Deep Learning with Minimal Coding Effort: A Tutorial on Theory and Implementation of Deep Artificial Neural Networks

Behrooz Davazdahemami

Hamed Zolbanin

Dursun Delen

Deep Learning with Minimal Coding Effort: A Tutorial on Theory and Implementation of Deep Artificial Neural Networks

Tutorial

Behrooz Davazdahemami
University of Wisconsin-Whitewater
davazdab@uww.edu

Hamed M. Zolbanin
University of Dayton
hmajidizolbanin1@udayton.edu

Dursun Delen
Oklahoma State University
dursun.delen@okstate.edu

Abstract

Advances in computer technologies in the past couple of decades has enabled data and computer scientists to employ deep neural networks to detect and analyze complex patterns in large and varied data repositories from a wide variety of application domains. Given the interest in big data and analytics coursework in most information systems departments, this paper provides a step-by-step tutorial on the design and implementation of deep neural networks using an open-source, low-code, intuitive analytics platform. This platform (KNIME) suits well for both technical and non-technical users. Although this tutorial focuses on an image processing (classification) project in the popular context of healthcare, we believe the provided guidelines, with slight modifications, can be applied to the design and implementation of various deep learning architectures built to analyze a wide variety of data types.

Keywords: *Deep Learning, Improving classroom teaching, Tutorial, Artificial Intelligence*

Introduction

Today, deep learning is increasingly becoming integral to many widely used software services and applications. The primary focus areas of deep learning include speech and audio processing, language modeling, and natural language processing, information retrieval, object recognition and computer vision, and multi-modal and multitask learning (Deng and Yu 2014; Esteva et al. 2019).

KNIME analytics platform (https://knime.org) is a java-based, open-source platform for implementing a wide variety of data mining and machine learning algorithms in a user-friendly, drag-and-drop environment. It provides data analysts with a lot of functionalities to perform almost every operation required in a data analysis project, ranging from data preprocessing and visualization to building and deploying advanced models by creating node and branch workflows[1]. KNIME is currently the first analytics platform of its kind that has integrated with deep learning frameworks and has provided its users with the ability to implement resource-intensive deep learning models using GPUs.

Our purpose in this tutorial is illustrating how the technology can be used for analyzing real medical data. We make use of KNIME, as an easy-to-learn and easy-to-use platform that suits well for technical and nontechnical users alike, particularly due to its drag-&-drop structure, which eliminates the need for

[1]. We assume that the reader has some basic knowledge of working with the KNIME analytics platform. If you are not familiar with KNIME, you may begin by reviewing a quick tutorial provided by the tool's developer: https://www.knime.com/knime

knowledge of computer programming. Additionally, because deep learning is popular for its computer vision applications, and given the abundance of medical image data available to medical decision makers, we choose medical image classification as the application area in our demonstrations. Specifically, we employ deep learning technology to develop a medical decision support system to help physicians diagnose pneumonia using chest x-ray images.

Convolutional Neural Networks

Convolutional Neural Networks (LeCun et al. 1989), also known as CNNs, are among the most popular types of deep learning architectures. CNNs are, essentially, a variation of the Multi-Layer Perceptron (MLP) architecture designed to recognize visual patterns directly from pixel images with minimal preprocessing (LeCun and Bengio 1998); however, they are also applicable to non-image data sets. The main characteristic of the convolutional networks is having at least one layer involving a convolution weight function, instead of the general matrix multiplication.

Convolution is a linear operation whose purpose is extracting simple patterns from sophisticated patterns observed in input data. For instance, in processing an image containing several objects and colors, convolution functions can extract simple patterns such as the existence of horizontal or vertical lines or edges in different parts of the picture. A layer containing a convolution function in a CNN is called a *convolution layer*.

In a convolution layer, a set of weights, referred to as *convolution kernel* or *filter*, are shared between inputs. This kernel, which is commonly represented by a small matrix of size $W_{r\times c}$, moves around the input matrix to produce the outputs (Cornelisse 2018). For a given input matrix *V*, the convolution function can be expressed as:

$$z_{i,j} = \sum_{k=1}^{r} \sum_{l=1}^{c} w_{k,l} v_{i+k-1,j+l-1}$$

As an example, suppose the input matrix to a layer *(V)* and the convolution kernel *(W)* are given as follows:

$$V = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad W = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

Figure 1 illustrates how the convolution function computes the output. As shown, each element of the output matrix is obtained by adding the one-by-one point multiplications of the kernel elements by the corresponding *r*×*c* subset of the input matrix elements (in this example, *r* = *c* = 2 because the kernel is 2×2). For instance, Z_{12} (the second element in the first row of the output matrix) equals 0(0) + 1(1) + 1(1) + 1(0) = 2.

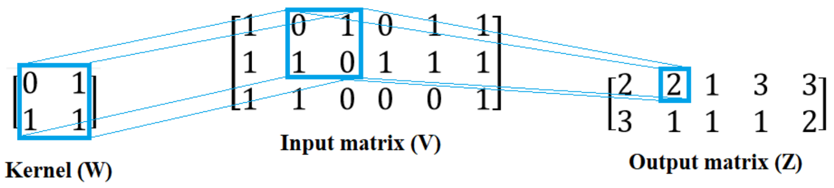


Figure 1- Convolution of a 2×2 kernel by a 3×6 input matrix

In the above example, the magnitude of each element in the output matrix depends directly on the extent to which the kernel matches with the 2×2 subset of the input matrix. For example, Z_{14} is the result of convoluting the kernel by the part of the input matrix that is the same as the kernel (see Figure 2). Hence, the convolution operation indeed converts the input matrix to an output in which those elements that have a particular, desired characteristic (as reflected by the kernel) are amplified.

$$\left[\begin{array}{ccc|cc|c} \cline{4-5} 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ \cline{4-5} 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right]$$

Figure 2- Output of convolution operation is maximized if kernel matches the input submatrix

This feature of convolution functions is especially very useful in practical image processing applications. For instance, if the input matrix represents pixels of an image, a kernel representing a particular shape (e.g., a diagonal line) may be convoluted into that image to extract parts of it that include the desired shape. Figure 3 depicts the result of applying a 3×3 horizontal line kernel to a 15×15 image of a square.

