Deel II. Deep learning

Hoofdstuk 3. Neurale netwerken

Inleiding

Dit is het Jupyter Notebook behorende bij hoofdstuk 3 van het vak *Advanced Datamining* (BFVH4DMN2). Op BlackBoard tref je eveneens een module `data.py` aan die diverse functies bevat die helpen bij het genereren en het visualiseren van de gebruikte datasets. Kopieer het bestand `model.py` van het vorige hoofdstuk en sla deze bestanden gezamenlijk op in één werkmap. Open je `model` module in een code-editor naar keuze om hiermee verder te werken.

Laten we weer beginnen om deze functies te importeren, samen met wat initialisatie-code en enkele onderdelen van de modules `pandas`, `numpy` en `tensorflow`. Plaats de cursor in de cel hieronder en druk op Ctrl+Enter (of Shift+Enter om meteen naar de volgende cel te gaan).

In [1]:
```
import warnings
warnings.filterwarnings("ignore", category=UserWarning)

%matplotlib inline
%reload_ext autoreload
%autoreload 2

from sys import version
print(f'Using python version {version.split(" ")[0]}')

from pandas import DataFrame, __version__
print(f'Using pandas version {__version__}')

from numpy import array, __version__
print(f'Using numpy version {__version__}')

from tensorflow import keras, __version__
print(f'Using tensorflow version {__version__}')
```

```
import vlearning
from vlearning import data, __version__
from vlearning import activation_functions, loss_functions, layers
print(f'Using vlearning version {__version__}')
```

```
Using python version 3.11.8
Using pandas version 2.2.1
Using numpy version 1.26.4
Using tensorflow version 2.15.0
Using vlearning version 0.3.7
```

> **Opmerking:**
>
> Als `numpy` of `tensorflow` niet geïnstalleerd is op je systeem, voer dan `pip3 install numpy tensorflow` uit (of `pip3 install numpy tensorflow[and-cuda]` als je ondersteuning voor een GPU wil inschakelen). Herstart de python kernel via de menu-optie `Kernel` > `Restart` van dit notebook.
>
> Tensorflow geeft waarschijnlijk allerlei mededelingen en waarschuwingen. Deze vormen voor het uitvoeren van dit notebook gewoonlijk geen probleem, en zelfs zonder tensorflow kun je evengoed je eigen module verder ontwikkelen.

Het multi-layer perceptron

In dit hoofdstuk gaan we de eerder gemaakte neuronen tot parallel en serieel aan elkaar gekoppelde lagen uitbreiden om hiermee *deep learning* te bedrijven. We gaan meerdere typen lagen definiëren, waaronder een input laag, lagen met neuronen die lineaire combinaties van attributen maken, afzonderlijke lagen die daar activatiefuncties op toepassen, en tenslotte nog een finale laag die een loss-functie toepast om de kwaliteit van de fit te berekenen. Omdat de lagen eigenschappen delen leent dit zich bij uitstek voor een object-georiënteerde opzet.

We beginnen met het definiëren van onze Mother Of All Layers: een parent-class `Layer()` waarvan we diverse child-classes zullen afleiden. De `Layer()` class hieronder houdt een instance variabele `next` bij die verwijst naar de volgende neurale laag (of `None` als het de laatste laag betreft), vergelijkbaar met een *linked list* datastructuur. Verder wordt het aantal `inputs` naar de laag en het aantal `outputs` vanuit de laag bijgehouden. Het aantal outputs dient door de gebruiker te worden gespecificeerd tijdens het initialiseren van het `Layer()` object; het aantal inputs wordt later automatisch bepaald middels de `set_inputs()` methode zodra de laag aan een voorafgaande laag wordt gekoppeld met de `add()` methode. Tenslotte heeft elke laag een naam opdat we opeenvolgende lagen eenvoudiger kunnen onderscheiden.

Je kan het onderstaande fragment letterlijk overnemen. Bestudeer de werking zodat je begrijpt wat deze code doet, en voeg desgewenst commentaren en docstrings toe.

```
from collections import Counter

class Layer():

        layercounter = Counter()

    def __init__(self, outputs, *, name=None, next=None):
        Layer.layercounter[type(self)] += 1
        if name is None:
            name =
f'{type(self).__name__}_{Layer.layercounter[type(self)]}'
        self.inputs = 0
        self.outputs = outputs
        self.name = name
        self.next = next

        def __repr__(self):
                text = f'Layer(inputs={self.inputs}, outputs=
{self.outputs}, name={repr(self.name)})'
                if self.next is not None:
                        text += ' + ' + repr(self.next)
                return text

    def add(self, next):
        if self.next is None:
            self.next = next
            next.set_inputs(self.outputs)
        else:
            self.next.add(next)

    def set_inputs(self, inputs):
        self.inputs = inputs
```

Hieronder wordt een neuraal netwerk gedefinieerd bestaande uit een aantal opeenvolgende lagen. Momenteel hebben de layers nog geen nuttige functionaliteit, maar de structuur van het netwerk kan wel getoond worden. Verifieer dat het aantal outputs van een voorgaande laag altijd automatisch gelijk is aan het aantal inputs van een volgende laag.

In [2]:
```
my_network = layers.Layer(num_outputs=3, name='Input')
my_network.add(layers.Layer(num_outputs=2, name='Hidden'))
my_network.add(layers.Layer(num_outputs=1, name='Output'))
print(my_network)
```

```
Layer(num_outputs=3, name='Input') +
       Layer(num_outputs=2, name='Hidden') +
       Layer(num_outputs=1, name='Output')
```

Vaak construeer je modellen van vele lagen. Om de syntax wat te vereenvoudigen voegen we de onderstaande `__add__()` methode toe die het mogelijk maakt om de `+` operator te gebruiken.

```
from copy import deepcopy

        def __add__(self, next):
                result = deepcopy(self)
                result.add(deepcopy(next))
                return result
```

Dit geeft een verkorte notatie om hetzelfde te bereiken als hierboven met de onderstaande compacte one-liner.

In [3]:
```
my_network = layers.Layer(3, name='Input') + layers.Layer(2, name='Hidden')
print(my_network)
```

```
Layer(num_outputs=3, name='Input') +
       Layer(num_outputs=2, name='Hidden') +
       Layer(num_outputs=1, name='Output')
```

Het is soms handig om toegang te hebben tot de verschillende lagen nadat het model eenmaal is gecreëerd. De onderstaande `__getitem__()` methode maakt het mogelijk om de opeenvolgende lagen te indexeren met een getalwaarde (gelijk aan het volgnummer van de laag, vergelijkbaar met hoe je een `list` indexeert) of een string (gelijk aan de naam van de laag, vergelijkbaar met hoe je een `dict` indexeert) met behulp van de gebruikelijke blokhaak-notatie.

```
def __getitem__(self, index):
                if index == 0 or index == self.name:
                        return self
                if isinstance(index, int):
                        if self.next is None:
                                raise IndexError('Layer index out
of range')
                        return self.next[index - 1]
                if isinstance(index, str):
                        if self.next is None:
                                raise KeyError(index)
                        return self.next[index]
                raise TypeError(f'Layer indices must be integers or
strings, not {type(index).__name__}')
```

Voeg deze methode aan de `Layer()` class toe, en ga na dat je begrijpt hoe deze werkt. Hieronder worden beide manieren van indexeren gedemonstreerd.

