Eindopdracht Advanced Datamining

Studiejaar 2023-2024, 1e gelegenheid

Inleiding

Dit is de *eindopdracht* behorende bij het vak *Advanced Datamining* (BFVH4DMN2) voor het *studiejaar 2023-2024 (1e gelegenheid)*. 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, en een bijbehorend data-bestand `Sign_MNIST_mini.zip`.

Gebruik de `model` module die je in werkcollegeopdrachten 1, 2, 3, 4, en 5 & 6 hebt gemaakt om de onderstaande opdrachten uit te voeren. Deze eindopdracht bestaat uit twee delen:

- in **Deel A** worden een aantal cellen code gedraaid die als het goed is onmiddellijk zouden moeten werken met je model;
- in **Deel B** wordt je gevraagd om je gemaakte model zelf toe te passen, en hoef je je module slechts licht uit te breiden.

> **Waarschuwing:**
>
> De code in je module mag gebruik maken van alle functies uit de Python Standard Library (zoals `math`, `random`, `itertools`, enzovoorts); het is *niet* toegestaan om functies toe te passen uit overige modules (zoals `sklearn`, `keras`, `tensorflow`, enzovoorts).

Eerst zetten we wat initialisatie op en importeren we naast de `data` en `model` modules enkele onderdelen van `pandas`, `numpy`, en `time`. 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]:
```
%matplotlib inline
%reload_ext autoreload
%autoreload 2
```

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

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

from time import perf_counter

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

```
Using pandas version 2.2.1
Using numpy version 1.26.4
Using vlearning version 0.6.1
```

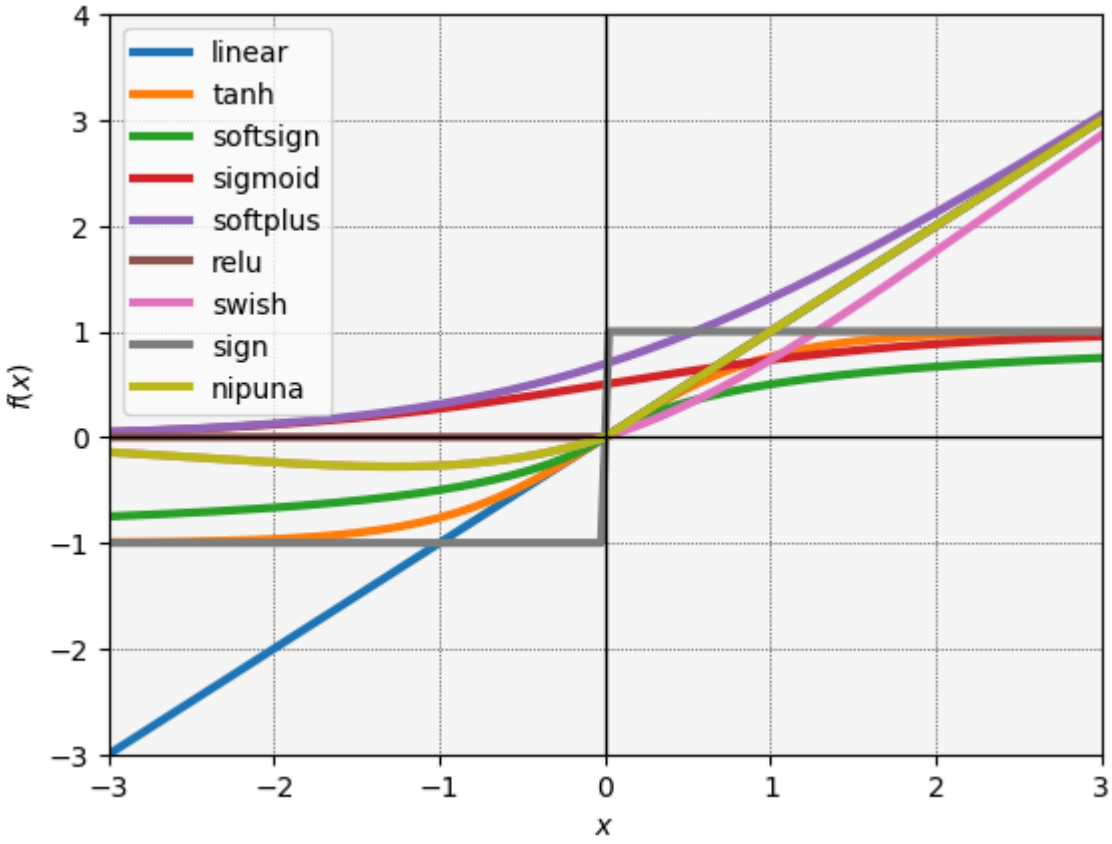
Deel A

Hieronder staan een aantal fragmenten code die je model *ongewijzigd* dient te kunnen uitvoeren. Voor verdere details omtrent deze gevraagde functionaliteiten, zie zonodig de werkcollege-opdrachten en/of de syllabus.

