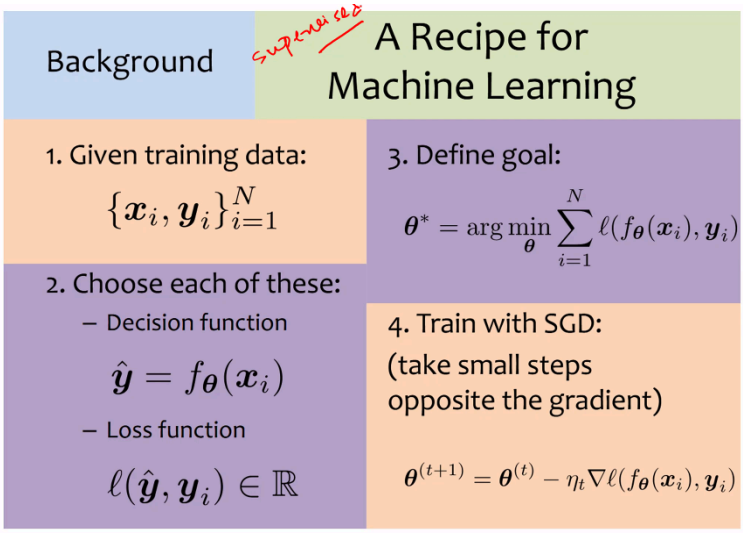
Introduction to Neural networks

A Background of Supervised Machine Learning



✅ TL;DR

The diagram outlines the **four essential steps** of building a supervised machine learning model:

1. Start with labeled training data.
2. Choose a decision function (model) and a loss function.
3. Define the goal as minimizing the average loss.
4. Use Stochastic Gradient Descent (SGD) to iteratively update model parameters in the direction that reduces loss.

🧠 Key Takeaways

1. Training Data (Step 1) – The foundation

- In supervised ML, the **starting point** is a set of labeled examples:

$$\{(x_i, y_i)\}_{i=1}^{N}$$

where:

- x_i (**input features**) could be:
  - Images (for vision tasks)
  - Audio waveforms (for speech tasks)
  - Tabular data (for structured ML tasks)
- y_i (**labels**) could be:
  - A class name or integer (classification)
  - A real-valued number (regression)
  - A structured output (segmentation masks, bounding boxes)

- **Example**:
  - Image of a dog → label: “dog”
  - House characteristics (size, location) → label: price in dollars
- **N** = total number of training examples.
- **Why it matters**:
  - The quality, quantity, and diversity of this data directly affect the model’s ability to generalize to new unseen examples.

---

2. Model (Decision Function) and Loss Function (Step 2) – The design choices

- **Decision Function (f_θ)**:
  - A mathematical function mapping inputs xx to predictions \hat{y}:

    $\hat{y} = f_\theta(x_i)$
  - θ are the **parameters** of the model (weights, biases, etc.).
  - Examples:
    - Linear regression → $f_\theta(x) = w^T x + b$

        - Neural network → Layers of weights + nonlinear activations
- **Loss Function (ℓ)**:
    - Measures how far predictions are from the true label.
    - Returns a **real number** representing the error for a single example.
    - Common examples:
        - Classification: **Cross-Entropy Loss**
        - Regression: **Mean Squared Error (MSE)**
        - Detection: **Localization Loss + Classification Loss**
- **Why it matters**:
    - The choice of model determines what patterns can be learned.
    - The choice of loss determines what "good performance" means mathematically.

---

3. Define Goal (Step 3) – The optimization target

- The objective in supervised ML is to find the **best parameters** θ^* that **minimize** the average loss across all examples:

$$\theta^* = \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \ell(f_\theta(x_i), y_i)$$

- **Key points**:
    - **arg min** means: choose θ that gives the smallest possible average loss.
    - This is **empirical risk minimization** — we use training data to estimate how the model will perform on unseen data.
    - In deep learning, θ can be **millions of parameters**.
- **Why it matters**:
    - Without a clear goal, training becomes directionless.
    - This formalism allows us to systematically improve a model through optimization.

