

ROBOT DESIGN AND APPLICATIONS

Instructor

Furkan HANİLÇİ

• • •



Department
Computer Engineering



Academic Term
Fall Semester 2025-2026



LinkedIn
furkanhanilci



Github
furkanhanilci

Based on lecture materials from the University of Tübingen - Self-Driving Cars (Prof. Dr. Andreas Geiger)

ULMICHARD KARLS
UNIVERSITÄT
TÜBINGEN



e 1 1 i s

MAX PLANCK
INSTITUTE FOR
INTELLIGENT SYSTEMS

Agenda

2.1 Approaches to Self-Driving

2.2 Deep Learning Recap

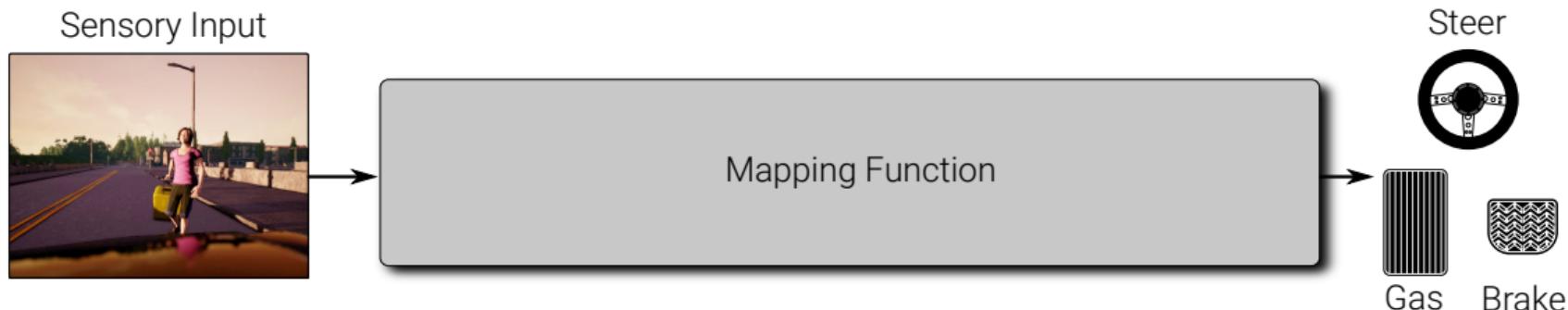
2.3 Imitation Learning

2.4 Conditional Imitation Learning

2.1

Approaches to Self-Driving

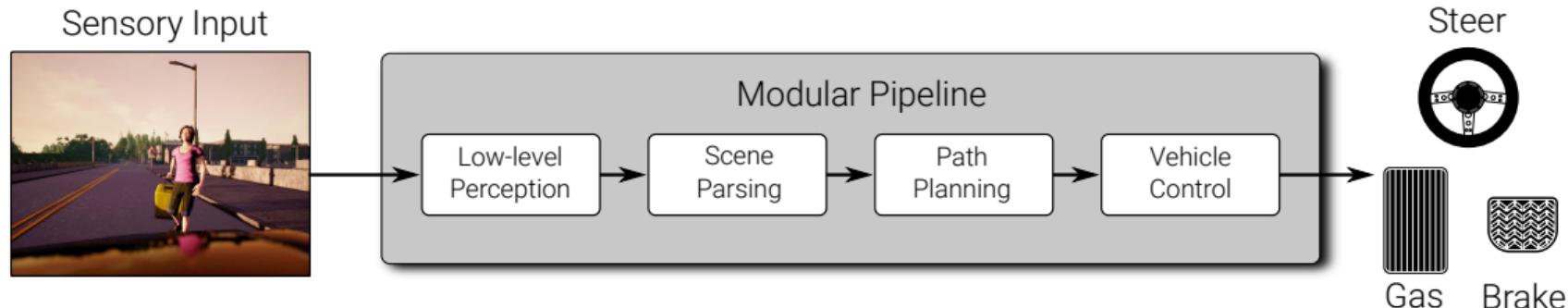
Autonomous Driving



Dominating Paradigms:

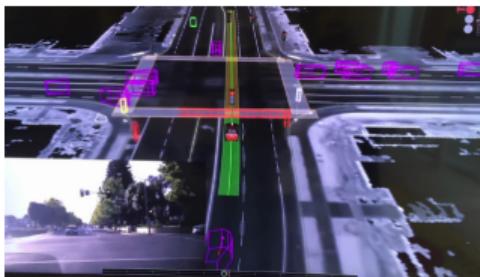
- ▶ Modular Pipelines
- ▶ End-to-End Learning (Imitation Learning, Reinforcement Learning)
- ▶ Direct Perception

Autonomous Driving: Modular Pipeline

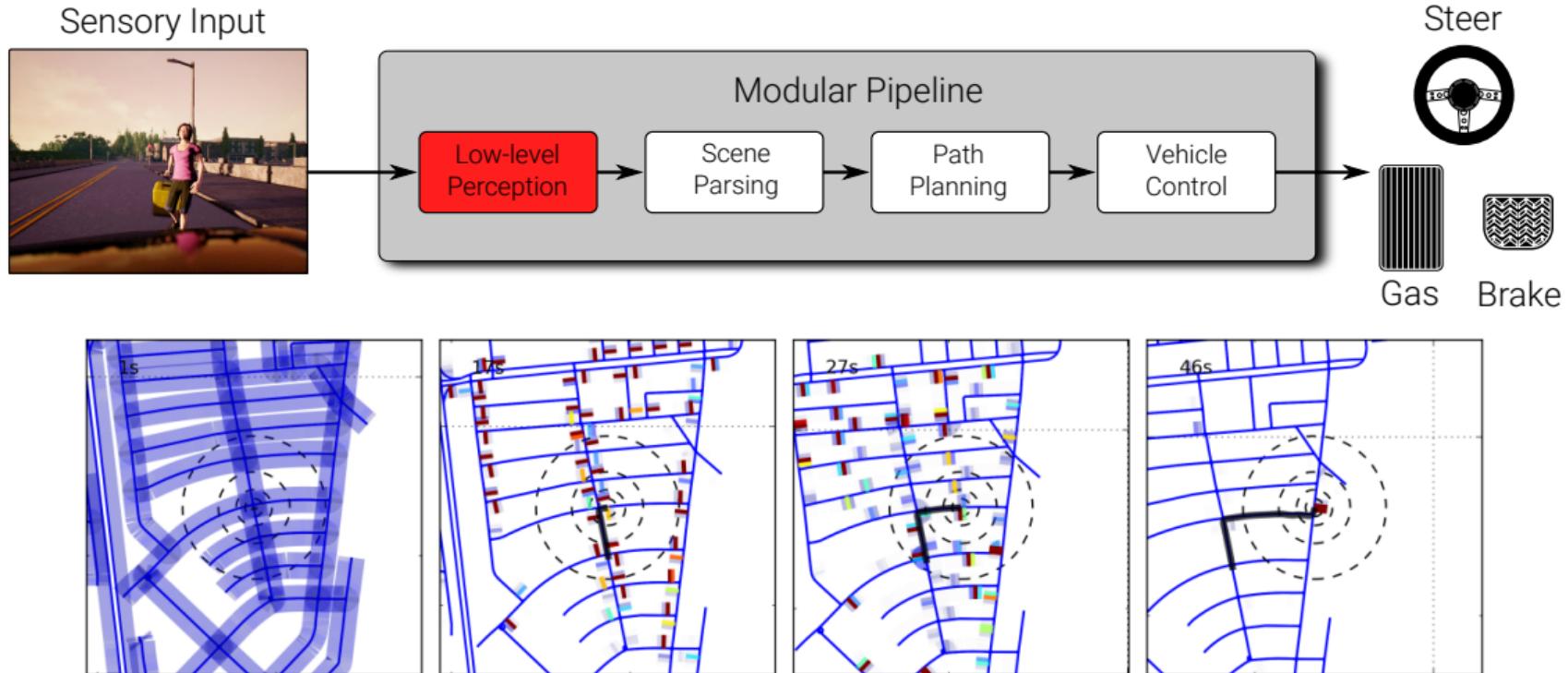


Examples:

