

CS280 Spring 2023 Assignment 3

Part A

RNN, LSTM and GRU

April 13, 2023

Name: Bingnan Li

Student ID: 2020533092

1. Parity-check network (16 points)

Note that the initial parity bit is 1, what's the relation between each input and the previous parity bit? Determine the relation between the parity and inputs and complete the parity bits(p_1, p_2, p_3, p_4) and design and draw a RNN to predict parity.

$$\begin{array}{l} f\text{Parity bits :} \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad p_1 \quad p_2 \quad p_3 \quad p_4 \quad \rightarrow \\ \text{Input :} \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \end{array}$$

Solution 1 The relation between the parity and inputs is that the parity bit p_i is 1 if the number of 0s from x_0 to x_i is even and 0 if the number of 0s from x_0 to x_i is odd. With the formula above, we can get the parity bits as follows: $p_1 = 1$, $p_2 = 1$, $p_3 = 0$, $p_4 = 1$. The corresponding RNN is shown in Figure 1.

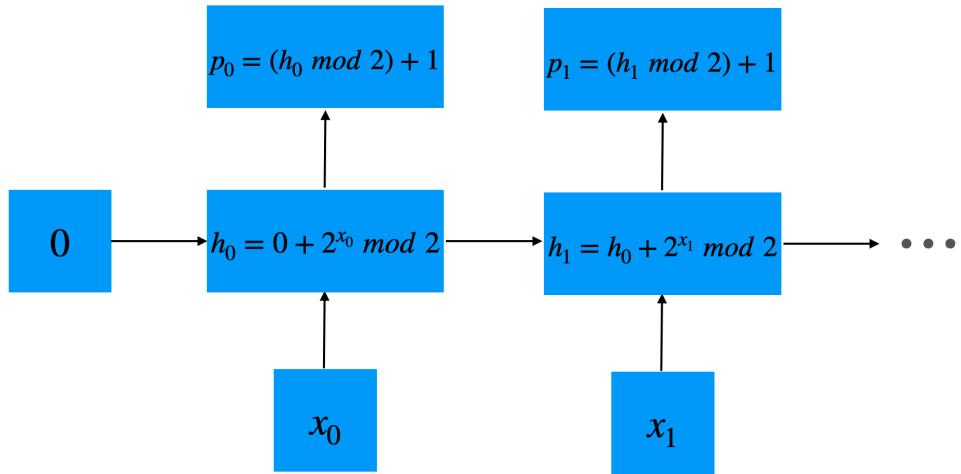


Figure 1: RNN

2. GRU (17 points)

1. Please describe the forward propagation process of LSTM, and analyze the reasons for the disappearance of gradient.
2. Please describe the differences between LSTM and GRU.
3. Please provide examples of scenarios where GRU and LSTM are used respectively. Explain your reason.

Solution 2

1. *The LSTM takes the current input x_t , previous cell state C_{t-1} and hidden state h_{t-1} as inputs. The forward process consists three parts: forget gate, input gate and output gate.*

Forget gate: *This gate takes in x_t and h_{t-1} and output f_t which determines how much information is removed from C_{t-1} . The corresponding formula is*

$$f_t = \delta(W_f[h_{t-1}, x_t]) + b_f$$

where δ is the sigmoid function, W_f, b_f are the weight matrix and the bias term.

Input gate: *This gate takes in x_t and h_{t-1} and output new cell information \tilde{C}_t and i_t which determines how much information is added into cell state. The corresponding formula is*

$$i_t = \delta(W_i[h_{t-1}, x_t] + b_i) \quad \tilde{C}_t = \tanh(W_C[h_{t-1}, x_t] + b_C)$$

Cell state update: *This process utilizes $f_t, C_{t-1}, i_t, \tilde{C}_t$ to output the new cell state C_t with formula:*

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$$

Output gate: *This gate takes in x_t, h_{t-1} and C_t and output the new hidden state h_t and o_t which determines how much information is used to generate h_t . The corresponding formula is*

$$o_t = \delta(W_o[h_{t-1}, x_t] + b_o)) \quad h_t = o_t \cdot \tanh(C_t)$$

Why gradient vanish? Consider the gradient of C_t , by chain rule, we have

$$\bar{C}_t = \bar{C}_{t+1} \frac{\partial f_{t+1} C_t + i_{t+1} \tilde{C}_{t+1}}{\partial C_t} = \bar{C}_{t+1} f_{t+1}$$

Thus,

$$\bar{C}_1 = \bar{C}_T \prod_{i=2}^T f_i$$

When one of the f_i is 0 or all the f_i s are very small, then the gradient will goes to 0.

2. *GRU is a simplified variant of LSTM with three main modifications:*

1. *GRU combines the forget and input gate into an update gate.*
2. *GRU combines the cell state and the hidden state.*
3. *GRU resets the previous hidden state before using it to extract the new information from the current input.*
3. *GRU is used when we require the performance and the input sequence is relatively short while LSTM is used when the input sequence is much longer. This is because GRU is simpler and has much faster inference time. Moreover, since GRU combines the cell state with hidden state, the long memory capability will be relatively weaker than LSTM.*

RNN_Captioning

April 13, 2023

1 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.

1.1 Install h5py

The COCO dataset we will be using is stored in HDF5 format. To load HDF5 files, we will need to install the `h5py` Python package. From the command line, run: `pip install h5py` If you receive a permissions error, you may need to run the command as root: `sudo pip install h5py`

You can also run commands directly from the Jupyter notebook by prefixing the command with the “!” character:

```
[1]: !pip install h5py
```

```
Collecting h5py
  Downloading h5py-3.8.0-cp39-cp39-macosx_11_0_arm64.whl (2.6 MB)
    2.6/2.6 MB
  8.5 MB/s eta 0:00:00a 0:00:01
Requirement already satisfied: numpy>=1.14.5 in
/Users/caleblee/Library/Python/3.9/lib/python/site-packages (from h5py) (1.24.2)
Installing collected packages: h5py
Successfully installed h5py-3.8.0
```

```
[notice] A new release of pip
available: 22.3.1 -> 23.0.1
[notice] To update, run:
pip install --upgrade pip
```

```
[2]: # As usual, a bit of setup
import time, os, json
import numpy as np
import matplotlib.pyplot as plt

from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
from cs231n.rnn_layers import *
from cs231n.captioning_solver import CaptioningSolver
```

```

from cs231n.classifiers.rnn import CaptioningRNN
from cs231n.coco_utils import load_coco_data, sample_coco_minibatch, □
    ↳decode_captions
from cs231n.image_utils import image_from_url

%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'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/
    ↳autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))

```

2 Microsoft COCO

For this exercise we will use the 2014 release of the [Microsoft COCO dataset](#) 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.

You should have already downloaded the data by changing to the `cs231n/datasets` directory and running the script `get_assignment3_data.sh`. If you haven't yet done so, run that script now. Warning: the COCO data download is ~1GB.

We have preprocessed the data and extracted features for you already. For all images we have extracted features from the fc7 layer of the VGG-16 network pretrained on ImageNet; these features are stored in the files `train2014_vgg16_fc7.h5` and `val2014_vgg16_fc7.h5` respectively. To cut down on processing time and memory requirements, we have reduced the dimensionality of the features from 4096 to 512; these features can be found in the files `train2014_vgg16_fc7_pca.h5` and `val2014_vgg16_fc7_pca.h5`.

The raw images take up a lot of space (nearly 20GB) so we have not included them in the download. However all images are taken from Flickr, and URLs of the training and validation images are stored in the files `train2014_urls.txt` and `val2014_urls.txt` respectively. This allows you to download images on the fly for visualization. Since images are downloaded on-the-fly, **you must be connected to the internet to view images**.

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 in the file `coco2014_vocab.json`, and you can use the function `decode_captions` from the file `cs231n/coco_utils.py` to convert numpy arrays 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.

You can load all of the MS-COCO data (captions, features, URLs, and vocabulary) using the `load_coco_data` function from the file `cs231n/coco_utils.py`. Run the following cell to do so:

```
[3]: # Load COCO data from disk; this returns a dictionary
# We'll work with dimensionality-reduced features for this notebook, but feel
# free to experiment with the original features by changing the flag below.
data = load_coco_data(pca_features=True)

# Print out all the keys and values from the data dictionary
for k, v in data.items():
    if type(v) == np.ndarray:
        print(k, type(v), v.shape, v.dtype)
    else:
        print(k, type(v), len(v))
```

```
base dir /Users/caleblee/Desktop/CS280/CS280-Spring23-Assignment3/Part_B_Coding
/cs231n/datasets/coco_captioning
trainCaptions <class 'numpy.ndarray'> (400135, 17) int32
trainImageIdxs <class 'numpy.ndarray'> (400135,) int32
valCaptions <class 'numpy.ndarray'> (195954, 17) int32
valImageIdxs <class 'numpy.ndarray'> (195954,) int32
trainFeatures <class 'numpy.ndarray'> (82783, 512) float32
valFeatures <class 'numpy.ndarray'> (40504, 512) float32
idx_to_word <class 'list'> 1004
word_to_idx <class 'dict'> 1004
trainUrls <class 'numpy.ndarray'> (82783,) <U63
valUrls <class 'numpy.ndarray'> (40504,) <U63
```

2.1 Look at the data

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

You can use the `sample_coco_minibatch` function from the file `cs231n/coco_utils.py` to sample minibatches of data from the data structure returned from `load_coco_data`. 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 and that we download the images on-the-fly using their Flickr URL, so **you must be connected to the internet to view images**.

```
[4]: # Sample a minibatch and show the images and captions
batch_size = 3

captions, features, urls = sample_coco_minibatch(data, batch_size=batch_size)
for i, (caption, url) in enumerate(zip(captions, urls)):
    plt.imshow(image_from_url(url))
    plt.axis('off')
    caption_str = decode_caption(caption, data['idx_to_word'])
    plt.title(caption_str)
    plt.show()
```

