SAiDl - Induction Assignment 2023

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. Here we go!

▼ The Core ML Task

This was an enjoyable 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 an excellent 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 understand why the need to have different implementations of the Softmax function is required.

Vanilla Softmax was simple, and it didn't take me much time to code it up. I also used it in my earlier implementation, so it was smooth sailing. However, knowing I had to implement different variations, I went through it again by watching videos and reading articles.

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

ToDo: Add a dropout layer in the fully connected layers, and tune the hyperparameters, 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 needed to understand what the formula was doing instead of what the approach was trying to do. However, once I realised that after a few hours, it was pretty simple. In cases when 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 further 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 look out for how to use it temporarily.

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

ToDo: 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 we were doing it exactly as we did. I have yet to gain experience with NLP, so it was tough to understand many of the terms and concepts involved. After reading several articles and watching many videos, I finally understood what vector representations of words are, why we need the previous comments in a sentence to predict the next term, and why it becomes a recursive process. Only then did the use of the binary tree and the whole procedure make sense. The binary tree explanation was mainly satisfactory, 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 on it. 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 about how to use it exactly here since the CIFAR 100 dataset has equal frequencies of all classes. This can balance the binary tree, which is usually desirable but defeats the purpose of using hierarchical softmax. By assigning lower depths to the most frequent words/classes and higher depths to rarer words/categories, 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 about adaptive softmax was that it's another speedup technique used often in NLP contexts. It is suited for working on GPUs. It maintains the accuracy of actual softmax and gains in efficiency. I know that its main principles are similar to hierarchical softmax and that it keeps a concise list of frequent words. It gains a time advantage by leveraging the parallel processing power of GPUs, and it divides the terms into clusters based on their frequency. It is still determined how this would work when the frequency of all the words/classes is the same. I found a built-in Pytorch function to implement this. Still, I got terrible accuracies(worse off than random guessing) after multiple epochs and a 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 still a lot left to be done

ToDo: Reading up 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 I spent the most time on it. Doing this task involved a lot of reading, looking up resources and reading papers(which I believe 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 read multiple documents for a given project, understanding how and what information to obtain within the time constraints will be a handy skill.

I started the task by experimenting with CLIP and trying to understand what it does. Using the starter code available on their 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 phrases that don't match the image context and using multiple words that describe the image. For the former case, the probability of the matching expression remains insignificant if no close match is found. For the latter case, the matching word has possibilities relative to each other and then add up to make up for a significant proportion of the options. I also read up about CLIP's zero-shot classification ability and what exactly made it possible to do so.

I then went ahead with reading up 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 CLIP is replacing. In UNet, the encoder is tasked with extracting high-level features from the image at a low resolution and then feeds these extracted features to the decoder. The decoder then uses trans convolution to upsample the images, and I assumed the embedding replaces these extracted features. The segmentation regions can be represented as polygon coordinates or a binary mask, with 1s being the region of interest while 0s were to be ignored. I read 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 I couldn't gather from reading up from sources.

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

For the next 3-4 days, I learned about transformers from “Attention is all you Need” and the CLIPSeg paper. I did not know what transformers were or what context it was used in, so much 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 time wrapping my head around self-attention and the math behind it. The main reason behind an attention mechanism is to encode the information of elements in different locations in a sequence. By allowing access to all aspects of an input sequence, we better understand the context and information held by other factors. And since we have multi-modal inputs(image and text phrase), I planned to use cross-attention. I couldn't find any PyTorch builtins, so I implemented my cross-attention module, which served as an excellent 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 critical 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 to ensure the development of weights is manageable.

I next went on to work with the dataset. The API had functions for data loader and how to exactly use it, but I was having trouble using it(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 which were missing from the given dataset). I instead tried to write my own data loader from scratch and spent the whole day reading the source code for the loader, and tried to make necessary changes but couldn't get it working. I finally switched back to the original data loader and got it working with some help from a POC.

The CLIPSeg paper talks about activating activations from the CLIP encoder's 3rd, 6th and 9th layers as skip connections to the decoder. The last layer's activations condition the output embedding using FiLM. I didn't know what FiLM was, so I planned to obtain the activations and determine the conditioning. I needed to see the model's architecture to get the CLIP's activations and manually wrote code for a forward pass. For this, I also read 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 the code, ran the forward pass and extracted the activations. I attempted to write the decoder, but it's still incomplete. At this point, I realised I also had to do the Core ML task and just had a few days, so I paused it for now. However, I couldn't get the time to complete it.

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

ToDo: Train and eval script, figure out how to perform conditioning and read the FiLM paper

This was my report for the CV Task.

▼ The RL Task

I started this task by reviewing the David Silver videos and following Sutton and Barto. It took me a lot of time since I was doing RL for the first time. A lot of time was just spent on the basic concepts themselves, such as understanding what MDPs are, why they are helpful here, How dynamic programming is used for policy improvement, and then in value estimation, then methods such as using Monte Carlo sampling, bootstrapping and also about the temporal difference. Though I still haven't got the hang of them perfectly, I felt that I knew enough to start reading the paper and making sense of it.

My work in the CV task familiarised me with Transformers, so reading the paper and understanding the methodology took little time. I again reviewed sequence modelling and decided to proceed with an LSTM-based architecture.

The paper used D4RL benchmarks as the dataset for training, so I decided to go ahead with it. However, I faced problems using it since Hopper requires the MuJoCo physics engine and, thus, a workaround for using it for free. All the methods I found ran into difficulties again, such as Nvidia driver issues on my local machine or versioning issues on Colab. Eventually, I did end up finding a procedure which worked on Colab.

Model details: The Model has an LSTM backbone. I used two layers, and the model takes a sequence of (states, actions, returns-to-go) tuples of length k (k = 30 in my case, inspired by the context length of the transformer used in the paper) and the outputs of the actions to be taken to obtain the said rewards. I normalised the rewards and actions using a standard scalar approach. I wrote a custom dataset to sample a k timestep trajectory every time a particular episode is tested. I embed the three inputs to bring them to the same size and feed it to the LSTM, which returns the output and the hidden state. I embedded the LSTM output and then used it to predict action. Since Hopper actions range from -1 to 1, I applied a tanh layer at the end. The hidden state was saved separately so I could use its previous confidential form the next time the episode is sampled to feed the LSTM. I used Huber Loss since it deals with the skewed domination of outliers in square loss, which I thought would exist plentiful in the dataset since it's just recorded episodes and there's a good chance of finding non-optimal actions. I trained the model for 400 epochs, but hyperparameter tuning is yet to be done.

Implementation status: completed

ToDo: perform more hyperparameter tuning and model architecture, changing the number of timesteps and checking its performance on the gymnasium.

Thanks for reading!