**3035345306**

**Agrawal Shubhankar**

**COMP3314: Machine Learning**

**Prof. Yizhou Yu**

**Programming Assignment 2**

|  |  |
| --- | --- |
| **Model** | LeNet5 |
| **Datasets Tested** | MNIST |
| **Language** | Python 3.6 |

**Code Structure**

1. model.py
   1. LeNet5
      1. print\_net()
      2. softmax()
      3. softmax\_delta()
      4. save\_layers()
      5. entropy\_loss()
      6. Forward\_Propagation()
      7. Back\_Propagation()
      8. one\_hot\_y()
2. layers.py
   1. LeNetLayer
      1. \_calc\_output\_size()
      2. \_gen\_kernels()
      3. \_load\_path()
      4. print\_layer()
   2. Input
      1. forward\_prop()
      2. backward\_prop()
   3. Convolution
      1. forward\_prop()
      2. backward\_prop()
   4. MaxPooling
      1. forward\_prop()
      2. backward\_prop()
   5. FullyConnected
      1. forward\_prop()
      2. backward\_prop()
3. activations.py
   1. ReLU
      1. forward\_prop()
      2. backward\_prop()
   2. Sigmoid
      1. forward\_prop()
      2. backward\_prop()
   3. tanh
      1. forward\_prop()
      2. backward\_prop()
4. helper.py
   1. Helper
      1. im2col()
      2. col2im()

**Additional Files:**

* Report.docx
* README.md
* layer/
  + C1.npz
  + C3.npz
  + C5.npz
  + F6.npz
  + F7.npz
* model/
  + model\_data\_19.pkl
* out.txt
  + Contains the output of running the code
* error\_rate.png

**Additional Dependencies:**

1. abc
2. numpy

**Design:**

An object-oriented design has been followed to build the LeNet5 classifier to ensure modularity. The use of classes with access modifiers ensures functions are only called within the relevant spaces. Inheritance has been used to derive the Neural Net layers given a base class.

**Implementation:**

The LeNet5 Neural Network Classifier is implemented as follows.

**Layers:**

The layers.py file contains the required code for the layers of the neural network. This has five classes which enable us to construct the different types of layers as per each of their individual requirements. Of these, the LeNetLayer class is the base class to construct an LeNet layer from which the other classes are derived. The classes derived for the other classes are Input, Convolution, MaxPooling and FullyConnected. All the derived classes created take in initializing parameters.

**LeNetLayer:**

This is the base class to create a layer for the LeNet5 implementation. It takes in the parameters passed in from initializing the derived classes and instantiates the layer. The parameters taken in are as follows.

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Function** | **Default** |
| id | The identification tag of the layer (Eg. C1, S2) . | - |
| num\_kernels | The number of kernels contained in the layer. | 1 |
| kernel\_dims | The dimensions of each of the individual kernels. Takes in a tuple of two elements. | (0,0) |
| input\_size | The input size for each of the elements inside a mini batch. Accepts a tuple of 3 elements. | - |
| padding | The amount of zero padding to be performed. | 0 |
| stride | The stride to be taken while performing the propagation. | 1 |
| activation | The activation function to be applied to the layer | ReLU |
| load | Whether to load the kernel weights and bias from storage | False |

**\_calc\_output\_size():**

This function takes in the parameters with which the layer is initialized and calculates the dimensions of the output shape for that layer using the standard formula maintained across the different types of layers. If there are any missing values, an exception is thrown.

**\_gen\_kernels():**

This function is used to generate the initial values of the kernel weights and biases as per the kernel shape and the number of kernels. It uses the random generator from numpy to create these values.

**\_load\_path():**

This function is used to load the kernel weights and biases from the compressed files.

**print\_layer():**

This function is used to print out the configurations of the layer to understand the architecture of the neural network.

**Input:**

The input class derives from the LeNetLayer and is simply used to reshape the data provided.

**forward\_prop():**

As the initial data provided is of the format (height, width, depth), this layer converts it to the forms of (depth, height, width) for faster processing through the network.

**backward\_prop():**

This function does nothing and is provided as a dummy function to maintain consistency with the other derived functions.

**Convolution:**

The convolution class derives from the LeNetLayer and is used to perform the functions of a Convolutional Layer in a neural network.

**forward\_prop():**

During forward propagation, this function initializes an empty numpy array for the desired shape. The input data and the vectors are reshaped using helper functions and the dot product is then taken. This is reshaped again to provide the necessary shape. In addition, if an activation function is specified, the forward propagation for the same is also performed here by calling it. The output is then returned.

