I Übung zur Vorlesung Deep Learning

Einführung in Python

Die Vorlesung Deep Learning wird durch praktische Übungen begleitet, mit dem Ziel verschiedene Architekturen von Deep Neural Networks für eine Reihe von verschiedenen Anwendungsfällen zu bauen, zu trainieren und zu evaluieren. Dabei setzen wir auf die Programmiersprache Python (https://www.python.org/) und ein modernes Deep Learning Framework PyTorch (https://pytorch.org/). Die Übungen setzen keinerlei Vorkenntnis in Bezug auf Python oder PyTorch voraus und beginnen mit einem Einstieg in die Grundlagen. Allerdings werden Kenntnisse über grundlegende Konzepte aus der Programmierung vorausgesetzt, wie zum Beispiel über Variablen, Datentypen, arithmetische Operationen, Kontrollstrukturen und Schleifen, funktionale Programmierung, objektorientierte Programmierung, etc…

Einrichten der Entwicklungsumgebung

Python Distribution

Anaconda ist eine kostenlose, unternehmenstaugliche Python-Distribution für Datenanalyse und wissenschaftliche Datenverarbeitung. Im Rahmen der Vorlesung und den begleitenden Übungen wird Python 3.7 verwendet. Der Download der entsprechenden Anaconda Version befindet sich auf https://www.anaconda.com/download/.

Bei der Installation ist zu beachten, dass die Anaconda PATH Umgebungsvariable gesetzt wird. Dies kann während der Installation automatisch gemacht werden (entgegen der Warnung „nicht empfohlen“) oder nach der Installation manuell. Wenn Anaconda erfolgreich installiert wurde, kann dies in der Kommandozeile mit „*python --version*“ oder „*conda list*“ geprüft werden.

Python IDE

PyCharm ist eine leichtgewichtige IDE für die Entwicklung von Pythonprogrammen. Natürlich kann auch eine beliebige andere IDE verwendet werden. PyCharm kann auf https://www.jetbrains.com/pycharm/download/ heruntergeladen werden. Wenn die Installation abgeschlossen ist, lege ein neues Projekt an und erstelle eine Datei „main.py“. Für weitere Hinweise zur Nutzung von PyCharm siehe https://www.jetbrains.com/pycharm/documentation/.

Python Syntax

Grundlagen

Python’s Syntax ist auf Einfachheit und Lesbarkeit ausgelegt. Sie ist im Grundsatz ähnlich zu Perl, C und JAVA, allerdings gibt es ein paar definitive Unterschiede, die es zu beachten gilt.

Python Identifier

Ein Python Identifier ist ein Name, der verwendet wird, um eine Variable, Funktion, Klasse, Modul oder ein anderes Objekt zu identifizieren. Ein Identifier beginnt mit einem Buchstaben A bis Z oder a bis z oder einem Unterstrich (_), gefolgt von null oder mehr Buchstaben, Unterstrichen und Ziffern (0 bis 9). Python erlaubt keine Satzzeichen wie @, $ und % innerhalb von Bezeichnern. Python ist eine case-sensitive Programmiersprache. Somit sind **variable** und **Variable** zwei verschiedene Identifikatoren in Python.

Linien und Einrückungen

Python stellt keine Klammern zur Verfügung, um Codeblöcke für Klassen- und Funktionsdefinitionen oder Ablaufsteuerung anzuzeigen. Codeblöcke werden durch Zeileneinrückungen gekennzeichnet, die streng durchgesetzt werden. Die Anzahl der Leerzeichen in der Einrückung ist variabel, aber alle Anweisungen innerhalb des Blocks müssen um den gleichen Betrag eingerückt sein.

Beispiel:

```
if True:
    print("True")
else:
    print("False")
```

Der folgende Block erzeugt jedoch einen Fehler

```
if True:
print("True")
else:
print("False")
```

So würden in Python alle mit der gleichen Anzahl von Leerzeichen eingerückten fortlaufenden Linien einen Block bilden.

Mehrzeilige Anweisungen

Anweisungen in Python enden typischerweise mit einer neuen Zeile. Python erlaubt jedoch die Verwendung des Zeilenfortsetzungszeichens (\), um anzugeben, dass die Zeile fortgesetzt werden soll.