---

4. Training with SGD (Step 4) – The learning process

- **Gradient Descent**:
  - Finds the parameters that minimize loss by iteratively adjusting them in the opposite direction of the gradient:

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta \ell(f_\theta(x_i), y_i)$$

    - η_t = learning rate (controls step size).
    - ∇_θ = derivative of loss with respect to parameters (tells us the direction of steepest increase; we go in the opposite direction).
- **Stochastic Gradient Descent (SGD)**:
  - Instead of computing gradients over the whole dataset (expensive), we compute them using:
    - **Mini-batches** (small random subsets of the data).
  - This introduces noise into updates, which can help escape poor local minima.
- **Why it matters**:
  - SGD is the **engine** that turns data + model + loss into an actual trained system.
  - Variants like **Adam**, **RMSProp**, and **Momentum SGD** can speed up convergence and improve stability.

🔍 How do all the steps connect

1. **Data** is the source of knowledge.
2. **Model** is the learner trying to map inputs to outputs.
3. **Loss function** measures how wrong the model is.
4. **Optimization (SGD)** updates the model to reduce errors.

📚 Important Definitions

- **Supervised Learning**: Learning a mapping from inputs to outputs using labeled examples.
- **Decision Function**: The model's prediction rule.

- **Loss Function**: Numerical measure of prediction error.
- **Gradient Descent**: An Optimization algorithm to minimize loss.
- **SGD**: A faster, noisier variant of gradient descent using subsets of data.

🚀 Actionable Insights / Next Steps

- Recognize that this **recipe** applies to almost any supervised ML task—classification, regression, object detection—only the decision function and loss change.
- For deep learning, f_θ is a neural network and the parameters θ are the weights and biases across layers.
- To go deeper:
  - Study **types of loss functions** and when to use each.
  - Understand **SGD variants** (Momentum, Adam, RMSProp).
  - Practice by implementing this recipe in Python using frameworks like PyTorch or TensorFlow.

Basic Perceptron Model

High-Level Summary

The perceptron is the **simplest type of artificial neural network**, modeling a single neuron that makes decisions by **weighing inputs, summing them up, and applying an activation function** to produce an output.

Detailed Explanation

- **Invented by** Frank Rosenblatt in 1958.
- **Purpose**: Perform **binary classification** (e.g., yes/no, 0/1 decisions).
- **How it works**:
  1. **Inputs** $x_1, x_2, \ldots, x_n$ are each multiplied by a **weight** $w_1, w_2, \ldots, w_n$.
  2. These weighted inputs are **summed** together, plus a **bias** bb:

$$z = w_1x_1 + w_2x_2 + ... + w_nx_n + b$$

3. The result z is passed through an **activation function** (in the original perceptron, a **step function**):

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

4. The output y is the predicted class label.

- **Learning rule** (Perceptron Learning Algorithm):
  - Compare the prediction $\hat{y}$ with the true label y.
  - If wrong, adjust weights:

    $w_j \leftarrow w_j + \eta \cdot (y - \hat{y}) \cdot x_j$

    - η = learning rate (small positive number controlling step size).
    - Repeat over training examples until convergence.
- **Limitation**:
  - Can only classify **linearly separable data** — cannot solve problems like XOR.
  - Solved later with **multi-layer perceptrons (MLPs)** using nonlinear activations.

Analogy

Think of the perceptron like a **weighted voting system**:

- Each input "votes" for a decision, but votes have different importance (weights).
- If the total votes cross a threshold (bias), the perceptron says "YES"; otherwise, "NO".

Mathematical Foundation

1. **Linear combination**:

   $z = \mathbf{w} \cdot \mathbf{x} + b$

2. **Activation function (step)**:

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

3. **Weight update**:

$$w_j \leftarrow w_j + \eta(y - \hat{y})x_j$$

$$b \leftarrow b + \eta(y - \hat{y})$$

Use Case

- **Spam detection**: Inputs = word frequencies, Output = spam (1) or not spam (0).
- **Early image classification**: Handwritten digit recognition for simple datasets.

