KAPITEL 2
Erstes neuronales Netz mit PyTorch

In diesem Kapitel lernen Sie PyTorch weiter kennen. Wir erstellen ein richtiges neuronales Netz, das eine einfache, aber schon vertraute Aufgabe löst.

Das MNIST-Bilddatensatz

In meinem Buch *Neuronale Netze selbst programmieren* haben wir ein neuronales Netz entwickelt, das gelernt hat, Bilder von handgeschriebenen Ziffern zu klassifizieren.

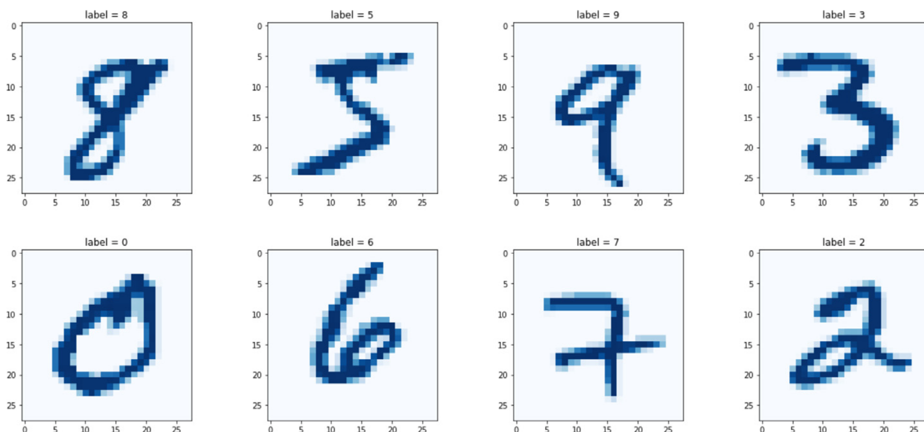

Abbildung 2-1: Beispiele für handgeschriebene Ziffern aus dem MNIST-Datensatz

Der *MNIST*-Datensatz ist ein bekannter Satz von Bildern, der oftmals herangezogen wird, um die Performance von Algorithmen für maschinelles Lernen zu messen und zu vergleichen. Er enthält 60.000 Bilder, die für das Training eines Modells für maschinelles Lernen gedacht sind, und 10.000 Bilder, um dessen Performance zu testen.

Die Bilder haben eine Größe von 28×28 Pixeln und sind monochrom, also nicht farbig. Die numerischen Werte der einzelnen Pixel geben an, wie hell oder dunkel

das jeweilige Pixel ist, und je nachdem, woher Sie die Daten erhalten, liegen die Werte zwischen 0 und 255.

Die MNIST-Daten abrufen

Gehen Sie in *Drive* zu Ihrem *Colab Notebooks*-Ordner, in dem Ihre Python-Notebooks gespeichert sind. Klicken Sie auf *Neu*, um einen neuen Ordner namens *mnist_data* zu erstellen.

Dieser *mnist_data*-Ordner sollte sich im selben Ordner wie Ihre Notebooks befinden (siehe Abbildung 2-2).

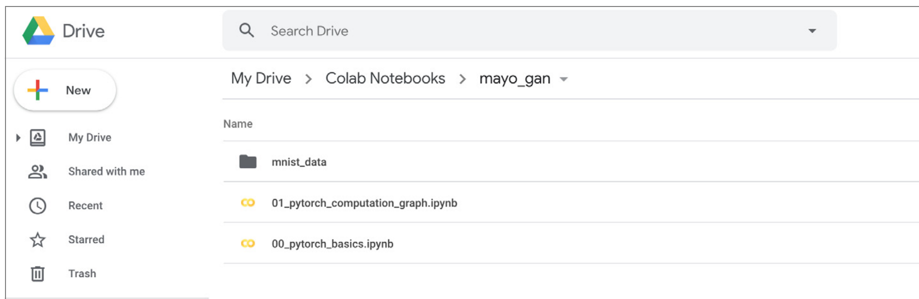

Abbildung 2-2: Der neu erstellte Ordner »mnist_data«

Laden Sie die MNIST-Daten über die folgenden Links auf Ihren Computer herunter:

- Trainingsdaten: *https://pjreddie.com/media/files/mnist_train.csv*
- Testdaten: *https://pjreddie.com/media/files/mnist_test.csv*

Nachdem Sie diese beiden Dateien heruntergeladen haben, laden Sie sie in den Ordner *mnist_data* hoch. Gehen Sie hierfür zum Ordner *mnist_data*, klicken Sie auf die Schaltfläche *Neu* und wählen Sie den Befehl *Dateien hochladen*.

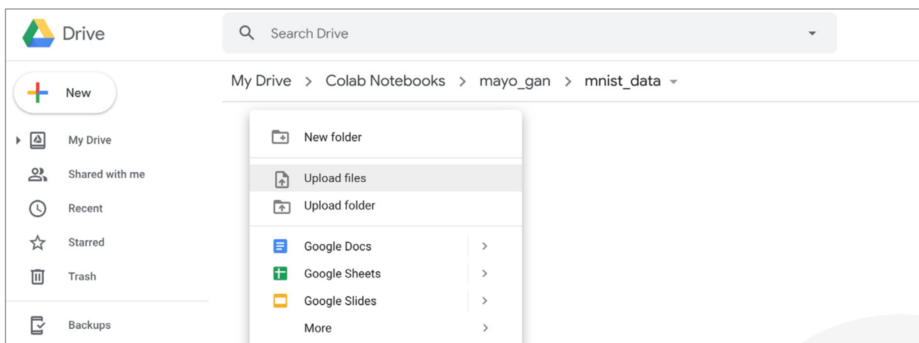

Abbildung 2-3: Die heruntergeladenen MNIST-Daten in den Ordner »mnist_data« hochladen

Nach kurzer Zeit sollten die beiden Dateien im Ordner *mnist_data* aufgelistet werden.

Ein Blick auf die Daten

Es ist eine gute Angewohnheit, alle neuen Daten vorab zu inspizieren, um ein Gefühl dafür zu bekommen, bevor Sie sie mit Tools und Algorithmen attackieren!

Wir müssen unsere hochgeladenen Daten für unseren Python-Code zugänglich machen. Hierfür mounten wir unser Laufwerk (Drive), sodass es als Ordner erscheint.

Beginnen Sie ein neues Notebook und führen Sie den folgenden Code aus:

```
# Laufwerk mounten, um auf Datendateien zuzugreifen
from google.colab import drive
drive.mount('./mount')
```

Sie werden aufgefordert, auf einen Link zu klicken, der Sie zu einem neuen Register bringt. Auf dieser Registerkarte sollen Sie das Konto bestätigen und Berechtigungen für das Laufwerk gewähren, das gemountet und zugänglich gemacht werden soll. Daraufhin erhalten Sie einen Code, den Sie kopieren und in die Notebook-Zelle einfügen (siehe Abbildung 2-4).

