Universität Zürich[UZH]

Performance of Univariate Kernel Density Estimation methods in TensorFlow

Bachelor Thesis

Author: Marc Steiner

Supervisors: Jonas Eschle, Prof. Dr. Nicola Serra

University of Zurich

Acknowledgements

I am grateful to Prof. Dr. Nicola Serra, for giving me the opportunity to complete my bachelor thesis under his domain.

I would also like to thank my supervisor Jonas Eschle, which advised me a lot during the research, implementation and writing. He provided both technical and emotional guidance and helped me fulfill my potential.

Finally, I would like to thank my girlfriend Christa Schläppi, who unconditionally supported me through every challenge in the last year and acted as a sounding board for all the intelligent as well as not so intelligent ideas I had.

Abstract

Multiple implementations of a one-dimensional Kernel Density Estimation based on TensorFlow and 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. The newly proposed KDE implementation achieves state-of-the-art accuracy as well as efficiency for large one-dimensional data ($n \geq 10^8$) and may also provide runtime benefits for cases where the probability density estimate needs to be computed once but evaluated repeatedly. 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.

Contents

1 Introduction

1.1 Purpose of this thesis

The purpose of this thesis is to propose four novel implementations of Univariate Kernel Density Estimation based on TensorFlow[1], TensorFlow Probability[2] and zfit[3], which incorporate insights from recent papers to decrease the computational complexity and therefore runtime. The newly proposed implementations are then compared to the state of the art of Kernel Density Estimation in Python and shown to be competitive. By leveraging TensorFlow's graph based computation, the newly proposed methods to calculate a Kernel Density Estimate can benefit from parallelization and efficient computation on CPU/GPU.

First the mathematical theory will be summarized in chapter 2, before currently existing implementations of Kernel Density Estimations(KDE) in Python are discussed in chapter 3. In chapter 4 the four novel KDE implementations are proposed, which are then compared against the current state of the art in chapter 5. The findings are then summarized in chapter 6.

1.2 Kernel Density Estimation

Experimental science is based on accumulating and interpreting data, however the process of interpreting the data can be difficult. Suppose we are trying to understand some physical system. We might want to find out how probable certain events are, which is described by the probability density function (PDF) of the system. If we already have some knowledge of the system and the underlying mechanisms are well understood, we might be able to guess the shape of the probability density function and define it mathematically as a function depending on some parameters of the system (which might have physical meaning). If we then fit the function to the experimentally found data by some goodness-of-fit criterion like log-likelihood or χ^2, we can get information about the parameters and learn more about the system itself. But what if the underlying mechanisms are too complex to be fully described analytically and the knowledge of the system is too poor to describe the model as an exact mathematical function?

In the particle accelerator at CERN for instance, a whooping 25 gigabytes of data is recorded per second[4], resulting from the process of many physical interactions that occur almost simultaneously. It is often not feasible to anticipate features of the distribution one observes experimentally, as the distribution is comprised of many different distributions, which result from all the different physical interactions. However, through Monte Carlo based simulation, samples can be drawn from the theoretical distribution. These can be used in conjunction with so called non-parametric methods, that approximate the shape of the drawn distribution without the need for a predefined mathematical model. The perhaps simplest non-parametric method is the Histogramm. By summing the the data up in discrete bins the underlying probability distribution can be approximated, without needing any prior knowledge of the system itself. However Histograms tend to produce PDFs that are highly dependent on bin width and bin positioning, meaning the interpretation of the data changes a lot by two arbitrary parameters.

A more sophisticated non-parametric method is the Kernel Density Estimation (KDE), which can be looked at as a sort of generalized histogram[5]. In a Kernel Density Estimation each data point is substituted with a so called kernel function that specifies how much it influences its neighboring regions. This kernel functions can then be summed up to get an estimate of the probability density distribution, quite similarly as summing up data points inside bins. However since the kernel functions are centered on the data points directly, KDE circumvents the problem of arbitrary bin positioning[6]. Since KDE still depends on kernel bandwidth (a measure of the spread of the kernel function) 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 bandwidth than histograms do on bin width and it is much easier to specify rules for an approximately optimal kernel bandwidth than it is to do so for bin width[7]. Mathematical research has shown that it is possible to compute an approximately optimal bandwidth value, which is not possible for bin width. Another benefit is that one gets a smooth distribution by specifying a smooth kernel function, 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)[8]. However there is still room for improvement in terms of accuracy and computation speed and certain more sophisticated approaches to Kernel Density Estimation have been proposed in dependence on specific areas of application[8].

1.3 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 High Energy Physics (HEP) that includes KDE is the ROOT/RooFit toolkit written in C++. However as the amount of experimental data grows (like at CERN), so grows the computational burden and traditional methods to calculate kernel density estimation become cumbersome. In addition, 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 recently 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[3].

TensorFlow provides the intuitive accessibility of Python while ensuring speed and efficiency because the underlying operations are implemented in C++. TensorFlow implements so-called graph-based computation, which means that it builds a graph describing the computation to be done before it actually executes it. By analyzing the graph TensorFlow can then optimize the algorithm and schedule parts that can be executed in parallel to be run on different CPUs or GPUs, which is especially important for large data and lengthy calculations. Additionally TensorFlow supports automatic differentiation. Every operation in the graph implements its own derivative and the therefore the whole graph can be differentiated by using the chain rule. This is especially important for applications where we want to minimize the gradient such as in neural network based computations. Finally, TensorFlow can be used together with other scientific libraries in Python and TensorFlow accepts NumPy[9] arrays as input, although computations outside of the graph don't benefit of TensorFlow's optimizations and its automatic differentiation. Being able to calculate a Kernel Density Estimation using TensorFlow and zfit has therefore numerous advantages for large data.

So far only exact Kernel Density Estimations exist for TensorFlow, however as seen in chapter 2, the KDE can be approximated using multiple mathematical tricks with only a negligible decrease in accuracy, while decreasing the computational complexity substantially.

1.4 Univariate case

The newly proposed implementations in this thesis are limited to the one-dimensional case, since this is the case which is most often used and therefore benefits the most of decreased runtime. It is feasible to extend the implementation to the multi-dimensional case in the future, however this would require more work due to not quite identical APIs to the univariate case. In addition one must ensure that the kernel functions used would be multi-dimensional themselves.

2 Theory

2.1 Exact Kernel Density Estimation

Given a set of n sample points x_k ($k = 1, \cdots, n$), an exact Kernel Density Estimation $\hat{f}_h(x)$ can be calculated as

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

where $K(x)$ is called the kernel function, h is the bandwidth of the kernel and x is the value for which the estimate is calculated. The kernel function defines the shape and size of influence of a single data point over the estimation, whereas the bandwidth defines the range of influence. Most typically a simple Gaussian distribution ($K(x) := \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$) is used as kernel function. The larger the bandwidth parameter h the larger is the range of influence of a single data point on the estimated distribution.

The computational complexity of the exact KDE above is given by $\mathcal{O}(nm)$ 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). There exist several approximative methods to decrease this complexity and therefore decrease the runtime as well.

2.2 Binning

The most straightforward way to decrease the computational complexity 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 larger 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 $G = \{g_0, g_1, ..., g_l, ..., g_{n-1}, g_N\}$ 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 to calculate the kernel density estimation instead.

$$\hat{f}_h(x) = \frac{1}{nh} \sum_{l=1}^{N} c_l \cdot K\Big(\frac{x - g_l}{h}\Big) \tag{2}$$

This lowers the computational complexity down to $\mathcal{O}(N \cdot m)$. Depending on the number of grid points N there is tradeoff between accuracy and speed. However as we will see in the comparison chapter later as well, even for ten million sample points, a grid of size 1024 is enough to capture the true density with high accuracy[10]. As described in the extensive overview by Artur Gramacki[11] 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.

2.2.1 Simple binning

Simple binning is just the standard process of taking a weighted histogram and then normalizing it by dividing each bin by the sum of the sample points weights. In one dimension simple binning is binary in that it assigns a sample point's weight ($w_k = 1$ for an unweighted histogram) either to the grid point (bin) left or right of itself.

$$c_l = c(g_l) = \sum_{\substack{x_k \in X \\ \frac{g_l + g_{l-1}}{2} < x_k < \frac{g_{l+1} + g_l}{2}}} w_k \tag{3}$$

where c_l is the value for grid point g_l depending on sample points x_k and their associated weights w_k.

2.2.2 Linear binning

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).

$$c_l = c(g_l) = \sum_{\substack{x_k \in X \\ g_l < x_k < g_{l+1}}} \frac{g_{k+1} - x_k}{g_{l+1} - g_l} \cdot w_k + \sum_{\substack{x_k \in X \\ g_{l-1} < x_k < g_l}} \frac{x_k - g_{l-1}}{g_{l+1} - g_l} \cdot w_k \tag{4}$$

where c_l is the value for grid point g_l depending on sample points x_k and their associated weights w_k.

2.3 Using convolution and the Fast Fourier Transform

Another technique to speed up the computation is rewriting the Kernel Density Estimation as convolution operation between the kernel function and the grid counts (bin counts) calculated by the binning routine given above.

By using the fact that a convolution is just a multiplication in Fourier space and only evaluating the KDE at grid points one can reduce the computational complexity down to $\mathcal{O}(\log N \cdot N)$.[11]

Using the equation (2) from above only evaluated at grid points gives us

$$\hat{f}_h(g_j) = \frac{1}{nh} \sum_{l=1}^{N} c_l \cdot K\Big(\frac{g_j - g_l}{h}\Big) = \frac{1}{nh} \sum_{l=1}^{N} k_{j-l} \cdot c_l \tag{5}$$

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 (5) to a discrete convolution as follows

$$\hat{f}_h(g_j) = \frac{1}{nh} \sum_{l=-N}^{N} k_{j-l} \cdot c_l = \vec{c} * \vec{k} \tag{6}$$

where the two vectors look like this

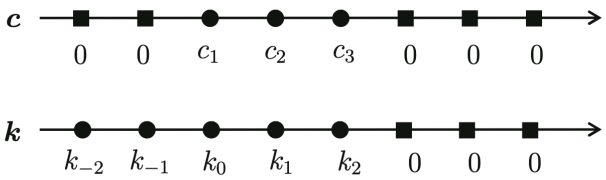

Figure 1: Vectors $\vec{c}$ and $\vec{k}$

By using the well known convolution theorem we can fourier transform $\vec{c}$ and $\vec{k}$, multiply them and inverse fourier transform them again to get the result of the discrete convolution.

However, due to the limitation of evaluating only at the grid points themselves, one needs to interpolate to get values for the estimated distribution at points in between.

2.4 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. A more detailed and extensive explanation of the algorithm as well as an implementation in Matlab is given in the 'Handbook of Monte Carlo Methods'[13] by the original paper authors. However the general idea is briefly sketched below.

The optimal bandwidth is often defined as the one that minimizes the mean integrated square error ($MISE$), which is given by the expectation value $\mathbb{E}_f$ of the integrated square error between the Kernel Density Estimation $\hat{f}_{h,norm}(x)$ and the true probability density function $f(x)$.

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

To find the optimal bandwidth it is useful to look at the second order derivative $f^{(2)}$ of the unknown distribution as it indicates how many peaks the distribution has and how steep they are. For a distribution with many narrow peaks close together a smaller bandwidth leads to better result since the peaks do not get smeared together to a single peak for instance.

As derived by Wand and Jones an asymptotically optimal bandwidth h_{AMISE} which minimizes a first-order asymptotic approximation of the $MISE$ is then given by[14]

$$h_{AMISE}(x) = \Big(\frac{1}{2N\sqrt{\pi}\|f^{(2)}(x)\|^2}\Big)^{\frac{1}{5}} \tag{8}$$

where N is the number of sample points (or grid points if binning is used).

