Experiment-3

Activation Functions & Optimization

1. Aim

To study and compare different activation functions and optimization algorithms by training the same MLP on the Fashion-MNIST dataset. To analyze the performance of ReLU, Sigmoid, and Tanh activations with SGD and Adam optimizers. To evaluate their impact on training dynamics using overlaid loss/accuracy curves and gradient-flow visualizations.

2. Theory

Neural network learning is governed by the interaction between activation functions and optimization algorithms. Activation functions introduce non-linearity into the network, enabling it to learn complex input–output mappings, while optimization algorithms determine how model parameters are updated to minimize the loss function. The choice of activation function and optimizer significantly affects gradient propagation, convergence speed, training stability, and overall model performance. This experiment studies these effects using a Multilayer Perceptron trained on the Fashion-MNIST dataset.

2.1 Activation Functions

In a Multilayer Perceptron (MLP), the activation function determines the output of a neuron based on the weighted sum of its inputs and bias. Activation functions transform the output of neurons before passing information to subsequent layers, thereby influencing the network's representational capability.

In this experiment, three widely used activation functions, namely Sigmoid, Hyperbolic Tangent (Tanh), and Rectified Linear Unit (ReLU), are studied to compare their mathematical properties and practical behavior during training on the Fashion-MNIST dataset.

Mathematical Formulation

Let the net input to a neuron be:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

where w_i represents weights, x_i represents input features, and b is the bias term. The output of the neuron is obtained by applying an activation function $f(z)$.

I. Sigmoid Activation Function: -

The sigmoid activation function is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

The output of the sigmoid function lies in the range (0, 1). It was commonly used in early neural networks due to its smooth and differentiable nature. However, for large positive or negative input values, the function saturates, resulting in very small gradients and slow convergence during training. Figure 1 shows the behavior of the Sigmoid activation function.

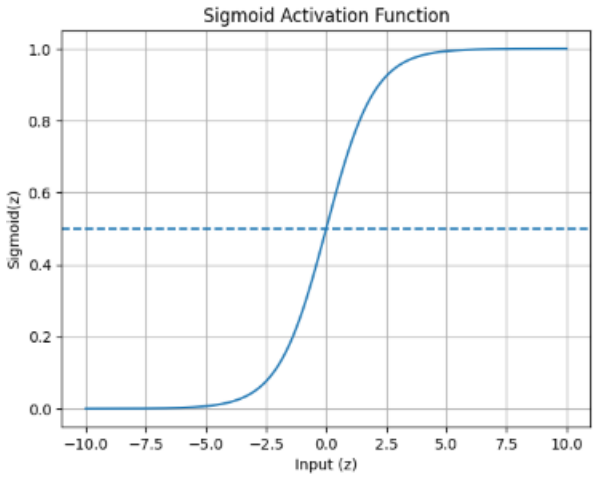


Figure 1: Sigmoid Activation Function

Merits of Sigmoid Activation Function

- **Smooth and differentiable:**
  The sigmoid function is continuously differentiable, which enables the use of gradient-based optimization techniques during training.

- **Output in the range (0, 1):**
  The output lies between 0 and 1, making it suitable for probability-based interpretation in binary classification problems.

- **Simple mathematical formulation:**
  The function is easy to understand and implement due to its straightforward mathematical expression.

Demerits of Sigmoid Activation Function

- **Vanishing gradient problem:**
  For large positive or negative input values, the gradients become extremely small, slowing down the learning process.

- **Non-zero-centred output:**
  The output is not zero-centred, which can cause inefficient weight updates and slower convergence during training.

- **Saturation at extreme values:**
  The function saturates at both ends of its output range, reducing its effectiveness in deep neural networks.

II. Hyperbolic Tangent (Tanh) Activation Function: -

The hyperbolic tangent activation function is given by:

$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

The output range of tanh is (−1,1), making it zero-centred. Compared to sigmoid, tanh provides better gradient flow near zero and generally results in faster convergence. However, it still suffers from gradient saturation for large input values. Figure 2 illustrates the behavior of the Tanh activation function.

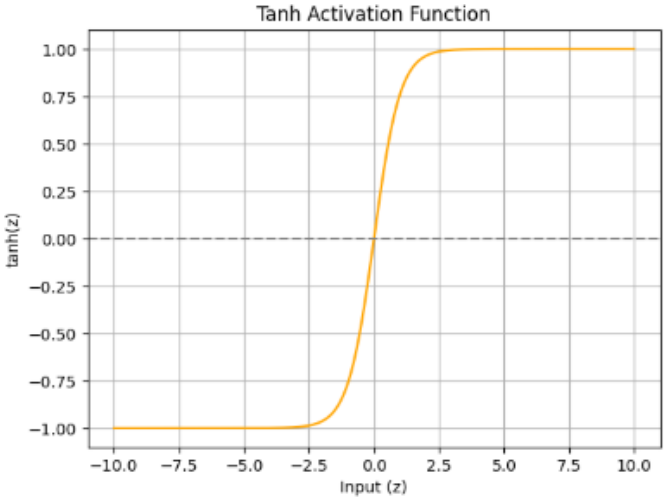


Figure 2: Tanh Activation Function

Merits of Hyperbolic Tangent (Tanh) Activation Function

- **Zero-centred output:**
  Helps improve gradient flow and speeds up convergence compared to the sigmoid.
- **Smooth and differentiable:**
  The function supports stable gradient-based optimization due to its continuous differentiability.
- **Stronger gradients near zero:**
  Compared to sigmoid, tanh provides larger gradients around zero, which enhances learning efficiency.

