**Simulating Material Properties, Hofmann**

————————————————————————————————————————————————————————————

**Frage ChatGPT:**  
My output are 2 vectors which have each an x and y component. I want to predict the x and y components of those two vectors. So summarized I have 4 outputs. The input are N points where each point has x and y components, so 2 × N inputs.

**Antwort:**  
Perfect, thanks for the details! Based on your description, here's what we can say:

**🧠 Your Problem Is a Regression Task**  
Why?

• You're predicting real-valued components (x and y) of two vectors

• That gives you 4 continuous outputs (2 vectors × 2 components each)

• The model should minimize the difference between predicted and true vector components  
✅ So this is clearly a multivariate regression problem.

**🧮 Summary of Your Setup**

• Input: N 2D points → shape: (2 × N) or (N, 2)

• Output: 2 vectors (each with x and y) → 4 floats

• Loss Function: MSELoss or L1Loss

• Activation: Typically none (linear)

**🧪 PyTorch Example:**

import torch

import torch.nn as nn

model = YourModel()

input = torch.randn(batch\_size, N, 2)

target = torch.randn(batch\_size, 4)

output = model(input)

loss\_fn = nn.MSELoss()

loss = loss\_fn(output, target)

**🛠️ Tips:**

• If the vectors have properties (e.g. orthogonal, unit length), add constraints or use custom loss

• Normalize your inputs (e.g., scale x and y to [-1, 1])

————————————————————————————————————————————————————————————

**Frage:**  
I have my batch inputs like (x1,y1,x2,y2,...,xn,yn). I want to take m random (x, y) pairs and use that in get\_test\_data().

**Antwort:**  
Great! Here's how you can modify get\_test\_data() to randomly select m pairs:

import torch

from torch.utils.data import DataLoader

def get\_test\_data(batch\_size, batch\_length, shuffle=True):

dataset = VectorDataset(X, Y)

dataloader = DataLoader(dataset, batch\_size=batch\_size, shuffle=shuffle)

for inputs, targets in dataloader:

N = inputs.shape[1] // 2

assert batch\_length <= N, "Requested more pairs than available"

new\_inputs = []

for sample in inputs:

xy\_pairs = sample.view(N, 2)

indices = torch.randperm(N)[:batch\_length]

selected = xy\_pairs[indices]

new\_inputs.append(selected.flatten())

new\_inputs = torch.stack(new\_inputs)

return new\_inputs, targets

**Result:**  
Now each sample has batch\_length random (x, y) pairs instead of the full 2N.

————————————————————————————————————————————————————————————

**Frage:**  
Wie mache ich ein vollständiges Training meines Modells?

**Antwort:**  
Nutze einen Trainingsloop mit mehreren Epochen:

num\_epochs = 1000

learning\_rate = 0.001

model = BravaisLatticeRecognitionNet(input\_train\_data.shape[1])

loss\_fn = nn.MSELoss()

optimizer = torch.optim.Adam(model.parameters(), lr=learning\_rate)

for epoch in range(num\_epochs):

model.train()

output = model(input\_train\_data)

loss = loss\_fn(output, target\_train\_data)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

if (epoch + 1) % 100 == 0 or epoch == 0:

print(f"Epoch [{epoch+1}/{num\_epochs}], Loss: {loss.item():.6f}")

Optional: Füge eine Validierungsschleife mit model.eval() und torch.no\_grad() hinzu.

————————————————————————————————————————————————————————————

**Frage:**  
Wie viele Trainingsdaten brauche ich und wie sollten sie aussehen?

**Antwort:**

• Qualität vor Quantität

• Je komplexer das Modell, desto mehr Daten (Faustregel: 10× bis 100× so viele Datenpunkte wie Parameter)

• Datenaufteilung: 70–80 % Training, 10–15 % Validation, 10–15 % Test

• Gleichmäßige Abdeckung des Eingaberaums ist wichtig

• Bei physikalischen Problemen: auch Extremfälle und Symmetrien abdecken

• Synthetische Daten sind oft gut geeignet – z. B. durch Sampling, Simulationen (Monte Carlo, DFT etc.)

————————————————————————————————————————————————————————————

**Frage:**  
Ist es sinnvoll, als Input direkt die x- und y-Komponenten der G-Punkte zu übergeben?

**Antwort:**  
Ja, absolut! Eine flachgeklappte Liste der x- und y-Komponenten im reziproken Raum wie:  
Input = [Gx1, Gy1, Gx2, Gy2, ..., Gxn, Gyn]  
✅ Vorteile:

• Kein Bild nötig

• Geometrie vollständig enthalten

• Gut für MLPs oder graphbasierte Netze

**Achtung:**

1 Eingaben normalisieren (z. B. auf [-1, 1])

2 Konsistente Sortierung (nach Radius, Winkel etc.)

3 Optional: Datenaugmentierung (Rotationen, Noise)

————————————————————————————————————————————————————————————

**Frage:**  
Ist es besser, eine Bravais-Klasse zu klassifizieren oder die Gittervektoren direkt zu prädizieren?

**Antwort:**  
Kommt auf dein Ziel an:

**🔹 Ziel 1: Klassifikation**

• Output: Softmax-Vektor

• Vorteile: Einfach, robust, gut interpretierbar

• Nachteile: Kein direkter Gittervektor

**🔸 Ziel 2: Regression der Gittervektoren**

• Output: [a1x, a1y, a2x, a2y]

• Vorteile: Volle Kontrolle über Struktur

• Nachteile: Schwerer zu trainieren, Ambiguitäten durch Basiswechsel etc.

**🧠 Empfehlung:**

• Nur Klasse → Klassifikation

• Struktur rekonstruieren → Regression

• Beides? → Multi-Output-Netz (mit CrossEntropy + MSE kombiniert)

Wenn du möchtest, baue ich dir ein Beispielmodell, das gleichzeitig:

• die Bravais-Klasse klassifiziert

• und die Gittervektoren rekonstruiert.