Idées projets

December 19, 2024

1 ConvNext

Ce sujet essaye de reproduire le papier "A ConvNet for the 2020s" (https://arxiv.org/abs/2201.03545) en petite échelle. Dans le papier, les auteurs introduisent certaines modifications (inspirées par les transformers) à une ResNet à fur et à mesure, et à chaque fois ils testent sur ImageNet.

Le but du projet est de faire le même, en testant chaque étape (ou en tout cas plusieurs étapes) intermédiaire(s) sur le petit jeu de données ImageNette.

Une implémentation du modèle complet est disponible sur https://pytorch.org/vision/main/_modules/torchvision/models/convnext.html, mais il conviendra en réécrire la plupart pour avoir flexibilité sur les étapes introduites.

2 Entraîner une petite (Ro)BERT(a) en français

Le but est d'entraîner un (très) petit modèle de langage "encodeur" en français, suivant la méthode de BERT ou (mieux, c'est plus simple) RoBERTa. Papiers originaux : https://arxiv.org/abs/1810.04805 et https://arxiv.org/abs/1907.11692. Un blog post a fait ça en anglais : https://sidsite.com/posts/bert-from-scratch/ mais ce projet devrait être différent sous plusieurs aspects :

1. Données en français. On peut utiliser le jeu de données Oscar. Le plus simple est utiliser HuggingFace pour cela, voir instructions ici : https://oscar-project.github.io/documentation/accessing/ et ici : https://huggingface.co/datasets/oscar-corpus/OSCAR-2301. Taille : 62 milliards de mots, pour un total de 430 GB (pas besoin de tout télécharger :-) on peut faire en "streaming" et on ne va utiliser qu'une partie du dataset, ce n'est pas grave) ;
2. Par contre, pour le modèle et entraînement, on ne va pas utiliser HuggingFace, mais directement une boucle d'entraînement PyTorch. Pour le modèle, on peut utiliser torch.nn.TransformerEncoder ; pour les paramètres, on va utiliser un petit modèle, par exemple en suivant les valeurs ici : https://github.com/google-research/bert (BERT-tiny ou BERT-mini ou BERT-small) ;
3. La partie de code PyTorch à implémenter sera surtout la fonction de perte et le Dataset / DataLoader pour masquer des tokens. On peut aussi s'appuyer sur nombreuses pages en ligne qui le font ;
4. Finalement, pour évaluer le modèle final, on peut prendre des phrases qui ne sont sans doute pas dans le jeu d'entraînement (par exemple, des articles de journal sorties dans le dernier mois), masquer des mots et voir s'il le complète correctement (et analyser les cas d'erreur). On peut aussi faire du fine-tuning sur une autre tâche (par exemple, classification du texte), qui est ce pour lequel on a pensé BERT dès le début, mais ça devient trop de boulot peut-être.

3 Entraîner un petit GPT / Llama en français

La différence, par rapport à (Ro)BERT(a) ci-dessus, est que c'est un modèle décodeur, non pas encodeur. Cela rend la tâche un peu plus compliquée, mais en même temps, comme c'est très à la mode en ce moment, il y a plus de ressources. Par exemple :

- https://www.youtube.com/watch?v=kCc8FmEb1nY vidéo en anglais par Karpathy, qui a évolué chez OpenAI et Tesla. Disclaimer : je ne l'ai pas regardé.
- https://github.com/bkitano/llama-from-scratch qui se concentre sur Llama, la version open source de chez Meta. Cette guide utilise un très simple tokenisateur au niveau des caractères, mais cela va réduire fortement la qualité du modèle ; il serait mieux d'utiliser un tokenisateur niveau style SentencePiece. Le faire de zéro (en dessus du reste) serait trop de travail, pour ce projet on peut utiliser un tokenisateur pré-entraîné (sur des textes français) par HuggingFace (par exemple https://huggingface.co/camembert-base).

On peut utiliser les mêmes données Oscar que dans le projet ci-dessus. Par rapport à ce projet ci-dessus, il y a moins de difficultés niveau préparation données, et plus niveau modélisation.

Finalement, on peut tester ce modèle en lui proposant un début de phrase et en voyant comment il le complète. Le plus intéressant c'est de le laisser inventer des phrases entières en faisant une boucle où chaque token suivant est prédit en utilisant aussi les prédictions précédentes.

