Opening the “Black Box”

From Linear Regression to ChatGPT in 60 Minutes

Eduardo Dueñez

University of Texas at San Antonio

Universidad Panamericana
21 May 2026

https://github.com/eduenez/public-site

Three mathematical surprises

Modern AI rests on three pillars from undergraduate mathematics:

$$\textbf{linear algebra} \quad + \quad \textbf{calculus} \quad + \quad \textbf{probability} \quad — \quad \textit{at scale.}$$

Today: three mathematical surprises hiding inside that combination.

The three surprises

1. Universal approximation. Any continuous function lives in the space neural networks can represent.
2. Double descent. Making the model *bigger* past the data size makes it generalize *better*.
3. In-context learning. Large language models learn at inference time, without changing a weight.

Each surprise opens onto an unsolved problem of contemporary research.

Before we begin — a live demo

Most of you have used **ChatGPT**, **Claude**, **Gemini**.

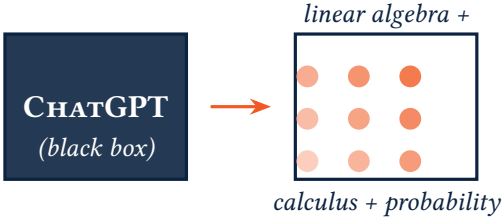
But, probably… you haven't peeked inside.

Let us open the "Black Box".

A single forward pass of one of these models is described by exactly three numbers:

- ~ 3 billion parameters (weights)
- ~ 2 trillion tokens of training data
- ~ 30,000 GPU-hours of training

By the end of this hour, we will know what each of these numbers "buys you".



[Switch to terminal. `ollama run llama3.2`*. Activity Monitor reveal.]*

The simplest learner: linear regression

Given data $\{(x_i, y_i)\}_{i=1}^n$,
fit a linear regression model:

$$\widehat{\mathbf{y}}(x) \;=\; w\,x + b.$$

"Loss". Sum of squared errors:

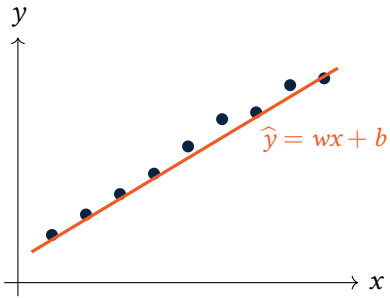
$$L(w, b) \;=\; \sum_{i=1}^{n} \big(y_i - \widehat{y}(x_i)\big)^2.$$

Optimization. Gradient descent:

$$w \;\leftarrow\; w - \rho \cdot \partial_w L,$$
$$b \;\leftarrow\; b - \rho \cdot \partial_b L.$$

(Small ρ: Learning rate.)

Calculus does all the work.



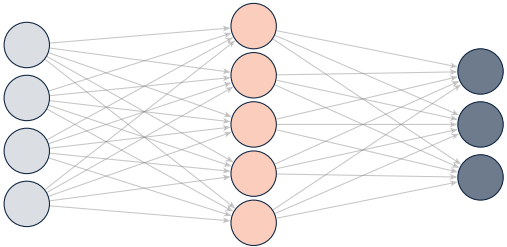
The "neural network" with:

- 1 **neuron**;
- 2 **parameters**:
  - w: "**weight**",
  - b: "**bias**";
- 0 "**activations**" (non-linear functions).

From linear regression to non-linear neural network

(**1**) "Stack"/compose two or more linear (affine) maps... **but** (**2**) "squeeze" a non-linear **activation** function (applied *coordinate-wise*) between them! **That is all!**

$$\widehat{\mathbf{y}}(\mathbf{x}) \;=\; \mathbf{W}_2 \cdot \sigma\big(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1\big) + \mathbf{b}_2, \qquad \sigma = \text{ReLU}, \tanh, \text{GeLU}, \ldots : \text{Non-linear activation function.}$$



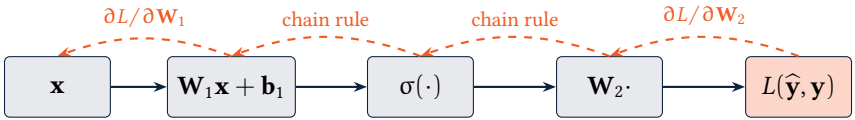
input $\mathbf{x} \in \mathbb{R}^{784}$ hidden $\sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \in \mathbb{R}^{128}$ output $\widehat{\mathbf{y}} \in \mathbb{R}^{10}$

(2 linear maps) + (1 non-linear "activation"). All the magic is in finding the right $\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2$.

Backpropagation = Chain Rule

Training means minimizing the loss L over $\mathbf{w} = (\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2)$.

Each step: $\mathbf{w} \leftarrow \mathbf{w} - \boldsymbol{\rho} \cdot \nabla_{\mathbf{w}} L(\mathbf{w})$.



forward pass: *compute*

backward pass: *differentiate*

Gradient of a composition is a product (chain rule). Backpropagation just automates that product for arbitrarily deep networks.

