Deep Learning for Computer Vision (CS776A)
Indian Institute of Technology Kanpur
Assignment 1

Name: Shivam Tripathi
Email: shivamtr21@iitk.ac.in
Roll Number: 21111408
Date: February 3, 2022

Multi-Layer Perceptron for Classification on CIFAR-10

1 About the Dataset

The CIFAR-10 dataset is a popular image classification dataset [4] that contains a total of 60000 color images of 32x32 size with 3 channels. The dataset has 10 classes and each class has exactly 6000 images. The training set has 50000 images and the test set contains rest of the images.

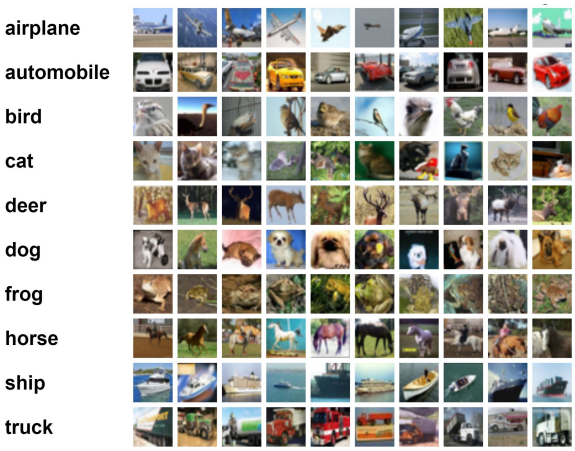


Figure 1: CIFAR-10 dataset

2 Architecture of the Multi-Layer Perceptron

A multi-layer perceptron is a neural network that has input layer followed by multiple hidden layers and ends with an output layer [3]. The output from each layer is activated using some activation function. This activation function introduces non-linearity in the model.

Each edge in the network corresponds to a weight value. These weight values are the model parameters which are updated by an optimization algorithm.

The multi-layer perceptron follows feed-forward algorithm where the output of a layer is equal to the weighted combination of last layer outputs with the current layer weights. These outputs are passed through an activation function like ReLu or Sigmoid which decided whether the neuron is activated or not.

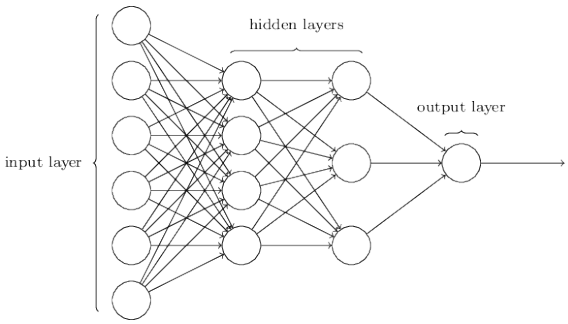


Figure 2: Example of a Multi-Layer Perceptron consisting of two hidden layers

After the feed-forward step, we calculate how bad our model has performed based on a chosen loss function. Now the next step is to update the model parameters, i.e. weights and biases, such that the loss value is reduced.

To update the parameters we calculate the gradient of loss with respect to each of these model parameters and tweak them by a little to match the actual output more closely. This process of updating the model parameters by calculating the gradients of model parameters and performing gradient descent to update the parameters is called as Backpropagation.

In order to train our multi-layer perceptron [2] we repeat the above process until convergence (or till a fixed number of iterations). In summary, we perform forward-pass, calculate the loss, and back-propagate the loss to update the weights and biases.

The multi-layer perceptron for our 10-class classification problem has the following architecture:

1. Nodes: Input layer: 512, Hidden layer: 64, Output layer: 10
2. Notations
    - X: Input data (batch-size x 512)
    - w_1: Weights between input and hidden layer (512x64)
    - b_1: Biases for hidden layer (64,)
    - o_1: Weighted outputs of the first hidden layer (512x64)
    - z_1: Activated outputs of the first hidden layer (512x64)
    - w_2: Weights between hidden and output layer (64x10)
    - b_2: Biases for output layer (10,)
    - o_2: Weighted outputs of the output layer (64x10)
    - z_2: Activated outputs of the output layer (64x10)

3 Building the Model in Python

```
class MLP():
    '''
    The MLP class initializes a single hidden-layer perceptron with the given number
        of neurons of the hidden layer. It has necessary functions to run the forward
        pass and backward pass which can be used to train the MLP model.
    '''
    def __init__(self, train_data, train_labels, n_neurons=64):
```

