Machine Learning Essentials for XR-Driven Robotics

Extended Reality @ NIST

THE UNIVERSITY of ADELAIDE

Feras Dayoub

Australian Institute for Machine Learning

Lecture Outline

- **Why XR + ML in Robotics – motivation & challenges**
- **Core ML Concepts**
  - **Supervised Learning in Robotics**
    - **Neural Networks & CNNs**
    - **RNNs for Sequential Data**
    - **Transformers in Robotics**
  - **Reinforcement learning in Robotics**
- **Conclusion & Future Trends: XR-Powered Robot Learning**

Why Combine Machine Learning + XR in Robotics?

Robotics Challenge	ML Benefit	XR Super-Charge
Real-world is complex and unpredictable	Learn perception & control policies that adapt over time	VR simulation generates unlimited labelled data and lets robots explore safely
Data collection on robots is slow / risky	Imitation & RL need many trials	Teleoperation in VR/MR – an operator "steps inside" the robot to stream expert demos in minutes.
Robots struggle to interpret ambiguous human intent	Deep models map multi-modal inputs to actions	AR overlays show paths, labels giving both robot and human a shared, unambiguous context for perception & commands
Safety & trust in mixed workspaces	Learned policies need transparent reasoning	MR safety fences & planned-path previews make the robot's future actions visible, raising situational awareness and compliance
Sim-to-real gap	Domain randomisation & fine-tuning	XR bridges the gap: train in VR, validate with AR overlays on the real robot, iterate quickly


Isaac Visual Navigation

Core ML Concepts in Robotics

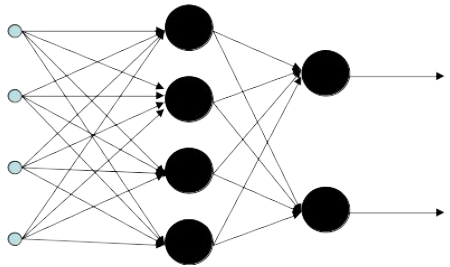
- **Key Learning Paradigms:**
  - Supervised Learning – learning from labeled examples
    - Neural Networks & Deep Learning – models that learn rich representations
    - Transformers & Attention – scalable models for multi-modal input
  - Reinforcement Learning (RL) – learning from trial-and-error
- **Why These Matter in Robotics:**
  - Map sensor data (e.g. images, LiDAR) to actions
  - Interpret uncertain environments using data
  - Enable robots to learn manipulation and navigation policies

Supervised Learning in Robotics

- **What is Supervised Learning?**
  - Learn a mapping from **inputs** (e.g. camera images) to **outputs** (e.g. object labels or motor commands)
  - Requires a dataset of **labeled examples** (input-output pairs)
- **Why It's Useful in Robotics:**
  - Learn to **perceive** (e.g. classify objects, detect obstacles)
  - Learn to **imitate** expert behavior (e.g. behavioral cloning)
- **Key Benefits:**
  - Straightforward to implement
  - Strong performance with good data
  - Often the first step real-world deployment

Neural Networks & Deep Learning

- **What Are Neural Networks?**
  - Inspired by the brain — layers of interconnected “neurons” that learn patterns in data
  - ➡ Input → Processed through layers → Output prediction
- **Deep Learning = Many Layers**
  - Learns **features automatically** from raw data
  - Enables **end-to-end learning** from perception to action
- Key Architectures in Robotics:
  - **Feedforward Networks** – simple perception or control
  - **CNNs (Convolutional NNs)** – image-based perception (e.g. object detection)
  - **RNNs/LSTMs** – time-series or sequential data
  - **Transformers** – powerful for multimodal input (vision + language)
- Why It Matters in Robotics?
  - Handles noisy, high-dimensional sensor input
  - Enables complex behaviors (e.g. grasping, driving)
  - Forms the core of modern robot learning systems



Feedforward Networks

Why use NN?

Robots operate in noisy environments. Sensors give us partial information. Instead of handcrafting mappings from input to action, we can **learn** them from **data**.

A feedforward network learns the mapping:

$$f_\theta(x) = \hat{y}$$

Where:

- $x \in \mathbb{R}^n$: input vector (e.g., 3 sensor readings),
- $\hat{y} \in \mathbb{R}^m$: output vector (e.g., motor command or action label),
- $\theta = \{W^{(l)}, b^{(l)}\}$: parameters (weights and biases),
- f_θ: composed of linear + nonlinear layers.

A 2-layer FFNN Example:

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

With:

- $\sigma(\cdot)$: nonlinearity (e.g., ReLU, tanh),
- $\hat{y}$: predicted action (e.g., one-hot class, or continuous command).

A Toy Example: Wall-Following Robot

Scenario

- 3 ultrasonic sensors: $x_1 =$ left, $x_2 =$ front, $x_3 =$ right.
- Desired output: turn left, go straight, or turn right.

Input:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \in \mathbb{R}^3$$

Output: One-hot vector:

$$\hat{y} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \quad \text{(turn left)}$$

A Toy Example: Wall-Following Robot

Say:

- Input: $n = 3$,
- Hidden layer: $h = 5$,
- Output: $m = 3$.

Then:

- Parameters in layer 1: $W^{(1)} \in \mathbb{R}^{5\times 3}$, $b^{(1)} \in \mathbb{R}^5$,
- Parameters in layer 2: $W^{(2)} \in \mathbb{R}^{3\times 5}$, $b^{(2)} \in \mathbb{R}^3$.

Total: 38 parameters.

Data Collection

Use supervised learning:

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

Sources:

- **Teleoperation:** human controls robot, log x and y.
- **Simulation:** generate synthetic labeled episodes (e.g., Gazebo).
- **Manual labeling** (for small-scale tasks): record sensors + annotate desired action.

Loss Function - Cross-Entropy for Classification

To train our network, we minimize a loss function. For classification tasks, we commonly use:

$$\mathcal{L}(\theta) = -\sum_{i=1}^{N}\sum_{j=1}^{m} y_{ij} \log \hat{y}_{ij}$$

Where:

- $\mathcal{L}(\theta)$: The total loss for all data points, dependent on parameters θ.
- N: Total number of training examples.
- m: Number of output classes (e.g., 3 for left/straight/right).
- y_{ij}: The true label for the i-th example and j-th class. It's 1 if the i-th example belongs to class j, and 0 otherwise (one-hot encoding).
- $\hat{y}_{ij}$: The predicted probability that the i-th example belongs to class j.
- log: Natural logarithm.

