hw 8

Chen Yuanteng

3039725444

1. Backprop through a Simple RNN

(a). $p = u_1$, $q = w \cdot u_1$ $r = u_2 + q = u_2 + w \cdot u_1$

$s = w \cdot r = w \cdot u_2 + w^2 \cdot u_1$

$$\begin{cases} t = s + u_3 = u_3 + w \cdot u_2 + w^2 \cdot u_1 \\ y = w \cdot t = w \cdot u_3 + w^2 \cdot u_2 + w^3 u_1 \end{cases}$$

(b). $\frac{dy}{dw} = u_3 + 2 \cdot w \cdot u_2 + 3w^2 \cdot u_1$

(c) $\frac{\partial y}{\partial t} = w$; $\frac{\partial y}{\partial s} = w$; $\frac{\partial y}{\partial r} = w^2$; $\frac{\partial y}{\partial q} = \frac{\partial y}{\partial r} = w^2$

$\frac{\partial y}{\partial p} = \frac{\partial y}{\partial q} \frac{\partial q}{\partial u_1} \cdot \frac{\partial u_1}{\partial p} = w^2 \cdot w \cdot 1 = w^3$

(d):

$$\frac{dy}{dw} = t \frac{\partial y}{\partial y} + r \cdot \frac{\partial y}{\partial s} + p \cdot \frac{\partial y}{\partial q}$$

$= u_3 + w \cdot u_2 + w^2 u_1 + (u_2 + w \cdot u_1) \cdot w + u_1 \cdot w^2$

$= 3w^2 u_1 + 2w u_2 + u_3$

5. self-supervised Linear Autoencoders

(a). (i) 2 layers (1 for encoder and 1 for decoder)

(ii) use nn.MSELoss as we hope for each vector can be close to its reconstruction.

(iii)
Weight-Decay + SGD-optimizer

(b). not sure

7. Homework Process and Study Group

(a). GPT. CSDN

(b) None

(c) two days

Introduction

In this notebook, we'll implement simple RNNs and LSTMs, then explore how gradients flow through these different networks.

This notebook does not require a Colab GPU. If it's enabled, you can turn it off through Runtime -> Change runtime type. (This will make it more likely for you to get Colab GPU access later in the REAL_RNN_LSTM.ipynb problem.)

Imports ¶

Note: the ipympl installation will require you to restart the colab runtime.

In [ ]:
```
! pip install ipympl
```

In [1]:
```
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
```

In [2]:
```
import copy

# If you are not using colab you can delete these two lines
#from google.colab import output
#output.enable_custom_widget_manager()

import torch as th
from torch import nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interactive, widgets, Layout
```

1.A: implementing a RNN layer

Consider using Pytorch's nn.Linear (https://pytorch.org/docs/stable/generated/torch.nn.Linear.html#torch.nn.Linear). You can implement this with either one Linear layer or two. If you use two, remember that you only need to include a bias term for one of the linear layers.

In [3]:
```
class RNNLayer(nn.Module):
  def __init__(self, input_size, hidden_size, nonlinearity=th.tanh):
    """
    Initialize a single RNN layer.

    Inputs:
    - input_size: Data input feature dimension
    - hidden_size: RNN hidden state size (also the output feature dimension)
    - nonlinearity: Nonlinearity applied to the rnn output
    """
    super().__init__()
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.nonlinearity = nonlinearity
    ##############################################################################
    # TODO: Initialize any parameters your class needs.
    # ht = σ (W_h * h_t-1 + W_x * X_t + bias)
    self.mixed_w = nn.Linear(input_size + hidden_size, hidden_size, bias = True)
    ##############################################################################

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

  def forward(self, x):
    """
    RNN forward pass

    Inputs:
    - x: input tensor (B, seq_len, input_size)

    Returns:
    - all_h: tensor of size (B, seq_len, hidden_size) containing hidden states
             produced for each timestep
    - last_h: hidden state from the last timestep (B, hidden_size)
    """
    h_list = []  # List to store the hidden states [h_1, ... h_T]
    ##############################################################################
    # TODO: Implement the RNN forward step                                       #
    # 1. Initialize h0 with zeros                                                #
    # 2. Roll out the RNN over the sequence, storing hidden states in h_list     #
    # 3. Return the appropriate outputs                                          #
    ##############################################################################

    batch_size, seq_len = x.shape[:2]

    begin_pad = th.zeros((batch_size, self.hidden_size)).float()
    h_i = begin_pad

    for i in range(seq_len):
      x_i = x[:, i]
      inputs = th.cat([x_i, h_i], dim=1)
      h_i = self.nonlinearity(self.mixed_w(inputs))
      h_list.append(h_i)

    last_h = h_i

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

```
    # h_list should now contain all hidden states, each of size (B, hidden_size)
    # We will store the hidden states so we can analyze their gradients later
    self.store_h_for_grad(h_list)

    print("batch_size: ", batch_size)
    print("seq_len: ", seq_len)
    print("hidden_size: ", self.hidden_size)

    print(len(h_list), h_list[0].shape)
    all_h = th.stack(h_list, dim=1)
    print(all_h.shape)
    return all_h, last_h

  def store_h_for_grad(self, h_list):
    """
    Store input list and allow gradient computation for all list elements
    """
    for h in h_list:
      h.retain_grad()
    self.h_list = h_list
```

Test Cases

If your implementation is correct, you should expect to see errors of less than 1e-4.

In [4]:

```python
rnn = RNNLayer(1, 1)
# Overwrite initial parameters with fixed values.
# Should give deterministic results even with different implementations.
rnn.load_state_dict({k: v * 0 + .1 for k, v in rnn.state_dict().items()})
data = th.ones((1, 1, 1))
expected_out = th.FloatTensor([[[0.1973753273487091]]])
all_h, last_h = rnn(data)
assert all_h.shape == expected_out.shape
assert th.all(th.isclose(all_h, last_h))
print(f'Expected: {expected_out.item()}, got: {last_h.item()}, max error: {th.max(th

rnn = RNNLayer(2, 3, nonlinearity=lambda x: x)  # no nonlinearity

num_params = sum(p.numel() for p in rnn.parameters())
assert num_params == 18, f'expected 18 parameters but found {num_params}'

rnn.load_state_dict({k: v * 0 - .1 for k, v in rnn.state_dict().items()})
data = th.FloatTensor([[[.1, .15], [.2, .25], [.3, .35], [.4, .45]], [[-.1, -1.5], [
expected_all_h = th.FloatTensor([[[-0.1250, -0.1250, -0.1250],
         [-0.1075, -0.1075, -0.1075],
         [-0.1328, -0.1328, -0.1328],
         [-0.1452, -0.1452, -0.1452]],

        [[ 0.0600,  0.0600,  0.0600],
         [ 0.1520,  0.1520,  0.1520],
         [ 0.2344,  0.2344,  0.2344],
         [-0.0853, -0.0853, -0.0853]]])
expected_last_h = th.FloatTensor([[-0.1452, -0.1452, -0.1452],
        [-0.0853, -0.0853, -0.0853]])
all_h, last_h = rnn(data)
assert all_h.shape == expected_all_h.shape
assert last_h.shape == expected_last_h.shape
print(f'Max error all_h: {th.max(th.abs(expected_all_h - all_h)).item()}')
print(f'Max error last_h: {th.max(th.abs(expected_last_h - last_h)).item()}')
```

```
batch_size:  1
seq_len:  1
hidden_size:  1
1 torch.Size([1, 1])
torch.Size([1, 1, 1])
Expected: 0.1973753273487091, got: 0.1973753273487091, max error: 0.0
batch_size:  2
seq_len:  4
hidden_size:  3
4 torch.Size([2, 3])
torch.Size([2, 4, 3])
Max error all_h: 4.999339580535889e-05
Max error last_h: 2.498924732208252e-05
```

1.B Implementing a RNN regression model.

In [5]:
```
class RecurrentRegressionModel(nn.Module):
  def __init__(self, recurrent_net, output_dim=1):
    """
    Initialize a simple RNN regression model

    Inputs:
    - recurrent_net: an RNN or LSTM (single or multi layer)
    - output_dim: feature dimension of the output
    """
    super().__init__()
    self.recurrent_net = recurrent_net
    self.output_dim = output_dim
    ################################################################################
    # TODO: Initialize any parameters you need                                     #
    # HINT: use recurrent_net.hidden_size to find the hidden state size            #
    ################################################################################

    # final_layer

    # input: (batch_size, seq_len, hidden_size)

    self.final_w = nn.Linear(self.recurrent_net.hidden_size, output_dim)

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

  def forward(self, x):
    """
    Forward pass

    Inputs:
    - x: input tensor (B, seq_len, input_size)

    Returns:
    - out: predictions of shape (B, seq_len, self.output_dim).
    - all_h: tensor of size (B, seq_len, hidden_size) containing hidden states
             produced for each timestep.
    """
    ################################################################################
    # TODO: Implement the forward step.                                            #
    ################################################################################

    all_h, last_h = self.recurrent_net(x)
    print("all_h shape: ", all_h.shape)

    out = self.final_w(all_h)

    # output size: (batch_size, seq_len, output_dim)
    print("out shape: ", out.shape)

    ################################################################################
    #                                 END OF YOUR CODE                             #
    ################################################################################
    return out, all_h
```

Tests

In [6]:
```
rnn = RecurrentRegressionModel(RNNLayer(2, 3), 4)

num_params = sum(p.numel() for p in rnn.parameters())
assert num_params == 34, f'expected 34 parameters but found {num_params}'

rnn.load_state_dict({k: v * 0 - .1 for k, v in rnn.state_dict().items()})
data = th.FloatTensor([[[.1, .15], [.2, .25], [.3, .35], [.4, .45]], [[-.1, -1.5], [
expected_preds = th.FloatTensor([[[-0.0627, -0.0627, -0.0627, -0.0627],
         [-0.0678, -0.0678, -0.0678, -0.0678],
         [-0.0604, -0.0604, -0.0604, -0.0604],
         [-0.0567, -0.0567, -0.0567, -0.0567]],

        [[-0.1180, -0.1180, -0.1180, -0.1180],
         [-0.1453, -0.1453, -0.1453, -0.1453],
         [-0.1692, -0.1692, -0.1692, -0.1692],
         [-0.0748, -0.0748, -0.0748, -0.0748]]])
expected_all_h = th.FloatTensor([[[-0.1244, -0.1244, -0.1244],
         [-0.1073, -0.1073, -0.1073],
         [-0.1320, -0.1320, -0.1320],
         [-0.1444, -0.1444, -0.1444]],

        [[ 0.0599,  0.0599,  0.0599],
         [ 0.1509,  0.1509,  0.1509],
         [ 0.2305,  0.2305,  0.2305],
         [-0.0840, -0.0840, -0.0840]]])
preds, all_h = rnn(data)
assert all_h.shape == expected_all_h.shape
assert preds.shape == expected_preds.shape
print(f'Max error all_h: {th.max(th.abs(expected_all_h - all_h)).item()}')
print(f'Max error last_h: {th.max(th.abs(expected_preds - preds)).item()}')
```

```
batch_size:  2
seq_len:  4
hidden_size:  3
4 torch.Size([2, 3])
torch.Size([2, 4, 3])
all_h shape:  torch.Size([2, 4, 3])
out shape:  torch.Size([2, 4, 4])
Max error all_h: 4.699826240539551e-05
Max error last_h: 4.312396049499512e-05
```

Problem 1.C: Dataset and loss function

1.C.i: Understanding the dataset (no implementation needed)

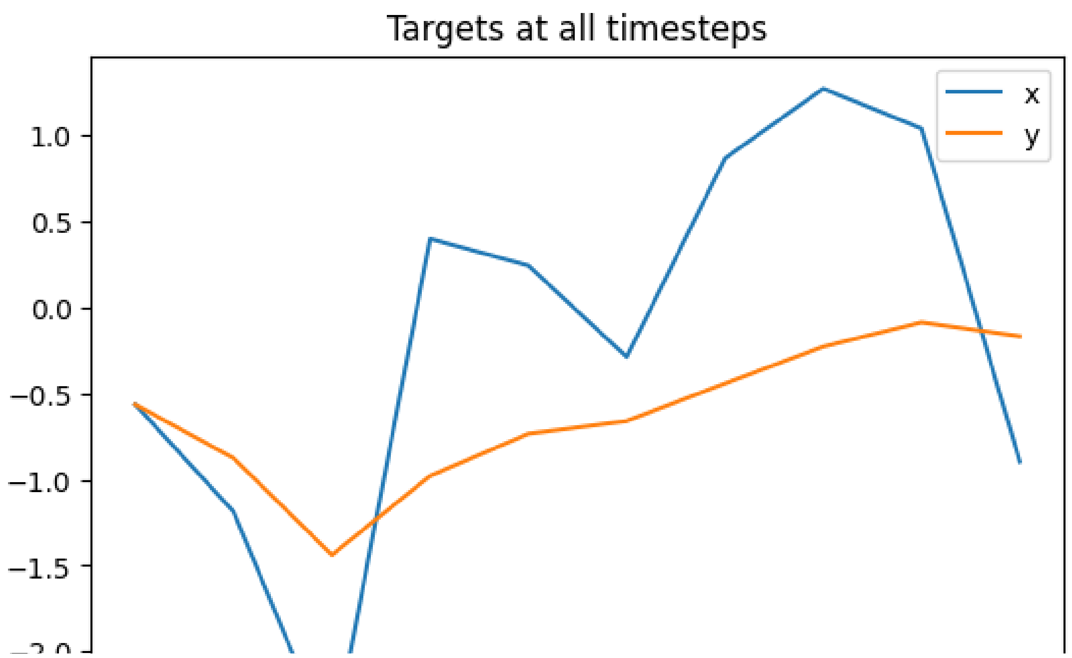
Inspect the code and plots below to visualize the dataset

In [7]:

```
def generate_batch(seq_len=10, batch_size=1):
  data = th.randn(size=(batch_size, seq_len, 1))
  sums = th.cumsum(data, dim=1)
  div = (th.arange(seq_len) + 1).unsqueeze(0).unsqueeze(2)
  target = sums / div
  return data, target
```

In [8]:

```
x, y = generate_batch(seq_len=10, batch_size=4)
for i in range(4):
  fig, ax1 = plt.subplots(1)
  ax1.plot(x[i, :, 0])
  ax1.plot(y[i, :, 0])
  ax1.legend(['x', 'y'])
  plt.title('Targets at all timesteps')
  plt.show()

for i in range(4):
  fig, ax1 = plt.subplots(1)
  ax1.plot(x[i, :, 0])
  ax1.plot(np.arange(10), [y[i, -1].item()] * 10)
  ax1.legend(['x', 'y'])
  plt.title('Predict only at the last timestep')
  plt.show()
```



1.C.ii Implement the loss function

In [9]:
```
def loss_fn(pred, y, last_timestep_only=False):
  """
  Inputs:
  - pred: model predictions of size (batch, seq_len, 1)
  - y: targets of size (batch, seq_len, 1)
  - last_timestep_only: boolean indicating whether to compute loss for all
    timesteps or only the lat

  Returns:
  - loss: scalar MSE loss between pred and true labels
  """
  ##############################################################################
  # TODO: implement the loss (HINT: look for pytorch's MSELoss function)       #
  ##############################################################################
  if last_timestep_only:
    pred = pred[:, -1]
    y = y[:, -1]

  loss_fn = nn.MSELoss()
  loss = loss_fn(pred, y)
  ##############################################################################
  #                               END OF YOUR CODE                             #
  ##############################################################################
  return loss
```

Tests

You should see errors < 1e-4

In [10]:
```
pred = th.FloatTensor([[.1, .2, .3], [.4, .5, .6]])
y = th.FloatTensor([[-1.1, -1.2, -1.3], [-1.4, -1.5, -1.6]])
loss_all = loss_fn(pred, y, last_timestep_only=False)
loss_last = loss_fn(pred, y, last_timestep_only=True)
assert loss_all.shape == loss_last.shape == th.Size([])
print(f'Max error loss_all: {th.abs(loss_all - th.tensor(3.0067)).item()}')
print(f'Max error loss_last: {th.abs(loss_last - th.tensor(3.7)).item()}')
```

```
Max error loss_all: 3.314018249511719e-05
Max error loss_last: 2.384185791015625e-07
```

1.D: Analyzing RNN Gradients

You do not need to understand the details of the GradientVisualizer class in order to complete this problem.

In [15]:
```
def biggest_eig_magnitude(matrix):
  """
  Inputs: a square matrix
  Returns: the scalar magnitude of the largest eigenvalue
  """
  h, w = matrix.shape
  assert h == w, f'Matrix has shape {matrix.shape}, but eigenvalues can only be com
  eigs = th.linalg.eigvals(matrix)
  eig_magnitude = eigs.abs()
  eigs_sorted = sorted([i.item() for i in eig_magnitude], reverse=True)
  first_eig_magnitude = eigs_sorted[0]
  return first_eig_magnitude

class GradientVisualizer:

  def __init__(self, rnn, last_timestep_only):
    """
    Inputs:
    - rnn: rnn module
    - last_timestep_only: boolean indicating whether to compute loss for all
      timesteps or only the lat

    Returns:
    - loss: scalar MSE loss between pred and true labels
    """


    self.rnn = rnn
    self.last_timestep_only = last_timestep_only
    self.model = RecurrentRegressionModel(rnn)
    self.original_weights = copy.deepcopy(rnn.state_dict())

    # Generate a single batch to be used repeatedly
    self.x, self.y = generate_batch(seq_len=10)
    print(f'Data point: x={np.round(self.x[0, :, 0].detach().cpu().numpy(), 2)}, y={

  def plot_visuals(self):
    """ Generate plots which will be updated in realtime."""
    fig, (ax1, ax2) = plt.subplots(1, 2)
    ax1.set_title('RNN Outputs')
    ax1.set_xlabel('Unroll Timestep')
    ax1.set_ylabel('Hidden State Norm')
    ax1.set_ylim(-1, 5)
    plt_1 = ax1.plot(np.arange(1, 11), np.zeros(10) + 1)  # placeholder vals
    plt_1 = plt_1[0]

    ax2.set_title('Gradients')
    ax2.set_xlabel('Unroll Timestep')
    ax2.set_ylabel('RNN dLoss/d a_t Gradient Magitude')
    ax2.set_ylim( (10**-6,1e5) )
    ax2.set_yscale('log')
    # X-axis labels are reversed since the gradient flow is from later layers to ear
    ax2.set_xticks(np.arange(10), np.arange(10, 0, -1))
    plt_2 = ax2.plot(np.arange(10), np.arange(10) + 1)  # placeholder vals
    plt_2 = plt_2[0]
    self.fig = fig
    self.plots = [plt_1, plt_2]
    return plt_1, plt_2, fig

  # Main update function for interactive plot
  def update_plots(self, weight_val=0, bias_val=0):
    # Scale the original RNN weights by a constant
```

```python
    w_dict = copy.deepcopy(self.original_weights)
    ################################################################################
    # TODO: Scale all W matrixes by weight_val, and all bias matrices by bias_val#
    # If you're using PyTorch nn.Linear layers, you don't need to modify the code#
    # provided, but if you're using custom layers, modify this block.           #
    ################################################################################
    for k in w_dict.keys():
      if 'weight' in k:
        w_dict[k][:] *= weight_val
      elif 'bias' in k:
        w_dict[k][:] *= bias_val
    ################################################################################
    #                                END OF YOUR CODE                              #
    ################################################################################
    self.rnn.load_state_dict(w_dict)

    # Don't compute for LSTMs, which don't have behavior dependent on a single eigen
    if isinstance(self.rnn, RNNLayer):
      ################################################################################
      # TODO: Set W = the weight which most affects exploding/vanishing gradients  #
      # Hint: Call module.weight or module.bias on the module you want to use      #
      # If you used a single Linear layer, slice a square matrix from it.          #
      ################################################################################

      # rnn.mixed_w = nn.Linear(input_size + hidden_size, hidden_size)
      # but in weight, shape of store is weight.t
      # so shape is (hidden_size, mixed_size)

      hidden_size, mixed_size = self.rnn.mixed_w.weight.shape
      # we want to get the W_h part (elim the W_x part)
      W = self.rnn.mixed_w.weight[:, -hidden_size:]
      ################################################################################
      #                                END OF YOUR CODE                              #
      ################################################################################
      biggest_eig = biggest_eig_magnitude(W)
      print(f' Biggest eigenvalue magnitude: {biggest_eig:.3}')

