UNIVERSITY OF SCIENCE – VNUHCM

FACULTY OF INFORMATION TECHNOLOGY

PROJECT REPORT
KERAS

Introduction to Big Data

Class: 21KHMT1

Group ID: KHMT1-02

Table of contents

1. Introduction

Keras acts as a user-friendly front-end for TensorFlow, simplifying the process of tackling Machine Learning, particularly Deep Learning, challenges. Encompassing the entire workflow, from data preparation to deployment and optimization, Keras empowers rapid experimentation.

Deep learning is a subset of machine learning that loosely mimics the structure and function of the human brain. It uses artificial neural networks with multiple layers to learn complex patterns and relationships from data. These networks are trained on large amounts of data, and as they learn, they become better at recognizing and making predictions on new, unseen data.

In the early 2000s, Keras was developed as a personal project to simplify Deep Learning experimentation, by Francois Chollet. As a result, due to its simplicity, modularity, and cross-platform compatibility, Keras rapidly gained traction within the research community in the 2011-2015 period. In 2015, Keras became an officially supported API within the TensorFlow framework [1]. This integration provided access to TensorFlow's computational power and wider ecosystem, significantly boosting Keras's capabilities and user base. Since then, Keras has continued to evolve with ongoing development and contributions from the community.

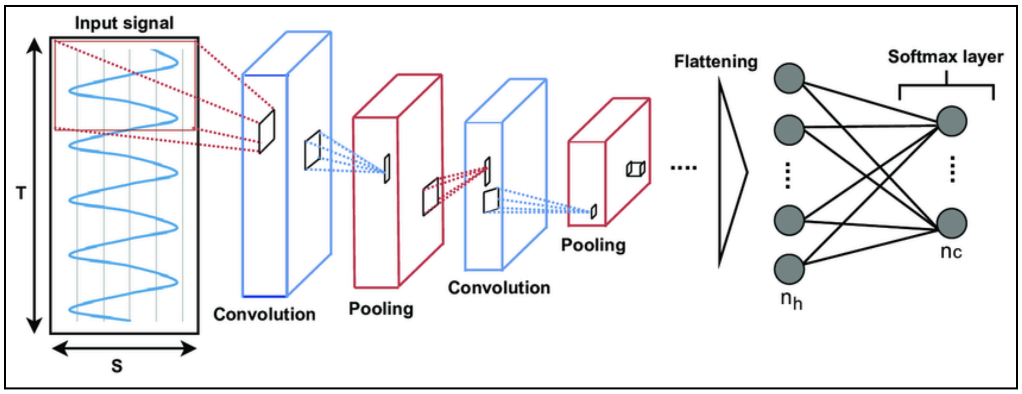
In contemporary times, Keras stands out as a favored option across diverse industries and domains such as automotive engineering, medical diagnostics, virtual assistance, financial modeling, and more. Its adoption brings forth numerous advantages during the developmental phase, including rapid progress, adaptability, and extensive community backing. For a closer examination, let's delve deeply into the impact of this tool on each phase of a deep learning project in the case of big data.

2. Insights

2.1. Components and functionalities

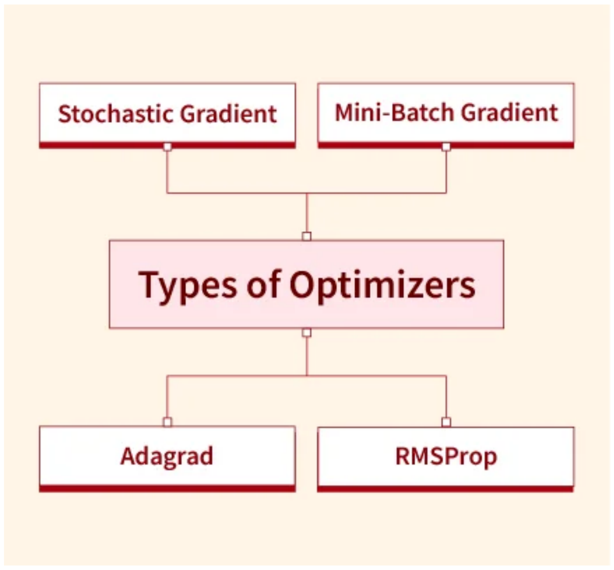
To gain deeper insights into how Keras facilitates numerous decent experiments in Deep Learning, let's begin by examining its primary components:

- **Layers**: These are the fundamental building blocks of neural networks, representing different functionalities such as input, activation, and output. Keras offers a wide variety of pre-built layers for common tasks, allowing us to easily construct complex models.



Example of Keras Layers [2]

