Kolmogorov Arnold Networks (KANs)

Supervised Learning for Deployment

Shamanth Kuthpadi Seethakantha
sk@iomics.us

June 6, 2025

Abstract

This report explores Kolmogorov–Arnold Networks (KANs) in the context of supervised learning, with a particular focus on understanding their learning dynamics. To effectively deploy KANs in practical applications, we require insight into the training process—specifically, how internal parameters evolve and how hyperparameters influence learning behavior. To this end, I instrument the `pykan` package to log and analyze changes in internal model parameters throughout training. These insights will hopefully guide hyperparameter tuning and provide some abstraction of the learning process. Additionally, a controlled perturbation ("shock") is applied midway through training to probe network resilience and parameter sensitivity. The technical implementation of this instrumentation and its implications for supervised learning are the core focus of this report.

GitHub Repository Link: `https://github.com/shamanth-kuthpadi/KANs-IOMICS`

1 Instrumentation for Training Dynamics Logging

To gain deeper insight into the internal learning dynamics of Kolmogorov–Arnold Networks (KANs), I extended the `fit()` method within the `MultKAN.py` script of the `pykan` library to include fine-grained logging capabilities. This instrumentation enables the monitoring of both performance metrics (e.g., train/test loss, regularization loss) and internal parameter statistics during training.

Specifically, I added a CSV-based logger, controlled via the `logger='csv'` flag, which records the evolution of internal/learned parameters at each training step. These include:

- **Training/Test Loss**: Computed as root mean squared error (RMSE).
- **Regularization Term**: Capturing the contribution of various regularization components, scaled by the `lamb` hyperparameter.
- **Mean Activation Scales** across:
  - Subnodes (`subnode_actscale`)
  - Edges (`edge_actscale`)
  - Activation functions (`acts_scale`)
  - Spline components (`acts_scale_spline`)
- **Mean Coefficient Magnitude**: Averaged across layers that contain trainable coefficients (`layer.coef`).

This logging allows us to understand how various hyperparameters influence the learning trajectory and convergence. By exporting these metrics to a structured `.csv` file, I make the training process more transparent for post-hoc analysis or visualization.

This instrumentation also makes it easier to diagnose convergence behavior, detect overfitting, and tune regularization strengths (e.g., `lamb_l1`, `lamb_entropy`, `lamb_coef`, `lamb_coefdiff`) in a principled way.

1.1 Logged Variables

At each training step, the following variables are tracked and written to the log file:

- `step`: The current training step index.
- `train_loss`: Root mean squared error (RMSE) on the training batch.
- `test_loss`: RMSE on the test batch.
- `reg`: The regularization term

- `subnode_act_mean`: The mean of activation scales across all subnodes. Computed by averaging the mean of each tensor in `self.subnode_actscale`:

$$\texttt{subnode\_act\_mean} = \frac{1}{N}\sum_{i=1}^{N}\texttt{mean}(\texttt{subnode\_actscale}[i])$$

  where N is the number of subnodes.

- `edge_act_mean`: The mean of activation scales across all edges, computed similarly from `self.edge_actscale`.

- `act_scale_mean`: The mean scale across all learned activation functions, computed from `self.acts_scale`.

- `act_scale_spline_mean`: The mean scale of the spline-based components of the activation functions, computed from `self.acts_scale_spline`.

- `coef_mean`: The mean absolute value of the trainable coefficients across all layers that include a `coef` parameter. If M such layers exist, and each layer's coefficient tensor is denoted `layer.coef`, this is computed as:

$$\texttt{coef\_mean} = \frac{1}{M}\sum_{j=1}^{M}\texttt{mean}(|\texttt{layer}_j\texttt{.coef}|)$$

  If no such layers exist, `coef_mean` is recorded as zero.

All mean calculations are detached from the computation graph to avoid gradient tracking.

1.2 Identification of Learnable Parameters

In PyTorch, only tensors wrapped as `torch.nn.Parameter` are recognized as learnable parameters. To identify which parameters to log during training, I inspected all attributes wrapped as `nn.Parameter` in the model. This ensures that our instrumentation captures the relevant internal parameters that are being optimized. Note that in my logging, some learned parameters are not being tracked. This was a choice made in retrospect when I picked variables that had theoretical significance and those that had a clear way to aggregate values. However, this should be and will be changed to include ALL internal learned parameters – the GitHub will be updated for this. Architecturally, the Kolmogorov-Arnold Network (KAN) is realized as a `MultKAN` model, composed of multiple `KANLayer` instances. Each `KANLayer` contains learned parameters within its activation functions (as defined in `KANLayer.py`), while `MultKAN.py` defines the overall architecture and its associated learnable parameters (Figure 1).

Learned Parameters in `KANLayer.py` Each `KANLayer` contains activation functions parameterized by learnable tensors, typically wrapped as `torch.nn.Parameter` objects. These parameters include:

- `coef`
- `grid`
- `mask`
- `scale_base`
- `scale_sp`

These can all be traced from the `KANLayer.__init__` definition.

Learned Parameters in `MultKAN.py` The `MultKAN` class defines the overall architecture, which composes multiple `KANLayer` instances.

- `node_bias`
- `node_scale`
- `subnode_bias`
- `subnode_scale`

These can all be traced from the `MultKAN.__init__` definition.

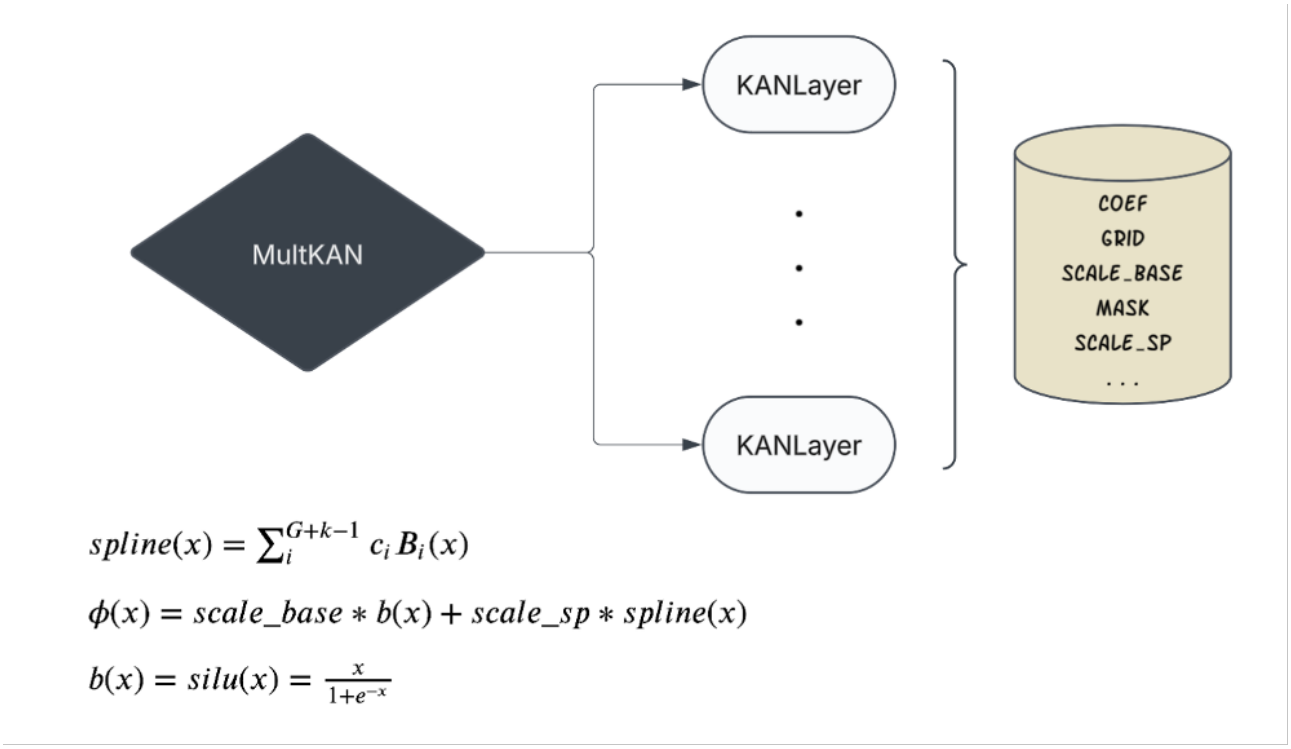

Figure 1: Representation of KANs architecture

2 Coefficient Perturbation via Shock

To better understand the robustness and dynamics of the learning process in KANs, I introduce a perturbation mechanism referred to as a *shock*. The idea is to intentionally disturb the internal state of the network during training and observe how the model adapts.

In our experiments, the shock is applied midway through the training process. For example, in a 50-epoch training run, the shock occurs at epoch 25.

The shock consists of perturbing all learned coefficients across all layers with additive Gaussian noise. For each coefficient tensor, we apply:

$$\theta \leftarrow \theta + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2)$$

where θ represents the original learned coefficient, and ϵ is Gaussian noise sampled independently for each element. In the current implementation, a standard deviation of $\sigma = 1.5$ is used to ensure a non-trivial but perturbation. "Non-trivial" in this case is subjective and the σ could be automated in the future.

This method provides insight into:

- The network's resilience to internal disruptions.
- How well the model can re-adapt or re-converge following a significant parameter disturbance.

Using shock to perturb the model mid-training, together with detailed logging of internal parameters, gives us valuable insight into how the model learns and adapts.

By watching how the network recovers from the shock, we can see which parameters are most sensitive and how different hyperparameters affect stability and performance. This helps us understand which settings make the model more robust and reliable.

Overall, this approach guides us in tuning hyperparameters more effectively, leading to better training outcomes and improved generalization.

The perturbation is injected in-place using PyTorch's `torch.no_grad()` context to avoid interfering with the gradient tracking mechanism.