    # Run model
    pred, h = self.model(self.x)
    loss = loss_fn(pred, self.y, self.last_timestep_only)
    n_steps = len(h[0])

    plt_1, plt_2 = self.plots

    # Plot the hidden state magnitude
    max_h = th.linalg.norm(h[0], dim=-1).detach().cpu().numpy()
    print('Max H', ' '.join([f'{num:.3}' for num in max_h]))
    plt_1.set_data(np.arange(1, n_steps + 1), np.array(max_h))
    # Compute the gradient for the loss wrt the stored hidden states
    # Gradients are plotted backward since we go from later layers to earlier
    grads = [th.linalg.norm(num).item() for num in th.autograd.grad(loss, self.rnn.
    print('gradients d Loss/d h_t', ' '.join([f'{num:.3}' for num in grads]))
    # Add 1e-6 since it throws an error for gradients near 0
    plt_2.set_data(np.arange(n_steps), np.array(grads) + 1e-6)
    self.fig.canvas.draw_idle()

  def create_visualization(self):
    # Include sliders for relevant quantities
    self.plot_visuals()
    ip = interactive(self.update_plots,
                     weight_val=widgets.FloatSlider(value=0, min=-5, max=5, step=.05
```

```
                    bias_val=widgets.FloatSlider(value=0, min=-5, max=5, step=.05,
                    )
    return ip
```

Adjust the sliders rescale the weight and bias parameters in the RNN. Observe the effect on exploding and vanishing gradients.

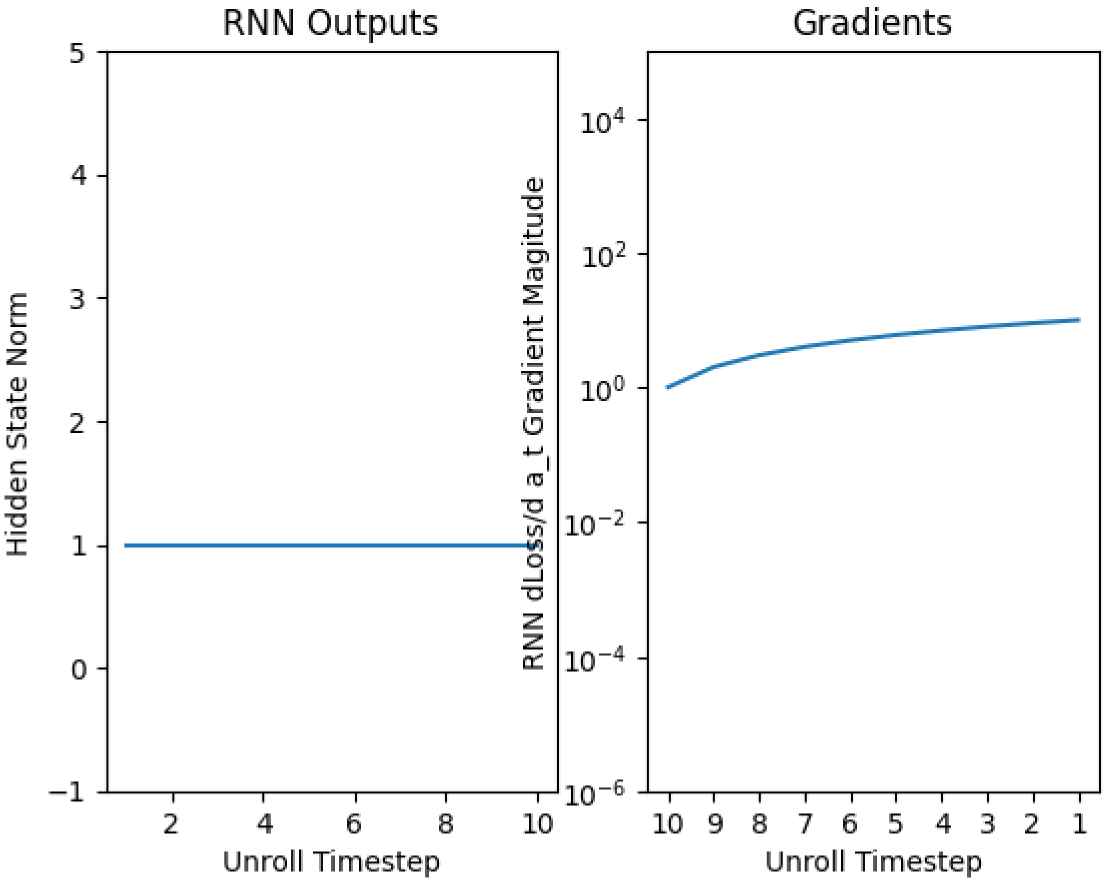
Parameters to try varying:

- nonlinearity
- last_target_only

（1）使用no nonlinearity时，当weight_scale过大或过小(负值)时，梯度都会爆炸
（2）使用relu时，当weight_scale过下(负值)时，梯度会爆炸
（3）使用tanh时，weight_scale过大或过小(负值)时，梯度都不会爆炸

In [21]:
```
hidden_size = 16
nonlinearity = lambda x: x  # options include lambda x: x (no nonlinearity), nn.fun
last_target_only = True
rnn = RNNLayer(1, hidden_size, nonlinearity=nonlinearity)
gv = GradientVisualizer(rnn, last_target_only)
gv.create_visualization()

# If for some reason the slider doesn't work for you, try calling gv.update_plots
# with various values for weight and bias
```

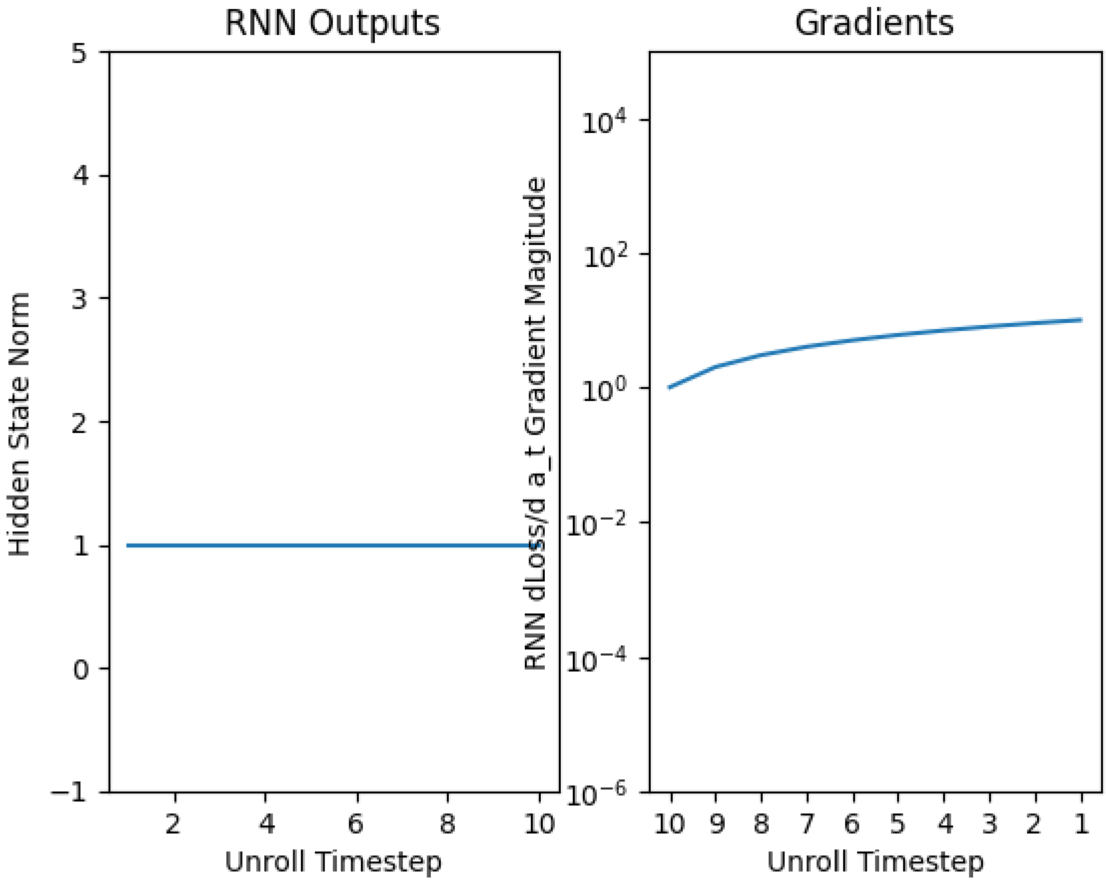
```
Data point: x=[ 1.51 -0.32  1.36  0.72 -0.19  0.13 -0.19 -0.43 -0.65 -0.95], y=
[1.51 0.59 0.85 0.82 0.61 0.53 0.43 0.32 0.21 0.1 ]
```



Out[21]:
```
interactive(children=(FloatSlider(value=0.0, description='weight_scale', layout=L
ayout(width='100%'), max=5.0,…
```

In [23]:
```
hidden_size = 16
nonlinearity = nn.functional.relu
last_target_only = True
rnn = RNNLayer(1, hidden_size, nonlinearity=nonlinearity)
gv = GradientVisualizer(rnn, last_target_only)
gv.create_visualization()

# If for some reason the slider doesn't work for you, try calling gv.update_plots
# with various values for weight and bias
```

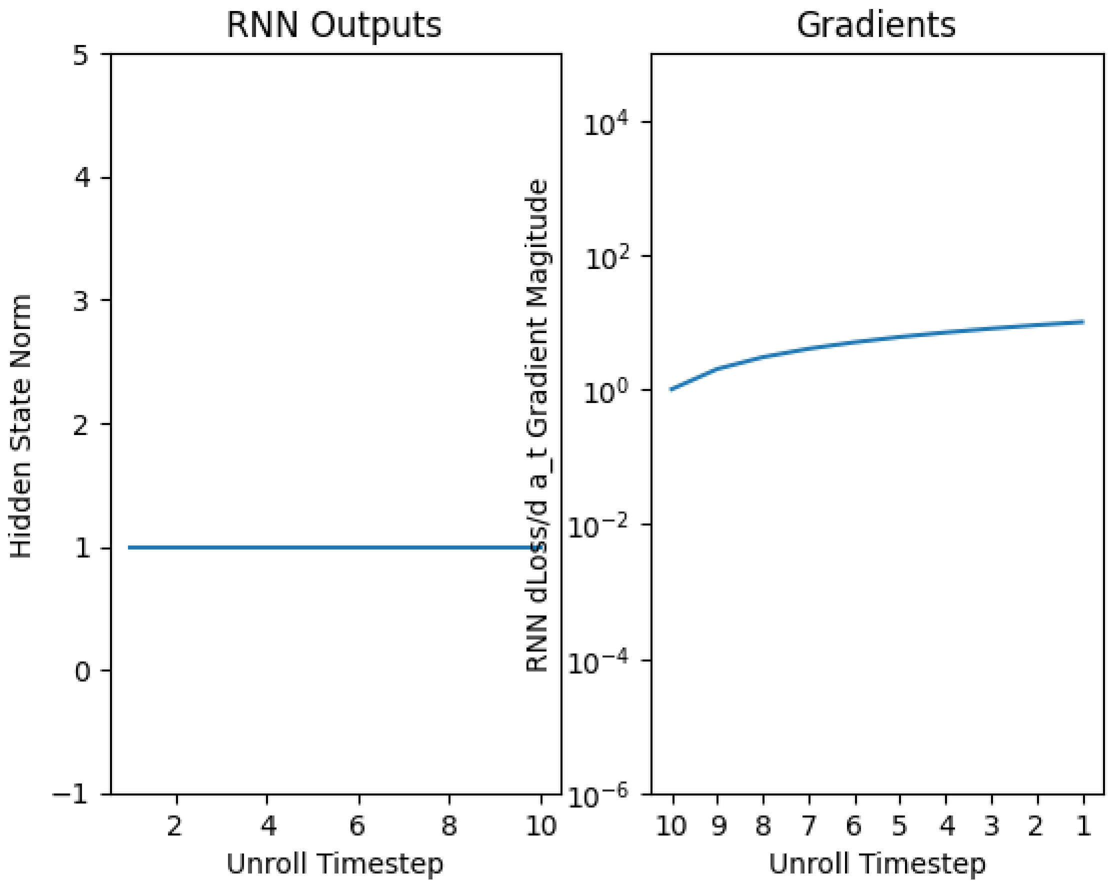
```
Data point: x=[ 1.92 -0.3  -0.28 -0.15 -0.75  1.45  0.52  1.14  2.27 -0.41], y=
[1.92 0.81 0.45 0.3  0.09 0.31 0.34 0.44 0.65 0.54]
```



Out[23]:
```
interactive(children=(FloatSlider(value=0.0, description='weight_scale', layout=L
ayout(width='100%'), max=5.0,…
```

In [27]:
```
hidden_size = 16
nonlinearity = th.tanh
last_target_only = True
rnn = RNNLayer(1, hidden_size, nonlinearity=nonlinearity)
gv = GradientVisualizer(rnn, last_target_only)
gv.create_visualization()

# If for some reason the slider doesn't work for you, try calling gv.update_plots
# with various values for weight and bias
```

```
Data point: x=[-1.68  1.45  2.    -0.13  0.19 -0.39 -0.31 -0.18 -0.24  0.08], y=[-1.68 -0.11  0.59  0.41  0.37  0.24  0.16  0.12  0.08  0.08]
```



Out[27]:
```
interactive(children=(FloatSlider(value=0.0, description='weight_scale', layout=Layout(width='100%'), max=5.0,…
```

Problem 1.H: Implementing a single-layer LSTM

Hint: consider creating parameters using Pytorch's [nn.Linear (https://pytorch.org/docs/stable/generated/torch.nn.Linear.html#torch.nn.Linear)](https://pytorch.org/docs/stable/generated/torch.nn.Linear.html#torch.nn.Linear). You can implement this with either one Linear layer or two for each equation. If you use two, remember that you only need to include a bias term for one of the linear layers.

In [35]:
```python
class LSTMLayer(nn.Module):
  def __init__(self, input_size, hidden_size):
    """
    Initialize a single LSTM layer.

    Inputs:
    - input_size: Data input feature dimension
    - hidden_size: RNN hidden state size (also the output feature dimension)
    """
    super().__init__()
    self.input_size = input_size
    self.hidden_size = hidden_size
    ##############################################################################
    # TODO: Initialize any parameters your class needs.                          #
    ##############################################################################

    self.w = nn.Linear(input_size + hidden_size, hidden_size * 4)

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

  def forward(self, x):
    """
    LSTM forward pass

    Inputs:
    - x: input tensor (B, seq_len, input_size)

    Returns:
    - all_h: tensor of size (B, seq_len, hidden_size) containing hidden states
             produced for each timestep
    - (h_last, c_last): hidden and cell states from the last timestep, each of
             size (B, hidden_size)
    """
    h_list = []
    ##############################################################################
    # TODO: Implement the LSTM forward step                                      #
    # 1. Initialize the hidden and cell states with zeros                        #
    # 2. Roll out the LSTM over the sequence, populating h_list along the way    #
    # 3. Return the appropriate outputs                                          #
    ##############################################################################

    # f(t)  = Sigmoid(linear1(input_size + hidden_size, hidden_size)(concat(X_t,h_t)
    # i(t)  = Sigmoid(linear2(input_size + hidden_size, hidden_size)(concat(X_t,h_t)
    # o(t)  = Sigmoid(linear3(input_size + hidden_size, hidden_size)(concat(X_t,h_t)
    # C(t)' = tanh(linear4((input_size + hidden_size, hidden_size)(concat(X_t,h_t))
    # C(t)  = f(t) * C(t-1) + i(t) * C(t)'
    # h(t)  = tanh(C(t)) * o(t)

    batch_size, seq_len = x.shape[:2]
    hs = self.hidden_size
    h_i = th.zeros((batch_size, hs)).float()
    c_i = th.zeros((batch_size, hs)).float()

    for i in range(seq_len):
      X_i = x[:, i]
      inputs = th.cat([X_i, h_i], dim = 1)
      outputs = self.w(inputs)
      #print(outputs.shape)
```

```
      f_t = nn.Sigmoid()(outputs[:, :hs])
      i_t = nn.Sigmoid()(outputs[:, hs:2*hs])
      o_t = nn.Sigmoid()(outputs[:, 2*hs:3*hs])
      c_i_hat = th.tanh(outputs[:, 3*hs:4*hs])

      c_i = f_t * c_i + i_t * c_i_hat
      h_i = th.tanh(c_i) * o_t

      h_list.append(h_i)

    h_last = h_i
    c_last = c_i

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

    # h_list should now contain all hidden states, each of size (B, hidden_size)
    # We will store the hidden states so we can analyze their gradients later
    self.store_h_for_grad(h_list)
    all_h = th.stack(h_list, dim=1)

    return all_h, (h_last, c_last)

  def store_h_for_grad(self, h_list):
    """
    Store input list and allow gradient computation for all list elements
    """
    for h in h_list:
      h.retain_grad()
    self.h_list = h_list
```

Test Cases

A correct implementation should have errors < 1e-4.

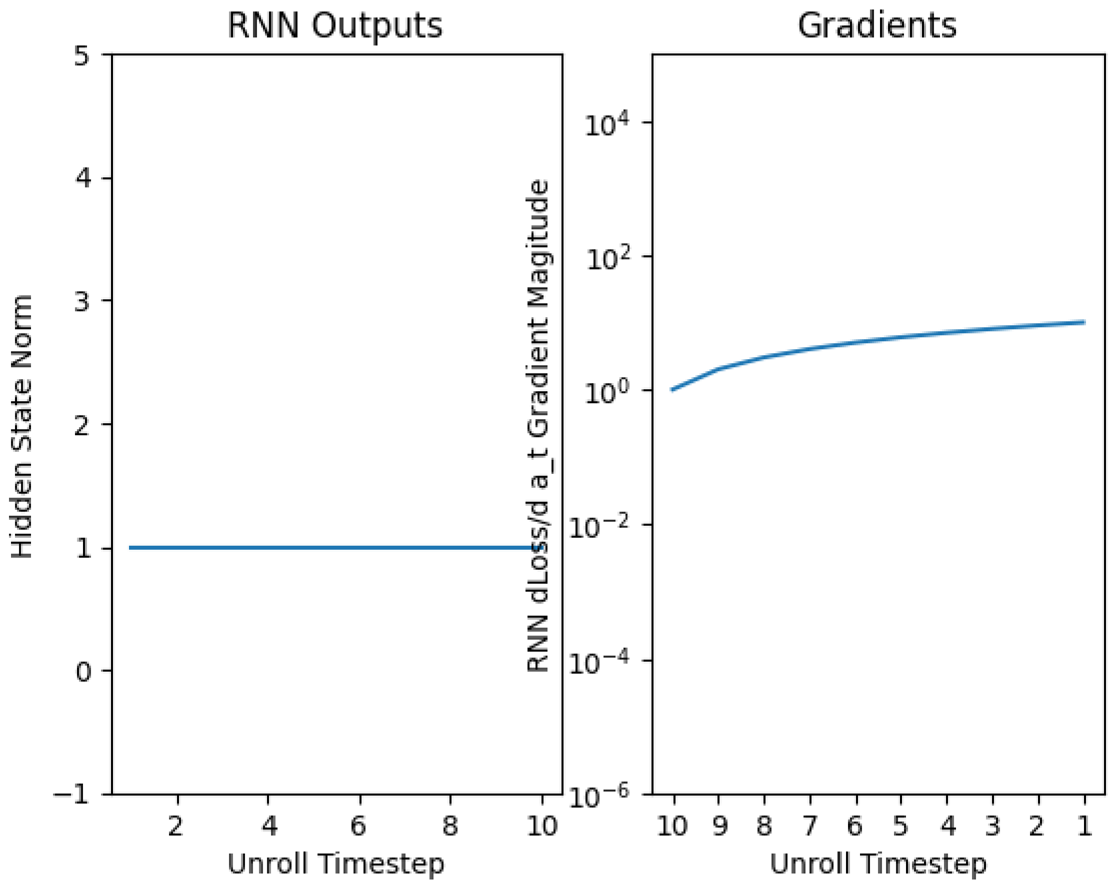
In [36]:
```
lstm = LSTMLayer(2, 3)
lstm.load_state_dict({k: v * 0 - .1 for k, v in lstm.state_dict().items()})
data = th.FloatTensor([[[.1, .15], [.2, .25], [.3, .35], [.4, .45]], [[-.1, -1.5], [
expected_all_h = th.FloatTensor([[[-0.0273, -0.0273, -0.0273],
         [-0.0420, -0.0420, -0.0420],
         [-0.0514, -0.0514, -0.0514],
         [-0.0583, -0.0583, -0.0583]],

