Neural Networks and its Role in Image Classification:

A Detailed Review

Anthony Arnold

aa4170@mynsu.nova.edu

Nova Southeastern University

MMIS 643 Data Mining

Spring 2024

Table of Contents

1. Introduction

In this project, the concept of artificial neural networks will be reviewed and applied to a binary image classification problem. First, an in-depth description of neural networks is done, fundamentally defining the key components of their structure and the mathematical concepts used to construct them. The concepts involving activation functions, gradient descent, and backpropagation will also be touched upon. In addition, customizable variables will be introduced to deter overfitting and enable user-led optimization practices. This information would then be used to define convolutional neural networks and what enables them to be effective at feature detection. Finally, a binary image classification example using Python and the Keras library is showcased to show the effectiveness of neural networks in solving real-world problems.

2. Review of Artificial Neural Networks

2.1 Intro and Structure of Neural Networks

Artificial neural networks are data-driven machine learning algorithms that can be used for prediction and classification. The term "neural" comes from "the biological system's exceptional information processing features" (Thakur & Konde, 2021). The process of a neural network in simple terms consists of gathering data through input, learning important relationships throughout a series of layers, and outputting the predicted/classified value. There are many structures of neural networks, but the focus will be on multilayer feedforward networks. These are built based on three parts: the input layer, the hidden layer(s), and the output layer (Fig. 1).

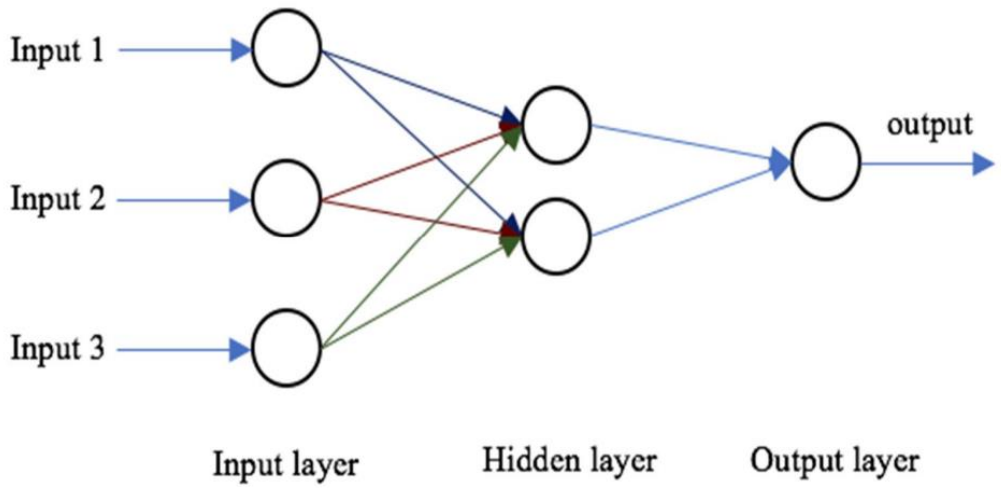


Figure 1: Architecture of a Multilayer Feedforward Neural Network (Thakur & Konde, 2021)

The input layer is where the network begins its computations. Predictor data is fed into input nodes which act as "neurons" to send the data off into the next layers of the network. The hidden layer(s) are the layers between the input and output layers. These layers along with the weights between them are used to calculate the potency of each predictor in

selecting the target value. Usually, one hidden layer is used but can be multiple if needed. Lastly, the output layer collects all the data/relationships that were collected throughout the network and outputs a target value. For classification, the number of output nodes is directly in line with the number of classes while the number of input nodes is associated with the number of predictors. This type of network is "feedforward", meaning it is run sequentially in order and doesn't route back to any other layers.