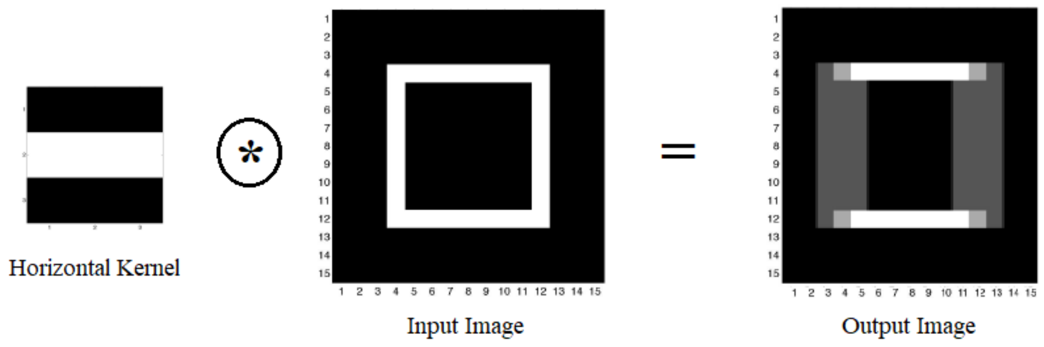


Figure 3- Using convolution for extracting features from images (horizontal lines in this example)

Convolution layers are often followed by a pooling (a.k.a. subsampling) layer. Pooling layers are in charge of consolidating large tensors (i.e., multidimensional vectors) to one with a smaller size, and reducing the number of model parameters while keeping their important features (Sharda et al. 2019).

Pooling can be thought of as an operation that summarizes large inputs - whose features are already extracted by the convolution layer - and shows us just the important parts (i.e., desired features) in each small neighborhood in the input space. For instance, in the case of the example shown in Figure 3, if we use a 3×3 consolidation window to place a max-pooling layer after the convolution layer, the output will be similar to what is shown in Figure 4. As it can be seen, the already convoluted 15×15 image is consolidated in a 5×5 image, while the main features (i.e., horizontal lines) are preserved.

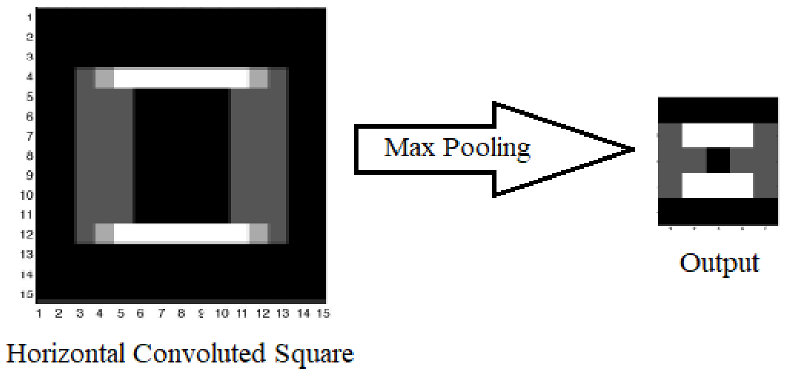


Figure 4- Example of applying max pooling on an image to reduce its size

Pooling is occasionally used just to modify the size of matrices coming from the previous layer so that it conforms to the requirements of the following layer in the network. The choice of proper pooling operation, as well as the decision to include a pooling layer in the network at all, depends highly on the context and properties of the problem that the network is trying to solve. There are some guidelines in the literature to help in making such decisions by the network designers (Boureau et al. 2010, 2011; Scherer et al. 2010).

Image Classification using CNNs

An image, in terms of data type, is nothing but a tensor of numbers each representing the color intensity of its corresponding pixel. For a grayscale image, each pixel can be represented by a number between 0 and 255 indicating the intensity of the black color. Therefore, grayscale images are *single-channel*, which means each pixel in them can be represented by a single number. For colored images, on the other hand, each pixel should be represented with three distinct numbers, indicating intensities of the three primary colors *Red*, *Green*, and *Blue* (*RGB*) for reproducing the color of that particular pixel. Hence, with color images we deal with three channels.

Any image processing task, then treats each image as a two- (grayscale) or three-dimensional (color) tensor. Given that, even a relatively small image would involve thousands of numbers each representing a pixel of that. While ach of those numbers can be treated as an input to an MLP deep network, that would require assigning a unique weight parameter to each input which leads to increasing the time required for optimizing the network in an exponential manner. For instance, a small image of size 150x150 pixels would require 22,500 weight parameters to be assigned to the inputs per each neuron they go into throughout the network, each of which needs to be updated several times during the network training.

To avoid this issue CNNs use a set of convolution kernels (a.k.a. filters) as shared weight parameters between inputs. Each kernel moves around the image matrix/tensor to produce the output matrix. By convolving a kernel into a matrix, we actually are converting the input matrix to an output in which the parts that have a particular feature (reflected by the kernel) are bolded. This characteristic is especially useful in practical image processing applications, where each kernel can represent a particular pattern or shape (e.g. edges, roundness, horizontal lines, etc.) and convolving that into a given image leads to bolding parts of the image including that specific pattern or shape (see the example provided in Figure 3).

In a typical image processing task using CNNs hundreds of kernels of a given size (normally 2x2 or 3x3) will be produced randomly and applied sequentially to each image to extract various features (i.e., patterns) involved in that. Each kernel will then be updated (i.e., trained) throughout the network training process such that the extracted features become more relevant and distinctive in determining the specific class to which the image belongs. The general process for an image classification task using CNN networks is shown in Figure 5 and described below.

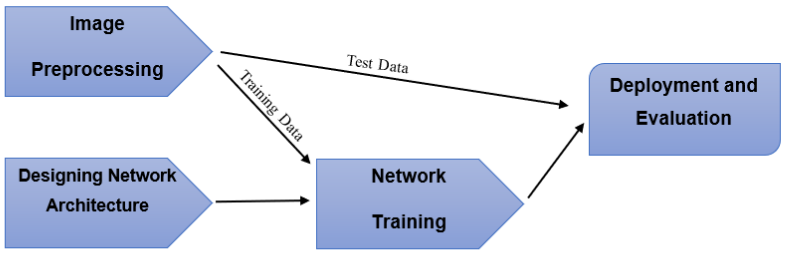


Figure 5- The general process of image classification using CNN deep networks

KNIME Keras Deep Learning Integration

In its recent versions, KNIME has released a new extension called Keras deep learning integration[2] that facilitates implementation of deep learning models without any coding requirements. Using Keras deep learning integration in KNIME one can design, train, and deploy various kinds of deep neural networks (not limited to CNNs) just by a few drags and drops. Various types of deep neural network layers are encapsulated in the form of nodes in KNIME's node and branch context, and a user can simply create a network by dropping the required type of layers (e.g., input, convolution, pooling, etc.) in a workflow area and connecting them using arrows in the sequential order specified in the design stage.

[2] Installation guides for the Keras extension are available at: https://www.knime.com/deeplearning/keras

Each node in KNIME has a configuration window, where the users can determine corresponding parameters. In the context of Keras nodes, users can specify, through the nodes' configuration window, such parameters as input shape, number of kernels, kernel shape, number of epochs, batch size, and optimization method without any coding effort.

