KAPITEL 2

Tensoren

Bevor wir tief in die Welt der PyTorch-Entwicklung eintauchen, ist es wichtig, sich mit der grundlegenden Datenstruktur in PyTorch vertraut zu machen: dem `torch.Tensor`. Wenn Sie den Tensor verstehen, werden Sie auch begreifen, wie PyTorch Daten verarbeitet und speichert, und da bei Deep Learning im Wesentlichen Gleitkommazahlen gesammelt und manipuliert werden, verstehen Sie ebenfalls, wie PyTorch anspruchsvollere Funktionen für Deep Learning implementiert. Darüber hinaus werden Sie häufig auf Tensoroperationen zurückgreifen, wenn Sie bei der Modellentwicklung Eingabedaten vorverarbeiten oder Ausgabedaten manipulieren.

Dieses Kapitel dient als Kurzreferenz, sodass Sie Tensoren verstehen und Tensorfunktionen in Ihrem Code implementieren können. Zunächst beschreibe ich, was ein Tensor ist, und zeige Ihnen einige einfache Beispiele dazu, wie sich Tensoroperationen auf einer GPU mit entsprechenden Funktionen erstellen, manipulieren und beschleunigen lassen. Als Nächstes werfen wir einen umfassenden Blick auf die API, um Tensoren zu erzeugen und mathematische Operationen durchzuführen, sodass Sie schnell auf eine ausführliche Liste von Tensorfähigkeiten verweisen können. In jedem Abschnitt untersuchen wir einige der wichtigeren Funktionen, zeigen Schwerpunkte ihrer Verwendung und weisen auf häufige Fallstricke hin.

Was ist ein Tensor?

In PyTorch ist ein Tensor eine Datenstruktur, die verwendet wird, um Daten zu speichern und zu manipulieren. Wie ein NumPy-Array ist ein Tensor ein mehrdimensionales Array, das Elemente eines einzigen Datentyps enthält. Mit Tensoren lassen sich Skalare, Vektoren, Matrizen und n-dimensionale Arrays darstellen. Abgeleitet werden sie von der Klasse torch.Tensor. Allerdings sind Tensoren mehr als nur Arrays von Zahlen. Wenn Sie ein Tensorobjekt von der Klasse torch.Tensor erstellen oder instanziieren, erhalten Sie Zugriff auf eine Reihe integrierter Klassenattribute und Operationen oder Klassenmethoden, die einen robusten Satz von integrierten Fähigkeiten bereitstellen. Dieses Kapitel beschreibt dieses Attribute und Operationen im Detail.

Tensoren können auch zusätzliche Vorteile bieten, die sie für Berechnungen im Bereich Deep Learning geeigneter machen als NumPy-Arrays. Erstens können Tensoroperationen per GPU-Beschleunigung deutlich schneller ausgeführt werden. Zweitens lassen sich Tensoren mit verteilter Verarbeitung auf mehreren CPUs und GPUs über mehrere Server hinweg in großem Umfang speichern und manipulieren. Und drittens verfolgen Tensoren ihre Berechnungsgraphen, was bei der Implementierung einer Deep-Learning-Bibliothek sehr wichtig ist, wie der Abschnitt »Automatische Differentiation (Autograd)« auf Seite 49 zeigt.

Zur weitergehenden Erklärung, was ein Tensor tatsächlich ist und wie man einen Tensor verwendet, gehe ich zunächst ein einfaches Beispiel durch, das einige Tensoren erzeugt und eine Tensoroperation durchführt.

Ein einfaches CPU-Beispiel

Hier ist ein einfaches Beispiel, das einen Tensor erzeugt, eine Tensoroperation durchführt und eine integrierte Methode auf dem Tensor selbst verwendet. Standardmäßig wird der Tensordatentyp aus dem Eingabedatentyp abgeleitet und der Tensor dem CPU-Gerät zugeordnet. Zuerst importieren wir die PyTorch-Bibliothek und erstellen dann zwei Tensoren `x` und `y` aus zweidimensionalen Listen. Als Nächstes addieren wir die beiden Tensoren und speichern das Ergebnis in `z`. Da die Klasse `torch.Tensor` das Überladen von Operatoren unterstützt, können wir hier einfach den Operator `+` verwenden. Schließlich geben wir den neuen Tensor `z` aus, der die Matrixsumme von `x` und `y` ist, und wir geben die Größe von `z` aus. Beachten Sie, dass `z` selbst ein Tensorobjekt ist und die Methode `size()` die Matrixdimensionen – nämlich 2 × 3 – zurückgibt:

```
import torch

x = torch.tensor([[1,2,3],[4,5,6]])
y = torch.tensor([[7,8,9],[10,11,12]])
z = x + y
print(z)
# out:
#  tensor([[ 8, 10, 12],
#          [14, 16, 18]])

print(z.size())
# out: torch.Size([2, 3])
```

In Legacy-Code finden Sie gegebenenfalls den Konstruktor `torch.Tensor()` mit großem T. Dabei handelt es sich um einen Alias für den standardmäßigen Tensortyp `torch.FloatTensor`. Sie sollten stattdessen `torch.tensor()` verwenden, um Ihre Tensoren zu erstellen.

Ein einfaches GPU-Beispiel

Gegenüber NumPy-Arrays bieten Tensoren den großen Vorteil, dass sich die Tensoroperationen auf einer GPU beschleunigen lassen. Ein einfaches Beispiel soll das veranschaulichen. Es handelt sich um das gleiche Beispiel wie im letzten Abschnitt, aber hier verschieben wir die Tensoren auf das GPU-Gerät, wenn eines verfügbar ist. Der ausgegebene Tensor wird ebenfalls der GPU zugeordnet. Mit dem Geräteattribut (z.B. `z.device`) können Sie kontrollieren, wo sich der Tensor befindet.

Die Funktion `torch.cuda.is_available()` in der ersten Codezeile gibt `True` zurück, wenn Ihr Computer über GPU-Unterstützung verfügt. Auf diese komfortable Art und Weise lässt sich robusterer Code schreiben, der beschleunigt werden kann, wenn eine GPU vorhanden ist, aber auch auf einer CPU läuft, wenn keine GPU existiert. In der Ausgabe zeigt `device='cuda:0'` an, dass die erste GPU verwendet wird. Sind in Ihrem Computer mehrere GPUs installiert, können Sie auch steuern, welche GPU zum Einsatz kommt:

```
device = "cuda" if torch.cuda.is_available()
  else "cpu"
x = torch.tensor([[1,2,3],[4,5,6]],
                 device=device)
y = torch.tensor([[7,8,9],[10,11,12]],
                 device=device)
z = x + y
print(z)
# out:
# tensor([[ 8, 10, 12],
#         [14, 16, 18]], device='cuda:0')

print(z.size())
# out: torch.Size([2, 3])

print(z.device)
# out: cuda:0
```

