Image Captioning with RNNs

In this exercise you will implement a vanilla recurrent neural networks and use them it to train a model that can generate novel captions for images.

Install starter code

We will continue using the utility functions that we've used for previous assignments: [coutils package](). Run this cell to download and install it.

```
!pip install git+https://github.com/deepvision-class/starter-code
```

```
Collecting git+https://github.com/deepvision-class/starter-code
  Cloning https://github.com/deepvision-class/starter-code to /tmp/pip-req-build-ix_1
  Running command git clone -q https://github.com/deepvision-class/starter-code /tmp/
Requirement already satisfied: pydrive in /usr/local/lib/python3.6/dist-packages (fro
Requirement already satisfied: oauth2client>=4.0.0 in /usr/local/lib/python3.6/dist-p
Requirement already satisfied: google-api-python-client>=1.2 in /usr/local/lib/python
Requirement already satisfied: PyYAML>=3.0 in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: rsa>=3.1.4 in /usr/local/lib/python3.6/dist-packages (
Requirement already satisfied: pyasn1>=0.1.7 in /usr/local/lib/python3.6/dist-package
Requirement already satisfied: pyasn1-modules>=0.0.5 in /usr/local/lib/python3.6/dist
Requirement already satisfied: six>=1.6.1 in /usr/local/lib/python3.6/dist-packages (
Requirement already satisfied: httplib2>=0.9.1 in /usr/local/lib/python3.6/dist-packa
Requirement already satisfied: google-auth>=1.4.1 in /usr/local/lib/python3.6/dist-pa
Requirement already satisfied: google-auth-httplib2>=0.0.3 in /usr/local/lib/python3
Requirement already satisfied: uritemplate<4dev,>=3.0.0 in /usr/local/lib/python3.6/d
Requirement already satisfied: setuptools>=40.3.0 in /usr/local/lib/python3.6/dist-pa
Requirement already satisfied: cachetools<5.0,>=2.0.0 in /usr/local/lib/python3.6/dis
Building wheels for collected packages: Colab-Utils
  Building wheel for Colab-Utils (setup.py) ... done
  Created wheel for Colab-Utils: filename=Colab_Utils-0.1.dev0-cp36-none-any.whl size
  Stored in directory: /tmp/pip-ephem-wheel-cache-j57fy6uc/wheels/63/d1/27/a208931527
Successfully built Colab-Utils
Installing collected packages: Colab-Utils
Successfully installed Colab-Utils-0.1.dev0
```

Setup code

Run some setup code for this notebook: Import some useful packages and increase the default figure size.

```
import math
import torch
from torch import nn
from torch.nn.parameter import Parameter
import torch.nn.functional as F
import coutils
from coutils import fix_random_seed, rel_error, compute_numeric_gradient, \
```

```
                    tensor_to_image, decode_captions, attention_visualizer
import matplotlib.pyplot as plt
import time

# for plotting
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# data type and device for torch.tensor
to_float = {'dtype': torch.float, 'device': 'cpu'}
to_float_cuda = {'dtype': torch.float, 'device': 'cuda'}
to_double = {'dtype': torch.double, 'device': 'cpu'}
to_double_cuda = {'dtype': torch.double, 'device': 'cuda'}
to_long = {'dtype': torch.long, 'device': 'cpu'}
to_long_cuda = {'dtype': torch.long, 'device': 'cuda'}
```

We will use GPUs to accelerate our computation in this notebook. Run the following to make sure GPUs are enabled:

```
if torch.cuda.is_available:
  print('Good to go!')
else:
  print('Please set GPU via Edit -> Notebook Settings.')
```

```
Good to go!
```

Microsoft COCO

For this exercise we will use the 2014 release of the [Microsoft COCO dataset](http://mscoco.org/) which has become the standard testbed for image captioning. The dataset consists of 80,000 training images and 40,000 validation images, each annotated with 5 captions written by workers on Amazon Mechanical Turk.

We have preprocessed the data for you already and saved them into a serialized data file. It contains 10,000 image-caption pairs for training and 500 for testing. The images have been downsampled to 112x112 for computation efficiency and captions are tokenized and numericalized, clamped to 15 words. You can download the file named `coco.pt` (378MB) with the link below and run some useful stats.

You will later use MobileNet v2 to extract features for the images. A few notes on the caption preprocessing:

Dealing with strings is inefficient, so we will work with an encoded version of the captions. Each word is assigned an integer ID, allowing us to represent a caption by a sequence of integers. The mapping between integer IDs and words is saved in an entry named `vocab` (both `idx_to_token`

and `token_to_idx`), and we use the function `decode_captions` from the starter code to convert tensors of integer IDs back into strings.

There are a couple special tokens that we add to the vocabulary. We prepend a special `<START>` token and append an `<END>` token to the beginning and end of each caption respectively. Rare words are replaced with a special `<UNK>` token (for "unknown"). In addition, since we want to train with minibatches containing captions of different lengths, we pad short captions with a special `<NULL>` token after the `<END>` token and don't compute loss or gradient for `<NULL>` tokens. Since they are a bit of a pain, we have taken care of all implementation details around special tokens for you.

```
# Download and load serialized COCO data from coco.pt
# It contains a dictionary of
# "train_images" - resized training images (112x112)
# "val_images" - resized validation images (112x112)
# "train_captions" - tokenized and numericalized training captions
# "val_captions" - tokenized and numericalized validation captions
# "vocab" - caption vocabulary, including "idx_to_token" and "token_to_idx"

!wget http://web.eecs.umich.edu/~justincj/teaching/eecs498/coco.pt
data_dict = torch.load('coco.pt')

# print out all the keys and values from the data dictionary
for k, v in data_dict.items():
    if type(v) == torch.Tensor:
        print(k, type(v), v.shape, v.dtype)
    else:
        print(k, type(v), v.keys())

num_train = data_dict['train_images'].size(0)
num_val = data_dict['val_images'].size(0)
assert data_dict['train_images'].size(0) == data_dict['train_captions'].size(0) and \
       data_dict['val_images'].size(0) == data_dict['val_captions'].size(0), \
       'shapes of data mismatch!'

print('\nTrain images shape: ', data_dict['train_images'].shape)
print('Train caption tokens shape: ', data_dict['train_captions'].shape)
print('Validation images shape: ', data_dict['val_images'].shape)
print('Validation caption tokens shape: ', data_dict['val_captions'].shape)
print('total number of caption tokens: ', len(data_dict['vocab']['idx_to_token']))
print('mappings (list) from index to caption token: ', data_dict['vocab']['idx_to_token'])
print('mappings (dict) from caption token to index: ', data_dict['vocab']['token_to_idx'])

# declare variables for special tokens
NULL_index = data_dict['vocab']['token_to_idx']['<NULL>']
START_index = data_dict['vocab']['token_to_idx']['<START>']
END_index = data_dict['vocab']['token_to_idx']['<END>']
UNK_index = data_dict['vocab']['token_to_idx']['<UNK>']
```

```
--2020-09-13 01:31:58--  http://web.eecs.umich.edu/~justincj/teaching/eecs498/coco.pt
Resolving web.eecs.umich.edu (web.eecs.umich.edu)... 141.212.113.214
Connecting to web.eecs.umich.edu (web.eecs.umich.edu)|141.212.113.214|:80... connecte
HTTP request sent, awaiting response... 200 OK
Length: 396583632 (378M)
Saving to: ‘coco.pt’

coco.pt             100%[===================>] 378.21M  40.6MB/s    in 11s

2020-09-13 01:32:09 (33.8 MB/s) - ‘coco.pt’ saved [396583632/396583632]

train_images <class 'torch.Tensor'> torch.Size([10000, 3, 112, 112]) torch.uint8
train_captions <class 'torch.Tensor'> torch.Size([10000, 17]) torch.int64
val_images <class 'torch.Tensor'> torch.Size([500, 3, 112, 112]) torch.uint8
val_captions <class 'torch.Tensor'> torch.Size([500, 17]) torch.int64
vocab <class 'dict'> dict_keys(['idx_to_token', 'token_to_idx'])

Train images shape:  torch.Size([10000, 3, 112, 112])
Train caption tokens shape:  torch.Size([10000, 17])
Validation images shape:  torch.Size([500, 3, 112, 112])
Validation caption tokens shape:  torch.Size([500, 17])
total number of caption tokens:  864
mappings (list) from index to caption token:  ['<NULL>', '<START>', '<END>', '<UNK>',
```

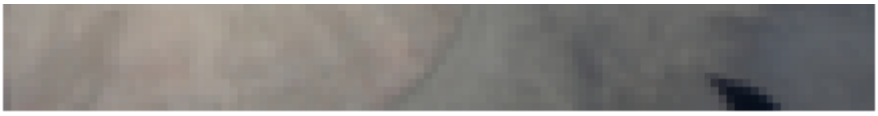
Look at the data

It is always a good idea to look at examples from the dataset before working with it.

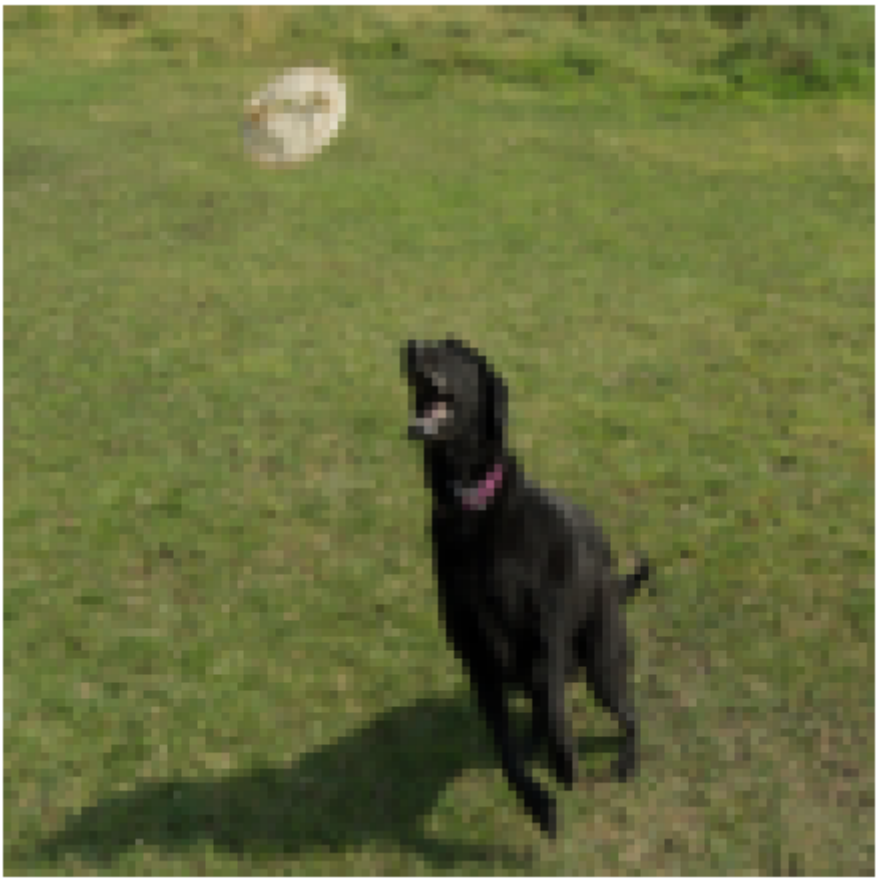
Run the following to sample a small minibatch of training data and show the images and their captions. Running it multiple times and looking at the results helps you to get a sense of the dataset.

Note that we decode the captions using the `decode_captions` function.

```
# Sample a minibatch and show the reshaped 112x112 images and captions
batch_size = 3

sample_idx = torch.randint(0, num_train, (batch_size,))
sample_images = data_dict['train_images'][sample_idx]
sample_captions = data_dict['train_captions'][sample_idx]
for i in range(batch_size):
  plt.imshow(sample_images[i].permute(1, 2, 0))
  plt.axis('off')
  caption_str = decode_captions(sample_captions[i], data_dict['vocab']['idx_to_token'])
  plt.title(caption_str)
  plt.show()
```



<START> a black dog is jumping in the air to catch a frisbee <END>



<START> a man and women who are underneath a purple umbrella <END>

Recurrent Neural Networks

As discussed in lecture, we will use Recurrent Neural Network (RNN) language models for image captioning. We will cover the vanilla RNN model first and later LSTM and attention-based language models.