The predicted probability $\hat{y}_{ij}$ is obtained using the **softmax** activation function on the raw output of the network $f_\theta(x_i)$:

$$\hat{y}_{ij} = \text{softmax}(f_\theta(x_i))_j$$

Optimisation Basics: Gradient Descent

To train the network, we iteratively adjust the parameters θ to minimize the loss function $\mathcal{L}(\theta)$.

Core Idea: Gradient Descent

- We want to find the "bottom" of the loss landscape.
- The **gradient** ($\nabla_\theta \mathcal{L}(\theta)$) points in the direction of the steepest **increase** of the loss.
- So, we move in the **opposite** direction to decrease the loss.

Stochastic Gradient Descent (SGD) Update Rule:

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(\theta)$$

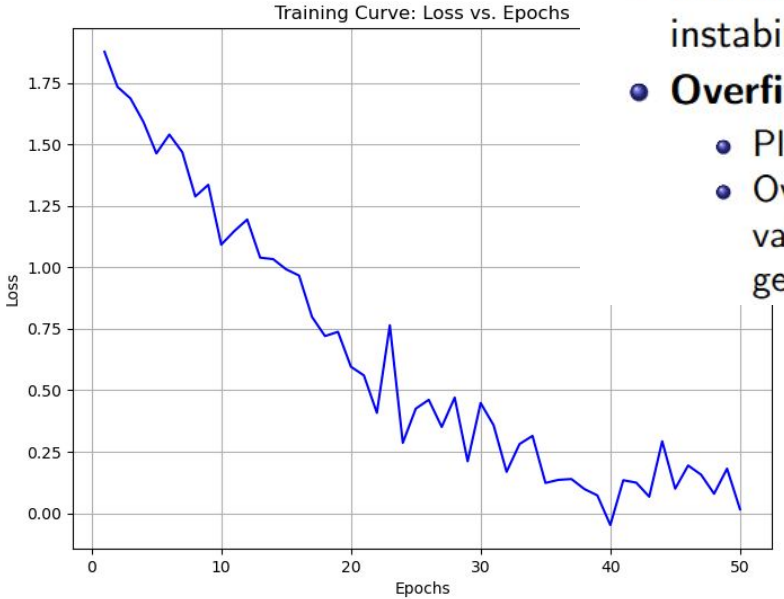
Where:

- θ: The model parameters (weights and biases).
- η: The **learning rate** (step size). A crucial hyperparameter.
- $\nabla_\theta \mathcal{L}(\theta)$: The **gradient** of the loss with respect to parameters.

The Training Loop

1. Initialize model parameters randomly.
2. Iterate for a specified number of **epochs** (full passes over the dataset).
3. Within each epoch, iterate through the training data in **mini-batches**.
4. For each mini-batch:
   - **Forward Pass:** Compute the network's output ($\hat{y}$) for the current mini-batch.
   - **Compute Loss:** Evaluate the loss function $\mathcal{L}(\theta)$ based on $\hat{y}$ and true labels.
   - **Backward Pass (Backpropagation):** Calculate the gradients $\nabla_\theta \mathcal{L}(\theta)$.
   - **Parameter Update:** Adjust the parameters θ using an optimizer.

Evaluation Metrics - raining Curve (Loss vs. Epochs)



What to Look For:

- **Decreasing Loss:** Indicates the model is learning and parameters are optimizing.
- **Flat Loss:** Could mean convergence, or the learning rate is too low.
- **Increasing Loss:** Suggests problems like a too-high learning rate or instability.
- **Overfitting Detection:**
  - Plot both **training loss** and **validation loss**.
  - Overfitting occurs when training loss continues to decrease, but validation loss starts to increase (model memorizing training data, not generalizing).

Evaluation Metrics

After training, we assess our model's performance on unseen data.

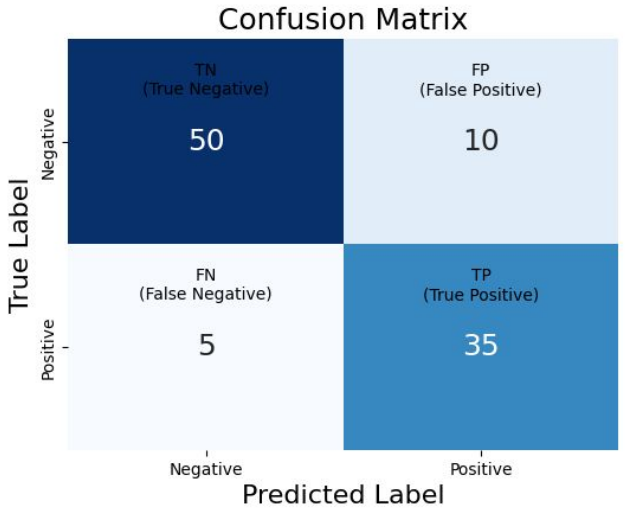
1. **Accuracy** Measures the proportion of correct predictions:

$$\text{Accuracy} = \frac{\#\text{ correct predictions}}{N_{\text{test}}}$$

Where N_{test} is the total number of examples in the test set.

- Simple and intuitive for balanced datasets.
- Can be misleading for imbalanced datasets (e.g., in rare disease detection).

2. **Confusion Matrix**
   - A table summarizing classification model performance.
   - Shows counts of:
     - **True Positives (TP):** Correctly predicted positive.
     - **True Negatives (TN):** Correctly predicted negative.
     - **False Positives (FP):** Incorrectly predicted positive (Type I error).
     - **False Negatives (FN):** Incorrectly predicted negative (Type II error).



Why Move Beyond FFNNs?

FFNNs work well on low-dimensional inputs (like 3 sonar readings). But, what happens when the robot receives a **camera image**?

