Lesson 34: MLPs and Activations

Data Science with Python – BSc Course

Data Science Program

BSc Course

45 Minutes

Previously on L33...

Where we left off: the perceptron hit a wall.

- A single perceptron can only draw **one straight line**
- XOR requires at least **two lines** – perceptron fails
- Today: stack neurons into **layers** to learn any pattern

Minsky & Papert (1969) proved the XOR limitation. Solution: multi-layer networks.

Learning Objectives

The Question: One neuron draws straight lines. How do we learn curves, circles, spirals?

After this lesson, you will be able to:

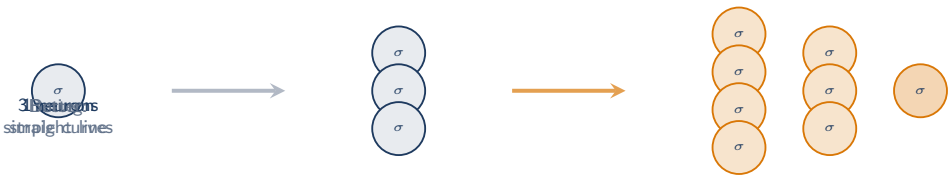
1. Explain **MLP architecture** (input, hidden, output layers)
2. Choose the right **activation function** for each layer
3. Build neural networks with **Keras** in 5 lines of code
4. Apply MLPs to **non-linear classification** problems

Key Insight: Stacking simple neurons creates something far greater than the sum of its parts.

Finance Application: Non-linear regime detection and pattern recognition

The Lego Insight

One brick is boring. But stacked bricks build anything.



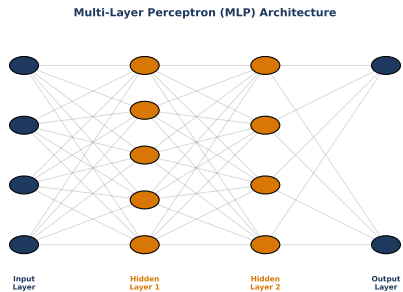
Neural networks follow the same principle: individually simple neurons, combined in layers, can approximate **any function**.

This is the core idea behind deep learning: composing simple transformations.

MLP Architecture

Input → Hidden → Output

- **Input layer:** receives raw features (not learned)
- **Hidden layers:** where the magic happens – non-linear transformations
- **Output layer:** produces final prediction



Each connection has a weight. Each neuron has a bias. All are learned during training.

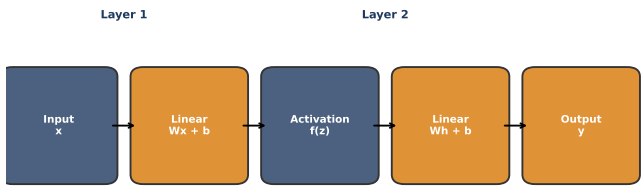
Feedforward Computation

How Data Flows Through the Network

Each layer computes: $\mathbf{a} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$

- $\mathbf{x}$: input from previous layer
- $\mathbf{W}$: weight matrix, $\mathbf{b}$: bias vector
- g: activation function (e.g., ReLU)
- $\mathbf{a}$: output (activation) passed to next layer