Tensoren zwischen CPUs und GPUs verschieben

Der vorherige Code erzeugt mit `torch.tensor()` einen Tensor auf einem bestimmten Gerät. Es ist jedoch üblicher, einen vorhandenen Tensor auf ein Gerät zu verschieben, nämlich auf eine GPU, falls verfügbar. Dies bewerkstelligen Sie mit der Methode `torch.to()`. Wenn als Ergebnis von Tensoroperationen neue Tensoren erzeugt werden, legt PyTorch den neuen Tensor auf demselben Gerät an. Im folgenden Code befindet sich `z` auf der GPU, weil `x` und `y` auf der GPU angelegt wurden. Der Tensor `z` wird mittels `torch.to("cpu")` zur weiteren Verarbeitung zurück auf die CPU verschoben. Beachten Sie auch, dass sich alle Tensoren innerhalb der Operation auf demselben Gerät befinden müssen. Wenn Sie nämlich `x` auf der GPU und `y` auf der CPU unterbringen, ernten Sie eine Fehlermeldung:

```
device = "cuda" if torch.cuda.is_available()
  else "cpu"
x = x.to(device) y = y.to(device) z = x + y
z = z.to("cpu")
# out:
# tensor([[ 8, 10, 12],
#   [14, 16, 18]])
```

Als Geräteparameter können Sie direkt Strings anstelle von Geräteobjekten angeben. Die folgenden Anweisungen sind alle gleichwertig:

- `device="cuda"`
- `device=torch.device("cuda")`
- `device="cuda:0"`
- `device=torch.device("cuda:0")`

Tensoren erstellen

Der vorherige Abschnitt hat eine einfache Möglichkeit gezeigt, Tensoren zu erstellen. Allerdings lässt sich dies auf noch viele weitere Arten bewerkstelligen. So kann man Tensoren aus bereits vorhandenen numerischen Daten oder aus Zufallsstichproben erzeugen. Die bereits vorhandenen Daten können in Array-ähnlichen Strukturen wie Listen, Tupeln, Skalaren oder serialisierten Datendateien wie auch in NumPy-Arrays gespeichert sein, um daraus Tensoren zu erstellen.

Der folgende Code veranschaulicht einige gängige Methoden, um Tensoren zu erstellen. Als Erstes wird ein Tensor mittels `torch.tensor()` aus einer Liste erzeugt. Diese Methode kann auch eingesetzt werden, um Tensoren aus anderen Datenstrukturen wie Tupeln, Mengen oder NumPy-Arrays zu erzeugen:

```
import numpy

# Aus bereits vorhandenen Arrays erzeugt.
w = torch.tensor([1,2,3]) ❶
w = torch.tensor((1,2,3)) ❷
w = torch.tensor(numpy.array([1,2,3])) ❸

# Nach der Größe initialisiert.
w = torch.empty(100,200) ❹
w = torch.zeros(100,200) ❺
w = torch.ones(100,200)  ❻
```

❶ Aus einer Liste.

❷ Aus einem Tupel.

❸ Aus einem NumPy-Array.

❹ Uninitialisiert, Elementwerte sind nicht vorhersagbar.

❺ Alle Elemente mit `0.0` initialisiert.

❻ Alle Elemente mit `1.0` initialisiert.

Wie im vorherigen Codebeispiel gezeigt, können Sie Tensoren auch mit Funktionen wie `torch.empty()`, `torch.ones()` und `torch.zeros()` erstellen und die gewünschte Größe angeben.

Um einen Tensor mit zufälligen Werten zu initialisieren, können Sie in PyTorch auf eine Reihe von robusten Funktionen zurückgreifen, wie `torch.rand()`, `torch.randn()` und `torch.randint()`. Der folgende Code zeigt dazu ein Beispiel:

```
# Nach Größe mit Zufallswerten initialisiert.
w = torch.rand(100,200)           ❶
w = torch.randn(100,200)          ❷
w = torch.randint(5,10,(100,200)) ❸

# Mit angegebenem Datentyp oder Gerät initialisiert.
w = torch.empty((100,200), dtype=torch.float64,
                device="cuda")

# Mit gleicher Größe, gleichem Datentyp und gleichem
# Gerät wie ein anderer Tensor initialisiert.
x = torch.empty_like(w)
```

❶ Erzeugt einen 100 × 200-Tensor mit Elementen aus einer Gleichverteilung im Intervall `[0, 1)`.

❷ Elemente sind Zufallszahlen aus einer Normalverteilung mit einem Mittelwert von 0 und einer Varianz von 1.

❸ Elemente sind zufällige Ganzzahlen zwischen 5 und 10.

Bei der Initialisierung können Sie den Datentyp und das Gerät (z.B. CPU oder GPU) angeben, wie es das vorherige Codebeispiel gezeigt hat. Außerdem geht aus dem Beispiel hervor, wie Sie mit PyTorch Tensoren erzeugen, die die gleichen Eigenschaften wie andere Tensoren aufweisen, aber mit anderen Daten initialisiert werden. Funktionen mit dem Postfix `_like`, wie `torch.empty_like()` und `torch.ones_like()`, geben Tensoren zurück, die in Größe, Datentyp und Gerät mit einem anderen Tensor übereinstimmen, aber anders initialisiert werden (siehe »Tensoren aus zufälligen Stichproben erstellen« auf Seite 39).

Es gibt einige Legacy-Funktionen wie `from_numpy()` und `as_tensor()`, die in der Praxis durch den Konstruktor `torch.tensor()` ersetzt wurden, mit dem sich alle Fälle abhandeln lassen.

Tabelle 2-1 listet PyTorch-Funktionen auf, mit denen sich Tensoren erzeugen lassen. Die Funktionen sollten Sie jeweils mit dem Namespace `torch` verwenden, wie zum Beispiel `torch.empty()`. Weitere Details finden Sie in der Tensordokumentation von PyTorch (*https://pytorch.tips/torch*).

Tabelle 2-1: Funktionen zum Erstellen von Tensoren

Funktion	Beschreibung
`torch.`**`tensor`**`(`*`data, dtype=None, device=None, requires_grad=False, pin_memory=False`*`)`	Erstellt einen Tensor aus einer vorhandenen Datenstruktur.
`torch.`**`empty`**`(`*`*size, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False`*`)`	Erstellt einen Tensor aus uninitialisierten Elementen basierend auf dem zufälligen Zustand von Werten im Arbeitsspeicher.
`torch.`**`zeros`**`(`*`*size, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False`*`)`	Erstellt einen Tensor und initialisiert alle Elemente mit `0.0`.
`torch.`**`ones`**`(`*`*size, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False`*`)`	Erstellt einen Tensor und initialisiert alle Elemente mit `1.0`.
`torch.`**`arange`**`(`*`start=0, end, step=1, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False`*`)`	Erstellt einen eindimensionalen Tensor mit Werten über einem Bereich mit einem gemeinsamen Schrittwert.
`torch.`**`linspace`**`(`*`start, end, steps=100, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False`*`)`	Erstellt einen eindimensionalen Tensor mit Punkten in linearen Abständen zwischen `start` und `end`.
`torch.`**`logspace`**`(`*`start, end, steps=100, base=10.0, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False`*`)`	Erstellt einen eindimensionalen Tensor mit Punkten in logarithmischen Abständen zwischen `start` und `end`.
`torch.`**`eye`**`(`*`n, m=None, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False`*`)`	Erstellt einen zweidimensionalen Tensor mit Einsen in der Diagonalen und Nullen an den übrigen Positionen.
`torch.`**`full`**`(`*`size, fill_value, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False`*`)`	Erzeugt einen Tensor, der mit `fill_value` gefüllt ist.
`torch.`**`load`**`(f)`	Lädt einen Tensor aus einer serialisierten Pickle-Datei.
`torch.`**`save`**`(f)`	Speichert einen Tensor in einer serialisierten Pickle-Datei.

