IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Construction and Analysis of LeNet 5 Model From Strach on Mnist Dataset

ADHIP BHATTARAI[1] and BISHAL RIJAL[2]
[1]IOE Thapathali Campus, Kathmandu (e-mail: adhipbh200@gmail.com)
[2]IOE Thapathali Campus, Kathmandu (e-mail: bishalrijal5467@gmail.com)

Corresponding author: Adhip Bhattarai (e-mail: adhipbh200@gmail.com) and Bishal Rijal (e-mail: bishalrijal5467@gmail.com).

ABSTRACT In the realm of deep learning, convolutional neural networks (CNNs) have revolutionized the field of image recognition and classification. LeNet-5, a pioneering architecture designed by Yann LeCun et al., laid the foundation for modern CNNs and remains a fundamental model for understanding their structure and functionality. This article presents an in-depth exploration of the LeNet-5 architecture and demonstrates its implementation on the MNIST dataset—a widely used benchmark in the realm of handwritten digit recognition. Through this tutorial, readers will gain insights into the layers, concepts, and principles that constitute LeNet-5, fostering a deeper understanding of how CNNs work.

INDEX TERMS CNN, deep learning, Lenet-5, MNIST-dataset

I. INTRODUCTION

CONVOLUTIONAL Neural Networks (CNNs) have emerged as a pivotal advancement in the field of machine learning, particularly in tasks involving image analysis and recognition. The inception of the LeNet-5 architecture by Yann LeCun et al. marked a significant milestone in the evolution of CNNs. LeNet-5 was originally designed for handwritten digit recognition and paved the way for subsequent developments in deep learning. Its elegant yet effective design showcased the power of convolutional layers, pooling operations, and fully connected layers in extracting meaningful features from images and making accurate predictions.

In this article, we embark on a journey to demystify the LeNet-5 architecture, unraveling its intricate layers and mechanisms. Through a step-by-step approach, we'll guide you in constructing the LeNet-5 model from scratch using TensorFlow/Keras and training it on the MNIST dataset. By delving into the architectural nuances, activation functions, and layer interactions, we aim to equip you with a solid grasp of how LeNet-5 functions at its core.

This tutorial assumes a basic understanding of neural networks and machine learning concepts. Whether you're a newcomer seeking to comprehend the inner workings of CNNs or an experienced practitioner aiming to revisit the roots of deep learning, this article will serve as a comprehensive resource to enhance your knowledge and practical skills.

In the following sections, we'll dive into the architectural components of LeNet-5, explain its layer-by-layer construction, demonstrate the implementation process, and finally, present the results of applying LeNet-5 to the MNIST dataset.

II. METHODOLOGY

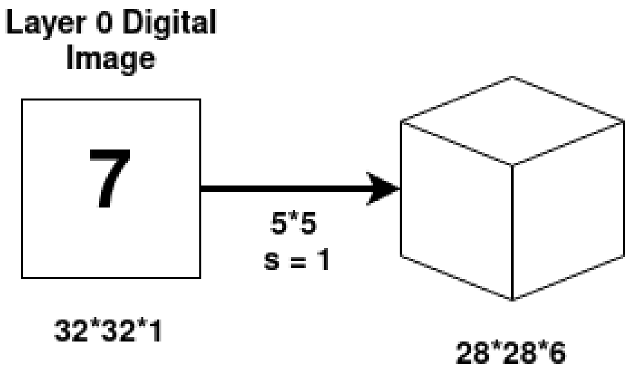
A. THEORY

Convolutional neural networks (CNNs) and related architectures are the go-to architecture for computer vision tasks. They are heavily inspired by our own visual cortex. Computer scientists once studied the human brain and then transferred that knowledge to computers. Around the 1960s, two researchers, Hubel and Wiesel, performed a series of experiments on the brains of half-awake cats. They showed different shapes to the cats and noticed that one particular neuron, now known as the "Hubel and Wiesel neuron," was active for specific cells. During their research, they also discovered two types of cells in the visual cortex: simple cells, which detect features, and complex cells, which have a larger receptive field and can summarize the output of simple cells. This groundbreaking work laid the foundation for the development of Convolutional Neural Networks, computer scientist took inspiration from this work revolutionizing computer vision and pattern recognition tasks.

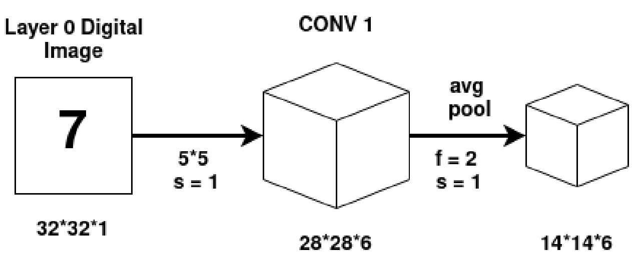
The main useful features of CNNs are shift invariance and parameter sharing which make them useful for tasks like image classification, object detection, segmentation and many more.

Typical CNN architecture consists of a stack of convolution, pooling, and activation layers, and finally fully connected dense layers. Different permutations and combinations of these layers, kernel sizes, pooling operations, regularization, normalization, and a few other techniques have resulted in different architectures.

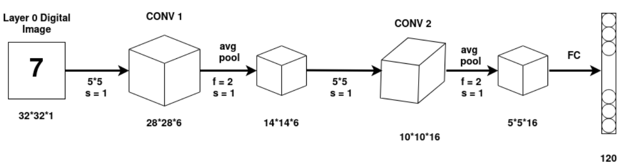
The network has 5 layers with learnable parameters and hence named Lenet-5. [1] It has three set of convolution layers with a combination of average pooling. After the convolution and average pooling layers, we have two fully connected layers. At last, a softmax classifier which classifies the images into respective class. The input of this model



is a 32*32 grayscale image hence the number of channels is one. Then, the first convolution operation with the filter size 5*5 and it contains 6 filters. As a result, output get a feature map of size 28*28*6. Here, the number of channels is equal to the number of filters applied. After the first pooling



operation, we apply the average pooling and the size of the feature map is reduced by half. Then, next convolution layer with sixteen filters of size 5*5 also the feature map changed output to 10*10*16. The output size is calculated in a similar manner then average pooling or subsampling layer is applied which again reduce the size of the feature map by half i.e 5*5*16. Then, it has a final convolutional layer of size 5*5



with 120 filters as shown in figure. Leaving the feature map size 1*1*120. After which flatten result is 120 values. It also has a fully connected layer with eighty-four neurons and at last, it has an output layer with ten neurons using softmax activation function since the data have ten classes.

