# 3_Optimization_Experiments

April 25, 2025

[37]:

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
import matplotlib.pyplot as plt
import numpy as np
import sys
import os
import time
import copy
from sklearn.model_selection import KFold
from torch_lr_finder import LRFinder
import importlib
import inspect

# --- Add src to path ---
module_path = os.path.abspath(os.path.join('..', 'src'))
if module_path not in sys.path:
    sys.path.append(module_path)

# --- Import necessary modules ---
try:
    from utils import load_processed_data
    from models import Model_1, Model_2, Model_3, ACTIVATION_FUNCTIONS #␣
↪Assuming revised models.py
except ImportError as e:
    print(f"Initial import failed: {e}. Ensure src is in path and files exist.")
    raise
```

[38]:

```python
# --- Configuration ---
BATCH_SIZE = 128
INITIAL_MODEL_CLASS = Model_2 # From Part 2
SEED = 42
PLOT_SAVE_DIR = "../results/plots/"
N_EPOCHS_VERIFY = 5
K_FOLDS = 5
N_EPOCHS_KFOLD = 10
```

```
WEIGHT_DECAY_VALUES = [0, 1e-5, 1e-4, 1e-3, 1e-2]
N_EPOCHS_COMPONENT_TEST = 15

os.makedirs(PLOT_SAVE_DIR, exist_ok=True)
```

[39]:
```
# --- Set Seed ---
torch.manual_seed(SEED)
np.random.seed(SEED)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(SEED)

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "mps" if torch.
 ↪backends.mps.is_available() else "cpu")
print(f"Using device: {DEVICE}")
print(f"Reproducibility seed set to: {SEED}")
```

```
Using device: mps
Reproducibility seed set to: 42
```

[40]:
```
# --- Load Data ---
print("\nLoading data...")
X_train, y_train, X_val, y_val, X_test, y_test = load_processed_data()
print("Data loaded.")

# Create datasets and dataloaders
train_dataset = TensorDataset(X_train, y_train)
val_dataset = TensorDataset(X_val, y_val)
test_dataset = TensorDataset(X_test, y_test)
g = torch.Generator()
g.manual_seed(SEED)
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True,␣
 ↪generator=g)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE * 2)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE * 2)
```

```
2025-04-25 14:47:09,359 - INFO - Loading data from ../data/processed/…
2025-04-25 14:47:09,394 - INFO - Processed data loaded successfully.
2025-04-25 14:47:09,395 - INFO -   Train shapes: X=torch.Size([372336, 90]),
y=torch.Size([372336])
2025-04-25 14:47:09,395 - INFO -   Val shapes: X=torch.Size([71504, 90]),
y=torch.Size([71504])
2025-04-25 14:47:09,395 - INFO -   Test shapes: X=torch.Size([71505, 90]),
y=torch.Size([71505])


Loading data…
Data loaded.
```

[41]:
```
# --- Define Evaluation Function ---
def evaluate(model, loader, criterion, device):
    model.eval()
    total_loss = 0.0
    correct_predictions = 0
    total_samples = 0
    with torch.no_grad():
        for batch in loader:
            inputs, targets = batch
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            total_loss += loss.item() * inputs.size(0)
            _, predicted = torch.max(outputs.data, 1)
            total_samples += targets.size(0)
            correct_predictions += (predicted == targets).sum().item()
    if total_samples == 0: return 0.0, 0.0
    avg_loss = total_loss / total_samples
    accuracy = correct_predictions / total_samples
    return avg_loss, accuracy
```

[42]:
```
# --- Helper function for component training loop ---
def run_component_training(model, optimizer_class, criterion, train_loader,␣
↪val_loader,
                           lr, wd, epochs, device, model_name="Model"):
    optimizer = optimizer_class(model.parameters(), lr=lr, weight_decay=wd)
    history = {'train_loss': [], 'val_loss': [], 'val_acc': []}
    print(f"Starting training: {model_name} - {epochs} epochs, LR={lr},␣
↪WD={wd}, Optim={optimizer_class.__name__}")
    train_start_time = time.time()
    for epoch in range(epochs):
        epoch_start_time = time.time()
        model.train()
        running_loss = 0.0
        for batch in train_loader:
            inputs, targets = batch
            inputs, targets = inputs.to(device), targets.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            loss.backward()
            optimizer.step()
            running_loss += loss.item() * inputs.size(0)
        epoch_train_loss = running_loss / len(train_loader.dataset)
        history['train_loss'].append(epoch_train_loss)
        epoch_val_loss, epoch_val_acc = evaluate(model, val_loader, criterion,␣
↪device)
```

```
        history['val_loss'].append(epoch_val_loss)
        history['val_acc'].append(epoch_val_acc)
        epoch_end_time = time.time()
        if (epoch + 1) % 5 == 0 or epoch == 0 or (epoch + 1 == epochs) :
            print(f"  Epoch {epoch+1}/{epochs} -> Train Loss: {epoch_train_loss:
↪.4f}, Val Loss: {epoch_val_loss:.4f}, Val Acc: {epoch_val_acc:.4f}␣
↪({(epoch_end_time - epoch_start_time):.2f}s)")
    train_end_time = time.time()
    print(f"Finished training {model_name}. Total time: {(train_end_time -␣
↪train_start_time):.2f}s")
    return history
```

[44]:
```
print("\n\n" + "="*30 + " Phase 4: Learning Rate Optimization " + "="*30)

# --- LR Finder Setup ---
print("\n--- Setting up Learning Rate Finder ---")
model_lr = INITIAL_MODEL_CLASS().to(DEVICE)
optimizer_lr = optim.Adam(model_lr.parameters(), lr=1e-7) # Start low
criterion_lr = nn.CrossEntropyLoss()
lr_finder = LRFinder(model_lr, optimizer_lr, criterion_lr, device=DEVICE)
num_iterations_lr = len(train_loader)
print(f"LR Finder will run for approximately {num_iterations_lr} iterations (1␣
↪epoch).")
```

```
============================== Phase 4: Learning Rate Optimization
==============================

--- Setting up Learning Rate Finder ---
LR Finder will run for approximately 2909 iterations (1 epoch).
```

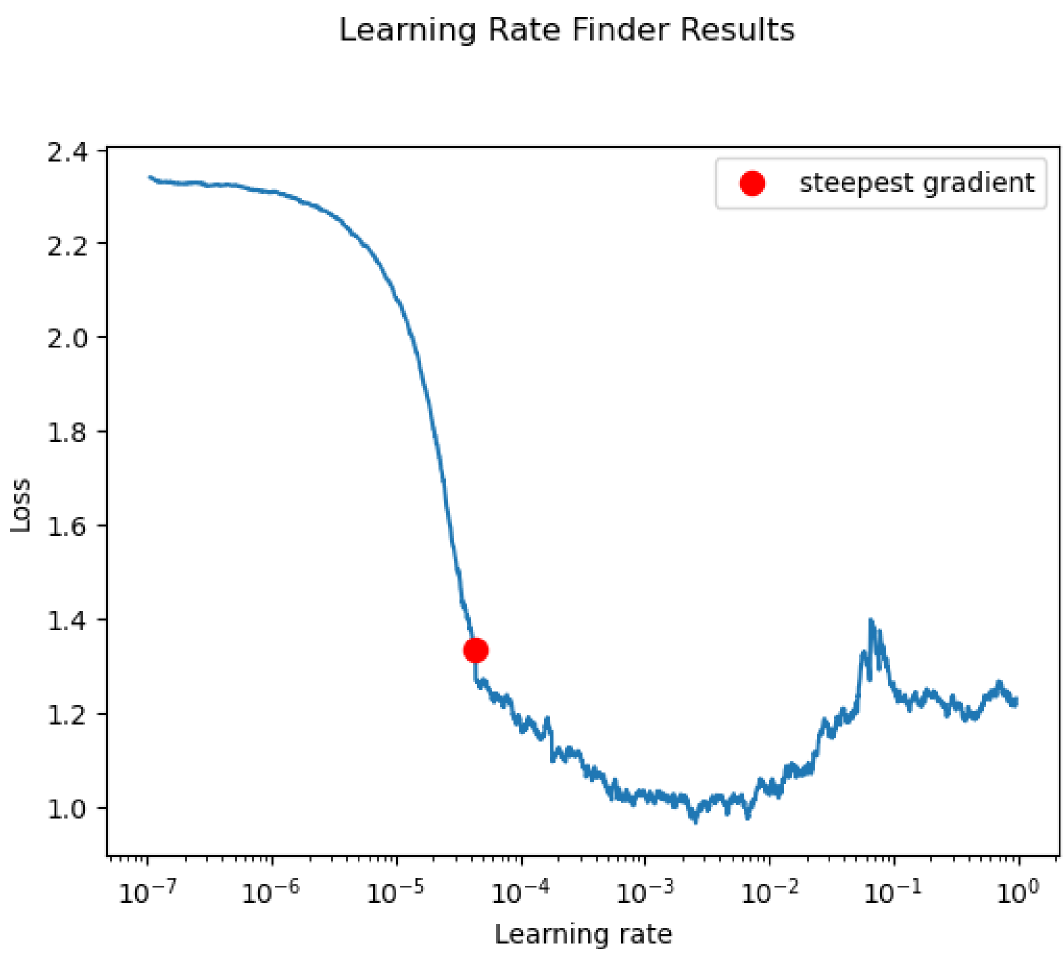
[45]:
```
# --- Run LR Finder ---
print("Running LR Finder...")
start_time = time.time()
# CORRECTED CALL: Removed skip_start, skip_end
lr_finder.range_test(train_loader, end_lr=1, num_iter=num_iterations_lr,␣
↪step_mode="exp")
end_time = time.time()
print(f"LR Finder finished in {end_time - start_time:.2f} seconds.")
```

```
Running LR Finder…

  0%|          | 0/2909 [00:00<?, ?it/s]

Learning rate search finished. See the graph with {finder_name}.plot()
LR Finder finished in 6.91 seconds.
```

[46]:

```python
# --- Plot LR Finder Results ---
print("\nPlotting LR vs. Loss...")
lr_finder_fig_path = os.path.join(PLOT_SAVE_DIR, 'lr_finder_plot.png')
# CORRECTED CALL: Removed unsupported args, plotting to current figure
fig, ax = plt.subplots() # Create figure manually
lr_finder.plot(ax=ax, log_lr=True) # Plot to the created axes
fig.suptitle("Learning Rate Finder Results", y=1.02)
fig.savefig(lr_finder_fig_path) # Save manually
print(f"LR Finder plot saved to {lr_finder_fig_path}")
plt.show() # Show plot
```

