ChatGPT

Identifying and Mitigating Shortcut Learning via Task Arithmetic

Executive Summary

Neural networks often exploit **shortcut features**—spurious correlations that yield high training accuracy but fail under distribution shift [1] [2]. Prior work shows DNNs tend to latch onto easy shortcuts early in training (a "simplicity bias") [3] [4]. Here, we outline a comprehensive *implementation plan* to detect and correct such shortcuts using **task arithmetic** (weight-space edits) and weight-trajectory analysis. We review relevant literature, then propose concrete experimental protocols and code design. Our plan covers data/ model setups (e.g. CLIP-ViT and GPT-2 models on vision/text tasks [5] [6]), training hyperparameters, weight-snapshot logging, task-vector computation (difference of pretrained and finetuned weights [7]), and edit operations (negation, addition, analogies). We will analyze weight trajectories via PCA/CCA, probes, gradient alignment and other metrics to spot shortcut directions. Using task vectors representing unwanted features (e.g. background, gender), we will edit models to "unlearn" those cues. We detail a **reproducible pipeline**: environment setup (Docker/conda), repository structure, key scripts, and CLI commands. Outputs include tables of model accuracies under different edits, suggested figures (weight-space PCA plots, mermaid pipeline flowcharts), and code pseudocode. All claims are cited from recent sources [1] [5] [6] [8] [9].

1. Literature Survey

Shortcut learning and spurious features. Shortcuts are decision rules that "perform well on standard benchmarks but fail to transfer to more challenging testing conditions" [1]. In practice, they are spurious patterns (e.g. background color, texture) correlated with labels in the training data [2]. Geirhos et al. formally define shortcuts as features that work i.i.d. but fail o.o.d. [1]. Contemporary studies (e.g. Murali et al. 2023) highlight that harmful spurious features tend to be learned **early** in training. This aligns with the concept of *simplicity bias*: deep nets first fit the easiest features, even if spurious [3] [4]. For example, in image tasks with a background-label correlation, networks will "anchor" on background cues early [3]. This behavior has been empirically observed in vision and NLP benchmarks with known biases (e.g. Waterbirds, CelebA) where models memorize a shortcut subgroup first [3] [2].

Task arithmetic and model editing. Ilharco et al. introduced *task vectors*: for a model pre-trained at weights w_{pre} and fine-tuned on task T to w_T, the task vector is $v_T = w_T - w_{\text{pre}}$ [7]. By adding or subtracting these vectors, one can emulate multi-task learning or "unlearn" a task. Crucially, **negating** v_T (i.e. applying $w_{\text{pre}} - v_T$) drastically reduces performance on task T with minimal effect on other tasks [10] [6]. Ilharco et al.'s experiments on vision (CLIP models, tasks like Cars, DTD, etc.) and text (GPT-2 and toxicity task) validate this [10] [6]. This editing approach requires no data access at edit time, only the saved weights. Related model-editing work includes ROME/MEMIT for factual knowledge in LMs [11] [12] (which, like task arithmetic, manipulate weights for targeted behavior) and single-weight editing for spurious features [8]. For instance, Hakemi et al. (2021) treat a spurious feature as a "fictitious sub-class" and remove it via a careful single-weight edit [8]. Ding et al. (2025) apply task arithmetic to reduce site-provenance bias in medical models [13]. Dynamic editing methods (e.g. Yang et al. ICLR 2025) locate the layer responsible for spurious behavior and apply rank-1 edits to correct it [9]. These works confirm weight-space editing is a viable post-hoc mitigation strategy.

Training dynamics. Examining how and when networks learn features is key. Murali et al. show that if a spurious feature is "easier" than core features, it will be learned and strengthened over training, detectable in gradient norms and accuracy curves [14]. The Google "early readouts" approach uses probes on intermediate layers: it finds that if an early-layer probe makes confident but incorrect predictions on an instance, that instance likely contains a spurious cue [15]. In practice, we will track per-sample learning curves (Prediction Depth) and layer-wise probe accuracies to flag shortcut reliance. In summary, literature suggests (1) clearly define and identify shortcuts [1] [2], (2) use weight-space edits to remove them [10] [8], and (3) analyze training trajectories to catch shortcuts early [3] [4].

