*NanoGPT Experiments | Builders Blog | Feb 2025*

---

For this task I decided to test various modifications to the `nanoGPT` repo related to training, model architecture, and dataset preparation. For the majority of experimental runs I trained the model with an **A100 (40 GB SXM4) on Lambda Labs.** For one subset of my conditions, which I will indicate further in this section, it was necessary to use an **H100 (80 GB PCIe)** instead.

Below, I am also attaching my annotated copy of the recommended reading for this section:

character_level_language_modeling_with_deeper_self_attention.pdf

Regarding the note on reporting loss in terms of bpc instead of nats, I addressed this by dividing losses by a scale factor of `ln(2)` in the `train.py` file of `nanoGPT`.

```
losses = estimate_loss()

# Convert loss from nats to bits
train_bpc = losses["train"] / math.log(2)
val_bpc = losses["val"] / math.log(2)
```

---

***IMPLEMENTATION***

The training runs I have conducted are a combination of the 5 experimental conditions below.

```
BASE                                nanoGPT exactly as is with the
                                    exception of reporting loss in bpc

ROPE                                Replace position embeddings with
                                    rotary position embeddings

TIPA                                Save reverse character mappings in
                                    dataset curation, utilize during
                                    training and then forward pass

DEEP                                Increase layers: 16 -> 32,
                                    increase context length: 256 -> 512

SubData Pre/Post Training Scheme       1. Pre-train the base model for
```

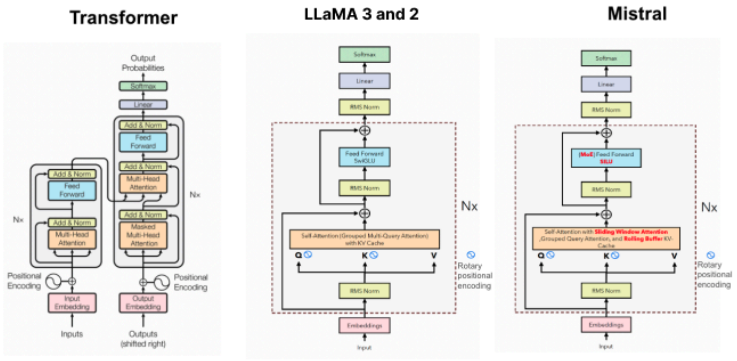
---

```
                                        some number of steps on
                                        original enwik8 dataset
                                    2. Sequentially fine-tune on 16
                                        “expert” subsets of enwik8
                                    3. Post-train for a small number
                                        of steps on original enwik8
```

My motivation for each of these conditions was as follows.

## ROPE

After its introduction, ROPE has been a commonplace implementation in place of position embeddings in transformers. It is currently a part of the transformer architecture for LLaMA models, as seen in the attached diagram.




| Feature | Plain Transformers | LLaMA 3 and 2 | Mistral |
|---|---|---|---|
| Architecture | Encoder-Decoder | Single Layer | Single Layer + Enhancements |
| Positional Encoding | Standard | Rotary | Rotary + Additional Innovations |
| Normalization | LayerNorm | RMS Norm | RMS Norm + Enhanced Mechanisms |
| Attention Mechanism | Self-Attention | Grouped Multi-Query Attention | Sliding Window Attention |
| Memory Efficiency | N/A | KV Cache | Rolling Buffer KV Cache |
| Activation in Feedforward | RELU | SwiGLU | SwiGLU + Mixture of Experts |

*[Diagram originally from [lightning.ai/fareedhassankhan12](lightning.ai/fareedhassankhan12)]*

## TIPA

Token Internal Position Awareness (TIPA) is introduced in the paper “Enhancing Character-Level Understanding in LLMs through Token Internal Structure Learning” by `(Xu et al., 2024)`. It hopes to address the limited understanding of character positions within tokens, especially in tasks like Chinese Spelling Correction (CSC) where character-position within tokens is more critical than in English language tasks. The paper implements an algorithm that

“capture character positions within tokens by training them on reverse character prediction tasks using the tokenizer’s vocabulary” `(Xu et al., 2024)`.

While I will not be using a tokenizer since this task’s focus is character-level modelling, my TIPA implementation still draws inspiration from the original paper. Below, I’ve included a pseudocode diagram for the modified data curation. Note, large parts of this remain very similar to Karpathy’s initial `prepare.py` file. **From here, these reverse character mappings are used to define additional linear layers in the model architecture.**

**Algorithm 1** TIPA Character-Level Language Model Data Processing

**Ensure:** processed training data, val data, test data with TIPA mappings

```
# Directly from Karpathy's original nanoGPT
Function CreateCharacterMappings(data):
    chars ← sort(unique(data))
    vocab_size ← length(chars)
    stoi ← {char: index for all chars}
    itos ← {index: char for all indices}
return  vocab_size, stoi, itos

Function CreateReverseMapping(sequence):
    n ← length(sequence)
    mapping ← n - i: sequence[i] for i in [0,...,n-1]
return  mapping

# Directly from Karpathy's original nanoGPT
# except for appending the reverse mappings to train_mappings
Function ProcessData(file_path, context_length):
    data ← ReadFile(file_path)
    vocab_size, stoi, itos ← CreateCharacterMappings(data)
    n ← length(data)
    // Split data
    train_data ← data[0:0.9n]
    val_data ← data[0.9n:0.95n]
    test_data ← data[0.95n:n]
    // Create TIPA mappings
    train_mappings ← [ ]
for i = 0 to length(train_data) - context_length do
        window ← train_data[i:i+context_length]
        reverse_map ← CreateReverseMapping(window)
        Append reverse_map to train_mappings
end for
    // Save metadata
    meta ← {vocab_size, itos, stoi, train_mappings}
    SaveMetadata(meta)
    // Encode data
    train_ids ← Encode(train_data)
    val_ids ← Encode(val_data)
    test_ids ← Encode(test_data)
    SaveEncodedData(train_ids, val_ids, test_ids)
return  train_ids, val_ids, test_ids
```

## DEEP

My implementation of deeper networks was straightforward: I increased layer depth from 16 to 32 and context length from 256 to 512. Note, increasing layer depth exceeded my VRAM capacity on the A100, so I ran my experiments related to deeper networks on one **H100 (80 GB PCIe)** via Lambda Labs.

```
SubData Pre/Post Training Scheme
```

This strategy is essentially a “macro-level” Mixture of Experts fine-tuning run on top of a pre-training of the base nanoGPT. Specifically, I intend to categorize the enwik8 dataset into 16 learned subsets by subject matter. The subset categories are first learned through an iterative sampling process from the original dataset. 50 random samples from enwik8 are passed to an LLM, and the LLM assigns a 16-subject categorization to that sample content. This is repeated for 20 rounds, for a total of 320 (theoretically overlapping) subject categories. A final LLM call is used to extract the 16 most prominent subjects from that list of 320 overlapping subjects. Once the final categories are determined, the entire enwik8 dataset is split into reasonably-sized chunks and sorted into the 16 categories.

**As a sanity check, both the original enwik8 dataset and a concatenation of the 16 subclasses were found to be of length 99,621,832 characters. This ensures all data from the original enwik8 was sorted into a category.**

I have published my enwik8-categories16 dataset on Hugging Face here:

*https://huggingface.co/datasets/Shivamkak/enwik8-categories16*

**The 16 learned categories:**

```
History                          "Events, figures, and periods
                                 significant in human history."

Science                          "Scientific disciplines,
                                 discoveries, and research."

Technology                       "Technological advancements,
                                 devices, and computing."

Arts and Literature              "Artistic movements, visual arts,
                                 literary works, and authors."

Geography                        "Geographical locations, features,
                                 and regional studies."

Politics and Government          "Political systems, events, leaders,
                                 and governmental structures."

Economics and Business           "Economic theories, business
                                 practices, and financial systems."

Religion and Philosophy          "Religious beliefs, practices,
                                 philosophical ideas, and thinkers."
```

---

| | |
|---|---|
| Health and Medicine | "Medical practices, health conditions, and healthcare systems." |
| Culture and Society | "Cultural practices, societal norms, and social dynamics." |
| Sports and Recreation | "Athletic activities, events, and recreational pursuits." |
| Education and Academia | "Educational systems, institutions, and academic research." |
| Language and Linguistics | "Languages, dialects, and studies in linguistics." |
| Military and Warfare | "Military history, strategies, and defense systems." |
| Entertainment and Media | "Film, television, music, and other forms of entertainment." |
| Biographies | "Life stories and achievements of notable individuals." |

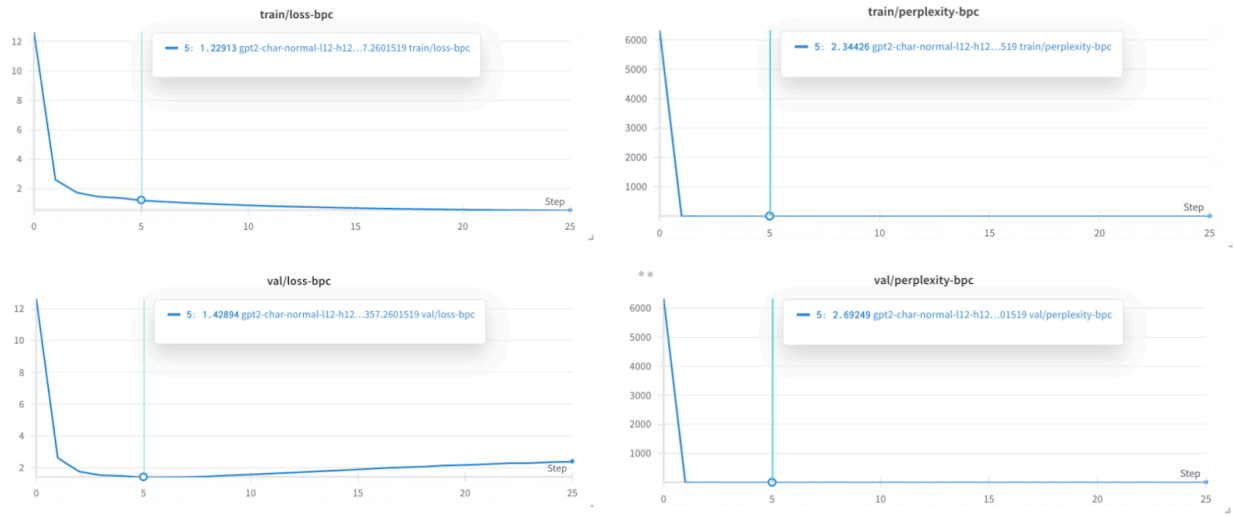
The training structure then follows as such:

```
1.  Pre-train a model on the original enwik8 dataset for X steps
2.  Sequentially fine-tune for Y steps on each of enwik8 sub-datasets
3.  Determine which of the subject fine-tuning runs yields the most
    performant checkpoints
4.  Post-train on best-in-class subject fine-tunings with original enwik8
```

Heuristically, this approach hopes to learn “expert” subnetworks from each sequential fine-tuning run on the sub-datasets, and then relearn the dependencies between these expert subnetworks through the final post-training run on the original dataset.

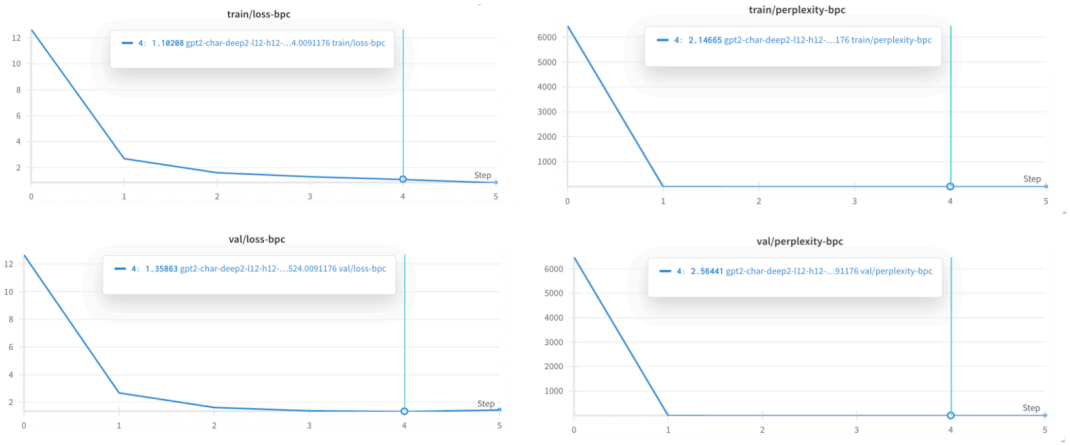
---

***RESULTS***



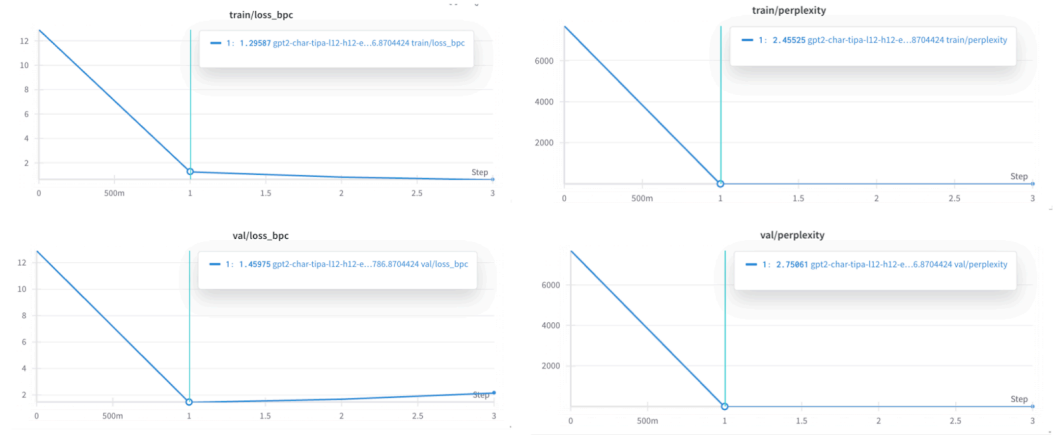

**[Base NanoGPT Train/Val Metrics | Best: 2500 Iterations ]**

---

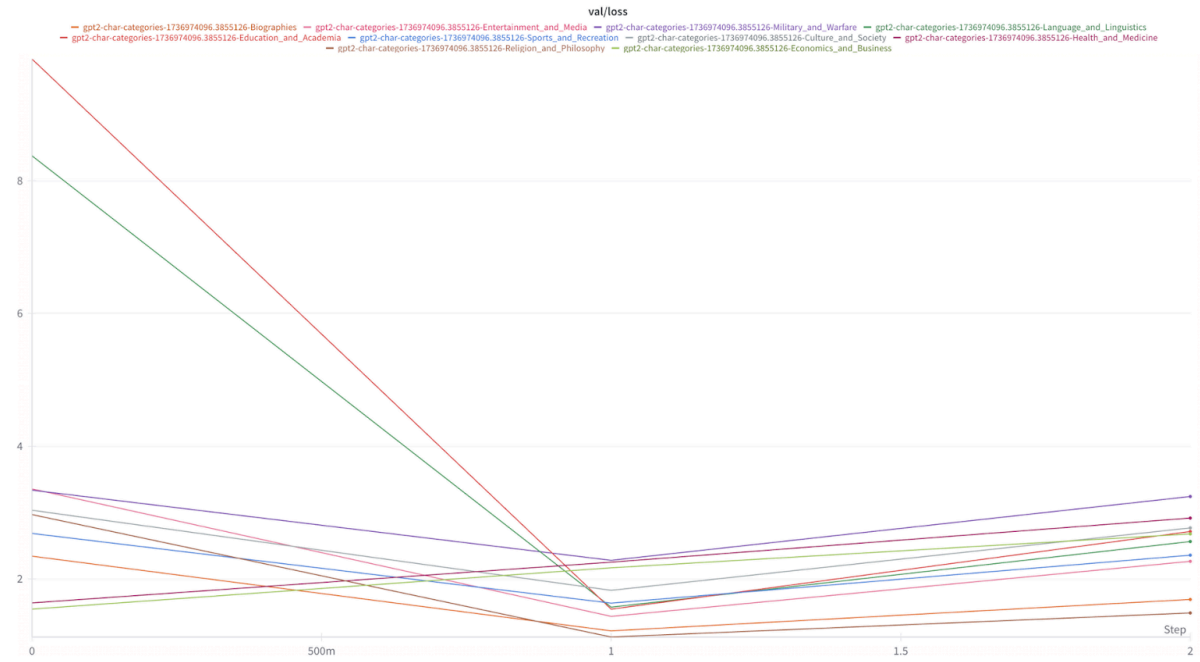



**[Deep Network Train/Val Metrics | Best: 2000 Iterations ]**

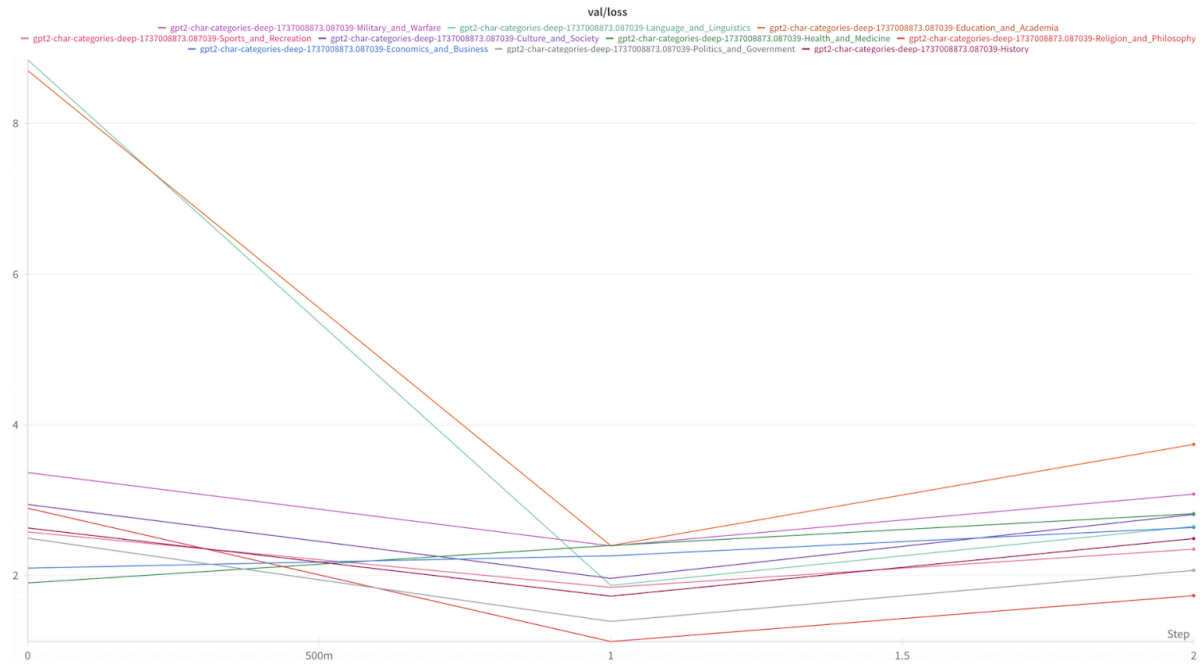

**[ROPE Network Train/Val Metrics | Best: 3000 Iterations ]**



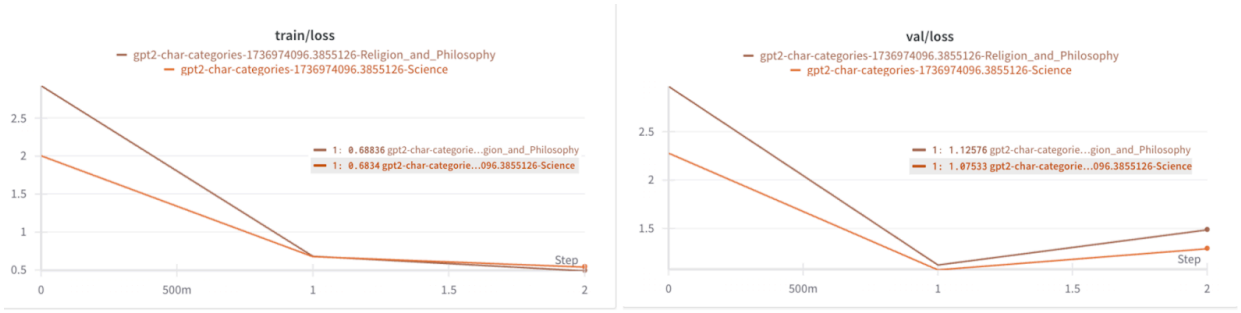

**[TIPA Network Train/Val Metrics | Best: 2000 Iterations ]**



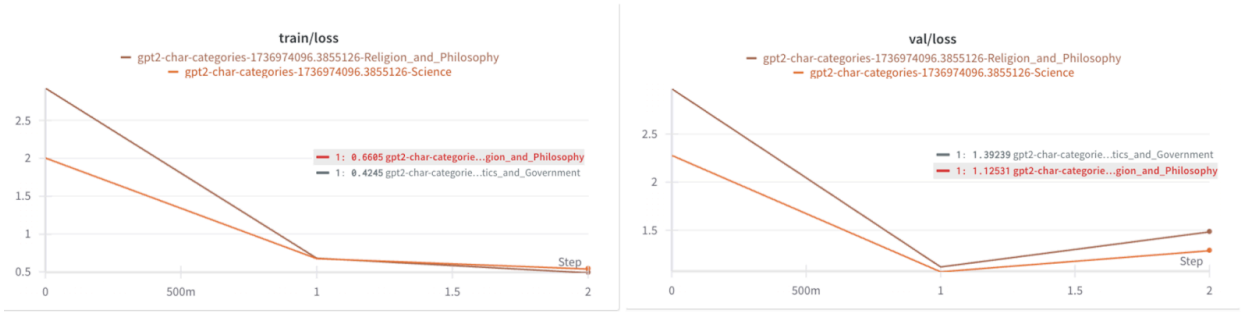

**[Expert Fine-Tuning Runs on BASE Model Val Loss]**



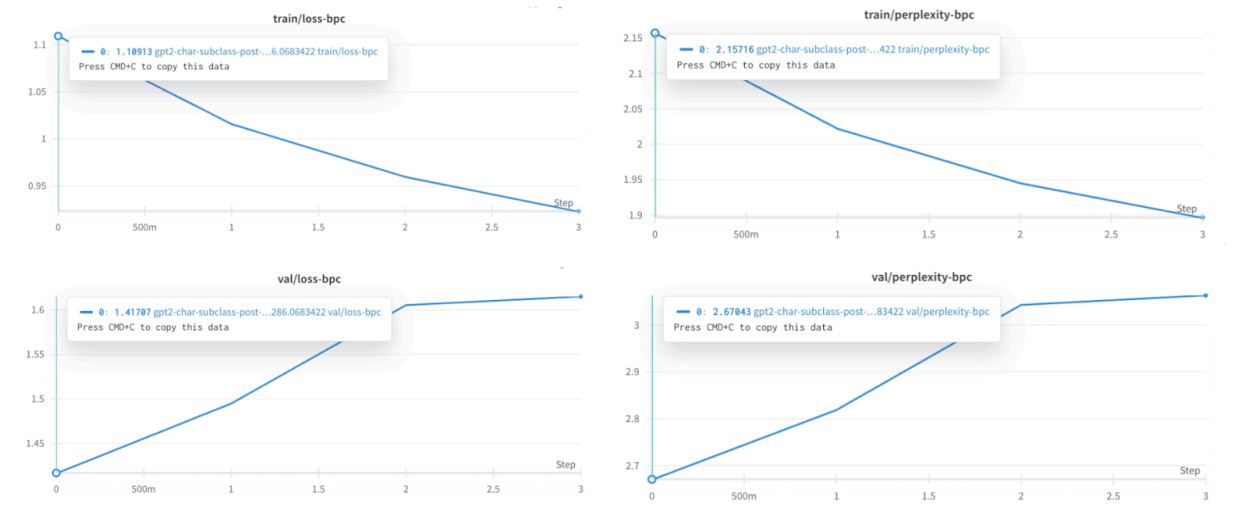

**[Expert Fine-Tuning Runs on DEEP Model Val Loss]**




**[Expert Fine-Tuning on BASE Top Performers | Val/Train Loss ]**




**[Expert Fine-Tuning on DEEP Top Performers | Val/Train Loss ]**




**[BASE + Science Post-Train Network Train/Val Metrics | Best: 500 Iterations ]**


**[BASE + Philosophy Post-Train Network Train/Val Metrics | Best: 500 Iterations ]**

Below, I've curated the validation loss for the best-performance checkpoint for each experiment:

| Experimental Condition | Best Validation Loss in BPC |
|---|---|
| BASE + Expert Fine-Tune : Science | 1.075 |
| DEEP + Expert Fine-Tune : Philosophy | 1.125 |
| BASE + Expert Fine-Tune : Philosophy | 1.126 |
| DEEP | 1.359 |
| BASE + Philosophy Post-Train | 1.390 |
| DEEP + Expert Fine-Tune : Government | 1.392 |
| BASE + Science Post-Train | 1.417 |
| BASE | 1.429 |
| ROPE | 1.429 |
| TIPA | 1.460 |

***ANALYSIS & NOTES***

It was very exciting to see my expert fine-tuning scheme perform to the extent that it did with the science category fine-tuning validation loss of 1.075 far exceeding that of the base implementation at 1.429. However, this scheme certainly did not perform as I initially hypothesized. The sequential nature of the fine-tuning runs was expected to aggregate information in newly learned subnetworks; this did not seem to be the case, as the best-performing fine-tuning runs were very close to the start of the process. In fact, the best-in-class science category was second in the list of sequential runs, and all subsequent learnings from other category fine-tunings only seem to have reduced performance.

Further, it was interesting to see that while the deep network outperformed the base network, none of the deep expert fine-tuning runs were able to outperform the best-in-class science fine-tuning run on the base model. For this reason along with the fact that the post-training process failed to improve validation loss for any of the base model fine-tuning runs, I decided not to conduct any post-training experiments for fine-tuning runs off of the deep model.

I was very optimistic about performance for TIPA, as it seems to be a very intuitive idea to train on both forward and backward positional relationships. I still fear that my implementation of the TIPA Module fails to utilize the reverse character mappings in a way that is actually conducive to learning better network representations. I say this because these reverse character mappings were simply used in a linear layer. Future work on this task would be very well spent improving the use of the reverse character mappings in model architecture and loss estimation. Heuristically, however, it does make sense that reverse character mappings may not be as useful in a character-level language model: if the task at hand is next-character prediction, learning the dynamics of previous-character prediction may not be the most optimal use of the language model.

Overall, I really loved this task. Once again, I would really like to thank Andrej Karpathy. Truly the G.O.A.T. His educational resources made this task very straightforward for someone with very little prior experience in training language models.