```
Plotting LR vs. Loss…
LR suggestion: steepest gradient
Suggested LR: 4.35E-05
LR Finder plot saved to ../results/plots/lr_finder_plot.png
```



[47]:
```python
# --- Analyze the Plot and Select LR ---
print("\n--- Analysis ---")
print("Examine the generated plot ('lr_finder_plot.png').")
# **MANUALLY ADJUST THIS BASED ON YOUR PLOT**
suggested_lr_from_finder = 1e-3
print(f"Suggested LR based on visual inspection (ADJUST IF NEEDED):␣
↪{suggested_lr_from_finder}")

lr_finder.reset()
print("LR Finder state and model weights have been reset.")
```

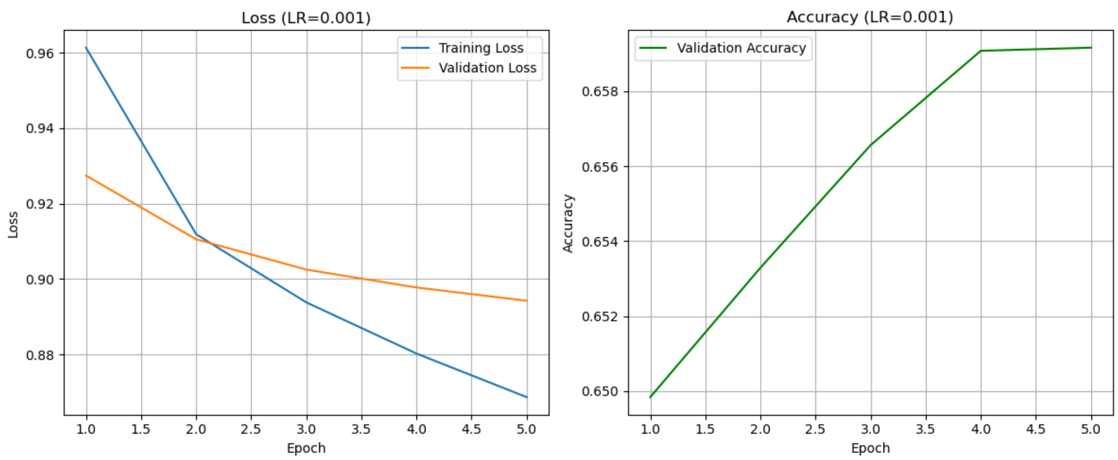
```
--- Analysis ---
Examine the generated plot ('lr_finder_plot.png').
Suggested LR based on visual inspection (ADJUST IF NEEDED): 0.001
LR Finder state and model weights have been reset.
```

[48]:
```python
# --- Verification ---
print("\n--- Verifying Suggested LR ---")
model_verify = INITIAL_MODEL_CLASS().to(DEVICE)
# Note: Using Adam here consistent with LR finder optimizer
optimizer_verify = optim.Adam(model_verify.parameters(),␣
↪lr=suggested_lr_from_finder)
criterion_verify = nn.CrossEntropyLoss()

verification_history = run_component_training(
    model=model_verify, optimizer_class=optim.Adam, criterion=criterion_verify,
    train_loader=train_loader, val_loader=val_loader,␣
↪lr=suggested_lr_from_finder, wd=0,
    epochs=N_EPOCHS_VERIFY, device=DEVICE,␣
↪model_name=f"LR_Verify_{suggested_lr_from_finder}"
)
```

```
--- Verifying Suggested LR ---
Starting training: LR_Verify_0.001 - 5 epochs, LR=0.001, WD=0, Optim=Adam
  Epoch 1/5 -> Train Loss: 0.9613, Val Loss: 0.9274, Val Acc: 0.6498 (7.94s)
  Epoch 5/5 -> Train Loss: 0.8687, Val Loss: 0.8942, Val Acc: 0.6592 (7.90s)
Finished training LR_Verify_0.001. Total time: 39.16s
```

[49]:
```python
# Plot verification results
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1); plt.plot(range(1, N_EPOCHS_VERIFY + 1),␣
↪verification_history['train_loss'], label='Training Loss'); plt.
↪plot(range(1, N_EPOCHS_VERIFY + 1), verification_history['val_loss'],␣
↪label='Validation Loss'); plt.xlabel('Epoch'); plt.ylabel('Loss'); plt.
↪title(f'Loss (LR={suggested_lr_from_finder})'); plt.legend(); plt.grid(True)
```

```
plt.subplot(1, 2, 2); plt.plot(range(1, N_EPOCHS_VERIFY + 1),␣
↪verification_history['val_acc'], label='Validation Accuracy', color='green');
↪ plt.xlabel('Epoch'); plt.ylabel('Accuracy'); plt.title(f'Accuracy␣
↪(LR={suggested_lr_from_finder})'); plt.legend(); plt.grid(True)
plt.tight_layout(); plt.savefig(os.path.join(PLOT_SAVE_DIR,␣
↪f'lr_{suggested_lr_from_finder}_verification.png')); plt.show()

print("\nVerification Complete. Ensure learning is stable.")
OPTIMAL_LR = suggested_lr_from_finder
print(f"Set OPTIMAL_LR = {OPTIMAL_LR}")
```




```
Verification Complete. Ensure learning is stable.
Set OPTIMAL_LR = 0.001
```

[50]:
```
print("\n\n" + "="*50); print("--- Phase 5: Advanced Optimization Techniques␣
↪---"); print("="*50 + "\n")

# --- 5.1 Weight Decay Optimization (using K-Fold CV) ---
print("\n--- 5.1 Weight Decay (L2 Regularization) Optimization ---")
kf = KFold(n_splits=K_FOLDS, shuffle=True, random_state=SEED)
kfold_results = {wd: [] for wd in WEIGHT_DECAY_VALUES}
kfold_val_losses = {wd: [] for wd in WEIGHT_DECAY_VALUES}
print(f"Starting K-Fold CV (k={K_FOLDS}) for WD values: {WEIGHT_DECAY_VALUES}");
↪ print(f"Training each fold for {N_EPOCHS_KFOLD} epochs with␣
↪LR={OPTIMAL_LR}")
fold_start_time = time.time()
```

```
==================================================
--- Phase 5: Advanced Optimization Techniques ---
```

```
==================================================
```

```
--- 5.1 Weight Decay (L2 Regularization) Optimization ---
Starting K-Fold CV (k=5) for WD values: [0, 1e-05, 0.0001, 0.001, 0.01]
Training each fold for 10 epochs with LR=0.001
```

[51]:
```python
for wd_value in WEIGHT_DECAY_VALUES:
    print(f"\n-- Testing Weight Decay = {wd_value} --")
    fold_accuracies = []
    fold_losses = []
    for fold, (train_idx, val_idx) in enumerate(kf.split(X_train)):
        fold_loop_start_time = time.time()
        X_train_fold, y_train_fold = X_train[train_idx], y_train[train_idx]
        X_val_fold, y_val_fold = X_train[val_idx], y_train[val_idx]
        train_fold_dataset = TensorDataset(X_train_fold, y_train_fold);␣
↪val_fold_dataset = TensorDataset(X_val_fold, y_val_fold)
        train_fold_loader = DataLoader(train_fold_dataset,␣
↪batch_size=BATCH_SIZE, shuffle=True); val_fold_loader =␣
↪DataLoader(val_fold_dataset, batch_size=BATCH_SIZE * 2)
        model_fold = INITIAL_MODEL_CLASS().to(DEVICE)
        optimizer_fold = optim.Adam(model_fold.parameters(), lr=OPTIMAL_LR,␣
↪weight_decay=wd_value); criterion_fold = nn.CrossEntropyLoss()
        for epoch in range(N_EPOCHS_KFOLD):
            model_fold.train()
            for batch in train_fold_loader:
                inputs, targets = batch; inputs, targets = inputs.to(DEVICE),␣
↪targets.to(DEVICE)
                optimizer_fold.zero_grad(); outputs = model_fold(inputs); loss␣
↪= criterion_fold(outputs, targets)
                loss.backward(); optimizer_fold.step()
        final_val_loss, final_val_acc = evaluate(model_fold, val_fold_loader,␣
↪criterion_fold, DEVICE)
        fold_accuracies.append(final_val_acc); fold_losses.
↪append(final_val_loss)
        fold_loop_end_time = time.time()
        print(f"    Fold {fold+1} finished. Val Loss: {final_val_loss:.4f}, Val␣
↪Acc: {final_val_acc:.4f} (Time: {fold_loop_end_time - fold_loop_start_time:.
↪2f}s)")
    kfold_results[wd_value] = fold_accuracies; kfold_val_losses[wd_value] =␣
↪fold_losses
    print(f"  Finished testing WD = {wd_value}. Avg Acc: {np.
↪mean(fold_accuracies):.4f} +/- {np.std(fold_accuracies):.4f}")

fold_end_time = time.time(); print(f"\nK-Fold CV finished in {(fold_end_time -␣
↪fold_start_time)/60:.2f} minutes.")
```

```
-- Testing Weight Decay = 0 --
    Fold 1 finished. Val Loss: 0.9239, Val Acc: 0.6545 (Time: 56.24s)
    Fold 2 finished. Val Loss: 0.9138, Val Acc: 0.6575 (Time: 56.72s)
    Fold 3 finished. Val Loss: 0.9185, Val Acc: 0.6556 (Time: 55.94s)
    Fold 4 finished. Val Loss: 0.9171, Val Acc: 0.6558 (Time: 55.01s)
    Fold 5 finished. Val Loss: 0.9190, Val Acc: 0.6539 (Time: 56.65s)
  Finished testing WD = 0. Avg Acc: 0.6555 +/- 0.0012

-- Testing Weight Decay = 1e-05 --
    Fold 1 finished. Val Loss: 0.9234, Val Acc: 0.6547 (Time: 60.38s)
    Fold 2 finished. Val Loss: 0.9072, Val Acc: 0.6571 (Time: 58.92s)
    Fold 3 finished. Val Loss: 0.9191, Val Acc: 0.6561 (Time: 58.71s)
    Fold 4 finished. Val Loss: 0.9095, Val Acc: 0.6580 (Time: 58.98s)
    Fold 5 finished. Val Loss: 0.9112, Val Acc: 0.6547 (Time: 58.57s)
  Finished testing WD = 1e-05. Avg Acc: 0.6561 +/- 0.0013

-- Testing Weight Decay = 0.0001 --
    Fold 1 finished. Val Loss: 0.9077, Val Acc: 0.6556 (Time: 58.83s)
    Fold 2 finished. Val Loss: 0.9001, Val Acc: 0.6587 (Time: 58.32s)
    Fold 3 finished. Val Loss: 0.9050, Val Acc: 0.6587 (Time: 58.58s)
    Fold 4 finished. Val Loss: 0.8972, Val Acc: 0.6586 (Time: 58.32s)
    Fold 5 finished. Val Loss: 0.9053, Val Acc: 0.6558 (Time: 59.32s)
  Finished testing WD = 0.0001. Avg Acc: 0.6575 +/- 0.0014

-- Testing Weight Decay = 0.001 --
    Fold 1 finished. Val Loss: 0.9263, Val Acc: 0.6517 (Time: 60.70s)
    Fold 2 finished. Val Loss: 0.9352, Val Acc: 0.6498 (Time: 59.59s)
    Fold 3 finished. Val Loss: 0.9229, Val Acc: 0.6517 (Time: 58.49s)
    Fold 4 finished. Val Loss: 0.9178, Val Acc: 0.6520 (Time: 58.38s)
    Fold 5 finished. Val Loss: 0.9231, Val Acc: 0.6497 (Time: 59.09s)
  Finished testing WD = 0.001. Avg Acc: 0.6510 +/- 0.0010

-- Testing Weight Decay = 0.01 --
    Fold 1 finished. Val Loss: 0.9992, Val Acc: 0.6311 (Time: 58.35s)
    Fold 2 finished. Val Loss: 0.9943, Val Acc: 0.6326 (Time: 58.29s)
    Fold 3 finished. Val Loss: 0.9998, Val Acc: 0.6251 (Time: 58.34s)
    Fold 4 finished. Val Loss: 0.9927, Val Acc: 0.6311 (Time: 58.25s)
    Fold 5 finished. Val Loss: 0.9913, Val Acc: 0.6284 (Time: 58.43s)
  Finished testing WD = 0.01. Avg Acc: 0.6297 +/- 0.0026

K-Fold CV finished in 24.29 minutes.
```

[52]:

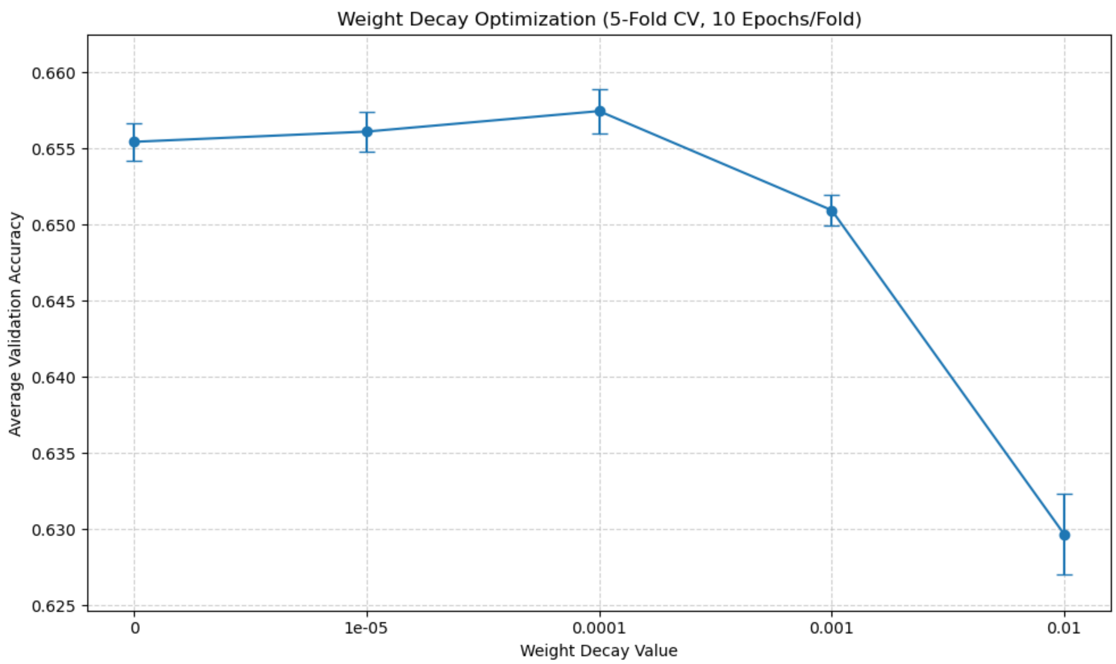
```python
# --- Analyze K-Fold Results ---
print("\n--- Weight Decay K-Fold CV Results Summary ---")
avg_accuracies = {wd: np.mean(accs) for wd, accs in kfold_results.items()};␣
↪std_accuracies = {wd: np.std(accs) for wd, accs in kfold_results.items()}
```

```
avg_losses = {wd: np.mean(losses) for wd, losses in kfold_val_losses.items()}
print(f"{'Weight Decay':<15} | {'Avg Val Acc':<15} | {'Std Val Acc':<15} |␣
↪{'Avg Val Loss':<15}"); print("-" * 65)
for wd_value in WEIGHT_DECAY_VALUES: print(f"{wd_value:<15} |␣
↪{avg_accuracies[wd_value]:.4f}{' ':<10} | {std_accuracies[wd_value]:.4f}{' ':
↪<10} | {avg_losses[wd_value]:.4f}")
```

```
--- Weight Decay K-Fold CV Results Summary ---
Weight Decay    | Avg Val Acc     | Std Val Acc     | Avg Val Loss
-----------------------------------------------------------------
0               | 0.6555          | 0.0012          | 0.9185
1e-05           | 0.6561          | 0.0013          | 0.9141
0.0001          | 0.6575          | 0.0014          | 0.9030
0.001           | 0.6510          | 0.0010          | 0.9251
0.01            | 0.6297          | 0.0026          | 0.9954
```

[53]:
```
# --- Plot K-Fold Results ---
wd_values_str = [str(wd) for wd in WEIGHT_DECAY_VALUES]; avg_acc_list =␣
↪[avg_accuracies[wd] for wd in WEIGHT_DECAY_VALUES]; std_acc_list =␣
↪[std_accuracies[wd] for wd in WEIGHT_DECAY_VALUES]
plt.figure(figsize=(10, 6)); plt.errorbar(wd_values_str, avg_acc_list,␣
↪yerr=std_acc_list, marker='o', capsize=5); plt.xlabel('Weight Decay Value');␣
↪plt.ylabel('Average Validation Accuracy')
plt.title(f'Weight Decay Optimization ({K_FOLDS}-Fold CV, {N_EPOCHS_KFOLD}␣
↪Epochs/Fold)'); plt.grid(True, linestyle='--', alpha=0.6); plt.
↪ylim(min(avg_acc_list) - 0.005, max(avg_acc_list) + 0.005)
plt.tight_layout(); plt.savefig(os.path.join(PLOT_SAVE_DIR,␣
↪'weight_decay_kfold_cv.png')); plt.show()
```

```
2025-04-25 15:13:01,991 - INFO - Using categorical units to plot a list of
strings that are all parsable as floats or dates. If these strings should be
plotted as numbers, cast to the appropriate data type before plotting.
2025-04-25 15:13:01,992 - INFO - Using categorical units to plot a list of
strings that are all parsable as floats or dates. If these strings should be
plotted as numbers, cast to the appropriate data type before plotting.
2025-04-25 15:13:01,993 - INFO - Using categorical units to plot a list of
strings that are all parsable as floats or dates. If these strings should be
plotted as numbers, cast to the appropriate data type before plotting.
2025-04-25 15:13:01,994 - INFO - Using categorical units to plot a list of
strings that are all parsable as floats or dates. If these strings should be
plotted as numbers, cast to the appropriate data type before plotting.
2025-04-25 15:13:01,994 - INFO - Using categorical units to plot a list of
strings that are all parsable as floats or dates. If these strings should be
plotted as numbers, cast to the appropriate data type before plotting.
```




[54]:
```
# --- Select Best Weight Decay ---
OPTIMAL_WEIGHT_DECAY = max(avg_accuracies, key=avg_accuracies.get)
print(f"\nBest Weight Decay (based on K-Fold): {OPTIMAL_WEIGHT_DECAY} (Avg Acc:␣
↪{avg_accuracies[OPTIMAL_WEIGHT_DECAY]:.4f})")
```

```
Best Weight Decay (based on K-Fold): 0.0001 (Avg Acc: 0.6575)
```

[55]:
```
# --- 5.2 Neural Network Component Optimization ---
print("\n\n" + "="*50); print("--- 5.2 Neural Network Component Optimization␣
↪---"); print("="*50 + "\n")

# --- Baseline Configuration ---
BASELINE_LR = OPTIMAL_LR; BASELINE_WD = OPTIMAL_WEIGHT_DECAY;␣
↪BASELINE_MODEL_CLASS = INITIAL_MODEL_CLASS; BASELINE_OPTIMIZER = optim.Adam
print(f"Baseline Config: LR={BASELINE_LR}, WD={BASELINE_WD},␣
↪Model={BASELINE_MODEL_CLASS.__name__}, Optimizer={BASELINE_OPTIMIZER.
↪__name__}"); print(f"Training each component test for␣
↪{N_EPOCHS_COMPONENT_TEST} epochs.")
component_results = {}
```

```
==================================================
--- 5.2 Neural Network Component Optimization ---
==================================================
```

```
Baseline Config: LR=0.001, WD=0.0001, Model=Model_2, Optimizer=Adam
Training each component test for 15 epochs.
```

[56]:

```python
# --- 5.2.1 Weight Initialization ---
print("\n--- 5.2.1 Testing Weight Initialization ---")
initialization_strategies = {"Default (Kaiming Uniform for ReLU)": None,␣
↪"Xavier Uniform": nn.init.xavier_uniform_, "Kaiming Normal": nn.init.
↪kaiming_normal_}
initialization_results = {}
for init_name, init_func in initialization_strategies.items():
    print(f"\n-- Testing Initialization: {init_name} --")
    model_init = BASELINE_MODEL_CLASS().to(DEVICE) # Using default activation␣
↪(ReLU)
    criterion_init = nn.CrossEntropyLoss()
    if init_func is not None:
        def initialize_weights(m):
            if isinstance(m, nn.Linear):
                try: gain = nn.init.calculate_gain('relu') if 'kaiming' in␣
↪init_name.lower() else 1.0; init_func(m.weight, gain=gain)
                except TypeError: init_func(m.weight)
                if m.bias is not None: nn.init.constant_(m.bias, 0)
        model_init.apply(initialize_weights); print("Applied custom weight␣
↪initialization.")
    else: print("Using default PyTorch weight initialization.")
    history = run_component_training(model=model_init,␣
↪optimizer_class=BASELINE_OPTIMIZER, criterion=criterion_init,␣
↪train_loader=train_loader, val_loader=val_loader, lr=BASELINE_LR,␣
↪wd=BASELINE_WD, epochs=N_EPOCHS_COMPONENT_TEST, device=DEVICE,␣
↪model_name=f"Init_{init_name}")
    initialization_results[init_name] = history;␣
↪component_results[f"Init_{init_name}"] = history
```

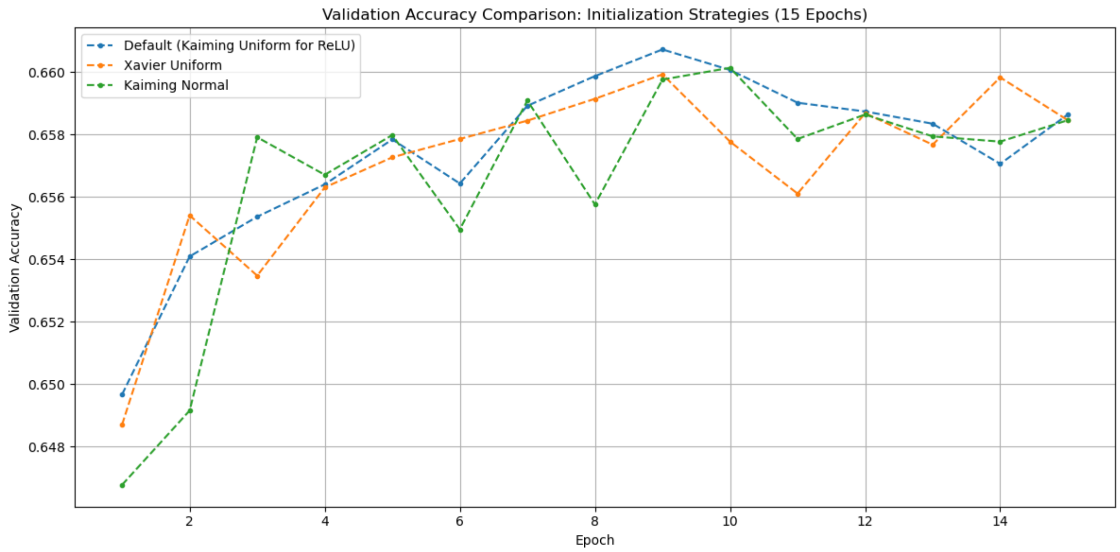
```
--- 5.2.1 Testing Weight Initialization ---

-- Testing Initialization: Default (Kaiming Uniform for ReLU) --
Using default PyTorch weight initialization.
Starting training: Init_Default (Kaiming Uniform for ReLU) - 15 epochs,
LR=0.001, WD=0.0001, Optim=Adam
  Epoch 1/15 -> Train Loss: 0.9627, Val Loss: 0.9264, Val Acc: 0.6497 (8.22s)
  Epoch 5/15 -> Train Loss: 0.8826, Val Loss: 0.8975, Val Acc: 0.6578 (8.19s)
  Epoch 10/15 -> Train Loss: 0.8556, Val Loss: 0.8964, Val Acc: 0.6601 (8.32s)
  Epoch 15/15 -> Train Loss: 0.8389, Val Loss: 0.9031, Val Acc: 0.6586 (8.06s)
Finished training Init_Default (Kaiming Uniform for ReLU). Total time: 124.20s

-- Testing Initialization: Xavier Uniform --
Applied custom weight initialization.
```

```
Starting training: Init_Xavier Uniform - 15 epochs, LR=0.001, WD=0.0001,
Optim=Adam
  Epoch 1/15 -> Train Loss: 0.9610, Val Loss: 0.9264, Val Acc: 0.6487 (8.23s)
  Epoch 5/15 -> Train Loss: 0.8771, Val Loss: 0.9008, Val Acc: 0.6573 (8.11s)
  Epoch 10/15 -> Train Loss: 0.8494, Val Loss: 0.8974, Val Acc: 0.6578 (8.13s)
  Epoch 15/15 -> Train Loss: 0.8304, Val Loss: 0.9021, Val Acc: 0.6585 (8.13s)
Finished training Init_Xavier Uniform. Total time: 122.11s

-- Testing Initialization: Kaiming Normal --
Applied custom weight initialization.
Starting training: Init_Kaiming Normal - 15 epochs, LR=0.001, WD=0.0001,
Optim=Adam
  Epoch 1/15 -> Train Loss: 0.9752, Val Loss: 0.9301, Val Acc: 0.6468 (8.11s)
  Epoch 5/15 -> Train Loss: 0.8778, Val Loss: 0.8980, Val Acc: 0.6580 (8.33s)
  Epoch 10/15 -> Train Loss: 0.8495, Val Loss: 0.8959, Val Acc: 0.6601 (8.07s)
  Epoch 15/15 -> Train Loss: 0.8306, Val Loss: 0.9018, Val Acc: 0.6585 (8.14s)
Finished training Init_Kaiming Normal. Total time: 122.13s
```

[57]:
```python
# --- Analyze Initialization Results ---
print("\n--- Weight Initialization Results Summary ---")
print(f"{'Initialization':<35} | {'Final Val Loss':<15} | {'Final Val Acc':<15}␣
↪| {'Max Val Acc':<15}"); print("-" * 85)
best_init_name = ""; best_init_max_acc = -1.0
for name, history in initialization_results.items():
    final_val_loss = history['val_loss'][-1]; final_val_acc =␣
↪history['val_acc'][-1]; max_val_acc = max(history['val_acc'])
    print(f"{name:<35} | {final_val_loss:<15.4f} | {final_val_acc:<15.4f} |␣
↪{max_val_acc:<15.4f}")
    if max_val_acc > best_init_max_acc: best_init_max_acc = max_val_acc;␣
↪best_init_name = name
print(f"\nBest Initialization Strategy (Max Val Acc): {best_init_name}␣
↪({best_init_max_acc:.4f})")
# Sticking with default unless a clear winner emerges
```

```
--- Weight Initialization Results Summary ---
Initialization                      | Final Val Loss  | Final Val Acc   | Max
Val Acc
--------------------------------------------------------------------------------
-----
Default (Kaiming Uniform for ReLU)  | 0.9031          | 0.6586          | 0.6607
Xavier Uniform                      | 0.9021          | 0.6585          | 0.6599
Kaiming Normal                      | 0.9018          | 0.6585          | 0.6601

Best Initialization Strategy (Max Val Acc): Default (Kaiming Uniform for ReLU)
(0.6607)
```

[58]:
```
# --- Plot Initialization comparison ---
plt.figure(figsize=(12, 6));
for name, history in initialization_results.items(): plt.plot(range(1,␣
↪N_EPOCHS_COMPONENT_TEST + 1), history['val_acc'], label=name, marker='.',␣
↪linestyle='--')
plt.title(f'Validation Accuracy Comparison: Initialization Strategies␣
↪({N_EPOCHS_COMPONENT_TEST} Epochs)'); plt.xlabel('Epoch'); plt.
↪ylabel('Validation Accuracy'); plt.legend(); plt.grid(True); plt.
↪tight_layout()
plt.savefig(os.path.join(PLOT_SAVE_DIR, 'component_initialization_comparison.
↪png')); plt.show()
```




[59]:
```
print("\n" + "="*30 + " Reloading Models Module " + "="*30)
if 'models' in sys.modules:
    print("Attempting to reload 'models' module..."); import models; importlib.
↪reload(models); from models import Model_1, Model_2, Model_3,␣
↪ACTIVATION_FUNCTIONS
    print("'models' module reloaded."); print("Model_2 __init__ signature:",␣
↪inspect.signature(Model_2.__init__))
    BASELINE_MODEL_CLASS = Model_2 # Update reference
else:
    print("Importing 'models' module..."); import models; from models import␣
↪Model_1, Model_2, Model_3, ACTIVATION_FUNCTIONS
    BASELINE_MODEL_CLASS = Model_2
    print("'models' module imported."); print("Model_2 __init__ signature:",␣
↪inspect.signature(Model_2.__init__))
```

```
============================== Reloading Models Module
```

```
==============================
Attempting to reload 'models' module…
'models' module reloaded.
Model_2 __init__ signature: (self, input_size=90, num_classes=10,
activation_fn=<class 'torch.nn.modules.activation.ReLU'>,
activation_name='ReLU')
```

[60]:

```
print("\n--- 5.2.2 Testing Activation Functions ---")
activation_functions_to_test = {"ReLU": nn.ReLU, "LeakyReLU": nn.LeakyReLU,␣
↪"GELU": nn.GELU}
activation_results = {}
Activation_Model_Class = BASELINE_MODEL_CLASS # Use reloaded Model_2

for act_name, act_fn_class in activation_functions_to_test.items():
    print(f"\n-- Testing Activation: {act_name} --")
    model_act = Activation_Model_Class(activation_fn=act_fn_class,␣
↪activation_name=act_name).to(DEVICE) # Pass class and name
    criterion_act = nn.CrossEntropyLoss()
    print("Using default PyTorch weight initialization."); print(model_act)
    history = run_component_training(model=model_act,␣
↪optimizer_class=BASELINE_OPTIMIZER, criterion=criterion_act,␣
↪train_loader=train_loader, val_loader=val_loader, lr=BASELINE_LR,␣
↪wd=BASELINE_WD, epochs=N_EPOCHS_COMPONENT_TEST, device=DEVICE,␣
↪model_name=f"Activation_{act_name}")
    activation_results[act_name] = history;␣
↪component_results[f"Activation_{act_name}"] = history
```

```
--- 5.2.2 Testing Activation Functions ---

-- Testing Activation: ReLU --
Using default PyTorch weight initialization.
Model Architecture: Model_2
  Input Size: 90
  Num Classes: 10
  Activation: ReLU
  (layer_1): Linear(in_features=90, out_features=256, bias=True)
  (layer_2): Linear(in_features=256, out_features=256, bias=True)
  (output_layer): Linear(in_features=256, out_features=10, bias=True)
Total Trainable Parameters: 91,658

Starting training: Activation_ReLU - 15 epochs, LR=0.001, WD=0.0001, Optim=Adam
  Epoch 1/15 -> Train Loss: 0.9619, Val Loss: 0.9269, Val Acc: 0.6492 (8.16s)
  Epoch 5/15 -> Train Loss: 0.8818, Val Loss: 0.8997, Val Acc: 0.6567 (8.20s)
  Epoch 10/15 -> Train Loss: 0.8559, Val Loss: 0.8985, Val Acc: 0.6572 (8.10s)
  Epoch 15/15 -> Train Loss: 0.8398, Val Loss: 0.9000, Val Acc: 0.6570 (8.16s)
Finished training Activation_ReLU. Total time: 122.86s
```

```
-- Testing Activation: LeakyReLU --
Using default PyTorch weight initialization.
Model Architecture: Model_2
  Input Size: 90
  Num Classes: 10
  Activation: LeakyReLU
  (layer_1): Linear(in_features=90, out_features=256, bias=True)
  (layer_2): Linear(in_features=256, out_features=256, bias=True)
  (output_layer): Linear(in_features=256, out_features=10, bias=True)
Total Trainable Parameters: 91,658

Starting training: Activation_LeakyReLU - 15 epochs, LR=0.001, WD=0.0001,
Optim=Adam
  Epoch 1/15 -> Train Loss: 0.9630, Val Loss: 0.9364, Val Acc: 0.6471 (8.23s)
  Epoch 5/15 -> Train Loss: 0.8824, Val Loss: 0.8991, Val Acc: 0.6572 (8.09s)
  Epoch 10/15 -> Train Loss: 0.8561, Val Loss: 0.8940, Val Acc: 0.6602 (8.27s)
  Epoch 15/15 -> Train Loss: 0.8389, Val Loss: 0.8986, Val Acc: 0.6588 (8.30s)
Finished training Activation_LeakyReLU. Total time: 124.69s

-- Testing Activation: GELU --
Using default PyTorch weight initialization.
Model Architecture: Model_2
  Input Size: 90
  Num Classes: 10
  Activation: GELU
  (layer_1): Linear(in_features=90, out_features=256, bias=True)
  (layer_2): Linear(in_features=256, out_features=256, bias=True)
  (output_layer): Linear(in_features=256, out_features=10, bias=True)
Total Trainable Parameters: 91,658

Starting training: Activation_GELU - 15 epochs, LR=0.001, WD=0.0001, Optim=Adam
  Epoch 1/15 -> Train Loss: 0.9578, Val Loss: 0.9209, Val Acc: 0.6492 (8.41s)
  Epoch 5/15 -> Train Loss: 0.8744, Val Loss: 0.8942, Val Acc: 0.6582 (8.26s)
  Epoch 10/15 -> Train Loss: 0.8396, Val Loss: 0.8944, Val Acc: 0.6606 (8.14s)
  Epoch 15/15 -> Train Loss: 0.8134, Val Loss: 0.9012, Val Acc: 0.6576 (8.08s)
Finished training Activation_GELU. Total time: 123.08s
```

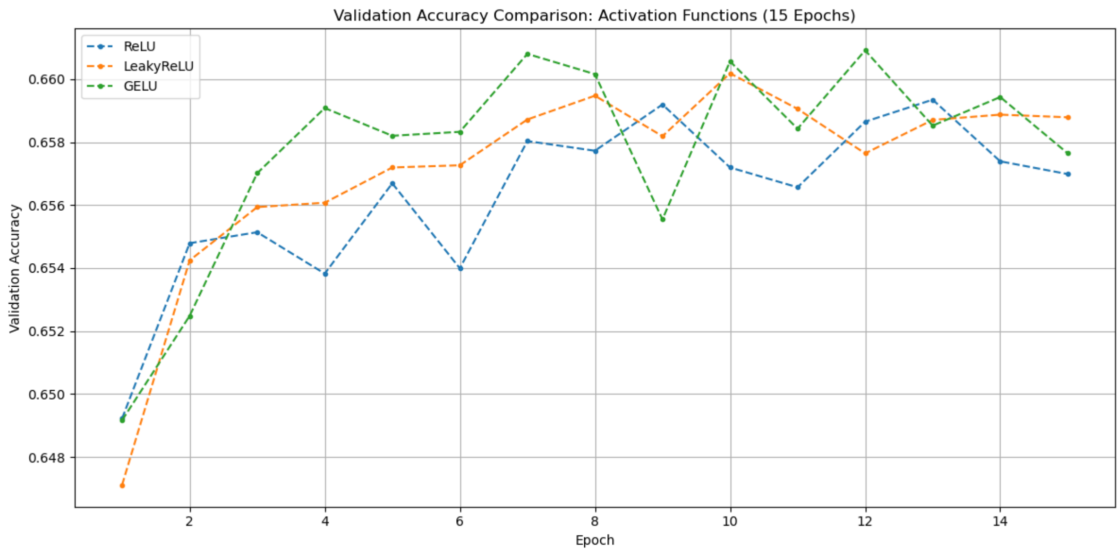
[61]:

```python
print("\n--- Activation Function Results Summary ---")
print(f"{'Activation':<15} | {'Final Val Loss':<15} | {'Final Val Acc':<15} |␣
↪{'Max Val Acc':<15}"); print("-" * 65)
best_act_name = ""; best_act_max_acc = -1.0
for name, history in activation_results.items():
    final_val_loss = history['val_loss'][-1]; final_val_acc =␣
↪history['val_acc'][-1]; max_val_acc = max(history['val_acc'])
    print(f"{name:<15} | {final_val_loss:<15.4f} | {final_val_acc:<15.4f} |␣
↪{max_val_acc:<15.4f}")
```

```
    if max_val_acc > best_act_max_acc: best_act_max_acc = max_val_acc;␣
↪best_act_name = name
print(f"\nBest Activation Function (Max Val Acc): {best_act_name}␣
↪({best_act_max_acc:.4f})")
if best_act_name in activation_functions_to_test: OPTIMAL_ACTIVATION_FN =␣
↪activation_functions_to_test[best_act_name]; print(f"Stored␣
↪OPTIMAL_ACTIVATION_FN: {OPTIMAL_ACTIVATION_FN}")
else: print("Error: Best activation name not found!"); OPTIMAL_ACTIVATION_FN =␣
↪nn.ReLU
```

```
--- Activation Function Results Summary ---
Activation      | Final Val Loss  | Final Val Acc   | Max Val Acc
-----------------------------------------------------------------
ReLU            | 0.9000          | 0.6570          | 0.6593
LeakyReLU       | 0.8986          | 0.6588          | 0.6602
GELU            | 0.9012          | 0.6576          | 0.6609

Best Activation Function (Max Val Acc): GELU (0.6609)
Stored OPTIMAL_ACTIVATION_FN: <class 'torch.nn.modules.activation.GELU'>
```

[62]:
```
# --- Plot Activation comparison ---
plt.figure(figsize=(12, 6));
for name, history in activation_results.items(): plt.plot(range(1,␣
↪N_EPOCHS_COMPONENT_TEST + 1), history['val_acc'], label=name, marker='.',␣
↪linestyle='--')
plt.title(f'Validation Accuracy Comparison: Activation Functions␣
↪({N_EPOCHS_COMPONENT_TEST} Epochs)'); plt.xlabel('Epoch'); plt.
↪ylabel('Validation Accuracy'); plt.legend(); plt.grid(True); plt.
↪tight_layout()
plt.savefig(os.path.join(PLOT_SAVE_DIR, 'component_activation_comparison.png'));
↪ plt.show()
```




[63]:
```
print("\nNext steps: Test Normalization Layers and Optimizers.")
```

```
Next steps: Test Normalization Layers and Optimizers.
```

[64]:
```
# ==========================================================================
# == Module Reloading (Ensure latest models.py for normalization tests) ==
# ==========================================================================
print("\n" + "="*30 + " Reloading Models Module (Before Norm Tests) " + "="*30)
if 'models' in sys.modules:
    print("Attempting to reload 'models' module..."); import models; importlib.
↪reload(models); from models import Model_1, Model_2, Model_3,␣
↪ACTIVATION_FUNCTIONS
    print("'models' module reloaded."); print("Model_2 __init__ signature:",␣
↪inspect.signature(Model_2.__init__))
    BASELINE_MODEL_CLASS = Model_2 # Update reference to potentially new Model_2
else:
    print("Importing 'models' module..."); import models; from models import␣
↪Model_1, Model_2, Model_3, ACTIVATION_FUNCTIONS
    BASELINE_MODEL_CLASS = Model_2
    print("'models' module imported."); print("Model_2 __init__ signature:",␣
↪inspect.signature(Model_2.__init__))
```

```
============================== Reloading Models Module (Before Norm Tests)
==============================
Attempting to reload 'models' module…
'models' module reloaded.
Model_2 __init__ signature: (self, input_size=90, num_classes=10,
```

```
activation_fn=<class 'torch.nn.modules.activation.ReLU'>,
activation_name='ReLU', norm_layer_type=None)
```

[65]:
```python
# ==============================================================================
# == Normalization Layer Test ==
# ==============================================================================
print("\n--- 5.2.3 Testing Normalization Layers ---")

# Define normalization strategies to test
# Use string identifiers that match get_norm_layer function
normalization_strategies_to_test = {
    "None": None,
    "BatchNorm": "batch",
    "LayerNorm": "layer",
}

normalization_results = {}
Normalization_Model_Class = BASELINE_MODEL_CLASS # Should be reloaded Model_2
Best_Activation_Class = OPTIMAL_ACTIVATION_FN   # Use GELU determined previously
Best_Activation_Name = best_act_name            # 'GELU'

for norm_name, norm_type_str in normalization_strategies_to_test.items():
    print(f"\n-- Testing Normalization: {norm_name} --")

    # Instantiate model with the specific normalization type and best activation
    model_norm = Normalization_Model_Class(
        activation_fn=Best_Activation_Class,
        activation_name=Best_Activation_Name,
        norm_layer_type=norm_type_str # Pass the string identifier or None
    ).to(DEVICE)

    criterion_norm = nn.CrossEntropyLoss()
    print(f"Using default initialization, Activation={Best_Activation_Name},␣
↪Norm={norm_name}")
    print(model_norm) # Print model info

    # Train the model
    history = run_component_training(
        model=model_norm,
        optimizer_class=BASELINE_OPTIMIZER, # Adam
        criterion=criterion_norm,
        train_loader=train_loader,
        val_loader=val_loader,
        lr=BASELINE_LR,
        wd=BASELINE_WD,
        epochs=N_EPOCHS_COMPONENT_TEST, # Use same number of epochs
        device=DEVICE,
```

```python
        model_name=f"Norm_{norm_name}"
    )
    normalization_results[norm_name] = history
    component_results[f"Norm_{norm_name}"] = history # Store in main results
```

```
--- 5.2.3 Testing Normalization Layers ---

-- Testing Normalization: None --
Using default initialization, Activation=GELU, Norm=None
Model Architecture: Model_2
  Input Size: 90
  Num Classes: 10
  Activation: GELU
  Normalization: None
Total Trainable Parameters: 91,658

Starting training: Norm_None - 15 epochs, LR=0.001, WD=0.0001, Optim=Adam
  Epoch 1/15 -> Train Loss: 0.9568, Val Loss: 0.9203, Val Acc: 0.6508 (8.22s)
  Epoch 5/15 -> Train Loss: 0.8741, Val Loss: 0.8935, Val Acc: 0.6594 (8.09s)
  Epoch 10/15 -> Train Loss: 0.8407, Val Loss: 0.8923, Val Acc: 0.6612 (8.23s)
  Epoch 15/15 -> Train Loss: 0.8146, Val Loss: 0.9033, Val Acc: 0.6576 (8.07s)
Finished training Norm_None. Total time: 122.50s

-- Testing Normalization: BatchNorm --
Using default initialization, Activation=GELU, Norm=BatchNorm
Model Architecture: Model_2
  Input Size: 90
  Num Classes: 10
  Activation: GELU
  Normalization: batch
Total Trainable Parameters: 92,682

Starting training: Norm_BatchNorm - 15 epochs, LR=0.001, WD=0.0001, Optim=Adam
  Epoch 1/15 -> Train Loss: 0.9562, Val Loss: 0.9271, Val Acc: 0.6469 (11.82s)
  Epoch 5/15 -> Train Loss: 0.8937, Val Loss: 0.9047, Val Acc: 0.6546 (11.01s)
  Epoch 10/15 -> Train Loss: 0.8732, Val Loss: 0.8982, Val Acc: 0.6576 (11.09s)
  Epoch 15/15 -> Train Loss: 0.8616, Val Loss: 0.8984, Val Acc: 0.6578 (11.07s)
Finished training Norm_BatchNorm. Total time: 167.17s

-- Testing Normalization: LayerNorm --
Using default initialization, Activation=GELU, Norm=LayerNorm
Model Architecture: Model_2
  Input Size: 90
  Num Classes: 10
  Activation: GELU
  Normalization: layer
Total Trainable Parameters: 92,682
```

```
Starting training: Norm_LayerNorm - 15 epochs, LR=0.001, WD=0.0001, Optim=Adam
  Epoch 1/15 -> Train Loss: 0.9444, Val Loss: 0.9215, Val Acc: 0.6502 (11.79s)
  Epoch 5/15 -> Train Loss: 0.8778, Val Loss: 0.8981, Val Acc: 0.6586 (11.29s)
  Epoch 10/15 -> Train Loss: 0.8547, Val Loss: 0.8948, Val Acc: 0.6609 (11.22s)
  Epoch 15/15 -> Train Loss: 0.8398, Val Loss: 0.8926, Val Acc: 0.6603 (11.42s)
Finished training Norm_LayerNorm. Total time: 169.91s
```

[66]:

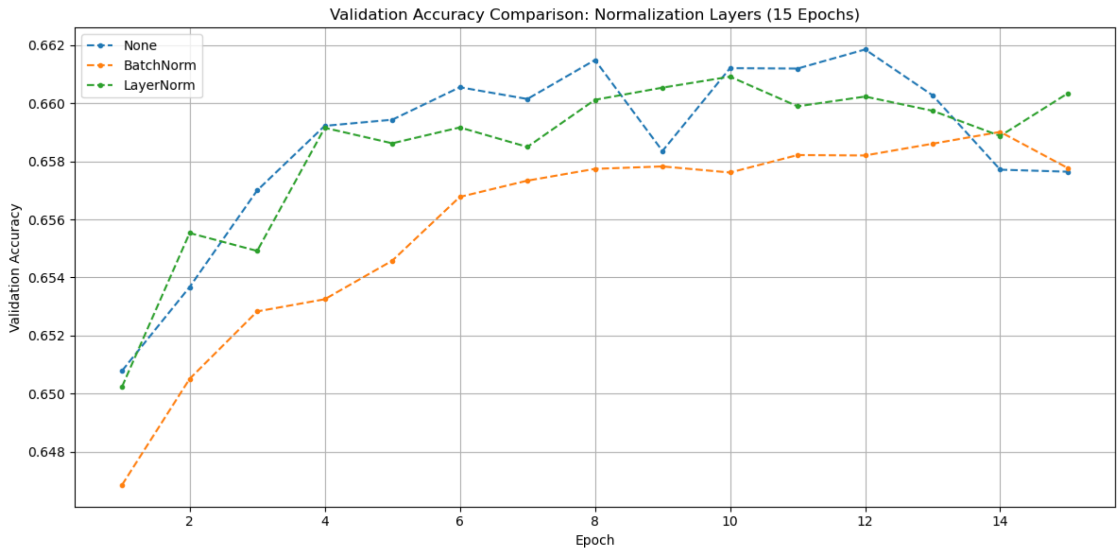
```
# --- Analyze Normalization Layer Results ---
print("\n--- Normalization Layer Results Summary ---")
print(f"{'Normalization':<15} | {'Final Val Loss':<15} | {'Final Val Acc':<15}␣
↪| {'Max Val Acc':<15}")
print("-" * 65)
best_norm_name = ""
best_norm_max_acc = -1.0

for name, history in normalization_results.items():
    final_val_loss = history['val_loss'][-1]
    final_val_acc = history['val_acc'][-1]
    max_val_acc = max(history['val_acc'])
    print(f"{name:<15} | {final_val_loss:<15.4f} | {final_val_acc:<15.4f} |␣
↪{max_val_acc:<15.4f}")
    if max_val_acc > best_norm_max_acc:
        best_norm_max_acc = max_val_acc
        best_norm_name = name

print(f"\nBest Normalization Strategy (based on Max Validation Accuracy):␣
↪{best_norm_name} ({best_norm_max_acc:.4f})")
OPTIMAL_NORM_TYPE = normalization_strategies_to_test[best_norm_name] # Store␣
↪the type ('batch', 'layer', or None)
print(f"Stored OPTIMAL_NORM_TYPE: {OPTIMAL_NORM_TYPE}")
```

```
--- Normalization Layer Results Summary ---
Normalization   | Final Val Loss  | Final Val Acc   | Max Val Acc
-----------------------------------------------------------------
None            | 0.9033          | 0.6576          | 0.6619
BatchNorm       | 0.8984          | 0.6578          | 0.6590
LayerNorm       | 0.8926          | 0.6603          | 0.6609

Best Normalization Strategy (based on Max Validation Accuracy): None (0.6619)
Stored OPTIMAL_NORM_TYPE: None
```

[67]:

```
# --- Plot Normalization comparison ---
plt.figure(figsize=(12, 6))
for name, history in normalization_results.items():
    plt.plot(range(1, N_EPOCHS_COMPONENT_TEST + 1), history['val_acc'],␣
↪label=name, marker='.', linestyle='--')
```

```
plt.title(f'Validation Accuracy Comparison: Normalization Layers␣
↪({N_EPOCHS_COMPONENT_TEST} Epochs)')
plt.xlabel('Epoch')
plt.ylabel('Validation Accuracy')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig(os.path.join(PLOT_SAVE_DIR, 'component_normalization_comparison.
↪png'))
plt.show()
```




[72]:
```
# ============================================================================
# == Module Reloading (Optional - Check if needed before optimizer tests) ==
# ============================================================================
# print("\n" + "="*30 + " Reloading Models Module (Before Optim Tests) " +␣
↪"="*30)
# if 'models' in sys.modules:
#     print("Attempting to reload 'models' module..."); import models;␣
↪importlib.reload(models); from models import Model_2 # Ensure Model_2 is the␣
↪latest
#     print("'models' module reloaded."); print("Model_2 __init__ signature:",␣
↪inspect.signature(Model_2.__init__))
#     BASELINE_MODEL_CLASS = Model_2
# else:
#     print("Importing 'models' module..."); import models; from models import␣
↪Model_2
#     BASELINE_MODEL_CLASS = Model_2
```

```
#     print("'models' module imported."); print("Model_2 __init__ signature:",␣
↪inspect.signature(Model_2.__init__))
```

[73]:
```
# ==============================================================================
# == Optimizer Test ==
# ==============================================================================
print("\n--- 5.2.4 Testing Optimizers ---")

# Define optimizers to test
# Note: We may ideally want to re-tune LR slightly for SGD/RMSprop,
# but for a direct comparison, we often start with the same LR found for Adam.
# --- Redefine optimizers dictionary without lambda for simplicity ---
optimizers_to_test = {
    "Adam": optim.Adam,
    "SGD (momentum=0.9)": optim.SGD, # Store the SGD class directly
    "RMSprop": optim.RMSprop,
}

optimizer_results = {}
Optimizer_Test_Model_Class = BASELINE_MODEL_CLASS
Optimizer_Activation_Class = OPTIMAL_ACTIVATION_FN
Optimizer_Norm_Type = OPTIMAL_NORM_TYPE

for optim_name, optim_class in optimizers_to_test.items():
    print(f"\n-- Testing Optimizer: {optim_name} --")

    model_optim = Optimizer_Test_Model_Class(
        activation_fn=Optimizer_Activation_Class,
        activation_name=best_act_name,
        norm_layer_type=Optimizer_Norm_Type
    ).to(DEVICE)

    criterion_optim = nn.CrossEntropyLoss()
    print(f"Using config: Activation={best_act_name}, Norm={best_norm_name},␣
↪Init=Default")

    # --- Instantiate Optimizer Correctly ---
    # Create optimizer instance here, handling SGD parameters specifically
    if optim_class == optim.SGD:
        optimizer_instance = optim.SGD(model_optim.parameters(),␣
↪lr=BASELINE_LR, momentum=0.9, weight_decay=BASELINE_WD)
        print(f"Instantiated SGD with momentum=0.9, LR={BASELINE_LR},␣
↪WD={BASELINE_WD}")
    else: # For Adam, RMSprop
        optimizer_instance = optim_class(model_optim.parameters(),␣
↪lr=BASELINE_LR, weight_decay=BASELINE_WD)
```

```python
        print(f"Instantiated {optim_name} with LR={BASELINE_LR}, WD={BASELINE_WD}")


    # --- Modify run_component_training to accept an INSTANCE ---
    # (Need to adjust the helper function definition as well)

    # --- OR Adjust how we call the helper (Easier) ---
    # We will pass the CLASS to the helper, and it will instantiate it.
    # BUT, the helper needs modification to handle SGD momentum.
    # Let's stick to the original helper and instantiate here, then run manually.

    # --- Training Loop (Manual - since helper expects class) ---
    history = {'train_loss': [], 'val_loss': [], 'val_acc': []}
    print(f"Starting training: {optim_name} - {N_EPOCHS_COMPONENT_TEST} epochs")
    train_start_time = time.time()

    for epoch in range(N_EPOCHS_COMPONENT_TEST):
        epoch_start_time = time.time()
        model_optim.train()
        running_loss = 0.0
        for batch in train_loader:
            inputs, targets = batch
            inputs, targets = inputs.to(DEVICE), targets.to(DEVICE)
            optimizer_instance.zero_grad() # Use the created instance
            outputs = model_optim(inputs)
            loss = criterion_optim(outputs, targets)
            loss.backward()
            optimizer_instance.step() # Use the created instance
            running_loss += loss.item() * inputs.size(0)
        epoch_train_loss = running_loss / len(train_loader.dataset)
        history['train_loss'].append(epoch_train_loss)

        epoch_val_loss, epoch_val_acc = evaluate(model_optim, val_loader, criterion_optim, DEVICE)
        history['val_loss'].append(epoch_val_loss)
        history['val_acc'].append(epoch_val_acc)
        epoch_end_time = time.time()
        if (epoch + 1) % 5 == 0 or epoch == 0 or (epoch + 1 == N_EPOCHS_COMPONENT_TEST):
            print(f"  Epoch {epoch+1}/{N_EPOCHS_COMPONENT_TEST} -> Train Loss: {epoch_train_loss:.4f}, Val Loss: {epoch_val_loss:.4f}, Val Acc: {epoch_val_acc:.4f} ({(epoch_end_time - epoch_start_time):.2f}s)")

    train_end_time = time.time()
```

```
    print(f"Finished training Optim_{optim_name.split(' ')[0]}. Total time:␣
↪{(train_end_time - train_start_time):.2f}s")
    # --- End Manual Training Loop ---

    optimizer_results[optim_name] = history
    component_results[f"Optim_{optim_name.split(' ')[0]}"] = history
```

```
--- 5.2.4 Testing Optimizers ---

-- Testing Optimizer: Adam --
Using config: Activation=GELU, Norm=None, Init=Default
Instantiated Adam with LR=0.001, WD=0.0001
Starting training: Adam - 15 epochs
  Epoch 1/15 -> Train Loss: 0.9565, Val Loss: 0.9225, Val Acc: 0.6494 (8.34s)
  Epoch 5/15 -> Train Loss: 0.8742, Val Loss: 0.8951, Val Acc: 0.6602 (8.04s)
  Epoch 10/15 -> Train Loss: 0.8392, Val Loss: 0.8997, Val Acc: 0.6603 (8.39s)
  Epoch 15/15 -> Train Loss: 0.8122, Val Loss: 0.9014, Val Acc: 0.6589 (8.08s)
Finished training Optim_Adam. Total time: 122.21s

-- Testing Optimizer: SGD (momentum=0.9) --
Using config: Activation=GELU, Norm=None, Init=Default
Instantiated SGD with momentum=0.9, LR=0.001, WD=0.0001
Starting training: SGD (momentum=0.9) - 15 epochs
  Epoch 1/15 -> Train Loss: 1.2140, Val Loss: 1.0785, Val Acc: 0.6026 (6.16s)
  Epoch 5/15 -> Train Loss: 0.9669, Val Loss: 0.9646, Val Acc: 0.6372 (6.11s)
  Epoch 10/15 -> Train Loss: 0.9367, Val Loss: 0.9394, Val Acc: 0.6462 (6.15s)
  Epoch 15/15 -> Train Loss: 0.9218, Val Loss: 0.9283, Val Acc: 0.6488 (6.12s)
Finished training Optim_SGD. Total time: 92.32s

-- Testing Optimizer: RMSprop --
Using config: Activation=GELU, Norm=None, Init=Default
Instantiated RMSprop with LR=0.001, WD=0.0001
Starting training: RMSprop - 15 epochs
  Epoch 1/15 -> Train Loss: 0.9468, Val Loss: 0.9238, Val Acc: 0.6473 (7.09s)
  Epoch 5/15 -> Train Loss: 0.8729, Val Loss: 0.8966, Val Acc: 0.6598 (7.13s)
  Epoch 10/15 -> Train Loss: 0.8376, Val Loss: 0.9004, Val Acc: 0.6609 (7.13s)
  Epoch 15/15 -> Train Loss: 0.8120, Val Loss: 0.9055, Val Acc: 0.6571 (7.22s)
Finished training Optim_RMSprop. Total time: 107.41s
```

[74]:
```
# --- Analyze Optimizer Results ---
print("\n--- Optimizer Results Summary ---")
print(f"{'Optimizer':<25} | {'Final Val Loss':<15} | {'Final Val Acc':<15} |␣
↪{'Max Val Acc':<15}")
print("-" * 75)
best_optim_name = ""
best_optim_max_acc = -1.0
```

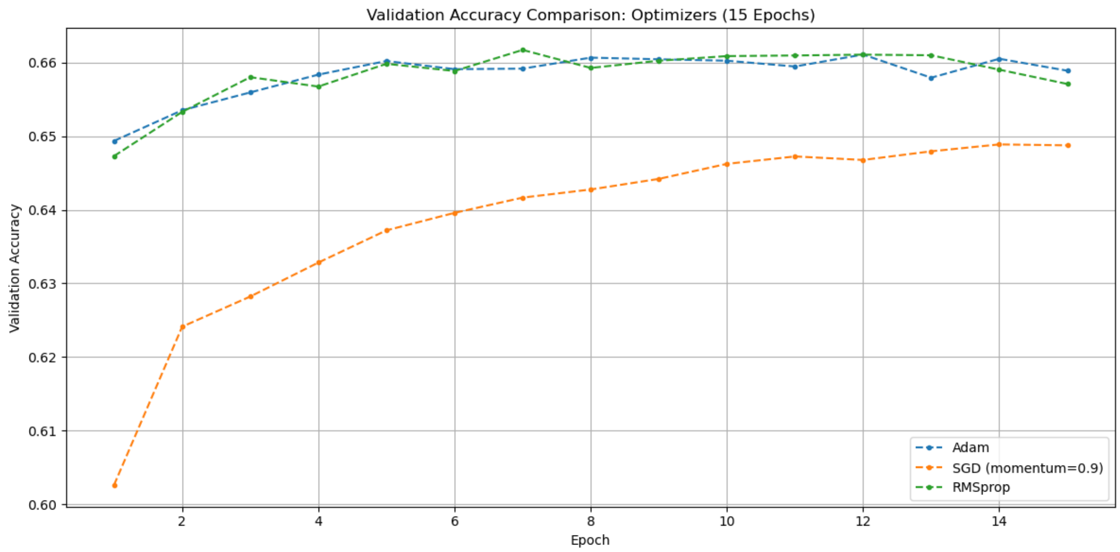
```python
for name, history in optimizer_results.items():
    final_val_loss = history['val_loss'][-1]
    final_val_acc = history['val_acc'][-1]
    max_val_acc = max(history['val_acc'])
    print(f"{name:<25} | {final_val_loss:<15.4f} | {final_val_acc:<15.4f} |␣
↪{max_val_acc:<15.4f}")
    if max_val_acc > best_optim_max_acc:
        best_optim_max_acc = max_val_acc
        best_optim_name = name

print(f"\nBest Optimizer (based on Max Validation Accuracy): {best_optim_name}␣
↪({best_optim_max_acc:.4f})")
# Store the best optimizer class for the final model
if best_optim_name in optimizers_to_test:
    # Need to handle the lambda case for storing
    if isinstance(optimizers_to_test[best_optim_name], type): # Check if it's a␣
↪class like Adam/RMSprop
         OPTIMAL_OPTIMIZER_CLASS = optimizers_to_test[best_optim_name]
    else: # It's the SGD lambda
         OPTIMAL_OPTIMIZER_CLASS = optim.SGD # Store the base SGD class
         print("Note: Best optimizer was SGD. Storing base class. Remember to␣
↪use momentum=0.9.")
    print(f"Stored OPTIMAL_OPTIMIZER_CLASS: {OPTIMAL_OPTIMIZER_CLASS}")
else:
     print("Error: Best optimizer name not found!")
     OPTIMAL_OPTIMIZER_CLASS = optim.Adam # Fallback
```

```
--- Optimizer Results Summary ---
Optimizer                 | Final Val Loss  | Final Val Acc   | Max Val Acc
---------------------------------------------------------------------------
Adam                      | 0.9014          | 0.6589          | 0.6611
SGD (momentum=0.9)        | 0.9283          | 0.6488          | 0.6489
RMSprop                   | 0.9055          | 0.6571          | 0.6617

Best Optimizer (based on Max Validation Accuracy): RMSprop (0.6617)
Stored OPTIMAL_OPTIMIZER_CLASS: <class 'torch.optim.rmsprop.RMSprop'>
```

[75]:

```python
# --- Plot Optimizer comparison ---
plt.figure(figsize=(12, 6))
for name, history in optimizer_results.items():
    plt.plot(range(1, N_EPOCHS_COMPONENT_TEST + 1), history['val_acc'],␣
↪label=name, marker='.', linestyle='--')

plt.title(f'Validation Accuracy Comparison: Optimizers␣
↪({N_EPOCHS_COMPONENT_TEST} Epochs)')
```

```
plt.xlabel('Epoch')
plt.ylabel('Validation Accuracy')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig(os.path.join(PLOT_SAVE_DIR, 'component_optimizer_comparison.png'))
plt.show()
```




[76]:
```
# ============================================================================
# == End of Component Optimization ==
# ============================================================================
print("\nComponent optimization tests complete.")
print("Final selected components based on these tests:")
print(f" - Model Architecture: {BASELINE_MODEL_CLASS.__name__}")
print(f" - Learning Rate: {OPTIMAL_LR}")
print(f" - Weight Decay: {OPTIMAL_WEIGHT_DECAY}")
print(f" - Initialization: {best_init_name}") # From previous test summary
print(f" - Activation Function: {best_act_name}") # From previous test summary
print(f" - Normalization: {best_norm_name}") # From previous test summary
print(f" - Optimizer: {best_optim_name}")
```

```
Component optimization tests complete.
Final selected components based on these tests:
 - Model Architecture: Model_2
 - Learning Rate: 0.001
 - Weight Decay: 0.0001
 - Initialization: Default (Kaiming Uniform for ReLU)
 - Activation Function: GELU
```

```
- Normalization: None
- Optimizer: RMSprop
```

[77]:
```python
print("\n\n" + "="*50)
print("--- Phase 6: Final Model Training & Evaluation ---")
print("="*50 + "\n")

# --- Final Configuration ---
FINAL_MODEL_CLASS = BASELINE_MODEL_CLASS      # Model_2
FINAL_ACTIVATION_FN = OPTIMAL_ACTIVATION_FN # GELU Class
FINAL_ACTIVATION_NAME = best_act_name        # 'GELU' String
FINAL_NORM_TYPE = OPTIMAL_NORM_TYPE           # None
FINAL_OPTIMIZER_CLASS = OPTIMAL_OPTIMIZER_CLASS # RMSprop Class
FINAL_LR = OPTIMAL_LR                         # 0.001
FINAL_WD = OPTIMAL_WEIGHT_DECAY             # 0.0001
# FINAL_INIT = best_init_name # Default - no special function needed
```

```
==================================================
--- Phase 6: Final Model Training & Evaluation ---
==================================================
```

[84]:
```python
# Number of epochs for final training - adjust as needed
# Monitor validation loss/accuracy to decide when to stop or implement early␣
↪stopping
N_EPOCHS_FINAL = 15

print("--- Final Model Configuration ---")
print(f" Model: {FINAL_MODEL_CLASS.__name__}")
print(f" Activation: {FINAL_ACTIVATION_NAME}")
print(f" Normalization: {FINAL_NORM_TYPE if FINAL_NORM_TYPE else 'None'}")
print(f" Optimizer: {FINAL_OPTIMIZER_CLASS.__name__}")
print(f" Learning Rate: {FINAL_LR}")
print(f" Weight Decay: {FINAL_WD}")
print(f" Training Epochs: {N_EPOCHS_FINAL}")
print(" Initialization: Default")
print("-" * 30)
```

```
--- Final Model Configuration ---
 Model: Model_2
 Activation: GELU
 Normalization: None
 Optimizer: RMSprop
 Learning Rate: 0.001
 Weight Decay: 0.0001
 Training Epochs: 15
 Initialization: Default
```

------------------------------

[85]:
```
# --- Instantiate Final Model ---
final_model = FINAL_MODEL_CLASS(
    activation_fn=FINAL_ACTIVATION_FN,
    activation_name=FINAL_ACTIVATION_NAME,
    norm_layer_type=FINAL_NORM_TYPE
).to(DEVICE)

criterion_final = nn.CrossEntropyLoss()
optimizer_final = FINAL_OPTIMIZER_CLASS(final_model.parameters(), lr=FINAL_LR,
 ↪weight_decay=FINAL_WD)

print(f"Instantiated Final Model:\n{final_model}")
```

```
Instantiated Final Model:
Model Architecture: Model_2
  Input Size: 90
  Num Classes: 10
  Activation: GELU
  Normalization: None
Total Trainable Parameters: 91,658
```

[86]:
```
# --- Final Training ---
print("\n--- Starting Final Model Training ---")
final_history = {'train_loss': [], 'val_loss': [], 'val_acc': []}
final_training_start_time = time.time()

for epoch in range(N_EPOCHS_FINAL):
    epoch_start_time = time.time()
    # Training phase
    final_model.train()
    running_loss = 0.0
    for batch in train_loader: # Train on original training set
        inputs, targets = batch
        inputs, targets = inputs.to(DEVICE), targets.to(DEVICE)
        optimizer_final.zero_grad()
        outputs = final_model(inputs)
        loss = criterion_final(outputs, targets)
        loss.backward()
        optimizer_final.step()
        running_loss += loss.item() * inputs.size(0)
    epoch_train_loss = running_loss / len(train_loader.dataset)
    final_history['train_loss'].append(epoch_train_loss)

    # Validation phase (Monitor on validation set)
```

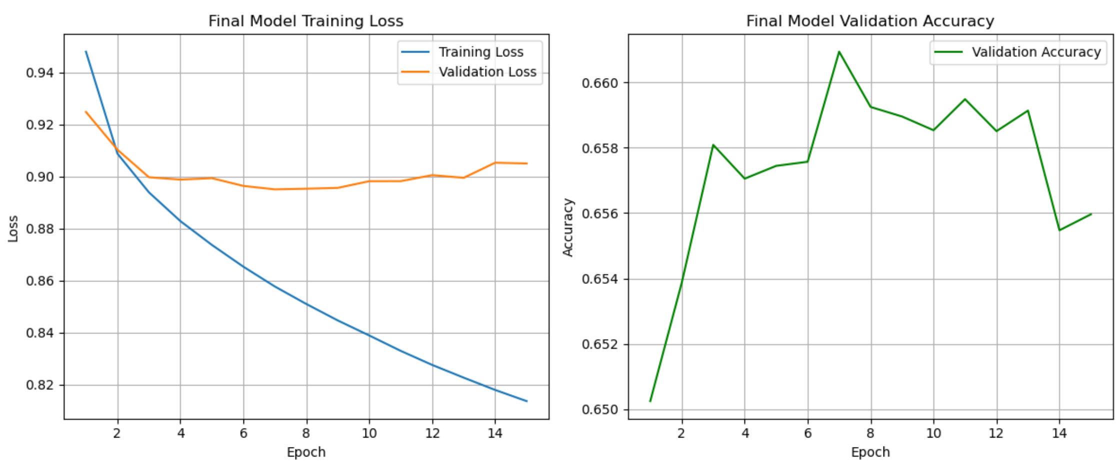
```
    epoch_val_loss, epoch_val_acc = evaluate(final_model, val_loader,␣
↪criterion_final, DEVICE)
    final_history['val_loss'].append(epoch_val_loss)
    final_history['val_acc'].append(epoch_val_acc)
    epoch_end_time = time.time()

    print(f"Epoch {epoch+1}/{N_EPOCHS_FINAL} -> Train Loss: {epoch_train_loss:.
↪4f}, Val Loss: {epoch_val_loss:.4f}, Val Acc: {epoch_val_acc:.4f}␣
↪({(epoch_end_time - epoch_start_time):.2f}s)")

    # Basic Early Stopping Check (Example - can be made more robust)
    # Stop if validation loss hasn't improved for N epochs (e.g., patience=5)
    patience = 5
    if epoch >= patience:
        # Check if current val_loss is worse than loss 'patience' epochs ago
        if epoch_val_loss > min(final_history['val_loss'][-(patience+1):-1]):
             print(f"Validation loss has not improved for {patience} epochs.␣
↪Consider stopping early.")
             # break # Uncomment to actually stop training

final_training_end_time = time.time()
print(f"\nFinished final training. Total time: {(final_training_end_time -␣
↪final_training_start_time)/60:.2f} minutes.")
```

```
--- Starting Final Model Training ---
Epoch 1/15 -> Train Loss: 0.9481, Val Loss: 0.9249, Val Acc: 0.6502 (8.70s)
Epoch 2/15 -> Train Loss: 0.9088, Val Loss: 0.9104, Val Acc: 0.6539 (7.52s)
Epoch 3/15 -> Train Loss: 0.8940, Val Loss: 0.8998, Val Acc: 0.6581 (7.08s)
Epoch 4/15 -> Train Loss: 0.8829, Val Loss: 0.8989, Val Acc: 0.6571 (7.29s)
Epoch 5/15 -> Train Loss: 0.8737, Val Loss: 0.8994, Val Acc: 0.6574 (7.27s)
Epoch 6/15 -> Train Loss: 0.8654, Val Loss: 0.8964, Val Acc: 0.6576 (7.15s)
Epoch 7/15 -> Train Loss: 0.8578, Val Loss: 0.8951, Val Acc: 0.6609 (7.04s)
Epoch 8/15 -> Train Loss: 0.8510, Val Loss: 0.8954, Val Acc: 0.6592 (7.23s)
Validation loss has not improved for 5 epochs. Consider stopping early.
Epoch 9/15 -> Train Loss: 0.8447, Val Loss: 0.8957, Val Acc: 0.6590 (7.03s)
Validation loss has not improved for 5 epochs. Consider stopping early.
Epoch 10/15 -> Train Loss: 0.8389, Val Loss: 0.8982, Val Acc: 0.6585 (7.12s)
Validation loss has not improved for 5 epochs. Consider stopping early.
Epoch 11/15 -> Train Loss: 0.8330, Val Loss: 0.8983, Val Acc: 0.6595 (7.08s)
Validation loss has not improved for 5 epochs. Consider stopping early.
Epoch 12/15 -> Train Loss: 0.8276, Val Loss: 0.9006, Val Acc: 0.6585 (7.07s)
Validation loss has not improved for 5 epochs. Consider stopping early.
Epoch 13/15 -> Train Loss: 0.8227, Val Loss: 0.8996, Val Acc: 0.6591 (7.08s)
Validation loss has not improved for 5 epochs. Consider stopping early.
Epoch 14/15 -> Train Loss: 0.8180, Val Loss: 0.9053, Val Acc: 0.6555 (7.11s)
Validation loss has not improved for 5 epochs. Consider stopping early.
```

```
Epoch 15/15 -> Train Loss: 0.8137, Val Loss: 0.9051, Val Acc: 0.6560 (7.07s)
Validation loss has not improved for 5 epochs. Consider stopping early.

Finished final training. Total time: 1.81 minutes.
```

[87]:

```
# --- Plot Final Training History ---
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1); plt.plot(range(1, len(final_history['train_loss']) + 1),
 ↪final_history['train_loss'], label='Training Loss'); plt.plot(range(1,
 ↪len(final_history['val_loss']) + 1), final_history['val_loss'],
 ↪label='Validation Loss'); plt.xlabel('Epoch'); plt.ylabel('Loss'); plt.
 ↪title('Final Model Training Loss'); plt.legend(); plt.grid(True)
plt.subplot(1, 2, 2); plt.plot(range(1, len(final_history['val_acc']) + 1),
 ↪final_history['val_acc'], label='Validation Accuracy', color='green'); plt.
 ↪xlabel('Epoch'); plt.ylabel('Accuracy'); plt.title('Final Model Validation
 ↪Accuracy'); plt.legend(); plt.grid(True)
plt.tight_layout(); plt.savefig(os.path.join(PLOT_SAVE_DIR,
 ↪'final_model_training_history.png')); plt.show()
```




[88]:

```
# --- Final Evaluation on Test Set ---
print("\n--- Evaluating Final Model on Test Set ---")
test_loss, test_accuracy = evaluate(final_model, test_loader, criterion_final,
 ↪DEVICE)
print(f"\nPerformance on the HELD-OUT TEST SET:")
print(f"  Test Loss: {test_loss:.4f}")
print(f"  Test Accuracy: {test_accuracy:.4f} ({test_accuracy*100:.2f}%)")
```

```
--- Evaluating Final Model on Test Set ---

Performance on the HELD-OUT TEST SET:
  Test Loss: 0.8993
```

```
Test Accuracy: 0.6605 (66.05%)
```

[89]:

```
# --- (Optional) Save the Final Model ---
model_save_path = os.path.join("../results/models/", "final_optimized_model.
 ↪pth")
print(f"\nSaving final model state_dict to {model_save_path}")
torch.save(final_model.state_dict(), model_save_path)
print("Model saved.")
```

```
Saving final model state_dict to ../results/models/final_optimized_model.pth
Model saved.
```