As Sheather and Jones showed, this second order derivative can be estimated, starting from an even higher order derivative $\|f^{(l+2)}\|^2$ by using the fact that $\|f^{(j)}\|^2 = (-1)^j \mathbb{E}_f[f^{(2j)}(X)],\ j \geq 1$

$$h_j = \left(\frac{1 + 1/2^{j+1/2}}{3} \frac{1 \times 3 \times 5 \times \cdots \times (2j-1)}{N\sqrt{\pi/2}\left\|f^{(j+1)}\right\|^2} \right)^{1/(3+2j)} = \gamma_j(h_{j+1}) \tag{9}$$

where h_j is the optimal bandwidth for the j-th derivative of f and the function γ_j defines the dependency of h_j on h_{j+1}

Their proposed plug-in method works as follows:

1. Compute $\|\widehat{f}^{(l+2)}\|^2$ by assuming that f is the normal pdf with mean and variance estimated from the sample data
2. Using $\|\widehat{f}^{(l+2)}\|^2$ compute h_{l+1}
3. Using h_{l+1} compute $\|\widehat{f}^{(l+1)}\|^2$
4. Repeat steps 2 and 3 to compute h^l, $\|\widehat{f}^{(l)}\|^2$, h^{l-1}, $\cdots$ and so on until $\|\widehat{f}^{(2)}\|^2$ is calculated
5. Use $\|\widehat{f}^{(2)}\|^2$ to compute h_{AMISE}

The weakest point of this procedure is the assumption that the true distribution is a Gaussian density function in order to compute $\|\widehat{f}^{(l+2)}\|^2$. This can lead to arbitrarily bad estimates of h_{AMISE}, when the true distribution is far from being normal.

Therefore Botev et al. took this idea further[12]. Given the function $\gamma^{[k]}$ such that

$$\gamma^{[k]}(h) = \underbrace{\gamma_1\left(\cdots \gamma_{k-1}\left(\gamma_k\right.\right.}_{k \text{ times}}(h))\cdots) \tag{10}$$

h_{AMISE} can be calculated as

$$h_{AMISE} = h_1 = \gamma^{[1]}(h_2) = \gamma^{[2]}(h_3) = \cdots = \gamma^{[l]}(h_{l+1}) \tag{11}$$

By setting $h_{AMISE} = h_{l+1}$ and using fixed point iteration to solve the equation

$$h_{AMISE} = \gamma^{[l]}(h_{AMISE}) \tag{12}$$

the optimal bandwidth h_{AMISE} can be found directly.

This eliminates the need to assume normally distributed data for the initial estimate and leads to improved performance, especially for density distributions that are far from normal as seen in the next chapter. According to their paper increasing l beyond $l = 5$ does not increase the accuracy in any practically meaningful way. The computation is especially efficient if $\gamma^{[5]}$ is computed using the Discrete Cosine Transform - an FFT related transformation.

The optimal bandwidth h_{AMISE} can then either be used for other kernel density estimation methods (like the FFT-approach discussed above) or also to compute the kernel density estimation directly using another Discrete Cosine Transform.

2.5 Using specialized kernel functions and their series expansion

Lastly there is an interesting approach described by Hofmeyr[15] that uses special kernels of the form $K(x) := poly(|x|) \cdot exp(-|x|)$ where $poly(|x|)$ denotes a polynomial of finite degree.

Given the kernel with a polynom of order α

$$K(x) := \sum_{j=0}^{\alpha} |x|^j \cdot e^{-|x|} \tag{13}$$

the kernel density estimation is given by (equation (1))

$$\hat{f}_h(x) = \frac{1}{nh} \sum_{k=1}^{n} \sum_{j=0}^{\alpha} (\frac{|x - x_k|}{h})^j \cdot e^{(-\frac{|x-x_k|}{h})} \tag{14}$$

Hofmeyr showed that the above kernel density estimator can be rewritten as

$$\hat{f}_h(x) = \sum_{j=0}^{\alpha} \sum_{i=0}^{j} \binom{j}{i} (\exp(\frac{x_{(\tilde{n}(x))} - x}{h}) x^{j-i} \ell(i, \tilde{n}(x)) + \exp(\frac{x - x_{(n(x))}}{h})(-x)^{j-i} r(i, \tilde{n}(x))) \tag{15}$$

where $\tilde{n}(x)$ is defined to be the number of sample points less than or equal to x ($\tilde{n}(x) = \sum_{k=1}^{n} \delta_{x_k}((-\infty, x])$, where $\delta_{x_k}(\cdot)$ is the Dirac measure of x_k) and $\ell(i, \tilde{n})$ and $r(i, \tilde{n})$ are given by

$$\ell(i, \tilde{n}) = \sum_{k=1}^{\tilde{n}} (-x_k)^i \exp(\frac{x_k - x_{\tilde{n}}}{h}) \tag{16}$$

$$r(i, \tilde{n}) = \sum_{k=\tilde{n}+1}^{\tilde{n}} (x_k)^i \exp(\frac{x_{\tilde{n}} - x_k}{h}) \tag{17}$$

Or put differently, all values of $\hat{f}_h(x)$ can be specified as linear combinations of terms in $\bigcup_{i,\tilde{n}} \{\ell(i, \tilde{n}), r(i, \tilde{n})\}$. Finally, the critical insight lies in the fact that $\ell(i, \tilde{n})$ and $r(i, \tilde{n})\}$ can be computed recursively as follows

$$\ell(i, \tilde{n} + 1) = \exp(\frac{x_{\tilde{n}} - x_{\tilde{n}+1}}{h}) \ell(i, \tilde{n}) + (-x_{\tilde{n}+1})^i \tag{18}$$

$$r(i, \tilde{n}-1) = \exp(\frac{x_{\tilde{n}-1} - x_{\tilde{n}}}{h})(r(i, \tilde{n}) + (x_{\tilde{n}})^i) \tag{19}$$

Using this recursion one can then calculate the Kernel Density Estimation with a single forward and a single backward pass over the ordered set of all $x_{\tilde{n}}$ leading to a computational complexity of $\mathcal{O}((\alpha + 1)(n + m))$ where α is the order of the polynom, n is the number of sample points and m is the number of evaluation points. What is important to note here is that this is the only method that defines a computational gain for an exact Kernel Density Estimation. Although we can also use binning to approximate it and reduce the computational complexity even further, it is already pretty fast for the exact computation.

3 Current state of the art

To get a sense of what the current state of Kernel Density Estimation in Python is, we will look at several current implementations and their distinctions. This will lead to an understanding of their different properties and allow us to compare and classify the new methods proposed in the next chapter inside Python's ecosystem.

The most popular KDe implementations in Python are SciPy's `gaussian_kde`[16], Statsmodels' KDE-`Univariate`[17] Scikit-learn's `KernelDensity` package[18] as well as KDEpy by Tommy Odland[19].

The question of the optimal KDE implementation for any situation 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. On the other hand, Scikit-learn includes a k-d-tree based kernel density estimation, which is not available in Statsmodels. As Jake VanderPlas was able to show in his comparison[20] Scikit-learn's tree based approach to compute the kernel density estimation was the most efficient in the vast majority of cases in 2013.

However the new implementation proposed by Tommy Odland in 2018 called KDEpy[19] was able to outperform all previous implementations (even Scikit-learn's tree based approach) in terms of runtime for a given accuracy by a factor of at least one order of magnitude, using an FFT based approach. Additionally it 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], which was discussed in the previous chapter.

This makes KDEpy the de-facto standard of Kernel Density Estimation in Python.

Table 1: Comparison between KDE implementations by Tommy Odland[10] (NR: normal reference rules, namely Scott/Silverman, CV: Cross Validation, ISJ: Improved Sheater Jones according to Botev et al.[12])

Feature / Library	scipy	sklearn	statsmodels	KDEpy
Number of kernel functions	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

Therefore the novel implementation for kernel density estimation based on TensorFlow and zfit proposed in this thesis is compared to KDEpy directly to show that it can outperform KDEpy in terms of runtime and accuracy for large datasets ($n \geq 10^8$).

4 Implementation

In addition to the rather simple case of an exact univariate Kernel Density Estimation (henceforth called `ZfitExact`), four conceptually different novel implementations in zfit and TensorFlow are proposed. A method based on simple or linear binning (called `ZfitBinned`), a method using the FFT algorithm (called `ZfitFFT`), a method based on the improved Sheather Jones algorithm (called `ZfitISJ`) and lastly a method based on Hofmeyr's method of using a specialized kernel of the form $poly(x) \cdot \exp(x)$ (called `ZfitHofmeyr`) and recursive computation of the bases needed to calculate the kernel density estimations as linear combination. All methods are implemented for the univariate case only.

Important to note is that for `ZfitISJ` and `ZfitFFT` simple or linear binning is preliminary.

4.1 Advantages of using zfit TensorFlow

The benefit of using zfit, which is based on TensorFlow is that it (and TensorFlow) is optimized for parallel processing and CPU as well as GPU processing. TensorFlow uses graph based computation, which means that it generates a computational graph of all operations to be done and their order, before actually executing the computation. This has to key advantages.

First it allows TensorFlow to act as a kind of Compiler and optimize the code before running and schedule graph branches, that are independent of each other to different processors to be executed in parallel. Operations in TensorFlow are often implemented twice, once for CPU and once for GPU to make use of the different environments available on each processor type. Also, similarly to NumPy[9], TensorFlow's underlying operations are programmed in C++ and therefore benefit from static typing and compile time optimization.

Secondly it allows fore automatic differentiation, meaning that every TensorFlow operation defines its own derivative. Using the chain rule, TensorFlow can then automatically compute the gradient of the whole program, which is especially useful for non-parametric fitting (i.e. gradient descent computations in function approximations using a neural network).