In [4]:
```
my_network = layers.Layer(3, name='Input') + layers.Layer(2, name='Hidden')
```

In [5]:
```
print(my_network['Output'])
```

```
Layer(num_outputs=1, name='Output')
```

In [6]:
```
print(my_network[2])
```

```
Layer(num_outputs=1, name='Output')
```

Opmerking:

Om deze parent class nog gebruiksvriendelijker te maken kun je ook andere dunder-methoden definiëren, hoewel dit niet strict noodzakelijk is om dit notebook te kunnen uitvoeren; in het bijzonder de `__iadd__()`, `__len__()` en `__iter__()` methoden liggen voor de hand.

Initialisatie

We hebben nu weliswaar een elegant raamwerk dat ons in staat stelt om neurale lagen aan elkaar te koppelen, maar op dit moment doen de lagen nog helemaal niets nuttigs. Daarom gaan we eerst diverse child-classes creëren waaraan we concrete functionaliteit kunnen toevoegen.

We beginnen met het afleiden van een invoerlaag waarmee de gebruiker exclusief interactie zal hebben. We zullen hier straks onder andere de inmiddels bekende `predict()` en `fit()` methoden aan toevoegen, maar voorlopig hoeft deze laag nog geen andere functionaliteit te bevatten dan diens parent-class `Layer()`. We passen alleen de `__repr__()` methode ietsjes aan.

```
class InputLayer(Layer):

        def __repr__(self):
                text = f'InputLayer(outputs={self.outputs}, name=
{repr(self.name)})'
                if self.next is not None:
                        text += ' + ' + repr(self.next)
                return text
```

Wanneer we nu een model opzetten met daarin de `InputLayer()` class als invoerlaag krijgen we netjes te zien dat het hier om een `InputLayer()` gaat; de `inputs` parameter is voor een invoerlaag niet relevant en wordt dan ook niet weergegeven.

In [7]:
```
my_network = layers.InputLayer(3, name='Input')
print(my_network)
```

```
InputLayer(num_outputs=3, name='Input')
```

De volgende stap is om de verborgen lagen te implementeren. Deze bestaan uit een parallelle serie van vele neuronen, elk met hun eigen gewichten. De laag ontvangt een aantal invoerwaarden zoals aangegeven in de instance-variable `inputs`, en in totaal heeft de laag een breedte gegeven door de instance-variable `outputs`.

Een neuron in een multi-layer perceptron voert twee operaties uit:

1. de inputs worden vermenigvuldigd met gewichten en samen met een bias opgeteld;
2. op de uitkomst hiervan wordt een activatiefunctie toegepast.

In tegenstelling tot de vorige les, waar één `Neuron()` class beide functionaliteiten bevatte, zullen we er hier voor kiezen om deze op te splitsen in twee aparte child-

classes van de `Layer()` class:

1. een class `DenseLayer()` die de gewogen lineaire combinatie uitvoert om de pre-activatiewaarden te berekenen;
2. een class `ActivationLayer()` die de activatiefunctie toepast om de post-activatiewaarden te berekenen.

De naam *dense layer* slaat op het feit dat we hier een *fully-* of *densely-connected* layer zullen definiëren waarin elke invoerwaarde met elk neuron wordt verbonden. Alle neuronen in een *activatielaag* krijgen dezelfde activatiefunctie.

Maak eerst de child-class `DenseLayer()` aan en begin weer met het overriden van de representatie-methode. Het zit echter wat ingewikkelder met de initialisatie. Omdat neurale lagen meerdere parallelle neuronen bevatten zullen ook de biases en gewichten meervoudig moeten worden uitgevoerd. De biases kunnen worden bijgehouden in een lijst met één index die overeenkomt met het nummer *o* van het uitvoerneuron in de laag; de gewichten vereisen een geneste lijst met twee indices die overeenkomen met het nummer *o* van het uitvoerneuron en het nummer i van de invoer naar het neuron. Echter, ten tijde van het instantiëren van een instance met `__init__()` is nog niet bekend hoeveel `inputs` i deze gaat hebben; dat gebeurt pas wanneer de laag aan een netwerk wordt toegevoegd middels de `set_inputs()` methode.

Maak daarom de instance-variabele `weights` weliswaar aan tijdens het instantiëren, maar vul deze pas met waarden in de `set_inputs()` methode zodra het aantal inputs bekend is. In tegenstelling tot de `Neuron()` class mogen de gewichten hier niet allemaal nul zijn. Deze worden geïnitialiseerd met een *uniforme random waarde* tussen $\pm\sqrt{\frac{6}{N_i+N_o}}$, met N_i en N_o gelijk aan het aantal inkomende en uitgaande verbindingen van een neuron (*Xavier-initialisatie*).

Denk zelf na waar je de `bias` kan initialiseren, en met welke waarden dit dan zou moeten.

> **Opmerking:**
>
> Wanneer een child-class de `__init__()` methode van de parent-class overridet, dan kun je middels `super().__init__()` de instantiatie-methode van de parent-class aanroepen om diens instance-variabelen te initialiseren.

Hieronder wordt eerst een `DenseLayer()` layer aangemaakt met $N_o = 2$ neuronen, en vervolgens toegevoegd aan de eerder gecreëerde `InputLayer()` die $N_i = 3$ invoerwaarden doorgeeft.

In [8]:
```
my_layer = layers.DenseLayer(2, name='Dense')
print(my_layer)
```

```
print(f'- biases = {my_layer.biases}')
print(f'- weights = {my_layer.weights}')
```

```
DenseLayer(num_outputs=2, name='Dense')
- biases = [0.0, 0.0]
- weights = None
```

In [9]:
```
my_network = layers.InputLayer(3, name='Input')
my_network.add(my_layer)
print(my_network)
print(f'- biases = {my_network[1].biases}')
print(f'- weights = {my_network[1].weights}')
```

```
InputLayer(num_outputs=3, name='Input') +
        DenseLayer(num_outputs=2, name='Dense')
- biases = [0.0, 0.0]
- weights = [[0.6074082594939483, 0.9361607746629916, -0.28546275290485024],
[0.8828198326953529, -0.2975070563386921, -0.43415599226345736]]
```

Maak als dit gelukt is ook de child-class `ActivationLayer()` aan en override de instantiatie- en representatie-methodes. Een instance-variabele `activation` bevat de activatiefunctie, met opnieuw als default de `linear()` functie. Deze dient geïnitialiseerd te worden in de `__init__()` methode en getoond te worden door de `__repr__()` methode.

Controleer hieronder dat je `ActivationLayer()` laag juist wordt aangemaakt, weergegeven, en aan het netwerk toegevoegd.

In [10]:
```
my_layer = layers.ActivationLayer(2, name='Activation')
print(my_layer)
```

```
ActivationLayer(num_outputs=2, name='Activation', activation='linear')
```

In [11]:
```
my_network = layers.InputLayer(3, name='Input') + \
             layers.DenseLayer(2, name='Dense')
my_network.add(my_layer)
print(my_network)
```

```
InputLayer(num_outputs=3, name='Input') +
        DenseLayer(num_outputs=2, name='Dense') +
        ActivationLayer(num_outputs=2, name='Activation', activation='linea
r')
```

De laatste child-class in dit hoofdstuk is de `LossLayer()`. Wij zullen deze als allerlaatste laag van een neuraal netwerk gebruiken om de loss te berekenen. Deze laag zal dus een instance-variabele moeten hebben die de door de gebruiker gewenste loss-functie bevat.