**backward\_prop():**

During the backward propagation, initially the derivative of the output from the previous layer passed is first transformed with respect to the activation function if required. Using helper functions, the delta is reshaped and then the product is calculated to give the deltas for the inputs, weights, and the bias. The delta for the inputs is then reshaped again to get the right shape using the helper functions. Finally, these deltas are multiplied by the learning rate to update the weights and the bias for the kernels respectively whereas the input delta is returned to the propagated further back.

**MaxPooling:**

The MaxPooling class derives from the LeNetLayer and is used to perform the functions of a MaxPooling Layer in a neural network. This layer does not require the generation of weighted kernels.

**forward\_prop():**

During forward propagation, this function reshapes the input image according to the size of the kernel and calculates the maximum values across the relevant axes. Further, if an activation function is specified, its forward propagation is called here.

**backward\_prop():**

During the backward propagation, the first step is to perform the back propagation for the activation function if present. Next the back propagation is carried out by repeating the maximum values across the size of the kernel and calculating the delta of inputs by checking for the positions where the values of the inputs match the values of the delta. The input delta is then returned.

**FullyConnected:**

The fullyconnected class derives from the LeNetLayer and is used to perform the functions of a FullyConnected Layer in a neural network.

**forward\_prop():**

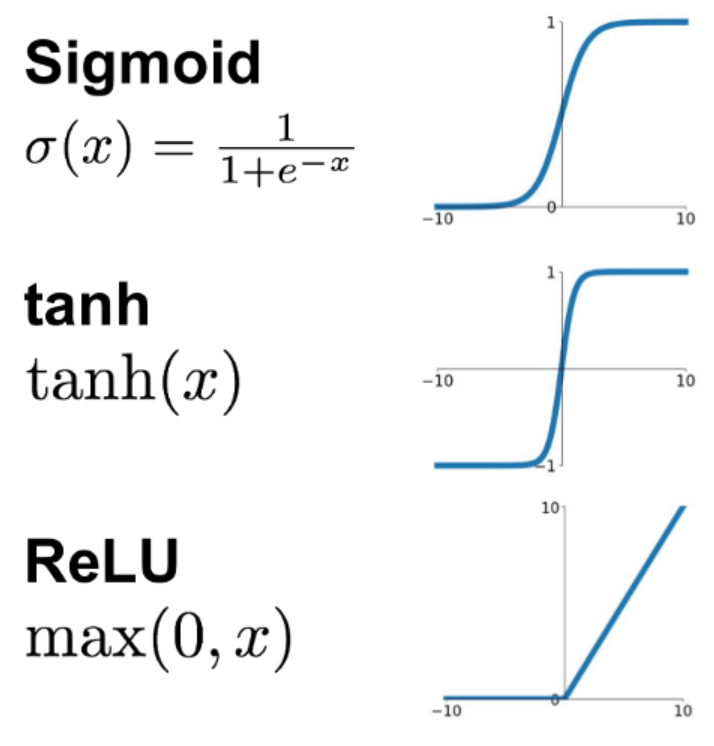
During forward propagation, this function calculated the dot product of the inputs and the kernel and then adds the values of the bias. Finally, if an activation function is present, another forward propagation for it is carried out in this step and the value is returned.

**backward\_prop():**

During the backward propagation, the activation function is dealt with first by calculating its back-propagation results if specified. Following this, the deltas for the weights, bias and inputs are calculated by calculating dot products of the delta with the inputs and weights. The deltas for the weights and the bias are multiplied with the learning rate to update the kernel parameters. The input delta is returned to be further propagated.

**Activations:**

The activations.py file contains the classes defined for the different activation functions. Each class contains a function for forward and backward propagation respectively which are called from the neural net layers.



**Model:**

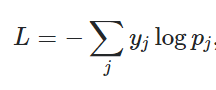
The model.py file contains the required code to build the model using the different layers. The initialization of the class takes only a parameter for the type of activation layer and consists of code to construct the model as per the specifications of the question. The model stores the layers in a dictionary with the keys being the layer identifiers and the values being the layer objects. The layers are added to the dictionary in such a way that the output dimensions for a certain layer are the input dimensions for the next.

**print\_net():**

This function is used to print out the neural network by individually printing out each of the layers to understand if the construction has been done correctly.

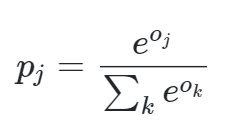
**entropy\_loss():**

The loss function is used to calculate the entropy loss after the softmax function is applied on a step of forward propagation once the class probabilities have been obtained.



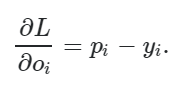
**softmax():**

Given the final output from the last layer, the softmax function converts them into probabilities for each of the 10 classes using the softmax function.



**softmax\_delta():**

Before starting the backpropagation, the softmax delta function is used to transform the output classes using the derivative of the softmax function.



**one\_hot\_y():**

This function is used to convert the categorical labels of the images into a one hot encoded numpy array for easier calculation of the entropy loss.

**save\_layers():**

This function saves the co-efficients of the convolutional and the fully connected layers, which require kernel weights.

**Forward\_Propagation():**

This function takes in parameters containing the input image, the labels for each of the corresponding images and the mode. The mode decides what is returned after the function. A **train** mode indicates that the loss is to be returned whereas the **eval** mode indicates that the predictions must be returned. The feedforward function passes through all the layers iteratively by using the output of one for the input of the next. Once it has passed through all the layers, it calls the loss function to calculate the entropy loss for the train mode. For the eval mode, the final step is to convert the probabilities into the classes having the highest probability and then return that data.

**Back\_Propagation():**

This function carries out the backpropagation through the neural network. Like the forward propagation function, this iterates through all the layers of the network although in reverse. The delta values of one layer after being calculated are passed as the delta values to the next layer. This function does not return anything as it is only responsible for updating the kernel values.

**Helper:**

The helper module contains relevant functions to assist during the forward and backward propagation of the convolutional layer. As the convolutional layer deals with the image directly as inputs, this module contains functions to convert the image to and from column vectors for fast and efficient processing. These helper functions were Python implementations of the MATLAB functions im2col and col2im that are used to vectorize images. The reference for these MATLAB transformation functions was taken from the book **Deep Learning from Scratch** **[1]**.

**im2col():**

The im2col function is called during the forward propagation phase of the convolutional layer. This function is used to convert the input image into the column vectors by stretching it to ease the calculation of the dot product.

**col2im():**

The col2im function is called during the backward propagation phase of the convolutional layer. This function is used to convert the stretched image as column vectors back into the format of the input image after the calculation of the dot product with the deltas.

**Additional Changes:**

This section talks about the changes made in the template provided for this assignment, to facilitate experimentation as well as to ease processes.

**A2.py**

* The pickled models are stored in a **model** directory instead of storing them in the same directory as this file.
* Instead of the running the entire train data (60,000) and test data (10,000) at one go, they were split into 10 iterations to reduce the memory load on the computer and provide faster calculations. This did not lead to any effects to the cross-entropy loss calculation.
* The test function was also split into 10 iterations as mentioned above to test the data with the model from the latest epoch.
* The plot of the error rate is saved as **error\_rate.png** instead of plotting as the code was run on a server and the image visualization was not possible.

**config.yaml**

* The path for storing the models was modified by adding the directory **model/** to the path.

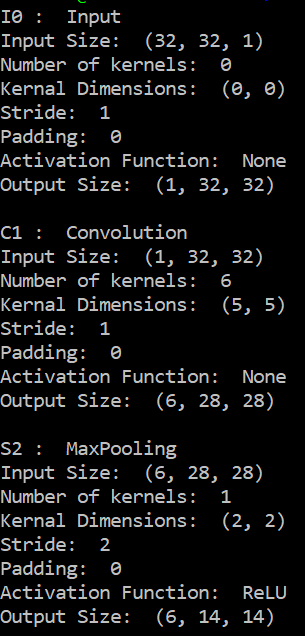
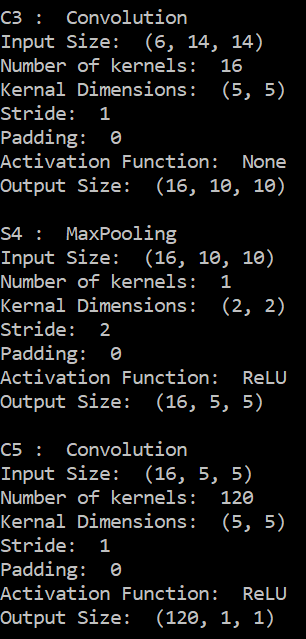
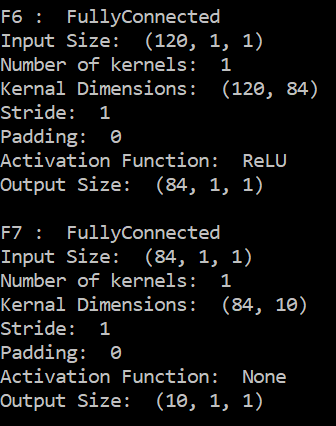
**Added Configurability:**

* The activation method within layers (ReLU/Sigmoid/tanh) is passed as an argument to the function. The default value is added in **config.yaml** as ReLU.

