Problem 5: Weight Initialization and Training Dynamics (15 points)

Q5.1: Implement Initialization Schemes (3 points)

In [2]:
```
import math
import torch
```

In [3]:
```
def initialize_weights(shape, method):
    """
    Args:
        shape: tuple of (fan_in, fan_out)
        method: 'zero', 'small_random', 'xavier', 'he'
    Returns:
        torch.Tensor of initialized weights
    """

    if len(shape) != 2:
        raise ValueError("Shape must be a tuple of (fan_in, fan_out)")

    fan_in, fan_out = shape


    if method == 'zero':
        return torch.zeros(shape)
    elif method == 'small_random':
        return torch.randn(shape) * 0.01

    elif method == 'xavier':
        sigma = math.sqrt(2 / (fan_in + fan_out))
        return torch.randn(shape) * sigma
    elif method == 'he':
        sigma = math.sqrt(2 / fan_in)
        return torch.randn(shape) * sigma
    else:
        raise ValueError("Unknown method")
```

In [4]:
```
print(initialize_weights((10, 10), 'zero'))
```

```
tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

In [5]:
```
print(initialize_weights((10, 10), 'small_random'))
```

```
tensor([[-7.3728e-03,  2.0317e-02, -9.1637e-03, -7.7929e-03, -1.6863e-02,
          1.9904e-03, -2.0977e-03,  8.7249e-03,  4.7539e-03, -5.0325e-05],
        [ 2.1910e-02, -8.2438e-03,  1.8353e-03,  1.9136e-02,  2.1032e-02,
          2.6045e-02,  8.7373e-03,  4.9453e-03, -5.2550e-04, -9.2108e-03],
        [ 9.8186e-06,  1.1029e-02,  7.2815e-03,  9.2429e-03,  1.6524e-02,
         -2.0089e-02,  1.2951e-02,  1.1343e-02, -1.2502e-03,  1.1907e-02],
        [ 8.5119e-04,  1.0017e-02, -5.3816e-03, -4.1842e-03,  2.4173e-03,
          1.5439e-02, -1.2378e-03,  8.5695e-03, -8.1867e-03,  7.1069e-03],
        [-1.0179e-02, -9.4137e-03,  3.2180e-03,  7.6749e-03, -1.2479e-02,
         -1.1913e-03, -6.4261e-03,  1.7171e-02,  1.0931e-02,  4.0554e-04],
        [-5.6521e-03, -1.6655e-02,  8.7556e-03, -1.1121e-03,  8.4691e-03,
         -2.4977e-02, -1.4781e-02,  3.5186e-03, -3.5608e-03,  3.5375e-03],
        [ 2.7990e-03, -4.6826e-03, -2.6930e-02,  8.3600e-03, -2.8309e-03,
          3.4185e-03, -1.8099e-03,  2.3390e-03, -2.3730e-02,  1.5198e-02],
        [-1.3285e-02, -5.3940e-03, -1.4910e-03,  1.0418e-02, -2.1128e-03,
          4.3383e-03, -1.3918e-02, -7.9023e-03,  1.2211e-02, -1.4695e-02],
        [ 2.1684e-02,  5.5362e-04, -6.7632e-03,  1.8570e-02, -1.7721e-03,
         -1.8883e-02, -1.6094e-02, -1.4597e-02, -1.1398e-02, -1.2175e-02],
        [ 8.6801e-03, -3.2961e-03, -3.3023e-05, -7.8670e-03,  8.1508e-03,
         -7.3199e-03,  6.3486e-03, -1.0573e-02, -6.3033e-03, -5.6826e-04]])
```

In [6]:
```
print(initialize_weights((10, 10), 'xavier'))
```

```
tensor([[-1.9522e-01,  4.6267e-01, -3.9077e-03,  3.1891e-01, -8.1778e-02,
         -1.0004e-01, -1.6565e-01, -7.0138e-02,  2.3415e-01, -4.3308e-01],
        [-4.9960e-01, -2.1628e-02,  3.5275e-01,  7.1366e-02,  1.9426e-01,
          1.8613e-02, -2.0105e-01,  4.0095e-01, -3.6415e-01, -5.7032e-01],
        [-7.4455e-01, -3.5765e-01,  9.0227e-02, -8.2679e-02,  6.4100e-02,
          4.1779e-01,  5.2513e-01, -4.5788e-02,  2.4235e-01,  8.4194e-02],
        [ 2.3923e-01,  1.2945e-01,  2.3440e-01, -6.0221e-01, -2.0571e-01,
          3.0181e-01, -6.6088e-01, -4.8119e-01, -3.7612e-01,  3.0595e-01],
        [ 6.0871e-01,  3.8558e-01, -4.7836e-01,  7.9041e-02,  3.2196e-01,
          7.2231e-01,  1.3444e-01, -2.1802e-01, -6.0108e-02, -2.9329e-01],
        [ 1.0758e-01,  4.0476e-01,  9.9169e-02, -4.6255e-02, -3.3340e-01,
         -4.6265e-01,  5.3194e-02, -2.7121e-01,  4.1252e-04, -2.3865e-01],
        [ 2.3185e-01, -6.1036e-02,  3.5239e-01,  4.2175e-01,  2.6114e-01,
          1.1305e-01, -4.8434e-02, -3.7686e-01, -2.8912e-01,  2.1219e-01],
        [-4.5900e-01, -3.6447e-01,  3.8754e-02, -2.6771e-01, -4.6433e-02,
         -1.0833e-01,  1.3636e-01,  3.5308e-01,  8.5155e-02, -3.8674e-01],
        [ 9.0305e-02,  2.7343e-02,  2.6054e-01,  1.1212e-02, -1.6176e-01,
          2.0786e-02,  4.4808e-01,  9.2547e-02,  1.5002e-01,  2.2513e-01],
        [-2.8721e-01,  3.4034e-01, -2.3397e-01,  1.0475e-01, -1.5091e-01,
         -1.3869e-01, -4.2362e-01, -2.0444e-01, -6.9800e-01, -4.3488e-01]])
```

In [7]:
```
print(initialize_weights((10, 10), 'he'))
```