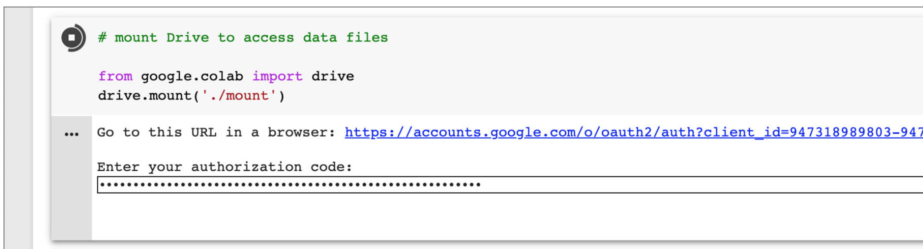

Abbildung 2-4: Den Autorisierungscode kopieren und einfügen, um das Laufwerk zu mounten und zugänglich zu machen

Sobald das erledigt ist, erhalten Sie eine Bestätigung, dass das Drive gemountet wurde und für Ihren Python-Code aus dem Ordner *./mount* sichtbar ist.

Unsere MNIST-Datendateien liegen im CSV-Format vor, d. h. Zeilen mit Werten, die jeweils durch Kommata getrennt sind (engl. *Comma Separated Values*). Es gibt viele Wege, um die Daten zu laden und anzuzeigen. Wir verwenden die *Pandas*-Bibliothek, da sie die Aufgabe wirklich einfach macht.

Importieren Sie in einer neuen Zelle die Pandas-Bibliothek:

```
# Pandas importieren, um CSV-Dateien zu lesen
import pandas
```

In der nächsten Zelle nutzen wir Pandas, um die Trainingsdaten in einen Dataframe zu laden. Der folgende Code ist eigentlich eine einzelne Zeile. Er ist nur so lang, weil der Dateipfad zur Datei *mnist_train.csv* so lang ist:

```
df = pandas.read_csv('mount/My Drive/Colab Notebooks/myo_gan/mnist_data/
mnist_train.csv', header=None)
```

Kontrollieren Sie, ob der von Ihnen verwendete Dateipfad auch dem entspricht, in dem Sie Ihre Dateien abgelegt haben. Mein eigener Code und die Daten befinden sich in einem Ordner *myo_gan* innerhalb von *Colab Notebooks*.

Ein *Pandas-Dataframe* ist einem NumPy-Array ähnlich, besitzt aber zusätzliche Features wie zum Beispiel benannte Spalten und Zeilen sowie viele Komfortfunktionen wie etwa solche zum Summieren und Filtern von Daten.

So können wir mit der Funktion `head()` einen Blick auf die Spitze eines großen Dataframes werfen:

```
df.head()
```

Als Ergebnis werden die ersten fünf Zeilen des Dataframes angezeigt.

```
[5] df = pandas.read_csv('mount/My Drive/Colab Notebooks/myo_gan/mnist_data/mnist_train.csv', header=None)

df.head()
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

5 rows × 785 columns

Abbildung 2-5: Die ersten fünf Zeilen eines großen Dataframes

Jede Zeile der MNIST-Daten besteht aus 785 Werten. Die erste Zahl ist die Ziffer, die das Bild darstellen soll, die restlichen 784 Zahlen sind die Pixelwerte für das 28×28-Bild. Mit der Funktion `info()` erhalten Sie einen Überblick über den Dataframe.

```
[9] df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 60000 entries, 0 to 59999
Columns: 785 entries, 0 to 784
dtypes: int64(785)
memory usage: 359.3 MB
```

Abbildung 2-6: Überblicksinformationen für einen Dataframe

Aus der Zusammenfassung erfahren wir, dass der Dataframe 60.000 Zeilen umfasst. Das sind die 60.000 Trainingsbilder. Außerdem wird bestätigt, dass eine Zeile aus 785 Werten besteht.

Wir visualisieren nun die Daten, indem wir aus einer Zeile mit Pixelwerten ein tatsächliches Bild erzeugen.

Hierzu greifen wir auf die vielseitige Bibliothek `matplotlib` zurück. Aktualisieren Sie die Zelle, in der die Bibliotheken importiert werden, um das Paket `pyplot` aus der Bibliothek `matplotlib` einzubinden:

```
# Pandas importieren, um CSV-Dateien zu lesen
import pandas
```

```
# matplotlib importieren, um Bilder anzuzeigen
import matplotlib.pyplot as plt
```

Führen Sie die aktualisierte Zelle erneut aus, um `pyplot` verfügbar zu machen. Sehen Sie sich den folgenden Code an:

```
# Daten von Dataframe abrufen
row = 0
data =  df.iloc[row]