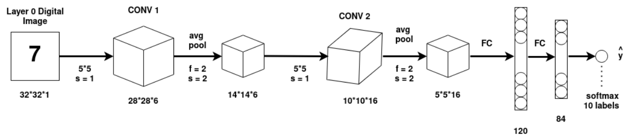


FIGURE 1. Architecture of Lenet-5

B. ARCHITECTURE DETAILS

Layer	# filters / neurons	Filter size	Stride	Size of feature map	Activation function
Input	-	-	-	32 X 32 X 1	
Conv 1	6	5 * 5	1	28 X 28 X 6	tanh
Avg. pooling 1		2 * 2	2	14 X 14 X 6	
Conv 2	16	5 * 5	1	10 X 10 X 16	tanh
Avg. pooling 2		2 * 2	2	5 X 5 X 16	
Conv 3	120	5 * 5	1	120	tanh
Fully Connected 1	-	-	-	84	tanh
Fully Connected 2	-	-	-	10	Softmax

C. CONVOLUTION NEURAL NETWORK (CNN)

Convolutional Neural Network (ConvNet or CNN) is a type of deep learning algorithm that is mainly used in image and video recognition tasks. It works by applying a set of filters to the input data, creating feature maps which are then processed through multiple layers to produce a prediction. The key features of CNNs include their ability to learn hierarchical representations of data, local connectivity, shared weights, and pooling operations. These properties allow CNNs to effectively extract features from images and video frames, making them well-suited for tasks such as object detection, image classification, and segmentation. The role of all these topics in this project is discussed below:

- ***Convolution Layer***:
  The primary purpose of Convolution in the case of a ConvNet, was to extract features from the input image and create a compact representation that was usable for further processing by the rest of the network. Convolution preserves the spatial relationship between pixels by learning image features using small squares of input data. It was a process where a small matrix of numbers (called kernel or filter) was taken and was passed over the image and transformed it based on the values from the filter. In a typical CNN architecture, multiple convolutional layers are stacked together, allowing the network to learn increasingly complex features. The dimensions of the tensors were visualized using the following equation:

$$z^i = w^i.A^{i-1} + b^i \tag{1}$$

$$A^i = g^i.z^i \tag{2}$$

Where,
$z^i = outputoftheneuronslocatedinlayer1$
$w^i = weightsofneuronsinlayer1$
$g^i = activationfunction$
b=bias in layer 1

- ***Activation Functions***:
Activation function is a non-linear mathematical function applied to the output of each neuron in the network. The activation function was used to introduce non-linearity into the Network, allowing it to model complex relationships between the inputs and outputs. Neuron could not learn with just a linear function attached to it. A non- linear activation function would let it learn as per the difference with respect to error. Thus, the following activation function had been decided to be used.
  - ***Softmax Activation Function***:
  The softmax activation function is a commonly used activation function in machine learning and deep learning for multiclass classification problems. It is used to convert the output of a neural network into a probability distribution over multiple classes.
  The softmax function takes a vector of scores as input and returns a vector of probabilities that sum to 1. Each element of the output vector represents the probability that the input belongs to the corresponding class.
  The mathematical formula for the softmax function is as follows:
  $$softmax(z)_i = \frac{e^{z_i}}{\sum_i^N e^{z_i}} \tag{3}$$
  Where,
  Z is the vector of raw outputs from the neural network
  e is Euler's number (approximately 2.71828)
- ***Average Pooling***:
Average pooling is a pooling operation that selects the average value within a small region of a feature map. Average pooling reduces the spatial dimensions of the feature maps, making the network more computationally efficient. Average pooling is typically used in combination with multiple convolutional layers to form a typical CNN architecture, and is applied repeatedly to extract increasingly complex features from the input data.
- ***Fully Connected Layer***:
A fully connected layer is a type of layer that connects every neuron in the previous layer to every neuron in the next layer. It takes the output from the previous layer, which can be the output from one or more convolutional or pooling layers, and applies a linear transformation to the data, followed by a non-linear activation function, such as ReLU. The result is then passed on to the next layer in the network. The purpose of the fully connected layer is to aggregate and interpret the features learned by the previous layers, allowing the network to make predictions based on high-level abstractions of the input data.

III. WORKING PRINCIPLE

A. DATASET PREPARATION

Consider a dataset:

$$X = (x_{ij})_{m*n}$$

having m number of instances with n number of features. For xij, i represents the data and j represents the feature. The MNIST dataset is a collection of pictures of handwritten digits [2]. In this case, "i" would represent the total number of pictures we have, and "j" would represent the different aspects we want to know about each picture. In total, each picture has 785 different aspects, where 784 aspects describe the intensity of individual pixel values in a 28x28 grid (that's 784 pixels in total), and the last aspect tells us which number the handwritten digit represents, ranging from 0 to 9.

B. DATASET PREPROCESSING

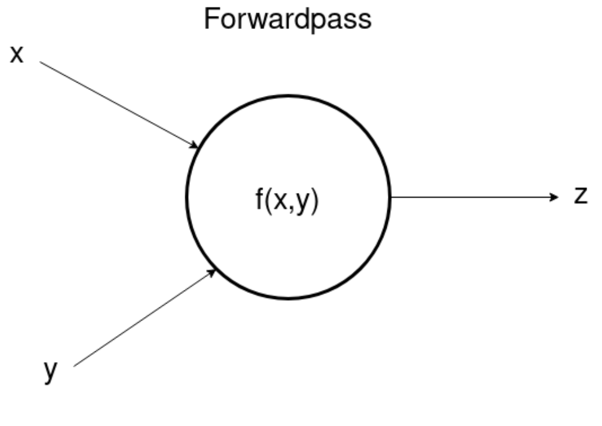
To make sure our analysis of the MNIST dataset is accurate and useful, we need to prepare the data properly. This involves a series of steps that help us get the data ready for processing by machine learning algorithms.

- Label Encoding: Unlike typical datasets with categorical variables, the MNIST dataset contains pixel values that represent the intensity of each pixel in grayscale images. These values are already numerical and don't require label encoding. However, since our output is multiclass, we performed one hot encoding to the output variable.
- Normalization: Normalization is crucial for ensuring that all the pixel values have a consistent scale. In the case of MNIST, pixel values are in the range of 0 to 255 (representing pixel intensity). So, we divided all the input columns by 255 to normalize the pixel values in the range between 0 and 1.
- Handling Missing Values: Fortunately, the MNIST dataset doesn't have missing values, as each image is well-defined with its pixel values.

C. FORWARD PROPAGATION

Forward propagation in a Convolutional Neural Network (CNN) is the process through which input data, such as images, is passed through the network's layers to produce an output prediction. Each layer in the CNN performs specific operations, such as convolutions, activations, and pooling, that transform the input data into a more abstract and representative feature space, eventually leading to a final prediction.

The Forward Pass starts from the Convolutional Layer where there are two things to note, one is the input image and the other is the filter or kernel. Now, a kernel or filter is also known as feature detector which is a set of weights that holds some of the most important features in the input image. A Convolution operation is performed on this layer with the input image and the kernels, then the resulting output also



known as a feature map is obtained. The mathematical form of this operation is given as:

$$(I*K)(i,j) = \sum_{i=1}^{m}\sum_{j=1}^{n}(I_{(i+m-1)(j+n-1)}*rot180(K_{ij}))+b_j \quad (4)$$

The I is the input(image) and K is the kernel(same dimensions) having i,j as indices which represents single pixel in the input image, kernel, and the produced output. m,n are used to represent the dimensions (height, width) of the filter or kernel. The $I_{(i+m-1)(j+n-1)}$ represents the specific pixel value of the input image at the position (i+m-1,j+n-1). As i and j change, the filter is applied through each of the input pixels. There are two sums, the outer summation $\sum_{i=1}^{m}$ is used to slide the kernel over the input image which iterates over i from 0 to m-1 slides vertically over the input image. Meanwhile, the inner summation $\sum_{j=1}^{n}$ is used for element-wise multiplication which iterates over j from 0 to n-1, representing the horizontal sliding of the kernel over the image. Finally, a bias b_i is added for providing flexibility for the network.

