Assignment 2

190029087

20 Dec, 2019

1 Program Overview

The submission intends to implement all three levels of the learning agent in Python through **minimal use of pre-defined library functions**[1]. The program can be ran with command:

```
$ python3 A4main.py [Bas/Int/Adv] [1/2]
```

where "[1/2]" is an optional argument applicable only to the advanced agent. Additionally, a screenshot of an instance of the program run and a text file listing requirements have been included for guidance. The PEAS model for the problem can be defined as:

1. Performance: Can be measured with evaluation metrics such as validation error, accuracy, precision, recall, f-score, etc.
2. Environment: The data set that the model gets to see during training/inference time.
3. Actuator: These are the weight and bias updates that the model makes during training time in order to minimize the overall loss.
4. Sensor: Could be the decrease in training loss with each epoch due to mis-classified instances that helps the model perceive how far is it from reaching the global minimum.

2 Design

2.1 Encoding scheme

The data set for tickets issuing contains nine boolean features deciding which response team will handle the tickets. The five different values of tickets - i.e. Emergencies, Networking, Credentials, Datawarehouse, Equipment make it a

[1]Instead of using scikit-learn, I have implemented the neural network from scratch using numpy arrays.

five-class classification problem. As a first step of encoding, all the features as well as the classes are label-encoded so that boolean features - Yes/No are labelled as 1/0 respectively while the five classes are labelled as shown in figure 2.2.

Since label encoding of classes might make the network assume an inherent order among these (e.g. Emergencies + 2 = Networking + 1 = Datawarehouse) as the training progresses, an additional one-hot encoding scheme is performed over these to avoid this (table 2.2). The resulting data is saved in the file *training_table.csv* for future references (see section 2.6).

Classes and labels		
Class	Label-encoding	One-hot encoding
Credentials	0	[1, 0, 0, 0, 0]
Datawarehouse	1	[0, 1, 0, 0, 0]
Emergencies	2	[0, 0, 1, 0, 0]
Equipment	3	[0, 0, 0, 1, 0]
Networking	4	[0, 0, 0, 0, 1]

2.2 Neural Network

As briefed in figure 1, the design for the three-layered neural network (input $\longrightarrow$ hidden layer $\longrightarrow$ output) contains two sets of weights and biases. The first set, $w1$ and $b1$ denote the weights for connections of inputs to hidden layers, i.e. $w1_{1,1}, w1_{1,2}, w1_{1_3}$ and the biases for the neurons in hidden layer while the second set of weights $w2$ adhere to connections of hidden layers neurons to output layers, i.e. $w2_{2,1}, w2_{2,2}, w2_{2,3}, w2_{2,4}, and w2_{2,5}$ and that of biases adhere to the five neurons in the output layer. Thus, all weights and biases can be thought of as arrays with shapes as mentioned in table 2.2. Each element of the weights array corresponds to a connection from neuron i of layer *l-1* to a neuron j of layer l while each element of the bias array belongs to the neuron j of layer l. All the weights undergo random initialization while the biases are all zeros at the beginning.

In order to avoid overfitting, the data set is split into training and test sets in 80:20 proportion, i.e. the first 80% is chosen for training and the remaining for testing. The size of input layer corresponds to the number of instances in the train set while that of the output layer is five, i.e. the number of classes in the dataset.

Dimensions for weights and biases	
Parameter	Shape
w_1	(num_of_inputs, hidden_layer_size)
b_1	(hidden_layer_size, 1)
w_2	(hidden_layer_size, num_of_output_classes)
b_2	(num_of_output_classes, 1)