# Das Label ist der erste Wert.
label = data[0]

# Die Bilddaten sind die restlichen 784 Werte.
img = data[1:].values.reshape(28,28)
plt.title("label  =  " +  str(label))
plt.imshow(img, interpolation='none',  cmap='Blues')
plt.show()
```

Der erste Teil des Codes wählt aus den MNIST-Daten das Bild aus, das uns interessiert. Das erste Bild, das in der ersten Zeile enthalten ist, wird mit `row = 0` ausgewählt. Die Codezeile `df.iloc[row]` holt die erste Zeile aus dem Datensatz und weist sie der Variablen `data` zu.

Im nächsten Codeabschnitt rufen wir die erste Zahl aus dieser Zeile ab und nennen sie `label`, da sie die jeweilige Ziffer kennzeichnet (von engl. label – Kennzeichen).

Der dritte Teil des Codes übernimmt die restlichen 784 Werte aus dieser Zeile, formt sie in ein quadratisches Array von 28 Zeilen und 28 Spalten um und weist sie einer Variablen `img` (von engl. image – Bild) zu. Das Array wird dann als Bitmap gezeichnet, wobei der Titel das Label zeigt, das wir aus den Daten extrahiert haben. Für die Funktion `imshow()`, die das Bitmap zeichnet, sind viele Optionen definiert. Wir verwenden hier die beiden Optionen, die die Farbpalette festlegen und `pyplot` anweisen, die Pixel nicht zu glätten.

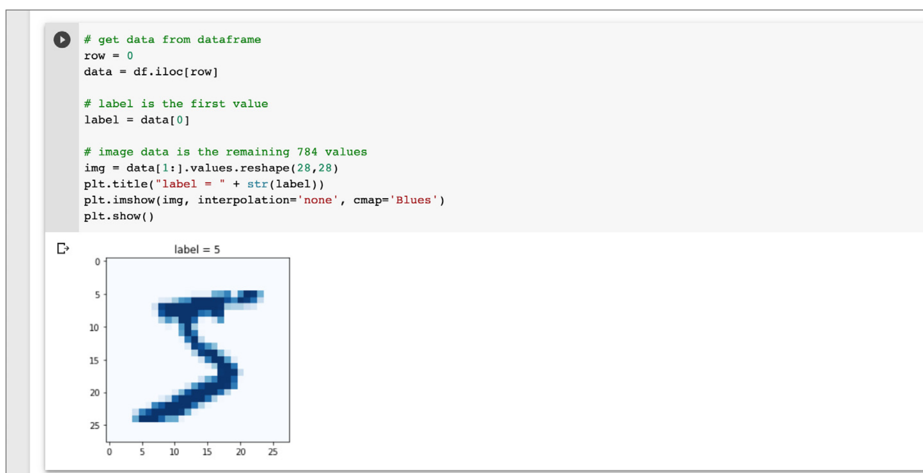

Abbildung 2-7: Darstellung der Bilddaten für die erste Ziffer

Abbildung 2-7 zeigt das Ergebnis mit dem ersten Bild im MNIST-Trainingsdatensatz. Es sieht wie eine Fünf aus, und das Label bestätigt, dass es eine Fünf sein soll.

Experimentieren Sie mit anderen Bildern aus dem Datensatz, indem Sie den Zeilenwert `row` ändern und die Zelle erneut ausführen. Wenn Sie zum Beispiel `row = 13` probieren, sollten Sie ein Bild sehen, das wie eine Sechs aussieht.

Der Code, den wir bisher entwickelt haben, ist online unter folgendem Link zu finden:

- *https://github.com/makeyourownneuralnetwork/gan/blob/master/02_mnist_data.ipynb*

Ein einfaches neuronales Netz

Bevor wir uns eingehend damit befassen, Code für ein neuronales Netz zu schreiben, treten wir einen Schritt zurück und machen uns ein Bild davon, was wir zu erreichen versuchen. Abbildung 2-8 zeigt unseren Ausgangspunkt und wo wir letztlich hinwollen.

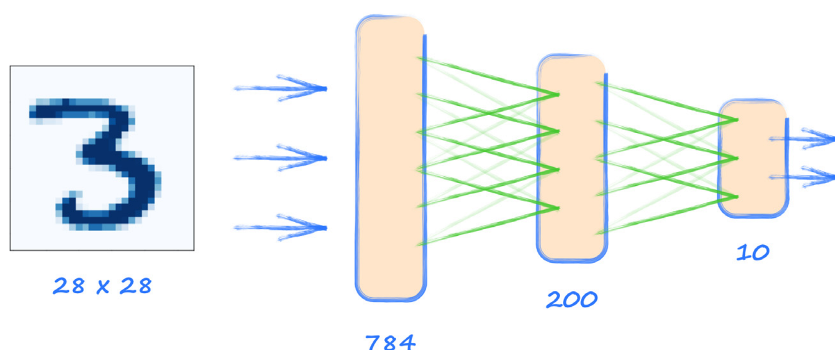

Abbildung 2-8: Ausgangspunkt und Ziel für ein neuronales Netz

Den Ausgangspunkt bildet ein MNIST-Bild, das 28 mal 28 oder 784 Pixelwerte umfasst. Das heißt, die erste Schicht unseres neuronalen Netzes muss aus 784 Knoten bestehen. Bei der Größe der Eingabeschicht haben wir kaum eine Wahl.

Die Größe der letzten Schicht, der Ausgabeschicht, können wir in gewissen Grenzen beeinflussen. Diese Schicht muss in der Lage sein, die Frage zu beantworten: »Welche Ziffer ist das?« Die Antwort könnte eine der Ziffern von 0 bis 9 sein, was 10 verschiedene Möglichkeiten bedeutet. Der einfachste Ansatz sieht einen Knoten für jede dieser zehn möglichen Klassen vor.

Mehr Entscheidungsfreiheit haben wir bei der mittleren – der versteckten – Schicht. Da wir vor allem etwas über PyTorch lernen und nicht die beste Größe finden wollen, verwenden wir das, was wir in *Neuronale Netze selbst programmieren* gelernt haben, und wählen für den Anfang eine Größe von 200.

Alle Knoten in einer Schicht sind mit jedem Knoten in der nächsten Schicht verbunden. Man spricht dann auch von *vollständig verbundenen Schichten*.

Diesem Bild fehlt ein wichtiges Element. Wir müssen eine Aktivierungsfunktion auswählen, die auf die Ausgaben von versteckter und Ausgabeschicht angewendet wird. In *Neuronale Netze selbst programmieren* haben wir die s-förmige *logistische Funktion* verwendet. Der Einfachheit halber bleiben wir dabei.

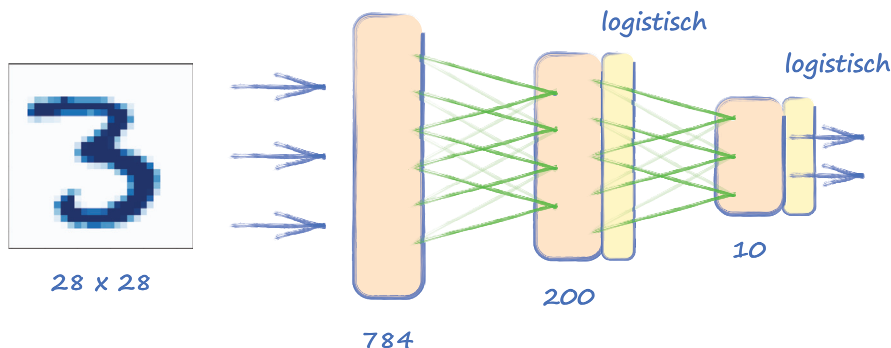

Abbildung 2-9: Das einfache neuronale Netz mit logistischen Funktionen in der versteckten und der Ausgabeschicht

Wir können nun diese *Architektur* des neuronalen Netzes in PyTorch-Code umsetzen. Mit PyTorch lassen sich neuronale Netze einfacher erstellen und ausführen, da diese Bibliothek eine Menge Arbeit hinter den Kulissen erledigt. Damit das funktioniert, müssen wir uns an das Codierungsmuster von PyTorch halten.

Wenn wir eine Klasse für ein neuronales Netz erstellen, müssen wir von der PyTorch-eigenen Klasse `torch.nn` erben. Das bringt eine Menge an PyTorch-Maschinerie mit sich, beispielsweise das automatische Erstellen des Berechnungsgraphen, die Behandlung der Gewichte und die Aktualisierung der Gewichte während des Trainings.

Legen Sie ein neues Notebook an und importieren Sie sowohl `torch` als auch `torch.nn`:

```
# Bibliotheken importieren
import torch
import torch.nn as nn
```

Das Modul `torch.nn` wird als `nn` importiert, was zur Namenskonvention geworden ist.

Erstellen wir nun die Klasse für unser neuronales Netz. Der folgende Code zeigt eine Klasse, die wir `Classifier` genannt haben und die von `nn.Module` erbt:

```
class Classifier(nn.Module):

    def __init__(self):
        # Übergeordnete PyTorch-Klasse initialisieren
        super().__init__()
```

Die spezielle Funktion `__init__(self)` wird aufgerufen, wenn ein Objekt aus einer Klasse erstellt wird. Oftmals dient sie dazu, ein Objekt einzurichten und es einsatzbereit zu machen. Vielleicht haben Sie schon gehört, dass sie *Konstruktor* genannt wird, was ein recht aussagekräftiger Name ist. Hier geschieht diese Einrichtung mit `super().__init()` – das sieht etwas eigenartig aus, ruft aber einfach den Konstruktor der übergeordneten Klasse auf. Somit wird `nn.Module` von PyTorch die Klasse `Classifier` für uns einrichten. Toll, oder?

Wir definieren nun die Architektur des neuronalen Netzes. Hierfür gibt es verschiedene Herangehensweisen. Bei einfachen Netzen können wir mit `nn.Sequential()` eine Liste der Netzelemente bereitstellen. Die Elemente müssen in der Reihenfolge erscheinen, in der die Informationen sie durchlaufen sollen.

