499-hw1-part3

Batuhan Karaca 2310191

November 2022

List of plots

1 Introduction

Note that the terms *combination* and *configuration* are used interchangeably.

This report aims to analyze the results obtained by a model on MNIST dataset using different combination of hyperparameters. The methodology is as follows. For the preprocessing stage, the pixel values are normalized to the values in the range [0,1]. Weighted pixel values would give another value in the same range. Tanh or sigmoid would output a value in a smaller subset of range $[0, \sim 0.7]$. In this manner, the output values won't explode avoiding the precision loss. The data is sampled using a shuffled sample (`shuffle=True` in `DataLoader`) to achieve more generalized results, avoiding overfitting. In the first step, with a grid parameter search the model is trained with each configuration on a train set, tested on a validation set. In the second step the configuration with the highest accuracy score is tested with another independent test set. This time the train and validation sets are combined into a new train set. The test set needs to be independent in order to avoid overfitting.

In the next section (section 2), the tested hyperparameters are introduced with the decisions made for the values preferred for each. Then, the results for the first step are analyzed for the parameter. There comes the result of the second step after that (section 3). In the Appendix section, there is a loss and accuracy graph for each combination. Each graph shows the epochs in its x-axis whereas the corresponding loss or accuracy value in its y-axis. The combinations can be seen in the figure captions. There are 36 combinations in total, as 3, 1, 1, 2, 3 and 2 different values are tested for learning rate, number of epochs, depth, breadth, activation function and batch size respectively.

2 First step

GPU is used when training the network. The hyperparameters are chosen accordingly. There are 6 hyperparameters tested in this experiment.

First and foremost, 3 different values of learning rates are evaluated, 0.0001, 0.01 and 1. The values are purely a matter of choice that no reasoning was made. As the results show, 0.0001 was so small that not enough advancements have been made in the given epochs. On the other hand, the value 1 was so large that the updates were unstable. The model is moving so uncontrollably and rapidly in the loss space. This divergent behavior resulted in model not converging, possibly needing more training epochs. Thus, the model performed poorly for this value. 0.01 gave the best results among the values.

Secondly, another main parameter, number of epochs, comes into consideration. The only value tested was 30. Reducing the execution time is a reason for why a single value is preferred. Furthermore, as seen in the plots (Appendix section), 30 is redundant for some of the combinations. The model skyrockets to about 99% in as small as 10 epochs for the combination 0.01, 30, 3, 128, any activation function, 463. We can say that 30 was a sufficient value for this task. Also note that as opposed to the theory that every batch of information is sampled in one epoch, we are iterating over every batches in a single epoch. Therefore, the number of epochs is relatively small compared to hundreds and thousands. Note that overfitting starts to be observed when the test loss is

minimum, or test accuracy is maximum. After that point, we would see the test loss or accuracy diverging from the train loss or accuracy, and the model starts to perform better when training, more poorly when testing. However, SGD or Mini-batch gradient descent's wavy behaviour makes finding this point more difficult. We can avoid overfitting using a regularization technique such as dropout, or batch normalization. Regularization was not used in the experiment.

Note that larger or smaller number of parameters do not necessarily increase the accuracy. Insufficient number of parameters would underfit needing more training time, whereas redundant parameters may overfit the model. Nevertheless, on an interval the accuracy keeps climbing as there are more parameters that can fit. To avoid overfitting, we need to set our breadth and depth values accordingly.

Thirdly, depth parameter (Number of hidden parameters + 1) is considered. Larger numbers are not considered to reduce the execution time and avoid overfitting. Compared to the number of parameters in a layer, adding layers requires sequential computation, hence more matrix multiplications, slowing down the execution. It is observed in the results that the number 3 is sufficient for this task. The choice is not purely coincidential as I came across an implementation successful with the MNIST dataset using the number 3. The video showing the implementation is in the YouTube link below; however, it is not the original source. The original author is Mike Cohen, giving the Udemy course *A deep understanding of deep learning (with Python intro)* [1] .
`https://www.youtube.com/watch?v=RLdxbvMZTsg`
Numbers 16 and 128 were chosen for the breadth (number of hidden parameters) parameter. I wanted to see if there is a difference between the values as small as 16 and as large as 128, and if 16 would perform well. I did not prefer to use larger values. Large parameters may mean more resources on the RAM (and VRAM for the GPU). Generally, increasing breadth value increases the accuracy. Leaving other parameters equal, only increasing the breadth parameter gives small to large improvements in the accuracy results. However the model performs depending on the other parameters as well. For the configuration 0.01, 30, 3, 16, *Sigmoid*(); the model performed slightly worse than its counterpart with 128 nodes (about 97 to 99). Hence, depending on the other hyperparameters, the value 16 either underfits or does not underfit.

In addition to the Sigmoid and Tanh functions, the commonly used ReLU(Rectified linear unit) is used. It solves the vanishing gradient and precision loss problems of both functions, and computationally less expensive. However, the results does not signify this. The reason could be that the vanishing problem is more visible with deeper networks. Because in the backpropagation step, as gradient is getting smaller, deeper networks have their earlier layers' gradients more likely to be zero. The value 3 seems to not let the observation of this phenomenon. Further note that if the pixel values are not normalized, ReLU should perform better than the other two, avoiding the preprocessing stage, reducing the computation cost more.

One last parameter, the batch size, is considered. Theoretically, the Stochas-

tic gradient descent (SGD) takes one sample at one epoch. This makes it more advantageous to the Batch gradient descent as it updates its parameters more frequently for the same amount of data. It can converge faster for large data. The instability of SGD may even make it find a better minima. However, the GPU cannot parallelize a single sample at a time and as our implementation requires, all the data should be scanned in a single epoch, slowing down the execution. The batch size is compromised with 463 samples per batch, removing 1 samples per batch (SGD) from the list for time concerns. The results show that for the batch size of 50004 (All the train data) the model converges monotonously, whereas for batch size of 463 the model is fluctuating. However, compared to the learning rate, the model is not diverging, instead converging and fluctuating around the minima. The results indicate that the batch size of 463 performs better than that of 50004. Mini-batch gradient descent still converges more earlier than the Batch gradient descent similar to SGD, also benefiting from the GPU.

Learning rate	Number of epochs	Depth	Breadth	Activation function	Batch size	Test accuracy
0.0001	30	3	16	$ReLU()$	463	$90.28077697753906 \pm 0.6855044960975647$
0.0001	30	3	16	$ReLU()$	50004	$13.48552131652832 \pm 1.7058191299438477$
0.0001	30	3	16	$Sigmoid()$	463	$65.70194244384766 \pm 4.243033409118652$
0.0001	30	3	16	$Sigmoid()$	50004	$10.742541313171387 \pm 0.889334499835968$
0.0001	30	3	16	$Tanh()$	463	$90.53995513916016 \pm 1.242717981338501$
0.0001	30	3	16	$Tanh()$	50004	$19.584632873535156 \pm 3.5564305782318115$
0.0001	30	3	128	$ReLU()$	463	$95.50755310058594 \pm 0.684922456741333$
0.0001	30	3	128	$ReLU()$	50004	$45.7717399597168 \pm 5.663984775543213$
0.0001	30	3	128	$Sigmoid()$	463	$91.5982666015625 \pm 0.5833438634872437$
0.0001	30	3	128	$Sigmoid()$	50004	$14.083272933959961 \pm 2.4819447994232178$
0.0001	30	3	128	$Tanh()$	463	$95.20518493652344 \pm 0.35586246848106384$
0.0001	30	3	128	$Tanh()$	50004	$62.70338821411133 \pm 2.0753366947174072$
0.01	30	3	16	$ReLU()$	463	$96.95463562011719 \pm 0.5730116367340088$
0.01	30	3	16	$ReLU()$	50004	$87.30882263183594 \pm 0.6131219267845154$
0.01	30	3	16	$Sigmoid()$	463	$97.86177062988281 \pm 0.430091917514801$
0.01	30	3	16	$Sigmoid()$	50004	$58.00975799560547 \pm 2.319347858428955$
0.01	30	3	16	$Tanh()$	463	$97.55940246582031 \pm 0.520574152469635$
0.01	30	3	16	$Tanh()$	50004	$87.34381866455078 \pm 0.49962684512138367$
0.01	30	3	128	$ReLU()$	463	$99.43843078613281 \pm 0.2292753905057907$
0.01	30	3	128	$ReLU()$	50004	$93.43932342529297 \pm 0.1511632800102234$
0.01	30	3	128	$Sigmoid()$	463	$99.95680236816406 \pm 0.08466586470603943$
0.01	30	3	128	$Sigmoid()$	50004	$88.25013732910156 \pm 0.18505258858203888$
0.01	30	3	128	$Tanh()$	463	$99.30885314941406 \pm 0.24277380108833313$
0.01	30	3	128	$Tanh()$	50004	$94.85101318359375 \pm 0.11484114825725555$
1	30	3	16	$ReLU()$	463	$9.740819931030273 \pm 0.8777143955230713$
1	30	3	16	$ReLU()$	50004	$10.315975189208984 \pm 0.3357134163379669$
1	30	3	16	$Sigmoid()$	463	$9.244059562683105 \pm 1.099924087524414$
1	30	3	16	$Sigmoid()$	50004	$11.169106483459473 \pm 1.9723540544509888$
1	30	3	16	$Tanh()$	463	$14.600430488586426 \pm 3.8624627590179443$
1	30	3	16	$Tanh()$	50004	$42.551597595214844 \pm 7.36905574798584$
1	30	3	128	$ReLU()$	463	$10.820734024047852 \pm 0.9327060580253601$
1	30	3	128	$ReLU()$	50004	$10.439764976501465 \pm 0.7678419947624207$
1	30	3	128	$Sigmoid()$	463	$9.438444137573242 \pm 0.7414588928222656$
1	30	3	128	$Sigmoid()$	50004	$9.80341625213623 \pm 0.4264678359031677$
1	30	3	128	$Tanh()$	463	$9.913606643676758 \pm 0.9241273999214172$
1	30	3	128	$Tanh()$	50004	$69.65682983398438 \pm 2.995284080505371$

Table 1: The results for each combination

3 Second step

The configuration with the highest accuracy score is 0.01, 30, 3, 128, $Sigmoid()$, 463. The new accuracy score is 100.0 ± 0.0.

