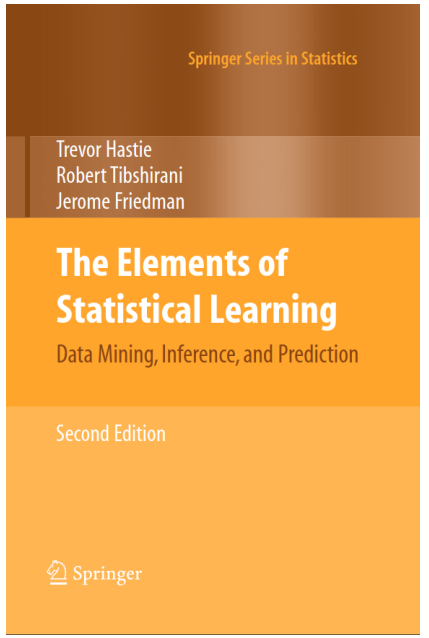
Stellenbosch
UNIVERSITY
IYUNIVESITHI
UNIVERSITEIT

Deep Learning

[Dr. Hanjo Odendaal]

The bible of machine learning

- Focuses on understanding the relationship between input variables (features) and output variables (responses).
    - **Supervised:** Learning with labeled data (e.g., regression, classification).
    - **Unsupervised:** Learning from unlabeled data (e.g., clustering, dimensionality reduction).
- Key Techniques:
    - Linear Methods (e.g., Linear Regression, Logistic Regression)
    - Nonlinear Methods (e.g., Decision Trees, SVMs, Neural Networks)
    - Model Assessment & Selection (e.g., Cross-Validation, Bias-Variance Tradeoff)
    - Ensemble Methods (e.g., Bagging, Boosting, Random Forests)



Introduction to prediction problems

In statistical modeling and machine learning, the core goal is often predicting an outcome y given inputs X:

$$y \sim f(X)$$

This can be unknown and complex (nonlinear, flexible) or specified with assumptions (linear, parametric).

Classical statistics - ("Is β different from zero?"):

- Focus on parameter estimation and inference.
- Models are often prescriptive: impose assumptions (e.g., linearity, normality).

Machine learning - "Can I predict y accurately from X?":

- Focus on prediction accuracy rather than interpretability.
- Models are often descriptive or agnostic: flexible function fitting without strong assumptions.

From ML to DL

Supervised ML Models:

- Input-output mappings learned from labeled data.
- Examples: LASSO (sparse linear models), Random Forests, Boosting.
- Emphasize interpretability and often require feature engineering.
- Often use basis expansions (e.g. polynomials or decision trees) to approximate $f(x)$

Deep Learning Models:

- Learn hierarchical representations automatically from raw input.
- Networks are typically composed of multiple layers of neurons (nonlinear basis functions).
- Nonlinearities (e.g. sigmoids, aka "activation functions") allow flexible approximations of complex functions.
- Require more data, computational power, and careful tuning.

Imagine you're trying to approximate a curve (remember the Gradient Descent image) using Lego blocks. **Ridge/Lasso** has fixed block, so it cannot 'learn'. **Boosting** lets you choose blocks adaptively as you build. **Neural nets** not only let you learn the best shapes, but also allow you to stack them in multiple layers, making intricate shapes possible.

What are basis expansions?

A basis expansion is a way to approximate a complex function $f(x)$ by combining simpler, known functions (called basis functions) like building blocks.

$$f(x) \approx \sum_{k=1}^{K} h_k(x)\theta_k$$

- Each $h_k(x)$ is a basis function - e.g., x, x^2, $\sin(x)$, or more generally $\phi_k(x)$.
- θ_k are weights that the model learns to best fit the data.

The idea is:

> We don't try to directly learn the complicated function $f(x)$. Instead, we express it as a weighted sum of simpler functions.

Boosting: Builds its basis functions as small decision trees (each $T_m(x)$ is a basis function).

