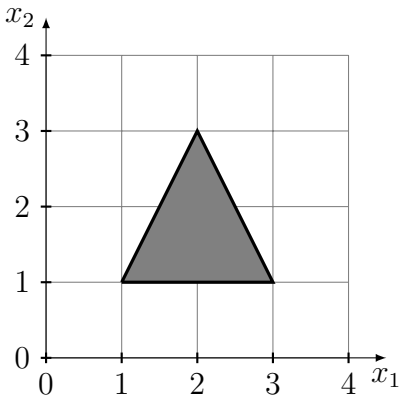
3. Übungsblatt

Aufgabe 12 Netze von Schwellenwertelementen

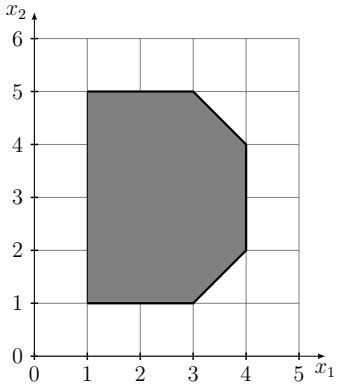
Geben Sie ein neuronales Netz aus Schwellenwertelementen an, das für Punkte (x_1, x_2) innerhalb des in der nebenstehenden Skizze gezeigten Dreiecks den Wert 1 und für Punkte außerhalb den Wert 0 liefert!



Hinweis: Erinnern Sie sich an das in der Vorlesung behandelte neuronale Netz zur Lösung des Biimplikationsproblems und interpretieren Sie die Berechnungen der Schwellenwertelemente der ersten Schicht als eine Koordinatentransformation.

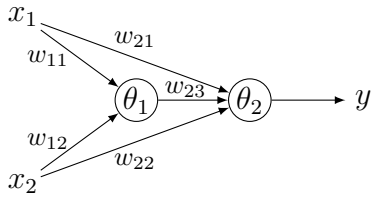
Aufgabe 13 Netze von Schwellenwertelementen

Geben Sie ein neuronales Netz aus Schwellenwertelementen an, das für Punkte (x_1, x_2) innerhalb des in der nebenstehenden Skizze gezeigten Buchstaben D den Wert 1 und für Punkte außerhalb den Wert 0 liefert! Achten Sie darauf, dass lediglich dick gedruckte Kanten des Randes eine 1 liefern sollen. Der dünn gedruckte Rand auf der linken Seite soll nicht Teil des Netzes sein.



Aufgabe 14 Netze von Schwellenwertelementen

Bestimmen Sie die Parameter w_{ji} und θ_j des in der nebenstehenden Skizze gezeigten neuronalen Netzes, sodass dieses Netz das exklusive Oder der Boole'schen Variablen x_1 und x_2 berechnet (d.h. $y = x_1 \dot{\vee} x_2$ bzw. $y = x_1 \oplus x_2$)!



Hinweis: Gehen Sie von einer geometrischen Interpretation der Berechnung im Eingaberaum des rechten Neurons aus und überlegen Sie, wie Sie die Ausgabe des linken Neurons verwenden können, um die Punkte (x_1, x_2), für die 1 bzw. 0 geliefert werden soll, so anzuordnen, dass sie durch eine Ebene trennbar werden.

Aufgabe 15 Darstellung Boole'scher Funktionen

Nuten Sie den Algorithmus aus der Vorlesung zur Repräsentation von Boole'schen Funktionen, um die folgende Funktion durch ein Netz aus Schwellenwertelementen darzustellen.

