1. What is Overfitting?

Overfitting happens when a **decision tree learns too much** —
it memorizes **training examples** (even noise or random patterns)
instead of learning the **real pattern**.It makes the tree too large and complex,
so it works well on training data but fails on new data.

2. What is Generalization?

Generalization means the model can **correctly predict new data** —
not just remember the training examples.

Good model → simple rules that fit most data
Bad model → memorizes every detail of training data (overfit)

3. How to Fix Overfitting — Decision Tree Pruning

Pruning = cutting off unnecessary parts of the tree to remove branches that are not truly useful.How Pruning Works:

1. Start with the **full tree**.
2. Look at small subtrees (with leaf nodes).
3. If a test **adds no real improvement**, remove it.
4. Replace it with a **leaf node** showing majority result (Yes/No).
5. Repeat for all nodes.

Result → smaller, simpler, better tree

4. Detecting Irrelevant Attributes — Information Gain

If a test splits data but **does not change** the ratio of positive/negative examples,
then the attribute is **irrelevant** → it has **low information gain**. High gain → good split (use it)
Low gain → probably useless (prune it)

5. Significance Test & Null Hypothesis

We use a **statistical test** to check if an attribute's effect is **real or random**.

- **Null Hypothesis:** "There is no real pattern; attribute is irrelevant."
- We calculate a value (Δ) that shows **how much actual results differ from expected** results.
- We use the **Chi-Square (χ^2)** test:
  - If Δ is large → pattern is **real**, reject null hypothesis
  - If Δ is small → pattern is random, prune the node

Example:
If $\chi^2 \geq 7.82$ (for 4 values attribute), we can **reject null hypothesis** at **5% level**.

This is called **χ^2 pruning (Chi-Square pruning)**.

6. Early Stopping vs. Pruning

- **Early stopping:** Stop building the tree early when no "good" split is found.
- **Pruning:** Build the full tree first, then remove unnecessary parts.

Early stopping is **not always good**, because:
Some patterns (like **XOR function**) only appear **after multiple splits**.
So pruning (build first, cut later) is **safer**.

7. Why Pruned Trees are Better

✅ Handle noisy data better
✅ Smaller and easier to understand
✅ Make fewer mistakes on new data

We want to learn a **hypothesis** (or model) that works well on **future data**,
not just the data we already have.

To do this, we must define what "future data" means and what "best" means.

Stationarity Assumption

We assume that:

- Data comes from a **fixed probability distribution** that **does not change over time**.
  (This is called **stationarity**.)
- Each example ($E_1, E_2, \ldots$) is **independent** and **identically distributed (i.i.d.)**.

👉 That means:
Each data point (like a training example) is drawn randomly from the **same source**,
and **does not depend** on the previous one.

Without this assumption, we can't learn anything meaningful —
because if the future is totally different from the past, learning is useless.

Error Rate

The **error rate** of a hypothesis =
👉 the **fraction of times** it gives a **wrong answer**.

Example:
If a model predicts correctly 80 out of 100 times,
then error rate = 20%.

But —
Low error on training data ≠ always good model!
It may be **overfitting** (memorizing training data).

So we must test it on **new data**.

Holdout Method (Simple Testing)

We divide the dataset into two parts:

- **Training set** → to train the model
- **Test set** → to check accuracy on unseen data

This is called **Holdout Cross-Validation**.

⚖️ Problem:

- If test set is too big → we train on less data (model becomes weak)
- If test set is too small → accuracy estimate becomes unreliable

So, we need a better method → **k-fold cross-validation**.

k-Fold Cross-Validation

This is a smarter way to use all the data efficiently.

Steps:

1. Split data into **k equal parts (folds)**.
2. Use **k–1 folds** for training and **1 fold** for testing.
3. Repeat **k times**, each time using a different fold as the test set.
4. Take the **average** of all test results → that's your final accuracy.

Common choices: k = 5 or 10
Gives more reliable results
Takes more computation time (k times longer)

Leave-One-Out Cross-Validation (LOOCV)

Extreme case:

- k = number of examples (n)
- That means → train on all but **1** example, test on that **1**
- Repeat for every example, Very accurate
  Very slow if data is large

Peeking Problem

Peeking happens when you **accidentally use test data** to make model decisions.

Example:
You try different settings (like tree depth) and pick the one that gives best accuracy **on test set**. But since you used test data to choose, you've **leaked information** → test set is no longer "unseen." Result: False confidence (your test results look better than they should).

How to Avoid Peeking

- Keep your **test set locked away** until the very end.
- Use a **validation set** to tune the model (not the test set).
- **Training set** → build the model
- **Validation set** → choose best model
- **Test set** → final evaluation only once

Model Selection: Complexity vs. Goodness of Fit

Sometimes we have many possible models (for example, small or large decision trees, or low-degree vs high-degree polynomials).
We must choose the **right complexity**.

- **Too simple model** → underfitting (misses patterns)
- **Too complex model** → overfitting (memorizes noise)

We want a **balance** — best performance on **validation data**.

10. How to Do Model Selection

We use a **wrapper algorithm** (like Figure 18.8):

1. Try different model sizes (simple → complex).
2. For each size, do **cross-validation** to get:
   - Training error (errT)
   - Validation error (errV)
3. Plot both errors:
   - Training error → always goes down (fits better)
   - Validation error → goes down first, then up (overfitting starts)

✅ Choose the size with **lowest validation error**
(that's the bottom of the "U" shape curve).

Example (Decision Trees)

"Size" = number of nodes.

- We can build tree **breadth-first** (level by level), choosing best attributes first.
- Stop when we reach the chosen size.
- Evaluate it with cross-validation.

So far, we only counted **how often** a hypothesis is wrong (the **error rate**).
But not all mistakes are **equally bad** — some are much **worse** in real life.

So, we introduce a **Loss Function** `L(x, y, ŷ)` → that tells **how bad** each mistake is.

Example: Spam Email Classification

Case	Real Label (y)	Predicted (ŷ)	Result	Which is worse?
1	spam	non-spam	You see a spam message	😐 small problem
2	non-spam	spam	You miss an important email	😩 big problem!

So even if both are 1 mistake each (same **error rate**),
👉 the **second one causes more loss**.

That's why **loss functions** matter — they let us measure how *badly* we are wrong, not just *how often*.

Definition of Loss Function

L(x, y, ^y) = {Utility(result using true label y)} - {Utility(result using predicted label }^y)
অর্থাৎ — তুমি যদি সঠিকভাবে predict করতে তাহলে যতটা লাভ (utility) পেতে,
ভুল করলে তার তুলনায় কতটা ক্ষতি (loss) হলো — সেটা মাপা হয়।

Simplified Loss (Independent of x)

In practice, we usually use simplified (L(y, \hat{y})) — doesn't depend on input x.

Example for spam:

```
L(spam, nospam) = 1
L(nospam, spam) = 10
```

So it's **10× worse** to classify a real email as spam!

✅ Also: `L(y, y) = 0` (no loss if correct).

Types of Loss Functions

Loss Type	Formula	Meaning	Used For
L_1 Loss	(	y - ^y	)
L_2 Loss (Squared)	((y - ^y)^2)	Penalizes large errors more heavily	Regression (smooth optimization)
$L_0/_1$ Loss (0-1 Loss)	0 if correct, else 1	Just counts wrong predictions	Classification

Expected Generalization Loss

We want a hypothesis (h) that minimizes the **expected loss** on future (unknown) data.

[
\text{GenLoss}*L(h) = \sum*{(x,y)} L(y, h(x)) P(x, y)
]

👉 It's the **average loss over all possible examples**, weighted by how likely each example is.

✅ Best hypothesis:
[
h^* = \arg\min_h \text{GenLoss}_L(h)
]

Empirical Loss (Approximation from Data)

Since we don't know true probability (P(x,y)),
we estimate using our training data (E) (size N):

[
\text{EmpLoss}*{L,E}(h) = \frac{1}{N} \sum*{(x,y) \in E} L(y, h(x))
]

Then we pick the hypothesis that minimizes this:

[
\hat{h} = \arg\min_h \text{EmpLoss}_{L,E}(h)
]

Why Learned h ≠ True f

Even after all this, our learned hypothesis (\hat{h}) may not equal the true function (f).
Four main reasons

Cause	Meaning	Example
Unrealizability	f not in hypothesis space H	Linear model can't capture non-linear data
Variance	Model gives different results on different datasets	High variance = overfitting

Noise	True f itself is uncertain	Random labeling errors
Computational Complexity	Too hard to find best model	Deep networks with billions of parameters

Small-scale vs Large-scale Learning

Type	Data Size	Main Error Source
Small-scale	Few 1000 examples	Approximation & estimation errors
Large-scale	Millions of examples	Computational limits (can't fully optimize)

- **Training error** ↓ (always improves),
- **Validation error** ↓ **then** ↑ (because of overfitting).

👉 We want a model that fits the data well **without becoming too complex**.

So now, instead of checking every model size separately (using cross-validation),
we directly combine two things into one single formula.

Regularization Formula

[
\text{Cost}(h) = \text{EmpLoss}(h) + \lambda , \text{Complexity}(h)
]

and

[
\hat{h} = \arg\min_h \text{Cost}(h)
]

- **EmpLoss(h)** → How wrong the hypothesis is on the training data (error).
- **Complexity(h)** → How complicated the model is (e.g., number of parameters, nodes, coefficients).
- **λ (lambda)** → Balancing factor between the two.

এই λ মানটা বলে দেয়,
আমরা কতটা "simplicity" কে গুরুত্ব দেব "accuracy" এর তুলনায়।

- বড় λ → বেশি penalty দেবে complex model কে → simpler model choose করবে
- ছোট λ → accuracy কে বেশি গুরুত্ব দেবে → complex model choose করবে

Suppose you are fitting a **polynomial curve** to data.

Term	Meaning
EmpLoss(h)	Training error (how close curve is to data points)
Complexity(h)	Size of coefficients (large = wiggly curve)
λ	Controls how smooth or wiggly the curve becomes

Small λ → curve becomes overfitted (too wiggly)
Large λ → curve becomes underfitted (too smooth)

Cross-validation for λ

Even though we combined loss and complexity,
we still don't know which λ gives the best generalization.
So we do **cross-validation** — test different λ values and pick the one with the lowest validation error.

Why It's Called Regularization

Because it **forces the hypothesis to be more "regular" or smoother**,
instead of being wildly flexible (overfitted).
In other words — it adds **discipline** to the learning process.

Choosing the Regularization Function

It depends on the **hypothesis type**:

Model Type	Regularization Function	Meaning
Polynomial Regression	Sum of squares of coefficients (Σw_i^2)	Prevents big coefficients → smoother curve
Decision Tree	Number of nodes or depth	Prevents large, overfitted trees
Neural Network	L2 norm or L1 norm on weights	Keeps weights small (simpler model)

Feature Selection

Another way to simplify models is to **reduce input features** (dimensionality reduction).
Remove attributes that **don't affect output much**.

Example:

- In decision trees, **χ^2 pruning** removes weak attributes → simpler tree.

অপ্রয়োজনীয় feature বাদ দিলে model ছোট হয়, আর generalization উন্নত হয়।

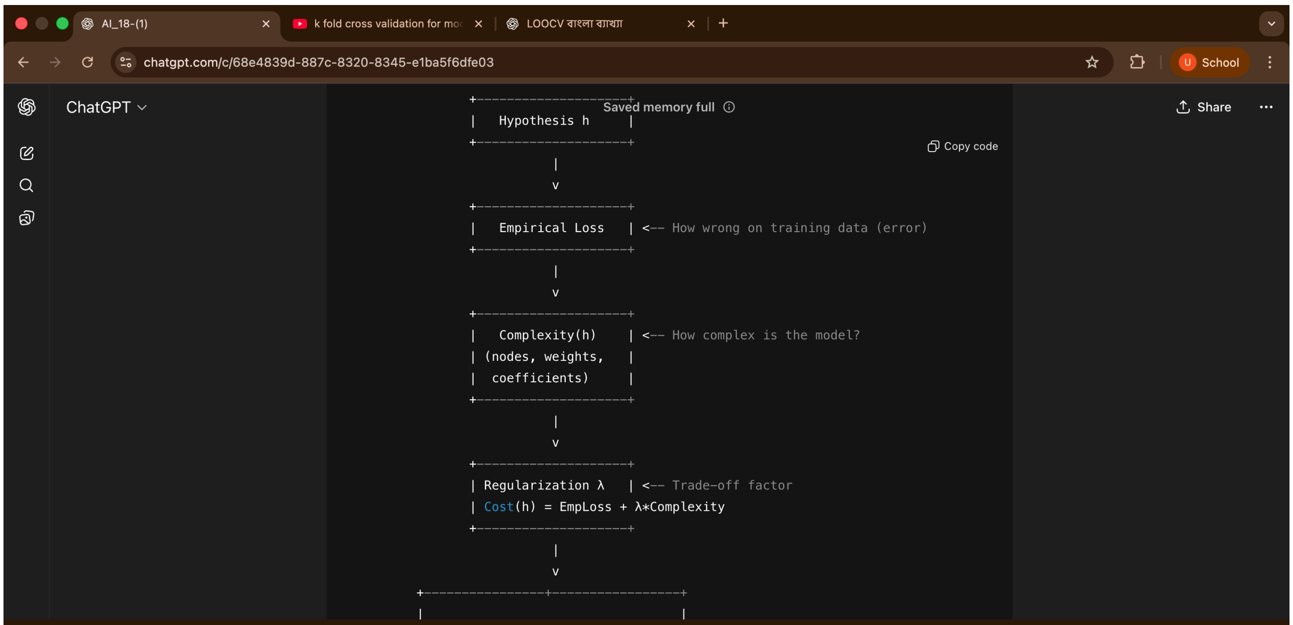
Minimum Description Length (MDL) Principle

Sometimes we can measure both *loss* and *complexity* in **the same unit — bits**.

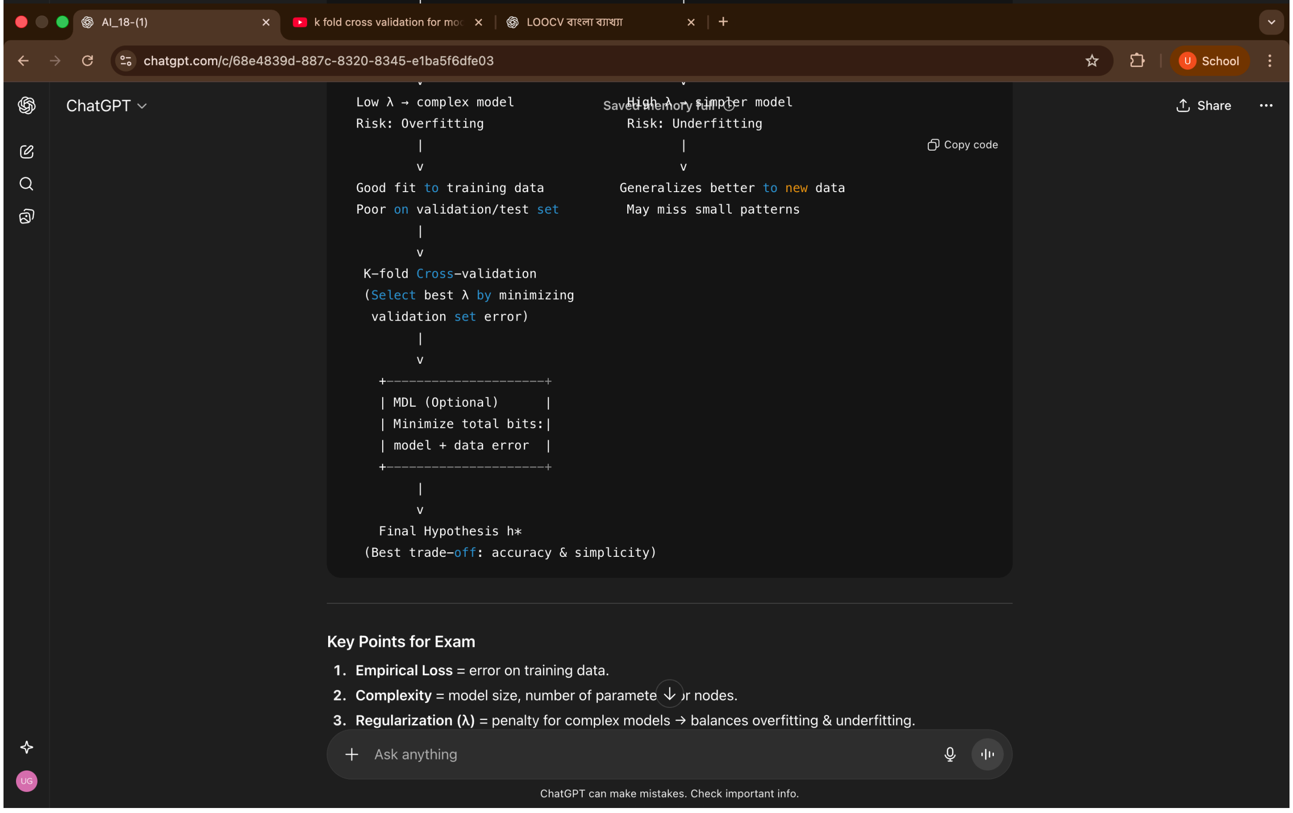
Choose the hypothesis that gives the **shortest total description** (in bits):
[
\text{Total Bits} = \text{Bits to encode hypothesis} + \text{Bits to encode data errors}
]

✅ Correct predictions cost 0 bits (no loss).
✅ Bigger errors = more bits needed to describe.

Then the **best hypothesis** = the one with **minimum total bits** →
that is **simple yet accurate**.



```
        +--------------------+
        |   Hypothesis h     |
        +--------------------+
                  |
                  v
        +--------------------+
        |   Empirical Loss   | <-- How wrong on training data (error)
        +--------------------+
                  |
                  v
        +--------------------+
        |   Complexity(h)    | <-- How complex is the model?
        | (nodes, weights,   |
        |  coefficients)     |
        +--------------------+
                  |
                  v
        +--------------------+
        | Regularization λ   | <-- Trade-off factor
        | Cost(h) = EmpLoss + λ*Complexity
        +--------------------+
                  |
                  v
 +----------------+-----------------+
 |                                  |
```



```
 +----------------+-----------------+
 |                                  |
 v                                  v
Low λ → complex model           High λ → simpler model
Risk: Overfitting               Risk: Underfitting
        |                                  |
        v                                  v
Good fit to training data      Generalizes better to new data
Poor on validation/test set     May miss small patterns
        |
        v
 K-fold Cross-validation
 (Select best λ by minimizing
  validation set error)
        |
        v
  +--------------------+
  | MDL (Optional)      |
  | Minimize total bits:|
  | model + data error  |
  +--------------------+
        |
        v
  Final Hypothesis h*
 (Best trade-off: accuracy & simplicity)
```

Key Points for Exam

1. **Empirical Loss** = error on training data.
2. **Complexity** = model size, number of paramete[illegible]r nodes.
3. **Regularization (λ)** = penalty for complex models → balances overfitting & underfitting.

MDL বলে — “সবচেয়ে ভালো hypothesis হলো যেটা দিয়ে ডেটা এবং মডেল দুটোই সবচেয়ে ছোট আকারে encode করা যায়।”
অর্থাৎ, ছোট model + কম ভুল = ভালো learning।

18.6 Regression with Linear Models (Univariate Case)

Linear Hypothesis

- A univariate linear model predicts (y) from (x) as:

[
h_w(x) = w_1 x + w_0
]

- (w_0, w_1) are **weights** (coefficients) to learn.
- Task: Find weights that **best fit training data**.

Batch vs Stochastic Gradient Descent

Method	Update Rule	Notes
Batch GD	Sum over all examples: (w_i \leftarrow w_i + \alpha \sum_j (y_j - h_w(x_j))\cdot x_j)	Guaranteed convergence for small α, slow if dataset is large
Stochastic GD	Update per single example: (w_i \leftarrow w_i + \alpha (y_j - h_w(x_j))\cdot x_j)	Faster, can work online, may oscillate around minimum

- **Learning rate α**: step size; too large → divergence, too small → slow convergence.

Visual Understanding

```
   Loss(w0,w1)
      ^
      |          Convex surface in weight space
      |        /\
      |       /  \
      |      /    \
      |     /      \
      |    /        \
      |   /          \
      +------------------> w1
          w0
- Convex → single global minimum
- Gradient descent moves “downhill” to minimum
```

18.6.2 Multivariate Linear Regression

Hypothesis

- For n-dimensional input (x_j = [x_{j,1}, x_{j,2}, ..., x_{j,n}]):
- Trick: define (x_{j,0} = 1) to include intercept (w_0) → now (h_w(x_j) = w \cdot x_j) (dot product).

Loss Function

- Use **squared error** (L2 loss):
- Goal: find (w) minimizing loss:

Gradient Descent Updates

- For each weight (w_i):
- Works like univariate case, but now considers all dimensions.

Analytical Solution

- Let **y** = vector of outputs, **X** = data matrix (rows = examples, columns = features including x0 = 1):

- Gives exact solution minimizing squared error.

Regularization (Prevent Overfitting)

- Multivariate regression in high dimensions → risk of overfitting.
- Regularized cost function:
- **q = 1 → L1 regularization:**
    - Minimizes sum of absolute values.
    - Produces **sparse model** (many weights = 0 → irrelevant features removed).
- **q = 2 → L2 regularization:**
    - Minimizes sum of squares.
    - Tends **not** to produce zero weights.

Why L1 vs L2

Aspect	L1 Regularization	L2 Regularization
Shape in weight space	Diamond → corners often hit axes → zero weights	Circle → no preference → weights rarely zero
Sparsity	Produces sparse models → feature	Dense model → all features usually
Dimensional axes	Treats axes as important → sensitive to feature meaning	Rotationally invariant → treats axes as arbitrary
Data required	Logarithmic in irrelevant features	Linear in irrelevant features

- **Intuition**: L1 → feature selection, simpler models. L2 → small weights, smooth models.

1. Hypothesis: ($h_w(x) = w \cdot x$) (vectorized).
2. Loss: L2 squared error.
3. Analytical solution: ($w = (X^T X)^{-1} X^T y$)
4. Regularization: L1 → sparse, L2 → smooth.
5. L1 → fewer examples needed for irrelevant features, L2 → rotationally invariant.

Linear Classifiers with Hard Threshold

- Linear functions can classify data into two classes.
- Input: ($x = [x_1, x_2, ..., x_n]$)
- Output: 0 or 1 (class label)
- Hypothesis with **dummy input ($x_0 = 1$)**:
- The **decision boundary** is the hyperplane ($w \cdot x = 0$).
- Data are **linearly separable** if a straight line (or hyperplane) can separate the classes.

Threshold Function

- Pass the linear function through a **hard threshold**:

Perceptron Learning Rule

- Gradient-based update cannot be used because threshold is non-differentiable almost everywhere.
- **Update rule (for one example ((x, y))):**

$$
w_i \leftarrow w_i + \alpha (y - h_w(x)) \cdot x_i
$$

- **Explanation:**

1. If prediction correct → no change.
2. If ($y = 1$) but ($h_w(x) = 0$) → increase weights corresponding to positive (x_i).
3. If ($y = 0$) but ($h_w(x) = 1$) → decrease weights corresponding to positive (x_i).

- Typically applied **stochastically**, one example at a time.

Convergence

- **Linearly separable data:** converges to zero-error solution.
- **Non-separable data:**
  - Standard perceptron may **never converge**.
  - Use **learning rate schedule**: ($\alpha(t) = O(1/t)$) → converges to **minimum-error solution**.
- Finding the **true minimum-error solution** is **NP-hard**.

Intuition

- Perceptron adjusts weights to push misclassified points across the decision boundary.
- Works perfectly if classes are linearly separable.
- In real-world noisy data, may oscillate → requires learning rate decay.

18.6.4 Linear Classification with Logistic Regression

- Hard threshold (perceptron) issues:

4. Non-differentiable → unpredictable learning.
5. Gives only 0 or 1 → no probability/gradated output.

- Solution: **soft threshold** using **logistic (sigmoid) function**:

$$
\text{Logistic}(z) = \frac{1}{1 + e^{-z}}
$$

- Smooth, differentiable, outputs in [0,1] → interpretable as **probability**.

- **Hypothesis**

- Replace hard threshold with logistic function:

$$
h_w(x) = \text{Logistic}(w \cdot x) = \frac{1}{1 + e^{-w \cdot x}}
$$

- Example: Probability that the class label = 1.
- Soft boundary: (h_w(x) = 0.5) at decision boundary; approaches 0 or 1 away from boundary.

Learning Weights (Logistic Regression)

- No closed-form solution → use **gradient descent**.
- Single example ((x, y)), squared error loss:

Advantages

- Handles **linearly separable** and **noisy/non-separable** data reliably.
- Outputs probabilities → better interpretability.
- Converges **predictably** using gradient-based methods.
- Widely used in real-world classification tasks.

This completes the **Linear Models → Classification section**, covering:

1. **Hard threshold** → perceptron
2. **Soft threshold** → logistic regression

18.7.1 Neural Network Structures

Basic idea:

- Inspired by the brain: neurons connected by weighted links.
- Each neuron/unit:
  - Inputs: (a_i) (activations of previous units)
  - Weights: (w_{i,j}) (strength of connection)
  - Activation: (a_j = g(\sum_i w_{i,j} a_i))
- Includes dummy bias input: (a_0 = 1) with weight (w_{0,j})

Activation functions:

- **Hard threshold → Perceptron**
- **Sigmoid/logistic → Sigmoid Perceptron**
  - Output between 0 and 1 → interpretable as probability
  - Differentiable → allows gradient-based learning

Network types:

1. **Feed-forward:** one direction, acyclic, no internal memory.
2. **Recurrent:** feedback loops, can model short-term memory, more complex dynamics.

Layer arrangement:

- Input layer → Hidden layer(s) → Output layer
- Single output or multiple outputs (e.g., multi-class classification)

18.7.2 Single-Layer Feed-Forward Networks (Perceptrons)

Structure:

- Inputs connect **directly** to outputs.
- Each output unit is **trained separately**:
    - **Perceptron learning rule** (hard threshold)
    - **Gradient descent** (sigmoid/logistic)

Limitations:

- Can only learn **linearly separable functions**:
    - Works: AND, OR, Majority function
    - Fails: XOR (non-linear, not separable)

Example: Two-bit adder

x1	x2	Carry (y3)	Sum (y4)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- Carry → learnable
- Sum → **cannot learn** with single-layer perceptron

Usefulness:

- Works well for linearly separable functions (e.g., majority function)
- Faster learning for some functions than decision trees

18.7.3 Multilayer Feed-Forward Neural Networks

Why multilayer?

- Single-layer cannot model non-linear functions (like XOR)
- Multilayer allows **non-linear decision boundaries**

Example network:

- 2 inputs → 2 hidden units → 2 outputs (fully connected)
- Output = nested functions of sigmoid activations:
  $$a_5 = g(w_{0,5} + w_{3,5} g(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2) + w_{4,5} g(w_{0,4} + w_{1,4}x_1 + w_{2,4}x_2))$$

Key idea:

- Each hidden unit = soft threshold in input space → combinations produce **ridges and bumps**
- One hidden layer → can approximate **any continuous function**
- Two hidden layers → can approximate **discontinuous functions**

Nonlinear regression:

- Network output = non-linear function of inputs
- Trainable with gradient descent

18.7.4 Learning in Multilayer Networks (Backpropagation)

Problem:

- Output layer error = easy ($Err_k = y_k - a_k$)
- Hidden layer error = unknown → **need to propagate backward**

Output layer Δ:

$$\Delta_k = Err_k \cdot g'(in_k)$$

$$w_{j,k} \leftarrow w_{j,k} + \alpha \cdot a_j \cdot \Delta_k$$

Hidden layer Δ:

$$\Delta_j = g'(in_j) \sum_k w_{j,k} \Delta_k$$

]
[
w_{i,j} \leftarrow w_{i,j} + \alpha \cdot a_i \cdot \Delta_j
]

Algorithm (Backpropagation):

1. Initialize weights randomly
2. Forward pass → compute activations
3. Compute output Δ values (observed error)
4. Backpropagate Δ values to hidden layers
5. Update weights using Δ
6. Repeat until convergence

Key points:

- Works for **any number of hidden layers**
- Gradient descent on differentiable activations
- Hidden layers learn **nonlinear transformations**
- Network structure often chosen via **cross-validation**

Example: Restaurant dataset

- 10 inputs → 1 hidden layer with 4 nodes → output
- Gradual error reduction, network converges
- Decision trees sometimes faster for simple, linearly separable data

Shortcuts

- **Perceptron** → linear separable only
- **Multilayer** → can model non-linear, complex functions
- **Backpropagation** = forward pass + backward Δ propagation + weight update
- **Sigmoid/logistic** → differentiable → allows gradient descent
- Output interpretation = probability for classification