Demerits of Hyperbolic Tangent (Tanh) Activation Function

- **Vanishing gradient problem:**
  Gradients become very small for large input values, which can slow down training in deeper networks.
- **Saturation at extreme values:**
  The function saturates at high positive and negative values, leading to reduced learning speed.
- **Higher computational cost:**
  The use of exponential functions makes tanh computationally more expensive than ReLU.

III. Rectified Linear Unit (ReLU): -

The Rectified Linear Unit activation function is defined as:

$$\text{ReLU}(z) = \max(0, z)$$

It can also be expressed in piecewise form as:

$$\text{ReLU}(z) = \begin{cases} 0, & \text{if } z < 0 \\ z, & \text{if } z \geq 0 \end{cases}$$

ReLU outputs zero for negative input values and a linear output for positive values. This behaviour significantly reduces the vanishing gradient problem and improves training efficiency. Due to its simplicity and effectiveness, ReLU is widely used in modern deep learning models. Figure 3 shows the ReLU activation function curve.

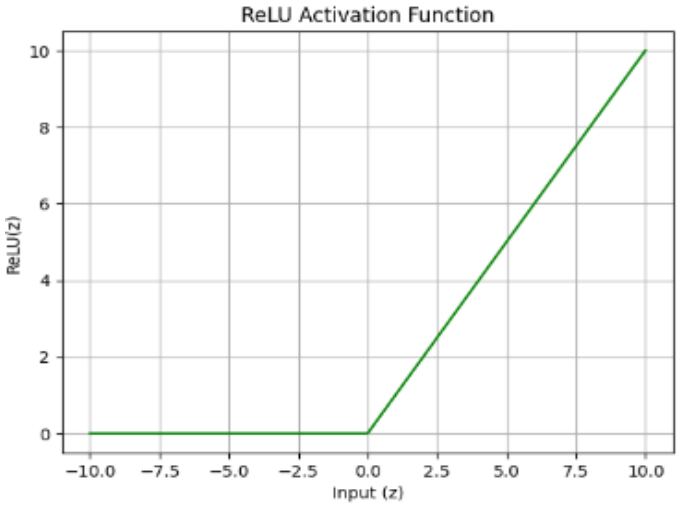


Figure 3: ReLU Activation Function

Merits of ReLU Activation Function

- **Introduces piecewise linearity:**
  The composition of ReLU activations across multiple layers enables neural networks to model complex and non-linear decision boundaries.
- **Efficient training:**
  Reduces vanishing gradient issues and accelerates convergence.
- **Sparsity in activations:**
  ReLU activates only a subset of neurons, improving computational efficiency and potentially providing a mild regularizing effect.
- **Simple and fast computation:**
  The function involves only a threshold operation, making it computationally efficient

Demerits of ReLU Activation Function

- **Dying ReLU problem:**
  Neurons can become inactive and permanently output zero, preventing further learning.
- **Non-differentiable at zero:**
  The function is not differentiable at zero, although this is handled using sub-gradients in practice.
- **Unbounded output for positive inputs:**
  The output for positive inputs is unbounded, which may lead to exploding activations if not properly controlled.

Comparison of Activation Functions

Activation Function	Mathematical Expression	Output Range	Merits	Demerits
Sigmoid	$\sigma(z) = \frac{1}{1 + e^{-z}}$	(0, 1)	Smooth and differentiable; suitable for probability-based outputs	Suffers from vanishing gradient and slow convergence
Tanh	$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	(−1, 1)	Zero-centred output; better gradient flow than sigmoid	Vanishing gradient for large input values
ReLU	$\mathrm{ReLU}(z) = \max(0, z)$	[0, ∞)	Fast convergence; reduces the vanishing gradient problem	Dying ReLU problem; non-differentiable at zero

2.2 Optimization Algorithms

Optimization algorithms are used to minimize a neural network's loss function by iteratively adjusting its parameters (weights and biases). During training, the optimizer determines the direction and magnitude of parameter updates based on the gradients of the loss function with respect to the model parameters. An efficient optimization algorithm ensures faster convergence, numerical stability, and improved generalization performance**.** In this experiment, Stochastic Gradient Descent (SGD) and Adaptive Moment Estimation (Adam) optimizers are studied and compared while training a Multilayer Perceptron (MLP) on the Fashion-MNIST dataset.

I. Stochastic Gradient Descent (SGD): -

Stochastic Gradient Descent is a fundamental optimization algorithm based on the theory of stochastic approximation, introduced by Herbert Robbins and Sutton Monro (1951). Unlike batch gradient descent, Stochastic Gradient Descent (SGD) updates model parameters using a single training example at a time, reducing computational cost and introducing stochasticity into the optimization process.

The parameter update rule is:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t)$$

where η is the learning rate.

SGD is simple and memory-efficient, and the randomness in updates can help escape shallow local minima. However, it is sensitive to the learning rate and may converge slowly or exhibit oscillations during training.

Merits of Stochastic Gradient Descent (SGD)

- **Simple and easy to implement:**
  Stochastic Gradient Descent has a straightforward update rule, making it easy to understand and implement.
- **Memory efficient:**
  SGD requires very little additional memory, as it does not store past gradients or extra parameters.
- **Ability to escape shallow local minima:**
  The stochastic nature of parameter updates introduces noise, which can help the algorithm escape shallow local minima.
- **Suitable for large datasets:**
  SGD processes one sample at a time, making it efficient for large-scale learning problems.