4.2 Exact univariate Kernel Density Estimation

The implementation of an exact univariate Kernel Density Estimation in TensorFlow is straightforward. As described in the original Tensorflow Probability Paper[2], a KDE can be constructed by using its `MixtureSameFamily` distribution class, given sampled `data`, their associated `weights` and bandwidth h as follows

```
import tensorflow as tf
from tensorflow_probability import distributions as tfd

data = [...]
weights = [...]
h = ...

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

probs = weights / tf.reduce_sum(weights)

kde = tfd.MixtureSameFamily(
    mixture_distribution=tfd.Categorical(
        probs=probs),
    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 as long as it follows the Distribution contract 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[21] implemented for TensorFlow in the python package

`tf_quant_finance` by Google. This allows us to speed up the calculation as negligible contributions from far away kernels are neglected.

However calculating an exact kernel density estimation is not always feasible as this can take a long time with a huge collection of data points. By implementing it in TensorFlow we already get a significant speed up compared to implementations in native Python, due to TensorFlow's advantages mentioned above. Nonetheless the computational complexity remains the same and for large data this can still make the exact KDE impractical.

An exact Kernel Density Estimation using zfit called `ZfitExact` is implemented as a `zfit.pdf.WrapDistribut` class, which is zfit's class type for wrapping TensorFlow Probability distributions.

4.3 Binned method

The method `ZfitBinned` also implements Kernel Density Estimation as a constructed `MixtureSameFamily` distribution, however it bins the data (either simple or linearly) to an equally spaced grid and uses only the grid points weighted by their grid count as kernel locations (see section 2.2).

Since simple binning is already implemented in TensorFlow in the function `tf.histogram_fixed_width`, the contribution of this thesis for `ZfitBinned` lies in an implementation of linear binning in TensorFlow. Implementing linear binning efficiently with TensorFlow is a bit tricky since loops should be avoided as the graph based computation is fastest with vectorized operations and loops pose a significant runtime overhead. However with some inspiration from the KDEpy package[19] this can be done without using loops at all.

First, every data point x_k is transformed to $\tilde{x}_k$ in the following way (the transformation can be vectorized)

$$\tilde{x}_k = \frac{x_k - g_0}{\Delta g} \tag{20}$$

where Δg is the grid spacing and g_0 is the left-most value of the grid.

Given this transformation every $\tilde{x}_k$ can then be described by an integral part $\tilde{x}_k^{int}$ (equal to its nearest left grid point index $l = x_k^{int}$) plus some fractional part $\tilde{x}_k^{frac}$ (corresponding to the additional distance between grid point g_l and data point x_k). The linear binning can then be solved in the following way.

For data points on the right side of the grid point g_l: The fractional parts of the data points are summed if the integral parts equal l.

For data points on the left side of the grid point g_l: 1 minus the fractional parts of the data points are summed if the integral parts equal $l - 1$.

Including the weights this looks as follows

$$c_l = c(g_l) = \sum_{\substack{\tilde{x}_k^{frac} \in \tilde{X}^{frac} \\ l = \tilde{x}_k^{int}}} \tilde{x}_k^{frac} \cdot w_k + \sum_{\substack{\tilde{x}_k^{frac} \in \tilde{X}^{frac} \\ l = \tilde{x}_k^{int}+1}} (1 - \tilde{x}_k^{frac}) \cdot w_k \tag{21}$$

Left and right side sums can then be calculated efficiently with the TensorFlow function `tf.math.bincount`.

The binned method `ZfitBinned` is implemented in the same class definition as `ZfitExact`, the binning can be enabled by specifying a constructor argument.

4.4 FFT based method

The KDE method called `ZfitFFT`, which uses the FFT based method (discussed in section 2.3), is implemented as a `zfit.pdf.BasePdf` class. It is not based on TensorFlow Probability as there is it does not use a `MixtureDistribution` but instead calculates the estimate for the given grid points directly. 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. In TensorFlow one-dimensional discrete convolutions are efficiently implemented already if we use `tf.nn.conv1d`. In benchmarking using this method to calculate the estimate 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 option as well.

4.5 ISJ based method

The method called `ZfitISJ` is also implemented as a `zfit.pdf.BasePdf` class. After using simple or linear binning to calculate the grid counts, the estimate for the grid points is calculated using the improved Sheather Jones method (discussed in section 2.4).

To find the roots for γ^l in equation (12) Brent's method[21] implemented `tf_quant_finance` is used again. To avoid loops the iterative function γ^l is statically unrolled for $l = 5$, since higher values would not lead to any practical differences according to the paper authors. For the Discrete Cosine Transform `tf.signal.dct` is used.

4.6 Specialized kernel method

The method called `ZfitHofmeyr` is again implemented as a `zfit.pdf.BasePdf` class. It uses specialized kernels of the form $poly(x) \cdot \exp(x)$ (as discussed in 2.5).

However due to the recursive nature of the method, an implementation in TensorFlow directly displayed the same performance as using an exact Kernel Density Estimation based on a mixture distribution. This is due to the fact, that recursive functions of this type can not be vectorized and have to

be implemented using loops. Implementing the recursion using NumPy and `tf.numpy_function` (which wraps a NumPy based function to create a single TensorFlow op) was an order of magnitude faster, but also slower than all approximative methods given above.

Finally, implementing the method in C++ directly as a custom TensorFlow operation named `tf.hofmeyr_kde` yielded the competitive execution runtime expected from theory. The code for the C++ based implementations is based on the C++ code used for the author's own R package FKSUM[22].

So far the custom TensorFlow operation is only implemented as a proof of concept and poses severe limitations. Its C++ library has to be compiled for every platform specifically and it currently does not compute its own gradient and therefore does not support TensorFlow's automatic differentiation. It also is implemented only for CPU and does therefore not benefit of using the GPU.

5 Comparison

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.

5.1 Benchmark setup

To compare the different implementations multiple popular test distributions mentioned in Wand et al.[14] 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. The data is sampled randomly from each test distribution.

All comparisons were made using a standard Gaussian kernel function. Although all loc-scale family distributions of TensorFlow Probability may be used for the new implementation proposed in this paper, the Gaussian kernel function is the most used one and provides best reference to compare different implementations against each other.

For all implementations that use binning $2^10 = 1024$ bins were used. This is the default used in KDEpy, a power of 2 (which is favorable for FFT based algorithms), results in an exact kernel density calculation for the lowest sample size used (10^3) but also yields results with high accuracy for the highest sample size used (10^8).

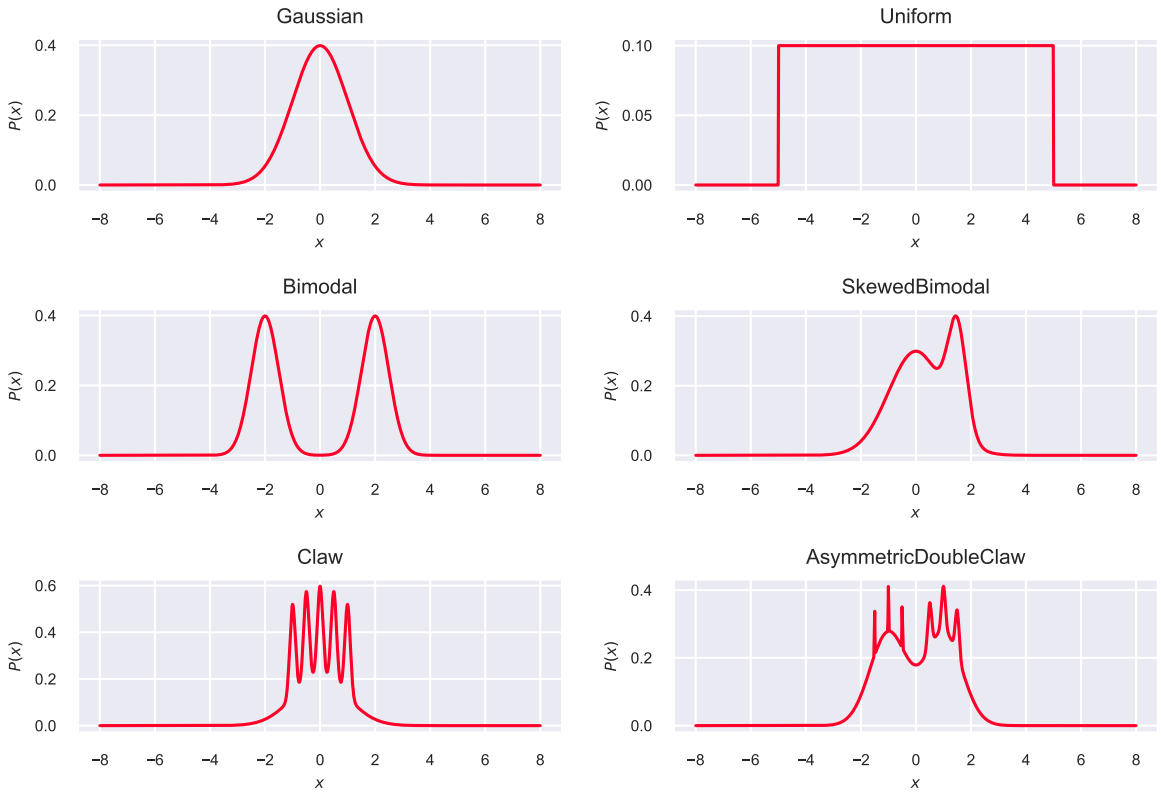

Figure 2: Distributions used for the comparisons

5.2 Differences of Exact, Binned, FFT and ISJ implementations

First, the exact kernel density estimation implementation is compared against linearly binned, FFT and ISJ implementations run on a Macbook Pro 2013 Retina using the CPU.

The sample sizes lie in the range of 10^3 to 10^4. The number of samples is restricted because calculating the exact kernel density estimation for more than 10^4 kernels is computationally unfeasible (larger datasets would lead to an exponentially larger runtime).

5.2.1 Accuracy

Plotted below are the comparisons for sample size of 10^4.

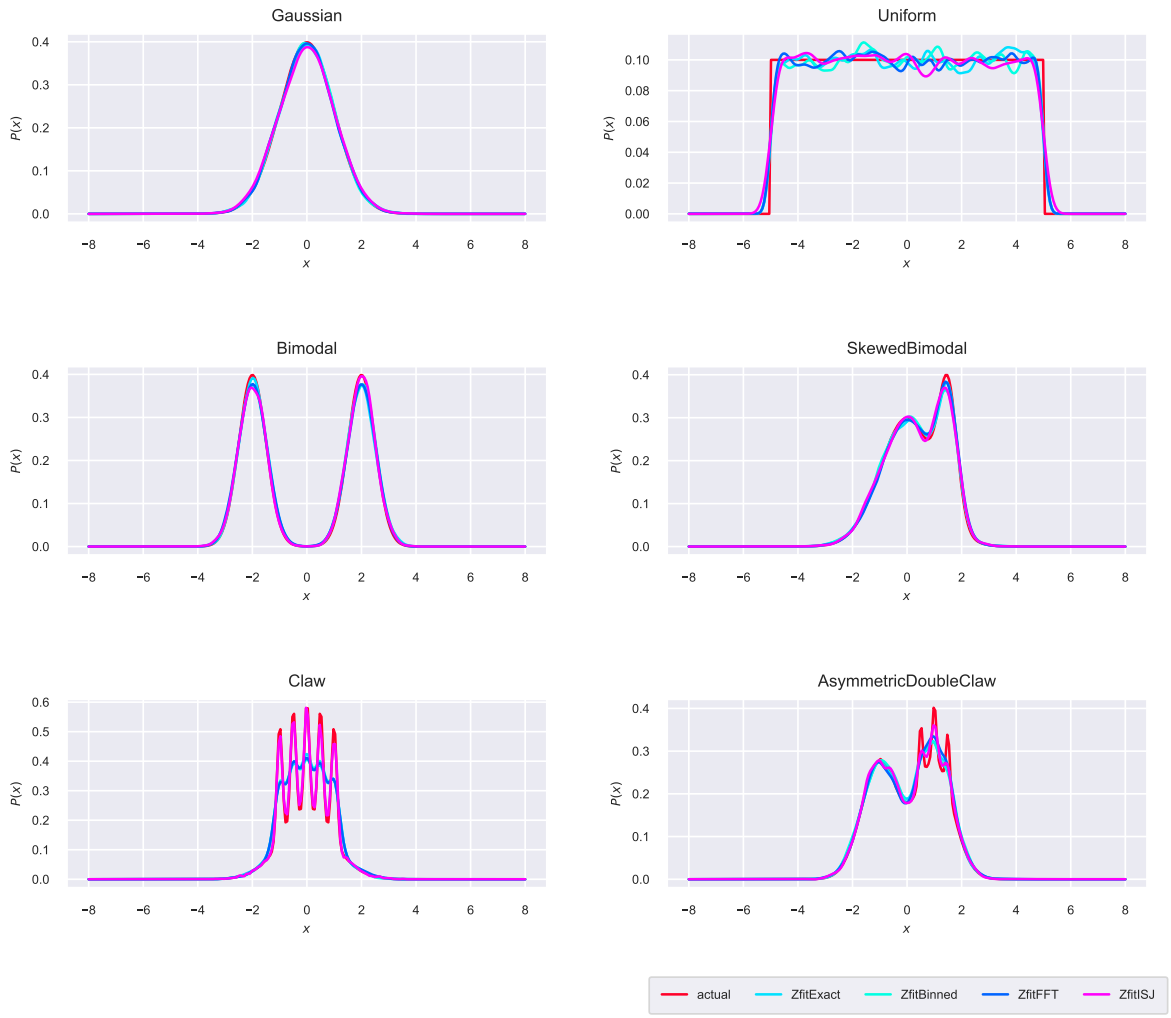

Figure 3: Comparison between the four algorithms 'Exact', 'Binned', 'FFT' and 'ISJ' with $n = 10^4$ sample points

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. Using the ISJ the integrated square error (ISE) is an order of magnitude lower.

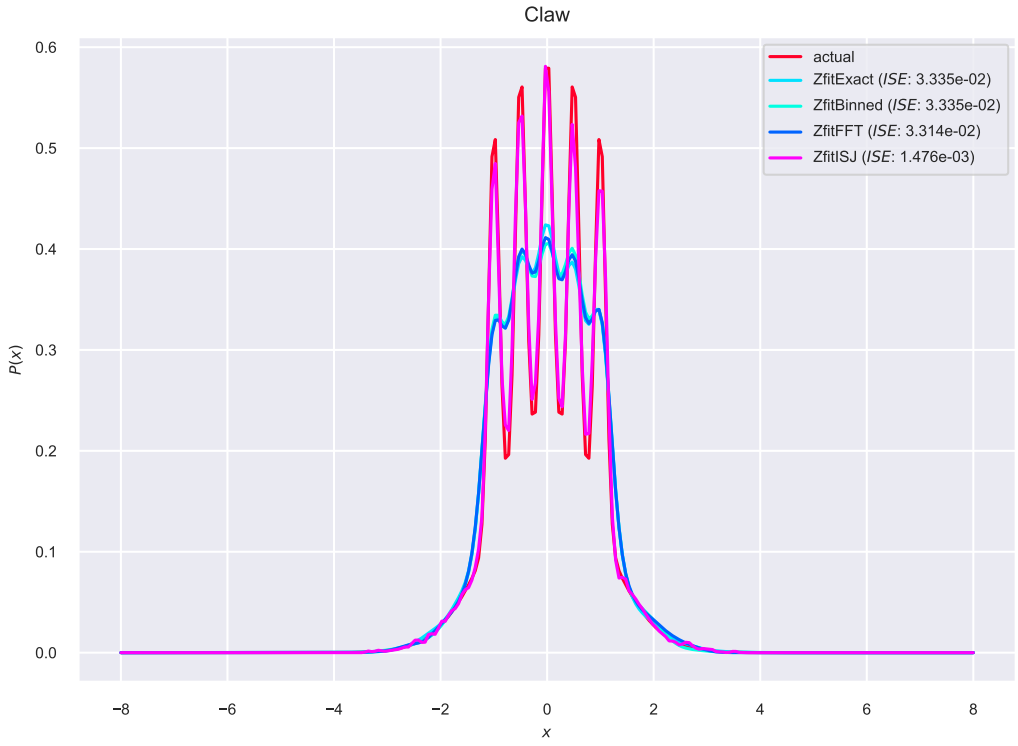

Figure 4: Comparison between the four basic algorithms 'Exact', 'Binned', 'FFT', 'ISJ' with $n = 10^4$ sample points on distribution 'Claw'

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

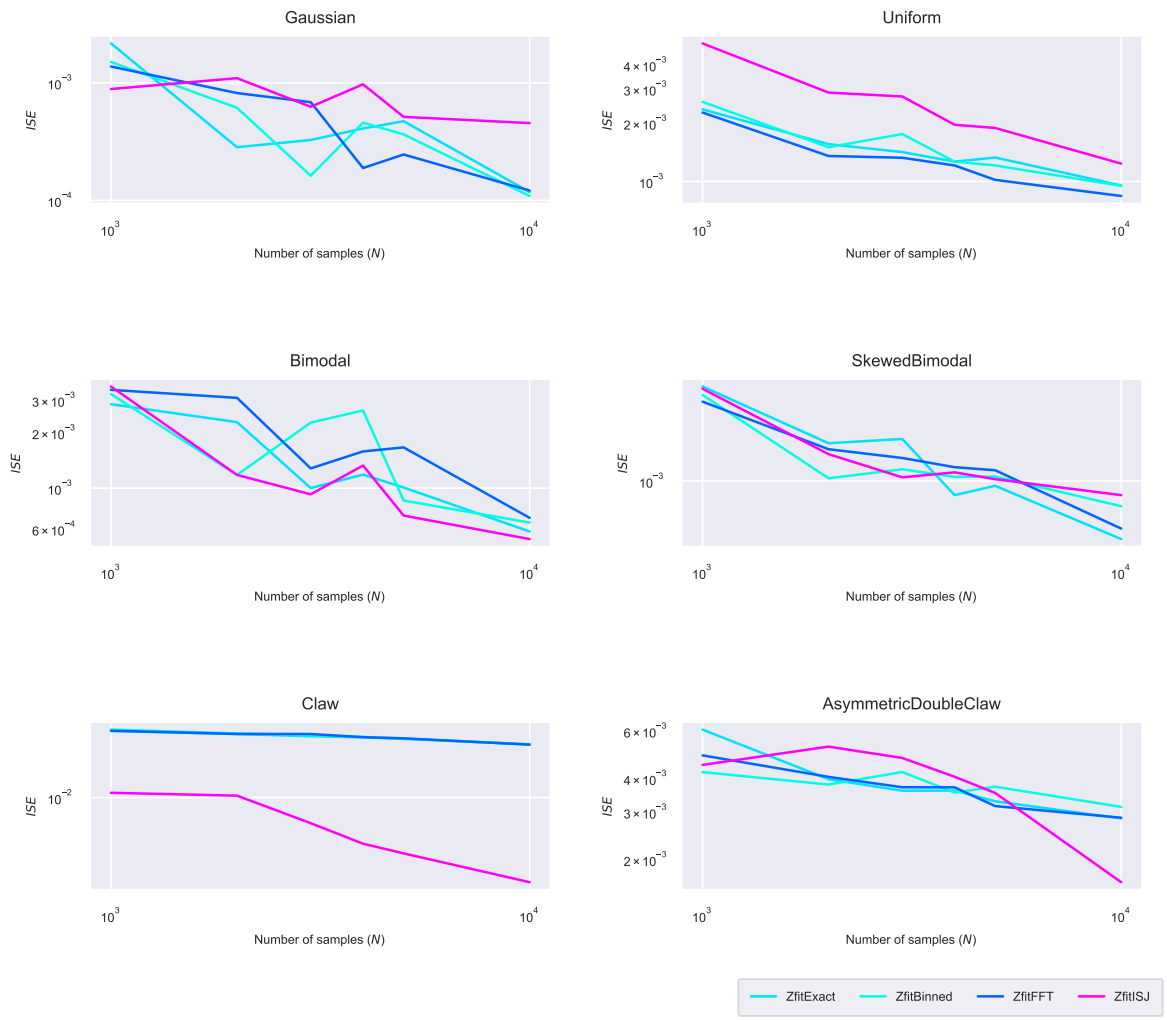

Figure 5: Integrated square errors (ISE) for the four algorithms 'Exact', 'Binned', 'FFT' and 'ISJ'

5.2.2 Runtime

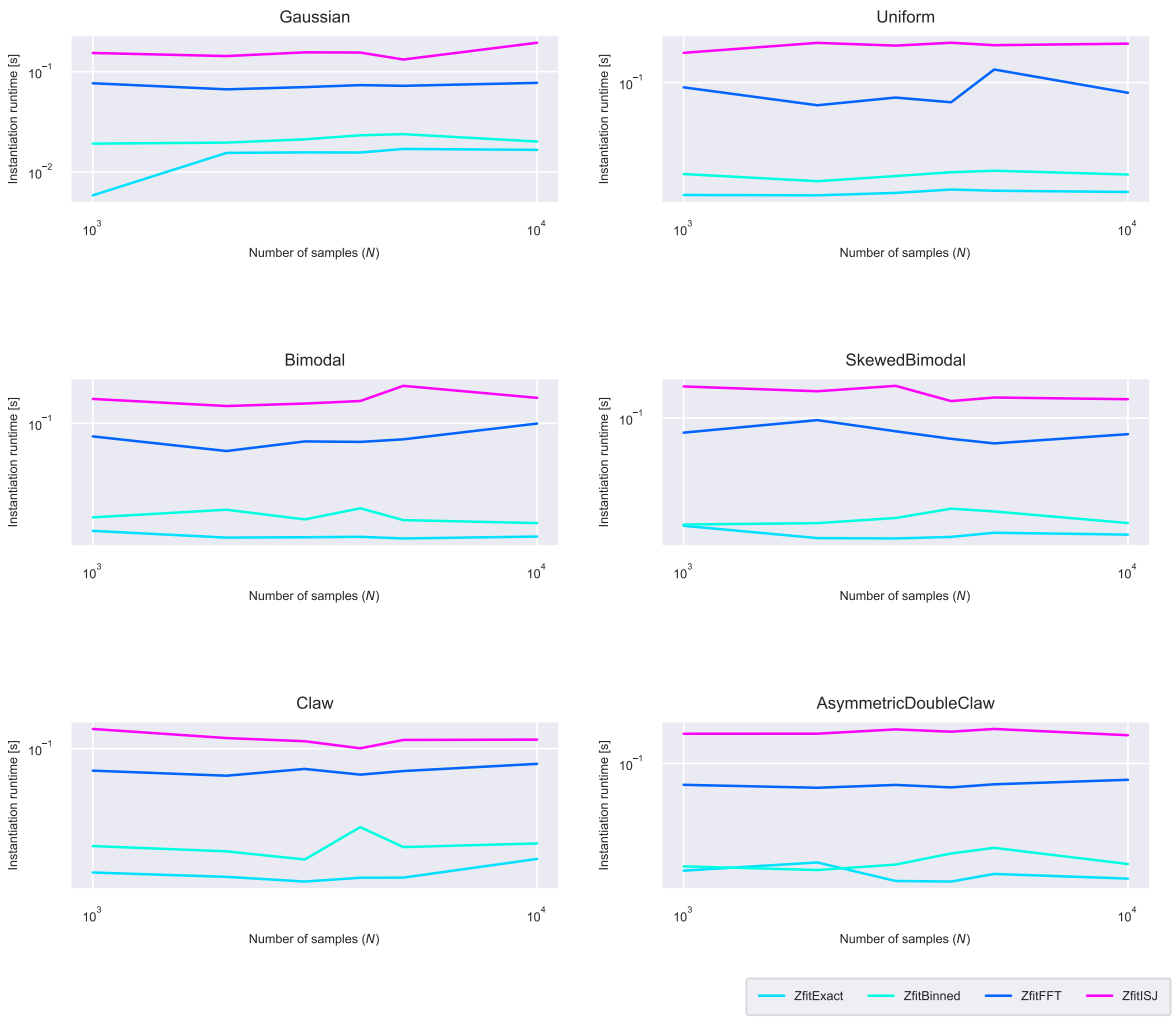

Figure 6: Runtime difference of the instantiaton step between the four algorithms ‘Exact,’ ‘Binned,’ ‘FFT’ and ‘ISJ’

Although the ISJ and the FFT based approach are slower to instantiate, they are significantly faster for larger datasets during the PDF evaluation step.

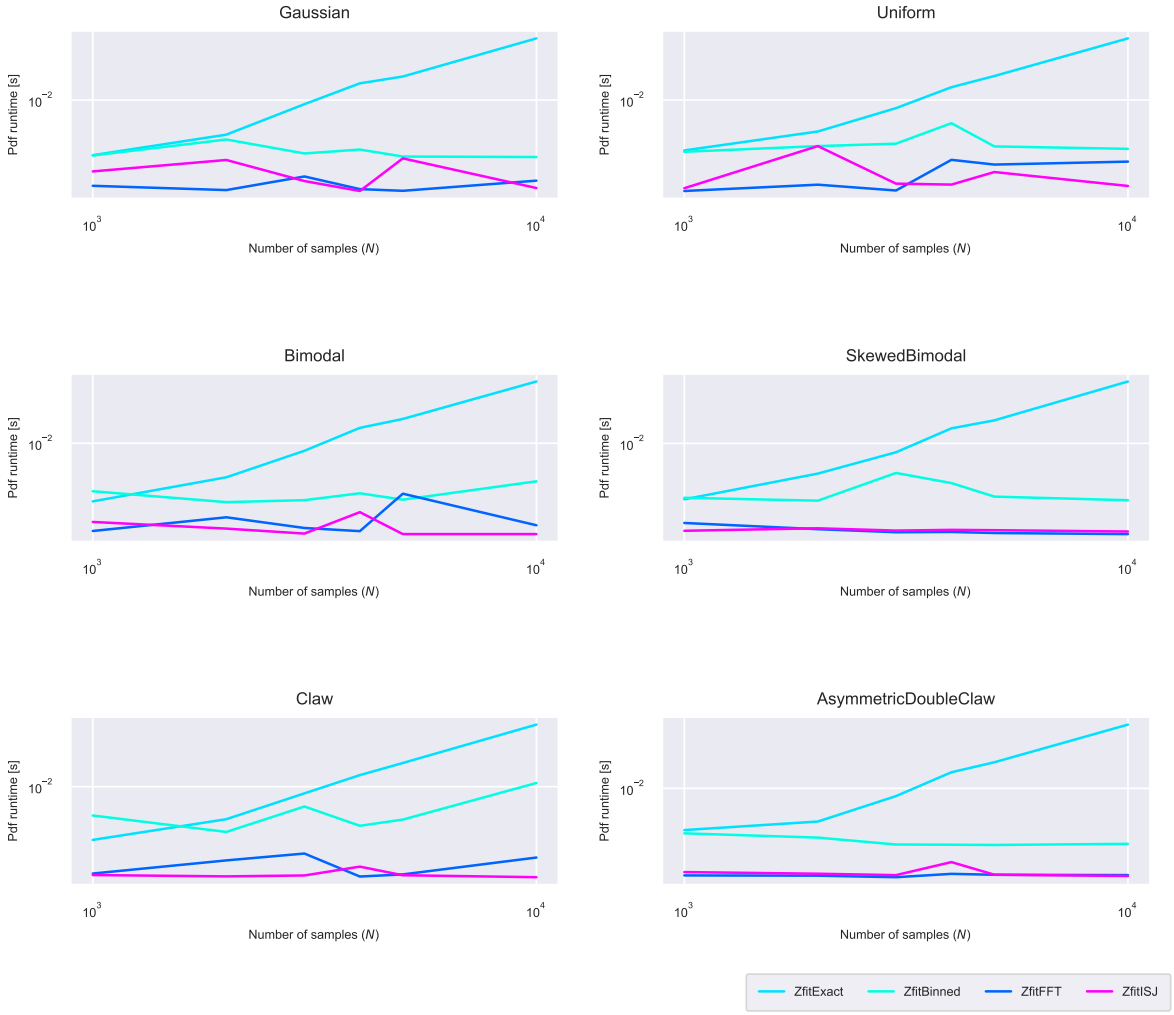

Figure 7: Runtime difference of the evaluation step between the four algorithms 'Exact', 'Binned', 'FFT' and 'ISJ'

5.3 Comparison to KDEpy on CPU

The number of samples per test distribution is in the range of 10^3 - 10^8. Larger datasets can be used, since the exact kernel density estimation is not part of the comparison.

5.3.1 Accuracy

Plotted below are the comparisons for sample size of 10^4.

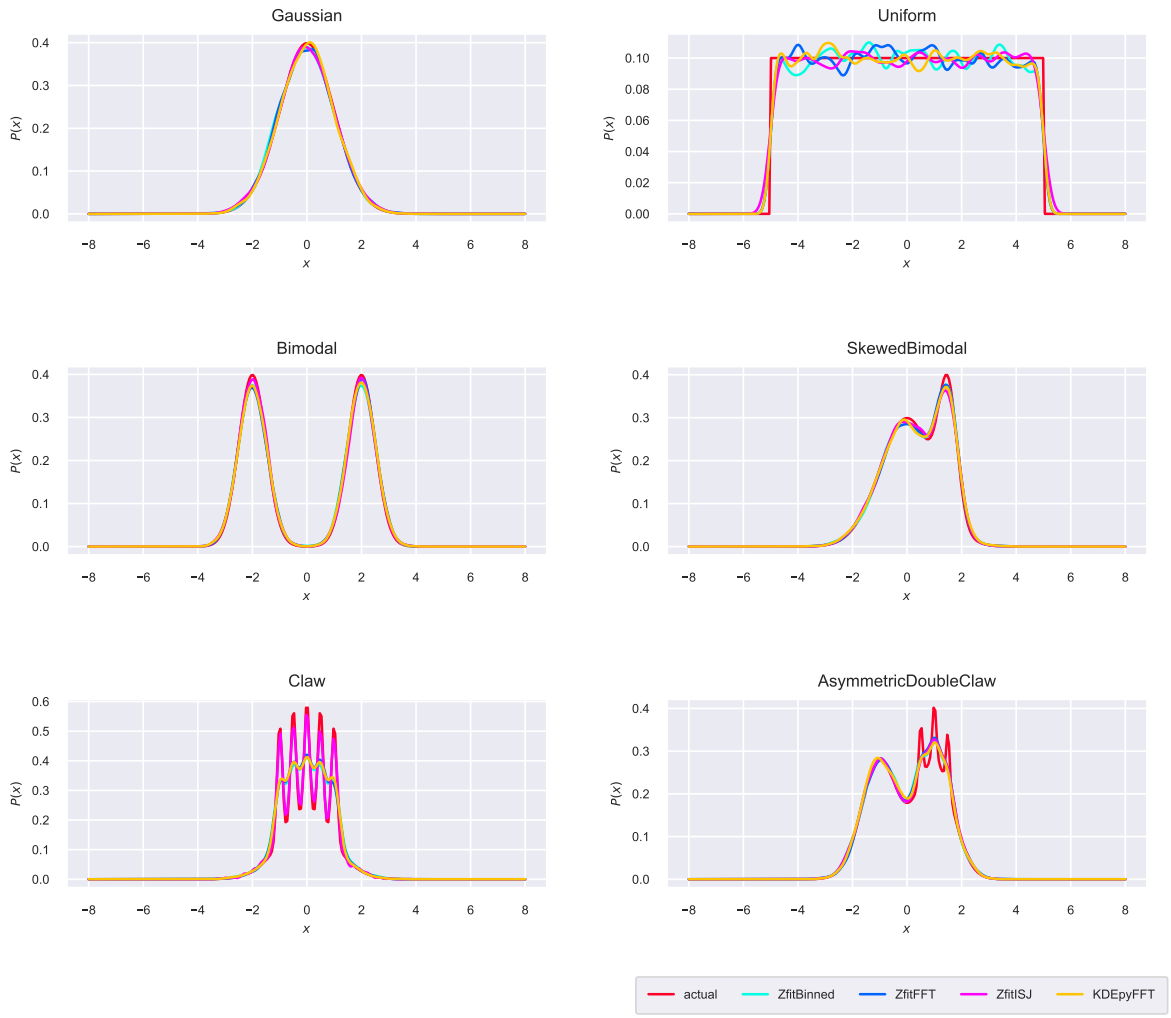

Figure 8: Comparison between the newly proposed algorithms 'Binned', 'FFT', 'ISJ' and the FFT based implementation in KDEpy with $n = 10^4$ sample points

The different methods behave the same as the reference implementation in KDEpy, again with the exception of the ISJ algorithm, which works better for spiky distributions.

The integrated square errors below, we can see that they are in the same order of magnitude for all implementations tested, except for the ISJ method on the Claw distribution. Here the ISE is an order of magnitude lower.

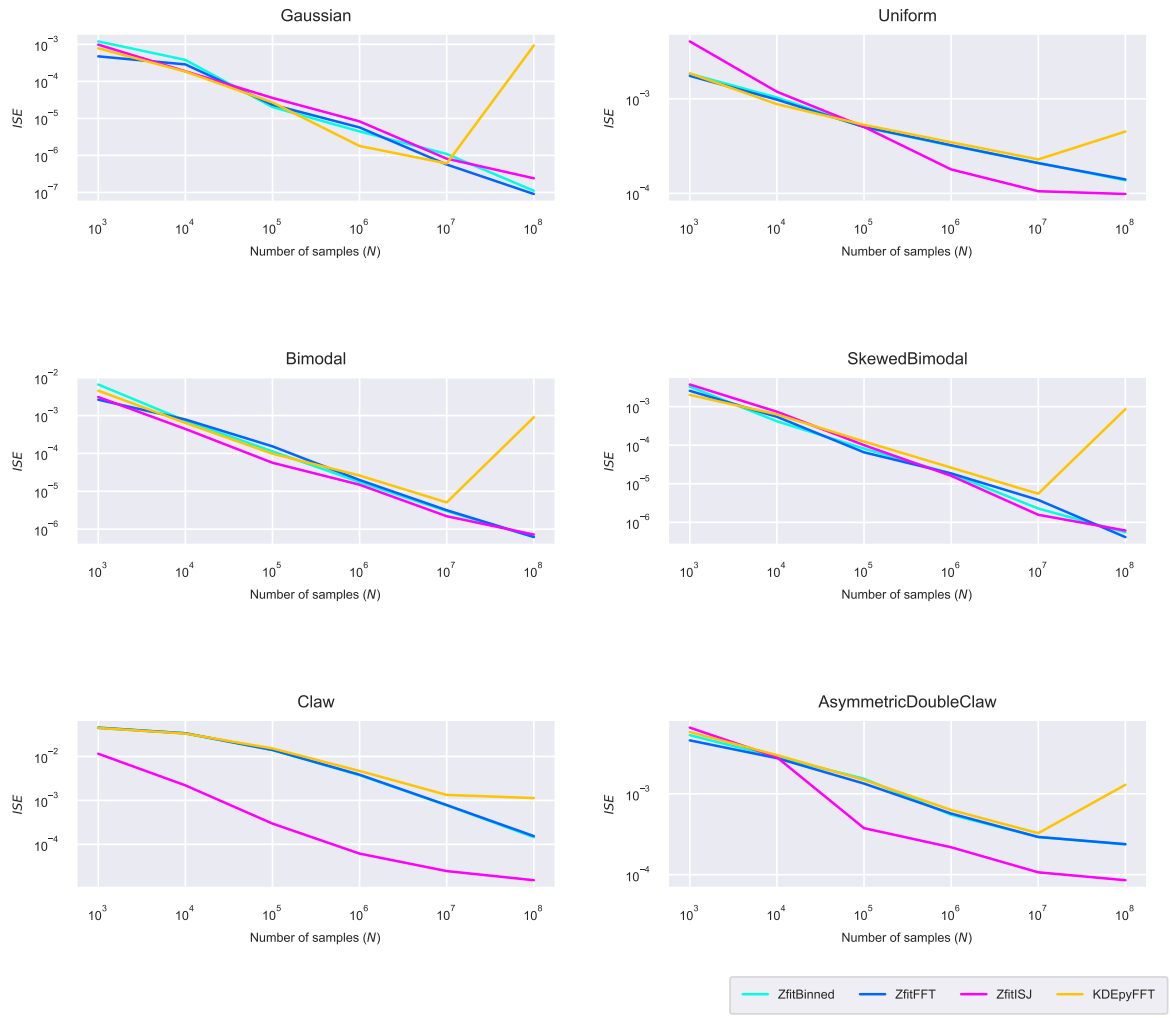

Figure 9: Integrated square errors (ISE) for the newly proposed algorithms 'Binned', 'FFT', 'ISJ' and the FFT based implementation in KDEpy

Additionally it can be shown that 2^{10} bins capture the underlying distributions with high accuracy even for a sample size of 10^8.

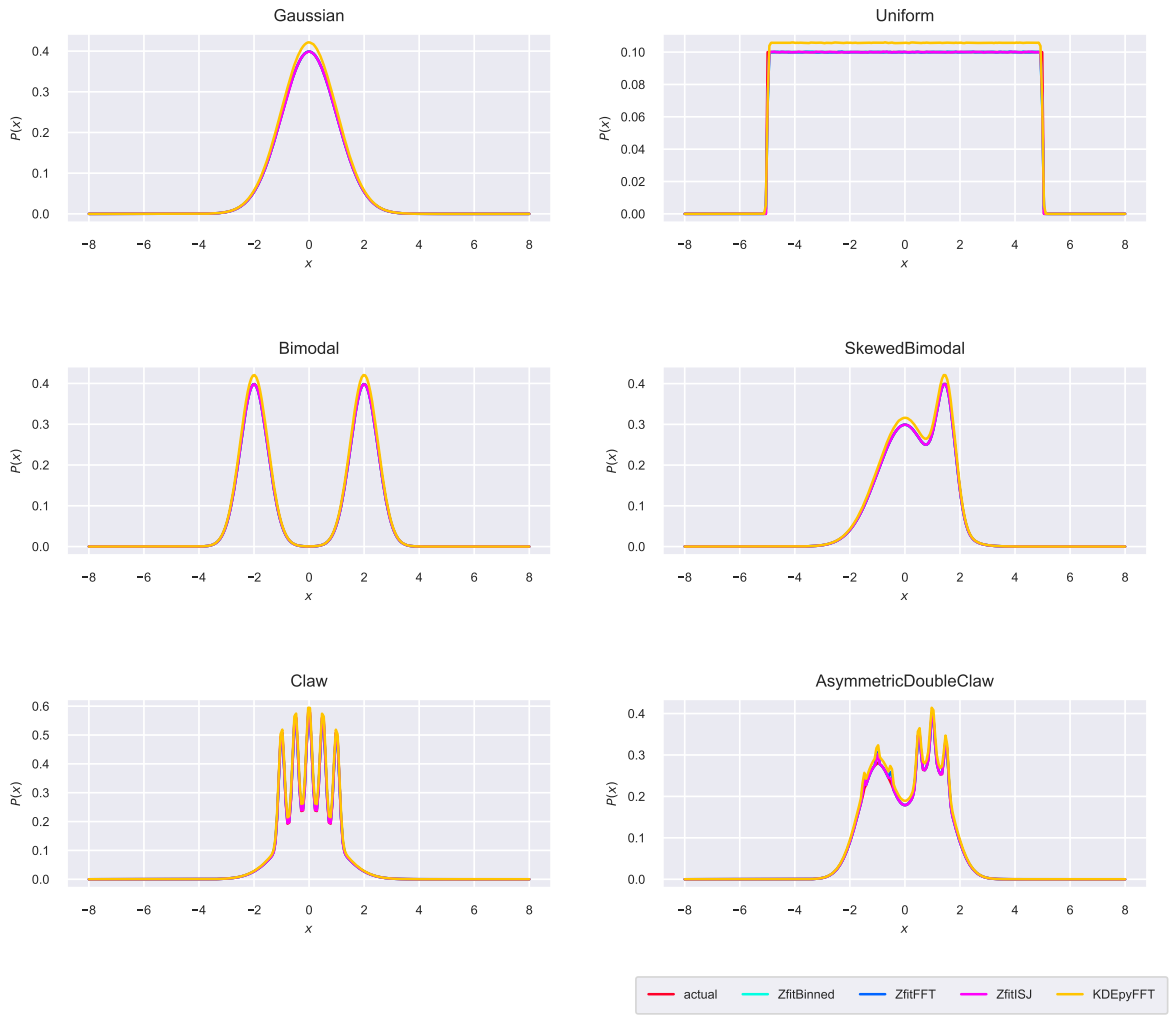

Figure 10: Comparison between the newly proposed algorithms 'Binned', 'FFT', 'ISJ' and the FFT based implementation in KDEpy with $n = 10^8$ sample points

5.3.2 Runtime

During the instantiation step the newly proposed binned, FFT, and ISJ methods are slower than KDEpy's FFT method by one or two orders of magnitude. This is predictable since calculating the TensorFlow graph generates some overhead.

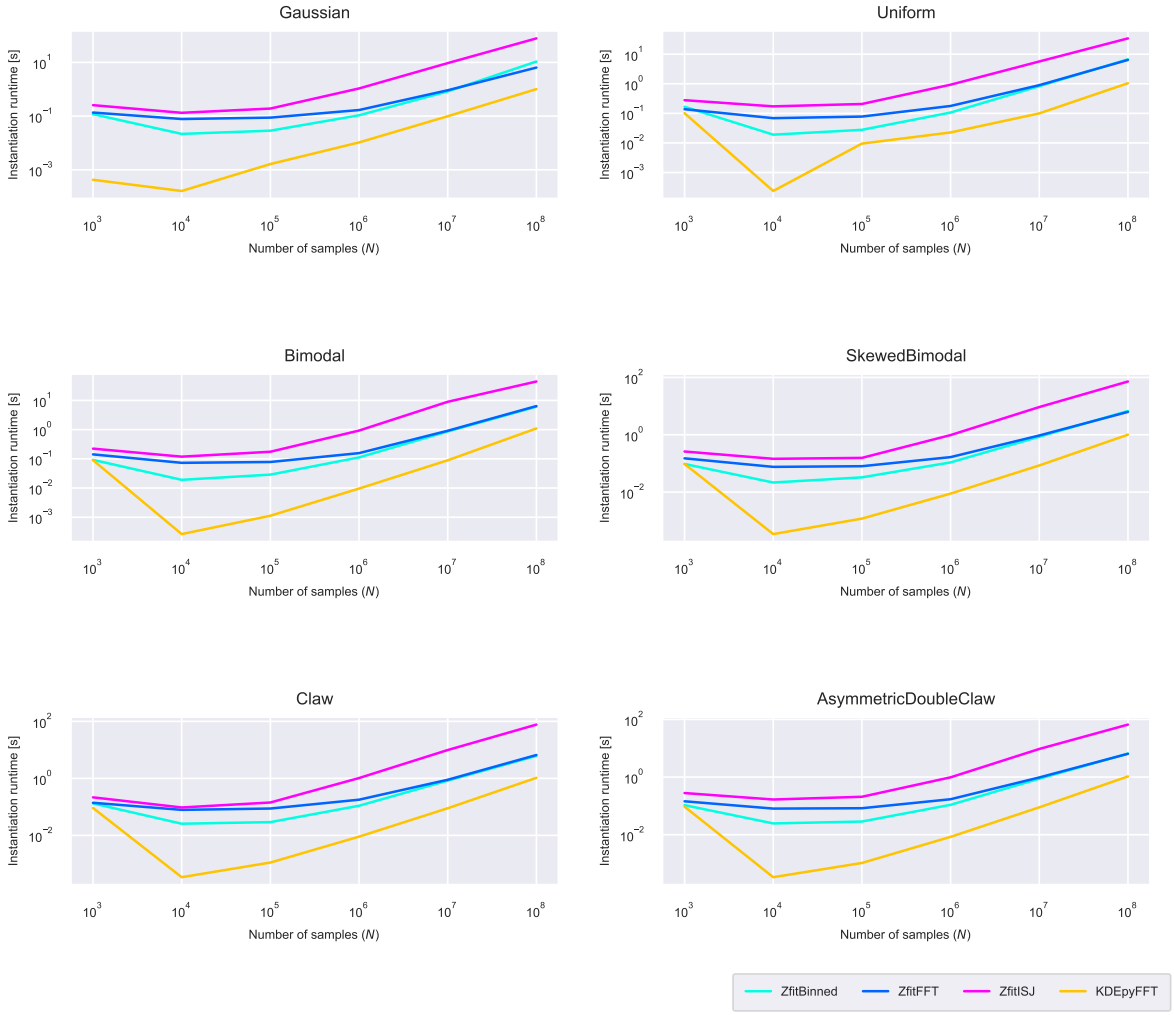

Figure 11: Runtime difference of the instantiaton step between the newly proposed algorithms 'Binned', 'FFT', 'ISJ' and the FFT based implementation in KDEpy

In many practical situtations in high energy physics however, generating the TensorFlow graph and the density distribution has to be done only once and the PDF is evaluated repeatedly. Therefore in such cases the PDF evaluation step is of higher importance. We can see, that once the initial graph is built, evaluating the PDF for different values of x is nearly constant instead increasing exponentially as in the case of KDEpy's FFT method.

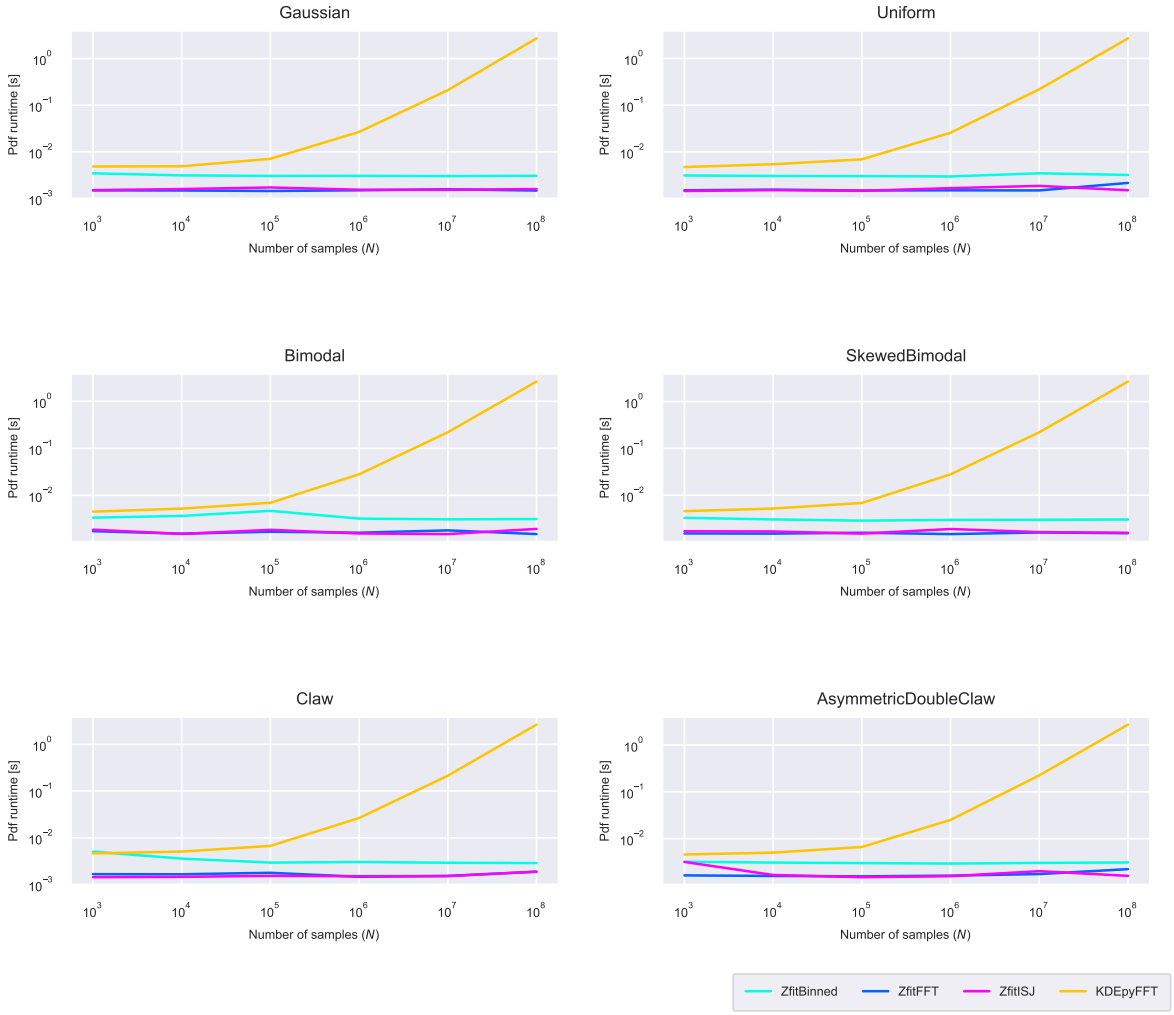