Multi-Layer Perceptron (MLP)

High-Level Summary

An MLP is a **neural network with multiple layers of perceptrons (neurons)** stacked together, allowing it to learn **nonlinear decision boundaries** and solve problems that a single perceptron cannot.

Detailed Explanation

1️⃣ From Single Perceptron → MLP

- **Problem with single perceptron**:
  - Can only separate data with a **straight line (linear separability)**.
  - Fails for problems like **XOR**.
- **Solution**:
  - Add **hidden layers** between input and output.
  - Introduce **nonlinear activation functions** (ReLU, sigmoid, tanh).
  - This allows the network to **model curves, corners, and complex shapes** in decision boundaries.

2️⃣ Structure of MLP

- **Input Layer**: Takes in features ($x_1, x_2, \ldots, x_n$).
- **Hidden Layers**:
  - Each neuron computes:

$$z_j^{(l)} = \mathbf{w}_j^{(l)} \cdot \mathbf{a}^{(l-1)} + b_j^{(l)}$$

    w_j is the weight and b_j is the bias of the current layer neuron
    - $\mathbf{a}^{(l-1)}$ = outputs from the previous layer.
    - Apply activation: $a_j^{(l)} = \sigma(z_j^{(l)})$
- **Output Layer**: Produces final predictions (classification probabilities, regression outputs).

3️⃣ Why MLPs Are More Powerful

- Hidden layers + nonlinear activations → ability to **approximate any continuous function** (Universal Approximation Theorem).
- Each layer learns increasingly abstract features:
  - **Layer 1**: basic patterns (edges, colors)
  - **Layer 2**: combinations of patterns (shapes, textures)
  - **Layer 3+**: high-level concepts (faces, objects)

4️⃣ Training MLPs — Backpropagation

- Uses **gradient descent** but with an extra step:
  - **Forward pass**: Compute outputs for all layers.
  - **Backward pass**: Compute gradients layer-by-layer using the chain rule.
  - Update parameters using:

$$\leftarrow w - \eta \frac{\partial L}{\partial w}$$

- Backpropagation allows all layers to learn simultaneously.

Analogy

If a single perceptron is like a **yes/no voter**,

then an MLP is like a **hierarchy of committees**:

- First committee looks at raw facts (edges, words).
- Second committee combines those facts into concepts.
- Final committee makes the decision.

Mathematical Foundation

For a 2-layer MLP with one hidden layer:

1. **Hidden layer**:

$$z^{(1)} = W^{(1)}x + b^{(1)}, \quad a^{(1)} = \sigma(z^{(1)})$$

2. **Output layer**:

$$z^{(2)} = W^{(2)}a^{(1)} + b^{(2)}, \quad \hat{y} = \sigma(z^{(2)})$$

3. **Loss** (example: cross-entropy for classification):

$$L = -\sum_{i} y_i \log(\hat{y}_i)$$

4. **Update weights** using backpropagation and gradient descent.

Use Case

- **Image classification** (e.g., MNIST handwritten digits).
- **Speech recognition**.
- **Predicting house prices** from multiple features.

✅ Key Transition to Deep Learning

When you **stack many layers** (sometimes hundreds) and train with large datasets + GPUs → you get **Deep Neural Networks** (DNNs), which form the basis of CNNs (for vision), RNNs (for

sequences), Transformers (for language), and more.

Concept: From Perceptron to MLP to CNN

High-Level Summary

The perceptron started as a **single decision unit**.

The MLP extended it to **multiple fully connected layers**, enabling nonlinear learning.

CNNs specialize MLPs for **image data** by using **convolutions** to capture spatial patterns efficiently.

Step-by-Step Evolution

1️⃣ Perceptron — The Single Neuron

- **Structure**: One layer of inputs connected to one output with weights and a bias.
- **Use**: Simple binary classification.
- **Limitation**: Can only separate linearly separable data.

📌 **Key Idea**: Weighted sum → activation → output.

2️⃣ MLP — Fully Connected Layers

- **Structure**: Multiple layers of perceptrons (neurons) where **every neuron is connected to every neuron in the next layer**.
- **Advantages**:
  - Can model **nonlinear decision boundaries**.
  - Learns complex functions (Universal Approximation).
- **Limitation for Images**:
  - Ignores spatial structure — treats pixels independently.
  - Too many parameters when images are large (e.g., a $224 \times 224 \times 3$ image has ~150k features → huge weight matrices).

📌 **Key Idea**: Deep stacking + nonlinear activations → powerful representation learning.

3️⃣ CNN — Convolutional Neural Network

- **Structure**:
  - **Convolutional layers**: Learn filters/kernels that detect edges, textures, shapes.
  - **Pooling layers**: Downsample features to keep only the most important spatial info.
  - **Fully connected layers** at the end: Combine features for final classification.
- **Advantages**:
  - Exploits **local connectivity**: each filter looks at a small patch (receptive field).
  - **Weight sharing**: Same filter is applied across the whole image → far fewer parameters.
  - Preserves **spatial relationships** between pixels.
- **Result**: Much more efficient and better suited for vision tasks.

📌 **Key Idea**: Detect simple visual patterns in early layers and combine them into complex object representations in deeper layers.

Analogy

- **Perceptron**: A single person making a yes/no decision.
- **MLP**: A group of committees, each learning different aspects, and passing summaries to the next committee.
- **CNN**: Committees that specialize in recognizing **specific shapes/patterns** and apply their knowledge everywhere in the image.

Visual Concept (described)

1. **Perceptron**

```
Inputs → [Weights + Bias] → Activation → Output
```

1. **MLP**

```
Inputs → Hidden Layer → Hidden Layer → Output
(All neurons fully connected between layers)
```

1. **CNN**

> Image → Convolution Filters → Feature Maps → Pooling → More Convolutions → Fully Connected Layers → Output

---

Mathematical Shift

- **MLP Layer**:
  $z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$
- **CNN Layer**:
  $z_{i,j,k} = \sum_{m,n,c} w_{m,n,c,k} \cdot x_{i+m,j+n,c} + b_k$

---

1. Input & Output Coordinates

- **i**: Row index (vertical position) in the **output feature map**.
- **j**: Column index (horizontal position) in the **output feature map**.
- **k**: Channel index in the **output feature map** (one channel per filter).

📌 **Think of i, j** as telling you *where* in the output image you are,

and k as *which filter's output* you're looking at.

---

2. Filter (Kernel) Coordinates

- **m**: Row index inside the filter (small vertical offset).
- **n**: Column index inside the filter (small horizontal offset).
- **c**: Input channel index (e.g., R, G, B for a color image).

📌 **Think of m, n, c** as *where inside the filter* you are when applying it.

---

3. Variables in the Formula

- $x_{i+m,j+n,c}$:

  The input pixel value from position (i+m,j+n) in **input channel c**.
- $w_{m,n,c,k}$:

  The weight at position (m,n) in the filter for **input channel c** and **output channel k**.

- b_k:

  The bias for the output channel k.

Step-by-Step Process

1. Pick an output position (i, j) and a filter k.
2. Look at the corresponding **patch** of the input image.
3. Multiply each pixel in the patch (x) by the corresponding weight in the filter (w).
4. Sum all these products ($\sum_{m,n,c}$).
5. Add bias b_k.
6. Store the result in output position (i, j, k).
7. Slide the filter to the next location and repeat.

Use in Computer Vision

- **Perceptron**: Almost never used directly for images today.
- **MLP**: Sometimes used for small images or after CNN feature extraction.
- **CNN**: Standard for modern image tasks (classification, detection, segmentation).

Wide vs. Deep Neural Networks

High-Level Summary

A **wide network** has **more neurons per layer** (greater width), while a **deep network** has **more layers** stacked on top of each other.

Width increases a model's capacity to memorize features, while depth increases its ability to learn **hierarchical representations**.

Detailed Explanation

1️⃣ Wide Networks

- **Definition**: Layers have many neurons, but there are **few layers** in total.
- **Effect**:
  - Can capture a lot of **parallel patterns** at the same level of abstraction.
  - Good at **memorizing** relationships directly from data.
  - Struggle to represent complex, multi-step transformations.
- **Example**: A single hidden layer with 10,000 neurons.

2️⃣ Deep Networks

- **Definition**: Many layers stacked in sequence, each with a moderate number of neurons.
- **Effect**:
  - Learn **hierarchical features**:
    - Early layers → low-level patterns (edges, colors)
    - Middle layers → intermediate patterns (shapes, textures)
    - Late layers → high-level concepts (faces, objects)
  - Depth enables **compositional learning** — building complex features from simpler ones.
- **Example**: 20+ layers in ResNet.

3️⃣ Trade-offs

Aspect	Wide Network	Deep Network
Computation	More neurons per layer → higher memory use	More layers → longer training time
Feature Learning	Mostly shallow features	Learns hierarchical abstractions
Overfitting Risk	Higher if too wide with small data	Can overfit if too deep without regularization
Training Stability	Usually stable	Risk of vanishing/exploding gradients in very deep nets (solved with ReLU, batch norm, residual connections)

Analogy

- **Wide network**: Like hiring **100 specialists** who each do one step of the job in parallel.

- **Deep network**: Like hiring a **chain of specialists**, where each person builds on the previous person's work.

Use Case

- **Wide networks**: Tabular data with many independent features; shallow but wide models (e.g., Wide & Deep models for recommender systems).
- **Deep networks**: Vision, NLP, speech — tasks where hierarchical feature extraction is key.

Activation Function

High-Level Summary

An activation function is a **mathematical operation applied to a neuron's output** that introduces **non-linearity**, allowing neural networks to learn complex patterns instead of just straight-line relationships.

Detailed Explanation

1️⃣ Why We Need Activation Functions

- Without activation functions, a neural network made of multiple layers would **collapse into a single linear transformation** — no matter how many layers you stack.
- Activation functions allow networks to:
  - Learn **nonlinear decision boundaries**.
  - Represent **complex functions**.
  - Extract **hierarchical features**.

2️⃣ How It Works

- Each neuron computes:

  $$z = \mathbf{w} \cdot \mathbf{x} + b$$

- The activation function σ\sigma is applied:

$$a = \sigma(z)$$

Where:

- z = raw output (weighted sum + bias)
- a = activated output passed to the next layer

3️⃣ Common Activation Functions

Function	Formula	Shape	Pros	Cons
Sigmoid	$\sigma(z) = \frac{1}{1+e^{-z}}$	S-curve	Smooth, output in (0,1), probabilistic interpretation	Vanishing gradients, slow convergence
Tanh	$\tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	S-curve centered at 0	Stronger gradients than sigmoid	Still vanishes for large (z)
ReLU	$f(z) = \max(0, z)$	Linear for positive, 0 for negative	Simple, fast, avoids vanishing gradient for positive values	Can "die" if neurons get stuck at 0
Leaky ReLU	$f(z) = \max(\alpha z, z), \alpha\ small$	Like ReLU with small slope for negatives	Prevents dead neurons	Slightly more computation
Softmax	$f(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$	Probability distribution	Used for multi-class classification	Not for hidden layers

---

Analogy

Think of the activation function like a **gate**:

- It decides **how much of the signal** from a neuron should pass on.
- Without it, all layers just behave like stacked glass sheets — still transparent (linear).
- With it, you can bend, curve, and twist the space to separate complex patterns.

Mathematical Foundation

For a neuron:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

$a = \sigma(z)$

- Without σ : a is linear in x.
- With σ : a can represent complex nonlinear mappings.

Use Case

- **Sigmoid**: Output layer in binary classification.
- **Softmax**: Output layer in multi-class classification.
- **ReLU/Leaky ReLU**: Hidden layers in CNNs and MLPs.

Concept: Choosing the Right Activation Function

High-Level Summary

The choice of activation function depends on **where in the network it's used** (hidden layers vs. output layer), the **task type** (classification, regression), and training considerations (speed, vanishing gradients, interpretability).

Detailed Explanation

🔹 1. Sigmoid

- **When to use**:
  - Output layer for **binary classification** (2 classes).
  - Sometimes, in shallow models, probabilistic interpretation is needed.
- **Why**:
  - Outputs values between **0 and 1**, which can be directly interpreted as probabilities.
- **Avoid in hidden layers**: Causes **vanishing gradients**; slows down learning.
- **Example**: Logistic regression, medical test classification (disease: yes/no).

🔹 2. Tanh

- **When to use**:
  - Hidden layers in **shallow networks** before ReLU became standard.
- **Why**:
  - Outputs in range **(-1, 1)** → zero-centered → helps optimization compared to sigmoid.
- **Avoid deep nets**: They still suffer from vanishing gradients for large inputs.
- **Example**: Early RNNs (before LSTMs/GRUs).

🔹 3. ReLU (Rectified Linear Unit)

- **When to use**:
  - **Default choice** for hidden layers in modern deep networks (CNNs, MLPs).
- **Why**:
  - Computationally cheap, sparse activations (many zeros → efficient).
  - Mitigates vanishing gradient for positive values.
- **Problem**: Can cause "Dying ReLU" — neurons stuck outputting 0 forever.
- **Example**: Most modern CNNs (AlexNet, VGG, ResNet).

🔹 4. Leaky ReLU

- **When to use**:
  - Hidden layers when ReLU neurons are dying.
- **Why**:
  - Small slope for negative inputs ($\alpha \approx 0.01$) keeps gradients alive.
- **Example**: GANs (Generative Adversarial Networks) often use Leaky ReLU in discriminators.

🔹 5. ELU (Exponential Linear Unit)

- **When to use**:
  - Hidden layers when you want **faster, smoother training**.
- **Why**:

- Outputs are closer to **zero mean**, which helps optimization.
- Negative side saturates smoothly instead of being flat.

- **Downside**: Slightly more computationally expensive than ReLU.
- **Example**: Some deep CNNs where batch normalization is not used.

🔹 6. Softmax

- **When to use**:
  - Output layer for **multi-class classification** (mutually exclusive classes).
- **Why**:
  - Converts raw scores into a **probability distribution** over classes.
- **Example**: ImageNet classification (1,000 classes).

🔹 7. Swish

- **When to use**:
  - Hidden layers in **very deep networks** (e.g., >100 layers).
- **Why**:
  - Smooth, non-monotonic → often improves performance over ReLU.
- **Downside**: Slower computation than ReLU.
- **Example**: Used in Google's **EfficientNet** models.

🔹 8. GELU (Gaussian Error Linear Unit)

- **When to use**:
  - Hidden layers in **Transformer-based models** (NLP, vision transformers).
- **Why**:
  - Combines the benefits of ReLU + probabilistic smoothness.
  - More stable in very deep architectures.
- **Example**: BERT, GPT, Vision Transformers (ViTs).

Task-Specific Recommendations

Task	Hidden Layers	Output Layer	Reason
Binary Classification	ReLU / Leaky ReLU	Sigmoid	Sigmoid outputs probability (0–1).
Multi-class Classification	ReLU / Leaky ReLU	Softmax	Softmax gives probability distribution over classes.
Regression (predicting numbers)	ReLU / Tanh	Linear (no activation)	No activation ensures unrestricted real-valued output.
Generative Models (GANs)	Leaky ReLU (Discriminator), ReLU (Generator)	Sigmoid / Tanh depending on output range	GAN stability benefits from Leaky ReLU.
Transformers (NLP, Vision)	GELU / Swish	Depends on task	GELU performs best in very deep nets.

Analogy

- **Sigmoid/Softmax**: Like a **vote counter** that normalizes everything into probabilities.
- **ReLU**: Like a **filter** that ignores negative signals.
- **Leaky ReLU**: Like a filter with a **small leak**, so negative info isn't lost.
- **GELU/Swish**: Like a **smooth dimmer switch** instead of an on/off switch.

Use Case Summary

- Use **ReLU** by default for hidden layers.
- Switch to **Leaky ReLU/ELU** if you face dead neurons or training stalls.
- Use **Sigmoid** (binary) or **Softmax** (multi-class) in the **output layer**.
- For **state-of-the-art NLP/CV models**, consider **GELU** or **Swish**.