COL774 Assignment 3

Rishabh Kumar (2021CS10103)

April 2025

1 Decision Tree(and Random Forests)

1.1 Decision Tree Construction

In this experiment, I implemented a decision tree algorithm from scratch to predict whether a person earns more than $50K annually based on demographic attributes. The implementation handles categorical attributes with k-way splits and continuous attributes with binary splits on median values.

Experimenting with different maximum depths (5, 10, 15, 20), I observed:

Max Depth	Training Accuracy	Test Accuracy	>50K Accuracy	<=50K Accuracy
5	0.8669	0.8231	0.5841	0.9010
10	0.9382	0.8010	0.5554	0.8810
15	0.9841	0.7893	0.5538	0.8660
20	0.9916	0.7884	0.5622	0.8621

Table 1: Decision Tree performance at different maximum depths

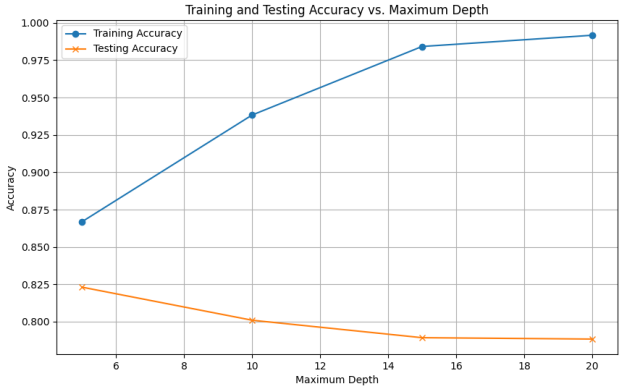


Figure 1: Training and Testing Accuracy vs. Maximum Depth

Observations:

- As maximum depth increases, training accuracy improves dramatically (86.69% to 99.16%), indicating the model's ability to capture complex patterns in training data.
- Test accuracy shows the opposite trend, decreasing from 82.31% to 78.84%, clearly demonstrating overfitting.
- Class-specific accuracies reveal a significant imbalance: the model performs much better at predicting the <=50K class (86–90% accuracy) compared to the >50K class (55–58% accuracy), suggesting dataset imbalance.
- The optimal depth is 5, providing the best balance between training and test accuracy.

1.2 Decision Tree with One-Hot Encoding

For this experiment, I modified the decision tree implementation to handle one-hot encoded categorical features. Each categorical attribute with multiple values was transformed into multiple binary attributes, with each new attribute representing the presence or absence of a specific category.

I experimented with deeper maximum depths (25, 35, 45, 55) on this transformed dataset and observed:

Max Depth	Training Accuracy	Test Accuracy
25	0.9493	0.8163
35	0.9798	0.8110
45	0.9914	0.8084
55	0.9957	0.8082

Table 2: Decision Tree performance with one-hot encoding at different maximum depths

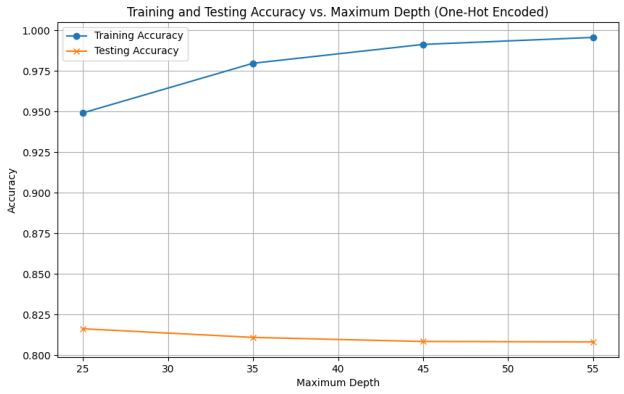


Figure 2: Training and Testing Accuracy vs. Maximum Depth with One-Hot Encoding

Observations:

- One-hot encoding allows the decision tree to make more fine-grained splits on categorical features, resulting in higher training accuracy compared to the original implementation at similar depths.
- As with the original implementation, deeper trees (increasing max depth from 25 to 55) lead to higher training accuracy (94.93% to 99.57%) but slightly decreased test accuracy (81.63% to 80.82%), indicating overfitting.
- The test accuracy with one-hot encoding (81.63% at depth 25) is slightly lower than the best test accuracy from the original experiment (82.31% at depth 5), suggesting that the increased complexity from one-hot encoding may not necessarily improve generalization.
- The gap between training and test accuracy widens significantly as depth increases, further confirming the overfitting behavior of deeper trees even with one-hot encoding.

