Maximal Update Parameterization

In this problem, we will examine the training of a simple MLP with hidden layers of varying widths. We will then investigate the maximal update parameterization (muP) which will allow us to use a single global learning rate to jointly train layers of any width.

Note: This homework question is new this year and it is messier than usual. We felt it was worth it to get it out so you can play with these new techniques. If you're feeling stuck, don't hesistate to ask questions on Ed.

In [2]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from keras.datasets import mnist
import matplotlib.pyplot as plt

%matplotlib inline

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

class MLP(nn.Module):
    def __init__(self, input_size=784, hidden_sizes = [8, 16, 32, 64, 128], num_
        super(MLP, self).__init__()
        all_hidden_sizes = [input_size] + hidden_sizes + [num_classes]
        self.layers = nn.ModuleList()
        for i in range(len(all_hidden_sizes)-1):
            self.layers.append(nn.Linear(all_hidden_sizes[i], all_hidden_sizes[i
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        activations = []
        x = x.view(x.size(0), -1)  # Flatten: (batch_size, 28*28)
        for layer in self.layers[:-1]:
            x = self.sigmoid(layer(x))
            activations.append(x)
        x = self.layers[-1](x)
        activations = activations[1:]
        return x, [a.detach() for a in activations]

# Load MNIST data
(train_images, train_labels), (valid_images, valid_labels) = mnist.load_data()

# Normalize pixel values to [0, 1]
train_images = train_images.astype(np.float32) / 255.0
valid_images = valid_images.astype(np.float32) / 255.0

# Convert to PyTorch tensors
train_images = torch.from_numpy(train_images)
train_labels = torch.from_numpy(train_labels).long()
valid_images = torch.from_numpy(valid_images)
valid_labels = torch.from_numpy(valid_labels).long()
```

```
def rms(x, dim):
    return torch.sqrt(torch.mean(x**2, dim=dim))
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-dataset
s/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 2s 0us/step
```

In [15]:
```
from torch.optim.optimizer import Optimizer
from typing import Any
class SimpleAdam(Optimizer):
    def __init__(
        self,
        params: Any,
        lr: float = 1e-1,
        b1: float = 0.9,
        b2: float = 0.999,
    ):
        defaults = dict(lr=lr, b1=b1, b2=b2,)
        super(SimpleAdam, self).__init__(params, defaults)

    @torch.no_grad()
    def step(self):
        for group in self.param_groups:
            for p in group['params']:
                grad = p.grad.data

                state = self.state[p]
                if len(state) == 0: # Initialization
                    state["step"] = torch.tensor(0.0)
                    state['momentum'] = torch.zeros_like(p)
                    state['variance'] = torch.zeros_like(p)

                state['step'] += 1
                m = state['momentum']
                m.lerp_(grad, 1-group["b1"])
                v = state['variance']
                v.lerp_(grad**2, 1-group["b2"])

                m_hat = m / (1 - group["b1"]**state['step'])
                v_hat = v / (1 - group["b2"]**state['step'])
                u = m_hat / (torch.sqrt(v_hat) + 1e-16)

                p.add_(u, alpha=-group['lr'])
        return None
```

In [16]:
```
batch_idx = np.random.randint(0, len(train_images), size=64)
def train_one_step(mlp=MLP, hiddens=[8, 16, 64, 64, 64, 256, 256, 1024], optimiz
    model = mlp(hidden_sizes=hiddens).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optimizer(model.parameters(), lr=lr)

    prev_activations = None
    for i in range(2):
        images_batch = train_images[batch_idx]
        labels_batch = train_labels[batch_idx]
        images_batch, labels_batch = images_batch.to(device), labels_batch.to(de

        optimizer.zero_grad()
        outputs, activations = model(images_batch)
        loss = criterion(outputs, labels_batch)
        loss.backward()
        optimizer.step()

        if i > 0:
            print([a.shape for a in activations])
            activation_deltas = [a - pa for a, pa in zip(activations, prev_activ
            activation_deltas_rms = [torch.mean(rms(a, dim=-1)) for a in activat
        prev_activations = activations

