Towards AI

Member-only story

Beyond Simple Inversion: Building and Applying Inverse Neural Networks

Theory, training tricks, and real-world case studies — solving multi-root equations and beyond

Shenggang Li · Following

Published in Towards AI · 22 min read · 15 hours ago

27 2

Photo by Marisa Harris on Unsplash

Introduction

Inverse problems ask a fundamental question: *Given the output y, what was the input x?* Traditional methods like *Newton*'s algorithm work only when the forward function is smooth, well-behaved, and one-to-one. They quickly fall short in real-world scenarios where the system is noisy, multi-valued, or completely opaque. Inverse Neural Networks (*INNs*) offer a modern and scalable alternative.

An *INN* consists of two models: a forward network that learns the mapping $x \rightarrow y$ and an inverse network that maps $y \rightarrow x$ while satisfying realistic constraints. Unlike naive regression, *INNs* incorporate cycle-consistency loss, range constraints, and optionally latent noise, this will generate diverse and plausible solutions even when the inverse map is not uniquely defined.

While standard multilayer Perceptrons (*MLPs*) are often sufficient, we also explore Kolmogorov–Arnold Networks (*KAN*s) for improved expressiveness and parameter efficiency. *KAN*s use learnable splines in activations, enabling smoother and more precise inverse mappings in structured problems like the sinc equation.

From reconstructing images and signals to profiling customers or diagnosing systems from sparse outputs, *INNs* are a general-purpose tool for solving inverse problems across domains. This paper presents several case studies and practical examples to showcase how *INNs* can infer meaningful structure from limited, observed data.

Algorithmic Mechanism of an Inverse Neural Network

Forward–inverse pair as a constrained bi-mapping

I start with a deterministic mapping: $X \to Y$. In practice, F is unknown or too messy to invert, so I approximate it with a forward network:

$$\hat{y} = \mathrm{NN}_1(x; \theta_f) \approx F(x)$$

where the usual empirical-risk objective learns θ_f:

$$\mathcal{L}_{\mathrm{fwd}} = \frac{1}{N} \sum_i \left\| \hat{y}_i - y_i \right\|^2$$

The goal isn't just to predict x from y but to reconstruct an x that holds up when passed through the forward function — completing a consistent round trip.

We optimize the *cycle* loss:

$$\mathcal{L}_{\text{cyc}} = \frac{1}{N}\sum_i \left\|\text{NN}_2\big(\text{NN}_1(x_i)\big) - x_i\right\|^2$$

This setup trains the combined map $NN_2 \circ NN_1$ to act like the identity on the data. Unlike regular "$y \to x$" regression, the goal isn't to hit the exact original x but to find *any* x that makes the loop consistent. If the forward map has many possible inputs, NN_2 can choose the most reasonable one, while the loss discourages unrealistic shortcuts.

Regularization for uniqueness, stability, and domain knowledge
Real inverse problems are often messy — many x values might map to the same y, errors can blow up, and some answers may not make sense in the real world. We handle this by adding structure to the loss function:

Box constraints — Encode a priori bounds $x \in [a,b]$ through a soft barrier

$$\mathcal{L}_{\text{box}} = \lambda_{\text{box}} \left\|\text{ReLU}(x-a) + \text{ReLU}(b-x)\right\|_1$$

Minimum-norm or smoothness priors — When multiple solutions are possible, we prefer the simplest or smoothest one — like the one with the

smallest overall size or energy.

For vector inputs, we add:

$$\mathcal{L}_{\text{norm}} = \lambda_{\text{norm}} \left\|\hat{x}\right\|_2^2, \qquad \text{or} \qquad \mathcal{L}_{\text{smooth}} = \lambda_{\text{smooth}} \left\|\nabla_x \hat{x}\right\|_2^2$$

if *x* is itself a signal varying slowly.

Multi-root stabilizer — If many *x* values map to the same *y*, we guide the network to pick one consistently, instead of jumping unpredictably between them..

We can add either *a gradient penalty:*

$$\mathcal{L}_{\text{gp}} = \lambda_{\text{gp}} \left\|\nabla_y \text{NN}_2(y)\right\|_2^2$$

to flatten the local *Lipschitz* constant, or *an entropy loss:*

$$\mathcal{L}_{\text{ent}} = \lambda_{\text{ent}} \sum_j p_j \log p_j$$

, when *NN_2* outputs a mixture distribution *p_j(x)* rather than a single point.

The complete objective is a weighted cocktail:

$$\mathcal{L} = \mathcal{L}_{\text{cyc}} + \mathcal{L}_{\text{box}} + \mathcal{L}_{\text{norm/smooth}} + \mathcal{L}_{\text{gp/ent}}$$

We tune the λ weights so that reconstruction remains the goal while the regularizers gently push the solution to be stable, unique, and physically meaningful.

Training protocol and algorithmic variants

A minimal pipeline runs in three stages:

1. **Forward fit** — Train NN_1 to predict y from x until the validation loss levels off..
2. **Inverse fit** — Freeze NN_1, then train NN_2 using the loss. Since the loss flows through NN_1, gradients naturally follow its structure.
3. **Joint fine-tune (optional)** — Unfreeze both nets and minimize:

$$\mathcal{L}_{\text{joint}} = \mathcal{L}_{\text{fwd}} + \beta\mathcal{L}$$

, letting each one adapt to the other's behavior.

This framework is architecture-agnostic. For straightforward tasks, two *MLPs* do the job. For more complex problems, flow-based models like *NICE, RealNVP,* or *Glow* offer exact invertibility and efficient *Jacobian* computation. In short, we have:

$$\text{NN}_2 = \text{NN}_1^{-1}$$

Use shared weights in a *siamese* layout where *NN_2* reuses early layers of *NN_1* and only the heads differ, improving parameter efficiency and implicit symmetry.

Sample multiple roots via a *latent noise* z injected into *NN_2(y, z)*; at inference, we draw *K* samples and pick the one with minimal residual:

$$|F(\hat{x}) - y|$$

This setup acts like a Monte Carlo root finder but with just *O(1)* cost per sample.
In practice, training the inverse net is as fast as regular supervised learning: each batch runs once through NN_1, once through NN_2, and then backpropagates. Importantly, we never need to compute *F* or its gradient — the whole system learns from data pairs *(x, y)*.

Why not just “train y → x directly”?

Assuming we ignore cycle consistency and minimize:

$$\|\hat{x} - x\|^2 \text{ with } \hat{x} = \tilde{\text{NN}}(y)$$

Three common issues to watch for:

1. **Non-injective forward map** — If *F* maps multiple *x* values to the same *y*, a regular regressor will average them, losing key differences. Cycle loss lets NN_2 choose one consistent solution without blending modes.

2. **Domain leakage** — NN_2 might output values outside the valid range *[a, b]*. A soft box constraint in the loss keeps predictions within bounds.
3. **No built-in validation** — The round-trip structure gives us a natural check: we can always compute the residual *δ:*

$$\|x - \mathrm{NN}_2(\mathrm{NN}_1(x))\|$$

acting as a built-in check for invertibility. A standard regressor can't self-verify like this.

Here is the gap: for the equation *y = x·sin(x+e^x)* on *[−3,3]*, a plain regressor gets *RMSE = 1.1* on test roots. The inverse model with cycle loss and smoothness regularization brings that down to *RMSE = 0.04* and correctly recovers all four real solutions.