$(x_1 \vee x_3) \wedge \neg(\neg x_1 \wedge x_2) \wedge (\neg x_2 \vee x_3)$

Aufgabe 16 Bonus: Automatische Differenzierungsframework (1)

In diesem Semester wollen wir ein automatisches Differenzierungsframework ähnlich wie Tensorflow, Pytorch oder JAX implementieren. Dafür erstellen wir nach und nach unterschiedliche differenzierbare Funktionen. Dabei wollen wir uns nicht auf die Operationen an sich konzentrieren sondern auf den Weg, wie die Funktionalität umgesetzt wird. Aus diesem Grund empfehlen wir die Aufgaben in Python mit numpy zu lösen (Wer experimentieren möchte und eine NVIDIA-GPU hat kann auch versuchen cupy zu verwenden).
Um die Aufgaben ein wenig zu erleichtern stellen wir die Vorlage *adf.py* zur Verfügung. Bevor wir zu den Aufgaben kommen, gehen wir kurz durch die vorhandenen Funktionen.
Wir fangen an mit der Klasse *Tensor*. Dies ist die Klasse auf der alle Operationen arbeiten sollen. Gehen wir einmal kurz durch die Parameter und wie Sie zu verwenden sind. Beim Betrachten der Vorlagen sind euch bestimmt die *@differentiable* Dekoratoren aufgefallen. Dieser nutzt die Funktionen am Anfang der Datei und erleichtert euch das Leben, indem er in den meisten Fällen die Verwaltung von _u übernimmt. Zusätzlich sorgt er dafür, dass nach dem Aufruf von *backward()* alle Vorgänger grad_fn()s aufgerufen werden.

d	An dieser Stelle werden die Werte die im Tensor sind als numpy Arrays gespeichert.
grad	Dieser Parameter beinhaltet den akkumulierten Gradienten als numpy Array der später zur Anpassung der Gewichte verwendet werden soll
_u	Dies ist ein Zähler, der zählt wie oft dieser Tensor in Operationen verwendet wurde.
grad_fn	Wenn der Tensor durch eine Operation erstellt wurde, wird hier eine Funktion gespeichert die den eigenen Gradienten als Parameter bekommt und ihren Einfluss auf den Gradienten der Vorgänger berechnet.
shape	Damit die Klasse bequemer zu nutzen wird stellt der Parameter direkten Zugriff auf die Dimensionen des Tensors
backward()	Startet die Gradientenberechnung von einem Skalar bezüglich aller Variablen die in die Berechnung eingeht.

a) Kommen wir nun zur eigentliche Aufgabe. Dieses mal wollen wir die ersten Operationen implementieren um ein lineares Model zu trainieren. Dafür sollt ihr im ersten Schritt die Funktionen ___sub___(), ___mul___(), ___matmul___() und sum() implementieren. Das bedeutet ihr erstellt einen neuen Tensor mit dem Ergebnis der Operation bei dem die grad_fn wie oben beschrieben gesetzt wird. Als Beispiel wie eine funktionierende Funktion aussieht könnt Ihr die Implementation von ___add___ ankucken. Die Operation erlauben dann die Benutzung von +, -, *, @ als elementweise Addition, Subtraktion und Multiplikation und Matrixmultiplikation. Die sum() Funktion soll alle Werte im Tensor addieren und einen Tensor mit einem Skalar zurückgeben.
Zum Testen könnt ihr die Datei einfach ausführen, da für die einfachsten Fälle Tests hinterlegt sind, die geeignete Implementationen erfüllen sollten.

b) Nun wollen wir unsere implementierten Funktionen an einer kleinen Beispielaufgabe testen. Dafür verwendet Ihr das Jupyter-Notebook Experiment.ipynb und den Datensatz 'gender_height_weight.csv'. Wir wollen ein lineares Model trainieren das $Wx + b$ berechnet. Anschließend sollen W und b mit Stochastischem Gradientenabstieg trainiert werden.
Ihr könnt zusätzlich noch mit der Lernrate und der Initialisierung experimentieren.

Materialien:

- Python, Jupyter-Notebook : `https://www.anaconda.com/distribution/`
- Numpy Einführung: `https://jalammar.github.io/visual-numpy/`
- Python Tutorial : `https://docs.python.org/3/tutorial/`