In the next section, we use a real dataset to demonstrate how a deep learning image classification project can be implemented in KNIME by the means of the Keras integration.

Step-by-Step Image Classification in KNIME

Problem

One of the major applications of image processing is within the healthcare industry. Medical images for a long time have been invaluable sources of information for the physicians in their diagnoses. While in most cases physicians are readily able to make the right diagnosis through visual assessment of medical images, yet they are not mistake-proof. Clinical Decision Support Systems (CDSSs) are computer information systems designed to aid clinicians in doing a more scrutinized assessment of the patients' situations from various aspects and make optimal decisions, which ultimately lead to saving more lives and money.

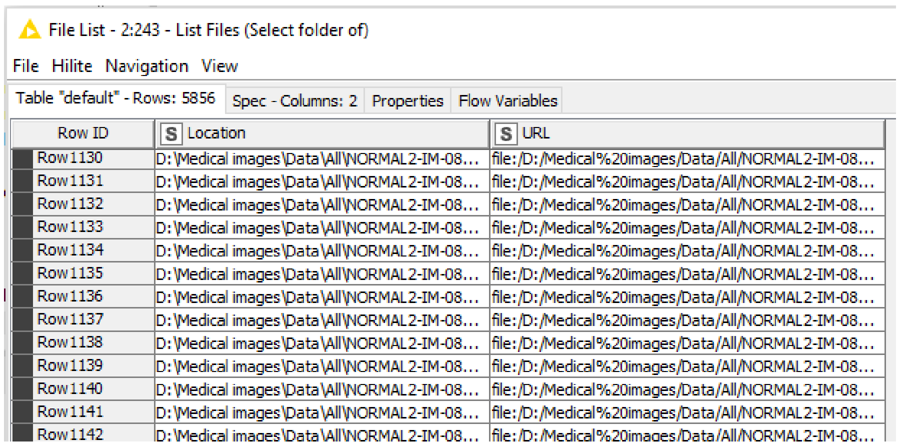
Processing medical images using deep learning techniques has recently been subject to a lot of interest as a cutting-edge CDSS solution to help reducing the chances of mistakes in physicians' diagnoses based on these sources of information (Deserno 2011; Eklund et al. 2013). Specifically, in these solutions, huge repositories of historical expert-annotated medical images are being used to train complex deep networks for classifying medical images of a given organ as evidence for various conditions relevant to that area in the human body.

In this tutorial, for demonstration purposes, we use a relatively small sample of expert-annotated chest X-Ray images taken from a larger set of annotated images provided by Wang et al. (Wang et al. 2017). While the original dataset includes multiple labels for each image, indicating the conditions shown by that, we limited our dataset to only 5,856 images, from which 4,273 (73%) indicating the existence of Pneumonia, whereas the rest (1,583; %27) are related to the normal patients (i.e., no condition). Therefore, the problem to be addressed by the image processing task, explained next, is to train a CNN deep network aiming at automatically classifying images in the dataset into either Pneumonia or Normal classes.

Importing images into KNIME and creating labels

Clearly, the first step in each data processing task is to read the dataset from an external environment. In KNIME, there are multiple nodes designed for inserting and reading various data types from either local hard disks or even clouds. Since in this case we need to insert around 6,000 image files into KNIME, we first need to create a list of files along with their location URL (either on the web or on local storage), so that KNIME can refer to those locations and read the images. This task can be easily done using a "List Files" node in KNIME.

Before bringing images into KNIME, we have already copied all the images in a single folder on our local hard disk. In addition, for simplicity, we have renamed the image files and added their labels (Normal or Pneumonia) to the beginning of their names. After adding the List Files node to the KNIME workflow, we can simply right-click on that and open the Configurations window for that node. All we need to do at this stage is to browse the folder including the images on our local disk. The node also provides us with multiple options to specify which files (based on their extension, naming patterns, etc.) from the corresponding folder should be listed. However, we do not need any filters to be applied in this particular example. Once you confirm the configurations and execute the List Files node, the node's output should look like Figure 6, where a URL has been created for each image file located in the folder.



File List - 2:243 - List Files (Select folder of)

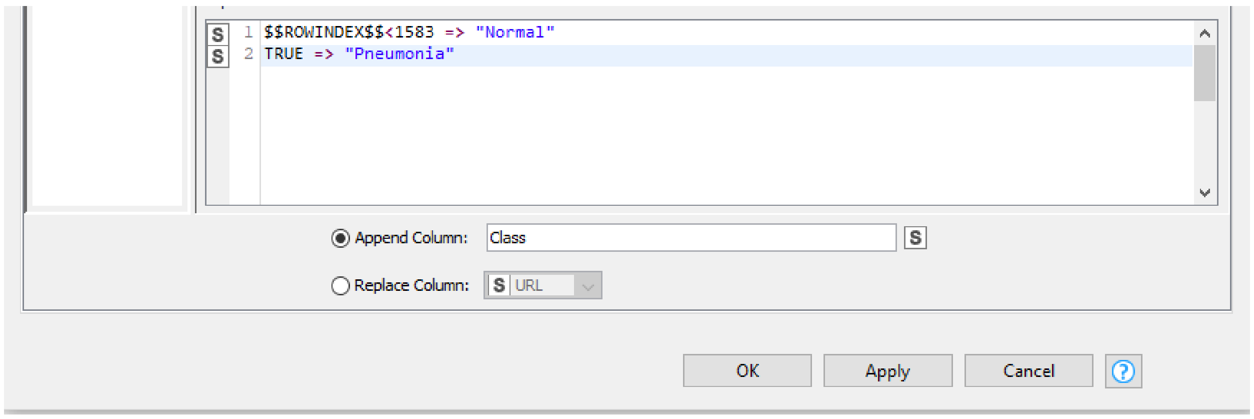
File Hilite Navigation View

Table "default" - Rows: 5856 | Spec - Columns: 2 | Properties | Flow Variables

Row ID	S Location	S URL
Row1130	D:\Medical images\Data\All\NORMAL2-IM-08...	file:/D:/Medical%20images/Data/All/NORMAL2-IM-08...
Row1131	D:\Medical images\Data\All\NORMAL2-IM-08...	file:/D:/Medical%20images/Data/All/NORMAL2-IM-08...
Row1132	D:\Medical images\Data\All\NORMAL2-IM-08...	file:/D:/Medical%20images/Data/All/NORMAL2-IM-08...
Row1133	D:\Medical images\Data\All\NORMAL2-IM-08...	file:/D:/Medical%20images/Data/All/NORMAL2-IM-08...
Row1134	D:\Medical images\Data\All\NORMAL2-IM-08...	file:/D:/Medical%20images/Data/All/NORMAL2-IM-08...
Row1135	D:\Medical images\Data\All\NORMAL2-IM-08...	file:/D:/Medical%20images/Data/All/NORMAL2-IM-08...
Row1136	D:\Medical images\Data\All\NORMAL2-IM-08...	file:/D:/Medical%20images/Data/All/NORMAL2-IM-08...
Row1137	D:\Medical images\Data\All\NORMAL2-IM-08...	file:/D:/Medical%20images/Data/All/NORMAL2-IM-08...
Row1138	D:\Medical images\Data\All\NORMAL2-IM-08...	file:/D:/Medical%20images/Data/All/NORMAL2-IM-08...
Row1139	D:\Medical images\Data\All\NORMAL2-IM-08...	file:/D:/Medical%20images/Data/All/NORMAL2-IM-08...
Row1140	D:\Medical images\Data\All\NORMAL2-IM-08...	file:/D:/Medical%20images/Data/All/NORMAL2-IM-08...
Row1141	D:\Medical images\Data\All\NORMAL2-IM-08...	file:/D:/Medical%20images/Data/All/NORMAL2-IM-08...
Row1142	D:\Medical images\Data\All\NORMAL2-IM-08...	file:/D:/Medical%20images/Data/All/NORMAL2-IM-08...

Figure 6- The output of List File node with image URLs

After creating the file list, we need to add an additional column to that to specify the class label of each image file. Since we have already added labels to the beginning of file names, and the files are sorted in alphabetical order in the file list, we can simply use the Row Indices to create class labels. That is, the first 1,583 image files in the list are assigned a "Normal" label and the rest of the list get a "Pneumonia" label. A Rule Engine node in KNIME can be used to apply this simple rule (or any other if-then type of rule) to our existing file list and create the class column. To this end, we need to connect the output terminal of the List Files node to the input terminal of Rule Engine and use the following self-explaining syntax in the Expression section of the Rule Engine configurations dialog, shown in Figure 7.



```
$$ROWINDEX$$<1583 => "Normal"
TRUE => "Pneumonia"
```

(●) Append Column: Class

( ) Replace Column: URL

Figure 7- Adding a "Class" column for image labels using Rule Engine node

Pre-processing images

After creating a list of image file URLs along with their corresponding labels, we can use an *Image Reader (Table)* node to read the image files from the provided URLs. The images used in this study are all in the PNG format, however the Image Reader node supports a wide variety of other image formats (i.e., all the file formats supported by the Bio-Formats library) as well. More information about the supported formats in this library is provided in the following website: http://loci.wisc.edu/bio-formats/formats . The Image Reader function in KNIME basically reads the provided images and converts and imports them to the KNIME internal image format, compatible with the KNIME's image processing nodes.

To configure the *Image Reader (Table)* node, we first need to specify which column of the file list table (i.e., output of the Rule Engine node) provides the path to the image files (i.e., Location in this example). In addition, we have to specify two other important elements in this section, namely Image Factory, and Pixel Type. The former determines the way we would like our images to be converted to arrays of numbers in KNIME. The Array Image Factory method converts each image (with all its channels) to a single array, whereas Planar Image Factory stores each channel in a separate array and Cell Image Factory creates multiple arrays of the same size for storing each image. The main difference among these options is different speed in accessing the resulted stored images (Array Image Factory is the fastest) as well as limitations in the number of pixels (Array Image Factory is the most limited). For this study, since the images are all reasonably small, we have set this option on Array Image Factory.

The Pixel Type option asks for the numeric type of pixel values (i.e., integer, binary, float, long, etc.). If you are not sure about your images' pixel types, it is always safe to choose the <Automatic> option and let KNIME to identify the right pixel type for us. After making and confirming these changes (leave the other options unchanged), we may execute the node to import the images (see Figure 8). Next, the imported images need to be processed before we can use them to train the deep networks.

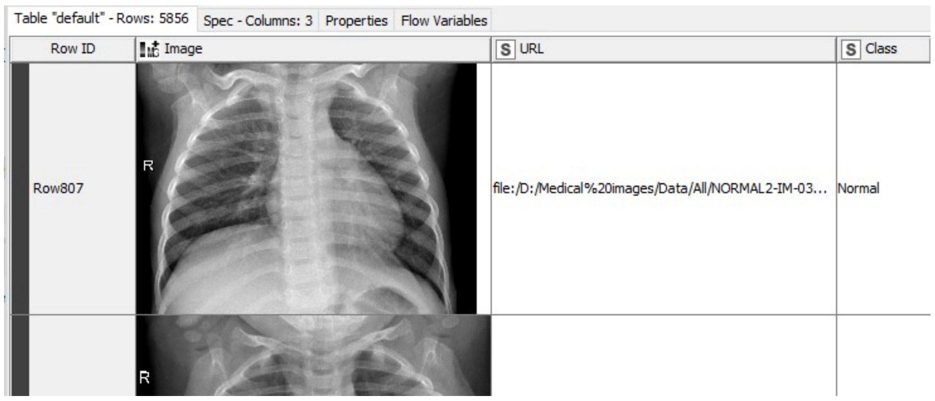


Table "default" - Rows: 5856 | Spec - Columns: 3 | Properties | Flow Variables

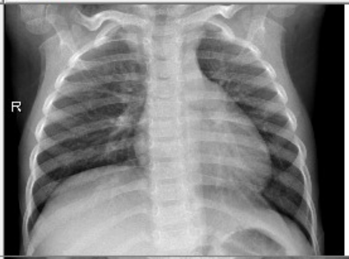
Row ID	Image	S URL	S Class
Row807		file:/D:/Medical%20images/Data/All/NORMAL2-IM-03...	Normal

Figure 8- Output of the Image Reader node

Each image channel is considered as a matrix of real numbers in the range of [0, 255], where each number indicating the intensity of the color corresponding to that channel in that pixel. These pixels' color intensities need to be *normalized* to a small range, typically [0, 1], in order to avoid any biases in training the classification models. This can be done using the *Image Calculator* node in KNIME. For normalization, the mathematical operation we need to apply to the pixels is simply dividing them by 255. We have also set this node to replace the original images by the normalized ones, and to use FLOATTYPE pixels for creating its output. It is important to choose the pixel types as either FLOATTYPE or DOUBLETYPE to precisely keep the original variety in their color intensity.

The next adjustment we need to perform on the images is to unify their sizes. Images in a dataset might have various sizes. In a given CNN, every input tensor must be of the same size. For this purpose, we may employ the KNIME node called *Image Resizer*. The choice of resizing strategy is particularly critical when at least some images need to be resized to a size larger than their original dimension. However, for reducing dimensions (which is the case in this project) typically all the interpolation strategies would work equally well. For this study, we chose a manual resizing mode, where the absolute size of both X and Y dimensions of the images are set to be resized to 150px.

Figure 9 shows the nodes described in this section in the KNIME workflow area.

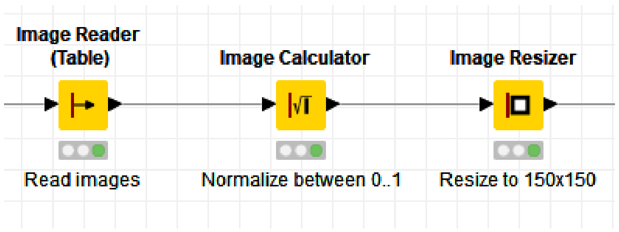


Figure 9- The image pre-processing nodes in the KNIME workflow

Designing network architecture

The network design stage in deep learning applications involves specifying several key elements, including number and type of layers, number of units of each layer, transfer function of each layer, and several other parameters specific to each layer type. The first layer in every typical deep network architecture is the *Input Layer*, which specifies the shape, data type, and data format of the input tensors. An input layer can be defined in the network architecture in KNIME using the *Keras Input Layer* node. In our example, the input tensors (images) are basically two-dimensional matrices of size 150×150. Because our images in this case are all grayscale, we have only 1 channel, which should be specified in the input shape. Therefore, the input shape that should be specified in the node configuration is either 150×150×1 or 1×150×150.

The input should then pass through one or multiple convolution layers to be processed. Each convolution layer has a limited capability in extracting features from a given input; therefore, depending on the complexity of the classification task and the variety of images, we might need to add more convolution layers in a sequential manner, so that the feature extraction load gets distributed on them. In the case of our example, since our images are highly similar (i.e., all are from the same organ in the same position), we chose to include only three convolution layers. The KNIME node designed for adding a single convolution layer to the network architecture for image classification is the *Keras Convolution 2D Layer* node. In the configuration dialog of a given convolution layer node we need to specify the number of filters (i.e., feature maps resulted from that layer), kernel size (i.e., size of the convolution window), stride (i.e., the kernel movement size), padding strategy (i.e. whether to pad the output feature map to make to the same size as the input image (same) vs. no padding (valid)), the dilation rate (i.e., number of pixels to be skipped during convolution), and the activation function.

As a rule of thumb, it is recommended that in case of using multiple convolution layers, we begin with a small number of filters in the first layers and gradually increase that in the following layers. In this project we decided to consider only 32 feature maps (filters) for the first layer. The second layer was also given 32 filters, however we increased that to 64 for the third convolution layer.

We have specified a stride size of (1, 1) for this example, which means the convolution window should move around each image with a step size of 1 pixel in horizontal as well as vertical directions. Using a (1, 1) stride size, always the pooled image will have the same size as the image coming into the pooling layer.

Each of the convolution layers in the network architecture is followed by a *Keras Max Pooling 2D* layer. As mentioned earlier, a pooling layer is generally meant to consolidate the elements of a convoluted image, while keeping its most important features. The *pool size* parameter is in fact the dimensions of the consolidation window. Like the convolution operation, for pooling we also need to specify a stride factor for moving of the pooling window around the images.

After including a sufficient number of convolution and pooling layers in a sequential manner, the processed images need to pass through one or more *dense* layers to be classified. However, dense layers only expect arrays as input and are not compatible with matrix representation of images. To address this issue, we need to pass the images through a *Keras Flatten Layer* before sending them into any dense layer. What happens in a flatten layer is that it simply stacks the rows of any given matrix after each other and converts it to an array. The flattened feature maps then should pass through one or more dense layers for the ultimate classification task to be accomplished. The output layer in a typical CNN classification network is a dense layer with only 1 unit. Depending on the number of feasible states of the output, the output dense layer should have either a single neuron with a *Sigmoid* activation function (for binary target variables) or *m* neurons with a *Softmax* activation function (for a multinomial target variable with *m* levels).

Additionally, and as an option, in many cases network designers consider one or more *Dropout Layers* in between two consecutive layers of each kind in the network architecture. A Dropout layer randomly sets a given fraction of input units of its following layer to zero at each update during the training (Hinton et al., 2012; Srivastava et al., 2014). Even though this leads to losing some information, it is shown to be very helpful in avoiding overfitting of the classification model to the training data during the training stage. In this project, for demonstration purposes we incorporated only one dropout layer between the two dense layers with a 0.5 drop rate.

Figure 10 demonstrates all the network layers included in the sequential order described above within the KNIME workflow.

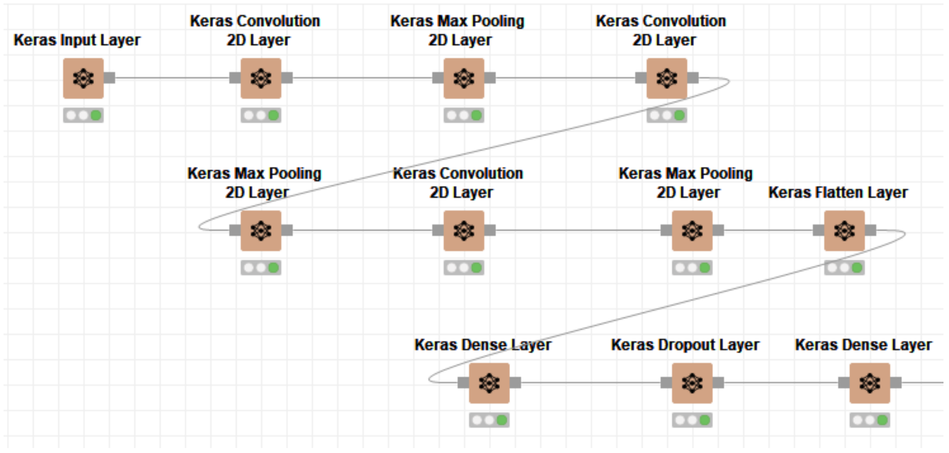

Keras Input Layer
Keras Convolution 2D Layer
Keras Max Pooling 2D Layer
Keras Convolution 2D Layer
Keras Max Pooling 2D Layer
Keras Convolution 2D Layer
Keras Max Pooling 2D Layer
Keras Flatten Layer
Keras Dense Layer
Keras Dropout Layer
Keras Dense Layer

Figure 10- The CNN sequential network architecture

Network Training

Once the network architecture is determined, we can start feeding the pre-processed data into the designed network and train network weights. Obtaining a large enough set of labeled medical images, however, is challenging in most of the practical cases as even in case of having access to a large pool of images, labeling them requires so much time and expertise. Fortunately, we can trick a neural network by providing it with multiple modified (rotated, zoomed, cropped, flipped, etc.) versions of the same image and improve its learning by such artificially augmented datasets. This trick is known as data augmentation and several different approaches have been proposed in the literature for this purpose (Lee et al. 2015; Perez and Wang 2017; Taylor and Nitschke 2017). Figure 11 demonstrates an example of data augmentation taken from (Liu et al. 2017) , in which 16 different modifications (right) of a given image are created just by a random combination of rotating (randomly between -20 and 20 degrees), shrinking (by a factor randomly chosen between 1 and 1.33), and cropping (to a 64x64 pixel image around a randomly chosen center) of the original image (left). Hence, before training our network we may perform some additional preprocessing and augmentation on the training data in KNIME.

