

# Deep Learning for Shape Retrieval

Tianning Yu, Vivien Kraus, École Centrale de Lyon

30 mars 2017

## Table des matières

<b>1</b>	<b>Présentation générale du projet</b>	<b>2</b>
1.1	Contexte . . . . .	2
1.2	Tensorflow . . . . .	2
1.3	Base de données . . . . .	2
<b>2</b>	<b>Application</b>	<b>3</b>
2.1	Initialisation de Tensorflow . . . . .	3
2.2	Lecture des données . . . . .	3
2.2.1	Extraction de l'archive . . . . .	4
2.2.2	Compilation de l'archive au format idx . . . . .	4
2.2.3	Lecture d'un tenseur à partir d'un fichier IDX . . . . .	4
2.2.4	Lecture d'étiquettes . . . . .	5
2.2.5	Lecture d'images . . . . .	5
2.2.6	Première visualisation d'une image de test . . . . .	6
2.3	Fabrication du graphe . . . . .	6
2.3.1	Éléments du graphe . . . . .	6
2.3.2	Création du graphe . . . . .	9
2.4	Descente du gradient . . . . .	10
2.4.1	Paramètres d'apprentissage . . . . .	10
2.4.2	Apprentissage . . . . .	11
2.4.3	Erreur commise . . . . .	12
2.5	Requêtage . . . . .	12
2.5.1	Obtention des descripteurs . . . . .	13
2.5.2	Distance entre deux ensembles de descripteurs . . . . .	13
2.5.3	Fonction de requêtage (KNN) . . . . .	14
2.5.4	Test du requêtage . . . . .	14
2.6	Évaluation . . . . .	15
<b>3</b>	<b>Conclusion</b>	<b>16</b>
3.1	Résultat et limites . . . . .	16
3.2	Perspectives . . . . .	17

## Table des figures

1	Premier objet à tester . . . . .	7
2	Graphe utilisé . . . . .	18
3	Suivi de l'erreur commise à l'apprentissage . . . . .	19
4	Les trois meilleurs résultats dans l'ordre . . . . .	20

## 1 Présentation générale du projet

Un système de requêtage de contenu 3D est un système informatique pour la navigation, la recherche et la requêtage de contenus en trois dimensions à partir d'une grande base de données d'images numériques. Dans l'industrie, il a un certain nombre d'applications possibles, y compris la conception assistée par ordinateur, la simulation, la visualisation et le divertissement.

Parmi les stratégies possibles, les méthodes d'appariement de formes pour la requêtage de contenu 3D sont devenues populaires. Elles sont basées sur des techniques qui comparent et contrastent les similitudes entre les modèles 3D. D'un autre côté, la technique de Deep learning nous permet d'obtenir les caractéristiques d'un modèle 3D en plusieurs niveaux d'abstraction. Ces dernières années, ces méthodes ont considérablement amélioré l'état de l'art dans un certain nombre de domaines, y compris la détection et la reconnaissance d'objets.

Dans ce projet, nous avons étudié une méthode de Deep Learning dans le but d'effectuer du requêtage de formes 3D.

*Une version PDF de ce document est disponible ici.*

### 1.1 Contexte

Le projet s'inscrit dans le cadre de l'activité Projet d'option telle que proposée à l'École Centrale de Lyon. Le projet est encadré par MM. Liming Chen et Emmanuel Dellandréa, les tuteurs du projet. Il reprend un projet présenté par le SHREC<sup>1</sup> en 2016.

Le projet a pour but à terme de pouvoir reconnaître, dans une base de cas appris, un nouvel objet 3D. Il s'agit ensuite de récupérer de l'information pertinente dans la base d'apprentissage. Par exemple, un robot est face à un objet inconnu qu'il doit saisir, et il se sert d'une base de données d'objets appris pour effectuer les bons mouvements et exercer la bonne force.

### 1.2 Tensorflow

Tensorflow est une bibliothèque de deep learning développée par Google<sup>2</sup>, et présentant des *bindings* pour le langage Python.

Nous avons présenté une introduction à Tensorflow pour le requêtage de chiffres MNIST, le support visuel est disponible en ligne.

### 1.3 Base de données

Les modèles 3D que nous avons utilisés est le sous-ensemble ShapeNetCore de ShapeNet qui contient environ 51 300 modèles 3D sur 55 catégories communes, chacune étant subdivisée en plu-

---

1. Shape Retrieval Contest, compétition de requêtage de forme

2. [<https://www.tensorflow.org/>]

sieurs sous-catégories. Ils sont aussi divisés en sous-ensemble de train / validation / test de 70% / 10% / 20%. Les modèles sont fournis en format OBJ et deux versions de données sont disponibles : uniformément alignées (ensemble de données régulier) et perturbés par des rotations aléatoires.

Il est à noter qu'un objet 3D de format OBJ est composé de plusieurs triangles et les coordonnées de leur sommet. Les relations entre ces triangles et faces représentent des caractéristiques en un niveau d'abstraction plus profond, mais à cette heure, on n'a pas de méthodes pour acquérir directement les caractéristiques d'objets 3D à partir de format OBJ. Une stratégie possible est de prendre plusieurs images de différents angles de vue pour chaque objet 3D, et les traiter comme d'autres images 2D. Dans ce cas, on peut facilement traiter les objets 3D en utilisant tous les méthodes qui sont déjà existantes pour les images 2D. Heureusement, il existe déjà une base de donnée d'images 2D en différents angles de vue pour les modèles d'objet 3D de ShapeNetCore. Dans cette base de données, chaque objet 3D a 12 images de différents angles de vue en format JPG.

À cause des ressources limitées dont nous disposons, nous n'avons cependant pu retenir que quelques individus d'apprentissage et de test, avec 3 vues pour chaque individu, et représentant seulement trois catégories (il n'y a pas de sous-catégories).

## 2 Application

Nous allons utiliser Tensorflow pour construire un réseau de neurones. Ce réseau de neurones prend en entrée un individu, et retourne un ensemble de descripteurs.

Ensuite, nous allons effectuer un appariement entre descripteurs selon l'algorithme du KNN (k-nearest-neighbors) pour obtenir pour chaque individu de test un ensemble d'individus d'apprentissage proches de l'individu de test.

Enfin, nous allons évaluer notre méthode de requêtage selon une adaptation la plus fidèle possible de celle employée au SHREC.

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 un ensemble de 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
```

Nous utiliserons notamment matplotlib pour l'affichage.

### 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, à ceci près que :

- notre capacité de calcul est très limitée, donc nous utilisons une très petite base de données ;
- nous n'utilisons que les trois premières vues, ce qui perd beaucoup d'information (on ne voit pas derrière).

### 2.2.1 Extraction de l'archive

L'archive est disponible [ici](#).

```
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, et tel qu'on l'a utilisé dans l'introduction à Tensorflow.

La conversion a fait l'objet d'une note séparée<sup>3</sup>.

```
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`.

La lecture des fichiers binaires nécessitera une ouverture en mode `rb`, ou sinon Python essaiera de lire le fichier en mode UTF-8, ce qui risque de poser un problème si on lit un caractère commençant par le bit 1.

```
6 def read_idx_ubyte (file):
7     file.seek (3)
8     depth = int.from_bytes (file.read (1), byteorder='big')
9     dims = []
10    n = 1
11    for i in range (depth):
12        d = int.from_bytes (file.read (4), byteorder='big')
13        n = n * d
14        dims.append (d)
15    all_bytes = file.read (n)
16    uints = np.frombuffer (all_bytes, dtype=np.uint8)
17    return np.reshape (uints, tuple (dims))
```

---

3. [[../shrec-to-mnist.org](https://shrec-to-mnist.org)]

### 2.2.4 Lecture d'étiquettes

On transforme les indices d'étiquettes en représentation *one-hot*. La représentation one-hot permet de transformer une modalité qualitative en un ensemble de booléens (quantitatifs).

Pour l'évaluation, nous aurons besoin de tester l'égalité des étiquettes. Ce sera possible en calculant le produit scalaire entre les représentations one-hot. Si le produit scalaire est proche de 1, il s'agit de la même catégorie.

```
18 def read_targets (file):
19     indices = read_idx_ubyte (file)
20     (n,) = np.shape (indices)
21     maxi = np.max (indices)
22     one_hot = np.zeros ((n, maxi + 1)).astype (np.float32)
23     one_hot[np.arange(n), indices] = 1
24     return one_hot
```

Application :

```
25 with open ("train-labels-idx1.ubyte", "rb") as f:
26     train_labels = read_targets (f)
27
28 with open ("test-labels-idx1.ubyte", "rb") as f:
29     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. C'est l'objet de la ligne 39.

```
30 def read_images (file):
31     pixels = read_idx_ubyte (file)
32     (n, v, h, w) = np.shape (pixels)
33     pixels_flt = pixels.astype (np.float32)
34     for i in range (n):
35         for j in range (v):
36             im = np.reshape (pixels_flt[i, j, :, :], (h * w))
37             im = (im - np.mean (im)) / np.std (im)
38             pixels_flt[i, j, :, :] = np.reshape (im, (1, 1, h, w))
39     return np.transpose (pixels_flt, (0, 2, 3, 1)) #
```

Application :

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

```

42
43 with open ("test-images-idx4.ubyte", "rb") as f:
44     test_images = read_images (f)

```

## 2.2.6 Première visualisation d'une image de test

Il est un peu abusif de parler d'une image, alors qu'en fait il s'agit d'un ensemble de vues. L'affichage concerne donc trois images en niveau de gris, ou avec une *colormap*.

```

45 plt.figure ()
46 (n_tests, h, w, n_views) = np.shape (test_images)
47 fig, axes = plt.subplots (1, n_views)
48 for i in range (n_views):
49     axes[i].imshow (np.reshape (test_images[1,:,:,i], (h, w)))
50     axes[i].set_title ('View number ' + str (i + 1))
51
52 plt.tight_layout()
53 plt.savefig('images/premier_test.png')
54 _ = 'images/premier_test.png'

```

Plus loin, nous effectuerons une requête sur cet objet.

## 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, de couches ReLu, de couches de max-pooling et de couches « fully-connected ».

### 2.3.1 Éléments du graphe

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

**Couche de convolution** Une couche de convolution (2D) permet d'apprendre un filtre de convolution à passer sur un tenseur d'entrée.

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.

```

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)

```

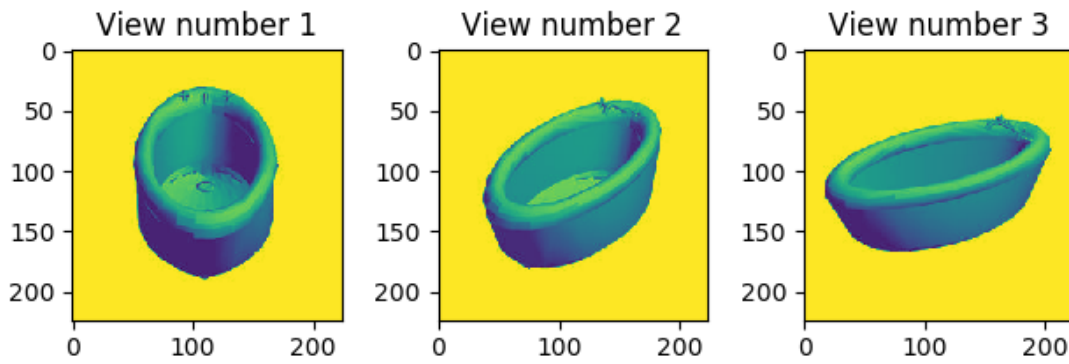


FIGURE 1 – Premier objet à tester

```

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. La partie aléatoire est volontairement choisie suffisamment faible pour que l'apprentissage puisse faire tendre les poids plus facilement.

**Couche ReLu** La couche ReLu (Rectified Linear Units) permet d'introduire une non-linéarité à la sortie d'une couche. 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_relu (input_data):
67     return tf.nn.relu (input_data)

```

**Couche de max-pooling** La couche de *max-pooling* permet de réduire la taille des données d'entrée.

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  $2 \times 2$  pixels.

**Couche « Fully connected »** La couche *fully connected* est une couche de réseau de neurones standard, c'est-à-dire sans convolution. Elle contient donc plus de poids à optimiser, c'est pourquoi on a tendance à la mettre en profondeur.

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. C'est un raccourci qui masque un descripteur (la sortie de la couche de convolution).

```
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_relu (conv_filter)
```



**Couche triconvrelu-pool** On enchaîne trois couches conv-relu puis une couche de max-pooling. Ce raccourci nous aidera pour construire des graphes complexes.

```

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.  
Le graphe que nous retiendrons est montré figure 2.

**Création des poids** On crée les poids pour le graphe. Ils sont stockés dans une arborescence de tuples, pour plus de clarté.

```

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 sur tous les poids 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
123     (w11, w12, w13) = w1
124     (w21, w22, w23) = w2
125     (w31, w32, w33) = w3
126     (w4w, w4b) = w4
127     (w11w, w11b) = w11
128     (w12w, w12b) = w12
129     (w13w, w13b) = w13
130     (w21w, w21b) = w21
131     (w22w, w22b) = w22
132     (w23w, w23b) = w23
133     (w31w, w31b) = w31
134     (w32w, w32b) = w32
135     (w33w, w33b) = w33
136     regularization = reg * \
137         (tf.nn.l2_loss (w4w) + tf.nn.l2_loss (w4b) \
138         + tf.nn.l2_loss (w11w) + tf.nn.l2_loss (w11b) \
139         + tf.nn.l2_loss (w12w) + tf.nn.l2_loss (w12b) \
140         + tf.nn.l2_loss (w13w) + tf.nn.l2_loss (w13b) \
141         + tf.nn.l2_loss (w21w) + tf.nn.l2_loss (w21b) \
142         + tf.nn.l2_loss (w22w) + tf.nn.l2_loss (w22b) \
143         + tf.nn.l2_loss (w23w) + tf.nn.l2_loss (w23b) \
144         + tf.nn.l2_loss (w31w) + tf.nn.l2_loss (w31b) \
145         + tf.nn.l2_loss (w32w) + tf.nn.l2_loss (w32b) \
146         + tf.nn.l2_loss (w33w) + tf.nn.l2_loss (w33b))
147     loss = tf.reduce_mean (loss_functions) + regularization
148     return ((conv1, conv2, conv3, prediction), loss, prediction, error)

```

## 2.4 Descente du gradient

Nous pouvons maintenant apprendre le réseau de neurones avec la méthode de la descente du gradient.

### 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.

```

149 n_steps = 3000
150 learning_rate = 0.5
151 loss_amount = np.zeros ((n_steps))
152 learned_weights = []

```

### 2.4.2 Apprentissage

Il s'agit du cœur de l'algorithme, c'est ici que tous les calculs sont faits. C'est principalement Tensorflow qui fait le travail.

Notons que la dernière couche « fully-connected » débouche sur un tenseur de largeur le nombre de catégories. Ce n'est absolument pas une obligation pour le requêtage, car on traite la sortie comme n'importe quel autre descripteur, mais on en a besoin pour effectuer l'apprentissage, puisque le but de l'apprentissage est de faire tendre ce descripteur vers la sortie attendue.

```
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)
197     e3 = (e31, e32, e33)
198     e4 = (w4w.eval (), w4b.eval ())
199     return (loss_amount, (e1, e2, e3, e4))
200
201 (loss_amount, learned_weights) = learn (n_steps, learning_rate)

```

Les paramètres du graphe sont des variables Tensorflow. Elles ne peuvent exister que dans le contexte de l'apprentissage, c'est pourquoi il faut les convertir en données constantes (grâce à la méthode `eval`) avant de sortir du contexte.

### 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`.

L'erreur commise mesure la différence entre l'application du graphe aux données d'apprentissage et la sortie attendue. Si elle vaut 0, alors le modèle classifie

```

202 fig, ax = plt.subplots ()
203 plt.plot (range (0, n_steps), loss_amount, c="g")
204 ax.set_ylabel ("Error")
205 ax.set_xlabel ("Number of steps")
206 fig.tight_layout()
207 plt.savefig('images/loss.png')
208 _ = '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 pourrions 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.

Les descripteurs sont la sortie de chaque bloc (3 convolution - Relu), et la sortie finale.

```

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

```

Les poids `weights` sont des poids à accorder à chaque descripteur. On pourrait les apprendre

### 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 4.

```
249 fig = plt.figure ()
250 (n_tests, h, w, n_views) = np.shape (test_images)
251 for i in range (3):
252     for j in range (n_views):
253         sp = fig.add_subplot (3, n_views, i * 3 + j + 1)
254         img = train_images[response[i], :, :, j]
255         img = np.reshape (img, (h, w))
256         imgplot = plt.imshow (img)
257         title = 'Choice ' + str (i + 1) + '/3: ' \
258             + str (response[i]) + ', view ' + str (j + 1)
```

```

259         sp.set_title (title)
260
261 fig.tight_layout ()
262 fig.savefig ('images/requete.png')
263 _ = 'images/requete.png'

```

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.

La précision, c'est le pourcentage d'éléments sélectionnés qui appartiennent à la classe réelle de l'objet requêté.

```

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

```

Le rappel, c'est le pourcentage d'éléments appartenant à la classe requêtée qui sont sélectionnés.

```

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

```

Pour calculer le score d'une matrice  $n_{\text{tests}} \times n_{\text{train}}$ , on fait la moyenne sur tous les tests, de la moyenne de l'aire sous la courbe pour tous les « cutoff ». Le « cutoff », c'est le nombre d'éléments sélectionnés dans la requête, c'est le 'k' du K-nearest-neighbor.

```

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)
300     avep = avep / ranks.shape[1]
301     map = map + avep
302     map = map / ranks.shape[0]
303     return map

```

Nous pouvons le tester avec les paramètres précédents :

```

304 n_test = test_labels.shape[0]
305 n_train = train_labels.shape[0]
306 ranks = np.zeros ((n_test, n_train)).astype (np.uint8)
307 for i in range (n_test):
308     X = np.array ([test_images[i, :, :, :]])
309     y = np.array ([test_labels[i]])
310     req = get_descriptors (X, y, learned_weights)
311     ranks[i, :] = rank (database, knn_weights, req)
312
313 the_map = score (ranks)

```

Le résultat est 42.70 %.

## 3 Conclusion

### 3.1 Résultat et limites

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, ce qui donne un résultat qui n'est pas comparable avec l'état de l'art.

Lors du passage de 2000 itérations à 3000 itérations, nous avons gagné environ 10 points de précision sur le score Mean Average Precision, il aurait donc été intéressant de pouvoir en effectuer plus. Cependant, le processus est très lent.

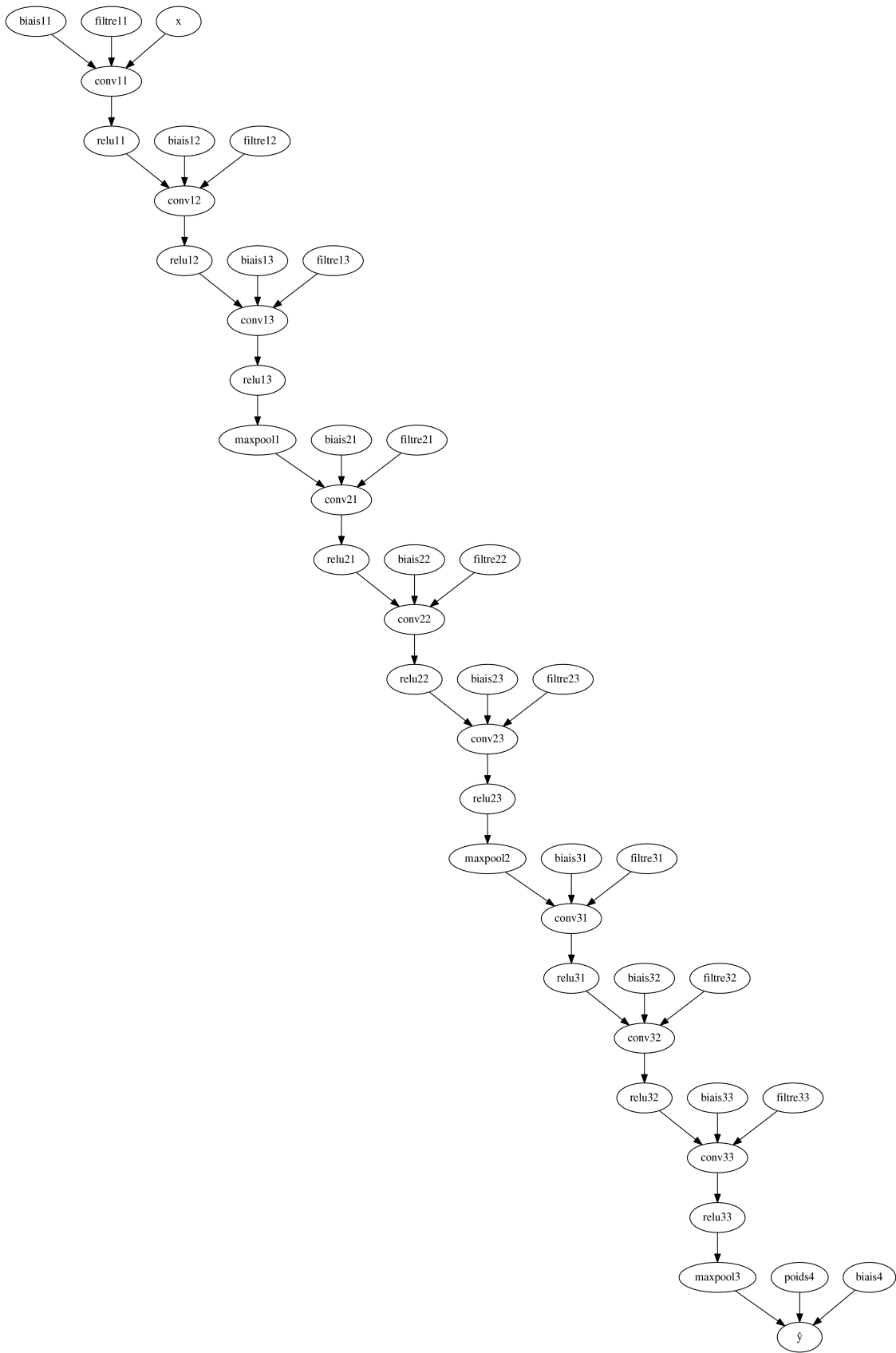
Nous n'avons pas été en mesure de prendre en compte un nombre suffisant d'individus d'apprentissage, et de vues, à cause de la quantité de mémoire vive limitée dont nous disposons.



## 3.2 Perspectives

Un calcul plus long ou sur une machine plus puissante pourrait nous permettre d'améliorer les résultats de notre étude.

La compétition SHREC cherche également à s'émanciper de l'orientation des objets. Pour ce faire, nous aurions pu conserver le modèle d'apprentissage et requêter chaque objet avec toutes les permutations cycliques de ses vues. Ainsi, nous aurions à réécrire la fonction de distance, qui prendrait cette fois un ensemble de descripteurs pour chaque apprentissage et un ensemble d'ensembles de descripteurs pour chaque test.



18  
FIGURE 2 – Graphe utilisé

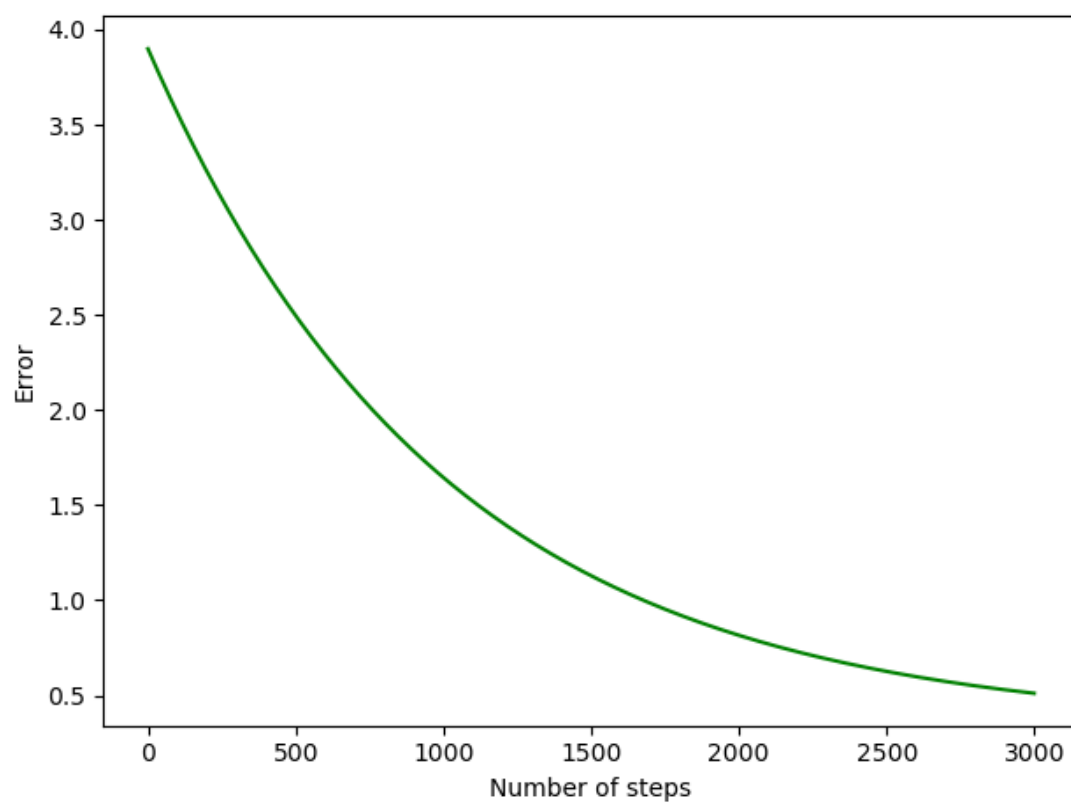


FIGURE 3 – Suivi de l'erreur commise à l'apprentissage

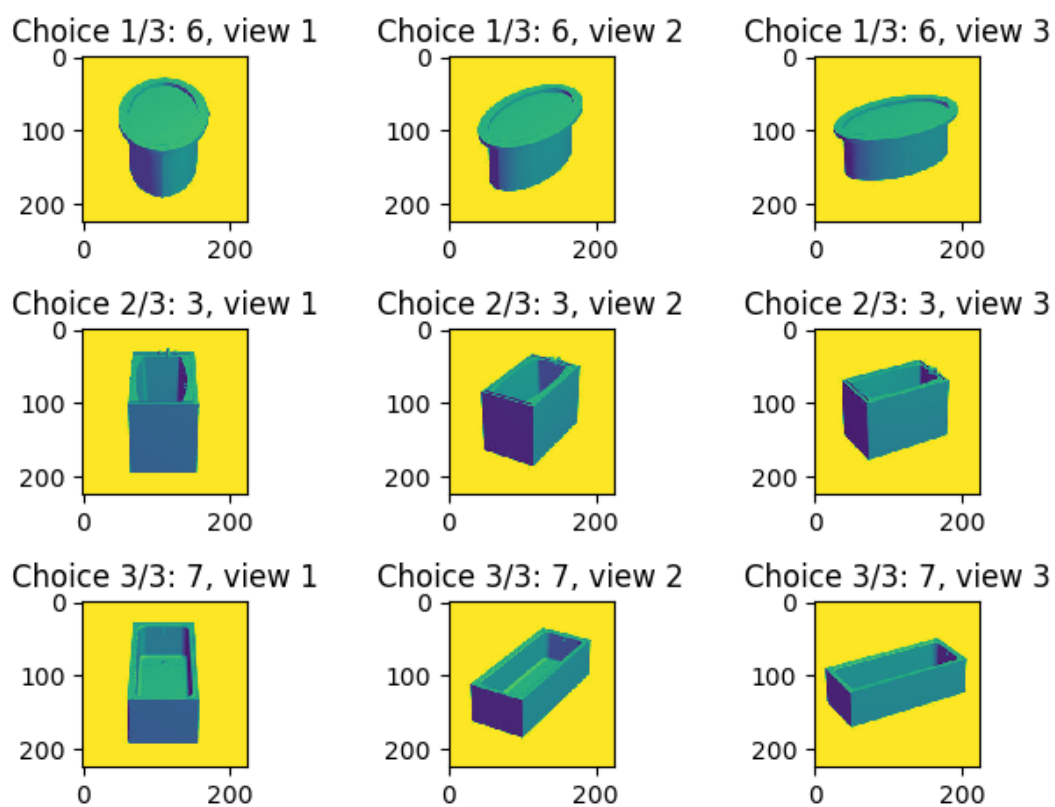


FIGURE 4 – Les trois meilleurs résultats dans l'ordre