Omdat de `LossLayer()` class de laatste laag is, heeft deze nul `outputs` (net zoals de `InputLayer()` nul inputs heeft). De `outputs` en de volgende laag `next` hoeven dus ook niet te worden gespecificeerd bij de instantiatie en niet te hoeven weergegeven in de representatie. Je krijgt dan een instantiatie-methode met een signatuur als

`__init__(self, loss=mean_squared_error, name=None)`; pas zelf de representatie-methode aan.

Het voorgaande betekent ook dat er geen extra lagen mogen worden toegevoegd aan een instance van `LossLayer()`. Dit kun je afdwingen door de `add()` methode van de parent-class `Layer()` te overriden met een functie die slechts een `NotImplementedError()` genereert. En nu we toch bezig zijn, dit kun je ook doen voor de `set_inputs()` methode van de `InputLayer()`, aangezien de invoerlaag niet mag worden gekoppeld aan een voorgaande laag.

Hieronder maken we ook deze laatste laag aan, en voegen we deze aan het eerder opgebouwde netwerkje toe.

In [12]:
```
my_layer = layers.LossLayer(name='Loss')
print(my_layer)
```

```
LossLayer(num_inputs=None, name='Loss', loss='mean_squared_error')
```

In [13]:
```
my_network = layers.InputLayer(3, name='Input') + \
             layers.DenseLayer(2, name='Dense') + \
             layers.ActivationLayer(2, name='Activation')
my_network.add(my_layer)
print(my_network)
```

```
InputLayer(num_outputs=3, name='Input') +
        DenseLayer(num_outputs=2, name='Dense') +
        ActivationLayer(num_outputs=2, name='Activation', activation='linea
r') +
        LossLayer(num_inputs=2, name='Loss', loss='mean_squared_error')
```

Nu je dit allemaal voor elkaar hebt kun je hieronder op een simpele manier in één keer een neuraal netwerk aanmaken waarin alle soorten lagen voorkomen.

In [14]:
```
my_network = layers.InputLayer(3) + layers.DenseLayer(2) + layers.Activation
print(my_network)
```

```
InputLayer(num_outputs=3, name='InputLayer_5') +
        DenseLayer(num_outputs=2, name='DenseLayer_4') +
        ActivationLayer(num_outputs=2, name='ActivationLayer_3', activation
='linear') +
        LossLayer(num_inputs=2, name='LossLayer_2', loss='mean_squared_erro
r')
```

Gefeliciteerd!

Je kan nu op flexibele wijze neurale netwerken opzetten met een invoerlaag, zo veel verborgen lagen als je wil, en een uitvoerlaag die wordt gevolgd door de berekening van een loss.

Het *XOR*-probleem

Met deze kleine mijlpaal achter de rug is nu het moment aangebroken om een nieuwe dataset te introduceren met slechts een viertal instances. Je kunt deze opvragen met de functie `data.xorproblem()`.

In [15]:
```
xs, ys = data.xorproblem()
DataFrame(xs, columns=['x1', 'x2']).assign(y=DataFrame(ys))
```

Out[15]:

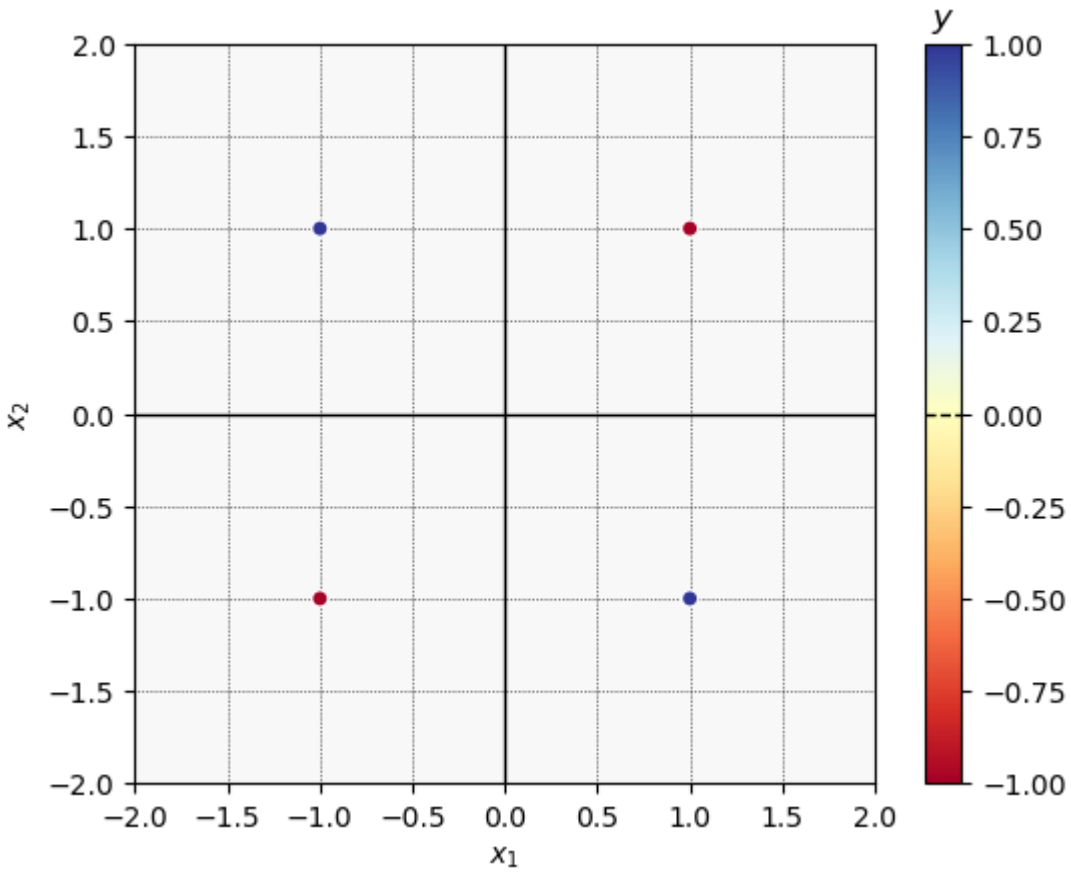
	x1	x2	y
0	-1.0	-1.0	-1.0
1	1.0	-1.0	1.0
2	-1.0	1.0	1.0
3	1.0	1.0	-1.0

De naam van de functie gaf het al weg: hopelijk herken je in deze dataset het *XOR-probleem*.

> **Opmerking:**
>
> Omdat neurale lagen meerdere outputs kunnen hebben bestaan de klasselabels in de variabele `ys` niet simpelweg meer uit een lijst getalwaarden, maar uit een geneste lijst. In dit geval is er maar één output, dus bevatten de geneste lijsten elk maar één element.

Ga in de figuur hieronder na dat deze data niet lineair separabel zijn.

In [16]:
```
data.scatter(xs, ys)
```



Nu volgt het lastige stuk waarin we ons neurale netwerk functioneel gaan maken. We zullen het neurale netwerk enerzijds moeten leren om in de *forward-propagation* fase uit gegeven attributen aan de invoerzijde een predictie af te leiden aan de uitvoerzijde, en om anderzijds in de *back-propagation* fase voor een bepaalde predictie aan de uitvoerzijde een loss te berekenen en met de gradiënten hiervan terugwerkend naar de invoerzijde de biases en gewichten bij te werken.

