Special Problems (8903) Final Report (Fall'23)

Yunlin Zeng

Project 1: Robust Statistical Inference with Studentising Flow

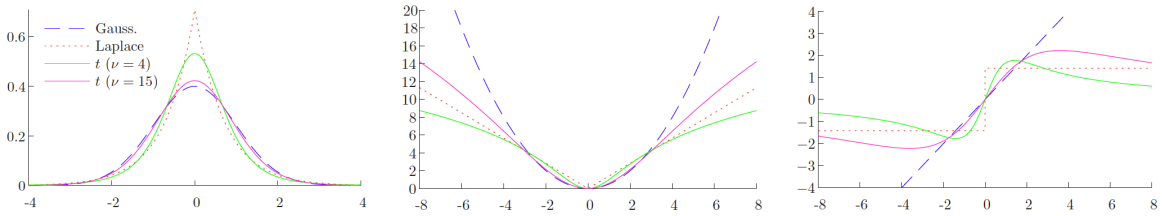
Background and method

Commonly, maximum likelihood estimation (MLE) is employed for parameter estimation, where parameters are optimized to maximize the likelihood of observing the data. This approach, however, has a drawback: the fitting will tends to include outliers. This inclusion could lead to large gradients and destabilize optimization. It's possible to avoid such exploding gradients via gradient clipping. However, this alters the true gradients computed from data and the training process could converge to a different point different from the true optimum, where the average of unaltered gradients is zero. Rather than developing a newer architecture, Alexanderson2020 shifts focus to a new base distribution. They advocate base distributions characterized by fatter tails, which have the following two advantages (1) It exerts an effect similar to gradient clipping, thereby stabilizing the training process. (2) It improves generalization capabilities.

Without changing the fitting principle, Alexanderson2020 suggest changing the parametric family by adopting distributions with fatter tails for MLE. Such distributions assign high probability to outlier occurrences, hence better fitting the remainder of the data. In Figure 1 (b), the penalty function $\rho(x - \mu)$ shows that the negative log-likelihood loss increases quadratically as the data points deviate from the mean. In contrast, distributions with fatter tails, like student t-distribution and Laplace distribution, have a milder rate of increase in their penalty functions. The influence on the penalty function remains bounded (Figure 1 (c)), mirroring the effect of gradient clipping.

Experiment and results

I have done one experiment to observe the empirical advantages of studentising flow by training the Glow network on a reduced-size CelebA dataset at 64×64 resolution. The network was configured with a flow depth of $K = 32$, $L = 3$ levels, a maximum of 1000 iterations, and a learning rate $\text{lr} = 10^{-4}$ with cosine decay. No gradient clipping was implemented in either case. Figure 2 displays the loss curves for the network trained using normalizing flow and



(a) Probability density functions $p(x)$. (b) Penalty functions $\rho(\varepsilon)$. (c) Influence functions $\psi(\varepsilon)$.

Figure 1: Penalty and influence functions of different distributions

studentising flow (with a degree of freedom $\nu = 50$). It is evident that training with normalizing flow is unstable as it experienced a lot of exploding gradients, while studentising flow mitigated those unstable steps and results in a smooth loss curve. Despite closely following the training configuration parameters from the paper, the loss network did not effectively learn the CelebA data.

It's notable to mention that training with neural spline flow (NSF) might diminish the benefit of using student t-distribution. The resulting loss curves are similar to each other (Figure 3). This phenomenon needs to be investigated further.

Future work

Robust statical methods like studentising flow are promising in significantly improve training stability and efficiency. The experiments I have done so far showed advantages of studentising flow but still need careful examination. My work in the near future will include (1) ensuring correct training of Glow with the CelebA data and examining factors that diminish the effects of studentising flow when the model is switched from Glow to NSF.

Project 2: Simulation-based Inference with Augmented Data

Background and method

Broadly speaking, there are two types of inference methods: Approximate Bayesian Computation (ABC) and density estimation methods. ABC iteratively updates model parameters by comparing simulation data with observed data but scales poorly in high dimensions, making it best suited for inferring summary statistics. On the other hand, the density estimation method is amortized; once trained, it can directly evaluate the probability density of new data points Cranmer2020. As sampling is expensive and sample efficiency is immensely important, a novel idea is to train surrogates via augmented data.

Brehmer2020 presented a suite of density estimation methods augmented with data. They claim that data consisting of simulated observations x_i, the joint likelihood ratio $r(x_i, z_i|\theta_0, \theta_i)$,

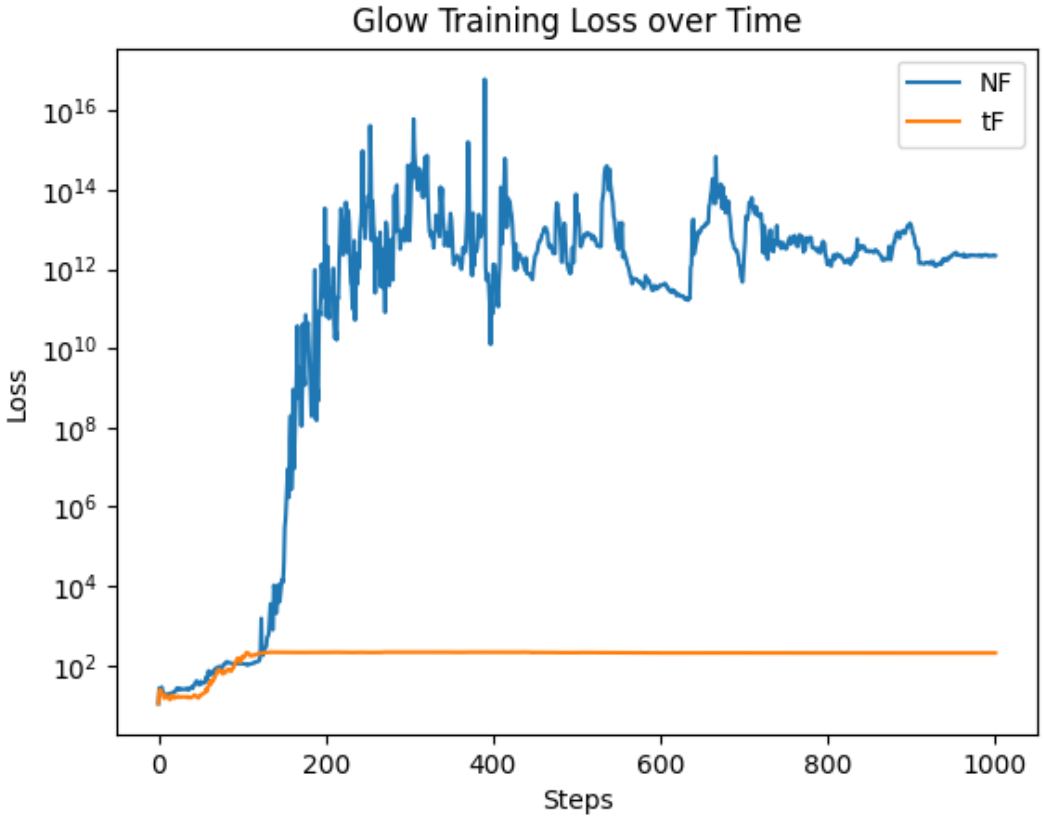


Figure 2: Training loss curves for the Glow network

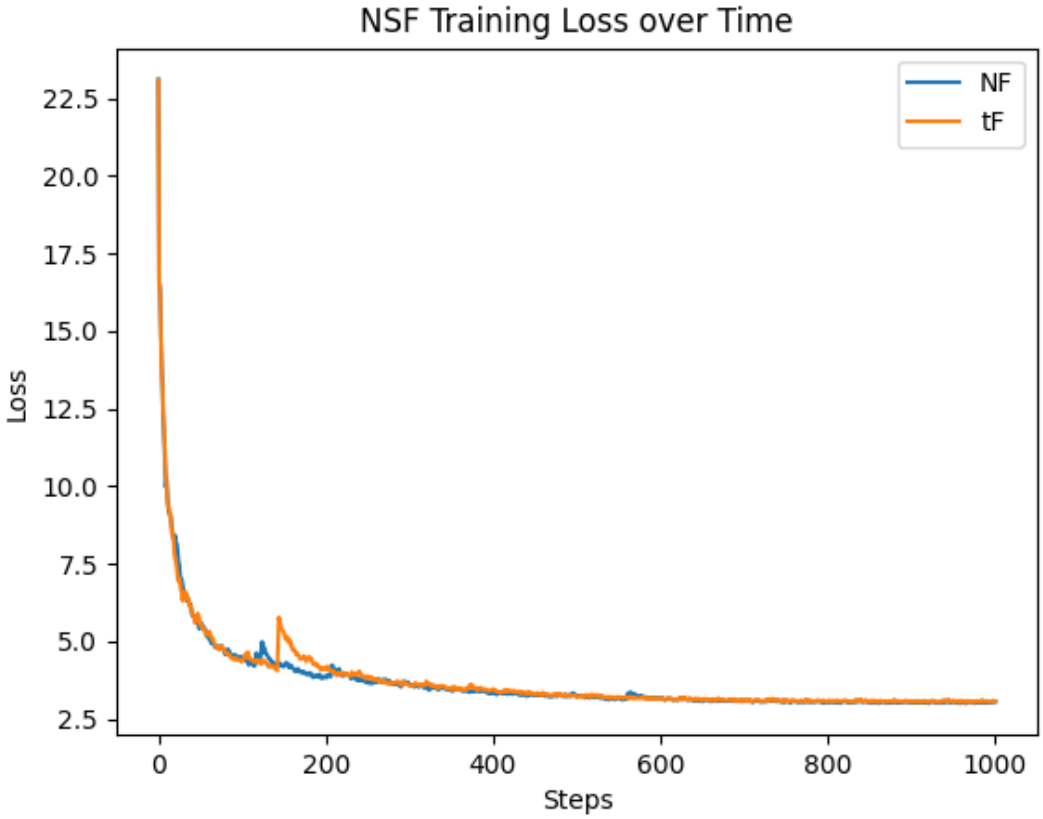


Figure 3: Training loss curves of the NSF network

and the joint score $t(x_i, z_i|\theta_0)$, help more efficiently learn surrogates for the likelihood function $p(x|\theta)$ (or likelihood ratio $r(x|\theta_0, \theta_1)$), and can dramatically improve the sample efficiency for training neural network surrogates.

I aim to apply this score matching method to conduct robust inference in my future work. Below is an outline of the Score-Augmented Neural Density Estimator (SCANDAL) method:

$$L_{\text{SCANDAL}}[\hat{p}] = \frac{1}{N}\sum_i \left(\underbrace{-\log \hat{p}(x_i|\theta_i)}_{\substack{\text{similar to MLE} \\ \text{How well the model matches the data}}} + \underbrace{\alpha \left| t(x_i, z_i|\theta_i) - \nabla_{\theta_i} \log \hat{p}(x_i|\theta_i) \right|^2}_{\text{How the distribution changes when } \theta \text{ varies}} \right) \tag{1}$$

- Notation
  - $\hat{p}(x_i|\theta)$ is a neural density estimator, where the input is an individual simulated observation x_i and the output is its likelihood $\hat{p}(x_i|\theta)$.
  - $t(x_i, z_i|\theta)$ assumes you have an analytical form of $p(x_i, z_i|\theta)$ and can take its gradient.
  - Assume $\hat{p}(x_i|\theta)$ is differentiable with respect to θ.

Experiment and results

In a simple experiment, I implemented the score matching technique to train a simple neural network in Julia. The simulator is a simple nonlinear function, `arctan`. It is straightforward to implement the score matching term by adding `(d_NN - d_simulator)^2` to the loss function and computing the gradient of the network and the simulator. Figure 4 shows the result. We can see from the figure that the addition of the score matching term dramatically improves the fitting. The code snippet is attached to the Appendix.

Future Work

I aim to use the score matching method to train a saturation surrogate model, given permeability. The key is to implement the score matching term (Equation 2) to the loss function when training the saturation surrogate model.

$$\nabla_{k=k_0} \log p(c, z|k) = \min_w l_2 \left[J^\top(k_0)(c - S(k_0, z)) \quad \text{and} \quad J_w^\top(k_0)(c - S_w(k_0, z)) \right] \tag{2}$$

In this equation, $J^\top(k_0)(c - S(k_0, z))$ represents the gradient of the numerical saturation simulation with respect to permeability at k_0. $c - S(k_0, z)$ is the residual between the observed saturation data and the numerical simulation result, and $J^\top(k_0)$ is the Jacobian transpose,

which gives the gradient of the loss with respect to the permeability when multiplied by the residual. Similarly, $J_w^\top(k_0)(c - S_w(k_0, z))$ represents the gradient of the surrogate network with respect to permeability at k_0. $c - S_w(k_0, z)$ is the residual for the surrogate model.

The objective is to minimize the L2 norm (squared Euclidean distance) between two quantities: the gradient of the numerical simulation and the gradient of the surrogate model with respect to permeability k at k_0. This minimization aligns the surrogate model with the physical model by adjusting the network parameters w. This score matching term $\nabla_{k=k_0} \log p(c, z|k)$ helps capture the true distribution and dynamics of the data, thereby enhancing the accuracy and robustness of the surrogate model.

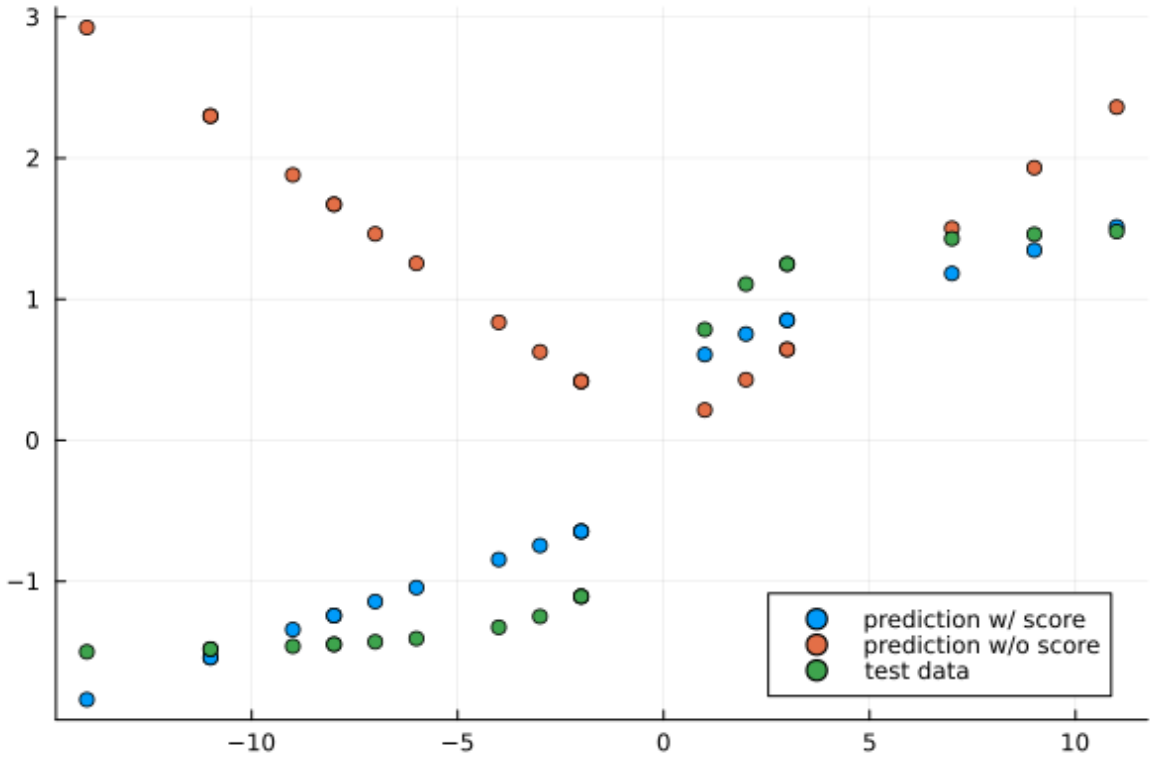


Figure 4: Comparative training results with and without augmented data

Appendix

```
using Flux, Zygote, Statistics
using ForwardDiff
using Plots
using Random
Random.seed!(1)

function d_model(model, x)
    return ForwardDiff.derivative(xi -> model([xi])[1], x)
end

function loss(x, y)
    predicted = model(x)
    mse_loss = Flux.mse(predicted, y)

    derivative_loss = 0
    lambda = 0.1
    for xi in eachcol(x)
        d_NN = d_model(model, xi[1])
        d_simulator = ForwardDiff.derivative(atan, xi[1])
        derivative_loss += (d_NN - d_simulator)^2
    end

    return mse_loss + lambda * derivative_loss
end

f(x) = atan(x)

x_train = rand(-15:15, 2000)
x_test = rand(-15:15, 20)
y_train, y_test = f.(x_train), f.(x_test)
x_train = reshape(x_train, 1, :)
y_train = reshape(y_train, 1, :)
x_test = reshape(x_test, 1, :)

data = [(x_train, y_train)]

hidden_units = 32
model = Chain(
    Dense(1, hidden_units, relu),
```

```
    Dense(hidden_units, hidden_units, relu),
    Dense(hidden_units, hidden_units, relu),
    Dense(hidden_units, 1)
)

opt = Descent(0.001)

maxiter = 2000
for i in 1:maxiter
    Flux.train!(loss, Flux.params(model), data, opt)
end

scatter(x_test[1, :], model(x_test)[1, :], label="prediction w/ score")
scatter!(x_test[1, :], model0(x_test)[1, :], label="prediction")
scatter!(x_test[1, :], f.(x_test[1, :]), label="test data")
```