Introduction to Parallel Scientific Computing

Project Report

High Performance Convolutional Neural Networks

Course Instructor
Prof Pawan Kumar

By
Kunal Garg
20161067
Aniket Joshi
20161166
Animesh Sahu
20161028

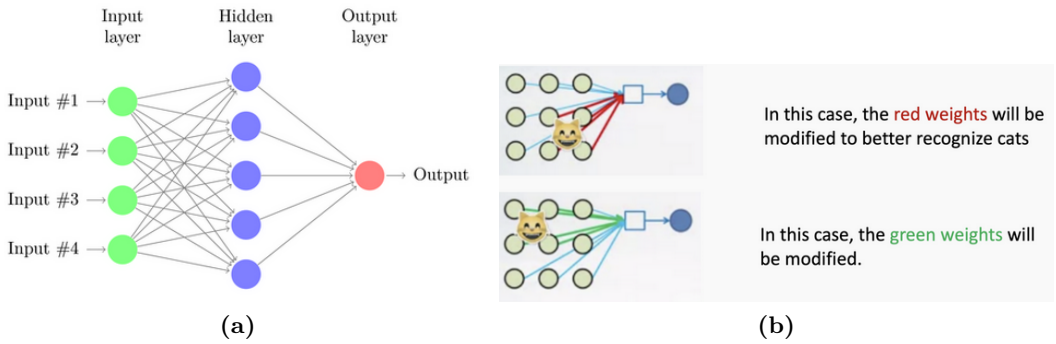
Abstract

In this project we implemented CNN model from scratch and performed different experiments to increase the performance of CNNs by coding up convolutional layer in different ways . Code can be found here

1 Motivation behind CNNs

There are several drawbacks of MLP's, especially when it comes to image processing.

1. MLPs do not scale well for images as they use one perceptron for each input. For a 224 x 224 pixel image with 3 color channels there are around 150,000 weights that must be trained!
2. MLPs are not translation invariant. For example, if a picture of a cat appears in the top left of the image in one picture and the bottom right of another picture, the MLP will try to correct itself and assume that a cat will always appear in this section of the image.
3. MLPs lose the spatial information because the image has to be flattened before input into an MLP. So, information brought by pixel position and correlation with neighbors is ignored. Thus, we need a way to leverage the spatial correlation of the image features (pixels) in such a way that we can see the cat in our picture no matter where it may appear.



(a) **(b)**

Figure 1: Figure 1a represents standard multilayer perceptron (traditional neural network) and Figure 1b represents a cat detector using an MLP which changes as the position of the cat changes.

2 CNNs

A Convolutional neural network (CNN) is a neural network that has one or more convolutional layers. There are three types of layers in a CNN:

1. Convolutional Layer
2. Pooling Layer
3. Fully Connected Layer

Each of these layers has different parameters that can be optimized and performs a different task on the input data.Each CNN layer learns filters of increasing complexity.

- First layers learn basic feature detection filters: edges, corners, etc.
- Middle layers learn filters that detect parts of objects. For faces, they might learn to respond to eyes, noses, etc.
- Last layers have higher representations: they learn to recognize full objects, in different shapes and positions. by using filters.

2.1 Padding

You should pad the input layer to CNN as per your use case. There are two types of padding:
Full Padding : It introduces zeros in equally in all directions, so all the pixels are visited the same amount of time by the filter. Also, output size is larger than input size.
Same Padding : It only pads such that the output will have same size as the input.

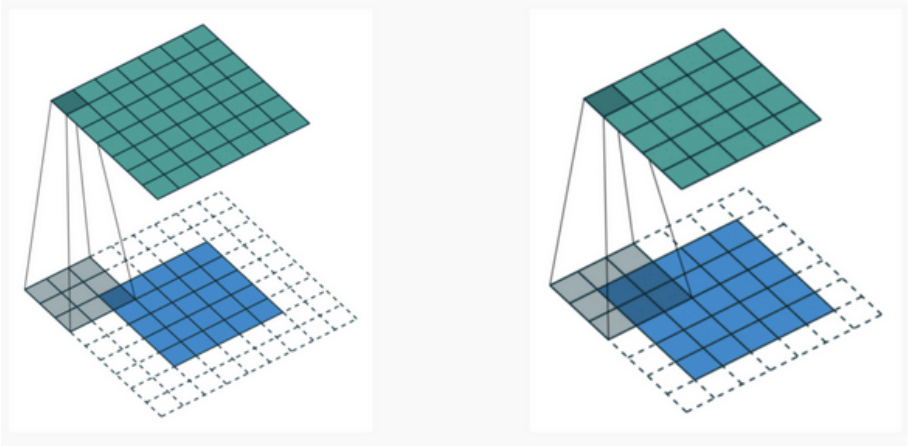


Figure 2: Illustration of how full padding and same padding are applied to CNN's.

2.2 Convolution layer

CNN's make use of filters (also known as kernels), to detect what features, such as edges, are present throughout an image. A filter is just a matrix of values, called weights, that are trained to detect specific features. The filter moves over each part of the image to check if the feature it is meant to detect is present. To provide a value representing how confident it is that a specific feature is present, the filter carries out a convolution operation, which is an element-wise product and sum between two matrices.

When the feature is present in part of an image, the convolution operation between the filter and that part of the image results in a real number with a high value. If the feature is not present, the resulting value is low.

2.3 Activation function : ReLU

The output of the convolution operation between the filter and the input image is summed with a bias term and passed through a non-linear activation function. The

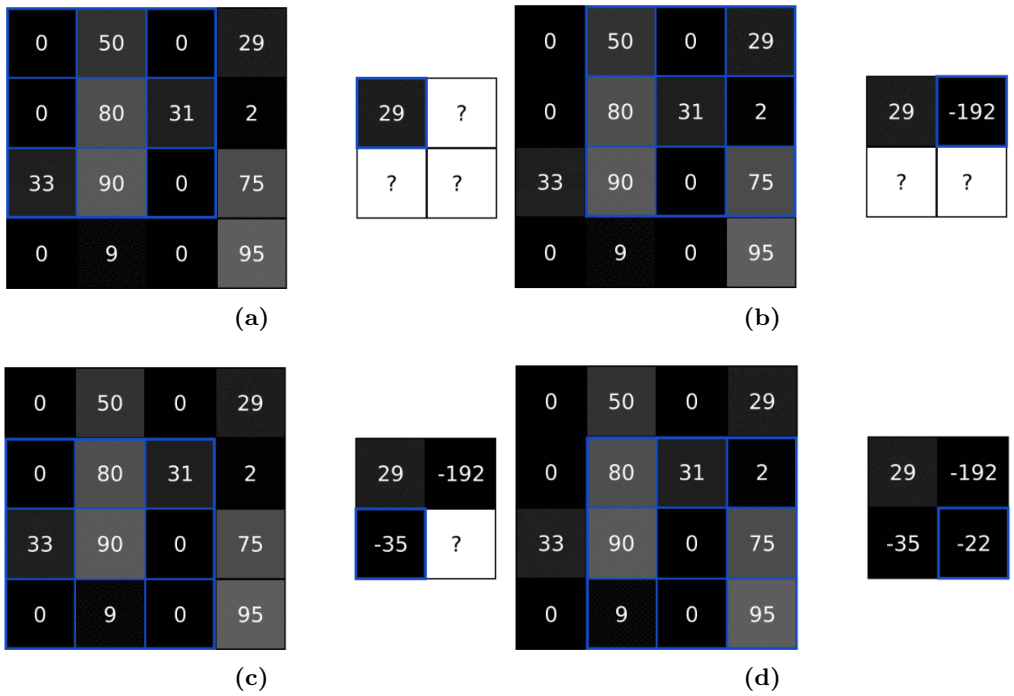


Figure 3: Convolution of 4x4 image using 3x3 filter with stride 1 to produce a 2x2 output

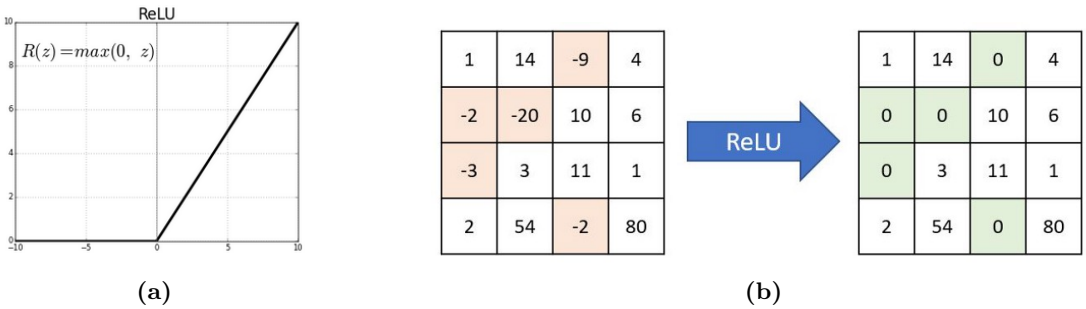


Figure 4: ReLU activation function

purpose of the activation function is to introduce non-linearity into our network.
Since our input data is non-linear (it is infeasible to model the pixels that form a handwritten signature linearly), our model needs to account for that. To do so, we use the Rectified Linear Unit (ReLU) activation function. After passing through ReLU function, values that are less than or equal to zero become zero and all positive values remain the same.

2.4 Pooling layers

These are typically used to reduce the dimensionality of the network and perform a specific function such as max pooling, which takes the maximum value in a certain filter region. This works because the neighboring pixels in images tend to have similar values, so conv layers will typically also produce similar values for neighboring pixels in outputs. There are two parameters that need to adjusted for this layer : Filter size and Stride.

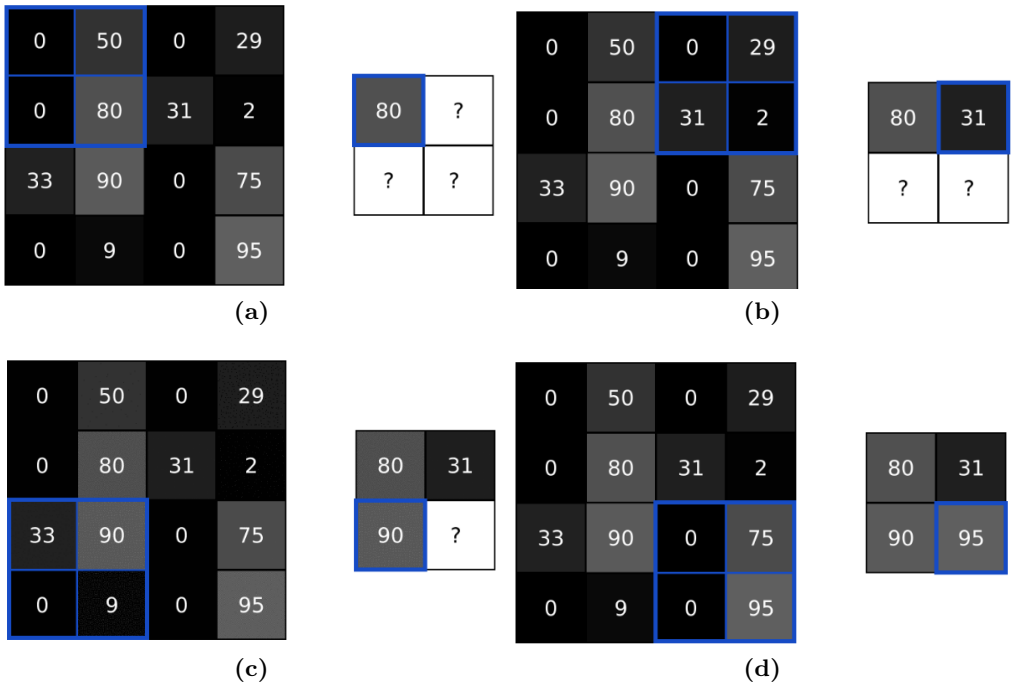


Figure 5: Max Pooling (pool size 2) on a 4x4 image to produce a 2x2 output

2.5 Fully connected layers

Fully connected layers are placed before the classification output of a CNN and are used to flatten the results before classification. This is similar to the output layer of an MLP. Two main parameters : Number of nodes, Activation function

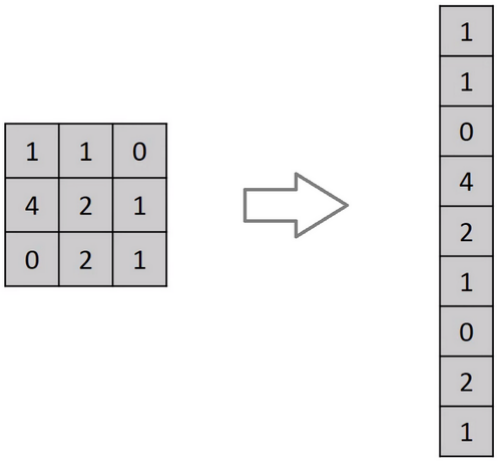
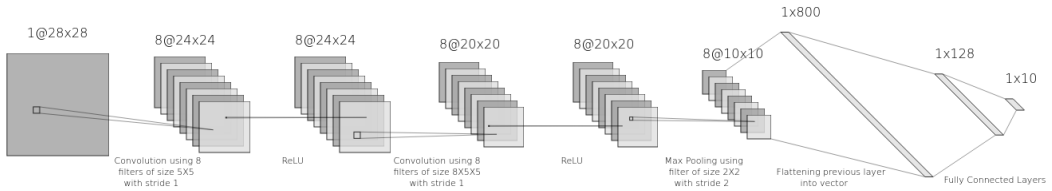


Figure 6: Fully connected layer

3 Network Architecture

Network uses two consecutive convolutional layers followed by a max pooling operation to extract features from the input image. After the max pooling operation, the repre-

sentation is flattened and passed through a Multi-Layer Perceptron (MLP) to carry out the task of classification.



Network Architecture

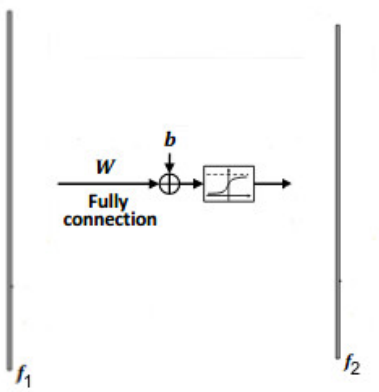
Total learnable parameters in network $= 8 * (5 \times 5 + 1) + 8 * (5 \times 5 \times 8 + 1) + 128 \times 800 + 10 \times 128 = 1,05,496$

4 Backpropagation in CNN

4.1 Backpropagation of Loss from Output layer

$$L = -\Sigma_i y_i \log(p_i)$$
$$\frac{\partial L}{\partial o_i} = p_i - y_i$$

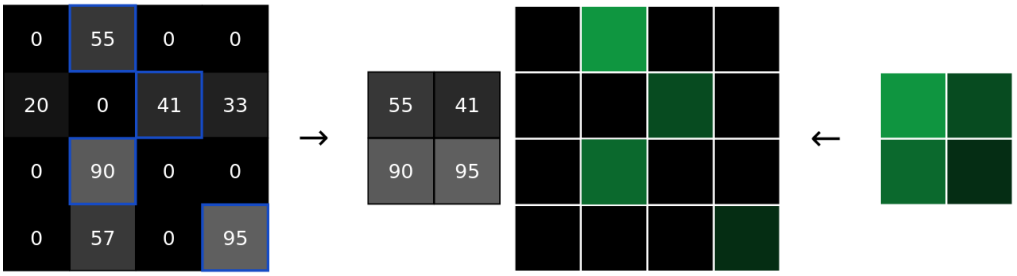
4.2 Backpropagation of Loss via FC layers



$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial f_2} \times f_1^T$$
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial f_2}$$
$$\frac{\partial L}{\partial f_1} = W^T \times \frac{\partial L}{\partial f_2}$$

4.3 Backpropagation of Loss via Max-pooling layer

Gradient value is only assigned to the positions where the original max value was, and every other value is zero.