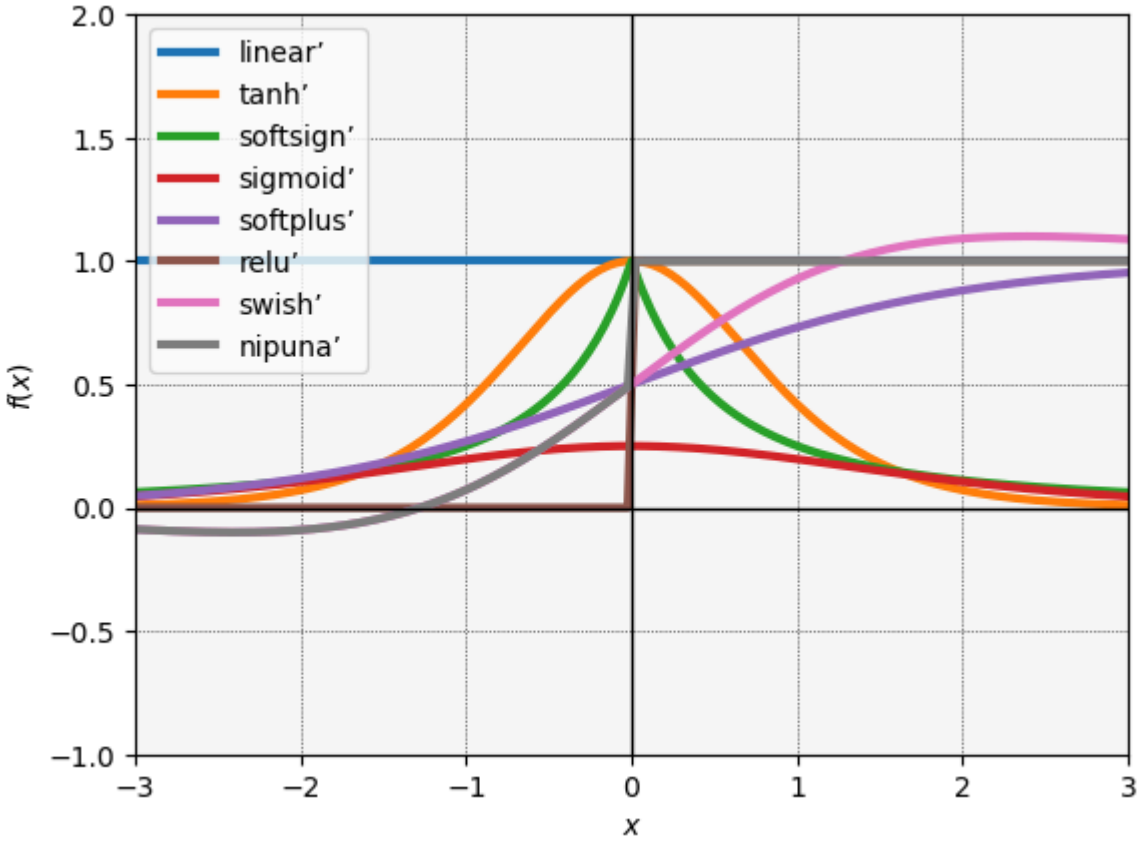
Activatiefuncties

In [2]:
```
my_activations = [
    activation_functions.linear,
    activation_functions.tanh,
    activation_functions.softsign,
    activation_functions.sigmoid,
    activation_functions.softplus,
    activation_functions.relu,
    activation_functions.swish,
    activation_functions.sign,
    activation_functions.nipuna,
]
my_arguments =  [-1000, -1, 0, 1, 1000]
my_table = [[φ(a) for a in my_arguments] for φ in my_activations]
my_columns = [f'φ({a})' for a in my_arguments]
my_rows = [φ.__name__ for φ in my_activations]
```

In [3]:
```
data.graph(my_activations)
```



In [4]:
```
data.graph([vlearning.derivative(φ) for φ in my_activations if φ != activati
```



In [5]:
```
DataFrame(my_table, columns=my_columns).set_index(array(my_rows))
```

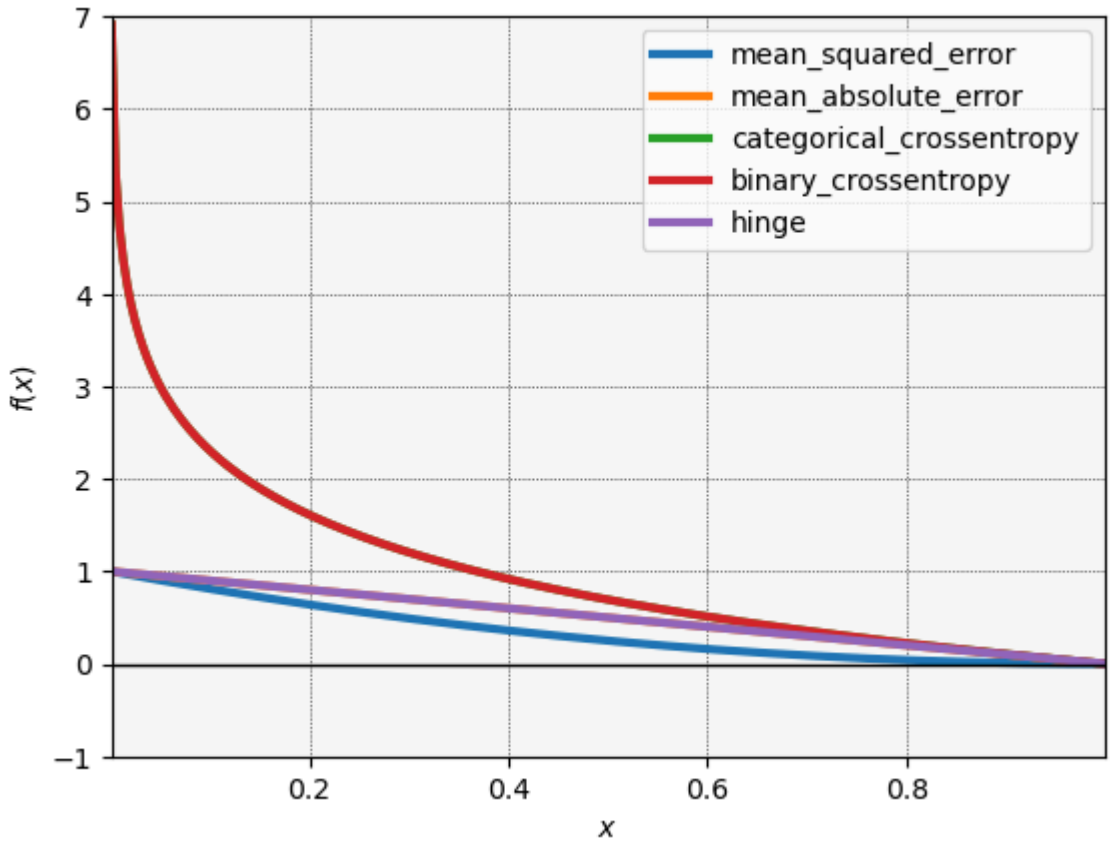
Out[5]:

	ϕ(-1000)	ϕ(-1)	ϕ(0)	ϕ(1)	ϕ(1000)
linear	-1000.000000	-1.000000	0.000000	1.000000	1000.000000
tanh	-1.000000	-0.761594	0.000000	0.761594	1.000000
softsign	-0.999001	-0.500000	0.000000	0.500000	0.999001
sigmoid	0.000000	0.268941	0.500000	0.731059	1.000000
softplus	0.000000	0.313262	0.693147	1.313262	1000.000000
relu	0.000000	0.000000	0.000000	1.000000	1000.000000
swish	-0.000000	-0.268941	0.000000	0.731059	1000.000000
sign	-1.000000	-1.000000	0.000000	1.000000	1.000000
nipuna	-0.000000	-0.268941	0.000000	1.000000	1000.000000

Lossfuncties

In [6]:
```
my_losses = [
    loss_functions.mean_squared_error,
    loss_functions.mean_absolute_error,
    loss_functions.categorical_crossentropy,
    loss_functions.binary_crossentropy,
    loss_functions.hinge
]
my_arguments =  [0.01, 0.1, 0.5, 0.9, 0.99]
my_table = [[L(a, 1.0) for a in my_arguments] for L in my_losses]
my_columns = [f'L({a}; 1)' for a in my_arguments]
my_rows = [L.__name__ for L in my_losses]
```

In [7]:
```
data.graph(my_losses, 1.0, xlim=(0.001, 0.999))
```



In [8]:
```
DataFrame(my_table, columns=my_columns).set_index(array(my_rows))
```

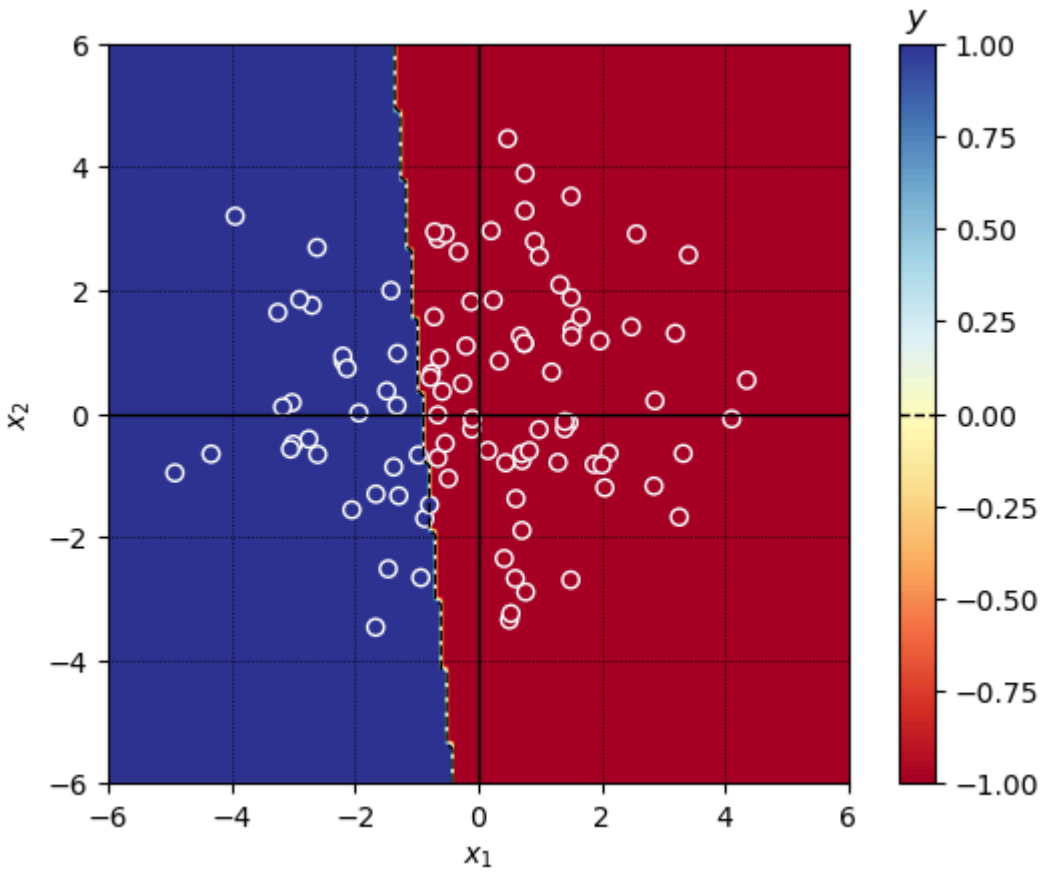
Out[8]:

	L(0.01; 1)	L(0.1; 1)	L(0.5; 1)	L(0.9; 1)	L(0.99; 1)
mean_squared_error	0.98010	0.810000	0.250000	0.010000	0.00010
mean_absolute_error	0.99000	0.900000	0.500000	0.100000	0.01000
categorical_crossentropy	4.60517	2.302585	0.693147	0.105361	0.01005
binary_crossentropy	4.60517	2.302585	0.693147	0.105361	0.01005
hinge	0.99000	0.900000	0.500000	0.100000	0.01000

Classificatie: single-layer perceptron

In [9]:
```
xs, ys = data.linear('nominal')
my_model = vlearning.Perceptron(dim=2)
my_model.fit(xs, ys)
data.scatter(xs, ys, model=my_model)
print(my_model)
```

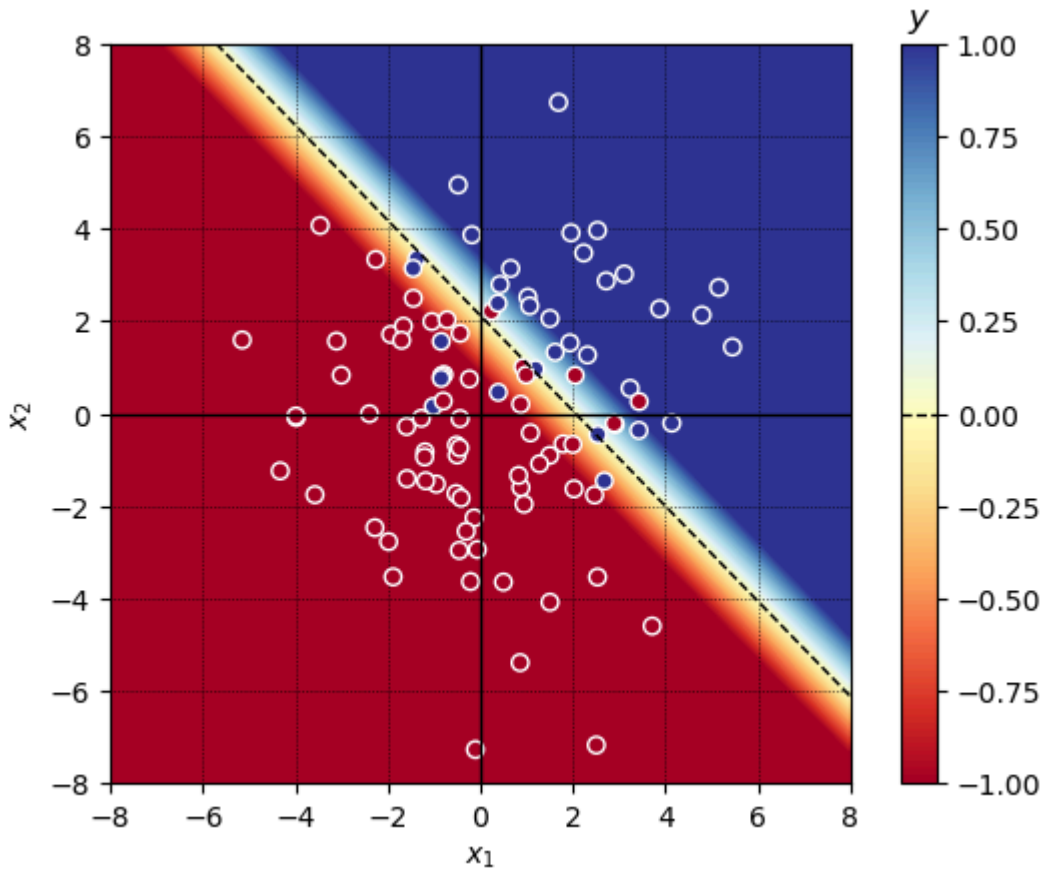
```
Model has been fully fitted after 4 epochs
```




```
Perceptron(dim=2)
```

Classificatie: support vector machine

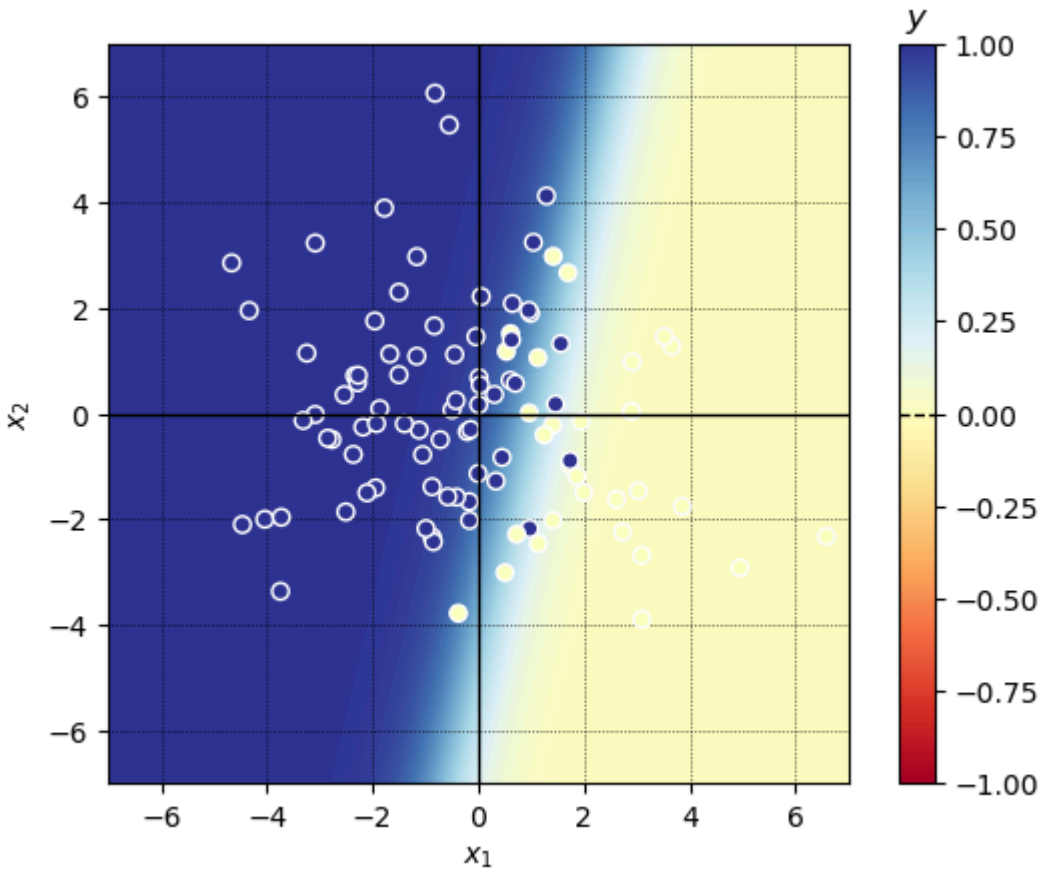
In [10]:
```
xs, ys = data.linear(outcome='nominal', noise=1.0)
my_model = vlearning.Neuron(dim=2, loss=loss_functions.hinge)
my_model.fit(xs, ys)
data.scatter(xs, ys, model=my_model)
print(my_model)
```




```
Neuron(dim=2, activation=linear, loss=hinge)
```

Classificatie: binomiale logistische regressie

In [11]:
```
xs, ys = data.linear(outcome='nominal', noise=1.0)
ys = [(y + 1.0) / 2.0 for y in ys]   # Convert labels -1/+1 to 0/1
my_model = vlearning.Neuron(dim=2, activation=activation_functions.sigmoid,
my_model.fit(xs, ys)
data.scatter(xs, ys, model=my_model)
print(my_model)
```

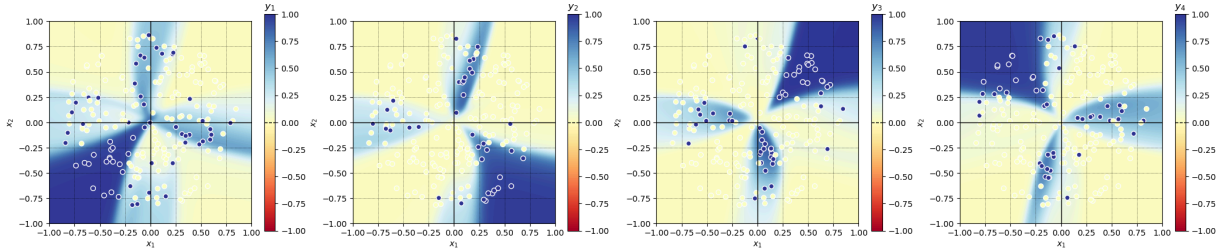



```
Neuron(dim=2, activation=sigmoid, loss=binary_crossentropy)
```

Classificatie: multi-layer perceptron

In [12]:
```
xs, ys = data.fractal(4)
my_model = (
    layers.InputLayer(2) +
    layers.DenseLayer(20) + layers.ActivationLayer(20, activation=activation
    layers.DenseLayer(10) + layers.ActivationLayer(10, activation=activation
    layers.DenseLayer(4) + layers.SoftmaxLayer(4) +
    layers.LossLayer(loss=loss_functions.categorical_crossentropy)
)
my_model.fit(xs, ys, alpha=0.5, epochs=100, batch_size=20); my_model.fit(xs,
data.scatter(xs, ys, model=my_model)
print(my_model)
```

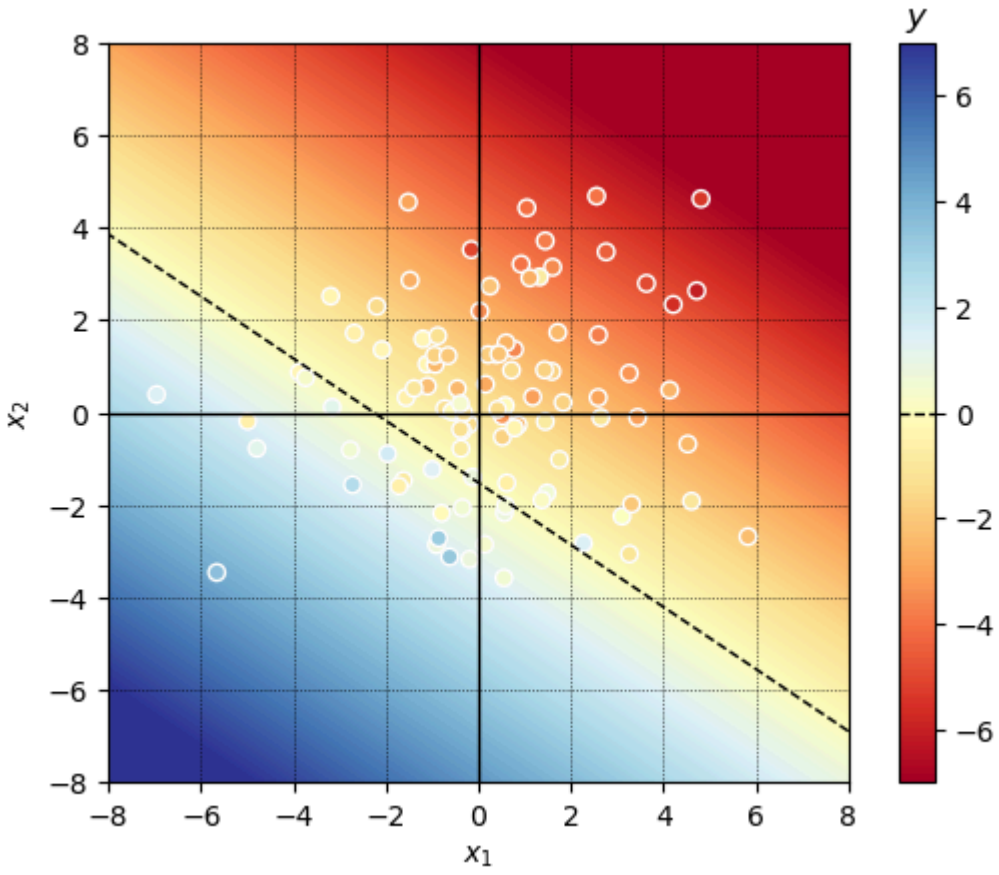
```
Epochs trained: 100%|████████████████████████████████████████████████
████████████████| 100/100 [00:01<00:00, 68.16epoch/s]
Epochs trained: 100%|████████████████████████████████████████████████
████████████████| 10/10 [00:00<00:00, 67.51epoch/s]
```



```
InputLayer(num_outputs=2, name='InputLayer_1') +
        DenseLayer(num_outputs=20, name='DenseLayer_1') +
        ActivationLayer(num_outputs=20, name='ActivationLayer_1', activation
='tanh') +
        DenseLayer(num_outputs=10, name='DenseLayer_2') +
        ActivationLayer(num_outputs=10, name='ActivationLayer_2', activation
='tanh') +
        DenseLayer(num_outputs=4, name='DenseLayer_3') +
        SoftmaxLayer(num_outputs=4, name='SoftmaxLayer_1') +
        LossLayer(num_inputs=4, name='LossLayer_1', loss='categorical_crosse
ntropy')
```

Regressie: lineaire regressie

In [13]:
```
xs, ys = data.linear('numeric', noise=1.0)
my_model = vlearning.LinearRegression(dim=2)
my_model.fit(xs, ys)
data.scatter(xs, ys, model=my_model)
print(my_model)
```



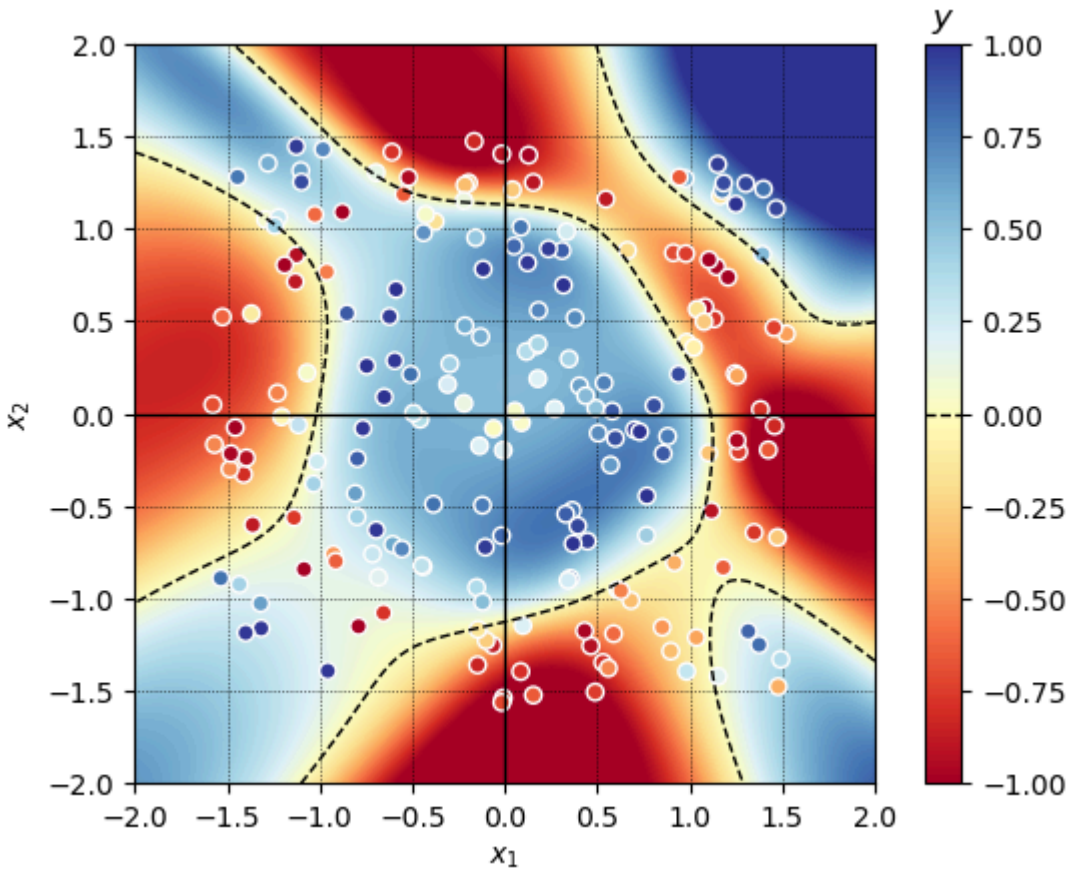

```
LinearRegression(dim=2)
```

Regressie: neuraal netwerk

In [14]:
```
xs, ys = data.concentric(noise=0.1)
my_model = (
    layers.InputLayer(2) +
    layers.DenseLayer(20) + layers.ActivationLayer(20, activation=activation
    layers.DenseLayer(10) + layers.ActivationLayer(10, activation=activation
```

```
    layers.DenseLayer(1) + layers.ActivationLayer(1, activation=activation_f
    layers.LossLayer(loss=loss_functions.mean_squared_error)
)
my_model.fit(xs, ys, alpha=0.05, epochs=200, batch_size=10)
my_model.fit(xs, ys, alpha=0.05, epochs=20)
data.scatter(xs, ys, model=my_model)
print(my_model)
```

```
Epochs trained: 100%|██████████████████████████████████████████████████
██████████████| 200/200 [00:02<00:00, 75.42epoch/s]
Epochs trained: 100%|██████████████████████████████████████████████████
██████████████| 20/20 [00:00<00:00, 77.53epoch/s]
```




```
InputLayer(num_outputs=2, name='InputLayer_2') +
        DenseLayer(num_outputs=20, name='DenseLayer_4') +
        ActivationLayer(num_outputs=20, name='ActivationLayer_3', activation
='tanh') +
        DenseLayer(num_outputs=10, name='DenseLayer_5') +
        ActivationLayer(num_outputs=10, name='ActivationLayer_4', activation
='tanh') +
        DenseLayer(num_outputs=1, name='DenseLayer_6') +
        ActivationLayer(num_outputs=1, name='ActivationLayer_5', activation
='linear') +
        LossLayer(num_inputs=1, name='LossLayer_2', loss='mean_squared_erro
r')
```

Deel B

In dit deel ga je aan de slag op een variant van een klassieke dataset die bestaat uit duizenden afbeeldingen van handgeschreven cijfers (de MNIST database). De huidige variant bevat gedigitaliseerde foto's van handgebaren uit het Amerikaanse gebarentaal spellingsalfabet. De oorspronkelijke dataset is te bekijken online. Beschikbaar op BlackBoard is een bestand **Sign_MNIST_mini.dat** (dat je dient te unzippen uit **Sign_MNIST_mini.zip**) met een geminiaturiseerde zwart-wit versie van afbeeldingen van slechts 12x12 pixels elk. In totaal zijn er maximaal 33.600 instances beschikbaar, 1.400 van elk van 24 letters (de letters J en Z zijn weggelaten omdat deze gebaren beweging vereisen). De functie `data.sign_mnist_mini()` kan gebruikt worden om een aantal instances op te vragen. Deze functie levert de attributen van de instances in de vorm van 144 pixel-intensiteiten tussen 0 en 1, en de klasselabels in de vorm van 24 getalwaarden met het juiste cijfer als een one-hot encoding.