Feedforward: Data Flows Forward Through Layers

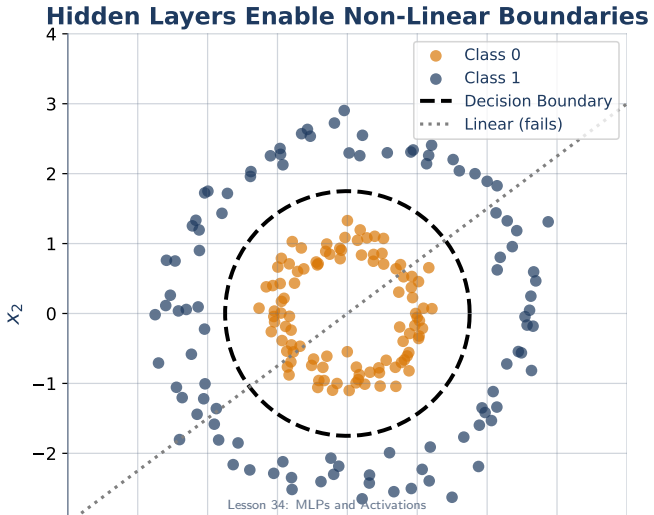


$$h^{(1)} = f(W^{(1)}x + b^{(1)}) \quad y = W^{(2)}h^{(1)} + b^{(2)}$$

Why Hidden Layers Work

Each layer transforms the feature space

- 1 layer = lines 2 layers = curves 3+ = arbitrary shapes
- Hidden layers "unfold" tangled data into separable regions



The Personality of a Neuron

Every neuron makes a decision: how strongly do I respond?

Without non-linear activation, stacking layers is **useless**:

$$\underbrace{W_3 \cdot (W_2 \cdot (W_1 \cdot \mathbf{x}))}_{\text{3 linear layers}} = \underbrace{(W_3 W_2 W_1) \cdot \mathbf{x}}_{\text{= 1 linear layer!}}$$

The fix: add a non-linear function $g(\cdot)$ after each layer:

$$\mathbf{a} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$$

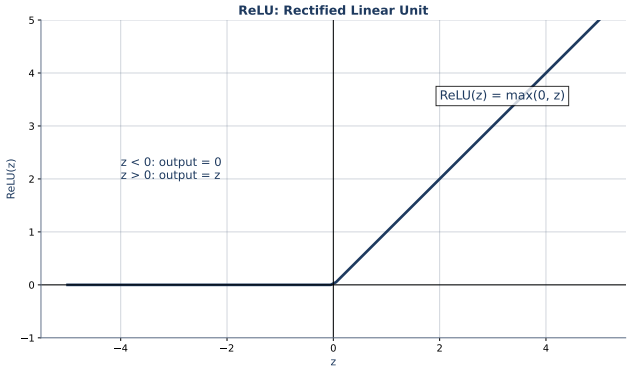
- Without g: 100 layers = 1 layer (just matrix multiplication)
- With g: each layer learns genuinely new transformations

Activation functions are what make depth meaningful. Without them, deep = shallow.

ReLU: The Modern Default

ReLU$(x) = \max(0, x)$ – dead simple and effective

- No vanishing gradient (gradient $= 1$ when active)
- Computationally cheap: just `max(0, z)`
- Sparse activation: many neurons output exactly 0

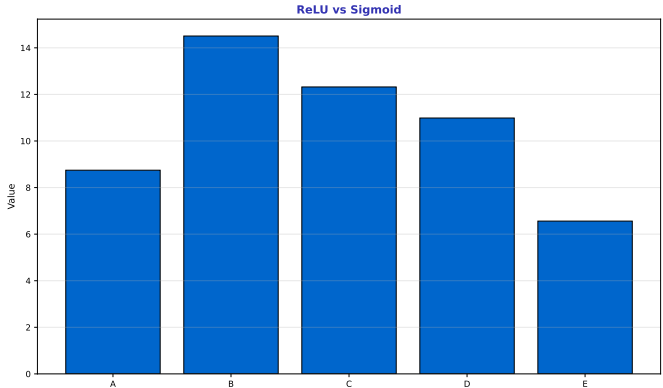


Use ReLU for hidden layers in 99% of cases. It just works.

