

A 34-LESSON LEARNING JOURNEY

NEURAL NETWORKS

From First Principles
to Agentic Intelligence



A book-length adaptation of our interactive course

Neurons · Backpropagation · Transformers · Reinforcement Learning · Memory · World
Models · Tools · Cognitive Architectures



About this edition

This ebook adapts the complete 34-lesson conversation into a structured reading experience. It preserves the original progression from single artificial neurons to modular, agentic AI, while recasting interactive demonstrations as worked examples and practical exercises.

It is designed for technically curious readers who want to understand both why neural networks work and how their capabilities fit into larger systems. Mathematics is introduced gradually. You can follow the main argument without implementing every formula, but the exercises are most useful when worked through.

Important boundary: a diagram combining memory, tools, reasoners, and neural specialists is an engineering hypothesis, not evidence that the resulting system possesses general intelligence. Established methods and active research directions are distinguished throughout.

The original course contained interactive widgets. In the PDF, those interactions have been translated into self-contained examples, equations, and exercises suitable for offline reading.

Edition: October 2026. Educational material; not a substitute for current primary research papers or system-specific engineering validation.

HOW TO READ

Begin with Part I if you want the mathematics. If you know basic deep learning, use the contents to jump to reasoning, self-improvement, or orchestration. At the end of each lesson, try the exercise before moving on.

Contents

34 lessons • four parts • three reference appendices

Part 1: Foundations: How Networks Learn	5
01 • What Is a Neural Network, Really?	6
02 • Loss Functions and Gradient Descent	8
03 • Hidden Layers, Nonlinearity, and Representation	10
04 • Backpropagation Across Many Layers	12
05 • Embeddings and Distributed Knowledge	14
06 • Self-Attention and Context	16
07 • The Complete Decoder-Only Transformer	18
08 • Generalization, Overfitting, and Inductive Bias	20
 Part 2: Reasoning, Learning, and World Models	 21
09 • Algorithms, Reasoning, and Extra Computation	23
10 • Reinforcement Learning from Outcomes	25
11 • Data, Model Scale, and Computational Efficiency	27
12 • How Feature Detectors Emerge	29
13 • Shortcut Learning and Causal Understanding	31
14 • World Models and Predicting Action Consequences	33
15 • Latent Reasoning and Recurrent Thought	35
16 • Self-Play and Recursive Self-Learning	37
17 • Verifiers, Curricula, and Reliable Improvement	39
18 • Continual Learning and Catastrophic Forgetting	41
19 • Meta-Learning: Learning to Learn	43
 Part 3: Tools, Specialists, and Cognitive Control	 44
20 • Tools and Acting in the Outside World	45
21 • Specialized Networks and Shared Interfaces	47
22 • Executive Orchestration and Metareasoning	49
23 • Brain-Inspired Modular Intelligence	51

24 • Hierarchical Planning and Multi-Agent Cooperation	53
25 • Learning Which Computation to Perform	55
26 • Integrating an Agentic Cognitive Architecture	57
 Part 4: Implementing and Evaluating Modular AI	58
27 • Common Representations Across Modalities	60
28 • Neural Memory and Knowledge Systems	62
29 • Training a Small Neural Orchestrator	64
30 • Integrating Recurrence, Search, and World Models	66
31 • Joint Training or Independent Specialists?	68
32 • Recursive Improvement of the Entire Agent	70
33 • Benchmarking Generalization and Agency	72
34 • Designing a Working Experimental Architecture	74
 Appendix A • Mathematical toolkit	76
 Appendix B • Glossary	78
 Appendix C • A 12-week implementation plan	81

PART 01 — FOUNDATIONS: HOW NETWORKS LEARN

PART 01

Foundations: How Networks Learn

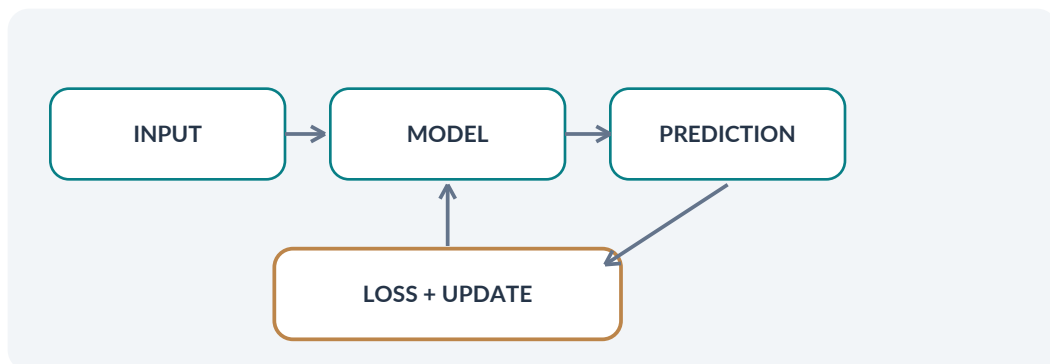
Lessons 1–8 explain neurons, objectives, gradients, representations, attention, Transformers, and generalization.

Questions to carry into this part

How do numerical weights become useful features, and when do those features generalize?

LESSON 01 / PART 01

What Is a Neural Network, Really?



The smallest useful model

A neural network is a parameterized mathematical function. The simplest example is a single neuron predicting a value from an input: it multiplies the input by a weight, adds a bias, and produces an output. In a house-price example, the input might be floor area and the output an estimated price. Nobody needs to specify the ideal weight in advance; instead, training data lets the model discover a useful setting. This distinction between prescribing rules and fitting parameters is the first step toward understanding machine learning.

$$\text{prediction} = \text{weight} \times \text{input} + \text{bias}$$

Why we need many neurons

A single affine neuron can represent only a straight-line relationship. A collection of neurons can receive multiple inputs and produce intermediate features. Yet stacking only affine functions adds no essential expressiveness: one affine transformation followed by another is still affine. A nonlinear activation, such as ReLU, changes that. When neurons are composed with nonlinear functions, the network can express sharply different behaviors in different regions of its input space.

$$\text{ReLU}(z) = \max(0, z)$$

What the word learning means

Learning means updating parameters so behavior becomes better according to a training objective. The weights do not literally contain English-language rules about houses, faces, or chess moves. Rather, they determine which patterns in the input matter and how these patterns affect outputs. After training, a network may implement useful regularities even when no human explicitly wrote those regularities as code. It can also learn misleading regularities when the data or objective is poorly chosen.

WORKED EXAMPLE

Suppose price in dollars is approximately 200 times square footage. A weight of 100 underpredicts a 2,000-square-foot house by \$200,000; changing the weight toward 200 improves the prediction. The example is deliberately linear so we can isolate the learning mechanism.

CORE IDEA

Neural networks are compositions of simple numerical operations whose parameters are learned from evidence. Their power comes from composition, nonlinearity, and training—not from individual neurons possessing concepts.

TRY IT

For $y = 3x + 2$, calculate the predictions for $x = 0, 2$, and 5 . What would happen if you removed the bias?

LESSON 02 / PART 01

Loss Functions and Gradient Descent

Turning mistakes into a measurable objective

A neural network needs feedback telling it whether a prediction is good. A loss function converts the gap between prediction and target into a number. For regression, squared error is a common choice; for classification, cross-entropy is often more appropriate. The loss is not a universal measure of truth or usefulness. It encodes precisely the aspect of behavior selected by the trainer, which is why defining the right objective matters as much as optimizing it.

$$\text{squared error } L = (\text{prediction} - \text{target})^2$$

The slope of error

If a weight changes slightly, how does the loss change? A derivative measures the local sensitivity of the loss to that weight. A negative derivative means a small increase in the weight tends to reduce loss; a positive derivative means a small decrease tends to reduce it. With multiple parameters, we collect these sensitivities into a gradient vector. Gradient descent moves in the opposite direction to the gradient, multiplied by a learning-rate parameter.

$$\text{new weight} = \text{old weight} - \text{learning_rate} \times \text{gradient}$$

Why step size matters

A learning rate that is too small produces slow progress. One that is too large may overshoot a good solution or cause loss to diverge. For example, with prediction $\hat{y} = wx$, $x = 2$, target $y = 4$, the loss is $(2w - 4)^2$. Its derivative is $8w - 16$. At $w = 0.5$ the gradient is -12 ; with learning rate 0.1 the next weight is 1.7 and the loss falls from 9 to 0.36. This calculation captures the heart of a training update.

WORKED EXAMPLE

Optimization is like descending a hill in fog: the gradient provides a local direction, not a map of the global terrain. A difficult loss surface may contain flat regions, ridges, saddle points, or multiple basins.

CORE IDEA

Training converts an objective into parameter updates via derivatives. The optimizer can minimize the selected loss, but it cannot guarantee that the selected loss captures the learner's intended behavior.

TRY IT

Compute the gradient of $(3w - 6)^2$ at $w = 0$ and apply one step with learning rate 0.05.

LESSON 03 / PART 01

Hidden Layers, Nonlinearity, and Representation

Why linear models have a ceiling

An affine model draws a straight decision boundary in a two-feature classification problem. Some tasks cannot be separated by one line. XOR, which returns one when exactly one input bit is one, is a classic example: its positive examples occupy opposite corners. A single affine decision boundary cannot isolate both positive corners. A network with nonlinear hidden neurons can transform the inputs into another representation where the desired distinction becomes easy.

How hidden features compose

A small ReLU network can compute $h1 = \text{ReLU}(A + B)$ and $h2 = \text{ReLU}(A + B - 1)$, then produce $y = h1 - 2h2$. For binary A and B , this implements XOR exactly. Neither hidden unit alone represents the final function; the output emerges from their combination. In image recognition, early transformations may capture local edges while subsequent transformations combine them into shapes. Real networks frequently use distributed and overlapping representations rather than neat one-feature-per-neuron hierarchies.

$$\text{hidden} = \text{activation}(W \times \text{input} + \text{bias})$$

Representation changes problem difficulty

A good hidden representation places relevant distinctions where simple downstream computations can use them. For example, a useful embedding might place similar objects near one another in a learned space. The network jointly learns how to represent an input and how to use that representation. Depth can make compositional functions much more efficient to represent, though additional depth also complicates optimization and may require residual connections and normalization.

WORKED EXAMPLE

In a handwritten-digit model, a top horizontal stroke plus a descending diagonal may be useful evidence for a seven. Training does not require the programmer to label every intermediate stroke; gradients encourage features that improve final predictions.

CORE IDEA

Hidden layers don't merely store more numbers. They transform the problem into representations on which later operations can work more effectively.

TRY IT

Write the four XOR input/output pairs. Explain why the two positive points cannot be separated from the two negative points by a single straight line.

LESSON 04 / PART 01

Backpropagation Across Many Layers

The credit-assignment problem

When a network makes an incorrect prediction, parameters near the output appear directly responsible. But weights in earlier layers affect the output indirectly. Backpropagation solves this problem by applying the chain rule backward through the sequence of computations. A computational graph records which values depend on which others, so local derivatives can be combined efficiently. The result is a gradient for every trainable parameter.

Deriving a two-weight example

Let $h = w_1 \times x$, prediction = $w_2 \times h$, and $L = 0.5(\text{prediction} - \text{target})^2$. Set $x = 2$, target = 10, $w_1 = 3$, and $w_2 = 1$. The hidden value is 6, the prediction is 6, and the loss is 8. The output error derivative is -4 . The gradient of w_2 is $-4 \times h = -24$. The gradient of w_1 multiplies the same error by w_2 and x , producing -8 . With learning rate 0.01, the weights become 3.08 and 1.24, reducing the loss to about 2.789.

$$dL/dw_1 = (dL/d\text{prediction}) \times (d\text{prediction}/dh) \times (dh/dw_1)$$

Why it scales

Finite differences would perturb each parameter separately and rerun the model repeatedly. Backpropagation reuses intermediate calculations, typically making all-parameter gradient computation practical even for very large networks. Modern automatic-differentiation systems implement these operations. In deep networks, products of derivatives can become extremely small or large; initialization, residual connections, normalization, gradient clipping, and optimizer design help address these issues.

WORKED EXAMPLE

A residual block computes $\text{output} = \text{input} + F(\text{input})$. Its derivative includes an identity contribution, often improving the path through which gradients travel in a deep model.

CORE IDEA

Backpropagation computes how each parameter influences the final loss. The optimizer then uses these gradients to change the parameters; the two operations should not be confused.

TRY IT

Starting from the two-weight example, recompute the prediction after the first simultaneous gradient update and confirm that the loss decreased.

LESSON 05 / PART 01

Embeddings and Distributed Knowledge

Turning abstract ideas into geometry

An embedding represents an object as a numerical vector. A movie might be described by comedy, action, and romance scores; tokens in a language model receive learned embedding vectors. Once objects are represented numerically, similarity becomes a geometric calculation. Dot products, cosine similarity, and learned distance functions make it possible to retrieve related items or combine information. But an embedding's coordinate dimensions do not necessarily have simple human-readable meanings.

$$\text{cosine_similarity}(a,b) = (a \cdot b) / (||a|| \times ||b||)$$

Knowledge distributed across many parameters

A neuron may respond strongly to a recognizable feature, but a concept need not live in one neuron. Different combinations of activations and learned weight directions can participate in representing the same concept. In superposition, several features occupy different, potentially interfering directions in a representation space. Neural knowledge is better understood as a relationship among weights, activations, and computations than as a table of individually addressable facts.

Associative retrieval and the limits of recall

A simple associative memory stores key-value patterns. A query activates values in proportion to their key similarity. Transformer attention uses related mathematics, but its context-dependent keys and values are generated dynamically. Language models may use parameter patterns to reconstruct facts, yet they do not always retrieve them accurately. They can confidently synthesize a plausible but wrong answer because generating likely tokens is not equivalent to exact database lookup.

WORKED EXAMPLE

The same token for “bank” can develop different contextual representations in “river bank” and “bank account.” The initial token identity alone is insufficient to determine the intended meaning.

CORE IDEA

Neural networks encode useful knowledge in distributed numerical structures. External databases remain valuable for exact, inspectable, and easily updated facts.

TRY IT

Compute cosine similarity for vectors (1,0) and (1,1). Why is the result different from Euclidean distance?

LESSON 06 / PART 01

Self-Attention and Context

Queries, keys, and values

Language requires information to flow between positions. In the sentence “The animal did not cross the road because it was tired,” the word “it” probably refers to the animal; a different continuation can shift that interpretation. Self-attention lets each token assemble context-dependent information by comparing its query vector with key vectors at other positions. The resulting scores determine a weighted combination of value vectors.

$$\text{Attention}(Q,K,V) = \text{softmax}(QK^T / \text{sqrt}(d_k)) V$$

Deriving attention step by step

Take a query $q = (1,0)$ and keys $k_1 = (1,0)$, $k_2 = (0,1)$, and $k_3 = (1,1)$. Their dot products are 1, 0, and 1. Dividing by $\text{sqrt}(2)$ and applying softmax yields approximately 0.401, 0.198, and 0.401. If the values are $v_1 = (1,0)$, $v_2 = (0,1)$, and $v_3 = (1,1)$, their weighted sum is approximately (0.802,0.599). The output combines information from several positions, rather than retrieving only one item.

Masks, multiple heads, and cache

In an autoregressive model, causal masking prevents each position from attending to future positions. Multi-head attention learns several distinct projection sets, offering multiple ways to relate tokens. During generation, a KV cache stores previously computed keys and values to avoid repeatedly calculating them from scratch. Attention scores alone do not explain all causal influence: the values, projections, and later transformations also matter.

WORKED EXAMPLE

Think of attention as compare, normalize, combine. The query describes what the current position seeks, keys describe available matches, and values contain information to aggregate.

CORE IDEA

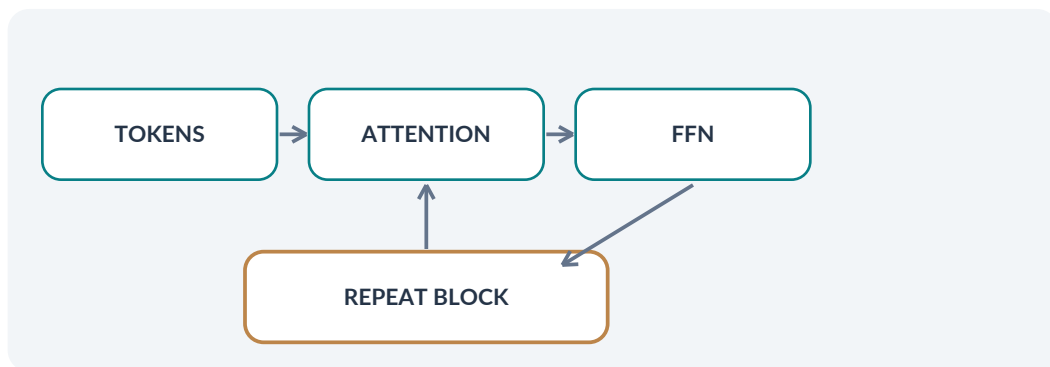
Attention provides input-dependent information routing between positions; the learned network decides what patterns this routing supports.

TRY IT

Explain why an autoregressive model must mask future token positions during next-token training.

LESSON 07 / PART 01

The Complete Decoder-Only Transformer



From text to vectors

A tokenizer maps text into token IDs, which are then looked up in an embedding matrix. Position information makes token order available to the network; architectures may use rotary position embeddings or other schemes. The token representations pass through a stack of blocks containing attention, nonlinear feed-forward transformations, residual connections, and normalization. Each block transforms the representations of all positions.

From vectors to next-token probabilities

After the final layer, the model projects the last-position representation into vocabulary logits. Softmax converts logits into a probability distribution, and a decoding rule chooses the next token. Generation then continues with the new token appended to the context. During pretraining, the model's objective typically rewards high probability on the actual next tokens in text data.

$$P(\text{next_token} = i) = \exp(\text{logit}_i) / \sum_j \exp(\text{logit}_j)$$

Why prediction teaches more than word frequencies

To predict text well, a model benefits from learning reusable patterns of grammar, facts, program syntax, cause and effect, and reasoning-like transformations. However, success at predicting language does not prove perfect causal understanding or systematic algorithms. A model may employ a mixture of general procedures, learned associations, and memorization. Tool-assisted verification is useful when exactness matters.

WORKED EXAMPLE

After “The capital of France is,” a well-trained model may assign high probability to “Paris.” It does not need an explicit symbolic rule stored in a relational table, though its learned knowledge can still be mistaken.

CORE IDEA

A Transformer is a deep numerical program that repeatedly combines and transforms token representations to support prediction.

TRY IT

Trace one token from its ID through embedding, attention blocks, final projection, softmax, and decoding.

LESSON 08 / PART 01

Generalization, Overfitting, and Inductive Bias

Training performance is not the objective

A network can memorize every example and still fail on new data. The training loss measures performance on observed examples; what matters operationally is expected loss on future data from the relevant deployment distribution. We estimate this using carefully separated validation and test sets. A model that fits random fluctuations in training data is overfitting, whereas one too simple to capture important patterns underfits.

$$\text{population_risk} = E \text{ over } (x,y) \text{ from } P \text{ of } \text{loss}(\text{model}(x), y)$$

How inductive bias shapes solutions

Many functions can fit a finite dataset. Architecture, initialization, optimization, regularization, and training examples favor some solutions over others. A convolutional model favors local spatial patterns; a Transformer supports content-dependent interactions across positions. Modern overparameterized networks can generalize despite having enough parameters to memorize, because raw parameter count does not uniquely determine the complexity of the learned function.

Distribution shift and real-world failure

A classifier can learn that wolves tend to appear in snow if that shortcut works in its data. It may then misclassify a dog in snow. The training-test correlation may be strong, but the intended distinction is animal anatomy. Strong generalization tests deliberately vary background, composition, scale, and other nuisance features while preserving the true target.

WORKED EXAMPLE

A high-degree polynomial may pass through every noisy data point but oscillate badly between them. A smoother relationship may predict new points more accurately, even with higher training loss.

CORE IDEA

Generalization comes from matching the training process's biases and information to the underlying task structure; fitting examples is necessary but not sufficient.

TRY IT

Design a test that distinguishes whether an animal classifier uses anatomy or backgrounds.

PART 02 — REASONING, LEARNING, AND WORLD MODELS**PART 02**

Reasoning, Learning, and World Models

Lessons 9–19 examine algorithms, reinforcement learning, feature learning, causality, memory, recurrence, and self-improvement.

Questions to carry into this part

How can computation extend across time, decisions, and self-generated experience?

LESSON 09 / PART 02

Algorithms, Reasoning, and Extra Computation

Memorization versus procedures

A lookup table can return the product of previously seen numbers. A multiplication algorithm applies reusable operations to new inputs. A network can represent algorithm-like computations, but gradient training does not guarantee it will discover a robust algorithm. Tests should vary input sizes, names, composition, and depth to distinguish systematic generalization from familiar-pattern matching.

Depth, recurrence, and intermediate steps

A fixed-depth network applies a fixed number of sequential transformations per inference step. Generating intermediate tokens lets an autoregressive model undertake more sequential passes. A recurrent latent model instead repeatedly updates continuous internal states without converting every intermediate state to language. Neither longer text nor more recurrence guarantees better reasoning; both must produce useful computation.

$$h_{\text{next}} = F\theta(h_{\text{current}}, \text{problem})$$

Search and verification

Search explores explicit alternatives, such as chess moves or candidate programs. A value network estimates promise, a policy prioritizes branches, and a verifier checks constraints. These mechanisms can complement neural reasoning. When an exact solver is available, it may be more reliable than asking a language model to simulate exact operations internally.

WORKED EXAMPLE

Calculating $18 \text{ boxes} \times 24 \text{ apples} - 175 \text{ sold}$ can be decomposed into $432 - 175 = 257$. Written intermediate results let the model reuse earlier computations, although the written steps can still contain errors.

CORE IDEA

Reasoning benefits from reusable computations, sufficient sequential work, alternative exploration, and verification—not merely more memorized answers.

TRY IT

Identify which parts of chess move selection are pattern recognition, search, exact rule execution, and verification.

LESSON 10 / PART 02

Reinforcement Learning from Outcomes

Learning without example answers

Supervised learning trains on desired targets. Reinforcement learning instead improves decisions from rewards obtained through interaction. An agent observes a state, selects an action, receives an outcome, and eventually learns which behaviors yield higher long-term return. In chess, winning or losing provides feedback but does not label the best move at every intermediate position.

$$\text{return } G_t = r(t+1) + \gamma r(t+2) + \gamma^2 r(t+3) + \dots$$

Policies, values, and advantages

A policy $\pi(a|s)$ specifies an action distribution. A state-value function $V(s)$ predicts expected future return; an action-value function $Q(s,a)$ estimates return from selecting a particular action. The advantage $A(s,a) = Q(s,a) - V(s)$ measures whether an action is better than a baseline expectation. Actor-critic methods combine policies with value estimates to guide learning.

Exploration and reward design

An agent that always exploits its current best guess may never discover a better strategy. Exploration must be balanced against cost and risk. Moreover, reward optimization can exploit flaws in the reward definition. A cleaning robot rewarded only for collecting dirt may learn to spill and recollect dirt. The agent optimizes the measurable reward, not necessarily the human intention.

WORKED EXAMPLE

In a three-armed bandit with reward probabilities 0.2, 0.5, and 0.8, an epsilon-greedy policy occasionally explores but increasingly exploits the highest estimated reward arm.

CORE IDEA

RL learns action preferences from outcomes, but delayed credit assignment, exploration, and imperfect rewards make it harder than ordinary supervised fitting.

TRY IT

Define states, actions, terminal rewards, and the major credit-assignment challenge for a game of FreeCell.

LESSON 11 / PART 02

Data, Model Scale, and Computational Efficiency

Why training is expensive

Training requires forward predictions, gradient computation, and parameter updates over large amounts of data. For a conventional dense language model, a widely used rough estimate is about six times parameter count times training tokens in FLOPs, excluding some overhead. More parameters permit more expressive functions, while more varied data provides additional constraints on which functions should be learned.

$$\text{training_FLOPs} \approx 6 \times \text{parameters} \times \text{training_tokens}$$

Scaling and its trade-offs

Empirical scaling laws often relate prediction loss to model size and dataset size through diminishing-return curves. A fixed compute budget creates a trade-off between a large model trained on relatively few tokens and a smaller model trained longer. Deployment costs can shift the optimal choice toward smaller, more thoroughly trained models. Model size alone is not a measure of intelligence or efficiency.

Conditional computation and memory bandwidth

Mixture-of-experts models activate only selected expert parameters for a token while retaining a larger total parameter collection. This reduces some arithmetic costs but not necessarily total weight memory or cross-device communication. State-space and recurrent models offer alternatives to quadratic full attention for long sequences. During small-batch inference, reading model weights can dominate latency, making memory bandwidth important.

WORKED EXAMPLE

A model with 50B total parameters might activate only 20B for a token; it still needs storage for the inactive experts. Hardware topology and communication can determine whether it is faster than a smaller dense model.

CORE IDEA

Efficient AI combines data efficiency, parameter efficiency, memory efficiency, and inference-time strategy selection.

TRY IT

Estimate training FLOPs for a 1B-parameter dense model trained on 100B tokens using the rough six-N-D rule.

LESSON 12 / PART 02

How Feature Detectors Emerge

A weighted pattern detector

A 5×5 image can be flattened into 25 brightness inputs. A neuron with positive weights down the center column and negative weights elsewhere responds strongly to a vertical line. The weight pattern acts as a template, even though the neuron merely calculates a dot product plus bias. This illustrates how learned weights can encode sensitivity to meaningful structures.

Learning the pattern through gradients

For a sigmoid classifier with binary cross-entropy, the derivative of the loss with respect to the preactivation is prediction minus target. A feature's weight gradient multiplies that error by the feature value. Across many examples, input directions that reduce prediction error are strengthened or suppressed. Hidden features emerge indirectly because they help downstream objectives, not because each neuron is explicitly told what concept to represent.

$$dL/dweight_i = (predicted_probability - target) \times input_i$$

Why interpretability is hard

In a large model, a feature may be distributed over many dimensions or involve several interacting layers. A single neuron can participate in multiple unrelated features. Mechanistic interpretability investigates circuits through activation measurements, interventions, and causal tests. Sparse autoencoders may decompose activations into a larger dictionary of sparsely active features, but their outputs are not guaranteed to reveal a unique true ontology of concepts.

WORKED EXAMPLE

A cross shape may activate the vertical-line detector because it contains the central vertical stroke. Additional neurons and nonlinear combinations are needed to distinguish “contains a vertical line” from “is exclusively a vertical line.”

CORE IDEA

A feature is a pattern in the input or activation space to which a learned computation responds; human-readable concepts are an imperfect interpretation of those patterns.

TRY IT

Construct a 3×3 weight template sensitive to a horizontal line and explain how it reacts to a plus sign.

LESSON 13 / PART 02

Shortcut Learning and Causal Understanding

Predictive shortcuts

A network trained to identify cows may rely on green backgrounds if nearly every cow appears in a meadow. If “cow anatomy” and “grass” always occur together in training, either feature can perfectly predict the label. The optimizer has no reason to prefer the intended feature unless the data, architecture, or objectives distinguish them.

Correlation is not intervention

Statistical association $P(Y|X)$ is not generally equivalent to the result of intervening and setting X . A causal relationship can be confounded by other variables. A system trained on observations alone may predict familiar patterns while failing to estimate what changes when an action is taken. Interventions, varied environments, and domain assumptions may be necessary to identify causal structure.

observational $P(Y | X=x)$ need not equal interventional $P(Y | \text{do}(X=x))$

Robustness from meaningful variation

Data augmentation can change background, lighting, or orientation while preserving the label, encouraging invariance to irrelevant properties. But invariance must be chosen carefully: reversing word order can change a sentence's meaning, and rotating certain digits can change their labels. Out-of-distribution tests are useful precisely because they separate intended features from easy shortcuts.

WORKED EXAMPLE

If both a wolf and a dog can appear in snow or grass, background is no longer a reliable shortcut. The model must rely more on animal-specific evidence.

CORE IDEA

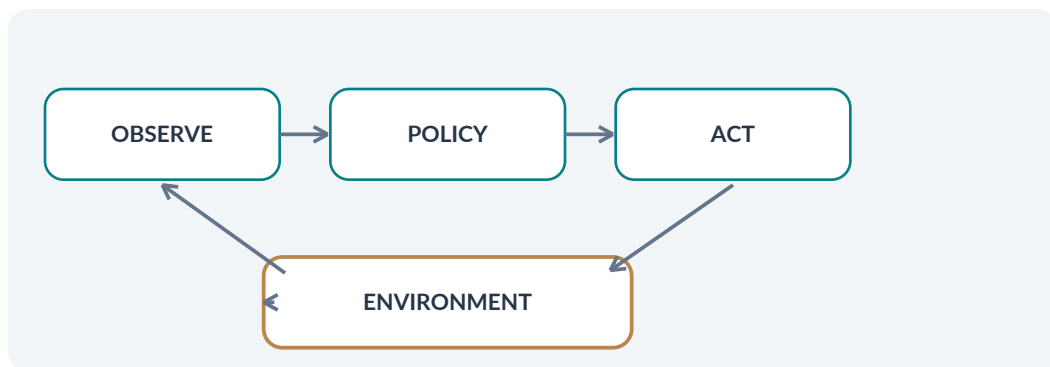
A model can optimize its training objective perfectly while learning the wrong explanatory structure. Better information and testing, not only larger models, are often needed.

TRY IT

Invent a software benchmark with an unintended shortcut. Design a counterfactual test that breaks the shortcut while preserving the correct answer.

LESSON 14 / PART 02

World Models and Predicting Action Consequences



From classification to dynamics

A classifier predicts a label; a world model predicts what an environment will do following an action. In a 5×5 grid, the true state might be a robot's coordinates, and the transition function changes those coordinates after a move while respecting boundaries. In a physical scene, the state may also need velocities, contacts, and hidden variables. The relevant state should contain enough information to predict the next state given an action.

$$\text{next_state} \sim P(\text{next_state} \mid \text{current_state}, \text{action})$$

Learning compact latent dynamics

A robot observes images, not a neat list of state variables. A neural encoder can map observations into a compact latent representation, and a transition model can predict how that representation changes with action. Predicting task-relevant latent states may be cheaper than rendering every pixel. Yet discarding important information—such as a small obstacle—can make planning unsafe or ineffective.

Planning and model error

A planner can simulate action sequences, score their predicted outcomes, execute a first action, observe reality, and replan. This is model-predictive control. Small state-estimation or dynamics errors may compound over long rollouts. When exact transition rules exist, as in chess or Sokoban, using an exact simulator is often preferable to training an approximate neural substitute.

WORKED EXAMPLE

MuZero-style algorithms learn representations, dynamics-related quantities, policies, and values that are useful for search without reconstructing every visual detail of the environment.

CORE IDEA

World models make action-conditioned prediction possible. Their value comes from supporting planning, not merely from producing visually plausible futures.

TRY IT

In a sliding-on-ice grid world, explain why position alone is not sufficient state and list the extra variables a useful model may need.

LESSON 15 / PART 02

Latent Reasoning and Recurrent Thought

Neural computation already occurs without words

A Transformer processes continuous hidden vectors through many layers before emitting an output token. Extended reasoning, however, may require more sequential computation than a fixed forward pass provides. One approach is to generate intermediate text. Another is to repeatedly refine continuous hidden states without verbalizing each step.

$$\text{hidden_state}(k+1) = F_{\theta}(\text{hidden_state}(k), \text{problem})$$

Learning useful internal iterations

Recurrent latent reasoning requires the update operator to make progress. Backpropagation through time can train repeated steps using a final loss, but long sequences may suffer unstable gradients and expensive memory demands. Intermediate supervision can help on tasks where useful partial states are known. Outcome-based training is more flexible but poses difficult credit assignment.

Knowing when to stop

A recurrent model may use a learned halting probability, change in hidden state, a computational budget, or an external verifier. Convergence does not imply correctness: a system may stabilize on a wrong solution. The strongest stopping signal in exactly verifiable tasks is satisfying the actual constraints. Latent reasoning is an active research area, and more iterations do not uniformly improve results.

WORKED EXAMPLE

In a grid shortest-path calculation, distance estimates propagate outward from the goal over successive iterations without a single sentence being generated. A neural system could try to learn such an iterative update.

CORE IDEA

Latent reasoning replaces some intermediate token generation with repeated hidden-state computation, but training stable, useful, and verifiable recurrence remains challenging.

TRY IT

Write the Bellman-style update for distance-to-goal estimates in an unweighted grid and explain why multiple passes are necessary.

LESSON 16 / PART 02

Self-Play and Recursive Self-Learning

Learning from one's own experience

Self-play allows an agent to generate data without requiring demonstrations from expert humans. A game-playing policy produces trajectories, a simulator calculates outcomes, and training increases the probability of successful strategies. A stronger policy can generate more challenging future experience, creating a recursive improvement loop.

$$\text{model}(k+1) = \text{Train}(\text{model}(k), \text{verified_experience}(k))$$

Self-training is not the same as self-correction

During inference, an agent might revise an answer or search more deeply without changing its parameters. During recursive self-training, the learned weights themselves change between generations. These are different timescales and mechanisms.

In-context adaptation can be useful without permanent training, while parameter updates can affect future behavior beyond the current task.

The role of independent feedback

If a model generates incorrect arithmetic answers and blindly fine-tunes on them, it can strengthen its mistakes. Self-generated data is valuable when paired with reliable reward, simulation, an exact checker, or independent evaluation. AlphaZero-style self-play combines search and outcome learning; verifier-guided reasoning data can support similar bootstrapping in mathematics and code.

WORKED EXAMPLE

For a puzzle with an exact solution checker, a model can generate several candidate solutions, retain verified successes, and train from those examples. Improvement must still be assessed on unseen puzzles.

CORE IDEA

Recursive self-learning is a loop of experience generation, trustworthy evaluation, and training. The loop is not guaranteed to improve on every iteration.

TRY IT

Explain why a generator-only training loop is vulnerable to reinforcing hallucinated answers.

LESSON 17 / PART 02

Verifiers, Curricula, and Reliable Improvement

Generator and verifier play different roles

A generator proposes possible solutions. A verifier checks them. The asymmetry is powerful because some problems are easier to verify than to solve. Sudoku solutions, arithmetic results, and legal game states can be checked exactly. Passing unit tests is informative for code but not a proof of complete correctness, because tests may omit important cases.

Three learning strategies

Verified examples can support filtered supervised fine-tuning, where the model imitates successful outputs. Reinforcement learning can use graded reward signals over sampled trajectories. Search-distillation uses a stronger computational process to generate targets for a cheaper model. Each has advantages and failure modes; selecting the method depends on how good the evaluator is and how expensive exploration is.

Curriculum and regression protection

An adaptive curriculum samples tasks near the model's current ability rather than spending all effort on trivial or impossible problems. A self-improvement pipeline should preserve independent held-out benchmarks and earlier skills. New candidate models should be accepted only when relevant metrics improve and protected capabilities remain above defined thresholds. Repeated training on unchecked generated output can narrow the distribution and reinforce errors.

WORKED EXAMPLE

A self-learning Sokoban agent can use an exact simulator and solution verifier, generate increasingly difficult layouts, train its policy on successful searches, and assess transfer to reserved puzzles.

CORE IDEA

Reliable self-improvement needs independent verification, a useful learning curriculum, and a promotion gate that detects regressions.

