cs577 Assignment 2 - Report

Girish Rajani-Bathija
A20503736
Department of Computer Science
Illinois Institute of Technology
October 4, 2022

Programming Question 1:

Problem Statement

The problem is building and evaluating a neural network for 3-class classification. The code should be written in Python, whereby the model should iterate over batches, perform forward pass and backpropagation, apply the gradient loss from the backpropagation to update the weights, and compute and print the training loss/accuracy and validation loss/accuracy after each epoch.

Proposed Solution

The proposed solution is to write a training loop from scratch and implement a neural network with two hidden layers and one output layer using categorical cross entropy for loss, sigmoid activation for hidden layers and softmax activation for the output layer. This will make it possible to classify the 3 class iris dataset which will be used for classification. The performance of the model will be evaluated appropriately and the results will be displayed after each epoch.

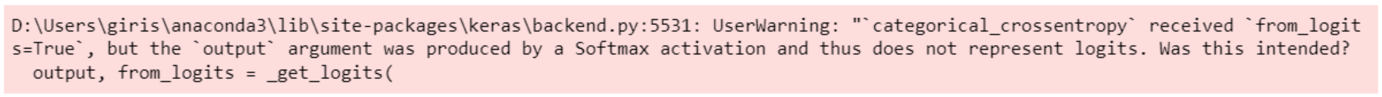
Implementation Details

When preprocessing the iris dataset, an error was encountered when vectorizing the label. Just using categorical encoding was not enough to vectorize the iris data labels. In doing so, the following error occurred.

“invalid literal for int() with base 10: 'Iris-versicolor' ”

This meant that first the labels must be encoded into integers which was done using LabelEncoder from sklearn and then it was one hot encoded.

When initializing the loss function as categorical cross entropy, from_logits was set to ‘True’ while the output layer activation was softmax. This resulted in an error message (but did not prevent the code from running) which is shown below.



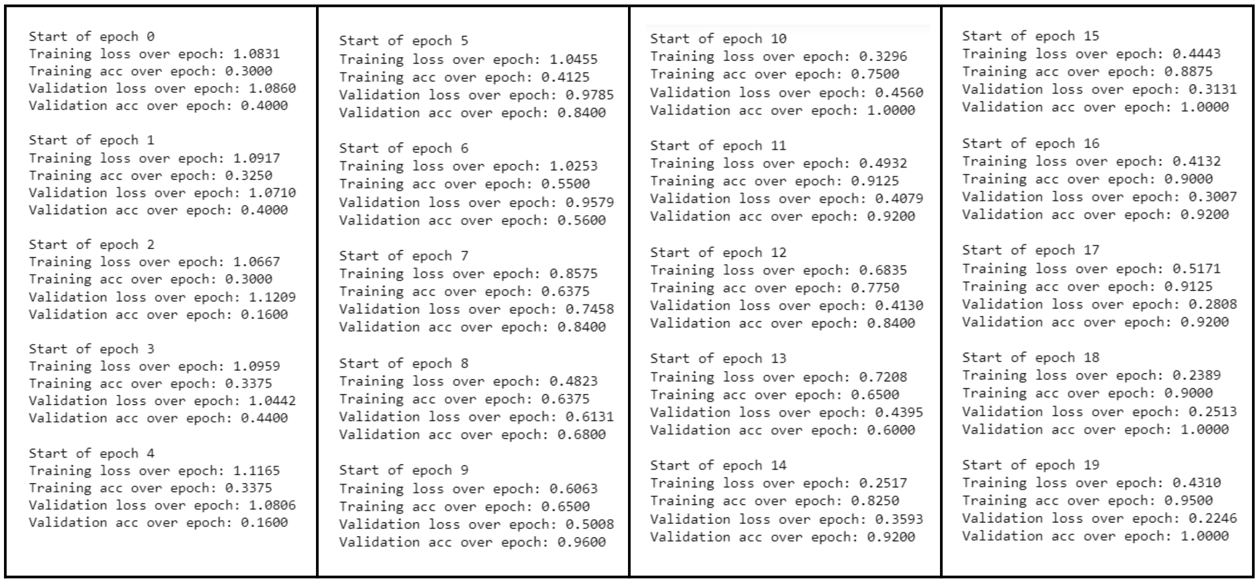
```
D:\Users\giris\anaconda3\lib\site-packages\keras\backend.py:5531: UserWarning: "`categorical_crossentropy` received `from_logit
s=True`, but the `output` argument was produced by a Softmax activation and thus does not represent logits. Was this intended?
  output, from_logits = _get_logits(
```

After conducting some research and reviewing the lectures, it was concluded that since the softmax activation in the output layer produces a probability score, there is no logits represented by this activation function therefore the ‘from_logits’ in the categorical cross entropy loss function was set to False. If the output layer was just a normal dense layer without an activation, then logits would be produced and ‘from_logits’ would have to be True.

