Springboard Data Science Capstone Project 2

CNN For Detecting Pneumonia from X-ray Images

Meral Balik

April 28, 2020

Contents

1. Problem Statement **3**

2. Dataset **3**

2.1. Data Acquisition using Google Colab 4

2.2. Data Inspection and Visualization 4

2.3. Data Preprocessing 5

3. Application of CNN **6**

3.1. Baseline Model 6

3.2. Compiling The Model 8

3.3. Training The Model 8

3.4. Evaluating Accuracy and Loss for the Model 9

3.5. Model Performance 10

4. Conclusion **11**

Capstone Project 2 - Milestone Report

CNN For Detecting Pneumonia from X-ray Images

1. Problem Statement

Pneumonia is a common infectious disease in the world. Globally, 450 million get infected by pneumonia in a year and 4 million people die from the disease. 1 million people each year have to seek care from hospitals and 50 thousand people die from the disease [1] in the United States of America. The numerical difference between the infection rates and death rates shows how crucial the early diagnosis of the disease is. Its main diagnostic method is chest x-ray examination. Analyzing and classifying chest x-rays can be very tedious for radiologists since x-rays are often affected by noise and require domain expertise and experience. Recently, a number of researchers have proposed different artificial intelligence (AI)-based solutions for different medical problems. Although currently, deep learning still cannot replace doctors/clinicians in medical diagnosis, it can provide support for experts in the medical domain in performing time-consuming works, such as examining chest radiographs for the signs of pneumonia.

2. Dataset

Data: https://www.kaggle.com/paultimothymooney/chest-xray-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).

I developed a Convolutional Neural Network from scratch to classify the medical images. The CNNs implemented in Python on Google Colab using the Keras interface to Tensorflow. General workflow for the model will be:

- Getting Google Colab ready to use
- Building training and validation image data generators in Keras
- Compiling the model
- Running the model and plotting training and validation accuracy scores over each epoch
- Optimizing the input parameters such as number of CNN layers and callbacks
- Applying regularization techniques such as dropout layers, batch normalization, data augmentation
- Evaluating the model on the hold-out dataset

2.1. Data Acquisition using Google Colab

I used Google Colab for this project knowing that it provides free GPU. The dataset was readily available on Kaggle. I uploaded the data to my personal Google drive and mounted my drive on Colab. The base directory contains train, test and validation subdirectories for the training, testing and validation datasets, which in turn each contain normal and pneumonia subdirectories. Then I assigned variables with the proper file path for the training, validation and test sets and also assigned variables with the proper file path for the normal and pneumonia images.

2.2. Data Inspection and Visualization

After assigning variables to the subdirectories of the dataset, I found out the total number of normal and pneumonia images in each directory:

```
Number of images in train_normal_dir: 1354
Number of images in train_pneumonia_dir: 3875
Number of images in val_normal_dir: 8
Number of images in val_pneumonia_dir: 8
Number of images in test_normal_dir: 234
Number of images in test_pneumonia_dir: 390
```