Enter Deep Learning

Neural networks are **basis expansion machines** - but unlike LASSO or Boosting, they **learn** both the basis functions *and* how to stack them.

$$f(x) = \sum_k \theta_k \, \sigma(x^\top \beta_k) \quad \text{(1 hidden layer)}$$

- $\sigma(\cdot)$ is a **nonlinear activation** (e.g., sigmoid, tanh, ReLU).
- β_k are the learned weights defining each basis function.
- The network learns both the shape of $h_k(x)$ and how to combine them.

What makes it *deep*?

> Neural networks build **layers of basis expansions** - each layer transforms the input and passes it forward:

$$f(x) = \sigma^{(L)}(W^{(L)} \cdots \sigma^{(2)}(W^{(2)} \, \sigma^{(1)}(W^{(1)} x)))$$

Unlike boosting, which builds basis functions sequentially, deep learning composes them *hierarchically* . This enables it to model very complex structures like language, vision and time series.

Multilayer perceptron (MLP)

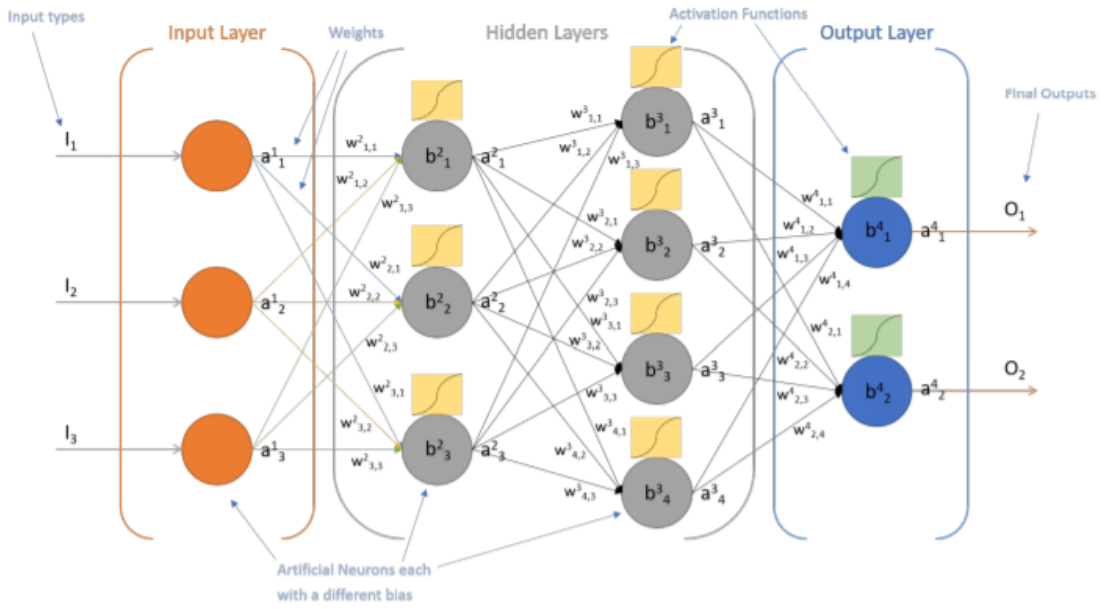
$$f(x) = \sigma^{(3)}\left(W^{(3)}\,\sigma^{(2)}\left(W^{(2)}\,\sigma^{(1)}\left(W^{(1)}x\right)\right)\right)$$

Input Layer ($x = [I_1, I_2, I_3]$)

- These orange nodes are the input features x.
- They are passed through the first layer via weights $W^{(1)}$.