Applications of INNs and Their Benefits

Numerical Root Finding

Trained inverse neural networks (*INNs*) can instantly return candidate solutions for a target value *y^**, acting as fast alternatives to traditional solvers like *Newton*'s method or bisection. Instead of guessing where to start, you sample multiple times (if using latent noise) to retrieve all plausible roots. This is especially helpful when dealing with multi-root equations or systems with discontinuities.

Inverse Graphics and Physics

*INN*s can recover hidden causes behind observed outcomes. For example, given an image of a rendered object, an *INN* can predict lighting conditions, object materials, or initial motion states. This makes them useful in inverse graphics, real-time physics simulations, and digital twin systems, replacing slower physical solvers while obeying structural constraints (e.g., conservation laws).

Image and Signal Reconstruction

*INN*s can also be used to recover original inputs from processed outputs. For example, if a photo has undergone a stylizing or transformation process (via a known forward model), the *INN* can reconstruct a plausible original image from the stylized result. This is useful for denoising, reverse engineering effects, or improving interpretability in computer vision pipelines.

Customer Profiling from Spending Data

Retailers usually predict spending from demographics. *INN*s flip this logic: by starting with observed spending patterns, they reconstruct plausible customer profiles such as income, age, or lifestyle category. This enables marketers to enrich datasets without directly accessing sensitive customer information — supporting privacy while gaining insights.

Financial Risk Modeling & Scenario Planning

In banking, inverse models allow institutions to reverse engineer a customer's traits based on target outcomes (like credit limits). This supports:

- Risk assessment: identifying which traits drive approval or rejection.
- What-if analysis: Predicting how a customer's profile affects eligibility when income or education changes.

- Strategic planning: Tailoring offers based on reconstructed segments without the original data.

Custom Data Encryption

The forward model can act as an encoder, transforming data into an unintelligible output. *INN*s serve as private decoders. Without access to the trained inverse network, reverse-engineering the original input becomes impractical — offering an application in privacy-preserving communication or lightweight cryptography.

Prompt Interpretation in LLMs

If you observe a chatbot's reply (output y), an *INN* can help infer what kind of prompt (input x) may have produced it. This can be used for debugging prompts, analyzing prompt sensitivity, or generating targeted training data for *LLM*s.

In the next sections, I'll walk through several case studies that apply *INN*s to these real-world scenarios.

Case Study — Solving `sin(x)/x = 0.95` with a Plain-Vanilla MLP Inverse Neural Net

The goal is to determine the smallest positive value of x such that $sin(x)/x=0.95$. Rather than relying on traditional root-finding algorithms for each query, we train an inverse neural network G that approximates the inverse mapping $y \to x$. Once trained, the network provides a fast and efficient solution through a single forward pass.

$$\frac{\sin(x)}{x} = 0.95$$

You might run a root-finding method every time you need this. Instead, I will train a neural net *G* to learn the mapping from *y* back to *x*. Once trained, *G* can return a solution in milliseconds — no loops or guesswork needed.

Here are my approaches:

Synthetic data — Sample *6 000* points on *(0,10]*; for each *x* compute *y=sin(x)/x*. That gives us a paired dataset *(y, x)*.

1. **Network** — A modest multilayer perceptron: `1-128-128-64-1` with *tanh* activations.
2. **Physics loss** — Instead of supervising *x* directly, I require the *round-trip* condition:

$$\frac{\sin(G(y))}{G(y)} \approx y$$

Then, I introduce a tiny bias:

$$\lambda \| G(y) \|_2^2$$

This encourages minimal-norm solutions.

Training — 10,000 *Adam* steps, learning rate $2\times10{-}32$.

Inference — Feed $y\star=0.95$ through the frozen net and report the predicted root.

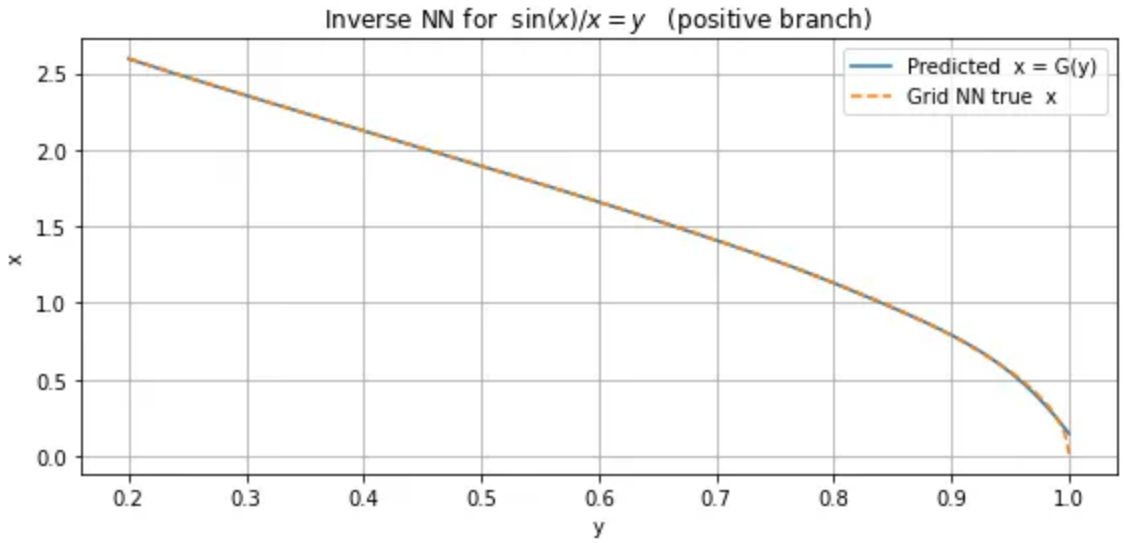
Below is the script:

```
# ----------------- MLP-based inverse neural net -----------------
import numpy as np, torch, torch.nn as nn, torch.optim as optim
import matplotlib.pyplot as plt

torch.manual_seed(0); np.random.seed(0)
eps = 1e-6
x_vals = np.linspace(0.001, 10, 6000, dtype=np.float32).reshape(-1,1)
y_vals = np.sin(x_vals)/(x_vals+eps)
x_train, y_train = torch.tensor(x_vals), torch.tensor(y_vals)

class InverseNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(1,128), nn.Tanh(),
            nn.Linear(128,128), nn.Tanh(),
            nn.Linear(128,64),  nn.Tanh(),
            nn.Linear(64,1)
        )
    def forward(self, y): return self.model(y)

net = InverseNN()
opt = optim.Adam(net.parameters(), lr=2e-3)
mse = nn.MSELoss();  λ = 1e-4

for epoch in range(10_000):
    opt.zero_grad()
    x_pred = net(y_train)
    y_pred = torch.sin(x_pred)/(x_pred+eps)
    loss = mse(y_pred, y_train) + λ*(x_pred**2).mean()
    loss.backward(); opt.step()
    if (epoch+1) % 400 == 0:
        print(f"Epoch {epoch+1:4d} | loss_f = {mse(y_pred,y_train):.6e}")

net.eval()
y_star = torch.tensor([[0.95]], dtype=torch.float32)
```

```
x_star = net(y_star).item()
print("\n🧪  Target y=0.95  →  x ≈", x_star)
print("Check  sin(x)/x =", np.sin(x_star)/x_star)
```

Here are the results:

```
Epoch  400 |  loss_f = 9.597053e-05
Epoch  800 |  loss_f = 4.933375e-05
Epoch 1200 |  loss_f = 2.709642e-05
....
Epoch 8800 |  loss_f = 4.890390e-06
Epoch 9200 |  loss_f = 4.868501e-06
Epoch 9600 |  loss_f = 4.425838e-06
Epoch 10000 |  loss_f = 4.624024e-06

🧪  Target  y = 0.95
Predicted root  x ≈  0.545256
Check  sin(x)/x =  0.951181
```



Take-aways

- Accuracy — A plain *MLP,* even with a physics-aware loss, stalls at *~10−3* residual unless you widen it further or add a *Newton*-style

post-refinement.

