ChatGPT

Mapping Code to Experiments and Proposed Fine-Tuning Extensions

Code Correspondence to Experimental Procedures

The GitHub code (notably in `llmfact/utils.py` and related modules) aligns with the key experiments from *"Brain-Inspired Exploration of Functional Networks and Key Neurons in LLMs."* Below is a mapping between each experimental procedure and the corresponding code blocks or functions:

Experiment / Procedure	Corresponding Code (Functions or Blocks)
ICA-Based Functional Network Extraction	*Independent Component Analysis for functional networks*: The code uses **FastICA** (from scikit-learn) to decompose neuron activations into independent components. In `FBNFeatureExtractor.fit`, the model's MLP layer outputs are first collected via hooks (using `LayerOutputExtractor`) and then passed to `FastICA` to extract latent components [1]. The resulting mixing matrix (component loadings for each neuron) is **normalized and thresholded** (e.g. values below 1.96 standard deviations set to 0) to create binary masks that indicate which neurons belong to each independent functional network [2]. Utility functions `apply_mask_and_average` and `apply_mask_any` (defined in *utils.py*) then apply these masks to the activation matrix – either zeroing out non-network neurons and averaging activations per network, or isolating network-specific features without averaging [3] [4]. These steps implement the ICA-based functional network extraction described in the paper.
Neuron Masking (Ablation) Experiments	*Masking key neurons/networks*: Implemented via the `MaskedGPT2Model` class (see `llmfact/mask.py`). This wrapper class injects forward **hooks** into specified transformer layers (in practice, the MLP output layers) and uses a pre-computed mask to zero-out certain neuron activations during the forward pass [5]. The core is `mask_hidden_states`: it takes the hook's output and fills selected positions (those neurons belonging to a target functional network) with 0.0 [6]. By registering such hooks with a binary mask (1 for neurons to mask), the code effectively **silences** the critical neurons or entire functional networks during inference. This corresponds to the experiments where "masking key functional networks" was done to assess performance drop – the code can wrap a GPT-2 (or other model) and generate or evaluate outputs with those neurons ablated.

Experiment / Procedure	Corresponding Code (Functions or Blocks)
Preservation (Network Retention) Experiments	*Retaining only certain networks*: While there isn't a single explicit function named "preservation," the code supports this via its masking utilities. To **preserve** a subset of functional networks, one can invert the mask logic – i.e. mask *all other* neurons except the chosen network(s). The binary masks from the ICA step can be combined for this purpose. For example, to keep one key network active, a mask is constructed that zeros out all neurons **not** in that network. The `LLMFC` helper (in `llmfact/stat.py`) demonstrates combining mask matrices: it expands a mask for each network and multiplies it with the layer activations to isolate each network's signal [7], then averages the neurons in each network to get a time-series activation (summing over neurons, divided by count) [8]. Using the same mask mechanism in `MaskedGPT2Model`, the experiment of "retaining just a subset of networks" can be done by gradually unmasking (activating) additional networks. In practice, one would start with only the essential network's neurons unmasked and all others forced to zero, then incrementally include more networks and observe performance improvement – mirroring the paper's preservation experiments.

Sources: The code is from the open-source implementation of the paper [9] [10], and the mapping above is based on functions in the repository's `llmfact` package. Key portions of code are cited inline to show how ICA decomposition, neuron masking, and network retention are realized in code.

Extending the Codebase for Selective Neuron Fine-Tuning

Building on this code, we can extend it to support **fine-tuning large language models** (LLMs) with PyTorch and Hugging Face Transformers, in ways that selectively update or freeze neurons based on their functional network importance. The idea is to either: (a) **focus training on critical neurons/networks** identified by ICA, or (b) **freeze those critical neurons** and train the remaining (less-utilized) parts of the network. Below we discuss practical considerations for implementing these extensions:

Applying Masks in Training: Forward vs. Backward

To incorporate neuron masks during fine-tuning, we can apply them at different stages of the training process: - **Masking during the forward pass (pre-gradient):** This is analogous to the current `MaskedGPT2Model` approach – we insert a mask in the forward computation to zero-out certain neuron activations. During fine-tuning, doing so will not only silence those neurons' contributions in the output, but also naturally *zero their gradients* during backpropagation (since masked-out outputs do not influence the loss). Essentially, masked neurons are effectively frozen because their output (and hence error signal) is 0. This approach is straightforward: for example, one could reuse the `register_hooks` mechanism to apply a binary mask to the hidden state outputs of each layer before the loss is computed. If a neuron's output h_i is set to 0 in forward pass, the gradient $\partial \text{loss} / \partial h_i$ will be zero as well. - **Masking gradients after the forward pass (post-gradient):** Alternatively, one could allow the forward pass to proceed normally (using all neurons), but then manually zero-out or scale the **gradients** for certain neurons/parameters during backpropagation. In PyTorch, this can be done by hooking into the `torch.nn.Parameter.grad` after `loss.backward()` – e.g., setting `param.grad[w] = 0` for specific weight indices, or using registered backward hooks on modules. This ensures the selected neurons' weights don't get updated even though they participated in the forward computation. Masking at the gradient level achieves a similar effect to forward masking in terms of freezing, but it incurs the overhead of computing those activations and their gradients (only to discard them). For efficiency, masking in the forward pass (so that those neurons contribute nothing and receive no gradient) is usually preferable to save computation.

Straight-Through Estimators (STE) for Non-Differentiable Masks: If we decide to use a *binary mask* in the forward pass (which is a non-differentiable operation), the scenario arises: what if we wanted gradients to flow **through** the masking operation (for example, if we were trying to *learn* which neurons to mask via gradient-based methods)? In our case, when we are explicitly freezing or activating predefined neurons, we *don't actually need* gradient flow through the mask – we want certain neurons truly off. However, for completeness, one could employ a straight-through estimator if needed. A **straight-through estimator** treats the masking function as identity in the backward pass [11]. In other words, if $m$ is a binary mask (0/1) applied to an activation $h$ (producing $h'=m*h$), an STE in backward would ignore the derivative of $m$ (which is zero almost everywhere) and pass the gradient from $h'$ back to $h$ as if $m=1$ (identity). This allows gradient-based optimization of a masking parameter by bypassing the discontinuity [12] [11]. In practice, using STE would be relevant if we introduce learnable gates for neurons. For the proposed fine-tuning strategy (selective freezing/updating), it's simpler to treat the mask as fixed and non-trainable – thus we can apply hard masking without needing an STE, because we *want* those neurons truly frozen (no gradient). If one did want to gradually tune the mask (e.g., to softly reduce certain neurons' influence rather than hard 0/1), techniques like using a sigmoid-based mask or STE could be explored, but this adds complexity and potential instability in training.

Integrating with Hugging Face Transformers Training

Hugging Face's `Trainer` API and the underlying PyTorch model make it possible to selectively freeze or train subsets of parameters: - **Freezing specific neurons via parameter freezing:** Each neuron's contribution in an MLP corresponds to certain weight parameters in the model. For example, a single feed-forward neuron (in the intermediate dense layer of the transformer) has an incoming weight vector (in the first linear layer) and an outgoing weight vector (in the second linear layer). To freeze a neuron, we should freeze both sets of parameters. Practically, one can identify the indices of critical neurons (from the ICA results) and then set the corresponding rows/columns in the weight matrices to `requires_grad=False`. For instance, if neuron $j$ in layer $\ell$ is critical and we want to freeze it, we would disable grad for the $j$th column of $W_{\ell}^{(in)}` (the input-to-hidden weight matrix) and the $j$th row of $W_{\ell}^{(out)}$ (the hidden-to-output matrix). In Hugging Face's `Trainer`, we can achieve this by modifying the model's parameters before training: e.g.,

```
for name, param in model.named_parameters():
    if ... condition on name (layer ℓ weights) and index in [j] ...:
        param.requires_grad = False
```

```
 This ensures the optimizer ignores those parameters. An even coarser approach is
to freeze entire layers or modules if needed (but here we want neuron-level
granularity).
- **Using the masking hooks during training:** Alternatively, one could incorporate
```

the existing masking mechanism (forward hooks that zero-out activations) directly into the model during fine-tuning. For example, wrap the model in MaskedGPT2Model and supply the mask matrix for neurons you want to freeze. Then train this wrapped model normally. The hooks will zero those neurons on each forward pass, effectively preventing them from learning. This approach has the benefit that you don't need to surgically manipulate individual weight gradients – the mask ensures those neurons simply never activate (and thus their weights get zero gradients). When using Trainer , you can still freeze parameters as above, but if the mask is already nullifying their effect, it might be redundant – still, explicitly freezing the parameters is safer to avoid any optimizer updates (and can slightly save computation by not computing momentum/adam updates for them).
- **Focusing updates on critical neurons (unfreezing only some):** In the converse scenario where we **only train the critical neurons** and freeze everything else, one can do the inverse mask if using hooks (mask out all non-critical neurons' activations so they don't learn, or simply freeze their weights). A simpler method is to exclude all other parameters from the optimizer. Hugging Face Trainer allows passing a custom list of params to the optimizer or one can set requires_grad=False for all but the chosen neurons. For example, to fine-tune only the top 2% neurons identified as crucial, mark all other weights requires_grad=False prior to training. This effectively performs a **sparse fine-tuning** where only a small subset of the model's neurons (and associated weights) get updated. This kind of targeted training can be integrated into Trainer by overriding the optimizer_step or by providing a custom optimization loop if needed (but simply freezing via requires_grad=False is usually sufficient).
- **Custom training loop:** If not using Trainer , a manual training loop can give even more control. One can compute the loss, call loss.backward() , then manually zero out certain gradients (e.g., model.layer[k].weight.grad[j] = 0) before optimizer.step() . This post-backward masking achieves selective training as well. The loop might look like:

```
optimizer.zero_grad()
outputs = model(**batch)
loss = outputs.loss
loss.backward()
# Manually zero grads for frozen neurons:
for idx in frozen_indices:
    model.transformer.h[ℓ].mlp.c_fc.weight.grad[:, idx] = 0
    model.transformer.h[ℓ].mlp.c_proj.weight.grad[idx, :] = 0
optimizer.step()
```

This ensures neuron idx in layer ℓ doesn't update. Managing this for many neurons and layers is tedious, so automating via masks or setting requires_grad=False` on those slices is preferable.

