Project 6

Gaurav Shilpakar

Pranesh Shrestha

Raju Khadka

Sohan Khadka

May 2020

Description:

Image classification of C.Elegans using Convolutional Neural Networks.

Dataset:

20000 Images of worms and noworms into two folders.

Network Architecture:

A screenshot of a cell phone

Description automatically generated

Figure 1 Network Architecture

Convolutional Neural Networks:

Convolutional Neural Network convolves kernels around input data for feature extraction. These networks are generally used for feature extraction from image file as it conserves the original spatial dimension of the image.

For this project, a single layer of CNN is used with 8 kernels measuring 3 pixels width and 3 pixels height. Weights for these kernels are initialized using Xavier Uniform Initialization,

Uniform Distribution = (-limits, limits)

This initializing technique ensures that the mean and standard deviation remain constant and does not affects the result during optimization.

ReLU Activation has been applied after the convolution layer for non-linearity. This non-linear function picks the max of input with zero and can be represented as:

Maxpooling Layer:

For Reducing the number of training parameter, the output image from convolution layers is reduced using maxpooling technique which creates the representational image of the input image. For this project pool shape of 2pixel width and 2pixel height is chosen with 2 strides in both direction ensuring the output image to be half of the original image.

Feed Forward Layer (Dense):

The output from the Maxpool layer is flattened since feed forward layer can only take 1 dimensional array as feature then passed through two Dense layers for further feature extraction and classification. First Dense layers consist of 32 units with activation function ReLU, the input shape for this layer equals the scaler product of the dimension of the output from Maxpooling layer. Second dense layer has 2 units with softmax as output activation function because worms will be classified into two classes.

y = predicted score, w = weight, j = number of classes, n = number of samples

Cross entropy loss function is used for optimization as this is a classification problem.

t = Target label, y = predicted label

Vectorization for Convolution and Maxpooling:

Naive implementation of convolution and maxpooling consists of atleast four For-Conditional loops which ridiculously increases the code execution time. To prevent slower code runtime, the image has been vectorized both before convolution and Maxpooling.

During convolution, the proposed region is extracted and then flatten to 1 dimensional vector for doing the matrix dot product with flattened filter. The resultant matrix is reshaped to the pre-determined output shape that can be calculated using input image, kernel size and stride. Same process. Proposed region extraction and flattening is again done for maxpooling for finding the maximum value of the proposed region. This implementation drastically reduces the code execution time. For our implementation, it resulted at least 100 time faster than naïve implementation.

Training and Hyperparameter Selection:

Back-Propagation algorithm is used for training this network. In this algorithm, the gradient of the output loss is chained until input layer for updating weights and biases for each layer.

Vanilla Gradient Descent and Stochastic Gradient Descent is not suitable for this application, as first one requires more memory and the other being too slow and inaccurate for this application so adaptive learning optimization has been used.

Adaptive learning optimization is very efficient for training neural networks because they do not require larger memory and can quickly reduce the cost then vanilla gradient descent.

For this application, Adam optimization technique has been used, it is probably most popular adaptive learning technique on practice which updates the weights using the average of the squared of the moving gradient, it can be represented as:

beta1 = 0.9, beta2 = 0.999 (Kingma & Ba, 2015)

beta values are defined in the original research paper for Adam optimizer and Hyperparameter selection is done through trial and error. Batch size is selected small enough to run in the memory available in local machine.

**Dataset Generation**

**Global Binarization (OpenCV)**

Initially, the C. Elegans images were binarized using global binarization. For this, a global value for threshold was used. It was set at 200 such that and value below 200 was set to 0 and the rest was set to 255. This value remained constant throughout all the images. But the total images had variations in lighting conditions so differences in brightness and contrast was visible. Because of this, the global binarization technique failed on most of the original images.

**Adaptive Binarization (OpenCV)**

Because of the lighting differences, there were irregularities in the binarized images and in finding the contours. Thus, adaptive binarization was used. For this, the threshold value is calculated for smaller regions of the image. This causes variable threshold with change in luminance values of the image. Gaussian Adaptive method was used to select the threshold value. In this method, a constant value is subtracted from the weighted sum of a certain block size from a point to return the threshold separator. For our purposes, the following values were used:

1. BlockSize = 99
2. ConstantValue = -35

Even after adaptive thresholding, variable lighting caused numerous binarization issues. So, method of contrast stretching was further utilized.

**Contrast Stretching**

To uniformly distribute the contrast throughout the whole set of images, contrast stretching was considered. First the mean value of all the individual images were calculated and stored. Then the absolute mean of these mean values was calculated. This was obtained as 163.03

Then, each image was processed again. If the mean of the individual image was higher than that of the absolute mean, difference between the means was subtracted and vice versa. After contrast stretching, the adaptive binarization algorithm worked effectively. So, it was adapted, and contours were calculated from the binarized image.

**Finding Contours (OpenCV)**

The findContours function from OpenCV library returned a numpy array of X-coordinate, Y-coordinate, width, and height of the bounding box. This function was adapted on the binarized image for optimum results. Then, if the area of the bounding box was less than 400 pixels or greater than 4900 pixels, the box was disregarded.

A small set of bounded images with classification for NoWorm and Worm were recorded first for an initial test. The mean values of each image from these classes were recorded. The mean value for the Worm class was observed to be between 40 and 85. This was used as a basis for classifying the data between two different labels.

Finally, the boxed images were padded or cropped to a constant size of 40x40 using a recursive algorithm. It checked if the return image was of the desired dimensions. This made the learning of the images during the training process easier.

**Data Augmentation**

The original dataset only contained 4135 (NoWorm) and 6293 (Worm) examples. To assist weight updates and the learning process, data augmentation was picked up to multiply the training examples. The following techniques were adapted for increasing the training samples:

