

HAW HAMBURG
INFORMATIK MASTER

GRUNDPROJEKT

Evaluation von TensorFlow Probability
für statistische Analysen

Bearbeiter:

Tom Schöner (2182801)

Betreuung:

Prof. Dr. Olaf Zukunft

24. April 2019

Inhaltsverzeichnis

1	Abstract	2
2	Tensorflow Probability Komponenten	2
2.1	Schicht 0: Tensorflow	2
2.2	Schicht 1: Statistical Building Blocks	3
2.3	Schicht 2: Model Building	5
2.4	Schicht 3: Probabilistic Inference	5
3	Semantik und Projektarchitektur	6
3.1	Fehlerbehandlung	6
3.2	Dokumentation	6
4	Anwendungsgebiete	7
4.1	Beispiel: Korrelation von Luftverschmutzung und Temperatur	7
4.1.1	Modellierung ohne Unsicherheit	8
4.1.2	Modellierung mit Unsicherheit	10
5	Fazit	12

1 Abstract

Die auf Tensorflow basierende Bibliothek Tensorflow Probability¹ - fortan mit *TFP* abgekürzt - erweitert das Framework um eine probabilistische Komponente. Mittels einer breiten Masse an bereitgestellten Tools, wie statistischen Verteilungen, Sampling oder verschiedenster probabilistischer Erweiterungen für neuronale Netze, können einfache bis hin zu komplexen Modellen erstellt werden. Berechnungen werden, wie man es aus Tensorflow gewohnt ist, durch *Dataflow Graphs*² abgebildet. Auf die verschiedenen Funktionsweisen und Schichten von TFP wird in Abschnitt 2 detaillierter eingegangen.

In dieser Evaluation soll die Bibliothek auf ihre Semantik und Pragmatik, Effektivität beim Erstellen von statistischen Modellen und Integration in das Framework Tensorflow untersucht werden. Das maschinelle Lernen mit Hilfe von neuronalen Netzen und deren Abstraktion durch Keras ist hierbei als Schwerpunkt anzusehen.

2 Tensorflow Probability Komponenten

Die Struktur von TFP lässt sich, wie aus der Dokumentation zu entnehmen ist³, in die folgenden vier Schichten einteilen. Die Schichten bauen hierarchisch aufeinander auf, abstrahieren die unterliegenden Schichten aber nicht zwangsläufig. Möchte man beispielsweise durch *MCMC* in Schicht 3 Parameter seines probabilistischen Modells mittels Sampling ermitteln, sollten *Bijectors* aus Schicht 1 kein Fremdwort sein.

2.1 Schicht 0: Tensorflow

TFP ist nicht als eigenständige Komponente neben Tensorflow anzusehen, sondern als Bestandteil dessen. Die probabilistischen Berechnungen werden innerhalb von Tensorflow Sessions oder im *Eager*-Modus ausgeführt. Tensorflow

¹<https://www.tensorflow.org/probability>

²<https://www.tensorflow.org/guide/graphs>

³<https://www.tensorflow.org/probability/overview>

wird von mehreren Programmiersprachen wie Python, JavaScript oder C++ unterstützt. Die Bibliothek TFP ist aktuell nur für die primär unterstützte Programmiersprache Python implementiert.

2.2 Schicht 1: Statistical Building Blocks

Als Fundament statistischer Modelle sind mehrere, in Python Module aufgeteilte, Klassen und Funktionen gegeben. Diese können in der API Dokumentation der TFP Website eingesehen werden. Ein Beispiel hierfür ist das Modul **tfp.stats**. Unter **tfp.stats** finden sich unter Anderem Funktionen für die Berechnung für Korrelationen, Quantilen oder Standardabweichungen. Diese Funktionen sind auf die Verwendung von Tensoren ausgelegt, lassen sich generell aber auch mit normalen n-dimensionalen Python Arrays oder Numpy Arrays aufrufen.

Verschiedenste, für probabilistische Modelle essentielle Verteilungen reihen sich unter dem Modul **tfp.distributions** in dieser Schicht ein: *Normal*-, *Bernoulli*-, *Exponential*- oder *Gammaverteilung*, um einige zu nennen. Generell erben Klassen für Verteilungen, wie etwa die Normalverteilung, von der Klasse **tfp.distributions.Distribution**⁴. Diese gemeinsame Schnittstelle standardisiert die Benutzung. Sie fordert zudem Implementationen mehrerer hilfreicher Methoden wie etwa für die logarithmische Wahrscheinlichkeit oder dem Mittelwert. Am Beispiel der Normalverteilung mit der Definition $X \sim \mathcal{N}(\mu, \sigma^2)$ sind im Folgenden einige der Funktionen aufgeführt (siehe Listing 1). Visualisiert man die Verteilung auf Basis der Samples *sample* (Zeile 5) ergibt sich Abbildung 1.

```
1 normal_dist = tfd.Normal(name="N", loc=0., scale=1.)
2 # tfp.distributions.Normal("N", batch_shape=(), event_shape=(), dtype=
   float32)
3
4 sample = normal_dist.sample(sample_shape=normal_sample_size)
5 # <tf.Tensor: id=188, shape=(2500,), dtype=float32, numpy=array
   ([ -0.45733708, -0.19126031, -0.33290815, ..., -1.1285563 , -0.6958163 ,
     0.552399  ], dtype=float32)>
6
```

⁴https://www.tensorflow.org/probability/api_docs/python/tfp/distributions/Distribution

```

7 normal_dist.mean()
8 # tf.Tensor(0.0, shape=(), dtype=float32)
9
10 normal_dist.prob(1.0)
11 # tf.Tensor(0.24197072, shape=(), dtype=float32)

```

Listing 1: Verwendung der Klasse `tfp.distributions.Normal`

Broadcasting, *Batching* und *Shapes* sorgen dafür, dass unabhängige Verteilungen als sogenannter Batch in einer Entität gekapselt werden können. Um beispielsweise ein zweidimensionales Batch für die Normalverteilung aus Listing 1 zu erzeugen, kann als Erwartungswert μ `loc=[0., 10.]` übergeben werden. Da TFP hier ebenfalls broadcasting unterstützt müssen die folgenden Aufrufe nicht angepasst werden, nur das Ergebnis wird ebenfalls Zweidimensional sein. Broadcasting verhält sich analog zu dem in Numpy etablierten Konzept⁵.

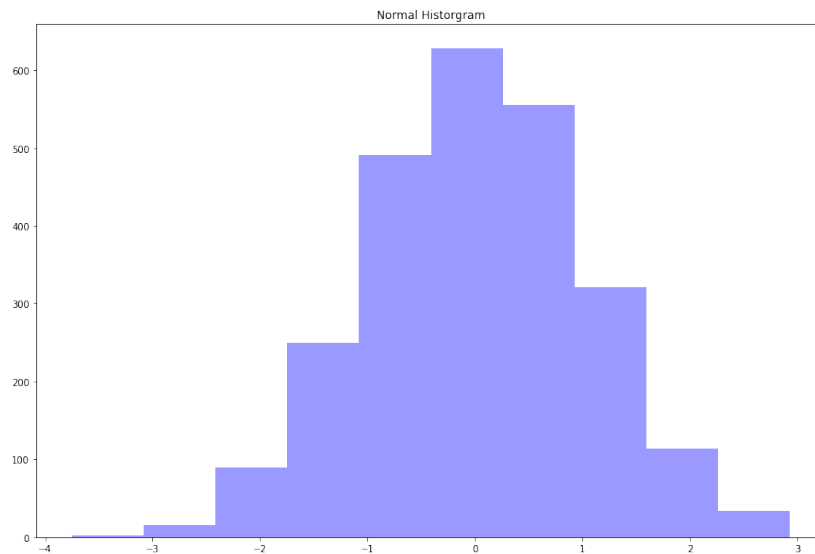


Abbildung 1: $X \sim \mathcal{N}(0, 1)$ mit 2500 Samples

Ein weiterer Bestandteil von Schicht 1 sind *Bijectors*. Bijectors bilden eine Zahl aus \mathbb{R}^n auf \mathbb{R}^m ab oder jeweils einer Submenge dieser. Aufgrund

⁵<https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html>

der im Bijector hinterlegten bijektiven Funktion ist dieser Schritt umkehrbar, daher: $x = f^{-1}(f(x))$. Dies ist besonders bei der Transformierung von Stichproben aus Verteilungen nützlich und wird folglich für das Erstellen von Modellen (Abschnitt 2.3) und Berechnungen probabilistischer Interferenzen (Abschnitt 2.4) eingesetzt. TFP bietet bereits einige vorgefertigte Bijectors unter **tfp.bijectors**⁶ an.

2.3 Schicht 2: Model Building

Edward2 / Probabilistic Layers with Keras / Trainable Distributions

In Abschnitt 4.1 wird die Verwendung und probabilistischen Layers mit Keras anhand eines Beispiels verdeutlicht.

2.4 Schicht 3: Probabilistic Inference

Als letzte Schicht enthält TFP Werkzeuge für probabilistische Inferenz. Dazu gehören *Markov chain Monte Carlo (MCMC)* Algorithmen, *Variational Inference* Algorithmen und stochastische Optimierungsverfahren.

Durch Sampling ermöglicht MCMC das Berechnen statistischer Modelle mit einer hohen Anzahl an Parametern. Eine analytische Herangehensweise ist bereits ab wenigen Parametern nicht mehr sinnvoll, da hierfür die Bestimmung eines multidimensionalen Integrals erforderlich wäre. TFP bietet mehrere MCMC Algorithmen unter **tfp.mcmc** an.

Listing 2 zeigt einen Ausschnitt der Verwendung des Hamiltonian Monte Carlo Algorithmus. Die hier nicht enthaltene Auswertung würde in einer approximierten Verteilung des Parameters *tau* für das gegebene Modell *_data_model* resultieren. Ein praxisbezogenes und ausführlicheres Beispiel ist unter <https://github.com/tom-schoener/ml-probability/blob/master/tfp-evaluation/notebooks/mcmc.ipynb> einsehbar.

```
1 # Set the chain's start state.
2 initial_chain_state = [ 0.5 * tf.ones([], dtype=tf.float32, name="init_tau")
3 ]
```

⁶https://www.tensorflow.org/probability/api_docs/python/tfp/bijectors

```

4 def joint_log_prob(_data_model, tau):
5     rv_tau = tfd.Uniform()
6     rv_observation = tfd.Poisson(rate=rv_tau)
7
8     return rv_tau.log_prob(tau) + tf.reduce_sum(rv_observation.log_prob(
9         _data_model))
10
11 # setup the chain
12 [ tau ], kernel_results = tfp.mcmc.sample_chain(
13     num_results=1000,
14     num_burnin_steps=500,
15     current_state=initial_chain_state,
16     kernel=tfp.mcmc.TransformedTransitionKernel(
17         inner_kernel=tfp.mcmc.HamiltonianMonteCarlo(
18             target_log_prob_fn=lambda tau: joint_log_prob(_data_model, tau)),
19             bijector=[ tfp.bijectors.Sigmoid() ] # Maps [0,1] to R
20 )

```

Listing 2: Verwendung des Hamiltonian Monte Carlo Algorithmus (gekürzt)

Die in der API verwendeten Bezeichnungen decken sich hier größtenteils mit der in der Literatur verwendeten Sprache, wodurch die Übertragung von Konzepten aus unterschiedlichen Quellen deutlich vereinfacht wird.

3 Semantik und Projektarchitektur

3.1 Fehlerbehandlung

3.2 Dokumentation

TFPs offizielle Dokumentation bietet neben der Beschreibung der API eine Auflistung an Beispielen und weiterführenden Ressourcen. Die, als Einführung anzusehende und kürzlich nach TFP portierte, Version von *Bayesian Methods for Hackers: Probabilistic Programming and Bayesian Inference*^[1] stellte sich als besonders Hilfreich heraus.

Die Bibliothek befindet sich in einem vergleichsweise frühem Stadium der Entwicklung und es gibt einen Hinweis, dass es zu Änderungen kommen kann. Teile der API sind entsprechend noch nicht vollständig dokumentiert (Stand März 2019). Undokumentierte Funktionen sind allerdings eher die Ausnahme

als die Norm. Ein Großteil der in dieser Evaluation verwendeten Schnittstellen ließ sich anhand von Beispielen, genaueren Erläuterungen der Parameter oder durch Verweise auf relevante Literatur und Paper nachvollziehen.

4 Anwendungsgebiete

4.1 Beispiel: Korrelation von Luftverschmutzung und Temperatur

Das Jupyter Notebook für das folgende Beispiel ist unter https://github.com/tom-schoener/ml-probability/blob/master/tfp-evaluation/notebooks/air_quality.ipynb einsehbar. Die Daten stammen aus dem *UC Irvine Machine Learning Repository*⁷.

Von März 2004 bis Februar 2005 wurden in einer italienischen Stadt nahe einer stark befahrenen Straße verschiedene Wetterdaten wie Temperatur, Luftfeuchtigkeit oder auch Stickstoffdioxid (NO_2) stündlich gemessen. Die Daten für Temperatur, relativer Luftfeuchtigkeit und NO_2 sind aggregiert in den Abbildungen 2, 3 und 4 visualisiert. Die blaue Linie stellt den gemittelten Wert um 00:00 Uhr Nachts und die orangene Linie um 12:00 Mittags dar.

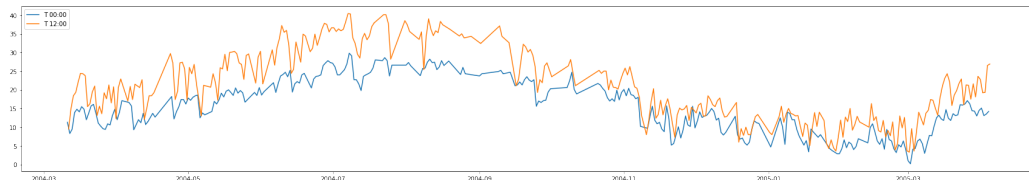


Abbildung 2: Temperatur in $^{\circ}C$

In Europa ist der Grenzwert von NO_2 gesetzlich auf max $200\mu g/m^3$ festgelegt. Die EU-Richtlinie 2008/50/EG verschärfte dieses Gesetz insofern, dass das Jahresmittel $40\mu g/m^3$ nicht überschreiten darf. In diesem Beispiel soll ein probabilistisches Modell für Aussagen über die Korrelation zwischen Temperatur und Luftverschmutzung auf Basis eines neuronalen Netzes in

⁷<https://archive.ics.uci.edu/ml/datasets/Air+quality>

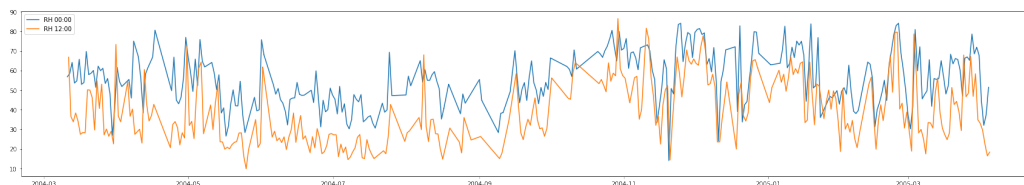


Abbildung 3: Relative Luftfeuchtigkeit in %

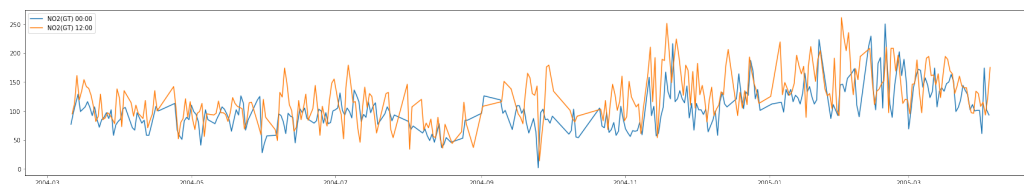


Abbildung 4: Stickstoffdioxid NO_2 in $\mu\text{g}/\text{m}^3$

TFP erstellt werden. Da bei probabilistischen Modellen Verlust (*loss*) mit modelliert werden kann, wird nicht nur die Relation der Daten, sondern auch die Aussagekraft des jeweiligen ermittelten Wertes, zum Beispiel über dessen Standardabweichung, verdeutlicht.

4.1.1 Modellierung ohne Unsicherheit

Um das Modell erstellen zu können, müssen die Daten zunächst bereinigt werden. Null- oder fehlende Werte werden der Einfachheit halber interpoliert. In Listing 3 ist das Erstellen eines einfachen linearen Modells mit TFP und Keras zu sehen. Das Modell besteht aus zwei Schichten: Einem *dense layer* mit einem eindimensionalen Output-Vektor und eine durch eine Normalverteilung beschriebene Schicht, deren Input-Vektor durch den Output-Vektor des *dense layers* befüllt wird. Die Normalverteilung benutzt seinen Input, um den Mittelwert zu bestimmen. Beim Trainieren des Modells wird der Verlust (*loss*) durch die Funktion *neg_log_lik*, daher dem negativen Logarithmus einer Probe der aktuell bestimmten Verteilung, festgelegt. Es sollte auffallen, dass die Standardabweichung konstant bei 1 liegt, wodurch das resultierende Modell ebenfalls eine konstante Standardabweichung für jede Temperatur aufweisen wird. Abschnitt 4.1.2 erweitert das Modell um diesen Aspekt.

```

1 def model_no_uncertainty(feature, label, epochs=100):
2     # feature
3     x = np.array(df[feature])
4     x = x[... , np.newaxis]
5
6     # label
7     y = np.array(df[label])
8
9     # model
10    model = tf.keras.Sequential([
11        tf.keras.layers.Dense(1),
12        tfp.layers.DistributionLambda(lambda t: tfd.Normal(loc=t, scale=1)),
13    ])
14
15    # inference
16    neg_log_lik = lambda y_: -rv_y.log_prob(y_)
17    model.compile(optimizer=tf.optimizers.Adam(learning_rate=0.01), loss=
18        neg_log_lik)
19    model.fit(x, y, epochs=epochs)
20    return (x, y, model)

```

Listing 3: Modell mit Keras ohne Unsicherheit

Verwendet man dieses Modell, um anhand der Temperatur eine Prognose für jeweils die Luftfeuchtigkeit und den Stickstoffgehalt in der Luft zu bestimmen ergeben sich Abbildung 5 und 6.

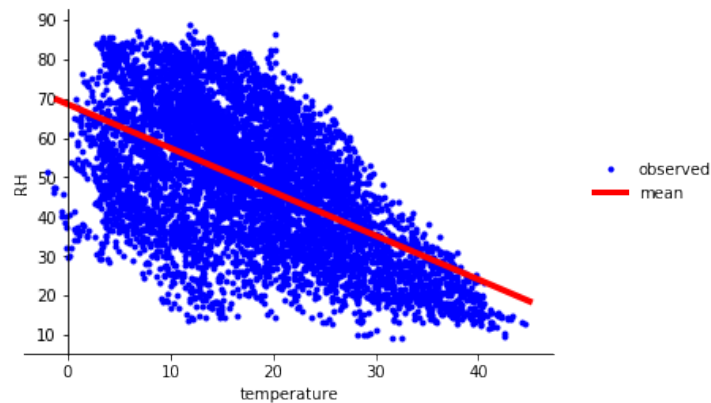


Abbildung 5: Prognose der relativen Luftfeuchtigkeit

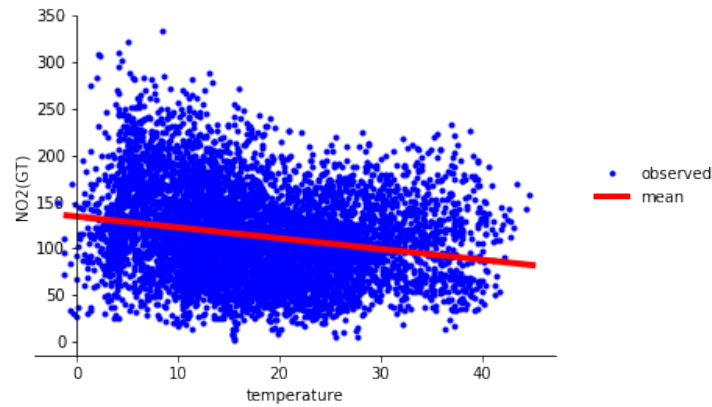


Abbildung 6: Prognose des Stickstoffgehalts in der Luft

4.1.2 Modellierung mit Unsicherheit

Betrachtet man sich Abbildung 5 fällt auf, dass die Unsicherheit des hier linearen Ansatzes bei geringen Temperaturen stark zunimmt. Mit anderen Worten: Die Genauigkeit lässt nach. Das Modell kann dieses Problem aber noch nicht repräsentieren, daher wird im folgenden, verbesserten Modell die Unsicherheit mit integriert. Listing 4 beschreibt die Änderung. Der Aufbau ist dem ersten sehr ähnlich und unterscheidet sich im Wesentlichen nur in zwei Aspekten:

- Die erste Schicht hat einen zweidimensionalen Output-Vektor, um die Unsicherheit mit darzustellen.
- Die zweite Schicht verwenden den zweiten Wert des Vektors, um seine Standardabweichung zu definieren.

```

1 def model_with_uncertainty(feature, label, epochs=5000):
2     # feature
3     x = np.array(df[feature])
4     x = x[..., np.newaxis]
5
6     # label
7     y = np.array(df[label])
8
9     model = tf.keras.Sequential([

```

```

10     tf.keras.layers.Dense(2),
11     tfp.layers.DistributionLambda(lambda t: tfd.Normal(
12         loc=t[..., :1],
13         scale=1e-3 + tf.math.softplus(5e-3 * t[..., 1:]))))
14 ])
15
16 # inference
17 negloglik = lambda y_: -rv.y.log_prob(y_)
18 model.compile(optimizer=tf.compat.v2.optimizers.Adam(learning_rate=0.01),
19               loss=negloglik)
20 model.fit(x, y, epochs=epochs);
21
22 return (x, y, model)

```

Listing 4: Modell mit Keras mit Unsicherheit

Das Ergebnis lässt sich in den Abbildungen 7 und 8 ablesen. Die Standardabweichung ist nun Teil des Modells. Auffällig ist hier, dass die Prognose der Unsicherheit des Stickstoffgehalts die Daten nicht sehr genau abbilden. Das kann mehrere Gründe haben. Hier wurde nur ein einfaches lineares Modell für Demonstrationszwecke verwendet. Einige der Hyperparameter könnten weiter angepasst werden und die Trainingszeit oder Datenmenge ist nicht unbedingt ausreichend. Natürlich kann es auch sein, dass die Temperatur kein sinnvoller Maßstab für den Stickstoffgehalt in der Luft ist, wodurch das Modell keine große Aussagekraft besitzt.

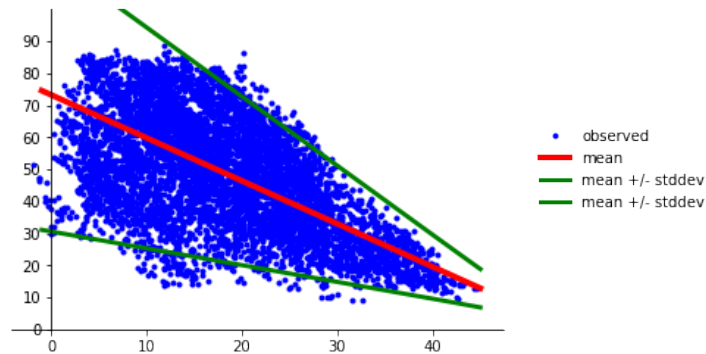


Abbildung 7: Prognose der relativen Luftfeuchtigkeit mit variabler Standardabweichung

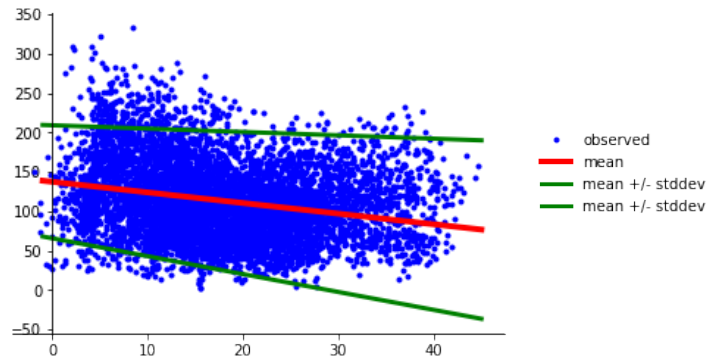


Abbildung 8: Prognose des Stickstoffgehalts in der Luft mit variabler Standardabweichung

5 Fazit

Die Bibliothek TFP ist eine praktische Erweiterung des Tensorflow Frameworks, um sein Modell um eine statistische Ebene zu erweitern. Die bereitgestellten Funktionen sind sinnvoll aufgeteilt und es sind im Normalfall Beispiele vorhanden. Auch beim Release von Tensorflow 2.0 wurden Anpassungen an TFP vorgenommen, was das Interesse von Google und der OpenSource Community an der Weiterentwicklung von TFP zeigt.

Die Einarbeitungszeit ist, auch unter Ansicht der Komplexität des Themengebietes, nicht zu Vernachlässigen. Ein Beispiel hiervon sind *Dataflow Graphs*. *Dataflow Graphs* von Tensorflow ermöglichen ein effizientes Berechnungsmodell, welches auf GPUs und auf neuere spezialisierte Hardware wie TPUs ausgelegt ist. Einfache Anwendungsfälle profitieren aber nicht unbedingt von der Abstraktionsebene von Tensorflow. Eine alternative und bekannte Bibliothek für probabilistische Programmierung in Python, anstelle von TFP, ist beispielsweise *PyMC3*⁸.

Die Integration in Keras ist einer der nennenswertesten Stärken von TFP, da hierdurch die Komplexität an der richtigen Stelle gelockert wird. Probabilistische Modelle zu Erstellen wird hierdurch nicht trivial, aber deutlich

⁸<https://docs.pymc.io/>

zugänglicher. Ein fundamentales Wissen über die Domäne und das breite Gebiet der Statistik ist immer noch erforderlich.

Besonders für maschinelles Lernen eignet sich der Einsatz von TFP daher gut, da es sich in das bestehende Framework mit einreicht.

Literatur

- [1] Cameron Davidson-Pilon. *Bayesian Methods for Hackers: Probabilistic Programming and Bayesian Inference*. Addison-Wesley Professional, 1st edition, 2015. ISBN 0133902838, 9780133902839. URL <https://camdavidsonpilon.github.io/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers/#tensorflow>.

Abbildungsverzeichnis

1	$X \sim \mathcal{N}(0, 1)$ mit 2500 Samples	4
2	Temperatur in °C	7
3	Relative Luftfeuchtigkeit in %	8
4	Stickstoffdioxid NO_2 in $\mu g/m^3$	8
5	Prognose der relativen Luftfeuchtigkeit	9
6	Prognose des Stickstoffgehalts in der Luft	10
7	Prognose der relativen Luftfeuchtigkeit mit variabler Standardabweichung	11
8	Prognose des Stickstoffgehalts in der Luft mit variabler Standardabweichung	12

Listings

1	Verwendung der Klasse <code>tfp.distributions.Normal</code>	3
2	Verwendung des Hamiltonian Monte Carlo Algorithmus (gekürzt)	5
3	Modell mit Keras ohne Unsicherheit	9
4	Modell mit Keras mit Unsicherheit	10