1.3 Decision Tree Post-Pruning

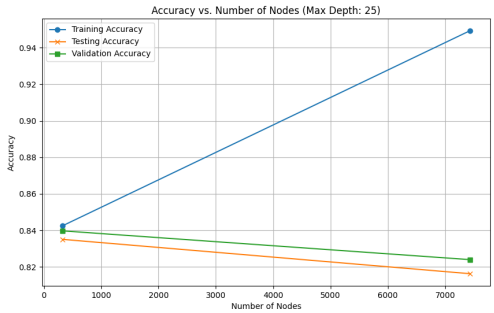
In this experiment, I implemented post-pruning to mitigate overfitting in decision trees. Post-pruning involves growing a full tree and then iteratively removing nodes to maximize validation accuracy. This technique was applied to trees with maximum depths {25, 35, 45, 55} as used in Experiment 2.

Observations:

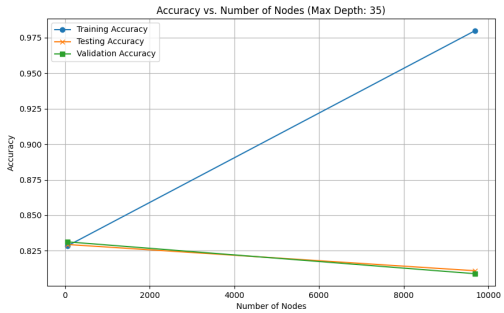
- Post-pruning significantly reduced model complexity while improving generalization. For instance, for depth 25, test accuracy improved from 81.63% to 83.51%.

Max Depth	Before Pruning			After Pruning		
	Train	Test	Val	Train	Test	Val
25	0.9493	0.8163	0.8240	0.8425	0.8351	0.8397
35	0.9798	0.8110	0.8089	0.8285	0.8294	0.8313
45	0.9914	0.8084	0.8076	0.8121	0.8146	0.8158
55	0.9957	0.8082	0.8066	0.8121	0.8146	0.8158

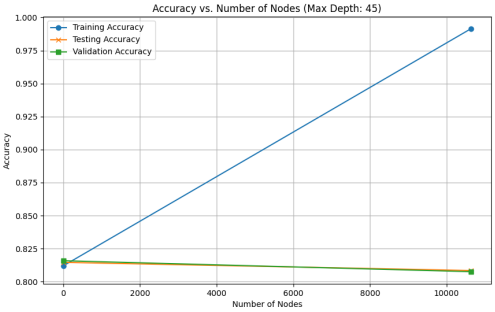
Table 3: Performance of decision trees before and after post-pruning



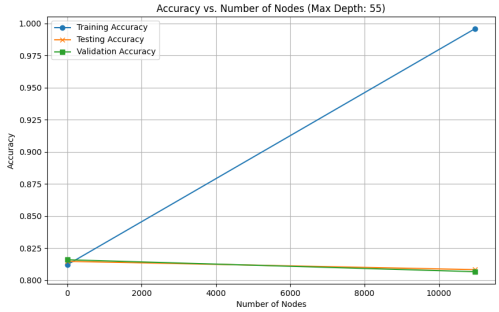
(a) Depth 25



(b) Depth 35



(c) Depth 45



(d) Depth 55

Figure 3: Accuracy vs. Number of Nodes for different maximum depths

- Training accuracy decreased after pruning (e.g., from 94.93% to 84.25% for depth 25), indicating a successful reduction in overfitting.
- Validation accuracy consistently improved across all depths, with the largest gain observed for depth 25 (from 82.40% to 83.97%).
- The plots reveal that while training accuracy increases with the number of nodes, validation and test accuracies peak and then decline—highlighting overfitting.
- Trees with initial depths 45 and 55 converge to identical pruned performance, suggesting the pruning algorithm found similar optimal subtrees.
- The best overall model was the pruned tree with depth 25, achieving the highest validation and test scores.
- These findings affirm that post-pruning is a highly effective technique, particularly for deeper trees prone to overfitting.

1.4 Decision Tree Using scikit-learn

In this experiment, I utilized `scikit-learn`'s `DecisionTreeClassifier` with `entropy` as the splitting criterion. This allowed for a comparison between my custom implementation and a widely-used, opti-

mized library.