In [15]:
```
help(data.sign_mnist_mini)
```

```
Help on function sign_mnist_mini in module vlearning.data:

sign_mnist_mini(filename, num=33600, seed=None)
    Returns a number of different random 12x12 Sign-MNIST images.

    Keyword arguments:
    filename -- full filename of the *.dat datafile
    num      -- number of images to randomly select (default 33600)
    seed     -- a seed to initialise the random number generator (default ra
ndom)

    Return values:
    xs       -- 144-element lists of pixel values (range 0.0-1.0)
    ys       -- 24-element lists of correct gestures using one-hot encoding
```

Hiervan genereren we aanvankelijk om het simpel te houden slechts tweehonderdveertig instances elk voor de trainings-, validatie- en testdata (dat wil zeggen, telkens tien per gebaar). Onderzoek zelf de organisatie van deze data nader.

In [16]:
```
# STAP 1: DATAGENERATIE
xs, ys = data.sign_mnist_mini('./Sign_MNIST_mini.dat', num=720)
trn_xs, trn_ys = xs[  0:240], ys[  0:240]
val_xs, val_ys = xs[240:480], ys[240:480]
tst_xs, tst_ys = xs[480:720], ys[480:720]
```

Hieronder wordt een eenvoudig dummy model aangemaakt; er is weliswaar vanalles aan te merken op dit overgesimplificeerde model, maar hanteer dit als een eerste uitgangspunt.

In [17]:
```
# STAP 2: MODELDEFINITIE
my_model = (
    layers.InputLayer(144, name='input') +
    layers.DenseLayer(24, name='hidden') + layers.ActivationLayer(24, name='
```

```
    layers.LossLayer(name='loss')
)
```

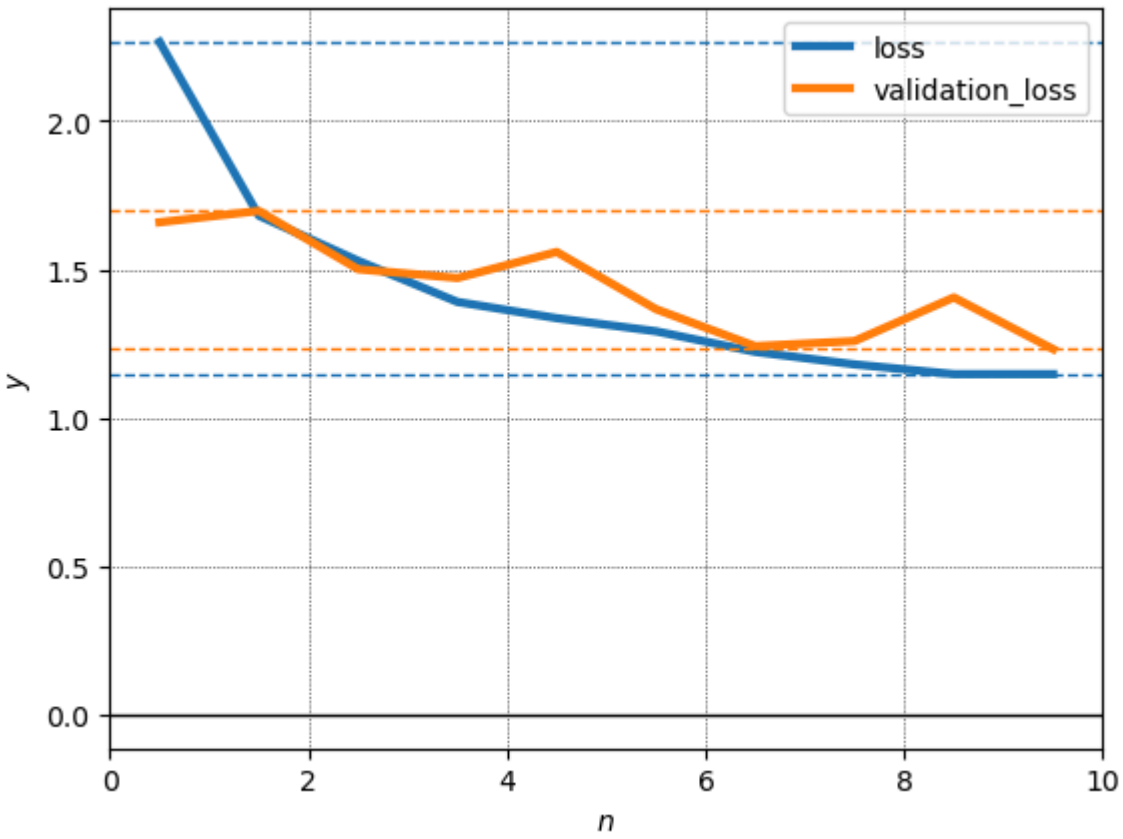
Vervolgens trainen en evalueren we dit model als volgt. Wederom zijn de gekozen parameters ongetwijfeld niet optimaal.

In [18]:
```
# STAP 3: TRAINING
my_history = my_model.fit(trn_xs, trn_ys, alpha=0.005, epochs=10, batch_size
```

```
Epochs trained: 100%|████████████████████████████████████████████████████
████████████████| 10/10 [00:01<00:00,  7.08epoch/s]
```

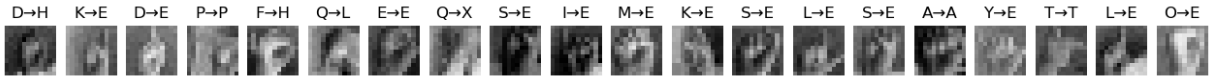
Echter, hiermee kunnen we een validatiecurve construeren.

In [19]:
```
# STAP 4: VALIDATIECURVE
data.curve(my_history)
```



Om inzicht te krijgen in de prestaties van het model, worden hieronder twintig instances uit de testdata getoond met voor en na de pijl respectievelijk de juiste en de voorspelde klasselabels. De resolutie van de afbeeldingen is nogal belabberd, dus het is wellicht niet zo vreemd dat de resultaten te wensen overlaten.