        [[ 0.0159,  0.0159,  0.0159],
         [ 0.0568,  0.0568,  0.0568],
         [ 0.1142,  0.1142,  0.1142],
         [ 0.0369,  0.0369,  0.0369]]])
expected_last_h = th.FloatTensor([[-0.0583, -0.0583, -0.0583],
        [ 0.0369,  0.0369,  0.0369]])
expected_last_c = th.FloatTensor([[-0.1280, -0.1280, -0.1280],
        [ 0.0759,  0.0759,  0.0759]])
all_h, (last_h, last_c) = lstm(data)
assert all_h.shape == (2, 4, 3)
assert last_h.shape == last_c.shape == (2, 3)
print(f'Max error all_h: {th.max(th.abs(expected_all_h - all_h)).item()}')
print(f'Max error last_h: {th.max(th.abs(expected_last_h - last_h)).item()}')
print(f'Max error last_h: {th.max(th.abs(expected_last_c - last_c)).item()}')
```

```
Max error all_h: 4.8238784074783325e-05
Max error last_h: 4.8238784074783325e-05
Max error last_h: 8.024275302886963e-06
```

Problem 1.8b: Analyzing gradient flow through a single-layer LSTM

In [37]:
```
hidden_size = 3
last_target_only = True
rnn = LSTMLayer(1, hidden_size)
gv = GradientVisualizer(rnn, last_target_only)
gv.create_visualization()
```

```
Data point: x=[ 1.42  1.77 -1.32 -0.86 -0.73  0.46  1.88  1.6  -0.38 -0.01], y=
[1.42 1.59 0.62 0.25 0.06 0.12 0.37 0.53 0.43 0.38]
```



Out[37]:
```
interactive(children=(FloatSlider(value=0.0, description='weight_scale', layout=L
ayout(width='100%'), max=5.0,…
```

Problem 1.K: Making a multi-layer RNN and LSTM

1.K.i: Implementing multi-layer models

In [ ]:
```
class RNN(nn.Module):
  def __init__(self, input_size, hidden_size, num_layers):
    """
    Initialize a multilayer RNN

    Inputs:
    - input_size: Data input feature dimension
    - hidden_size: hidden state size (also the output feature dimension)
    - num_layers: number of layers
    """
    super().__init__()
    assert num_layers >= 1
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.num_layers = num_layers
    ##############################################################################
    # TODO: Initialize any parameters your class needs.                          #
    # Consider using nn.ModuleList or nn.ModuleDict.                             #
    ##############################################################################

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

  def forward(self, x):
    """
    Multilayer RNN forward pass

    Inputs:
    - x: input tensor (B, seq_len, input_size)

    Returns:
    - last_layer_h: tensor of size (B, seq_len, hidden_size) containing the
             outputs produced for each timestep from the last layer
    - last_step_h: all hidden states from the last step (num_layers, B, hidden_size)
    """
    ##############################################################################
    # TODO: Implement the RNN forward step                                       #
    ##############################################################################

    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################
    return last_layer_h, last_step_h


class LSTM(nn.Module):
  def __init__(self, input_size, hidden_size, num_layers):
    """
    Initialize a multilayer LSTM

    Inputs:
    - input_size: Data input feature dimension
    - hidden_size: hidden state size (also the output feature dimension)
    - num_layers: number of layers
    """
    super().__init__()
    assert num_layers >= 1
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.num_layers = num_layers
```

```
        ##############################################################################
        # TODO: Initialize any parameters your class needs.                          #
        # Consider using nn.ModuleList or nn.ModuleDict.                             #
        ##############################################################################

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

    def forward(self, x, hc0=None):
        """
        Multilayer LSTM forward pass

        Inputs:
        - x: input tensor (B, seq_len, input_size)

        Returns:
        - last_layer_h: tensor of size (B, seq_len, hidden_size) containing the
                 outputs produced for each timestep from the last layer
        - (last_step_h, last_step_c): all hidden and cell states from the last step
                size (num_layers, B, hidden_size)
        """
        ##############################################################################
        # TODO: Implement the LSTM forward step                                      #
        ##############################################################################
        ##############################################################################
        #                               END OF YOUR CODE                             #
        ##############################################################################
        return last_layer_h, (last_step_h, last_step_c)
```

Test Cases

In [ ]:
```
rnn = RNN(2, 3, 1)
rnn.load_state_dict({k: v * 0 - .1 for k, v in rnn.state_dict().items()})
data = th.FloatTensor([[[.1, .15], [.2, .25], [.3, .35], [.4, .45]], [[-.1, -1.5], [
expected_all_h = th.FloatTensor([[[-0.1244, -0.1244, -0.1244],
         [-0.1073, -0.1073, -0.1073],
         [-0.1320, -0.1320, -0.1320],
         [-0.1444, -0.1444, -0.1444]],

        [[ 0.0599,  0.0599,  0.0599],
         [ 0.1509,  0.1509,  0.1509],
         [ 0.2305,  0.2305,  0.2305],
         [-0.0840, -0.0840, -0.0840]]])
expected_last_h = th.FloatTensor([[[-0.1444, -0.1444, -0.1444],
         [-0.0840, -0.0840, -0.0840]]])
all_h, last_h = rnn(data)
assert all_h.shape == expected_all_h.shape
assert last_h.shape == expected_last_h.shape
print(f'Max error all_h: {th.max(th.abs(expected_all_h - all_h)).item()}')
print(f'Max error last_h: {th.max(th.abs(expected_last_h - last_h)).item()}')

rnn = RNN(2, 3, 2)
rnn.load_state_dict({k: v * 0 - .1 for k, v in rnn.state_dict().items()})
data = th.FloatTensor([[[.1, .15], [.2, .25], [.3, .35], [.4, .45]], [[-.1, -1.5], [
expected_all_h = th.FloatTensor([[[-0.0626, -0.0626, -0.0626],
         [-0.0490, -0.0490, -0.0490],
         [-0.0457, -0.0457, -0.0457],
         [-0.0430, -0.0430, -0.0430]],
        [[-0.1174, -0.1174, -0.1174],
         [-0.1096, -0.1096, -0.1096],
         [-0.1354, -0.1354, -0.1354],
         [-0.0342, -0.0342, -0.0342]]])
expected_last_h = th.FloatTensor([[[-0.1444, -0.1444, -0.1444],
         [-0.0840, -0.0840, -0.0840]],
        [[-0.0430, -0.0430, -0.0430],
         [-0.0342, -0.0342, -0.0342]]])
all_h, last_h = rnn(data)
assert all_h.shape == (2, 4, 3)
assert last_h.shape == (2, 2, 3)
print(f'Max error all_h: {th.max(th.abs(expected_all_h - all_h)).item()}')
print(f'Max error last_h: {th.max(th.abs(expected_last_h - last_h)).item()}')


lstm = LSTM(2, 3, 1)
lstm.load_state_dict({k: v * 0 - .1 for k, v in lstm.state_dict().items()})
data = th.FloatTensor([[[.1, .15], [.2, .25], [.3, .35], [.4, .45]], [[-.1, -1.5], [
expected_all_h = th.FloatTensor([[[-0.0273, -0.0273, -0.0273],
         [-0.0420, -0.0420, -0.0420],
         [-0.0514, -0.0514, -0.0514],
         [-0.0583, -0.0583, -0.0583]],

        [[ 0.0159,  0.0159,  0.0159],
         [ 0.0568,  0.0568,  0.0568],
         [ 0.1142,  0.1142,  0.1142],
         [ 0.0369,  0.0369,  0.0369]]])
expected_last_h = th.FloatTensor([[[-0.0583, -0.0583, -0.0583],
         [ 0.0369,  0.0369,  0.0369]]])
expected_last_c = th.FloatTensor([[[-0.1280, -0.1280, -0.1280],
         [ 0.0759,  0.0759,  0.0759]]])
all_h, (last_h, last_c) = lstm(data)
assert all_h.shape == (2, 4, 3)
assert last_h.shape == last_c.shape == (1, 2, 3)
```

```python
print(f'Max error all_h: {th.max(th.abs(expected_all_h - all_h)).item()}')
print(f'Max error last_h: {th.max(th.abs(expected_last_h - last_h)).item()}')
print(f'Max error last_c: {th.max(th.abs(expected_last_c - last_c)).item()}')


lstm = LSTM(2, 3, 3)
lstm.load_state_dict({k: v * 0 - .1 for k, v in lstm.state_dict().items()})
data = th.FloatTensor([[[.1, .15], [.2, .25], [.3, .35], [.4, .45]], [[-.1, -1.5], [
expected_all_h = th.FloatTensor([[[-0.0212, -0.0212, -0.0212],
         [-0.0296, -0.0296, -0.0296],
         [-0.0329, -0.0329, -0.0329],
         [-0.0343, -0.0343, -0.0343]],
        [[-0.0211, -0.0211, -0.0211],
         [-0.0291, -0.0291, -0.0291],
         [-0.0320, -0.0320, -0.0320],
         [-0.0332, -0.0332, -0.0332]]])
expected_last_h = th.FloatTensor([[[-0.0583, -0.0583, -0.0583],
         [ 0.0369,  0.0369,  0.0369]],
        [[-0.0320, -0.0320, -0.0320],
         [-0.0430, -0.0430, -0.0430]],
        [[-0.0343, -0.0343, -0.0343],
         [-0.0332, -0.0332, -0.0332]]])
expected_last_c = th.FloatTensor([[[-0.1280, -0.1280, -0.1280],
         [ 0.0759,  0.0759,  0.0759]],
        [[-0.0666, -0.0666, -0.0666],
         [-0.0907, -0.0907, -0.0907]],
        [[-0.0716, -0.0716, -0.0716],
         [-0.0693, -0.0693, -0.0693]]])
all_h, (last_h, last_c) = lstm(data)
assert all_h.shape == (2, 4, 3)
assert last_h.shape == last_c.shape == (3, 2, 3)

print(f'Max error all_h: {th.max(th.abs(expected_all_h - all_h)).item()}')
print(f'Max error last_h: {th.max(th.abs(expected_last_h - last_h)).item()}')
print(f'Max error last_c: {th.max(th.abs(expected_last_c - last_c)).item()}')
```

```
Max error all_h: 4.699826240539551e-05
Max error last_h: 4.123896360397339e-05
Max error all_h: 4.3526291847229004e-05
Max error last_h: 4.123896360397339e-05
Max error all_h: 4.8238784074783325e-05
Max error last_h: 4.8238784074783325e-05
Max error last_c: 8.024275302886963e-06
Max error all_h: 4.732981324195862e-05
Max error last_h: 4.8238784074783325e-05
Max error last_c: 4.2885541915893555e-05
```

1.K.ii: Training your model

In [ ]:
```
def train(model, optimizer, num_batches, last_timestep_only, seq_len=10, batch_size=
  model
  model.train()

  losses = []
  from tqdm import tqdm
  t = tqdm(range(0, num_batches))
  for i in t:
      data, labels = generate_batch(seq_len=seq_len, batch_size=batch_size)
      pred, h = model(data)
      loss = loss_fn(pred, labels, last_timestep_only)
      losses.append(loss.item())

      optimizer.zero_grad()
      loss.backward()
      optimizer.step()
      if i % 100 == 0:
          t.set_description(f"Batch: {i} Loss: {np.mean(losses[-10:])}")
  return losses
```

In [ ]:
```
def train_all(hidden_size, lr, num_batches, last_timestep_only):
  input_size = 1
  rnn_1_layer = RecurrentRegressionModel(RNN(input_size, hidden_size, 1))
  lstm_1_layer = RecurrentRegressionModel(LSTM(input_size, hidden_size, 1))
  rnn_2_layer = RecurrentRegressionModel(RNN(input_size, hidden_size, 2))
  lstm_2_layer = RecurrentRegressionModel(LSTM(input_size, hidden_size, 2))
  models = [rnn_1_layer, lstm_1_layer, rnn_2_layer, lstm_2_layer]
  model_names = ['rnn_1_layer', 'lstm_1_layer', 'rnn_2_layer', 'lstm_2_layer']

  losses = []
  for model in models:
    optimizer = optim.Adam(model.parameters(), lr=lr)
    loss = train(model, optimizer, num_batches, last_timestep_only)
    losses.append(loss)

  # visualize the results
  fig, ax1 = plt.subplots(1)
  for loss in losses:
    ax1.plot(loss)
  ax1.legend(model_names)
  plt.show()

  batch_size = 4
  x, y = generate_batch(seq_len=10, batch_size=batch_size)
  preds_list = [model(x)[0] for model in models]
  for i in range(batch_size):
    fig, ax1 = plt.subplots(1)
    ax1.plot(x[i, :, 0])
    if last_timestep_only:
      ax1.plot(np.arange(10), [y[i, -1].item()] * 10, 'bo')
    else:
      ax1.plot(y[i, :, 0], 'bo')
    for pred in preds_list:
      if last_timestep_only:
        ax1.plot(np.arange(10), [pred[i, -1, 0].detach().cpu().numpy()] * 10)
      else:
        ax1.plot(pred[i, :, 0].detach().cpu().numpy())
    ax1.legend(['x', 'y'] + model_names)
    plt.show()
  return models, losses
```

In [ ]:

```python
hidden_size = 32
lr = 1e-4
num_batches = 5000
last_timestep_only = False

th.manual_seed(0)
predict_all_models, predict_all_losses = train_all(hidden_size, lr, num_batches, las
last_timestep_only = True
predict_one_models, predict_one_losses = train_all(hidden_size, lr, num_batches, las
```

```
Batch: 4900 Loss: 0.0038075688527897: 100%|██████████| 5000/5000 [0
0:19<00:00, 254.08it/s]
Batch: 4900 Loss: 0.004596875933930278: 100%|██████████| 5000/5000
[00:29<00:00, 171.82it/s]
Batch: 4900 Loss: 0.0009564614854753017: 100%|██████████| 5000/5000
[00:19<00:00, 258.37it/s]
Batch: 4900 Loss: 0.0008792090928182005: 100%|██████████| 5000/5000
[00:54<00:00, 91.51it/s]
```

Figure

Autoencoders

In this notebook, you will explore various design choices for AutoEncoders, including pretraining models with unsupervised learning and evaluating the learned representations with a linear classifier. Specifically, we will examine three different architectures:

- Vanilla Autoencoder
- Denoising Autoencoder
- Masked Autoencoder

By the end of this assignment, you will have gained a deep understanding of these techniques and their potential applications in real-world scenarios.

Note: You have to run this notebook with a CUDA GPU. Otherwise, the training will be very very slow. For example, you can run it on a GPU instance on Google Colab.

In [2]:
```
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
```

In [3]:
```
#@title Import packages

import time
import json
import inspect
import random
import argparse
from typing import List

import numpy as np
import torch
import torchvision
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

from tqdm import tqdm
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import seaborn as sns
sns.set_style('whitegrid')

%load_ext autoreload
%autoreload 2

def _set_seed(seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)

TO_SAVE = {"time": time.time()}
```

##Datasets

###Synthetic Dataset

This is the definition for a synthetic dataset. The purpose of the dataset is to generate input data with a specified mean and covariance matrix, where the covariance is *high along a small fraction of the dimensions*. The class label of each example only depends on those high-variance dimensions.

In [4]:
```python
class SyntheticDataset:
    """
    Create a torch compatible dataset by sampling
    features from a multivariate normal distribution with
    specified mean and covariance matrix. In particular,
    the covariance is high along a small fraction of the directions.
    """
    def __init__(self,
                 input_size,
                 samples=10000,
                 splits=None,
                 num_high_var_dims=2,
                 var_scale=100,
                 batch_size=100,
                 eval_batch_size=200):
        """
        input_size: (int) size of inputs
        samples: (int) number of samples to generate
        splits: list(float) of splitting dataset for [#train, #valid, #test]
        num_high_var_dims : (int) #dimensions with scaled variance
        var_scale : (float)
        """
        train_split, valid_split, test_split = splits
        self.input_size = input_size
        self.samples = samples
        self.num_high_var_dims = num_high_var_dims
        self.var_scale = var_scale
        self.batch_size = batch_size
        self.eval_batch_size = eval_batch_size
        self.num_train_samples = int(samples * train_split)
        self.num_valid_samples = int(samples * valid_split)
        self.num_test_samples = int(samples * test_split)
        self._build()

    def _build(self):
        """
        Covariance is scaled along num_high_var_dims.
        Create torch compatible dataset.
        """
        self.mean = np.zeros(self.input_size)
        self.cov = np.eye(self.input_size)
        self.cov[:self.num_high_var_dims, :self.num_high_var_dims] *= self.var_scale
        self.X = np.random.multivariate_normal(self.mean, self.cov, self.samples)

        # generate random rotation matrix with SVD
        u, _, v = np.linalg.svd(np.random.randn(self.input_size, self.input_size))
        sample = self.X @ u

        # create classification labels that depend only on the high-variance dimensi
        target = self.X[:, :self.num_high_var_dims].sum(axis=1) > 0

        self.train_sample = torch.from_numpy(sample[:self.num_train_samples]).float(
        self.train_target = torch.from_numpy(target[:self.num_train_samples]).long()

        # create validation set
        valid_sample_end = self.num_train_samples+self.num_valid_samples
        self.valid_sample = torch.from_numpy(
            sample[self.num_train_samples:valid_sample_end]).float()
        self.valid_target = torch.from_numpy(
            target[self.num_train_samples:valid_sample_end]).long()
```

```
        # create test set
        self.test_sample = torch.from_numpy(sample[valid_sample_end:]).float()
        self.test_target = torch.from_numpy(target[valid_sample_end:]).long()

    def __len__(self):
        return self.samples

    def get_num_samples(self, split="train"):
        if split == "train":
            return self.num_train_samples
        elif split == "valid":
            return self.num_valid_samples
        elif split == "test":
            return self.num_test_samples

    def get_batch(self, batch_idx, split="train"):
        batch_size = (
            self.batch_size
            if split == "train"
            else self.eval_batch_size
        )
        start_idx = batch_idx * batch_size
        end_idx = start_idx + batch_size

        if split == "train":
            return self.train_sample[start_idx:end_idx], self.train_target[start_id
        elif split == "valid":
            return self.valid_sample[start_idx:end_idx], self.valid_target[start_id
        elif split == "test":
            return self.test_sample[start_idx:end_idx], self.test_target[start_idx:
```

###MNIST Dataset

The MNIST dataset is defined in this code snippet. It loads each image in the dataset as a flattened vector of pixels.

```
In  [5]: class MNIST:
    def __init__(self, batch_size, splits=None, shuffle=True):
        """
        Args:
          batch_size : number of samples per batch
          splits : [train_frac, valid_frac]
          shuffle : (bool)
        """
        # flatten the images
        self.transform = torchvision.transforms.Compose(
            [torchvision.transforms.ToTensor(),
             torchvision.transforms.Lambda(lambda x: x.view(-1))])

        self.batch_size = batch_size
        self.eval_batch_size = 200
        self.splits = splits
        self.shuffle = shuffle

        self._build()

    def _build(self):
        train_split, valid_split = self.splits
        trainset = torchvision.datasets.MNIST(
                root="data", train=True, download=True, transform=self.transform)
        num_samples = len(trainset)
        self.num_train_samples = int(train_split * num_samples)
        self.num_valid_samples = int(valid_split * num_samples)

        # create training set
        self.train_dataset = torch.utils.data.Subset(
            trainset, range(0, self.num_train_samples))
        self.train_loader = list(iter(torch.utils.data.DataLoader(
            self.train_dataset,
            batch_size=self.batch_size,
            shuffle=self.shuffle,
        )))

        # create validation set
        self.valid_dataset = torch.utils.data.Subset(
            trainset, range(self.num_train_samples, num_samples))
        self.valid_loader = list(iter(torch.utils.data.DataLoader(
            self.valid_dataset,
            batch_size=self.eval_batch_size,
            shuffle=self.shuffle,
        )))

        # create test set
        test_dataset = torchvision.datasets.MNIST(
            root="data", train=False, download=True, transform=self.transform
        )
        self.test_loader = list(iter(torch.utils.data.DataLoader(
            test_dataset,
            batch_size=self.eval_batch_size,
            shuffle=False,
        )))
        self.num_test_samples = len(test_dataset)

    def get_num_samples(self, split="train"):
        if split == "train":
            return self.num_train_samples
        elif split == "valid":
```

```
            return self.num_valid_samples
        elif split == "test":
            return self.num_test_samples

    def get_batch(self, idx, split="train"):
        if split == "train":
            return self.train_loader[idx]
        elif split == "valid":
            return self.valid_loader[idx]
        elif split == "test":
            return self.test_loader[idx]
```

Vanilla Autoencoder

In this section, you will be implementing a vanilla autoencoder, which comprises of an encoder and a decoder, both of which are fully connected neural networks. The input $x \in \mathbb{R}^d$ is mapped to a latent representation z by the encoder, which is then mapped back to x'. During training, the mean squared error between x and x' is minimized using the following formula:

$$\text{Loss} = \frac{1}{n}\sum_{i=1}^{n} \|x_{i,j} - x'_{i,j}\|^2$$

Here, n is the number of samples in the dataset, d is the dimensionality of each sample, $x_{i,j}$ is the j-th feature of the i-th sample, and $x'_{i,j}$ is the predicted value of the j-th feature of the i -th sample.

In [6]:

```
class Autoencoder(nn.Module):
    """
    Autoencoder defines a general class of NN architectures

              _____________                                           ___________
             |             |                                         |           |
    x --> |    ENCODER    | --> z (latent representation) --> |  DECODER  | --> x'
             |_____________|                                         |___________|

    The `Autoencoder` class is a neural network architecture consisting of an
    encoder and a decoder, each of which is a fully connected neural network.

    The input `x` of size `input_size` is mapped to a latent representation `z`
    by the encoder, which is then mapped back to x' by the decoder.

    The architecture is defined by a list of hidden layer sizes for the encoder
    and decoder. The encoder and decoder are symmetric. The class provides
    methods for encoding, decoding, and computing the loss (mean squared error)
    between `x` and `x'`. A training step can be performed by calling the
    `train_step` method with an input tensor `x` and an optimizer.
    """
    def __init__(self, input_size: int, hidden_sizes: List[int],
                 activation_cls: nn.Module = nn.ReLU):
        super().__init__()
        self.input_size = input_size
        self.hidden_sizes = hidden_sizes
        self.activation_cls = activation_cls
        self.encoder = self._build_encoder()
        self.decoder = self._build_decoder()

    def _build_encoder(self):
        layers = []
        prev_size = self.input_size
        for layer_id, size in enumerate(self.hidden_sizes):
            layers.append(nn.Linear(prev_size, size))
            if layer_id < len(self.hidden_sizes)-1:
                layers.append(self.activation_cls())
            prev_size = size
        return nn.Sequential(*layers)

    def _build_decoder(self):
        layers = []
        ########################################################################
        # TODO: Implement the code to construct the decoder. The decoder should
        #       be symmetric to the encoder.
        #
        # Hint: Refer to the `_build_encoder` method above
        ########################################################################
        prev_size = self.hidden_sizes[-1]
        for size in reversed(self.hidden_sizes[:-1]):
            layers.append(nn.Linear(prev_size, size))
            layers.append(self.activation_cls())
            prev_size = size

        layers.append(nn.Linear(prev_size, self.input_size))
        ########################################################################
        return nn.Sequential(*layers)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        ########################################################################
        # TODO: Implement the forward pass of the (vanilla) autoencoder
        #       according to the diagram and documents above
```

```python
        #       The return value should be `x'`
        ##########################################################################
        z = self.encoder(x)
        x_hat = self.decoder(z)
        return x_hat
        ##########################################################################

    def get_loss(self, x):
        x_hat = self(x)
        return self.loss(x, x_hat)

    def encode(self, x):
        return self.encoder(x)

    def decode(self, z):
        return self.decoder(z)

    def loss(self, x: torch.Tensor, x_hat: torch.Tensor) -> torch.Tensor:
        ##########################################################################
        # TODO: Implement the loss function
        ##########################################################################
        loss = F.mse_loss(x, x_hat, reduction="sum") / x.size(0)

        return loss
        ##########################################################################

    def train_step(self, x: torch.Tensor, optimizer) -> torch.Tensor:
        x_hat = self(x)
        loss = self.loss(x, x_hat)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        return loss
```

In [7]:
```
_set_seed(2017)
model = Autoencoder(7, [5, 4], nn.ReLU)
assert set(model.state_dict().keys()) == {
    'encoder.0.weight',
    'encoder.0.bias',
    'encoder.2.weight',
    'encoder.2.bias',
    'decoder.0.weight',
    'decoder.0.bias',
    'decoder.2.weight',
    'decoder.2.bias'
}
TO_SAVE["ae.0"] = sorted(list(model.state_dict().keys()))
_set_seed(2022)
x1 = torch.randn(2, 7)
x2 = torch.randn(2, 7)
assert torch.allclose(
    model(x1).view(-1)[7:11],
    torch.tensor([-0.10894767940044403, 0.417645782232228455, 0.21026797592639923, 0.
    rtol=1e-03
)
TO_SAVE["ae.1"] = model(x2).view(-1)[3:7].tolist()
loss1 = model.loss(x1, model(x1))
loss2 = model.loss(x2, model(x2))
assert np.allclose(loss1.item(), 10.69554328918457, rtol=1e-03)
TO_SAVE["ae.2"] = loss2.item()
loss1.backward()
assert torch.allclose(
    model.encoder[0].weight.grad.view(-1)[9:13],
    torch.tensor([0.026928527280688286, 0.10433877259492874, -0.023865919560194016,
    rtol=1e-03
)
TO_SAVE["ae.3"] = model.encoder[2].weight.grad.view(-1)[11:15].tolist()
```

Denoising Autoencoder

In this section, you will be implementing a denoising autoencoder, which inherits vanilla autoencoder you implemented before, but with an added noise reduction part. The input $x \in \mathbb{R}^d$ should be corrupted with Gaussian noise during training, and then fed to the encoder to obtain the latent representation z. The decoder then maps the latent representation back to the original, noise-free input x'. During training, the mean squared error between x and x' is minimized, similar to the vanilla autoencoder.

In [8]:
```
class DenoisingAutoencoder(Autoencoder):
    def __init__(self, input_size: int, hidden_sizes: List[int],
                 activation_cls: nn.Module = nn.ReLU, noise_std: float = 0.5):
        super().__init__(input_size, hidden_sizes, activation_cls)
        self.noise_std = noise_std

    def train_step(self, x: torch.Tensor, optimizer) -> torch.Tensor:
        ########################################################################
        # TODO: Implement training step of the denoising autoencoder.
        #
        # Hint: Add a zero-mean i.i.d. gaussian noise of a standard deviation of
        #       `noise_std` to the input of the encoder
        ########################################################################
        x_noisy = x + self.noise_std * torch.randn_like(x)
        x_hat = self(x_noisy)
        ########################################################################
        loss = self.loss(x, x_hat)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        return loss
```

In [9]:
```
_set_seed(2017)
model = DenoisingAutoencoder(7, [5, 4], nn.ReLU)
optimizer = optim.SGD(model.parameters(), lr=1.0)
_set_seed(2022)
x1 = torch.randn(2, 7)
model.train_step(x1, optimizer)
assert torch.allclose(
    model.encoder[0].weight.view(-1)[2:6],
    torch.tensor([-0.09830013662576675, -0.08394217491149902, 0.1265936940908432, 0.
    rtol=1e-03
)
TO_SAVE["dae"] = model.encoder[2].weight.view(-1)[3:7].tolist()
```

Masked Autoencoder

In this section, you will be implementing a masked autoencoder, which is similar to the vanilla autoencoder, but with an added masking feature. During training, the input $x \in \mathbb{R}^d$ should be masked with some binary mask, which zeros-out some random features in the input. The masked input is then fed to the encoder to obtain the latent representation z, and the decoder maps the latent representation back to the original input x'. During training, the mean squared error between the unmasked part of x and the corresponding part of x' is minimized.

In [10]:
```
class MaskedAutoencoder(Autoencoder):
    def __init__(self, input_size: int, hidden_sizes: List[int],
                 activation_cls: nn.Module = nn.ReLU, mask_prob: float = 0.25):
        super().__init__(input_size, hidden_sizes, activation_cls)
        self.mask_prob = mask_prob

    def train_step(self, x: torch.Tensor, optimizer) -> torch.Tensor:
        ########################################################################
        # TODO: Implement training step of the masked autoencoder.
        #
        # Hint: Generate a mask with i.i.d. probabilities of mask_prob for each
        #       entry, and apply it to the input of the encoder, setting to zero
        #       the entries where the mask is activated.
        ########################################################################
        mask = torch.rand(x.shape, device = x.device) > self.mask_prob
        x_masked = x * mask
        x_hat = self(x_masked)
        ########################################################################
        loss = self.loss(x, x_hat)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        return loss
```

In [11]:
```
_set_seed(2017)
model = MaskedAutoencoder(7, [5, 4], nn.ReLU)
optimizer = optim.SGD(model.parameters(), lr=1.0)
_set_seed(2022)
x1 = torch.randn(2, 7)
model.train_step(x1, optimizer)
assert torch.allclose(
    model.encoder[0].weight.view(-1)[2:6],
    torch.tensor([-0.09830013662576675, -0.22797375917434692, 0.004662647843360901,
    rtol=1e-03
)
TO_SAVE["mae"] = model.encoder[2].weight.view(-1)[3:7].tolist()
```

Training Autoencoders

In this section, you will learn how to train and evaluate autoencoders. After each training epoch, you will calculate the linear probe accuracy on the test split of your dataset.

To achieve this, you will first use your trained autoencoder to encode each example in the dataset x_i into its latent representation z_i. Then, you will use these latent representations z_i, along with their corresponding labels y_i, to train a simple linear classifier called a linear probe.

The linear probe accuracy is the classification accuracy of this linear classifier on the test split of the dataset. By calculating this accuracy, you can evaluate how well your autoencoder is able to capture the important features of the data and how useful those features are for downstream tasks like classification.

In [12]:
```python
class Experiment:
    def __init__(self, dataset, model: nn.Module,
                 batch_size: int, num_classes: int, lr: float,
                 probe_train_batch = "full", probe_train_epochs: int = 50):
        self.train_batch_size = batch_size
        self.eval_batch_size = 200
        self.dataset = dataset
        self.model = model.cuda()
        self.optimizer = optim.Adam(self.model.parameters(), lr=lr)
        self.num_classes = num_classes
        self.probe_train_batch = probe_train_batch
        self.probe_train_epochs = probe_train_epochs

    def train(self, num_epochs: int) -> dict:
        self.model.train()
        train_losses, valid_losses, probe_accs = [], [], []

        pbar = tqdm(range(num_epochs))
        num_batches = self.dataset.num_train_samples // self.train_batch_size

        with torch.no_grad():
            valid_loss = self.get_loss(split="valid")
        valid_losses.append(valid_loss)
        probe_accs.append(
            self.evaluate_w_linear_probe(self.model.hidden_sizes[-1]))
        for epoch in pbar:
            for batch_idx in range(num_batches):
                x, y = self.dataset.get_batch(batch_idx, split="train")
                x, y = x.cuda(), y.cuda()
                loss = self.model.train_step(x, self.optimizer)
                train_losses.append(loss.item())
                pbar.set_description(f"Epoch {epoch}, Loss {loss.item():.4f}")
            with torch.no_grad():
                valid_loss = self.get_loss(split="valid")
            valid_losses.append(valid_loss)
            probe_accs.append(
                self.evaluate_w_linear_probe(self.model.hidden_sizes[-1]))

        return {
            "train_losses": train_losses,
            "valid_losses": valid_losses,
            "valid_accs": probe_accs
        }

    def get_loss(self, split="train") -> float:
        """
        Compute the average loss of the model on a specified dataset split.

        Parameters:
        - split (str, optional): The dataset split to compute the loss on.

        Returns:
        The average loss of the model on the specified dataset split.
        """
        self.model.eval()
        num_samples = self.dataset.get_num_samples(split=split)
        num_batches = num_samples // self.eval_batch_size
        assert num_samples % self.eval_batch_size == 0
        losses = []
        for batch_idx in range(num_batches):
            x, y = self.dataset.get_batch(batch_idx, split=split)
```

```python
            x, y = x.cuda(), y.cuda()
            loss = self.model.get_loss(x)
            losses.append(loss.item())
        return np.mean(losses)

    def _get_model_accuracy(self, classifier: nn.Module, split="test") -> float:
        """
        Compute the accuracy of the model on a specified dataset split using a
        given linear classifier (a.k.a. a linear probe). This method is invoked
        by `eval_w_linear_probe` that is defined below.

        Parameters:
        - classifier (nn.Module): The linear classifier to use for computing the
          accuracy.
        - split (str, optional): The dataset split to compute the accuracy on.

        Returns:
        The accuracy of the model on the specified dataset split.
        """
        self.model.eval()
        num_samples, num_correct = 0, 0
        num_batches = self.dataset.num_test_samples // self.eval_batch_size
        assert num_samples % self.eval_batch_size == 0
        for batch_idx in range(num_batches):
            x, y = self.dataset.get_batch(batch_idx, split="test")
            ####################################################################
            # TODO: Implement the following code in the evaluation loop to
            #       calculate accuracy using the autoencoder and the given
            #       classifier
            ####################################################################

            x = x.cuda()
            y = y.cuda()

            z = self.model.encode(x)
            y_hat = classifier(z)
            preds = (y_hat.argmax(dim=1) == y).cpu().numpy()

            num_samples += len(preds)
            num_correct += np.sum(preds)
            ####################################################################
        return num_correct / num_samples * 100

    def evaluate_w_linear_probe(self, feats_dim) -> float:
        """
        Evaluate the model using a linear probe on a small subset of the labeled
        data.

        Parameters:

        - feats_dim (int): The number of features in the model's output.
        - num_epochs (int, optional): The number of epochs to train the linear
          probe for. Defaults to 10.

        Returns:
        The accuracy of the model computed using the linear probe.
        """
        self.model.eval()
        probe = nn.Linear(feats_dim, self.num_classes)
        probe.cuda()
        probe.train()
```

```python
        probe_opt = optim.Adam(probe.parameters(), lr=1e-3)

        if self.probe_train_batch == "full":
            num_batches = (
                self.dataset.get_num_samples(split="valid")
                // self.eval_batch_size
            )
        else:
            num_batches = self.probe_train_batch

        frozen_batch = []
        with torch.no_grad():
            for batch_idx in np.random.permutation(num_batches):
                x, y = self.dataset.get_batch(batch_idx, split="valid")
                x, y = x.cuda(), y.cuda()
                feat = self.model.encode(x)
                frozen_batch.append((feat.cpu(), y.cpu()))

        for epoch in range(self.probe_train_epochs):
            for feat, y in frozen_batch:
                feat, y = feat.cuda(), y.cuda()
                y_hat = probe(feat)
                loss = F.cross_entropy(y_hat, y)
                probe_opt.zero_grad()
                loss.backward()
                probe_opt.step()

        # Evaluate linear probe
        probe.eval()
        with torch.no_grad():
            accuracy = self._get_model_accuracy(classifier=probe)
        return accuracy
```

In [13]:
```python
_set_seed(2017)
exp = Experiment(SyntheticDataset(7, 800, [0.4, 0.1, 0.5], 2, 15, 10), Autoencoder(7
classifier = nn.Linear(4, 2).cuda()
assert np.allclose(exp._get_model_accuracy(classifier), 53.00)
_set_seed(2022)
exp = Experiment(SyntheticDataset(7, 800, [0.4, 0.1, 0.5], 2, 15, 10), Autoencoder(7
classifier = nn.Linear(4, 2).cuda()
TO_SAVE["exp"] = exp._get_model_accuracy(classifier)
```

Linear AutoEncoders on Synthetic Dataset

In this experiment, we aim to investigate the performance of linear autoencoders/DAEs/MAEs on a synthetic dataset where there are 20 significant dimensions. Specifically, we will train four different autoencoder architectures with bottleneck sizes of 5, 20, 100, and 500. We will evaluate their performance on the synthetic dataset and report the results.

In [14]:
```
MODELS = {
    "vanilla": Autoencoder,
    "denoise": DenoisingAutoencoder,
    "masking": MaskedAutoencoder,
}

# we repeat each experiment and report mean performance
NUM_REPEATS = 3

data_cfg = argparse.Namespace(
    input_dims=100,
    num_samples=20000,
    data_splits=[0.7, 0.2, 0.1],
    num_high_var_dims=20,
    var_scale=10,
    num_classes=2
)

hparams = argparse.Namespace(
    batch_size=100,
    num_epochs=10,
    hidden_dims=[0],  # placeholder
    activation="Identity",  # linear AE
    lr=5e-4
)

dataset = SyntheticDataset(
    data_cfg.input_dims,
    data_cfg.num_samples,
    data_cfg.data_splits,
    data_cfg.num_high_var_dims,
    data_cfg.var_scale,
    hparams.batch_size
)

# logging metrics
train_losses, valid_losses = {}, {}
accuracy = {}

# run experiment w/ different models
for model_idx, model_cls in MODELS.items():
    for hidden_dim in [5, 20, 100, 500]:
        hparams.hidden_dims = [hidden_dim]
        feats_dim = hparams.hidden_dims[-1]
        _train_loss, _valid_loss, _acc = [], [], []
        for expid in range(NUM_REPEATS):
            _set_seed(expid * 227)
            print("run : {},  model : {},  hidden_dim : {}".format(
                expid, model_idx, feats_dim))
            model = model_cls(
                data_cfg.input_dims,
                hparams.hidden_dims,
                activation_cls=getattr(nn, hparams.activation)
            )
            experiment = Experiment(
                dataset,
                model,
                batch_size=hparams.batch_size,
                num_classes=data_cfg.num_classes,
                lr=hparams.lr
            )
```

```
            _set_seed(1998 + expid * 227)
            train_stats = experiment.train(num_epochs=hparams.num_epochs)
            _train_loss.append(train_stats["train_losses"])
            _valid_loss.append(train_stats["valid_losses"])
            _acc.append(train_stats["valid_accs"])

        train_losses[(model_idx, feats_dim)] = _train_loss
        valid_losses[(model_idx, feats_dim)] = _valid_loss
        accuracy[(model_idx, feats_dim)] = _acc

TO_SAVE["train1"] = {
    "train_losses": {f"{k[0]}.{k[1]}": v for k, v in train_losses.items()},
    "valid_losses": {f"{k[0]}.{k[1]}": v for k, v in valid_losses.items()},
    "accuracy": {f"{k[0]}.{k[1]}": v for k, v in accuracy.items()}
}

# report accuracy
for model_idx, acc in accuracy.items():
    print("Model : {}, Avg Accuracy : {}".format(
        model_idx, np.array(acc).mean(axis=0)[-1]))
```