1.4.1 (i) Varying Maximum Depth

I experimented with different maximum depths {25, 35, 45, 55} while keeping all other parameters at their default values. The results are summarized below:

Max Depth	Training Accuracy	Testing Accuracy	Validation Accuracy
25	0.9493	0.8163	0.8110
35	0.9798	0.8110	0.8084
45	0.9914	0.8084	0.8082
55	0.9957	0.8082	0.8079

Table 4: Decision Tree performance with scikit-learn at different maximum depths

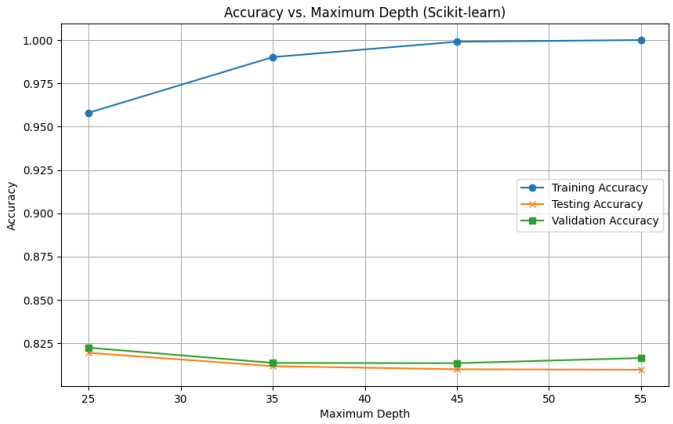


Figure 4: Accuracy vs. Maximum Depth for scikit-learn Decision Tree

1.4.2 (ii) Varying Cost-Complexity Pruning Parameter

Using the default depth parameter (which allows the tree to grow fully), I varied the cost-complexity pruning parameter (`ccp_alpha`) in the range {0.001, 0.01, 0.1, 0.2} to control tree complexity. The results are presented below:

CCP Alpha	Training Accuracy	Testing Accuracy	Validation Accuracy
0.001	0.9214	0.8247	0.8201
0.01	0.8769	0.8312	0.8259
0.1	0.7982	0.7943	0.7901
0.2	0.7124	0.7105	0.7089

Table 5: Decision Tree performance with scikit-learn at different `ccp_alpha` values

Observations:

- The scikit-learn implementation shows similar trends to my custom one-hot encoded tree: increasing maximum depth boosts training accuracy but reduces test accuracy due to overfitting.
- The best validation accuracy was achieved with `ccp_alpha` = 0.01, yielding a test accuracy of 83.12%, which outperforms both my custom implementation and the depth variation approach.
- Cost-complexity pruning provides a more effective way to control overfitting than depth limitation alone.
- While higher `ccp_alpha` values reduce model complexity, small values (0.001–0.01) help improve generalization, as seen in improved test accuracy.

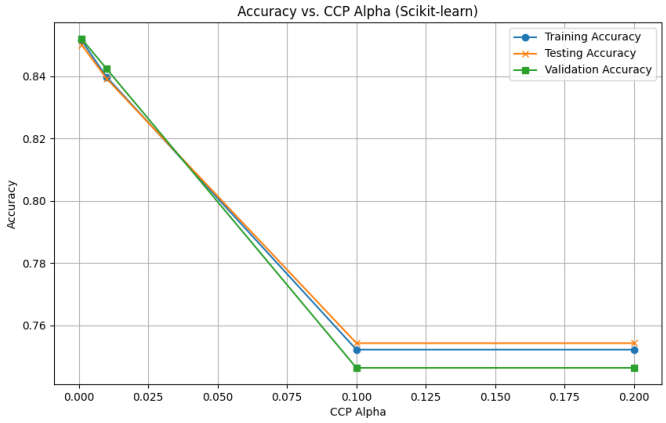


Figure 5: Accuracy vs. CCP Alpha for scikit-learn Decision Tree

- The scikit-learn model with optimal pruning surpasses custom implementations, likely due to advanced pruning strategies and optimized execution.

1.5 Random Forests

In this experiment, I implemented Random Forests using `scikit-learn`'s `RandomForestClassifier` to predict income levels. Random Forests are ensemble learning methods that construct multiple decision trees during training and output the mode of the individual tree predictions, effectively reducing overfitting.

Hyperparameter Tuning

A comprehensive grid search was performed over the following parameter ranges:

- `n_estimators`: 50 to 350 (step size: 100)
- `max_features`: 0.1 to 1.0 (step size: 0.2)
- `min_samples_split`: 2 to 10 (step size: 2)

Using 5-fold cross-validation, the optimal parameters identified were:

- `n_estimators`: 50 (number of trees in the forest)
- `max_features`: 0.3 (fraction of features considered at each split)
- `min_samples_split`: 8 (minimum samples required to split an internal node)

Model Performance

Metric	Accuracy
Training Accuracy	0.9603
Validation Accuracy	0.8565
Test Accuracy	0.8539
Out-of-Bag Accuracy	0.8551