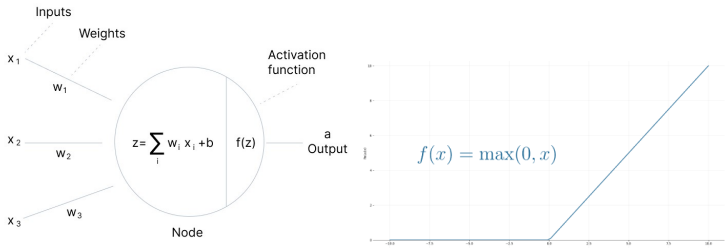
```
    # Initialize weights and biases for the network
def relu(self, x):
    '''
    The ReLU activation function returns max(0, x)
    '''
def softmax(self, x):
    '''
    This function returns the softmax probabilities.
    It performs exp() on each value and then divide each value by total of the
        exps.
    '''
def cross_entropy_loss(self, pred, truth):
    '''
    This function calculates softmax cross entropy loss.
    '''
def forwardpass(self, x):
    '''
    This function performs forward pass on the network with the given input data
    '''
def relu_gradient(self, x):
    '''
    The gradient of ReLU is 1 if the input is >=0 and 0 otherwise.
    '''
def accuracy(self, pred, truth):
    '''
    This function calculates accuracy bases on true labels and predicted labels
    '''
def backwardpass(self, x, y):
    '''
    This function performs backward pass on the network and calculates gradients
        of the network parameters i.e. weights and biases.
    '''
def train(self, n_epochs=10, batch_size=32, learning_rate=0.01):
    '''
    This function trains the model with given batch_size and learning_rate
    '''
def test(self, x, y):
    '''
    This function can be used to run the model on a test data set.
    It provides the loss and accuracy reported on the test data.
    '''
```

---

4 Activation Function and Loss Function

4.1 Activation Function

In the feed-forward process we apply some activation function in the weighted output to bring non-linearity into the picture.



(a) (b)

Figure 3: (a) Activation function is applied to the weighted output (b) Rectified-Linear Unit

This activation function decides whether the neuron should be activated or not. This helps the neural network use the information which is important and suppress the irrelevant information [1].

For our model we will use the Rectified-Linear Unit (ReLU) activation function. The ReLU function outputs same value if the input is positive, otherwise it returns zero.

$$ReLU(x) = max(0, x)$$

4.2 Loss Function

The loss function is an important aspect of our model. It tells how much your network output is close to the actual truth value. Our main task in the training is to reduce the loss value by tweaking the model parameters.

For our model we will use the cross-entropy loss which is usually good for multi-class classification problems. The equation for cross-entropy loss is:

$$L(y, \hat{y}) = -\sum_{c=1}^{C} y_{o,c} \log(\hat{y}_{o,c})$$

where $\hat{y}$ represents the predicted label, y represents the actual label, and C represents the number of classes.

5 Forward Propagation of the Input Data

The initial step in the multi-layer perceptron algorithm is forward propagation. In this step, each layer takes the input data, computes the weighted sum, applies the activation function and passes the result to the next layer in the network.

$$o_1 = X.w_1 + b_1 \tag{1}$$
$$z_1 = ReLU(o_1) \tag{2}$$
$$o_2 = z_1.w_2 + b_2 \tag{3}$$
$$z_2 = SoftMax(o_2) \tag{4}$$

In the assignment, the following code is used to perform the forward pass:

6 Backward Propagation of the Error

After we perform the forward-pass, we calculate the loss using our loss function. Since, we have to reduce the value of loss, we need to update the model parameters so that the output of the network more closely matches the truth values of the data.

For this step we will calculate the gradients of the network parameters, i.e. weights and biases. The following notations are used:

6.1 Derivation of Gradients

The gradient of softmax activation can be derived as follows [6]:

$$\frac{dL}{do_2^i} = \frac{d}{do_2^i}\Big[-\sum_{c=1}^{C} y_c \log(\hat{y}_c)\Big] = -\sum_{c=1}^{C} y_c \frac{d\big(log(\hat{y}_c)\big)}{d\hat{y}_c}.\frac{d\hat{y}_c}{do_2^i} \tag{5}$$

$$= -\sum_{c=1}^{C} \frac{y_c}{\hat{y}_c}.\frac{d\hat{y}_c}{do_2^i} = -\Big[\frac{y_i}{\hat{y}_i}.\frac{d\hat{y}_i}{do_2^i} + \sum_{c=1,c\neq i}^{C} \frac{y_c}{\hat{y}_c}\frac{d\hat{y}_c}{do_2^i}\Big] \tag{6}$$

$$= -\frac{y_i}{\hat{y}_i}.\hat{y}_i(1-\hat{y}_i) - \sum_{c=1,c\neq i}^{C} \frac{y_c}{\hat{y}_c}.(\hat{y}_c\hat{y}_i), \text{ since } \frac{d\hat{y}_i}{do_2^i} = \hat{y}_i(1-\hat{y}_i) \tag{7}$$

$$= -y_i + y_i\hat{y}_i + \sum_{c=1,c\neq i}^{c} y_c\hat{y}_i = \hat{y}_i\big(y_i + \sum_{c=1,c\neq i}^{C} y_c\big) - \hat{y}_i \tag{8}$$

$$= \hat{y}_i.1 - y_i, \text{ since } \sum_{c=1}^{C} y_c = 1 \tag{9}$$

$$= \hat{y}_i - y_i \tag{10}$$

The gradient of ReLU function is 1 if the input is positive else zero. Now, we will use these results to calculate the gradients of weights and biases.

