Universität Zürich[UZH]

Efficiency of Univariate Kernel Density Estimation with TensorFlow

Bachelor Thesis

Author: Marc Steiner

Supervisors: Jonas Eschle

University of Zurich

Abstract

An implementation of a one-dimensional Kernel Density Estimation in TensorFlow is 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 as well to density estimations by smoothed histograms it is shown as competitive and useful in real world applications.

Contents

1 Introduction

1.1 Kernel Density Estimation

In many fields of science and in physics especially, scientists need to estimate the probability density function (PDF) from which a set of data is drawn without having a approximate model of the underlying mechanisms, since they are too complex to be fully understood analytically. So called parametric methods fail, because the knowledge of the system is to poor to design a model, which parameters can then be fitted by some goodness-of-fit criterion like log-likelihood or χ^2. In the particle accelerator at CERN for instance, they record a whooping 25 gigabytes of data per second[1], resulting from the process of many physical interactions that occur almost simultaneously, making it impossible to anticipate features of the distribution one observes.

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

Histograms tend to produce PDFs that are highly dependent on bin width and bin positioning, meaning the interpretion of the data changes by a lot by two arbitrary parameters. KDE circumvents this problem by replacing each data point with a so called kernel that specifies how much it influences its

neighbouring regions as well. The kernels themselves are centered at the data point directly, eliminating the need for arbitrary bin positioning[3]. Since KDE still depends on kernel width (instead of bin width), one might argue that this is not a major improvement. However, upon closer inspection, one finds that the underlying PDF does depend less strongly on the kernel width than histograms on bin width and it is much easier to specify rules for an approximately optimal kernel width than it is to do so for bin width[4]. In addition, by specifying a smooth kernel, one gets a smooth distribution as well, which is often desirable or even expected from theory.

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

1.2 zfit and TensorFlow

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

So far there exists no direct implementation of Kernel Density Estimation in zfit nor TensorFlow, but various implementations in Python. This implementations will be discussed in the next chapter (2) before I propose an implementation of Kernel Density Estimation in TensorFlow (3). Starting with a rather simple implementation, multiple improvements from recent papers are integrated to ameliorate its efficiency. Efficiency and accuracy of the implementation are then tested by comparing it to other implementations in pure Python and simple smoothed histograms (4).

In the last chapter (5) I compare its accurracy and efficiency to smoothed histograms and implementations in pure Python.

2 Current state of the art

…

For humongous data streams (like at CERN 1), Kernel Density Estimation itself needs to be approximated.

…

3 Implementation

The following is an implementation of Kernel Density Estimation using TensorFlow.

4 Comparison

To compare the different implementations I created a simple test distribution comprised of three gaussian, one uniform and one exponential distribution. The distribution is created by using the TensorFlow Probability package and its Mixture Model.