**Model paths:**

The model from the final epoch is in the model/ directory. The models trained from the other epochs can be found at this link: <https://drive.google.com/drive/folders/1BddJwtr08f_AtHSIXNAEQ5esav5prQo_?usp=sharing>

**Model Architecture:**

 The model architecture is summarized in the images below containing for the layers included in the model.

**Time Constraints:**

The major bottleneck in processing was with the Convolutional Layer every time. Here is a summary of the improvements to the Convolutional layer to improve the speed of the program.

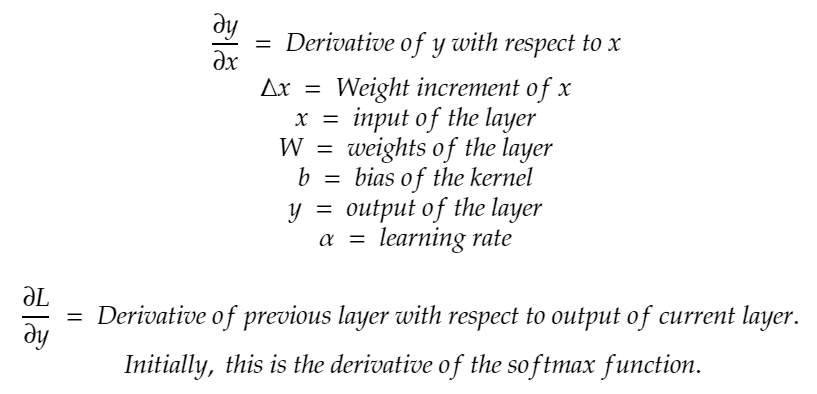
1. For loops over the image to calculate dot products. This took 5 minutes/mini-batch
2. Scipy’s signal.convolve2d. This took 1 minute/mini-batch
3. Dot product using Python version of MATLAB vectorizations. This took 1second/mini-batch

Thus the speed of every mini-batch was reduced to 1 second and the total time to process an epoch 2 minutes.

**Layer Forward and Backward Calculations and Jacobian Matrices**

The one hot encoded input labels are first passed through the softmax\_delta function, the results of which are fed through the layers sequentially in the matrix.

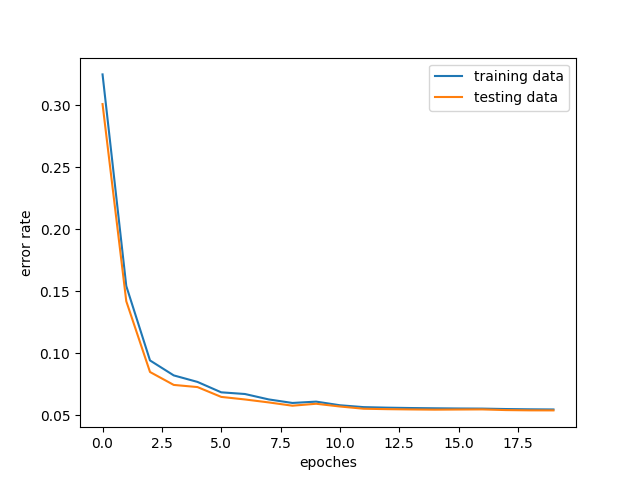
Before considering the layers, here is the terminology used in this section. Additionally, all representations shown are only in one dimension.



|  |
| --- |
| **Convolutional Layer**  **Forward Propagation:**    **Back Propagation:** |
| **MaxPooling Layer**  **Forward Propagation:**    **Back Propagation:** |
| **FullyConnected Layer**  **Forward Propagation:**    **Back Propagation:** |

|  |
| --- |
| **ReLU**  **Forward Propagation:**    **Back Propagation:** |
| **Sigmoid**  **Forward Propagation:**    **Back Propagation:** |
| **Tanh**  **Forward Propagation:**    **Back Propagation:** |

|  |
| --- |
| **Softmax**  **Forward Propagation:**    **Back Propagation:** |

**Results:**

As can be seen in the figure below, the training and the testing error quickly converged after a few epochs after the steep descents from the initial epochs. The error rate graph does not show any significant fluctuations in the error rate except for the steep decline in the beginning of the training period after which it gradually declines over time.