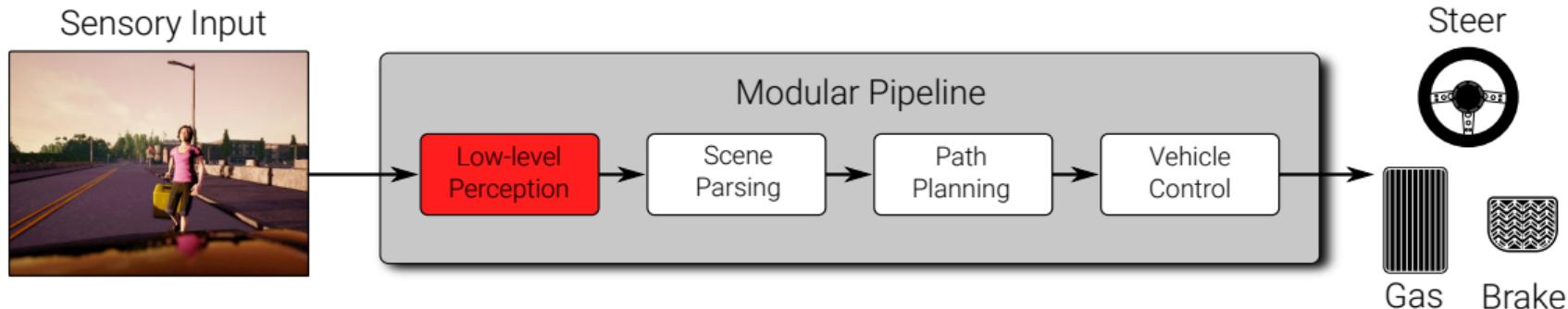
- ▶ [Montemerlo et al., JFR 2008]
- ▶ [Urmson et al., JFR 2008]
- ▶ Waymo, Uber, Tesla, Zoox, ...



Autonomous Driving: Modular Pipeline

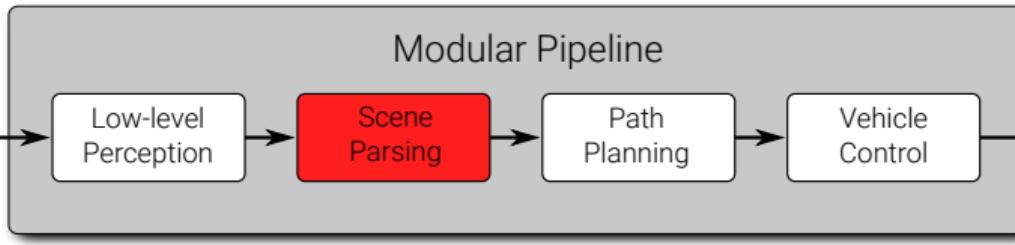


Autonomous Driving: Modular Pipeline



Autonomous Driving: Modular Pipeline

Sensory Input



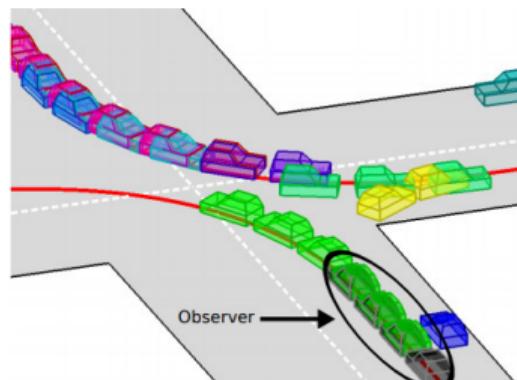
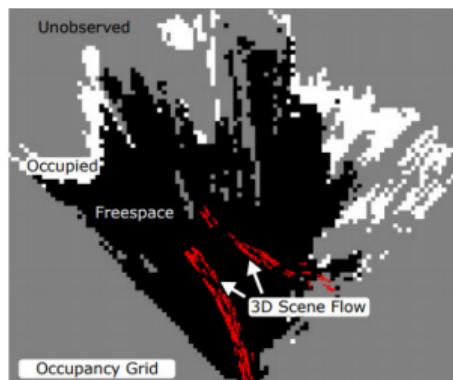
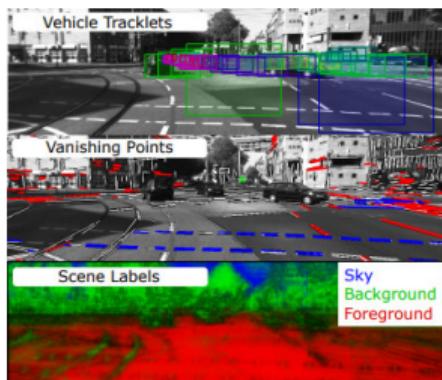
Steer