ReLU vs Sigmoid: Why ReLU Won

Sigmoid's fatal flaw: vanishing gradients

- Sigmoid: gradient $\to 0$ at extremes (deep nets can't learn)
- ReLU: constant gradient of 1 when active
- Warning: "dying ReLU" – if always $z < 0$, neuron is permanently dead

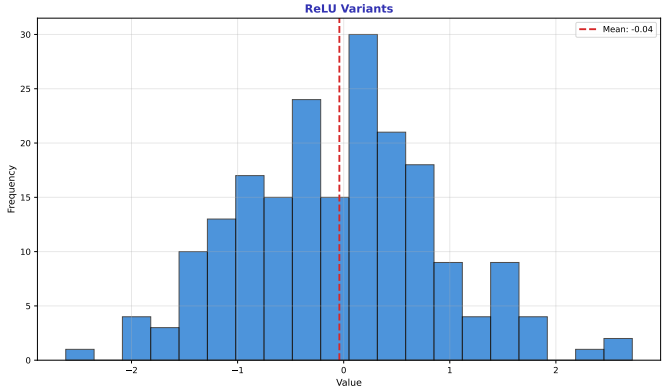


ReLU replaced sigmoid as the default hidden activation around 2012.

ReLU Variants

Fixes for the "Dying ReLU" Problem

- **Leaky ReLU:** small slope for $x < 0$ (e.g., $0.01x$)
- **ELU:** smooth curve for negatives, zero-centered
- **SELU:** self-normalizing, maintains mean/variance



Try Leaky ReLU if standard ReLU underperforms. Often not needed in practice.

Output Activations: Match Task to Function

Hidden layers: always ReLU. Output layer: depends on your TASK.

Task	Output Neurons	Activation	Loss Function
Regression	1	Linear (none)	MSE
Binary classif.	1	Sigmoid	Binary CE
Multiclass (K)	K	Softmax	Categorical CE
Multi-label (K)	K	Sigmoid	Binary CE

Key Rule: Loss function must match output activation!

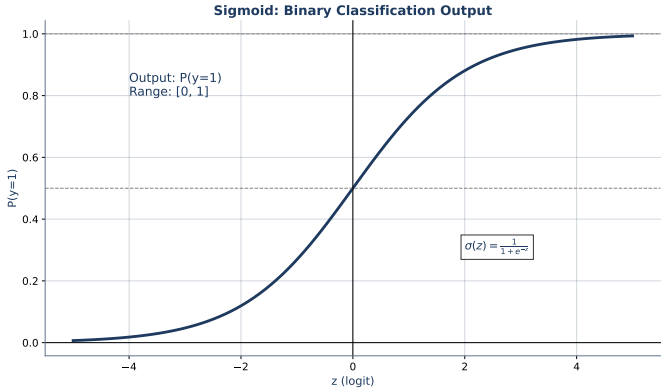
- Sigmoid: probability for ONE class (or independent labels)
- Softmax: probabilities for ALL classes (mutually exclusive, sum $= 1$)

Forgetting to match activation and loss is the #1 Keras beginner mistake.

Sigmoid for Binary Output

Full circle: logistic regression IS a one-layer neural network

- Same sigmoid from L25 (Logistic Regression) and L33 (Perceptron)
- Output $\in (0, 1)$: interpret as $P(\text{class} = 1)$
- Threshold at 0.5 for class prediction

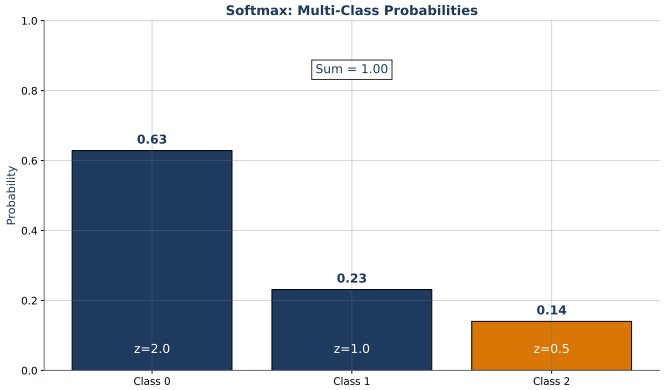


Logistic regression = neural network with 0 hidden layers and sigmoid output.

