

# LLM usage report

Derek Kentie

D.B.S.Kentie@student.tue.nl

## Instructions: how to use this template

- Fill out your name and email.
- Remove/replace everything that is written in blue.
- Follow the maximum number of words given for each section.
- Changing font, spacing, or document layout is not allowed.
- Screenshots of your prompts and the LLM answers have to be uploaded in the Appendix of this document.

## For what type of tasks did you used large language models (LLMs)? Explain how it helped you with these tasks (max 300 words)

I used ChatGPT to help me work with object oriented programming (like making a separate class to generate the architecture of the model), integrating the needed functions for backwards propagation (e.g. one-hot encoding, cross entropy and the different differentiated activation functions) and debugging (for example with extracting and implementing the data the right way). It easily helped me see how connections between different functions were not working due to minor errors in the code, and it helped me work out how the math behind backwards propagations can be coded.

## How did you implement the LLMs outputs in your solutions for this course? (max 200 words)

I tried avoiding the use of LLMs as much as possible, but quickly found out that the model I choose the program was way out of my league. I directly asked ChatGPT to write certain functions, after which I tried simplifying it by making more smaller functions instead of a few large ones, by describing variables in a more understandable way, and by commenting on around (parts of) functions that weren't clear. I made tried to see if the functions were valid enough, but I found this very difficult to do by looking at it with the naked eye. By testing every function I could validate them and implement them into my model.

## How did the usage of LLMs impact your learning experience? (max 300 words)

Using the LLM allowed me to understand object oriented programming quite well, however I do think that it made it to easy for me to surpass moments were critical thinking were really needed (like thinking about how the math could have been implemented).

## Did you ever get unexpected/incorrect/incomplete outputs from the LLM? If so how did you handle these outputs? (max 200 words)

Sometimes the LLM made improvements on (parts of) my model, where it forgot to take in account multiple necessities which I discussed with it beforehand. It had trouble remembering every step that itself applied before and every requirement I gave it. At one point I had to give up on the progress that I had with the model, due to continuous errors that it applied (I asked it to fully rewrite my code and implement the desired extra code with it, which it then messed up). So at that point I went back to my main model (because I was

working in a branch) and reworked it from there while rewriting the code step by step with the help of the LLM.

# Appendix

## ⭐ Beste datastructuur: één lijst van layer sizes

Je maakt een lijst zoals:

```
python  
layer_sizes = [input_size, 3, 2, output_size]
```

Code kopiëren

Vanaf hier kun je:

### ✓ ALLE gewichten automatisch genereren

```
python  
  
weights = []  
biases = []  
for i in range(len(layer_sizes)-1):  
    W = np.random.randn(layer_sizes[i], layer_sizes[i+1]) * 0.1  
    b = np.zeros((1, layer_sizes[i+1]))  
    weights.append(W)  
    biases.append(b)
```

Code kopiëren

### ✓ Forward pass generiek maken (for-loop)

### ✓ Backprop generiek maken (reverse-loop)

En je hoeft nergens meer een layer hard-coded te programmeren.

## 💡 Waar sla je deze architectuur het beste op?

+ Stel een vraag

0

...

ik wil een makkelijk aanpasbare functie maken waarmee ik de architectuur van mijn model kan beschrijven. Hoe ik het nu aanpak is door een functie te maken waarin ik de `n_features` van `X.shape` oproep, de `output_size` koppel aan de `len(outputs)` voor `outputs`, `counts = np.unique(y, return_counts=True)`, en de `hidden_size` creëer door het te beschrijven als een tuple (bijvoorbeeld (3,2) geeft h1 met 3 nodes en h2 met 2 nodes). Vervolgens wil ik dit allemaal in een bepaald data type opslaan (ik weet nog niet welke het beste zal zijn) om hieraan in één keer overall weights aan te koppelen. Mijn doel is namelijk om te vermijden dat ik voor elke laag individueel dit moet beschrijven, en zodat ik ook in één keer de backpropagation kan laten werken op elke weight. Mijn vraag aan jouw is of dit goed zal gaan werken, en wat de voor en nadelen zijn van het werken met een functie of subclass voor architecture.

Hier is het *korte, maar complete* antwoord dat je zoekt — precies afgestemd op jouw doel:

- een functie (of class) waarmee je in één keer de volledige architectuur van een MLP beschrijft, zodat je automatisch alle gewichten kunt genereren én backprop over alle lagen kunt laten lopen zonder handmatig elke laag te programmeren.

## ✅ TL;DR — Ja, jouw aanpak gaat werken

En het is precies hoe je een schaalbare MLP hoort te bouwen zonder PyTorch.

Maar er is een **betere manier** qua datastructuur.

## 💡 De cruciale designvraag

## De cruciale designvraag

Je vraagt in feite:

"Moet ik de architectuur beschrijven via  
1) een functie die tuples/parameters retourneert  
of  
2) een class die alles organiseert?"

Beide kunnen — maar er zijn duidelijke verschillen in **voordelen en nadelen** (zie verderop).

### ▀ Jij beschrijft nu ongeveer dit:

- `input_size` → `X.shape[1]`
- `hidden_sizes` → tuple zoals `(3, 2)`
- `output_size` → `len(np.unique(y))`
- architectuur = samengestelde lijst/tuple van alle layer sizes
- daarna generieke loops voor forward & backward

Dit is **exact** wat je moet willen bereiken:

- ➡ Één datastructuur waarin alle laaggroottes staan
- ➡ Automatisch alle gewichten genereren op basis van deze structuur
- ➡ Automatische loops voor forward/backward

### ★ Beste datastructuur: één lijst van layer sizes

Je maakt een lijst zoals:

```
python Code kopiëren
layer_sizes = [input_size, 3, 2, output_size] ▼
```

Vanaf hier kun je:

✓ ALLE gewichten automatisch genereren

```
python  
  
weights = []  
biases = []  
for i in range(len(layer_sizes)-1):  
    W = np.random.randn(layer_sizes[i], layer_sizes[i+1]) * 0.1  
    b = np.zeros((1, layer_sizes[i+1]))  
    weights.append(W)  
    biases.append(b)
```

Code kopiëren

✓ Forward pass generiek maken (for-loop)

✓ Backprop generiek maken (reverse-loop)

En je hoeft nergens meer een layer hard-coded te programmeren.

