Deep Learning Lab 2

Bayard Walsh

November 2023

1 Configurations

For this experiment, I used the pre-trained hugging face model `distilgpt2` [San+19] and the `wikitext` and `wikitext-2-raw-v1` data sets [Mer+16] for fine-tuning, trained on an A100 GPU accessed through Google Collab. To create sparsity in a given network, I recreated the Iterative Magnitude Pruning algorithm, applying global pruning across the 6 layers of weights [Che+20], with a reduction of 25 percent of the current model size with each iteration. I only pruned the weights at each layer before entering a transformer (instead of pruning the entire model) because the weights were far smaller in certain parts of the model so global parsing would disproportionately impact certain parts and I wanted to determine the distribution of high magnitude weights across layers. I followed the sparsity reduction heuristic of removing the nonzero weights with the smallest magnitude [Mor+19]. In terms of assessing initial sparsity, the pre-trained model I worked with had all nonzero weights. For recording model performance, for each sparsity benchmark, I recorded the distribution of sparsity across the 6 layers, training and validation loss, run time, and perplexity, as measured by the hugging face `evaluate` package.

2 Results

Surprisingly, layer 1 had the highest percent of low magnitude weights in the pre-trained model, and quickly became completely sparse for all experiments with global sparsity above 10 percent. In contrast, the deeper layers had a higher magnitude, conflicting with the notion that deeper layers are less important [Mor+19]. However, the researchers mention that this phenomenon could be a result of the deeper layers being generally larger, causing global pruning to disproportionately reduce their size, and because `distilgpt2` has layers with a constant size and I only considered sparsity for layers right before transformers, this could explain why this pattern didn't occur. However, by 99 percent global sparsity the sparsity is evenly distributed across layers, with slightly less weights in the first and more in the final layer. This shows that the highest magnitude features can occur throughout the model. Another key result is the drastic increase in run time as the model generates more and more sparsity. This is caused by the iterative implementation of the IMP algorithm, as removing 25 percent of current non-zero weights on each training round becomes more and more inefficient as the model becomes sparser. Therefore, using IMP to create very high sparsity faces the same challenge as training to complete optimization [SGM22], where the potential gains become so minimal that the trade-off in compute time becomes wasteful.

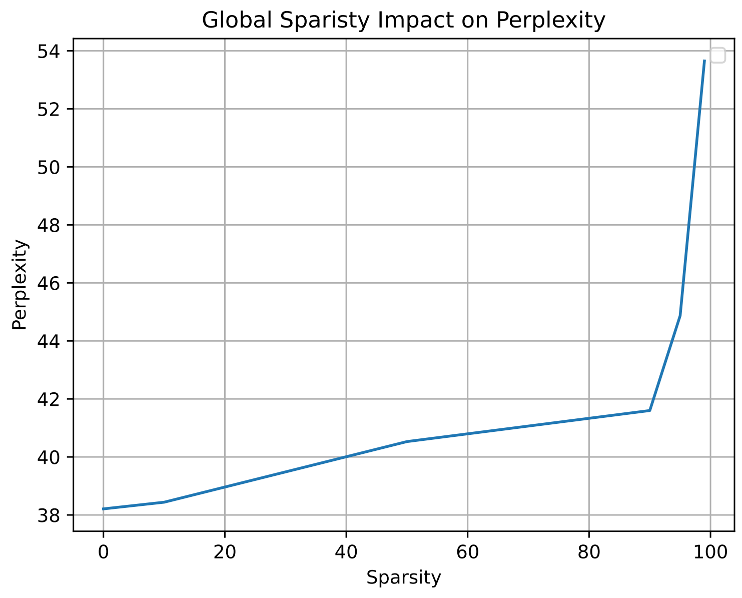
3 Challenges

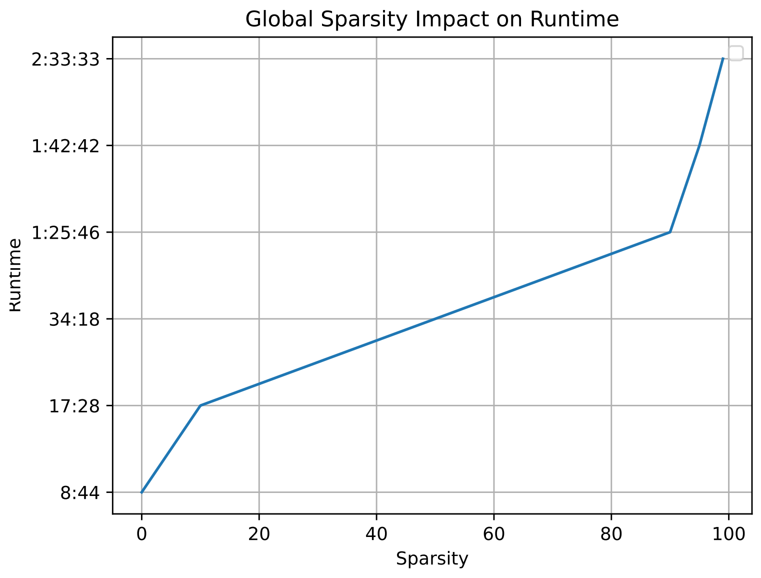
One of the key challenges in the experiment was over-fitting a specific data set. This is because the model was trained on the same data at every iteration. Because of the small size of the model and fear of over-training, I decided to *reset* (revert nonzero weights to pre-trained state each iteration) instead of *rewind* (revert nonzero weights to a previous (partially trained) state each iteration) the model [Che+20]. However, even though the weights of the model were reset to the initial pre-trained state, the weights that best fit the specific data set had the highest magnitude after a training iteration and were therefore not pruned, and as the results of the paper I found the small training set I used resulted in over-fitting [Che+20]. Over-fitting is reflected in the consistent decrease in training loss with sparser models, with a rise in perplexity that indicates a loss of generality, especially the version with 99 percent sparsity. Another key issue was the size of the model (the heavy computation required by the IMP algorithm required a smaller model), which had only 6 layers and 768 dimensions, so the distribution of sparsity across the layers in the network was tougher to determine with a smaller set of weights.

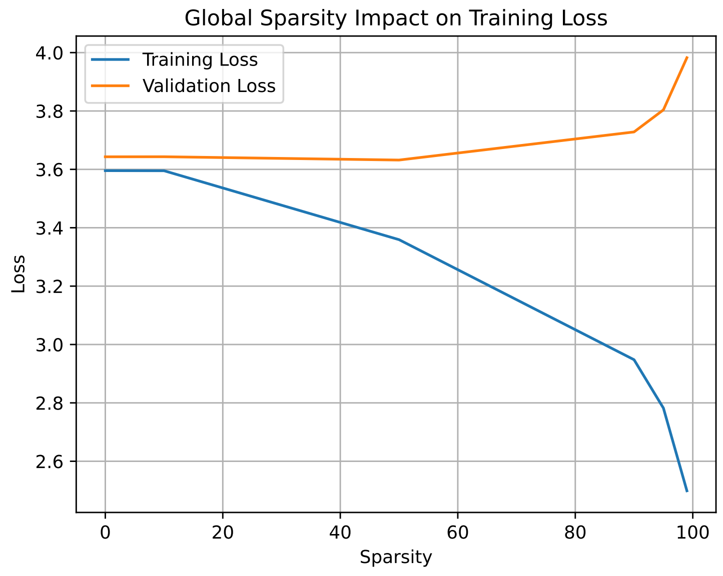
4 Improvements

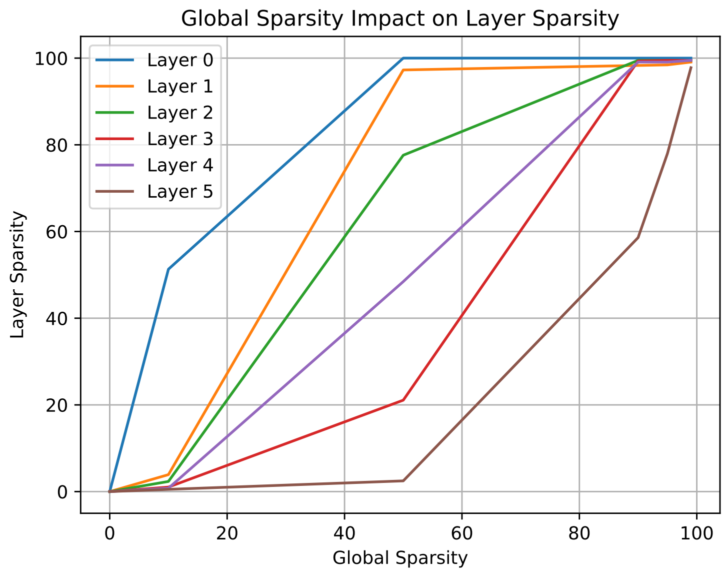
A potential area to improve the experiment would be to measure the connectivity across layers between nonzero weights after inducing sparseness, and measure which features are completely unused. With this metric, one could redesign the model shape to remove the nodes that don't have any connections across the entire model, which would reduce the overall storage needed for the model. Along with removing unused nodes, another improvement would be to add dynamic layer pruning across iterations. Because the IMP algorithm already resets the model on each pruning iteration when it reverts to the pre-trained model with increased sparsity, one could adjust the architecture to remove a layer with all zero weights between iterations to speed up training time, for example in this experiment, layer 1 could have been removed for all training runs after 50 percent sparsity. Therefore, reductions could be made to reduce the overall size of the model horizontally (considering connections between specific nodes) and vertically (considering entire layers). This would save computation and memory without majors changes to the model architecture.

5 Performance






Global Sparsity Impact on Training Loss
Training Loss
Validation Loss
Loss
Sparsity
4.0
3.8
3.6
3.4
3.2
3.0
2.8
2.6
0
20
40
60
80
100


Global Sparsity Impact on Layer Sparsity
Layer 0
Layer 1
Layer 2
Layer 3
Layer 4
Layer 5
Layer Sparsity
Global Sparsity
100
80
60
40
20
0
0
20
40
60
80
100

In [ ]:
```
!pip install huggingface_hub
!pip install -U accelerate
!pip install -U transformers
!pip install datasets

from huggingface_hub import notebook_login
import transformers
from transformers.utils import send_example_telemetry
from datasets import load_dataset
from transformers import AutoTokenizer
from transformers import Trainer, TrainingArguments
import torch
import re
import numpy as np
from transformers import AutoModelForCausalLM

notebook_login()
```

In [ ]:
```
send_example_telemetry("language_modeling_notebook", framework="pytorch")
datasets = load_dataset('wikitext', 'wikitext-2-raw-v1')
```

In [ ]:
```
model_checkpoint = "distilgpt2"
```

In [ ]:
```
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint, use_fast=True)
```

In [ ]:
```
def tokenize_function(examples):
    return tokenizer(examples["text"])
```

In [ ]:
```
tokenized_datasets = datasets.map(tokenize_function, batched=True, num_proc=4, remove_columns=["text"])
```

In [ ]:
```
block_size = 128
```

In [ ]:
```
def group_texts(examples):
    concatenated_examples = {k: sum(examples[k], []) for k in examples.keys()}
    total_length = len(concatenated_examples[list(examples.keys())[0]])
    total_length = (total_length // block_size) * block_size
    result = {
        k: [t[i : i + block_size] for i in range(0, total_length, block_size)]
        for k, t in concatenated_examples.items()
    }
    result["labels"] = result["input_ids"].copy()
    return result
```

In [ ]:
```
lm_datasets = tokenized_datasets.map(
    group_texts,
    batched=True,
    batch_size=1000,
    num_proc=4,
)
```

In [ ]:
```
model = AutoModelForCausalLM.from_pretrained(model_checkpoint)
```

In [ ]:
```
model_name = model_checkpoint.split("/")[-1]
training_args = TrainingArguments(
    f"{model_name}-finetuned-wikitext2",
    evaluation_strategy = "epoch",
    learning_rate=2e-5,
    weight_decay=0.01,
    push_to_hub=True,
)
```

In [ ]:
```
# for a given threshhold (using 0 as threshhold), print percentage of weights in each layer <= threshhold
def getSparsityLayer(model,thresh):
  for name, param in model.named_parameters():
      if re.match((r"transformer\.h\.[A-Za-z0-9_]+\.ln_1\.weight"), name):
        target_layer_name = name
        target_layer_params = dict(model.named_parameters())[target_layer_name]
        total_elements = target_layer_params.numel()
        num_zeros = (target_layer_params <= thresh).sum().item()
        percentage_zeros = (num_zeros / total_elements) * 100
        print(f"Percentage of zeros in layer '{target_layer_name}': {percentage_zeros:.2f}%")
```

In [ ]:
```
# get the overall sparsity accross 6 layers of model
def getSparsitySum(model,thresh):
  total=0
  for name, param in model.named_parameters():
    if re.match((r"transformer\.h\.[A-Za-z0-9_]+\.ln_1\.weight"), name):
      target_layer_name = name
      target_layer_params = dict(model.named_parameters())[target_layer_name]
      total_elements = target_layer_params.numel()
      num_zeros = (target_layer_params <= thresh).sum().item()
      percentage_zeros = (num_zeros / total_elements) * 100
      total+=percentage_zeros

  return total /6
```

In [ ]:
```
# given a percentage x and a model, return the minimum value where x% of values in a model are <= that threshold
def getThreshhold(model,percent):
  pattern = r"transformer\.h\.[A-Za-z0-9_]+\.ln_1\.weight"
  matching_values = []
  for name, param in model.named_parameters():
      if re.match(pattern, name):
          matching_values.extend(param.view(-1).cpu().detach().numpy())

  matching_values.sort()
  matching_values = [value for value in matching_values if value != 0]
  percentile_index = int(percent * len(matching_values))
  return matching_values[percentile_index]
```

In [ ]:
```
# given a set threshhold set all values in the model layers below that value to 0
def createSparsity(threshhold, model):
  pattern = r"transformer\.h\.[A-Za-z0-9_]+\.ln_1\.weight"
  for name, param in model.named_parameters():
      if re.match(pattern, name):
          param.data[param.data < threshhold] = 0.0
  return model
```

In [ ]:
```
# merge all nonzero values from modelA to modelB, used for resetting
def resetting(modelA,modelB):
  state_dict_with_zeros = modelB.state_dict()
  state_dict_to_update = modelA.state_dict()

  for key, value in state_dict_with_zeros.items():
      if key in state_dict_to_update:
          state_dict_to_update[key] = value

  modelA.load_state_dict(state_dict_to_update)
  return modelA
```

In [ ]:
```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=lm_datasets["train"],
    eval_dataset=lm_datasets["validation"],
)
```

In [ ]:
```
trainer.train()
```

In [ ]:
```
model.save_pretrained('inital training')
```

In [ ]:
```
model0I = AutoModelForCausalLM.from_pretrained('inital training')
```

In [ ]:
```
while getSparsitySum(model,0) < 95:
  createSparsity(getThreshhold(model,.25),model)

  print(getSparsitySum(model,0))

  trainer.train()
  model = resetting(model0I,model)
```

In [ ]:
```
getSparsityLayer(model,0)
```

In [ ]:
```
import math
eval_results = trainer.evaluate()
print(f"Perplexity: {math.exp(eval_results['eval_loss']):.2f}")
```

References

[Mer+16] Stephen Merity et al. *Pointer Sentinel Mixture Models*. 2016. arXiv: `1609.07843 [cs.CL]`.

[Mor+19] Ari S. Morcos et al. *One ticket to win them all: generalizing lottery ticket initializations across datasets and optimizers*. 2019. arXiv: `1906.02773 [stat.ML]`.

[San+19] Victor Sanh et al. *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. 2019.

[Che+20] Tianlong Chen et al. *The Lottery Ticket Hypothesis for Pre-trained BERT Networks*. 2020. arXiv: `2007.12223 [cs.LG]`.

[SGM22] Ruoqi Shen, Liyao Gao, and Yi-An Ma. *On Optimal Early Stopping: Over-informative versus Under-informative Parametrization*. 2022. arXiv: `2202.09885 [cs.LG]`.