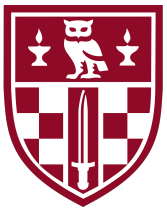
Birkbeck, University of London

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

August 1, 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.

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) [1] 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.

1.1.1 Objectives

Project will be implemented with execution of fallowing objectives:

1. **Carrying out general data exploration:** This part involves general check on dataset.

2. **Data pre-processing and augmentation:** Preparing the data for model ready state.

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

4. **Using pre-trained network to increase model performance:** Using pre-trained networks to help training and accuracy of the model.

5. **Visualizing neural network to ensure learning quality:** For making sure model learning as intended and focusing on correct parts of the image.

6. **Model ensembling:** Using ensemble method with different neural network architectures.

7. **Applying different deployment options:** Implementation of different deployment options. Based on their trade offs.

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.

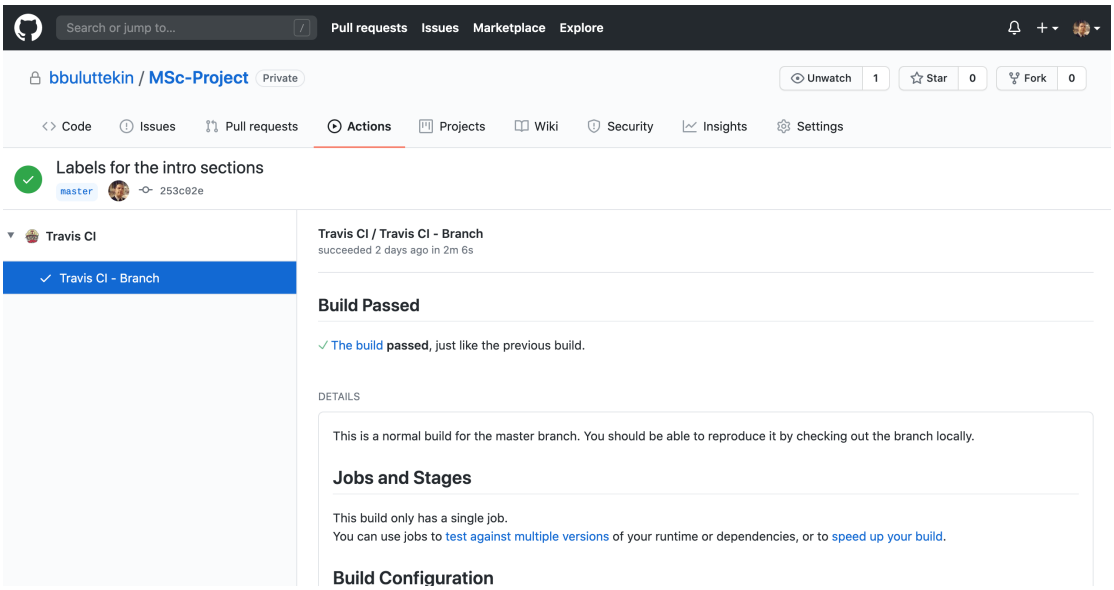


Figure 1.1: CI feedback received from Travis.

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

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

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 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 [7] 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

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

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

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 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 Material

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** [18] 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 [9], VGGNet [15] and ResNet [6] in the ILSVRC [1].

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 [11]. 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"* [13]. 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 [10] (also know as XOR) is discovered. 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*.

