**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\_model()
      5. entropy\_loss()
      6. Forward\_Propagation()
      7. Backward\_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. \_do\_dot()
      2. \_transf\_kernel()
      3. forward\_prop()
      4. 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()

**Additional Files:**

* Report.docx
* README.md
* model/
  + C1.npz
  + C3.npz
  + C5.npz
  + F6.npz
  + F7.npz

**Additional Dependencies:**

1. abc
2. numpy
3. scipy

**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 | None |

**\_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.

**\_do\_dot():**

This function performs the convolutions across the image and the kernel by using scipy’s convolve2d function to calculate the convolutional dot products.

**\_transf\_kernel():**

The transf\_kernel function is used to rotate the kernel twice before passing it into the convolution function to facilitate the calculation of the right values in performing the convolution.

**forward\_prop():**

During forward propagation, this function initializes an empty numpy array for the desired shape. It then iterates over the records in every mini batch, each of the kernels and the depth of the original image to calculate the dot products and finally adds the values of the bias to it. 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. Following that, empty numpy arrays are created to store the deltas calculated with respect to the inputs, the weights and the bias. 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():**

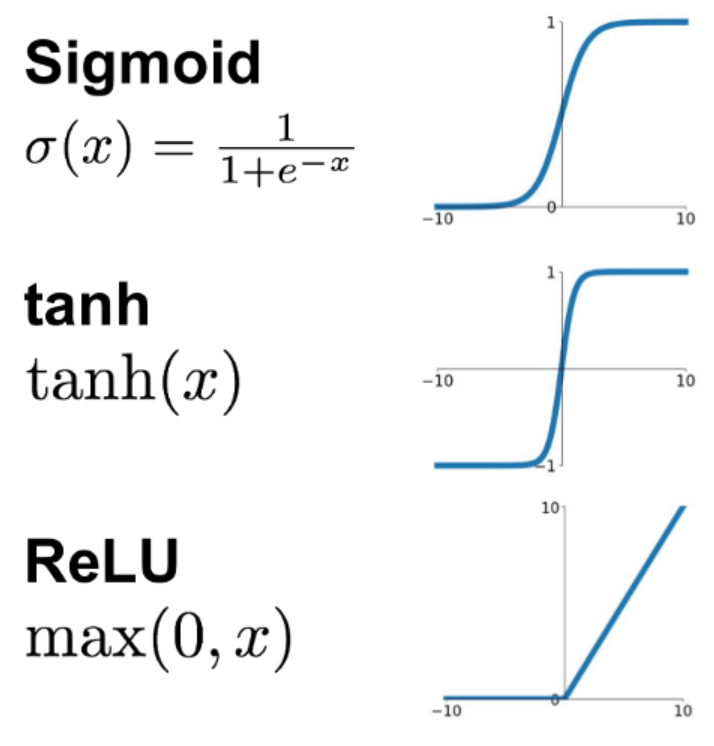
During forward propagation, this function calculated the dot product of the inputs and the kernel and then adds the values of the bias. Finally if an activation function is present, another forward propagation for it is carried out in this step and the value is returned.

**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:**

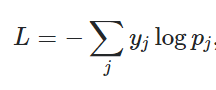
The model.py file contains the required code to build the model using the different layers. The initialization of the class does not take parameters and consists of code to construct the model as per the specifications of the question. The model stores the layers in a dictionary with the keys being the 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.

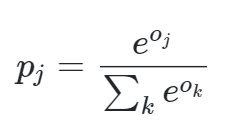
**entropy\_loss():**

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.



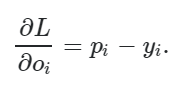
**softmax():**

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\_model():**

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

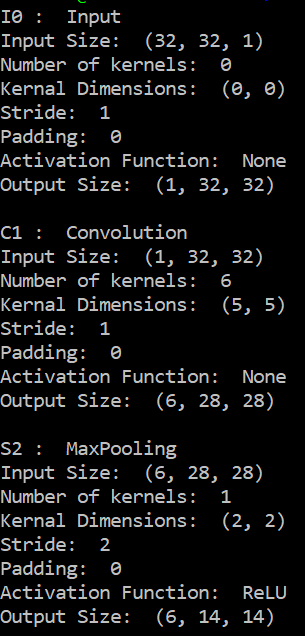
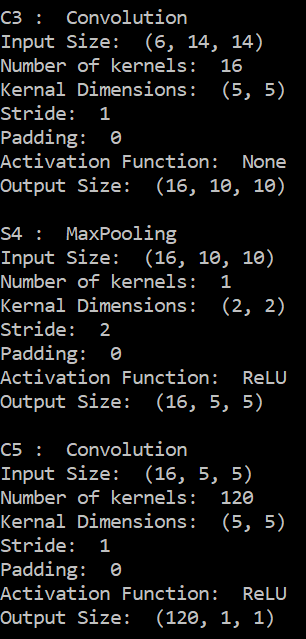
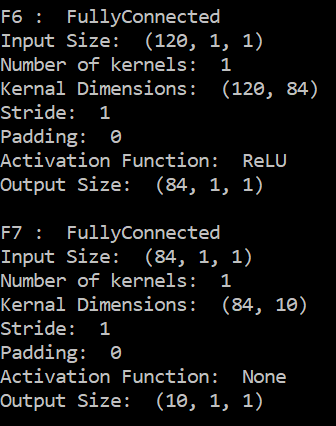
**Forward\_Propagation():**

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 returning 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.

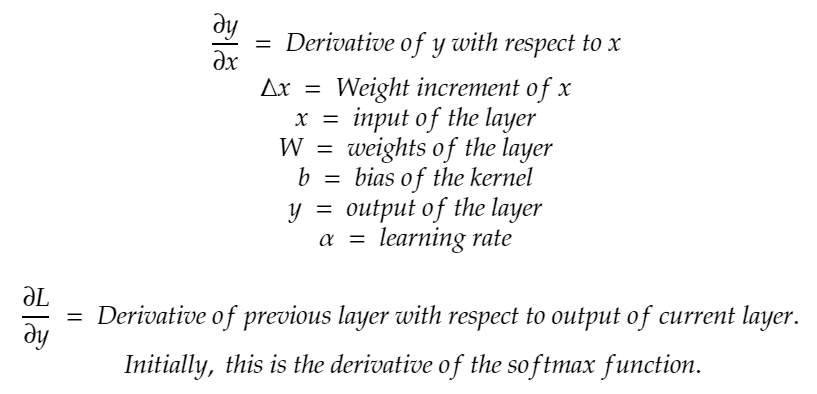
**Model Architecture:**

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

**Layer Forward and Backward Calculations**

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:** |