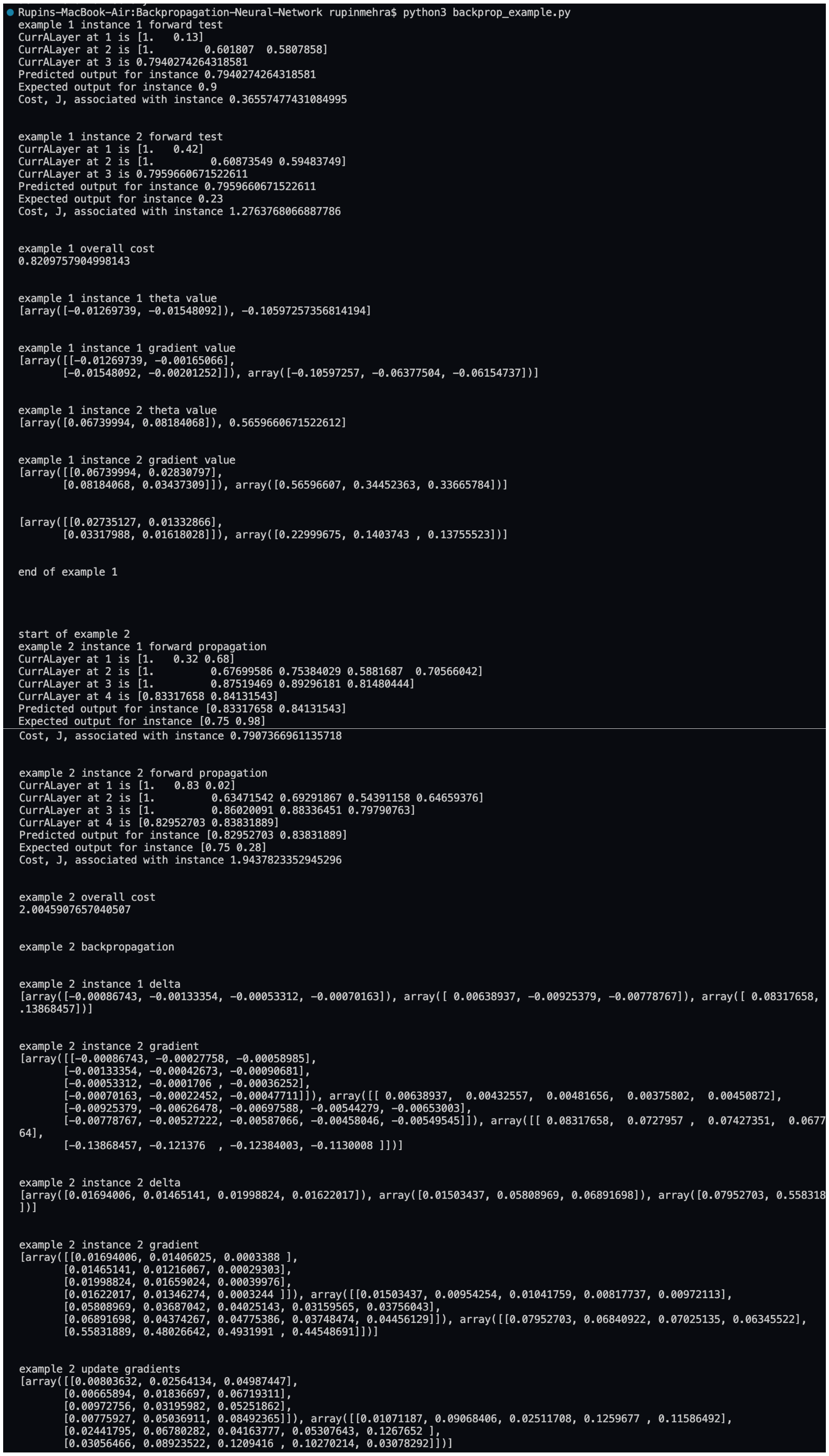
Running the backprop_example.py file:



```
● Rupins-MacBook-Air:Backpropagation-Neural-Network rupinmehra$ python3 backprop_example.py
example 1 instance 1 forward test
CurrALayer at 1 is [1.   0.13]
CurrALayer at 2 is [1.        0.601807  0.5807858]
CurrALayer at 3 is 0.7940274264318581
Predicted output for instance 0.7940274264318581
Expected output for instance 0.9
Cost, J, associated with instance 0.36557477431084995


example 1 instance 2 forward test
CurrALayer at 1 is [1.   0.42]
CurrALayer at 2 is [1.         0.60873549 0.59483749]
CurrALayer at 3 is 0.7959660671522611
Predicted output for instance 0.7959660671522611
Expected output for instance 0.23
Cost, J, associated with instance 1.2763768066887786


example 1 overall cost
0.8209757904998143


example 1 instance 1 theta value
[array([-0.01269739, -0.01548092]), -0.10597257356814194]


example 1 instance 1 gradient value
[array([[-0.01269739, -0.00165066],
       [-0.01548092, -0.00201252]]), array([-0.10597257, -0.06377504, -0.06154737])]


example 1 instance 2 theta value
[array([0.06739994, 0.08184068]), 0.5659660671522612]


example 1 instance 2 gradient value
[array([[0.06739994, 0.02830797],
       [0.08184068, 0.03437309]]), array([0.56596607, 0.34452363, 0.33665784])]


[array([[0.02735127, 0.01332866],
       [0.03317988, 0.01618028]]), array([0.22999675, 0.1403743 , 0.13755523])]


end of example 1




start of example 2
example 2 instance 1 forward propagation
CurrALayer at 1 is [1.   0.32 0.68]
CurrALayer at 2 is [1.         0.67699586 0.75384029 0.5881687  0.70566042]
CurrALayer at 3 is [1.         0.87519469 0.89296181 0.81480444]
CurrALayer at 4 is [0.83317658 0.84131543]
Predicted output for instance [0.83317658 0.84131543]
Expected output for instance [0.75 0.98]
Cost, J, associated with instance 0.7907366961135718


example 2 instance 2 forward propagation
CurrALayer at 1 is [1.   0.83 0.02]
CurrALayer at 2 is [1.         0.63471542 0.69291867 0.54391158 0.64659376]
CurrALayer at 3 is [1.         0.86020091 0.88336451 0.79790763]
CurrALayer at 4 is [0.82952703 0.83831889]