```
run : 0,  model : vanilla,  hidden_dim : 5

Epoch 9, Loss 229.0076: 100%|██████████| 10/10 [00:10<00:00,  1.09s/
it]

run : 1,  model : vanilla,  hidden_dim : 5

Epoch 9, Loss 228.9907: 100%|██████████| 10/10 [00:08<00:00,  1.13i
t/s]

run : 2,  model : vanilla,  hidden_dim : 5

Epoch 9, Loss 229.4635: 100%|██████████| 10/10 [00:09<00:00,  1.10i
t/s]

run : 0,  model : vanilla,  hidden_dim : 20

Epoch 9, Loss 79.7512: 100%|██████████| 10/10 [00:09<00:00,  1.09it/
s]

run : 1,  model : vanilla,  hidden_dim : 20
```

Visualization: Training Curves

You have saved all the training logs for the autoencoder models with different feature dimensions. In this section, you will implement a function to visualize the training curves and accuracy using linear probes. This visualization will help you compare the performance of autoencoder models of different hidden sizes.

To begin with, you need to visualize the training curves of the **vanilla** autoencoder with latent representations of different feature dimensions. You will need to complete the following code to draw two plots:

- The x-axis of both plots should represent the training epochs, while the y-axis of the first plot should display the validation loss (reconstruction error) and the y-axis of the second plot should display the linear probe accuracy.
- Both plots should be line plots with four curves, each representing a feature dimension of 5, 20, 100, and 1000, respectively.
- Each curve should have a major line and an area around the line:

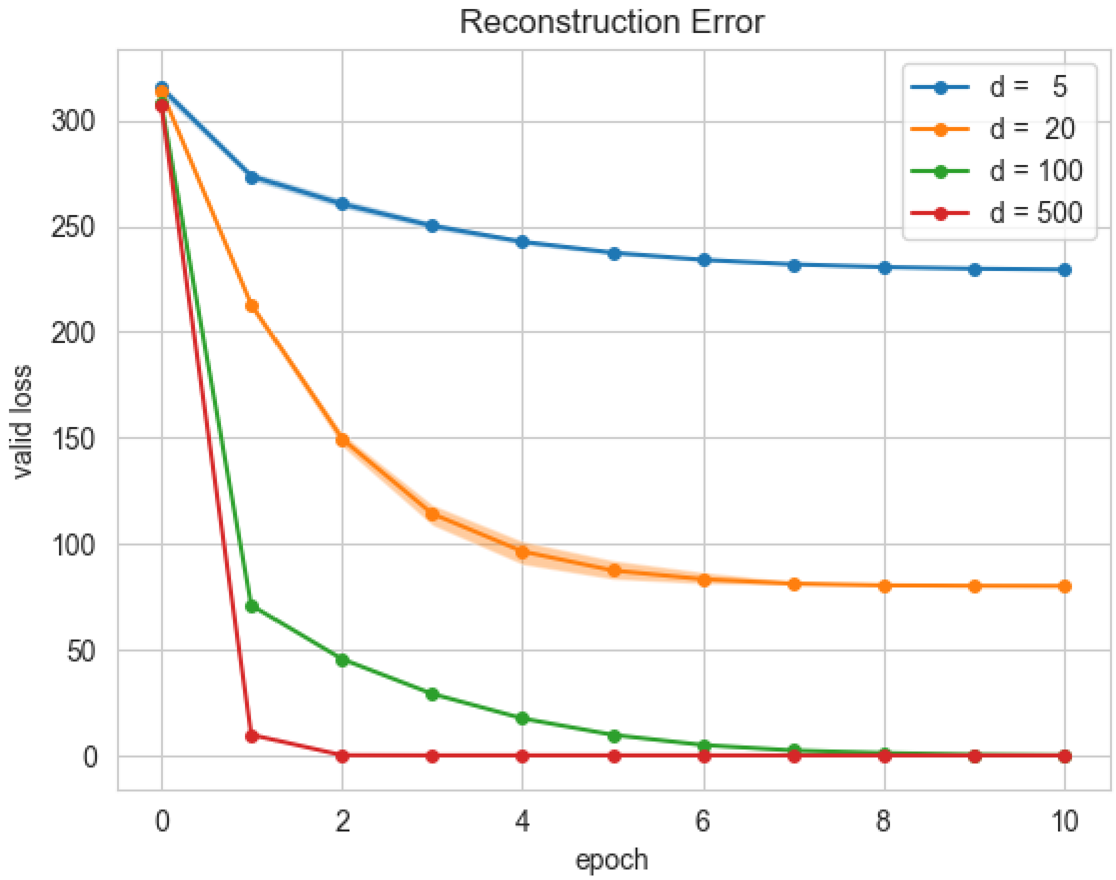
    - The major line should have one dot for each epoch showing the average validation loss/accuracy of that epoch over three runs.
    - The area should be filled between the minimum and maximum validation loss/accuracy of that epoch across the three runs.
    - The color of the line, dots, and area should be the same, with the area being translucent.
- Both plots should include axis labels (for the x and y axis), a legend, and a title.

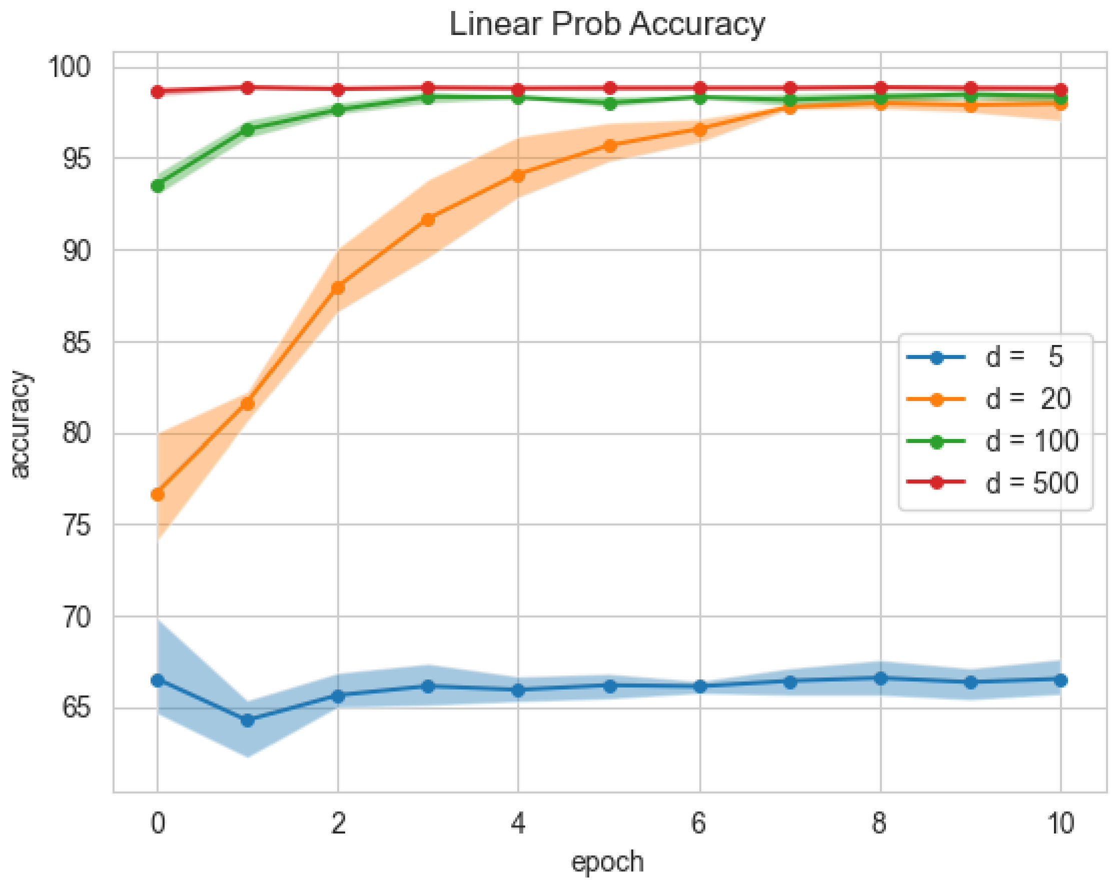
Ensure that your implementation accurately reflects the requirements outlined above.

Documents for reference:

- https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html (https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html)
- https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.fill_between.html (https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.fill_between.html)

In [15]:
```
def plot_single(values, feats_dim):
    ############################################################################
    # TODO: Implement the following code to draw a single curve with a filled
    #       area around it.
    #
    # Hint 1: `values` is a list of three lists, where each list corresponds to
    #         one run and each entry in the list corresponds to an epoch
    # Hint 2: `feats_dim` is useful for showing legends
    ############################################################################
    x = range(0, hparams.num_epochs + 1)
    y_avg = np.mean(values, axis=0)
    y_min = np.amin(values, axis=0)
    y_max = np.amax(values, axis=0)

    plt.plot(x, y_avg, marker = 'o', markersize = 4, label=f"d = {feats_dim:3d}")
    plt.fill_between(x, y_min, y_max, alpha=0.4)
    ############################################################################

# Visualize valid losses
for model_idx in valid_losses.keys():
    if model_idx[0] == 'vanilla':
        plot_single(valid_losses[model_idx], model_idx[1])
################################################################################
# TODO: Implement the following code to draw legends, axis labels, and the title
################################################################################
plt.legend()
plt.xlabel("epoch")
plt.ylabel("valid loss")
plt.title("Reconstruction Error")
################################################################################
plt.show()

# Visualize valid (linear probe) accuracy
for model_idx in accuracy.keys():
    if model_idx[0] == 'vanilla':
        plot_single(accuracy[model_idx], model_idx[1])
################################################################################
# TODO: Implement the following code to draw legends, axis labels, and the title
################################################################################
plt.legend()
plt.xlabel("epoch")
plt.ylabel("accuracy")
plt.title("Linear Prob Accuracy")
################################################################################
plt.show()


TO_SAVE["vis_fn"] = inspect.getsource(plot_single)
```





Question

Screenshot your visualization above and include it in your submission of the written assignment.

Question

In your written assignment submission, please answer the following question: **How does changing the latent representation size of the autoencoder affect the model's performance in terms of reconstruction accuracy and linear probe accuracy? Why?** Hint: each datapoint in the synthetic dataset has 100 dimensions, with 20 high-variance dimensions that affect the class label.

Nonlinear Dimensionality Reduction on MNIST

In the previous section, we observed that there is no advantage in terms of linear probe accuracy when we perform dimension reduction. The reason for this is that we use the entire *labeled* validation dataset to train the linear probe, rendering the use of autoencoders and self-supervised learning less useful in cases where we have ample labeled training data.

In this part, we will consider a different scenario where we have an abundance of *unlabeled* training data, but only a limited number of *labeled* examples. Specifically, we will train a non-linear autoencoder on the MNIST dataset using all images in the dataset, but only **200** labeled examples will be used to train the linear probe.

Your task is to train a non-linear autoencoder on the MNIST dataset. The objective is to achieve a few-shot linear probe accuracy of at least **79%** on the last epoch, averaged over two random runs. You may use any type of autoencoder that you have previously implemented, choose any latent representation sizes, and your grade will be evaluated on a linear scale, ranging from 0 to the maximum score.

The validation accuracy achieved on the last epoch should range from 70% to 79%. If the accuracy is less than 70%, you will receive a score of 0, and if it is greater than 79%, you will receive the full score for this autograding item.

In [20]:
```
# Do not change these
NUM_REPEATS = 2
input_dims = 28 * 28
num_classes = 10
data_splits = [0.9, 0.1]


################################################################################
# TODO: Set the hyperparameters
################################################################################
hparams = argparse.Namespace(
    batch_size=200,
    num_epochs=10,
    hidden_dims=[64, 32],
    activation="ReLU",
    lr=2e-4,
    noise_std=0.3,
    mask_prob=0.3
)
################################################################################


################################################################################
# TODO: Define a function to build the model. You are encouraged to experiment
#       with different types of autoencoders
################################################################################
def build_model():
    return Autoencoder(
        input_dims,
        hparams.hidden_dims,
        activation_cls=getattr(nn, hparams.activation)
    )
################################################################################

dataset = MNIST(hparams.batch_size, data_splits)

feats_dim = hparams.hidden_dims[-1]
train_loss, valid_loss, acc = [], [], []
for expid in range(NUM_REPEATS):
    _set_seed(expid * 227)
    model = build_model()
    experiment = Experiment(
        dataset,
        model,
        batch_size=hparams.batch_size,
        num_classes=num_classes,
        lr=hparams.lr,
        probe_train_batch=1,  # 1 batch = 200 examples
        probe_train_epochs=1000
    )
    _set_seed(1998 + expid * 227)
    train_stats = experiment.train(num_epochs=hparams.num_epochs)
    train_loss.append(train_stats["train_losses"])
    valid_loss.append(train_stats["valid_losses"])
    acc.append(train_stats["valid_accs"])

TO_SAVE["train2"] = {
    "train_loss": train_loss,
    "valid_loss": valid_loss,
    "acc": acc,
    "hparams": hparams.__dict__,
```

```
    "build_model_fn": inspect.getsource(build_model),
}

print("Avg Accuracy:", np.array(acc).mean(axis=0)[-1])
```

```
Epoch 9, Loss 14.9274: 100%|██████████| 10/10 [00:15<00:00,  1.60s/it]
Epoch 9, Loss 15.1341: 100%|██████████| 10/10 [00:15<00:00,  1.57s/it]

Avg Accuracy: 79.09
```

In [ ]:

```

```

RNN for Last Name Classification

Welcome to this assignment where you will train a neural network to predict the probable language of origin for a given last name / family name in Latin alphabets.

Throughout this task, you will gain expertise in the following areas:

- Preprocessing raw text data for suitable input into an RNN and (Optionally) LSTM.
- Utilizing PyTorch to train your recurrent neural network models.
- Evaluating your model's performance and making predictions on unseen data.

LSTM is out-of-scope this semester and will not be covered in the exams.

Download Data

In [2]:
```
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"

if not os.path.exists("data"):
    !wget https://download.pytorch.org/tutorial/data.zip
    !unzip data
```

Library imports

Before starting, make sure you have all these libraries.

In [3]:
```
root_folder = ""
import os
import sys
import inspect
sys.path.append(root_folder)
from collections import Counter
import torch
from torch import nn
import torch.nn.functional as F
import torch.optim as optim
from tqdm import tqdm

import random
import numpy as np
import json

import matplotlib.pyplot as plt
# from utils import validate_to_array

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

import IPython
from ipywidgets import interactive, widgets, Layout
from IPython.display import display, HTML
```

In [4]:
```
%load_ext autoreload
%autoreload 2
```

Implement the Neural Network

The main objective of this task is to predict the probability of a given class given a last name, represented as

$$\Pr(y|x_1, x_2, x_3, \ldots, x_i),$$

where y is the category label and each x_i is a character in the last name. Building a basic character-level NLP model has the advantage of understanding how the preprocessing works at a granular level. The character-level network reads words as a sequence of characters, producing a prediction and "hidden state" at each step by feeding its previous hidden state into the next step. The final prediction corresponds to the class to which the word belongs.

All models in PyTorch inherit from the nn.Module subclass. In this assignment, you will **implement a custom model named** `RecurrentClassifier` that runs either [nn.RNN (https://pytorch.org/docs/stable/generated/torch.nn.RNN.html)](https://pytorch.org/docs/stable/generated/torch.nn.RNN.html) or [nn.LSTM (https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html)](https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html) and define its forward function. The implementation of LSTMs is *optional*.

The forward pass of the model can be visualized with the following diagram:

```
[Embedding] -> [RNN Stack] -> [Extract Last Position] -> [Classifier]
```

- **Embedding:** This component maps each input word (integer) to a vector of real numbers.
  - Input: `[batch_size, seq_len]`

  - Output: `[batch_size, seq_len, rnn_size]`
- **RNN Stack:** This component consists of one or more RNN layers, which process the input sequence of vectors from the Embedding component.
  - Input: `[batch_size, seq_len, rnn_size]`
  - Output: `[batch_size, seq_len, rnn_size]`
- **Extract Last Position:** The RNN Stack component returns a sequence of vectors for each input example. However, for classification purposes, we only need a single vector that captures the full information of the input example. Since the RNN is left-to-right by default, the output state vector at the last position contains the full information of the input example. Therefore, for the i-th input example, we extract the output state vector at the last *non-pad* position, which is indicated by `last_pos[i]` .
  - Input: `[batch_size, seq_len, rnn_size]`
  - Output: `[batch_size, rnn_size]`
- **Classifier:** This component is a fully-connected layer that maps the output vectors extracted in the previous step to logits (scores before softmax), which can be used to make predictions about the language of origin for each input example.
  - Input: `[batch_size, rnn_size]`
  - Output: `[batch_size, n_categories]`

These documents would be helpful in this part:

- https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html (https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html)
- https://pytorch.org/docs/stable/generated/torch.nn.RNN.html (https://pytorch.org/docs/stable/generated/torch.nn.RNN.html)
- https://pytorch.org/docs/stable/generated/torch.gather.html (https://pytorch.org/docs/stable/generated/torch.gather.html)
- https://pytorch.org/docs/stable/generated/torch.Tensor.expand.html (https://pytorch.org/docs/stable/generated/torch.Tensor.expand.html)
- https://pytorch.org/docs/stable/generated/torch.Tensor.view.html (https://pytorch.org/docs/stable/generated/torch.Tensor.view.html)

In [5]:
```
class RecurrentClassifier(nn.Module):
    def __init__(
        self,
        vocab_size: int,
        rnn_size: int,
        n_categories: int,
        num_layers: int = 1,
        dropout: float = 0.0,
        model_type: str = 'lstm'
    ):
        super().__init__()
        self.rnn_size = rnn_size
        self.model_type = model_type

        ########################################################################
        # TODO: Create an embedding layer of shape [vocab_size, rnn_size]
        #
        # Hint: Use nn.Embedding
        # https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html
        # It will map each word into a vector of shape [rnn_size]
        ########################################################################
        self.embedding = nn.Embedding(vocab_size, rnn_size)
        ########################################################################

        ########################################################################
        # TODO: Create a RNN stack with `num_layers` layers with tanh
        #       nonlinearity. Between each layers, there is a dropout of
        #       `dropout`. Implement it with a *single* call to `torch.nn` APIs
        #
        # Hint: See documentations at
        # https://pytorch.org/docs/stable/generated/torch.nn.RNN.html
        # Set the arguments to call `nn.RNN` such that:
        # - The shape of the input is [batch_size, seq_len, rnn_size]
        # - The shape of the output should be [batch_size, seq_len, rnn_size]
        # Make sure that the dimension ordering is correct. One of the argument
        #   in the constructor of `nn.RNN` (or `nn.LSTM`) is helpful here
        #
        # Optional: Implement one LSTM layer when `model_type` is `lstm`
        ########################################################################
        if model_type == 'lstm':
            # set batch_first means input = (batch_size, seq_len, input_size)
            # no need to pass seq_len as lstm will get it automatically
            self.lstm = nn.LSTM(input_size = rnn_size, hidden_size = rnn_size,
                                num_layers = num_layers, dropout = dropout, batch_fi
        elif model_type == 'rnn':
            self.rnn = nn.RNN(input_size = rnn_size, hidden_size = rnn_size,
                              num_layers = num_layers, nonlinearity = "tanh",
                              dropout = dropout, batch_first = True)
        ########################################################################

        ########################################################################
        # TODO: Implement one dropout layer and the fully-connected classifier
        #       layer
        #
        # Hint: We add a dropout layer because neither nn.RNN nor nn.LSTM
        #   implements dropout after the last layer in the stack.
        # Since the input to the classifier is the output of the last position
        #   of the RNN's final layer, it has a shape of [batch_size, rnn_size].
        # The expected output should be logits, which correspond to scores
        #   before applying softmax, and should have a shape of
        #   [batch_size, n_categories].
```

```
        ##########################################################################
        self.drop = nn.Dropout(dropout)
        self.output = nn.Linear(rnn_size, n_categories)
        ##########################################################################

    def forward(self, x: torch.Tensor, last_pos: torch.Tensor) -> torch.Tensor:
        """
        x: integer tensor of shape [batch_size, seq_len]
        last_pos: integer tensor of shape [batch_size]

        The input tensor `x` is composed of a batch of sequences, where each
        sequence contains indices corresponding to characters. As sequences
        within the same batch may have different lengths, shorter sequences are
        padded on the right side to match the maximum sequence length of the
        batch, which is represented by `seq_len`.

        Additionally, the `last_pos` tensor records the position of the last
        character in each sequence. For instance, the first sequence in the
        batch can be represented as `[x[0, 0], x[0, 1], ..., x[0, last_pos[0]]`.
        `last_pos` is useful when extracting the output state associated with
        each sequence from the RNNs.
        """
        embeds = self.embedding(x)
        if self.model_type == 'lstm':
            rnn_out, _ = self.lstm(embeds)
        else:
            rnn_out, _ = self.rnn(embeds)

        ##########################################################################
        # TODO: Retrieve the output state associated with each sequence
        #
        # Hints:
        # - The output state of all positions is returned by the RNN stack,
        #   but we only need the state in the last position for classification
        #   - The shape of `rnn_out` is [batch_size, seq_len, rnn_size]
        #   - The expected shape of `out` is [batch_size, rnn_size]
        # - For the i-th sequence, we have out[i] == rnn_out[i, last_pos[i]]
        # - Try to condense your code into a single line, without using any
        #   loops. However, if you find it too challenging to do so, you may use
        #   a single layer of for-loop.
        ##########################################################################

        batch_size = x.size(0)
        seq_len = x.size(1)
        indices = last_pos.view(batch_size, 1, 1).expand(batch_size, 1, self.rnn_siz
        out = rnn_out.gather(1, indices).squeeze(1)

        ##########################################################################

        out = self.drop(out)
        logits = self.output(out)
        return logits
```

After completing your implementation, ensure that it passes the following tests. If your implementation fails some tests, but you believe that your implementation is correct, please post the error message along with a brief description on Ed. Please refrain from posting your actual code on Ed.

In [6]:
```
seed = 227
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
model = RecurrentClassifier(11, 13, 17, 2, 0.1, 'rnn')
```

In [7]:
```
assert list(model.state_dict().keys()) == ['embedding.weight',
 'rnn.weight_ih_l0',
 'rnn.weight_hh_l0',
 'rnn.bias_ih_l0',
 'rnn.bias_hh_l0',
 'rnn.weight_ih_l1',
 'rnn.weight_hh_l1',
 'rnn.bias_ih_l1',
 'rnn.bias_hh_l1',
 'output.weight',
 'output.bias']
assert model.embedding.weight.shape == torch.Size([11, 13])
assert (
    model.rnn.weight_ih_l0.shape
    == model.rnn.weight_hh_l0.shape
    == model.rnn.weight_ih_l1.shape
    == model.rnn.weight_hh_l1.shape
    == torch.Size([13, 13])
)
assert (
    model.rnn.bias_ih_l0.shape
    == model.rnn.bias_hh_l0.shape
    == model.rnn.bias_ih_l1.shape
    == model.rnn.bias_hh_l1.shape
    == torch.Size([13])
)
assert model.output.weight.shape == torch.Size([17, 13])
assert model.output.bias.shape == torch.Size([17])
```

In [8]:
```
x = torch.arange(20).view(5, 4) % 11
last_pos = torch.tensor([2, 3, 1, 2, 3])
seed = 1025
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
logits = model(x, last_pos)
print(logits.view(-1)[40:45])
assert logits.shape == torch.Size([5, 17])

'''
assert torch.allclose(
    logits.view(-1)[40:45],
    torch.tensor(
        [
            -0.27393126487731934,
            0.28421181440353394,
            0.2342953234910965,
            0.23580458760261536,
            0.06812290847301483
        ],
        dtype=torch.float
    )
)
'''


model.zero_grad()
logits.sum().backward()
print(model.rnn.weight_hh_l0.grad.view(-1)[40:45])
'''
assert torch.allclose(
    model.rnn.weight_hh_l0.grad.view(-1)[40:45],
    torch.tensor(
        [
            -0.9424352645874023,
            -0.488606333732605,
            0.6905138492584229,
            -0.0017577260732650757,
            1.1024625301361084
        ],
        dtype=torch.float
    )
)
'''
```

```
tensor([-0.2739,  0.2842,  0.2343,  0.2358,  0.0681], grad_fn=<SliceBackward0>)
tensor([-0.9424, -0.4886,  0.6905, -0.0018,  1.1025])
```

Out[8]:
```
'\nassert torch.allclose(\n    model.rnn.weight_hh_l0.grad.view(-1)[40:45],\n    torch.tensor(\n        [\n            -0.9424352645874023,\n            -0.488606333732605,\n            0.6905138492584229,\n            -0.0017577260732650757,\n            1.1024625301361084\n        ],\n        dtype=torch.float\n    )\n)\n'
```

Preprocess the dataset

The [dataset (https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html)](https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html) contains a few thousand surnames from 18 languages of origin. Included in the data/names directory are 18 text files named as "[Language].txt". Each file contains a bunch of names, one name per line, mostly romanized (but we still need to convert from Unicode to ASCII).

We'll end up with a dictionary of lists of names per language, {language: [names ...]}.

In [9]:

```
from __future__ import unicode_literals, print_function, division
from io import open
import glob
import os

def findFiles(path): return glob.glob(path)

assert findFiles('data/names/*.txt'), "Data not found!"

import unicodedata
import string

all_letters = string.ascii_letters + " .,;'"
n_letters = len(all_letters)

# Turn a Unicode string to plain ASCII, thanks to https://stackoverflow.com/a/518232
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
        and c in all_letters
    )

print("The normalized form of", 'Ślusàrski', "is", unicodeToAscii('Ślusàrski'))

# Build the category_lines dictionary, a list of names per language
category_lines = {}
all_categories = []

# Read a file and split into lines
def readLines(filename):
    lines = open(filename, encoding='utf-8').read().strip().split('\n')
    return [unicodeToAscii(line) for line in lines]

for filename in findFiles('data/names/*.txt'):
    category = os.path.splitext(os.path.basename(filename))[0]
    all_categories.append(category)
    lines = readLines(filename)
    category_lines[category] = lines

n_categories = len(all_categories)
```

```
The normalized form of Ślusàrski is Slusarski
```

In [10]:

```
print(category_lines['Italian'][:5])
```

```
['Abandonato', 'Abatangelo', 'Abatantuono', 'Abate', 'Abategiovanni']
```

In [11]:
```
all_letters
```

Out[11]:
```
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ .,;'"
```

Implement the function to encode a letter to an integer:

In [12]:
```
def letterToIndex(letter):
    ###########################################################################
    # TODO: implement the function to map a letter (a character) into its index
    #       in `all_letters`
    #
    # e.g. letterToIndex("a") == 0
    # Don't worry about efficiency here
    ###########################################################################
    return all_letters.index(letter)
    ###########################################################################

assert letterToIndex("a") == 0
assert letterToIndex("'") == 56
```

In [13]:
```
category_lines.keys()
```

Out[13]:
```
dict_keys(['Arabic', 'Chinese', 'Czech', 'Dutch', 'English', 'French', 'German',
'Greek', 'Irish', 'Italian', 'Japanese', 'Korean', 'Polish', 'Portuguese', 'Russi
an', 'Scottish', 'Spanish', 'Vietnamese'])
```

In [14]:
```
# For labels, we must have numbers instead of a string. These dictionaries convert
# between these two ways of representing the labels.
num_to_cat = dict(enumerate(category_lines.keys()))
print(num_to_cat)
cat_to_num = dict((v,k) for k,v in num_to_cat.items())
print(cat_to_num)

pad = 57 # this is the next available character
vocab_size = 58 # number of characters used in total
```

```
{0: 'Arabic', 1: 'Chinese', 2: 'Czech', 3: 'Dutch', 4: 'English', 5: 'French', 6:
'German', 7: 'Greek', 8: 'Irish', 9: 'Italian', 10: 'Japanese', 11: 'Korean', 12:
'Polish', 13: 'Portuguese', 14: 'Russian', 15: 'Scottish', 16: 'Spanish', 17: 'Vi
etnamese'}
{'Arabic': 0, 'Chinese': 1, 'Czech': 2, 'Dutch': 3, 'English': 4, 'French': 5, 'G
erman': 6, 'Greek': 7, 'Irish': 8, 'Italian': 9, 'Japanese': 10, 'Korean': 11, 'P
olish': 12, 'Portuguese': 13, 'Russian': 14, 'Scottish': 15, 'Spanish': 16, 'Viet
namese': 17}
```

In [15]:
```
np.ones(19, dtype=np.int64) * 57
```

Out[15]:
```
array([57, 57, 57, 57, 57, 57, 57, 57, 57, 57, 57, 57, 57, 57, 57, 57, 57,
       57, 57], dtype=int64)
```

In [16]:
```
def build_data():
  '''
  category_lines: a dictionary of lists of names per language, {language: [names ...

  We want to translate our dictionary into a dataset that has one entry per name.
  Each datapoint is a 3-tuple consisting of:
  - x: a length-19 array with each character in the name as an element,
   padded with zeros at the end if the name is less than 19 characters.
  - y: the numerical representation of the language the name corresponds to.
  - index: the index of the last non-pad token
  '''
  data = []
  for cat in category_lines:
    for name in category_lines[cat]:
      token = np.ones(19, dtype=np.int64) * pad
      numerized = np.array([letterToIndex(l) for l in name])
      n = len(numerized)
      token[:n] = numerized
      data.append((token, cat_to_num[cat], n-1))
  return data
```

In [17]:
```
data = build_data()
seed = 227
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
random.shuffle(data)
```

In [18]:
```
data[0]
```

Out[18]:
```
(array([32, 17, 14,  8, 18, 12,  0, 13, 57, 57, 57, 57, 57, 57, 57, 57, 57,
        57, 57], dtype=int64),
 14,
 7)
```

In [19]:
```
n_train = int(len(data) * 0.8)
train_data = data[:n_train]
test_data = data[n_train:]
```

In [20]:
```
len(train_data)
```

Out[20]:
```
16059
```

In [21]:
```
train_data[0]
```

Out[21]:
```
(array([32, 17, 14,  8, 18, 12,  0, 13, 57, 57, 57, 57, 57, 57, 57, 57, 57,
        57, 57], dtype=int64),
 14,
 7)
```

In [22]:
```
len(test_data)
```

Out[22]:
```
4015
```

In [23]:
```
test_data[0]
```

Out[23]:
```
(array([27,  4, 11, 14, 24,  0, 17, 19, 18,  4, 21, 57, 57, 57, 57, 57, 57,
        57, 57], dtype=int64),
 14,
 10)
```

Train the model

Training will be faster if you use the Colab GPU. If it's not already enabled, do so with Runtime -> Change runtime type.

In [24]:
```
def build_batch(dataset, indices):
    '''
    Helper function for creating a batch during training. Builds a batch
    of source and target elements from the dataset. See the next cell for
    when and how it's used.

    Arguments:
        dataset: List[db_element] -- A list of dataset elements
        indices: List[int] -- A list of indices of the dataset to sample
    Returns:
        batch_input: List[List[int]] -- List of tensorized names
        batch_target: List[int] -- List of numerical categories
        batch_indices: List[int] -- List of starting indices of padding
    '''

    # Recover what the entries for the batch are
    batch = [dataset[i] for i in indices]
    batch_input = np.array(list(zip(*batch))[0])
    batch_target = np.array(list(zip(*batch))[1])
    batch_indices = np.array(list(zip(*batch))[2])
    return batch_input, batch_target, batch_indices # lines, categories
```

In [25]:
```
build_batch(train_data, [1, 2, 3])
```

Out[25]:
```
(array([[31, 14, 17,  3,  7,  0, 12, 57, 57, 57, 57, 57, 57, 57, 57, 57,
         57, 57, 57],
        [32, 14, 11, 14,  7,  0, 57, 57, 57, 57, 57, 57, 57, 57, 57, 57,
         57, 57, 57],
        [26, 12,  4, 19,  8, 18, 19, 14, 21, 57, 57, 57, 57, 57, 57, 57,
         57, 57, 57]], dtype=int64),
 array([ 4, 14, 14]),
 array([6, 5, 8]))
```

Adjust the hyperparameters listed below to train an RNN with a minimum evaluation accuracy of 80% after 20 epochs. Your score will be graded on a linear scale, ranging from 0 to the maximum score, as the validation accuracy achieved after the last epoch changes from 70% to 80% (i.e., you get 0 if the accuracy is less than 70%, and get the full score if the accuracy is greater than 80% for this autograding item).

In [26]:
```
criterion = nn.CrossEntropyLoss()

# The build_batch function outputs numpy, but our model is built in pytorch,
# so we need to convert numpy to pytorch with the correct types.
batch_to_torch = lambda b_in,b_target,b_mask: (torch.tensor(b_in).long(),
                                               torch.tensor(b_target).long(),
                                               torch.tensor(b_mask).long())

############################################################################
# TODO: Tune these hyperparameters for a better performance
############################################################################
hidden_size = 32
num_layers = 1
dropout = 0.0
optimizer_class = optim.Adam
lr = 1e-4
batch_size = 256
############################################################################

# Do not change the number of epochs
epochs = 20

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("You are using", device, "for training")
list_to_device = lambda th_obj: [tensor.to(device) for tensor in th_obj]
```

```
You are using cuda for training
```

In [27]:
```
# Optional
# lstm_model = RecurrentClassifier(vocab_size=vocab_size, rnn_size=hidden_size, n_ca
# lstm_optimizer = optimizer_class(lstm_model.parameters(), lr=lr)
```

In [28]:
```
seed = 1998
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
rnn_model = RecurrentClassifier(vocab_size=vocab_size, rnn_size=hidden_size, n_categ
rnn_optimizer = optimizer_class(rnn_model.parameters(), lr=lr)
```

In [29]:
```
def train(model, optimizer, criterion, epochs, batch_size, seed):
    model.to(device)
    model.train()
    train_losses = []
    train_accuracies = []
    eval_accuracies = []
    for epoch in range(epochs):
        random.seed(seed + epoch)
        np.random.seed(seed + epoch)
        torch.manual_seed(seed + epoch)
        indices = np.random.permutation(range(len(train_data)))
        n_correct, n_total = 0, 0
        progress_bar = tqdm(range(0, (len(train_data) // batch_size) + 1))
        for i in progress_bar:
            batch = build_batch(train_data, indices[i*batch_size:(i+1)*batch_size])
            (batch_input, batch_target, batch_indices) = batch_to_torch(*batch)
            (batch_input, batch_target, batch_indices) = list_to_device((batch_input

            logits = model(batch_input, batch_indices)
            loss = criterion(logits, batch_target)
            train_losses.append(loss.item())

            predictions = logits.argmax(dim=-1)
            n_correct += (predictions == batch_target).sum().item()
            n_total += batch_target.size(0)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            if (i + 1) % 10 == 0:
                progress_bar.set_description(f"Epoch: {epoch}  Iteration: {i}  Loss:
        train_accuracies.append(n_correct / n_total * 100)
        print(f"Epoch: {epoch}  Train Accuracy: {n_correct / n_total * 100}")

        with torch.no_grad():
            indices = list(range(len(test_data)))
            n_correct, n_total = 0, 0
            for i in range(0, (len(test_data) // batch_size) + 1):
                batch = build_batch(test_data, indices[i*batch_size:(i+1)*batch_size
                (batch_input, batch_target, batch_indices) = batch_to_torch(*batch)
                (batch_input, batch_target, batch_indices) = list_to_device((batch_i

                logits = model(batch_input, batch_indices)
                predictions = logits.argmax(dim=-1)
                n_correct += (predictions == batch_target).sum().item()
                n_total += batch_target.size(0)
            eval_accuracies.append(n_correct / n_total * 100)
            print(f"Epoch: {epoch}  Eval Accuracy: {n_correct / n_total * 100}")

    to_save = {
        "history": {
            "train_losses": train_losses,
            "train_accuracies": train_accuracies,
            "eval_accuracies": eval_accuracies,
        },
        "hparams": {
            "hidden_size": hidden_size,
            "num_layers": num_layers,
            "dropout": dropout,
            "optimizer_class": optimizer_class.__name__,
```

```
            "lr": lr,
            "batch_size": batch_size,
            "epochs": epochs,
            "seed": seed
        },
        "model": [
            (name, list(param.shape))
            for name, param in rnn_model.named_parameters()
        ]
    }
    return to_save
```

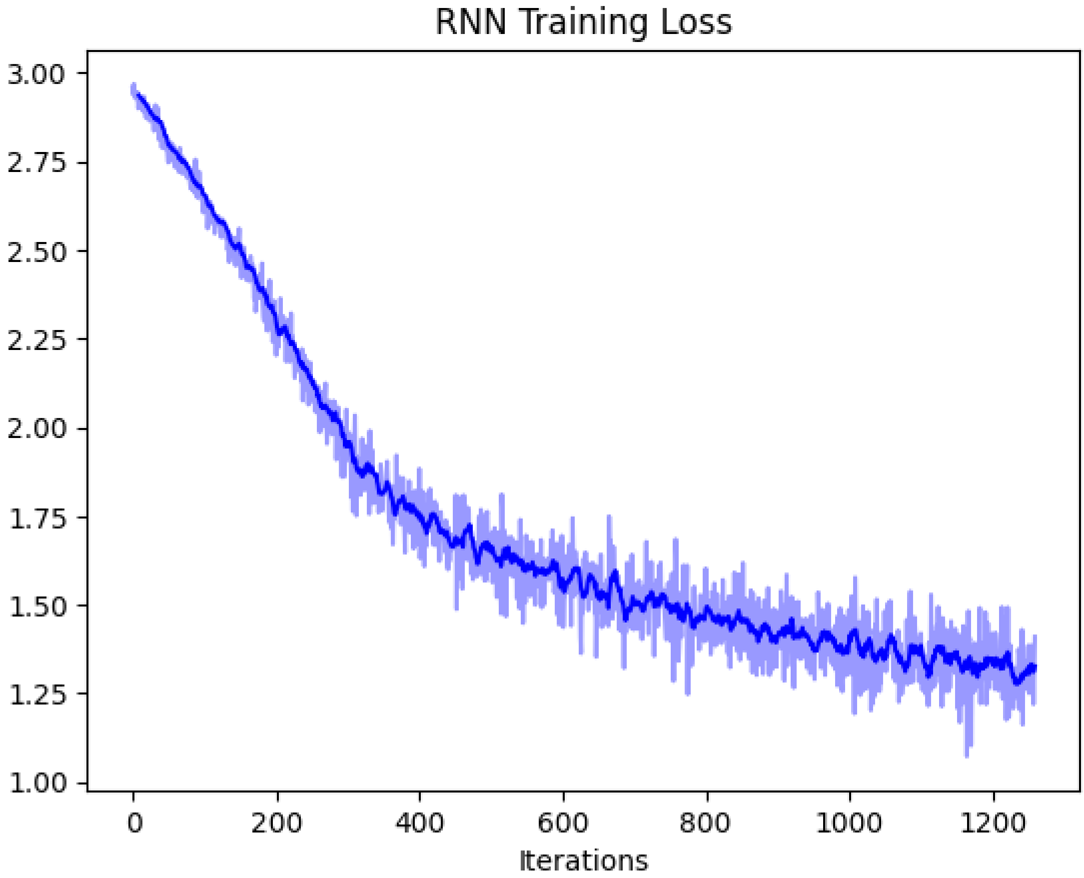
In [30]:
```
rnn_log = train(rnn_model, rnn_optimizer, criterion, epochs, batch_size, 1997)
```

```
Epoch: 0  Iteration: 59  Loss: 2.77957603931427: 100%|█████████████| 63/63
[00:00<00:00, 185.55it/s]

Epoch: 0  Train Accuracy: 11.91232330780248
Epoch: 0  Eval Accuracy: 24.73225404732254

Epoch: 1  Iteration: 59  Loss: 2.579754614830017: 100%|█████████████| 63/6
3 [00:00<00:00, 473.25it/s]

Epoch: 1  Train Accuracy: 31.309546048944515
Epoch: 1  Eval Accuracy: 35.2428393524284

Epoch: 2  Iteration: 59  Loss: 2.3747825622558594: 100%|█████████████| 63/
63 [00:00<00:00, 464.86it/s]

Epoch: 2  Train Accuracy: 38.19042281586649
Epoch: 2  Eval Accuracy: 40.473225404732254

Epoch: 3  Iteration: 59  Loss: 2.142752456665039: 100%|█████████████| 63/6
3 [00:00<00:00, 468.46it/s]

Epoch: 3  Train Accuracy: 44.99034809141291
Epoch: 3  Eval Accuracy: 45.70361145703611

Epoch: 4  Iteration: 59  Loss: 1.905372440814972: 100%|█████████████| 63/6
3 [00:00<00:00, 472.94it/s]

Epoch: 4  Train Accuracy: 47.71156360919111
Epoch: 4  Eval Accuracy: 47.073474470734745

Epoch: 5  Iteration: 59  Loss: 1.7912390232086182: 100%|█████████████| 63/
63 [00:00<00:00, 489.18it/s]

Epoch: 5  Train Accuracy: 47.79251510056666
Epoch: 5  Eval Accuracy: 47.3225404732254

Epoch: 6  Iteration: 59  Loss: 1.7023853540420533: 100%|█████████████| 63/
63 [00:00<00:00, 414.25it/s]

Epoch: 6  Train Accuracy: 48.04159661249144
Epoch: 6  Eval Accuracy: 47.72104607721046

Epoch: 7  Iteration: 59  Loss: 1.6512146830558776: 100%|█████████████| 63/
63 [00:00<00:00, 437.77it/s]

Epoch: 7  Train Accuracy: 49.056603773584904
Epoch: 7  Eval Accuracy: 49.115815691158154

Epoch: 8  Iteration: 59  Loss: 1.591255533695221: 100%|█████████████| 63/6
3 [00:00<00:00, 446.12it/s]

Epoch: 8  Train Accuracy: 50.495049504950494
Epoch: 8  Eval Accuracy: 50.2615193026152

Epoch: 9  Iteration: 59  Loss: 1.5436719179153442: 100%|█████████████| 63/
63 [00:00<00:00, 466.36it/s]

Epoch: 9  Train Accuracy: 51.678186686593186
Epoch: 9  Eval Accuracy: 50.90909090909091

Epoch: 10  Iteration: 59  Loss: 1.4651575446128846: 100%|█████████████| 6
3/63 [00:00<00:00, 434.46it/s]
```

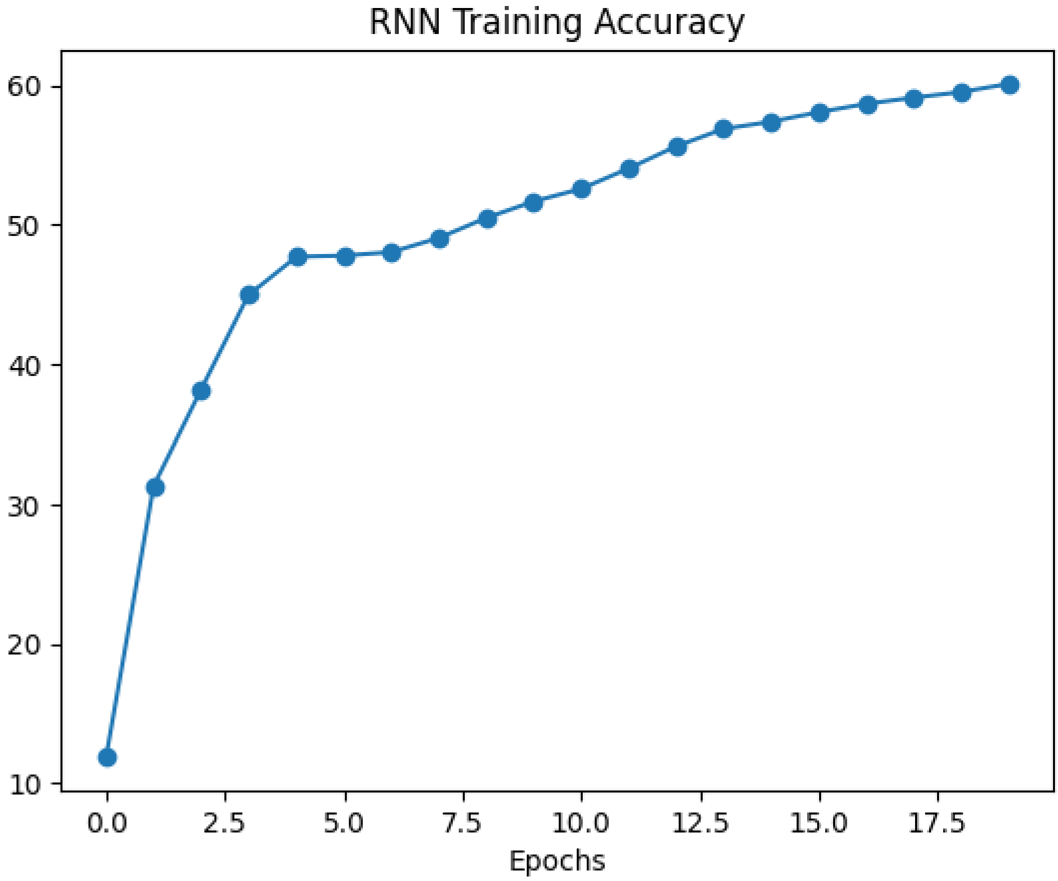
```
Epoch: 10  Train Accuracy: 52.5811071673205
Epoch: 10  Eval Accuracy: 51.78082191780822

Epoch: 11  Iteration: 59  Loss: 1.4844784259796142: 100%|██████████| 6
3/63 [00:00<00:00, 404.72it/s]

Epoch: 11  Train Accuracy: 54.03823401208045
Epoch: 11  Eval Accuracy: 53.97260273972603

Epoch: 12  Iteration: 59  Loss: 1.4506823778152467: 100%|██████████| 6
3/63 [00:00<00:00, 411.01it/s]

Epoch: 12  Train Accuracy: 55.62612865060091
Epoch: 12  Eval Accuracy: 55.342465753424655

Epoch: 13  Iteration: 59  Loss: 1.4293978095054627: 100%|██████████| 6
3/63 [00:00<00:00, 462.91it/s]

Epoch: 13  Train Accuracy: 56.877763248022916
Epoch: 13  Eval Accuracy: 55.666251556662516

Epoch: 14  Iteration: 59  Loss: 1.410203492641449: 100%|██████████| 63/
63 [00:00<00:00, 429.97it/s]

Epoch: 14  Train Accuracy: 57.39460738526682
Epoch: 14  Eval Accuracy: 56.68742216687422

Epoch: 15  Iteration: 59  Loss: 1.4162867546081543: 100%|██████████| 6
3/63 [00:00<00:00, 431.80it/s]

Epoch: 15  Train Accuracy: 58.079581543059966
Epoch: 15  Eval Accuracy: 57.359900373599004

Epoch: 16  Iteration: 59  Loss: 1.3556838870048522: 100%|██████████| 6
3/63 [00:00<00:00, 461.80it/s]

Epoch: 16  Train Accuracy: 58.67737717167943
Epoch: 16  Eval Accuracy: 57.98256537982566

Epoch: 17  Iteration: 59  Loss: 1.3696428894996644: 100%|██████████| 6
3/63 [00:00<00:00, 443.64it/s]

Epoch: 17  Train Accuracy: 59.11949685534591
Epoch: 17  Eval Accuracy: 58.43088418430884

Epoch: 18  Iteration: 59  Loss: 1.3503025412559508: 100%|██████████| 6
3/63 [00:00<00:00, 446.38it/s]

Epoch: 18  Train Accuracy: 59.524254312223675
Epoch: 18  Eval Accuracy: 58.92901618929016

Epoch: 19  Iteration: 59  Loss: 1.3224847793579102: 100%|██████████| 6
3/63 [00:00<00:00, 392.61it/s]

Epoch: 19  Train Accuracy: 60.1095958652469
Epoch: 19  Eval Accuracy: 59.60149439601494
```

In [31]:

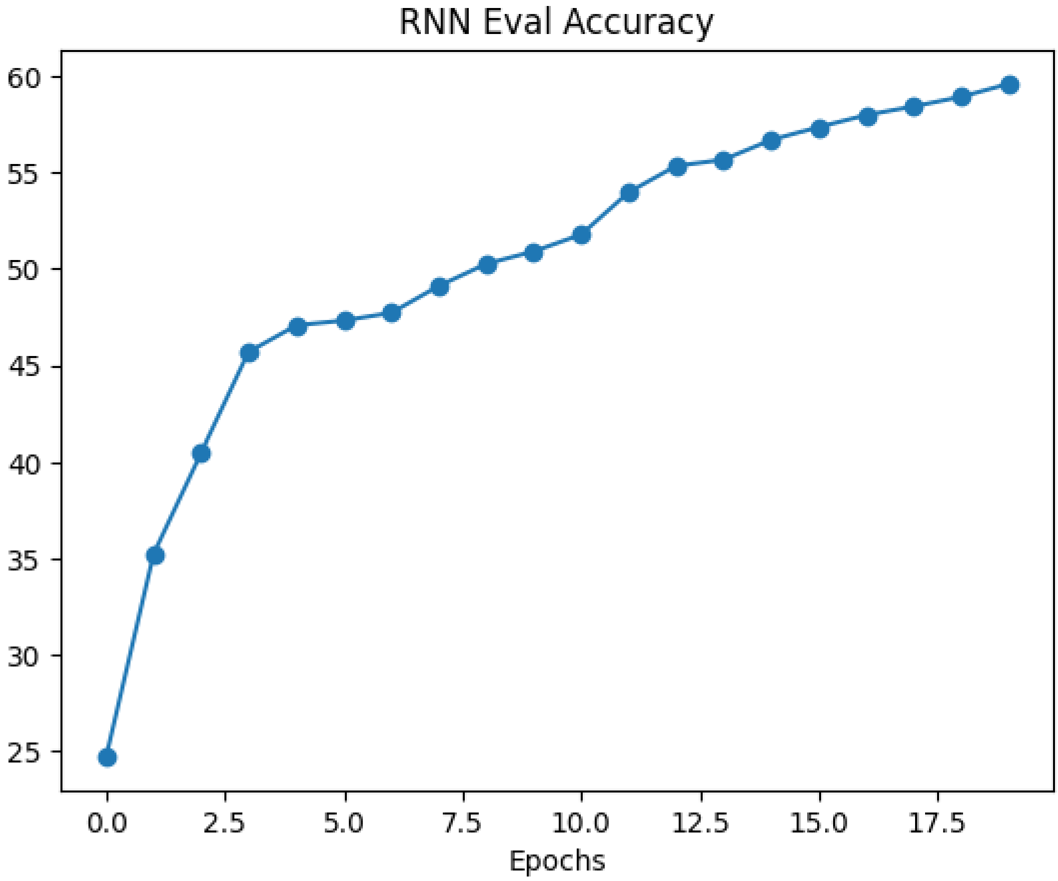
```
n_steps = len(rnn_log["history"]["train_losses"])
plt.plot(range(n_steps), rnn_log["history"]["train_losses"], alpha=0.4, color="blue"
moving_avg = np.convolve(np.array(rnn_log["history"]["train_losses"]), np.ones(10),
plt.plot(range(9, n_steps), moving_avg.tolist(), color="blue")
plt.xlabel("Iterations")
plt.title("RNN Training Loss")
plt.show()
```



In [32]:
```
plt.plot(rnn_log["history"]["train_accuracies"], marker='o')
plt.xlabel("Epochs")
plt.title("RNN Training Accuracy")
plt.show()
```



In [33]:
```
plt.plot(rnn_log["history"]["eval_accuracies"], marker='o')
plt.xlabel("Epochs")
plt.title("RNN Eval Accuracy")
plt.show()
```



In [39]:
```
# Optional
# train(lstm_model, lstm_optimizer, criterion, epochs, batch_size, 1997)

lstm_model = RecurrentClassifier(vocab_size=vocab_size, rnn_size=hidden_size, n_cate
lstm_optimizer = optimizer_class(lstm_model.parameters(), lr=lr)
train(lstm_model, lstm_optimizer, criterion, epochs, batch_size, 1997)
```

```
Epoch: 0  Iteration: 59  Loss: 2.9269559383392334: 100%|██████████|
63/63 [00:00<00:00, 187.38it/s]

Epoch: 0  Train Accuracy: 7.665483529485025
Epoch: 0  Eval Accuracy: 8.617683686176838

Epoch: 1  Iteration: 59  Loss: 2.8329697608947755: 100%|██████████|
63/63 [00:00<00:00, 378.12it/s]

Epoch: 1  Train Accuracy: 9.595865246902049
Epoch: 1  Eval Accuracy: 11.481942714819429

Epoch: 2  Iteration: 59  Loss: 2.7042388677597047: 100%|██████████|
63/63 [00:00<00:00, 416.99it/s]

Epoch: 2  Train Accuracy: 17.07453764244349
Epoch: 2  Eval Accuracy: 31.008717310087174

Epoch: 3  Iteration: 59  Loss: 2.490705442428589: 100%|██████████| 6
3/63 [00:00<00:00, 425.40it/s]
```

Use Your RNN: Try Your Own Name

Attempt to use the code cells below to **predict the origin of your own last name**.

Please refrain from entering the last names of your classmates, as the names you enter will be logged for anti-plagiarism purposes.

In [40]:
```
def classify_name(name, model):
    '''
    Numerize the name and return the most likely number representation of the
    predicted class.
    '''
    # change this if your last name is longer than 19 characters
    token = np.ones(19, dtype=np.int64) * pad
    numerized = np.array([letterToIndex(l) for l in name])
    n = len(numerized)
    token[:n] = numerized
    print(token)
    logits = model(
        torch.tensor(token, dtype=torch.long)[None, :],
        torch.tensor([n - 1], dtype=torch.long)
    )
    return logits.argmax(dim=-1).item()
```

In [45]:
```
model = rnn_model
model.eval()
model.cpu()
##############################################################################
# TODO: Enter your last name
##############################################################################
name = "Chen"
##############################################################################
rnn_log["last_name"] = name
rnn_log["source_init"] = inspect.getsource(RecurrentClassifier.__init__)
rnn_log["source_forward"] = inspect.getsource(RecurrentClassifier.forward)
print("Predicting origin language for name: "+ name)
c = classify_name(name, model)
print(num_to_cat[c])
```

```
Predicting origin language for name: Chen
[28  7  4 13 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57]
English
```

In [49]:
```
model = lstm_model
model.eval()
model.cpu()
##############################################################################
# TODO: Enter your last name
##############################################################################
name = "Chen"
##############################################################################
rnn_log["last_name"] = name
rnn_log["source_init"] = inspect.getsource(RecurrentClassifier.__init__)
rnn_log["source_forward"] = inspect.getsource(RecurrentClassifier.forward)
print("Predicting origin language for name: "+ name)
c = classify_name(name, model)
print(num_to_cat[c])
```

```
Predicting origin language for name: Chen
[28  7  4 13 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57]
English
```

Question

Although the neural network you have trained is intended to predict the language of origin for a given last name, it could potentially be misused. **In what ways do you think this could be problematic in real-world applications?** Include your answer in your submission of the written assignment.

In [1]:

```
!wget https://raw.githubusercontent.com/Berkeley-CS182/cs182fa23_public/main/q_wand
!pip install wandb
```

```
--2023-10-17 21:26:48--  https://raw.githubusercontent.com/Berkeley-CS182/cs182fa23_public/main/q_wandbai/architectures.py (https://raw.githubusercontent.com/Berkeley-CS182/cs182fa23_public/main/q_wandbai/architectures.py)
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1618 (1.6K) [text/plain]
Saving to: 'architectures.py'

     0K .                                                     100%  744K=0.002s

2023-10-17 21:26:48 (744 KB/s) - 'architectures.py' saved [1618/1618]
```

```
Collecting wandb
  Downloading wandb-0.15.12-py3-none-any.whl (2.1 MB)
     ---------------------------------------- 2.1/2.1 MB 14.8 MB/s eta 0:00:00
Collecting docker-pycreds>=0.4.0
  Downloading docker_pycreds-0.4.0-py2.py3-none-any.whl (9.0 kB)
Requirement already satisfied: appdirs>=1.4.3 in d:\anaconda\anaconda_setup\envs
\malning\lib\site-packages (from wandb) (1.4.4)
Requirement already satisfied: setuptools in d:\anaconda\anaconda_setup\envs\maln
ing\lib\site-packages (from wandb) (63.4.1)
Collecting pathtools
  Downloading pathtools-0.1.2.tar.gz (11 kB)
  Preparing metadata (setup.py): started
  Preparing metadata (setup.py): finished with status 'done'
Collecting setproctitle
  Downloading setproctitle-1.3.3-cp37-cp37m-win_amd64.whl (11 kB)
Collecting GitPython!=3.1.29,>=1.0.0
  Downloading GitPython-3.1.38-py3-none-any.whl (190 kB)
     ---------------------------------------- 190.6/190.6 kB ? eta 0:00:00
Requirement already satisfied: PyYAML in d:\anaconda\anaconda_setup\envs\malning
\lib\site-packages (from wandb) (6.0)
Requirement already satisfied: typing-extensions in d:\anaconda\anaconda_setup\en
vs\malning\lib\site-packages (from wandb) (4.4.0)
Requirement already satisfied: protobuf!=4.21.0,<5,>=3.19.0 in d:\anaconda\anacon
da_setup\envs\malning\lib\site-packages (from wandb) (3.19.6)
Requirement already satisfied: requests<3,>=2.0.0 in d:\anaconda\anaconda_setup\e
nvs\malning\lib\site-packages (from wandb) (2.28.1)
Requirement already satisfied: Click!=8.0.0,>=7.1 in d:\anaconda\anaconda_setup\e
nvs\malning\lib\site-packages (from wandb) (8.1.3)
Requirement already satisfied: psutil>=5.0.0 in d:\anaconda\anaconda_setup\envs\m
alning\lib\site-packages (from wandb) (5.9.3)
Collecting sentry-sdk>=1.0.0
  Downloading sentry_sdk-1.32.0-py2.py3-none-any.whl (240 kB)
     ------------------------------------- 241.0/241.0 kB 15.4 MB/s eta 0:00:00
Requirement already satisfied: importlib-metadata in d:\anaconda\anaconda_setup\e
nvs\malning\lib\site-packages (from Click!=8.0.0,>=7.1->wandb) (5.0.0)
Requirement already satisfied: colorama in d:\anaconda\anaconda_setup\envs\malnin
g\lib\site-packages (from Click!=8.0.0,>=7.1->wandb) (0.4.6)
Requirement already satisfied: six>=1.4.0 in d:\anaconda\anaconda_setup\envs\maln
ing\lib\site-packages (from docker-pycreds>=0.4.0->wandb) (1.16.0)
Collecting gitdb<5,>=4.0.1
  Downloading gitdb-4.0.10-py3-none-any.whl (62 kB)
     ---------------------------------------- 62.7/62.7 kB ? eta 0:00:00
Requirement already satisfied: charset-normalizer<3,>=2 in d:\anaconda\anaconda_s
etup\envs\malning\lib\site-packages (from requests<3,>=2.0.0->wandb) (2.1.1)
Requirement already satisfied: idna<4,>=2.5 in d:\anaconda\anaconda_setup\envs\ma
lning\lib\site-packages (from requests<3,>=2.0.0->wandb) (3.4)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in d:\anaconda\anaconda_setu
p\envs\malning\lib\site-packages (from requests<3,>=2.0.0->wandb) (1.26.12)
Requirement already satisfied: certifi>=2017.4.17 in d:\anaconda\anaconda_setup\e
nvs\malning\lib\site-packages (from requests<3,>=2.0.0->wandb) (2023.7.22)
Collecting smmap<6,>=3.0.1
  Downloading smmap-5.0.1-py3-none-any.whl (24 kB)
Requirement already satisfied: zipp>=0.5 in d:\anaconda\anaconda_setup\envs\malni
ng\lib\site-packages (from importlib-metadata->Click!=8.0.0,>=7.1->wandb) (3.10.
0)
Building wheels for collected packages: pathtools
  Building wheel for pathtools (setup.py): started
  Building wheel for pathtools (setup.py): finished with status 'done'
  Created wheel for pathtools: filename=pathtools-0.1.2-py3-none-any.whl size=879
2 sha256=3a43b3d11799db09ec92120b06a6c8f6deccae8ba0055f23c6f991ae93b4b1cf
  Stored in directory: c:\users\cyt\appdata\local\pip\cache\wheels\3e\31\09\fa59c
```

```
ef12cdcfecc627b3d24273699f390e71828921b2cbba2
Successfully built pathtools
Installing collected packages: pathtools, smmap, setproctitle, sentry-sdk, docker
-pycreds, gitdb, GitPython, wandb
Successfully installed GitPython-3.1.38 docker-pycreds-0.4.0 gitdb-4.0.10 pathtoo
ls-0.1.2 sentry-sdk-1.32.0 setproctitle-1.3.3 smmap-5.0.1 wandb-0.15.12
```

In [1]:

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import wandb
from architectures import BasicConvNet, ResNet18, MLP
from torch.utils.tensorboard import SummaryWriter
from tqdm import tqdm
from torch.utils.data import DataLoader
```

executed in 1.57s, finished 13:53:56 2023-10-18

Exploring Tensorboard

Tensorboard is a local tool for visualizing images, metrics, histograms, and more. It is designed for tensorflow, but can be integrated with torch. Let's explore tensorboard usage with an example:

```
from torch.utils.tensorboard import SummaryWriter

# To start a run, call the following
writer = SummaryWriter(comment=f'Name_of_Run')

# When you want to log a value, use the writer. When adding a scalar, the f
ormat is as follows:
# add_scalar(tag, scalar_value, global_step=None, walltime=None, new_style=
False, double_precision=False)
writer.add_scalar('Training Loss', loss.item(), step)

# Finally, when you are done logging values, close the writer
writer.close()
```

There are many other functionalities and methods that you are free to explore, but will not be mentioned in this notebook.

