

Projet Fouilles

Jérôme Skoda, Joaquim Lefranc



Arborescence du dossier

- **datasets** : Dossier contenant des archives de dataset
- **test_set** : Dossier contenant les images de test
- **training_set** : Dossier contenant les images de training
- **training-image-net-org** : Dossier contenant les urls des images à télécharger
- **clean_up.py** : Script permettant de supprimer les «mauvaises» images de image-net.org
- **data_augmentation.py** : Script permettant de créer des nouvelles variations des images de test
- **download_from_net.py** : Script permettant la récupération des images de image-net.org
- **generate-training-image-net-org.py** :
- **mkset_by_family.py** : Script permettant la création du set champignon par familles
- **mkset_by_toxicity.py** : Script permettant la création du set champignon par toxicité
- **training_split.py** : Permet de déplacer x% du **training_set/** dans **test_set/**
- **image_classifier.py** : Training du réseau de neurones



Utilisation des scripts

- **clean_up.py** : Ce script nécessite la création d'un dossier contenant les exemples d'images non souhaitées. Il se lance ensuite avec la commande suivante : **clean_up.py <dir> <dir_examples>**
- **data_augmentation.py** : Ce script augmente le dataset du dossier : **training_set/** (Attention ! Il faut prévoir quelques heures (10-12h) et 180Go de place libre)
- **download_from_net.py** : **download_from_net.py <dir_out> <first_indice> <urls_file>**
- **generate-training-image-net-org.py** : Télécharge les images contenues dans **training-image-net/** (très long)
- **mkset_by_family.py** : Pas d'arguments, il créer les deux dossiers **test_set/** et **training_set/**
- **mkset_by_toxicity.py** : Pas d'arguments, il créer les deux dossiers **test_set/** et **training_set/**
- **image_classifier.py** : Pas d'arguments, il lance directement le training du réseau.



Choix du projet et raisons

Nous avons choisis de travailler sur un classifieur d'images couleurs car nous trouvons cela plus excitant de travailler avec des images plutôt que du texte.

Au départ, nous avions en tête l'idée de classer des champignons en fonction de leur toxicité.

Cependant après plusieurs tests il s'avère que notre dataset est trop léger pour ce type de problème. En effet la toxicité des champignons n'est pas clairement visible sur toutes les espèces. De plus il faudrait avoir des machines avec une capacité de calcul monstrueuse, de façon à appliquer le réseau sur des images de tailles raisonnable dans le but d'analyser tout les détails permettant de distinguer précisément les caractères spécifique à la toxicité.

Nous avons donc bifurqués vers un autre problème : la classification d'espèces de champignon : même problème avec le faible nombre d'images. Alors nous avons choisis de nous intéresser aux espèces de fleurs. Ce problème de classification étant finalement assez général, le type d'images à classer importe peu. Par conséquent nous avons fait des tests avec différents datasets. Nous y reviendrons en détail plus loin dans ce rapport.



Environnement de développement

Les outils et frameworks

Nous utilisons TensorFlow pour créer notre réseau de neurones. Plusieurs scripts python sont disponibles dans le dossier pour traiter les données.

Paquets PIP nécessaires

- *beautifulsoup4*
- *opencv-python*
- *tensorflow*
- *unidecode*
- *python-magic*
- *glob2*

Les machines de test

M1 : i5 4670K @ 4.2Ghz - 24Go DDR3

M2 : i7-4790K @ 4.3Ghz - 32Go DDR3



Les datasets

Champignons (toxicité) :

Comme indiqué en préambule, nous avons commencés par la toxicité des champignons en récupérant les images du site suivant : <http://mycorance.free.fr/valchamp/toxiques.htm>

The screenshot shows a web page with a navigation bar at the top: "Les blancs", "Les bruns", "Jaunes / orangés", "Couleurs diverses", "Formes diverses", and "Sommaire". Below the navigation bar is a section titled "Index des champignons comestibles (moyens à excellents) et toxiques décrits dans le site".
Champignons comestibles:
Agaric à grandes spores
Agaric auguste
Agaric bulbeux
Agaric champêtre
Agaric des bois
Agaric des forêts
Agaric des sècheres
Agaric d'Espinette
Agaric Eminent
Agaric laqué
Agaric sylvestre
Agaric silvicole
Amanite des Césars
Amanite solitaire
Amanite vaginée [# voir fiche !](#)
Armillaire sans anneau
Auriculaire Oreille-de-Judas
Bidaou [# voir fiche !](#)
Bolette jaune d'oeuf
Bolet à chair jaune
Bolet à pied jaune

Champignons toxiques:
Agaric des robiniers
Agaric jaunissant
Agaric jaunissant var. gris
Agaric méléagre
Agaric méléagre var. couleur de terre
Agaric porphyre
Agaric rougeâtre
Amanite à pierrieries
Amanite à verrues
Amanite citrine
Amanite citrine var. blanche
Amanite élevée
Amanite fauve
Amanite jonquille
Amanite panthère
Amanite phalloïde **+++**
Amanite printanière **+++**
Amanite rougissante
Amanite safran
Amanite tue-mouches

Detailed description of the right panel:
Nom usuel : amanite tue-mouches
Chapeau : de 8 à 20 cm, d'abord globuleux, devenant convexe puis s'étalant, de couleur rouge à rouge orangé luisant couvert de verrues blanches puis jaunâtres, à marge lisse ou courtement cannelée selon les formes ou variétés
Lames : libres et serrées, inégales, de couleur blanche puis jaunissant légèrement
Anneau : ample, déchiqueté ou paraissant dentelé, persistant, de couleur blanche ou jaunâtre
Pied : d'abord court, massif et ferme, s'allongeant progressivement en devenant creux et pouvant mesurer jusqu'à 30 cm, pelucheux, renflé en bulbe orné de bourrelets concentriques à la base, de couleur blanche mais parfois jaune, surtout sous l'anneau et à la jonction avec le bulbe
Exhalaison : presque nulle
Période de cueillette : depuis la fin de l'été jusqu'à la fin de l'automne
Biotopes : lisière et bois de feuillus aérés (surtout les bouleaux) et de conifères (pins, épicéas, etc) mais aussi, plus rarement, sur chemins et herbiers remplaçant d'anciennes forêts
Confusions : possible avec l'[amanite des Césars](#)
Famille : amanitacées
Nom scientifique : amanita muscaria
Synonymes : fausse orange, tue-mouches

Avec un script python, nous avons pus télécharger toutes les images en leurs ajoutant un tag de toxicité. (`mkset_by_toxicity.py`)

Champignons (espèces) :

Avec ce même site, nous avons également récupérés toutes les images par espèce dans des dossiers différents grâce au script : `mkset_by_family.py`.



Les dataset par toxicité et par espèce ne sont pas exploitable car trop peu d'exemples par type dans le cas des espèces. Et trop peu de différences visuelles pour la toxicité. (Surtout sur des images de 64x64 ...)

Fleurs (espèces) :

Suite à de nouvelles recherches de dataset, nous avons trouvés le set de fleurs suivant : <http://www.robots.ox.ac.uk/~vgg/data/flowers/17/index.html>.

C'est un set contenant 17 espèces de fleurs avec 80 images par espèce. Il est de bonne qualité, cependant les images sont toutes en vracs dans le dossier... Nous avons fait manuellement le travail du réseau de neurone avec une fiabilité de 100% pour des non botanistes. C'est intéressant de voir la formidable capacité de reconnaissance des formes du cerveau humain. Les réseaux de neurones artificiels n'en sont pas encore là !

Accuracy : 75 %

Class Examples



C'est un set qui contient des exemples de bonne qualité. Avec le script de data augmentation, nous avons une bonne base de travail.

Autres (Fleurs, Cactus, Champignons) :

Pour diversifier nos tests nous voulions faire des essais sur des choses plus évidentes à classifier. En effet les différences entre fleurs, cactus et champignons sont bien marquées. De plus cela nous a permis de développer un script permettant de récupérer facilement des grosses quantités d'images du site image-net.org.

Accuracy : 92 %

IMAGENET

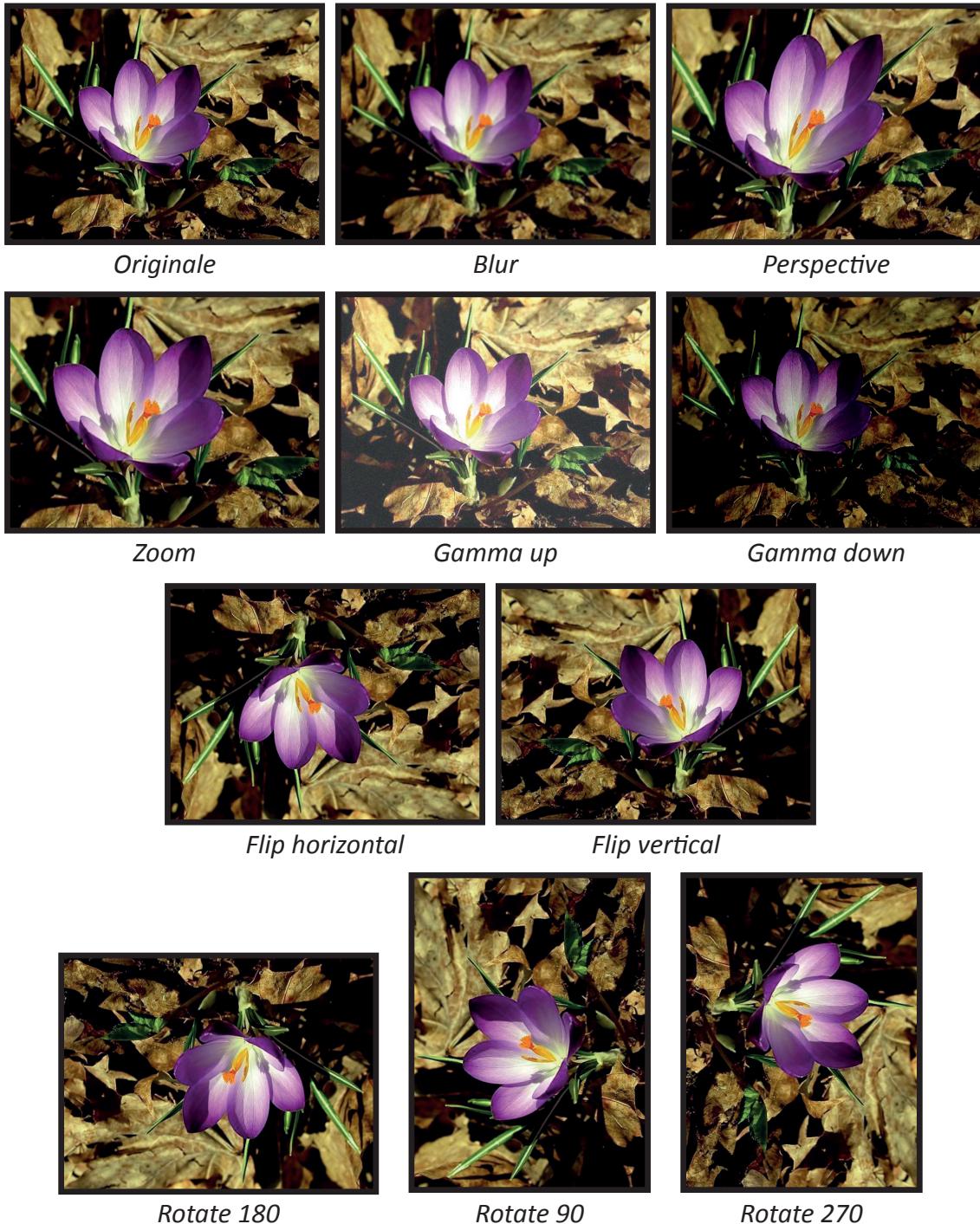




Data augmentation

L'augmentation artificielle est une des méthodes pour accroître le nombre d'exemples de training. Plusieurs opérations sont effectuées sur les images : flou, rotation, flipping, gamma etc. Ceci force le réseau de neurones à abstraire et ne pas se focaliser uniquement sur une seule position ou luminosité.

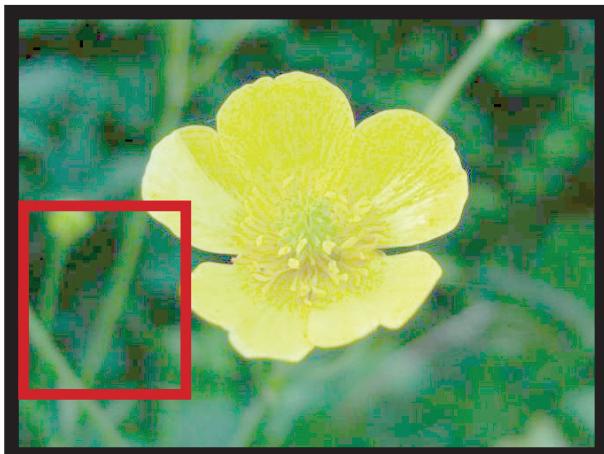
Nous avons donc conçus un script permettant d'automatiser cette création de variantes. Voici un exemple sur une image :



Il y a également toutes les variantes de blur et gamma sur chaque rotation et flip. Ce qui génère au total **864** images à partir de l'originale.



La première version de notre script permettant l'augmentation souffrait d'un problème sur l'augmentation du gamma. En effet notre méthode «grillait» les images. Ce qui nuisait à l'apprentissage du réseau.



Première version



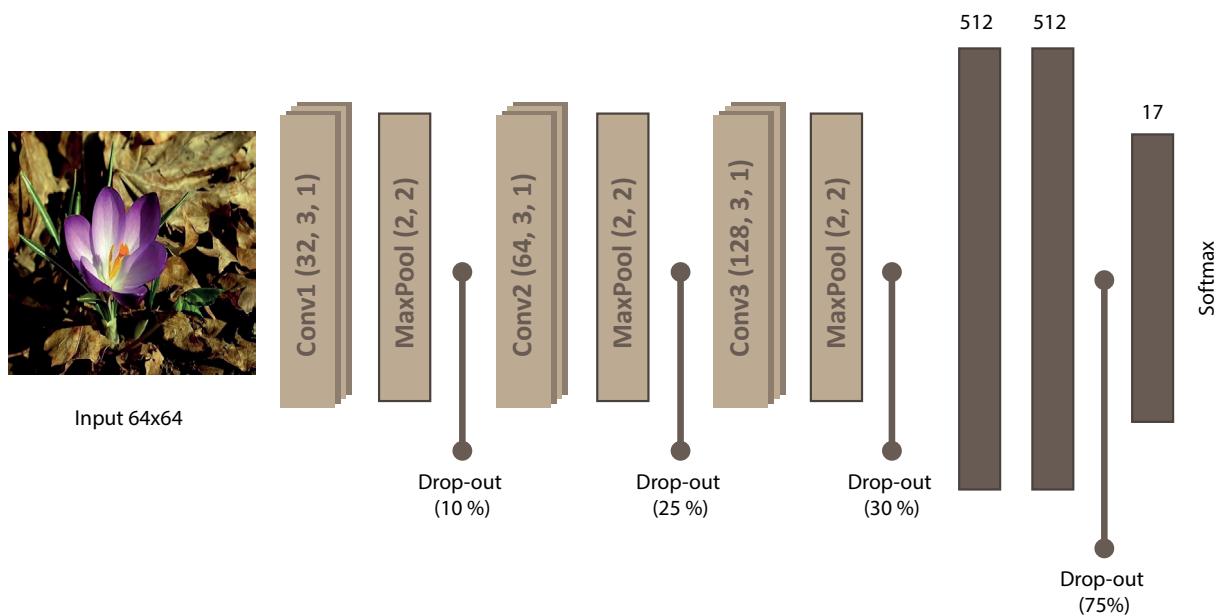
Deuxième version



Convolutional Neural Network Model

La classification d'images se fait avec un réseau convolutionnel, nous avons donc utilisés cette architecture en s'inspirant d'un code trouvé sur le web ([Liens dans image_classifier.py](#)). La modification et l'inspiration vient également d'autres models type AlexNet, VGG-16 et autres, quelques exemples sont disponibles à la fin du rapport.

Notre modèle :



Les différents paramètres :

Batch_size : Etant donné que nous avons des capacités en RAM limitées, un batch_size de 128 est un bon choix pour avoir la souplesse de tuning nécessaire tout en limitant le nombre de calculs par étapes. Cependant en contre partie l'optimizer à un peu plus de mal à trouver le minimum.

Optimizer : Adam Optimizer (Remplace le Gradient Descent) . Avec cet optimizer, le learning rate est automatiquement modifié en fonction de l'avancement de l'apprentissage.

Learning rate : C'est la vitesse d'apprentissage, ici 0.001 qui est la valeur standard pour l'AdamOptimizer.

Dropout : Ce paramètre permet d'éviter le sur-apprentissage (Overfitting), une valeur de 0.75 implique que les neurones des couches complètement connectées ont 75% de probabilité d'être mis à 0. Ceci force le réseau à créer de nouveaux chemins d'apprentissage. Nous avons un dropout de 75% sur la couche complètement connectée.

L2 Regularizer : C'est un paramètre de régularisation qui s'applique aux couches convolutives, 0.1 semble être le facteur conseillé. Nous appliquons cette régularisation sur toutes les couches convolutives.



Discussion

Commentaire sur nos résultats :

Soyons francs, nous sommes assez insatisfaits des résultats du réseau qui plafonnent entre 70-80% de réussite sur le set de validation contenant 17 classes avec un batch_size de 128. Malgré des centaines de tests différents, profondeur du réseau, largeur du réseau, simplification, complexification, changement de taille des filtres, changement du nombre de filtres, modifications des strides, du batch_size et tout les autres paramètres. Nous n'arrivons pas à passer cette barre des 75% en moyenne. Ce n'est en soit pas si mauvais, mais une telle débauche d'énergie et de temps pour ce résultat montre que effectivement il y a encore beaucoup de travail à faire dans ce domaine pour essayer de tirer des règles de conception.

Améliorations et idées pour accroître nos résultats :

Nous pensons que la limitation vient de la taille de nos images, en effet 64px de côté nous paraît légèrement faible pour déterminer des espèces dont les caractères sont parfois infimes. D'ailleurs durant nos recherches, la majorité des réseaux type AlexNet, VGG et autres travaillent sur des images de 224px, ce qui est déjà plus acceptable en terme de densité d'informations. Notre limitation aux images de 64px est déterminée par notre matériel de test à disposition (qui est pourtant pas si horrible, mais pour ce type d'algorithme c'est presque ridicule).

Avec plus de temps et matériel:

Une des choses principales que nous aurions aimés faire avec plus de temps : un script qui utilise le réseau déjà entraîné, qui prendrait une image en argument et qui retournerait la classe prédite avec la probabilité. Du côté matériel, avoir des GPU NVidia avec CUDA nous aurait permis de faire les calculs plus rapidement et donc de pouvoir essayer plusieurs autres tunning, et puis effectivement plus de ram aurait également été utile.



Equipe

Le découpage entre nous n'est pas strict, nous travaillons et échangeons très régulièrement, par conséquent nous pouvons grossièrement dire que tout les scripts concernant la récupération d'image-net viennent de Jérôme et que la data augmentation et la récupération des set de champignons du début de projet viennent de Joaquim. Pour ce qui est du main script image_classifier.py, nous nous sommes inspirés d'un script trouvé sur github (lien en début de fichier) que nous avons modifiés. Chaque un de nous testait des models différents sur le même script, histoire d'optimiser le temps car généralement il faut une nuit pour calculer 20000 steps...



Conclusion

Pour conclure, projet extrêmement intéressant, nous avons pu faire nos premiers pas dans le domaine des réseaux de neurones. C'est une discipline frustrante car s'apparentant souvent à la manche du magicien, cependant nous avons appris beaucoup de choses, les différents paramètres à prendre en compte et leurs influences.

Résumons ce que nous avons fait. L'idée de départ était la reconnaissance de la toxicité des champignons grâce à des images. Pour cela nous avons trouvé un site contenant un grand nombre d'espèces différentes avec leurs toxicités. Pour récupérer ces images nous avons mis en œuvre un script qui analyse des tags dans une page web pour récupérer les images et leurs tag de toxicité associée (présence d'une image «bonapp.png» ou «toxic.png»). Nous avons pu constater que le manque cruel d'exemples de ce set et une complexité du problème trop élevée ne nous permettrait pas d'avoir des résultats satisfaisants. Suite à cela nous avons essayés sur le même site de récupérer cette fois les champignons par espèce, mêmes problèmes que précédemment.

Après ces deux échecs, pour rester dans la botanique, nous sommes partis sur la reconnaissance d'espèces de fleurs. En effet les différences entre fleurs sont plus marquées, de plus nous avons trouvés le très connu set Oxford Flower 17. Ce sera donc notre base pour ce projet. Le set contenant uniquement 80 images par espèce, il fallait absolument de la data augmentation pour prévenir de l'overfitting. Pour cela nous avons utilisés la librairie open-cv en python pour transformer les images. Nous passons donc d'environ 2000 images à 800k exemplaires.

En parallèle nous avons créés des scripts pour récupérer de grosses quantités d'images de image-net.org, de façon à pouvoir tester nos algorithmes sur des choses plus différentes entre elles. Ceci dans le but de ne pas sombrer dans la folie, histoire de bien confirmer que c'est le problème de classification d'espèces qui est compliqué.

Bon et bien voilà, la suite est peu glorieuse, c'est du try, kill process and retry, encore et encore. Se lever le matin et observer qu'après 5-6h et 20000step, toujours en dessous de 80%... Mais bon, ça fait parti du folklore des réseaux de neurones faut croire !



Quelques résultats de différents models

Voici quelques screenshoot de quelques test de models que nous avons effectués, ils n'y sont pas tous heureusement.

```
65      # Input Layer
66      c1 = tf.layers.conv2d(x, 96, 5, strides=(3, 3), activation=tf.nn.relu, padding="SAME")
67      c1 = tf.layers.max_pooling2d(c1, 2, 2, padding="SAME")
68      #c1 = tf.nn.local_response_normalization(c1)
69
70      c2 = tf.layers.conv2d(c1, 192, 3, strides=(1, 1), activation=tf.nn.relu, padding="SAME")
71      c2 = tf.layers.max_pooling2d(c2, 2, 2, padding="SAME")
72      #c2 = tf.nn.local_response_normalization(c2)
73
74      #c3 = tf.layers.conv2d(c2, 384, 3, strides=(1, 1), activation=tf.nn.relu, padding="SAME")
75      #c3 = tf.layers.max_pooling2d(c3, 2, 2, padding="SAME")
76
77      fc1 = tf.contrib.layers.flatten(c2)
78      # Fully connecter layer
79      fc1 = tf.layers.dense(fc1, 1024, activation=tf.nn.relu)
80      fc1 = tf.layers.dropout(fc1, rate=dropout, training=is_training)
81
82      fc2 = tf.layers.dense(fc1, 512, activation=tf.nn.relu)
83      fc2 = tf.layers.dropout(fc2, rate=(dropout-0.25), training=is_training)
84
85      # Output layer
86      out = tf.layers.dense(fc2, n_classes)
```

```
-> Step 4700 : loss(0.92%) t_accuracy(100.00%) r_accuracy(76.56%) in 0.64 min
-> Step 4800 : loss(1.36%) t_accuracy(100.00%) r_accuracy(72.66%) in 0.64 min
-> Step 4900 : loss(1.40%) t_accuracy(100.00%) r_accuracy(67.97%) in 0.64 min
-> Step 5000 : loss(5.92%) t_accuracy(99.22%) r_accuracy(78.12%) in 0.64 min
-> Step 5100 : loss(1.30%) t_accuracy(100.00%) r_accuracy(71.88%) in 0.64 min
-> Step 5200 : loss(3.88%) t_accuracy(100.00%) r_accuracy(73.44%) in 0.65 min
-> Step 5300 : loss(3.19%) t_accuracy(100.00%) r_accuracy(67.19%) in 0.63 min
-> Step 5400 : loss(10.14%) t_accuracy(100.00%) r_accuracy(73.44%) in 0.62 min
-> Step 5500 : loss(2.64%) t_accuracy(100.00%) r_accuracy(71.88%) in 0.63 min
-> Step 5600 : loss(10.35%) t_accuracy(98.44%) r_accuracy(72.66%) in 0.65 min
-> Step 5700 : loss(3.40%) t_accuracy(99.22%) r_accuracy(72.66%) in 0.65 min
-> Step 5800 : loss(8.72%) t_accuracy(100.00%) r_accuracy(64.06%) in 0.65 min
-> Step 5900 : loss(0.31%) t_accuracy(100.00%) r_accuracy(75.78%) in 0.63 min
```

```

60     display_step = 100
61
62     def convolutional_network(x, n_classes, dropout, reuse, is_training):
63         with tf.variable_scope('ConvNet', reuse=reuse):
64
65             regularizer = tf.contrib.layers.l2_regularizer(scale=0.1);
66
67             # Input Layer
68             c1 = tf.layers.conv2d(x, 64, 3, strides=(1,1), activation=tf.nn.relu, kernel_regularizer=regularizer)
69             c1 = tf.layers.conv2d(c1, 64, 3, strides=(1,1), activation=tf.nn.relu, kernel_regularizer=regularizer)
70             c1 = tf.layers.max_pooling2d(c1, 2, 2)
71             c1 = tf.layers.dropout(c1, rate= 0.1, training=is_training)
72
73
74             c2 = tf.layers.conv2d(c1, 128, 3, strides=(1, 1), activation=tf.nn.relu, kernel_regularizer=regularizer)
75             c2 = tf.layers.conv2d(c2, 128, 3, strides=(1, 1), activation=tf.nn.relu, kernel_regularizer=regularizer)
76             c2 = tf.layers.max_pooling2d(c2, 2, 2)
77             c2 = tf.layers.dropout(c2, rate= 0.25, training=is_training)
78
79
80             c3 = tf.layers.conv2d(c2, 256, 3, strides=(1, 1), activation=tf.nn.relu, kernel_regularizer=regularizer)
81             c3 = tf.layers.conv2d(c3, 256, 3, strides=(1, 1), activation=tf.nn.relu, kernel_regularizer=regularizer)
82             c3 = tf.layers.max_pooling2d(c3, 2, 2)
83             c3 = tf.layers.dropout(c3, rate= 0.25, training=is_training)
84
85
86             #c4 = tf.layers.conv2d(c3, 512, 3, strides=(1, 1), activation=tf.nn.relu, kernel_regularizer=regularizer)
87             #c4 = tf.layers.max_pooling2d(c4, 2, 2)
88             #c4 = tf.layers.dropout(c4, rate= 0.3, training=is_training)
89

```

```

> Step 11800 : loss(31.82%) t_accuracy(96.88%) r_accuracy(62.50%) delta(34.38) time(1.68 min)
-> Step 11900 : loss(13.31%) t_accuracy(96.88%) r_accuracy(65.62%) delta(31.25) time(1.68 min)
-> Step 12000 : loss(32.14%) t_accuracy(93.75%) r_accuracy(75.00%) delta(18.75) time(1.68 min)
-> Step 12100 : loss(25.41%) t_accuracy(93.75%) r_accuracy(78.12%) delta(15.62) time(1.66 min)
-> Step 12200 : loss(26.74%) t_accuracy(93.75%) r_accuracy(75.00%) delta(18.75) time(1.68 min)
-> Step 12300 : loss(9.07%) t_accuracy(100.00%) r_accuracy(71.88%) delta(28.12) time(1.68 min)
-> Step 12400 : loss(28.17%) t_accuracy(93.75%) r_accuracy(68.75%) delta(25.00) time(1.68 min)
-> Step 12500 : loss(21.59%) t_accuracy(93.75%) r_accuracy(68.75%) delta(25.00) time(1.65 min)
-> Step 12600 : loss(34.40%) t_accuracy(93.75%) r_accuracy(87.50%) delta(6.25) time(1.68 min)
-> Step 12700 : loss(33.89%) t_accuracy(96.88%) r_accuracy(59.38%) delta(37.50) time(1.68 min)
-> Step 12800 : loss(9.39%) t_accuracy(100.00%) r_accuracy(75.00%) delta(25.00) time(1.70 min)
-> Step 12900 : loss(23.79%) t_accuracy(100.00%) r_accuracy(71.88%) delta(28.12) time(1.65 min)

```

```

image_classifier.py
55     regularizer = tf.contrib.layers.l2_regularizer(scale=0.1);
56
57
58     # Input Layer
59     c1 = tf.layers.conv2d(x, 32, 5, strides=(1,1), activation=tf.nn.relu, kernel_regularizer=regularizer)
60     #c1 = tf.layers.conv2d(c1, 64, 3, strides=(1,1), activation=tf.nn.relu, kernel_regularizer=regularizer)
61     c1 = tf.layers.max_pooling2d(c1, 2, 2)
62     c1 = tf.layers.dropout(c1, rate= 0.1, training=is_training)
63
64
65     c2 = tf.layers.conv2d(c1, 64, 5, strides=(1, 1), activation=tf.nn.relu, kernel_regularizer=regularizer)
66     #c2 = tf.layers.conv2d(c2, 64, 3, strides=(1, 1), activation=tf.nn.relu, kernel_regularizer=regularizer)
67     #c2 = tf.layers.conv2d(c2, 64, 3, strides=(1, 1), activation=tf.nn.relu, kernel_regularizer=regularizer)
68     c2 = tf.layers.max_pooling2d(c2, 2, 2)
69     c2 = tf.layers.dropout(c2, rate= 0.25, training=is_training)
70
71
72     c3 = tf.layers.conv2d(c2, 128, 3, strides=(1, 1), activation=tf.nn.relu, kernel_regularizer=regularizer)
73     #c3 = tf.layers.conv2d(c3, 256, 3, strides=(1, 1), activation=tf.nn.relu, kernel_regularizer=regularizer)
74     #c3 = tf.layers.conv2d(c3, 256, 3, strides=(1, 1), activation=tf.nn.relu, kernel_regularizer=regularizer)
75     c3 = tf.layers.max_pooling2d(c3, 2, 2)
76     c3 = tf.layers.dropout(c3, rate= 0.25, training=is_training)
77
78
79
80     #c4 = tf.layers.conv2d(c3, 512, 3, strides=(1, 1), activation=tf.nn.relu, kernel_regularizer=regularizer)
81     #c4 = tf.layers.max_pooling2d(c4, 2, 2)
82     #c4 = tf.layers.dropout(c4, rate= 0.3, training=is_training)
83
84
85     fc1 = tf.contrib.layers.flatten(c3)
86     # Fully connected layer
87     fc1 = tf.layers.dense(fc1, 1024, activation=tf.nn.relu)
88     fc1 = tf.layers.dropout(fc1, rate=0.75, training=is_training)
89
90
91
92
93
94
95

```

```

> Step 19000 : loss(10.38%) t_accuracy(99.22%) r_accuracy(75.00%) delta(24.22) time(2.00 min)
-> Step 19100 : loss(6.88%) t_accuracy(98.44%) r_accuracy(72.66%) delta(25.78) time(2.01 min)
-> Step 19200 : loss(20.32%) t_accuracy(97.66%) r_accuracy(70.31%) delta(27.34) time(2.00 min)
-> Step 19300 : loss(16.52%) t_accuracy(98.44%) r_accuracy(72.66%) delta(25.78) time(2.05 min)
-> Step 19400 : loss(7.87%) t_accuracy(99.22%) r_accuracy(73.44%) delta(25.78) time(2.00 min)
-> Step 19500 : loss(6.72%) t_accuracy(99.22%) r_accuracy(76.56%) delta(22.66) time(2.02 min)
-> Step 19600 : loss(19.73%) t_accuracy(98.44%) r_accuracy(71.09%) delta(27.34) time(2.01 min)
-> Step 19700 : loss(7.60%) t_accuracy(100.00%) r_accuracy(75.00%) delta(25.00) time(2.02 min)
-> Step 19800 : loss(9.44%) t_accuracy(100.00%) r_accuracy(71.09%) delta(28.91) time(2.03 min)
-> Step 19900 : loss(10.12%) t_accuracy(99.22%) r_accuracy(79.69%) delta(19.53) time(2.00 min)
-> Step 20000 : loss(6.30%) t_accuracy(100.00%) r_accuracy(78.12%) delta(21.88) time(2.04 min)

```

```

  ⚡ image_classifier.py
  65   regularizer = tf.contrib.layers.l2_regularizer(scale=0.1);
  66
  67   # Input Layer
  68   c1 = tf.layers.conv2d(x, 32, 3, strides=(1,1), activation=tf.nn.relu, kernel_regularizer=regularizer)
  69   #c1 = tf.layers.conv2d(c1, 32, 3, strides=(1,1), activation=tf.nn.relu, kernel_regularizer=regularizer)
  70   c1 = tf.layers.max_pooling2d(c1, 2, 2)
  71   c1 = tf.layers.dropout(c1, rate= 0.1, training=is_training)
  72
  73   c2 = tf.layers.conv2d(c1, 64, 3, strides=(1, 1), activation=tf.nn.relu, kernel_regularizer=regularizer)
  74   #c2 = tf.layers.conv2d(c2, 64, 3, strides=(1, 1), activation=tf.nn.relu, kernel_regularizer=regularizer)
  75   #c2 = tf.layers.conv2d(c2, 64, 3, strides=(1, 1), activation=tf.nn.relu, kernel_regularizer=regularizer)
  76   c2 = tf.layers.max_pooling2d(c2, 2, 2)
  77   c2 = tf.layers.dropout(c2, rate= 0.25, training=is_training)
  78
  79   c3 = tf.layers.conv2d(c2, 128, 3, strides=(1, 1), activation=tf.nn.relu, kernel_regularizer=regularizer)
  80   #c3 = tf.layers.conv2d(c3, 128, 3, strides=(1, 1), activation=tf.nn.relu, kernel_regularizer=regularizer)
  81   #c3 = tf.layers.conv2d(c3, 128, 3, strides=(1, 1), activation=tf.nn.relu, kernel_regularizer=regularizer)
  82   c3 = tf.layers.max_pooling2d(c3, 2, 2)
  83   c3 = tf.layers.dropout(c3, rate= 0.25, training=is_training)
  84
  85   #c4 = tf.layers.conv2d(c3, 512, 3, strides=(1, 1), activation=tf.nn.relu, kernel_regularizer=regularizer)
  86   #c4 = tf.layers.max_pooling2d(c4, 2, 2)
  87   #c4 = tf.layers.dropout(c4, rate= 0.3, training=is_training)
  88
  89   fc1 = tf.contrib.layers.flatten(c3)
  90   # Fully connected layer
  91   fc1 = tf.layers.dense(fc1, 1024, activation=tf.nn.relu)
  92   fc1 = tf.layers.dropout(fc1, rate=0.75, training=is_training)
  93
  94   fc2 = tf.layers.dense(fc1, 512, activation=tf.nn.relu)
  95   fc2 = tf.layers.dropout(fc2, rate=0.75, training=is_training)

  ⚡
  -> Step 7000 : loss(16.09%) t_accuracy(99.22%) r_accuracy(77.34%) delta(21.88) time(2.15 min)
  -> Step 7100 : loss(28.26%) t_accuracy(100.00%) r_accuracy(72.66%) delta(27.34) time(2.13 min)
  -> Step 7200 : loss(26.33%) t_accuracy(100.00%) r_accuracy(75.78%) delta(24.22) time(2.12 min)
  -> Step 7300 : loss(23.18%) t_accuracy(96.88%) r_accuracy(75.78%) delta(21.09) time(2.18 min)
  -> Step 7400 : loss(19.66%) t_accuracy(98.44%) r_accuracy(71.88%) delta(26.56) time(2.12 min)
  -> Step 7500 : loss(35.30%) t_accuracy(97.66%) r_accuracy(82.81%) delta(14.84) time(2.09 min)
  -> Step 7600 : loss(25.29%) t_accuracy(98.44%) r_accuracy(71.09%) delta(27.34) time(2.12 min)
  -> Step 7700 : loss(17.56%) t_accuracy(99.22%) r_accuracy(76.56%) delta(22.66) time(2.13 min)
  -> Step 7800 : loss(16.47%) t_accuracy(100.00%) r_accuracy(74.22%) delta(25.78) time(2.18 min)
  -> Step 7900 : loss(16.78%) t_accuracy(100.00%) r_accuracy(74.22%) delta(25.78) time(2.13 min)
  -> Step 8000 : loss(14.31%) t_accuracy(100.00%) r_accuracy(74.22%) delta(25.78) time(2.23 min)
  -> Step 8100 : loss(13.79%) t_accuracy(99.22%) r_accuracy(72.66%) delta(26.56) time(2.13 min)

```

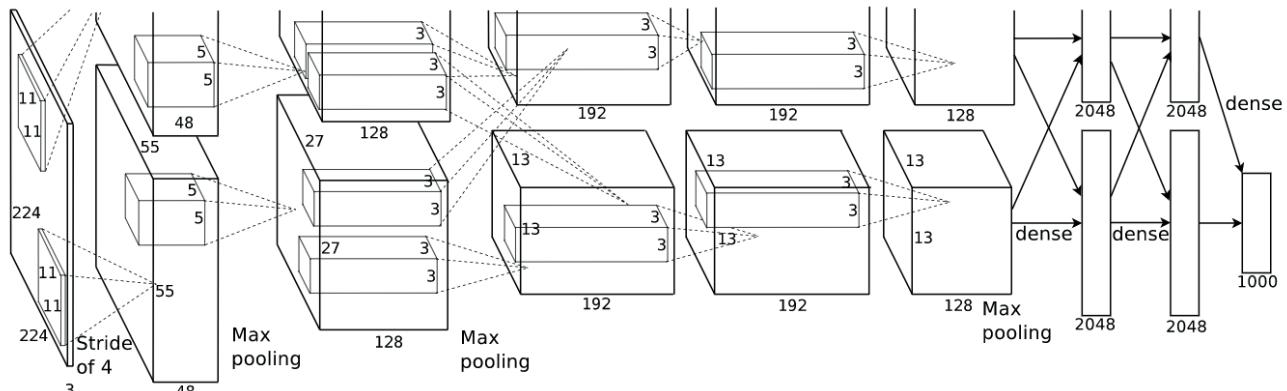
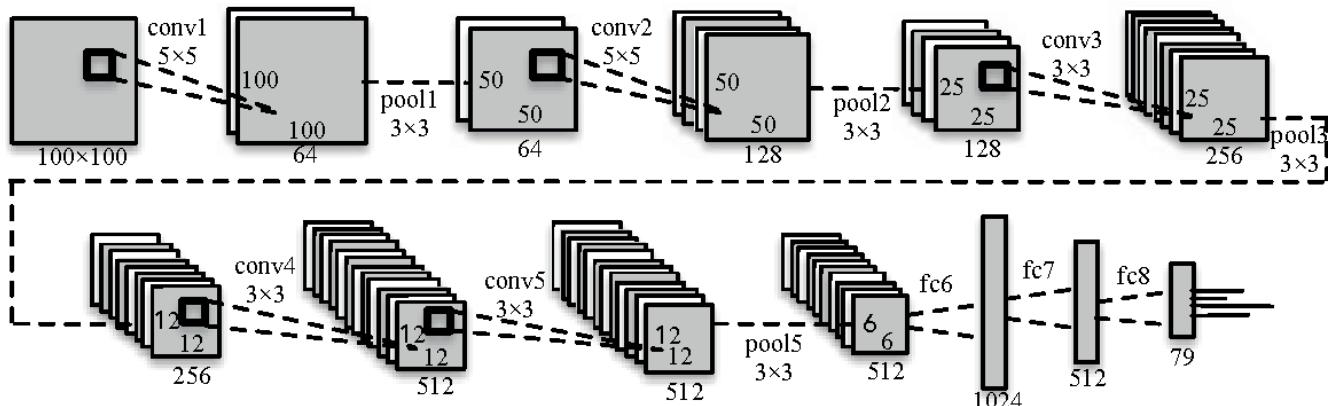


Documentation :

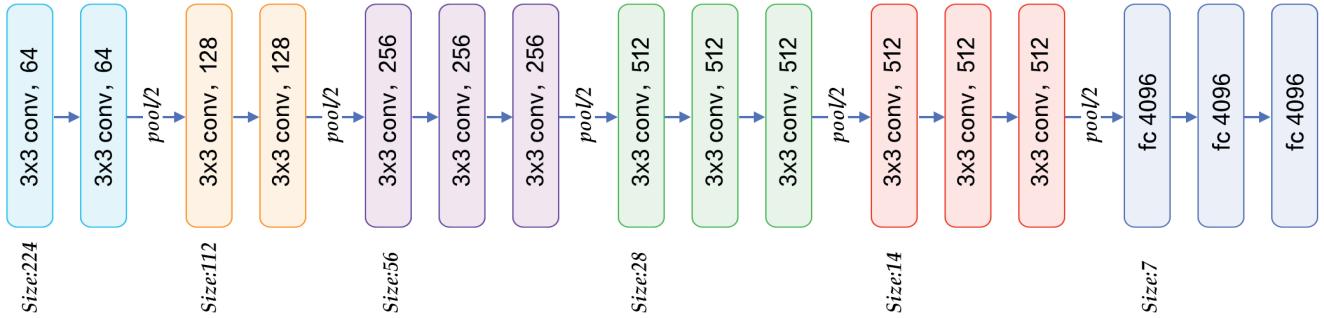
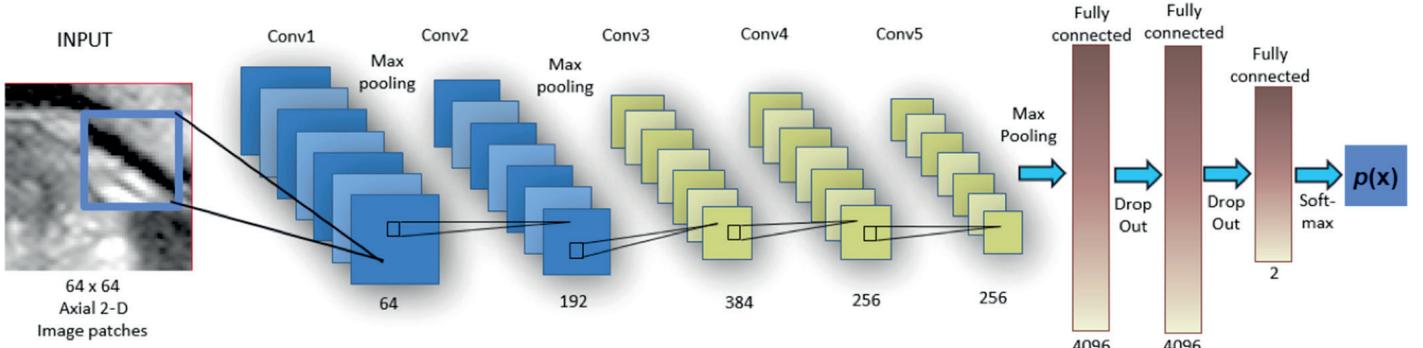
- <https://medium.com/ymedialabs-innovation/data-augmentation-techniques-in-cnn-using-tensorflow-371ae43d5be9>
- <http://www.sai-tai.com/blog/image-classification/>
- <http://ataspinar.com/2017/08/15/building-convolutional-neural-networks-with-tensorflow/>
- <https://blog.slavv.com/37-reasons-why-your-neural-network-is-not-working-4020854bd607>
- <http://mourafiq.com/2016/08/10/playing-with-convolutions-in-tensorflow.html>
- <https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/>
- <http://cs231n.github.io/convolutional-networks/#conv>

Models pour l'inspiration :

Voici quelques exemples de models qui nous ont servis d'inspiration pour modifier, améliorer le notre.



AlexNet



VGG-16

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					