4 Appendix

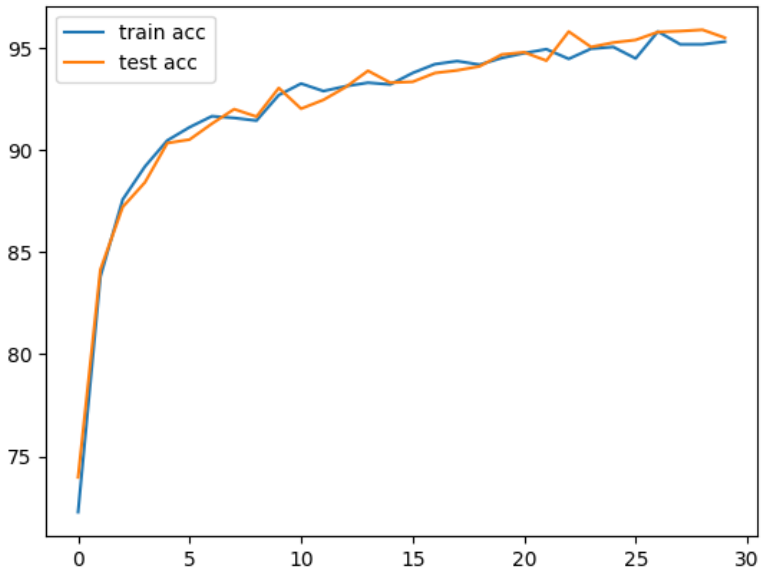

Figure 1: Accuracy values for 0.0001, 30, 3, 128, ReLU(), 463

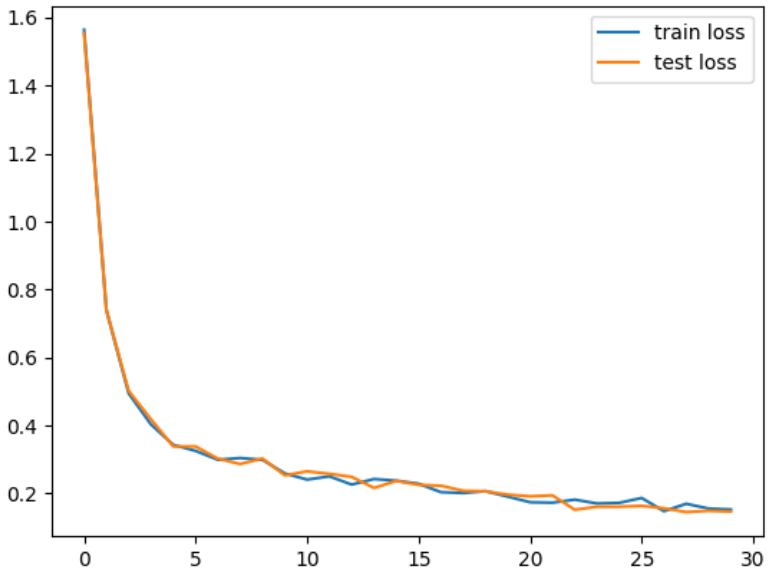

Figure 2: Loss values for 0.0001, 30, 3, 128, ReLU(), 463

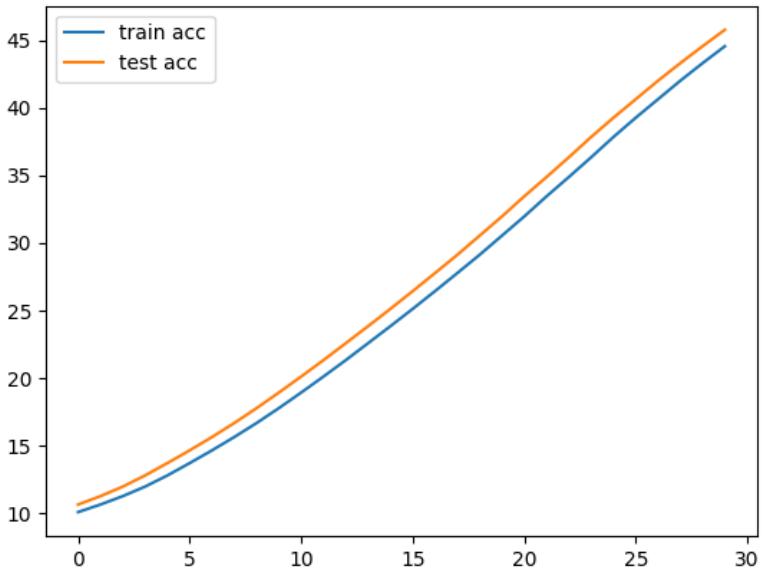

Figure 3: Accuracy values for 0.0001, 30, 3, 128, ReLU(), 50004

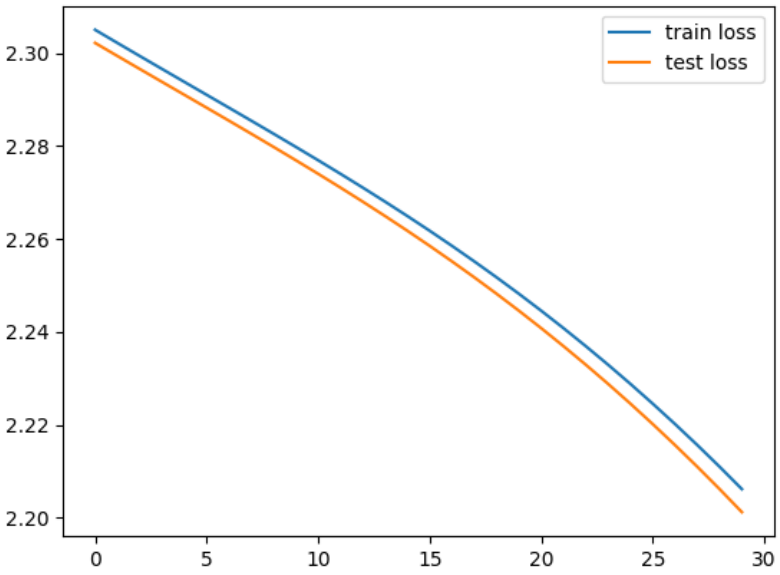

Figure 4: Loss values for 0.0001, 30, 3, 128, ReLU(), 50004

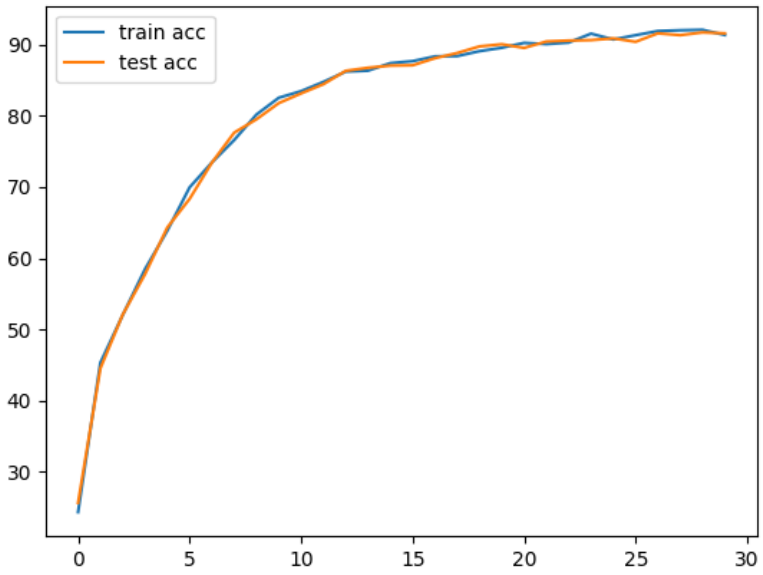

Figure 5: Accuracy values for 0.0001, 30, 3, 128, Sigmoid(), 463

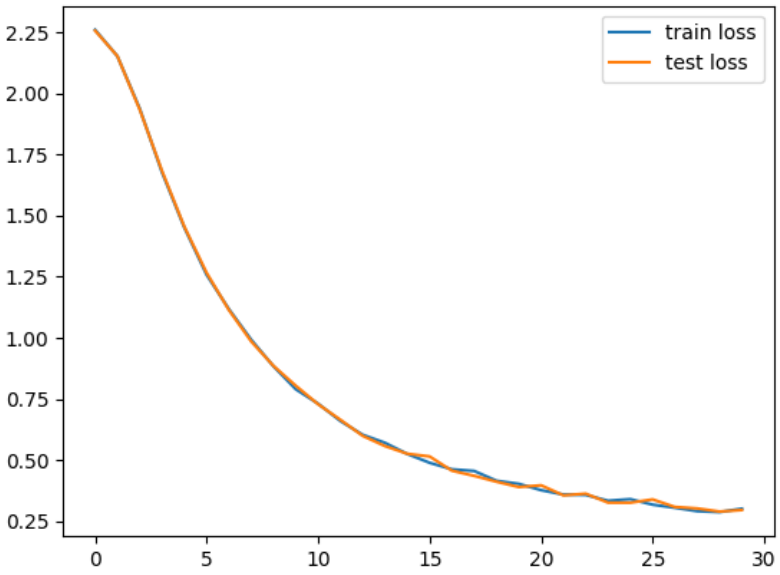

Figure 6: Loss values for 0.0001, 30, 3, 128, Sigmoid(), 463

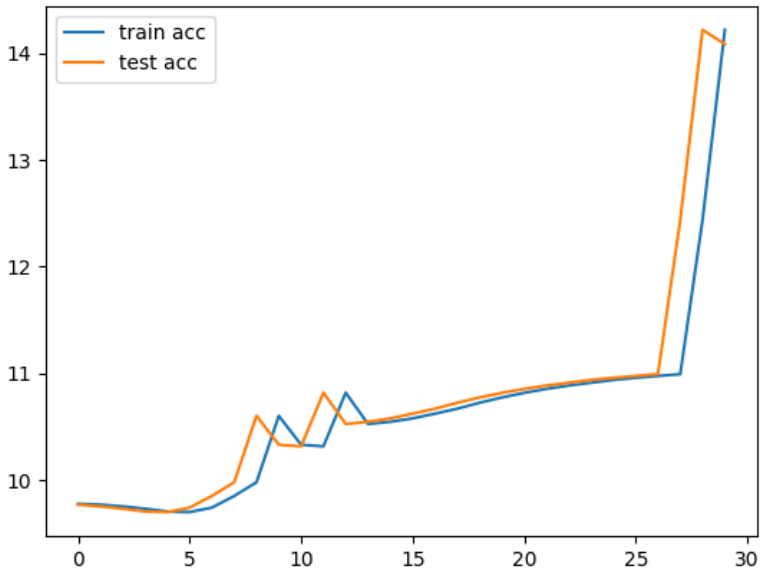

Figure 7: Accuracy values for 0.0001, 30, 3, 128, Sigmoid(), 50004

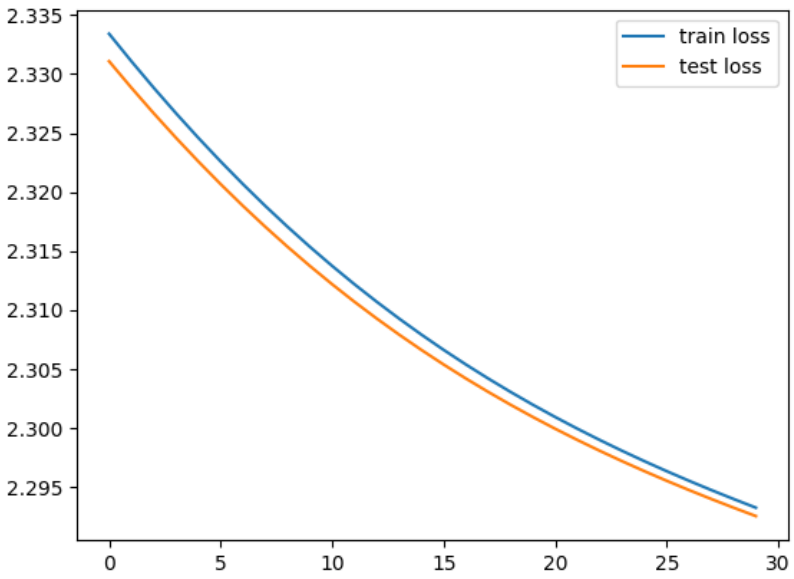

Figure 8: Loss values for 0.0001, 30, 3, 128, Sigmoid(), 50004

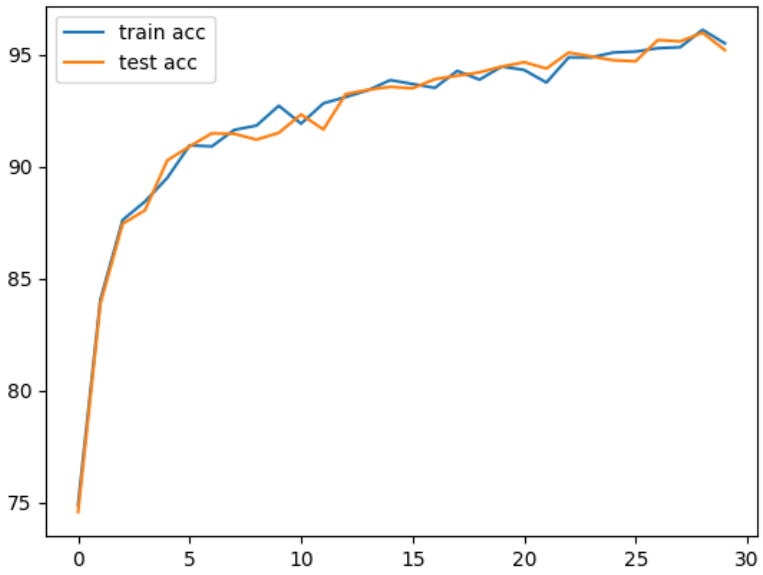

Figure 9: Accuracy values for 0.0001, 30, 3, 128, Tanh(), 463

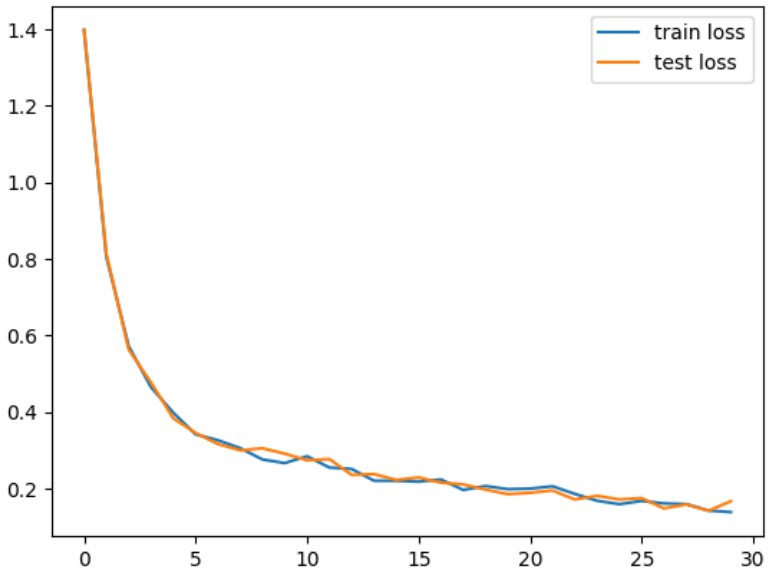

Figure 10: Loss values for 0.0001, 30, 3, 128, Tanh(), 463

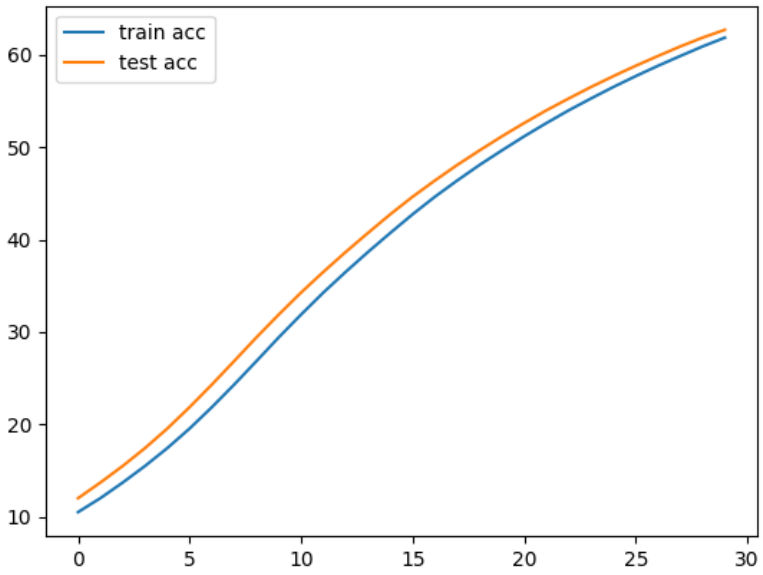

