Universität Zürich[UZH]

Performance of Univariate Kernel Density Estimation methods in TensorFlow

Bachelor Thesis

Author: Marc Steiner

Supervisors: Jonas Eschle, Nicola Serra

University of Zurich

Abstract

Multiple implementations of a one-dimensional Kernel Density Estimation in TensorFlow/Zfit are proposed. Starting from the basic algorithm, several optimizations from recent papers are introduced and combined to ameliorate the efficiency of the algorithm. By comparing its accuracy and efficiency to implementations in pure Python it is shown as competitive and useful in real world applications.

Contents

1 Introduction

1.1 Kernel Density Estimation

In many fields of science and in physics especially, scientists need to estimate the probability density function (PDF) from which a set of data is drawn without having a approximate model of the underlying mechanisms, since they are too complex to be fully understood analytically. So called parametric methods fail, because the knowledge of the system is to poor to design a model, which parameters

can then be fitted by some goodness-of-fit criterion like log-likelihood or χ^2. In the particle accelerator at CERN for instance, they record a whooping 25 gigabytes of data per second[1], resulting from the process of many physical interactions that occur almost simultaneously, making it impossible to anticipate features of the distribution one observes.

To combat this so called non-parametric methods like histograms are used. By summing the the data up in discrete bins we can approximate the underlying parent distribution, without needing any knowledge of the physical interactions. However there are also many more sophisticated non-parametric methods, one in particular is Kernel Density Estimation (KDE), which can be looked at as a sort of generalized histogram.[2]

Histograms tend to produce PDFs that are highly dependent on bin width and bin positioning, meaning the interpretion of the data changes by a lot by two arbitrary parameters. KDE circumvents this problem by replacing each data point with a so called kernel that specifies how much it influences its neighbouring regions as well. The kernels themselves are centered at the data point directly, eliminating the need for arbitrary bin positioning[3]. Since KDE still depends on kernel width (instead of bin width), one might argue that this is not a major improvement. However, upon closer inspection, one finds that the underlying PDF does depend less strongly on the kernel width than histograms on bin width and it is much easier to specify rules for an approximately optimal kernel width than it is to do so for bin width[4]. In addition, by specifying a smooth kernel, one gets a smooth distribution as well, which is often desirable or even expected from theory.

Due to this increased robustness, KDE is particular useful in High-Energy Physics (HEP) where it has been used for confidence level calculations for the Higgs Searches at the Large Electron Positron Collider (LEP)[5]. However there is still room for improvement and certain more sophisticated approaches to Kernel Density Estimation have been proposed in dependence on specific areas of application[5].

1.2 zfit and TensorFlow

Currently the basic principle of KDE has been implemented in various programming languages and statistical modeling tools. The standard framework used in HEP, that includes KDE is the ROOT/RooFit toolkit written in C++. However, Python plays an increasingly large role in the natural sciences due to support by corporations involved in Big Data and its superior accessibility. To elevate research in HEP, zfit, a new alternative to RooFit, was proposed. It is implemented on top of TensorFlow, one of the leading Python frameworks to handle large data and high parallelization, allowing a transparent usage of CPUs and GPUs[6].

So far there exists no direct implementation of Kernel Density Estimation in zfit nor TensorFlow, but various implementations in Python. This implementations will be discussed in the next chapter (2) before I propose an implementation of Kernel Density Estimation in TensorFlow for the univariate case (3). Starting with a rather simple implementation, multiple improvements from recent papers

are integrated to ameliorate its efficiency. Efficiency and accuracy of the implementation are then tested by comparing it to other implementations in pure Python(4).

2 Current state of the art

There are several options available for computing univariate kernel density estimates in Python.

The most popular ones are SciPy's `gaussian_kde`[7], Statsmodels' `KDEUnivariate`[8] as well as `KernelDensity` from the Scikit-learn package[9].

As Jake VanderPlas was able to show in his comparison[10] Scikit-learn's tree based approach to compute the kernel density estimation was the most efficient in the vast majority of cases in 2013. The question of the optimal KDE implementation for any situation, however, is not entirely straightforward, and depends a lot on what your particular goals are.

Statsmodels includes a computation based on Fast Fourier Transform (FFT) and normal reference rules for choosing the optimal bandwidth, which Scikit-learns package lacks for instance.

In 2018 Tommy Odland proposed an entirely new implementation called KDEpy[11] which incorporates features of all implementations mentioned before as well as additional kernels and an additional method to calculate the bandwidth using the Improved Sheather Jones (ISJ) algorithm first proposed by Botev et al[12].

He was also able to show that his FFT based computation was able to outperform previous implementations (even Scikit-learn's tree based approach) in terms of runtime by a factor of at least one order of magnitude, making KDEpy the de-facto standard of Kernel Density Estimation in Python.

Table 1: Comparison between KDE implementations by Tommy Odland[13] (NR: normal reference rules, namely Scott/Silverman, CV: Cross Validation, ISJ: Improved Sheater Jones according to Botev et al.)

Feature / Library	scipy	sklearn	statsmodels	KDEpy
Number of kernels	1	6	7 (6 slow)	9
Weighted data points	No	No	Non-FFT	Yes
Automatic bandwidth	NR	None	NR,CV	NR, ISJ
Multidimensional	No	No	Yes	Yes
Supported algorithms	Exact	Tree	Exact, FFT	Exact, Tree, FFT

In the next chapters I will propose a novel implementation for kernel density estimation and compare

it to KDEpy directly.

3 Implementation

3.1 Exact Kernel Density Estimation

The implementation of a simple Kernel Density Estimation in TensorFlow is straightforward. As described in the original Tensorflow Probability Paper[14], a KDE can be constructed by using its MixtureSameFamily Distribution, given sampled `data` as follows

```
from tensorflow_probability import distributions as tfd

f = lambda x: tfd.Independent(tfd.Normal(loc=x, scale=1.))
n = data.shape[0].value

kde = tfd.MixtureSameFamily(
    mixture_distribution=tfd.Categorical(
        probs=[1 / n] * n),
    components_distribution=f(data))
```

Interestingly, due to the smart encapsulated structure of TensorFlow Probability we can use any distribution of the loc-scale family type as a kernel, if there exists an implementation for it in TensorFlow Probability. If the used Kernel has only bounded support, the implementation proposed in this paper allows to specify the support upon instantiation of the class. If the Kernel has infinite support (like a Gaussian kernel for instance) a practical support estimate is calculated by searching for approximate roots with Brent's method[15] implemented for TensorFlow in the python package `tf_quant_finance` by Google. This allows us to speed up the calculation.