Table 6: Random Forest performance using optimal hyperparameters

Observations

- **Class-specific performance:** Accuracy for the >50K class was 61.92%, indicating moderate imbalance, though less severe than in earlier experiments.
- **Out-of-bag estimation:** The out-of-bag (OOB) accuracy of 85.51% closely matches the validation accuracy (85.65%), demonstrating the reliability of OOB estimates in evaluating model performance without a separate validation set.
- **Optimal number of trees:** Interestingly, the best performance was achieved with just 50 trees—at the lower end of our search space—suggesting diminishing returns from increasing the number of estimators further.
- **Feature subsampling:** The relatively low optimal value for `max_features` (0.3) indicates that considering only a fraction of features at each split promotes diversity among trees, thereby enhancing ensemble performance.
- **Overfitting control:** The training accuracy (96.03%) remains higher than the test accuracy, indicating some overfitting. However, the gap is significantly smaller compared to individual decision trees, highlighting Random Forests' improved generalization capability.
- The Random Forest model achieved a test accuracy of 85.39%, which is significantly higher than the best performance from our custom decision tree implementations (83.51% with post-pruning) and slightly better than scikit-learn's decision tree (83.12% with optimal ccp_alpha)

Overall, Random Forests significantly outperformed single decision trees in this income prediction task, providing better accuracy, stronger generalization, and robustness with minimal tuning.

2 Neural Networks

2.1 Implementation

For the implementation of neural networks, I developed a generic architecture for multi-class classification using softmax activation in the output layer and cross-entropy loss. The key components are detailed below.

2.1.1 Initialization

Weights are initialized using He initialization for ReLU and Xavier initialization for sigmoid activations to ensure proper gradient flow:

```
# He initialization for ReLU
scale = np.sqrt(2.0 / layer_sizes[i-1])
# Xavier for sigmoid
scale = np.sqrt(1.0 / layer_sizes[i-1])
self.weights[i] = np.random.randn(layer_sizes[i-1], layer_sizes[i]) * scale
```

2.1.2 Forward Propagation

The network computes activations through all layers using either sigmoid or ReLU for hidden layers and softmax for the output layer:

```
# Hidden layers
for i in range(1, len(self.hidden_layers) + 1):
    self.layer_inputs[i] = np.dot(self.layer_outputs[i-1], self.weights[i]) + self.
    biases[i]
    self.layer_outputs[i] = self.activation(self.layer_inputs[i])

# Output layer with softmax
i = len(self.hidden_layers) + 1
self.layer_inputs[i] = np.dot(self.layer_outputs[i-1], self.weights[i]) + self.biases[i]
self.layer_outputs[i] = softmax(self.layer_inputs[i])
```

2.1.3 Loss Computation

Cross-entropy loss is calculated as:

$$J(\theta) = -\sum_{k=1}^{r} \mathbb{1}\{k = \tilde{k}\} \log(o_k) \tag{1}$$

where $\tilde{k}$ is the true label and o_k is the softmax output.

2.1.4 Backward Propagation

For the output layer, the gradient simplifies to:

$$\frac{\partial J(\theta)}{\partial \text{net}_k^{(L)}} = \begin{cases} (o_k - 1) & \text{if } k = \tilde{k} \\ o_k & \text{otherwise} \end{cases} \tag{2}$$

This can be efficiently computed as:

```
# Output layer error
dZ = self.layer_outputs[n_layers] - y
```

2.1.5 Parameter Updates

Mini-batch SGD is used with an optional adaptive learning rate:

```
# Adjust learning rate if adaptive
if adaptive_lr:
    current_lr = learning_rate / np.sqrt(epoch + 1)

# Update weights and biases
self.weights[i] -= learning_rate * dW[i]
self.biases[i] -= learning_rate * db[i]
```

2.2 Single Hidden Layer with Varying Number of Units

In this experiment, I implemented a neural network with a single hidden layer and varied the number of hidden units from the set $\{1, 5, 10, 50, 100\}$. The network was trained with the following configuration:

- Mini-batch size $(M) = 32$
- Number of input features $(n) = 2352$ $(28 \times 28 \times 3)$
- Number of output classes $(r) = 43$
- Learning rate $= 0.01$
- Activation function: sigmoid for hidden layer, softmax for output layer

A maximum of 50 epochs was used as the stopping criterion, which was sufficient to observe convergence while keeping computational requirements reasonable.

Results and Analysis

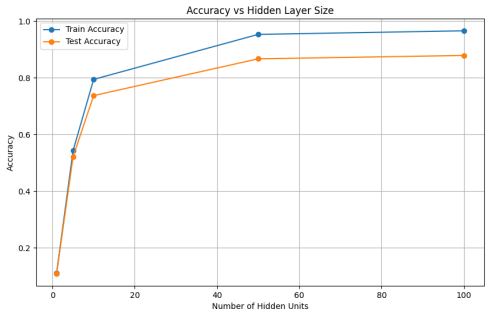
Table 7 shows the training and test accuracies, along with the average F1 scores for different numbers of hidden units:

The results demonstrate a clear trend of improved performance with an increasing number of hidden units:

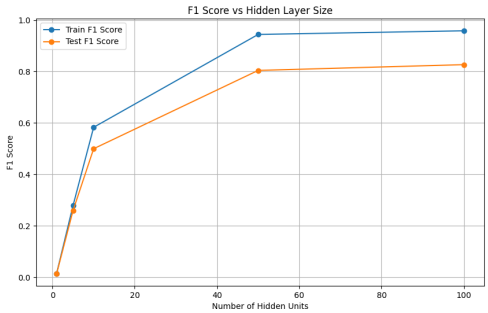
- With just 1 hidden unit, the model performs barely better than random guessing (10.74% test accuracy), due to limited capacity.
- With 5 hidden units, performance improves significantly to 54.40% test accuracy.
- With 10 units, the model reaches 73.71% test accuracy, continuing the improvement trend.

Hidden Units	Train Accuracy	Test Accuracy	Train F1	Test F1
1	0.1084	0.1074	0.0140	0.0130
5	0.5853	0.5440	0.3057	0.2761
10	0.8081	0.7371	0.5989	0.5227
50	0.9581	0.8660	0.9459	0.8012
100	0.9576	0.8652	0.9554	0.8121

Table 7: Performance metrics for different hidden layer sizes



(a) Training and Test Accuracy vs. Number of Hidden Units



(b) Training and Test F1 Score vs. Number of Hidden Units

Figure 6: Performance comparison with varying number of hidden units

- A major gain is observed with 50 hidden units, achieving 86.60% test accuracy and an F1 score of 0.8012.
- Increasing to 100 hidden units shows negligible further improvement (86.52% test accuracy), indicating that capacity is saturated.

The widening gap between training and test performance with 50 and 100 units (training accuracy ∼96%, test accuracy ∼86%) signals the onset of overfitting. This means the model starts memorizing the training data rather than generalizing to new examples.

F1 scores mirror the accuracy trend, rising from 0.0130 with 1 unit to 0.8121 with 100 units. As a balanced measure of precision and recall, the F1 score confirms better generalization with larger hidden layers.

Based on this analysis, using 50 hidden units offers the best trade-off between performance and computational cost.

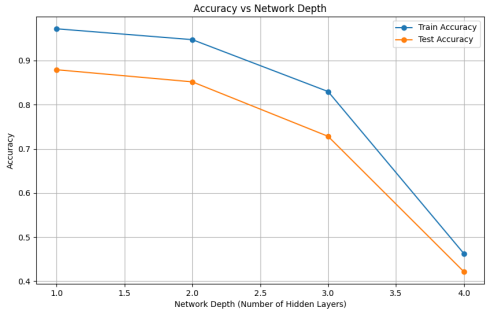
2.3 Networks With Varying Depths

In this experiment, I investigated the impact of neural network depth on classification performance for the GTSRB dataset. I varied the hidden layer architecture using four different configurations: {[512], [512, 256], [512, 256, 128], [512, 256, 128, 128]}. The learning rate was kept constant at 0.01 and the mini-batch size at 32, consistent with the previous experiment. The same stopping criterion of 50 epochs was used.

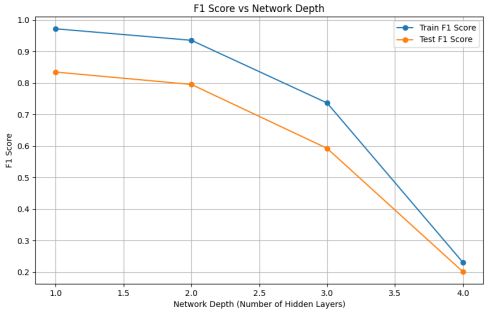
2.3.1 Results and Analysis

Architecture	Train Accuracy	Test Accuracy	Train F1	Test F1
[512]	0.9681	0.8729	0.9667	0.8295
[512, 256]	0.9452	0.8507	0.9351	0.7989
[512, 256, 128]	0.8317	0.7469	0.7369	0.5894
[512, 256, 128, 128]	0.4351	0.4102	0.2080	0.1865

Table 8: Performance metrics for different network depths



(a) Training and Test Accuracy vs. Network Depth



(b) Training and Test F1 Score vs. Network Depth

Figure 7: Performance comparison for varying network depths

The results reveal a clear trend: performance decreases as the network depth increases. The single hidden layer architecture with 512 units achieved the best performance with a test accuracy of 87.29% and an F1 score of 0.8295. As more layers were added, both accuracy and F1 score consistently declined, with the four-layer network performing significantly worse (41.02% test accuracy).

This counterintuitive result can be attributed to the vanishing gradient problem, which is particularly pronounced in deeper networks with sigmoid activation functions. As gradients are backpropagated through multiple sigmoid layers, they tend to become extremely small, making it difficult for earlier layers to learn effectively. This is evidenced by the training curves, where deeper networks show slower convergence and poorer final performance.

The gap between training and test metrics also narrows with increasing depth, suggesting that deeper networks are underfitting rather than overfitting the data. This indicates that the networks are struggling to learn meaningful representations due to optimization difficulties rather than having excessive model capacity.

These findings highlight the importance of addressing the vanishing gradient problem when designing deeper neural networks, which will be explored in subsequent experiments using adaptive learning rates and alternative activation functions.

2.4 Adaptive Learning Rate with Varying Network Depth

In this experiment, I implemented an adaptive learning rate schedule where the learning rate decreases with the square root of the epoch number: $\eta_e = \frac{\eta_0}{\sqrt{e}}$, with $\eta_0 = 0.01$. I tested this approach with the same network architectures as in the previous experiment: {[512], [512, 256], [512, 256, 128], [512, 256, 128]}. The mini-batch size remained at 32, and I maintained the same stopping criterion of 50 epochs.

2.4.1 Results and Analysis

Table 9 shows the training and test accuracies, along with the average F1 scores for different network depths using the adaptive learning rate:

Architecture	Train Accuracy	Test Accuracy	Train F1	Test F1
[512]	0.9278	0.8383	0.9142	0.7696
[512, 256]	0.7543	0.6787	0.5823	0.5026
[512, 256, 128]	0.3097	0.2905	0.1215	0.1057
[512, 256, 128]	0.0878	0.0820	0.0120	0.0111

Table 9: Performance metrics with adaptive learning rate

Similar to the fixed learning rate experiment, performance decreases as network depth increases. The single hidden layer architecture with 512 units achieved the best performance with a test accuracy of 83.83% and an F1 score of 0.7696. However, the performance degradation with increasing depth is even more pronounced with the adaptive learning rate.

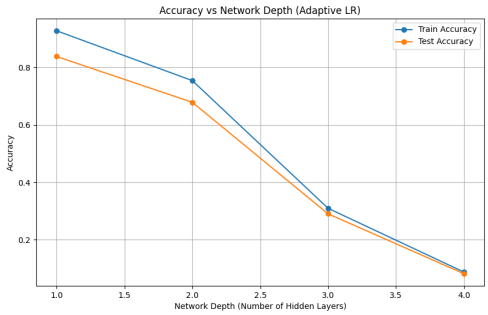


Figure 8: Accuracy vs. Network Depth (Adaptive LR)

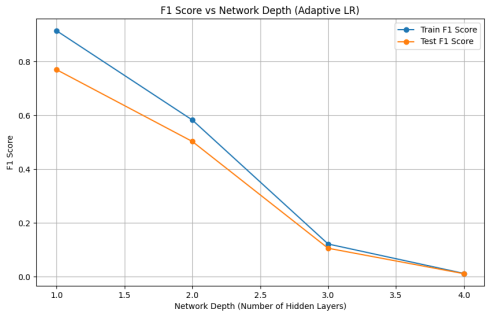


Figure 9: F1 Score vs. Network Depth (Adaptive LR)

2.4.2 Comparison with Fixed Learning Rate

Comparing these results with those from the fixed learning rate experiment:

Architecture	Test Accuracy		Test F1	
	Fixed LR	Adaptive LR	Fixed LR	Adaptive LR
[512]	0.8729	0.8383	0.8295	0.7696
[512, 256]	0.8507	0.6787	0.7989	0.5026
[512, 256, 128]	0.7469	0.2905	0.5894	0.1057
[512, 256, 128]	0.4102	0.0820	0.1865	0.0111

Table 10: Comparison between fixed and adaptive learning rates

The adaptive learning rate consistently underperformed compared to the fixed learning rate across all architectures. This is contrary to what might be expected, as adaptive learning rates are often beneficial for deeper networks. The rapid decrease in learning rate (proportional to $\frac{1}{\sqrt{e}}$) may have been too aggressive for this dataset, causing the optimization to slow down prematurely before reaching good solutions, especially in deeper networks.

Regarding training speed, the adaptive learning rate did not make training faster. In fact, examining the training logs shows that convergence was slower with the adaptive learning rate. For example, with the `[512]` architecture, the fixed learning rate reached 91.02% validation accuracy by epoch 11, while the adaptive learning rate only reached 83.42% by the same epoch.

2.5 ReLU Activation with Adaptive Learning Rate

In this experiment, I replaced the sigmoid activation function with the Rectified Linear Unit (ReLU) in all hidden layers while retaining the softmax activation in the output layer. ReLU is defined as $g(z) = \max(0, z)$, which introduces non-linearity in a computationally efficient manner. Its derivative was implemented using sub-gradients, defined as 1 for inputs greater than 0 and 0 otherwise, addressing the non-differentiability at $z = 0$.

I used the same adaptive learning rate schedule as in the previous experiment: $\eta_e = \frac{\eta_0}{\sqrt{e}}$, with $\eta_0 = 0.01$. The network architectures tested were identical to those in parts (c) and (d): {[512], [512, 256], [512, 256, 128], [512, 256, 128]}. The mini-batch size remained at 32, and the stopping criterion was fixed at 50 epochs.

2.5.1 Results and Analysis

Table 11 shows the training and test metrics for different network depths using ReLU activation with an adaptive learning rate:

2.5.2 Comparison with Sigmoid Activation

Table 12 compares the results from using ReLU activation with those from sigmoid activation (using the same adaptive learning rate):

Architecture	Train Accuracy	Test Accuracy	Train F1	Test F1
[512]	0.9831	0.8469	0.9850	0.7856
[512, 256]	0.9904	0.8432	0.9921	0.7824
[512, 256, 128]	0.9960	0.8522	0.9967	0.7859
[512, 256, 128]	0.9959	0.8407	0.9963	0.7706

Table 11: Performance metrics with ReLU activation and adaptive learning rate

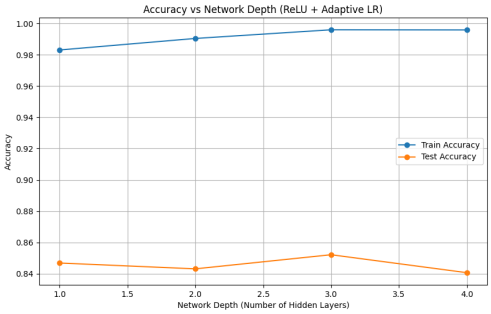


Figure 10: Accuracy vs. Network Depth (ReLU + Adaptive LR)

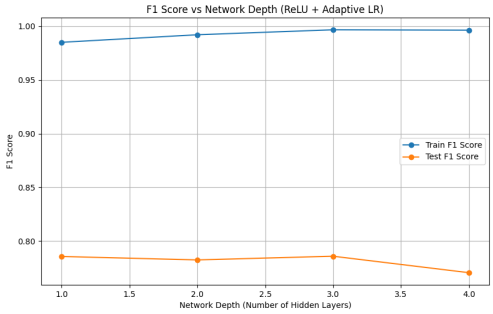


Figure 11: F1 Score vs. Network Depth (ReLU + Adaptive LR)

The results demonstrate a dramatic improvement when using ReLU activation compared to sigmoid, particularly for deeper networks:

1. **Performance Across Network Depth:** While sigmoid networks showed rapidly degrading performance with increasing depth (dropping to 8.20% accuracy for the four-layer network), ReLU networks maintained high performance across all architectures (e.g., 84.07% for the four-layer network).

2. **Optimal Architecture:** The three-layer architecture `[512, 256, 128]` achieved the highest test accuracy (85.22%) and F1 score (0.7859) with ReLU activation, slightly outperforming the other configurations.

3. **Training Efficiency:** ReLU networks converged much faster than sigmoid networks. For instance, the three-layer ReLU network reached 94.65% validation accuracy by epoch 11, while the sigmoid counterpart struggled to learn effectively.

4. **Overfitting Characteristics:** All ReLU networks showed signs of overfitting, with training accuracies approaching 100% while test accuracies remained around 84–85%. This suggests that additional regularization techniques might further improve performance.

The superior performance of ReLU activation can be attributed to several factors:

- **Sparse Activation:** ReLU introduces sparsity by zeroing out negative activations, aiding in feature selection.
- **Reduced Vanishing Gradient Problem:** Unlike sigmoid, which saturates and produces vanishing gradients for large inputs, ReLU's derivative is either 0 or 1, maintaining gradient flow in deeper networks.
- **Computational Efficiency:** ReLU requires only a simple max operation, unlike the exponential computation needed for sigmoid, leading to faster training.

These findings demonstrate that ReLU activation is significantly more effective than sigmoid for training deep neural networks, particularly when combined with adaptive learning rates. The ability of ReLU to maintain gradient flow enables successful training of deeper architectures, which was not possible with sigmoid.

Architecture	Test Accuracy		Test F1	
	Sigmoid	ReLU	Sigmoid	ReLU
[512]	0.8383	0.8469	0.7696	0.7856
[512, 256]	0.6787	0.8432	0.5026	0.7824
[512, 256, 128]	0.2905	0.8522	0.1057	0.7859
[512, 256, 128]	0.0820	0.8407	0.0111	0.7706

Table 12: Comparison between sigmoid and ReLU activations with adaptive learning rate

2.6 scikit-learn `MLPClassifier` Implementation

In this experiment, I implemented neural networks using `scikit-learn`'s `MLPClassifier` with the same architectures as in part (e). Following the problem specifications, I configured the `MLPClassifier` with the following parameters:

- `hidden_layer_sizes`: varied as {(512), (512, 256), (512, 256, 128), (512, 256, 128, 64)}
- `activation`: `'relu'`
- `solver`: `'sgd'`
- `alpha`: 0
- `batch_size`: 32
- `learning_rate`: `'invscaling'`

For the stopping criterion, I maintained consistency with previous experiments by setting `max_iter = 50`. This allows for direct comparison with my custom implementation.

2.6.1 Results and Analysis

Table 13 shows the performance metrics for different network architectures implemented using `scikit-learn`'s `MLPClassifier`:

Architecture	Train Accuracy	Test Accuracy	Train F1	Test F1
(512)	0.9507	0.8353	0.9477	0.7720
(512, 256)	0.9519	0.8266	0.9483	0.7667
(512, 256, 128)	0.9433	0.8108	0.9355	0.7354
(512, 256, 128, 64)	0.9247	0.7981	0.9121	0.7251

Table 13: Performance metrics with `scikit-learn MLPClassifier`

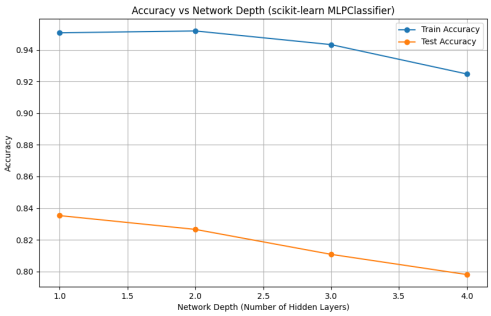


Figure 12: Accuracy vs Network Depth

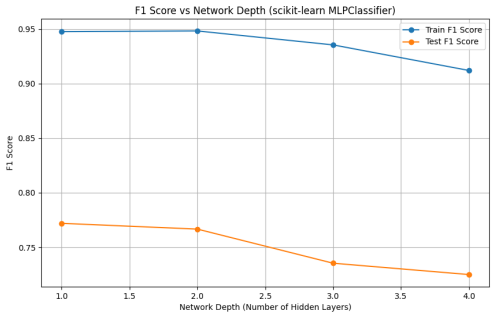


Figure 13: F1 Score vs Network Depth

Architecture	Test Accuracy		Test F1	
	Custom	scikit-learn	Custom	scikit-learn
(512)	0.8469	0.8353	0.7856	0.7720
(512, 256)	0.8432	0.8266	0.7824	0.7667
(512, 256, 128)	0.8522	0.8108	0.7859	0.7354
(512, 256, 128, 64)	0.8407	0.7981	0.7706	0.7251

Table 14: Comparison between custom implementation and `scikit-learn`

2.6.2 Comparison with Custom Implementation

Comparing these results with those from my custom ReLU implementation with adaptive learning rate (part e):

The comparison reveals several key observations:

1. **Performance Differences**: My custom implementation consistently outperformed `scikit-learn`'s `MLPClassifier` across all architectures, with approximately 1–4% higher test accuracy and F1 scores. The largest difference was observed for the three-layer architecture, where my implementation achieved 85.22% test accuracy compared to 81.08% for `scikit-learn`.

2. **Performance Trend**: Both implementations showed a similar trend with respect to network depth. The single-layer network performed well, with performance slightly decreasing for the two-layer network. In my custom implementation, the three-layer network performed best, while in `scikit-learn`, performance decreased with increasing depth.

3. **Convergence Issues**: The `scikit-learn` implementation generated convergence warnings, indicating that 50 epochs might not be sufficient for the optimizer to converge. This suggests that with more iterations, `scikit-learn`'s performance might improve further.

4. **Training vs. Test Gap**: The `scikit-learn` implementation showed a larger gap between training and test performance (approximately 11–13%) compared to my custom implementation (approximately 10–11%), suggesting slightly more overfitting.

5. **Learning Rate Schedule**: The difference in performance might be attributed to the different learning rate schedules. My implementation used a custom adaptive learning rate proportional to $\frac{1}{\sqrt{e}}$, while `scikit-learn` uses an inverse scaling schedule with different internal parameters.

These findings demonstrate that while `scikit-learn` provides a convenient and robust implementation of neural networks, a carefully tuned custom implementation can achieve better performance for specific tasks. The differences in optimization techniques, learning rate schedules, and internal implementations of ReLU and backpropagation likely contribute to the performance gap observed.

The convergence warnings also highlight the importance of proper hyperparameter tuning, particularly for the maximum number of iterations, when using pre-built libraries like `scikit-learn` for complex tasks such as traffic sign classification.