Leiner Suarez (3054312557)

Report of Homework 2

Part one:

Open the folder Hw2 on this link https://github.com/milansua/Homework-AI to gitlab  you will find most of the specific requirements submittals such as :

Training code ("traing code commented good")

Trained Model ("deep_nn_model")

Testing code("hw2")

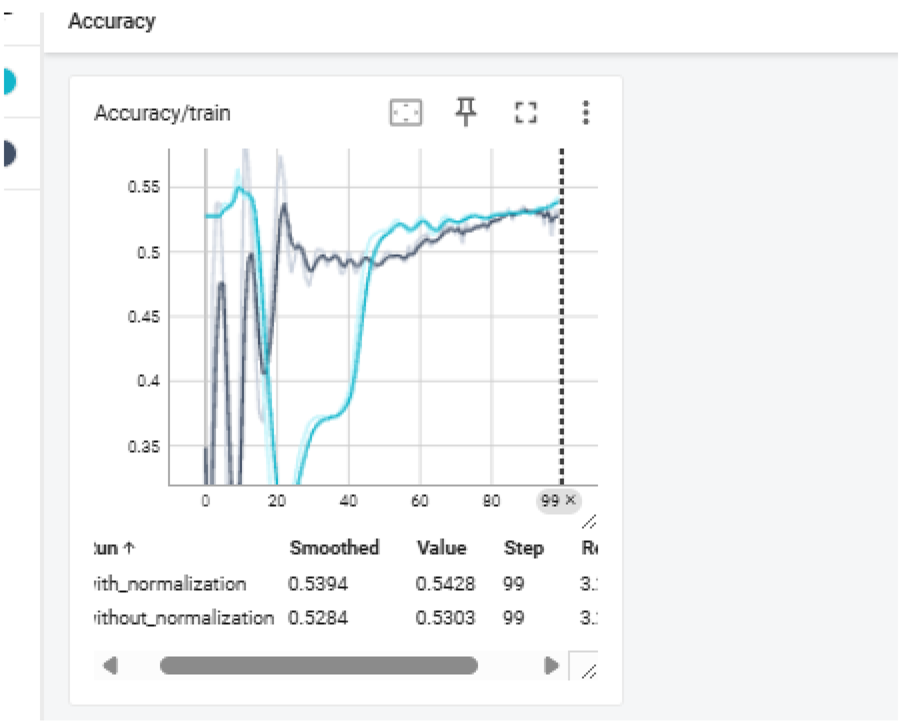
Some picture showing matplolib validation losses per epoch.

***Note:*** *Tensor Board were used for the second part. There are many other files in the gitlab repository  as well that were using for testing or data preparation, you can take a look but do not pay much attention since they are just preliminary.*

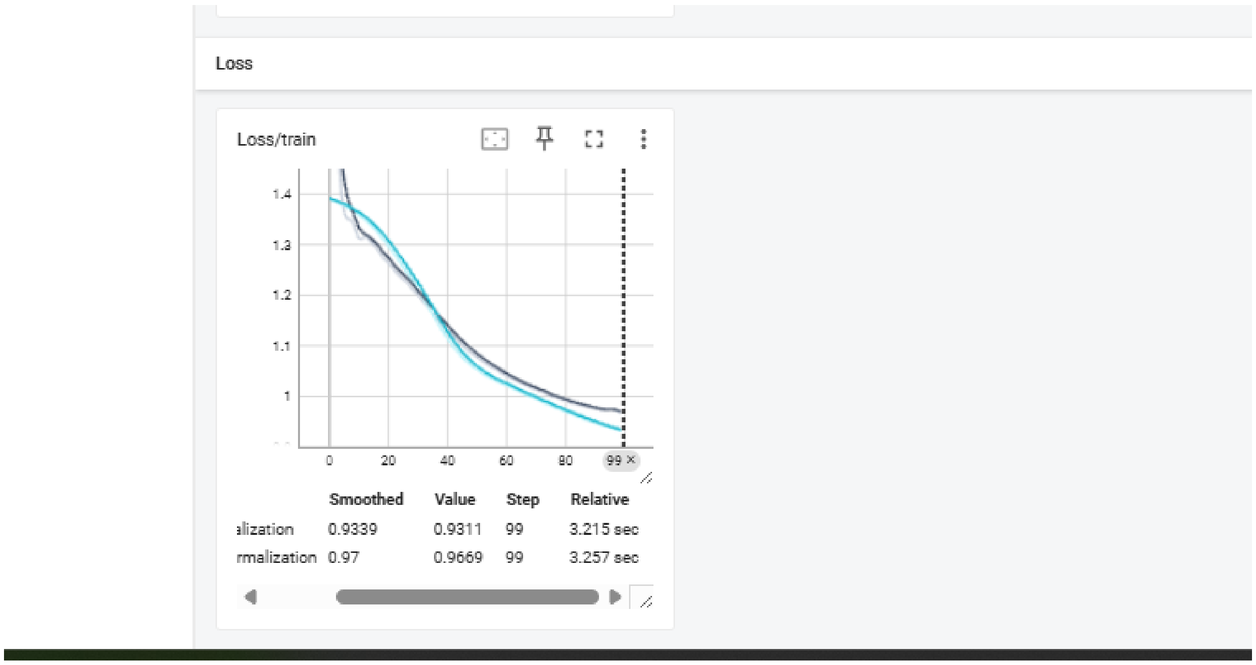
## 1-Comparison: With vs Without Normalization

In our experiments, we evaluated the performance of a deep neural network trained both with and without feature normalization (standard scaling). The results showed that normalization provided a modest improvement in final accuracy, increasing it from 53.0% without normalization to 54.3% with normalization. Although the accuracy gain was slight (~1.3%), the training process with normalization was significantly more stable, especially after the 40th epoch, where the model exhibited smoother learning curves and fewer fluctuations. In contrast, training without normalization showed more erratic behavior in the early stages.

In terms of loss, normalization also led to better convergence. The final training loss was slightly lower with normalization (0.931) compared to without (0.967), and the loss curve showed a faster and smoother decrease across epochs. This widening gap between the two curves after epoch 40 indicates that normalization enabled more efficient learning and optimization. Overall, standard scaling not only improved the model's accuracy but also enhanced its training stability, convergence behavior, and overall predictive performance.

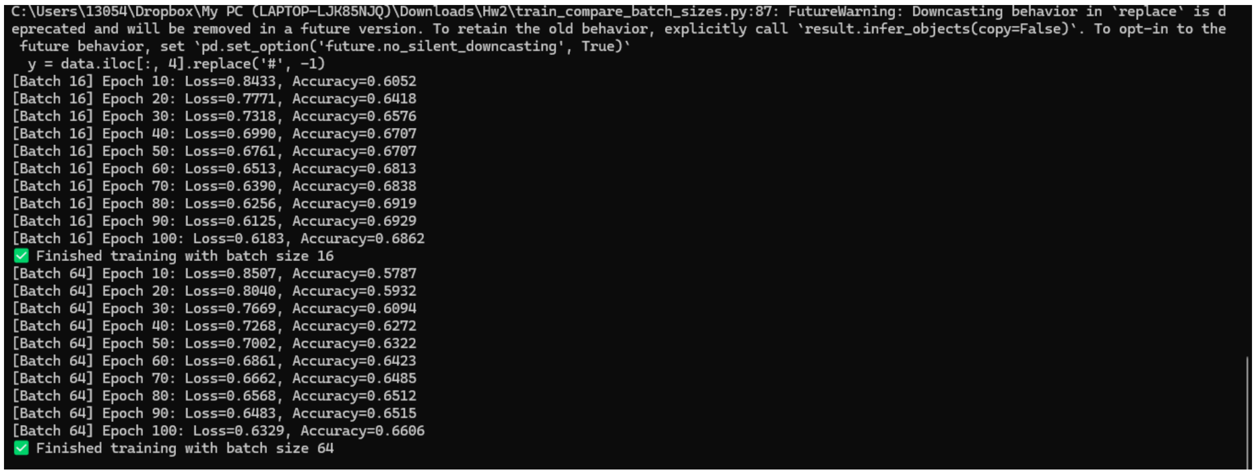


| un ↑ | Smoothed | Value | Step | R |
|---|---|---|---|---|
| ith_normalization | 0.5394 | 0.5428 | 99 | 3. |
| ithout_normalization | 0.5284 | 0.5303 | 99 | 3. |



