SAiDl - Induction Assignment 2023

Yash Bhisikar - 2021A7PS0483G

This is my report for the induction assignment. I have attempted 2 of the tasks, Core ML and CV, and this was a great learning experience [with ups and, more often than not, downs :)], and this report documents how I went about this, what I understood, my approach and what I learnt. I have also included the implementation status and the ToDos at the end of the report of every task. Here we go!

The Core ML Task

This was a really fun task; I had already implemented a classifier for the CIFAR-100 dataset when I was first doing the CNNs in the Andrew NG course, so it was a good opportunity for me to go through my old work again and play around with different techniques to make the model better.

I read the description of the task, and the first thing I did was google them and try to understand why there is a need for different implementations of the Softmax function.

Vanilla Softmax was fairly simple, and it didn't take me much time to code it up. I had happened to use it in my earlier implementation too, so it was smooth sailing. However, knowing that I had to implement different variations of it, I took this opportunity to go through it again by watching videos and reading articles.

Implementation status: code written and working. However, the model is overfitting to the data as training loss is decreasing while validation loss and accuracy remain constant over multiple epochs.

ToDo: Add a dropout layer in the fully connected layers, tune the hyperparameters, and use different training metrics (precision, recall, etc.), which are available as PyTorch built-ins.

I tried to understand the idea behind Gumbel, and admittedly, the math seemed daunting at first. I went in the wrong direction by trying to understand what the formula is doing instead of trying to understand what exactly the approach is trying to do. However, once I realised that after a few hours, it was fairly simple. In cases where you have a discrete representation (e.g., language and text generation), our nn (Neural Network) is going to output samples randomly from this non-continuous distribution. The problem lies in the backward pass, as stochastic gradient descent won't work on this (you can't calculate the derivative of non-continuous functions). The main idea behind Gumbel-Softmax is to randomly add a noise sampled from a continuous distribution (the Gumbel Distribution, hence the name). This transforms the discrete distribution into a continuous distribution, and to avoid any new bias added, the new distribution should be centred around 0. There is a lot to the math and implementation details behind this. However, this was enough for me to temporarily look into how to use it.

Implementation status: The code is written; however, a very weird bug came up in training (since the code is exactly the same as that of Vanilla Softmax, but that code is working), so have to patch that up.

To Do: fix the bug, play around with hyperparameters, and read up on Gumbel since the exact math still eludes me.

Hierarchical softmax took me some time to figure out why exactly we were doing it the way we were. I don't have any experience with NLP, so it was tough for me for a bit to understand a lot of the terms and concepts involved. After reading several articles and watching a lot of videos, I was finally able to understand what vector representations of words are, why we need the previous words in a sentence to be able to predict the next word, and why it becomes a recursive process. Only then did the use of the binary tree and the whole procedure start making sense. The binary tree explanation was mostly fine, and given the assumption that we already have the tree constructed with the leaves being the words of our corpus arranged according to their frequency, then encoding the directions (left and right) with a binary vector and using that to learn the vector representation of every internal node in the tree, The speedup in the computation, from O(V) to O(log V), is very impressive. However, it relies on having the frequency distribution of the lexicon in the corpus. My main problem was how to construct the binary tree itself, and that's what I struggled with the most. My intuition told

me that the words in the corpus are analogous to the classes we are trying to predict; however, how to assign a vector node to a binary tree and how to learn the vector representations was something I struggled to find much information on. At the time of writing this, I found out about the use of Huffman Encoding. I couldn’t implement the code. However, I’m sceptical as to how to use it exactly here since the CIFAR 100 dataset has equal frequencies of all classes. This can make the binary tree balanced, which is usually desirable, but it defeats the purpose of using hierarchical softmax. By assigning lower depths to the most frequent words or classes and higher depths to rarer words or classes, we save up on a lot of computation, and probability calculation reduces merely to calculating the probability at each node of whether to take a left or a right turn and traversing the tree to reach the target leaf node.

Implementation status: didn’t implement it

ToDo: implement it

My first impression of adaptive softmax was that it's another speedup technique used often in NLP contexts and is suited for working on GPUs. It maintains the accuracy of actual softmax and gains efficiency. I know that its main principles are similar to those of hierarchical softmax: that it keeps a very short list of frequent words, that it gains a time advantage by leveraging the parallel processing power of GPUs, and that it divides the words into clusters on the basis of their frequency. Again, I am doubtful of how this would work when the frequency of all the words and classes is the same. I found a built-in Pytorch function to implement this, but I happened to get terrible accuracies (worse than random guessing) after multiple epochs and a very noticeable increase in training time. It’s evident that I still don't completely understand it and have to read the paper properly to understand it.

Implementation status: code written, but there is still a lot left to be done.

ToDo: Reading up on the paper and making the necessary connections.

This is my report for the Core ML task.

The CV Task

I started the induction assignment directly with this task, as it seemed the most interesting, and ended up spending the most time on it. Doing this task involved a lot of reading, looking up resources, and reading papers (which I strongly think that this task has taught me to a decent extent).

Not directly related to the task: If I plan to conduct research, I need to improve my ability to read research papers, and this task has helped me realise this. Since I will frequently be reading multiple papers for a given project, understanding how and what information to obtain within the time constraints will be an extremely useful skill.

I started off the task by experimenting with CLIP and trying to understand what it does. Using starter code available on OpenAI's GitHub repo, I tried to figure out how to use the model to get the embeddings that the decoder was supposed to be trained on. I also experimented with using different phrases that didn't match the image context and using multiple phrases that described the image. In the former case, the probabilities of a matching phrase remain insignificant if no close match is found. In the latter case, the matching phrases end up having probabilities very close to each other, and they add up to make up for a major proportion of the probabilities. I also read up on CLIP's zero-shot classification ability and what exactly made it possible to do so.

I then went ahead with reading up on and understanding the UNet model, which is popularly used for semantic segmentation. At this point, I had not read the CLIPSeg paper, and thus I assumed the task referred to the decoder used in UNet instead of the transformer-based decoder. I tried to understand the encoder-decoder pipeline used in a fully convolutional neural network and what exactly CLIP is replacing. In UNet, the encoder is tasked with extracting high-level features from the image at a low resolution and then feeding these extracted features to the decoder. The decoder then uses transconvolution to upsample the images to generate the final output. This output can be of the form of polygon coordinates or a binary segmentation mask, where pixels with 1 denote region of interest while 0s should be ignored. I assumed the CLIP embedding replaces these extracted features; thus, I read up on the source code of UNet implementations to figure out how to train a decoder using this embedding. I also skimmed through the UNet paper, hoping for some implementation-specific information that I couldn't gather from reading up on sources.

After having a decent idea of what to do next, the next question was to decide the size of the decoder. I was doubtful of the size of the decoder to use, and I couldn’t find any implementation that was trained on the PhraseCut dataset. So I texted one of the POCs regarding this, after which I found out that we were expected to use a transformer-based decoder for this task. I was then directed to read the CLIPSeg paper to understand what the task was.

The next 3–4 days I spent my time learning about transformers from "Attention is all you Need" and the CLIPSeg paper. I did not know what transformers were or in what context they were used, so a lot of my time was spent understanding what they are, how they work and why we are using them instead of a CNN-based decoder. I also spent time wrapping my head around self-attention and the math behind it. The main reason behind using an attention mechanism is to encode the information of elements present at different locations in a sequence. By allowing access to all elements of an input sequence, we get a better understanding of the context and information held by other elements. And since we have multi-modal inputs (images and text phrase), my plan was to use cross-attention. I couldn’t find any PyTorch builtins, so I decided on implementing my own cross-attention module, which served as a nice excuse to understand the roles of query, key, and value vectors.

We use the input vector and apply a linear transformation (the weights of these matrices are learnable) to obtain the required vectors. The key vector learns the relationship of the input to the other elements of the sequence; the query is compared against the key vectors to measure its relevance; and the value vector determines the contribution of the input to the final output. Implementation-wise, the final product is scaled so as to avoid the product of weights becoming too large.

I next went on to working with the dataset. The API had functions for dataloader and how to exactly use them, but I was having trouble using them (the documentation had no mention of how to use the API functions on a subset of the dataset since all the official subsets of the complete dataset had dedicated config files that were missing from the given dataset). I instead tried to write my own dataloader from scratch and spent the whole day reading the source code for the loader. I tried to make the

necessary changes but couldn't get it to work. I finally switched back to the original dataloader and, with some help from a POC, got it working.

The CLIPSeg paper talks about using activations from the 3rd, 6th, and 9th layers of the CLIP encoder as skip connections to the decoder. The last layer's activations are used to condition the output embedding using FiLM. I didn't know what FiLM was, so my plan was to obtain the activations first and then go ahead with figuring out the conditioning. To obtain the CLIP's activations, I needed to know the architecture of the model and had to manually write code for a forward pass. For this as well, I ended up reading the source code for CLIP and had to condense it to get the activations. Using `encode_image` and `encode_text` functions, I get the embeddings only but not the activations. So I copied portions of code, ran the forward pass, and extracted the activations. I attempted to write the decoder, but it's still a work in progress and is not complete. I also have to set up the model for training and evaluation but the code for it wasn't written yet. At this point, I realised that I also had to do the Core ML task and only had a few days, so I paused it for the time being. However, I couldn't find the time to complete it.

Implementation Status: Encoder ready, extracted the activations, and got the embeddings

ToDo: Write scripts for training and use api methods for evaluation, figure out how to perform conditioning, and read the FiLM paper.

This was my report for the CV Task.

This report ended up being much longer than what I had expected.

Thanks for reading!