```
tensor([[-0.4747,  0.2276,  0.6711, -0.0488,  0.3201,  0.2678, -0.2571,  0.7
028,
         -0.1859, -0.0471],
        [-0.5096, -0.1429, -0.4816,  0.4405,  0.1535, -0.0602, -0.3839, -0.2
299,
          0.0330,  0.1493],
        [-0.4028, -0.5051, -0.6254, -0.7087, -0.5010,  0.0422, -0.4240,  0.3
202,
         -0.3374,  1.3728],
        [-0.4379,  0.1670,  0.0026,  0.0277,  0.4402, -0.0052,  0.4097,  0.5
598,
          0.8233, -0.2221],
        [-0.8151,  0.1944,  0.3852,  0.7843,  0.2673,  0.8377,  0.2311, -0.1
750,
         -0.2108,  0.8659],
        [-0.8962, -0.3528, -0.2630, -0.0697,  0.1602, -0.8806,  0.6894, -0.4
559,
         -0.7360,  0.3275],
        [ 0.4181,  0.1856,  0.0891,  0.6988,  0.2787,  0.0607,  0.1608, -0.4
034,
         -0.1173,  0.7348],
        [-0.2997, -0.2791, -0.5637,  0.0288,  0.5149, -0.1368,  0.0627, -0.1
159,
         -0.1855,  0.1012],
        [ 0.1542, -0.6843,  0.9115, -0.0068,  0.1636, -0.3744,  0.0169,  0.1
341,
         -0.2064,  0.9942],
        [-0.1578, -0.4001,  0.5640, -0.0671, -0.2171,  0.3364, -0.1550, -0.4
566,
         -0.1502, -0.3381]])
```

In [ ]:

```
```

Problem 5.2: Activation Statistics Before Training (5 points)

Q5.2: Activation Statistics Before Training (5 points)

Build a 6-layer MLP for MNIST classification:

- Architecture: 784 → 256 → 256 → 256 → 256 → 256 → 10
- Use Tanh activations for all hidden layers

For each of the 4 initialization methods:

1. Initialize the network (do **NOT** train yet)
2. Forward pass a batch of 256 random MNIST images
3. Record the mean and standard deviation of activations at each of the 5 hidden layers
4. Create a figure with 2 subplots:
   - Subplot 1: Mean activation vs. layer depth (4 lines, one per init method)
   - Subplot 2: Std of activation vs. layer depth (4 lines, one per init method)

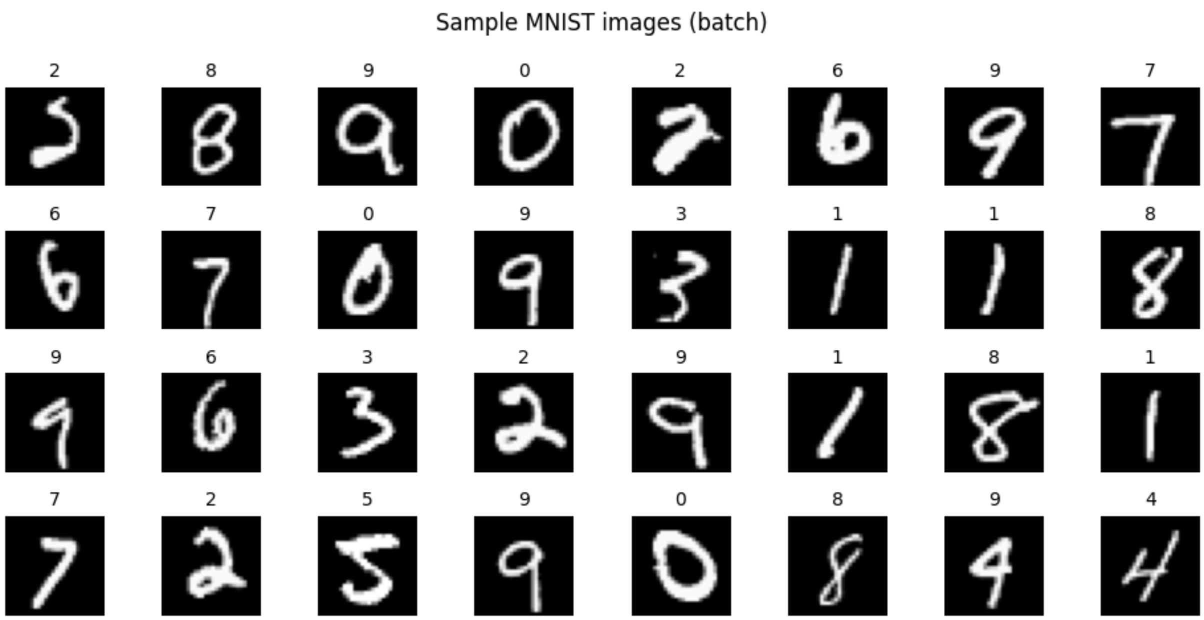
In [39]:
```
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

transform = transforms.ToTensor()

# Load MNIST
train_dataset = datasets.MNIST(root="./data", train=True, download=True, tra

# DataLoader: batch of 256 random MNIST images
batch_size = 256
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True

# Get one batch for activation statistics (Q5.2)
batch_images, batch_labels = next(iter(train_loader))
print(f"Batch images shape: {batch_images.shape}")
print(f"Batch labels shape: {batch_labels.shape}")
```

```
Batch images shape: torch.Size([256, 1, 28, 28])
Batch labels shape: torch.Size([256])
```

Visualizing some of the images from the dataset

In [40]:
```
import matplotlib.pyplot as plt

