Executive Summary

Modern Optimizers: Beyond Adam for Deep Learning

Context

The traditional approach of using Adam/AdamW with a single learning rate is becoming outdated. Modern deep learning training leverages:
- **Matrix-oriented optimizers** (Muon, Shampoo, SOAP) that capture parameter correlations
- **Memory-efficient alternatives** (Lion, Adafactor) that scale to billion-parameter models
- **muP principles** for consistent training dynamics across model scales

This Notebook

We implement and compare two modern optimizers against the AdamW baseline:

Optimizer	Type	Key Innovation	Memory	Best For
AdamW	Baseline	Diagonal preconditioning	2× params	General use
Shampoo	Matrix-oriented	Kronecker-factored full-matrix preconditioning	$O(m^2 + n^2)$	Dense layers, unlimited compute
Adafactor	Memory-efficient	Factorized second moments	~0.5× params	Very large models (T5, etc.)

What You'll Do

1. **Implement Shampoo** - Matrix power computation and Kronecker preconditioning
2. **Implement Adafactor** - Factorized second moment estimation
3. **Compare optimizers** - Training dynamics, speed, and final performance on VAE
4. **Hyperparameter exploration** - Find optimal settings for each optimizer

5. **Answer analysis questions** - Memory, compute, convergence trade-offs

Key Takeaways (Preview)

- Shampoo can converge faster in epochs but is computationally expensive
- Adafactor uses ~4× less memory than Adam for large weight matrices
- Hyperparameter tuning is crucial—defaults rarely optimal
- Choice depends on your constraints: compute budget, memory, model size

Extended VAE Problem: Modern Optimizers

Beyond Adam: Exploring Shampoo and Adafactor

As deep learning training best practices evolve, the traditional approach of using Adam(W) with a single learning rate is being replaced by more sophisticated approaches:

- **Matrix-oriented optimizers** like Muon and Shampoo that capture parameter correlations
- **Memory-efficient alternatives** like Adafactor that scale to massive models
- **muP (maximal update parameterization)** principles for consistent training dynamics

In this extended problem, you will:
1. Implement **Shampoo** - a matrix preconditioning optimizer using Kronecker factors
2. Implement **Adafactor** - a memory-efficient factorized Adam alternative
3. Compare training dynamics of AdamW vs Shampoo vs Adafactor on VAE training
4. Explore hyperparameter choices for each optimizer

Prerequisites: Complete the main VAE implementation (parts a-d) before starting this section.

In [ ]:
```
# Install dependencies
!pip install --no-deps --upgrade IPython
!pip install matplotlib numpy torch torchvision tqdm
```

In [ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np
from tqdm import tqdm
import time
from typing import Optional, Tuple, List, Dict
import math

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

Dataset and VAE Model Setup

We'll use MNIST with a simple VAE architecture for comparing optimizers.

In [ ]:

```
# Data loading
transform = transforms.Compose([
    transforms.ToTensor(),
])

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

batch_size = 128
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True,
num_workers=2)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False,
num_workers=2)
```

In [ ]:

```
class VAE(nn.Module):
    """Simple VAE for MNIST."""
    def __init__(self, input_dim=784, hidden_dim=400, latent_dim=20):
        super().__init__()
        self.input_dim = input_dim
        self.latent_dim = latent_dim

        # Encoder
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc_mu = nn.Linear(hidden_dim, latent_dim)
        self.fc_logvar = nn.Linear(hidden_dim, latent_dim)

        # Decoder
        self.fc3 = nn.Linear(latent_dim, hidden_dim)
        self.fc4 = nn.Linear(hidden_dim, input_dim)
    def encode(self, x):
        h = F.relu(self.fc1(x))
        return self.fc_mu(h), self.fc_logvar(h)

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def decode(self, z):
        h = F.relu(self.fc3(z))
        return torch.sigmoid(self.fc4(h))

    def forward(self, x):
        x = x.view(-1, self.input_dim)
        mu, logvar = self.encode(x)
        z = self.reparameterize(mu, logvar)
        recon = self.decode(z)
        return recon, mu, logvar

def vae_loss(recon_x, x, mu, logvar):
    """VAE loss = Reconstruction + KL divergence."""
    x = x.view(-1, 784)
    BCE = F.binary_cross_entropy(recon_x, x, reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + KLD
```

Part 1: Implementing Shampoo

Background

Shampoo (Gupta et al., 2018) is a matrix preconditioning optimizer that approximates full-matrix AdaGrad using Kronecker-factored statistics.