    # plot deltas
    deltas = np.array(activation_deltas_rms)
    fig, axs = plt.subplots(1, figsize=(8, 4))
    axs.set_title(f'RMS of activation deltas per layer ({label})')
    axs.set_xlabel('Hidden Size of activation')
    axs.bar(np.arange(deltas.shape[0]), deltas)
    axs.set_xticks(np.arange(deltas.shape[0]))
    axs.set_xticklabels(hiddens[1:])
    plt.show()
train_one_step(optimizer=SimpleAdam)
```

```
[torch.Size([64, 16]), torch.Size([64, 64]), torch.Size([64, 64]), torch.Size([6
4, 64]), torch.Size([64, 256]), torch.Size([64, 256]), torch.Size([64, 1024])]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[16], line 34
     32     axs.set_xticklabels(hiddens[1:])
     33     plt.show()
---> 34 train_one_step(optimizer=SimpleAdam)

Cell In[16], line 26, in train_one_step(mlp, hiddens, optimizer, label, lr)
     23     prev_activations = activations
     25 # plot deltas
---> 26 deltas = np.array(activation_deltas_rms)
     27 fig, axs = plt.subplots(1, figsize=(8, 4))
     28 axs.set_title(f'RMS of activation deltas per layer ({label})')

File d:\Anaconda\envs\cs182_env\lib\site-packages\torch\_tensor.py:1225, in Tenso
r.__array__(self, dtype)
   1223     return handle_torch_function(Tensor.__array__, (self,), self, dtype=d
type)
   1224 if dtype is None:
-> 1225     return self.numpy()
   1226 else:
   1227     return self.numpy().astype(dtype, copy=False)

TypeError: can't convert cuda:0 device type tensor to numpy. Use Tensor.cpu() to
copy the tensor to host memory first.
```

a. Examining the norms of a heterogenous MLP.

Run the above cell, which trains a neural network for a single gradient step, then examines the effect of that step on the resulting activations. What are the dimensions of each layer in the neural network?

Answer:

How does the dimensionality of the layer affect the RMS norm of the activation deltas?

Answer:

Change the widths of some of your neural network layers, and recreate the plot -- did the RMS values change as expected?

Answer:

In [ ]:
```
# TODO: Call some plotting code here.
```

b. Examining the norms of the updates to the weights.

In the provided code above, we plotted the change in norms of the *activation vectors*. Now, you will examine the change in the weights themselves. Create a version of the above function that runs a single gradient step, then for each dense layer plot:

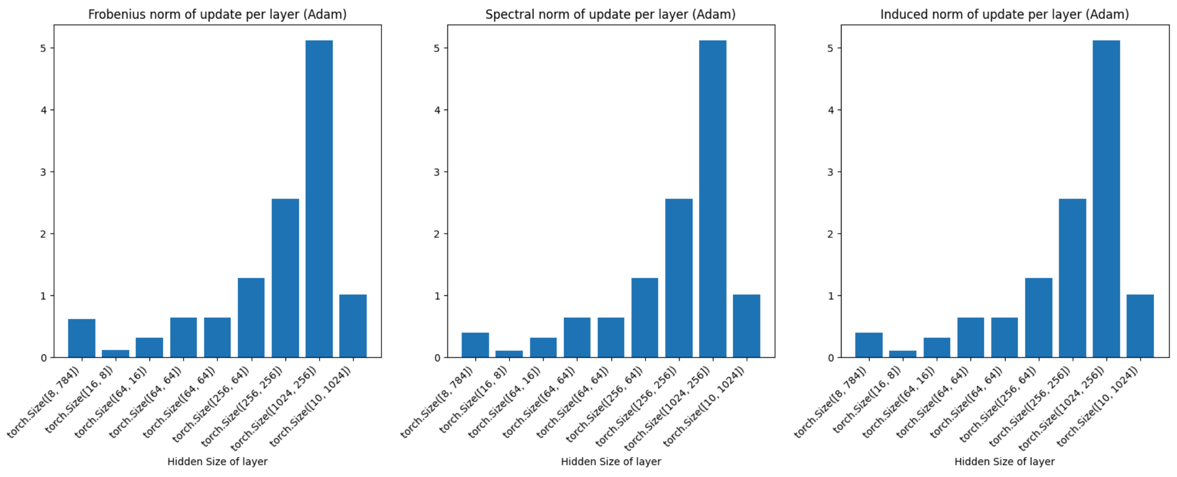
- The *Frobenius* norm of the update.
- The *spectral* norm of the update.
- The *RMS-RMS induced norm* of the update.

Which one of these norms correlates the most with the RMS norms of the activations?

Answer:

You should calculate your updates as `new_dense_parameter - old_dense_parameter`.

In [6]:
```
### Solution
batch_idx = np.random.randint(0, len(train_images), size=64)
def train_one_step_matrices(mlp=MLP, hiddens=[8, 16, 64, 64, 64, 256, 256, 1024]
    model = mlp(hidden_sizes=hiddens).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optimizer(model.parameters(), lr=lr)