```
class Classifier(nn.Module):

    def __init__(self):
        # Übergeordnete PyTorch-Klasse initialisieren
        super().__init__()

        # Schichten des neuronalen Netzes definieren
        self.model = nn.Sequential(
            nn.Linear(784, 200),
            nn.Sigmoid(),
            nn.Linear(200, 10),
            nn.Sigmoid()
        )
```

In der Funktion `nn.Sequential()` sind die folgenden Elemente definiert:

- `nn.Linear(784, 200)` ist eine vollständig verbundene Zuordnung von 784 Knoten auf 200 Knoten. Dieses Element enthält die Gewichte für die Verbindungen zwischen den Knoten, die während des Trainings aktualisiert werden.
- `nn.Sigmoid()` wendet die s-förmige logistische Aktivierungsfunktion auf die Ausgaben des vorherigen Elements an, in diesem Fall die 200 Knoten.
- `nn.Linear(200, 10)` bildet 200 Knoten auf 10 Knoten ab. Dieses Element enthält die Gewichte für die Verbindungen zwischen der mittleren, versteckten Schicht und der letzten Schicht mit den 10 Ausgabeknoten.
- `nn.Sigmoid()` wendet die s-förmige logistische Aktivierungsfunktion auf die Ausgabe der 10 Knoten an. Das Ergebnis ist die endgültige Ausgabe des Netzes.

Vielleicht fragen Sie sich, warum die Funktion `nn.Linear()` so genannt wird. Das liegt daran, dass sie eine lineare Funktion der Form `Ax + B` auf die Eingabewerte anwendet und an die Ausgabe liefert. Das `A` bezeichnet die Verbindungsgewichte, und das `B` kann man als Schwellenwert (engl. Bias) betrachten. Beide Parameter werden während des Trainings aktualisiert. Man spricht auch von *lernbaren Parametern*.

Für das neuronale Netz haben wir die Elemente definiert, die die Informationen in Vorwärtsrichtung weitergeben, doch wir haben noch nicht definiert, wie das Netz den Fehler ermittelt und damit dann die lernbaren Parameter des Netzes aktualisiert.

Es gibt viele Möglichkeiten, den Fehler für das Netz zu definieren. Für gängige Optionen stellt PyTorch komfortable Funktionen bereit. Eine der einfachsten berechnet den *mittleren quadratischen Fehler*. Dabei werden die Differenzen zwischen den tatsächlichen und den gewünschten Ausgaben an jedem der Ausgabeknoten quadriert, und daraus wird dann der Mittelwert gebildet. PyTorch realisiert dies mit `torch.nn.MSELoss()`.

Im Konstruktor können wir diese Fehlerfunktion auswählen und ihr einen Namen geben.

```
# Verlustfunktion erzeugen
self.loss_function = nn.MSELoss()
```

Gelegentlich werden die Begriffe *Fehlerfunktion* und *Verlustfunktion* gleichberechtigt nebeneinander verwendet, und oftmals ist das auch in Ordnung. Um genau zu sein: Der *Fehler* ist die einfache Differenz zwischen einer gewünschten und der tatsächlichen Ausgabe, und der *Verlust* wird aus dem Fehler in einer Weise berechnet, die für das zu lösende Problem zweckmäßig ist.

Wir müssen diesen *Fehler* – oder richtiger gesagt, den *Verlust* – verwenden, um die Verbindungsgewichte des Netzes zu aktualisieren. Auch hierfür gibt es mehrere Verfahren, und PyTorch bietet Funktionen für gängige Optionen. Beginnen wir mit der einfachen Version, die in *Neuronale Netze selbst programmieren* entwickelt wurde: dem sogenannten *stochastischen Gradientenabstieg* (SGD, von engl. *Stochastic Gradient Descent*) mit einer Lernrate von `0.01`.

```
# Optimierer erstellen, dafür einfachen stochastischen Gradientenabstieg verwenden
self.optimiser = torch.optim.SGD(self.parameters(), lr=0.01)
```

Interessant ist, dass wir alle lernbaren Parameter an den SGD-Optimierer übergeben. Diese sind über die Methode `self.parameters()` zugänglich, die die Maschinerie von PyTorch für uns erzeugt.

Um Informationen durch das Netz zu leiten, geht PyTorch von einer Methode `forward()` aus. Wir müssen also eine solche Methode erzeugen, die allerdings minimal ausfallen darf:

```
def forward(self, inputs):
    # Modell einfach ausführen
    return self.model(inputs)
```

Hier nehmen wir die gegebenen Eingaben und übergeben sie an die Methode `self.model()`, die wir oben mithilfe von `nn.Sequential()` definiert haben. Die Ausgabe des Modells wird einfach an den Aufrufer der Methode `forward()` zurückgegeben.

Gönnen wir uns eine Pause und gehen wir noch einmal durch, was wir bisher erledigt haben:

- Wir haben eine Klasse für ein neuronales Netz erstellt, die von `nn.Module` erbt. Durch das Vererben von `nn.Module` bekommen wir viel von der Maschinerie mit, die für das Trainieren eines neuronalen Netzes erforderlich ist.

- Wir haben die Elemente des neuronalen Netzes definiert, durch die Informationen fließen. Die Wahl ist auf die Methode `nn.Sequential` gefallen, da sie für einfache Netze kurz und bündig ist.
- Wir haben die Verlustfunktion und die Optimierermethode definiert, um die lernbaren Parameter des Netzes zu aktualisieren.
- Schließlich haben wir eine `forward()`-Funktion hinzugefügt, die PyTorch erwartet, um Informationen durch das Netz zu leiten.

Unsere Klasse für das neuronale Netz sieht jetzt folgendermaßen aus:

```
# Klassifiziererklasse

class Classifier(nn.Module):

    def __init__(self):
        # Übergeordnete PyTorch-Klasse initialisieren
        super().__init__()

        # Schichten des neuronalen Netzes definieren
        self.model = nn.Sequential(
            nn.Linear(784, 200),
            nn.Sigmoid(),
            nn.Linear(200, 10),
            nn.Sigmoid()
        )

        # Verlustfunktion erstellen
        self.loss_function = nn.MSELoss()

        # Optimierer mit einfachem stochastischem Gradientenabstieg erstellen
        self.optimiser = torch.optim.SGD(self.parameters(), lr=0.01)

        pass

    def forward(self, inputs):
      # Modell einfach ausführen
      return self.model(inputs)
```

Wie trainieren wir das Netz? Brauchen wir eine Funktion `train()`, so wie wir eine Funktion `forward()` haben? Eigentlich verlangt PyTorch keine `train()`-Methode. Es bleibt uns überlassen, unseren Code für das Training zu strukturieren.

Wir wollen unseren eigenen Code einfach und konsistent halten und erstellen neben der Methode `forward()` eine Methode `train()`, die die Methode `forward()` begleitet.

Eine `train()`-Methode benötigt sowohl die *Eingänge* in das Netz als auch die gewünschten *Ausgänge* des *Ziels*, damit sie diese mit dem tatsächlichen Ausgang vergleichen und den *Verlust* berechnen kann.

```
def train(self, inputs, targets):
    # Den Ausgang des Netzes berechnen
    outputs  = self.forward(inputs)
```

```
        # Verlust berechnen
        loss = self.loss_function(outputs, targets)
```

Die Funktion `train()` leitet zunächst die Eingänge mithilfe der Methode `forward()` durch das Netz weiter, um die Ausgänge zu erhalten.

Die Verlustfunktion, die wir zuvor definiert haben, berechnet den Verlust. PyTorch macht uns das Ganze hier sehr einfach. Wir übergeben lediglich der Funktion die Ausgaben und die gewünschten Ziele des Netzes.

Im nächsten Schritt verwenden wir den Verlust, um die Verknüpfungsgewichte des Netzes zu aktualisieren. Wie Sie aus *Neuronale Netze selbst programmieren* vielleicht wissen, müssen wir die Fehlergradienten an jedem Knoten ermitteln und mit ihnen die Verknüpfungsgewichte aktualisieren, die mit diesen Knoten verbunden sind.

PyTorch macht das wirklich einfach.

```
    # Gradienten nullen, einen Rückwärtsdurchgang ausführen und die Gewichte
    aktualisieren
    self.optimiser.zero_grad()
    loss.backward()
    self.optimiser.step()
```

Diese drei Schritte sind das Schlüsselmuster für fast alle neuronalen Netze, die mit PyTorch erstellt werden. Sehen wir sie uns im Einzelnen an:

- Zuerst werden alle Gradienten im Berechnungsgraphen mit `optimiser.zero_grad()` auf null gesetzt.
- Die Gradienten innerhalb des Netzes werden von der Verlustfunktion mit `loss.backward()` in Rückwärtsrichtung berechnet.
- Diese Gradienten werden verwendet, um die lernbaren Parameter des Netzes mit `optimiser.step()` zu aktualisieren.

Die Gradienten müssen wir bei jedem Training zunächst auf null setzen. Andernfalls akkumulieren sich die Werte bei jeder Berechnung der Gradienten mit `loss.backward()`.

Die Funktion `backward()` haben wir schon zuvor verwendet, um Gradienten für ein sehr einfaches Netz zu berechnen. Hier setzen wir sie genauso ein. Den letzten Knoten des Berechnungsgraphen können wir uns als Verlustfunktion vorstellen. Der an jedem Knoten berechnete Gradient, der in diesen Verlust eingeht, ist die Änderung im Verlust, wenn jeder lernbare Parameter variiert.

Der Optimierer aktualisiert mit den Gradienten die lernbaren Parameter, indem er auf dem Gradienten einen *Schritt* nach unten geht.

Wir haben nun endlich ein Netz, das trainiert werden kann. Bevor wir es trainieren, fügen wir noch eine Möglichkeit hinzu, um zu visualisieren, wie gut oder schlecht das Training verlaufen ist.

Das Training visualisieren

Als wir in *Neuronale Netze selbst programmieren* Netze trainiert haben, konnten wir den Fortschritt dieses Trainings praktisch nicht verfolgen. Zwar haben wir die Performance des Netzes nach dem Training gemessen, doch wir konnten uns kein Bild davon machen, wie reibungslos das Training selbst verlaufen ist oder ob weiteres Training hilfreich gewesen wäre.

Das Training lässt sich zum Beispiel verfolgen, wenn man den Verlust überwacht. Wir könnten dies bewerkstelligen, indem wir eine Kopie des Verlustwerts jedes Mal in einer Liste speichern, wenn er innerhalb von `train()` berechnet wird. Das würde aber eine sehr große Liste bedeuten, weil das Training neuronaler Netze oftmals mit Tausenden, wenn nicht Millionen von Beispielen durchgeführt wird. Der MNIST-Datensatz umfasst 60.000 Trainingsbeispiele, und wir könnten mehrere Epochen davon durchlaufen. Besser ist es, nach jeweils zehn Trainingsbeispielen eine Kopie des Verlustwerts zu behalten. Folglich müssten wir mit einem Zähler erfassen, wie oft `train()` ausgeführt wurde.

Der folgende Code erzeugt einen Zähler, der anfangs auf `0` gesetzt wird, und eine leere Liste namens `progress` im Konstruktor der Klasse für das neuronale Netz:

```
# Zähler und Akkumulator für progress
self.counter = 0
self.progress = []
```

In der Funktion `train()` können wir den Zähler (`counter`) inkrementieren und alle zehn Trainingsbeispiele den Verlustwert an das Ende der Liste anfügen.

```
# counter und accumulate-Fehler alle zehn Durchläufe inkrementieren
self.counter += 1
if (self.counter % 10 == 0):
    self.progress.append(loss.item())
    pass
```

Der Ausdruck `% 10` bezeichnet den Rest der ganzzahligen Division durch 10. Das Ergebnis ist nur `0`, wenn `counter` einen Wert wie `10`, `20`, `30` usw. enthält. Die hier verwendete Funktion `item()` dient lediglich dazu, einen Einzelwerttensor zu entpacken, um an die enthaltene Zahl zu gelangen.

Den Zähler `counter` können wir nach jeweils 10.000 Schritten ausgeben, um verfolgen zu können, wie langsam oder schnell das Training durch den Datensatz voranschreitet:

```
if (self.counter  % 10000 ==  0):
    print("counter = ", self.counter)
    pass
```

Um den Verlust als Diagramm zu veranschaulichen, fügen wir mit `plot_progress()` eine neue Funktion in die Klasse des neuronalen Netzes ein:

```
def plot_progress(self):
    df  = pandas.Dataframe(self.progress, columns=['loss'])
```

```
    df.plot(ylim=(0, 1.0), figsize=(16,8), alpha=0.1, marker='.',
                           grid=True, yticks=(0, 0.25,  0.5))
    pass
```

Der Code mag kompliziert aussehen, besteht aber nur aus zwei Zeilen. Die erste Zeile konvertiert die Liste der Verlustwerte `progress` in einen Pandas-Dataframe, damit wir diese Werte leicht in einem Diagramm darstellen können. Die an die Funktion `plot()` übergebenen Optionen legen Art und Erscheinungsbild des Diagramms fest.

Wir sind fast bereit, unser Netz zu trainieren!

Die Klasse für den MNIST-Datensatz

Wir haben zuvor die MNIST-Daten aus einer CSV-Datei in einen Pandas-Dataframe geladen. Die Daten könnten wir direkt aus dem Dataframe verwenden, was vollkommen in Ordnung wäre. Da wir jedoch etwas über PyTorch lernen wollen, sollten wir die Daten auch nach Art von PyTorch laden und verwenden.

PyTorch kann viele nützliche Dinge tun, beispielsweise die Daten automatisch mischen, mit mehreren Prozessen parallel laden und sie auch in Stapeln bereitstellen. Dies realisiert PyTorch mittels `torch.utils.data.DataLoader`, der die Daten selbst von einem `torch.utils.data.Dataset`-Objekt erwartet.

Um das Ganze überschaubar zu halten, verwenden wir weder Mischen noch Stapeln, sondern arbeiten mit der Klasse `torch.utils.data.Dataset`, um Erfahrungen im Umgang mit der Maschinerie von PyTorch zu sammeln.

Die PyTorch-Klasse `torch.utils.data.Dataset` importieren wir folgendermaßen:

```
from torch.utils.data import Dataset
```

Genau wie wir für ein neuronales Netz von der Klasse `nn.Module` erben und eine Funktion `forward()` bereitstellen, erben wir für ein Datensatz von `Dataset` und fügen die beiden folgenden speziellen Funktionen hinzu:

- `__len__()` gibt die Anzahl der Elemente im Datensatz zurück.
- `__getitem__()` gibt das *n*-te Element im Datensatz zurück.

Indem wir eine Klasse `MnistDataset` erstellen und die Methode `__len__()` definieren, kann PyTorch über `len(mnist_dataset)` die Größe des Datensatzes ermitteln. Die Methode `__getitem__()` erlaubt uns, auf Elemente über einen Index zuzugreifen, zum Beispiel mit `mnist_dataset[3]`.

Sehen Sie sich den folgenden Code mit der Klasse `MnistDataset` an:

```
class MnistDataset(Dataset):

    def __init__(self,  csv_file):
        self.data_df =  pandas.read_csv(csv_file, header=None)
        pass
```

```
    def __len__(self):
        return len(self.data_df)

    def __getitem__(self, index):
        # Zielbild (label)
        label = self.data_df.iloc[index,0]
        target =  torch.zeros((10))
        target[label] = 1.0

        # Bilddaten, von 0-255 auf 0-1 normalisiert
        image_values = torch.FloatTensor
                       (self.data_df.iloc[index,1:].values) / 255.0

        # Label, Bilddatentensor und Zieltensor zurückgeben
        return label, image_values, target

    pass
```

Wenn ein Objekt von dieser Klasse erstellt wird, liest es die Datei `csv_file` in einen Pandas-Dataframe namens `data_df`.

Die Funktion `__len__()` gibt einfach die Länge des Dataframes zurück.

Etwas interessanter ist die Funktion `__getitem__()`. Genau wie bei unseren früheren Experimenten mit den MNIST-Daten extrahieren wir ein Label aus dem `index`-ten Element im Datensatz (in die Variable `label`).

Außerdem erstellen wir einen Tensor, der die gewünschte Ausgabe des neuronalen Netzes repräsentiert. Wir füllen einen Tensor der Länge 10 mit Nullen, setzen aber den einen Eintrag, der dem Label entspricht, auf `1.0`. Ein Label `0` ergibt also einen Tensor `[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]`, und der Tensor für das Label `4` sieht wie `[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]` aus. Dies ist die sogenannte *1-aus-n-Codierung* (oder *One-Hot-Codierung*).

Aus den Pixelwerten des Bilds erzeugen wir dann einen Tensor `image_values`. Die Zahlenwerte dividieren wir durch `255`, um sie in einen Bereich von `0` bis `1` zu bringen.

Schließlich gibt `__getitem__()` alle drei Elemente zurück – das Label, die Bildwerte und das Ziel.

Auch wenn es für PyTorch nicht erforderlich ist, können wir eine Methode zur Klasse `MnistDataset` hinzufügen, um ein ausgewähltes Bild aus dem Datensatz grafisch darzustellen. Es hat es sich bewährt, eine Methode vorzusehen, um die verarbeiteten Daten anzuzeigen. Dafür müssen wir wie schon zuvor die Bibliothek `matplotlib.pyplot` importieren.

```
def plot_image(self, index):
    arr = self.data_df.iloc[index,1:].values.reshape(28,28)
    plt.title("label =  " + str(self.data_df.iloc[index,0]))
    plt.imshow(arr, interpolation='none', cmap='Blues')
    pass
```

Wir wollen nun überprüfen, ob alles bis dahin funktioniert. Zunächst erstellen wir ein Datensatzobjekt aus der Klasse und übergeben ihm den Speicherort der CSV-Datei:

```
mnist_dataset = MnistDataset('mount/My Drive/Colab Notebooks/myo_gan/mnist_data/
mnist_train.csv')
```

Der Konstruktor der Klasse lädt die Daten aus der CSV-Datei in einen Pandas-Dataframe. Mit unserer Funktion `plot_image()` zeichnen wir das zehnte Bild aus dem Datensatz. Der Index des zehnten Bilds ist `9`, da das erste Bild den Index `0` hat.

```
mnist_dataset.plot_image(9)
```

Dieser Code zeigt das Bild einer handgeschriebenen Vier an. Aus dem Label geht auch hervor, dass es eine Vier sein soll.

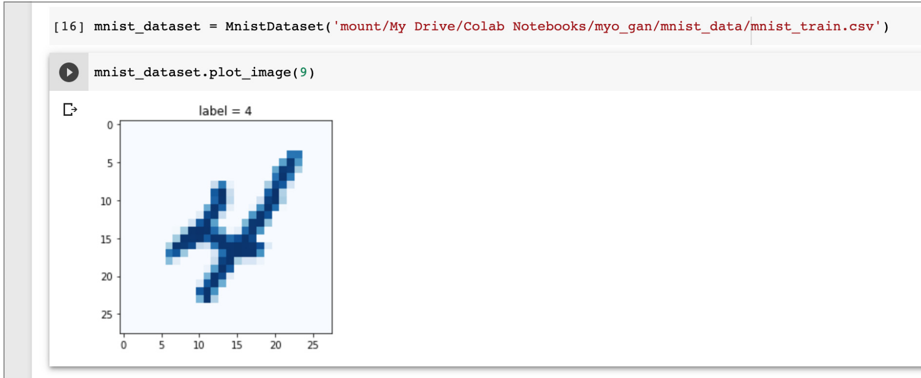

Abbildung 2-10: Das Bild einer handgeschriebenen Vier aus dem Datensatz anzeigen

Abbildung 2-10 zeigt die Bestätigung, dass unsere Datensatzklasse die Daten korrekt geladen hat.

Überprüfen Sie anhand von Beispielen wie `mnist_dataset[100]`, ob Sie über den Index auf `mnist_dataset` zugreifen können. Eine derartige Anweisung sollte das Label, die Pixelwerte und den Zieltensor zurückgeben.

Unsere Klassifizierer trainieren

Es ist jetzt wirklich einfach, ein klassifizierendes neuronales Netz zu trainieren, und zwar deshalb, weil wir bereits intensiv daran gearbeitet haben, die Datensatzklasse und die Klasse für das neuronale Netz einzurichten.

Zuerst erstellen wir ein neuronales Netz aus unserer Klassifiziererklasse `Classifier`:

```
# Neuronales Netz erstellen
C = Classifier()
```

Um das Netz zu trainieren, genügt wieder ein recht einfacher Code:

```
# Netz auf dem MNIST-Datensatz trainieren
for label, image_data_tensor, target_tensor in  mnist_dataset:
    C.train(image_data_tensor, target_tensor)
    pass
```

Da die mnist_dataset-Klasse von der PyTorch-Klasse Dataset erbt, können wir die gesamten Trainingsdaten mit einer eleganten for-Schleife durcharbeiten. Für jedes Trainingsbeispiel übergeben wir einfach die Bilddaten und den Zieltensor an die Methode train() des Klassifizierers.

Wie *Neuronale Netze selbst programmieren* gezeigt hat, kann es hilfreich sein, das gesamte Datensatz mehr als einmal zu durchlaufen, um das Netz zu trainieren. Die Trainingsschleife können wir ganz leicht mit einer äußeren Schleife umgeben, die die *Epochen* zählt.

Schließlich lässt sich mit Python-Notebooks sehr einfach ermitteln, wie viel Zeit eine Zelle beansprucht. Wir fügen lediglich ein %%time am Beginn der Zelle ein, deren Zeitbedarf wir messen wollen. Dies kann nützlich sein, wenn wir mit neuronalen Netzen experimentieren und abschätzen möchten, wie lange unsere Kaffeepause dauern darf, während das Netz trainiert!

Die Zelle sollte folgendermaßen aussehen:

```
%%time
# Neuronales Netz erstellen
C = Classifier()

# Netz auf dem MNIST-Datensatz trainieren
epochs  = 3

for i in range(epochs):
    print('training epoch', i+1, "of", epochs)
    for label, image_data_tensor, target_tensor in mnist_dataset:
        C.train(image_data_tensor, target_tensor)
        pass
    pass
```

Die Ausführung einer Zelle dauert einige Zeit. Nach jeweils 10.000 Aufrufen gibt die Methode train() den aktuellen Stand darüber aus, wie viele Beispiele sie abgearbeitet hat.

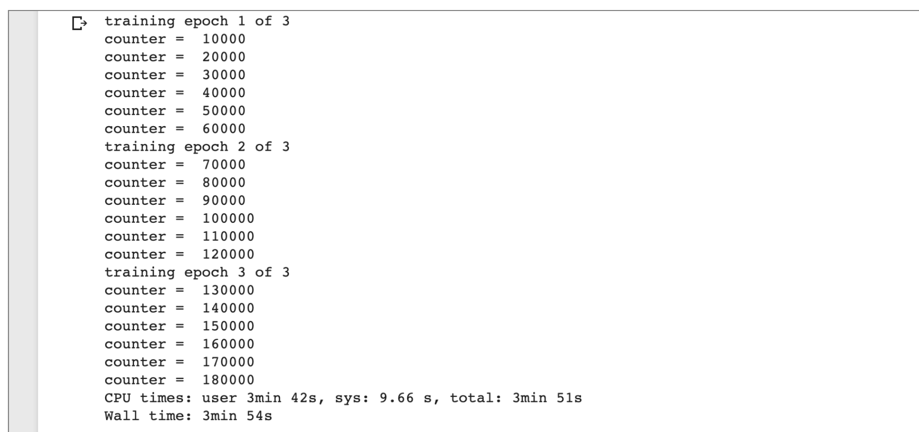

Abbildung 2-11: Training eines Netzes über mehrere Epochen hinweg und Anzeige der verstrichenen Zeit

Wie Abbildung 2-11 zeigt, haben drei Trainingsepochen rund vier Minuten benötigt. Das ist gar nicht schlecht, wenn man bedenkt, dass jede Epoche 60.000 Trainingsbeispiele bedeutet.

Stellen wir nun die gesammelten Verlustwerte grafisch dar, um einen Überblick über den Trainingsfortschritt zu bekommen:

```
# Klassifizierererfehler grafisch darstellen
C.plot_progress()
```

Es sollte ein Diagramm erscheinen, wie es in Abbildung 2-12 zu sehen ist. Die Darstellung kann bei Ihnen durchaus abweichen, denn das Training eines neuronalen Netzes ist grundsätzlich ein Zufallsprozess.

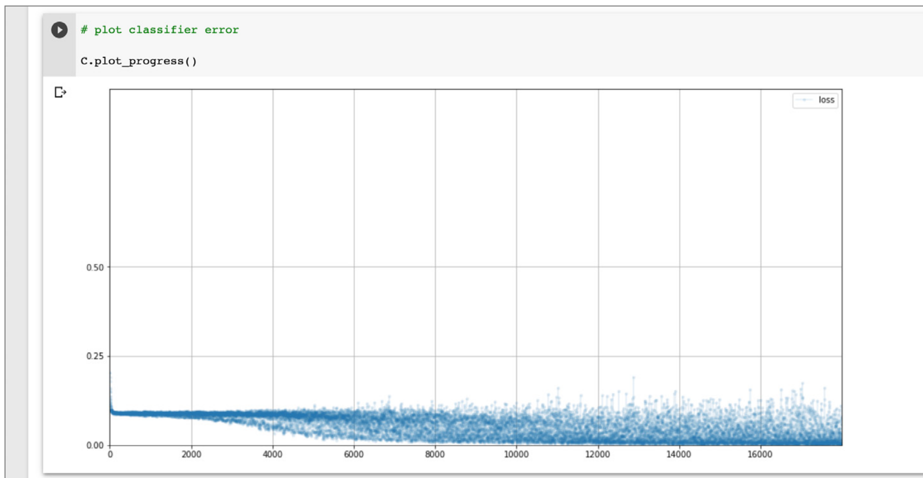

Abbildung 2-12: Grafische Darstellung des Klassifizierererfehlers über drei Epochen

Wie Abbildung 2-12 zeigt, fallen die Verlustwerte sehr schnell auf etwa `0.1` und gehen dann im Verlauf des Trainings langsamer und ziemlich verrauscht gegen null.

Ein fallender Verlust bedeutet, dass das Netz immer besser in der Lage ist, die Bilder korrekt zu klassifizieren.

Diese Verlustdiagramme sind wirklich nützlich. Wir können an ihnen ablesen, ob das Netztraining überhaupt funktioniert. Zudem vermitteln sie uns ein Gefühl dafür, ob das Training glatt und stetig oder instabil und chaotisch verläuft.

Das neuronale Netz abfragen

Nachdem wir über ein recht gut trainiertes Netz verfügen, wollen wir es abfragen, um Bilder zu klassifizieren. Wir wechseln zum MNIST-Testdatensatz mit 10.000 Bildern. Dabei handelt es sich um Bilder, die unser neuronales Netz noch nicht gesehen hat.

Wir laden nun den Datensatz mit einem neuen Dataset-Objekt:

```
# MNIST-Testdaten laden
mnist_test_dataset = MnistDataset('mount/My Drive/Colab Notebooks/gan/mnist_data/
mnist_test.csv')
```

Wir können aus dem Testdatensatz einen Datensatz auswählen und sehen, wie das Bild ausschaut. Der folgende Code wählt den 20. Datensatz aus, d.h. den Datensatz mit dem Index 19:

```
# Einen Datensatz auswählen
record  = 19

# Bild und richtiges Label darstellen
mnist_test_dataset.plot_image(record)
```

Das Bild zeigt eine Ziffer, die wie eine *4* aussieht. Auch das aus dem Datensatz extrahierte Label bestätigt, dass es eine *4* sein soll.

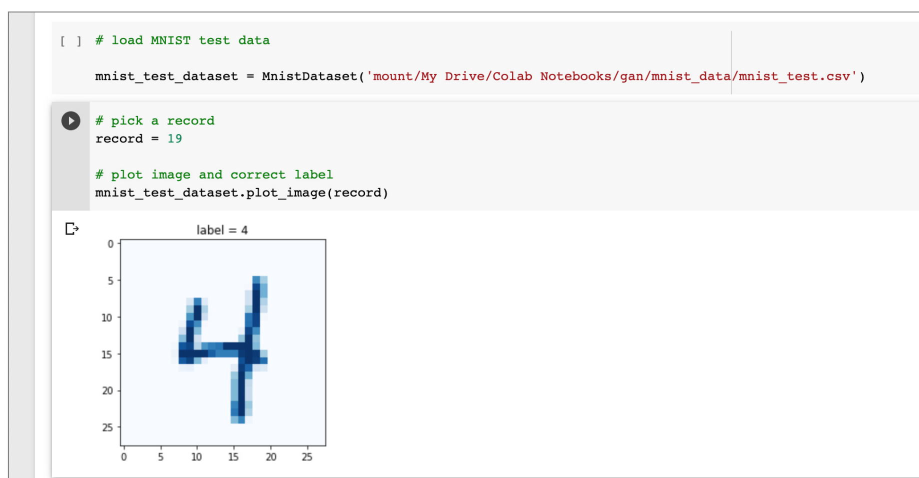

Abbildung 2-13: Aus dem Testdatensatz ausgewählter Datensatz, der eine handgeschriebene 4 darstellt

Sehen wir uns nun an, was unser trainiertes neuronales Netz von diesem Bild hält. Der folgende Code verwendet die oben auf 19 gesetzte Datensatznummer record, um die Pixelwerte des Bilds als image_data zu extrahieren. Das Bild wird mithilfe der Funktion forward() durch das neuronale Netz geschleust:

```
image_data = mnist_test_dataset[record][1]

# Aus trainiertem Netz abfragen
output  = C.forward(image_data)

# Ausgabetensor darstellen
pandas.DataFrame(output.detach().numpy()).plot(kind='bar',
legend=False, ylim=(0,1))
```

Der Code wandelt die Ausgabe `output` in ein einfacheres NumPy-Array um und hüllt sie dann in ein Dataframe, um sie einfach als Säulendiagramm darzustellen.

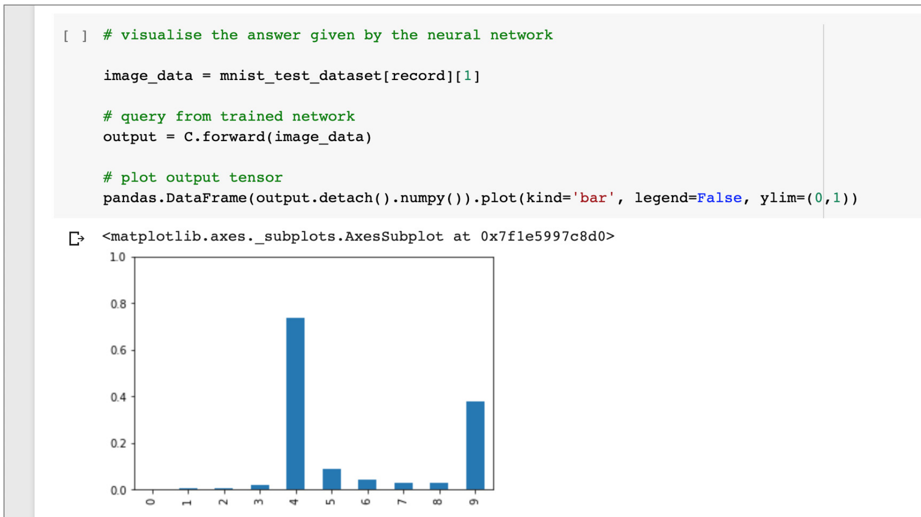

Abbildung 2-14: Die Ausgabe des neuronalen Netzes in Form eines Säulendiagramms

Die zehn Säulen verkörpern die Werte der zehn Ausgabeknoten des neuronalen Netzes. Den größten Wert weist Knoten *4* auf, woraus wir ableiten, dass das Netz das Bild als Ziffer Vier interpretiert.

Großartig! Das Netz hat das Testbild richtig klassifiziert.

Ein genauerer Blick auf die Ausgabe offenbart, dass die Werte aller anderen Knoten nicht null sind. Wir erwarten nicht, dass ein klassifizierendes neuronales Netz solch eindeutige Antworten liefert. Tatsächlich scheint das Netz die Bilddaten auch als *9* zu interpretieren, allerdings nicht so stark wie als *4*. Sieht man sich das eigentliche Bild genauer an, ist zu erkennen, dass der Unterschied zwischen einer *9* und einer *4* nicht immer klar ist.

Greifen Sie andere Datensätze heraus und probieren Sie es selbst aus. Datensatz 42 ist ein gutes Beispiel für ein mehrdeutiges Bild. Versuchen Sie auch, ein Bild zu finden, bei dem das neuronale Netz falsch liegt. Mein trainiertes Netz hat Datensatz 33 falsch interpretiert, und tatsächlich zeigt das Bild eine besonders schlecht geschriebene Ziffer.

Die Performance des Klassifizierers einfach ermitteln

Um festzustellen, wie treffsicher unser neuronales Netz die Bilder richtig klassifiziert, ist es am einfachsten, alle 10.000 Bilder im MNIST-Testdatensatz abzuarbeiten und zu zählen, wie viele Ziffern das Netz richtig erkennt. Hierzu vergleichen wir die Ausgabe des Netzes mit dem Label, das dem Bild zugeordnet ist.

Der folgende Code setzt die Trefferzahl score auf den Anfangswert 0 und geht dann die Testdaten durch. Die Trefferzahl wird jedes Mal inkrementiert, wenn das Label mit der Ausgabe des Netzes übereinstimmt.

```
# Neuronales Netz auf Trainingsdaten testen

score = 0
items = 0

for label, image_data_tensor, target_tensor in mnist_test_dataset:
    answer = C.forward(image_data_tensor).detach().numpy()
    if (answer.argmax() == label):
        score += 1
        pass
    items += 1

    pass

print(score, items, score/items)
```

Der Code answer.argmax() sucht den Index der Tensorantwort, die den größten Wert aufweist. Ist der erste Wert der größte, wird argmax auf 0 gesetzt. Das ist ein unkomplizierter Weg, um die Frage: »Welcher Knoten hat den größten Wert?« zu beantworten.

Schließlich geben wir die Trefferzahl, die Gesamtanzahl und die Trefferquote aus. Somit wissen wir, bei wie vielen Antworten das neuronale Netz richtig gelegen hat.

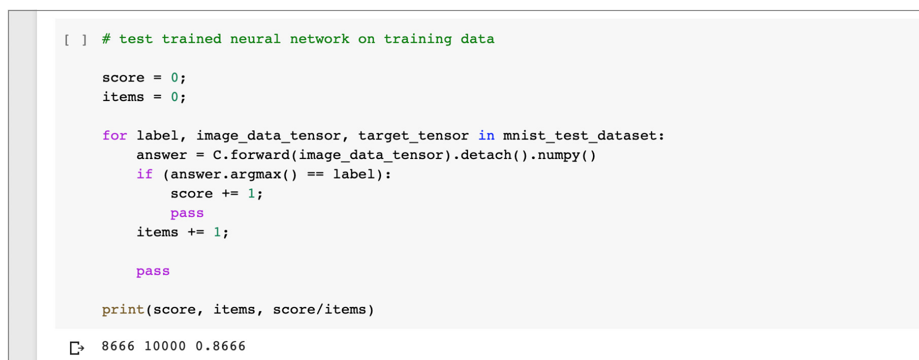

Abbildung 2-15: Die Performance des neuronalen Netzes einfach ermittelt

Ich habe eine Trefferquote von 87% erreicht, was angesichts der Einfachheit des neuronalen Netzes gar nicht schlecht ist.

Probieren Sie aus, ob Sie die Trefferquote verbessern können, wenn Sie das Netz für mehr als drei Epochen trainieren. Was passiert mit der Trefferquote, wenn Sie das Netz weniger als drei Epochen trainieren?

Den Code, den wir für diesen einfachen MNIST-Klassifizierer entwickelt haben, finden Sie online unter:

- *https://github.com/makeyourownneuralnetwork/gan/blob/master/03_mnist_classifier.ipynb*