Yann COMOR

Explication de l'algorithme de clustering K-means appliqué aux images - DS50

Ce document vise à fournir un aperçu complet de l'algorithme de clustering K-means appliqué (à la date du 09/04/25) aux images afin de développer une méthode d'échantillonnage limitant le biais de la méthode d'échantillonnage.

Présentation du pipeline de sélection d'images par clustering

Le code proposé met en œuvre un pipeline de traitement d'images visant à extraire un sous-ensemble diversifié d'un dataset, en s'appuyant sur des embeddings visuels obtenus à partir d'un réseau convolutif pré-entraîné. Ce pipeline repose sur les étapes suivantes :

---

1. Chargement du modèle et préparation des images

```
resnet = models.resnet18(pretrained=True)
resnet.fc = torch.nn.Identity()
resnet = resnet.to(DEVICE).eval()
```

Le modèle utilisé est **ResNet18**, un réseau convolutif de référence, pré-entraîné sur ImageNet. La dernière couche (`fc`) est remplacée par une identité, ce qui permet de récupérer directement le **vecteur d'embedding de dimension 512** produit par le réseau, sans effectuer la classification.

```
preprocess = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225]),
])
```

Les transformations appliquées sont **standardisées pour ResNet**, avec redimensionnement à `224x224` pixels (format d'entrée du modèle) et normalisation selon les statistiques d'ImageNet. (tiré de la documentation de torchvision).

---

2. Extraction des embeddings

```
def extract_embedding(image_path):
    ...
    embedding = resnet(img_tensor).squeeze().cpu().numpy()
```

Cette fonction transforme chaque image en un **vecteur de 512 réels**, représentant la structure visuelle de l'image. Ces vecteurs servent ensuite de base pour le regroupement via clustering.

---

3. Détermination automatique du nombre de clusters

```
def find_optimal_k(embeddings, k_range=range(5, 31)):
    ...
    score = silhouette_score(embeddings, kmeans.labels_)
```

L'algorithme de clustering utilisé est **KMeans**. Plutôt que de fixer arbitrairement le nombre de clusters, la fonction explore plusieurs valeurs (`k` de 5 à 30 par défaut), et sélectionne celle qui maximise le **score de silhouette**, une métrique mesurant la cohérence interne des clusters. (voir ici).

Ce choix garantit une **partition adaptée à la distribution des embeddings** pour chaque sous-dossier.

---

4. Sélection d'un sous-ensemble diversifié

```
def sample_by_auto_clustering(embeddings, paths, total_samples=1000):
    ...
    selected_paths.append(cluster_paths[idx])
```

Pour chaque cluster, on sélectionne un nombre fixe d'images (`samples_per_cluster`), en choisissant les plus proches du **centre du cluster**. Cette approche garantit que :

- Chaque groupe visuel est représenté,
- Les images choisies sont **représentatives** de leur groupe.

Le nombre total d'images à conserver est défini par `total_samples`, typiquement `1000` par sous-dossier.

---

5. Application par sous-dossier

```
def diverse_sample_per_subdirectory(subdirs, ...):
    ...
    total_samples = min(TOTAL_SAMPLES_PER_FOLDER, len(valid_paths))
```

Le pipeline est appliqué **à un ensemble de sous-dossiers** (par exemple, des classes de maladies dans notre cas), chacun étant traité de façon indépendante. Cela permet de garantir une diversité **au sein de chaque classe**, tout en maintenant un équilibre global dans l'échantillonnage.

Les paramètres de base tels que :

- `TOTAL_SAMPLES_PER_FOLDER` : nombre maximum d'images sélectionnées par sous-dossier,

- `IMAGE_EXTENSIONS` : types d'images considérées (`.jpg`, `.jpeg`, `.png`), sont facilement ajustables pour s'adapter à la taille du jeu de données et aux contraintes d'expérimentation. (dans une idée de réutilisation du code)

Finalité du pipeline

L'objectif du code est de créer, pour chaque catégorie, un **échantillon réduit mais varié**, représentatif du contenu original. Cette réduction permet :

- D'accélérer les expérimentations (entraînement, évaluation),
- De préserver une diversité visuelle essentielle pour l'apprentissage automatique,
- D'exploiter efficacement la structure sémantique extraite par le modèle de vision.

Questions annexes

Pourquoi 512 dimensions ?

L'architecture de ResNet18 est construite de façon hiérarchique : chaque bloc de convolution extrait des motifs de plus en plus complexes dans l'image. La dernière couche produit un tenseur de caractéristiques de taille (512,), ce qui représente une synthèse globale du contenu visuel de l'image. (voir ici pour plus de détails sur l'architecture). On peut voir sur ce document au début l'architecture, et on voit que la dernière couche est de 512 dimensions. (il y a la couche de classification après `fc`, mais on la supprime ici pour *finetuner* notre modèle).

Ce vecteur de 512 dimensions encode des informations visuelles telles que :

- La présence de bords, de textures, ou de motifs spécifiques,
- Des formes globales ou locales,
- Des indices de couleurs ou de contrastes,
- Des éléments structurants utiles pour distinguer des objets.

Ces dimensions ne sont pas directement interprétables individuellement, mais leur combinaison permet au réseau de capturer des représentations sémantiques complexes. Ces embeddings sont appris sur le dataset ImageNet, ce qui leur confère une capacité de généralisation importante sur des images naturelles.

Code source

```
import os
import shutil
import torch
import numpy as np
from tqdm import tqdm
from PIL import Image

from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from torchvision import models, transforms

DATA_FOLDER_PATH = "../"
```

```
INPUT_DIR = DATA_FOLDER_PATH + "00_archive/data/"
OUTPUT_DIR = DATA_FOLDER_PATH + "00_archive/data_samples/"

SAMPLES_PER_CLASS = 1000  # arbitrary number of samples per class

file_types = ["train", "test", "val"]
subdirectories = ["Coccidiosis", "Healthy", "New Castle Disease", "Salmonella"]


print(torch.cuda.is_available())
print(torch.cuda.get_device_name(0))

# Parameters
IMAGE_SIZE = 224
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using device: {DEVICE}")

# Load Pretrained ResNet (remove final classification layer)
resnet = models.resnet18(pretrained=True)
resnet.fc = torch.nn.Identity()
resnet = resnet.to(DEVICE).eval()

# Preprocessing
preprocess = transforms.Compose([
    transforms.Resize((IMAGE_SIZE, IMAGE_SIZE)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225]),
])

def extract_embedding(image_path):
    try:
        img = Image.open(image_path).convert("RGB")
        img_tensor = preprocess(img).unsqueeze(0).to(DEVICE)
        with torch.no_grad():
            embedding = resnet(img_tensor).squeeze().cpu().numpy()
        return embedding
    except Exception as e:
        print(f"Error processing {image_path}: {e}")
        return None

def find_optimal_k(embeddings, k_range=range(5, 21)):  # Default range for k
    max_possible_k = min(len(embeddings), max(k_range))
    best_k = 5
    best_score = -1

    for k in range(5, max_possible_k + 1):
        kmeans = KMeans(n_clusters=k, random_state=42).fit(embeddings)
        score = silhouette_score(embeddings, kmeans.labels_)
        if score > best_score:
            best_k = k
            best_score = score
```

```
    print(f"Optimal k found: {best_k} with silhouette score: {best_score:.4f}")
    return best_k

def sample_by_auto_clustering(embeddings, paths, total_samples=300):
    k = find_optimal_k(embeddings)
    k = min(k, len(embeddings))  # Ensure k is not greater than the number of
samples
    kmeans = KMeans(n_clusters=k, random_state=42).fit(embeddings)
    labels = kmeans.labels_

    samples_per_cluster = total_samples // k
    selected_paths = []

    for i in range(k):
        indices = [j for j, label in enumerate(labels) if label == i]
        cluster_embeddings = [embeddings[j] for j in indices]
        cluster_paths = [paths[j] for j in indices]

        center = kmeans.cluster_centers_[i]
        dists = np.linalg.norm(np.array(cluster_embeddings) - center, axis=1)
        sorted_indices = np.argsort(dists)

        max_samples = min(samples_per_cluster, len(cluster_paths))

        for idx in sorted_indices[:max_samples]:
            selected_paths.append(cluster_paths[idx])

    return selected_paths

# For all subdirectories, sample a fixed number of images per class
def diverse_sample_per_subdirectory_all(verbose=False):
    input_root = INPUT_DIR
    output_root = OUTPUT_DIR
