Member-only story

Transformer-Squared: Stop Finetuning LLMs

The architecture behind self-adaptive LLMs, the math and code behind Transformer-Squared, and Single Value Decomposition.

Cristian Leo · Following

Published in Level Up Coding · 23 min read · 1 hour ago

Image generated by DALL-E

We've reached a point where AI can write surprisingly coherent prose, answer complex questions, and generate code. It's genuinely impressive. But if you've spent some time actually using these models for specific tasks, you've probably run into the same thing I have: **the inevitable need to finetune**. Want your LLM to be a coding whiz? Finetune it. Need it to ace math problems? More finetuning. It's become the de facto approach, almost a reflex in the field.

And to be fair, finetuning works. It's how we've been able to take these general-purpose giants and mold them into something more specialized and, dare I say, useful in practical scenarios. However, if we take a step back, it's worth asking: is this constant cycle of finetuning really the most efficient or even the most elegant solution in the long run? It's computationally expensive, time-consuming, and each time we finetune, we're essentially creating a new, somewhat static version of the model. What happens when we need it to pivot, to adapt to a slightly different task or a completely new domain? Well, you guessed it — more finetuning. It feels a bit like constantly rebuilding a car engine every time you want to drive on a different type of road.

Perhaps there's a smarter way. What if, instead of these intensive, all-encompassing retraining sessions, our LLMs could learn to adapt on the fly, almost like they're thinking in real time about the task at hand and adjusting their internal mechanisms accordingly? This is precisely the intriguing proposition put forward in a recent paper, "Transformer-Squared: Self-adaptive LLMs" by Sun, Cetin, and Tang from Sakana AI. They introduce a framework called Transformer^2 that dares to suggest we might be able to move beyond the constant need for finetuning.

The core idea revolves around **Singular Value Decomposition**, or SVD. Don't worry if that sounds intimidating; we'll break it down later. Essentially, Transformer^2 allows the LLM to dynamically tweak specific components of its internal workings — think of it as adjusting a few key knobs and dials — during the actual process of answering a question, rather than undergoing a full-scale parameter overhaul beforehand. They achieve this by training what they call "expert vectors" using a clever technique called Singular Value Fine-tuning, combined with Reinforcement Learning. It's a two-pass system: the model first analyzes the incoming task, figures out what's needed, and *then* adapts itself before generating the final output.

This approach is noteworthy for a few reasons. Firstly, it promises efficiency. Imagine the resources we could save if we didn't have to finetune for every single niche application. Secondly, it hints at a new level of versatility — models that are inherently more flexible and less rigidly specialized. And thirdly, it might just be a step closer to how we, as humans, actually learn and adapt — by dynamically adjusting our thinking based on the context, rather than completely rewriting our knowledge base for every new challenge. Of course, as with any new approach, there are questions. How robust is this method in practice? Does it truly generalize across diverse tasks? And what are the limitations? These are precisely the questions we're going to explore together in this article. We'll dive into the math that makes Transformer^2 tick, unpack the code concepts, and discuss what this innovative framework could mean for the future of AI. So, are you ready to explore a potential future where our LLMs become masters of self-adaptation? Let's delve in.

Transformer-Squared

Animation extracted from original repo of Transformer-Squared

Transformer², by Sun and colleagues, propose a paradigm shift. Instead of seeing LLM adaptation as a process of wholesale retraining, they envision a model that can nimbly adjust itself to new tasks *as it encounters them*. It's a bit like moving from a fixed-gear bicycle to one with gears — suddenly, you can adapt to hills, headwinds, and varying terrains without fundamentally changing the bike itself, just by intelligently shifting gears. Transformer² aims to equip LLMs with a similar kind of dynamic gear-shifting mechanism.

The magic, if you can call it that, starts with a rather elegant mathematical concept: **Singular Value Decomposition**, or SVD. Think of SVD as a way to dissect a matrix — and in the world of neural networks, weight matrices are everywhere — into its fundamental components. Imagine you have a complex recipe. SVD is like breaking it down into a set of simpler, orthogonal cooking techniques, each contributing a unique flavor or texture to the final dish. Mathematically, SVD tells us that any matrix, let's call it W, can be decomposed into three other matrices: *U*, Σ, and V^T. Crucially, Σ is a diagonal matrix containing what we call "singular values." These values, in a sense, represent the "importance" or "strength" of each of these fundamental components. It's like identifying the core ingredients and techniques that truly define your complex recipe.

Transformer^2 leverages this SVD insight through a technique called **Singular Value Fine-tuning**, or SVF. The core idea of SVF is simple, but perhaps a little counterintuitive at first glance. Instead of tweaking all the countless parameters of a weight matrix during finetuning — which is what traditional methods and even some parameter-efficient methods often do — SVF focuses solely on adjusting these singular values, the Σ part in our UΣV^T decomposition. They do this using what they term "expert vectors." Imagine you have a set of specialized chefs, each an expert in a particular cuisine. These "expert vectors" are like the culinary profiles of these chefs, encoding their specific skills and styles. During training, using Reinforcement Learning, Transformer^2 learns these "expert vectors," each tailored to a particular type of task — say, coding, math, or reasoning. The brilliance here is in its efficiency: by only learning these relatively small "expert vectors" to modulate the singular values, SVF becomes incredibly parameter-efficient compared to methods that modify larger chunks of the model's weights. It's like giving each chef a small set of seasonings that subtly alter their base recipes, rather than rewriting entire cookbooks for every new dish.