Forward-propagation

We zullen voor de forward-propagation gebruik maken van de `__call__()` methoden van de child-classes. Deze zorgen ervoor dat een object kan worden aangeroepen alsof het zelf een functie is. De `__call__()` methode kan voor de parent-class `Layer()` niet zinvol geïmplementeerd worden omdat elk type laag een andere operatie toepast op diens invoerwaarden. Omdat deze daarentegen wel door elke child-class gedefinieerd dient te worden, is dit een voorbeeld van een *abstracte* methode. We implementeren deze in de parent-class `Layer()` door slechts een foutmelding te genereren.

```
def __call__(self, xs):
        raise NotImplementedError('Abstract __call__ method')
```

Opmerking:

Python kent de `@abc.abstractmethod` decorator van de Abstract Base Classes module die eigenlijk geschikter is voor dit doel; we gebruiken die nu niet omdat

hierdoor geen `Layer()` objecten gedefinieerd zouden kunnen worden, en een aantal van de eerdere code-cellen hierboven daardoor niet meer uitvoerbaar zouden zijn.

Hieronder zie je de syntax waarmee een laag nu als een functie kan worden aangeroepen, zij het dat je een foutmelding zal krijgen omdat deze functie voor de `Layer()` class abstract is.

In [17]:
```
my_layer = layers.Layer(5)
try:
    ys = my_layer(xs)   # Een laag wordt aangeroepen als een functie!
except NotImplementedError:
    print('Alleen child-layers kunnen worden aangeroepen als een functie!')
```

```
Alleen child-layers kunnen worden aangeroepen als een functie!
```

We implementeren eerst de methoden die zorgen voor predictie. Deze ontvangen de instances $\boldsymbol{x}_n$ en retourneren de voorspellingen $\hat{\boldsymbol{y}}_n$. Dit kan recursief geïmplementeerd worden, waarbij elke laag de volgende laag aanroept totdat de recursie beëindigd wordt door de laatste laag.

- De `InputLayer()` ontvangt een (geneste) lijst instances `xs` van de gebruiker en kan deze onveranderd doorgeven aan de eerstvolgende verborgen laag door deze aan te roepen als `self.next(xs)`. De predictie die uiteindelijk door die volgende layer wordt geretourneerd kan weer rechtstreeks terug naar de gebruiker. De `__call__()` methode is hierbij slechts een soort "doorgeefluik". Voeg voor het gebruiksgemak ook een methode met de naam `predict()` toe die een eenvoudige wrapper is om de `__call__()` methode van de `InputLayer()` zelf.

```
def __call__(self, xs):
        return self.next(xs)

    def predict(self, xs):
        yhats = self(xs)
        return yhats
```

- De `DenseLayer()` krijgt invoer binnen van een vorige laag, berekent hieruit middels lineaire combinatie voor elke instance en elk neuron een pre-activatiewaarde $a_{no} = b_o + \sum_i w_{oi} \cdot x_{ni}$ (waarbij de index n loopt over de instances, o over de `outputs`, en i over de `inputs`), en geeft die uitkomsten door aan de volgende laag. De volgende laag gaat hier vervolgens mee verder rekenen en retourneert tenslotte voorspellingen die de `DenseLayer()` kan gebruiken als return value. Vul het onderstaande code-skelet aan.

```
def __call__(self, xs):
        aa = []   # Uitvoerwaarden voor alle instances xs
        for x in xs:
            a = []   # Uitvoerwaarde voor één instance x
```

```
            for o in range(self.outputs):
                # Bereken voor elk neuron o met de lijst
invoerwaarden x de uitvoerwaarde
                ...
                a.append(...)
            aa.append(a)
        yhats = self.next(...)
        return yhats
```

Opmerking:

Hierboven worden `for` - `append()` -loops gesuggereerd om de uitvoeren $\boldsymbol{a}_n$ van de laag te berekenen, maar deze opzet leent zich ook uitstekend voor list comprehensions.

- De `ActivationLayer()` maakt van elke pre-activatiewaarde een post-activatiewaarde door middel van de activatiefunctie $h_{no} = \varphi(a_{no})$, en geeft die door aan de volgende laag. Pas weer het code-skelet aan.

```
def __call__(self, xs):
        hh = []   # Uitvoerwaarden voor alle instances xs
        for x in xs:
            h = []   # Uitvoerwaarde voor één instance x
            for o in range(self.outputs):
                # Bereken voor elk neuron o met de lijst
invoerwaarden x de uitvoerwaarde
                ...
                h.append(...)
            hh.append(h)
        yhats = self.next(...)
        return yhats
```

- Een `LossLayer()` tenslotte krijgt diens invoer binnen uit de uitvoerlaag van het model, dus dat vormt reeds de voorspellingen $\hat{\boldsymbol{y}}_n$. De `LossLayer()` kan daardoor rechtstreeks de invoer retourneren als voorspelling. Je mag ervan uitgaan dat de laatste laag in het netwerk *altijd* een `LossLayer()` zal zijn. Vul de onderstaande code aan.

```
def __call__(self, xs):
        yhats = ...
        return yhats
```

Implementeer op deze manier de `__call__()` methoden voor alle child-classes. Je kan hierbij vermoedelijk onderdelen hergebruiken uit je uitwerking van het `Neuron()`.

De onderstaande code zet een neuraal netwerk op met meerdere lagen en stelt de biases en gewichten zo in dat de *XOR*-dataset correct gemodelleerd zou moeten worden.

In [18]:
```
my_network = layers.InputLayer(2) + \
             layers.DenseLayer(2) + \
             layers.ActivationLayer(2, activation=activation_functions.sign)
             layers.DenseLayer(1) + \
             layers.LossLayer()
my_network[1].biases = [1.0, -1.0]
my_network[1].weights = [[1.0, 1.0], [1.0, 1.0]]
my_network[3].biases = [-1.0]
my_network[3].weights = [[1.0, -1.0]]
```

Verifieer hieronder dat alle instances exact juist voorspeld worden.

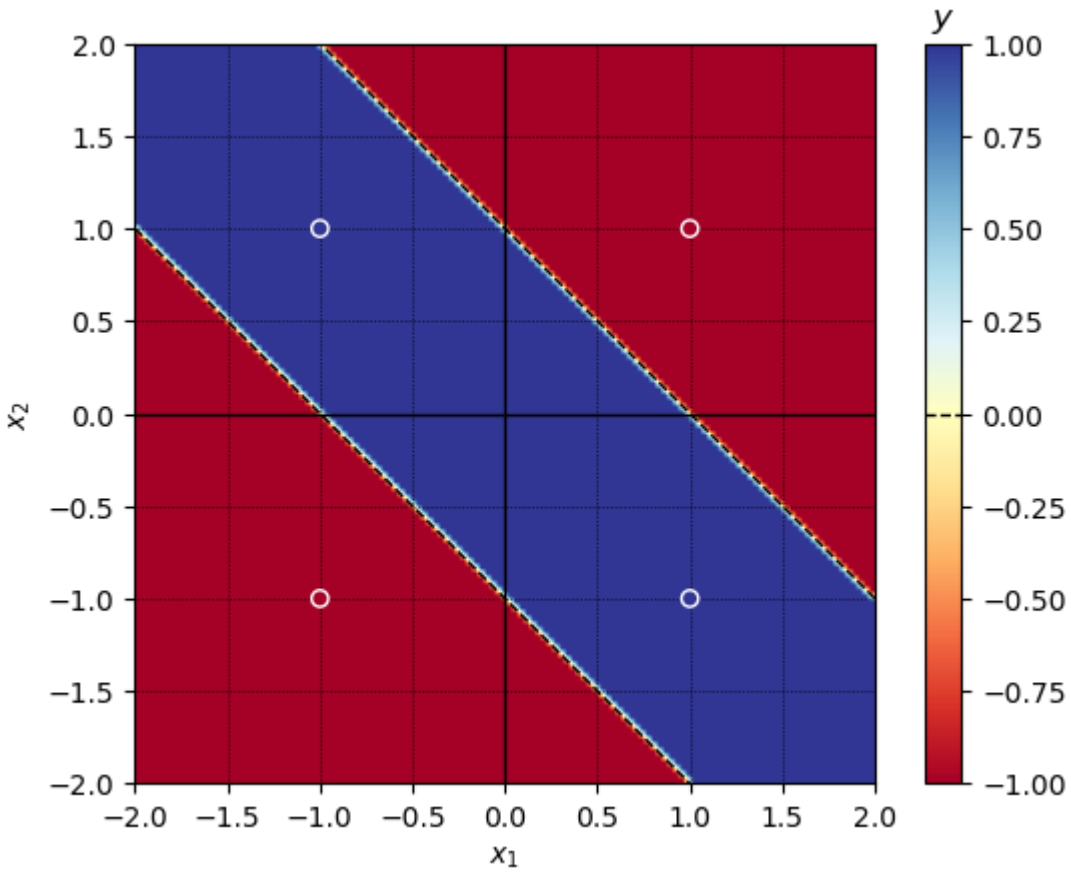
In [19]:
```
yhats = my_network.predict(xs)
DataFrame(xs, columns=['x1', 'x2']).assign(y=DataFrame(ys), ŷ=DataFrame(yhat
```