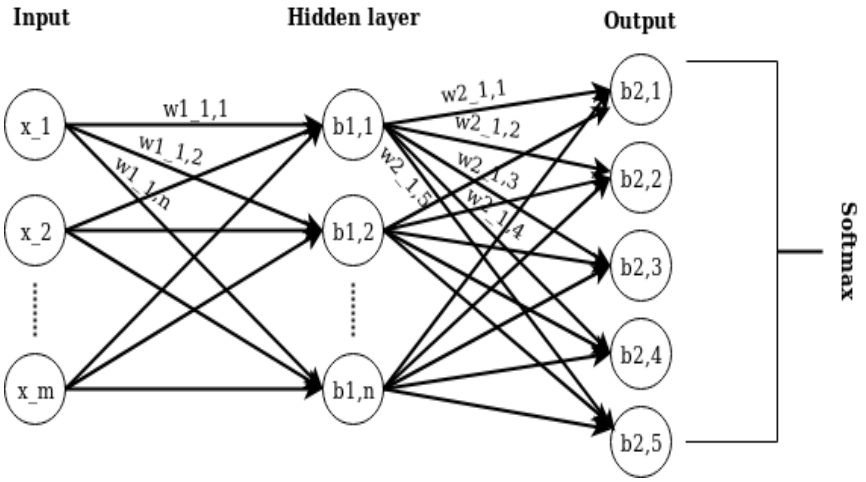

Figure 1: Figure showing the neural net architecture for the problem.

2.3 Hyperparameters

After repeated training using 28, 56, 128, 256 and 512 neurons, a hidden layer size of 128 was found to be optimal for achieving the minimum training error. Additionally, the training with backpropagation uses a learning rate of 0.5 which decides the size of steps for updating the weights and biases. In addition a momentum rate is used as a hyperparameter to add the previous gradient sum directions during the training for a faster convergence. The value for momentum was chosen to be 0.8 after several runs based on observing the minimum training error.

2.4 Activation functions

All the neurons in the input and the hidden layer use the sigmoid activation function. This in turn guarantees that the output of each neuron remains in the range [-1,1]. For the output layer, a softmax activation ensures that the output of all neurons sum to 1. This simplifies the process of choosing most likely output since the outputs now form a sample space denoting a probability distribution. For inference on new data, the final weights and biases obtained after training is used.

2.5 Termination condition for training

Training continues until following two stopping conditions have been met:

(a) the minimum validation error has reached below 0.1, and

(b) the validation error has not decreased since last 5 epochs.

2.6 Text-based interface

The user interface of the program lets users input their own test cases by asking the values for each feature. After having a complete feature set, the program uses the previous label encoding (section 2.1) to convert these into 1/0 and then carries out a forward pass using these as inputs to the neural network. The output with the highest probability can then be decoded to get the corresponding class name. If the output is correct, the program is ready to accept another query from the user.

Early prediction On feeding each input feature, the UI asks the user if (s)he is tired of supplying inputs. If yes, the program skips taking further inputs, computes the average of remaining input features (from the given training table) and fills up the remaining features with these using a threshold of 0.5 (e.g., if average value for the column 'Login' = 0.34, value for 'Login' = 0).

Output correction Additionally, if the user thinks that the output to their inputs is wrong, they can manually feed the correct choice in which case the train data is updated in two ways: (i) if there exist one or more entries with matching features, the value for their label(s) is simply updated, and (ii) if there is no previous entry for the given combination of features, a new entry is appended to the train set, the network is re-trained and the new weights and biases are stored. After training, the program is again ready to accept new inputs from the user.

3 Implementation

3.1 Packages Required:

Pandas: For csv-file related operations: read/write and data manipulation.

Scikit-learn: For train-validation split.

Numpy: For matrix-related operations.

Pickle: For saving the states of neural nets (weights and biases) as well as input-label mappings for future use.

Table 3.1 shows the validation error obtained with different sizes of the hidden layer. As the error is least for a size of 128 units, this was chosen for further experiments.

All the weights and biases are numpy arrays that are initialized randomly. The hyper-parameters along with their values for the basic agent are as described in table 3.1.