Figure 11: Accuracy values for 0.0001, 30, 3, 128, Tanh(), 50004

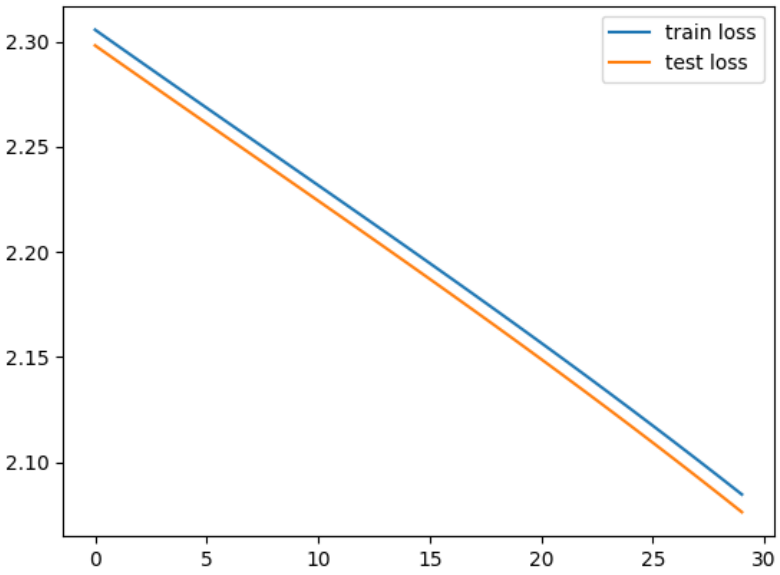

Figure 12: Loss values for 0.0001, 30, 3, 128, Tanh(), 50004

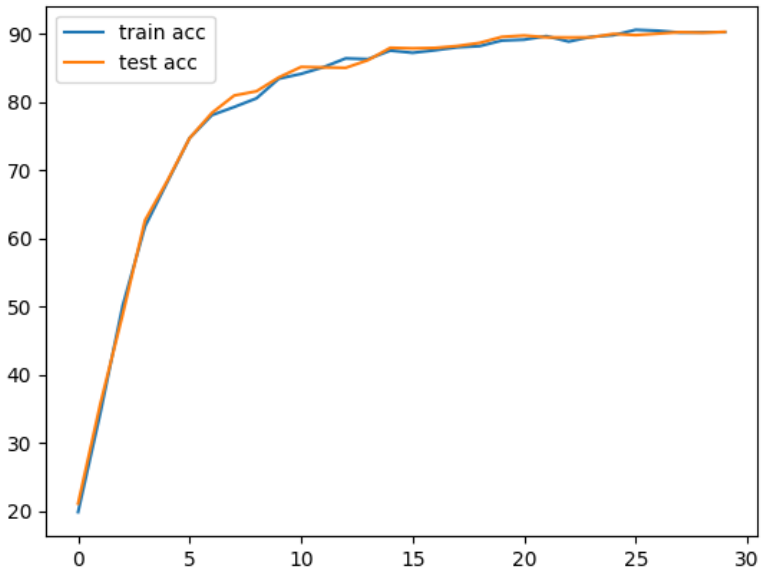

Figure 13: Accuracy values for 0.0001, 30, 3, 16, ReLU(), 463

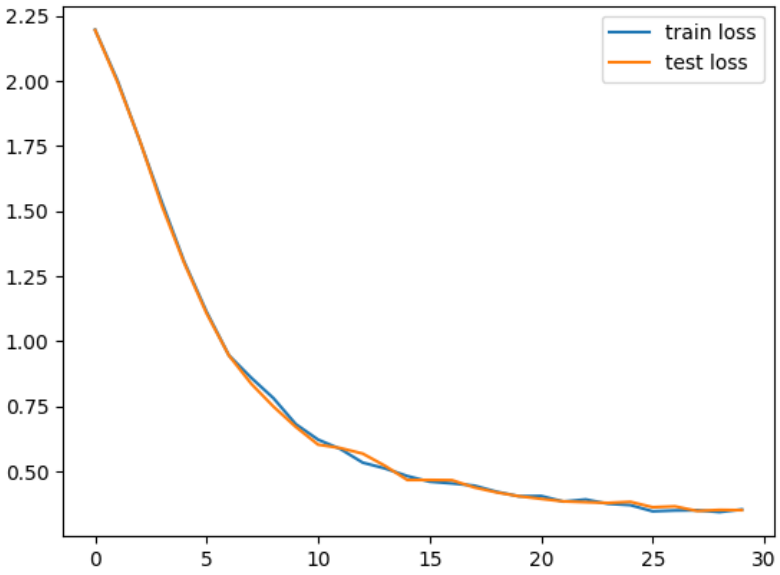

Figure 14: Loss values for 0.0001, 30, 3, 16, ReLU(), 463

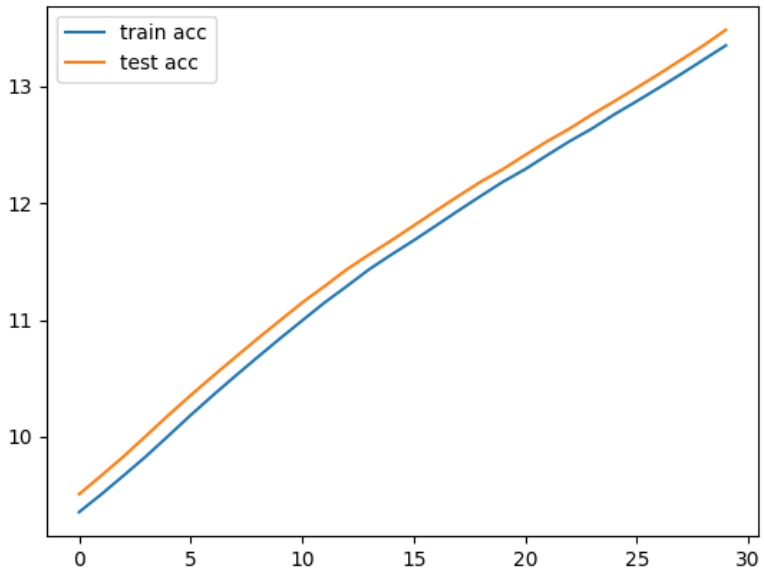

Figure 15: Accuracy values for 0.0001, 30, 3, 16, ReLU(), 50004

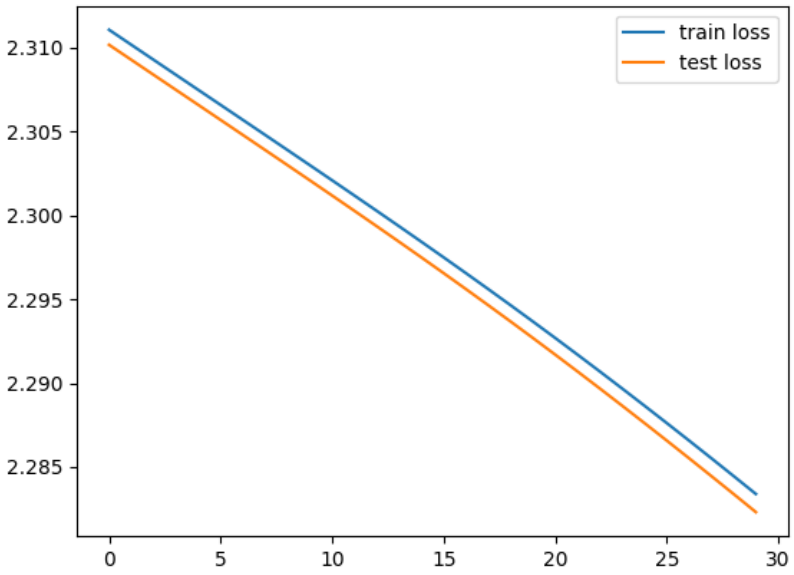

Figure 16: Loss values for 0.0001, 30, 3, 16, ReLU(), 50004

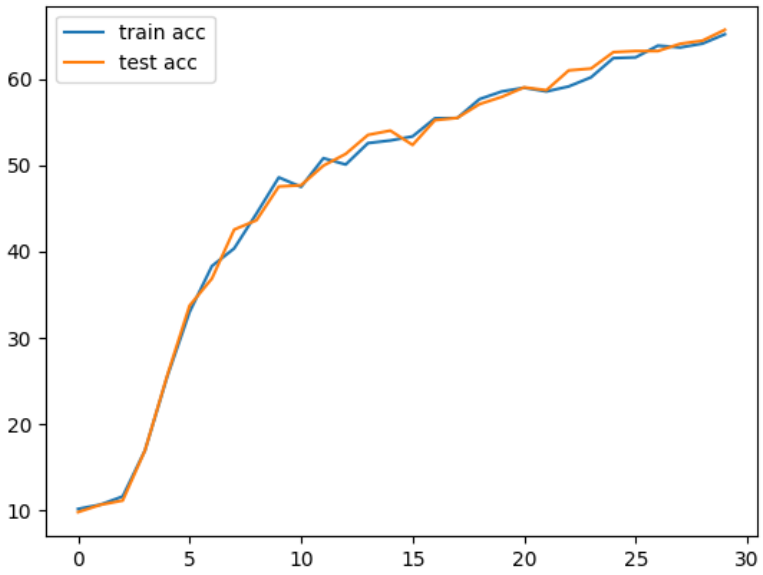

Figure 17: Accuracy values for 0.0001, 30, 3, 16, Sigmoid(), 463

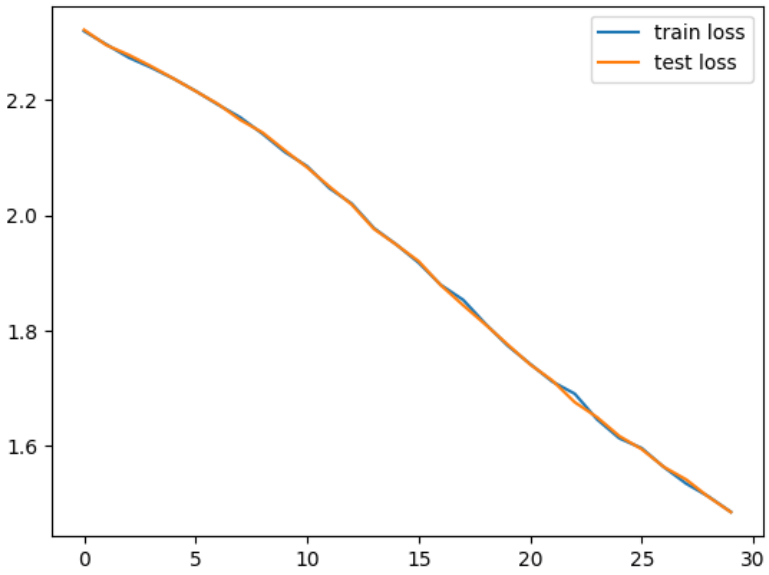

Figure 18: Loss values for 0.0001, 30, 3, 16, Sigmoid(), 463

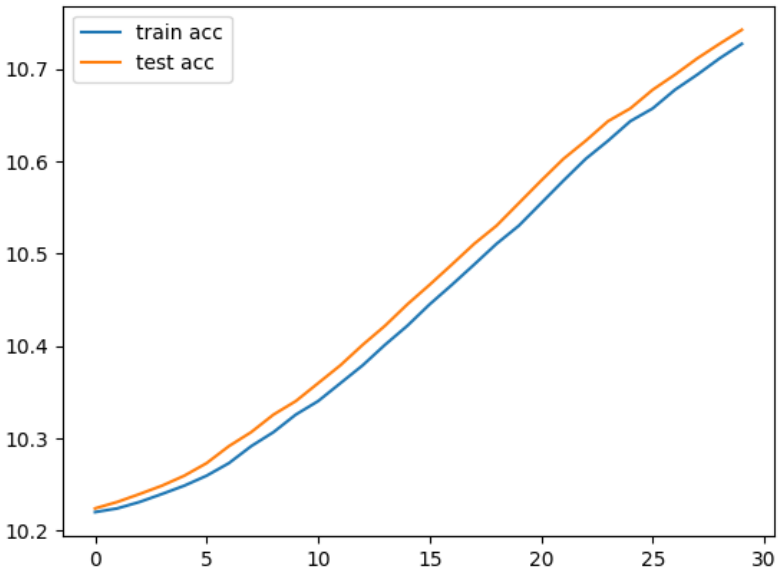

Figure 19: Accuracy values for 0.0001, 30, 3, 16, Sigmoid(), 50004

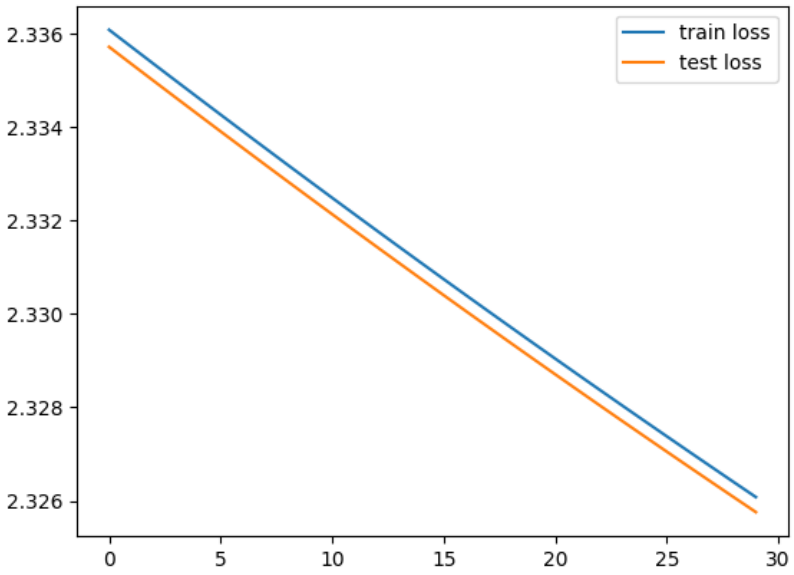

Figure 20: Loss values for 0.0001, 30, 3, 16, Sigmoid(), 50004

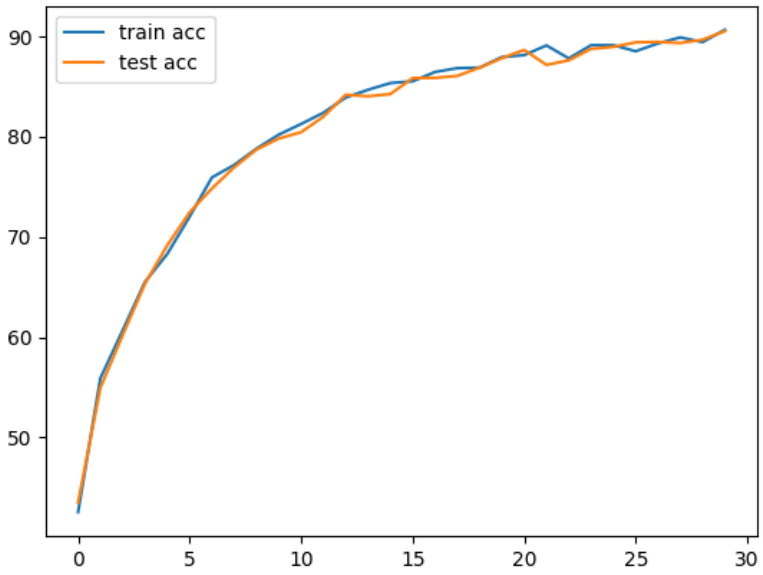