Hidden Layer 1 (gray circles: b_1, b_2, b_3)

- Each neuron computes a linear combination: $z_j = x^\top w_j + b_j$
- Then applies an activation function σ (yellow blocks).
- This gives the first hidden layer activations: $a^{(1)} = \sigma(W^{(1)}x)$



Multilayer perceptron (MLP)

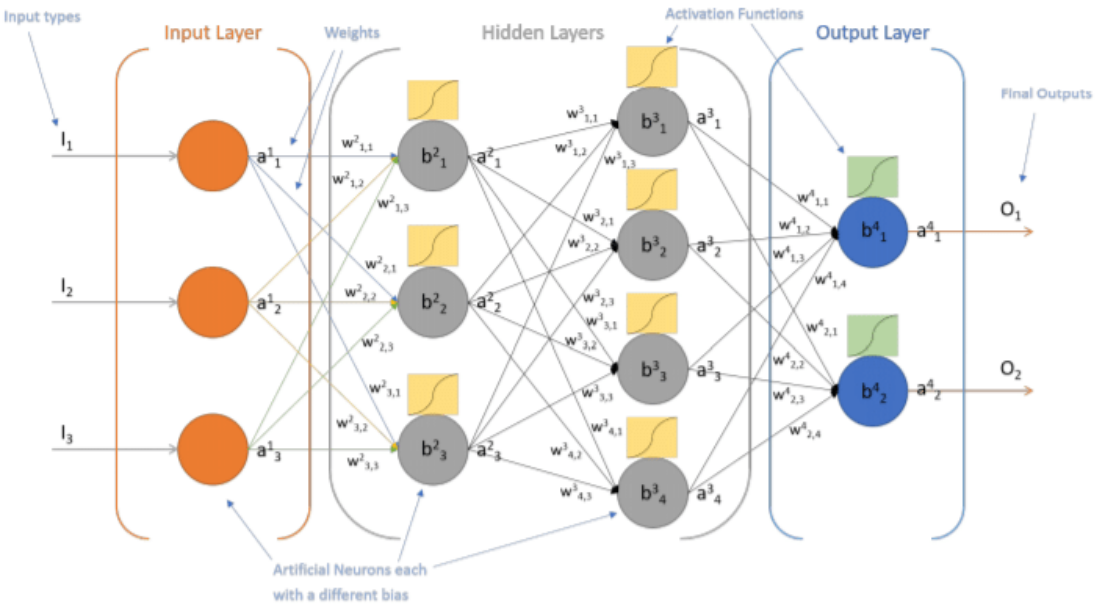
$$f(x) = \sigma^{(3)}\left(W^{(3)}\,\sigma^{(2)}\left(W^{(2)}\,\sigma^{(1)}\left(W^{(1)}x\right)\right)\right)$$

Hidden Layer 2

- The activations $a^{(1)}$ become the inputs to layer 2.
- The process repeats: linear combination $\rightarrow$ activation $\rightarrow$ new output $a^{(2)} = \sigma(W^{(2)}a^{(1)})$

Output Layer (blue circles: O_1, O_2)

- The final layer computes $a^{(3)} = f(x)$ - the model's output.
- If this is a classification task, the last activation $\sigma^{(L)}$ could be a softmax or sigmoid.



What is a Multi-Layer Perceptron (MLP)?

Think of an MLP as a **Lego tower** where each layer:

1. Builds a **linear block** (like regression)
2. Applies a **nonlinear twist** (activation)
3. Stacks this transformed block to the next layer

Linear combination:

$$z = x^\top w + b$$

- Like linear regression.
- w are the **weights** (just like β).
- b is a bias term (intercept).

Activation function:

$$a = \sigma(z)$$

- Nonlinear transformation: ReLU, sigmoid, etc.
- Gives us a **nonlinear building block** (basis function).

Where, z is like a prediction from linear regression. a is the non-linearly transformed output (the "activated" signal passed forward).

What is a Multi-Layer Perceptron (MLP)?

Think of an MLP as a **Lego tower** where each layer:

1. Builds a **linear block** (like regression)
2. Applies a **nonlinear twist** (activation)
3. Stacks this transformed block to the next layer

Feedforward pass:

- Data flows forward layer by layer.
- Each layer does: **linear + activation**, then passes the output on.

Backpropagation (learning):

- We compare the prediction $\hat{y}$ with the true y.
- Compute the **loss** (how wrong we are).
- Then adjust weights w **backward** through the network using gradients (chain rule).
- This is how the model learns - like nudging Lego blocks into better positions.