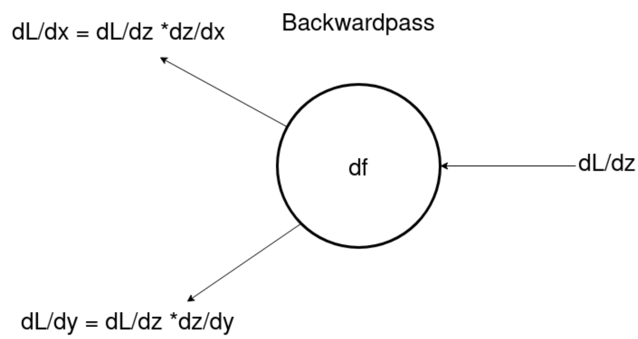
- ***Forward Pass in Pooling Layer***
  Feature map is obtained after convolving the input image and kernel, now the next process is to apply pooling which is done on the pooling layer.
  In case of **Average Pooling**, a window slides over the input feature map and the average value within each window is selected as the output.
  (I*K)(i,j) = $\begin{bmatrix} 23 & 14 & 25 \\ 27 & 29 & 35 \end{bmatrix}$
  When applying average pooling of shape 2*2, we get,
  AvgPool((I*K)(i,j)) = [avg(23,14,27,29) avg(14,25,29,35]
  AvgPool((I*K)(i,j)) = [23 25]
- ***Forward Pass in Flattening***
  Here the output obtained from convolution and Pooling is transformed into a one-dimensional vector so that it can be passed to the Dense or Fully connected Layer.
- ***Forward Pass in Fully Connected Layer***
  A full connection layer is something the same as a Multi-Layer Perceptron (MLP). In a Fully Connected Layer, each neuron is connected to every neuron in the previous layer, forming a fully interconnected network. The term "Fully Connected" is specifically used in CNNs to distinguish it from other layers we have discussed. Unlike convolutional and pooling layers that share weights across inputs, the Fully Connected Layer assigns unique weights to each input value.
  The forward pass on each neuron of the Fully Connected Layer is represented as:

$$output = \sigma(W_{ij}.X_j + bJ) \quad (5)$$

  Where, W = Weights and X = inputs from the preceding layer, σ = Activation function. This equation is simple and we have learned it when starting the basics of Neural Networks. Here we are computing the weighted sum of inputs, and adding a bias.

D. BACK PROPAGATION IN CNN

Backpropagation in a Convolutional Neural Network (CNN) is the process of updating the network's weights and biases based on the calculated gradients of the loss function with respect to these parameters. This process enables the network to learn from the training data by iteratively adjusting its parameters to minimize the prediction error. [3]



- ***Backward Propagation in Fully Connected Layer***
  The fully connected layer has two parameters - weight matrix and bias matrix. Calculating the change in error with respect to weights $-\partial E/\partial W$.
  Since the error is not directly dependent on the weight matrix, the concept of chain rule is used to find this value as:

$$\partial E/\partial W = \partial E/\partial O.\partial O/\partial Z_2.\partial z/\partial W \quad (6)$$

  - ***Change in error with respect to output***
    Suppose the actual values for the data are denoted as y' and the predicted output is represented as O. Then the error would be given by this equation:

$$E = (y' - O)^2/2$$

    If differentiate the error with respect to the output, it will get the following equation:

$$\partial E/\partial O = -(y' - O)$$

  - ***Change in output with respect to Z_2(linear transformation output)***

To find the derivative of output O with respect to Z_2, we must first define O in terms of Z_2. If you look at the computation graph from the forward propagation section above, you would see that the output is simply the sigmoid of Z2. Thus, $\partial O/\partial Z_2$ is effectively the derivative of Sigmoid. Recall the equation for the Sigmoid function:

$$f(x) = 1/(1+e^{-x}$$

The derivative of this function comes out to be:

$$f'(X) = (1+e^{-x})^{-1}[1-(1+e^{-x})^{-1}]$$

$$f'(x) = sigmoid(x)(1-sigmoid(x))$$

$$\partial O/\partial Z_2 = (O)(1-O)$$

- ***Change in Z_2 with respect to Weights***
  The value of Z_2 is the result of the linear transformation process. Here is the equation of Z_2 in terms of weights:

  $$Z_2 = W^T.A_1 + b$$

  On differentiating Z_2 with respect to W, we will get the value A_1 itself:

  $$\partial Z_2/\partial W = A_1$$

  After finding the individual derivation, the chain rule is applied to find the change in error with respect to weights:

  $$\partial E/\partial W = \partial E/\partial O.\partial O/\partial Z_2.\partial Z_2/\partial W$$

  $$\partial E/\partial W = -(y'-O).sigmoid'.A_1$$

  The shape of $\partial E/\partial W$ will be the same as the weight matrix W. We can update the values in the weight matrix using the following equation:

  $$W_{new} = W_{old} - lr * \partial E/\partial W$$

- ***Backward Propagation in Convolution Layer*** For the convolution layer, It had the filter matrix as our parameter. During the forward propagation process, it randomly initialized the filter matrix. Here, It will now update these values using the following equation:
  $new_{parameter} = old_{parameter} - (learning_{rate} * gradient_{of-parameter}$
  To update the filter matrix, the gradient of the parameter -$\partial E/\partial f$ need to be find out. The derivative of $\partial E/\partial f$ can be defined as :

  $$\partial E/\partial f = \partial E/\partial O.\partial O/\partial Z_2.\partial Z_2/\partial A_1.\partial A_1/\partial Z_1.\partial Z_1/\partial f$$

  The values of the derivatives can be calculated as:

  - ***Change in Z_2 with respect to A_1***
    To find the value for $\partial Z_2/\partial A_1$, it need to have the equation for $Z-2$ in terms of A_1:

    $$Z_2 = W^2.A_1 + b$$

    On differentiating the above equation with respect to A_1, we get $W^T astheresult$:

    $$\partial Z_2/\partial A_1 = W^T$$

  - ***Change in A_1 with respect to Z_1***
    The next value that need to determine is $\partial A_1/\partial Z_1$.

    $$A_1 = sigmoid(Z_1)$$

    This is simply the Sigmoid function. The derivative of Sigmoid would be:

    $$\partial A_1/\partial Z_1 = (A_1)(1-A_1)$$

  - Change in Z_1 with respect to filter f
    Finally, there is the need of value for $\partial Z_1/\partial f$.

    $$Z_1 = X * f$$

    Differentiating Z with respect to X will simply give X:

    $$\partial Z_1/\partial f = X$$

    Now replacing all the derivatives, overall change in error with respect to the filter:

    $$\partial E/\partial f = \partial E/\partial O.\partial O/\partial Z_2.\partial Z_2/\partial A_1.\partial A_1/\partial Z_1.\partial Z_1/\partial f$$

    Once the value for $\partial E/\partial f$ is figure out, this value is used to update the orginal filter value:

    $$f = f - lr * (\partial E/\partial f)$$

IV. RESULTS

The mnist Dataset is the standard since ages and we are able to use it in our project so that we can analyze the model properly with the predefined model using Pytorch and conquer the outcomes. This is the sample example of mnist dataset as it contains all the handwritten digits as data upto 10 classes (0 to 9) as shown in Figure 2. The pixel intensity values of the sample give the actual status of data as shown in Figure 3 The complete information of the data in dataset is visualized in Figure 4. The Actual Architecture Used can be visualized from Figure 5 as explained in the methodolody of the report. After, building the architecture, the dataset is split into the training and testing dataset which are used in the training and validation as well as verification propose.

