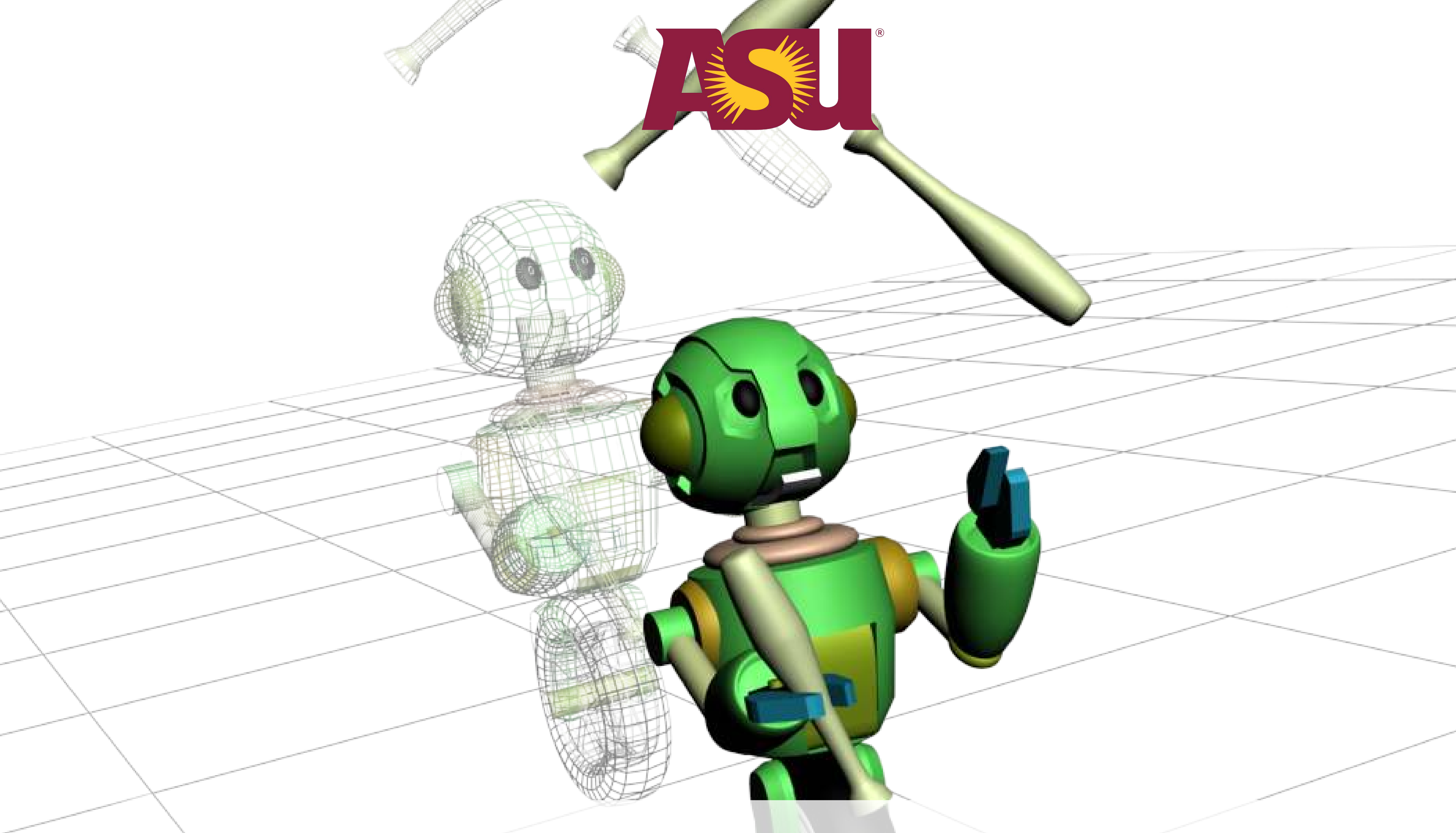


INTRODUCTION TO ROBOT LEARNING

HENI BEN AMOR, PH.D.
ARIZONA STATE UNIVERSITY

Interactive Robotics Lab

The development of this course was supported by an Intel AI Academy grant. We thank the sponsor for the continuing support of open-source efforts in research and education.

Interactive Robotics Lab

Robotics Today

Towards Next-Generation Robots

Robots Today

- Structured environment
- Static in position and task
- Pre-programmed behavior
- Limited vision capabilities
- Limited human interaction

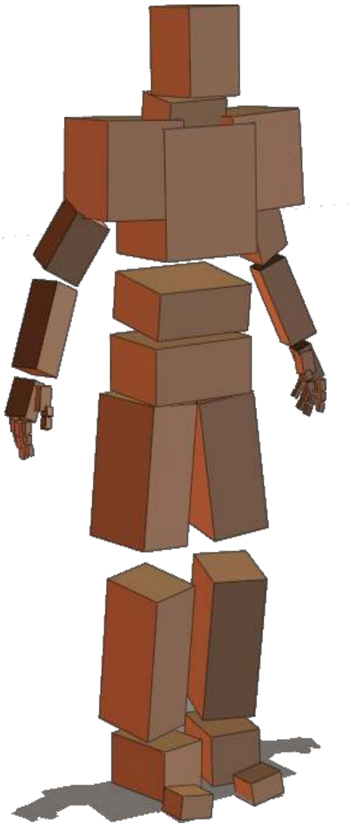
Next-Generation Robots

- Unstructured environment
- Navigation and mobility
- Adaptation to changes
- Safe human-robot contact
- Longterm autonomy

Interactive Robotics Lab

What is Robot Learning?

Robot



Machine Learning

Environment

Challenges

- Learning is an iterative process
- Automatization of the learning process
- Robot wear and tear
- Simulation ≠ reality a.k.a. reality gap
- Stochastic, dynamic environments
- Often involves human contact → safety

Comparison to Traditional Machine Learning Scenarios

- Data is coming in at 20Hz to 1000Hz
- Continuous stream of data, **online**
- **High dimensionality** of controlled system
- Learning needs to be **sample efficient**
- Learning needs to be robust to **shifting input distribution**
- Learning needs to **detect relevant features**

[Schaal and Atkeson, 2010] “Learning Control in Robotics”

Robot Learning: Applications

Learning to Control

Examples:

- Learning Motor Skills
- Inverse Kinematics
- Forward Dynamics
- Inverse Dynamics

State Estimation

Examples:

- Robot States
- Sensor Fusion
- Human Intention
- Detecting Events

Metrics & Conditions

Examples:

- Pre- / Post-conditions
- Desirability of States
- Stability of Grasp

Learning Perception

Examples:

- Detecting Objects
- Segmentation
- Recognition
- Tracking

Control Architectures

Examples:

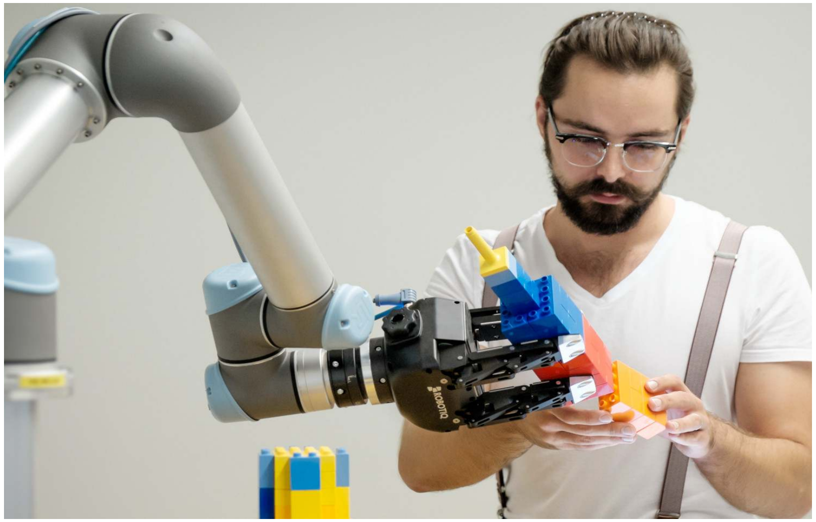
- Hierarchies of Behaviors
- Behavior Switching
- Multi-Robot Motion

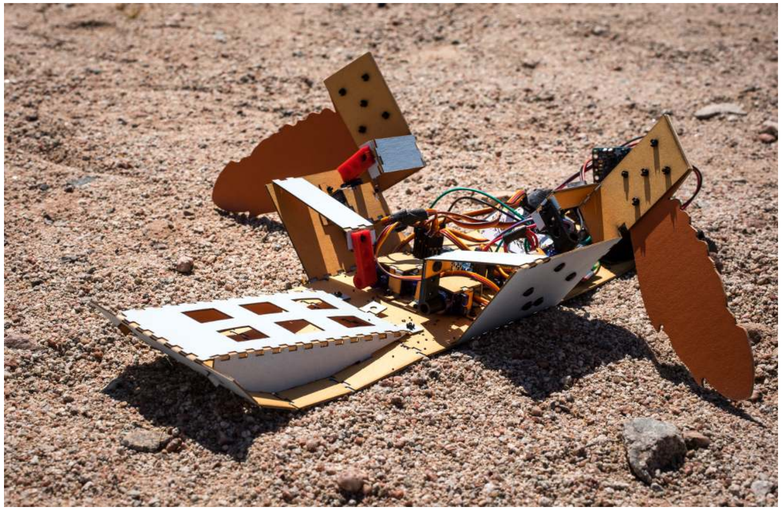
Morphologies

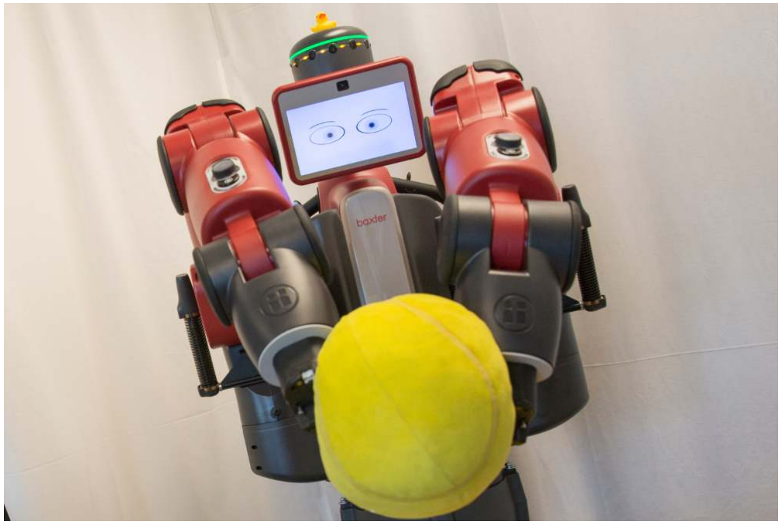
Examples:

- Robot Design
- Kinematic Structure
- Emergent Self-Model

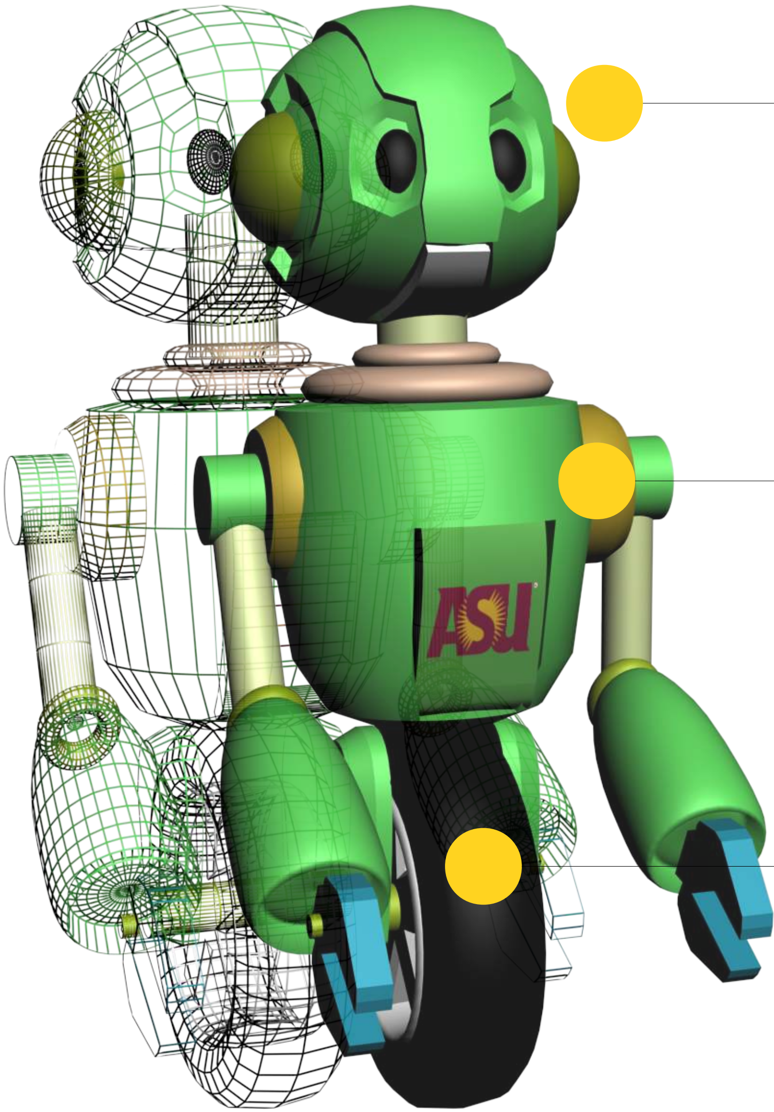
Examples







Our Robot: To-be-named



Brain:

Requires algorithms for navigation and collision avoidance

Arms:

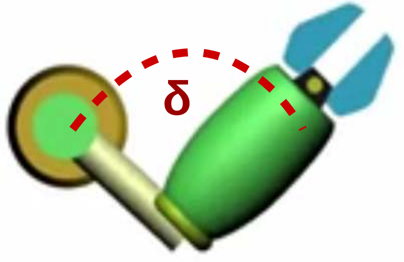
Requires algorithms for grasping and manipulation

Uniwheel:

Requires algorithms for stabilization and control

Robot Control

- We send control commands to robot for execution
- **Position control** (PC): command specifies the joint angle position the robot should take on
- **Torque control** (TC): command specifies the torques to be executed by the robot
- Torque control enables a richer variety of motor skills but is more challenging in implementation
- We will use position control, e.g. send angle δ of lower arm position

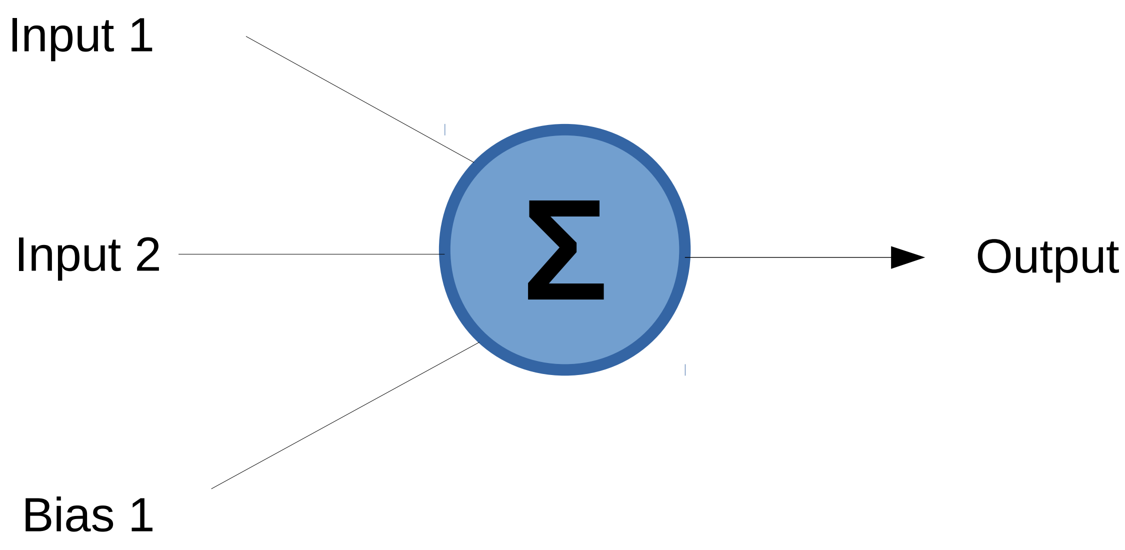


Biological Neural Networks

- Human brain ~86,000,000,000 neurons
- Each neuron connected to ~1000 others
- Electrochemical **inputs**
- **Only fire if** signal exceeds voltage threshold
- Signals are **spikes**
- All-or-nothing response

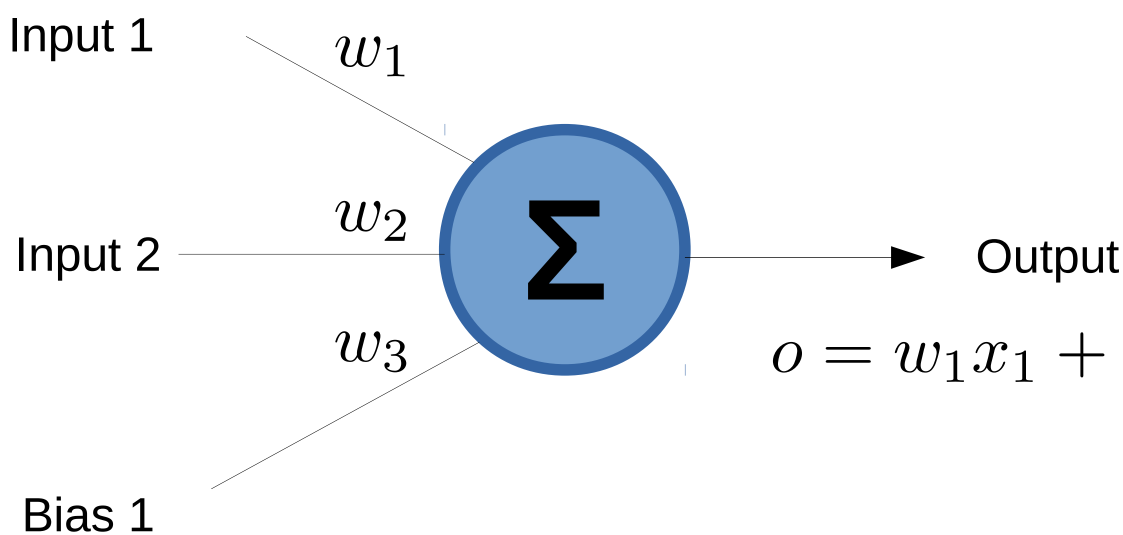
Linear Perceptron

- Inspired by biological neuron



Linear Perceptron

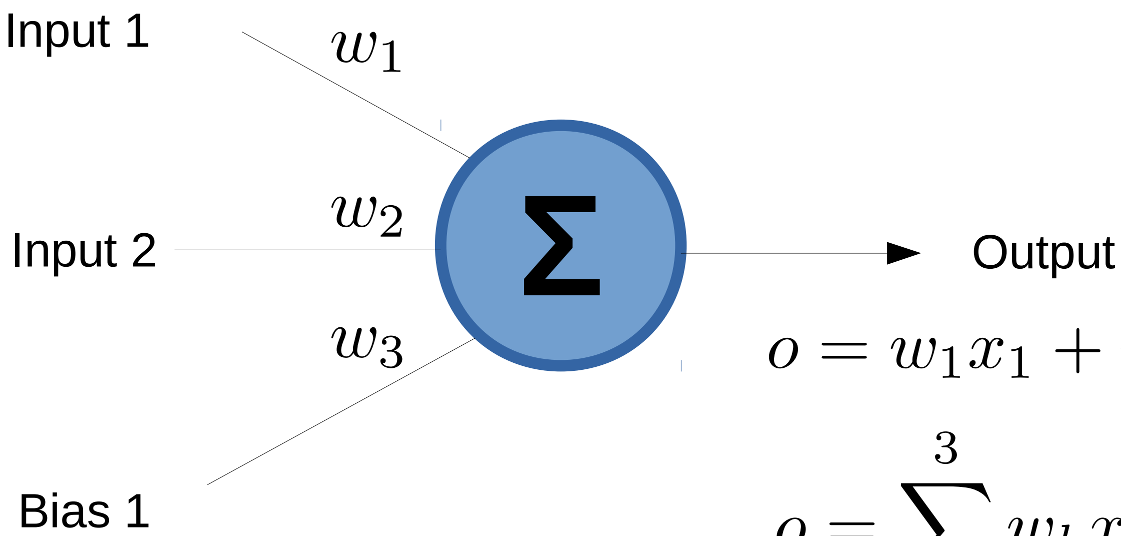
- Inspired by biological neuron



$$o = w_1 x_1 + w_2 x_2 + w_3 x_3$$

Linear Perceptron

- Inspired by biological neuron

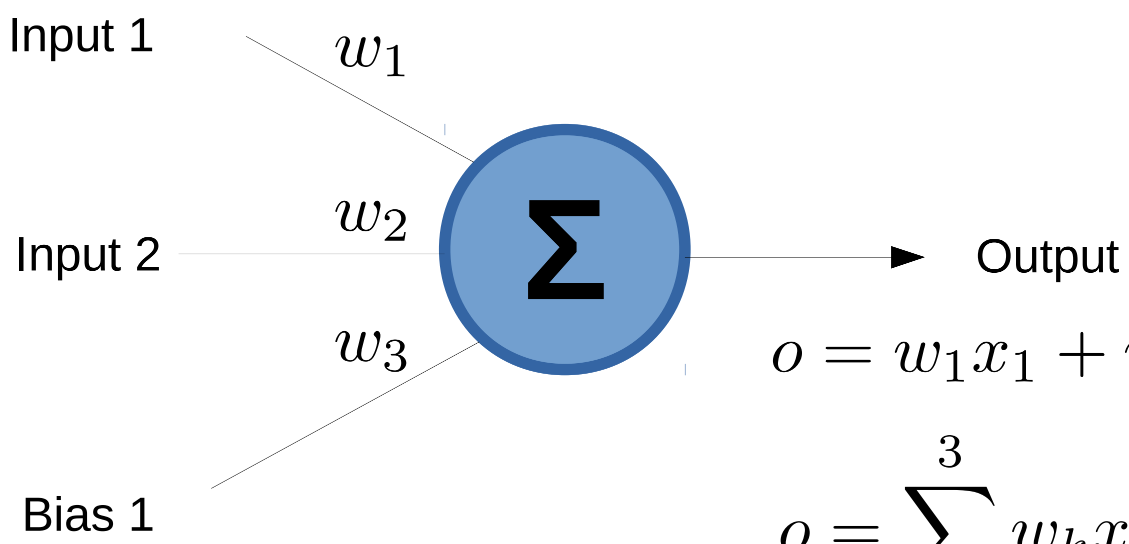


$$o = w_1 x_1 + w_2 x_2 + w_3 x_3$$

$$o = \sum_{k=1}^{3} w_k x_k$$

Linear Perceptron

- Inspired by biological neuron



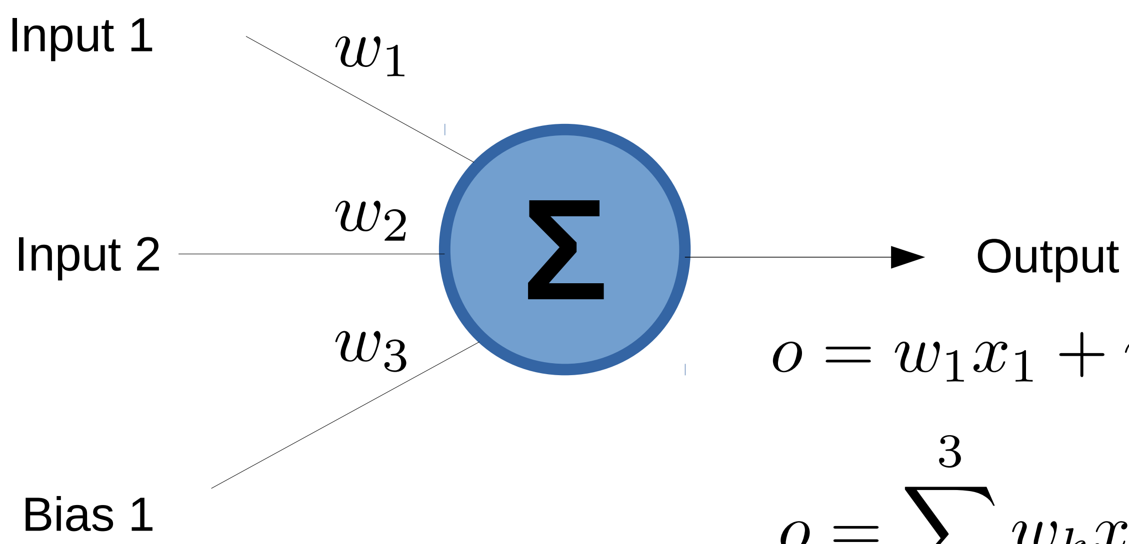
$$o = w_1 x_1 + w_2 x_2 + w_3 x_3$$

$$o = \sum_{k=1}^{3} w_k x_k$$

$$o = \mathbf{w}^T \mathbf{x}$$

Linear Perceptron

- Inspired by biological neuron



$$o = w_1 x_1 + w_2 x_2 + w_3 x_3$$

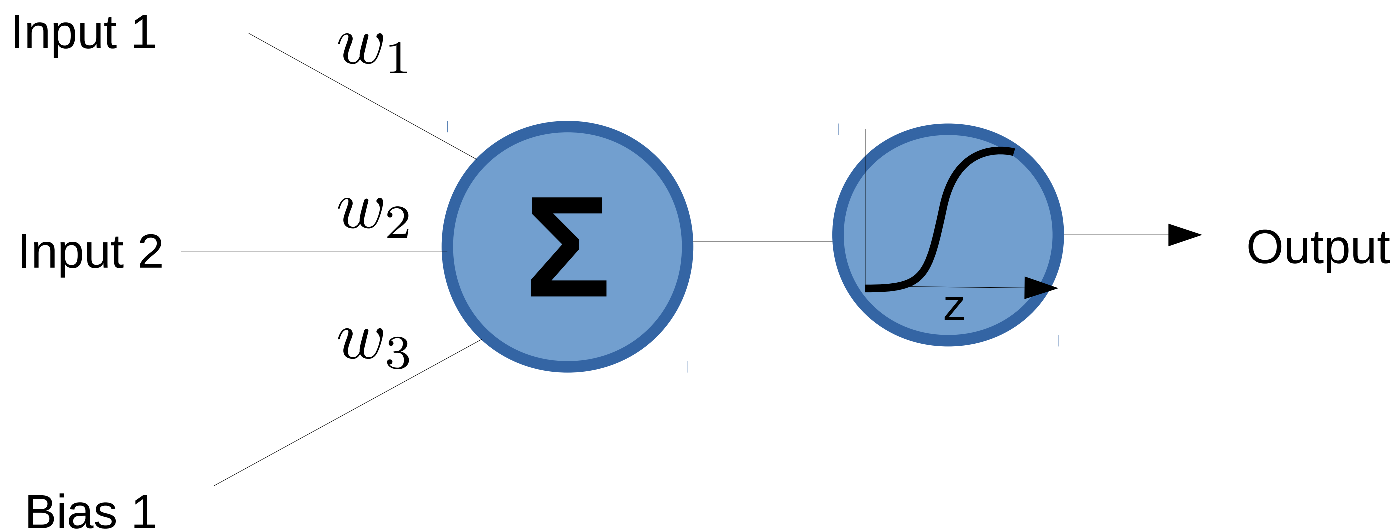
$$o = \sum_{k=1}^{3} w_k x_k$$

$$o = \mathbf{w}^T \mathbf{x}$$

Learning = determining the weights **(for now)**