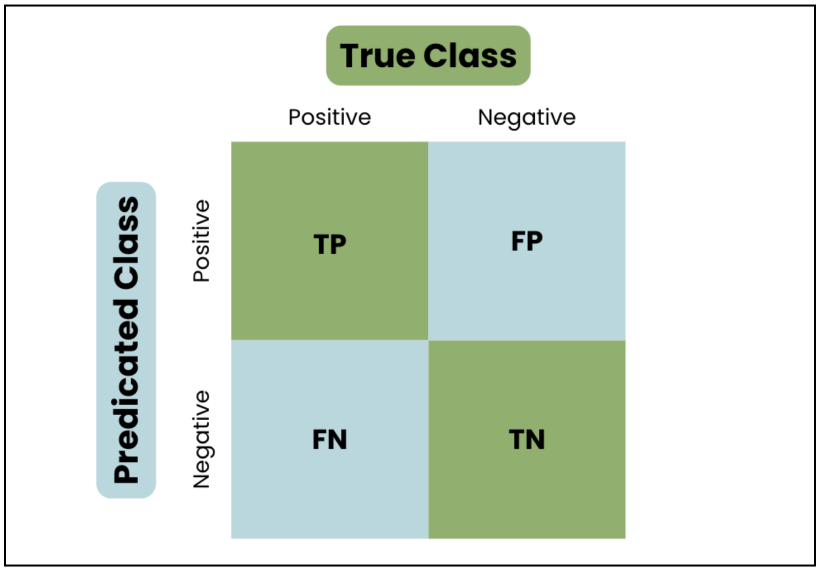
- **Models**: created by arranging and connecting layers in a specific architecture. Keras provides various tools and utilities for defining, compiling, and training models.
- **Optimizers**: These algorithms adjust the weights of the model during training to minimize the loss function and improve performance. Keras offers a selection of optimizers with different learning rates and update rules.



Type of Optimizers [8]

- **Loss functions**: quantify the model's performance on a given task, measured by the difference between predicted and actual values. Keras provides various loss functions depending on the problem type (classification, regression, etc.).

- **Metrics**: These are measures used to evaluate the model's performance during training and validation, like accuracy, precision, recall, or F1 score.



Confusion matrix [9]

Keras offers a range of functionalities that contribute to its popularity and effectiveness in Deep Learning:

- **Modular design**: The component-based approach allows mixing and matching layers, optimizers, and loss functions to create customized models for specific tasks.
- **User-friendly API**: Keras is known for its clean and intuitive API, making it easier to learn and build models compared to lower-level libraries.
- **Cross-platform compatibility**: Keras can run seamlessly on various backends like TensorFlow, PyTorch, or JAX, offering flexibility and leveraging the strengths of different platforms.

- **Extensive community support**: With a vibrant community, Keras offers a wealth of resources, tutorials, and examples, making it easier to get started and find solutions to problems.
- **Scalability**: Keras models can be scaled to handle large datasets and complex tasks efficiently, making them suitable for big data applications.

In addition to its core functionalities, Keras offers several supplementary features that enhance the model development process:

- **Data preprocessing tools**: For transforming and preparing data before feeding it to models.
- **Callbacks**: Allow to monitor and control the training process and perform actions like early stopping or saving checkpoints.
- **Visualization tools**: Help visualize model architecture and understand its behaviour.

2.2. Applications

Academic research:

- Rapid prototyping and experimentation: Keras's simple API allows researchers to quickly build and test various deep learning models on large datasets, accelerating research progress.
- Reproducible research: The modular nature of Keras models facilitates sharing and reproducing research findings, enhancing transparency and collaboration.
- Exploration of diverse data sources: Keras supports a wide range of data formats and backends, enabling researchers to analyze complex and multi-modal datasets.

Industry applications:

- Scalable model development: Keras's cross-platform compatibility allows leveraging distributed computing frameworks like Spark, handling massive datasets efficiently.

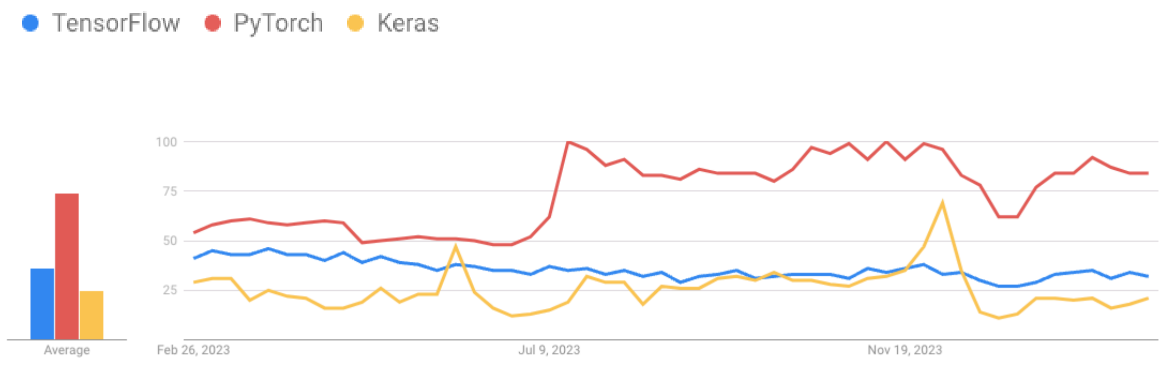
- Faster deployment and adaptation: Keras models can be easily deployed and adapted to production environments, leading to faster time-to-market and improved responsiveness to changing needs.
- Reduced development costs: The user-friendly API and large community support contribute to quicker development cycles and lower overall costs.

Specific examples:

- **Academia**: Analyzing large genomic datasets to identify disease markers, using satellite imagery for climate change research, or developing natural language processing models for analyzing social media data.
- **Industry**: Building recommendation systems for e-commerce platforms, developing fraud detection models for financial institutions, or optimizing production processes in manufacturing using sensor data.

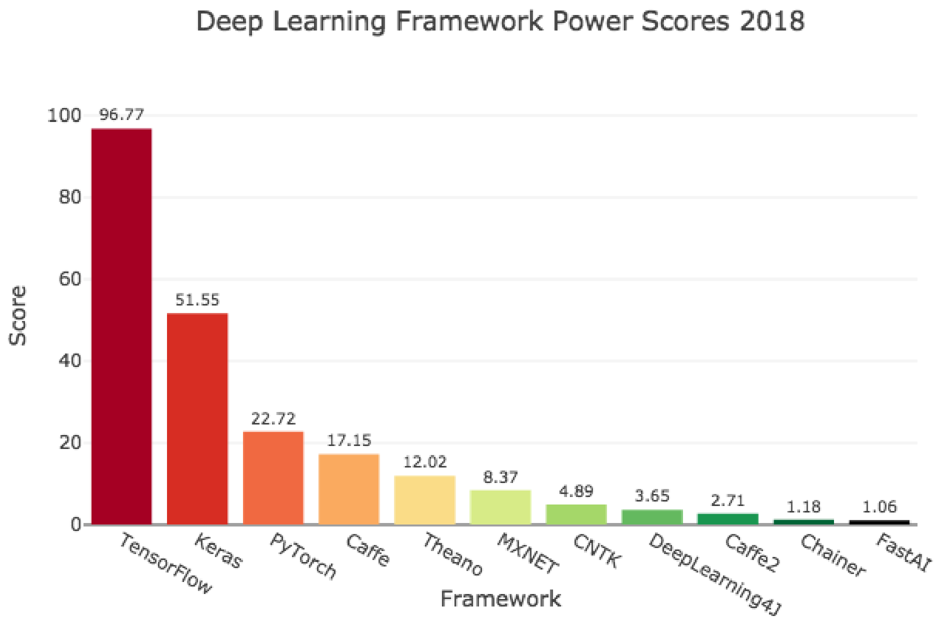
2.3. Popularity and Comparison

2.3.1 Popularity



Keyword searching interest comparison

In the past 12 months, it can be seen that Pytorch is the keyword of most interest among the three, Keras and TensorFlow have fewer searching times but they remain a reasonable amount over time.



Deep Learning Framework comparison [10]

2.3.2 Comparison

PyTorch vs Keras:

- Both PyTorch and Keras serve as excellent choices for beginners in deep learning. PyTorch appeals more to mathematicians and seasoned researchers due to its preferred syntax. On the other hand, Keras suits developers seeking a user-friendly framework for quick model building, training, and evaluation, with robust deployment options and simplified model export. PyTorch outperforms Keras in terms of speed and debugging capabilities. Both platforms offer extensive learning resources: Keras provides reusable code and tutorials, while PyTorch boasts strong community support and ongoing development.
- Keras shines when dealing with small datasets, rapid prototyping, and diverse back-end support, making it the most widely used framework for its simplicity across Linux, MacOS, and Windows platforms.

TensorFlow vs Keras:

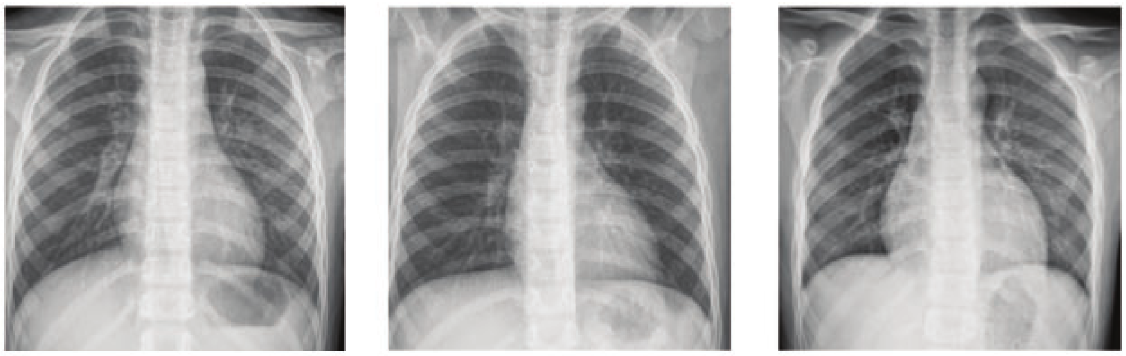
- TensorFlow serves as an open-source platform offering end-to-end solutions and a library catering to multiple machine-learning tasks. Conversely, Keras operates as a high-level neural network library running atop TensorFlow. Both platforms provide high-level APIs for constructing and training models, with Keras being more user-friendly due to its Python integration.
- Researchers typically opt for TensorFlow when handling extensive datasets and tasks like object detection, seeking top-notch functionality and performance. TensorFlow supports various operating systems, including Linux, MacOS, Windows, and Android, and was developed by Google Brain to meet Google's research and production requirements.
- It's crucial to note that directly comparing TensorFlow and Keras may not be the most effective approach, given that Keras acts as a wrapper for TensorFlow's framework. Consequently, users can define models using Keras' user-friendly interface and seamlessly integrate TensorFlow when specific functionalities are required. This flexibility allows users to incorporate TensorFlow code directly into the Keras training pipeline or model as needed.

3. Demonstration

For a more detailed analysis of the tools and functionalities offered by Keras, we will directly investigate a specific image processing task.

3.1. Formulate the problem

Chest X-ray Images: With the advancement of artificial intelligence and the ever-growing wealth of medical data, combining AI with healthcare to provide diagnoses aids in early disease detection. Applying these current conditions to a specific problem, using chest X-rays to predict whether a patient has symptoms of pneumonia is a pertinent example.



Chest X-ray images [11]

The normal chest X-ray (left panel) depicts clear lungs without any areas of abnormal opacification in the image. Bacterial pneumonia (middle) typically exhibits a focal lobar consolidation, in this case in the right upper lobe (white arrows), whereas viral pneumonia (right) manifests with a more diffuse ‘‘interstitial’’ pattern in both lungs.

3.2. Prepare the dataset

Chest X-Ray Images (Pneumonia): The dataset is organized into 3 folders (train, test, val) and contains subfolders for each image category (Pneumonia/Normal). There are 5,863 X-Ray images (JPEG) and 2 categories (Pneumonia/Normal).

After obtaining the train, test, and validation sets from three folders (train, test, val) on Kaggle, the following preprocessing steps are applied:

- **Data Normalization**: The pixel values of the images are normalized to a range between 0 and 1. This normalization is achieved by dividing each pixel value by 255. Normalization simplifies model training by ensuring consistent data ranges.
- **Reshaping for Deep Learning**: Images are reshaped into a 4D array with dimensions (batch_size, img_size, img_size, 1). Here, batch_size represents the number of images, img_size is the desired image size, and 1 signifies the grayscale channel.
- These preprocessing steps are crucial for effectively training convolutional neural networks (CNNs) for image classification tasks. Once completed, the

prepared data is ready for model training and evaluation, facilitating the development of accurate and robust image classification systems.

3.3. Build the model

3.3.1. Build the model with Keras

Image Augmentation using Keras **ImageDataGenerator** [3]:

- These image augmentation techniques not only expand the size of the dataset but also incorporate variation in the dataset which allows the model to generalize better on unseen data. Also, the model becomes more robust when it is trained on new, slightly altered images.
- Keras ImageDataGenerator class provides a quick and easy way to augment images. It provides a host of different augmentation techniques like standardization, rotation, shifts, flips, brightness change, and many more.
- ImageDataGenerator class ensures that the model receives new variations of the images at each epoch. However, it only returns the transformed images and does not transform the original corpus of images. If this was the case, then the model would be seeing the original images multiple times which would definitely overfit the model.
- Another advantage of ImageDataGenerator is that it requires lower memory usage. This is so because without using this class, we load all the images at once. But by using it, we are loading the images in batches which saves a lot of memory.

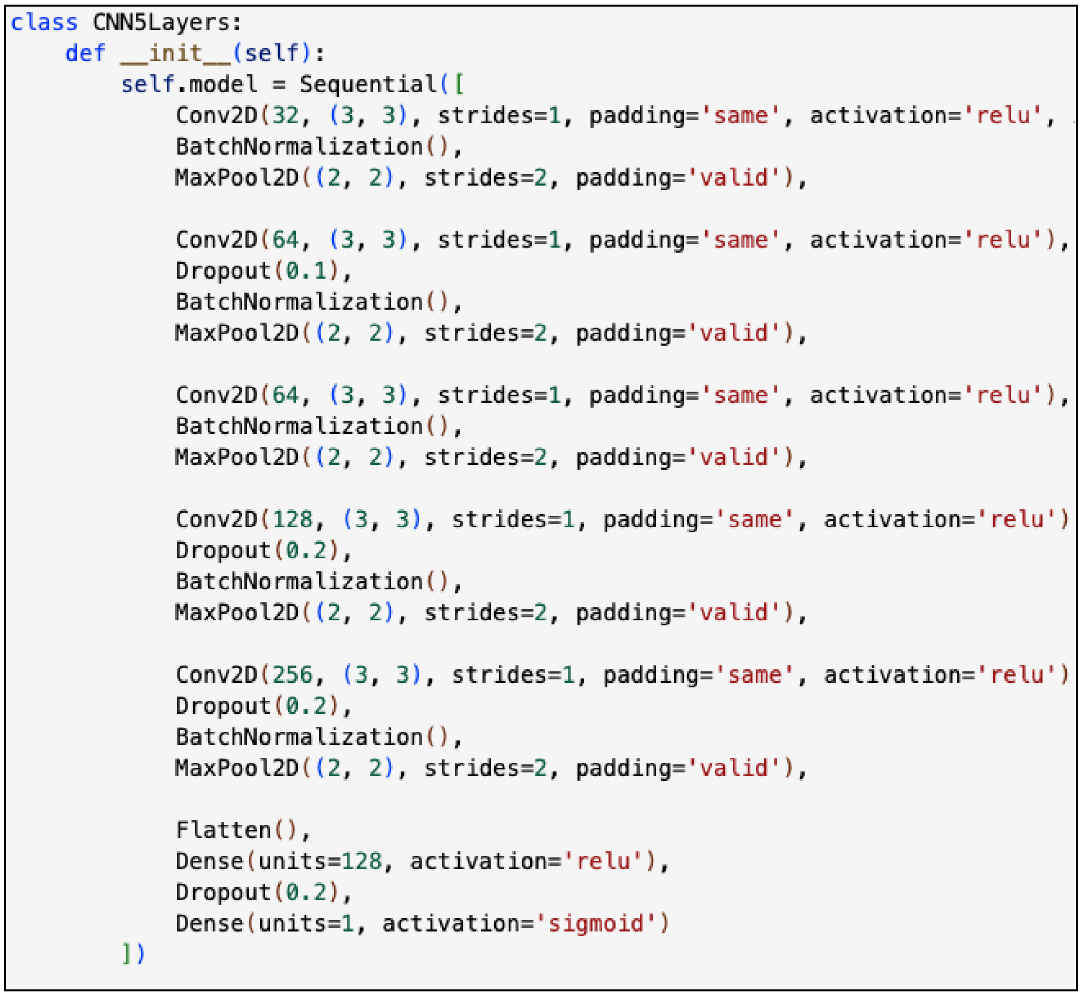
Build model [4]:

- **Layers** :
  - o Convolutional Layers: 5
  - o MaxPooling2D Layers: 5
  - o BatchNormalization Layers: 4
  - o Dropout Layers: 3
  - o Flatten Layer: 1

- Dense Layers: 2
- Input Layer: 1

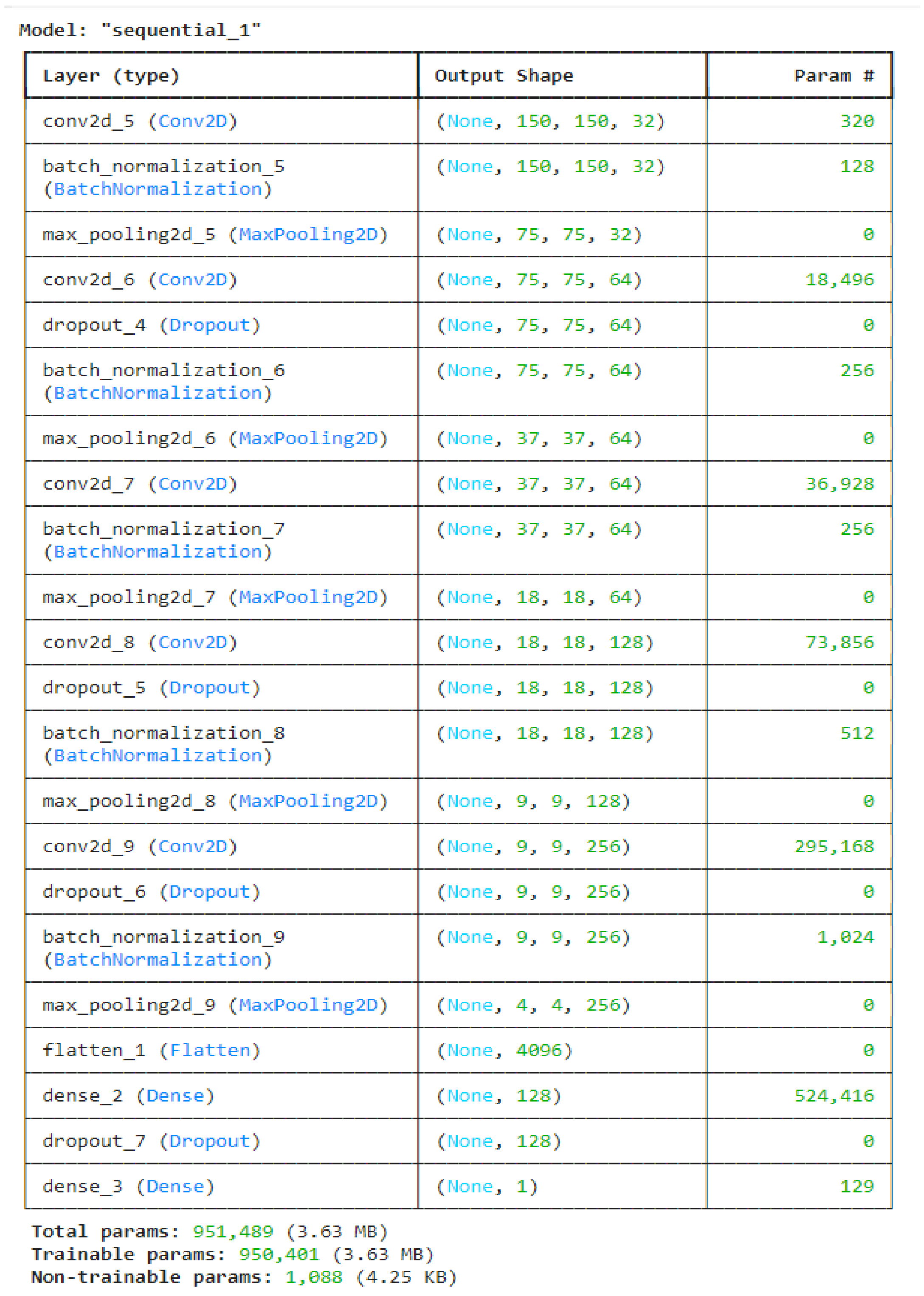
Total layers: 21 layers.

- **Configuration**:



```
class CNN5Layers:
    def __init__(self):
        self.model = Sequential([
            Conv2D(32, (3, 3), strides=1, padding='same', activation='relu', 
            BatchNormalization(),
            MaxPool2D((2, 2), strides=2, padding='valid'),

            Conv2D(64, (3, 3), strides=1, padding='same', activation='relu'),
            Dropout(0.1),
            BatchNormalization(),
            MaxPool2D((2, 2), strides=2, padding='valid'),

            Conv2D(64, (3, 3), strides=1, padding='same', activation='relu'),
            BatchNormalization(),
            MaxPool2D((2, 2), strides=2, padding='valid'),

            Conv2D(128, (3, 3), strides=1, padding='same', activation='relu')
            Dropout(0.2),
            BatchNormalization(),
            MaxPool2D((2, 2), strides=2, padding='valid'),

            Conv2D(256, (3, 3), strides=1, padding='same', activation='relu')
            Dropout(0.2),
            BatchNormalization(),
            MaxPool2D((2, 2), strides=2, padding='valid'),

            Flatten(),
            Dense(units=128, activation='relu'),
            Dropout(0.2),
            Dense(units=1, activation='sigmoid')
        ])
```

- **Optimizer**: Adam optimizer with a learning rate of 0.001 is used for training the model.



```
Model: "sequential_1"
┌─────────────────────────────────┬────────────────────────┬───────────────┐
│ Layer (type)                    │ Output Shape           │       Param # │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_5 (Conv2D)               │ (None, 150, 150, 32)   │           320 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ batch_normalization_5           │ (None, 150, 150, 32)   │           128 │
│ (BatchNormalization)            │                        │               │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_5 (MaxPooling2D)  │ (None, 75, 75, 32)     │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_6 (Conv2D)               │ (None, 75, 75, 64)     │        18,496 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_4 (Dropout)             │ (None, 75, 75, 64)     │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ batch_normalization_6           │ (None, 75, 75, 64)     │           256 │
│ (BatchNormalization)            │                        │               │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_6 (MaxPooling2D)  │ (None, 37, 37, 64)     │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_7 (Conv2D)               │ (None, 37, 37, 64)     │        36,928 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ batch_normalization_7           │ (None, 37, 37, 64)     │           256 │
│ (BatchNormalization)            │                        │               │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_7 (MaxPooling2D)  │ (None, 18, 18, 64)     │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_8 (Conv2D)               │ (None, 18, 18, 128)    │        73,856 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_5 (Dropout)             │ (None, 18, 18, 128)    │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ batch_normalization_8           │ (None, 18, 18, 128)    │           512 │
│ (BatchNormalization)            │                        │               │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_8 (MaxPooling2D)  │ (None, 9, 9, 128)      │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ conv2d_9 (Conv2D)               │ (None, 9, 9, 256)      │       295,168 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_6 (Dropout)             │ (None, 9, 9, 256)      │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ batch_normalization_9           │ (None, 9, 9, 256)      │         1,024 │
│ (BatchNormalization)            │                        │               │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling2d_9 (MaxPooling2D)  │ (None, 4, 4, 256)      │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ flatten_1 (Flatten)             │ (None, 4096)           │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_2 (Dense)                 │ (None, 128)            │       524,416 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_7 (Dropout)             │ (None, 128)            │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_3 (Dense)                 │ (None, 1)              │           129 │
└─────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 951,489 (3.63 MB)
 Trainable params: 950,401 (3.63 MB)
 Non-trainable params: 1,088 (4.25 KB)
```

3.3.2. Build the model with PyTorch

Additional Data Preprocessing

- In PyTorch, we have to create a custom dataset [7]. In the original dataset, there are three folders: train, test, val. Each folder contains two subfolders (PNEUMONIA and NORMAL). This style of organizing input data is convenient because we can decide the label of an image based on the folder containing it. Therefore, there is no need for additional annotation files.
- However, this organizing structure is not suitable for creating a custom dataset in PyTorch. Specifically, PyTorch requires two things to create a custom dataset: a directory containing images, and a .csv file storing annotations. Therefore, we must convert each input folder (e.g. train) from a nested directory into a directory containing only images, and a .csv file to store annotations for these images.

Data Augmentation

- The motivation to perform data augmentation (help the model generalize well and avoid overfitting) and the augmentation techniques (resizing, rotating, flipping, shifting, etc.) are almost the same as in Keras. An implementation detail worth mentioning here is that PyTorch does not have the ImageDataGenerator class. In Keras, the ImageDataGenerator can be fitted to data and automatically compute the internal data stats (e.g. mean and standard deviation) required for transformations. However, in PyTorch, we have to compute the stats on our own and in advance in order to perform some transformations (e.g. normalization). Specifically, after obtaining the necessary stats, we use the torchvision.transforms module [5] from the TorchVision library within PyTorch to perform transformations.
- We have to tackle the issue of imbalanced datasets. Specifically, the number of samples from class PNEUMONIA (3875) is almost three times larger than the number of samples from class NORMAL (1341). Therefore, we need data augmentation to enrich the samples, especially the samples from class NORMAL, and we want to balance the data during training. A technique we employ is **Weighted Random Sampler** [6], which basically assigns a probability of being chosen to each sample, so that the sample from the

popular class will have a lower probability of being selected, and the sample from the minority class will have a higher probability of being chosen.

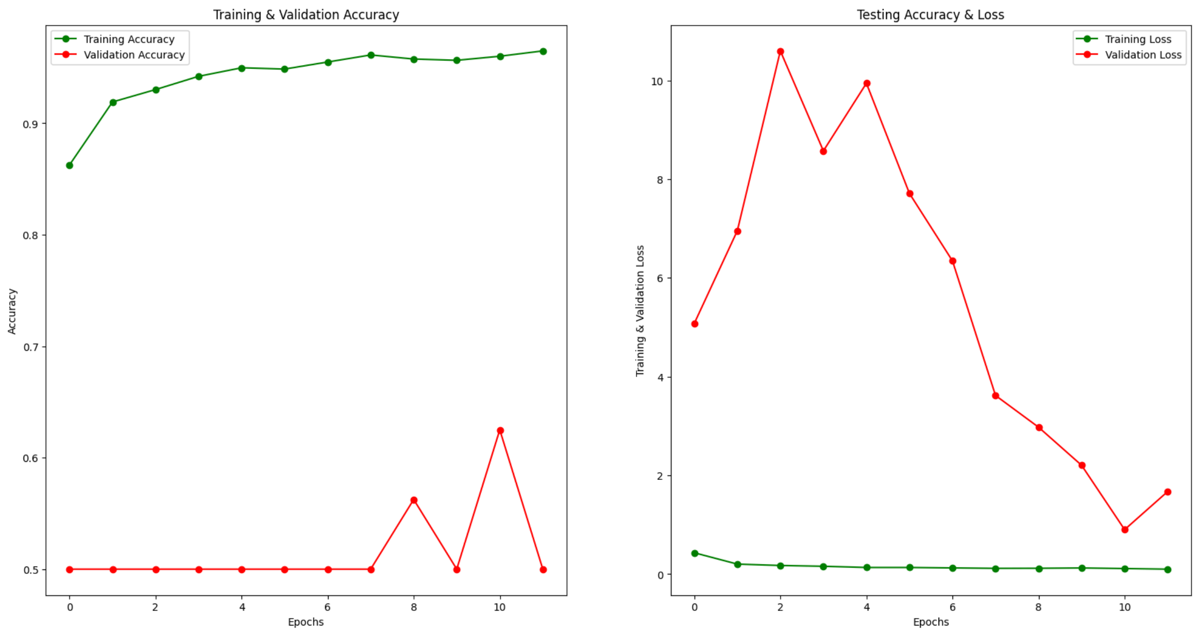
Model

- The model architecture and the number of parameters are exactly the same as in Keras. One thing to notice is that in PyTorch, we need to define the model architecture and write the forward() method, and the training script in more detail.

3.4. Model evaluation

3.4.1 Model evaluation with Keras:

Accuracy & Loss:

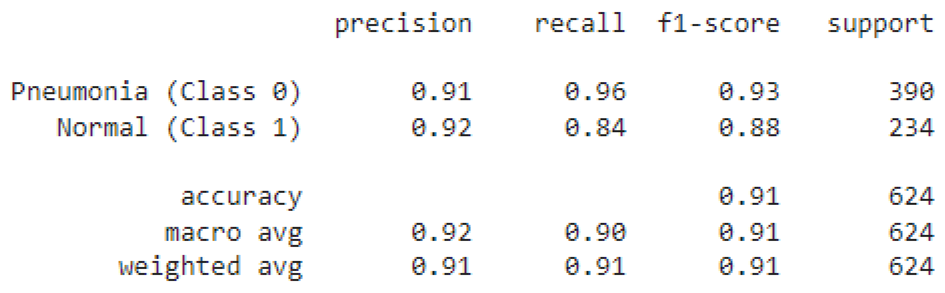


- **Initial Performance**: The training process starts with relatively low accuracy (around 80%) and high loss (around 0.85), which is typical as the model begins to learn patterns from the data.
- **Validation Performance**: Throughout the training process, the validation accuracy remains stagnant at 50%, and the validation loss fluctuates, indicating potential issues with model generalization. This suggests that

the model is likely overfitting to the training data, as it performs well on the training set but fails to generalize to unseen data.

- **Final Epoch**: In the final epoch, the training accuracy improves to 96.43%, and the training loss decreases to 0.1055. However, the validation accuracy remains at 50%, and the validation loss increases to 1.6719. This further indicates overfitting, as the model's performance on the training data continues to improve while its performance on the validation data deteriorates.

Classification Report:



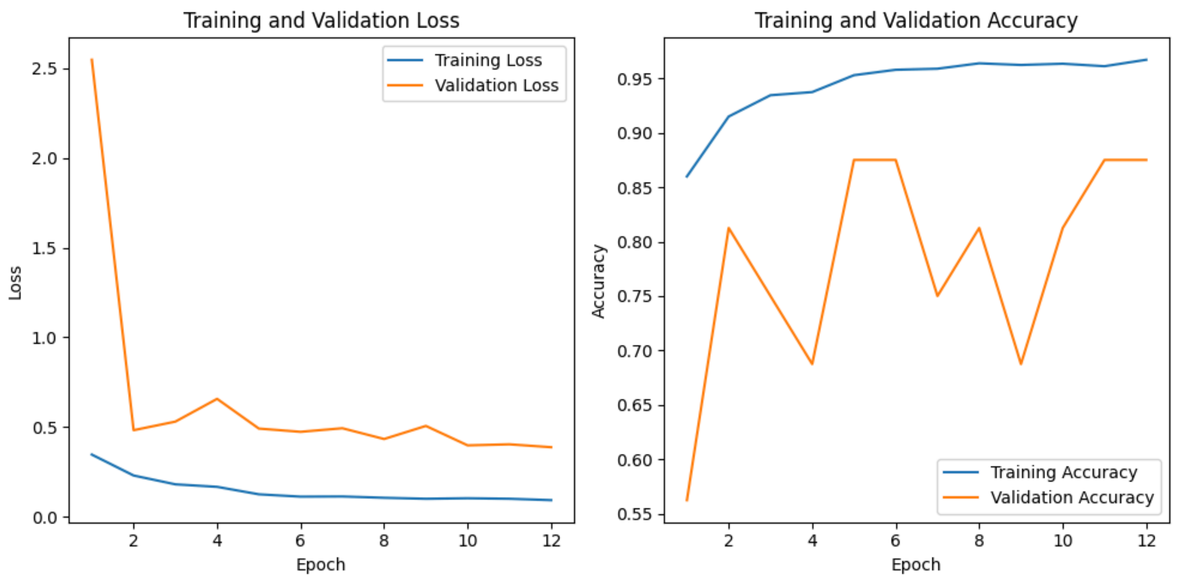
```
                    precision    recall  f1-score   support

Pneumonia (Class 0)       0.91      0.96      0.93       390
   Normal (Class 1)       0.92      0.84      0.88       234

           accuracy                           0.91       624
          macro avg       0.92      0.90      0.91       624
       weighted avg       0.91      0.91      0.91       624
```

- **High Overall Performance**: The model shows strong performance overall with an accuracy of 91%. This suggests that the model is effective at distinguishing between pneumonia and normal cases.
- **Good Precision and Recall**: The precision and recall values for both classes are relatively high, indicating that the model is making accurate predictions and effectively identifying instances of pneumonia and normal cases. However, there is a slight difference in recall between the two classes, with normal cases having lower recall compared to pneumonia.
- **F1-Score**: The F1-score, which combines precision and recall into a single metric, is high for both classes, indicating a good balance between precision and recall. This suggests that the model is performing well in terms of both identifying positive cases (pneumonia) and negative cases (normal).

Although the validation accuracy is quite low in all epochs (because the validation set provided is relatively small, only about 16 samples), the accuracy on the test set reaches 91%, a result much better than the validation accuracy.

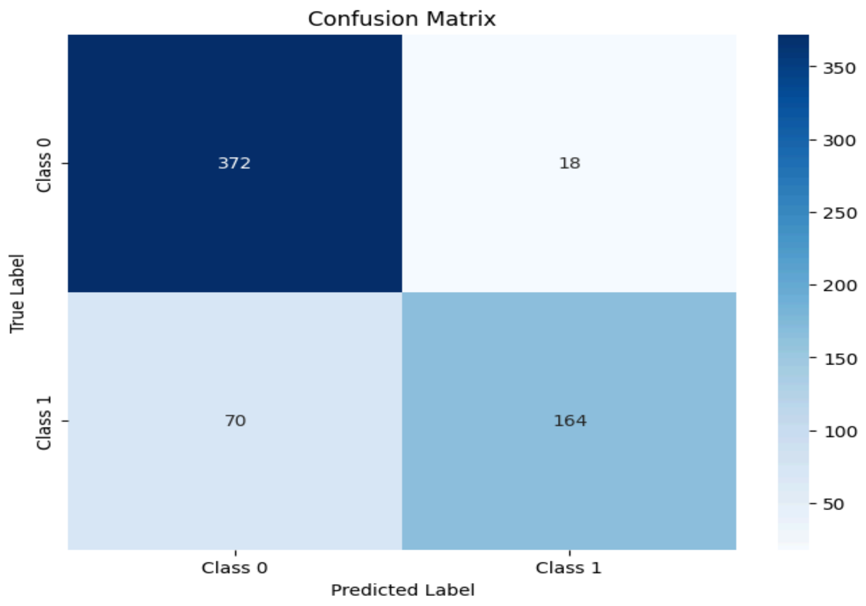
3.4.1 Model evaluation with Pytorch:

Accuracy & Loss:



- **Initial Performance**: The training process starts with relatively high training accuracy (over 80%) and low training loss (below 0.5), which demonstrates the ability of the model to learn from data quickly.
- **Validation Performance**: Throughout the training process, both the validation accuracy and validation loss fluctuate substantially. However, the validation accuracy follows an overall upward trajectory, while the validation loss follows an overall downward trajectory. This suggests that the model can improve its performance over the training process.
- **Final Epoch**: In the final epoch, the training accuracy improves to over 95%, and the training loss decreases to around 0.1. On the other hand, the validation accuracy increases to over 85%, and the validation loss drops to below 0.5.

Confusion Matrix:



- The model may have a tendency to miss some instances of class 1.
- One possible reason may be due to the imbalanced dataset, in which the number of examples from class 1 is only one-third compared to the number of examples from class 0.

3.5. Comparison, observations

- **Keras:**
  - Keras provides very intuitive classes and functions, from data processing to model training and evaluation.
  - Specifically, Keras code is simple and concise, but still able to build very good AI systems.
  - Therefore, it may be a very good framework to build fast demos or prototypes.
  - Keras can also be used to build industrial AI systems, since it requires minimal code but provides maximal functionalities.
- **PyTorch:**
  - PyTorch is faster than Keras.
  - Specifically, PyTorch works better with larger datasets and yields higher performance.

    - However, PyTorch code is more complex and requires an understanding of data structure and the training process.
    - In return, PyTorch offers greater flexibility and allows developers to gain more insights into the entire workflow.

4. Discussions and Conclusion

- Our team are complete beginners to Keras and PyTorch. After this project, we have an impression that Keras implementation is fast and convenient, but PyTorch is definitely worth a try.
- Specifically, the time we spend on PyTorch is roughly three times as much as the time we spend on Keras. At some points, we even thought we would give up on the PyTorch version.
- After successfully implementing PyTorch, we realize that PyTorch offers so many insights that Keras hides from us, although we literally implement the same idea using both frameworks.
- It does not mean we learn nothing from Keras. In fact, Keras helped us build the first successful model, and that is a very good foundation to start implementing the PyTorch version.
- After all, we are grateful for this opportunity to learn and get familiar with Keras and PyTorch. We hope we can work in Machine Learning and Big Data later on in our careers, and Keras and PyTorch shall be our dears along the journey.

References

[1] Tensor flow basis (link)
[2] Keras Layers example (link)
[3] Image Augmentation using Keras ImageDataGenerator (link)
[4] Keras: Fast Neural Network Experimentation (link)
[5] PyTorch: Transforming and augmenting images (link)
[6] TORCH.UTILS.DATA (link)
[7] PyTorch: Writing Custom Datasets, DataLoaders and Transforms (link)
[8] Type of Optimizers (link)
[9] Confusion Matrix (link)
[10] Deep Learning Framework comparison (link)
[11] Chest X-ray images (link)