AIML Lab Report: MLP Optimization Analysis

Assem Chebly, Youcef bellouche, Mahmoud El Sayed

November 30, 2025

1 Introduction

This report analyzes the performance of Multi-Layer Perceptrons (MLPs) on the Fashion MNIST dataset. We systematically evaluate the impact of four key hyperparameters: Network Depth (Layers), Optimizer choice, Learning Rate, and Batch Size.

2 Methodology

We utilized a standard MLP architecture with ReLU activation. All experiments were conducted on Google Colab using Keras. In each test section, one parameter was varied while others remained fixed to the baseline configuration (4 Layers, Adam, LR=0.001, Batch=64).

3 Evaluation and Results

3.1 Test 1: Impact of Network Depth (Layers)

In this section, we evaluated the network performance with 4, 6, 8, and 12 hidden layers to understand the effect of depth on learning.

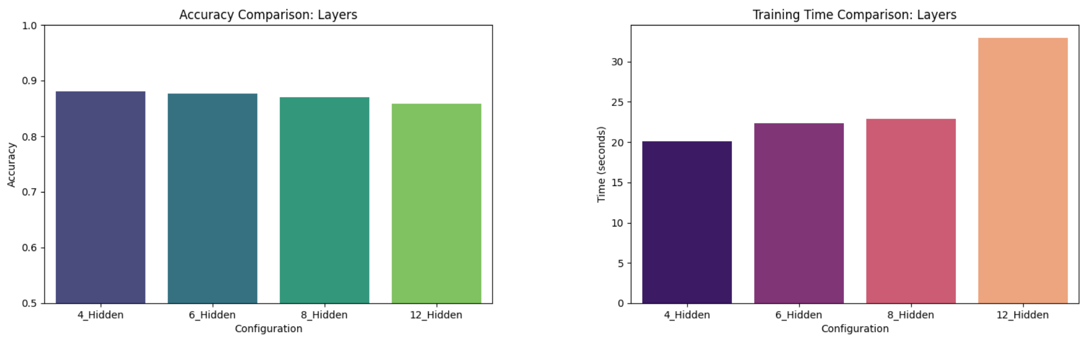

Figure 1: Accuracy (Left) and Training Time (Right) vs Number of Layers

Observation: The shallowest network (4 Hidden Layers) achieved the highest accuracy of **88.02%**. Surprisingly, increasing the depth to 12 layers resulted in a performance drop to 85.90%, while increasing the training time from 20s to 33s.

Analysis & Conclusion: This result highlights the **Vanishing Gradient Problem**. As the network gets deeper without residual connections (skip-connections), the gradients used to update the weights become increasingly small as they backpropagate to the earlier layers. This makes the initial layers learn very slowly. Furthermore, for a dataset as simple as Fashion MNIST, a 12-layer model introduces unnecessary complexity, leading to **overfitting** where the model struggles to generalize.

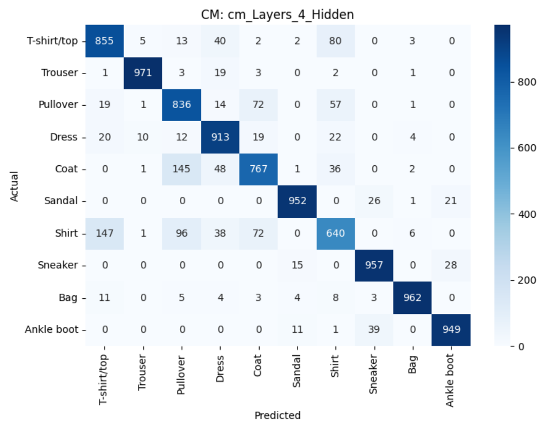

Figure 2: CM: 4 Layers (Best Model)

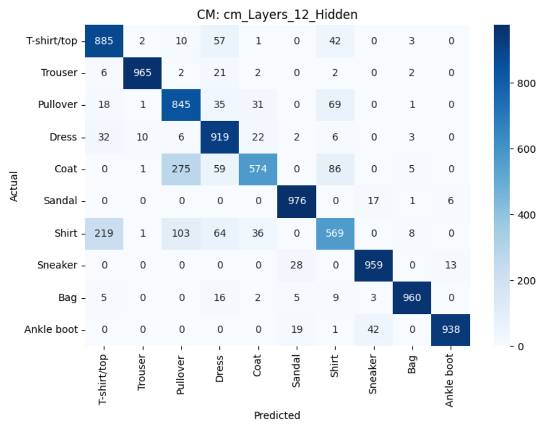

Figure 3: CM: 12 Layers (Over-complex)

3.2 Test 2: Impact of Optimizer Choice

We compared the two most fundamental optimizers in deep learning: **Adam** (Adaptive Moment Estimation) and **SGD** (Stochastic Gradient Descent).

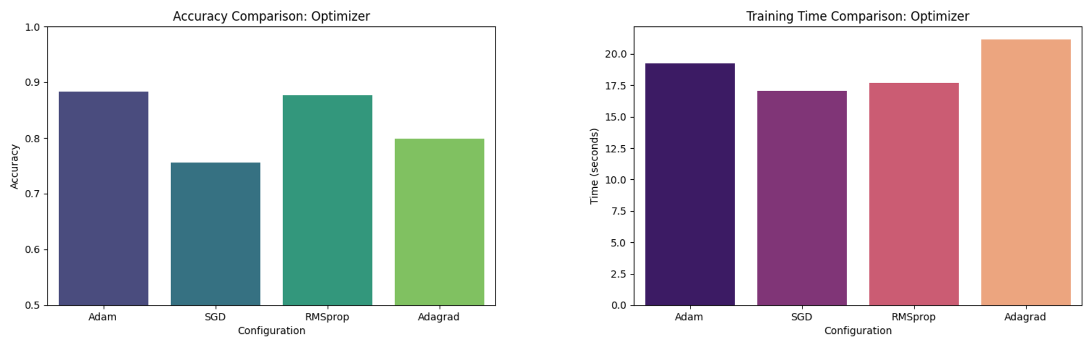

Figure 4: Performance Comparison: Adam vs SGD

Observation: There is a drastic performance gap. **Adam** reached an accuracy of **88.33%** within 10 epochs, whereas **SGD** only achieved **75.64%** under identical conditions.

Analysis & Conclusion: The superior performance of Adam is due to its **Adaptive Learning Rate**. Unlike SGD, which applies a single fixed learning rate to all weights, Adam calculates individual learning rates for different parameters and incorporates ”momentum.” This allows it to navigate the loss landscape much more efficiently, escaping local minima and converging quickly. SGD, lacking these adaptive features, requires significantly more epochs or careful tuning to reach comparable results.

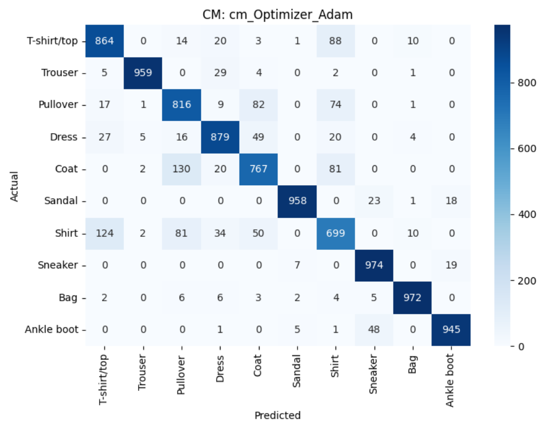
Figure 5: CM: Adam (High Accuracy)

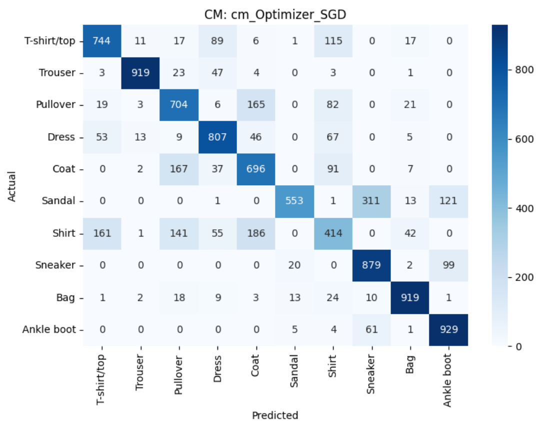
Figure 6: CM: SGD (Low Accuracy)

3.3 Test 3: Impact of Learning Rate

We tested the model's sensitivity to learning rates of 0.0001, 0.001, 0.01, and 0.1.

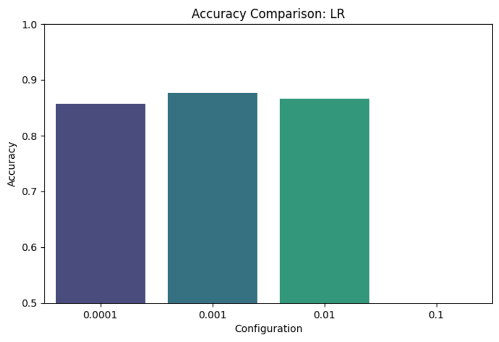
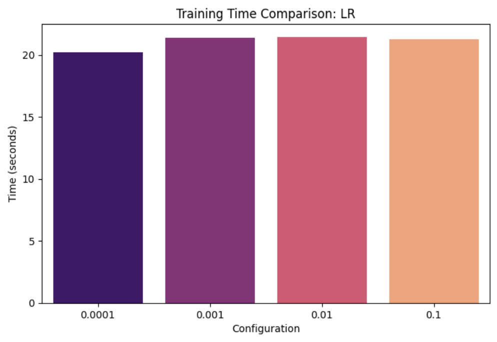
Figure 7: Impact of Learning Rate on Accuracy and Time

Observation: The optimal learning rate was **0.001**. However, setting the learning rate to **0.1** caused the model to fail completely, resulting in 10% accuracy (random guessing).

Analysis & Conclusion: The failure at 0.1 is caused by **Gradient Overshooting**. When the learning rate is too large, the weight updates are massive, causing the optimizer to "overshoot" the minimum point of the loss function. The model oscillates violently or diverges, never settling on a solution. This confirms that the learning rate is the most sensitive hyperparameter; a small change can mean the difference between state-of-the-art performance and complete model failure.

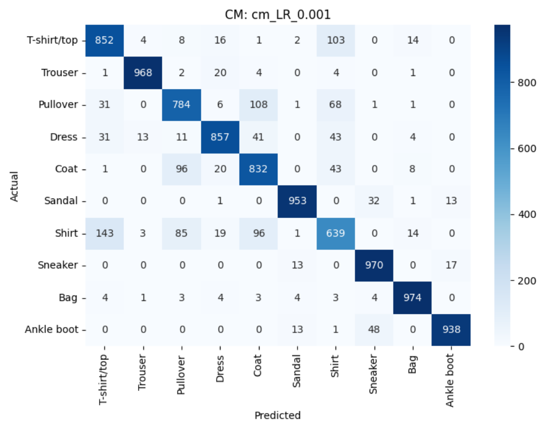
Figure 8: CM: LR 0.001 (Stable)

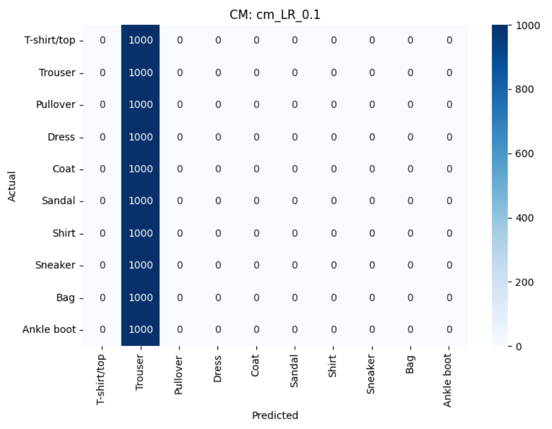
Figure 9: CM: LR 0.1 (Exploding Gradient)

3.4 Test 4: Impact of Batch Size

We evaluated batch sizes of 32, 64, 128, and 256 to test computational efficiency.

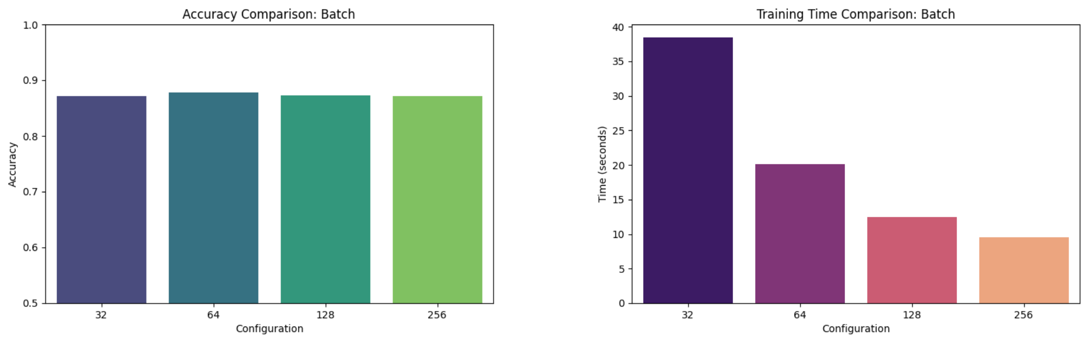

Figure 10: Impact of Batch Size on Accuracy and Time

Observation: Accuracy remained relatively stable across all sizes (87%), but the training time dropped significantly. Using a batch size of 256 was roughly **4x faster** (9.6s) than using a batch size of 32 (38.4s).

Analysis & Conclusion: This speedup is due to **Hardware Parallelization**. Modern GPUs are designed to perform matrix operations on large blocks of data simultaneously. By increasing the batch size, we feed more data to the GPU at once, maximizing its utilization. While smaller batch sizes (like 32) theoretically offer a "noisy" gradient that can help generalization, for this task, the computational efficiency of the larger batch size far outweighed the negligible difference in accuracy.

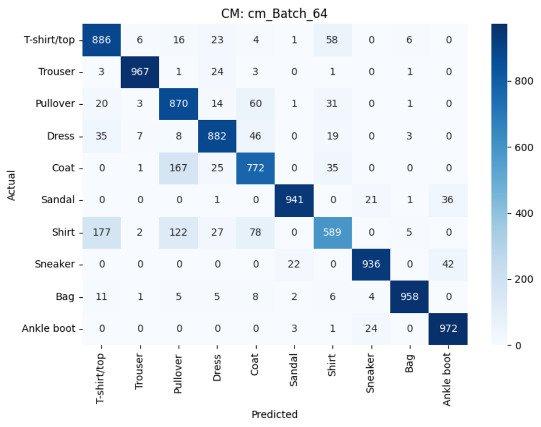

Figure 11: CM: Batch 64

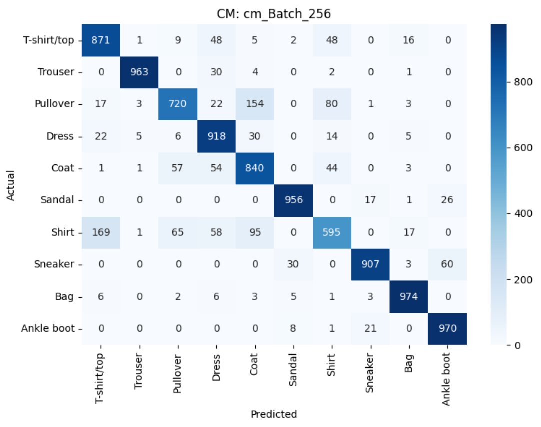

Figure 12: CM: Batch 256 (Fastest)

Category	**Variation**	**Accuracy**	**Time (s)**
Layers	4 Hidden	0.8802	20.06
Layers	6 Hidden	0.8765	22.33
Layers	8 Hidden	0.8701	22.83
Layers	12 Hidden	0.8590	32.89
Optimizer	Adam	0.8833	19.27
Optimizer	SGD	0.7564	17.06
LR	0.0001	0.8566	20.21
LR	0.001	0.8767	21.42
LR	0.01	0.8667	21.44
LR	0.1	0.1000	21.25
Batch	32	0.8712	38.41
Batch	64	0.8773	20.08
Batch	128	0.8724	12.51
Batch	256	0.8714	9.58

Table 1: Complete Experimental Results

4 Conclusion

Our experiments on the Fashion MNIST dataset lead to the following conclusions:

- **Efficiency:** The most significant efficiency gain came from increasing the **Batch Size** to 256.
- **Stability:** The **Adam** optimizer provided the most stable baseline.
- **Sensitivity:** The model is highly sensitive to Learning Rate; a value of 0.1 resulted in complete failure.

```python
import tensorflow as keras
from tensorflow.keras import layers, models, optimizers
from tensorflow.keras.datasets import fashion_mnist
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
import time
import os
```

```python
# 1. Setup & Data Loading

if not os.path.exists('lab_results'):
    os.makedirs('lab_results')

# Load Fashion MNIST
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

# Normalize
x_train, x_test = x_train / 255.0, x_test / 255.0

# Flatten
x_train_flat = x_train.reshape((-1, 784))
x_test_flat = x_test.reshape((-1, 784))

class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

# Global Results List
results = []

# 2. Helper Functions
# -------------------
def build_model(num_layers=4, neurons=128, activation='relu'):
    model = models.Sequential()
    model.add(layers.InputLayer(input_shape=(784,)))
    for _ in range(num_layers):
        model.add(layers.Dense(neurons, activation=activation))
    model.add(layers.Dense(10, activation='softmax'))
    return model

def save_confusion_matrix(model, x_test, y_test, filename):
    y_pred = np.argmax(model.predict(x_test), axis=1)
    cm = confusion_matrix(y_test, y_pred)

    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=class_names, yticklabels=class_names)
    plt.title(f'CM: {filename}')
    plt.ylabel('Actual')
    plt.xlabel('Predicted')
    plt.tight_layout()
    plt.savefig(f'lab_results/{filename}.png')
    plt.close()
```

```
def run_experiment(exp_group, variation_name, num_layers=4, neurons=64,
                   activation='relu', optimizer_name='adam', lr=0.001,
                   batch_size=64, epochs=10):

    full_name = f"{exp_group}_{variation_name}"
    print(f"Running: {full_name}...")

    # Build
    model = build_model(num_layers, neurons, activation)

    # Optimizer
    if optimizer_name == 'adam': opt = optimizers.Adam(learning_rate=lr)
    elif optimizer_name == 'sgd': opt = optimizers.SGD(learning_rate=lr)
    elif optimizer_name == 'rmsprop': opt = optimizers.RMSprop(learning_rate=lr)
    elif optimizer_name == 'adagrad': opt = optimizers.Adagrad(learning_rate=lr)

    model.compile(optimizer=opt, loss='sparse_categorical_crossentropy', metrics=['accuracy'])

    # Train
    start = time.time()
    history = model.fit(x_train_flat, y_train, epochs=epochs,
                        batch_size=batch_size, validation_split=0.1, verbose=0)
    duration = time.time() - start

    # Evaluate
    loss, acc = model.evaluate(x_test_flat, y_test, verbose=0)

    # Save Data
    results.append({
        'Group': exp_group,
        'Variation': variation_name,
        'Accuracy': acc,
        'Time': duration,
        'Loss': loss
    })

    # Save Artifacts
    save_confusion_matrix(model, x_test_flat, y_test, f"cm_{full_name}")
    model.save(f"lab_results/model_{full_name}.h5")
    print(f"   -> Acc: {acc:.4f} | Time: {duration:.2f}s")
```

```
# 3. Running Experiments (4 Groups, ~4 tests each)


# Group 1: Impact of Layers (Depth)

run_experiment('Layers', '4_Hidden', num_layers=4)
run_experiment('Layers', '6_Hidden', num_layers=6)
run_experiment('Layers', '8_Hidden', num_layers=8)
run_experiment('Layers', '12_Hidden', num_layers=12)

# Group 2: Impact of Optimizer

print("\n--- GROUP 2: OPTIMIZERS ---")
run_experiment('Optimizer', 'Adam', optimizer_name='adam')
run_experiment('Optimizer', 'SGD', optimizer_name='sgd')
run_experiment('Optimizer', 'RMSprop', optimizer_name='rmsprop')
run_experiment('Optimizer', 'Adagrad', optimizer_name='adagrad')
```

```
# Group 3: Impact of Learning Rate

print("\n--- GROUP 3: LEARNING RATE ---")
run_experiment('LR', '0.0001', lr=0.0001)
run_experiment('LR', '0.001', lr=0.001)
run_experiment('LR', '0.01', lr=0.01)
run_experiment('LR', '0.1', lr=0.1)

# Group 4: Impact of Batch Size

print("\n--- GROUP 4: BATCH SIZE ---")
run_experiment('Batch', '32', batch_size=32)
run_experiment('Batch', '64', batch_size=64)
run_experiment('Batch', '128', batch_size=128)
run_experiment('Batch', '256', batch_size=256)
```

```
  warnings.warn(
   -> Acc: 0.8763 | Time: 24.69s
Running: Optimizer_Adagrad...
```

```
Running: Batch_128...
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/input_layer.py:27: UserWarning: Ar
  warnings.warn(
   -> Acc: 0.8754 | Time: 16.99s
Running: Batch_256...
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_
   -> Acc: 0.8589 | Time: 13.02s
```

```
# 4. Generate Comparison Plots per Group

df = pd.DataFrame(results)
groups = df['Group'].unique()

for group in groups:
    subset = df[df['Group'] == group]

    # Plot Acc
    plt.figure(figsize=(8, 5))
    sns.barplot(x='Variation', y='Accuracy', data=subset, palette='viridis')
    plt.title(f'Accuracy Comparison: {group}')
    plt.ylim(0.5, 1.0)
    plt.ylabel('Accuracy')
    plt.xlabel('Configuration')
    plt.savefig(f'lab_results/plot_{group}_acc.png')
    plt.close()

    # Plot Time
    plt.figure(figsize=(8, 5))
    sns.barplot(x='Variation', y='Time', data=subset, palette='magma')
    plt.title(f'Training Time Comparison: {group}')
    plt.ylabel('Time (seconds)')
    plt.xlabel('Configuration')
    plt.savefig(f'lab_results/plot_{group}_time.png')
    plt.close()

# Save final CSV table for the report
df.to_csv('lab_results/final_results.csv', index=False)
print("\nDone! All images and models saved in 'lab_results'.")
print(df[['Group', 'Variation', 'Accuracy', 'Time']].to_markdown())
```

```
/tmp/ipython-input-1034548255.py:11: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign t

  sns.barplot(x='Variation', y='Accuracy', data=subset, palette='viridis')
/tmp/ipython-input-1034548255.py:21: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign t

  sns.barplot(x='Variation', y='Time', data=subset, palette='magma')
/tmp/ipython-input-1034548255.py:11: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign t

  sns.barplot(x='Variation', y='Accuracy', data=subset, palette='viridis')
/tmp/ipython-input-1034548255.py:21: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign t

  sns.barplot(x='Variation', y='Time', data=subset, palette='magma')
```

```
/tmp/ipython-input-1034548255.py:11: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign t

  sns.barplot(x='Variation', y='Accuracy', data=subset, palette='viridis')
/tmp/ipython-input-1034548255.py:21: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign t

  sns.barplot(x='Variation', y='Time', data=subset, palette='magma')
/tmp/ipython-input-1034548255.py:11: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign t

  sns.barplot(x='Variation', y='Accuracy', data=subset, palette='viridis')
/tmp/ipython-input-1034548255.py:21: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign t

  sns.barplot(x='Variation', y='Time', data=subset, palette='magma')

Done! All images and models saved in 'lab_results'.
|    | Group     | Variation   |   Accuracy |    Time |
|---:|:----------|:------------|-----------:|--------:|
|  0 | Layers    | 4_Hidden    |     0.8756 | 29.6887 |
|  1 | Layers    | 6_Hidden    |     0.8782 | 33.5621 |
|  2 | Layers    | 8_Hidden    |     0.8776 | 32.2006 |
|  3 | Layers    | 12_Hidden   |     0.8708 | 39.9772 |
|  4 | Optimizer | Adam        |     0.8768 | 25.9653 |
|  5 | Optimizer | SGD         |     0.7727 | 24.8559 |
|  6 | Optimizer | RMSprop     |     0.8763 | 24.6866 |
|  7 | Optimizer | Adagrad     |     0.7998 | 25.255  |
|  8 | LR        | 0.0001      |     0.8614 | 27.0555 |
|  9 | LR        | 0.001       |     0.8696 | 26.4387 |
| 10 | LR        | 0.01        |     0.8571 | 25.7227 |
| 11 | LR        | 0.1         |     0.4552 | 28.5376 |
| 12 | Batch     | 32          |     0.8769 | 49.6917 |
```

```
import shutil

output_filename = 'lab_results_compressed'
shutil.make_archive(output_filename, 'zip', 'lab_results')

print(f"Folder 'lab_results' compressed into '{output_filename}.zip'")
```

```
Folder 'lab_results' compressed into 'lab_results_compressed.zip'
```