> MLP = A stack of linear regressions + nonlinear Lego adapters, trained via feedback.

How Do Neural Networks Learn?

Once we compute the prediction $\hat{y}$ via feedforward steps, we compare it to the actual y using a **loss function**.

Mean Squared Error (MSE)

$$\mathcal{L} = \frac{1}{2}(y - \hat{y})^2$$

We want to **minimize this loss** by adjusting the weights and we do this via the Chain Rule which is core to "Backpropagation". To update weights, we use **gradient descent**:

$$w \leftarrow w - \eta \frac{\partial \mathcal{L}}{\partial w}$$

- η = learning rate
- $\frac{\partial \mathcal{L}}{\partial w}$ = how changing w affects the loss

But w doesn't affect $\mathcal{L}$ directly - it flows through:

$$w \to z \to a = \sigma(z) \to \hat{y} \to \mathcal{L}$$

So we apply the **chain rule**:

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w}$$

How does a weight affect the loss?

Although we want to understand how the loss $\mathcal{L}$ depends on a weight w, the relationship isn't direct.

- The weight w affects the **linear combination** z
- Which passes through an **activation function** to become a
- Which is used to compute the **prediction** $\hat{y}$
- Which is finally compared to the true y using the **loss** $\mathcal{L}$

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial w}$$

To understand how the loss $\mathcal{L}$ depends on a weight w, we break it into parts:

$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial \hat{y}}(\text{how loss changes with prediction}) \times \frac{\partial \hat{y}}{\partial a}(\text{how prediction changes with activation}) \times \frac{\partial a}{\partial z}(\text{how activation changes with } z) \times \frac{\partial z}{\partial w}(\text{how } z \text{ changes with } w)$

Each part is a link in the chain from output back to input.

> This process is called **backpropagation** - it's just a smart way of using the chain rule to compute how each weight in the network contributes to the final error, so we can adjust it and learn.

Putting it all together

- The chain rule is a mathematical tool we use to compute gradients - i.e., to find out how a change in a weight affects the loss: $\frac{\partial \mathcal{L}}{\partial w}$.
- Gradient descent is the optimization algorithm that uses those gradients to actually update the weights:

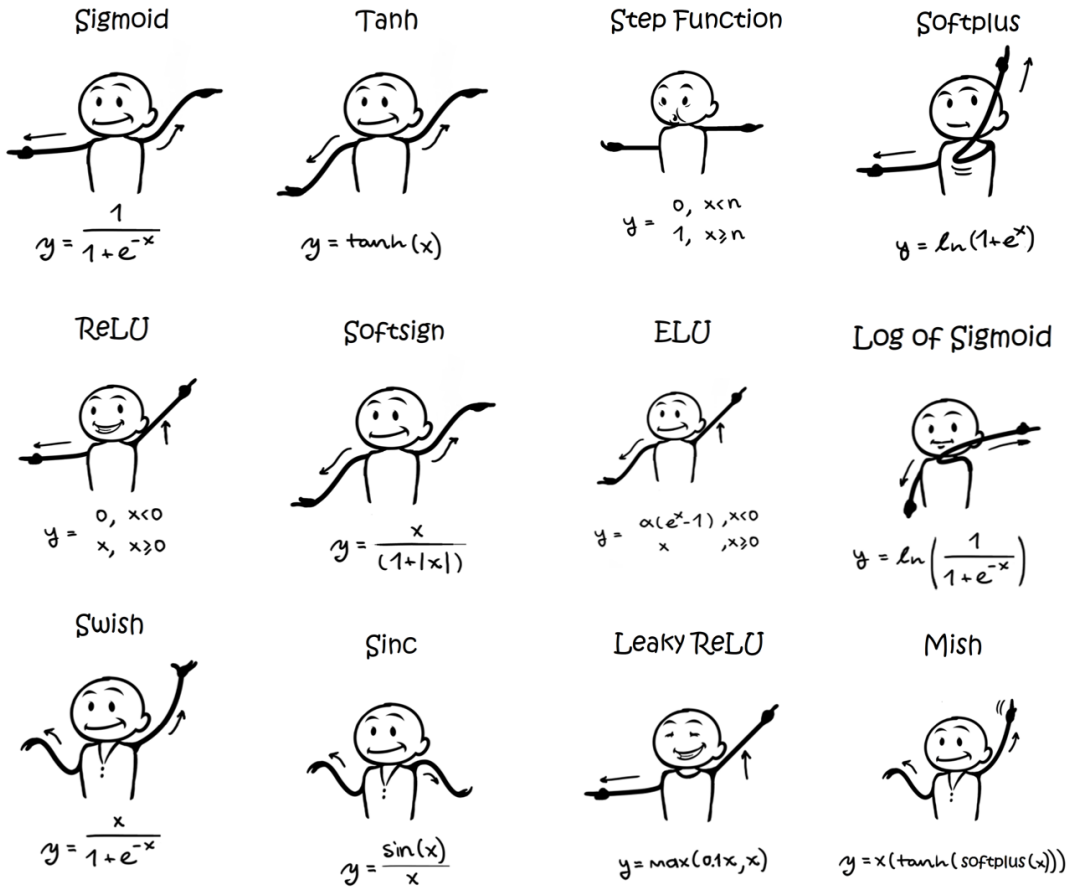
$$w \leftarrow w - \eta \frac{\partial L}{\partial w}$$

Concept	Role
Chain rule	Helps us calculate the gradient of the loss w.r.t. each weight
Backpropagation	Systematic application of the chain rule across all layers
Gradient descent	Uses those gradients to update the weights and learn

You can think of it like this:

- 🔧 Chain rule = the tool to compute how to fix each Lego block
- 🧠 Gradient descent = the actual fixing action that adjusts them

Activation fun(ctions)



https://sefiks.com/2020/02/02/dance-moves-of-deep-learning-activation-functions/

Activation fun(ctions)

Activation functions add **nonlinearity** to neural networks - essential for learning complex patterns. There are many, but here are the top 5:

Name	Formula	When to Use
ReLU	$\max(0, x)$	Most common default; fast to compute; good for hidden layers
Sigmoid	$\frac{1}{1+e^{-x}}$	For binary classification outputs; can squash inputs into $[0, 1]$
Tanh	$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	Like sigmoid but outputs in $[-1, 1]$; used when zero-centered activations help
Softmax	$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$	For *multiclass* classification output layer (probability distribution)
Leaky ReLU	$\begin{cases} x & x > 0 \\ \alpha x & x \leq 0 \end{cases}$	Fixes "dead neuron" problem in ReLU; used in deeper networks

> Rule of thumb: use **ReLU** in hidden layers, and **sigmoid**/**softmax** in output layers depending on the task.

Activation fun(ctions) in Time-Series

Unlike feedforward networks, time series models reuse weights across time and maintain state, so the choice of activation functions affects both:

- (1) How memory is updated, (2) How the network outputs a prediction at each time step

Function	Formula	Where It's Used	Why It's Used
Tanh	$\tanh(x)$	Hidden state update (h_t) in RNNs, LSTM, GRU	Keeps values in $[-1, 1]$, stabilizes gradients and memory over time
Sigmoid	$\frac{1}{1+e^{-x}}$	Gates in LSTM/GRU: forget, input, update, output	Outputs in $[0, 1]$ - acts like a soft switch to control memory flow
ReLU	$\max(0, x)$	Occasionally in RNNs or hybrid models' feedforward components	Promotes sparsity but risks unstable gradients in long sequences
Linear (None)	$f(x) = x$	Output layer for **regression/forecasting**	No transformation - outputs real values directly (e.g., next value prediction)

Stellenbosch
UNIVERSITY
IYUNIVESITHI
UNIVERSITEIT

Algo friends ⚒

Recurrent models: your time-aware algo friends

Some problems have **memory** - where what came *before* matters. These models are built to handle *sequences*, like time series, speech and text.

