Project ideas

December 19, 2024

1 ConvNext

This subject try of reproduce THE paper "HAS ConvNet for tea 2020s" (https://arxiv.org/abs/2201.03545) on a small scale. In the paper, the authors introduce some modifications (inspired by transformers) to a ResNet as they go along, and each time they test on ImageNet.

The goal of the project is to do the same, testing each intermediate step (or at least several steps) on the small ImageNette dataset.

An implementation of the full model https://pytorch.org/vision/mai Ea st available on _modules/torchvision/models/convnext.html, will rewrite most of them to but he con- have flexibility on the steps introduced.

2 Train a small (Ro)BERT(a) in French

The goal is to train a (very) small “encoder” language model in French, following the BERT method or (better, it’s simpler) RoBERTa. Original papers: https://arxiv.org/abs/1810.04805 and https://arxiv.org/abs/1907.11692. A blog post did that in English: https://sidsite.com/posts/bert-from-scratch/ but this project should be different in several aspects:

1. Data in French. We can use the Oscar dataset. The most simple is to use HuggingFace for this, see instructions here: project.github.io/https://oscar-documentation/accessing/ and here: https://huggingface.co/datasets/oscarcorpus/OSCAR-2301. Size: 62 billion words, for a total of 430 GB (no need to download everything :-) we can do it in “streaming” and we will only use part of the dataset, it does not matter);
2. On the other hand, for the model and training, we will not use HuggingFace, but directly a PyTorch training loop. For the model, we can use torch.nn.TransformerEncoder; for the parameters, we will use a small model, for example by following the values here: https://github.com/google-research/bert (BERT-tiny or BERT-mini or BERT-small);

3. The PyTorch code part to implement will be mainly the loss function and the Dataset / DataLoader to hide tokens. We can also rely on many online pages that do this;

4. Finally, to evaluate the final model, we can take sentences that are probably not in the training set (e.g. newspaper articles published in the last month), mask words and see if it completes it correctly (and analyze the error cases). We can also do fine-tuning on another task (e.g. text classification), which is what we thought BERT for from the beginning, but it's getting too much work maybe.

3 Training a little GPT / Llama in French

The difference, compared to (Ro)BERT(a) above, is that it is a decoder model, not an encoder. This makes the task a bit more complicated, but at the same time, as it is very fashionable at the moment, there are more resources. For example:

- https://www.youtube.com/watch?v=kCc8FmEb1nY English video by Karpathy, who worked at OpenAI and Tesla. Disclaimer: I have not watched it.
- https://github.com/bkitano/llama-from-scratch which focuses on Llama, the open source version from Meta. This guide uses a very simple tokenizer at the character level, but this will greatly reduce the quality of the model; it would be better to use a SentencePiece style level tokenizer. Doing it from scratch (on top of the rest) would be too much work, for this project we can use a pre-trained tokenizer (on French texts) by HuggingFace (for example https://huggingface.co/camembert-base).

The same Oscar data can be used as in the above project. Compared to the above project, there are fewer difficulties in data preparation, and more in modeling.

Finally, we can test this model by proposing a beginning of a sentence and seeing how it completes it. The most interesting thing is to let it invent entire sentences by making a loop where each following token is predicted using also the previous predictions.

Resources : https://github.com/karpathy/nanoGPT for a model of kind GPT; for Llama there are many resources of varying quality; sample eg- https://medium.com/@venkat.ramrao/training-a-mini-114m-parameter-llama-3-like-model-from-scratch-97525185aa9c for the newer (and better) Llama 3, https://github.com/bkitano/llama-from-scratch for an older version. The HuggingFace link https://huggingface.co/blog/nroggendorff/train-with-llama-architecture probably allows you to get better results with less effort, but it doesn't go into detail, for the purpose of the project it will be necessary to study more in depth.

4 Diffusion model for generating images

The diffusion model that is probably the best known is StableDiffusion: https://stability.ai/news/stable-diffusion-public-release. It produces interesting images, given a sentence, but implementing it is beyond our capabilities. Especially mixing text and images is too difficult.

Diffusion models for generating images were introduced by this paper: https://arxiv.org/abs/2006.11239v2 (a bit heavy in math, but interesting to understand the idea behind)

We can try to reproduce a simple version, where:

- The model is simplified
- It is trained on a homogeneous dataset, in a way that it is only able to reproduce images of the same type.

For This can be followed this guide : https://tree.rocks/make-diffusion-model-from-scratch-easy-way-to-implement-quick-diffusion-model-e60d18fd0f2e (THE code on this page and in TensorFlow, but there is the PyTorch version here: https://github.com/Seachaos/Tree.Rocks/blob/main/QuickDiffusionModel/QuickDiffusionModel_torch.ipynb).

However, copy-pasting a template is not very interesting :-) so there are a few tasks to do:

1. Test other datasets. so 6000 This version uses cars in CIFAR10, other datasets images in total. can be used more big and interesting ones, for example https://www.kaggle.com/datasets/jessicali9530/celebadataset (200k celebrity face pictures), or if you want cars https://www.kaggle.com/datasets/prondeau/the-car-connection-picture-dataset or whatever you prefer. If the dataset is big enough, you can do without the augmentation. However, always use a fixed and small resolution, for example 32x32 as in the guide, or a bit more (48x48? 64x64? It's up to you)
2. Once the dataset is fixed, test other model variants to improve the quality of the results. (On the other hand, always test first on the small CIFAR10 dataset used above, it will be faster to iterate)
3. Once you have a model that works well enough, we will modify the model to allow "inpainting". A bit like this: https://getimg.ai/guides/inpainting-withstable-diffusion but simpler, without text. That is to say, given an image, we will just implement a version of the model that allows to create an image similar to the one in the dataset, but following a scribble (or even modify an existing image by erasing and sketching a part), like this: https://mccormickml.com/2022/12/06/how-img2imgworks/. To do this, we will implement this paper: https://arxiv.org/abs/2108.01073 above the already implemented model, which is a very simple but effective idea.

5 LoRA - Low Rank Adaptation, but for convolutions

This paper: https://arxiv.org/abs/2106.09685 introduced a simple method to finetune a large language model, by training only a few parameters. This is necessary if you want to train large models on your own data, if you don't have a lot of data and computing power. And it has become very popular in the open source world (for example on the Mistral or Llama models). For example, the modulepefteasily integrates with HuggingFace.

In this project I propose to try a scratch implementation of this method. Since integrating with HuggingFace models would probably take more time than LoRA itself, and to try to do something new, I propose instead to try it on image processing models, where it is rarely used.

For example :

- We can start from the models distributed bytorchvision (ResNet, ConvNeXt, whatever you want, starting with a small one, then using a big one), which are distributed pretrained on ImageNet.
- We choose another image dataset that is not in ImageNet; by example, https://www.kaggle.com/datasets/gpiosenka/100-bird species Or https://www.kaggle.com/datasets/crowww/a-large-scale-fish-dataset Or https://www.kaggle.com/datasets/grassknoted/asl-alphabet Or https://www.kaggle.com/datasets/jessicali9530/stanford-cars-dataset etc.
- We try to fine-tune the model on this dataset. We measure: memory used in the GPU / maximum batch size that can be used, execution time, model size, accuracy.

- Next, we implement a LoRA-style model on some or all of the convolutional layers. The standard definition of LoRA is created for linear layers, but the same concept can be adapted to convolutional layers: if the base model has a certain layerConv2d(in_channels, out_channels, kernel_size),in the notation of Figure 1 in the LoRA paper we can for example useA = Conv2d(in_channels, h, kernel_size)AndB = Conv2d(h, out_channels, 1) (or reversekernel_sizeAnd1,or usekernel_sizetwice), withhsmall. So we can do fine-tuning only of the layersHASAndBthus constructed, and see if we arrive at a comparable accuracy, in how much time, with what batch size / memory usage, and what is the size of the weights that were trained.

6 Deep Reinforcement Learning for Playing Doom

Note: This is a vast topic that may not be easy to master. So I don't recommend it unless you have time to dedicate to it, and are looking for a challenge. Also, I haven't tried it myself, so there may be unexpected difficulties.

The idea would be to reproduce this code: https://tinyurl.com/y2as8b25 to make a program that plays (a very simplified version of) Doom.

The necessary theory is explained in the first three parts of this tutorial by the same author: https://medium.com/free-code-camp/an-introduction-to-reinforcement-learning-4339519de419

Difficulties to overcome:

- You have to learn to work with the libraryvizdoomwhich allows you to simulate the game of Doom
- The notebook above is in TensorFlow, it will need to be translated into PyTorch (but the neural networks part is not very complicated)
- It is necessary to study some theory on learning strategies (Bellman equation, strategy - greedy - or more advanced -, memory, etc.)

We can even rely on this script : https://github.com/Farama-Foundation/ViZDoom/blob/master/examples/python/learning_pytorch.py which is already written in PyTorch, but has no explanations.

7 Fine-tuning language models

This topic is very trendy at the moment. Therefore, there are many guides online; the goal of this project is to explore several methods and compare them. So there are more tasks than in other projects, but for each task there are more resources available. We will use the HuggingFace library for all the pieces.

1. We start with the traditional method: we take a pre-trained language model from HuggingFace, we create/adapt a dataset, and we fine-tune. Here, we will consider an encoder-only model, of the BERT type (one ofbert-base-uncased, roberta-base, etc.), with a simple classification dataset such asfinancial_phrasebank.The goal is to train a fine-tuned model that predicts whether a sentence from financial news is negative, positive, or neutral. To do this, simply follow the instructions here: https://huggingface.co/docs/transformers/training. After having obtained a first model that works, we can compare different models (type:bertvs.robertavsxlm-roberta vs.beggar,and size:basevs.wide,etc.)

2. For larger models, we cannot do full fine-tuning, because there are too many parameters (we don't have enough memory). We then use "efficient" methods like LoRA. We can follow this tutorial: https://tinyurl.com/3wead4hr to perform the same fine-tuning on the familyflan-t5,which has more parameters (up to 12 billion parameters). It is also an encoder-decoder type model; in particular, the data must be prepared differently: instead of performing a classification, a sentence is predicted (in this tutorial, a word among “positive”, “neutral”, “negative”). Once it works, the exercise is to try models of different sizes (ofsmallto... what comes into memory, probably notxxl).Then, we eliminate LoRA to try a fine-tune of all the parameters in a “classic” way. Code-wise, there is very little, almost nothing to change (just eliminate any reference to LoRA...). Does it work better? We will have to significantly reduce the number of parameters.

3. Finally, we use this same structure to do something more original. These models are trained on text, but can handle anything that can be written in text form. We will take tabular data, for example https://www.kaggle.com/competitions/playground-series-s3e24/overview. Here we have a file csvwhere we want to predict the last column, smoker or not. We will have to transform each line of this filecsvin two pieces: all columns except the last one will be the text the model sees, and the last column (translated to text, as above) the “sentence” the model should predict. Try again several strategies (LoRA or not) and several model sizes.

8 OCR (optical character recognition)

This is a problem that uses image processing as well as language, to teach a machine to “read” a page of text. Here we will focus on just one line, or even just one word, for the sake of simplicity.

1. Download a dataset of handwritten text. A standard source is: https://fki.tic.heiafr.ch/databases/iam-handwriting-database, if you can register. Otherwise there is also https://github.com/facebookresearch/IMGUR5K-Handwriting-Dataset which downloads images from Imgur and extracts the words.
2. The Dataset class should take this type of images (lines or words, depending on the dataset and your preference), transform them into a standardized size (always the same,HxW),in grayscale, fairly high resolution.
3. The model should be trained by convolution layers, with poolings that reduce the size to something small enough (at most 128 pixels in length, as little as possible in height) and then “squash” the dimensionHentirely, by translating it into the dimensionC ;that is to say, we move fromNCHW,as typical in the pictures,NC'W,as typical in word processing, with ′ = · .Then we use a few layers, either LSTM or transformer encoder.
4. Finally, learning is best done by the CTC loss function: https://pytorch.org/docs/stable/generated/torch.nn.CTCLoss.html.

Train such a model, and evaluate it on the test set. Visually analyze the error cases, to see if the model still does something “reasonable”. Typically, errors are measured with the “Character Error Rate”, i.e. the sum of the Levenshtein distances (https://fr.wikipedia.org/wiki/Distance_de_Levenshtein) between each predicted sentence/word, divided by the sum of the lengths of the sentences/words to be predicted. A good model should have an error rate < 5%, but this is not very easy with such a small dataset.

(A reference paper is: https://arxiv.org/abs/2104.07787 but it doesn't give much more detail than above...)

9 WGAN (-GP)

GANs (Generational Adversarial Networks) are networks that simulate points of an unknown probability distribution. In practice, they were popularized to create, for example, images similar to other images in a dataset. Today, the diffusion method is preferred; compared to this method, GANs have advantages (faster to train and use for generation) but also disadvantages (notably, more repeatability). The original GANs were also very "tricky" to train, with parameters that had to be chosen carefully, to avoid degenerations to trivial results. This changed with the introduction of Wasserstein-GANs (WGAN) in this paper: https://arxiv.org/abs/1701.07875 (slightly improved to "WGAN-GP" here: https://arxiv.org/abs/1704.00028). We will try the same method to generate images similar to that of a chosen dataset. For example the CelebA datasets or car models mentioned in the broadcast project.

Many resources are available, for example:

- The GitHub of the original paper: https://github.com/martinarjovsky/WassersteinGAN
- Another GitHub repo that implements WGAN and WGAN-GP: https://github.com/Zeleni9/pytorch-wgan
- A video that also does both: https://www.youtube.com/watch?v=pG0QZ7OddX4
- A blog that describes what he does: https://machinelearningmastery.com/how-to-code-awasserstein-generative-adversarial-network-wgan-from-scratch/ (in TensorFlow and not PyTorch, but that changes very little).

The goal of this project is to rewrite one of these versions to train it on a new dataset (the video may already use CelebA, if you follow the video, try another dataset too). Also, try not to copy the code directly, but to take inspiration from it to create models iteratively, starting with very small and fast to train models (which, on the other hand, will give bad results; for example, start with a very simple game like a single digit of the MNIST game, as in the blog at the last link), and then growing the model progressively.

10 (Large) Question-Answering Language Model

The “as is” application of pre-trained models is not so much the topic of this course, but is now also an expertise requested from data scientists, so this exercise aims at such an application. This topic is only to be chosen if someone wants to focus more on the “engineering” side of the applications and less on the scientific side. Moreover, depending on the details chosen below (Weaviate, Flask) it may be necessary to work on your own computer and not on a notebook like in Colab. In this case, the computer unfortunately has to be quite powerful (normally, no GPU needed, but waiting times can become long).

We will use two pre-trained models:

1. A “text embedding” model, which transforms a given text into a vector $\in \mathbb{R}$, of unitary norm, with the property that the more things two texts have in common (subject, meaning, form, etc.), the closer the corresponding vectors will be ($\| 1 - 2\| \approx 0$).For this, we can choose any model well placed in this ranking: https://huggingface.co/spaces/mteb/leaderboard. It is possible that we need a model with a large enough “Sequence length”, and we choose a model available in open source on HuggingFace (we are not going to have fun paying OpenAI or Cohere or others who appear in the ranking)

2. A large language model. To have a fairly quick set, we will use at most 7 billion parameters, which is the minimum to have acceptable results, but will already be difficult to process. At the moment, the best open source model is Mistral https://huggingface.co/mistralai/Mistral-7B-v0.1 or, a little better, Llama v3.3 https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_3/ . To start, it is better to use smaller models, such as SmolLM: https://huggingface.co/HuggingFaceTB/SmolLM-1.7B (in fact, it is quite good, maybe we can even limit ourselves to this one!)

We will use them to be able to ask questions on a database of documents to an intelligent chatbot, which is called “Retrieval Augmented Generation” (RAG). We can follow for example this guide: https://medium.com/@onkarmishra/using-langchain-for-question-answering-onown-data-3af0a82789ed. The steps to follow are:

1. Find documents that cover a certain topic, that you are interested in. It can be anything: all those long documents that no one reads when you sign a contract with a bank and to which you just want to ask “do I have to pay to withdraw money?”, all the handouts of your favorite Python courses (but for math it works so-so), all the lyrics of all the songs in the world to ask “give me three songs that are about a cat but also about an astronaut”, etc.
2. Partition these documents into small pieces (in any case, smaller than the “Sequence length” of the embedding model), and calculate their embeddings.
3. When asking a question, also calculate the embedding of the question.
4. Find the vectors whose distance between vectors is the least (for some to choose). We can decide to use a vector database (for example, Weaviate, but there are many) to store all the document embeddings; if we do, it will find the most elements closest to us, with optimized algorithms.
5. Formulate a question to be completed by the language model, such as:

Here are the contexts:

. . .
With the above contexts, answer the following question: . . .

(you might have to speak to him in English, though)

As in the article above, we can use Langchain to make some of these queries simpler. Ideally, we would like to have an interface, for example a web page with Flask, that presents this nicely. But otherwise a command line interface will be okay too.

11 U-Net in medical imaging

The goal is to perform a segmentation of an image, to identify what is in each pixel. A source you can use for the code is here:

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

The code should be explained in this video (but I haven't watched it):

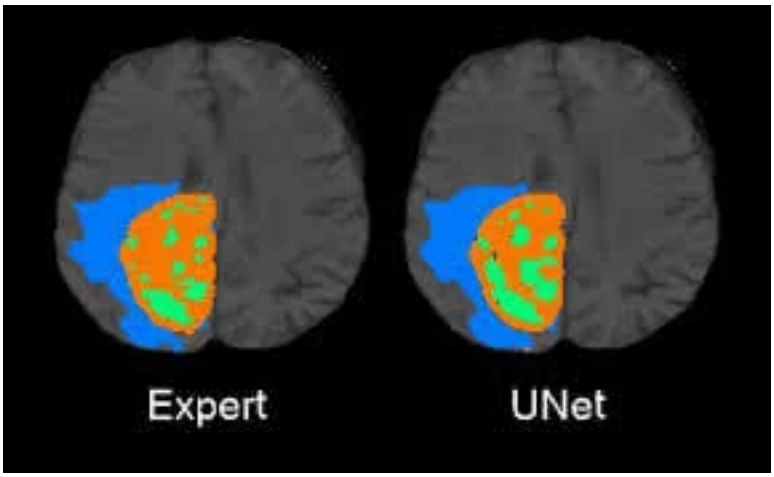
https://www.youtube.com/watch?v=IHq1t7NxS8k

Regarding the data, you can choose a dataset of your preference here: https://paperswithcode.com/datasets?task=medical-image-segmentation. It will need to contain "mask" labels, which mark the areas to be found. It is possible that some work will need to be done on the data: the model in the video is trained on this dataset: https://www.kaggle.com/c/carvana-image-masking-challenge which only contains car / no car masks. I have not looked at the code on loan, but it is possible that it only works in this binary case. In a medical game it is possible that there are several annotated objects (for example: nothing, healthy cell, tumor cell), you will either have to adjust the code to support several classes, or modify the masks to have only a binary problem.

Finally, the repository only contains code in script format. For the presentation, I advise you to run it for example in a jupyter (or better colab), which will allow you to draw images to visualize the initial image, the "ground truth" (the pixels marked according to their class) and the prediction of your model on the same image. The result should be something like the following images (in the first, the masks are superimposed on the image; in the second, the masks are drawn just below):

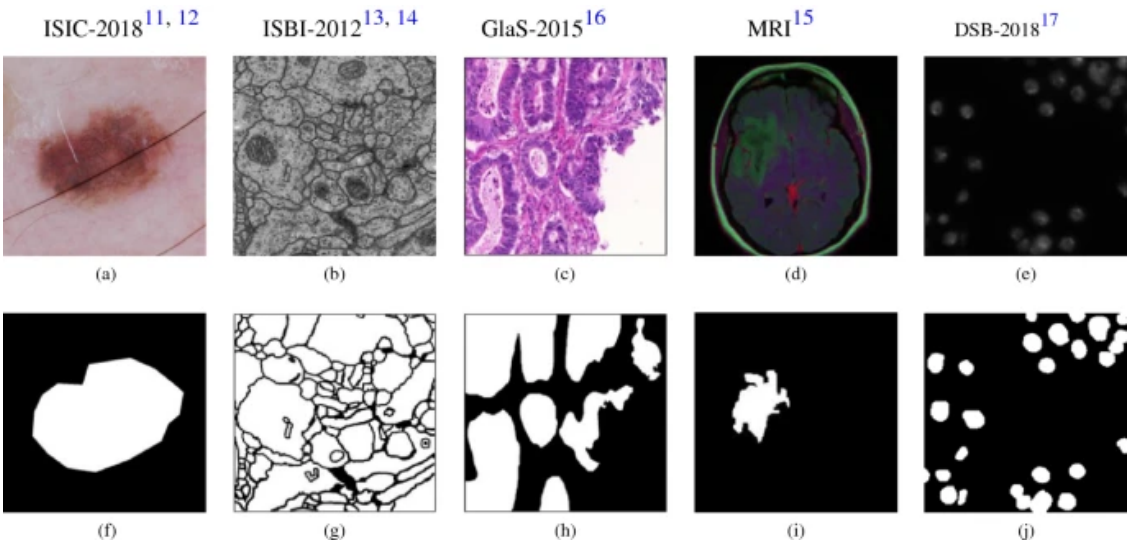
[2]:
```
Picture('unet_cerveau.png')
```

[2]:



[3]:
```
Picture('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)