Results and Discussion

When building the model and writing the training loop from scratch, various layers and hyperparameters were used to classify the iris dataset. For the first hidden layer, a dense layer of 64 units with a sigmoid activation function was used and the second hidden layer utilized 32 layers with the sigmoid activation function. The output layer used a softmax activation function with a 3 unit output size since the sample belonged to one of three classes. With the softmax activation, the appropriate loss function used was categorical cross entropy with an Adam optimizer and a learning rate of 0.01.

When creating the training loop from scratch, a loop was created to iterate through the batches then the inputs were pushed through the network via forward pass and the GradientTape records the operations of the inputs. The loss value was computed and the backpropagation was done by using the GradientTape to compute the gradients with respect to the loss. The gradients were then simply used to update the weights and the performance during that epoch was calculated. A batch size of 1 and training epochs of 20 was used when training the model.



```
Start of epoch 0
Training loss over epoch: 1.0831
Training acc over epoch: 0.3000
Validation loss over epoch: 1.0860
Validation acc over epoch: 0.4000

Start of epoch 1
Training loss over epoch: 1.0917
Training acc over epoch: 0.3250
Validation loss over epoch: 1.0710
Validation acc over epoch: 0.4000

Start of epoch 2
Training loss over epoch: 1.0667
Training acc over epoch: 0.3000
Validation loss over epoch: 1.1209
Validation acc over epoch: 0.1600

Start of epoch 3
Training loss over epoch: 1.0959
Training acc over epoch: 0.3375
Validation loss over epoch: 1.0442
Validation acc over epoch: 0.4400

Start of epoch 4
Training loss over epoch: 1.1165
Training acc over epoch: 0.3375
Validation loss over epoch: 1.0806
Validation acc over epoch: 0.1600
```

```
Start of epoch 5
Training loss over epoch: 1.0455
Training acc over epoch: 0.4125
Validation loss over epoch: 0.9785
Validation acc over epoch: 0.8400

Start of epoch 6
Training loss over epoch: 1.0253
Training acc over epoch: 0.5500
Validation loss over epoch: 0.9579
Validation acc over epoch: 0.5600

Start of epoch 7
Training loss over epoch: 0.8575
Training acc over epoch: 0.6375
Validation loss over epoch: 0.7458
Validation acc over epoch: 0.8400

Start of epoch 8
Training loss over epoch: 0.4823
Training acc over epoch: 0.6375
Validation loss over epoch: 0.6131
Validation acc over epoch: 0.6800

Start of epoch 9
Training loss over epoch: 0.6063
Training acc over epoch: 0.6500
Validation loss over epoch: 0.5008
Validation acc over epoch: 0.9600
```

```
Start of epoch 10
Training loss over epoch: 0.3296
Training acc over epoch: 0.7500
Validation loss over epoch: 0.4560
Validation acc over epoch: 1.0000

Start of epoch 11
Training loss over epoch: 0.4932
Training acc over epoch: 0.9125
Validation loss over epoch: 0.4079
Validation acc over epoch: 0.9200

Start of epoch 12
Training loss over epoch: 0.6835
Training acc over epoch: 0.7750
Validation loss over epoch: 0.4130
Validation acc over epoch: 0.8400

Start of epoch 13
Training loss over epoch: 0.7208
Training acc over epoch: 0.6500
Validation loss over epoch: 0.4395
Validation acc over epoch: 0.6000

Start of epoch 14
Training loss over epoch: 0.2517
Training acc over epoch: 0.8250
Validation loss over epoch: 0.3593
Validation acc over epoch: 0.9200
```

```
Start of epoch 15
Training loss over epoch: 0.4443
Training acc over epoch: 0.8875
Validation loss over epoch: 0.3131
Validation acc over epoch: 1.0000

Start of epoch 16
Training loss over epoch: 0.4132
Training acc over epoch: 0.9000
Validation loss over epoch: 0.3007
Validation acc over epoch: 0.9200

Start of epoch 17
Training loss over epoch: 0.5171
Training acc over epoch: 0.9125
Validation loss over epoch: 0.2808
Validation acc over epoch: 0.9200

Start of epoch 18
Training loss over epoch: 0.2389
Training acc over epoch: 0.9000
Validation loss over epoch: 0.2513
Validation acc over epoch: 1.0000

Start of epoch 19
Training loss over epoch: 0.4310
Training acc over epoch: 0.9500
Validation loss over epoch: 0.2246
Validation acc over epoch: 1.0000
```