## 💡 Waar sla je deze architectuur het beste op?

Hier zijn je opties:

◆ Optie A: een simpele dictionary (kort & functioneel)

```
python  
  
architecture = {  
    "input": n_features,  
    "hidden": (3, 2),  
    "output": n_classes  
}  
↓
```

Code kopiëren

Voordeel:

- ✓ makkelijk te printen
- ✓ makkelijk aan te passen

Nadeel:

- ✗ moet nog steeds telkens omgezet worden naar 1 lijst van layer\_sizes

◆ Optie B: een functie die layer\_sizes teruggeeft (clean en flexibel)

```
python  
  
def build_architecture(n_features, hidden_layers, n_classes):  
    return [n_features] + list(hidden_layers) + [n_classes]
```

Code kopiëren

Gebruik:

```
python  
  
layer_sizes = build_architecture(X.shape[1], (3,2), len(np.unique(y)))
```

Code kopiëren

Voordeel:

- ✓ super simpel
- ✓ geen class nodig
- ✓ ideaal voor kleine MLP's

Nadeel:

- ✗ minder geschikt als je later extra configuratie wil toevoegen (dropout, activatie per laag, etc.)

#### ◆ Optie C: een Architecture class (beste keuze voor schaalbaarheid)

```
python Code kopiëren
class Architecture:
    def __init__(self, input_size, hidden_sizes, output_size):
        self.input = input_size
        self.hidden = hidden_sizes
        self.output = output_size

    @property
    def layer_sizes(self):
        return [self.input] + list(self.hidden) + [self.output]
```

Gebruik:

```
python Code kopiëren
arch = Architecture(X.shape[1], (3,2), len(np.unique(y)))
print(arch.layer_sizes)
```

Voordeel:

- ✓ je kunt later *per laag activatiefuncties toevoegen*
- ✓ je kunt regularization per laag toevoegen
- ✓ code blijft extreem overzichtelijk
- ✓ makkelijk unit-testen
- ✓ makkelijk door te geven aan het MLP-model

Nadeel:

- ✗ iets meer code
- ✗ misschien iets "zwaarder" dan een dict of functie



### ⌚ Wat raad ik jou aan?

Omdat jij:

- een volledig eigen MLP moet bouwen,
  - de architectuur makkelijk wil aanpassen,
  - later waarschijnlijk extra opties wil toevoegen (batchnorm? activatiefuncties? dropout?),
  - backprop automatisch wil laten loopen,
- 👉 gebruik een clean Architecture class.

Het ziet er dan zo uit:

```
python Code kopiëren
class Architecture:
    def __init__(self, X, y, hidden_layers):
        self.input = X.shape[1]
        self.output = len(np.unique(y))
        self.hidden = list(hidden_layers)

    @property
    def sizes(self):
        return [self.input] + self.hidden + [self.output]
```

En in je MLP:

```
python Code kopiëren
self.layer_sizes = arch.sizes
```

Daarna kun je ALLES generiek doen.



## ⚡ Voordelen en nadelen samengevat

### ✓ Functie

- simpel
- weinig code
  - beperkt uitbreidbaar

### ✓ Subclass (class Architecture)

- schaalbaar
- schoon ontwerp
- makkelijk meerdere configuraties (activatie, dropout, batchnorm)
- makkelijk integreren met MLP class
  - iets meer code

### ✓ Dictionary

- snel
- simpel
  - minder structuur
  - minder uitbreidbaar

## 💡 Conclusie

Ja, jouw aanpak werkt, maar de beste manier is:

→ **1 Architecture class → 1 lijst van layer\_sizes → 1 generieke MLP class.**

Zodra je dit hebt, hoef je nooit meer handmatig lagen te coderen.

Forward en backward worden één for-loop in beide richtingen.



oke top ik heb nog Architecture los van de MLP staan en het geeft me een lijst met nodes per layer, dit is wat ik wou hebben. kan je me nu helpen met het binden van weights aan elke connectie?

Helemaal goed — je hebt nu de **architectuur** (bijvoorbeeld `[10, 3, 2, 1]`) en je wilt nu **automatisch alle weight-matrices en bias-vectors genereren**, één per laag, zonder handmatig iets te hoeven schrijven.

Perfect. Ik geef je een **superduidelijk systeem**, zoals je dat ook in frameworks als PyTorch ziet — maar dan *helemaal handmatig met numpy*.

## 🎯 Doel

Voor een netwerk met `layer_sizes`:

csharp

Code kopiëren

```
[input_dim, h1, h2, output_dim]
```

wil je automatisch:

✓ **W1 met shape (input\_dim × h1)**

✓ **b1 met shape (1 × h1)**

✓ **W2 met shape (h1 × h2)**



- ✓ b2 met shape  $(1 \times h2)$
- ✓ W3 met shape  $(h2 \times \text{output\_dim})$
- ✓ b3 met shape  $(1 \times \text{output\_dim})$

En dan deze opslaan in:

```
ini
```

```
self.weights = [W1, W2, W3]
self.biases = [b1, b2, b3]
```

Code kopiëren

↳ Dit maakt forward en backprop *heel eenvoudig* met loops.

## ✓ Stap 1 — Weight initialisatie-functie binnen je MLP

Zodra je weet:

```
lua
```

```
self.architecture.layer_sizes = [input, h1, h2, ..., output]
```

Code kopiëren

kun je de gewichten genereren met een **for-loop over opeenvolgende paren**:

```
python
```

```
def initialise_parameters(self):
    layer_sizes = self.architecture.layer_sizes

    self.weights = []
    self.biases = []

    for i in range(len(layer_sizes) - 1):
        in_dim = layer_sizes[i]
        out_dim = layer_sizes[i+1]

        # Xavier/He initialisation is beter dan random:
        W = np.random.randn(in_dim, out_dim) * np.sqrt(2 / in_dim)
        b = np.zeros((1, out_dim))

        self.weights.append(W)
        self.biases.append(b)
```

Code kopiëren

## ✓ Plaats deze in je MLP class

python

```
class MultilayerPerceptron:
    def __init__(self, X, y, hidden_size=(3,2)):
        # - jouw init -
        n_features = X.shape[1]
        outputs = np.unique(y)

        self.input_size = n_features
        self.hidden_size = hidden_size
        self.output_size = len(outputs)

        self.architecture = Architecture(
            input_size=self.input_size,
            hidden_sizes=self.hidden_size,
            output_size=self.output_size
        )

        # Initialiseer gewichten
        self.initialise_parameters()
```



```
def initialise_parameters(self):
    layer_sizes = self.architecture.layer_sizes

    self.weights = []
    self.biases = []

    for i in range(len(layer_sizes) - 1):
        in_dim = layer_sizes[i]
        out_dim = layer_sizes[i+1]

        W = np.random.randn(in_dim, out_dim) * np.sqrt(2 / in_dim)
        b = np.zeros((1, out_dim))

        self.weights.append(W)
        self.biases.append(b)
```

## Resultaat

Voor:

```
python □ Code kopieren  
layer sizes = [10, 3, 2, 1]
```

krijg je:

- `self.weights[0]` →  $(10 \times 3)$
  - `self.weights[1]` →  $(3 \times 2)$
  - `self.weights[2]` →  $(2 \times 1)$

281

#### **± Stel een vraag**

9

zou je een forward pass function voor me willen maken?