For a weight matrix $W \in \mathbb{R}^{m \times n}$ with gradient $G$:

1. **Accumulate Kronecker factors:**
$$L \leftarrow \beta \cdot L + (1-\beta) \cdot G G^T \quad \text{(left factor, } m \times m \text{)}$$
$$R \leftarrow \beta \cdot R + (1-\beta) \cdot G^T G \quad \text{(right factor, } n \times n \text{)}$$

2. **Compute preconditioned update:**
$$\text{update} = L^{-1/4} \cdot G \cdot R^{-1/4}$$

3. **Apply update:**
$$W \leftarrow W - \eta \cdot \text{update}$$

The key insight is that $L^{-1/4}$ and $R^{-1/4}$ "whiten" the gradients, leading to better conditioning.

Problem 1.1: Implement Matrix Power via Eigendecomposition

Complete the `matrix_power` function that computes $M^p$ for symmetric positive semi-definite matrices using eigendecomposition.

In [ ]:
```
def matrix_power(M: torch.Tensor, p: float, epsilon: float = 1e-6) ->
torch.Tensor:
    """
    Compute M^p for symmetric positive semi-definite matrix M.

    Uses eigendecomposition: M = V @ diag(eigenvalues) @ V^T
    So M^p = V @ diag(eigenvalues^p) @ V^T

    Args:
        M: Symmetric PSD matrix of shape (n, n)
        p: Power to raise the matrix to (e.g., -0.25 for inverse fourth root)
        epsilon: Small constant for numerical stability

    Returns:
        M^p: Matrix of shape (n, n)
    """

##############################################################################
    # TODO: Implement matrix power using eigendecomposition
#
    # 1. Compute eigenvalues and eigenvectors of M using torch.linalg.eigh
#
    # 2. Clamp eigenvalues to be >= epsilon for numerical stability
#
    # 3. Raise eigenvalues to power p
#
    # 4. Reconstruct matrix: V @ diag(eigenvalues^p) @ V^T
#

##############################################################################
```

```
################################################################################
    #                              END OF YOUR CODE                             
#
################################################################################
    return result
```

In [ ]:
```
# Test matrix_power
torch.manual_seed(42)
A = torch.randn(4, 4)
M = A @ A.T + 0.1 * torch.eye(4)  # Symmetric PSD

M_inv_sqrt = matrix_power(M, -0.5)
identity_approx = M_inv_sqrt @ M @ M_inv_sqrt

print("M^(-0.5) @ M @ M^(-0.5) should be close to identity:")
print(identity_approx)
print(f"Max error from identity: {(identity_approx -
torch.eye(4)).abs().max():.6f}")
assert (identity_approx - torch.eye(4)).abs().max() < 1e-4, "matrix_power
implementation is incorrect"
```

Problem 1.2: Implement Shampoo Optimizer

Complete the Shampoo optimizer class. Key implementation details:

- For 1D parameters (biases), fall back to standard SGD with momentum
- For 2D+ parameters, use Kronecker-factored preconditioning
- Update preconditioners every `update_freq` steps (computing matrix powers is expensive)

In [ ]:
```
class Shampoo(torch.optim.Optimizer):
    """
    Shampoo optimizer with Kronecker-factored preconditioning.

    Reference: https://arxiv.org/abs/1802.09568

    Args:
        params: Model parameters
        lr: Learning rate
        momentum: Momentum factor (applied to preconditioned gradients)
        weight_decay: L2 regularization
        epsilon: Numerical stability constant
        update_freq: How often to update preconditioners (every N steps)
    """
    def __init__(self, params, lr=0.1, momentum=0.0, weight_decay=0.0,
                 epsilon=1e-4, update_freq=1):
```

```
        defaults = dict(lr=lr, momentum=momentum, weight_decay=weight_decay,
                        epsilon=epsilon, update_freq=update_freq)
        super().__init__(params, defaults)
        self.step_count = 0

    @torch.no_grad()
    def step(self, closure=None):
        loss = None
        if closure is not None:
            with torch.enable_grad():
                loss = closure()

        self.step_count += 1

        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:
                    continue

                grad = p.grad
                state = self.state[p]

                # Weight decay
                if group['weight_decay'] != 0:
                    grad = grad.add(p, alpha=group['weight_decay'])

                # Handle 1D parameters (biases) with simple SGD + momentum
                if p.dim() == 1:
                    if 'momentum_buffer' not in state:
                        state['momentum_buffer'] = torch.zeros_like(p)

                    buf = state['momentum_buffer']
                    buf.mul_(group['momentum']).add_(grad)
                    p.add_(buf, alpha=-group['lr'])
                    continue

                ##############################################################
                # TODO: Implement Shampoo update for 2D parameters           #
                #                                                            #
                # 1. Reshape gradient to 2D if needed (for conv layers)      #
                # 2. Initialize L and R preconditioners if not present       #
                # 3. Update L and R with current gradient:                   #
                #    L = 0.9 * L + 0.1 * (G @ G.T)                           #
                #    R = 0.9 * R + 0.1 * (G.T @ G)                           #
                # 4. Every update_freq steps, compute:                       #
                #    L_inv = matrix_power(L, -0.25, epsilon)                 #
                #    R_inv = matrix_power(R, -0.25, epsilon)                 #
                # 5. Compute preconditioned gradient:                        #
                #    precond_grad = L_inv @ G @ R_inv                        #
                # 6. Apply momentum if specified                             #
                # 7. Update parameters                                       #
                ##############################################################
                # Reshape to 2D for consistent handling
                original_shape = p.shape
                if p.dim() > 2:
                    grad_2d = grad.view(grad.size(0), -1)
                else:
                    grad_2d = grad

                m, n = grad_2d.shape

                # Initialize state
                if 'L' not in state:
                    state['L'] = group['epsilon'] * torch.eye(m,
device=p.device, dtype=p.dtype)
                    state['R'] = group['epsilon'] * torch.eye(n,
device=p.device, dtype=p.dtype)
                    state['L_inv'] = torch.eye(m, device=p.device,
dtype=p.dtype)
                    state['R_inv'] = torch.eye(n, device=p.device,
dtype=p.dtype)
                    if group['momentum'] > 0:
                        state['momentum_buffer'] = torch.zeros_like(grad_2d)

                ###############################################################
                #                    YOUR CODE HERE                           #
                ###############################################################

                ###############################################################
                #                    END OF YOUR CODE                         #
                ###############################################################

        return loss
```

Solution for Problem 1.2

In [ ]:

```
# SOLUTION

def matrix_power_solution(M: torch.Tensor, p: float, epsilon: float = 1e-6) ->
torch.Tensor:
    """Solution for matrix power."""
    eigenvalues, eigenvectors = torch.linalg.eigh(M)
    eigenvalues = torch.clamp(eigenvalues, min=epsilon)
    eigenvalues_p = eigenvalues ** p
    result = eigenvectors @ torch.diag(eigenvalues_p) @ eigenvectors.T
    return result


class ShampooSolution(torch.optim.Optimizer):
    """Complete Shampoo implementation."""

    def __init__(self, params, lr=0.1, momentum=0.0, weight_decay=0.0,
                 epsilon=1e-4, update_freq=1, beta=0.9):
        defaults = dict(lr=lr, momentum=momentum, weight_decay=weight_decay,
                       epsilon=epsilon, update_freq=update_freq, beta=beta)
        super().__init__(params, defaults)
        self.step_count = 0

    @torch.no_grad()
    def step(self, closure=None):
        loss = None
        if closure is not None:
            with torch.enable_grad():
                loss = closure()

        self.step_count += 1

        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:
                    continue

                grad = p.grad
                state = self.state[p]

                # Weight decay
                if group['weight_decay'] != 0:
                    grad = grad.add(p, alpha=group['weight_decay'])

                # Handle 1D parameters with simple SGD + momentum
                if p.dim() == 1:
                    if 'momentum_buffer' not in state:
                        state['momentum_buffer'] = torch.zeros_like(p)
                    buf = state['momentum_buffer']
                    buf.mul_(group['momentum']).add_(grad)
                    p.add_(buf, alpha=-group['lr'])
                    continue

                # Reshape to 2D
                original_shape = p.shape
                if p.dim() > 2:
                    grad_2d = grad.view(grad.size(0), -1)
                    p_2d = p.view(p.size(0), -1)
                else:
                    grad_2d = grad
                    p_2d = p

                m, n = grad_2d.shape

                # Initialize state
                if 'L' not in state:
                    state['L'] = group['epsilon'] * torch.eye(m,
device=p.device, dtype=p.dtype)
                    state['R'] = group['epsilon'] * torch.eye(n,
device=p.device, dtype=p.dtype)
```

```
                    state['L_inv'] = torch.eye(m, device=p.device,
dtype=p.dtype)
                    state['R_inv'] = torch.eye(n, device=p.device,
dtype=p.dtype)
                    if group['momentum'] > 0:
                        state['momentum_buffer'] = torch.zeros_like(grad_2d)

                # Update preconditioner statistics
                beta = group['beta']
                state['L'] = beta * state['L'] + (1 - beta) * (grad_2d @
grad_2d.T)
                state['R'] = beta * state['R'] + (1 - beta) * (grad_2d.T @
grad_2d)

                # Recompute inverse fourth roots periodically
                if self.step_count % group['update_freq'] == 0:
                    state['L_inv'] = matrix_power_solution(state['L'], -0.25,
group['epsilon'])
                    state['R_inv'] = matrix_power_solution(state['R'], -0.25,
group['epsilon'])

                # Compute preconditioned gradient
                precond_grad = state['L_inv'] @ grad_2d @ state['R_inv']

                # Apply momentum
                if group['momentum'] > 0:

state['momentum_buffer'].mul_(group['momentum']).add_(precond_grad)
                    update = state['momentum_buffer']
                else:
                    update = precond_grad

                # Update parameters
                if p.dim() > 2:
                    p.add_(update.view(original_shape), alpha=-group['lr'])
                else:
                    p.add_(update, alpha=-group['lr'])

        return loss
```

Part 2: Implementing Adafactor

Background

Adafactor (Shazeer & Stern, 2018) is a memory-efficient optimizer designed for training large models. It was used to train T5 and other large language models.

Key ideas:
1. **Factorized second moments**: Instead of storing full $m \times n$ second moment matrix, store row and column statistics ($O(m+n)$ vs $O(mn)$)
2. **No momentum** by default (saves memory)
3. **Relative step sizing**: Automatically scale learning rate based on parameter RMS

For a weight matrix $W \in \mathbb{R}^{m \times n}$ with gradient G:

$$R_t = \rho \cdot R_{t-1} + (1-\rho) \cdot \text{mean}(G^2, \text{dim}=1) \quad \text{(row statistics)}$$
$$C_t = \rho \cdot C_{t-1} + (1-\rho) \cdot \text{mean}(G^2, \text{dim}=0) \quad \text{(column statistics)}$$

$$\hat{V}_t = \frac{R_t \otimes C_t}{\text{mean}(R_t)} \quad \text{(reconstructed second moment)}$$

$$\text{update} = \frac{G}{\sqrt{\hat{V}_t} + \epsilon}$$

Problem 2.1: Implement Adafactor

In [ ]:
```
class Adafactor(torch.optim.Optimizer):
    """
    Adafactor optimizer with factorized second moment estimation.

    Reference: https://arxiv.org/abs/1804.04235

    Args:
        params: Model parameters
        lr: Learning rate (if None, uses relative step sizing)
        beta2_decay: Decay rate for second moment (-0.8 means rho = 1 -
t^(-0.8))
        epsilon1: Regularization constant for second moment
        epsilon2: Regularization constant for RMS
        clip_threshold: Gradient clipping threshold
        weight_decay: L2 regularization
        scale_parameter: Whether to use relative step sizing
    """
    def __init__(self, params, lr=None, beta2_decay=-0.8, epsilon1=1e-30,
                 epsilon2=1e-3, clip_threshold=1.0, weight_decay=0.0,
                 scale_parameter=True):
```

```
        if lr is None and not scale_parameter:
            raise ValueError("lr must be specified if scale_parameter is
False")

        defaults = dict(lr=lr, beta2_decay=beta2_decay, epsilon1=epsilon1,
                       epsilon2=epsilon2, clip_threshold=clip_threshold,
                       weight_decay=weight_decay,
scale_parameter=scale_parameter)
        super().__init__(params, defaults)

    def _get_rho(self, step, beta2_decay):
        """Compute adaptive decay rate: rho = 1 - step^(beta2_decay)"""
        return min(1.0 - math.pow(step, beta2_decay), 0.999)

    def _rms(self, tensor):
        """Root mean square of tensor."""
        return tensor.norm() / (tensor.numel() ** 0.5)

    @torch.no_grad()
    def step(self, closure=None):
        loss = None
        if closure is not None:
            with torch.enable_grad():
                loss = closure()

        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:
                    continue

                grad = p.grad
                state = self.state[p]

                # State initialization
                if len(state) == 0:
                    state['step'] = 0

                    # For 2D+ parameters, use factorized representation
                    if p.dim() >= 2:
                        # Row and column factors
                        state['exp_avg_sq_row'] = torch.zeros(p.shape[:-1],
device=p.device, dtype=p.dtype)
                        state['exp_avg_sq_col'] = torch.zeros(p.shape[:-2] +
p.shape[-1:], device=p.device, dtype=p.dtype)
                    else:
                        # For 1D, use full second moment (like Adam)
                        state['exp_avg_sq'] = torch.zeros_like(p)

                state['step'] += 1
                step = state['step']

                # Weight decay
                if group['weight_decay'] != 0:
                    grad = grad.add(p, alpha=group['weight_decay'])

                rho = self._get_rho(step, group['beta2_decay'])

                ##############################################################
                # TODO: Implement Adafactor update                           #
                #                                                            #
                # For 2D+ parameters:                                        #
                # 1. Compute row mean of grad^2: mean(grad^2, dim=-1)        #
                # 2. Compute col mean of grad^2: mean(grad^2, dim=-2)        #
                # 3. Update exp_avg_sq_row and exp_avg_sq_col with rho       #
                # 4. Reconstruct variance: row[:, None] * col[None, :] /     #
                #                          mean(row)                         #
                # 5. Compute update: grad / (sqrt(variance) + epsilon1)      #
                #                                                            #
                # For 1D parameters:                                         #