    old_params = [p.detach().clone() for p in model.parameters()]

    for i in range(1):
        images_batch = train_images[batch_idx]
        labels_batch = train_labels[batch_idx]
        images_batch, labels_batch = images_batch.to(device), labels_batch.to(de

        optimizer.zero_grad()
        outputs, activations = model(images_batch)
        loss = criterion(outputs, labels_batch)
        loss.backward()
        optimizer.step()

    new_params = [p.detach().clone() for p in model.parameters()]
    delta_params = [new_p - old_p for new_p, old_p in zip(new_params, old_params

    frob_norms = []
    spectral_norms = []
    induced_norms = []
    p_shapes = []

    for p in delta_params:
        if len(p.shape) == 2:
            ### TODO: Log the respective norms.
            # Frobenius norm
            frob_norm = torch.norm(p, p='fro').item()
            frob_norms.append(frob_norm)

            # Spectral norm (using PyTorch's built-in function)
            spectral_norm = torch.linalg.matrix_norm(p, ord=2).item()
            spectral_norms.append(spectral_norm)

            # Induced norm (operator norm) - same as spectral norm for matrices
            induced_norm = torch.linalg.matrix_norm(p, ord=2).item()
            induced_norms.append(induced_norm)

            # Store parameter shape for plotting
            p_shapes.append(str(p.shape))
            ###


    fig, axs = plt.subplots(1, 3, figsize=(20, 6))

    axs[0].set_title(f'Frobenius norm of update per layer ({label})')
    axs[0].set_xlabel('Hidden Size of layer')
    axs[0].bar(np.arange(len(frob_norms)), frob_norms)
    axs[0].set_xticks(np.arange(len(frob_norms)), p_shapes, rotation=45, ha='rig

    axs[1].set_title(f'Spectral norm of update per layer ({label})')
    axs[1].set_xlabel('Hidden Size of layer')
    axs[1].bar(np.arange(len(spectral_norms)), spectral_norms)
    axs[1].set_xticks(np.arange(len(spectral_norms)), p_shapes, rotation=45, ha=

    axs[2].set_title(f'Induced norm of update per layer ({label})')
    axs[2].set_xlabel('Hidden Size of layer')
    axs[2].bar(np.arange(len(induced_norms)), induced_norms)
    axs[2].set_xticks(np.arange(len(induced_norms)), p_shapes, rotation=45, ha='
    plt.show()
train_one_step_matrices(optimizer=SimpleAdam)
```



c. Implementing muP

We will now implement muP scaling. Modify the starter code below to set a per-layer learning rate such that the resulting RMS activation-deltas are uniform scale, regardless of the layer widths. Plot the resulting activation-deltas on at least two sets of widths.

Note: Even with the correct scaling, the first 2-3 activation-deltas may have a lower norm than the rest. Can you think of a reason why this might be the case?

In [12]:
```
from torch.optim.optimizer import Optimizer
from typing import Any
# Before (causes error if tensor is on GPU)

class SimpleAdamMuP(Optimizer):
    def __init__(
        self,
        params: Any,
        lr: float = 1e-1,
        b1: float = 0.9,
        b2: float = 0.999,
    ):
        defaults = dict(lr=lr, b1=b1, b2=b2,)
        super(SimpleAdamMuP, self).__init__(params, defaults)

    @torch.no_grad()
    def step(self):
        for group in self.param_groups:
            for p in group['params']:
                grad = p.grad.data

                state = self.state[p]
                if len(state) == 0: # Initialization
                    state["step"] = torch.tensor(0.0)
                    state['momentum'] = torch.zeros_like(p)
                    state['variance'] = torch.zeros_like(p)

                state['step'] += 1
                m = state['momentum']
                m.lerp_(grad, 1-group["b1"])
                v = state['variance']
                v.lerp_(grad**2, 1-group["b2"])

                m_hat = m / (1 - group["b1"]**state['step'])
                v_hat = v / (1 - group["b2"]**state['step'])
                u = m_hat / (torch.sqrt(v_hat) + 1e-16)

