TP description d'image avec un RNN

Au cours de ce TP, vous allez implémenter un RNN dans le but d'obtenir un système de description d'image, c'est-à-dire un système capable de générer une phrase décrivant une image.

1. Lancer une session linux (et non pas windows)
2. Aller dans "Applications", puis "Autre", puis "conda_pytorch" (un terminal devrait s'ouvrir)
3. Dans ce terminal, taper la commande suivante pour lancer Spyder : `spyder &`
4. Configurer Spyder en suivant ces instructions : Lien configuration Spyder.
5. Créer un dossier `TP_description_image_RNN` et placer le dossier `utils` à l'intérieur.
6. Créer un script python `tp.py` dans le dossier `TP_description_image_RNN` et coller les lignes de code suivantes :

```
import time, os, json
import numpy as np
import matplotlib.pyplot as plt
from utils.coco_utils import load_coco_data, sample_coco_minibatch, decode_captions
from utils.image_utils import image_from_url
from utils.rnn_layers import *
from utils.gradient_check import eval_numerical_gradient_array, eval_numerical_gradient
from utils.rnn import CaptioningRNN
from utils.captioning_solver import *



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

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))))
```

Base de données Microsoft COCO

Nous utiliserons la base de données Microsoft COCO qui est une référence dans le domaine de la description d'une image. La base de données contient 82783 images d'entraînement et 40504 de validation, chacune étiquetée par environ 5 phrases différentes.

Attention : la base de données est d'environ ~256Mo. Il faudra donc la télécharger et la décompresser dans `/tmp`. La base de données est disponible sur Thor, dans l'onglet "Documents".

Les données ont été pré-traitées. Pour chaque image, un vecteur (de taille 4096) issu de la couche fc7 d'un VGG-16 (pré-entraîné sur ImageNet) a été extrait, puis réduit à une taille de 512 (avec une ACP) avant d'être stocké dans les fichiers `train2014_vgg16_fc7_pca.h5` et `val2014_vgg16_fc7_pca.h5`.

Les images de la base de données ne sont pas présentes dans l'archive (sinon l'archive ferait ~20Go) mais les URLs sont stockées dans les fichiers `train2014_urls.txt` et `val2014_urls.txt`. Les images peuvent donc être téléchargées si besoin.

Nous ne travaillerons pas directement avec des mots mais plutôt avec des entiers. Un entier a été attribué à chaque mot (fichier `coco2014_vocab.json`). La fonction `decode_captions` permet de convertir un tableau numpy d'entiers en mots.

Un certain nombre de jetons ont été ajoutés : `<START>`, `<END>` (pour le début et la fin d'une phrase) ainsi que `<UNK>` (pour "unknown") pour les mots rares. De plus, afin de réaliser des entraînements avec des minibatches dont les phrases sont initialement de tailles différentes, un jeton `<NULL>` est défini pour compléter les phrases

jusqu'à une longueur prédéfinie. **Toute cette partie du code est déjà réalisée afin que le TP se focalise sur l'implémentation du RNN.**

Charger la base de données à l'aide des lignes de code suivantes :

```
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))
```

Visualiser les données

De manière générale, il est indispensable de regarder les données que nous nous apprêtons à traiter.

Vous pouvez visualiser un minibatch de la manière suivante :

```
# 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)):
    im_cur = image_from_url(url)
    if(type(im_cur)!=type(None)):# if the image was downloaded
        plt.imshow(im_cur)
        plt.axis('off')
        caption_str = decode_captions(caption, data['idx_to_word'])
        plt.title(caption_str)
        plt.show()
```

Réseau de neurones récurrent

Nous utiliserons un réseau de neurones récurrent (RNN) pour réaliser la description d'image. Le fichier `utils/rnn_layers.py` contient les implémentations de différentes couches permettant d'obtenir un RNN. Le fichier `utils/rnn.py` utilise les couches définies dans le fichier précedent pour implémenter le modèle de description d'image.

Nous débuterons avec l'implémentation des couches dans le fichier `utils/rnn_layers.py`.

