Deep Learning for Shape Retreival

Tianning Yu, Vivien Kraus, École Centrale de Lyon

30 mars 2017

Table des matières

1 Introduction

Nous allons dans ce projet utiliser Tensorflow pour une application de requêtage de forme.

2 Application

Le programme se déroule en plusieurs parties : lecture des données, fabrication du graphe, descente du gradient, requêtage, évaluation.

N'ayant finalement pas eu accès à la capacité de calcul proposée par le laboratoire, nous ne pouvons présenter les résultats que sur des données jouet.

2.1 Initialisation de Tensorflow

Nous avons besoin de charger le module `tensorflow` de python, et les outils dont nous aurons besoin.

```
1 import tensorflow as tf
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from functools import reduce
5 from operator import mul
```

2.2 Lecture des données

Les données d'entrée sont un ensemble de vues de l'objet 3D. Nous réutilisons les images de l'équipe SHREC2016.

2.2.1 Extraction de l'archive

```
tar xf database.tar.gz
```

Les images sont dans le dossier `database/`, réparties en deux sous-dossiers : `test` et `train`.

2.2.2 Compilation de l'archive au format idx

Pour faciliter le code, nous utilisons le format idx, tel qu'utilisé par Yann LeCun pour la base de données MNIST.

```
rm -f shrec-to-mnist.tar.gz
wget https://planete-kraus.eu/~vivien/centrale/shrec-to-mnist.tar.gz
tar xf shrec-to-mnist.tar.gz
cd idx
ocamlbuild -use-ocamlfind src/main.native -- ../database ../
```

2.2.3 Lecture d'un tenseur à partir d'un fichier IDX

On suit les spécifications du fichier IDX, en supposant qu'on dispose de données au format **uint8**.

```
def read_idx_ubyte (file):
    file.seek (3)
    depth = int.from_bytes (file.read (1), byteorder='big')
    dims = []
    n = 1
    for i in range (depth):
        d = int.from_bytes (file.read (4), byteorder='big')
        n = n * d
        dims.append (d)
    all_bytes = file.read (n)
    uints = np.frombuffer (all_bytes, dtype=np.uint8)
    return np.reshape (uints, tuple (dims))
```

2.2.4 Lecture d'étiquettes

On transforme les indices d'étiquettes en représentation *one-hot*.

```
def read_targets (file):
    indices = read_idx_ubyte (file)
    (n,) = np.shape (indices)
    maxi = np.max (indices)
    one_hot = np.zeros ((n, maxi + 1)).astype (np.float32)
    one_hot[np.arange(n), indices] = 1
    return one_hot
```

Application :

```
with open ("train-labels-idx1.ubyte", "rb") as f:
    train_labels = read_targets (f)

with open ("test-labels-idx1.ubyte", "rb") as f:
    test_labels = read_targets (f)
```

2.2.5 Lecture d'images

On normalise les images : on enlève la moyenne des valeurs des pixels et on divise par l'écart-type.

Attention : dans le fichier, les données sont sous la forme individu > vue > ligne > pixel, mais Tensorflow les veut sous la forme individu > ligne > pixel > vue. Il faut donc transposer le tenseur.

```
def read_images (file):
    pixels = read_idx_ubyte (file)
    (n, v, h, w) = np.shape (pixels)
    pixels_flt = pixels.astype (np.float32)
    for i in range (n):
```

```
        for j in range (v):
            im = np.reshape (pixels_flt[i, j, :, :], (h * w))
            im = (im - np.mean (im)) / np.std (im)
            pixels_flt[i, j, :, :] = np.reshape (im, (1, 1, h, w))
    return np.transpose (pixels_flt, (0, 2, 3, 1))
```

Application :

```
with open ("train-images-idx4.ubyte", "rb") as f:
    train_images = read_images (f)

with open ("test-images-idx4.ubyte", "rb") as f:
    test_images = read_images (f)
```

2.2.6 Visualisation de la première image de test

```
plt.figure ()
(n_tests, h, w, n_views) = np.shape (test_images)
fig, axes = plt.subplots (1, n_views)
for i in range (n_views):
    axes[i].imshow (np.reshape (test_images[1,:,:,i], (h, w)))
    axes[i].set_title ('View number ' + str (i + 1))

plt.tight_layout()
plt.savefig('images/premier_test.png')
_ = 'images/premier_test.png'
```

2.3 Fabrication du graphe

Étant donné la faible puissance de calcul dont nous disposons, nous n'allons pas pouvoir faire un vrai graphe de production. Nous allons devoir nous contenter de quelques couches de convolution, une couche ReLu, une couche de max-pooling et une couche « fully-connected ».