Out[19]:

	x1	x2	y	ŷ
0	-1.0	-1.0	-1.0	-1.0
1	1.0	-1.0	1.0	1.0
2	-1.0	1.0	1.0	1.0
3	1.0	1.0	-1.0	-1.0

Je ziet hieronder dat alle punten correct worden geclassificeerd. In dit geval loopt er een blauwe band diagonaal naar beneden; een andere correcte oplossing zou een rode band kunnen bevatten die diagonaal naar boven loopt. Probeer de bias en gewichten eens aan te passen om die andere oplossing te bereiken.

In [20]:
```
data.scatter(xs, ys, model=my_network)
print(my_network)
```




```
InputLayer(num_outputs=2, name='InputLayer_6') +
        DenseLayer(num_outputs=2, name='DenseLayer_5') +
        ActivationLayer(num_outputs=2, name='ActivationLayer_4', activation
='sign') +
        DenseLayer(num_outputs=1, name='DenseLayer_6') +
        LossLayer(num_inputs=1, name='LossLayer_3', loss='mean_squared_erro
r')
```

Losses

De `predict()` methode is nuttig om voorspellingen te genereren voor nieuwe testdata. Echter, voor bestaande trainingsdata weten we de gewenste uitkomsten al. Dan is het zinvol om ook de loss te kunnen berekenen om een idee te hebben hoe goed ons model is. Immers, hoe beter het voorspellingen, hoe lager de loss.

We zullen daartoe de bestaande `__call()__` methoden uitbreiden zodat ze niet alleen de predicties `yhats` maar ook de losses `ls` van alle instances retourneren. De losses bestaan uit een lijst met voor elke instance n één getalwaarde $l_n = \sum_o \mathcal{L}\left(\hat{y}_{no}; y_{no}\right)$, waarbij o loopt over alle outputs van de uitvoerlaag van het model. Deze zullen door de `LossLayer()` berekend moeten worden, aangezien deze de vorm van de loss-functie kent. Daarvoor moet de laag ook de correcte uitkomsten `ys` kennen.

Maak de `__call__()` methode van de `LossLayer()` zo dat je de correcte uitkomsten `ys` niet per se hoeft mee te geven als parameter:

- als de $\boldsymbol{y}_n$ *niet* worden meegegeven bereken je *alleen* de predicties $\hat{\boldsymbol{y}}_n$ voor alle instances;
- als de $\boldsymbol{y}_n$ *wel* worden meegegeven bereken je daarnaast *ook* de losses l_n voor alle instances.

Je krijgt zoiets als hieronder.

```
def __call__(self, xs, ys=None):
        yhats = ...
        ls = None
        if ys is not None:
            ls = ...
        return yhats, ls
```

De andere lagen hoeven het resultaat alleen maar door te geven van de volgende naar de vorige laag. Pas de andere child-classes aan zodat ze dat doen. Ook voor hen is het argument `ys` optioneel.

Tenslotte voegen we een methode `evaluate()` toe aan onze `InputLayer()` class die de *gemiddelde* loss over alle instances bepaalt. Daarvoor kunnen we weer volstaan met een simpele wrapper methode naar de gebruiker toe. In deze methode zijn de `ys` *niet* optioneel.

```
def evaluate(self, xs, ys):
        _, ls = self(xs, ys)
        l_mean = ...
        return l_mean
```

> **Opmerking:**
>
> De `predict()` methode zal ietwat aangepast moeten worden om om te gaan met het feit dat nu ook een resultaat voor de losses wordt geretourneerd, hoewel dat daar niet daadwerkelijk gebruikt wordt.

We testen de code hieronder. Als het goed is kun je nu zien dat in het voorgaande voorbeeld met de *XOR*-dataset inderdaad alle punten juist geclassificeerd werden: de loss is voor alle instances gelijk aan de laagst mogelijke waarde, dat wil zeggen nul.

In [21]:
```
l_mean = my_network.evaluate(xs, ys)
print(f'De gemiddelde loss is gelijk aan {l_mean:.3f}.')
```

```
De gemiddelde loss is gelijk aan 0.000.
```

Back-propagation

Hierboven hebben we zelf de biases en gewichten ingesteld om te zorgen dat het model de *XOR*-dataset kon beschrijven. Voor algemene datasets is dat meestal niet zo eenvoudig natuurlijk. Om op een systematischere manier algemene oplossingen te

vinden voor de modelparameters in een multi-layer perceptron zullen we opnieuw de optimalisatie-methode gebaseerd op *gradiënt descent* moeten toepassen.

De algemene vorm van de update-regel luidde $b \leftarrow b - \alpha \cdot \frac{\partial l}{\partial b}$ en $w \leftarrow w - \alpha \cdot \frac{\partial l}{\partial w}$. In dit geval hebben we het echter over de *gemiddelde* loss over alle N instances in `xs`: $l = \frac{1}{N} \sum_n l_n$. Dit leidt tot de volgende update-regel die *per instance* kan worden toegepast:

$$\begin{cases} b_o \leftarrow b_o - \frac{\alpha}{N} \cdot \frac{\partial l_n}{\partial b_o} \\ w_{oi} \leftarrow w_{oi} - \frac{\alpha}{N} \cdot \frac{\partial l_n}{\partial w_{oi}} \end{cases}$$

De indices lopen weer over de inputs (i), de outputs (o), en de instances (n). Om deze formule toe te kunnen passen dienen we voor de bias en gewichten van elk neuron in elke `DenseLayer()` te bepalen hoe de loss van een instance l_n verandert als die parameters b_o en w_{oi} gewijzigd worden.

We beginnen te kijken hoe de loss afhangt van de *input* van elke layer. Oftewel, als we de invoerwaarden `xs` van een laag een klein beetje zouden kunnen verhogen of verlagen, hoe verandert dan de loss van het model voor de huidige instance? In formulevorm, hoe groot is $\frac{\partial l_n}{\partial x_{ni}}$? We zullen hierbij terugwerken van de losslaag richting de invoerlaag. Bestudeer sectie *3.3. Back-propagation* van de Syllabus zodat je weet hoe dit proces in zijn werk gaat.