Ressources : https://github.com/karpathy/nanoGPT pour un modèle de type GPT ; pour Llama il y a beaucoup de ressources de qualité variée ; par exemple https://medium.com/@venkat.ramrao/training-a-mini-114m-parameter-llama-3-like-model-from-scratch-97525185aa9c pour le plus récent (et meilleur) Llama 3, https://github.com/bkitano/llama-from-scratch pour une version moins récent. Le lien HuggingFace https://huggingface.co/blog/nroggendorff/train-with-llama-architecture permet probablement d'obtenir des meilleurs résultats avec moins d'effort, mais il ne rentre pas dans les détails, pour le but du projet il faudra étudier plus en profondeur.

4 Modèle de diffusion pour générer des images

Le modèle de diffusion qui est probablement le plus connu est StableDiffusion : https://stability.ai/news/stable-diffusion-public-release. Il produit des images intéressantes, donnée une phrase, mais l'implémenter est au-delà de nos possibilités. Surtout mélanger texte et images est trop difficile.

Les modèles de diffusion pour engendrer des images ont été introduits par ce papier : https://arxiv.org/abs/2006.11239v2 (un peu lourd en maths, mais intéressant pour comprendre l'idée derrière)

On peut essayer de reproduire une version simple, où :

- Le modèle est simplifié
- Il est entraîné sur un jeu de données homogène, d'une façon qu'il n'est capable qu'à reproduire des images du même type.

Pour cela, on peut suivre cette guide : https://tree.rocks/make-diffusion-model-from-scratch-easy-way-to-implement-quick-diffusion-model-e60d18fd0f2e (le code sur cette page et en TensorFlow, mais il y a la version PyTorch ici : https://github.com/Seachaos/Tree.Rocks/blob/main/QuickDiffusionModel/QuickDiffusionModel_torch.ipynb).

Or, faire du copié-collé d'un modèle n'est pas très intéressant :-) donc il y a quelques tâches à faire :

1. Tester d'autres jeux de données. Cette version utilise les voitures dans CIFAR10, donc 6000 images en total. On peut utiliser d'autres jeux de données plus grands et intéressants, par exemple https://www.kaggle.com/datasets/jessicali9530/celeba-dataset (200k photos du visage de célébrités), ou si vous voulez des voitures https://www.kaggle.com/datasets/prondeau/the-car-connection-picture-dataset ou bien ce que vous préférez. Si le jeu de données est assez grand, on peut se passer de l'augmentation. Par contre, toujours utiliser une résolution fixe et petite, par exemple 32x32 comme dans la guide, ou un peu plus (48x48 ? 64x64 ? À vous de voir)
2. Une fois fixé le jeu de données, tester d'autres variantes de modèles pour améliorer la qualité des résultats. (En revanche, d'abord toujours tester sur le petit jeu de données CIFAR10 utilisé ci-dessus, ça va être plus rapide à itérer)
3. Une fois que vous avez un modèle qui marche assez bien, on va modifier le modèle pour permettre du "inpainting". Un peu comme ceci : https://getimg.ai/guides/inpainting-with-stable-diffusion mais plus simple, sans texte. C'est à dire, donnée une image, on va juste implémenter une version du modèle qui permet de créer une image similaire à celle dans le jeu de données, mais suivant un gribouillage (ou même modifier une image existante en effaçant et esquissant une partie), comme ça : https://mccormickml.com/2022/12/06/how-img2img-works/. Pour le faire, on va implémenter ce papier : https://arxiv.org/abs/2108.01073 au-dessus du modèle déjà implémenté, qui est une idée très simple mais efficace.

5 LoRA - Low Rank Adaptation, mais pour convolutions

Ce papier : https://arxiv.org/abs/2106.09685 a introduit une méthode simple pour faire du fine-tuning d'un modèle de langage large, en n'entraînant que peu de paramètres. Ceci est nécessaire si on veut entraîner des modèles larges sur ses propres données, si on n'a pas énormément de données et de puissance de calcul. Et c'est devenu très populaire dans le monde open source (par exemple sur les modèles Mistral ou Llama). Par exemple, le module `peft` s'intègre facilement avec HuggingFace.

Dans ce projet, je propose d'essayer une implémentation à partir de zéro de cette méthode. Comme l'intégration avec des modèles HuggingFace prendrait probablement plus de temps que LoRA elle-même, et pour essayer de faire quelque chose de nouveau, je propose plutôt de l'essayer sur des modèles de traitement d'images, où il est peu utilisé.

Par exemple :

- On peut partir des modèles distribués par `torchvision` (ResNet, ConvNeXt, ce que vous voulez, en commençant par un petit, ensuite en utilisant un gros), qui sont distribués pré-entraînés sur ImageNet.
- On choisit un autre jeu de données d'images qui ne soient pas dans ImageNet ; par exemple, https://www.kaggle.com/datasets/gpiosenka/100-bird-species ou https://www.kaggle.com/datasets/crowww/a-large-scale-fish-dataset ou https://www.kaggle.com/datasets/grassknoted/asl-alphabet ou https://www.kaggle.com/datasets/jessicali9530/stanford-cars-dataset etc.
- On essaye de faire du fine-tuning du modèle sur ce jeu de données. On mesure : mémoire utilisée dans la GPU / taille des lots maximale qu'on peut utiliser, temps d'exécution, taille du modèle, accuracy.
- Ensuite, on implémente un modèle style LoRA sur certaines ou toutes les couches convolutives. La définition standard de LoRA est créée pour des couches linéaires, mais le même concept peut s'adapter aux couches convolutives : si le modèle base a une certaine couche `Conv2d(in_channels, out_channels, kernel_size)`, dans la notation de la Figure 1 dans le papier de LoRA on peut par exemple utiliser `A = Conv2d(in_channels, h, kernel_size)` et `B = Conv2d(h, out_channels, 1)` (ou bien inverser `kernel_size` et `1`, ou bien utiliser `kernel_size` deux fois), avec `h` petit. On peut donc faire du fine-tuning uniquement des couches `A` et `B` ainsi construites, et voir si on arrive à une accuracy comparable, en combien de temps, avec quelle taille des lots / utilisation mémoire, et quelle est la taille des poids qui ont été entraînés.

6 Deep Reinforcement Learning pour jouer à Doom

Remarque : c'est un sujet vaste qu'on ne peut peut-être pas maîtriser facilement. Donc je le déconseille, sauf si vous avez du temps à y dédier, et que vous cherchez un challenge. De plus, je ne l'ai pas essayé moi-même, il peut y avoir des difficultés inattendues.

L'idée serait de reproduire ce code : https://tinyurl.com/y2as8b25 pour faire un programme qui joue à (une version très simplifiée de) Doom.

La théorie nécessaire est expliquée dans les trois premières parties de ce tutoriel par le même auteur : https://medium.com/free-code-camp/an-introduction-to-reinforcement-learning-4339519de419

Difficultés à surmonter :

- Il faut apprendre à travailler avec la bibliothèque `vizdoom` qui permet de simuler le jeu de Doom
- Le notebook ci-dessus est en TensorFlow, il faudra le traduire en PyTorch (mais la partie réseaux de neurones n'est pas très compliquée)
- Il faut étudier un peu de théorie sur l'apprentissage des stratégies (équation de Bellman, stratégie ε-greedy - ou plus avancée -, mémoire, etc.)

On peut même s'appuyer sur ce script : https://github.com/Farama-Foundation/ViZDoom/blob/master/examples/python/learning_pytorch.py qui est déjà écrit en PyTorch, mais qui n'a pas d'explications.

7 Fine-tuning des modèles de langage

Ce sujet est très à la mode en ce moment. Par conséquent, il y existe beaucoup de guides en ligne ; le but de ce projet est d'explorer plusieurs méthodes et les comparer. Il y a donc plus de tâches que dans les autres projets, mais pour chaque tâche il y a plus de ressources disponibles. On utilisera la bibliothèque HuggingFace pour tous les morceaux.

1. On commence par la méthode traditionnelle : on prend un modèle de langage pré-entraîné de HuggingFace, on crée/adapte un jeu de données, et on fine-tune. Ici, on va considérer un modèle encodeur seul, de type BERT (l'un de `bert-base-uncased`, `roberta-base`, etc.), avec un jeu de données simple de classification tel que `financial_phrasebank`. Le but est d'entraîner un modèle fine-tuné qui prédit si une phrase issue des infos financières est négative, positive, ou neutre. Pour cela, il suffit de suivre les instructions ici : https://huggingface.co/docs/transformers/training. Après avoir obtenu un premier modèle qui marche, on peut comparer des modèles différents (type : `bert` vs. `roberta` vs `xlm-roberta` vs. `deberta`, et taille : `base` vs. `large`, etc.)
2. Pour des modèles plus grands, on ne peut pas faire du fine-tuning complet, car il y a trop de paramètres (on n'a pas assez de mémoire). On utilise alors des méthodes "efficientes" comme LoRA. On peut suivre ce tutoriel : https://tinyurl.com/3wead4hr pour effectuer le même fine-tuning sur la famille `flan-t5`, qui a plus de paramètres (jusqu'à 12 milliards de paramètres). Il est aussi un modèle de type encodeur-décodeur ; en particulier, les données doivent être préparées de façon différente : à la place d'effectuer une classification, on prédit une phrase (dans ce tutoriel, un mot parmi "positive", "neutral", "negative"). Une fois que ça marche, l'exercice est d'essayer des modèles de tailles différentes (de `small` à… ce qui rentre dans la mémoire, sans doute pas `xxl`). Ensuite, on élimine LoRA pour essayer un fine-tune de tous les paramètres de façon "classique". Niveau code, il y a très peu, presque rien à changer (juste éliminer toute référence à LoRA…). Est-ce que ça marche mieux ? On devra réduire de façon significative le nombre de paramètres.
3. Finalement, on utilise cette même structure pour faire quelque chose de plus original. Ces modèles sont entraînés sur du texte, mais peuvent traiter tout ce qu'on peut écrire sous forme de texte. On va prendre des données tabulaires, par exemple https://www.kaggle.com/competitions/playground-series-s3e24/overview. Ici on a un fichier `csv` où on veut prédire la dernière colonne, fumeur ou pas. Il faudra transformer chaque ligne de ce fichier `csv` en deux pièces : toutes les colonnes, sauf la dernière, seront le texte que le modèle voit, et la dernière colonne (traduite en texte, comme ci-dessus) la "phrase" que le modèle doit prédire. Essayez à nouveau plusieurs stratégies (LoRA ou pas) et plusieurs tailles de modèle.

8 OCR (reconnaissance optique des caractères)

C'est un problème qu'utilise du traitement d'images ainsi que du langage, pour apprendre à une machine à "lire" une page de texte. Ici on va se concentrer sur une ligne seulement, ou même un seul mot, pour souci de simplicité.

1. Télécharger un jeu de données de texte écrit. Une source standard est : https://fki.tic.heia-fr.ch/databases/iam-handwriting-database, si vous arrivez à vous inscrire. Sinon il y a aussi https://github.com/facebookresearch/IMGUR5K-Handwriting-Dataset qui télécharge les images de Imgur et extrait les mots.
2. La classe Dataset doit prendre ce type d'images (lignes ou mots, selon le jeu de données et votre préférence), les transforme en taille standardisée (toujours la même, `HxW`), en échelle de gris, résolution assez haute.
3. Le modèle doit être formé par des couches de convolutions, avec des poolings qui réduisent la taille jusqu'à quelque chose d'assez petit (au plus 128 pixels de longueur, aussi peu que possible d'hauteur) et ensuite "écrase" la dimension `H` entièrement, en la traduisant dans la dimension `C` ; c'est-à-dire, on passe de `NCHW`, comme typique dans les images, à `NC'W`, comme typique dans le traitement de texte, avec $C' = C \cdot H$. Ensuite, on utilise quelques couches, soit LSTM soit de transformer encodeur.
4. Finalement, l'apprentissage se fait au mieux par la fonction de perte CTC : https://pytorch.org/docs/stable/generated/torch.nn.CTCLoss.html.

Entraîner un tel modèle, et l'évaluer sur le jeu de test. Analyser visuellement les cas d'erreurs, pour voir si le modèle fait quand même quelque chose de "raisonnable". Typiquement on mesure les erreurs avec le "taux d'erreurs de caractères" (Character Error Rate), c'est à dire la somme des distances de Levenshtein (https://fr.wikipedia.org/wiki/Distance_de_Levenshtein) entre chaque phrase / mot prédit(e), divisée par la somme des longueurs des phrases / mots à prédire. Un bon modèle doit avoir un taux d'erreur < 5 %, mais ce n'est pas très simple avec un jeu de données si petit.

(Un papier de référence est : https://arxiv.org/abs/2104.07787 mais il ne donne pas beaucoup plus de détails que ci-dessus…)

9 WGAN (-GP)

Les GANs (réseaux "antagonistes" de génération) sont des réseaux de simulation de points d'une distribution de probabilité inconnue. En pratique, elles ont été popularisées pour créer, par exemple, des images similaires à d'autres images dans un jeu de données. Aujourd'hui, on préfère la méthode par diffusion ; par rapport à cette méthode, les GANs ont des avantages (plus rapides à entraîner et à utiliser pour la génération) mais aussi des désavantages (notamment, plus de répétitivité). Les GANs originales étaient aussi très "délicats" à entraîner, avec des paramètres qui devaient être choisis avec attention, pour éviter des dégénérations à des résultats triviaux. Ceci a changé avec l'introduction des Wasserstein-GAN (WGAN) dans ce papier : https://arxiv.org/abs/1701.07875 (légèrement amélioré en "WGAN-GP" ici : https://arxiv.org/abs/1704.00028). On va essayer la même méthode pour générer des images similaires à celle d'un jeu de données choisi. Par exemple les jeux de données CelebA ou modèles de voitures mentionnés dans le projet de la diffusion.

Beaucoup de ressources sont disponibles, par exemple :

- Le GitHub du papier original : https://github.com/martinarjovsky/WassersteinGAN
- Un autre repo GitHub qui implémente WGAN et WGAN-GP : https://github.com/Zeleni9/pytorch-wgan
- Un vidéo qui fait aussi les deux : https://www.youtube.com/watch?v=pG0QZ7OddX4
- Un blog qui décrit ce qu'il fait : https://machinelearningmastery.com/how-to-code-a-wasserstein-generative-adversarial-network-wgan-from-scratch/ (en TensorFlow et non pas PyTorch, mais ça change très peu).

Le but de ce projet est de réécrire l'une de ces versions pour l'entraîner sur un nouveau jeu de données (la vidéo utilise peut-être déjà CelebA, si vous suivez la vidéo, essayé un autre jeu de données aussi). De plus, essayez de ne pas copier le code directement, mais de s'en inspirer pour créer des modèles de façon itérative, en commençant par des modèles très petits et rapides à entraîner (qui, par contre, donneront des mauvais résultats ; par exemple, commencer par un jeu très simple comme un seul chiffre du jeu MNIST, comme dans le blog au dernier lien), et ensuite en grandissant le modèle de façon progressive.

10 (Grand) modèle de langage pour question-réponse

L'application "telle quelle" de modèles pré-entraînés n'est pas tellement le sujet de ce cours, mais désormais est aussi une expertise demandée aux scientifiques des données, donc cet exercice vise à une telle application. Ce sujet n'est à choisir que si quelqu'un veut se focaliser plus sur le côté "engineering" des applications et moins sur le côté scientifique. De plus, selon les détails choisis ci-dessous (Weaviate, Flask) il faudra peut-être travailler sur votre propre ordinateur et non pas sur un notebook comme dans Colab. Dans ce cas, l'ordinateur doit malheureusement être assez puissant (normalement, pas de GPU nécessaire, mais les temps d'attente peuvent devenir longs).

On va utiliser deux modèles pré-entraînés :

1. Un modèle de "embedding du texte", qui transforme un texte donné en un vecteur $v \in \mathbb{R}^d$, de norme unitaire, avec la propriété que plus de choses deux textes ont en commun (sujet, signification, forme, etc.), plus les vecteurs correspondants seront proches ($\|v1 - v2\| \approx 0$). Pour cela, on peut choisir n'importe quel modèle bien placé dans ce classement : https://huggingface.co/spaces/mteb/leaderboard. Il est possible qu'on ait besoin d'un modèle avec assez grande "Sequence length", et on choisit un modèle disponible en open source sur HuggingFace (on ne va pas s'amuser à payer OpenAI ou Cohere ou autres qui figurent dans le classement)
2. Un grand modèle de langage. Pour avoir un ensemble assez rapide, on va utiliser au plus 7 milliards de paramètres, ce qui est le minimum pour avoir des résultats acceptables, mais sera déjà difficile à traiter. En ce moment, le meilleur modèle open sources est Mistral https://huggingface.co/mistralai/Mistral-7B-v0.1 ou, un peu meilleur, Llama v3.3 https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_3/. Pour commencer, c'est mieux d'utiliser des modèles plus petits, tels que SmolLM : https://huggingface.co/HuggingFaceTB/SmolLM-1.7B (en fait, c'est assez bon, peut-être on peut même se limiter à celui-ci !)

On va les utiliser pour pouvoir poser des questions sur une base de données de documents à un chatbot intelligent, ce qui s'appelle "Retrieval Augmented Generation" (RAG). On peut suivre par exemple ce guide : https://medium.com/@onkarmishra/using-langchain-for-question-answering-on-own-data-3af0a82789ed. Les étapes à suivre sont :

1. Trouver des documents qui traitent un certain sujet, dont vous êtes intéressés. Ça peut être n'importe quoi : tous ces longs documents que personne ne lit quand on signe un contrat avec une banque et auquel vous voulez juste demander "est-ce que je dois payer pour retirer de l'argent ?", tous les polys de vos cours de Python préférés (mais pour les maths ça marche moyennement), toutes les paroles de toutes les chansons du monde pour demander "donne-moi trois chansons qui parlent d'un chat mais aussi d'un astronaute", etc.
2. Partitionner ces documents en petits morceaux (en tout cas, plus petits que la "Sequence length" du modèle de embedding), et calculer leurs embeddings.
3. Quand on pose une question, calculer aussi l'embedding de la question.
4. Trouver les k vecteurs dont la distance entre les vecteurs est la moindre (pour quelque k à choisir). On peut décider d'utiliser une base de données vectorielles (par exemple, Weaviate, mais il y en a beaucoup) pour stocker tous les embeddings des documents ; si on le fait, il va trouver les plus k éléments les plus proches pour nous, avec des algorithmes optimisés.
5. Formuler une question à faire compléter par le modèle de langage, du style :

```
Voici les contextes :
```

```
...
Avec les contextes ci-dessus, répondre à la question suivante :
...
```

(il faudra peut-être lui parler en anglais, par contre)

Comme dans l'article ci-dessus, on peut utiliser Langchain pour rendre certaines de ces requêtes plus simples. Idéalement, on voudrait avoir une interface, par exemple une page web avec Flask, qui présente cela joliment. Mais sinon une interface de ligne de commande sera aussi okay.

11 U-Net en imagerie medicale

Le but est d'effectuer une segmentation d'une image, pour identifier qu'est ce qu'il y a dans chaque pixel. Une source que vous pouvez utiliser pour le code est ici :

https://github.com/aladdinpersson/Machine-Learning-Collection/tree/master/ML/Pytorch/image_segmentation

Le code devrait être expliqué dans cette vidéo (mais je ne l'ai pas regardée) :

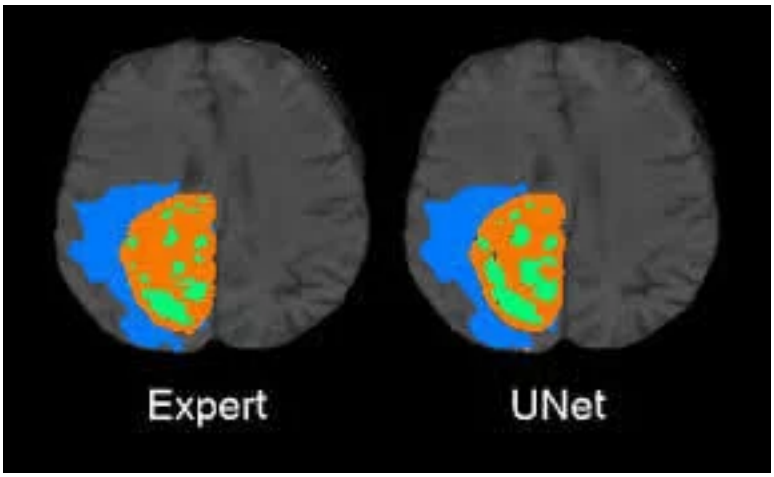
https://www.youtube.com/watch?v=IHq1t7NxS8k

Concernant les données, vous pouvez choisir un dataset de votre préférence ici : https://paperswithcode.com/datasets?task=medical-image-segmentation. Il faudra qu'il contienne des étiquettes de type "masque", qui marquent les zones à trouver. Il est possible qu'il faudra faire un peu de travail sur les données : le modèle dans la vidéo est entraîné sur ce jeu de données : https://www.kaggle.com/c/carvana-image-masking-challenge qui ne contient que des masques voiture / pas de voiture. Je n'ai pas regardé le code de prêt, mais c'est possible que ça ne marche que dans ce cas binaire. Dans un jeu médical c'est possible qu'il y ait plusieurs objets annotées (par exemple : rien, cellule saine, cellule tumorale), vous devrez soit ajuster le code pour supporter plusieurs classes, soit modifier les masques pour n'avoir qu'un problème binaire.

Finalement, le répositoire ne contient que du code en format script. Pour la présentation, je vous conseil de le faire tourner par exemple dans un jupyter (ou mieux colab), qui vous permettra de tracer des images pour visualiser l'image initiale, la "ground truth" (les pixels marqués selon leur classe) et la prédiction de votre modèle sur la même image. Le résultat devrait être quelque chose comme les images suivantes (dans la première, les masques sont superposées à l'image ; dans la deuxième, les masques sont dessinées juste au dessous) :

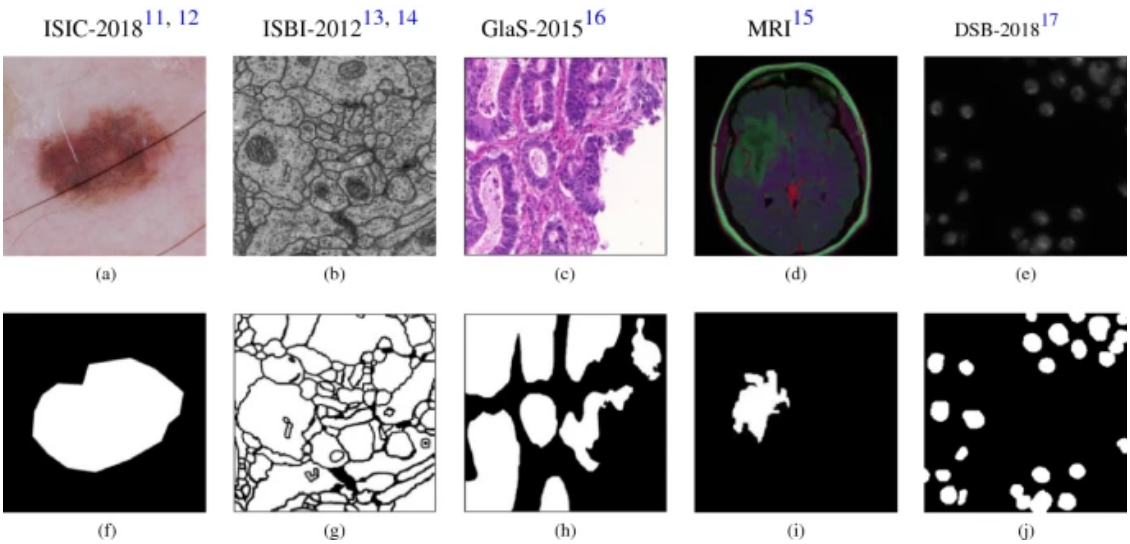
[2]:
```
Image('unet_cerveau.png')
```

[2]:



[3]:
```
Image('unet_mixed.png')
```

[3]:


ISIC-2018[11, 12]
ISBI-2012[13, 14]
GlaS-2015[16]
MRI[15]
DSB-2018[17]
(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)