Since, We have constructed our own Lenet-5 Model from Strach as well as Lenet-5 Model using Pytorch which is very easy to implement as all the modules are already specified. For the Pytorch's Lenet Model, we obtain the accuracy of 98.722 and Loss of 0.04889 in Just 11 Epochs Which is satisfactory. We can also visualize it from Performance Matrics:

We evaluated both models using standard performance metrics, including accuracy and loss, on both the training and testing datasets. We constructed Graph of Accuracy Graph as in Figure 6 and Loss Graph as in Figure 7.

LeNet-5 Implementation from Scratch:

The Dataset is similarly divided into the training and testing dataset as we construct the lenet model similar to the

Pytorch's model also including loss function - Cross-entropy loss which is absent in orginal model. After Training For 25 Epochs, Result's are complementary and satisfactory as we obtain the traib accuracy of 90.1266 and test accuracy of 90.27 as shown in Figure 9. We constructed Loss Graph of Manual Model as shown in Figure 8 which in comparision to pytorch model is noisy but if increase the batch size then the noise is reduced as well as pytorch model contains auto adjust function which will adjust the values so that accuracy and loss is maintained.

The Predicted Output of the model is shown in Figure 10 as we can see that pixel intensity of the model is complementary and the model predicted actual value as right.

The model is performing perfectly with no single incorrect prediction as above. One can run more inference to find the incorrect one, but overall the model's accuracy is acceptable.

It's clearly visible with the attribution scores calculated by Integrated Gradients what part of the input image the model's parameters are giving weightage to as shown in Figure 11

V. DISCUSSION

The MNIST dataset, a steadfast benchmark in the machine learning landscape, serves as a reliable foundation for evaluating model performance. The dataset's enduring relevance and standardized structure make it an ideal testing ground for assessing the capabilities of digit recognition models. Each image in the dataset encapsulates a handwritten digit from 0 to 9, encompassing ten distinct classes. The pixel intensity values within these images encapsulate the core essence of the handwritten digits, as artfully depicted in Figure 2.

As we delve into model architecture, the venerable LeNet-5 takes the center stage—a neural network design that has withstood the test of time. Figure 5 visually encapsulates this architecture, which we have thoughtfully applied through two distinct lenses: a manual construction from the ground up and the utilization of PyTorch's pre-existing LeNet-5 model. To allow for both model training and validation, we partitioned the dataset into training and testing subsets.

Comparing PyTorch's LeNet-5 with Manual Implementation:

The spotlight shines brightly on PyTorch's LeNet-5, showcasing its exceptional prowess in the realm of digit recognition. With an astounding accuracy of 98.722 achieved in a mere 11 epochs, PyTorch's design, fortified with sophisticated optimization techniques, delivers results that echo its efficiency. Notably, the diminutive loss value of 0.04889 adds further credence to the prowess of PyTorch's machinery. The vivid accuracy and loss graphs (Figures 6 and 7) beautifully illustrate the model's unwavering convergence within a compact temporal window.

On the manual construction front, our LeNet-5 model presents a commendable training accuracy of 90.1266, harmonized with a testing accuracy of 90.27, reached after a patient span of 25 epochs. These achievements, while slightly trailing behind PyTorch's model, demonstrate the potential of a hands-on approach to model creation. It's important to note that the loss graph (Figure 8) exhibits subtle fluctuations, which might be tamed by refining the batch size. Additionally, the intriguing aspect of PyTorch's self-adjusting capabilities contributes to its stable accuracy and loss trajectories.

VI. CONCLUSION

In the captivating journey through the corridors of the LeNet-5 architecture applied to the venerable MNIST dataset, a tapestry of insights emerges, weaving together the threads of manual ingenuity and framework sophistication. This fusion of methods engenders a profound understanding of the quintessence of deep learning model development.

The MNIST dataset, standing as a paragon of enduring relevance, enriches our narrative by providing a comprehensive landscape for evaluating model efficacy. Its standardized nature and portrayal of handwritten digits from 0 to 9 establish a fertile ground for our explorations.

The symphony of model architecture resonates through LeNet-5, an emblem of neural network innovation. This musical piece unfolds through two perspectives: the meticulous construction from scratch and the embrace of PyTorch's preconceived masterpiece. The partitioning of the dataset into training and testing domains cultivates an environment where these models can flourish and be scrutinized.

In the spotlight, PyTorch's LeNet-5 shines as a virtuoso. Its swift ascent to an accuracy zenith of 98.722 within a mere 11 epochs serves as a testament to PyTorch's orchestration of optimization techniques. The orchestral ensemble is enriched by a minuscule loss of 0.04889, attesting to the harmony between its components. Figures 6 and 7 illuminate the performance journey with vivid clarity, encapsulating the model's graceful convergence.

In contrast, our manually sculpted LeNet-5 model, though slightly trailing in accuracy at 90.27, underscores the value of crafting a neural masterpiece from scratch. This creation echoes the meticulous craftsmanship of artisans, offering an alternative lens to model development. The animated loss graph in Figure 8 is a visual symphony of its own, illustrating the model's journey through epochs.

This symposium of results and discourse concludes by casting a spotlight on the artistry of deep learning. PyTorch's harmonious orchestration and our artisanal endeavor intersect, offering practitioners a dynamic spectrum of choices. As we tread the path ahead, the harmonization of personalized craftsmanship and the efficiency of frameworks beckons us, promising an orchestra of discoveries waiting to be composed in the realm of machine learning.

REFERENCES

[1] Y. LeCun et al., “Lenet-5, convolutional neural networks,” URL: http://yann. lecun. com/exdb/lenet, vol. 20, no. 5, p. 14, 2015.
[2] H. Xiao, K. Rasul, and R. Vollgraf, “Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms,” arXiv preprint arXiv:1708.07747, 2017.
[3] C.-C. J. Kuo, “Understanding convolutional neural networks with a mathematical model,” Journal of Visual Communication and Image Representation, vol. 41, pp. 406–413, 2016.

ADHIP BHATTARAI is a dedicated individual pursuing a Bachelor’s degree in Computer Engineering at Tribhuvan University. With a strong passion for machine learning and data science, he is constantly exploring the latest advancements in these fields. Although he may not have notable accomplishments just yet, Adhip’s enthusiasm and drive for learning and applying cutting-edge technologies make him a promising and ambitious individual in the world of computer engineering.

BISHAL RIJAL is a dedicated individual currently studying Bachelor’s in Computer and Technology at Tribhuvan University. Bishal’s enthusiasm for research and innovation has led him to undertake various projects and engage in practical applications of his knowledge. He continually seeks to deepen in understanding of the subject matter, staying up-to-date with the latest advancements and trends. With his relentless determination, inquisitive mindset, and expertise in machine learning and data science, Bishal Rijal is poised to make significant contributions to the ever-evolving field of technology.

VII. APPENDIX

A. APPENDIX A: FIGURES

://www.overleaf.com/project/64d0e36f16fee799786083e4

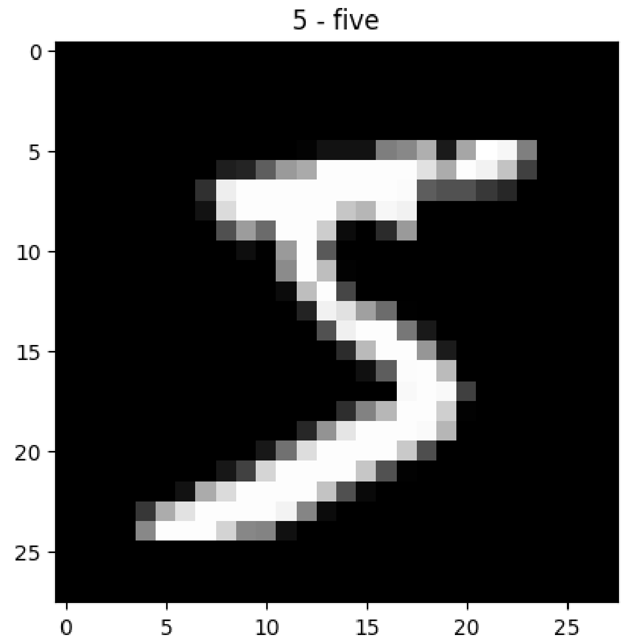


FIGURE 2. Orginal Sample of Mnistnet dataset

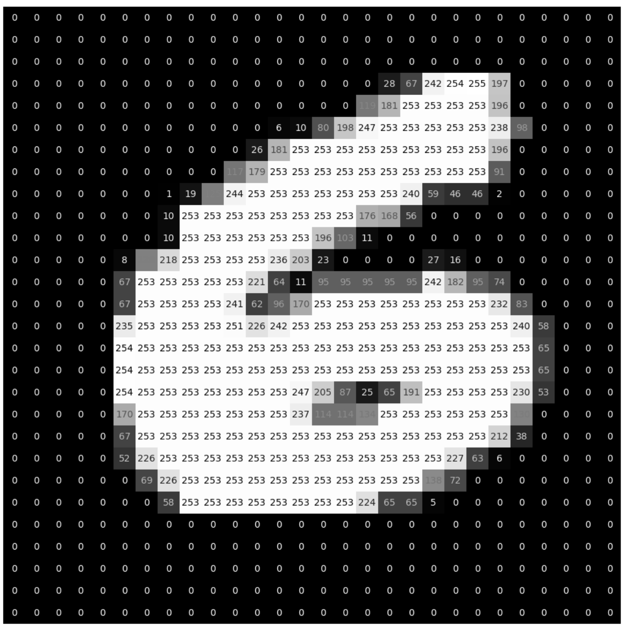


FIGURE 3. Pixel Hierarchy of Samples

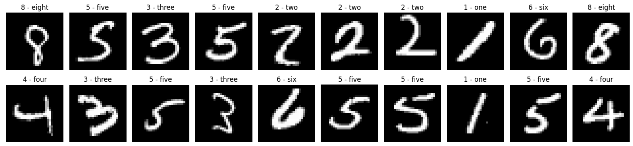
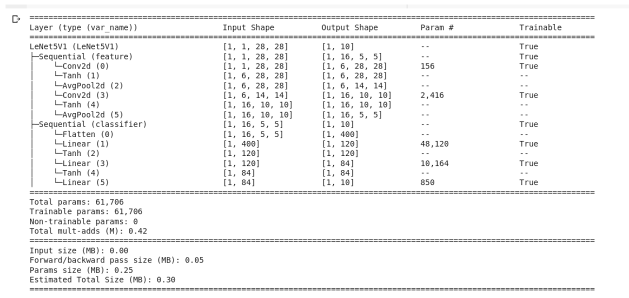


FIGURE 4. Preprocessed Samples of Mnist Dataset



```
==========================================================================================
Layer (type (var_name))              Input Shape        Output Shape       Param #        Trainable
==========================================================================================
LeNet5V1 (LeNet5V1)                  [1, 1, 28, 28]     [1, 10]            --             True
├─Sequential (feature)               [1, 1, 28, 28]     [1, 16, 5, 5]      --             True
│    └─Conv2d (0)                    [1, 1, 28, 28]     [1, 6, 28, 28]     156            True
│    └─Tanh (1)                      [1, 6, 28, 28]     [1, 6, 28, 28]     --             --
│    └─AvgPool2d (2)                 [1, 6, 28, 28]     [1, 6, 14, 14]     --             --
│    └─Conv2d (3)                    [1, 6, 14, 14]     [1, 16, 10, 10]    2,416          True
│    └─Tanh (4)                      [1, 16, 10, 10]    [1, 16, 10, 10]    --             --
│    └─AvgPool2d (5)                 [1, 16, 10, 10]    [1, 16, 5, 5]      --             --
├─Sequential (classifier)            [1, 16, 5, 5]      [1, 10]            --             True
│    └─Flatten (0)                   [1, 16, 5, 5]      [1, 400]           --             --
│    └─Linear (1)                    [1, 400]           [1, 120]           48,120         True
│    └─Tanh (2)                      [1, 120]           [1, 120]           --             --
│    └─Linear (3)                    [1, 120]           [1, 84]            10,164         True
│    └─Tanh (4)                      [1, 84]            [1, 84]            --             --
│    └─Linear (5)                    [1, 84]            [1, 10]            850            True
==========================================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
Total mult-adds (M): 0.42
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.05
Params size (MB): 0.25
Estimated Total Size (MB): 0.30
==========================================================================================
```

