Grandient Descent

- **Title:** *Feeling the Slope: Gradient Descent with the Night-Hiker Analogy*
- **Course / Session:**
- **Instructor:** □ Prof. Ramesh Babu
- **Date:** □ Sep5 (Week4 day3)
- **Links/QR:** □ Colab □ GitHub □ Slides □ Dataset

Learning Objectives (by the end of class students can…)

1. Explain gradient descent using the night-hiker analogy (local info ↔ gradient; stride ↔ learning rate).
2. Write and interpret the core GD update rule and stopping criteria.
3. Choose appropriate loss functions for regression vs. classification (MSE, BCE-with-logits, softmax cross-entropy).
4. Describe backprop at a high level (chain rule; autograd).
5. Diagnose common training failures (bad LR, logits/probabilities mix-ups, missing `zero_grad`) and apply fixes.
6. Implement a minimal PyTorch/Keras training loop and apply early stopping / LR scheduling.

2) Time-boxed Agenda (60–90 min)

00:00–05:00 Icebreaker + Analogy setup (night hike in fog).
05:00–15:00 Core concepts: loss, gradient, learning rate, update rule.
15:00–25:00 Loss functions tour (MSE, BCE-with-logits, softmax CE; logits vs prob).
25:00–40:00 Backprop intuition (chain rule) + autograd demo.
40:00–55:00 Live coding: minimal training loop (+ early stopping).
55:00–65:00 Learning rate: symptoms, schedules, LR range test.
65:00–75:00 Pitfalls & debugging lab (pairs).
75:00–90:00 Wrap-up, quiz, Q&A, next session preview (batch vs SGD vs mini-batch).

RECAP

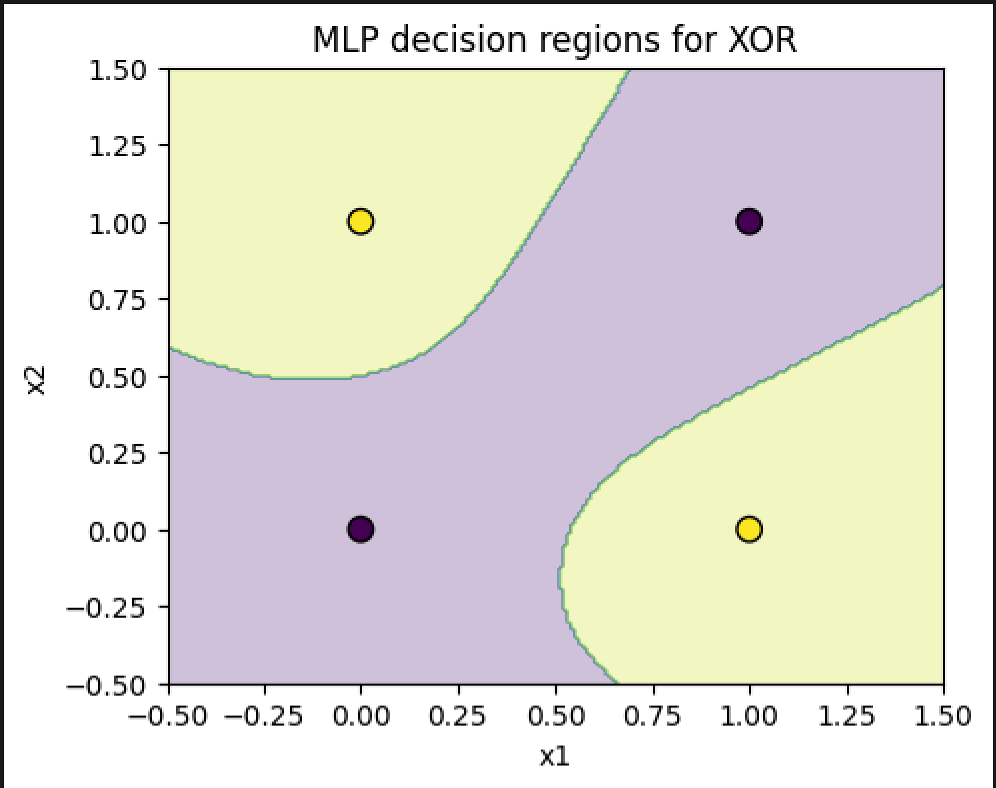
WEEK 1

Approach	How it Works	Example
Rule-Based	Human writes explicit rules	"If temperature > 30°C, recommend shorts"
Traditional ML	Human defines features, algorithm finds patterns	"Extract 20 weather features, train decision tree"
Deep Learning	Algorithm learns features AND patterns	"Give raw weather data, predict clothing"

WEEK 2

GRAPH

```
Final accuracy on XOR: 1.0
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 67ms/step
Predictions: [0 1 1 0]
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 1s 952us/step
```



WEEK 3

WHAT IS ACTIVATION

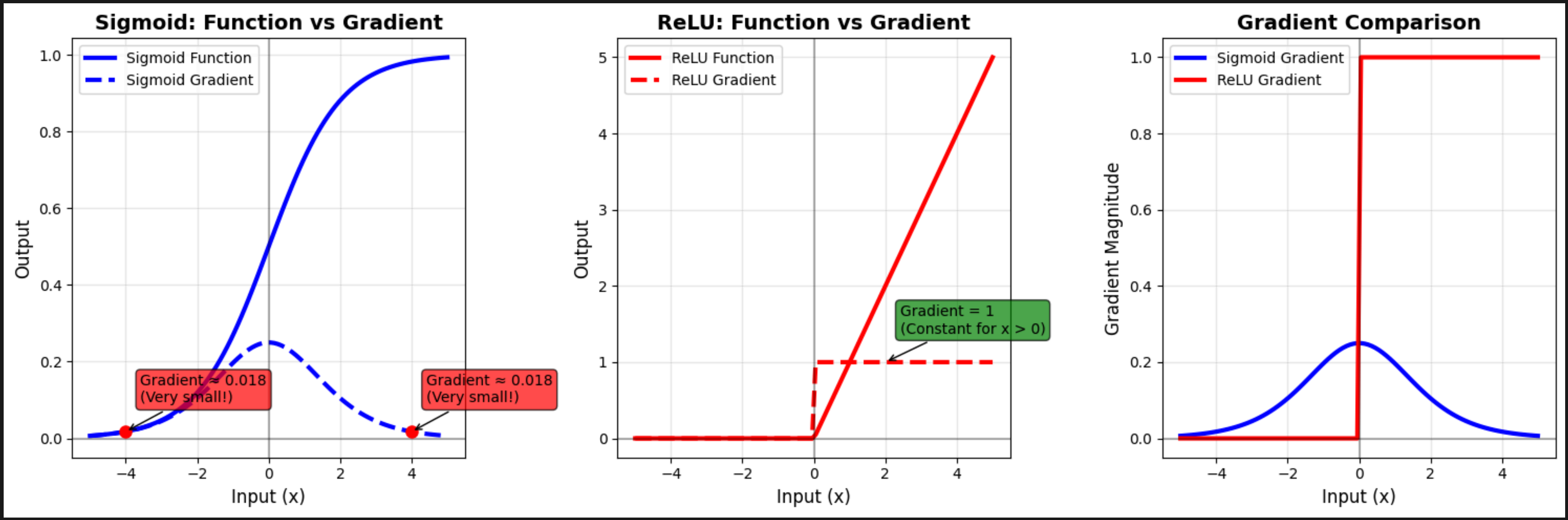
🔑 **Activations (the "gatekeepers" in a neural net)**

1. **ReLU (Rectified Linear Unit)**
   - Rule: pass positive values, block negatives (set them to 0).
   - Think: a **light switch** — off below 0, on above 0.
2. **Leaky ReLU**
   - Rule: same as ReLU, but negatives are not killed — they leak a little.
   - Think: a **safety valve** — lets a trickle of negative flow.
3. **Swish**
   - Rule: multiply input by a smooth sigmoid → negatives shrink but don't vanish.
   - Think: an **auto-dimmer** — dims weak signals smoothly.
4. **GELU (Gaussian Error Linear Unit)**
   - Rule: input gets passed depending on probability (via Gaussian curve).
   - Think: a **confidence gate** — only strong signals get fully through.

WHAT IS DERIVATIVES

📉 **Derivatives (how much the function "pushes" during learning)**

1. **ReLU derivative**
   - 0 for x < 0 → dead neurons possible.
   - 1 for x > 0 → strong, stable gradient.
2. **Leaky ReLU derivative**
   - Small slope (e.g. 0.1) when x < 0 → prevents dead neurons.
   - Slope = 1 when x > 0.
3. **Swish derivative**
   - Never flat zero → always some gradient.
   - Smoother changes help gradients flow better in deep nets.
4. **GELU derivative**
   - Curved like a Gaussian → soft, probabilistic slope.
   - Keeps gradients alive while tapering extremes.


Sigmoid: Function vs Gradient
Sigmoid Function
Sigmoid Gradient
Gradient ≈ 0.018
(Very small!)
Gradient ≈ 0.018
(Very small!)
Output
Input (x)
ReLU: Function vs Gradient
ReLU Function
ReLU Gradient
Gradient = 1
(Constant for x > 0)
Output
Input (x)
Gradient Comparison
Sigmoid Gradient
ReLU Gradient
Gradient Magnitude
Input (x)

WEEK-4

MODULE -2

OPTIMIZATION

- Optimization in deep neural network architecture is the systematic process of iteratively tuning network parameters using loss functions and gradient-based algorithms so that the model learns to map inputs to outputs with minimal error

Optimization (Definition)

Optimization is the **process of adjusting the parameters (weights and biases) of a neural network** to minimize the difference between the network's predictions and the actual target outputs.

Formally, optimization means finding the set of parameters **θ = {W, b}** that minimize a chosen **loss (or cost) function**:

$$\theta^* = \arg\min_{\theta} J(\theta; X, y)$$

where:

- θ → parameters of the network (weights, biases)
- $J(\theta; X, y)$ → loss function measuring error between prediction $f(X; \theta)$ and ground truth y
- X → input data, y → true labels

Core Concepts in Optimization for DNNs

1. **Loss Function (Objective Function)**
    - Guides the optimization process. Examples: Mean Squared Error (MSE), Cross-Entropy Loss.
    - Defines what "error" means in the problem.
2. **Optimization Algorithm (Optimizer)**
    - The method used to update weights.
    - Examples: **Gradient Descent, Stochastic Gradient Descent (SGD), Adam, RMSProp.**
3. **Gradient Computation (Backpropagation)**
    - Gradients of the loss w.r.t. parameters are computed using **backpropagation.**
    - Provides the "direction" in which weights should be adjusted.
4. **Learning Rate**
    - A critical hyperparameter that controls the step size in parameter updates.
    - Too high → divergence, too low → slow convergence.
5. **Convergence**
    - Optimization aims to reach (or approximate) a **global minimum** of the loss, though in deep networks often a **good local minimum** or **saddle point escape** is sufficient.
6. **Regularization & Constraints**
    - Techniques like **L1/L2 regularization, dropout, weight decay** help optimization avoid overfitting and improve generalization.

1. Loss Function (Objective Function)

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f(x_i; \theta), y_i)$$

🔷 **Analogy**:

Think of **loss** like the **distance between your current location and your destination** on a map.

- If you're hiking, the **loss** is how far you are from your camp.
- Your goal is to minimize that distance (loss) to eventually reach camp (best model).

2. Optimization Algorithm (Gradient Descent)

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta J(\theta^{(t)})$$

🔷 **Analogy**:

Imagine hiking **downhill in a foggy mountain**:

- You cannot see the whole path, but you feel the **slope of the ground under your feet** (the gradient).
- Step in the direction of **steepest descent** (negative gradient).
- With each step, you get closer to the valley (minimum loss).

3. Gradient Computation (Backpropagation)

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial W}$$

🔷 **Analogy:**

Imagine you're cooking and the dish tastes too salty.

- You trace back: **Taste (loss) → Too much salt (weight issue) → Recipe step (activation).**
- Backpropagation is like figuring out **which ingredient at which step** caused the bad taste.
- Then you adjust only that ingredient (weight update).

4. Learning Rate

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta J(\theta)$$

🔷 **Analogy:**

- **Learning rate = size of your steps while hiking downhill.**
  - If steps are **too big** → you may overshoot the valley and keep stumbling.
  - If steps are **too small** → you'll crawl very slowly.
  - A balanced step size helps you reach the valley efficiently.

5. Convergence

$$\theta^* = \arg\min_{\theta} J(\theta)$$

🔷 **Analogy:**

Reaching the **valley floor** while hiking.

- Once the slope feels nearly flat (small gradient), you've converged.
- You don't need to reach the absolute lowest point (global minimum), just a **good flat spot** where you can safely set up camp (local minimum).

. Convergence Checks

$$\|\nabla J(\theta)\| < \epsilon \quad \text{or} \quad |J_{t+1} - J_t| < \delta$$

6. Regularization

- **L2 (Weight Decay):**

$$J_{\text{reg}}(\theta) = J(\theta) + \lambda \sum_j \|w_j\|^2$$

- **L1 (Sparsity):**

$$J_{\text{reg}}(\theta) = J(\theta) + \lambda \sum_j |w_j|$$

🔷 **Analogy:**

Imagine packing for a trip:

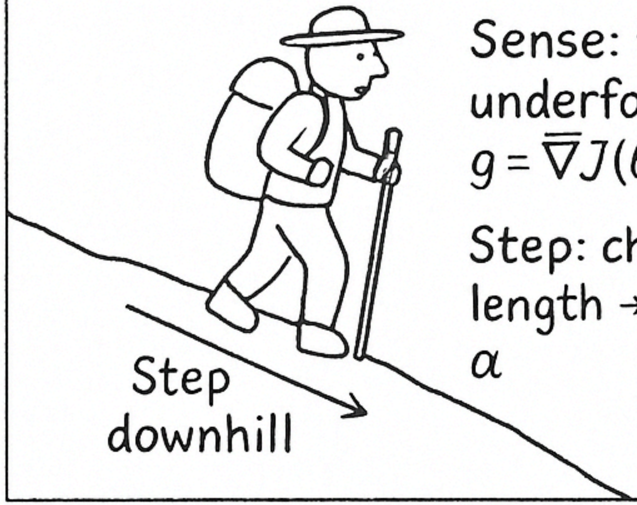
- If you carry **too much stuff (too many parameters)**, your journey becomes harder.
- Regularization is like an **extra fee for every extra item in your backpack.**
  - L2 = discourages carrying heavy things (large weights).
  - L1 = encourages you to pack fewer items (sparse weights).
- The result: You travel lighter and generalize be

Anchor Analogy (Night-Hiker → Neural Net)

- **Valley (goal):** minimum of loss function.
- **Local feel of ground:** gradient (points **uphill**; we step **against** it).
- **Stride length:** learning rate α.
- **Flat ground:** $\|\nabla J(\theta)\| \approx 0$ or $|\Delta J|$ small → **convergence**.
- **Treacherous terrain:** plateaus, saddles, cliffs → need momentum/adaptive LRs.

Pocket mantra: *Sense* → *Step* → *Move* → *Stop*.

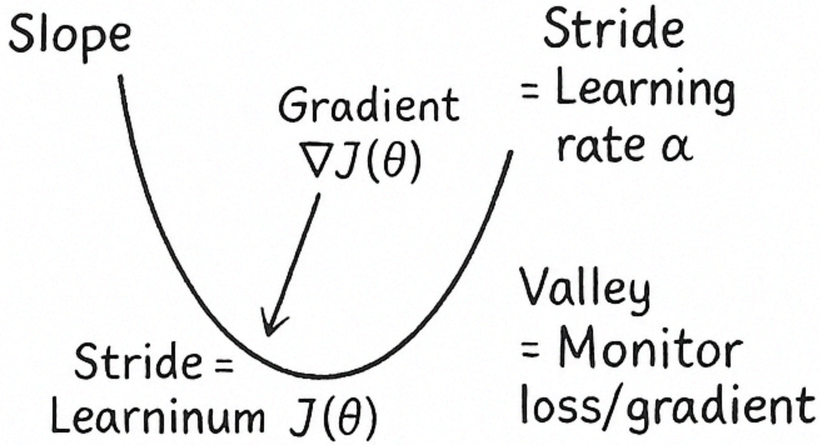
Story Hook: Hiking Donwhill



Sense: feel slope underfoot → gradient $g = \bar{\nabla} J(\theta)$

Step: choose stride length → learning rate α

From Hiking to Gradient Descent



Hiker's Rule (Algorithm)

Compute gradient $g = \bar{\nabla} J(\theta)$

↓

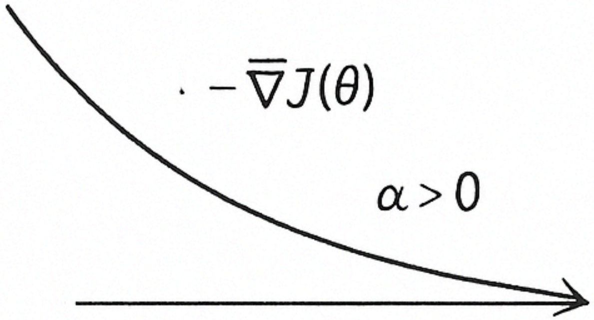
Update $\theta = \theta - \alpha g$

↓

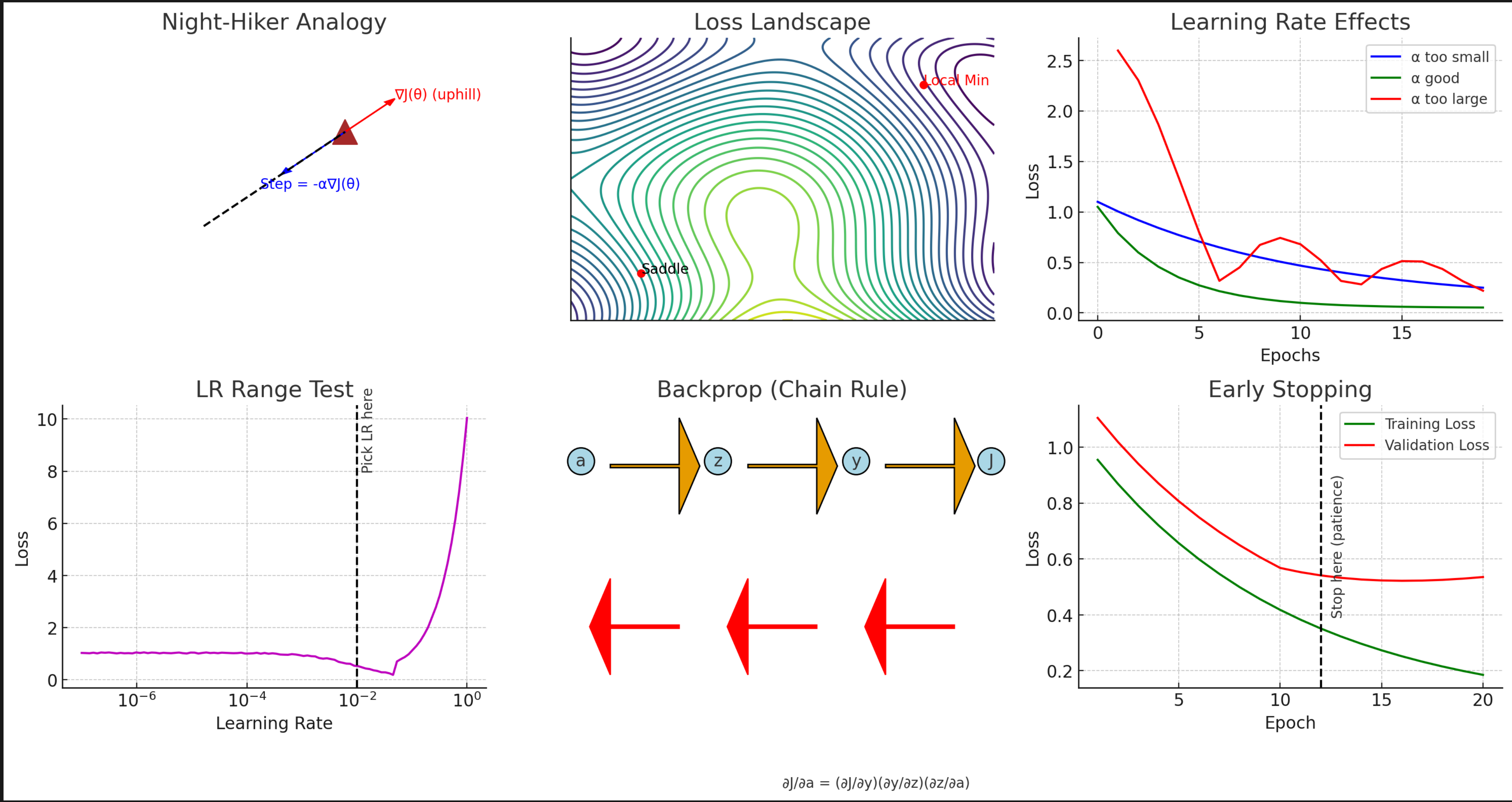
Stop when $|g| < \varepsilon$

If zigzagging → adjust α

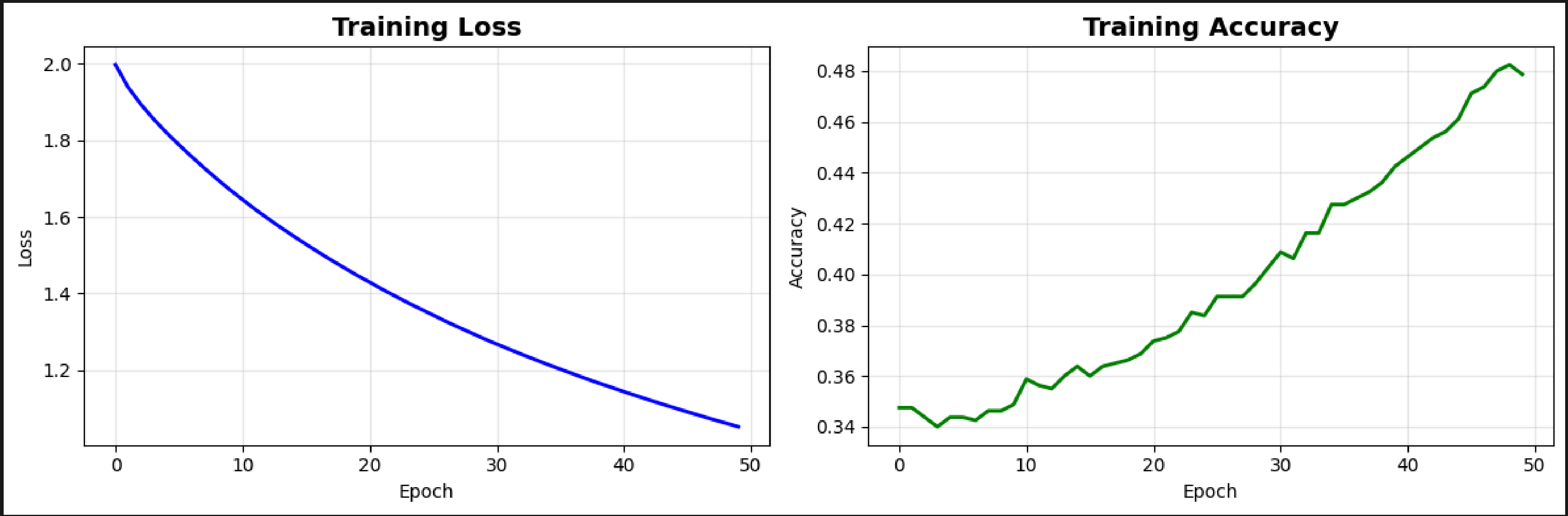
Gradient Descent in One Line



$$\theta_{new} = \theta + \alpha\, d = \theta - \alpha \nabla J(\theta)$$


Night-Hiker Analogy
∇J(θ) (uphill)
Step = -α∇J(θ)
Loss Landscape
Local Min
Saddle
Learning Rate Effects
α too small
α good
α too large
Loss
Epochs
LR Range Test
Pick LR here
Loss
Learning Rate
Backprop (Chain Rule)
a
z
y
J
Early Stopping
Training Loss
Validation Loss
Stop here (patience)
Loss
Epoch
∂J/∂a = (∂J/∂y)(∂y/∂z)(∂z/∂a)

CAN ANYONE EXPLAIN THIS DIAGRAM



```python
print("\n🏋 Training the network...")
classifier.fit(X_train, y_train, epochs=50, learning_rate=0.01, batch_size=32, verbose=True)

# Test the trained network
test_predictions = classifier.predict(X_test)
test_accuracy = classifier.compute_accuracy(y_test, test_predictions)

print(f"\n📊 Final Results:")
print(f"Test Accuracy: {test_accuracy:.4f} ({test_accuracy*100:.1f}%)")

# Plot training history
classifier.plot_training_history()

print("\n🎉 Training demonstration complete!")
print("This shows that your implementation can actually learn from data!")
```

```
🎯 Training a Neural Network on Synthetic Data
Training set: 800 samples
Test set: 200 samples
Features: 20
Classes: 3
✅ Dense layer created: 20 -> 32, activation: relu
   Weights shape: (20, 32)
   Bias shape: (1, 32)
✅ Dense layer created: 32 -> 16, activation: relu
   Weights shape: (32, 16)
   Bias shape: (1, 16)
✅ Dense layer created: 16 -> 3, activation: softmax
   Weights shape: (16, 3)
   Bias shape: (1, 3)

🏗 Neural Network Architecture:
   Layer 1: 20 -> 32 (relu)
   Layer 2: 32 -> 16 (relu)
   Layer 3: 16 -> 3 (softmax)
   Total parameters: 1,251

🏋 Training the network...
Epoch 10/50 - Loss: 1.6710 - Accuracy: 0.3488
Epoch 20/50 - Loss: 1.4473 - Accuracy: 0.3688
Epoch 30/50 - Loss: 1.2814 - Accuracy: 0.4025
Epoch 40/50 - Loss: 1.1546 - Accuracy: 0.4425
Epoch 50/50 - Loss: 1.0515 - Accuracy: 0.4788
```

TENSORFLOW / KERAS: OPTIMIZATION CORE CONCEPTS

1. Loss Function (Objective Function)

📦 **Module:** `tf.keras.losses`

- Provides standard loss functions.
- **Examples:**

```python
tf.keras.losses.MeanSquaredError()
tf.keras.losses.CategoricalCrossentropy(from_logits=True)
tf.keras.losses.BinaryCrossentropy()
```

2. Optimization Algorithm (Optimizer)

📦 **Module:** `tf.keras.optimizers`

- Implements algorithms for updating weights.
- **Examples:**

python

Copy code

```python
tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9)
tf.keras.optimizers.Adam(learning_rate=1e-3)
tf.keras.optimizers.RMSprop(learning_rate=0.001)
```

3. Gradient Computation (Backpropagation)

📦 **Module:**

- Built-in when you call `.fit()` in `tf.keras.Model`.
- For **manual control:**
  - `tf.GradientTape()` is used to record operations and compute gradients.

python

Copy code

```python
with tf.GradientTape() as tape:
    y_pred = model(x)
    loss = loss_fn(y_true, y_pred)
grads = tape.gradient(loss, model.trainable_variables)
optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

4. Learning Rate

📦 **Module:** `tf.keras.optimizers.schedules`

- Fixed or dynamic learning rates.
- **Examples:**

python

Copy code

```python
# Constant learning rate
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)

# Schedule (decays over time)
lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=0.1,
    decay_steps=100000,
    decay_rate=0.96,
    staircase=True
)
optimizer = tf.keras.optimizers.Adam(learning_rate=lr_schedule)
```

5. Convergence (Callbacks & Monitoring)

📦 **Module:** `tf.keras.callbacks`

- Control training stop conditions.
- **Examples:**

```python
tf.keras.callbacks.EarlyStopping(monitor="val_loss", patience=5, restore_best_weights=
tf.keras.callbacks.ReduceLROnPlateau(monitor="val_loss", factor=0.5, patience=3)
```

6. Regularization & Constraints

📦 **Modules:**

- `tf.keras.regularizers` → adds penalties on weights.
- `tf.keras.constraints` → restricts values of weights.
- `tf.keras.layers.Dropout` → randomly drops units during training.
- **Examples:**

python

Copy code

```python
# L2 regularization
tf.keras.layers.Dense(64, activation="relu",
                      kernel_regularizer=tf.keras.regularizers.l2(0.01))

# L1 regularization
tf.keras.layers.Dense(64, activation="relu",
                      kernel_regularizer=tf.keras.regularizers.l1(0.01))

# Dropout
tf.keras.layers.Dropout(0.5)

# Constraints (e.g., max norm on weights)
tf.keras.layers.Dense(64, activation="relu",
                      kernel_constraint=tf.keras.constraints.MaxNorm(2))
```

✅ Summary: Packages vs. Concepts

Concept	TensorFlow / Keras Module	Key Classes / Functions
Loss Function	`tf.keras.losses`	`MeanSquaredError`, `CategoricalCrossentropy`, `BinaryCrossentropy`
Optimizers	`tf.keras.optimizers`	`SGD`, `Adam`, `RMSprop`
Gradients / Backprop	`tf.GradientTape`	`.gradient()`, `.apply_gradients()`
Learning Rate	`tf.keras.optimizers.schedules`	`ExponentialDecay`, `PiecewiseConstantDecay`
Convergence	`tf.keras.callbacks`	`EarlyStopping`, `ReduceLROnPlateau`
Regularization	`tf.keras.regularizers`, `tf.keras.constraints`, `tf.keras.layers.Dropout`	`l1`, `l2`, `Dropout`, `MaxNorm`

QUESTIONS