Deep Learning Programming Assignment 1

Rushil Ashish Shah

B20AI036

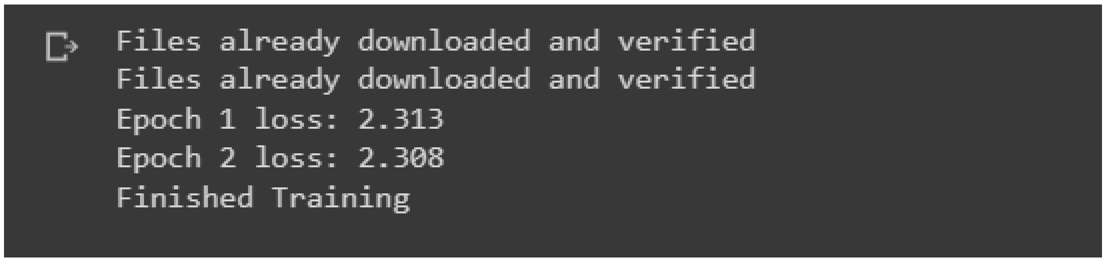
31st January 2023

Question 1

Part 1

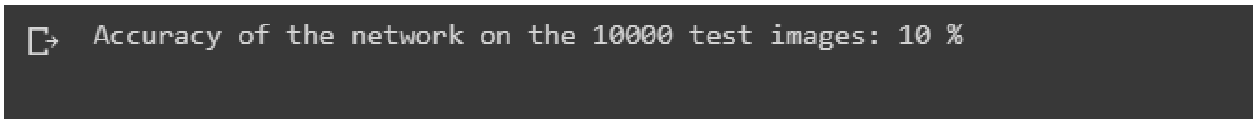
The CIFAR - 10 Dataset is an image dataset that we have loaded using the torchvision library and created the trainloader and testloader of the same

After that I have written the code for implementing a Convolutional neural network on the dataset which consists of 3 Fully-connected linear layers. The activation function used in this case is **sigmoid**. The loss function chosen is crossEntropy Loss and the SGD optimizer. Learning rate is 0.01. I have trained the model for a couple of epochs and have reported the loss.



```
Files already downloaded and verified
Files already downloaded and verified
Epoch 1 loss: 2.313
Epoch 2 loss: 2.308
Finished Training
```

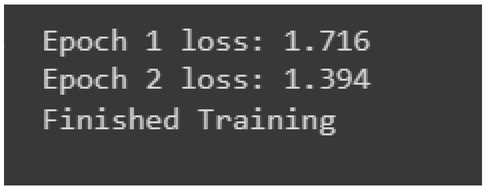
After this I predicted my model for the test data and calculated the accuracy.



```
Accuracy of the network on the 10000 test images: 10 %
```

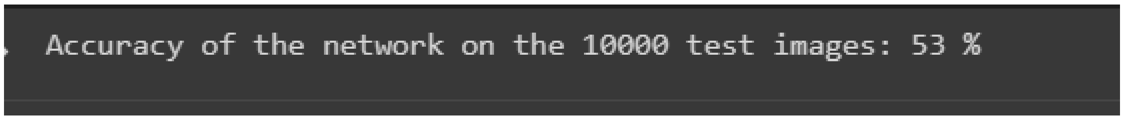
After this I have performed the same process using **tanh** as the activation function and I have listed my results below.

Training:



```
Epoch 1 loss: 1.716
Epoch 2 loss: 1.394
Finished Training
```

Testing:



```
Accuracy of the network on the 10000 test images: 53 %
```

As we can see the accuracy on sigmoid is quite low. This is mainly due to the following factors in my opinion. One is that since the sigmoid function has the range -inf to inf whereas the tanh ranges from -1 to 1 and hence the mean of tanh would always be closer to 0. The other reason might be that since we are training only for 2 epochs this time is not enough for sigmoid function to converge and hence the lower accuracy. If we had trained for more epochs the accuracy difference between sigmoid and tanh would not have been so high.

Part 2

Now I have gradually increased the number of Fully connected Layers and have reported the accuracy of all of them layer-wise. We can clearly see that when the number of FCs increases from 5 to 6 there is a sudden drop in accuracy which clearly indicates vanishing gradient. The same is also checked by calculating the values of gradient at all positions during backward propagation.

Number Of Layers	Accuracy	Epoch 2 Training Loss
3	52%	1.411
4	55%	1.391
5	57%	1.107
6	34%	1.771

Part 3

Few ways to overcome vanishing gradient are:

- Use proper initialization of weights: By initializing the weights of the network with a specific distribution, it can help prevent the vanishing gradient problem
- Use batch normalization: By normalizing the activations of a layer, it can reduce the internal covariate shift and prevent the vanishing gradient problem.
- Use LeakyReLU and ReLU activation function
- Use skip connections: Skip Connections allow layers to skip layers and connect to layers further up the network, allowing for information to flow more easily up the network.

