1. Neural Networks Without the Magic

You’ve seen the explanations. Neurons that “fire.” Networks that “learn.” Backpropagation that “flows backward.” Gradient descent that “walks downhill.”

Metaphors. All of them.

Useful for intuition, perhaps. But you’re still left wondering: what actually happens? What are the real computations? When someone says “the network learns,” what changes, where, and why?

If you’ve watched the YouTube videos, read the blog posts, nodded along to the analogies — but still feel like you’re looking through frosted glass — this book is for you.

We’re going to touch the iron.

What This Book Is

This is a book for programmers. Not data scientists. Not researchers. Programmers who want to understand systems by building them.

We will implement neural networks from scratch in Rust. We will:

- Write the forward pass and see exactly what “feeding data through a network” means
- Implement backpropagation by hand.
- Build loss functions and understand why some numbers go down while the network gets smarter
- Construct a transformer piece by piece

We use metaphors too — they’re good teaching tools. But every metaphor will be followed by the real thing. The code. The math. The numbers flowing through arrays.

Why Rust

Not because Rust is fashionable. Because Rust is explicit.

In Python, a tensor is a magical object. You don't see its memory. You don't see its shape until something crashes. You call functions and trust that something sensible happens inside.

Rust doesn't allow that. Every dimension must be accounted for. Every allocation is visible. When something goes wrong, the compiler tells you where and why — often before you even run the code.

This explicitness is uncomfortable at first. But it's precisely what you need when learning. You can't hide from what you don't understand.

What You Will Build

Chapter	What You'll Touch
Tensors	The raw arrays that hold everything — create them, reshape them, see how operations really work
Forward Pass	Data flows through layers — watch the matrix multiplications happen
Loss Functions	Numbers that measure wrongness — compute them yourself, see what makes them shrink
Backpropagation	The algorithm that makes learning possible — implement it, step through it, finally understand it
Optimizers	The rules for updating weights — not magic, just arithmetic with a purpose
CNNs	Convolutions as sliding windows of multiplication — see every pixel's contribution

Chapter	What You’ll Touch
Transformers	Attention as matrix operations — the architecture that powers modern AI, demystified

By the end, you won’t just know how to use neural networks. You’ll know how they work.

Why Build From Scratch?

You have options. You could use pre-trained models — download someone else’s work and call their API. You could fine-tune — take an existing model and adapt it to your data. Both are valid. Both are faster than what we’ll do.

But neither teaches you how it works.

When you build from scratch, you:

- Debug problems by understanding what’s actually happening
- Choose architectures based on knowledge, not guesswork
- Understand why fine-tuning fails when it fails
- Make informed trade-offs between size, speed, and accuracy

The engineer who understands the machinery makes better decisions than the one who only knows the API.

The Honest Truth

Neural networks are powerful. They’ve enabled capabilities that seemed impossible a decade ago. But they’re not magic:

- They learn patterns from data, nothing more
- They fail when data is bad or missing
- They don’t understand anything — they approximate functions
- High confidence doesn’t mean correctness

- Bigger models aren’t always better for your use case

Understanding these limitations makes you a better practitioner. You’ll know when to use neural networks, when to avoid them, and how to build systems that account for their weaknesses.

Book Structure

Part I: Foundations (Chapters 1-4)

- **Chapter 1**: You are here — why we’re doing this
- **Chapter 2**: History — failures and breakthroughs that shaped the field
- **Chapter 3**: Rust for neural networks — the language patterns you’ll use
- **Chapter 4**: Tensors — the fundamental data structure

Part II: Building Blocks (Chapters 5-9)

- **Chapter 5**: Building your first neural network — a complete implementation
- **Chapter 6**: Loss functions and optimizers — measuring and minimizing error
- **Chapter 7**: Backpropagation from scratch — the algorithm that makes learning possible
- **Chapter 8**: Activation functions — the non-linearity that enables complex patterns
- **Chapter 9**: Learning rate — controlling the speed of learning

Part III: Architectures (Chapters 10-12)

