The Universal Approximation Theorem Is Right. You're Using It Wrong.

19 Oct 2025

How a misunderstood theorem and a poorly formulated problem created one of machine learning's most persistent myths

---

The Myth That Won't Die

Ask any ML student about the XOR problem and you'll hear: "XOR is impossible for a single-layer perceptron. You need a hidden layer to solve it."

This has become machine learning folklore. It's in textbooks, tutorials, and countless blog posts. There's just one problem:

It's based on a misunderstanding of what XOR actually is.

What the Universal Approximation Theorem Actually Says

The Universal Approximation Theorem (UAT) states that a neural network with one hidden layer can approximate any continuous function to arbitrary accuracy.

The theorem is **correct**. It's not the problem.

The problem is how people apply it - specifically, how they formulate problems and what they think "solving XOR" means.

The Traditional XOR "Problem"

Here's how XOR is usually presented as unsolvable for single-layer networks:

Binary Classification:

- Input (0, 0) → Output 0
- Input (0, 1) → Output 1
- Input (1, 0) → Output 1
- Input (1, 1) → Output 0

"See?" they say. "The 1s are on opposite corners. You can't draw a single line to separate them. Single layer fails!"

But wait. Who said XOR has to be **binary classification**?

XOR: The Correct Formulation

XOR is a **function** that maps two binary inputs to four possible outcomes. Each (input1, input2) pair produces a unique result.

So why are we forcing it into a binary classification framework? Let's formulate it correctly as **4-class classification**:

- Input (0, 0) → Class 0
- Input (0, 1) → Class 1
- Input (1, 0) → Class 2
- Input (1, 1) → Class 3

Each input combination is unique. Each output is unique. This is the **natural** representation of the XOR truth table.

Let's Solve It (Without a Hidden Layer)

Here's a single-layer perceptron - no hidden layer, just input → output with softmax:

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

torch.manual_seed(42)

class SingleLayerClassifier(nn.Module):
    def __init__(self):
        super(SingleLayerClassifier, self).__init__()
        self.output = nn.Linear(2, 4)  # 2 inputs → 4 outputs directly
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.output(x)
        x = self.softmax(x)
        return x

# XOR truth table: 4 unique input-output pairs
X = torch.tensor([[0, 0],
                  [0, 1],
                  [1, 0],
                  [1, 1]], dtype=torch.float32)

y = torch.tensor([0, 1, 2, 3], dtype=torch.long)
```

```
model = SingleLayerClassifier()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Train
print("Training single-layer perceptron on XOR...")
epochs = 5000
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()

    logits = model.output(X)
    loss = criterion(logits, y)

    loss.backward()
    optimizer.step()

    if (epoch + 1) % 1000 == 0:
        print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')

# Evaluate
model.eval()
with torch.no_grad():
    logits = model.output(X)
    probabilities = model.softmax(logits)
    _, predicted = torch.max(probabilities, 1)

    print("\n" + "="*60)
    print("RESULTS")
    print("="*60)
    for i in range(len(X)):
        print(f"Input: {X[i].numpy()} → True: Class {y[i].item()} | "
              f"Predicted: Class {predicted[i].item()}")

    accuracy = (predicted == y).sum().item() / len(y)
    print(f"\nAccuracy: {accuracy * 100:.0f}%")
    print(f"Final Loss: {loss.item():.4f}")
    print("="*60)
```

The Results

```
Training single-layer perceptron on XOR...
Epoch [1000/5000], Loss: 0.0498
Epoch [2000/5000], Loss: 0.0148
Epoch [3000/5000], Loss: 0.0066
Epoch [4000/5000], Loss: 0.0034
Epoch [5000/5000], Loss: 0.0019

============================================================
RESULTS
```

```
============================================================
Input: [0. 0.] → True: Class 0 | Predicted: Class 0
Input: [0. 1.] → True: Class 1 | Predicted: Class 1
Input: [1. 0.] → True: Class 2 | Predicted: Class 2
Input: [1. 1.] → True: Class 3 | Predicted: Class 3