Zeker! Ik maak een perfect generieke forward pass functie die:

- werkt met jouw bestaande architectuur
  - de weights & biases uit `self.weights` en `self.biases` gebruikt
  - ALLE lagen ondersteunt (maakt niet uit hoeveel)
  - activaties en z-waardes opslaat (nodig voor backprop!)
  - ReLU of Sigmoid (kan je beide krijgen)
  - superduidelijk en uitbreidbaar

Hier is de cleanste vorm die in elk zelfgebouwd MLP past.

Forward pass voor jouw MLP (volledig generiek)

Voeg dit toe in class MultilayerPerceptron:

```
python
```

Code kopieren

```
def forward(self, X):
    ...
    Voert een forward pass uit door het netwerk.
    Retourneert:
        activations: lijst van activatie outputs per laag (inclusief input)
        zs: lijst van pre-activaties ( $z = W @ a + b$ ) per laag
    ...
    activations = [X]      #  $a_0 = X$ 
    zs = []                # pre-activaties
```

```

python

def forward(self, X):
    """
    Voert een forward pass uit door het netwerk.
    Retourneert:
        activations: lijst van activatie outputs per laag (inclusief input)
        zs: lijst van pre-activaties ( $z = W@a + b$ ) per laag
    """
    activations = [X] #  $a^0 = X$ 
    zs = [] # pre-activaties

    a = X
    for i in range(len(self.weights)):
        W = self.weights[i]
        b = self.biases[i]

        z = a @ W + b
        zs.append(z)

        # Laatste laag Lineair (geschikt voor regressie)
        if i == len(self.weights) - 1:
            a = z
        else:
            # ReLU
            a = np.maximum(0, z)

        activations.append(a)

    return activations, zs

```



## 💡 Uitleg wat deze functie doet

### 1. Activaties opslaan

```

python

activations = [X]

```

Input is activatie van laag 0:  $a^{(0)} = X$ .

### 2. Voor elke laag:

- Bereken pre-activatie:  

$$z = aW + b$$
- Voor tussenlagen: ReLU
- Voor de laatste laag: geen activatie (lineair)

### 3. Return voor backprop:

- `activations`
- `zs`

Deze heb je *nodig* voor je backward pass.

---

## 📝 Voorbeeld gebruik

```

python

acts, zs = model.forward(X)
print("output shape:", acts[-1].shape)

```



## ⭐ Wil ie Sigmoid, Tanh of Softmax inbouwen?

## 💡 Vervang sigmoid, tanh of softmax implementeren.

Heel makkelijk door een extra regel toe te voegen:

```
python
```

```
if self.activation == "relu":  
    a = np.maximum(0, z)  
elif self.activation == "sigmoid":  
    a = 1 / (1 + np.exp(-z))
```

Code

Zeg het maar, dan bouw ik het netjes in je volledige class.

## 👉 Klaar voor de backpropagation stap?

Ik kan nu:

- de complete backward pass
- loss functions (MSE / cross entropy)
- weight updates (gradient descent)
- training loop

voor je implementeren in hetzelfde strakke format.

Wil je dat ik **de hele MLP structureer** inclusief training?

🔗 ↻ 🔍 ⌂ ...

omdat mijn trainingsdata 3 categorieën heeft wil ik een softmax gebruiken voor de laatste laag

Perfect!

Als je **3 categorieën** hebt, dan is Softmax absoluut de juiste keuze voor de **laatste laag** van je MLP — dit maakt je netwerk een **multiclass classifier**.

Ik geef je nu:

1. ✓ De Softmax functie (stabiel, met clipping)
2. ✓ Een aangepaste forward-pass die Softmax alleen op de laatste laag toepast
3. ✓ De shape-checks (heel belangrijk bij 3 klassen)
4. ✓ Tips voor welke loss je moet gebruiken (cross-entropy)

Alles sluit naadloos aan op jouw huidige MLP-structuur.

## 🧠 1. Softmax functie (numeriek stabiel)

Gebruik deze — hij voorkomt overflow door subtractie van `max`:

```
python
```

```
def softmax(self, z):  
    z_shifted = z - np.max(z, axis=1, keepdims=True) # stabiliteit  
    exp_scores = np.exp(z_shifted)  
    return exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
```

Code kopiëren

## 🔥 2. Forward pass met Softmax op de laatste laag

Dit is jouw aangepaste forward functie:

```
python
```

```
def forward(self, X):
```

Code kopiëren

```

activations = [X]
zs = []
a = X

for i in range(len(self.weights)):
    W = self.weights[i]
    b = self.biases[i]

    z = a @ W + b
    zs.append(z)

    # 🟢 Laatste laag --> Softmax
    if i == len(self.weights) - 1:
        a = self.softmax(z)

    else:
        # 🟡 Tussenlagen: ReLU
        a = np.maximum(0, z)

    activations.append(a)

return activations, zs

```

- ✓ Softmax alleen op de laatste laag
- ✓ ReLU op alle hidden layers
- ✓ Output shape wordt `(batch_size, 3)`
- ✓ Perfect voor cross-entropy loss

### ✓ 3. shape-check (hoeveel neuronen in output laag?)

Je moet de laatste waarde in je architecture gelijk maken aan het aantal klassen:

hoezo moet ik de activaties opslaan?

Heel goede vraag — en dit is *essentieel* om backpropagation te begrijpen.

Het opslaan van activaties (en z-waarden) is **niet optioneel** → het is de kern van hoe backprop werkt.

Ik leg het je zo duidelijk mogelijk uit.

## 🎯 Waarom moet je activaties opslaan?

Omdat je tijdens **backpropagation** deze drie dingen nodig hebt:

1. De activatie van elke laag (`a^{(l)}`)
2. De pre-activatie z-waarde van elke laag (`z^{(l)}`)
3. De gewichten van elke laag (`W^{(l)}`)

Backprop komt namelijk neer op:

$$\delta^{(l)} = (\delta^{(l+1)} W^{(l+1)T}) \odot f'(z^{(l)})$$

Je ziet dat je nodig hebt:

- $\delta$  van de volgende laag
- $W^T$  van de volgende laag
- $f'(z)$  van de huidige laag
- activatie  $a^{(l)}$  voor de gradient van de gewichten

Als je deze waarden **niet bewaart**, kun je *niet* terugrekenen door het netwerk.