(a) (b)

Figure 7: Figure 7a represents forward phase that transforms a 4x4 input to a 2x2 output and Figure 7b represents backward phase for the same layer that transforms a 2x2 gradient to a 4x4 gradient.

4.4 Backpropagation of Loss via ReLU

Gradient value is only assigned to the positions which had positive value in the forward pass.

(a) (b)

Figure 8: Figure 8a represents forward phase and Figure 8b represents backward phase for the same layer.

4.5 Backpropagation of Loss via Convolutional layer

- Flip Filter by 180 degree.
- Perform full convolution between flipped filter and loss gradient from the next layer $\frac{\partial L}{\partial O}$. Basically we need to pad fsize - 1 zeros on every side of loss gradient and then perform convolution.

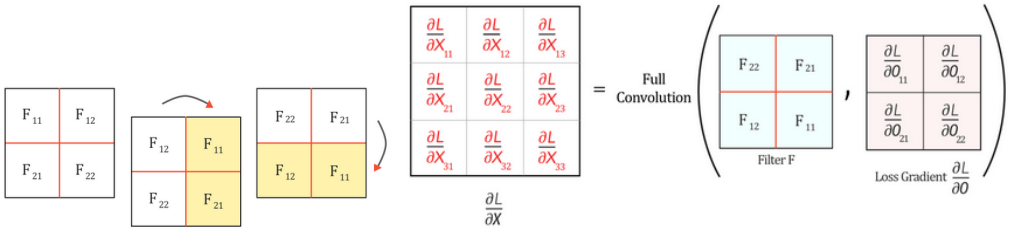


Figure 9: Figure 9a represents Flipping Filter F by 180 degrees — flipping it vertically and horizontally 9b represents Full Convolution operation visualized between 180-degree flipped Filter F and loss gradient $\frac{\partial L}{\partial O}$.

4.6 Loss gradient of Filter Weights in Convolutional layer

Computed by performing convolution between Input X and loss gradient from the next layer $\frac{\partial L}{\partial O}$.

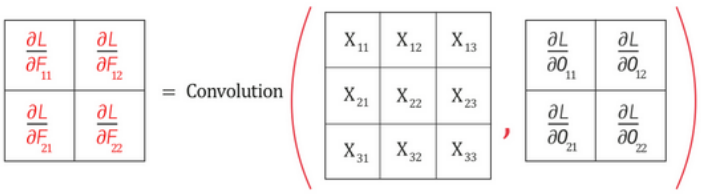


Figure 10: $\frac{\partial L}{\partial F}$ = Convolution of input matrix X and loss gradient $\frac{\partial L}{\partial O}$

4.7 Loss gradient of Filter Bias via Convolutional layer

Sum all the values of loss gradient from the next layer

5 Adam's optimizer

- Adam is an adaptive learning rate method, it computes individual learning rates for different parameters and updates network weights iterative based in training data as opposed to classical gradient descent which maintains a single learning rate for all weight updates, which does not change during training.
- Adam can be looked at as a combination of RMSprop and Stochastic Gradient Descent with momentum. It uses the squared gradients like RMSprop and it takes advantage of momentum by using moving average of the gradient instead of gradient itself like SGD with momentum.
- We used beta1 = 0.95, beta2 = 0.99 in code.

$$For\ each\ Parameter\ w^j$$
$$_{(j\ subscript\ dropped\ for\ clarity)}$$
$$\nu_t = \beta_1 * \nu_{t-1} - (1 - \beta_1) * g_t$$
$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2$$
$$\Delta\omega_t = -\eta \frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t$$
$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

$\eta : Initial\ Learning\ rate$
$g_t : Gradient\ at\ time\ t\ along\ \omega^j$
$\nu_t : Exponential\ Average\ of\ gradients\ along\ \omega_j$
$s_t : Exponential\ Average\ of\ squares\ of\ gradients\ along\ \omega_j$
$\beta_1, \beta_2 : Hyperparameters$

Figure 11: Update rules of Adam's optimizer

6 Experiments

6.1 Compared running speeds of 7 different implementations of performing convolution:

- Naive Convolution (using for loops)
- Convolution using matrix operations
- Convolution using matrix operations + parallelization within each batch
- Unrolled Convolution (using matrix multiplication)
- Unrolled Convolution + parallelization within each batch
- Convolution in Cuda
- Convolution in Keras

6.2 Parameters varied:

- Batch Size
- Learning Rate
- Training examples

7 Results

7.1 Comparison of Running Speeds

Implementation Method / Batch Size	64	32
Naive Conv	262	154
Normal Conv	12.71	7.56
Normal Conv + Parallelized (40 CPU cores)	2.42	1.61
Normal Conv + Parallelized (4 CPU cores)	7.12	4.23
Unrolled Conv	7.01	4.94
Unrolled Conv + Parallelized(40 CPU cores)	3.02	2.25
Unrolled Conv + Parallelized(4 CPU cores)	5.25	3
Cuda Conv	4.06	2.83
Keras	0.6	0.45

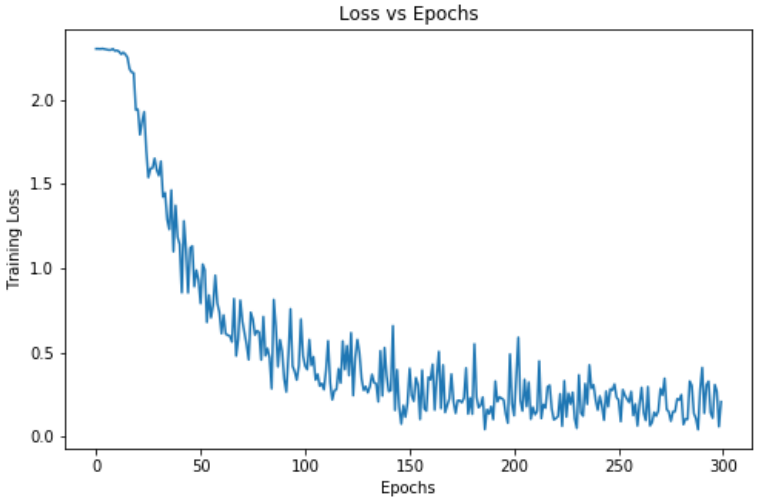
Table 1: Running Speeds (s/it) for different Experiments

7.2 Comparison of Test Accuracy

Implementation Method / Batch Size	64	32
Naive Conv	-	-
Normal Conv	97.06	95.95
Normal Conv + Parallelized (40 CPU cores)	97.00	97.55
Unrolled Conv	97.45	97.75
Unrolled Conv + Parallelized(40 CPU cores)	97.63	97.7
Cuda Conv	96	96.5
Keras	98.7	98.8

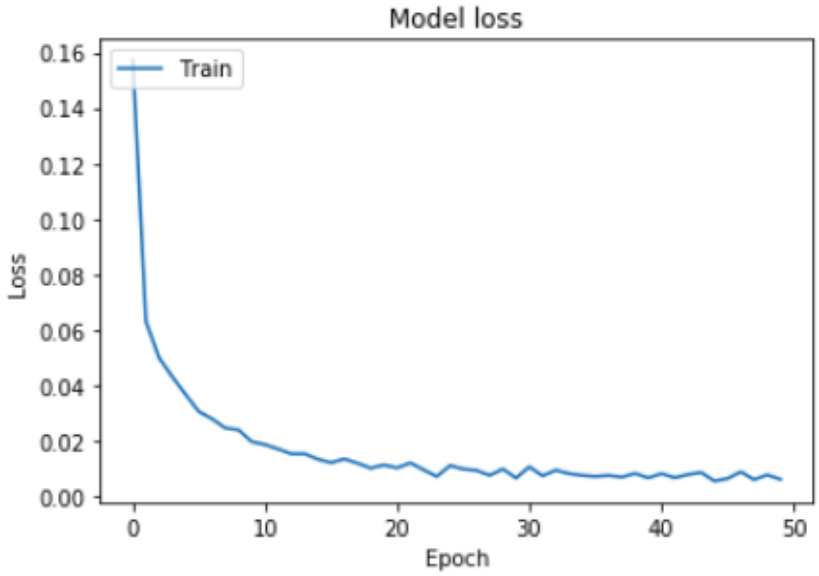
Table 2: Test accuracy for different Experiments

7.3 Training loss vs Epochs (Our implementation)

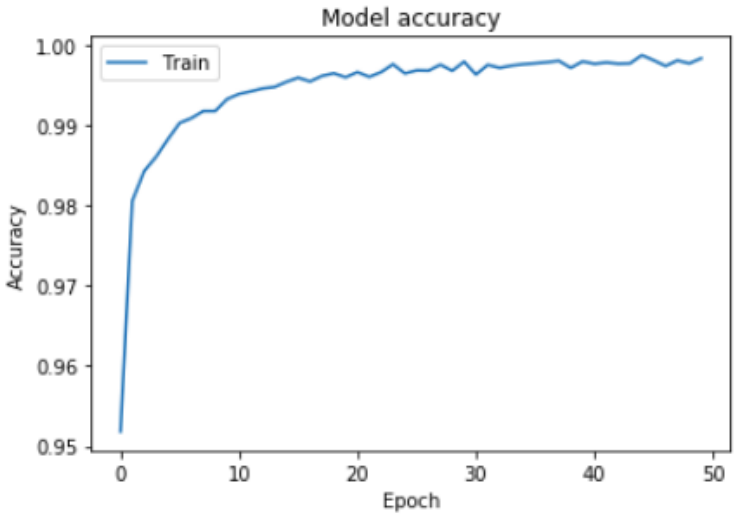


7.4 Recall on Test Set

7.5 Loss vs Epochs (Keras Model)



7.6 Accuracy vs Epochs (Keras Model)



References

[1] Kumar Chellapilla, Sidd Puri, Patrice Simard. **High Performance Convolutional Neural Networks for Document Processing**. Tenth International Workshop on Frontiers in Handwriting Recognition, Université de Rennes 1, Oct 2006, La Baule (France). inria-00112631

[2] **An Introduction to CNNs**, https://victorzhou.com/blog/intro-to-cnns-part-1

[3] **Training a CNN**, https://victorzhou.com/blog/intro-to-cnns-part-2

[4] **CNNs from the ground up**, https://towardsdatascience.com/convolutional-neural-networks-from-the-ground-up-c67bb41454e1

[5] **Convolutions and Backpropagations**, https://medium.com/@pavisj/convolutions-and-backpropagations-46026a8f5d2c

[6] **Derivation of Backpropagation in CNN)**, https://pdfs.semanticscholar.org/5d79/11c93ddcb34cac088d99bd0cae9124e5dcd1.pdf

[7] **CUDA C/C++ Basics Supercomputing 2011 Tutorial**, https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf

[8] **Implementing Convolutions in CUDA**, http://alexminnaar.com/2019/07/12/implementing-convolutions-in-cuda.html

[9] **Accelerating Convolution Operations by GPU**, https://qiita.com/naoyuki_ichimura/items/8c80e67a10d99c2fb53c

[10] **CuPy Documentation**, https://readthedocs.org/projects/cupy/downloads/pdf/stable/