Your Task

We will be once again building classifiers for the CIFAR-10. There are various architectures set up for you to use in the architectures.py file. Using tensorboard, please search through 5 different hyperparameter configurations. Examples of choices include: learning rate, batch size, architecture, optimization algorithm, etc. Please submit the generated plots on your pdf and answer question A.

In [2]:
```
epochs = 2
cuda = torch.cuda.is_available()
device = torch.device("cuda" if cuda else "cpu")
```
executed in 1.35s, finished 13:53:59 2023-10-18

In [3]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./../cifar-10/', train=True,
                                        download=True, transform=transform)
testset = torchvision.datasets.CIFAR10(root='./../cifar-10/', train=False,
                                       download=True, transform=transform)
```
executed in 2.60s, finished 13:54:02 2023-10-18

```
Files already downloaded and verified
Files already downloaded and verified
```

In [4]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device
```
executed in 16ms, finished 13:54:02 2023-10-18

Out[4]:
```
device(type='cuda')
```

In [5]:
```python
def get_optimizer(params, optim_type, lr):
    if optim_type == "sgd":
        optimizer = optim.SGD(params, lr=lr)
    elif optim_type == "adam":
        optimizer = optim.Adam(params, lr=lr)
    else:
        raise ValueError(optim_type)

    return optimizer

def get_model(model_type):
    if model_type == "basicconvnet":
        model = BasicConvNet()
    elif model_type == "resnet18":
        model = ResNet18()
    elif model_type == "mlp":
        model = MLP()
    else:
        raise ValueError(model_type)

    return model

def get_criterion(loss_type):
    if(loss_type == "mse"):
        criterion = nn.MSELoss()
    elif(loss_type == "cross"):
        criterion = nn.CrossEntropyLoss()
    else:
        raise ValueError(loss_type)

    return criterion
```

executed in 13ms, finished 13:54:03 2023-10-18

In [6]:
```python
def train(writer, dataloader, model, loss_fn, optimizer, epoch):
    size = len(dataloader.dataset)
    num_batch = len(dataloader)
    model.train()

    total_loss = 0
    correct = 0

    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        pred = model(X)
        loss = loss_fn(pred, y)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        correct += (pred.argmax(1) == y).type(torch.float).sum().item()

        if(batch % 100 == 0):
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f} [{current:>5d} / {size:>5d}]")

    avg_loss = total_loss / num_batch
    correct /= size

    # write into tensorboard
    writer.add_scalar("Train Loss", avg_loss, epoch)
    writer.add_scalar("Train Acc", correct, epoch)

    print(f"Train Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {avg_loss:>8


def test(writer, dataloader, model, loss_fn, epoch):

    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()

    test_loss = 0
    correct = 0.1
    with torch.no_grad():
        for batch, (X, y) in enumerate(dataloader):
            X, y = X.cuda(), y.cuda()
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size

    # write into tensorboard
    writer.add_scalar('Test Loss', test_loss, epoch)
    writer.add_scalar('Test Acc', correct, epoch)

    print(f"Evaluation Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_l
```

executed in 20ms, finished 13:54:04 2023-10-18

In [10]:
```
def run_training(trainset, testset, hyperparameters, log_dir = "logs"):

    print("---------------------------config-----------------------------")
    print(hyperparameters)
    print("--------------------------------------------------------------")

    name = ""
    for i, key in enumerate(hyperparameters.keys()):
        value = hyperparameters[key]
        if i != (len(hyperparameters.keys()) - 1):
            item = key + "_" + str(value) + "_"
        else:
            item = key + "_" + str(value)
        name = name + item

    model_type = hyperparameters['model']
    model = get_model(model_type)
    loss_type = hyperparameters['loss_fn']
    criterion = get_criterion(loss_type)
    learning_rate = hyperparameters['lr']
    optim_type = hyperparameters['optimizer']
    optimizer = get_optimizer(model.parameters(), optim_type, lr=learning_rate)
    batch_size = hyperparameters['batch_size']
    num_epochs = hyperparameters['epochs']

    # build train data loader
    trainloader = DataLoader(trainset, batch_size=batch_size, shuffle=True)
    # build test data loader
    testloader = DataLoader(testset, batch_size=batch_size, shuffle=False)

    # create a tensorboard writer
    path = log_dir + "/" + name
    writer = SummaryWriter(path)
    print(f"log will be written to {path}")

    model.cuda()

    for t in range(num_epochs):
        print(f"Epoch {t+1}\n-------------------------------")
        train(writer, trainloader, model, criterion, optimizer, t+1)
        test(writer, testloader, model, criterion, t+1)

    writer.close()
```

executed in 15ms, finished 14:05:20 2023-10-18

In [11]:
```
hyperparameters1 = {
    "model" : "basicconvnet",
    "lr" : 0.0001,
    "loss_fn" : "cross",
    "optimizer" : "adam",
    "epochs" : 20,
    "batch_size" : 16
}

hyperparameters2 = {
    "model" : "resnet18",
    "lr" : 0.0001,
    "loss_fn" : "cross",
    "optimizer" : "adam",
    "epochs" : 20,
    "batch_size" : 16
}

hyperparameters3 = {
    "model" : "mlp",
    "lr" : 0.0001,
    "loss_fn" : "cross",
    "optimizer" : "adam",
    "epochs" : 20,
    "batch_size" : 16
}
```

executed in 8ms, finished 14:05:23 2023-10-18

In [12]:
```
def run():
    # Perhaps you want to make a function to train on a certain set of hyperparamete
    # Don't forget to use tensorboard
    run_training(trainset, testset, hyperparameters1)
    run_training(trainset, testset, hyperparameters2)
    run_training(trainset, testset, hyperparameters3)
```

executed in 6ms, finished 14:05:45 2023-10-18

In [ ]:

Exploring Tooling with Weights and Biases

Similar to tensorboard, weights and biases is an application that tracks all your training metrics, and performs visualizations for you. This tool allows you to cleanly sort, organize, and visualize your experiments. In this notebook, we will go through an example of how to use wandb.ai and have you practice.

1. Make an account at [https://wandb.ai/site (https://wandb.ai/site)](https://wandb.ai/site)
2. pip install wandb
3. wandb login
4. After step 3, please paste your wandb API key

In [ ]:
```
!wget https://raw.githubusercontent.com/Berkeley-CS182/cs182fa23_public/main/q_wand
!pip install wandb
```

In [1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import wandb
from architectures import BasicConvNet, ResNet18, MLP
```

First try the example provided by wandb

In [2]:

```python
import wandb
import random

# start a new wandb run to track this script
wandb.init(
    # set the wandb project where this run will be logged
    project="my-awesome-project",

    # track hyperparameters and run metadata
    config={
    "learning_rate": 0.02,
    "architecture": "CNN",
    "dataset": "CIFAR-100",
    "epochs": 10,
    }
)

# simulate training
epochs = 10
offset = random.random() / 5
for epoch in range(2, epochs):
    acc = 1 - 2 ** -epoch - random.random() / epoch - offset
    loss = 2 ** -epoch + random.random() / epoch + offset

    # log metrics to wandb
    wandb.log({"acc": acc, "loss": loss})

# [optional] finish the wandb run, necessary in notebooks
wandb.finish()
```

```
Failed to detect the name of this notebook, you can set it manually with the WAND
B_NOTEBOOK_NAME environment variable to enable code saving.
wandb: Currently logged in as: mingzwhy. Use `wandb login --relogin` to force
relogin
```

Tracking run with wandb version 0.15.12

Run data is saved locally in
`f:\new_gitee_code\berkeley_class\Deep_Learning\hw8\wandb\run-20231018_144542-ncz8rcp1`

Syncing run **woven-grass-1 (https://wandb.ai/mingzwhy/my-awesome-project/runs/ncz8rcp1)** to Weights & Biases (https://wandb.ai/mingzwhy/my-awesome-project) (docs (https://wandb.me/run))

View project at https://wandb.ai/mingzwhy/my-awesome-project (https://wandb.ai/mingzwhy/my-awesome-project)

View run at https://wandb.ai/mingzwhy/my-awesome-project/runs/ncz8rcp1 (https://wandb.ai/mingzwhy/my-awesome-project/runs/ncz8rcp1)

Waiting for W&B process to finish... **(success).**

```
VBox(children=(Label(value='0.001 MB of 0.012 MB uploaded (0.000 MB deduped)\r'),
FloatProgress(value=0.115196…
```

Run history:

acc 

loss 

Run summary:

acc	0.82327
loss	0.09432

View run **woven-grass-1** at: [https://wandb.ai/mingzwhy/my-awesome-project/runs/ncz8rcp1 (https://wandb.ai/mingzwhy/my-awesome-project/runs/ncz8rcp1)](https://wandb.ai/mingzwhy/my-awesome-project/runs/ncz8rcp1)
Synced 5 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)

Find logs at: `.\wandb\run-20231018_144542-ncz8rcp1\logs`

Organizing wandb Projects

With each run, you will want to have a set of parameters associated with it. For example, I want to be able to log different hyperparameters that I am using, so let's clearly list them below

In [3]:
```
project = 'CS182 WANDB.AI Practice Notebok'
learning_rate = 0.01
epochs = 2
architecture ='CNN'
dataset = 'CIFAR-10'
batch_size = 64
momentum = 0.9
log_freq = 20
print_freq = 200
cuda = torch.cuda.is_available()
device = torch.device("cuda" if cuda else "cpu")
```

Initializing the Run

In [4]:
```
wandb.init(
    # set the wandb project where this run will be logged
    project=project,

    # track hyperparameters and run metadata
    config={
    "learning_rate": learning_rate,
    "architecture": architecture,
    "dataset": dataset,
    "epochs": epochs,
    "batch_size": batch_size,
    "momentum": momentum
    }
)
```

Tracking run with wandb version 0.15.12

Run data is saved locally in `f:\new_gitee_code\berkeley_class\Deep_Learning\hw8\wandb\run-20231018_145146-erzbsuxc`

Syncing run **[pretty-universe-1 (https://wandb.ai/mingzwhy/CS182%20WANDB.AI%20Practice%20Notebok/runs/erzbsu**
to [Weights & Biases (https://wandb.ai/mingzwhy/CS182%20WANDB.AI%20Practice%20Notebok)] ([docs (https://wandb.me/run)])

View project at [https://wandb.ai/mingzwhy/CS182%20WANDB.AI%20Practice%20Notebok (https://wandb.ai/mingzwhy/CS182%20WANDB.AI%20Practice%20Notebok)]

View run at
[https://wandb.ai/mingzwhy/CS182%20WANDB.AI%20Practice%20Notebok/runs/erzbsuxc (https://wandb.ai/mingzwhy/CS182%20WANDB.AI%20Practice%20Notebok/runs/erzbsuxc)]

Out[4]: Display W&B run

From here on, we have some standard CIFAR training definitions.

In [6]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./../cifar-10/', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./../cifar-10/', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

In [7]:
```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

In [8]:
```
net = Net()
```

In [9]:
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=learning_rate, momentum=momentum)
```

Training with wandb

As you can see, similar to tensorboard, each gradient step we will want to log the accuracy and loss. See below for an example.

In [10]:
```python
for epoch in range(epochs):  # loop over the dataset multiple times
    running_loss = 0.0
    running_acc = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        accuracy = torch.mean((torch.argmax(outputs, dim=1) == labels).float()).iter

        # print statistics
        running_acc += accuracy
        running_loss += loss.item()
        if i % log_freq == log_freq - 1:
            wandb.log({'accuracy': accuracy, 'loss': loss.item()})

        if i % print_freq == print_freq - 1:    # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / print_freq:.5f}
            running_loss = 0.0
            running_acc = 0.0
```

```
[1,   200] loss: 2.25610 accuracy: 15.78125
[1,   400] loss: 1.90479 accuracy: 29.90625
[1,   600] loss: 1.69474 accuracy: 37.71875
[2,   200] loss: 1.48649 accuracy: 45.19531
[2,   400] loss: 1.43179 accuracy: 48.33594
[2,   600] loss: 1.38526 accuracy: 50.20312
```

After we are done with this run, we will want to call `wandb.finish()`

In [11]:
```python
wandb.finish()
```

Waiting for W&B process to finish... **(success).**

```
VBox(children=(Label(value='0.001 MB of 0.001 MB uploaded (0.000 MB deduped)\r'),
FloatProgress(value=1.0, max…
```

Run history:

Run summary:

accuracy	56.25
loss	1.31502

View run **pretty-universe-1** at:
[https://wandb.ai/mingzwhy/CS182%20WANDB.AI%20Practice%20Notebok/runs/erzbsuxc (https://wandb.ai/mingzwhy/CS182%20WANDB.AI%20Practice%20Notebok/runs/erzbsuxc)](https://wandb.ai/mingzwhy/CS182%20WANDB.AI%20Practice%20Notebok/runs/erzbsuxc)
Synced 6 W&B file(s), 0 media file(s), 0 artifact file(s) and 0 other file(s)

Find logs at: `.\wandb\run-20231018_145146-erzbsuxc\logs`

Your Task

We will be once again building classifiers for the CIFAR-10. There are various architectures set up for you to use in the architectures.py file. Using wandb, please search through 10 different hyperparameter configurations. Examples of choices include: learning rate, batch size, architecture, optimization algorithm, etc. Please submit the hyperparameters that result in the highest accuracies for this classification task. Please then explore wandb for all the visualization that you may need. In addition, feel free to run as many epochs as you like.

In [ ]:
```python
def run(params):
    raise NotImplementedError
```

In [13]:
```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import wandb
from architectures import BasicConvNet, ResNet18, MLP
from torch.utils.tensorboard import SummaryWriter
from tqdm import tqdm
from torch.utils.data import DataLoader
```

In [12]:
```python
device = torch.device("cuda" if cuda else "cpu")

transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./../cifar-10/', train=True,
                                        download=True, transform=transform)
testset = torchvision.datasets.CIFAR10(root='./../cifar-10/', train=False,
                                        download=True, transform=transform)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

In [18]:
```python
def get_optimizer(params, optim_type, lr):
    if optim_type == "sgd":
        optimizer = optim.SGD(params, lr=lr)
    elif optim_type == "adam":
        optimizer = optim.Adam(params, lr=lr)
    else:
        raise ValueError(optim_type)

    return optimizer

def get_model(model_type):
    if model_type == "basicconvnet":
        model = BasicConvNet()
    elif model_type == "resnet18":
        model = ResNet18()
    elif model_type == "mlp":
        model = MLP()
    else:
        raise ValueError(model_type)

    return model

def get_criterion(loss_type):
    if(loss_type == "mse"):
        criterion = nn.MSELoss()
    elif(loss_type == "cross"):
        criterion = nn.CrossEntropyLoss()
    else:
        raise ValueError(loss_type)

    return criterion
```

In [19]:

```
def train(dataloader, model, loss_fn, optimizer, epoch):
    size = len(dataloader.dataset)
    num_batch = len(dataloader)
    model.train()

    total_loss = 0
    correct = 0

    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        pred = model(X)
        loss = loss_fn(pred, y)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        correct += (pred.argmax(1) == y).type(torch.float).sum().item()

        if(batch % 100 == 0):
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f} [{current:>5d} / {size:>5d}]")

    avg_loss = total_loss / num_batch
    correct /= size

    # write into wandb
    wandb.log({'train accuracy': correct, 'train loss': avg_loss})

    print(f"Train Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {avg_loss:>8


def test(dataloader, model, loss_fn, epoch):

    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    model.eval()

    test_loss = 0
    correct = 0.1
    with torch.no_grad():
        for batch, (X, y) in enumerate(dataloader):
            X, y = X.cuda(), y.cuda()
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size

    # write into wandb
    wandb.log({'test accuracy': correct, 'test loss': test_loss})

    print(f"Evaluation Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_l
```

In [20]:
```python
def run_training(trainset, testset, hyperparameters, log_dir = "logs"):

    print("---------------------------config-----------------------------")
    print(hyperparameters)
    print("----------------------------------------------------------------")

    name = ""
    for i, key in enumerate(hyperparameters.keys()):
        value = hyperparameters[key]
        if i != (len(hyperparameters.keys()) - 1):
            item = key + "_" + str(value) + "_"
        else:
            item = key + "_" + str(value)
        name = name + item

    model_type = hyperparameters['model']
    model = get_model(model_type)
    loss_type = hyperparameters['loss_fn']
    criterion = get_criterion(loss_type)
    learning_rate = hyperparameters['lr']
    optim_type = hyperparameters['optimizer']
    optimizer = get_optimizer(model.parameters(), optim_type, lr=learning_rate)
    batch_size = hyperparameters['batch_size']
    num_epochs = hyperparameters['epochs']

    # build train data loader
    trainloader = DataLoader(trainset, batch_size=batch_size, shuffle=True)
    # build test data loader
    testloader = DataLoader(testset, batch_size=batch_size, shuffle=False)

    # create a wandb project

    wandb.init(
        # set the wandb project where this run will be logged
        project = name,

        # track hyperparameters and run metadata
        config={
        "learning_rate": learning_rate,
        "architecture": model_type,
        "dataset": 'CIFAR-10',
        "epochs": num_epochs,
        "batch_size": batch_size,
        }
    )

    print(f"log will be written to project {name}")

    model.cuda()

    for t in range(num_epochs):
        print(f"Epoch {t+1}\n-------------------------------")
        train(trainloader, model, criterion, optimizer, t+1)
        test(testloader, model, criterion, t+1)

    wandb.finish()
```

Type *Markdown* and LaTeX: α^2

In [22]:
```
hyperparameters1 = {
    "model" : "basicconvnet",
    "lr" : 0.0001,
    "loss_fn" : "cross",
    "optimizer" : "adam",
    "epochs" : 3,
    "batch_size" : 16
}
```

In [23]:
```
run_training(trainset, testset, hyperparameters1)
```

```
--------------------------config----------------------------
{'model': 'basicconvnet', 'lr': 0.0001, 'loss_fn': 'cross', 'optimizer': 'ada
m', 'epochs': 3, 'batch_size': 16}
-------------------------------------------------------------
```

Tracking run with wandb version 0.15.12

Run data is saved locally in
`f:\new_gitee_code\berkeley_class\Deep_Learning\hw8\wandb\run-20231018_150815-781t4xy1`

Syncing run **swept-cloud-1**
(https://wandb.ai/mingzwhy/model_basicconvnet_lr_0.0001_loss_fn_cross_optimizer
to Weights & Biases
(https://wandb.ai/mingzwhy/model_basicconvnet_lr_0.0001_loss_fn_cross_optimizer_adam
(docs (https://wandb.me/run))

View project at

In [ ]: