



Indian Institute of Technology, Delhi
Mathematics Department

MTL851 : Project report

Convolutional Neural Networks for Efficient Preconditioner Generation

by

Amit Kumar

2019MT60744

Sajal Tyagi

2019MT60761

Sem-II, 2022-2023

Table of Contents

Table of Contents	iii
1 Preconditioners	1
1.1 Jacobi Preconditioning	2
1.2 Block Jacobi Predconditioning	2
2 Convolutional Neural Networks	5
2.1 Understanding Convolutions	5
2.2 Activation Functions	6
2.3 Fully Connected Layer	7
2.4 Training the Network	8
3 CNNs and Preconditioner Generation	9
4 Experimental Setup and Results	13
4.1 Dataset	13
4.2 Experimental Environment	14
4.3 Network Training	14
4.4 Improvements	15
References	17
5 GMRES Method	19
5.1 The GMRES Algorithm	19
5.2 Convergence of the Algorithm	20

Chapter 1

Preconditioners

There are many occasions and applications where iterative methods for solving a System of Linear equations fail to converge or converge very slowly. Methods of preconditioning these systems exist so that their subsequent solution by iterative methods is made more computationally feasible.

The general problem of finding a preconditioner for a linear system $Ax = b$ is to find a matrix K (the preconditioner or preconditioning matrix) with the properties that -

1. K is a good approximation to A in some sense.
2. The cost of the construction of K is not prohibitive.
3. The system $Kx = b$ is much easier to solve than the original system.

The idea is that the matrix $K^{-1}A$ may have better properties in the sense that well chosen iterative methods converge much faster. In this case, we solve the system $K^{-1}Ax = K^{-1}b$ instead of the given system $Ax = b$.

In this case, we solve the system $K^{-1}Ax = K^{-1}b$ instead of the given system $Ax = b$. Krylov subspace methods need the operator of the linear system only for computing matrix vector products. This means that we can avoid forming $K^{-1}A$ explicitly. Instead, we compute $u = K^{-1}Av$ by first computing $w = Av$ and then obtain u by solving $Ku = w$. Note that, when solving the preconditioned system using a Krylov subspace method, we will get quite different sub-spaces than for the original system. The aim is that approximations in this new sequence of sub-spaces will approach the solution more quickly than in the original sub-spaces.

1.1 Jacobi Preconditioning

The simplest preconditioner consists of just the diagonal of the matrix:

$$D_{i,j} = \begin{cases} a_{i,i} & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases} \quad (1.1)$$

This is known as the (point) Jacobi preconditioner.

So, original Matrix A can be represented as sum of upper, lower and diagonal matrices as- $A = U + D + L$

It is possible to use this preconditioner without using any extra storage beyond that of the matrix itself. However, division operations are usually quite costly, so in practice storage is allocated for the reciprocals of the matrix diagonal. This strategy applies to many preconditioners below.

We assume D to be a good enough approximation of A in this case and use it as the preconditioner, hence we solve -

$$D^{-1}Ax = c(= D^{-1}b) \quad (1.2)$$

D^{-1} can be computed in a linear time by just taking inverse of every non-zero element. It is possible to use this preconditioner without using any extra storage beyond that of the matrix itself. However, division operations are usually quite costly, so in practice storage is allocated for the reciprocals of the matrix diagonal. This strategy applies to many preconditioners below.

1.2 Block Jacobi Predconditioning

Block versions of the Jacobi preconditioner can be derived by a partitioning of the variables. If the index set $S = \{1, \dots, n\}$ is partitioned as $S = \bigcup_i S_i$ with the sets S_i mutually disjoint, then -

$$D_{i,j} = \begin{cases} a_{i,j} & \text{if } i \text{ and } j \text{ are in the same index subset} \\ 0 & \text{otherwise.} \end{cases} \quad (1.3)$$

The preconditioner is now a block-diagonal matrix after certain rearrangement are made so that rows corresponding to connected unknowns are adjacent.

Let the block-diagonal matrix be -

$$D = (D_1, D_2, \dots, D_N), \quad D_i \in \mathbb{R}^{m_i \times m_i}, \quad i = 1, 2, \dots, N \text{ such that } n = \sum_{i=1}^N m_i$$

So, original Matrix A can be represented as sum of block upper, block lower and block diagonal matrices as- $A = U + D + L$

We assume D to be a good enough approximation of A in this case and use it as the preconditioner, hence we solve -

$$D^{-1}Ax = c (= D^{-1}b) \quad (1.4)$$

We can invert D parallelly and find $D^{-1} = \{D_i^{-1} | i \in \{1, \dots, n\}\}$, this makes calculation much easier and faster as these block dimensions are much smaller and matrix inversion is $O(n^3)$