Figure 21: Accuracy values for 0.0001, 30, 3, 16, Tanh(), 463

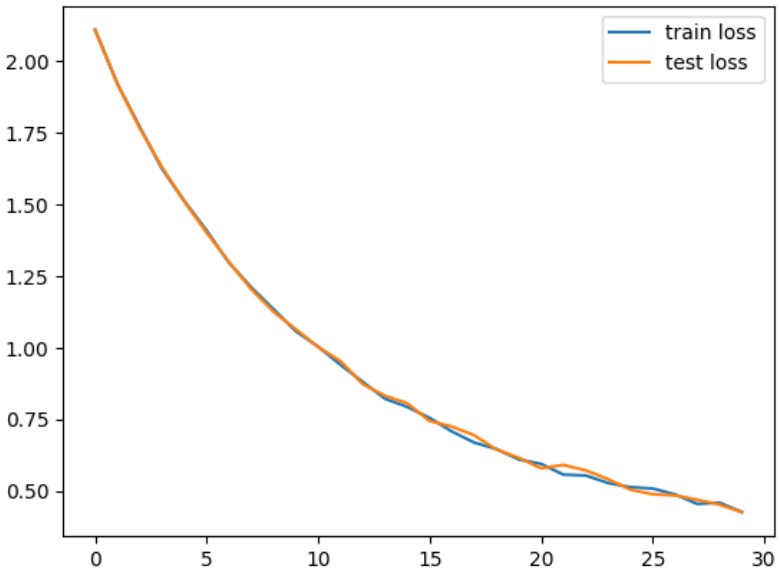

Figure 22: Loss values for 0.0001, 30, 3, 16, Tanh(), 463

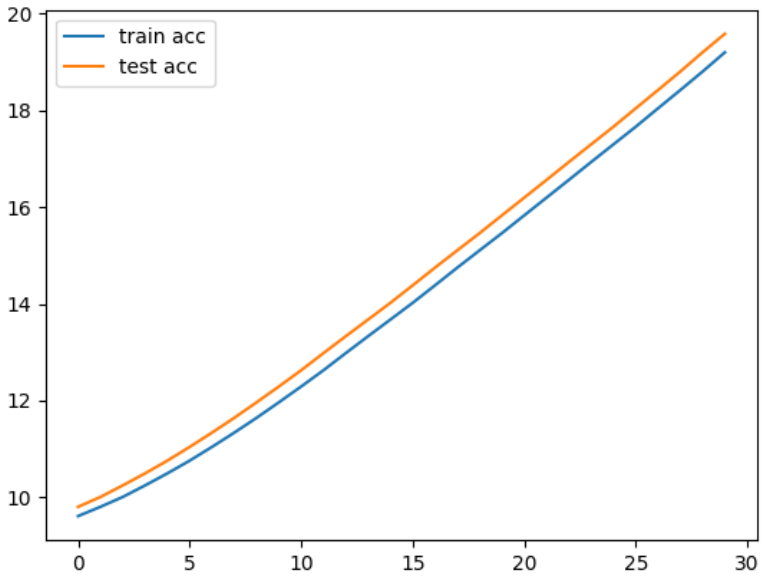

Figure 23: Accuracy values for 0.0001, 30, 3, 16, Tanh(), 50004

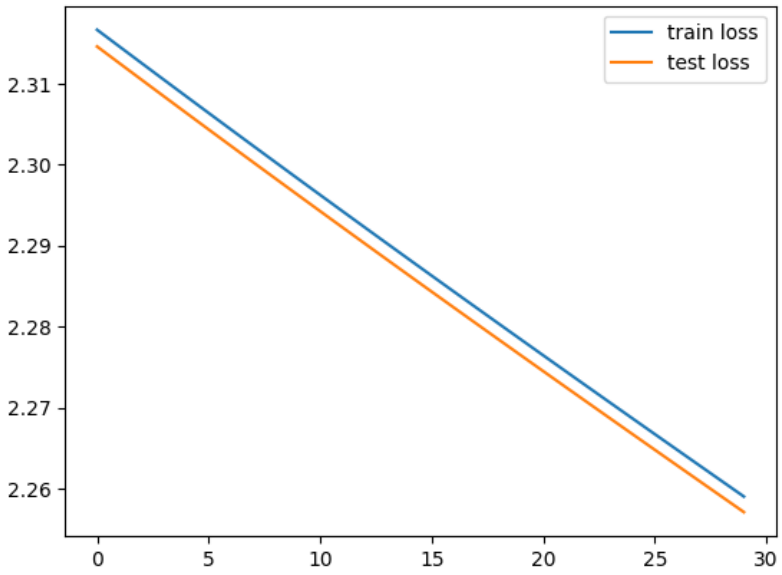

Figure 24: Loss values for 0.0001, 30, 3, 16, Tanh(), 50004

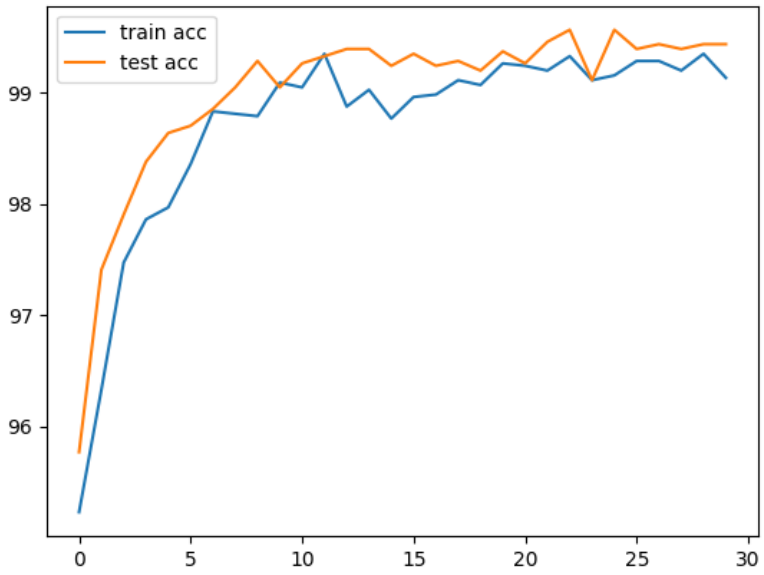

Figure 25: Accuracy values for 0.01, 30, 3, 128, ReLU(), 463

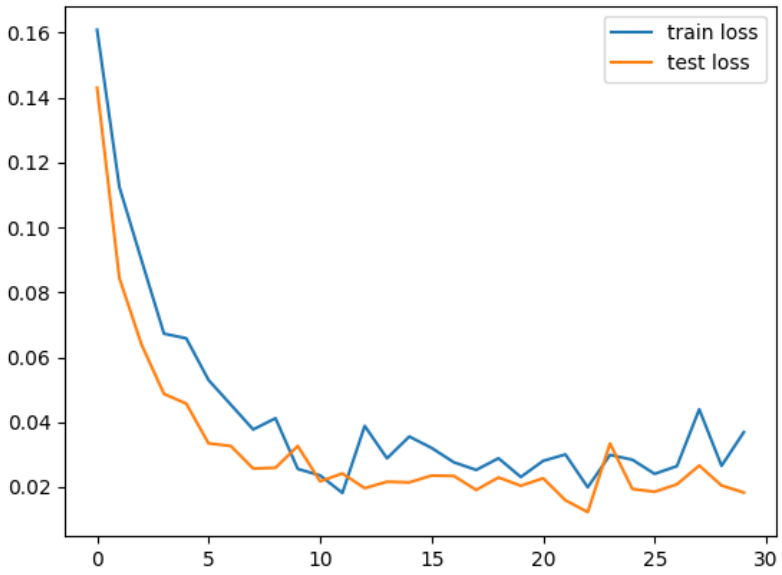

Figure 26: Loss values for 0.01, 30, 3, 128, ReLU(), 463

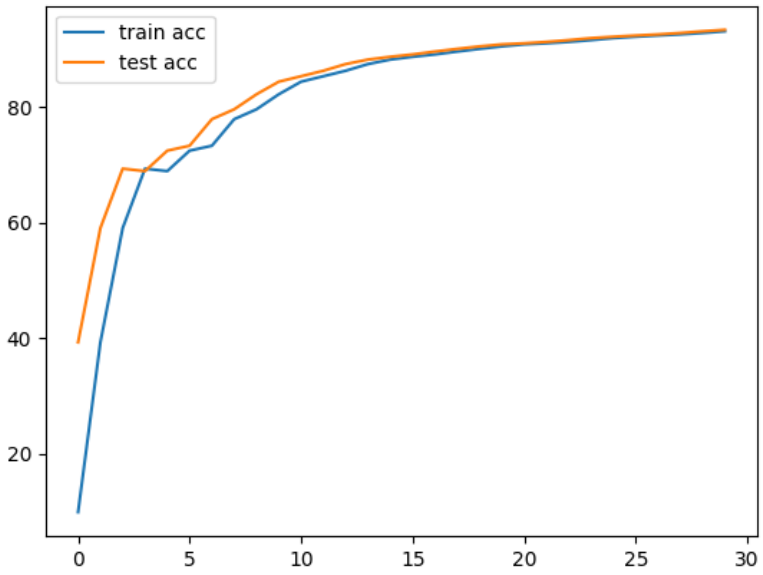

Figure 27: Accuracy values for 0.01, 30, 3, 128, ReLU(), 50004

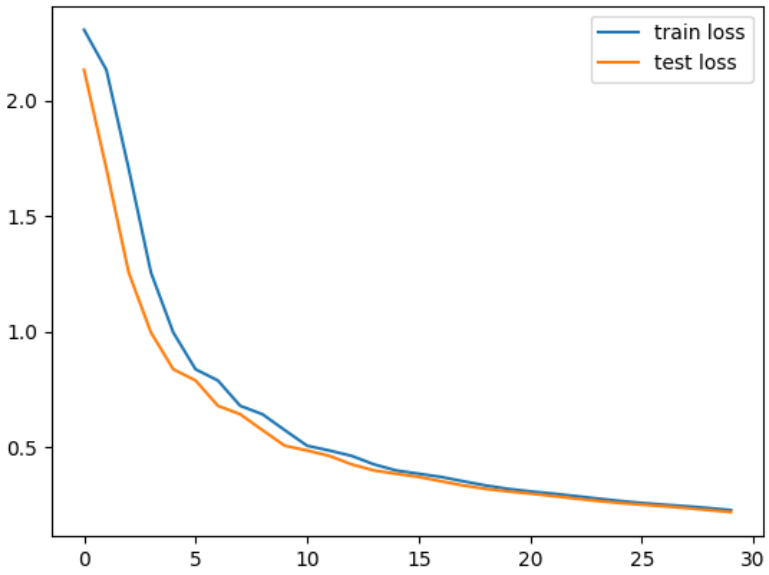

Figure 28: Loss values for 0.01, 30, 3, 128, ReLU(), 50004

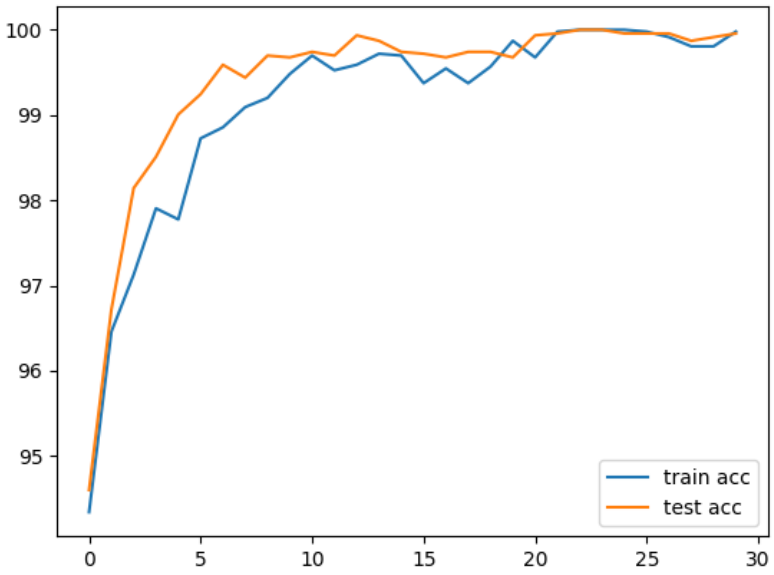

Figure 29: Accuracy values for 0.01, 30, 3, 128, Sigmoid(), 463

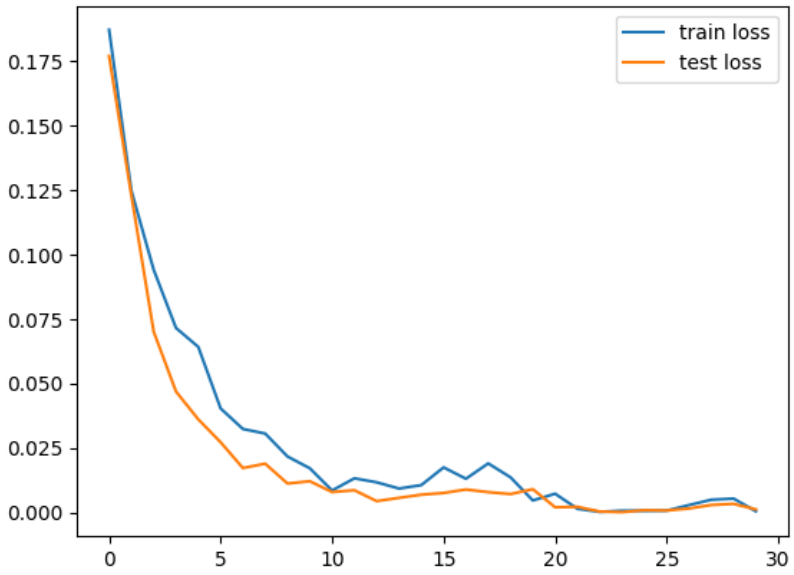

Figure 30: Loss values for 0.01, 30, 3, 128, Sigmoid(), 463

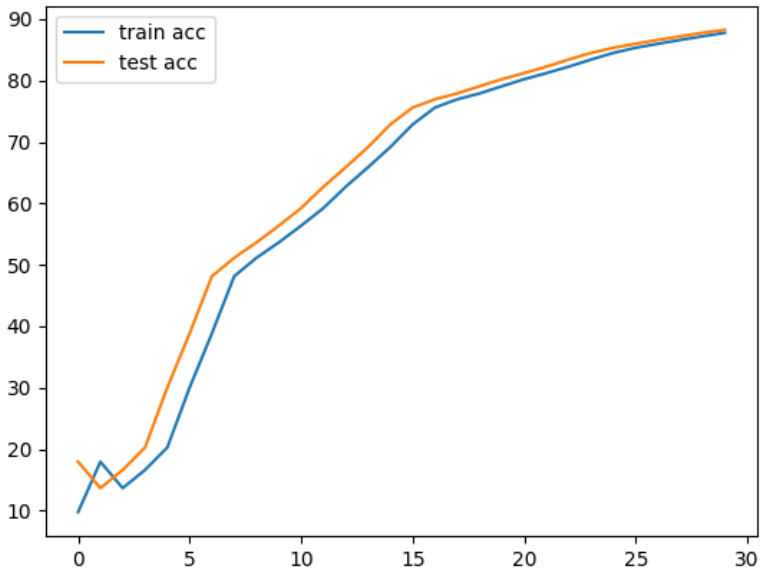

Figure 31: Accuracy values for 0.01, 30, 3, 128, Sigmoid(), 50004