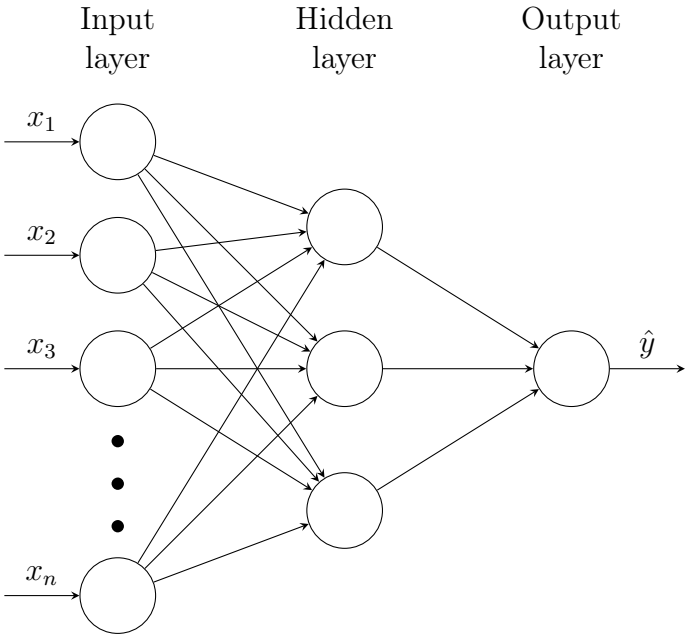


Figure 2.1: Illustration of a MLP model.

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 [14] 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 [4]. 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.

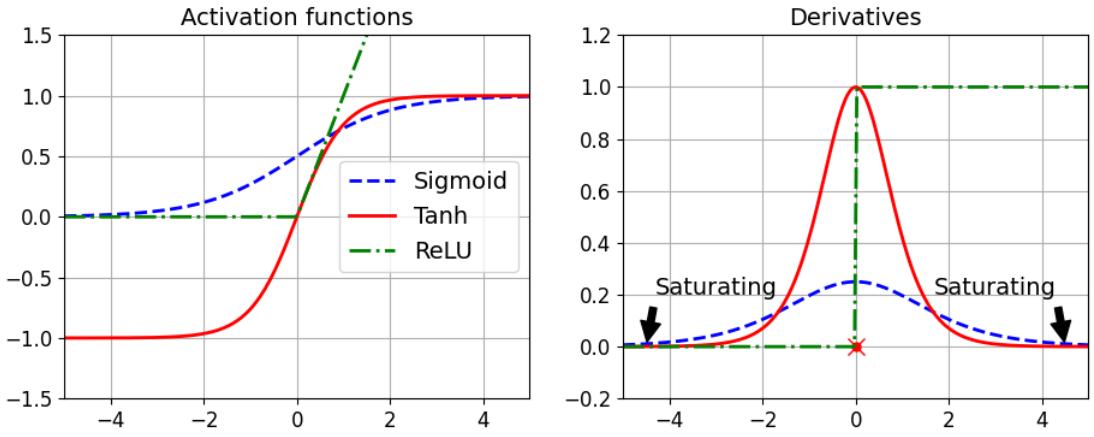


Figure 2.2: Saturation points in activation functions.

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 [3], RMSProp [16], Adam [8] and Nadam [2]. Despite the fact, these adaptive optimizers usually converge to good solutions faster research by Ashia C. Wilson et al [17]. 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 fundamental challenge in machine learning is that our model should perform well on previously unseen data points. During the training of the machine learning model we worked with the training set where we can calculate training loss and performance metrics of the training data. Utilizing the optimization techniques I detailed earlier, we can reduce the loss we initially had. However that would just be optimization, what separate optimization from machine learning is how well the model perform when predicting for the new unseen data points.

2.5 Convolutional Networks

For the definition of Convolutional Neural Networks (CNNs), I used the definition from Deep Learning book [5] in the proposal of this project which I believe makes a very good job of explaining the concept. 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."

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.
- Monitor wights

Another challenge is 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 spend 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.

Role of hardware accelerators. Debugging related challenges of Neural Networks Talk about core modules and how it will fit into general experimentations. Touch on the resilience of the training because of failure in compute environment.

3 Data

Reminder of the dataset. [12] Data set comprises of jpeg files stored in classification based folders.

3.1 Data Augmentation

Data augmentations applied and example results of the augmentation.

3.2 Limitations of the Dataset

Issues related to validation set size. Variance of the image resolution.

3.3 Data Procession

Information about tf.data processing pipeline and advantages compare to other data feeds.

Information about data processing for scikit-learn models. Talk about the decisions for image size and how it relates to information loss and preservation. Briefly mention the high variance of the image sizes and image resizing options and choice. Discussing different representation of the dataset will be covered. Talk about data module design and functionality.

References

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

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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`.

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

[9] 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`.

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

[11] 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`.

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.