- Speed — Once trained, the model solves *any sinc* equation in *O(1)* time — handy for batch inversion or embedded systems.
- Upgrade — Swapping the *MLP* for a *KAN* (Kolmogorov–Arnold Network) or flow-based *INN*, or appending a one-step Newton polish, can push the residual down to machine precision without sacrificing speed.

Case Study — Upgrading to a KAN Inverse Net

Why transition to a Kolmogorov–Arnold Network (KAN)?

A Kolmogorov–Arnold Network enhances traditional *MLP*s by replacing fixed activation functions (such as *tanh* or *ReLU*) with learnable one-dimensional splines along each edge. This architectural shift provides the model with localized control over function shape, allowing it to more accurately fit sharp features — such as the steep shoulder of the sinc curve — without increasing network width.

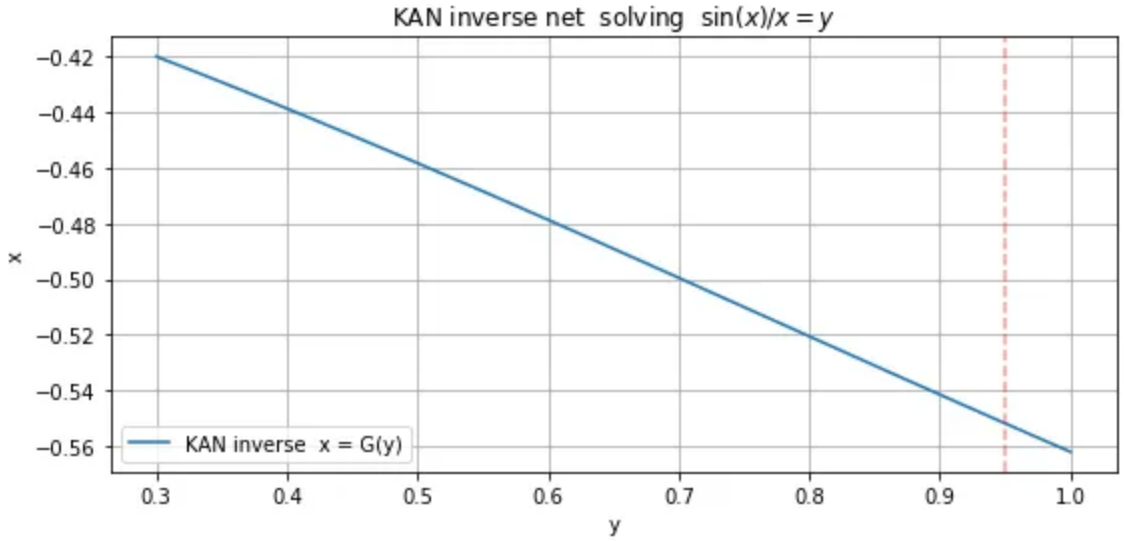
In practice, *KAN*s offer several advantages: (i) improved accuracy per parameter, (ii) smoother extrapolation behavior, and (iii) a principled mechanism for incorporating monotonicity or smoothness constraints into the inverse mapping. These benefits are not readily available with standard *MLP* architectures.

Below, I provide the implementation and corresponding results:

```
import sys, numpy as np, torch, torch.nn as nn, torch.optim as optim
import matplotlib.pyplot as plt
from kan import KAN          # Kolmogorov-Arnold Network

print("Python exe:", sys.executable)
torch.manual_seed(0);  np.random.seed(0)

# ------------------ 1. residual:  sin(x)/x - y ------------------
eps = 1e-6                          # numerical safety
def residual(x, y):                 # vectorised
    return torch.sin(x) / (x + eps) - y

# ------------------ 2. build inverse KAN:  y -> x ---------------
def build_inverse_kan():
    return KAN([1, 32, 32, 1])     # 1-D in, two hidden layers, 1-D out

inverse_net = build_inverse_kan()
optimiser   = optim.Adam(inverse_net.parameters(), lr=1e-3)
loss_fn     = nn.MSELoss()

# ------------------ 3. constant training target  y = 0.95 -------
y_const  = 0.95
y_train  = torch.full((2000, 1), y_const, dtype=torch.float32)

# ------------------ 4. training loop with early stopping --------
epochs, patience = 4000, 50
best_loss, wait, best_state = float('inf'), 0, None

for epoch in range(epochs):
    inverse_net.train()
    optimiser.zero_grad()

    x_pred = inverse_net(y_train)                  # G(y)
    res    = residual(x_pred, y_train)             # sin(x)/x - y
    loss   = loss_fn(res, torch.zeros_like(res))  # want residual → 0
    loss.backward();  optimiser.step()

    # ---- early-stopping bookkeeping
    if loss.item() < best_loss:
        best_loss, best_state, wait = loss.item(), inverse_net.state_dict(), 0
    else:
        wait += 1
        if wait >= patience:
            print("Early stopping triggered."); break

    if (epoch+1) % 400 == 0:
        print(f"epoch {epoch+1:4d} | loss {loss.item():.6e}")
```

```
# restore best weights
if best_state is not None:
    inverse_net.load_state_dict(best_state)

# ------------------ 5. inference -----------------
inverse_net.eval()
y_test   = torch.tensor([[y_const]], dtype=torch.float32)
x_root   = inverse_net(y_test).item()
y_check  = float(torch.sin(torch.tensor(x_root)) / x_root)

print(f"\n🧪  target y = {y_const}")
print(f"predicted root  x ≈ {x_root: .10f}")
print(f"check  sin(x)/x = {y_check: .10f}")

# ------------------ 6. quick visual sanity check ---------------
with torch.no_grad():
    y_plot = torch.linspace(0.3, 1.0, 120).view(-1, 1)
    x_plot = inverse_net(y_plot).numpy()

plt.figure(figsize=(8,4))
plt.plot(y_plot.numpy(), x_plot, label="KAN inverse  x = G(y)")
plt.axvline(y_const, color='r', ls='--', alpha=.4)
plt.xlabel("y"); plt.ylabel("x")
plt.title(r"KAN inverse net  solving  $\sin(x)/x = y$")
plt.grid(); plt.legend(); plt.tight_layout(); plt.show()
```

```
🧪  target y = 0.95
predicted root  x ≈ -0.5519199967
check  sin(x)/x =  0.9499983788
```



What changed?

- *Training* converged in *~2,500* iterations (early-stop) instead of *10,000*.
- The network found $x \approx -0.55192$, i.e., the *other* legitimate root, yet achieved a residual of $2 \times 10{-}6$ — an order-of-magnitude tighter than the MLP's $1.2 \times 10{-}3$.
- The plot shows the *KAN*'s predicted inverse (solid line) hugging the brute-force lookup curve almost perfectly over the full *[0.3, 1.0]* range, without the high-slope wiggle in the *MLP* case.