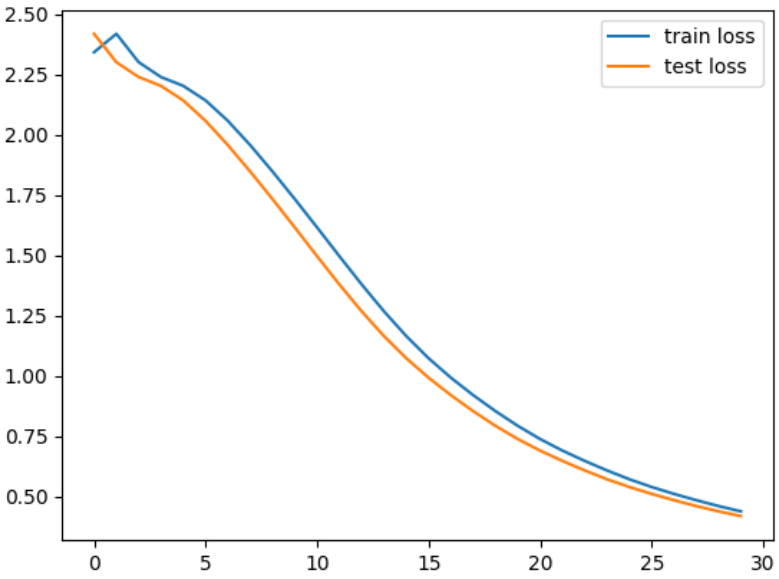

Figure 32: Loss values for 0.01, 30, 3, 128, Sigmoid(), 50004

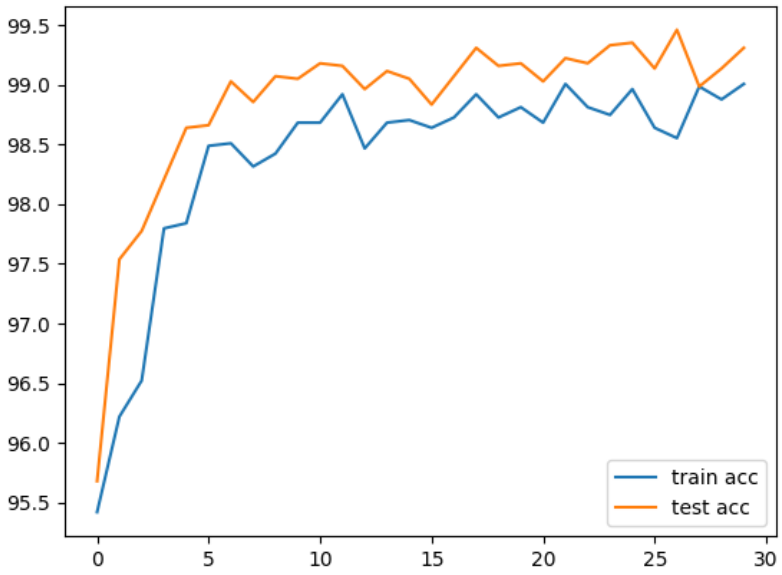

Figure 33: Accuracy values for 0.01, 30, 3, 128, Tanh(), 463

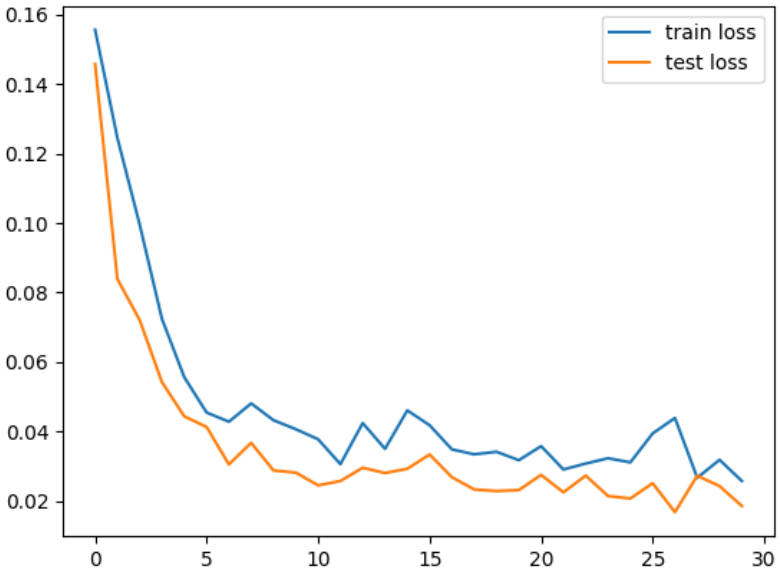

Figure 34: Loss values for 0.01, 30, 3, 128, Tanh(), 463

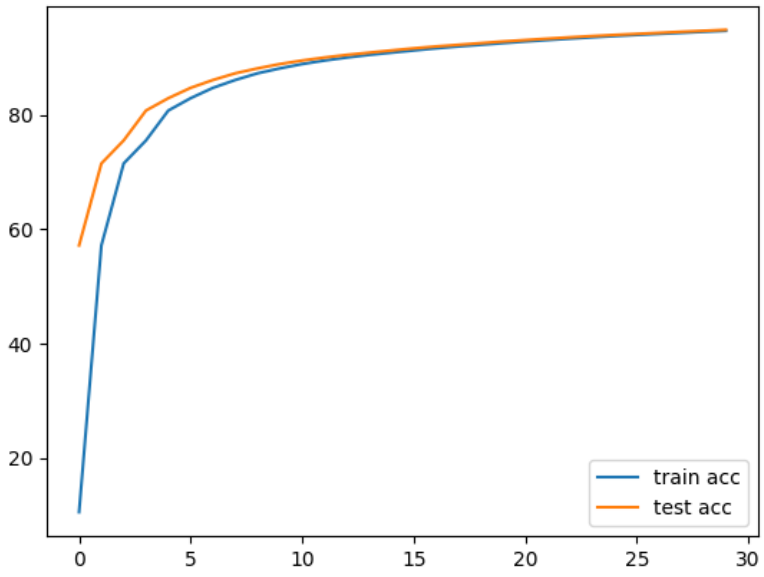

Figure 35: Accuracy values for 0.01, 30, 3, 128, Tanh(), 50004

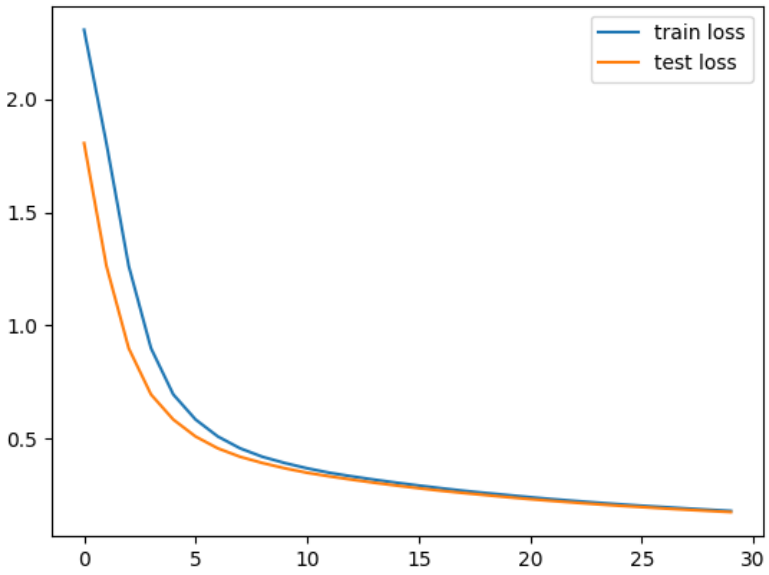

Figure 36: Loss values for 0.01, 30, 3, 128, Tanh(), 50004

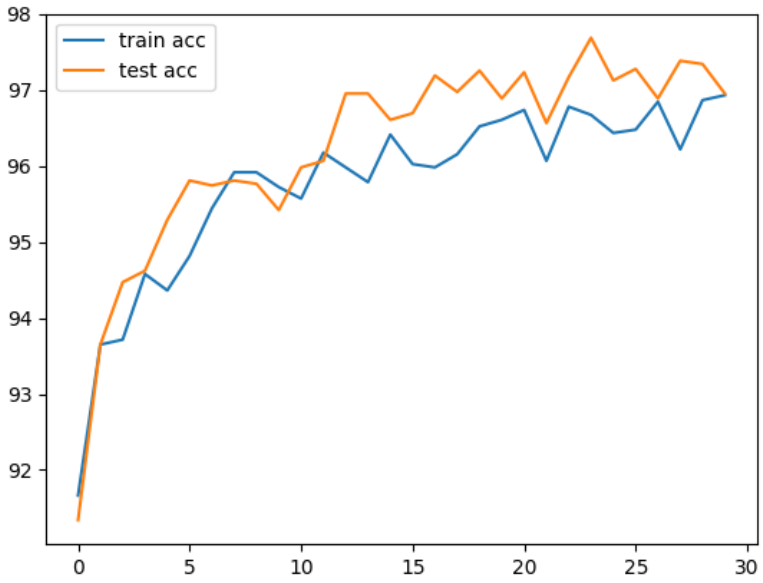

Figure 37: Accuracy values for 0.01, 30, 3, 16, ReLU(), 463

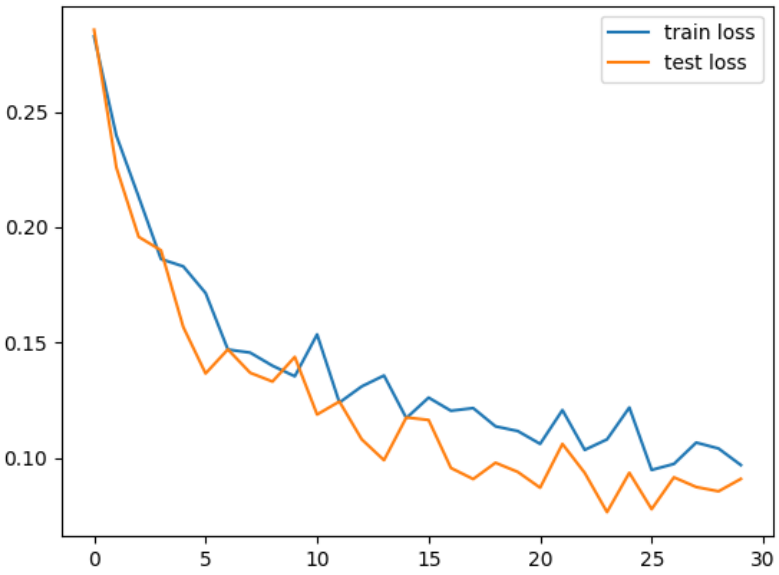

Figure 38: Loss values for 0.01, 30, 3, 16, ReLU(), 463

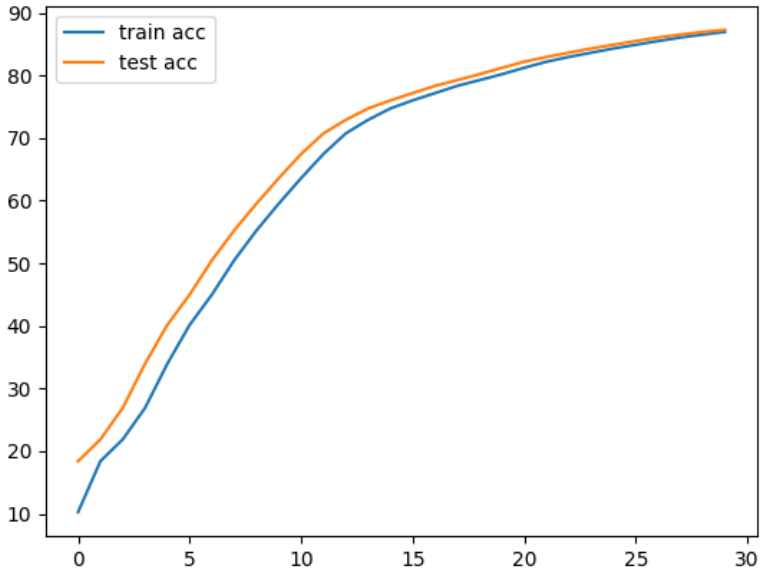

Figure 39: Accuracy values for 0.01, 30, 3, 16, ReLU(), 50004

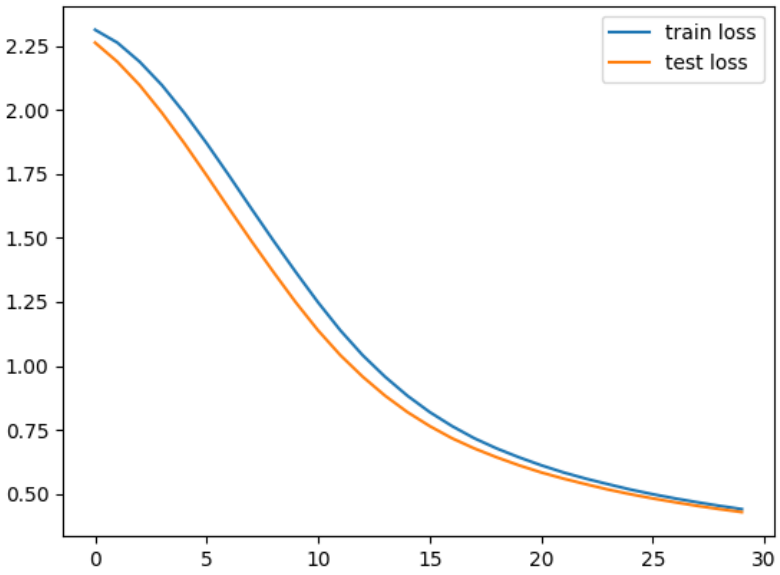

Figure 40: Loss values for 0.01, 30, 3, 16, ReLU(), 50004

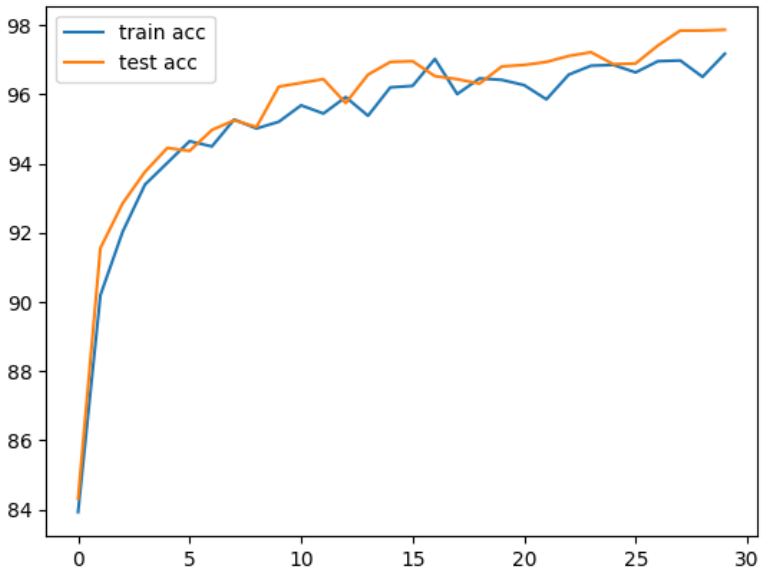

Figure 41: Accuracy values for 0.01, 30, 3, 16, Sigmoid(), 463

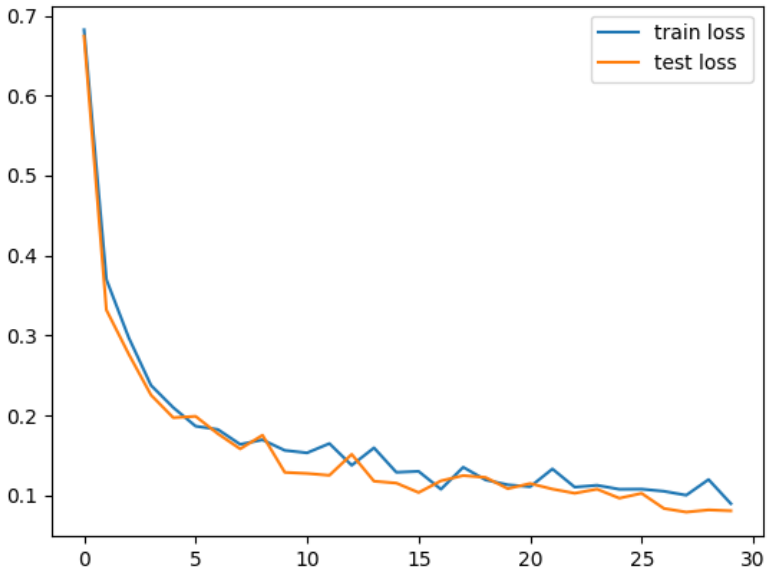