Beispiel:

Die folgenden beiden Anweisungen sind äquivalent

```
total = item_one + item_two + item_three
```

bzw.

```
total = item_one + \
        item_two + \
        item_three
```

Anweisungen, die in den Klammern [], {} oder () enthalten sind, müssen nicht das Zeilenfortsetzungszeichen verwenden.

Beispiel:

```
days = ["Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday"]
```

Zeichenketten in Python

Python akzeptiert einfache ('), doppelte (") und dreifache (''' oder """) Anführungszeichen, um Zeichenkettenliterale zu bezeichnen, solange die gleiche Art von Anführungszeichen die Zeichenkette beginnt und beendet. Die dreifachen Anführungszeichen werden verwendet, um die Zeichenkette über mehrere Zeilen zu verteilen.

Beispiel:

```
word = "word"
sentence = "This is a sentence."
paragraph = """This is a paragraph.
It is made up of multiple lines and sentences."""
```

Kommentare in Python

Ein Hashzeichen (#), das sich nicht in einem Zeichenkettenliteral befindet, beginnt einen Kommentar. Alle Zeichen nach dem # und bis zum Ende der physikalischen Zeile sind Teil des Kommentars und werden vom Python-Interpreter ignoriert.

Beispiel:

```
# First comment
print("Hello, World!")  # second comment
```

Weitere Details, Einzelheiten und Beispiele zur Python Syntax findest du unter https://www.tutorialspoint.com/python/python_basic_syntax.htm.

Variablen und Datentypen

Zuweisungen

Python-Variablen benötigen keine explizite Deklaration, um Speicherplatz zu reservieren. Die Deklaration erfolgt automatisch, wenn Sie einer Variablen einen Wert zuweisen. Das Gleichheitszeichen (=) wird verwendet, um Variablen Werte zuzuweisen. Der Operand links vom = Operator ist der Name der Variablen und der Operand rechts vom = Operator ist der in der Variablen gespeicherte Wert.

Beispiel:

```
counter = 100  # An integer assignment
miles = 1000.0  # A floating point
name = "John"  # A string
```

Mehrfachzuweisungen

Python erlaubt mehrere Zuweisungen in einer Zeile

Beispiel:

```
a = 1; b = 1; c = 1
```

oder

```
a = b = c = 1
```

oder

```
a, b, c = 1, 2, "John"
```

Datentypen

Python hat verschiedene Standarddatentypen, die verwendet werden, um die möglichen Operationen und die Speichermethode für jeden von ihnen zu definieren. Python bietet fünf gängig verwendete Standard Datentypen.

Numbers

Zahlen-Datentypen speichern numerische Werte. Zahlenobjekte werden angelegt, wenn ihnen ein Wert zugewiesen wird. Die Referenz auf ein Zahlenobjekt kann mit der Del-Anweisung gelöscht werden.

Beispiel:

```
# Erstelle Variable
var1 = 1

# Lösche Referenz
del var1
```

Python unterstützt vier verschiedene numerische Typen

- int (vorzeichenbehaftete Ganzzahlen)
- long (lange ganze Zahlen, sie können auch oktal und hexadezimal dargestellt werden)
- float (Gleitkomma-Istwerte)
- complex (komplexe Zahlen)

Strings

Zeichenketten in Python werden als eine zusammenhängende Menge von Zeichen identifiziert, die in den Anführungszeichen dargestellt werden. Python erlaubt entweder Paare von einfachen oder doppelten Anführungszeichen. Teilmengen von Zeichenketten können mit dem Slice-Operator ([ ] und [:]) aufgenommen werden, wobei die Indizes am Anfang der Zeichenkette bei 0 beginnen und am Ende von -1 weggehen. Das Pluszeichen (+) ist der Zeichenkettenverkettungsoperator und das Sternchen (*) ist der Wiederholungsoperator.

