing which algorithms to train on the data for benchmarking. Given we don't know the distribution of the data, training fundamental machine learning algorithms together with the neural network algorithms is a cautious step to take. For that reason, I have chosen to train two fundamental machine learning algorithm, namely the Random Forest classifier and the Support vector classifier as a part of establishing a benchmark.

4.1.1 Random Forest Classifier

For the past years, tree-based algorithms have been very popular in the academic community as well as in the industry with numerous papers demonstrated in ICML, NIPS and JMLR. Random forest emerges in this category as a very strong algorithm by Breiman [2] that achieves remarkable performance in small to a medium dataset. The final baseline will be determined by finding an optimal number of estimators and maximum features using cross-validation.

4.1.2 SVM Classifier

In addition to Random Forest, the secondary mainstream machine learning algorithm is Support Victor Machine classifier [13]. The biggest factor for choosing this algorithm was its robustness in detecting non-linear features in the data using kernel trick. Similarly to Random Forest hyper-parameters chosen with cross-validation.

4.1.3 LeNet-5

LeNet-5 [19] is the first one of the well-known CNNs that I will apply to this problem to consider in baseline benchmark. I kept the attributes of the network as close to the origin network as possible but some characteristic of the network had to be changed when it is trained on this classification problem. The first difference in this project needed in input and output layers, MNIST dataset have ten classes therefore required ten dense neurons at the output layer in the original network but in this project, there are two classes to predict which can be achieved by having one dense output neuron instead. Input layer also changed because 32×32 is not a suitable resolution for the pneumonia detection and changed to 224×224 for better representation. Another change made to this model is using relu activation function rather than hyperbolic tangent function (tanh) because of the less than ideal gradient flow in saturated tanh function prevented model to converge to a good solution. Detail of the model summary given below for reference.

```
    Model: "LeNet-5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_6 (Conv2D)            (None, 220, 220, 6)       456
_________________________________________________________________
average_pooling2d_4 (Average (None, 110, 110, 6)       0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 106, 106, 16)      2416
_________________________________________________________________
average_pooling2d_5 (Average (None, 53, 53, 16)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 49, 49, 120)       48120
_________________________________________________________________
flatten_2 (Flatten)          (None, 288120)            0
_________________________________________________________________
dense_6 (Dense)              (None, 120)               34574520
_________________________________________________________________
dense_7 (Dense)              (None, 84)                10164
_________________________________________________________________
dense_8 (Dense)              (None, 1)                 85
=================================================================
Total params: 34,635,761
Trainable params: 34,635,761
Non-trainable params: 0
_________________________________________________________________
```

4.1.4 AlexNet

Similar to LeNet-5, AlexNet [18] also aimed to the kept original structure as much as possible. AlexNet comprises of eight layers, five of those were convolution layers where some of them connected to max-pooling layer. Despite the fact, the architect is preserved almost same as the original implementation additional steps such as adding local response normalization or PCA augmentation is not applied because of the ad hoc nature of the process and limited effect of this processes in the final performance. AlexNet is ultimately a very influential paper that steers the direction for how the CNNs are designed and fueled the adoption of the use of neural networks in many fields. Fallowing quote from the paper also explains the reason why neural networks gain so much popularity and pushing the state of the art results year after year.

> *"All of our experiments suggest that our results can be improved simply by waiting for faster GPUs and bigger datasets to become available."*

```
    Model: "AlexNet"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
```

```
=================================================================
conv2d_10 (Conv2D)           (None, 54, 54, 96)        34944
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 26, 26, 96)        0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 26, 26, 256)       614656
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 12, 12, 256)       0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 12, 12, 384)       885120
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 12, 12, 384)       1327488
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 12, 12, 256)       884992
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 5, 5, 256)         0
_________________________________________________________________
flatten_2 (Flatten)          (None, 6400)              0
_________________________________________________________________
dense_6 (Dense)              (None, 4096)              26218496
_________________________________________________________________
dropout_4 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_7 (Dense)              (None, 4096)              16781312
_________________________________________________________________
dropout_5 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_8 (Dense)              (None, 1)                 4097
=================================================================
Total params: 46,751,105
Trainable params: 46,751,105
Non-trainable params: 0
_________________________________________________________________
```

4.1.5 VGGNet

VGGNet is one of the best performant in 2014 ILSVRC competition that get a best performance in classification and localization task. There are two different variation of this network is available namely the VGGNet 19 and VGGNet 16. For this project 16 layered VGGNet 16 is a good choice it has a sufficient capacity given that the dataset is relatively small. Main contribution of the VGGNet is that it demonstrated the importance of the network depth to good performance. Layer structure of the architecture is straightforward, only performs 3×3 convolution with 2×2 pooling. Similarly, for the purpose of the benchmarking architecture kept as close to original as possible with only change is made to final layer replaced with one neuron dense layer to accommodate the binary classification task. Advantage

of using this network is that it is implemented in most of the modern software packages and it also available to initialize with the weights of imagenet. That property will be very useful for comparing the performance when transfer learning is explored in subsection 4.2.1. Detail of full network listed below.

```
    Model: "VGGNet"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
vgg16 (Functional)           (None, 512)               14714688
_________________________________________________________________
dense_3 (Dense)              (None, 4096)              2101248
_________________________________________________________________
dropout_2 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_4 (Dense)              (None, 4096)              16781312
_________________________________________________________________
dropout_3 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 4097
=================================================================
Total params: 33,601,345
Trainable params: 33,601,345
Non-trainable params: 0
```

4.2 Improving Performance

After establishing a benchmark for minimum acceptable performance, this step of will focus on continuously improving the performance on the problem. Materialize that goal I have chosen two method, applying transfer learning and building custom neural.

4.2.1 Transfer Learning

Transfer learning is fundamentally taking weights (or features) learned on one problem, and applying them to similar or new problem that could benefit from it. Transfer learning usually considered when the training data is too small to train a deep neural network from a scratch. Process for transfer learning is simple, architecture that will be use in problem in hand is initialized with the weights learned from another problem without the top part of the model. Here top part is refers to the layer(s) at the end of the model which has got the high level of the features specific to the problem. And given that the lower layers of the model contribute to simply features like vertical and horizontal lines or edges,

weights in these layers held frozen to allow feature extraction from the images in new problem. After that step top part of the model initialized with the random weight to be trained on the new dataset to learn higher level features. Transfer learning sometimes get adopted with the concept of *fine-tuning*, which is a process of unfreezing the part of it or the entire model and re-training it with the new data.

Because the dataset of this project is suffering from a small data problem, transfer learning is utilized to discover if generic datasets like imagenet [4] can be helpful to improve existing model. Effectiveness of the transfer learning will be measured by training the VGG16 network with the imagenet weights and comparing the performance against same network with randomly initialized version from the benchmarking experiments.

4.2.2 Custom Model Architecture

- Learning rate α
- Momentum term β
- Number of layers
- Number of units in a layer
- Kernel size for convolution layers
- Learning rate decay
- Mini batch size

4.3 Model Interpretability

4.3.1 GradCAM

4.4 Deployments with CI/CD

5 | Design and Experiments

5.1 Benchmark Experiments

base (https://tensorboard.dev/experiment/p2mkNoLORACZlCXL2UQRbg)
augmented (https://tensorboard.dev/experiment/dmQIbMeJQtqF70MhQQqySA)

5.1.1 Random Forest Classifier

5.1.2 SVM Classifier

5.1.3 LeNet-5

5.1.4 AlexNet

5.2 Custom NN Architecture

5.3 Transfer Learning

5.4 Interpreting Model Decisions

6 | Model Deployments

7 | Conclusion

References

[1] Martin Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[2] Leo Breiman. "Random Forests". In: *Machine Learning* 45.1 (2001), pp. 5–32. ISSN: 0885-6125. DOI: 10.1023/A:1010933404324. URL: http://dx.doi.org/10.1023/A%3A1010933404324.

[3] François Chollet et al. *Keras*. https://keras.io. 2015.