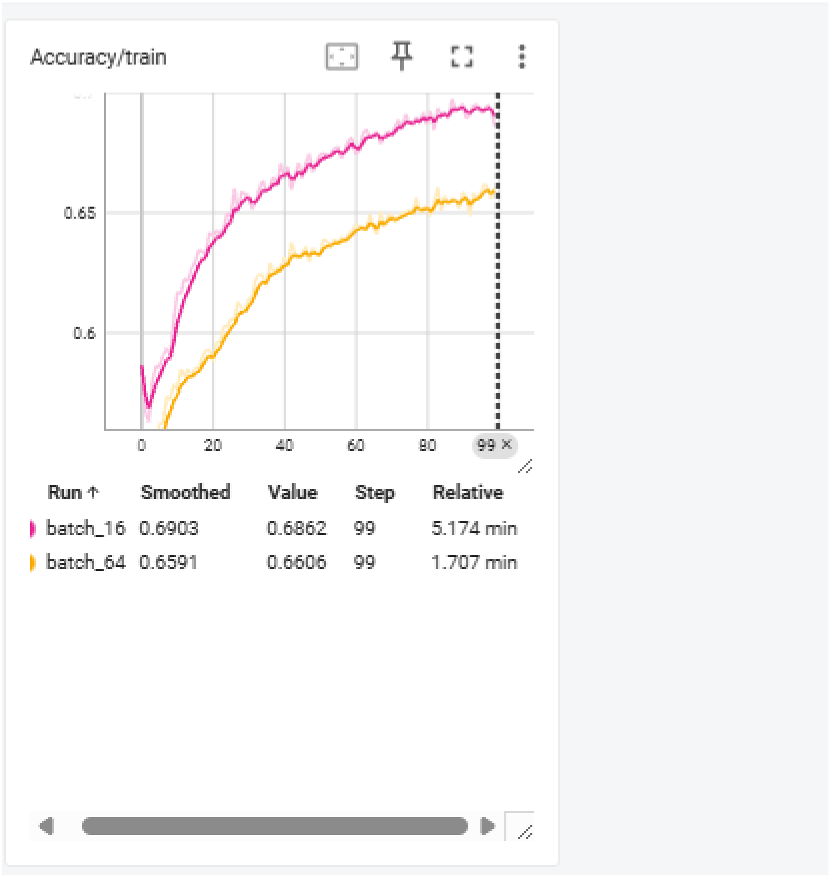
| | Smoothed | Value | Step | Relative |
|---|---|---|---|---|
| alization | 0.9339 | 0.9311 | 99 | 3.215 sec |
| rmalization | 0.97 | 0.9669 | 99 | 3.257 sec |

2-

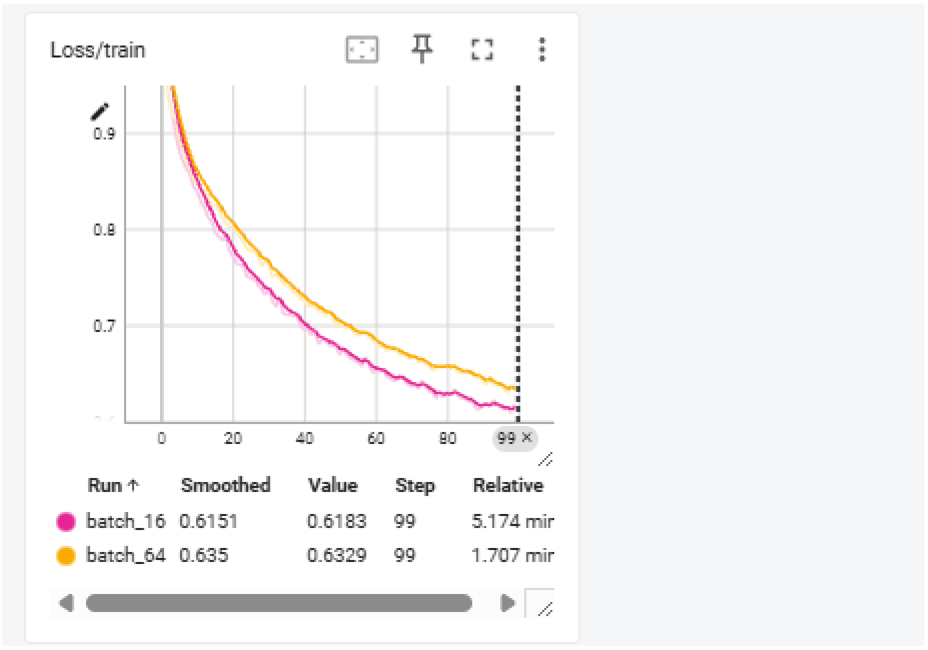


```
C:\Users\13054\Dropbox\My PC (LAPTOP-LJK85NJQ)\Downloads\Hw2\train_compare_batch_sizes.py:87: FutureWarning: Downcasting behavior in `replace` is d
eprecated and will be removed in a future version. To retain the old behavior, explicitly call `result.infer_objects(copy=False)`. To opt-in to the
 future behavior, set `pd.set_option('future.no_silent_downcasting', True)`
  y = data.iloc[:, 4].replace('#', -1)
[Batch 16] Epoch 10: Loss=0.8433, Accuracy=0.6052
[Batch 16] Epoch 20: Loss=0.7771, Accuracy=0.6418
[Batch 16] Epoch 30: Loss=0.7318, Accuracy=0.6576
[Batch 16] Epoch 40: Loss=0.6990, Accuracy=0.6707
[Batch 16] Epoch 50: Loss=0.6761, Accuracy=0.6707
[Batch 16] Epoch 60: Loss=0.6513, Accuracy=0.6813
[Batch 16] Epoch 70: Loss=0.6390, Accuracy=0.6838
[Batch 16] Epoch 80: Loss=0.6256, Accuracy=0.6919
[Batch 16] Epoch 90: Loss=0.6125, Accuracy=0.6929
[Batch 16] Epoch 100: Loss=0.6183, Accuracy=0.6862
✅ Finished training with batch size 16
[Batch 64] Epoch 10: Loss=0.8507, Accuracy=0.5787
[Batch 64] Epoch 20: Loss=0.8040, Accuracy=0.5932
[Batch 64] Epoch 30: Loss=0.7669, Accuracy=0.6094
[Batch 64] Epoch 40: Loss=0.7268, Accuracy=0.6272
[Batch 64] Epoch 50: Loss=0.7002, Accuracy=0.6322
[Batch 64] Epoch 60: Loss=0.6861, Accuracy=0.6423
[Batch 64] Epoch 70: Loss=0.6662, Accuracy=0.6485
[Batch 64] Epoch 80: Loss=0.6568, Accuracy=0.6512
[Batch 64] Epoch 90: Loss=0.6483, Accuracy=0.6515
[Batch 64] Epoch 100: Loss=0.6329, Accuracy=0.6606
✅ Finished training with batch size 64
```

I evaluated how mini-batch size affects training performance by comparing batch sizes of 16 and 64. The model trained with a batch size of 16 reached a slightly higher final accuracy (68.6%) compared to the model trained with a batch size of 64 (66.0%). Additionally, the final training loss was marginally lower with batch size 16 (0.6183 vs 0.6324). These results suggest that using a smaller batch size may allow the model to learn finer-grained gradients, leading to better convergence and generalization. However, smaller batch sizes may also increase training time, so the optimal batch size should balance both performance and efficiency depending on the application.