1. Vertical Flip
2. Horizontal Flip
3. Clockwise Rotation (-90)
4. Anti-Clockwise Rotation (90)

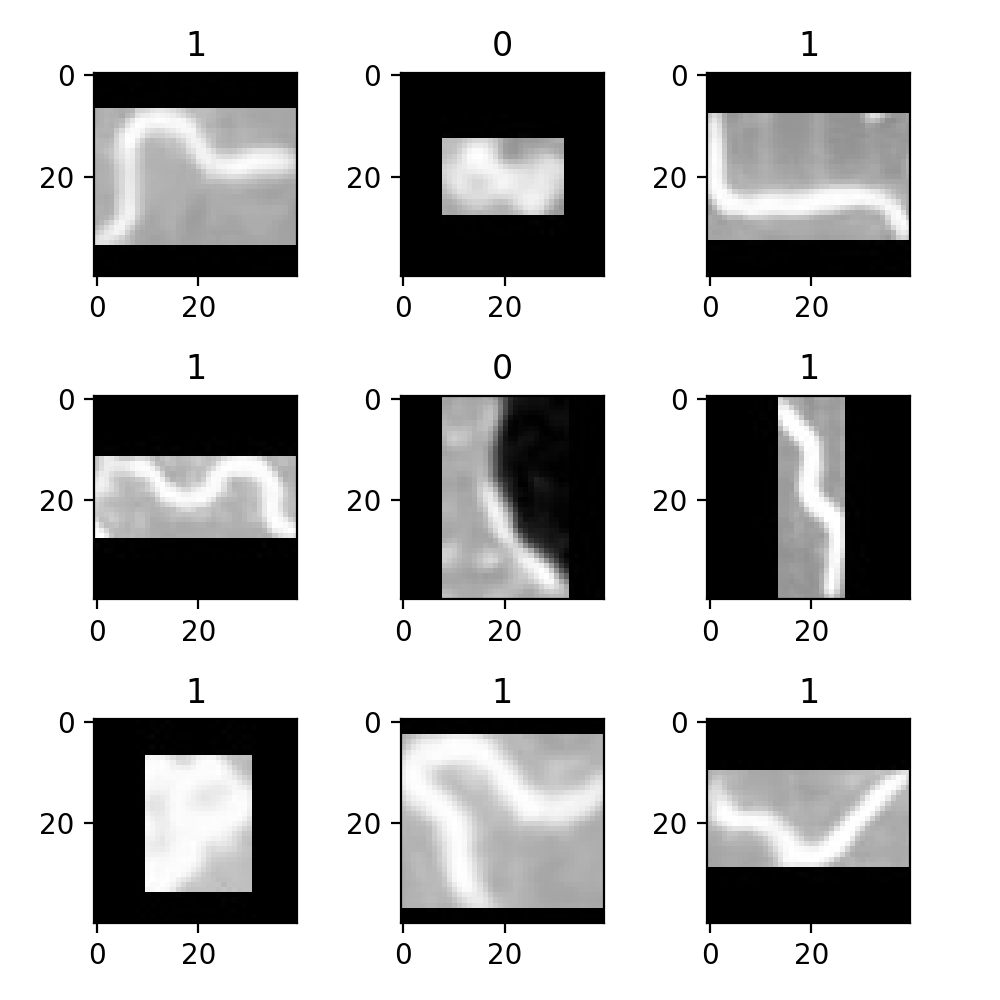


Figure 2 C.Elegans

Translation or addition of noise were skipped with a hypothesis that such indifferences would cause irregularities in the weight updates. The dataset is currently populated with 10,000 of each class for Worm and NoWorm. The Worm class is labelled as ‘1’ and the NoWorm class is labelled as ‘0’.

Loading data and Training:

We have decided to do binary classification for this problem i.e. Worm and No Worm considering the amount of data generated. For that reason, preprocessed image has been saved in two separate folders for easily labeling and generating training and testing data. The images are loaded to a numpy array using scikit-learn in grayscale from the both the folders and concatenated. Image label equaling the size of No Worm and size of Worm are concatenated in similar fashion. After that images are shuffled along with the image label using numpy random permutation then it is divided into training data and testing data in the ratio of 8:2.

a) Training data: 16000 40\*40 pixels images and its labels

b) Testing data: 4000 40\*40 pixels Images and its labels

For memory management, training data is further split into batch size of 32, then passed into the network for training.

Results:

Considering image size and the network architecture, training time is expected to be slow and require lots of memory. For current configuration and training for 20 epochs took approximately 350-380 seconds which come out to 16-18 seconds per epoch.

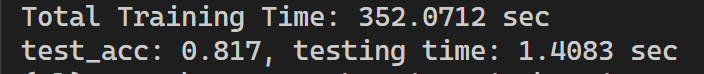


Figure 3 Training and testing time

After training for 20 epochs, the loss dropped down to around 0.40. Considering hardware limitation, training was only done for 20 epochs although the loss curve is indicating good learning rate. And for testing, it required 1.4 seconds and the resulted accuracy was 81%, considerably better than logistic regression (70%) implementation.

A screenshot of a cell phone

Description automatically generated

Figure 4 Training Cost per epoch

**Summary**

**Convolutional Neural Network**

|  |  |  |  |
| --- | --- | --- | --- |
| Layer | Size | Padding | Stride |
| Input | 40x40x1 |  |  |
| Kernel | 3x3x8 | 0 | 1 |
| Convolution | 38x38x8 |  |  |
| Pool | 2x2 | 0 | 2 |
| Max Pool | 19x19x8 |  |  |
| Dense | 2888x32 |  |  |
| Softmax | 2 |  |  |

Learning Rate: 1e-3

Epochs: 20

Batch Size: 32

Training Loss: 0.40

Training Accuracy:

Testing Accuracy: 81.7%

Classes: 2

Images per class: 10000

Training Images: 16000

Testing Images: 4000

**Keras Implementation**

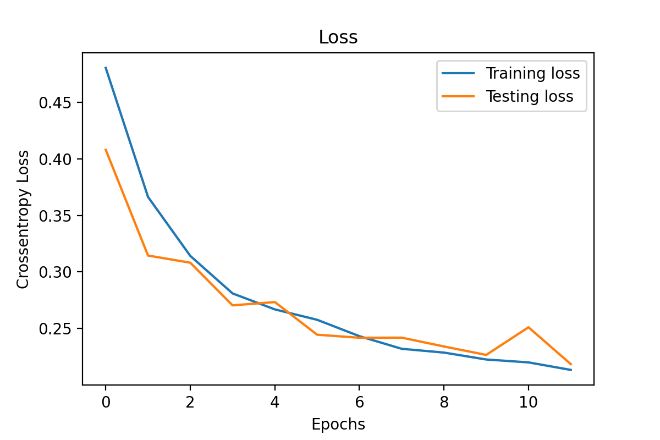
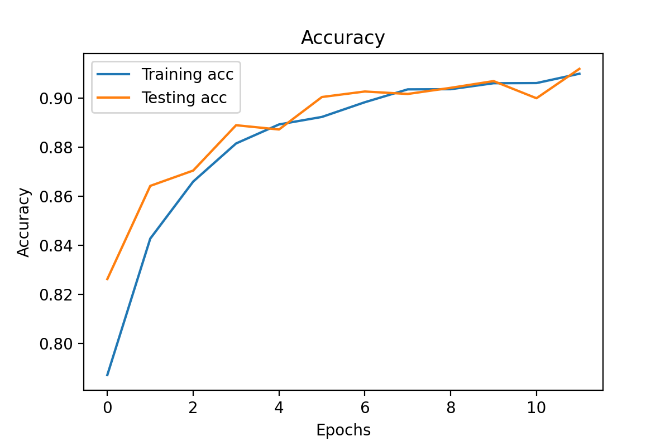
**Model Summary**

Input image of size 40x40 was split into training and testing set randomly at a ratio 8:2. A kernel size 3x3 and stride 1 with ‘valid’ padding returned a Conv layer of 38x38x8. A pool size of 2x2 with stride 2 returned a MaxPool layer of size 19x19x8. This layer was flattened and pushed to a Dense layer with 32 hidden units and ‘relu’ activation. Finally, a softmax layer with 2 classes was used for the output layer.

|  |  |  |
| --- | --- | --- |
| Layer (type) | Shape | Param # |
| (Conv2D) | (None, 38, 38, 8) | 80 |
| (MaxPooling2D) | (None, 19, 19, 8) | 0 |
| (Flatten) | (None, 2888) | 0 |
| (Dense) | (None, 32) | 92448 |
| (Dense) | (None, 2) | 66 |
|  |  |  |
| Total Params: 92,594 | | |
| Trainable Params: 92,594 | | |

**Model Accuracy and Loss**

With a batch size of 32 and 12 epochs, the keras implementation of the CNN received an accuracy of 91% and loss of 0.2134.



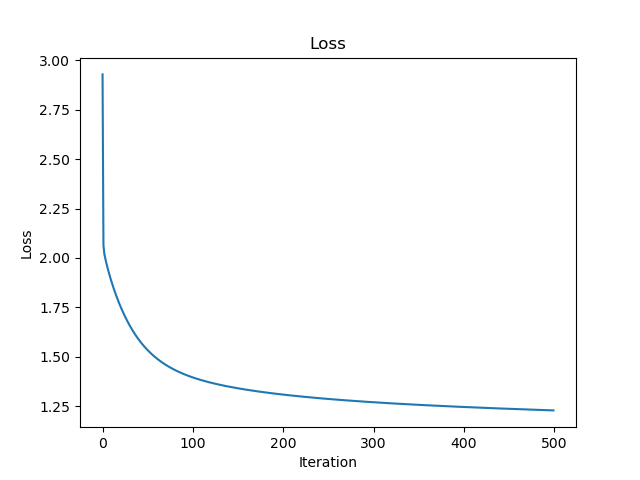


Figure 5 Logistic Regression Loss

A screenshot of a cell phone

Description automatically generated

Figure 6 Logistic Regression Training Accuracy

**Conclusion**

Based of off our implementation of Logistic Regression, a training accuracy of 70.935% was achieved which is miles worse than our implementation of CNN with an accuracy of 81%. Similar trend is also visible for the testing set. The training loss also dropped from 1.252 to 0.4. This could’ve been even better if run on an instance with better hardware configurations considering higher epochs and hidden units.

Since Numpy does not support GPU acceleration, Future implementation can replace numpy arrays with tensors from tensorflow or any other frameworks that supports parallel computation and memory efficiency.