Predicted output for instance [0.82952703 0.83831889]
Expected output for instance [0.75 0.28]
Cost, J, associated with instance 1.9437823352945296


example 2 overall cost
2.0045907657040507


example 2 backpropagation


example 2 instance 1 delta
[array([-0.00086743, -0.00133354, -0.00053312, -0.00070163]), array([ 0.00638937, -0.00925379, -0.00778767]), array([ 0.08317658,
.13868457])]


example 2 instance 2 gradient
[array([[-0.00086743, -0.00027758, -0.00058985],
       [-0.00133354, -0.00042673, -0.00090681],
       [-0.00053312, -0.0001706 , -0.00036252],
       [-0.00070163, -0.00022452, -0.00047711]]), array([[ 0.00638937,  0.00432557,  0.00481656,  0.00375802,  0.00450872],
       [-0.00925379, -0.00626478, -0.00697588, -0.00544279, -0.00653003],
       [-0.00778767, -0.00527222, -0.00587066, -0.00458046, -0.00549545]]), array([[ 0.08317658,  0.0727957 ,  0.07427351,  0.0677
64],
       [-0.13868457, -0.121376  , -0.12384003, -0.1130008 ]])]


example 2 instance 2 delta
[array([0.01694006, 0.01465141, 0.01998824, 0.01622017]), array([0.01503437, 0.05808969, 0.06891698]), array([0.07952703, 0.558318
])]


example 2 instance 2 gradient
[array([[0.01694006, 0.01406025, 0.0003388 ],
       [0.01465141, 0.01216067, 0.00029303],
       [0.01998824, 0.01659024, 0.00039976],
       [0.01622017, 0.01346274, 0.0003244 ]]), array([[0.01503437, 0.00954254, 0.01041759, 0.00817737, 0.00972113],
       [0.05808969, 0.03687042, 0.04025143, 0.03159565, 0.03756043],
       [0.06891698, 0.04374267, 0.04775386, 0.03748474, 0.04456129]]), array([[0.07952703, 0.06840922, 0.07025135, 0.06345522],
       [0.55831889, 0.48026642, 0.4931991 , 0.44548691]])]


example 2 update gradients
[array([[0.00803632, 0.02564134, 0.04987447],
       [0.00665894, 0.01836697, 0.06719311],
       [0.00972756, 0.03195982, 0.05251862],
       [0.00775927, 0.05036911, 0.08492365]]), array([[0.01071187, 0.09068406, 0.02511708, 0.1259677 , 0.11586492],
       [0.02441795, 0.06780282, 0.04163777, 0.05307643, 0.1267652 ],
       [0.03056466, 0.08923522, 0.1209416 , 0.10270214, 0.03078292]])]
```