Gas

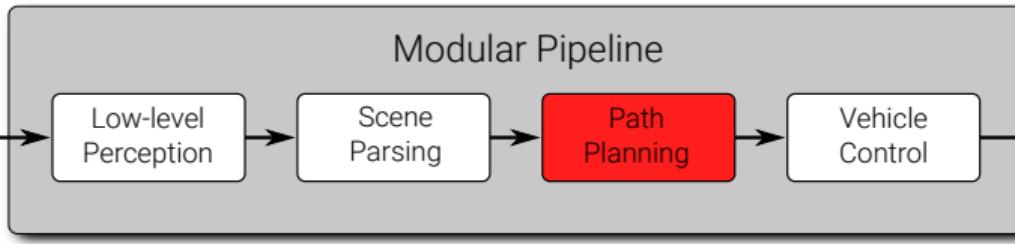


Brake



Autonomous Driving: Modular Pipeline

Sensory Input



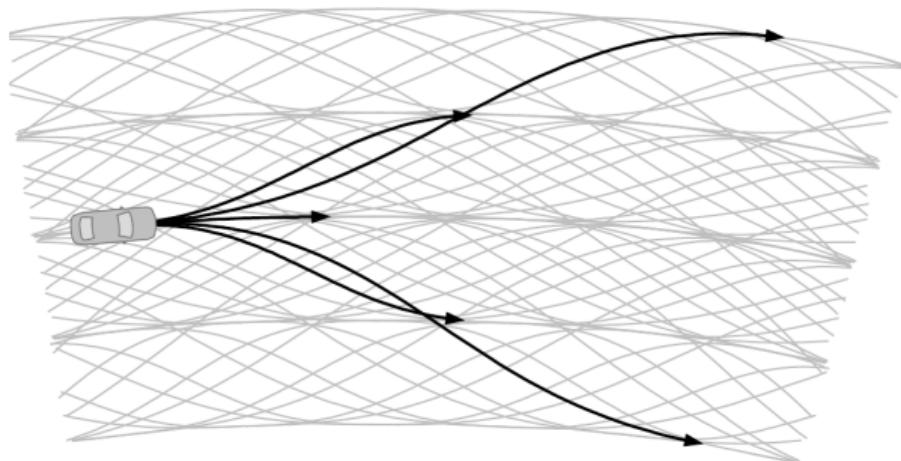
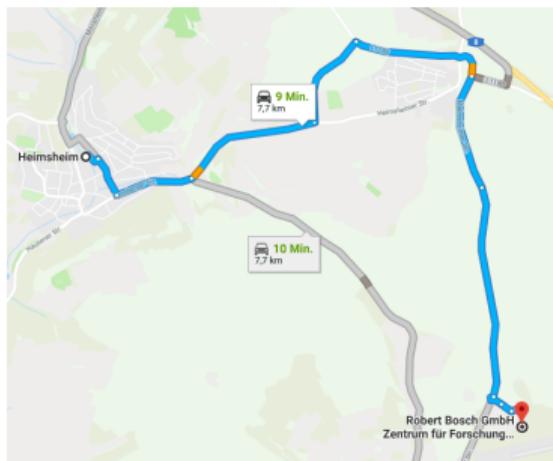
Steer



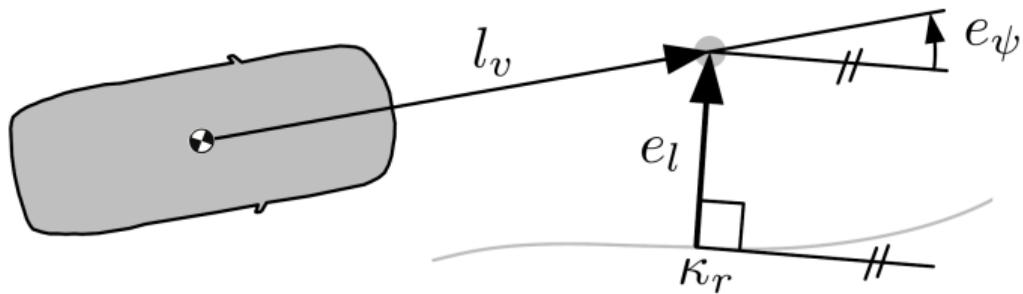
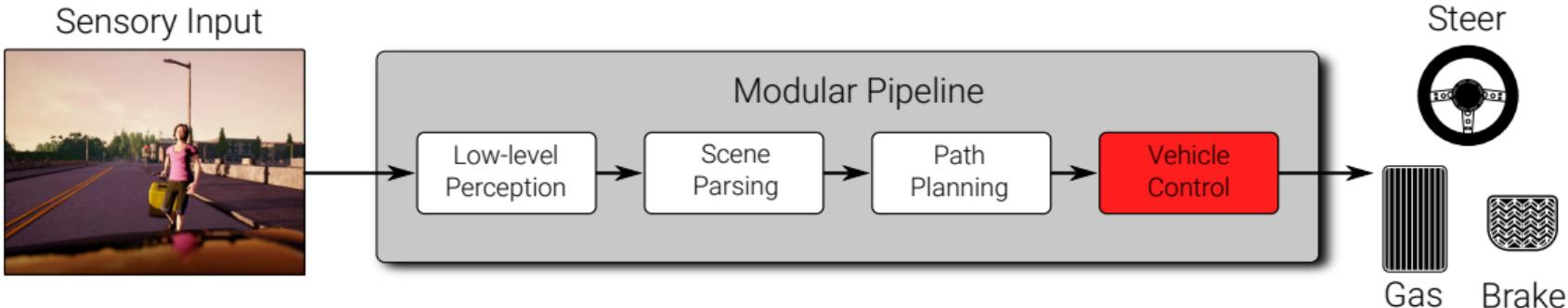
Gas



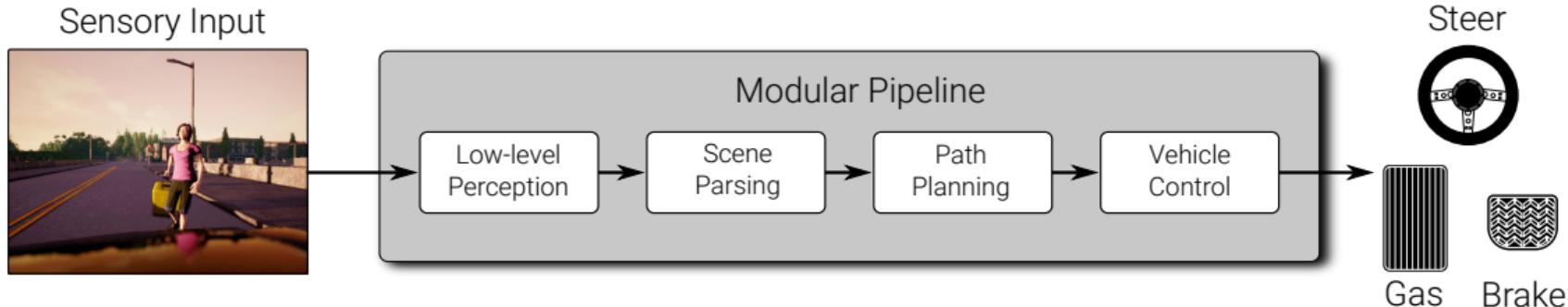
Brake



Autonomous Driving: Modular Pipeline



Autonomous Driving: Modular Pipeline



Pros:

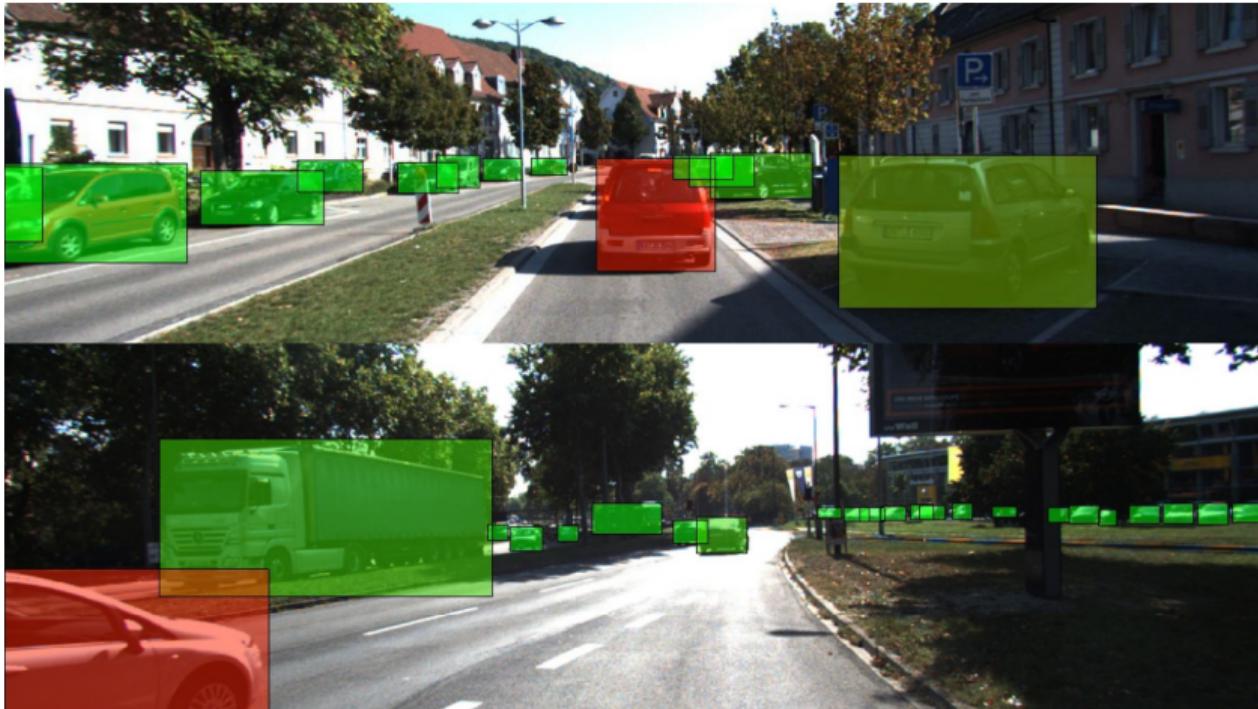
- ▶ Small components, easy to develop in parallel
- ▶ Interpretability

Cons:

- ▶ Piece-wise training (not jointly)
- ▶ Localization and planning heavily relies on HD maps