2.2 Calculating the Output of Nodes

Consider the format of the neural network posed in Figure 1. Let x_i be the input values (in this case there are three) and $w_{i,j}$ be the weights. In the figure, weights are denoted by the arrows that can be found between each layer. Starting from the input layer, the weighted sum is done to receive the next layer of values. The weighted sum formula can be written as

$$\theta + \sum_{i=1}^{n} w_{i,j} x_i$$

where n is the number of predictors (number of nodes within a layer if further down the network), i is the prior node, j is the node where the calculation is headed, and θ is a bias term that signifies the level of importance a node should have within the network. At this point, it is important to note that the weights and bias terms are not predetermined; these terms are "learned" as the neural network functions over time. Thus, for the first time initializing the network, the weights and biases are random. Let the weighted sum be denoted as α to simplify the next series of equations. After a weighted sum is computed, it is then run through an activation function. Activation functions are used to make the weighted sums non-linear. Since neural networks and the data used are usually non-linear, activation functions

are commonly used. There are plenty of activation functions that are used within a neural network (Fig. 2).

$f(\alpha) = \frac{1}{1+e^{-\alpha}}$ **Sigmoidal Activation Function**	$f(\alpha) = \frac{e^{2\alpha}-1}{e^{2\alpha}+1}$ **Hyperbolic Tangent Activation Function**
$f(\alpha) = k\alpha$ **Linear Activation Function**	$f(\alpha) = \begin{cases} \alpha, \alpha > 0 \\ 0, \alpha < 0 \end{cases}$ **Rectified Linear Unit Activation Function (ReLU)**

Figure 2: Types of Activation Functions

Figure 3 shows an example computation of the value of a node using the weighted sum formula and the sigmoidal activation function.

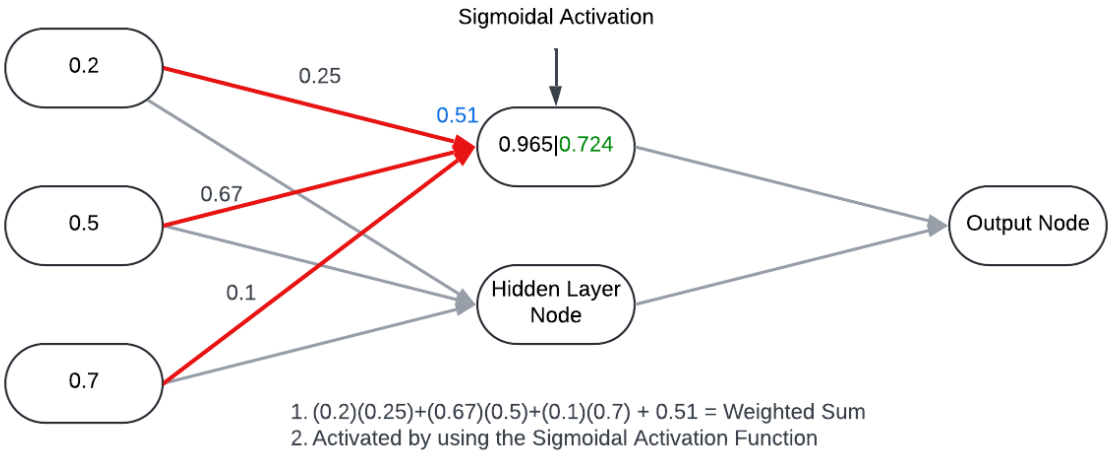


Figure 3: Example of calculating the output of a node

These steps are done recursively until the weights between the last hidden layer and the output layer are reached. Here, the weighted sum(s) are calculated and activated normally but are then injected into a function called softmax. Softmax normalizes the values and

recalculates them to sum up to one. Let the vector $\vec{z} = \begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix}$ be the collection of n outputs from the output layer. The softmax is defined as:

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{i=1}^{n} e^{z_i}}$$

After this is done, the output can be used for classification. For binary classification, cutoff values are used to split the decision. For example, if 0.5 is the cutoff value, class 0 has a probability of 0.6, and class 1 has a probability of 0.4, it would be classified as class 0. For binary classification, the sigmoidal function (Fig. 2) can be used in replacement of the softmax function since the softmax function is technically an extension of the sigmoidal function to more than two classes. Multi-variable classification involves choosing the class that has the largest output value among all.

