Advanced Topics in Deep Learning: the rise of transformers CLIP 2023 Project

Ali Noshad
Politecnico di Milano

Reza Paki
Politecnico di Milano

Abstract

In this project, we aimed to implement the contrastive language-image pretraining model (CLIP) at the radiology objects in the context dataset. We approach this problem by implementing the CLIP model both in Keras/Tensorflow and PyTorch because the original CLIP model is initially implemented in the PyTorch framework, so by doing this, we have created a reference for us to compare our model with this reference. Both models are implemented from scratch. Firstly, we directly tried to work the dataset provided by the course. However, since the original dataset of the course was ambiguous and it was very specialized in the medical domain, it was challenging to interpret the results of the model to see that model was working correctly, so instead, we initially built and tested the model based on Flicker8k dataset, which is widely used as a benchmark dataset for these procedures. After we solved the models' problems regarding the correct architecture and training procedure for both datasets (ROCO and Flicker8k), we moved forward with applications that the CLIP can provide. We first tried to use the CLIP for the image retrieval task, in which, given a query, we retrieve the corresponding images that state the query using a pre-trained CLIP. Then we employed the CLIP for the zero-shot learning procedure, in which we use a set of labels, and for each image, we assign the top five closest labels in the means of the similarity. Regarding the Flicker8k dataset, we employed this procedure by using the labels from CIFAR100, and for the ROCO dataset, we used the "concept.txt" file. In both of these applications, we reached a satisfactory performance in both datasets.

1. Introduction

The objective of this report is to document the implementation of a CLIP (Contrastive Language-Image Pretraining) model for the purpose of image retrieval. The task was part of the Hackathon for the PhD course "Rise of the Transformer," and it involved training a CLIP model using a simplified version of the ROCO (Radiology Objects in COntext) dataset. The CLIP model enables the embedding of images and corresponding captions into a shared embedding space. By learning such embeddings, the model can perform tasks such as image retrieval, where the most related image or caption can be identified based on a given query. To complete the task, the implementation of the CLIP model was carried out using TensorFlow 2 and Keras, as instructed in the Hackathon task description. We also write another version based on the pytorch framework from scratch, to ensure the correctness of architecture. The subsequent sections of this report will cover the methodology, data collection and preparation, model implementation, model evaluation, results and discussion, applications and impact, conclusion, references, and optional appendix, as outlined in the initial response.

2. Methodology

2.1. Data preparation

We performed out experiments with two datasets, one was the dataset provided by the course, ROCO, and the other was flicker image dataset. This dataset is recognized as a standard benchmark for sentence-based image description. This dataset, augments the 158k captions from Flickr30k with 244k coreference chains, linking mentions of the same entities across different captions for the same image, and associating them with 276k manually annotated bounding boxes. In this project because of the limits in computing power we used a shorter version of the this dataset called Flicker8k, but also we did some tests on the original one. The reason behind the selection of these dataset for testing our model was, because the the dataset provided by the course is highly ambiguous, and really professional in medical domain, we could not interpreted the results to analyze the performance of the model so we use the mentioned dataset to test the correctness of the model and solve the errors and then applying the working model on the original dataset provided by the course. First for both datasets, we begin by analyzing CSV file which contains the images and the captions informations, since the number of images is quite large loading all of them in the memory is not efficient so we create an custom data generator which take batch size, image filenames and corresponding captions and then based on this we create a address path for loading the image, after loading the images we resized them and then performed normalization, we examined images in different sizes to see the effects on performance of the clip model but finally we selected the 64x64 for the ROCO and 224x224 and 100x100 also tested for Flicker8k, we also considered images with or without normalization. However, finally we use the preprocessing function of the selected model for performing such normalization and discarded this procedure.

2.2. Data splitting

After preparing our dataset, we then performed the data splitting procedure. For selecting appropriate models and hyperparameter tuning, we divided the dataset into 2 parts, 10% of data were considered as the validation set, and the rest of data were considered for training set. The reasons behind the The training data used for training the model. The hyperparameter tuning procedure was done using the validation data, and finally the best obtained results were considered. For credibility of our experiments and accurate tuning of the models, in the first phase of our experiments we defined a seed for the random function which allows reproducibility of the results. This is how we could fully observe the effect of hyperparameters on the models and compare them after we tuned them.

2.3. Image encoder

First we started with basic models such as pure multi-layer CNN. This CNN is composed of 6 “Convolutional” layers, 4 “MaxPooling” layers followed by two “Fully connected” layers with “ReLu” and “Softmax” activations, respectively. However using this approach we couldn’t reach to desired performance even in the training the loss was too high meaning that to model was too simple for this complex task, so we opt to try more complex architecture such as, VGG-16, VGG-19, ResNet50, ResNet152, InceptionV3, MobileNetV2, EfficientNetB0, EfficientNetB4 and EfficientNetB6. These architectures are more complex and more suitable for this task, we should also note that at the end we also tried the visual transformers to examine their performance on the task, and compared it with other architectures. The reason that we started our implementation with the simple CNN or other proven architecture, since the procedure of implementation of the CLIP model is quite complicated and it was really hard to find the errors or mistakes in implementation. This way after creating a clean procedure and checking the correctness of the loss function and the training procedure. After that we created a clean procedure for training, we tried to optimize the performance by testing different models mentioned above.

2.4. Text encoder

As far as we know, a deep learning model does not receive raw data as input. We need a way to encode the text. Here we used the Bert model to tokenize the text input. Inside the TextEncoder class, we are calling the Bert model. This encoding mechanism receives text as input, tokenizes it, and then gives us the embedding vector. The Bert model is a pre-trained model that can provide an embedding vector for corresponding input. Here we have to possibility of using Bert:

- Not only using it as a tokenizer but also using its provided embedding vector.
- Using it just as tokenizer.

While using option A, we can set the trainability of the Bert model to True or False. If we put this as True, the model weights will be updated for incoming new data. However, if we put this False, the weights will not be updated. We compared both possibilities and understood that while this parameter is False, our loss function does not converge to optimality. This way, we faced underfitting, and our model needed to be stronger in learning our data. However, while we are putting it as True, the model converges and seems able to learn the data. In doing so, we selected this parameter as True for the rest of the experiments.

In option, A output of the encoder is pooled_output, which is a fixed-length representation of the entire input sequence. By comparing validation loss in both options A and B, we conclude that using embedding vectors of Bert instead of just using tokens is a better choice, and we moved forward with our experiments based on this choice.

2.5. Transfer learning

After making multiple modifications in our CNN architectures, we realized that the accuracy could not experience further improvements. So, we decided to use pre-training weights with the selected architectures. Regarding this, we started using some basic architectures and analyzed more advanced methods as we moved forward. We started with VGG16, VGG19, ReNet50, ResNet152, InceptionV3, MobileNetV2, EfficientNetB0, EfficientNetB4 and EfficientNetB6. We used these architectures pre-trained on the imagenet, and we just modified the top of the network in all the pre-trained architectures and optimized them to the task we are trying to solve. In the top we experimented with several layers such as “GlobalAveragePooling”, “GlobalMaxPooling”. Furthermore, we analyzed the behavior of these architectures by manipulating the number of trainable layers and tuned these hyperparameters and model configurations on the validation set. Training with different batch sizes was also tested and the best ones were 16, 8, and 32. However, due to the error of resource limitation we could not increase the batch size from 32. With regards to normalizing images in this part, we imported and used the specific preprocessing functions from tensorflow library and placed it in the custom creating “Imagedatagenerator” function, and we preprocessed the test images using these functions as well. Moreover, to further improve the model performance we found out only using pretrained model on the imagenet is not enhance the models performance as much as we thought because the task that we are trying to do is very specific and using only the pretrained weights of imagenet which captures the features of the imagenet dataset cannot be helpful very much, so we tried to used the pretrained weights of the imagenet as base weight for image classifier for classifying the images based on the labels provided. This way we fine tune the weights of the image encoder on the dataset by performing classification, and using those weights for image encoder in the clip model. Because of this way, the clip model will converge faster because the image encoder is already tuned on the dataset to capture the prominent and meaningful features of the dataset.

2.6. Projection Head

Projection head is defined to transform the image embedding and text embedding to the same embedding dimension. The projection head is composed of a Gelu activation, a Dense layer, a Dropoout, and a Add layer which adds resulted output from previous layers and the projected embedding after the dense layer.

2.7. Loss function

This part was tricky for us. First, we tried to use the loss function used in the Keras dual encoder model, which takes the image and text embeddings over a defined batch size. Then we can calculate the similarity matrix of the embeddings by performing the dot product until now; this is the same as the one described in the CLIP paper, but calculating the target in this loss function is quite different; we perform dot product both for image and text embeddings to themselves, ideally in this case we came up with matrix with diagonal have the value of 1 and the rest is the similarities of images together and the similarities of the texts together, we add these two matrices. Then we divide them by two for normalization. Then we feed these matrices to a Softmax; this is how the targets are made. Then, we applied the categorical cross-entropy to calculate the loss for both images and texts and divide that by two. This procedure worked very well for the Flicker datasets both by the model we implemented based on the Tensorflow/Keras framework and the model we implemented based on the Pytorch framework. With this procedure on the Flicker, we managed to optimize our model faster, and we managed to create a more robust model in both of the frameworks (Tensorflow and Pytorch). However, when we employed this loss function on the ROCO dataset, we encountered the huge problem. The reason is that the similarities are really close in the ROCO dataset, so using the above approach will result in really close similarities in the target matrix (it is not correctly built, because all the labels are the same). We cannot train the model properly; in the training, the similarity matrices result in the same value for all of the images and the corresponding text. Hence, the training loss becomes steady after a few iterations. To address this issue, we opt to the original clip loss function in which, as it described in the clip paper, we define the target matrices as *np.arrange(n)* and used the sparse categorical cross entropy as a loss function. This way, the targets differ, making the training more meaningful.

2.8. Training procedure

After integrating all the above parts into a single CLIP model, we continued the training. We created a custom training loop for the training procedure in which we trained the model for five epochs with the previously mentioned batch sizes. One of the critical elements here that affected the training process significantly was the learning rate, at first we started at 0.001; however, this corrupted our training procedure with this learning rate in both the reference model implemented in the PyTorch and the model in Keras/Tensorflow the training was shortly converged to local minima and attained the worst performance based on the obtained result from the image retrieval procedure. However, after testing multiple values for the learning rate, we reached an optimal value of 0.00001, which allowed a more robust training and better performance in terms of image retrieval. Of course, we tried even more lower values, but it only resulted in the slow training procedure and finally reached to the same results as the selected value.

2.9. Image retrieval

For the purpose of image retrieval, using the pre-trained model CLIP on the Filcker8k, and ROCO training dataset, we extracted the image embeddings from the validation dataset using the pre-trained model. Then we provided a query, as example in the Filcker8k dataset we used the query *”dogs on the grass”* and for the ROCO dataset since the performance was as much as accurate like on the Filcker8k we used the key words or short sentences such as *”pericardial tamponade”, ”angiography”, ...*, as it can be demonstrated by the figure 1 and figure 2 on the Flicker8k dataset the two models (both implemented by Keras/Tensorflow and Pytorch) result in similar meaningful pictures for the query mentioned above. We should also note that the model implemented with pytorch framework was a bit more accurate, which it can be the results of longer training process. Regarding the ROCO dataset, we tested more queries and used keywords to compare both models in implemented. Figures 3, 4, 5, 6, 7 and 8 illustrate and compare the performance of the models implemented both by the PyTorch framework, and keras/Tensorflow, respectively.

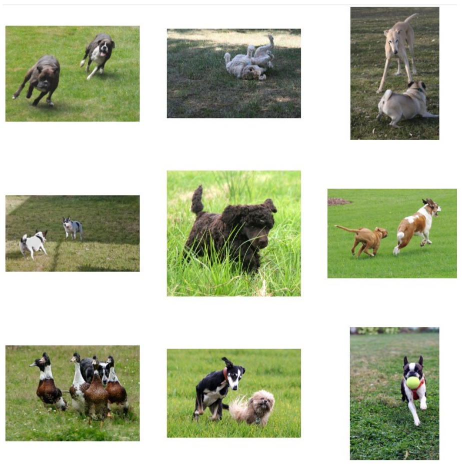
Figure 1. Reference model implementation on PyTorch from scratch; image retrieval for a give query : *"dogs on grass"*

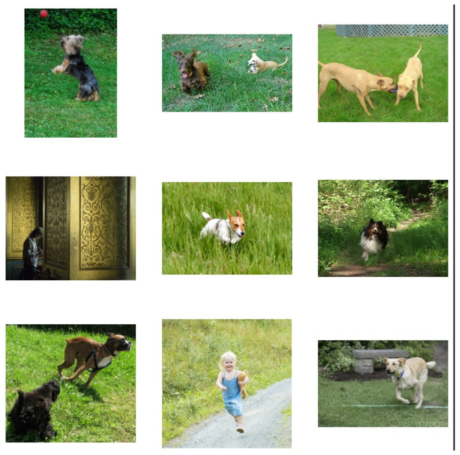
Figure 2. Tensorflow/Keras implementation from scratch; image retrieval for a give query : *"dogs on grass"*

2.10. Zero-shot learning

In the next step of our project, we tried to implement zero-shot learning. In this approach for the Flicker8k dataset, we tried to use another dataset called CIFAR100, in which we extracted all of the labels. Then we created a sentence for each label in the dataset: *"A photo of a [label]"*; we fed these sentences to the pre-trained clip on the Flicker8k dataset to calculate the embeddings. Furthermore, we also calculated the embedding of the images presented in the validation dataset, and then by the dot product,

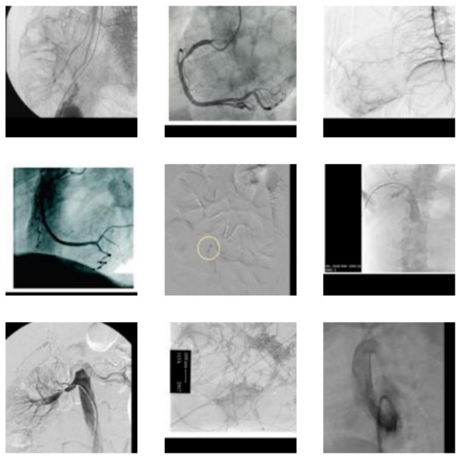
Figure 3. Reference model implementation on PyTorch from scratch; image retrieval for a give query : *"angiography"*

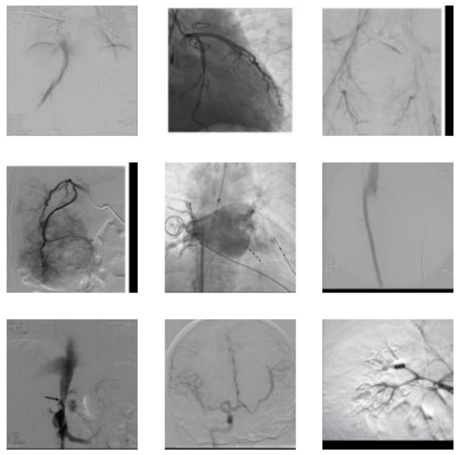
Figure 4. Tensorflow/Keras implementation from scratch; image retrieval for a give query : *"angiography"*

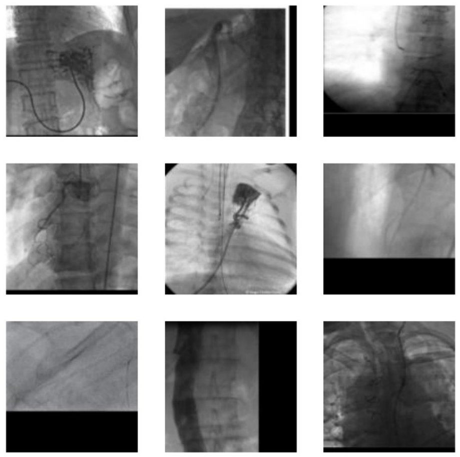
Figure 5. Reference model implementation on PyTorch from scratch; image retrieval for a give query : *"balloon catheter"*

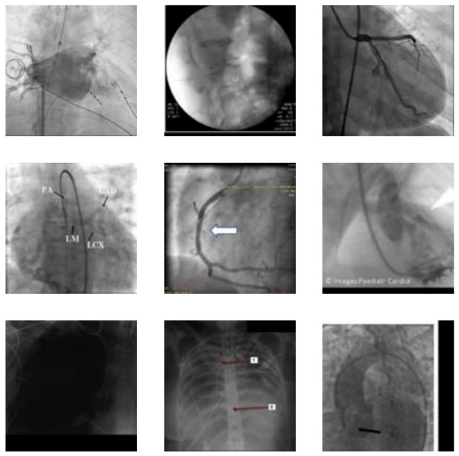
Figure 6. Tensorflow/Keras implementation from scratch; image retrieval for a give query : *"balloon catheter"*

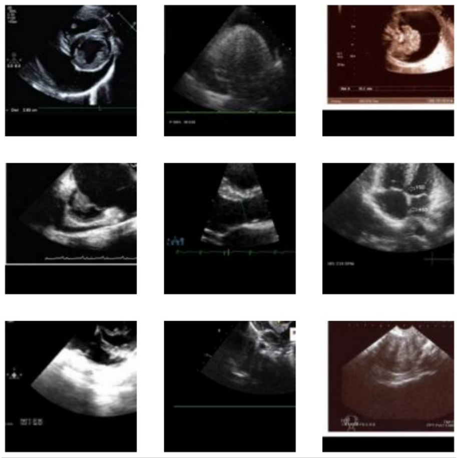
Figure 7. Reference model implementation on PyTorch from scratch; image retrieval for a give query : *"pericardial tamponade"*

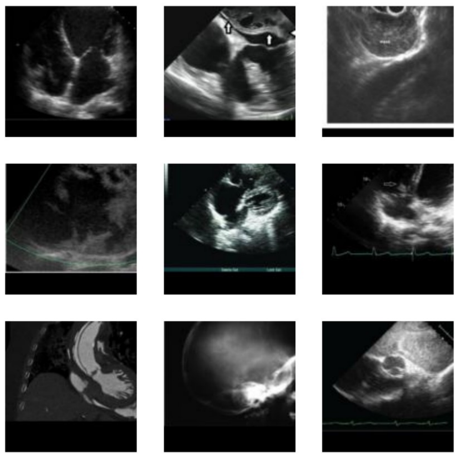
Figure 8. Tensorflow/Keras implementation from scratch; image retrieval for a give query : *"pericardial tamponade"*

we calculated the similarity between the image embeddings and text embeddings and took the highest probable label for each image in the validation dataset. However, this procedure was different on the ROCO dataset, in which we calculated the embeddings for the labels provided in the file "concepts.txt," which represent the category/type of each image in a general way, then similar to another dataset; we also calculate the embeddings for the images presented in the validation set, we calculated the similarity between the images in the validation set and concepts, and we showed the most probable concept for each image in the validation set. Like the previous section, we again tested the model implemented in PyTorch and Keras/TensorFlow frameworks.

Regarding the Flicker8k dataset using the model implemented in PyTorch, we achieved the following results in zero-shot learning. However, some labels were present in the CIFAR100, which was not the Flicker dataset, so we experienced some inaccuracy in finding the labels of some pictures. Regarding the model implemented in PyTorch, figure 9 illustrates the zero-shot learning results, in which the model managed to find accurate labels or close related labels for a given image. Figure 10 illustrates an inaccurate example of this model; in this example, we can see that the model labeled a dog as a rabbit. Of course, the image is similar to the image of the rabbit; we assumed that the problem results from the existence of some labels in CIFAR100 which are not present in the Flicker8k dataset. Figure 11 demonstrate the closely labeled images using the model implemented in Keras/Tensorflow, similarly figure 12 illustrates the inaccurate example labeled by the model due to the problem described above. The procedure was littel different regarding the ROCO dataset. First, similar to the procedure we adopt Flicker8k dataset we performed zero-shot learning in the whole set of labels presented in the "concept.txt" (8375 labels). Figure 13 illustrates the attained the results from the model implemented in PyTorch.

However, during some analysis on the dataset and the concepts we discoverd the some of the labels are repeated many times in the images in the dataset, so based on this and to gain better performance in zero-shot learning we tried to consider the most frequent labels for performing zero-shot learning. In particular, we tried to use 10 most frequent labels in the validation set, the labels are the following: 1) X-Ray Computed Tomography (C0040405): 2632, 2) Plain x-ray (C1306645): 2424, 3) Magnetic Resonance Imaging (C0024485): 1428, 4) Ultrasonography (C0041618): 1119, 5) Chest (C0817096): 808, 6) Angiogram (C0002978): 616 7) Abdomen (C0000726): 565, 8) Bone structure of cranium (C0037303): 506, Lesion (C0221198): 391, Axial (C0205131): 348. Our validation set contains 8328 images, and if we some the labels above we can see that at least one of the these most frequent labels are present in nearly all the

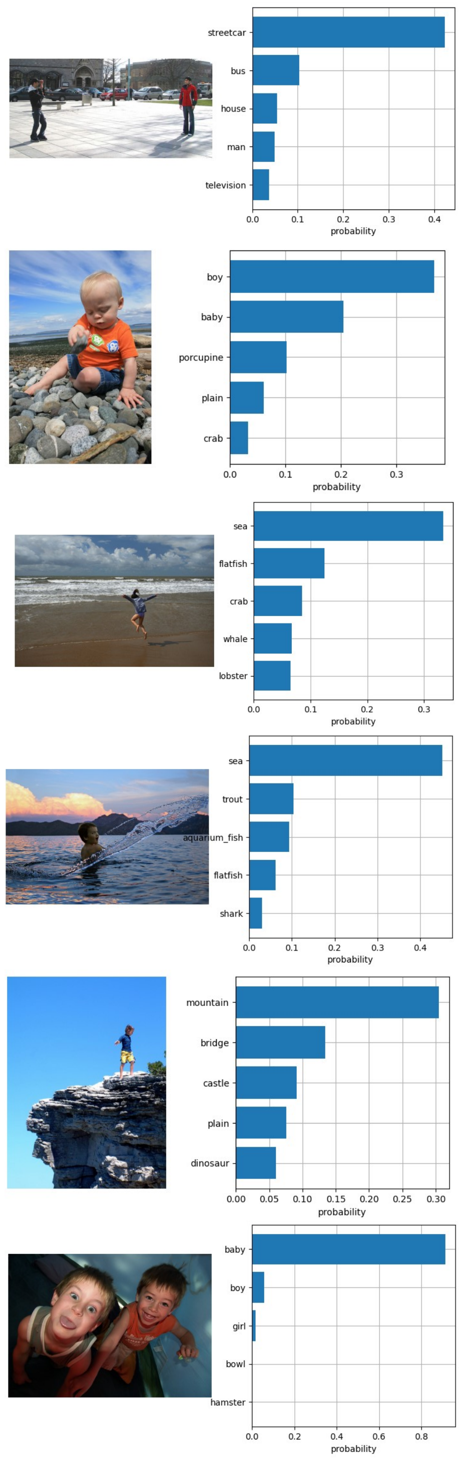

Figure 9. Reference model implementation on PyTorch from scratch: zero-shot learning for CIFAR100 labels (accurate examples)

images, so implementing the zero-shot learning with this labels can somehow increase the performance, and prevents misdiagnosis with other labels which are not very familiar.

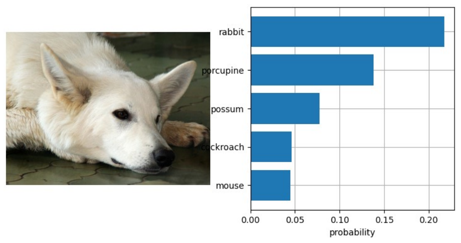

Figure 10. Reference model implementation on PyTorch from scratch: zero-shot learning for CIFAR100 labels (inaccurate example)

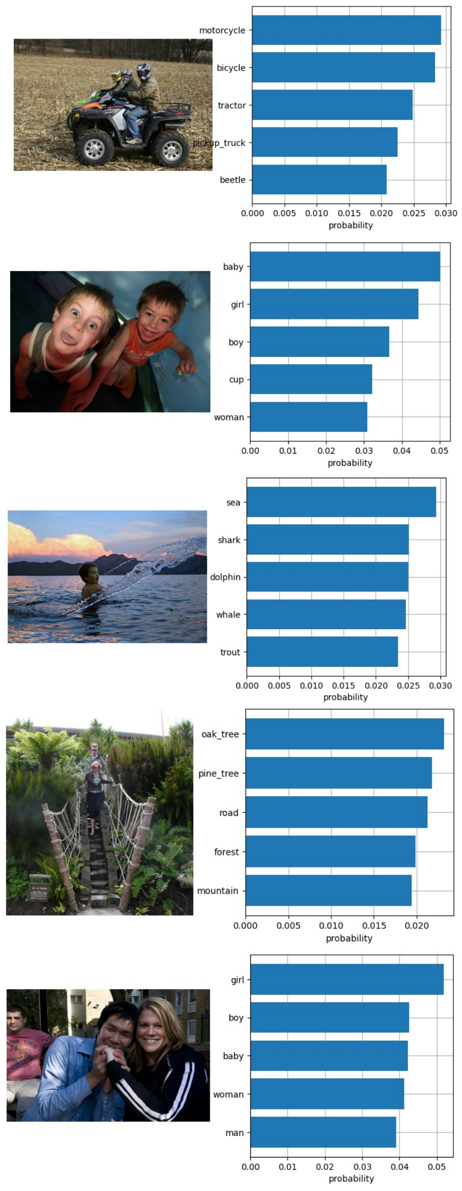

Figure 11. Tensorflow/Keras implementation from scratch: zero-shot learning for CIFAR100 labels (accurate examples)

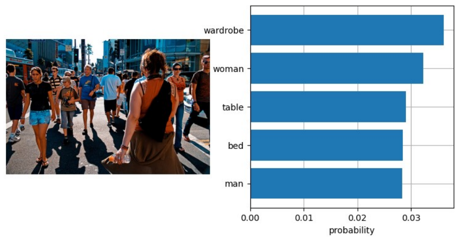

Figure 12. Tensorflow/Keras implementation from scratch: zero-shot learning for CIFAR100 labels (inaccurate example)

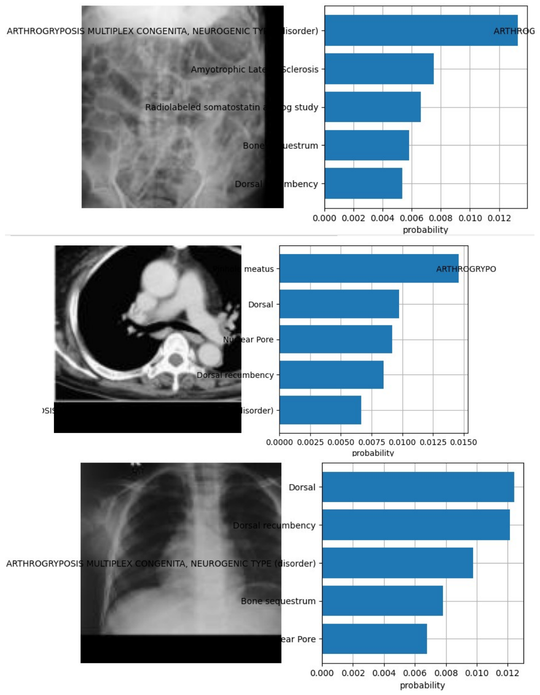
ARTHROGRYPOSIS MULTIPLEX CONGENITA, NEUROGENIC TYPE (disorder)
Amyotrophic Lateral Sclerosis
Radiolabeled somatostatin analog study
Bone sequestrum
Dorsal recumbency
probability
meatus
Dorsal
Nuclear Pore
Dorsal recumbency
probability
Dorsal
Dorsal recumbency
ARTHROGRYPOSIS MULTIPLEX CONGENITA, NEUROGENIC TYPE (disorder)
Bone sequestrum
probability

Figure 13. Reference model implementation on PyTorch from scratch: zero-shot learning for Concept labels (accurate examples)

Figure 14 illustrates the zero-shot learning results for top 10 most frequent labels. However, showing this results we cannot estimate the model performance so we decided to check our results with the true labels presented in the dataset, so as results of this comparison we created a confusion matrix shown in the figure 15.

Similarly, we implemented the same procedure for the model implemented in the Keras/Tensorflow. First, we applied the zero-shot learning for all the labels presented in the "concept.txt", then, we considered the most frequent labels, figure 16 demonstrates the results of the zero-shot learning on the top 10 most frequent labels in validation set.

2.11. Systematic analysis

In section, we tried to understand the behaviour of the clip by modifying the configuration of the CLIP. We tried to train the clip using different batch sizes such 16, 32. Then

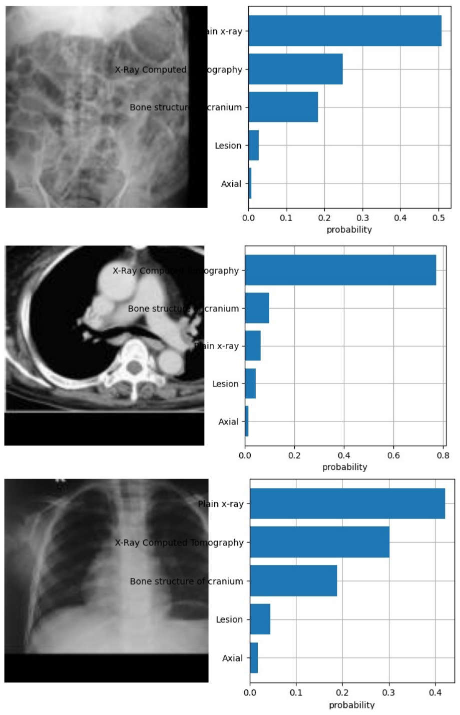

Figure 14. Reference model implementation on PyTorch from scratch: zero-shot learning for top 10 frequent Concept labels (accurate examples)

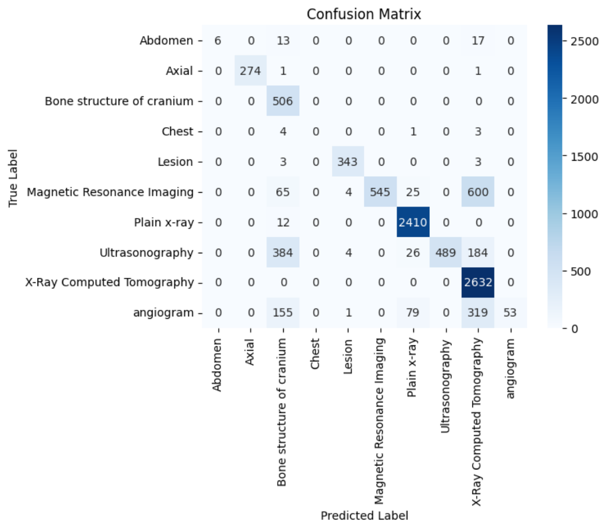

Figure 15. Reference model implementation on PyTorch from scratch: zero-shot learning for top 10 frequent Concept labels confusion matrix

we tried to modify the projection head by changing the embedding dimension. We trained several methods and gathered loss values for each iteration. Here we only plotted loss for three models like Resnet50 with different dimensions and batches. Training loss converges in all cases and has smooth behavior during different iterations. The most

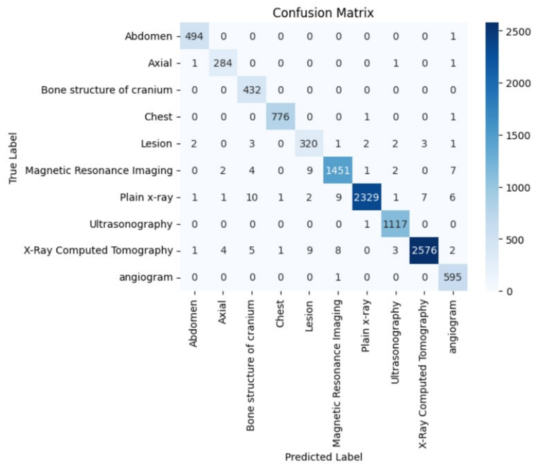

Figure 16. Tensorflow/Keras implementation from scratch: zero-shot learning confusion matrix for top 10 frequent labels in validation set

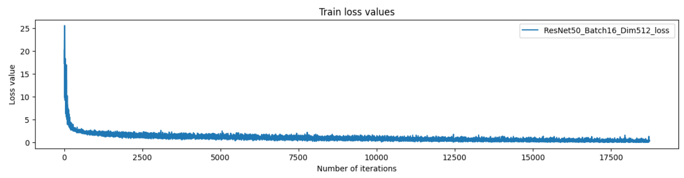

Figure 18. Tensorflow/Keras implementation from scratch: Training loss curve for batch 16 and dimensions 512

important this for us is validation loss. As much as we increased the dimension of the Resnet50, we had lower validation loss. The second critical parameter is the batch size. We compared both batch sizes of 32 and 16. In both cases, in the first three epochs, loss decreases gradually and then increases. As far we can see in plot 19, batch size 16 has a lower validation loss with respect to 32. By doing so, we do the rest of the experiments with this batch size (Figures 17, 18, and 19 are demonstrate the results).

Furthermore, as our project's last step, we applied a pre-trained visual transformer as the image encoder to see the difference. However, I should note that this architecture is complex and computationally expensive, and it takes a long time to train. Compared to other architecture, such as Resnet50 or Xception, the convergence of the loss did not improve much. Regarding this, for the final model, we used the Resnet50.

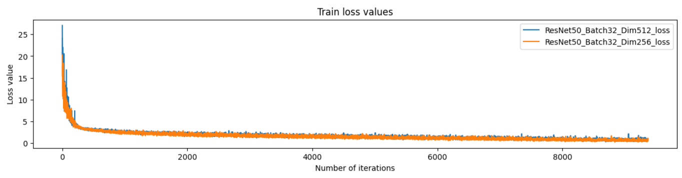

Figure 17. Tensorflow/Keras implementation from scratch: Training loss curve for batch 32 and dimensions 512, 256

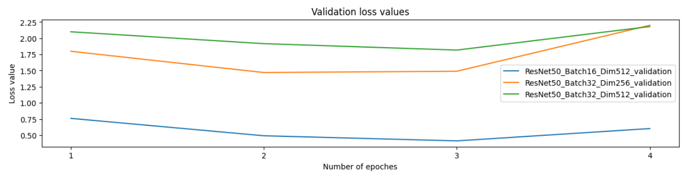

Figure 19. Tensorflow/Keras implementation from scratch: Validation loss curve for batch 16, 32 and dimensions 512, 256