In [20]:
```
# STAP 5: VISUALISATIE
data.gestures(tst_xs[:20], tst_ys[:20], model=my_model)
```



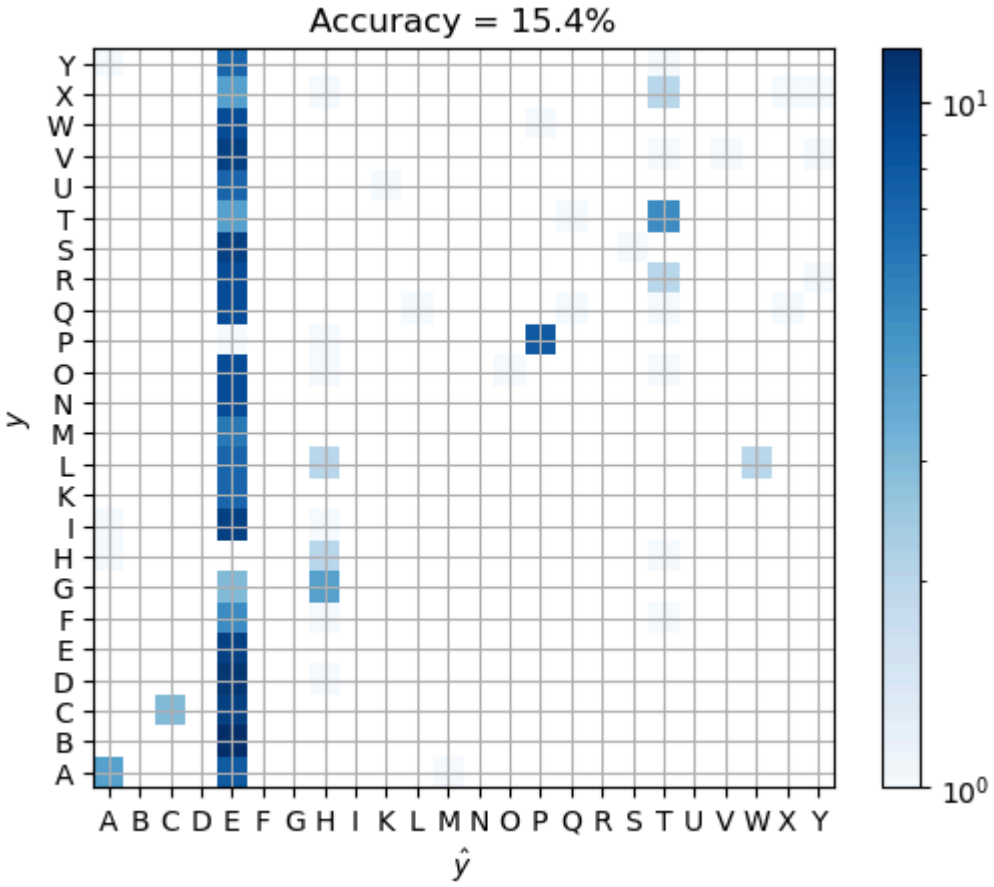
Berekenen we eens de gemiddelde loss op alle testdata.

In [21]:
```
# STAP 6: EVALUATIE
print(f'Loss: {my_model.evaluate(tst_xs, tst_ys)}')
```

```
Loss: 1.2023253800334754
```

Dit getal zegt misschien nog niet zoveel. Daarom bekijken we een grafische weergave van de confusion matrix die weergeeft welke voorspelde klasselabels op de x-as aan alle echte klasselabels op de y-as worden toegekend (let op de logaritmische kleurschaal).

In [22]:
```
# STAP 7: CONFUSIONMATRIX
data.confusion(tst_xs, tst_ys, model=my_model)
```



Hoewel er best wat fouten worden gemaakt en er waarschijnlijk een paar lettergebaren zijn die bij voorkeur worden gekozen, liggen er toch wel wat juist geclassificeerde instances op de diagonaal. Kortom, een aantal van de gebaren worden al ietwat redelijk herkend. Dit overdreven simpele model bereikt - afhankelijk van de willekeurig gekozen instances en initialisatiewaarden - een nauwkeurigheid van 10 à 15%, wat meer is dan de $\frac{1}{24} \approx 4\%$ die je op grond van toeval zou verwachten. Dit is nog niet indrukwekkend goed, maar gezien de eenvoudige opbouw van het model en de matige kwaliteit van de data al enigszins verrassend.

Pas nu hieronder eens het bovenstaande model aan tot een neuraal netwerk dat deze afbeeldingen redelijk betrouwbaar kan classificeren. Kies zelf een geschikte opzet van het model en bepaal door te experimenteren geschikte waarden voor de diverse parameters. Voer dezelfde zeven stappen uit als hierboven, maar dan met een effectiever en beter geoptimaliseerd model.

Opdracht:

Gebruik tenminste ergens in je model de *NIPUNA* activatiefunctie (met parameter $\beta = 1$). Zoek hier zonodig informatie over op en implementeer deze in je module.

In [23]:
```
# Verander deze cel niet
starttime = perf_counter()
```

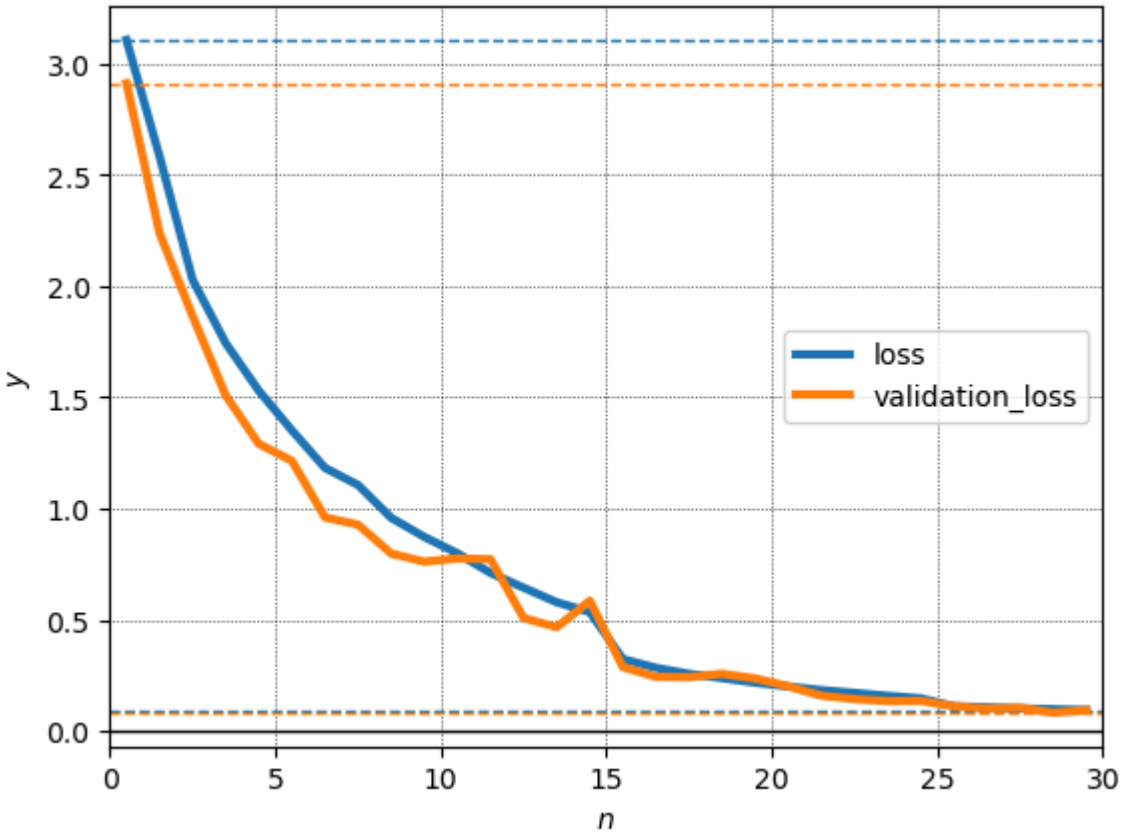
In [24]:
```
# STAP 1: DATAGENERATIE
# amount_of_instances = 33600
amount_of_instances = 24000
# amount_of_instances = 16800
# amount_of_instances = 7200
xs, ys = data.sign_mnist_mini('./Sign_MNIST_mini.dat', num=amount_of_instanc

trn_size = int(0.90 * amount_of_instances)
val_size = int(0.05 * amount_of_instances)
tst_size = int(0.05 * amount_of_instances)

trn_xs, trn_ys = xs[:trn_size], ys[:trn_size]
val_xs, val_ys = xs[trn_size:trn_size+val_size], ys[trn_size:trn_size+val_si
tst_xs, tst_ys = xs[trn_size+val_size:], ys[trn_size+val_size:]
```

In [25]:
```
# STAP 2: MODELDEFINITIE
my_model = (
    layers.InputLayer(144) +
    layers.DenseLayer(120) + layers.ActivationLayer(120, activation=activati
    layers.DenseLayer(60) + layers.ActivationLayer(60, activation=activation
    layers.DenseLayer(30) + layers.ActivationLayer(30, activation=activation
    layers.DenseLayer(24) + layers.SoftmaxLayer(24) +
    layers.LossLayer(loss=loss_functions.categorical_crossentropy)
)
```

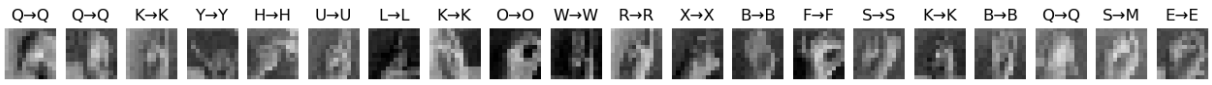
In [26]:
```
# STAP 3: TRAINING
my_history = my_model.fit(trn_xs, trn_ys, alpha=0.05, epochs=15, batch_size=
my_history = my_model.fit(trn_xs, trn_ys, alpha=0.01, epochs=10, batch_size=
my_history = my_model.fit(trn_xs, trn_ys, alpha=0.0025, epochs=5, batch_size
```

```
Epochs trained: 100%|████████████████████████████████████████████████
████████████████| 15/15 [19:31<00:00, 78.09s/epoch]
Epochs trained: 100%|████████████████████████████████████████████████
████████████████| 10/10 [13:04<00:00, 78.47s/epoch]
Epochs trained: 100%|████████████████████████████████████████████████
████████████████| 5/5 [06:33<00:00, 78.76s/epoch]
```

In [27]:
```
# STAP 4: VALIDATIECURVE
data.curve(my_history)
```



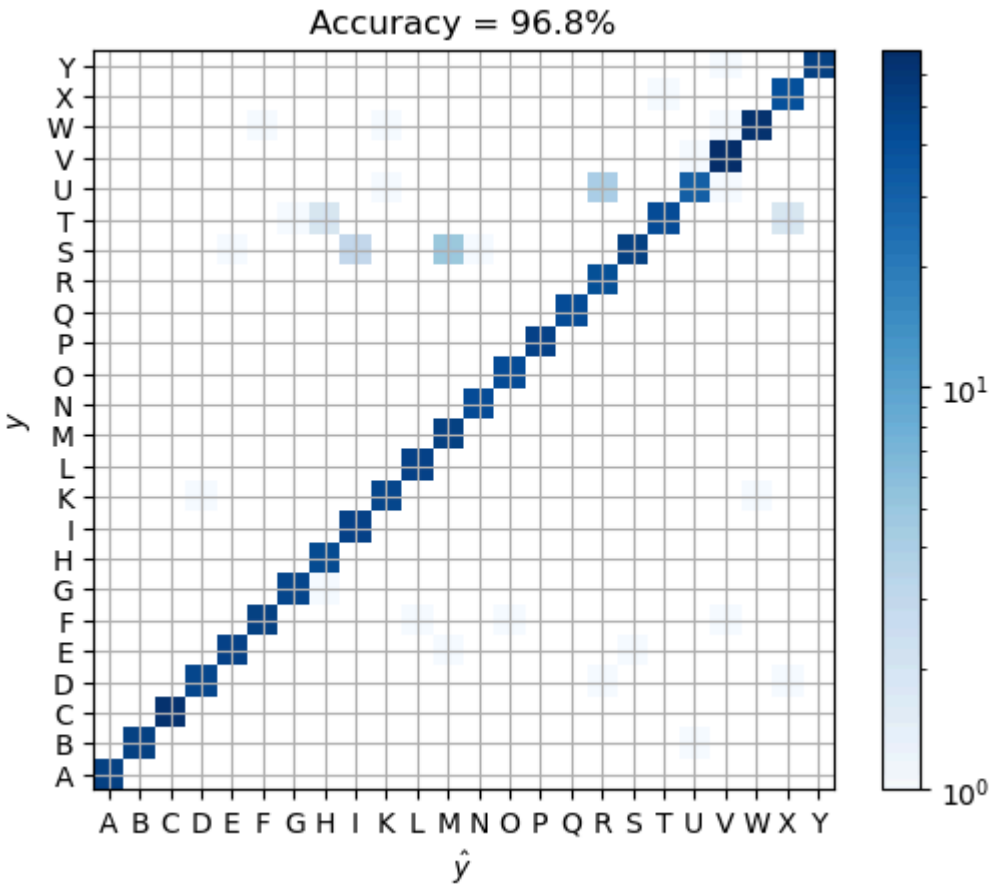
In [28]:
```
# STAP 5: VISUALISATIE
data.gestures(tst_xs[:20], tst_ys[:20], model=my_model)
```



In [29]:
```
# STAP 6: EVALUATIE
print(f'Loss: {my_model.evaluate(tst_xs, tst_ys)}')
```

```
Loss: 0.11683317347395782
```

In [30]:
```
# STAP 7: CONFUSIONMATRIX
data.confusion(tst_xs, tst_ys, model=my_model)
```



In [31]:
```
# Verander deze cel niet
print(f'Verstreken tijd: {(perf_counter() - starttime) / 60.0:.1f} minuten.'
```

```
Verstreken tijd: 39.2 minuten.
```

Hint:

Voer achtereenvolgens de onderstaande ontwikkelstappen uit.