[Switch to Jupyter: `notebooks/act1_mnist_demo.ipynb`*]*

Surprise #1 — Universal approximation

Stone–Weierstrass-style result: Neural networks are **uniformly dense** among continuous fn's.

Theorem (**Universal Approximation** (Cybenko 1989; Hornik 1991))

Let $K \subset \mathbb{R}^n$ be compact, and let $\sigma : \mathbb{R} \to \mathbb{R}$ be any non-constant, bounded (necessarily non-linear), continuous function. Then finite-width feedforward networks of the form

$$g(\mathbf{x}) \;=\; \sum_{k=1}^{N} c_k \, \sigma\big(\mathbf{w}_k^{\top} \mathbf{x} + b_k\big)$$

are uniformly dense in the space $C_b(K)$ of continuous (bounded) real functions on K:
For every continuous f: $K \to \mathbb{R}$ and every $\varepsilon > 0$, some such g satisfies $\sup_{\mathbf{x} \in K} |g(\mathbf{x}) - f(\mathbf{x})| < \varepsilon$.

The mystery is *not* representation but *training*

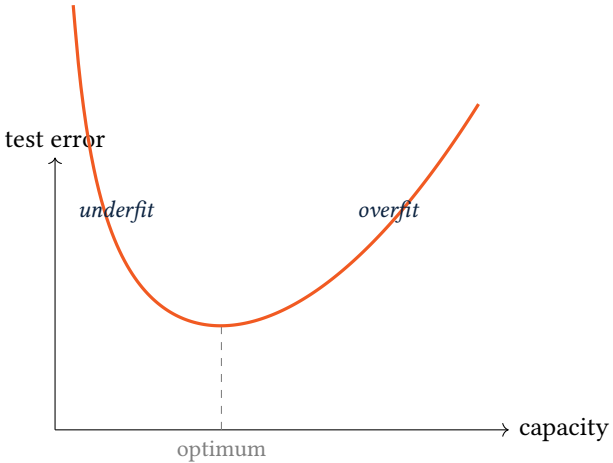
Neural networks can represent *any* continuous function. The unexplained question is *why* stochastic gradient descent on a non-convex loss surface, starting from random weights, finds a *good* representation in 30 seconds. Act II picks up this thread.

The classical wisdom: “Bias–Variance” Tradeoff

Every statistics textbook before ~ 2018:

- Too few parameters: Underfit (high bias).
- Too many parameters: Overfit (high variance).
- There is an optimal capacity in the middle.

Empirically: The “Test-Error vs. Capacity” curve is U-shaped.



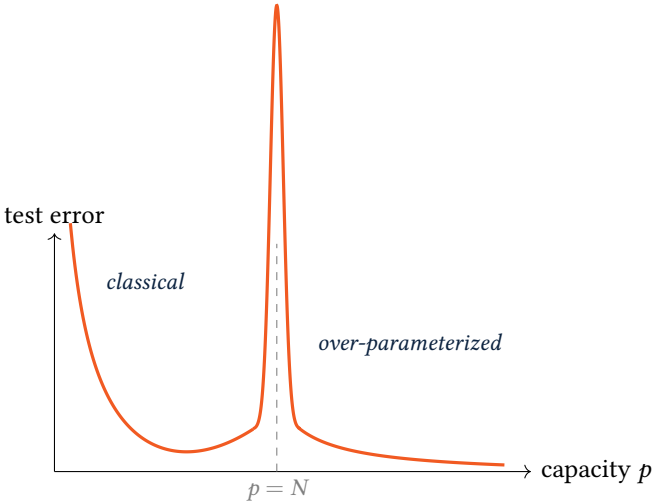
bias^2 + variance + noise = (test MSE).

What if we keep going *past* the threshold?

Classical theory/earlier wisdom:
Beyond a certain *interpolation threshold capacity*, test error *steadily increases*.

Reality: For high-capacity models that are *numerically stable*, test error spikes… then decreases again.

Double descent (Belkin et al. 2019).



[Switch to Jupyter: `notebooks/act2_double_descent_demo.ipynb`*]*

Surprise #2 — Over-parameterization works

- Past the interpolation threshold capacity, infinitely many parameter vectors can interpolate the training data essentially perfectly.
- `lstsq` (and, by analogy, SGD) selects the minimum-norm one.
- Among interpolators, minimum norm $\Leftrightarrow$ smoothness $\Leftrightarrow$ good generalization.

The *critical* caveat (mathematical insight)

Double descent appears only for numerically stable model classes.

- **Polynomials** of increasing/unbounded degree: **no** "fixed scale" $\Rightarrow$ **no** second descent.
- **Neural networks** [bounded 1-Lipschitz activations —ReLU, tanh, σ, GeLU— and $1/\sqrt{p}$ normalization]: scale stable $\Rightarrow$ double descent appears.

This is why modern architectures work hard to stay scale-stable:
Layer normalization, residual connections, careful initialization.

Open problem. Why does SGD typically finds the "smooth interpolator"?

From words to vectors

A transformer's input is not text. It is a sequence of vectors.