Nonlinear Perceptron

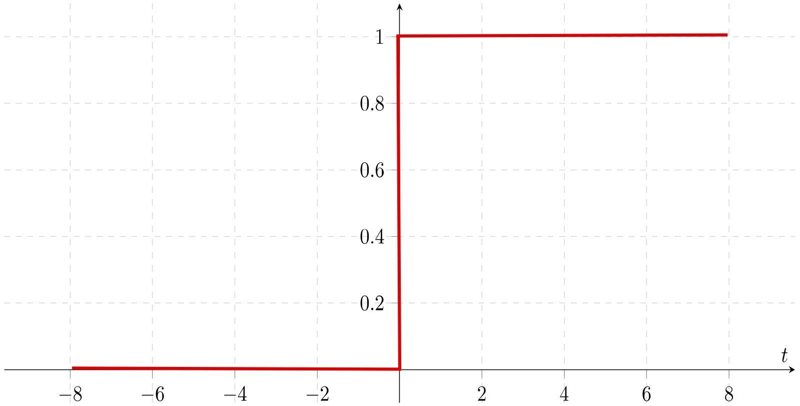
- Add nonlinear activation function ϕ

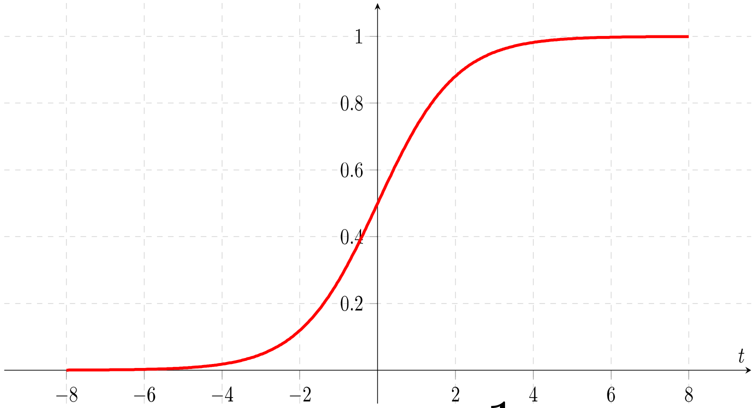


- Output is calculated via: $o = \phi(\mathbf{w}^T \mathbf{x})$
- Possible activation function: $\phi(z) = \dfrac{1}{1 + \exp(-z)}$

Sigmoid Units

- A **soft** version of a threshold unit





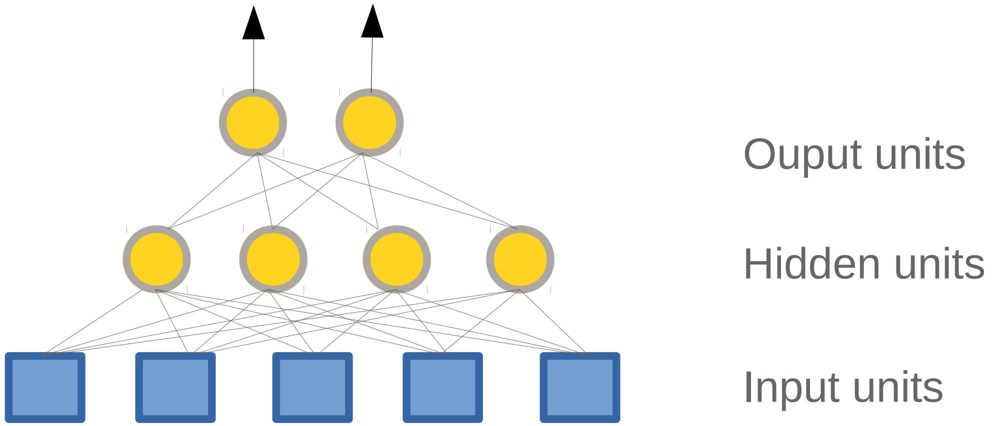
$$f(z)\begin{cases} 1 \text{ if } z > 0 \\ 0 \text{ if } z \leq 0 \end{cases}$$

$$\phi(z) = \frac{1}{1+\exp(-z)}$$

Nice property $\dfrac{\partial \phi(z)}{\partial z} = \phi(z)(1-\phi(z))$

Multi Layer Perceptron

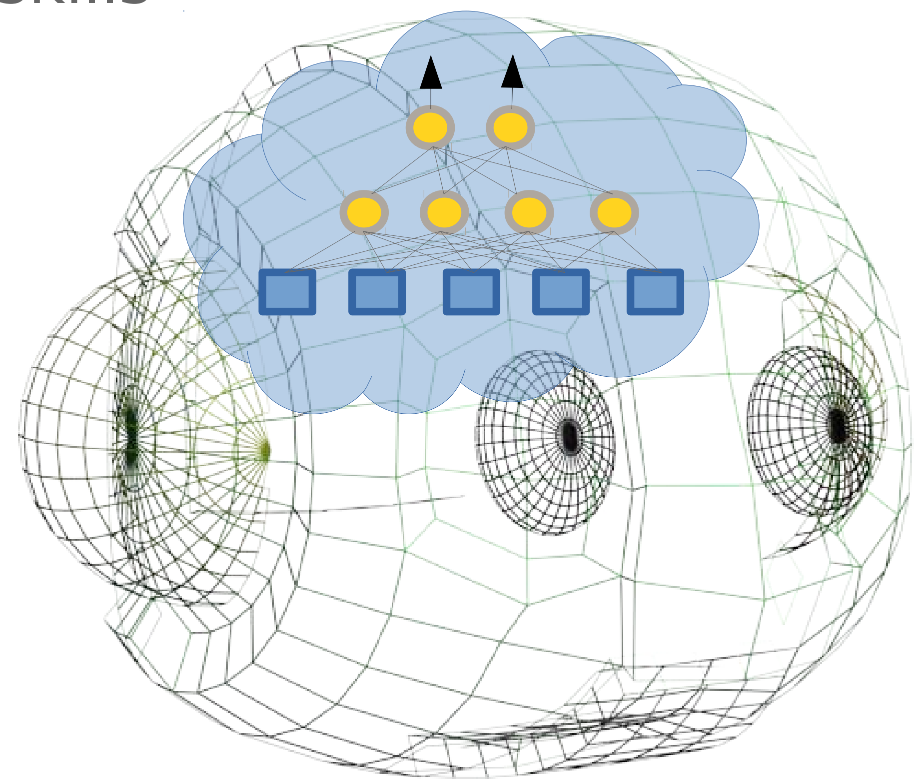
- Artificial Neural Network
- Hierarchy of neurons
- Input layer, hidden layers, output layer
- 2 Layers = all continuous functions
- 3+ Layers = all functions



Interactive Robotics Lab

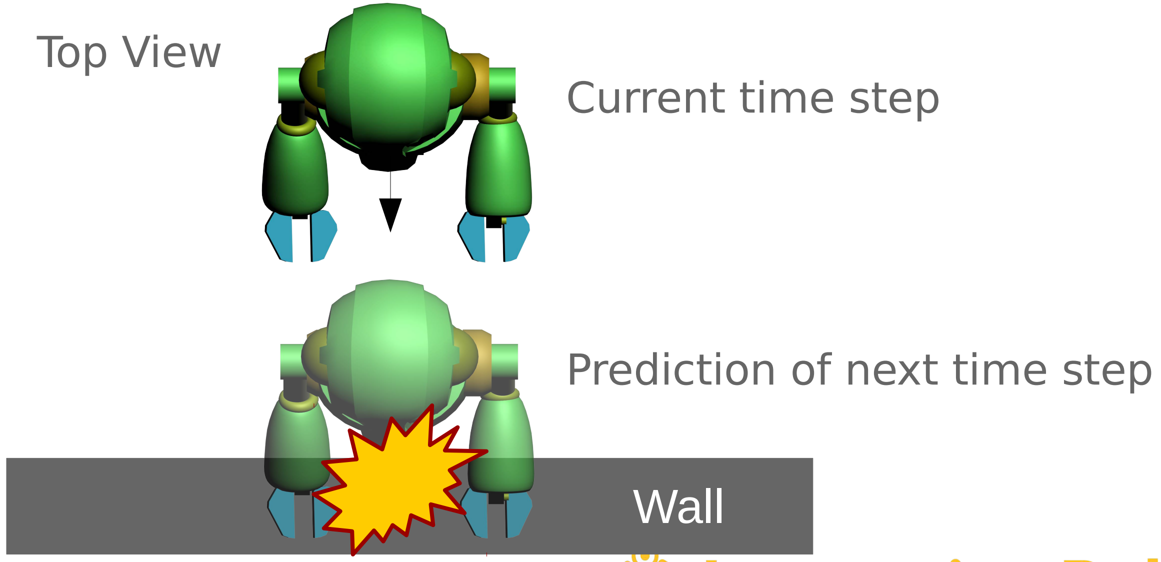
Neural Networks for Robot AI

- We can train ANNs to impart robot with decision-making skills



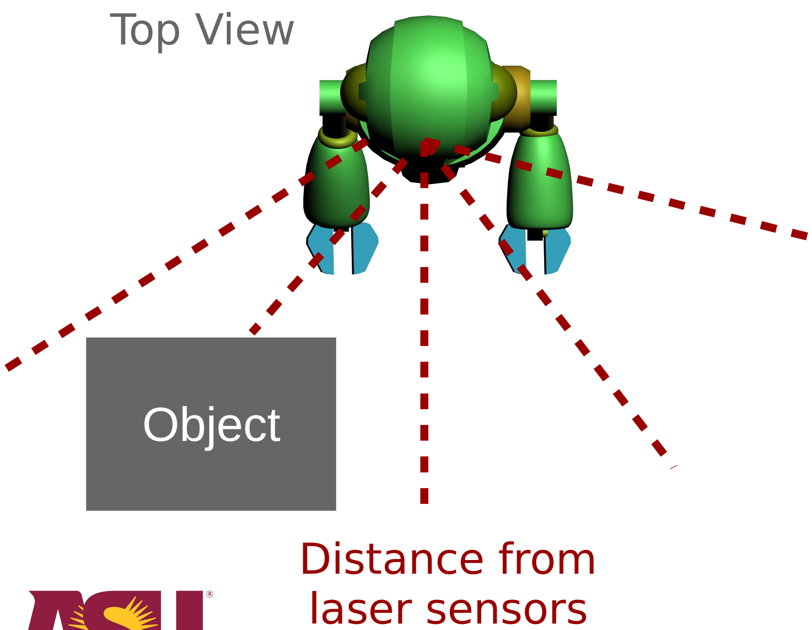
Example 1: Learning to Predict Collision

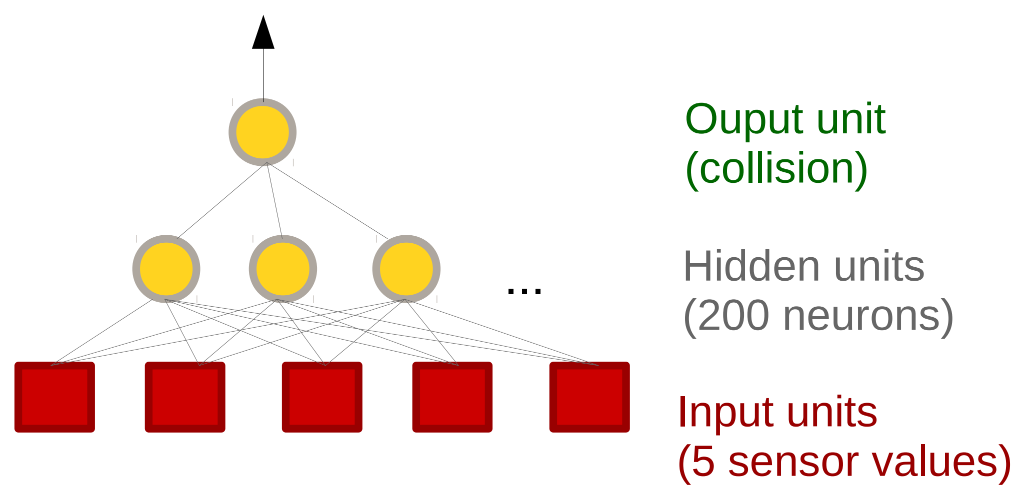
- Example task for Deep Learning in robotics
- Learning a predictive model of collisions
- Input to neural network: distance sensor values
- Output of neural network: {collision, !collision}



Example 1: Creating the Network

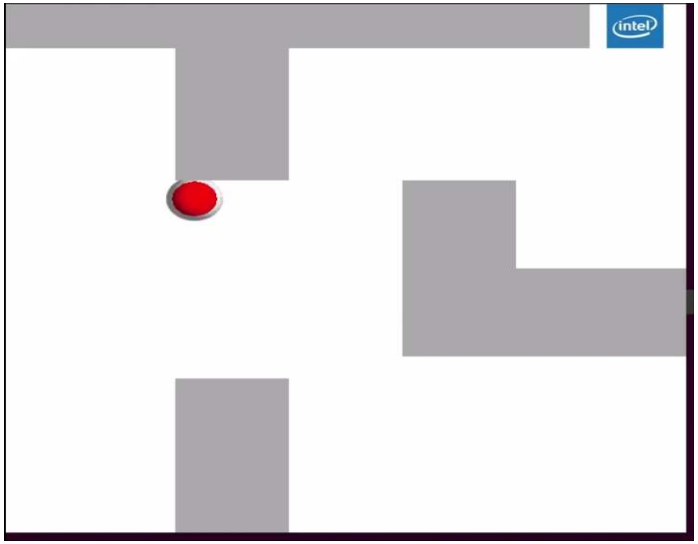
- Our goal: predict collision before they occur
- Input to neural network: distance sensor values
- Output of neural network: {collision, !collision}
- Network output will be in the range [0..1]





Example 1: Collecting Training Data

- Let the robot wander randomly
- Each step record: sensor values + {collision, !collision}
- Collision can be measured by a bumper sensor



Play Video

Example 1: Training the Network

- We want our ANN to predict collision
- We have training data where for each input we have the corresponding desired output (label)
- This is called: **supervised learning**
- Now: change ANN weights to mimick labelled data
- To check accuracy of ANN we will test it on a set of unseen data (test set)

Supervised Learning

Backpropagation

- Given input and output learn **weights**
- Gradient descent minimizing quadratic error

$$E = \frac{1}{2}\sum_{i=1}^{N} ||a_i - y_i||^2$$

- Aka minimize quadratic difference between target and output of the network

Approach

- Given a set of training data
- Each sample a tuple $< \mathbf{x}, \mathbf{t} >$
- Where $\mathbf{x}$ is the input and $\mathbf{t}$ is the desired output
- Train network such that

$$\mathrm{NN}(\mathbf{x}) \approx \mathbf{t} \quad \forall \mathbf{x} \in X$$

- Assumes labeled training data
- Typically labels are provided by human annotation

BP = Gradient Descent

- Calculate gradient of network

$$\nabla E = \left(\frac{\partial E}{w_1}, \frac{\partial E}{w_2}, \cdots, \frac{\partial E}{w_l} \right)$$

- Update weights according to gradient descent
- Update equation

$$w_i \leftarrow w_i - \underbrace{\alpha}_{\text{Learning Rate}} \underbrace{\frac{\partial E}{w_i}}_{\text{Gradient}}$$

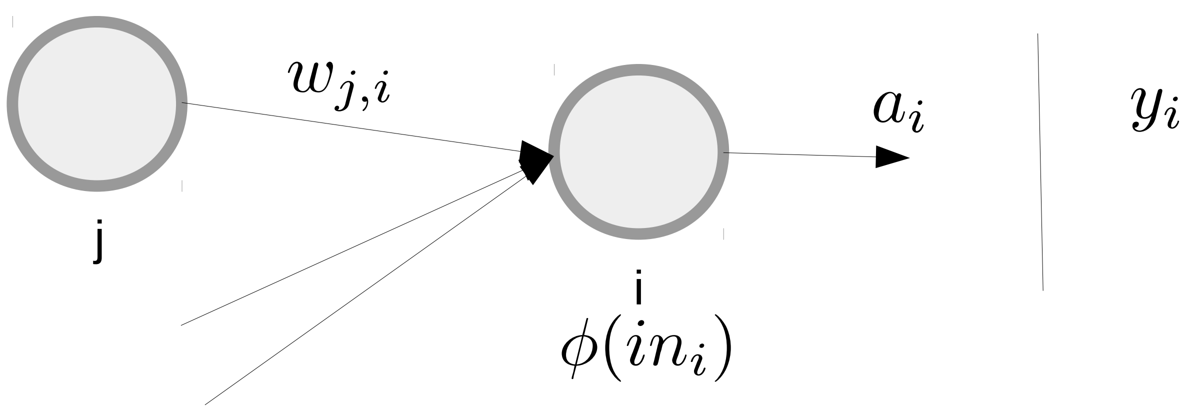
Gradient Calculation

- How to get partial derivatives:

$$\nabla E = \left(\frac{\partial E}{w_1}, \frac{\partial E}{w_2}, \cdots, \frac{\partial E}{w_l} \right) \ ?$$

- Apply **chain rule:**

$$D\{f(g(x))\} = f'(g(x))g'(x)$$



Derivation of Gradient

Following [Russel, Norvig]

$$
\begin{aligned}
\frac{\partial E}{\partial w_{j,i}} &= -(y_i - a_i)\frac{\partial a_i}{\partial w_{j,i}} \\
&= -(y_i - a_i)\frac{\partial \phi(in_i)}{\partial w_{j,i}} \\
&= -(y_i - a_i)\phi'(in_i)\frac{\partial in_i}{\partial w_{j,i}} \\
&= -(y_i - a_i)\phi'(in_i)\frac{\partial}{\partial w_{j,i}}\left(\sum_j w_{j,i} a_j\right) \\
&= \underbrace{-(y_i - a_i)}_{\text{Difference to target}}\underbrace{\phi'(in_i)}_{\text{Derivative of activation function}}\underbrace{a_j}_{\text{Activation}}
\end{aligned}
$$

Backpropagation Algorithm

- Initialize all weight randomly
- Until convergence do
  - Input example and calculate network output
  - For each output unit do

$$\delta_k \leftarrow a_k(1 - a_k)(y_k - a_k)$$

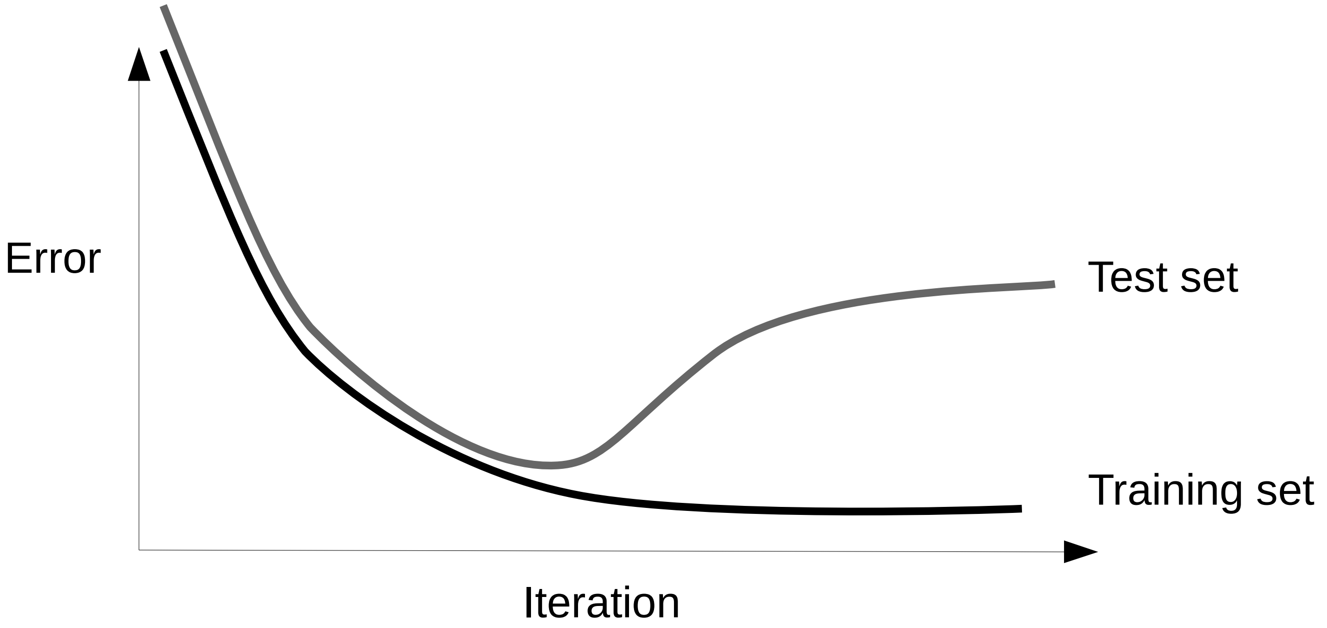
  - Foe each hidden unit do

$$\delta_h \leftarrow a_h(1 - a_h) \sum_{k \in succ(h)} w_{h,k}\delta_k$$

  - Update weights $w_{i,j} \leftarrow w_{i,j} + \alpha\ \delta_i x_{i,j}$

Ensuring Generalization

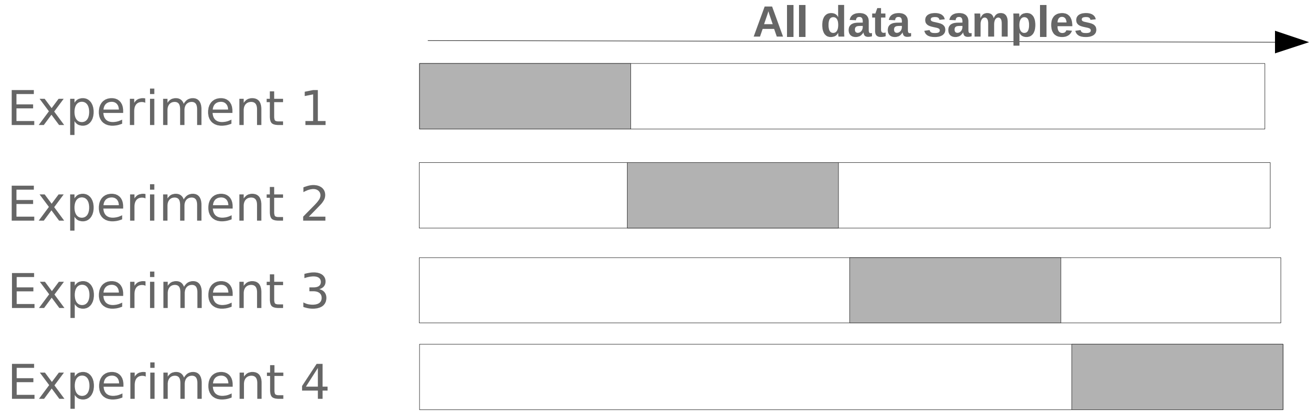
- **Overfitting** to data is bad
- Early stopping



Interactive Robotics Lab

K-Fold Cross-validation

- Divide data in K-folds
- Train and test on remaining fold



- True error is **average** of individual errors

Neural Networks with PyTorch

- PyTorch is an open source machine learning tensor library for Python
- Freely available under: http://pytorch.org/
- Wide range of networks and training algorithms
- Allows for dynamic networks
- Very accessible
- We will use it in the remainder of course

PYTORCH

A Simple Network in PyTorch

- Defining a simple network

```
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out

net = Net(input_size, hidden_size, num_classes)
```

Frequently used Functions

nn.Module() - Neural Network Module in pytorch

nn.Linear(in_features,out_features) - Applies linear transformation to incoming data

nn.Relu() - Applies Rectified Linear Unit function element wise

Net.parameters() - Returns all the learnable parameters

The Loss Function

- A Loss function takes the (input, output) pair and computes a measure which indicates how far away the output is from the target
- There are several loss function that can be used.

  Let's use nn.MSELoss()

```
criterion = nn.MSELoss()
loss  = criterion(output,target)
loss.backward()
```

- When we call loss.backward(), the whole graph is differentiated w.r.t. the loss, and all Variables in the graph will have their .grad Variable accumulated with the gradient

Training the Weights of a Network

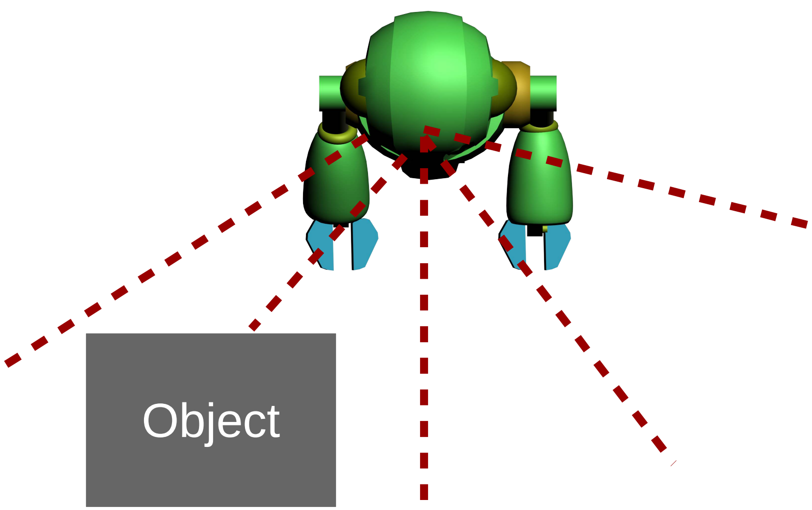
- The most frequent update rule used in practice is stochastic gradient descent (SGD)
- Many sophisticated learning methods are also implemented: Nesterov-SGD, Adam, RMSProp
- Torch.optim allows you to learning method

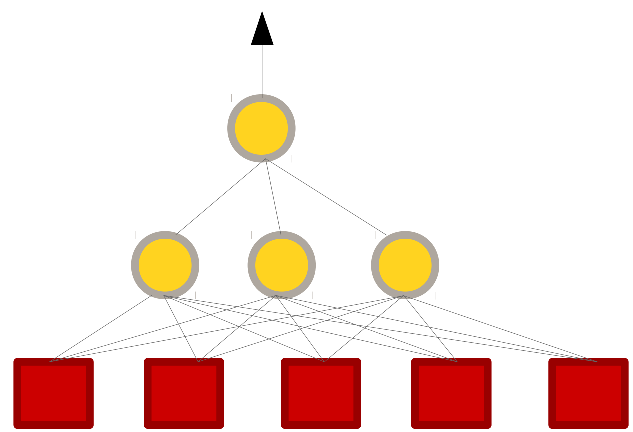
```
optimizer = optim.SGD(net.parameters(), lr=0.01)
optimizer.zero_grad()
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()
```

- optimizer.zero_grad() - zeros the gradient buffer and optimizer.step() - updates the weights

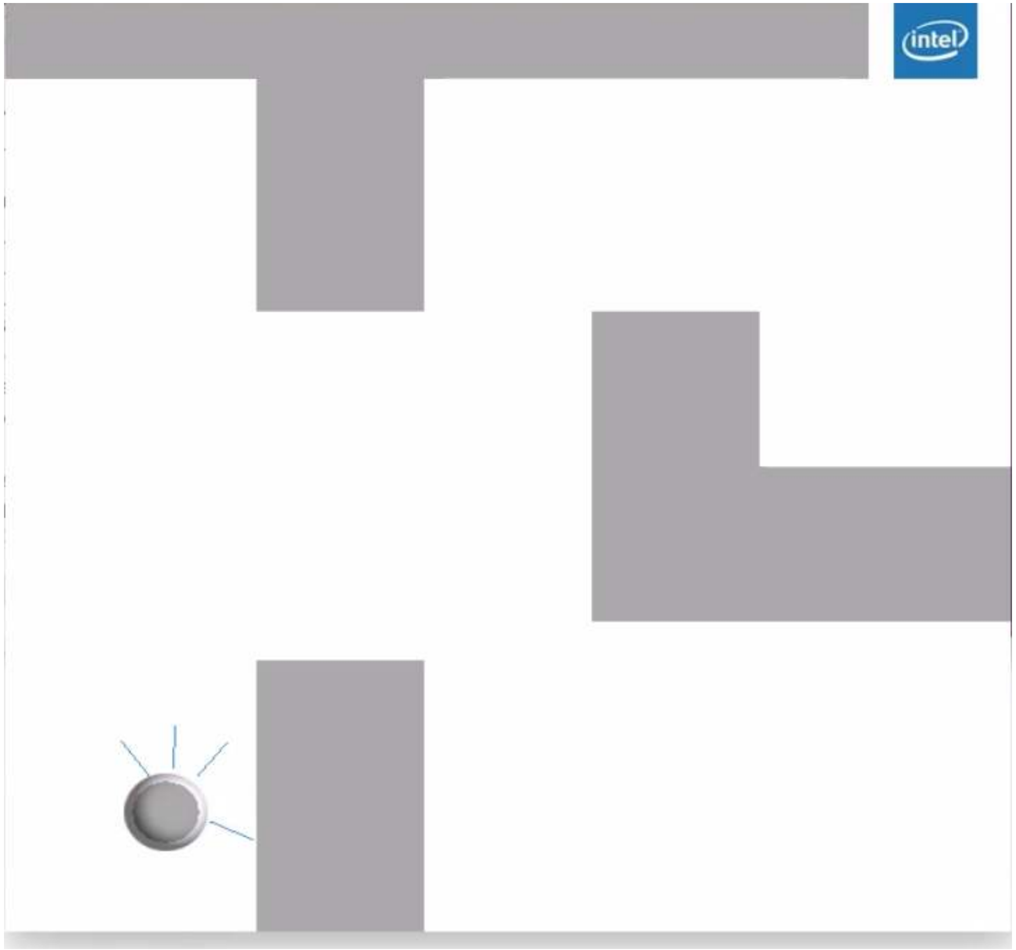
Example 1: After Training

- Our goal was to predict collisions
- After training, we use network to predict collision
- If collision is imminent → turn away from direction
- If no collision → turn to goal location





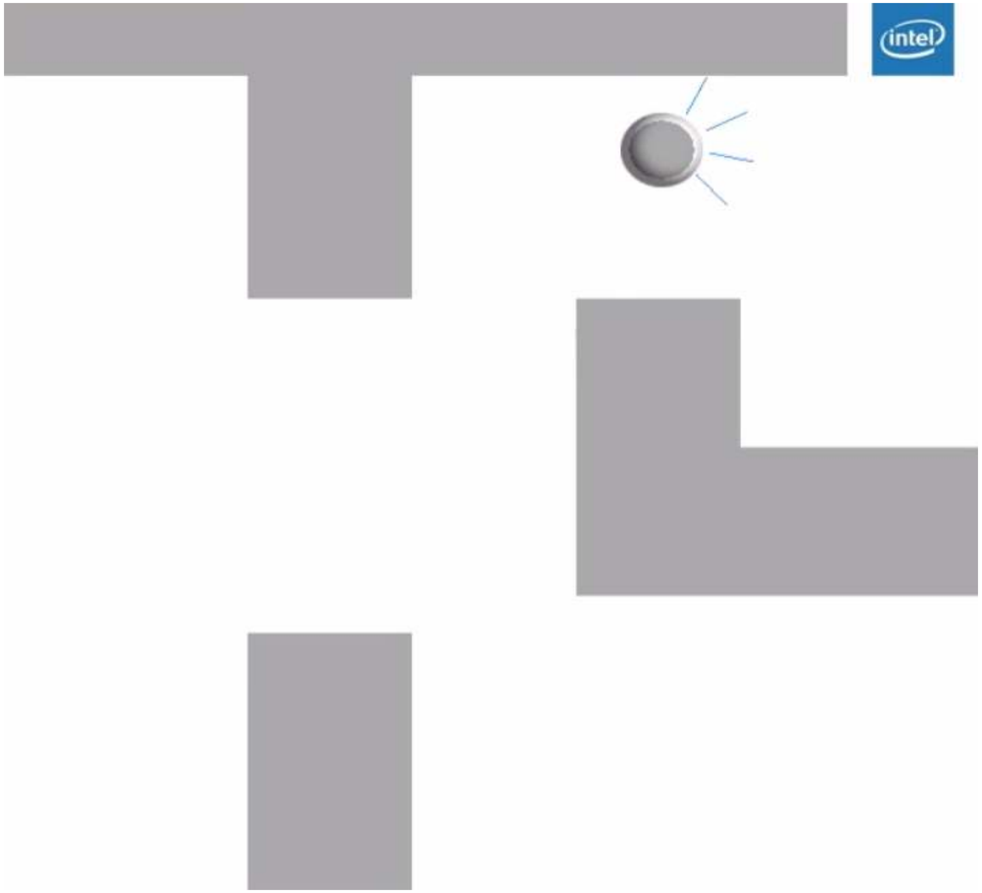
Example 1: During Training (Video)



Play Video

Interactive Robotics Lab

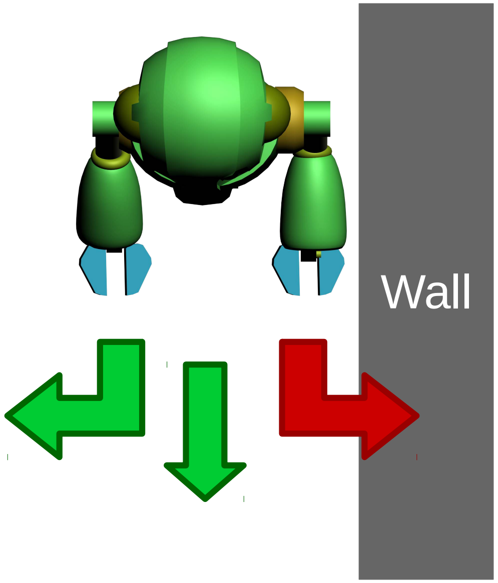
Example 1: After Training (Video)



Play Video

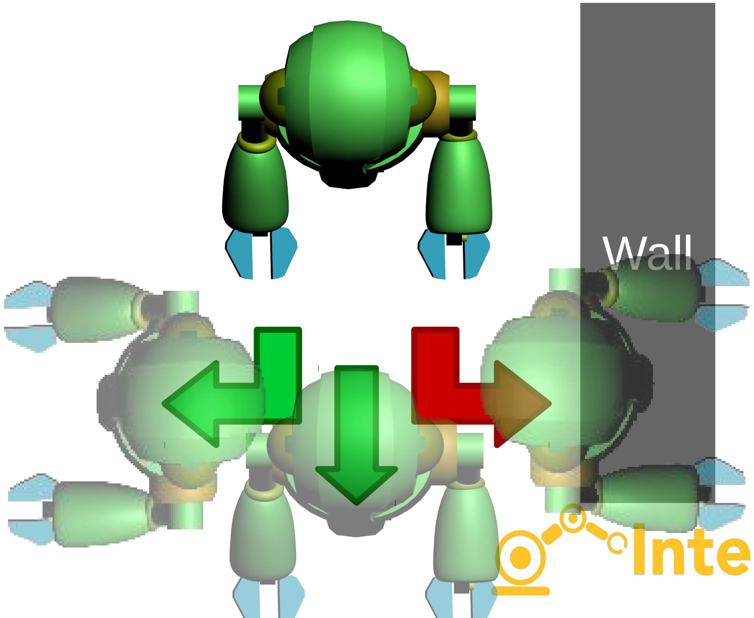
Action-Conditioned Predictive Models

- Our neural network does not take into account the robots action
- As a result it cannot disambiguate between situations where collision is dependent on action
- Example scenario:
- Collision only occurs if robot turns right!



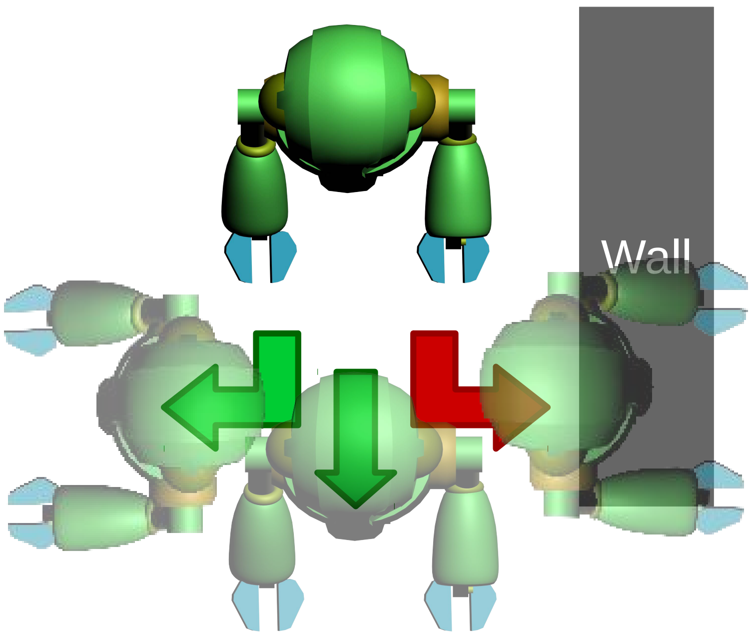
Action-Conditioned Predictive Models

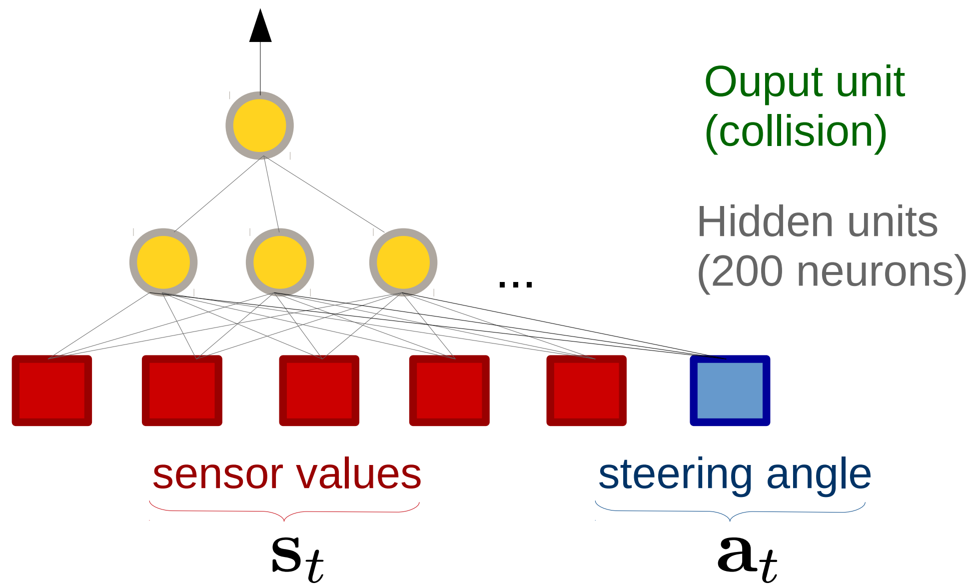
- Solution: add the action of the robot into the predictive model
- Action becomes an input to the network
- Generally, action-conditioned predictive models are functions of form $f(\mathbf{s}_t, \mathbf{a}) \rightarrow \mathbf{s}_{t+1}$



Action-Conditioned Predictive Models

$$f(\mathbf{s}_t, \mathbf{a}_t) \rightarrow \mathbf{s}_{t+1}$$





Example Application

- Python code implementing the above example can be found in folder “Supervised”
- The README includes instructions on learning and testing a model

Faster Computation with Intel MKL

Intel® MKL Optimized Mathematical Building Blocks

Linear Algebra
- BLAS
- LAPACK and ScaLAPACK
- Sparse BLAS
- PARDISO* Direct Sparse Solver
- Parallel Direct Cluster Sparse Solver
- Iterative sparse solvers

Fast Fourier Transforms
- Multidimensional
- FFTW* interfaces
- Cluster FFT

Vector Math
- Trigonometric
- Hyperbolic
- Exponential
- Log
- Power
- Root
- Vector RNGs

Deep Neural Networks
- Convolution
- Pooling
- Normalization
- ReLU
- Inner Product

Summary Statistics
- Kurtosis
- Central moments
- Variation coefficient
- Order statistics and quantiles
- Min/max
- Variance-covariance
- Robust estimators

And More
- Splines
- Interpolation
- Trust Region
- Fast Poisson Solver

Intel® Math Kernel Library

Intel® MKL

- Speeds computations for scientific, engineering, financial and machine learning applications
- Provides key functionality for dense and sparse linear algebra (BLAS, LAPACK, PARDISO), FFTs, vector math, summary statistics, deep learning, splines and more
- Included in Intel® Parallel Studio XE and Intel® System Studio Suites
- Available at no cost and royalty free

- Optimized for single core vectorization and cache utilization
- Automatic parallelism for multi-core and many-core
- Scales from cores to clusters
- Great performance with minimal effort

Intel® MKL DNN (Deep Neural Network) Functions

Highly optimized basic building blocks for DNNs	
Use cases	Inference and training Image recognition, semantic segmentation, object detection
Functions	Convolution, Inner Product Activation, Normalization, Pooling, Sum, Split/ Concat, Data transformation
Applications	Supported in Tensorflow, MXNet, IntelCaffe and more

Open Source Version under: https://github.com/intel/mkl-dnn

Summary

- We introduced supervised learning
- Used learning to predict robot collisions
- The network architecture defines input/output
- Learning the network weights with BackProp
- However, so far **no memory**
- Later we will introduce recurrent neural nets

The development of this course was supported by an Intel AI Academy grant. We thank the sponsor for the continuing support of open-source efforts in research and education.

Interactive Robotics Lab