Softmax for Multiclass

K output neurons for K classes, probabilities sum to 1

- Each neuron gives probability of one class
- $\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$
- Prediction: $\hat{y} = \arg\max_i \text{softmax}(z_i)$



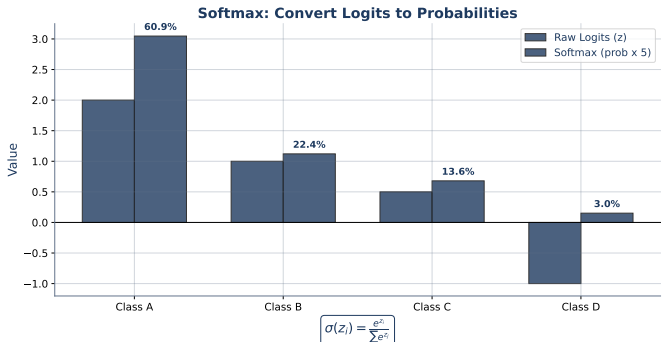
Softmax generalizes sigmoid to multiple classes. Sigmoid is softmax with $K = 2$.

Softmax Formula Deep Dive

Understanding the Mechanics

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

- Exponentiation makes all values positive
- Division by sum normalizes to probability distribution
- Larger $z_i \rightarrow$ exponentially larger probability
- Temperature scaling: $\text{softmax}(z_i/T)$ controls sharpness



Keras: Building Networks in 5 Lines

The entire workflow:

```
1. model = Sequential([  Dense(64, 'relu', input_shape=(10,)),
       Dense(32, 'relu'),
       Dense(1, 'sigmoid')])

2. model.compile(optimizer='adam',
       loss='binary_crossentropy',
       metrics=['accuracy'])

3. model.fit(X_train, y_train, epochs=50)

4. model.evaluate(X_test, y_test)

5. model.predict(X_new)
```

That's it. Define → Compile → Fit → Evaluate → Predict.

Keras makes neural networks accessible. The hard part is choosing the right architecture.

Keras: Compile, Train, Evaluate

Compile – configure the learning process:

- `optimizer`: how weights update (`'adam'` is the default)
- `loss`: what to minimize (must match output activation)
- `metrics`: what to monitor (`['accuracy']` for classification)

Fit – train the model:

- `epochs`: number of passes through data (start with 50–100)
- `batch_size`: samples per gradient update (32 is common)
- `validation_split=0.2`: hold 20% for validation

After training:

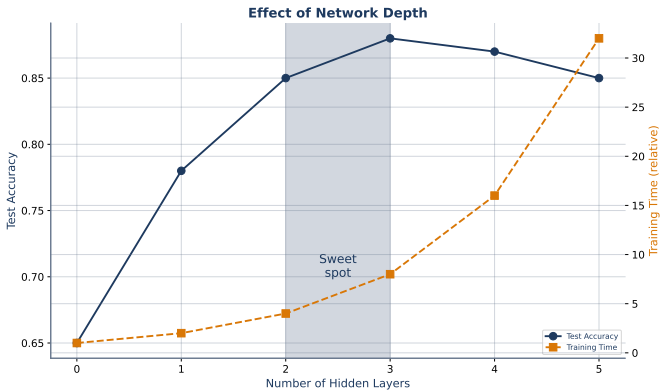
- `history.history['loss']` – plot training curves
- `model.evaluate()` – test set performance
- `model.predict()` – generate predictions

Always monitor validation loss to detect overfitting early.

Architecture Design: How Big?

Start simple. Add complexity only when needed.

- **Start:** 1 hidden layer, 64 neurons
- **Funnel shape:** 128 $\rightarrow$ 64 $\rightarrow$ 32 (compress features)
- **Powers of 2:** 32, 64, 128, 256 (GPU-friendly)
- Underfitting? Add capacity. Overfitting? Regularize (L36).

