Big Data Analysis for Covid Detection

Weiting Lin, Yuxi Jiang, Hungta Chen, Jaehyeon Park

13 June 2023

1 Abstract

This project revolves around image classification using various types of lung X-ray images: those infected with viruses, those infected with Covid-19, and uninfected X-ray images. To begin with, we will employ the CLAHE method (Contrast Limited Adaptive Histogram Equalization) to enhance the image contrast and brightness. Additionally, we will resize the images to ensure consistent image sizes throughout the dataset. Moving forward, we will develop an image classification model using the powerful CNN (Convolutional Neural Network) architecture, and DNN (Deep Neural Networks) leveraging the capabilities of the tensorflow.keras package.
Lastly, we will explore an alternative approach by implementing the CNN and DNN algorithm using NumPy, allowing us to compare its performance against the results obtained using popular deep learning packages such as Tensorflow and PyTorch.

2 Introduction

The Covid-19 pandemic has had a profound impact on global public health, specifically concerning the respiratory system. Given the direct targeting of the virus on the lungs, the susceptibility to respiratory infections has become particularly heightened. However, the ability to differentially diagnose lung X-ray images depicting viral infections from those specifically indicative of Covid-19 presents a paramount challenge. This differentiation is paramount in facilitating precise diagnostic outcomes and ensuring appropriate therapeutic interventions.

To address this formidable challenge, the present study endeavors to comprehensively classify diverse lung X-ray images encompassing viral infections, Covid-19 infections, and uninfected instances. By harnessing the potential of cutting-edge Convolutional Neural Networks (CNNs) and Deep Neural Networks (DNNs), this research aims to establish an efficient image classification system that significantly enhances the diagnostic accuracy of Covid-19 cases based on X-ray imaging.

In order to facilitate this research endeavor, an extensive dataset comprising distinct files housing uninfected lung X-ray images, viral-infected lung X-ray images, and Covid-19-infected lung images was procured from the Kaggle platform. Capitalizing on this invaluable resource, the proposed methodology entails the development of a sophisticated image classification system that effectively harnesses the discriminatory capabilities of CNNs and DNNs. These advanced neural network architectures have demonstrated their efficacy in extracting salient features from medical images, yielding exceptional classification accuracy.

By leveraging the available dataset and employing state-of-the-art deep learning techniques, this study aims to make a noteworthy contribution to the medical community by augmenting the precision of Covid-19 diagnosis through meticulous analysis of lung X-ray images. The successful realization of a robust image classification system holds tremendous potential in empowering healthcare professionals to make well-informed decisions, enabling timely interventions, and significantly mitigating the incidence of misdiagnosis associated with respiratory infections amidst the Covid-19 pandemic.

3 Proposed method

Due to the reason that our dataset consists in chest X-ray images, and the purpose is to classify the images in three categories, Covid-19 infected images, virus infected images, normal chest images.

Based on the purpose and data type, we utilize CNNs (Convolutional Neural Networks) and DNNs (Deep Neural Networks) to achieve our goal. For CNNs, we employ various optimizer methods to identify the most suitable model.
Before building model for image classification, we will first do data pre-processing to make sure the images are normalized. The method we employed here is CLAHE method (Contrast Limited Adaptive Histogram Equalization).

3.1 CLAHE

Contrast Limited Adaptive Histogram Equalization (CLAHE) is a technique utilized to enhance the contrast in image. The way CLAHE works is dividing the image into smaller parts and analyzing the pixel intensities in each part. It then adjusts the pixel intensities to make the dark part darker and the bright part brighter. This procedure aims to bring out more information and make the image look more vibrant, and close to the reality.

However, this method may make image look weird due to excessive contrast amplification. To avoid over-amplifying the differences in contrast level, CLAHE sets the limit on the range of pixel intensities it adjusts. This method is also known as clipping level or the maximum slope, and it helps maintain visual appeal and natural appearance of the image.

Besides, we also adjust the brightness after enhancing contract level, it is listed as the third picture. To fulfill the CLAHE process, we use cv2 package in python.