Figure 12: Runtime difference of the evaluation step between the newly proposed algorithms 'Binned', 'FFT', 'ISJ' and the FFT based implementation in KDEpy

5.4 Comparison to KDEpy on GPU

The number of samples per test distribution is again in the range of 10^3 - 10^8. All computations were executed using two Tesla P100 GPU's on the openSUSE Leap operating system.

5.4.1 Accuracy

Plotted below are the comparisons for sample size of 10^4.

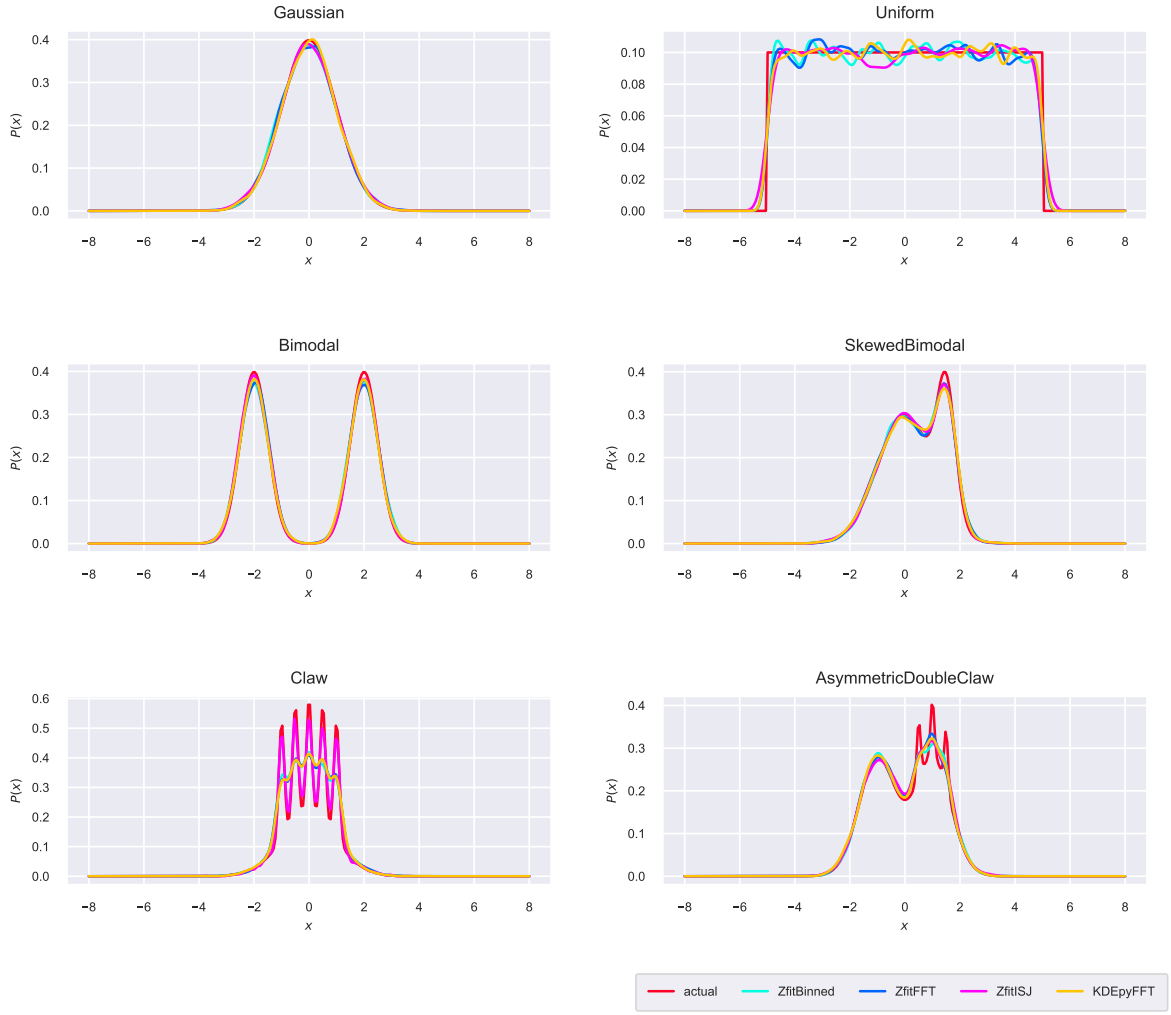

Figure 13: Comparison between the newly proposed algorithms 'Binned', 'FFT', 'ISJ' and the FFT based implementation in KDEpy with $n = 10^4$ sample points (run on GPU)

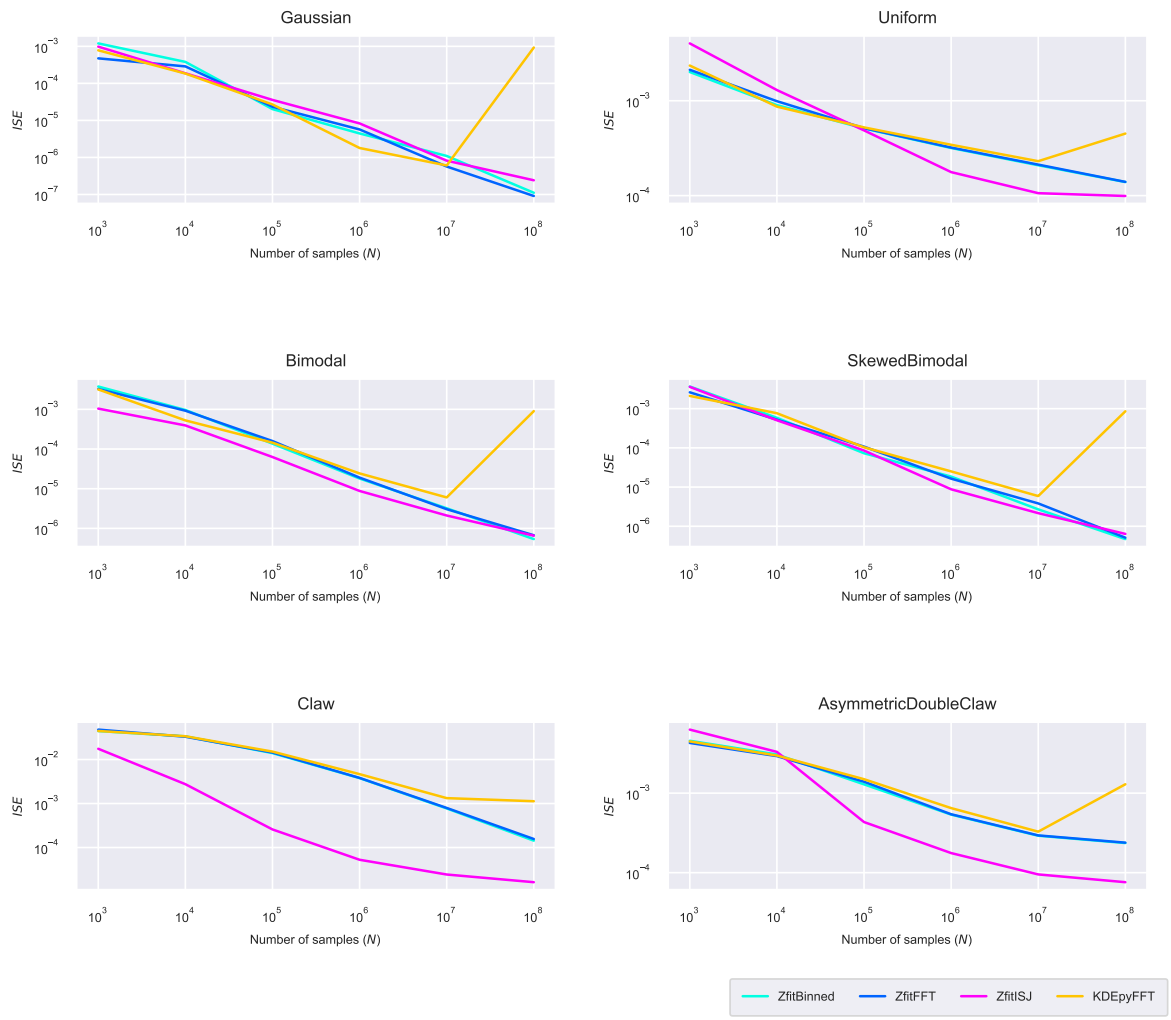

Figure 14: Integrated square errors (ISE) for the newly proposed algorithms 'Binned', 'FFT', 'ISJ' and the FFT based implementation in KDEpy with $n = 10^4$ sample points (run on GPU)

Looking at the integrated square errors, we see the same behavior as for the comparison on CPU.

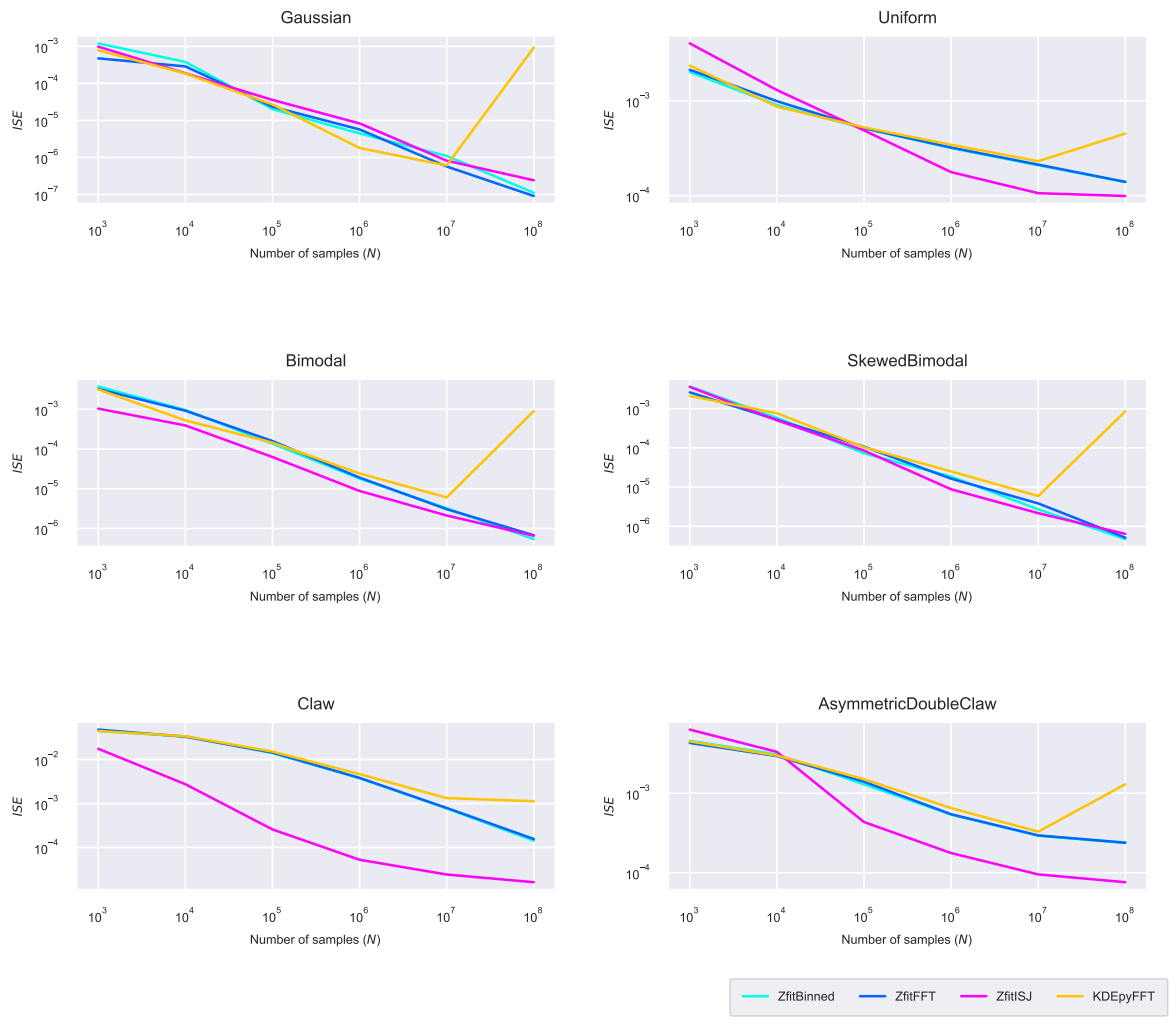

Figure 15: Integrated square errors (ISE) for the newly proposed algorithms 'Binned', 'FFT', 'ISJ' and the FFT based implementation in KDEpy

For larger datasets ($n \geq 10^8$) the accuracy of all the newly proposed methods is higher than for KDEpy's FFT method.

5.4.2 Runtime

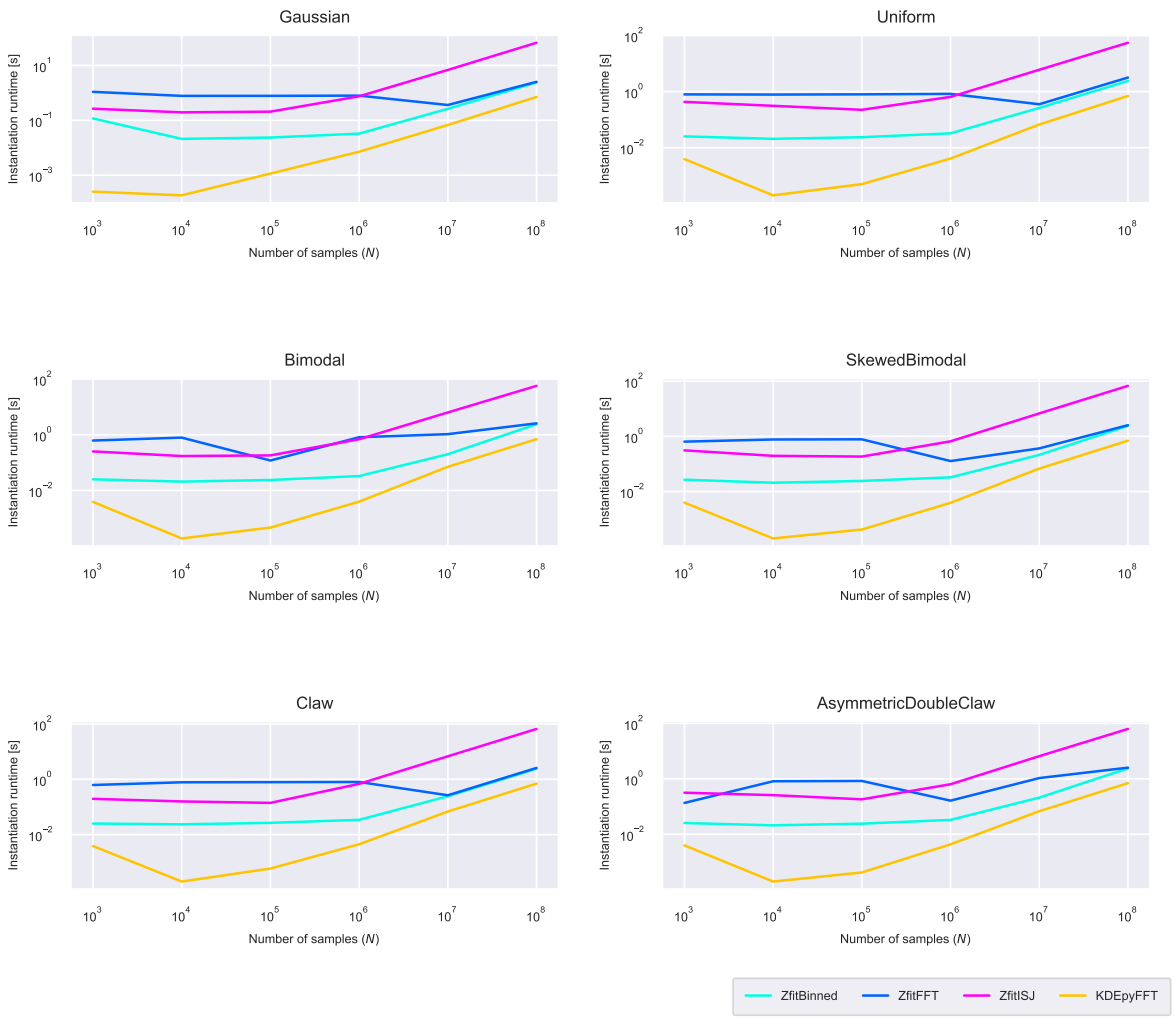

Figure 16: Runtime difference of the instantiaton step between the newly proposed algorithms 'Binned,' 'FFT,' 'ISJ' and the FFT based implementation in KDEpy (run on GPU)

The instantiation of the newly proposed implementations runs faster on the GPU than the CPU. This is no surprise as many operations in TensorFlow benefit from the parallel processing on the GPU. For a high number of sample points the newly proposed binned as well as the newly proposed FFT implementation are instantiated nearly as fast as KDEpy's FFT implementation if run on a GPU.

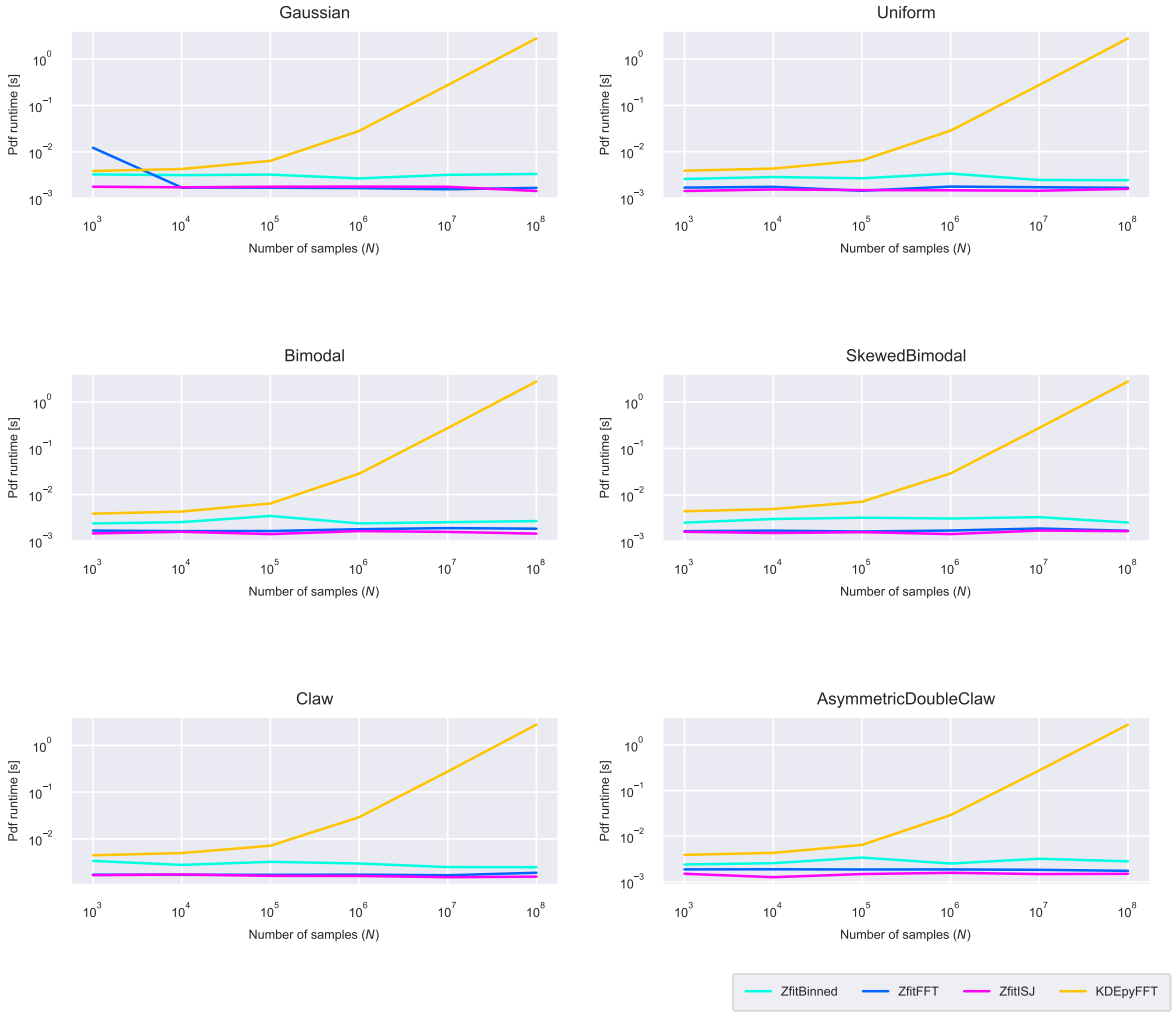

Figure 17: Runtime difference of the evaluation step between the newly proposed algorithms 'Binned', 'FFT', 'ISJ' and the FFT based implementation in KDEpy (run on GPU)

The runtime of the PDF evaluation step does not differ much from the one seen on the CPU. All new methods are evaluated in near constant time.

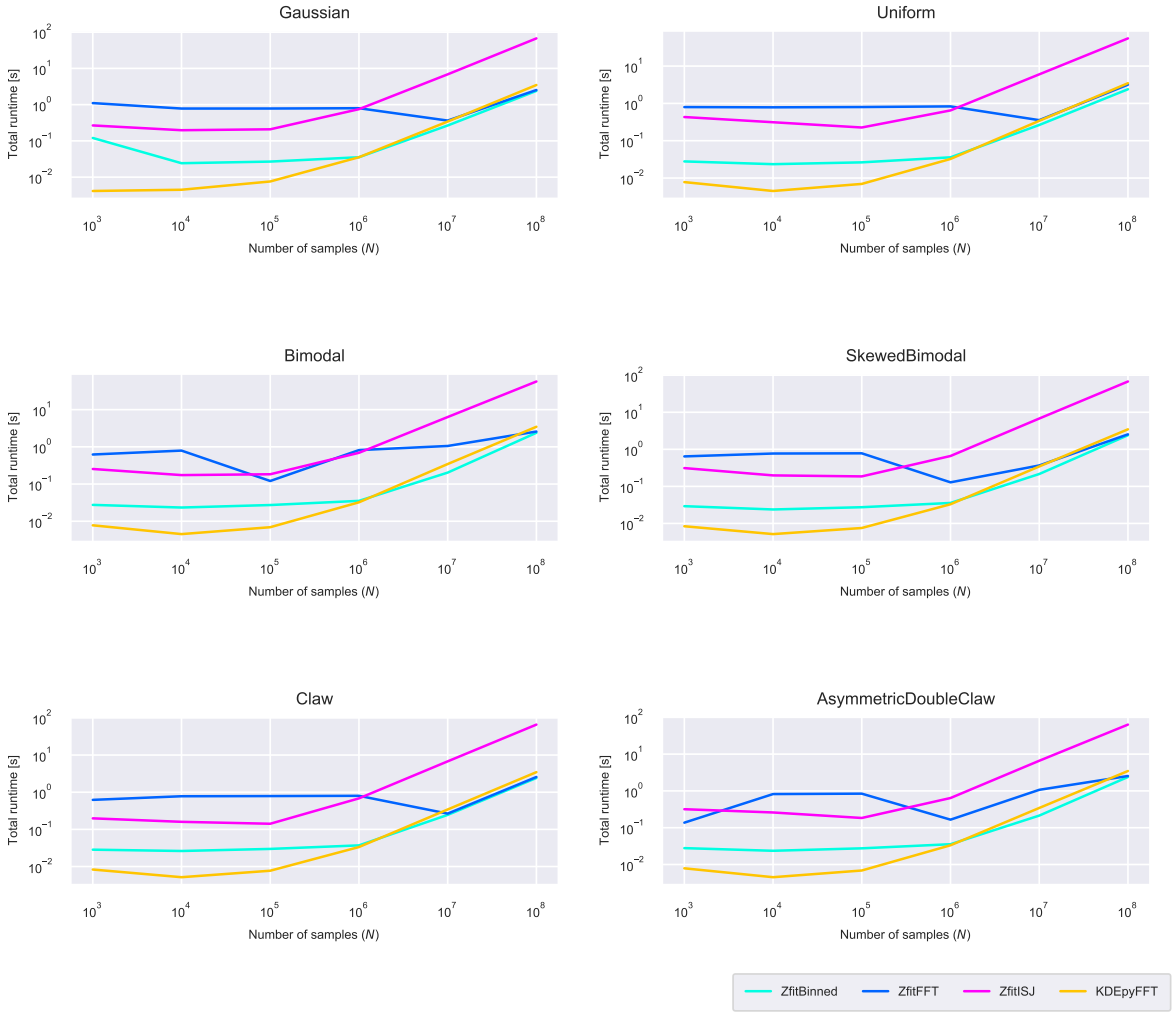

Figure 18: Runtime difference of the total calculation (instantiation and evaluation step) between the newly proposed algorithms 'Binned', 'FFT', 'ISJ' and the FFT based implementation in KDEpy (run on GPU)

For larger datasets ($n \geq 10^8$) even the total runtime (instantiation and PDF evaluation combined) is faster than KDEpy's FFT method.

6 Summary

The benchmarking has shown that the new implementation can outperform the de-facto state of the art library 'KDEpy' in terms of runtime and accuracy for large datasets ($n \geq 10^8$).

For smaller datasets it provides accuracy of the same order of magnitude and may be favorable (faster) in cases where the PDF is built once but evaluated repeatedly.

Both cases are important in high energy physics.

Additionally it has the benefit of allowing any distribution of the loc-scale family that follows the Distribution contract of TensorFlow Probability[2] as kernel function. This includes twelve distributions at the moment, namely `Cauchy`, `DoublesidedMaxwell`, `Gumbel`, `Laplace`, `LogLogistic`, `LogNormal`, `Logistic`, `LogitNormal`, `Moyal`, `Normal`, `PoissonLogNormalQuadratureCompound`, `SinhArcsinh`.

The proposed implementation restricts itself to the one-dimensional case as described in section 1.4. So far only KDEpy and Statsmodels implement a multidimensional KDE. Generalization to higher dimensionel kernel density estimation is feasible, although this would require substantially more work due to some different APIs for multi-dimensional data in TensorFlow. In addition one must ensure that the kernels used are multidimensional themselves, which is not the case for many of the TensorFlow Probability distributions specified above.

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 kernel functions	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 and zfit) achieves state-of-the-art accuracy as well as efficiency for large one-dimensional data ($n \geq 10^8$). 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.

7 Appendix

7.1 Source Code

The source code of the newly proposed implementations can be found at https://github.com/AstroViking/tf-kde

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems* (2015).

[2] 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).

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

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

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

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

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

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

[9] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del R'ıo, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, *Array Programming with NumPy*, Nature **585**, 357 (2020).

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

[11] 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.

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

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

[15] D. Hofmeyr, *Fast Exact Evaluation of Univariate Kernel Sums*, IEEE Trans. Pattern Anal. Mach. Intell. 1 (2019).

[16] *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).

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

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

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

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

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

[22] D. P. Hofmeyr, *Fast Kernel Smoothing in R with Applications to Projection Pursuit*, arXiv:2001.02225 [stat] (2020).

List of Tables

List of Figures