RNN (Recurrent Neural Network)

- Remembers **short-term patterns**
- Uses a hidden state updated at each time step: $h_t = f(x_t, h_{t-1})$
- Struggles with **long-term memory** due to vanishing gradients

LSTM (Long Short-Term Memory)

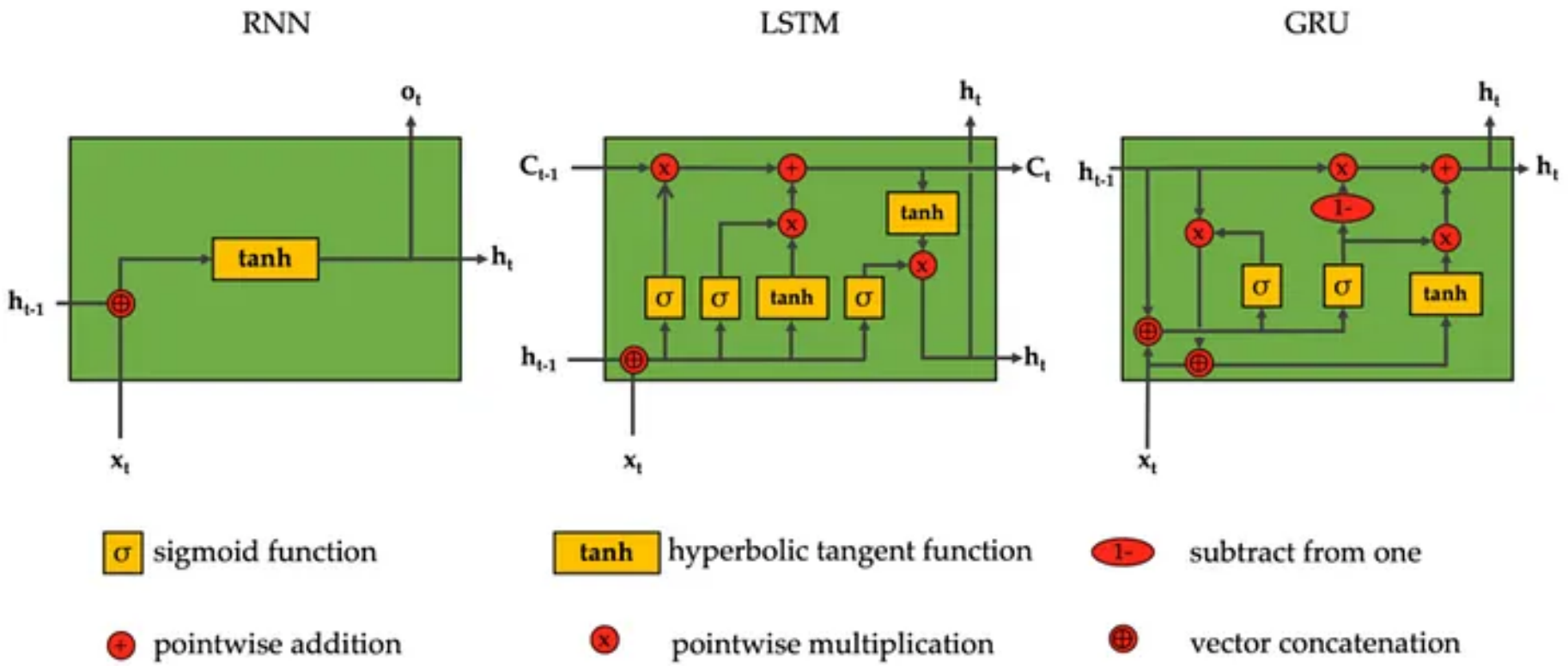
- Designed to capture **long-term dependencies**
- Has separate memory cell and multiple gates: input, forget, output
- Great for tasks with complex or distant time relationships (e.g., inflation regimes, language)

GRU (Gated Recurrent Unit)

- A more efficient cousin of LSTM
- Uses *fewer gates* (update & reset) to control memory
- Faster to train, good when *data is limited or simpler*

> These models don't just look at *what is*, but *what came before*. Memory is built into their design.