Epochs and validation error for training		
Hidden layer size	No. of epochs	Final validation error
32	996	0.061
64	954	0.067
128	804	0.058
256	283	0.071
512	1352	0.200

Hyperparameter and values	
Parameter	Shape
hidden layer size	128
learning rate	0.5
momentum	0.8
batch size	32
early stopping	if validation error does not decrease for 5 epochs
loss function	cross-entropy

3.2 Level-wise operations

For all three levels of the agent, the program first reads the given data, performs necessary encodings and stores these to the file **'training_table.csv'**. Additionally, each level performs its own specific operations:

Basic: Invoking the basic agent trains the neural network on the above processed data set and saves its configurations.

Intermediate: The intermediate agent invokes the text-based user interface, gathers necessary inputs from the user and returns the predictions as discussed in section 2.6.

Advanced: In addition to the common steps for all three levels, the advanced agent performs the two implementations detailed in section 6.

4 Evaluation

Running the basic model on the test set gives an error on maximum likelihood estimation of 0.669. However, the dataset provided does not contain the same number of instances for all five classes, using only standard maximum likelihood error might not give us an insight into how effective the model is at predicting individual classes. Therefore, the precision, recall and f-scores for each class are used as additional evaluation metrics (Table 4). The classes 1 to 5 are in the order mentioned in table 2.2. We can now examine that the model generalizes the best for instances belonging to "Networking" class while the precision degrades the most for test instances belonging to "Equipment" class.

Overall, the F-score (i.e., harmonic mean of precision and recall) is least for "Credentials" and hence, the instances belonging to this will suffer from more misclassification errors.

Metrics	Class 1	Class 2	Class 3	Class 4	Class 5
Precision	0.9	1	0.92	0.89	1
Recall	0.9	0.9	0.92	1	1
F-score	0.9	0.95	0.92	0.94	1

5 Testing

A range of test cases were fed to the intermediate agent using UI to observe the outputs (see Table 5). However, the predictions get interesting when the user manually corrects the output. The correction of one combination of input features can be seen to generate a chain effect which eventually effects the weights and biases of the network for a range of other entries.

Predictions	
User inputs	Prediction
{Yes, No, Yes, No, Yes, No, Yes, No, No}	Networking
{No, Yes, Yes, Yes, Yes, Yes, No, No, Yes}	Emergencies
{No, Yes, Yes, No, No, No, Yes, Yes, No}	Datawarehouse

For instance, the initial prediction for the input sets: (a) {No, Yes, Yes, Yes, Yes, No, No, No, No} and (b) {No, Yes, Yes, Yes, Yes, Yes, No, No, Yes } is *"Emergencies"*, which is inconsistent to the train data. Now, if an unhappy user corrects the prediction for (a) to "Equipment" and retrains the network, then the updated network outputs the class "Equipment" for (b) as well, although the entry for (b) is still labelled with "Emergencies" in the train set. However, this does not work the other way round, i.e., correcting the label of (b) to "Equipment" has no effect on the updated network's output of (a). Such behaviour of the network can be backed by the number of input entries in the training set that resemble to (a) and (b). Since there are more entries matching the exact configurations of (a), i.e., 18 than those of (b), i.e., 14, the weights of network will clearly be biased more towards reflecting the input-output mapping of (a). Thus, changing the label of (a) also makes the network think that (b) now has an updated label.

6 Advanced Agents

1. **Predicting waiting time**: The agent can be run as:

```
$ python3 A4main.py Adv 1
```

Assigning days - The file ***train_data_for_days.csv*** is generated on the go and contains the original train data with an additional column for the number of days. For ease of complexity, the number of days is a linear function of the number of features that have been assigned a 'Yes' value. Mathematically,

$$Days = 3 * sum(X_i) + 2 \tag{1}$$

where sum(X_i) is the sum total of 1s (assuming 'Yes' = 1 and 'No' = 0) in the i^{th} instance of the feature set X.

Configuring Network - Considering the prediction of days as a regression problem, an identical neural network is constructed (code in *daysOutputNeuralNet.py*) which uses the mean squared error function instead of maximum likelihood error (as in former classification task). Instead of 5 classes, the network now learns to predict a single continuous value so that the output shape is now *(n X 1)*, where n is the number of test instances. Additionally, some hyper-parameters have been adjusted through several runs - e.g. early stopping patience = 500. Also, the activation function of the output layer is now ReLU instead of softmax.

Evaluation: Two evaluation metrics used are mean squared error and scikit-learn's explained variance score[2]. Evaluating on the test data gives a mean squared error of 0.849 and an explained variance score of 0.948. Taking into account that a perfect variance score is 1.0, we can say that the model is generalizing well on the test set.

Testing: Result of the advanced program on some test cases fed through the UI are given in table 1. The number of days predicted align with that of the output of applying the function mentioned in equation 1.

Prediction comparison of agents		
User inputs	Basic agent output	Days output
{No, No, Yes, No, Yes, No, No, No, No}	Networking	7
{No, Yes, Yes, No, No, No, Yes, Yes, No}	Datawarehouse	14
{Yes, Yes, No, Yes, Yes, Yes, No, No, Yes}	Credentials	20

[2]https://scikit-learn.org/stable/modules/generated/sklearn.metrics.explained_variance_score.html

2. **Using different training algorithms**: In addition to training the network on the entire batch, the submission implements a variant of gradient descent, *viz.* **mini-batch gradient descent** (MBGD) for training the network. A brief description of the process is given in algorithm 1. The code for second advanced agent can be executed with:

```
$ python3 A4main.py Adv 2
```

Two minor differences in hyper-parameters are: (a) the early stopping patience is now increased to 25 epochs, and (b) the terminating condition is now when the validation error reaches below 0.8 and has not improved since last 25 epochs.

Also, a **support vector machine** (SVM) based classifier is used to verify the results further during testing. The classifier uses default hyper-parameters of scikit-learn library. It essentially works by training a maximal margin classifier to find a hyperplane that can separated the five classes by maintaining maximum possible distance from each class on a hypothetical five-dimensional space.

Algorithm 1 Mini-batch gradient descent

procedure TRAIN OVER MINI BATCHES
For epoch in n_epochs:
    $listOfMiniBatches = getBatchesFromData(Data)$
    $while(nextBatch : listOfMiniBatches)\{$
    forwardPropagate()
    backpropagate using gradient descent to update weights
    }(do Until *nextBatch*)
    $val\_error \leftarrow$ compute error on validation set
    if $val\_error < min\_val\_error$ **then**
        $min\_val\_error = val\_error$
        $save\_network\_configuration()$
    $check\_for\_early\_stopping\_condition()$

Evaluation: A comparison of precision, recall and f-scores for each of the five individual classes is shown in Table 2. Using mini-batch gradient descent clearly reduces the validation error. However, the effect on f-score is little since the recall for some classes actually degrade with latter algorithm.

Testing: The outputs of all three agents are shown in table 2. The SVM classifier was seen to be misclassifying several instances such as the one shown in the third row of the table. The weak generalization capacity of SVM can thus be attributed to the absence of any validation data during

Metrics (across five classes)	Algorithm	
	Basic Gradient Descent	MBGD
Precision	[0.9, 1, 0.92, 0.89, 1]	[0.91, 1, 0.92, 0.88, 1]
Recall	[0.9, 0.9, 0.92, 1, 1]	[1, 0.9, 0.92, 0.88, 1]
F-score	[0.9, 0.95, 0.92, 0.94, 1]	[0.95, 0.95, 0.92, 0.88, 1]
Validation Error	0.669	0.469

training. Also, the use of robust training methods such as early stopping and momentum have imparted the gradient descent based methods an upper hand over SVM.

Predictions of various agents			
User inputs	Basic agent	MBGD	SVM
{No, No, Yes, No, Yes, No, No, No, No}	Networking	Networking	Networking
{No, Yes, Yes, No, Yes, No, No, No, No}	Emergencies	Emergencies	Datawarehouse
{Yes, Yes, Yes, Yes, Yes, No, No, No, No}	Emergencies	Emergencies	Credentials