2.3 Training the Network Using Gradient Descent and Backpropagation

Since the weights and bias terms are learned throughout the network, it is apparent that one forward pass through the neural network won't suffice. Epochs are the number of times the network is run through so that learning can occur. To learn which weights and bias terms fit the best within the network, the cost function is investigated. Simply put, the cost function can tell how well the neural network performed. Consider the output layer and the final values it calculates. The cost can be found by using the formula:

$$\text{Cost} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

Note that there are plenty of cost functions, but this one is used for the sake of simplicity. This is the mean squared error function, where $\hat{y}_i$ is the ith node's output value and y_i is 0 or 1

depending on whether the output node is the desired classification. The neural network aims to lower the cost value as much as possible. To do this, the weights and biases within the network must be tweaked to sway the classification in the right direction. If the desired output value is not close to 1, an increase in the weights and biases is needed so that the output value converges to 1 in the next run of the network. Similarly, decreasing the output values of the rest of the output nodes to 0 is needed, thus needing a decrease in weights and bias values. This may sound easy, but some obstacles make the process more complex. First, optimally calculating these updates is needed so that the network doesn't spend time with redundant weights and biases. This is where the gradient descent algorithm is needed as it tells us the optimal "path" to decrease the cost function. Let $\vec{\boldsymbol{w}} = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}$ be a vector that consists of all the weights and biases. This algorithm can be defined as

$$\nabla C(w_i) = -\eta \frac{\delta C}{\delta w_i}$$

$$\vec{\boldsymbol{w}} := \vec{\boldsymbol{w}} + \nabla \vec{\boldsymbol{w}}$$

Where η is the learning rate and C is the cost function. After every epoch, the gradient can be derived and added to each weight, optimally changing the weights and biases to decrease the network's cost. Due to the layer-by-layer structure of a neural network, implementing changes within the nodes requires a system to travel backward throughout the layers. Backpropagation is the system in question, starting at the output layer and traveling backward to the first weights/biases within the network. Beginning at the weights that connect to the output layer, the proportional changes needed within the previous layer are computed by adding all changes needed to increase the desired node's value and to decrease

the rest. These changes are proportional to the potency of a weight, meaning that weights/biases that affect the outcome more have a greater shift within its value. Note that changes within a previous layer to modify one outcome can inadvertently affect the others. Once this is done, the network updates the second-to-last layer and repeats the process once again between the next two layers. Another issue to consider is the gradient descent algorithm converging to a relative minimum. The best-case scenario is if the cost function converges to the absolute minimum since it is at the lowest value it can achieve (Fig. 4). The math behind these concepts can get complex as more weights/nodes are specified within the network. This results in many different optimizers capable of swaying the error function from reaching only one outcome. One such optimizer is called Adam, which updates the learning rates within the network to deter the network's cost from reaching a local minimum.

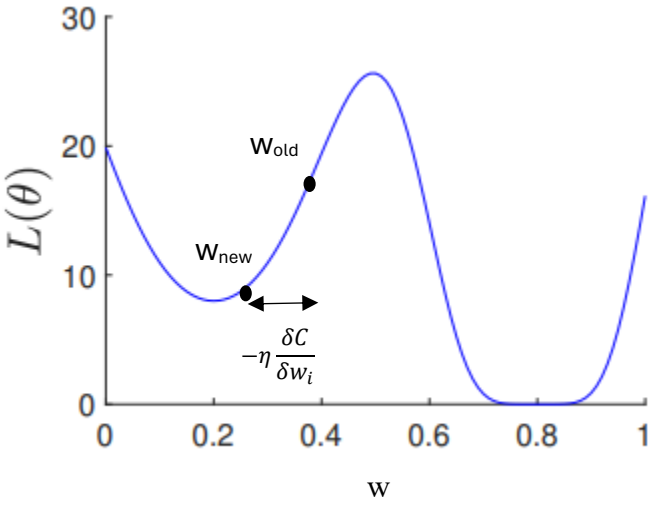


Figure 4: Weight values near the local minimum would converge to that minimum even though the absolute minimum is what is needed.

2.4 The Disadvantages of Neural Networks

As powerful as neural networks may seem, there are still some downsides that need to be taken into consideration. First, a lot of input data is needed for a neural network to function

well. Having too little training data can lead to an underfit model. For minority cases, it is important to oversample the training data so that the model can learn effectively in those cases. Lastly, the creator of the model must make sure the hyperparameters are reasonable. They must keep tabs on the training/validation accuracy to make sure the training accuracy isn't rising while the validation accuracy is decreasing. This is the direct result of overfitting, making changes within the hyperparameters of the network a necessity.