Voor de losslaag zelf kan de afgeleide van de loss naar diens invoer $\frac{\partial l_n}{\partial x_{ni}}$ numeriek worden bepaald door de functie `derivative()` toe te passen op de loss-functie. Immers, voor de losslaag is de invoer $\boldsymbol{x}_n$ gelijk aan de voorspelling $\hat{\boldsymbol{y}}_n$, waardoor $\frac{\partial l_n}{\partial x_{ni}} = \frac{\partial l_n}{\partial \hat{y}_{ni}} = \frac{\partial}{\partial \hat{y}_{ni}} \mathcal{L}(\hat{y}_{ni}; y_{ni}) = \mathcal{L}'(\hat{y}_{ni}; y_{ni})$.

We breiden de definitie van de `__call__()` methode van de `LossLayer()` nog een (laatste) maal uit zodat deze naast de predicties en losses óók de gradiënten van de loss $\nabla_{\boldsymbol{x}} l_n$ als retourwaarde `gs` geeft. Omdat we de gradiënten alleen nodig hebben als we het model trainen, en we hierbij een learning rate α zullen moeten specificeren, spreken we af dat de gradiënten alleen berekend hoeven te worden als een parameter `alpha` is meegegeven aan de functie. De code komt er nu ongeveer als volgt uit te zien:

```
def __call__(self, xs, ys=None, alpha=None):
        yhats = ...
        ls = None
        gs = None
        if ys is not None:
            ls = ...
            if alpha is not None:
                gs = ...
        return yhats, ls, gs
```

Voor de `ActivationLayer()` en de `DenseLayer()` geldt hierna dat ze berekende gradiënten binnenkrijgen van de volgende laag. In tegenstelling tot de losses `ls` kunnen de gradiënten `gs` helaas niet gewoon worden doorgegeven.

- Als de `ActivationLayer()` een gradiënt binnenkrijgt van de volgende laag gelijk aan $\boldsymbol{q}$, dan is dit de gradiënt van de loss naar de invoer van de volgende laag, oftewel naar de uitvoer van de huidige laag. De laag krijgt dus een lijst met waarden $q_{ni} = \frac{\partial l_n}{\partial h_{ni}}$ binnen (waarbij $\boldsymbol{h}$ de uitvoerwaarden van de activatie-laag zijn). De activatie-laag dient hieruit de gradiënt van de loss naar diens inputs $g_{ni} = \frac{\partial l_n}{\partial x_{ni}}$ te berekenen. Hiervoor kan worden gesteld dat $g_{ni} = \frac{\partial h_{ni}}{\partial x_{ni}} \cdot q_{ni}$. Omdat $h_{ni} = \varphi(x_{ni})$, is de afgeleide $\frac{\partial h_{ni}}{\partial x_{ni}}$ precies gelijk aan de helling van de activatiefunctie ter plekke van de invoer x_{ni}. We vinden de formule

$$g_{ni} = \varphi'(x_{ni}) \cdot q_{ni}$$

- Ook de `DenseLayer()` krijgt een lijst met gradiënten $q_{no} = \frac{\partial l_n}{\partial a_{no}}$ binnen van de volgende laag en dient hieruit de gradiënt van de loss naar de inputs $g_{ni} = \frac{\partial l_n}{\partial x_{ni}}$ te berekenen. In een fully-connected laag geldt $g_{ni} = \sum_o \frac{\partial a_{no}}{\partial x_{ni}} \cdot q_{no}$. Omdat $a_{no} = b_o + \sum_i w_{oi} \cdot x_{ni}$, is de afgeleide $\frac{\partial a_{no}}{\partial x_{ni}}$ precies gelijk aan het gewicht w_{oi}. Dit leidt tot

$$g_{ni} = \sum_o w_{oi} \cdot q_{no}$$

We kennen nu de gradiënten van de loss naar de in- en uitvoerwaarden van alle neurale lagen. De laatste stap is hieruit de gradiënten van de loss naar de instelbare netwerkparameters af te leiden. Deze zijn nodig om in de `DenseLayer()` de biases en gewichten bij te werken. Voor deze laag geldt dat er een pre-activatiewaarde wordt berekend volgens de formule $a_{no} = b_o + \sum_i w_{oi} \cdot x_{ni}$:

- De gradiënt naar een bias b_o kan worden geschreven als $\frac{\partial l_n}{\partial b_o} = \frac{\partial l_n}{\partial a_{no}} \cdot \frac{\partial a_{no}}{\partial b_o}$. Hierin is $\frac{\partial l_n}{\partial a_{no}}$ inmiddels berekend (zie hierboven) en is $\frac{\partial a_{no}}{\partial b_o} = 1$.
- De gradiënt naar een gewicht w_{oi} kan worden geschreven als $\frac{\partial l_n}{\partial w_{oi}} = \frac{\partial l_n}{\partial a_{no}} \cdot \frac{\partial a_{no}}{\partial w_{oi}}$. Hierin is $\frac{\partial l_n}{\partial a_{no}}$ weer bekend, en de afgeleide $\frac{\partial a_{no}}{\partial w_{oi}} = x_{ni}$, net als in het vorige hoofdstuk voor het single-layer perceptron.

Kortom, zodra de gradiënten via back-propagation zijn doorgerekend, kunnen de afgeleiden van de loss naar de bias en gewichten hiermee ook worden bepaald. Hierop wordt tenslotte stochastic gradient descent toegepast. Alleen de `DenseLayer()` dient hierbij de update-regel toe te passen om diens parameters bij te werken. Zoals eerder gezegd luidt de update-regel hiervoor:

$$\begin{cases} b_o \leftarrow b_o - \frac{\alpha}{N} \cdot \frac{\partial l_n}{\partial b_o} \\ w_{oi} \leftarrow w_{oi} - \frac{\alpha}{N} \cdot \frac{\partial l_n}{\partial w_{oi}} \end{cases}$$

Ook hier voegen we weer een eenvoudige wrapper methode toe aan de `InputLayer()` class. Dit is precies de `partial_fit()` functie die we ook kennen uit vorige hoofdstukken. Specificeer zelf weer een geschikte default waarde voor de learning rate α.

