Regularly Oversimplifying Neural Networks

Nick Kross, Mica White

Interpretability Hackathon Report

Apart Research

PIs: Esben Kran, Neel Nanda, Fazl Barez

Date: 13th November, 2022

Abstract

What we want to do is be able to look at two neural networks and find similar features between them. Our approach is to create a data format that makes this easy. We created a regular language (regular as in a regular expression) to represent neural networks, which should make this process easier. This discards much needed information about the neural network, but should serve as a decent pruning technique before trying more expensive algorithms.

So what is lost in this? Our representation contains the weights of each neuron, and the bias for each neuron, but does not specify which connection the weight applies to. This is undoubtedly very important, and no good solution to interpretability can completely remove this information. Our goal here is to simply figure out what patterns might exist in the network, so that less time needs to be spent on expensive algorithms which could prove that there is some pattern in the neural network.

Representing a neural network using a regular language might seem impossible. Keep in mind that Assembly language is actually a regular language, and it describes Turing-complete languages. Although, semantic analysis is required to determine what labels mean. By discarding the connection information from the representation, we can skip this step, saving us valuable computation time.

Regularly Oversimplifying Neural Networks: Extracting the Essential Substructures of Model Weights

One way to approach determining if two neural networks share similar features would be to attempt subgraph isomorphism. This is a fancy way of saying, “find the small neural network inside the big neural network”. Although we have not tested it, this seems terribly slow. There are many possible bijective functions that could translate a large neural network that we don’t understand into a small neural network which we do understand. It’s not immediately clear how this would be optimized. Our approach is to think about what life would be like if the world were simple, and everything was easy.

The easiest would be a simple substring search. We would convert both neural networks into strings, and call the `find` function. This does not work. We would need to create a representation that puts all of the neurons from the small network next to each other somehow. There are lots of small neural networks, and so eventually we would have a contradicting order of neurons.

So here's another idea, which should work better. If we have a neural network that does addition, that network is going to contain multiple layers. Let's say a layer is a sublayer of another layer if all of its neurons are contained in the larger layer.

```
class LayerModel:
    def __init__(self, weights: Tensor, biases: Tensor):
        self.neurons = [NeuronModel(weights[i], biases[i]) for i in range(len(weights))]

    def __len__(self):
        return len(self.neurons)

    def __iter__(self):
        return iter(self.neurons)

    def __getitem__(self, i: int) -> NeuronModel:
        return self.neurons[i]

    def is_superlayer_of(self, other) -> bool:
        for neuron in other:
            if neuron not in self: return False
        return True
```

If a neural network is contained in a larger neural network, then all of its layers should be sublayers of layers in the larger neural network, and those layers should be next to each other in the large neural network, in order.

```
def is_possible_subnetwork(big: NNModel, sub: NNModel) -> bool:
    for i in range(len(big)):
        if big[i].is_superlayer_of(sub[0]):
            is_subnetwork = True
            for j in range(1, len(sub)):
                k = i + j
                if not big[k].is_superlayer_of(sub[j]):
                    is_subnetwork = False
                    break
            if is_subnetwork: return True
    return False
```

This leaves the question of how to determine if two neurons are equivalent. Consider the following table:

Input	Neuron 1 Output	Neuron 2 Output
0, 0	0	0
0, 1	0	0
1, 0	0	0
1, 1	1	1

These two neurons are equivalent, because the same input gives them the same output. It's implemented in Python as shown below. The `excited_states` function gives us the inputs which would cause the two neurons to emit a one. After that, checking for equivalent neurons is as simple as seeing if the two lists are equal to each other.

```
next_neuron_id = 0

class NeuronModel:
    def __init__(self, weights, bias):
        global next_neuron_id
        self.id = next_neuron_id
        self.bias = float(bias)
        self.weights = sorted([float(weight) for weight in weights])
        next_neuron_id += 1

    def __str__(self) -> str:
        string = ""
        string += f"\nNeuron {self.id}:"
        string += "\n\tWeight #: " + str(len(self.weights))
        string += "\n\tWeights : " + str(self.weights[0:2]) + "..."
        string += "\n\tBias    : " + str(self.bias)
        string += "\n\tStates  : " + str(len(self.excited_states()))
        return string

    def __eq__(self, other) -> bool:
        # optimization: don't bother if the neurons have different weight counts
        if len(self.weights) != len(other.weights): return False
        return self.excited_states() == other.excited_states()

    def excited_states(self) -> List[List[int]]:
        states = []

        if sum(self.weights) > 0: states.append(list(range(len(self.weights))))

        for i in range(len(self.weights)):
            for state in itertools.combinations(range(len(self.weights)), i):
                total = 0
                for w in state:
                    total += float(self.weights[w])
                if total + self.bias > 0: states.append(list(state))

        return states
```

This is the slowest part of our approach, and it's discussed more in the **Future Work** section. The reason why this is slow is because it's checking for every possible series of inputs, of which there are 2^n possibilities. For large neural networks, this will need to be improved before the approach becomes useful.

However, we did find that finding a sublayer on a small neural network, with 16 inputs, is reasonably efficient, and this is still a large step, in our opinion. This solution helps narrow down the list of what to look for, and makes a search through a neural network faster, which may help to efficiently determine what a neural network is actually doing in the future. This is a step towards checking that an artificial intelligence is not deceiving its user about what it's actually doing.

As for novelty, we're not aware of much progress being made on regular-language representations of neural networks, due to the large-runtime comparison operations described above. Any progress on creating such representations will be useful for interpretability at large, as it will make it more feasible to search through and compare the substructures of different neural networks. This could e.g. unearth deceptive sub-circuits, or discover obscure or counterintuitive operations within large models.

We don't expect this to work perfectly on every, or really any useful neural networks. What we do expect is that it will give all of the possible patterns from a predefined list. Although we have not rigorously proven this, we don't expect the concept will be disproven. We have set up a Colab document which may be used to play with this algorithm. There's also a GitHub repository.

Future Work

Even on a small neural network, checking for equal neurons is pretty slow. Making this fast would allow this to work on large neural networks. It's not possible with the algorithm we used though. Here's one quick idea for optimizing the `__eq__` method on `NeuronModel`.

```
def __eq__(self, other: Self):
    if len(self.weights) != len(other.weights): return False
    epsilon = 1 / len(self.weights)
    for i in range(len(self.weights)):
        my_weight = float(self.weights[i])
        other_weight = float(other.weights[i])
        difference = abs(my_weight, other_weight)
        if difference > epsilon: return False
    return True
```

The idea is that if the weights and biases are similar, then the output is probably similar. The problem with this is that there are multiple ways to represent the same neuron. As one example, a neuron that represents an AND gate can be represented with any negative number for the bias, as long as the two inputs are small enough.

It also seems that our equivalence function could be highly parallelized. Moving the work over to the GPU may be quite feasible. If that doesn't work, rewriting the project in Rust should be several dozens of times faster.

Another possible improvement is to, instead of having an `excited_states` method, handle the creation of the excited states in the `__eq__` method, and terminate early if something is not found in both. Of course, this only helps if the two neurons are not equal. This is the case for most neurons though, so it would still help.

Putting aside optimization, not all of the neuron's connections are actually useful. Some have weights so small that they could never change the output of the neuron. This does have performance implications. We might not have to check every single combination of inputs. There's also a functionality implication here. If there are 700 weights in a neuron, it should still be theoretically possible for it to be equivalent to an AND gate. Our implementation does not allow for it at this time.

On small subnetworks, there's another optimization we can try. We can actually use the same method that was used for finding equivalent neurons to determine if two subnetworks are equivalent. Just try every possible input and see if the output is the same. This obviously isn't possible for the entire graph with 784 inputs, but it might be faster than comparing the individual neurons in some cases, if we limit the number of input neurons.

As for future tokenizers, hopefully our provided `GeneralTokenizer` class can serve as a springboard for future compression efforts. It currently only uses the information from each neuron-to-neuron connection individually, though it could be modified to, for instance, look at clusters of similar-weight connections within a layer. We've barely scratched the surface of potential attack-angles on the regularly-oversimplifying problem.

To clarify, the core desiderata for an ideal NN tokenizer can be written as follows:

1. It reveals more-fundamental structure in the NeuralModel data
2. That structure is (ideally) invariant across different NNs that have the same concept (e.g. if two store "addition", we want to inch closer to "both tokenizations contain the same token/token sequence, pointing to their addition connections", while balancing desideratum 3...
3. The tokenization makes searching more intuitive / human-readable, without going to gigantic runtimes.

While these problems won't be solved in our short time allotment, the `GeneralTokenizer` framework can aid future work on them.

Finally, this work has been heavily focused on neural networks, but we expect that some version of this can be done for transformers that are used in language models. We focused on neural networks because that seemed to be the easiest way to get started.

Repo link: https://github.com/nicholaskross/RegularlyOversimplifyingNNs