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 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 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. If the underlying mechanisms are well understood a so called parametric approach can be used. One can describe the model mathematically and fit it to the experimental data by some goodness-of-fit criterion like log-likelihood or χ^2.

Oftentimes however the underlying mechanisms are too complex to be fully understood analytically and the parametric methods fail. The knowledge of the system is to poor to describe the model as mathematical function.

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. It is not feasible to anticipate features of the distribution one observes experimentally, as it is comprised of many different distributions, which result from all the different physical interactions.

To combat this, so called non-parametric methods are used, which do not need a predefined mathematical model.

The perhaps simplest non-parametric method is the Histogramm. By summing the the data up in discrete bins the underlying parent distribution can be approximated, without needing any knowledge of underlying processes.

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 interpretation 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 function that specifies how much it influences its neighbouring regions as well. The kernel functions themselves are centered at the data point directly, eliminating the need for arbitrary bin positioning[3]. Since KDE still depends on kernel bandwidth (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 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[4]. In addition, by specifying a smooth kernel function, 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].

1.3 Purpose of this thesis

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 third chapter (3) before a new implementation of Kernel Density Estimation based on TensorFlow and zfit is proposed. (4). Starting with a rather simple implementation, multiple improvements from recent papers are integrated to enhance its efficiency. Efficiency and accuracy of the implementation are then tested by comparing it to other implementations in pure Python(5).

1.4 Univariate case

The proposed implementation is 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

An exact Kernel Density Estimation 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, which defines the range and size of influence of a single data point over the estimation. Most typically a simple Gaussian distribution is used as kernel function. The

wider the kernel function is, the farther is the influence of a single data point, this is characterized by the bandwidth paramter h.

The computational complexity of the 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).

There exist several methods to combat this complexity.

2.2 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 lowers the computational complexity down to $\mathcal{O}(N \cdot m)$.

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 given a million sample points[7].

As described in the extensive overview by Artur Gramacki[8] 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 sample point's weight ($w_k = 1$ for an unweighted histogram) either to the grid point (bin) left or right of itself.

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

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).

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

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

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

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 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 $\mathcal{O}(\log N \cdot N)$.[8]

Using the equation (4) 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

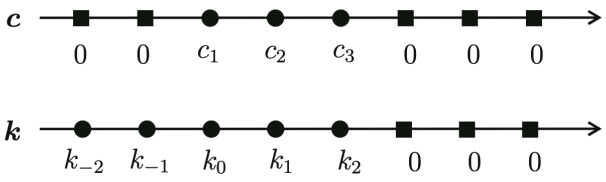

Figure 1: Vectors $\vec{c}$ and $\vec{k}$

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.[9] 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'[10] 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($MISE$)

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

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 appoximation of the $MISE$ of the bandwidth is then given by[11]

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

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.

Their proposed plug-in method works as follows:

1. Compute $\|\hat{f}^{(l+2)}\|^2$ by assuming that f is the normal pdf with mean and variance estimated from the sample data
2. Using $\|\hat{f}^{(l+2)}\|^2$ compute h_{l+1}
3. Using h_{l+1} compute $\|\hat{f}^{(l+1)}\|^2$
4. Repeat steps 2 and 3 to compute h^l, $\|\hat{f}^{(l)}\|^2$, h^{l-1}, $\cdots$ and so on until $\|\hat{f}^{(2)}\|^2$ is calculated
5. Use $\|\hat{f}^{(2)}\|^2$ 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 $\|\hat{f}^{(l+2)}\|^2$. 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[9]. 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

$$\begin{aligned} &h_{AMISE} = \xi h_1 = \xi\gamma^{[1]}\left(h_2\right) = \xi\gamma^{[2]}\left(h_3\right) = \cdots = \xi\gamma^{[l]}\left(h_{l+1}\right) \\ &\xi = \left(\tfrac{6\sqrt{2}-3}{7}\right)^{2/5} \approx 0.90 \end{aligned} \tag{11}$$

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

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

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.

3 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`[12], Statsmodels' `KDEUnivariate`[13] Scikit-learn's `KernelDensity` package[14] as well as KDEpy by Tommy Odland[15].

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[16] 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[15] 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[9].

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

Table 1: Comparison between KDE implementations by Tommy Odland[7] (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 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

4.1 Exact Kernel Density Estimation

The implementation of an exact Kernel Density Estimation in TensorFlow is straightforward. As described in the original Tensorflow Probability Paper[17], 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 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[18] 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 data points, 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. Even if the theoretical computational complexity remains the same.

4.2 Simple and linear Binning

Simple binning is already implemented in TensorFlow in the function `tf.histogram_fixed_width`.

Implementing linear binning efficiently with TensorFlow is a bit tricky since loops should be avoided. However with some inspiration from the KDEpy package[15] this can be done without using loops at all.

First, every data point x_k can be described by an integral part $x_k^{integral}$ (equal to its nearest left grid point number $l = x_k^{integral}$) plus some fractional part $x_k^{fractional}$ (corresponding to the additional distance between grid point g_l and data point x_k).

Then we can solve the linear binning 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

$$c_l = c(g_l) = \sum_{\substack{x_k^{fractional} \in X^{fractional} \\ l = x_k^{integral}}} x_k^{fractional} \cdot w_k + \sum_{\substack{x_k^{fractional} \in X^{fractional} \\ l = x_k^{integral}+1}} (1 - x_k^{fractional}) \cdot w_k \quad (13)$$

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

4.3 Using convolution and the Fast Fourier Transform

In TensorFlow one-dimensional convolutions are efficiently implemented already 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 specified 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.

4.4 Improved Sheather Jones Algorithm

To find the roots for equation (12) Brent's method[18] was used, which is implemented in TensorFlow in the python package `tf_quant_finance`. For the Discrete Cosine Transform `tf.signal.dct` is used.

4.5 Source Code

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

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.[11] 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).

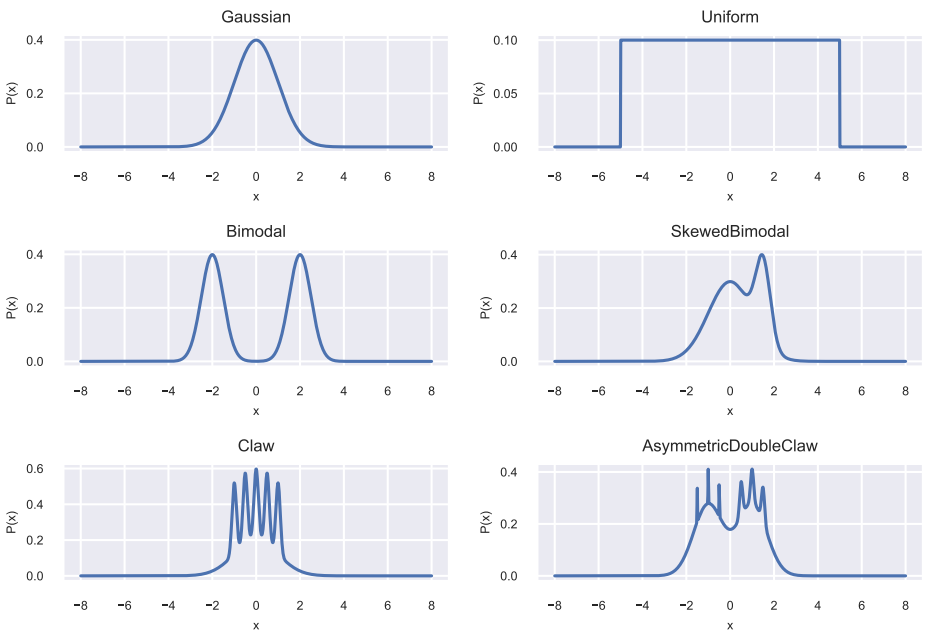

Figure 2: Distributions used for the comparisons

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.

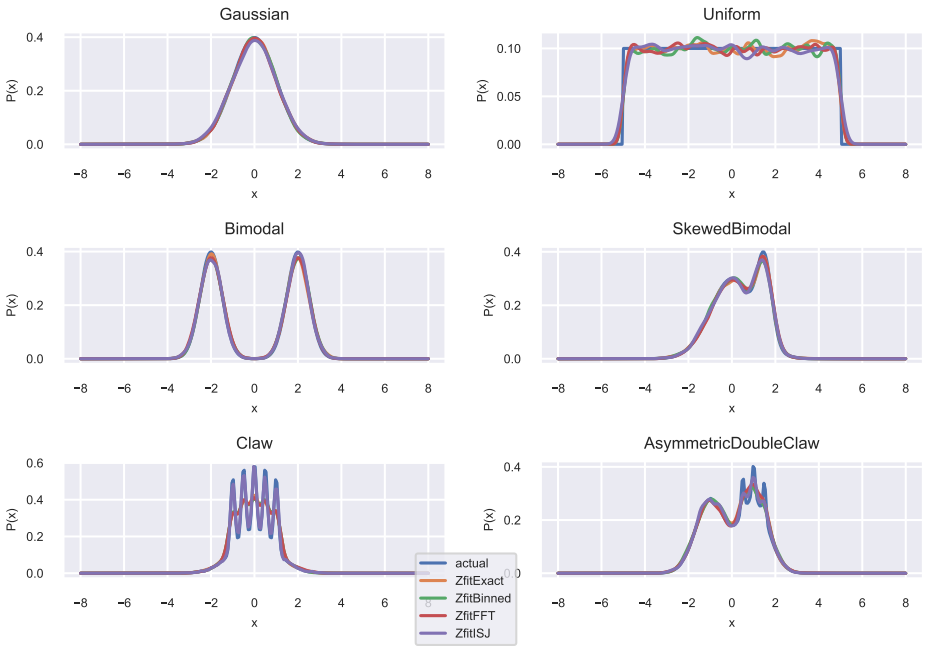

Figure 3: Comparison between the four algorithms 'Exact', 'Binned', 'FFT' and 'ISJ' with $n = 10^4$ sample points

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.

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

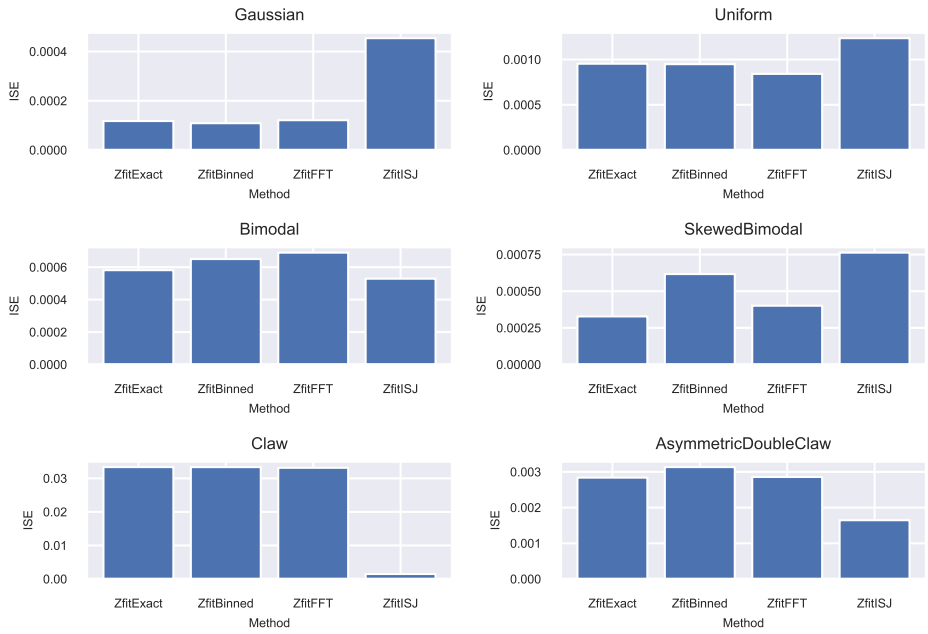

Figure 4: Integrated square errors (ISE) for the four algorithms 'Exact', 'Binned', 'FFT' and 'ISJ' with $n = 10^4$ sample points

5.2.2 Runtime

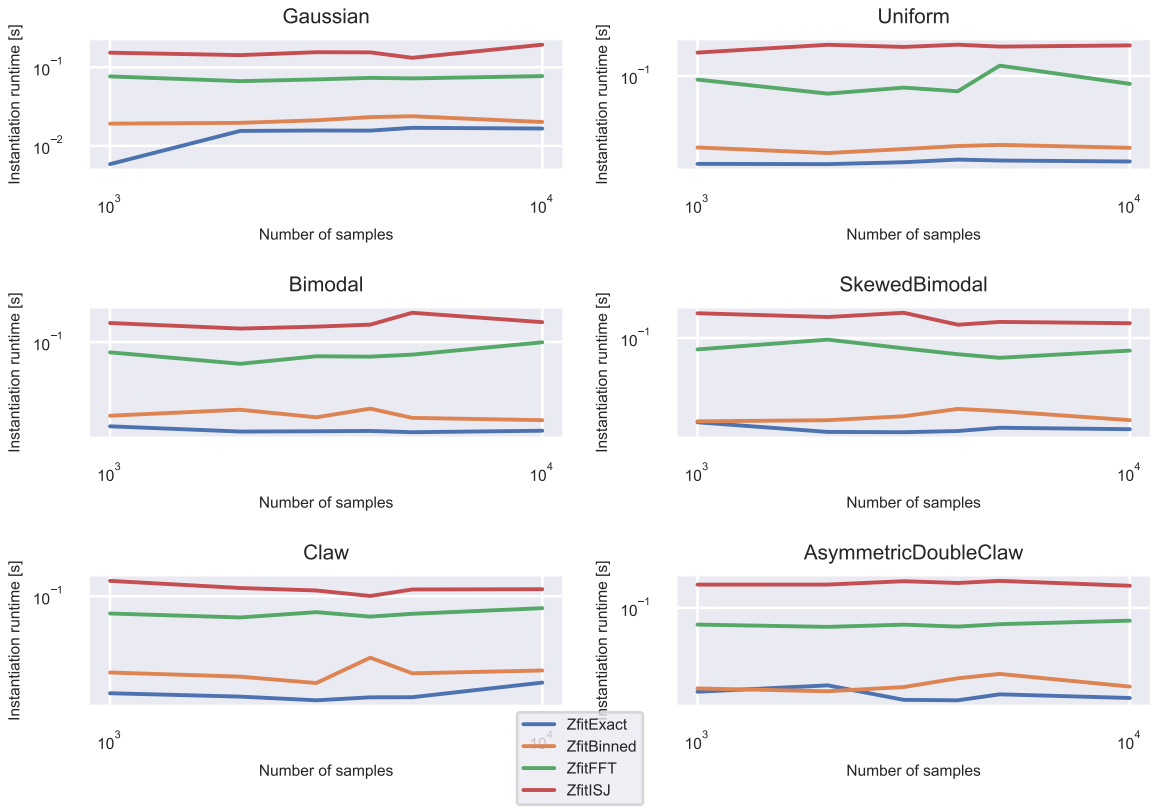

Figure 5: Runtime difference of the instantiaton step between the four algorithms 'Exact,' 'Binned,' 'FFT' and 'ISJ'

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

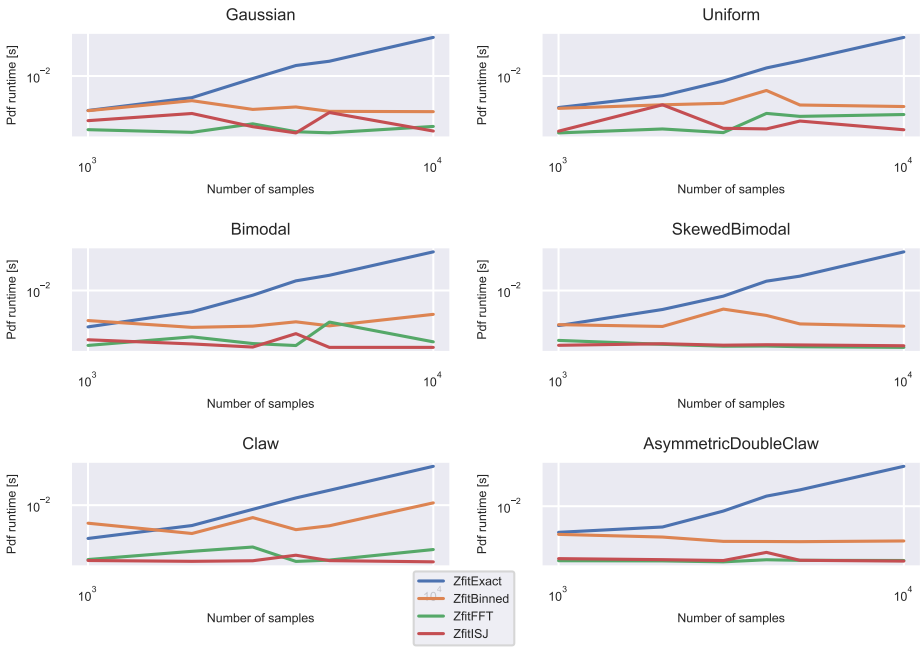

Figure 6: Runtime difference of the evaluation step between the four algorithms 'Exact', 'Binned', 'FFT' and 'ISJ'

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.

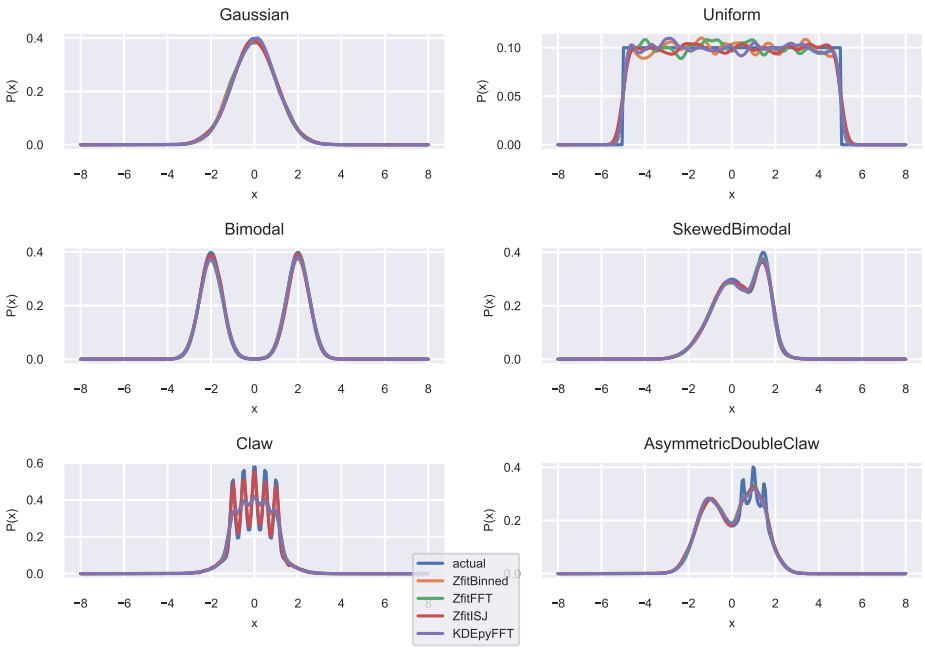

Figure 7: Comparison between the newly proposed algorithms 'Binned', 'FFT', 'ISJ' and the FFT based implementation in KDEpy with $n = 10^4$ sample points

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.

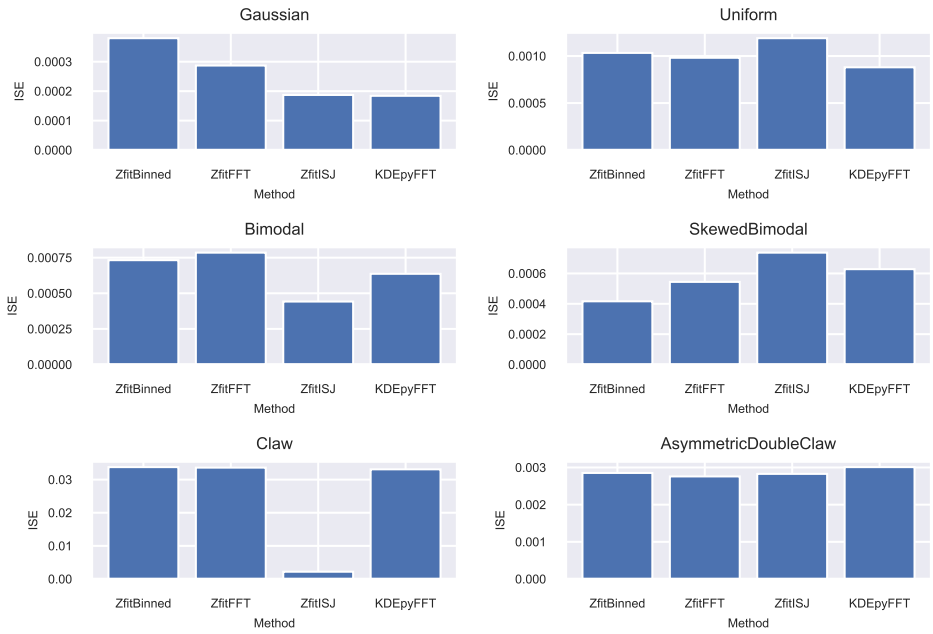

Figure 8: Integrated square errors (ISE) for the newly proposed algorithms 'Binned', 'FFT', 'ISJ' and the FFT based implementation in KDEpy with $n = 10^4$ sample points

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

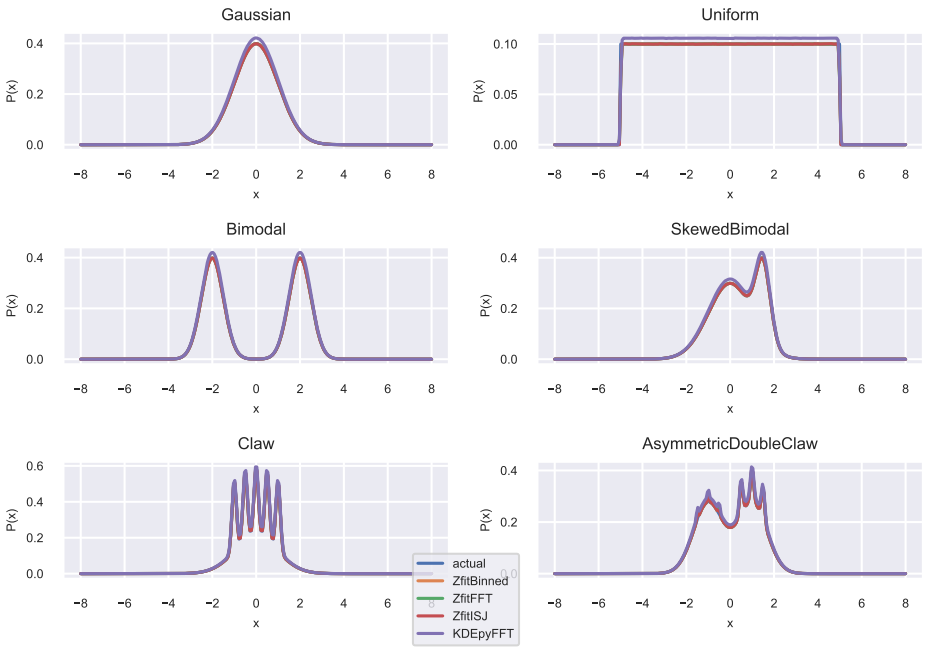

Figure 9: Comparison between the newly proposed algorithms 'Binned', 'FFT', 'ISJ' and the FFT based implementation in KDEpy with $n = 10^8$ sample points

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.

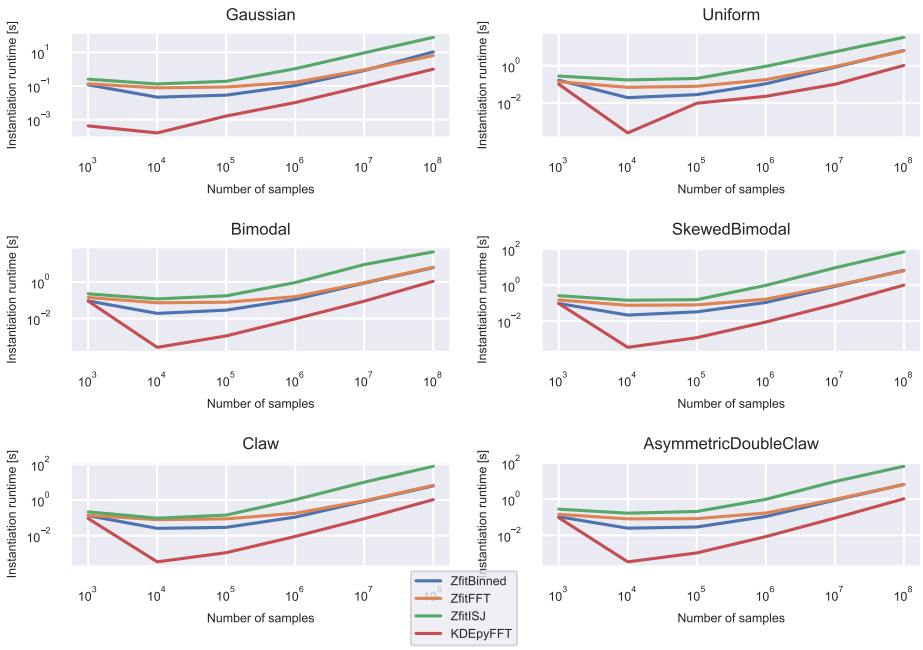

Figure 10: Runtime difference of the instantiaton step between the newly proposed algorithms 'Binned', 'FFT', 'ISJ' and the FFT based implementation in KDEpy

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.

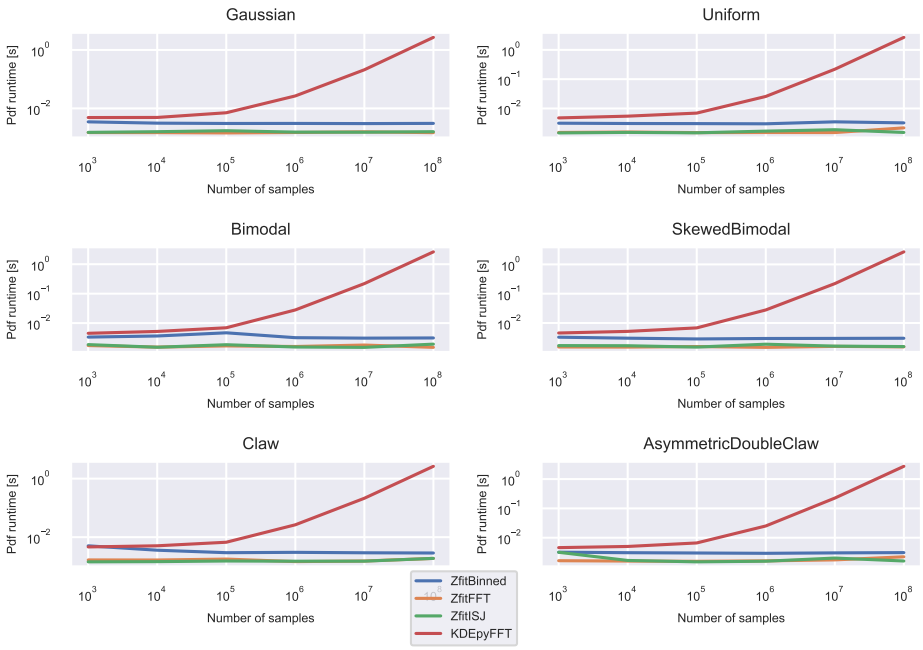

Figure 11: Runtime difference of the evaluation step between the newly proposed algorithms 'Binned', 'FFT', 'ISJ' and the FFT based implementation in KDEpy

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.

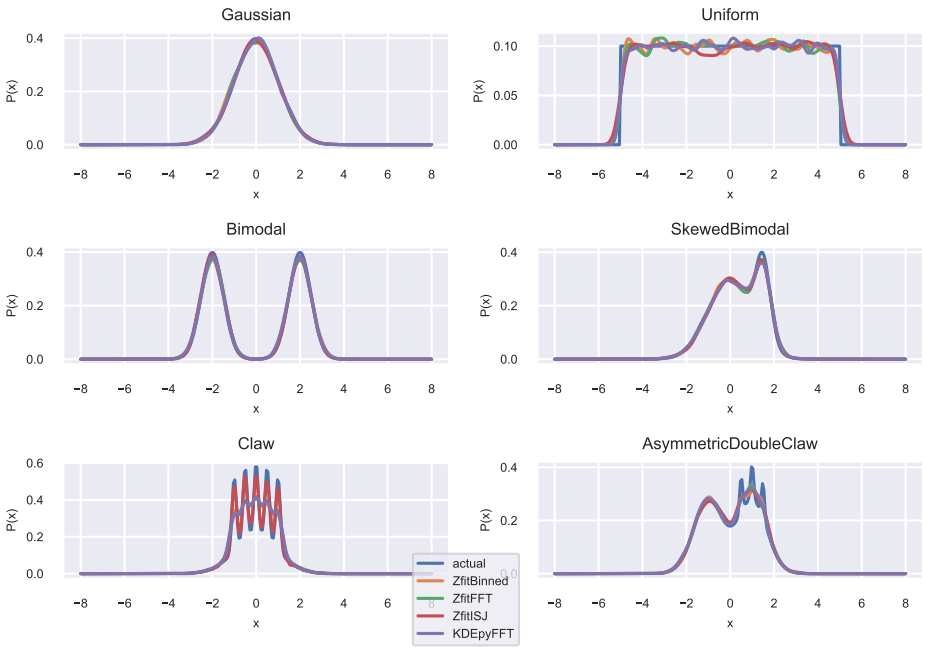

Figure 12: Comparison between the newly proposed algorithms 'Binned', 'FFT', 'ISJ' and the FFT based implementation in KDEpy with $n = 10^4$ sample points (run on GPU)

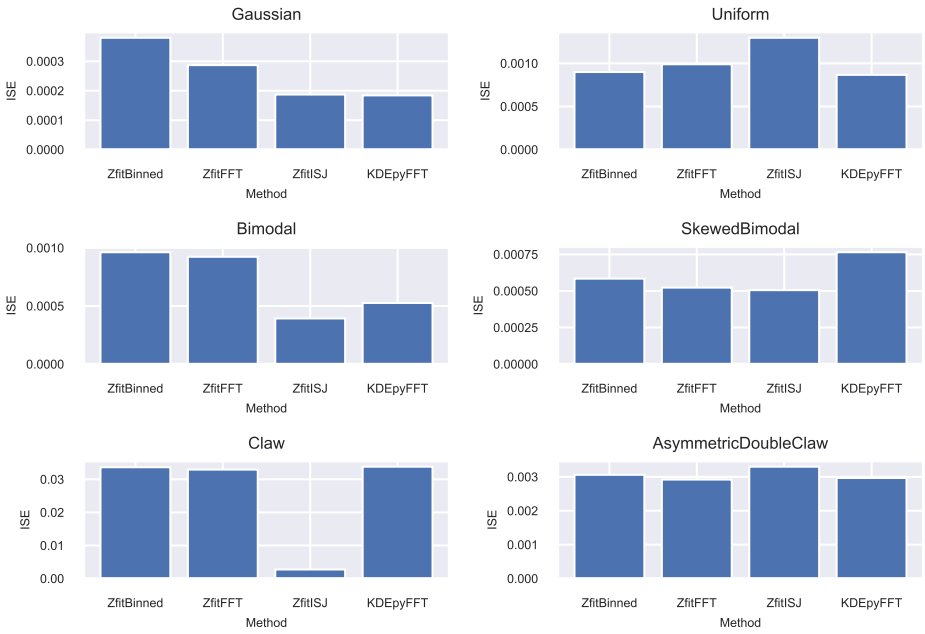

Figure 13: Integrated square errors (ISE) for the newly proposed algorithms 'Binned', 'FFT', 'ISJ' and the FFT based implementation in KDEpy with $n = 10^4$ sample points (run on GPU)

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

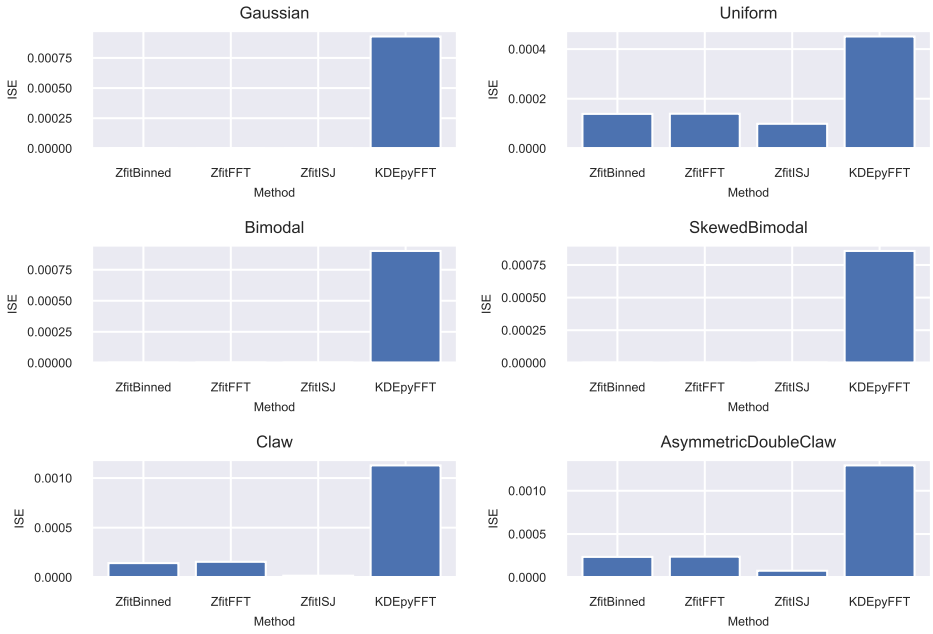

Figure 14: Integrated square errors (ISE) for the newly proposed algorithms 'Binned', 'FFT', 'ISJ' and the FFT based implementation in KDEpy with $n = 10^8$ sample points (run on GPU)

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

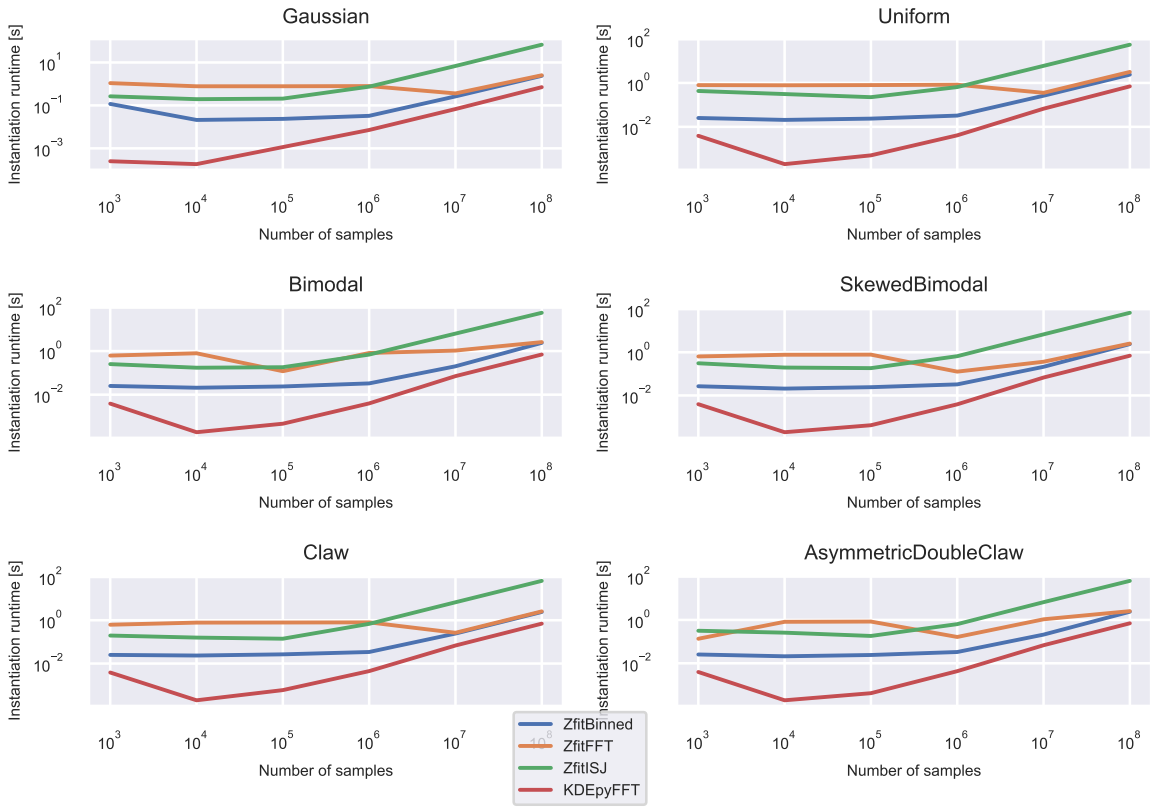

Figure 15: Runtime difference of the instantiaton step between the newly proposed algorithms 'Binned,' 'FFT,' 'ISJ' and the FFT based implementation in KDEpy (run on GPU)

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.

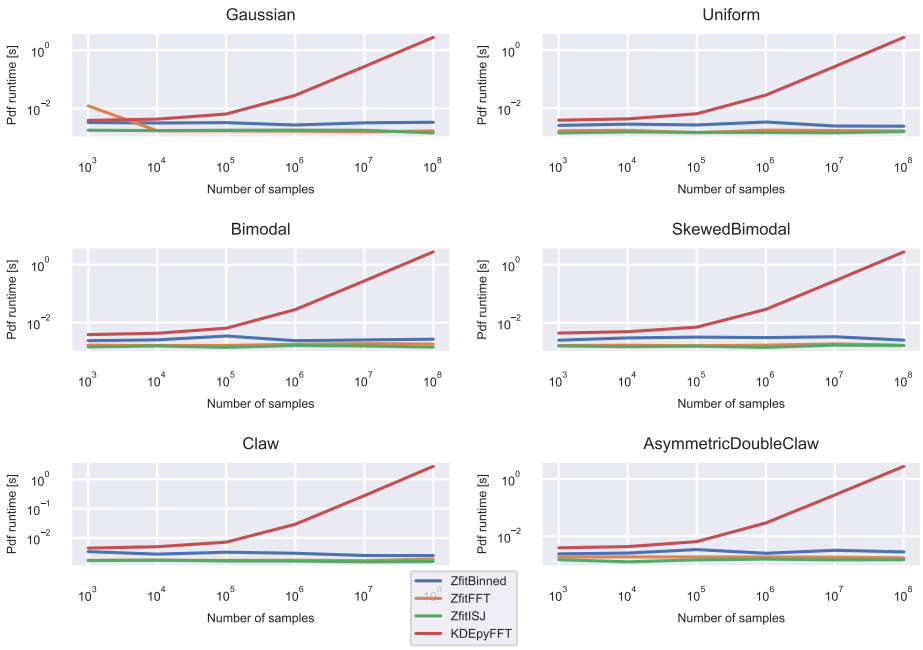

Figure 16: Runtime difference of the evaluation step between the newly proposed algorithms 'Binned', 'FFT', 'ISJ' and the FFT based implementation in KDEpy (run on GPU)

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.

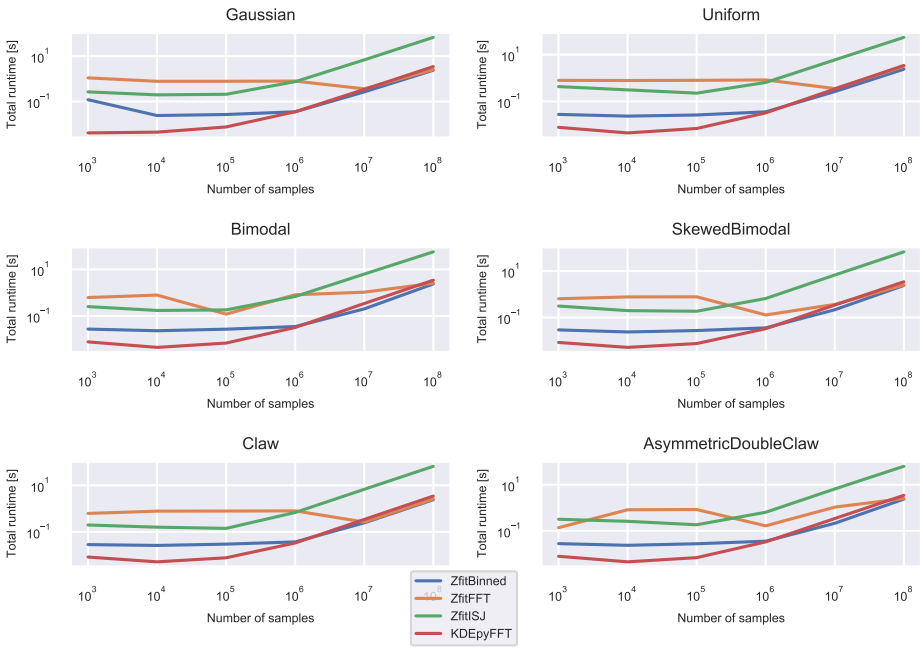

Figure 17: Runtime difference of the total calculation (instantiation and evaluation step) between the newly proposed algorithms 'Binned', 'FFT', 'ISJ' and the FFT based implementation in KDEpy (run on GPU)

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 $> 10^8$.

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[17] 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 di-

mensionel 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.

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

[8] 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.

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

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

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

[12] *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).

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

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

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

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

[17] 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).

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

List of Tables

List of Figures