NNW-Übung 6

1 PyTorch: Tensors

Tensoren sind verallgemeinerte Matrizen: Matrizen sind 2D, Tensoren können beliebige Dimensionen haben. PyTorch-Tensoren verhalten sich **sehr ähnlich zu Numpy-Arrays** und können von/nach Numpy konvertiert werden. PyTorch-Tensoren können einiges mehr als Numpy-Arrays, z.B.:

- Mit PyTorch-Tensoren können Berechnungen erstellt werden, die *automatisch abgeleitet* (differenziert) werden können („*autograd*“).
- PyTorch-Tensoren können sich auf der CPU oder auf einem „Accelerator“ wie z.B. einer GPU befinden.

Entsprechend haben PyTorch-Tensoren einige zusätzliche Methoden, z.B.:

- Die Methode `numpy()` gibt den PyTorch-Tensor als Numpy-Array zurück – wenn möglich: „only if the tensor is on the CPU, does not require grad, does not have its conjugate bit set, and is a dtype and layout that NumPy supports“. Durch den Parameter `force=True` lässt sich das erzwingen.
- Eine Methode torch.Tensor.cpu() kopiert den Tensor auf die CPU (falls nötig). Für die umgekehrte Richtung, bzw. allgemein zum Konvertieren von Tensoren (sowohl des Datentyps als auch wo sich der Tensor befindet), gibt es die Methode `torch.Tensor.to`.
- Eine wichtige Methode im Zusammenhang mit *autograd* ist torch.Tensor.detach(), die einen neuen Tensor *ohne* Gradientberechnung zurückliefert – nur Tensoren ohne Gradientberechnung (die zudem auf der CPU sind) können nach Numpy konvertiert werden.

a) Eine *kompakte Einführung* zu PyTorch-Tensoren finden Sie hier: `https://pytorch.org/tutorials/beginner/basics/tensorqs_tutorial.html`. Sie werden viele Parallelen zu Numpy feststellen. Lesen Sie diese Einführung in Ruhe durch!

b) Was ist der Unterschied zwischen `torch.tensor(X_np)` und `torch.tensor.from_numpy(X_np)`? Dabei sei `X_np` ein Numpy-Array.

c) Was ist der Unterschied zwischen den `torch.tensor`-Methoden `numpy()`, `tolist()` und `item()`?

d) Was ist der Unterschied zwischen `X[:, -1]` und `X[..., -1]`? Dabei sei `X` ein Numpy-Array oder ein PyTorch-Tensor.

e) Optional (erst *nach* den übrigen Aufgaben bei Interesse ansehen): Eine *ausführlichere Einführung* („deeper tutorial“) mit einem YouTube-Video gibt es hier (für das heutige Aufgabenblatt reicht aber die vorher erwähnte kompakte Einführung): `https://pytorch.org/tutorials/beginner/introyt/tensors_deeper_tutorial.html`

2 PyTorch Autograd: Automatic Differentiation

Die Funktionsweise der automatischen Differentiation (Ableitung) bei PyTorch wird hier erklärt: `https://pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html`.

Besonders wichtig sind:

- Der Parameter `requires_grad` von `torch.tensor`.
- Die Methode `backward()` von `torch.tensor`.

a) Lesen Sie das genannte Tutorial bis VOR den Abschnitt *Optional Reading* in Ruhe durch!

b) Schreiben Sie PyTorch-Code, der die Funktion $f(\vec{x}) = f(x_1, x_2) = x_1^2 + sin(x_2) + 1$ implementiert.

c) Berechnen Sie selbst (auf Papier) die Funktionswerte an den Stellen $\vec{u} = (0, 0)$ und $\vec{v} = (1, \pi/2)$ ... und lassen danach PyTorch diese Funktionswerte berechnen und vergleichen Sie die Ergebnisse.

d) Berechnen Sie selbst (auf Papier) die Gradienten an den oben genannten Stellen $\vec{u}$ und $\vec{v}$ ... und lassen Sie danach PyTorch die Gradienten berechnen und vergleichen Sie die Ergebnisse.

e) Optional (erst *nach* den übrigen Aufgaben bei Interesse ansehen): Auch zu Autograd gibt es eine ausführlichere Einführung mit einem YouTube-Video (für das heutige Aufgabenblatt reicht aber die vorher erwähnte kompakte Einführung):
`https://pytorch.org/tutorials/beginner/introyt/autogradyt_tutorial.html`

3 Umbau der MLN-Klasse für PyTorch

Erstellen Sie eine Lösung zum optionalen Aufgabenblatt 5 so, dass die Berechnung des Gradienten von PyTorch übernommen wird.

Einige Hinweise hierzu:

- *Im Hauptprogramm* sollen nur Numpy-Arrays verwendet werden – entsprechend soll auch `MLN.DeltaTrain` Numpy-Arrays als Parameter erhalten.
- Verwenden Sie für Berechnungen der Gewichts-Updates die Methode `MLN.y(...)` (also ohne die Aktivierungsfunktion der Ausgabeschicht).
- Setzen SIe die Gradienten-Komponenten nach dem Update der Gewichte durch Aufruf der Methode `zero_()` auf 0, also zum Beispiel `self._W1.grad.zero_()`. Grund: `backprop()` addiert den berechneten Gradienten auf. Mehr dazu auf dem nächsten Aufgabenblatt.
- Beachten Sie: Gradienten-Komponenten (also `.grad`) gibt es (im Normalfall) nur für „leaf tensors“ (Blätter des Graphen), also Tensoren, die *nicht* durch Berechnung aus anderen Tensoren hervorgegangen sind.
- `nnwplot.plotTwoFeatures` soll *nicht* verändert werden und kommt (ohne Änderungen) nur mit Numpy-Arrays zurecht. Daher müssen die übergebenen Parameter (`X` und `T`) Numpy-Arrays sein und die übergebene Funktion (Parameter `pred_func`) muss so geschrieben sein, dass sie Numpy-Arrays als Input akzeptiert und als Outoput liefert.
- Für das Gewichts-Update muss für die zu ändernden Tensoren im Moment der Änderung die Gradientenberechnung abgeschaltet sein. Dazu gibt es den context manager `torch.no_grad`: Einen Block mit abgeschalteter Gradientenberechnung beginnen Sie mit:
`with torch.no_grad():`
- Ein `+=` können Sie mit der Methode `add_` von `torch.tensor` realisieren. Der Unterstrich „_“ ist *kein* Tippfehler, er bedeutet „in-place“ Änderung – auch bei anderen PyTorch-Methoden.