As shown in the results above, the training loss and accuracy, and the validation loss and accuracy were computed and displayed after each epoch. It can be seen that the training accuracy does start off relatively low and increases substantially as the model is being trained whereby at the end, the training accuracy is in the 90's which is a good sign. However, overfitting does occur, as shown in the various fluctuations. Various parameters will be adjusted and the effect of the adjustment of the hyperparameters will be evaluated below.

Through experimentation in hopes of optimizing the model, different learning rates and optimizers were used. Specifically, instead of Adam optimizer, RMSprop, Adagrad, and SGD were used but none of them gave great results, the accuracy when using the various optimizers other than Adam was around 30-40% except for RMSprop which was able to reach around 70-80% accuracy but again, did not perform as good as Adam. The number of units per layer did not.

Instead of using 64 and 32 units in the hidden layers, 4 units in each hidden layer was used instead but the model performed worse. The training accuracy was around 30-40% which showed that having a higher number of units per layer resulted in better performance.

Programming Question 2:

Problem Statement

The problem is to implement a three layer neural network without using any deep learning frameworks (Keras, TensorFlow, or PyTorch) for a 3-class classification. The forward and backward pass for the computation graph should be hard coded into the program and the model implementation should support:

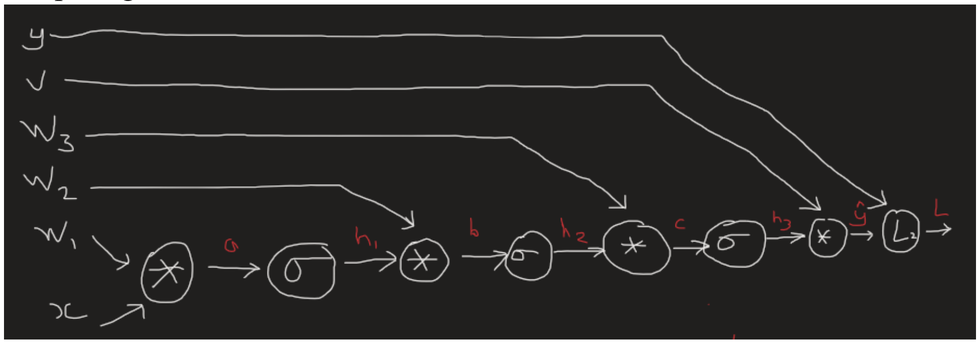
a. A loss function and an evaluation function
b. A class for each node type with methods for computing a forward and backward pass.
c. Forward and backward traversal of the computation graph where data batches are pushed forward and gradient loss values are pushed backward.
d. Implementation of stochastic gradient descent with learning rate and decay parameters for updating model weights.

Proposed Solution

The proposed solution is to create and implement a neural network from scratch consisting of classes for each node type with forward and backward pass. Various functions and calculations will be performed to manually compute the forward and backward traversal through the network. Libraries such as Pandas will be used for datapreprocessing and NumPy will be used to perform the necessary computations.

Implementation Details

In the implementation of the neural network, first, the addition, multiplication, sigmoid and softmax classes were created. These classes are essential in the computation graph for forward pass to push the inputs through the network and during the backpropagation when computing the derivatives.



The above computation graph is just a rough sketch (not accurate for this neural network) to give an understanding as to why we need these 4 classes. For forward pass, the input x and the weights are pushed through the nodes where the multiplication, addition and activation function take place to eventually reach the y_hat prediction and then compute the loss.

