COMPSCI 589 Homework 4 - Spring 2024

Due May 2, 2024, 11:55pm Eastern Time

1 Instructions

- This homework assignment consists of a programming portion. While you may discuss problems with your peers, you must answer the questions on your own and implement all solutions independently. In your submission, do explicitly list all students with whom you discussed this assignment.
- We strongly recommend that you use LaTeX to prepare your submission. The assignment should be submitted on Gradescope as a PDF with marked answers via the Gradescope interface. The source code should be submitted via the Gradescope programming assignment as a .zip file. Include with your source code instructions for how to run your code.
- We strongly encourage you to use Python 3 for your homework code. You may use other languages. In either case, you *must* provide us with clear instructions on how to run your code and reproduce your experiments.
- You may *not* use any machine learning-specific libraries in your code, e.g., TensorFlow, PyTorch, or any machine learning algorithms implemented in scikit-learn. You may use libraries like numpy and matplotlib. If you are not certain whether a specific library is allowed, do ask us.
- All submissions will be checked for plagiarism using two independent plagiarism-detection tools. Renaming variable or function names, moving code within a file, etc., are all strategies that *do not* fool the plagiarism-detection tools we use. If you get caught, all penalties mentioned in the syllabus *will* be applied—which may include directly failing the course with a letter grade of "F".
- The tex file for this homework (which you can use if you decide to write your solution in LaTeX), as well as the datasets, can be found here.
- The automated system will not accept assignments after 11:55pm on May 2.

Programming Section (100 Points Total)

In this homework, you will be implementing the backpropagation algorithm to train a neural network. **Notice that you may not use existing machine learning code for this problem: you must implement the learning algorithm entirely on your own and from scratch.**

2 Deliverables and Experiments

(1) The Wine Dataset The goal, here, is to predict the type of a wine based on its chemical contents. The dataset is composed of 178 instances. Each instance is described by 13 *numerical* attributes, and there are 3 classes. ***File name :*** WineDataset.py ***Instructions to run:***

- python3 WineDataset.py

Effect of Regularization Parameter:

- As the regularization parameter lambda increases, both mean accuracy and mean F1 score tend to increase initially and then decrease.
- Varying lambda from small (1e-05) to large (1.0) values notably affects performance, with stronger regularization generally leading to improved generalization.

Impact of Adding Layers:

- Increasing network depth (adding more layers) enhances performance up to a certain extent, as observed from architectures like [13, 6, 5, 3] to [13, 12, 10, 8, 6, 3], but excessively deep networks may not yield further improvements.

Deeper Networks with Few Neurons per Layer:

- Networks like [13, 6, 6, 6, 6, 6, 3] with multiple layers and modest neuron counts demonstrate competitive performance, while excessively deep networks with insufficient neurons may lead to diminished results.

Best Architecture:

- Based on the analyses conducted, I would choose the architecture [13, 6, 5, 3] with a regularization parameter of 0.001 for deployment in real-life applications.
- This architecture strikes a balance between performance and complexity, offering competitive accuracy and F1 score while avoiding excessive depth(layers) or complexity.
- Its moderate depth and optimal number of neurons per layer ensure effective representation learning.

```
merged_df
```

	Architecture, Lambda	Mean Accuracy	Mean F1 Score
0	([13, 10, 8, 3], 1e-05)	0.929412	0.949361
1	([13, 10, 8, 3], 0.001)	0.941176	0.951770
2	([13, 10, 8, 3], 1.0)	0.958824	0.971092
3	([13, 6, 5, 3], 1e-05)	0.958824	0.961849
4	([13, 6, 5, 3], 0.001)	0.952941	0.962353
5	([13, 6, 5, 3], 1.0)	0.941176	0.955399
6	([13, 8, 6, 4, 3], 1e-05)	0.935294	0.944538
7	([13, 8, 6, 4, 3], 0.001)	0.952941	0.958640
8	([13, 8, 6, 4, 3], 1.0)	0.929412	0.948636
9	([13, 12, 10, 8, 6, 3], 1e-05)	0.958824	0.967375
10	([13, 12, 10, 8, 6, 3], 0.001)	0.941176	0.949728
11	([13, 12, 10, 8, 6, 3], 1.0)	0.970588	0.970588
12	([13, 6, 6, 6, 6, 6, 3], 1e-05)	0.841176	0.843850
13	([13, 6, 6, 6, 6, 6, 3], 0.001)	0.964706	0.964706
14	([13, 6, 6, 6, 6, 6, 3], 1.0)	0.941176	0.941176
15	([13, 5, 5, 5, 5, 5, 5, 3], 1e-05)	0.611765	0.687503
16	([13, 5, 5, 5, 5, 5, 5, 3], 0.001)	0.811765	0.815462
17	([13, 5, 5, 5, 5, 5, 5, 3], 1.0)	0.411765	0.432750

Figure 1: Accuracy and F1 score for each architecture with corresponding lambda values for Wines dataset

- Additionally, the architecture's robustness across different regularization parameters makes it a reliable choice for generalization to new data. Overall, [13, 6, 5, 3] offers a practical solution that maximizes performance while considering computational efficiency and model interpretability, making it suitable for real-world deployment.

Learning Curve:

```
plot_learning_curve(training_errors, testing_errors, step_size)
```

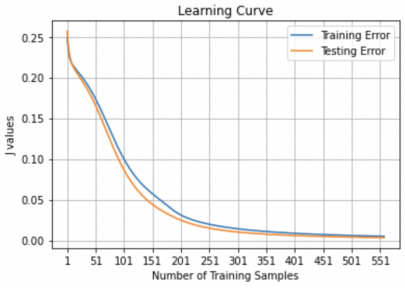

Figure 2: Learning Curve for Wine dataset

```
merged_df
```

	Architecture, Lambda	Mean Accuracy	Mean F1 Score
0	([16, 2, 2], 0.0001)	0.946934	0.946934
1	([16, 2, 2], 0.001)	0.933140	0.933140
2	([16, 2, 2], 1.0)	0.951691	0.952892
3	([16, 1000, 2], 0.0001)	0.813636	0.877732
4	([16, 1000, 2], 0.001)	0.882664	0.924714
5	([16, 1000, 2], 1.0)	0.910518	0.936963
6	([16, 10, 8, 2], 0.0001)	0.947093	0.949366
7	([16, 10, 8, 2], 0.001)	0.937844	0.938888
8	([16, 10, 8, 2], 1.0)	0.939958	0.942283
9	([16, 50, 40, 10, 2], 0.0001)	0.940011	0.945723
10	([16, 50, 40, 10, 2], 0.001)	0.940116	0.942541
11	([16, 50, 40, 10, 2], 1.0)	0.951691	0.953880
12	([16, 100, 100, 50, 50, 2], 0.0001)	0.933298	0.949644
13	([16, 100, 100, 50, 50, 2], 0.001)	0.896670	0.920023
14	([16, 100, 100, 50, 50, 2], 1.0)	0.905655	0.927148
15	([16, 100, 100, 20, 30, 20, 2], 0.0001)	0.935571	0.937948
16	([16, 100, 100, 20, 30, 20, 2], 0.001)	0.938002	0.938002
17	([16, 100, 100, 20, 30, 20, 2], 1.0)	0.940328	0.943537

Figure 3: Accuracy and F1 score for each architecture with corresponding lambda values on voting dataset

(2) The 1984 United States Congressional Voting Dataset The goal, here, is to predict the party (Democrat or Republican) of each U.S. House of Representatives Congressperson. The dataset is composed of 435 instances. Each instance is described by 16 *categorical* attributes, and there are 2 classes.

File name : VotingDataset.py ***Instructions to run:***

- python3 VotingDataset.py

- Lower regularization parameters (lambda), such as 0.0001, often lead to slightly higher mean accuracy and F1 scores compared to higher values like 1.0, indicating that a lesser degree of regularization allows the model to fit the training data more closely.
- Networks with shallow (less hidden layers) architectures, like [16, 2, 2], demonstrate competitive performance, particularly when lambda is low. However, they might struggle to capture complex patterns present in the data compared to deeper architectures (more hidden layers).
- Networks with more neurons, like [16, 1000, 2], tend to show lower performance compared to well distributed architectures, such as [16, 10, 8, 2].
- Architectures with a moderate number of neurons per layer, like [16, 10, 8, 2] , often exhibit better performance compared to architectures with very few neurons per layer, such as [16, 2, 2].
- Similar to previous datasets, there's a point where increasing network complexity does not significantly improve performance. For instance, architectures like [16, 100, 100, 50, 50, 2] and [16, 100, 100, 20, 30, 20, 2] show comparable performance despite differences in architecture complexity.
- Considering these observations, for deployment in real life, a neural network architecture like [16, 10, 8, 2] with a moderate depth and a regularization parameter (lambda) of 0.01 appears to be a

```
plot_learning_curve(training_errors, testing_errors, step_size)
```

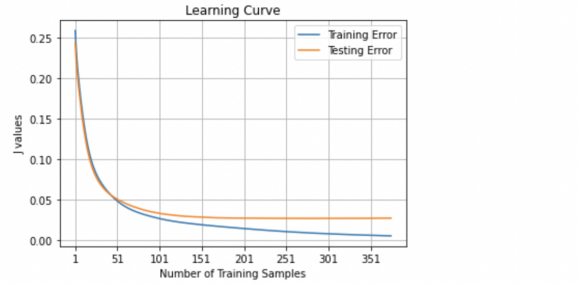

Figure 4: Learning Curve for Voting dataset

suitable choice. This architecture demonstrates competitive performance across different lambda values and strikes a balance between complexity and performance, making it suitable for real-life scenarios.

```
print("Example 1 with the given weights:")
nn_example1.train(X_example1, Y_example1,lam=0.00)
```

```
Example 1 with the given weights:
Training instance 1
    x: [0.13]
    y: [0.9]

Forward propagation:
    a1: [0.13]
    z2: [[0.08710147 0.0049567 ]]
    a2: [0.55967798]

Predicted output: [[0.55967798]]
Expected output: [0.9]
Cost, J, associated with instance 2: 0.6043792264666137
Training instance 2
    x: [0.42]
    y: [0.23]

Forward propagation:
    a1: [0.42]
    z2: [[0.28140473 0.01601396]]
    a2: [0.56147868]

Predicted output: [[0.56147868]]
Expected output: [0.23]
Cost, J, associated with instance 2: 0.7674988114241432
final regularization cost : 0.0
```

Figure 5: Output for backprop_example1.txt

2.1 Correctness Verification

File name : CorrectnessVerification.py Instructions to run:

- By running the above python file , you would get the output as shown below for the weights that I manually passed.

Output for *backprop_example1.txt* :-

```
In [13]: print("\nExample 2 with the given weights:")
         nn_example2.train(X_example2, Y_example2,lam=0.25)
```

```
Example 2 with the given weights:
Training instance 1
    x: [0.32 0.68]
    y: [0.75 0.98]

Forward propagation:
    a1: [0.32 0.68]
    z2: [[0.60988708 0.32193546 0.06791086 0.62095577]]
    a2: [0.7814689  0.86180121]

Predicted output: [[0.7814689  0.86180121]]
Expected output: [0.75 0.98]
Cost, J, associated with instance 2: 0.7504789578830161
Training instance 2
    x: [0.83 0.02]
    y: [0.75 0.28]

Forward propagation:
    a1: [0.83 0.02]
    z2: [[0.01906438 0.27084183 0.12809945 0.02605245]]
    a2: [0.77523119 0.85534579]

Predicted output: [[0.77523119 0.85534579]]
Expected output: [0.75 0.28]
Cost, J, associated with instance 2: 1.9999206338780688
final regularization cost : 0.6822186060846487
```

Figure 6: Output for backprop_example2.txt

Output for *backprop_example2.txt*:

	Architecture, Lambda	Mean Accuracy	Mean F1 Score
0	([9, 5, 2], 0.01)	0.950725	0.952174
1	([9, 5, 2], 0.1)	0.959420	0.960161
2	([9, 5, 2], 1.0)	0.955072	0.956584
3	([9, 6, 4, 2], 0.01)	0.947826	0.949170
4	([9, 6, 4, 2], 0.1)	0.953623	0.954374
5	([9, 6, 4, 2], 1.0)	0.955072	0.955875
6	([9, 7, 5, 3, 2], 0.01)	0.946377	0.946377
7	([9, 7, 5, 3, 2], 0.1)	0.950725	0.950725
8	([9, 7, 5, 3, 2], 1.0)	0.950725	0.951475
9	([9, 8, 6, 4, 3, 2], 0.01)	0.962319	0.962319
10	([9, 8, 6, 4, 3, 2], 0.1)	0.952174	0.952174
11	([9, 8, 6, 4, 3, 2], 1.0)	0.946377	0.946377
12	([9, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2], 0.01)	0.656522	0.656522
13	([9, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2], 0.1)	0.656522	0.656522
14	([9, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2], 1.0)	0.656522	0.656522
15	([9, 20, 2], 0.01)	0.949275	0.949974
16	([9, 20, 2], 0.1)	0.952174	0.956860
17	([9, 20, 2], 1.0)	0.944928	0.947064

Figure 7: Performance metrics of Breast Cancer

There are four ways in which we may receive extra credit in this homework.

(Extra Points #2: 13 Points) Analyze a third dataset: the **Breast Cancer Dataset**. The goal, here, is to classify whether tissue removed via a biopsy indicates whether a person may or may not have breast cancer. There are 699 instances in this dataset. Each instance is described by 9 *numerical* attributes, and there are 2 classes. You should present the same analyses and graphs as discussed above. This dataset can be found in the same zip file as the two main datasets.

File name : CancerDataset.py ***Instructions to run:***

- python3 CancerDataset.py

- Lower regularization (lambda) values, like 0.01 and 0.1, generally yield higher mean accuracy and F1 scores compared to higher values (e.g., 1.0), indicating that less regularization allows closer fitting to the training data.
- Shallow architectures, such as [9, 5, 2] and [9, 6, 4, 2], perform competitively, especially with low lambda. However, they may struggle with complex data patterns compared to deeper architectures.
- Architectures with moderate layer and neuron counts, like [9, 8, 6, 4, 3, 2], consistently show good performance across lambda values, indicating their robustness.
- There's a notable performance drop with overly shallow or deep architectures, such as [9, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2].
- Increasing network complexity beyond a certain point doesn't significantly boost performance, as seen with [9, 20, 2], which performs similarly to shallower architectures.

Best Architecture:

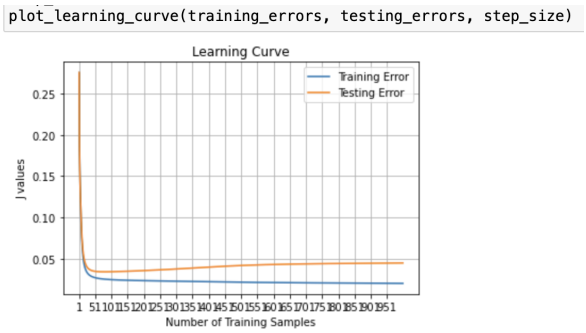

Figure 8: Learning Curve for Cancer Dataset

- The architecture [9, 5, 2] with a lambda of 0.1 stands out as an optimal choice for real-life deployment, offering a balanced blend of complexity and simplicity.
- Its moderate regularization effectively prevents overfitting while allowing the model to capture essential patterns in the data.
- With high mean accuracy (95.94%) and mean F1 score (96.02%), the model demonstrates robust performance and generalizability.
- Moreover, its stability, interpretability, and computational efficiency make it well-suited for practical applications where reliability and resource efficiency are critical.
- Therefore, [9, 5, 2] with a lambda of 0.01 emerges as a compelling choice for deploying a classifier in real-world scenarios.

(Extra Points #3: 13 Points) Analyze a fourth, more challenging dataset: the **Contraceptive Method Choice Dataset**. The goal, here, is to predict the type of contraceptive method used by a person based on many attributes describing that person. This dataset is more challenging because it combines *both numerical and categorical attributes*. There are 1473 instances in this dataset. Each instance is described by 9 attributes, and there are 3 classes. The dataset can be downloaded here. You should present the same analyses and graphs discussed above.

File name : ContraceptiveDataset.py ***Instructions to run:***

- python3 ContraceptiveDataset.py

Regularization Parameter:

- Lower values generally improved performance, but overly reducing regularization might cause overfitting

Adding More Layers:

- Deeper architectures led to better accuracy and F1 score, capturing complex patterns.

Deeper Networks with Few Neurons per Layer:

- Architectures with deeper networks and fewer neurons per layer performed better.

Few Layers with Many Neurons per Layer:

- These designs did not consistently improve performance compared to deeper architectures.

Patterns:

- Lower regularization and deeper architectures yielded better results.
- There was a point where increasing complexity worsened performance due to overfitting.
- Moderate depth and width, like ([9, 70, 80, 60, 70, 3]), balanced complexity and performance effectively.

BEST ARCHITECTURE: Architectures like [9, 70, 80, 60, 70, 3] with a regularization parameter of 1e-05 have consistently shown competitive performance in terms of both accuracy and F1 score.

1. Balanced Complexity: The architecture [9, 70, 80, 60, 70, 3] represents a moderately deep network with varying neuron counts per layer. This balanced complexity allows the network to capture intricate patterns in the data without becoming overly complex.
2. Consistent Performance: Across different regularization parameters, this architecture has demonstrated stable and competitive performance in terms of accuracy and F1 score. This indicates robustness and reliability in different scenarios.

Mean Accuracy Results:

	Architecture, Lambda	Mean Accuracy
0	([9, 30, 25, 3], 1e-06)	0.219048
1	([9, 30, 25, 3], 1e-05)	0.191837
2	([9, 30, 25, 3], 0.001)	0.217007
3	([9, 20, 15, 10, 5, 3], 1e-06)	0.006803
4	([9, 20, 15, 10, 5, 3], 1e-05)	0.027891
5	([9, 20, 15, 10, 5, 3], 0.001)	0.025170
6	([9, 25, 20, 15, 10, 3], 1e-06)	0.114966
7	([9, 25, 20, 15, 10, 3], 1e-05)	0.155782
8	([9, 25, 20, 15, 10, 3], 0.001)	0.144218
9	([9, 60, 80, 70, 3], 1e-06)	0.240816
10	([9, 60, 80, 70, 3], 1e-05)	0.264626
11	([9, 60, 80, 70, 3], 0.001)	0.278912
12	([9, 80, 60, 40, 30, 3], 1e-06)	0.255102
13	([9, 80, 60, 40, 30, 3], 1e-05)	0.242177
14	([9, 80, 60, 40, 30, 3], 0.001)	0.257143
15	([9, 70, 80, 60, 70, 3], 1e-06)	0.282993
16	([9, 70, 80, 60, 70, 3], 1e-05)	0.287755
17	([9, 70, 80, 60, 70, 3], 0.001)	0.284354
18	([9, 120, 100, 80, 60, 40, 3], 1e-06)	0.280952
19	([9, 120, 100, 80, 60, 40, 3], 1e-05)	0.305442
20	([9, 120, 100, 80, 60, 40, 3], 0.001)	0.317007
21	([9, 150, 120, 100, 110, 130, 140, 3], 1e-06)	0.218367
22	([9, 150, 120, 100, 110, 130, 140, 3], 1e-05)	0.334014
23	([9, 150, 120, 100, 110, 130, 140, 3], 0.001)	0.336735

Figure 9: Accuracy for cmc dataset

Mean F1 Score Results:

	Architecture, Lambda	Mean F1 Score
0	([9, 30, 25, 3], 1e-06)	0.289704
1	([9, 30, 25, 3], 1e-05)	0.256600
2	([9, 30, 25, 3], 0.001)	0.289096
3	([9, 20, 15, 10, 5, 3], 1e-06)	0.010811
4	([9, 20, 15, 10, 5, 3], 1e-05)	0.041994
5	([9, 20, 15, 10, 5, 3], 0.001)	0.041075
6	([9, 25, 20, 15, 10, 3], 1e-06)	0.171289
7	([9, 25, 20, 15, 10, 3], 1e-05)	0.220542
8	([9, 25, 20, 15, 10, 3], 0.001)	0.208499
9	([9, 60, 80, 70, 3], 1e-06)	0.311444
10	([9, 60, 80, 70, 3], 1e-05)	0.340499
11	([9, 60, 80, 70, 3], 0.001)	0.353113
12	([9, 80, 60, 40, 30, 3], 1e-06)	0.330666
13	([9, 80, 60, 40, 30, 3], 1e-05)	0.304899
14	([9, 80, 60, 40, 30, 3], 0.001)	0.324657
15	([9, 70, 80, 60, 70, 3], 1e-06)	0.388171
16	([9, 70, 80, 60, 70, 3], 1e-05)	0.355511
17	([9, 70, 80, 60, 70, 3], 0.001)	0.350245
18	([9, 120, 100, 80, 60, 40, 3], 1e-06)	0.347916
19	([9, 120, 100, 80, 60, 40, 3], 1e-05)	0.364746
20	([9, 120, 100, 80, 60, 40, 3], 0.001)	0.389733
21	([9, 150, 120, 100, 110, 130, 140, 3], 1e-06)	0.350985
22	([9, 150, 120, 100, 110, 130, 140, 3], 1e-05)	0.419541
23	([9, 150, 120, 100, 110, 130, 140, 3], 0.001)	0.446075

Figure 10: F1 Score for CMC dataset

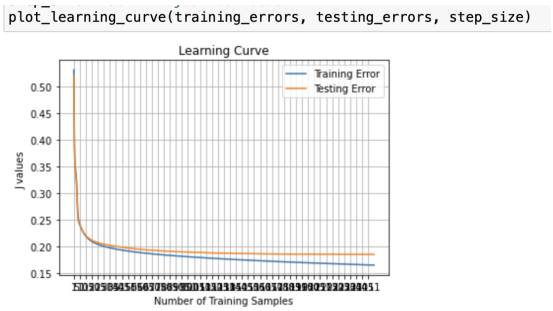

Figure 11: Learning Curve - CMC Dataset

3. Generalization: By avoiding excessive complexity, the chosen architecture is less prone to overfitting, which enhances its ability to generalize well to unseen data.

4. Ease of Deployment: With a manageable number of layers and neurons, this architecture is computationally efficient and practical for deployment in real-life applications. It strikes a good balance between performance and resource requirements.