FIGURE 5. Architecture of Lenet-5

FIGURE 6. Accuracy Graph of Pytorch Lenet-5

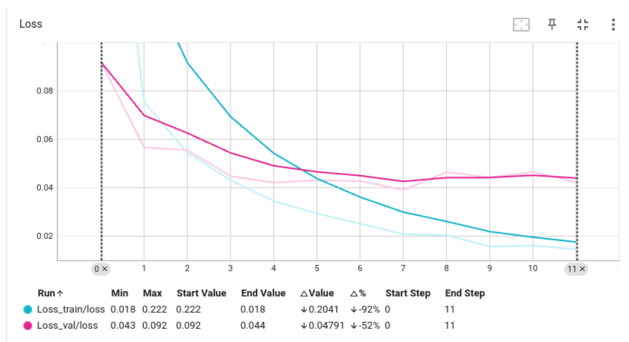


FIGURE 7. Loss Graph of Pytorch Lenet-5

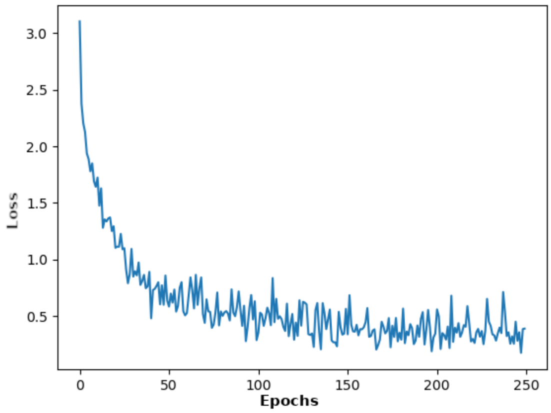


FIGURE 8. Loss Graph of Manual Lenet-5

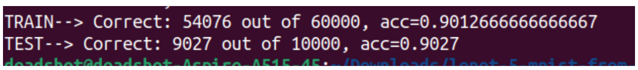


FIGURE 9. Train Accuracy Vs Loss Accuracy of Manual Lenet-5

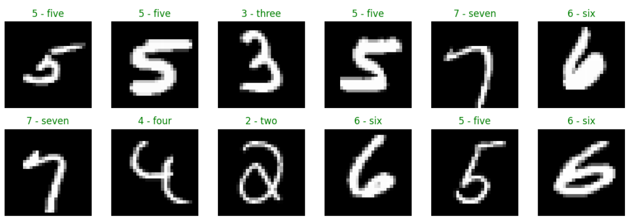


FIGURE 10. Predicted Output of Lenet-5

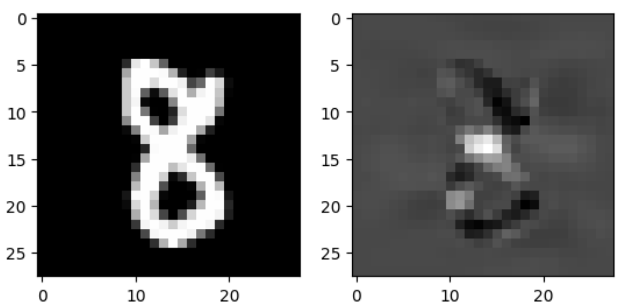


FIGURE 11. Predicted output Gradient View of Lenet-5

B. APPENDIX B: CODE

```
import pickle
import random
import numpy as np
import matplotlib.pyplot as plt
from abc import ABCMeta, abstractmethod

filename = [
  ["training_images","train-images-idx3-ubyte.gz"],
  ["test_images","t10k-images-idx3-ubyte.gz"],
  ["training_labels","train-labels-idx1-ubyte.gz"],
  ["test_labels","t10k-labels-idx1-ubyte.gz"]
]

def download_mnist():
    base_url = "http://yann.lecun.com/exdb/mnist/"
    for name in filename:
        print("Downloading "+name[1]+"...")
        request.urlretrieve(base_url+name[1], name[1])
    print("Download complete.")

def save_mnist():
    mnist = {}
    for name in filename[:2]:
        with gzip.open(name[1], 'rb') as f:
            mnist[name[0]] = np.frombuffer(f.read(), np.uint8, offset=16).reshape(-1,28*28)
    for name in filename[-2:]:
        with gzip.open(name[1], 'rb') as f:
            mnist[name[0]] = np.frombuffer(f.read(), np.uint8, offset=8)
    with open("mnist.pkl", 'wb') as f:
        pickle.dump(mnist,f)
    print("Save complete.")

def init():
    download_mnist()
    save_mnist()

def load():
    with open("mnist.pkl",'rb') as f:
        mnist = pickle.load(f)
    return mnist["training_images"], mnist["training_labels"], mnist["test_images"], mnist["test_labels"]

def MakeOneHot(Y, D_out):
    N = Y.shape[0]
    Z = np.zeros((N, D_out))
    Z[np.arange(N), Y] = 1
    return Z

def draw_losses(losses):
    t = np.arange(len(losses))
    plt.plot(t, losses)
    plt.show()

def get_batch(X, Y, batch_size):
    N = len(X)
    i = random.randint(1, N-batch_size)
    return X[i:i+batch_size], Y[i:i+batch_size]

class FC():
    """
    Fully connected layer
    """
    def __init__(self, D_in, D_out):
        #print("Build FC")
        self.cache = None
        #self.W = {'val': np.random.randn(D_in, D_out), 'grad': 0}
        self.W = {'val': np.random.normal(0.0, np.sqrt(2/D_in), (D_in,D_out)), 'grad': 0}
        self.b = {'val': np.random.randn(D_out), 'grad': 0}

    def _forward(self, X):
        #print("FC: _forward")
        out = np.dot(X, self.W['val']) + self.b['val']
        self.cache = X
        return out

    def _backward(self, dout):
        #print("FC: _backward")
        X = self.cache
        dX = np.dot(dout, self.W['val'].T).reshape(X.shape)
        self.W['grad'] = np.dot(X.reshape(X.shape[0], np.prod(X.shape[1:])).T, dout)
        self.b['grad'] = np.sum(dout, axis=0)
        #self._update_params()
        return dX

    def _update_params(self, lr=0.001):
        # Update the parameters
        self.W['val'] -= lr*self.W['grad']
        self.b['val'] -= lr*self.b['grad']

class ReLU():
    """
    ReLU activation layer
    """
    def __init__(self):
        #print("Build ReLU")
        self.cache = None

    def _forward(self, X):
        #print("ReLU: _forward")
        out = np.maximum(0, X)
        self.cache = X
        return out

    def _backward(self, dout):
        #print("ReLU: _backward")
        X = self.cache
        dX = np.array(dout, copy=True)
        dX[X <= 0] = 0
        return dX

class Sigmoid():
    """
    Sigmoid activation layer
    """
    def __init__(self):
        self.cache = None

    def _forward(self, X):
        self.cache = X
        return 1 / (1 + np.exp(-X))

    def _backward(self, dout):
        X = self.cache
        dX = dout*X*(1-X)
        return dX

class tanh():
    """
    tanh activation layer
    """
    def __init__(self):
        self.cache = X

    def _forward(self, X):
```

```
        self.cache = X
        return np.tanh(X)

    def _backward(self, X):
        X = self.cache
        dX = dout*(1 - np.tanh(X)**2)
        return dX

class Softmax():
    """
    Softmax activation layer
    """
    def __init__(self):
        #print("Build Softmax")
        self.cache = None

    def _forward(self, X):
        #print("Softmax: _forward")
        maxes = np.amax(X, axis=1)
        maxes = maxes.reshape(maxes.shape[0], 1)
        Y = np.exp(X - maxes)
        Z = Y / np.sum(Y, axis=1).reshape(Y.shape[0], 1)
        self.cache = (X, Y, Z)
        return Z # distribution

    def _backward(self, dout):
        X, Y, Z = self.cache
        dZ = np.zeros(X.shape)
        dY = np.zeros(X.shape)
        dX = np.zeros(X.shape)
        N = X.shape[0]
        for n in range(N):
            i = np.argmax(Z[n])
            dZ[n,:] = np.diag(Z[n]) - np.outer(Z[n],Z[n])
            M = np.zeros((N,N))
            M[:,i] = 1
            dY[n,:] = np.eye(N) - M
        dX = np.dot(dout,dZ)
        dX = np.dot(dX,dY)
        return dX

class Dropout():
    """
    Dropout layer
    """
    def __init__(self, p=1):
        self.cache = None
        self.p = p

    def _forward(self, X):
        M = (np.random.rand(*X.shape) < self.p) / self.p
        self.cache = X, M
        return X*M

    def _backward(self, dout):
        X, M = self.cache
        dX = dout*M/self.p
        return dX

class Conv():
    """
    Conv layer
    """
    def __init__(self, Cin, Cout, F, stride=1, padding=0, bias=True):
        self.Cin = Cin
        self.Cout = Cout
        self.F = F
        self.S = stride
        #self.W = {'val': np.random.randn(Cout, Cin, F, F), 'grad': 0}