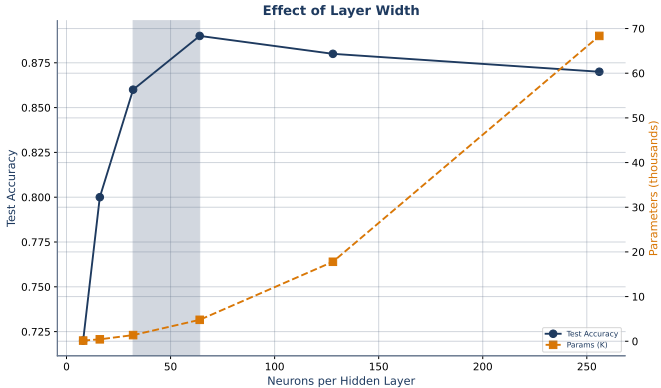


Architecture tuning uses validation data, never test data.

Effect of Layer Width

Wider Layers Capture More Features

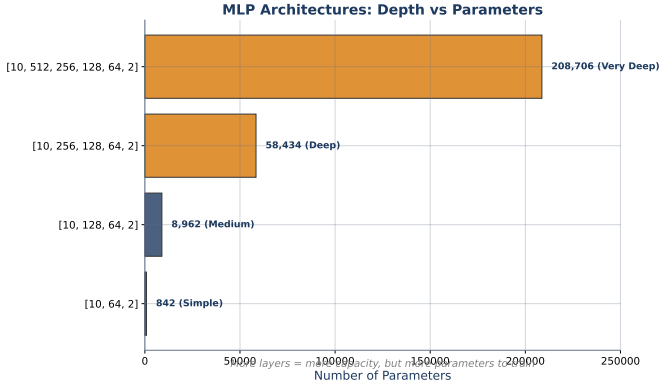
- More neurons per layer = more features learned at that level
- But diminishing returns: 256 → 512 often adds little
- Wider networks also need more data to generalize



Width vs depth: depth creates hierarchical features; width captures parallel features.

Example Architectures

Common Patterns for Different Problems



Common patterns: 64-32, 128-64-32, pyramidal (decreasing width).

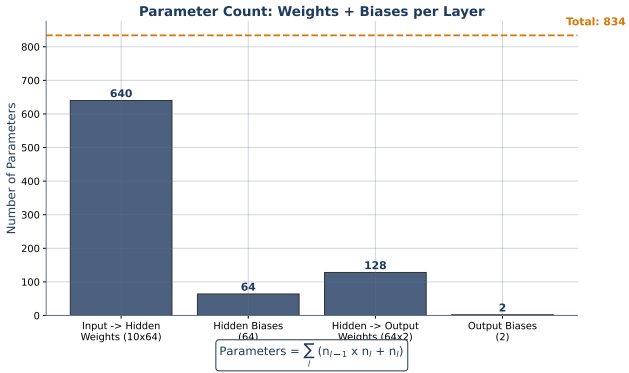
Counting Parameters

Each Dense layer: (inputs $+1$) $\times$ outputs parameters

The $+1$ is the **bias** term (one per output neuron).

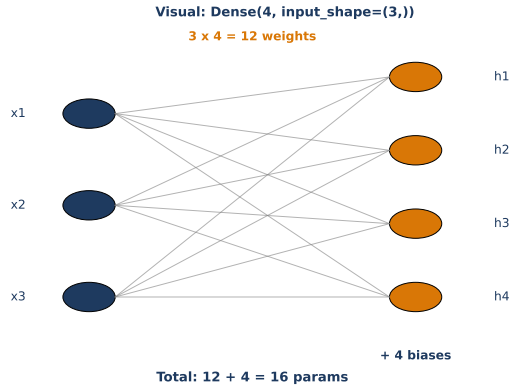
Worked example: (4, 5, 3, 1) network

- Layer 1: $(4+1) \times 5 = 25$ params
- Layer 2: $(5+1) \times 3 = 18$ params
- Layer 3: $(3+1) \times 1 = 4$ params
- **Total: 47 parameters**



Visual: Dense(4, input_shape=3)

Seeing the Connections



3 inputs $\times$ 4 outputs + 4 biases = 16 parameters

model.summary() Output

Always Verify Your Architecture

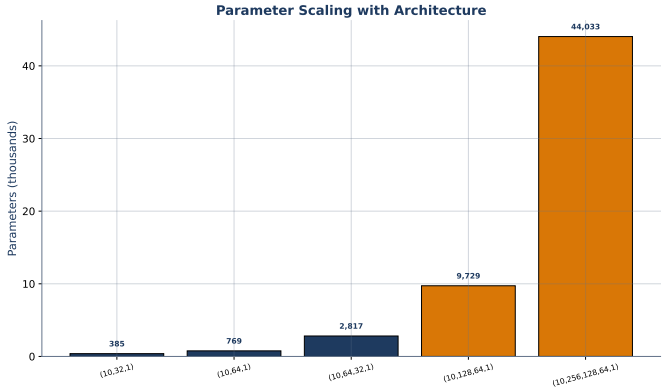
`model.summary()` shows layer-by-layer:

- Layer name and type
- Output shape (`None` = batch size)
- Parameter count per layer
- Total trainable vs non-trainable params

Pro tip: Run `model.summary()` immediately after building. Catches shape mismatches before training wastes time.

Always check model.summary() to verify architecture before training.

Parameter Scaling with Depth



Parameters grow quickly with width and depth – balance complexity vs data size.

Activation Cheat Sheet

ReLU
Hidden layers
$\max(0, x)$
Default choice

Sigmoid
Binary output
$(0, 1)$
1 class probability

Softmax
Multiclass output
Sums to 1
K class probabilities

Linear
Regression output
No bound
Any real number

Rule of thumb: Hidden = ReLU. Output = match your task.

Memorize this table. It covers 99% of practical neural network design.

The Universal Approximation Theorem

CLIMAX: A network with ONE hidden layer can approximate ANY continuous function.

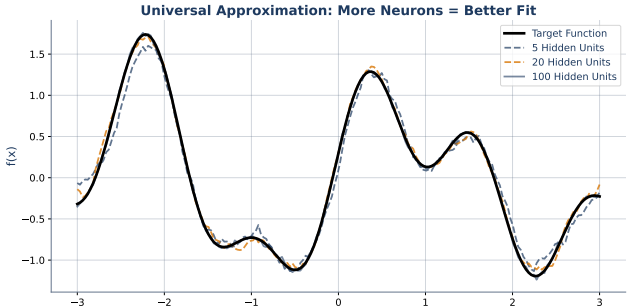
Cybenko (1989) / Hornik (1991)

What it says:

- Given enough neurons, a single hidden layer MLP can get arbitrarily close to any continuous function on a bounded domain

The catch:

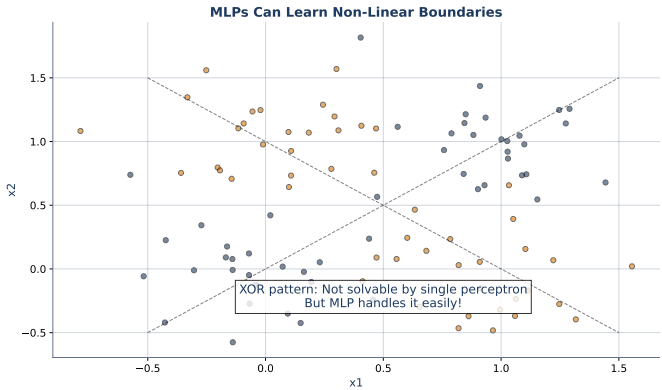
- "Enough neurons" may mean **exponentially many**
- In practice, **depth beats width** – fewer total parameters needed



Universal Approximation in Action

More neurons = better fit. XOR? No problem.

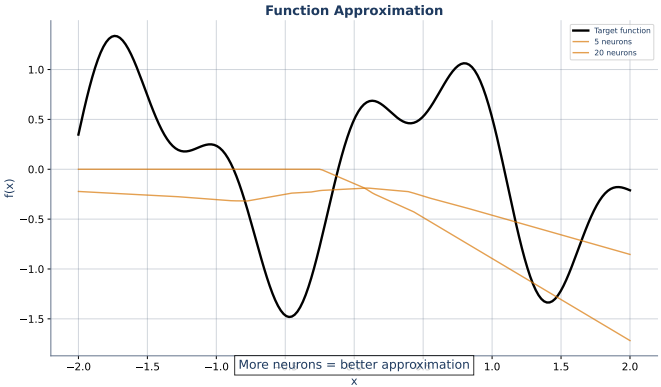
- Networks learn curves, circles, spirals – any shape
- The Lego tower from slide 4 is now mathematically justified



XOR, circles, spirals – all learnable with sufficient network capacity.

Watching Universal Approximation Happen

- As neurons increase: approximation gets tighter
- 3 neurons: rough shape. 10: good fit. 100: near-perfect.
- But more neurons = more parameters = overfitting risk

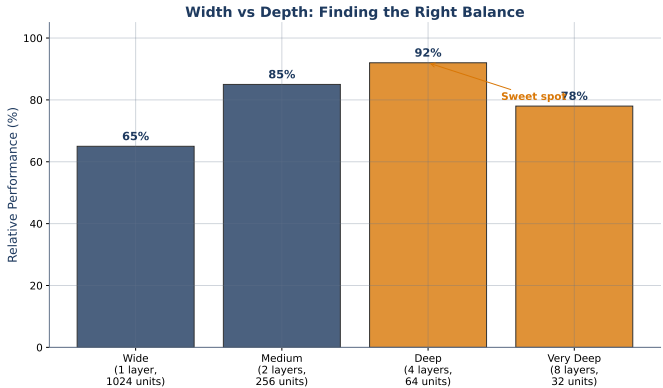


More neurons = better fit, but also more risk of memorizing noise.

Practical Implications

From Theory to Practice

- Width alone works in theory, depth works in practice
- Deep networks reuse features hierarchically
- Regularization (L36) prevents overfitting the extra capacity

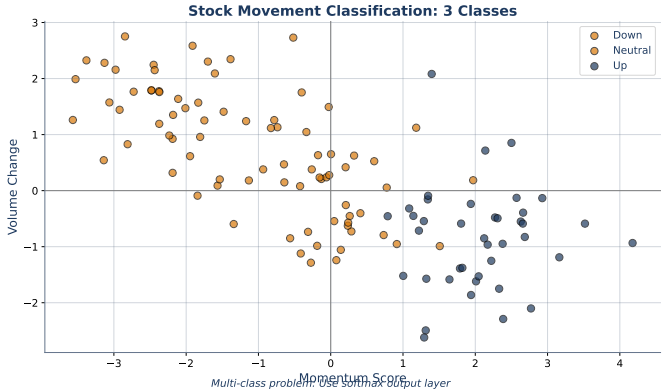


Deeper often beats wider. Regularization prevents overfitting.

Finance: Market Regime Detection

3-class problem: bull / bear / sideways

- Inputs: volatility, momentum, correlation features
- Output: softmax over 3 regime classes
- Non-linear boundaries that logistic regression misses

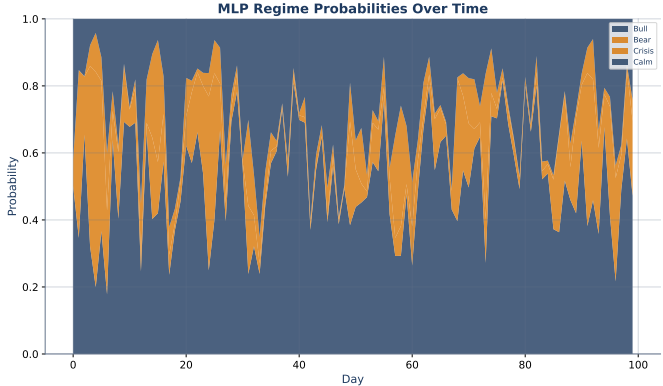


MLPs capture non-linear regime boundaries that linear models miss entirely.