TRY IT

Define an evaluation gate for a self-trained coding agent that checks both improved bug fixing and unchanged performance on earlier tasks.

LESSON 18 / PART 02

Continual Learning and Catastrophic Forgetting

How learning one task damages another

Weights may support multiple behaviors. An update that reduces loss on a new task can increase loss on an earlier task if their gradient directions conflict. This is catastrophic forgetting. It becomes important when an agent learns sequentially from new experiences instead of repeatedly training on all previous data.

$$\text{change_in_old_loss} \approx \text{gradient_old} \cdot \text{parameter_update}$$

Replay, regularization, and modularity

Experience replay mixes old examples into new training. Elastic Weight Consolidation penalizes changes to parameters estimated to be important for earlier tasks. Modular architectures isolate some updates in specialists or adapters. None completely eliminates interference when tasks truly compete for the same representational capacity. External memory offers another route: store new facts immediately without changing model weights.

Multiple learning timescales

An agent can update its working state during inference, save episodes between tasks, and periodically train its neural components. These timescales need not coincide. New library API details may belong in semantic memory; a frequently repeated debugging strategy may eventually justify procedural fine-tuning. The system should decide what information deserves durable storage and what should become a reusable skill.

WORKED EXAMPLE

A chess-and-Go system with fully shared parameters may lose chess skill after intensive Go training. Separate adapters and a shared frozen backbone can help preserve previous specialization.

CORE IDEA

Learning indefinitely requires preserving useful old structure while acquiring new structure. External memory and modular updates are practical complements to weight training.

TRY IT

For old loss $(w-2)^2$ and new loss $(w+2)^2$, evaluate both losses at $w = 2, 0$, and -2 .

LESSON 19 / PART 02

Meta-Learning: Learning to Learn

From task training to adaptation training

Ordinary supervised learning optimizes performance on known examples.

Meta-learning optimizes how quickly a model adapts to new tasks drawn from a task family. Training episodes contain both adaptation examples and held-out evaluation examples. The objective rewards useful performance after a small adaptation budget, not merely accuracy before adaptation.

$$\text{meta_objective} = \mathbb{E} \text{ over tasks } [\text{loss_after_adaptation}(\text{task})]$$

Gradient-based and metric-based approaches

Model-Agnostic Meta-Learning (MAML) learns a parameter initialization that can adapt quickly through a few gradient steps. Metric-based approaches learn embedding spaces where a new class can be recognized from a few prototypes. In-context learning is another adaptation mechanism: a pretrained model changes behavior in response to prompt examples without updating parameters. These mechanisms should not be conflated.

Learning to allocate learning resources

A meta-learning orchestrator could learn which specialist to fine-tune, which examples are informative, and when to use retrieval rather than training. But an adaptation strategy learned across one distribution of tasks may not transfer to unrelated domains. The broader meta-learning problem is to discover reusable learning biases without overspecializing to the meta-training suite.

WORKED EXAMPLE

In two tasks with preferred scalar weights 2 and 8, a shared starting weight of 5 minimizes average post-adaptation squared loss for symmetric one-step updates with suitable learning rates.

CORE IDEA

Meta-learning aims to improve the process of adaptation itself. It is central to systems that must confront unfamiliar tasks efficiently.

TRY IT

Contrast changing model weights with adding three demonstrations to the prompt. Which changes persist after the current inference ends?

PART 03 — TOOLS, SPECIALISTS, AND COGNITIVE CONTROL**PART 03**

Tools, Specialists, and Cognitive Control

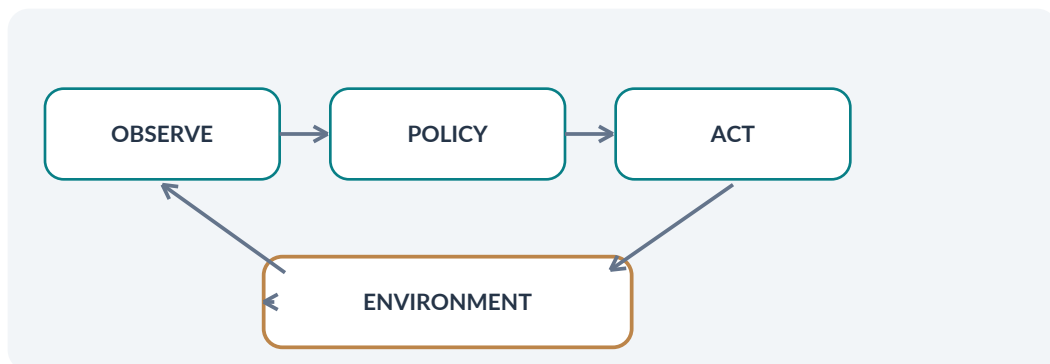
Lessons 20–26 examine tool use, modular models, executive orchestration, brain-inspired design, and hierarchical cooperation.

Questions to carry into this part

Which computations should be delegated, which information shared, and who decides?

LESSON 20 / PART 03

Tools and Acting in the Outside World



From a predictor to an agent

A neural predictor maps an input to an output. An agent maintains state, chooses actions, observes consequences, and revises its behavior over time. Tool use connects a learned policy to externally implemented operations such as calculators, compilers, databases, web search, and robotic controls. The model proposes a structured invocation; an execution runtime validates permissions, runs the operation, and returns a result.

observe → decide → act → observe_again

Tool selection and sequential evidence

Choosing a tool is a decision problem. A coding agent should often run a failing test before editing the source; the error message changes what it should inspect next. Tools may fail, return stale information, or produce incomplete evidence. The agent needs an execution history, explicit distinction between observation and hypothesis, and recovery strategies such as retry, switch, inspect, or stop.

Permissions and real-world effects

Reading a file and deleting a file are not equivalent actions. An agentic runtime should enforce authorization, validation, and limits independently of what a neural model proposes. A language model predicting a valid tool-call format does not guarantee the requested action is safe, permitted, or relevant. External enforcement is part of reliable agency.

WORKED EXAMPLE

A calculator call returns 9,716 for 347×28 . The result becomes another observation that the model can use, not a new model-weight update.

CORE IDEA

Tool use makes neural intelligence interactive, but robust action requires runtime controls, feedback, and correct interpretation of environmental results.

TRY IT

Design the observation-action sequence for debugging a failed Python test, including one likely failure-recovery branch.

LESSON 21 / PART 03

Specialized Networks and Shared Interfaces

Unified, modular, and hybrid architectures

A unified multimodal model learns many tasks using shared parameters. A modular system uses independent specialists such as vision, speech, reasoning, world models, and control. A hybrid shares some representations while retaining specialized computation. Specialized models can be updated or deployed independently, but communication and orchestration introduce new complexity.

Communication through text, structure, or latents

A vision model describing a chessboard could emit prose, an exact board state, or a continuous embedding. Prose is flexible, structured output is easy to verify, and latent vectors can preserve richer learned information when downstream models can interpret them. Independently trained vector spaces are not automatically aligned merely because they have the same dimension.

```
downstream_representation =  
learned_adapter(upstream_representation)
```

Mixture-of-experts is not the same thing

An internal MoE activates selected expert blocks inside one network, often on a per-token basis. A modular system chooses among independent full models or tools, often on a per-subtask basis. Both exploit specialization but operate at different granularities. A larger system can include an MoE language model alongside separate vision or audio networks.

WORKED EXAMPLE

An AI analyzing a video might run visual event detection and speech transcription in parallel, then synchronize the results into a temporal event graph.

CORE IDEA

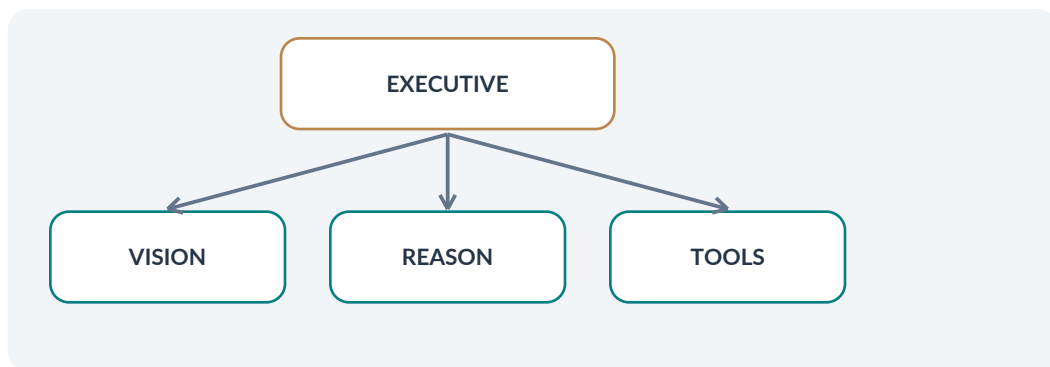
Specialization is valuable when tasks require different computational operations; common interfaces should preserve structure rather than forcing every modality into prose.

TRY IT

For a robot picking up a cup, identify what should be represented as language, structured state, and continuous control values.

LESSON 22 / PART 03

Executive Orchestration and Metareasoning



A router is not an executive

A router chooses which specialist receives a request. An orchestrator manages a process: goals, subgoals, dependencies, working state, resource budgets, uncertainties, execution, and verification. It must decide whether to calculate, search, retrieve, simulate, invoke specialists, gather evidence, or stop. The decision is a policy over computational actions.

```
next_computation ~ policy(goal, working_state, uncertainty, budget)
```

Value of computation and information

An expensive model may be justified when errors are costly; a cheap model may suffice for familiar low-stakes tasks. Sometimes the best next action is a diagnostic rather than an attempted solution. The expected value of information compares the improved decisions possible after observing new evidence against the observation's cost. This is central to intelligent resource allocation.

Training an orchestrator

Supervised demonstrations can teach common routing decisions. Contextual-bandit methods can learn cost-sensitive specialist selection. Reinforcement learning can optimize longer sequences of tool and model calls when rewards are trustworthy. An orchestrator should be compared with a rule-based baseline because extra learned complexity does not guarantee better results.

WORKED EXAMPLE

For software debugging, a diagnostic costing 5 units may be better than taking an immediate action with expected error cost of 40 units.

CORE IDEA

An executive controller's competence lies in choosing useful computational processes and revising them based on evidence.

TRY IT

Define a utility function balancing success probability, latency, tool cost, and failure severity for an agent choosing a specialist.

LESSON 23 / PART 03

Brain-Inspired Modular Intelligence

What neuroscience suggests—and does not prove

Biological brains contain interacting systems specialized in perception, memory, movement, executive control, and learning. This suggests organizational principles such as local autonomy, recurrent communication, and coordinated attention. It does not imply that copying brain-region names into an AI diagram will reproduce biological intelligence or consciousness.

Shared workspace and attention allocation

A shared workspace can publish high-priority observations and active goals to several specialists. Its communication protocol might carry text, structured events, tensors, and references to memory. Selecting what reaches the workspace is a systems-level attention problem, distinct from the dot-product attention computed inside a Transformer. Without gating and priority, a global workspace can become overloaded.

Multiple timescales

Low-level control may need millisecond-level feedback, perception may update more slowly, executive planning may run over seconds or minutes, and model training may happen only periodically. A large language model should not micromanage every robotic joint movement. Local specialist policies can execute fast loops while higher-level controllers manage objectives, exceptions, and scarce resources.

WORKED EXAMPLE

When a vision specialist detects a nearby person crossing a robot's path, the event should reach the motion controller promptly without waiting for a prose summary and another long model inference.

CORE IDEA

Brain-inspired modularity is most useful as an engineering source of principles—specialization, shared access, action selection, memory, and predictive correction—not as literal biological imitation.

TRY IT

Identify which parts of a household robot require fast local control, a shared workspace event, or deliberative planning.

LESSON 24 / PART 03

Hierarchical Planning and Multi-Agent Cooperation

Abstract skills reduce long-horizon complexity

Planning directly over primitive actions can create enormous branching factors. Hierarchical reinforcement learning uses temporally extended skills called options, defined by initiation conditions, an internal policy, and termination conditions. A high-level controller chooses an option while a lower-level policy executes its details. This resembles invoking reusable software functions.

```
option = (initiation_set, internal_policy, termination_rule)
```

Dependencies and concurrency

A long software-development task should be represented as a dependency graph, not just a flat list. Game rules must be defined before move validation is implemented; UI and core logic may proceed in parallel; integration tests wait for both. The orchestrator needs version checks, ownership, idempotent operations, and failure recovery. Ordinary distributed-systems mechanisms often solve these consistency problems more reliably than neural reasoning.

Teams of agents introduce new problems

Autonomous subagents may maintain different local goals and working memories. Their actions can conflict, duplicate work, or optimize local metrics at the expense of the global task. Multi-agent credit assignment asks which contributions truly improved the outcome. Centralized coordination, shared contracts, and independent verification help manage these problems. More agents are not automatically better when delegation costs exceed benefits.

WORKED EXAMPLE

A coding agent that edits a file while another agent simultaneously refactors it needs conflict detection and integration procedures, not merely better language understanding.

CORE IDEA

Long-horizon intelligence requires reusable skills, explicit dependencies, coherent objectives, and coordinated execution across many steps.

TRY IT

Draw a task graph for producing a narrated tutorial video, marking which stages can execute in parallel.

LESSON 25 / PART 03

Learning Which Computation to Perform

Routing as a contextual decision problem

An intelligent controller should know when to invoke a cheap model, a larger reasoner, a verifier, or an exact tool. A contextual bandit can learn from records of task features, selected strategies, outcomes, and costs. The learned success probabilities must be calibrated when used to make cost-sensitive choices. Exploring alternative strategies is necessary to avoid locking into a poor initial preference.

$$\text{expected_cost(strategy)} = \text{execution_cost} + \text{failure_probability} \times \text{failure_penalty}$$

Value of information and adaptive depth

Sometimes a diagnostic yields greater value than an immediate solution attempt. For internal reasoning, another iteration or search expansion is worthwhile only if its expected improvement exceeds its cost. A learned halting policy can allocate more computation to difficult tasks, but self-reported confidence is unreliable without external calibration or verification.

Policy-induced data bias

The orchestrator collects data mostly from actions it already selects. If it rarely tests a specialist, it may incorrectly conclude that the specialist is inferior. Offline benchmarks, controlled exploration, and careful evaluation help correct this data bias. A strong rule-based baseline is essential to determine whether learned metareasoning genuinely adds value.

WORKED EXAMPLE

A high-cost mathematical failure can justify using an exact solver even when a smaller neural model gives plausible answers almost all the time.

CORE IDEA

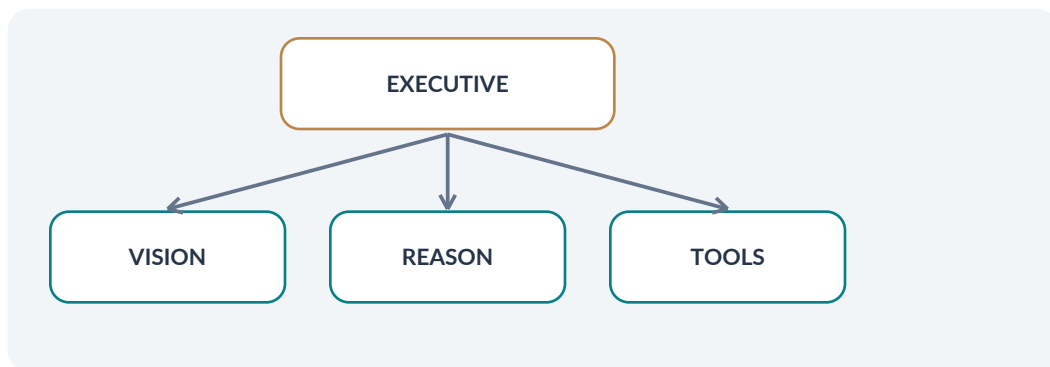
Adaptive intelligence includes deciding what to compute, how much to compute, and when information is worth acquiring.

TRY IT

A diagnostic costs 3 units and lowers the expected failure loss from 20 to 6. Calculate its net value.

LESSON 26 / PART 03

Integrating an Agentic Cognitive Architecture



Three interacting systems

A practical modular agent contains a cognitive layer that understands goals and chooses actions, an execution layer that enforces permissions and runs tools, and a learning layer that improves selected components from verified experience. Keeping these concerns separate makes failures easier to diagnose and updates safer to deploy.

An unfamiliar task from start to finish

Imagine an agent receiving the rules of a new board game and a current position. It parses the rules, constructs a typed state representation, identifies legal actions, selects between search and learned evaluation, verifies the proposed move, explains the decision, and stores useful experiences. A neural policy need not memorize every game's rules if a known exact simulator can supply reliable transitions.

What is established and what remains experimental

Tool execution, retrieval, specialist routing, structured task graphs, and fixed orchestration are practical engineering capabilities. Learned routing and domain-specific world-model planning are demonstrable but require evaluation. Broad continual neural learning, adaptive latent reasoning, and autonomous architecture-level self-improvement remain research challenges. Combining components doesn't prove general intelligence.

WORKED EXAMPLE

Compare one unified multimodal model, several specialists with fixed rules, and the same specialists with a learned router. Hold the workload and computation budget constant, then assess accuracy, cost, latency, and robustness.

CORE IDEA

The central hypothesis is not that modularity guarantees intelligence, but that coordinating the right computations can improve task performance and efficiency.

TRY IT

List the smallest five components needed for an experimental agent and define one measurable success criterion for each.

PART 04 — IMPLEMENTING AND EVALUATING MODULAR AI

PART 04

Implementing and Evaluating Modular AI

Lessons 27–34 turn the principles into interfaces, memory systems, learned routers, experiments, and a testable agent architecture.

**Questions to carry into this part**

How do we turn architectural hypotheses into credible experiments?

LESSON 27 / PART 04

Common Representations Across Modalities

Numbers do not imply shared meaning

Language, images, graphs, board states, and robotic observations all become tensors, but equal-dimensional vectors need not carry compatible meanings. A text embedding may preserve semantics while an image-control representation must preserve precise positions. A practical system should maintain shared semantic representations plus specialist-specific state and typed structured information.

Typed intermediate representations

A common task interface can describe entities, features, relations, legal actions, observations, and outcomes. For chess this might be board tensors and legal-move records; for a robot, object states and control constraints; for software, syntax trees and symbol graphs. This resembles the role of an intermediate representation in a compiler: distinct components can exchange well-defined structure without reducing everything to prose.

Learning adapters and graph reasoning

A learned projection can align representations from independently trained encoders, but it cannot recover information already discarded. Contrastive objectives align semantically related inputs; predictive and downstream objectives preserve task-relevant information. Graph neural networks exchange messages along entity relationships, supporting relational processing without converting every edge into natural language.

```
node_update =  $\psi$ (node_state, sum of neighbor_messages)
```

WORKED EXAMPLE

A vision model reading a chessboard can output a typed board state. A legal-move engine then uses exact rules, a policy network scores candidate moves, and a language model explains the selected action.

CORE IDEA

A common representation is an interface contract, not necessarily one universal latent vector. Structured data, embeddings, and adapters complement one another.

TRY IT

Define a minimal common schema for three game entities and two typed relationships. Which fields must remain exact?

LESSON 28 / PART 04

Neural Memory and Knowledge Systems

Fast memory and slow learning

A continually useful agent needs working memory for current tasks, episodic storage for past events, semantic memory for facts and relations, and procedural memory for reusable skills. Some of these can be neural states, others ordinary databases, and others trained weights. Updating an external record is often safer and faster than fine-tuning a model after every new fact.

Associative retrieval

A learned query can score stored keys and retrieve weighted values. Vector retrieval is useful for approximate semantic matches, while graph and relational queries capture exact dependencies. A trustworthy memory system should track timestamps, source provenance, versions, confidence, and access constraints. Similarity alone cannot distinguish a current valid API description from an obsolete one.

$$\text{retrieved_memory} = \sum_i \text{softmax}(\text{query} \cdot \text{key}_i) \times \text{value}_i$$

Consolidation, forgetting, and memory policy

Repeated successful episodes can suggest a generalized procedure that eventually becomes a trained skill. But premature consolidation may turn an accident into a false rule. Memory management must decide what to store, retrieve, summarize, discard, and protect. These decisions can be learned or engineered based on expected future usefulness and the cost of errors.

WORKED EXAMPLE

A coding agent stores an individual stack trace as an episode, a library-version change as a semantic fact, and a frequently successful debugging sequence as a procedural skill.

CORE IDEA

Effective memory is selective and verifiable. More stored information is not automatically more usable knowledge.

TRY IT

Propose rules for storing a new fact, a failed tool call, and a repeated successful workflow in different memory systems.

LESSON 29 / PART 04

Training a Small Neural Orchestrator

Represent the task and available actions

Begin with a small feature vector describing task difficulty, modality, required exactness, time budget, and failure cost. A lightweight MLP can output logits over computational actions such as direct answer, retrieve, reason, search, and verify. Training first on a supervised dataset of preferred strategies is simpler and more interpretable than immediate online reinforcement learning.

```
routing_policy = softmax(MLP(task_features, execution_state))
```

Move from routing to sequential decisions

A single routing choice cannot handle workflows where new observations change the optimal next computation. A recurrent or history-aware controller can read tool results, maintain execution state, and choose the next action. Outcome-based reinforcement learning is suitable when task completion is objectively measurable and exploration is controlled. Failure recovery and stopping should be trained and evaluated separately.

Experimental baselines

Compare a fixed rule-based router, a language-model orchestrator, and a learned neural controller using identical tools and task budgets. Record quality, latency, cost, intervention count, and regressions. If the learned controller adds complexity without measurable gains, simplify it. A model that appears flexible during demonstrations may not generalize to novel task combinations.

WORKED EXAMPLE

A model may learn that arithmetic favors a calculator, source debugging favors tests and inspection, and image understanding favors the vision encoder—but these are conditional choices, not hard-coded declarations of model intelligence.

CORE IDEA

Train orchestration as an empirical decision-making problem, beginning with strong deterministic baselines.

TRY IT

Specify six task features and five routing actions for a first MLP orchestrator. How would you label training examples?

LESSON 30 / PART 04

Integrating Recurrence, Search, and World Models

Three complementary operations

A recurrent neural core repeatedly refines a latent state. Search explores explicit candidate actions and their consequences. A world model predicts transitions when exact rules are unknown. For a deterministic game with known rules, the strongest design often uses the exact simulator with neural policy and value estimates. For unfamiliar physical control, learned latent dynamics may be useful for planning.

Training search-guided representations

A policy/value network can provide move priors and leaf evaluations for tree search. Search-improved action distributions and outcomes provide targets for further network training. Adding a recurrent reasoning core provides variable internal computation, but must be tested against simpler baselines. Extra latent iterations only help when the update function learns useful refinements.

```
latent_next = recurrent_core(latent_current, state); policy,value =  
heads(latent_final)
```

Adaptive planning depth

The meta-controller may choose direct action for trivial states, recurrence for uncertain evaluation, or deeper search for high-stakes choices. Learned dynamics can accumulate prediction errors; executing one action and replanning after real observations limits compounding mistakes. Independent verifiers offer strong stopping criteria when exact constraints exist.

WORKED EXAMPLE

A Sokoban policy can learn to avoid corner deadlocks, while exact search handles legal moves and proof of solution. A recurrent state may refine estimates of which boxes are trapped.

CORE IDEA

Recurrence refines internal estimates, search explores alternatives, and world models simulate action consequences. These are not interchangeable sources of computation.

TRY IT

Design an ablation study separating the gain from search, recurrent refinement, and better state encoding.

LESSON 31 / PART 04

Joint Training or Independent Specialists?

Full end-to-end gradients

If all components are differentiable, a downstream loss can train encoders, reasoning modules, and decoders together. This may produce closely coordinated representations but also increases coupling, training cost, and regression risk. Exact tools and external databases are not ordinary differentiable layers, so a fully end-to-end learning graph is often impossible or undesirable.

Frozen specialists and adapters

An alternative is to train specialists independently, freeze their weights, and fit learned projections or connector modules. This is modular and easier to deploy but only works if the upstream representation contains the information downstream tasks need. Fine-tuning selected adapters or layers is a compromise between independent and fully joint training.

Learned interfaces need versioning

A neural embedding interface is more than a shape declaration. Replacing a vision checkpoint may preserve its vector width while changing its semantic geometry, silently breaking downstream models. Component replacement requires compatibility benchmarks, not just type checks. Comparing frozen, adapter-tuned, partially tuned, and jointly trained systems can reveal whether the extra coupling is worthwhile.

WORKED EXAMPLE

An image encoder may discard exact object locations when optimized for semantic retrieval. No downstream projection can reliably recover coordinates absent from its outputs.

CORE IDEA

Start with strong independent specialists and train only the connections that clearly need improvement. Joint training is an option, not a prerequisite.

TRY IT

Design an experiment comparing an unchanged specialist, a learned adapter, and partial fine-tuning on the same downstream vision-to-control task.

LESSON 32 / PART 04

Recursive Improvement of the Entire Agent

System improvement is broader than weight updates

An agentic system can improve by updating neural weights, replacing tools, changing search algorithms, improving memory indexing, or training a better routing policy. All can raise end-to-end capability without requiring a larger foundational model.

Architecture-level improvement is more ambitious than self-training on generated answers because it changes the mechanisms through which future tasks are solved.

```
candidate_system = propose_update(current_system,  
                                  verified_experience)
```

Promotion gates and regression checks

The proposing model should not be the only judge of success. A candidate release must pass independent tests and compare against the previous version. Overall average improvement can hide critical regressions, so important capabilities need minimum performance thresholds. Versioning, isolated experiments, rollback, and provenance of changes are ordinary software practices that become essential for self-modifying systems.

Why the loop may plateau

New examples can be redundant, rewards can be exploited, evaluation suites can be overfit, and synthetic data may reinforce a narrowing task distribution. A curriculum generator may improve the learner by proposing solvable challenges near its current competence, but novelty must be tested independently. There is no established method guaranteeing unlimited recursive capability growth.

WORKED EXAMPLE

Replacing a slow exact-search implementation with a faster verified algorithm can improve system performance while leaving all neural weights untouched.

CORE IDEA

Recursive improvement should be a controlled scientific experiment: propose, evaluate, compare, accept or reject, and repeat.

TRY IT

Write three release gates for an AI system that improves coding performance but must preserve reliability on other tasks.

LESSON 33 / PART 04

Benchmarking Generalization and Agency

Performance is multidimensional

An agent can succeed on familiar tasks while failing on new combinations, longer horizons, or tool errors. A single benchmark score cannot distinguish memorization, learned algorithms, external search, or clever tool use. Useful evaluations should separately measure known-task accuracy, unseen-task transfer, adaptation speed, long-horizon reliability, calibration, and computational efficiency.