Trained with MiniBatch, vectorized neural network, divided to 2 groups, (batchsize = 435/2 = 217). Done by the virtue of the miniBatchK variable. Also, lambda = 0.1, 0.15 and learning rate = 0.1

House Data Analysis:

Hidden Layers	[4, 4, 4]	[4, 4]	[2]	[4]	[8, 8]	[8]	[16, 16]
Epoch	1000	1000	1000	1000	1000	1000	1000
Accuracy (Lambda 0.1, 0.15)	0.9584 0.9578	0.9586 0.9583	0.9634 0.9629	0.9632 0.9630	0.9607 0.9598	0.9609 0.9602	0.9610 0.9608
F-1 Score (Lambda 0.1, 0.15)	0.9465 0.9462	0.9459 0.9448	0.9525 0.9526	0.9529 0.9522	0.9486 0.9481	0.9499 0.9495	0.9490 0.9486

The House Voting Data set has demonstrated a high level of accuracy, with most models achieving around 0.9 accuracy. Upon closer inspection of the data, it can be observed that the best performance is from models with one hidden layer, specifically [2] or [4]. These models achieved an accuracy of 0.9634 and 0.9525, respectively, with corresponding F-scores of 0.9632 and 0.9529.
Interestingly, increasing the number of neurons per layer did not lead to significant improvements in performance for this dataset. In fact, it is plausible that logistic regression, with an empty hidden layer and direct calculation of output from input, may achieve high accuracy due to the data's relative lack of complexity.
For [4] and [4,4], both with 1000 epochs, the performance of [4] was better than [4,4]. My reasoning is that with more layers and neurons, there will be more weights, requiring the model more epochs to converge to a minimum value, but with only 1000 epochs, the model still hasn't converged to a nice value yet.
Analysis of the graph reveals several noteworthy points. When the number of layers is limited to 1 or 2, the j graph displays a smoother convergence, indicating a more straightforward path towards a final weight without getting stuck at a local minimum. While adding more layers and neurons generally led to better performance, there was a sharp drop-off in performance after a certain point.

Hence, in practice, a [2] or [4] hidden layer model may be the most suitable for this dataset. However, it may be beneficial to train the model using a slightly smaller learning rate and more epochs to achieve better convergence. Overall, these observations underscore the importance of carefully considering the dataset's characteristics when selecting and fine-tuning neural network models.

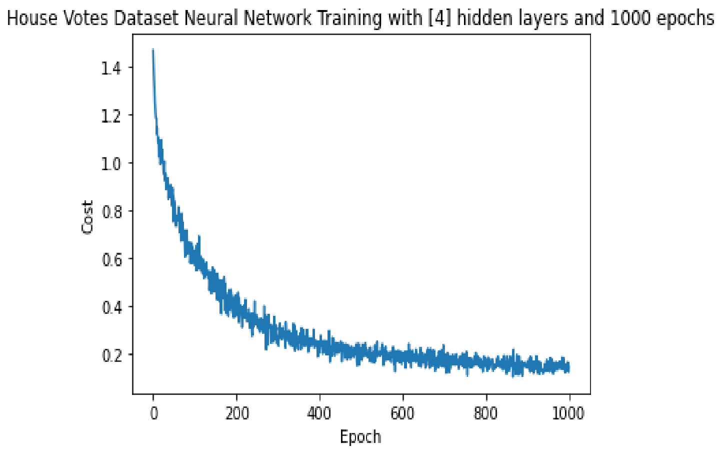
Wine Data Analysis:

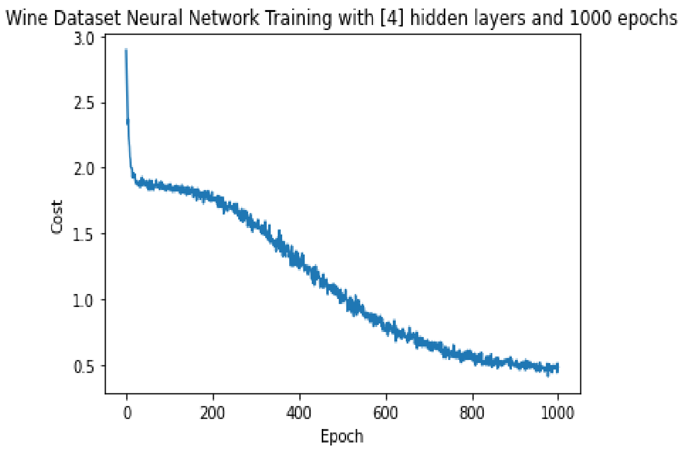
Trained with MiniBatch, vectorized neural network, divided to 2 groups, (batchsize = 435/2 = 217). Done by the virtue of the miniBatchK variable. The learning rate of them are 0.05 and 0.1

Hidden Layers	[4, 4, 4]	[4, 4]	[2]	[4]	[8, 8]	[8]	[16, 16]
Epoch	1000	1000	1000	1000	1000	1000	1000
Accuracy (LR 0.05, 0.1)	0.5167 0.5162	0.9502 0.9498	0.9722 0.9724	0.9889 0.9892	0.9889 0.9885	0.9764 0.9768	0.9778 0.9772
F-1 Score (LR 0.05, 0.1)	0.6008 0.6012	0.9790 0.9786	0.9631 0.9632	0.987 0.9873	0.9856 09852	0.9714 0.9723	0.9689 0.9692

The performance of neural networks on the wine dataset is generally quite good, with an accuracy and f-score of around 0.9, except for the case of [4, 4, 4]. In order to improve the performance of the network, I conducted various tests, adjusting the number of epochs, learning rate, and layer data.
First, I tested the single hidden layer [2] with different epochs. The results showed that with an epoch of 500, the performance was around 0.95, with 1000 epochs the accuracy increased to around 0.983, and with 2000 epochs, the accuracy improved to about 0.9899. This demonstrates that with higher epochs, the j function converges to its smallest value. As shown in the plot figures, we can see how j converges for all [2] hidden layer neural networks.
Next, I changed the number of layers to see how it affected performance. For [4] and [4,4], both with 1000 epochs, the performance of [4] was better than [4,4]. My reasoning is that with more layers and neurons, there will be more weights, requiring the model more epochs to converge to a minimum value, but with only 1000 epochs, the model still hasn't converged to a nice value yet.
Then, I modified the learning rate of the data, changing [4,4] from a learning rate of 0.1 to 0.05, and doubling the epoch. Intuitively, I thought the learning rate would make the gradient descent process slower, requiring more epochs to converge. The results showed that the performance improved, with accuracy increasing from 0.9498 to 0.9502, and f-score improving from 0.9786 to 0.9790.
Finally, I changed the [4,4] to [8,8], holding the other parameters unchanged. However, the performance did not improve, demonstrating that the maximum performance is around [4,4] with one or two layers.

Hence, In practice, a [4] hidden layer model may be the most suitable for this dataset.





Implement the vectorized form of backpropagation. As previously mentioned, this is not mandatory, but you can get extra credit if you do it. Most modern implementations of neural networks use vectorization, so it is useful for you to familiarize yourself with this type of approach.

I have implemented the vectorized form of backpropagation.