RNN : propagation avant sur un pas de temps

Ouvrir le fichier `utils/rnn_layers.py`. Ce fichier contient les implémentations des étapes de propagation avant (forward) et de rétropropagation du gradient (backward) pour différentes couches.

À Coder : coder la fonction `rnn_step_forward` qui implémente la propagation avant d'un seul pas de temps. Vous pouvez tester votre code en exécutant le morceau de code suivant (vous devriez obtenir une erreur très faible, de l'ordre de e-8) :

```
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))
```

RNN : rétropropagation du gradient sur un pas de temps

À Coder : coder la fonction `rnn_step_backward` qui implémente la fonction de rétropropagation du gradient sur un seul pas de temps.
Aide : Vous trouverez l'expression de la dérivée de la fonction tangente hyperbolique sur wikipedia. La rétropropagation du gradient au travers d'une transformation affine a déjà été réalisée lors du TP sur le MLP, elle est également rappelée dans le document `derivative_fully_connected.pdf` présent dans l'archive de ce TP.
Vous pouvez tester votre code en exécutant le morceau de code suivant (vous devriez obtenir une erreur très faible, de l'ordre de e-8) :

```
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))
```

RNN : propagation avant

À ce stade, la propagation avant et la rétropropagation du gradient ont été implémentées sur un seul pas de temps. Vous allez maintenant utiliser ces fonctions pour implémenter un RNN capable de traiter une séquence de données.

À coder : Toujours dans le fichier `utils/rnn_layers.py`, implementer la fonction `rnn_forward` (en faisant appel à la fonction `rnn_step_forward`). Vous pouvez tester votre code en exécutant le morceau de code suivant (vous devriez obtenir une erreur très faible, de l'ordre de e-7) :

```
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))
```

RNN : rétropropagation du gradient

À coder : coder la fonction `rnn_backward` qui implémente la rétropropagation du gradient sur la totalité de la séquence de données. Pour ce faire, utiliser la fonction `rnn_step_backward`. Vous pouvez tester votre code en exécutant le morceau de code suivant (vous devriez obtenir une erreur très faible, de l'ordre de e-6) :

```
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))
```

Encodage d'un mot en un vecteur ("word embedding") : propagation avant

Comme nous l'avons vu précédemment, un mot est représenté par un entier. Il est classique de transformer cet entier en un vecteur de variables réelles (de taille prédéfinies par l'utilisateur). En anglais cette étape d'encodage d'un mot s'appelle "word embedding".

À coder : Implémenter la fonction `word_embedding_forward` qui permet d'encoder des mots (représentés par des entiers) en vecteurs. Vous pouvez tester votre code en exécutant le morceau de code suivant (vous devriez obtenir une erreur très faible, de l'ordre de e-8) :

```
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))
```

Encodage d'un mot en un vecteur ("word embedding") : rétropropagation du gradient

À coder : Implémenter la fonction `word_embedding_backward`. Vous pouvez tester votre code en exécutant le morceau de code suivant (vous devriez obtenir une erreur très faible, de l'ordre de e-11) :

```
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))
```

Transformation affine de sortie

À chaque pas de temps, une transformation affine ("fully connected") est appliquée au vecteur "état caché" du RNN pour obtenir des scores pour chaque mot du vocabulaire. Cette fonction est déjà implémentée (voir `temporal_affine_forward` et `temporal_affine_backward`). Vous pouvez tester cette implémentation en exécutant le morceau de code suivant (vous devriez obtenir une erreur très faible, de l'ordre de e-9) :

```
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))
```

Fonction de coût "cross-entropy" temporelle

Dans un RNN produisant une séquence de mots, à chaque pas de temps un score est produit pour chaque mot du vocabulaire. Puisque que la vérité terrain nous est fournie (nous savons quel mot du vocabulaire devrait avoir un score très élevé, alors que tous les autres devraient avoir un score très faible), nous utilisons à chaque pas de temps une fonction de coût "cross-entropy".