Beispiel:

```
str = 'Hello World!'

print(str)  # Prints complete string
>>> Hello World!

print(str[0])  # Prints first character of the string
>>> H

print(str[2:5])  # Prints characters starting from 3rd to 5th
>>> llo

print(str[2:])  # Prints string starting from 3rd character
>>> llo World!

print(str * 2)  # Prints string two times
>>> Hello World!Hello World!

print(str + "TEST")  # Prints concatenated string
>>> Hello World!TEST
```

Lists

Listen sind die vielseitigsten der zusammengesetzten Datentypen von Python. Eine Liste enthält durch Kommata getrennte Elemente, die in eckige Klammern ([]) eingeschlossen sind. In gewissem Maße sind Listen den Arrays in C ähnlich. Ein Unterschied besteht darin, dass alle Elemente, die zu einer Liste gehören, einen unterschiedlichen Datentyp haben können. Auf die in einer Liste gespeicherten Werte kann mit dem Slice-Operator ([ ] und [:]) zugegriffen werden, wobei die Indizes am Anfang der Liste bei 0 beginnen und bis Ende -1 arbeiten. Das Pluszeichen (+) ist der Listenverkettungsoperator und das Sternchen (*) ist der Wiederholungsoperator.

Beispiel:

```
list = ["abcd", 786, 2.23, "John", 70.2]
tinylist = [123, "John"]

print(list)  # Prints complete list
>>> ["abcd", 786, 2.23, "John", 70.2]

print(list[0])  # Prints first element of the list
>>> abcd

print(list[1:3])  # Prints elements starting from 2nd till 3rd
>>> [786, 2.23]

print(list[2:])  # Prints elements starting from 3rd element
>>> [2.23, "John", 70.2]

print(tinylist * 2)  # Prints list two times
>>> [123, "John", 123, "John"]

print(list + tinylist)  # Prints concatenated lists
>>> ["abcd", 786, 2.23, "John", 70.2, 123, "John"]
```

Tuples

Ein Tupel ist ein weiterer Sequenzdatentyp, der der Liste ähnlich ist. Ein Tupel besteht aus einer Reihe von Werten, die durch Kommas getrennt sind. Im Gegensatz zu Listen werden Tupel jedoch in Klammern eingeschlossen. Die Hauptunterschiede zwischen Listen und Tupeln sind: Listen sind in Klammern ([ ]) eingeschlossen und ihre Elemente und Größe können geändert werden, während Tupel in Klammern (( )) eingeschlossen sind und nicht aktualisiert werden können. Tupel können als schreibgeschützte Listen betrachtet werden.

Beispiel:

```
tuple = ("abcd", 786 , 2.23, "John", 70.2)
tinytuple = (123, "John")

print(tuple)  # Prints complete list
>>> ("abcd", 786, 2.23, "John", 70.2)

print(tuple[0])  # Prints first element of the list
>>> abcd

print(tuple[1:3])  # Prints elements starting from 2nd till 3rd
>>> (786, 2.23)

print(tuple[2:])  # Prints elements starting from 3rd element
>>> (2.23, "John", 70.2)

print(tinytuple * 2)  # Prints list two times
>>> (123, "John", 123, "John")

print(tuple + tinytuple)  # Prints concatenated lists
>>> ("abcd", 786, 2.23, "John", 70.2, 123, "John")
```

Der folgende Code ist mit Tupel ungültig, da wir versucht haben, ein Tupel zu aktualisieren, was nicht erlaubt ist. Ein ähnlicher Fall ist bei Listen möglich.

```
tuple = ("abcd", 786, 2.23, "John", 70.2)
list = ["abcd", 786, 2.23, "John", 70.2]
tuple[2] = 1000  # Invalid syntax with tuple
list[2] = 1000  # Valid syntax with list
```

Dictionaries

Die Dictionaries von Python sind eine Art Hash-Tabelle. Sie funktionieren wie assoziative Arrays oder Hashes in Perl und bestehen aus Schlüssel-Wert-Paaren. Ein Dictionary-Schlüssel kann fast jeder Python-Typ sein, ist aber in der Regel eine Zahl oder eine Zeichenkette. Werte können dagegen jedes beliebige Python-Objekt sein. Dictionaries werden von geschweiften Klammern ({ }) umschlossen und Werte können über eckige Klammern ([]) zugewiesen und aufgerufen werden. Dictionaries haben keinen Begriff von Ordnung zwischen den Elementen.