        self.W = {'val': np.random.normal(0.0,np.sqrt(2/Cin),(Cout,Cin,F,F)), 'grad': 0} # Xavier Initialization
        self.b = {'val': np.random.randn(Cout), 'grad': 0}
        self.cache = None
        self.pad = padding

    def _forward(self, X):
        X = np.pad(X, ((0,0),(0,0),(self.pad,self.pad),(self.pad,self.pad)), 'constant')
        (N, Cin, H, W) = X.shape
        H_ = H - self.F + 1
        W_ = W - self.F + 1
        Y = np.zeros((N, self.Cout, H_, W_))

        for n in range(N):
            for c in range(self.Cout):
                for h in range(H_):
                    for w in range(W_):
                        Y[n, c, h, w] = np.sum(X[n, :, h:h+self.F, w:w+self.F] * self.W['val'][c, :, :, :]) + self.b['val'][c]

        self.cache = X
        return Y

    def _backward(self, dout):
        # dout (N,Cout,H_,W_)
        # W (Cout, Cin, F, F)
        X = self.cache
        (N, Cin, H, W) = X.shape
        H_ = H - self.F + 1
        W_ = W - self.F + 1
        W_rot = np.rot90(np.rot90(self.W['val']))

        dX = np.zeros(X.shape)
        dW = np.zeros(self.W['val'].shape)
        db = np.zeros(self.b['val'].shape)

        # dW
        for co in range(self.Cout):
            for ci in range(Cin):
                for h in range(self.F):
                    for w in range(self.F):
                        dW[co, ci, h, w] = np.sum(X[:,ci,h:h+H_,w:w+W_] * dout[:,co,:,:])

        # db
        for co in range(self.Cout):
            db[co] = np.sum(dout[:,co,:,:])

        dout_pad = np.pad(dout, ((0,0),(0,0),(self.F,self.F),(self.F,self.F)), 'constant')
        #print("dout_pad.shape: " + str(dout_pad.shape))
        # dX
        for n in range(N):
            for ci in range(Cin):
                for h in range(H):
                    for w in range(W):
                        #print("self.F.shape: %s", self.F)
                        #print("%s, W_rot[:,ci,:,:].shape: %s, dout_pad[n,:,h:h+self.F,w:w+self.F].shape: %s" % ((n,ci,h,w),W_rot[:,ci,:,:].shape, dout_pad[n,:,h:h+self.F,w:w+self.F].shape))
                        dX[n, ci, h, w] = np.sum(W_rot[:,ci,:,:] * dout_pad[n, :, h:h+self.F,w:w+self.F])

        return dX
```

```
class MaxPool():
    def __init__(self, F, stride):
        self.F = F
        self.S = stride
        self.cache = None

    def _forward(self, X):
        # X: (N, Cin, H, W): maxpool along 3rd, 4th dim
        (N,Cin,H,W) = X.shape
        F = self.F
        W_ = int(float(W)/F)
        H_ = int(float(H)/F)
        Y = np.zeros((N,Cin,W_,H_))
        M = np.zeros(X.shape) # mask
        for n in range(N):
            for cin in range(Cin):
                for w_ in range(W_):
                    for h_ in range(H_):
                        Y[n,cin,w_,h_] = np.max(X[n,cin,F*w_:F*(w_+1),F*h_:F*(h_+1)])
                        i,j = np.unravel_index(X[n,cin,F*w_:F*(w_+1),F*h_:F*(h_+1)].argmax(), (F,F))
                        M[n,cin,F*w_+i,F*h_+j] = 1
        self.cache = M
        return Y

    def _backward(self, dout):
        M = self.cache
        (N,Cin,H,W) = M.shape
        dout = np.array(dout)
        #print("dout.shape: %s, M.shape: %s" % (dout.shape, M.shape))
        dX = np.zeros(M.shape)
        for n in range(N):
            for c in range(Cin):
                #print("(n,c): (%s,%s)" % (n,c))
                dX[n,c,:,:] = dout[n,c,:,:].repeat(2, axis=0).repeat(2, axis=1)
        return dX*M

def NLLLoss(Y_pred, Y_true):
    """
    Negative log likelihood loss
    """
    loss = 0.0
    N = Y_pred.shape[0]
    M = np.sum(Y_pred*Y_true, axis=1)
    for e in M:
        #print(e)
        if e == 0:
            loss += 500
        else:
            loss += -np.log(e)
    return loss/N

class CrossEntropyLoss():
    def __init__(self):
        pass

    def get(self, Y_pred, Y_true):
        N = Y_pred.shape[0]
        softmax = Softmax()
        prob = softmax._forward(Y_pred)
        loss = NLLLoss(prob, Y_true)
        Y_serial = np.argmax(Y_true, axis=1)
        dout = prob.copy()
        dout[np.arange(N), Y_serial] -= 1
        return loss, dout

class SoftmaxLoss():
    def __init__(self):
        pass
```

```

    def get(self, Y_pred, Y_true):
        N = Y_pred.shape[0]
        loss = NLLLoss(Y_pred, Y_true)
        Y_serial = np.argmax(Y_true, axis=1)
        dout = Y_pred.copy()
        dout[np.arange(N), Y_serial] -= 1
        return loss, dout

class Net(metaclass=ABCMeta):
    # Neural network super class

    @abstractmethod
    def __init__(self):
        pass

    @abstractmethod
    def forward(self, X):
        pass

    @abstractmethod
    def backward(self, dout):
        pass

    @abstractmethod
    def get_params(self):
        pass

    @abstractmethod
    def set_params(self, params):
        pass


class TwoLayerNet(Net):

    #Simple 2 layer NN

    def __init__(self, N, D_in, H, D_out, weights=''):
        self.FC1 = FC(D_in, H)
        self.ReLU1 = ReLU()
        self.FC2 = FC(H, D_out)

        if weights == '':
            pass
        else:
            with open(weights,'rb') as f:
                params = pickle.load(f)
                self.set_params(params)

    def forward(self, X):
        h1 = self.FC1._forward(X)
        a1 = self.ReLU1._forward(h1)
        h2 = self.FC2._forward(a1)
        return h2

    def backward(self, dout):
        dout = self.FC2._backward(dout)
        dout = self.ReLU1._backward(dout)
        dout = self.FC1._backward(dout)

    def get_params(self):
        return [self.FC1.W, self.FC1.b, self.FC2.W, self.FC2.b]

    def set_params(self, params):
        [self.FC1.W, self.FC1.b, self.FC2.W, self.FC2.b] = params


class ThreeLayerNet(Net):

    #Simple 3 layer NN