The gradient of loss with respect to the weights and biases of the network layers can be derived as follows:

$$\frac{dL}{dw_{2\,(64,10)}} = \frac{dL}{do_2}\frac{do_2}{dW_2} = z_1^T.(z_2 - y) \tag{11}$$

$$\frac{dL}{db_{2\,(batch-size,10)}} = \frac{dL}{do_2}\frac{do_2}{db_2} = (z_2 - y) \tag{12}$$

$$\frac{dL}{dw_{1\,(512,64)}} = \frac{dL}{do_2}\frac{do_2}{dz_1}\frac{dz_1}{do_1}\frac{do_1}{dw_1} = X^T.(z_2 - y).w_2^T * ReLUgrad(z_1) \tag{13}$$

$$\frac{dL}{db_{1\,(batch-size,64)}} = \frac{dL}{do_2}\frac{do_2}{dz_1}\frac{dz_1}{do_1}\frac{do_1}{db_1} = (z_2 - y).w_2^T * ReLUgrad(z_1) \tag{14}$$

NOTE: Please ignore the order of derivatives written in the above equations. They have been written in the order of their matching shapes. For example: $\frac{dL}{do_2}\frac{do_2}{dW_2} = (z_2 - y)z1$ but we have written $z_1^T.(z_2 - y)$ so that their shapes match when you write the code in python.

The model parameters, i.e. weights and biases are updated using the gradient descent method. Its update equations are as follows:

$$w_i = w_i - \eta * dw_i$$
$$b_i = b_i - \eta * db_i$$

where η is the learning rate which impacts the size of step taken in the opposite direction of gradient.

7 Training and Testing Set-Up

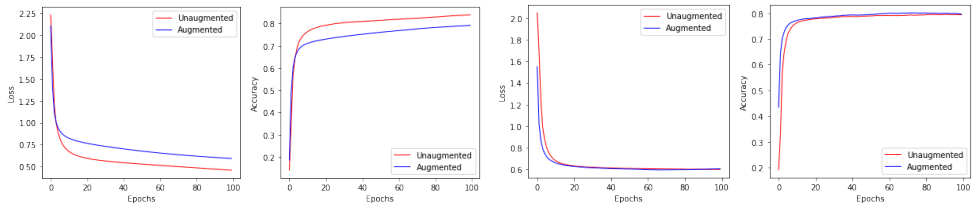
The model is trained with following settings for both augmented and un-augmented data:

- Initial learning rate (η) = 0.01 with a decay of 10% after every 10 epochs.
- Batch Size: 64
- Number of Epochs: 100
- Initalization of weights using numpy.random.rand() which initializes values from a standard normal distribution and biases are initialized with zeros
- Loss function: Categorical cross-entropy
- Accuracy: Match the one-hot encoded labels of prediction with the truth values, i.e number of labels correctly identified as either truly positive or truly negative out of the total number of items

8 Results and Conclusion

We trained our model with above settings and reported the following loss and accuracy values:

	Training Loss	**Training Accuracy**	**Testing Loss**	**Testing Accuracy**
Original data	0.46	83.78%	0.59	79.76%
Augmented data	0.59	79.04%	0.59	80.05%



(a) (b)

Figure 4: (a) Training loss and accuracy on original (un-augmented) vs augmented data (b) Test loss and accuracy on original (un-augmented) vs augmented data

The model trained on un-augmented data overfits easily and we get higher training accuracy in comparison to the accuracy on the test data. The augmented data contains original images along with their transformed counterpart. With the additional variations in the images, the model is forced to learn robust features that generalize better on the test data [5]. On our experiment, we have reported slightly better accuracy on the test data by using the model trained on augmented data.

References

[1] 12 Types of Neural Network Activation Functions: How to Choose? https://www.v7labs.com/blog/neural-networks-activation-functions.

[2] Building a Neural Network with a Single Hidden Layer using Numpy. https://towardsdatascience.com/building-a-neural-network-with-a-single-hidden-layer-using-numpy-

[3] Multilayer Perceptron Explained with a Real-Life Example and Python Code: Sentiment Analysis. https://towardsdatascience.com/multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-analy

[4] The CIFAR-10 dataset. https://www.cs.toronto.edu/~kriz/cifar.html.

[5] Training with Image Data Augmentation in Keras. https://stepup.ai/train_data_augmentation_keras/.

[6] Understanding and implementing Neural Network with SoftMax in Python from scratch. https://www.adeveloperdiary.com/data-science/deep-learning/neural-network-with-softmax-in-python/.