- **Chapter 10**: Convolutional Neural Networks — image processing
- **Chapters 11–12**: Recurrent Neural Networks — sequence processing

Part IV: Transformers (Chapters 13-16)

- **Chapter 13**: Tokenizers — turning text into numbers
- **Chapter 14**: Embeddings — turning numbers into vectors
- **Chapter 15**: Self-attention — the core mechanism
- **Chapter 16**: Shakespeare Transformer — putting it all together

Appendices

- **Appendix A**: Pre-trained Models — using Hugging Face models in Candle
- **Appendix B**: Tensor Shape Errors — debugging common shape mismatches
- **Appendix C**: Inference Optimizations — making models faster on laptops

What We Assume

- **You can program.** Any language. If you've built non-trivial software, you're ready.
- **You're not afraid of math.** We'll use algebra and some calculus. We'll explain what we use, but we won't apologize for it.
- **You've seen Rust, or will learn alongside.** Chapter 3 covers what you need, but The Rust Book is there if you want more.
- **You're curious.** The kind of person who reads source code to understand libraries. Who wants to know, not just use.

What We Don't Assume

- Prior machine learning experience
- Python or PyTorch knowledge
- A GPU (everything runs on CPU; GPU acceleration is a bonus)
- Academic background in AI

How to Read This Book

Don’t just read. Build.

Every chapter has code. Type it. Run it. Then break it:

- What happens if you remove the activation function?
- What if the learning rate is 100x larger?
- What if you initialize all weights to zero?

The understanding comes from experimentation. The book gives you the foundation. The experiments make it yours.

Let’s Begin

We start with history in Chapter 2 — not as filler, but because the failures and breakthroughs of neural networks reveal the real constraints. Why did the field die twice? Why did it come back? What changed?

Then we set up Rust, learn tensors, and touch the iron.

5. Building Your First Neural Network

Time to write code.

Everything in the previous chapters — tensors, shapes, Rust patterns — comes together here. We’ll build a complete neural network that classifies Iris flowers based on four measurements. By the end, you’ll have working code that:

- Loads and preprocesses data
- Defines a neural network architecture
- Trains with backpropagation
- Evaluates accuracy

This is the pattern for every neural network. Once you understand it, you’ll recognize it in every paper, every tutorial, every production system.

The Problem: Iris Classification

The Iris dataset is the “Hello World” of machine learning. It’s small enough to understand completely, but real enough to demonstrate actual learning.

The data: 150 iris flowers, 3 species, 4 measurements each:

- Sepal length
- Sepal width
- Petal length
- Petal width

The task: Given the 4 measurements, predict which species.

Shape flow:

```
Input:   [batch, 4]    → 4 features per flower
Hidden:  [batch, 32]   → internal representation
Output:  [batch, 3]    → probability for each species
```

Simple. Let’s build it.

The Complete Code Structure

Before diving into details, here’s what we’re building:

```
1. Hyperparameters     → Constants that control training
2. Model struct        → The neural network architecture
3. Data loading        → Read CSV, normalize, create batches
4. Training loop       → Forward pass → loss → backward pass →
    update
5. Evaluation          → Check accuracy, confusion matrix
```

Every neural network follows this structure. The details change; the structure doesn’t.

1. Setup and Hyperparameters

```rust
use anyhow::Result;
use candle_core::{DType, Device, IndexOp, Tensor};
use candle_nn::{Module, Optimizer, VarBuilder, VarMap};
use rand::{rngs::StdRng, Rng, SeedableRng};
use std::fs::File;
use std::io::{BufRead, BufReader};
use std::path::Path;

// Hyperparameters — the knobs you can turn
const INPUT_SIZE: usize = 4;      // Fixed by dataset: 4 features
const HIDDEN_SIZE: usize =
    32;    // Your choice: capacity of the network
const OUTPUT_SIZE: usize = 3;     // Fixed by dataset: 3 classes
const BATCH_SIZE: usize = 32;     // Your choice: samples per update
const LEARNING_RATE: f64 = 0.01;  // Your choice: step size
const EPOCHS: usize =
    100;        // Your choice: passes through data
```

What these control:

- `HIDDEN_SIZE` : Larger = more capacity, but slower and risk of overfitting
- `BATCH_SIZE` : Larger = more stable gradients, but less frequent updates
- `LEARNING_RATE` : Larger = faster training, but risk of overshooting
- `EPOCHS` : More = more training, but risk of overfitting

Experiment: After running the code, try changing these. What happens with `HIDDEN_SIZE = 4` ? With `LEARNING_RATE = 0.5` ? With `EPOCHS = 10` ?

2. The Model

Here's the neural network — everything in one struct:

```rust
struct IrisClassifier {
    layer1: candle_nn::Linear,  // 4 → 32
    layer2: candle_nn::Linear,  // 32 → 3
}

impl IrisClassifier {
    fn new(vb: VarBuilder) -> Result<Self> {
        // Create layers with named parameters
        let layer1 = candle_nn::linear(
            INPUT_SIZE, HIDDEN_SIZE, vb.pp("layer1"))?;
        let layer2 = candle_nn::linear(
            HIDDEN_SIZE, OUTPUT_SIZE, vb.pp("layer2"))?;
        Ok(Self { layer1, layer2 })
    }

    fn forward(&self, x: &Tensor) -> Result<Tensor> {
        // Trace the shapes:
        // x:      [batch, 4]
        let h = self.layer1.forward(x)?;
        // h:      [batch, 32]
        let h = h.relu()?;
        // h:      [batch, 32]  (same shape, just zeroed negatives)
        let out = self.layer2.forward(&h)?;
        // out:    [batch, 3]
        Ok(out)
    }
}
```

What's happening:

1. `layer1` transforms 4 inputs → 32 hidden values (matrix multiply

- bias)

2. `relu()` zeros out negative values (adds non-linearity)
3. `layer2` transforms 32 hidden → 3 output values

Why ReLU? Without non-linearity, stacking layers does nothing — the whole network collapses to a single linear transformation. ReLU is the simplest non-linearity: `max(0, x)`.

Why these dimensions? Input (4) and output (3) are fixed by the problem. Hidden (32) is a choice — enough capacity to learn, not so much that it memorizes.

3. Data Loading and Preprocessing

The unglamorous but essential part:

```rust
fn load_iris_dataset(device: &Device) -> Result<(Tensor, Tensor)> {
    let file_path = Path::new("data/iris.csv");
    let file = File::open(file_path)?;
    let reader = BufReader::new(file);

    let mut features_data: Vec<f32> = Vec::new();
    let mut labels_data: Vec<u32> = Vec::new();

    for (i, line_result) in reader.lines().enumerate() {
        if i == 0 { continue; }  // Skip header

        let line = line_result?;
        let values: Vec<&str> = line.split(',').collect();

        // Parse 4 feature values
        for j in 0..4 {
            let value = values[j].parse::<f32>()?;
            features_data.push(value);
        }

        // Parse label (species name → number)
        let label = match values[4].trim() {
            "Iris-setosa" => 0,
            "Iris-versicolor" => 1,
            "Iris-virginica" => 2,
            other => {
                return Err(anyhow::anyhow!("Unknown: {}", other))
            }
        };
        labels_data.push(label);
    }

    // Create tensors
    let num_samples = labels_data.len();
    let features = Tensor::from_vec(features_data, (num_samples,
        4), device)?;
    let labels = Tensor::from_slice(&labels_data, (num_samples,),
        device)?;

    // Normalize: scale each feature to [0, 1]
    let min = features.min(0)?;
    let max = features.max(0)?;
    let range = max.sub(&min)?;
    let normalized = features
        .broadcast_sub(&min)?.broadcast_div(&range)?;
    Ok((normalized, labels))
}
```

Why normalize? Raw features have different scales (sepal length ~5-8cm, petal width ~0.1-2.5cm). Without normalization, larger-scale features dominate the learning. Normalizing puts all features on equal footing.

The shapes:

- `features` : `[150, 4]` — 150 samples, 4 features each
- `labels` : `[150]` — 150 labels (0, 1, or 2)

Batch Generation

For training, we shuffle and split into batches:

```rust
fn generate_batches(
    features: &Tensor,
    labels: &Tensor,
    batch_size: usize,
    device: &Device,
    rng: &mut StdRng,
) -> Result<Vec<(Tensor, Tensor)>> {
    let num_samples = features.dim(0)?;

    // Shuffle indices
    let mut indices: Vec<usize> = (0..num_samples).collect();
    for i in (1..indices.len()).rev() {
        let j = rng.gen_range(0..=i);
        indices.swap(i, j);
    }

    // Create batches
    let mut batches = Vec::new();
    for chunk in indices.chunks(batch_size) {
        let mut batch_features = Vec::new();
        let mut batch_labels = Vec::new();

        for &idx in chunk {
            let f = features.i(idx)?.to_vec1::<f32>()?;
            batch_features.extend(f);
            batch_labels.push(labels.i(idx)?.to_scalar::<u32>()?);
        }

        let bf = Tensor::from_slice(&batch_features, (chunk.len(),
            4), device)?;
        let bl = Tensor::from_slice(&batch_labels, (chunk.len(),),
            device)?;
        batches.push((bf, bl));
    }

    Ok(batches)
}
```

Why shuffle? If samples come in order (all Setosa, then all Versicolor, then all Virginica), the network sees skewed batches. Shuffling gives each batch a mix of classes.

Why batches? Processing all 150 samples at once is fine for this dataset. But for large datasets, batches let you update weights more frequently and fit in memory.

4. The Training Loop

This is where learning happens:

```rust
fn main() -> Result<()> {
    // Device setup
    let device = Device::Cpu;  // Or Device::cuda_if_available(0)?
    println!("Using device: {:?}", device);

    // Load data
    let (features, labels) = load_iris_dataset(&device)?;
    println!("Loaded {} samples", features.dim(0)?);

    // Create model
    let varmap = VarMap::new();
    let vb = VarBuilder::from_varmap(&varmap, DType::F32, &device);
    let model = IrisClassifier::new(vb)?;

    // Create optimizer
    let params = varmap.all_vars();
    let mut opt = candle_nn::AdamW::new_lr(params, LEARNING_RATE)?;

    // Random seed for reproducibility
    let mut rng = StdRng::seed_from_u64(42);

    // Training loop
    println!("\nTraining...");
    for epoch in 0..EPOCHS {
        let batches = generate_batches(
            &features, &labels, BATCH_SIZE, &device, &mut rng)?;

        let mut total_loss = 0.0;
        let mut total_correct = 0;
        let mut total_samples = 0;

        for (batch_features, batch_labels) in &batches {
            // Forward pass
            let logits = model.forward(batch_features)?;

            // Loss: cross-entropy for classification
            let loss = candle_nn::loss::cross_entropy(&logits,
                batch_labels)?;

            // Backward pass + weight update
            opt.backward_step(&loss)?;

            // Track metrics
            total_loss += loss.to_scalar::<f32>()?;

            let predictions = logits.argmax(1)?;
            for i in 0..batch_labels.dim(0)? {
                let pred = predictions.i(i)?.to_scalar::<u32>()?;
                let true_label =
                    batch_labels.i(i)?.to_scalar::<u32>()?;
                if pred == true_label {
                    total_correct += 1;
                }
                total_samples += 1;
            }
        }

        // Print progress
        if epoch % 10 == 0 || epoch == EPOCHS - 1 {
            let avg_loss = total_loss / batches.len() as f32;
            let accuracy = total_correct as f32 /
                total_samples as f32;
            println!(
                "Epoch {:3}/{}: loss = {:.4}, accuracy = {:.2}%",
                epoch + 1,
                EPOCHS,
                avg_loss,
                accuracy * 100.0
            );
        }
    }

    Ok(())
}
```

What `backward_step` does:

1. Computes gradients of loss with respect to all weights (backpropagation)
2. Updates weights in the direction that reduces loss (optimization)

This is the magic line — everything before is setup, everything after is bookkeeping.

Why cross-entropy? For classification, cross-entropy penalizes confident wrong predictions heavily. If the model says “99% Setosa” and it’s actually Versicolor, the loss is huge. MSE doesn’t have this property.

5. Evaluation

After training, check how well it worked:

```
// Final evaluation
let logits = model.forward(&features)?;
let predictions = logits.argmax(1)?;

// Confusion matrix
let mut confusion = vec![vec![0usize; 3]; 3];
for i in 0..features.dim(0)? {
    let pred = predictions.i(i)?.to_scalar::<u32>()? as usize;
    let true_label = labels.i(i)?.to_scalar::<u32>()? as usize;
    confusion[true_label][pred] += 1;
}

println!("\nConfusion Matrix:");
println!("           Predicted");
println!("           Set  Ver  Vir");
let c = &confusion;
println!("Actual Set  {:2}   {:2}   {:2}", c[0][0], c[0][1],
    c[0][2]);
println!("       Ver  {:2}   {:2}   {:2}", c[1][0], c[1][1],
    c[1][2]);
println!("       Vir  {:2}   {:2}   {:2}", c[2][0], c[2][1],
    c[2][2]);

let correct: usize = (0..3).map(|i| confusion[i][i]).sum();
let total = features.dim(0)?;
let acc = 100.0 * correct as f32 / total as f32;
println!("\nFinal accuracy: {:.1}% ({}/{})", acc, correct, total);
```

Expected output (approximately):

```
Confusion Matrix:
              Predicted
              Setosa  Versicolor  Virginica
Actual Setosa     50           0          0
    Versicolor     0          47          3
    Virginica      0           1         49

Final accuracy: 97.3% (146/150)
```

What this tells you:

- Setosa is perfectly classified (linearly separable from others)
- Versicolor and Virginica have some overlap (a few misclassifications)
- 97% accuracy on a simple network — the features are predictive

What Can Go Wrong

Loss stays high / accuracy stays low

Possible causes:

- Learning rate too high (overshooting) or too low (not moving)
- Not enough hidden units
- Bug in data loading (labels don’t match features)

Debug: Print shapes at each step. Check that loss decreases at all in first few epochs.

Loss goes to NaN

Possible causes:

- Learning rate too high
- Missing normalization (huge input values)
- Division by zero somewhere

Debug: Print intermediate values. Check for zeros in your data.

Accuracy stuck at ~33%

Possible causes:

- Model always predicting same class
- Labels shuffled incorrectly relative to features

Debug: Print actual predictions. Check that different inputs give different outputs.

Experiments to Try

Don't just run the code. Break it. Fix it. Learn.

1. **Remove ReLU**: Comment out `h.relu()` . What happens to accuracy? Why?
2. **Change hidden size**: Try 4, 8, 64, 256. What's the effect on accuracy? On training speed?
3. **Change learning rate**: Try 0.001, 0.1, 1.0. When does training diverge?
4. **Remove normalization**: Skip the min-max scaling. Does it still learn?
5. **Overfit**: Set EPOCHS = 1000 with a small hidden size. What happens?
6. **Add a layer**: Add a third linear layer. Does it help? (Hint: probably not for this simple dataset)

The Pattern

Every neural network you'll ever build follows this structure:

```
1. Define hyperparameters
2. Define model (struct + forward)
3. Load and preprocess data
4. Training loop:
   for epoch in epochs:
       for batch in batches:
           output = model.forward(input)
           loss = loss_function(output, target)
           optimizer.backward_step(loss)  ← the magic line
5. Evaluate on test data
```

The model architecture changes. The loss function changes. The data changes. The structure doesn't.

What's Next

You've built a neural network. It works. You understand every line.

But we skipped over some things:

- What other loss functions exist? → Chapter 6 (Loss Functions and Optimizers)
- What exactly happens in `backward_step`? → Chapter 7 (Backpropagation)
- Why ReLU and not something else? → Chapter 8 (Activation Functions)
- How to choose a learning rate? → Chapter 9 (The Learning Rate)

The next chapters dig into these details. Now that you have working code, the theory will make more sense.