Il faut néanmoins faire attention au fait que des jetons `<NULL>` ont été ajoutés à la fin des phrases pour

qu'elles aient toutes la même longueur. Ces jetons ne doivent pas être pris en compte dans la fonction de coût, ainsi la fonction de coût prend également en entrée un `mask` indiquant les éléments à prendre en compte et ceux qu'il convient d'ignorer.

Cette fonction est déjà implémentée (voir `temporal_softmax_loss` dans `rnn_layers`). Vous pouvez tester cette implémentation en exécutant le morceau de code suivant (vous devriez obtenir une erreur très faible, de l'ordre de e-7).

```
# Sanity check for temporal softmax loss
from utils.rnn_layers import temporal_softmax_loss

N, T, V = 100, 1, 10

def check_loss(N, T, V, p):
    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 about 2.3

# 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))
```

RNN et description d'image

Les couches nécessaires à l'implémentation d'un RNN sont désormais prêtent. Ouvrir le fichier `utils/rnn.py` et lire la classe `CaptioningRNN`.

Dans la fonction `loss`, implémenter la propagation avant ainsi que la rétropropagation du gradient, puis tester votre code (vous devriez constater une erreur d'environ e-10) :

```
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))
```

Utiliser le code suivant pour vérifier l'implémentation de la rétropropagation du gradient (vous devriez obtenir une erreur de e-6 ou moins).

```
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], verbose=False, h=1e-6)
    e = rel_error(param_grad_num, grads[param_name])
    print('%s relative error: %e' % (param_name, e))
```

Lancer un apprentissage sur une petite base de données

Pour lancer un apprentissage du RNN, nous utiliserons la classe `CaptioningSolver` du fichier `utils/captioning_solver.py`. Prendre le temps de lire cette classe, puis exécuter le code suivant pour lancer un apprentissage (on devrait plutôt parler de sur-apprentissage) sur une base de données de 50 couples image/phrase. Le coût final devrait être inférieur à 0.1.

```
np.random.seed(231)

small_data = load_coco_data(max_train=50)

small_rnn_model = CaptioningRNN(
          cell_type='rnn',
          word_to_idx=small_data['word_to_idx'],
          input_dim=small_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()
```

Génération d'une description d'une image ("Test-time")

Comme nous l'avons vu en cours, un modèle de description d'image reposant sur un RNN fonctionne différemment lors de l'étape d'apprentissage par rapport à l'étape d'inférence. En effet, lors de l'étape d'apprentissage, les mots "vérité terrain" sont passés en entrée du RNN à chaque pas de temps. En revanche, lors de l'étape d'inférence, un mot passé en entrée du RNN à un instant t correspond à la prédiction (ou à un échantillonnage) du RNN à l'instant précédent $(t-1)$.

Dans le fichier `utils/rnn.py`, implementer la fonction `sample` qui permet de réaliser une inférence. Ensuite, tester la fonction en générant des descriptions (en utilisant le modèle précedemment sur-entraîné) sur la base d'apprentissage ainsi que sur la base de validation. Vous devriez constater que les descriptions obtenues pour les exemples de la base d'apprentissage sont très satisfaisants (car le RNN a sur-appris sur cette base). En revanche, les résultats obtenus pour les exemples de la base de validation n'auront probablement aucun sens.

```
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.figure()
    plt.imshow(image_from_url(url))
    plt.title('%s\n%s\nGT:%s' % (split, sample_caption, gt_caption))
    plt.axis('off')
    plt.show()
```

QUESTION

Le précédent modèle de description d'image produit un mot à chaque pas de temps. Une autre manière de faire aurait consisté à générer un caractère à chaque pas de temps. Par exemple, au lieu de générer ['un','chat','noir'], le réseau générerait ['u','n',' ','c','h','a','t',' ','n','o','i','r'].

Un modèle basé mot présente des avantages et des inconvénients par rapport à un modèle basé caractère. Donner un avantage et un inconvénient. Indice : Le nombre de classes à chaque pas de temps est-il le même dans les deux cas ? Comment gérer l'orthographe ?

Source: http://cs231n.github.io/assignments2019/assignment3/