                # - Use standard Adam-style second moment                    #
                #                                                            #
                # Apply gradient clipping and relative step sizing as needed #
                ##############################################################

                ##############################################################
                #                    END OF YOUR CODE                        #
                ##############################################################

        return loss
```

Solution for Problem 2.1

In [ ]:
```
# SOLUTION

class AdafactorSolution(torch.optim.Optimizer):
    """Complete Adafactor implementation."""

    def __init__(self, params, lr=None, beta2_decay=-0.8, epsilon1=1e-30,
                 epsilon2=1e-3, clip_threshold=1.0, weight_decay=0.0,
                 scale_parameter=True):

        if lr is None and not scale_parameter:
            raise ValueError("lr must be specified if scale_parameter is
False")

        defaults = dict(lr=lr, beta2_decay=beta2_decay, epsilon1=epsilon1,
                       epsilon2=epsilon2, clip_threshold=clip_threshold,
                       weight_decay=weight_decay,
scale_parameter=scale_parameter)
        super().__init__(params, defaults)

    def _get_rho(self, step, beta2_decay):
        return min(1.0 - math.pow(step, beta2_decay), 0.999)
    def _rms(self, tensor):
        return tensor.norm() / (tensor.numel() ** 0.5)

    def _approx_sq_grad(self, exp_avg_sq_row, exp_avg_sq_col):
        """Reconstruct factorized second moment approximation."""
        r_factor = (exp_avg_sq_row / exp_avg_sq_row.mean(dim=-1,
keepdim=True)).unsqueeze(-1)
        c_factor = exp_avg_sq_col.unsqueeze(-2)
        return r_factor * c_factor

    @torch.no_grad()
    def step(self, closure=None):
        loss = None
        if closure is not None:
            with torch.enable_grad():
                loss = closure()

        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:
                    continue

                grad = p.grad
                state = self.state[p]
                grad_shape = grad.shape
                factored = len(grad_shape) >= 2

                # State initialization
                if len(state) == 0:
                    state['step'] = 0

                    if factored:
                        state['exp_avg_sq_row'] = torch.zeros(grad_shape[:-1],
device=p.device, dtype=p.dtype)
                        state['exp_avg_sq_col'] = torch.zeros(grad_shape[:-2]
+ grad_shape[-1:], device=p.device, dtype=p.dtype)
                    else:
                        state['exp_avg_sq'] = torch.zeros_like(grad)

                    state['RMS'] = 0.0

                state['step'] += 1
                step = state['step']

                # Weight decay (decoupled)
                if group['weight_decay'] != 0:
                    p.mul_(1 - group['lr'] * group['weight_decay'] if
group['lr'] else 1 - group['weight_decay'])

                rho = self._get_rho(step, group['beta2_decay'])

                # Update second moment estimate
                if factored:
                    # Factorized update
                    grad_sqr = grad.pow(2)
                    row_mean = grad_sqr.mean(dim=-1)
                    col_mean = grad_sqr.mean(dim=-2)

                    state['exp_avg_sq_row'].mul_(rho).add_(row_mean, alpha=1 -
rho)
                    state['exp_avg_sq_col'].mul_(rho).add_(col_mean, alpha=1 -
rho)

                    # Reconstruct variance estimate
                    variance = self._approx_sq_grad(state['exp_avg_sq_row'],
state['exp_avg_sq_col'])
                    variance = variance.clamp(min=group['epsilon1'])
                else:
                    # Standard (non-factorized) update for 1D params
                    state['exp_avg_sq'].mul_(rho).addcmul_(grad, grad, value=1
- rho)
                    variance =
state['exp_avg_sq'].clamp(min=group['epsilon1'])

                # Compute update
                update = grad / variance.sqrt()

                # Gradient clipping
                update_rms = self._rms(update)
                if update_rms > group['clip_threshold']:
                    update.mul_(group['clip_threshold'] / update_rms)

                # Compute learning rate
                if group['lr'] is not None:
                    lr = group['lr']
                else:
                    # Relative step sizing
                    lr = max(group['epsilon2'], self._rms(p))

                if group['scale_parameter'] and group['lr'] is None:
                    # Scale by 1/sqrt(step) for warmup
                    lr = lr * max(1e-3, 1.0 / math.sqrt(step))

                p.add_(update, alpha=-lr)

        return loss
```