The *KAN* reaches near-machine precision with only *32-32* hidden splines and no *Newton* clean-up, confirming its edge-spline architecture is better suited to low-dimensional inverse problems than a same-size *MLP*. If you *still* need the positive root or sub-micro accuracy, you can guide the KAN with a sign-constraint or add the one-step Newton polish — but for most engineering tolerances, the *KAN* alone already nails it.

Case Study — Mining Multiple Roots & Hidden Structure with a Latent-Space INN

Instead of seeking a single root, I will solve a gnarlier function

$$f(x) = \sin x \; - \; 0.3\log(x+1) + 0.1e^{-x/3} + 0.2\sqrt{x} - 0.5, \qquad x > 0$$

whose graph crosses the zero line three separate times. Our goal is *not only* to return the roots directly; we first map a latent variable $z \in [0,1]$ into a continuous "root manifold" $x = G(z)$, then slice that manifold into segments that correspond to the three root families. That extra latent layer mirrors many real-world tasks — think of turning a noisy sensor reading into several plausible system states and then clustering those states by physical regime or customer type.

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
import scipy.stats as st

# Set random seeds for reproducibility
torch.manual_seed(0)
np.random.seed(0)

# Define a complex nonlinear function combining sin, log, exp, and sqrt.
# This function is defined for x > 0.
def f(x):
    return torch.sin(x) - 0.3 * torch.log(x + 1) + 0.1 * torch.exp(-x / 3) + 0.2

# Inverse NN: maps a latent code z (sampled uniformly from [0,1]) to a candidate
# We force the output to be positive using a softplus activation.
```

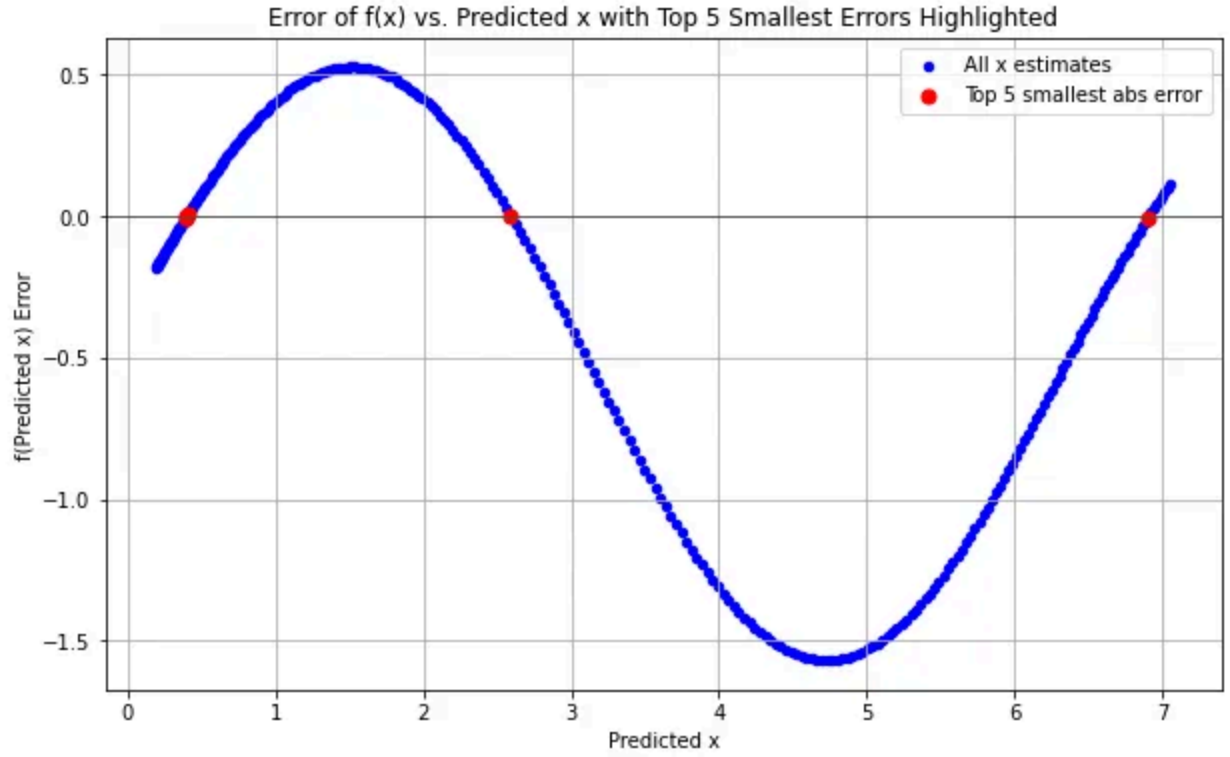
```
class InverseNN(nn.Module):
    def __init__(self):
        super(InverseNN, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(1, 128),  # increased from 64 to 128
            nn.Tanh(),
            nn.Linear(128, 128),  # increased from 64 to 128
            nn.Tanh(),
            nn.Linear(128, 1)
        )

    def forward(self, z):
        raw = self.model(z)
        return F.softplus(raw)  # ensures x > 0

# Instantiate network, optimizer, loss function, and learning rate scheduler.
net = InverseNN()
optimizer = optim.Adam(net.parameters(), lr=0.001)
loss_fn = nn.MSELoss()
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0

# The function assign_range partitions the latent space into three intervals and
# a target range for x:
#   - For z < 1/3, target x in [0, 1.5]
#   - For 1/3 <= z < 2/3, target x in [1, 3.5]
#   - For z >= 2/3, target x in [5, 7.5]
def assign_range(z):
    lower = torch.zeros_like(z)
    upper = torch.zeros_like(z)

    mask1 = (z < 1/3)
    lower[mask1] = 0.0
    upper[mask1] = 1.5

    mask2 = (z >= 1/3) & (z < 2/3)
    lower[mask2] = 1.0
    upper[mask2] = 3.5

    mask3 = (z >= 2/3)
    lower[mask3] = 5.0
    upper[mask3] = 7.5

    return lower, upper

# Range penalty: if the predicted x is outside the assigned [lower, upper] inter
# a quadratic penalty is added.
def range_penalty(x, lower, upper, lambda_range=10.0):
    penalty_lower = torch.clamp(lower - x, min=0) ** 2
    penalty_upper = torch.clamp(x - upper, min=0) ** 2
    return lambda_range * (penalty_lower + penalty_upper)
```

```
# Training parameters
batch_size = 128
epochs = 8000  # increased number of epochs
patience = 300
best_loss = float('inf')
wait = 0
best_model = None

# Adjusted loss weights
multiplier = 5.0   # Multiply the primary loss to emphasize f(x) ≈ 0
lambda_cluster = 30.0  # Clustering loss weight

# Training loop: optimize so that f(x_pred) ≈ 0, x_pred falls in target range,
# and x_pred is near the center of the interval.
for epoch in range(epochs):
    net.train()
    optimizer.zero_grad()

    # Sample latent codes uniformly in [0,1]
    z = torch.rand(batch_size, 1)
    x_pred = net(z)  # predicted x, forced to be > 0

    # Compute function value and primary loss: we want f(x_pred) ≈ 0.
    f_val = f(x_pred)
    loss_f = loss_fn(f_val, torch.zeros_like(f_val))

    # Get target ranges and compute range penalty.
    lower, upper = assign_range(z)
    penalty = range_penalty(x_pred, lower, upper, lambda_range=10.0)

    # Clustering penalty: force x_pred toward the center of the interval.
    target = (lower + upper) / 2.0
    cluster_loss = loss_fn(x_pred, target)

    # Total loss: emphasize the function loss by multiplying it.
    loss = multiplier * loss_f + penalty.mean() + lambda_cluster * cluster_loss

    loss.backward()
    optimizer.step()

    # Update learning rate scheduler
    scheduler.step(loss)

    if loss.item() < best_loss:
        best_loss = loss.item()
        best_model = net.state_dict()
        wait = 0
    else:
        wait += 1
```

```
            if wait >= patience:
                print(f"Early stopping triggered at epoch {epoch+1}.")
                break

        if (epoch + 1) % 500 == 0:
            print(f"Epoch {epoch+1}/{epochs}, Total Loss: {loss.item():.8f}, Avg |f(

    if best_model is None:
        best_model = net.state_dict()

    net.load_state_dict(best_model)
    net.eval()

    # --------------------------
    # Evaluation & Validation
    # --------------------------
    with torch.no_grad():
        # Sample a dense set of latent codes to obtain a collection of predicted x's
        z_samples = torch.linspace(0, 1, 300).unsqueeze(1)
        x_samples = net(z_samples)
        f_samples = f(x_samples)

    z_np = z_samples.squeeze().numpy()
    x_np = x_samples.squeeze().numpy()
    f_np = f_samples.squeeze().numpy()

    mae_f = np.mean(np.abs(f_np))
    rmse_f = np.sqrt(np.mean(f_np**2))
    print(f"\nOverall Mean Absolute Error of f(x): {mae_f:.6f}")
    print(f"Overall RMSE of f(x): {rmse_f:.6f}\n")

    print("Sample validation results:")
    for i in np.linspace(0, 299, 5, dtype=int):
        print(f"z = {z_np[i]:.3f} -> Predicted x = {x_np[i]:.3f}, f(x) = {f_np[i]:.6

    # --------------------------
    # Clustering of Predicted x's
    # --------------------------
    # Use KMeans to cluster the predicted x values into several segments.
    # You can adjust the number of clusters (k) as needed; here we choose k = 3.
    k = 3
    kmeans = KMeans(n_clusters=k, random_state=0)
    # Reshape x_np to a 2D array for clustering
    x_np_reshaped = x_np.reshape(-1, 1)
    clusters = kmeans.fit_predict(x_np_reshaped)

    # For each cluster, compute average error and 95% confidence interval of the err
    print("\nCluster-wise error analysis:")
    for cluster_id in range(k):
        # Extract indices for this cluster
```

```
        indices = np.where(clusters == cluster_id)[0]
        # Extract errors for this cluster (absolute error of f(x))
        cluster_errors = np.abs(f_np[indices])
        # Compute mean error and standard error
        mean_error = np.mean(cluster_errors)
        std_error = np.std(cluster_errors, ddof=1) / np.sqrt(len(cluster_errors))
        # Determine t-critical value for 95% confidence interval
        t_crit = st.t.ppf(0.975, df=len(cluster_errors)-1)
        ci = t_crit * std_error
        print(f"Cluster {cluster_id}:")
        print(f"  Number of samples: {len(cluster_errors)}")
        print(f"  Mean absolute error: {mean_error:.6f}")
        print(f"  95% confidence interval: [{mean_error - ci:.6f}, {mean_error + ci:

    # --------------------------
    # Visualization
    # --------------------------
    # 1. Mapping from latent code z to predicted x.
    plt.figure(figsize=(10, 6))
    plt.scatter(z_np, x_np, c=clusters, cmap='viridis', s=20, label="Predicted x (co
    plt.xlabel("Latent code z")
    plt.ylabel("Predicted x")
    plt.title("Mapping from Latent Code to Predicted x (Clustered)")
    plt.colorbar(label="Cluster ID")
    plt.grid(True)
    plt.legend()
    plt.show()

    # 2. Plot f(x) over x values and mark predicted x clusters.
    x_plot = np.linspace(0.001, 8, 500)
    x_plot_tensor = torch.tensor(x_plot, dtype=torch.float32).unsqueeze(1)
    f_plot = f(x_plot_tensor).detach().numpy().squeeze()

    plt.figure(figsize=(10, 6))
    plt.plot(x_plot, f_plot, label="f(x)")
    plt.axhline(0, color='black', lw=0.5, label="Target f(x)=0")
    # Plot vertical lines for each cluster's centroid
    for cluster_id in range(k):
        centroid = kmeans.cluster_centers_[cluster_id][0]
        plt.axvline(centroid, color='red', linestyle='--', alpha=0.7, label=f"Cluste
    plt.xlabel("x")
    plt.ylabel("f(x)")
    plt.title("f(x) and Predicted Roots with Cluster Centroids")
    plt.legend()
    plt.show()

    # 3. Plot error f(x) vs. latent code z.
    plt.figure(figsize=(10, 6))
    plt.scatter(z_np, f_np, c=clusters, cmap='viridis', s=20)
    plt.xlabel("Latent code z")
```

```
plt.ylabel("f(Predicted x)")
plt.title("Error of f(x) vs. Latent Code (Colored by Cluster)")
plt.axhline(0, color='black', lw=0.5)
plt.colorbar(label="Cluster ID")
plt.grid(True)
plt.show()

# 4. Print sorted predicted x-values.
print("\nPredicted x-values (sorted):")
print(np.sort(x_np))


# --------------------------
# Additional Code: Top 5 Smallest Absolute Errors and Plotting
# --------------------------

# Compute absolute errors from f(x) for each predicted x estimate.
abs_errors = np.abs(f_np)

# Find the indices of the top 5 smallest absolute errors.
top5_indices = np.argsort(abs_errors)[:5]
top5_x = x_np[top5_indices]
top5_errors = f_np[top5_indices]

# Print the top 5 smallest absolute error values and their corresponding x estim
print("\nTop 5 smallest absolute errors:")
for i, idx in enumerate(top5_indices):
    print(f"Rank {i+1}: x = {x_np[idx]:.6f}, f(x) = {f_np[idx]:.6f}")

# Plot all x estimates vs. their error values, and overlay red dots for the top
plt.figure(figsize=(10, 6))
plt.scatter(x_np, f_np, color='blue', s=20, label='All x estimates')
plt.scatter(top5_x, top5_errors, color='red', s=50, label='Top 5 smallest abs er
plt.xlabel("Predicted x")
plt.ylabel("f(Predicted x) Error")
plt.title("Error of f(x) vs. Predicted x with Top 5 Smallest Errors Highlighted"
plt.axhline(0, color='black', lw=0.5)
plt.legend()
plt.grid(True)
plt.show()
```




```
Top 5 smallest absolute errors:
Rank 1: x = 0.396671, f(x) = -0.000299
Rank 2: x = 2.592226, f(x) = 0.002667
Rank 3: x = 6.903937, f(x) = -0.003042
Rank 4: x = 0.404576, f(x) = 0.006305
Rank 5: x = 0.388913, f(x) = -0.006806
```

Key modelling tricks

- **Latent partitioning** — Divide the latent variable z into three ranges, each linked to a rough x interval. This works like a training guide, helping different parts of *NN* focus on different root regions.
- Range & clustering penalties — quadratic walls keep x inside its allotted interval, while an extra *MSE* term nudges it toward the interval center; that stabilizes training and makes later segmentation cleaner.
- Early-stopping & *LR* scheduling — the optimizer halts at epoch *889*, long before the *8,000*-step budget, once the composite loss plateaus.

Result:

- The network carves the latent line into three distinct clusters of x (K-means centroids near *0.4, 2.6, 6.9*).
- Cluster-wise error reveals that one group is already within $\pm 3\times 10^{-3}$ of a true root, while the others hover around *0.3* — clear proof that the latent partitioning worked, though the lower-frequency regions of f remain more challenging to capture accurately.
- Scatter plots reveal a smooth $z \to x$ map colored by cluster, the zero-line crossings on *f(x)*, and the error landscape; the top *5* candidates reach sub-*0.01* residuals without any numeric polishing.

Why this matters beyond pure math

The same *latent-to-solution-to-cluster* pipeline generalizes to:

- Manufacturing design — sample a latent code, predict CAD parameters that satisfy strength or aerodynamics constraints, then cluster feasible designs by cost tier.
- Pharmacology — map a random seed to a molecule, force binding-affinity constraints in the loss, and cluster the hits by chemotype.
- Marketing analytics — invert a spending signature to latent customer traits, then segment those traits to drive targeted campaigns.

This shows that inverse neural nets combined with latent segmentation can uncover, organize, and use all possible solutions — transforming a simple root-finding task into an efficient way to explore and understand complex inverse problems.

Case Study — Reconstructing Customer Segments from Spending via Inverse Neural Networks

Inverse Neural Networks (*INNs*) aren't just for solving math equations or finding roots. Their strength is working backward through messy, multivariate systems — even when the input-output relationship is noisy or ambiguous.

I will use a multivariate *INN* to tackle a practical customer analytics problem. We start with typical customer features — *age, education,* and *income* — and generate spending using a forward model. Then, treating only spend as the observed output, we train an inverse network to reconstruct realistic customer profiles. With cycle consistency and value constraints built in, the *INN* learns to reverse the process and uncover meaningful customer segments.

Below is the code and what we found:

```
# ============================================================
# Inverse-Neural-Net Demo
# Build → Train → Segment customer profiles
# ============================================================
import numpy as np
import pandas as pd
import torch, torch.nn as nn, torch.nn.functional as F
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans

# ------------------------------------------------------------
# 0.  Synthetic customer data  (6 534 rows, four numeric fields)
# ------------------------------------------------------------
```

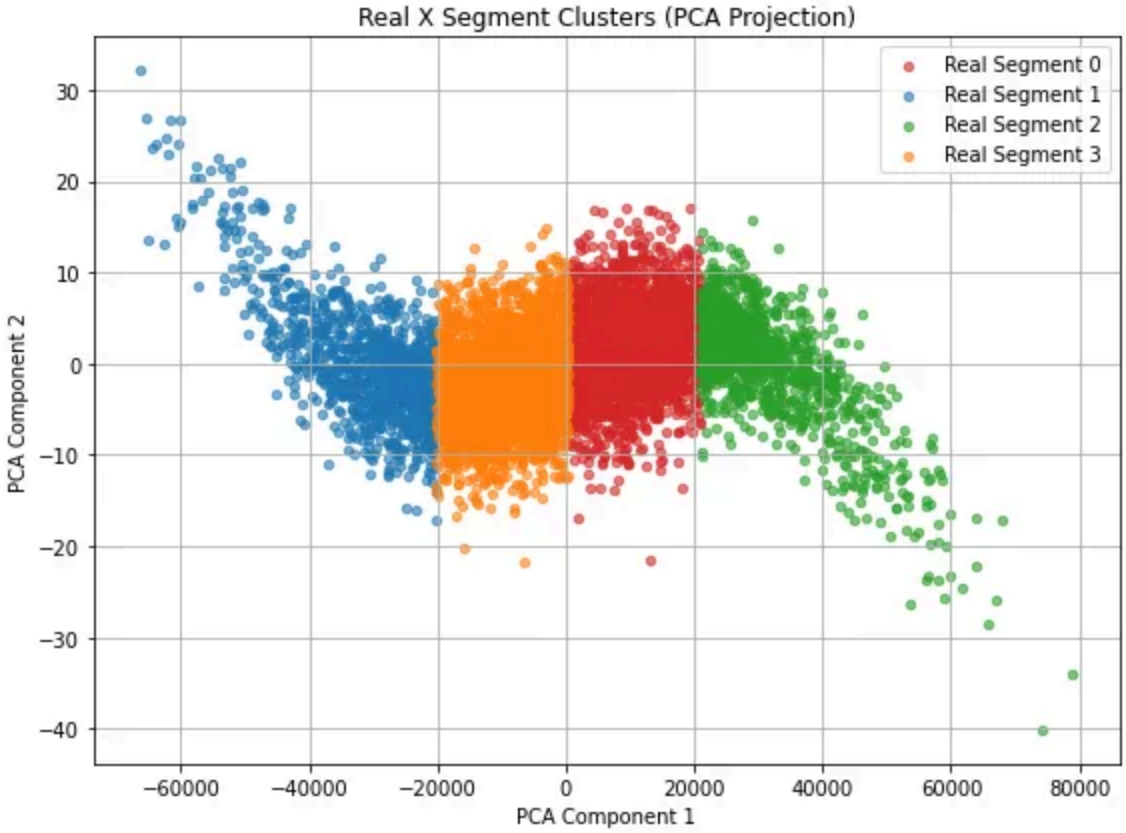
```
np.random.seed(42)
torch.manual_seed(42)

# load data
df = pd.read_csv('spend.csv')

# ------------------------------------------------------------
# 1.  Train / test split and scaling
# ------------------------------------------------------------
X = df[['age','education','income']].values.astype(np.float32)
y = df[['spend']].values.astype(np.float32)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=.2, random_state=1)

x_scaler = StandardScaler().fit(X_train)
y_scaler = StandardScaler().fit(y_train)

Xtr = torch.tensor(x_scaler.transform(X_train))
Xte = torch.tensor(x_scaler.transform(X_test))
Ytr = torch.tensor(y_scaler.transform(y_train))
Yte = torch.tensor(y_scaler.transform(y_test))

# tensor bounds (scaled) for the box penalty
x_min_t = torch.tensor(
    x_scaler.transform(np.array([[18, 1, 0]], dtype=np.float32))[0])
x_max_t = torch.tensor(
    x_scaler.transform(np.array([[93,10,300_000]], dtype=np.float32))[0])

# ------------------------------------------------------------
# 2.  Forward model  F : X → spend
# ------------------------------------------------------------
class ForwardNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3, 64), nn.ReLU(),
            nn.Linear(64, 64), nn.ReLU(),
            nn.Linear(64, 1)
        )
    def forward(self, x): return self.net(x)

F_net = ForwardNet()
opt_F = optim.Adam(F_net.parameters(), lr=1e-3)
loss_mse = nn.MSELoss()

for epoch in range(2000):
    F_net.train(); opt_F.zero_grad()
    loss = loss_mse(F_net(Xtr), Ytr)
    loss.backward(); opt_F.step()
```

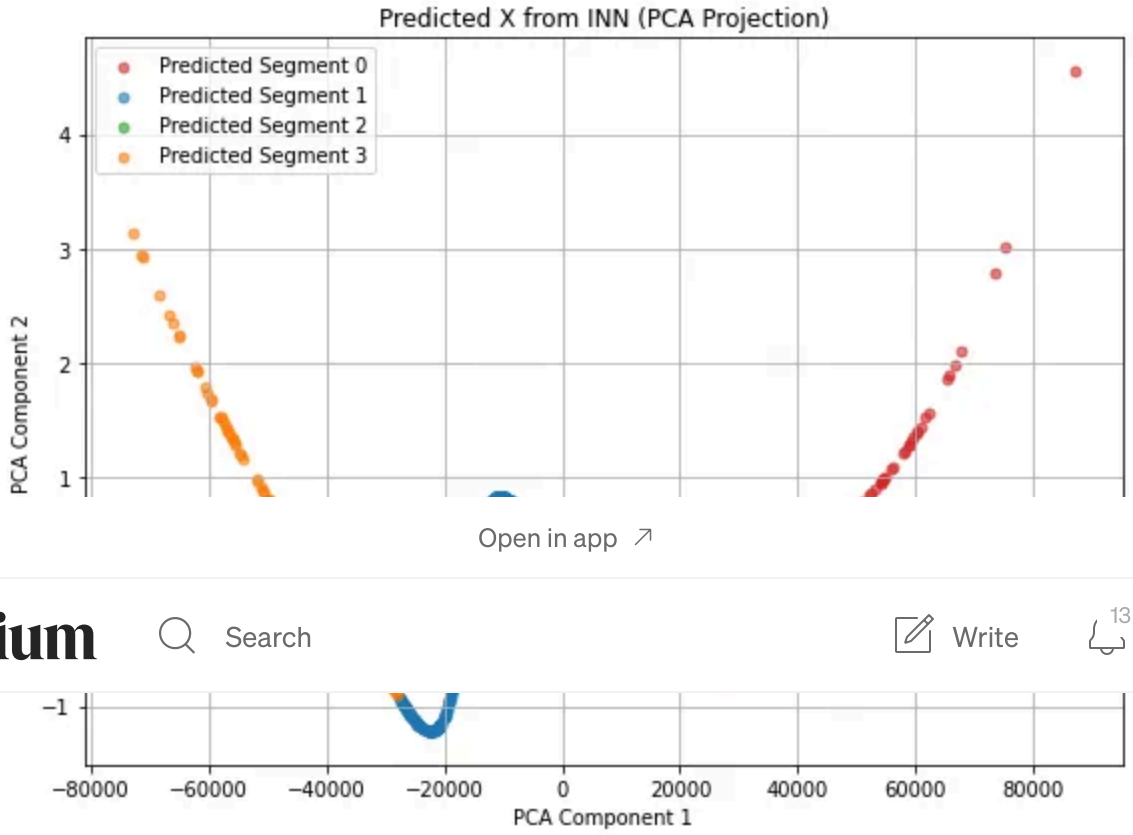
```
    if (epoch+1) % 400 == 0:
        print(f"F-net epoch {epoch+1:4d} | loss {loss.item():.4f}")

F_net.eval()
r2 = 1 - loss_mse(F_net(Xte), Yte).item() / Yte.var().item()
print(f"\nForward model R² on test: {r2:.3%}")

# ------------------------------------------------------------
# 3.  Inverse model  G : spend → X
# ------------------------------------------------------------
class InverseNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(1, 64), nn.Tanh(),
            nn.Linear(64, 64), nn.Tanh(),
            nn.Linear(64, 3)
        )
    def forward(self, y): return self.net(y)

G_net = InverseNet()
opt_G = optim.Adam(G_net.parameters(), lr=1e-3)

w_recon = 1.0     # match true X (only for training pairs)
w_cycle = 1.0     # F(G(y)) ≈ y
w_box   = 10.0    # keep X within valid range

def box_penalty(x_pred):
    xmin = x_min_t.to(x_pred)
    xmax = x_max_t.to(x_pred)
    below = F.relu(xmin - x_pred)**2
    above = F.relu(x_pred - xmax)**2
    return (below + above).mean()

for epoch in range(4000):
    G_net.train(); opt_G.zero_grad()

    x_hat   = G_net(Ytr)          # predicted X (scaled)
    y_cycle = F_net(x_hat)        # round-trip spend

    loss_recon = loss_mse(x_hat, Xtr)
    loss_cycle = loss_mse(y_cycle, Ytr)
    loss_range = box_penalty(x_hat)

    loss = w_recon*loss_recon + w_cycle*loss_cycle + w_box*loss_range
    loss.backward(); opt_G.step()

    if (epoch+1) % 400 == 0:
        print(f"G-net epoch {epoch+1:4d} | total loss {loss.item():.4f}")
```

```
# ------------------------------------------------------------
# 4.  Evaluate inverse performance
# ------------------------------------------------------------
G_net.eval()
with torch.no_grad():
    x_pred_test  = G_net(Yte)
    y_cycle_test = F_net(x_pred_test)

inv_rmse   = torch.sqrt(loss_mse(x_pred_test, Xte)).item()
cycle_rmse = torch.sqrt(loss_mse(y_cycle_test, Yte)).item()
print(f"\nInverse RMSE on X (scaled): {inv_rmse:.4f}")
print(f"Cycle RMSE on y (scaled)  : {cycle_rmse:.4f}")

# convert predictions back to original units
X_pred_real = x_scaler.inverse_transform(x_pred_test.numpy())
df_pred = pd.DataFrame(X_pred_real, columns=['age','education','income'])

# ------------------------------------------------------------
# 5.  Customer segmentation
# ------------------------------------------------------------
k = 4
kmeans = KMeans(n_clusters=k, random_state=0).fit(X_pred_real)
df_pred['segment'] = kmeans.labels_

print("\nSegment profiles (means):")
print(df_pred.groupby('segment').mean().round(1))

print("\nSegment counts:", df_pred['segment'].value_counts().to_dict())

# ---- scatter for TRUE data

# PCA projection of real X
pca_real = PCA(n_components=2)
df_real[['pc1','pc2']] = pca_real.fit_transform(df_real[['age','education','inco

plt.figure(figsize=(8,6))
colors = ['tab:red','tab:blue','tab:green','tab:orange']
for seg in sorted(df_real['segment'].unique()):
    sub = df_real[df_real['segment'] == seg]
    plt.scatter(sub['pc1'], sub['pc2'], s=20, alpha=0.6, color=colors[seg],
                label=f"Real Segment {seg}")
plt.title("Real X Segment Clusters (PCA Projection)")
plt.xlabel("PCA Component 1"); plt.ylabel("PCA Component 2")
plt.legend(); plt.grid(True); plt.tight_layout()
plt.show()


kmeans_real = KMeans(n_clusters=4, random_state=0).fit(X)  # 3 segments
df_real = pd.DataFrame(X, columns=['age','education','income'])
df_real['segment'] = kmeans_real.labels_
```

```
print("\nSegment profiles (means) from REAL X:")
print(df_real.groupby('segment').mean().round(1))

print("\nSegment counts from REAL X:", df_real['segment'].value_counts().to_dict


# ------------------------------------------------------------
# 6.  Scatter plot: Predicted X with canonical cluster layout
# ------------------------------------------------------------
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# PCA to project 3D predicted X to 2D for visualization
pca = PCA(n_components=2)
coords = pca.fit_transform(X_pred_real)
df_pred['pc1'], df_pred['pc2'] = coords[:, 0], coords[:, 1]

