Deep Learning for 3D Shape Retreival

Vivien Kraus, Tianning Yu, École Centrale de Lyon

16 février 2017

Plan

Plan

Généralités

- Tensorflow : un outil de *deep learning* pour le python ;
- « Open Source » : peut nécessiter CUDA ;
- initialement développé par Google ;
- installation par distribution, `pip`, ou `docker`.

Calcul matriciel

```
1  import tensorflow as tf
2  from functools import reduce
3  from operator import mul
4
5  with tf.Session (): #
6      vector_a = tf.constant ([1, 1, 1]) #
7      vector_b = tf.constant ([2, 2, 2]) #
8      vector_sum = tf.add (vector_a, vector_b) #
9      list_result = vector_sum.eval () #
10
11 _ = list_result.tolist ()
```

Listing 1: Sémantique de calcul par Tensorflow

Calcul matriciel : discussion

Résultat

3 3 3

Discussion

- Ligne 5 : tous les calculs se font à l'intérieur d'une `Session` Tensorflow ;
- Ligne 6, 7 : on définit des variables Python comme des constantes Tensorflow : ce sont des objets, paramètres constants ;
- Ligne 8 : on construit la somme, le calcul n'est pas encore fait ;
- Ligne 9 : effectue le calcul.

Calcul matriciel : variables

- Lorsqu'on veut effectuer des opérations de façon répétée, on stocke le résultat dans des variables ;
- on définit les opérations à effectuer et on laisse Tensorflow calculer ;
- exemple : 2

Calcul matriciel : variables (code)

```
12  with tf.Session () as session:
13    # Declare variables and operations
14    mat_sum = tf.Variable ([[0., 0., 0.], [0., 0., 0.]])
15    apply_op = tf.assign (mat_sum, \
16      tf.add (mat_sum, tf.random_uniform ([2, 3], 0, 1)))
17    # Initialize variables
18    tf.global_variables_initializer ().run ()
19    # Apply operation 100 times
20    for _ in range (100):
21      session.run (apply_op)
22    _ = mat_sum.eval ().tolist ()
```

Listing 2: Sémantique de calcul par Tensorflow : les variables

Calcul matriciel : variables (discussion)

Résultat

	x	y	z	moy
L1	48.13	50.52	50.04	49.56
L2	48.80	54.40	50.15	51.12
moy	48.47	52.46	50.09	50.34

Discussion (1)

- On a bien ajouté 100 fois un nombre uniformément pris entre 0 et 1 ;

Calcul matriciel : variables (discussion)

Discussion (2)

- on déclare la variable dans le bloc ligne 13, en indiquant bien qu'il s'agit de flottants ;
- les initialisations à 0 sont effectuées ligne 17 ;
- on déclare les opérations aussi dans le bloc ligne 13 : on ajoute à `mat_sum` une perturbation de moyenne $\frac{1}{2}$;
- on applique 100 fois l'opération dans le bloc ligne 19 ;
- on récupère enfin le résultat sous forme de vecteur NumPy.

Plan

Schéma du neurone seul

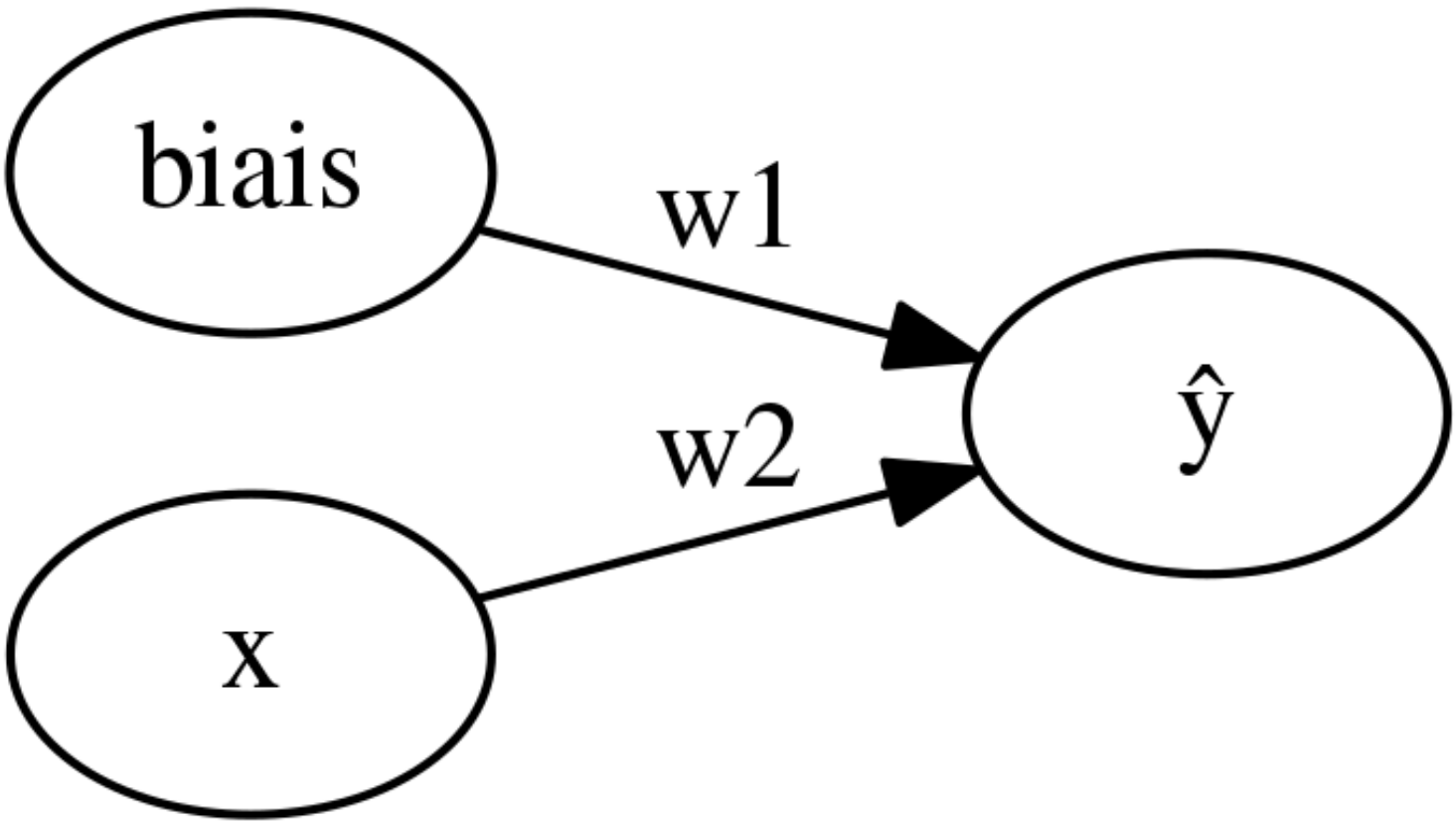

Figure 1: Un neurone tout seul

Démarche

- On souhaite entraîner le neurone ;
- on voudrait obtenir

$$y = 1 + 0.493x$$

- on fabrique artificiellement des données bruitées, et on compare la sortie du neurone à la valeur réelle

$$y = 1 + 0.493x$$

- on minimise la différence entre prédiction et valeur réelle en optimisant les paramètres w_1 et w_2.

Création de données

```
23  import numpy as np
24  from numpy import linalg as la
25  real_w1 = 1
26  real_w2 = 49.3 / 100
27  noise = 0.5
28  n_train = 100
29  y_noise = noise * np.random.randn (n_train, 1);
30  real_error = la.norm (y_noise)
31  grid = np.transpose ([np.linspace (0, 1, n_train)])
32  x = grid + noise * np.random.randn (n_train, 1)
33  y = real_w1 + real_w2 * x + y_noise
34  X = np.concatenate ((np.ones ((n_train, 1)), x), axis=1)
```

Création de données (discussion)

- On utilise uniquement `numpy` pour composer les données ;
- on utilisera 100 données d'apprentissage ;
- on fabrique les données à partir d'un modèle que l'on connaît, pour vérifier la correction de l'apprentissage ;
- il faut tenir compte du biais dans la matrice d'entrée, d'où la distinction entre `x` et `X`.

Création des données d'apprentissage

```
35  n_steps = 500
36  loss_amount = np.zeros ((n_steps))
37  learning_rate = 0.0002
```

- On va mesurer l'évolution de l'erreur (`loss_amount`) en suivant les étapes d'apprentissage ;
- on effectue `n_steps` étapes d'apprentissage ;
- le `learning_rate` est le taux d'apprentissage du neurone, qu'on a spécialement choisi assez petit pour éviter que le calcul de la descente du gradient ne diverge.

Définition des tenseurs

```
38  def define_tensors (X, y):
39      inputs = tf.constant (X)
40      expectations = tf.constant (y)
41      params_32 = tf.Variable (tf.random_normal \
42       ([2, 1], 0, 1.))
43      params = tf.cast (params_32, tf.float64) #
44      prediction = tf.matmul (inputs, params)
45      error = tf.subtract (prediction, expectations)
46      loss = tf.nn.l2_loss (error) # scalar
47      return (params, loss)
```

Définition des tenseurs (discussion)

- Cette fonction doit être appelée dans un contexte TensorFlow disposant d'une Session ;
- on dispose des données d'entrée (`inputs`) et des sorties attendues (`expectations`), et on calcule la prédiction (`prediction`) ;
- le but est d'obtenir une valeur de `loss`, qui est la norme de l'erreur sur tous les échantillons (cf ligne 46), on cherche à minimiser cette norme dans le contexte des moindres carrés ;
- la constante `tf.random_normal (...)` est de type `float32` et non pas `float64`, d'où le cast explicite ligne 43 ;
- on n'a pas besoin de retourner les nœuds de calcul intermédiaires.

Définition des calculs (code)

```
48  def define_computations (learning_rate, \
49   loss_criterion):
50    descend = tf.train \
51        .GradientDescentOptimizer (learning_rate) \
52        .minimize (loss_criterion)
53    return (descend)
```

Définition des calculs (discussion)

- Tout le travail de Tensorflow se situe ici : le calcul de la descente du gradient est masqué, mais on suppose qu'il s'appuie sur la définition de `loss_criterion` (qui sera `loss`, i.e. une norme d'une différence entre produit de constantes et une constante), ce qui n'est pas très compliqué à faire à la main (pour l'instant) ;
- il faut toujours appeler cette fonction dans le contexte d'une session.

Calcul

```
54  with tf.Session () as session:
55    params, loss = define_tensors (X, y)
56    descend = define_computations (learning_rate, loss)
57    tf.global_variables_initializer ().run ()
58    for i in range (n_steps):
59      descend.run ()
60      loss_amount[i] = loss.eval ()
61    learned_weights = params.eval ()
```

Calcul (discussion)

- On constate que si le `learning rate` est trop élevé (cf slide 16), on obtient des valeurs « nan », c'est dû au fait que la correction est de plus en plus agressive ;
- le code a toujours la même forme : définitions des calculs, application des calculs, évaluation des calculs.

Affichage (scatter plot)

```
62  import matplotlib.pyplot as plt
63  fig, (ax1, ax2) = plt.subplots (1, 2)
64  plt.subplots_adjust (wspace=.3)
65  fig.set_size_inches (8, 4)
66  ax1.scatter (x, y, c="b", marker="+", alpha=.7)
67  model_x = [-2, 3]
68  model_y = [real_w1 + real_w2 * x for x in model_x]
69  learned_y = [learned_weights[0] \
70       + learned_weights[1] * x for x in model_x]
71  ax1.plot (model_x, model_y, c="b", alpha=0.6)
72  ax1.plot (model_x, learned_y, c="g", alpha=0.6)
```

Affichage (fin)

```
73  ax2.plot (range (0, n_steps), loss_amount, c="g")
74  ax2.plot (range (0, n_steps), \
75    real_error * np.ones ((n_steps, 1)), c="b")
76  ax2.set_ylabel ("Error")
77  ax2.set_xlabel ("Number of steps")
78  fig.tight_layout()
79  plt.savefig('images/neurone_seul.png')
80  _ = 'images/neurone_seul.png'
```

Résultats (affichage)

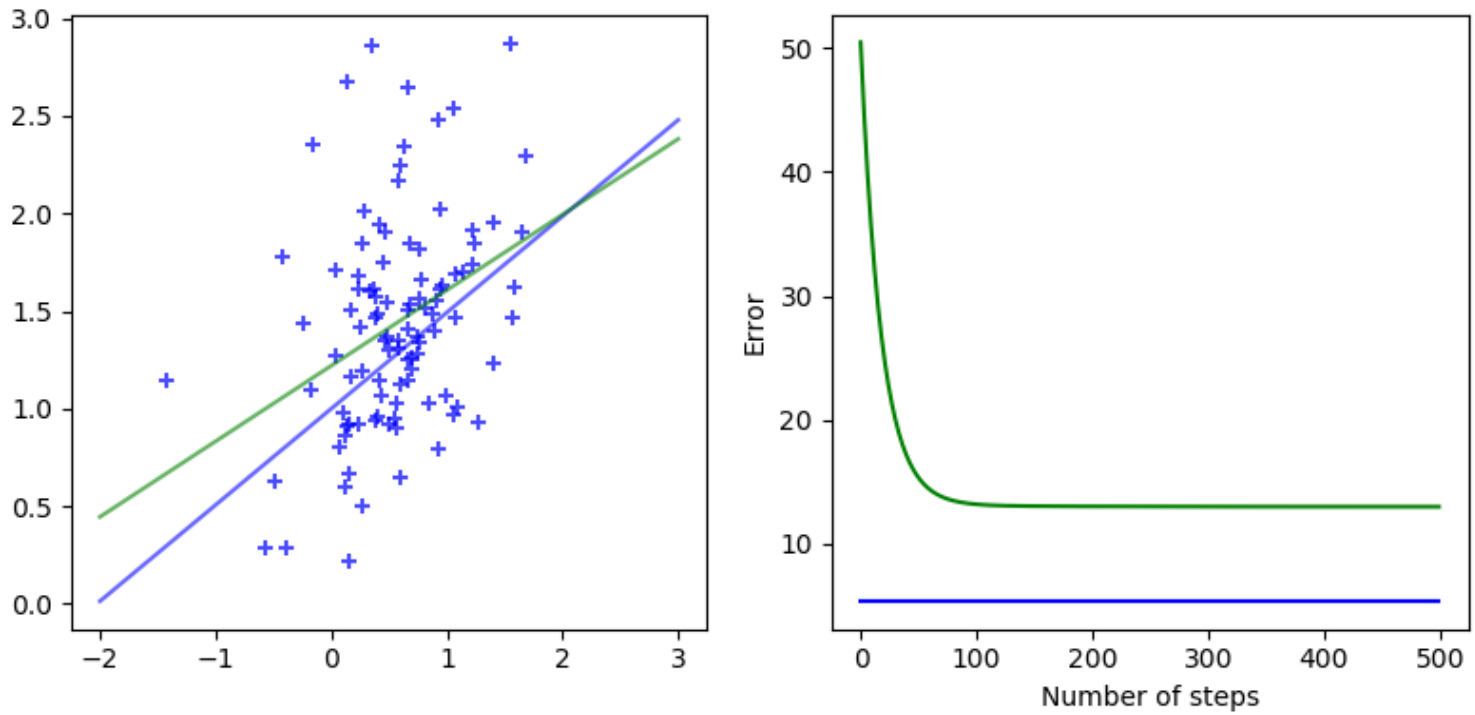

Figure 2: Résultats du neurone seul

Résultats (interprétation)

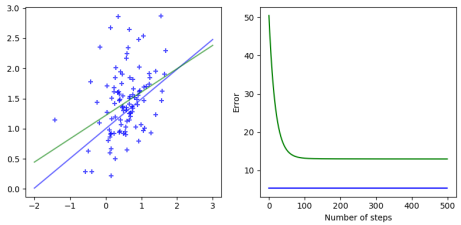

Figure 3: Neurone seul

- bleu : données réelles (positions, modèle, erreur), vert : apprentissage ;
- le biais et la pente sont bien approximés, le modèle converge en environ 100 itérations ;
- il reste toujours une erreur non nulle, et largement supérieure à l'erreur due à la dispersion des données, à cause de l'erreur sur la pente.

Plan

Téléchargement des données

```
if [ -d "./mnist-data" ]
then echo "Already downloaded"
else mkdir "./mnist-data"
  export SOURCE="http://yann.lecun.com/exdb/mnist/";
  export FILES="train-images-idx3-ubyte.gz \
train-labels-idx1-ubyte.gz t10k-images-idx3-ubyte.gz \
t10k-labels-idx1-ubyte.gz"
  for file in $FILES
  do  wget -O "./mnist-data/"$file $SOURCE$file;
    gunzip -d "./mnist-data/"$file
    echo "$SOURCE$file downloaded"
  done;
fi;
```

Lecture des données

- Le site répertorie comment on doit lire les données : `http://yann.lecun.com/exdb/mnist/`
- ce sont des fichiers binaires, on utilise par exemple les instructions sur `http://www.devdungeon.com/content/working-binary-data-python`

Lecture des étiquettes

```
81  def read_labels (file):
82      file.seek (4)
83      # Network byteorder is "big"
84      n = int.from_bytes (file.read (4), byteorder='big')
85      if (n > 100):
86  n=100
87      labels_bytes = file.read (n);
88      i_label = np.frombuffer (labels_bytes, dtype=np.uint8)
89      maxi = np.max (i_label)
90      one_hot = np.zeros ((n, maxi + 1)).astype (np.float32)
91      one_hot[np.arange(n), i_label] = 1
92      return one_hot
```

Lecture des images

```
93  def read_images (file):
94    file.seek (4)
95    n_i = int.from_bytes (file.read (4), byteorder='big')
96    if (n_i > 100):
97      n_i=100
98    n_rows = int.from_bytes (file.read (4), byteorder='big')
99    n_cols = int.from_bytes (file.read (4), byteorder='big')
100   b = file.read (n_i * n_rows * n_cols)
101   data = np.reshape (np.frombuffer (b, dtype=np.uint8), \
102      (n_i, n_rows * n_cols))#
103   f = data.astype (np.float32)
104   data_c = (f - np.mean (f, axis=1, keepdims=True)) \
105     / np.std (f, axis=1, keepdims=True) #
106   return np.reshape (data_c, (n_i, n_rows, n_cols, 1)) #
```

Noms des fichiers

On définit les noms des fichiers d'images et de labels.

```
107 train_labels_f = "./mnist-data/train-labels-idx1-ubyte"
108 test_labels_f = "./mnist-data/t10k-labels-idx1-ubyte"
109 train_img_f = "./mnist-data/train-images-idx3-ubyte"
110 test_img_f = "./mnist-data/t10k-images-idx3-ubyte"
```

Discussion

- On utilise pour les labels la représentation 1-hot, qui permet de ne manipuler que des données quantitatives ;
- pour les images, afin de pouvoir définir la convolution, il faut une matrice de pixels (et pas un vecteur) ;
- ligne 102, on écrit les données sous forme de matrice (chaque image étant une ligne) pour le centrage ;
- ligne 105, on centre et on réduit les données ;
- ligne 106, on utilise la fonction `reshape` de NumPy, qui permet de retrouver un bloc d'images à partir d'une série de données, chaque image étant une matrice de pixel (pour les chiffres, chaque pixel a une seule dimension) ;
- on utilisera ces fonctions après avoir ouvert un fichier en lecture.

Chargement des données

```
111  with open (test_img_f, "rb") as f:
112      test_images = read_images (f)

113  with open (test_labels_f, "rb") as f:
114      test_labels = read_labels (f)

115  with open (train_img_f, "rb") as f:
116      train_images = read_images (f)

117  with open (train_labels_f, "rb") as f:
118      train_labels = read_labels (f)
```

Première image de test

```
119  plt.figure ()
120  plt.imshow (np.reshape (test_images[0,:,:,:], (28, 28)))
121  plt.tight_layout()
122  plt.savefig('images/premier_test.png')
123  _ = 'images/premier_test.png'
```

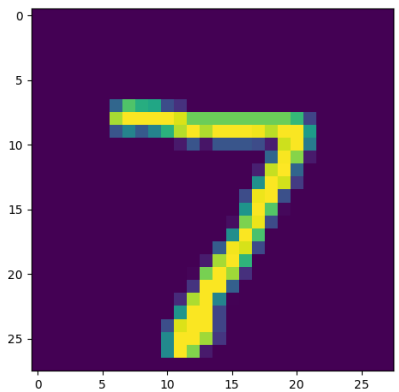

Figure 4: Première image de test

Modèle de réseau de neurones

Le modèle utilisé pour entraîner le réseau de neurones contient une couche de convolution à 10 filtres 5x5, une couche ReLu, une couche de max-pooling et une couche « fully connected » (dont la sortie est de dimension 10x1).

Fabrication d'une couche de convolution

```
124  def define_convlayer_weights (size, n_channels, n_layers):
125      weights = tf.Variable (tf.random_normal \
126  ([size, size, n_channels, n_layers]))
127      bias = tf.Variable (tf.zeros ([n_layers]))
128      return (weights, bias)
129
130  def define_convlayer (input_data, weights, bias):
131      conv_filter = tf.nn.conv2d (input_data, weights, \
132  strides=[1, 1, 1, 1], padding='SAME')
133      return tf.nn.bias_add (conv_filter, bias)
```

Fabrication d'une couche de convolution - Discussion

- Comme précédemment, on crée des poids aléatoires ;
- on crée un biais nul ;
- contrairement au neurone seul (ligne 43), on n'utilise que des données de type `float32` (chez moi, ça plante) ;
- NB : pour les chiffres qui sont en noir et blanc, `n_channels` sera tout le temps égal à 1, mais si on veut utiliser plusieurs vues, on peut en mettre une par canal.

Fabrication d'une couche ReLu

```
134 def define_relulayer (input_data):
135     return tf.nn.relu (input_data)
```

On n'a pas besoin de modifier `tf.nn.relu` pour l'adapter.

Fabrication d'une couche de max-pooling

```
136  def define_maxpoolinglayer (input_data, size):
137      shape = [1, 2, 2, 1] #
138      return tf.nn.max_pool (input_data, ksize=shape, strides=
139     padding='SAME')
```

La définition de `shape` (ligne 137) signifie que le maximum sera effectué image par image, par blocs de 2×2 pixels, canal par canal. Autant on pourrait effectuer un max pooling sur plusieurs images à la fois, si leur ordre indiquait leur proximité (ex : des vues d'un objet 3D obtenues dans un ordre cohérent), autant on ne peut absolument rien dire sur le nombre de canaux.

Fabrication d'une couche « Fully Connected »

Il y a une petite difficulté : pour retrouver une couche de réseaux de neurones classique, on ne peut plus travailler sur des images, il faut des vecteurs. C'est ce qui se passe ligne 149.

Fabrication d'une couche « Fully Connected » - code

```
140  def define_fclayer_weights (input_shape, out_dim):
141      n = int (reduce (mul, input_shape))
142      weights = tf.Variable (tf.random_normal ([n, out_dim]))
143      bias = tf.Variable (tf.zeros ((1, out_dim)))
144      return (weights, bias)
145
146  def define_fclayer (in_data, weight, bias):
147      n_input = in_data.get_shape ().as_list ()[0]
148      in_dim = weight.get_shape ().as_list ()[0]
149      vector_input = tf.reshape (in_data, [n_input, in_dim])
150      bias_mat = tf.tile (bias, [n_input, 1])
151      return tf.matmul (vector_input, weight) + bias_mat
```

Définition du modèle - création des poids

```
152  def define_graph_weights (data_shape, output_dim):
153      n = data_shape[1];
154      p = data_shape[2];
155      n_chan = data_shape[3];
156      (weights_conv, bias_conv) = define_convlayer_weights (5,
157      (weights_fc, bias_fc) = \
158  define_fclayer_weights ((n / 2, p / 2, n_chan * 10), output_
159      return (weights_conv, bias_conv, weights_fc, bias_fc)
```

On crée tous les poids pour le graphe tout entier.

Définition du modèle

```
160  def define_graph (X, y, w1, b1, w2, b2, reg):
161      conv = define_convlayer (tf.constant (X), w1, b1)
162      relu = define_relulayer (conv)
163      max_pooling = define_maxpoolinglayer (relu, 2)
164      prediction = define_fclayer (max_pooling, w2, b2)
165      error = tf.subtract (prediction, tf.constant (y))
166      loss_functions = tf.nn.softmax (error)
167      loss = tf.reduce_mean (loss_functions) \
168    + reg * (tf.nn.l2_loss (w1) + tf.nn.l2_loss (b1) \
169      + tf.nn.l2_loss (w2) + tf.nn.l2_loss (b2))
170      return ((conv, relu, max_pooling, prediction), loss)
```

On utilise les valeurs déjà créées des poids, comme ça on pourra faire un graphe pour le test qui réutilisera les mêmes poids.

Paramètres d'apprentissage

```
171 n_steps = 500
172 loss_amount = np.zeros ((n_steps))
173 learned_weights = []
```

Comme pour le neurone seul, on va suivre l'évolution de l'erreur en fonction de l'étape.
Comme pour le neurone seul, on utilise exactement le même code de base.

Apprentissage

```
174  with tf.Session () as session:
175    (w, b, W, B) = define_graph_weights \
176      (train_images.shape, 10)
177    ((d1, d2, d3, d4), loss) = \
178      define_graph(train_images,train_labels,w,b,W,B,0.001)
179    tf.global_variables_initializer ().run ()
180    descend = tf.train.GradientDescentOptimizer (0.002)\
181      .minimize (loss)
182    for i in range (n_steps):
183      descend.run ()
184      loss_amount[i] = loss.eval ()
185    learned_weights=(w.eval(),b.eval(),W.eval(),B.eval())
```

Erreur commise en apprentissage

```
186  fig = plt.figure ()
187  plt.plot (range (0, n_steps), loss_amount, c="g")
188  plt.set_ylabel ("Error")
189  plt.set_xlabel ("Number of steps")
190  fig.tight_layout()
191  plt.savefig('images/loss.png')
192  _ = 'images/loss.png'
```

Erreur commise en apprentissage (résultats)

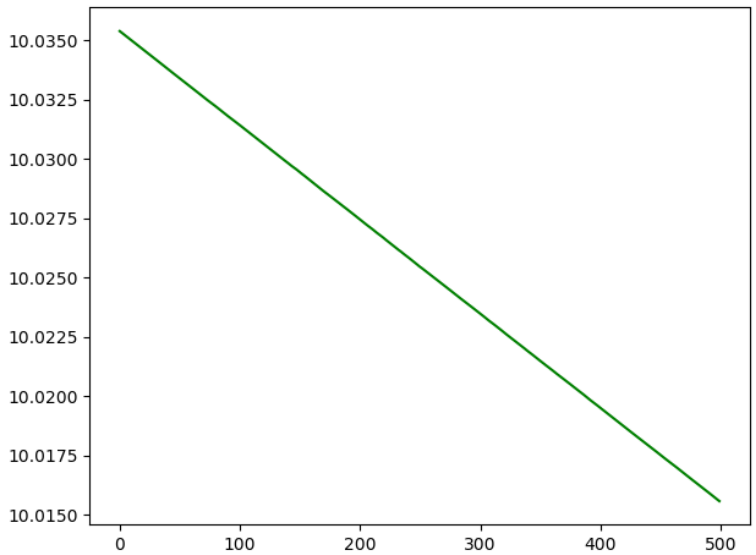

Figure 5: Erreur commise en apprentissage

Comme on n'est pas sur une bête de calcul, on s'arrête à 500 itérations, bien que dans la réalité il en faudrait beaucoup, beaucoup plus.

Obtention des descripteurs

```
193  def get_descriptors (images, labels):
194    with tf.Session () as session:
195      w1 = tf.constant (learned_weights[0])
196      b1 = tf.constant (learned_weights[1])
197      w2 = tf.constant (learned_weights[2])
198      b2 = tf.constant (learned_weights[3])
199      ((d1, d2, d3, d4), loss) = \
200        define_graph (images, labels, w1, b1, w2, b2, 0.001)
201      tf.global_variables_initializer ().run ()
202      data = (d1.eval (), d2.eval (), d3.eval (), d4.eval ())
203    return data
```

Il s'agit simplement d'appliquer le graphe et de récupérer chaque couche.

Requêtage (fonction distance)

```
204  database = get_descriptors (train_images, train_labels)
205  def distance (weights, descr_test, descr_train):
206      n = weights.shape[0]
207      s = 0
208      for i in range (n):
209  diff = np.subtract (descr_test[i], descr_train[i])
210  s += weights[i] * np.linalg.norm (diff)
211      return s
```

Calcule une distance entre deux ensembles de descripteurs, en donnant un poids différent à chacun. Typiquement, un poids faible en surface et un poids plus important en profondeur.

Requêtage (par rapport à la BDD)

```
212  def knn (k, weights, test):
213      db_1, db_2, db_3, db_4 = database
214      n = db_1.shape[0]
215      dist_to_test = np.zeros ((n))
216      for i in range (n):
217  db = (db_1[i, :, :, :], db_2[i, :, :, :], \
218       db_3[i, :, :, :], db_4[i, :])
219  dist_to_test[i] = distance (weights, test, db)
220      return np.argsort (dist_to_test)[:k]
```

On utilise un KNN modifié, qui agrège les résultats de KNN sur toutes les couches avec des poids.

Test du requêtage

Souvenons-nous de la première image de test :

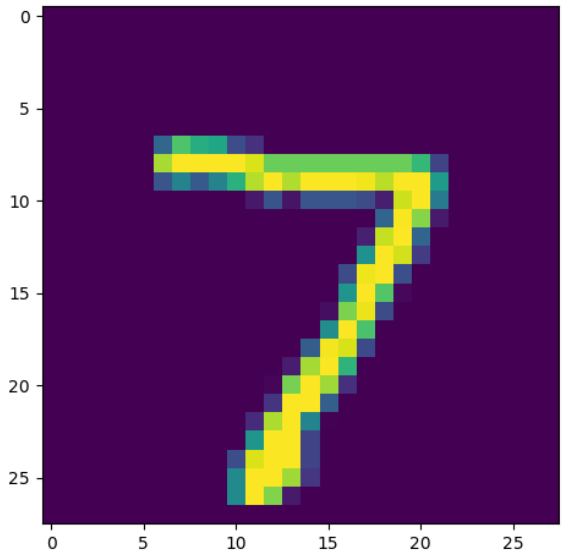

Figure 6: La première image de test

Nous allons maintenant la requêter, en donnant autant une importance croissante des couches selon la profondeur.

Test du requêtage (code)

```
221  X = np.array ([test_images[0]])
222  y = np.array ([test_labels[0]])
223  request = get_descriptors (X, y) #
224  knn_weights = np.array ([0.125, 0.125, 0.25, 0.5])
225  response = knn (3, knn_weights, request) #
```

On applique le réseau sur le premier test (ligne 223), puis on obtient les trois meilleurs représentants (ligne 225).

Test du requêtage (affichage)

```
226  fig = plt.figure ()
227  for i in range (3):
228      sp = fig.add_subplot (1, 3, i+1)
229      img = train_images[response[i], :, :, :]
230      img = np.reshape (img, (28, 28))
231      imgplot = plt.imshow (img)
232      title = 'Choice number ' + str (i + 1) + ': ' \
233      + str (response[i])
234      sp.set_title (title)
235
236  fig.tight_layout ()
237  fig.savefig ('images/requete.png')
238  _ = 'images/requete.png'
```

Résultat du requêtage

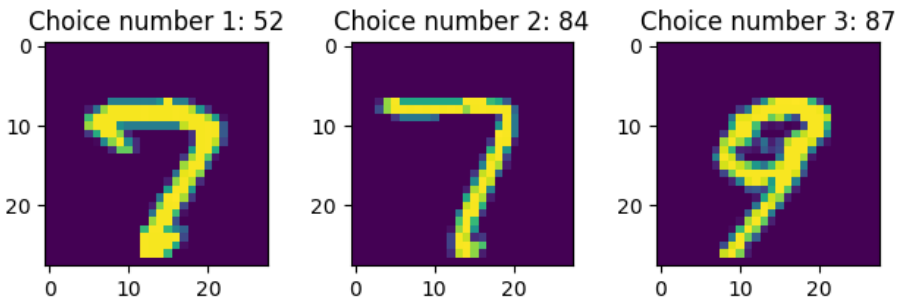

Figure 7: Résultat de la requête de la figure 6

Plan

Conclusion

- On fait du requêtage ! :-)
- La très petite base et le très petit réseau donnent des résultats qui ne sont pas dignes d'être considérés comme de production ;
- Tensorflow permet d'avoir une image avec un nombre arbitraire de canaux (et pas juste RGB), donc on peut mettre une vue par canal et n'avoir qu'une image représentative pour un individu.