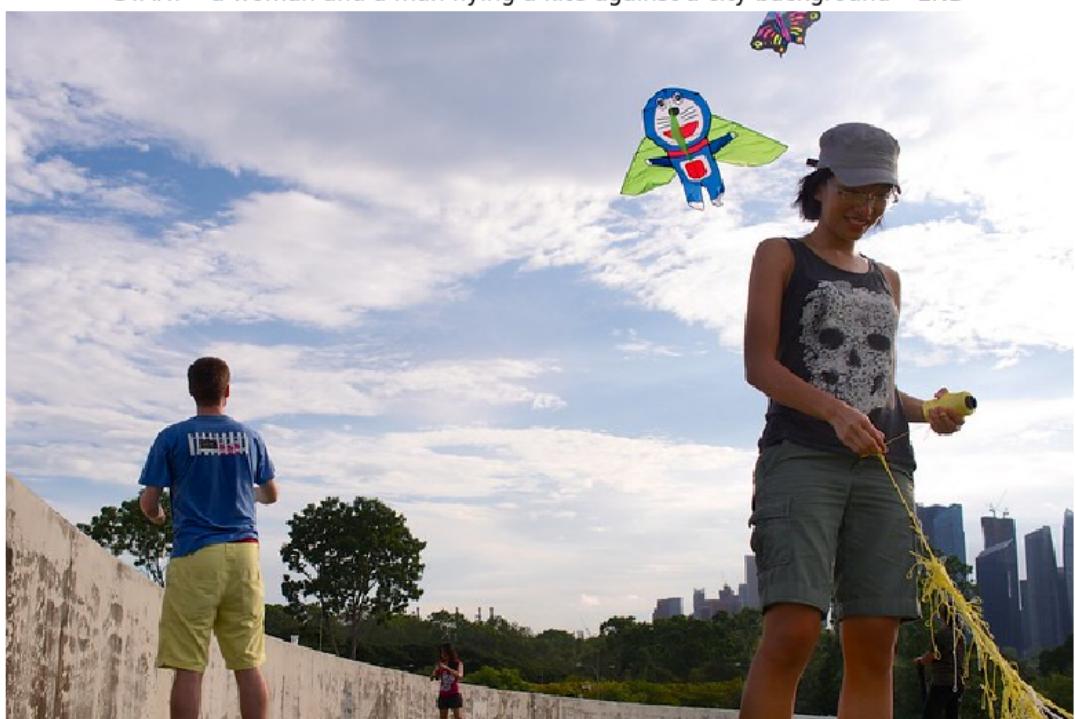
<START> a guy giving a <UNK> up holding a skateboard <END>



<START> a dog sleeping next to a person covered up by a <UNK> in a bed <END>



<START> a woman and a man flying a kite against a city background <END>



3 Recurrent Neural Networks

As discussed in lecture, we will use recurrent neural network (RNN) language models for image captioning. The file `cs231n/rnn_layers.py` contains implementations of different layer types that are needed for recurrent neural networks, and the file `cs231n/classifiers/rnn.py` uses these layers to implement an image captioning model.

We will first implement different types of RNN layers in `cs231n/rnn_layers.py`.

4 Vanilla RNN: step forward

Open the file `cs231n/rnn_layers.py`. This file implements the forward and backward passes for different types of layers that are commonly used in recurrent neural networks.

First implement the function `rnn_step_forward` which implements the forward pass for a single timestep of a vanilla recurrent neural network. After doing so run the following to check your implementation. You should see errors on the order of e-8 or less.

```
[6]: N, D, H = 3, 10, 4

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

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

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

next_h error: 6.292421426471037e-09

5 Vanilla RNN: step backward

In the file `cs231n/rnn_layers.py` implement the `rnn_step_backward` function. After doing so run the following to numerically gradient check your implementation. You should see errors on the order of e-8 or less.

```
[8]: from cs231n.rnn_layers import rnn_step_forward, rnn_step_backward
np.random.seed(231)
N, D, H = 4, 5, 6
x = np.random.randn(N, D)
h = np.random.randn(N, H)
Wx = np.random.randn(D, H)
Wh = np.random.randn(H, H)
```

```

b = np.random.randn(H)

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

dnext_h = np.random.randn(*out.shape)

fx = lambda x: rnn_step_forward(x, h, Wx, Wh, b)[0]
fh = lambda prev_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 = eval_numerical_gradient_array(fx, x, dnext_h)
dprev_h_num = eval_numerical_gradient_array(fh, h, dnext_h)
dWx_num = eval_numerical_gradient_array(fWx, Wx, dnext_h)
dWh_num = eval_numerical_gradient_array(fWh, Wh, dnext_h)
db_num = eval_numerical_gradient_array(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))

```

```

dx error: 2.779555119801537e-10
dprev_h error: 2.732469876125559e-10
dWx error: 9.709162249874773e-10
dWh error: 5.034242362962849e-10
db error: 1.7087625711225576e-11

```

6 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.

In the file `cs231n/rnn_layers.py`, implement the function `rnn_forward`. This should be implemented using the `rnn_step_forward` function that you defined above. After doing so run the following to check your implementation. You should see errors on the order of e-7 or less.

[11]: N, T, D, H = 2, 3, 4, 5

```

x = np.linspace(-0.1, 0.3, num=N*T*D).reshape(N, T, D)
h0 = np.linspace(-0.3, 0.1, num=N*H).reshape(N, H)
Wx = np.linspace(-0.2, 0.4, num=D*H).reshape(D, H)
Wh = np.linspace(-0.4, 0.1, num=H*H).reshape(H, H)

```

```

b = np.linspace(-0.7, 0.1, num=H)

h, _ = rnn_forward(x, h0, Wx, Wh, b)
expected_h = np.asarray([
    [
        [-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]
    ]
])
print('h error: ', rel_error(expected_h, h))

```

h error: 7.728466151011529e-08

7 Vanilla RNN: backward

In the file `cs231n/rnn_layers.py`, 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. You should see errors on the order of e-6 or less.

```

[19]: np.random.seed(231)

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

x = np.random.randn(N, T, D)
h0 = np.random.randn(N, H)
Wx = np.random.randn(D, H)
Wh = np.random.randn(H, H)
b = np.random.randn(H)

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

dout = np.random.randn(*out.shape)

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 = eval_numerical_gradient_array(fx, x, dout)

```

```

dh0_num = eval_numerical_gradient_array(fh0, h0, dout)
dWx_num = eval_numerical_gradient_array(fWx, Wx, dout)
dWh_num = eval_numerical_gradient_array(fWh, Wh, dout)
db_num = eval_numerical_gradient_array(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))

```

```

dx error: 1.535447791566295e-09
dh0 error: 3.3830818037258833e-09
dWx error: 7.235837766433311e-09
dWh error: 1.3049599974515582e-07
db error: 1.519766144544074e-10

```

8 Word embedding: forward

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.

In the file `cs231n/rnn_layers.py`, implement the function `word_embedding_forward` to convert words (represented by integers) into vectors. Run the following to check your implementation. You should see an error on the order of `e-8` or less.

[21]: N, T, V, D = 2, 4, 5, 3

```

x = np.asarray([[0, 3, 1, 2], [2, 1, 0, 3]])
W = np.linspace(0, 1, num=V*D).reshape(V, D)

out, _ = word_embedding_forward(x, W)
expected_out = np.asarray([
    [[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]]])

```

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

```
out error: 1.000000094736443e-08
```

9 Word embedding: backward

Implement the backward pass for the word embedding function in the function `word_embedding_backward`. After doing so run the following to numerically gradient check your implementation. You should see an error on the order of e-11 or less.

```
[35]: np.random.seed(231)

N, T, V, D = 50, 3, 5, 6
x = np.random.randint(V, size=(N, T))
W = np.random.randn(V, D)

out, cache = word_embedding_forward(x, W)
dout = np.random.randn(*out.shape)
dW = word_embedding_backward(dout, cache)

f = lambda W: word_embedding_forward(x, W)[0]
dW_num = eval_numerical_gradient_array(f, W, dout)

print('dW error: ', rel_error(dW, dW_num))
```

dW error: 3.2774595693100364e-12

10 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. Because this is very similar to the affine layer that you implemented in assignment 2, we have provided this function for you in the `temporal_affine_forward` and `temporal_affine_backward` functions in the file `cs231n/rnn_layers.py`. Run the following to perform numeric gradient checking on the implementation. You should see errors on the order of e-9 or less.

```
[36]: np.random.seed(231)

# Gradient check for temporal affine layer
N, T, D, M = 2, 3, 4, 5
x = np.random.randn(N, T, D)
w = np.random.randn(D, M)
b = np.random.randn(M)

out, cache = temporal_affine_forward(x, w, b)

dout = np.random.randn(*out.shape)

fx = lambda x: temporal_affine_forward(x, w, b)[0]
fw = lambda w: temporal_affine_forward(x, w, b)[0]
fb = lambda b: temporal_affine_forward(x, w, b)[0]
```

```

dx_num = eval_numerical_gradient_array(fx, x, dout)
dw_num = eval_numerical_gradient_array(fw, w, dout)
db_num = eval_numerical_gradient_array(fb, b, dout)

dx, dw, db = temporal_affine_backward(dout, cache)

print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))

```

```

dx error: 2.9215945034030545e-10
dw error: 1.5772088618663602e-10
db error: 3.252200556967514e-11

```

11 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 `mask` array that tells it which elements of the scores count towards the loss.

Since this is very similar to the softmax loss function you implemented in assignment 1, we have implemented this loss function for you; look at the `temporal_softmax_loss` function in the file `cs231n/rnn_layers.py`.

Run the following cell to sanity check the loss and perform numeric gradient checking on the function. You should see an error for `dx` on the order of e-7 or less.