Beispiel:

```
dict = {}
dict["one"] = "This is one"
dict[2] = "This is two"

tinydict = {"name": "John",
            "code":6734,
            "dept": "sales"}

print(dict["one"])  # Prints value for "one" key
>>> This is one

print(dict[2])  # Prints value for 2 key
>>> This is two

print(tinydict)  # Prints complete dictionary
>>> {"dept": "sales", "code": 6734, "name": "John"}

print(tinydict.keys())  # Prints all the keys
>>> ["dept", "code", "name"]

print(tinydict.values())  # Prints all the values
>>> ["sales", 6734, "John"]
```

Operatoren

Operatoren sind die Konstrukte, die den Wert von Operanden manipulieren können. Python unterstützt die folgenden Arten von Operatoren.

- Arithmetische Operatoren
- Vergleich (Vergleichs-)Operatoren
- Zuweisungsoperatoren
- Logische Operatoren
- Bitweise Operatoren
- Mitgliedschaftsoperatoren
- Identitätsoperatoren

Beispiele zu den einzelnen Operatoren findest du unter https://www.tutorialspoint.com/python/python_basic_operators.htm

Kontrollstrukturen und Schleifen

Python bietet alle gängigen Arten von Entscheidungsanweisungen, wie einfache if-statements, if-else-statements und verschachtelte if-statements.

Beispiel:

```
var = 100
if var < 200:
   print("Expression value is less than 200")
   if var == 150:
      print("Which is 150")
   elif var == 100:
      print("Which is 100")
   elif var == 50:
      print("Which is 50")
   elif var < 50:
      print("Expression value is less than 50")
else:
   print("Could not find true expression")

print("Good bye!")

>>> Expression value is less than 200
>>> Which is 100
>>> Good bye!
```

Darüber hinaus bietet Python die gängigen Arten von Schleifen, wie while-Schleifen, for-Schleifen und verschachtelte Schleifen. Innerhalb von Schleifen lassen sich Schleifenkontrollanweisungen verwenden um die Ausführung von ihrer normalen Reihenfolge zu ändern. Wenn die Ausführung einen Bereich verlässt, werden alle automatischen Objekte, die in diesem Bereich erstellt wurden, zerstört.

Beispiel:

```
max = 20
for i in range(2, max):
   j = 2
   while j <= (i/j):
      if i%j != 0:
          break
      j += 1
   if j > i/j:
       print(i, "is prime")

print("Good bye!")

>>> 2 is prime
>>> 3 is prime
>>> 5 is prime
>>> 7 is prime
>>> 11 is prime
>>> 13 is prime
>>> 17 is prime
>>> 19 is prime
```

Weitere Beispiele zu Kontrollstrukturen findest du unter https://www.tutorialspoint.com/python/python_decision_making.htm und https://www.tutorialspoint.com/python/python_loops.htm

Funktionen

Python bietet die Möglichkeit eigene Funktionen zu erstellen. Funktionsblöcke beginnen mit dem Schlüsselwort **def**, gefolgt von dem Funktionsnamen und Klammern (()). Alle Eingabeparameter oder Argumente sollten innerhalb dieser Klammern platziert werden. Es können auch Parameter innerhalb dieser Klammern definiert werden. Die erste Anweisung einer Funktion kann eine optionale Anweisung sein. Der Codeblock innerhalb jeder Funktion beginnt mit einem Doppelpunkt (:) und ist eingerückt. Die Anweisung **return** [expression] beendet eine Funktion und gibt optional einen Ausdruck an den Aufrufer zurück. Eine **return**-Anweisung ohne Argumente ist gleichbedeutend mit **return None**.

Beispiel:

```
def f_prime(max):
    primes = list()
    for i in range(2, max):
       j = 2
       while j <= (i/j):
          if i%j != 0:
              break
          j += 1
       if j > i/j:
           primes.append(i)

    return primes

primes = f_prime(max=20)
print(primes)

>>> [2, 3, 5, 7, 11, 13, 17, 19]
```

Scope von Variablen

Alle Variablen in einem Programm sind möglicherweise nicht an allen Stellen in diesem Programm zugänglich. Dies hängt davon ab, wo die Variable deklariert wurde. Der Scope einer Variablen bestimmt den Teil des Programms, in dem auf diese Variable zugegriffen werden kann. Es gibt zwei grundlegende Bereiche von Variablen in Python. Variablen, die innerhalb eines Funktionskörpers definiert sind, haben einen lokalen und solche, die außerhalb eines Funktionskörpers definiert sind, einen globalen Umfang. Das bedeutet, dass auf lokale Variablen nur innerhalb der Funktion zugegriffen werden kann, in der sie deklariert sind, während auf globale Variablen im gesamten Programmkörper von allen Funktionen zugegriffen werden kann. Wenn Sie eine Funktion aufrufen, werden die darin deklarierten Variablen in den Geltungsbereich gebracht.

Beispiel:

```
total = 0  # This is global variable.

def sum(arg1, arg2):
   # Add both the parameters and return them."
   total = arg1 + arg2  # Here total is local variable.
   print("Inside the function local total:", total)
   return total

sum(10, 20)
print("Outside the function global total: ", total)

>>> Inside the function local total: 30
>>> Outside the function global total: 0
```

Weitere Einzelheiten und Details zu Funktionen in Python findest du unter https://www.tutorialspoint.com/python/python_functions.htm.

Klassen und Objekte

Python ist seit seiner Existenz eine objektorientierte Sprache. Aus diesem Grund ist das Erstellen und Verwenden von Klassen und Objekten ausgesprochen einfach. Die Einführung von Objektorientierter Programmierung geht über den Rahmen dieser Übung hinaus, kann aber speziell für Python unter https://www.tutorialspoint.com/python/python_classes_objects.htm gefunden werden.

Übungsaufgaben

Aufgabe 1

a) Schreibe eine Funktion, welche die Fibonacci Zahlen ausgibt. Erlaube eine variable Wahl der Anzahl an ausgegebenen Zahlen über ein Argument der Funktion. Bedenke das Verhalten deiner Funktion für eine negative Anzahl.

b) Schreibe eine zweite Variante deiner Funktion, die nur die letzte Zahl der Zahlenfolge ausgibt und sie sich selber aufruft, d.h. rekursiv ist.

Aufgabe 2

a) Schreibe eine Funktion, welche die Fakultät einer gegebenen Zahl ausgibt. Erlaube eine variable Wahl der Zahl als Funktionsargument. Beachte die Definition der Fakultät für negative Zahlen.

b) Schreibe eine zweite Variante deiner Funktion, die sie sich selber aufruft und rekursiv ist.

Aufgabe 3

a) Gegeben sind zwei Listen A und B mit jeweils vier Elementen, $A = [1, 2, 3, 9]$ und $B = [1, 2, 4, 4]$, sowie eine vorgegebene Summe $Y = 8$. Schreibe eine Funktion, die jeweils eine der beiden Listen und die Summe als Argument zugeführt bekommt und prüft, ob die Addition zweier Elemente aus der Liste die Summe ergibt. D.h. prüfe, ob die Aussage $\exists(x_i, x_j)$ mit $x \in \{A, B\}$ und $i \in \{1, 2, 3, 4\}$, sodass $x_i + x_j = Y$ wahr ist. Die Funktion soll entweder **True** (für B), oder **False** (für A) zurückgeben.

b) Eine einfache Lösung hat eine Komplexität von n^2, wobei n die Anzahl an Elementen in den Listen A und B ist (im obigen Beispiel $n = 4$). Das bedeutet, dass im Mittel n^2 Prüfvorgänge durchgeführt werden, bis die Funktion die richtige Antwort zurückgibt. Schreibe eine weitere Variante der Funktion, welche die Komplexität n hat, sodass im Mittel nur n Prüfvorgänge durchgeführt werden. **Hinweis:** Nutze aus, dass die beiden Listen geordnet sind!

c) **Die Listen sind nun nicht mehr geordnet!** Schreibe eine dritte Variante der Funktion mit der Komplexität n, welche für ungeordnete Listen funktioniert. Prüfe deine Funktion, indem die Elemente der Argumente A und B zunächst durchgemischt werden. **Hinweis:** Benutze die Tatsache, dass der Lookup einer hashmap in der Regel von der Ordnung $\mathcal{O}(1)$ ist. In Python sind dictionaries das Äquivalent zu hashmaps.