Figure 42: Loss values for 0.01, 30, 3, 16, Sigmoid(), 463

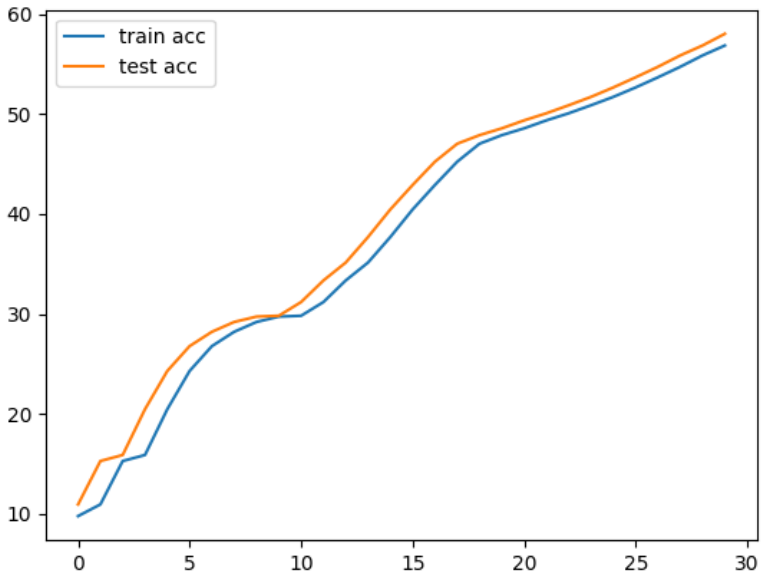

Figure 43: Accuracy values for 0.01, 30, 3, 16, Sigmoid(), 50004

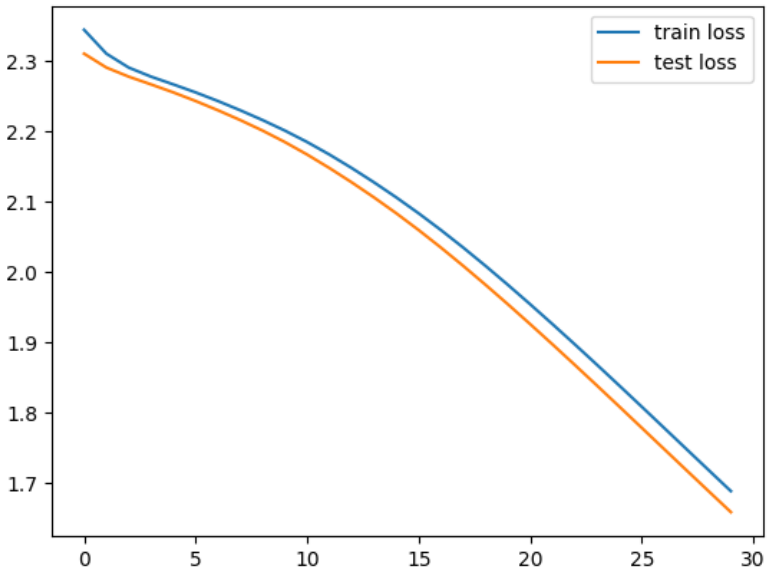

Figure 44: Loss values for 0.01, 30, 3, 16, Sigmoid(), 50004

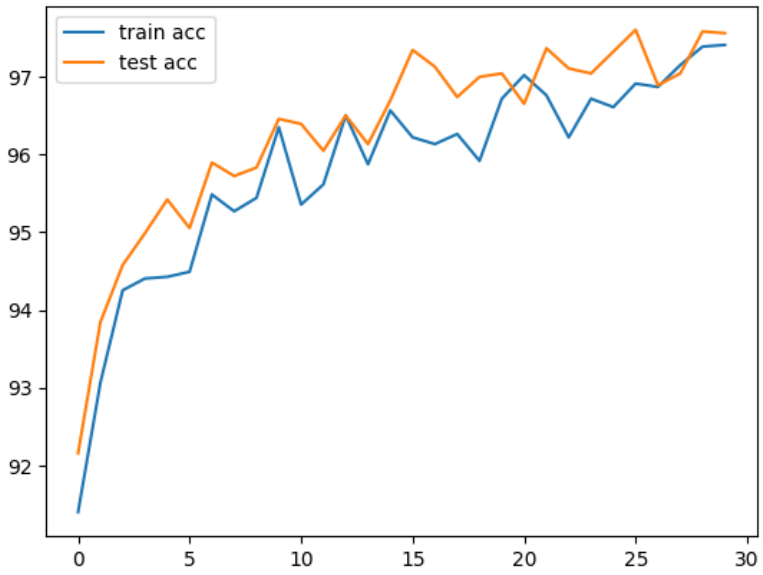

Figure 45: Accuracy values for 0.01, 30, 3, 16, Tanh(), 463

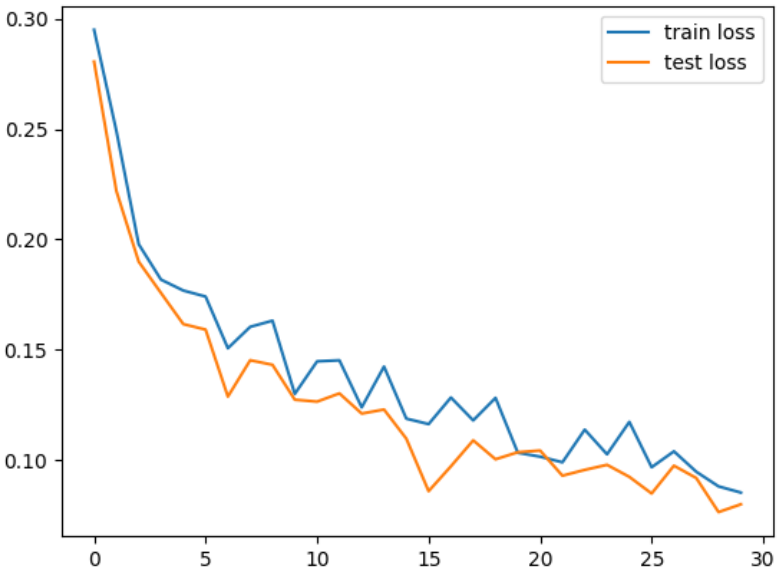

Figure 46: Loss values for 0.01, 30, 3, 16, Tanh(), 463

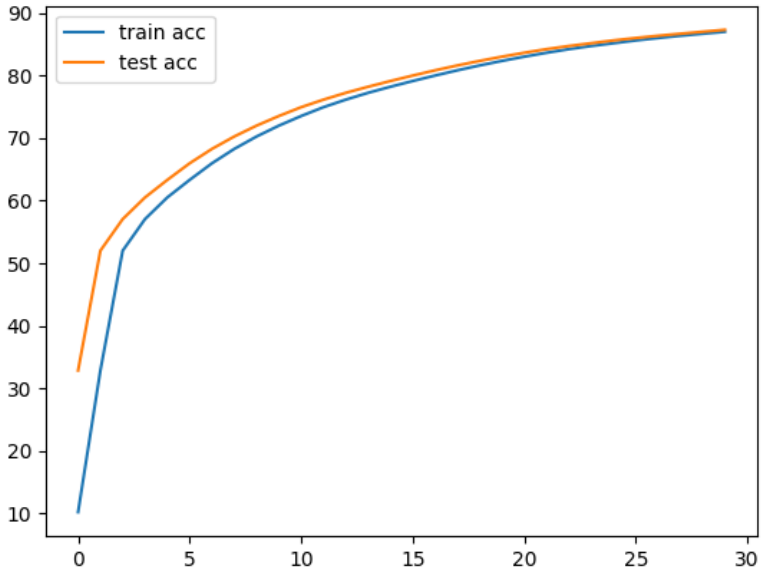

Figure 47: Accuracy values for 0.01, 30, 3, 16, Tanh(), 50004

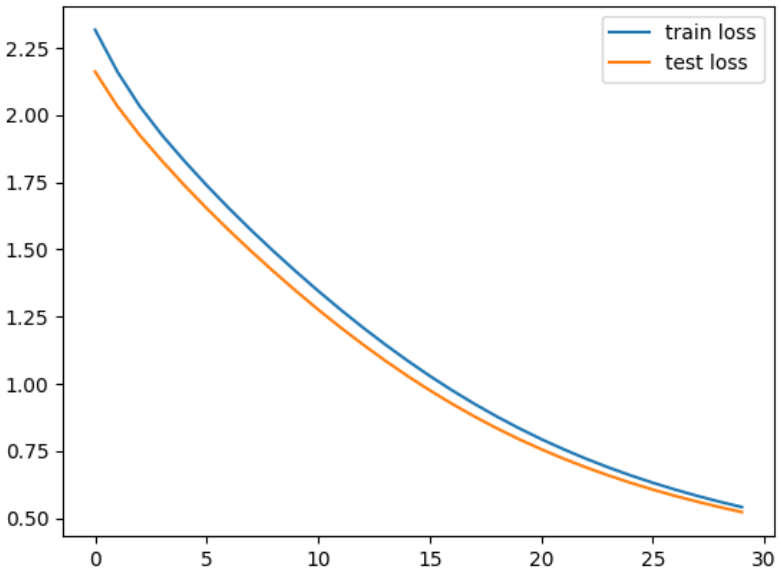

Figure 48: Loss values for 0.01, 30, 3, 16, Tanh(), 50004

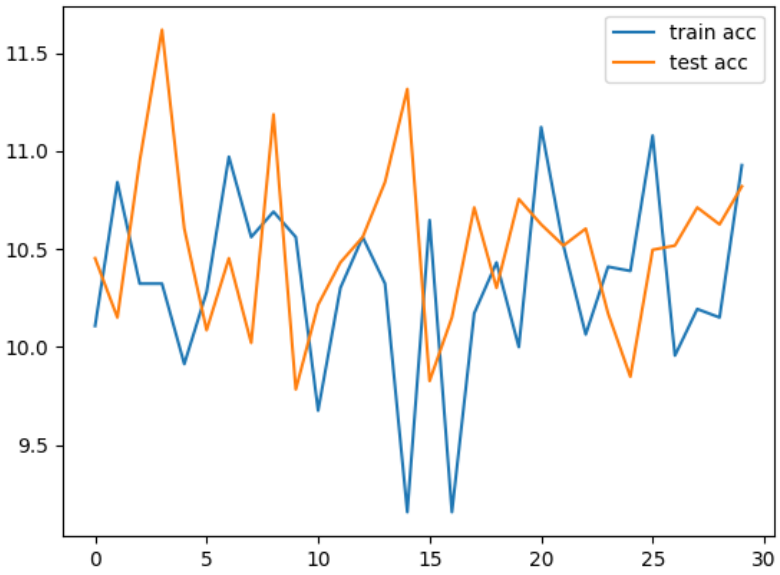

Figure 49: Accuracy values for 1, 30, 3, 128, ReLU(), 463

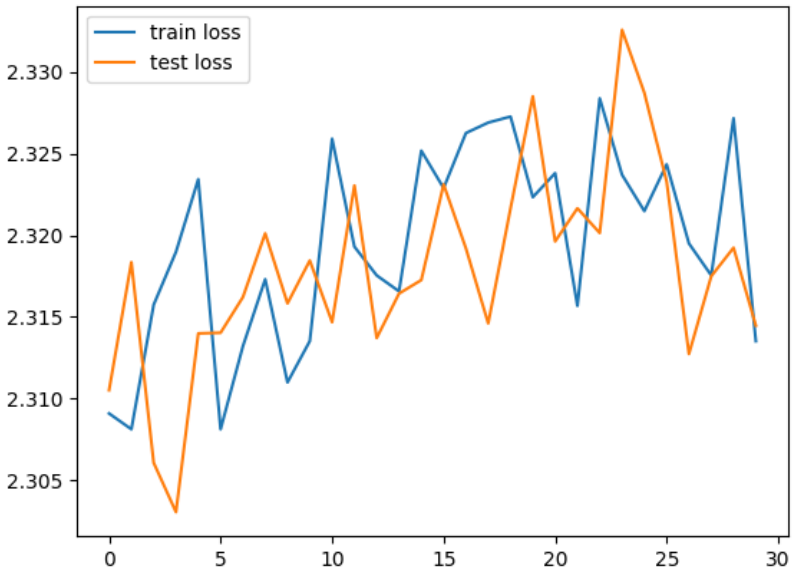

Figure 50: Loss values for 1, 30, 3, 128, ReLU(), 463

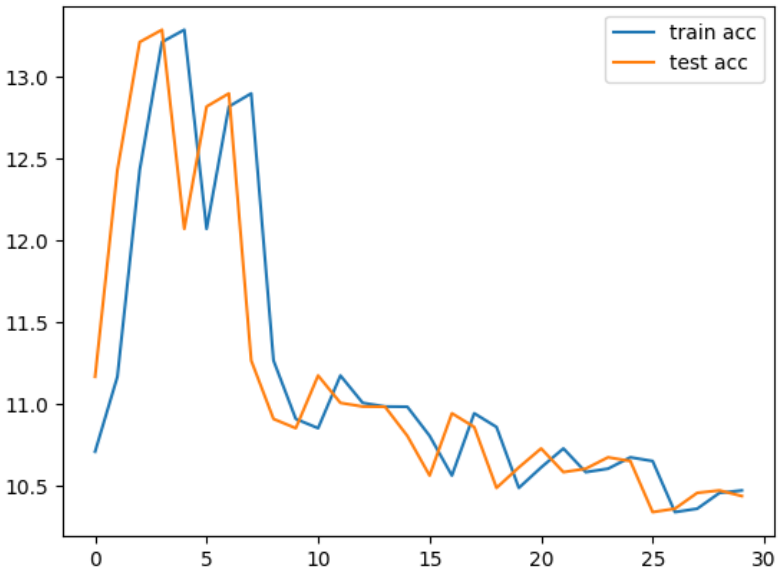

Figure 51: Accuracy values for 1, 30, 3, 128, ReLU(), 50004

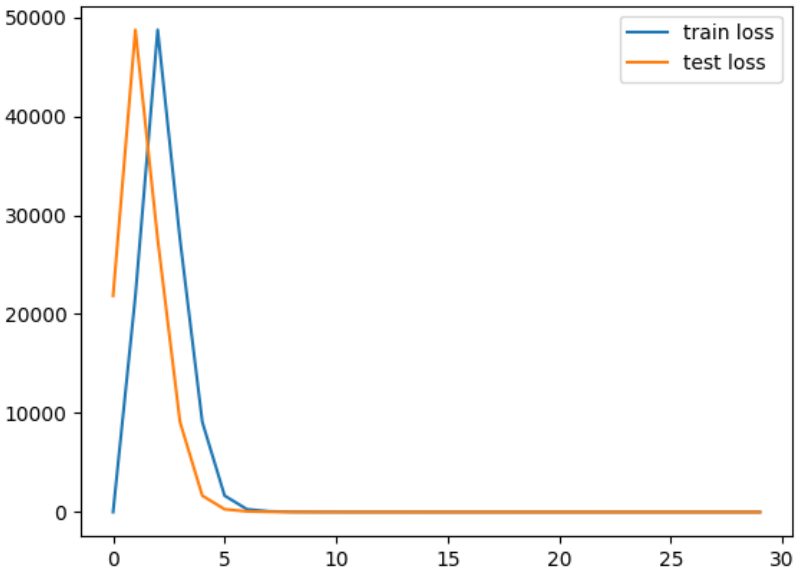

Figure 52: Loss values for 1, 30, 3, 128, ReLU(), 50004

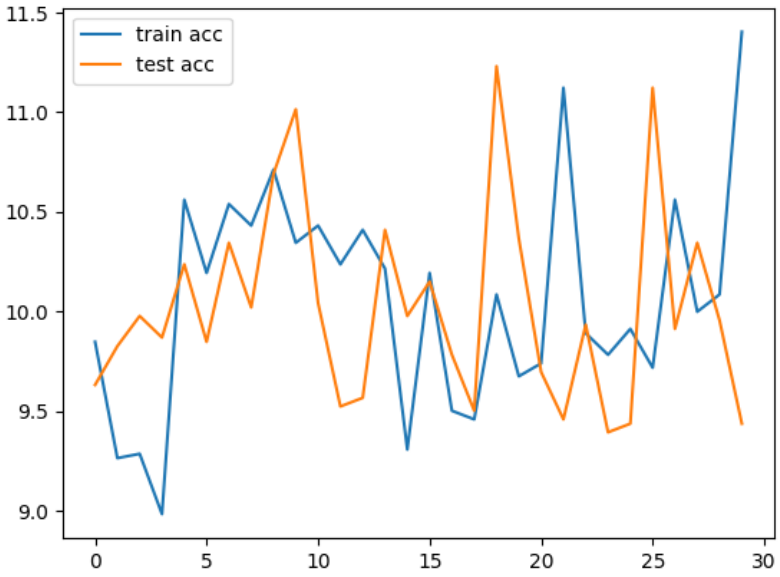

