LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

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

October 2, 2024

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.

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 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)**, offers a flexible, non-parametric alternative to classical methods like the Cox model.

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.

By migrating the models from Keras to PyTorch, we aim to maintain the core structure of the original models while taking advantage of PyTorch's dynamic computation graph, efficient hardware utilization, and broader community support.

2 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`.

3 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.

3.1 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.

4 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`.

4.1 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:

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

5 INTER (Interconnected) Model Computation Details

The INTER model is more flexible in how it processes its inputs. Like the LS and TP models, it uses the trunk networks to process `inpX` and `inpY`, but it does not strictly concatenate them. Instead, the outputs from `net_x_arch_trunk` and `net_y_size_trunk` are used in more flexible ways to capture complex non-linear interactions between features.

5.1 Step-by-Step Computation

- **Concatenation of Inputs**: The input vectors `inpX` and `inpY` are optionally concatenated depending on the configuration:

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

  if a combined representation is needed.

- **Network Layers**: The concatenated input (if applicable) is passed through several fully connected layers, each layer applying a linear transformation and 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 using a linear transformation on the last layer:

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

6 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:

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

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

6.1 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:

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

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.

References

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