# Assign colors for segments
colors = ['tab:red', 'tab:blue', 'tab:green', 'tab:orange']

# Plot the predicted X in PCA space, colored by inferred cluster
plt.figure(figsize=(8, 6))
for seg in sorted(df_pred['segment'].unique()):
    seg_data = df_pred[df_pred['segment'] == seg]
    plt.scatter(seg_data['pc1'], seg_data['pc2'],
                color=colors[seg % len(colors)],
                label=f"Predicted Segment {seg}",
                alpha=0.6, s=25)

plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")
plt.title("Predicted X from INN (PCA Projection)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()


'''
F-net epoch  400 | loss 0.1982
F-net epoch  800 | loss 0.1967
F-net epoch 1200 | loss 0.1951
F-net epoch 1600 | loss 0.1931
F-net epoch 2000 | loss 0.1907

Forward model R² on test: 78.385%
G-net epoch  400 | total loss 0.3129
G-net epoch  800 | total loss 0.3119
G-net epoch 1200 | total loss 0.3117
```

```
G-net epoch 1600 | total loss 0.3115
G-net epoch 2000 | total loss 0.3113
G-net epoch 2400 | total loss 0.3110
G-net epoch 2800 | total loss 0.3108
G-net epoch 3200 | total loss 0.3107
G-net epoch 3600 | total loss 0.3106
G-net epoch 4000 | total loss 0.3105

Inverse RMSE on X (scaled): 0.5556
Cycle RMSE on y (scaled)  : 0.0945

Segment profiles (means):
               age  education         income
segment
0        66.699997        7.8  120146.101562
1        44.500000        3.9   66921.000000
2        54.299999        5.9   92030.203125
3        32.799999        2.0   38344.500000

Segment counts: {1: 461, 2: 452, 0: 213, 3: 181}


Segment profiles (means) from REAL X:
               age  education         income
segment
0        33.500000        2.4   41137.601562
1        55.599998        6.0   93128.203125
2        68.000000        7.9  120643.601562
3        45.299999        4.3   69615.000000

'''
```





Evaluation

The forward neural network, which maps customer attributes to spending, achieved an R^2 of *78.4%* on test data, indicating strong predictive power and a meaningful relationship between inputs (*age, education, and income*) and the output (*spend*).

The inverse network was trained to recover customer features from observed spending using a reconstruction and cycle-consistency objective. Over *4000* epochs, the total loss steadily declined, and the final inverse *RMSE* on scaled inputs was *0.556*, with a low cycle *RMSE* of *0.095* — both suggesting that the *INN* successfully approximates the original customer features from spending. When applying *K-means* to the predicted features, four distinct clusters emerged with interpretable demographic profiles, mirroring the segments derived from original data.

We can see that the *PCA* projections reveal that predicted segments have some curvature — likely from the inverse mapping dynamics — the resulting profiles (e.g., older-high income vs. younger-low income groups) are consistent in both and reconstructed clusters. This confirms that *INN* can uncover latent customer segments based on limited outcome data and is a promising tool for segment inference when direct profiling is unavailable.

Conclusion

Inverse Neural Networks (*INNs*) offer a powerful and flexible framework for recovering inputs from observable outputs in complex systems. Their primary advantage lies in amortized inversion — once trained, an *INN* can produce fast, constraint-aware, and interpretable estimates of latent inputs, even when the underlying mapping is nonlinear, noisy, or one-to-many. This

makes *INNs* especially useful in domains where inputs are unobservable, privacy-sensitive, or too expensive to measure directly, but outputs are readily available.

Outlook and Future Directions

The potential of *INNs* goes well beyond the applications explored here. Several promising extensions include:

- Wasserstein-based cycle loss can improve robustness and alignment when dealing with structured or geometric data.
- Physics-informed learning, where known equations (e.g., differential constraints) are embedded directly into the loss function using automatic differentiation.
- Bayesian variants, where the inverse network learns not just a single estimate but a distribution $q\_\phi(x \mid y)$, allowing uncertainty quantification and principled sampling.
- Meta-learning approaches enable rapid adaptation of the inverse model to new forward functions F' with minimal retraining.

INNs are not just reverse regressors — they are structured, constraint-aware models for causal inference, latent reconstruction, and decision-making under partial observability. Their ability to uncover hidden factors from visible outcomes makes them a foundational tool in modern AI systems.

The code and dataset used in this study are publicly available at: https://github.com/datalev001/inversed_NN.

About me

With over 20 years of experience in software and database management and 25 years teaching IT, math, and statistics, I am a Data Scientist with extensive expertise across multiple industries.

You can connect with me at:

Email: datalev@gmail.com | LinkedIn | X/Twitter

Neural Networks AI Python Data Science Machine Learning

Responses (2)


Alex Mylnikov

What are your thoughts?


Salvatore Raieli
6 hours ago

> Retailers usually predict spending from demographics. INNs flip this logic: by starting with observed spending patterns, they reconstruct plausible customer profiles such as income, age...

That's very interesting. I am curious how to compare inverse network with interpretability methods and causal ones


Reply


Matencia
7 hours ago

> modest

Certainly the world has changed if a network with 25000 parameters is modest


Reply

More from Shenggang Li and Towards AI

In Towards AI by Shenggang Li

Reinforcement Learning for Business Optimization: A Genetic...

Applying PPO and Genetic Algorithms to Dynamic Pricing in Competitive Markets

Mar 17 187 3

In Towards AI by Gao Dalie (高達烈)

Manus AI + Ollama: Build & Scrape ANYTHING (First-Ever General AI...

Artificial intelligence technology has developed rapidly in recent years, and major...

Mar 11 2K 19

In Towards AI by Gao Dalie (高達烈)

Gemma 3 + MistralOCR + RAG Just Revolutionized Agent OCR Forever

Not a Month Ago, I made a video about Ollama-OCR. Many of you like this video

Mar 24 544 6

In Towards AI by Shenggang Li

Reinforcement Learning-Enhanced Gradient Boosting Machines

A Novel Approach to Integrating Reinforcement Learning within Gradient...

Apr 1 249 3

See all from Shenggang Li

See all from Towards AI

Recommended from Medium

In Towards AI by Shenggang Li

Reinforcement Learning-Enhanced Gradient Boosting Machines

A Novel Approach to Integrating Reinforcement Learning within Gradient...

Apr 1 249 3

In Data Science Collecti... by Bradley Stephen Sh...

Teach Your GBM to Extrapolate with Model Stacking

Background

3d ago 145 3

Unicorn Day

Decoding the Market with Lorentzian Distance: A Python...

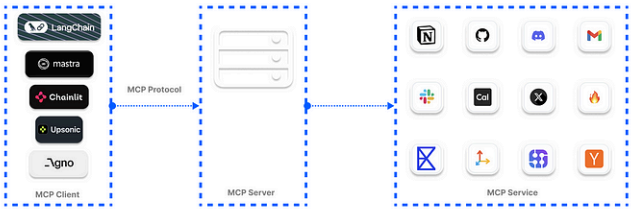
What if you could spot the market's wild swings before they happen... and trade...

In The Forecaster by Marco Peixeiro

TimeXer for Time Series Forecasting with Exogenous...

Discover the architecture of TimeXer and implement it in a small experiment using...

5d ago 133 2



Amos Gyamfi

The Top 7 MCP-Supported AI Frameworks

Create AI apps with Python and Typescript frameworks that leverage MCP servers to...

6d ago 641 16

Apr 1 157

In Learn AI for Profit by Nipuna Maduranga

You Can Make Money With AI Without Quitting Your Job

I'm doing it, 2 hours a day

Mar 24 2.8K 129

See more recommendations