Vanilla RNN: step forward

First implement the forward pass for a single timestep of a vanilla recurrent neural network.

```
def rnn_step_forward(x, prev_h, Wx, Wh, b):
    """
    Run the forward pass for a single timestep of a vanilla RNN that uses a tanh
    activation function.

    The input data has dimension D, the hidden state has dimension H, and we use
    a minibatch size of N.

    Inputs:
    - x: Input data for this timestep, of shape (N, D).
    - prev_h: Hidden state from previous timestep, of shape (N, H)
    - Wx: Weight matrix for input-to-hidden connections, of shape (D, H)
    - Wh: Weight matrix for hidden-to-hidden connections, of shape (H, H)
    - b: Biases, of shape (H,)

    Returns a tuple of:
    - next_h: Next hidden state, of shape (N, H)
    - cache: Tuple of values needed for the backward pass.
    """
    next_h, cache = None, None
    ##############################################################################
    # TODO: Implement a single forward step for the vanilla RNN. Store the next  #
    # hidden state and any values you need for the backward pass in the next_h   #
    # and cache variables respectively.                                          #
    ##############################################################################
    # Replace "pass" statement with your code
    next_h = torch.tanh(x.mm(Wx) + prev_h.mm(Wh) + b )
    cache = (x, prev_h, Wx, Wh, next_h)
    ##############################################################################
    #                               END OF YOUR CODE                             #
```

```
    #                                END OF YOUR CODE                            #
    ##############################################################################
    return next_h, cache
```

Run the following to check your implementation. You should see errors on the order of 1e-8 or less.

```
N, D, H = 3, 10, 4

x = torch.linspace(-0.4, 0.7, steps=N*D, **to_double_cuda).reshape(N, D)
prev_h = torch.linspace(-0.2, 0.5, steps=N*H, **to_double_cuda).reshape(N, H)
Wx = torch.linspace(-0.1, 0.9, steps=D*H, **to_double_cuda).reshape(D, H)
Wh = torch.linspace(-0.3, 0.7, steps=H*H, **to_double_cuda).reshape(H, H)
b = torch.linspace(-0.2, 0.4, steps=H, **to_double_cuda)

next_h, _ = rnn_step_forward(x, prev_h, Wx, Wh, b)
expected_next_h = torch.tensor([
  [-0.58172089, -0.50182032, -0.41232771, -0.31410098],
  [ 0.66854692,  0.79562378,  0.87755553,  0.92795967],
  [ 0.97934501,  0.99144213,  0.99646691,  0.99854353]], **to_double_cuda)

print('next_h error: ', rel_error(expected_next_h, next_h))
```

```
next_h error:  1.258484277375294e-08
```

Vanilla RNN: step backward

Then implement the backward pass for a single timestep of a vanilla recurrent neural network.

```
def rnn_step_backward(dnext_h, cache):
    """
    Backward pass for a single timestep of a vanilla RNN.

    Inputs:
    - dnext_h: Gradient of loss with respect to next hidden state, of shape (N, H)
    - cache: Cache object from the forward pass

    Returns a tuple of:
    - dx: Gradients of input data, of shape (N, D)
    - dprev_h: Gradients of previous hidden state, of shape (N, H)
    - dWx: Gradients of input-to-hidden weights, of shape (D, H)
    - dWh: Gradients of hidden-to-hidden weights, of shape (H, H)
    - db: Gradients of bias vector, of shape (H,)
    """
    dx, dprev_h, dWx, dWh, db = None, None, None, None, None
    ##############################################################################
    # TODO: Implement the backward pass for a single step of a vanilla RNN.      #
    #                                                                            #
    # HINT: For the tanh function, you can compute the local derivative in terms #
    # of the output value from tanh.                                             #
    ##############################################################################
    # Replace "pass" statement with your code
```

```
    # Replace "pass" statement with your code
    (x, prev_h, Wx, Wh, next_h) = cache
    dnon_tanhx = (dnext_h) *  (1 - next_h ** 2) #just before the tanhx
    dx = dnon_tanhx.mm(Wx.t())
    dWx = x.t().mm(dnon_tanhx)
    dprev_h = dnon_tanhx.mm(Wh.t())
    dWh = prev_h.t().mm(dnon_tanhx)
    db = torch.sum(dnon_tanhx, axis = 0 )
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################
    return dx, dprev_h, dWx, dWh, db
```

Run the following to numerically gradient check your implementation. You should see errors on the order of `1e-8` or less.

```
fix_random_seed(0)
N, D, H = 4, 5, 6
x = torch.randn(N, D, **to_double_cuda)
h = torch.randn(N, H, **to_double_cuda)
Wx = torch.randn(D, H, **to_double_cuda)
Wh = torch.randn(H, H, **to_double_cuda)
b = torch.randn(H, **to_double_cuda)

out, cache = rnn_step_forward(x, h, Wx, Wh, b)

dnext_h = torch.randn(*out.shape, **to_double_cuda)

fx = lambda x: rnn_step_forward(x, h, Wx, Wh, b)[0]
fh = lambda h: rnn_step_forward(x, h, Wx, Wh, b)[0]
fWx = lambda Wx: rnn_step_forward(x, h, Wx, Wh, b)[0]
fWh = lambda Wh: rnn_step_forward(x, h, Wx, Wh, b)[0]
fb = lambda b: rnn_step_forward(x, h, Wx, Wh, b)[0]

dx_num = compute_numeric_gradient(fx, x, dnext_h)
dprev_h_num = compute_numeric_gradient(fh, h, dnext_h)
dWx_num = compute_numeric_gradient(fWx, Wx, dnext_h)
dWh_num = compute_numeric_gradient(fWh, Wh, dnext_h)
db_num = compute_numeric_gradient(fb, b, dnext_h)

dx, dprev_h, dWx, dWh, db = rnn_step_backward(dnext_h, cache)

print('dx error: ', rel_error(dx_num, dx))
print('dprev_h error: ', rel_error(dprev_h_num, dprev_h))
print('dWx error: ', rel_error(dWx_num, dWx))
print('dWh error: ', rel_error(dWh_num, dWh))
print('db error: ', rel_error(db_num, db))
```

Vanilla RNN: forward

Now that you have implemented the forward and backward passes for a single timestep of a vanilla RNN, you will combine these pieces to implement a RNN that processes an entire sequence of data. First implement the forward pass by making calls to the `rnn_step_forward` function that you defined earlier.

```
def rnn_forward(x, h0, Wx, Wh, b):
    """
    Run a vanilla RNN forward on an entire sequence of data. We assume an input
    sequence composed of T vectors, each of dimension D. The RNN uses a hidden
    size of H, and we work over a minibatch containing N sequences. After running
    the RNN forward, we return the hidden states for all timesteps.

    Inputs:
    - x: Input data for the entire timeseries, of shape (N, T, D).
    - h0: Initial hidden state, of shape (N, H)
    - Wx: Weight matrix for input-to-hidden connections, of shape (D, H)
    - Wh: Weight matrix for hidden-to-hidden connections, of shape (H, H)
    - b: Biases, of shape (H,)

    Returns a tuple of:
    - h: Hidden states for the entire timeseries, of shape (N, T, H).
    - cache: Values needed in the backward pass
    """
    h, cache = None, None
    ##############################################################################
    # TODO: Implement forward pass for a vanilla RNN running on a sequence of    #
    # input data. You should use the rnn_step_forward function that you defined  #
    # above. You can use a for loop to help compute the forward pass.            #
    ##############################################################################
    # Replace "pass" statement with your code
    N, T, D = x.shape
    H = h0.shape[1]
    h = torch.zeros((N, T, H), dtype=h0.dtype, device=h0.device)
    prev_h = h0
    cache = []
    for i in range(T):
      next_h, cache_h = rnn_step_forward(x[:,i,:], prev_h, Wx, Wh, b)
      prev_h = next_h
      h[:, i, :] = prev_h
      cache.append(cache_h)
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################
    return h, cache
```

Run the following to check your implementation. You should see errors on the order of `1e-6` or less.

```
N, T, D, H = 2, 3, 4, 5

x = torch.linspace(-0.1, 0.3, steps=N*T*D, **to_double_cuda).reshape(N, T, D)
h0 = torch.linspace(-0.3, 0.1, steps=N*H, **to_double_cuda).reshape(N, H)
Wx = torch.linspace(-0.2, 0.4, steps=D*H, **to_double_cuda).reshape(D, H)
Wh = torch.linspace(-0.4, 0.1, steps=H*H, **to_double_cuda).reshape(H, H)
b = torch.linspace(-0.7, 0.1, steps=H, **to_double_cuda)

h, _ = rnn_forward(x, h0, Wx, Wh, b)
expected_h = torch.tensor([
  [
    [-0.42070749, -0.27279261, -0.11074945,  0.05740409,  0.22236251],
    [-0.39525808, -0.22554661, -0.0409454,   0.14649412,  0.32397316],
    [-0.42305111, -0.24223728, -0.04287027,  0.15997045,  0.35014525],
  ],
  [
    [-0.55857474, -0.39065825, -0.19198182,  0.02378408,  0.23735671],
    [-0.27150199, -0.07088804,  0.13562939,  0.33099728,  0.50158768],
    [-0.51014825, -0.30524429, -0.06755202,  0.17806392,  0.40333043]]], **to_double_cuda)
print('h error: ', rel_error(expected_h, h))
```

Vanilla RNN: backward

Implement the backward pass for a vanilla RNN in the function `rnn_backward`. This should run back-propagation over the entire sequence, making calls to the `rnn_step_backward` function that you defined earlier.

```
def rnn_backward(dh, cache):
    """
    Compute the backward pass for a vanilla RNN over an entire sequence of data.

    Inputs:
    - dh: Upstream gradients of all hidden states, of shape (N, T, H).

    NOTE: 'dh' contains the upstream gradients produced by the
    individual loss functions at each timestep, *not* the gradients
    being passed between timesteps (which you'll have to compute yourself
    by calling rnn_step_backward in a loop).

    Returns a tuple of:
    - dx: Gradient of inputs, of shape (N, T, D)
    - dh0: Gradient of initial hidden state, of shape (N, H)
    - dWx: Gradient of input-to-hidden weights, of shape (D, H)
    - dWh: Gradient of hidden-to-hidden weights, of shape (H, H)
    - db: Gradient of biases, of shape (H,)
    """
    dx, dh0, dWx, dWh, db = None, None, None, None, None
    ##############################################################################
    # TODO: Implement the backward pass for a vanilla RNN running an entire      #
    # sequence of data. You should use the rnn_step_backward function that you   #
    # defined above. You can use a for loop to help compute the backward pass    #
```

```
    # defined above. You can use a for loop to help compute the backward pass.   #
    ##############################################################################
    # Replace "pass" statement with your code
    for i in range(dh.shape[1]-1, -1, -1):  #iterate from the last in reverse order
      if (i == dh.shape[1] - 1):
        dprev_h_tstp = dh[:,i,:]
      else:
        dprev_h_tstp += dh[:,i,:]
      dx_tstp, dprev_h_tstp, dWx_tstp, dWh_tstp, db_tstp = rnn_step_backward(dprev_h_tstp,
      if (i == dh.shape[1] - 1):
        dx = torch.zeros([dh.shape[0], dh.shape[1], dWx_tstp.shape[0]]).to(x.dtype).to(x.d
        dh0 = torch.zeros([dh.shape[0], dh.shape[2]]).to(x.dtype).to(x.device)
        dWx = torch.zeros([dWx_tstp.shape[0], dWh_tstp.shape[0]]).to(x.dtype).to(x.device)
        dWh = torch.zeros([dWh_tstp.shape[0], dWh_tstp.shape[0]]).to(x.dtype).to(x.device)
        db = torch.zeros(dWh_tstp.shape[0]).to(x.dtype).to(x.device)
      dx[:,i,:] = dx_tstp
      dh0 = dprev_h_tstp
      dWx += dWx_tstp
      dWh += dWh_tstp
      db += db_tstp
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################
    return dx, dh0, dWx, dWh, db
```

You should see errors on the order of 1e-7 or less.

```
fix_random_seed(0)

N, D, T, H = 2, 3, 10, 5

x = torch.randn(N, T, D, **to_double_cuda)
h0 = torch.randn(N, H, **to_double_cuda)
Wx = torch.randn(D, H, **to_double_cuda)
Wh = torch.randn(H, H, **to_double_cuda)
b = torch.randn(H, **to_double_cuda)

out, cache = rnn_forward(x, h0, Wx, Wh, b)

dout = torch.randn(*out.shape, **to_double_cuda)

dx, dh0, dWx, dWh, db = rnn_backward(dout, cache)

fx = lambda x: rnn_forward(x, h0, Wx, Wh, b)[0]
fh0 = lambda h0: rnn_forward(x, h0, Wx, Wh, b)[0]
fWx = lambda Wx: rnn_forward(x, h0, Wx, Wh, b)[0]
fWh = lambda Wh: rnn_forward(x, h0, Wx, Wh, b)[0]
fb = lambda b: rnn_forward(x, h0, Wx, Wh, b)[0]

dx_num = compute_numeric_gradient(fx, x, dout)
dh0_num = compute_numeric_gradient(fh0, h0, dout)
dWx_num = compute_numeric_gradient(fWx, Wx, dout)
dWh_num = compute_numeric_gradient(fWh, Wh, dout)
db_num = compute_numeric_gradient(fb, b, dout)
```

```
print('dx error: ', rel_error(dx_num, dx))
print('dh0 error: ', rel_error(dh0_num, dh0))
print('dWx error: ', rel_error(dWx_num, dWx))
print('dWh error: ', rel_error(dWh_num, dWh))
print('db error: ', rel_error(db_num, db))
```

Vanilla RNN: backward with autograd

Now it's time to introduce the lifesaver PyTorch Automatic Differantiation package - `torch.autograd`!

`torch.autograd` provides classes and functions implementing **automatic differentiation** of arbitrary scalar valued functions. It requires minimal changes to the existing code - if you pass tensors with `requires_grad=True` to the forward function you wrote earlier, you can just call `.backward(gradient=grad)` on the output to compute gradients on the input and weights.

Now we can compare the manual backward pass with the autograd backward pass. **Read through the following.** You should get a relative error less than 1e-12.

```
fix_random_seed(0)

N, D, T, H = 2, 3, 10, 5

# set requires_grad=True
x = torch.randn(N, T, D, **to_double_cuda, requires_grad=True)
h0 = torch.randn(N, H, **to_double_cuda, requires_grad=True)
Wx = torch.randn(D, H, **to_double_cuda, requires_grad=True)
Wh = torch.randn(H, H, **to_double_cuda, requires_grad=True)
b = torch.randn(H, **to_double_cuda, requires_grad=True)

out, cache = rnn_forward(x, h0, Wx, Wh, b)

dout = torch.randn(*out.shape, **to_double_cuda)

# manual backward
with torch.no_grad():
  dx, dh0, dWx, dWh, db = rnn_backward(dout, cache)

# backward with autograd
out.backward(dout) # the magic happens here!
dx_auto, dh0_auto, dWx_auto, dWh_auto, db_auto = \
  x.grad, h0.grad, Wx.grad, Wh.grad, b.grad

print('dx error: ', rel_error(dx_auto, dx))
```

```
print('dh0 error: ', rel_error(dh0_auto, dh0))
print('dWx error: ', rel_error(dWx_auto, dWx))
print('dWh error: ', rel_error(dWh_auto, dWh))
print('db error: ', rel_error(db_auto, db))
```

RNN Module

We can now wrap the vanilla RNN we wrote into an `nn.Module`. `nn.Module` is a base class for all neural network modules, more details regarding its attributes, functions, and methods could be found here.

Here we want to set up a module for RNN, where function `__init__` sets up weight and biases, and function `forward` call the `rnn_forward` function from before.

We have written this part for you but you are highly recommended to go through the code as you will write `modules` on your own later.