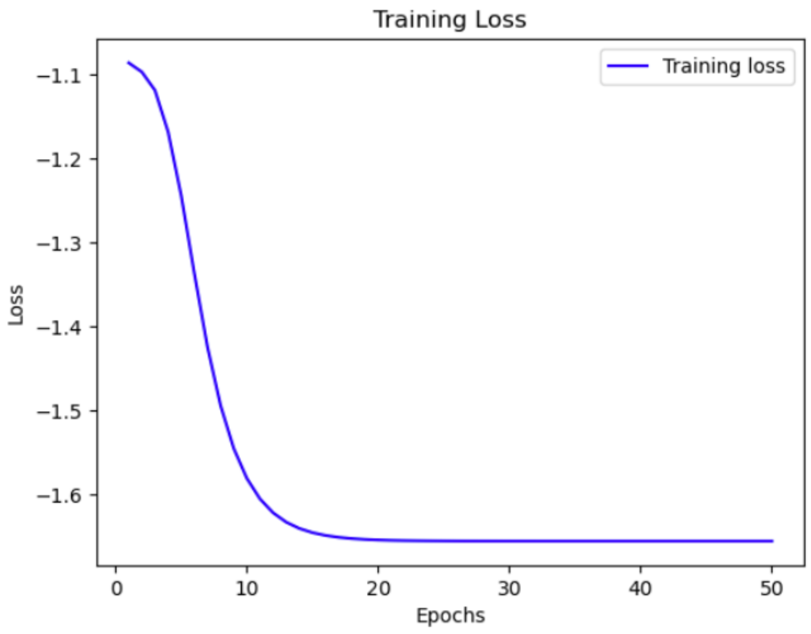
This problem required the use of categorical cross entropy so a class was created with a forward pass function to compute this loss and a backward pass function to compute the derivative of the categorical cross entropy loss. The final major step in implementing the model was creating a computation graph class to put everything together. A function to initialize the network with the appropriate nodes, weights, and loss before computation. The forward function performed the forward pass by calling the classes mentioned earlier to take the input and pass it through the network to compute the prediction, y_hat. The backward function computes derivative of the prediction y_hat and uses that to compute the derivative of the children in the backward pass and repeats this process to get to the derivative of the weights. After that we need to now train and test the network so functions were created for that. For training the network, a training loop was used to push the input through the network and compute the weight via backprop and then update the weights using the derived gradients. This is done for each epoch. Finally, to test the network on unseen data, we will output the predicted value and the actual value and compare to see if the network predicted it right then count how many predictions it got right after which the accuracy is computed by find the average

Results and Discussion

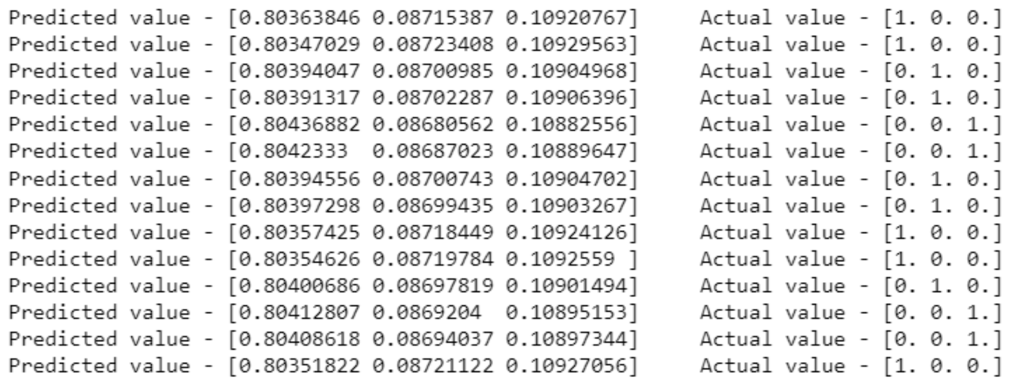
When building the model, 3 layers were used. 1 input layer with a sigmoid activation, 1 hidden layer with a sigmoid activation and 1 output layer with a softmax activation. Three nodes were used to initialize the weights which was used in the forward and backward pass of the computation graph.

Using 50 epochs with a decay rate of 0.01 and a learning rate of 0.01, the following results were obtained:

```
Accuracy on Test Data: 70.0 %
```



As shown from the results above, the model obtained an accuracy of 70% on test data with the training loss decreasing per epoch. However, after approximately 12 epochs, the training loss stops decreasing.

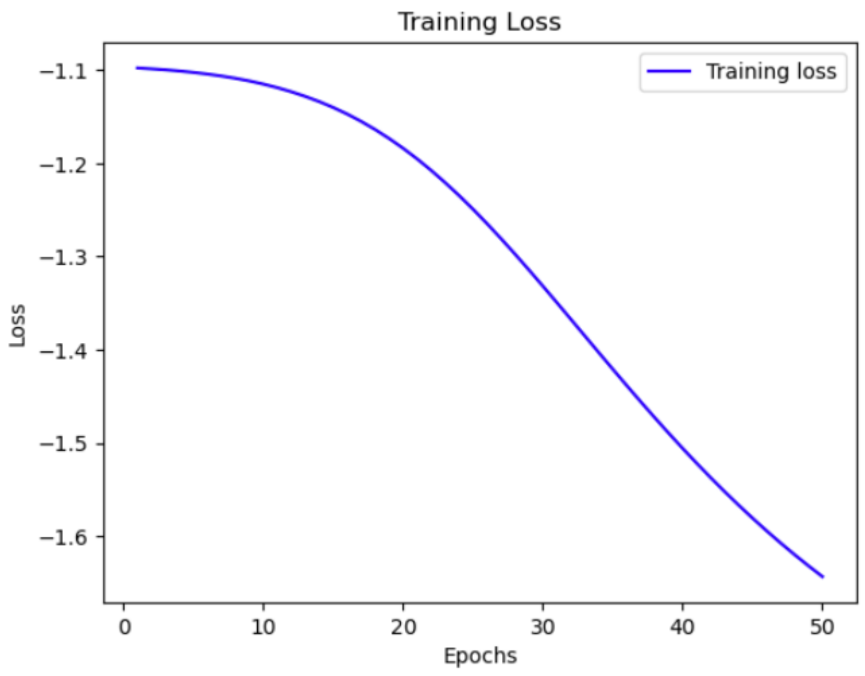


```
Predicted value - [0.80363846 0.08715387 0.10920767]     Actual value - [1. 0. 0.]
Predicted value - [0.80347029 0.08723408 0.10929563]     Actual value - [1. 0. 0.]
Predicted value - [0.80394047 0.08700985 0.10904968]     Actual value - [0. 1. 0.]
Predicted value - [0.80391317 0.08702287 0.10906396]     Actual value - [0. 1. 0.]
Predicted value - [0.80436882 0.08680562 0.10882556]     Actual value - [0. 0. 1.]
Predicted value - [0.8042333  0.08687023 0.10889647]     Actual value - [0. 0. 1.]
Predicted value - [0.80394556 0.08700743 0.10904702]     Actual value - [0. 1. 0.]
Predicted value - [0.80397298 0.08699435 0.10903267]     Actual value - [0. 1. 0.]
Predicted value - [0.80357425 0.08718449 0.10924126]     Actual value - [1. 0. 0.]
Predicted value - [0.80354626 0.08719784 0.1092559 ]     Actual value - [1. 0. 0.]
Predicted value - [0.80400686 0.08697819 0.10901494]     Actual value - [0. 1. 0.]
Predicted value - [0.80412807 0.0869204  0.10895153]     Actual value - [0. 0. 1.]
Predicted value - [0.80408618 0.08694037 0.10897344]     Actual value - [0. 0. 1.]
Predicted value - [0.80351822 0.08721122 0.10927056]     Actual value - [1. 0. 0.]
```

Above is a portion of the accuracy output obtained. It can be seen that firstly the predicted value, y_hat is displayed which is a probability distribution. It can be seen in the first prediction for example, that the actual value is the flower which in one hot encoding is [1,0,0] and we can also see that the y_hat has the highest probability (0.8) on the first value which means that it guessed that sample correctly.

To try an obtain better results, the various hyperparameters were adjusted. The results can be seen below.

```
Accuracy on Test Data: 42.5 %
```



For the results above, the decay rate was reduced from 0.01 to 0.001 and the training loss was increased from 0.01 to 0.1 and the accuracy reduced as well as the loss decreased slower so it seems for optimal results, having a decay rate and learning rate of 0.01 achieved good results.