ML (CS60050) Project 2 Report

Project Code: WSNN

Group: 13

Group members:

Anuj Parashar - 22CS60R82

Raj Shekhar Vaghela - 22CS60R32

Shaswata Dutta - 19EC39034

In this project, we built two neural network models from scratch in order to predict the classes of different varieties of wheat. Our dataset has 210 samples in total, with 7 attributes and 3 different classes (labeled '1', '2' and '3').

We used simple **numpy** functions to extract the data from the text file *seeds_dataset.txt*, and also performed the 80:20 train-test splitting.

We created a class **Layer** with the following attributes:

- **weights:** Contains the weights from the previous layer to this layer. If the previous layer has n1 neurons and the current layer has n2 neurons, then weights is a numpy ndarray of shape (n1, n2)
- **biases:** Contains the biases from the previous layer to this layer. This is a numpy array of shape (1, n2), where n2 is the number of neurons in the current (i.e., this) layer
- **output_hidden:** This is the final activation output from the current hidden layer, and is of the form $\sigma(W^T X + b)$, where X is the input (from the previous layer), and W and b are the weights and biases from previous layer to current layer respectively (their shapes are described as above).
- **final_output:** This is the linear output from the current layer, and is of the form $W^T X + b$. The parameters in this form are the same as described in *output_hidden*.

The following functions are also used:

- **softmax()**: Returns a numpy ndarray whose entries are the softmax values of the corresponding indices of the input. If the input is $y = [y_1,\ y_2, \dots y_n]$, then the output is $y\_out = [f(y_1),\ f(y_2),\ \dots f(y_n)]$, where $f(x) = \frac{e^x}{\sum_j e^j}$ represents the mathematical *softmax* function.
- **sigmoid_deriv()**: Returns a numpy ndarray which contains the derivative of sigmoid function applied upon each of the elements of the input numpy array.
- **relu()**: Returns a numpy ndarray which contains the value of relu function applied upon each of the elements of the input numpy array.

For performing the weight updates, the following functions are used:

- **forward():** It takes parameters batch_data (set of training samples for the current input batch) and y_train (training labels of the current input batch) as input. The batch_data is passed through the hidden layers. The list last_output_softmax is computed from the W^TX+b output of the last hidden layer. Also, the one-hot encoded train labels are saved into the *actual* list.
- **backward_final():** It takes parameters X (set of samples for the current input batch), y (actual labels of the current input batch), y_pred (predicted labels for the current input batch), no_of_hidden_layer (total number of hidden layers within the model), y_train (to compute accuracy after the backward propagation), t (to check whether its training phase or test phase; t = 1 for training phase, and t = 0 for test phase).

  The initial function will calculate the derivative of loss and the softmax function can be done cumulative by using (y_pred – y). Now we need to multiply the output of the inner layer to this gradient to find **total loss**. After finding **total loss**, we will back propagate the error to update weights between inner layer and outer layer. Now the same error will be back propagated to update the weights of the inner layer, so that the derivative of the sigmoid function will be used. Finally we will use learning rate i.e. 0.01 (given) to update weights of layers.

There are important functions like:

- **accuracy()**: Computes the accuracy of the current model based on the output of the softmax functions and the one-hot-encoded training label. If the maximum among the softmax values corresponds to the index of '1' within the one-hot-encoded label, its correctly predicted, otherwise it's incorrectly predicted. Correspondingly we maintain a counter to check the total accuracy.
- **loss():** This function computes the **categorical cross entropy loss** based on the computed softmax values and the already provided labels, and returns this computed loss.

We also computed the training and test accuracy after every 10 epochs using **scikit-learn**. The implementation is clearly mentioned in the attached *ipynb* notebook.

Accuracy values (after 200 epochs) obtained:

- For ANN specification 1 (using neural network built from scratch):
  Training accuracy: 87.5%
  Test accuracy: 87.30%

- For ANN specification 2 (using neural network built from scratch) (the accuracies are in %):
  Training accuracy: 90.10%
  Test accuracy: 88.49%

- For ANN specification 1 (using MLP implementation of sklearn):
  Training accuracy: 92.86%
  Test accuracy: 97.62%

- For ANN specification 2 (using MLP implementation of sklearn):
  Training accuracy: 92.26%
  Test accuracy: 95.24%

The corresponding training and test accuracy plots (with accuracies after every 10 epochs) are mentioned in the *ipynb* notebook itself.

Observations:

- Clearly, the **MLP implementation of sklearn is superior compared to that of our built artificial neural networks**, with respect to the training and test set accuracies.
- In some instances, the **test set accuracy may be higher compared to the training set accuracy**. This occurs when the entire notebook is run again and again, thus the training set and test set are changed, the corresponding mini-batches are changed and hence the **accuracy varies whenever the entire notebook is re-run**. However, the same dataset is used for ANN 1, ANN 2 and the sklearn implementations of the two given specifications. Also, on an average, the accuracy for
  - ANN 1 (built from scratch) lies in the range of 85-90% (for both train and test sets)
  - ANN 2 (built from scratch) lies in the range of 88-94% (for both train and test sets)
  - ANN 1 (using MLP implementation of sklearn) lies in the range of 90-97% (for both train and test sets)
  - ANN 2 (using MLP implementation of sklearn) lies in the range of 93-96% (for both train and test sets)
- Clearly, the **accuracy of the built-from-scratch model for ANN2 is higher compared to that for ANN1** because the former has **2 hidden layers** with **added non-linearity**, which results not only in better training accuracy, but also allows for **better generalization** yielding higher test set accuracy compared to that of ANN1 (built from scratch).

Note: There is no such separately-constructed module for *dataloader*, *training* and *predict*. All these modules, there related concepts and functionalities have been implemented in appropriate code cells within the *ipynb* notebook.