Let's calculate:

- 64 X 64 x 3 = 12,288 inputs
- First hidden layer with 100 units → **1.2 Million parameters!**

Consequences:

- Training becomes extremely inefficient and slow.
- Overfitting risk is significantly higher due to the massive number of parameters

Solution: Exploit spatial locality and shared weights

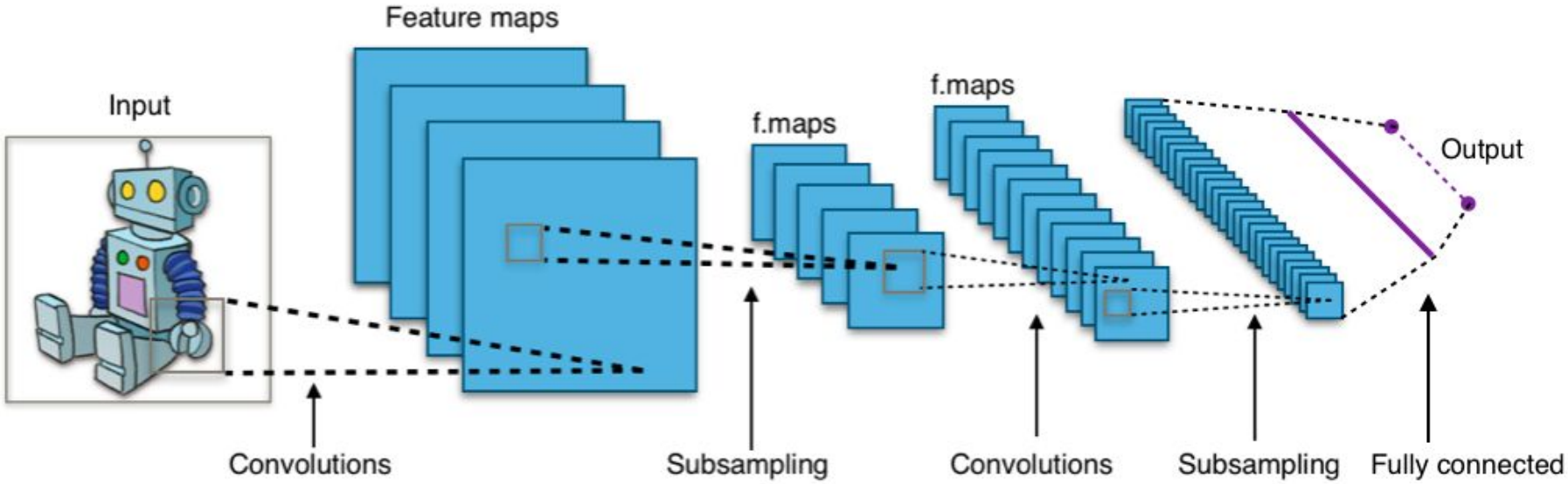
Convolutional Neural Networks (CNNs)

CNNs: Efficient and scalable architectures designed for learning from high-dimensional, spatially structured data like images. They are critical for visual perception tasks in **robotics**.

- **Key Advantages for Image Data:**
  - **Shared Weights (Filters/Kernels):** A small filter (e.g., 3×3) is convolved across the image using shared weights, enabling the network to learn position-invariant features like edges and textures.
  - **Preserving Spatial Structure:** Unlike FFNNs, which flatten images, CNNs process them as 2D or 3D tensors, preserving spatial relationships crucial for recognizing visual patterns. Pooling layers further summarize this spatial information.
  - **Drastically Reduced Parameter Count:** Weight sharing in CNNs reduces the number of learnable parameters compared to fully connected layers, helping prevent overfitting, **even in deep networks**.

CNN Architecture Overview

- **Convolutional Layer:**
  - Applies **learned filters** to detect local patterns (e.g., edges, textures).
  - Produces **feature maps**.
- **Activation Function:**
  - Adds **non-linearity** after convolutions.
  - Enables learning of **complex patterns**. Common choices: ReLU, Tanh, Sigmoid.
- **Pooling Layer:**
  - **Downsamples** feature maps to reduce size and overfitting.
  - Adds **spatial invariance**. Common types: Max, Average.
- **Fully Connected Head:**
  - Flattens features and performs **final classification or regression.**
  - Encodes high-level **reasoning**.


Input
Feature maps
f.maps
f.maps
Output
Convolutions
Subsampling
Convolutions
Subsampling
Fully connected

Convolution Operation

Convolution is the core operation in CNNs, where a small filter slides over the input to produce feature maps.

Inputs:

- Input $X \in \mathbb{R}^{H \times W \times C}$: height H, width W, channels C (e.g., RGB has $C = 3$)
- Filter $K \in \mathbb{R}^{k \times k \times C}$: spatial size $k \times k$, matching input channels

Convolution Output at (i, j):

$$Y_{i,j} = \sum_{u=1}^{k} \sum_{v=1}^{k} \sum_{c=1}^{C} K_{u,v,c} \cdot X_{i+u,j+v,c}$$

- Computes a weighted sum over a local region of the input.
- The weights are defined by the filter K.

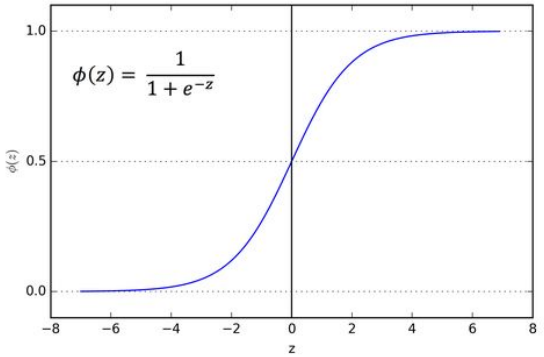
Say:
16 filters of size 5×5×3, then:
16 x (5 x5 x 3) = **1,200** compared to over **1 million** parameters in FFNN

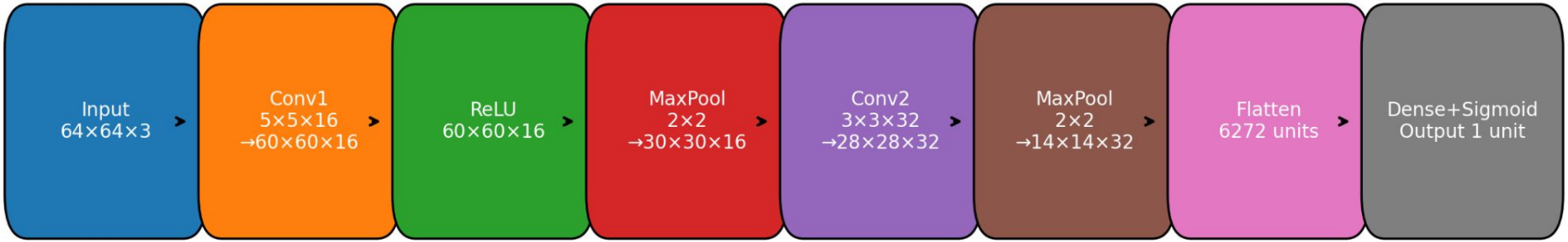
Toy Example: Object Detector

- **Task:**
  - A robot sees a c**luttered tabletop** via its camera.
  - It must detect the **red cube** to pick it up.
  - This is a **binary classification** problem: Is the red cube present in the image?
- **Input for our CNN:**
  - A 64 × 64 × 3 image from the robot's camera.
- **Output from our CNN**
  - A Binary Class prediction
    - Red cube present" (Yes / 1)
    - Red cube not present" (No / 0)
  - This would typically be a **single output neuron** with a **sigmoid activation function**, indicating the probability of the red cube being present.