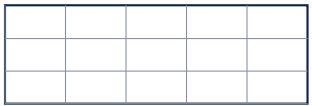
1. Vocabulary Size and Embedding Dimension. Fix vocabulary size **V** and embedding dimension **d**. ($\mathbf{V} = 50{,}257$, $\mathbf{d} = 768$ for GPT-2).
2. Tokenization. Fix convention to convert/encode Text $\to$ "token ID" sequence. Each token ID i is an integer $0 \leqslant i < V = 50{,}257$ corresponding to specific vocabulary item per a "real-world" meaning.
3. Context Window Size. An integer T ($T = 1{,}024$ for GPT-2): How many tokens are kept in active memory at a time.
3. Learned Token Embeddings. Learn a *meaningful/optimal* transformation token $\to$ vector ("embedding"): $0 \mapsto e_0$, $1 \mapsto e_1$, ..., $V-1 \mapsto e_{V-1} \in \mathbb{R}^{d=768}$.
4. Context matrix: A matrix $\mathbf{X}$ of size $T \times d$) input to GPT at a time.
   Order of the T rows is critical; repetitions possible.

"A neural network learns to..."

↓ tokenize

$[\,32, 17019, 3127, 4661, 284 \ldots\,]_{1,024}$

↓ embed each token in $\mathbb{R}^d$



sequence of vectors

$X \in \mathbb{R}^{T \times d}$

Attention: The entire magic trick

Every operation inside the transformer is a linear or nearly-linear map on the context matrix $\mathbf{X}$. Starting from context $X \in \mathbb{R}^{T\times d}$, obtain matrices $Q \in \mathbb{R}^{T\times d}$ ("query"), $K \in \mathbb{R}^{T\times d}$ ("key") and $V \in \mathbb{R}^{T\times d}$ ("value") each obtained as XW for some fixed matrix $W \in \mathbb{R}^{d\times d}$ (3 different such W). Then define the "Attention Operation" on X:

$$\mathbf{A} \;=\; \mathrm{Attn}(X) \;=\; \text{Row-softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V.$$

"Row-softmax" basically "normalizes" across rows to have positive entries with sum 1 ("probabilities"); these are used as coefficients of convex combinations of the rows of V.

- $A_{ij} = \text{Row-softmax}(QK^\top/\sqrt{d_k})_{ij}$ is the matrix of the attention pattern: How much token i "looks at" token j.

$$\sum_j A_{ij} = 1.$$

- Each token's output is a weighted average of the values V — weighted by *data-dependent* attention A.
- Stack 12–96 such layers, with MLPs in between. *That is the entire transformer.*

Inferences/deductions made by the Transformer are based on interpreting the *last row v of the (last) Value matrix V* via "softmax dot products" $p_i = \text{softmax}(v^\top e_i)$ as genuine probabilities to generate a (next) "new" token pseudo-randomly. This new token is "pushed" as the last row of the context matrix X, which loses its "top" row (loss of context).

[Switch to Jupyter: `notebooks/act3_attention_demo.ipynb`*] Then back to Ollama for the in-context learning callback.*

Surprise #3 — In-context learning

Put a few input–output examples in the prompt. The model infers the pattern and continues it — without changing a single weight.

```
hot : cold
big : small
day : ?
```

The model says `night`. Why?

- Nothing in the architecture says this *should* work.
- It emerges only in models above a certain size.
- Empirically (Garg et al. 2022; Akyürek et al. 2023): Transformers appear to run *gradient-descent-like algorithms inside their forward pass.*
- No complete theory yet. (One of the most active research areas in mathematical ML.)

Sutton's Bitter Lesson (2019). General methods + scale + compute beat clever inductive biases. In-context learning is one big demonstration of this.

Three surprises, three open problems

The three surprises

1. Universal approximation. Neural networks *can* represent any continuous function.
2. Double descent. With a *numerically stable* basis, bigger = better, past the threshold.
3. In-context learning. Large transformers learn at inference time, without any weight update.

The three open problems

1. *Why* does SGD on a non-convex loss find a *good* representation?
2. *Why* is the minimum-norm interpolator so often the right one?
3. *Why* does in-context learning emerge? What is the implicit algorithm?

Each of these is an open thesis topic. They are not "engineering details."
They are mathematics waiting to be done.

Tools to try tonight

Run a real LLM on your laptop

```
$ ollama install
$ ollama run llama3.2
```

Free GPU notebooks in your browser

`colab.research.google.com`

Hub of pretrained models

`huggingface.co`

The three frameworks

- **Keras** — easiest to learn
- **PyTorch** — most popular for research
- **JAX** — most mathematical, functional style

Three things to do this week

1. Install `ollama` tonight; run a model on your own machine.
2. Open the Colab notebooks from today; change a parameter; re-run.
3. Pick one open problem from the talk that bothers you, and read one paper about it.

Course materials and these notebooks:

`github.com/eduenez/MathAIspring2026UTSA`

¡Muchas gracias!

Thank you

Questions, comments, and pushback welcome.

Eduardo Dueñez
University of Texas at San Antonio
eduardo.duenez@utsa.edu