2.5 Convolution Neural Networks

By now, the concepts of neural networks and the math behind it has been introduced. How are these ideas implemented into image classification? In a traditional neural network, predictor data is usually fed into the model to classify/predict an outcome familiar to the user, but data present in photos doesn't explain how to identify what the photo is depicting. How does one go from pixel-intensity data to classifying a real-world object? Feature discovery is one of the main concepts of deep learning, where patterns are recognized as the model progresses.

Convolutional Neural Networks (CNN) are deep learning models specializing in feature discovery in supervised and unsupervised settings. The main thing that sets traditional neural networks and CNNs apart is their structure. The three layers that make up CNNs are the convolutional layers, the pooling layers, and the fully connected layers. Convolutional layers are the main components of CNNs, containing complex calculations that can detect features. These types of layers contain input data (pixels of an image), a kernel, and a feature map. The kernel (or filter) is a matrix that contains weights corresponding to what is wanted to be found within the input image. The kernel is the part that scans throughout the input image for

features, which is known as convolution. The feature map is the output of the layer, which is a matrix collection of all dot product values between the kernel and the input data.

Consider an example that aims to detect horizontal lines within a hand-drawn image. Grayscale values will be used, ensuring that the data present within the neural network is between 0 (white) and 225 (black). Since the model is searching for horizontal lines within the input, it must be represented within the kernel with weights (Fig. 5.a). Once preprocessing of the data is done to contain the same data format as the kernel (Fig. 5.b), convolution can occur.

0	0	0
1	1	1
0	0	0

(a)

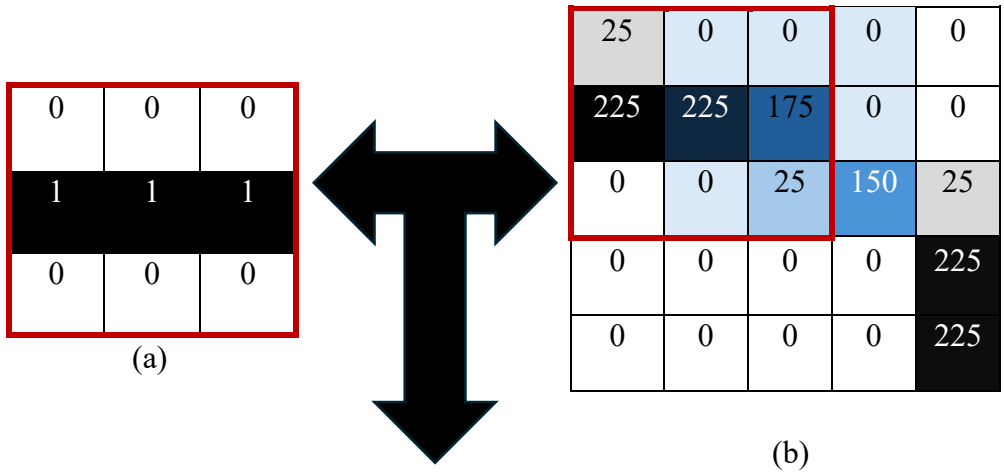
25	0	0	0	0
225	225	175	0	0
0	0	25	150	25
0	0	0	0	225
0	0	0	0	225

(b)

Figure 5: Kernel used to detect horizontal lines (a) and a 5x5 piece of a handwritten image (b)

The kernel is placed in comparison with the first 3x3 matrix formed on the top left corner. The dot product is then computed between the kernel and the top left matrix. Figure 6 shows the process for the first case within the image. Once the first result is implemented into the feature map, convolution occurs, shifting the kernel right to left and top to bottom to complete the feature map matrix. Figure 6 shows the next matrix (tinted in blue) in the

sequence if the kernel were set to move one unit at a time. Stride is a parameter that allows human input to choose how many units the kernel should shift through the image.



$$(0)(25) + (0)(0) + (0)(0) + (1)(225) + (1)(225) + (1)(175) + (0)(0) + (0)(0) + (0)(25)$$