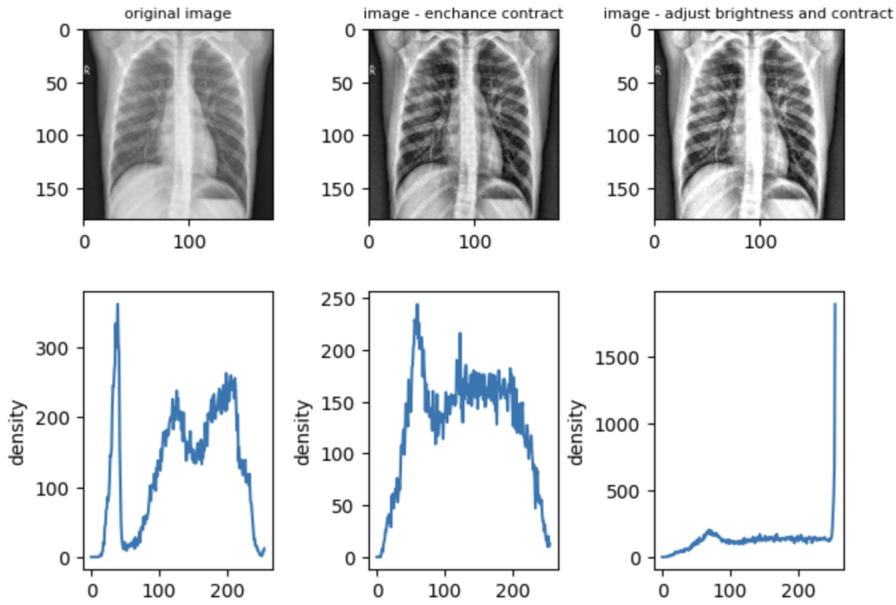

Figure 1: original X-ray image versus image after adjustment

The algorithm behind the CLAHE is Histogram Equalization. The following bullet points are the knowledge that we will used in the Histogram Equalization.Histogram in following context represents actual number of occurrences of each intensity value.

- $I(x, y)$ be the pixel intensity at coordinates (x, y) in the input image.
- $I'(x, y)$ be the intensity of the corresponding pixel in the equalized image.
- $H(i)$ be the histogram of pixel intensities in the input image.

- $cdf(i)$ be the cumulative distribution function of the histogram.

The Histogram Equalization algorithm can be summarized using the following equations:
Compute the histogram of the input image:

$$H(i) = \text{count of pixels with intensity } i$$

Compute the cumulative distribution function (CDF) of the histogram:

$$cdf(i) = \sum_{j=0}^{i} H(j)$$

Normalize the cdf to obtain the transformation function:

$$I'(i) = \frac{cdf(i) - cdf_{\min}}{(M \times N) - cdf_{\min}}$$

where M and N are the dimensions of the image, and $cdf_{\min}$ is the minimum non-zero CDF value. This step is used to make sure that the CDF values lie between 0 and 1, and it represents the probability of corresponding pixel intensities
Map the intensity values of each pixel using the transformation function:

$$T'(x, y) = T(I(x, y))$$

The transformation function $T(i)$ is applied to each pixel's intensity value in the original image, the range will be 0 to 255 based on the maximum intensity level of 8-bit images. This mapping operation causes a new equalized image $I'(x, y)$, where $I(x, y)$ is the intensity value of the pixel at position (x, y) in the original image. Eventually, you could normalize the intensities of the equalized image to the desired output range if needed.

3.2 Optimizers from CNNs with tensorflow.keras

- **Adadelta**: Adadelta is a modification of Adagrad that aims to address the issue of a monotonically decreasing learning rate. Unlike Adagrad, which accumulates all past squared gradients, Adadelta keeps track of only a fixed window size of accumulated past gradients.

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2$$

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta]_t + \epsilon}$$

$$\Delta\theta_t = \frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$

  where,
  γ - Usually around 0.9
  ϵ - Smoothing term that avoids division by zero and is usually of the order of $1e-8$

- **RMSprop**: RMSprop is an adaptive learning rate technique, introduced by Geoff Hinton, which has not been officially published. The core concept involves dividing the gradient by a moving average of its recent magnitude. While it shares similarities with Adadelta, RMSprop was independently developed to address the limitations of the Adagrad algorithm.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

where,
γ - Usually around 0.9
η - Learning rate, usually around 0.001
$E[g^2]_t$ - Decaying average of the past squared gradients at time step t.

- **Adam**: Adam, short for Adaptive Moment Estimation, is a technique that calculates adaptive learning rates for individual parameters. It incorporates two key components: the decaying average of previous gradients (similar to momentum) denoted as "m_t," and the decaying average of squared gradients (similar to RMSprop and Adadelta) denoted as "v_t." By combining the strengths of both methods, Adam provides a comprehensive optimization approach. Consequently, it has become the default optimizer choice for a wide range of applications.

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\Delta\theta_t = -\frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t$$

where,
β_1 - usually 0.9
β_2 - usually 0.999
η - Learning rate
ϵ - usually of the order of $1e-8$

- **SGD**: Stochastic gradient descent (SGD) updates parameters based on individual training examples in the training dataset. It is unable to take advantage of vectorization because it needs to iterate through each training example and perform an update for each one. As a result, it exhibits significant fluctuations before eventually converging to the solution.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

where,
J - represents the cost function that gradient descent minimizes by iteratively adjusting model parameters

- **Adagrad**: AdaGrad is an optimization technique that enables varying step sizes for different features. It enhances the impact of rare yet informative features by adapting the learning rate based on the parameters, resulting in larger updates for infrequent parameters and smaller updates for frequent ones. This characteristic makes AdaGrad particularly effective in handling sparse data.

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t+\epsilon}} \odot g_t$$

where,
G_t - Sum of the squares of the past gradients w.r.t all parameters θ along its diagonal
$\odot$ - Matrix-vector dot product
g_t - Gradient at time step t
η - Learning rate
ϵ - Smoothing term that avoids division by zero and is usually of the order of $1e-8$

3.3 Activation function from CNN model with tensorflow.keras

- ReLU (Rectified Linear Unit): converge fast and comutationaly cheap. It's defined as:

$$ReLU(x) = \begin{cases} X & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

- Softmax: convert input vector of real numbers into a probability distribution in which each element represents the probability of the corresponding class.
  It's defined as

$$\sigma(\overrightarrow{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

  Notations:
  $\sigma(\overrightarrow{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$
  σ = softmax function
  $\overrightarrow{z}$ = probability of the input vector
  k = number of class label for multi-class classifier
  e^{z_i} = standard exponential function for input vector
  e^{z_j} = standard exponential function for output vector

3.4 Implementation of CNN from scratch

The other part of our project is the implementation of CNN from scratch. The implementation part can be mainly divided into two parts, including forward propagation and backpropagation. Let's take a CNN model with a convolution layer, a maxpooling layer, a fully-connected layer with activation and a softmax layer for example to illustrate the math behind this algorithm.

Forward Propagation

- **Convolution Layer**

  In the convolutional layer, we perform convolutions between the input feature maps and learnable filters to generate output feature maps. Denote the input feature map as X, the learnable filters as X, the bias term as b, and the output feature map as Z. The formulas for forward propagation in the convolutional layer is:

$$Z = W * X + b$$

  where $*$ denotes the convolution operation.

- **Max Pooling Layer**

  The max pooling layer downsamples the feature maps by selecting the maximum value within each pooling region. Denote the input feature maps as X and output feature maps after max pooling as P.The formula for forward propagation in the max pooling layer is:

$$P_{i,j,k} = \max\left(X_{i',j',k}\right)$$

  where i', j' represents the indices of the pooling region centered at (i, j), and k represents the index of channel.

- **Fully Connected Layer**

  Denote the input to the fully connected layer as X (here the input is the flattened vector of feature maps), the weight matrix as W, the bias term as b, and the output as Z. The formula for forward propagation in the fully connected layer is:

$$Z = WX + b$$

.

- **Activation Layer**

  The activation layer applies an activation function f element-wise to the output feature maps. Let's denote the activated feature maps as A. The formula for forward propagation in the fully connected layer is:

$$A = f(Z)$$

.

- **Softmax Layer**

  The softmax layer is commonly used for multi-class classification problems. It takes the input Z and computes the probability distribution over the classes. Denote the input to the softmax layer as Z, the output as S, and the number of classes as C. The formulas for forward propagation in the softmax layer are:

$$S_j = \frac{e^{Z_j}}{\sum_{i=1}^{C} e^{Z_i}},\ j = 1, 2, ..., C$$

  where S_j represents the jth element of the softmax output and S is a vector with length C.

- **Cross-Entropy Loss**

  The cross-entropy loss is commonly used as a loss function for multi-class classification tasks. It measures the dissimilarity between the predicted class probabilities and the true class labels. Denote the true class labels as Y (one-hot encoded) and the predicted class probabilities as S. The formula for forward propagation in the cross-entropy loss is:

$$L = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{C} Y_{i,j} \log\left(S_{i,j}\right)$$

  where N represents the number of samples in the batch, C represents the number of classes, $Y_{i,j}$ represents the true label for sample i and class j, and $S_{i,j}$ represents the predicted probability for sample i and class j.

Backpropagation

Based on the chain rule, we calculate the gradient in the reversed order from the loss function.

- **Cross Entropy Loss and Softmax Layer**

  In the softmax layer, the backward pass involves computing the gradients with respect to the input Z based on the gradients with respect to the softmax output S.

$$\frac{\partial S_i}{\partial Z_j} = \begin{cases} S_i(1 - S_i) & \text{if } i = j \\ -S_i S_j & \text{otherwise} \end{cases}$$

  where $i, j = 1, 2, \ldots, C$.

  Using this result, we can compute the gradients of the cross-entropy loss with respect to the input Z. The formula for backpropagation in the softmax layer and cross-entropy loss is:

$$dZ_j = S_j - Y_j$$

  where dX_j represents the gradient with respect to the j-th element of the input Z.

- **Activation Layer**

  In the activation layer, the backward pass involves computing the gradients with respect to the activated output A and propagating them back to the previous layers. The formula for backpropagation in the activation layer is:

$$dZ = dA \odot f'(Z)$$

where $\odot$ denotes element-wise multiplication, and $f'(\cdot)$ represents the derivative of the activation function.

- **Fully Connected Layer**

  In the fully connected layer, the backward pass involves computing the gradients with respect to the weight matrix W, the bias term b, and the input to the dense layer X. The formula for backpropagation in the fully connected layer is:

$$dX = W^\top dZ$$
$$dW = X^\top dZ$$
$$db = \sum dZ$$

  where dX represents the gradients with respect to the input to the fully connected layer and dW represents the gradients with respect to the weight matrix.

- **Max Pooling Layer**

  In the max pooling layer, the backward pass involves distributing the gradients from the output feature maps P to the input feature maps X based on the locations of the maximum values. The formula for backpropagation in the max pooling layer is:

$$dX_{i',j',k} = \begin{cases} dP_{i,j,k} & \text{if } X_{i',j',k} = \max X_{i',j',k} \\ 0 & \text{otherwise} \end{cases}$$

  for i', j' in pooling region centered at (i, j).

- **Convolutional Layer** In the convolutional layer, the backward pass involves computing the gradients with respect to the learnable filters W, the bias term b, and the input feature maps X. The formula for backpropagation in the convolutional layer is:

$$dX = W^\top * dZ$$
$$dW = X * dZ$$
$$db = \sum dZ$$

  where $*$ denotes the convolution operation, dX represents the gradients with respect to the input feature maps, dW represents the gradients with respect to the filters, and db represents the gradients with respect to the bias term.

Further implementation details will be included in Appendix.

4 Real data study

4.1 Result with X-ray dataset

CNN model using tensorflow.keras:
As shown in Figure 1, we observe that

- **RMSprop:** The accuracy rate starts from 0.9983 in the first epoch and stays at 1 till the end of the 50 epochs. On the other hand, the loss value starts at 0.0383 and stays around 0 for most of the epochs.
- **SGD:** The accuracy rate starts from 0.9286 in the first epoch and quickly increases to 1 over the course of 50 epochs. The loss value starts at around 0.6425 and gradually decreases and fluctuates around 0.5.
- **Adadelta:** The accuracy rate starts at 0.9939 at first and increases all the way to 1 in the 2^{nd} epoch. The loss value starts from 0.0453, decreases drastically in the first few epochs and fluctuates around 0 at the end.
- **Adagrad:** The accuracy rate starts at 0.9817 and rapidly increases to 1 throughout the 50 epochs. The loss value starts from 0.9830 and stays around 0.1 to 0.2 for the most of time.
- **Adam:** The accuracy rate starts from 0.9904 and stays above 1 the whole time. The oss value starts from 0.0852, first drastically decreases to 0.0018 and stays around 0 afterwards.