Figure 53: Accuracy values for 1, 30, 3, 128, Sigmoid(), 463

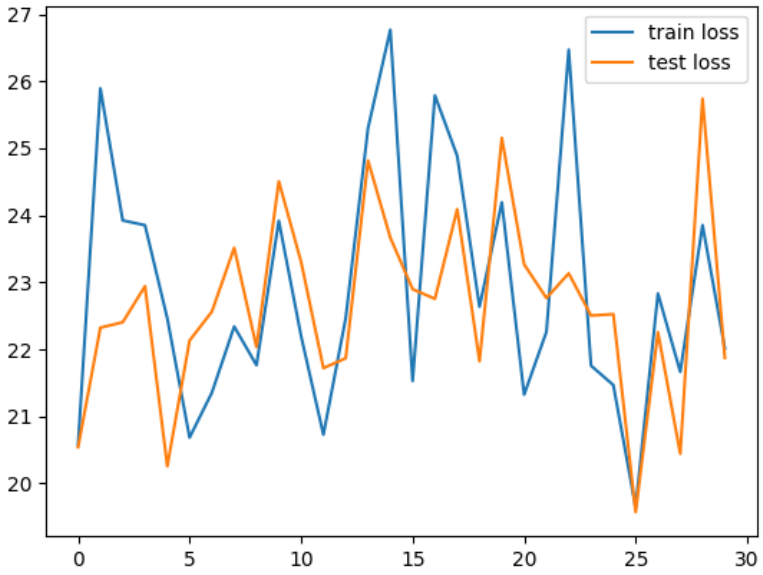

Figure 54: Loss values for 1, 30, 3, 128, Sigmoid(), 463

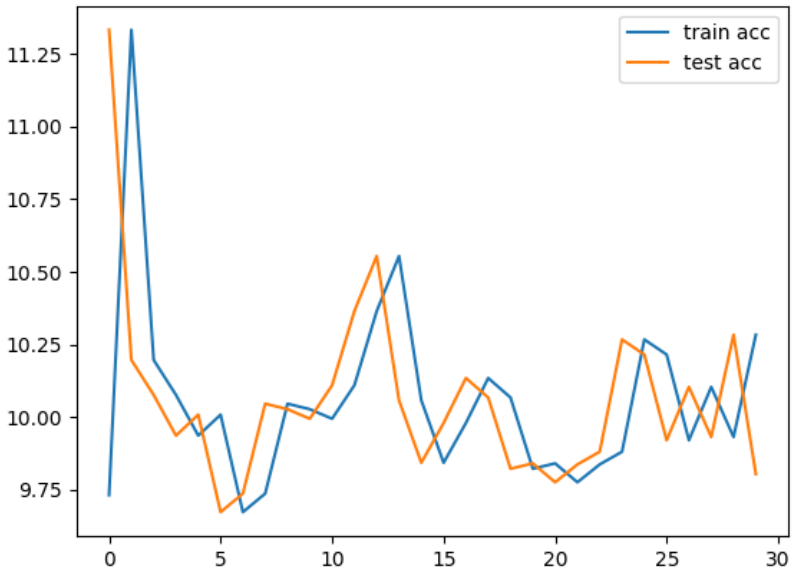

Figure 55: Accuracy values for 1, 30, 3, 128, Sigmoid(), 50004

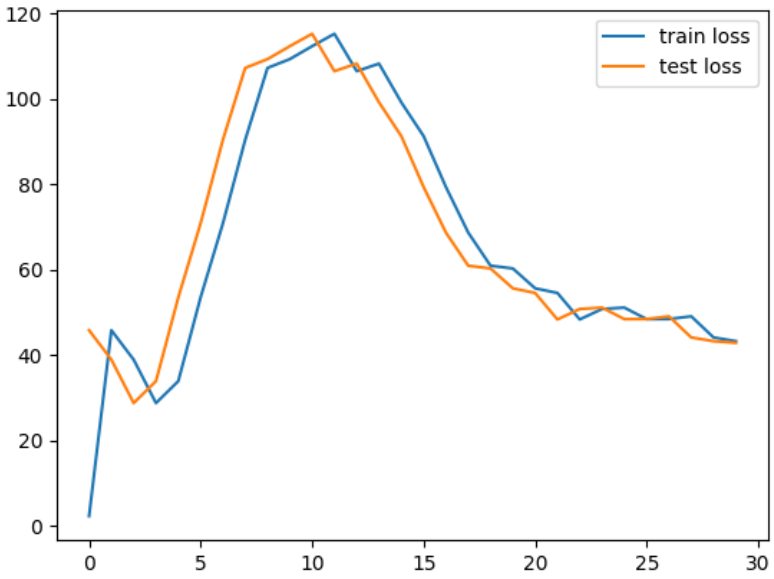

Figure 56: Loss values for 1, 30, 3, 128, Sigmoid(), 50004

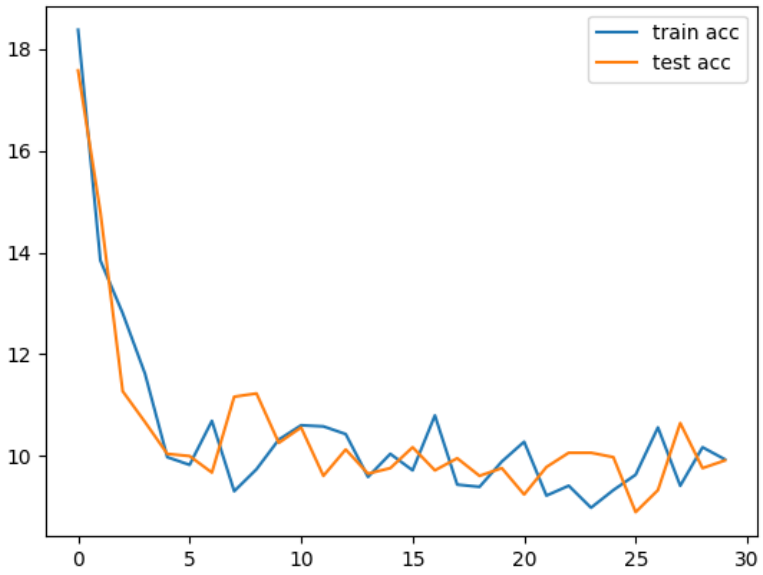

Figure 57: Accuracy values for 1, 30, 3, 128, Tanh(), 463

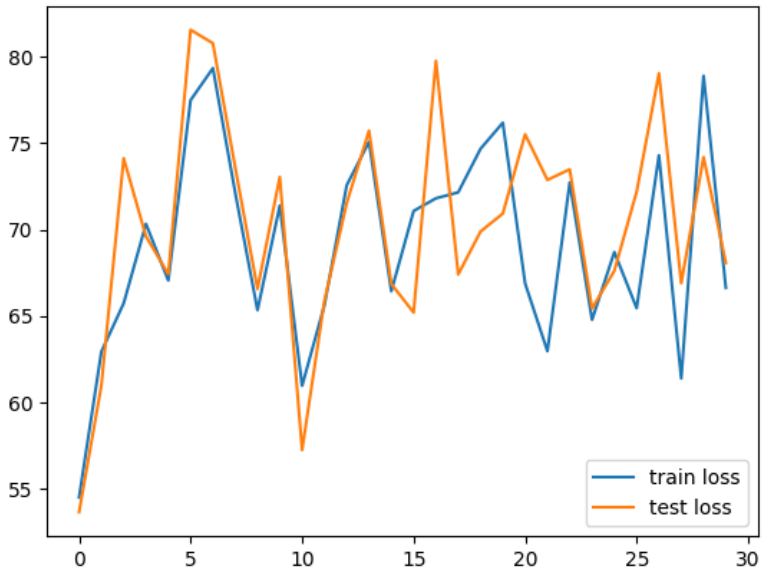

Figure 58: Loss values for 1, 30, 3, 128, Tanh(), 463

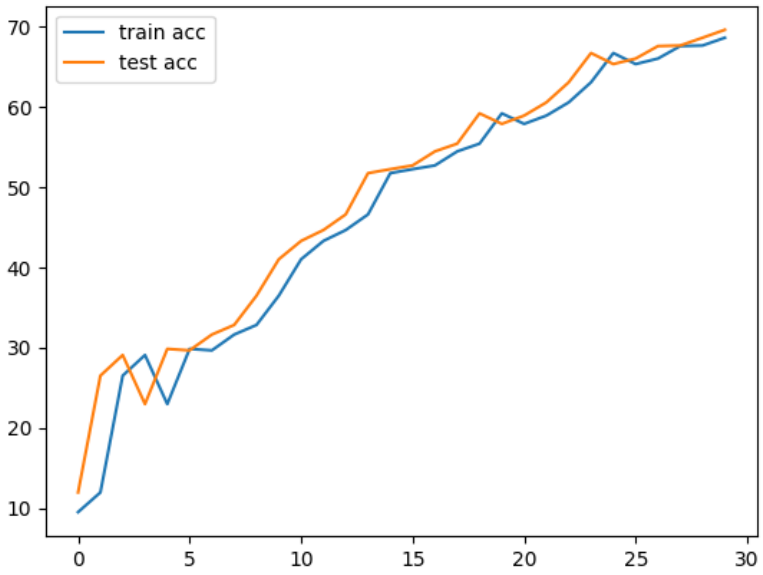

Figure 59: Accuracy values for 1, 30, 3, 128, Tanh(), 50004

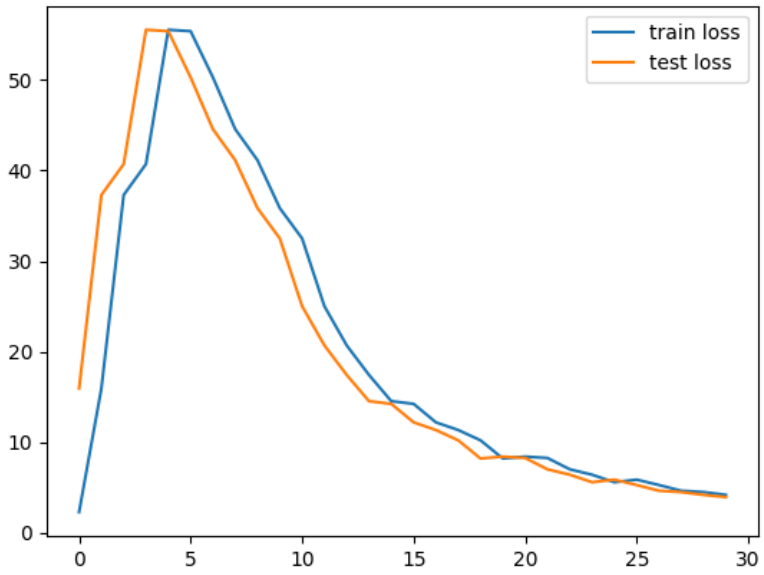

Figure 60: Loss values for 1, 30, 3, 128, Tanh(), 50004

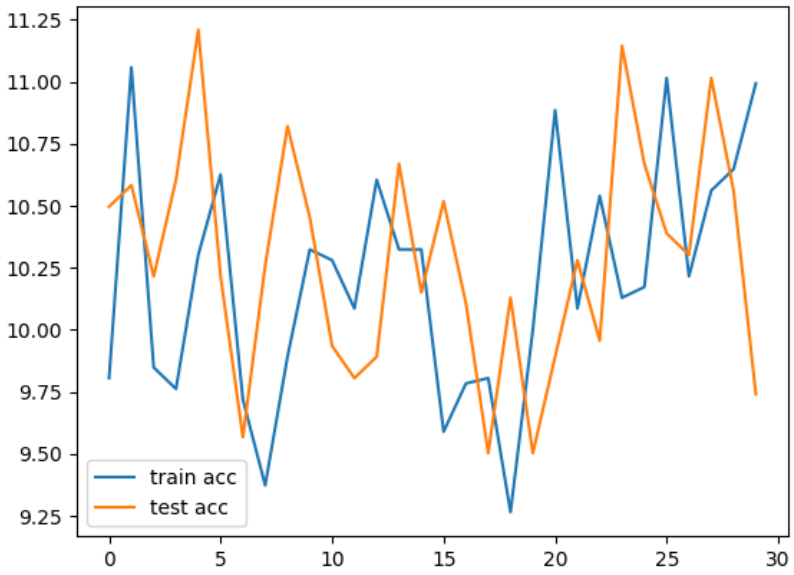

Figure 61: Accuracy values for 1, 30, 3, 16, ReLU(), 463

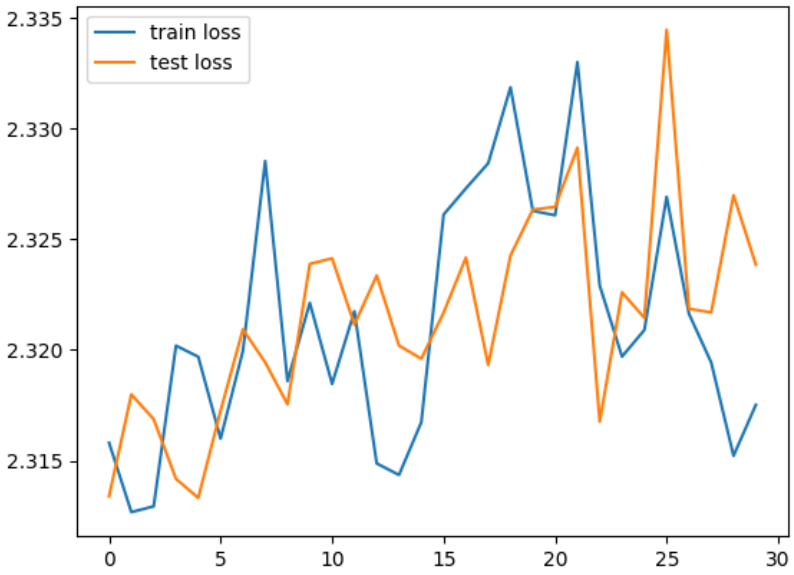

Figure 62: Loss values for 1, 30, 3, 16, ReLU(), 463

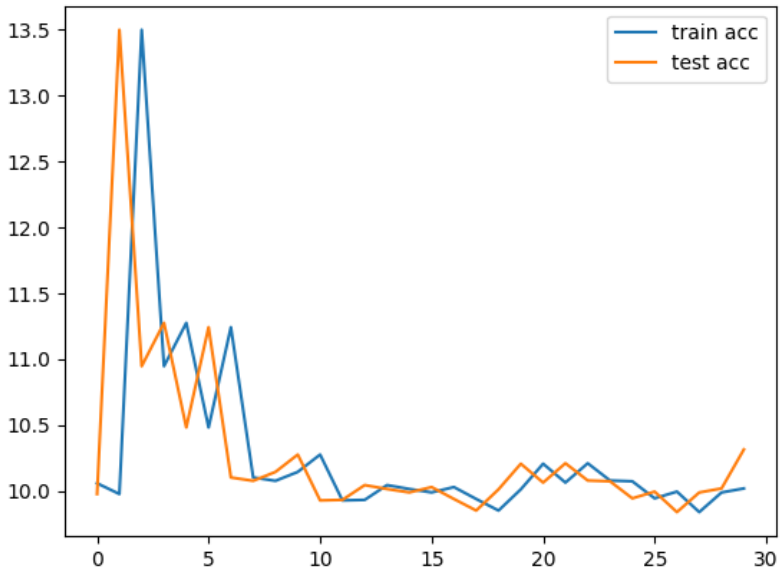

Figure 63: Accuracy values for 1, 30, 3, 16, ReLU(), 50004