Demerits of Stochastic Gradient Descent (SGD)

- **Sensitivity to learning rate:**
  Choosing an inappropriate learning rate can result in slow convergence or unstable training.
- **Slow convergence:**
  SGD may require many iterations to reach the optimal solution, especially for complex problems.
- **No adaptive learning rate:**
  A fixed learning rate is used for all parameters, which may limit performance if not carefully tuned.
- **Oscillations during training:**
  SGD can exhibit oscillatory behavior near minima due to noisy gradient updates.

II. Adaptive Moment Estimation (Adam): -

Adaptive Moment Estimation (Adam) was proposed by Diederik P. Kingma and Jimmy Lei Ba (2015). It is an advanced optimization algorithm that combines the benefits of momentum-based methods and adaptive learning rate techniques. Adam adapts the learning rate for each parameter by maintaining exponentially decaying averages of past gradients and squared gradients.

The update rule for Adam is:

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m_t}}{\sqrt{\hat{v_t}} + \epsilon}$$

Where:

θ_t is the model parameter at iteration t

θ_{t+1} is the updated parameter

η is the learning rate

$\hat{m_t}$ is the bias-corrected first moment estimate (mean of gradients)

$\hat{v_t}$ is the bias-corrected second moment estimate (mean of squared gradients)

ϵ is a small constant added for numerical stability

Adam provides faster convergence, stable training, and performs well with noisy or sparse gradients. Due to these advantages, it is widely used in deep learning applications.

Merits of Adaptive Moment Estimation (Adam)

- **Adaptive learning rates:**
  Adam automatically adjusts the learning rate for each parameter based on first- and second-moment estimates.
- **Fast convergence:**
  The optimizer converges faster than traditional gradient-based methods, especially on deep networks.
- **Robust to noisy gradients:**
  Adam performs well even when gradients are noisy or sparse.
- **Minimal hyperparameter tuning:**
  Default parameter values often work well, reducing the need for extensive tuning.

Demerits of Adaptive Moment Estimation (Adam)

- **Higher computational cost:**
  Adam requires additional computations to maintain moving averages of gradients and squared gradients.
- **Increased memory usage:**
  Extra memory is needed to store moment estimates for each parameter.

- **Risk of overfitting:**
  Adam may converge too aggressively, leading to overfitting on training data.
- **Less theoretical convergence guarantees:**
  Compared to SGD, Adam has weaker theoretical convergence properties in some cases.

Comparison of Optimization Algorithms

Optimization Algorithm	Learning Rate Type	Key Characteristics	Merits	Demerits
Stochastic Gradient Descent (SGD)	Fixed	Updates parameters using stochastic gradient updates	Simple, memory-efficient, good generalization	Sensitive to learning rate; slow convergence
Adaptive Moment Estimation (Adam)	Adaptive	Uses first and second moment estimates	Fast convergence; robust to noisy gradients	Higher computation and memory cost

3. Code and Result with Parameters:

Activation: relu

Optimizer: adam

Learning Rate: 0.001

1 Parameterized Training Dynamics Notebook

Goal: Compare *how* and *why* different training choices behave

This notebook investigates the training dynamics of various hyperparameter configurations: - **Activation functions**: How they affect gradient flow - **Optimizers**: How they affect convergence speed - **Learning rate schedulers**: How they affect final performance - **Learning rates**: How they affect stability

1.1 Core Design Philosophy

1. **One variable changes at a time** → comparisons are causal, not anecdotal
2. **Short, repeatable runs** → focus on dynamics, not peak accuracy
3. **Everything observable is logged** → loss alone is insufficient

[2]:

```
# Parameters
ACTIVATION = "relu"
OPTIMIZER = "adam"
LEARNING_RATE = 0.001
SCHEDULER = "none"
EXPERIMENT_ID = "relu_adam_0e001"
EPOCHS = 30
BATCH_SIZE = 128
SEED = 42
```

[3]:

```
print(f"\nExperiment Configuration:")
print(f"    ID: {EXPERIMENT_ID}")
print(f"    Activation: {ACTIVATION}")
print(f"    Optimizer: {OPTIMIZER}")
print(f"    Scheduler: {SCHEDULER}")
print(f"    Learning Rate: {LEARNING_RATE}")
print(f"    Epochs: {EPOCHS}")
print(f"    Batch Size: {BATCH_SIZE}")
print(f"    Seed: {SEED}")
```

```
Experiment Configuration:
ID: relu_adam_0e001

Activation: relu
Optimizer: adam
Scheduler: none
Learning Rate: 0.001
Epochs: 30
Batch Size: 128
Seed: 42
```

[4]:
```
# Imports and Setup
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from collections import defaultdict
import time
import json
import os
from pathlib import Path

# Set style
sns.set_style("whitegrid")
plt.rcParams['figure.figsize'] = (12, 6)

# Set random seeds for reproducibility
torch.manual_seed(SEED)
np.random.seed(SEED)
if torch.cuda.is_available():
    torch.cuda.manual_seed(SEED)

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")
```

```
Using device: cuda
```

1.2 Dataset Preparation

Fashion-MNIST is used because: - Simple enough to train fast (28x28 grayscale images) - Hard enough to expose failures in configuration - Well-known baseline for MLPs

We normalize to [0,1] and flatten to 784-dimensional vectors.

[5]:
```
# Data preprocessing
transform = transforms.Compose([
```

```
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))       # Normalize to [-1, 1]
])
# Load Fashion-MNIST
train_dataset = datasets.FashionMNIST(
    root='./data',
    train=True,
    download=True,
    transform=transform
)

test_dataset = datasets.FashionMNIST(
    root='./data',
    train=False,
    download=True,
    transform=transform
)

# Create validation split (10% of training data)
train_size = int(0.9 * len(train_dataset))
val_size = len(train_dataset) - train_size
train_subset, val_subset = torch.utils.data.random_split(
    train_dataset,
    [train_size, val_size],
    generator=torch.Generator().manual_seed(SEED)
)

# Create data loaders
train_loader = DataLoader(train_subset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(val_subset, batch_size=BATCH_SIZE, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False)

print(f"Dataset Statistics:")
print(f"    Training samples: {len(train_subset)}")
print(f"    Validation samples: {len(val_subset)}")
print(f"    Test samples: {len(test_dataset)}")
print(f"    Classes: 10 (Fashion-MNIST categories)")
print(f"    Input dimension: 784 (28x28 flattened)")
```

```
Dataset Statistics:
  Training samples: 54000
  Validation samples: 6000
  Test samples: 10000
  Classes: 10 (Fashion-MNIST categories)
  Input dimension: 784 (28x28 flattened)
```

1.3 Model Architecture (Frozen Backbone)

To ensure **fair comparison**, we use the same architecture for all runs:

Input (784)
→ Linear(784 → 256)
→ Activation
→ Linear(256 → 128)
→ Activation
→ Linear(128 → 10)

Only the activation function varies based on the parameter.

[6]:
```
# Activation function factory
def get_activation(name):
    activations = {
        'relu': nn.ReLU(),
        'sigmoid': nn.Sigmoid(),
        'tanh': nn.Tanh(),
        'leaky_relu': nn.LeakyReLU(0.1),
        'gelu': nn.GELU(),
        'elu': nn.ELU(),
    }
    if name not in activations:
        raise ValueError(f"Unknown activation: {name}. Available:
 s{list(activations.keys())}")
    return activations[name]

# Model definition
class FashionMLP(nn.Module):
    def __init__(self, activation_name='relu'):
        super(FashionMLP, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(784, 256)
        self.act1 = get_activation(activation_name)
        self.fc2 = nn.Linear(256, 128)
        self.act2 = get_activation(activation_name)
        self.fc3 = nn.Linear(128, 10)

        # Store layer names for gradient tracking
        self.layer_names = ['fc1', 'fc2', 'fc3']

    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.act1(x)
        x = self.fc2(x)
        x = self.act2(x)
        x = self.fc3(x)
```

```
        return x

# Initialize model
model = FashionMLP(ACTIVATION).to(device)
print(f"\nModel Architecture:")
print(model)
print(f"\nTotal parameters: {sum(p.numel() for p in model.parameters()):,}")
```

```
Model Architecture:
FashionMLP(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (fc1): Linear(in_features=784, out_features=256, bias=True)
  (act1): ReLU()
  (fc2): Linear(in_features=256, out_features=128, bias=True)
  (act2): ReLU()
  (fc3): Linear(in_features=128, out_features=10, bias=True)
)

Total parameters: 235,146
```

1.4 Optimizer and Scheduler Setup

Different optimizers have different convergence characteristics: - **SGD**: Baseline, requires careful tuning - **SGD + Momentum**: Faster convergence, smoother updates - **Adam**: Adaptive learning rates, robust default - **AdamW**: Adam with weight decay decoupling - **RMSprop**: Good for RNNs, adaptive learning rates

Schedulers can improve final performance or convergence speed.

[7]:
```
# Optimizer factory
def get_optimizer(name, parameters, lr):
    if name == 'sgd':
        return optim.SGD(parameters, lr=lr)
    elif name == 'sgd_momentum':
        return optim.SGD(parameters, lr=lr, momentum=0.9)
    elif name == 'adam':
        return optim.Adam(parameters, lr=lr)
    elif name == 'adamw':
        return optim.AdamW(parameters, lr=lr)
    elif name == 'rmsprop':
        return optim.RMSprop(parameters, lr=lr)
    else:
        # Use a list of valid names for the error message
        valid_names = ['sgd', 'sgd_momentum', 'adam', 'adamw', 'rmsprop']
        raise ValueError(f"Unknown optimizer: {name}. Available: {valid_names}")

# Scheduler factory
```

```
def get_scheduler(name, optimizer, epochs):
    if name == 'none':
        return None
    schedulers = {
        'step': optim.lr_scheduler.StepLR(optimizer, step_size=epochs//3,␣
 ↪gamma=0.1),
        'cosine': optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=epochs),
        'exponential': optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.9),
    }
    if name not in schedulers:
        raise ValueError(f"Unknown scheduler: {name}. Available:␣
 ↪{list(schedulers.keys())} + 'none'")
    return schedulers[name]

# Initialize optimizer and scheduler
optimizer = get_optimizer(OPTIMIZER, model.parameters(), LEARNING_RATE)
scheduler = get_scheduler(SCHEDULER, optimizer, EPOCHS)

# Loss function
criterion = nn.CrossEntropyLoss()

print(f"Training Configuration:")
print(f"  Optimizer: {type(optimizer).__name__}")
print(f"  Scheduler: {type(scheduler).__name__ if scheduler else 'None'}")
print(f"  Loss function: CrossEntropyLoss")
print(f"  Initial LR: {LEARNING_RATE}")
```

```
Training Configuration:
  Optimizer: Adam
  Scheduler: None
  Loss function: CrossEntropyLoss
  Initial LR: 0.001
```

