Proseminar „Softwareentwicklung in der Wissenschaft“

Probabilistic Programming

Katja Tilli Julia Neumann

Betreuer: Tobias Finn

Abgabedatum: 31. August 2020

Inhalt

I. Einleitung

Künstliche Intelligenz (KI) und Machine Learning (ML) sind die Stichworte der Stunde, wenn es um die aktuellen Entwicklungen in der Informatik geht. Doch die Arbeit der KI- und ML-Systeme ist abhängig von starken Werkzeugen im Hintergrund, die den Nutzern dabei helfen, deren Funktionsweise zu implementieren. Eines der vielversprechendsten Werkzeuge in diesem Gebiet findet seit einigen Jahren unter dem Namen Probabilistic Programming verstärkte Aufmerksamkeit. Bereits heute sind die Einsatzgebiete vielfältig. Stets liegt den Einsätzen aber eine simple Idee zugrunde: Eine Künstliche Intelligenz nähert sich ihren Schlussfolgerungen, indem sie berechnet, welche Schlussfolgerung aufgrund ihres Inputs die wahrscheinlichste ist. Probabilistic Programming liefert die Grundlage, die dafür nötigen Wahrscheinlichkeiten zu modellieren.

Die Anwendungsfelder reichen von Einsätzen in der Medizin- und Klimaforschung bis hin zur Computervision und Generierung von Computergrafik. Nutzer können mit Probabilistic Programming Systeme implementieren, die selbstständig die Umgebung in einem Computerspiel erstellen und dabei gewisse Regeln beachten können, wie z.B. Bäume generieren, die andere Gegenstände nicht berühren (van de Meent et al., 2018). Systeme können Spieler so einteilen, dass stets solche gegeneinander antreten, die ungefähr gleich stark sind (Herbrich et al., 2006). Probabilistic Programming kann dabei unterstützen, Dunkle Materie zu untersuchen (Chianese, 2019) oder die Epidemiologie von Malaria-Erkrankungen besser zu verstehen (Gram-Hansen et al., 2019).

Wahrscheinlichkeiten zu berechnen ist eine schwierige Aufgabe, zumal wenn große Datenmengen hinzukommen und komplexere Fragen beantwortet werden sollen. Zugleich beruhen jegliche Aufgaben in diesem Bereich auf denselben mathematischen Grundlagen. Es ist daher naheliegend, den mathematischen Hintergrund weitgehend auf Computerprogramme zu übertragen. Für die Nutzer bleibt dann nur noch übrig, ihre Beobachtungen als Ausgangspunkt und ihre Forschungsfrage als Ziel dem Programm zu übermitteln.

Hier zeigt sich schon, dass Probabilistic Programming Languages nur relativ spezifische Aufgaben ausführen können. Auch wenn einige von ihnen eigenständige Sprachen sind, so sind sie nicht universell einsetzbar. Viele Probabilistic Programming Languages bauen daher auch auf bestehenden Programmiersprachen auf. Ihre Arbeitserleichterung zeigt sich besonders eindrucksvoll in einem Beispiel des Massachusetts Institute of Technology (MIT; Kulkarni et al., 2015): Mit einer neuen Probabilistic Programming Language namens Picture haben die Forscher in weniger als 50 Zeilen Code ein Programm erstellen können, das gelernt hat, aus 2D-Darstellungen von Gesichtern 3D-Modelle zu generieren. Ein solches Programm hätte ohne Probabilistic Programming mehrere tausend Zeilen Code benötigt. Wie genau funktioniert die Technologie also, die eine solche Arbeitserleichterung ermöglicht?

II. Grundlagen

Probabilistic Programming ist ein zusammenfassender Begriff für Sprachen und Werkzeuge, die Wahrscheinlichkeitsverteilungen berechnen und analysieren können. Um Wahrscheinlichkeiten zu berechnen, existieren bewährte mathematische Verfahren, die in ihren Grundzügen

leicht zu verstehen sind, im Verständnis und der Umsetzung der Feinheiten jedoch Herausforderungen mit sich bringen. Das Versprechen von Probabilistic Programming ist nun, dass es zwar weiterhin erforderlich bleibt, Wahrscheinlichkeitsrechnung grundlegend zu verstehen, um die Technologie anwenden zu können (sogleich 1.). Komplexere Prozesse, um Wahrscheinlichkeitsverteilungen mit vielen Zufallszahlen und großen Datenmengen zu modellieren, übernimmt allerdings mit nur wenigen Befehlen das Probabilistic Program (2.).

1. Berechnung von Wahrscheinlichkeiten

Wahrscheinlichkeitsverteilungen beschreiben die Wahrscheinlichkeit, dass ein bestimmtes Ereignis eintritt – genauer, dass eine Zufallszahl einen bestimmten Wert aus einer gegebenen Wertemenge annimmt. Die Wahrscheinlichkeitsverteilung beschreibt dann für jeden möglichen Wert der Wertemenge, wie wahrscheinlich es ist, dass die Zufallszahl bei einer Stichprobe genau diesen annimmt.

Dabei gibt es unterschiedliche Formen: Grundlegend aufteilen lassen sich die Verteilungen in diskrete (Abb. 1) und kontinuierliche Wahrscheinlichkeitsverteilungen (Abb. 2), also solche, bei denen die Zufallszahl einen Wert einer endlichen Menge annehmen kann, oder solche, bei denen die Zufallszahl einen beliebigen Wert aus einem Intervall annehmen kann. Eine Sonderform der diskreten Wahrscheinlichkeitsverteilung ist eine Bernoulliverteilung[1], die die Wahrscheinlichkeit beschreibt, dass von zwei möglichen Ereignissen genau eines eintritt (Abb. 3). Eine Form der kontinuierlichen Wahrscheinlichkeitsverteilung ist eine Gauß'sche Normalverteilung, die die charakteristische Form einer „Glockenkurve" einnimmt (Abb. 4). Die durch sie beschriebene Zufallszahl hat also eine besonders hohe Wahrscheinlichkeit für einen Wert in der Mitte der Verteilung, eine besonders geringe Wahrscheinlichkeit für einen Wert an den Rändern der Verteilung. Mit diesen Verteilungen lässt sich nun nicht nur abbilden, ob eine Münze nach einem Wurf Kopf oder Zahl zeigen wird (eine Bernoulli-

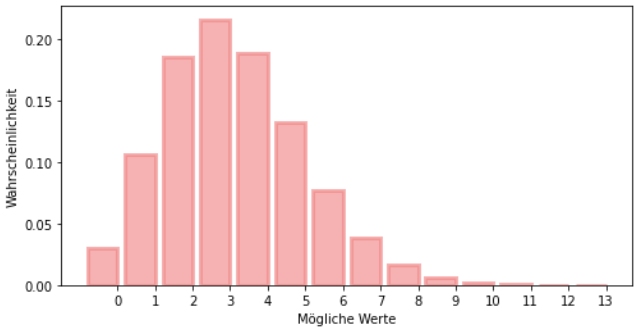

Abbildung 1: Diskrete Verteilung

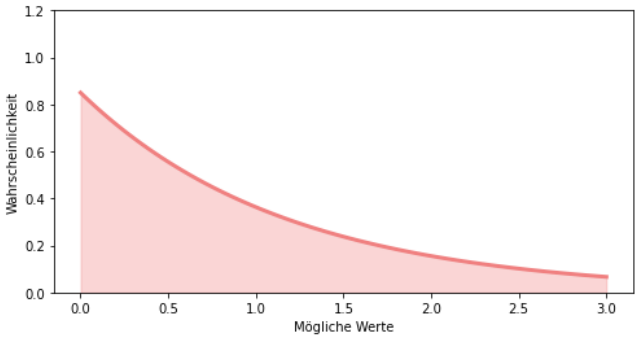

Abbildung 2: Kontinuierliche Verteilung

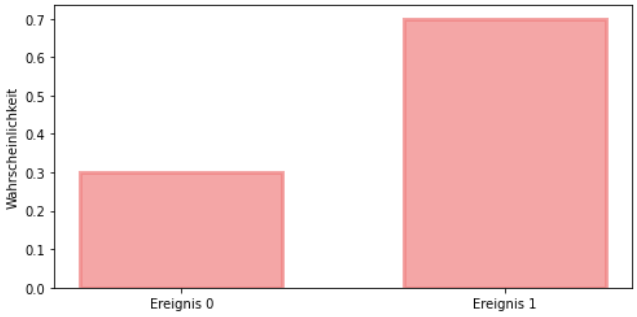

Abbildung 3: Bernoulliverteilung

[1] Die Bernoulliverteilung ist ebenfalls unter dem Namen Binomialverteilung bekannt, da sie aber in Pyro unter distributions.Bernoulli() aufgerufen wird (s.u.), soll sie in dieser Arbeit ausschließlich so genannt werden.

verteilung mit zwei möglichen Ereignissen, vgl. noch unten III.2.), sondern beispielsweise auch, welche Temperatur morgen in Hamburg herrschen wird (eine kontinuierliche Verteilung mit beliebigen Werten zwischen beispielhaften -20 und +35 Grad Celsius).

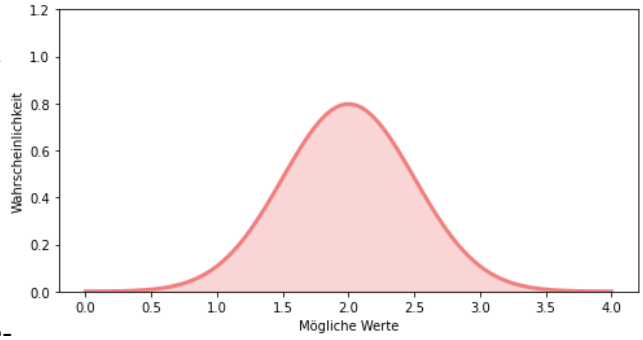

Abbildung 4: Gauß'sche Normalverteilung

Wie oben gezeigt wurde, gibt es verschiedene Gründe, die Wahrscheinlichkeitsverteilung einer bestimmten Zufallsvariablen herauszufinden – sei es, um das Wetter oder die Börsenkurse vorherzusagen, sei es, um eine Diagnose anhand von Zellbildern zu stellen. Allen gemein ist, dass die Wahrscheinlichkeitsverteilung der Zufallsvariable oder der Faktoren, die auf sie einwirken, nicht bekannt sind und sich nicht einfach ermitteln lassen. So steht nirgends geschrieben, dass in Hamburg mit 10-prozentiger Wahrscheinlichkeit morgen 15 Grad Celsius herrschen werden oder eine bestimmte Münze mit exakt 49-prozentiger Wahrscheinlichkeit Kopf zeigen wird. Sog. latente Variablen lassen sich nicht einmal direkt beobachten, sondern es ist erforderlich, über andere Ereignisse indirekt Schlüsse zu ziehen. Allen Zufallsvariablen ist gemein, dass man sich den Wahrscheinlichkeiten stets nur annähern und versuchen kann, eine Verteilung zu modellieren, die die tatsächlichen Werte möglichst genau abbildet.

Um eine Verteilung zu finden, die der Wirklichkeit möglichst nahekommt, ist es nötig, Inferenz zu betreiben. Dazu bildet man aus einzelnen Werten, die die Zufallszahl tatsächlich angenommen hat (z.B. die Temperaturen in Hamburg der letzten 365 Tage, die letzten 10 Münzwürfe), eine Verteilung ab, sodass sie den tatsächlich gemessenen Werten maximal eng entspricht bzw. minimal abweicht. Das Finden dieser Wahrscheinlichkeitsverteilung lässt sich daher z.B. als Optimierungsproblem betrachten: Aus einer Vielzahl möglicher Verteilungen, aus denen die gemessenen Werte entstammen könnten, versucht man, diejenige zu finden, die möglichst gut dazu passt.

In der Praxis ist die Modellierung von Wahrscheinlichkeitsverteilungen und der betriebenen Inferenz deutlich komplexer. Zum einen ist es von Interesse, Wahrscheinlichkeitsverteilungen auch aus sehr großen Datensätzen abzuleiten. Zum anderen ist es auch möglich, verschiedene Zufallszahlen zu verknüpfen: So ist die Wahrscheinlichkeit, dass eine Studierende ihre Seminararbeit rechtzeitig abgibt, möglicherweise davon abhängig, welche Temperaturen in Hamburg herrschen – je höher, desto unwahrscheinlicher. Um die Wahrscheinlichkeit herauszufinden, mit der die Studierende ihre Arbeit rechtzeitig abgibt, muss man also zunächst die Wahrscheinlichkeit höherer Temperaturen im fraglichen Zeitraum berechnen, um anschließend in Abhängigkeit davon erstere Wahrscheinlichkeit berechnen zu können. Solche Verknüpfungen von Wahrscheinlichkeitsverteilungen ließen sich um beliebig viele weitere Zufallszahlen erweitern, die ebenfalls einen Einfluss auf die fragliche Wahrscheinlichkeit haben. Es liegt nahe, dass bei sehr vielen verknüpften Zufallszahlen, deren Wahrscheinlichkeitsverteilungen jeweils auf sehr vielen Daten beruhen, der Einsatz von Computern, die die Inferenz übernehmen, vorteilhaft ist.

Wahrscheinlichkeiten lassen sich auf zwei unterschiedliche Weisen interpretieren, mithilfe des frequentistischen und des Bayes'schen Verständnisses. Die frequentistische Sicht auf Wahrscheinlichkeiten bedeutet, dass Wahrscheinlichkeit die Häufigkeit darstellt, mit der eine Zufallsvariable einen bestimmten Wert annimmt, wenn man ihren Wert häufig abfragt. Man berechnet also mit der Wahrscheinlichkeit die sehr langfristige Häufigkeit von möglichen Werten der Zufallsvariablen. Hingegen erscheint das Bayes'sche Verständnis von Wahrscheinlichkeiten intuitiver: Hiernach beschreibt Wahrscheinlichkeit den Grad der Sicherheit, dass eine Zufallsvariable diesen oder jenen Wert annimmt. Das erlaubt zum einen, dass unterschiedliche Modelle von Wahrscheinlichkeitsverteilungen unterschiedlich sicher sein können, dass eine Zufallsvariable diesen oder jenen Wert annehmen wird. Zum anderen geht dieses Verständnis transparenter damit um, dass Inferenz nicht eine definitive Antwort („am wahrscheinlichsten ist es, dass die Variable diesen Wert annimmt"), sondern abgestufte Wahrscheinlichkeiten für unterschiedliche Werte der Zufallsvariablen zurückliefert („mit 51-prozentiger Wahrscheinlichkeit nimmt unsere Variable diesen Wert an, mit 49-prozentiger Wahrscheinlichkeit aber diesen Wert"). Bayes'sche Inferenz ist daher vorteilhaft, wenn man Rückschlüsse aus unperfekten, unvollständigen oder kleinen Datensätzen ziehen möchte (Davidson-Pilon, 2015). Sie unterliegt weniger starken Schwankungen, wenn neue Daten hinzukommen.

Das Bayes'sche Verständnis schlägt sich auch im Satz von Bayes nieder: $P(A|X) = \frac{P(X|A)P(A)}{P(X)}$. P(A) verkörpert dabei die Wahrscheinlichkeit, dass eine Zufallszahl einen bestimmten Wert annimmt. P(A|X) steht für die Wahrscheinlichkeit, dass diese Zufallszahl einen bestimmten Wert annimmt, wenn X gegeben ist. Die Formel steht damit zugleich für das Verfahren, mit dem eine Wahrscheinlichkeitsverteilung näherungsweise an die reale Verteilung angepasst werden kann: Zu Beginn steht eine Vermutung (P(A)), welche Werte die Zufallszahl wohl am wahrscheinlichsten annehmen kann. Man nennt sie im Zusammenhang mit Probabilistic Programming oft auch die A-priori-Wahrscheinlichkeit. Diese Vermutung kann ihrerseits bereits auf Daten beruhen oder eine bloße Mutmaßung sein. Kommen nun die Daten X hinzu, kann man über die Formel eine aktualisierte Vermutung P(A|X) aufstellen, die A-posteriori-Wahrscheinlichkeit. Dazu ist dann auch Wissen oder möglichst präzise Annahmen über die Wahrscheinlichkeiten vonnöten, dass die Daten X gemessen werden (P(X)) und dass die Daten X gemessen werden, wenn A gegeben ist (P(X|A)). Indem das Verfahren, das auch als Bayes'sches Lernen bekannt ist, bereits getroffene Vorannahmen durch neue Daten verändert und präzisiert, eignet es sich somit ideal für alle Situationen, in denen man mithilfe von Daten Erkenntnisprozesse durchführen möchte.

2. Erleichterungen durch Probabilistic Programming

Probabilistic Programming kann den Prozess, Wahrscheinlichkeiten zu berechnen, vereinfachen. Es vereint dabei Methoden, um Wahrscheinlichkeitsverteilungen konzis zu beschreiben und sich ihrer realen Form möglichst weit anzunähern. Sein Versprechen ist: Die Nutzer von Probabilistic Programming Languages beschreiben die gemessenen Daten und die grundlegende Form der Wahrscheinlichkeitsverteilung, die diese Daten erklären soll. Probabilistic Programming übernimmt dann mithilfe von Programmbefehlen den mathematischen Hintergrund der Berechnungen, betreibt also Inferenz und präsentiert dem Nutzer anschließend die gesuchten Rückschlüsse.

Ein Probabilistic Program ist ein Programm, das zwei besondere Fähigkeiten hat: Zum einen kann es konkrete Werte aus einer Wahrscheinlichkeitsverteilung ziehen. Zum anderen kann es aus tatsächlich beobachteten Daten eine Wahrscheinlichkeitsverteilung modellieren. Diese beiden Fähigkeiten sind für sich genommen bereits nützlich, doch seine Stärken spielt ein Probabilistic Program erst dann wirklich aus, wenn es beide miteinander kombiniert. Seine Besonderheit ist, dass es in zwei Richtungen arbeitet (Cronin, 2013): Zuerst zieht es aus dem vom Nutzer vorgegebenen Modell Stichproben, die in großer Zahl gemeinsam einer Nachbildung des Modells gleichkommen. Das Programm bekommt also Daten und ermittelt aus diesen Werte – es läuft wie gewohnt „vorwärts". Anschließend jedoch nimmt das Programm die soeben gefundenen Stichproben und läuft „rückwärts": Es sucht eine möglichst gute Erklärung für diese ermittelten Werte, betreibt also Inferenz. Dazu modifiziert und untersucht es die Ziel-Wahrscheinlichkeitsverteilungen, bis es eine Verteilung findet, die den gefundenen Werten am nächsten kommt. Da es hierfür keine überprüfbar beste Lösung gibt, haben Mathematiker Formeln entwickelt, die im Inferenzprozess ein Ziel darstellen können, auf das das Programm hinarbeiten kann.[2]

Als Ergebnis kann das Probabilistic Program dann unterschiedliche Fragen beantworten, je nachdem, was für Nutzer eines Probabilistic Programs von Interesse ist (Gordon et al., 2014): Zum einen kann das Programm den Wert herausgeben, den die Zufallsvariable am wahrscheinlichsten annehmen wird. Das Programm kann aber auch den Erwartungswert einer Zufallsvariablen herausgeben. Das entspricht dem nach den Wahrscheinlichkeiten gewichteten Mittel der möglichen Ergebniswerte. Der Erwartungswert kann also auch einen Wert annehmen, der dieser Menge nicht entstammt. Einer dieser beiden Werte wird zumeist das Ziel des Nutzers sein, der ungewisse Situationen abschätzen möchte und dazu wissen muss, welches Szenario am wahrscheinlichsten ist. Zuletzt ist es auch denkbar und möglich, dass Nutzer sich Stichproben aus der Verteilung geben lassen.

Probabilistic Programming verallgemeinert und abstrahiert, was zuvor aufwändig in Handarbeit getan werden musste: Üblicherweise muss der Algorithmus, der für ein Modell eine passende Wahrscheinlichkeitsverteilung berechnet, für jedes Modell neu konzipiert werden. Neben dem großen zeitlichen Aufwand ist dies auch eine mathematisch anspruchsvolle Aufgabe, die daher Spezialkenntnisse erfordert – und die nur wenige ausreichend beherrschen. Die Arbeit, den Algorithmus anzupassen und auszuführen, übernimmt nunmehr auf wenige Befehle hin das Probabilistic Program selbst. Damit ergibt sich die Chance, dass Wahrscheinlichkeitsberechnungen deutlich effizienter werden und eine breitere Zahl an Personen diese vornehmen kann.

Damit fällt dem Nutzer einzig die Aufgabe zu, das Modell zu erstellen, aus dem das Probabilistic Program Rückschlüsse ziehen kann. Eine Schwierigkeit ist dabei, dass das Modell flexibel genug sein muss, um Eigenheiten des Datensets umfassend abbilden zu können (Ghahramani, 2015). Zugleich muss es sensibel genug sein, damit das Programm die Eigenheiten erfasst, die es für die Bearbeitung seiner Aufgabe benötigt.

[2] Im folgenden Codebeispiel (III.2.) ist das die *evidence lower bound*, kurz ELBO.

Probabilistic Programming ist zwar nicht die einzige Art von Werkzeugen, mithilfe derer Systeme Machine Learning betreiben können. Seine Stärke spielt es jedoch dort aus, wo es wichtig ist, die Unsicherheit bei den Schlussfolgerungen, die das System zieht, abzubilden (Ghahramani, 2015). Gleichzeitig ist ein Vorteil gegenüber anderen Werkzeugen des Machine Learning, dass jederzeit ein Blick in die Einzelvorgänge von Modellteilen, in die Annahmen des Systems möglich ist (Ghahramani, 2015).

III. Umsetzung durch Pyro

Eine relativ neue Vertreterin der stetig wachsenden Familie von Probabilistic Programming Languages ist Pyro (Bingham et al., 2019). Da es über umfassende Einstiegshilfen und Tutorials verfügt und mit Inferenzalgorithmen ausgestattet ist, die sich mit nur wenigen genaueren Kenntnissen der Funktionsweise einsetzen lassen, eignet es sich gut für einen ersten Einstieg. Anhand von Pyro wollen wir im Folgenden die konkrete konzeptionelle Umsetzung der oben beschriebenen Besonderheiten von Probabilistic Programming Languages betrachten und sodann anhand eines Codebeispiels den grundsätzlichen Aufbau eines Probabilistic Programs nachvollziehen.

1. Ein modernes Probabilistic Programming Framework

Pyro baut auf Python und PyTorch auf und ist ein Deep Probabilistic Programming Framework. Es ist damit ein Werkzeug der neueren Generation, die Wahrscheinlichkeitsberechnung für Deep Learning verfügbar machen will, und als Unterstützung und Grundlage für die KI-Forschung gedacht. Uber AI Labs hat die Sprache mit vier grundlegenden Prinzipien als Ziele entworfen, die auch davon beeinflusst sind, dass die Sprache primär für Forscher, weniger für Programmierspezialisten gedacht ist: Zum einen soll sie expressiv sein, d.h. die Sprache ist im Idealfall daraufhin optimiert, dass Nutzer die nötigen Anweisungen an das Programm mit wenig Programmieraufwand erzeugen können. Das gilt gerade für die großen Mengen an Daten, die Programme effizient verarbeiten können sollen, sowie die komplexen Vernetzungen zwischen verschiedenen Variablen. Eng verwandt damit ist die Anforderung, dass Pyro minimal bleibt. Pyro führt effektiv nur wenige eigene Neuerungen und Befehle ein. Wo es möglich ist, nutzt Pyro umfassend bereits bestehende Werkzeuge. Beide Anforderungen zielen also vor allem darauf ab, die Arbeit für Forscher zu erleichtern, indem diese nur wenige neue Befehle erlernen und implementieren müssen. Weiterhin soll die Sprache skalierbar sein. Probabilistic Programming und die KI-Forschung hängen davon ab, dass sie mit großen Mengen an Daten umgehen können. Pyro löst dieses Problem u.a. über die Integration von PyTorch, das erlaubt, Graphics Processing Units (GPUs) für seine Berechnungen zu nutzen. Schließlich haben sich die Entwickler zum Ziel gesetzt, dass Pyro bei alldem flexibel bleibt. Hierin verbirgt sich zugleich noch einmal die Besonderheit von Probabilistic Programming Languages: Üblicherweise müssen Forscher die Inferenzalgorithmen für komplexe Modelle individuell für jede Fragestellung neu konfigurieren. Pyros Ziel ist es, mit wiederum wenigen, präzisen Befehlen den Inferenzprozess auf jedes Modell neu präzise zuschneiden zu können.

Pyro führt zwei neue primitive Typen ein: Der erste ist `pyro.sample()`, der als Parameter eine benannte Wahrscheinlichkeitsverteilung erhält und eine Stichprobe, also einen konkreten Wert aus dieser Verteilung liefert. Zusätzlich erhält bei Pyro jede Stichprobe `sample` einen

eigenen Namen. Der zweite primitive Typ ist `pyro.param()`, also Parameter, die ebenfalls einen eigenen Namen erhalten und die Form einer bestimmten Wahrscheinlichkeitsverteilung entscheidend mitgestalten. Sie können entweder fixiert sein oder vom System mithilfe von Daten trainiert werden (Michael, 2015).

2. Codebeispiel: Fairness einer Münze

Als simples Beispiel für die Funktionsweise eines Probabilistic Programs fungiert die Berechnung der Fairness einer Münze. Das Ergebnis eines Münzwurfs kann im Idealfall exakt einen von zwei Werten annehmen: Kopf oder Zahl. Die Fairness der Münze ist umso größer, je geringer die Differenz zwischen den beiden Wahrscheinlichkeiten für Kopf oder Zahl ist. Im Idealfall sind beide Wahrscheinlichkeiten exakt gleich groß. Wie wahrscheinlich das eine oder andere Ergebnis ist, steht der Münze allerdings nicht auf den Rand geschrieben. Die für die Fairness entscheidenden Eigenheiten der Münze und die wirkliche Wahrscheinlichkeitsverteilung können wir nur näherungsweise bestimmen. Für die Rechenarbeit können wir jedoch ein Probabilistic Program zuhilfenehmen. Das folgende Programm wertet die Fairness einer Münze nach zehn hypothetischen Münzwürfen aus. Es ist in Pyro, also basierend auf PyTorch und Python geschrieben und stammt aus den offiziellen Beispielen zum Framework (Uber Technologies, 2017):

```
1  import math
2  import torch
3  import torch.distributions.constraints as constraints
4  import pyro
5  from pyro.optim import Adam
6  from pyro.infer import SVI, Trace_ELBO
7  import pyro.distributions as dist
```

Zunächst importieren wir die nötigen Werkzeuge aus `math`, `torch` und `pyro` (Z. 1-7).

```
8  pyro.enable_validation(True)
9  pyro.clear_param_store()
10
11 data = []
12 for _ in range(6):
13     data.append(torch.tensor(1.0))
14 for _ in range(4):
15     data.append(torch.tensor(0.0))
```

Um sicherzugehen, dass vorherige Programmdurchläufe keinen Einfluss auf unser Ergebnis haben können, leeren wir als erstes unseren ParamStore (Z. 9). Dies ist erforderlich, da ParamStore ein globaler Speicher für alle Parameter ist, die einen Einfluss auf Wahrscheinlichkeitsverteilungen innerhalb unseres Programms ausüben können. In Pyro erhält jeder Parameter einen eindeutigen Namen. Der Wert eines benannten Parameters, der bereits im ParamStore enthalten ist, kann sich nur noch im Wege der Inferenz verändern. Daher lassen sich nur mithilfe eines initial leeren ParamStores Einflüsse durch zuvor vorhandene Parameter auf Wahrscheinlichkeitsverteilungen unseres Programms vollständig ausschließen. Anschließend (Z. 11-15) erstellen wir unsere Datenbasis, die zehn hypothetischen Münzwürfe: Dargestellt

durch Tensoren mit den Werten 1.0 und 0.0 teilen wir dem Probabilistic Program mit, dass unsere Münze nach 6 von 10 Würfen „Kopf“ (1.0) und nach 4 von 10 Würfen „Zahl“ (0.0) gezeigt hat.

```
16 def model(data):
17     alpha0 = torch.tensor(10.0)
18     beta0 = torch.tensor(10.0)
19 
20     f = pyro.sample("latent_fairness", dist.Beta(alpha0, beta0))
21     for i in range(len(data)):
22         pyro.sample("obs_{}".format(i), dist.Bernoulli(f), obs=data[i])
```

Nun können wir unser `model` definieren. Es soll die nötigen Ausgangsinformationen liefern, aus denen das Programm später Rückschlüsse ziehen kann, und benennt daher unsere gesuchte Verteilung (Z. 20) sowie die gemessenen Ergebnisse unserer Münzwürfe (Z. 22). Das `model` ist gleichsam der vorwärts laufende Teil unseres Programms: Mit dessen Hilfe erzeugt Pyro die Daten, die es als Grundlage für den Inferenzprozess benötigt. Unser Ziel ist es, herauszufinden, mit welcher Wahrscheinlichkeit die Münze Kopf oder Zahl zeigt, also wie die dazugehörige Bernoulliverteilung konkret aussieht. Die Erfolgswahrscheinlichkeit[3] ersetzen wir daher durch einen Parameter `f` (Z. 22), der als `sample` seinerseits aus einer Betaverteilung stammt (Z. 20). Die Betaverteilung wird in der Bayes-Statistik typischerweise als konjugierte A-priori-Wahrscheinlichkeitsverteilung für eine Bernoulliverteilung eingesetzt. Sie kann präzise die Wahrscheinlichkeiten für jede bestimmte Erfolgswahrscheinlichkeit der dazugehörigen Bernoulliverteilung abbilden. Daher stellt sie für unsere Zwecke die gesuchte, latente Zufallsvariable dar und wir nennen sie `latent_fairness`. Die beiden Parameter, die die Betaverteilung in ihrer Form charakterisieren, definieren wir in Z. 17-18. Da wir zunächst davon ausgehen, dass die Münze Kopf und Zahl genau gleich häufig anzeigt, wählen wir an dieser Stelle ausgeglichene Werte für `alpha0` und `beta0`.

Die gemessenen Ergebnisse unserer Münzwürfe fügen wir schließlich in Z. 22 mithilfe des `obs`-Arguments hinzu.[4] Diese werden im Inferenzprozess wie `sample`s aus der dort bezeichneten Bernoulliverteilung behandelt, auch wenn sie keinen zufälligen Wert annehmen, sondern den mit dem `obs`-Argument spezifizierten. Jede Verteilung, die einen der Datenpunkte als `sample` liefert, erhält – bei 10 Datenpunkten - einen eindeutigen Namen von `obs_0` bis `obs_9` (Z. 21-22).

```
23 def guide(data):
24     alpha_q = pyro.param("alpha_q", torch.tensor(15.0),
25                          constraint=constraints.positive)
26     beta_q = pyro.param("beta_q", torch.tensor(15.0),
27                         constraint=constraints.positive)
28     pyro.sample("latent_fairness", dist.Beta(alpha_q, beta_q))
```

[3] In unserem Fall also die Wahrscheinlichkeit, dass die Münze nach einem Wurf Kopf zeigt.

[4] Mithilfe des pyro.condition()-Befehls ist es auch möglich, die Bernoulliverteilung explizit zu konditionieren.

Der `guide` stellt eine erste Annäherung an die posteriore Verteilung (P(A|X)) dar. Er enthält daher anders als das `model` keine konkreten Datenpunkte, auch wenn wir ihm dennoch dieselben Daten als Input benennen müssen (Z. 23). Allerdings muss er für jedes `sample`-Statement des `model`, das nicht durch Datenpunkte konditioniert wurde, ein paralleles `sample`-Statement mit demselben Namen vorhalten. Das ist bei uns lediglich ein `sample` der `latent_fairness`-Verteilung (Z. 28). Während des Inferenzprozesses sind die Parameter `param` des `guide` diejenigen, die der Inferenzalgorithmus auf der Suche nach einer Verteilung verändert, die die gemessenen Werte und dem Prior optimal abbildet (`alpha_q` und `beta_q`, Z. 24-27).

```
29 adam_params = {"lr": 0.0005, "betas": (0.90, 0.999)}
30 optimizer = Adam(adam_params)
31 svi = SVI(model, guide, optimizer, loss=Trace_ELBO())
```

Bevor das Programm den Inferenzprozess beginnen kann, müssen wir nun noch einige Spezifikationen festlegen: Adam (Z. 30) bezeichnet einen Inferenzalgorithmus, der leicht zu implementieren ist, effizient rechnet sowie wenig Speicherplatz bedarf und in der Regel wenig Anpassung benötigt, um ein Problem gut lösen zu können (Kingma & Ba, 2017). Seine Parameter legen wir in Z. 29 fest. Zuletzt fassen wir alle geschaffenen Werkzeuge für das SVI-Interface zusammen (Z. 31). Neben unserem `model`, `guide` und `optimizer` legen wir hier noch die `loss`-Funktion fest, die im Inferenzprozess unsere Zielfunktion darstellt.

```
32 n_steps = 2000
33 for step in range(n_steps):
34     svi.step(data)
35     if step % 100 == 0:
36         print('.', end='')
```

Schließlich kann das Programm den Inferenzprozess durchführen. Wir legen fest, dass es 2000 Optimierungsschritte unternehmen wird, in denen es die Wahrscheinlichkeitsverteilung `latent_fairness` nach und nach immer genauer an die zuvor erzeugten Stichproben anpasst (Z. 32). Die Schritte selbst führt das Programm mit Z. 33-34 aus. Um das Fortschreiten des Inferenzprozesses darzustellen, lassen wir uns alle 100 Schritte einen Punkt ausgeben (Z. 35-36).

```
37 alpha_q = pyro.param("alpha_q").item()
38 beta_q = pyro.param("beta_q").item()
39
40 inferred_mean = alpha_q / (alpha_q + beta_q)
41 factor = beta_q / (alpha_q * (1.0 + alpha_q + beta_q))
42 inferred_std = inferred_mean * math.sqrt(factor)
43
44 print("\nbased on the data and our prior belief, the fairness " +
45       "of the coin is %.3f +- %.3f" % (inferred_mean, inferred_std))
```

Im Anschluss an den Inferenzprozess haben wir verschiedene Möglichkeiten, dessen Ergebnisse darzustellen. In diesem Fall entscheiden wir uns dafür, uns die beiden Parameter ausgeben zu lassen, die gemeinsam die Verteilung `latent_fairness` abbilden (Z. 37-38). Wir be-

rechnen aus diesen Werten nun noch unseren Mittelwert sowie die Standardabweichung unserer Verteilung (Z. 40-42) und lassen uns diese anschließend als Ergebnis ausgeben (Z. 44-45).

In Tab. 1 sehen wir exemplarisch einige Ergebnisse, die uns das Programm bei mehreren Programmdurchläufen geliefert hat. Da die Berechnungen des Programms entscheidend auch auf zufälligen Stichproben beruhen, die das Programm gezogen hat, weichen die errechneten Mittelwerte (links) leicht voneinander ab. Das ist also zu erwarten gewesen. Da unsere Münze bei zehn Münzwürfen häufiger „Kopf“ (1.0) als „Zahl“ (0.0) gezeigt hat, ist es ebenso einleuchtend, dass unser Mittelwert eine leichte Tendenz zu „Kopf“ anzeigt. Gleichzeitig zeigt der jeweilige Mittelwert in Verbindung mit der Standardabweichung, dass es auch gut möglich ist, dass die Wahrscheinlichkeiten für Kopf und Zahl exakt gleich groß sind. Die Standardabweichung (rechts) betrug stets 0.090. In anderen Probedurchläufen zu anderen Zeitpunkten hat das Programm vereinzelt auch ein anderes Ergebnis (0.089) für die Standardabweichungen geliefert. Da die Standardabweichung jedoch auch die Unsicherheit eines Modells beziffert, ist es ebenfalls zu erwarten gewesen, dass diese im Wesentlichen gleich bleibt: Die Unsicherheit hängt auch maßgeblich von der Zahl der Stichproben ab, die hier in allen Durchläufen gleich groß ist.

Mittelwert	Std.abw.
0.537	0.090
0.529	0.090
0.531	0.090
0.534	0.090
0.532	0.090
0.532	0.090

Tabelle 1: Ergebnisse unseres Probabilistic Programs

Insgesamt haben wir gesehen, dass der entscheidende Aufwand, ein Probabilistic Program mit Pyro zu erstellen, darin liegt, Daten und Verteilungen zutreffend zu modellieren. Hierzu ist es auch nötig, einige Grundkenntnisse aus der Statistik mitzubringen. Hingegen erfordert der Inferenzprozess selbst nur wenige Zeilen Code, da Pyro in diesem Zusammenhang den Schwerpunkt der Arbeit für seine Nutzer übernimmt. Darin zeigt sich zugleich der entscheidende Vorteil, den Probabilistic Programming Languages mit sich bringen: Gerade die Funktionsweise der Inferenzalgorithmen zu verstehen und sie für jedes neue Datenmodell individuell zu implementieren, bringt die entscheidenden Schwierigkeiten sowie Aufwand mit sich.

IV. Weitere Probabilistic Programming Languages

Auch wenn Probabilistic Programming erst seit einigen Jahren verstärkte Aufmerksamkeit und Forschung erfährt, gibt es heute bereits eine Vielzahl an Sprachen und Libraries, mit denen Nutzer Probabilistic Programming betreiben und die unterschiedliche Bedürfnisse erfüllen können. Häufig setzen sie als Libraries auf bereits bestehende Programmiersprachen mit ihren speziellen Befehlen auf. Eigenständige Sprachen sind hingegen seltener. Eine gut dokumentierte, häufig genutzte eigenständige Sprache ist bspw. Stan, die der Statistikforschung entstammt (Carpenter et al., 2017). Als eine weitere bekannte und besonders frühe Vertreterin der universell einsetzbaren Sprachen soll noch Church genannt werden, die auf dem Lisp-Dialekt Scheme und damit auf einem funktionalen Programmierparadigma aufbaut (Goodman et al., 2008).

Viele der neueren Sprachen sollen demgegenüber vor allem der Forschung im Machine Learning-Bereich dienen: Eine Sprache, die sich in den letzten Jahren zu einer der bedeutsamsten entwickelt hat, ist Edward, die auf TensorFlow aufbaut und mit Python nutzbar ist (Tran et al.,

2017). Sie firmiert ebenso wie Pyro unter dem Stichwort Deep Probabilistic Programming, womit gemeint ist, dass die Sprachen ihren wichtigsten Einsatz im Deep Learning finden sollen. Das bedeutet vor allem, dass Nutzer diese Sprachen für mehrschichtige („deep“) Netzwerke von Zufallsvariablen einsetzen können (vgl. Tran et al., 2017). Qualitative Unterschiede zwischen Pyro und Edward liegen darin, dass Edward einige Inferenzalgorithmen besser unterstützt als Pyro, während sich Pyro als expressiver gegenüber Edward erweist (Baudart et al., 2018). Neben den genannten Beispielen ist eine große Anzahl weiterer Sprachen leicht auffindbar (z.B. über Ghahramani, 2015; van de Meent et al., 2018). Sie wird voraussichtlich auch in den kommenden Jahren noch weiter wachsen.

V. Fazit und Ausblick

Nicht ohne Grund haben die Ideen, die unter dem Namen Probabilistic Programming zirkulieren, mittlerweile weite Verbreitung in einer Vielzahl von Sprachen und Anwendungsgebieten gefunden. Probabilistic Programming verspricht, Aufgaben, die sich mit der Bemessung von Wahrscheinlichkeiten und Unsicherheiten befassen, insbesondere auch das Machine Learning, erheblich zu vereinfachen.

Dabei ist das zugrundeliegende Konzept des Probabilistic Programming so bestechend wie simpel: Um einen ungewissen Zustand, ob aktuell oder zukünftig, einschätzen zu können, stellt man diesen zunächst als Zufallszahl mit möglichen Werten, die für die möglichen realen Zustände stehen, dar. Welchen Wert die Zufallszahl am wahrscheinlichsten annimmt, bildet eine Wahrscheinlichkeitsverteilung ab. Die Probabilistic Programs setzen die grundsätzlich stets gleich vorgeprägten Inferenzalgorithmen, die daher automatisiert arbeiten können, sodann ohne stärkeres Zutun der Nutzerin ein. Wirkungsvoll werden die Programme, indem man die Wahrscheinlichkeiten mit einer großen Menge an Daten weiter präzisiert und mit weiteren Faktoren, ggf. ebenfalls als Zufallszahlen dargestellt, in Verbindung bringt.

Das Codebeispiel in Pyro hat gezeigt, dass die Nutzerin die komplizierten Einzelheiten des Inferenzprozesses dem Programm überlassen kann. Dafür obliegt es weiterhin weitgehend der Nutzerin, das zugrundeliegende Modell zu erstellen. Hierfür werden auch zukünftig Kenntnisse über Statistik nötig sein, um die Daten mit einer A-priori-Annahme zusammenbringen zu können, sodass das Programm damit arbeiten kann. Dennoch erweist sich Pyro wie angekündigt als sehr expressiv. Die Sprache ist aktuell stark in der Entwicklung, sodass ihre Entwickler sie stetig erneuern und verbessern. Das illustriert auch die jüngste Erweiterung durch NumPyro (Phan et al., 2019).

Dass sich das Probabilistic Programming noch in einer eher frühen Entwicklungsphase befindet, zeigt sich auch daran, dass nicht nur für Pyro, sondern generell eher wenige ausführliche Erklärungen zu finden sind. Auch wenn Forscher zwischenzeitlich sogar Systeme entwickelt haben, mithilfe derer sie den Einstieg in die Thematik erleichtern wollen (Gorinova et al., 2016), sind gewisse Grundkenntnisse in der Statistik weiterhin vonnöten. Wer komplexere Fragen mithilfe von Probabilistic Programming beantworten will, muss ebenfalls weiterhin ein tieferes Verständnis von der Materie mitbringen, um die zweifellos mächtigen Werkzeuge zielgenau einsetzen zu können. Dafür müssen Benutzer Inferenzalgorithmen nicht mehr im Einzelnen bis ins Detail beherrschen und implementieren. Während also Probabilistic Programming eine deutliche Arbeitserleichterung mit sich bringt, muss es seinen Anspruch an sich

selbst, Bereiche wie das Machine Learning durch einen erleichterten Einstieg zu demokratisieren, noch einlösen. Dessen ungeachtet ist davon auszugehen, dass sich das Probabilistic Programming auch in den nächsten Jahren weiterhin stark entwickeln wird und daher ein attraktives Gebiet bleibt für alle, die in einem dynamischen Feld an komplexen Fragestellungen mitwirken wollen.

VI. Literaturverzeichnis

Guillaume Baudart, Martin Hirzel, Louis Mandel, 2018: **Deep Probabilistic Programming Languages: A Qualitative Study**: arXiv:1804.06458v1

Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus A. Brubaker, Jiqiang Guo, Peter Li, Allen Riddell, 2017: **Stan: A Probabilistic Programming Language**: Journal of Statistical Software, Vol. 76, 2017, doi: 10.18637/jss.v076.i01

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, Noah D. Goodman, 2019: **Pyro: Deep Universal Probabilistic Programming**: Journal of Machine Learning Research 20 (2019), 1-6

Marco Chianese, 2019: **Differentiable Probabilistic Programming for Strong Gravitational Lensing**: arXiv:1910.06617v1

Beau Cronin, 2013: **What Is Probabilistic Programming?**: https://www.oreilly.com/content/probabilistic-programming/

Cam Davidson-Pilon, 2015**: Bayesian Methods for Hackers**: https://nbviewer.jupyter.org/github/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers/tree/master/

Zoubin Ghahramani, 2015: **Probabilistic machine learning and artificial intelligence**: Nature Vol. 521, 452-459

Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, Joshua B. Tenenbaum, 2008: **Church: A language for generative models**: Proc. 24th Conf. Uncertainty in Artificial Intelligence (UAI), 220–229, arXiv:1206.3255v2

Maria I. Gorinova, Advait Sarkar, Alan F. Blackwell, Don Syme, 2016: **A Live, Multiple-Representation Probabilistic Programming Environment for Novices**: CHI '16: Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, 2533–2537

Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, Sriram K. Rajamani, 2014: **Probabilistic Programming**: https://hci-kdd.org/wordpress/wp-content/uploads/2016/10/GORDON-HENZINGER-NORI-RAJAMANI-2014-Probabilistic-Programming.pdf

Bradley J. Gram-Hansen, Christian Schröder de Witt, Tom Rainforth, Philip H.S. Torr, Yee Whye Teh, Atılım Güneş Baydin, 2019: **Hijacking Malaria Simulators with Probabilistic Programming**: arXiv:1905.12432v1

Ralf Herbrich, Tom Minka, Thore Graepel, 2006: **TrueSkill(TM): A Bayesian Skill Rating System**: Proceedings of the 19th International Conference on Neural Information Processing Systems, 2006, 569–576

Diederik P. Kingma, Jimmy Lei Ba, 2017: **Adam: A Method for Stochastic Optimization**: arXiv:1412.6980v9

Tejas D. Kulkarni, Pushmeet Kohli, Joshua B. Tenenbaum, Vikash Mansinghka, 2015: **Picture: A Probabilistic Programming Language for Scene Perception**: https://openaccess.thecvf.com/content_cvpr_2015/papers/Kulkarni_Picture_A_Probabilistic_2015_CVPR_paper.pdf

Richard Michael, 2015: **Single-Parameter Models | Pyro vs. STAN**: https://towardsdatascience.com/single-parameter-models-pyro-vs-stan-e7e69b45d95c

Du Phan, Neeraj Pradhan, Martin Jankowiak, 2019: **Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro**: arXiv:1912.11554v1

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, Frank Wood, 2018: **An Introduction to Probabilistic Programming**: arXiv:1809.10756v1

Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, David M. Blei, 2017: **Deep Probabilistic Programming**: arXiv:1701.03757v2

Uber Technologies, Inc, 2017-2018: **SVI Part I: An Introduction to Stochastic Variational Inference in Pyro**: https://pyro.ai/examples/svi_part_i.html