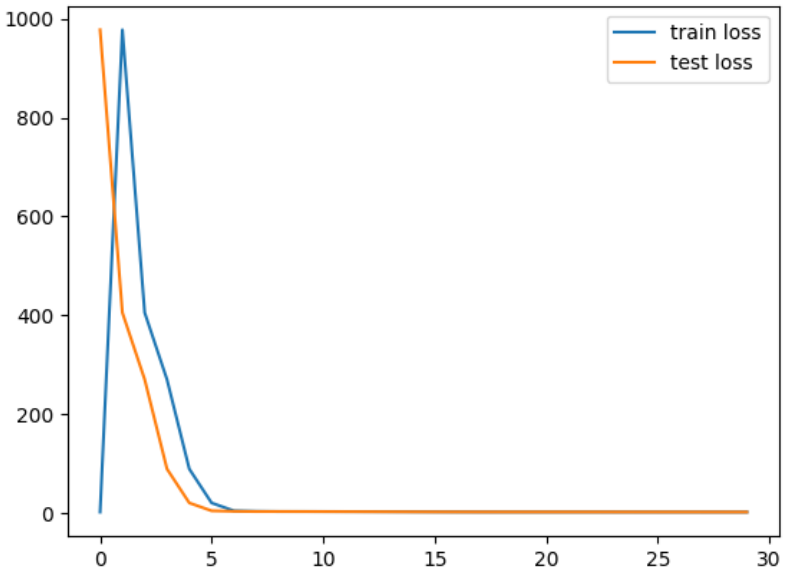

Figure 64: Loss values for 1, 30, 3, 16, ReLU(), 50004

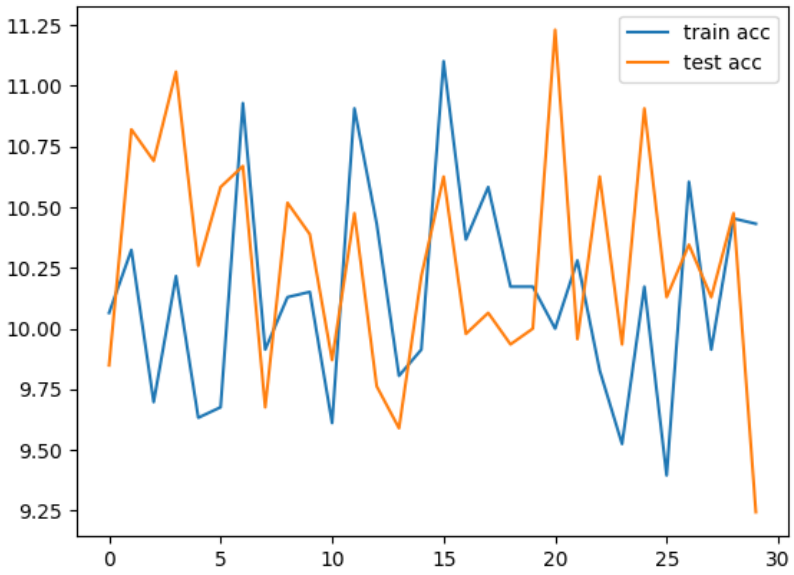

Figure 65: Accuracy values for 1, 30, 3, 16, Sigmoid(), 463

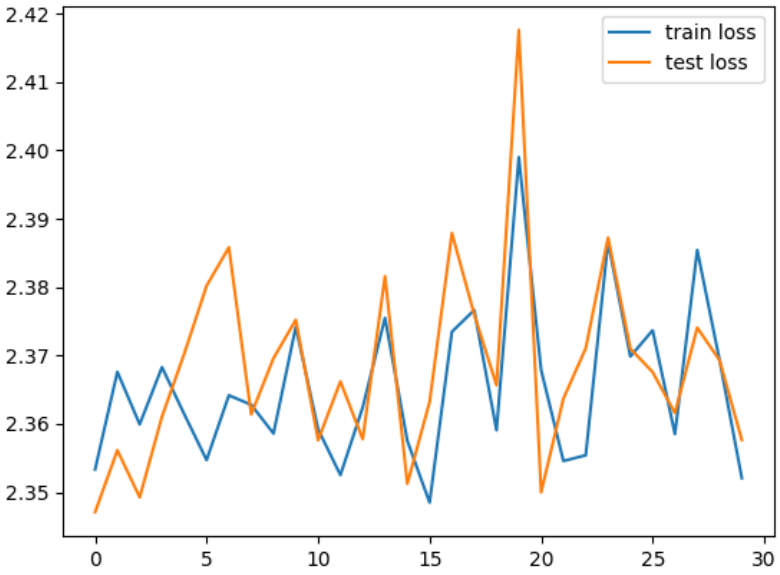

Figure 66: Loss values for 1, 30, 3, 16, Sigmoid(), 463

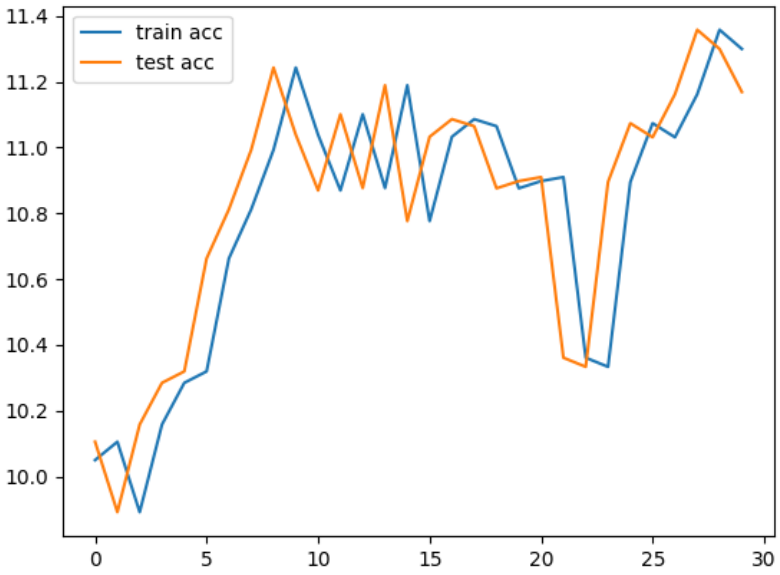

Figure 67: Accuracy values for 1, 30, 3, 16, Sigmoid(), 50004

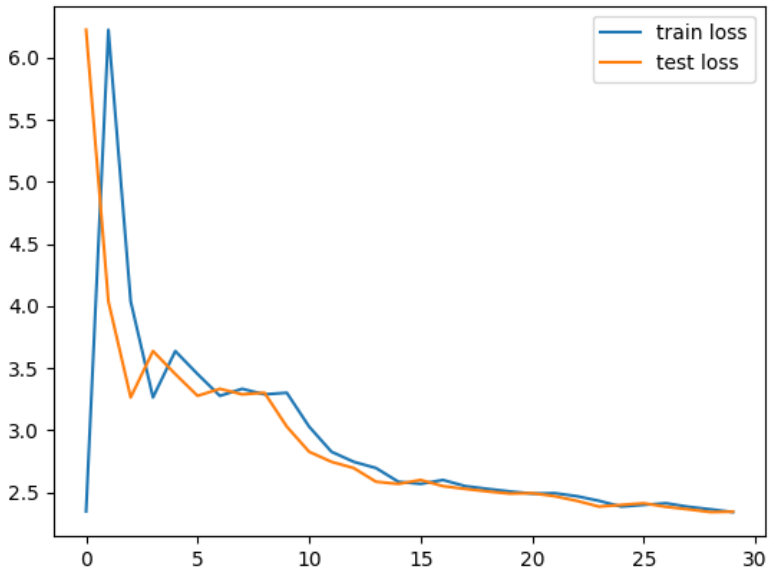

Figure 68: Loss values for 1, 30, 3, 16, Sigmoid(), 50004

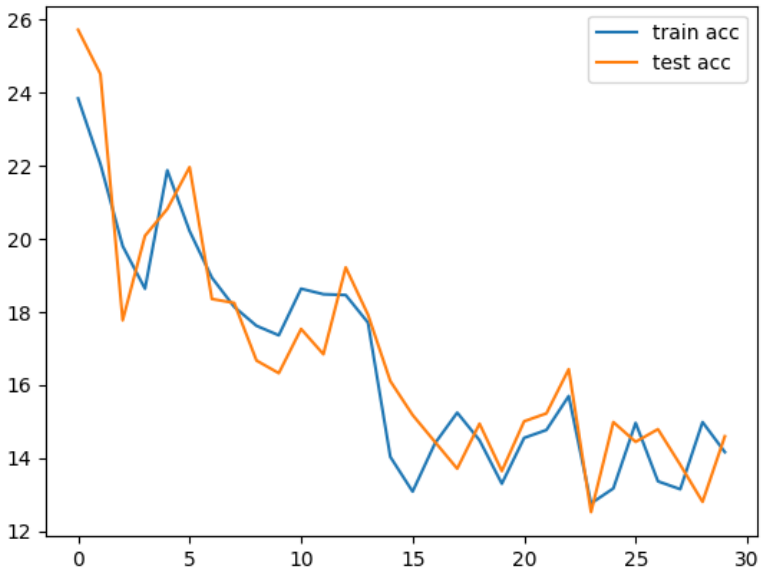

Figure 69: Accuracy values for 1, 30, 3, 16, Tanh(), 463

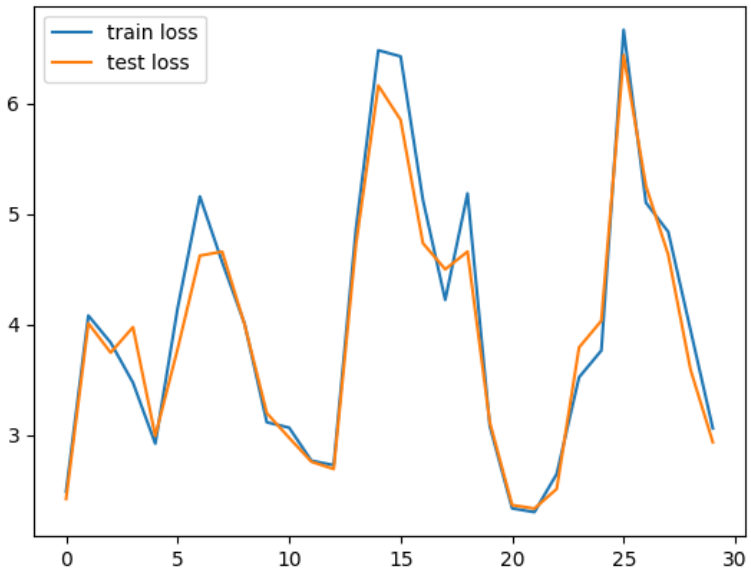

Figure 70: Loss values for 1, 30, 3, 16, Tanh(), 463

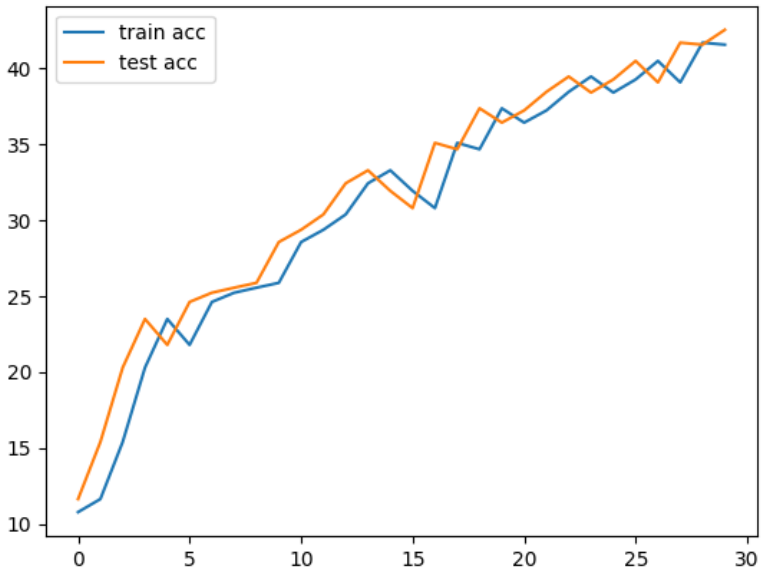

Figure 71: Accuracy values for 1, 30, 3, 16, Tanh(), 50004

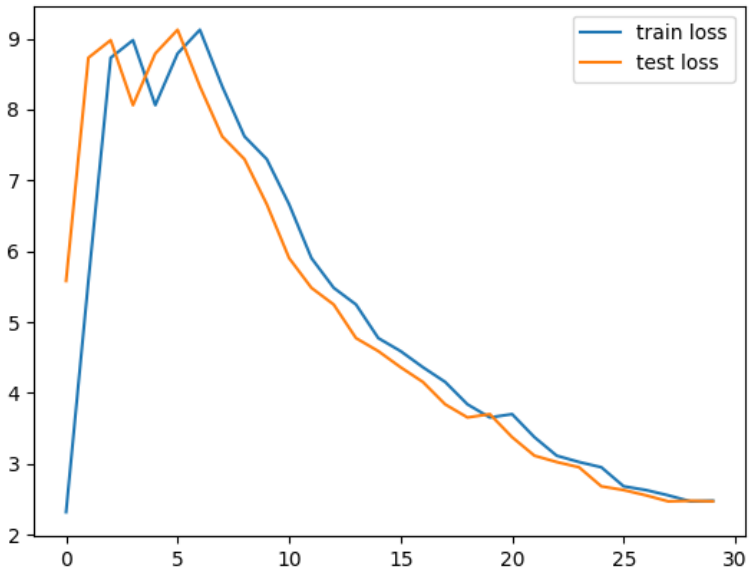

Figure 72: Loss values for 1, 30, 3, 16, Tanh(), 50004

5 References

References

[1] Cohen, M. X. (2022, October). A deep understanding of deep learning (with python intro). Udemy. Retrieved November 6, 2022, from `https://www.udemy.com/course/deeplearning_x`