1.5 Training Loop with Comprehensive Logging

We track: - **Loss**: Training loss per epoch - **Accuracy**: Validation accuracy per epoch - **Learning rate**: Current LR (important for schedulers) - **Gradient statistics**: Per-layer gradient magnitude and variance - **Time**: Training time per epoch

[8]:
```
# Initialize logging structure
training_log = {
    'epoch': [],
    'train_loss': [],
    'val_accuracy': [],
    'learning_rate': [],
    'epoch_time': [],
    'gradient_stats': defaultdict(lambda: defaultdict(list)),  #␣
 ↪[layer][metric][epoch]
```

```
}

def compute_gradient_stats(model):
    """Compute per-layer gradient statistics."""
    stats = {}
    for name, param in model.named_parameters():
        if param.grad is not None:
            grad = param.grad.detach().cpu().numpy().flatten()
            stats[name] = {
                'mean_abs': float(np.mean(np.abs(grad))),
                'std': float(np.std(grad)),
                'pct_zero': float(np.mean(grad == 0) * 100),
            }
    return stats

def train_epoch(model, loader, criterion, optimizer, device):
    """Train for one epoch."""
    model.train()
    total_loss = 0

    for inputs, targets in loader:
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    return total_loss / len(loader)

def validate(model, loader, device):
    """Compute validation accuracy."""
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, targets in loader:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs, 1)
            total += targets.size(0)
            correct += (predicted == targets).sum().item()
```

```
    return 100 * correct / total

print("Starting training...\n")
```

```
Starting training...
```

[9]:
```
# Main training loop
best_val_acc = 0
best_epoch = 0

for epoch in range(1, EPOCHS + 1):
    start_time = time.time()

    # Train
    train_loss = train_epoch(model, train_loader, criterion, optimizer, device)

    # Validate
    val_acc = validate(model, val_loader, device)

    # Get current learning rate
    current_lr = optimizer.param_groups[0]['lr']

    # Compute gradient stats (run a backward pass on a single batch)
    model.train()
    sample_batch = next(iter(train_loader))
    inputs, targets = sample_batch[0].to(device), sample_batch[1].to(device)
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    loss.backward()
    grad_stats = compute_gradient_stats(model)

    # Update scheduler
    if scheduler is not None:
        scheduler.step()

    # Log metrics
    epoch_time = time.time() - start_time
    training_log['epoch'].append(epoch)
    training_log['train_loss'].append(train_loss)
    training_log['val_accuracy'].append(val_acc)
    training_log['learning_rate'].append(current_lr)
    training_log['epoch_time'].append(epoch_time)

    for layer_name, stats in grad_stats.items():
        for metric, value in stats.items():
```

```
                training_log['gradient_stats'][layer_name][metric].append(value)

    # Track best model
    if val_acc > best_val_acc:
        best_val_acc = val_acc
        best_epoch = epoch

    # Print progress
    print(f"Epoch {epoch:2d}/{EPOCHS} | "
          f"Loss: {train_loss:.4f} | "
          f"Val Acc: {val_acc:.2f}% | "
          f"LR: {current_lr:.6f} | "
          f"Time: {epoch_time:.2f}s")

print(f"\nTraining completed!")
print(f"Best validation accuracy: {best_val_acc:.2f}% at epoch {best_epoch}")
```

```
Epoch  1/30 | Loss: 0.5300 | Val Acc: 84.88% | LR: 0.001000 | Time: 9.61s
Epoch  2/30 | Loss: 0.3770 | Val Acc: 84.98% | LR: 0.001000 | Time: 12.96s
Epoch  3/30 | Loss: 0.3392 | Val Acc: 86.48% | LR: 0.001000 | Time: 10.93s
Epoch  4/30 | Loss: 0.3113 | Val Acc: 88.23% | LR: 0.001000 | Time: 12.43s
Epoch  5/30 | Loss: 0.2941 | Val Acc: 88.37% | LR: 0.001000 | Time: 11.01s
Epoch  6/30 | Loss: 0.2777 | Val Acc: 88.20% | LR: 0.001000 | Time: 11.02s
Epoch  7/30 | Loss: 0.2630 | Val Acc: 88.32% | LR: 0.001000 | Time: 11.22s
Epoch  8/30 | Loss: 0.2482 | Val Acc: 88.98% | LR: 0.001000 | Time: 13.07s
Epoch  9/30 | Loss: 0.2354 | Val Acc: 87.82% | LR: 0.001000 | Time: 12.24s
Epoch 10/30 | Loss: 0.2269 | Val Acc: 89.02% | LR: 0.001000 | Time: 11.89s
Epoch 11/30 | Loss: 0.2167 | Val Acc: 88.28% | LR: 0.001000 | Time: 12.62s
Epoch 12/30 | Loss: 0.2060 | Val Acc: 89.43% | LR: 0.001000 | Time: 12.92s
Epoch 13/30 | Loss: 0.1980 | Val Acc: 88.97% | LR: 0.001000 | Time: 12.26s
Epoch 14/30 | Loss: 0.1884 | Val Acc: 88.52% | LR: 0.001000 | Time: 12.99s
Epoch 15/30 | Loss: 0.1820 | Val Acc: 89.20% | LR: 0.001000 | Time: 11.31s
Epoch 16/30 | Loss: 0.1719 | Val Acc: 89.07% | LR: 0.001000 | Time: 11.11s
Epoch 17/30 | Loss: 0.1656 | Val Acc: 89.60% | LR: 0.001000 | Time: 12.06s
Epoch 18/30 | Loss: 0.1600 | Val Acc: 88.57% | LR: 0.001000 | Time: 12.37s
Epoch 19/30 | Loss: 0.1495 | Val Acc: 89.15% | LR: 0.001000 | Time: 12.33s
Epoch 20/30 | Loss: 0.1452 | Val Acc: 88.72% | LR: 0.001000 | Time: 9.76s
Epoch 21/30 | Loss: 0.1423 | Val Acc: 89.30% | LR: 0.001000 | Time: 10.55s
```

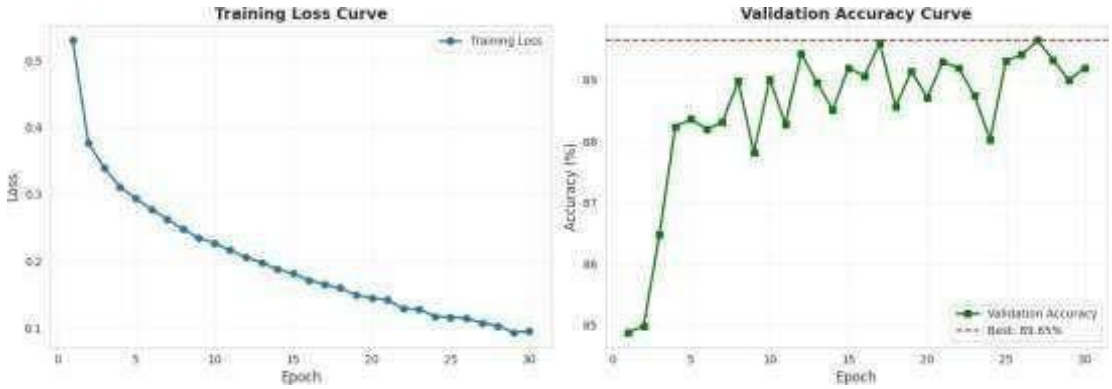
```
Epoch 22/30 | Loss: 0.1299 | Val Acc: 89.20% | LR: 0.001000 | Time: 11.64s
Epoch 23/30 | Loss: 0.1287 | Val Acc: 88.75% | LR: 0.001000 | Time: 11.92s
Epoch 24/30 | Loss: 0.1179 | Val Acc: 88.03% | LR: 0.001000 | Time: 12.97s
Epoch 25/30 | Loss: 0.1162 | Val Acc: 89.32% | LR: 0.001000 | Time: 12.71s
Epoch 26/30 | Loss: 0.1151 | Val Acc: 89.42% | LR: 0.001000 | Time: 12.94s
Epoch 27/30 | Loss: 0.1079 | Val Acc: 89.65% | LR: 0.001000 | Time: 11.22s
Epoch 28/30 | Loss: 0.1033 | Val Acc: 89.33% | LR: 0.001000 | Time: 12.33s
Epoch 29/30 | Loss: 0.0939 | Val Acc: 89.00% | LR: 0.001000 | Time: 9.91s
Epoch 30/30 | Loss: 0.0956 | Val Acc: 89.20% | LR: 0.001000 | Time: 11.84s

Training completed!
Best validation accuracy: 89.65% at epoch 27
```

1.6 Visualization: Training Dynamics

These plots help us understand **how** and **why** training succeeded or failed.

[10]:
```
# Plot 1: Loss vs Epoch
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Training Loss
axes[0].plot(training_log['epoch'], training_log['train_loss'],
             marker='o', linewidth=2, label='Training Loss')
axes[0].set_xlabel('Epoch', fontsize=12)
axes[0].set_ylabel('Loss', fontsize=12)
axes[0].set_title('Training Loss Curve', fontsize=14, fontweight='bold')
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Validation Accuracy
axes[1].plot(training_log['epoch'], training_log['val_accuracy'],
             marker='s', linewidth=2, color='green', label='Validation␣
  ↪Accuracy')
axes[1].axhline(y=best_val_acc, color='r', linestyle='--',
                label=f'Best: {best_val_acc:.2f}%')
axes[1].set_xlabel('Epoch', fontsize=12)
axes[1].set_ylabel('Accuracy (%)', fontsize=12)
axes[1].set_title('Validation Accuracy Curve', fontsize=14, fontweight='bold')
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig(f'results_{EXPERIMENT_ID}_loss_acc.png', dpi=150,␣
  ↪bbox_inches='tight')
```

```
plt.show()

print(" Saved: Loss and Accuracy plots")
```



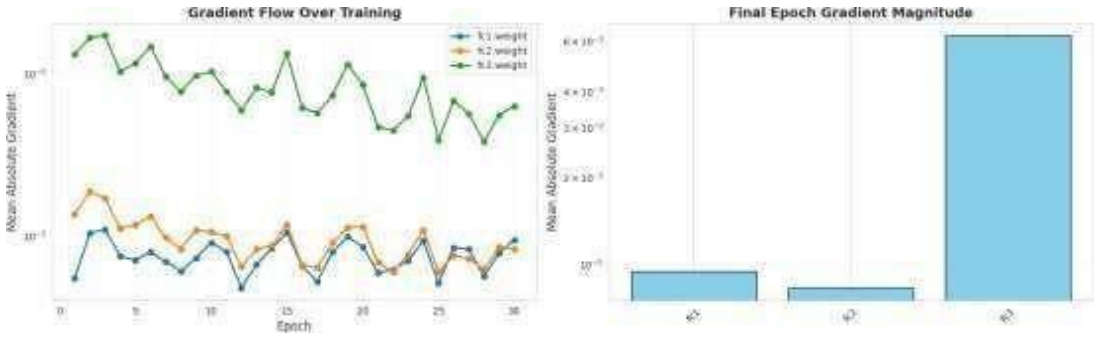
Saved: Loss and Accuracy plots

[11]:
```
# Plot 2: Learning Rate Schedule (if scheduler is used)
if SCHEDULER != 'none':
    plt.figure(figsize=(10, 5))
    plt.plot(training_log['epoch'], training_log['learning_rate'],
             marker='o', linewidth=2, color='purple')
    plt.xlabel('Epoch', fontsize=12)
    plt.ylabel('Learning Rate', fontsize=12)
    plt.title(f'Learning Rate Schedule ({SCHEDULER})', fontsize=14,
 fontweight='bold')
    plt.grid(True, alpha=0.3)
    plt.yscale('log')
    plt.tight_layout()
    plt.savefig(f'results_{EXPERIMENT_ID}_lr_schedule.png', dpi=150,
 bbox_inches='tight')
    plt.show()
    print(" Saved: Learning Rate schedule plot")
else:
    print("No scheduler used - skipping LR plot")
```

No scheduler used - skipping LR plot

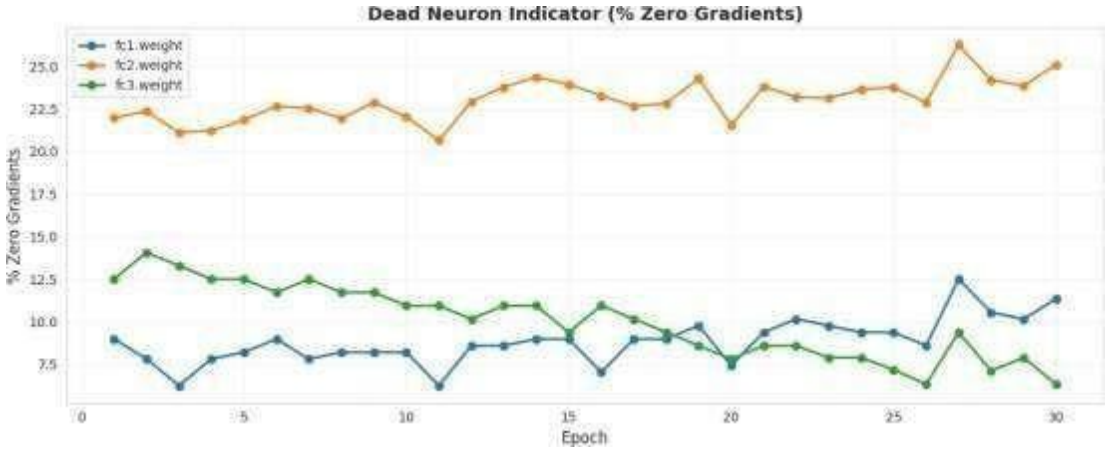
[12]:
```
# Plot 3: Gradient Flow Analysis (Most Important!)
fig, axes = plt.subplots(1, 2, figsize=(16, 5))

# Gradient magnitude over epochs
for layer_name in training_log['gradient_stats'].keys():
    if 'weight' in layer_name: # Only plot weight gradients
        mean_abs_grads = training_log['gradient_stats'][layer_name]['mean_abs']
```

```
            axes[0].plot(training_log['epoch'], mean_abs_grads,
                         marker='o', linewidth=2, label=layer_name)

axes[0].set_xlabel('Epoch', fontsize=12)
axes[0].set_ylabel('Mean Absolute Gradient', fontsize=12)
axes[0].set_title('Gradient Flow Over Training', fontsize=14, fontweight='bold')
axes[0].legend()
axes[0].set_yscale('log')
axes[0].grid(True, alpha=0.3)

# Gradient magnitude at final epoch (bar plot)
layer_names = []
final_grads = []
for layer_name in training_log['gradient_stats'].keys():
    if 'weight' in layer_name:
        layer_names.append(layer_name.replace('.weight', ''))
        final_grads.
    append(training_log['gradient_stats'][layer_name]['mean_abs'][-1])

axes[1].bar(range(len(layer_names)), final_grads, color='skyblue',
    edgecolor='black')
axes[1].set_xticks(range(len(layer_names)))
axes[1].set_xticklabels(layer_names, rotation=45)
axes[1].set_ylabel('Mean Absolute Gradient', fontsize=12)
axes[1].set_title(f'Final Epoch Gradient Magnitude', fontsize=14,
    fontweight='bold')
axes[1].set_yscale('log')
axes[1].grid(True, alpha=0.3, axis='y')

plt.tight_layout()
plt.savefig(f'results_{EXPERIMENT_ID}_gradients.png', dpi=150,
    bbox_inches='tight')
plt.show()

print(" Saved: Gradient flow plots")
```



Saved: Gradient flow plots

[13]:
```
# Plot 4: Dead Neuron Analysis (for ReLU-family activations)
if ACTIVATION in ['relu', 'leaky_relu', 'elu']:
    plt.figure(figsize=(12, 5))

    for layer_name in training_log['gradient_stats'].keys():
        if 'weight' in layer_name:
            pct_zero = training_log['gradient_stats'][layer_name]['pct_zero']
            plt.plot(training_log['epoch'], pct_zero,
                     marker='o', linewidth=2, label=layer_name)

    plt.xlabel('Epoch', fontsize=12)
    plt.ylabel('% Zero Gradients', fontsize=12)
    plt.title('Dead Neuron Indicator (% Zero Gradients)', fontsize=14,␣
 ↪fontweight='bold')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.savefig(f'results_{EXPERIMENT_ID}_dead_neurons.png', dpi=150,␣
 ↪bbox_inches='tight')
    plt.show()
    print(" Saved: Dead neuron analysis plot")
else:
    print(f"Dead neuron analysis not applicable for {ACTIVATION}")
```



Saved: Dead neuron analysis plot

1.7 Summary Statistics

Compact metrics for comparison across runs.

[14]:
```
# Compute summary statistics
final_loss = training_log['train_loss'][-1]
best_val_acc = max(training_log['val_accuracy'])
epoch_to_90pct = None
target_acc = 0.9 * best_val_acc
for i, acc in enumerate(training_log['val_accuracy']):
    if acc >= target_acc:
        epoch_to_90pct = i + 1
        break

# Mean gradient in early layers (indicator of vanishing gradients)
early_layer_grads = []
for layer_name in ['fc1.weight', 'fc2.weight']:
    if layer_name in training_log['gradient_stats']:
        early_layer_grads.append(np.
 smean(training_log['gradient_stats'][layer_name]['mean_abs']))
mean_early_grad = np.mean(early_layer_grads) if early_layer_grads else 0

# Training stability (inverse of loss variance)
loss_variance = np.var(training_log['train_loss'])
stability_score = 1 / (1 + loss_variance)  # Higher is better

# Average training time
avg_epoch_time =    np.mean(training_log['epoch_time'])

summary = {
    'experiment_id': EXPERIMENT_ID,
    'activation': ACTIVATION,
    'optimizer': OPTIMIZER,
    'scheduler': SCHEDULER,
    'learning_rate': LEARNING_RATE,
    'final_loss': final_loss,
    'best_val_accuracy': best_val_acc,
    'epoch_to_90pct_best': epoch_to_90pct,
    'mean_gradient_early_layers': mean_early_grad,
    'training_stability_score': stability_score,
    'avg_epoch_time': avg_epoch_time,
}

# Display summary
print("\n" + "="*60)
print("EXPERIMENT SUMMARY")
print("="*60)
print(f"Experiment ID:                {summary['experiment_id']}")
print(f"Configuration:                {ACTIVATION} + {OPTIMIZER} + {SCHEDULER}␣
 s(LR={LEARNING_RATE})")
print("-"*60)
```

```
print(f"Final Loss:                {summary['final_loss']:.4f}")
print(f"Best Val Accuracy:         {summary['best_val_accuracy']:.2f}%")
print(f"Epoch to 90% of Best:      {summary['epoch_to_90pct_best'] if
  ↪epoch_to_90pct else 'N/A'}")
print(f"Mean Early Layer Gradient: {summary['mean_gradient_early_layers']:.
  ↪6f}")
print(f"Training Stability Score:  {summary['training_stability_score']:.4f}")
print(f"Avg Epoch Time:            {summary['avg_epoch_time']:.2f}s")
print("="*60 + "\n")

# Save summary to JSON
with open(f'summary_{EXPERIMENT_ID}.json', 'w') as f:
    json.dump(summary, f, indent=2)

print(f"  Saved: summary_{EXPERIMENT_ID}.json")
```

```
============================================================
EXPERIMENT SUMMARY
============================================================
Experiment ID:              relu_adam_0e001
Configuration:              relu + adam + none (LR=0.001)
------------------------------------------------------------
Final Loss:                 0.0956
Best Val Accuracy:          89.65%
Epoch to 90% of Best:       1
Mean Early Layer Gradient:  0.000858
Training Stability Score:   0.9908
Avg Epoch Time:             11.81s
============================================================

  Saved: summary_relu_adam_0e001.json
```

[15] :
```
# Save complete training log
# Convert gradient_stats to regular dict for JSON serialization
training_log_serializable = {
    'epoch': training_log['epoch'],
    'train_loss': training_log['train_loss'],
    'val_accuracy': training_log['val_accuracy'],
    'learning_rate': training_log['learning_rate'],
    'epoch_time': training_log['epoch_time'],
    'gradient_stats': {k: dict(v) for k, v in training_log['gradient_stats'].
  ↪items()},
    'config': {
        'activation': ACTIVATION,
        'optimizer':  OPTIMIZER,
        'scheduler': SCHEDULER,
```

```
            'learning_rate': LEARNING_RATE,
            'epochs': EPOCHS,
            'batch_size': BATCH_SIZE,
            'seed': SEED,
        }
}

with open(f'training_log_{EXPERIMENT_ID}.json', 'w') as f:
    json.dump(training_log_serializable, f, indent=2)

print(f"  Saved: training_log_{EXPERIMENT_ID}.json")
```

```
 Saved: training_log_relu_adam_0e001.json
```