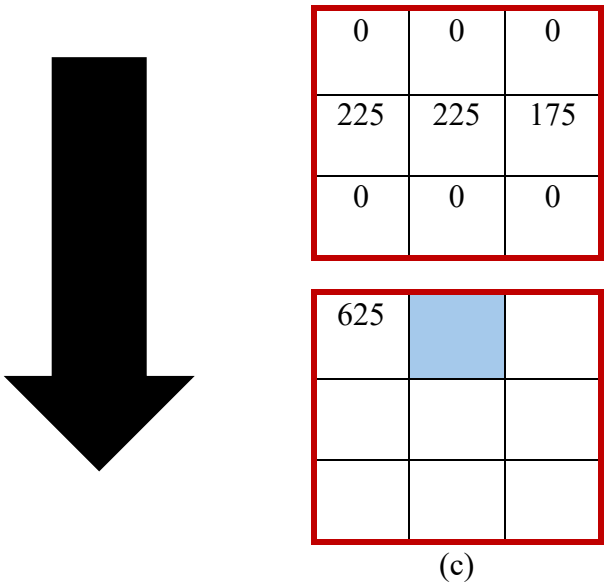


Figure 6: An example of the convolution layer at work. The dot product is done on the kernel (a) and the first 3x3 matrix with the image (b). It is then consolidated into the feature map (c)

Another parameter used in convolutional layers is padding, which adds extra pixel data around the border of the input data so that the kernel can take data that lies at the edges of the input into better consideration. For example, consider the case where the input data contains a horizontal line on the top edge of the input. Using the same kernel in the previous example (Fig. 5.a), the feature map would not recognize the horizontal line since it is not aligned with the kernel. Adding a zero-padding layer would aid the network in recognizing the horizontal line but will increase the feature map size as a result (Fig. 7).

225	225	225	150	0
0	0	0	225	50
0	0	0	50	175
0	0	0	0	225
0	0	0	0	225

(a)

0	0	0	0	0	0	0
0	225	225	225	150	0	0
0	0	0	0	225	50	0
0	0	0	0	50	175	0
0	0	0	0	0	225	0
0	0	0	0	0	225	0
0	0	0	0	0	0	0

(b)

Figure 7: The kernel (Fig. 4a) does not read the horizontal line well without padding (a) but identifies the horizontal line better with padding (b)

A pooling layer is very important within a CNN as it "provides a typical downsampling operation which reduces the in-plane dimensionality of the feature maps in order to introduce

a translation invariance to small shifts and distortions, and decrease the number of subsequent learnable parameters" (Yamashita, Nishio, Glan Do, Togashi, 2018). The most common type of pooling technique is max pooling, which takes the highest number out of a pooling matrix and eliminates the rest. For example, consider the 4x4 feature map and a 2x2 pooling matrix in Figure 8.

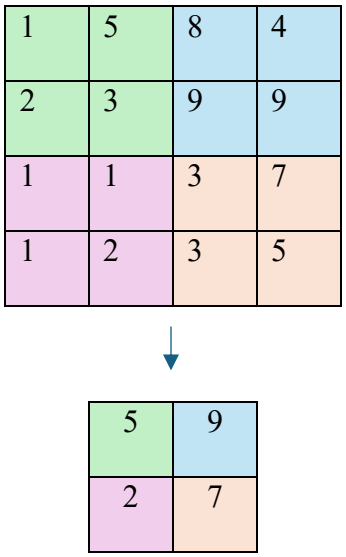


Figure 8: Example of Max Pooling

In the first green 2x2 matrix, 5 is the highest value so it is chosen as the carried value and the rest are dropped. Similarly, it is done for the rest of the 2x2 matrices within the feature map. Note that if the feature map contains an odd number of rows/columns, some values will never be taken into consideration. That is why padding, pooling size, and stride remain parameters within the pooling layer to ensure the consideration of crucial features regardless of their position.

The last building block of a CNN is the fully connected layer, which aims to flatten the last feature map of the pooling/convolution layers within the CNN. These flattened columns

are then grouped to form the fully connected layer and are connected to the outputs of the network. In classification, probabilities are calculated within the fully connected layer to produce an outcome. In simpler terms, the fully connected layer acts as a traditional neural network after the convolution/pooling processes have been done on the input data.

3. Binary Classification Problem Using Neural Networks

Now that the main concepts of neural networks have been introduced and the feature detection capabilities of convolutional neural networks have been investigated, it si not time to go over a binary image classification example. The example will be created using Python and the Keras library[1].

The purpose of this model will be to classify whether an X-ray image of a patient shows signs of pneumonia or not. The dataset contains:

- 5,216 images for training (2,608 “normal” and 2,608 “pneumonia”)
- 612 images for validation (304 “normal” and 304 “pneumonia”)
- 16 images for testing (8 “normal” and 8 “pneumonia”)

The first thing tackled within the *pneumoniaclass.py* file is transforming the dataset into a dataset type within Python. This is done by using the very handy *image_dataset_from_directory* function. This function also contains a bunch of preprocessing parameters to clean the data into a common format. Here, the images were resized to 128x128, and binary classification was specified. The photos were also set to grayscale since X-ray photos are already in black and white. The labels of “normal” and “pneumonia” were designated within the parameters.

Next, it is time to start working with the convolution neural network model. Keras does this process by building the model layer by layer and then fitting it with the data after. The first couple of layers consist of convolution and pooling layers. Hyperparameters such as the

[1] The Python code mentioned in this project was written by me while the data used within the project was gathered from a Kaggle dataset.

number of filters, kernel size, and desired activation function are noted in each convolution layer. In this example, ReLU activation functions are used among all convolution layers. Padding and stride are also hyperparameters that can be set within these layers, but for this example, they are not used. The max pooling layers all contain a 2x2 matrix for downsampling. After each successful set of a convolutional layer followed by a max pooling layer, the number of filters is increased by double to capture more complex features. After two sets of convolution and pooling layers, the flattened layer and the dense layers fully connect the learned features with the classification problem at hand. The last layer used the sigmoidal function to make both values of the two classification labels sum to one. The cut-off value is set to 0.5.

After the model has been built, it is now time to fit the training/validation data. The number of epochs listed within this example is 10 but experiments with other amounts were done to assess overfitting. Once the model is done with fitting the data, print statements allow for the testing loss and accuracy to be displayed for comparison.

Since the model has been built and hence saved, the *pneumoniaclasstest.py* puts the model to the test. Here, predictions were made on the testing dataset and compared to their actual classifications. With the use of *matplotlib*, the code allows for users to view the X-ray image, the probability of pneumonia, the predicted classification, and the actual classification.

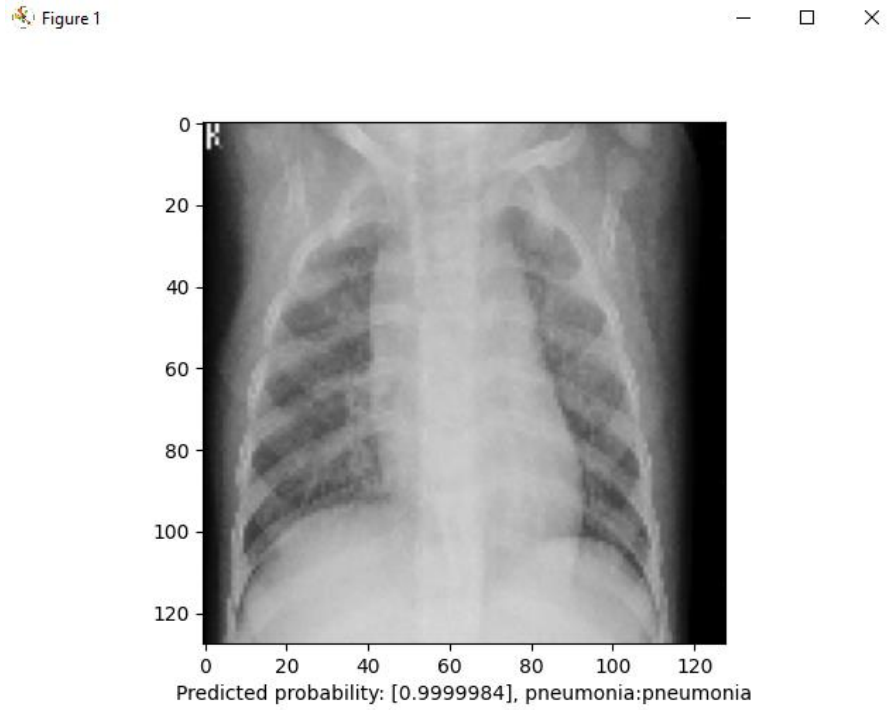


Figure 9: An X-ray image that has was classified as having pneumonia

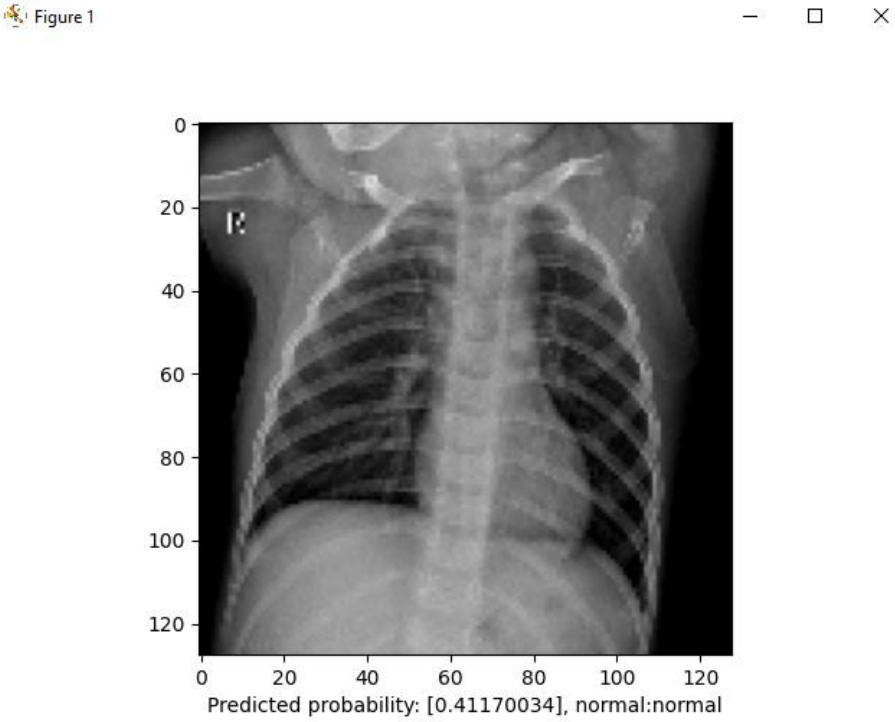


Figure 10: An X-ray image that has was classified as not having pneumonia

4. Conclusion

Convolutional neural networks have proven to be highly effective in performing classification tasks, especially when using features to classify what is depicted in images. The pneumonia example shows just how revolutionary neural networks are and the impact they could have in businesses such as the healthcare industry. Although the example was a binary classification model, convolutional neural networks can be extended to classify many diseases across multiple X-ray scans.

Like many machine learning models, neural networks have lots of room to be improved upon. The pneumonia example scratches the surface; hyperparameter tweaking and augmenting datasets can be done to improve the accuracy of both the training and validation sets. In this case, experimenting with the number of epochs, convolution/pooling layers, and stride/padding options could potentially improve the model's performance. As research continues to develop neural networks further, the impact of convolutional neural networks in image classification remains exceptional and promising.

References

Heaton, J. (2012). *Introduction to the Math of Neural Networks*. Heaton Research, Inc.

Shmueli, G., Bruce, P. C., Deokar, K. R., & Patel, N. R. (2023). *Machine learning for business analytics concepts, techniques, and applications with Analytic Solver® Data Mining*. Wiley.

Yamashita, R., Nishio, M., Do, R. K., & Togashi, K. (2018). Convolutional Neural Networks: An overview and application in Radiology. *Insights into Imaging*, *9*(4), 611–629. https://doi.org/10.1007/s13244-018-0639-9

Thakur, A., & Konde, A. (2021). Fundamentals of Neural Networks. *International Journal for Research in Applied Science and Engineering Technology*, *9*(VIII), 407–426. https://doi.org/10.22214/ijraset.2021.37362

National Institutes of Health Chest X-Ray Dataset. (2017, November 23). *Random sample of NIH chest X-ray dataset*. Kaggle. https://www.kaggle.com/datasets/nih-chest-xrays/sample

6. Table of Figures

7. Annex 1 - Code

7.1 File pneumoniaclass.py

```
# Necessary Packages
from keras import layers, models
from keras.utils import image_dataset_from_directory

# This function is preparing the input training set. Here, the labels are
introduced, the sizes are selected,
# and the grayscale modification is done.
train_data = image_dataset_from_directory('C:/Users/cactu/Downloads/archive
(5)/chest_xray/train',
                                          labels="inferred",
                                          label_mode='binary',
                                          image_size=(128, 128),
                                          class_names=["NORMAL", "PNEUMONIA"],
                                          color_mode='grayscale',
                                          seed=10)

# This function is preparing the input validation set. Here, the labels are
introduced, the sizes are selected,
# and the grayscale modification is done.
val_data = image_dataset_from_directory('C:/Users/cactu/Downloads/archive
(5)/chest_xray/test',
                                          labels="inferred",
                                          label_mode='binary',
                                          image_size=(128, 128),
                                          class_names=["NORMAL", "PNEUMONIA"],
                                          color_mode='grayscale',
                                          seed=10)

# This starts building a sequential model.
model = models.Sequential()

# The convolution and pooling layers are formed here. The parameters seen are
the number of filters, the kernel
# size, and the desired activation function. Within the pooling layers, the
parameter is the pooling matrix size.

# Note that there are two convolution layers that are all followed by max
pooling layers.
model.add(layers.Conv2D(16, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(32, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

# Here is the start of the fully connected layer. The data is squished into
one vector and fed into the dense network.
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dropout(0.2))
model.add(layers.Dense(1, activation='sigmoid'))


# The optimizer adam is used here so that learning rates can be adjusted to
```

```
speed up processing time.
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# The number of epochs can be changed here. The testing data was used for
validation because the validation set
# is very small. Instead, I will use the validation set for testing.
history = model.fit(
    train_data,
    epochs=10,
    validation_data=val_data
)

# The model is then saved to be used in the future.
model.save('img_class_pneumonia.model')
```

7.2 File pneumoniaclasstest.py

```
from keras import models
from keras.utils import image_dataset_from_directory
from tensorflow_datasets import as_numpy
import matplotlib.pyplot as plt
from keras.preprocessing.image import array_to_img

# This code loads the model and allows for input images to be fed into the
convolutional
# neural network for classification.
model = models.load_model('img_class_pneumonia.model')

# Setting up the testing dataset.
test_data = image_dataset_from_directory('C:/Users/cactu/Downloads/archive
(5)/chest_xray/val',
                                         labels="inferred",
                                         label_mode='binary',
                                         image_size=(128, 128),
                                         class_names=["NORMAL", "PNEUMONIA"],
                                         color_mode='grayscale',
                                         seed=10)

# This last bit outputs the loss and accuracy of the model when done fitting.
test_loss, test_acc = model.evaluate(test_data)
print(f'Test loss: {test_loss}')
print(f'Test accuracy: {test_acc}')


test_np = as_numpy(test_data)

# Getting predicted outputs for the test data.
predictions = model.predict(test_data)

# Using matplotlib, this code shows the image and it predicted and actual
classification. This is under the assumption
# that the cutoff value is 0.5.
for images_batch, label_batch in test_data:
```

```
    for i in range(len(images_batch)):
        img_array = images_batch[i]
        img_label = label_batch[i]
        print(label_batch)
        img = array_to_img(img_array)
        plt.imshow(img, cmap='gray')
        plt.axis('on')
        plt.xlabel("Predicted probability: {}, {}:{}".format
                   (predictions[i], "pneumonia" if predictions[i] >= 0.5
else "normal",
                    "pneumonia" if img_label == 1 else "normal"))
        plt.show()
        print("Predicted probability: {}".format(predictions[i]))
```