Die PyTorch-Dokumentation enthält eine vollständige Liste von Funktionen zum Erstellen von Tensoren sowie weitere detaillierte Erläuterungen dazu, wie man sie verwendet. Die folgenden Punkte nennen häufige Fallstricke und zusätzliche Erkenntnisse, die Sie beim Erstellen von Tensoren beachten sollten:

- Die meisten Erstellungsfunktionen übernehmen die optionalen Parameter `dtype` und `device`, sodass Sie diese Werte zur Erstellungszeit festlegen können.
- Verwenden Sie `torch.arange()` anstelle der veralteten Funktion `torch.range()`. Nutzen Sie `torch.arange()`, wenn die Schrittweite bekannt ist, und `torch.linspace()`, wenn Sie die Anzahl der Elemente kennen.
- Mit `torch.tensor()` können Sie Tensoren aus Array-ähnlichen Strukturen wie Listen, NumPy-Arrays, Tupeln und Mengen erstellen. Vorhandene Tensoren konvertieren Sie mit `torch.numpy()` und `torch.tolist()` in NumPy-Arrays bzw. -Listen.

Tensorattribute

Zur Beliebtheit von PyTorch trägt die Eigenschaft bei, dass die Bibliothek stark auf Python ausgerichtet und von Haus aus objektorientiert ist. Da ein Tensor sein eigener Datentyp ist, können Sie Attribute des Tensorobjekts selbst lesen. Zudem ist es hilfreich, Informationen über die erstellten Tensoren zu ermitteln. Das geschieht über Attribute. Wenn `x` ein Tensor ist, können Sie auf mehrere Attribute von `x` wie folgt zugreifen:

`x.dtype`
: Gibt den Datentyp des Tensors an (siehe Tabelle 2-2 mit einer Liste der PyTorch-Datentypen).

`x.device`
: Gibt den Geräteort des Tensors an (z. B. CPU- oder GPU-Speicher).

`x.shape`
: Zeigt die Dimensionen des Tensors an.

`x.ndim`
: Gibt die Anzahl der Dimensionen oder den Rang eines Tensors an.

`x.requires_grad`
: Ein boolesches Attribut, das anzeigt, ob der Tensor die Berechnungsgraphen verfolgt (siehe »Automatische Differentiation (Autograd)« auf Seite 49).

`x.grad`
: Enthält die eigentlichen Gradienten, wenn `requires_grad` auf `True` gesetzt ist.

`x.grad_fn`
: Speichert die Funktion des Berechnungsgraphen, wenn `requires_grad` auf `True` gesetzt ist.

`x.s_cuda, x.is_sparse, x.is_quantized, x.is_leaf, x.is_mkldnn`
: Boolesche Attribute, die anzeigen, ob der Tensor bestimmte Bedingungen erfüllt.

`x.layout`
: Zeigt an, wie ein Tensor im Arbeitsspeicher angeordnet ist.

Denken Sie daran, dass Sie beim Zugriff auf Objektattribute keine runden Klammern angeben, wie es bei einer Klassenmethode erforderlich wäre (verwenden Sie zum Beispiel `x.shape` und nicht `x.shape()`).

Datentypen

Bei der Entwicklung im Rahmen von Deep Learning müssen Sie auf den Datentyp achten, den Ihre Daten und die entsprechenden Berechnungen nutzen. Wenn Sie also Tensoren erstellen, sollten Sie steuern, welche Datentypen verwendet werden. Wie bereits erwähnt, müssen alle Tensorelemente vom selben Datentyp sein. Den Datentyp können Sie mit dem Parameter `dtype` festlegen, wenn Sie den Tensor erstel-

len. Es ist auch möglich, einen Tensor mit der passenden Typumwandlungsmethode oder der Methode `to()` in einen neuen `dtype` zu konvertieren, wie es der folgende Code zeigt:

```
# Den Datentyp beim Erstellen mit dtype festlegen.
w = torch.tensor([1,2,3], dtype=torch.float32)

# Die Typumwandlungsmethode verwenden, um in einen neuen Datentyp zu konvertieren.
w.int()      # w bleibt nach der Typumwandlung ein float32.
w = w.int()  # w wird nach der Typumwandlung in einen int32 geändert.

# Mit der Methode to() in einen neuen Typ konvertieren.
w = w.to(torch.float64)       ❶
w = w.to(dtype=torch.float64) ❷

# Python konvertiert Datentypen in Operationen automatisch.
x = torch.tensor([1,2,3], dtype=torch.int32)
y = torch.tensor([1,2,3], dtype=torch.float32)
z = x + y  ❸
print(z.dtype)
# out: torch.float32
```

❶ Der Datentyp wird übergeben.

❷ Den Datentyp wird direkt mit `dtype` definiert.

❸ Python konvertiert `x` automatisch in `float32` und gibt `z` als `float32` zurück.

Beachten Sie, dass die Typumwandlungs- und `to()`-Methoden den Datentyp des Tensors nicht ändern, außer wenn Sie den Tensor neu zuweisen. Und bei Operationen mit gemischten Datentypen wandelt PyTorch die Tensoren automatisch in den passenden Typ um.

Bei den meisten Funktionen zum Erstellen von Tensoren können Sie den Datentyp beim Erstellen mit dem Parameter `dtype` festlegen. Achten Sie darauf, den Namespace `torch` zu verwenden, wenn Sie den `dtype` festlegen oder den Typ von Tensoren umwandeln (z. B. `torch.int64` und nicht einfach `int64`).

Tabelle 2-2 listet alle in PyTorch verfügbaren Datentypen auf. Jeder Datentyp führt, abhängig vom Gerät des Tensors, zu einer anderen Tensorklasse. Die entsprechenden Tensorklassen sind in den beiden Spalten ganz rechts für CPUs bzw. GPUs angegeben.

Tabelle 2-2: Tensordatentypen

Datentyp	dtype	CPU-Tensor	GPU-Tensor
Gleitkomma 32 Bit (Standard)	`torch.float32` oder `torch.float`	`torch.FloatTensor`	`torch.cuda.FloatTensor`
Gleitkomma 64 Bit	`torch.float64` oder `torch.double`	`torch.DoubleTensor`	`torch.cuda.DoubleTensor`
Gleitkomma 16 Bit	`torch.float16` oder `torch.half`	`torch.HalfTensor`	`torch.cuda.HalfTensor`

Tabelle 2-2: Tensordatentypen (Fortsetzung)

Datentyp	dtype	CPU-Tensor	GPU-Tensor
Ganzzahl 8 Bit (vorzeichenlos)	`torch.uint8`	`torch.ByteTensor`	`torch.cuda.ByteTensor`
Ganzzahl 8 Bit (vorzeichenbehaftet)	`torch.int8`	`torch.CharTensor`	`torch.cuda.CharTensor`
Ganzzahl 16 Bit (vorzeichenbehaftet)	`torch.int16` oder `torch.short`	`torch.ShortTensor`	`torch.cuda.ShortTensor`
Ganzzahl 32 Bit (vorzeichenbehaftet)	`torch.int32` oder `torch.int`	`torch.IntTensor`	`torch.cuda.IntTensor`
Ganzzahl 64 Bit (vorzeichenbehaftet)	`torch.int64` oder `torch.long`	`torch.LongTensor`	`torch.cuda.LongTensor`
Boolean	`toorch.bool`	`torch.BoolTensor`	`torch.cuda.BoolTensor`

Um die Komplexität von Speicherplatz zu verringern, werden Sie manchmal Arbeitsspeicher wiederverwenden und Tensorwerte mit *In-Place-Operationen* überschreiben wollen. Um In-Place-Operationen auszuführen, fügen Sie den Unterstrich (_) als Postfix an den Funktionsnamen an. Zum Beispiel addiert die Funktion `y.add_(x)` den Wert von `x` zu `y`, speichert das Ergebnis aber in `y`.

Tensoren aus zufälligen Stichproben erstellen

Während der Deep-Learning-Entwicklung sind oftmals Zufallsdaten zu erzeugen. Manchmal muss man Gewichte mit zufälligen Werten initialisieren oder zufällige Eingaben mit bestimmten Verteilungen erzeugen. PyTorch unterstützt einen sehr robusten Satz von Funktionen, mit denen Sie Tensoren aus Zufallsdaten erstellen können.

Wie bei anderen Erstellungsfunktionen können Sie den Datentyp und das Gerät festlegen, wenn Sie den Tensor erstellen. Tabelle 2-3 enthält einige Beispiele für Stichprobenfunktionen.

Tabelle 2-3: Stichprobenfunktionen

Funktion	Beschreibung
`torch.`**`rand`**`(*size, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False)`	Wählt zufällige Werte aus einer Gleichverteilung im Intervall [0, 1] aus.
`torch.`**`randn`**`(*size, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False)`	Wählt zufällige Werte aus einer standardmäßigen Normalverteilung mit dem Mittelwert 0 und der Varianz 1 aus.
`torch.`**`normal`**`(mean, std, *, generator=None, out=None)`	Wählt Zufallszahlen aus einer Normalverteilung mit gegebenem Mittelwert und gegebener Varianz aus.
`torch.`**`randint`**`(low=0, high, size, *, generator=None, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False)`	Wählt zufällige Ganzzahlen aus, die gleichförmig zwischen den unteren und oberen Werten generiert werden.

Tabelle 2-3: Stichprobenfunktionen (Fortsetzung)

Funktion	Beschreibung
`torch.`**`randperm`**`(n, out=None, dtype=torch.int64, layout=torch.strided, device=None, requires_grad=False)`	Erzeugt eine zufällige Permutation von Ganzzahlen aus dem Bereich von 0 bis *n* – 1.
`torch.`**`bernoulli`**`(input, *, generator=None, out=None)`	Zieht binäre Zufallszahlen (0 oder 1) aus einer Bernoulli-Verteilung.
`torch.`**`multinomial`**`(input, num_samples, replacement=False, *, generator=None, out=None)`	Wählt eine Zufallszahl aus einer Liste entsprechend den Gewichten aus einer Multinomialverteilung aus.

Tensoren können Sie auch von Werten erzeugen, die aus komplexeren Verteilungen gezogen werden, wie Cauchy-, Exponential-, geometrischen und Log-Normal-Verteilungen. Hierzu erzeugen Sie mit `torch.empty()` den Tensor und wenden eine In-Place-Funktion für die Verteilung (z. B. Cauchy) an. Denken Sie daran, dass In-Place-Methoden das Unterstrich-Postfix (_) verwenden. Zum Beispiel erzeugt `x = torch.empty([10,5]).cauchy_()` einen Tensor mit Zufallszahlen, die aus der Cauchy-Verteilung gezogen werden.

Tensoren wie andere Tensoren erstellen

Vielleicht möchten Sie einen Tensor, der ähnliche Eigenschaften – einschließlich `dtype`, `device` und `layout` – besitzt wie ein anderer Tensor, erstellen und initialisieren, um Berechnungen zu erleichtern. Viele der Erstellungsoperationen für Tensoren verfügen über eine Ähnlichkeitsfunktion, mit der Sie dies leicht bewerkstelligen können. Die Ähnlichkeitsfunktionen sind mit dem Postfix `_like` gekennzeichnet. So erstellt `torch.empty_like(tensor_a)` einen leeren Tensor mit den Eigenschaften `dtype`, `device` und `layout` von `tensor_a`. Beispiele für Ähnlichkeitsfunktionen sind `empty_like()`, `zeros_like()`, `ones_like()`, `full_like()`, `rand_like()`, `randn_like()` und `rand_int_like()`.

Tensoroperationen

Nachdem Sie wissen, wie Sie Tensoren erstellen, untersuchen wir, was man mit ihnen anstellen kann. PyTorch unterstützt einen robusten Satz von Tensoroperationen, mit denen Sie auf Ihre Tensordaten zugreifen und diese transformieren können.

Zuerst beschreibe ich, wie man auf Teile der Daten zugreift, ihre Elemente manipuliert und Tensoren kombiniert, um neue Tensoren zu bilden. Dann zeige ich Ihnen, wie Sie einfache Berechnungen sowie anspruchsvollere mathematische Berechnungen durchführen können, und zwar oftmals in konstanter Zeit. PyTorch bietet viele integrierte Funktionen. Es ist hilfreich, sich über die verfügbaren Funktionen zu informieren, bevor Sie eigene Funktionen schreiben.

Tensoren indizieren, slicen, kombinieren und aufteilen

Nachdem Sie Tensoren erstellt haben, werden Sie auf Teile der Daten zugreifen und Tensoren kombinieren oder aufteilen wollen, um neue Tensoren zu bilden. Der folgende Code demonstriert, wie Sie derartige Operationen durchführen. Tensoren können Sie in der gleichen Weise slicen und indizieren, wie Sie NumPy-Arrays slicen und indizieren. Die ersten Zeilen des folgenden Codes zeigen dies. Indizieren und Slicen geben Tensoren zurück, selbst wenn das Array nur ein einzelnes Element enthält. Um einen einelementigen Tensor in einen Python-Wert zu konvertieren, wenn Sie ihn an andere Funktionen wie `print()` übergeben, müssen Sie die Funktion `item()` verwenden:

```
x = torch.tensor([[1,2],[3,4],[5,6],[7,8]])
print(x)
# out:
# tensor([[1, 2],
#   [3, 4],
#   [5, 6],
#   [7, 8]])

# Indizierung, gibt einen Tensor zurück.
print(x[1,1])
# out: tensor(4)

# Indizierung, gibt einen Wert als Python-Zahl zurück.
print(x[1,1].item())
# out: 4
```

Im folgenden Code sehen Sie, dass sich Slicing mit dem gleichen Format [*start*:*end*:*step*] ausführen lässt wie Slicing von Python-Listen und NumPy-Arrays. Wir können auch boolesches Indizieren verwenden, um Teile der Daten zu extrahieren, die bestimmte Kriterien erfüllen:

```
# Slicing
print(x[:2,1])
# out: tensor([2, 4])

# Boolesches Indizieren.
# Nur Elemente kleiner als 5 behalten.
print(x[x<5])
# out: tensor([1, 2, 3, 4])
```

PyTorch unterstützt ebenfalls das Transponieren und Umformen von Arrays, wie es die nächsten Codezeilen zeigen:

```
# Array transponieren; x.t() oder x.T können verwendet werden.
print(x.t())
# tensor([[1, 3, 5, 7],
#         [2, 4, 6, 8]])

# Gestalt ändern; normalerweise wird view() gegenüber
# reshape() bevorzugt.
```

```
print(x.view((2,4)))
# tensor([[1, 3, 5, 7],
#         [2, 4, 6, 8]])
```

Es ist auch möglich, Tensoren zu kombinieren und aufzuteilen. Hierfür verwenden Sie Funktionen wie `torch.stack()` oder `torch.unbind()`:

```
# Tensoren kombinieren.
y = torch.stack((x, x))
print(y)
# out:
# tensor([[[1, 2],
#          [3, 4],
#          [5, 6],
#          [7, 8]],

#         [[1, 2],
#          [3, 4],
#          [5, 6],
#          [7, 8]]])

# Tensoren aufteilen.
a,b = x.unbind(dim=1)
print(a,b)
# out:
# tensor([1, 3, 5, 7]); tensor([2, 4, 6, 8])
```

PyTorch bietet einen robusten Satz von integrierten Funktionen, um auf Tensoren in verschiedener Art und Weise zuzugreifen, sie aufzuteilen und zu kombinieren. Tabelle 2-4 listet häufig verwendete Funktionen auf, mit denen sich Tensorelemente manipulieren lassen.

Tabelle 2-4: Operationen zum Indizieren, Slicen, Kombinieren und Aufteilen

Funktion	Beschreibung
`torch.`**`cat`**`()`	Verkettet die gegebene Sequenz von Tensoren in der angegebenen Dimension.
`torch.`**`chunk`**`()`	Teilt einen Tensor in eine festgelegte Anzahl von Chunks auf. Jeder Chunk ist eine Sicht des Eingabetensors.
`torch.`**`gather`**`()`	Sammelt Werte entlang einer Achse, die durch die Dimension gegeben wird.
`torch.`**`index_select`**`()`	Gibt einen neuen Tensor zurück, der den Eingabetensor entlang einer Dimension indiziert, und zwar mit den Einträgen im Index, der ein `LongTensor` ist.
`torch.`**`masked_select`**`()`	Gibt einen neuen eindimensionalen Tensor zurück, der den Eingabetensor entsprechend einer booleschen Maske, die ein `BoolTensor` ist, indiziert.
`torch.`**`narrow`**`()`	Gibt einen Tensor zurück, der eine schmale Version des Eingabetensors ist.
`torch.`**`nonzero`**`()`	Gibt die Indizes von Nicht-Null-Elementen zurück.
`torch.`**`reshape`**`()`	Gibt einen Tensor mit den gleichen Daten und der gleichen Anzahl von Elementen wie im Eingabetensor zurück, allerdings mit einer anderen Gestalt. Verwenden Sie stattdessen `view()`, um sicherzustellen, dass der Tensor nicht kopiert wird.

Tabelle 2-4: Operationen zum Indizieren, Slicen, Kombinieren und Aufteilen (Fortsetzung)

Funktion	Beschreibung
`torch.split()`	Teilt den Tensor in Chunks auf. Jeder Chunk ist eine Sicht oder Unterteilung des ursprünglichen Tensors.
`torch.squeeze()`	Gibt einen Tensor zurück, aus dem alle Dimensionen des Eingabetensors der Größe 1 entfernt wurden.
`torch.stack()`	Verkettet eine Sequenz von Tensoren entlang einer neuen Dimension.
`torch.t()`	Erwartet als Eingabe einen zweidimensionalen Tensor und transponiert die Dimensionen 0 und 1.
`torch.take()`	Gibt einen Tensor für die angegebenen Indizes zurück, wenn Slicing nicht kontinuierlich ist.
`torch.transpose()`	Transponiert nur die angegebenen Dimensionen.
`torch.unbind()`	Entfernt eine Tensordimension, indem ein Tupel der entfernten Dimension zurückgegeben wird.
`torch.unsqueeze()`	Gibt einen neuen Tensor zurück, bei dem eine Dimension der Größe 1 an der angegebenen Position eingefügt wurde.
`torch.where()`	Gibt einen Tensor ausgewählter Elemente zurück, die abhängig von der angegebenen Bedingung aus einem von zwei Tensoren stammen.

Einige dieser Funktionen mögen redundant erscheinen. Dabei sollten Sie jedoch die folgenden wichtigen Unterscheidungen und Best Practices im Hinterkopf behalten:

- `item()` ist eine wichtige und häufig verwendete Funktion, um die Python-Zahl aus einem Tensor mit einem einzelnen Wert zurückzugeben.
- Verwenden Sie in den meisten Fällen `view()` anstelle von `reshape()`, um die Gestalt von Tensoren zu ändern. Die Funktion `reshape()` kann bewirken, dass der Tensor je nach Layout im Arbeitsspeicher kopiert wird. Mit `view()` ist gewährleistet, dass er nicht kopiert wird.
- Ein- oder zweidimensionale Tensoren lassen sich ganz einfach mit `x.T` oder `x.t()` transponieren. Verwenden Sie `transpose()`, wenn Sie mit mehrdimensionalen Tensoren arbeiten.
- Die Funktion `torch.squeeze()` wird beim Deep Learning häufig verwendet, um eine ungenutzte Dimension zu entfernen. Zum Beispiel lässt sich ein Batch von Bildern mit einem einzelnen Bild mit `squeeze()` von vier auf drei Dimensionen reduzieren.
- Die Funktion `torch.unsqueeze()` wird beim Deep Learning oftmals verwendet, um eine Dimension der Größe 1 hinzuzufügen. Da die meisten PyTorch-Modelle einen Batch von Daten als Eingabe erwarten, können Sie `unsqueeze()` anwenden, wenn Sie nur eine einzige Datenstichprobe haben. Zum Beispiel können Sie ein dreidimensionales Bild an `torch.unsqueeze()` übergeben, um einen Batch mit einem Bild zu erstellen.

PyTorch ist von Haus aus stark auf Python ausgerichtet. Wie die meisten Python-Klassen können auch einige PyTorch-Funktionen mithilfe einer integrierten Methode wie zum Beispiel `x.size()` direkt auf einen Tensor angewendet werden.

Andere Funktionen werden direkt über den Namespace `torch` aufgerufen. Diese Funktionen übernehmen einen Tensor als Eingabe, wie es bei `x` in `torch.save(x, 'tensor.pt')` der Fall ist.

Tensoroperationen für die Mathematik

Da die Deep-Learning-Entwicklung stark auf mathematischen Berechnungen beruht, unterstützt PyTorch einen sehr robusten Satz von integrierten mathematischen Funktionen. Ob Sie nun neue Datentransformationen erstellen, Verlustfunktionen anpassen oder Ihre eigenen Optimierungsalgorithmen entwickeln, Ihre Forschung und Entwicklung können Sie mit den von PyTorch bereitgestellten mathematischen Funktionen beschleunigen.

Dieser Abschnitt soll Ihnen einen Überblick über viele der in PyTorch verfügbaren mathematischen Funktionen bieten, sodass Sie sich schnell mit den derzeit vorhandenen Funktionen bekannt machen und bei Bedarf die passenden Funktionen heraussuchen können.

PyTorch unterstützt viele verschiedene Arten von mathematischen Funktionen, darunter punktweise Operationen, Reduktionsfunktionen, Vergleichsberechnungen und Operationen der linearen Algebra sowie spektrale und andere mathematische Berechnungen. Die erste Kategorie nützlicher mathematischer Operationen, die wir uns ansehen, sind *punktweise Operationen*. Dabei wird auf jedem Punkt im Tensor eine Operation individuell ausgeführt und ein neuer Tensor zurückgegeben.

Nützlich sind derartige Operationen zum Runden und Abschneiden sowie für trigonometrische und logische Berechnungen. Standardmäßig erzeugen die Funktionen einen neuen Tensor oder verwenden einen Tensor, der im Parameter `out` übergeben wird. Vergessen Sie nicht, einen Unterstrich an den Funktionsnamen anzuhängen, wenn Sie eine In-Place-Operation durchführen wollen.

Tabelle 2-5 listet einige häufig verwendete punktweise Operationen auf.

Tabelle 2-5: Punktweise Operationen

Operationstyp	Beispielfunktionen
Einfache mathematische Operationen	`add()`, `div()`, `mul()`, `neg()`, `reciprocal()`, `true_divide()`
Abschneiden	`ceil()`, `clamp()`, `floor()`, `floor_divide()`, `fmod()`, `frac()`, `lerp()`, `remainder()`, `round()`, `sigmoid()`, `trunc()`
Komplexe Zahlen	`abs()`, `angle()`, `conj()`, `imag()`, `real()`
Trigonometrie	`acos()`, `asin()`, `atan()`, `cos()`, `cosh()`, `deg2rad()`, `rad2deg()`, `sin()`, `sinh()`, `tan()`, `tanh()`

Tabelle 2-5: Punktweise Operationen (Fortsetzung)

Operationstyp	Beispielfunktionen
Exponential- und Logarithmusfunktionen	`exp()`, `expm1()`, `log()`, `log10()`, `log1p()`, `log2()`, `logaddexp()`, `pow()`, `rsqrt()`, `sqrt()`, `square()`
Logische Funktionen	`logical_and()`, `logical_not()`, `logical_or()`, `logical_xor()`
Kumulative Mathematik	`addcdiv()`, `addcmul()`
Bitweise Operatoren	`bitwise_not()`, `bitwise_and()`, `bitwise_or()`, `bitwise_xor()`
Fehlerfunktionen	`erf()`, `erfc()`, `erfinv()`
Gammafunktionen	`digamma()`, `lgamma()`, `mvlgamma()`, `polygamma()`

Einzelheiten zur Verwendung der Funktionen erfahren Sie über die Python-Hinweise oder in der PyTorch-Dokumentation. Beachten Sie, dass `true_divide()` die Tensordaten zunächst in Gleitkommazahlen konvertiert und für eine Ganzzahldivision verwendet werden sollte, um echte Divisionsergebnisse zu erhalten.

Für die meisten Tensoroperationen gibt es drei verschiedene Syntaxformen. Tensoren unterstützen das Überladen von Operatoren, sodass Sie Operatoren wie in `z = x + y` direkt verwenden können. Obwohl sich das Gleiche auch mit PyTorch-Funktionen wie `torch.add()` ausführen lässt, ist dies weniger üblich. Schließlich sind In-Place-Operationen mit dem Unterstrich (`_`) als Postfix möglich. Die Funktion `y.add_(x)` erzielt das gleiche Ergebnis, das aber in `y` gespeichert wird.

Die zweite Kategorie der mathematischen Funktionen, die wir uns ansehen werden, sind die *Reduktionsoperationen*. Diese Operationen reduzieren ein Bündel von Zahlen auf eine einzelne Zahl oder eine kleinere Menge von Zahlen. Das heißt, sie verringern die *Dimensionalität* oder den *Rang* des Tensors. Zu den Reduktionsoperationen gehören Funktionen, die die Maximal- oder Minimalwerte ermitteln, sowie viele statistische Berechnungen wie die Ermittlung von Mittelwert oder Standardabweichung.

Im Deep Learning werden diese Operationen häufig verwendet. Beispielsweise reduziert man bei der Deep-Learning-Klassifizierung oftmals mit der Funktion `argmax()` die Softmax-Ausgaben auf eine dominante Klasse.

Tabelle 2-6 listet einige häufig verwendete Reduktionsoperationen auf.

Tabelle 2-6: Reduktionsoperationen

Funktion	Beschreibung
`torch.`**`argmax`**`(`*`input, dim, keepdim=False, out=None`*`)`	Liefert den Index/die Indizes des Maximalwerts über alle Elemente oder nur eine Dimension, wenn sie angegeben ist.
`torch.`**`argmin`**`(`*`input, dim, keepdim=False, out=None`*`)`	Liefert den Index/die Indizes des Minimalwerts über alle Elemente oder nur eine Dimension, wenn sie angegeben ist.
`torch.`**`dist`**`(`*`input, dim, keepdim=False, out=None`*`)`	Berechnet die *p*-Norm zweier Tensoren.

Tabelle 2-6: Reduktionsoperationen (Fortsetzung)

Funktion	Beschreibung
`torch.`**`logsumexp`**`(input, dim, keepdim=False, out=None)`	Berechnet den Logarithmus der summierten Exponentialwerte jeder Zeile des Eingabetensors in der angegebenen Dimension.
`torch.`**`mean`**`(input, dim, keepdim=False, out=None)`	Berechnet den Mittelwert oder Durchschnitt über alle Elemente oder nur eine Dimension, wenn sie angegeben ist.
`torch.`**`median`**`(input, dim, keepdim=False, out=None)`	Berechnet den Median oder mittleren Wert über alle Elemente oder nur eine Dimension, wenn sie angegeben ist.
`torch.`**`mode`**`(input, dim, keepdim=False, out=None)`	Berechnet den Modus oder häufigsten Wert über alle Elemente oder nur eine Dimension, wenn sie angegeben ist.
`torch.`**`norm`**`(input, p='fro', dim=None, keepdim=False, out=None, dtype=None)`	Berechnet die Matrix oder die Vektornorm über alle Elemente oder nur eine Dimension, wenn sie angegeben ist.
`torch.`**`prod`**`(input, dim, keepdim=False, dtype=None)`	Berechnet das Produkt aller Elemente oder von jeder Zeile des Eingabetensors, wenn sie angegeben ist.
`torch.`**`std`**`(input, dim, keepdim=False, out=None)`	Berechnet die Standardabweichung über alle Elemente oder nur eine Dimension, wenn sie angegeben ist.
`torch.`**`std_mean`**`(input, unbiased=True)`	Berechnet die Standardabweichung und den Mittelwert über alle Elemente oder nur eine Dimension, wenn sie angegeben ist.
`torch.`**`sum`**`(input, dim, keepdim=False, out=None)`	Berechnet die Summe aller Elemente oder nur einer Dimension, wenn sie angegeben ist.
`torch.`**`unique`**`(input, dim, keepdim=False, out=None)`	Entfernt Duplikate über den gesamten Tensor oder nur eine Dimension, wenn sie angegeben ist.
`torch.`**`unique_consecutive`**`(input, dim, keepdim=False, out=None)`	Ähnlich wie `torch.unique()`, entfernt aber nur aufeinanderfolgende Duplikate.
`torch.`**`var`**`(input, dim, keepdim=False, out=None)`	Berechnet die Varianz über alle Elemente oder nur eine Dimension, wenn sie angegeben ist.
`torch.`**`var_mean`**`(input, dim, keepdim=False, out=None)`	Berechnet den Mittelwert und die Varianz über alle Elemente oder nur eine Dimension, wenn sie angegeben ist.

Viele dieser Funktionen akzeptieren den Parameter `dim`, der die Dimension der Reduktion für mehrdimensionale Tensoren angibt. Dies ähnelt dem Parameter `axis` in NumPy. Wenn `dim` nicht angegeben ist, erfolgt die Reduktion standardmäßig über alle Dimensionen. Geben Sie `dim = 1` an, wird die Operation für jede Zeile ausgeführt. Zum Beispiel berechnet `torch.mean(x,1)` den Mittelwert für jede Zeile im Tensor `x`.

Es ist üblich, Methoden miteinander zu verketten. Zum Beispiel erzeugt `torch.rand(2,2).max().item()` einen 2 × 2-Tensor mit zufälligen Gleitkommazahlen, sucht den Maximalwert und gibt den Wert selbst aus dem resultierenden Tensor zurück.

Wir kommen nun zu den *Vergleichsfunktionen* von PyTorch. Normalerweise vergleichen diese Funktionen alle Werte innerhalb eines Tensors oder den Wert eines

Tensors mit dem Wert eines anderen Tensors. Die Funktionen können einen Tensor zurückgeben, der aus booleschen Werten besteht, die auf dem Wert jedes Elements basieren, wie zum Beispiel `torch.eq()` oder `torch.is_boolean()`. Es gibt auch Funktionen, die den Maximal- oder Minimalwert ermitteln, Tensorwerte sortieren, die oberste Teilmenge von Tensorelementen zurückgeben und mehr.

Tabelle 2-7 gibt eine Übersicht über häufig verwendete Vergleichsoperationen.

Tabelle 2-7: Vergleichsoperationen

Operationstyp	Beispielfunktionen
Einen Tensor mit anderen Tensoren vergleichen	`eq()`, `ge()`, `gt()`, `le()`, `lt()`, `ne()` oder ==, >, >=, <, <= bzw. !=
Tensorstatus oder Bedingungen testen	`isclose()`, `isfinite()`, `isinf()`, `isnan()`
Einen einzelnen booleschen Wert für den gesamten Tensor zurückgeben	`allclose()`, `equal()`
Wert(e) über den gesamten Tensor oder entlang einer angegebenen Dimension ermitteln	`argsort()`, `kthvalue()`, `max()`, `min()`, `sort()`, `topk()`

Zwar scheinen Vergleichsfunktionen recht einfach zu sein, doch es gibt einige wichtige Punkte zu beachten. Häufige Fallstricke sind zum Beispiel:

- Die Funktion `torch.eq()` oder == gibt einen Tensor derselben Größe mit einem booleschen Ergebnis für jedes Element zurück. Die Funktion `torch.equal()` prüft, ob die Tensoren gleich groß sind, und wenn alle Elemente innerhalb des Tensors gleich sind, gibt sie einen einzelnen booleschen Wert zurück.
- Die Funktion `torch.allclose()` gibt ebenfalls einen einzelnen booleschen Wert zurück, wenn alle Elemente nahe bei einem angegebenen Wert liegen.

Als nächsten Typ mathematischer Funktionen betrachten wir *Funktionen für lineare Algebra*. Diese Funktionen erleichtern Matrixoperationen und sind wichtig für Deep-Learning-Berechnungen.

Viele Berechnungen, einschließlich der Algorithmen für Gradientenabstieg und Optimierung, werden mithilfe von linearer Algebra implementiert. PyTorch unterstützt einen robusten Satz von integrierten Operationen der linearen Algebra, von denen viele auf den standardisierten Bibliotheken *Basic Linear Algebra Subprograms* (BLAS) und *Linear Algebra Package* (LAPACK) basieren.

Tabelle 2-8 listet einige häufig verwendete Operationen der linearen Algebra auf.

Tabelle 2-8: Operationen der linearen Algebra

Funktion	Beschreibung
`torch.matmul()`	Berechnet das Matrixprodukt zweier Tensoren, unterstützt Broadcasting.
`torch.chain_matmul()`	Berechnet das Matrixprodukt aus *N* Tensoren.
`torch.mm()`	Berechnet das Matrixprodukt zweier Tensoren (verwenden Sie `matmul()`, wenn Broadcasting erforderlich ist).

Tabelle 2-8: Operationen der linearen Algebra (Fortsetzung)

Funktion	Beschreibung
`torch.addmm()`	Berechnet ein Matrixprodukt von zwei Tensoren und addiert es zur Eingabe.
`torch.bmm()`	Berechnet einen Batch von Matrixprodukten.
`torch.addbmm()`	Berechnet einen Batch von Matrixprodukten und addiert ihn zur Eingabe.
`torch.baddbmm()`	Berechnet einen Batch von Matrixprodukten und addiert ihn zum Eingabe-Batch.
`torch.mv()`	Berechnet das Produkt aus Matrix und Vektor.
`torch.addmv()`	Berechnet das Produkt aus Matrix und Vektor und addiert es zur Eingabe.
`torch.matrix_power`	Gibt einen Tensor zur *n*-ten Potenz zurück (für quadratische Tensoren).
`torch.eig()`	Ermittelt die Eigenwerte und Eigenvektoren eines reellen quadratischen Tensors.
`torch.inverse()`	Berechnet die Inverse eines quadratischen Tensors.
`torch.det()`	Berechnet die Determinante einer Matrix oder eines Batches von Matrizen.
`torch.logdet()`	Berechnet die logarithmische Determinante einer Matrix oder eines Batches von Matrizen.
`torch.dot()`	Berechnet das innere Produkt zweier Tensoren.
`torch.addr()`	Berechnet das äußere Produkte zweier Tensoren und addiert es zur Eingabe.
`torch.solve()`	Gibt die Lösung für ein System linearer Gleichungen zurück.
`torch.svd()`	Führt eine Einzelwertzerlegung durch.
`torch.pca_lowrank()`	Führt eine lineare Hauptkomponentenanalyse durch.
`torch.cholesky()`	Berechnet eine Cholesky-Zerlegung.
`torch.cholesky_inverse()`	Berechnet die Inverse einer symmetrischen positiv definiten Matrix und gibt den Cholesky-Faktor zurück.
`torch.cholesky_solve()`	Löst ein System linearer Gleichungen mithilfe der Cholesky-Zerlegung.

Die Funktionen in Tabelle 2-8 reichen von Matrixmultiplikation und Funktionen für Batch-Berechnungen bis zu Solvern. Beachten Sie, dass eine Matrixmultiplikation nicht dasselbe ist wie eine punktweise Multiplikation mit der Funktion `torch.mul()` oder dem Operator *.

Eine vollständige Abhandlung der linearen Algebra ginge über den Rahmen dieses Buchs hinaus. Doch manchmal ist es hilfreich, auf einige der Funktionen für lineare Algebra zurückzugreifen, wenn es um Feature-Reduktion oder die Entwicklung spezieller Deep-Learning-Algorithmen geht. Eine komplette Liste der verfügbaren Funktionen mit weiteren Einzelheiten zu ihrer Verwendung finden Sie in der PyTorch-Dokumentation zur linearen Algebra (*https://pytorch.tips/linear-algebra*).

Zum Schluss betrachten wir *spektrale und andere mathematische Operationen*. Je nach Interessengebiet können diese Funktionen für Datentransformationen oder -analysen nützlich ist. Zum Beispiel können spektrale Operationen wie die schnelle Fourier-Transformation (*Fast Fourier Transform*, FFT) eine wichtige Rolle in der Computervision oder in Anwendungen der digitalen Signalverarbeitung spielen.

Tabelle 2-9 beschreibt einige integrierte Operationen für spektrale Analysen und andere mathematische Operationen.

Tabelle 2-9: Spektrale Analysen und andere mathematische Operationen

Operationstyp	Beispielfunktionen
Schnelle, inverse und Kurzzeit-Fourier-Transformationen	`fft()`, `ifft()`, `stft()`
FFT reell zu komplex und FFT komplex zu reell invers (IFFT)	`rfft()`, `irfft()`
Fensteralgorithmen	`bartlett_window()`, `blackman_window()`, `hamming_window()`, `hann_window()`
Histogramm- und Bin-Zähler	`histc()`, `bincount()`
Kumulative Operationen	`cummax()`, `cummin()`, `cumprod()`, `cumsum()`, `trace()` (Summe der Diagonalen), `einsum()` (Summe der Produkte mit einsteinscher Summation)
Normalisierungsfunktionen	`cdist()`, `renorm()`
Kreuzprodukt, Punktprodukt und kartesisches Produkt	`cross()`, `tensordot()`, `cartesian_prod()`
Funktionen, die einen diagonalen Tensor mit Elementen des Eingabetensors erzeugen	`diag()`, `diag_embed()`, `diag_flat()`, `diagonal()`
einsteinsche Summation	`einsum()`
Funktionen für Matrixreduktion und Restrukturierung	`flatten()`, `flip()`, `rot90()`, `repeat_interleave()`, `meshgrid()`, `roll()`, `combinations()`
Funktionen, die die unteren oder oberen Dreiecke und deren Indizes zurückgeben	`tril()`, `tril_indices`, `triu()`, `triu_indices()`

Automatische Differentiation (Autograd)

Die Funktion `backward()` verdient einen eigenen Unterabschnitt, weil sie PyTorch für die Deep-Learning-Entwicklung so leistungsfähig macht. Diese Funktion nutzt das PyTorch-Paket `torch.autograd` für die automatische Differentiation, um Gradienten von Tensoren nach der Kettenregel zu differenzieren und zu berechnen. Es folgt ein einfaches Beispiel für die automatische Differentiation. Wir definieren eine Funktion $f = sum(x^2)$, wobei x eine Matrix von Variablen ist. Wenn wir df / dx für jede Variable in der Matrix ermitteln wollen, müssen wir das Flag `requires_grad` für den Tensor `x` auf `True` setzen, wie es der folgende Code zeigt:

```
x = torch.tensor([[1,2,3],[4,5,6]],
        dtype=torch.float, requires_grad=True)
print(x)
# out:
# tensor([[1., 2., 3.],
#         [4., 5., 6.]], requires_grad=True)

f = x.pow(2).sum()
print(f)
```

```
# tensor(91., grad_fn=<SumBackward0>)

f.backward()
print(x.grad) # df/dx = 2x
# tensor([[ 2., 4., 6.],
#         [ 8., 10., 12.]])
```

Die Funktion `f.backward()` führt die Differentiation in Bezug auf *f* durch und speichert *df / dx* im Attribut `x.grad`. Gemäß den Regeln der Differentialrechnung ist *df / dx = 2x* die Ableitung von *f* in Bezug auf *x*. Als Ausgabe erscheinen die Ergebnisse der Auswertung von *df / dx* für die Werte von *x*.

Nur Tensoren mit einem Gleitkommadatentyp können Gradienten erfordern.

Beim Training von neuronalen Netzen müssen die Gewichtsgradienten für die Fehlerrückführung berechnet werden. Wenn neuronale Netze tiefer und komplexer werden, lassen sich mit diesem Feature die komplexen Berechnungen automatisieren. Weitere Informationen zur Funktionsweise von Autograd finden Sie im Tutorial unter *https://pytorch.tips/autograd-explained*.

Dieses Kapitel soll Ihnen als Kurzreferenz für das Erstellen von Tensoren und für die Ausführung von Operationen dienen. Da Sie nun eine gute Grundlage für Tensoren besitzen, konzentrieren wir uns darauf, wie Sie Tensoren und PyTorch für Ihre Deep-Learning-Forschung verwenden. Das nächste Kapitel befasst sich mit dem Deep-Learning-Entwicklungsprozess, bevor Sie dann beginnen, Code zu schreiben.