But how does this translate to real-time adaptation? This is where the **two-pass inference mechanism** of Transformer^2 comes into play. Think of it as a two-step process: "understand the task, then adapt and answer." In the **first pass**, when you give Transformer^2 a prompt — a question, a request, whatever it may be — the model runs through its initial processing steps, using its base, pre-trained weights. This first pass isn't about generating the final answer yet; it's more about "assessing the situation," trying to understand the *nature* of the task at hand. Based on this initial assessment, Transformer^2 then moves to the **second pass**. Here, the model dynamically selects and mixes the pre-trained "expert vectors" we talked about earlier. If the first pass suggests the task is math-heavy, it might emphasize the "math expert vector." If it seems to require logical reasoning, it might lean towards the "reasoning expert vector." This mixing of expert vectors effectively adjusts the model's weight matrices on the fly, tailoring them to the specific demands of the incoming prompt. Then, with these adapted weights, the model performs the second pass, generating the final answer.

The potential benefits of this Transformer^2 approach are quite compelling. Firstly, and perhaps most practically, it promises **efficiency**. By sidestepping full-scale finetuning and focusing on these lightweight expert vectors, it could significantly reduce the computational burden of adapting LLMs to new tasks. Secondly, it hints at a new kind of **real-time adaptability**. Imagine an LLM that can fluidly switch between different skill sets without needing to be explicitly retrained for each. This could lead to more versatile and responsive AI systems. And thirdly, there's the allure of **versatility**. The paper suggests that Transformer^2 isn't tied to a specific LLM architecture; it could potentially be applied across different models and even modalities, like vision-language tasks.

Of course, it's still early days. We need to rigorously evaluate how well Transformer² performs in practice, across a wide range of tasks and real-world scenarios. Are these "expert vectors" truly capturing meaningful skills? How robust is this dynamic adaptation mechanism? And are there limitations we haven't yet uncovered? These are crucial questions, and to begin to answer them, we need to delve a little deeper, starting with the mathematical heart of Transformer². Let's unpack those formulas and see what's really going on under the hood.

The Math Behind Transformer-Squared

Alright, let's put on our math hats for a moment and peek under the hood of Transformer². Don't worry, we'll keep it as grounded as possible, focusing on the core ideas without getting lost in overly dense equations. The goal here is to understand the mathematical operations that enable this dynamic adaptation and to appreciate both the elegance and the potential limitations of this approach.

As we discussed, the foundation is **Singular Value Decomposition (SVD)**. If we have a weight matrix *W*, imagine it as a transformation that reshapes and reorients data as it flows through the neural network. SVD allows us to decompose this complex transformation into a sum of simpler, more fundamental transformations. Mathematically, we express this as:

$$W = U\Sigma V^T$$

Singular Value Decomposition — Image by Author

Let's break down each component:

- *U* and *V* are what we call semi-orthogonal matrices. Think of them as representing sets of orthogonal axes — directions in space that are perpendicular to each other. *U* captures the output directions, and *V* captures the input directions.
- Σ is the crucial part of our discussion. It's a diagonal matrix, meaning it only has values along its diagonal, and zeros everywhere else. These diagonal values, *σ_i*, are the **singular values**. They are always non-negative and are typically ordered from largest to smallest. Each *σ_i* essentially scales the contribution of the corresponding pair of columns from *V* and columns from *U*.

So, what does this mean in practice? When we perform a matrix multiplication *y=Wx*, we can rewrite it using the SVD decomposition as:

$$y = Wx = (U\Sigma V^T)x = \sum_{i=1}^{r} \sigma_i u_i v_i^T x$$

SVD with matrix multiplication — Image by Author

Here, *u_i* and *v_i* are the *i*-th columns of *U* and *V* respectively, and *r* is the rank of the matrix *W*. This equation tells us something quite profound: the complex transformation *W* is actually a *sum* of simpler, rank-one transformations, each scaled by a singular value *σ_i*. The larger the singular value *σ_i*, the more "important" or impactful the corresponding rank-one component *u_i v_i^T* is in the overall transformation. It's like saying a complex melody is built from a sum of simpler musical notes, and the singular values dictate the volume or prominence of each note.

Now comes the clever part: **Singular Value Fine-tuning (SVF).** Transformer²'s innovation lies in realizing that we don't need to adjust everything in the weight matrix W to adapt its behavior. Instead, we can achieve targeted modifications by just tweaking the singular values Σ. They propose to learn a vector, which they call an "expert vector", let's denote it as z. This vector z has the same dimension as the number of singular values. The SVF operation modifies the original singular value matrix Σ into a new matrix Σ′ by performing an element-wise multiplication:

$$\Sigma' = \Sigma \otimes \text{diag}(z)$$

Singular Value Fine-tuning — Image by Author

Here, diag(z) creates a diagonal matrix from the vector z, and ⊗ represents element-wise multiplication. Essentially, each element z_i of the expert vector scales the corresponding singular value σ_i. The new, adapted weight matrix W' then becomes:

$$W' = U\Sigma'V^T = U(\Sigma \otimes \text{diag}(z))V^T$$

SVD with SVF — Image by Author

Notice that U and V remain unchanged from the original SVD of W. Only the diagonal matrix Σ is modified by the expert vector z. This is the core of SVF's parameter efficiency: we are only learning the relatively small vector z, instead of modifying the potentially massive matrices U, Σ, and V directly.

During the two-pass inference, the first pass uses the original weight matrices W. Let's say the input prompt is x. The hidden states from the first

pass can be represented conceptually as *h1 = LLM(x ; W)*.
These hidden states are then used to determine the appropriate "expert vector", let's call it *z'*. The adaptation happens in the second pass. The weight matrices are modified to *W'_adapted' = U(Σ ⊗ diag(z'))V^T* using the chosen expert vector *z'*. Finally, the output *y* is generated using these adapted weights: *y = LLM(x ; W'_adapted)*.
The choice of *z'* depends on the "adaptation strategy" — whether it's prompt-based classification, a dedicated classifier, or a more sophisticated few-shot adaptation method, as we'll discuss later.