                ############################
                ### Todo: Adjust the per-layer learning rate scaling factor so p
                ### Hint for part e: The following tricks will help you retain p
                ###  - Treat biases as a hidden layer with size (d_out, 1). You
                ###  - For the input layer, a fudge factor of 10 appears to help
                ###  - For the output layer, we find it is best to ignore the mu
                ############################
                lr = group['lr']
                # Determine if parameter is a weight or bias based on its shape
                param_shape = p.shape
                is_bias = len(param_shape) == 1  # Bias parameters are 1D

                if is_bias:
                    # For biases: treat as hidden layer with size (d_out, 1) and
                    d_out = param_shape[0]
                    scaling = 0.01 * (1.0 / (d_out ** 0.5))  # Square root scali
                else:
                    # For weights: determine if it's input, hidden, or output la
                    if len(param_shape) == 2:
                        d_in, d_out = param_shape

                        # Check if it's the output layer (heuristic: if this is
                        # Since we can't easily determine the position, we'll us
                        # Looking at the hiddens arrays, the last element is 102
                        # The output layer would likely have a different pattern
                        is_output_layer = d_out == 10  # Assuming output is 10 c
                        is_input_layer = d_in == 784  # Assuming input is 28x28=

                        if is_output_layer:
                            # For output layer: use fixed learning rate of 0.003
                            lr = 0.003
                            scaling = 1.0
                        elif is_input_layer:
                            # For input layer: use fudge factor of 10 and square
                            scaling = 10.0 * (d_in ** 0.5)  # Square root scalin
                        else:
                            # For hidden layers: standard mup scaling
                            scaling = (d_in ** 0.5)  # Square root scaling for m
                    else:
                        # For other parameter shapes, use default scaling
                        scaling = 1.0

                # Apply scaling to the learning rate
                lr = lr * scaling
                ############################
                ############################

                p.add_(u, alpha=-lr)
        return None
train_one_step(optimizer=SimpleAdamMuP, lr=2, label="Adam MuP", hiddens=[8, 16,
train_one_step(optimizer=SimpleAdamMuP, lr=2, label="Adam MuP", hiddens=[8, 16,
```

```
[torch.Size([64, 16]), torch.Size([64, 64]), torch.Size([64, 64]), torch.Size([6
4, 64]), torch.Size([64, 256]), torch.Size([64, 256]), torch.Size([64, 1024])]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[12], line 87
     85                 p.add_(u, alpha=-lr)
     86         return None
---> 87 train_one_step(optimizer=SimpleAdamMuP, lr=2, label="Adam MuP", hiddens=
[8, 16, 64, 64, 64, 256, 256, 1024])
     88 train_one_step(optimizer=SimpleAdamMuP, lr=2, label="Adam MuP", hiddens=
[8, 16, 32, 64, 128, 256, 512, 1024])

Cell In[4], line 26, in train_one_step(mlp, hiddens, optimizer, label, lr)
     23     prev_activations = activations
     25 # plot deltas
---> 26 deltas = np.array(activation_deltas_rms)
     27 fig, axs = plt.subplots(1, figsize=(8, 4))
     28 axs.set_title(f'RMS of activation deltas per layer ({label})')

File d:\Anaconda\envs\cs182_env\lib\site-packages\torch\_tensor.py:1225, in Tenso
r.__array__(self, dtype)
   1223     return handle_torch_function(Tensor.__array__, (self,), self, dtype=d
type)
   1224 if dtype is None:
-> 1225     return self.numpy()
   1226 else:
   1227     return self.numpy().astype(dtype, copy=False)