Recurrent models: your time-aware algo friends



https://medium.com/@hassaanidrees7/rnn-vs-lstm-vs-gru-a-comprehensive-guide-to-sequential-data-modeling-03aab16647bb

Understanding the RNN

🧱 Lego Analogy:

- Think of each time step as a *Lego* block.
- The RNN passes a memory block (h_{t-1}) from one block to the next.
- It combines the current input x_t and the past state h_{t-1} to update its memory h_t.

But there's *only* one gate, a **tanh** nonlinearity.

So if a pattern depends on far-back time steps, the signal fades - this is the *vanishing gradient problem*.

What the Math Means:

- Inputs: x_t (current input), h_{t-1} (previous hidden state)
- The RNN computes:

$$h_t = \tanh(W_h h_{t-1} + W_x x_t + b)$$

- Only one transformation → **tanh** squashes the result between $[-1, 1]$

Problem:

- As we move through time, gradients shrink
- This makes learning **long-term patterns** very difficult

> RNNs are elegant and simple, but forgetful - they're best for **short-term memory** tasks.

Understanding the LSTM

🧱 Lego Analogy:

- LSTM blocks are like **deluxe Legos** - they come with **internal memory** (C_t) and multiple **gates**.
- The cell decides:
  - What to *forget* from the past,
  - What *new information* to add,
  - And what to *output* to the next step.
- This makes LSTM great at remembering patterns over *long distances* - like stacking many blocks in a row.

- Inputs: x_t, h_{t-1} (hidden state), C_{t-1} (cell state)
- It uses **three gates**:
  - **Forget gate**: $\sigma(W_f x_t + U_f h_{t-1} + b_f)$
    Decides what part of C_{t-1} to forget
  - **Input gate**: $\sigma(W_i x_t + U_i h_{t-1} + b_i)$
    Controls what new info enters the cell
  - **Output gate**: $\sigma(W_o x_t + U_o h_{t-1} + b_o)$
    Determines what to send out as h_t
- **Tanh** is used to propose new content and to squash outputs. The **cell state** C_t carries memory directly through time, with minimal disruption

> LSTMs are memory champions - they keep what matters, forget what doesn't and learn what to remember.

Understanding the GRU

🧱 Lego Analogy:

- GRU blocks are like *streamlined Legos*. Fewer parts, faster to assemble.
- GRUs *combine* the memory cell and hidden state (h_t) into one.
- Only **two gates** control everything:
  - How much of the past to keep
  - How much new info to add

> GRUs are efficient learners - they blend memory and new info using just two gates, making them fast and effective.

- Inputs: x_t, h_{t-1} (previous hidden state). It uses **two gates**:
  - **Update gate**: $z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$ Decides how much of the past to keep vs. overwrite
  - **Reset gate**: $r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$
    Controls how much of the past to forget before mixing with current input
- Then it computes a **candidate state**:

$$\tilde{h}_t = \tanh(W_h x_t + U_h(r_t \cdot h_{t-1}) + b_h)$$

- And the final output:

$$h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t$$

From RNNs to Transformers

Why Move Beyond RNNs?

- RNNs, LSTMs and GRUs process sequences *step-by-step*
- This makes them *slow* and hard to parallelize
- They also *struggle* with very long sequences. Even LSTMs forget eventually

Enter Transformers:

- Transformers *remove recurrence entirely*
- They process *entire sequences at once* using **attention**
- Instead of remembering the past, they **look at all time steps directly**

The Key Shift:

> Memory → Attention
> Sequence order → Position encoding
> Step-by-step → Parallel processing

Why it matters:

- Transformers power **ChatGPT**, **BERT**, **GPT** and other modern AI models
- They're faster, scalable and better at learning **long-range dependencies**

Cited ~180,000 times: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf