Intro to Neural Networks II

The Problem: Learning Slowdown I

- The *Quadratic Cost* (or Mean Squared Error) is a common starting point:

$$J = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2$$

- The gradient terms for weight (w) and bias (b) in the output layer are:

$$\frac{\partial J}{\partial w^L_{jk}} = (a^L_j - y_j)\sigma'(z^L_j)a^{L-1}_k$$

$$\frac{\partial J}{\partial b^L_j} = (a^L_j - y_j)\sigma'(z^L_j)$$

The Problem: Learning Slowdown II

- **The Problem:** The $\sigma'(z_j^L)$ term.
- When the neuron is "saturated" (output a_j^L is close to 0 or 1), the sigmoid function becomes very flat.
- This means $\sigma'(z_j^L) \approx 0$.

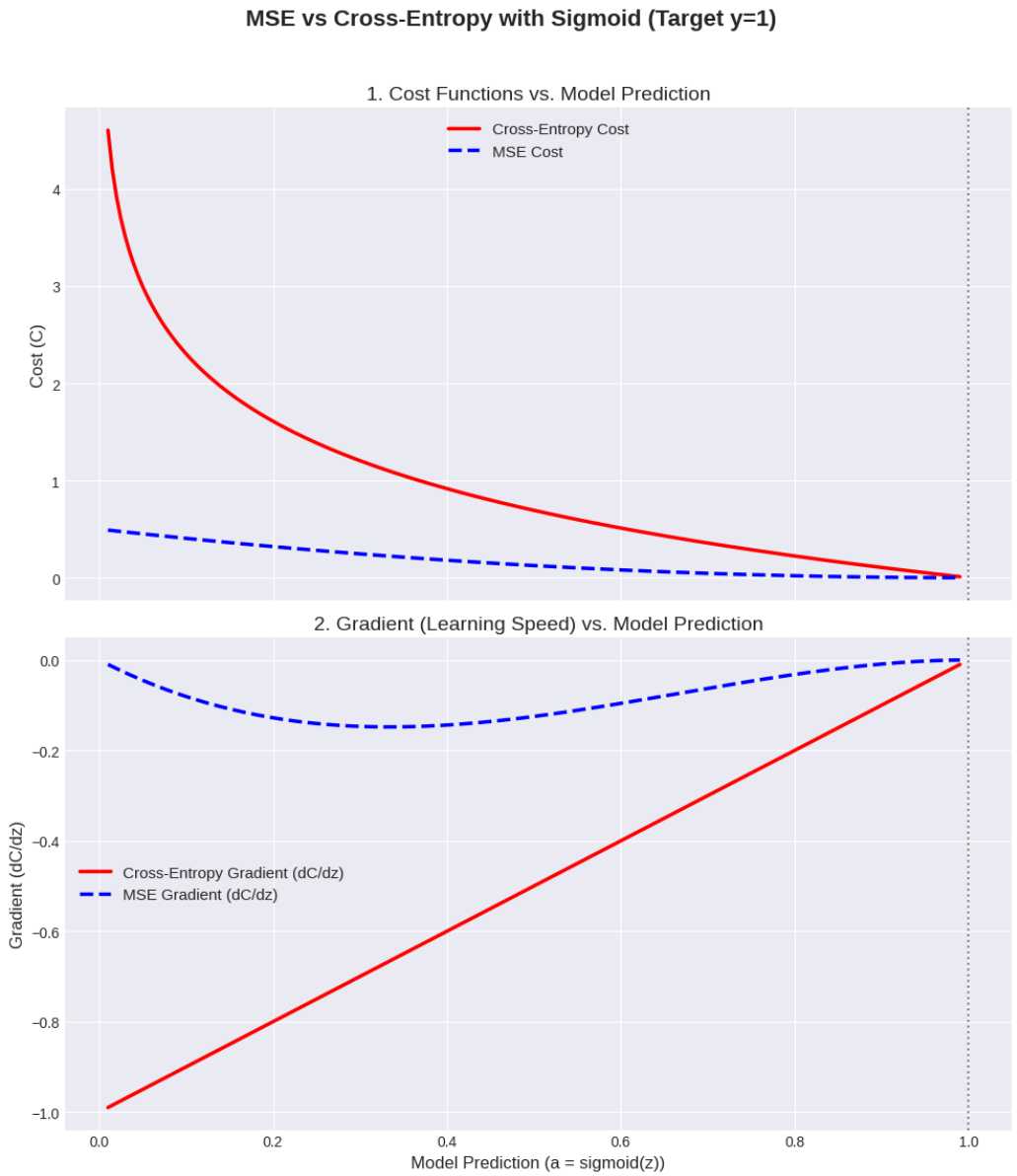
Consequence

Even if the error $(a_j^L - y_j)$ is large, the gradients become tiny, and the network learns **very slowly**.

Visualizing Learning Slowdown

Top: Both are 0 when prediction is 1 (correct); both increase when prediction goes to 0 (wrong).

Bottom: MSE gradient goes to zero when the prediction is wrong (vanishing gradient).
Cross-entropy gradient is zero only when prediction is 1!



The Cross-Entropy Cost Function

Definition

For a single neuron with one input x, output $a = \sigma(z)$, and target y:

$$J = -\frac{1}{n} \sum_x [y \ln(a) + (1 - y) \ln(1 - a)]$$

where $a = \sigma(wx + b)$.

- **Why is this a cost function?**
  - It's non-negative ($J > 0$).
  - If the neuron's output a is close to the target y, the cost $J \to 0$. (e.g., if $y = 0$ and $a \to 0$, then $-\ln(1 - a) \to 0$).
- **The Big Idea:** It's designed to solve the learning slowdown problem.

Cross-Entropy: Derivation & Properties I

- Let's calculate the gradient for the output layer weights. We need $\frac{\partial J}{\partial w_{jk}^L}$.
- By the chain rule: $\frac{\partial J}{\partial w_{jk}^L} = \frac{\partial J}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial z_j^L}{\partial w_{jk}^L}$
- Let's compute the terms (for a single training example x):

$$\frac{\partial J}{\partial a_j^L} = -\left(\frac{y_j}{a_j^L} - \frac{1-y_j}{1-a_j^L}\right) = \frac{a_j^L - y_j}{a_j^L(1-a_j^L)}$$

$$\frac{\partial a_j^L}{\partial z_j^L} = \sigma'(z_j^L) = \sigma(z_j^L)(1-\sigma(z_j^L)) = a_j^L(1-a_j^L)$$

proof

$$\frac{\partial z_j^L}{\partial w_{jk}^L} = x$$

Cross-Entropy: Derivation & Properties II

- **Combine them:**

$$\frac{\partial J}{\partial w_{jk}^L} = \frac{a_j^L - y_j}{a_j^L(1-a_j^L)} \cdot \left(a_j^L(1-a_j^L)\right) \cdot x$$

The Result

The $a_j^L(1-a_j^L)$ and $\sigma'(z)$ terms **cancel out**!

$$\frac{\partial J}{\partial w_{jk}^L} = (a_j^L - y_j)x$$

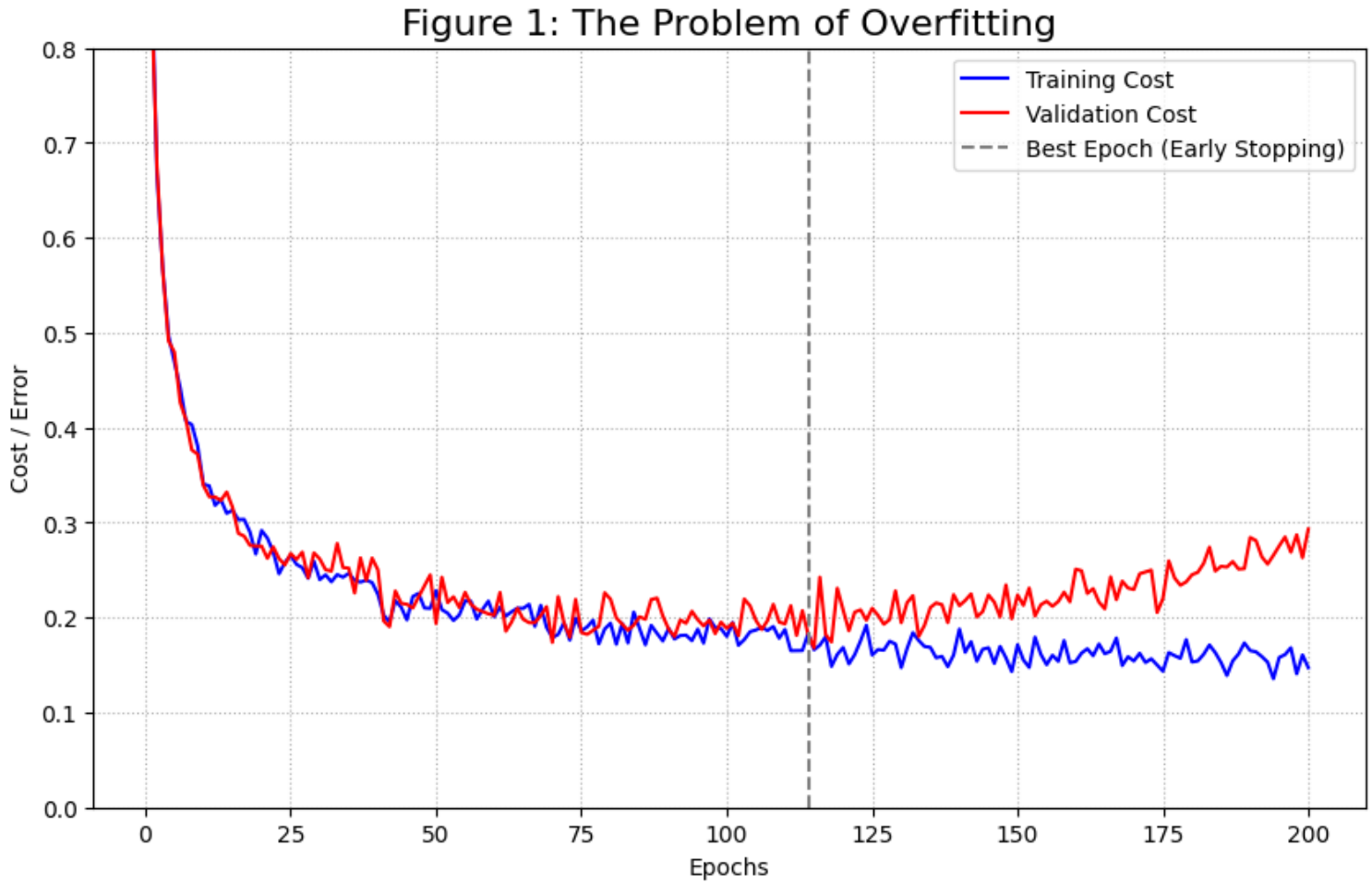
$$\frac{\partial J}{\partial b_j^L} = (a_j^L - y_j)$$

- The gradient is now proportional to the **error** $(a_j^L - y_j)$. A larger error means a larger gradient and faster learning.

The Problem: Overfitting

- **Definition:** A model is **overfitting** when it learns the training data *too well*, including its noise.
- It fails to **generalize** to new, unseen data (like the validation or test sets).
- **Symptom:**
  - Training accuracy keeps improving (or training cost keeps decreasing).
  - Validation accuracy plateaus or, even worse, starts to get *worse*.
- This often happens when a model has too much capacity (too many parameters) for the amount of data available.

Visualizing Overfitting



Regularization: The General Idea

- If overfitting is the problem, how do we fight it?
- **1. Get More Training Data (The Best Way)**
  - A larger, more diverse dataset is the most effective defense.
  - If this isn't possible, we can...
- **2. Artificially Augment Training Data**
  - Create "new" training samples by transforming existing ones.
  - **For images:** Rotate, translate (shift), add random noise, or apply elastic distortions.
  - This teaches the network to be robust to these variations.
- **3. Use Regularization Techniques**
  - Add a "complexity penalty" to the cost function to discourage the network from learning complex patterns.
  - We will look at L2, L1, and Dropout.

Regularization: The General Idea

- If overfitting is the problem, how do we fight it?
- **1. Get More Training Data (The Best Way)**
  - A larger, more diverse dataset is the most effective defense.
  - If this isn't possible, we can...
- **2. Artificially Augment Training Data**
  - Create "new" training samples by transforming existing ones.
  - **For images:** Rotate, translate (shift), add random noise, or apply elastic distortions.
  - This teaches the network to be robust to these variations.
- **3. Use Regularization Techniques**
  - Add a "complexity penalty" to the cost function to discourage the network from learning complex patterns.
  - We will look at L2, L1, and Dropout.

Regularization: The General Idea

- If overfitting is the problem, how do we fight it?
- **1. Get More Training Data (The Best Way)**
  - A larger, more diverse dataset is the most effective defense.
  - If this isn't possible, we can...
- **2. Artificially Augment Training Data**
  - Create "new" training samples by transforming existing ones.
  - **For images:** Rotate, translate (shift), add random noise, or apply elastic distortions.
  - This teaches the network to be robust to these variations.
- **3. Use Regularization Techniques**
  - Add a "complexity penalty" to the cost function to discourage the network from learning complex patterns.
  - We will look at L2, L1, and Dropout.

Regularization: L2 (Weight Decay) I

Idea

Penalize large weights. A network with small weights is "simpler" and less prone to overfitting.

- We add a regularization term to the cost function J_0:

$$J = J_0 + \frac{\lambda}{2n} \sum_w w^2$$

- J_0 = original cost (e.g., cross-entropy)
- $\lambda > 0$ = the regularization hyperparameter.

Regularization: L2 (Weight Decay) II

- **How does this change the gradient?**

$$\frac{\partial J}{\partial w} = \frac{\partial J_0}{\partial w} + \frac{\lambda}{n} w$$

$$\frac{\partial J}{\partial b} = \frac{\partial J_0}{\partial b} \quad \text{(Biases are usually not regularized)}$$

- **The SGD Update Rule:**

$$\begin{aligned} w &\to w - \eta \left(\frac{\partial J_0}{\partial w} + \frac{\lambda}{n} w \right) \\ &= \left(1 - \frac{\eta \lambda}{n} \right) w - \eta \frac{\partial J_0}{\partial w} \end{aligned}$$

Regularization: L2 (Weight Decay) III

Weight Decay

The term $\left(1 - \frac{\eta\lambda}{n}\right)$ scales the weight down on every step. This is why L2 is also called **weight decay**.

Regularization: L1 I

Idea

Penalize the *absolute value* of the weights.

- The L1-regularized cost function:

$$J = J_0 + \frac{\lambda}{n} \sum_w |w|$$

- **The SGD Update Rule:** (where sgn(w) is -1 if $w < 0$, $+1$ if $w > 0$)

$$\begin{aligned} w &\to w - \eta \left(\frac{\partial J_0}{\partial w} + \frac{\lambda}{n} \text{sgn}(w) \right) \\ &= w - \frac{\eta \lambda}{n} \text{sgn}(w) - \eta \frac{\partial J_0}{\partial w} \end{aligned}$$

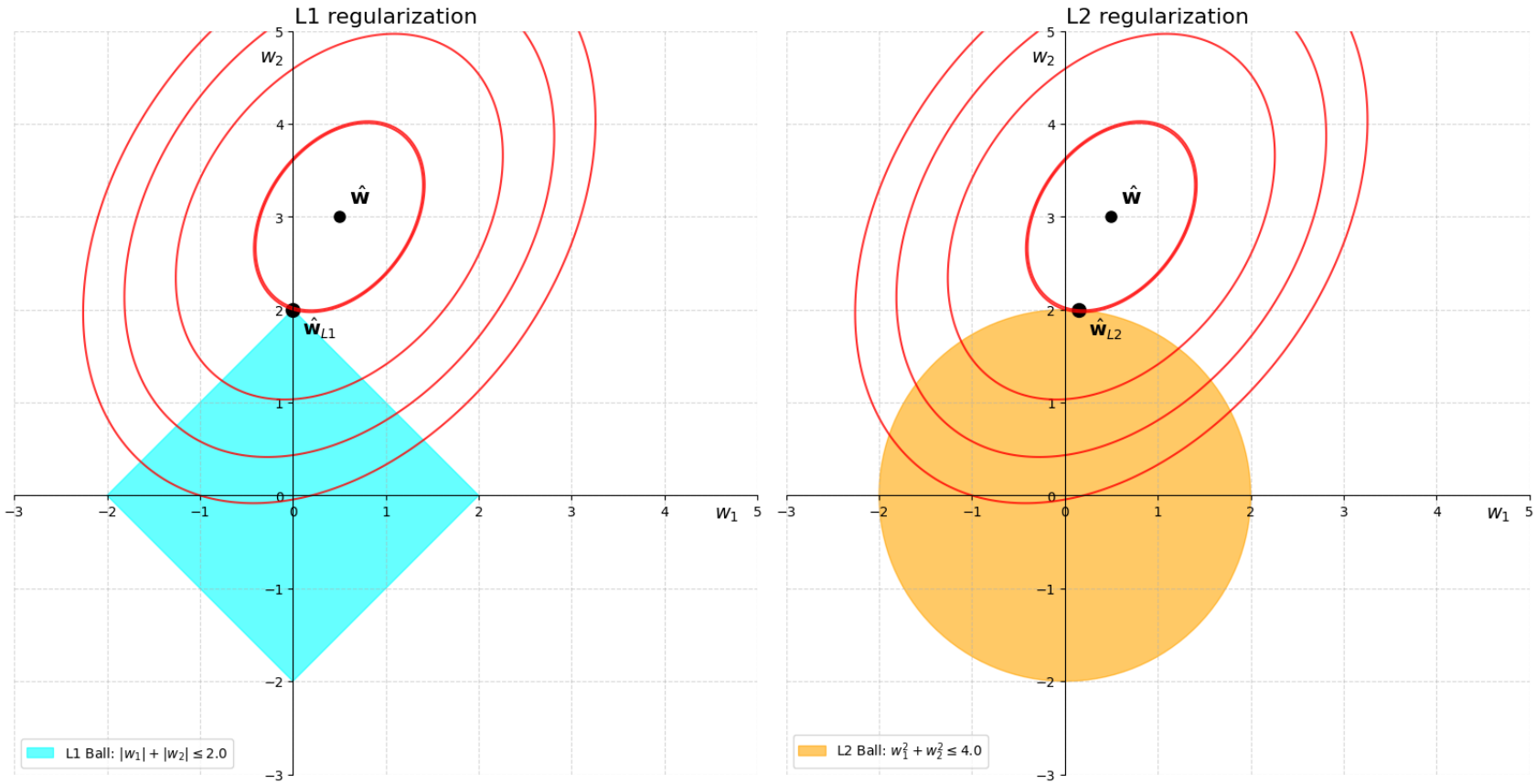
Regularization: L1 II

- **Difference from L2:**
  - **L2 (Weight Decay):** Shrinks weights by a *proportional* amount. Large weights are shrunk more than small weights.
  - **L1:** Shrinks weights by a *constant* amount (proportional to $\eta\lambda/n$), pushing them towards 0.

Sparsity

L1 regularization tends to set many weights to be **exactly zero**. This creates a **sparse model**, which can be useful for feature selection.

Visualizing L1 vs. L2



Regularization: Dropout I

Idea

A radically different approach. Instead of modifying the cost, it modifies the network itself.

- **During Training (for each mini-batch):**
  1. Go through each hidden layer.
  2. For each neuron, **randomly** "drop" it (set its output to 0) with some probability p (e.g., $p = 0.5$).
  3. Perform forward- and back-propagation on this "thinned" network.
  4. Update weights and biases.
  5. Repeat for the next mini-batch, with a new random set of dropped-out neurons.

Regularization: Dropout II

- **During Testing / Evaluation:**
  - Use the full, complete network (no neurons are dropped).
  - **BUT:** Multiply the weights outgoing from the hidden layers by the probability p (e.g., 0.5) to compensate for the fact that more neurons are active now.
- **Intuition:** It's like training a massive **ensemble** of different, smaller networks. It forces the network to learn redundant representations, as it cannot rely on any single neuron (it might be dropped!).

The Problem with Initialization

- **Classical Method:** Initialize weights w and biases b from a standard Gaussian (Normal) distribution with mean 0 and standard deviation 1.
- **The Problem:** Consider a neuron with n_{in} inputs.

$$z = \sum_{j=1}^{n_{in}} w_j x_j + b$$

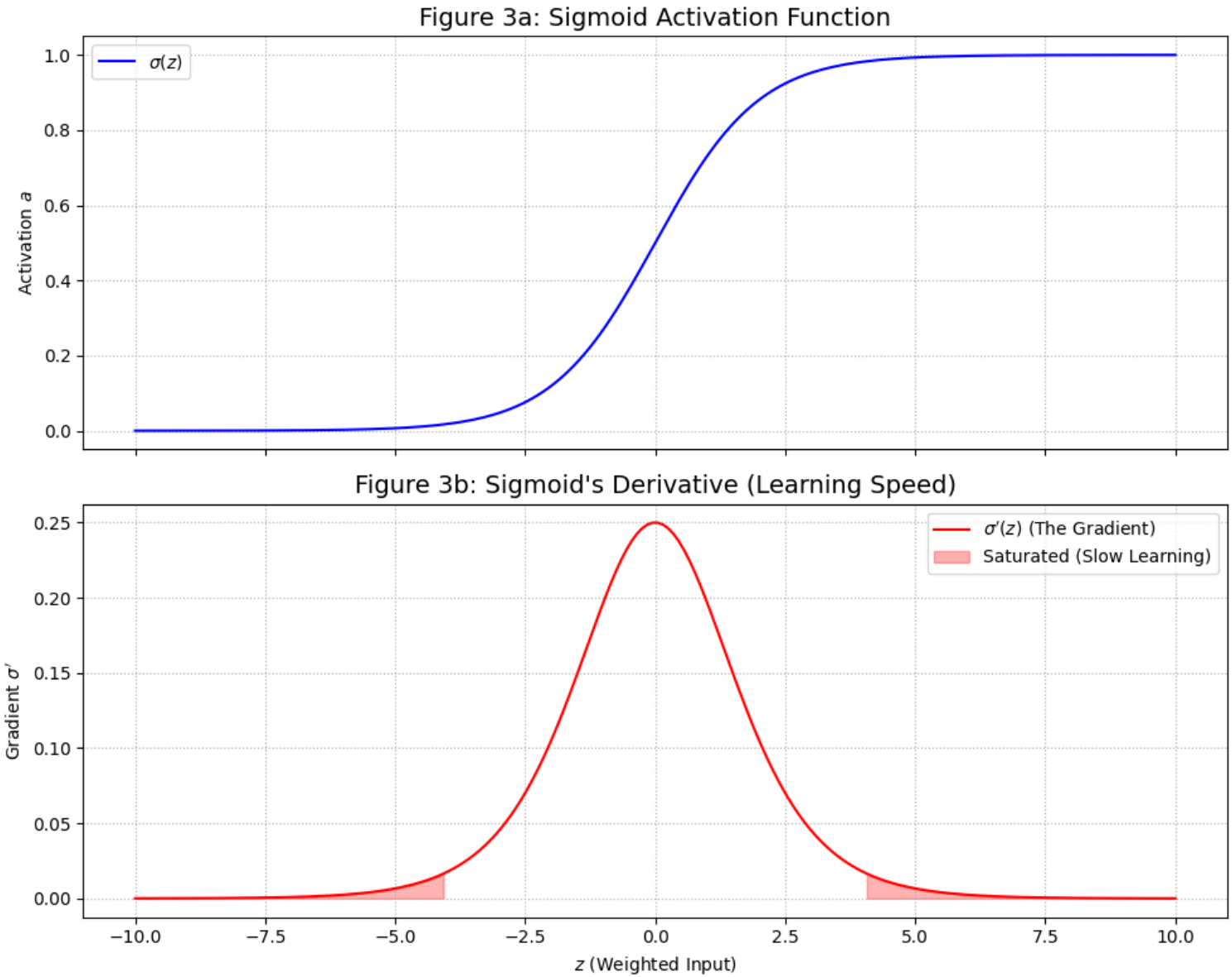
- Assume inputs x_j are normalized (e.g., $\sim 50\%$ are 0, $\sim 50\%$ are 1).
- The weighted sum z will also be a Gaussian distribution.
- **Its standard deviation will be large:** $\text{std}(z) \approx \sqrt{n_{in} \cdot \text{std}(w)^2} = \sqrt{n_{in}}$.

Consequence

If n_{in} is large (e.g., 1000 inputs), $\text{std}(z) \approx 31.6$. This means z is very likely to be a large positive or negative number.

- This pushes the neuron's output $\sigma(z)$ into the **saturated** regions (near 0 or 1), causing $\sigma'(z) \approx 0$ and **slow learning** right from the start.

Visualizing Saturation



A Better Weight Initialization I

The Solution

Initialize weights from a Gaussian distribution with mean 0, but change the standard deviation.

$$w \sim \mathcal{N}(0, 1/\sqrt{n_{in}})$$

where n_{in} is the number of input connections to the neuron.

Why does this work?

- Let's re-calculate the standard deviation of $z = \sum w_j x_j + b$:

$$\text{std}(z) = \sqrt{n_{in} \cdot \text{std}(w)^2} = \sqrt{n_{in} \cdot (1/\sqrt{n_{in}})^2}$$

$$\text{std}(z) = \sqrt{n_{in} \cdot (1/n_{in})} = \sqrt{1} = 1$$

A Better Weight Initialization II

Benefit

The weighted sum z is now a well-behaved Gaussian $\mathcal{N}(0, 1)$.

- This keeps the neuron in the "active" region of the sigmoid function, where $\sigma'(z)$ is not close to zero.
- This (along with cross-entropy) allows learning to start at a much healthier pace.
- Biases can be initialized as $\mathcal{N}(0, 1)$ or just as 0.

Hyperparameter Tuning as a "Black Art" I

- Hyperparameters (HPs) are the settings we don't learn, but *set* before training:
  - Learning rate η
  - Regularization parameter λ
  - Mini-batch size (see later)
  - Number of epochs
  - Network architecture (layers, neurons)
- There is no simple rule; it's an iterative process of experimentation.

Hyperparameter Tuning as a "Black Art" II

Broad Strategy

1. **Simplify first.** Turn off all regularization ($\lambda = 0$, no dropout).
2. **Find η.** Find the "order of magnitude" for the learning rate η that causes the *training cost* to start decreasing.
3. **How to find η:** Try $\eta = 0.01$.
   - If cost explodes/oscillates, decrease (e.g., $\eta = 0.001$).
   - If cost decreases too slowly, increase (e.g., $\eta = 0.1, 1.0$).
4. **Tune λ:** Once you have a reasonable η, turn on regularization. Start with $\lambda = 1.0$ and try tuning it on a log-scale (e.g., $0.1, 10.0$). Use the **validation accuracy** to pick the best λ.
5. **Re-tune η:** Your best λ might require a new, fine-tuned η.
6. **Tune Mini-batch size:** This often interacts with η. A smaller batch size might need a smaller η. Often, you pick a size that fits your GPU memory and tune η accordingly.

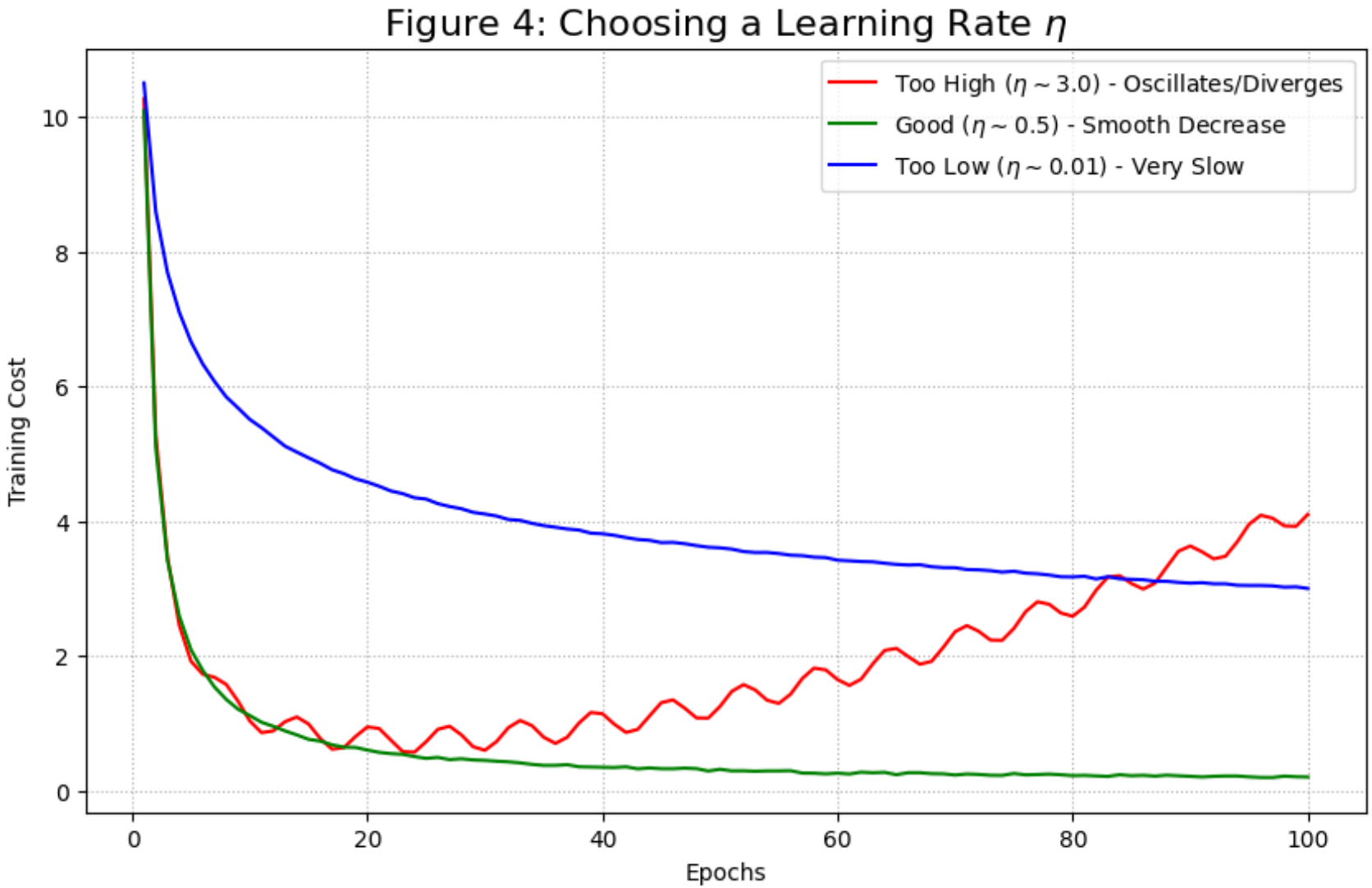
Tuning: The Learning Rate (η)

- η is arguably the most important hyperparameter.
- **Too High:** The cost will oscillate wildly or "explode" (NaNs).
- **Too Low:** The cost will decrease, but *extremely* slowly. You'll waste time.
- **Just Right:** The cost decreases steadily.

The "Golden Rule"

Monitor **validation accuracy**. Train with your starting η (e.g., $\eta = 0.1$). When the validation accuracy stops improving ("plateaus"), divide η by a factor (e.g., 2 or 10) and keep training.
Repeat this "learning rate schedule" several times.

Visualizing Learning Rates



Final Summary

- **Cross-Entropy:** Solves the learning slowdown from saturated neurons by ensuring the gradient is proportional to the error.
- **Regularization:** A set of techniques (L2, L1, Dropout, Augmenting Data) to combat overfitting and help the model generalize.
- **Weight Initialization:** Using $\mathcal{N}(0, 1/\sqrt{n_{in}})$ prevents neurons from saturating at the start of training.
- **Hyperparameters:** Tuning is an iterative process. Start simple, find a working η, then tune λ on the validation set.

Key Takeaway

These techniques work together to make training deep networks **faster**, **more stable**, and **more effective**.

The Sigmoid Function

Definition

The sigmoid function, $\sigma(x)$, is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Alternative Form

For differentiation, it is helpful to write it as:

$$\sigma(x) = (1 + e^{-x})^{-1}$$

We will prove that the derivative of the sigmoid function is:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Step 1: Apply the Chain Rule

Let $u = 1 + e^{-x}$ and $\sigma(u) = u^{-1}$. The chain rule is: $\frac{d\sigma}{dx} = \frac{d\sigma}{du} \cdot \frac{du}{dx}$

Part 1: Find $\frac{d\sigma}{du}$

$$\frac{d\sigma}{du} = \frac{d}{du}(u^{-1})$$

$$\frac{d\sigma}{du} = -1 \cdot u^{-2} = -\frac{1}{u^2}$$

Part 2: Find $\frac{du}{dx}$

$$\frac{d}{dx}(1 + e^{-x})$$

$$\frac{du}{dx} = -e^{-x}$$

Step 1: Combine the Parts

- **Substitute the parts:**

$$\sigma'(x) = \left(-\frac{1}{u^2}\right) \cdot (-e^{-x})$$

- **Substitute u back in:**

$$\sigma'(x) = \left(-\frac{1}{(1+e^{-x})^2}\right) \cdot (-e^{-x})$$

- **Simplify to get the derivative:**

Derivative

$$\sigma'(x) = \frac{e^{-x}}{(1+e^{-x})^2}$$

Step 2: Algebraic Manipulation

- **Start with our derivative:**

$$\sigma'(x) = \frac{e^{-x}}{(1+e^{-x})^2}$$

- **Add and subtract 1 in the numerator:**

$$\sigma'(x) = \frac{(1+e^{-x})-1}{(1+e^{-x})^2}$$

- **Split into two fractions:**

$$\sigma'(x) = \frac{1+e^{-x}}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2}$$

Conclusion: The Final Form

- **Simplify the expression:**

$$\sigma'(x) = \frac{1}{1+e^{-x}} - \left(\frac{1}{1+e^{-x}}\right)^2$$

- **Recall the definition:**

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

- **Substitute $\sigma(x)$ back in:**

$$\sigma'(x) = \sigma(x) - (\sigma(x))^2$$

- **Factor out $\sigma(x)$:**

Final Result

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

back