Architecture Sketch

- Task: Binary classification to detect a red cube
- Input: 64×64×3 RGB image
- Output: score via sigmoid





In PyTorch

```
RedCubeCNN(

  (conv1): Conv2d(3, 16, kernel_size=(5, 5),
stride=(1, 1))

  (relu1): ReLU()

  (pool1): MaxPool2d(kernel_size=2, stride=2,
padding=0, dilation=1, ceil_mode=False)

  (conv2): Conv2d(16, 32, kernel_size=(3, 3),
stride=(1, 1))

  (pool2): MaxPool2d(kernel_size=2, stride=2,
padding=0, dilation=1, ceil_mode=False)

  (flatten): Flatten(start_dim=1, end_dim=-1)

  (fc): Linear(in_features=6272, out_features=1,
bias=True)

  (sigmoid): Sigmoid()

)

Output shape: torch.Size([1, 1])
```

```python
import torch
import torch.nn as nn

class RedCubeCNN(nn.Module):
    def __init__(self):
        super(RedCubeCNN, self).__init__()
        # Convolution + ReLU + Pooling layers
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=5, stride=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Compute flatten size: after conv+pool -> 14×14×32 = 6272
        self.flatten = nn.Flatten()

        # Fully connected layer + Sigmoid
        self.fc = nn.Linear(14 * 14 * 32, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.pool1(self.relu1(self.conv1(x)))   # Conv1 + ReLU + Pool1
        x = self.pool2(self.conv2(x))               # Conv2 + Pool2
        x = self.flatten(x)                         # Flatten to vector
        x = self.fc(x)                              # Fully connected
        x = self.sigmoid(x)                         # Sigmoid activation
        return x

# Example usage
if __name__ == "__main__":
    model = RedCubeCNN()
    print(model)

    # Test with a dummy input
    sample_input = torch.randn(1, 3, 64, 64)  # batch size 1, RGB 64×64
    output = model(sample_input)
    print("Output shape:", output.shape)  # should be [1, 1]
```

Data Collection for Vision Tasks

- ## Primary Sources and Methods:
  - Use object detection datasets (**COCO**) or annotate your own (**LabelImg**).
  - Use simulation (**Gazebo, Isaac Sim**) to generate scenes with labels.

- ## Best Practices for Robustness:
  - **Diverse Views:** Capture objects from multiple camera angles.
  - **Varying Lighting Conditions:** Include different ambient light,
  - shadows, and reflections.
  - **Clutter and Occlusion:** Ensure data includes objects in cluttered scenes and partially occluded.
  - **Background Variety:** Train on different backgrounds to prevent the model from memorizing specific environments.

Training - Binary Cross-Entropy Loss

$$L = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i)\right]$$

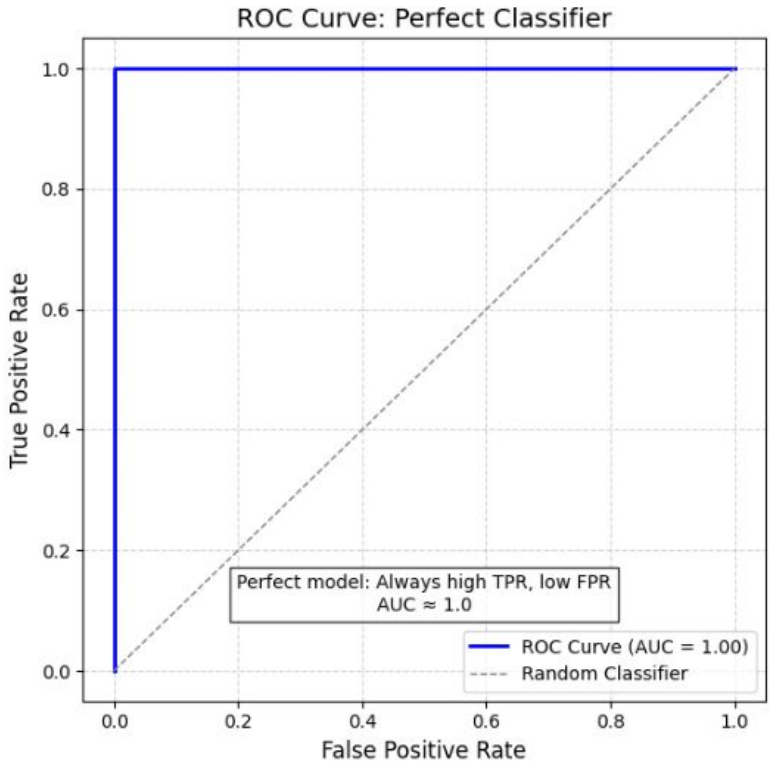
$$\hat{y}_i = \sigma(x_i) \quad \text{with} \quad \sigma(x) = \frac{1}{1 + e^{-x}}$$

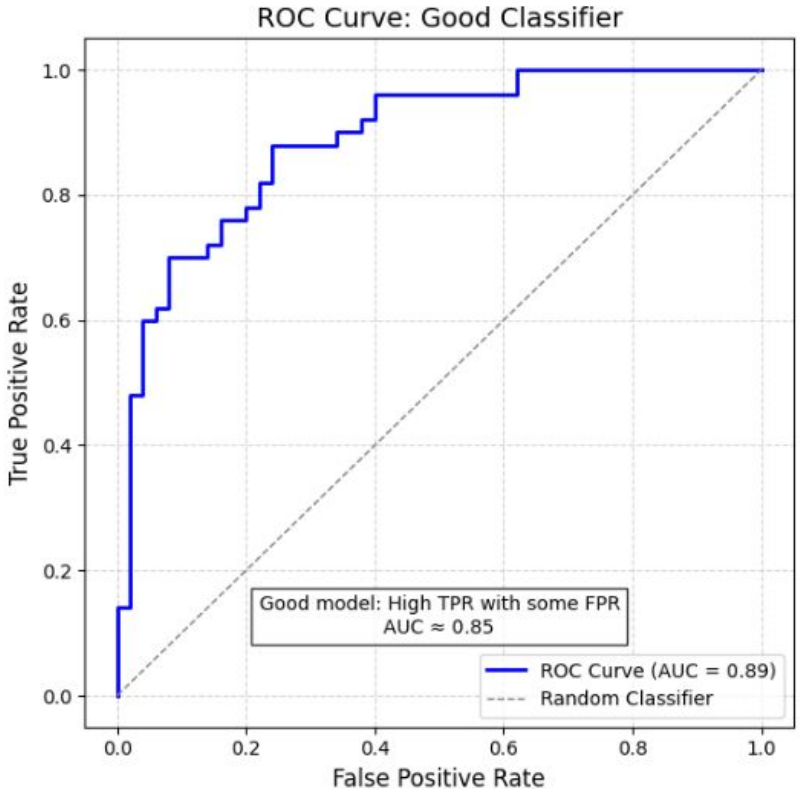
- $y_i \in \{0, 1\}$
- If $y_i = 1$: loss $= -\log \hat{y}_i$; if $y_i = 0$: loss $= -\log(1 - \hat{y}_i)$
- Heavily penalises confident errors

Evaluation Metrics

- **Accuracy = (TP + TN) / (TP + TN + FP + FN)**
- **Precision (Positive Predictive Value) = TP / (TP + FP)**
  - Of all instances predicted as positive (red cube present), what proportion were actually positive?
- **Recall (Sensitivity / True Positive Rate) = TP / (TP + FN)**
  - Of all actual positive instances (true red cubes), what proportion were correctly detected?
- **F1-Score = 2 × (Precision × Recall) / (Precision + Recall)**
  - Provides a single score that balances both precision and recall, especially useful for imbalanced datasets
- **ROC Curve and AUC (Area Under Curve)**
  - Plots the true positive rate (recall) against the false positive rate at various threshold settings.
  - AUC: ranging from 0 to 1, with higher values indicating better overall discrimination across all thresholds.

ROC Curve and AUC (Area Under Curve)

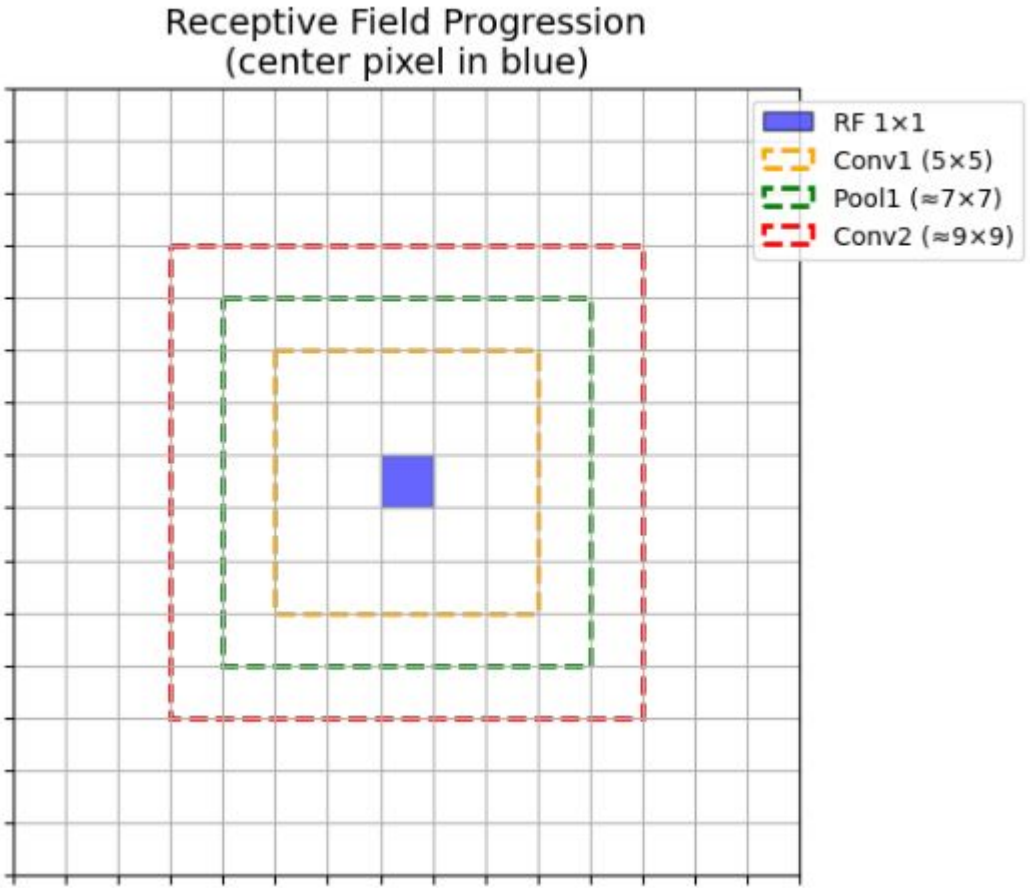




Receptive Field in CNNs

Receptive Field (RF): Number of pixels a neuron "sees" in the image.

- Early layers see only tiny, local patches.
- Deeper layers integrate information over a wider area.
- Pooling and larger strides enlarge RFs, making outputs more robust to small shifts or noise, but precise localization can suffer



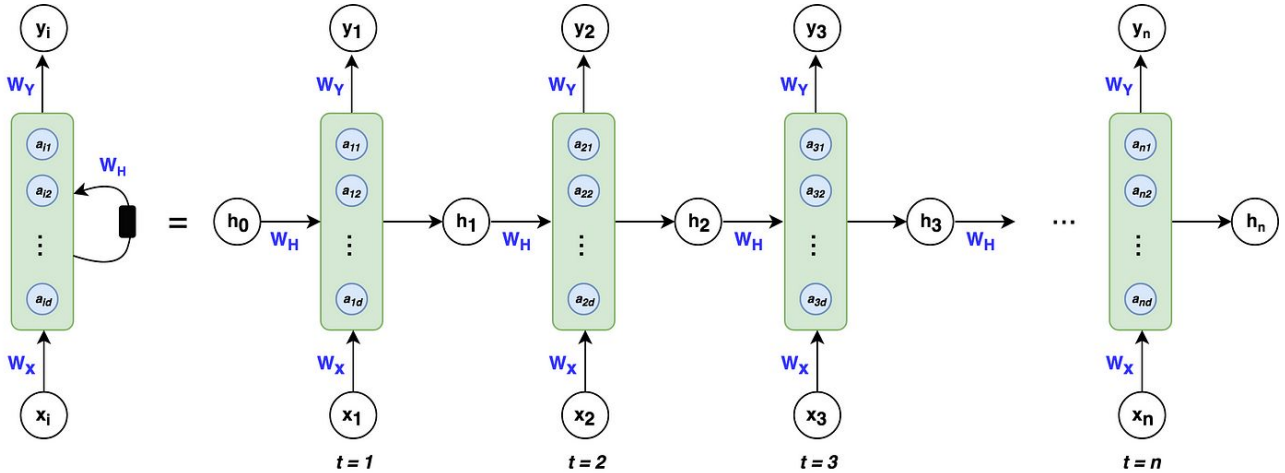
Why Move Beyond CNNs?

- CNNs are great at seeing, but **they don’t know about time**.
- What if you want your robot to understand a spoken command, or track an object as it moves?
- To model sequences, we need memory.

RNN Basics: Core Architecture

Unlike Feedforward Networks (FFNNs) or CNNs, Recurrent Neural Networks (RNNs) are specifically designed to process sequential data.

- **Processing Sequences:**
  - RNNs handle inputs as **sequences**: x1, x2, . . . , xn .
  - At each time step t, the model t**akes both** the c**urrent input xi** and the information from the **previous time step** (encoded in a hidden state) into account.



RNN Basics: Key Concept

The Recurrent Equations: At each time step t, the model updates a **hidden state** h_t (its "memory") based on the current input and the previous hidden state:

$$h_t = \tanh(W_x x_t + W_h h_{t-1} + b)$$

And produces a predicted output $\hat{y}_t$:

$$\hat{y}_t = \text{softmax}(W_y h_t + c)$$

Where:

- $x_t \in \mathbb{R}^n$: Input vector at time t.
- $h_t \in \mathbb{R}^d$: Hidden state vector at time t.
- $\hat{y}_t \in \mathbb{R}^m$: Predicted output vector at time t (e.g., a class probability distribution, a control signal).
- W_x, W_h, W_y, b, c: Learnable parameters (weight matrices and bias vectors). These are **shared across all time steps**.
- $\tanh(\cdot)$: A common non-linear activation function.

Toy Example: Collision Anticipation

Task:

- A robot drives forward using continuous sonar readings.
- It must predict if a **collision is expected in the next second** based on its recent measurement history.

Input for our RNN:

- A **sequence of sonar readings** over time.
- Example: $x_1, x_2, \ldots, x_5$
  - Each x_t represents the distance reading at time t.

Output from our RNN:

- **Binary Classification** at each time step t:

$$\hat{y}_t = \begin{cases} 1 & \text{if collision expected} \\ 0 & \text{otherwise} \end{cases}$$

- The network learns to infer future events from past observations.

Network Architecture

Input Layer:

- **Sequence of Inputs:** $x_1, x_2, \ldots, x_T$
- **Sequence Length (T):** $T = 5$ (meaning we consider the last 5 sonar readings).
- Each x_t is a single scalar (distance reading).

Recurrent Layer (Core of the Network):

- **Hidden State Size:** $d = 32$
  - This determines the dimensionality of the internal "memory" of the network at each time step.

Output Layer:

- The hidden state h_T (or h_t at each step, depending on the prediction strategy) is fed into a final layer.
- **Final Output:** A single neuron with a **Sigmoid activation function**.
- **Prediction:** Represents the likelihood of a collision (1 for collision expected, 0 otherwise).

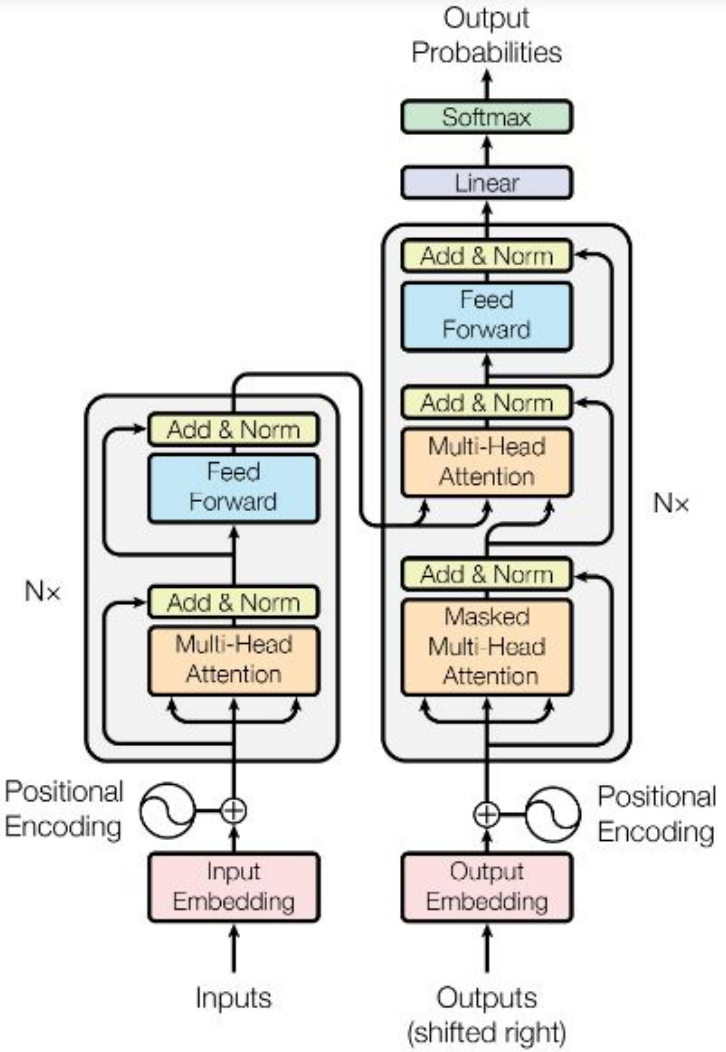
```python
import torch
import torch.nn as nn

# Define the RNN model for collision anticipation
class CollisionAnticipationRNN(nn.Module):
    def __init__(self, input_size=1, hidden_size=32, output_size=1):
        super(CollisionAnticipationRNN, self).__init__()
        # Input size: 1 (scalar sonar reading)
        # Hidden size: 32 (dimensionality of hidden state)
        # Output size: 1 (probability of collision)
        self.hidden_size = hidden_size

        # RNN layer: processes sequence of scalar inputs
        self.rnn = nn.RNN(input_size=input_size, hidden_size=hidden_size, batch_first=True)

        # Fully connected layer: maps final hidden state to output
        self.fc = nn.Linear(hidden_size, output_size)

        # Sigmoid activation for binary classification (collision probability)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # x shape: (batch_size, seq_len, input_size) = (batch_size, 5, 1)
        # Initialize hidden state: (num_layers, batch_size, hidden_size)
        batch_size = x.size(0)
        h0 = torch.zeros(1, batch_size, self.hidden_size).to(x.device)

        # Pass through RNN
        # out: (batch_size, seq_len, hidden_size), contains hidden states for all time steps
        # h_n: (num_layers, batch_size, hidden_size), final hidden state
        out, h_n = self.rnn(x, h0)

        # Use the final hidden state (h_T) for prediction
        # h_n shape: (1, batch_size, hidden_size)
        out = self.fc(h_n[-1])  # Take the last layer's hidden state

        # Apply sigmoid to get collision probability
        out = self.sigmoid(out)  # Shape: (batch_size, 1)
        return out
```

Why Move Beyond RNNs?

RNNs gave us memory, but are slow, hard to train, and struggles with long-term dependencies.

Transition to Transformers

- Using **attention** to dynamically focus on relevant input, no recurrence.
- Processing **all inputs in parallel**, enabling scale.
- Attractive to robotics as it **unify language**, **vision**, and **action** via attention and **tokenization**.



Toy Example

Scene: The robot sees a table with several objects:

- Red cube
- **Blue** ball
- Green cylinder

The goal is simple: **"Pick up the red cube."**

Representing the Objects (Embeddings)

Each object on the table is converted into a **vector** that represents its **features**, such as color, shape, and size. These vectors are called **embeddings**.

- Red cube → a vector like [0.5, -0.2, 0.8, ...]
- Blue ball → [0.1, 0.4, -0.3, ...]
- Green cylinder → [-0.6, 0.7, 0.2, ...]
- **Instruction**: “Pick up the red cube” is tokenized into words or subwords (e.g., “Pick”, “up”, “the”, “red”, “cube”), each converted to an embedding, like [0.3, 0.9, -0.1, ...] for “Pick.”

A single sequence might look like: [INST] Pick up the red cube **[OBJ]** red_cube **[OBJ]** blue_ball **[OBJ]** green_cylinder **[ACT]**,

where [INST], [OBJ], and [ACT] are special tokens to differentiate instruction, objects, and the action prompt.

Adding Positional Information (Positional Encoding)

Since transformers **process all objects at once**, we need to tell the model where each object is in the sequence. This is done by adding **positional encodings** to the embeddings.

- Red cube (position 1) → embedding + positional encoding for **position 1**
- Blue ball (position 2) → embedding + positional encoding for **position 2**
- Green cylinder (position 3) → embedding + positional encoding for **position 3**
- Instruction tokens (e.g., “Pick” at **position 4**, “up” at **position 5**, etc.) → each gets its own positional encoding.

Note:

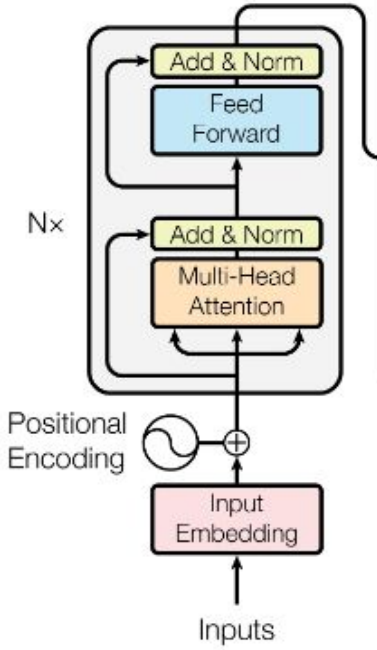
In *set*-structured inputs (unordered collections of object features) we don’t want absolute positions 1, 2, 3. We often use **learned “object-centric” tokens**, **relative position encodings**, or even **permutation-invariant set** transformers.

Processing the Scene (Encoder with Self-Attentior

The transformer's **encoder** processes the entire input sequence using **self-attention**. Each embedding attends to all others, allowing the model to capture relationships between the instruction and objects.

- The embedding for "red" in the instruction attends strongly to the "red_cube" object embedding, reinforcing their connection.

- The "red_cube" embedding attends to other objects (**blue** ball, green cylinder) to strengthen differences (e.g., color, shape).



How attention works [Attention = Learnable Relevance

For each embedding, the transformer computes:

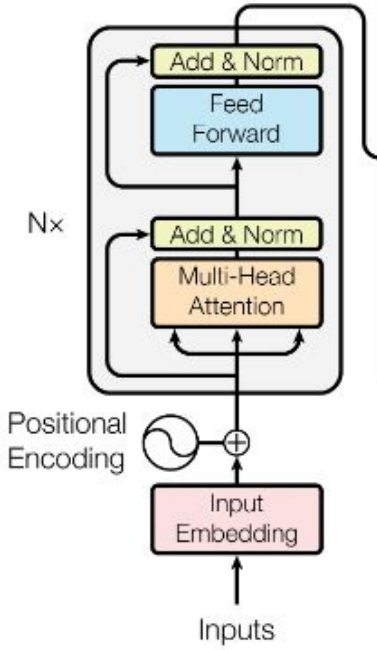
- **Query (Q) – what we're looking for**
- **Key (K) – what each element offers**
- **Value (V) – the information to pass forward**

Attention score = similarity (e.g., **dot products**) between Q and K
Output = weighted sum over V (based on the scores)

Multiple attention heads capture different relationships (e.g., color vs. shape).

Result:

The encoder outputs enhanced representations that integrate the scene and instruction, capturing their contextual relationships.



How attention works [Attention = Learnable Relevance Scoring]

Query, Key, Value (QKV): For each input embedding x (or X for the whole sequence), three linear projections are made:

- $Q = XW^Q$: **Query** vector (What I'm looking for).
- $K = XW^K$: **Key** vector (What I offer).
- $V = XW^V$: **Value** vector (Information I contain).

(W^Q, W^K, W^V are learnable weight matrices.)

Attention Calculation: The output of attention is a weighted sum of Value vectors, where weights are determined by the similarity of Queries and Keys:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

- $\sqrt{d_k}$: Scaling factor to prevent large dot products.
- This enables long-range dependencies and parallel computation across the sequence.

Focusing on the Target (Decoder with Cross-Attention)

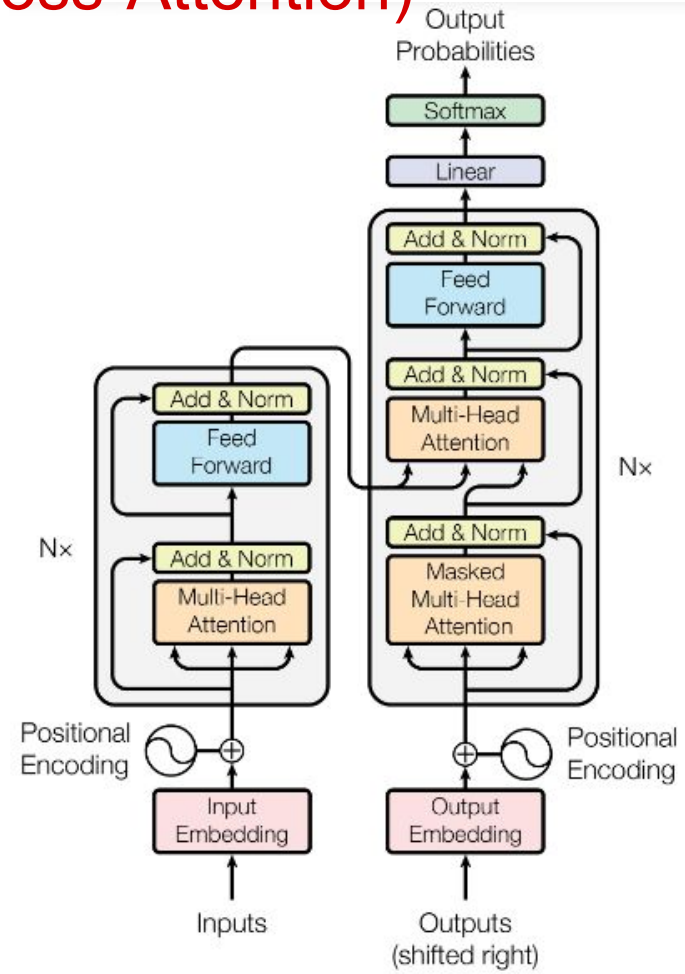
Guide the model to **generate actions** that are relevant to the instruction and the scene (e.g., focus on “pick up the red cube”).

- The decoder is given a partial output sequence (starting with [ACT]) and is tasked to predict the next token.
- It uses **cross-attention** to attend to the encoded Instruction and Scene features.

In our toy example

Input: "Pick up the red cube" + scene with colored blocks

Decoder output tokens: `[ACT] grasp move-up move-back stop`



Generating the Action (Output)

Autoregressive Token Generation

- The decoder generates one action token at a time, each conditioned on:
  - The full scene and instruction (via cross-attention), and
  - All previously generated action tokens .

Action Token Vocabulary

Encoding Style	What a Token Means
Low-level (quantised)	**Discrete version of velocity or joint commands for the next timestep**
Symbolic (semantic)	**Each token indexes an action embedding (e.g. grasp, move up, rotate 15°, etc.)**

Training a Transformer Policy for Robotics

Objective: A model to map from **scenes + instructions** to valid **action sequences**.

Training Process (Behavior Cloning)

- **Data Collection:** Gather input-output pairs (e.g., via teleoperation, simulation, or manual labeling).
- Predict next action token conditioned on ground-truth prior tokens.

Dataset Requirements

- Diverse scenes (objects, lighting, occlusion).
- Time-aligned action labels (e.g 14 joint positions, base velocities) per frame.
- Consistent action representation across demonstrations.

Reinforcement learning (RL)

What is Reinforcement Learning?

- A type of machine learning where an **agent** learns to make decisions by performing actions in an **environment**.
- The agent receives **rewards** or penalties based on its actions.
- The goal is to learn a **policy** that maximizes the cumulative reward over time.

Learning by Trial and Error:

- Unlike supervised learning, RL learns from its own experiences.
- The agent **discovers** the optimal behavior **without explicit** programming for every possible situation.

Why RL Is Still Hard in Robotics

While Reinforcement Learning holds **great promise**, its practical application in robotics faces **significant challenges**.

- Requires an **extensive amount** of interaction data (trials and errors) to learn effective policies.
- **Collecting** this data in the real world is often time-consuming, **expensive**, and can cause wear and tear on robots.
- **Adapting** to unpredictable and constantly changing real-world environments is **difficult**.
- Policies learned in one setting may **not generalize** well to others.
- **RL alone** often **struggles** with tasks demanding **high precision**, repeatability, and guaranteed safety constraints.

Why RL Is Still Hard in Robotics — and How XR Moves (but Doesn't Remove) the Obstacles

Core Limitation of RL	How XR Helps	What Still Remains
Sample inefficiency – millions of interactions needed, impractical on real hardware	High-fidelity VR simulation lets you fast-forward time and parallelise thousands of robots	Sim compute bills grow fast; simulator ≠ reality
Damage & safety risk – exploratory actions can break robots or endanger people	MR safety fences and "ghost preview" allow safe human veto before execution	Human-in-the-loop still slows learning; unexpected contacts can still occur
Reward design is brittle – sparse or poorly shaped rewards stall learning	VR teleoperation demos bootstrap with behaviour cloning; AR reward markers visualise goals	Dense rewards can still lead to reward hacking; annotating every new task is labour-intensive
Reality gap – policies over-fit simulator quirks	Domain randomisation + AR validation	Sub-millimetre physics, sensor latency, and contacts remain hard to model
Generalisation – trained policy locks onto one embodiment or workspace	XR lets you randomise scenes and even robot size during training	Robots may still fail on truly novel objects or dynamics
Opaque decision-making – hard to debug why the policy chose an action	MR action re-play visualises attention heat-maps or next-state predictions	Interpretability tools add overhead and are not fool-proof

Conclusion & Future Trends: XR-Powered Robot Learning

Key Takeaways

XR solves data & safety bottlenecks while ML provides adaptable intelligence

Simulation-first + AR/MR validation creates robust sim-to-real pipelines

Human-in-the-loop via MR boosts transparency and trust

Emerging Trends

Foundation policies trained on billions of XR scenes

Cloud-scale sim clusters & edge rendering for low-latency MR

Multi-human teleoperation & data crowdsourcing in VR

Regulatory push for MR safety visualisation standards

Open Questions

How to benchmark XR-generated data quality?

Certifying MR safety layers for industrial robots

Balancing on-device vs. cloud inference latency

Privacy in immersive data collection