Accuracy and loss rate of CNN model using tensorflow.keras paired with 5 optimizers					
	RMSprop	SGD	Adadelta	Adagrad	Adam
Accuracy(first epoch)	0.9983	0.9286	0.9939	0.9817	0.9904
Accuracy(final epoch)	1	1	1	1	1
Loss(first epoch)	0.0383	0.6425	0.0453	0.9830	0.0852
Loss(final epoch)	$2.3842*10^{-7}$	0.6931	$1.0417*10^{-6}$	$2.3842*10^{-7}$	$2.3842*10^{-7}$

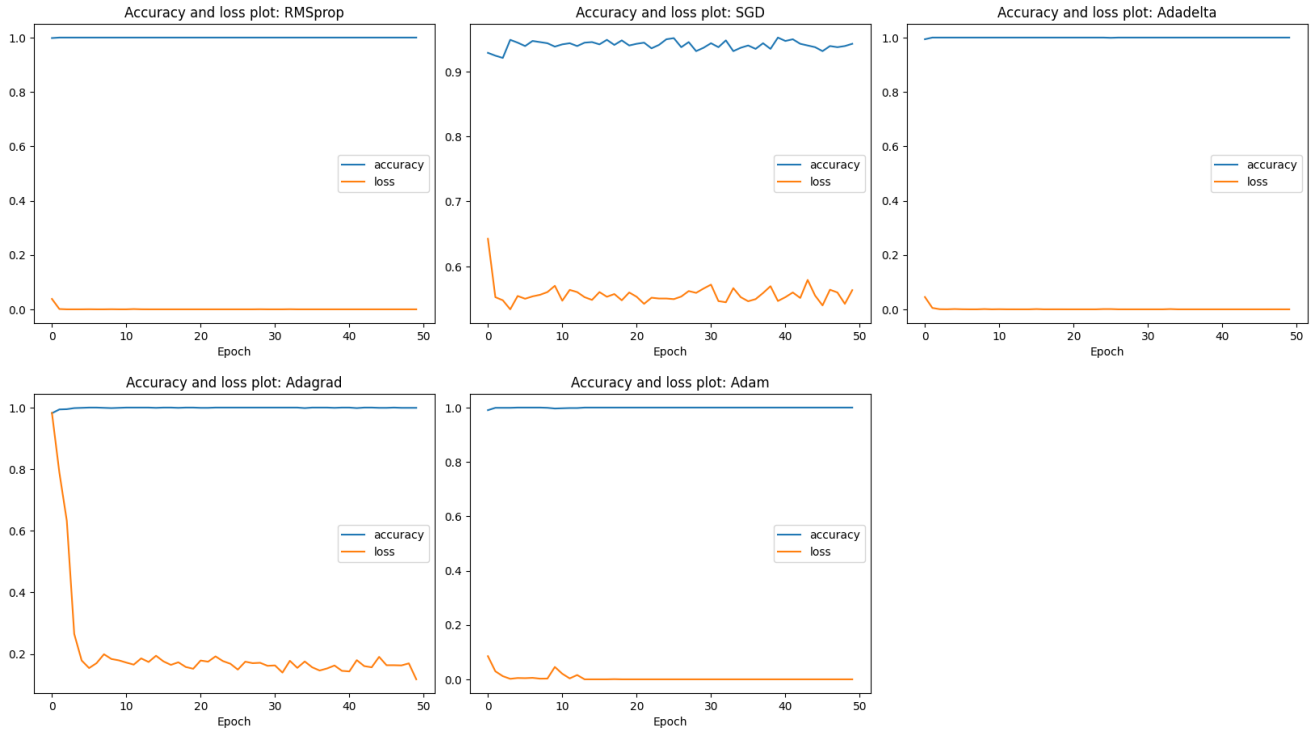

Figure 2: Accuracy and loss plots using different optimizers

4.2 Simulated data study with MNIST dataset

To see if our implementation of convolutional neural networks is valid, we test it on a simplified version of MNIST dataset. We only use digit 0 to 4 in this test, and each digit has 2000 examples for training. Also, all images are downsampled to 14*14 for a shorter training time.

DNN model from scratch:
We first construct a simple DNN model, including 2 fully connected layers and a ReLU activation layer, using layers implemented by us. After 10 epochs of training, we get a testing accuracy of 0.9628.

CNN model from scratch:
We then construct a CNN model, including 1 convolutional layer, 1 max pooling layer, 2 fully connected layers, and a ReLU activation layer, using layers implemented by us. After 10 epochs of training, we get a testing accuracy of 0.8644.

DNN using PyTorch:
In order to see if the performances of our implementation are valid, we compare them with other popular deep learning packages. We construct a simple DNN model using the PyTorch package, setting all hyperparameters identical to our own model. After 10 epochs of training, we get a testing accuracy of 0.9624, which is virtually identical to the testing accuracy of our own model. Therefore, our implementation indeed has the capability of handling simple DNN.

CNN using Tensorflow:
Lastly, We construct a CNN model using the Tensorflow package, setting all hyperparameters identical to our own model. The loss value decreases all the way from 1.5903 to 0.2370 during the training process. The accuracy rate also improves from as low as 0.2880 to 0.9431. Ultimately, we get a testing accuracy of 0.9528, which is obviously higher than the testing accuracy of our model. The reason behind this can be a lack of optimization in our implementation. Since convolution involves a large amount of computation, our naive model takes a way longer time to train. Also, further optimization can help with feature extraction, which can lead to a better performance.

5 Conclusion

In conclusion, we aimed to perform image classification on various types of lung X-ray images, including those infected with viruses, those infected with Covid-19, and uninfected images. The CLAHE method enhanced image contrast and brightness while resizing ensured consistent image sizes within the dataset. The image classification models were developed using CNN architecture and Depthwise Separable CNNs, utilizing the tensorflow.keras package. The results of the experiment demonstrated the effectiveness of the implemented models. The CNN models trained with different optimizers, such as RMSprop, SGD, Adadelta, Adagrad, and Adam, consistently achieved high accuracy rates, with some models reaching 100% accuracy. These results indicate the effectiveness of CNN models in accurately classifying lung X-ray images. The utilization of large-scale datasets with diverse and representative samples further contributed to the models' performance. Therefore, we successfully achieved our goal of accurate image classification in this project.

Additionally, our implementation of the DNN model achieved a testing accuracy of 96.28%, which is almost identical to the testing accuracy of the model implemented by the popular PyTorch package. Also, our implementation of the CNN model achieved a testing accuracy of 86.44%, while the CNN model implemented with TensorFlow yielded a testing accuracy of 95.28%
According to the result, it is evident that all the models we implemented from scratch yield a decent performance in contrast to the models implemented with popular packages, which undoubtedly demonstrates the capability of handling simple image classification tasks. Given such performance, we believe our goal for the algorithm implementation part is also achieved.

6 Reference

- Siddhartha, M. (2021, July 19). Covid CXR Image Dataset (Research). Kaggle.
https://www.kaggle.com/datasets/sid321axn/covid-cxr-image-dataset-research
- Siddhartha, M. (2021a, July 19). Covid CXR Image Dataset (Research). Kaggle.
https://www.kaggle.com/datasets/sid321axn/covid-cxr-image-dataset-research - Awesome drawing tools for neural net architecture: Data science and machine learning. Kaggle. (n.d.). https://www.kaggle.com/getting-started/253300
- Optimization Algorithms. Optimization Algorithms - A Brief Overview:- · GSoC'18 @ CERN. (n.d.).
https://www.sravikiran.com/GSOC18//2018/05/16/optimizers/
- JAUMIER, P. (2019, July 17). Backpropagation in a convolutional layer. Medium.
https://towardsdatascience.com/backpropagation-in-a-convolutional-layer-24c8d64d8509
- Liu, D. (2017, November 30). A practical guide to relu. Medium.
https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7
- Cross-entropy loss: Everything you need to know. Pinecone. (n.d.).
https://www.pinecone.io/learn/cross-entropy-loss/
- OpenAI. (n.d.). https://chat.openai.com/chat
- CLAHE (Contrast Limited Adaptive Histogram Equalization). Clahe (Contrast Limited adaptive histogram equalization). (2017, November 27).
https://amroamroamro.github.io/mexopencv/opencv/clahe_demo_gui.html
- Wikimedia Foundation. (2022, June 29). Histogram equalization. Wikipedia.
https://en.wikipedia.org/wiki/Histogram_equalization

7 Appendix

Following are some additional implementation details. We use object-oriented programming to better implement the algorithms

7.1 Convolutional layer

```
class ConvolutionalLayer(Layer):
    def __init__(self, filters, kernel_size):
        super().__init__()
        self.num_filters = filters
        self.kernel_size = kernel_size
        self.weights = None
        self.bias = None


    def initialize(self, input_shape):
        num_channels = input_shape[0]
        self.weights = np.random.randn(self.num_filters, num_channels, self.kernel_size,
    self.kernel_size)
        self.bias = np.zeros((self.num_filters, 1))


    def forward(self, input_data):
        self.input = input_data
        self.initialize(self.input.shape)
        num_channels, input_height, input_width = input_data.shape
        output_height = input_height - self.kernel_size + 1
        output_width = input_width - self.kernel_size + 1
        self.output = np.zeros((self.num_filters, output_height, output_width))

        for f in range(self.num_filters):
            for i in range(output_height):
                for j in range(output_width):
```

```
                    receptive_field = input_data[:, i: i + self.kernel_size, j:j + self.
    kernel_size]
                    self.output[f, i, j] = np.sum(receptive_field * self.weights[f]) + self.
    bias[f]

        return self.output


    def backprop(self, output_grad, learning_rate):
        num_channels, input_height, input_width = self.input.shape
        num_filters, output_height, output_width = self.output.shape

        weight_grad = np.zeros_like(self.weights)
        bias_grad = np.zeros_like(self.bias)
        input_grad = np.zeros_like(self.input)

        for f in range(num_filters):
            for i in range(output_height):
                for j in range(output_width):
                    receptive_field = self.input[:, i: i + self.kernel_size, j: j + self.
    kernel_size]
                    weight_grad[f] += receptive_field * output_grad[f, i, j]
                    bias_grad[f] += output_grad[f, i, j]
                    input_grad[:, i: i + self.kernel_size, j: j + self.kernel_size] += self.
    weights[f].T * output_grad[f, i, j]

        self.weights -= learning_rate * weight_grad
        self.bias -= learning_rate * bias_grad

        return input_grad
```

- **Initialization**

  We randomly initialize the weights and biases of the filters.

- **Forward**

  We generate the output feature maps by scanning through the input images with our kernels and computing the convolution by doing element-wise multiplication in the receptive field in each iteration.

- **Backprop**

  We compute the convolution in the same way as the forward part, using it to calculate the gradient of weights, biases, and input according to the formula provided in section 3.4. We then update the weights and biases using their corresponding gradient and return the gradient of input.

7.2 Max pooling layer

```
class MaxPoolingLayer(Layer):
    def __init__(self, pool_size):
       super().__init__()
       self.pool_size = pool_size


    def forward(self, input_data):
        self.input = input_data
        num_channels, input_height, input_width = input_data.shape
        pool_height = input_height // self.pool_size
        pool_width = input_width // self.pool_size
        self.output = np.zeros((num_channels, pool_height, pool_width))

        for c in range(num_channels):
            for i in range(pool_height):
                for j in range(pool_width):
                    receptive_field = input_data[c, i*self.pool_size : (i+1)*self.pool_size,
     j*self.pool_size : (j+1)*self.pool_size]
```

```
                    self.output[c, i, j] = np.max(receptive_field)

        return self.output


    def backprop(self, output_grad, learning_rate):
        num_channels, input_height, input_width = self.input.shape
        pool_height = input_height // self.pool_size
        pool_width = input_width // self.pool_size
        input_grad = np.zeros_like(self.input)

        for c in range(num_channels):
            for i in range(pool_height):
                for j in range(pool_width):
                    receptive_field = self.input[c, i*self.pool_size : (i+1)*self.pool_size, j*self.pool_size : (j+1)*self.pool_size]
                    max_value = np.max(receptive_field)
                    mask = (receptive_field == max_value)
                    input_grad[c, i*self.pool_size : (i+1)*self.pool_size, j*self.pool_size : (j+1)*self.pool_size] = mask * output_grad[c, i, j]

        return input_grad
```

- **Forward**

  This is a process of downsampling. We separate the input feature maps into several pools and then generate the output feature maps by finding the maximum value in each pool.

- **Backprop**

  According to the formula provided in section 3.4, we only need the gradients of those maximum values in the input feature maps. Therefore, we generate a mask, where only the position of the max value is 1, for each pool. By doing the element-wise multiplication of the mask and the gradient, we can keep the gradients of those maximum values and eliminate the others.

7.3 Fully connected layer

```
class FullyConnectedLayer(Layer):
    def __init__(self, input_size, output_size):
        super().__init__()
        self.weights = np.random.randn(input_size, output_size) / np.sqrt(input_size)  # Xavier initialization
        self.biases = np.zeros((1, output_size))

    def forward(self, input_data):
        self.input = input_data
        self.output = np.dot(input_data, self.weights) + self.biases
        return self.output

    def backprop(self, output_grad, learning_rate):
        input_grad = np.dot(output_grad, self.weights.T)
        weight_grad = np.dot(self.input.T, output_grad)
        bias_grad = np.sum(output_grad, axis=0, keepdims=True)

        self.weights -= learning_rate * weight_grad
        self.biases -= learning_rate * bias_grad

        return input_grad
```

- **Initialization**

  We randomly initialize the weights and biases of the layer.

- **Forward**

  We generate the output vector by computing the matrix product of the input and the weights and adding the biases.

- **Backprop**

  We calculate the gradient of weights, biases, and input according to the formula provided in section 3.4. We then update the weights and biases using their corresponding gradient and return the gradient of input.

7.4 Flatten layer

```
class FlattenLayer(Layer):
    def __init__(self):
        super().__init__()

    def forward(self, input_data):
        self.input = input_data
        flattened_size = np.prod(input_data.shape)
        self.output = input_data.reshape(1, flattened_size)
        return self.output

    def backprop(self, output_grad, learning_rates):
        input_grad = output_grad.reshape(self.input.shape)
        return input_grad
```

- **Forward**

  We flatten the input feature maps using ‘reshape()‘ method.

- **Backprop**

  We reshape the gradient into the same shape as the input data and return it.

7.5 Neural Network

```
class Network:
    def __init__(self, loss, loss_prime):
        self.layers = []
        self.loss_func = loss
        self.loss_prime = loss_prime

    def add_layer(self, layer):
        self.layers.append(layer)

    def predict(self, input_data):
        sample_size = len(input_data)
        result = []

        for i in range(sample_size):
            output = input_data[i]

            for layer in self.layers:
                output = layer.forward(output)

            output = softmax(output)  # Apply softmax activation
            result.append(output)

        return np.array(result)

    def train(self, x_train, y_train, epoch_num, lr):
        sample_size = len(x_train)
        error_list = []
        print("==== Start training ====")

        for epoch in range(epoch_num):
            err = 0
            output_list = []
            correct = 0
```

```
34
35            # Shuffle the training data and labels
36            shuffled_indices = np.random.permutation(sample_size)
37            x_train_shuffled = x_train[shuffled_indices]
38            y_train_shuffled = y_train[shuffled_indices]
39
40            for i in range(sample_size):
41                output = x_train_shuffled[i]
42
43                # Reshape input if it is 1D
44                if output.ndim == 1:
45                    output = output.reshape(1, -1)
46
47                # Forward propagation
48                for layer in self.layers:
49                    output = layer.forward(output)
50
51                output_list.append(output)
52                err += self.loss_func(y_train_shuffled[i], output)
53                print(f"{epoch+1}-{i}th sample")
54                print(f"true: {np.argmax(y_train_shuffled[i])}, pred: {np.argmax(output)}")
55
56                # Backpropagation
57                grad = self.loss_prime(y_train_shuffled[i], output)
58                #print(grad)
59                for layer in reversed(self.layers):
60                    grad = layer.backprop(grad, lr)
61
62            err /= sample_size
63            error_list.append(err)
64
65            print('epoch %d/%d   loss=%f' % (epoch + 1, epoch_num, err))
66
67        return error_list
```

- **Add layer**

  We use a member list 'layers' to aggregate all layers added to the model. When a new layer is passed into the model, we append it into 'layers'.

- **Predict**

  We pass each input sample through the layers and return all outputs as a list.

- **Train**

  We implement the training process using the SGD method. In each epoch, we randomly shuffle the dataset and consider each example as an independent batch. For each batch, we forward the batch through all the layers add to compute the loss, and then we pass the gradient through all the layers again but in the reversed order to update the parameters of each layer.