    TOTAL_SAMPLES_PER_FOLDER = SAMPLES_PER_CLASS  # You can adjust this per folder
    IMAGE_EXTENSIONS = ('.jpg', '.jpeg', '.png')

    os.makedirs(output_root, exist_ok=True)
    print("Scanning dataset structure...")

    file_types = [d for d in os.listdir(input_root) if
os.path.isdir(os.path.join(input_root, d))]

    for file_type in file_types:
        file_type_path = os.path.join(input_root, file_type)
        subdirectories = [d for d in os.listdir(file_type_path) if
os.path.isdir(os.path.join(file_type_path, d))]

        for subdirectory in subdirectories:
            input_path = os.path.join(file_type_path, subdirectory)
            output_path = os.path.join(output_root, file_type, subdirectory)
            os.makedirs(output_path, exist_ok=True)

            images = [f for f in os.listdir(input_path) if
```

```
f.lower().endswith(IMAGE_EXTENSIONS)]
            image_paths = [os.path.join(input_path, f) for f in images]

            embeddings = []
            valid_paths = []

            for path in tqdm(image_paths, desc=f"🔍 {file_type}/{subdirectory}",
leave=False):
                emb = extract_embedding(path)
                if emb is not None:
                    embeddings.append(emb)
                    valid_paths.append(path)

            if len(valid_paths) == 0:
                print(f"No valid images in {file_type}/{subdirectory}")
                continue

            print(f"{file_type}/{subdirectory}: {len(valid_paths)} images,
extracting clusters...")

            selected_paths = sample_by_auto_clustering(
                embeddings,
                valid_paths,
                total_samples=min(TOTAL_SAMPLES_PER_FOLDER, len(valid_paths))
            )

            for src in selected_paths:
                dst = os.path.join(output_path, os.path.basename(src))
                shutil.copy2(src, dst)

            if verbose:
                print(f"Copied {len(selected_paths)} images to {output_path}")

    print("Sampling complete for all subdirectories.")


def diverse_sample_per_subdirectory(subdirs, verbose=False):
    input_root = INPUT_DIR
    output_root = OUTPUT_DIR
    TOTAL_SAMPLES_PER_FOLDER = SAMPLES_PER_CLASS
    IMAGE_EXTENSIONS = ('.jpg', '.jpeg', '.png')

    os.makedirs(output_root, exist_ok=True)
    print("Sampling from specific subdirectories...")

    for relative_subdir in subdirs:
        input_path = os.path.join(input_root, relative_subdir)
        output_path = os.path.join(output_root, relative_subdir)
        os.makedirs(output_path, exist_ok=True)

        if not os.path.isdir(input_path):
            print(f"Skipping non-directory: {input_path}")
            continue
```

```
        images = [f for f in os.listdir(input_path) if
f.lower().endswith(IMAGE_EXTENSIONS)]
        image_paths = [os.path.join(input_path, f) for f in images]

        embeddings = []
        valid_paths = []

        for path in tqdm(image_paths, desc=f"{relative_subdir}", leave=False):
            emb = extract_embedding(path)
            if emb is not None:
                embeddings.append(emb)
                valid_paths.append(path)

        if len(valid_paths) == 0:
            print(f"No valid images in {relative_subdir}")
            continue

        print(f"{relative_subdir}: {len(valid_paths)} images, extracting
clusters...")

        selected_paths = sample_by_auto_clustering(
            embeddings,
            valid_paths,
            total_samples=min(TOTAL_SAMPLES_PER_FOLDER, len(valid_paths))
        )

        for src in selected_paths:
            dst = os.path.join(output_path, os.path.basename(src))
            shutil.copy2(src, dst)

        if verbose:
            print(f"Copied {len(selected_paths)} images to {output_path}")

    print("Sampling complete.")

all_subdirs = [
    "train/Coccidiosis",
    "train/Healthy",
    "train/New Castle Disease", #DONE
    "train/Salmonella", #IN PROGRESS
    "val/Coccidiosis",
    "val/Healthy",
    "val/New Castle Disease",
    "val/Salmonella",
    "test/Coccidiosis", #DONE
    "test/Healthy", #DONE
    "test/New Castle Disease", #DONE
    "test/Salmonella" #DONE
]

subdir_to_process = ["train/Salmonella"]
diverse_sample_per_subdirectory(subdir_to_process,verbose=True)
```