Regime Probabilities

Softmax gives a probability distribution over regimes

- Not just "bull" or "bear" – probabilities for each state
- Risk management: act differently when model is uncertain
- Smooth transitions between regimes (no hard switches)

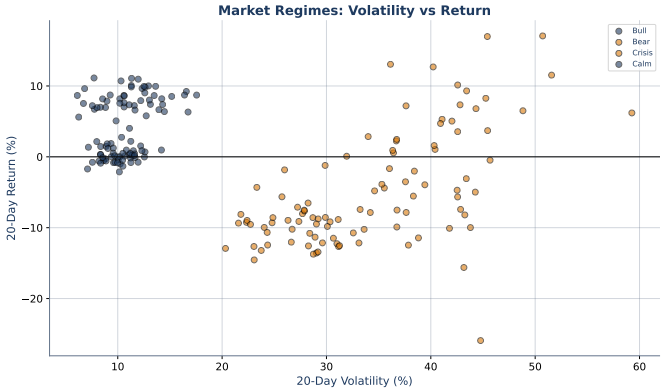


Probability outputs enable nuanced trading strategies based on confidence.

Market Regimes and Volatility Clustering

Volatility Regimes in Real Markets

- High volatility clusters with high volatility (GARCH effect)
- Regimes persist: markets don't switch randomly
- MLP features: rolling volatility, return momentum, VIX



Regime persistence makes regime detection a valuable prediction problem.

Hands-On Exercise (25 min)

Task: Build MLP for XOR and Beyond

1. Create XOR dataset and verify perceptron **fails**
2. Build MLP: 2 inputs $\rightarrow$ 4 hidden (ReLU) $\rightarrow$ 1 output (sigmoid)
3. Train and verify **100% accuracy** on XOR
4. Visualize decision boundary – observe non-linearity

Extension: 3-class classification with softmax output

Deliverable: XOR decision boundary plot + accuracy report.

Try different hidden sizes (2, 4, 8, 16) – how does the boundary change?

Common Mistakes with MLPs

Three errors that waste hours of debugging:

1. **Forgetting to scale features**
   - Neural networks are sensitive to feature magnitudes
   - Always use `StandardScaler` before training
2. **Using sigmoid in hidden layers**
   - Causes vanishing gradients – network can't learn
   - Fix: use **ReLU** for all hidden layers
3. **Wrong output activation for the task**
   - Binary with softmax, multiclass with sigmoid, regression with sigmoid
   - Fix: check the cheat sheet on slide 16

All three are easy to make and hard to debug. Check these first when stuck.

Key Takeaways

What we learned today – the "Stacking Legos" story:

1. **Hidden layers** = non-linear power (without them, deep = shallow)
2. **ReLU** for hidden layers, **task-specific** activation for output
3. **Universal Approximation:** MLPs can learn ANY continuous function
4. **Keras:** define → compile → fit → evaluate → predict
5. **Start simple:** 1 hidden layer, 64 neurons, add complexity as needed

The Lego principle: simple bricks, stacked with purpose, build anything.

Memory: ReLU = max(0,x). Hidden layers = non-linear power. Keras = 5 lines.

Preview: Learning from Mistakes

We can BUILD networks. But how do they LEARN?

- We know how to stack layers (architecture)
- We know which activations to use (ReLU, sigmoid, softmax)
- But how does the network figure out the right **weights**?

Next lesson (L35): Backpropagation

- The chain rule applied to networks
- Gradient descent through layers
- Why depth makes learning harder (and how to fix it)

"The network makes a prediction, compares to truth, and adjusts every weight to do better next time."

L35 Backpropagation: the algorithm that made deep learning possible.