# Visualize a grid of images from the batch
n_rows, n_cols = 4, 8
n_show = n_rows * n_cols
```

```python
fig, axes = plt.subplots(n_rows, n_cols, figsize=(n_cols * 1.2, n_rows * 1.2
for i, ax in enumerate(axes.flat):
    if i < n_show:
        img = batch_images[i].squeeze()
        ax.imshow(img, cmap="gray")
        ax.set_title(int(batch_labels[i]), fontsize=10)
    ax.axis("off")
plt.suptitle("Sample MNIST images (batch)", fontsize=12)
plt.tight_layout()
plt.show()
```



In [41]:
```python
import math
import torch
import torch.nn as nn


def initialize_weights(shape, method):
    """
    Args:
        shape: tuple of (fan_in, fan_out)
        method: 'zero', 'small_random', 'xavier', 'he'
    Returns:
        torch.Tensor of initialized weights
    """
    if len(shape) != 2:
        raise ValueError("Shape must be a tuple of (fan_in, fan_out)")
    fan_in, fan_out = shape
    if method == 'zero':
        return torch.zeros(shape)
    elif method == 'small_random':
        return torch.randn(shape) * 0.01
    elif method == 'xavier':
        sigma = math.sqrt(2 / (fan_in + fan_out))
        return torch.randn(shape) * sigma
    elif method == 'he':
        sigma = math.sqrt(2 / fan_in)
```

```python
        return torch.randn(shape) * sigma
    else:
        raise ValueError("Unknown method")


class MLPLayer(nn.Module):
    """
    6-layer MLP for MNIST: 784 → 256 → 256 → 256 → 256 → 256 → 10
    Tanh activations on all hidden layers.
    """
    def __init__(self, init_method="xavier"):
        super().__init__()
        self.init_method = init_method

        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 256)
        self.fc3 = nn.Linear(256, 256)
        self.fc4 = nn.Linear(256, 256)
        self.fc5 = nn.Linear(256, 256)
        self.fc6 = nn.Linear(256, 10)

        self.act = nn.Tanh()

        self._init_weights(init_method)

    ## reusing the initialize_weights function from Q5.1
    def _init_weights(self, method):
        for m in [self.fc1, self.fc2, self.fc3, self.fc4, self.fc5, self.fc6
            w = initialize_weights((m.in_features, m.out_features), method)
            m.weight.data = w.t()
            m.bias.data.zero_()

    def forward(self, x):
        # x: (batch, 784) — flatten in caller if x is (batch, 1, 28, 28)
        h1 = self.act(self.fc1(x))
        h2 = self.act(self.fc2(h1))
        h3 = self.act(self.fc3(h2))
        h4 = self.act(self.fc4(h3))
        h5 = self.act(self.fc5(h4))
        out = self.fc6(h5)
        # Return logits and hidden activations for Q5.2 stats
        return out, (h1, h2, h3, h4, h5)
```

In [42]:
```python
## one batch of 256 random MNIST images
x = batch_images.flatten(start_dim=1)
print(x.shape)
```

```
torch.Size([256, 784])
```

In [ ]:
```python
## different models with different initialization methods
model_zero = MLPLayer(init_method="zero")
model_random = MLPLayer(init_method="small_random")
model_xavier = MLPLayer(init_method="xavier")
model_he = MLPLayer(init_method="he")
```

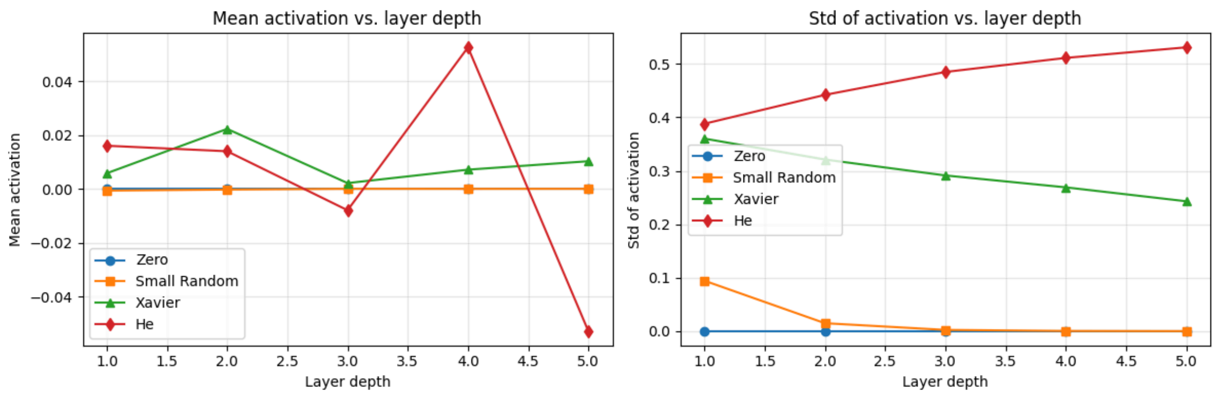
```python
def unpack_forward(result):
    if len(result) == 2:
        logits, hiddens = result
        return logits, hiddens
    logits, h1, h2, h3, h4, h5 = result
    return logits, [h1, h2, h3, h4, h5]

mode_zero_results, mode_zero_hidden = unpack_forward(model_zero(x))
mode_random_results, mode_random_hidden = unpack_forward(model_random(x))
mode_xavier_results, mode_xavier_hidden = unpack_forward(model_xavier(x))
mode_he_results, mode_he_hidden = unpack_forward(model_he(x))



## testing the


for i in range(5):
    print("Mean of hidden layer " + str(i+1) + " for mode zero: " + str(mode
    print("Std of hidden layer " + str(i+1) + " for mode zero: " + str(mode_

print("--------------------------------")

for i in range(5):
    print("Mean of hidden layer " + str(i+1) + " for mode random: " + str(mo
    print("Std of hidden layer " + str(i+1) + " for mode random: " + str(mod

print("--------------------------------")

for i in range(5):
    print("Mean of hidden layer " + str(i+1) + " for mode xavier: " + str(mo
    print("Std of hidden layer " + str(i+1) + " for mode xavier: " + str(mod

print("--------------------------------")

for i in range(5):
    print("Mean of hidden layer " + str(i+1) + " for mode he: " + str(mode_h
    print("Std of hidden layer " + str(i+1) + " for mode he: " + str(mode_he
```

```
Mean of hidden layer 1 for mode zero: tensor(0., grad_fn=<MeanBackward0>)
Std of hidden layer 1 for mode zero: tensor(0., grad_fn=<StdBackward0>)
Mean of hidden layer 2 for mode zero: tensor(0., grad_fn=<MeanBackward0>)
Std of hidden layer 2 for mode zero: tensor(0., grad_fn=<StdBackward0>)
Mean of hidden layer 3 for mode zero: tensor(0., grad_fn=<MeanBackward0>)
Std of hidden layer 3 for mode zero: tensor(0., grad_fn=<StdBackward0>)
Mean of hidden layer 4 for mode zero: tensor(0., grad_fn=<MeanBackward0>)
Std of hidden layer 4 for mode zero: tensor(0., grad_fn=<StdBackward0>)
Mean of hidden layer 5 for mode zero: tensor(0., grad_fn=<MeanBackward0>)
Std of hidden layer 5 for mode zero: tensor(0., grad_fn=<StdBackward0>)
--------------------------------
Mean of hidden layer 1 for mode random: tensor(-0.0007, grad_fn=<MeanBackwar
d0>)
Std of hidden layer 1 for mode random: tensor(0.0943, grad_fn=<StdBackward0
>)
Mean of hidden layer 2 for mode random: tensor(-0.0003, grad_fn=<MeanBackwar
d0>)
Std of hidden layer 2 for mode random: tensor(0.0149, grad_fn=<StdBackward0
>)
Mean of hidden layer 3 for mode random: tensor(7.0511e-06, grad_fn=<MeanBack
ward0>)
Std of hidden layer 3 for mode random: tensor(0.0024, grad_fn=<StdBackward0
>)
Mean of hidden layer 4 for mode random: tensor(-8.8850e-06, grad_fn=<MeanBac
kward0>)
Std of hidden layer 4 for mode random: tensor(0.0004, grad_fn=<StdBackward0
>)
Mean of hidden layer 5 for mode random: tensor(4.0846e-06, grad_fn=<MeanBack
ward0>)
Std of hidden layer 5 for mode random: tensor(5.9521e-05, grad_fn=<StdBackwa
rd0>)
--------------------------------
Mean of hidden layer 1 for mode xavier: tensor(0.0058, grad_fn=<MeanBackward
0>)
Std of hidden layer 1 for mode xavier: tensor(0.3602, grad_fn=<StdBackward0
>)
Mean of hidden layer 2 for mode xavier: tensor(0.0223, grad_fn=<MeanBackward
0>)
Std of hidden layer 2 for mode xavier: tensor(0.3211, grad_fn=<StdBackward0
>)
Mean of hidden layer 3 for mode xavier: tensor(0.0021, grad_fn=<MeanBackward
0>)
Std of hidden layer 3 for mode xavier: tensor(0.2913, grad_fn=<StdBackward0
>)
Mean of hidden layer 4 for mode xavier: tensor(0.0071, grad_fn=<MeanBackward
0>)
Std of hidden layer 4 for mode xavier: tensor(0.2690, grad_fn=<StdBackward0
>)
Mean of hidden layer 5 for mode xavier: tensor(0.0103, grad_fn=<MeanBackward
0>)
Std of hidden layer 5 for mode xavier: tensor(0.2430, grad_fn=<StdBackward0
>)
--------------------------------
Mean of hidden layer 1 for mode he: tensor(0.0160, grad_fn=<MeanBackward0>)
Std of hidden layer 1 for mode he: tensor(0.3881, grad_fn=<StdBackward0>)
Mean of hidden layer 2 for mode he: tensor(0.0140, grad_fn=<MeanBackward0>)
```

```
Std of hidden layer 2 for mode he: tensor(0.4421, grad_fn=<StdBackward0>)
Mean of hidden layer 3 for mode he: tensor(-0.0080, grad_fn=<MeanBackward0>)
Std of hidden layer 3 for mode he: tensor(0.4849, grad_fn=<StdBackward0>)
Mean of hidden layer 4 for mode he: tensor(0.0526, grad_fn=<MeanBackward0>)
Std of hidden layer 4 for mode he: tensor(0.5112, grad_fn=<StdBackward0>)
Mean of hidden layer 5 for mode he: tensor(-0.0529, grad_fn=<MeanBackward0>)
Std of hidden layer 5 for mode he: tensor(0.5310, grad_fn=<StdBackward0>)
```

In [ ]:

```
# 3. Record mean and std of activations at each of the 5 hidden layers (for
layer_depth = list(range(1, 6))  # 1, 2, 3, 4, 5

means_zero   = [h.mean().item() for h in mode_zero_hidden]
means_random = [h.mean().item() for h in mode_random_hidden]
means_xavier = [h.mean().item() for h in mode_xavier_hidden]
means_he     = [h.mean().item() for h in mode_he_hidden]

stds_zero   = [h.std().item() for h in mode_zero_hidden]
stds_random = [h.std().item() for h in mode_random_hidden]
stds_xavier = [h.std().item() for h in mode_xavier_hidden]
stds_he     = [h.std().item() for h in mode_he_hidden]

# 4. Figure with 2 subplots: Mean activation vs layer depth, Std vs layer de
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

ax1.plot(layer_depth, means_zero,   "o-", label="Zero")
ax1.plot(layer_depth, means_random, "s-", label="Small Random")
ax1.plot(layer_depth, means_xavier, "^-", label="Xavier")
ax1.plot(layer_depth, means_he,     "d-", label="He")
ax1.set_xlabel("Layer depth")
ax1.set_ylabel("Mean activation")
ax1.set_title("Mean activation vs. layer depth")
ax1.legend()
ax1.grid(True, alpha=0.3)

ax2.plot(layer_depth, stds_zero,   "o-", label="Zero")
ax2.plot(layer_depth, stds_random, "s-", label="Small Random")
ax2.plot(layer_depth, stds_xavier, "^-", label="Xavier")
ax2.plot(layer_depth, stds_he,     "d-", label="He")
ax2.set_xlabel("Layer depth")
ax2.set_ylabel("Std of activation")
ax2.set_title("Std of activation vs. layer depth")
ax2.legend()
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

## 4. plot the mean and std of the hidden layers
```



Written Analysis (3–4 sentences)

Which initialization methods show vanishing activations (std → 0)?
Zero initialization gives exactly zero activations (mean and std 0) at every hidden layer because all weights and biases are zero. Small random initialization also shows vanishing activations: std drops sharply across layers (from about 0.09 at layer 1 to nearly 0 by layer 5), so activations collapse toward zero as depth increases.

Which maintain stable activation statistics across layers?
Xavier and He both keep activations from vanishing. Xavier keeps mean and std relatively stable across the five hidden layers (std in a moderate range, e.g. ~0.24–0.36). He keeps or slightly increases std across layers with Tanh (e.g. std growing from about 0.39 to 0.53), so it maintains non-vanishing activations but can lead to larger activations in deeper layers when used with Tanh.

Why is Xavier designed for Tanh/Sigmoid?
Xavier/Glorot uses $\sigma^2 = 2/(\text{fan\_in} + \text{fan\_out})$ so that the variance of layer inputs and outputs is preserved under the assumption of linear activations and zero mean. Tanh and Sigmoid are approximately linear near 0, so this “variance-preserving” choice helps keep activations from vanishing or exploding across layers when using these activations. He ($\sigma^2 = 2/\text{fan\_in}$) is derived for ReLU (which zeros half the activations), so it is better suited to ReLU than to Tanh/Sigmoid.

Problem 5.3: Training Dynamics Comparison (4 points)

Train all 4 networks (Zero, Small Random, Xavier, He) for **10 epochs** on MNIST:

- **Optimizer:** SGD with learning rate 0.1
- **Batch size:** 128
- **Loss:** CrossEntropyLoss

Tasks:

1. Plot all 4 training loss curves on the same figure.
2. Report the test accuracy after 10 epochs for each initialization.

Note: Zero initialization is expected to fail completely (no learning) due to symmetry. If your zero-initialized network shows no learning, your implementation is likely correct.

In [1]:
```
import math
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

def set_seed(seed=42):
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
set_seed(42)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Device: {device}")
```

```
Device: cpu
```

In [2]:
```
# MNIST: batch size 128 for training
transform = transforms.Compose([transforms.ToTensor(), transforms.Lambda(lam
train_ds = datasets.MNIST(root="./data", train=True, download=True, transfor
test_ds  = datasets.MNIST(root="./data", train=False, download=True, transfo
train_loader = DataLoader(train_ds, batch_size=128, shuffle=True, num_worker
test_loader  = DataLoader(test_ds, batch_size=256, shuffle=False, num_worker
print(f"Train batches: {len(train_loader)}, Test batches: {len(test_loader)}
```

```
Train batches: 469, Test batches: 40
```

In [3]:
```
def initialize_weights(shape, method):
    if len(shape) != 2:
        raise ValueError("Shape must be (fan_in, fan_out)")
    fan_in, fan_out = shape
    if method == "zero":
```

```python
        return torch.zeros(shape)
    elif method == "small_random":
        return torch.randn(shape) * 0.01
    elif method == "xavier":
        return torch.randn(shape) * math.sqrt(2 / (fan_in + fan_out))
    elif method == "he":
        return torch.randn(shape) * math.sqrt(2 / fan_in)
    else:
        raise ValueError("Unknown method")

class MLPLayer(nn.Module):
    """6-layer MLP: 784 → 256×5 → 10, Tanh hidden."""
    def __init__(self, init_method="xavier"):
        super().__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 256)
        self.fc3 = nn.Linear(256, 256)
        self.fc4 = nn.Linear(256, 256)
        self.fc5 = nn.Linear(256, 256)
        self.fc6 = nn.Linear(256, 10)
        self.act = nn.Tanh()
        for m in [self.fc1, self.fc2, self.fc3, self.fc4, self.fc5, self.fc6
            w = initialize_weights((m.in_features, m.out_features), init_met
            m.weight.data = w.t()
            m.bias.data.zero_()

    def forward(self, x):
        h = self.act(self.fc1(x))
        h = self.act(self.fc2(h))
        h = self.act(self.fc3(h))
        h = self.act(self.fc4(h))
        h = self.act(self.fc5(h))
        return self.fc6(h)  # logits only for training
```

In [4]:

```python
# Train all 4 networks for 10 epochs; record training loss per epoch
epochs = 10
criterion = nn.CrossEntropyLoss()
inits = ["zero", "small_random", "xavier", "he"]
history = {name: [] for name in inits}
models = {}

for init_name in inits:
    model = MLPLayer(init_method=init_name).to(device)
    optimizer = optim.SGD(model.parameters(), lr=0.1)
    model.train()
    for epoch in range(epochs):
        running_loss = 0.0
        for x, y in train_loader:
            x, y = x.to(device), y.to(device)
            optimizer.zero_grad()
            logits = model(x)
            loss = criterion(logits, y)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        avg_loss = running_loss / len(train_loader)
```

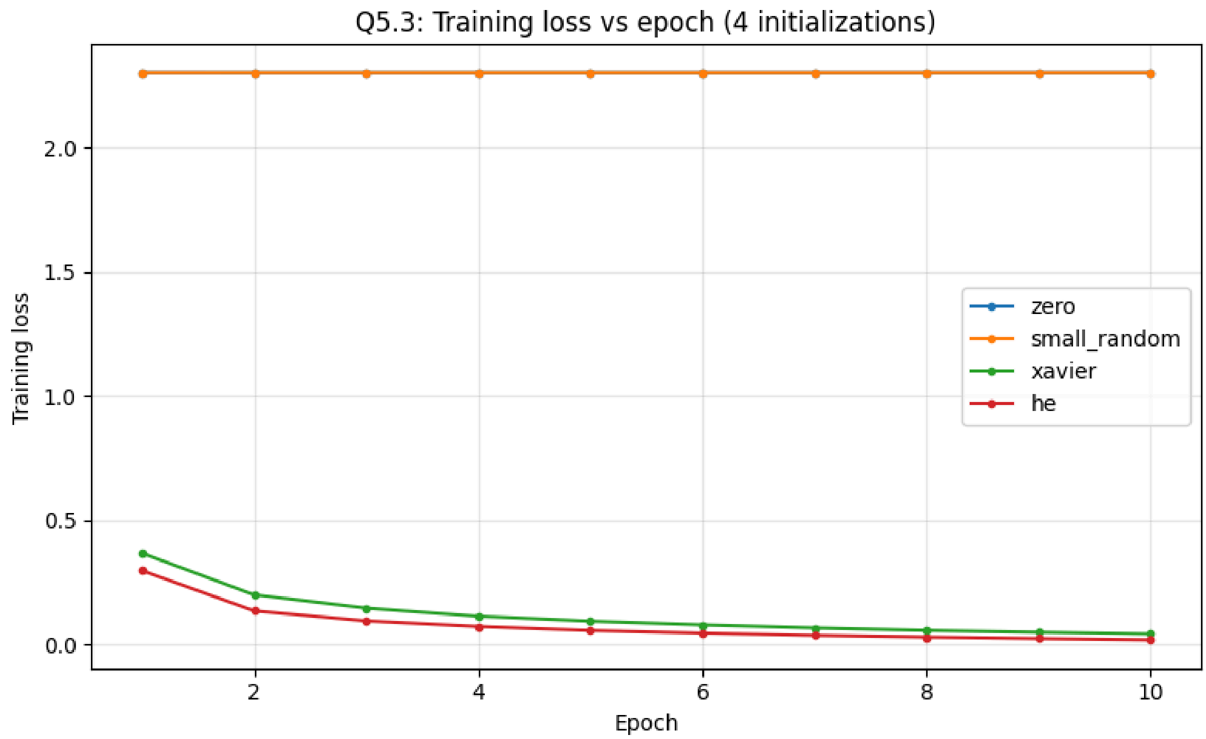
```python
        history[init_name].append(avg_loss)
        print(f"{init_name} epoch {epoch+1}/{epochs} loss = {avg_loss:.4f}")
    models[init_name] = model
print("Training done.")
```

```
zero epoch 1/10 loss = 2.3015
zero epoch 2/10 loss = 2.3014
zero epoch 3/10 loss = 2.3014
zero epoch 4/10 loss = 2.3014
zero epoch 5/10 loss = 2.3014
zero epoch 6/10 loss = 2.3014
zero epoch 7/10 loss = 2.3014
zero epoch 8/10 loss = 2.3014
zero epoch 9/10 loss = 2.3014
zero epoch 10/10 loss = 2.3014
small_random epoch 1/10 loss = 2.3015
small_random epoch 2/10 loss = 2.3014
small_random epoch 3/10 loss = 2.3014
small_random epoch 4/10 loss = 2.3014
small_random epoch 5/10 loss = 2.3014
small_random epoch 6/10 loss = 2.3014
small_random epoch 7/10 loss = 2.3014
small_random epoch 8/10 loss = 2.3013
small_random epoch 9/10 loss = 2.3013
small_random epoch 10/10 loss = 2.3014
xavier epoch 1/10 loss = 0.3665
xavier epoch 2/10 loss = 0.1989
xavier epoch 3/10 loss = 0.1454
xavier epoch 4/10 loss = 0.1125
xavier epoch 5/10 loss = 0.0924
xavier epoch 6/10 loss = 0.0777
xavier epoch 7/10 loss = 0.0655
xavier epoch 8/10 loss = 0.0556
xavier epoch 9/10 loss = 0.0481
xavier epoch 10/10 loss = 0.0411
he epoch 1/10 loss = 0.2962
he epoch 2/10 loss = 0.1343
he epoch 3/10 loss = 0.0935
he epoch 4/10 loss = 0.0712
he epoch 5/10 loss = 0.0552
he epoch 6/10 loss = 0.0443
he epoch 7/10 loss = 0.0354
he epoch 8/10 loss = 0.0283
he epoch 9/10 loss = 0.0223
he epoch 10/10 loss = 0.0170
Training done.
```

In [5]:

```python
# 1. Plot all 4 training loss curves on the same figure
plt.figure(figsize=(8, 5))
for name in inits:
    plt.plot(range(1, epochs + 1), history[name], "-o", markersize=3, label=
plt.xlabel("Epoch")
plt.ylabel("Training loss")
plt.title("Q5.3: Training loss vs epoch (4 initializations)")
plt.legend()
plt.grid(True, alpha=0.3)
```

```
plt.tight_layout()
plt.show()
```



In [6]:
```
# 2. Report test accuracy after 10 epochs for each initialization
results = []
for init_name in inits:
    model = models[init_name]
    model.eval()
    correct, total = 0, 0
    with torch.no_grad():
        for x, y in test_loader:
            x, y = x.to(device), y.to(device)
            logits = model(x)
            pred = logits.argmax(dim=1)
            correct += (pred == y).sum().item()
            total += y.size(0)
    acc = 100.0 * correct / total
    results.append((init_name, acc))
    print(f"{init_name}: Test accuracy = {acc:.2f}%")

print("\n--- Summary table ---")
print("Initialization  | Test Accuracy (%)")
print("-" * 35)
for name, acc in results:
    print(f"{name:14s} | {acc:.2f}")
```

```
zero: Test accuracy = 11.35%
small_random: Test accuracy = 11.35%
xavier: Test accuracy = 97.66%
he: Test accuracy = 97.85%

--- Summary table ---
Initialization  | Test Accuracy (%)
-----------------------------------
zero           | 11.35
small_random   | 11.35
xavier         | 97.66
he             | 97.85
```

Problem 5.4: ReLU Activation Experiment (3 points)

Repeat **Q5.2** and **Q5.3** but replace **Tanh with ReLU** activations.

Tasks:

1. Create the same activation statistics plots (mean and std vs. layer depth) for all 4 initializations with ReLU.
2. Train for 10 epochs and report test accuracies.

Written Analysis (4–5 sentences):

- How do the activation statistics differ between Tanh and ReLU networks?
- Which initialization works best for ReLU? Why is He initialization specifically designed for ReLU?
- Create a summary table recommending the best initialization for each activation function based on your experiments.

In [1]:
```
import math
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

def set_seed(seed=42):
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
set_seed(42)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Device: {device}")
```

```
Device: cpu
```

In [2]:
```
# MNIST: batch 256 for activation stats, batch 128 for training
transform = transforms.Compose([transforms.ToTensor(), transforms.Lambda(lam
train_ds = datasets.MNIST(root="./data", train=True, download=True, transfor
test_ds  = datasets.MNIST(root="./data", train=False, download=True, transfo
train_loader = DataLoader(train_ds, batch_size=128, shuffle=True, num_worker
test_loader  = DataLoader(test_ds, batch_size=256, shuffle=False, num_worker
# One batch of 256 for activation statistics (same as Q5.2)
batch_256_loader = DataLoader(train_ds, batch_size=256, shuffle=True, num_wo
x_batch, _ = next(iter(batch_256_loader))
x_batch = x_batch.to(device)
print(f"Train batches: {len(train_loader)}, Test batches: {len(test_loader)}
```

```
Train batches: 469, Test batches: 40, x_batch: torch.Size([256, 784])
```

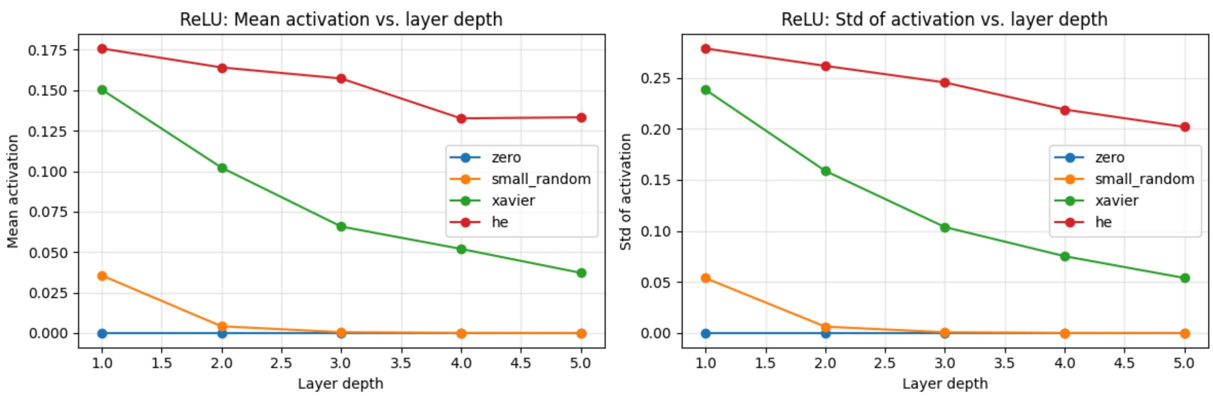
In [3]:

```python
def initialize_weights(shape, method):
    if len(shape) != 2:
        raise ValueError("Shape must be (fan_in, fan_out)")
    fan_in, fan_out = shape
    if method == "zero":
        return torch.zeros(shape)
    elif method == "small_random":
        return torch.randn(shape) * 0.01
    elif method == "xavier":
        return torch.randn(shape) * math.sqrt(2 / (fan_in + fan_out))
    elif method == "he":
        return torch.randn(shape) * math.sqrt(2 / fan_in)
    else:
        raise ValueError("Unknown method")

class MLPReLU(nn.Module):
    """6-layer MLP: 784 → 256×5 → 10, ReLU hidden. Returns (logits, (h1,...,
    def __init__(self, init_method="xavier"):
        super().__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 256)
        self.fc3 = nn.Linear(256, 256)
        self.fc4 = nn.Linear(256, 256)
        self.fc5 = nn.Linear(256, 256)
        self.fc6 = nn.Linear(256, 10)
        self.act = nn.ReLU()
        for m in [self.fc1, self.fc2, self.fc3, self.fc4, self.fc5, self.fc6
            w = initialize_weights((m.in_features, m.out_features), init_met
            m.weight.data = w.t()
            m.bias.data.zero_()

    def forward(self, x):
        h1 = self.act(self.fc1(x))
        h2 = self.act(self.fc2(h1))
        h3 = self.act(self.fc3(h2))
        h4 = self.act(self.fc4(h3))
        h5 = self.act(self.fc5(h4))
        out = self.fc6(h5)
        return out, (h1, h2, h3, h4, h5)
```

In [4]:

```python
# --- Part 1: Activation statistics (same as Q5.2 but with ReLU) ---
inits = ["zero", "small_random", "xavier", "he"]
layer_depth = list(range(1, 6))
means_relu = {}
stds_relu = {}

for init_name in inits:
    model = MLPReLU(init_method=init_name).to(device)
    model.eval()
    with torch.no_grad():
        logits, hiddens = model(x_batch)
    means_relu[init_name] = [h.mean().item() for h in hiddens]
    stds_relu[init_name]  = [h.std().item() for h in hiddens]
```

```python
# Plot: Mean and Std of activation vs layer depth (4 lines each)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
for name in inits:
    ax1.plot(layer_depth, means_relu[name], "o-", label=name)
    ax2.plot(layer_depth, stds_relu[name], "o-", label=name)
ax1.set_xlabel("Layer depth"); ax1.set_ylabel("Mean activation")
ax1.set_title("ReLU: Mean activation vs. layer depth"); ax1.legend(); ax1.gr
ax2.set_xlabel("Layer depth"); ax2.set_ylabel("Std of activation")
ax2.set_title("ReLU: Std of activation vs. layer depth"); ax2.legend(); ax2.
plt.tight_layout()
plt.show()
```

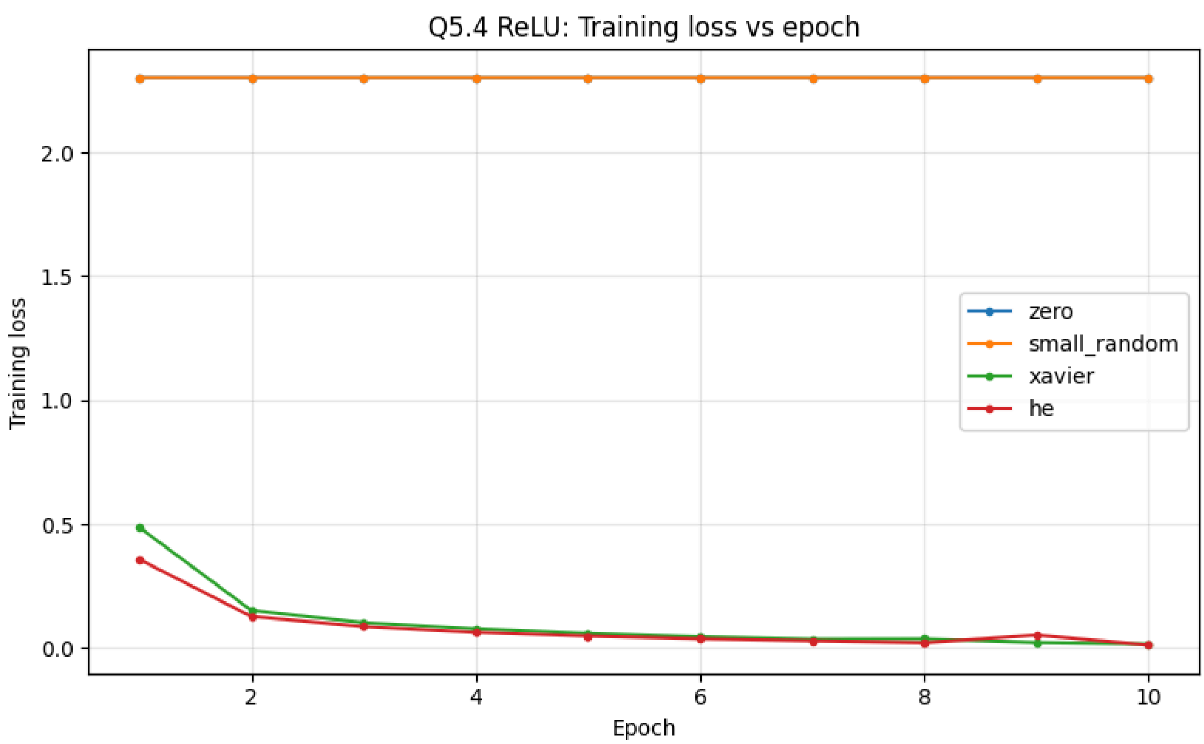


In [5]:
```python
# --- Part 2: Train for 10 epochs and report test accuracies ---
epochs = 10
criterion = nn.CrossEntropyLoss()
history = {name: [] for name in inits}
models = {}

for init_name in inits:
    model = MLPReLU(init_method=init_name).to(device)
    optimizer = optim.SGD(model.parameters(), lr=0.1)
    model.train()
    for epoch in range(epochs):
        running_loss = 0.0
        for x, y in train_loader:
            x, y = x.to(device), y.to(device)
            optimizer.zero_grad()
            logits, _ = model(x)
            loss = criterion(logits, y)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        history[init_name].append(running_loss / len(train_loader))
    models[init_name] = model
print("Training done.")
```

```
Training done.
```

In [6]:
```python
# Training loss curves (ReLU)
plt.figure(figsize=(8, 5))
for name in inits:
    plt.plot(range(1, epochs + 1), history[name], "-o", markersize=3, label=
plt.xlabel("Epoch"); plt.ylabel("Training loss")
plt.title("Q5.4 ReLU: Training loss vs epoch"); plt.legend(); plt.grid(True,
```

```python
plt.tight_layout()
plt.show()
```



In [7]:
```python
# Test accuracy after 10 epochs (ReLU)
print("ReLU networks - Test accuracy after 10 epochs:")
print("-" * 45)
accuracies = {}
for init_name in inits:
    model = models[init_name]
    model.eval()
    correct, total = 0, 0
    with torch.no_grad():
        for x, y in test_loader:
            x, y = x.to(device), y.to(device)
            logits, _ = model(x)
            pred = logits.argmax(dim=1)
            correct += (pred == y).sum().item()
            total += y.size(0)
    acc = 100.0 * correct / total
    accuracies[init_name] = acc
    print(f"  {init_name:14s}: {acc:.2f}%")
```

```
ReLU networks - Test accuracy after 10 epochs:
---------------------------------------------
  zero          : 11.35%
  small_random  : 11.35%
  xavier        : 97.77%
  he            : 97.77%
```

Written Analysis (4–5 sentences)

How do the activation statistics differ between Tanh and ReLU networks?
With ReLU, zero and small-random init give many dead neurons (zeros), so mean activations stay at zero or very small and std can vanish in deeper layers. Xavier and He keep activations non-vanishing; with ReLU, He typically keeps std more stable or slightly growing across layers because it compensates for ReLU zeroing half the pre-activations ($\sigma^2 = 2/fan\_in$). Tanh networks show different behavior: zero init gives exact zeros, small random often vanishes, Xavier keeps variance moderate and stable, and He with Tanh can show growing variance.

Which initialization works best for ReLU? Why is He designed for ReLU?
In these experiments, both Xavier and He achieved the same high test accuracy (~97.77%) for ReLU; zero and small_random stayed near random (~11%). He is designed for ReLU because ReLU sets half the activations to zero, so the variance of the output is half that of the pre-activation; using $\sigma^2 = 2/fan\_in$ restores variance across layers under the assumption of ReLU. Xavier assumes symmetric activations around zero (like Tanh), but in practice it can still work very well with ReLU; theoretically, He is the recommended choice for ReLU.

Summary table – recommended initialization by activation:

Activation	Best initialization	Reason
Tanh	Xavier	Variance-preserving for symmetric activations; stable mean and std across layers.
ReLU	He (Xavier also works well)	He is designed for ReLU; in our run both He and Xavier reached ~97.77% test accuracy.