Strong splits and ablations

For a multi-game system, evaluate new positions in known games, new rules using familiar mechanics, unseen game families, and changing environments. Keep hidden evaluation tasks independent of the agent's generated curriculum. Remove components one at a time—memory, search, recurrence, router, specialists—to measure their contribution. Interacting components may require additional joint ablations.

Compounding failures

Even a system with 99% independent step success has only about 36.6% probability that all 100 steps succeed. Real outcomes are often correlated and agents can recover, but the calculation explains why whole-workflow evaluation matters. Tool reliability, checkpoints, verification, and error recovery may improve end-to-end success more than a small gain in isolated model accuracy.

$$P(100 \text{ independent successful steps}) = 0.99^{100} \approx 0.366$$

WORKED EXAMPLE

A game agent that learns five training games may show excellent scores while failing to adapt to a sixth unfamiliar game. The latter result is more informative about cross-game transfer.

CORE IDEA

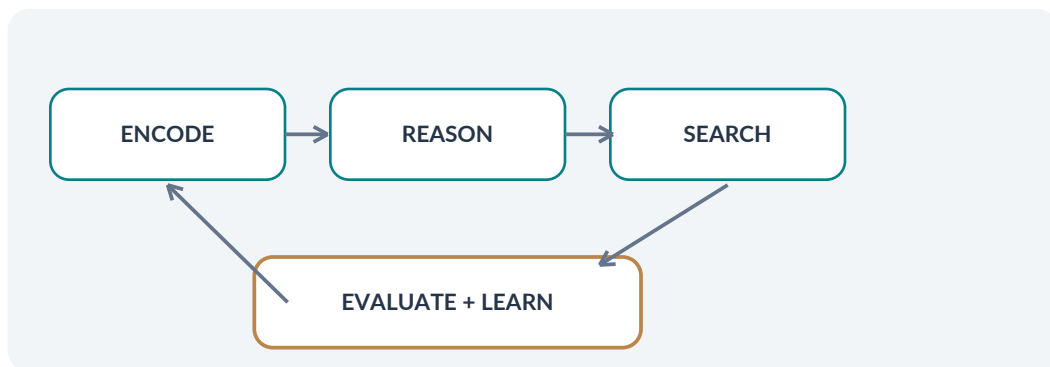
Improvement claims need independent generalization tests, controlled budgets, reliable task-level outcomes, and transparent failure analysis.

TRY IT

Design four held-out test sets that progressively increase the novelty of tasks for a card-and-board-game agent.

LESSON 34 / PART 04

Designing a Working Experimental Architecture



Begin with a tractable domain

Card games, board games, and puzzles make useful testbeds because rules can be represented explicitly, states are structured, and outcomes can often be verified. Start with several games that exercise different mechanics—such as FreeCell, Sokoban, Chess, and 2048—rather than immediately attempting unrestricted general intelligence. Use a typed task representation so every game supplies legal actions, observations, transitions, objectives, and termination rules.

Five components worth building first

Use a state encoder, policy/value network, optional recurrent reasoning core, small meta-controller, and offline learning coordinator. Keep exact simulators, action masks, search procedures, and evaluation gates as conventional software at first. A 256-dimensional embedding and tens of millions of trainable parameters are a reasonable experimental scale to investigate architectural properties, not an empirically established universal optimum.

```
state_encoder → reasoning_core → policy/value → search_or_act →  
verified_feedback
```

A disciplined training ladder

First train the state encoder and policy/value head against known solutions or search targets. Then add search-guided self-play. Compare fixed and adaptive search budgets. Introduce recurrent latent refinement only after establishing strong nonrecurrent baselines. Finally, train the meta-controller using records of which computational strategies succeed efficiently. Reserve unfamiliar games and rule combinations for genuinely held-out evaluation.

What successful research would demonstrate

The strongest evidence is not a beautiful architecture diagram; it is measurable transfer. Does the shared model learn new game mechanics with fewer examples? Does adaptive compute beat a fixed budget? Do stored experiences reduce repeated errors? Can verified training improve held-out outcomes without catastrophic forgetting? If yes, we have learned something scientifically meaningful about modular neural intelligence.

WORKED EXAMPLE

Train on selected grid puzzles and card games, then evaluate on rule variants never seen during training. Compare against individual specialist models and an exact-search baseline under matched compute budgets.

CORE IDEA

The complete architecture is a research hypothesis. Build incrementally, verify each component's contribution, and let measured generalization determine what deserves to scale.

TRY IT

Write a 12-week experiment plan with a baseline, two architecture changes, and a reserved evaluation set.

Appendix A • Mathematical toolkit

A short reference to the mathematical ideas used throughout the lessons.

Vectors and dot products

A vector is an ordered collection of numbers. Its dot product with another vector multiplies corresponding elements and adds the results. Dot products underlie linear layers, similarity calculations, and attention logits.

Matrix multiplication

A weight matrix transforms one vector into another. Stacking matrix products composes transformations; inserting nonlinear activations between them allows richer functions than one affine mapping.

Derivatives and gradients

A derivative measures how an output changes with a small input change. A gradient collects derivatives with respect to all adjustable parameters and provides local information for optimization.

Chain rule

When y depends on h and h depends on x , the derivative dy/dx is the product of the relevant local derivatives. Backpropagation applies this rule to an entire computational graph.

Probability and expectation

A policy outputs action probabilities. Expected return averages outcomes weighted by their probability, allowing decisions to consider uncertain futures.

Softmax

Exponentiate logits and divide by their sum to create nonnegative weights totaling one. The transformation is used for classification, attention, and action selection.

Information and compression

Predictive models assign probabilities to sequences. Better probability estimates correspond to lower expected coding length under appropriate entropy-coding assumptions; they can reflect learned regularities in data.

Appendix B • Glossary

Selected terms from the course, ordered alphabetically for quick reference.

Activation

The output of a neuron or layer after applying its transformation and optional nonlinearity.

Actor-critic

Reinforcement learning methods combining a policy with a value estimator.

Adapter

A small learned connector aligning the representations or behaviors of larger modules.

Attention

A content-dependent mechanism for combining value vectors according to query-key compatibility.

Backpropagation

Efficient reverse-mode computation of the gradient of a loss with respect to model parameters.

Calibration

Agreement between predicted confidence levels and observed frequencies of correctness.

Catastrophic forgetting

Loss of earlier capabilities after training on new tasks.

Contrastive learning

Training representations by favoring designated matching pairs over mismatched pairs.

Curriculum learning

Organizing training examples so task difficulty and informativeness evolve with the learner.

Embedding

A numerical vector representation of an object, token, state, or concept.

Generalization

Performance on examples or conditions not used directly for fitting the model.

Gradient descent

Parameter updates that move locally against a loss gradient.

Inductive bias

Preferences induced by architecture, initialization, objective, optimization, or data.

KV cache

Stored Transformer attention keys and values from preceding positions during generation.

Latent state

An internal learned representation used for computation rather than direct user presentation.

Meta-learning

Learning representations or adaptation procedures that support fast learning on new tasks.

Mixture-of-experts

Conditional computation selecting a subset of expert sub-networks for each input.

Option

A temporally extended action consisting of initiation conditions, policy, and stopping rule.

Orchestrator

A controller selecting and coordinating computations, specialists, tools, and actions.

Overfitting

Fitting training-specific details that do not generalize well to future examples.

Policy

A function or distribution defining how an agent chooses actions from states.

Recurrent network

A system that reuses a learned state-update transformation across steps.

Replay buffer

A collection of prior experiences reused for training or evaluation.

Reward hacking

Achieving a specified reward through behavior that does not meet the underlying intent.

Superposition

Representation of several features in overlapping directions of an activation space.

Verifier

A component checking whether outputs satisfy task constraints or correctness criteria.

World model

A learned or programmed predictor of environment transitions conditioned on actions.

Appendix C • A 12-week implementation plan

A suggested starting schedule for turning the architecture into a controlled research experiment.

Weeks 1–2 • Define environments

Implement the typed state and action interface for two or three simple games. Create exact move validators, reproducible test states, and protected evaluation sets.

Weeks 3–4 • Establish non-neural baselines

Implement simple heuristics and conventional search. Measure task quality, solve rate, timing, and worst-case failures before adding learned components.

Weeks 5–6 • Train a compact encoder and policy

Build a modest graph or tensor encoder with policy and value heads. Train from known solutions or search-generated targets. Log training and held-out learning curves.

Weeks 7–8 • Add search-guided learning

Use the neural network to prioritize exact search, then collect improved targets. Compare fixed-search and learned evaluations under matched resource budgets.

Weeks 9–10 • Add metareasoning

Train a small router to choose direct policy action, deeper search, or external solver. Compare against fixed threshold rules and an expensive language-model controller.

Week 11 • Evaluate recurrence and memory

Ablate latent recurrent refinement and retrieval of prior experiences. Measure whether either component improves held-out tasks relative to added compute.

Week 12 • Conduct the final comparison

Run independent evaluation on unseen states, game variants, and failure cases.

Document negative results, resource use, regressions, and research questions before scaling.