TypeError: can't convert cuda:0 device type tensor to numpy. Use Tensor.cpu() to
copy the tensor to host memory first.
```

d. Per-Weight Multipliers

An alternative way to implement muP is to adjust the *network graph* itself, rather than the optimizer. Implement this below, and recreate the above uniformly-scaled graph when using the *Adam* (not muP) optimizer. We have disabled biases to simplify the problem.

Why is multiplying the output of a layer by a constant the same as adjusting the learning-rate of that layer (when using Adam or SignGD)?

Answer:

In [ ]:
```
class ScaledMLP(nn.Module):
    def __init__(self, input_size=784, hidden_sizes = [8, 16, 32, 64, 128], num_
        super(ScaledMLP, self).__init__()
        all_hidden_sizes = [input_size] + hidden_sizes + [num_classes]
        self.layers = nn.ModuleList()
        for i in range(len(all_hidden_sizes)-1):
            self.layers.append(nn.Linear(all_hidden_sizes[i], all_hidden_sizes[i
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        activations = []
        x = x.view(x.size(0), -1)  # Flatten: (batch_size, 28*28)
        for layer in self.layers[:-1]:
            x = layer(x)
            ## TODO
            pass
            ##
            x = self.sigmoid(x)
            activations.append(x)
        x = self.layers[-1](x)
        activations = activations[1:]
        return x, [a.detach() for a in activations]

train_one_step(mlp=ScaledMLP, optimizer=SimpleAdam)
```

e. Hyperparameter Transfer

Run the following code, which will perform a sweep over learning rates for 3-layer MLPs of increasing width using Adam. How does the optimal learning rate change as the network increases in size?

Answer:

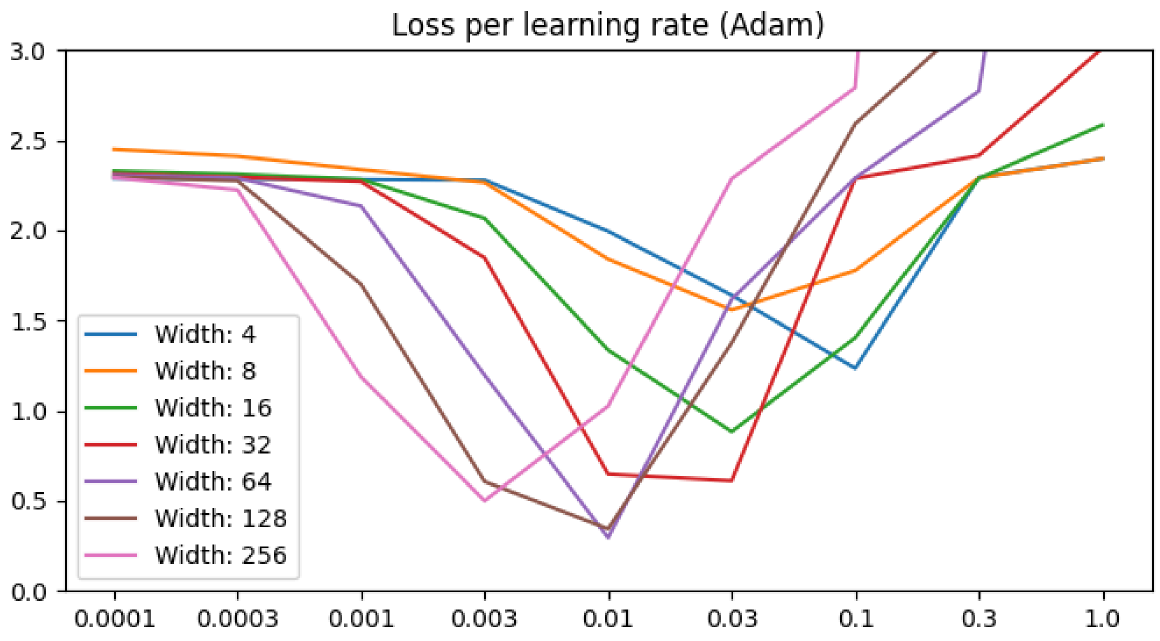
In the second cell, we will instead use the muP optimizer you implemented. How does the optimal learning rate work now? You should aim to show that there is a single global learning rate that works on a majority of widths. The 256-width network should achieve a loss of 0.5, comparable to Adam.

Answer:

In [17]:
```
valid_idx = np.random.randint(0, len(train_images), size=64)
valid_images = train_images[valid_idx]
valid_labels = train_labels[valid_idx]
valid_images, valid_labels = valid_images.to(device), valid_labels.to(device)


def train_with_lr(hiddens=[64, 64, 64], optimizer=SimpleAdam, lr=0.01):
    torch.manual_seed(4)
    np.random.seed(4)
    model = MLP(hidden_sizes=hiddens).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optimizer(model.parameters(), lr=lr)
    losses = []

    for i in range(100):
        batch_idx = np.random.randint(0, len(train_images), size=64)

        images_batch = train_images[batch_idx]
        labels_batch = train_labels[batch_idx]
        images_batch, labels_batch = images_batch.to(device), labels_batch.to(de

        optimizer.zero_grad()
        outputs, _ = model(images_batch)
        loss = criterion(outputs, labels_batch)
        loss.backward()
        optimizer.step()

        with torch.no_grad():
            outputs_valid, _ = model(valid_images)
            valid_losses = criterion(outputs_valid, valid_labels)
            losses.append(valid_losses.item())
    return np.mean(np.array(losses)[-5:])
```

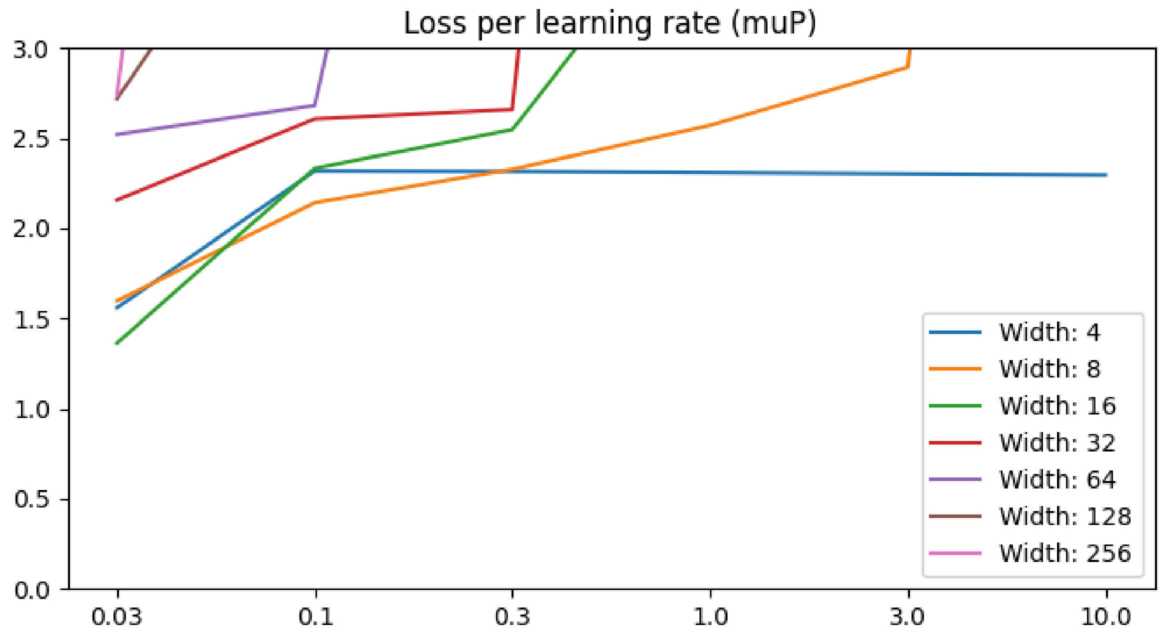
In [18]:
```
all_widths = [4, 8, 16, 32, 64, 128, 256]
all_lrs = [0.0001, 0.0003, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1.0]
adam_results = np.zeros((len(all_widths), len(all_lrs)))
for wi, width in enumerate(all_widths):
    for lri, lr in enumerate(all_lrs):
        loss = train_with_lr(hiddens=[width, width, width], lr=lr)
        adam_results[wi, lri] = loss

fig, axs = plt.subplots(1, figsize=(8, 4))
axs.set_title(f'Loss per learning rate (Adam)')
for wi, width in enumerate(all_widths):
    axs.plot(np.arange(len(all_lrs)), adam_results[wi], label=f'Width: {width}')
axs.set_xticks(np.arange(len(all_lrs)))
axs.set_xticklabels(all_lrs)
axs.set_ylim(bottom=0, top=3)
axs.legend()
fig.show()
```

```
C:\Users\gutia\AppData\Local\Temp\ipykernel_27112\3251529947.py:17: UserWarning: 
FigureCanvasAgg is non-interactive, and thus cannot be shown
  fig.show()
```



In [19]:
```
all_widths = [4, 8, 16, 32, 64, 128, 256]
all_lrs = [0.03, 0.1, 0.3, 1.0, 3.0, 10.0]
mup_results = np.zeros((len(all_widths), len(all_lrs)))
for wi, width in enumerate(all_widths):
    for lri, lr in enumerate(all_lrs):
        loss = train_with_lr(hiddens=[width, width, width], lr=lr, optimizer=Sim
        mup_results[wi, lri] = loss

fig, axs = plt.subplots(1, figsize=(8, 4))
axs.set_title(f'Loss per learning rate (muP)')
for wi, width in enumerate(all_widths):
    axs.plot(np.arange(len(all_lrs)), mup_results[wi], label=f'Width: {width}')
axs.set_xticks(np.arange(len(all_lrs)))
axs.set_xticklabels(all_lrs)
axs.set_ylim(bottom=0, top=3)
axs.legend()
fig.show()
```

```
C:\Users\gutia\AppData\Local\Temp\ipykernel_27112\475151.py:17: UserWarning: Figu
reCanvasAgg is non-interactive, and thus cannot be shown
  fig.show()
```



e. Shampoo and Orthogonalization

In lecture, we discussed a simplified version of the Shampoo update, which can be viewed as *orthogonalizing* the update to a dense layer. In the following code block, implement this simplified Shampoo update:

$$momentum \to U\Sigma V^T. \qquad update = UV^T.$$

Feel free to use linear algebra functions such as `torch.linalg.svd`.

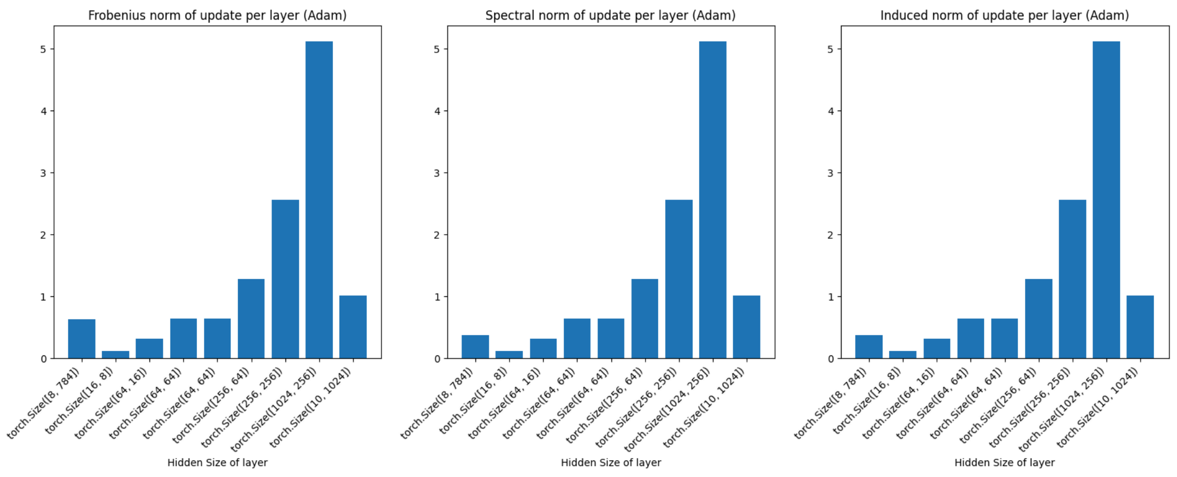
In [22]:
```
from torch.optim.optimizer import Optimizer
from typing import Any
class SimpleShampoo(Optimizer):
    def __init__(
        self,
        params: Any,
        lr: float = 1e-1,
        b1: float = 0.9,
    ):
        defaults = dict(lr=lr, b1=b1)
        super(SimpleShampoo, self).__init__(params, defaults)

    @torch.no_grad()
    def step(self):
        for group in self.param_groups:
            for p in group['params']:
                grad = p.grad.data

                state = self.state[p]
                if len(state) == 0: # Initialization
                    state["step"] = torch.tensor(0.0)
                    state['momentum'] = torch.zeros_like(p)

                state['step'] += 1
                m = state['momentum']
                m.lerp_(grad, 1-group["b1"])

                ############ TODO
                if len(m.shape) == 1:
                    u = m # Ignore biases for this question, it's not important.
                else:
                    # Implementation for Shampoo optimizer for matrix parameters
                    # 1. Update covariance matrices
                    for dim in range(len(m.shape)):
                        # Expand dimensions to compute outer product along each
                        expanded = m.unsqueeze(dim)
                        # Compute outer product and update covariance matrix
                        if dim == 0:
                            # For first dimension (rows)
                            outer = torch.bmm(expanded, expanded.transpose(1, 2)
                        else:
                            # For second dimension (columns)
                            outer = torch.bmm(expanded.transpose(1, 2), expanded
                        state['precond'][dim].lerp_(outer, 1-group["b1"])

                    # 2. Compute inverse square roots of covariance matrices
                    # Adding small epsilon for numerical stability
                    epsilon = 1e-6
                    precond_inv_sqrt = []
                    for i in range(len(state['precond'])):
                        # Compute eigenvalue decomposition
                        cov = state['precond'][i] + epsilon * torch.eye(state['p
                        eigenvalues, eigenvectors = torch.linalg.eigh(cov)
                        # Compute inverse square root
                        inv_sqrt_eigenvalues = 1.0 / torch.sqrt(eigenvalues)
                        inv_sqrt = eigenvectors @ torch.diag(inv_sqrt_eigenvalue
                        precond_inv_sqrt.append(inv_sqrt)

                    # 3. Apply preconditioning to get the update direction
                    # For 2D tensor, we apply preconditioning along both dimensi
                    u = torch.einsum('ij,jk->ik', precond_inv_sqrt[0], m)
                    u = torch.einsum('ij,jk->ik', u, precond_inv_sqrt[1])
                #############
                p.add_(u, alpha=-group['lr'])
        return None
```

Now, we will examine the relationship between the Frobenius norm and the Spectral norm for Adam vs. Shampoo. Plot these norms using your code from part c. What relationship do you see? Can you come up for a reason why this makes sense?

Answer:

Bonus: How should we scale the Shampoo update so the *induced RMS-RMS norm* is equal? Implement this change.

In [23]:
```
train_one_step_matrices(optimizer=SimpleAdam)
train_one_step_matrices(optimizer=SimpleShampoo, label="Shampoo")
```



```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[23], line 2
      1 train_one_step_matrices(optimizer=SimpleAdam)
----> 2 train_one_step_matrices(optimizer=SimpleShampoo, label="Shampoo")

Cell In[6], line 19, in train_one_step_matrices(mlp, hiddens, optimizer, label, l
r)
     17     loss = criterion(outputs, labels_batch)
     18     loss.backward()
---> 19     optimizer.step()
     21 new_params = [p.detach().clone() for p in model.parameters()]
     22 delta_params = [new_p - old_p for new_p, old_p in zip(new_params, old_par
ams)]

File d:\Anaconda\envs\cs182_env\lib\site-packages\torch\optim\optimizer.py:485, i
n Optimizer.profile_hook_step.<locals>.wrapper(*args, **kwargs)
    480         else:
    481             raise RuntimeError(
    482                 f"{func} must return None or a tuple of (new_args, new_kw
args), but got {result}."
    483             )
--> 485 out = func(*args, **kwargs)
    486 self._optimizer_step_code()
    488 # call optimizer step post hooks

File d:\Anaconda\envs\cs182_env\lib\site-packages\torch\utils\_contextlib.py:116,
in context_decorator.<locals>.decorate_context(*args, **kwargs)
    113 @functools.wraps(func)
    114 def decorate_context(*args, **kwargs):
    115     with ctx_factory():
--> 116         return func(*args, **kwargs)

Cell In[22], line 44, in SimpleShampoo.step(self)
     41     else:
     42         # For second dimension (columns)
     43         outer = torch.bmm(expanded.transpose(1, 2), expanded).mean(0)
---> 44     state['precond'][dim].lerp_(outer, 1-group["b1"])
     46 # 2. Compute inverse square roots of covariance matrices
     47 # Adding small epsilon for numerical stability
     48 epsilon = 1e-6

KeyError: 'precond'
```