```
[38]: # Sanity check for temporal softmax loss
from cs231n.rnn_layers import temporal_softmax_loss

N, T, V = 100, 1, 10

def check_loss(N, T, V):
    x = 0.001 * np.random.randn(N, T, V)
    y = np.random.randint(V, size=(N, T))
    mask = np.random.rand(N, T) <= p
    print(temporal_softmax_loss(x, y, mask)[0])

check_loss(100, 1, 10, 1.0)    # Should be about 2.3
check_loss(100, 10, 10, 1.0)   # Should be about 23
check_loss(5000, 10, 10, 0.1)  # Should be within 2.2-2.4

# Gradient check for temporal softmax loss
```

```

N, T, V = 7, 8, 9

x = np.random.randn(N, T, V)
y = np.random.randint(V, size=(N, T))
mask = (np.random.rand(N, T) > 0.5)

loss, dx = temporal_softmax_loss(x, y, mask, verbose=False)

dx_num = eval_numerical_gradient(lambda x: temporal_softmax_loss(x, y, mask)[0], x, verbose=False)

print('dx error: ', rel_error(dx, dx_num))

```

```

2.3025273255366447
23.025427014880307
2.245475699933817
dx error: 6.298543382976544e-08

```

12 RNN for image captioning

Now that you have implemented the necessary layers, you can combine them to build an image captioning model. Open the file `cs231n/classifiers/rnn.py` and look at the `CaptioningRNN` class.

Implement the forward and backward pass of the model in the `loss` function. For now you only need to implement the case where `cell_type='rnn'` for vanilla RNNs; you will implement the LSTM case later. After doing so, run the following to check your forward pass using a small test case; you should see error on the order of `e-10` or less.

```

[39]: N, D, W, H = 10, 20, 30, 40
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',
                      dtype=np.float64)

# Set all model parameters to fixed values
for k, v in model.params.items():
    model.params[k] = np.linspace(-1.4, 1.3, num=v.size).reshape(*v.shape)

features = np.linspace(-1.5, 0.3, num=(N * D)).reshape(N, D)
captions = (np.arange(N * T) % V).reshape(N, T)

```

```

loss, grads = model.loss(features, captions)
expected_loss = 9.83235591003

print('loss: ', loss)
print('expected loss: ', expected_loss)
print('difference: ', abs(loss - expected_loss))

```

```

loss: 9.832355910027387
expected loss: 9.83235591003
difference: 2.6130209107577684e-12

```

Run the following cell to perform numeric gradient checking on the `CaptioningRNN` class; you should see errors around the order of `e-6` or less.

```

[40]: np.random.seed(231)

batch_size = 2
timesteps = 3
input_dim = 4
wordvec_dim = 5
hidden_dim = 6
word_to_idx = {'<NULL>': 0, 'cat': 2, 'dog': 3}
vocab_size = len(word_to_idx)

captions = np.random.randint(vocab_size, size=(batch_size, timesteps))
features = np.random.randn(batch_size, input_dim)

model = CaptioningRNN(word_to_idx,
                      input_dim=input_dim,
                      wordvec_dim=wordvec_dim,
                      hidden_dim=hidden_dim,
                      cell_type='rnn',
                      dtype=np.float64,
                      )

loss, grads = model.loss(features, captions)

for param_name in sorted(grads):
    f = lambda _: model.loss(features, captions)[0]
    param_grad_num = eval_numerical_gradient(f, model.params[param_name], u
    ↵verbose=False, h=1e-6)
    e = rel_error(param_grad_num, grads[param_name])
    print('%s relative error: %e' % (param_name, e))

```

```

W_embed relative error: 2.331071e-09
W_proj relative error: 9.974426e-09
W_vocab relative error: 4.274378e-09

```

```
Wh relative error: 1.313259e-08
Wx relative error: 8.455229e-07
b relative error: 9.727213e-10
b_proj relative error: 1.934807e-08
b_vocab relative error: 7.087090e-11
```

13 Overfit small data

Similar to the `Solver` class that we used to train image classification models on the previous assignment, on this assignment we use a `CaptioningSolver` class to train image captioning models. Open the file `cs231n/captioning_solver.py` and read through the `CaptioningSolver` class; it should look very familiar.

Once you have familiarized yourself with the API, run the following to make sure your model overfits a small sample of 100 training examples. You should see a final loss of less than 0.1.

```
[41]: np.random.seed(231)

small_data = load_coco_data(max_train=50)

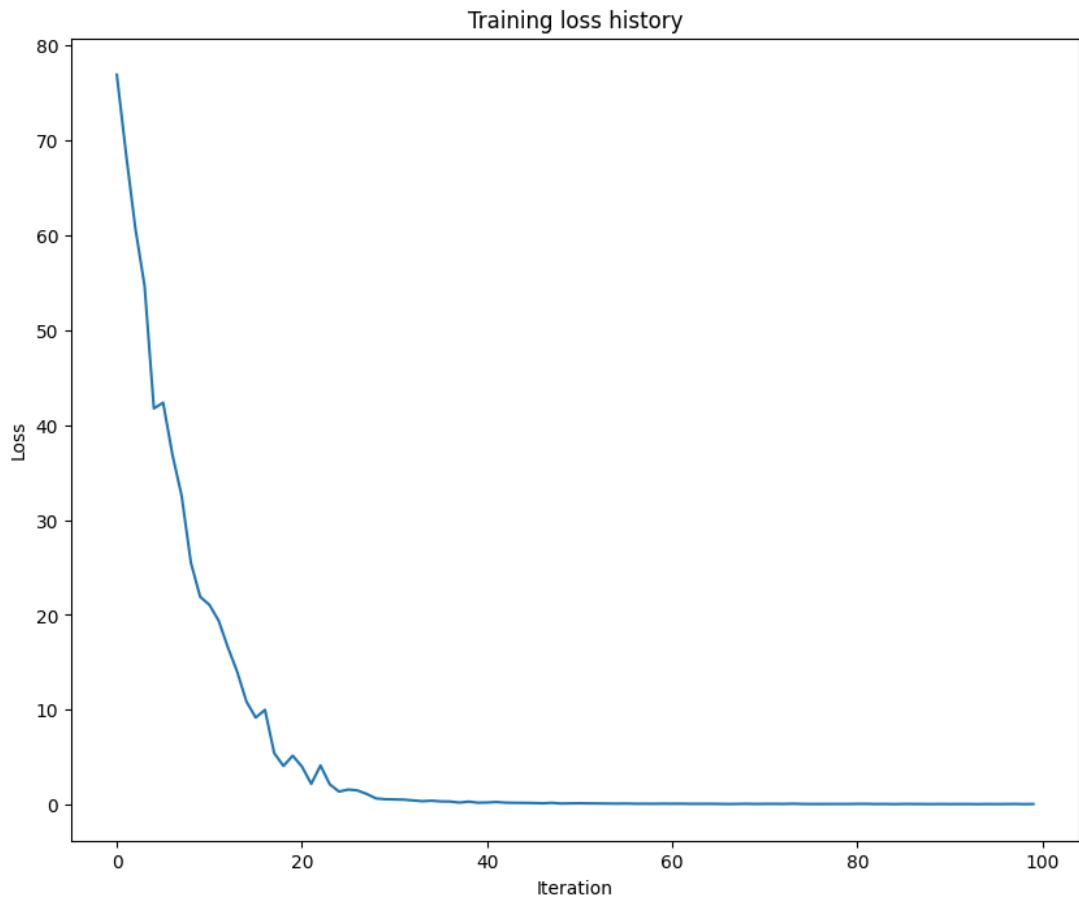
small_rnn_model = CaptioningRNN(
    cell_type='rnn',
    word_to_idx=data['word_to_idx'],
    input_dim=data['train_features'].shape[1],
    hidden_dim=512,
    wordvec_dim=256,
)

small_rnn_solver = CaptioningSolver(small_rnn_model, small_data,
    update_rule='adam',
    num_epochs=50,
    batch_size=25,
    optim_config={
        'learning_rate': 5e-3,
    },
    lr_decay=0.95,
    verbose=True, print_every=10,
)

small_rnn_solver.train()

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

```
base dir  /Users/caleblee/Desktop/CS280/CS280-Spring23-Assignment3/Part_B_Coding  
/cs231n/datasets/coco_captioning  
(Iteration 1 / 100) loss: 76.913486  
(Iteration 11 / 100) loss: 21.063428  
  
/Users/caleblee/Desktop/CS280/CS280-Spring23-Assignment3/Part_B_Coding/cs231n/rn  
n_layers.py:73: RuntimeWarning: overflow encountered in square  
    dz = dnext_h * 4 / ((np.exp(z) + np.exp(-z)) ** 2)  
  
(Iteration 21 / 100) loss: 4.016339  
(Iteration 31 / 100) loss: 0.567104  
(Iteration 41 / 100) loss: 0.239413  
(Iteration 51 / 100) loss: 0.162026  
(Iteration 61 / 100) loss: 0.111557  
(Iteration 71 / 100) loss: 0.097546  
(Iteration 81 / 100) loss: 0.099095  
(Iteration 91 / 100) loss: 0.073986
```



Print final training loss. You should see a final loss of less than 0.1.

```
[42]: print('Final loss: ', small_rnn_solver.loss_history[-1])
```

```
Final loss: 0.08208786010742188
```