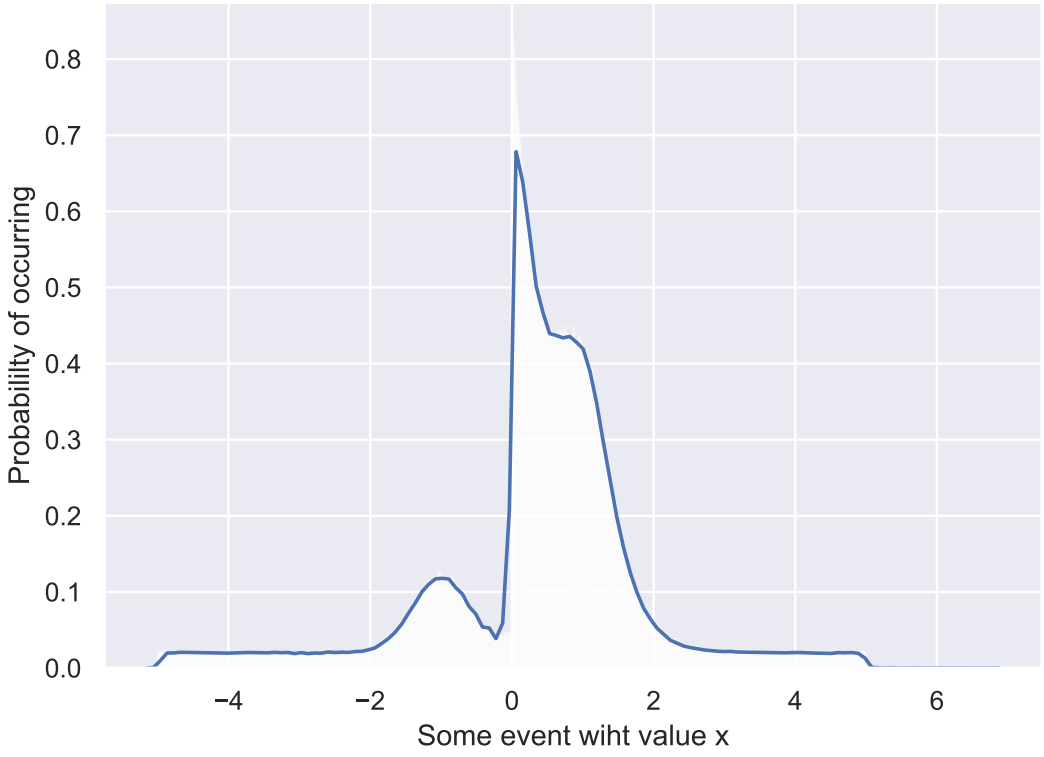
4.1 Generation of Test Distribution

Listing: Test Distribution generation

```
import numpy as np
import tensorflow as tf
import tensorflow_probability as tfp

r_seed = 1978239485

n_datapoints = 1000000

tfd = tfp.distributions

mix_3gauss_1exp_1uni = tfd.Mixture(

  cat=tfd.Categorical(probs=[0.1, 0.2, 0.1, 0.4, 0.2]),

  components=[

    tfd.Normal(loc=-1., scale=0.4),

    tfd.Normal(loc=+1., scale=0.5),

    tfd.Normal(loc=+1., scale=0.3),

    tfd.Exponential(rate=2),

    tfd.Uniform(low=-5, high=5)

])
```

```
data = mix_3gauss_1exp_1uni.sample(sample_shape=n_datapoints, seed=r_seed)

# Why is this needed???
data = tf.cast(data, tf.float64).numpy()
```

```
## [Text(0, 0.5, 'Probabililty of occurring'), Text(0.5, 0, 'Some event wiht value x
```




```
from zfit_benchmark.timer import Timer
import zfit as zfit
import pandas as pd

n_testpoints = 200

def kde_basic(data, x):

  fac = 1.0 / np.sqrt(2.0 * np.pi)
  exp_fac = -1.0/2.0
  h = 0.01
```

```
  y_fac = 1.0/(h*data.size)

  gauss_kernel = lambda x: fac * np.exp(exp_fac * x**2)

  y = np.zeros(x.size)

  for i, x_i in enumerate(x):
      y[i] = y_fac * np.sum(gauss_kernel((x_i-data)/h))

  return y

def kde_seaborn(data, x):
  sns.distplot(data, bins=1000, kde=True, rug=False)
  return np.NaN

@tf.function(autograph=False)
def kde_basic_tf_internal(data, x, n_datapoints):

  # TODO: Use tf-kde package here

  h1 = 0.01

  fac = tf.constant(1.0 / np.sqrt(2.0 * np.pi), tf.float64)
  exp_fac = tf.constant(-1.0/2.0, tf.float64)
  y_fac = tf.constant(1.0/(h1 * n_datapoints), tf.float64)
  h = tf.constant(h1, tf.float64)

  gauss_kernel = lambda x: tf.math.multiply(fac,
 ↪  tf.math.exp(tf.math.multiply(exp_fac, tf.math.square(x))))
  calc_value = lambda x: tf.math.multiply(y_fac,
 ↪  tf.math.reduce_sum(gauss_kernel(tf.math.divide(tf.math.subtract(x,
 ↪  data), h))))

  return tf.map_fn(calc_value, x)

def kde_basic_tf(data, x):
  n_datapoints = data.size
  return kde_basic_tf_internal(data, x, n_datapoints).numpy()

methods = pd.DataFrame({
  'identifier': [
    'basic',
    'seaborn',
```

```
    'basicTF'
  ],
  'label': [
    'Basic KDE with Python',
    'Using seaborn.distplot',
    'Basic KDE in TensorFlow'
  ],
  'function':[
    kde_basic,
    kde_seaborn,
    kde_basic_tf
  ]
})
methods.set_index('identifier', drop=False, inplace=True)

estimations = pd.DataFrame()
estimations['x'] = np.linspace(-5.0, 5.0, num=n_testpoints,
 ↪  dtype=np.float64)

methods['runtime'] = np.NaN
for index, method in methods.iterrows():
  with Timer('Benchmarking') as timer:
    estimations[method['identifier']] = method['function'](data,
 ↪  estimations['x'])
    timer.stop()
  print(methods.loc[method['identifier']])
  methods.at[method['identifier'], 'runtime'] = timer.elapsed
```

```
print(estimations)
```

```
print(methods)
```

```
methods.drop('function', axis=1, inplace=True)
```

Running this, leads to the following comparison:

```
knitr::kable(py$methods, booktabs = TRUE, caption = 'Runtime comparison')
```

Table 1: Runtime comparison

	identifier	label	runtime
basic	basic	Basic KDE with Python	9.948889
seaborn	seaborn	Using seaborn.distplot	6.096260
basicTF	basicTF	Basic KDE in TensorFlow	2.927457

```
knitr::kable(py$estimations, booktabs = TRUE, caption = 'Estimations
 ↪  comparison')
```

Plotted for reference:

```
## <matplotlib.axes._subplots.AxesSubplot object at 0x7ff7fbd3d490>
```

```
## <matplotlib.axes._subplots.AxesSubplot object at 0x7ff7fbd3d490>
```

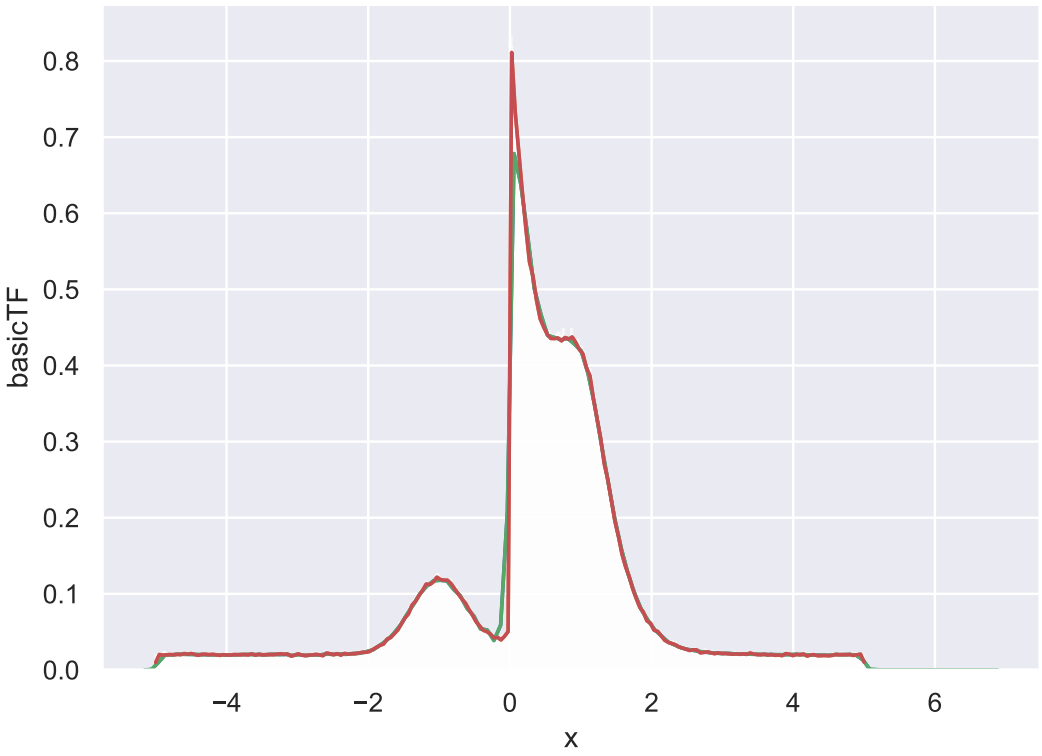


Table 2: Estimations comparison

x	basic	seaborn	basicTF
-5.0000000	0.0098180	NaN	0.0098180
-4.9497487	0.0203164	NaN	0.0203164
-4.8994975	0.0194127	NaN	0.0194127
-4.8492462	0.0198054	NaN	0.0198054
-4.7989950	0.0202779	NaN	0.0202779
-4.7487437	0.0199428	NaN	0.0199428
-4.6984925	0.0205519	NaN	0.0205519
-4.6482412	0.0201145	NaN	0.0201145
-4.5979899	0.0215724	NaN	0.0215724
-4.5477387	0.0206660	NaN	0.0206660
-4.4974874	0.0215725	NaN	0.0215725
-4.4472362	0.0202788	NaN	0.0202788
-4.3969849	0.0192163	NaN	0.0192163
-4.3467337	0.0205872	NaN	0.0205872
-4.2964824	0.0205579	NaN	0.0205579
-4.2462312	0.0200755	NaN	0.0200755
-4.1959799	0.0205893	NaN	0.0205893
-4.1457286	0.0196969	NaN	0.0196969
-4.0954774	0.0191521	NaN	0.0191521
-4.0452261	0.0201330	NaN	0.0201330
-3.9949749	0.0196654	NaN	0.0196654
-3.9447236	0.0195574	NaN	0.0195574
-3.8944724	0.0202507	NaN	0.0202507
-3.8442211	0.0200037	NaN	0.0200037
-3.7939698	0.0202515	NaN	0.0202515
-3.7437186	0.0202642	NaN	0.0202642
-3.6934673	0.0198620	NaN	0.0198620
-3.6432161	0.0212942	NaN	0.0212942
-3.5929648	0.0193476	NaN	0.0193476
-3.5427136	0.0207794	NaN	0.0207794
-3.4924623	0.0197790	NaN	0.0197790
-3.4422111	0.0201728	NaN	0.0201728
-3.3919598	0.0203412	NaN	0.0203412
-3.3417085	0.0200584	NaN	0.0200584
-3.2914573	0.0205937	NaN	0.0205937
-3.2412060	0.0212890	NaN	0.0212890

5 Summary

In summary we can conclude that…

References

[1]CERN (n.d.).

[2] M. Rosenblatt, (1956).

[3] T. Duong, An Introduction to Kernel Density Estimation (2001).

[4] M. Lerner, Histograms and Kernel Density Estimation | Biophysics and Beer (2013).

[5] K.S. Cranmer, (2000).

[6] J. Eschle, A.P. Navarro, R.S. Coutinho, and N. Serra, (2019).

List of Tables

List of Figures

Listings