2.3.1 Éléments du graphe

On spécialise Tensorflow pour créer les « briques » de base de notre graphe.

Couche de convolution La couche de convolution contient `n_layers` filtres, et est de taille size × size.

On définit deux fonctions :

— `define_convlayer_weights` (ligne 55) : crée un tenseur Tensorflow pour les paramètres ;

— `define_convlayer` (ligne 61) : insère l'opération de convolution dans le graphe de calcul.

La séparation est nécessaire pour pouvoir réutiliser les poids calculés par l'apprentissage pour les tests.

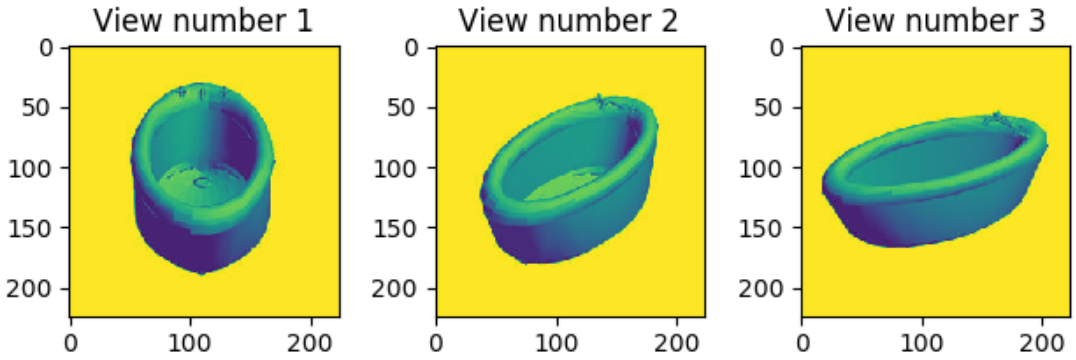

FIGURE 1 – Premier objet à tester

```
55 def define_convlayer_weights (size, n_views, n_layers): #
56     weights = tf.Variable (0.1 * tf.random_normal \
57         ([size, size, n_views, n_layers]))
58     bias = tf.Variable (tf.zeros ((n_layers)))
59     return (weights, bias)
60
61 def define_convlayer (input_data, w): #
62     (weights, bias) = w
63     conv_filter = tf.nn.conv2d (input_data, weights, \
64         strides=[1, 1, 1, 1], padding='SAME')
65     return tf.nn.bias_add (conv_filter, bias)
```

Les poids sont initialisés aléatoirement, avec un biais nul.

Couche ReLu La couche ReLu (Rectified Linear Units) permet d'introduire une non-linéarité. Il n'y a pas de paramètres à optimiser sur cette couche, c'est pourquoi on n'a qu'une seule fonction.

```
66 def define_relulayer (input_data):
67     return tf.nn.relu (input_data)
```

Couche de max-pooling Ici non plus, il n'y a pas de paramètres.

```
68 def define_maxpoolinglayer (input_data):
69     shape = [1, 2, 2, 1] #
70     return tf.nn.max_pool (input_data, ksize=shape, strides=shape, \
71                            padding='SAME')
```

La définition de `shape` (ligne 69) signifie que le maximum sera effectué objet par objet, vue par vue, par blocs de 2×2 pixels.

Couche « Fully connected » On retrouve dans cette couche la séparation en deux fonctions.

```
72 def define_fclayer_weights (input_shape, out_dim):
73     n = int (reduce (mul, input_shape))
74     weights = tf.Variable (tf.random_normal ([n, out_dim]))
75     bias = tf.Variable (tf.zeros ((1, out_dim)))
76     return (weights, bias)
77
78 def define_fclayer (in_data, w):
79     (weights, bias) = w
80     n_input = in_data.get_shape ().as_list ()[0]
81     in_dim = weights.get_shape ().as_list ()[0]
82     vector_input = tf.reshape (in_data, [n_input, in_dim]) #
83     bias_mat = tf.tile (bias, [n_input, 1])
84     return tf.matmul (vector_input, weights) + bias_mat
```

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 82.

Couche conv-ReLu On crée une couche de convolution suivie immédiatement d'un ReLu.

```
85 def define_convrelu_weights (size, n_views, n_layers):
86     return define_convlayer_weights (size, n_views, n_layers)
87
88 def define_convrelu (input_data, w):
89     conv_filter = define_convlayer (input_data, w)
90     return define_relulayer (conv_filter)
```

Couche triconvrelu-pool On enchaîne trois couches conv-relu puis une couche de max-pooling.

```
91  def define_triconvrelu_weights (size, n_views, n_layers):
92      w1 = define_convrelu_weights (size, n_views, n_layers)
93      w2 = define_convrelu_weights (size, n_views, n_layers)
94      w3 = define_convrelu_weights (size, n_views, n_layers)
95      return (w1, w2, w3)
96
97  def define_triconvrelu (input_data, w):
98      (w1, w2, w3) = w
99      exit_1 = define_convrelu (input_data, w1)
100     exit_2 = define_convrelu (exit_1, w2)
101     exit_3 = define_convrelu (exit_2, w3)
102     return define_maxpoolinglayer (exit_3)
```

2.3.2 Création du graphe

Nous allons utiliser les différentes briques de base pour créer le graphe total.

Création des poids On crée les poids pour le graphe.

```
103 def define_graph_weights (data_shape, out_dim):
104     n_items = data_shape[0]
105     n = data_shape[1]
106     p = data_shape[2]
107     n_views = data_shape[3]
108     tcr1 = define_triconvrelu_weights (5, n_views, n_views)
109     tcr2 = define_triconvrelu_weights (5, n_views, n_views)
110     tcr3 = define_triconvrelu_weights (5, n_views, n_views)
111     fc = define_fclayer_weights ((n / 8, p / 8, n_views), \
112             out_dim)
113     return (tcr1, tcr2, tcr3, fc)
```

Création du graphe On lie tous les éléments ensemble.
On utilise une régularisation L2 et une norme SoftMax.

```
114 def define_graph (X, y, weights, reg):
115     (tcr1, tcr2, tcr3, fc) = weights
116     conv1 = define_triconvrelu (tf.constant (X), tcr1)
117     conv2 = define_triconvrelu (conv1, tcr2)
118     conv3 = define_triconvrelu (conv2, tcr3)
119     prediction = define_fclayer (conv3, fc)
120     error = tf.subtract (prediction, tf.constant (y))
121     loss_functions = tf.nn.softmax (error)
122     (w1, w2, w3, w4) = weights
```

```
    (w11, w12, w13) = w1
    (w21, w22, w23) = w2
    (w31, w32, w33) = w3
    (w4w, w4b) = w4
    (w11w, w11b) = w11
    (w12w, w12b) = w12
    (w13w, w13b) = w13
    (w21w, w21b) = w21
    (w22w, w22b) = w22
    (w23w, w23b) = w23
    (w31w, w31b) = w31
    (w32w, w32b) = w32
    (w33w, w33b) = w33
    regularization = reg * \
        (tf.nn.l2_loss (w4w) + tf.nn.l2_loss (w4b) \
         + tf.nn.l2_loss (w11w) + tf.nn.l2_loss (w11b) \
         + tf.nn.l2_loss (w12w) + tf.nn.l2_loss (w12b) \
         + tf.nn.l2_loss (w13w) + tf.nn.l2_loss (w13b) \
         + tf.nn.l2_loss (w21w) + tf.nn.l2_loss (w21b) \
         + tf.nn.l2_loss (w22w) + tf.nn.l2_loss (w22b) \
         + tf.nn.l2_loss (w23w) + tf.nn.l2_loss (w23b) \
         + tf.nn.l2_loss (w31w) + tf.nn.l2_loss (w31b) \
         + tf.nn.l2_loss (w32w) + tf.nn.l2_loss (w32b) \
         + tf.nn.l2_loss (w33w) + tf.nn.l2_loss (w33b))
    loss = tf.reduce_mean (loss_functions) + regularization
    return ((conv1, conv2, conv3, prediction), loss, prediction, error)
```

2.4 Descente du gradient

Nous pouvons maintenant apprendre le réseau de neurones.

2.4.1 Paramètres d'apprentissage

Nous allons effectuer un certain nombre d'itérations, en suivant à chaque fois l'erreur marginale commise.

Nous nous apprêtons également à sauver les poids appris.

```
n_steps = 2000
learning_rate = 0.5
loss_amount = np.zeros ((n_steps))
learned_weights = []
```

2.4.2 Apprentissage

Il s'agit du cœur de l'algorithme, c'est ici que tous les calculs sont faits.

```
153 def learn (n_steps, learning_rate):
154   loss_amount = np.zeros ((n_steps))
155   with tf.Session () as session:
156     graph_weights = define_graph_weights \
157         (train_images.shape, train_labels.shape[1])
158     tf.global_variables_initializer ().run ()
159     ((d1, d2, d3, d4), loss, prediction, error) = \
160         define_graph(train_images,train_labels,graph_weights,1e-3)
161     descend = tf.train.GradientDescentOptimizer (learning_rate)\
162                       .minimize (loss)
163     for i in range (n_steps):
164       descend.run ()
165       loss_amount[i] = loss.eval ()
166       # print ("Attendu\unhbox \voidb@x \penalty \@M \ :")
167       # print (train_labels)
168       # print ("Prction\unhbox \voidb@x \penalty \@M \ :")
169       # print (prediction.eval ())
170       # print ("Erreur\unhbox \voidb@x \penalty \@M \ :")
171       # print (error.eval ())
172     (w1, w2, w3, w4) = graph_weights
173     (w11, w12, w13) = w1
174     (w21, w22, w23) = w2
175     (w31, w32, w33) = w3
176     (w4w, w4b) = w4
177     (w11w, w11b) = w11
178     (w12w, w12b) = w12
179     (w13w, w13b) = w13
180     (w21w, w21b) = w21
181     (w22w, w22b) = w22
182     (w23w, w23b) = w23
183     (w31w, w31b) = w31
184     (w32w, w32b) = w32
185     (w33w, w33b) = w33
186     e11 = (w11w.eval (), w11b.eval ())
187     e12 = (w12w.eval (), w12b.eval ())
188     e13 = (w13w.eval (), w13b.eval ())
189     e21 = (w21w.eval (), w21b.eval ())
190     e22 = (w22w.eval (), w22b.eval ())
191     e23 = (w23w.eval (), w23b.eval ())
192     e31 = (w31w.eval (), w31b.eval ())
193     e32 = (w32w.eval (), w32b.eval ())
194     e33 = (w33w.eval (), w33b.eval ())
195     e1 = (e11, e12, e13)
196     e2 = (e21, e22, e23)
```

```
    e3 = (e31, e32, e33)
    e4 = (w4w.eval (), w4b.eval ())
    return (loss_amount, (e1, e2, e3, e4))

(loss_amount, learned_weights) = learn (n_steps, learning_rate)
```

2.4.3 Erreur commise

Nous avons suivi l'erreur commise à chaque descente de gradient. Il s'agit de la figure `fig:code-loss`.

```
fig, ax = plt.subplots ()
plt.plot (range (0, n_steps), loss_amount, c="g")
ax.set_ylabel ("Error")
ax.set_xlabel ("Number of steps")
fig.tight_layout()
plt.savefig('images/loss.png')
_ = 'images/loss.png'
```

Sur la figure `fig:code-loss`, nous voyons que l'erreur commise décroît exponentiellement et tend vers 0. Si nous la laissons décroître trop longtemps, il risque d'y avoir sur-apprentissage. Nous y reviendrons.

2.5 Requêtage

Contrairement aux problèmes de classification, nous voulons ici faire du requêtage. Plus exactement, étant donné un individu de test, il faut trouver les individus d'apprentissage desquels il est le plus proche.

Pour pouvoir effectuer une évaluation de notre requêtage, nous allons définir la **précision** et le **rappel**, pour pouvoir ensuite tracer une courbe ROC de notre requêtage en faisant varier le nombre d'individus sélectionnés.

Nous allons donc suivre la procédure décrite sur le site de SHREC 2016, cependant nous n'avons pas de sous-catégorie, et donc nous ne pourrons qu'utiliser la précision, le rappel, la F-mesure, et le MAP (Minimum Average Precision, i.e. la moyenne sur tous les tests de l'aire sous la courbe ROC).

La définition de précision et de rappel n'est pas nécessairement très adaptée, car elle ne tient pas compte de l'ordre dans lequel apparaissent les résultats.

Pour effectuer le requêtage, nous allons comparer le résultat de l'application du graphe à chaque test au résultat de l'application du graphe à chaque apprentissage. Nous allons donc effectuer une sorte de KNN (K-nearest-neighbor), en définissant clairement le résultat de l'application du graphe : c'est la sortie de **chaque couche** du graphe. On compare deux résultats entre eux en comparant les descripteurs deux à deux, et en pondérant chaque descripteur.

2.5.1 Obtention des descripteurs

L'idée est très simple : on crée les poids avec les données de l'apprentissage et on applique le graphe, puis on récupère les valeurs des descripteurs.

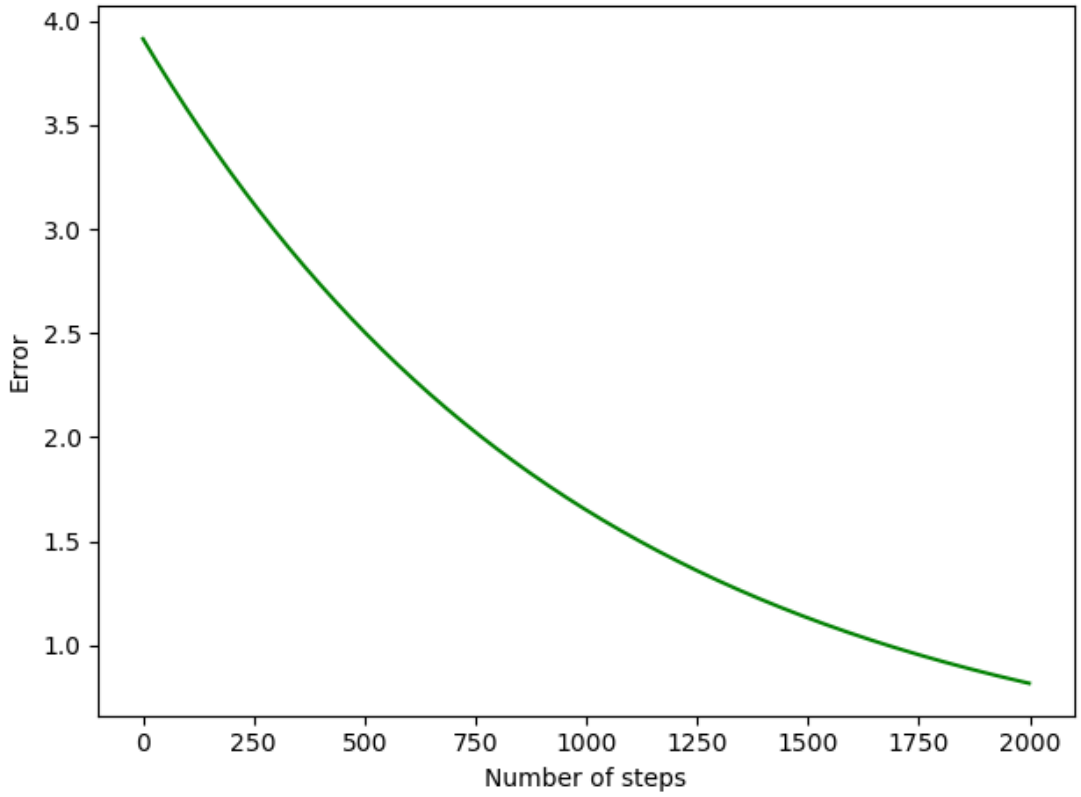

FIGURE 2 – Suivi de l'erreur commise à l'apprentissage

```
209 def get_descriptors (images, labels, learned_weights):
210   with tf.Session () as session:
211     (l1, l2, l3, l4) = learned_weights
212     def make_constant_tuple (t):
213       return tuple (map (tf.constant, t))
214     w1 = tuple (map (make_constant_tuple, l1))
215     w2 = tuple (map (make_constant_tuple, l2))
216     w3 = tuple (map (make_constant_tuple, l3))
217     w4 = make_constant_tuple (l4)
218     w = (w1, w2, w3, w4)
219     ((d1, d2, d3, d4), loss, prediction, error) = \
220       define_graph (images, labels, w, 0.001)
221     tf.global_variables_initializer ().run ()
222     data = (d1.eval (), d2.eval (), d3.eval (), d4.eval ())
223   return data
```

2.5.2 Distance entre deux ensembles de descripteurs

Afin de pouvoir créer le KNN, il faut pouvoir mesurer les distances entre ensembles de descripteurs. Pour cela, nous pondérons ceux-ci.

```
224 def distance (weights, descr_test, descr_train):
225     n = weights.shape[0]
226     s = 0
227     for i in range (n):
228         diff = np.subtract (descr_test[i], descr_train[i])
229         s += weights[i] * np.linalg.norm (diff)
230     return s
```

2.5.3 Fonction de requêtage (KNN)

Le KNN sélectionne les k plus proches apprentissages, sur présentation d'un test.

```
231 def rank (database, weights, test):
232     db_1, db_2, db_3, db_4 = database
233     n = db_1.shape[0]
234     dist_to_test = np.zeros ((n))
235     for i in range (n):
236         db = (db_1[i, :, :, :], db_2[i, :, :, :], \
237               db_3[i, :, :, :], db_4[i, :])
238         dist_to_test[i] = distance (weights, test, db)
239     return np.argsort (dist_to_test)
240
241 def knn (database, k, weights, test):
242     return rank (database, weights, test)[:k]
```

Notons que la base de données (`database`) est un tableau où chaque ligne contient les descripteurs de l'apprentissage correspondant.

Le paramètre important est le paramètre `weights`, qui signifie quels termes ont de l'importance. C'est un tuple de poids, chaque poids correspondant à une couche. On s'attend à ce que plus la couche soit éloignée, meilleur sera le résultat.

2.5.4 Test du requêtage

Nous allons requêter la figure 1.

```
243 database = get_descriptors (train_images, train_labels, learned_weights)
244 X = np.array ([test_images[1]])
245 y = np.array ([test_labels[1]])
246 request = get_descriptors (X, y, learned_weights)
247 knn_weights = np.array ([0.125, 0.125, 0.25, 0.5])
248 response = knn (database, 3, knn_weights, request)
```

Nous allons maintenant dessiner nos trois meilleurs candidats : cf figure 3.

```
fig = plt.figure ()
(n_tests, h, w, n_views) = np.shape (test_images)
for i in range (3):
    for j in range (n_views):
        sp = fig.add_subplot (3, n_views, i * 3 + j +1)
        img = train_images[response[i], :, :, j]
        img = np.reshape (img, (h, w))
        imgplot = plt.imshow (img)
        title = 'Choice ' + str (i + 1) + '/3: ' \
                + str (response[i]) + ', view ' + str (j + 1)
        sp.set_title (title)

fig.tight_layout ()
fig.savefig ('images/requete.png')
_ = 'images/requete.png'
```

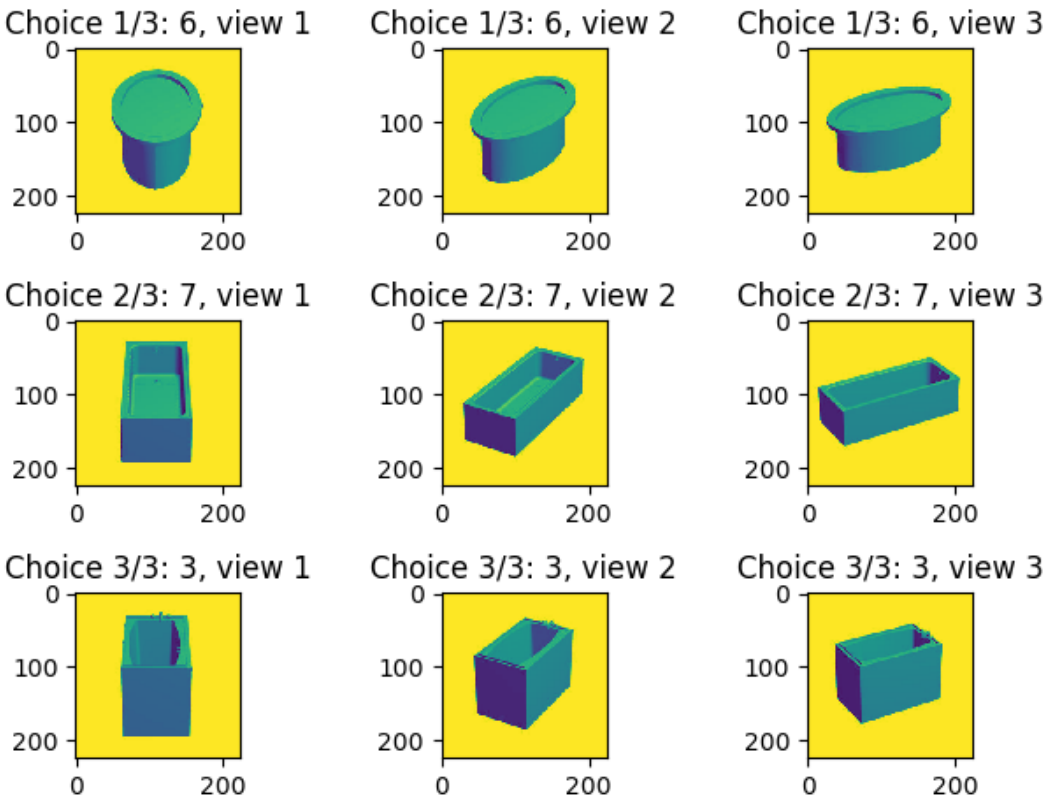

FIGURE 3 – Les trois meilleurs résultats dans l'ordre

Le résultat est assez décevant : notre réseau n'est probablement pas bien adapté, et il faudrait effectuer plus d'opérations de descente.

2.6 Évaluation

Comme évoqué plus haut, nous allons mesurer la précision et le rappel.

```
264 def compute_precision (tp, fp, fn, tn):
265     if tp + fp == 0:
266         return (0)
267     return (tp / (tp + fp))
```

```
268 def compute_recall (tp, fp, fn, tn):
269     if tp + fn == 0:
270         return (0)
271     return (tp / (tp + fn))
```

```
272 def score (ranks):
273     map = 0
274     for i in range (ranks.shape[0]):
275         reality = test_labels[i, :]
276         last_recall = 0
277         avep = 0
278         for cutoff in range (ranks.shape[1]):
279             tp = 0
280             fp = 0
281             tn = 0
282             fn = 0
283             for j in ranks[i, :cutoff]:
284                 other = train_labels[j, :]
285                 if np.dot (other, reality) > 0.99:
286                     tp = tp + 1
287                 else:
288                     fp = fp + 1
289             for j in ranks[i, cutoff:]:
290                 other = train_labels[j, :]
291                 if np.dot (other, reality) > 0.99:
292                     fn = fn + 1
293                 else:
294                     tn = tn + 1
295 
296             precision = compute_precision (tp, fp, fn, tn)
297             recall = compute_recall (tp, fp, fn, tn)
298             # f = (2 * precision * recall) / (precision + recall)
299             avep = avep + precision * (recall - last_recall)
```

```
        avep = avep / ranks.shape[1]
        map = map + avep
    map = map / ranks.shape[0]
    return map
```

Nous pouvons le tester avec les paramètres précédents :

```
n_test = test_labels.shape[0]
n_train = train_labels.shape[0]
ranks = np.zeros ((n_test, n_train)).astype (np.uint8)
for i in range (n_test):
    X = np.array ([test_images[i, :, :, :]])
    y = np.array ([test_labels[i]])
    req = get_descriptors (X, y, learned_weights)
    ranks[i, :] = rank (database, knn_weights, req)

the_map = score (ranks)
```

Le résultat est `30.29 %`.

3 Conclusion

Nous avons effectué un apprentissage de requêtage de forme avec Tensorflow. Malheureusement, nous avons dû nous contenter d'une version très restreinte de la base de données, ainsi que de très peu d'itérations.