2. Experimental Protocols (Reproducing Key Studies)

Datasets and Tasks. We gather the vision and language tasks used in Ilharco et al. (ICLR 2023) [5] [6], plus relevant shortcut benchmarks. Specifically:
- **Vision tasks:** Stanford Cars [16], Describable Textures (DTD) [17], EuroSAT [18], GTSRB (German Traffic Signs) [19], MNIST [20], RESISC45 (remote sensing scenes) [21], SUN397 (scene recognition) [22], and SVHN (street numbers) [5]. As control we use ImageNet (for CLIP) and Wikitext-103 (for GPT) as Ilharco did [10] [6]. We also plan experiments on **Waterbirds** (birds on land vs water backgrounds) and **CelebA** (hair color vs gender bias) to test generic shortcut mitigation (these are publicly available datasets).

- **Language tasks:** We use GPT-2 Large (1.5B) and create a *toxic generation* task by fine-tuning on comments with high toxicity (e.g. CivilComments >0.8) [23]. The control is measuring perplexity on Wikitext-103 [6].

Models and Checkpoints. We start from these publicly released models:
- **CLIP-ViT**: ViT-B/32, ViT-B/16, ViT-L/14 checkpoints from OpenAI/Meta (available via HuggingFace or OpenAI release). These models have image and text encoders; we use only the image encoder in supervised tasks [24].
- **GPT-2 Large** (1.5B): downloadable from HuggingFace (gpt2-large).

We will cite official sources or repos: e.g., CLIP by Radford et al. (openai/CLIP) and GPT-2 by OpenAI. These checkpoints should be logged as assets.

Training Regime: For each task, we fine-tune the pretrained model as follows (matching Ilharco et al. Appendix):
- **Optimizer:** AdamW (weight decay=0.1).
- **Learning rate:** 1e-5 with cosine annealing; linear warm-up of 200 steps [25].
- **Batch size:** 128 (vision) or appropriate for text (e.g. 32 contexts).
- **Number of steps:** ~2000 gradient steps for vision tasks [25]; for GPT-2, we may fine-tune ~10000 steps (or until loss stabilizes). These numbers can be adjusted.
- **Freezing:** We freeze any new classification layers so that only original model weights change [26]. For CLIP, we fix the text encoder and any added head (as Ilharco did) [26]. For GPT-2, we fine-tune all transformer weights (but can optionally freeze embeddings).

- **Data splits:** Use standard train/val/test splits (if not pre-specified, use 80/10/10). For small tasks (Cars, GTSRB) use random splits; for CelebA/Waterbirds use provided splits.

After fine-tuning, we obtain final weights w_T. Task vectors are $v_T = w_T - w_{\text{pre}}$ [7]. We save intermediate snapshots of weights (e.g. every 100 steps) for trajectory analysis.

Baselines: In all experiments, we compare: (a) **No edit (fine-tuned)** model; (b) **Gradient ascent edit**: fine-tune with *negative* loss (as in Golatkar et al.) to maximize error on T [27]; (c) **Random vector**: add a random noise vector of same layer-wise norm as v_T [28]; (d) **Data-based**: e.g. fine-tune on "non-toxic" data (for GPT) [23]. These check whether the task-vector edit is special.

3. Code Architecture and Repo Layout

We propose a modular code repository (`shortcut_patcher/`) with this structure:

```
shortcut_patcher/
├─ data/                    # Scripts to download and preprocess datasets
│   ├─ vision_tasks.py
│   └─ text_tasks.py
├─ src/
│   ├─ train.py             # Main training loop (finite training on a task)
│   ├─ task_vector.py       # Compute and apply task vectors
│   ├─ edit_model.py        # Utilities for editing (negation, addition)
│   ├─ analyze.py           # Analysis modules (PCA, CCA, probing)
│   ├─ utils.py             # Data loaders, common functions
│   └─ visualize.py         # Plotting (PCA/UMAP, accuracy curves)
├─ env/                     # Environment and container files
│   ├─ Dockerfile           # Docker config with PyTorch, CUDA, etc.
│   └─ environment.yml      # Conda env spec (Python, PyTorch, numpy, sklearn,
etc.)
├─ run_pipeline.sh          # Example bash script to run full pipeline
├─ experiments/
│   ├─ logs/                # Saved logs and checkpoints
│   └─ results/             # Tables and figures output
└─ README.md               # Repo overview
```

Key modules: - **train.py**: Parses CLI (task, model), loads dataset (from `data/`), fine-tunes model, logs weights at intervals, saves final weights.
- **task_vector.py**: Loads pretrained and finetuned weights, computes v_T, supports scaling and layer filtering.
- **edit_model.py**: Applies edits: `apply_edit(model, v_task, alpha)` returns a new model. Implements negation (`alpha=-1`), addition of multiple vectors, layer-wise application.
- **analyze.py**: Contains functions for PCA on weight differences, CCA on layer activations, training linear probes on intermediate features, computing EL2N or Prediction Depth. E.g. `run_pca(weights_matrix)` or `train_probe(activations, labels)`.

- **utils.py**: Data loaders for each dataset, metric functions (accuracy, toxicity% via Detoxify), checkpoint saving/loading.
- **visualize.py**: Functions to plot: 2D PCA of weight trajectories, line plots of accuracy over steps, UMAP of task vectors.

Pseudocode Example (file contents):
train.py:

```
def train_on_task(task_name, model_name, seed, output_dir):
    model = load_pretrained(model_name)
    train_data, val_data, test_data = load_dataset(task_name)
    optimizer = AdamW(model.parameters(), lr=1e-5, weight_decay=0.1)
    scheduler = CosineAnnealingLR(optimizer, T_max=2000, warmup=200)
    for step in range(1, 2001):
        batch = sample_batch(train_data)
        loss = compute_loss(model, batch)
        loss.backward(); optimizer.step(); scheduler.step();
optimizer.zero_grad()
        if step % 100 == 0:
            save_weights(model, f"{output_dir}/ckpt_{step}.pth")
    save_weights(model, f"{output_dir}/final.pth")
```

task_vector.py:

```
def compute_task_vector(pretrained_path, finetuned_path, output_path):
    w_pre = load_weights(pretrained_path)
    w_ft = load_weights(finetuned_path)
    v = {k: w_ft[k] - w_pre[k] for k in w_pre.keys()}
    save_weights(v, output_path)  # store vector diff
```

edit_model.py:

```
def apply_task_edit(model, task_vector, alpha=1.0, layers=None):
    for name, param in model.named_parameters():
        if layers is None or name in layers:
            param.data += alpha * task_vector[name]
    return model
```

analyze.py:

```
def run_pca_on_trajectories(weight_snapshots):
    # weight_snapshots: list of flattened weight arrays
    pca = PCA(n_components=2).fit(weight_snapshots)
    return pca
```

```
def train_linear_probe(features, labels):
    clf = LogisticRegression().fit(features, labels)
    return clf.score(features, labels)
```

Environment: We will provide a `Dockerfile` or `environment.yml` with packages: PyTorch (GPU), torchvision, transformers, scikit-learn, umap-learn, matplotlib, seaborn. Example: `FROM pytorch/pytorch:2.0-cuda11.7` in Dockerfile, then `pip install transformers scikit-learn ...`. For reproducibility, version-lock key libs (PyTorch 2.x, Python 3.10+).

CLI Usage Examples:
- Finetune vision task:

```
python train.py --task Cars --model CLIP-ViT-L-14 --seed 42 --output
experiments/visual_cars/
```

- Compute & apply negative edit:

```
python task_vector.py --pre trained_models/clip_vitl14.pth \
                      --finetuned experiments/visual_cars/final.pth \
                      --output vectors/cars_vector.pth
python edit_model.py --model clip_vitl14 --task_vector vectors/cars_vector.pth \
                      --alpha -1.0 --out edited_models/
clip_vitl14_forget_cars.pth
```

- Run analysis:

```
python analyze.py --trajectory experiments/visual_cars/weights_snapshots/ \
                  --method pca --output figures/weight_pca.png
python analyze.py --activations experiments/visual_cars/activations.pkl \
                  --probe linear --labels experiments/visual_cars/labels.npy
```

4. Weight-Trajectory Analysis Methods

We log model weights at intervals (e.g. every 100 steps) during fine-tuning, producing a weight trajectory. We focus on analyzing these to identify *shortcut directions*:
- **Principal Component Analysis (PCA):** Flatten each saved model (or selected layers) into a vector; stack them in time order. Apply PCA to the matrix of snapshots. Visualize the top 2 PCs with time coloring. If a strong shortcut is learned early, the trajectory will quickly move along a principal axis [10]. We can also PCA across *tasks* (stack v_T vectors for different tasks) to see cluster of similar tasks (e.g. EuroSAT and RESISC45 are both satellite tasks [29]).
- **Canonical Correlation (CCA):** Run CCA on hidden activations of the model between different feature sets (e.g. spurious vs core examples, or before vs after editing). High canonical correlations indicate shared

subspaces. For example, compare activations on original vs edited models to see what subspace changed.
- **Linear Probes:** Train simple linear classifiers on intermediate representations to predict the shortcut label (e.g. background vs object) or core label. A high probe accuracy on early features means the network has encoded that distinction. For NLP, probe hidden states for presence of bias attributes.
- **Representer Scores:** Use the representer theorem (Koh & Liang 2017) to attribute model decisions to training points. In a classifier, compute representer-point scores to identify which training examples influence a given test decision. If a spurious shortcut is used, the highest-scoring points will often share that shortcut attribute.
- **Gradient Alignment:** At each training step, compute the cosine similarity between the weight update (or gradient) and the final task vector v_T. A positive alignment early on indicates the model is moving in the direction of learning that task (potentially via its shortcuts). We can plot alignment vs steps to see when a shortcut vector starts dominating.
- **Early Learning Metrics:** Measure *Prediction Depth* or *EL2N* score for each training example [30]. Prediction Depth records the first epoch where an example is learned; examples with spurious cues may have lower depths. EL2N (loss * gradient norm) can identify “easy” (low EL2N) vs hard (high EL2N) samples. A bimodal pattern may signal that “easy” (spurious) ones drive early learning.
- **Statistical Tests:** For each analysis metric (e.g. probe accuracy, gradient alignment), run multiple seeds and test significance. We will use paired t-tests or bootstrap confidence intervals to confirm differences (e.g. alignment vs random baseline).

These methods, combined with visualization, will pinpoint **shortcut directions** in parameter space. We will include figures such as *2D PCA plots* of trajectories, *UMAP clustering* of v_T, and *learning curves* over epochs.

5. Task-Vector Computation and Model Editing

Computing Task Vectors: After fine-tuning, load the pretrained model weights (w_{pre}) and finetuned weights (w_T). For each layer tensor, compute $v_T = w_T - w_{\text{pre}}$ and save this vector structure (same keys) [25]. We provide pseudocode in `task_vector.py` (see above). This can be done either layer by layer or by flattening. We may compress vectors (e.g. store as sparse or float16) if memory is a concern, but full precision is ideal.

Applying Edits: Editing means adding a scaled task vector back to the pretrained model: $w_{\text{new}} = w_{\text{pre}} + \alpha v_T$. In code, we load the pretrained model, then for each parameter `param`, do `param += alpha * v_T[param_name]`. A negative α (e.g. $\alpha=-1$) *subtracts* the task knowledge (unlearning) [10]. We implement:
- **Negation:** $\alpha=-1$ for target task vectors.
- **Scaling:** α in [0,1] to partially edit (choose α so that control accuracy $\ge95\%$ as Ilharco did [31]).
- **Addition/Combination:** For multitask, apply multiple vectors: e.g. $w_{\text{new}} = w_{\text{pre}} + v_{T1} + v_{T2}$ to get a model that knows tasks $T1,T2$ [32]. For analogy “A is to B as C is to D”, do $v_A - v_B + v_C$ to generate a model for D [33].

We provide a function `apply_edit(model, task_vectors, alphas)` to add several vectors at once. We also support **layer-wise edits**: sometimes only editing the last MLP layers or attention layers is sufficient. For example, the **Rome** method targets specific feed-forward modules [11]. We can optionally pass a layer list to only apply edits there.

Safety and Rollback: Because edits change weights in-place, we keep a copy of the original weights. In pseudocode, we always operate on a copy of the pretrained state. All applied edits should be logged (which vector, alpha, layers) so we can reproduce. We also save the edited model checkpoint for evaluation.

6. Visualization & Flowcharts

We will include clear visual aids. Suggested figures:
- **Weight PCA/UMAP:** A scatter plot of model weights (projected to 2D) at different training steps, colored by step or by known shortcut strength. This shows the trajectory shape.
- **Accuracy/Loss Curves:** Plot target vs control accuracy across steps or vs edit scale α. This highlights the trade-off.
- **Mermaid Pipeline Flowchart:** For example:

```
flowchart LR
  PretrainedModel[Pretrained Model (CLIP, GPT2)] --> FineTune[Fine-tune on Task T]
  FineTune --> ComputeV[Compute Task Vector v_T]
  ComputeV -->|Alpha=-1| ForgetEdit[Edited Model (forget T)]
  ComputeV -->|Alpha=+1| AddEdit[Edited Model (add T)]
  FineTune --> Baselines[Baseline Models (gradient-ascent, random, non-toxic fine-tune)]
  ForgetEdit --> Evaluate[Evaluate on Task T and Control]
  AddEdit --> Evaluate
  Baselines --> Evaluate
```

This flowchart clarifies the edit operations and evaluation pipeline.

- **Timeline (Mermaid):** A high-level timeline of project milestones could be shown, e.g.:

```
timeline
  title Project Milestones
  2022 : Task Arithmetic (Ilharco et al. ICLR) introduced [7]
  2023 : ROME/MEMIT factual editing (Meng et al.) [11] [12]
  2024 : Spurious feature dynamics (Murali et al.) [14] and simplicity bias (Tiwari & Shenoy) [3]
  2025 : Single-weight spurious edits (Hakemi et al.) [8] ; Task-vector bias mitigation (Ding et al.) [13]
  2026 : Our pipeline implementation and ablations
```

All visualizations will be saved in `experiments/figures/` and referenced in deliverable documents.

7. Mitigation Interventions and Evaluation

Task-Vector Subtraction: Using the computed vectors for known shortcuts (e.g. a model trained to predict background class), we subtract them from the main model. For instance, if v_{bg} is from a background-classifier, we do $w_{\text{new}} = w - v_{\text{bg}}$. We will measure how target accuracy and control accuracy change. A successful mitigation is a drop in accuracy on spurious-containing examples without hurting core examples.

Multiple Vectors: If multiple spurious cues exist, combine their vectors. For example, subtracting both "background" and "texture" vectors. Conversely, if we have a beneficial task (e.g. de-noising), we could add its vector. We will experiment with adding or subtracting up to 3 vectors simultaneously.

Adversarial Training / Reweighting: As an alternative intervention, we will train a model with an adversarial loss or reweight examples to discourage the shortcut. For example, use GroupDRO or a similar method on Waterbirds (treat background groups as robust groups). We will compare the effectiveness and cost of this to task-vector editing.

Evaluation Metrics: We evaluate in multiple ways:
- **In-distribution accuracy** on the original task. We report accuracy on target and control sets for each edit (like Ilharco's Tables).
- **Out-of-distribution accuracy:** E.g. for Waterbirds, test on images where background is flipped or for CelebA, test on balanced hair color.
- **Worst-Group (robust) accuracy:** If data can be partitioned by spurious attribute, report accuracy on the hardest subgroup.
- **Robustness Gap:** The difference between overall accuracy and worst-group accuracy (or OOD accuracy). Smaller gap means less shortcut reliance.
- **Perplexity (for NLP):** For GPT-2 edits, report perplexity on Wikitext-103 (control) and the proportion of toxic outputs (with Detoxify [6]).
- **Accuracy Trade-offs:** We may plot control vs target accuracy for various edit scales.

Statistical significance will be assessed by running at least 3 random seeds per setting and reporting mean±std. Significant improvements ($p<0.05$) will be noted.

8. Experimental Pipeline and Compute Estimates

Step-by-Step Pipeline: 1. **Environment setup:** Build Docker image / conda env (`docker build -t shortcut .`). 2. **Data prep:** Run `python data/vision_tasks.py` and `python data/text_tasks.py` to download datasets. 3. **Training:** For each task-model pair, run `train.py` (e.g. as in CLI above). 4. **Snapshot Logging:** We save weights every 100 steps to `experiments/logs/[task]/snapshots/`. 5. **Compute Vectors:** For each task, run `task_vector.py`. 6. **Apply Edits:** Use `edit_model.py` for negation/addition (CLI examples above). 7. **Evaluation:** Evaluate edited models on test data. 8. **Analysis:** Run `analyze.py` for PCA, probes, etc. 9. **Record Results:** Save accuracies and metrics to CSV for tables.

Compute Resources: - Fine-tuning CLIP-ViT (ViT-L/14, ~307M params) for 2000 steps: ~0.5 GPU-hours on an A100 (quick per task). We have 8 vision tasks × 3 model sizes = 24 finetunes. Total ~12 GPU-hours.

- GPT-2 Large finetuning: moderate (1.5B params, 10k steps) ~2 GPU-hours.
- Analysis (PCA, probes) is CPU-light.
- Overall, estimate ~20 GPU-hours. On cloud pricing, ~ $2-$5 per GPU-hour, so under ~$100 total.

Hyperparameter Grid & Ablations: We will vary: learning rate (5e-6,1e-5,5e-5), steps (1000,2000,5000), and AdamW vs Adam; edit scale α (0,0.5,1.0,1.5); which layers to edit (all vs last 2 FFN layers). This yields ~3×3×3×2=54 runs per (task, model) combination. With seeds, the total experiments ~hundreds, which is parallelizable.

Expected Outcomes:
- **Reproduce Ilharco:** Expect to match Ilharco's results [10]: e.g. negating a task vector drops target task accuracy by ~40% with minimal control loss. Fine-tuning via gradient ascent should collapse performance [34].
- **Shortcut detection:** Weight trajectory PCA should reveal that tasks sharing a shortcut cluster together (e.g. EuroSAT/RESISC45) [29]. Early portions of the trajectory will align with the principal components of spurious features. Probes should show early-layer encodings of spurious attributes.
- **Mitigation:** We hope that subtracting a spurious task vector (e.g. a background classifier) improves robust metrics (higher worst-group accuracy) more efficiently than heavy retraining.
- **Trade-offs:** Larger edits ($|\alpha|>1$) will cause some drop in overall accuracy, which will be quantified.

9. Deliverables (Tables, Figures, Code Snippets)

- **Table 1:** *Vision Task Forgetting*. Columns: Method, ViT-B/32 Target Acc (%), Control Acc (%); similarly for B/16, L/14. Rows: "Pretrained", "Fine-tuned on T", "Gradient Ascent", "Random Vec", "Negative Task Vec" [10]. (As per Ilharco's Table 1.)
- **Table 2:** *GPT-2 Toxicity*. Methods (Pretrained, Finetuned, Gradient Ascent, Non-toxic FT, Random Vec, Negative Vec) vs (% toxic generations, Avg toxicity, Perplexity) [6].
- **Table 3:** *Multi-task Addition Results*. (Based on Ilharco's Fig.2/Table.) Subset size vs normalized accuracy, comparing single vs multi-task.

- **Table 4:** *Intervention Comparison*. Rows: "Edit (task vec)", "Adversarial Training", "Reweighting", "Baseline". Columns: In-dist Accuracy, OOD Accuracy, Worst-group, Compute Cost.

- **Figure 1:** Weight trajectory PCA (mermaid code included above).

- **Figure 2:** Task-vector UMAP: show clustering of v_T for vision tasks.
- **Figure 3:** Example accuracy curves (target vs control) for different edits.

- **Figure 4:** CLI usage or pipeline flowchart (as mermaid above).

- **Code Snippets:** In the answer we provide key pseudocode blocks (as above) for `train_on_task()`, `compute_task_vector()`, `apply_edit()`, `run_pca()`. We will also include the file-tree snippet and Dockerfile/conda spec example.

- **Mermaid Visualizations:** The pipeline flowchart is given above. Additional templates (e.g. timeline code) can be provided in `visualize.py` docs.

10. References and Resources

Where available, we link official sources: Ilharco's code on GitHub (for task vectors), CLIP and GPT-2 repos, dataset links. All statements above are supported by citations: e.g., definition of shortcuts [1], task-arithmetic facts [10] [6], ROME/MEMIT [11] [12], spurious learning bias [3] [14], single-weight editing [8], and so on.

When details are unspecified (e.g. exact dataset split, model layer names for editing), we note this and choose reasonable defaults (e.g. ImageNet 50k-val as control, "features" layers for CLIP edits).

Overall, this plan lays out a fully detailed, reproducible pipeline (code layout, pseudocode, commands, and environment) to **identify shortcut directions via weight trajectories** and to **mitigate them by task-vector editing**, achieving the report's objectives. Each step is backed by recent literature [1] [5] [6] [8] and the experiments are designed to be fully replicable.

---

[1] arxiv.org
https://arxiv.org/pdf/2004.07780

[2] [14] [30] Beyond Distribution Shift: Spurious Features Through the Lens of Training Dynamics - PMC
https://pmc.ncbi.nlm.nih.gov/articles/PMC11029547/

[3] [15] Intervening on early readouts for mitigating spurious features and simplicity bi
https://research.google/blog/intervening-on-early-readouts-for-mitigating-spurious-features-and-simplicity-bias/

[4] [2302.09344] Beyond Distribution Shift: Spurious Features Through the Lens of Training Dynamics
https://arxiv.org/abs/2302.09344

[5] [6] [7] [10] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [31] [32] [33] [34] [2212.04089] Editing Models with Task Arithmetic
https://ar5iv.labs.arxiv.org/html/2212.04089

[8] Single-weight Model Editing for Post-hoc Spurious Correlation Neutralization
https://arxiv.org/html/2501.14182v2

[9] Dynamic Model Editing to Rectify Unreliable Behavior in Neural Networks | OpenReview
https://openreview.net/forum?id=1dkL3MVBfV

[11] [2202.05262] Locating and Editing Factual Associations in GPT
https://arxiv.org/abs/2202.05262

[12] Model Editing Harms General Abilities of Large Language Models: Regularization to the Rescue
https://arxiv.org/html/2401.04700v3

[13] Tailoring task arithmetic to address bias in models trained on multi-institutional datasets - ScienceDirect
https://www.sciencedirect.com/science/article/abs/pii/S1532046425000875