## 🧠 Stap voor stap intuïtie

### 👉 Tijdens forward pass

Het netwerk berekent:

- eerst z's
- dan activaties a's
- voor elke laag in volgorde

Je gaat dus van:

```
makefile
```

```
a0 = X
z1 = XW1 + b1
a1 = ReLU(z1)
z2 = a1W2 + b2
a2 = ReLU(z2)
...
```

Code kopiëren

### 👉 Tijdens backward pass

Je moet die lagen precies **in omgekeerde richting** doorlopen en:

- de afgeleide van de activatiefunctie gebruiken ( $f'(z)$ )
- de inputactivatie van die laag kennen ( $a$ )
- de output-error van de volgende laag gebruiken

Als je de activaties en z's **niet had opgeslagen**, zou je de forward pass opnieuw moeten uitvoeren voor iedere gradientberekening.

Dat is extreem duur en inefficiënt.



## ⭐ Concreet voorbeeld

Stel je wil de gradient van  $W_2$  berekenen:

$$\frac{\partial \text{Loss}}{\partial W^{(2)}} = a^{(1)T} \cdot \delta^{(2)}$$

Dan heb je  $a^{(1)}$  nodig.

Dat is de activatie **van de laag vóór  $W_2$** .

👉 Maar op het moment dat je in de backward pass bent, is die forward activatie allang overschreven als je hem niet had opgeslagen.

Daarom slaan we op:

```
python
```

```
activations = [a0(input), a1, a2, ..., a_last]
zs = [z1, z2, ..., z_last]
```

Code kopiëren

## 🔥 Zonder activaties → geen backprop

Backprop heeft de volgende data per laag nodig:

Nodig voor

Variable

afgeleide  $f'(z)$

z-values per laag

gradient van  $W$  ( $a^{(1)T} \cdot \delta^{(2)}$ )

activaties

$\delta^{(l)}$  berekenen



activaties + weights

bias gradient (sum of deltas)

### delta values

Je hebt dus **activering** én pre-activatie van *elke* laag nodig.

## 📦 Samenvatting

Tijdens forward pass moeten we opslaan:

Wat	Waarom
<input type="checkbox"/>	nodig voor derivative $f(z)$
<input checked="" type="checkbox"/>	nodig voor gradient: $a^T \delta$
output van elke laag	nodig om backward stap per laag te doen

Conceptueel (mooi inzicht!)

Backprop is:

een forward pass om de activaties op te slaan  
een backward pass om ze te gebruiken

Je rekent dus:

- vooruit om op te slaan
  - achteruit om te optimaliseren

👉 Wil je dat ik nu de volledige backward pass schrijf voor jouw softmax + multiclass cross-entropy?

**אֲדֹנָי-כְּסֹעַד אֱשֹׁבֵנִים וְיָבִין**



zou ik de forward pass dan niet beter in een eigen class kunnen beschrijven?

Goede vraag — en het laat zien dat je begint na te denken in termen van **software-architectuur**, niet alleen wiskunde.

Laten we het goed analyseren.

### Korte conclusie

Je kunt de forward pass in een aparte class zetten, maar het is meestal NIET de beste keuze.

Een forward pass hoort bij het *model* zelf, niet bij een externe helper-class.

Maar

Er zijn situaties waarin een aparte class wel handig is. 

Ik leg je alle opties uit zodat je precies begrijpt wat het beste is voor jouw project.

💡 Waarom forward pass normaal in de MLP-class hoort

In jouw model bestaat een forward pass uit:

- weights ( self.weights )
  - biases ( self.biases )
  - activatiefuncties

- architectuur (`self.architecture`)

De forward pass heeft al deze modelgegevens nodig.

Daarom is het logisch dat de forward functie onderdeel is van je `MultilayerPerceptron` zelf.

Het is dus een *method* van het model:

```
python
class MultilayerPerceptron:
    def forward(self, X):
        ...

```

Want:

- de forward pass verandert afhankelijk van *hoe* het model gebouwd is
- de forward pass gebruikt interne attributen van dit specifieke MLP
- het MLP heeft verantwoordelijkheid voor hoe data "door het netwerk" stroomt

Dit heet **object encapsulation**:

het model beheert zijn eigen gedrag.

## 🔥 Wanneer een aparte class WEL zinvol kan zijn

Er zijn 2 situaties waarin het logisch kan zijn:

### ✓ 1. Als je een *losse computational graph engine* wil bouwen

Zoals PyTorch of TensorFlow:

- Layers zijn objecten
- Forward-pass wordt opgebouwd door chaining 
- Model is een lijst van layers

waarom assign je `a = X` en gebruik je niet gewoon direct `X` in de formules?



Goede vraag — en dit is precies het soort inzicht dat laat zien dat je **de interne logica van een MLP echt begint te begrijpen**.

Dit is niet iets wat beginners vragen.

Het korte antwoord:

We gebruiken `a = X` zodat we in de forward-pass hetzelfde algemene mechanisme kunnen gebruiken voor alle lagen — inclusief de inputlaag — en zodat backprop later correct werkt.

Maar ik leg je het stap voor stap uit.

## ⭐ 1. Een MLP werkt in termen van activaties $a^{(l)}$

In een neuraal netwerk noemen we:

- de input **activatie van laag 0**

$$a^{(0)} = X$$

- de activatie van laag 1:

$$a^{(1)} = f(W^{(1)}a^{(0)} + b^{(1)})$$



- enzovoort...

Afbeelding: Neuraal netwerk met 3 lagen. De inputlaag heeft 3 activeringen. De verbindingen tussen de lagen zijn aangegeven door pijlen.

Als jij direct `X` gebruikt, dan moet je in je code **special cases** maken:

```
python
z1 = X @ W1 + b1
a1 = activation(z1)
z2 = a1 @ W2 + b2
a2 = activation(z2)
...
```

Code kopiëren

Dat wil je **NIET**, want je model moet schaalbaar zijn.

Daarom maak je:

```
python
a = X
```

Code kopiëren

en dan gebruik je **dezelfde formule voor alle lagen**:

```
python
z = a @ W + b
a = activation(z)
```

Code kopiëren

Dit laat je MLP automatisch werken voor:

- 1 hidden layer
- 10 hidden layers
- 100 hidden layers

zonder dat je code opnieuw moet worden geschreven.



2. Omdat `a` steeds de activatie van de vorige laag represeneert

2. Omdat `a` steeds de activatie van de vorige laag represeneert

Dit is het meest fundamentele punt.

