ChatGPT

ICA-Based Functional Network Masking for LLM Fine-Tuning

Large Language Models have been shown to contain *functional networks* of neurons that frequently co-activate and are crucial to model performance [1]. In this approach, we fine-tune a Hugging Face transformer model while **masking out selected neurons** based on an Independent Component Analysis (ICA) of neuron activations [2]. By toggling which functional networks (groups of neurons) are active during training, we can explore the model's reliance on those networks. Below, we detail the implementation steps, CLI integration, use of precomputed ICA masks or on-the-fly computation, and how to integrate this extension with Hugging Face's SFTTrainer (supporting LoRA and quantization). We also provide an example fine-tuning on a logic/code task to demonstrate usage.

Functional Network Masking in LLMs

Researchers have found that neurons in LLMs form *functional networks* analogous to functional brain networks [3] [4]. These are sets of neurons that consistently co-activate under certain conditions [5]. Crucially, only a small fraction of neurons may constitute key networks essential for performance: *masking* these key networks (setting their outputs to zero) significantly degrades model performance, whereas **retaining only** these networks (masking all others) can still maintain much of the model's functionality [1] [6]. Prior work even showed that manipulating important neurons' outputs via amplification or masking can steer model behavior [7].

Our goal is to leverage these insights by introducing binary neuron masks during fine-tuning. This mask will zero-out either a chosen functional network (to ablate it) or all but that network (to isolate it). The masking is applied in the forward pass to the outputs of specific neurons, thereby affecting which neurons contribute to model computations and which gradients are updated. This allows us to fine-tune the model *with or without* certain functional subnetworks, potentially leading to insights on their role or even a lighter model focusing on key neurons [8].

Implementing Binary Neuron Masks in the Forward Pass

To apply binary masks during the forward pass, we inject PyTorch **forward hooks** at the appropriate locations in the model's architecture. In transformer decoders, each block contains an MLP (feed-forward network) that expands to a higher-dimensional "intermediate" layer (often 4× the hidden size) and then projects back to the hidden size [9]. We treat each dimension in this intermediate layer as a "neuron" and target them for masking (since these are the neurons identified by ICA analysis [10]).

Where to mask: We apply the mask right **after the non-linear activation** in the MLP, just before the second linear projection back to the hidden size. By zeroing out selected intermediate features at this point, we effectively remove those neurons' contribution to the model's output. This is equivalent to "masking

their outputs," as described in prior research [7]. It ensures masked neurons produce no activation and receive no gradient, while unmasked neurons function normally.

How to mask via hooks: We register a forward pre-hook on each MLP's second linear layer (often named `proj` or `down_proj`), so that we can intercept its input. The hook multiplies the input tensor elementwise by a binary mask (1s for active neurons, 0s for masked neurons). Using a forward pre-hook means the mask is applied **before** the linear layer computes its output. This approach cleanly separates masking logic from the model code (no need to manually edit model source) and works with LoRA or quantized layers as long as we target the correct module.

Below is a code snippet that prepares and applies masks to a given Hugging Face model:

```
import torch
from transformers import AutoModelForCausalLM

def apply_ica_masks(model, mask_dict, mask_mode="key"):
    """
    Apply functional network masks to the model's MLP layers.
    mask_dict: dict mapping layer index -> list of neuron indices (the 'key'
neurons).
    mask_mode: "key" to mask out the listed key neurons, or "complement" to mask
out all **except** the listed neurons.
    Returns a list of hook handles for later removal if needed.
    """
    handles = []
    # Determine the model's hidden size to identify MLP output layers
    hidden_size = getattr(model.config, "hidden_size", None) \
                  or getattr(model.config, "n_embd", None) \
                  or getattr(model.config, "d_model", None)
    if hidden_size is None:
        hidden_size = model.get_input_embeddings().embedding_dim

    # Iterate over transformer blocks to find MLP layers
    if hasattr(model, "transformer"):  # e.g., GPT-2, GPT-J, etc.
        blocks = getattr(model.transformer, "h", None) or
getattr(model.transformer, "blocks", None)
    elif hasattr(model, "model"):
# e.g., LLaMA, OPT, etc., where base model is model.model
        blocks = getattr(model.model, "layers", None) or getattr(model.model,
"decoder", None)
    else:
        blocks = None
    if blocks is None:
        print("Could not find transformer blocks in model to apply masks.")
        return handles
```

```
    for layer_idx, block in enumerate(blocks):

# Find the second Linear in the MLP (where in_features > out_features ==
hidden_size)
        for name, module in block.named_modules():
            # Identify linear or equivalent modules by type and shape
            if isinstance(module, torch.nn.Linear):
                in_features, out_features = module.in_features,
module.out_features
            elif module.__class__.__name__ == "Linear8bitLt":  # bitsandbytes
quantized linear
                in_features, out_features = module.in_features,
module.out_features
            else:
                continue
            if out_features == hidden_size and in_features > out_features:
                # This is the MLP down-project layer.
                # Construct a binary mask tensor for this layer
                neuron_indices = mask_dict.get(str(layer_idx), [])  # layer
index as string key
                if mask_mode == "key":
                    # Mask out the key neurons: 0 for listed neurons, 1 for
others
                    mask = torch.ones(in_features, dtype=torch.float32)
                    if neuron_indices:
                        mask[neuron_indices] = 0.0
                else:  # "complement"
                    # Mask out everything except the key neurons: 1 for listed,
0 for others
                    mask = torch.zeros(in_features, dtype=torch.float32)
                    if neuron_indices:
                        mask[neuron_indices] = 1.0
                # Register forward pre-hook to apply the mask
                def mask_pre_hook(module, inputs, mask_tensor=mask):
                    # inputs: tuple of (hidden_states, *args)
                    x = inputs[0]
                    # Ensure mask is on the same device/dtype as input
                    mask_dev = mask_tensor.to(x.device, dtype=x.dtype)
                    return (x * mask_dev, ) + inputs[1:]
                handle = module.register_forward_pre_hook(mask_pre_hook)
                handles.append(handle)
    return handles

# Example: loading a model and applying masks (mask_dict to be defined as per
ICA results)
model = AutoModelForCausalLM.from_pretrained("gpt2")  # using GPT-2 small for
example
mask_dict = {"0": [10, 20, 30], "1": [5, 15]}  # toy example: mask some neurons
```

```
in layer 0 and 1
handles = apply_ica_masks(model, mask_dict, mask_mode="key")
```

In this code, `mask_dict` contains the neuron indices of the **key functional network** per layer (as identified by ICA). By default (`mask_mode="key"`), those indices are set to 0 in the mask (excluded), meaning we **mask out the key network neurons** and keep all others active. If `mask_mode="complement"`, the mask is inverted to **keep only** the key neurons and zero-out all other neurons. We convert the mask to the appropriate device/dtype inside the hook to ensure compatibility (especially important if the model is on GPU or in half-precision).

Verifying hook placement: We identify the correct linear layer by checking for a linear whose `out_features == hidden_size` and `in_features > hidden_size` (since the intermediate layer is larger). This reliably catches the second projection of the MLP in architectures like GPT, LLaMA, OPT, BERT, etc., but skips attention output layers (which have in_features == out_features == hidden_size). For example, in GPT-2 each block's MLP has `c_fc` (1024→4096) and `c_proj` (4096→1024); our code finds `c_proj` and masks its input of size 4096. In LLaMA-7B, each layer's MLP uses two projections (`gate_proj` and `up_proj` both 4096→11008) and a `down_proj` (11008→4096); the hook targets `down_proj`'s input of size 11008. By masking that input, we zero out selected intermediate neurons after the SiLU activation and gating, which is effectively removing those functional units from the network's computations.

Trainable parameters: With this forward masking in place, any neuron set to 0 will also receive zero gradient (since its output does not affect the loss). Thus, during fine-tuning those masked neurons' weights won't update (their gradient is essentially suppressed). The rest of the model trains normally. This fulfills the requirement that the mask affects which neurons are **active and trainable**: masked ones are inactive (and effectively not trained), while unmasked ones continue to learn.

CLI Argument for Masking Mode

We add a command-line option to easily toggle the masking behavior between masking out the key networks versus masking out all other neurons (i.e. isolating the key network). For instance, one could use a flag like `--mask_mode key` vs `--mask_mode complement`, or a boolean `--keep_only_key`. In our implementation, we choose a string `--mask_mode`:

- `--mask_mode key`: **Exclude** the identified key functional network(s) (i.e. set those neurons to zero during forward pass).
- `--mask_mode complement`: **Include only** the key network, masking *all other* neurons.

In practice, you would integrate this into your argument parser. For example:

```
import argparse

parser = argparse.ArgumentParser()
# ... other arguments like model name, dataset, etc.
parser.add_argument("--mask_mode", choices=["key", "complement"], default=None,
```

```
help="Masking mode: 'key' to mask out key functional network neurons, "
                         "'complement' to mask out all but the key network. "
                         "Omit or set to None for no masking.")
parser.add_argument("--ica_mask_path", type=str, default=None,

help="Path to JSON file containing ICA-derived neuron masks. If not provided,
will compute masks.")
args = parser.parse_args()
```

In the training script, after parsing args and loading the model (and applying any LoRA/quantization as needed), you would conditionally apply the masks:

```
# Pseudo-code integration in training script
model = AutoModelForCausalLM.from_pretrained(args.model_name_or_path)
# If using LoRA or 8-bit quantization, apply those here (e.g., PeftModel or
prepare_model_for_kbit_training) before masking
if args.lora_config:
    model = prepare_peft_model(model, args.lora_config)  # hypothetical function
# Ensure any quantization (e.g., int8) is done before masking as well

mask_handles = []
if args.mask_mode:
    # Load or compute the masks
    if args.ica_mask_path:
        import json
        mask_dict = json.load(open(args.ica_mask_path))
    else:
        mask_dict = compute_ica_masks_for_model(model, train_dataset)  #
function defined below
    mask_handles = apply_ica_masks(model, mask_dict, mask_mode=args.mask_mode)
    print(f"Applied functional network masking in '{args.mask_mode}' mode.")
```

This CLI design lets the user easily switch modes. For example, to **mask the key neurons** (testing how the model performs without them), they run:

```
python finetune_with_mask.py --mask_mode key --ica_mask_path path/to/masks.json
[...other args...]
```

Or to **keep only** the key neurons active (masking the complement):

```
python finetune_with_mask.py --mask_mode complement --ica_mask_path path/to/
masks.json [...args...]
```

If `--mask_mode` is not set (or left as default `None`), the code will skip applying any masks, and the model trains normally.

Using Precomputed ICA Masks or Computing on the Fly

The functional networks are derived via **Independent Component Analysis (ICA)** on the neuron activations [4]. The user may have already performed this analysis and saved the results (e.g., as a JSON mapping of neurons to networks). Our extension supports two modes of obtaining masks:

1. **Precomputed ICA masks (JSON file):** If `--ica_mask_path` is provided, we load the JSON file. We expect it to contain the indices of neurons corresponding to key functional networks. For example, it could have a structure like:

   ```
   {
       "0": [10, 50, 123, ...],    // Neuron indices in layer 0 belonging to
   the key network(s)
       "1": [77, 250, 300, ...],  // Neuron indices in layer 1, etc.
       "...": ...
   }
   ```

   Here, keys are layer numbers (as strings) and values are lists of neuron indices identified as part of the "key" networks in that layer. This format can accommodate networks that span multiple layers (just include the respective neurons in each layer). If multiple independent components were identified as important, their neuron sets could be merged or listed separately; for simplicity, we assume a merged set of all key neurons to mask. **Note:** The JSON should align with the model architecture (e.g., if the model has N layers, keys `"0"` through `"<N-1>"` correspond to those layers' MLPs).

2. **On-the-fly ICA computation:** If no mask file is given, the extension can compute the masks at the start of training. This involves:

3. **Collecting neuron activations:** We run the unmodified model on a sample of data and record the output of every MLP neuron. Specifically, we pass a batch (or several batches) from the training dataset through the model in evaluation mode, and capture the **intermediate layer outputs** of each MLP. Each neuron's activations across many inputs form a time series analogous to fMRI signals [4]. We can use forward hooks (similar to above) or modify the model to output these intermediate activations. For example:

   ```
   activations = {layer: [] for layer in range(num_layers)}
   def capture_hook(module, inp, out, layer_idx=None):
       # out is the intermediate activation (before projection) of shape
   [batch, seq_len, intermediate_dim]
       activations[layer_idx].append(out.detach().cpu().numpy())
   # Attach hooks on the MLP intermediate output (after activation). If not
   ```

```
directly accessible, attach on first linear to get its output pre-
activation.
```

We would attach `capture_hook` to the appropriate point (e.g., after the first MLP linear + activation, or as a forward hook on the second linear to grab its input as in our masking but before applying mask). Accumulate these outputs over a sample of the dataset (covering different tasks if possible).

4. **Performing ICA:** We treat the collected activation matrix for each neuron as *mixed signals* and apply Fast Independent Component Analysis (FastICA) to decompose them into independent source components [2] [11]. In practice, we flatten the activations such that we have shape `[num_timepoints, num_neurons]` (timepoints are all token positions across the sampled data, and num_neurons = total MLP neurons across the model or per layer). FastICA will return a set of components (say, K components) and a mixing matrix. Each independent component corresponds to a functional network, with a **spatial map of neuron weights** indicating how strongly each neuron contributes to that component [12] [13]. We use `sklearn.decomposition.FastICA` for convenience:

```
import numpy as np
from sklearn.decomposition import FastICA
X = np.concatenate([np.concatenate(activations[layer], axis=0) for layer in
range(num_layers)], axis=1)
# X shape: (total_timepoints, total_neurons) if combining all layers.
ica = FastICA(n_components=K, random_state=0)
S = ica.fit_transform(X)        # shape: (timepoints, K), independent
signals
A = ica.mixing_                 # shape: (num_neurons, K), mixing matrix
```

(Alternatively, one can run ICA per layer on each layer's activations separately. The code above combines all neurons; for per-layer, run FastICA on each layer's activation submatrix.)

5. **Deriving binary masks:** For each independent component's weight vector (a column in `A`), we identify the neurons with the largest weights in absolute value. We can threshold by percentile or standard deviation – e.g., select neurons whose absolute weight > 2σ above mean, or the top X% of weights [13]. These neurons are considered as forming that functional network (they exhibit *synchronized activity* indicating a network [12]). Once we identify the neuron indices for a component, we add them to our mask dictionary. If multiple components are deemed **"key networks"**, we might select them based on explained variance or performance impact. For example, Liu *et al.* (2025) found that certain components (often <2% of neurons) are **key** – masking them causes a significant performance drop [14]. We can simulate this by measuring validation loss or accuracy drop when ablating each component one by one, and pick the component(s) whose removal hurts performance most (those are likely "key").
6. **Output mask_dict:** Combine the selected neurons (from one or more top components) into the `mask_dict` structure as described above, then proceed to apply the masks with `apply_ica_masks`. It's wise to save this `mask_dict` to a JSON file for reproducibility (so next run can use precomputed masks).

Note on computing ICA: This can be time-consuming for large models, so it's recommended to either use a smaller subset of the model (e.g., fewer layers) or a limited dataset sample, or perform this analysis offline. In our integration, if `--ica_mask_path` is not provided, we call a `compute_ica_masks_for_model(model, train_dataset)` function (as referenced in the code snippet) to handle these steps. This function should implement the logic above: run a forward pass on a portion of `train_dataset`, perform ICA, identify key neurons, and return the mask dictionary. For clarity, we won't provide the full implementation here, but the pseudocode above outlines how it can be done using **FastICA** and thresholding [13].

By default, we might use a moderate number of components (e.g. `K=20`) for ICA. The user can adjust this or other parameters (perhaps expose `--ica_components` as an argument). We also emphasize that *computing the masks is ideally a one-time preprocessing step* – once derived, the masks can be reused across runs.

Integration with Hugging Face SFTTrainer (LoRA & Quantization Compatible)

The masking extension is designed to be **minimally invasive** and compatible with the user's existing fine-tuning pipeline (which uses Hugging Face's `SFTTrainer`, likely from the TRL library). Here are key points for integration:

- **Model loading and preparation:** Load the model as usual (e.g., via `AutoModelForCausalLM.from_pretrained`). If you are using **LoRA** (via PEFT) or **4-bit/8-bit quantization**, apply those *before* injecting the masking hooks. For example, if using QLoRA, you might call `prepare_model_for_kbit_training` (which replaces certain `Linear` modules with quantized versions) and then `get_peft_model` to add LoRA adapters. After these, the model's modules (like `Linear8bitLt` instead of `Linear`) are in place – our `apply_ica_masks` function already checks for `Linear8bitLt` by name and treats it like a linear. The forward pre-hook approach works regardless of parameter data type or LoRA injection, because it operates on the forward activations. The mask simply multiplies the combined output of base weights + LoRA weights (for that neuron) before it goes into the final projection, thereby zeroing it out. This means **LoRA-added weights for a masked neuron will also have no effect**, which is expected.
- **Trainer integration:** You don't need to subclass the trainer – simply ensure the model passed to `SFTTrainer` has the hooks attached. For example:

  ```
  trainer = SFTTrainer(model=model, train_dataset=train_dataset,
  args=training_args, data_collator=collator, ...)
  trainer.train()
  ```

  Since we modified the model in-place with `apply_ica_masks` *before* creating the trainer, the trainer will use the masked-forward model for training. From the trainer's perspective, everything is normal (loss computation, backprop, etc.), except that certain neurons are consistently zeroed out. This seamless integration is possible because hooks are internal to the model's forward pass.
- **Removing hooks (optional):** If you later want to save the model or use it for inference without masks, you can remove the hooks by calling `handle.remove()` on each stored handle in

`mask_handles`. Our `apply_ica_masks` returns the list of hook handles. For example, after training:

```
for h in mask_handles:
    h.remove()
```

Now the model will function as originally (which might be useful for evaluation to compare masked vs unmasked performance).

- **Logging:** It's good practice to log which neurons or what percentage of neurons are being masked, for traceability. For instance, you could output something like: *"Masking 2% of neurons (80/4096 per layer) corresponding to key functional network."* This aligns with the findings that masking <2% can significantly impair performance [14], so it will help interpret results.

By following these steps, the functional network masking extension should **integrate smoothly** with the user's training script, not interfering with dataset preparation, logging, or other trainer callbacks. It basically acts as an augmentation to the model architecture at training time.

Example: Fine-Tuning with Masking on a Logic/Code Task

To illustrate usage, let's consider fine-tuning a model on a *non-standard* downstream task – e.g. a logical reasoning puzzle or a code understanding problem – using the Hugging Face `datasets` library. Such tasks are less likely to be already solved by the model's pretraining, making the effect of masking clearer. In the paper by Liu *et al.*, they used data from **MathQA (math reasoning)** and **CodeNet (code understanding)** among others [11] to probe functional networks. Here, we'll demonstrate with a dataset from the **BIG-Bench Hard** suite which contains challenging logical deduction tasks (as hosted on HuggingFace datasets, e.g., `"maveriq/bigbenchhard"`). The steps are:

1. **Select dataset:** We choose a task like *Logical Deduction* from BIG-Bench Hard or a code-related dataset (for example, CodeNet-16K if accessible, or a smaller code classification task). For this example, assume a dataset `bigbenchhard` with a configuration for logical deduction. We load it via `datasets.load_dataset`.
2. **Format data for SFT:** The SFTTrainer likely expects a certain format (maybe a single text field with instruction and response). We ensure the dataset is preprocessed into the right format (the user's provided script presumably handles this, formatting prompts and completions).
3. **Run training with masks:** We will call our script with the `--mask_mode` argument. For demonstration, let's say we want to **keep only the key network active** during training to see if the model can still learn the task with a tiny fraction of neurons:

```
python finetune_with_mask.py \
    --model_name_or_path huggyllama/llama-7b \
    --dataset_name maveriq/bigbenchhard --dataset_config logical_deduction
\
    --output_dir outputs/llama_logic_masked \
```

```
    --mask_mode complement --ica_mask_path key_neurons_llama7b.json \
    --per_device_train_batch_size 2 --num_train_epochs 3 [other args...]
```

This will load LLaMA-7B, apply LoRA (if configured) and our mask hooks such that only the key functional neurons (as defined in `key_neurons_llama7b.json`) are active. The model is then fine-tuned on the logical deduction task for 3 epochs. During training, we expect a lower performance initially because most neurons are masked, but if those key neurons indeed carry meaningful functionality, the model may still learn to some extent (as observed by Liu *et al.* that retaining just a subset of networks can maintain operation [8]). One could compare this to a baseline fine-tune without masks, or with `--mask_mode key` (masking out the key neurons) to see the difference in learning curve.

4. **Evaluate results:** After training, evaluate the model on validation data or test questions. If masking had a strong effect, you might see that **masking key neurons hurts accuracy** (for `mask_mode=key` run) or that **training with only key neurons is limiting** but perhaps the model still achieves some non-zero score (for `mask_mode=complement` run). These outcomes would reinforce the notion that those neurons were indeed crucial [14].

Below is a concise example code snippet for loading a dataset and initiating an SFTTrainer with our masking extension:

```
from datasets import load_dataset
from trl import SFTTrainer, SFTTrainingArguments

# 1. Load dataset (e.g., logical deduction task)
dataset = load_dataset("maveriq/bigbenchhard", "logical_deduction")
# For SFT, we might need to combine prompt and target into a single text or
dict;
# assume dataset is ready or use a preprocessing function.

# 2. Initialize model (with LoRA and quantization if needed)
model = AutoModelForCausalLM.from_pretrained(args.model_name_or_path)
# (Apply LoRA prep if any, not shown for brevity)

# 3. Apply functional network masking
mask_dict = json.load(open(args.ica_mask_path)) if args.ica_mask_path else
compute_ica_masks_for_model(model, dataset["train"])
mask_handles = apply_ica_masks(model, mask_dict, mask_mode=args.mask_mode)

# 4. Set up Trainer (using Hugging Face Transformers Trainer or TRL SFTTrainer)
training_args = SFTTrainingArguments(
    output_dir=args.output_dir,
    num_train_epochs=3,
    per_device_train_batch_size=2,
    logging_steps=50,
    evaluation_strategy="epoch",
```

```
    save_strategy="epoch",
    # ... other training args
)
trainer = SFTTrainer(model=model, args=training_args,
                     train_dataset=dataset["train"],
eval_dataset=dataset["validation"],
                     data_collator=your_data_collator)
trainer.train()

# After training, you can remove hooks if you want to use the model without
masking:
for h in mask_handles:
    h.remove()
```

In this example, we fine-tune on the *logical_deduction* task. One could similarly use a code understanding task – for instance, a subset of CodeNet (if available) or a code-related challenge from BIG-Bench Hard (there are tasks involving code as well). The key is that these tasks require reasoning or knowledge not directly trivial from pretraining, so we can observe how masking neurons influences the learning.

Isolating effect of masking: Because we chose an unusual domain (logic/code), the model likely needs to adapt its weights significantly to succeed. By training with different mask settings, we isolate the contribution of certain neurons: - If `mask_mode=key` (key neurons off), we expect the model struggles more to learn (if those neurons are truly important) – e.g., slower reduction in loss or lower final accuracy. - If `mask_mode=complement` (only key neurons on), we are essentially training a pruned model that uses <10% of its MLP neurons (as suggested by Liu *et al.*) [8] . We might see the model can still learn the task but perhaps with a performance drop compared to using all neurons. Interestingly, prior research noted that gradually **adding back** more networks (unmasking additional neurons) raises performance from low to high [8] . You could experiment by fine-tuning with incremental unmasking to confirm this behavior.

Throughout training, all other aspects (optimizer, learning rate, LoRA updates, etc.) function as normal. The masking simply forces the model to operate with a restricted (or specifically ablated) set of features in each layer's MLP.

Conclusion

We implemented a fine-tuning extension that integrates **ICA-derived functional network masking** into Hugging Face models. The approach uses forward hooks to zero out targeted neurons during the forward pass, guided by ICA analysis that identifies groups of co-activating neurons (functional networks). A CLI flag allows toggling between *masking the key networks* (ablating them) and *masking the complement* (using only the key networks), enabling comparative experiments. The extension is designed to work with existing fine-tuning setups, including LoRA and quantization, by inserting itself post-model-loading and prior to training.

By focusing on challenging tasks like logical reasoning or code understanding (e.g., MathQA, BIG-Bench Hard tasks, CodeNet snippets), we ensure that the impact of these masks is observable without being overshadowed by the model's prior knowledge. This extension not only helps validate findings from **"Brain-Inspired Exploration of Functional Networks and Key Neurons in LLMs"** – such as that masking <2% of

neurons can cripple performance [14] – but also provides a tool for researchers to intentionally fine-tune models on specific sub-networks, potentially leading to more efficient or interpretable LLMs [8] [15].

Sources:

- Liu *et al.*, *Brain-Inspired Exploration of Functional Networks and Key Neurons in Large Language Models*, 2025 [1] [5] [14] – Introduces the concept of functional networks in LLMs and demonstrates that masking key neuron networks (<2% of neurons) significantly impacts performance, while a small subset of networks can sustain operation.
- Liu *et al.*, *Supplemental: Mapping Code to Experiments and Proposed Fine-Tuning Extensions*, 2025 – Provided implementation insights for connecting the ICA-based analysis with fine-tuning modifications (e.g., recommended use of forward hooks and integration with Hugging Face training pipelines).
- Song *et al.*, 2024 – Demonstrated that directly manipulating neuron outputs (masking/amplifying) can control LLM behavior [7], motivating our forward-hook masking strategy.
- FastICA algorithm – Used to decompose neuron activation signals into independent components corresponding to functional networks [2], with thresholding to identify the most active neurons per network [13].
- Datasets: AGNEWS, WikiText2, MathQA, CodeNet – used by Liu *et al.* to find functional networks spanning diverse domains [11]. In our example, we choose analogous tasks (logical deduction, code understanding) from Hugging Face datasets to illustrate fine-tuning under masked conditions.

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [2502.20408] Brain-Inspired Exploration of Functional Networks and Key Neurons in Large Language Models
https://ar5iv.labs.arxiv.org/html/2502.20408