In the preprocessing stage, we created a categorical target variable (i.e., Normal vs. Pneumonia); however, neural networks expect only numeric variables for both predictors and target. Hence, after partitioning we used a *Category to Number* node in KNIME to convert those string variables to numeric values (0 vs. 1).

The training dataset is now ready for data augmentation. One of the key features of the KNIME analytics platform is the flexibility it provides to the users by being integrable to several popular programming languages including Python, R, and Java. Since by the time of writing this paper there is no standard node in KNIME to perform data augmentation for image processing, we simply used a *Python Script (1=>1)* node to write a small code in Python for data augmentation. This is a standard KNIME node designed for inserting a piece of customized Python code in a workflow to process data. The (1 =>1) part of the node name indicates that it has only one input and one output port, which means it can receive input from a single node in the workflow and deliver its output to a single node as well. There are other varieties of the Python Script node available as well, in case one needs to combine data coming from multiple nodes or to send out multiple outputs to different nodes.

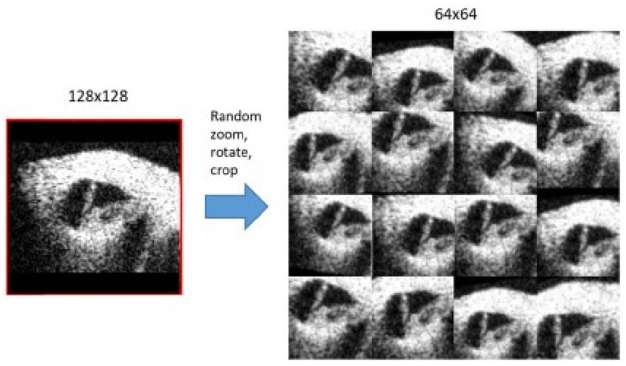


Figure 11- An example of image augmentation, a method to artificially increase the sample size in image processing problems

The following Python script was used to augment the training image dataset. First, we should import the required Python libraries as follows.

```
import os
os.environ["KERAS_BACKEND"] = "tensorflow"

from keras.preprocessing.image import ImageDataGenerator
import numpy as np
import pandas as pd
from pandas import DataFrame
from KNIPImage import KNIPImage
from scipy import ndimage
import keras
```

Next, we need to prepare our input data. That is, to read images from the previous node and convert them to python arrays as well as to read the Class associated with each image. The parts of the code that may be modified when you process your own datasets have been highlighted. In this case, the highlighted elements are the names of target and image variables from the input table as well as the number of categories of the target variable.

```
y_train = keras.utils.to_categorical(input_table['Class'],
num_classes=2)

x_train = []
for img in input_table['Image']:
   x_train.append(img.array)
```

The converted image python list (i.e., x-train) will have a shape of (50, 150, 150), where 50 indicates the default batch size of the Python Script node (does not matter for this operation), and 150, 150 indicates the dimensions of each image. However, in the following steps of the code, python expects an array of rank 4 in the form of (batch size, dimension1, dimension2, channel) for image augmentation. The problem is that with single channel (i.e., grayscale) images, python considers each image as a 2-dimensional tensor when it reads them from the input table. We should fix this problem before proceeding by adding an extra dimension to *x_train* using numpy reshape function and then transposing the result so that the dimensions become compatible with the expected format mentioned. Finally, we may convert the python list x_train, to a python array format for the augmentation operation. Note that the first line of the following code is not needed when colored images are analyzed as python automatically identifies the number of channels when reading images.

```
x_train=np.reshape(x_train,(1,)+np.shape(x_train))
x_train=x_train.transpose(1,2,3,0)
```

```
x_train = np.asarray(x_train)
```

Now we have the data ready for augmentation. The following loop can be used for generating modified versions of the training images. What happens is that it first defines an Image Data Generator function, which in this case allows random shearing transformation (shearing angles in counter-clockwise direction), random zoom into images, and random horizontal flip of them. There are several other options for creating a data generator which you may read about in Keras documentation for image preprocessing (https://keras.io/preprocessing/image/). Next, it takes each of the 4,000 images within the training data set and using a loop creates 20 randomly modified versions of that (i.e., `x_out`). It also assigns the same class as the original image to all its variations (i.e., `y_out`). Therefore, the output table would include 80,000 images (4,000×20) along with their corresponding classes.

```
train_datagen = ImageDataGenerator(shear_range=0.2, zoom_range=0.2,
horizontal_flip=True)
res = train_datagen.flow(x_train,y_train,batch_size=20)
i = 0
x_out = []
y_out = []
nBatches = 0
for batch in res:
      for r in range(np.shape(batch[0])[0]):
            x_out.append(KNIPImage(batch[0][r]))
            if (batch[1][r][0] == 1 ):
                  y_out.append(0)
            else:
                  y_out.append(1)
      nBatches = nBatches + 1
      if nBatches >= 4000:
            break
```

Finally, we should create an output table with `x_out` and `y_out` as the values of its columns.

```
output_table = DataFrame(columns=['Img', 'Class'])
output_table['Img'] = x_out
output_table['Class'] = y_out
```

The output images again lack the channel dimension due to being grayscale. To fix that, we can connect the output of the *Python Script (1=>1)* node to a *Dimension Extender* node in KNIME to add the channel dimension to the images. In this case, the dimension extender node converts the image tensor formats from (150, 150) to (150, 150, 1), which is expected by Keras for training the network. This format conforms to the format we set in the first layer of the deep network (i.e., the input layer). Note that this step may not be required if you are working with colored images. The part of workflow used for further preprocessing and augmentation of the training dataset are shown in Figure 12.

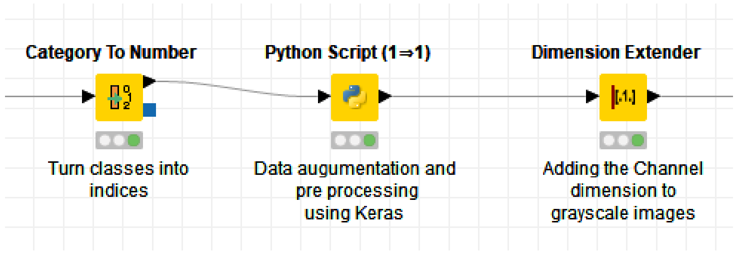

Category To Number
Python Script (1⇒1)
Dimension Extender
Turn classes into indices
Data augmentation and pre processing using Keras
Adding the Channel dimension to grayscale images

Figure 12- KNIME nodes for training data processing and augmentation

After this step our training dataset is ready to feed into the designed network to train its parameters. In order to train a deep learning network using Keras in KNIME we can use a *Keras Network Learner* node.

This node requires two main input elements to operate; first it needs to be connected to the designed sequential network (i.e., a connection from the last dense layer to it). Also, the training data set (coming from the Dimension Extender node) has to be connected to the first data port of the learner node. There is also a third (optional) data port for connecting validation dataset to this node. In that case, during training we may monitor the accuracy and loss of the model with regard to both training and validation datasets and possibly stop the training process whenever we feel the model is going to be overfit to the training data. Due to using a relatively small dataset just for demonstrating purposes, we chose not to use this option. Nevertheless, it is a very handy option to consider, especially when the model's performance on the training and test datasets differs dramatically (i.e., overfitting).

The *Keras Network Learner* node has multiple tabs in its configuration dialog, each of which involve critical parameters to be set. In the first tab of this window (i.e., "Input Data" tab), we need to specify in which column of the input table our input data (i.e., images) exist. The second tab (Target Data) asks for the column in which the target variable exists, as well as the loss function to be used for training the network. While Keras provides us with a variety of *standard loss functions* (e.g., binary cross entropy, categorical cross entropy, hinge, mean absolute error, etc.), each of which appropriate for a specific group of problems, there is also an option to define a *custom loss function* in python and use that for training. We used *Binary Cross Entropy*, which is a common standard loss function for binary classification problems.

The next tab (i.e., *Options*) is where the main training parameters can be set. In this tab, we should specify the number of epochs, batch size, and the optimizer algorithm along with its hyper parameters. The choice of batch size usually depends on the nature of the task and data, and can vary between one single example to the whole sample in each step of training. In this example, we chose to set the batch size to 50. Also, proper number of epochs depends on the complexity of the task, the number of network parameters that need to be trained, and the data. Typically, we need to train the model multiple times with different number of epochs and observe the model's performance with regard to both training and test data, and then adjust it accordingly.

Once we set the learner parameters, we can start training the network by executing the *Keras Network Learner* node. Depending on the size of data, batch size, number of epochs, network architecture, and the processing capacities of the machine, the training step might take a while to complete. The KNIME's Keras integration provides the users with a very helpful option to monitor the learning process while it is going on. They provide real time information with regard to the loss, accuracy, the batch, and epoch being processed. In addition, users are provided with a "Stop Learning" button to manually stop the learning process whenever they feel the model is about to overfit the training data.

Deployment and Evaluation

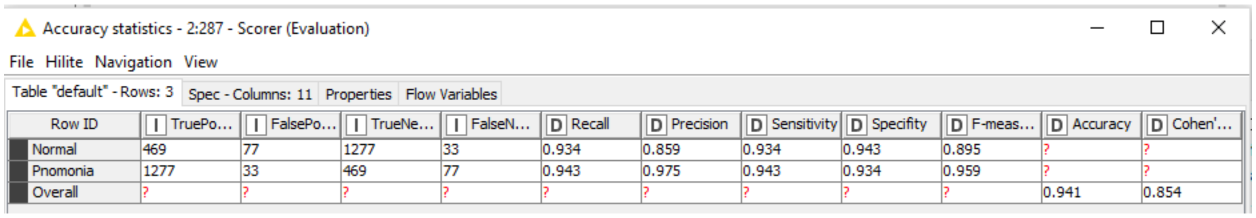
Once the network learner node execution is complete (or manually stopped by the user), we may apply the trained network weights to the test dataset we had set aside in the data partitioning step. The KNIME node designed for this purpose is *DL Network Executor*. It needs two main input elements to operate. First, it should get the trained network from the output of the *Keras Network Learner* node; and second, we should provide it with the test dataset coming from the second output port of the *Partitioning* node. Additionally, in the *Outputs* section of this dialog, we may add an output and specify the output layer of the network (i.e., the sigmoid dense layer in this case). Then we can confirm the changes and execute the node.

The output of this node is the input test data table plus an additional column indicating the probability of each image (record) being related to class 1 (i.e., Pneumonia in this case). These probabilities then need to be converted to the class labels to be comparable with the target variable. To this end, we may use a *Rule Engine* node to append a new column to the output table, in which every example with a predicted probability less than 0.5 is labeled as "Normal", and it is labeled "Pneumonia" otherwise. The output of the *Rule Engine* node will have two columns called Class and Prediction, which should be compared and contrasted in the next step to evaluate the model's performance.

The two main tools in KNIME for evaluating the performance of any binary classification problem are the nodes *Scorer* and *ROC Curve*. The scorer node compares the predicted categories against the real classes across the whole test dataset and returns the confusion matrix along with a table including various accuracy statistics (i.e., accuracy, sensitivity, specificity, precision, F-Measure, etc.). Also, ROC Curve node plots Receiver Operating Characteristics curve (true positive rate against false positive rate) which gives us a general idea of how powerful the model in is distinguishing positive from negative cases in a binary classification problem (i.e., the higher the area under the ROC curve, the more powerful the distinction power of the model).

To plot the ROC curve, we first should specify which column from the input table contains the real class of the images (i.e., "Class" column). Then, we should tell KNIME which category among all possible class values is the *positive class value*. This is actually the category in which we are interested in practice. For instance, it is more critical to a physician to identify a patient with Pneumonia than a normal patient. Finally, we need to specify the column from the input table containing the predicted probabilities of the cases belonging to the positive class. This is, in fact, the column created during the network execution process on the test dataset.

Figure 13 demonstrates the accuracy statistics (output of the Scorer node) of the model. As shown, our image classification model has worked very well, with a 94% overall accuracy and a 94.3% sensitivity in classifying chest X-Ray images (Pneumonia). Also, the area under the ROC curve (0.9795) confirms the high ability of the model in distinguishing normal patients from those with Pneumonia.


Accuracy statistics - 2:287 - Scorer (Evaluation)
File Hilite Navigation View
Table "default" - Rows: 3
Spec - Columns: 11
Properties
Flow Variables

Row ID	I TruePo...	I FalsePo...	I TrueNe...	I FalseN...	D Recall	D Precision	D Sensitivity	D Specifity	D F-meas...	D Accuracy	D Cohen'...
Normal	469	77	1277	33	0.934	0.859	0.934	0.943	0.895	?	?
Pnomonia	1277	33	469	77	0.943	0.975	0.943	0.934	0.959	?	?
Overall	?	?	?	?	?	?	?	?	?	0.941	0.854

Figure 13- The performance metrics of the trained network on the test dataset

Figure 14 shows an overview of the KNIME workflow we used for model building and deployment. The nodes included in the network configuration, as well as the Preprocessing blocks in the workflow, were compacted into two separate KNIME meta nodes to reduce the clutter and to make the workflow more understandable. The raw image dataset, data preprocessing, and model building workflows (.knwf format, importable in KNIME) are available at https://bit.ly/3lJmmak or download and practice. As soon as you open the workflows provided KNIME automatically will identify required but missing extensions (e.g., KNIME image processing extensions) on your installed version and suggests to install them before running the workflow.

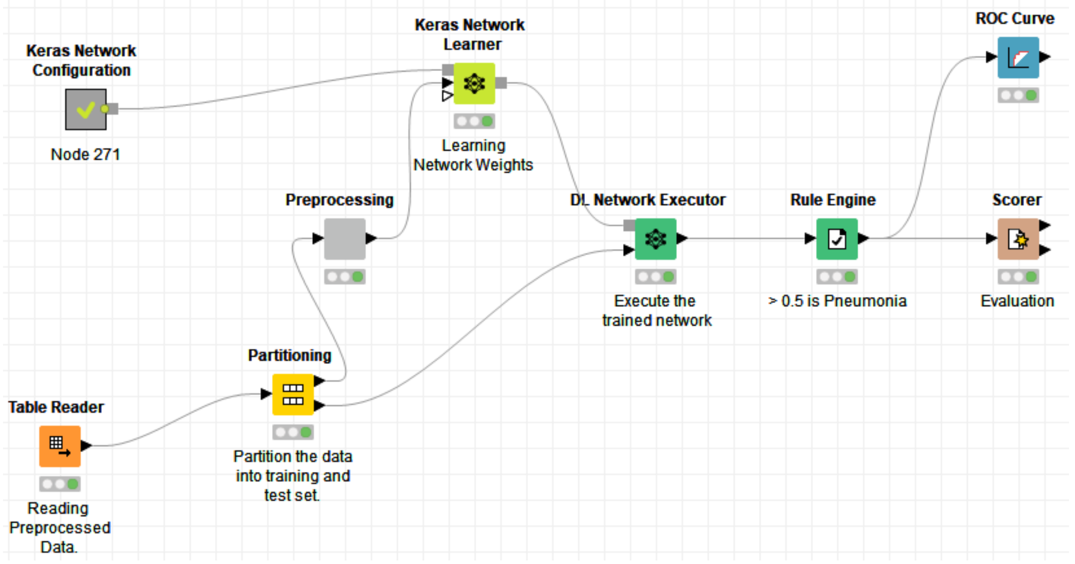


Figure 14- An overview of the nodes and connections in the KNIME workflow

Conclusion

In this paper, we provided a tutorial for performing image classification using Convolution Neural Networks (CNNs) in a medical context. This tutorial is particularly helpful for practitioners who do not have a strong background in computer programming but are interested in using a low-code environment for data-driven decision making. Moreover, this tutorial can also be helpful for educational purposes, specifically to teach neural network and deep learning concepts and applications to college students.

It should be noted that even though in this tutorial we demonstrated the entire process of design and training of a deep convolutional network for educational purposes, a common practice (specially for users with limited computational resources) is an approach called transfer learning, in which we use a well-established pre-trained network (e.g., ResNet50 or VGG16, …) and quickly adjust its hyperparameters using their training data.

References

Boureau, Y.-L., Ponce, J., and LeCun, Y. 2010. "A Theoretical Analysis of Feature Pooling in Visual Recognition," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 111–118.

Boureau, Y.-L., Le Roux, N., Bach, F., Ponce, J., and LeCun, Y. 2011. "Ask the Locals: Multi-Way Local Pooling for Image Recognition," in *Computer Vision (ICCV), 2011 IEEE International Conference On*, IEEE, pp. 2651–2658.

Cornelisse, D. 2018. "An Intuitive Guide to Convolutional Neural Networks."

Deng, L., and Yu, D. 2014. "Deep Learning: Methods and Applications," *Foundations and Trends® in Signal Processing* (7:3–4), pp. 197–387. (https://doi.org/10.1561/2000000039).

Esteva, A., Robicquet, A., Ramsundar, B., Kuleshov, V., DePristo, M., Chou, K., Cui, C., Corrado, G., Thrun, S., and Dean, J. 2019. "A Guide to Deep Learning in Healthcare," *Nature Medicine* (25:1), pp. 24–29. (https://doi.org/10.1038/s41591-018-0316-z).

LeCun, Y., and Bengio, Y. 1998. *The Handbook of Brain Theory and Neural Networks*, M. A. Arbib (ed.), Cambridge, MA, USA: MIT Press, pp. 255–258.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. 1989. "Backpropagation Applied to Handwritten Zip Code Recognition," *Neural Computation* (1:4), MIT Press, pp. 541–551.

Lee, C.-Y., Xie, S., Gallagher, P., Zhang, Z., and Tu, Z. 2015. "Deeply-Supervised Nets," in *Artificial Intelligence and Statistics*, pp. 562–570.

Liu, G. S., Zhu, M. H., Kim, J., Raphael, P., Applegate, B. E., and Oghalai, J. S. 2017. "ELHnet: A Convolutional Neural Network for Classifying Cochlear Endolymphatic Hydrops Imaged with Optical Coherence Tomography," *Biomedical Optics Express* (8:10), Optical Society of America, pp. 4579–4594.

Perez, L., and Wang, J. 2017. "The Effectiveness of Data Augmentation in Image Classification Using Deep Learning," *ArXiv Preprint ArXiv:1712.04621*.

Scherer, D., Müller, A., and Behnke, S. 2010. "Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition," in *International Conference on Artificial Neural Networks*, Springer, pp. 92–101.

Sharda, R., Delen, D., and Turban, E. 2019. *Analytics, Data Science, & Artificial Intelligence: Systems for Decision Support*, (11th Editi.), Pearson.

Taylor, L., and Nitschke, G. 2017. "Improving Deep Learning Using Generic Data Augmentation," *ArXiv Preprint ArXiv:1708.06020*.

Wang, X., Peng, Y., Lu, L., Lu, Z., Bagheri, M., and Summers, R. M. 2017. "Chestx-Ray8: Hospital-Scale Chest x-Ray Database and Benchmarks on Weakly-Supervised Classification and Localization of Common Thorax Diseases," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2097–2106.