All the implementation will be with `autograd` and `nn.Module` going forward.

```
class RNN(nn.Module):
  """
  A single-layer vanilla RNN module.

  Arguments for initialization:
  - input_size: Input size, denoted as D before
  - hidden_size: Hidden size, denoted as H before
  """
  def __init__(self, input_size, hidden_size, device='cpu',
               dtype=torch.float32):
    """
    Initialize a RNN.
    Model parameters to initialize:
    - Wx: Weight matrix for input-to-hidden connections, of shape (D, H)
    - Wh: Weight matrix for hidden-to-hidden connections, of shape (H, H)
    - b: Biases, of shape (H,)
    """
    super().__init__()

    # Register parameters
    self.Wx = Parameter(torch.randn(input_size, hidden_size,
                        device=device, dtype=dtype).div(math.sqrt(input_size)))
    self.Wh = Parameter(torch.randn(hidden_size, hidden_size,
                        device=device, dtype=dtype).div(math.sqrt(hidden_size)))
    self.b = Parameter(torch.zeros(hidden_size,
                        device=device, dtype=dtype))
```

```
  def forward(self, x, h0):
    """
    Inputs:
    - x: Input data for the entire timeseries, of shape (N, T, D)
    - h0: Initial hidden state, of shape (N, H)

    Outputs:
    - hn: The hidden state output
    """
    hn, _ = rnn_forward(x, h0, self.Wx, self.Wh, self.b)
    return hn

  def step_forward(self, x, prev_h):
    """
    Inputs:
    - x: Input data for one time step, of shape (N, D)
    - prev_h: The previous hidden state, of shape (N, H)

    Outputs:
    - next_h: The next hidden state, of shape (N, H)
    """
    next_h, _ = rnn_step_forward(x, prev_h, self.Wx, self.Wh, self.b)
    return next_h
```

RNN for image captioning

You will implement a few necessary tools and layers in order to build an image captioning model (class `CaptioningRNN`).

Image Feature Extraction

Here, we use MobileNet v2 for image feature extraction. For vanilla RNN and LSTM, we use the pooled CNN feature activation. For Attention LSTM, we use the CNN feature activation map after the last convolution layer.

```
# !pip install torchsummary

class FeatureExtractor(object):
  """
  Image feature extraction with MobileNet.
  """
  def __init__(self, pooling=False, verbose=False,
               device='cpu', dtype=torch.float32):

    from torchvision import transforms, models
    from torchsummary import summary
    self.preprocess = transforms.Compose([
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ])
```

```
    self.device, self.dtype = device, dtype
    self.mobilenet = models.mobilenet_v2(pretrained=True).to(device)
    self.mobilenet = nn.Sequential(*list(self.mobilenet.children())[:-1]) # Remove the las

    # average pooling
    if pooling:
      self.mobilenet.add_module('LastAvgPool', nn.AvgPool2d(4, 4)) # input: N x 1280 x 4 x

    self.mobilenet.eval()
    if verbose:
      summary(self.mobilenet, (3, 112, 112))

  def extract_mobilenet_feature(self, img, verbose=False):
    """
    Inputs:
    - img: Batch of resized images, of shape N x 3 x 112 x 112

    Outputs:
    - feat: Image feature, of shape N x 1280 (pooled) or N x 1280 x 4 x 4
    """
    num_img = img.shape[0]

    img_prepro = []
    for i in range(num_img):
      img_prepro.append(self.preprocess(img[i].type(self.dtype).div(255.)))
    img_prepro = torch.stack(img_prepro).to(self.device)

    with torch.no_grad():
      feat = []
      process_batch = 500
      for b in range(math.ceil(num_img/process_batch)):
        feat.append(self.mobilenet(img_prepro[b*process_batch:(b+1)*process_batch]
                                   ).squeeze(-1).squeeze(-1)) # forward and squeeze
      feat = torch.cat(feat)

      # add l2 normalization
      F.normalize(feat, p=2, dim=1)

    if verbose:
      print('Output feature shape: ', feat.shape)

    return feat
```

Now, let's see what's inside MobileNet v2. Assume we have a 3x112x112 image input. We pass argument `pooling=True` to the model so the CNN activation is spatially-pooled from `1280x4x4` to `1280`.

```
model = FeatureExtractor(pooling=True, verbose=True, device='cuda')
```

Word embedding

In deep learning systems, we commonly represent words using vectors. Each word of the vocabulary will be associated with a vector, and these vectors will be learned jointly with the rest

of the system.

Implement the module `WordEmbedding` to convert words (represented by integers) into vectors.

```
class WordEmbedding(nn.Module):
  """
  Simplified version of torch.nn.Embedding.

  We operate on minibatches of size N where
  each sequence has length T. We assume a vocabulary of V words, assigning each
  word to a vector of dimension D.

  Inputs:
  - x: Integer array of shape (N, T) giving indices of words. Each element idx
    of x muxt be in the range 0 <= idx < V.

  Returns a tuple of:
  - out: Array of shape (N, T, D) giving word vectors for all input words.
  """
  def __init__(self, vocab_size, embed_size,
               device='cpu', dtype=torch.float32):
      super().__init__()

      # Register parameters
      self.W_embed = Parameter(torch.randn(vocab_size, embed_size,
                          device=device, dtype=dtype).div(math.sqrt(vocab_size)))

  def forward(self, x):

      out = None
      ##############################################################################
      # TODO: Implement the forward pass for word embeddings.                      #
      #                                                                            #
      # HINT: This can be done in one line using PyTorch's array indexing.           #
      ##############################################################################
      # Replace "pass" statement with your code
      out = self.W_embed[x]
      ##############################################################################
      #                               END OF YOUR CODE                             #
      ##############################################################################
      return out
```

Run the following to check your implementation. You should see an error on the order of `1e-7` or less.

```
N, T, V, D = 2, 4, 5, 3

x = torch.tensor([[0, 3, 1, 2], [2, 1, 0, 3]], **to_long_cuda)
W = torch.linspace(0, 1, steps=V*D, **to_double_cuda).reshape(V, D)

model_emb = WordEmbedding(V, D, **to_double_cuda)
model_emb.W_embed.data.copy_(W)
out = model_emb(x)
```

```
expected_out = torch.tensor([
 [[ 0.,          0.07142857,  0.14285714],
  [ 0.64285714,  0.71428571,  0.78571429],
  [ 0.21428571,  0.28571429,  0.35714286],
  [ 0.42857143,  0.5,         0.57142857]],
 [[ 0.42857143,  0.5,         0.57142857],
  [ 0.21428571,  0.28571429,  0.35714286],
  [ 0.,          0.07142857,  0.14285714],
  [ 0.64285714,  0.71428571,  0.78571429]]], **to_double_cuda)

print('out error: ', rel_error(expected_out, out))
```

(Temporal) Affine layer

At every timestep we use an affine function to transform the RNN hidden vector at that timestep into scores for each word in the vocabulary. This could be easily done with the `nn.Linear` module. It can also work as a regular affine layer, like the one you have implemented from previous assignments. Run the following examples to see how it works. You will intensively use `nn.Linear` later.

```
fix_random_seed(0)

N, T, D, M = 2, 3, 4, 3

w = torch.linspace(-0.2, 0.4, steps=D*M, **to_double_cuda).reshape(D, M).permute(1, 0)
b = torch.linspace(-0.4, 0.1, steps=M, **to_double_cuda)

temporal_affine = nn.Linear(D, M).to(**to_double_cuda)
temporal_affine.weight.data.copy_(w)
temporal_affine.bias.data.copy_(b)

# For regular affine layer
x = torch.linspace(-0.1, 0.3, steps=N*D, **to_double_cuda).reshape(N, D)
out = temporal_affine(x)
print('affine layer - input shape: {}, output shape: {}'.format(x.shape, out.shape))
correct_out = torch.tensor([[-0.35584416, -0.10896104,  0.13792208],
                    [-0.31428571, -0.01753247,  0.27922078]], **to_double_cuda)

print('dx error: ', rel_error(out, correct_out))


# For temporal affine layer
x = torch.linspace(-0.1, 0.3, steps=N*T*D, **to_double_cuda).reshape(N, T, D)
out = temporal_affine(x)
print('\ntemporal affine layer - input shape: {}, output shape: {}'.format(x.shape, out.sh
correct_out = torch.tensor([[[-0.39920949, -0.16533597,  0.06853755],
                             [-0.38656126, -0.13750988,  0.11154150],
                             [-0.37391304, -0.10968379,  0.15454545]],
```

```
                                [[-0.36126482, -0.08185771,  0.19754941],
                                 [-0.34861660, -0.05403162,  0.24055336],
                                 [-0.33596838, -0.02620553,  0.28355731]]], **to_double_cuda)

print('dx error: ', rel_error(out, correct_out))
```

Temporal Softmax loss

In an RNN language model, at every timestep we produce a score for each word in the vocabulary. We know the ground-truth word at each timestep, so we use a softmax loss function to compute loss and gradient at each timestep. We sum the losses over time and average them over the minibatch.

However there is one wrinkle: since we operate over minibatches and different captions may have different lengths, we append `<NULL>` tokens to the end of each caption so they all have the same length. We don't want these `<NULL>` tokens to count toward the loss or gradient, so in addition to scores and ground-truth labels our loss function also accepts a `ignore_index` that tells it which index in caption should be ignored when computing the loss.

```
def temporal_softmax_loss(x, y, ignore_index=NULL_index):
    """
    A temporal version of softmax loss for use in RNNs. We assume that we are
    making predictions over a vocabulary of size V for each timestep of a
    timeseries of length T, over a minibatch of size N. The input x gives scores
    for all vocabulary elements at all timesteps, and y gives the indices of the
    ground-truth element at each timestep. We use a cross-entropy loss at each
    timestep, *summing* the loss over all timesteps and *averaging* across the
    minibatch.

    As an additional complication, we may want to ignore the model output at some
    timesteps, since sequences of different length may have been combined into a
    minibatch and padded with NULL tokens. The optional ignore_index argument
    tells us which elements in the caption should not contribute to the loss.

    Inputs:
    - x: Input scores, of shape (N, T, V)
    - y: Ground-truth indices, of shape (N, T) where each element is in the range
         0 <= y[i, t] < V

    Returns a tuple of:
    - loss: Scalar giving loss
    """
    loss = None
```

```
    ##############################################################################
    # TODO: Implement the temporal softmax loss function.                        #
    #                                                                            #
    # REQUIREMENT: This part MUST be done in one single line of code!            #
    #                                                                            #
    # HINT: Look up the function torch.functional.cross_entropy, set             #
    # ignore_index to the variable ignore_index (i.e., index of NULL) and        #
    # set reduction to either 'sum' or 'mean' (avoid using 'none' for now).      #
    #                                                                            #
    # We use a cross-entropy loss at each timestep, *summing* the loss over      #
    # all timesteps and *averaging* across the minibatch.                        #
    ##############################################################################
    # Replace "pass" statement with your code
    N, T, V = x.shape
    x_flat = x.reshape(N*T, V)
    y_flat = y.reshape(N*T)
    loss = F.cross_entropy(x_flat, y_flat, ignore_index=ignore_index, reduction='sum')
    loss = loss / N
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################

    return loss
```

Sanity check

```
def check_loss(N, T, V, p):
    x = 0.001 * torch.randn(N, T, V, **to_double_cuda)
    y = torch.randint(V, size=(N, T), **to_long_cuda)
    mask = torch.rand(N, T, **to_double_cuda)
    y[mask > p] = 0
    print(temporal_softmax_loss(x, y).item())

check_loss(1000, 1, 10, 1.0)   # Should be about 2.00-2.11
check_loss(1000, 10, 10, 1.0)  # Should be about 20.6-21.0
check_loss(5000, 10, 10, 0.1) # Should be about 2.00-2.11
```

Captioning Module

Now we are wrapping everything into the captioning module. Implement the `__init__` function for initialization and the `captioning_forward` for the forward pass. For now you only need to implement for the case where `cell_type='rnn'`, indicating vanialla RNNs; you will implement the `LSTM` case and `AttentionLSTM` case later.

```
class CaptioningRNN(nn.Module):
```

```
    """
    A CaptioningRNN produces captions from images using a recurrent
    neural network.

    The RNN receives input vectors of size D, has a vocab size of V, works on
    sequences of length T, has an RNN hidden dimension of H, uses word vectors
    of dimension W, and operates on minibatches of size N.

    Note that we don't use any regularization for the CaptioningRNN.

    You will implement the `__init__` method for model initialization and
    the `forward` method first, then come back for the `sample` method later.
    """
    def __init__(self, word_to_idx, input_dim=512, wordvec_dim=128,
                 hidden_dim=128, cell_type='rnn', device='cpu', dtype=torch.float32):
        """
        Construct a new CaptioningRNN instance.

        Inputs:
        - word_to_idx: A dictionary giving the vocabulary. It contains V entries,
          and maps each string to a unique integer in the range [0, V).
        - input_dim: Dimension D of input image feature vectors.
        - wordvec_dim: Dimension W of word vectors.
        - hidden_dim: Dimension H for the hidden state of the RNN.
        - cell_type: What type of RNN to use; either 'rnn' or 'lstm'.
        - dtype: datatype to use; use float32 for training and float64 for
          numeric gradient checking.
        """
        super().__init__()
        if cell_type not in {'rnn', 'lstm', 'attention'}:
            raise ValueError('Invalid cell_type "%s"' % cell_type)

        self.cell_type = cell_type
        self.word_to_idx = word_to_idx
        self.idx_to_word = {i: w for w, i in word_to_idx.items()}

        vocab_size = len(word_to_idx)

        self._null = word_to_idx['<NULL>']
        self._start = word_to_idx.get('<START>', None)
        self._end = word_to_idx.get('<END>', None)

        ############################################################################
        # TODO: Initialize the image captioning module. Refer to the TODO        #
        # in the captioning_forward function on layers you need to create        #
        #                                                                          #
        # Hint: You can use nn.Linear for both                                     #
        # i) output projection (from RNN hidden state to vocab probability) and  #
        # ii) feature projection (from CNN pooled feature to h0)                 #
        #                                                                          #
        # Hint: In FeatureExtractor, set pooling=True to get the pooled CNN      #
        #       feature and pooling=False to get the CNN activation map.         #
        ############################################################################
        # Replace "pass" statement with your code
        self.wordvec_dim = wordvec_dim
```

```
        self.feat_extract = None
        self.affine = None
        self.rnn = None
        self.affine = nn.Linear(input_dim, hidden_dim).to(device=device, dtype=dtype)
        if cell_type == 'rnn' or cell_type == 'lstm':
          self.feat_extract = FeatureExtractor(pooling=True, device=device, dtype=dtype)
          # self.affine = nn.Linear(input_dim, hidden_dim).to(device=device, dtype=dtype)
          if cell_type == 'rnn':
            self.rnn = RNN(wordvec_dim, hidden_dim, device=device, dtype=dtype)
          else:
            self.rnn = LSTM(wordvec_dim, hidden_dim, device=device, dtype=dtype)
        elif cell_type == 'attention':
          self.feat_extract = FeatureExtractor(pooling=False, device=device, dtype=dtype)
          self.rnn = AttentionLSTM(wordvec_dim, hidden_dim, device=device, dtype=dtype)
        else:
          raise ValueError
        nn.init.kaiming_normal_(self.affine.weight)
        nn.init.zeros_(self.affine.bias)
        self.word_embed = WordEmbedding(vocab_size, wordvec_dim, device=device, dtype=dtyp
        self.temporal_affine = nn.Linear(hidden_dim, vocab_size).to(device=device, dtype=d
        nn.init.kaiming_normal_(self.temporal_affine.weight)
        nn.init.zeros_(self.temporal_affine.bias)
        ############################################################################
        #                               END OF YOUR CODE                           #
        ############################################################################

    def forward(self):
      raise NotImplementedError

    def sample(self):
      raise NotImplementedError
```