To train these expert vectors *z*, the authors uses **Reinforcement Learning (RL)**. This is a departure from the typical next-token prediction objective used in pre-training and standard finetuning. The idea is to directly optimize for task performance. For a given prompt *x_i* and a generated answer *y^i*, a reward *r(y^i,yi)* is defined based on the correctness of the answer compared to the ground truth *y_i*. The objective function *J(θ_z)* they aim to maximize (in a simplified form) looks something like:

$$J(\theta_z) = \mathbb{E}\left[\log \pi_{\theta_{W'}}(\hat{y}_i|x_i) \cdot r(\hat{y}_i, y_i) - \lambda D_{KL}(\pi_{\theta_{W'}} \| \pi_{\theta_W})\right]$$

Objective Function — Image by Author

While this formula might look a bit dense, the key components are:

$$\pi_{\theta_{W'}}(\hat{y}_i|x_i)$$

Objective Function Probability Component — Image by Author

This represents the probability of generating the answer *y^i* given the prompt *x_i* using the adapted model with weights *W'*.

$$r(\hat{y}_i, y_i)$$

Objective Function Reward Component — Image by Author

This is the reward signal, indicating how good the generated answer is.

$$D_{KL}(\pi_{\theta_{W'}} \| \pi_{\theta_W})$$

Objective function Divergence Component — Image by Author

This is a KL divergence term that penalizes the adapted model π_θ_W' for deviating too much from the original model π_θ_W. This acts as a regularizer, preventing drastic changes and maintaining some of the base model's general capabilities.

Lastly, λ is a coefficient that controls the strength of this regularization.

Essentially, the RL objective encourages the expert vectors to modify the model in a way that maximizes the reward for the specific task, while also keeping the changes relatively constrained, thanks to the KL divergence penalty.

Compared to traditional finetuning, which often modifies a large portion of the model's parameters, and even parameter-efficient methods like LoRA, which introduce low-rank matrices, SVF offers a fundamentally different and arguably more efficient approach. By operating directly on the singular values, it targets the core "strengths" of the weight matrices, allowing for nuanced and task-specific adjustments with a minimal parameter footprint. However, it's also worth considering potential limitations. Is modifying only the singular values enough to capture the full spectrum of adaptations needed for all tasks? Could it be too restrictive in some scenarios? These are

questions we need to keep in mind as we continue to explore Transformer^2 and its practical implications. But for now, hopefully, this mathematical unpacking gives you a clearer picture of the elegant machinery driving this self-adaptive LLM framework.

The Code Behind Transformer-Squared

The researchers published along the paper a GitHub repo, which we will dissect in this section of the article, to understand how to translate the math formulas above into actionable code.

GitHub - SakanaAI/self-adaptive-llms: A Self-adaptation Framework🐙 that adapts LLMs for unseen...

A Self-adaptation Framework🐙 that adapts LLMs for unseen tasks in real-time! - SakanaAI/self-adaptive-llms

github.com

Overall Picture

Let's quickly summarize everything we have said until now and get ready for the fun part. Traditional finetuning methods update many of a model's parameters. In contrast, Transformer^2 proposes to "fine-tune" only the "strengths" of the weight matrices by decomposing them via SVD and then adjusting just the singular values. In practice, this means that for a given weight matrix

$$W = U \Sigma V^T,$$

SVD Formula — Image by Author

the framework learns a small "expert vector" z that modulates the singular values (the diagonal elements of Σ) via an element-wise operation. The new

weight matrix becomes something like

$$W' = U\left(\Sigma \odot \mathrm{diag}(z)\right) V^T,$$

SVF with SVD formula — Image by Author

possibly with a scaling factor to preserve overall magnitude. This is the core of "Singular Value Fine-tuning" (SVF): rather than re-optimizing millions of parameters, only a compact vector is learned using reinforcement learning (RL).

The provided scripts implement this idea. They (1) decompose certain layers' weights via SVD, (2) define a "policy" (or set of expert vectors) that produces masks or coefficients to modify the singular values, (3) combine (or even mix) several expert vectors when needed, and (4) use a two-pass evaluation mechanism that may first classify a prompt and then apply the corresponding "expert" modification. Let's now walk through the files.

Main Script: `svd_reinforce_hydra.py`

This script orchestrates the entire adaptation process, training the "policy" that learns the expert vectors and then applying them to the model.

It uses **Hydra** and **OmegaConf** to load experiment settings (number of iterations, batch size, seed, whether to run in test-only mode, etc.).

```
@hydra.main(version_base=None, config_path="cfgs", config_name="config")
def main(cfg):
    ...
```

This structure allows for dynamic configuration management where all parameters — from training iterations to model selections — can be specified in YAML files. The script's configuration handling is particularly sophisticated, as shown in the wandb initialization:

```
def wandb_init(cfg, run_name: str, group_name: str, log_dir: str):
    config_dict = OmegaConf.to_container(
        cfg,
        resolve=True,
        throw_on_missing=False,
    )
    config_dict["log_dir"] = log_dir
    config_dict["wandb_run_name"] = run_name
    config_dict["wandb_group_name"] = group_name
```

This code converts Hydra's OmegaConf configuration into a format suitable for experiment tracking, ensuring all parameters are properly logged and experiments are reproducible.

The model initialization phase is particularly interesting. Look at how the script handles the SVD decomposition:

```
if not os.path.exists(decomposed_param_file):
    print("Decomposed params not found. Decomposing...")
    decomposed_params = {}
    for k, v in base_params.items():
```

```
            if "norm" not in k:
                print(k)
                U, S, V = torch.svd(v)
                decomposed_params[f"{k}.U"] = U
                decomposed_params[f"{k}.S"] = S
                decomposed_params[f"{k}.V"] = V
        torch.save(decomposed_params, decomposed_param_file)
```

This performs SVD on the model's weight matrices, but crucially, it's selective — note the `if "norm" not in k` condition. This means it's only decomposing specific layers, avoiding unnecessary computation on normalization layers where SVD wouldn't be meaningful.

This mirrors the SVD:

$$W = U \Sigma V^T,$$

SVD — Image by Author

where Σ (here called `s`) holds the singular values that "measure the importance" of each direction.

The policy initialization and optimization setup shows the system's flexibility:

```
policy: Policy = hydra.utils.instantiate(
    cfg.shakeoff_policy,
    base_params=base_params,
    decomposed_params=decomposed_params,
    gpu=gpu,
)
```

```
optimization_algorithm: OptimizationAlgorithm = hydra.utils.instantiate(
    cfg.optimization_algorithm,
    policy=policy,
    gpu=gpu,
)
```

It instantiates both the policy network and its optimization algorithm through Hydra's configuration system. The policy is responsible for generating the "expert vectors" that will modulate the singular values, while the optimization algorithm determines how these vectors are updated based on task performance.

The checkpoint loading system is particularly robust:

```
if resuming_from_ckpt and os.path.exists(load_ckpt):
    if use_lora:
        assert os.path.isdir(load_ckpt), "ckpt for lora must be dir to lora adap
        from peft import PeftModel
        lora_model = PeftModel.from_pretrained(model, load_ckpt)
        merged_model = lora_model.merge_and_unload()
        new_params = merged_model.state_dict()
    elif "learnable_params" in load_ckpt:
        learnable_params = torch.load(load_ckpt)
        for k, v in learnable_params.items():
            learnable_params[k] = v.to(gpu)
```

This handles multiple types of checkpoints — standard model states, LoRA adaptations, and learned parameter vectors. The system can seamlessly resume training from any of these formats.

The training loop implements a sophisticated optimization process:

```
for i in range(num_iters):
    batch_ix = np_random.choice(train_ix, size=clipped_batch_size, replace=False

    optimization_algorithm.step_optimization(
        model_id=model_id,
        model=model,
        tokenizer=tokenizer,
        policy=policy,
        task_loader=task_loader,
        batch_ix=batch_ix,
        train_data=train_data,
        train_eval=train_eval,
        base_params=base_params,
        decomposed_params=decomposed_params,
        original_model_params=original_model_params,
        metrics_to_log=metrics_to_log,
        vllm_model=vllm_model,
    )
```

Each iteration samples a random batch of training data and passes it through the optimization algorithm. The `step_optimization` method handles the complex process of:

1. Applying the current policy to modify singular values
2. Running the modified model on the batch
3. Computing the reward signal
4. Updating the policy parameters

The evaluation system is particularly sophisticated:

```
if test_only and prompt_based_eval:
    test_data_dict = eval_model_experts_prompt_based(
        vllm_model,
        test_eval,
        experts_path_dict,
        policy,
        model,
```

```
            base_params,
            decomposed_params,
            task_loader.target_metric_test,
        )
```

This implements the two-phase evaluation system where inputs are first classified and then processed using appropriate expert vectors. The evaluation results are comprehensively logged:

```
data_dict = {
 "iter": i,
 "best_val_acc": best_val_acc,
 "test_at_best_val": test_at_best,
 "train_acc": train_res.aggregate_metrics[task_loader.target_metric_train],
 "valid_acc": valid_res.aggregate_metrics[task_loader.target_metric_valid],
 "test_acc": test_res.aggregate_metrics[task_loader.target_metric_test],
 **metrics_to_log.get(),
}
```

The logging system captures not just basic metrics but also detailed statistics about the policy's behavior and parameter distributions. This comprehensive logging is crucial for understanding the adaptation process and debugging issues.

Policy and Optimization

The Policy class (policy/base.py) inherits from PyTorch's nn Module and implements the core logic for generating and managing the "expert vectors" that modulate the model's singular values. First, let's look at the initialization:

```
def __init__(self, base_params, gpu, init_val, max_mult=1, **kwargs):
    super().__init__()
    self.learnable_params = {}
    self.num_params = 0
    self.max_mult = max_mult
```

This setup establishes crucial parameters including `max_mult` which caps the maximum multiplication factor for singular values, preventing extreme modifications that could destabilize the model.

The core parameter initialization happens in the following loop:

```
for k, v in base_params.items():
    if "mlp" in k:
        self.learnable_params[k] = torch.nn.Parameter(
            data=(
                torch.randn(
                    min(v.shape),
                    device=gpu,
                    dtype=torch.bfloat16,
                ) * 0.01 + init_val
            ),
            requires_grad=True,
        )
        self.num_params += self.learnable_params[k].numel()
```

This code is particularly interesting because it:

1. Only creates parameters for MLP layers (`if "mlp" in k`)
2. Initializes them with small Gaussian noise (multiplied by 0.01) plus an initialization value

3. Uses `min(v.shape)` to create vectors that match the smaller dimension of the weight matrix - this is crucial because SVD decomposition only needs as many singular values as the smaller dimension
4. Uses bfloat16 precision for memory efficiency

The helper function `get_soft_mask` is an implementation of a smooth thresholding mechanism:

```
def get_soft_mask(n, fraction):
    indices = torch.linspace(0, n - 1, n, dtype=torch.bfloat16) + 1
    scaled_indices = indices.to(fraction.device) - fraction * n
    result = torch.clamp(scaled_indices, 0, 1)
    return 1.0 - result
```

This function creates a continuous mask that gradually transitions from 1 to 0, allowing for smooth parameter modulation rather than hard cutoffs. The implementation:

1. Creates evenly spaced indices
2. Scales them based on the fraction parameter
3. Clamps values between 0 and 1
4. Inverts the result to create a mask that starts at 1 and smoothly decreases

The policy's mask generation method is equally important:

```
def get_mask(self, p):
```

```
        return torch.sigmoid(p).to(torch.bfloat16) * self.max_mult
```

This applies a sigmoid function to the learned parameters and scales them by max_mult, ensuring that:

1. The output is always positive (due to sigmoid)
2. The values are properly bounded (due to max_mult)
3. The gradients flow smoothly (due to sigmoid's continuous nature)

The class includes several utility methods for parameter management:

```
def get_learnable_params(self, detach=False):
    return self.learnable_params

def set_trainable_params_values(self, new_values):
    with torch.no_grad():
        for p, v in zip(self.trainable_params, new_values):
            p.data.copy_(v)
```

These methods facilitate parameter access for forward passes, safe parameter updates during optimization, and state management for checkpointing

When this policy is used in the main training loop, it generates masks that modify the singular values according to:

```
new_param = U @ diag(S * mask) @ V^T * scaling_factor
```

Where:

- The mask is generated by the policy's `get_mask` method
- U and V are the original singular vectors
- S contains the singular values
- The scaling_factor ensures the modifications remain bounded

This is the implementation of

$$W' = U\left(\Sigma \odot \text{diag}(z)\right) V^T,$$

SVD with SVF — Image by Author

exactly how Transformer^2 "fine-tunes" the model by only modifying the singular values via a small expert vector zz.

Next, policy/weighted_combination.py extends the base Policy class to enable combining multiple expert vectors. Starting with the initialization:

```
def __init__(
    self,
    base_params,
    decomposed_params,
    base_policy_cfg: Optional[Union[DictConfig, int]],
    params_paths: List[str],
    gpu,
```

```
        norm_coeffs,
        per_layer,
        init_values: Optional[List[float]] = None,
        **kwargs,
    ):
```

This complex initialization handles multiple configuration options, including:

- Multiple parameter paths for different experts
- Normalization options for coefficients
- Per-layer adaptation settings
- Optional initialization values for weights

Next the expert loading process:

```
weights_dict_list: List[Dict[str, torch.Tensor]] = []
if base_policy_cfg is None:
    base_policy = Policy(base_params=base_params, gpu=gpu, init_val=0)
elif isinstance(base_policy_cfg, DictConfig):
    base_policy: Policy = hydra.utils.instantiate(
        base_policy_cfg,
        base_params=base_params,
        decomposed_params=decomposed_params,
        gpu=gpu,
    )
```

This code provides flexibility in base policy initialization, either creating a simple Policy or using Hydra's configuration system for more complex setups. The expert loading continues:

```
with torch.no_grad():
    for i, load_ckpt in enumerate(params_paths):
        print(f"Loading checkpoint {i} at {load_ckpt}...")
        if "learnable_params" in load_ckpt:
            learnable_params = torch.load(load_ckpt)
        else:
            state_dict = torch.load(load_ckpt, weights_only=True)
            base_policy.load_state_dict(state_dict=state_dict)
            learnable_params = base_policy.get_learnable_params()
```

Here, we load multiple expert vectors from different checkpoints, handling both direct parameter files and full model states.

Then, we initialize the adaptive weights:

```
if init_values is None:
    init_values = torch.Tensor(
        [1 / self.num_weights_dict for _ in range(self.num_weights_dict)]
    )
else:
    assert len(init_values) == self.num_weights_dict
    init_values = torch.Tensor(init_values)

if per_layer:
    self.learned_params_per_weight_dict = self.num_params_per_weight_dict
init_values = torch.stack(
    [init_values for _ in range(self.learned_params_per_weight_dict)], dim=1
)
```

This code:

1. Creates uniform initial weights if none provided
2. Supports per-layer adaptation through the `per_layer` flag

3. Stacks weights appropriately for either global or per-layer adaptation

The parameter registration process is thorough:

```
self.parameter_keys = []
self.original_params = {}
for k, v in weights_dict_list[0].items():
    self.parameter_keys.append(k)
    self.original_params[k] = []
    for i, weight_dict in enumerate(weights_dict_list):
        weight_tensor = self.get_mask(p=weight_dict[k])
        new_key = k.replace(".", "_")
        self.register_buffer(
            f"weights_{i}_k_{new_key}",
            tensor=weight_tensor,
        )
```

This creates a structured storage system for expert parameters, using PyTorch's buffer system for efficient memory management.

Next, we implement coefficient generation:

```
def get_coeff_per_layer(self):
    if self.norm:
        adaptive_weights = self.adaptive_weights / self.adaptive_weights.sum(0)
    else:
        adaptive_weights = self.adaptive_weights
    weights_per_layer = adaptive_weights.expand(
        [
            self.num_weights_dict,
            self.num_params_per_weight_dict,
        ]
    )
    return weights_per_layer
```

This method:

1. Optionally normalizes weights to sum to 1
2. Expands weights to cover all layers
3. Maintains efficient tensor operations

The parameter combination happens in `get_learnable_params`:

```
def get_learnable_params(self):
    adaptive_coeff_per_layer = self.get_coeff_per_layer()
    output_params = {}
    for i, (k, vs) in enumerate(self.original_params.items()):
        cs_coeff = adaptive_coeff_per_layer[:, i]
        out = vs[0] * cs_coeff[0]
        for j, other_v in enumerate(vs[1:]):
            v_idx = j + 1
            out = out + other_v * cs_coeff[v_idx]
        output_params[k] = out
```

This implements the weighted combination of expert vectors, efficiently computing the linear combination of parameters based on the learned coefficients.

Finally, the monitoring system:

```
def record_state(self, metrics_to_log):
    avg_weights = self.adaptive_weights.mean(-1).detach().cpu().numpy()
    dict_to_log = {
        f"adaptive_weight/mean_across_params_w{i}": w
        for i, w in enumerate(avg_weights.tolist())
```

```
        }
        metrics_to_log.update(**dict_to_log)
```

This provides crucial visibility into how the model combines different experts, logging the average weight given to each expert across all parameters.

In summary, `policy/base.py` and `weighted_combination.py` define how expert vectors are stored, processed (e.g. via a sigmoid to produce a mask), and—even in the case of multiple experts—combined into a final set of modifications.

Additionally, `utils.py` provides the glue functions that:

- Recompose new weight matrices from the SVD components and the learned modifications,
- Copy these updated parameters into a serving model (vLLM),
- And facilitate evaluation (including a two-step classification to decide which expert to use).

Together, these scripts decompose weight matrices into U, Σ, and V^T, and then fine-tuning only the "volume knobs" (the singular values) via lightweight, learned expert vectors, the system can adapt to new tasks quickly and efficiently — without re-training the entire model. This is the practical realization of the math described in the Transformer^2 paper.

Advantages of Transformer Squared

Let's now zoom out and appreciate the potential broader impact of Transformer^2. What are the key advantages that make this framework so noteworthy, and why should we be excited about this new direction in LLM research?

Perhaps the most compelling benefit is **efficiency**. In a world where scaling AI models often feels synonymous with exponentially increasing computational costs, Transformer^2 offers a refreshing counterpoint. The parameter-efficient nature of Singular Value Fine-tuning is a game changer. By focusing solely on tuning the singular values through those compact "expert vectors," Transformer^2 drastically reduces the number of trainable parameters compared to traditional finetuning or even many Parameter-Efficient Fine-Tuning (PEFT) methods. Think back to our car engine analogy — it's like optimizing fuel consumption by tweaking a few key settings rather than overhauling the entire engine design. This efficiency translates to faster training of these "expert vectors," lower computational demands during adaptation, and potentially reduced storage requirements for specialized models. In resource-constrained environments, or when rapid iteration and experimentation are crucial, this efficiency advantage could be a significant enabler.

Beyond efficiency, Transformer^2 promises a new level of **real-time adaptability**. The two-pass inference mechanism is the engine of this dynamic behavior. By allowing the LLM to analyze the input prompt in the first pass and then adapt its weights on-the-fly in the second pass, Transformer^2 moves beyond the static nature of finetuned models. It's no longer about creating a separate, specialized model for every task. Instead, we have a base model capable of dynamically morphing its behavior to suit the task at hand. Imagine an AI assistant that can seamlessly switch between

coding assistance, math problem-solving, and creative writing, all within the same model instance, adapting its internal "gears" in real-time as you shift between requests. This dynamic adaptability could unlock new possibilities for truly interactive and context-aware AI applications.

Moreover, Transformer^2 hints at greater **versatility and generalization**. The paper's experimental results suggest that this framework isn't confined to a specific LLM architecture. It appears to be applicable across different model families (Llama, Mistral) and even across modalities, showing promising results in vision-language tasks. This cross-architecture and cross-modal potential is intriguing. It suggests that the core principles of Transformer^2 — SVF and dynamic adaptation — might be fundamental and broadly applicable, rather than niche techniques tied to specific model designs. Moreover, the notion of "expert vectors" opens up possibilities for knowledge transfer and reuse. Could we potentially "transplant" expert vectors trained on one model to another, or combine expert vectors from different domains to create even more versatile and capable adaptive models? The paper's preliminary experiments on cross-model transfer are certainly encouraging in this direction.

While the focus is primarily on efficiency and adaptability, there's also a subtle suggestion of improved **compositionality and interpretability**, although these aspects are less fully explored in the initial paper and remain areas for future research. The idea that expert vectors, by modulating singular values, are acting on more fundamental, orthogonal components of the weight matrices could potentially lead to more modular and interpretable adaptations. If we can understand what each expert vector is "tuning" in terms of these singular components, it might offer a more transparent view into how the model is adapting its behavior for different tasks. However, this is still largely speculative. Whether Transformer^2 truly

leads to more interpretable or compositional models requires further investigation and rigorous analysis.

However, Transformer^2, while promising, is still a relatively nascent framework. We need more research to fully understand its strengths and limitations. Some open questions and potential caveats come to mind:

- **Complexity of Adaptation Strategies:** The paper explores a few initial adaptation strategies (prompt-based, classifier-based, few-shot). How do we choose the *best* strategy for a given task or application? And how do we design more sophisticated and robust dispatch mechanisms to accurately identify task properties and select the most appropriate expert vectors?
- **Generalization and Robustness:** While initial results are encouraging, we need to evaluate Transformer^2 on a much wider range of tasks, datasets, and real-world scenarios. How well does it generalize to truly novel, out-of-distribution tasks? And how robust is it to noisy or ambiguous inputs?
- **Scalability to a Large Number of Experts:** As we envision increasingly versatile self-adaptive LLMs, we might need to train and manage a large number of expert vectors, potentially for hundreds or even thousands of specialized domains. How does Transformer^2 scale in such scenarios? Are there efficient ways to organize, select, and combine a vast library of expert vectors?
- **Trade-offs between Adaptation Speed and Performance:** The two-pass inference mechanism introduces a slight overhead compared to standard inference. While the paper suggests this overhead is manageable, we need to carefully analyze the trade-offs between adaptation speed and performance in different applications, especially those with strict latency requirements.

Despite these open questions, Transformer^2 presents a compelling vision for the future of LLMs. As research in this direction progresses, it will be fascinating to see how Transformer^2 and related self-adaptive frameworks reshape the landscape of Large Language Models and their applications.

Conclusion

As we reach the end of our exploration, it's clear that Transformer^2 isn't just another incremental improvement in the ever-evolving world of Large Language Models. It represents a more fundamental shift, a potential **paradigm change** in how we approach LLM adaptation. It dares to imagine a future where we might, just perhaps, begin to "stop finetuning" in the traditional sense, and instead embrace models that are inherently more

Singular Value Fine-tuning (SVF) and the **two-pass inference mechanism**. SVF, with its parameter efficiency and focus on modulating singular values, offers a compelling alternative to the resource-intensive and often static nature of traditional finetuning. The two-pass process, in turn, empowers the LLM to become a more active participant in its own task adaptation, dynamically adjusting its internal workings based on the real-time demands of the input prompt.

The potential advantages are hard to ignore: **enhanced efficiency**, promising reduced computational costs and faster adaptation cycles; **real-time adaptability**, hinting at LLMs that can fluidly transition between tasks without explicit retraining; and **broadened versatility**, suggesting applicability across diverse architectures and modalities. These are not just incremental gains; they point towards a fundamentally different kind of AI —

one that is less like a collection of specialized tools and more like a general-purpose intelligence capable of dynamically tailoring its skills.

Of course, it's crucial to remember that Transformer^2 is still a relatively early exploration in this direction. Many questions remain unanswered, and further research is essential to fully realize its potential and address its limitations. The optimal adaptation strategies, the robustness of generalization, the scalability to a vast ecosystem of expert skills, and the practical trade-offs between adaptation speed and performance — these are all areas ripe for deeper investigation.

However, even at this stage, Transformer^2 offers a valuable and inspiring blueprint. It challenges us to rethink our approach to LLM adaptation, to move beyond the limitations of static finetuning, and to explore the exciting possibilities of self-adaptive AI systems. Whether Transformer^2 itself becomes the dominant paradigm, or serves as a stepping stone towards even more sophisticated self-adapting frameworks, one thing seems increasingly clear: the journey towards truly dynamic, on-the-fly adaptable Large Language Models has only just begun, and it promises to be a fascinating and transformative one.

References

- Qi Sun, et al. "TRANSFORMER-SQUARED: SELF-ADAPTIVE LLMS". [arXiv link: http://arxiv.org/abs/2501.06252v3].

Further Reading Recommendations

The Math Behind Transformers

Deep Dive into the Transformer Architecture, the key element of LLMs. Let's explore its math, and build it from scratch...

medium.com

The Math Behind Multi-Head Attention in Transformers

Deep Dive into Multi-Head Attention, the secret element in Transformers and LLMs. Let's explore its math, and build it...

medium.com

Don't Do RAG: Cache is the future

CAG or RAG? Let's explore Cached Augmented Generation, its math, and trade-offs. Let's dig into its research paper to...

levelup.gitconnected.com

Data Science | Artificial Intelligence | Large Language Models | Deep Learning | Machine Learning

Published in Level Up Coding

201K Followers · Last published 1 hour ago

Follow

Coding tutorials and news. The developer homepage gitconnected.com && skilled.dev && levelup.dev

Written by Cristian Leo

34K Followers · 11 Following

Following

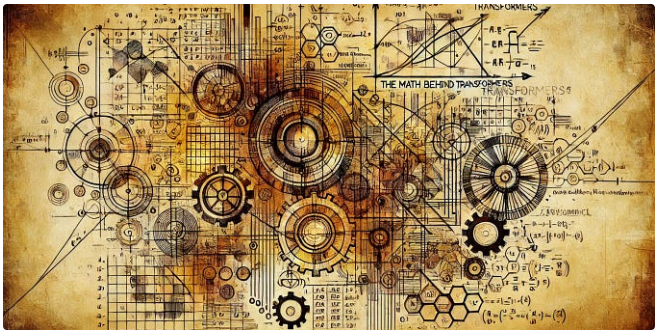
Data Scientist @ Amazon with a passion about recreating all the popular machine learning algorithm from scratch.

No responses yet

What are your thoughts?

Respond

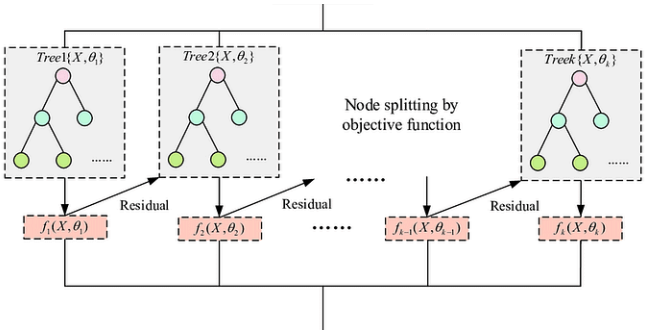
More from Cristian Leo and Level Up Coding



Cristian Leo

The Math Behind Transformers

Deep Dive into the Transformer Architecture, the key element of LLMs. Let's explore its...



Cristian Leo

The Math Behind XGBoost

Building XGBoost from Scratch Using Python

Jul 25, 2024 957 8

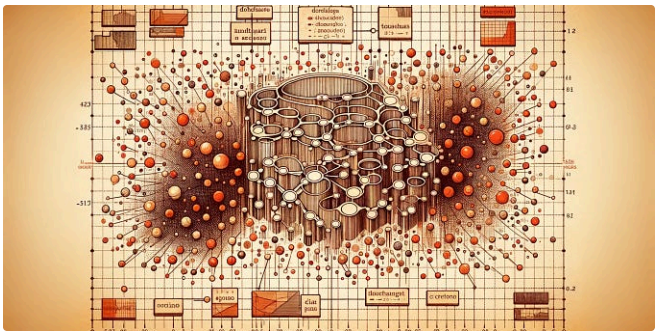
Jan 10, 2024 510 5

In Towards Data Science by Cristian Leo

Reinforcement Learning 101: Building a RL Agent

Decoding the Math behind Reinforcement Learning, introducing the RL Framework, an...

Feb 19, 2024 893 10



In Towards Data Science by Cristian Leo

The Math Behind K-Means Clustering

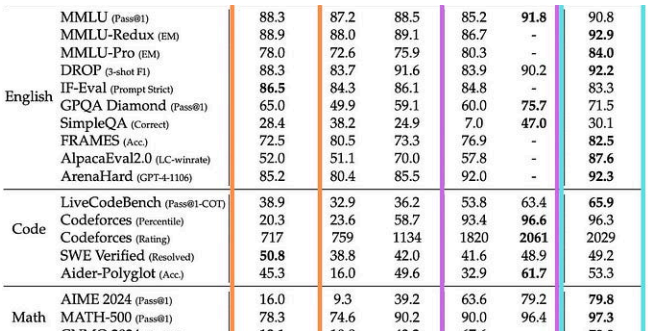
Why is K-Means the most popular algorithm in Unsupervised Learning? Let's dive into its...

Feb 13, 2024 729 3

See all from Cristian Leo

See all from Level Up Coding

Recommended from Medium



English	MMLU (Pass@1)	88.3	87.2	88.5	85.2	**91.8**	90.8
	MMLU-Redux (EM)	88.9	88.0	89.1	86.7	-	**92.9**
	MMLU-Pro (EM)	78.0	72.6	75.9	80.3	-	**84.0**
	DROP (3-shot F1)	88.3	83.7	91.6	83.9	90.2	**92.2**
	IF-Eval (Prompt Strict)	**86.5**	84.3	86.1	84.8	-	83.3
	GPQA Diamond (Pass@1)	65.0	49.9	59.1	60.0	**75.7**	71.5
	SimpleQA (Correct)	28.4	38.2	24.9	7.0	**47.0**	30.1
	FRAMES (Acc.)	72.5	80.5	73.3	76.9	-	**82.5**
	AlpacaEval2.0 (LC-winrate)	52.0	51.1	70.0	57.8	-	**87.6**
	ArenaHard (GPT-4-1106)	85.2	80.4	85.5	92.0	-	**92.3**
Code	LiveCodeBench (Pass@1-COT)	38.9	32.9	36.2	53.8	63.4	**65.9**
	Codeforces (Percentile)	20.3	23.6	58.7	93.4	**96.6**	96.3
	Codeforces (Rating)	717	759	1134	1820	**2061**	2029
	SWE Verified (Resolved)	**50.8**	38.8	42.0	41.6	48.9	49.2
	Aider-Polyglot (Acc.)	45.3	16.0	49.6	32.9	**61.7**	53.3
Math	AIME 2024 (Pass@1)	16.0	9.3	39.2	63.6	79.2	**79.8**
	MATH-500 (Pass@1)	78.3	74.6	90.2	90.0	96.4	**97.3**

In Towards AI by Thuwarakesh Murallie

Resource-Efficient Fine-Tuning of DeepSeek-R1

How to make DeepSeek R1 to reason with your private data

6d ago 20

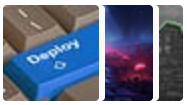
Isaak Kamau

A Simple Guide to DeepSeek R1: Architecture, Training, Local...

DeepSeek's Novel Approach to LLM Reasoning

Jan 23 3.6K 36

Lists

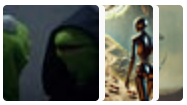


Predictive Modeling w/ Python

20 stories · 1821 saves

Practical Guides to Machine Learning

10 stories · 2191 saves



Natural Language Processing

1928 stories · 1583 saves

AI Regulation

6 stories · 686 saves

Henry Navarro

Ollama vs vLLM: which framework is better for inference? 👊 (Part II)

Exploring vLLM: The Performance-Focused Framework

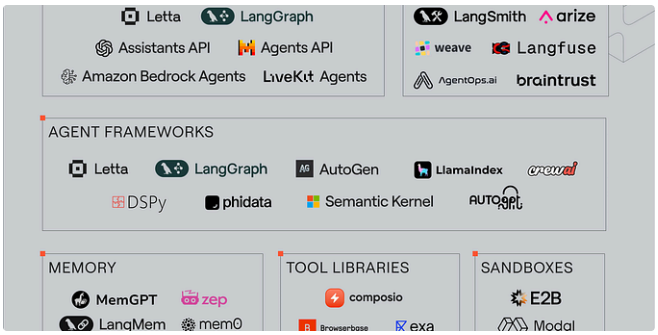
Feb 3 124 1

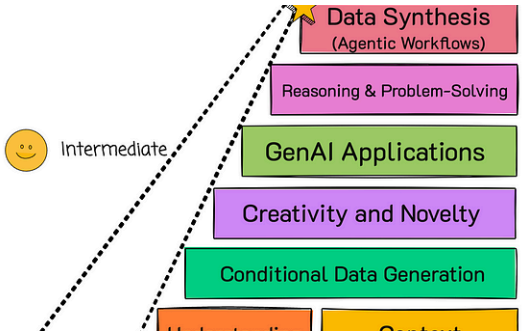
Daniel Avila

Step-by-Step: Running DeepSeek locally in VSCode for a Powerful,...

This step-by-step guide will show you how to install and run DeepSeek locally, configure it...

Feb 2 317 11





Vipra Singh

AI Agents: Introduction (Part-1)

Discover AI agents, their design, and real-world applications.

Feb 2 221 5

Cobus Greyling

Why The Focus Has Shifted from AI Agents to Agentic Workflows

We find ourselves on a stairway from where Large Language Models were introduced to...

5d ago 264 1

See more recommendations