Deep learning / apprentissage profond
Pratique Tensorflow basique

Hervé Le Borgne – Youssef Tamaazousti

2017-2018

Contenu

- Préalable: installer Tensorflow
- Pratique: jeu « Fizz Buzz »... En apprentissage

Fizz Buzz

Afficher les chiffres de 1 à 100, en remplaçant:

- les multiples de 3 par *fizz*
- les multiples de 5 par *buzz*
- les multiples de 3 et 5 par *fizzbuzz*

- Trivial à réaliser en procédural (1 FOR + 2 IF)
- Peut être fait par apprentissage (!)

Rappel: installation Tensorflow

Exemple : installer avec méthode `virtualenv` sous Kubuntu 16.04:

```
sudo apt-get install python-pip python-dev python-virtualenv
virtualenv --system-site-packages ~/Applis/tensorflow
source ~/Applis/tensorflow/bin/activate
# CPU only
pip install --upgrade https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-1.6.0-cp27-none-linux_x86_64.whl
```

- Très facile à mettre en œuvre (... sauf installation CUDA et CuDNN...)
- Très peu intrusive sur le système:
  - `python-pip` : gestion de package
  - `python-dev` : pour utiliser des librairies python en C (*headers* API C)
  - `python-virtualenv` : installations de Python isolées de l'OS
- Utilisation:

```
# activer l'environnement
source ~/Applis/tensorflow/bin/activate

# lancer python
ipython

# faire sa cuisine en python...
  import tensorflow as tf
  import numpy as np

  ...

# pour quitter l'environnement tensorflow
deactivate
```

Pour utiliser IPython du virtualEnv:

```
alias ipy="python -c 'import IPython;
IPython.terminal.ipapp.launch_new_instance()'"
```

Approche procédurale

Algorithm 1 Fizz Buzz

```
i ← 1
while i ≤ 100 do
    if i%3 == 0 then
        if i%5 == 0 then
            print "fizzbuzz"
        else
            print "fizz"
        end if
    else if i%5 == 0 then
        print "buzz"
    else
        print i
    end if
end while
```

Approche procédurale: octave/matlab

```
1  for i=1:100
2  if mod(i,3)==0
3    if mod(i,5)==0
4    printf('fizzbuzz ')
5    else
6    printf('fizz ',i)
7    endif
8  elseif mod(i,5)==0
9      printf('buzz ',i)
10 else
11   printf('%d ',i)
12 endif
13 end
14 printf('\n')
```

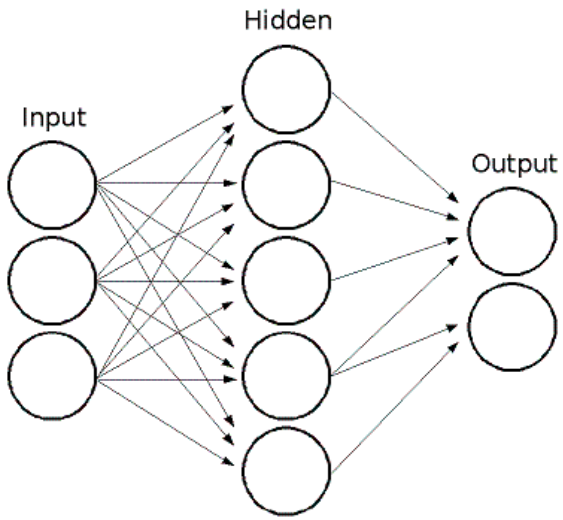
```
herve@IS225554:~/Documents/enseignement/2017/ECP/exercices/fizzbuzz$ octave
>> fizzbuzz
1 2 fizz 4 buzz fizz 7 8 fizz buzz 11 fizz 13 14 fizzbuzz 16 17 fizz 19 buzz fizz 22 23 fizz buzz 26 fizz 28 29 fizzbuzz
 31 32 fizz 34 buzz fizz 37 38 fizz buzz 41 fizz 43 44 fizzbuzz 46 47 fizz 49 buzz fizz 52 53 fizz buzz 56 fizz 58 59 fi
zzbuzz 61 62 fizz 64 buzz fizz 67 68 fizz buzz 71 fizz 73 74 fizzbuzz 76 77 fizz 79 buzz fizz 82 83 fizz buzz 86 fizz 88
 89 fizzbuzz 91 92 fizz 94 buzz fizz 97 98 fizz buzz
```

Peut-on *apprendre* le Fizz Buzz ?

Source (liens): Billet de Joël Grus et code sur Github

Principe:

- On apprend sur $[101 - 1000]$ et on teste sur $[1 - 100]$
  - approche procédurale pour générer l'ensemble d'apprentissage!
- On utilise un réseau de neurones pour modéliser le problème
  - MLP à une couche cachée



- Quelles entrées / sorties ? (vectorielles...)

Modélisation entrées sorties

- Sorties
  - probabilité de chaque (4) réponse possible
    $P(''fizz''|N)P(''buzz''|N)P(''fizzbuzz''|N)P(''N''|N)$
  - le choix sera la probabilité maximale
- Entrées
  - doit être une représentation vectorielle qui contienne suffisamment d'information pour refléter les multiples/diviseurs
  - écriture décimale ? $\ldots a_3 a_2 a_2 a_0 = \sum a_i 10^i$
  - ordinateur $\rightarrow$ écriture binaire!