Forward part

Implement the forward function.

```
    def captioning_forward(self, images, captions):
        """
        Compute training-time loss for the RNN. We input images and
        ground-truth captions for those images, and use an RNN (or LSTM) to compute
        loss. The backward part will be done by torch.autograd.

        Inputs:
        - images: Input images, of shape (N, 3, 112, 112)
        - captions: Ground-truth captions; an integer array of shape (N, T + 1) where
          each element is in the range 0 <= y[i, t] < V

        Outputs:
        - loss: A scalar loss
        """
        # Cut captions into two pieces: captions_in has everything but the last word
        # and will be input to the RNN; captions_out has everything but the first
        # word and this is what we will expect the RNN to generate. These are offset
        # by one relative to each other because the RNN should produce word (t+1)
```

```
        # by one relative to each other because the RNN should produce word (t+1)
        # after receiving word t. The first element of captions_in will be the START
        # token, and the first element of captions_out will be the first word.
        captions_in = captions[:, :-1]
        captions_out = captions[:, 1:]

        loss = 0.0
        ############################################################################
        # TODO: Implement the forward pass for the CaptioningRNN.                  #
        # In the forward pass you will need to do the following:                   #
        # (1) Use an affine transformation to project the image feature to         #
        #     the initial hidden state $h0$ (for RNN/LSTM, of shape (N, H)) or     #
        #     the projected CNN activation input $A$ (for Attention LSTM,          #
        #     of shape (N, H, 4, 4).                                               #
        # (2) Use a word embedding layer to transform the words in captions_in     #
        #     from indices to vectors, giving an array of shape (N, T, W).         #
        # (3) Use either a vanilla RNN or LSTM (depending on self.cell_type) to    #
        #     process the sequence of input word vectors and produce hidden state  #
        #     vectors for all timesteps, producing an array of shape (N, T, H).    #
        # (4) Use a (temporal) affine transformation to compute scores over the    #
        #     vocabulary at every timestep using the hidden states, giving an      #
        #     array of shape (N, T, V).                                            #
        # (5) Use (temporal) softmax to compute loss using captions_out, ignoring  #
        #     the points where the output word is <NULL>.                          #
        #                                                                          #
        # Do not worry about regularizing the weights or their gradients!          #
        ############################################################################
        # Replace "pass" statement with your code
        feature = self.feat_extract.extract_mobilenet_feature(images)
        # print(feature.shape)
        if self.cell_type == 'attention':
          feature = feature.permute(0, 2, 3, 1) # make it N * 4 * 4 * input_dim
          # print(feature.shape)
        #else:
        h0 = self.affine(feature)
        # print(h0.shape)
        if self.cell_type == 'attention':
          h0 = h0.permute(0, 3, 1, 2) # permute back
          # print(h0.shape)
        x = self.word_embed(captions_in)
        h = self.rnn(x, h0)
        score = self.temporal_affine(h)
        loss = temporal_softmax_loss(score, captions_out, ignore_index=self._null)
        ############################################################################
        #                              END OF YOUR CODE                            #
        ############################################################################

        return loss

    CaptioningRNN.forward = captioning_forward
```

Inference part

We will come back to this part later.

```
    def sample_caption(self, images, max_length=15):
        """
        Run a test-time forward pass for the model, sampling captions for input
        feature vectors.

        At each timestep, we embed the current word, pass it and the previous hidden
        state to the RNN to get the next hidden state, use the hidden state to get
        scores for all vocab words, and choose the word with the highest score as
        the next word. The initial hidden state is computed by applying an affine
        transform to the image features, and the initial word is the <START>
        token.

        For LSTMs you will also have to keep track of the cell state; in that case
        the initial cell state should be zero.

        Inputs:
        - images: Input images, of shape (N, 3, 112, 112)
        - max_length: Maximum length T of generated captions

        Returns:
        - captions: Array of shape (N, max_length) giving sampled captions,
          where each element is an integer in the range [0, V). The first element
          of captions should be the first sampled word, not the <START> token.
        """
        N = images.shape[0]
        captions = self._null * images.new(N, max_length).fill_(1).long()

        if self.cell_type == 'attention':
          attn_weights_all = images.new(N, max_length, 4, 4).fill_(0).float()

        ##############################################################################
        # TODO: Implement test-time sampling for the model. You will need to      #
        # initialize the hidden state of the RNN by applying the learned affine   #
        # transform to the image features. The first word that you feed to  #
        # the RNN should be the <START> token; its value is stored in the         #
        # variable self._start. At each timestep you will need to do to:          #
        # (1) Embed the previous word using the learned word embeddings           #
        # (2) Make an RNN step using the previous hidden state and the embedded   #
        #     current word to get the next hidden state.                          #
        # (3) Apply the learned affine transformation to the next hidden state to #
        #     get scores for all words in the vocabulary                          #
        # (4) Select the word with the highest score as the next word, writing it #
        #     (the word index) to the appropriate slot in the captions variable   #
        #                                                                         #
        # For simplicity, you do not need to stop generating after an <END> token #
        # is sampled, but you can if you want to.                                 #
        #                                                                         #
        # HINT: You will not be able to use the rnn_forward or lstm_forward       #
        # functions; you'll need to call the `step_forward` from the              #
        # RNN/LSTM/AttentionLSTM module in a loop.                                #
        #                                                                         #
        # NOTE: we are still working over minibatches in this function. Also if   #
        # you are using an LSTM, initialize the first cell state to zeros.        #
        # For AttentionLSTM, first project the 1280x4x4 CNN feature activation to #
```

```
        # For AttentionLSTM, first project the 1280x1x1 CNN feature activation to #
        # $A$ of shape Hx4x4. The LSTM initial hidden state and cell state        #
        # would both be A.mean(dim=(2, 3)).                                       #
        ###########################################################################
        # Replace "pass" statement with your code
        feature = self.feat_extract.extract_mobilenet_feature(images)
        # prev_h = self.affine(feature)
        A = None
        if self.cell_type == 'attention':
          feature = feature.permute(0, 2, 3, 1) # make it N * 4 * 4 * input_dim
        prev_h = self.affine(feature)
        prev_c = torch.zeros_like(prev_h)
        if self.cell_type == 'attention':
          A = prev_h.permute(0, 3, 1, 2) # permute back
          prev_h = A.mean(dim=(2, 3))
          prev_c = A.mean(dim=(2, 3))

        x = torch.ones((N, self.wordvec_dim), dtype=prev_h.dtype, device=prev_h.device) *
        for i in range(max_length):
          # print(x.shape)
          # print(prev_h.shape)
          next_h = None
          if self.cell_type == 'rnn':
            next_h = self.rnn.step_forward(x, prev_h)
          elif self.cell_type == 'lstm':
            next_h, prev_c = self.rnn.step_forward(x, prev_h, prev_c)
          else:
            attn, attn_weights = dot_product_attention(prev_h, A)
            attn_weights_all[:, i] = attn_weights
            next_h, prev_c = self.rnn.step_forward(x, prev_h, prev_c, attn)

          score = self.temporal_affine(next_h)
          # loss = temporal_softmax_loss(score, captions_out, ignore_index=self._null)
          max_idx = torch.argmax(score, dim=1)
          captions[:, i] = max_idx
          x = self.word_embed(max_idx)
          # print(x.shape)
          prev_h = next_h
        ############################################################################
        #                             END OF YOUR CODE                             #
        ############################################################################
        if self.cell_type == 'attention':
          return captions, attn_weights_all.cpu()
        else:
          return captions

    CaptioningRNN.sample = sample_caption
```

Sanity check

Run the following to check your forward pass using a small test case; you should see difference on the order of 1e-7 or less.

```
fix_random_seed(0)
```

```
fix_random_seed(0)

N, D, W, H = 10, 1280, 30, 40
D_img = 112
word_to_idx = {'<NULL>': 0, 'cat': 2, 'dog': 3}
V = len(word_to_idx)
T = 13

model = CaptioningRNN(word_to_idx,
          input_dim=D,
          wordvec_dim=W,
          hidden_dim=H,
          cell_type='rnn',
          **to_float_cuda) # use float here to be consistent with MobileNet v2


for k,v in model.named_parameters():
  # print(k, v.shape) # uncomment this to see the weight shape
  v.data.copy_(torch.linspace(-1.4, 1.3, steps=v.numel()).reshape(*v.shape))

images = torch.linspace(-3., 3., steps=(N * 3 * D_img * D_img),
                        **to_float_cuda).reshape(N, 3, D_img, D_img)
captions = (torch.arange(N * T, **to_long_cuda) % V).reshape(N, T)

loss = model(images, captions).item()
expected_loss = 150.6090393066

print('loss: ', loss)
print('expected loss: ', expected_loss)
print('difference: ', rel_error(torch.tensor(loss), torch.tensor(expected_loss)))
```

Image Captioning solver

Different from the `Solver` class that we used to train image classification models on the previous assignment, on this assignment we use the `torch.optim` package to train image captioning models.

We have written this part for you and you need to train the model and generate plots on the training loss.

```
def CaptioningTrain(rnn_model, image_data, caption_data, lr_decay=1, **kwargs):
  """
  Run optimization to train the model.
  """
  # optimizer setup
  from torch import optim
  optimizer = optim.Adam(
    filter(lambda p: p.requires_grad, rnn_model.parameters()),
    learning_rate) # leave betas and eps by default
```

```
  lr_scheduler = optim.lr_scheduler.LambdaLR(optimizer,
                                             lambda epoch: lr_decay ** epoch)

  # sample minibatch data
  iter_per_epoch = math.ceil(image_data.shape[0] // batch_size)
  loss_history = []
  rnn_model.train()
  for i in range(num_epochs):
    start_t = time.time()
    for j in range(iter_per_epoch):
      images, captions = image_data[j*batch_size:(j+1)*batch_size], \
                         caption_data[j*batch_size:(j+1)*batch_size]

      loss = rnn_model(images, captions)
      optimizer.zero_grad()
      loss.backward()
      loss_history.append(loss.item())
      optimizer.step()
    end_t = time.time()
    print('(Epoch {} / {}) loss: {:.4f} time per epoch: {:.1f}s'.format(
        i, num_epochs, loss.item(), end_t-start_t))

    lr_scheduler.step()

  # plot the training losses
  plt.plot(loss_history)
  plt.xlabel('Iteration')
  plt.ylabel('Loss')
  plt.title('Training loss history')
  plt.show()
```

Overfit small data

Once you have familiarized yourself with the `optim` API used above, run the following to make sure your model overfits a small sample of 50 training examples. You should see a final loss of less than 1.6.

```
fix_random_seed(0)

# data input
small_num_train = 50
sample_idx = torch.linspace(0, num_train-1, steps=small_num_train, **to_float_cuda).long()
small_image_data = data_dict['train_images'][sample_idx].to('cuda')
small_caption_data = data_dict['train_captions'][sample_idx].to('cuda')

# optimization arguments
num_epochs = 80
batch_size = 50

# create the image captioning model
model = CaptioningRNN(
          cell_type='rnn',
```

```
            word_to_idx=data_dict['vocab']['token_to_idx'],
            input_dim=1280, # hard-coded, do not modify
            hidden_dim=512,
            wordvec_dim=256,
            **to_float_cuda)

    for learning_rate in [1e-3]:
      print('learning rate is: ', learning_rate)
      CaptioningTrain(model, small_image_data, small_caption_data,
                  num_epochs=num_epochs, batch_size=batch_size,
                  learning_rate=learning_rate)
```

Caption sampling

Unlike classification models, image captioning models behave very differently at training time and at test time. At training time, we have access to the ground-truth caption, so we feed ground-truth words as input to the RNN at each timestep. At test time, we sample from the distribution over the vocabulary at each timestep, and feed the sample as input to the RNN at the next timestep.

Implement the `sample` method in captioning module `CaptioningRNN` for test-time sampling. After doing so, run the following to train a captioning model and sample from the model on both training and validation data.

Train the net

After you are done implementing the `sample` method, perform the training on the entire training set. You should see a final loss less than `2.0`.

```
fix_random_seed(0)

# data input
small_num_train = num_train
sample_idx = torch.randint(num_train, size=(small_num_train,), **to_long_cuda)
small_image_data = data_dict['train_images'][sample_idx].to('cuda')
small_caption_data = data_dict['train_captions'][sample_idx].to('cuda')

# optimization arguments
num_epochs = 60
batch_size = 250

# create the image captioning model
rnn_model = CaptioningRNN(
          cell_type='rnn',
          word_to_idx=data_dict['vocab']['token_to_idx'],
          input_dim=1280, # hard-coded, do not modify
          hidden_dim=512,
          wordvec_dim=256,
          **to_float_cuda)

for learning_rate in [1e-3]:
  print('learning rate is: ', learning_rate)
```

```
CaptioningTrain(rnn_model, small_image_data, small_caption_data,
              num_epochs=num_epochs, batch_size=batch_size,
              learning_rate=learning_rate)
```

Test-time sampling

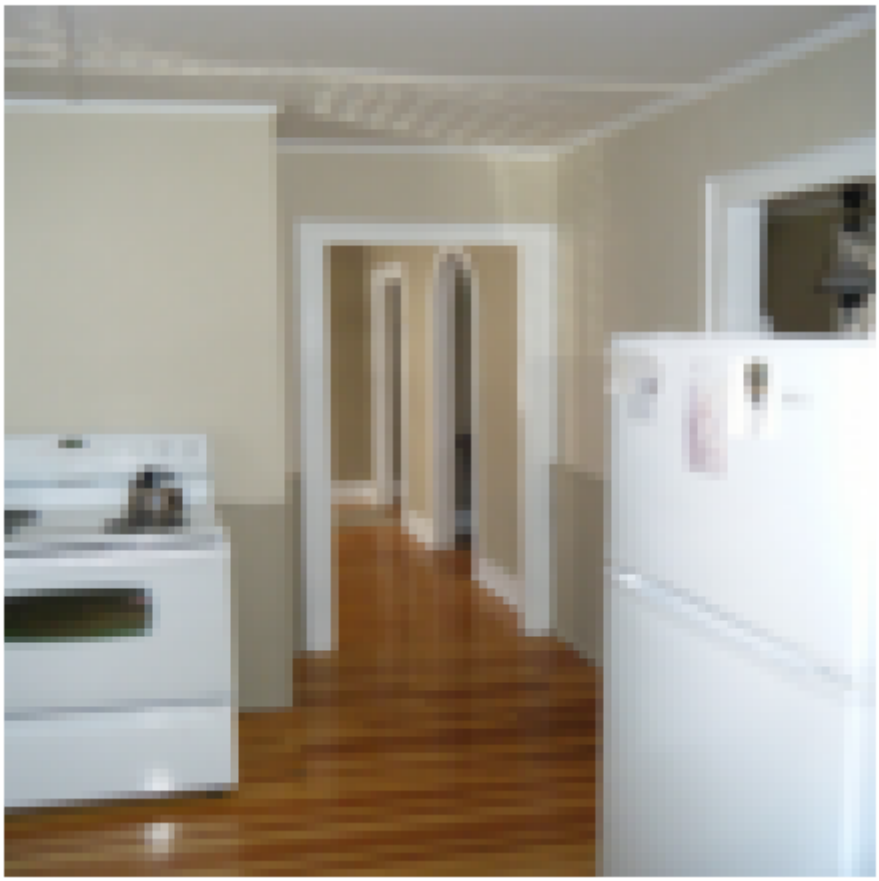
The samples on training data should be very good; the samples on validation data will probably make less sense.

```
# Sample a minibatch and show the reshaped 112x112 images,
# GT captions, and generated captions by your model.
batch_size = 3

for split in ['train', 'val']:
  sample_idx = torch.randint(0, num_train if split=='train' else num_val, (batch_size,))
  sample_images = data_dict[split+'_images'][sample_idx]
  sample_captions = data_dict[split+'_captions'][sample_idx]

  gt_captions = decode_captions(sample_captions, data_dict['vocab']['idx_to_token'])
  rnn_model.eval()
  generated_captions = rnn_model.sample(sample_images)
  generated_captions = decode_captions(generated_captions, data_dict['vocab']['idx_to_toke

  for i in range(batch_size):
    plt.imshow(sample_images[i].permute(1, 2, 0))
    plt.axis('off')
    plt.title('%s\nRNN Generated:%s\nGT:%s' % (split, generated_captions[i], gt_captions[i
    plt.show()
```

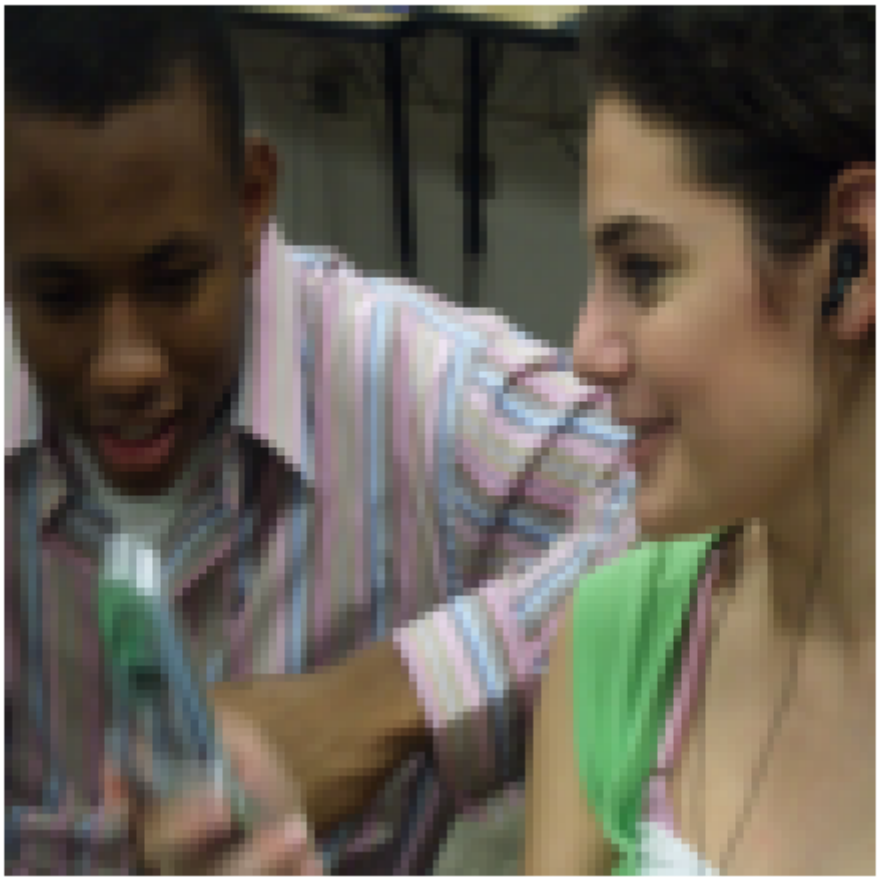
train
RNN Generated:a man riding skis in a snow covered slope on a ski slope <END>
GT:<START> a group of skiers are standing in the snow <END>

train
RNN Generated:a group of people walking along a dirty snow covered road <END>
GT:<START> a group of people walking along a dirty snow covered road <END>

train
RNN Generated:a <UNK> standing with a <UNK> on a wooden floor <END>
GT:<START> a view of a kitchen and a <UNK> in a home <END>



val
RNN Generated:two young boys play with a toy room <END>
GT:<START> a boy and a girl sitting next to each other looking at an <UNK> <UNK> <END>



val
RNN Generated:a bathroom with a toilet sink and a toilet tub <END>
GT:<START> a close up of a white toilet and <UNK> can <END>

Image Captioning with LSTMs

In the previous exercise you implemented a vanilla RNN and applied it to image captioning. Next, we will implement the LSTM update rule and use it for image captioning.

LSTM

If you read recent papers, you'll see that many people use a variant on the vanilla RNN called Long-Short Term Memory (LSTM) RNNs. Vanilla RNNs can be tough to train on long sequences due to vanishing and exploding gradients caused by repeated matrix multiplication. LSTMs solve this problem by replacing the simple update rule of the vanilla RNN with a gating mechanism as follows.

Similar to the vanilla RNN, at each timestep we receive an input $x_t \in \mathbb{R}^D$ and the previous hidden state $h_{t-1} \in \mathbb{R}^H$; the LSTM also maintains an H-dimensional *cell state*, so we also receive the previous cell state $c_{t-1} \in \mathbb{R}^H$. The learnable parameters of the LSTM are an *input-to-hidden* matrix $W_x \in \mathbb{R}^{4H \times D}$, a *hidden-to-hidden* matrix $W_h \in \mathbb{R}^{4H \times H}$ and a *bias vector* $b \in \mathbb{R}^{4H}$.

At each timestep we first compute an *activation vector* $a \in \mathbb{R}^{4H}$ as $a = W_x x_t + W_h h_{t-1} + b$. We then divide this into four vectors $a_i, a_f, a_o, a_g \in \mathbb{R}^H$ where a_i consists of the first H elements of a, a_f is the next H elements of a, etc. We then compute the *input gate* $g \in \mathbb{R}^H$, *forget gate* $f \in \mathbb{R}^H$, *output gate* $o \in \mathbb{R}^H$ and *block input* $g \in \mathbb{R}^H$ as

$$i = \sigma(a_i) \qquad f = \sigma(a_f) \qquad o = \sigma(a_o) \qquad g = \tanh(a_g)$$

where σ is the sigmoid function and $\tanh$ is the hyperbolic tangent, both applied elementwise.

Finally we compute the next cell state c_t and next hidden state h_t as

$$c_t = f \odot c_{t-1} + i \odot g \qquad\qquad h_t = o \odot \tanh(c_t)$$

where $\odot$ is the elementwise product of vectors.

In the rest of the notebook we will implement the LSTM update rule and apply it to the image captioning task.

In the code, we assume that data is stored in batches so that $X_t \in \mathbb{R}^{N \times D}$, and will work with *transposed* versions of the parameters: $W_x \in \mathbb{R}^{D \times 4H}$, $W_h \in \mathbb{R}^{H \times 4H}$ so that activations $A \in \mathbb{R}^{N \times 4H}$ can be computed efficiently as $A = X_t W_x + H_{t-1} W_h$

LSTM: step forward

Implement the forward pass for a single timestep of an LSTM in the `lstm_step_forward` function. This should be similar to the `rnn_step_forward` function that you implemented above, but using the LSTM update rule instead.

Don't worry about the backward part! `autograd` will handle it.

```
def lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b, attn=None, Wattn=None):
    """
    Forward pass for a single timestep of an LSTM.

    The input data has dimension D, the hidden state has dimension H, and we use
    a minibatch size of N.

    Inputs:
    - x: Input data, of shape (N, D)
    - prev_h: Previous hidden state, of shape (N, H)
    - prev_c: previous cell state, of shape (N, H)
    - Wx: Input-to-hidden weights, of shape (D, 4H)
    - Wh: Hidden-to-hidden weights, of shape (H, 4H)
    - b: Biases, of shape (4H,)
    - attn and Wattn are for Attention LSTM only, indicate the attention input and
      embedding weights for the attention input

    Returns a tuple of:
    - next_h: Next hidden state, of shape (N, H)
    - next_c: Next cell state, of shape (N, H)
    """
    next_h, next_c = None, None
    ##############################################################################
    # TODO: Implement the forward pass for a single timestep of an LSTM.         #
    # You may want to use torch.sigmoid() for the sigmoid function.              #
    ##############################################################################
    # Replace "pass" statement with your code
    N, H = prev_h.shape
    a = None
    if attn is None: # regular lstm
      a = x.mm(Wx) + prev_h.mm(Wh) + b
```

```
    else:
      a = x.mm(Wx) + prev_h.mm(Wh) + b + attn.mm(Wattn)
    i = torch.sigmoid(a[:,0:H])
    f = torch.sigmoid(a[:,H:2*H])
    o = torch.sigmoid(a[:,2*H:3*H])
    g = torch.tanh(a[:,3*H:4*H])
    next_c = f * prev_c + i * g
    next_h = o * torch.tanh(next_c)
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################

    return next_h, next_c
```

Once you are done, run the following to perform a simple test of your implementation. You should see errors on the order of `1e-7` or less.

```
N, D, H = 3, 4, 5
x = torch.linspace(-0.4, 1.2, steps=N*D, **to_double_cuda).reshape(N, D)
prev_h = torch.linspace(-0.3, 0.7, steps=N*H, **to_double_cuda).reshape(N, H)
prev_c = torch.linspace(-0.4, 0.9, steps=N*H, **to_double_cuda).reshape(N, H)
Wx = torch.linspace(-2.1, 1.3, steps=4*D*H, **to_double_cuda).reshape(D, 4 * H)
Wh = torch.linspace(-0.7, 2.2, steps=4*H*H, **to_double_cuda).reshape(H, 4 * H)
b = torch.linspace(0.3, 0.7, steps=4*H, **to_double_cuda)

next_h, next_c = lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)

expected_next_h = torch.tensor([
    [ 0.24635157,  0.28610883,  0.32240467,  0.35525807,  0.38474904],
    [ 0.49223563,  0.55611431,  0.61507696,  0.66844003,  0.7159181 ],
    [ 0.56735664,  0.66310127,  0.74419266,  0.80889665,  0.858299  ]], **to_double_cuda)
expected_next_c = torch.tensor([
    [ 0.32986176,  0.39145139,  0.451556,    0.51014116,  0.56717407],
    [ 0.66382255,  0.76674007,  0.87195994,  0.97902709,  1.08751345],
    [ 0.74192008,  0.90592151,  1.07717006,  1.25120233,  1.42395676]], **to_double_cuda)

print('next_h error: ', rel_error(expected_next_h, next_h))
print('next_c error: ', rel_error(expected_next_c, next_c))
```

LSTM: forward

Implement the `lstm_forward` function to run an LSTM forward on an entire timeseries of data.

Again, don't worry about the backward part! `autograd` will handle it.

```
def lstm_forward(x, h0, Wx, Wh, b):
```

```
    """
    Forward pass for an LSTM over an entire sequence of data. We assume an input
    sequence composed of T vectors, each of dimension D. The LSTM uses a hidden
    size of H, and we work over a minibatch containing N sequences. After running
    the LSTM forward, we return the hidden states for all timesteps.

    Note that the initial cell state is passed as input, but the initial cell
    state is set to zero. Also note that the cell state is not returned; it is
    an internal variable to the LSTM and is not accessed from outside.

    Inputs:
    - x: Input data, of shape (N, T, D)
    - h0: Initial hidden state, of shape (N, H)
    - Wx: Weights for input-to-hidden connections, of shape (D, 4H)
    - Wh: Weights for hidden-to-hidden connections, of shape (H, 4H)
    - b: Biases, of shape (4H,)

    Returns a tuple of:
    - h: Hidden states for all timesteps of all sequences, of shape (N, T, H)
    """
    h = None
    c0 = torch.zeros_like(h0) # we provide the intial cell state c0 here for you!
    ##############################################################################
    # TODO: Implement the forward pass for an LSTM over an entire timeseries.   #
    # You should use the lstm_step_forward function that you just defined.      #
    ##############################################################################
    # Replace "pass" statement with your code
    N, T, D = x.shape
    _, H = h0.shape
    h = torch.zeros((N, T, H), dtype=h0.dtype, device=h0.device)
    prev_h = h0
    prev_c = c0
    for i in range(T):
      next_h, next_c = lstm_step_forward(x[:,i,:], prev_h, prev_c, Wx, Wh, b)
      prev_h = next_h
      prev_c = next_c
      h[:, i, :] = prev_h
    ######################
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################

    return h
```

When you are done, run the following to check your implementation. You should see an error on the order of `1e-7` or less.

```
N, D, H, T = 2, 5, 4, 3
x = torch.linspace(-0.4, 0.6, steps=N*T*D, **to_double_cuda).reshape(N, T, D)
h0 = torch.linspace(-0.4, 0.8, steps=N*H, **to_double_cuda).reshape(N, H)
Wx = torch.linspace(-0.2, 0.9, steps=4*D*H, **to_double_cuda).reshape(D, 4 * H)
Wh = torch.linspace(-0.3, 0.6, steps=4*H*H, **to_double_cuda).reshape(H, 4 * H)
b = torch.linspace(0.2, 0.7, steps=4*H, **to_double_cuda)
```

```
h = lstm_forward(x, h0, Wx, Wh, b)

expected_h = torch.tensor([
 [[ 0.01764008,  0.01823233,  0.01882671,  0.0194232 ],
  [ 0.11287491,  0.12146228,  0.13018446,  0.13902939],
  [ 0.31358768,  0.33338627,  0.35304453,  0.37250975]],
 [[ 0.45767879,  0.4761092,   0.4936887,   0.51041945],
  [ 0.6704845,   0.69350089,  0.71486014,  0.7346449 ],
  [ 0.81733511,  0.83677871,  0.85403753,  0.86935314]]], **to_double_cuda)

print('h error: ', rel_error(expected_h, h))
```

LSTM Module

We can now wrap the LSTM functions we wrote into an nn.Module.

```
class LSTM(nn.Module):
  """
  This is our single-layer, uni-directional LSTM module.

  Arguments for initialization:
  - input_size: Input size, denoted as D before
  - hidden_size: Hidden size, denoted as H before
  """
  def __init__(self, input_size, hidden_size, device='cpu',
               dtype=torch.float32):
    """
    Initialize a LSTM.
    Model parameters to initialize:
    - Wx: Weights for input-to-hidden connections, of shape (D, 4H)
    - Wh: Weights for hidden-to-hidden connections, of shape (H, 4H)
    - b: Biases, of shape (4H,)
    """
    super().__init__()

    # Register parameters
    self.Wx = Parameter(torch.randn(input_size, hidden_size*4,
                        device=device, dtype=dtype).div(math.sqrt(input_size)))
    self.Wh = Parameter(torch.randn(hidden_size, hidden_size*4,
                        device=device, dtype=dtype).div(math.sqrt(hidden_size)))
    self.b = Parameter(torch.zeros(hidden_size*4,
                        device=device, dtype=dtype))

  def forward(self, x, h0):
    """
    Inputs:
    - x: Input data for the entire timeseries, of shape (N, T, D)
    - h0: Initial hidden state, of shape (N, H)
```

```
    Outputs:
    - hn: The hidden state output
    """
    hn = lstm_forward(x, h0, self.Wx, self.Wh, self.b)
    return hn

  def step_forward(self, x, prev_h, prev_c):
    """
    Inputs:
    - x: Input data for one time step, of shape (N, D)
    - prev_h: The previous hidden state, of shape (N, H)
    - prev_c: The previous cell state, of shape (N, H)

    Outputs:
    - next_h: The next hidden state, of shape (N, H)
    - next_c: The next cell state, of shape (N, H)
    """
    next_h, next_c = lstm_step_forward(x, prev_h, prev_c, self.Wx, self.Wh, self.b)
    return next_h, next_c
```

LSTM captioning model

Now that you have implemented an LSTM, update the implementation of the `init` method in class `CaptioningRNN` **ONLY** to also handle the case where `self.cell_type` is `lstm`. **This should require adding less than 5 lines of code.**

Once you have done so, run the following to check your implementation. You should see a difference on the order of `1e-7` or less.

```
fix_random_seed(0)

N, D, W, H = 10, 1280, 30, 40
D_img = 112
word_to_idx = {'<NULL>': 0, 'cat': 2, 'dog': 3}
V = len(word_to_idx)
T = 13

model = CaptioningRNN(word_to_idx,
          input_dim=D,
          wordvec_dim=W,
          hidden_dim=H,
          cell_type='lstm',
          **to_float_cuda)

for k,v in model.named_parameters():
  # print(k, v.shape) # uncomment this to see the weight shape
  v.data.copy_(torch.linspace(-1.4, 1.3, steps=v.numel()).reshape(*v.shape))

images = torch.linspace(-3., 3., steps=(N * 3 * D_img * D_img),
                        **to_float_cuda).reshape(N, 3, D_img, D_img)
captions = (torch.arange(N * T, **to_long_cuda) % V).reshape(N, T)
```

```
loss = model(images, captions).item()
expected_loss = 146.3161468505

print('loss: ', loss)
print('expected loss: ', expected_loss)
print('difference: ', rel_error(torch.tensor(loss), torch.tensor(expected_loss)))
```

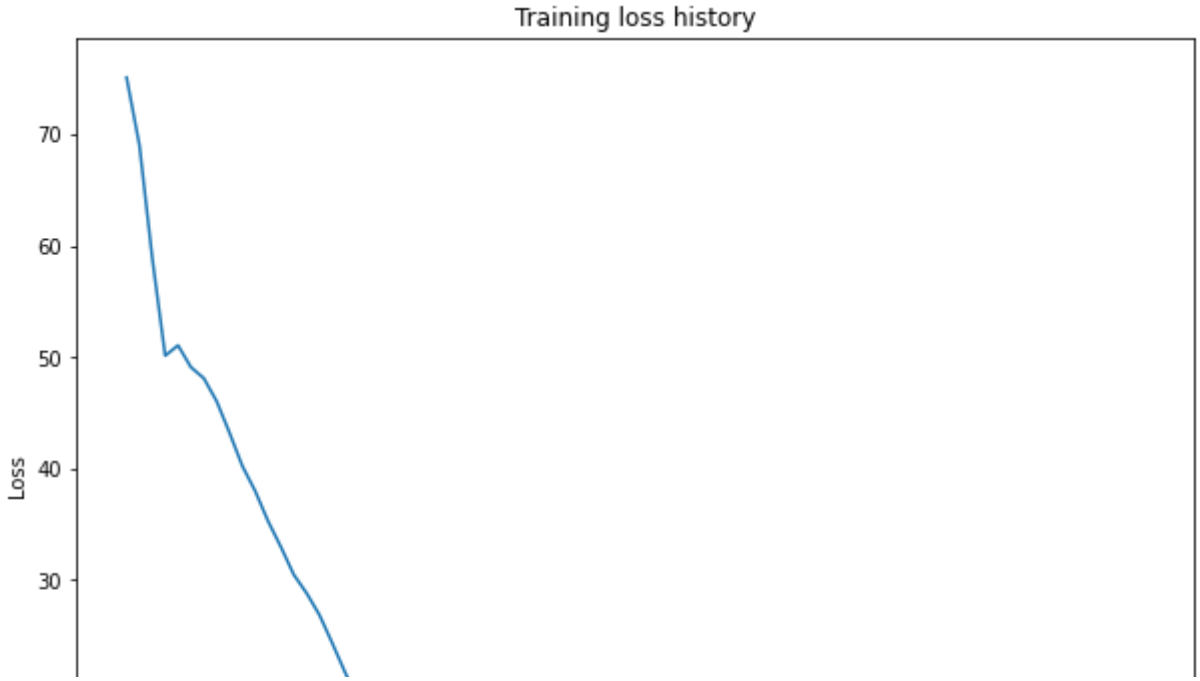
Overfit small data

We have written this part for you. Run the following to overfit an LSTM captioning model on the same small dataset as we used for the RNN previously. You should see a final loss less than `4` after 80 epochs.

```
fix_random_seed(0)

# data input
small_num_train = 50
sample_idx = torch.linspace(0, num_train-1, steps=small_num_train, **to_float_cuda).long()
small_image_data = data_dict['train_images'][sample_idx].to('cuda')
small_caption_data = data_dict['train_captions'][sample_idx].to('cuda')

# optimization arguments
num_epochs = 80
batch_size = 50

# create the image captioning model
model = CaptioningRNN(
          cell_type='lstm',
          word_to_idx=data_dict['vocab']['token_to_idx'],
          input_dim=1280, # hard-coded, do not modify
          hidden_dim=512,
          wordvec_dim=256,
          **to_float_cuda)

for learning_rate in [1e-2]:
  print('learning rate is: ', learning_rate)
  CaptioningTrain(model, small_image_data, small_caption_data,
                num_epochs=num_epochs, batch_size=batch_size,
                learning_rate=learning_rate)
```

```
learning rate is:  0.01
(Epoch 0 / 80) loss: 75.1398 time per epoch: 0.0s
(Epoch 1 / 80) loss: 69.0482 time per epoch: 0.0s
(Epoch 2 / 80) loss: 58.8821 time per epoch: 0.0s
(Epoch 3 / 80) loss: 50.1367 time per epoch: 0.0s
(Epoch 4 / 80) loss: 51.0620 time per epoch: 0.0s
(Epoch 5 / 80) loss: 49.1005 time per epoch: 0.0s
(Epoch 6 / 80) loss: 48.1122 time per epoch: 0.0s
(Epoch 7 / 80) loss: 46.0477 time per epoch: 0.0s
(Epoch 8 / 80) loss: 43.2133 time per epoch: 0.0s
(Epoch 9 / 80) loss: 40.2146 time per epoch: 0.0s
(Epoch 10 / 80) loss: 37.9648 time per epoch: 0.0s
(Epoch 11 / 80) loss: 35.2592 time per epoch: 0.0s
(Epoch 12 / 80) loss: 32.9365 time per epoch: 0.0s
(Epoch 13 / 80) loss: 30.4485 time per epoch: 0.0s
(Epoch 14 / 80) loss: 28.7821 time per epoch: 0.0s
(Epoch 15 / 80) loss: 26.8193 time per epoch: 0.0s
(Epoch 16 / 80) loss: 24.2718 time per epoch: 0.0s
(Epoch 17 / 80) loss: 21.6318 time per epoch: 0.0s
(Epoch 18 / 80) loss: 19.1726 time per epoch: 0.0s
(Epoch 19 / 80) loss: 16.9401 time per epoch: 0.0s
(Epoch 20 / 80) loss: 14.5941 time per epoch: 0.0s
(Epoch 21 / 80) loss: 12.6835 time per epoch: 0.0s
(Epoch 22 / 80) loss: 11.0697 time per epoch: 0.0s
(Epoch 23 / 80) loss: 9.6277 time per epoch: 0.0s
(Epoch 24 / 80) loss: 8.4808 time per epoch: 0.0s
(Epoch 25 / 80) loss: 7.4937 time per epoch: 0.0s
(Epoch 26 / 80) loss: 6.6500 time per epoch: 0.0s
(Epoch 27 / 80) loss: 6.0174 time per epoch: 0.0s
(Epoch 28 / 80) loss: 5.4441 time per epoch: 0.0s
(Epoch 29 / 80) loss: 5.0898 time per epoch: 0.0s
(Epoch 30 / 80) loss: 4.7773 time per epoch: 0.0s
(Epoch 31 / 80) loss: 4.5587 time per epoch: 0.0s
(Epoch 32 / 80) loss: 4.3982 time per epoch: 0.0s
(Epoch 33 / 80) loss: 4.2745 time per epoch: 0.0s
(Epoch 34 / 80) loss: 4.2004 time per epoch: 0.0s
(Epoch 35 / 80) loss: 4.1429 time per epoch: 0.0s
(Epoch 36 / 80) loss: 4.0935 time per epoch: 0.0s
(Epoch 37 / 80) loss: 4.0561 time per epoch: 0.0s
(Epoch 38 / 80) loss: 4.0342 time per epoch: 0.0s
(Epoch 39 / 80) loss: 4.0136 time per epoch: 0.0s
(Epoch 40 / 80) loss: 4.0029 time per epoch: 0.0s
(Epoch 41 / 80) loss: 3.9867 time per epoch: 0.0s
(Epoch 42 / 80) loss: 3.9729 time per epoch: 0.0s
(Epoch 43 / 80) loss: 3.9742 time per epoch: 0.0s
(Epoch 44 / 80) loss: 3.9644 time per epoch: 0.0s
(Epoch 45 / 80) loss: 3.9593 time per epoch: 0.0s
(Epoch 46 / 80) loss: 3.9505 time per epoch: 0.0s
(Epoch 47 / 80) loss: 3.9480 time per epoch: 0.0s
(Epoch 48 / 80) loss: 3.9447 time per epoch: 0.0s
(Epoch 49 / 80) loss: 3.9374 time per epoch: 0.0s
(Epoch 50 / 80) loss: 3.9423 time per epoch: 0.0s
(Epoch 51 / 80) loss: 3.9375 time per epoch: 0.0s
(Epoch 52 / 80) loss: 3.9353 time per epoch: 0.0s
(Epoch 53 / 80) loss: 3.9341 time per epoch: 0.0s
(Epoch 54 / 80) loss: 3.9322 time per epoch: 0.0s
(Epoch 55 / 80) loss: 3.9310 time per epoch: 0.0s
(Epoch 56 / 80) loss: 3.9298 time per epoch: 0.0s
(Epoch 57 / 80) loss: 3.9284 time per epoch: 0.0s
(Epoch 58 / 80) loss: 3.9283 time per epoch: 0.0s
(Epoch 59 / 80) loss: 3.9277 time per epoch: 0.0s
```

```
(Epoch 60 / 80) loss: 3.9259 time per epoch: 0.0s
(Epoch 61 / 80) loss: 3.9306 time per epoch: 0.0s
(Epoch 62 / 80) loss: 3.9249 time per epoch: 0.0s
(Epoch 63 / 80) loss: 3.9269 time per epoch: 0.0s
(Epoch 64 / 80) loss: 3.9267 time per epoch: 0.0s
(Epoch 65 / 80) loss: 3.9255 time per epoch: 0.0s
(Epoch 66 / 80) loss: 3.9245 time per epoch: 0.0s
(Epoch 67 / 80) loss: 3.9236 time per epoch: 0.0s
(Epoch 68 / 80) loss: 3.9232 time per epoch: 0.0s
(Epoch 69 / 80) loss: 3.9235 time per epoch: 0.0s
(Epoch 70 / 80) loss: 3.9220 time per epoch: 0.0s
(Epoch 71 / 80) loss: 3.9220 time per epoch: 0.0s
(Epoch 72 / 80) loss: 3.9217 time per epoch: 0.0s
(Epoch 73 / 80) loss: 3.9212 time per epoch: 0.0s
(Epoch 74 / 80) loss: 3.9208 time per epoch: 0.0s
(Epoch 75 / 80) loss: 3.9208 time per epoch: 0.0s
(Epoch 76 / 80) loss: 3.9201 time per epoch: 0.0s
(Epoch 77 / 80) loss: 3.9203 time per epoch: 0.0s
(Epoch 78 / 80) loss: 3.9200 time per epoch: 0.0s
(Epoch 79 / 80) loss: 3.9196 time per epoch: 0.0s
```



Caption sampling

Modify the `sample` method in class `CaptioningRNN` to handle the case where `self.cell_type` is `lstm`. **This should take fewer than 10 lines of code.**

When you are done, run the following to train a captioning model and sample from your the model on some training and validation set samples.

Train the net

Now, perform the training on the entire training set. You should see a final loss less than `2.8`.

```
fix_random_seed(0)

# data input
small_num_train = num_train
```

```
small_num_train = num_train
sample_idx = torch.randint(num_train, size=(small_num_train,), **to_long_cuda)
small_image_data = data_dict['train_images'][sample_idx].to('cuda')
small_caption_data = data_dict['train_captions'][sample_idx].to('cuda')

# optimization arguments
num_epochs = 60
batch_size = 250

# create the image captioning model
lstm_model = CaptioningRNN(
          cell_type='lstm',
          word_to_idx=data_dict['vocab']['token_to_idx'],
          input_dim=1280, # hard-coded, do not modify
          hidden_dim=512,
          wordvec_dim=256,
          **to_float_cuda)

for learning_rate in [1e-3]:
  print('learning rate is: ', learning_rate)
  CaptioningTrain(lstm_model, small_image_data, small_caption_data,
                num_epochs=num_epochs, batch_size=batch_size,
                learning_rate=learning_rate)
```

```
learning rate is:  0.001
(Epoch 0 / 60) loss: 48.0273 time per epoch: 5.4s
(Epoch 1 / 60) loss: 41.6710 time per epoch: 5.4s
(Epoch 2 / 60) loss: 38.3474 time per epoch: 5.4s
(Epoch 3 / 60) loss: 35.8686 time per epoch: 5.4s
(Epoch 4 / 60) loss: 33.2827 time per epoch: 5.3s
(Epoch 5 / 60) loss: 31.2372 time per epoch: 5.4s
(Epoch 6 / 60) loss: 29.5754 time per epoch: 5.5s
(Epoch 7 / 60) loss: 27.8342 time per epoch: 5.5s
(Epoch 8 / 60) loss: 26.5170 time per epoch: 5.4s
(Epoch 9 / 60) loss: 25.2786 time per epoch: 5.4s
(Epoch 10 / 60) loss: 23.9453 time per epoch: 5.4s
(Epoch 11 / 60) loss: 22.7346 time per epoch: 5.4s
(Epoch 12 / 60) loss: 21.4632 time per epoch: 5.4s
(Epoch 13 / 60) loss: 20.3216 time per epoch: 5.4s
(Epoch 14 / 60) loss: 19.0960 time per epoch: 5.3s
(Epoch 15 / 60) loss: 18.0609 time per epoch: 5.3s
(Epoch 16 / 60) loss: 17.0149 time per epoch: 5.3s
(Epoch 17 / 60) loss: 16.1505 time per epoch: 5.4s
(Epoch 18 / 60) loss: 15.0134 time per epoch: 5.4s
(Epoch 19 / 60) loss: 13.8686 time per epoch: 5.5s
(Epoch 20 / 60) loss: 12.9711 time per epoch: 5.4s
(Epoch 21 / 60) loss: 12.1076 time per epoch: 5.3s
(Epoch 22 / 60) loss: 11.7461 time per epoch: 5.3s
(Epoch 23 / 60) loss: 10.9183 time per epoch: 5.3s
(Epoch 24 / 60) loss: 10.1189 time per epoch: 5.3s
(Epoch 25 / 60) loss: 9.3973 time per epoch: 5.4s
(Epoch 26 / 60) loss: 8.7725 time per epoch: 5.3s
(Epoch 27 / 60) loss: 8.5292 time per epoch: 5.3s
(Epoch 28 / 60) loss: 7.6079 time per epoch: 5.4s
(Epoch 29 / 60) loss: 6.9225 time per epoch: 5.4s
(Epoch 30 / 60) loss: 6.5833 time per epoch: 5.4s
(Epoch 31 / 60) loss: 6.2431 time per epoch: 5.4s
(Epoch 32 / 60) loss: 5.8997 time per epoch: 5.4s
(Epoch 33 / 60) loss: 6.0608 time per epoch: 5.4s
(Epoch 34 / 60) loss: 5.5518 time per epoch: 5.4s
(Epoch 35 / 60) loss: 5.1741 time per epoch: 5.4s
(Epoch 36 / 60) loss: 4.6947 time per epoch: 5.4s
(Epoch 37 / 60) loss: 3.7801 time per epoch: 5.3s
(Epoch 38 / 60) loss: 3.2414 time per epoch: 5.4s
(Epoch 39 / 60) loss: 2.9094 time per epoch: 5.5s
(Epoch 40 / 60) loss: 2.9248 time per epoch: 5.4s
(Epoch 41 / 60) loss: 2.5309 time per epoch: 5.3s
(Epoch 42 / 60) loss: 2.2791 time per epoch: 5.4s
(Epoch 43 / 60) loss: 2.7174 time per epoch: 5.4s
(Epoch 44 / 60) loss: 2.1239 time per epoch: 5.3s
(Epoch 45 / 60) loss: 1.9405 time per epoch: 5.4s
(Epoch 46 / 60) loss: 1.8847 time per epoch: 5.4s
(Epoch 47 / 60) loss: 1.7666 time per epoch: 5.3s
(Epoch 48 / 60) loss: 1.6542 time per epoch: 5.3s
(Epoch 49 / 60) loss: 1.4997 time per epoch: 5.3s
(Epoch 50 / 60) loss: 1.8304 time per epoch: 5.3s
(Epoch 51 / 60) loss: 1.3259 time per epoch: 5.4s
(Epoch 52 / 60) loss: 1.4615 time per epoch: 5.4s
(Epoch 53 / 60) loss: 0.9938 time per epoch: 5.3s
```

Test-time sampling

As with the RNN, the samples on training data should be very good; the samples on validation data will probably make less sense.

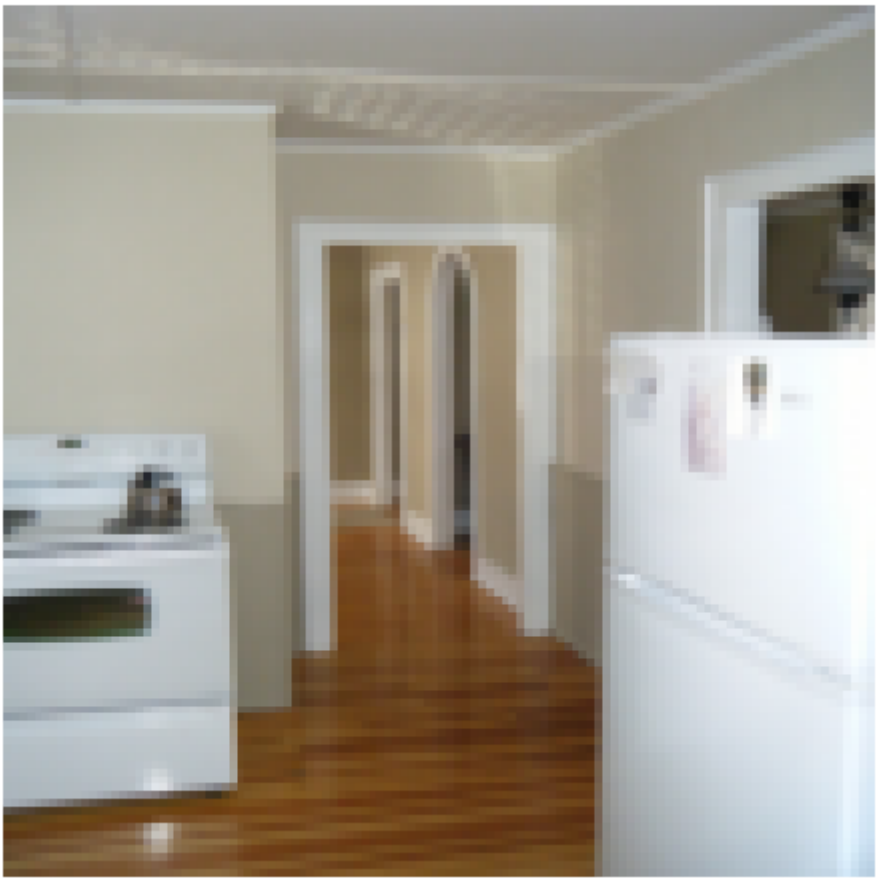
Training loss history

```
# Sample a minibatch and show the reshaped 112x112 images,
# GT captions, and generated captions by your model.
batch_size = 3

for split in ['train', 'val']:
  sample_idx = torch.randint(0, num_train if split=='train' else num_val, (batch_size,))
  sample_images = data_dict[split+'_images'][sample_idx]
  sample_captions = data_dict[split+'_captions'][sample_idx]

  gt_captions = decode_captions(sample_captions, data_dict['vocab']['idx_to_token'])
  lstm_model.eval()
  generated_captions = lstm_model.sample(sample_images)
  generated_captions = decode_captions(generated_captions, data_dict['vocab']['idx_to_toke

  for i in range(batch_size):
    plt.imshow(sample_images[i].permute(1, 2, 0))
    plt.axis('off')
    plt.title('%s\nLSTM Generated:%s\nGT:%s' % (split, generated_captions[i], gt_captions[
    plt.show()
```

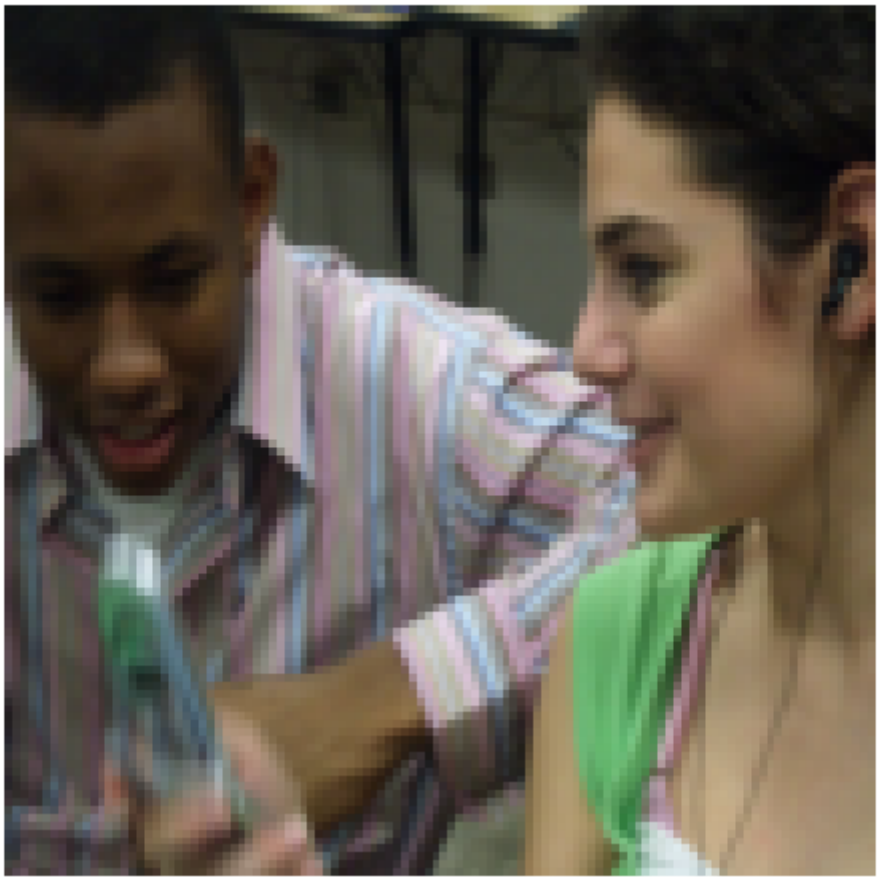
train
LSTM Generated:a person riding skis on a snowy hill <END>
GT:<START> a group of skiers are standing in the snow <END>

train
LSTM Generated:a group of people walking along a dirty snow covered road <END>
GT:<START> a group of people walking along a dirty snow covered road <END>

train
LSTM Generated:a white bathroom sink sitting under a mirror <END>
GT:<START> a view of a kitchen and a <UNK> in a home <END>



val
LSTM Generated:a young boy holding a banana in his hand <END>
GT:<START> a boy and a girl sitting next to each other looking at an <UNK> <UNK> <END>



val
LSTM Generated:a bathroom has a toilet and sink in it <END>
GT:<START> a close up of a white toilet and <UNK> can <END>

Attention LSTM

Attention LSTM essentially adds an attention input $x_{attn}^t \in \mathbb{R}^H$ into LSTM, along with $x_t \in \mathbb{R}^D$ and the previous hidden state $h_{t-1} \in \mathbb{R}^H$.

To get the attention input x_{attn}^t, here we adopt a method called `scaled dot-product attention`, as covered in the lecture. We first project the CNN feature activation from $\mathbb{R}^{1280\times4\times4}$ to $\mathbb{R}^{H\times4\times4}$ using an affine layer. Given the projected activation $A \in \mathbb{R}^{H\times4\times4}$ and the LSTM hidden state from the previous time step h_{t-1}, we formuate the attention weights on A at time step t as $M_{attn}^t = h_{t-1}A/\sqrt{H} \in \mathbb{R}^{4\times4}$.

To simplify the formuation here, we flatten the spatial dimensions of A and M_{attn}^t which gives $\tilde{A} \in \mathbb{R}^{H\times16}$ and $\tilde{M}^t{}_{attn} = h_{t-1}A \in \mathbb{R}^{16}$. We add a **`softmax`** activation function on $\tilde{M}^t{}_{attn}$ so that the attention weights at each time step are normalized and sum up to one.

The attention embedding given the attention weights is then $x_{attn}^t = \tilde{A}\tilde{M}^t{}_{attn} \in \mathbb{R}^H$.

You will implement a batch version of the attention layer we have described here.

Scaled dot-product attention

Implement the scaled dot-product attention function. Given the LSTM hidden state from the previous time step `prev_h` (or h_{t-1}) and the projected CNN feature activation `A`, compute the attention weights `attn_weights` (or $\tilde{M}^t{}_{attn}$ with a reshaping to $\mathbb{R}^{4\times4}$) attention embedding output `attn` (or x_{attn}^t) using the formulation we provided.

```
def dot_product_attention(prev_h, A):
    """
    A simple scaled dot-product attention layer.
    Inputs:
    - prev_h: The LSTM hidden state from the previous time step, of shape (N, H)
    - A: **Projected** CNN feature activation, of shape (N, H, 4, 4),
         where H is the LSTM hidden state size
```

```
    Outputs:
    - attn: Attention embedding output, of shape (N, H)
    - attn_weights: Attention weights, of shape (N, 4, 4)

    """
    N, H, D_a, _ = A.shape

    attn, attn_weights = None, None
    ##############################################################################
    # TODO: Implement the scaled dot-product attention we described earlier.    #
    # You will use this function for `attention_forward` and `sample_caption`   #
    # HINT: Make sure you reshape attn_weights back to (N, 4, 4)!               #
    ##############################################################################
    # Replace "pass" statement with your code
    from math import sqrt
    h_tilt = prev_h.reshape(N, 1, H)
    A_tilt = A.reshape(N, H, -1)
    Matt = (torch.bmm(h_tilt, A_tilt).div(sqrt(H))).reshape(N, -1, 1)
    Matt_tilt = F.softmax(Matt, dim=1)
    # print(Matt_tilt.shape)
    attn = torch.bmm(A_tilt, Matt_tilt).reshape(N, H)
    # print(attn.shape)
    attn_weights = Matt_tilt.reshape(N, 4, 4)
    # print(attn_weights.shape)
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################

    return attn, attn_weights
```

When you are done, run the following to check your implementation. You should see an error on the order of `1e-7` or less.

```
N, H = 2, 5
D_a = 4

prev_h = torch.linspace(-0.4, 0.6, steps=N*H, **to_double_cuda).reshape(N, H)
A = torch.linspace(-0.4, 1.8, steps=N*H*D_a*D_a, **to_double_cuda).reshape(N, H, D_a, D_a)

attn, attn_weights = dot_product_attention(prev_h, A)

expected_attn = torch.tensor([[-0.29784344, -0.07645979,  0.14492386,  0.36630751,  0.5876
        [ 0.81412643,  1.03551008,  1.25689373,  1.47827738,  1.69966103]], **to_double_cu
expected_attn_weights = torch.tensor([[[0.06511126, 0.06475411, 0.06439892, 0.06404568],
         [0.06369438, 0.06334500, 0.06299754, 0.06265198],
         [0.06230832, 0.06196655, 0.06162665, 0.06128861],
         [0.06095243, 0.06061809, 0.06028559, 0.05995491]],

        [[0.05717142, 0.05784357, 0.05852362, 0.05921167],
         [0.05990781, 0.06061213, 0.06132473, 0.06204571],
         [0.06277517, 0.06351320, 0.06425991, 0.06501540],
         [0.06577977, 0.06655312, 0.06733557, 0.06812722]]], **to_double_cuda)
```

```
print('attn error: ', rel_error(expected_attn, attn))
print('attn_weights error: ', rel_error(expected_attn_weights, attn_weights))
```

```
attn error:  1.8611673688572432e-08
attn_weights error:  7.549315963595102e-08
```

Attention LSTM: step forward

Modify the `lstm_step_forward` function from earlier to support the extra attention input `attn` (or x_{attn}) and its embedding weight matrix `Wattn` (or W_{attn}) in the LSTM cell. Hence, at each timestep the *activation vector* $a \in \mathbb{R}^{4H}$ in LSTM cell is formulated as:

$$a = W_x x_t + W_h h_{t-1} + W_{attn} x_{attn}^t + b.$$

This should require adding less than 5 lines of code.

Once you are done, run the following to perform a simple test of your implementation. You should see errors on the order of `1e-8` or less.

```
N, D, H = 3, 4, 5

x = torch.linspace(-0.4, 1.2, steps=N*D, **to_double_cuda).reshape(N, D)
prev_h = torch.linspace(-0.3, 0.7, steps=N*H, **to_double_cuda).reshape(N, H)
prev_c = torch.linspace(-0.4, 0.9, steps=N*H, **to_double_cuda).reshape(N, H)
Wx = torch.linspace(-2.1, 1.3, steps=4*D*H, **to_double_cuda).reshape(D, 4 * H)
Wh = torch.linspace(-0.7, 2.2, steps=4*H*H, **to_double_cuda).reshape(H, 4 * H)
b = torch.linspace(0.3, 0.7, steps=4*H, **to_double_cuda)
attn = torch.linspace(0.6, 1.8, steps=N*H, **to_double_cuda).reshape(N, H)
Wattn = torch.linspace(1.3, 4.2, steps=4*H*H, **to_double_cuda).reshape(H, 4 * H)


next_h, next_c = lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b, attn, Wattn)

expected_next_h = torch.tensor([
    [0.53704256, 0.59980774, 0.65596820, 0.70569729, 0.74932626],
    [0.78729857, 0.82010653, 0.84828362, 0.87235677, 0.89283167],
    [0.91017981, 0.92483119, 0.93717126, 0.94754073, 0.95623746]], **to_double_cuda)
expected_next_c = torch.tensor([
    [0.59999328, 0.69285041, 0.78570758, 0.87856479, 0.97142202],
    [1.06428558, 1.15714276, 1.24999992, 1.34285708, 1.43571424],
    [1.52857143, 1.62142857, 1.71428571, 1.80714286, 1.90000000]], **to_double_cuda)

print('next_h error: ', rel_error(expected_next_h, next_h))
print('next_c error: ', rel_error(expected_next_c, next_c))
```

```
next_h error:  7.749791467532482e-09
next_c error:  3.6613350811635177e-09
```

Attention LSTM: forward

Now, implement the `attention_forward` function to run an Attention LSTM forward on an entire timeseries of data. You will have to use the `dot_product_attention` function and the `lstm_step_forward` function you implemented.

Again, don't worry about the backward part! `autograd` will handle it.

```
def attention_forward(x, A, Wx, Wh, Wattn, b):
    """
    h0 and c0 are same initialized as the global image feature (meanpooled A)
    For simplicity, we implement scaled dot-product attention, which means in
    Eq. 4 of the paper (https://arxiv.org/pdf/1502.03044.pdf),
    f_{att}(a_i, h_{t-1}) equals to the scaled dot product of a_i and h_{t-1}.

    Forward pass for an LSTM over an entire sequence of data. We assume an input
    sequence composed of T vectors, each of dimension D. The LSTM uses a hidden
    size of H, and we work over a minibatch containing N sequences. After running
    the LSTM forward, we return the hidden states for all timesteps.

    Note that the initial cell state is passed as input, but the initial cell
    state is set to zero. Also note that the cell state is not returned; it is
    an internal variable to the LSTM and is not accessed from outside.

    Inputs:
    - x: Input data, of shape (N, T, D)
    - A: **Projected** activation map, of shape (N, H, 4, 4)
    - Wx: Weights for input-to-hidden connections, of shape (D, 4H)
    - Wh: Weights for hidden-to-hidden connections, of shape (H, 4H)
    - Wattn: Weights for attention-to-hidden connections, of shape (H, 4H)
    - b: Biases, of shape (4H,)

    Returns a tuple of:
    - h: Hidden states for all timesteps of all sequences, of shape (N, T, H)
    """

    h = None

    # The initial hidden state h0 and cell state c0 are initialized differently in
    # Attention LSTM from the original LSTM and hence we provided them for you.
    h0 = A.mean(dim=(2, 3)) # Initial hidden state, of shape (N, H)
    c0 = h0 # Initial cell state, of shape (N, H)

    ##############################################################################
    # TODO: Implement the forward pass for an LSTM over an entire timeseries.    #
    # You should use the lstm_step_forward function and dot_product_attention    #
    # function that you just defined.                                            #
    ##############################################################################
    # Replace "pass" statement with your code
    N, T, D = x.shape
    _, H = h0.shape
    h = torch.zeros((N, T, H), dtype=h0.dtype, device=h0.device)
    prev_h = h0
    prev_c = c0
    for i in range(T):
```

```
        attn, attn_weights = dot_product_attention(prev_h, A)
        # print(attn.shape)
        # print(Wattn.shape)
        next_h, next_c = lstm_step_forward(x[:,i,:], prev_h, prev_c, Wx, Wh, b, attn=attn, W
        prev_h = next_h
        prev_c = next_c
        h[:, i, :] = prev_h
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################

    return h
```

When you are done, run the following to check your implementation. You should see an error on the order of `1e-8` or less.

```
N, D, H, T = 2, 5, 4, 3
D_a = 4

x = torch.linspace(-0.4, 0.6, steps=N*T*D, **to_double_cuda).reshape(N, T, D)
A = torch.linspace(-0.4, 1.8, steps=N*H*D_a*D_a, **to_double_cuda).reshape(N, H, D_a, D_a)
Wx = torch.linspace(-0.2, 0.9, steps=4*D*H, **to_double_cuda).reshape(D, 4 * H)
Wh = torch.linspace(-0.3, 0.6, steps=4*H*H, **to_double_cuda).reshape(H, 4 * H)
Wattn = torch.linspace(1.3, 4.2, steps=4*H*H, **to_double_cuda).reshape(H, 4 * H)
b = torch.linspace(0.2, 0.7, steps=4*H, **to_double_cuda)

h = attention_forward(x, A, Wx, Wh, Wattn, b)

expected_h = torch.tensor([
        [[0.56141729, 0.70274849, 0.80000386, 0.86349400],
         [0.89556391, 0.92856726, 0.94950579, 0.96281018],
         [0.96792077, 0.97535465, 0.98039623, 0.98392994]],

        [[0.95065880, 0.97135490, 0.98344373, 0.99045552],
         [0.99317679, 0.99607466, 0.99774317, 0.99870293],
         [0.99907382, 0.99946784, 0.99969426, 0.99982435]]], **to_double_cuda)

print('h error: ', rel_error(expected_h, h))
```

```
h error:  5.953926115196726e-09
```

Attention LSTM Module

We can now wrap the Attention LSTM functions we wrote into an nn.Module.

```
class AttentionLSTM(nn.Module):
  """
  This is our single-layer, uni-directional Attention module.

  Arguments for initialization:
  - input_size: Input size, denoted as D before
```

```
    - hidden_size: Hidden size, denoted as H before
    """
    def __init__(self, input_size, hidden_size, device='cpu',
                 dtype=torch.float32):
      """
      Initialize a LSTM.
      Model parameters to initialize:
      - Wx: Weights for input-to-hidden connections, of shape (D, 4H)
      - Wh: Weights for hidden-to-hidden connections, of shape (H, 4H)
      - Wattn: Weights for attention-to-hidden connections, of shape (H, 4H)
      - b: Biases, of shape (4H,)
      """
      super().__init__()

      # Register parameters
      self.Wx = Parameter(torch.randn(input_size, hidden_size*4,
                        device=device, dtype=dtype).div(math.sqrt(input_size)))
      self.Wh = Parameter(torch.randn(hidden_size, hidden_size*4,
                        device=device, dtype=dtype).div(math.sqrt(hidden_size)))
      self.Wattn = Parameter(torch.randn(hidden_size, hidden_size*4,
                        device=device, dtype=dtype).div(math.sqrt(hidden_size)))
      self.b = Parameter(torch.zeros(hidden_size*4,
                        device=device, dtype=dtype))

    def forward(self, x, A):
      """
      Inputs:
      - x: Input data for the entire timeseries, of shape (N, T, D)
      - A: The projected CNN feature activation, of shape (N, H, 4, 4)

      Outputs:
      - hn: The hidden state output
      """
      hn = attention_forward(x, A, self.Wx, self.Wh, self.Wattn, self.b)
      return hn

    def step_forward(self, x, prev_h, prev_c, attn):
      """
      Inputs:
      - x: Input data for one time step, of shape (N, D)
      - prev_h: The previous hidden state, of shape (N, H)
      - prev_c: The previous cell state, of shape (N, H)
      - attn: The attention embedding, of shape (N, H)

      Outputs:
      - next_h: The next hidden state, of shape (N, H)
      - next_c: The next cell state, of shape (N, H)
      """
      next_h, next_c = lstm_step_forward(x, prev_h, prev_c, self.Wx, self.Wh,
                                         self.b, attn=attn, Wattn=self.Wattn)
      return next_h, next_c
```

Attention LSTM captioning model

Now that you have implemented an attention module, update the implementation of the `init` method and `forward` method in module `CaptioningRNN` to also handle the case where `self.cell_type` is `attention`. **This should require adding less than 10 lines of code.**

Once you have done so, run the following to check your implementation. You should see a difference on the order of `1e-7` or less.

```
fix_random_seed(0)

N, D, W, H = 10, 1280, 30, 40
D_img = 112
word_to_idx = {'<NULL>': 0, 'cat': 2, 'dog': 3}
V = len(word_to_idx)
T = 13

model = CaptioningRNN(word_to_idx,
          input_dim=D,
          wordvec_dim=W,
          hidden_dim=H,
          cell_type='attention',
          **to_float_cuda)

for k,v in model.named_parameters():
  # print(k, v.shape) # uncomment this to see the weight shape
  v.data.copy_(torch.linspace(-1.4, 1.3, steps=v.numel()).reshape(*v.shape))

images = torch.linspace(-3., 3., steps=(N * 3 * D_img * D_img),
                        **to_float_cuda).reshape(N, 3, D_img, D_img)
captions = (torch.arange(N * T, **to_long_cuda) % V).reshape(N, T)

loss = model(images, captions).item()
expected_loss = 46.9113769531

print('loss: ', loss)
print('expected loss: ', expected_loss)
print('difference: ', rel_error(torch.tensor(loss), torch.tensor(expected_loss)))
```

```
loss:  46.9113883972168
expected loss:  46.9113769531
difference:  2.4395131958954153e-07
```

Overfit small data

We have written this part for you. Run the following to overfit an Attention LSTM captioning model on the same small dataset as we used for the RNN previously. You should see a final loss less than `9`.

```
fix_random_seed(0)

# data input
small_num_train = 50
```

```
sample_idx = torch.linspace(0, num_train-1, steps=small_num_train, **to_float_cuda).long()
small_image_data = data_dict['train_images'][sample_idx].to('cuda')
small_caption_data = data_dict['train_captions'][sample_idx].to('cuda')

# optimization arguments
num_epochs = 80
batch_size = 50

# create the image captioning model
model = CaptioningRNN(
          cell_type='attention',
          word_to_idx=data_dict['vocab']['token_to_idx'],
          input_dim=1280, # hard-coded, do not modify
          hidden_dim=512,
          wordvec_dim=256,
          **to_float_cuda)

for learning_rate in [1e-3]:
  print('learning rate is: ', learning_rate)
  CaptioningTrain(model, small_image_data, small_caption_data,
                num_epochs=num_epochs, batch_size=batch_size,
                learning_rate=learning_rate)
```