ID5130 COURSE PROJECT REPORT
PARALLELISED NEURAL NETWORKS FOR IMAGE CLASSIFICATION USING OPENACC

Abhminayu Swaroop
MM17B008
Student
Metallurgical and Materials Engineering department
IIT Madras
mm17b008@smail.iitm.ac.in

Raghav Mallampalli
MM18B109
Student
Metallurgical and Materials Engineering department
IIT Madras
mm18b109@smail.iitm.ac.in

ABSTRACT

Image classification algorithms have a several use-cases in industry and academia, seeing application in microscopy, biotechnology, medicine, astrophysics, materials engineering etc. The industry standard is to use a deep learning (neural network) based model to classify images. An example is given in Figure 1.[1]

However, for a low visual information density problem like microstructure classification or number classification, convolution and a "shallow" neural network should suffice to obtain reasonable accuracy.

INTRODUCTION

We use convolution, pooling and a dense neural network in sequence for the task of classifying microstructures. Our chosen dataset is the NIST Ultra High Carbon Steels micrograph dataset. The microstructure type (pearlite, martensite, etc) is the label to be classified into. 5766 images were extracted from 961 micrographs and the model classifies them into one of the 7 microstructure type labels.
OpenACC GPU parallelisation will be carried out.
Time performance profiling and test accuracy will be used to judge the implementation. A successful implementation would yield identical or sufficiently similar results in a lesser amount of time for the entire gamut of training operations.

[1]Sumit Saha: A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way, medium post

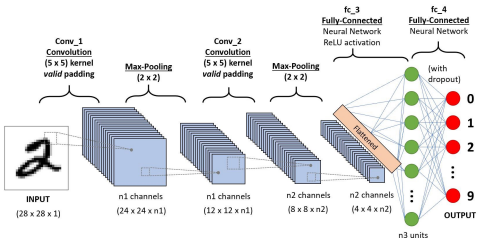

FIGURE 1. EXAMPLE DEEP LEARNING MODEL ARCHITECTURE

MODEL ARCHITECTURE

Convolution kernels

Convolution was performed with stride 3 and 0 padding.

Sobel: The Sobel kernels to be used in the edge detection convolution are:

$$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, \begin{pmatrix} -1 & 2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

They detect edges along vertical and horizontal direction respectively.

Sharpen: The standard sharpen kernel was used. It will increase the contrast of the detected edges in a lower dimension

FIGURE 2. COMPARISON OF ALGORITHMS: CLOCKWISE; ORIGINAL IMAGE, SOBEL ALGORITHM, FUZZY RELATIVE PIXEL ALGORITHM

representation of the image.

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & +5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

Manual: A manually tuned kernel was also used. The form of the manual kernel turned out to be similar to the contrast kernel for best performance. Note that only centre symmetric kernels were explored for the tuning. It is possible that greater performance can be obtained if more eccentric kernels are used.

$$\begin{pmatrix} -1 & -1 & -1 \\ -1 & +8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

Neural Network

The neural network consists of one hidden nodes and one output layer. The input for the neural network is obtained by pre-processing the images using the above mentioned kernels. The output layer consists of 7 nodes (each corresponding to a class label) and the number of nodes in the hidden layer can be varied according to the need of the user.

The neural network uses softmax activation function to compute the "belongingness" of a particular image to all classes. The vector output by the softmax function sums to 1 and can be mathematically described as,

$$y_i(z_i) = \frac{\exp z_i}{\sum e^{z_i}} \tag{1}$$

Cross-entropy loss function is used to calculate the loss for any given iteration as it is known to outperform other cost functions for a multi-class classification problem. Cross-entropy is calculated by adding the product of true probability and negative log of the predicted probabilities. Mathematically speaking,

$$H(y,\hat{y}) = -\sum y_i log(\hat{y}_i) \tag{2}$$

EQUATIONS

Convolution Take a kernel K (a 3x3 matrix with real numbers values). Consider input and output matrix ϕ_{in}, ϕ_{out}

$$\begin{aligned} \phi_{out,i,j} = K_{0,0} * \phi_{in,i-1,j-1} + \\ K_{0,1} * \phi_{in,i-1,j} + K_{0,2} * \phi_{in,i-1,j+1} + \\ K_{1,0} * \phi_{in,i,j-1} + K_{1,1} * \phi_{in,i,j} + \\ K_{1,2} * \phi_{in,i,j+1} + K_{2,0} * \phi_{in,i+1,j-1} + \\ K_{2,1} * \phi_{in,i+1,j} + K_{2,2} * \phi_{in,i+1,j+1} \end{aligned} \tag{3}$$

This operation is carried over the entire input matrix to generate the output matrix.

Neural Network: Feed Forward

Since the model described above uses two different activation functions for the hidden nodes and the output layer, we split the forward feed into two phases. First, to calculate the output of each node in the hidden layer, the input vector X is multiplied with the weights (wh) for each node along with the addition of a bias term (bh). The value obtained is passed through the sigmoid function to obtain the final value (ah) for a given node. Mathematically speaking,

$$zh = X.wh + bh \tag{4}$$

$$ah = \frac{1}{1+e^{-zh}} \tag{5}$$

The second step is to calculate the output of the output nodes which will be used to predict the class of the image. The value from each node in the hidden layer is taken as input and multiplied with the weights of the output layer (wo) along with the addition of a bias (bo). The activation function used here is a sigmoid function which makes sure that the sum of values at every output nodes for a given image sums to 1. This allows us to treat them as probabilities and use it to classify the image.

$$zo = ah.wo + bo \tag{6}$$

$$ao_i = \frac{zo_i}{\sum e^{zo_i}} \tag{7}$$

Neural Network: Back Propagation

A gradient descent algorithm is used to update the weights of the both the hidden layer and the output layer. We take the derivative of the cost function with respect to each of the weights. Calculation of the derivatives for the output layer can be done using the following equations,

$$\frac{dcost}{dwo} = \frac{dcost}{dao} * \frac{dao}{dzo} * \frac{dzo}{dwo} \tag{8}$$

The product of the first two terms can be found out using the following equation where y is the actual label for the image. The final term in the above equation is simply the output from the hidden node.

$$\frac{dcost}{dao} * \frac{dao}{dzo} = ao - y \tag{9}$$

$$\frac{dzo}{dwo} = ah \tag{10}$$

To update the bias function we use the following derivative,

$$\frac{dcost}{dbo} = \frac{dcost}{dao} * \frac{dao}{dzo} * \frac{dzo}{dbo} \tag{11}$$

The first two terms of the product are the same as shown above and the last term can be written as,

$$\frac{dzo}{dbo} = ao - y \tag{12}$$

Now we compute the derivative of the cost function for the hidden layer.

$$\frac{dcost}{dwh} = \frac{dcost}{dah} * \frac{dah}{dzh} * \frac{dzh}{dwh} \tag{13}$$

The first term of the derivative can be re-written as,

$$\frac{dcost}{dah} = \frac{dcost}{dzo} * \frac{dzo}{dah} \tag{14}$$

From the previous equations we know that,

$$\frac{dcost}{dzo} = \frac{dcost}{dao} * \frac{dao}{dzo} = ao - y \tag{15}$$

and,

$$\frac{dzo}{dah} = wo \tag{16}$$

The remaining values in the derivative of cost function can be found out using the following equations,

$$\frac{dah}{dzh} = sigmoid(zh) * (1 - sigmoid(zh)) \tag{17}$$

$$\frac{dzh}{dwh} = X \tag{18}$$

Finally we have to calculate the derivative of cost function to update the bias of the hidden layer. The derivative is given by,

$$\frac{dcost}{dbh} = \frac{dcost}{dah} * \frac{dah}{dzh} * \frac{dzh}{dbh} \tag{19}$$

This can be simplified to only the first two terms in the product, that have already been calculated above, since,

$$\frac{dzh}{dbh} = 1 \tag{20}$$

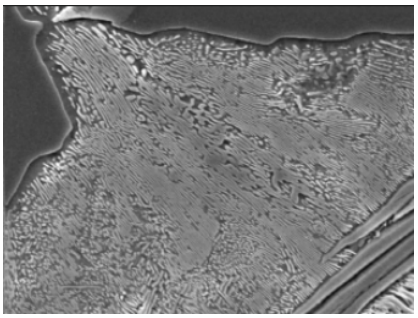

FIGURE 3. SAMPLE MICROGRAPH FROM DATASET

FIGURE 4. LABELS

IMPLEMENTATION NOTES

Extracting input images from the micrographs was done using Python. The images were saved into space separated files (1 for each image). One hot encoding was created for the labels and it was stored as a 5766*7 array (each column represents the class - 1 implies image belongs to that class, 0 implies it does not).

Pre-processing was done on an image-by-image basis. The file was opened and saved into an array. Once processed, the data was saved into another array.

To ensure fair bench-marking, only arrays were used for both the serial and parallel versions. No libraries were used apart from the built-in C ones. The time taken to load the data is included in our computations since it is identical for serial and parallel and makes no difference.

OBSERVATIONS

Maximum accuracy of approximately 39% was obtained for all tested neural network architectures and maximum epoch combinations. The accuracy obtained in the serial and parallel mode of the code was the same giving confidence that the parallel execution of the code is correct. The figures above show the variation of execution time for the code in different parameter settings. The figures also compare the parallel execution time with the serial execution time.

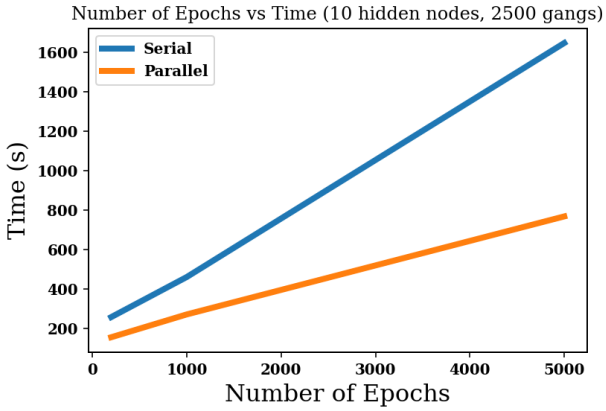

FIGURE 5. COMPARISON OF SERIAL AND PARALLEL RUN TIME WITH VARYING NUMBER OF EPOCHS

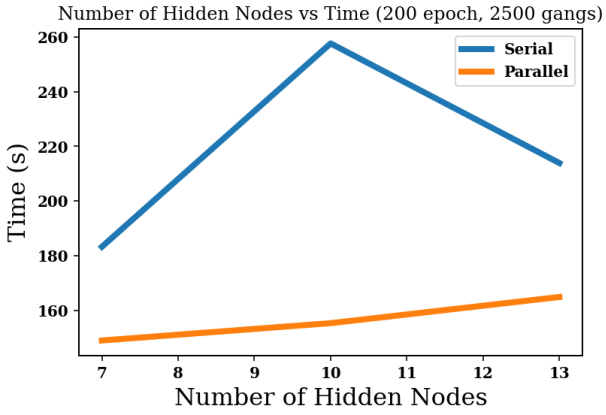

FIGURE 6. COMPARISON OF SERIAL AND PARALLEL RUN TIME WITH VARYING NUMBER OF NODES IN THE HIDDEN LAYER

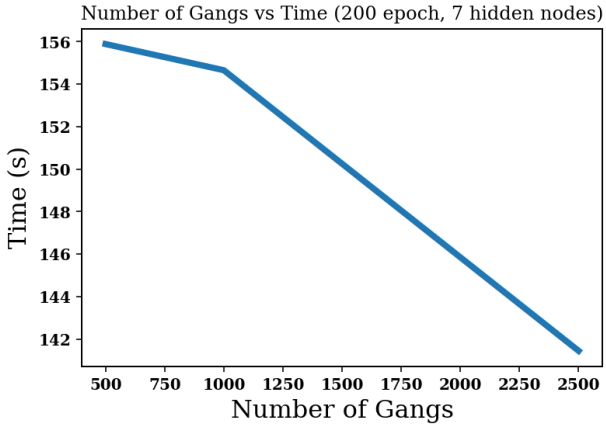

FIGURE 7. ILLUSTRATION DEPICTING THE PARALLEL RUN TIME WITH VARYING NUMBER OF GANGS

INFERENCES

The implemented architecture was found to be highly parallelisable, with the parallel version performing better than the serial for all combinations of tested parameters.

For high epoch and hidden layer nodes, the speed-up was ≈ 3.

The accuracy saturation at a low epoch number seems to point out that the pre-processing is not carrying over the entire information present in the image.

A possible next step would be do reduce the number of pooling layers or increase pooling stride to allow greater information pass-through.

Obtaining maximum ($> 95\%$) accuracy will likely require a full convolutional neural network.

Further scope for parallelisation: It may be possible to parallelise the pre-processing code over the input_size loop and the label loading loop, however it was avoided as it would result in the same file pointer attempting to read the same file at different locations.

CONCLUSION

The encoded architecture has performed satisfactorily. Scope for parallelisation was explored to the extent possible and methods not explored were listed. Parallelisation greatly improved performance.

ACKNOWLEDGMENT

We thank Matthew D. Hecht et al. for providing access to the NIST UHCS database. We also thank Professors Rupesh Nasre and Kameswararao Anupindi for the material provided for teaching parallelisation concepts. Thanks go to D. E. Knuth and L. Lamport for developing the wonderful word processing software packages TEX and LATEX. We also would like to thank Ken Sprott, Kirk van Katwyk, and Matt Campbell for fixing bugs in the ASME style file `asme2e.cls`. Lastly, we would like to thank HPCE IITM for providing access to the AQUA computing cluster.