Moreover, we know that overfitting occurs when the machine learning model has been fit too closely regarding the training data. An overfit model can easily be identified if the testing error does not converge like the pattern of the training data. However, as we can see that the training and the testing data follow a similar error rate over the epochs, it can be said that there is no overfitting. No overfitting justifies a proper bias-variance trade-off. A relatively higher variance compared to bias occurs when the model is overfit whereas a relatively high bias compared to the variance occurs when the model is underfit. Thus, we can say that the bias-variance trade-off is appropriate as a higher bias or variance would have skewed this graph to show a higher error rate for training and a higher error rate for testing over the epochs, respectively.

|  |  |  |
| --- | --- | --- |
| Epoch | Cost | 0/1 Error (size 10000) |
| 1 | 64305.6817 | 3011.2652 |
| 2 | 9562.9608 | 1419.2508 |
| 3 | 5943.1909 | 847.8947 |
| 4 | 5239.9415 | 743.5490 |
| 5 | 4690.5766 | 726.1453 |
| 6 | 4223.3461 | 646.7868 |
| 7 | 4033.0408 | 626.1701 |
| 8 | 3884.1784 | 602.3434 |
| 9 | 3675.6519 | 575.7034 |
| 10 | 3608.0737 | 591.4806 |
| 11 | 3541.4559 | 569.3276 |
| 12 | 3467.2707 | 551.3353 |
| 13 | 3377.7280 | 547.8500 |
| 14 | 3366.6763 | 545.5894 |
| 15 | 3352.1045 | 543.8129 |
| 16 | 3339.4654 | 544.9614 |
| 17 | 3325.3440 | 545.8700 |
| 18 | 3315.7997 | 540.4173 |
| 19 | 3303.4291 | 538.1894 |
| 20 | 3290.2735 | 537.8174 |

Results from the output which can also be found in out.txt reflect numerically the information displayed in the graph. A convergence of both the cost and the testing error can be seen around 8-10 epochs where the relative change in the values becomes very small compared to the actual values themselves.

It is interesting to note that these results are not consistent across multiple runs of the same LeNet5 model. As the weights and the bias for the kernel are initialised randomly every time the model is trained, how quick the model converges differs every time. However, it was noticed, that the model reached convergence very soon across several different runs of the same model with the same parameters.

Moreover, the final values obtained for the error rate were also not consistent and varied across a range but would often remain around the value mentioned.

The conclusive results of our model are as follows:

|  |  |
| --- | --- |
| Total Time Used | 2746.2340 seconds |
| Error Rate | 0.0537 |

**Experiments:**

Further experiments were carried out to test the model with the different types of activation functions constructed. The results were compared across the 3 activation functions tested, the ReLU, the Sigmoid and the tanh, which led to interesting observations.

The ReLU activation function provides us with the best performance across all 3 activation functions with the lowest error rate and the fastest convergence. This can be attributed to the fact that the error derivatives do not saturate compared to the other two models and thus serve to improve the model at a quicker pace.

It was seen that the Sigmoid activation function had an extremely high error rate thus not being able to predict the right classes and being unfavourable to the LeNet5 framework. This can be attributed to the fact that the Sigmoid function is susceptible to the vanishing gradient problem, owing to the which the derivatives of the gradient become smaller and smaller thus not leading to any improvements in the kernel weights and bias over the iterations.

The tanh activation function performs better in this case compared to the Sigmoid as the tanh function has stronger gradient values when compared to the Sigmoid since the centring of the data is around 0, the gradients are higher. The better performance can be attributed to the fact the derivatives of the error term are stronger than those compared to the Sigmoid function.

|  |
| --- |
| **ReLU** Error rate: 0.053 |
| **Sigmoid** Error rate: 2.288 |
| **Tanh** Error rate: 0.112 |

**REFERENCES:**

1. Weidman, S. (2019). *Deep learning from scratch: building with Python from first principles*. Bejing ; Sebastopol, CA: OReilly Media, Inc.