```

```
    def __init__(self, N, D_in, H1, H2, D_out, weights=''):
        self.FC1 = FC(D_in, H1)
        self.ReLU1 = ReLU()
        self.FC2 = FC(H1, H2)
        self.ReLU2 = ReLU()
        self.FC3 = FC(H2, D_out)

        if weights == '':
            pass
        else:
            with open(weights,'rb') as f:
                params = pickle.load(f)
                self.set_params(params)

    def forward(self, X):
        h1 = self.FC1._forward(X)
        a1 = self.ReLU1._forward(h1)
        h2 = self.FC2._forward(a1)
        a2 = self.ReLU2._forward(h2)
        h3 = self.FC3._forward(a2)
        return h3

    def backward(self, dout):
        dout = self.FC3._backward(dout)
        dout = self.ReLU2._backward(dout)
        dout = self.FC2._backward(dout)
        dout = self.ReLU1._backward(dout)
        dout = self.FC1._backward(dout)

    def get_params(self):
        return [self.FC1.W, self.FC1.b, self.FC2.W, self.FC2.b, self.FC3.W, self.FC3.b]

    def set_params(self, params):
        [self.FC1.W, self.FC1.b, self.FC2.W, self.FC2.b, self.FC3.W, self.FC3.b] = params


class LeNet5(Net):
    # LeNet5

    def __init__(self):
        self.conv1 = Conv(1, 6, 5)
        self.ReLU1 = ReLU()
        self.pool1 = MaxPool(2,2)
        self.conv2 = Conv(6, 16, 5)
        self.ReLU2 = ReLU()
        self.pool2 = MaxPool(2,2)
        self.FC1 = FC(16*4*4, 120)
        self.ReLU3 = ReLU()
        self.FC2 = FC(120, 84)
        self.ReLU4 = ReLU()
        self.FC3 = FC(84, 10)
        self.Softmax = Softmax()

        self.p2_shape = None

    def forward(self, X):
        h1 = self.conv1._forward(X)
        a1 = self.ReLU1._forward(h1)
        p1 = self.pool1._forward(a1)
        h2 = self.conv2._forward(p1)
        a2 = self.ReLU2._forward(h2)
        p2 = self.pool2._forward(a2)
        self.p2_shape = p2.shape
        fl = p2.reshape(X.shape[0],-1) # Flatten
        h3 = self.FC1._forward(fl)
        a3 = self.ReLU3._forward(h3)
        h4 = self.FC2._forward(a3)
        a5 = self.ReLU4._forward(h4)
        h5 = self.FC3._forward(a5)
        a5 = self.Softmax._forward(h5)
        return a5

    def backward(self, dout):
        #dout = self.Softmax._backward(dout)
        dout = self.FC3._backward(dout)
        dout = self.ReLU4._backward(dout)
        dout = self.FC2._backward(dout)
        dout = self.ReLU3._backward(dout)
        dout = self.FC1._backward(dout)
        dout = dout.reshape(self.p2_shape) # reshape
        dout = self.pool2._backward(dout)
        dout = self.ReLU2._backward(dout)
        dout = self.conv2._backward(dout)
        dout = self.pool1._backward(dout)
        dout = self.ReLU1._backward(dout)
        dout = self.conv1._backward(dout)

    def get_params(self):
        return [self.conv1.W, self.conv1.b, self.conv2.W, self.conv2.b, self.FC1.W, self.FC1.b, self.FC2.W, self.FC2.b, self.FC3.W, self.FC3.b]

    def set_params(self, params):
        [self.conv1.W, self.conv1.b, self.conv2.W, self.conv2.b, self.FC1.W, self.FC1.b, self.FC2.W, self.FC2.b, self.FC3.W, self.FC3.b] = params

class SGD():
    def __init__(self, params, lr=0.001, reg=0):
        self.parameters = params
        self.lr = lr
        self.reg = reg

    def step(self):
        for param in self.parameters:
            param['val'] -= (self.lr*param['grad'] + self.reg*param['val'])

class SGDMomentum():
    def __init__(self, params, lr=0.001, momentum=0.99, reg=0):
        self.l = len(params)
        self.parameters = params
        self.velocities = []
        for param in self.parameters:
            self.velocities.append(np.zeros(param['val'].shape))
        self.lr = lr
        self.rho = momentum
        self.reg = reg

    def step(self):
        for i in range(self.l):
            self.velocities[i] = self.rho*self.velocities[i] + (1-self.rho)*self.parameters[i]['grad']
            self.parameters[i]['val'] -= (self.lr*self.velocities[i] + self.reg*self.parameters[i]['val'])


"""
(1) Prepare Data: Load, Shuffle, Normalization, Batching, Preprocessing
"""

#mnist.init()
X_train, Y_train, X_test, Y_test = load()
X_train, X_test = X_train/float(255), X_test/float(255)
X_train -= np.mean(X_train)
X_test -= np.mean(X_test)
```

```

batch_size = 64
D_in = 784
D_out = 10

print("batch_size: " + str(batch_size) + ", D_in: " + str(D_in) + ", D_out: " + str(D_out))

### TWO LAYER NET FORWARD TEST ###
#H=400
#model = nn.TwoLayerNet(batch_size, D_in, H, D_out)
H1=300
H2=100
model = ThreeLayerNet(batch_size, D_in, H1, H2, D_out)


losses = []
#optim = optimizer.SGD(model.get_params(), lr=0.0001, reg=0)
optim = SGDMomentum(model.get_params(), lr=0.0001, momentum=0.80, reg=0.00003)
criterion = CrossEntropyLoss()

# TRAIN
ITER = 2500
for i in range(ITER):
  # get batch, make onehot
  X_batch, Y_batch = get_batch(X_train, Y_train, batch_size)
  Y_batch = MakeOneHot(Y_batch, D_out)

  # forward, loss, backward, step
  Y_pred = model.forward(X_batch)
  loss, dout = criterion.get(Y_pred, Y_batch)
  model.backward(dout)
  optim.step()

  if i % 10 == 0:
    print("%s%% iter: %s, loss: %s" % (10*i/ITER,i, loss))
    losses.append(loss)


# save params
weights = model.get_params()
with open("weights.pkl","wb") as f:
  pickle.dump(weights, f)

draw_losses(losses)



# TRAIN SET ACC
Y_pred = model.forward(X_train)
result = np.argmax(Y_pred, axis=1) - Y_train
result = list(result)
print("TRAIN--> Correct: " + str(result.count(0)) + " out of " + str(X_train.shape[0]) + ", acc=" + str(result.count(0)/X_train.shape[0]))

# TEST SET ACC
Y_pred = model.forward(X_test)
result = np.argmax(Y_pred, axis=1) - Y_test
result = list(result)
print("TEST--> Correct: " + str(result.count(0)) + " out of " + str(X_test.shape[0]) + ", acc=" + str(result.count(0)/X_test.shape[0]))
```

• • •