In laag 1:

```
ini
a = X
z1 = X @ W1 + b1
a1 = activation(z1)
```

Code kopiëren

In laag 2:

```
ini
a = a1
z2 = a1 @ W2 + b2
a2 = activation(z2)
```

Code kopiëren

In laag 3:

```
ini
a = a2
z3 = a2 @ W3 + b3
a3 = activation(z3)
```

Code kopiëren

Je ziet:

`a` is altijd de *input* voor de huidige laag

en altijd de *output* van de vorige laag

Daarom heet het "activatie".



## ★ 3. Omdat je activaties moet opslaan voor backpropagation

Backprop heeft dit nodig:

$$\frac{\partial L}{\partial W^{(l)}} = a^{(l-1)T} \delta^{(l)}$$

Dat betekent dat je altijd:

- de activaties van de vorige laag
- de z-waarden van de huidige laag

moet terugvinden.

Dus we doen:

```
python Code kopiëren
activations = [X]    # a^{(0)}
...
activations.append(a)
```

Als je `a` niet gebruikt maar direct `X` overal hard-coded inzet, dan kun je backprop niet generiek bouwen.

## ★ 4. Omdat anders forward-pass en backward-pass NIET bij elkaar passen

De forward-pass bestaat uit:

```
python Code kopiëren
a0 = X
z1 = a0 W1 + b1
z2 = a1 W2 + b2
a1 = f(z1)
a2 = f(z2)
...
```

De backward-pass in reverse:

```
python
delta_L = ...
delta_L-1 = delta_L @ W_L^T ⊕ f'(z_L-1)
delta_L-2 = delta_L-1 @ W_{L-1}^T ⊕ f'(z_{L-2})
...
```

Dit werkt alleen als de forward-pass activaties als:

- `a0`
- `a1`
- `a2`

zijn opgeslagen in één variabele: `a`.

## ★ 5. Omdat `a` een buffer is

Tijdens een forward-pass is `a` een soort "flow variable":

```
python
a = X                      # start
for each layer:
    z = a @ W + b          # compute z
    a = activation(z)       # compute new a
```

Dus:

- a verandert elke laag
- x is alleen de eerste laag
- je kunt niet elke keer X gebruiken, want je moet verder kunnen gaan naar de volgende lagen

## ★ Conclusie

We schrijven:

```
python
```

Code kopiëren

```
a = X
```

omdat:

1. het model schaalbaar wordt (geen hard-coded lagen)
2. activaties en z's perfect kunnen worden opgeslagen
3. backprop exact de activaties krijgt die het nodig heeft
4. forward-pass logica consistent blijft voor alle lagen
5.  altijd de output van de vorige laag represeneert

Je zou dit kunnen zien als:

"X is alleen de eerste activatie. a is activatie in het algemeen."

## ✍ Wil je nu dat ik jouw backpropagation toevoeg?

Inclusief:

- delta's voor softmax + cross-entropy
- generieke backward-pass voor ALLE lagen
- update-regels voor W en b
- complete training loop

Zeg maar: ja, backprop graag!



Train				
	Age	Sleep_Duration	Study_Hours	Screen_Time
13	23	5.1	9.8	1.1
14	24	4.2	8.8	1.6
15	19	8.3	3.8	3.1
16	18	7.6	7.9	1.3
17	21	8.1	7.2	3
18	23	4.5	3.8	2.8

ja, backprop graag! ik vraag me nu wel af waar we de thresholds van het model moeten gaan definiëren, ik weet niet of ik het al had geüpload maar bij deze geef ik je nog een keertje de training set zodat je weet welke categorieën er zijn



Perfect — we gaan **backprop** bouwen, maar ik begin eerst met jouw vraag over “thresholds”, want dit is belangrijk vóór we de training-loop maken.

Er is namelijk een groot misverstand bij studenten die softmax-modellen trainen:

### ★ Een Softmax-gebaseerd MLP heeft GEEN handmatige thresholds nodig.

Softmax maakt automatisch een kansverdeling:

csharp

Code kopiëren

```
[0.70, 0.20, 0.10]
```

Daaruit kiezen we gewoon:

```
class = argmax(probabilities)
```

En dat is alles.

Geen thresholds zoals bij sigmoid of binary classification.

Maar jouw dataset heeft labels:

“bad”, “medium”, “good” → 3 klassen.

Dus:

- ✓ Jij traint een softmax-classifier met 3 outputs
- ✓ We kiezen de klasse met de hoogste kans
- ✓ Er zijn GEEN thresholds nodig

Tenzij je zelf een “custom scoring rule” wil (zeldzaam).



### ★ Maar waarom stond er in jouw opdracht “threshold\_medium” en “threshold\_good”?

Omdat dat van toepassing zou zijn bij:

- BINARY classification (sigmoid → 0/1)
- REGRESSION op score 0–1

Maar NIET bij softmax.

Softmax doet:

ini

Code kopiëren

```
output = [p(bad), p(medium), p(good)]
```

en de voorspelling is gewoon:

python

Code kopiëren

```
np.argmax(output, axis=1)
```

Dus ook voor jouw dataset gewoon **argmax**.

### ★ Wat moeten we dus doen?

✗ GEEN thresholds defineren

✓ Labels omzetten naar:

rust

Code kopiëren

```
bad    -> 0
```

```
medium -> 1
```



0

...

```
medium -> 1  
good   -> 2
```

- ✓ Softmax + cross-entropy trainen
- ✓ Backprop implementeren
- ✓ Predict = argmax

En dat gaan we nu bouwen.

## 🚀 BACKPROPAGATION (Softmax + Cross-Entropy)

Omdat jouw model softmax gebruikt, krijgen we een **perfecte simplificatie**:

$$\delta(L) = a(L) - y$$

(de "true labels" moeten one-hot encoded zijn)

En dan:

$$\delta(l) = (\delta(l+1) @ W(l+1).T) \odot f'(z(l))$$

Gradients:

$$dW(l) = a(l-1).T @ \delta(l)$$

$$db(l) = \text{sum over samples} (\delta(l))$$

Daarna:

$$W(l) -= \text{learning\_rate} * dW(l)$$

$$b(l) -= \text{learning\_rate} * db(l)$$



### ⭐ Stap 1 — One-hot encoding helper

Voeg deze toe in jouw class:

```
python  
  
def one_hot(self, y):  
    n_classes = self.output_size  
    onehot = np.zeros((y.size, n_classes))  
    onehot[np.arange(y.size), y] = 1  
    return onehot
```

### ⭐ Stap 2 — Cross entropy loss

```
python  
  
def cross_entropy(self, y_true_onehot, y_pred):  
    eps = 1e-12  
    y_pred = np.clip(y_pred, eps, 1 - eps)  
    return -np.mean(np.sum(y_true_onehot * np.log(y_pred), axis=1))
```

### ⭐ Stap 3 — Backprop functie (volledig generiek!)

Hier komt de grote jongen.

```
python  
  
def backpropagation(self, activations, zs, y_true, learning_rate):  
    ...  
      
    Performs backprop using softmax + cross-entropy.
```

```

activations: list of a[1]
zs: list of z[1]
y_true: integer labels
"""

m = y_true.shape[0]
y_onehot = self.one_hot(y_true)

#  $\delta(L)$ : softmax + cross entropy simplifies beautifully
delta = activations[-1] - y_onehot

# gradients storage
dW = [None] * len(self.weights)
db = [None] * len(self.biases)

# Last layer gradients
dW[-1] = activations[-2].T @ delta / m
db[-1] = np.sum(delta, axis=0, keepdims=True) / m

# backprop through hidden layers
for l in range(len(self.weights) - 2, -1, -1):
    W_next = self.weights[l + 1]
    delta = delta @ W_next.T

    # derivative of activation
    if self.activation_function == "ReLU":
        delta *= (zs[l] > 0)
    else:
        s = self.sigmoid(zs[l])
        delta *= s * (1 - s)

    dW[l] = activations[l].T @ delta / m
    db[l] = np.sum(delta, axis=0, keepdims=True) / m

```

```

# update weights
for l in range(len(self.weights)):
    self.weights[l] -= learning_rate * dW[l]
    self.biases[l] -= learning_rate * db[l]

```

## ★ Step 4 — Train loop

```

python

def train_model(self, X_train, y_train, learning_rate=0.01, epochs=200):
    losses = []

    for epoch in range(epochs):
        activations, zs = self.forward(X_train)
        loss = self.cross_entropy(self.one_hot(y_train), activations[-1])
        losses.append(loss)

        self.backpropagation(activations, zs, y_train, learning_rate)

        if epoch % 50 == 0:
            print(f"Epoch {epoch}, Loss: {loss:.4f}")

    return losses

```

## ★ Step 5 — Predict function

```

python
def predict(self, X):

```

```
activations, _ = self.forward(X)
probs = activations[-1]
return np.argmax(probs, axis=1)
```

 Code kop

 Totale flow na jouw aanpassingen

1. Train dataset laden
  2. Labels converteren naar integer (bad=0, medium=1, good=2)
  3. Model initialiseren
  4. `model.train_model(X, y)`
  5. `model.predict(X_test)`



Maar voor softmax is het niet nodig en niet standaard.

👉 Wil je dat ik nu:

- de volledige complete class uitschrijf met forward + backprop + train + predict?

OF

  - zal ik eerst je train.csv even analyseren en de labels correct mappen naar integers?

Je foutmelding is glashelder:

pgsql

 Code kopieren

UFUNC\_NOLOOP\_ERROR: ufunc 'matmul' did not contain a loop with signature matching types (dtype('<U4'), dtype('float64')) -> None

Dit betekent:

**! JE INPUT X IS GEEN NUMPY FLOAT ARRAY, MAAR EEN ARRAY VAN STRINGS**

(dtype <u4 betekent "unicode string van max 4 chars")

En het netwerk probeert dan:

css

 Code kopieren

a @ W

wat is:

nginx

 Code kopieren

```
string_matrix @ float_matrix
```

→ matrixvermenigvuldiging tussen strings en floats is onmogelijk

Dus de forward-pass faalt hier:

python

 Code kopieren

## ⭐ WAAROM IS DIT GEBEURD?

In je `data_extraction_csv()` functie converteer je de CSV direct:

python

Code kopiëren

```
X = np.array(X)  
y = np.array(y)
```

Maar CSV's bevatten **strings**, ook als het cijfers zijn.

Dus je krijgt zoiets als:

css

Code kopiëren

```
[[ '21' '6.5' '3' ...], ['20' '6.2' ...]]
```

- ➡ numpy maakt hier arrays van dtype `<U4`
- ➡ dat zijn **strings**
- ➡ je MLP wil floats

## ⭐ ✅ OPLOSSING: CAST X NAAR FLOATS + y NAAR INTEGER LABELS

Voeg deze **ENIGE** regel toe in `data_extraction_csv`:

python

Code kopiëren

```
X = np.array(X, dtype=float)
```



En voor y (categorische labels):

1. eerst y laten zoals het is
2. later mappen naar integers

## ⭐ DE VOLLEDIG CORRECTE FUNCTIE:

python

Code k

```
def data_extraction_csv(csv_file):  
    data = [line.strip().split(',') for line in open(csv_file, 'r')]  
    data.pop(0)  
  
    X = []  
    y = []  
    for line in data:  
        y.append(line.pop(-1))  
        X.append(line)  
  
    X = np.array(X, dtype=float) # ⚡ FIX: convert to float array  
    y = np.array(y) # keep as strings for now  
    return X, y
```

## ⭐ JA, MAAR JE LABELS ZIJN NOG STRINGS ("bad", "medium", "good")

Klopt — daarna moet je y converteren naar integers voor training.

Voeg toe NA het laden van X,y:

python



Code k

```

label_map = {label: idx for idx, label in enumerate(np.unique(y))}
y_int = np.array([label_map[label] for label in y])

```

Bijv.:

```

bash

{'bad': 0, 'good': 1, 'medium': 2}

```

Of eventueel in alfabetische volgorde:

```

bash

{"bad": 0, "medium": 1, "good": 2}

```

Beide zijn prima — zolang het consistent gebeurt.

## ★ COMPLETE FIX VOOR HET BEGIN VAN JE SCRIPT

```

python

X, y = data_extraction_csv("data\\train.csv")

# convert labels to integers
unique_labels = np.unique(y)
label_map = {label: idx for idx, label in enumerate(unique_labels)}
y = np.array([label_map[label] for label in y], dtype=int)

X_test, y_test = data_extraction_csv("data\\test.csv")
y_test = np.array([label_map[label] for label in y_test], dtype=int)

```

```

def softmax(z):
    """der softmax(z) is niet in de vorm van een vector want we moet de wolk te classificeren
    der softmax(z[i], z[j])
    print(softmax, "\n")
    z = [1, 2, 3]
    for observation in z:
        softmax = np.exp(observation - np.max(observation)) / np.sum(np.exp(observation))
        softmax = softmax / np.sum(softmax)
    print(softmax)
    print(softmax.tolist())
    return softmax

class ActivationFunction:
    def __init__(self, activation_function):
        self.activation_function = activation_function
        if self.activation_function == "ReLU":
            return self._relu()
        elif self.activation_function == "Sigmoid":
            return self._sigmoid()
        elif self.activation_function == "tanh":
            return self._tanh()
        else:
            raise ValueError("Unknown activation function")

    def _relu(self, x):
        if x < 0:
            return 0
        else:
            return x

    def _sigmoid(self, x):
        if self.activation_function == "ReLU":
            return self._relu(x)
        elif self.activation_function == "Sigmoid":
            return 1 / (1 + np.exp(-x))
        elif self.activation_function == "tanh":
            return np.tanh(x)
        else:
            raise ValueError("Unknown activation function")

    def _tanh(self, x):
        if self.activation_function == "ReLU":
            return self._relu(x)
        elif self.activation_function == "Sigmoid":
            return 1 / (1 + np.exp(-x))
        elif self.activation_function == "tanh":
            return np.tanh(x)
        else:
            raise ValueError("Unknown activation function")

```

hoezo zijn deze twee soorten lijsten verschillend? je ziet namelijk dat de softmax lijsten retourneert met commas tussen elke waarde, maar bij de z\_value layer 3 zie je dat niet



Goede observatie — dit is geen wiskundig verschil, maar **een verschil in type en hoe Python ze print**. Ik leg het stap voor stap uit.

### 1. z\_value layer 3 → NumPy array

Dit zie je hier:

text



Code kopiëren

text

 Code kopiëren

```
z_value layer 3 [[-13.63 84.24 108.51]
[-1.50 9.32 12.00]
...
]
```

Dit is een **NumPy array**.

NumPy print arrays **zonder komma's** om het compacter en leesbaarder te maken voor numerieke data.

Bijvoorbeeld:

python

 Code kopiëren

```
import numpy as np

a = np.array([1.0, 2.0, 3.0])
print(a)
```

Output:

csharp

 Code kopiëren

```
[1. 2. 3.]
```

 **Geen komma's = NumPy printstijl**



## 2. softmax → Python list

In jouw softmax doe je expliciet:

python

 Code kopiëren

```
softmax.tolist()
```

En daarna:

python

 Code kopiëren

```
z_softmax.append(softmax.tolist())
```

Dat maakt er een **gewone Python list** van.

Python lists worden altijd geprint **met komma's**:

python

 Code kopiëren

```
b = [1.0, 2.0, 3.0]
print(b)
```

Output:

csharp

 Code kopiëren

```
[1.0, 2.0, 3.0]
```



 **Wel komma's = Python list**

[1.0, 2.0, 3.0]

### ➡ Wel komma's = Python list

## 3. Samenvatting (dit is de kern)

Wat je ziet	Type	Waarom
[1. 2. 3.]	numpy.ndarray	NumPy-printstijl
[1.0, 2.0, 3.0]	list	Python-printstijl
Softmax-output	list	Door .tolist()
z-values	NumPy array	Komt direct uit matrixberekening

## 4. Belangrijk voor je MLP (advies)

Voor een MLP **wil je eigenlijk overal NumPy arrays gebruiken**, ook voor softmax.

Lists maken later je backprop **onnodig lastig**.

**Betere softmax (blijft NumPy):**

python



Code kopiëren

def softmax(self, z):

```
def softmax(self, z):
    z_softmax = []
    for observation in z:
        observation = np.array(observation, dtype=float)
        exp_obs = np.exp(observation - np.max(observation))
        softmax = exp_obs / np.sum(exp_obs)
        z_softmax.append(softmax)
    return np.array(z_softmax)
```

python

Code kopiëren

def softmax(self, z):

```
z_softmax = []
for observation in z:
    observation = np.array(observation, dtype=float)
    exp_obs = np.exp(observation - np.max(observation))
    softmax = exp_obs / np.sum(exp_obs)
    z_softmax.append(softmax)
return np.array(z_softmax)
```

Dan krijg je:

text

Code kopiëren

[[8.92e-54 2.88e-11 9.99e-01]

...]

➡ consistent

➡ makkelijker differentiëren

➡ minder bugs later

## 5. Kort antwoord op jouw vraag

Waarom zijn deze lijsten verschillend?



Omdat:

- `z_value` layer 3 een **NumPy array** is → geen komma's
  - `softmax` een **Python list** is (door `.tolist()`) → wel komma's

Als je wilt, kan ik met je meekijken of je **forward + backward propagation** nu overal netjes NumPy-consistent is (dat scheelt je straks veel debugtijd).



ik krijg deze error dat cross\_entropy drie argumenten ontvangt, terwijl ik alleen prediction en y\_onehot geef, wat is hier aan de hand?

Dit is een **klassieke Python-valkuil**, en je zit er inhoudelijk goed — het probleem is niet je aanroep, maar **hoe Python methodes telt**.

## Wat de error echt betekent

### De foutmelding:

SCSS

 Code kopieren

categorical cross entropy() takes 2 positional arguments but 3 were given

komt van deze regel:

python

 Code kopieren

```
loss = self.categorical_crossentropy(prediction, y_onehot)
```

En jij denkt terecht:

"Ik geef er maar twee: prediction en y\_onehot "

Maar Python ziet dit zo:

python

 Code kopieren

```
self.categorical_crossentropy(self, prediction, y_onehot)
```

 self wordt automatisch meegegeven

👉 `self` wordt automatisch meegegeven

## Waar gaat het mis?

Je methode is waarschijnlijk zo gedefinieerd:

```
python                                     ⌂ Code kopiëren  
def categorical_cross_entropy(prediction, y_true):  
    ...
```

✗ `Zonder self`

Maar je roept hem aan als een **instance method**:

```
python                                     ⌂ Code kopiëren  
self.categorical_cross_entropy(...)
```

➡ Dan verwacht Python:

```
python                                     ⌂ Code kopiëren  
def categorical_cross_entropy(self, prediction, y_true):  
    ...
```



## De twee correcte oplossingen (kies er één)

✓ **Oplossing 1 — maak het een echte class-method (aanrader)**

```
python                                     ⌂ Code kopiëren  
def categorical_cross_entropy(self, prediction, y_true):  
    eps = 1e-15  
    prediction = np.clip(prediction, eps, 1 - eps)  
    return -np.sum(y_true * np.log(prediction)) / y_true.shape[0]
```

- ✓ past bij de rest van je MLP
- ✓ consistent met `self.softmax`, `self.relu`, etc.

✓ **Oplossing 2 — maak het expliciet een static function**

Als je **geen `self` nodig hebt**:

```
python                                     ⌂ Code kopiëren  
@staticmethod  
def categorical_cross_entropy(prediction, y_true):  
    ...
```

Dan mag je hem ook zo aanroepen:



```
python                                     ⌂ Code kopiëren  
categorical_cross_entropy(prediction, y_true)
```

```
python
```

Code kopiëren

```
loss = self.categorical_crossentropy(prediction, y_onehot)
```

of:

```
python
```

Code kopiëren

```
loss = MultiLayerPerceptron.categorical_crossentropy(prediction, y_onehot)
```

## Hoe weet je dit voortaan meteen?

### Vuistregel

Als je een functie via `self.` aanroeft → eerste argument moet `self` zijn

## Extra (belangrijk voor jouw backprop!)

Ik zie hier:

```
python
```

Code kopiëren

```
delta = error * self.d_nonlin_selector(activations[layer])
```



⚠ Bij softmax + categorical cross-entropy hoort géén extra afgeleide

⚠ Bij softmax + categorical cross-entropy hoort géén extra afgeleide

Correct is:

```
python
```

Code kopiëren

```
delta = prediction - y_onehot
```

en daarna pas terugpropageren.

Als je wilt, kan ik:

- je backprop exact corrigeren voor softmax + CCE
- of je helpen dit netjes te structureren per layer (scheelt bugs)

Je zit technisch echt op het juiste spoor — dit is precies het soort fout dat iedereen tegenkomt bij "MLP from scratch".



```
def categorical_crossentropy(wfit, y_predict, y):  
    |    return -(y_predict * np.log(y))  
yfit  
X_train, y_train = data_extraction_csv("data/train.csv")  
model = MultiLayerPerceptron(6, 3)  
print(model.architecture.layer_sizes)  
activations, z_values = model.forward(X_train)  
model.backpropagation(activations, z_values, y_train)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS  
[2.9646275e-01 5.4208442e-02 6.4929383e-01]  
[3.9921414e-01 3.2095605e-01 9.7075000e-01]  
[3.9338660e-01 1.8918929e-01 9.9338759e-01]

```

1 def categorical_cross_entropy(self, y_predict, y):
2     | return -(np.log(y_predict)*np.log(y))
3
4 X_train, y_train = data_extraction_csv("data/train.csv")
5
6 model = MultilayerPerceptron(3, 2)
7 print(model.architecture_layer_sizes)
8 activations, z_values = model.forward(X_train)
9 model.backpropagation(activations, z_values, y_train)

```

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
[7.9646275e-01 5.420942e-02 6.402923e-01]
[1.39924147e-02 3.282656e-01 9.797588e-01]
[ 9.9184046e-01 -1.3807602e-08 -9.938373e-01]
[ 2.7093859e-01 6.5211625e-01 5.8117766e-01]
[-6.2830943e-02 -2.6109723e-02 3.5641057e-02]

```

```

[ 0.83669479e-01 -2.45996858e-08 -0.86339679e-01]
[ 2.36464275e-01 5.420942e-02 3.58706712e-01]
[ 9.0007050e-01 -3.282656e-01 9.797588e-01]
[ 0.83669479e-01 -2.45996858e-08 -0.86339679e-01]
log
return (np.log(y_predict)*np.log(y))
[ 0. -inf -inf]
[ 0. -inf -inf]
[ -inf -inf 0.]
...
[ 0. -inf -inf]
[ 0. -inf 0.]
[ 0. -inf -inf]]

```

hoezo zijn mijn resultaten voor cross entropy 0 of -inf?



Goede vraag — dit gedrag is **exact wat je zou verwachten** met de formule die je nu gebruikt. Er spelen **twee fundamentele problemen** tegelijk, waardoor je `0` en `-inf` krijgt.

Ik loop ze rustig met je door.

## 1 Je cross-entropy-formule is wiskundig fout

Je hebt nu:

python



Code kopiëren

Je hebt nu:

python

Code kopiëren

```
def categorical_cross_entropy(self, y_predict, y):
    return -(np.log(y_predict) * np.log(y))
```

### Waarom dit fout gaat

- `y` is **one-hot encoded** → bevat alleen `0` of `1`
- `log(1) = 0`
- `log(0) = -inf`

Dus:

- `np.log(y) → 0 of -inf`
- `np.log(y_predict) * np.log(y) → 0 of inf`
- Met `-()` ervoor → `0` of `-inf`

→ Dit verklaart je output perfect.

## 2 Wat cross-entropy wel is (belangrijk)

Voor **softmax + one-hot labels** is de categorical cross-entropy:

$$\mathcal{L} = - \sum_c y_c \log(\hat{y}_c)$$

👉 niet `log(y)`

### 3 De correcte implementatie (numeriek stabiel)

Gebruik dit:

```
python
```

```
def categorical_crossentropy(self, y_pred, y_true):
    eps = 1e-15 # voorkomt log(0)
    y_pred = np.clip(y_pred, eps, 1 - eps)
    return -np.sum(y_true * np.log(y_pred)) / y_true.shape[0]
```

Code kopiëren

Waarom dit werkt

- `y_true` selecteert alleen de juiste klasse
- `log(y_pred)` is altijd gedefinieerd
- `clip` voorkomt `log(0)`
- Gemiddelde loss per sample

### 4 Waarom jij nu `0` of `-inf` ziet (samengevat)

Oorzaak	Effect
<code>log(0) in np.log(y)</code>	<code>-inf</code>
<code>log(1)</code>	<code>0</code>
Vermenigvuldigen met <code>log(y_pred)</code>	 <code>inf / 0</code>
Geen clipping	numerieke crash

→ Je code doet exact wat NumPy moet doen

### 5 Heel belangrijk voor je backprop (dit scheelt 50% bugs)

Omdat jij:

- softmax in de outputlaag
- categorical cross-entropy gebruikt

dan geldt:

$$\frac{\partial L}{\partial z} = \hat{y} - y$$

Dus in je backprop:

✗ fout:

```
python
```

```
delta = error * self.d_nonlin_selector(activations[layer])
```

Code kopiëren

✓ correct:

```
python
```

```
delta = prediction - y_onehot
```

Code kopiëren

➡ geen afgeleide van softmax nodig

## 6 Mini-check (handig om te debuggen)

Print dit één keer:

```
python
print("y_true:", y_true[0])
print("y_pred:", y_pred[0])
print("loss:", self.categorical_crossentropy(y_pred, y_true))
```

Code kopiëren

Je loss moet nu:

- positief
- niet 0
- niet  $\infty$

## Conclusie (coach-modus 😊)

Je probleem is **niet NumPy, niet softmax, niet je data** —  
het is puur een **verkeerde wiskundige definitie** van cross-entropy.

Dit is een *heel* veelgemaakte fout bij "MLP from scratch", en eerlijk: het is goed nieuws, want dit is makkelijk  
en clean te fixen.



Als ie wilt kan ik: