**SVM**

**Summary: Introduction to Support Vector Machine (SVM)**

The video introduces **Support Vector Machine (SVM)**, a powerful machine learning algorithm used for both **classification** and **regression** tasks. The two main variants are:

* **Support Vector Classifier (SVC)** for classification problems.
* **Support Vector Regression (SVR)** for regression problems.

**Key Concepts Covered:**

1. **Prerequisite Knowledge:**
   * Understanding **logistic regression** is essential before diving into SVM, as both deal with creating decision boundaries.
2. **Geometric Intuition:**
   * In logistic regression:
     + A **line** separates two classes in 2D.
     + A **plane** separates classes in 3D.
     + A **hyperplane** is used in higher dimensions.
   * In SVM:
     + We also create a **hyperplane** to separate data.
     + Additionally, we create **two marginal planes** that are equidistant from the hyperplane.
     + The goal is to **maximize the margin** (distance between marginal planes).
3. **Support Vectors:**
   * Data points that lie closest to the decision boundary (hyperplane) and help define the position of the margin.
   * These are crucial in constructing the optimal hyperplane.
4. **Why Maximize Margin?**
   * A larger margin helps improve model generalization and reduces overfitting.
5. **2D vs 3D Intuition:**
   * In 2D: Line + marginal lines.
   * In 3D: Plane + marginal planes.
   * In higher dimensions: Hyperplane + corresponding margins.

**Summary: Soft Margin vs Hard Margin in Support Vector Machines (SVM)**

In this session, the concept of **Soft Margin** and **Hard Margin** in **Support Vector Classifiers (SVC)** is explained.

**Key Concepts:**

1. **Hard Margin:**
   * Used when data is **perfectly linearly separable**.
   * The **best fit line** and **marginal planes** can clearly divide the two classes **without any misclassification**.
   * **No tolerance for errors**.
   * Rare in real-world scenarios due to noise and overlaps.
2. **Soft Margin:**
   * Used when there is **overlap between classes**, i.e., data is **not perfectly separable**.
   * Allows for **some misclassifications or errors** by introducing flexibility.
   * Helps in better **generalization** on unseen data.
   * **More practical** for real-world applications.

**Conclusion:**

* Real-world data often requires **soft margin** SVM due to overlapping points and noise.
* **Hard margin** is ideal only when data is perfectly clean and separable, which is rare.

**❗ Problem in Real World**

* In **real datasets**, points are **not perfectly separable**. There can be **overlapping or noise**, so the hard-margin assumption fails.

**✅ Solution: Soft Margin SVM**

To handle misclassified or nearly misclassified points, we use:

**➤ Slack Variables ξi\xi\_iξi​:**

* Represent how much a data point **violates** the margin.
* If ξi=0\xi\_i = 0ξi​=0, point is correctly classified with margin.
* If ξi>0\xi\_i > 0ξi​>0, point is **within margin** or **misclassified**.

**✅ Soft Margin Cost Function (with Hinge Loss)**

We modify the cost function to allow some misclassification:

Minimize: 12∥w∥2+C∑i=1nξi\text{Minimize: } \frac{1}{2} \|w\|^2 + C \sum\_{i=1}^{n} \xi\_iMinimize: 21​∥w∥2+Ci=1∑n​ξi​

Where:

* ∥w∥2\|w\|^2∥w∥2: still tries to **maximize the margin** (make the classifier confident).
* CCC: is a **hyperparameter** controlling the trade-off between **margin size** and **misclassification penalty**.
* ∑ξi\sum \xi\_i∑ξi​: **Hinge Loss** = total amount of misclassification or margin violation.

**💡 Linear SVC:**

* Works well when **data is linearly separable**.
* Draws a straight line (or hyperplane) between classes with margins.
* Fails when data points are **not linearly separable** (e.g., overlapping or complex patterns).

**❌ Problem with Linear SVC:**

* In complex scenarios (e.g., circular or XOR-shaped distributions), a linear line cannot effectively separate classes.
* Leads to **low accuracy** and **high error**.

**🔁 Solution: SVM Kernels:**

* Kernels **transform the data into a higher dimension** where it becomes linearly separable.
* This transformation is done using **mathematical formulas** (i.e., kernel functions).
* After transformation, **linear SVC** can be applied in the new space.

**🧪 Example:**

* A 1D dataset (line) where classes are intermixed.
* Transform using y = x² to convert to 2D.
* Now, data becomes **clearly separable**, and linear SVC works.

**🔧 Types of Kernels:**

1. **Polynomial Kernel**
2. **RBF (Radial Basis Function) Kernel**
3. **Sigmoid Kernel**

Each applies a different transformation to map data to a higher-dimensional space.

| **Step** | **Code** | **Why It’s Used (SVC context)** |
| --- | --- | --- |
| Split Data | train\_test\_split | To ensure fair evaluation |
| Create Model | SVC() | Build SVM classifier (linear or nonlinear) |
| Train Model | fit() | Learn best decision boundary between classes |
| Predict | predict() | Test model on unseen data |
| Evaluate | accuracy\_score, report | Measure classification performance |

**CNN**

**🌟 Introduction to Deep Learning Series**

This video gives a **brief overview of Deep Learning**, and explains how it differs from **AI** and **Machine Learning (ML)**.

**🤖 1. Artificial Intelligence (AI)**

* Broad field aimed at building machines that can **perform tasks autonomously**.
* Examples: Netflix recommendations, Amazon shopping suggestions, self-driving cars.

**📊 2. Machine Learning (ML)**

* A **subset of AI** that uses statistical techniques to analyze data and make predictions.
* Used for tasks like **forecasting**, **classification**, **regression**, etc.
* Includes **supervised** and **unsupervised** learning techniques.

**🧠 3. Deep Learning (DL)**

* A **subset of ML** that focuses on mimicking the **human brain**.
* Uses **multi-layered neural networks** to learn from large volumes of data.
* Mainly works with **supervised and unsupervised** approaches.

**🔧 Key Deep Learning Techniques**

**a. Artificial Neural Networks (ANN)**

* Solves **classification** and **regression** problems.

**b. Convolutional Neural Networks (CNN)**

* Used for **image** and **video** data.
* Tasks: **Image classification**, **object detection**, **image segmentation**.
* Tools/Models: **RCNN**, **Mask-RCNN**, **YOLO (v5, v6, v7)**, **Detectron**.

**c. Recurrent Neural Networks (RNN)**

* Designed for **sequential data** like **text** or **time series**.
* Key techniques:
  + **Word Embeddings**
  + **LSTM**, **GRU**, **Bidirectional LSTM**
  + **Encoder-Decoder architectures**
  + **Transformers**, **BERT**

**🛠️ Framework**

* **TensorFlow** (open-source by Google) will be used to implement deep learning models.

Here’s a **summary** of the lecture on **why deep learning is becoming popular**:

**🔍 Context & Evolution**

* Around **2005**, platforms like **Facebook, YouTube, WhatsApp, LinkedIn, Twitter** started gaining popularity.
* These platforms led to an **explosion in data generation** — photos, videos, text, etc.
* By **2011-12**, most major social platforms had launched, and the **volume of data was increasing exponentially**.

**💾 Big Data & Storage**

* The main challenge initially was **how to store this massive data efficiently**.
* This gave rise to **Big Data technologies** like **Hadoop, HBase, Hive**, etc., provided by companies like **Cloudera** and **Hortonworks**.
* These systems could handle **structured and unstructured data**, making data accessible quickly and reliably.

**⚙️ Why Deep Learning Became Popular**

1. **Affordable & Accessible Hardware (GPUs)**:
   * **GPUs**, essential for training deep learning models, became **cheaper and more powerful** due to companies like **NVIDIA**.
   * **Cloud platforms** now offer GPU access at low cost, enabling model training at scale.
2. **Abundant Data Availability**:
   * Deep learning models **perform better with more data**.
   * The surge in data from social media, emails, videos, etc., provides ideal conditions for training DL models.
3. **Widespread Application Across Domains**:
   * Deep learning is used in **medical imaging, e-commerce, retail, marketing**, and many more domains.
   * Its ability to solve complex problems across fields has fueled its growth.
4. **Open Source Frameworks**:
   * Tools like **TensorFlow (by Google)** and **PyTorch (by Facebook)** made deep learning **accessible to everyone**.
   * Strong community support encourages innovation and faster development.

**📊 Performance Comparison with Machine Learning**

* **Machine learning models** plateau in performance as data increases.
* In contrast, **deep learning models improve significantly** with more data, making them better suited for large-scale applications.

**📈 Real-World Examples**

* **Netflix** uses DL to build **recommendation systems**, keeping users engaged and increasing subscriptions.
* **Self-driving cars**, **automated systems**, and **personalized content delivery** are powered by deep learning.

**🔍 What is a Perceptron?**

* A **Perceptron** is the simplest form of a neural network.
* It’s also known as a **single-layer neural network** because it has **only one layer of computation** (excluding the input layer).
* Used primarily for **binary classification** tasks (e.g., predict Pass/Fail, Yes/No, Cat/Dog).

**🧠 Architecture of a Perceptron**

**1. Input Layer**

* Takes in features (like IQ and Study Hours).
* Inputs are denoted as x1,x2,...,xnx\_1, x\_2, ..., x\_nx1​,x2​,...,xn​.

**2. Weights & Bias**

* Each input is multiplied by a weight wiw\_iwi​.
* A **bias (b)** is added to shift the decision boundary.
* This results in a **linear combination**:

z=w1x1+w2x2+…+wnxn+bz = w\_1x\_1 + w\_2x\_2 + \ldots + w\_nx\_n + bz=w1​x1​+w2​x2​+…+wn​xn​+b

or simply:

z=∑i=1nwixi+bz = \sum\_{i=1}^{n} w\_i x\_i + bz=i=1∑n​wi​xi​+b

**3. Activation Function**

* Applies a transformation on zzz to decide the output.
* **Purpose**: To introduce non-linearity and produce a binary output.
* Common choices:
  + **Step Function**:

Output={0if z≤01if z>0\text{Output} = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0 \end{cases}Output={01​if z≤0if z>0​

* + **Sigmoid Function**:

σ(z)=11+e−z\sigma(z) = \frac{1}{1 + e^{-z}}σ(z)=1+e−z1​

Converts any real number into a value between 0 and 1.

**🔁 Training the Perceptron**

**Goal:**

* Learn the correct **weights (w)** and **bias (b)** such that the predicted output matches the actual label.

**Process:**

1. Compute output using current weights and bias.
2. Compare it with actual label to find the **error**.
3. Update weights using the error (via a learning algorithm like the **Perceptron Learning Rule**).
4. Repeat for all data points until error is minimized.

**🧪 Example**

Input features:

* x1x\_1x1​: IQ
* x2x\_2x2​: Study Hours  
  Output: Pass (1) or Fail (0)

| **IQ** | **Study Hours** | **Output** |
| --- | --- | --- |
| 95 | 3 | 0 |
| 110 | 4 | 1 |
| 100 | 5 | 1 |

You feed this data to the Perceptron. If output prediction doesn’t match the actual label, you adjust the weights and bias.

**🧬 Why Bias is Important?**

* Prevents all output from becoming zero when inputs or weights are zero.
* Helps the model generalize better (just like the intercept in linear regression).

**✅ Key Takeaways**

* **Perceptron** = basic unit of an artificial neural network.
* Performs **weighted sum + bias**, then passes through an **activation function**.
* Learns by adjusting weights using error minimization.
* Good for **linearly separable** binary classification tasks.
* Can be extended to **Multi-Layer Perceptrons (MLP)** for more complex problems.

**🔹 Single-Layer Perceptron (SLP)**

**✅ Advantages:**

* Good for solving **linearly separable** problems (e.g., binary classification).
* Simple and intuitive.
* Works by:
  + Assigning weights to inputs.
  + Performing weighted summation + bias.
  + Applying an activation function (e.g., step function) to give output (0 or 1).

**❌ Disadvantages:**

* Can only solve **linearly separable** problems.
* Cannot handle **non-linear data** like XOR.
* Weight update is done through repeated **feedforward and manual adjustments**, which is inefficient.
* No concept of **loss functions, optimizers, or backpropagation** for smart weight updates.

**🔹 Why Multi-Layer Perceptron (MLP) / Artificial Neural Networks (ANN)?**

To overcome SLP limitations, we move to **MLP**, which:

* Can solve **non-linearly separable** problems.
* Consists of **multiple layers** (input, hidden, output).
* Supports **deep learning** with many hidden layers.

**✅ Key Components in MLP:**

1. **Forward Propagation** – Computes predictions layer by layer.
2. **Loss Function** – Measures prediction error (e.g., MSE, cross-entropy).
3. **Backpropagation** – Efficiently updates weights based on error gradients.
4. **Activation Functions** – Introduce non-linearity (ReLU, Sigmoid, Tanh, etc.).
5. **Optimizers** – Algorithms (like SGD, Adam) that fine-tune weights using gradients.

**🔚 Conclusion:**

* **SLP is basic and limited** to simple tasks.
* **MLP enables deeper learning**, handling complex, real-world problems through a systematic learning process using forward propagation, backpropagation, and optimization techniques.

**🔁 What is Forward Propagation?**

Forward propagation is the process by which input data is passed through the layers of a neural network to generate a predicted output.

**🧠 Example Setup**

| **Input Features (X)** | **Meaning** |
| --- | --- |
| X₁ = 95 | IQ of the student |
| X₂ = 4 | Study hours |
| X₃ = 4 | Play hours |

Target Output: 1 (student passed) or 0 (failed)

**🏗️ Network Architecture (2-layer MLP)**

* **Input Layer:** X₁, X₂, X₃
* **Hidden Layer 1:** 1 neuron
* **Hidden Layer 2 (Output Layer):** 1 neuron
* **Weights:** W₁, W₂, W₃, W₄
* **Biases:** B₁ (for hidden layer), B₂ (for output)

**🧮 Forward Propagation Steps**

**🔹 Step 1: Compute Z₁ (Linear combination in hidden layer)**

Z1=X1⋅W1+X2⋅W2+X3⋅W3+B1Z\_1 = X\_1 \cdot W\_1 + X\_2 \cdot W\_2 + X\_3 \cdot W\_3 + B\_1Z1​=X1​⋅W1​+X2​⋅W2​+X3​⋅W3​+B1​

Assume:

* W₁ = 0.01, W₂ = 0.02, W₃ = 0.03
* B₁ = 0.001

Z1=95⋅0.01+4⋅0.02+4⋅0.03+0.001=1.151Z\_1 = 95 \cdot 0.01 + 4 \cdot 0.02 + 4 \cdot 0.03 + 0.001 = 1.151Z1​=95⋅0.01+4⋅0.02+4⋅0.03+0.001=1.151

**🔹 Step 2: Apply Activation Function (Sigmoid)**

A1=σ(Z1)=11+e−Z1=11+e−1.151≈0.759A\_1 = \sigma(Z\_1) = \frac{1}{1 + e^{-Z\_1}} = \frac{1}{1 + e^{-1.151}} \approx 0.759A1​=σ(Z1​)=1+e−Z1​1​=1+e−1.1511​≈0.759

This is the output of the hidden layer neuron.

**🔹 Step 3: Compute Z₂ (input to output layer)**

Z2=A1⋅W4+B2Z\_2 = A\_1 \cdot W\_4 + B\_2Z2​=A1​⋅W4​+B2​

Assume:

* W₄ = 0.02
* B₂ = 0.03

Z2=0.759⋅0.02+0.03=0.04518Z\_2 = 0.759 \cdot 0.02 + 0.03 = 0.04518Z2​=0.759⋅0.02+0.03=0.04518

**🔹 Step 4: Apply Activation Function Again**

A2=σ(Z2)=11+e−0.04518≈0.51129A\_2 = \sigma(Z\_2) = \frac{1}{1 + e^{-0.04518}} \approx 0.51129A2​=σ(Z2​)=1+e−0.045181​≈0.51129

🟢 **Predicted output = 0.511**  
This is a probability. You might threshold it (e.g., ≥ 0.5 → 1, otherwise → 0)

**📌 Summary**

| **Layer** | **Input** | **Weights** | **Bias** | **Z (Linear)** | **A (Activated)** |
| --- | --- | --- | --- | --- | --- |
| Hidden Layer 1 | [95, 4, 4] | [0.01, 0.02, 0.03] | 0.001 | 1.151 | 0.759 |
| Output Layer | [0.759] | 0.02 | 0.03 | 0.04518 | 0.511 |

**🧠 Why Use Activation Functions?**

* They introduce **non-linearity** so the model can learn complex mappings.
* **Sigmoid** squashes values between 0 and 1, useful for binary classification.

**🎥 Deep Dive: Backward Propagation in Neural Networks**

In our previous session, we looked at a **high-level overview** of neural networks — including **forward propagation** and **backward propagation**. Now, let’s dive deeper into **backward propagation** and understand **why it’s crucial**.

**🧠 Quick Recap: Neural Network Structure**

We’re working with a **two-layer neural network**:

* **Input layer**: 3 inputs (let's say X1, X2, X3)
* **Hidden layer**: 2 neurons + bias
* **Output layer**: 1 neuron (binary output)

**Why one output neuron?**  
Because we’re solving a **binary classification** problem — the output is either 0 or 1. For multi-class classification, you would use more neurons (e.g., 3 outputs → 2 or 3 neurons).

**🔁 Forward Propagation Recap**

Here’s what happens in forward propagation:

1. Inputs are multiplied with weights and summed with bias.
2. Activation function is applied.
3. The output from the hidden layer is passed to the output layer.
4. Another set of weights is applied, and final output is calculated.

**Matrix dimensions**:

* Inputs: X1, X2, X3 → shape (1×3)
* Weights (to hidden layer): W1 to W6 → shape (3×2)
* After multiplication: output shape is (1×2) (two neurons in hidden layer)

Then, this output passes through to the output neuron via weights W7 and W8.

**📉 Loss Function**

Once the predicted output is obtained, we compute the **loss**, which is the difference between the actual output and predicted output.

**Loss functions:**

* **Regression**: MSE (Mean Squared Error), MAE (Mean Absolute Error), Huber Loss
* **Classification**: Binary Cross Entropy, Categorical Cross Entropy

For a binary classification task, **Binary Cross Entropy** is commonly used.

**🔁 Backward Propagation: The Core Idea**

Now comes the critical part — **updating the weights** so that the **loss gets minimized**.

**🎯 Objective**

Minimize the **loss** by **adjusting the weights** in the network.

The formula to update any weight is:

Wnew=Wold−α⋅∂L∂WoldW\_{\text{new}} = W\_{\text{old}} - \alpha \cdot \frac{\partial L}{\partial W\_{\text{old}}}Wnew​=Wold​−α⋅∂Wold​∂L​

Where:

* α\alphaα is the **learning rate**
* ∂L∂W\frac{\partial L}{\partial W}∂W∂L​ is the **gradient of the loss w.r.t. weight**

This is called **Gradient Descent** — we move the weights in the direction where the loss decreases.

**📉 Visualizing Gradient Descent**

Imagine a curve (loss function) — our goal is to reach the **lowest point** (called the **global minimum**). At each step, we:

1. Compute the slope of the loss curve.
2. If the slope is:
   * **Negative**: Increase the weight
   * **Positive**: Decrease the weight
3. Keep repeating until we reach the minimum (where slope = 0).

Let’s simplify:

* If your current point on the loss curve gives a **negative slope**, the formula:

Wnew=Wold−α⋅(−value)W\_{\text{new}} = W\_{\text{old}} - \alpha \cdot (-\text{value})Wnew​=Wold​−α⋅(−value)

Results in **W increasing**.

* If slope is **positive**, W **decreases**.

This helps us navigate toward the minimum of the curve.

**⚙️ What Is the Learning Rate?**

The **learning rate α\alphaα** controls:

* **Step size** during weight update
* Too **small**: Training is **slow**
* Too **large**: Can **overshoot** the minimum

It needs to be **just right** — small enough to converge, but not so small that training takes forever.

**🧠 Recap of Weight Update Flow**

Let’s summarize what happens during **backward propagation**:

1. Compute the **loss** between prediction and actual output.
2. Compute **gradients** (partial derivatives) for each weight.
3. Update weights using the gradient descent formula.
4. Repeat until loss is minimized or reaches a threshold.

**🔄 Final General Formula**

For any weight WWW, the update rule is:

Wnew=Wold−α⋅∂Loss∂WoldW\_{\text{new}} = W\_{\text{old}} - \alpha \cdot \frac{\partial \text{Loss}}{\partial W\_{\text{old}}}Wnew​=Wold​−α⋅∂Wold​∂Loss​

And this is how we **train a neural network** by improving it over time through **backpropagation**.

**🧠 1. Weight Update Formula**

*"We know the formula over here... W\_new = W\_old - learning\_rate × dLoss/dW\_old"*

This is the basic formula used in **Gradient Descent**:

* W\_new = updated weight
* W\_old = current weight
* learning\_rate = small value (like 0.01) that controls the size of steps we take to minimize loss
* dLoss/dW\_old = gradient (i.e., how much the loss is affected by a small change in weight)

**🔗 2. Chain Rule of Derivatives**

*"Now let's go ahead and understand what exactly chain rule is..."*

When we have a function inside another function (like a neural network), to compute the derivative we use the **chain rule**:

dLdW=dLdO2⋅dO2dW\frac{dL}{dW} = \frac{dL}{dO\_2} \cdot \frac{dO\_2}{dW}dWdL​=dO2​dL​⋅dWdO2​​

This allows us to break complex derivatives into smaller parts. Why? Because we can't compute dLoss/dW directly when multiple layers are in between.

**🧮 3. Example: Simple Network (Input → Hidden Layer → Output)**

*"This is my input layer, this is my hidden layer (1 neuron), this is my output layer"*

Let’s say:

* Input layer → sends input to hidden layer
* Hidden layer → does computation and applies activation → sends output to output layer
* Output layer → gives final prediction → we compute **loss**

Then we go **backward** (backpropagation) to **update weights**.

**🎯 Updating W4 (Weight from hidden neuron to output neuron)**

*"If I really want to update W4..."*

Use the chain rule:

dLdW4=dLdO2⋅dO2dW4\frac{dL}{dW4} = \frac{dL}{dO2} \cdot \frac{dO2}{dW4}dW4dL​=dO2dL​⋅dW4dO2​

Why? Because:

* Loss depends on the output O2
* O2 depends on weight W4

So:

W4new=W4old−learning rate⋅dLdW4W4\_{\text{new}} = W4\_{\text{old}} - \text{learning rate} \cdot \frac{dL}{dW4}W4new​=W4old​−learning rate⋅dW4dL​

**🎯 Updating W1 (Weight from input to hidden)**

*"Now let's go ahead and try to update W1..."*

This is more complicated. Why?

Because:

* Loss depends on O2
* O2 depends on O1 (output from hidden layer)
* O1 depends on W1

So:

dLdW1=dLdO2⋅dO2dO1⋅dO1dW1\frac{dL}{dW1} = \frac{dL}{dO2} \cdot \frac{dO2}{dO1} \cdot \frac{dO1}{dW1}dW1dL​=dO2dL​⋅dO1dO2​⋅dW1dO1​

Each part is:

* dL/dO2: How the loss changes with output
* dO2/dO1: How the output neuron changes with hidden output
* dO1/dW1: How hidden neuron changes with weight W1

This breakdown is the **chain rule in action**.

**🧠 4. Assignment: Deep Network**

*"Now this is a deeper neural network..."*

Architecture:

* Input layer
* Hidden layer 1 (1 neuron)
* Hidden layer 2 (2 neurons)
* Output layer (1 neuron)

You're asked to update W1.

**🧠 Let’s analyze:**

* Final loss L depends on O3 (output layer neuron)
* O3 depends on both O21 and O22 (hidden layer 2 neurons)
* Each of O21, O22 depends on O11 (output of hidden layer 1)
* O11 depends on W1

So we’ll have **two paths** from L to W1:

**📌 First path (via O21):**

dLdW1=dLdO3⋅dO3dO21⋅dO21dO11⋅dO11dW1\frac{dL}{dW1} = \frac{dL}{dO3} \cdot \frac{dO3}{dO21} \cdot \frac{dO21}{dO11} \cdot \frac{dO11}{dW1}dW1dL​=dO3dL​⋅dO21dO3​⋅dO11dO21​⋅dW1dO11​

**📌 Second path (via O22):**

dLdW1+=dLdO3⋅dO3dO22⋅dO22dO12⋅dO12dW1\frac{dL}{dW1} += \frac{dL}{dO3} \cdot \frac{dO3}{dO22} \cdot \frac{dO22}{dO12} \cdot \frac{dO12}{dW1}dW1dL​+=dO3dL​⋅dO22dO3​⋅dO12dO22​⋅dW1dO12​

Finally:

W1new=W1old−learning rate⋅(sum of both paths)W1\_{\text{new}} = W1\_{\text{old}} - \text{learning rate} \cdot \left( \text{sum of both paths} \right)W1new​=W1old​−learning rate⋅(sum of both paths)

**🎓 Summary**

|  |  |
| --- | --- |
| **Weight Update** | Gradient descent formula: W\_new = W\_old - lr \* dL/dW |
| **Chain Rule** | Used to compute dL/dW when multiple layers are in between |
| **Backpropagation** | Applies chain rule to compute gradients from output to input |
| **Multiple Paths** | In deeper networks, the gradient may have multiple contributing paths |
| **🔁 Overview of Concepts**  **1. Neural Network Setup**   * **Input layer → Hidden Layer 1 → Hidden Layer 2 → Output Layer.** * **Each hidden layer contains only one neuron (for simplicity).** * **Uses sigmoid activation function at each layer.** * **Forward propagation:**   + **z=∑wixi+bz = \sum w\_ix\_i + bz=∑wi​xi​+b**   + **a=sigmoid(z)a = \text{sigmoid}(z)a=sigmoid(z)**     **🧠 Vanishing Gradient Problem**  **Why It Happens:**   * During **backpropagation**, gradients are computed using **chain rule**. * Sigmoid's derivative is always **≤ 0.25**. * As the gradients backpropagate through layers, they are **multiplied** by these small values. * After several layers, the gradients become **very close to zero**. * This means **weights in early layers update very slowly** or **not at all**.   **Chain Rule Breakdown:**  To update w1w\_1w1​, derivative is:  ∂Loss∂w1=∂Loss∂o31⋅∂o31∂o21⋅∂o21∂o11⋅∂o11∂w1\frac{\partial \text{Loss}}{\partial w\_1} = \frac{\partial \text{Loss}}{\partial o\_{31}} \cdot \frac{\partial o\_{31}}{\partial o\_{21}} \cdot \frac{\partial o\_{21}}{\partial o\_{11}} \cdot \frac{\partial o\_{11}}{\partial w\_1}∂w1​∂Loss​=∂o31​∂Loss​⋅∂o21​∂o31​​⋅∂o11​∂o21​​⋅∂w1​∂o11​​  And:  ∂o31∂o21=∂σ(z)∂z⋅∂z∂o21=σ(z)(1−σ(z))⋅w3\frac{\partial o\_{31}}{\partial o\_{21}} = \frac{\partial \sigma(z)}{\partial z} \cdot \frac{\partial z}{\partial o\_{21}} = \sigma(z)(1 - \sigma(z)) \cdot w\_3∂o21​∂o31​​=∂z∂σ(z)​⋅∂o21​∂z​=σ(z)(1−σ(z))⋅w3​  Because:   * σ(z)(1−σ(z))∈(0,0.25)\sigma(z)(1 - \sigma(z)) \in (0, 0.25)σ(z)(1−σ(z))∈(0,0.25) * Repeated multiplication of such small derivatives leads to **vanishing gradients**.   **🚧 Why This Is a Problem**   * In **deep networks**, early layers **learn very slowly**. * Network fails to capture **low-level features** properly. * Makes **training inefficient** or even impossible.    Sigmoid squashes values between 0 and 1.   Its derivative never exceeds 0.25.   When using sigmoid in deep networks, **gradient values shrink during backpropagation**.   This causes **vanishing gradient problem**, especially in early layers.   The root issue is **multiplying several small gradient terms** in chain rule.  Sigmoid activation is good for small neural networks but if we have deep layers like 100, then learning will be slow  **🔹 Basics of Sigmoid Function**   * **“We take inputs, multiply by weights, add bias, and apply an activation function.”** → Standard perceptron operation: z=w⋅x+bz = w \cdot x + bz=w⋅x+b; then pass zzz through an activation function. * **“Sigmoid is responsible for transforming the z value between 0 to 1.”** → Sigmoid maps real-valued input to a range (0, 1):   σ(z)=11+e−z\sigma(z) = \frac{1}{1 + e^{-z}}σ(z)=1+e−z1​   * **“This x is nothing but weights multiplied by the inputs.”** → The input to sigmoid (z) is the weighted sum of inputs + bias.   **🔹 Importance and Usage**   * **“Sigmoid is used after weights and inputs are multiplied and bias is added.”** → It’s applied at the end of the neuron’s forward computation to introduce non-linearity.   **🔹 Derivative and Vanishing Gradient**   * **“The derivative of sigmoid ranges from 0 to 0.25.”** → Maximum value of derivative occurs at z = 0 and is:   σ′(z)=σ(z)(1−σ(z))≤0.25\sigma'(z) = \sigma(z)(1 - \sigma(z)) \leq 0.25σ′(z)=σ(z)(1−σ(z))≤0.25   * **“This leads to vanishing gradient in deep neural networks.”** → As gradients are multiplied layer by layer (via chain rule), they shrink and approach 0. * **“Weights stop updating because gradients become too small.”** → If gradients vanish, weight updates become negligible, causing the network to stop learning.   **🔹 Summary of Vanishing Gradient Issue**   * **“Deep networks suffer as gradients become too small at early layers.”** → Deep architectures especially face this, because each layer’s update relies on the previous.   **🔹 Key Properties of Sigmoid**   1. **“Used frequently in the beginning of deep learning.”** → Historically popular due to simplicity and ability to model probabilities. 2. **“Smooth and easy to derive.”** → Its derivative exists everywhere and is continuous — beneficial for gradient-based methods. 3. **“Output lies between 0 and 1 — often interpreted as probability.”** → Not true probability, but useful in binary classification or last-layer output. 4. **“Firing rate of neuron”** → Sigmoid’s value indicates how "active" the neuron is — close to 0 = off, close to 1 = on.   **🔹 Disadvantages of Sigmoid**  **1. Vanishing Gradient Problem**   * **“When input is far from origin, the gradient becomes almost zero.”** → For large |z|, sigmoid saturates (flattens), and derivative becomes nearly 0. * **“Backpropagation uses chain rule, so gradients vanish.”** → Affects deep networks badly since small gradients multiply and shrink further.   **2. Not Zero-Centered**   * **“Function output is not centered on zero.”** → Sigmoid output ranges from 0 to 1, not -1 to 1. * **“Zero-centered helps in faster convergence.”** → If outputs are centered around 0, positive and negative gradients balance, improving weight updates. * **“For example, standard scaling in ML centers features to mean = 0.”** → Same idea: zero-centered values help models converge better and faster.   **3. Computational Cost**   * **“Sigmoid uses exponential operations which are slower.”** → Calculating e−ze^{-z}e−z is more computationally expensive compared to ReLU (which is just max).   **🔹 Advantages of Sigmoid**   * **Smooth gradient** → No sudden jumps — useful in stable learning. * **Output between 0 and 1** → Great for final layer in binary classification (like logistic regression). * **Thresholding** → You can threshold the output to 0 or 1 using a cutoff (e.g., 0.5).   **🔹 Final Summary**   | **Property** | **Explanation** | | --- | --- | | **Output range** | 0 to 1 | | **Derivative range** | 0 to 0.25 | | **Used in** | Binary classification, early neural nets | | **Pros** | Smooth gradient, interpretable output, historical popularity | | **Cons** | Vanishing gradient, non-zero-centered output, slow due to exponential ops | |  |
|  |  |