Accuracy: 100%
Final Loss: 0.0019
============================================================
```

100% accuracy. Zero errors. Single layer. No hidden units.

XOR: Solved.

What Just Happened?

The single-layer perceptron learned the XOR truth table perfectly. How is this possible when everyone says it can't be done?

Because the "XOR problem" was artificially constrained.

The Traditional Formulation Is Broken

When you force XOR into binary classification:

- You're saying outputs 1 and 3 must share the same label
- You're saying outputs 0 and 2 must share the same label
- You're **imposing** a constraint that makes the problem non-linearly separable

But XOR doesn't inherently require this! The truth table has four distinct rows with four distinct outcomes. By forcing two outcomes to share labels, you're creating an artificial problem.

The Correct Formulation Just Works

When you represent XOR naturally as 4-class classification:

- Each input pair is unique
- Each output class is unique
- The four points in 2D space can be separated with linear decision boundaries
- A single layer is **sufficient**

This isn't cheating. This is the **correct** way to represent a function with four distinct input-output mappings.

Why Does the Myth Persist?

The "XOR requires hidden layers" story comes from Minsky and Papert's 1969 book *Perceptrons*, which showed that single-layer networks can't solve certain problems. This was historically important and correct.

But here's what got lost:

1. **They were talking about perceptrons with no activation functions** (linear threshold units)
2. **They used binary classification** formulation
3. **Modern neural networks with softmax are fundamentally different**

The myth persists because:

- It makes a neat teaching example
- It's in all the textbooks now
- Nobody questions the problem formulation
- It "proves" why we need deep learning

But it's based on an **arbitrary constraint** (binary outputs) that doesn't reflect the actual XOR function.

What About "Arbitrarily Small Error"?

Notice the final loss: **0.0019**. Not zero. Why?

This is where the **real** limitations appear - not in the UAT, but in **implementation**:

Floating-Point Precision

To get a softmax output of exactly `[1.0, 0.0, 0.0, 0.0]`, you'd need:

`softmax([∞, -∞, -∞, -∞]) = [1.0, 0.0, 0.0, 0.0]`

But computers use finite precision (float32/float64). The best you can do is:

`softmax([10.5, -5.2, -5.3, -8.1]) ≈ [0.998, 0.001, 0.001, 0.000]`

This is not a theoretical limitation. The UAT doesn't guarantee infinite precision - it operates in the realm of real numbers. The 0.0019 error is a **computational artifact**, not a violation of the theorem.

Even on this trivially simple problem with:

- ✓ Perfect model capacity
- ✓ Complete, noise-free data
- ✓ Successful optimization
- ✓ Correct problem formulation

We hit the **hardware limit**: finite-precision arithmetic.

The Real Barriers to Arbitrarily Small Error

When you can't reach arbitrarily small error, it's not because UAT is wrong. It's because:

1. Computational Limits

- **Floating-point precision** (as we just saw)
- **Numerical stability** (softmax, gradients)
- **Finite optimization steps** (convergence criteria)

2. Optimization Challenges

- **Local minima** in the loss landscape
- **Poor initialization**
- **Vanishing/exploding gradients**

These aren't theorem failures - they're **implementation challenges**.

3. Problem Formulation

- **Wrong architecture** for the task
- **Inappropriate loss function**
- **Artificial constraints** (like forcing XOR into binary classification!)

4. Data Issues

- **Insufficient samples**
- **Noisy labels**
- **Distribution shift** between train and test

None of these invalidate the UAT. They're just practical obstacles.

Alternative Formulation: 4 Inputs → 2 Outputs

There's another way to represent XOR that also works with a single layer: **expand the input space**.

Instead of 2 inputs (the actual binary values), use **4 inputs** (one-hot encoding of the four possible states):

```
import numpy as np

np.random.seed(42)
```

```python
def softmax(x):
    """Compute softmax values"""
    exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=1, keepdims=True)

def cross_entropy_loss(y_pred, y_true):
    """Cross entropy loss"""
    n = y_true.shape[0]
    log_likelihood = -np.log(y_pred[range(n), y_true])
    return np.sum(log_likelihood) / n

# XOR with 4 INPUTS → 2 OUTPUTS
# Each of the 4 binary combinations becomes a separate input feature
X = np.array([[1, 0, 0, 0],  # (0,0) as one-hot
              [0, 1, 0, 0],  # (0,1) as one-hot
              [0, 0, 1, 0],  # (1,0) as one-hot
              [0, 0, 0, 1]], # (1,1) as one-hot
             dtype=np.float32)

# Binary XOR outputs: (0,0)→0, (0,1)→1, (1,0)→1, (1,1)→0
y = np.array([0, 1, 1, 0], dtype=np.int32)

# Single layer: 4 inputs → 2 outputs
W = np.random.randn(4, 2) * 0.1
b = np.zeros(2)

learning_rate = 0.1
epochs = 5000

print("Training single-layer: 4 inputs → 2 outputs")
for epoch in range(epochs):
    # Forward pass
    logits = X @ W + b
    probs = softmax(logits)
    loss = cross_entropy_loss(probs, y)

    # Backward pass
    grad_logits = probs.copy()
    grad_logits[range(len(y)), y] -= 1
    grad_logits /= len(y)

    grad_W = X.T @ grad_logits
    grad_b = np.sum(grad_logits, axis=0)

    # Update weights
    W -= learning_rate * grad_W
    b -= learning_rate * grad_b

    if (epoch + 1) % 1000 == 0:
        print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss:.4f}')

# Evaluate
logits = X @ W + b
probs = softmax(logits)
predictions = np.argmax(probs, axis=1)
```

```
print("\nResults:")
input_labels = ["(0,0)", "(0,1)", "(1,0)", "(1,1)"]
for i in range(len(X)):
    print(f"{input_labels[i]}: True={y[i]}, Pred={predictions[i]}, "
          f"Probs=[0: {probs[i][0]:.3f}, 1: {probs[i][1]:.3f}]")

accuracy = np.mean(predictions == y)
print(f"\nAccuracy: {accuracy * 100:.0f}%")
```

Results

```
Training single-layer: 4 inputs → 2 outputs
Epoch [1000/5000], Loss: 0.0209
Epoch [2000/5000], Loss: 0.0103
Epoch [3000/5000], Loss: 0.0068
Epoch [4000/5000], Loss: 0.0051
Epoch [5000/5000], Loss: 0.0041

Results:
(0,0): True=0, Pred=0, Probs=[0: 0.996, 1: 0.004]
(0,1): True=1, Pred=1, Probs=[0: 0.004, 1: 0.996]
(1,0): True=1, Pred=1, Probs=[0: 0.004, 1: 0.996]
(1,1): True=0, Pred=0, Probs=[0: 0.996, 1: 0.004]

Accuracy: 100%
```

Again, 100% accuracy with a single layer!

What's Happening Here?

By expanding from 2D to 4D input space (one-hot encoding), we've transformed the problem:

- **Before (2D):** Points at (0,1) and (1,0) need the same label, but they're on opposite corners
- **After (4D):** Each point has its own dimension, trivially separable

The network learns simple weights:

- Input dimension 0 (for 0,0) → push toward class 0
- Input dimension 1 (for 0,1) → push toward class 1
- Input dimension 2 (for 1,0) → push toward class 1
- Input dimension 3 (for 1,1) → push toward class 0

This is exactly what hidden layers do implicitly - they transform the representation into a space where linear separation becomes possible!

The Lesson

The Universal Approximation Theorem is **correct**. When you can't reach arbitrarily small error, ask:

1. **Is my problem formulation correct?** (Don't artificially constrain it)
2. **Is my representation appropriate?** (Sometimes changing input/output encoding solves everything)
3. **Is my model capacity sufficient?** (Maybe you need more outputs, wider layers, etc.)
4. **Is optimization working?** (Check gradients, learning rate, convergence)
5. **Am I hitting computational limits?** (Precision, stability, convergence criteria)

The "XOR is impossible" myth exists because people:

- Used the **wrong formulation** (binary outputs with 2D inputs)
- Blamed the **theorem** instead of their setup
- Never questioned the **artificial constraints**
- Never tried alternative representations

We've shown **two** ways to solve XOR with a single layer:

1. **4 outputs** (4-class classification) - each input-output pair is unique
2. **4 inputs** (one-hot encoding) - expand the representation space

Both work perfectly. Both prove the same point.

Try It Yourself

Run the code above. You'll see a single-layer network solve XOR perfectly in multiple formulations. Then ask yourself:

Why did we ever think this was impossible?

The answer: We were using the theorem wrong. We were formulating the problem wrong. We were teaching the wrong lesson.

The Universal Approximation Theorem stands vindicated.

It's time to update the textbooks.

Complete code is provided for you to verify these results yourself. The "impossible" problem turns out to be quite simple when you formulate it correctly.

What do you think?

0 Responses

Upvote Funny Love Surprised Angry Sad

0 Comments Login

Start the discussion...

LOG IN WITH OR SIGN UP WITH DISQUS

Name

Share Best Newest Oldest

All articles

The Scalable Quasi-Perpetual Photonic Machine
17 Nov 2025

How to Make a Ball Orbit Itself
08 Nov 2025

The Boron-Nitrogen Catastrophe: A Critical Gap in Beta Decay Verification
28 Oct 2025

X-Ray Data Pipeline: Ultra-High-Speed Communication Through Limestone Tubes
26 Oct 2025

The Universal Approximation Theorem Is Right. You're Using It Wrong.
19 Oct 2025

The Power Law Illusion: A Measurement Artifact Hypothesis
11 Oct 2025

Fractional Gamma Function via Fractional Derivatives
18 Sep 2025

On the cruel irony of the P-NP problem
18 Dec 2023

The Journey from India to Germany: A Guide for IT Professionals
20 Jun 2023

How does AutoGPT work under the hood?
03 May 2023

Unit Test Recorder - Automatically generate unit tests as you use your application
25 Jun 2020

How to migrate from vanilla Kubernetes to Istio service mesh?
14 Oct 2019