Practical Examples and Strategies to Reduce Computational Cost

Exploring learning dynamics in functional vs. non-functional neuron groups can be expensive if done naïvely on a full LLM, but there are strategies to mitigate this: - **Train fewer parameters:** One clear advantage of focusing on critical neurons is that they are typically a very small fraction of all neurons (the paper notes often <2% of neurons form the key networks [13]). Fine-tuning only this tiny subset means the number of trainable parameters is dramatically lower (for example, 2% of neurons in GPT-2 corresponds to 2% of the MLP weight rows/cols, which is a ~2% subset of those weight matrices). This not only speeds up training (fewer gradients to compute and apply) but also reduces memory usage. It's similar in spirit to parameter-efficient tuning methods – here we're choosing neurons based on functional importance. - **Freeze vs. unfreeze trade-off:** Conversely, when we freeze the critical neurons and train the *other 98%*, we aren't reducing parameter count by much. The motivation in that scenario is different: we want to see if the network can "re-route" and learn to perform the task using previously dormant or less-used neurons. To keep this experiment efficient, one might consider freezing the critical 2% and also perhaps not fully training all remaining layers from scratch but using a smaller learning rate or fewer training steps (since the model without the key neurons may learn slower or differently). Monitoring validation performance early can tell if the non-critical neurons are picking up the slack. - **Efficient mask implementation:** If using the forward mask approach, the additional cost of the hook operation and elementwise masking is relatively small (just a tensor fill or multiplication). The main compute cost in a transformer comes from matrix multiplies; masking neurons doesn't remove those multiplies (the model still multiplies by weights for masked neurons, then we zero the output). If we were extremely concerned with efficiency, one could physically prune those neurons from the model for the duration of training (to avoid even computing their outputs). However, pruning weights in Hugging Face models would require custom model surgery (e.g., replacing the weight matrices with smaller ones) – this is doable but beyond a simple extension. In practice, leaving the neurons in and just masking will be fine for most experiments, given that 98% of the work still happens and we're mostly saving on gradient updates. The **gradient computation** for masked neurons will be minimal because their output was zero, but the forward pass had their calculation. Therefore, the difference between hard-masking vs. actual weight freezing on forward compute is negligible unless a very large portion of neurons are masked. Since we're talking about small fractions, it's acceptable. - **Using built-in Trainer features:** The Hugging Face Trainer doesn't (as of now) have a built-in argument to freeze specific neurons, but it's straightforward to freeze entire layers (e.g., all but the last few layers). For neuron-level freezing, as discussed, one would handle it via the model's `forward` or parameter settings. One can also utilize callbacks in the Trainer to, say, unfreeze additional networks progressively if one wanted to mimic the incremental network addition experiment. For example, you could start a run with only network A active, then at a certain epoch callback, unfreeze network B (update the mask to allow network B's neurons, or change those parameters to trainable) and continue training, and so on. This would simulate the preservation experiment in a training setting – observing how performance improves as more functional networks are allowed to train. - **Monitoring and dynamic adjustment:** A practical example could involve fine-tuning an LLM on a downstream task with three setups: (1) train only the top-N neurons (most important network) – a fast, lightweight fine-tuning; (2) freeze those N and train the rest – to see if performance lags or if other neurons compensate; (3) full model fine-tuning – as a control. During these, one should monitor metrics like loss and accuracy. It's possible that training only key neurons converges faster (since those have high influence on outputs), but might plateau at lower accuracy if other neurons needed adjustment; whereas training all except key neurons might start off worse but potentially catch up if the model can reorganize. To minimize cost, one might use a smaller learning rate or early stopping in scenario (2) if it's clear the model is struggling without the key neurons. - **Batching and gradient accumulation:** Since we are potentially freezing a lot of parameters, one could increase the batch size or use gradient accumulation to better utilize the available computation (the frozen parameters won't contribute to gradient all-reduce, etc., so you might handle larger batches with the same memory). However, these are general training considerations.

In summary, extending the codebase for selective fine-tuning involves using the same mask-based neuron selection to control *which parts of the model learn*. By masking or freezing, we can **focus learning on critical functional networks** or deliberately **exclude** them to stress-test the model's unused capacity. The integration with Hugging Face's Trainer can be achieved by wrapping the model or freezing parameters, and careful use of hooks or gradient manipulation ensures that the chosen neurons are updated as intended. This approach opens up a lightweight fine-tuning paradigm (training a tiny fraction of neurons) and a targeted analysis tool (observing how learning dynamics shift when key neurons are fixed), all while keeping computational costs manageable through selective parameter updates.

References: The strategies above draw on standard practices for freezing layers in PyTorch and the concept of straight-through estimators for handling non-differentiable operations [12] [11]. These techniques, combined with the functional network masks identified by ICA, provide a powerful way to conduct controlled fine-tuning experiments on large language models.

---

[1] [2] [3] [4] [5] [6] [7] [8] add mask fc group-wise extractor · WhatAboutMyStar/LLM_ACTIVATION@fe5fbad · GitHub
https://github.com/WhatAboutMyStar/LLM_ACTIVATION/commit/fe5fbad60765f7d44336baaccf618bace12ac9e0

[9] GitHub - WhatAboutMyStar/LLM_ACTIVATION: The implement of "Brain-Inspired Exploration of Functional Networks and Key Neurons in Large Language Models"
https://github.com/WhatAboutMyStar/LLM_ACTIVATION

[10] [2502.20408] Brain-Inspired Exploration of Functional Networks and Key Neurons in Large Language Models
https://ar5iv.org/abs/2502.20408

[11] [12] Intuitive Explanation of Straight-Through Estimators with PyTorch Implementation | by Hassan Askary | Medium
https://hassanaskary.medium.com/intuitive-explanation-of-straight-through-estimators-with-pytorch-implementation-71d99d25d9d0

[13] 2502.20408v1.pdf
file://file-3V3JGtTA4wAH5kdq82N2Xw