Deep Learning Notes – Template

Prepared by Hasan Mia

Machine Learning and AI Notes

Example Topic: Gradient Descent and Adam Optimizer

What is Gradient Descent?

Answer:

Gradient Descent is an optimization algorithm used to minimize the cost function by iteratively updating parameters in the opposite direction of the gradient.

$$w_j := w_j - \alpha \frac{\partial J(w)}{\partial w_j} \tag{1}$$

Key Points:

- α = learning rate (controls step size)
- Large $\alpha \rightarrow$ overshoot, oscillation
- Small $\alpha \rightarrow$ slow convergence

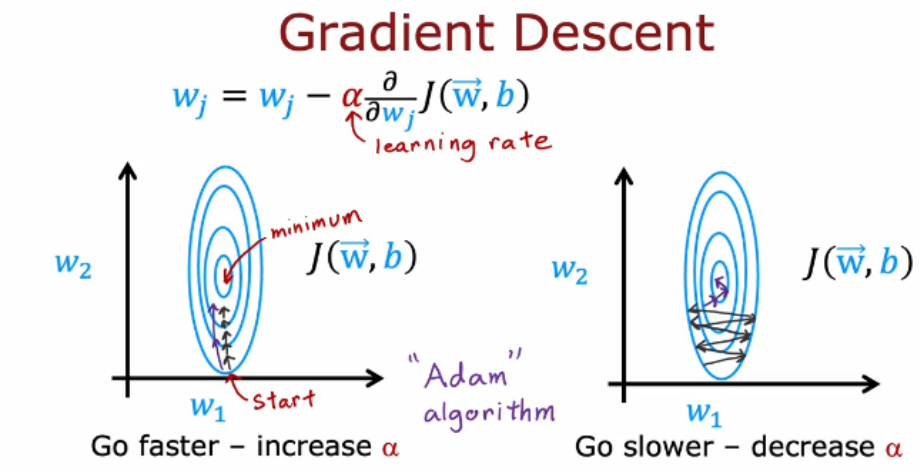

Figure 1: **Gradient Descent Visualization(left) and Adam algorithim(right)** The contour plot shows how the cost function decreases as the algorithm iteratively updates parameters toward the minimum.

Why is Adam Optimizer Faster than Gradient Descent?

Answer:

Adam (*Adaptive Moment Estimation*) adjusts the learning rate automatically for each parameter.

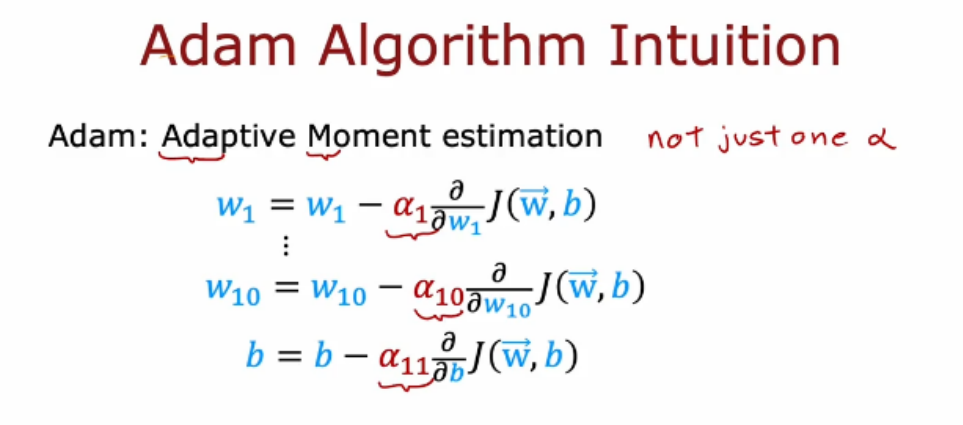

Figure 2: **Not just single learning rate(alphs) it's update continuously** .

- Increases α if a parameter moves consistently in one direction.
- Decreases α if a parameter oscillates.
- Combines advantages of RMSProp and Momentum.

Equation (Simplified):

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Typical Default: $\alpha = 10^{-3}$

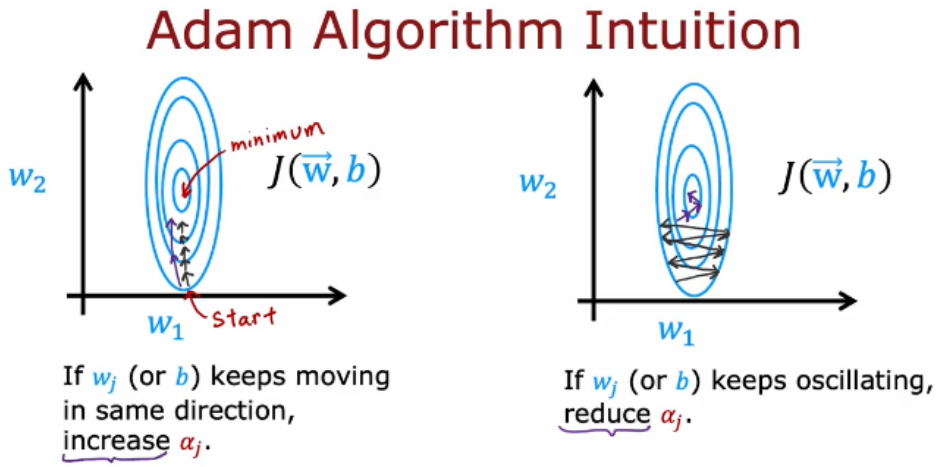

Figure 3: **Increase learning rate(alpha) same direction and reduce when it's oscillating** .

Implementation Example (TensorFlow):

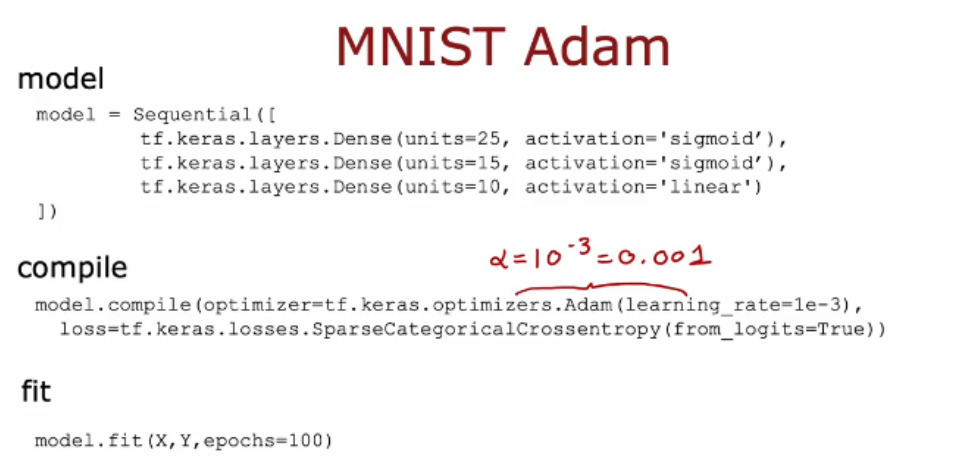

Figure 4: **MNIST using Adam optimization** .

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
loss='categorical_crossentropy', metrics=['accuracy'])
```

Advantages:

- Adapts learning rate dynamically.
- Works well for sparse gradients.
- Fast convergence in deep networks.

Key Takeaways:

- Gradient Descent = uniform learning rate.
- Adam = adaptive learning rate per parameter.
- Use Adam as a safe, fast default optimizer.

Convolutional Neural Networks (CNNs)

What are Convolutional Layers?

Answer:

So far, we've seen **dense layers** — where every neuron in a layer connects to all activations from the previous layer. However, some neural networks use other types of layers with unique properties, such as the **convolutional layer**.

In a convolutional layer, each neuron only looks at a small, localized region of the input

rather than the entire input. This local connectivity allows the model to focus on spatial or sequential patterns efficiently.

Example – EKG (Electrocardiogram) Signal Classification:
A convolutional neural network (CNN) can be used to analyze time-series data such as an EKG signal to detect the presence of heart disease.

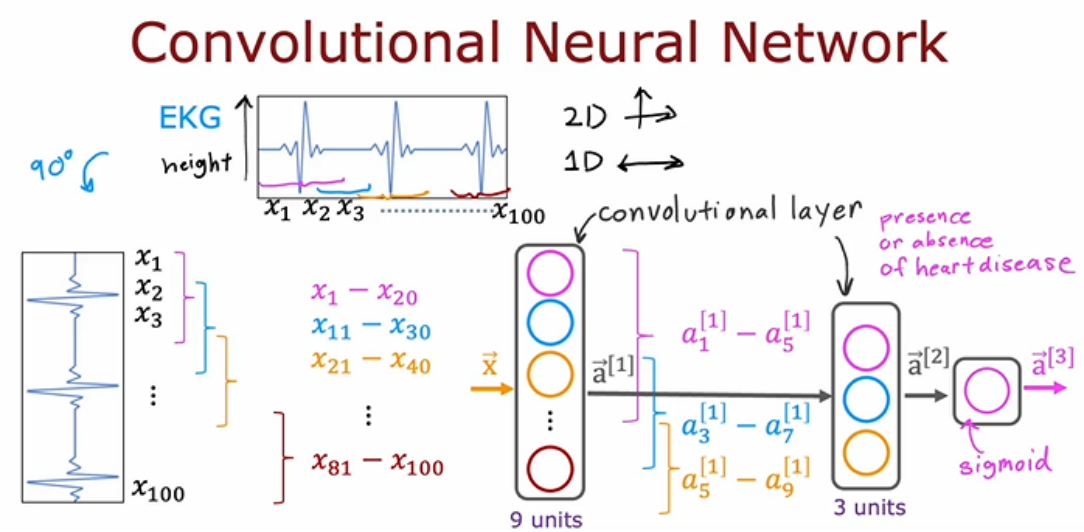

Figure 5: **Convolutional Neural Network for EKG Classification.** The 1D input (EKG signal) is processed by convolutional layers, each neuron analyzing a small window of inputs (e.g., x_1–x_{20}, x_{11}–x_{30}, etc.). Higher layers combine local patterns into global features, enabling the final sigmoid output layer to predict heart disease presence or absence.

How does a Convolutional Layer Work?

- Each neuron receives input only from a small region (called its *receptive field*).
- The same set of parameters (filter) slides over different regions of the input — a process called **convolution**.
- This parameter sharing reduces the total number of weights.

Advantages:

1. **Faster computation:** Fewer parameters compared to dense layers.
2. **Less overfitting:** Fewer parameters means simpler models and better generalization.
3. **Data efficiency:** Requires less training data for similar performance.

Example – Step-by-Step Process:

1. Input EKG signal $X = [x_1, x_2, ..., x_{100}]$.
2. First convolutional layer:
   - Neuron 1: looks at x_1–x_{20}
   - Neuron 2: looks at x_{11}–x_{30}
   - $\vdots$
   - Neuron 9: looks at x_{81}–x_{100}
3. Second convolutional layer:
   - Neuron 1: uses activations $a_1^{[1]}$–$a_5^{[1]}$
   - Neuron 2: uses $a_3^{[1]}$–$a_7^{[1]}$
   - Neuron 3: uses $a_5^{[1]}$–$a_9^{[1]}$
4. Final layer (Sigmoid): combines all features to output presence/absence of heart disease.

Key Takeaways:

- Convolutional layers detect local spatial or temporal features.
- CNNs are powerful for image and signal data.
- Parameter sharing and local connectivity improve training efficiency.
- CNNs can reduce overfitting and computation cost.

In modern AI, many architectures (like Transformers, LSTMs, and Attention models) build on the same idea — designing new types of layers to learn better from complex data.

Understanding Derivatives and Backpropagation Intuition

In TensorFlow or any deep learning framework, we specify a neural network architecture to compute the output y as a function of the input x, define a cost function J, and let the framework automatically use **backpropagation** to compute derivatives of J with respect to the parameters (weights and biases). These derivatives are then used in **Gradient Descent** or **Adam Optimizer** to update parameters and minimize the cost.

What is Backpropagation?

Backpropagation is the algorithm that computes how much each parameter in the neural network contributes to the total error. It applies the **chain rule of calculus** to compute gradients efficiently for all layers.

However, before diving into neural networks, let's understand derivatives intuitively using a simple example.

Example 1: A Simple Cost Function

Let the cost function be:

$$J(w) = w^2$$

Here, J is a function of a single parameter w. Suppose $w = 3$, then:

$$J(3) = 3^2 = 9$$

Now, let's increase w by a very small amount $\varepsilon = 0.001$. Then:

$$w = 3.001, \quad J(3.001) = (3.001)^2 = 9.006001$$

The change in J is:

$$\Delta J = J(3.001) - J(3) = 0.006001$$

Since w increased by $\varepsilon = 0.001$, we can compute the ratio:

$$\frac{\Delta J}{\Delta w} \approx \frac{0.006001}{0.001} = 6.001 \approx 6$$

So, the derivative (rate of change) is approximately 6.

Interpretation: If w increases by a tiny amount ε, $J(w)$ increases roughly by $6 \times \varepsilon$. Thus, the **derivative of J with respect to w** is:

$$\frac{dJ}{dw} = 6$$

Example 2: Testing for Other Values of w

Let's compute derivatives for different w values.

- When $w = 2$:

$$J(2) = 4, \quad J(2.001) = 4.004001 \Rightarrow \frac{dJ}{dw} \approx 4$$

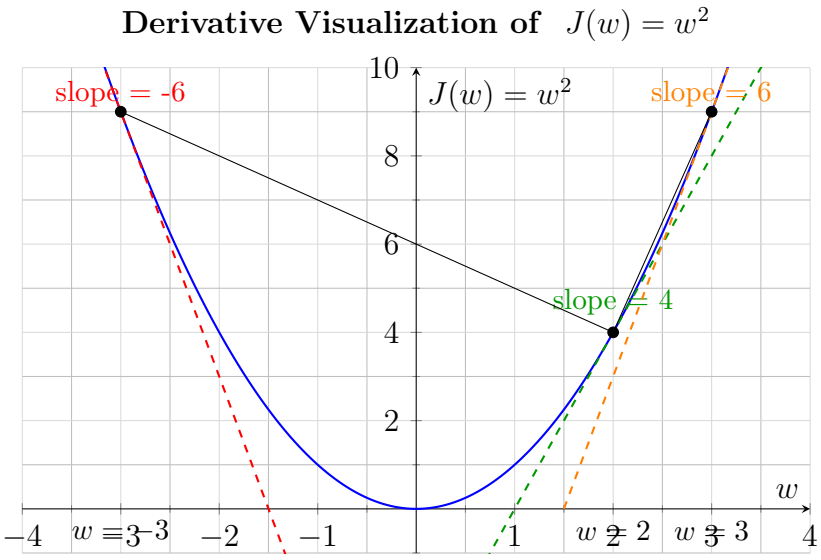

Figure 6: **Tangent Lines and Slopes of $J(w) = w^2$.** The slope (derivative) is negative at $w = -3$, small at $w = 2$, and steep positive at $w = 3$.

- When $w = -3$:

$$J(-3) = 9, \quad J(-2.999) = 8.994001 \Rightarrow \frac{dJ}{dw} \approx -6$$

Observation: The derivative changes depending on w. When w is positive, $\frac{dJ}{dw}$ is positive; when w is negative, $\frac{dJ}{dw}$ is negative.

In calculus, for $J(w) = w^2$,

$$\frac{dJ}{dw} = 2w$$

Hence:

$$w = 3 \Rightarrow \frac{dJ}{dw} = 6, \quad w = 2 \Rightarrow 4, \quad w = -3 \Rightarrow -6$$

Graphical Understanding: Slope of a Function

The derivative of a function at a point is the **slope of the tangent line** that just touches the function $J(w)$ at that point.

At $w = 3$, slope $= 6$; At $w = 2$, slope $= 4$; At $w = -3$, slope $= -6$.

Gradient Descent Update Rule

Once we know the derivative, Gradient Descent updates w as:

$$w := w - \alpha \frac{dJ}{dw}$$

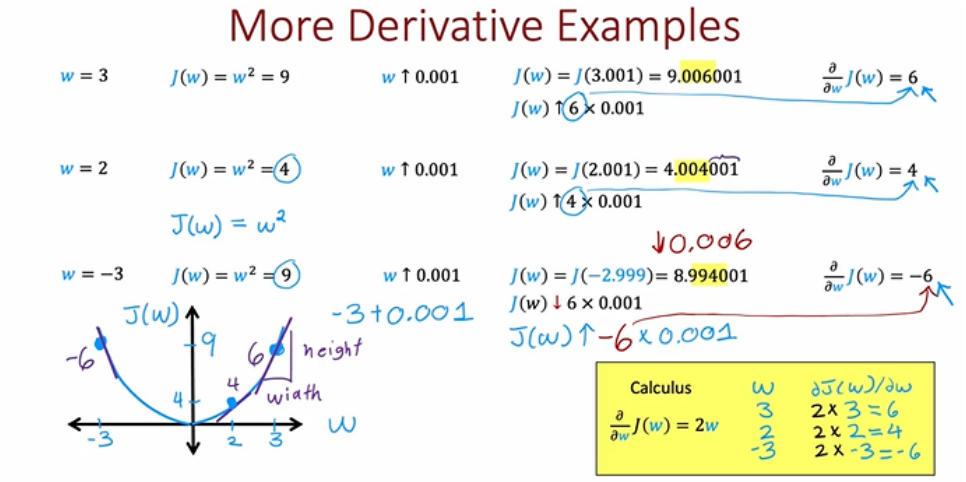

Figure 7: **Derivative as Slope.** The slope is positive when $J(w)$ increases with w, negative when it decreases.

where α is the learning rate.

Explanation:

- If the derivative is large, the cost function changes rapidly, so we make a larger update.
- If the derivative is small, $J(w)$ is almost flat — we take a small step.

Example 3: Using SymPy to Compute Derivatives in Python

Using the symbolic math library `SymPy`, we can calculate derivatives automatically.

```
from sympy import symbols, diff
w = symbols('w')
J = w**2
dJdw = diff(J, w)
print(dJdw) # Output: 2*w
```

You can substitute $w = 2$:

$$\frac{dJ}{dw} = 2 \times 2 = 4$$

Example 4: More Derivative Examples

Cost Function $J(w)$	**Derivative** $\frac{dJ}{dw}$	**At** $w = 2$
w^2	$2w$	4
w^3	$3w^2$	12
w	1	1
$\frac{1}{w}$	$-\frac{1}{w^2}$	-0.25

Intuition:

- The derivative shows how sensitive $J(w)$ is to changes in w.
- A positive derivative means J increases with w.
- A negative derivative means J decreases as w increases.

Derivative Notation in Calculus

You may see two types of notations:

$$\frac{dJ}{dw} \quad \text{(for single variable)} \quad \text{and} \quad \frac{\partial J}{\partial w_i} \quad \text{(for multiple variables)}$$

For neural networks, since J usually depends on many parameters $(w_1, w_2, \ldots, w_n)$, we use the **partial derivative notation** $\frac{\partial J}{\partial w_i}$.

Key Takeaways

- Derivative measures how much $J(w)$ changes when w changes slightly.
- $\frac{dJ}{dw} = 2w$ for $J(w) = w^2$.
- Backpropagation uses derivatives to update weights efficiently.
- Gradient Descent moves parameters in the direction that reduces J.
- For functions with multiple parameters, use partial derivatives.

Next, we'll visualize how backpropagation uses derivatives inside a neural network using a ***computation graph****.*

Computation Graphs — Forward Propagation & Backpropagation (Step-by-step)

The **computation graph** is the backbone of how modern deep learning frameworks (TensorFlow, PyTorch, etc.) compute outputs and *automatically* obtain derivatives via backpropagation. Below we use a tiny neural network (one input, one linear unit, one training example) to show the full forward pass and the backward pass (backprop) in a clear, numeric, step-by-step way.

Problem setup (Small neural network)

We have a single example and a single linear output unit:

$$a = wx + b \quad \text{(linear activation)}, \qquad J(w, b) = \tfrac{1}{2}(a - y)^2.$$

For this worked example we take:

$$x = -2, \qquad y = 2, \qquad w = 2, \qquad b = 8.$$

Forward propagation — compute J from left to right

Break the computation into atomic nodes and compute values step by step.

1. Compute the product $c = w \cdot x$.

$$c = wx = 2 \times (-2) = -4.$$

2. Compute the pre-activation / linear output $a = c + b = wx + b$.

$$a = c + b = -4 + 8 = 4.$$

3. Compute the difference $d = a - y$.

$$d = a - y = 4 - 2 = 2.$$

4. Compute the cost $J = \frac{1}{2}d^2$.

$$J = \tfrac{1}{2}d^2 = \tfrac{1}{2} \times 2^2 = 2.$$

At the end of this left-to-right pass we have evaluated the loss J given current parameters

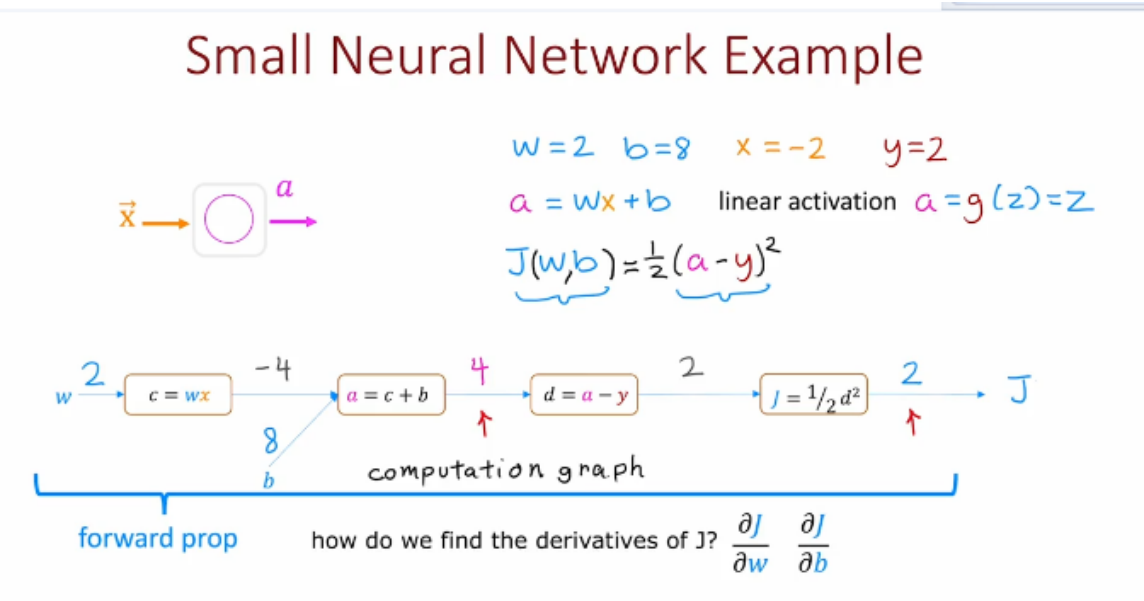

Figure 8: **Forward propagation computation graph.** Each node is a small operation; numbers above arrows are the computed values.

w, b. Next: **How would a small change in a parameter (say w) change the cost J?** That is answered by the backward pass.

Backward propagation — compute derivatives (right to left)

Backpropagation computes $\dfrac{\partial J}{\partial(\cdot)}$ for every intermediate quantity and parameter by moving from outputs back to inputs. We will compute:

$$\frac{\partial J}{\partial d}, \quad \frac{\partial J}{\partial a}, \quad \frac{\partial J}{\partial c}, \quad \frac{\partial J}{\partial b}, \quad \frac{\partial J}{\partial w}.$$

Step A — derivative at the final node (cost wrt d):

$$J = \tfrac{1}{2}d^2 \quad \Longrightarrow \quad \frac{\partial J}{\partial d} = d.$$

Numerically $d = 2 \Rightarrow \dfrac{\partial J}{\partial d} = 2.$

Step B — propagate to a: Because $d = a - y$, we have $\dfrac{\partial d}{\partial a} = 1$. Using the chain rule:

$$\frac{\partial J}{\partial a} = \frac{\partial J}{\partial d} \cdot \frac{\partial d}{\partial a} = (2) \cdot (1) = 2.$$

Step C — propagate to c and b: Since $a = c + b$,

$$\frac{\partial a}{\partial c} = 1, \qquad \frac{\partial a}{\partial b} = 1.$$

Therefore

$$\frac{\partial J}{\partial c} = \frac{\partial J}{\partial a} \cdot \frac{\partial a}{\partial c} = 2 \cdot 1 = 2,$$

and similarly

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial a} \cdot \frac{\partial a}{\partial b} = 2 \cdot 1 = 2.$$

Step D — propagate to w: Recall $c = wx$. So $\dfrac{\partial c}{\partial w} = x$. Apply the chain rule:

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial c} \cdot \frac{\partial c}{\partial w} = 2 \cdot x = 2 \times (-2) = -4.$$

Summary of computed derivatives (numeric):

$$\boxed{\frac{\partial J}{\partial d} = 2, \quad \frac{\partial J}{\partial a} = 2, \quad \frac{\partial J}{\partial c} = 2, \quad \frac{\partial J}{\partial b} = 2, \quad \frac{\partial J}{\partial w} = -4}$$

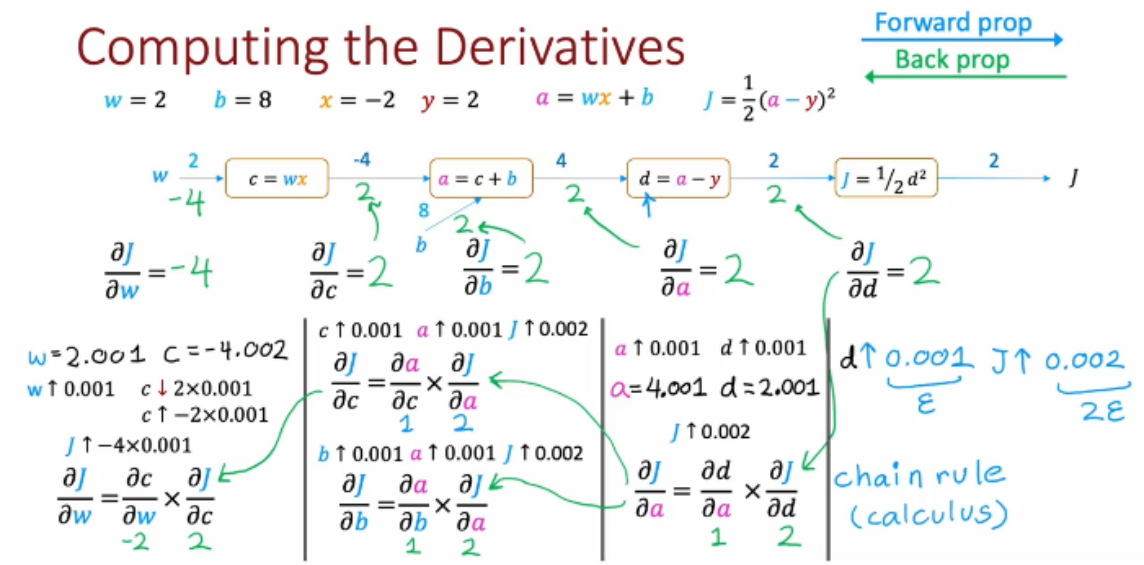

Figure 9: **Backward propagation on the computation graph.** Values written beside arrows show computed derivatives; arrows illustrate the right-to-left flow of gradients.

Small-ε intuition (finite-difference viewpoint)

Another way to understand the derivatives is to imagine increasing a variable by a small ε and observing the change in the cost:

- If d increases by ε, then $J = \frac{1}{2}d^2$ increases by approximately $d \cdot \varepsilon$. At $d = 2$, this is 2ε. So $\partial J/\partial d = 2$.
- If w increases by ε, then $c = wx$ changes by $x\varepsilon$. With $x = -2$, c changes by -2ε. Since $\partial J/\partial c = 2$, the net change in J is $2 \times (-2\varepsilon) = -4\varepsilon$. So $\partial J/\partial w = -4$. This matches the chain-rule computation above.

This finite-difference check is a helpful sanity test for hand-derived gradients.

Putting derivatives into the parameter update (Gradient Descent)

Using a simple gradient descent update with learning rate α:

$$w \leftarrow w - \alpha \frac{\partial J}{\partial w}, \qquad b \leftarrow b - \alpha \frac{\partial J}{\partial b}.$$

With $\dfrac{\partial J}{\partial w} = -4$ and $\dfrac{\partial J}{\partial b} = 2$, the updates move parameters in the direction that reduces J. For example, if $\alpha = 0.01$:

$$w \leftarrow 2 - 0.01 \times (-4) = 2 + 0.04 = 2.04, \qquad b \leftarrow 8 - 0.01 \times 2 = 7.98.$$

Note the sign: a *negative* derivative for w means increasing w will reduce J, so gradient descent increases w in this step.

Key lessons and why computation graphs help

1. **Modularity:** The computation graph decomposes a complex function into small nodes (operations). Each node computes a local forward value and a local derivative contribution.
2. **Local chain-rule application:** Backprop works by applying the chain rule locally at each node and passing gradients backward. This is efficient because each node reuses already computed partials.
3. **Automatic differentiation:** Frameworks build the computation graph during the forward pass and then perform a backward pass to accumulate gradients for every parameter — so you don't have to compute derivatives by hand.
4. **Numerical sanity check:** Small finite-difference changes (increase by ε) provide an intuitive check for the analytic derivatives.

Next: With this concrete example in mind, we can generalize to multi-layer networks: every layer contributes its local gradient and the chain rule stitches them together, which is exactly what backpropagation does for deep neural networks.

Larger Network: Forward & Backward Propagation (Worked Example)

In this section we apply the same computation-graph/backprop ideas to a slightly larger network with a single hidden layer (one hidden unit). We compute the forward pass numerically, then run a clean, step-by-step backward pass to get gradients for every parameter. This is exactly what automatic differentiation libraries do for you.

Network, data and notation

We use ReLU activations $g(z) = \max(0, z)$ and the squared-error cost:

$$a^{[1]} = g(z^{[1]}), \quad z^{[1]} = w^{[1]}x + b^{[1]}$$

$$a^{[2]} = g(z^{[2]}), \quad z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$$

$$J(w, b) = \tfrac{1}{2}\big(a^{[2]} - y\big)^2$$

Single training example and parameters (given):

$$x = 1, \quad y = 5, \qquad w^{[1]} = 2,\ b^{[1]} = 0, \qquad w^{[2]} = 3,\ b^{[2]} = 1.$$

Forward propagation — compute intermediate values (left → right)

Compute each node value and annotate the computation graph.

1. $t^{[1]} = w^{[1]}x = 2 \times 1 = 2.$
2. $z^{[1]} = t^{[1]} + b^{[1]} = 2 + 0 = 2.$
3. $a^{[1]} = g(z^{[1]}) = \max(0, 2) = 2.$ (ReLU is in its linear region here.)
4. $t^{[2]} = w^{[2]}a^{[1]} = 3 \times 2 = 6.$
5. $z^{[2]} = t^{[2]} + b^{[2]} = 6 + 1 = 7.$
6. $a^{[2]} = g(z^{[2]}) = \max(0, 7) = 7.$
7. Cost: $J = \frac{1}{2}(a^{[2]} - y)^2 = \frac{1}{2}(7 - 5)^2 = \frac{1}{2} \cdot 4 = 2.$

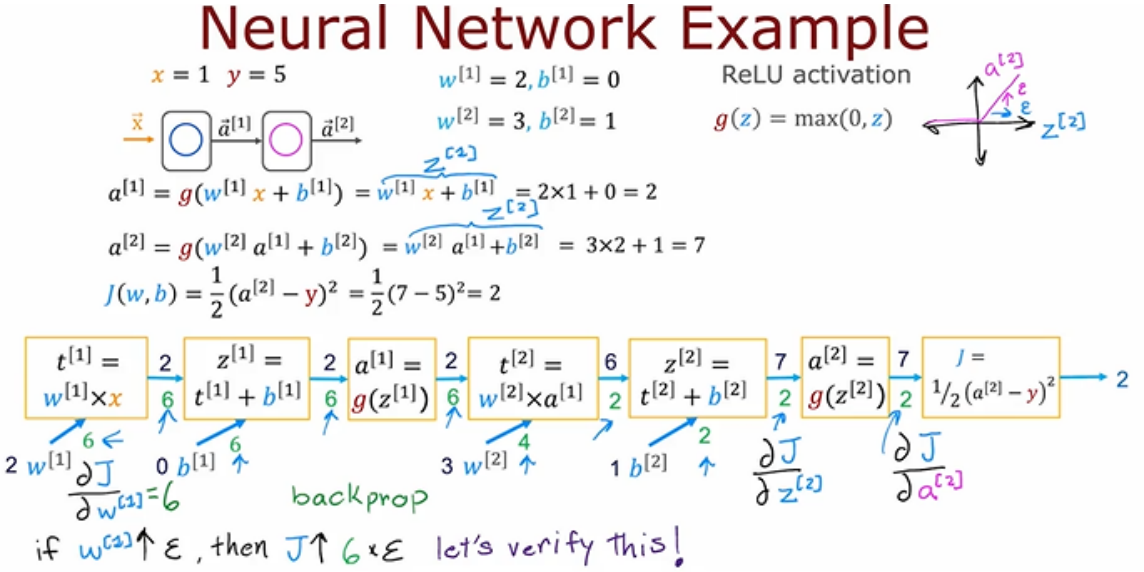

Figure 10: **Forward pass computation graph.** Values on arrows are the numeric forward values we computed above.

Backward propagation — compute gradients (right → left)

We compute the partial derivatives $\frac{\partial J}{\partial(\cdot)}$ at each node using the chain rule. For ReLU, recall:

$$g'(z) = \begin{cases} 1 & \text{if } z > 0, \\ 0 & \text{if } z \leq 0. \end{cases}$$

Here $z^{[1]} = 2 > 0$, $z^{[2]} = 7 > 0$, so both ReLU derivatives are 1.

1. Start at the cost: derivative of J w.r.t. output activation $a^{[2]}$

$$\frac{\partial J}{\partial a^{[2]}} = a^{[2]} - y = 7 - 5 = 2.$$

2. Backprop through ReLU at output:

$$\frac{\partial J}{\partial z^{[2]}} = \frac{\partial J}{\partial a^{[2]}} \cdot g'(z^{[2]}) = 2 \cdot 1 = 2.$$

3. Gradients for parameters of layer 2:

$$\frac{\partial J}{\partial b^{[2]}} = \frac{\partial J}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial b^{[2]}} = 2 \cdot 1 = 2.$$

$$\frac{\partial J}{\partial w^{[2]}} = \frac{\partial J}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial w^{[2]}} = 2 \cdot a^{[1]} = 2 \cdot 2 = 4.$$

4. Backprop into the hidden activation $a^{[1]}$:

$$\frac{\partial J}{\partial a^{[1]}} = \frac{\partial J}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial a^{[1]}} = 2 \cdot w^{[2]} = 2 \cdot 3 = 6.$$

5. Backprop through ReLU at hidden unit:

$$\frac{\partial J}{\partial z^{[1]}} = \frac{\partial J}{\partial a^{[1]}} \cdot g'(z^{[1]}) = 6 \cdot 1 = 6.$$

6. Gradients for parameters of layer 1:

$$\frac{\partial J}{\partial b^{[1]}} = \frac{\partial J}{\partial z^{[1]}} \cdot \frac{\partial z^{[1]}}{\partial b^{[1]}} = 6 \cdot 1 = 6.$$

$$\frac{\partial J}{\partial w^{[1]}} = \frac{\partial J}{\partial z^{[1]}} \cdot \frac{\partial z^{[1]}}{\partial w^{[1]}} = 6 \cdot x = 6 \cdot 1 = 6.$$

Summary of numeric gradients:

$$\boxed{\frac{\partial J}{\partial w^{[1]}} = 6, \quad \frac{\partial J}{\partial b^{[1]}} = 6, \quad \frac{\partial J}{\partial w^{[2]}} = 4, \quad \frac{\partial J}{\partial b^{[2]}} = 2}$$

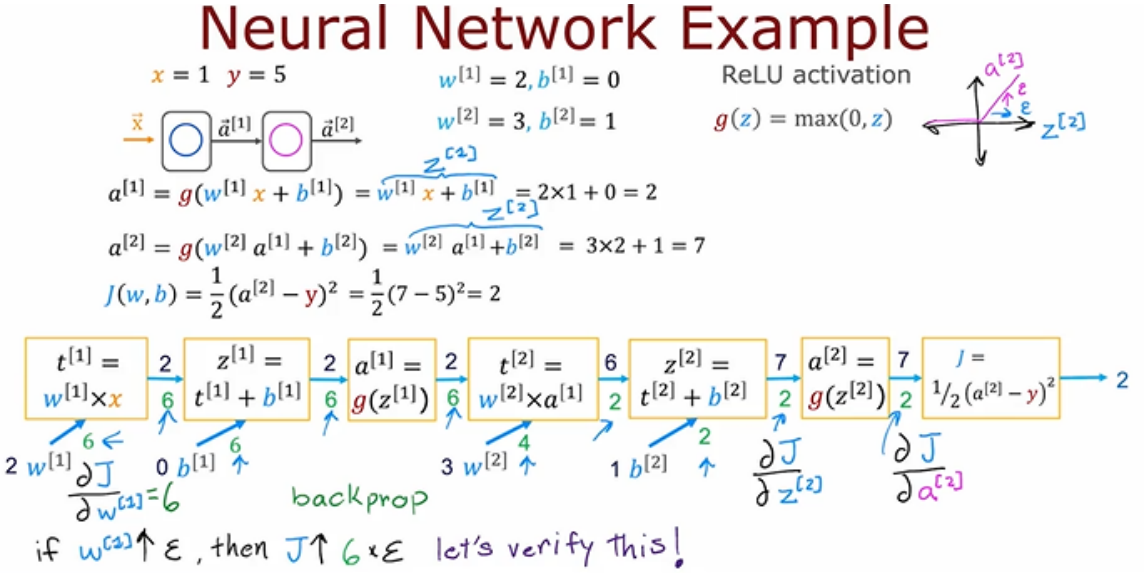

Figure 11: **Backpropagation on the large network (same figure reused).** Gradients flow right-to-left; numbers beside nodes are the computed partial derivatives.

Finite-difference sanity check (intuition)

Pick one parameter (e.g. $w^{[1]}$) and increase it by a tiny ε to see the approximate effect on the cost:

$$\Delta J \approx \frac{\partial J}{\partial w^{[1]}} \varepsilon = 6\varepsilon.$$

So if $w^{[1]}$ increases by $\varepsilon = 0.001$, we expect J to change by about 0.006. This helps verify the analytic gradient.

Gradient descent step (example)

Using learning rate α, parameter updates are:

$$w \leftarrow w - \alpha \frac{\partial J}{\partial w}, \qquad b \leftarrow b - \alpha \frac{\partial J}{\partial b}.$$

Example with $\alpha = 0.01$:

$$w^{[1]} \leftarrow 2 - 0.01 \times 6 = 1.94, \quad b^{[1]} \leftarrow 0 - 0.01 \times 6 = -0.06,$$

$$w^{[2]} \leftarrow 3 - 0.01 \times 4 = 2.96, \quad b^{[2]} \leftarrow 1 - 0.01 \times 2 = 0.98.$$

Key insights (why this scales to deep nets)

- **Local computations:** Each node only needs its local derivative and the gradient coming from the node(s) to its right.
- **Chain rule stitches local pieces:** Backprop applies the chain rule repeatedly to propagate gradients efficiently from outputs to every parameter.
- **ReLU simplifies derivatives:** For positive pre-activations the derivative of ReLU is 1, which often simplifies computations (but remember the derivative is 0 when pre-activation ≤ 0).
- **Automatic differentiation:** Modern frameworks build this same computation graph and do the backward pass for you, but understanding the numeric steps helps debug, sanity-check and reason about training dynamics.

Exercise (optional): perform the finite-difference check for $w^{[2]}$ by increasing $w^{[2]}$ by $\varepsilon = 0.001$ and confirm $\Delta J \approx 4\varepsilon$.

Building Machine Learning Systems Effectively

Hi, and welcome back. By now, you've seen a wide range of learning algorithms, including linear regression, logistic regression, and even deep learning (neural networks). Next week, you will explore decision trees as well. At this point, you already have many powerful tools in machine learning — but how do you use them effectively?

Efficient Use of Machine Learning Tools

I've observed that some teams may take six months to build a machine learning system that a more skilled team could complete in just a few weeks. The efficiency of how quickly

you can get a machine learning system to work well depends largely on how effectively you make decisions about what to do next during a project.

This week's lessons aim to provide key strategies on making those decisions efficiently — potentially saving you weeks or even months of time.

Example: Regularized Linear Regression

Suppose you've implemented regularized linear regression to predict housing prices, using the usual cost function (squared error plus a regularization term). However, after training, you find that the model produces unacceptably large prediction errors. What should you try next?

The cost function for **Regularized Linear Regression** is given by:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 + \frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2$$

where:

- m — number of training examples,
- $h_\theta(x^{(i)}) = \theta^T x^{(i)}$ — hypothesis or predicted value,
- $y^{(i)}$ — actual target value,
- λ — regularization parameter that controls overfitting,
- θ_j — model parameters (excluding θ_0).

Regularization adds the penalty term

$$\frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2$$

which discourages large parameter values and helps reduce overfitting.

When building a machine learning model, there are usually many directions you could take:

- Collect more training examples — sometimes more data helps significantly.
- Reduce the number of features — perhaps the current set is too large or noisy.
- Add new, relevant features — e.g., additional house attributes.
- Create polynomial features (x_1^2, x_2^2, $x_1 x_2$, etc.) to capture nonlinearities.
- Adjust the regularization parameter λ — it might be too large or too small.

Making Good Decisions in ML Development

In any given ML project, some of these strategies may yield substantial improvement, while others may not. The key skill lies in identifying where your time is best invested.

For instance, I've seen teams spend months collecting more data, expecting performance gains, only to find it doesn't help. The ability to diagnose what truly limits your model's performance can save enormous amounts of time and resources.

The Role of Diagnostics

This week, you will learn how to perform a series of diagnostics — tests that provide insight into what is or isn't working in your learning algorithm. These diagnostics help guide your next steps by answering questions such as:

- Is collecting more training data likely to help?
- Should I add or remove features?
- Is my regularization parameter well chosen?

Running diagnostics can take time, but they are often an excellent investment. They provide valuable information that can save you weeks or even months of trial and error.

Next Step: Evaluating Model Performance

Before improving your learning algorithm, it's crucial to learn how to **evaluate its performance**. In the next lesson, we'll explore effective methods to assess how well your model is doing — and how to use those evaluations to decide what to do next.

Choosing the Regularization Parameter λ

In the previous lesson, we saw how different choices of the polynomial degree D affect the bias and variance of a learning algorithm and, consequently, its performance. In this section, we explore how the choice of the **regularization parameter** λ impacts the model's bias, variance, and overall performance.

Effect of Regularization on Model Complexity

Consider a fourth-order polynomial regression model trained with regularization:

$$f_{\mathbf{w},b}(x) = w_1x + w_2x^2 + w_3x^3 + w_4x^4 + b$$

The regularized cost function is:

$$J(\mathbf{w}, b) = \frac{1}{2m} \sum_{i=1}^{m} \left(f_{\mathbf{w},b}(x^{(i)}) - y^{(i)} \right)^2 + \frac{\lambda}{2m} \sum_{j=1}^{n} w_j^2$$

where λ is the **regularization parameter** controlling the trade-off between fitting the training data and keeping the parameters w_j small.

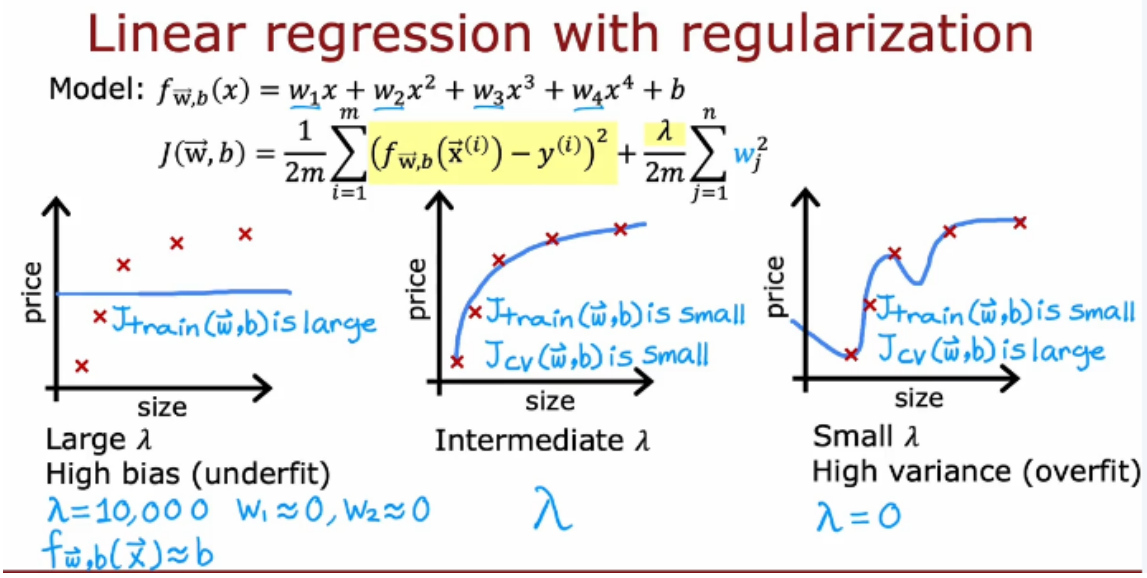

Figure 12: **Effect of λ on bias and variance.** Large λ leads to high bias (underfitting), small λ leads to high variance (overfitting), and an intermediate λ yields good generalization.

Understanding the Impact of λ

- **Large λ (e.g., $\lambda = 10{,}000$):** The algorithm heavily penalizes large weights, resulting in parameters $w_1, w_2, \cdots \approx 0$. The model simplifies to approximately $f_{\mathbf{w},b}(x) \approx b$, producing an almost flat line. This model has **high bias** and **underfits** the data, with large training error J_{train}.

- **Small λ (e.g., $\lambda = 0$):** With no regularization, the model fits the training data very closely, leading to a highly complex polynomial curve. This causes **high variance** and **overfitting**, characterized by a small J_{train} but large J_{cv} (cross-validation error).

- **Intermediate λ:** A moderate value of λ provides the best balance between bias and variance. Both J_{train} and J_{cv} remain low, indicating good generalization performance.

Selecting λ Using Cross-Validation

To choose an optimal λ, we use the cross-validation set to evaluate performance for multiple λ values. The procedure is as follows:

1. Select a range of candidate values for λ, e.g. $\{0, 0.01, 0.02, 0.04, 0.08, \ldots, 10\}$.
2. For each value of $\lambda^{(i)}$, train the model by minimizing:

$$\min_{\mathbf{w},b} \; J_{\text{train}}^{(\lambda^{(i)})}(\mathbf{w}, b)$$

   yielding parameters $(\mathbf{w}^{\langle i \rangle}, b^{\langle i \rangle})$.
3. Compute the cross-validation error:

$$J_{\text{cv}}^{(\lambda^{(i)})} = J_{\text{cv}}\left(\mathbf{w}^{\langle i \rangle}, b^{\langle i \rangle}\right)$$

4. Choose the λ that gives the smallest J_{cv}:

$$\lambda^* = \arg\min_{\lambda} J_{\text{cv}}^{(\lambda)}$$

   and denote the corresponding parameters as $(\mathbf{w}^{\langle 5 \rangle}, b^{\langle 5 \rangle})$.
5. Finally, evaluate the test set error to estimate generalization:

$$J_{\text{test}}\left(\mathbf{w}^{\langle 5 \rangle}, b^{\langle 5 \rangle}\right)$$

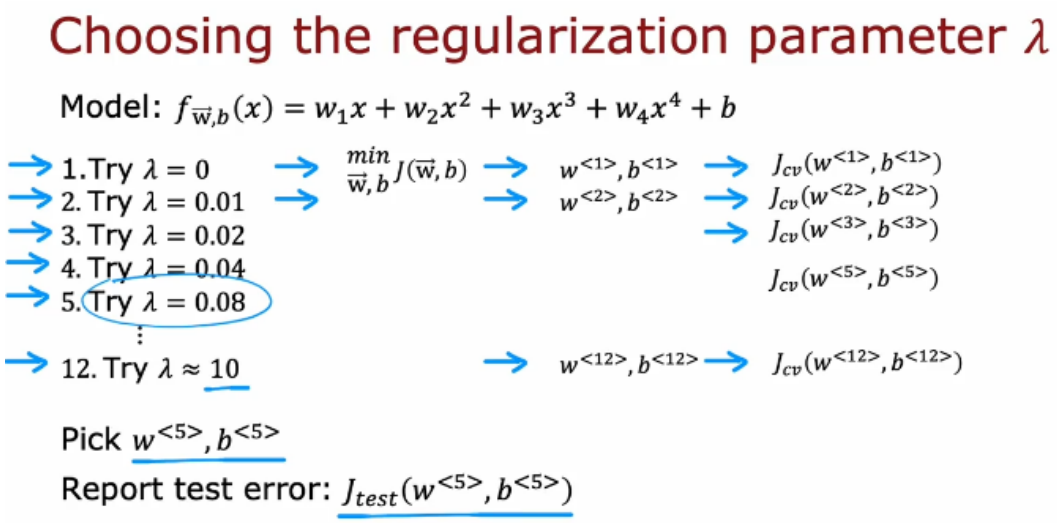

Figure 13: **Cross-validation procedure for selecting λ.** Multiple λ values are tested, cross-validation errors are compared, and the λ giving the lowest J_{cv} is selected.

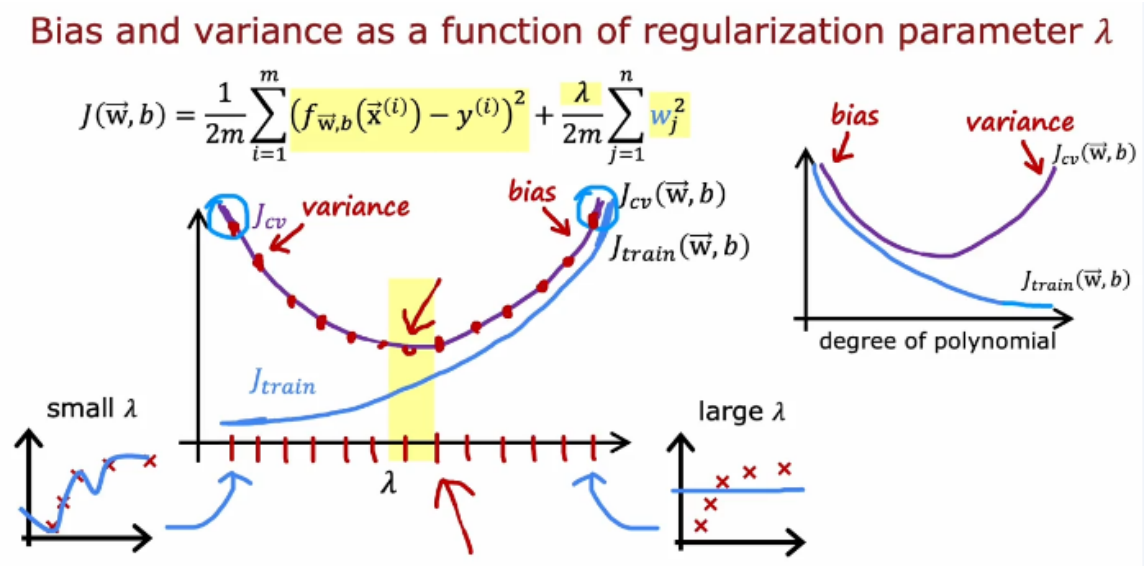

Figure 14: Bias and variance as a function of regularization parameter λ

Summary

- The regularization parameter λ directly controls model complexity.
- A very large λ increases bias (underfitting), while a very small λ increases variance (overfitting).
- Cross-validation is an effective way to automatically select λ that minimizes J_{cv} and improves generalization.

Thus, through this process, we can efficiently find the optimal regularization strength and achieve a balance between fitting the data well and maintaining robustness to unseen examples.

Bias and Variance as a Function of Regularization Parameter λ

To further hone intuition about what this algorithm is doing, let's take a look at how training error and cross-validation error vary as a function of the parameter λ.

In this figure, the x-axis is annotated with the value of the regularization parameter λ. When $\lambda = 0$, there is no regularization, leading to a very wiggly curve with high variance. In this case, J_{train} is small while J_{cv} is large because the model performs well on the training data but poorly on the cross-validation data.

At the other extreme, for very large values of λ (e.g., $\lambda = 10{,}000$), the model becomes overly constrained and underfits the data, resulting in high bias. Both J_{train} and J_{cv} become large.

The behavior of J_{train} as λ increases can be explained by the optimization cost function:

$$J(\mathbf{w}, b) = \frac{1}{2m} \sum_{i=1}^{m} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m} \sum_{j=1}^{n} w_j^2$$

As λ grows, the algorithm gives more importance to keeping $\|\mathbf{w}\|^2$ small, paying less attention to minimizing the training error. Hence, J_{train} increases with λ.

On the other hand, the cross-validation error J_{cv} exhibits a U-shaped pattern. When λ is too small or too large, the model performs poorly on cross-validation data, indicating overfitting or underfitting, respectively. There exists an intermediate λ that minimizes J_{cv} — this is the optimal value that balances bias and variance.

Cross-validation helps find this optimal λ by evaluating different candidate values (e.g., $\lambda = 0, 0.01, 0.02, \dots$) and selecting the one with the lowest J_{cv}.

Conceptually, this plot mirrors the earlier diagram showing model performance versus the degree of polynomial: the regions of high bias and high variance swap sides. In both cases, cross-validation is a powerful tool for selecting the right level of model complexity — either via the polynomial degree or the regularization strength λ.

In summary:

- Small $\lambda \Rightarrow$ low bias, high variance (overfitting)
- Large $\lambda \Rightarrow$ high bias, low variance (underfitting)
- Optimal λ minimizes cross-validation error J_{cv}

Learning Curves, High Bias, and High Variance

Understanding Learning Curves

Learning curves help us visualize how a learning algorithm performs as a function of the amount of **training data** it has. They show how well the model learns as it gains more experience.

Let J_{train} denote the training error, and J_{cv} denote the cross-validation error. On the horizontal axis, we have the training set size m_{train}, and on the vertical axis, we plot the error value.

- As m_{train} increases, the model has more data to learn from.
- Typically, J_{cv} (cross-validation error) decreases as more data is added because the model generalizes better.
- Meanwhile, J_{train} (training error) usually increases slightly with more data since the model cannot perfectly fit every new example.

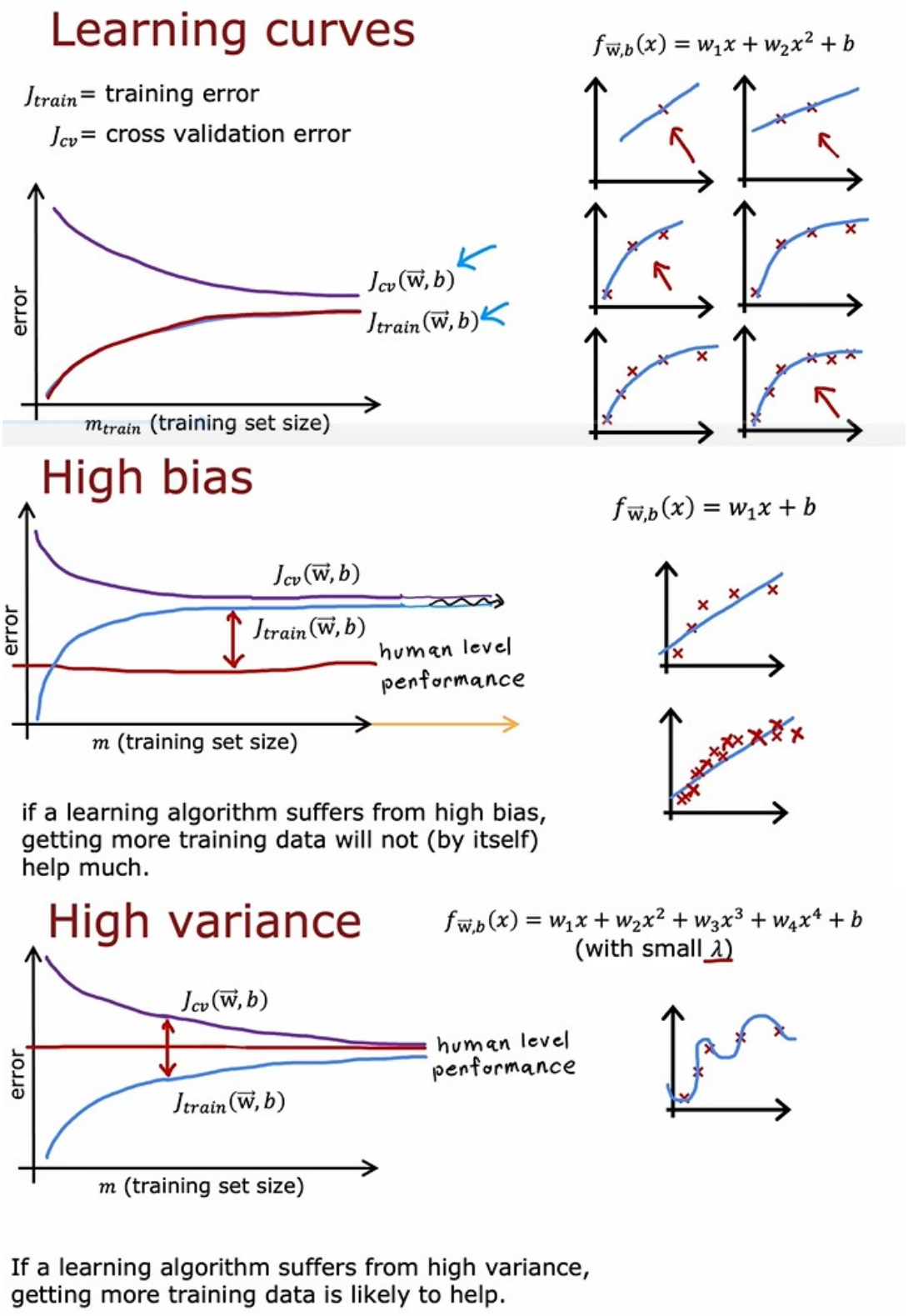

Figure 15: Learning curves showing training and cross-validation error for models with different bias and variance characteristics.

Why does the training error increase with data?

When the training set is very small, the model can easily overfit and achieve nearly zero error. As more data is introduced, it becomes harder to fit all training examples perfectly, causing J_{train} to rise slightly.

In general:

$$J_{\text{cv}}(w, b) > J_{\text{train}}(w, b)$$

because the model parameters are optimized for the training set, not unseen data.

High Bias (Underfitting)

High bias occurs when the model is too simple to capture the underlying structure of the data. For example, fitting a linear function $f_{w,b}(x) = w_1 x + b$ to a dataset that requires a quadratic curve.

- Both J_{train} and J_{cv} are large and close to each other.
- As training size increases, both errors flatten (plateau) and stop improving.
- The model cannot achieve human-level performance because it underfits the data.

Key insight: Increasing training data will *not* help reduce error in the high-bias scenario. The model's structure (e.g., linear when the problem is nonlinear) limits its capability.

If model has high bias, more data does not help.

High Variance (Overfitting)

High variance occurs when the model is too complex and fits noise in the training data instead of learning the underlying pattern. For example, fitting a high-order polynomial:

$$f_{w,b}(x) = w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4 + b$$

with a very small regularization parameter λ.

- J_{train} is very low (model fits training data perfectly).
- J_{cv} is much higher — the model fails to generalize.
- The gap between J_{cv} and J_{train} is large.

Key insight: In this case, adding more training data can help reduce variance. As m_{train} increases, J_{train} rises slightly, and J_{cv} decreases, narrowing the gap between them.

If model has high variance, more data is likely to help.

Scenario	Training Error (J_{train})	Cross-Validation Error (J_{cv})
High Bias (Underfitting)	High	High, similar to J_{train}
High Variance (Overfitting)	Low	High, much larger than J_{train}
Ideal Model	Low	Low, close to each other

Table 1: Comparison of bias and variance using training and validation errors.

Practical Interpretation

When diagnosing your learning algorithm:

- Plot J_{train} and J_{cv} versus training set size.
- Observe whether your learning curve resembles a **high-bias** or **high-variance** pattern.
- Decide your next step:
  - High bias: increase model complexity (e.g., use more features, deeper network).
  - High variance: get more training data or use regularization.

Summary

In summary: Learning curves are a vital tool for understanding model performance. They help determine whether an algorithm is limited by bias (underfitting) or variance (overfitting) and guide what corrective actions to take next.

Speech Recognition Example and Bias/Variance Analysis

Let's look at some concrete numbers for what J_{train} and J_{cv} might be, and see how you can judge if a learning algorithm has high bias or high variance. As an example, consider a speech recognition system designed for mobile voice search queries such as "What is today's weather?" or "Coffee shops near me." The system attempts to output accurate text transcripts of these audio clips.

Suppose that:

$$\text{Human-level performance} = 10.6\%$$
$$J_{train} = 10.8\%$$
$$J_{cv} = 14.8\%$$

Even though the training error seems high (10.8%), notice that the human-level performance is 10.6%. Since no model can realistically outperform noisy human-transcribed data, a small gap of 0.2% between J_{train} and human-level error suggests that the system

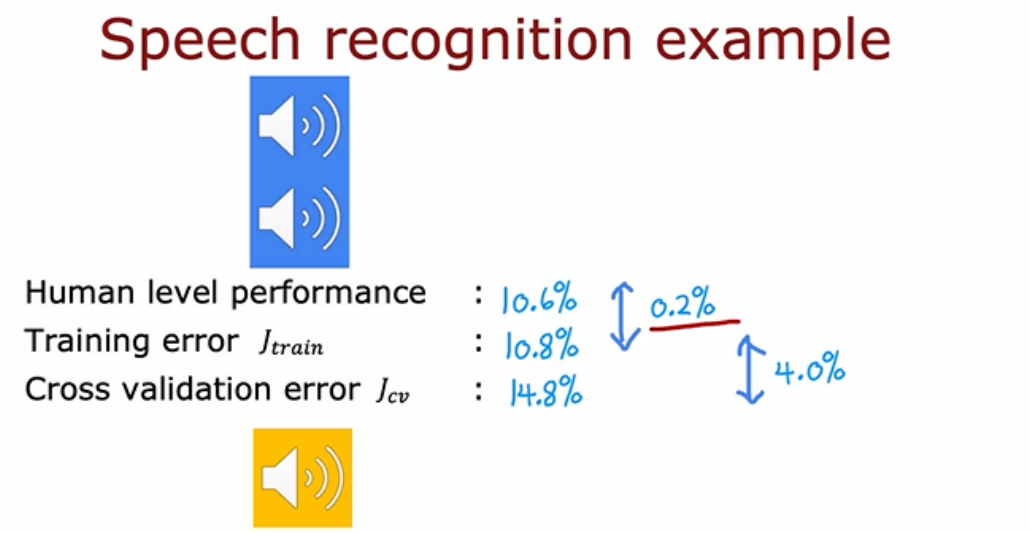

Figure 16: Speech recognition example illustrating human-level performance, training error (J_{train}), and cross-validation error (J_{cv}).

has low bias. However, the larger 4.0% gap between J_{train} and J_{cv} indicates that the system suffers from **high variance**—it performs well on training data but generalizes poorly to unseen data.

Thus, we conclude that the algorithm has **high variance** rather than high bias.

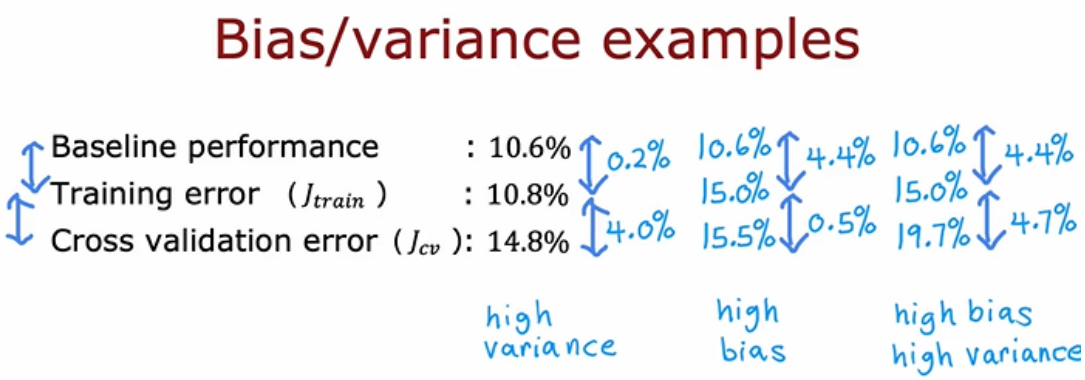

Figure 17: Different bias/variance examples showing the relationship between baseline performance, training error, and cross-validation error.

To further analyze, it's important to define a **baseline level of performance**, such as human-level accuracy or a strong competing model. This baseline helps determine whether training error is actually "high" or "acceptable."

- A large gap between the baseline and J_{train} $\Rightarrow$ **High bias problem**.
- A large gap between J_{train} and J_{cv} $\Rightarrow$ **High variance problem**.

In some cases, both gaps can be large, leading to both **high bias and high variance**. For example:

$$\begin{aligned} \text{Baseline performance} &= 10.6\% \\ J_{train} &= 15.0\% \\ J_{cv} &= 19.7\% \end{aligned}$$

Here, the model performs significantly worse than the human baseline (high bias), and it also performs poorly on cross-validation (high variance).

This diagnostic method—comparing human-level or baseline performance with J_{train} and J_{cv}—provides a more realistic measure of your model's bias and variance characteristics, especially for unstructured data such as audio, text, and images.

Debugging a Learning Algorithm

When training a machine learning model, it's crucial to understand whether poor performance is due to **high bias** (underfitting) or **high variance** (overfitting). By examining the training error J_{train} and cross-validation error J_{cv}, or by plotting a learning curve, we can often determine which issue is present and take the right corrective steps.

Regularized Linear Regression

For regularized linear regression, the cost function is given by:

$$J(\mathbf{w}, b) = \frac{1}{2m} \sum_{i=1}^{m} \left(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)} \right)^2 + \frac{\lambda}{2m} \sum_{j=1}^{n} w_j^2$$

Here:

- $\mathbf{w}$ – weight parameters
- b – bias term
- λ – regularization parameter controlling model complexity

The first term measures the prediction error on training data, while the second term penalizes large weights to prevent overfitting.

Diagnosing High Bias vs High Variance

If your model makes large prediction errors, it could be due to either high bias or high variance. Different strategies can be used to fix each issue.

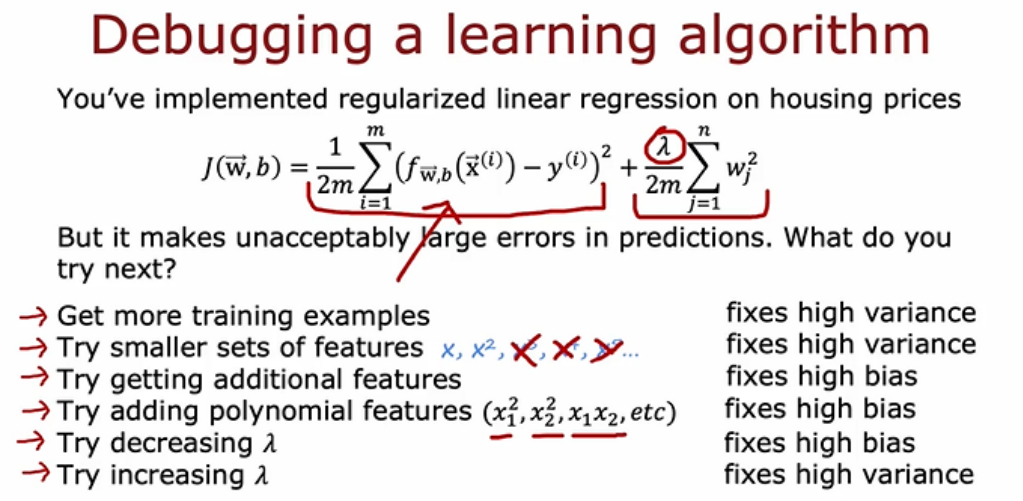

Figure 18: Debugging a Learning Algorithm — Regularized Linear Regression example.

Strategy	**Helps Fix**
Get more training examples	High Variance
Try smaller sets of features	High Variance
Try getting additional features	High Bias
Try adding polynomial features (x_1^2, x_2^2, x_1x_2, etc.)	High Bias
Try decreasing λ	High Bias
Try increasing λ	High Variance

Understanding the Trade-off

- **High Bias (Underfitting):** The model performs poorly even on training data. It lacks the complexity needed to capture patterns in the data. *Fixes:* Add more or more complex features, decrease λ, or use a more powerful model.
- **High Variance (Overfitting):** The model performs well on training data but poorly on validation/test data. *Fixes:* Get more data, reduce the number of features, or increase λ.

Key Takeaways

- More training data helps most when your model has high variance.
- Simplifying your model (fewer features or larger λ) reduces variance.
- Increasing model complexity (adding features, decreasing λ) reduces bias.
- Randomly discarding training examples never helps fix high bias.

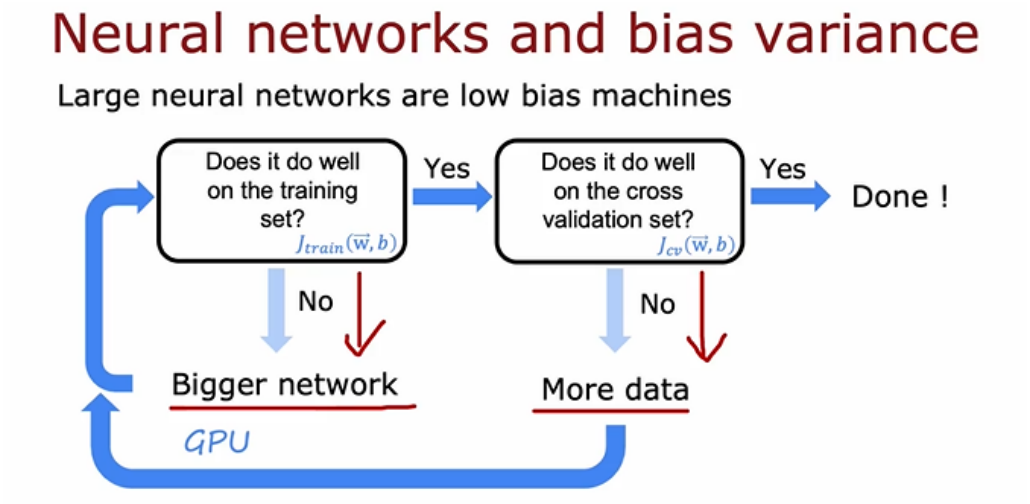

Figure 19: Neural networks and bias-variance relationship.

Bias and variance are powerful concepts that guide the iterative improvement of machine learning algorithms. Mastery of these ideas comes through continuous experimentation and practice.

Neural Networks and Bias-Variance

Bias and Variance in Neural Networks

We have seen that both high bias and high variance are detrimental to a model's performance. One of the key reasons neural networks have achieved such remarkable success is that, when combined with large datasets, they provide a new way to address both bias and variance effectively.

Before neural networks became mainstream, machine learning engineers often discussed the **bias-variance tradeoff** in terms of balancing model complexity—such as the degree of a polynomial or the strength of a regularization parameter λ—to ensure neither bias nor variance is excessively high. However, large neural networks offer a way out of this tradeoff, provided we can manage computational costs and have enough data.

Large neural networks tend to be **low bias machines**. That is, if you make your network sufficiently large, it can almost always fit the training set well (assuming it's not extremely large). This gives us a new recipe for reducing bias or variance:

- **Step 1:** Train the neural network on the training set and evaluate $J_{\text{train}}(\mathbf{w}, b)$.
- **Step 2:** If the training error is high (high bias), increase the size of the network — either by adding more layers or more units per layer.
- **Step 3:** Once the model performs well on the training set, evaluate $J_{\text{cv}}(\mathbf{w}, b)$ on the cross-validation set.

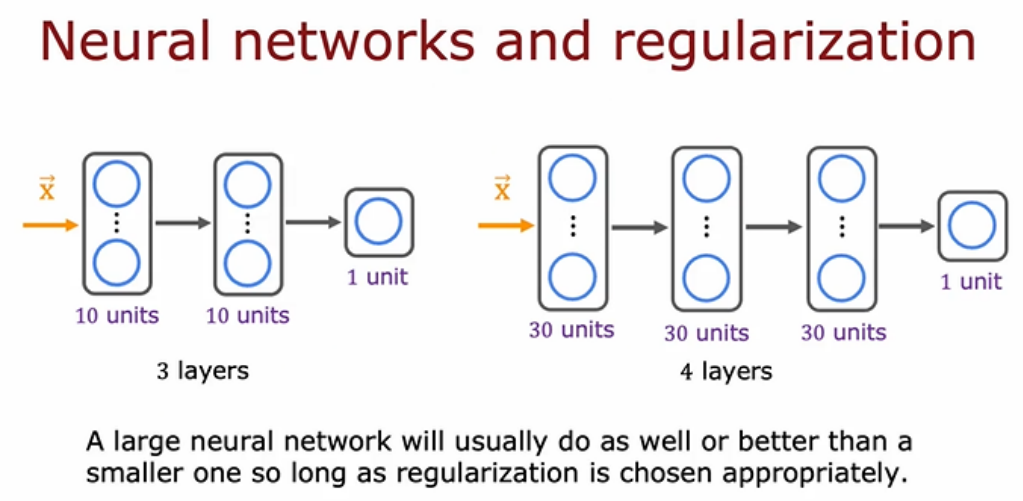

Figure 20: The role of regularization in large neural networks.

- **Step 4:** If the cross-validation error is high (indicating high variance), collect more training data to improve generalization.

By iterating through this loop — increasing model capacity to reduce bias and collecting more data to reduce variance — we can build powerful models that generalize well. However, the two main limitations are:

1. **Computational cost:** Larger networks require more computational resources, often GPUs or TPUs, to train efficiently.

2. **Data availability:** Sometimes, sufficient training data isn't available to reduce variance further.

This cycle explains much of the success behind modern deep learning — large datasets combined with large neural networks and sufficient computational power enable strong performance on complex tasks.

Neural Networks and Regularization

A large neural network will typically perform as well as, or better than, a smaller one if regularization is chosen appropriately. For instance, increasing the number of layers or hidden units generally improves performance, but without regularization, the risk of overfitting increases.

The regularization term for a neural network is similar to that used in linear or logistic regression:

$$J(\mathbf{w}, b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{j=1}^{n} w_j^2$$

where $L(\cdot)$ represents the loss function (e.g., squared error or logistic loss), and the second term penalizes large weights to prevent overfitting.

In practice, the bias term b is often not regularized, although doing so has minimal effect. Implementing L2 regularization in deep learning frameworks such as TensorFlow can be done as follows:

```
Dense(64, activation='relu', kernel_regularizer=l2(0.01))
```

Regularization ensures that larger neural networks generalize well to unseen data. Therefore, it's often safe — and even beneficial — to use larger networks, provided they are regularized properly.

Key Takeaways

- Larger neural networks tend to be **low bias** models.
- Regularization (such as L2) helps control **variance**.
- It rarely hurts to make a neural network larger — as long as it is well-regularized.
- High bias can be reduced by increasing model size; high variance can be mitigated by adding more data or stronger regularization.

Iterative Loop of Machine Learning Development

In this section, we explore the typical process of developing a machine learning (ML) system using the example of a **spam email classifier**. The ML development process is inherently **iterative**, involving repeated cycles of design, training, and diagnostics to achieve the desired performance. This section is divided into three major parts:

Part 1: The Iterative Loop of Machine Learning Development

The process of building a machine learning system rarely succeeds in a single attempt. Instead, it follows an iterative loop consisting of three main stages:

1. **Choosing the architecture:** Decide the model type (e.g., logistic regression, neural network), the data to use, and key hyperparameters such as learning rate, number of layers, and regularization strength.
2. **Training the model:** Implement and train the model on the dataset. The first model is rarely perfect—it typically performs below expectations.
3. **Running diagnostics:** Analyze the model's bias, variance, and errors. These diagnostics guide what to improve next.

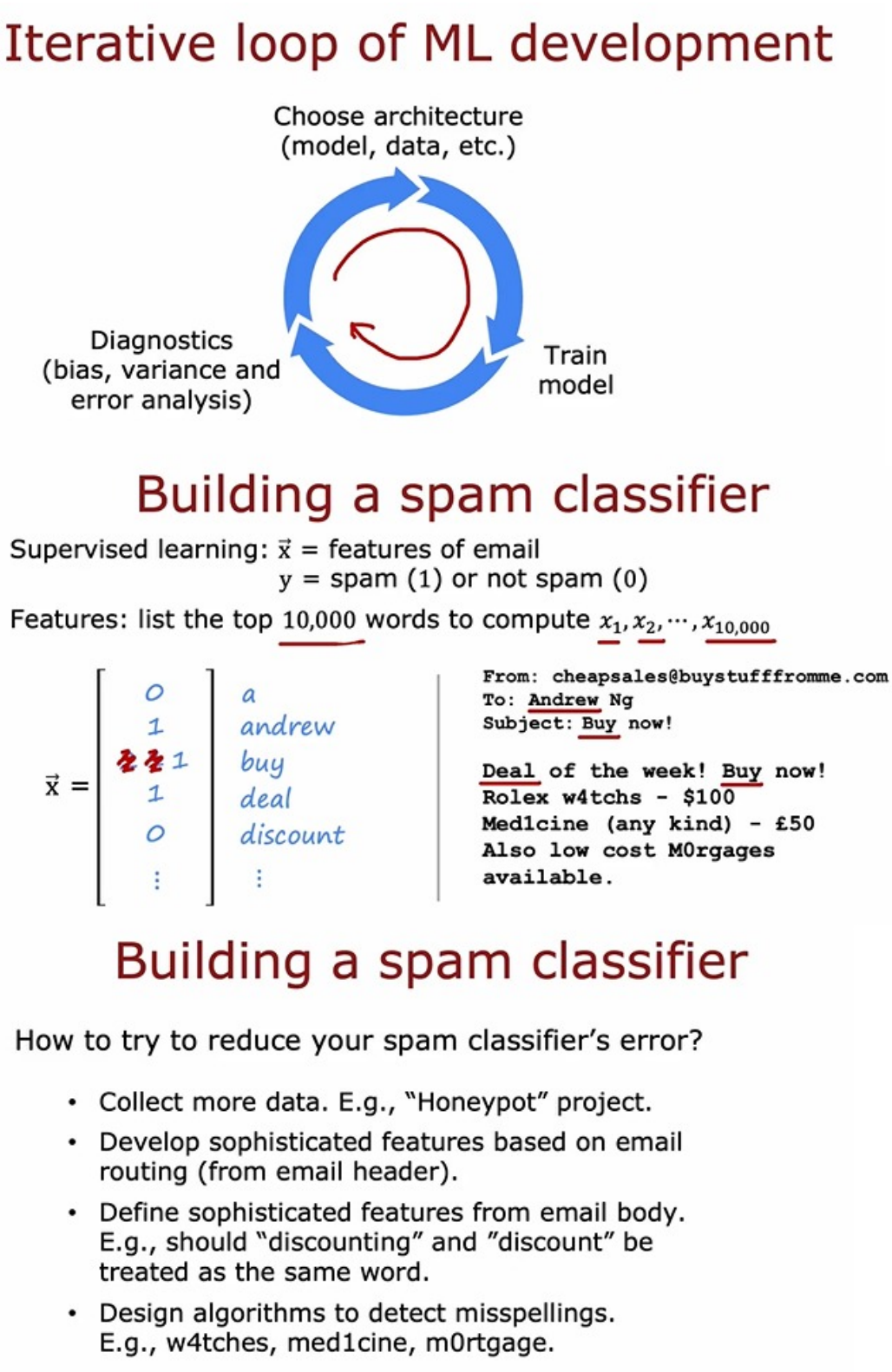

Figure 21: Iterative loop of machine learning development and the process of building a spam classifier.

After analyzing diagnostics, adjustments are made—such as collecting more data, modifying the model architecture, tuning regularization, or changing feature sets. This loop is repeated multiple times until the desired accuracy is achieved. The iterative process can be summarized as:

$$\text{Architecture choice} \rightarrow \text{Train model} \rightarrow \text{Diagnostics} \rightarrow \text{Refine model}$$

This approach ensures that each iteration moves the system closer to optimal performance.

Part 2: Building a Spam Classifier

Let's consider an example: **Building a spam classifier**. The goal is to distinguish between spam and non-spam (ham) emails using supervised learning.

- Input features: $\vec{x}$ = features extracted from an email.
- Output label: $y = 1$ if spam, $y = 0$ if not spam.

A common method is to define features from the most frequent words in the dataset. Suppose we take the top 10,000 words across all emails. Then, each email can be represented as:

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{10,000} \end{bmatrix} \quad \text{where } x_i = \begin{cases} 1, & \text{if word}_i \text{ appears in email} \\ 0, & \text{otherwise} \end{cases}$$

For example, if the dictionary contains the words "a," "Andrew," "buy," "deal," and "discount," and the email contains the words "Andrew," "buy," and "deal," then:

$$\vec{x} = [0, 1, 1, 1, 0, \dots]$$

This binary representation can also be extended to word counts (term frequencies). Once the features are extracted, a classification model such as logistic regression or a neural network can be trained to predict whether an email is spam based on these inputs.

$$\hat{y} = h_\theta(\vec{x}) = \frac{1}{1 + e^{-\theta^T x}}$$

The model learns parameters θ that minimize the loss function over all training examples.

Part 3: Reducing Spam Classifier Error

Once the initial classifier is trained, it may not perform perfectly. The next step is to iteratively improve it. Here are systematic strategies to reduce errors:

1. **Collect more data:** For instance, using a "honeypot project," where fake email addresses are created to attract spam and generate labeled training data.
2. **Develop advanced features:** Extract information from the email routing or header fields (e.g., server path) to identify potential spam sources.
3. **Enhance text-based features:**
   - Normalize similar words (e.g., "discount" and "discounting").
   - Detect misspellings or deliberate obfuscations such as "w4tches," "med1cine," or "m0rtgage."
4. **Iterate using diagnostics:** Analyze bias and variance to guide decisions.
   - High bias → increase model complexity (add layers or features).
   - High variance → gather more data or apply stronger regularization.

This approach emphasizes diagnostic-driven development—using error analysis to identify which improvements are most impactful. For example, if your algorithm exhibits high bias, collecting more data (like in a honeypot project) may not help much. But if it has high variance, that approach could be highly beneficial.

Key Insights

- Machine learning development is an **iterative loop**—not a linear process.
- Each iteration improves model performance through diagnostics and refinement.
- The spam classifier example illustrates how feature engineering, data collection, and error analysis combine to create a robust system.

Error Analysis in Learning Algorithms

In terms of the most important ways to help you run diagnostics to choose what to try next to improve your learning algorithm performance, **bias and variance** is probably the most important idea, and **error analysis** would probably be second on the list.

Understanding Error Analysis

Concretely, let's say you have $m_{cv} = 500$ cross-validation examples and your algorithm misclassifies 100 of these 500 examples. The error analysis process refers to manually looking through these 100 examples and trying to gain insights into where the algorithm is going wrong.

- Identify examples that were misclassified.
- Group them into categories with common traits (e.g., topic, formatting, spelling).
- For instance:
  - 21 examples: Pharmaceutical spam (medicine/drug sales)
  - 3 examples: Deliberate misspellings
  - 7 examples: Unusual email routing
  - 18 examples: Phishing emails (password theft)
  - Remaining examples: Embedded image spam (spam message inside an image)

This breakdown shows that **pharmaceutical spam** and **phishing emails** are major problem areas, while deliberate misspellings are minor. Even if you build a sophisticated algorithm to detect misspellings, it only solves 3 out of 100 errors — meaning limited overall impact.

Insights and Prioritization

- Focus on categories with large impact (e.g., pharmaceutical or phishing emails).
- Low-impact issues (like minor spelling errors) can be deprioritized.
- Categories can overlap — one email might fit multiple error types.
- If your dataset is large (e.g., $m_{cv} = 5000$ and 1000 misclassifications), manually inspect a random subset ($\approx$ 100–200) to estimate error distribution.

Using Insights to Improve the Model

After analysis:

- Collect more data from problematic categories (e.g., more phishing or pharma spam).
- Engineer new features:
  - Drug or brand name detection features.

  - Suspicious URL patterns for phishing detection.

- Consider model adjustments based on bias-variance diagnostics:
  - If high variance → add more data or regularization.
  - If high bias → increase model complexity or improve features.

When Error Analysis is Harder

- Error analysis is easier when humans can recognize the correct label (e.g., spam detection).
- Harder tasks include predicting user clicks or preferences, where even humans cannot easily judge correctness.

Conclusion

By carefully performing error analysis, you can:

- Identify which types of errors matter most.
- Prioritize model improvements that yield the biggest gains.
- Avoid wasting time on low-impact fixes.

This process can save weeks or months of unproductive experimentation and help you make data-driven decisions on how to improve your learning algorithm.

Adding or Creating More Data

In this section, we discuss practical strategies for adding or creating data to improve machine learning model performance. While approaches vary depending on the application, many of the following methods can be useful for a wide range of tasks.

Motivation

In training machine learning algorithms, it often feels like we could always use more data. However, collecting more data indiscriminately can be slow and expensive. Instead, it is often more effective to collect data strategically — focusing on areas where error analysis suggests the model performs poorly.

Targeted Data Collection

From error analysis, if we observe that certain categories (e.g., pharmaceutical spam) are frequently misclassified, then rather than collecting more data of every kind, we can:

- Focus on collecting additional data for those specific problem areas.
- Label existing unlabeled data to increase examples in underperforming categories.
- Achieve higher performance with smaller, more focused data additions.

Example:

- If spam classification shows poor performance on pharmaceutical spam,
- Label more pharmaceutical spam examples from a pool of unlabeled emails,
- Instead of labeling random emails of all categories.

This targeted approach improves performance more efficiently than collecting a large general dataset.

Data Augmentation

Beyond collecting new data, a widely used technique—especially for image and audio data—is **data augmentation**. This involves transforming existing training examples (x, y) to create new ones while keeping the label unchanged.

Image Data Example:

- For OCR tasks (recognizing letters A–Z), create new examples by:
  - Rotating, enlarging, or shrinking the image.
  - Adjusting brightness or contrast.
  - Flipping horizontally (if applicable).
- Each transformed image remains a valid example of the same letter.
- This teaches the algorithm to generalize across distortions.

Advanced Image Augmentation:

- Apply a grid and perform *random warping* to generate realistic distortions.
- Converts one training image into many, improving robustness.

Audio Data Example:

- Original clip: "What is today's weather?"

- Augmentations:
  - Add background noise (crowd, car, office).
  - Simulate poor microphone quality or phone transmission.
- Each augmented sample teaches the system to handle realistic conditions.

Tip: Ensure that augmentations mimic real test-time variations — distortions should represent real-world noise or transformations the model will encounter.

Data Synthesis

While augmentation modifies existing examples, **data synthesis** creates brand-new examples from scratch.

Example: Photo OCR

- Real data: cropped letter images from photos.
- Synthetic data: generate letters using computer fonts and various styles (colors, contrasts, backgrounds).
- The synthetic examples resemble real data and significantly enlarge the dataset.

Writing code to generate realistic synthetic data can be time-consuming, but it can drastically boost performance—especially in computer vision tasks.

Data-Centric vs. Model-Centric Approaches

Traditional machine learning research was **model-centric**:

- Data was fixed.
- Focus was on improving model architecture or training algorithms.

However, in practice, taking a **data-centric approach**—improving the quality, quantity, and diversity of data—often yields larger performance gains.

Data-Centric Examples:

- Collect more examples from underperforming categories (e.g., pharmaceutical spam).
- Apply effective data augmentation or synthesis techniques.
- Clean and balance datasets to improve label accuracy.

When Data is Limited: Transfer Learning

Sometimes, collecting or synthesizing data is difficult. In such cases, **transfer learning** can help:

- Use knowledge from a related but different task.
- Pretrain on a large dataset, then fine-tune on your smaller dataset.
- Example: Use a neural network pretrained on ImageNet for a medical image classification task.

Summary

- Adding more data improves performance, but should be targeted and efficient.
- Use **data augmentation** to create new examples from existing ones.
- Use **data synthesis** to generate entirely new examples.
- Adopt a **data-centric approach** to engineering your ML pipeline.
- Apply **transfer learning** when data collection is difficult.

These strategies together make your model more robust, generalizable, and efficient to train.

Transfer Learning: Learn from Large Models When Data is Scarce

When you do not have much labeled data for your target task, **transfer learning** is one of the most effective techniques available. The idea is simple and powerful: leverage a model that was trained on a large related dataset to give your smaller model a strong head start.

High-level two-step recipe (what you do)

1. **Supervised pretraining:** Train a neural network on a very large dataset (e.g., ImageNet with ∼1M images and 1000 classes) until it learns good general-purpose image features. This produces parameters

$$W^{[1]}, b^{[1]},\ W^{[2]}, b^{[2]},\ \ldots,\ W^{[L]}, b^{[L]}.$$

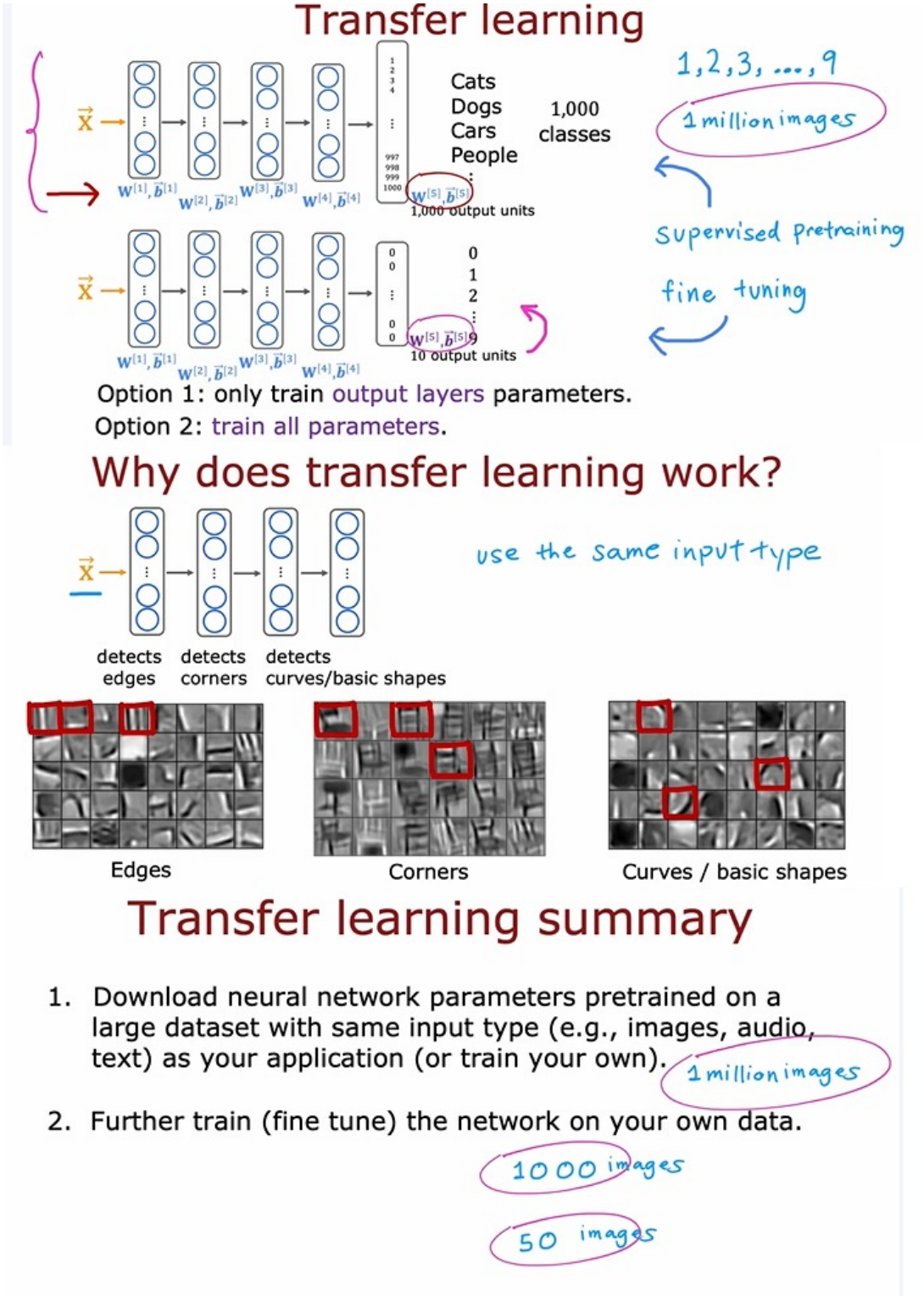

Figure 22: Transfer learning: supervised pretraining on a large dataset, then fine-tuning on a smaller dataset for your task. Visualizations show low-level features (edges, corners, curves) learned by early layers.

2. **Fine-tuning (transfer):** Replace the original output layer (e.g., 1000-way softmax) with a new output layer suited to your task (e.g., 10 classes for digits). Then either

    - **Option A — Freeze earlier layers:** Keep earlier layers fixed and only train the new output layer. Good when you have very little data.
    - **Option B — Fine-tune all layers:** Initialize all weights with pretrained values and continue training (fine-tuning) on your dataset. Good when you have more data and want better task-specific performance.

Why transfer learning works (intuitive explanation)

- Early convolutional layers learn *generic* visual primitives: edges, gradients, corners. These primitives are useful across many vision tasks (cats, cars, handwritten digits, etc.).
- Later layers combine primitives into higher-level patterns. By reusing the early-layer weights, the new model does not need to learn low-level detectors from scratch.
- Pretraining provides a parameter initialization that is already in a good region of parameter space — fine-tuning can reach strong performance with far fewer labeled examples.

Concrete example: digit recognition with limited data

1. Pretrain a network on a large dataset (1M images, 1000 classes) to learn $W^{[1:4]}, b^{[1:4]}$ and $W^{[5]}, b^{[5]}$ for the 1000-way output.
2. For digit recognition (10 classes): remove the 1000-way output, add a new 10-way output layer with parameters $W^{[5]}_{\text{new}}, b^{[5]}_{\text{new}}$.
3. Train on your small digit dataset:

    - If you have very few labeled examples (e.g., 50–1,000), freeze layers 1–4 and only train the new output layer.
    - If you have more examples (e.g., tens of thousands), initialize with pretrained weights and fine-tune all layers.

Practical tips and variations

- **Match input type:** Pretraining and target tasks should share the same input modality (images, audio, text). An image-pretrained model is not helpful for audio.

- **Layer freezing schedule:** A common practical strategy is to (1) train only the new output layer for a few epochs, then (2) unfreeze some earlier layers and fine-tune at a smaller learning rate.
- **Learning rates:** When fine-tuning pretrained layers, use a smaller learning rate than for the randomly initialized layers (the final layer).
- **Regularization:** Continue using regularization (weight decay, dropout) during fine-tuning to avoid overfitting when data is limited.
- **Data scale examples:** With a good pretrained model you may get good results with as few as a few dozen to a few thousand labeled examples, depending on task similarity.
- **Pretrained models:** Many pretrained models are publicly available (e.g., ResNet, VGG, BERT). Downloading and fine-tuning these saves massive compute and time.

Why early-layer features generalize

Visualizations of early-layer filters typically show edge detectors, corner detectors, and small curve detectors. These building blocks are common across many visual tasks, which explains why pretraining on generic image datasets helps many downstream vision problems.

Summary checklist

- If you have **very little data**: freeze pretrained layers, train only output layer.
- If you have **moderate data**: initialize with pretrained weights and fine-tune all (or some) layers.
- Always ensure **input modality and size** match between pretrained model and your task.
- Use smaller learning rates for pretrained weights and larger rates for newly initialized layers.
- Leverage publicly available pretrained models whenever possible.

Transfer learning is one of the fastest practical ways to obtain strong performance when labeled data is scarce — it reuses the community's large-scale training efforts and lets you build accurate models with far less data and compute.

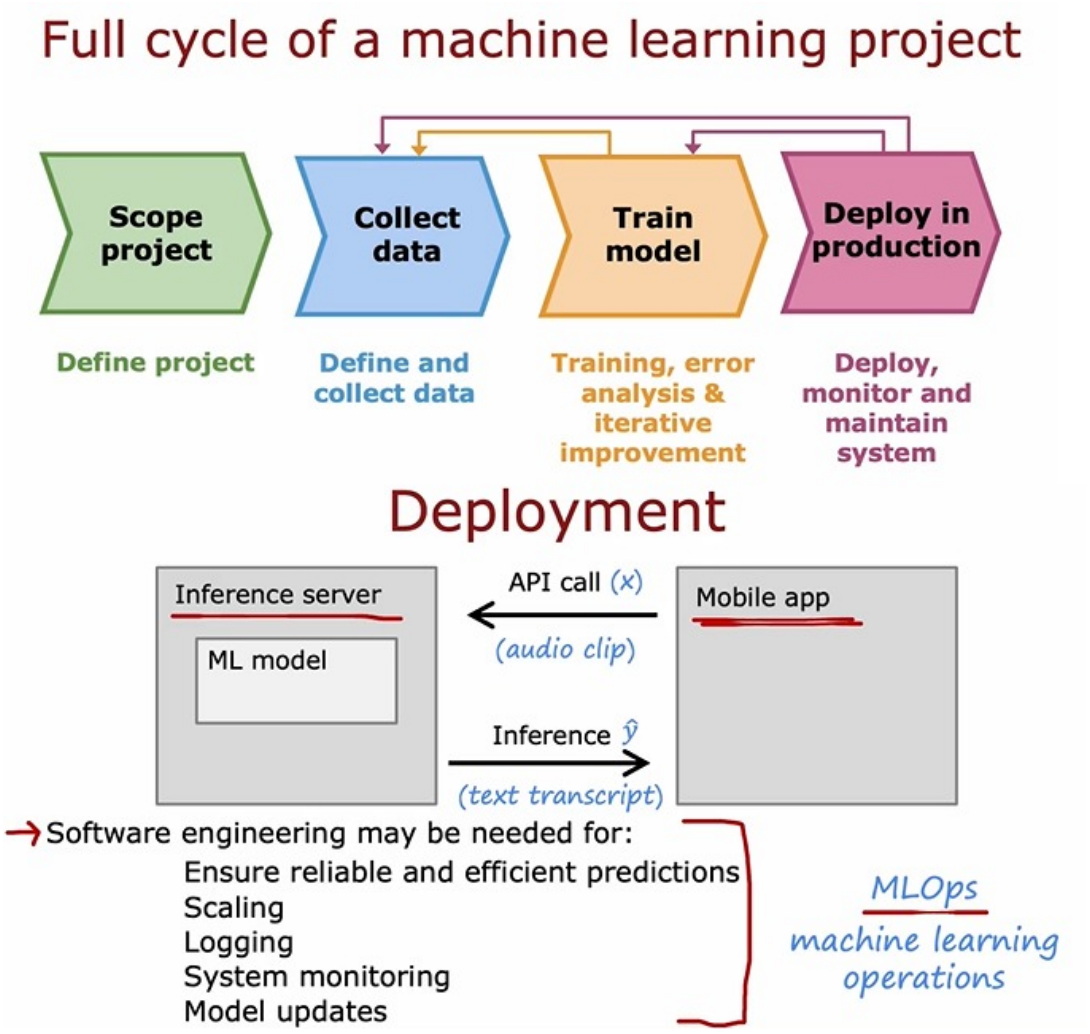

Figure 23: The full cycle of a machine learning project: from scoping the project to collecting data, training models, and deploying them into production (MLOps).

Full Cycle of a Machine Learning Project

When building a valuable machine learning (ML) system, training the model is only one part of the overall process. A successful ML project requires planning across the entire lifecycle — from defining the problem to deploying and maintaining the model in production. Below, we outline each stage of this cycle.

1. Scope the Project (Define the Goal)

The first step is to clearly define what problem you want to solve and what success looks like.

- Identify the application domain — for example, speech recognition for voice search.
- Determine the system's purpose: What inputs (x) and outputs ($\hat{y}$) are expected?
- Set performance goals, such as accuracy, latency, or cost constraints.

A well-scoped project provides a clear direction for what data to collect and what type of model to train.

2. Collect Data

Once the project is defined, the next step is to gather the right dataset.

- Identify and obtain labeled data (e.g., audio clips and their transcripts for speech recognition).
- Perform data cleaning, augmentation, and labeling to ensure high quality.
- Revisit data collection iteratively — error analysis may reveal where more or better data is needed (e.g., noisy environments, accents, lighting variations).

Key idea: Data collection is rarely "done once." It evolves with your understanding of model performance.

3. Train the Model

Train a suitable ML model on the collected dataset.

- Start with an initial model architecture and baseline metrics.
- Perform **error analysis** to understand model weaknesses.
- Iterate: improve model design, collect additional data, and re-train.

- Use techniques such as bias-variance analysis to decide whether you need more data or model regularization.

This cycle of training, evaluation, and improvement may repeat many times before deployment.

4. Deploy in Production

Once the model performs well, it can be deployed so that real users can benefit from it. Typically, deployment involves:

- Hosting the model on an **inference server**.
- Connecting it via an **API** to user-facing applications (e.g., mobile apps).
- The app sends inputs (x, such as an audio clip) to the inference server.
- The server runs the model to produce predictions ($\hat{y}$, such as a text transcript) and returns them to the app.

5. Maintain and Monitor the System

After deployment, continuous monitoring and maintenance are essential to ensure the system remains reliable and accurate.

- **Logging:** Record inputs, predictions, and system performance (while maintaining user privacy).
- **Monitoring:** Detect when the data distribution changes (e.g., new slang, accents, or objects not seen during training).
- **Model updates:** Periodically retrain or fine-tune the model to address data drift or new requirements.
- **Scaling:** Ensure the system can handle growing user demand efficiently.

6. MLOps: Machine Learning Operations

As ML systems grow in complexity and user base, deployment and maintenance become more sophisticated. This is where **MLOps** (Machine Learning Operations) plays a crucial role. MLOps refers to the combination of **software engineering practices** and **machine learning workflows** to automate and manage the ML lifecycle. It involves:

- Ensuring reliable and efficient predictions in real-world conditions.
- Automating scaling, logging, and system monitoring.

- Streamlining model retraining, version control, and updates.
- Managing infrastructure costs and ensuring robustness for millions of users.

7. Continuous Improvement Loop

Even after deployment, the project cycle does not end. You often:

- Gather new data from production usage.
- Perform new rounds of error analysis.
- Retrain and redeploy improved models.

This creates a continuous improvement loop that strengthens both the dataset and model performance over time.

Summary: End-to-End ML Lifecycle

1. **Scope Project:** Define what problem to solve.
2. **Collect Data:** Obtain and label relevant training data.
3. **Train Model:** Build, evaluate, and iterate on models.
4. **Deploy in Production:** Host and serve the model to users.
5. **Monitor and Maintain:** Use MLOps principles to ensure long-term reliability.

This full-cycle approach transforms a machine learning model from an academic experiment into a reliable, continuously improving real-world system.

Error Metrics for Skewed Datasets: Precision and Recall

In many real-world machine learning problems, especially those involving rare events, the dataset can be highly **skewed** — meaning the number of positive examples is much smaller than the number of negative examples. In such cases, using only the **classification accuracy** or **error rate** can be misleading.

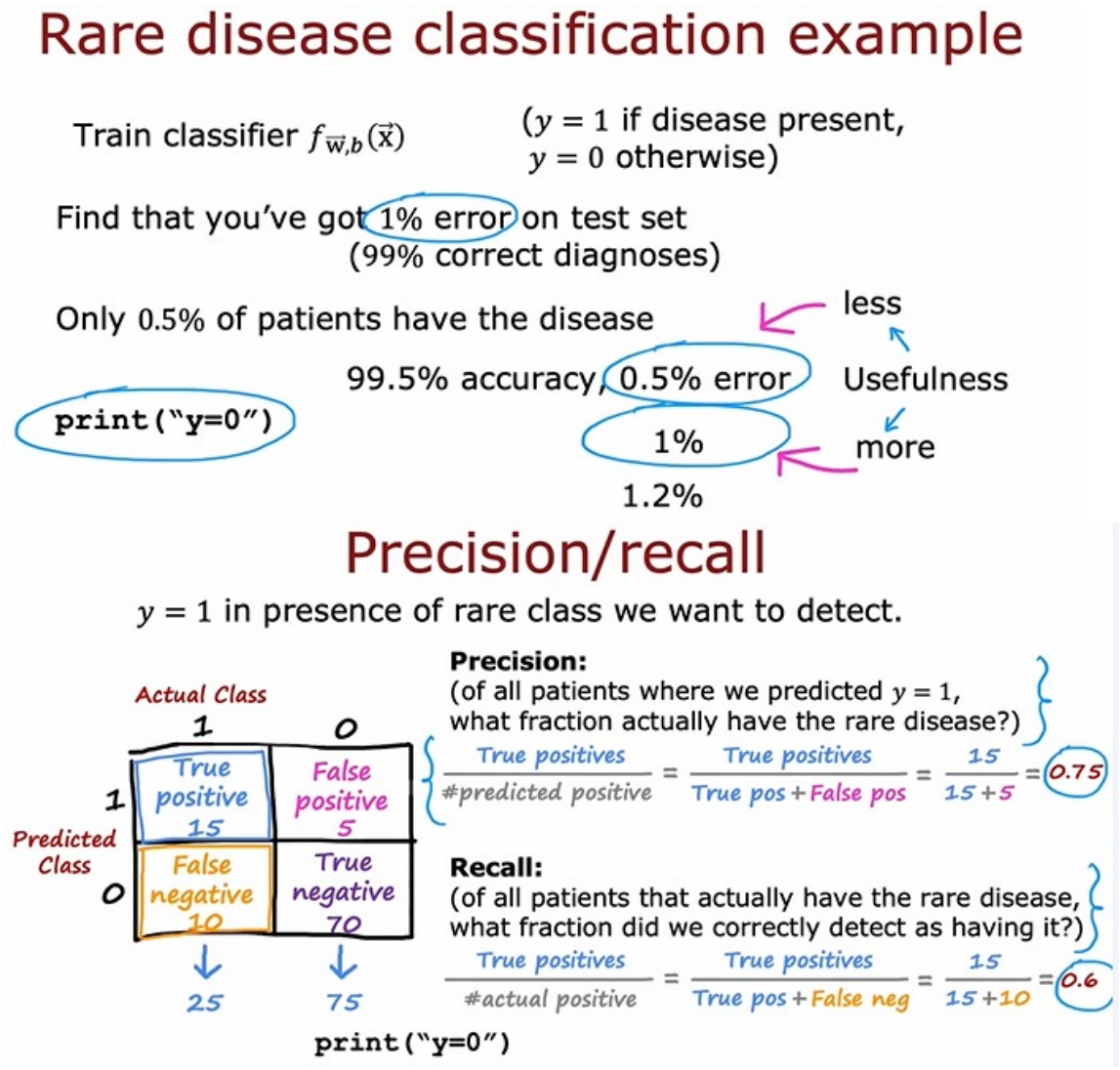

Figure 24: Example of rare disease classification and illustration of precision and recall metrics.

1. Problem with Accuracy in Skewed Datasets

Consider a binary classification task to detect a rare disease:

$$y = \begin{cases} 1 & \text{if disease is present,} \\ 0 & \text{otherwise.} \end{cases}$$

Suppose your learning algorithm achieves **1% error** on the test set (i.e., 99% accuracy). At first, this seems impressive. However, if only **0.5% of patients actually have the disease**, even a trivial algorithm that always predicts:

```
print("y = 0")
```

would achieve 99.5% accuracy (0.5% error). Hence, despite high accuracy, the algorithm is **useless for detecting rare cases**. Accuracy alone fails to reflect how well the model identifies rare but critical events.

2. The Need for Better Metrics: Precision and Recall

To better evaluate algorithms on skewed data, we use two key metrics:

- **Precision:** Of all the examples predicted as positive, what fraction are actually positive?
- **Recall:** Of all the actual positive examples, what fraction were correctly identified as positive?

3. Confusion Matrix

To define these metrics, we use the **confusion matrix:**

	Actual: $y = 1$	Actual: $y = 0$
Predicted: $y = 1$	True Positive (TP)	False Positive (FP)
Predicted: $y = 0$	False Negative (FN)	True Negative (TN)

Each cell represents a combination of prediction and actual class:

- **True Positive (TP):** Correctly predicted positive.
- **True Negative (TN):** Correctly predicted negative.
- **False Positive (FP):** Predicted positive but actually negative.
- **False Negative (FN):** Predicted negative but actually positive.

4. Example Calculation

Suppose we have the following confusion matrix from a test set of 100 examples:

	$y = 1$	$y = 0$
Predicted $y = 1$	15	5
Predicted $y = 0$	10	70

Here:

$$\text{TP} = 15, \quad \text{FP} = 5, \quad \text{FN} = 10, \quad \text{TN} = 70.$$

5. Precision and Recall Formulas

Precision:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} = \frac{TP}{TP + FP}$$

Substituting values:

$$\text{Precision} = \frac{15}{15 + 5} = 0.75$$

Thus, 75% of the time when the algorithm predicts disease, it is correct.

Recall:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} = \frac{TP}{TP + FN}$$

Substituting values:

$$\text{Recall} = \frac{15}{15 + 10} = 0.6$$

This means the model detects 60% of patients who actually have the disease.

6. Interpretation

Precision and recall give complementary views of performance:

- High **precision** means the model's positive predictions are usually correct.
- High **recall** means the model identifies most of the actual positives.

A model that always predicts $y = 0$ will have both **precision = 0** and **recall = 0**, showing that it's not useful at all — even if its accuracy is high.

7. Summary

For rare or imbalanced datasets:

- Accuracy can be misleading.

- Precision and recall provide more insight.
- Both metrics should be considered together to assess usefulness.

Example Summary:

$$\text{Precision} = 0.75, \quad \text{Recall} = 0.60$$

These values indicate that the classifier correctly identifies 60% of all patients with the disease while maintaining a 75% correctness rate for its positive predictions.

8. Motivation for the Next Step

Now that we have metrics for evaluating performance on skewed datasets, the next step is to understand how to **trade off between precision and recall** — adjusting the model's threshold to optimize for specific applications.

Trading Off Precision and Recall

Precision and Recall: Definitions

In binary classification, particularly in rare event prediction (e.g., disease diagnosis), two important evaluation metrics are **precision** and **recall**.

- **Precision:** The fraction of correctly predicted positives among all predicted positives.

$$\text{Precision} = \frac{\text{True Positives}}{\text{Total Predicted Positives}}$$

- **Recall:** The fraction of actual positives that were correctly predicted.

$$\text{Recall} = \frac{\text{True Positives}}{\text{Total Actual Positives}}$$

In an ideal scenario, a model should achieve both high precision and high recall. However, in practice, there is often a **trade-off** between the two.

Logistic Regression and Thresholding

For logistic regression, the model outputs probabilities:

$$0 < f_{w,b}(x) < 1$$

We typically predict:

$$\hat{y} = \begin{cases} 1, & \text{if } f_{w,b}(x) \geq 0.5 \\ 0, & \text{if } f_{w,b}(x) < 0.5 \end{cases}$$

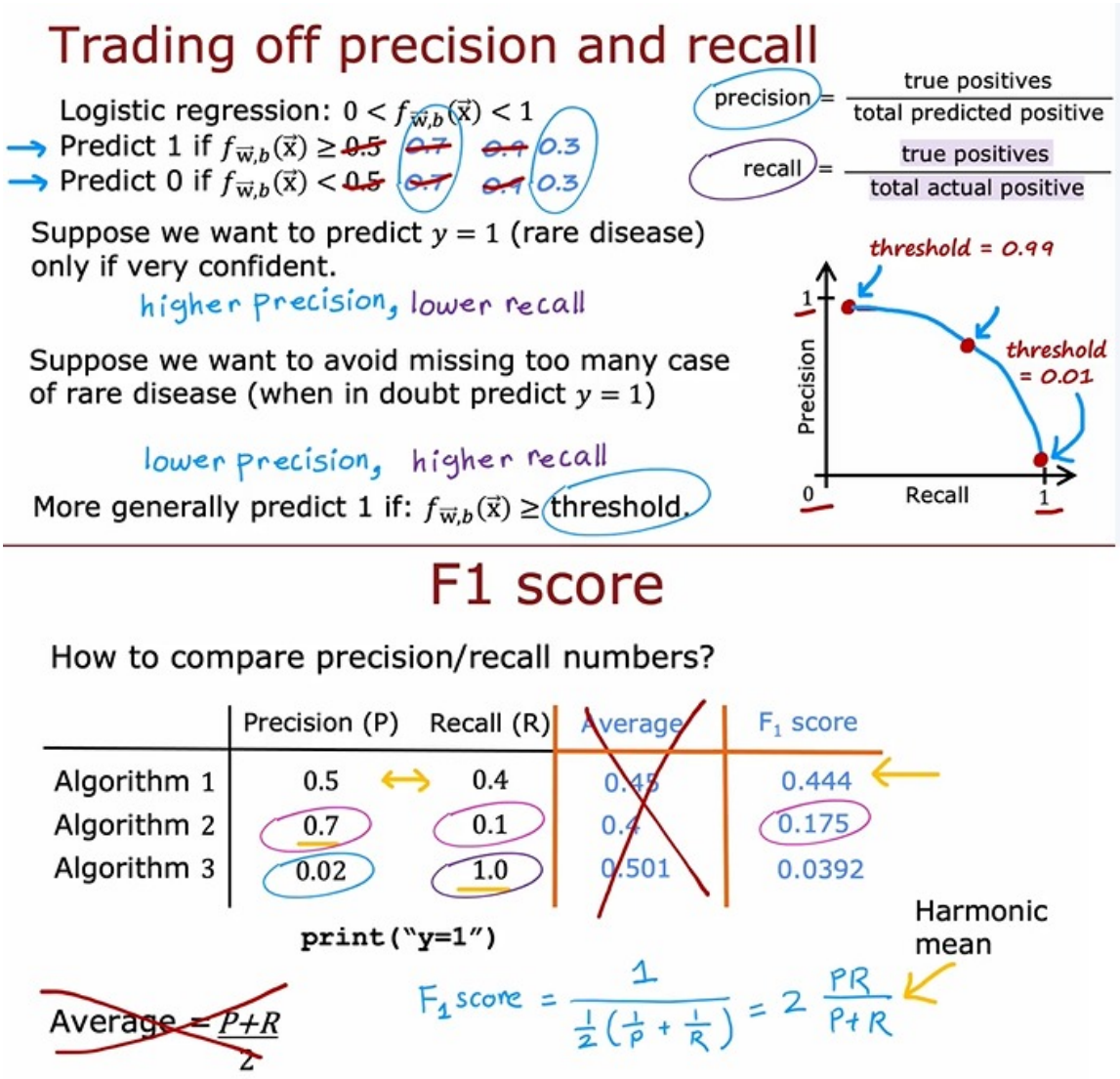

Figure 25: Trade-off between precision and recall, and calculation of F1 score.

The threshold value (commonly 0.5) determines how confident the model must be to predict a positive case.

Raising the Threshold

- Predict $y = 1$ only when $f_{w,b}(x) \geq 0.7$ or 0.9.
- The model becomes more conservative—only very confident predictions are labeled as positive.
- **Result:** Higher precision, lower recall.

Lowering the Threshold

- Predict $y = 1$ even when $f_{w,b}(x) \geq 0.3$ or lower.
- The model predicts more positives, even with lower confidence.
- **Result:** Higher recall, lower precision.

Key takeaway: Adjusting the threshold changes the balance between precision and recall. This trade-off can be visualized on a **Precision–Recall curve**, where different threshold values produce different points along the curve.

Interpreting the Trade-off

- A very high threshold (e.g., 0.99) $\Rightarrow$ high precision, low recall.
- A very low threshold (e.g., 0.01) $\Rightarrow$ low precision, high recall.
- The shape of the Precision–Recall curve helps identify the right balance point.

Choosing the threshold is usually a design decision based on the application's needs:

- In medical diagnosis, high recall may be preferred (to catch all possible cases).
- In spam detection, high precision may be preferred (to avoid false alarms).

The F1 Score: Balancing Precision and Recall

When comparing algorithms, evaluating both precision and recall separately can be difficult. To combine them into a single metric, we use the **F1 score**.

$$F_1 = 2 \cdot \frac{P \times R}{P + R}$$

where P is precision and R is recall. This formula represents the **harmonic mean** of precision and recall.

Algorithm	Precision (P)	Recall (R)	F1 Score
1	0.5	0.4	0.444
2	0.7	0.1	0.175
3	0.02	1.0	0.039

Table 2: Comparison of three algorithms using Precision, Recall, and F1 score.

Why Harmonic Mean?

The harmonic mean emphasizes the smaller value among P and R, ensuring that a model with very low precision or recall gets a low overall F1 score. This avoids rewarding models that perform well on one metric but poorly on the other.

Example Comparison

- Algorithm 1 achieves a balanced trade-off (moderate precision and recall).
- Algorithm 2 has high precision but extremely low recall.
- Algorithm 3 has high recall but almost zero precision.

The F1 score automatically identifies Algorithm 1 as the best among the three since it performs reasonably well on both metrics.

Summary

- Increasing the threshold $\Rightarrow$ higher precision, lower recall.
- Decreasing the threshold $\Rightarrow$ higher recall, lower precision.
- Precision–Recall trade-off helps select an optimal operating point.
- F1 score combines both metrics into a single interpretable measure.

$$\boxed{F_1 = 2 \cdot \frac{P \times R}{P + R}}$$

The F1 score provides a robust way to compare classifiers and select models that maintain a reasonable balance between precision and recall—especially when both false positives and false negatives carry significant costs.

Introduction to Decision Trees

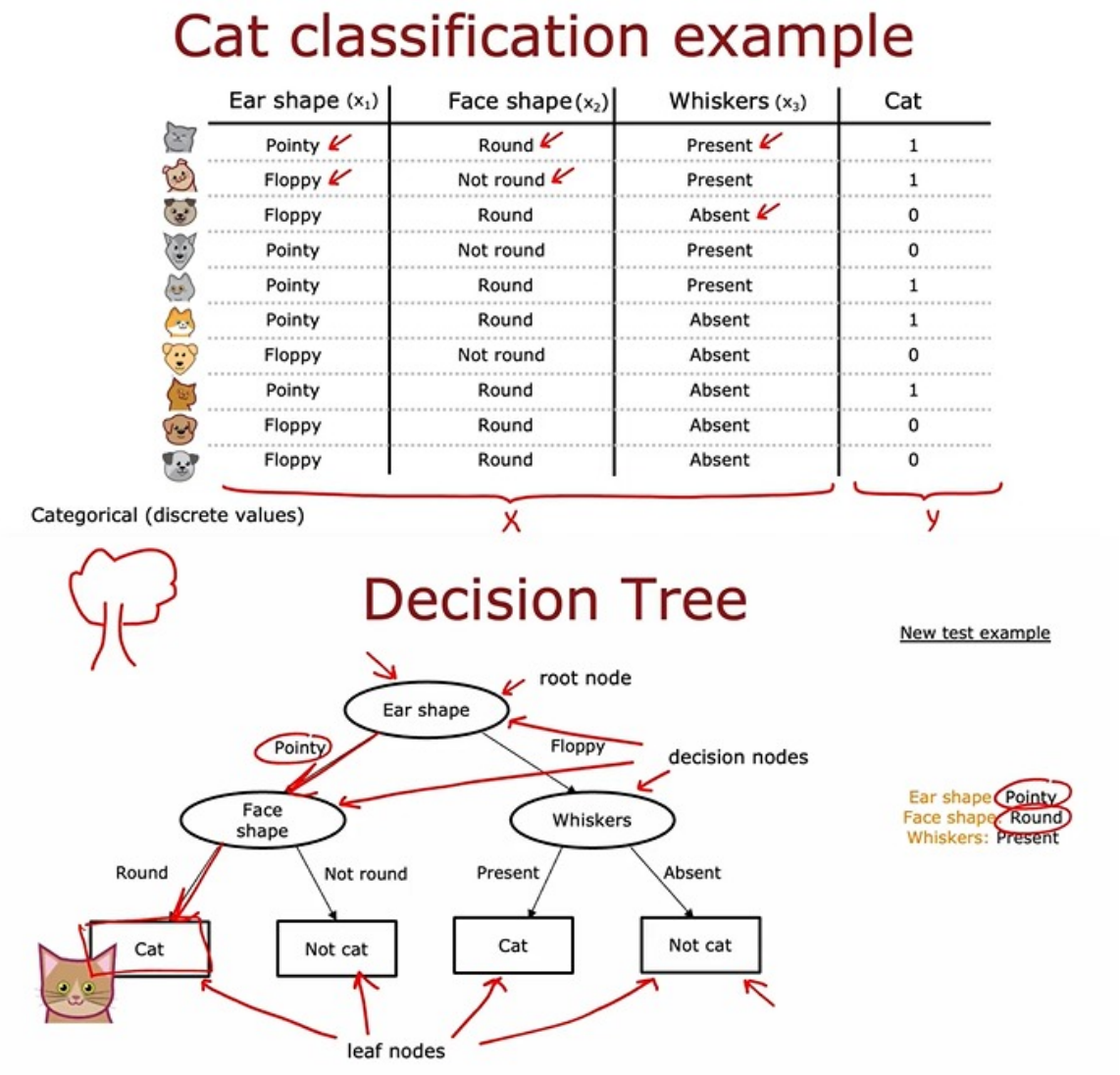

Figure 26: Cat classification example and corresponding decision tree.

Overview

Welcome to the final week of this course on **Advanced Learning Algorithms**. One of the most powerful and widely used algorithms in machine learning is the **Decision Tree**. Decision trees are used in numerous applications — from healthcare diagnostics to financial prediction — and have also helped many practitioners win machine learning competitions.

Despite their success, decision trees often receive less attention in academia compared to other algorithms. However, they are an essential part of any machine learning toolbox, and understanding how they work provides intuition for more advanced ensemble methods such as **Random Forests** and **Gradient Boosted Trees**.

Cat Classification Example

To illustrate how decision trees operate, consider the following example. You are running a **cat adoption center**, and you want to train a model to automatically identify whether an animal is a cat or not based on a few observed features.

- **Feature 1:** Ear shape (x_1) – either *Pointy* or *Floppy*.
- **Feature 2:** Face shape (x_2) – either *Round* or *Not round*.
- **Feature 3:** Whiskers (x_3) – either *Present* or *Absent*.

The dataset consists of 10 training examples. Each example contains these three features (X) and a label (y) indicating whether the animal is a cat (1) or not a cat (0). Half of the examples are cats, and half are dogs.

$$X = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}, \qquad y = \{0, 1\}$$

This setup defines a **binary classification task** where:

$$y = \begin{cases} 1, & \text{if the animal is a cat} \\ 0, & \text{otherwise} \end{cases}$$

Each of the features is **categorical**, taking on only a few discrete values. Later, we will discuss how decision trees handle continuous-valued features as well.

Understanding the Decision Tree Structure

After training a decision tree learning algorithm on this dataset, we obtain a model like the one shown in Figure 26. While this structure may not resemble a biological tree, in computer science, such a hierarchical structure is also referred to as a **tree**.

- Each **oval** in the diagram represents a **node**.
- The **topmost node** is called the **root node**.
- The **ovals in the middle** are **decision nodes**.
- The **rectangular boxes at the bottom** are **leaf nodes**.

The **root node** corresponds to the first feature used to split the data — in this example, *Ear shape*. At each decision node, the tree checks the value of a specific feature and follows a corresponding branch.

Example of Classification Step-by-Step

Let's classify a new test example with the following attributes:

Ear shape: Pointy, Face shape: Round, Whiskers: Present.

The decision-making process proceeds as follows:

1. Start at the **root node**: Check the *Ear shape*.
2. Since it is *Pointy*, follow the left branch.
3. Move to the next decision node: Check the *Face shape*.
4. The face is *Round*, so follow the left arrow again.
5. Arrive at the **leaf node**, which predicts: **Cat (1)**.

Thus, the decision tree predicts that this animal is a cat.

Terminology Recap

- **Root node:** The topmost node of the tree; the starting point of classification.
- **Decision nodes:** Intermediate nodes that test a specific feature and branch accordingly.
- **Leaf nodes:** Terminal nodes that output a prediction (e.g., "Cat" or "Not cat").

Note: In computer science, trees are typically drawn upside-down — the root is at the top, and the leaves are at the bottom. You can think of this as an "indoor hanging plant," where the roots are at the top and the leaves hang down.

Alternative Decision Trees

There are often multiple ways to construct a decision tree for the same dataset. For example:

- One tree might start with *Ear shape*, then branch by *Face shape*.
- Another might start with *Whiskers*, then check *Ear shape*.

Each configuration results in a different tree structure. Some trees will perform better on the training data, while others generalize better to unseen data (validation or test sets).

Goal of the Decision Tree Learning Algorithm

The job of the learning algorithm is to:

- Explore possible tree structures.
- Evaluate their performance on the training data.
- Choose the tree that achieves the best balance between accuracy and generalization.

This process ensures that the model not only fits the training examples but also performs well on new, unseen examples — avoiding both underfitting and overfitting.

Summary

- Decision trees split data step-by-step using feature-based rules.
- Each path from root to leaf corresponds to a sequence of decisions leading to a prediction.
- The structure is intuitive, interpretable, and useful for both categorical and continuous features.
- The learning algorithm aims to build the optimal tree that balances bias and variance.

In the next section, we will explore **how a decision tree learning algorithm selects which feature to split on** — using measures such as **Information Gain** and **Gini Impurity**.

Decision Tree Learning

Decision Tree Learning is a supervised learning algorithm used for both classification and regression tasks. The process of constructing a decision tree involves several key steps and decisions aimed at maximizing the purity of each node while minimizing overfitting.

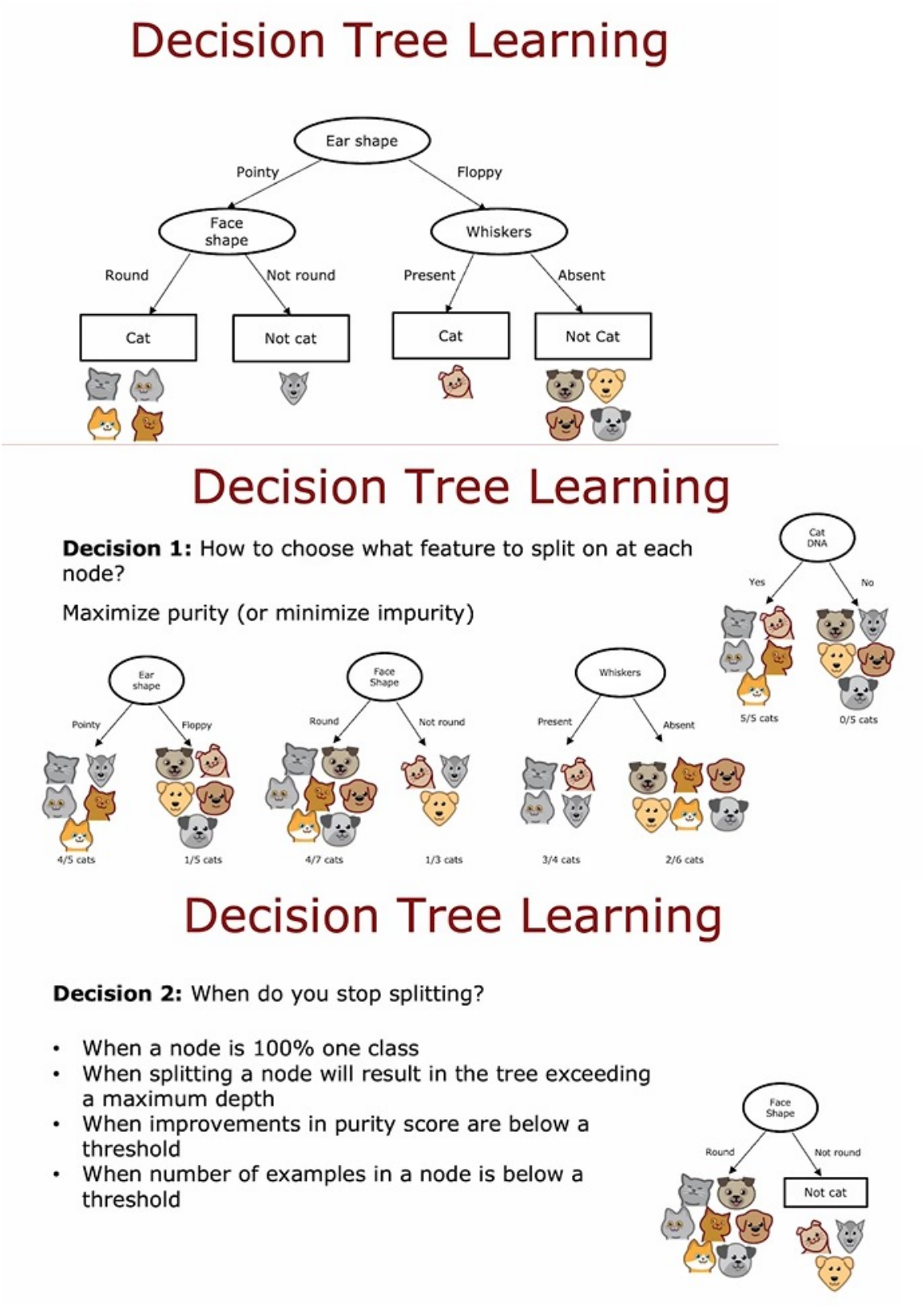

Figure 27: Illustration of Decision Tree Learning Process

Step 1: Choosing the Root Node Feature

Given a training set (e.g., 10 examples of cats and dogs), the first step is to decide which feature to use at the root node — the topmost node in the tree.

For instance, suppose we choose **Ear Shape** as the root feature. The dataset is then divided into two subsets:

- Examples with **Pointy Ears** → Left branch
- Examples with **Floppy Ears** → Right branch

This split partitions the data according to the chosen feature values.

Step 2: Building Sub-Branches

Next, we recursively build sub-branches for each subset.

Example: Left Branch (Pointy Ears)

- The next chosen feature is **Face Shape**.
- The subset is split again:
  - **Round Face:** 4 out of 5 examples are cats → create a leaf node labeled **Cat**.
  - **Not Round Face:** 1 example, not cat → create a leaf node labeled **Not Cat**.

Example: Right Branch (Floppy Ears)

- Choose the next feature, e.g., **Whiskers**.
- Split based on whisker presence:
  - **Whiskers Present:** All examples are cats → leaf node labeled **Cat**.
  - **Whiskers Absent:** All examples are not cats → leaf node labeled **Not Cat**.

Decision 1: Choosing What Feature to Split On

At each node, we must decide which feature to split on to best separate the data.

Goal: Maximize purity (or equivalently, minimize impurity).

If we had a feature like **Cat DNA**, it would perfectly separate all cats and non-cats (pure subsets). Since such ideal features are rare, we use features like **Ear Shape**, **Face Shape**, or **Whiskers** to split in a way that yields the highest possible purity.

Example of Purity:

$$\text{Ear Shape: } \frac{4}{5} \text{ cats on left, } \frac{1}{5} \text{ on right}$$

$$\text{Face Shape: } \frac{4}{7} \text{ cats on left, } \frac{1}{3} \text{ on right}$$

$$\text{Whiskers: } \frac{3}{4} \text{ cats on left, } \frac{2}{6} \text{ on right}$$

The algorithm evaluates each feature's resulting purity and selects the one that maximizes it.

Decision 2: When to Stop Splitting

A node becomes a leaf (i.e., no further splitting) under one or more of the following conditions:

- The node contains examples from **only one class** (100% pure).
- Further splitting would cause the tree to exceed a **maximum depth**.
- **Improvements in purity** after a split fall below a defined threshold.
- The **number of examples** in a node is below a minimum threshold.

Depth Definition:

- Root node → Depth 0
- Child nodes → Depth 1
- Grandchild nodes → Depth 2, and so on.

Restricting depth helps prevent the tree from growing too large and reduces the risk of overfitting.

Summary of Key Insights

- Decision Trees recursively split the training set using features that maximize class purity.
- Each leaf node represents a final decision or prediction.
- Controlling tree depth and minimum samples per node ensures generalization.
- Despite its complexity, Decision Tree Learning is an effective, interpretable, and widely used algorithm.

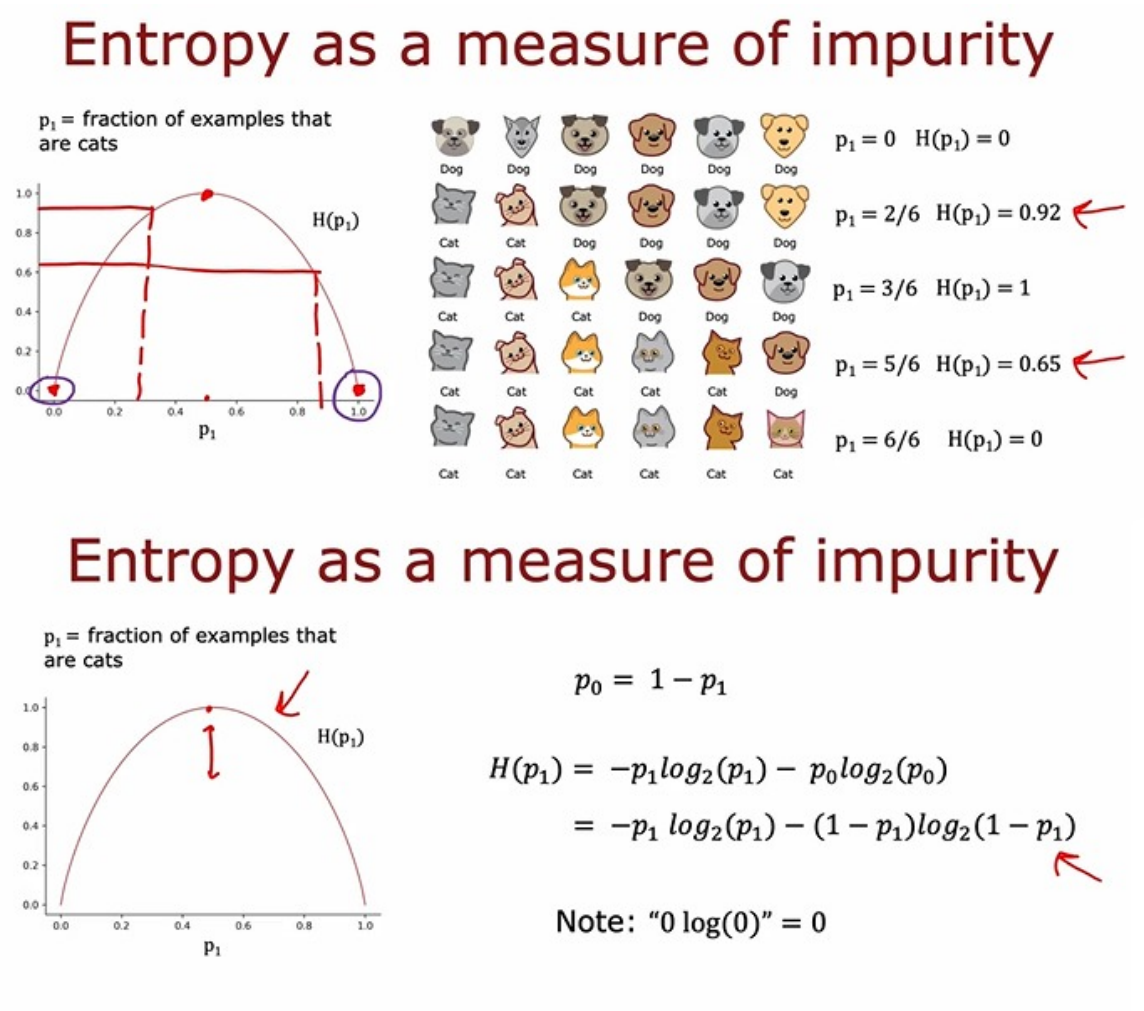

Figure 28: Entropy as a Measure of Impurity

Entropy as a Measure of Impurity

When constructing a decision tree, we need a way to measure how **pure** or **impure** a node is after a split.

If a node contains examples of only one class (for example, all cats), then the node is **pure**. If the examples are evenly split between two classes (for example, half cats and half dogs), then the node is **impure**.

Defining Entropy

Entropy is a quantitative measure of impurity. Suppose in a node, the fraction of examples that belong to class 1 (e.g., "cat") is p_1, and those belonging to class 0 (e.g., "not cat") is $p_0 = 1 - p_1$.

The **entropy** of that node is defined as:

$$H(p_1) = -p_1 \log_2(p_1) - p_0 \log_2(p_0)$$

By convention, if $p_1 = 0$ or $p_0 = 0$, then the corresponding term $0 \log_2(0)$ is taken as 0.

Understanding Entropy Values

- When $p_1 = 0$ (all examples are not cats) or $p_1 = 1$ (all examples are cats), the entropy is:

  $$H(p_1) = 0$$

  indicating perfect purity.

- When $p_1 = 0.5$ (half cats, half dogs), the entropy reaches its maximum:

  $$H(0.5) = 1$$

  indicating maximum impurity.

Thus, entropy ranges from 0 (completely pure) to 1 (completely impure).

Example Calculations

The graph in the figure above (*measuring_impurity.jpg*) shows that entropy increases as the class mix becomes more balanced, peaking at $p_1 = 0.5$, and decreases again as one class dominates.

Table 3: Entropy Examples for Different Class Distributions

Fraction of Cats (p_1)	**Description**	**Entropy ($H(p_1)$)**
0.0	All Dogs (Pure)	0.00
0.2	Mostly Dogs	0.72
0.5	Equal Mix (Max Impurity)	1.00
0.8	Mostly Cats	0.72
1.0	All Cats (Pure)	0.00

Intuitive Meaning

Entropy measures the uncertainty of a system:

- Low entropy means high certainty (node is mostly one class).
- High entropy means high uncertainty (node contains mixed classes).

Therefore, in decision tree learning, we aim to split nodes such that entropy decreases — that is, each child node becomes purer than its parent.

Summary

- Entropy quantifies impurity in a node.
- $H = 0 \rightarrow$ perfectly pure node.
- $H = 1 \rightarrow$ completely impure node (balanced mix).
- Decision trees prefer splits that minimize entropy (i.e., increase node purity).

Choosing a Split and Information Gain

When building a decision tree, the key question at each node is: *Which feature should we split on?* The goal is to choose the feature that results in the **largest reduction in impurity** (or equivalently, the largest increase in purity). This reduction in entropy is known as the **Information Gain**.

Motivation

Entropy measures impurity in a dataset. When a dataset is split based on some feature, each subset will generally have a different entropy. We aim to find a feature split that produces subsets with the *lowest average impurity*.

Thus, we can view the **Information Gain (IG)** as the amount by which entropy decreases after the split.

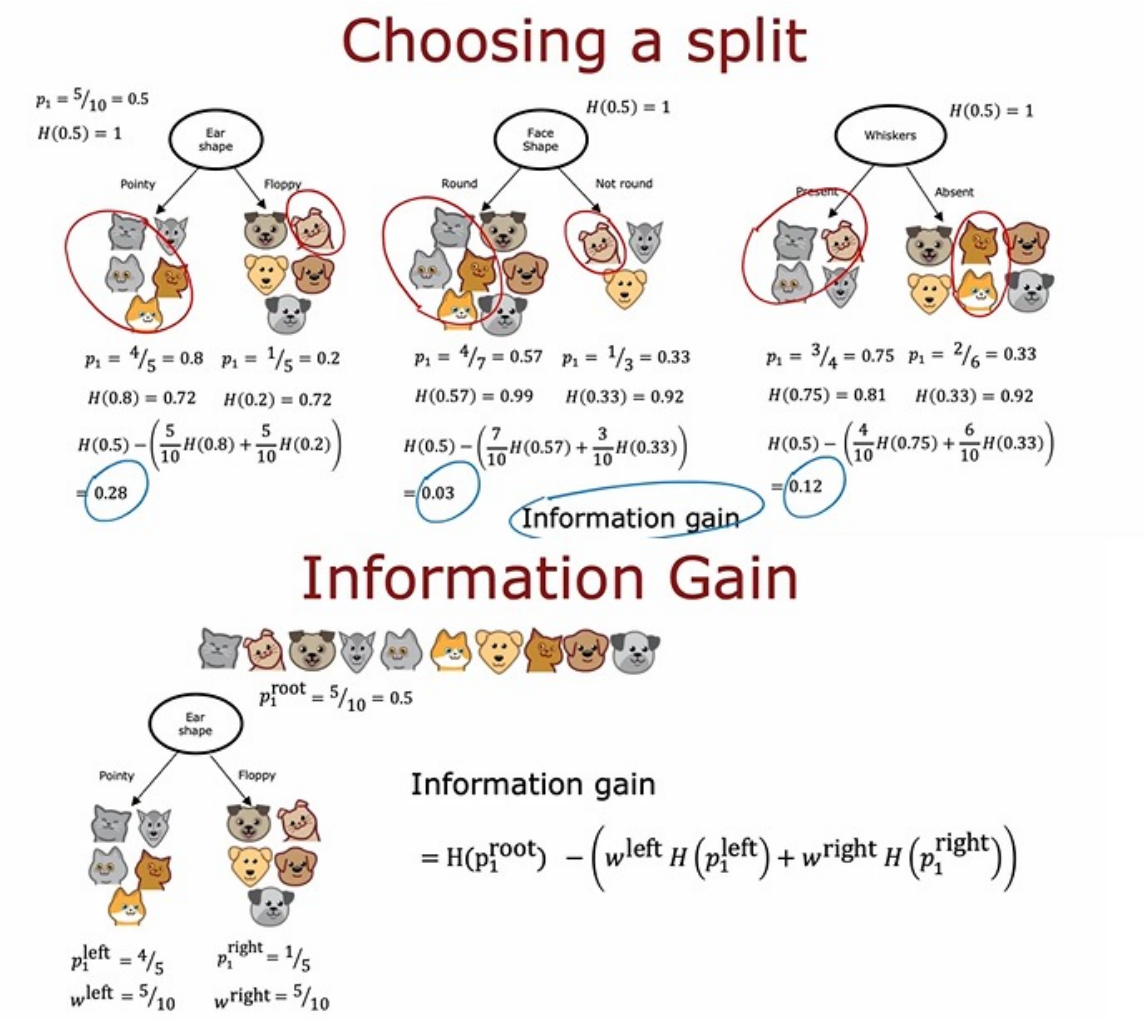

Figure 29: Choosing a Split and Computing Information Gain in Decision Trees

Example: Choosing a Feature to Split On

Suppose we have 10 examples, 5 cats and 5 dogs. At the root node:

$$p_1^{root} = \frac{5}{10} = 0.5 \quad \Rightarrow \quad H(p_1^{root}) = 1.$$

We consider three possible features to split on:

- **Ear Shape (Pointy / Floppy)**
- **Face Shape (Round / Not Round)**
- **Whiskers (Present / Absent)**

For each split, we compute the fraction of cats in the left and right subsets and their entropies.

Example 1: Ear Shape

Left branch: $p_1^{left} = \frac{4}{5} = 0.8$, $H(0.8) = 0.72$ Right branch: $p_1^{right} = \frac{1}{5} = 0.2$, $H(0.2) = 0.72$

Weighted average entropy:

$$\frac{5}{10}H(0.8) + \frac{5}{10}H(0.2) = 0.72.$$

Information Gain:

$$IG = H(0.5) - 0.72 = 1 - 0.72 = 0.28.$$

Example 2: Face Shape

Left branch: $p_1^{left} = \frac{4}{7} = 0.57$, $H(0.57) = 0.99$ Right branch: $p_1^{right} = \frac{1}{3} = 0.33$, $H(0.33) = 0.92$

Weighted average entropy:

$$\frac{7}{10}H(0.57) + \frac{3}{10}H(0.33) = 0.97.$$

Information Gain:

$$IG = H(0.5) - 0.97 = 1 - 0.97 = 0.03.$$

Example 3: Whiskers

Left branch: $p_1^{left} = \frac{3}{4} = 0.75$, $H(0.75) = 0.81$ Right branch: $p_1^{right} = \frac{2}{6} = 0.33$, $H(0.33) = 0.92$

Weighted average entropy:

$$\frac{4}{10}H(0.75) + \frac{6}{10}H(0.33) = 0.88.$$

Information Gain:

$$IG = H(0.5) - 0.88 = 1 - 0.88 = 0.12.$$

Selecting the Best Feature

Table 4: Comparison of Information Gain for Different Features

Feature	**Weighted Entropy**	**Information Gain**
Ear Shape	0.72	0.28
Face Shape	0.97	0.03
Whiskers	0.88	0.12

From the table above, **Ear Shape** yields the highest information gain (0.28), so it is chosen as the feature to split on at the root node.

General Formula for Information Gain

Let:

$$p_1^{root} = \text{fraction of positive examples at root,}$$

$$p_1^{left}, p_1^{right} = \text{fractions of positive examples in left and right subsets,}$$

$$w^{left}, w^{right} = \text{fractions of total examples that go left or right.}$$

Then the **Information Gain** from splitting on a feature is:

$$IG = H(p_1^{root}) - \left[w^{left}H(p_1^{left}) + w^{right}H(p_1^{right})\right].$$

Interpretation

- A high Information Gain means the split significantly reduces impurity.
- A low Information Gain means the split doesn't improve purity much.
- Splitting stops when Information Gain becomes too small (to prevent overfitting).

In summary, **Information Gain** quantifies how much uncertainty (entropy) is reduced by splitting the data on a particular feature. The feature that produces the largest Information Gain is selected at each node.

Continuous Features in Decision Trees

Decision trees can handle not only discrete features such as ear shape, face shape, and whiskers, but also continuous features like weight. Continuous features take numeric values, and the algorithm must determine the best threshold to split the data.

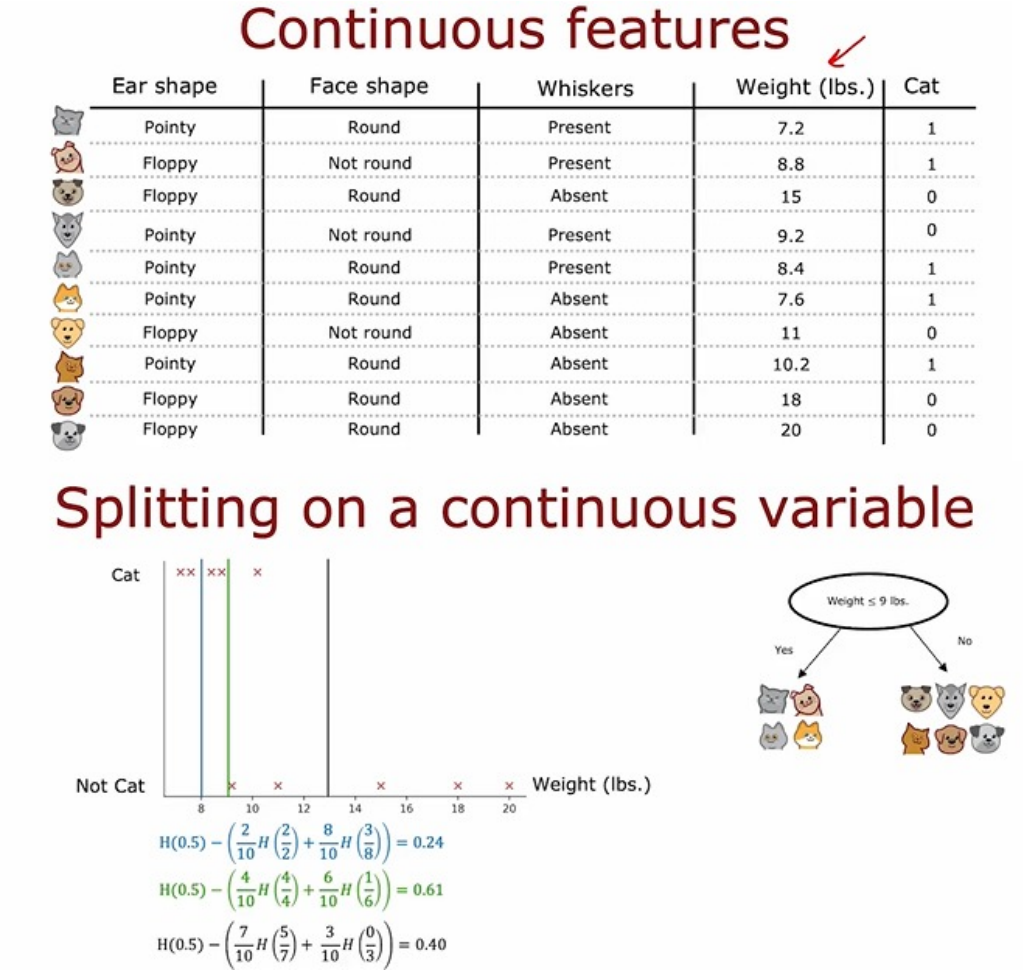

Figure 30: Example of splitting a continuous feature (weight) for classification.

Consider a modified cat adoption dataset where each animal also has a weight feature in pounds. Because cats tend to be lighter than dogs (with some overlap), weight can help classify whether an animal is a cat or not.

To use continuous features, the decision tree tries splits of the form:

$$\text{Weight} \leq t$$

for several possible threshold values t. For each threshold, the training set is divided into:

- Left subset: feature value less than or equal to t
- Right subset: feature value greater than t

For each candidate threshold, the algorithm computes the information gain:

$$IG = H(\text{parent}) - \left(\frac{n_L}{n} H(\text{left}) + \frac{n_R}{n} H(\text{right})\right)$$

Example: Trying different split thresholds

Threshold: 8 pounds Left subset: two cats Right subset: three cats and five dogs

$$IG = 0.24$$

Threshold: 9 pounds Left subset: four cats Right subset: one cat and five dogs

$$IG = 0.61$$

Threshold: 13 pounds Information gain value:

$$IG = 0.40$$

The threshold at nine pounds gives the highest information gain.

General Procedure for Continuous Features

1. Sort all training examples by the continuous feature.
2. For n examples, evaluate the $n - 1$ midpoints between consecutive values.
3. Compute information gain for each candidate threshold.
4. Select the threshold that yields the highest information gain.

If this best threshold gives higher information gain than any other feature, the decision tree splits on this continuous feature.

In this example, the selected split is:

$$\text{Weight} \leq 9 \text{ pounds}$$

and the tree continues recursively from the resulting subsets.

Decision Tree Learning

Decision tree learning uses information gain repeatedly to decide which feature to split on at each node. The overall process for building a decision tree is a recursive procedure that starts at the root node and continues until a chosen stopping criterion is met.

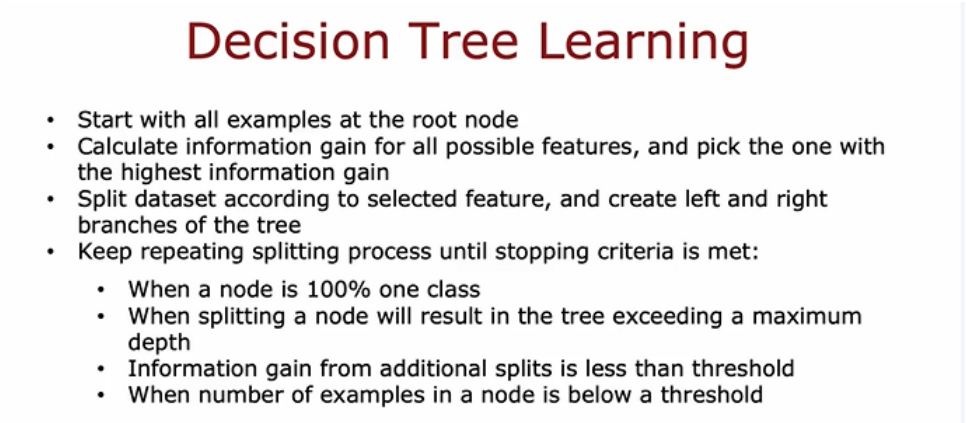

Figure 31: Overview of the decision tree learning process.

Overall Learning Procedure

- Start with all training examples at the root node.
- Compute the information gain for all available features.
- Choose the feature that yields the highest information gain.
- Split the dataset into left and right subsets according to that feature.
- Create left and right branches, sending each example to the correct branch.
- Repeat the splitting process recursively on each branch.

This splitting continues until one or more stopping criteria are met.

Stopping Criteria

A node will stop splitting when any of the following conditions is met:

- The node contains examples that all belong to a single class (entropy equals zero).
- Further splitting would cause the tree to exceed the maximum depth allowed.
- The information gain from any additional split is smaller than a chosen threshold.
- The number of training examples in the node falls below a threshold.

Example of the Splitting Process

At the root node, the algorithm evaluates all features (for example, ear shape, face shape, whiskers). Suppose *ear shape* provides the highest information gain. The dataset is then split into:

- A left subtree for examples with pointy ears.
- A right subtree for examples with floppy ears.

Each subtree now becomes a smaller decision tree training problem. For the left subtree, the algorithm again computes information gain using only the five examples that belong to this branch. Ear shape may no longer be useful (if all five examples share the same ear shape), so the algorithm evaluates remaining features such as face shape and whiskers. If face shape provides the highest information gain, the left branch is split again.

Leaf nodes are created when the stopping criteria are met. For example:

- Left child node: all cats → leaf predicts **cat**.
- Right child node: all dogs → leaf predicts **not cat**.

A similar recursive process is applied to the right subtree. The algorithm again computes information gain for the remaining examples and selects the best feature to split on. If whiskers gives the highest information gain, the data is split based on presence or absence of whiskers, and leaf nodes are created as appropriate.

Recursive Nature of Decision Trees

Building a decision tree is a classic example of a recursive algorithm. To build the entire tree:

- At each node, build a smaller decision tree on the subset of examples that reach that node.
- The complete tree is formed by combining these smaller subtrees.

Even if the concept of recursion feels unfamiliar, you can still successfully use libraries to train decision trees. Recursion only matters if implementing the tree-building algorithm from scratch.

Choosing Maximum Depth and Other Parameters

The maximum depth controls how large and complex the tree can grow. A deeper tree can fit more complex patterns but may overfit the training data. Common strategies for choosing this parameter include:

- Using cross-validation to evaluate different maximum depths.
- Using thresholds for minimum information gain.
- Requiring a minimum number of training examples in a node before splitting.

Many open-source libraries automatically choose good default values for these hyperparameters.

Making Predictions

To make a prediction on a new example, start at the root node and follow the decisions at each split until a leaf node is reached. The leaf node's label becomes the prediction.

This completes the core description of the decision tree learning algorithm. The next topic explores how to handle features that take on more than two discrete values.

One-Hot Encoding for Categorical Features

In earlier examples, each feature could take only one of two possible values. For instance:

- Ear shape: pointy or floppy
- Face shape: round or not round
- Whiskers: present or absent

However, some features may take on more than two discrete values. Consider the ear shape feature, which can now take three values: *pointy*, *floppy*, or *oval*. This feature is still categorical, but instead of two possible values, it now has three. If a decision tree splits on this feature directly, it would create three branches.

A more flexible approach is to use **one-hot encoding**. Rather than using a single feature with three values, we replace it with three binary features:

- Does the animal have pointy ears?
- Does the animal have floppy ears?
- Does the animal have oval ears?

For example, if an animal has pointy ears:

$$[1, 0, 0]$$

If it has oval ears:

$$[0, 0, 1]$$

Thus, instead of one 3-valued feature, we now have three binary features. If a categorical feature has k possible values, one-hot encoding replaces it with k binary features, each taking values in $\{0, 1\}$. Importantly, in each row, **exactly one** of these k features is equal to 1. This is why the method is called *one-hot encoding*.

With this encoding, all features become binary, allowing the decision tree algorithm to operate without modification.

Although this video focuses on decision trees, one-hot encoding is widely used in other machine learning models as well. Neural networks, logistic regression, and linear

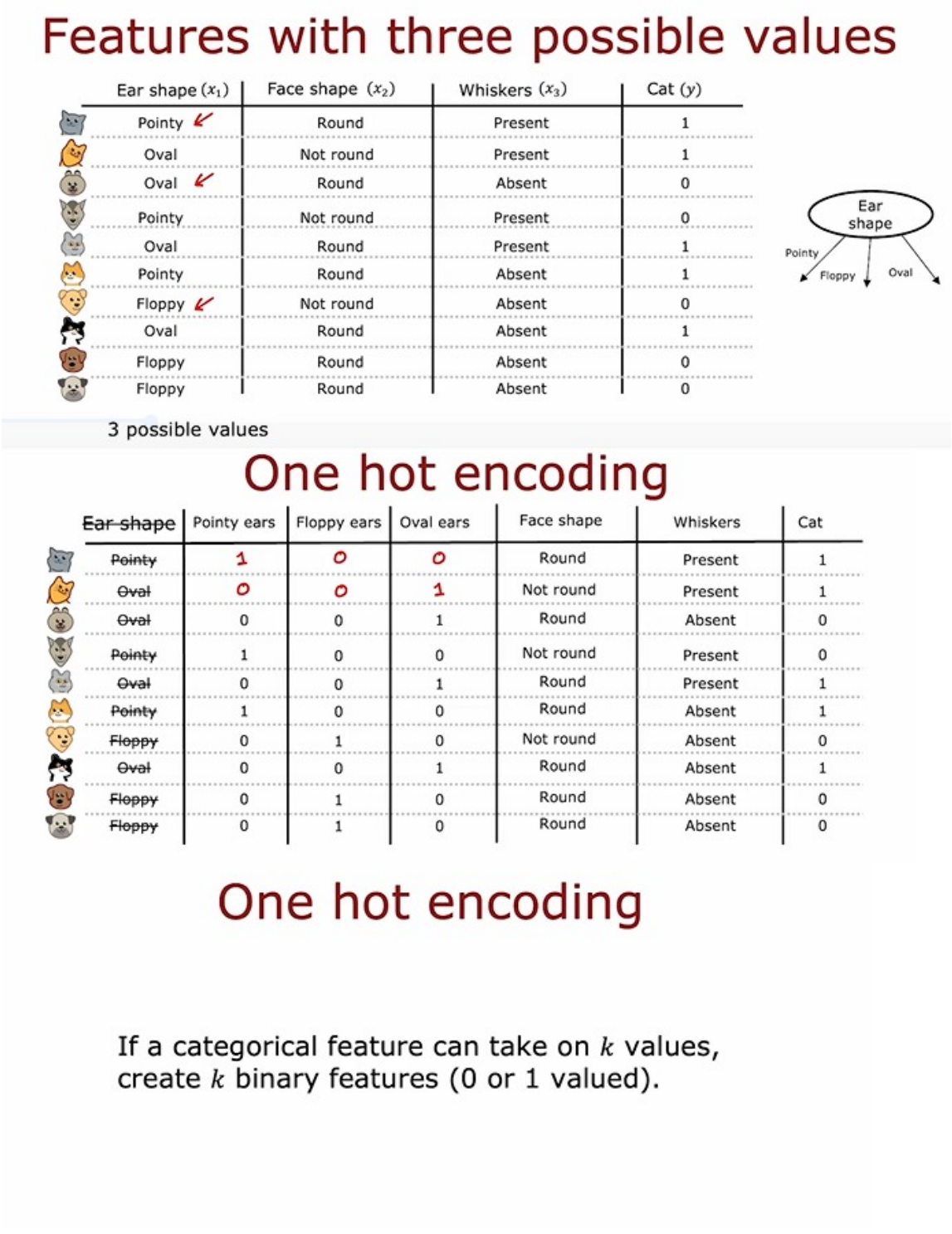

Figure 32: Example of converting a 3-valued categorical feature (ear shape) into three binary features using one-hot encoding.

regression all expect numerical input, and one-hot encoding provides a systematic way to convert categorical features into numerical inputs.

After one-hot encoding:

- Ear shape (3 values) $\rightarrow$ 3 binary features
- Face shape (2 values) $\rightarrow$ 1 binary feature (round = 1, not round = 0)
- Whiskers (2 values) $\rightarrow$ 1 binary feature (present = 1, absent = 0)

This results in a total of 5 numerical input features that can be used by decision trees, neural networks, or logistic regression.

In the next section, we will explore how decision trees handle **continuous-valued features** that can take any numerical value.

Tree Ensembles and Model Robustness

A single decision tree can be highly sensitive to small changes in the training data. Even modifying just one training example may lead to a different choice of root feature, producing an entirely different tree. This lack of robustness is one of the weaknesses of using only a single decision tree.

For example, in our original dataset, the highest information gain at the root node came from splitting on the *ear shape* feature. However, if we change just one training example—such as altering a single animal's ear shape, whisker presence, or face shape—the feature with the highest information gain may switch to the *whiskers* feature. As a result, the structure of the left and right subtrees becomes completely different, and a new overall tree is produced. Such dramatic changes caused by a single modified example show that a single decision tree can be unstable.

To address this issue, we can build not just one decision tree but a large number of different decision trees. This collection of trees is called a **tree ensemble**. Instead of relying on the prediction of a single tree, we allow multiple trees to vote. By averaging or taking a majority vote over their predictions, we obtain a result that is more stable and often more accurate.

Suppose we construct an ensemble of three trees, each representing a plausible way to classify whether an example is a cat or not. For a new test example with *pointy ears*, a *not round* face shape, and *whiskers present*, we run all three trees on this example:

- Tree 1 predicts: Cat
- Tree 2 predicts: Not cat
- Tree 3 predicts: Cat

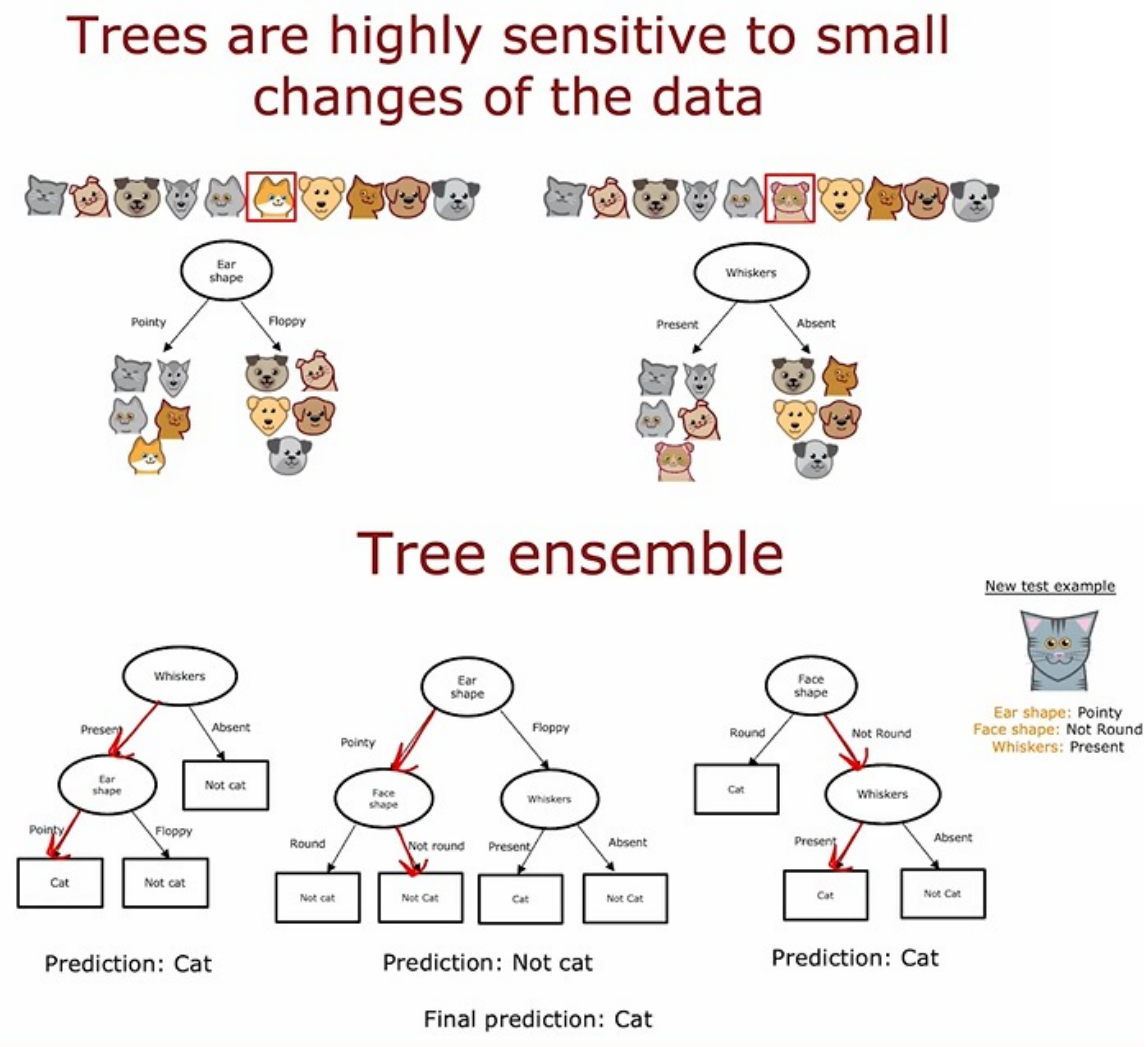

Figure 33: Decision trees are sensitive to small changes in the data. A tree ensemble combines multiple trees and predicts via majority vote, resulting in a more robust final prediction.

The final prediction of the ensemble is determined by majority vote. In this case, two out of the three trees predict "cat", so the ensemble predicts that the example is a cat. Because no single tree determines the outcome, the method becomes far more robust to noise and variations in the training data.

This voting-based approach is the key reason why tree ensembles tend to outperform individual trees. Each tree contributes one vote, reducing the impact of any tree that makes an error or is overly sensitive to the data.

In the next video, we will introduce a statistical technique called *sampling with replacement*. This will be an important step toward constructing the ensemble of slightly different decision trees used in methods such as bagging and random forests.

Bagged Decision Trees and the Random Forest Algorithm

We now have a method to generate multiple variations of a training set using *sampling with replacement*. This idea lets us build an ensemble of decision trees rather than relying on a single tree. In this section, we describe how to construct such an ensemble and how an additional modification leads to the **random forest algorithm**, one of the most powerful and widely used tree ensemble methods.

Generating a Tree Sample (Bagging)

Suppose we are given a training set of size M. To build an ensemble of B trees, we repeat the following procedure for $b = 1$ to B:

1. Use sampling with replacement to create a new training set of size M. This new dataset will look similar to the original training set but will contain repeated examples and will omit others.

2. Train a decision tree on the new dataset.

The result of this process is called a **bagged decision tree** ensemble. The name "bagging" refers to placing the training examples into a "virtual bag" and randomly sampling from it with replacement.

After generating B such datasets and training B trees, we obtain a collection of slightly different trees. Typical values for B range from 64 to 128, and often around 100. Increasing B generally does not hurt performance, but after a certain point the improvement becomes negligible while computation time increases significantly.

When making predictions, each tree votes, and the majority vote determines the final prediction. Because the ensemble averages over many small variations of the training set, the resulting model is much more stable and robust than a single decision tree.

Randomizing the Feature Choice

While bagging itself improves robustness, decision trees often still select the same feature at the root and near the root of each tree. This causes many of the trees in the ensemble to resemble each other closely, limiting the benefit of averaging.

To introduce further diversity, the random forest algorithm adds an additional step:

> At each node, instead of considering all n features when choosing the best split, we randomly choose a subset of $k < n$ features, and the algorithm is allowed to split only using this subset.

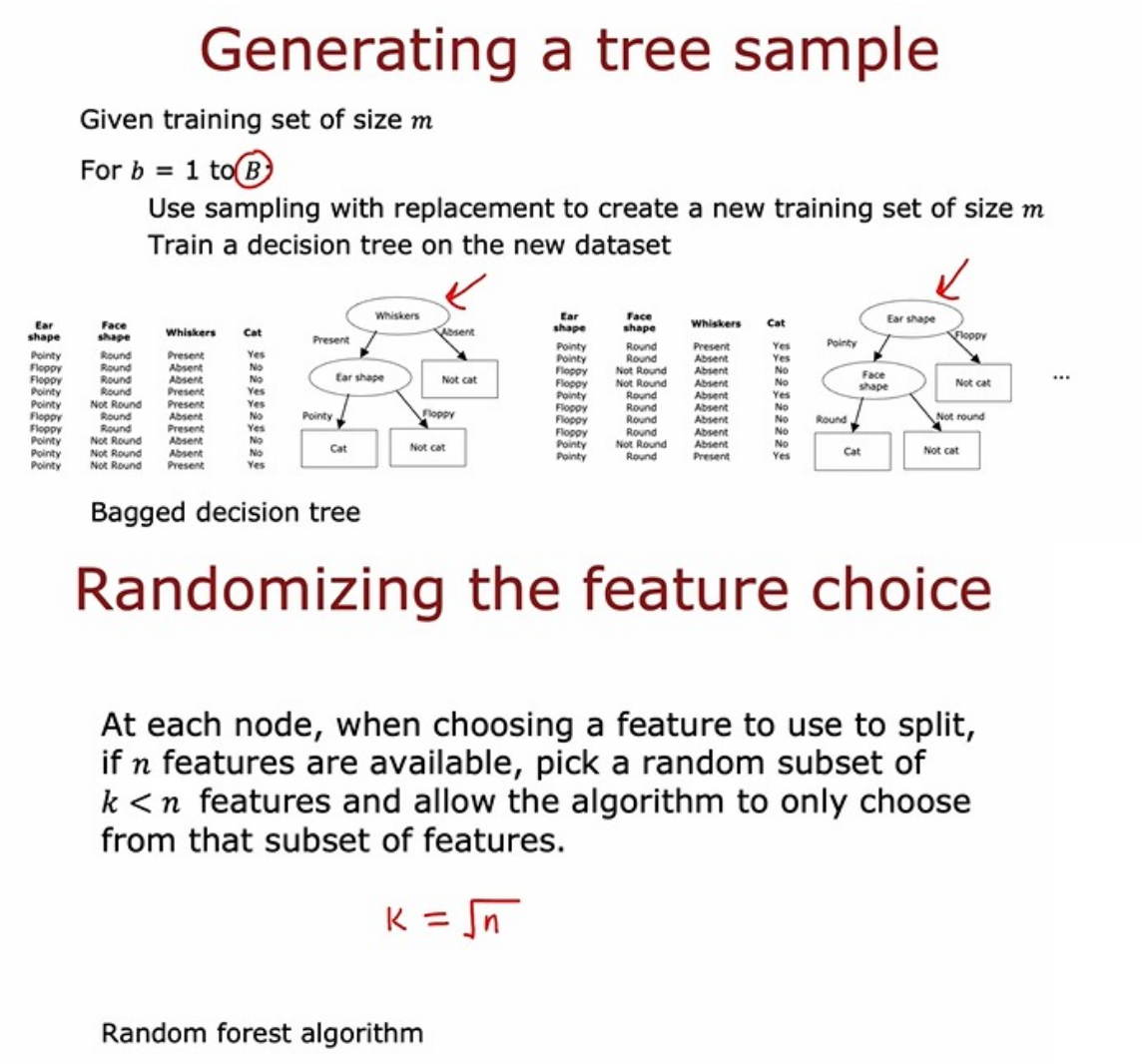

Figure 34: Bagging (sampling with replacement) creates multiple datasets for training multiple decision trees. Further randomization of feature choice leads to the random forest algorithm.

A common choice for k is:

$$k = \sqrt{n}.$$

This restriction forces different trees to consider different features at various nodes, increasing diversity and producing a stronger ensemble.

Why Random Forests Are More Robust

The random forest algorithm combines:

- variability introduced by sampling with replacement, and
- variability introduced by randomizing the feature choice.

Because the model is already averaging across many small perturbations of the training data, further small changes to the dataset have little effect on the final output. Random

forests are therefore far less sensitive to noise and produce more accurate predictions compared to a single decision tree.

This makes random forests one of the most effective and widely used ensemble methods in machine learning.

Before concluding, here is a joke often told in machine learning:

> Where does a machine learning engineer go camping? **In a random forest.**

In the next section, we will look at an even more powerful technique: **boosted decision trees**, such as the popular XGBoost algorithm.

Boosted Trees Intuition and XGBoost

Boosted Trees Intuition

Boosting is a powerful technique for improving the performance of decision tree ensembles. Consider a training set of size m. The boosting idea modifies the bagging procedure as follows:

1. For $b = 1$ to B:
   (a) Use sampling with replacement to generate a new training set of size m. However, instead of selecting each example with equal probability $1/m$, increase the probability of selecting examples that were misclassified by the trees trained so far.
   (b) Train a new decision tree on this newly weighted dataset.

In the first iteration ($b = 1$), all training examples have equal probability of selection. After the first tree is trained, we evaluate its performance on the original dataset and mark which examples were classified correctly or incorrectly. In subsequent iterations, examples that the ensemble still misclassifies are given higher probability when sampling, causing the next tree to focus more attention on these harder examples.

This idea is analogous to *deliberate practice*: when learning a complex skill, focusing on the parts that are most difficult leads to faster and more efficient learning. Similarly, boosting directs each new tree to concentrate on the mistakes of the previous trees, helping the ensemble improve more quickly.

Thus, boosting repeatedly updates the sampling distribution so that tree b tries to do well on the examples that trees 1 through $(b-1)$ are still struggling with. Although the exact mathematical formula for changing the sampling probabilities is intricate, users do not need to understand these details to use modern boosted tree implementations.

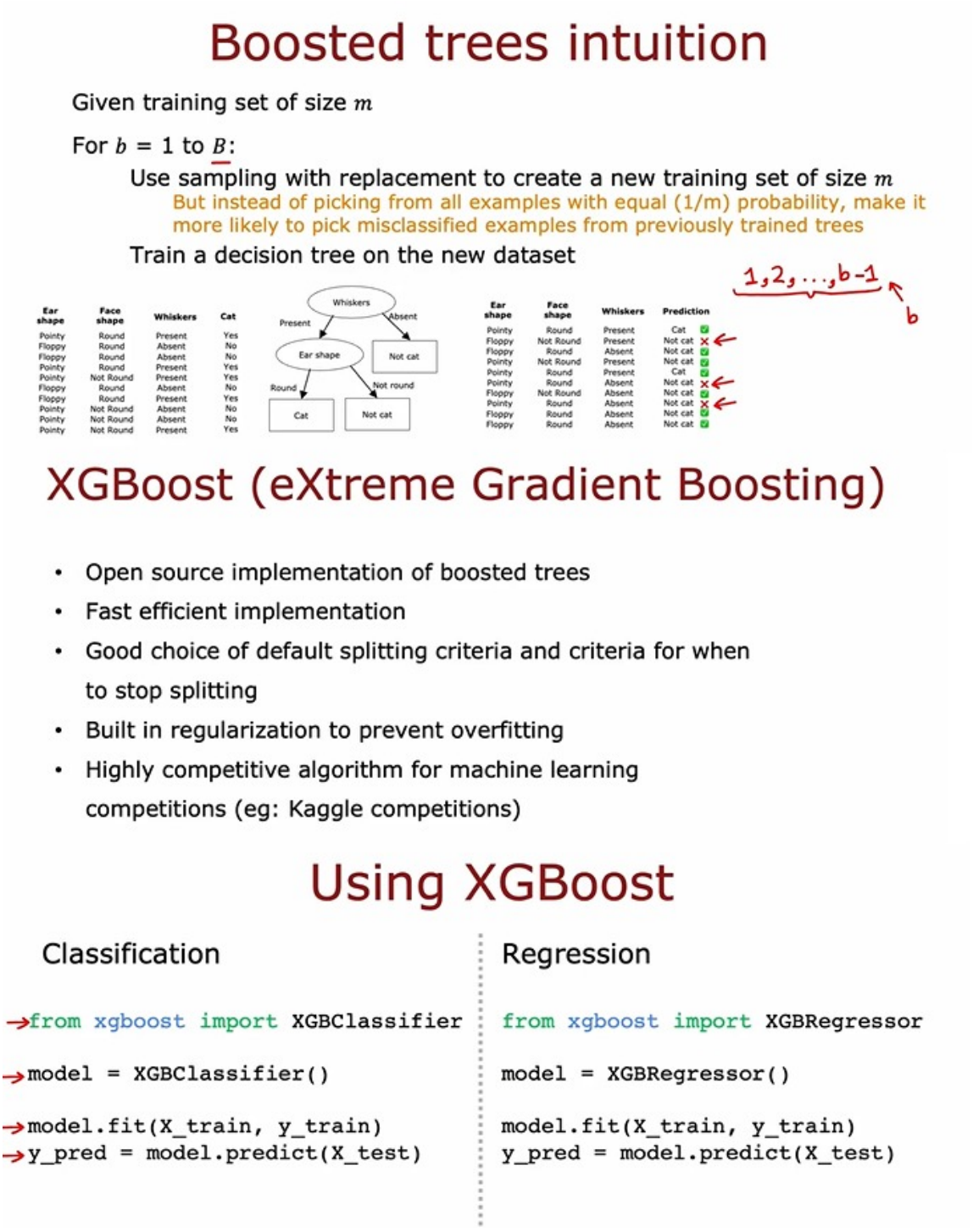

Figure 35: Boosted trees intuition: later trees focus more on previously misclassified examples, improving overall ensemble accuracy.

XGBoost (Extreme Gradient Boosting)

Over the years, many methods for building decision tree ensembles have been proposed. Today, **XGBoost (Extreme Gradient Boosting)** is by far one of the most widely used and effective algorithms. It is:

- an open-source, highly optimized implementation of boosted trees,
- extremely fast and efficient,
- equipped with excellent default splitting criteria,
- equipped with effective criteria for determining when to stop splitting,
- built with regularization to reduce overfitting,
- widely used in commercial applications and machine learning competitions such as Kaggle.

A major innovation in XGBoost is that it does not need to generate many bootstrapped datasets. Instead of performing sampling with replacement, XGBoost assigns *weights* to training examples and updates these weights after each iteration. This makes the algorithm more efficient while preserving the same intuition: later trees focus more heavily on examples the ensemble is still misclassifying.

Although the full mathematical details of XGBoost are complex, practitioners rarely need to implement them manually. Modern libraries provide efficient and easy-to-use interfaces.

Using XGBoost

XGBoost can be used for both classification and regression. The basic workflow is straightforward:

Classification

```
from xgboost import XGBClassifier

model = XGBClassifier()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

Regression

```
from xgboost import XGBRegressor

model = XGBRegressor()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

XGBoost remains one of the most competitive algorithms in machine learning. Together with deep learning, it is a frequent winner in machine learning competitions. In practice, many data scientists use the library implementation directly, allowing them to benefit from the speed, regularization, and robust performance of boosted decision trees.

In the next lesson, we discuss when to prefer decision trees or when a neural network may be the better choice.

Decision Trees vs Neural Networks

Overview

Both decision trees (including tree ensembles) and neural networks are powerful and widely used learning algorithms. Choosing between them depends on the type of data you are working with, computational constraints, interpretability needs, and the structure of the learning problem.

Decision Trees and Tree Ensembles

Decision trees and tree ensembles (such as Random Forests and XGBoost) often perform extremely well on **tabular, structured data**. If your dataset resembles a spreadsheet — with features such as numerical values, categories, or mixed categorical–numerical columns — then tree-based models are typically strong candidates. Examples include:

- Housing price prediction
- Customer churn classification
- Medical datasets with discrete and continuous features

Tree-based models support both classification and regression tasks and can easily handle both categorical and continuous-valued features.

However, they are **not recommended** for unstructured data such as:

- Images
- Video

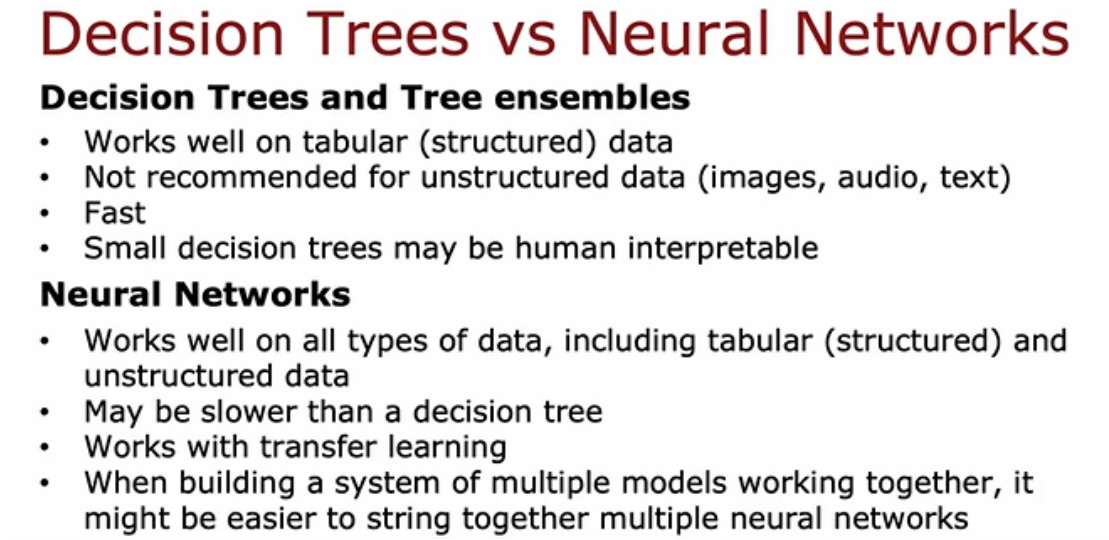

Figure 36: Comparison of Decision Trees and Neural Networks for different data types and use cases.

- Audio
- Text

Neural networks dominate these domains.

A key advantage of decision trees is that they are generally **fast to train**. This speed accelerates the iterative loop of machine learning development, allowing you to test ideas and improve model performance more quickly. Small trees may also be **human interpretable**, allowing you to inspect the learned structure and understand how decisions are made.

Tree ensembles like XGBoost are usually more accurate than a single tree. If computational resources allow, you would almost always prefer a tree ensemble. Only in extremely resource-constrained settings might you choose a single, simpler tree.

For most practical applications, **XGBoost** is a strong default choice because of its efficiency, regularization, and excellent predictive performance.

Neural Networks

Neural networks work well on **all types of data**, including:

- Tabular (structured) data
- Unstructured data (images, video, audio, text)
- Mixed structured + unstructured datasets

On structured tabular problems, neural networks are often competitive with tree ensembles. But on unstructured data, neural networks are generally the preferred algorithm because they exploit patterns that trees cannot capture.

A disadvantage is that neural networks may be **slower to train** — large networks can take hours, days, or even weeks to train depending on dataset size and architecture.

A major benefit is that neural networks support **transfer learning**, which is crucial when your dataset is small. Pretrained models learned from large datasets (such as ImageNet or large text corpora) can be fine-tuned to perform extremely well with relatively little data.

Neural networks are also easier to combine into complex multi-model systems. Multiple networks can be chained together and trained jointly using gradient descent, which is not possible with multiple decision trees.

Final Remarks

This concludes the comparison between decision trees and neural networks. You have learned how to build and use both families of algorithms and how to choose between them based on the problem at hand. While supervised learning requires labeled datasets, future topics such as unsupervised learning will allow you to explore powerful algorithms that do not require labeled outputs.

Before proceeding, we hope you enjoy experimenting with decision trees and tree ensembles in the practice quizzes and labs.

End of Section – Add next topic below using \section{}