Part 3: Optimizer Comparison Experiments

Now we'll compare AdamW, Shampoo, and Adafactor on VAE training.

Problem 3.1: Training Loop and Comparison

In [ ]:
```
def train_vae(model, optimizer, train_loader, epochs, device):
    """Train VAE and return training history."""
    model.train()
    history = {'loss': [], 'time_per_epoch': []}

    for epoch in range(epochs):
        epoch_loss = 0
        start_time = time.time()

        for batch_idx, (data, _) in enumerate(train_loader):
            data = data.to(device)
            optimizer.zero_grad()

            recon_batch, mu, logvar = model(data)
            loss = vae_loss(recon_batch, data, mu, logvar)

            loss.backward()
            optimizer.step()

            epoch_loss += loss.item()

        epoch_time = time.time() - start_time
        avg_loss = epoch_loss / len(train_loader.dataset)
        history['loss'].append(avg_loss)
        history['time_per_epoch'].append(epoch_time)

        if (epoch + 1) % 5 == 0:
            print(f"Epoch {epoch+1}/{epochs}, Loss: {avg_loss:.4f}, Time:
{epoch_time:.2f}s")

    return history


def evaluate_vae(model, test_loader, device):
    """Evaluate VAE on test set."""
    model.eval()
    test_loss = 0

    with torch.no_grad():
        for data, _ in test_loader:
            data = data.to(device)
            recon, mu, logvar = model(data)
            test_loss += vae_loss(recon, data, mu, logvar).item()
    return test_loss / len(test_loader.dataset)
```

In [ ]:

```
# Run comparison experiment
torch.manual_seed(42)
epochs = 20

results = {}

# AdamW baseline
print("=" * 50)
print("Training with AdamW")
print("=" * 50)
model_adamw = VAE().to(device)
optimizer_adamw = torch.optim.AdamW(model_adamw.parameters(), lr=1e-3,
weight_decay=1e-4)
results['AdamW'] = train_vae(model_adamw, optimizer_adamw, train_loader,
epochs, device)
results['AdamW']['test_loss'] = evaluate_vae(model_adamw, test_loader, device)
print(f"Final test loss: {results['AdamW']['test_loss']:.4f}\n")

# Shampoo
print("=" * 50)
print("Training with Shampoo")
print("=" * 50)
torch.manual_seed(42)
model_shampoo = VAE().to(device)
optimizer_shampoo = ShampooSolution(model_shampoo.parameters(), lr=0.1,
momentum=0.9,
                                     weight_decay=1e-4, update_freq=10)
results['Shampoo'] = train_vae(model_shampoo, optimizer_shampoo, train_loader,
epochs, device)
results['Shampoo']['test_loss'] = evaluate_vae(model_shampoo, test_loader,
device)
print(f"Final test loss: {results['Shampoo']['test_loss']:.4f}\n")

# Adafactor
print("=" * 50)
print("Training with Adafactor")
print("=" * 50)
torch.manual_seed(42)
model_adafactor = VAE().to(device)
optimizer_adafactor = AdafactorSolution(model_adafactor.parameters(), lr=1e-2,
                                         weight_decay=1e-4,
scale_parameter=False)
results['Adafactor'] = train_vae(model_adafactor, optimizer_adafactor,
train_loader, epochs, device)
results['Adafactor']['test_loss'] = evaluate_vae(model_adafactor, test_loader,
device)
print(f"Final test loss: {results['Adafactor']['test_loss']:.4f}\n")
```

In [ ]:
```
# Plot comparison
fig, axes = plt.subplots(1, 3, figsize=(15, 4))

# Loss curves
ax = axes[0]
for name, data in results.items():
    ax.plot(data['loss'], label=name, linewidth=2)
ax.set_xlabel('Epoch')
ax.set_ylabel('Training Loss')
ax.set_title('Training Loss Comparison')
ax.legend()
ax.grid(True, alpha=0.3)

# Time per epoch
ax = axes[1]
names = list(results.keys())
avg_times = [np.mean(results[n]['time_per_epoch']) for n in names]
ax.bar(names, avg_times, color=['tab:blue', 'tab:orange', 'tab:green'])
ax.set_ylabel('Average Time per Epoch (s)')
ax.set_title('Computational Cost')

# Final test loss
ax = axes[2]
test_losses = [results[n]['test_loss'] for n in names]
ax.bar(names, test_losses, color=['tab:blue', 'tab:orange', 'tab:green'])
ax.set_ylabel('Test Loss')
ax.set_title('Final Test Performance')

plt.tight_layout()
plt.savefig('optimizer_comparison.png', dpi=150)
plt.show()
```

Part 4: Hyperparameter Exploration

Problem 4.1: Shampoo Hyperparameter Study

Explore the effect of key Shampoo hyperparameters:
- `update_freq` : How often to recompute preconditioners (1, 10, 50, 100)
- `lr` : Learning rate (0.01, 0.05, 0.1, 0.5)

In [ ]:
```
# Hyperparameter search for Shampoo
shampoo_results = {}

update_freqs = [1, 10, 50]
learning_rates = [0.01, 0.05, 0.1]

for update_freq in update_freqs:
    for lr in learning_rates:
        name = f"freq={update_freq}, lr={lr}"
        print(f"Training Shampoo with {name}")

        torch.manual_seed(42)
        model = VAE().to(device)
        optimizer = ShampooSolution(model.parameters(), lr=lr, momentum=0.9,
                                    update_freq=update_freq)

        history = train_vae(model, optimizer, train_loader, epochs=10,
device=device)
        history['test_loss'] = evaluate_vae(model, test_loader, device)
        shampoo_results[name] = history
        print(f"  Final test loss: {history['test_loss']:.4f}")
        print()
```

In [ ]:
```
# Visualize Shampoo hyperparameter study
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

# Loss curves
ax = axes[0]
for name, data in shampoo_results.items():
    ax.plot(data['loss'], label=name, linewidth=1.5)
ax.set_xlabel('Epoch')
ax.set_ylabel('Training Loss')
ax.set_title('Shampoo: Learning Rate & Update Frequency Study')
ax.legend(fontsize=8)
ax.grid(True, alpha=0.3)

# Heatmap of final test loss
ax = axes[1]
test_losses = np.zeros((len(update_freqs), len(learning_rates)))
for i, uf in enumerate(update_freqs):
    for j, lr in enumerate(learning_rates):
        name = f"freq={uf}, lr={lr}"
        test_losses[i, j] = shampoo_results[name]['test_loss']

im = ax.imshow(test_losses, cmap='RdYlGn_r')
ax.set_xticks(range(len(learning_rates)))
ax.set_xticklabels([str(lr) for lr in learning_rates])
ax.set_yticks(range(len(update_freqs)))
ax.set_yticklabels([str(uf) for uf in update_freqs])
ax.set_xlabel('Learning Rate')
ax.set_ylabel('Update Frequency')
ax.set_title('Shampoo: Test Loss Heatmap')
```

```
plt.colorbar(im, ax=ax)

# Annotate heatmap
for i in range(len(update_freqs)):
    for j in range(len(learning_rates)):
        ax.text(j, i, f'{test_losses[i,j]:.1f}', ha='center', va='center',
fontsize=9)

plt.tight_layout()
plt.savefig('shampoo_hyperparams.png', dpi=150)
plt.show()
```