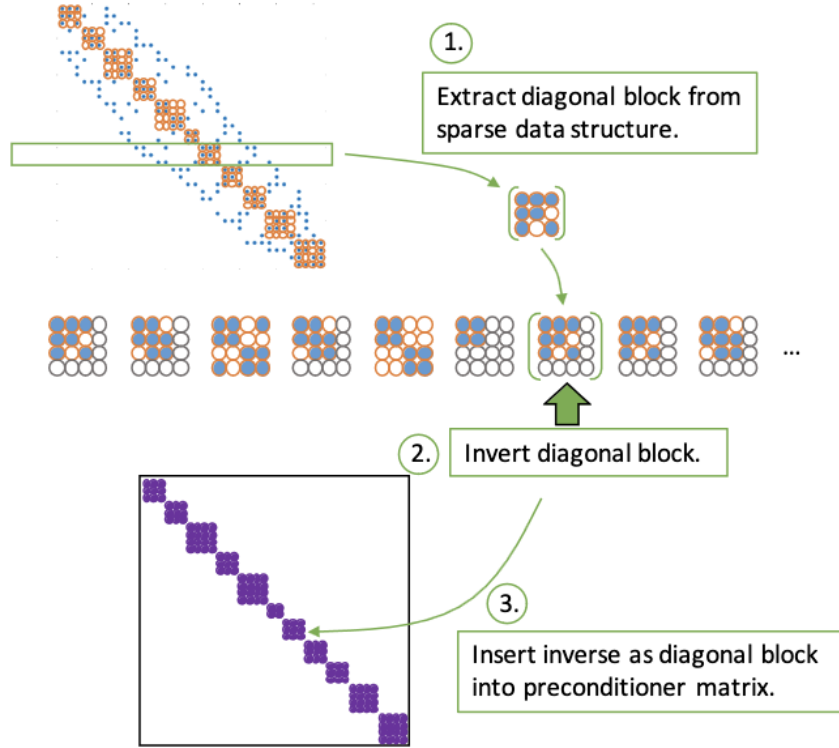


FIGURE 1.1: Block Jacobi Scheme Visualization

Often, natural choices for the partitioning suggest themselves:

- In problems with multiple physical variables per node, blocks can be formed by grouping the equations per node.

- In structured matrices, such as those from partial differential equations on regular grids, a partitioning can be based on the physical domain. Examples are a partitioning along lines in the 2D case, or planes in the 3D case.
- On parallel computers it is natural to let the partitioning coincide with the division of variables over the processors.

The main challenge while creating an effective preconditioner is finding reasonable S_i 's. The preconditioner works particularly well, if the diagonal blocks, to be inverted, represent strongly connected unknowns. Given an appropriate ordering of the components, these strongly connected unknowns appear as dense “natural blocks” in the, otherwise sparse, matrix. Clustering techniques from graph analytics, and reordering of the unknowns combined with priority blocking can be used generate efficient preconditioners, but the high computational cost and the inferior scalability properties of these algorithms make their use questionable. In consequence, the preprocessing step of generating an efficient block pattern for a block-Jacobi preconditioner remains a challenging and computationally expensive task. One of the most popular method for efficient block patterns generation was using supervariable blocking techniques which are highly scalable and of comparable time-complexity to the method we'll be suggesting thus, acting as a good baseline.

Chapter 2

Convolutional Neural Networks

Convolutional neural networks are a subclass of traditional artificial neural network, they compose of a number of weight- trainable neurons arranged in a layer-wise fashion. Typically, CNNs are tailored towards the pattern matching in two-dimensional matrices, with the goal of providing functionality analogous to the visual cortex enabling image perception in animals and humans. The main design characteristic of CNNs is the use of a convolutional layer (or multiple convolutional layers) consisting of very particular neurons.

2.1 Understanding Convolutions

We assume the input I to our CNN model belongs to $\mathbb{R}^{n \times m \times d_1}$ and let F be a filter $F = \{K_i | K_i \in \mathbb{R}^{k \times k \times d_1}, i \in \{1, \dots, d_2\}\}$, $k \leq \min(n, m)$ with stride S , this leads to output $O \in \mathbb{R}^{n^* \times m^* \times d_2}$ here $n^* = \frac{n-k}{S} + 1$ and $m^* = \frac{m-k}{S} + 1$ and if $n - k$ or $m - k$ are not divisible by S then the configuration is not valid. The output $O_{(a,b,c)}$ is given by the update function -

$$O_{(a,b,c)} = \sum_{d=1}^{d_1} \sum_{x,y=1}^k (K_c)_{(x,y,d)} \times I_{(1+(i-1)s+x, 1+(j-1)s+y, d)} \quad (2.1)$$

here, we assume Identity activation (see next heading) and $X_{(a,b,c)}$ represents the element with index (a, b, c) in a three dimensional matrix where the first two dimensions can be seen as the dimensions of a regular 2-D matrix and the third can be considered as depth, so we can visualize these matrices as cuboids or multiple stacked 2-D matrices. We call the process of getting O from I as passing I through a convolutional layer with filter F .

We use the two diagrams below to intuitively understand how convolutions work -

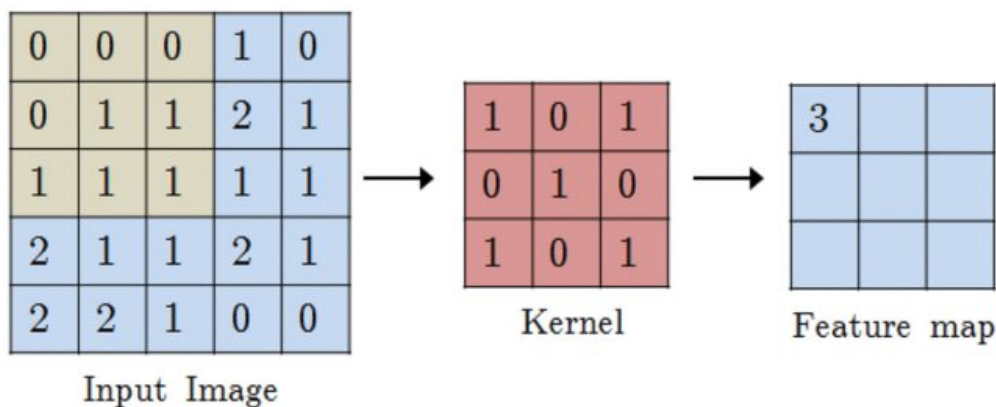


FIGURE 2.1: Action of a kernel on a 1-D Matrix input

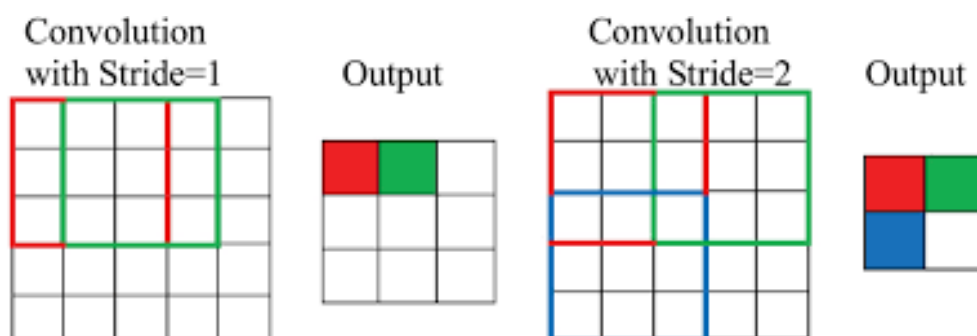


FIGURE 2.2: Effect of Stride on Convolutions

2.2 Activation Functions

The activation function of a node defines the output of that node given an input or set of inputs. Only nonlinear activation functions allow such networks to compute nontrivial problems using only a small number of nodes, and such activation functions are called nonlinearities. This leads to correct Output updation under activation function $F^*(x)$ being-

$$O_{(a,b,c)} = F^*\left(\sum_{d=1}^{d_1} \sum_{x,y=1}^k (K_c)_{(x,y,d)} \times I_{(1+(i-1)s+x, 1+(j-1)s+y,d)}\right) \quad (2.2)$$

There are a wide variety of Activation functions and different activation functions perform best for different tasks. We provide a table with all the necessary information about the activation functions we'll be using in our final model.



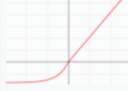
Name	Plot	Function, $g(x)$	Derivative of $g, g'(x)$	Range	Order of continuity
Logistic, sigmoid, or soft step		$\sigma(x) \doteq \frac{1}{1 + e^{-x}}$	$g(x)(1 - g(x))$	$(0, 1)$	C^∞
Rectified linear unit (ReLU) ^[8]		$(x)^+ \doteq \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$ $= \max(0, x) = x \mathbf{1}_{x>0}$	$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$[0, \infty)$	C^0
Scaled exponential linear unit (SELU) ^[11]		$\lambda \begin{cases} \alpha(e^x - 1) & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$ with parameters $\lambda = 1.0507$ and $\alpha = 1.67326$	$\lambda \begin{cases} \alpha e^x & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$(-\lambda\alpha, \infty)$	C^0

FIGURE 2.3: Table of relevant activation functions

2.3 Fully Connected Layer

Unlike Convolutional Layers where we have 3-D inputs and 3-D outputs; these layers are relatively straightforward, they transform input $I \in R^{n_1}$ to output $O \in R^{n_2}$ under weights $W \in R^{n_2 \times n_1}$ in the below given manner -

$$O_i = F^*\left(\sum_{x=1}^{n_1} (W)_{(i,x)} \times I_x\right) \quad (2.3)$$

Here, $F^*(x)$ is an activation function. These layers form the basis of the traditional Artificial Neural Network (ANN) architecture. We use the below image to better understand the process -

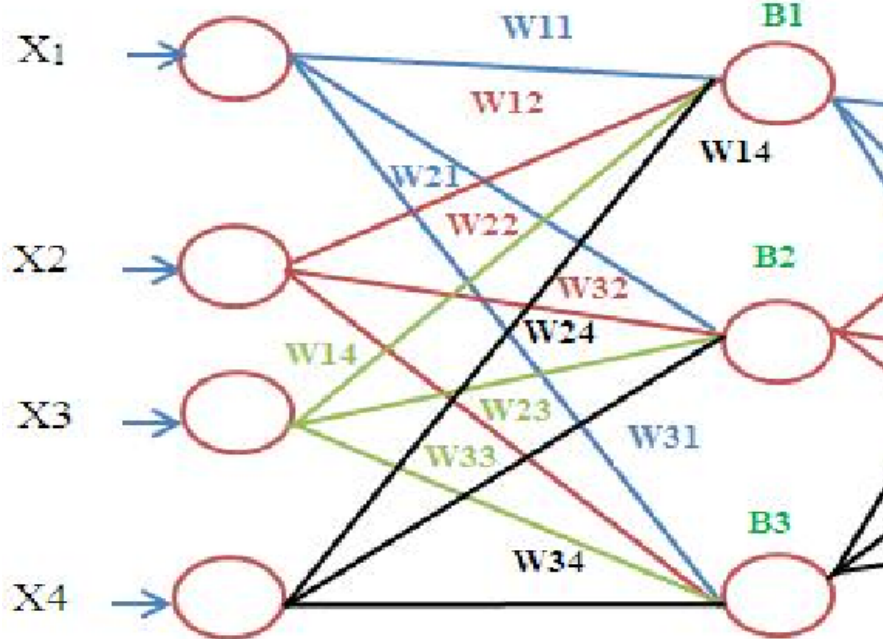


FIGURE 2.4: ANN Visualization

Generally, the application of a Convolution operation with activation is referred to as a "Layer" of the model. This is because the output O received is not the final output of

the model but multiple such "Layers" are applied one after the other (i.e. output of a layer is the input of the next layer) followed by Fully-Connected layers to transform the 3-D matrix formed after multiple Convolution Layers to the Final output $\hat{y} \in \{0, 1\}^n$.

2.4 Training the Network

The weights in fully connected layers and Kernel Weights in the Convolutional layers are initially randomly initialized but for the model to work correctly these need to learn these parameters. This learning is done by applying a loss function to the output \hat{y} of the whole network. But for the calculation of this loss for a particular input A , we'd need the correct output (w.r.t the task we're performing) y to be present too and then if \hat{y} is the output received from the network, we define a popular loss for classification tasks as -

$$\text{Loss} = -\frac{1}{\text{output size}} \sum_{i=1}^{\text{output size}} y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log (1 - \hat{y}_i) \quad (2.4)$$

This loss is called binary-cross entropy task and the choice for this would be justified in the next section.

Chapter 3

CNNs and Preconditioner Generation

For all further discussion, we'll be talking about generating a Block-Jacobi Preconditioner by using a CNN for binary classification tasks. The identification of block patterns in two-dimensional square matrices can be considered as a series of binary classification problems. For each row i (respectively column) of the system matrix A , the task is to decide whether or not it should be considered the start, i.e. the upper (left) bounding edge, of a diagonal block. These diagonal blocks aim at reflecting clusters of strongly connected components, which implies that these blocks typically appear "denser" than the rest of the matrix. Hence, one approach is to view the matrix sparsity pattern as an image, and to detect the diagonal blocks with a combination of traditional image processing algorithms which involve Feature Extraction and Preprocessing Steps followed by multiple denoising filters, which will give of the dense output blocks only.

Utilizing a convolutional neural network is particularly suitable for this task, as it is able to learn the necessary preprocessing and denoising features on its own. This training process is realized in a supervised fashion, i.e. the weights of the convolutional filters are inferred from examples where the block starts have previously been annotated ("labeled data"). For a matrix A of size n a label vector y with $y \in \{0, 1\}^n$ is used such that each row is assigned a separate and independent output neuron where 1 indicates the start of a block in the respective row. In the prediction process, the neural network then generates a prediction vector \hat{y} where each $\hat{y}_i \in [0, 1]$ indicates the probability of a block start in row i . The loss function for this multi-label classification problem is given as the sum of the binary cross-entropy across all the individual row predictions

and available samples S -

$$\mathcal{L}(y, \hat{y}) = - \sum_{s=1}^S \sum_{i=1}^n y_{s,i} * \log(\hat{y}_{s,i}) + (1 - y_{s,i}) * \log(1 - \hat{y}_{s,i}) \quad (3.1)$$

As we're dealing with sparse input matrices we suggest an optimization step of decreasing model complexity (number of parameters). The idea is to crop the matrix parallel to the main diagonal, and to base the prediction process on the diagonal matrix band only. This is motivated by the assumption that any block of strongly connected components is “denser” than the rest of the matrix also in the area close to the main diagonal 1. For a diagonal band width w , the reduction of the sparse matrix image to a “diagonal image” is realized by cropping parallel to the main diagonal at the pre-defined distance w , and arranging all elements of the diagonal band row-wise in a new matrix that is right-aligned. The missing values on the left are filled with an arbitrary constant c , representing the diagonal image extracted from the matrix image. Considering only the diagonal band of a matrix efficiently reduces the amount of pixels in the input images. In our experiments, we consider matrix images of size 128×128 and set $w = 10$. This results in diagonal images of size $(2 \times w + 1) \times n = 21 \times 128$.

The neural network we design is a feed-forward convolutional neural network composed of three logical parts. The first major part of the network, is a modified residual network block that aims at denoising the sparse matrix image. It consists of two two-dimensional convolutional layers with post-batch normalization and scaled exponential linear units as activations. The obtained filtered images are added to the original matrix image to generate a denoised copy.

In the second major block, consisting of non-standard, discrete convolutions, the image is reduced to a vector of length $2w + 1$. To that end, the convolution with mask height $2w + 1$ and width $2k + 1$ is applied to each matrix column, see Figure 4. To accommodate for the k left- and right- most columns, the diagonal image is horizontally padded with constant values (here: zeros), such that the convolution is applied to a matrix with dimensions $(2w + 1) \times (n + 2k)$. Each element within the resulting vector is activated using the tanh function to model a binary choice between the two options “block start” or “no block start”. Using a one- dimensional convolution, the choice is cross-correlated with the neighboring elements.

Finally, in the third and last part of the network, the actual prediction is derived using a fully-connected dense layer and output in the l -sized output. To prevent overfitting and increase out-of-sample accuracy, the convolutional layers are regularized with an l_2 -norm of 0.02, and the fully-connected layer with dropouts. Applying the argmax function to the prediction vector \hat{y} we obtain the rows/column indices i_0, i_1, \dots , where the diagonal blocks start

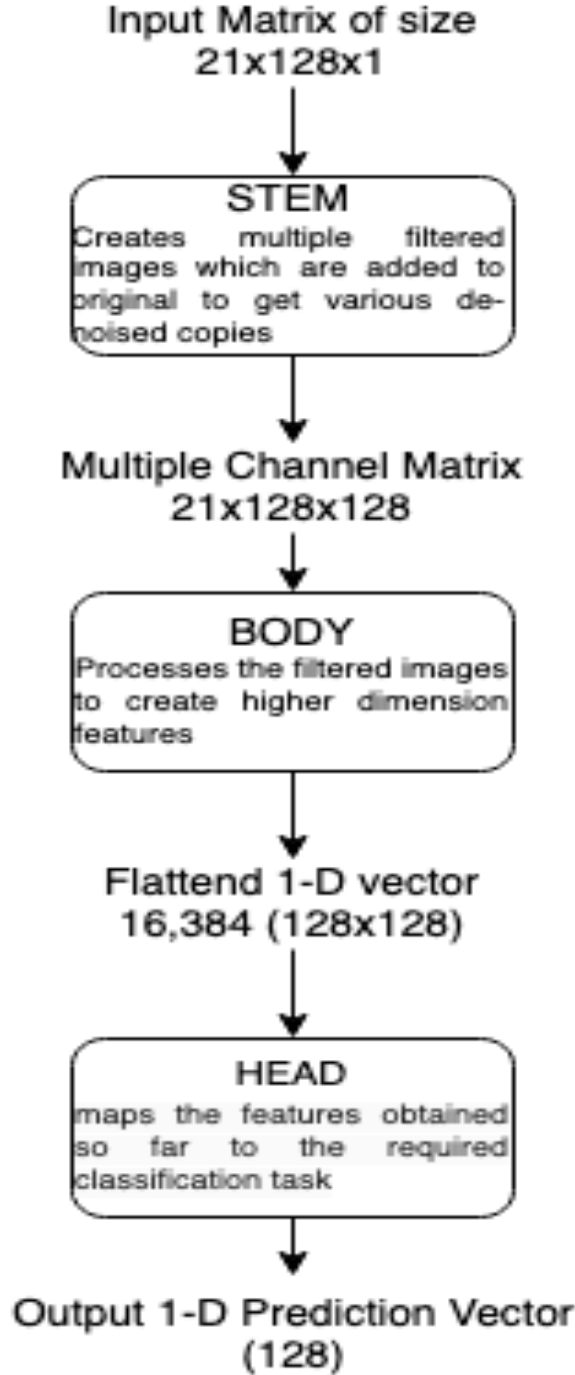


FIGURE 3.1: low level CNN Architecture Visualization

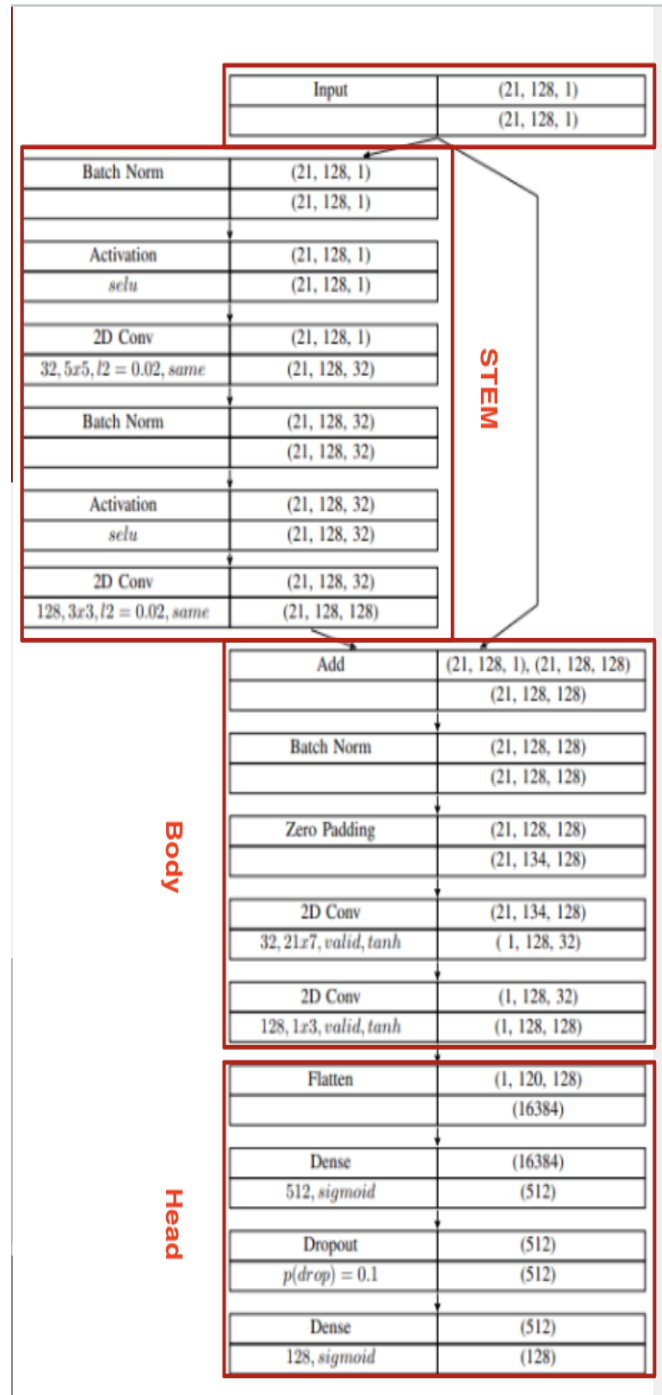


FIGURE 3.2: Actual CNN Architecture

Chapter 4

Experimental Setup and Results

4.1 Dataset

As discussed before, we need labelled data for our supervised learning approach i.e. matrix data with annotated Jacobi block patterns. Authors provide a set of 3000 such labelled examples which they artificially generated for training and validating the neural network. In total, we consider 3,000 matrices of size 128×128 . All matrices contain a set of randomly-sized diagonal blocks that are adjacent and not overlapping in the matrix sparsity pattern. The average size of the diagonal blocks is 10. The test matrices are all generated in the following fashion -

1. First, the empty matrix is filled with “background noise”, which is randomly scattered non-zero values. Across the test matrices we set the density of the background noise to values between 0 and 0.5 (following a uniform distribution).
2. Next, “noise blocks” of arbitrary size are added to the main diagonal. These noise blocks contain between 30 and 50
3. Finally, the diagonal blocks of arbitrary size with density values between 0.5 and 0.7 are generated on the main diagonal. This is realized by generating a vector of size 128, and then randomly selecting 10 as vector block starts.

The dataset was divided in the ratio of 0.8 for Training and 0.2 for Validation. The training set is the one which the model uses to learn weights and the validation set is the one which we use to find an optimal ending point for training i.e. when validation

loss stops decreasing we stop the training process.

4.2 Experimental Environment

We used google colab for training and testing our model and the specifications of the instance we used are provided below -

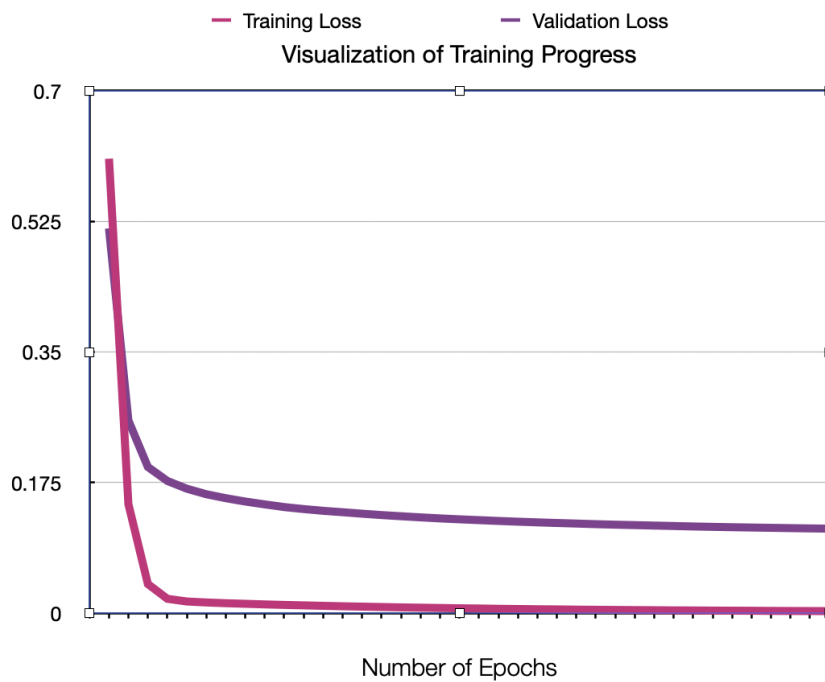
Intel(R) Xeon(R) CPU @ 2.30GHz

NVIDIA Tesla T4 with 15 GB VRAM and CUDA Version: 12.0

12.7 GB RAM clocked at 2237 MHz

4.3 Network Training

We provide the training progress of our model below -



To ensure the networks learning the right patterns and not simply reacting to the irregularities of the randomly generated noise, we look at the learned convolution filters after the stem part of the CNN.

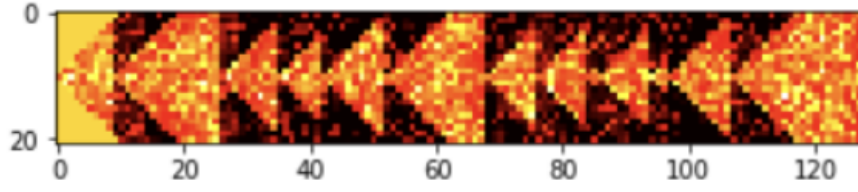


FIGURE 4.1: Input to the CNN Model

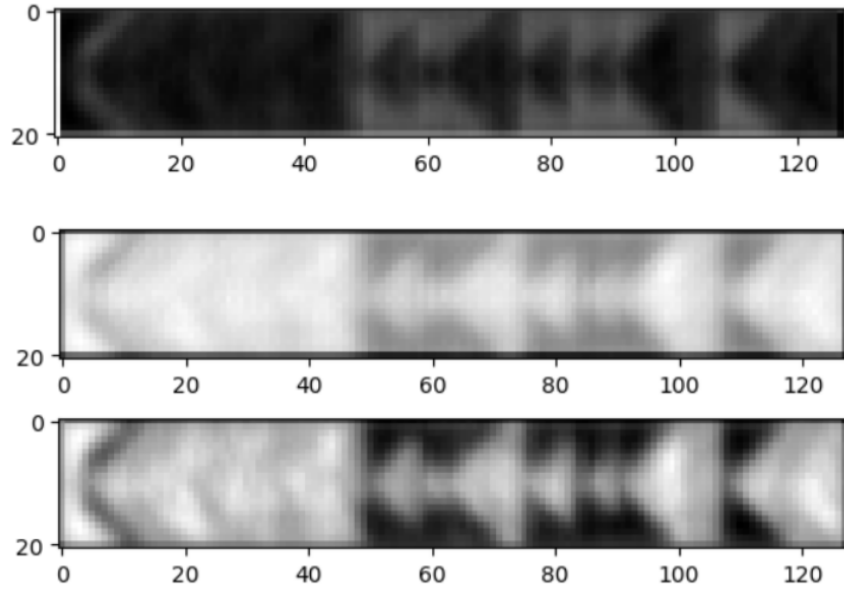
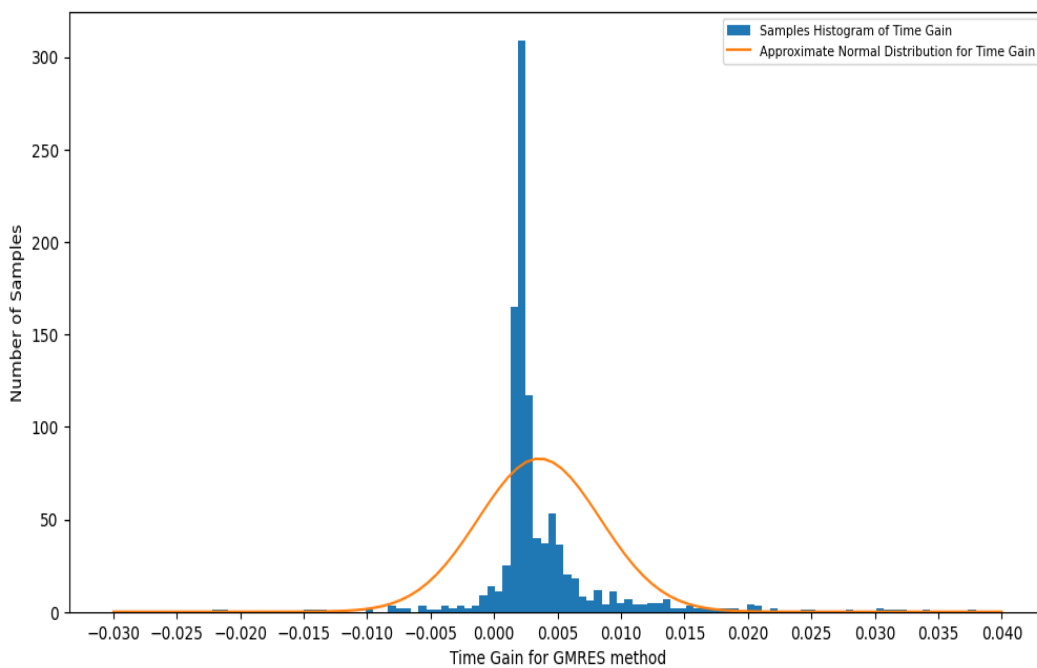
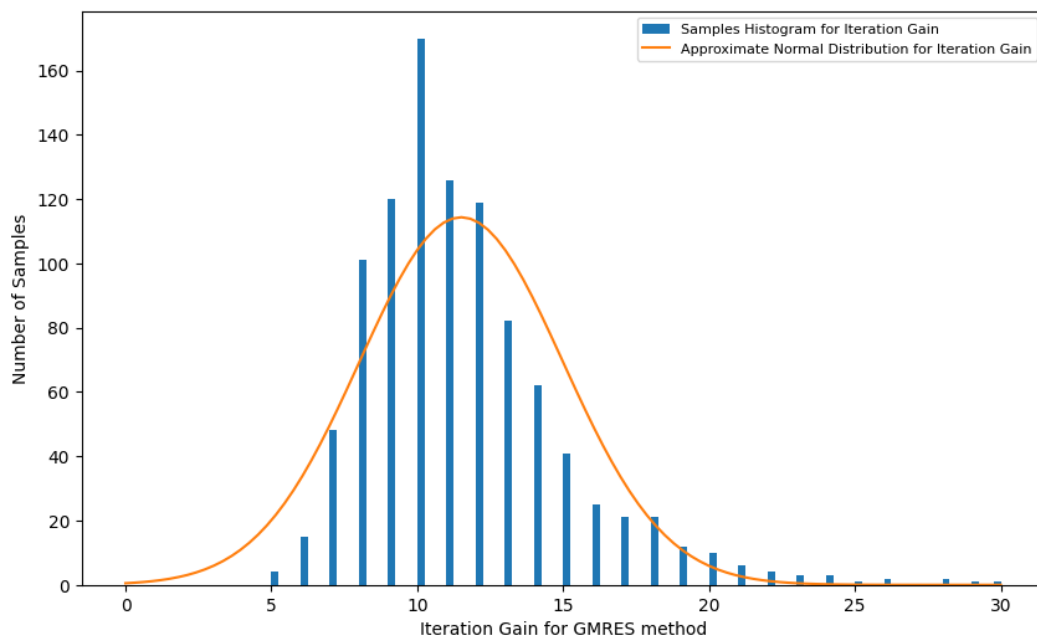


FIGURE 4.2: Convolution Filters learned by the CNN model

4.4 Improvements

We provide 2 histograms over 1000 randomly generated test matrices, one of which plots the difference of number of iterations for GMRES without preconditioning and GMRES with preconditioning.

The other histogram plots the difference of time required for GMRES without preconditioning and GMRES with preconditioning, where we used the time required for calculating the preconditioner, the inverse calculation process and the matrix multiplication for setting up the new GMRES problem too for the second process.



	Average Original Value	Average Gain Value	Average Percentage Gain
Iterations	128	11.507	8.990%
Time	0.0161	0.0036	22.141%

FIGURE 4.3: Summarized Gain Results

References

- [1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998. ISSN 1558-2256. doi: 10.1109/5.726791.
- [2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25, 01 2012. doi: 10.1145/3065386.
- [3] Youcef Saad and Martin H. Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986. doi: 10.1137/0907058. URL <https://doi.org/10.1137/0907058>.
- [4] 12. *Parallel Preconditioners*, pages 377–405. doi: 10.1137/1.9780898718003.ch12. URL <https://epubs.siam.org/doi/abs/10.1137/1.9780898718003.ch12>.
- [5] Markus Götz and Hartwig Anzt. Machine learning-aided numerical linear algebra: Convolutional neural networks for the efficient preconditioner generation. 11 2018. doi: 10.1109/ScalA.2018.00010.

Chapter 5

GMRES Method

5.1 The GMRES Algorithm

GMRES is an iterative method that uses Krylov subspaces to reduce a high-dimensional problem to a sequence of smaller dimensional problems. Let A be an invertible $m \times m$ matrix and let b be a vector of length m . Let $\mathcal{K}_n(A, b)$ be the order- n Krylov subspace generated by A and b . Instead of solving the system $Ax = b$ directly, GMRES uses least squares to find $x_n \in \mathcal{K}_n$ that minimizes the residual $r_n = \|b - Ax_n\|_2$. The algorithm terminates when this residual is smaller than some predetermined value. In many situations, this happens when n is much smaller than m .

The GMRES algorithm uses the Arnoldi iteration for numerical stability. The Arnoldi iteration produces H_n , an $(n+1) \times n$ upper Hessenberg matrix, and Q_n , a matrix whose columns make up an orthonormal basis of $\mathcal{K}_n(A, b)$, such that $AQ_n = Q_{n+1}H_n$. The GMRES algorithm finds the vector x_n which minimizes the norm $\|b - Ax_n\|_2$, where $x_n = Q_n y_n + x_0$ for some y_n . Since the columns of Q_n are orthonormal, the residual can be equivalently computed as -

$$\|b - Ax_n\|_2 = \|Q_{n+1}(\beta e_1 - H_n y_n)\|_2 = \|H_n y_n - \beta e_1\|_2. \quad (5.1)$$

Here e_1 is the vector $[1, 0, \dots, 0]^T$ of length $n+1$ and $\beta = \|b - Ax_0\|_2$, where x_0 is an initial guess of the solution. Thus, to minimize $\|b - Ax_n\|_2$, the right side of (5.1) can be minimized, and x_n can be computed as $x_n = Q_n y_n + x_0$.

```

1: procedure GMRES( $A, \mathbf{b}, \mathbf{x}_0, k, \text{tol}$ )
2:    $Q \leftarrow \text{empty}(\text{size}(\mathbf{b}), k + 1)$  ▷ Initialization.
3:    $H \leftarrow \text{zeros}(k + 1, k)$ 
4:    $\mathbf{r}_0 \leftarrow \mathbf{b} - A(\mathbf{x}_0)$ 
5:    $Q_{:,0} = \mathbf{r}_0 / \|\mathbf{r}_0\|_2$ 
6:   for  $j = 0 \dots k - 1$  do ▷ Perform the Arnoldi iteration.
7:      $Q_{:,j+1} \leftarrow A(Q_{:,j})$ 
8:     for  $i = 0 \dots j$  do
9:        $H_{i,j} \leftarrow Q_{:,i}^\top Q_{:,j+1}$ 
10:       $Q_{:,j+1} \leftarrow Q_{:,j+1} - H_{i,j} Q_{:,i}$ 
11:       $H_{j+1,j} \leftarrow \|Q_{:,j+1}\|_2$ 
12:      if  $|H_{j+1,j}| > \text{tol}$  then ▷ Avoid dividing by zero.
13:         $Q_{:,j+1} \leftarrow Q_{:,j+1} / H_{j+1,j}$ 
14:         $\mathbf{y} \leftarrow \text{least squares solution to } \|H_{:,j+2,j+1} \mathbf{x} - \beta \mathbf{e}_1\|_2$  ▷  $\beta$  and  $\mathbf{e}_1$  as in (17.1).
15:         $\text{res} \leftarrow \|H_{:,j+2,j+1} \mathbf{y} - \beta \mathbf{e}_1\|_2$ 
16:        if  $\text{res} < \text{tol}$  then
17:          return  $Q_{:,j+1} \mathbf{y} + \mathbf{x}_0, \text{res}$ 
18:   return  $Q_{:,j+1} \mathbf{y} + \mathbf{x}_0, \text{res}$ 

```

FIGURE 5.1: GMRES Algorithm Pseudo Code

5.2 Convergence of the Algorithm

One of the most important characteristics of GMRES is that it will always arrive at an exact solution (if one exists). At the n -th iteration, GMRES computes the best approximate solution to $Ax = b$ for $x_n \in \mathcal{K}_n$. If A is full rank, then $K_m = F_m$, so the m th iteration will always return an exact answer. Sometimes, the exact solution $x \in \mathcal{K}_n$ for some $n < m$, in this case x_n is an exact solution. In either case, the algorithm is convergent after n steps if the n -th residual is sufficiently small.