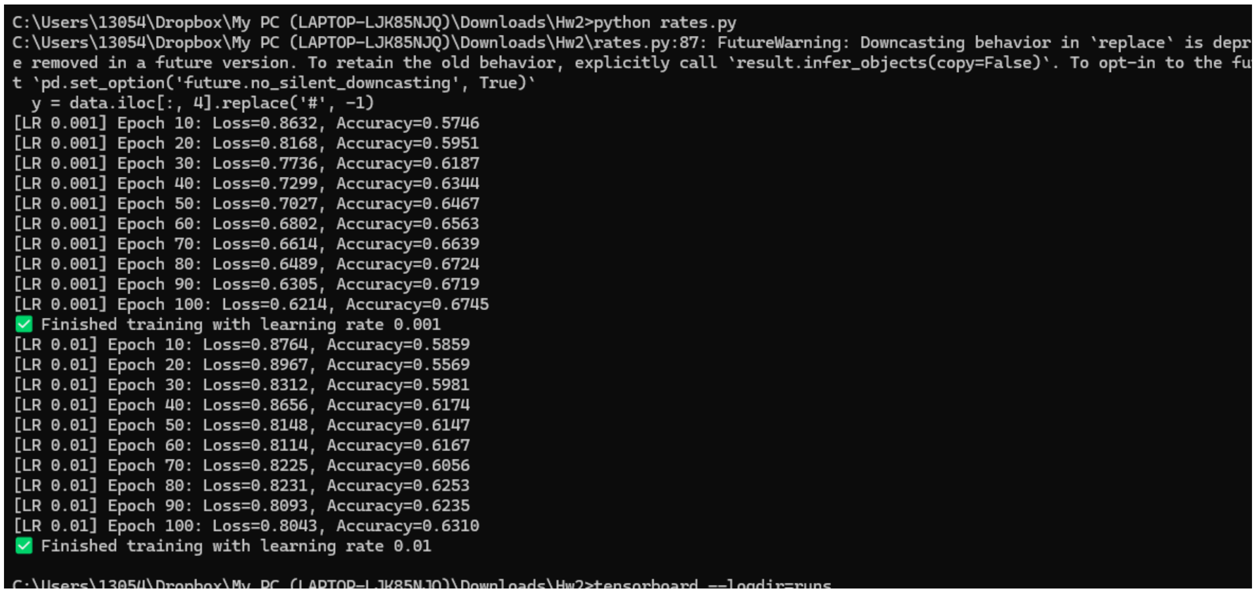


| Run ↑ | Smoothed | Value | Step | Relative |
|---|---|---|---|---|
| batch_16 | 0.6903 | 0.6862 | 99 | 5.174 min |
| batch_64 | 0.6591 | 0.6606 | 99 | 1.707 min |



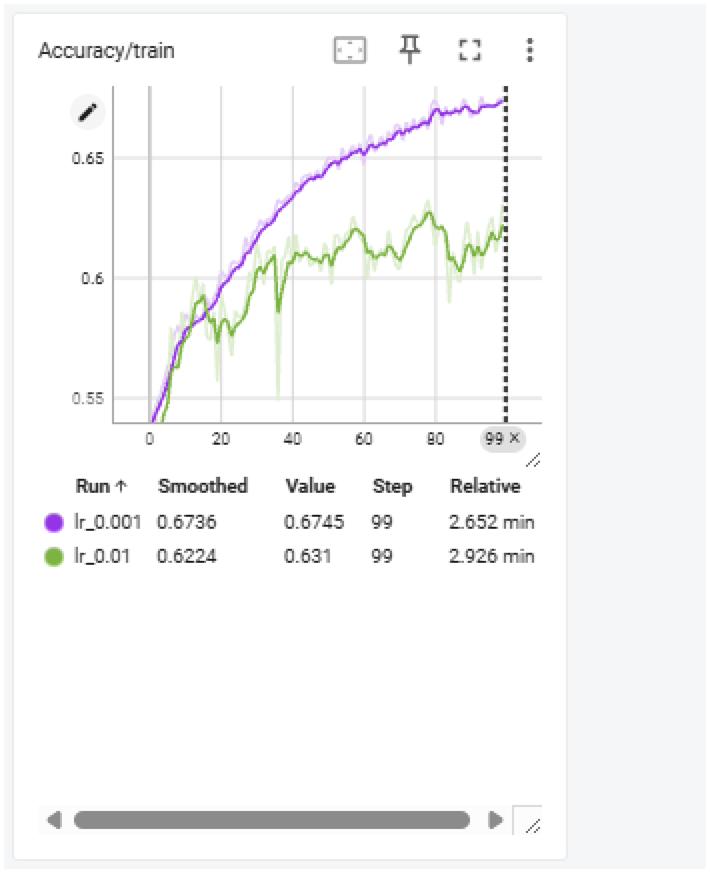
| Run ↑ | Smoothed | Value | Step | Relative |
|---|---|---|---|---|
| ● batch_16 | 0.6151 | 0.6183 | 99 | 5.174 mir |
| ● batch_64 | 0.635 | 0.6329 | 99 | 1.707 mir |

3-

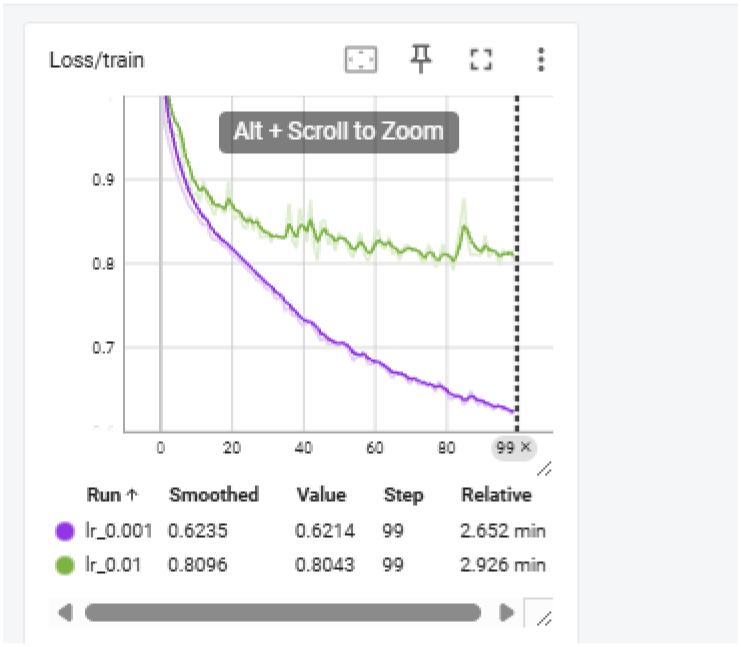


```
C:\Users\13054\Dropbox\My PC (LAPTOP-LJK85NJQ)\Downloads\Hw2>python rates.py
C:\Users\13054\Dropbox\My PC (LAPTOP-LJK85NJQ)\Downloads\Hw2\rates.py:87: FutureWarning: Downcasting behavior in `replace` is depr
e removed in a future version. To retain the old behavior, explicitly call `result.infer_objects(copy=False)`. To opt-in to the fu
t `pd.set_option('future.no_silent_downcasting', True)`
  y = data.iloc[:, 4].replace('#', -1)
[LR 0.001] Epoch 10: Loss=0.8632, Accuracy=0.5746
[LR 0.001] Epoch 20: Loss=0.8168, Accuracy=0.5951
[LR 0.001] Epoch 30: Loss=0.7736, Accuracy=0.6187
[LR 0.001] Epoch 40: Loss=0.7299, Accuracy=0.6344
[LR 0.001] Epoch 50: Loss=0.7027, Accuracy=0.6467
[LR 0.001] Epoch 60: Loss=0.6802, Accuracy=0.6563
[LR 0.001] Epoch 70: Loss=0.6614, Accuracy=0.6639
[LR 0.001] Epoch 80: Loss=0.6489, Accuracy=0.6724
[LR 0.001] Epoch 90: Loss=0.6305, Accuracy=0.6719
[LR 0.001] Epoch 100: Loss=0.6214, Accuracy=0.6745
✅ Finished training with learning rate 0.001
[LR 0.01] Epoch 10: Loss=0.8764, Accuracy=0.5859
[LR 0.01] Epoch 20: Loss=0.8967, Accuracy=0.5569
[LR 0.01] Epoch 30: Loss=0.8312, Accuracy=0.5981
[LR 0.01] Epoch 40: Loss=0.8656, Accuracy=0.6174
[LR 0.01] Epoch 50: Loss=0.8148, Accuracy=0.6147
[LR 0.01] Epoch 60: Loss=0.8114, Accuracy=0.6167
[LR 0.01] Epoch 70: Loss=0.8225, Accuracy=0.6056
[LR 0.01] Epoch 80: Loss=0.8231, Accuracy=0.6253
[LR 0.01] Epoch 90: Loss=0.8093, Accuracy=0.6235
[LR 0.01] Epoch 100: Loss=0.8043, Accuracy=0.6310
✅ Finished training with learning rate 0.01

C:\Users\13054\Dropbox\My PC (LAPTOP-LJK85NJQ)\Downloads\Hw2>tensorboard --logdir=runs
```



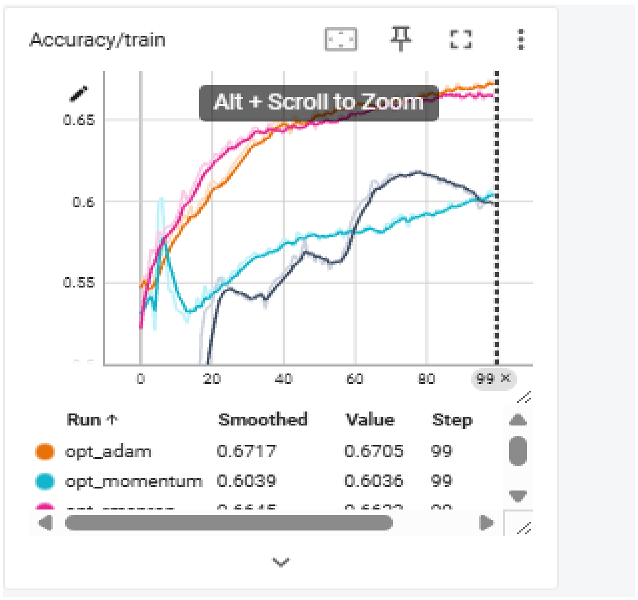
| | Run ↑ | Smoothed | Value | Step | Relative |
|---|---|---|---|---|---|
| ● | lr_0.001 | 0.6736 | 0.6745 | 99 | 2.652 min |
| ● | lr_0.01 | 0.6224 | 0.631 | 99 | 2.926 min |



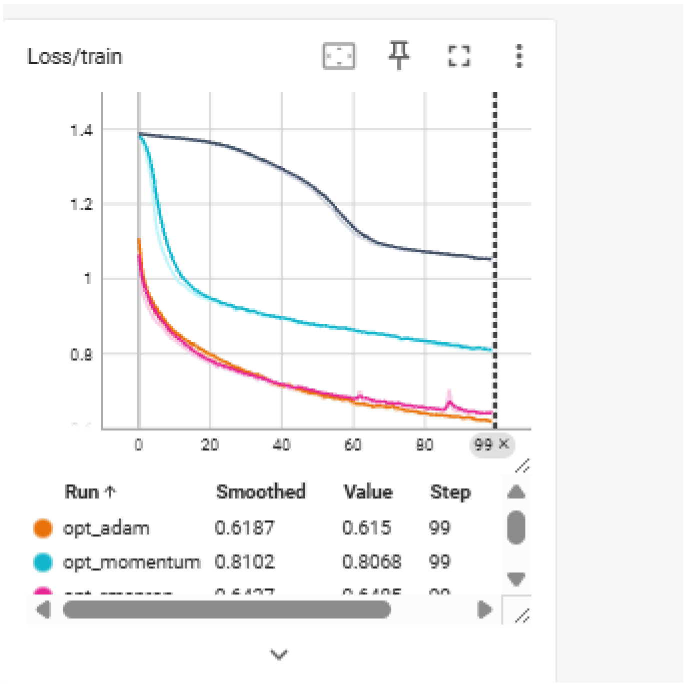
| Run | Smoothed | Value | Step | Relative |
|---|---|---|---|---|
| ● lr_0.001 | 0.6235 | 0.6214 | 99 | 2.652 min |
| ● lr_0.01 | 0.8096 | 0.8043 | 99 | 2.926 min |

I also compared the training performance of a deep neural network using two different learning rates: 0.001 and 0.01. The model trained with a lower learning rate (0.001) achieved better results, with a final accuracy of 67.5% and a lower final loss of 0.6214. In contrast, the model trained with a higher learning rate (0.01) converged more quickly in the early epochs but plateaued at a lower final accuracy of 63.1% and a higher loss of 0.8043. These results highlight the importance of selecting an appropriate learning rate — while higher rates may speed up training, they risk overshooting the optimal solution, leading to suboptimal convergence. A learning rate of 0.001 offered a more stable and effective learning process for this task.

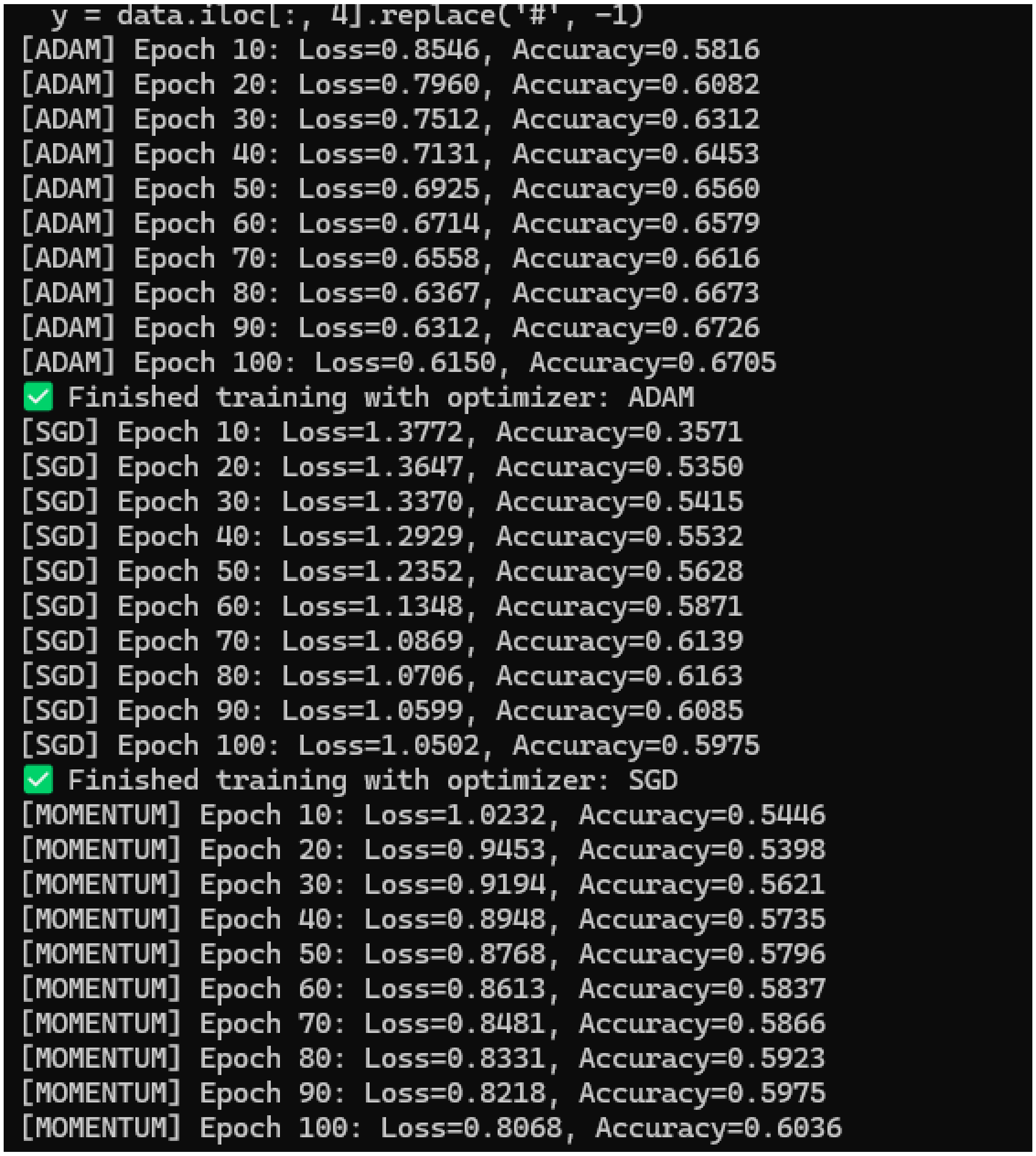
4-



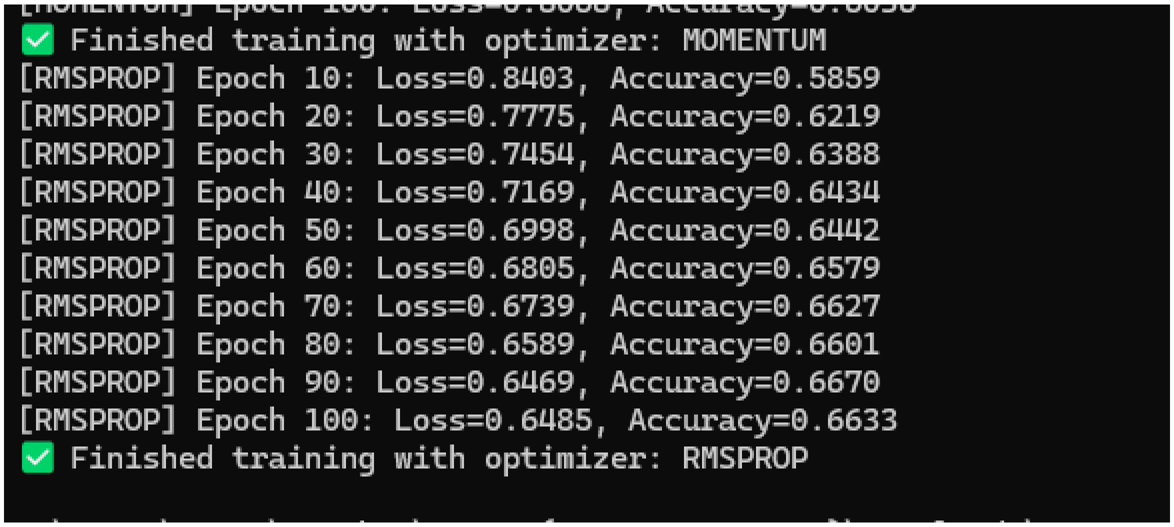
| Run ↑ | Smoothed | Value | Step |
|---|---|---|---|
| ● opt_adam | 0.6717 | 0.6705 | 99 |
| ● opt_momentum | 0.6039 | 0.6036 | 99 |



| Run ↑ | Smoothed | Value | Step |
|---|---|---|---|
| ● opt_adam | 0.6187 | 0.615 | 99 |
| ● opt_momentum | 0.8102 | 0.8068 | 99 |



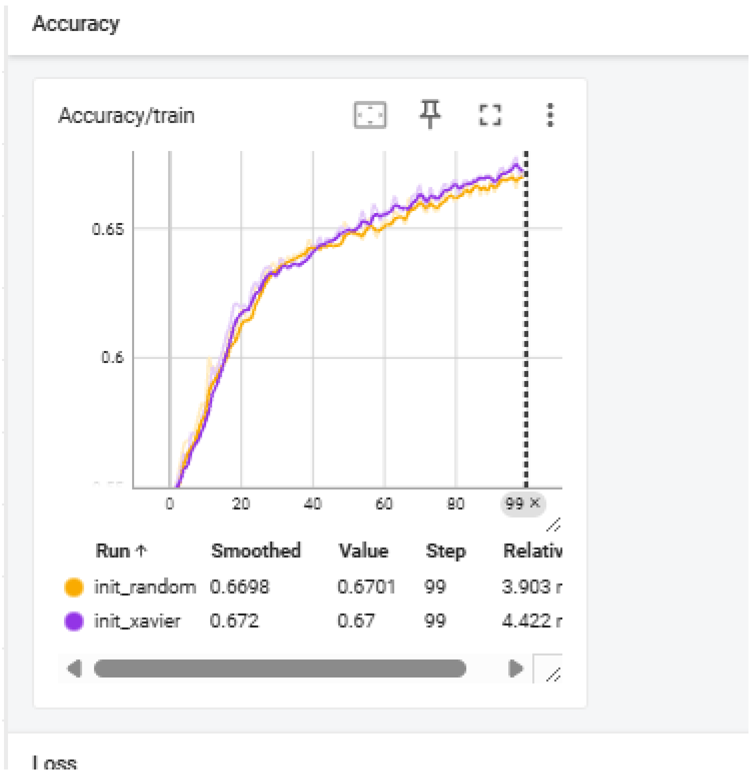
```
  y = data.iloc[:, 4].replace('#', -1)
[ADAM] Epoch 10: Loss=0.8546, Accuracy=0.5816
[ADAM] Epoch 20: Loss=0.7960, Accuracy=0.6082
[ADAM] Epoch 30: Loss=0.7512, Accuracy=0.6312
[ADAM] Epoch 40: Loss=0.7131, Accuracy=0.6453
[ADAM] Epoch 50: Loss=0.6925, Accuracy=0.6560
[ADAM] Epoch 60: Loss=0.6714, Accuracy=0.6579
[ADAM] Epoch 70: Loss=0.6558, Accuracy=0.6616
[ADAM] Epoch 80: Loss=0.6367, Accuracy=0.6673
[ADAM] Epoch 90: Loss=0.6312, Accuracy=0.6726
[ADAM] Epoch 100: Loss=0.6150, Accuracy=0.6705
✅ Finished training with optimizer: ADAM
[SGD] Epoch 10: Loss=1.3772, Accuracy=0.3571
[SGD] Epoch 20: Loss=1.3647, Accuracy=0.5350
[SGD] Epoch 30: Loss=1.3370, Accuracy=0.5415
[SGD] Epoch 40: Loss=1.2929, Accuracy=0.5532
[SGD] Epoch 50: Loss=1.2352, Accuracy=0.5628
[SGD] Epoch 60: Loss=1.1348, Accuracy=0.5871
[SGD] Epoch 70: Loss=1.0869, Accuracy=0.6139
[SGD] Epoch 80: Loss=1.0706, Accuracy=0.6163
[SGD] Epoch 90: Loss=1.0599, Accuracy=0.6085
[SGD] Epoch 100: Loss=1.0502, Accuracy=0.5975
✅ Finished training with optimizer: SGD
[MOMENTUM] Epoch 10: Loss=1.0232, Accuracy=0.5446
[MOMENTUM] Epoch 20: Loss=0.9453, Accuracy=0.5398
[MOMENTUM] Epoch 30: Loss=0.9194, Accuracy=0.5621
[MOMENTUM] Epoch 40: Loss=0.8948, Accuracy=0.5735
[MOMENTUM] Epoch 50: Loss=0.8768, Accuracy=0.5796
[MOMENTUM] Epoch 60: Loss=0.8613, Accuracy=0.5837
[MOMENTUM] Epoch 70: Loss=0.8481, Accuracy=0.5866
[MOMENTUM] Epoch 80: Loss=0.8331, Accuracy=0.5923
[MOMENTUM] Epoch 90: Loss=0.8218, Accuracy=0.5975
[MOMENTUM] Epoch 100: Loss=0.8068, Accuracy=0.6036
```



```
✅ Finished training with optimizer: MOMENTUM
[RMSPROP] Epoch 10: Loss=0.8403, Accuracy=0.5859
[RMSPROP] Epoch 20: Loss=0.7775, Accuracy=0.6219
[RMSPROP] Epoch 30: Loss=0.7454, Accuracy=0.6388
[RMSPROP] Epoch 40: Loss=0.7169, Accuracy=0.6434
[RMSPROP] Epoch 50: Loss=0.6998, Accuracy=0.6442
[RMSPROP] Epoch 60: Loss=0.6805, Accuracy=0.6579
[RMSPROP] Epoch 70: Loss=0.6739, Accuracy=0.6627
[RMSPROP] Epoch 80: Loss=0.6589, Accuracy=0.6601
[RMSPROP] Epoch 90: Loss=0.6469, Accuracy=0.6670
[RMSPROP] Epoch 100: Loss=0.6485, Accuracy=0.6633
✅ Finished training with optimizer: RMSPROP
```

In this experiment, we compared the training performance of four common optimizers: Adam, RMSProp, SGD with Momentum, and basic SGD. The model trained with the Adam optimizer achieved the best results, finishing with the highest accuracy (67.1%) and lowest loss (0.6150). RMSProp followed closely with 66.3% accuracy and a slightly higher loss of 0.6485, showing stable convergence. In contrast, SGD with Momentum reached 60.4% accuracy and a loss of 0.8068, while basic SGD lagged behind with a final accuracy of 59.8% and the highest loss (1.0502). These results reinforce the efficiency and robustness of adaptive optimizers like Adam and RMSProp for this dataset, as they tend to adjust learning rates dynamically, resulting in faster and more effective convergence compared to fixed-rate methods like SGD.

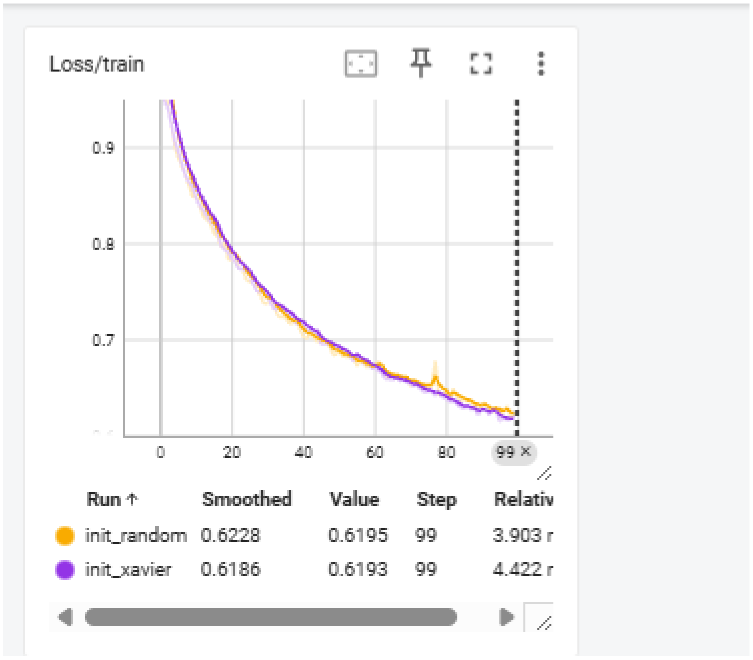
5-


Accuracy

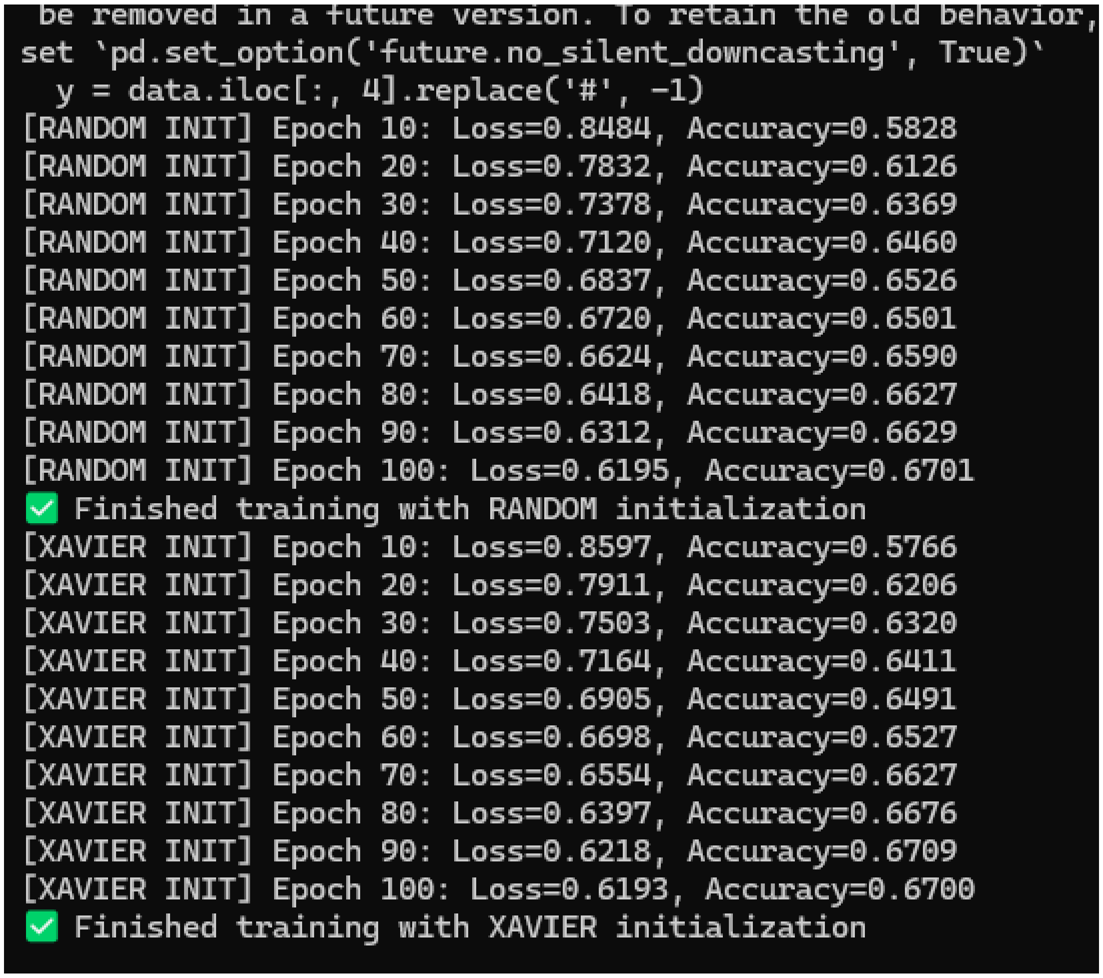
Accuracy/train

| Run ↑ | Smoothed | Value | Step | Relativ |
|---|---|---|---|---|
| ● init_random | 0.6698 | 0.6701 | 99 | 3.903 r |
| ● init_xavier | 0.672 | 0.67 | 99 | 4.422 r |

Loss



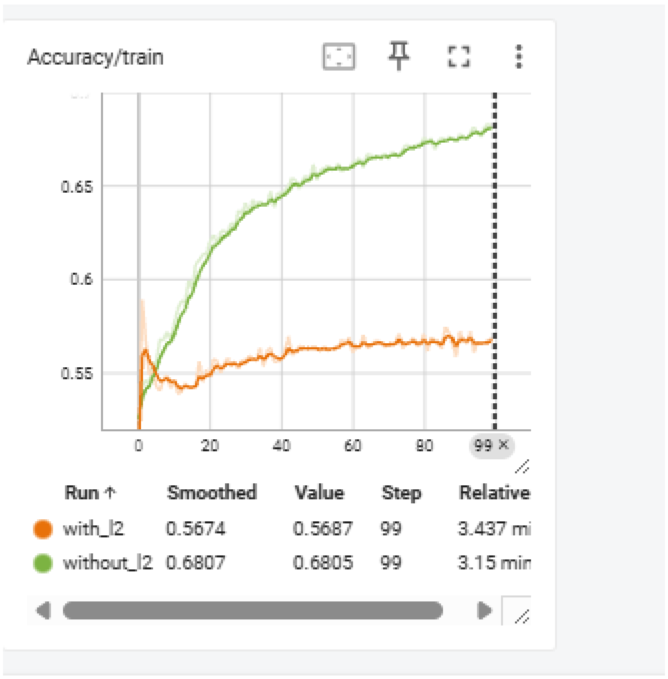
| Run ↑ | Smoothed | Value | Step | Relativ |
|---|---|---|---|---|
| ● init_random | 0.6228 | 0.6195 | 99 | 3.903 r |
| ● init_xavier | 0.6186 | 0.6193 | 99 | 4.422 r |



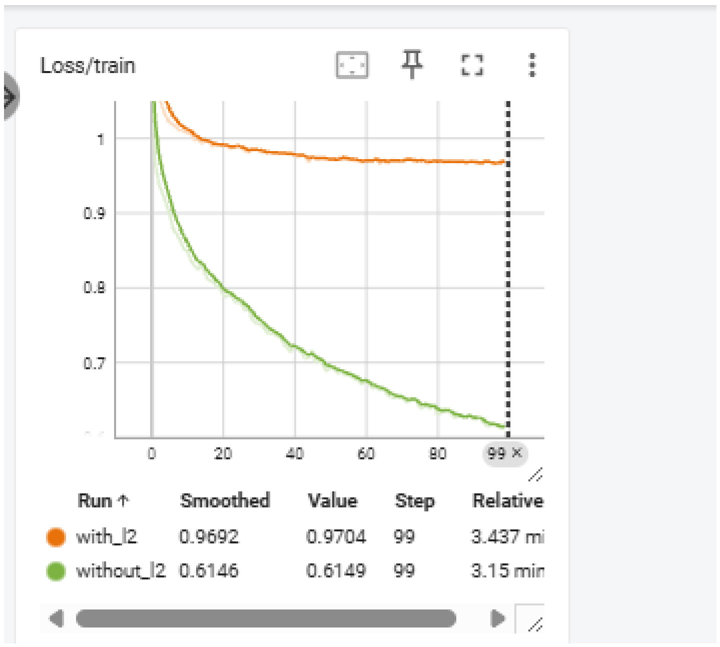
```
 be removed in a future version. To retain the old behavior,
set `pd.set_option('future.no_silent_downcasting', True)`
  y = data.iloc[:, 4].replace('#', -1)
[RANDOM INIT] Epoch 10: Loss=0.8484, Accuracy=0.5828
[RANDOM INIT] Epoch 20: Loss=0.7832, Accuracy=0.6126
[RANDOM INIT] Epoch 30: Loss=0.7378, Accuracy=0.6369
[RANDOM INIT] Epoch 40: Loss=0.7120, Accuracy=0.6460
[RANDOM INIT] Epoch 50: Loss=0.6837, Accuracy=0.6526
[RANDOM INIT] Epoch 60: Loss=0.6720, Accuracy=0.6501
[RANDOM INIT] Epoch 70: Loss=0.6624, Accuracy=0.6590
[RANDOM INIT] Epoch 80: Loss=0.6418, Accuracy=0.6627
[RANDOM INIT] Epoch 90: Loss=0.6312, Accuracy=0.6629
[RANDOM INIT] Epoch 100: Loss=0.6195, Accuracy=0.6701
✅ Finished training with RANDOM initialization
[XAVIER INIT] Epoch 10: Loss=0.8597, Accuracy=0.5766
[XAVIER INIT] Epoch 20: Loss=0.7911, Accuracy=0.6206
[XAVIER INIT] Epoch 30: Loss=0.7503, Accuracy=0.6320
[XAVIER INIT] Epoch 40: Loss=0.7164, Accuracy=0.6411
[XAVIER INIT] Epoch 50: Loss=0.6905, Accuracy=0.6491
[XAVIER INIT] Epoch 60: Loss=0.6698, Accuracy=0.6527
[XAVIER INIT] Epoch 70: Loss=0.6554, Accuracy=0.6627
[XAVIER INIT] Epoch 80: Loss=0.6397, Accuracy=0.6676
[XAVIER INIT] Epoch 90: Loss=0.6218, Accuracy=0.6709
[XAVIER INIT] Epoch 100: Loss=0.6193, Accuracy=0.6700
✅ Finished training with XAVIER initialization
```

In this experiment, I compared two different weight initialization strategies for a deep neural network: random initialization (PyTorch default) and Xavier (Glorot) initialization. Both models converged similarly, reaching the same final accuracy of 67.0% after 100 epochs. The difference in final loss was negligible (0.6195 vs 0.6193), suggesting that for this dataset and architecture, both initialization methods are equally effective. While Xavier initialization is designed to maintain signal variance throughout layers and often improves training stability, its impact in this case was minimal — likely due to the consistent architecture and use of ReLU activations. This result shows that, although important in some settings, initialization method may have limited influence on performance when the model and data are well-behaved.

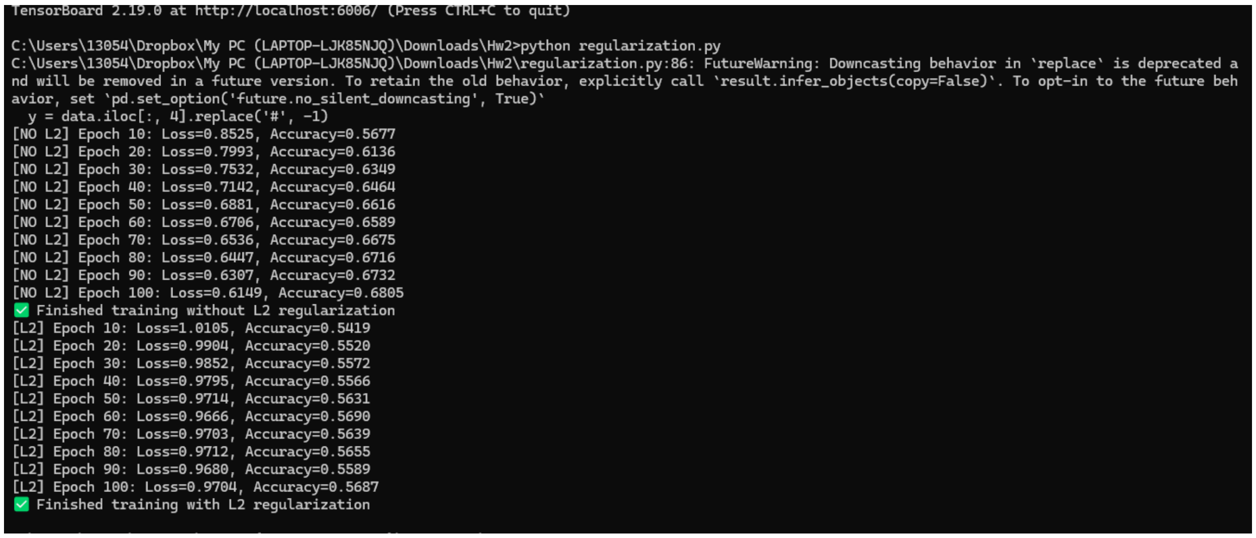
6-



| | Run ↑ | Smoothed | Value | Step | Relative |
|---|---|---|---|---|---|
| ● | with_l2 | 0.5674 | 0.5687 | 99 | 3.437 mi |
| ● | without_l2 | 0.6807 | 0.6805 | 99 | 3.15 mi |



| Run ↑ | Smoothed | Value | Step | Relative |
|---|---|---|---|---|
| ● with_l2 | 0.9692 | 0.9704 | 99 | 3.437 mi |
| ● without_l2 | 0.6146 | 0.6149 | 99 | 3.15 min |



```
TensorBoard 2.19.0 at http://localhost:6006/ (Press CTRL+C to quit)

C:\Users\13054\Dropbox\My PC (LAPTOP-LJK85NJQ)\Downloads\Hw2>python regularization.py
C:\Users\13054\Dropbox\My PC (LAPTOP-LJK85NJQ)\Downloads\Hw2\regularization.py:86: FutureWarning: Downcasting behavior in `replace` is deprecated a
nd will be removed in a future version. To retain the old behavior, explicitly call `result.infer_objects(copy=False)`. To opt-in to the future beh
avior, set `pd.set_option('future.no_silent_downcasting', True)`
  y = data.iloc[:, 4].replace('#', -1)
[NO L2] Epoch 10: Loss=0.8525, Accuracy=0.5677
[NO L2] Epoch 20: Loss=0.7993, Accuracy=0.6136
[NO L2] Epoch 30: Loss=0.7532, Accuracy=0.6349
[NO L2] Epoch 40: Loss=0.7142, Accuracy=0.6464
[NO L2] Epoch 50: Loss=0.6881, Accuracy=0.6616
[NO L2] Epoch 60: Loss=0.6706, Accuracy=0.6589
[NO L2] Epoch 70: Loss=0.6536, Accuracy=0.6675
[NO L2] Epoch 80: Loss=0.6447, Accuracy=0.6716
[NO L2] Epoch 90: Loss=0.6307, Accuracy=0.6732
[NO L2] Epoch 100: Loss=0.6149, Accuracy=0.6805
✅ Finished training without L2 regularization
[L2] Epoch 10: Loss=1.0105, Accuracy=0.5419
[L2] Epoch 20: Loss=0.9904, Accuracy=0.5520
[L2] Epoch 30: Loss=0.9852, Accuracy=0.5572
[L2] Epoch 40: Loss=0.9795, Accuracy=0.5566
[L2] Epoch 50: Loss=0.9714, Accuracy=0.5631
[L2] Epoch 60: Loss=0.9666, Accuracy=0.5690
[L2] Epoch 70: Loss=0.9703, Accuracy=0.5639
[L2] Epoch 80: Loss=0.9712, Accuracy=0.5655
[L2] Epoch 90: Loss=0.9680, Accuracy=0.5589
[L2] Epoch 100: Loss=0.9704, Accuracy=0.5687
✅ Finished training with L2 regularization
```

This experiment compared training a deep neural network with and without L2 regularization (also known as weight decay). The model trained without L2 regularization performed significantly better, reaching a final accuracy of 68.0% and a loss of 0.6149. In contrast, applying L2 regularization with a weight decay of 0.01 caused the model to underperform, ending with a lower accuracy of 56.8% and a noticeably higher loss of 0.9704.

L2 regularization is typically beneficial in reducing overfitting, especially on noisy or complex data. However, in this case, the regularization may have been too aggressive, preventing the model from learning important patterns in the data. This suggests that while L2 can be helpful in many contexts, its strength (λ) must be tuned carefully, and its usefulness depends heavily on the dataset's characteristics.

## Conclusion

After finishing the experiment, I could learn that the impact of various training strategies on the performance of a deep neural network. This included normalization, batch size, learning rate, optimizer choice, weight initialization, and L2 regularization. Each technique had its own influence, but some proved significantly more effective than others.

Normalization clearly improved both accuracy and training stability. It resulted in smoother convergence and slightly better generalization, making it a highly recommended preprocessing step.

Among the optimizers, Adam consistently outperformed others, achieving the best accuracy and lowest loss. Its adaptive learning rate mechanism allows faster and more stable convergence, especially on datasets like this one.

In terms of batch size, using a smaller batch size (e.g., 16) helped the model converge faster and led to slightly better performance. Smaller batches introduce useful gradient noise that often improves generalization.

For learning rates, a lower learning rate (0.001) offered more reliable and gradual learning. Higher learning rates showed instability and poorer convergence.

Both weight initialization methods — random (Kaiming) and Xavier — performed similarly, but Xavier maintained variance more effectively and may be better suited for deeper or more sensitive architectures.

However, L2 regularization did not help in this case. When applied with a weight decay of 0.01, it led to worse accuracy and higher loss, suggesting the model didn’t suffer from overfitting, and the regularization term penalized important weights.

Better implementation will be to use a setup that combines several key strategies for optimal training. These include applying standard normalization using StandardScaler, utilizing the Adam optimizer for its adaptive learning capabilities, and training with a smaller batch size of around 16, which promotes better generalization. A learning rate of 0.001 provided stable and effective convergence, while Xavier (Glorot) weight initialization helped maintain variance throughout the network layers. In contrast, L2 regularization (with $\lambda = 0.01$) negatively impacted performance and should be avoided unless carefully tuned. Altogether, this configuration is most likely to yield the best balance between accuracy, convergence speed, and training stability.

I will definitely keep al that I have learned in this homework for the final project.