Problem 4.2: Adafactor Hyperparameter Study

Explore Adafactor with different settings:
- With and without relative step sizing
- Different learning rates

In [ ]:

```
# Hyperparameter search for Adafactor
adafactor_results = {}

configs = [
    {'lr': 1e-3, 'scale_parameter': False, 'name': 'lr=1e-3'},
    {'lr': 5e-3, 'scale_parameter': False, 'name': 'lr=5e-3'},
    {'lr': 1e-2, 'scale_parameter': False, 'name': 'lr=1e-2'},
    {'lr': None, 'scale_parameter': True, 'name': 'relative_step'},
]

for config in configs:
    print(f"Training Adafactor with {config['name']}")

    torch.manual_seed(42)
    model = VAE().to(device)
    optimizer = AdafactorSolution(model.parameters(), lr=config['lr'],
                                  scale_parameter=config['scale_parameter'])

    history = train_vae(model, optimizer, train_loader, epochs=10,
device=device)
    history['test_loss'] = evaluate_vae(model, test_loader, device)
    adafactor_results[config['name']] = history
    print(f"  Final test loss: {history['test_loss']:.4f}")
    print()
```

In [ ]:

```
# Visualize Adafactor results
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

ax = axes[0]
```

```
for name, data in adafactor_results.items():
    ax.plot(data['loss'], label=name, linewidth=2)
ax.set_xlabel('Epoch')
ax.set_ylabel('Training Loss')
ax.set_title('Adafactor: Learning Rate Study')
ax.legend()
ax.grid(True, alpha=0.3)

ax = axes[1]
names = list(adafactor_results.keys())
test_losses = [adafactor_results[n]['test_loss'] for n in names]
ax.bar(names, test_losses)
ax.set_ylabel('Test Loss')
ax.set_title('Adafactor: Final Test Performance')
ax.tick_params(axis='x', rotation=45)

plt.tight_layout()
plt.savefig('adafactor_hyperparams.png', dpi=150)
plt.show()
```

Part 5: Analysis Questions

Answer the following questions in your writeup:

Question 1: Memory Efficiency

Compare the memory requirements of AdamW, Shampoo, and Adafactor for a weight matrix of shape (m, n). Which optimizer is most memory-efficient? When might this matter in practice?

Question 2: Computational Trade-offs

Shampoo requires computing matrix inverse fourth roots. Why do we use `update_freq > 1` in practice? How does this affect the trade-off between computational cost and optimization quality?

Question 3: Convergence Properties

Based on your experiments, compare the convergence behavior of the three optimizers. Which converges fastest in terms of epochs? Which is fastest in wall-clock time?

Question 4: Hyperparameter Sensitivity

Based on your hyperparameter exploration, which optimizer seems most sensitive to hyperparameter choices? Which is most robust?

Question 5: Relation to Muon

Explain how Shampoo relates to Muon. Both are "matrix-oriented" optimizers, but they use different techniques. What is the key difference in how they achieve preconditioning?

Summary

In this extended problem, you:

1. **Implemented Shampoo** - a matrix preconditioning optimizer using Kronecker factors
2. **Implemented Adafactor** - a memory-efficient factorized Adam alternative
3. **Compared optimizers** on VAE training, analyzing convergence, speed, and final performance
4. **Explored hyperparameters** for each optimizer

Key takeaways:
- **Shampoo** captures parameter correlations but is computationally expensive
- **Adafactor** is memory-efficient, making it suitable for very large models
- Modern optimizers can outperform Adam, but require careful hyperparameter tuning
- The best optimizer choice depends on model size, compute budget, and task