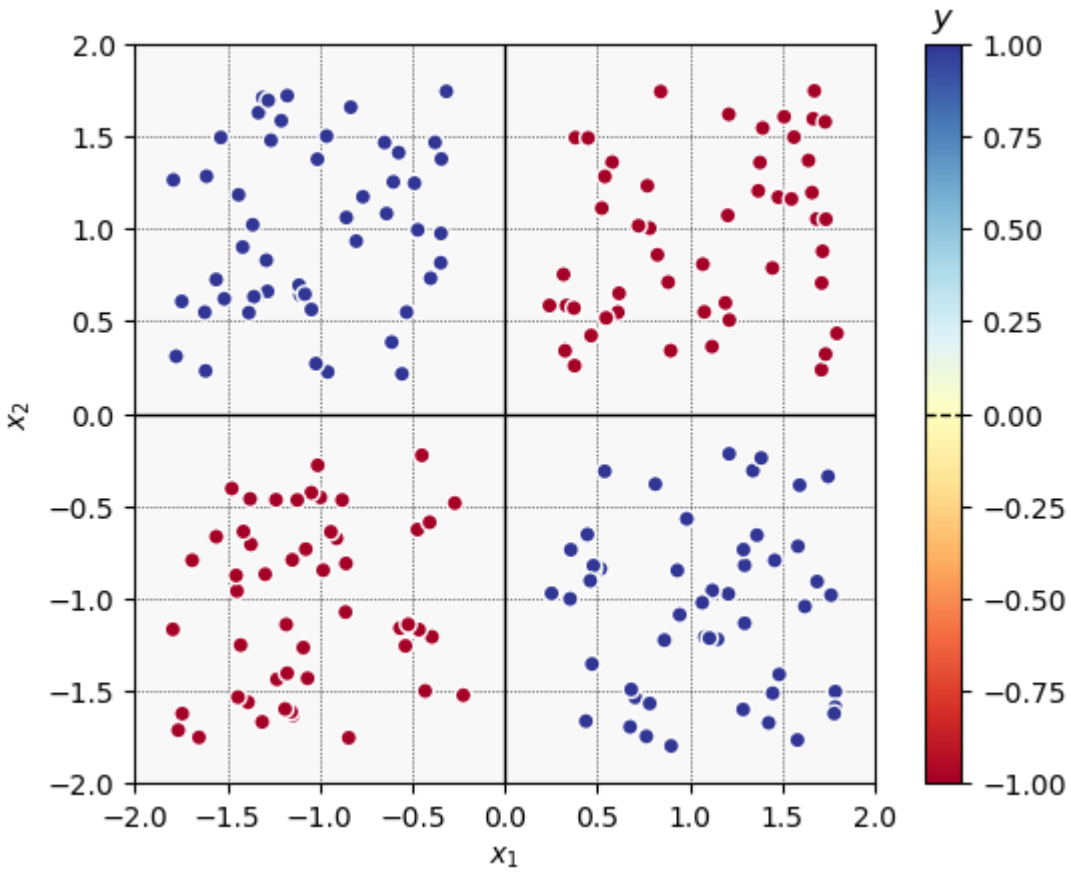
```
def partial_fit(self, xs, ys, alpha=...):
        self(xs, ys, alpha)
```

Voeg tenslotte ook de `fit()` functie weer toe die een aantal epochs traint; deze code zul je waarschijnlijk identiek uit de vorige les kunnen overnemen.

> **Opmerking:**
>
> De `predict()` en `evaluate()` methoden zullen weer ietwat aangepast moeten worden omdat nu de `__call__()` methoden drie soorten gegevens retourneren.

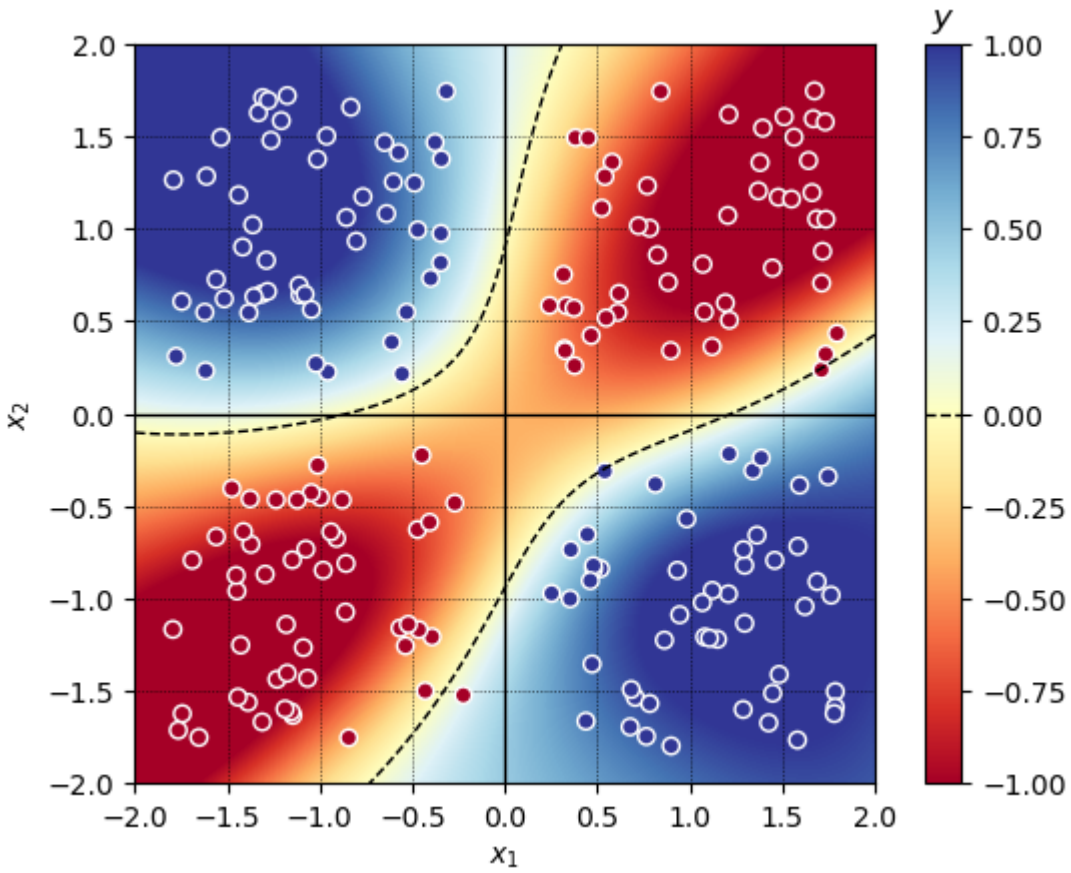
Laten we dit multi-layer perceptron model eens gebruiken om het *XOR*-probleem te fitten. Eerst definiëren we een wat uitgebreidere dataset waaraan enige ruis is toegevoegd.

In [22]:
```
xs, ys = data.xorproblem(num=200, noise=0.8)
data.scatter(xs, ys)
```



Dit keer kunnen we geen signum-functie gebruiken als activatie-functie omdat deze geen geschikte afgeleide heeft. Daarom passen we het model een beetje aan en kiezen we voor de afgevlakte versie in de vorm van de $\tanh$-functie, zoals we ook bij logistische regressie deden. We stoppen ook wat meer parallelle neuronen in de hidden layer om het model krachtiger te maken zodat het een grotere kans heeft om te convergeren naar een bruikbare oplossing.

In [23]:
```
my_network = layers.InputLayer(2, name='Input')
my_network.add(layers.DenseLayer(5, name='Dense'))
my_network.add(layers.ActivationLayer(5, activation=activation_functions.tar
my_network.add(layers.DenseLayer(1, name='Output'))
my_network.add(layers.LossLayer(name='Loss'))
my_network.fit(xs, ys, alpha=0.1, epochs=200)
data.scatter(xs, ys, model=my_network)
print(my_network)
print(f'- Loss: {my_network.evaluate(xs, ys)}')
```




```
InputLayer(num_outputs=2, name='Input') +
        DenseLayer(num_outputs=5, name='Dense') +
        ActivationLayer(num_outputs=5, name='Activation', activation='tanh')
+
        DenseLayer(num_outputs=1, name='Output') +
        LossLayer(num_inputs=1, name='Loss', loss='mean_squared_error')
- Loss: 0.11808285612776113
```

Zijn de modellen die je krijgt als je de fit meerdere keren opnieuw berekent vergelijkbaar van vorm?

We kijken ook naar de voorspellingen voor de eerste handvol instances. Merk op dat $\hat{y}$ nu niet altijd tussen -1 en $+1$ in ligt, zoals bij logistische regressie het geval was. Het model dat we nu hebben opgezet behandelt dit *XOR*-probleem als (niet-lineaire) regressie, en voorspelt dus getallen in plaats van (kansen op) labels.

In [24]:
```
yhats = my_network.predict(xs)
DataFrame(xs, columns=['x1', 'x2']).assign(y=DataFrame(ys), ŷ=DataFrame(yhat
```

Out[24]:

	x1	x2	y	ŷ
0	-0.566758	-1.159854	-1.0	-0.693572
1	0.898811	-1.796891	1.0	0.884377
2	-1.289477	0.828213	1.0	1.118394
3	0.579848	1.358183	-1.0	-0.590251
4	-1.312083	-1.667234	-1.0	-1.066420

Gefeliciteerd!

Je hebt nu een krachtig neuraal netwerk geïmplementeerd dat zelf willekeurige niet-lineaire problemen kan leren oplossen.

Ter vergelijking passen we hieronder een neuraal netwerk op deze dataset toe uit de deep-learning module keras van Google's tensorflow bibliotheek. Omdat dit model de data in de vorm van een numpy array verwacht (vergelijkbaar met een matrix array in de programmeertalen *R* of *MATLAB*) converteren we eerst de data naar dit formaat.

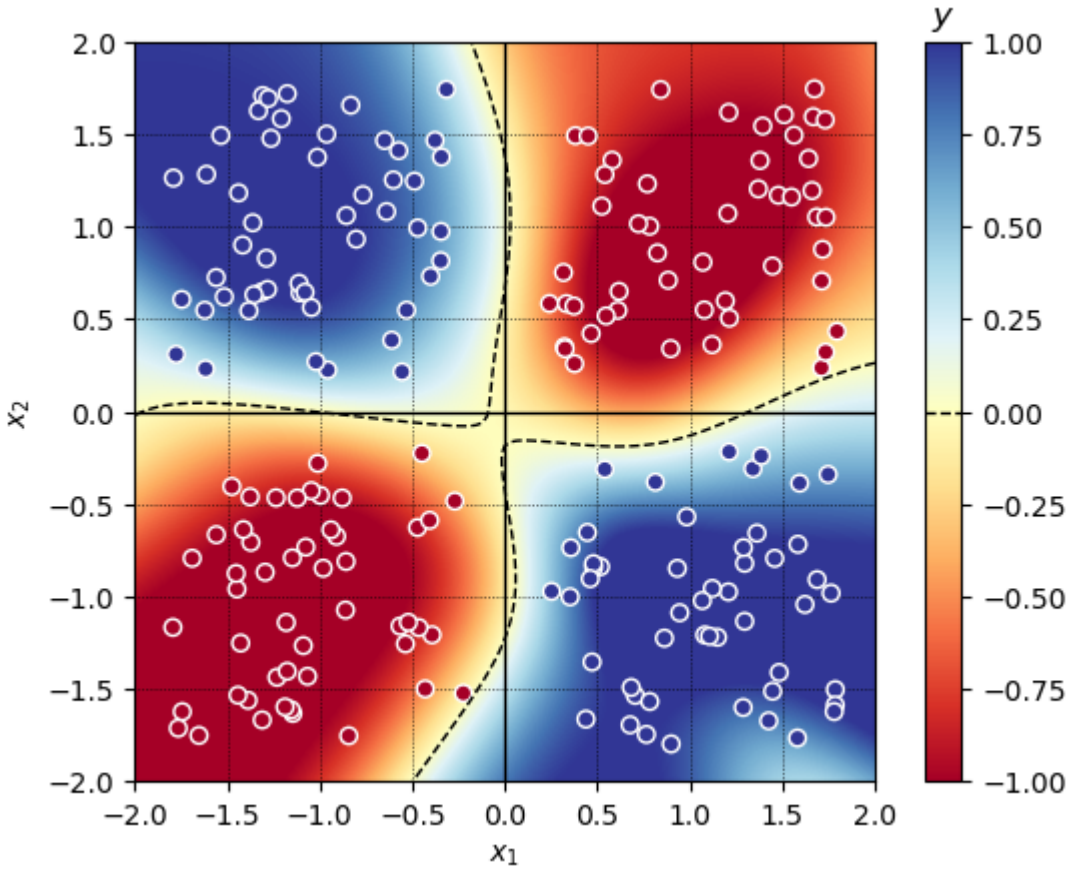
In [25]:
```
krs_xs, krs_ys = array(xs), array(ys)
```

Voel je vrij de eigenschappen van `krs_xs` en `krs_ys` nader te onderzoeken. Deze gedragen zich soortgelijk als geneste Python lijsten, maar hebben wat meer wiskundige functionaliteit.

De syntax van `tensorflow.keras` zoals die hieronder volgt is zeer vergelijkbaar met die van je eigen model. Een verschil is dat het type model eerst expliciet als een *sequentieel* model dient te worden gedefinieerd. Daarnaast wordt de loss functie niet in een aparte uitvoerlaag toegevoegd maar middels een speciale compilatie-stap, en daarbij moet tevens de optimalisatie-methode en diens learning rate worden ingesteld (hier: *stochastic gradient descent*, `SGD()`). De weergave van het model in tekstvorm ziet er ook ietsjes anders uit. Ondanks die kleine verschillen zou de onderstaande code inmiddels prima te begrijpen moeten zijn.

In [26]:
```
krs_network = keras.models.Sequential()
krs_network.add(keras.layers.InputLayer(input_shape=(2, ), name='Input'))
krs_network.add(keras.layers.Dense(5, name='Dense'))
krs_network.add(keras.layers.Activation(activation=keras.activations.tanh, n
krs_network.add(keras.layers.Dense(1, name='Output'))
krs_network.compile(loss=keras.losses.MeanSquaredError(), optimizer=keras.op
krs_network.fit(krs_xs, krs_ys, verbose=0, epochs=200)
data.scatter(krs_xs, krs_ys, model=krs_network)
krs_network.summary()
print(f'- Loss: {krs_network.evaluate(xs, ys, verbose=0)}')
```

```
WARNING:absl:At this time, the v2.11+ optimizer `tf.keras.optimizers.SGD` ru
ns slowly on M1/M2 Macs, please use the legacy Keras optimizer instead, loca
ted at `tf.keras.optimizers.legacy.SGD`.
521/521 [==============================] - 0s 305us/step
```




```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 Dense (Dense)               (None, 5)                 15

 Activation (Activation)     (None, 5)                 0

 Output (Dense)              (None, 1)                 6

=================================================================
Total params: 21 (84.00 Byte)
Trainable params: 21 (84.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
- Loss: 0.07304263114929199
```

De opzet van het neurale netwerk is identiek gekozen aan je eerdere eigen model: de diepte en breedte van het model zijn hetzelfde (dat wil zeggen, het aantal lagen en het aantal neuronen per laag), en dezelfde activatie- en loss-functies zijn gekozen. Ziet de oplossing er daardoor ongeveer hetzelfde uit als voor je eigen model, en is de kwaliteit van de voorspellingen zoals gemeten met de gemiddelde loss vergelijkbaar? Waar zouden eventuele verschillen door kunnen komen, denk je?

Opmerking:

Een soortgelijke dataset als deze met bijbehorend model is ook online te vinden om interactief mee te experimenteren.