- Gradient Clipping: By setting a threshold on the gradients, it prevents them from becoming too large and causing the vanishing gradient problem.

Accuracy Changes after implementation:

● Skip Connection and Batch Normalization:	34 to 39%
● LeakyReLU act Function:	34 to 46%
● Initialization of weights:	34 to 43%

To conclude we can finally say that one should be very careful while incrementing the number of Fully connected layers so that they do not lose much information. We can see that even after using all the possible methods for overcoming vanishing gradient it is difficult to achieve the previous accuracy of 57% which we had achieved when the number of fully connected layers was 5.

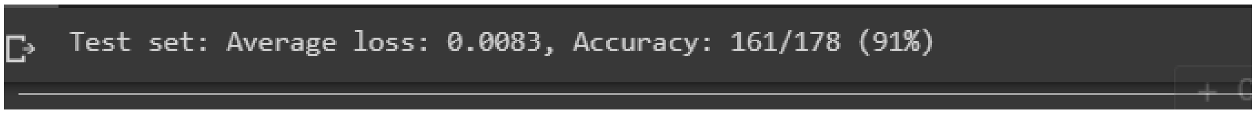
Question 2

We have imported the dataset from google drive and created the train and validation loader of the same and have also applied the transformation. After that I have done some visualization for context.

After this I have written the code for a convolutional neural network consisting of 2 FCs. Then I have implemented the L1 regularization function. The basic utility of the regularization process is to calibrate the model. Instead of simply aiming to minimize loss, we'll now minimize loss+complexity, which is called structural risk minimization:

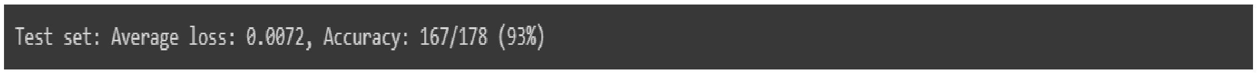
$$\text{argmin}_{w,b} \sum_{i=1}^{n} loss(yy') + \lambda \; regularizer(w,b)$$

The **L1** function basically adds the penalty i.e., lambda times the absolute value of weight coefficient to the loss. I have calculated the loss and accuracy for the model.



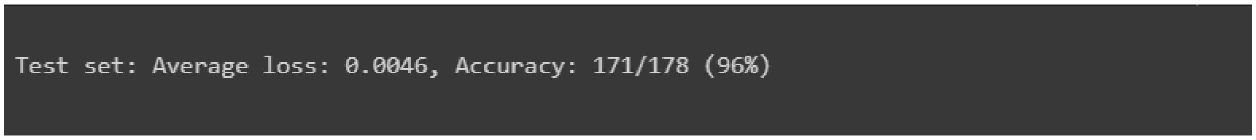
```
Test set: Average loss: 0.0083, Accuracy: 161/178 (91%)
```

After doing this I changed the regularization function to **L2** which adds the penalty Lambda times the square of weight coefficient to the loss. I have calculated the loss and accuracy for the model.



```
Test set: Average loss: 0.0072, Accuracy: 167/178 (93%)
```

Another form of regularization I have implemented after this is the **dropout** function. The dropout function is basically an ensemble machine learning method which uses various types of models to build the final model. In Every epoch it randomly drops nodes. The end result is a collection of all the models which basically covers all the nodes rather than covering them multiple times at each iteration.



```
Test set: Average loss: 0.0046, Accuracy: 171/178 (96%)
```

	Average Loss on test set	Accuracy
L1 Regularization	0.0083	91%
L2 Regularization	0.0072	93%
Dropout Regularization	0.0046	96%

Gradient checking in Python can be performed by comparing the analytically computed gradients (via backpropagation) with the numerical approximations of the gradients. The steps to perform gradient checking in Python are:

1. Compute the analytical gradients of the model's parameters using backpropagation.
2. For each parameter, initialize a small perturbation and compute the forward pass with the perturbed parameter.
3. Compute the loss for the perturbed forward pass.
4. Compute the numerical approximation of the gradient using the formula: numerical_gradient = (loss_plus - loss_minus) / (2 * epsilon) where loss_plus and loss_minus are the losses computed for the forward pass with the perturbed parameter in the positive and negative direction respectively, and epsilon is the magnitude of the perturbation.
5. Compare the numerical approximation with the analytical gradient to calculate the relative error: rel_error = abs(numerical_gradient - analytical_gradient) / (abs(numerical_gradient) + abs(analytical_gradient))
6. Repeat steps 2 to 5 for all parameters in the model and compare the relative errors with a threshold value to determine if the gradients are correct. If the relative error is larger than

the threshold, it may indicate a problem with the implementation of the model or the optimization process.

It is important to note that gradient checking is a computationally expensive process and should only be used for debugging purposes. In practice, it is usually sufficient to check the gradients during the development phase and then turn off gradient checking for the final version of the model.