[4] J. Deng et al. "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR09*. 2009.

[5] Timothy Dozat. "Incorporating Nesterov Momentum into Adam". In: 2016.

[6] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *J. Mach. Learn. Res.* 12.null (July 2011), pp. 2121–2159. ISSN: 1532-4435.

[7] Andre Esteva et al. "Dermatologist-level classification of skin cancer with deep neural networks". In: *Nature* 542 (Jan. 2017). DOI: 10.1038/nature21056.

[8] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*. 2010.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[10] Varun Gulshan et al. "Development and Validation of a Deep Learning Algorithm for Detection of Diabetic Retinopathy in Retinal Fundus Photographs". In: *JAMA* 316 (Nov. 2016). DOI: 10.1001/jama.2016.17216.

[11] Kaiming He, Ross B. Girshick, and P. Dollár. "Rethinking ImageNet Pre-Training". In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)* (2019), pp. 4917–4926.

[12] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 770–778.

[13] Marti A. Hearst. "Support Vector Machines". In: *IEEE Intelligent Systems* 13.4 (July 1998), pp. 18–28. ISSN: 1541-1672. DOI: 10.1109/5254.708428. URL: https://doi.org/10.1109/5254.708428.

[14] Mohammad Tariqul Islam et al. "Abnormality Detection and Localization in Chest X-Rays using Deep Convolutional Neural Networks". In: *CoRR* abs/1705.09850 (2017). arXiv: 1705.09850. URL: http://arxiv.org/abs/1705.09850.

[15] Daniel Kermany and Kang Zhang. *"Labeled Optical Coherence Tomography (OCT) and Chest X-Ray Images for Classification"*. 2018. DOI: http://dx.doi.org/10.17632/rscbjbr9sj.2.

[16] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2015).

[17] Simon Kornblith, Jon Shlens, and Quoc V. Le. "Do better ImageNet models transfer better?" In: 2019. URL: https://arxiv.org/pdf/1805.08974.pdf.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[19] Yann Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE*. 1998, pp. 2278–2324.

[20] Minsky Marvin and A Papert Seymour. *Perceptrons*. 1969.

[21] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259.

[22] Travis E Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.

[23] *Open Access Biomedical Image Search Engine*. https://openi.nlm.nih.gov/. Accessed: 2019-03-12.

[24] Maithra Raghu et al. *Transfusion: Understanding Transfer Learning for Medical Imaging*. 2019. arXiv: 1902.07208 [cs.CV].

[25] Pranav Rajpurkar et al. "CheXNet: Radiologist-Level Pneumonia Detection on Chest X-Rays with Deep Learning". In: *CoRR* abs/1711.05225 (2017).

[26] F. Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain". In: *Psychological Review* (1958), pp. 65–386.

[27] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning Internal Representations by Error Propagation". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. Cambridge, MA, USA: MIT Press, 1986, pp. 318–362. ISBN: 026268053X.

[28] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. 2015. URL: http://arxiv.org/abs/1409.1556.

[29] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[30] T. Tieleman and G. Hinton. *Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude*. COURSERA: Neural Networks for Machine Learning. 2012.

[31] Ashia C Wilson et al. "The Marginal Value of Adaptive Gradient Methods in Machine Learning". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., 2017, pp. 4148–4158. URL: http://papers.nips.cc/paper/7003-the-marginal-value-of-adaptive-gradient-methods-in-machine-learning.pdf.

[32] D. H. Wolpert and W. G. Macready. "No Free Lunch Theorems for Optimization". In: *Trans. Evol. Comp* 1.1 (Apr. 1997), pp. 67–82. ISSN: 1089-778X. DOI: 10.1109/4235.585893. URL: https://doi.org/10.1109/4235.585893.

[33] Jason Yosinski et al. "How transferable are features in deep neural networks?" In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani et al. Curran Associates, Inc., 2014, pp. 3320–3328. URL: http://papers.nips.cc/paper/5347-how-transferable-are-features-in-deep-neural-networks.pdf.