However calculating an exact kernel density estimation is not always feasible as this can take a long time with a huge collection of events, especially in high energy physics. By implementing it in TensorFlow we already get a significant speed up compared to implementations in native Python, since most of TensorFlow is actually implemented in C++ and the code is optimized before running. But the computational complexity however, remains the same of course.

$$\hat{f}_h(x) = \frac{1}{nh} \sum_{k=1}^{n} K\Big(\frac{x - x_k}{h}\Big) \tag{1}$$

The computational complexity of the basic exact KDE above is $\mathcal{O}(nm)$ where n is the number of sample points to estimate from and m is the number of evaluation points (the points where you want to calculate the estimate).

To combat this complexity several methods exist.

3.2 Simple and linear Binning

The most straightforward way to decrease runtime is by limiting the number of sample points. This can be done by a binning routine, where the values at a smaller number of regular grid points are estimated from the original large number of sample points. Given a set of sample points $X = \{x_0, x_1, ..., x_k, ..., x_{n-1}, x_n\}$ with weights w_k and a set of equally spaced grid points $G = \{g_0, g_1, ..., g_l, ..., g_{n-1}, g_M\}$ where $N < n$ we can assign an estimate (or a count) c_l to each grid point g_l and use the newly found g_l's to calculate the kernel density estimation instead. This brings the computational complexity down to $\mathcal{O}(Nm)$. Depending on the number of grid points N the estimate is either more accurate and slower or less accurate and faster. However as we will see in the comparison chapter later as well, even a grid of size 1024 is enough to capture the true density with high accuracy on a million data points.[13].

As described in the excellent overview by Artur Gramacki[16] simple binning or linear binning can be used, although the last is often preferred since it is more accurate and the difference in computational complexity is negligible.

Simple binning is just the standard process of taking a weighted histogram that is divided by the sum of the sample points weights (normalization). In one dimension simple binning is binary in that it assigns a data points weight (1 for an unweighted histogram) either to the grid point (bin) left or right of itself. Linear binning on the other hand assigns a fraction of the whole weight to both grid points (bins) on either side, proportional to the closeness of grid point and data point in relation to the distance between grid points (bin width).

Mathematically linear binning in one dimension can be calculated like this:

$$c_l = c(g_l) = \sum_{\substack{x_k \in X \\ g_l < x_k < g_{l+1}}} \frac{g_{k+1} - x_k}{g_{l+1} - g_l} \cdot w_k + \sum_{\substack{x_k \in X \\ g_{l-1} < x_k < g_l}} \frac{x_k - g_{l-1}}{g_{l+1} - g_l} \cdot w_k \tag{2}$$

Implementing linear binning efficiently with TensorFlow is a bit tricky since loops should be avoided. However with some inspiration from the excellent KDEpy package[11] this can be done without using loops at all. By transforming the data such that every data point x_k can be described by an integral part (corresponding to its nearest left grid point number l) plus some fractional part (corresponding to the distance between grid point g_l and data point x_k) and applying `tf.math.bincount` twice to the transformed integral part weighting it with the fractional part times the initial weight.

The kernel density estimation can then be calculated as a mixture distribution of kernels located at the grid points, weighted with their associated grid count.

$$\hat{f}_h(x) = \frac{1}{nh} \sum_{l=1}^{N} c_l \cdot K\Big(\frac{x - g_l}{h}\Big) \tag{3}$$

3.3 Using convolution and the Fast Fourier Transform

With binning implemented, another technique to speed up the computation is rewriting the Kernel Density Estimation as convolution operation between the kernel and the grid counts calculated by the binning routine. By using the fact that a convolution is just a multiplication in Fourier space one can reduce the computational complexity down to $\mathcal{O}(\log N \cdot m)$.[16]

Using the equation (3) from above but also only evaluating it at grid points gives us

$$\hat{f}_h(g_j) = \frac{1}{nh} \sum_{l=1}^{N} c_l \cdot K\Big(\frac{g_j - g_l}{h}\Big) = \frac{1}{nh} \sum_{l=1}^{N} k_{j-l} \cdot c_l \tag{4}$$

where $k_{j-l} = K(\frac{g_j - g_l}{h})$. If we set $c_l = 0$ for all l not in the set $\{1, ..., N\}$ and notice that $K(-x) = K(x)$ we can extend equation (4) to a discrete convolution as follows

$$\hat{f}_h(g_j) = \frac{1}{nh} \sum_{l=-N}^{N} k_{j-l} \cdot c_l = \vec{c} * \vec{k} \tag{5}$$

where the two vectors look like this

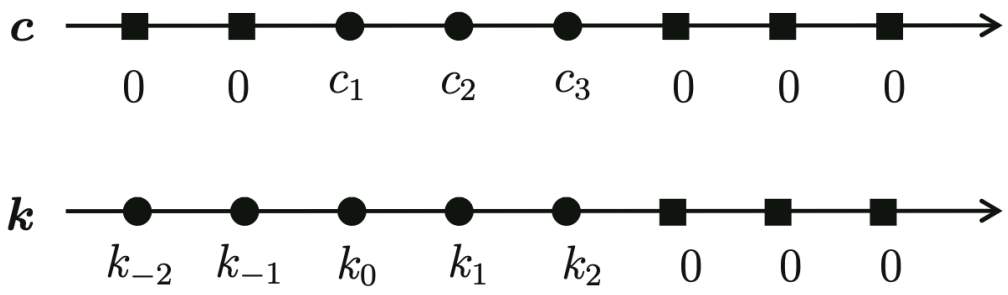


Figure 1: Vectors $\vec{c}$ and $\vec{k}$

By using the well known convolution theorem we can fourier transform $\vec{c}$ and $\vec{k}$ multiply them and inverse fourier transform them back into real space.

In TensorFlow convolutions are efficiently implemented already in this way if we use `tf.nn.conv1d`. In benchmarking using this method proved significantly faster than using `tf.signal.rfft` and

`tf.signal.irfft` to transform, multiply and inverse transform the vectors, which is implemented as an alternative as well.

This algorithm is implemented as its own class since it does not represent a complete mixture distribution anymore but calculates just the density distribution values at the speficied grid points. To still infer values for other points in the range of x `tfp.math.interp_regular_1d_grid` is used which computes a linear interpolation of values between the grid.

3.4 Improved Sheather Jones Algorithm

A different take on Kernel Density Estimators is described in the paper 'Kernel density estimation by diffusion' by Botev et al.[12]. The authors present a new adaptive kernel density estimator based on linear diffusion processes which also includes an estimation for the optimal bandwidth.

The algorithm is quite difficult to understand, a detailed explanation is given in the 'Handbook of Monte Carlo Methods'[17] by the original paper authors. However the general idea is briefly sketched below.

A critical insight is that the Gaussian kernel density estimator $\hat{f}_{h,norm}$ is the solution of the partial differential equation

$$\frac{\partial}{\partial t}\hat{f}_{h,norm}(x,t) = \frac{1}{2}\frac{\partial^2}{\partial x^2}\hat{f}_{h,norm}(x,t),\ t > 0 \tag{6}$$

with $x \in \mathbb{R}$, $\lim_{x\to\pm\infty}\hat{f}_{h,norm}(x,t) = 0$ and initial condition $\hat{f}_{h,norm}(x,0) = \Delta(x)$, where $\Delta(x) = \frac{1}{N}\sum_{k=0}^{N}\delta_{x_k}(x)$ is the empirical density of the given sample points $X = \{x_0, x_1, ..., x_k, ..., x_{n-1}, x_n\}$ and $\delta_{x_k}(x)$ is the Dirac measure at x_k.

This means the kernel density estimator can be obtained by evolving the solution of the partial differential equation (6) up to time t. The key observation is that (6) can be solved on a finite domain efficiently using the fast cosine transform - an FFT-related transform.[17]

The optimal bandwidth is often defined as the one that minimizes the mean integrated square error($MISE$)

$$MISE(t) = \mathbb{E}_f \int [\hat{f}_{h,norm}(x,t) - f(x)]^2 dx \tag{7}$$

An asymptotically optimal value t^* which minimizes a first-order asymptotic appoximation of the $MISE$ is then given by[17]

$$t^* = \Big(\frac{1}{2N\sqrt{\pi}\|f''\|^2}\Big)^{\frac{2}{5}} \tag{8}$$

Using the fact that $\|f^{(j)}\|^2 = (-1)^j \mathbb{E}_f[f^{(2j)}(X)]$, $j \geq 1$ and an initial estimation for $\|\widehat{f}_{h,norm}^{(l+2)}\|^2$ for some $l \geq 3$ one can then iteratively get a an estimation for $\|\widehat{f}_{h,norm}^{(2)}\|^2$ which can then be used to estimate t^* instead of $\|f''\|^2$. According to their handbook $l = 7$ is a suitable value to yield good practical results.

The improvment compared to the standard Sheather-Jones plug-on method[18] consists in the fact that the initial estimation of $\|\widehat{f}_{h,norm}^{(l+2)}\|^2$ is calculated by solving the partial differental equation using the fast cosine transform as described above, eliminating the need to assume normally distributed data for the initial estimate and leading to improved performance, especially for density distributions that are far from normal as seen in the next chapter.

The implementation of the algorithm in TensorFlow proposed in this paper was also inspired a lot by the python package KDEpy[11] and uses Brent's method[15] to find roots, implemented in TensorFlow in the python package `tf_quant_finance`.

One shortcoming of the improved Sheather-Jones algorithm (ISJ) is that with few data points to estimate from, it is not guarenteed to converge. If that happens, one has to use the exact, binned or FFT kernel density estimators as described above.

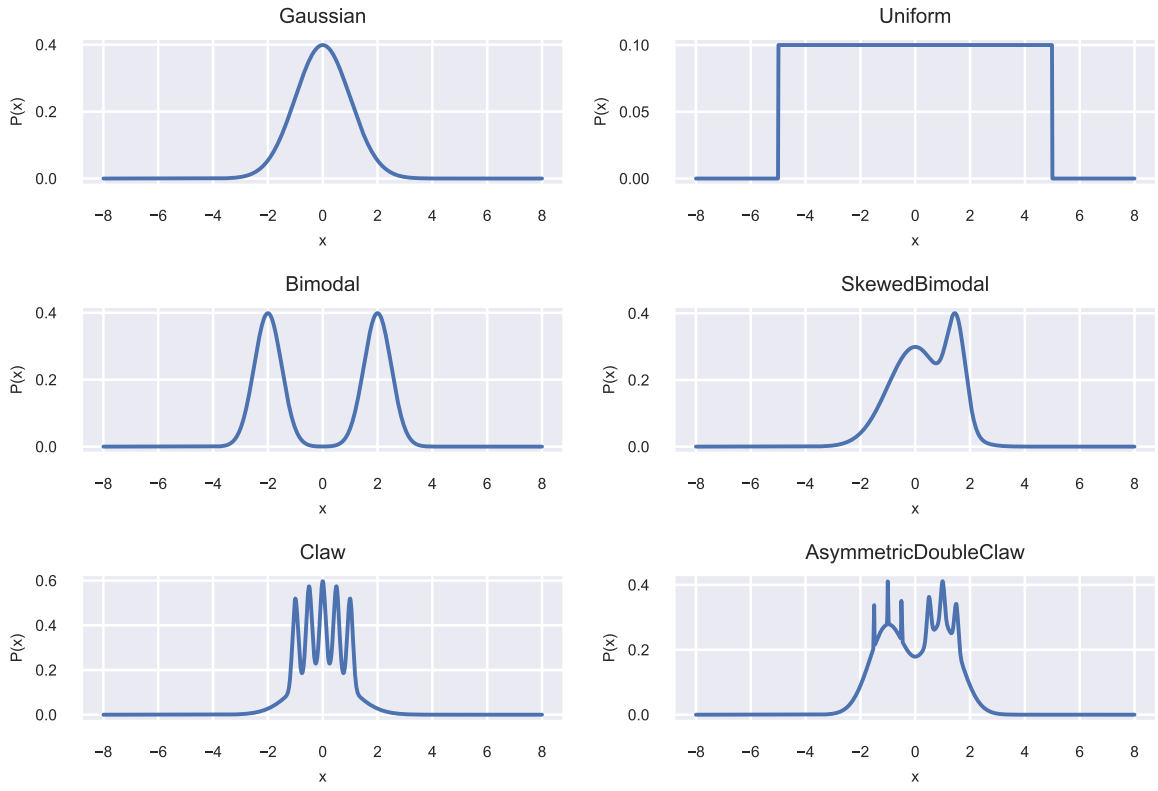
4 Comparison

```
## /opt/miniconda3/envs/ba-thesis/lib/python3.7/site-packages/zfit/util/execution
##  warnings.warn("Not running on Linux. Determining available cpus for thread can t
```

To show the efficiency and performance of the different kernel density estimation methods implemented with TensorFlow a benchmarking suite was developed. It consists of three parts: a collection of distributions to use, a collection of methods to compare and a runner module that implements helper methods to execute the methods to test against the different distributions and plot the generated datasets nicely.

4.1 Benchmark setup

To compare the different implementations multiple popular test distributions mentioned in Wand et al.[19] were used. A simple normal distribution, a simple uniform distribution, a bimodal distribution comprised of two normals, a skewed bimodal distribution, a claw distribution that has spikes and one called asymmetric double claw that has different sized spikes left and right. All comparisons were made using a standard Gaussian Kernel. Although all loc-scale family distributions of TensorFlow Probability may be used for the new implmentation proposed in this paper, the Gaussian kernel is the most used one and provides best reference to compare different implementations against eachother.



Basic implementation against Binned and FFT implementations

First we compare the basic exact kernel density estimation implementation against binned and FFT implementations run on a Macbook Pro 2013 Retina using the CPU.

4.1.1 Accuracy

For this randomly sampled data for from each test distribution is used. The number of samples per test distribution is 1000. The number of samples is restricted because calculating the exact kernel density estimation for more than 1000 kernels takes a really long time.

For the binned, FFT and ISJ algorithms we use 256 bins each.

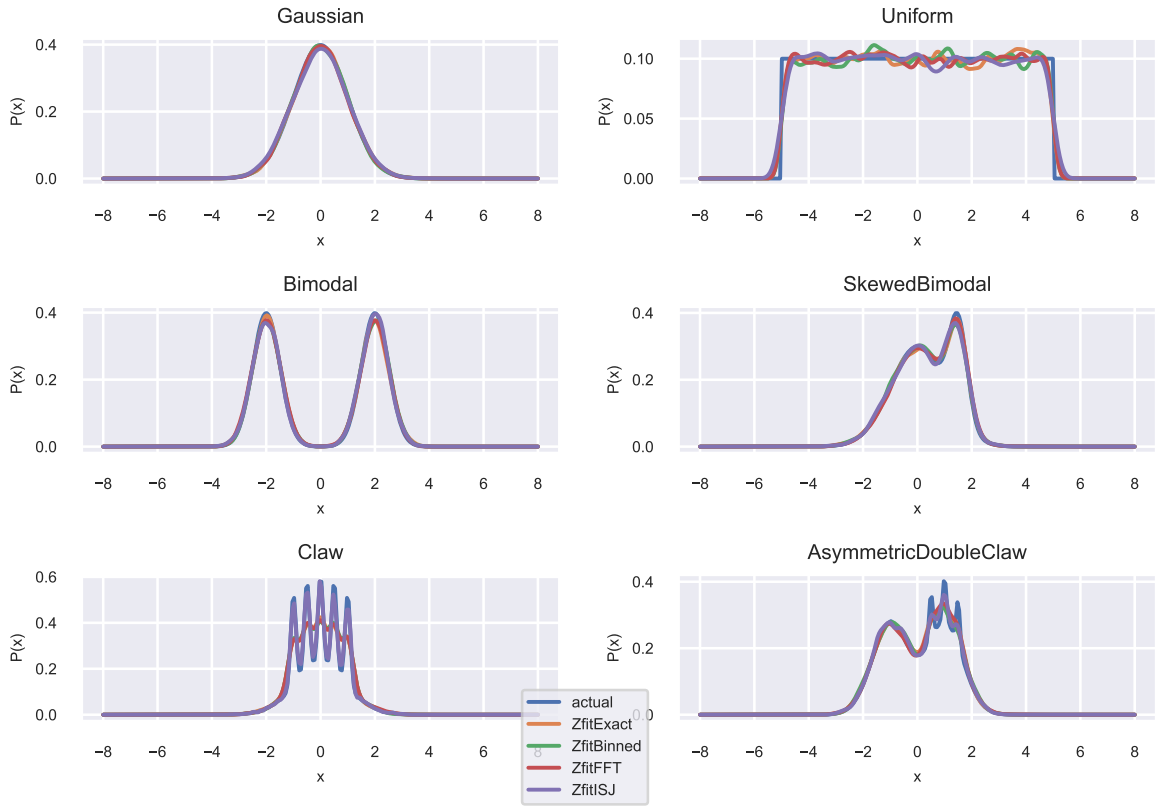
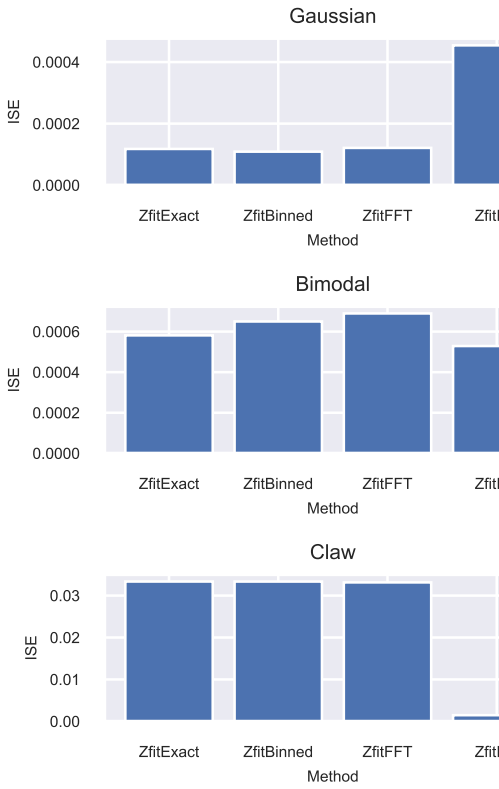


Figure 2: Comparison between the four basic algorithms 'Exact,' 'Binned,' 'FFT,' 'ISJ' with $N = 10^4$ sample points

Here it becomes obvious that the ISJ approach is especially favorable for complicated spiky distributions like the two bottom ones. We can see this in more detail below. The integrated square error (ISE) is an order of magnitude lower.

```
## <Figure size 650x450 with 1 Axes>
```



The calculated integrated square errors for all distributions are as follows:

4.1.2 Runtime

For this we use randomly sampled data for each test distribution in the range of 100 to 1000. The number of samples is restricted because larger datasets would require exponentially longer runtimes for the exact kernel density estimation.

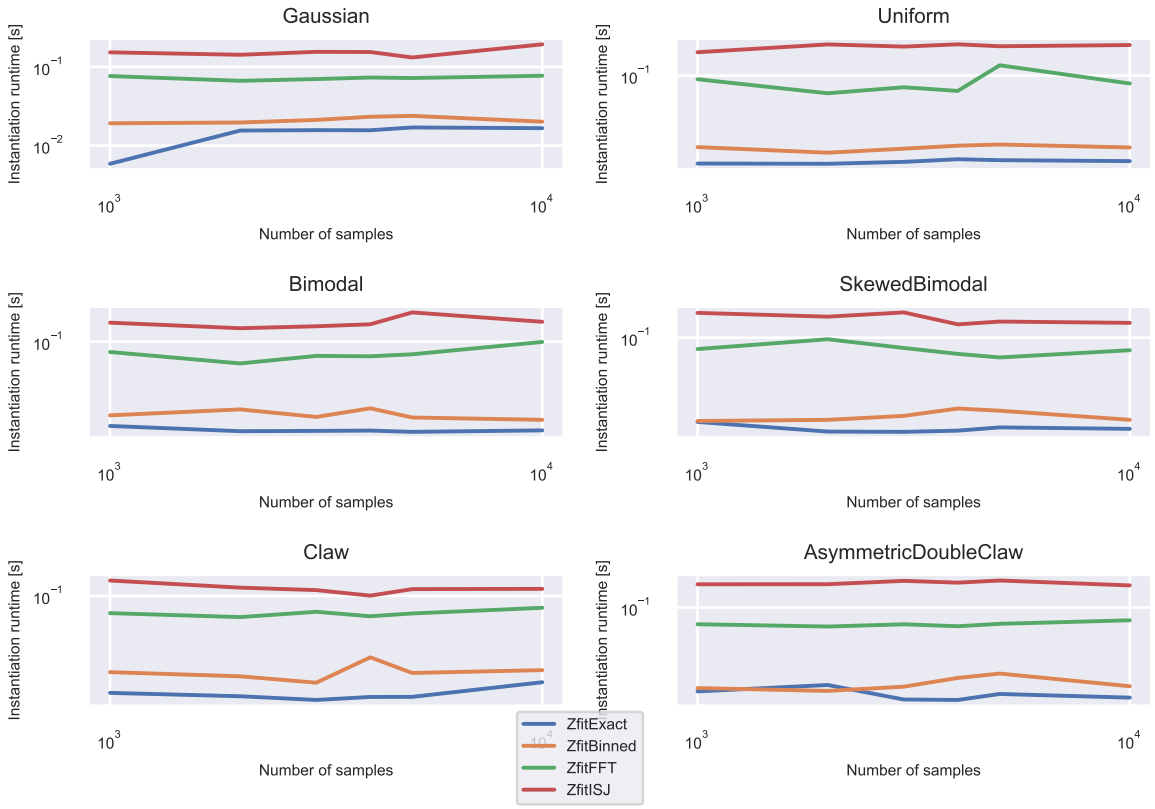


Figure 3: Runtime difference of the instantiaton step between the four basic algorithms ‘Exact,’ ‘Binned,’ ‘FFT,’ ‘ISJ’

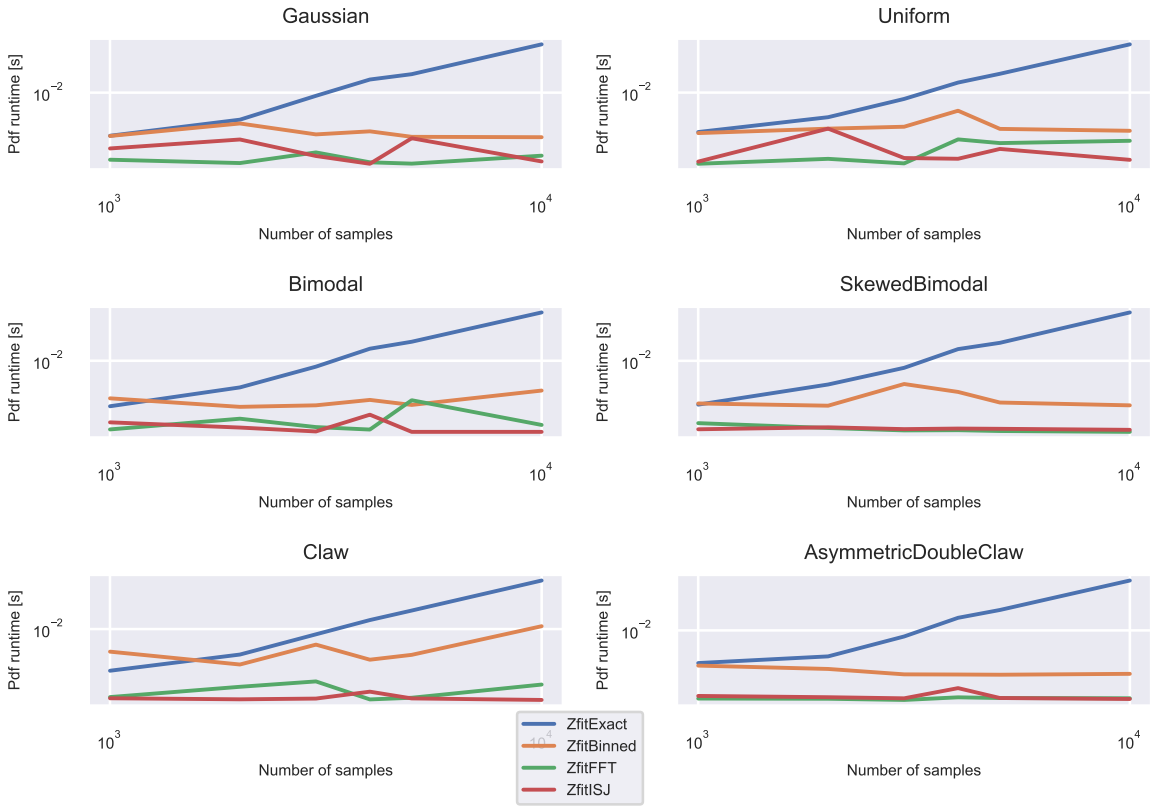


Figure 4: Runtime difference of the calculation step between the four basic algorithms ‘Exact,’ ‘Binned,’ ‘FFT,’ ‘ISJ’

4.2 New implementation against KDEpy

4.2.1 Accuracy

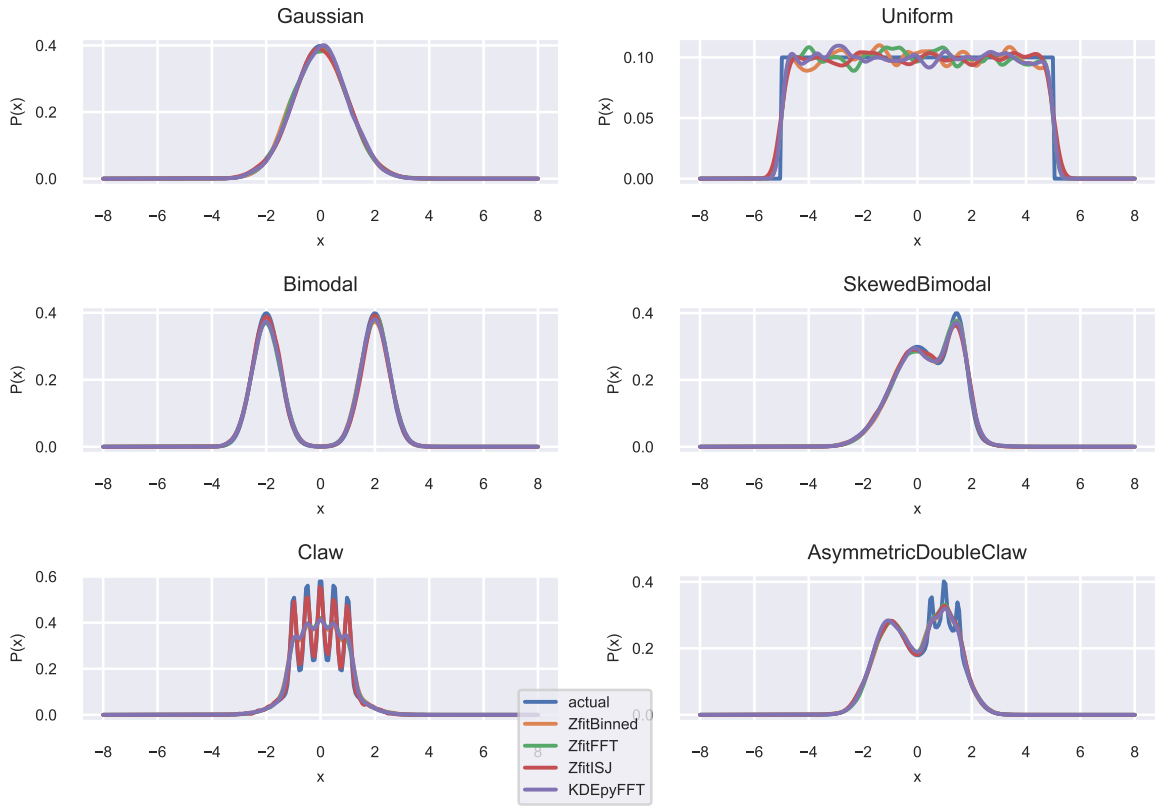


Figure 5: Comparison between the newly proposed algorithms 'Binned,' 'FFT,' 'ISJ' and the FFT based implementation in KDEpy with N=10^4 sample points

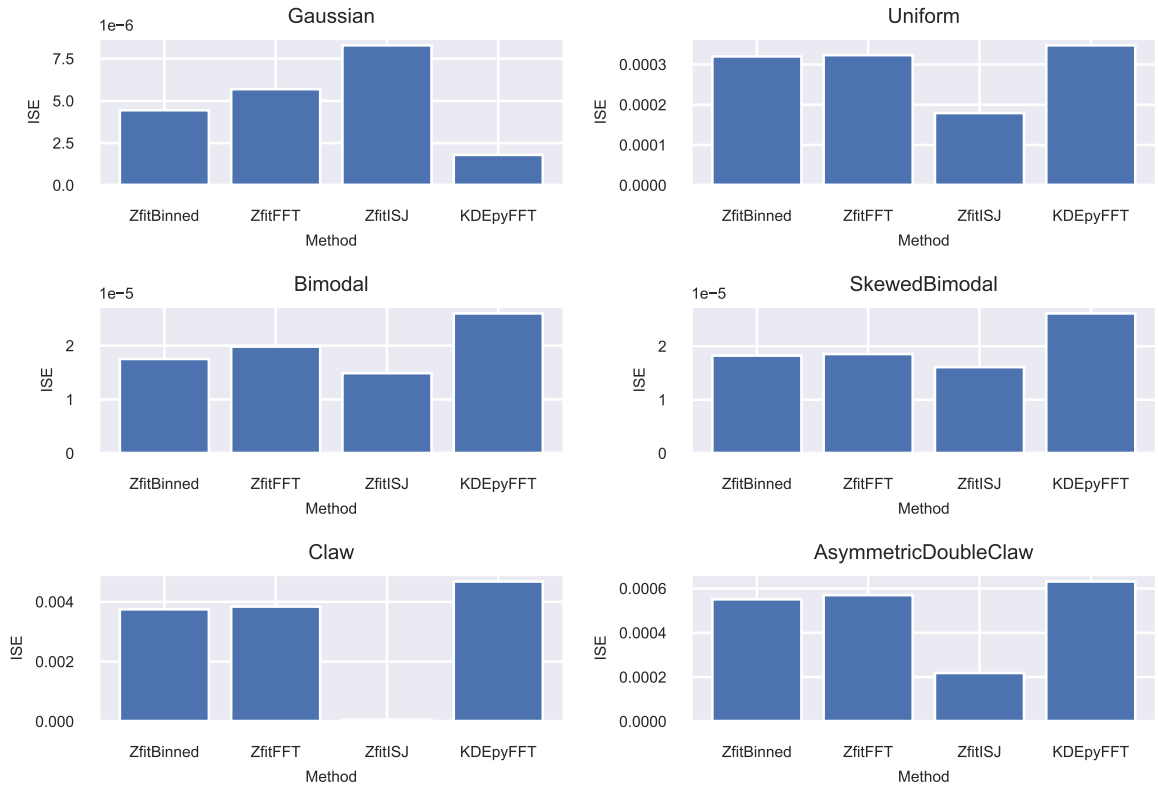


Figure 6: Integrated square errors (ISE) for the newly proposed algorithms 'Binned,' 'FFT,' 'ISJ' and the FFT based implementation in KDEpy with $N = 10^4$ sample points

4.2.2 Runtime

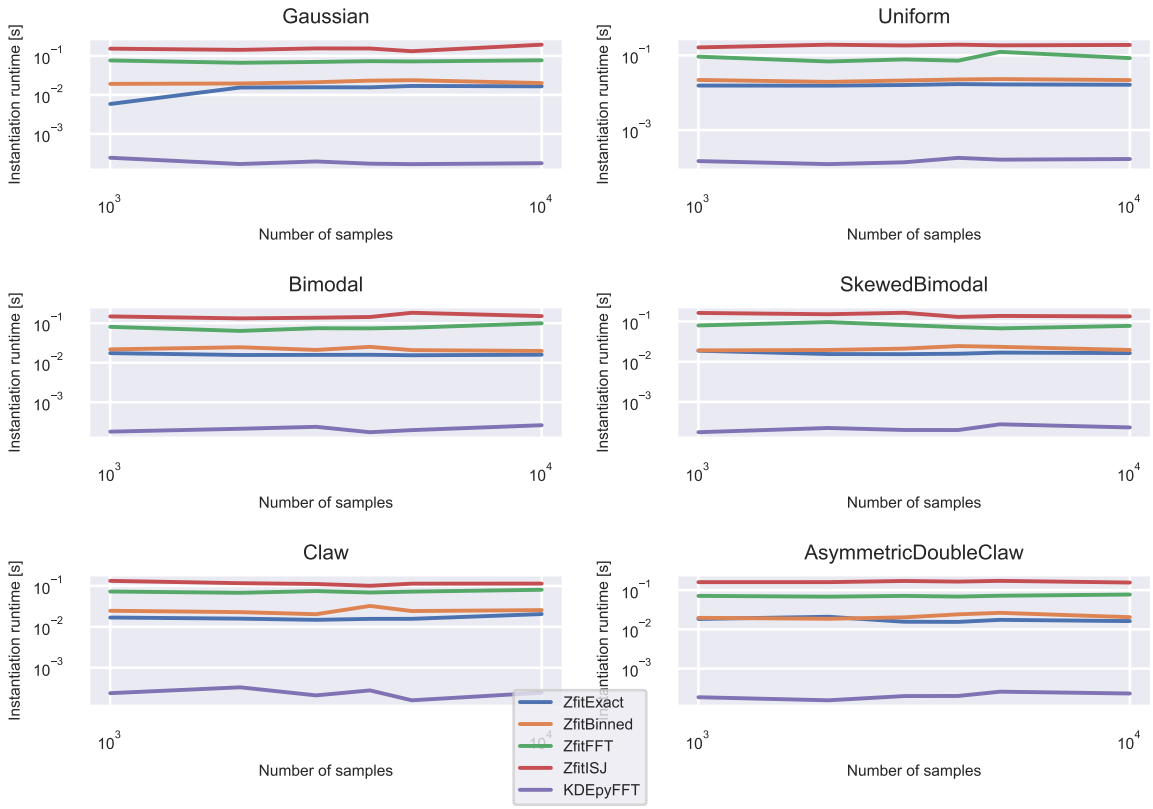


Figure 7: Runtime difference of the instantiaton step between the newly proposed algorithms ‘Binned,’ ‘FFT,’ ‘ISJ’ and the FFT based implementation in KDEpy

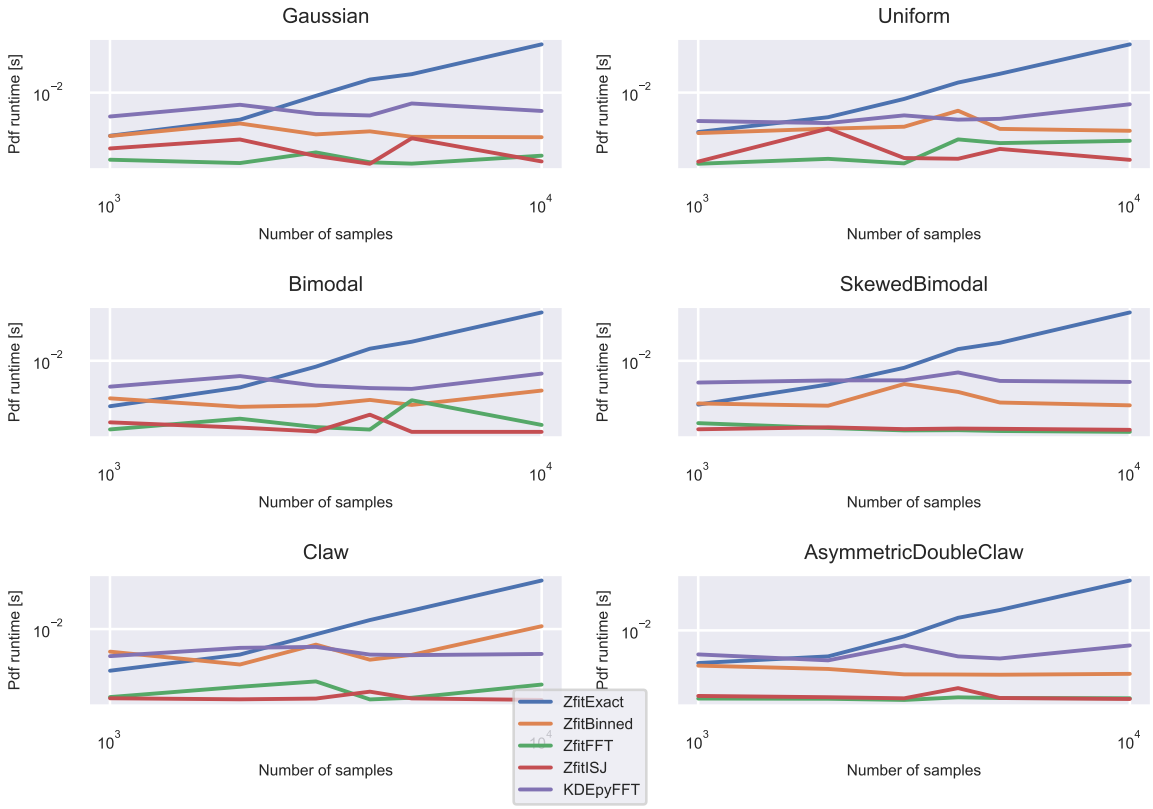


Figure 8: Runtime difference of the calculation step between the newly proposed algorithms 'Binned,' 'FFT,' 'ISJ' and the FFT based implementation in KDEpy

4.3 New implementation run with GPU support

4.3.1 Accuracy

4.3.2 Runtime

5 Summary

The benchmarking has shown that the new implementation can outperform the de-facto state of the art library 'KDEpy' in terms of runtime.

In terms of accuracy it is slightly better in most cases and in general comparable to KDEpy.

Additionally it has the benefit of allowing any distribution of the loc-scale family as kernel implemented in TensorFlow Probability[14]. At the moment this includes twelve kernels, namely `Cauchy`, `DoublesidedMaxwell`, `Gumbel`, `Laplace`, `LogLogistic`, `LogNormal`, `Logistic`, `LogitNormal`, `Moyal`, `Normal`, `PoissonLogNormalQuadratureCompound`, `SinhArcsinh`.

The proposed implementation restricts itself to the one-dimensional case. So far only KDEpy and Statsmodels implement a multidimensional KDE. Generalization to higher dimensionel kernel density estimation is however relatively straightforward to implement, since most TensorFlow API's are dimensionally agnostic.

Table 2: Comparison between KDE implementations (NR: normal reference rules, namely Scott/Silverman, CV: Cross Validation, ISJ: Improved Sheater Jones according to Botev et al.)

Feature / Library	scipy	sklearn	statsmodels	KDEpy	Zfit
Number of kernels	1	6	7 (6 slow)	9	12
Weighted data points	No	No	Non-FFT	Yes	Yes
Automatic bandwidth	NR	None	NR,CV	NR, ISJ	NR, ISJ
Multidimensional	No	No	Yes	Yes	No(planned)
Supported algorithms	Exact	Tree	Exact, FFT	Exact, Tree, FFT	Exact, Binned, FFT, ISJ

The newly proposed KDE implementation (based on TensorFlow/Zfit) achieves state-of-the-art accuracy as well as efficiency for the one-dimensional case. Furthermore it is designed to be run on parallel on multiple machines/GPUs. It is therefore optimally suited for very large datasets, as the ones produced in experimental high energy physics.

References

[1] *Processing: What to Record? - CERN Accelerating Science*, https://home.cern/science/computing/processing-what-record (accessed Nov. 16, 2020).

[2] M. Rosenblatt, *Remarks on Some Nonparametric Estimates of a Density Function*, Ann. Math. Statist. **27**, 832 (1956).

[3] T. Duong, *Kernel Density Estimation in Python*, https://www.mvstat.net/tduong/research/seminars/seminar-2001-05/ (accessed Nov. 16, 2020).

[4] M. Lerner, *Kernel Density Estimation in Python*, https://mglerner.github.io/posts/histograms-and-kernel-density-estimation-kde-2.html (accessed Nov. 16, 2020).

[5] K. Cranmer, *Kernel Estimation in High-Energy Physics*, Computer Physics Communications **136**, 198 (2001).

[6] J. Eschle, A. Puig Navarro, R. Silva Coutinho, and N. Serra, *Zfit: Scalable Pythonic Fitting*, SoftwareX **11**, 100508 (2020).

[7] *Scipy.stats.gaussian_kde — SciPy V1.5.4 Reference Guide*, https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.gaussian_kde.html (accessed Nov. 16, 2020).

[8] *Kernel Density Estimation — Statsmodels*, https://www.statsmodels.org/devel/examples/notebooks/generated/kernel_density.html (accessed Nov. 16, 2020).

[9] *Sklearn.neighbors.KernelDensity — Scikit-Learn 0.23.2 Documentation*, https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KernelDensity.html (accessed Nov. 16, 2020).

[10] J. VanderPlas, *Kernel Density Estimation in Python*, https://jakevdp.github.io/blog/2013/12/01/kernel-density-estimation/ (accessed Nov. 16, 2020).

[11] T. Odland, *KDEpy*, https://github.com/tommyod/KDEpy (accessed Nov. 16, 2020).

[12] Z. I. Botev, J. F. Grotowski, D. P. Kroese, and others, *Kernel Density Estimation via Diffusion*, The Annals of Statistics **38**, 2916 (2010).

[13] T. Odland, *Comparison | KDEpy*, https://kdepy.readthedocs.io/en/latest/comparison.html (accessed Nov. 16, 2020).

[14] J. V. Dillon, I. Langmore, D. Tran, E. Brevdo, S. Vasudevan, D. Moore, B. Patton, A. Alemi, M. D. Hoffman, and R. A. Saurous, *TensorFlow Distributions*, CoRR **abs/1711.10604**, (2017).

[15] R. P. Brent, *An Algorithm with Guaranteed Convergence for Finding a Zero of a Function*, The Computer Journal **14**, 422 (1971).

[16] A. Gramacki, *FFT-Based Algorithms for Kernel Density Estimation and Bandwidth Selection*, in *Nonparametric Kernel Density Estimation and Its Computational Aspects* (Springer, 2018), pp. 85–118.

[17] D. P. Kroese, T. Taimre, and Z. I. Botev, *Handbook of Monte Carlo Methods*, Vol. 706 (John Wiley & Sons, 2013).

[18] S. J. Sheather and M. C. Jones, *A Reliable Data-Based Bandwidth Selection Method for Kernel Density Estimation*, Journal of the Royal Statistical Society: Series B (Methodological) **53**, 683 (1991).

[19] M. P. Wand and M. C. Jones, *Kernel Smoothing* (Crc Press, 1994).

List of Tables

List of Figures

Listings