14 Test-time 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.

In the file `cs231n/classifiers/rnn.py`, implement the `sample` method for test-time sampling. After doing so, run the following to sample from your overfitted model on both training and validation data. The samples on training data should be very good; the samples on validation data probably won't make sense.

```
[53]: for split in ['train', 'val']:
    minibatch = sample_coco_minibatch(small_data, split=split, batch_size=2)
    gt_captions, features, urls = minibatch
    gt_captions = decode_captions(gt_captions, data['idx_to_word'])

    sample_captions = small_rnn_model.sample(features)
    sample_captions = decode_captions(sample_captions, data['idx_to_word'])

    for gt_caption, sample_caption, url in zip(gt_captions, sample_captions, ↴
                                                urls):
        plt.imshow(image_from_url(url))
        plt.title('%s\n%s\nGT:%s' % (split, sample_caption, gt_caption))
        plt.axis('off')
        plt.show()
```

train

a large truck parked on the cement in front of a building <END>
GT:<START> a large truck parked on the cement in front of a building <END>



train

a plane flying close to the ground as <UNK> coming in for landing <END>
GT:<START> a plane flying close to the ground as <UNK> coming in for landing <END>

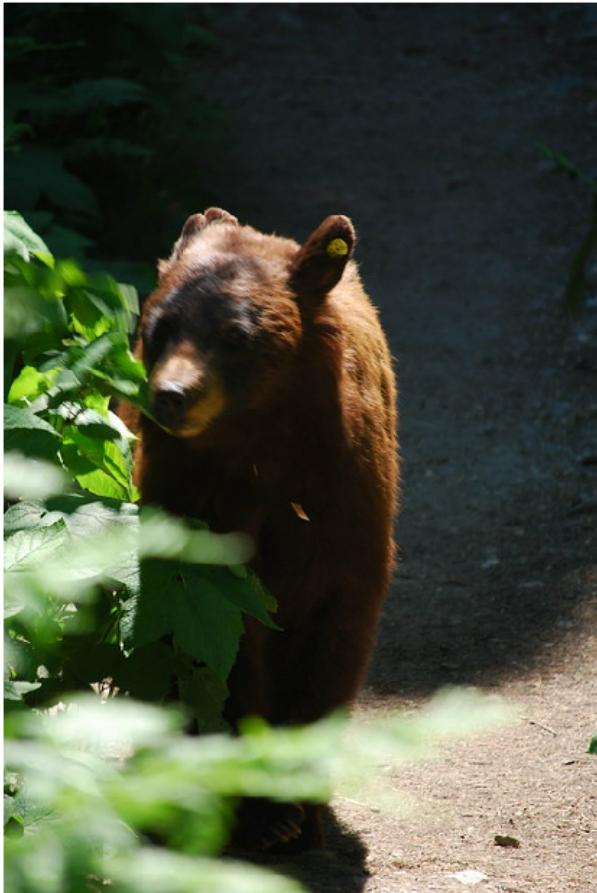


val

half plane flying walk surfer the blanket the sun <END>
GT:<START> a large fork <UNK> stands in the water as a large boat <UNK> <END>



val
with is bear looking building wooded on <UNK> woods hot dogs a drinking dock <END>
GT:<START> a bear standing at the <UNK> between the shade and the <UNK> <END>



15 INLINE QUESTION 1

In our current image captioning setup, our RNN language model produces a word at every timestep as its output. However, an alternate way to pose the problem is to train the network to operate over *characters* (e.g. ‘a’, ‘b’, etc.) as opposed to words, so that at every timestep, it receives the previous character as input and tries to predict the next character in the sequence. For example, the network might generate a caption like

‘A’, ‘ ‘, ‘c’, ‘a’, ‘t’, ‘ ‘, ‘o’, ‘n’, ‘ ‘, ‘a’, ‘ ‘, ‘b’, ‘e’, ‘d’

Can you describe one advantage of an image-captioning model that uses a character-level RNN? Can you also describe one disadvantage? HINT: there are several valid answers, but it might be useful to compare the parameter space of word-level and character-level models.

Your Answer:

Advantage: Using character-level RNN can reduce the number of parameters in the model. This

is because the candidate characters are much less than that of words. For example, the number of characters in English is 26, while the number of words is much larger. **Disadvantage** However, character-level RNN is less robust to spelling errors, which may cause ambiguous captions like “cat -> cet”.

[]:

LSTM_Captioning

April 13, 2023

1 Image Captioning with LSTMs

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

```
[1]: # As usual, a bit of setup
import time, os, json
import numpy as np
import matplotlib.pyplot as plt

from cs231n.gradient_check import eval_numerical_gradient, u
    ↪eval_numerical_gradient_array
from cs231n.rnn_layers import *
from cs231n.captioning_solver import CaptioningSolver
from cs231n.classifiers.rnn import CaptioningRNN
from cs231n.coco_utils import load_coco_data, sample_coco_minibatch, u
    ↪decode_captions
from cs231n.image_utils import image_from_url

%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'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/
    ↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

2 Load MS-COCO data

As in the previous notebook, we will use the Microsoft COCO dataset for captioning.

```
[2]: # Load COCO data from disk; this returns a dictionary
# We'll work with dimensionality-reduced features for this notebook, but feel
# free to experiment with the original features by changing the flag below.
data = load_coco_data(pca_features=True)

# Print out all the keys and values from the data dictionary
for k, v in data.items():
    if type(v) == np.ndarray:
        print(k, type(v), v.shape, v.dtype)
    else:
        print(k, type(v), len(v))
```

```
base dir  /Users/caleblee/Desktop/CS280/CS280-Spring23-Assignment3/Part_B_Coding
/cs231n/datasets/coco_captioning
train_captions <class 'numpy.ndarray'> (400135, 17) int32
train_image_idxs <class 'numpy.ndarray'> (400135,) int32
val_captions <class 'numpy.ndarray'> (195954, 17) int32
val_image_idxs <class 'numpy.ndarray'> (195954,) int32
train_features <class 'numpy.ndarray'> (82783, 512) float32
val_features <class 'numpy.ndarray'> (40504, 512) float32
idx_to_word <class 'list'> 1004
word_to_idx <class 'dict'> 1004
train_urls <class 'numpy.ndarray'> (82783,) <U63
val_urls <class 'numpy.ndarray'> (40504,) <U63
```

3 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

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

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

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

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 $A = X_t W_x + H_{t-1} W_h$

4 LSTM: step forward

Implement the forward pass for a single timestep of an LSTM in the `lstm_step_forward` function in the file `cs231n/rnn_layers.py`. This should be similar to the `rnn_step_forward` function that you implemented above, but using the LSTM update rule instead.

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

```
[4]: N, D, H = 3, 4, 5
x = np.linspace(-0.4, 1.2, num=N*D).reshape(N, D)
prev_h = np.linspace(-0.3, 0.7, num=N*H).reshape(N, H)
prev_c = np.linspace(-0.4, 0.9, num=N*H).reshape(N, H)
Wx = np.linspace(-2.1, 1.3, num=4*D*H).reshape(D, 4 * H)
Wh = np.linspace(-0.7, 2.2, num=4*H*H).reshape(H, 4 * H)
b = np.linspace(0.3, 0.7, num=4*H)

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

expected_next_h = np.asarray([
    [ 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  ]])
expected_next_c = np.asarray([
    [ 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]])

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:  5.7054131967097955e-09
next_c error:  5.8143123088804145e-09
```

5 LSTM: step backward

Implement the backward pass for a single LSTM timestep in the function `lstm_step_backward` in the file `cs231n/rnn_layers.py`. Once you are done, run the following to perform numeric gradient checking on your implementation. You should see errors on the order of `e-7` or less.

```
[7]: np.random.seed(231)

N, D, H = 4, 5, 6
x = np.random.randn(N, D)
prev_h = np.random.randn(N, H)
prev_c = np.random.randn(N, H)
Wx = np.random.randn(D, 4 * H)
Wh = np.random.randn(H, 4 * H)
b = np.random.randn(4 * H)

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

dnext_h = np.random.randn(*next_h.shape)
dnext_c = np.random.randn(*next_c.shape)

fx_h = lambda x: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[0]
fh_h = lambda h: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[0]
fc_h = lambda c: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[0]
fWx_h = lambda Wx: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[0]
fWh_h = lambda Wh: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[0]
fb_h = lambda b: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[0]

fx_c = lambda x: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[1]
fh_c = lambda h: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[1]
fc_c = lambda c: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[1]
fWx_c = lambda Wx: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[1]
fWh_c = lambda Wh: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[1]
fb_c = lambda b: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[1]

num_grad = eval_numerical_gradient_array

dx_num = num_grad(fx_h, x, dnext_h) + num_grad(fx_c, x, dnext_c)
dh_num = num_grad(fh_h, prev_h, dnext_h) + num_grad(fh_c, prev_h, dnext_c)
dc_num = num_grad(fc_h, prev_c, dnext_h) + num_grad(fc_c, prev_c, dnext_c)
dWx_num = num_grad(fWx_h, Wx, dnext_h) + num_grad(fWx_c, Wx, dnext_c)
dWh_num = num_grad(fWh_h, Wh, dnext_h) + num_grad(fWh_c, Wh, dnext_c)
db_num = num_grad(fb_h, b, dnext_h) + num_grad(fb_c, b, dnext_c)

dx, dh, dc, dWx, dWh, db = lstm_step_backward(dnext_h, dnext_c, cache)

print('dx error: ', rel_error(dx_num, dx))
print('dh error: ', rel_error(dh_num, dh))
```

```

print('dc error: ', rel_error(dc_num, dc))
print('dWx error: ', rel_error(dWx_num, dWx))
print('dWh error: ', rel_error(dWh_num, dWh))
print('db error: ', rel_error(db_num, db))

```

```

dx error: 5.833161792422105e-10
dh error: 3.416855254679335e-10
dc error: 1.5221771913099803e-10
dWx error: 1.6933732338045776e-09
dWh error: 2.731092413260415e-08
db error: 1.734944900562979e-10

```

6 LSTM: forward

In the function `lstm_forward` in the file `cs231n/rnn_layers.py`, implement the `lstm_forward` function to run an LSTM forward on an entire timeseries of data.

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

```

[9]: N, D, H, T = 2, 5, 4, 3
x = np.linspace(-0.4, 0.6, num=N*T*D).reshape(N, T, D)
h0 = np.linspace(-0.4, 0.8, num=N*H).reshape(N, H)
Wx = np.linspace(-0.2, 0.9, num=4*D*H).reshape(D, 4 * H)
Wh = np.linspace(-0.3, 0.6, num=4*H*H).reshape(H, 4 * H)
b = np.linspace(0.2, 0.7, num=4*H)

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

expected_h = np.asarray([
    [[ 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]]])

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

```

```

h error: 8.610537442272635e-08

```

7 LSTM: backward

Implement the backward pass for an LSTM over an entire timeseries of data in the function `lstm_backward` in the file `cs231n/rnn_layers.py`. When you are done, run the following to perform numeric gradient checking on your implementation. You should see errors on the order of `e-8` or less. (For `dWh`, it's fine if your error is on the order of `e-6` or less).

```
[18]: from cs231n.rnn_layers import lstm_forward, lstm_backward
np.random.seed(231)

N, D, T, H = 2, 3, 10, 6

x = np.random.randn(N, T, D)
h0 = np.random.randn(N, H)
Wx = np.random.randn(D, 4 * H)
Wh = np.random.randn(H, 4 * H)
b = np.random.randn(4 * H)

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

dout = np.random.randn(*out.shape)

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

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

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

```
dx error: 7.14819501020126e-10
dh0 error: 2.37913534524136e-08
dWx error: 1.0057879800404648e-09
dWh error: 4.72212862848908e-07
db error: 6.252302517938386e-10
```

8 INLINE QUESTION

Recall that in an LSTM the input gate i , forget gate f , and output gate o are all outputs of a sigmoid function. Why don't we use the ReLU activation function instead of sigmoid to compute these values? Explain.

Your Answer: This is because those three gate outputs represents the ratio about information

conservations, replacements and combinations, which should be a number in [0, 1]. However, ReLU is an activation without upperbound. So, we can not replace sigmoid with ReLU.

9 LSTM captioning model

Now that you have implemented an LSTM, update the implementation of the `loss` method of the `CaptioningRNN` class in the file `cs231n/classifiers/rnn.py` to handle the case where `self.cell_type` is `lstm`. 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 e-10 or less.

```
[19]: N, D, W, H = 10, 20, 30, 40
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',
                      dtype=np.float64)

# Set all model parameters to fixed values
for k, v in model.params.items():
    model.params[k] = np.linspace(-1.4, 1.3, num=v.size).reshape(*v.shape)

features = np.linspace(-0.5, 1.7, num=N*D).reshape(N, D)
captions = (np.arange(N * T) % V).reshape(N, T)

loss, grads = model.loss(features, captions)
expected_loss = 9.82445935443

print('loss: ', loss)
print('expected loss: ', expected_loss)
print('difference: ', abs(loss - expected_loss))
```

loss: 9.82445935443226
expected loss: 9.82445935443
difference: 2.261302256556519e-12

10 Overfit LSTM captioning model

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 0.5.

```
[20]: np.random.seed(231)

small_data = load_coco_data(max_train=50)

small_lstm_model = CaptioningRNN(
    cell_type='lstm',
    word_to_idx=data['word_to_idx'],
    input_dim=data['train_features'].shape[1],
    hidden_dim=512,
    wordvec_dim=256,
    dtype=np.float32,
)

small_lstm_solver = CaptioningSolver(small_lstm_model, small_data,
    update_rule='adam',
    num_epochs=50,
    batch_size=25,
    optim_config={
        'learning_rate': 5e-3,
    },
    lr_decay=0.995,
    verbose=True, print_every=10,
)

small_lstm_solver.train()

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

base dir /Users/caleblee/Desktop/CS280/CS280-Spring23-Assignment3/Part_B_Coding
/cs231n/datasets/coco_captioning
(Iteration 1 / 100) loss: 79.551152

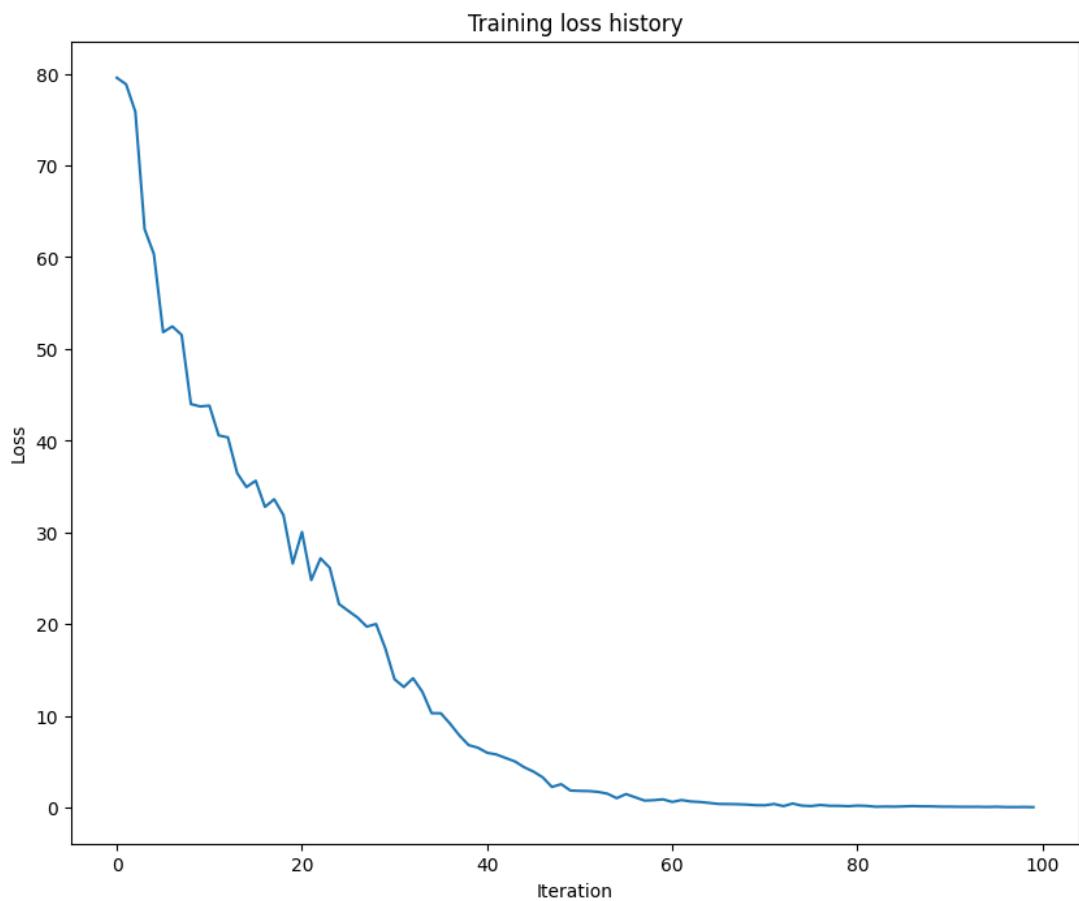
/Users/caleblee/Desktop/CS280/CS280-Spring23-Assignment3/Part_B_Coding/cs231n/rn
n_layers.py:339: RuntimeWarning: overflow encountered in square
dA_g = dG * 4 / (np.exp(A_g) + np.exp(-A_g)) ** 2

(Iteration 11 / 100) loss: 43.829097

/Users/caleblee/Desktop/CS280/CS280-Spring23-Assignment3/Part_B_Coding/cs231n/rn
n_layers.py:339: RuntimeWarning: overflow encountered in exp
dA_g = dG * 4 / (np.exp(A_g) + np.exp(-A_g)) ** 2

(Iteration 21 / 100) loss: 30.062437
(Iteration 31 / 100) loss: 14.020040
```

```
(Iteration 41 / 100) loss: 6.012042
(Iteration 51 / 100) loss: 1.853500
(Iteration 61 / 100) loss: 0.652350
(Iteration 71 / 100) loss: 0.286066
(Iteration 81 / 100) loss: 0.250284
(Iteration 91 / 100) loss: 0.132723
```



Print final training loss. You should see a final loss of less than 0.5.

```
[21]: print('Final loss: ', small_lstm_solver.loss_history[-1])
```

```
Final loss: 0.08398165702819824
```

11 LSTM test-time sampling

Modify the `sample` method of the `CaptioningRNN` class 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 sample from your overfit LSTM model on some training

and validation set samples. As with the RNN, training results should be very good, and validation results probably won't make a lot of sense (because we're overfitting).

```
[27]: for split in ['train', 'val']:
    minibatch = sample_coco_minibatch(small_data, split=split, batch_size=2)
    gt_captions, features, urls = minibatch
    gt_captions = decode_captions(gt_captions, data['idx_to_word'])

    sample_captions = small_lstm_model.sample(features)
    sample_captions = decode_captions(sample_captions, data['idx_to_word'])

    for gt_caption, sample_caption, url in zip(gt_captions, sample_captions, urls):
        plt.imshow(image_from_url(url))
        plt.title('%s\n%s\nGT:%s' % (split, sample_caption, gt_caption))
        plt.axis('off')
        plt.show()
```

train
many people standing near near apples apples <END>
GT:<START> many people standing near boxes of many apples <END>



train

a surfer rides large wave while the the the the the the <END>
GT:<START> a surfer rides a large wave while the sun <UNK> the <UNK> <END>



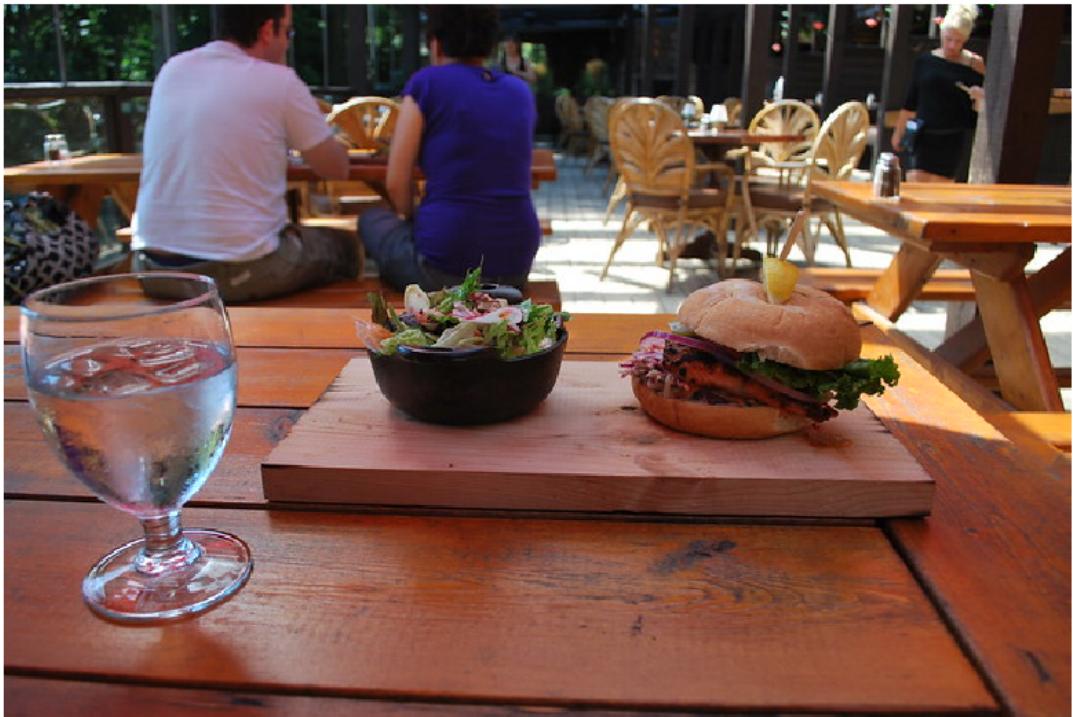
val

papers five five grazing grazing sleeping cute cute cute up on on on the the <END>
GT:<START> a bowl of chicken and vegetables is shown <END>



val

an open open refrigerator refrigerator plays with with on on suitcases <END>
GT:<START> a salad and a sandwich <UNK> to be eaten at a restaurant <END>



[]:

NetworkVisualization-PyTorch

April 13, 2023

1 Network Visualization (PyTorch)

In this notebook we will explore the use of *image gradients* for generating new images.

When training a model, we define a loss function which measures our current unhappiness with the model's performance; we then use backpropagation to compute the gradient of the loss with respect to the model parameters, and perform gradient descent on the model parameters to minimize the loss.

Here we will do something slightly different. We will start from a convolutional neural network model which has been pretrained to perform image classification on the ImageNet dataset. We will use this model to define a loss function which quantifies our current unhappiness with our image, then use backpropagation to compute the gradient of this loss with respect to the pixels of the image. We will then keep the model fixed, and perform gradient descent *on the image* to synthesize a new image which minimizes the loss.

In this notebook we will explore three techniques for image generation:

1. **Saliency Maps:** Saliency maps are a quick way to tell which part of the image influenced the classification decision made by the network.
2. **Fooling Images:** We can perturb an input image so that it appears the same to humans, but will be misclassified by the pretrained network.
3. **Class Visualization:** We can synthesize an image to maximize the classification score of a particular class; this can give us some sense of what the network is looking for when it classifies images of that class.

This notebook uses **PyTorch**; we have provided another notebook which explores the same concepts in TensorFlow. You only need to complete one of these two notebooks.

```
[56]: import torch
import torchvision
import numpy as np
import random
import matplotlib.pyplot as plt
from PIL import Image
from cs231n.image_utils import SQUEEZENET_MEAN, SQUEEZENET_STD

%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'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/
#       ↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

```

The autoreload extension is already loaded. To reload it, use:

```
%reload_ext autoreload
```

1.0.1 Helper Functions

Our pretrained model was trained on images that had been preprocessed by subtracting the per-color mean and dividing by the per-color standard deviation. We define a few helper functions for performing and undoing this preprocessing in `cs23n/net_visualization_pytorch`. You don't need to do anything here.

```
[57]: from cs231n.net_visualization_pytorch import preprocess, deprocess, rescale, blur_image
```

2 Pretrained Model

For all of our image generation experiments, we will start with a convolutional neural network which was pretrained to perform image classification on ImageNet. We can use any model here, but for the purposes of this assignment we will use SqueezeNet [1], which achieves accuracies comparable to AlexNet but with a significantly reduced parameter count and computational complexity.

Using SqueezeNet rather than AlexNet or VGG or ResNet means that we can easily perform all image generation experiments on CPU.

[1] Iandola et al, “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5MB model size”, arXiv 2016

```
[58]: # Download and load the pretrained SqueezeNet model.
model = torchvision.models.squeezenet1_1(pretrained=True)

# We don't want to train the model, so tell PyTorch not to compute gradients
# with respect to model parameters.
for param in model.parameters():
    param.requires_grad = False

# you may see warning regarding initialization deprecated, that's fine, please
#       ↪continue to next steps
```

2.1 Load some ImageNet images

We have provided a few example images from the validation set of the ImageNet ILSVRC 2012 Classification dataset. To download these images, descend into `cs231n/datasets/` and run `get_imagenet_val.sh`.

Since they come from the validation set, our pretrained model did not see these images during training.

Run the following cell to visualize some of these images, along with their ground-truth labels.

```
[59]: from cs231n.data_utils import load_imagenet_val
X, y, class_names = load_imagenet_val(num=5)

plt.figure(figsize=(12, 6))
for i in range(5):
    plt.subplot(1, 5, i + 1)
    plt.imshow(X[i])
    plt.title(class_names[y[i]])
    plt.axis('off')
plt.gcf().tight_layout()
```



3 Saliency Maps

Using this pretrained model, we will compute class saliency maps as described in Section 3.1 of [2].

A **saliency map** tells us the degree to which each pixel in the image affects the classification score for that image. To compute it, we compute the gradient of the unnormalized score corresponding to the correct class (which is a scalar) with respect to the pixels of the image. If the image has shape $(3, H, W)$ then this gradient will also have shape $(3, H, W)$; for each pixel in the image, this gradient tells us the amount by which the classification score will change if the pixel changes by a small amount. To compute the saliency map, we take the absolute value of this gradient, then take the maximum value over the 3 input channels; the final saliency map thus has shape (H, W) and all entries are nonnegative.

[2] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps”, ICLR Workshop 2014.

3.0.1 Hint: PyTorch gather method

Recall in Assignment 1 you needed to select one element from each row of a matrix; if s is a numpy array of shape (N, C) and y is a numpy array of shape $(N,)$ containing integers $0 \leq y[i] < C$, then $s[\text{np.arange}(N), y]$ is a numpy array of shape $(N,)$ which selects one element from each element in s using the indices in y .

In PyTorch you can perform the same operation using the `gather()` method. If s is a PyTorch Tensor of shape (N, C) and y is a PyTorch Tensor of shape $(N,)$ containing longs in the range $0 \leq y[i] < C$, then

```
s.gather(1, y.view(-1, 1)).squeeze()
```

will be a PyTorch Tensor of shape $(N,)$ containing one entry from each row of s , selected according to the indices in y .

run the following cell to see an example.

You can also read the documentation for [the gather method](#) and [the squeeze method](#).

```
[60]: # Example of using gather to select one entry from each row in PyTorch
def gather_example():
    N, C = 4, 5
    s = torch.randn(N, C)
    y = torch.LongTensor([1, 2, 1, 3])
    print(s)
    print(y)
    print(s.gather(1, y.view(-1, 1)).squeeze())
gather_example()
```

```
tensor([[[-0.5681, -0.1324,  0.5143,  0.3365,  0.8193],
        [-0.8917, -0.2219, -2.0967,  1.3940, -1.0449],
        [-0.7421,  1.3888,  0.8302,  1.6993,  1.3568],
        [ 0.2433, -0.6326, -0.3872, -1.6884, -0.9863]]])
tensor([1, 2, 1, 3])
tensor([-0.1324, -2.0967,  1.3888, -1.6884])
```

Implement `compute_saliency_maps` function inside `cs231n/net_visualization_pytorch.py`

```
[61]: # Load saliency maps computation function
from cs231n.net_visualization_pytorch import compute_saliency_maps
```

Once you have completed the implementation above, run the following to visualize some class saliency maps on our example images from the ImageNet validation set:

```
[62]: def show_saliency_maps(X, y):
    # Convert X and y from numpy arrays to Torch Tensors
    X_tensor = torch.cat([preprocess(Image.fromarray(x)) for x in X], dim=0)
    y_tensor = torch.LongTensor(y)

    # Compute saliency maps for images in X
    saliency = compute_saliency_maps(X_tensor, y_tensor, model)
```

```

# Convert the saliency map from Torch Tensor to numpy array and show images
# and saliency maps together.
saliency = saliency.numpy()
N = X.shape[0]
for i in range(N):
    plt.subplot(2, N, i + 1)
    plt.imshow(X[i])
    plt.axis('off')
    plt.title(class_names[y[i]])
    plt.subplot(2, N, N + i + 1)
    plt.imshow(saliency[i], cmap=plt.cm.hot)
    plt.axis('off')
    plt.gcf().set_size_inches(12, 5)
plt.show()

show_saliency_maps(X, y)

```



4 INLINE QUESTION

A friend of yours suggests that in order to find an image that maximizes the correct score, we can perform gradient ascent on the input image, but instead of the gradient we can actually use the saliency map in each step to update the image. Is this assertion true? Why or why not?

Your Answer: The assertion is false since the saliency map is always positive and contains no information about different channels.

5 Fooling Images

We can also use image gradients to generate “fooling images” as discussed in [3]. Given an image and a target class, we can perform gradient **ascent** over the image to maximize the target class,

stopping when the network classifies the image as the target class. Implement the following function to generate fooling images.

[3] Szegedy et al, “Intriguing properties of neural networks”, ICLR 2014

Implement `make_fooling_image` function inside `cs231n/net_visualization_pytorch.py`

Run the following cell to generate a fooling image. You should ideally see at first glance no major difference between the original and fooling images, and the network should now make an incorrect prediction on the fooling one. However you should see a bit of random noise if you look at the 10x magnified difference between the original and fooling images. Feel free to change the `idx` variable to explore other images.

```
[63]: from cs231n.net_visualization_pytorch import make_fooling_image
idx = 0
target_y = 6

X_tensor = torch.cat([preprocess(Image.fromarray(x)) for x in X], dim=0)
X_fooling = make_fooling_image(X_tensor[idx:idx+1], target_y, model)

scores = model(X_fooling)
assert target_y == scores.data.max(1)[1][0].item(), 'The model is not fooled!'
```

After generating a fooling image, run the following cell to visualize the original image, the fooling image, as well as the difference between them.

```
[64]: X_fooling_np = deprocess(X_fooling.clone())
X_fooling_np = np.asarray(X_fooling_np).astype(np.uint8)

plt.subplot(1, 4, 1)
plt.imshow(X[idx])
plt.title(class_names[y[idx]])
plt.axis('off')

plt.subplot(1, 4, 2)
plt.imshow(X_fooling_np)
plt.title(class_names[target_y])
plt.axis('off')

plt.subplot(1, 4, 3)
X_pre = preprocess(Image.fromarray(X[idx]))
diff = np.asarray(deprocess(X_fooling - X_pre, should_rescale=False))
plt.imshow(diff)
plt.title('Difference')
plt.axis('off')

plt.subplot(1, 4, 4)
diff = np.asarray(deprocess(10 * (X_fooling - X_pre), should_rescale=False))
plt.imshow(diff)
```

```

plt.title('Magnified difference (10x)')
plt.axis('off')

plt.gcf().set_size_inches(12, 5)
plt.show()

```



6 Class visualization

By starting with a random noise image and performing gradient ascent on a target class, we can generate an image that the network will recognize as the target class. This idea was first presented in [2]; [3] extended this idea by suggesting several regularization techniques that can improve the quality of the generated image.

Concretely, let I be an image and let y be a target class. Let $s_y(I)$ be the score that a convolutional network assigns to the image I for class y ; note that these are raw unnormalized scores, not class probabilities. We wish to generate an image I^* that achieves a high score for the class y by solving the problem

$$I^* = \arg \max_I (s_y(I) - R(I))$$

where R is a (possibly implicit) regularizer (note the sign of $R(I)$ in the argmax: we want to minimize this regularization term). We can solve this optimization problem using gradient ascent, computing gradients with respect to the generated image. We will use (explicit) L2 regularization of the form

$$R(I) = \lambda \|I\|_2^2$$

and implicit regularization as suggested by [3] by periodically blurring the generated image. We can solve this problem using gradient ascent on the generated image.

[2] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps”, ICLR Workshop 2014.

[3] Yosinski et al, “Understanding Neural Networks Through Deep Visualization”, ICML 2015 Deep Learning Workshop

In `cs231n/net_visualization_pytorch.py` complete the implementation of the `image_visualization_update_step` used in the `create_class_visualization` function below. Once you have completed that implementation, run the following cells to generate an image of a Tarantula:

```
[65]: from cs231n.net_visualization_pytorch import class_visualization_update_step, ▾  
      jitter, blur_image  
def create_class_visualization(target_y, model, dtype, **kwargs):  
    """  
        Generate an image to maximize the score of target_y under a pretrained  
        ↵model.  
  
    Inputs:  
    - target_y: Integer in the range [0, 1000) giving the index of the class  
    - model: A pretrained CNN that will be used to generate the image  
    - dtype: Torch datatype to use for computations  
  
    Keyword arguments:  
    - l2_reg: Strength of L2 regularization on the image  
    - learning_rate: How big of a step to take  
    - num_iterations: How many iterations to use  
    - blur_every: How often to blur the image as an implicit regularizer  
    - max_jitter: How much to jitter the image as an implicit regularizer  
    - show_every: How often to show the intermediate result  
    """  
    model.type(dtype)  
    l2_reg = kwargs.pop('l2_reg', 1e-3)  
    learning_rate = kwargs.pop('learning_rate', 25)  
    num_iterations = kwargs.pop('num_iterations', 100)  
    blur_every = kwargs.pop('blur_every', 10)  
    max_jitter = kwargs.pop('max_jitter', 16)  
    show_every = kwargs.pop('show_every', 25)  
  
    # Randomly initialize the image as a PyTorch Tensor, and make it requires  
    ↵gradient.  
    img = torch.randn(1, 3, 224, 224).mul_(1.0).type(dtype).requires_grad_()  
  
    for t in range(num_iterations):  
        # Randomly jitter the image a bit; this gives slightly nicer results  
        ox, oy = random.randint(0, max_jitter), random.randint(0, max_jitter)  
        img.data.copy_(jitter(img.data, ox, oy))  
        class_visualization_update_step(img, model, target_y, l2_reg, ▾  
                                       ↵learning_rate)  
        # Undo the random jitter  
        img.data.copy_(jitter(img.data, -ox, -oy))  
  
        # As regularizer, clamp and periodically blur the image
```

```

for c in range(3):
    lo = float(-SQUEEZENET_MEAN[c] / SQUEEZENET_STD[c])
    hi = float((1.0 - SQUEEZENET_MEAN[c]) / SQUEEZENET_STD[c])
    img.data[:, c].clamp_(min=lo, max=hi)
if t % blur_every == 0:
    blur_image(img.data, sigma=0.5)

# Periodically show the image
if t == 0 or (t + 1) % show_every == 0 or t == num_iterations - 1:
    plt.imshow(deprocess(img.data.clone().cpu()))
    class_name = class_names[target_y]
    plt.title('%s\nIteration %d / %d' % (class_name, t + 1, num_iterations))
    plt.gcf().set_size_inches(4, 4)
    plt.axis('off')
    plt.show()

return deprocess(img.data.cpu())

```

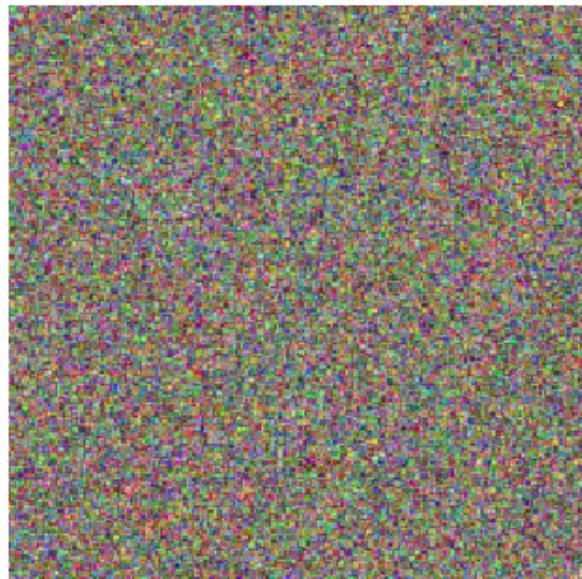
```

[66]: dtype = torch.FloatTensor
# dtype = torch.cuda.FloatTensor # Uncomment this to use GPU
model.type(dtype)

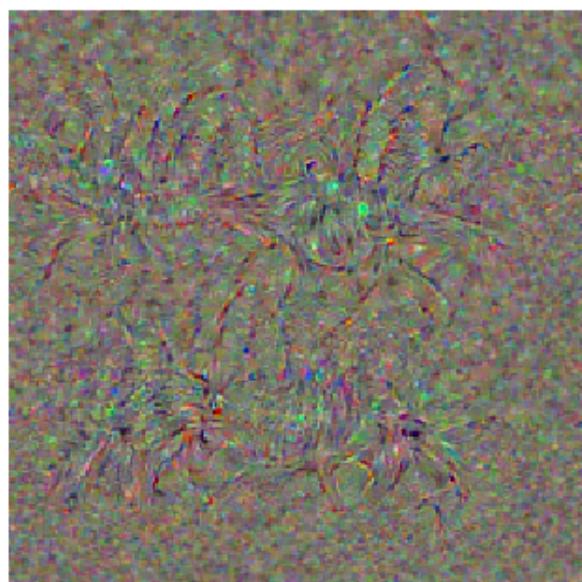
target_y = 76 # Tarantula
# target_y = 78 # Tick
# target_y = 187 # Yorkshire Terrier
# target_y = 683 # Oboe
# target_y = 366 # Gorilla
# target_y = 604 # Hourglass
out = create_class_visualization(target_y, model, dtype)

```

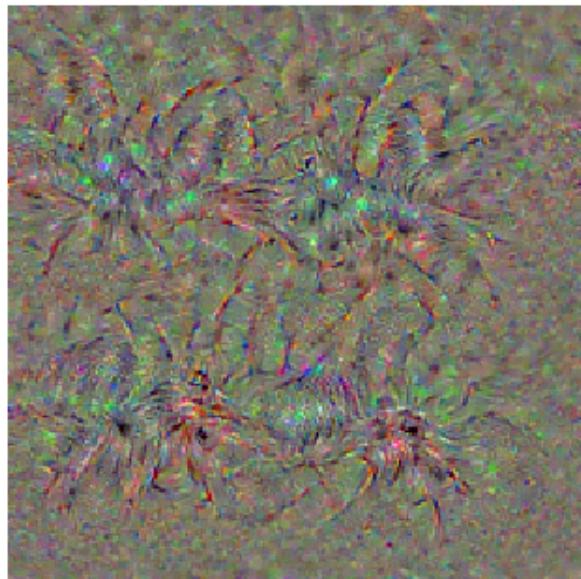
tarantula
Iteration 1 / 100



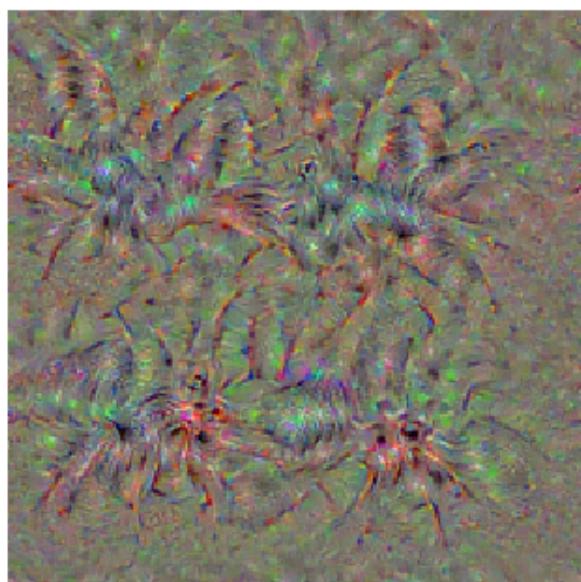
tarantula
Iteration 25 / 100

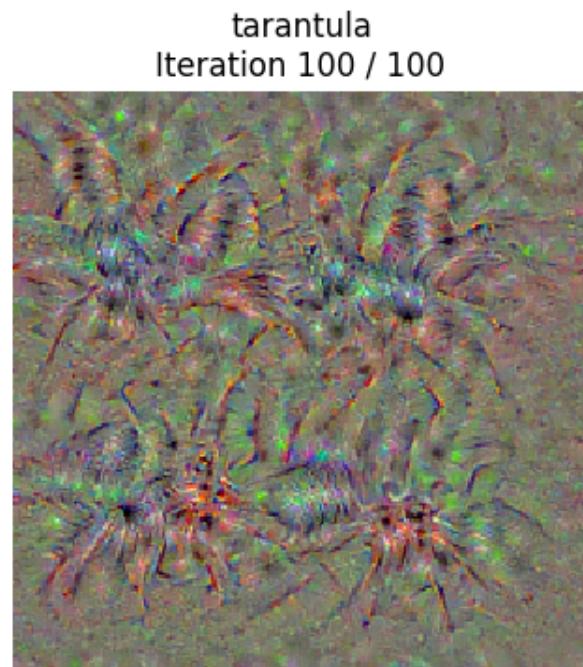


tarantula
Iteration 50 / 100



tarantula
Iteration 75 / 100



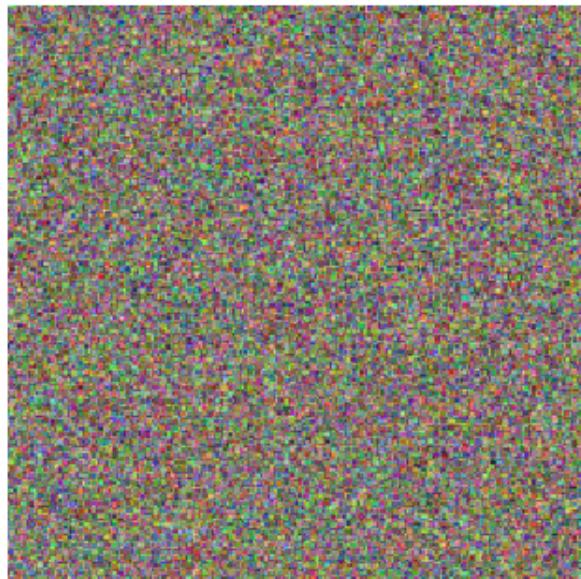


Try out your class visualization on other classes! You should also feel free to play with various hyperparameters to try and improve the quality of the generated image, but this is not required.

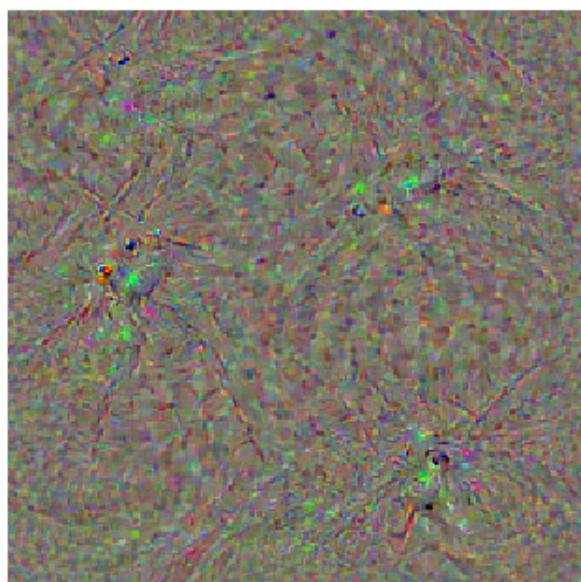
```
[69]: # target_y = 78 # Tick
# target_y = 187 # Yorkshire Terrier
# target_y = 683 # Oboe
# target_y = 366 # Gorilla
# target_y = 604 # Hourglass
target_y = np.random.randint(1000)
print(class_names[target_y])
X = create_class_visualization(target_y, model, dtype)
```

lacewing, lacewing fly

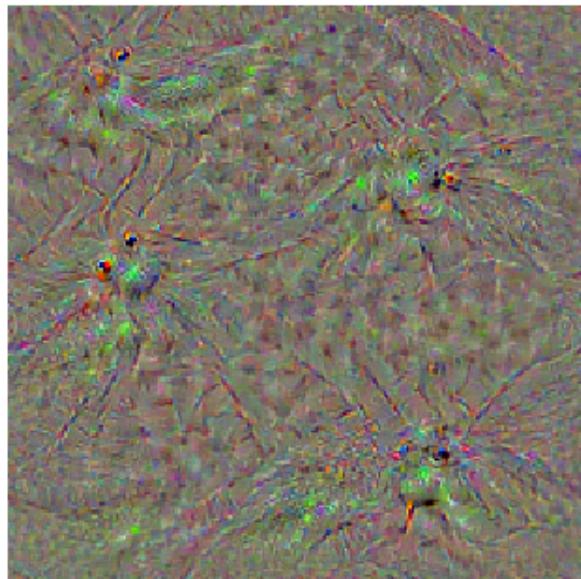
lacewing, lacewing fly
Iteration 1 / 100



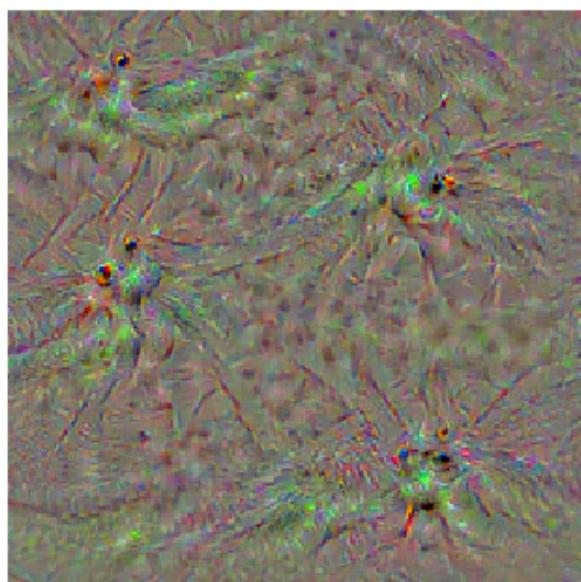
lacewing, lacewing fly
Iteration 25 / 100



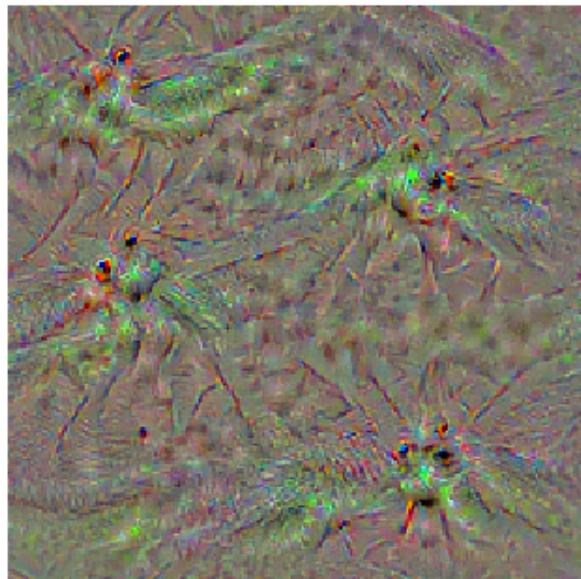
lacewing, lacewing fly
Iteration 50 / 100



lacewing, lacewing fly
Iteration 75 / 100



lacewing, lacewing fly
Iteration 100 / 100



[67] :