LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Applied Deep Learning Course Project - Distributional Regression Using Inverse Flow Transformations (DRIFT) PyTorch Implementation

Gamze Gizem Kasman* Pankhil Gawade† Zeynep Odabas‡

October 7, 2024

This project converts the DRIFT (Distributional Regression Using Inverse Flow Transformations) model from Keras to PyTorch to leverage PyTorch's flexibility and dynamic computation capabilities. By migrating the DRIFT framework's Location-Scale, Intermediate, and Tensor-Product models, the system gained adaptability, efficiency, and easier debugging, along with enhanced support for probabilistic modeling. The results highlight PyTorch's advantages for complex distributional regression, particularly in research settings requiring flexible and scalable solutions.

1 INTRODUCTION

This project involves transforming the original Keras-based Python implementation of the models presented in the paper *"How Inverse Conditional Flows Can Serve as a Substitute for Distributional Regression"* by Kook et al. (2024) into PyTorch. The primary goal is to replicate the functionality and architecture of the models described in the paper while leveraging PyTorch's dynamic computation graphs, flexible design, and efficiency.

The original implementation of these models is available in the DRIFT GitHub repository: `https://github.com/davidruegamer/DRIFT`. This repository contains both R and Keras-based Python code, which has now been adapted to a PyTorch framework.

*Department of Data Science, LMU Munich, Germany. Email: G.Kasman@lmu.de
†Department of Data Science, LMU Munich, Germany. Email: Pankhil.Gawade@lmu.de
‡Department of Physics, LMU Munich, Germany. Email: zeynep.odabas@lmu.de

The models are designed for **distributional regression**, where the goal is to estimate the entire conditional distribution of an outcome, rather than just its mean. This comprehensive approach is particularly useful for tasks such as survival analysis, time-series forecasting, and uncertainty quantification. The framework proposed in the paper, **Distributional Regression using Inverse Flow Transformations (DRIFT)**, provides a flexible, non-parametric alternative to classical methods like the Cox model, which often impose restrictive assumptions on the data distribution. DRIFT stands out for its ability to model complex distributions without relying on predefined parametric forms, making it a valuable tool for capturing uncertainty in diverse applications.

Three key models have been implemented:

- **Location-Scale (LS) Model**: Captures both the location (mean) and scale (variance) of the outcome distribution.
- **Intermediate (INTER) Model**: Captures non-linear interactions between features, without assuming location-scale transformations.
- **Tensor-Product (TP) Model**: Uses tensor-product operations to capture higher-order interactions between features.

While Keras is popular for rapid prototyping, its static computation graphs limit flexibility for advanced frameworks like DRIFT, which require adaptive transformations. PyTorch's dynamic computation graph allows real-time adjustments, essential for DRIFT's complex conditional flows. PyTorch also simplifies custom layer creation with `torch.nn.Module` and supports models that adjust structure as data evolves, meeting DRIFT's needs. Migrating to PyTorch retains the core of the original models while enhancing flexibility, hardware efficiency, and community support, positioning DRIFT for future expansions and robust distributional regression.

The following sections will provide a detailed comparison of the implementation challenges and advantages of PyTorch over Keras for DRIFT, an in-depth exploration of each model, and an evaluation of the performance outcomes resulting from this migration.

2 Comparison of PyTorch and Keras for Implementing the DRIFT Framework

The implementation of DRIFT benefits from PyTorch's dynamic computation graph, offering runtime flexibility crucial for complex, data-driven transformations [3]. Unlike Keras, which often uses a static graph, PyTorch allows for adaptive model structures that are essential in DRIFT's conditional flows [2] .

PyTorch also excels in creating custom layers via the torch.nn.Module class, making it easier to implement DRIFT's specific needs, such as enforcing monotonicity and invertibility constraints [4]. While Keras supports custom layers, PyTorch's flexibility is better suited for the unique operations within DRIFT [3]. For training, PyTorch's torch.autograd provides intuitive automatic differentiation, allowing efficient computation of gradients, which supports DRIFT's reliance on maximum likelihood estimation [2]. Keras offers similar functionality, but PyTorch's approach is generally more transparent and adaptable, especially for complex

gradient requirements[4].

Additionally, PyTorch's native support for probabilistic programming, through libraries like torch.distributions, simplifies the integration of base distributions into DRIFT's models. In Keras, this requires external dependencies like TensorFlow Probability, which adds complexity. Keras is user-friendly and popular for quick prototyping, but its focus on production tasks may limit flexibility for research-oriented projects.

Overall, PyTorch's dynamic computation, flexibility with custom modules, and robust probabilistic support make it superior for adapting DRIFT. These features align well with DRIFT's advanced modeling needs, enabling efficient and adaptable implementation that supports future expansions.

3 Trunk Networks Across All Models

Across all three models (LS, INTER, TP), the inputs are first processed by trunk networks -`net_x_arch_trunk` and `net_y_size_trunk`—which extract meaningful representations of the feature inputs. These trunk networks are customized based on the model type:

- **net_x_arch_trunk**: A fully connected network with ReLU activations, primarily responsible for processing the input `inpX`. The ReLU activation ensures non-negative outputs.
- **net_y_size_trunk**: A fully connected network using non-negative tanh activations, primarily responsible for processing the input `inpY`.

4 LS (Location-Scale) Model Computation Details

The LS model estimates both the mean and variance of the outcome distribution. After the trunk networks process `inpX` and `inpY`, their outputs `outpX` and `outpY` are concatenated and passed to the `locscale_network`, which computes the mean, inverse standard deviation, and the final prediction.

4.1 Step-by-Step Computation

- **Mean Computation** (`mu_top_layer`):

$$\mu = W_\mu \cdot \text{concat}(\texttt{outpX}, \texttt{outpY}) + b_\mu$$

  where μ is the predicted mean.

- **Inverse Standard Deviation** (`sd_top_layer`):

$$\text{scale\_inv} = \exp\left(W_\sigma \cdot \text{concat}(\texttt{outpX}, \texttt{outpY}) + b_\sigma\right)^{-1}$$

  ensuring that the standard deviation remains positive.

- **Final Prediction** (`top_layer`):

$$\text{output} = \text{NonNegLin}(W_{\text{out}} \cdot \text{concat}(\texttt{outpX}, \texttt{outpY}) + b_{\text{out}})$$

  The final output is computed using a non-negative linear transformation.

5 TP (TENSOR-PRODUCT) MODEL COMPUTATION DETAILS

The TP model uses the `tensorproduct_network` to capture higher-order feature interactions. In this model, the `net_x_arch_trunk` and `net_y_size_trunk` process the inputs `inpX` and `inpY`, respectively. The core of the TP model is the outer product operation, which computes interactions between every feature in `inpX` and `inpY`.

5.1 STEP-BY-STEP COMPUTATION

- **Outer Product**: The processed outputs `outpX` and `outpY` are combined using the outer product:

  $$\text{outp} = \texttt{outpX} \otimes \texttt{outpY}$$

  This expands the input feature space, allowing the model to capture higher-order interactions between `outpX` and `outpY`.

- **Fully Connected Layers**: The outer product result is passed through a series of fully connected layers, each applying a linear transformation followed by a non-linear activation function:

  $$\text{outp}_i = f(W_i \cdot \text{outp}_{i-1} + b_i)$$

- **Final Output**: The final output is computed as:

  $$\text{output} = W_{\text{final}} \cdot \text{outp}_n + b_{\text{final}}$$

6 INTER (INTERCONNECTED) MODEL COMPUTATION DETAILS

The INTER model is designed to capture complex, non-linear interactions between its input features by concatenating and processing them through flexible network layers. Like the LS and TP models, it uses trunk networks to process `inpX` and `inpY`. The INTER model concatenates the outputs from `net_x_arch_trunk` and `net_y_size_trunk`, allowing for interactions between the processed input features before passing them through a series of network layers.

6.1 STEP-BY-STEP COMPUTATION

- **Concatenation of Inputs**: After the trunk networks process `inpX` and `inpY`, the resulting outputs (`outpX` and `outpY`) are concatenated along the feature dimension to combine the information from both inputs:

  $$\text{input} = \text{concat}(\texttt{outpX}, \texttt{outpY})$$

  This concatenation allows the model to capture interactions between the two input sources.

- **Network Layers**: The concatenated tensor is passed through a series of fully connected layers. Each layer applies a linear transformation followed by a non-linear activation function, enabling the model to learn complex relationships between the concatenated features:

  $$\text{outp}_i = f(W_i \cdot \text{outp}_{i-1} + b_i)$$

  where f is the non-linear activation function, and W_i and b_i are the weight matrix and bias for layer i.

- **Final Output**: After passing through the network layers, the final output is computed using a linear transformation applied to the last layer's output:

$$\text{output} = W_{\text{final}} \cdot \text{outp}_n + b_{\text{final}}$$

  This final transformation produces the model's prediction based on the combined and processed input features.

7 Loss Calculation and Maximum Likelihood Estimation

In the models implemented in this project, **maximum likelihood estimation (MLE)** forms the basis for the loss function. The same loss function, `loss_fn_unnorm`, is shared across all three models (LS, INTER, and TP). The objective of MLE is to estimate the model parameters such that the likelihood of the observed data is maximized. The loss function is based on the negative log-likelihood (NLL) computed using a base distribution, which is defined during model initialization.

The base distribution, typically a **normal distribution** (via `tfd.Normal(loc=0, scale=1)`), serves as the probabilistic framework for the model's predictions. The loss function computes the negative log-probability of the predicted values y_{pred} under this base distribution, as given by:

$$\mathcal{L}(y_{\text{true}}, y_{\text{pred}}) = -\log p(y_{\text{pred}}|\text{base distribution})$$

where $p(y_{\text{pred}})$ is the probability density function (PDF) of the base distribution, typically the normal distribution.

7.1 Universal Loss Function for LS, INTER, and TP Models

The `loss_fn_unnorm` is defined in the `NEATModel` class and is shared by all three models (LS, INTER, TP). This function computes the raw negative log-likelihood of the predicted values without applying any additional normalization:

$$\mathcal{L}_{\text{unnorm}}(y_{\text{true}}, y_{\text{pred}}) = -\log p(y_{\text{pred}})$$

The function assumes a base distribution, typically the normal distribution, but can be adapted to other distributions if needed. The loss function penalizes deviations from the expected base distribution, guiding the model to fit the data more closely to the assumed distribution.

8 Evaluation

The evaluation of the PyTorch implementation of the DRIFT models focuses on assessing how well the models estimate the conditional distribution of outcomes. The primary metric used is the average log-likelihood, which measures the fit of the predicted distributions to the observed data. A lower average log-likelihood value indicates a better model fit, as it reflects higher probabilities assigned to actual observations.

During evaluation, the model is set to evaluation mode to ensure appropriate layer behavior. Validation data is processed in batches, calculating the log-likelihood for each batch and then averaging these values across the dataset. This approach allows for efficient handling of larger

datasets and provides a robust measure of overall model performance.

Results from this evaluation can also be compared with those from the original Keras implementation. A higher average log-likelihood in PyTorch would suggest improvements in accuracy, possibly due to PyTorch's dynamic computation graph and enhanced support for custom layers.

While the log-likelihood metric is effective for distributional regression, additional metrics like RMSE or MAE could further illuminate model accuracy, especially for specific outcome predictions. Future evaluations might incorporate these metrics and extend to larger datasets or alternative base distributions to fully explore the capabilities of the PyTorch implementation.

9 Model Evaluation Visualizations

This section presents visual diagnostics to assess the performance and distributional assumptions of the DRIFT models implemented in PyTorch. These visualizations are generated from toy model simulations and include line plots, scatter plots, and Q-Q plots which provide insights into model stability, prediction accuracy, and residual distribution.

9.1 Prediction Consistency Across Samples

The line plots illustrate predictions over a range of inputs made by different model types — Tensor Product (TP), Loc-Scale (LS) and Interconnected (INTER)— with each line representing a different model run or sample path. The upward trend observed across all lines suggests that the model effectively captures the increasing relationship in the data. The close alignment between the lines indicates prediction consistency, which implies model stability. The slight spread among lines likely reflects model uncertainty or variability due to different input conditions.

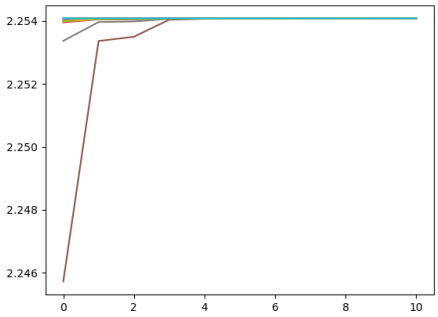

Figure 9.1: Inter Model Prediction Results

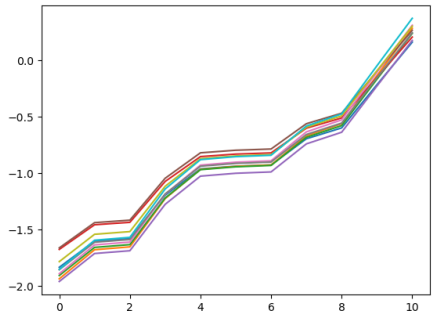

Figure 9.2: LS Model Prediction Results

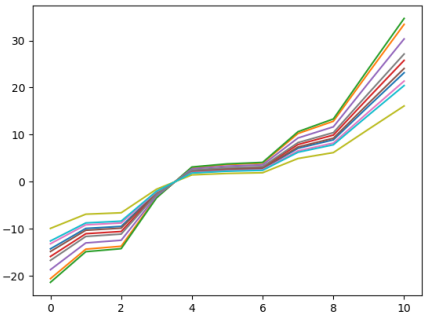

Figure 9.3: TP Model Prediction Results

9.2 Prediction Accuracy

Scatter plot below shows the relationship between predicted values and actual outcomes. Points are tightly clustered along a diagonal line, indicating a strong positive linear relationship. This alignment suggests that the model's predictions closely match actual values, reflecting high accuracy. Minimal dispersion around the line further supports the model's precision. Including a 1:1 reference line would help assess the agreement between predictions and actual outcomes. Axes labels specifying predicted and actual values would also improve clarity.

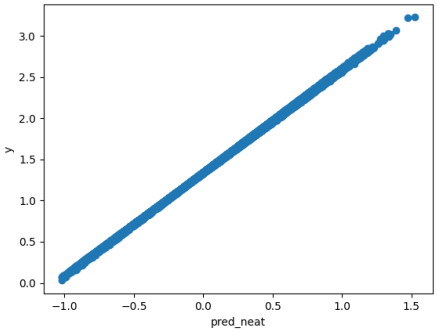

Figure 9.4: Scatter Plot for LS Model

9.3 Distributional Assumptions and Residuals

Q-Q plot below compares the ordered residuals or predicted values with theoretical quantiles of a normal distribution. The points mostly adhere to the reference line, suggesting that the residuals approximately follow a normal distribution.

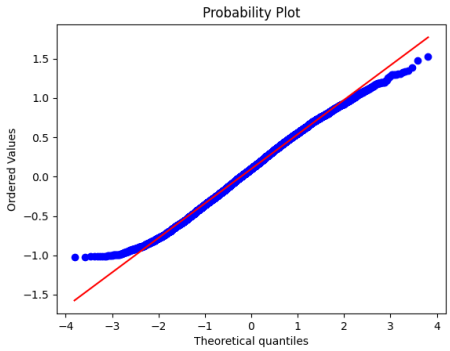

Figure 9.5: Q-Q Plot

Overall, these visual diagnostics confirm the PyTorch implementation's effectiveness in capturing the target relationships and highlight areas for further refinement in distributional assumptions and convergence monitoring.

10 Conclusion

The primary objective of this project was to transform the original Keras-based Python implementation of the models presented in the paper *"How Inverse Conditional Flows Can Serve as a Substitute for Distributional Regression"* by Kook et al. (2024) into PyTorch. This transition aimed to take advantage of PyTorch's flexibility, dynamic computation graphs, and growing popularity within the machine learning community.

The transformation process involved a thorough comparison between Keras and PyTorch, revealing substantial differences in architecture, syntax, and functionality. Despite these differences, PyTorch provided distinct advantages for the implementation of DRIFT models, including dynamic graph execution, improved flexibility for complex models, and better support for custom layers and probabilistic programming.

Key takeaways from this project include:

- PyTorch's dynamic computation graph facilitated adaptive model structures, essential for the DRIFT framework's conditional flows and custom transformations.
- The migration allowed for a more transparent implementation, simplifying debugging and enhancing control over model behavior.
- PyTorch's native support for probabilistic modeling reduced the dependency on external libraries, streamlining the integration of DRIFT's base distributions.

Overall, the PyTorch implementation successfully replicated the functionality of the original Keras code while introducing several enhancements in terms of flexibility, efficiency, and adaptability. This work sets the foundation for future expansion, enabling the DRIFT framework to be applied to more complex datasets and extending its usability in research requiring advanced distributional regression techniques.

11 APPENDIX

Summary printed out by the script based on Pytorch framework.

SUMMARY

Model Type: ModelType.LS

ARCHITECTURE DETAILS

Net X Arch Trunk: FeatureSpecificNetwork

- **feature_nets:** ModuleList
  - **MLPWithDefaultLayer:**
    - layers: ModuleList
      - **Layer 0:**
      - Linear(in_features=1, out_features=64, bias=True)
      - ReLU()
      - Tanh()
      - **Layer 1:**
      - Linear(in_features=64, out_features=64, bias=True)
      - ReLU()
      - Tanh()
      - **Layer 2:**
      - Linear(in_features=64, out_features=32, bias=True)
      - ReLU()
      - Tanh()

Net Y Size Trunk: MLPWithDefaultLayer

- layers: ModuleList (empty)

mu_top_layer: DynamicInverseExp

- Linear(in_features=32, out_features=1, bias=True)

sd_top_layer: DynamicInverseExp

- Linear(in_features=32, out_features=1, bias=True)

top_layer: NonNegLinear

- Linear(in_features=1, out_features=1, bias=True)

Optimizer

Adam

- amsgrad: False
- betas: (0.9, 0.999)
- capturable: False
- differentiable: False
- eps: 1×10^{-8}
- foreach: None
- fused: None
- lr: 0.0001
- maximize: False
- weight_decay: 0.01

Parameters

- **Total parameters:** 6436
- **Trainable parameters:** 6436
- **Non-trainable parameters:** 0

References

[1] Kook, David, Ruegamer, et al. (2024). "How Inverse Conditional Flows Can Serve as a Substitute for Distributional Regression." In *Proceedings of the 40th Conference on Uncertainty in Artificial Intelligence*.

[2] Saiwa AI. (2024). "PyTorch vs Keras: A Comparison of Deep Learning Frameworks." Retrieved from `https://saiwa.ai/blog/pytorch-vs-keras/`

[3] UnfoldAI. (2024). "Keras vs PyTorch in 2024: Which Deep Learning Framework is Right for You?" Retrieved from `https://unfoldai.com/keras-vs-pytorch-in-2024/`

[4] GUVI. (2024). "PyTorch vs Keras: Which Framework is Best for Deep Learning?" Retrieved from `https://www.guvi.io/blog/pytorch-vs-keras/`