SAiDl Induction Assignment 2024

This is my report for the induction assignment. When the assignment was announced I was slowly and steadily following the cs229 course from Stanford. Still, thankfully the announcement made me fast-track my progress in deep learning and artificial intelligence.

When I first read the questions, I honestly could not make much sense of them questions so I could not judge how easy or what would be better to attempt, so I first started to attempt the Pruning and Sparsity Task.

- Pruning and Sparsity

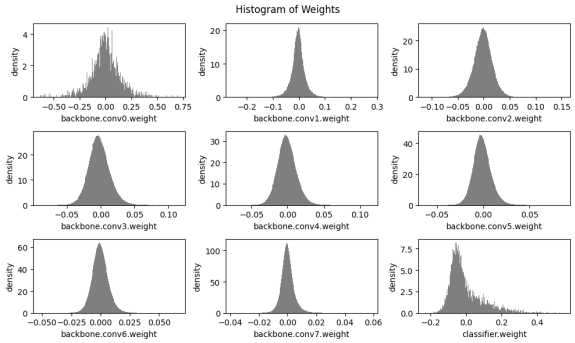
Before beginning the task I had watched a few lectures of cs229 and at that point, I had finally wrapped my head around how any form of intelligence is just math. I watched the first lecture of MIT's6.5940 (as suggested by the task) and was left clueless, as I did not understand convolutional neural networks and still did not have a strong grasp of loss functions, optimizers, and the exact coding up of models. So I started watching lectures of CS231 to understand all of this, till I understood the concept of convolutional networks, pooling, loss functions, gradients, backpropagation, MLP, and training of Neural Networks. I intend to watch the rest of the lectures of CS229 as I still lack exact mathematical knowledge of Attention and Transformers.

Then, I continued MIT's6.5940 till lecture 4(slides only as lectures were too time-consuming) and also solved their first two labs. The first lab was pretty easy however the second lab used Pytorch extensively and I was overwhelmed because I had no prior experience with pytorch. I found a good tutorial on YouTube for Pytorch and I followed that to learn how to prepare data, create a model, and train and evaluate it. After this, I was ready to do the second lab of MIT 's6.5940. After understanding Pytorch, as it turns out, the lab had a lot of code that I was able to hack together to code up the first assignment.

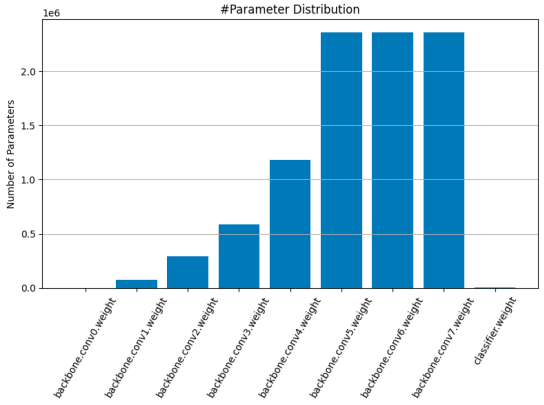
I first coded up a VGG model which was pretty easy and standardly gave around 90.56% accuracy in around 50 epochs with a model size of 35.20 Mb.

For actual experimentation, I loaded a pre-trained model to experiment and revert the model to its original state. This pre-trained model had an accuracy of 92.95% and a size of 35.20Mb.

First, I visualized the Weight Histogram of all layers in the model.



To correctly decide how much to prune it is necessary to see how big(number of parameters) the layers are hence I visualized that too.



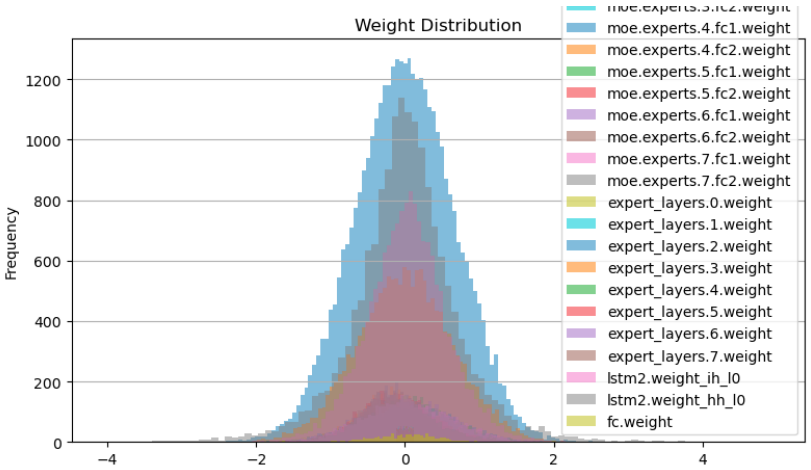
As seen clearly from the graph many weights were distributed around zero, hence it's intuitive that we can make do without them. Initially, I pruned the conv0 layer too with very low sparsity however, that turned out to be brutal as I lost a lot of accuracy in the model. I think that's because conv0 is the first layer and it has the most information about the data. So I then chose not to prune conv0 at all. I went hard on the rest of the layers and pruned them by >= 0.7. I pruned the classifier layer also around that much initially however after trying even more pruning it didn't seem to affect accuracy. So this is the final list of sparsity values I used:

- backbone.conv0.weight: 0
- backbone.conv1.weight: 0.7
- backbone.conv2.weight: 0.8
- backbone.conv3.weight: 0.7
- backbone.conv4.weight: 0.7
- backbone.conv5.weight: 0.8
- backbone.conv6.weight: 0.8
- backbone.conv7.weight: 0.9
- classifier.weight: 0.95

This reduced the model size to 6.86Mb which was 19.50% of dense model size. Sparse model had accuracy=66.54% before fintuning. After fine-tuning for 5 epochs, the accuracy was increased to 92.60%.

I enjoyed doing this assignment a lot. By the end of this assignment, I could say I know somewhat deep learning and while I did not code things starting from scratch for this as I hacked up much code from MIT's course, I had a good foundation built for pytorch.

The concept of pruning and the fact that many values lie around zero intrigued me a lot, so I kept in mind to check it out on the later models I built too. So after completing the NLP task, I visualized the weight distribution of its layers too. It looked like this (without the embedding layer (which too was distributed majorly around 0) as it was too big and dwarfed the other layers)



I suspect it's possible to prune this model too. Its current size is around 19MB. I will try to prune this model later just for fun.

Next, I decided to attempt Graph Neural Networks, as graphs in computing interest me. I had previously worked on a Graph Optimization problem for DisCo Project where I used linear programming to solve it.

- # Graph Neural Networks

Before jumping into the paper, I first learned how convoluted neural networks could be built on top of the graph structure. Then I started watching Antonio Longa's Tutorials. Here, I learned the math behind GCNs and GAT convolutional layers. I implemented my own GATConv layer from scratch, however, I'm not exactly sure how PyG's MessagePassing class works. So, I made some assumptions about how things work by looking at the implementation of the GCN layer in PyG's tutorial. It was enjoyable to implement the GATConv layer as it was my first time implementing a model from scratch. The countless dimensions mismatch while stacking the layers was hard to manage because of the nature of graph structure however it was quite enjoyable.

I found it difficult to import and preprocess data from networkrepository.com so I decided to use PyG's inbuilt datasets. PyG has a dataset called MoleculeNet. From that, I imported a

dataset called ESOL of size 2.5Mb. I wanted to run it on my local machine so chose a low-sized dataset. The ESOL dataset has data about molecules. The goal of the problem with this dataset is to predict the value of solubility of a molecule given its features.

While I was learning about graph models on the internet I came across a model that used 3 GCN layers. To save time, I used that code and my GAT layer to develop a few models.

These models took about 24 hrs to train. So I lost patience often and couldn't document results properly. But still, here are the results of my experiments :

Layers	Results
• My GAT layer • 3 GCNconv layers • fc layer	Epoch 1600 | Train Loss 0.05965699627995491
• GAT layer(PyG implemented) • fc layer	Epoch 1900 | Train Loss 0.7036450505256653
• GAT layer(PyG implemented) • 3 GCNconv layers • fc layer	Epoch 1800 | Train Loss 0.09881985932588577

From the experiments that I was able to document, I noted that when I used my implemented GATConv layer and 3 GCN layers, I was able to reduce the loss a lot and the model was good(singlePJzGAT)

A single GAT layer implemented by PyG wasn't able to perform well on the dataset and even after 1900 epochs it demonstrated high loss(singleGATConv)

A model with a PyG-implemented GAT layer and 3 GCN layers demonstrated a loss of 0.098819859 in 1800 epochs(result1)

I realized that I should have documented the results well as my current results are not very satisfactory.

TODO: evaluate the results of the model, more thoroughly.

- # Natural Language Processing

I watched the first lecture of cs224n to gain some insight into how NLP works, as I had no previous knowledge. I then learned the concept of tokenization and word embeddings. I did not use Bert tokenizer to tokenize my inputs, instead, I followed the traditional approach of

NLP where I had an embedding layer as the first layer in my model. I tokenized the inputs by a simple vocab dictionary where every word was simply assigned one number.

MoE layers build upon the observation that language models can be decomposed into sub-models or 'experts' that focus on distinct aspects of the input data, thereby enabling more efficient computation and resource allocation.

Coding up an MoE layer was pretty straightforward, just stacking expert layers on top of each other where one expert layer has two fully connected layers with ReLU activation) however, I did not understand how to use Pytorch's LSTM layer and code up a NER model. It's easy to understand(relatively) theoretically, however practically I at first did not understand how to implement an NER model.

Thankfully, I found a blog by Andrew NG on Named Entity Recognition, which taught me to build a model for NER. Next, I built the model after much Cursing the Dimensionality(Matching dimensions when stacking layers).

The architecture of My Network consists of:

- Embedding Layer: This layer converts input tokens into dense vector representations using an embedding matrix of size 30292, with each token represented by a vector of dimension 150.
- First LSTM Layer (lstm1): This layer is a Long Short-Term Memory (LSTM) unit with an input size of 150 and a hidden size of 64. It processes the embedded tokens in a batch-first manner.
- Mixture of Experts (MoE): The MoE component employs a set of eight experts, each consisting of two fully connected layers (fc1 and fc2) with input and output sizes of 64. These experts are combined using a sigmoid gating mechanism and dropout with a probability of 0.5.
- Second LSTM Layer (lstm2): Another LSTM layer with an input and hidden size of 64 processes the outputs from the MoE component in a batch-first manner.
- Fully Connected Layer (fc): The final layer is a linear transformation mapping the output of the second LSTM layer to a 10-dimensional vector, suitable for the classification task.

This was the first iteration(first documented iteration🙂) of my model. It has a size of 19 MB and achieved an F1 score of 90.56 in 5 epochs.

Learning from my previous failure in documenting results, This time I duplicated my notebook and saved the models every time I wanted to change something.

In the second iteration, I removed the MoE layer in between and kept just the two LSTM layers (along with embedding and a fully connected layer). Surprisingly it gave me better results. This model has a size of 18.5 MB and achieved a score of 91.08 in 5 epochs.

In the third iteration, I increased the number of expert layers to 25 to see if the model could learn better, but to no avail. This model achieved an F1 score of 89 in 10 epochs.

In the fourth iteration, I removed the first LSTM layer and used just an MoE and then an LSTM layer and achieved very bad results. In hindsight, removing the LSTM was a very bad decision because the first LSTM layer carries information in between and relating words in the input. when I removed the first LSTM layer I lost all information between words resulting in very bad results. The model had an F1 score of 83 after 10 epochs.

As a last experiment, I just kept one LSTM layer as I realized that the MoE layer somehow wasn't giving results. This model gave an F1 score of 90.64 in 10-15 epochs. (I'm not sure because I started training it once, then interrupted it and again trained...) . Also, in this model, I maintained the lowest loss variable and reverted the model to the new lowest losses.

Still, I couldn't make sense of why an MoE layer wasn't improving the model, So, I reduced the number of experts to 4, following a not-very mathematical logic that since the dataset has tags that can be broadly classified into four categories, that is Person, Location, Organisation, and Miscellaneous, there could be four experts who would identify words into these categories and then into their further categories.

Surprisingly this model gave an F1 score of 90.66 in 5 epochs and ran relatively faster than when I had unnecessarily more experts. Encouraged by this I trained the model for 5 more epochs, but that led to overfitting on the training set and my F1 score dropped to 89.61.

I couldn't complete the second part of this project due to lack of time.

Thanks for Reading!