Code

- initialisation `numpy` et `tensorflow`
- codage des vecteur d'entrée
- approche procédurale pour vérité terrain

```
import numpy as np
import tensorflow as tf

NUM_DIGITS = 10

# codage binaire d'un chiffre (max NUM_DIGITS bits)
def binary_encode(i, num_digits):
    return np.array([i >> d & 1 for d in range(num_digits)])

# création vérité terrain: [number, "fizz", "buzz", "fizzbuzz"]
def fizz_buzz_encode(i):
    if   i % 15 == 0: return np.array([0, 0, 0, 1])
    elif i % 5  == 0: return np.array([0, 0, 1, 0])
    elif i % 3  == 0: return np.array([0, 1, 0, 0])
    else:             return np.array([1, 0, 0, 0])

# données d'entraînement (X) et labels (Y)
trX = np.array([binary_encode(i, NUM_DIGITS) for i in range(101, 2 ** NUM_DIGITS)])
trY = np.array([fizz_buzz_encode(i)          for i in range(101, 2 ** NUM_DIGITS)])
```

Code

- définition du MLP à 1 couche cachée
- définition des variables entrée/sortie (`placeholder`)
- initialisation des paramètres (poids)
- définition de la fonction de prédiction

```
# définition du MLP à 1 couche cachée (non linéaité ReLU)
# la fonction de coût (sortie finale) est définie séparément
def model(X, w_h, w_o):
    h = tf.nn.relu(tf.matmul(X, w_h))
    return tf.matmul(h, w_o)

# Variables d'entrée et de sortie du réseau
X = tf.placeholder("float", [None, NUM_DIGITS])
Y = tf.placeholder("float", [None, 4])

# How many units in the hidden layer.
NUM_HIDDEN = 100

# initialisation aléatoire des paramètres (gaussienne)
def init_weights(shape):
    return tf.Variable(tf.random_normal(shape, stddev=0.01))
w_h = init_weights([NUM_DIGITS, NUM_HIDDEN])
w_o = init_weights([NUM_HIDDEN, 4])

# fonction de prédiction (estimation de la sortie du réseau)
py_x = model(X, w_h, w_o)
```

Code

- Comment apprendre ?
  - couche de sortie et fonction de coût
  - méthode d'optimisation
- conversion estimation $\rightarrow$ affichage attendu

```
# Definition de l'apprentissage:
#    - fonction de cout (cross entropie sur softmax)
#    - methode de minimisation (descente de gradient)
# WARNING en python 3, il faut préciser (logits=py_x,labels=Y)
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=py_x, labels=Y))
train_op = tf.train.GradientDescentOptimizer(0.05).minimize(cost)

# prediction = plus grande (proba de) sortie
predict_op = tf.argmax(py_x, 1)

# affichage attendu par l'application
def fizz_buzz(i, prediction):
    return [str(i), "fizz", "buzz", "fizzbuzz"][prediction]
```

Code

- Calcul dans une session

```
# on lance les calculs dans une "session"
BATCH_SIZE = 128 # taille minibatch
with tf.Session() as sess:
    tf.initialize_all_variables().run()

    for epoch in range(10000):
        # mélange des données à chaque 'epoch' (~itération d'apprentissage)
        p = np.random.permutation(range(len(trX)))
        trX, trY = trX[p], trY[p]

        # Apprentissage avec des minibatches de taille 128
        for start in range(0, len(trX), BATCH_SIZE):
            end = start + BATCH_SIZE
            sess.run(train_op, feed_dict={X: trX[start:end], Y: trY[start:end]})

        # affichage de la performance courante (1-erreur empirique)
        print(epoch, np.mean(np.argmax(trY, axis=1) ==
                             sess.run(predict_op, feed_dict={X: trX, Y: trY})))

    # Affichage sur les données de test
    numbers = np.arange(1, 101)
    teX = np.transpose(binary_encode(numbers, NUM_DIGITS))
    teY = sess.run(predict_op, feed_dict={X: teX})
    output = np.vectorize(fizz_buzz)(numbers, teY)

    print(output)
```

Résultat

- erreur empirique nulle
- erreur de test non nulle

Exercice 1

1. afficher la performance finale
2. implémenter un meilleur estimateur que $1 - l_{emp}$ pendant l'apprentissage

Exercice 2

Étudiez l'influence des éléments suivants:

- taille de la couche cachée
- taille de l'ensemble d'apprentissage
- valeur du pas d'apprentissage

Bonus: version PyTorch

- On fournit une implémentation pyTorch
  - Code plus lisible (selon les goûts)
  - Pas de contrôle de l'initialisation (mais c'est faisable)
  - Un peu moins rapide (selon configuration)
- Nombreux exemples sur le site

```
# définition du MLP à 1 couche cachée (non linearite ReLU)
model = torch.nn.Sequential(
    torch.nn.Linear(NUM_DIGITS, NUM_HIDDEN),
    torch.nn.ReLU(),
    torch.nn.Linear(NUM_HIDDEN, 4)
    )

# fonction de coût
loss_fn = torch.nn.CrossEntropyLoss()

# optimiseur
optimizer = torch.optim.SGD(model.parameters(), lr = 0.05)

# affichage attendu par l'application
def fizz_buzz(i, prediction):
    return [str(i), "fizz", "buzz", "fizzbuzz"][prediction]

# on lance les calculs
BATCH_SIZE = 128
for epoch in range(10000):
    for start in range(0, len(X), BATCH_SIZE):
        end = start + BATCH_SIZE
        batchX = X[start:end]
        batchY = Y[start:end]

        # prediction et calcul loss
        y_pred = model(batchX)
        loss = loss_fn(y_pred, batchY)

        # mettre les gradients à 0 avant la passe retour (ba
        optimizer.zero_grad()

        # rétro-propagation
        loss.backward()
        optimizer.step()

    # calcul coût  (et affichage)
    loss = loss_fn( model(X), Y)
    if epoch%100 == 0:
        print(epoch, loss.data[0])
```