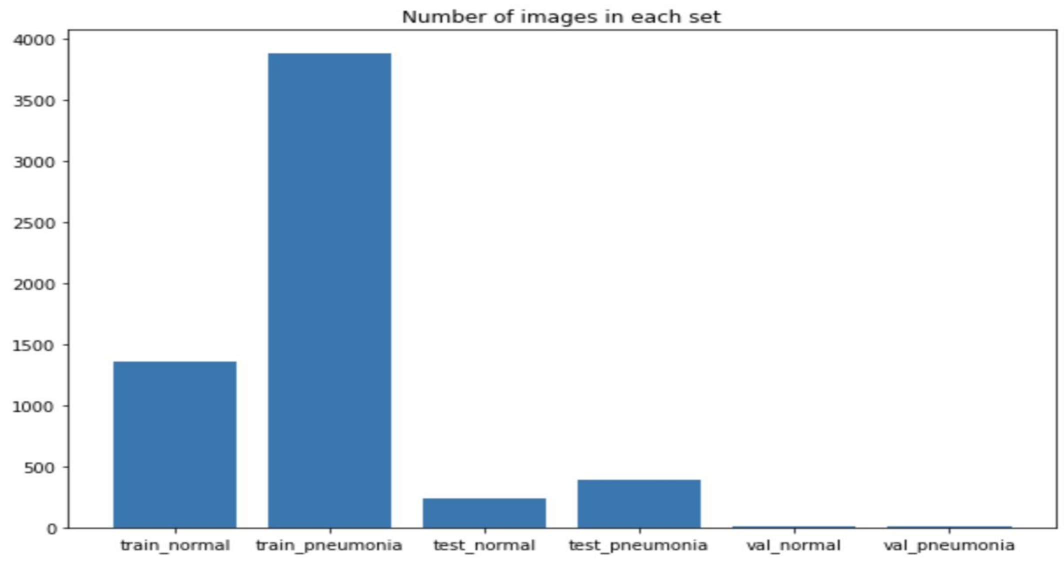

Figure 1. *Number of images in each set*

Figure 1 illustrates that the number of images per class is not equally distributed; the number of normal images is relatively fewer than the number of pneumonia images in train set.

Next I displayed a few images to get a better sense of what the normal and pneumonia datasets look like.

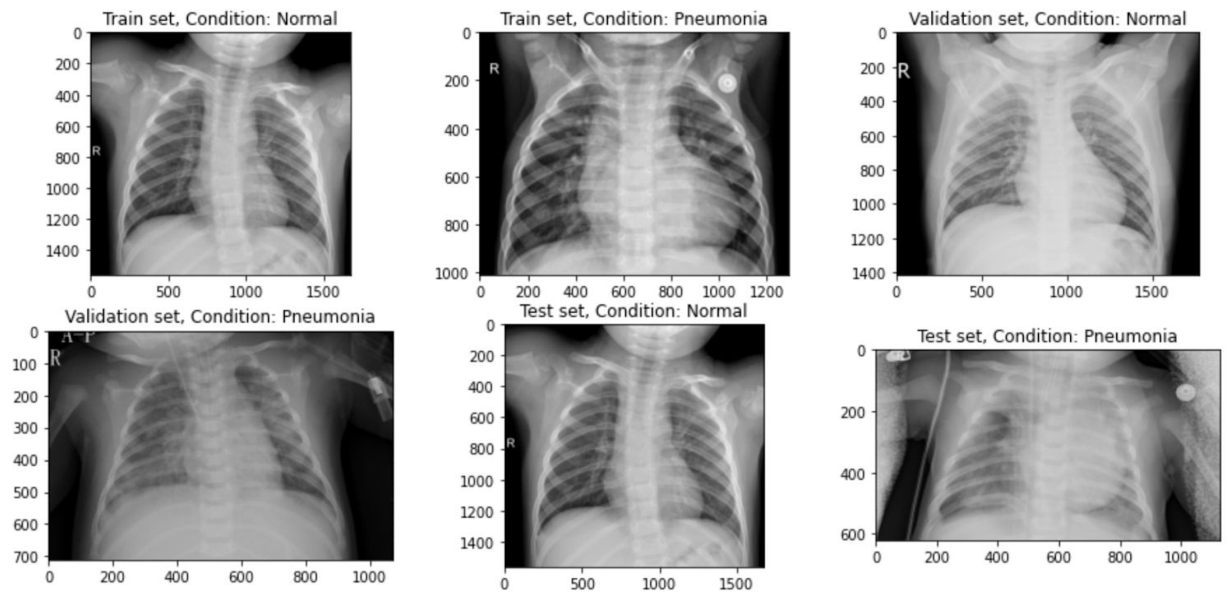
Train set, Condition: Normal
Train set, Condition: Pneumonia
Validation set, Condition: Normal
Validation set, Condition: Pneumonia
Test set, Condition: Normal
Test set, Condition: Pneumonia

Figure 2. *Sample images*

Figure 2 illustrates that the x-ray images come in different sizes. These images need to be resized to the same aspect ratio before feeding them into the neural network.

