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)](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]:
```python
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                             #
    ##############################################################################
```

```python
    # 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]:
```
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]:
```python
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
```

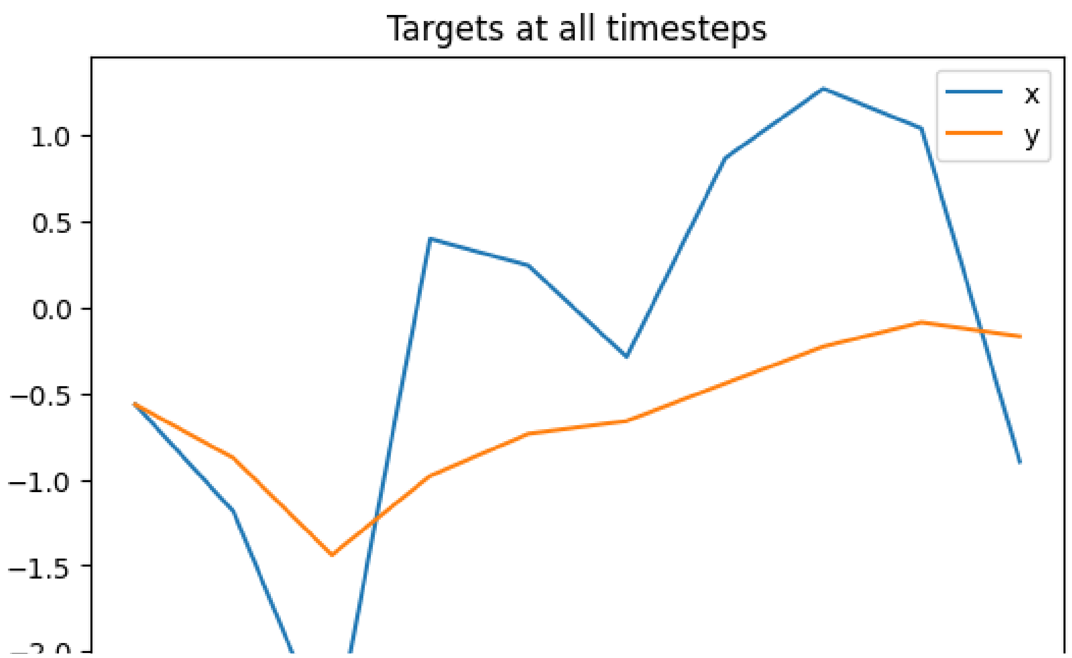
Problem 1.C: Dataset and loss function

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]:
```python
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
```

```
    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.

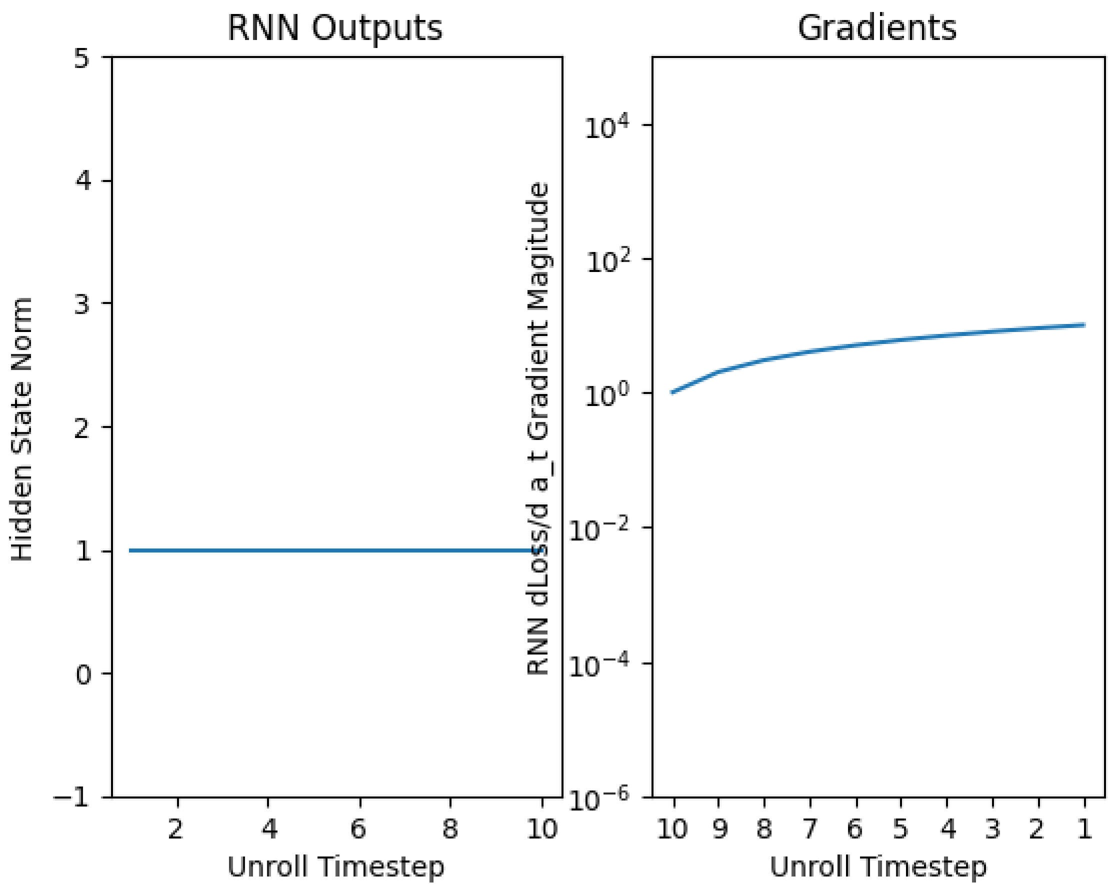
Parameters to try varying:

- 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
```

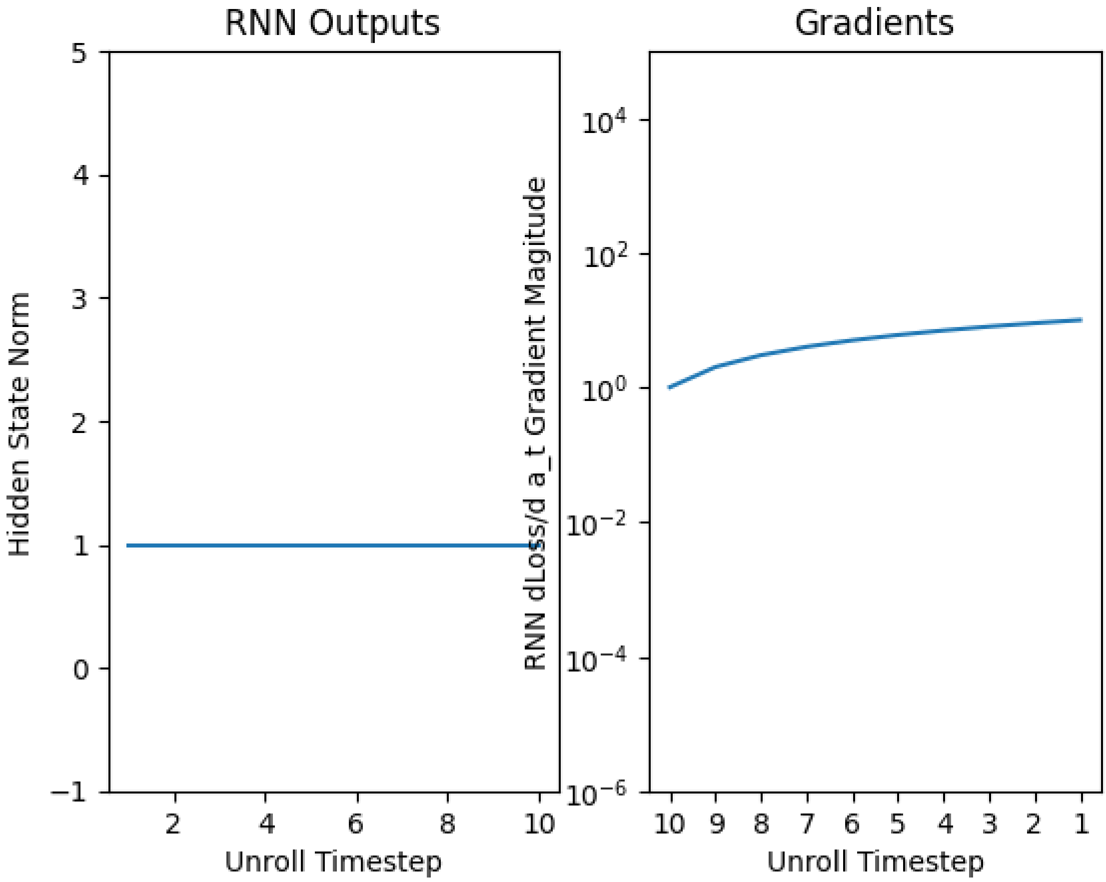
```
Data point: x=[ 1.51 -0.32  1.36  0.72 -0.19  0.13 -0.19 -0.43 -0.65 -0.95], y=
[1.51 0.59 0.85 0.82 0.61 0.53 0.43 0.32 0.21 0.1 ]
```



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
```

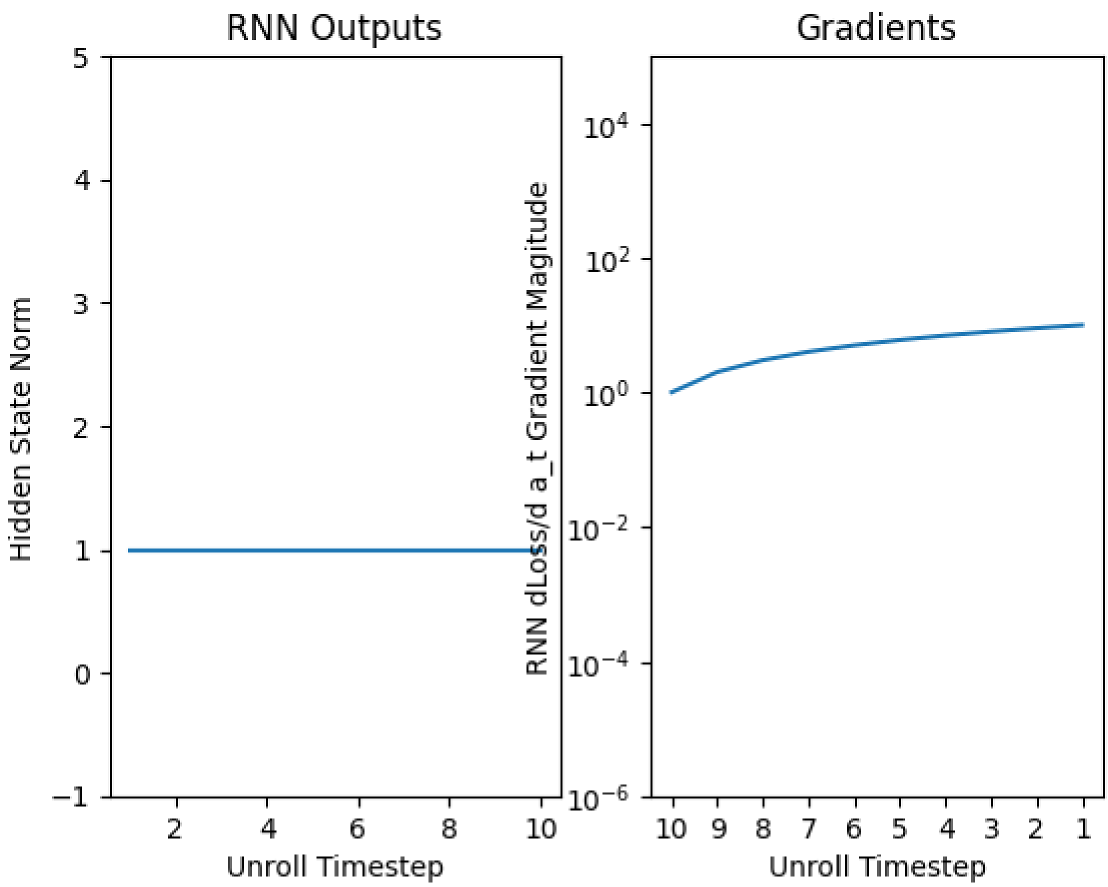
```
Data point: x=[ 1.92 -0.3  -0.28 -0.15 -0.75  1.45  0.52  1.14  2.27 -0.41], y=
[1.92 0.81 0.45 0.3  0.09 0.31 0.34 0.44 0.65 0.54]
```



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]:
```
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)
```

```python
        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.

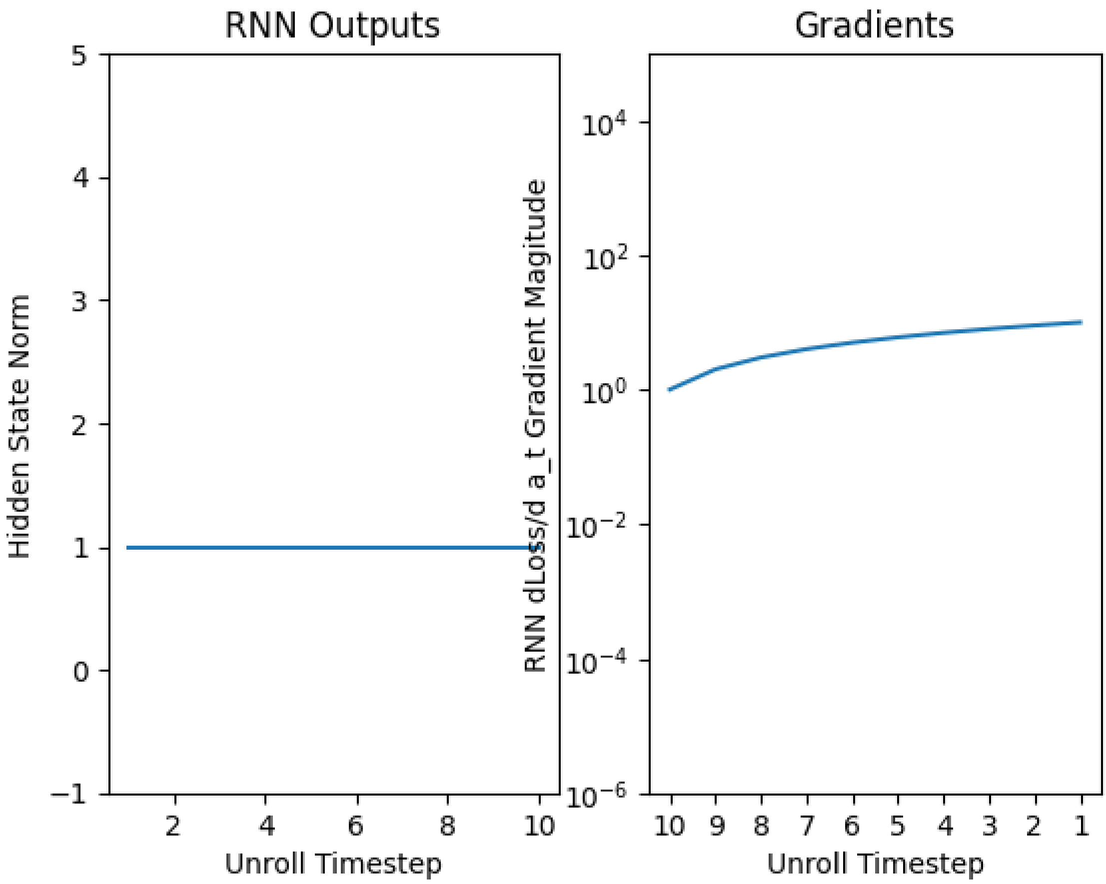
In [36]:
```python
lstm = LSTMLayer(2, 3)
lstm.load_state_dict({k: v * 0 - .1 for k, v in lstm.state_dict().items()})
data = th.FloatTensor([[[.1, .15], [.2, .25], [.3, .35], [.4, .45]], [[-.1, -1.5], [
expected_all_h = th.FloatTensor([[[-0.0273, -0.0273, -0.0273],
         [-0.0420, -0.0420, -0.0420],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

```python
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
```

```python
        ##############################################################################
        # 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)
```

```
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 [ ]:

```
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 [00:19<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