2.3. Data Preprocessing

In this section, I used ImageDataGenerator class provided by tf.keras which can read images from disk and convert them to float32 tensors, and feed them (with their labels) to the network. It also sets up generators that are capable of loading the required amount of data (a mini batch of images) directly from the source folder, converting them into training data (fed to the model) and training targets. I set up the batch size as 32. I defined three data generators: one for training data, one for testing data and the other for validation data.

Data that goes into neural networks should usually be normalized in some way to make it more amenable to processing by the network. I preprocessed the images by normalizing the pixel values to be in the [0, 1] range (originally all values are in the [0, 255] range). In Keras this can be done via the keras.preprocessing.image.ImageDataGenerator class using the rescale parameter. I set the resize parameter as rescale = 1. / 255.

After defining the generators for training and validation images, the flow_from_directory method loads images from the disk, applies rescaling, and resizes the images into the required dimensions. The generators will yield batches of 32 images of size 150x150 and their labels (binary).

I employed some data augmentation methods to artificially increase the size and quality of the dataset. Augmenting the training examples allow the network to see more diversified, but still representative, data points during training. I set the zoom_range to 0.3 and vertical_flip to True. The zoom range randomly zooms the images to the ratio of 0.3 percent, and when verical_flip is set to True, the images are flipped vertically. I also tried a couple of variations of these parameters such as setting the horizontal_ flip to True or rotation degree to 40.

3. Application of CNN

3.1. Baseline Model

The convolutional neural network (CNN) is a deep learning algorithm commonly used in computer vision applications. I deployed Keras open-source deep learning framework with tensorflow backend to build and train the convolutional neural network model from scratch. To build the model, I used a series of convolutional layers, max-pooling and batch-normalization. On top of it, I used a flatten layer and followed it by four fully connected layers. Also in between I have used dropouts to reduce over-fitting. Activation function was rectified linear unit (relu) throughout except for the last layer where it was sigmoid as this is a binary classification problem. I have used Adam as the optimizer and binary cross-entropy as the loss function.

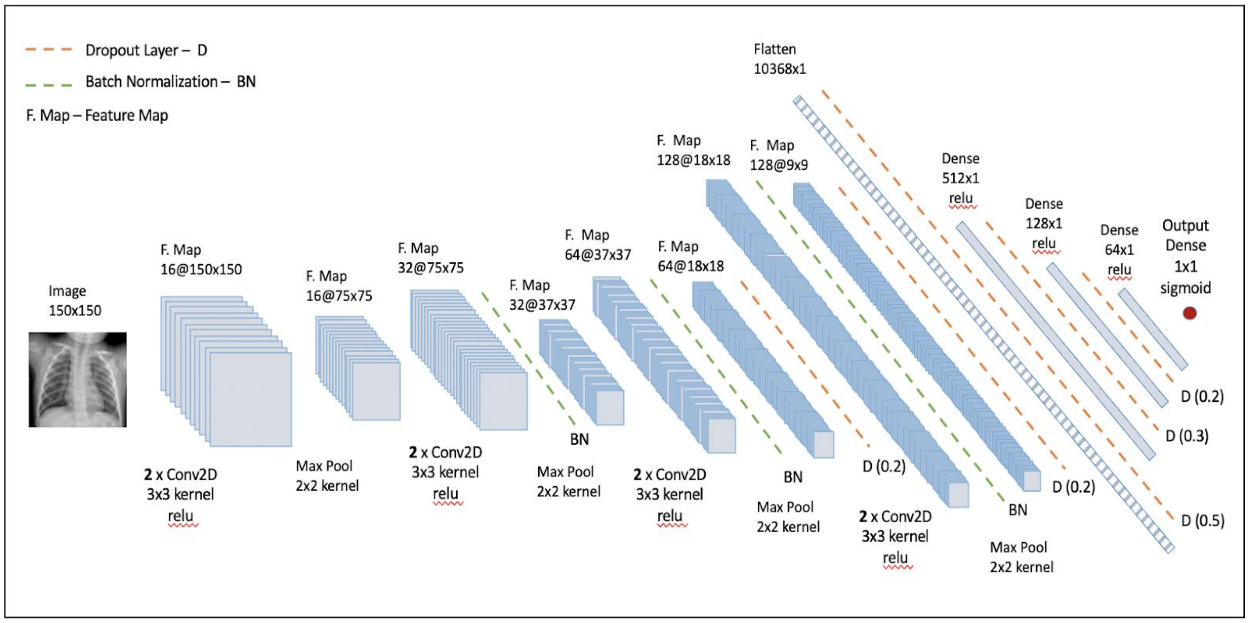

Figure 3. *Baseline Model*

Figure 3 shows the overall architecture of the proposed CNN model which consists of two major parts: the feature extractors and a classifier (sigmoid activation function). Each layer in the feature extraction layer takes its immediate preceding layer's output as input, and its output is passed as an input to the succeeding layers. The proposed architecture in Figure 3 consists of the convolution, max-pooling, and classification layers combined together. The feature extractors comprise 2 × Conv2D, 16; 2 × Conv2D, 32; 2 × Conv2D, 64; 2 × Conv2D, 128, max-pooling layer of size 2 × 2, and a relu activator between them.

The first convolution layer applies 16 convolutions (with a 3 × 3 kernel) to the input image followed by the relu activation function to introduce non-linearity into the model. The result is a feature map with dimensions 150 × 150 ×16. This convolution layer is followed by a max-pooling layer which downsamples the model by a factor of 2 which yields a feature map with dimensions 75 × 75 × 16. This downsampling allows each convolution stage to learn patterns at a different scale.

For the following feature maps, I also used batch-normalization in between the convolution layers and the max-pooling layers. Batch normalization has the effect of dramatically accelerating the training process of a neural network, and in some cases improves the performance of the model via a modest regularization effect. The output of the convolution and max-pooling operations with batch normalizations are 75 × 75 × 32, 37 × 37 × 64 and 18 × 18 × 128 sizes of feature maps, respectively, for the convolution operations and 37× 37 × 32, 18 × 18 × 64 and 9 × 9 × 128 sizes of feature maps from the pooling operations.

The classifier is placed at the far end of the convolutional neural network (CNN) model. This classifier requires individual features (vectors) to perform computations. The output of the feature extractor (CNN part) is converted into a 1D feature vector for the classifiers. This process is known as flattening where the output of the convolution operation is flattened to generate one lengthy feature vector for the dense layer to utilize in its final classification process. The classification layer contains a flattened layer (10368 × 1) , three dropout layers of size 0.5, 0.3 and 0.2 and four dense layers of size 512, 128, 64 and 1, respectively, a relu between the dense layers and a sigmoid activation function that classifies the images as normal or pneumonia. The dropout layer randomly zeros out of the output features of the prior layer during training, and is used as a regularization tool to prevent overfitting.

This model has a total of 5,677,329 parameters; 5,676,881 of them are trainable and 448 of them are non-trainable. Approximately 294,000 of these parameters are the weights to be trained in the convolution filters and the bias values to be applied after the convolutions. The majority of parameters are the weights of the first fully-connected dense layer of the classifier.

3.2. Compiling The Model

Compiling a model in Keras involves selecting a loss function, optimizer function and output metrics to be reported during training. I trained the model with the binary_crossentropy loss, because it's a binary classification problem and the final activation is a sigmoid function. Cross-entropy calculates a score that summarizes the average difference between the actual and predicted probability distributions. I used Adam as the optimizer, it automatically adapts the learning rate during training. Since this is a classification problem, I had Keras report on the accuracy metric.

3.3. Training The Model

Before training the model, I used Keras callback functions to get a view on internal states and statistics of the model during training. These functions help to visualize how the model's training is going, and help prevent overfitting by implementing early stopping or customizing the learning rate on each iteration.

The first callback function that I applied was ModelCheckpoint. ModelCheckpoint callback was used in conjunction with training using model.fit() to save a model or weights at some interval, so the model or weights can be loaded later to continue the training from the state saved. I assigned the file path to save the model and set the 'save_best_only', 'save_weights_only' parameters equal to True.

The second callback function that I applied was ReduceLROnPlateau which monitors a quantity and if no improvement is seen for a 'patience' number of epochs, the learning rate is reduced. Without using this callback, validation accuracy did not show any improvement at a couple of training sessions. So I applied the ReduceLROnPlateau callback and set the 'monitor' parameter equal to ‘val_loss' and ‘mode’ parameter equal to ‘max’ so that mode learning rate would be reduced when the validation accuracy has stopped increasing.

Finally, I used EarlyStopping callback to prevent overtraining of the model by terminating the training process if it’s not really learning anything. I monitored the ‘val_loss’ parameter, set the patience parameter equal to 5 that is the number of epochs with no improvement after which training will be stopped.

Next, the baseline model was fit using the fit_generator method applied to the training and testing generators. The steps_per_epoch value was set to the total number of training data (5229) divided by the batch size (32) which yielded 163 steps per epoch and 20 for testing data. This process ensures that the full dataset would be used for training over each epoch. The number of epochs was set to 20 with the callback functions enabled.

3.4. Evaluating Accuracy and Loss for the Model

The accuracy and loss values for the training and testing datasets were recorded over the course of training. Figure 4 illustrates the training and validation accuracy vs epoch and training and validation loss vs epoch. The training accuracy (in blue) gets close to 93% while the validation accuracy (in orange) gets close to 89% which is an indicator of slight overfitting. The validation loss reaches its minimum after only four epochs. Training process stops after 10 epochs since the validation loss stabilized before reaching to 20 epochs.

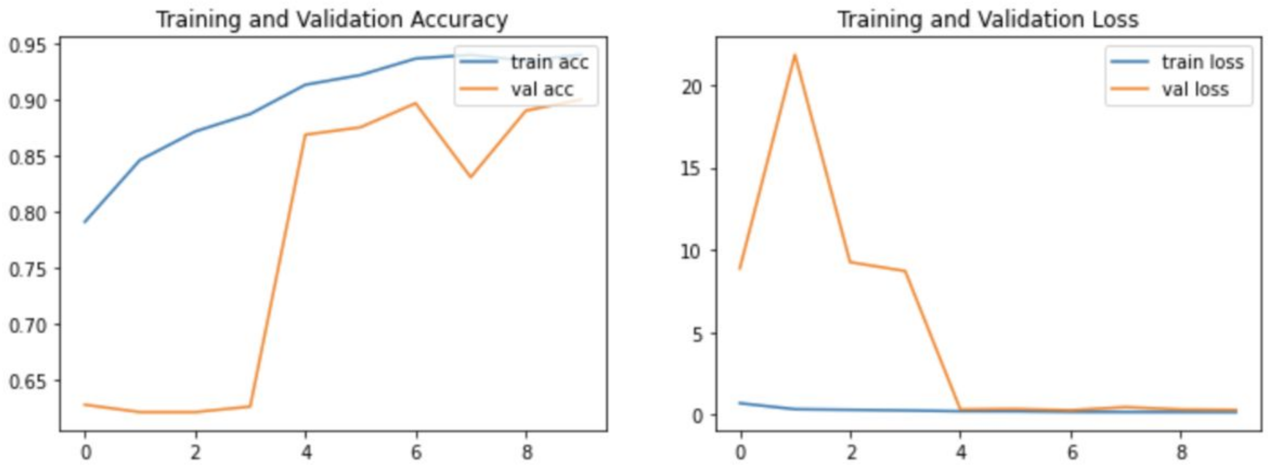

Figure 4. *Training History Plots*

.

3.5. Model Performance

The model predicted the image classes on the test data with a 90% accuracy and 0.29 loss. The test data had 624 images, 234 of them belonged to the ‘normal’ class and 390 of them belonged to the ‘pneumonia’ class.

```
              precision    recall  f1-score   support

           0       0.95      0.77      0.85       234
           1       0.88      0.97      0.92       390

    accuracy                           0.90       624
   macro avg       0.91      0.87      0.89       624
weighted avg       0.90      0.90      0.89       624
```

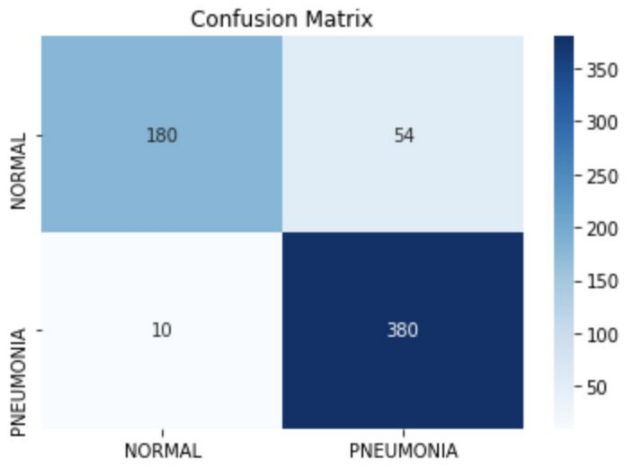

Figure 5. *Classification Report and Confusion Matrix*

The performance of the baseline model was also observed using classification report and confusion matrix as shown in Figure 5. Precision determines what percent of the model’s predictions were correct. For class 0 (‘normal’ images), precision is 95 %, meaning that 180 images were classified as ‘normal’ out of a total number of 190 (180 + 10) ‘normal’ classified images. For class 1 (‘pneumonia’ images), precision is 88 %, meaning that 380 images were classified as ‘pneumonia’ out of a total number of 434 (380 + 54) ‘pneumonia’ classified images. Recall is the ability of a classifier to find all positive instances. Recall determines what percent of the positive cases the model identified. Class 1 (‘pneumonia’) has a higher recall score than class 0 (‘normal’). This means that the model is performing better in terms of identifying the images that have pneumonia. Recall for the ‘normal’ class is 77 %. This means that 180 out of 234 (180 + 54) actually ‘normal’ images were classified as ‘normal’. And recall for the ‘pneumonia’ class is 97 %, meaning that 380 out of 390 (380 + 10) actually ‘pneumonia’ images were classified as ‘pneumonia '.

4. Conclusion

In this project, I applied convolutional neural networks to the binary classification problem of determining which of two classes ‘ normal’ or ‘pneumonia’ a chest x-ray falls under. The algorithm begins by transforming chest X-ray images into sizes smaller than the original. The next step involves the identification and classification of images by the convolutional neural network framework, which extracts features from the images and classifies them. I obtained a baseline model that achieved a 90% accuracy on the test set. Although this project is far from complete, it is remarkable to see the success of deep learning in such varied real world problems.

The study was limited by depth of data. With increased access to data and training of the model with radiological data from patients and nonpatients in different parts of the world, significant improvements can be made. The lack of sufficient numbers of medical images seems to be a common problem for other medical issues besides pneumonia.

One approach to solve this problem could be data augmentation technique. Data augmentation promises to be a useful tool in any case in which there aren’t sufficient numbers of images. It is used to artificially increase the size of the training dataset. In medical imaging, this is typically done with transformations that are applied to both the images and labels equally, creating warped versions of the training data. Augmentation methods commonly employ transformations such as rotations, reflections which produce training images that closely resemble one particular training example.

The second approach to increase the number of images could be using Generative adversarial networks (GANs). Generative adversarial networks (GANs) are an exciting recent innovation in deep learning. GANs are generative models: they create new data instances that resemble the training data. GANs consist of two parts: generators and discriminators. The generator model produces synthetic examples (e.g., images) from random noise sampled using a distribution, which along with real examples from a training data set are fed to the discriminator, which attempts to distinguish between the two. Both the generator and discriminator improve in their respective abilities until the discriminator is unable to tell the real examples from the synthesized examples with better than the 50% accuracy expected of chance.

Follow-on work to extend this application could include using ‘Transfer Learning’ which enables to use pre-trained models from other people by making small changes. A pre-trained model is a model created by someone else to solve a similar problem. Instead of building a model from scratch to solve a similar problem, you use the model trained on other problem as a starting point.