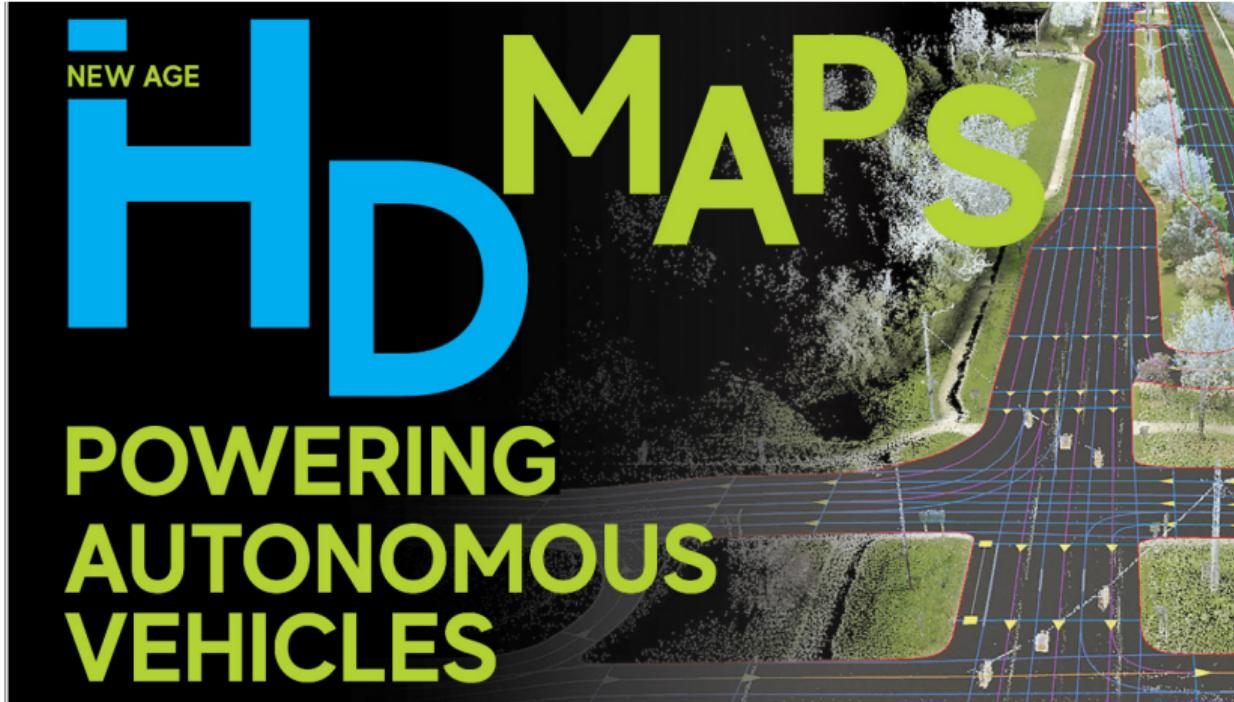
HD maps: Centimeter precision lanes, markings, traffic lights/signs, human annotated

Autonomous Driving: Modular Pipeline



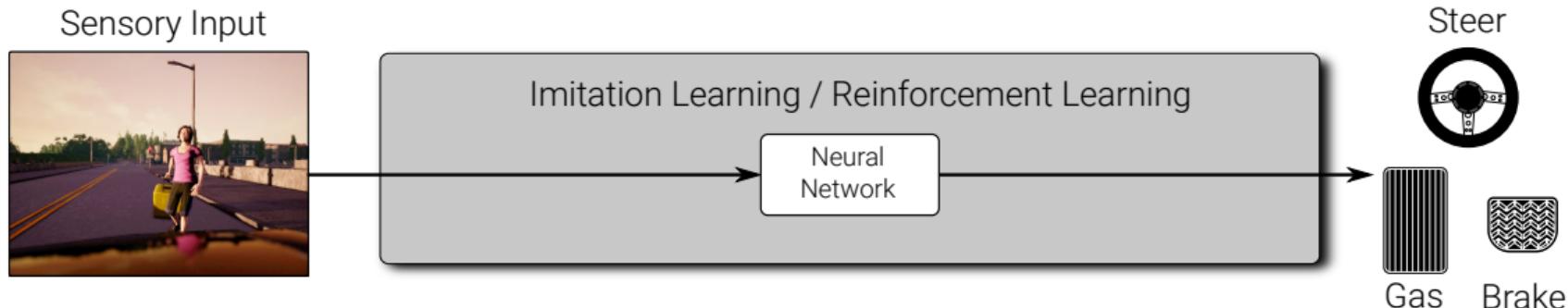
- ▶ Piece-wise training difficult: not all objects are equally important!

Autonomous Driving: Modular Pipeline



- ▶ HD Maps are expensive to create (data collection & annotation effort)

Autonomous Driving: End-to-End Learning

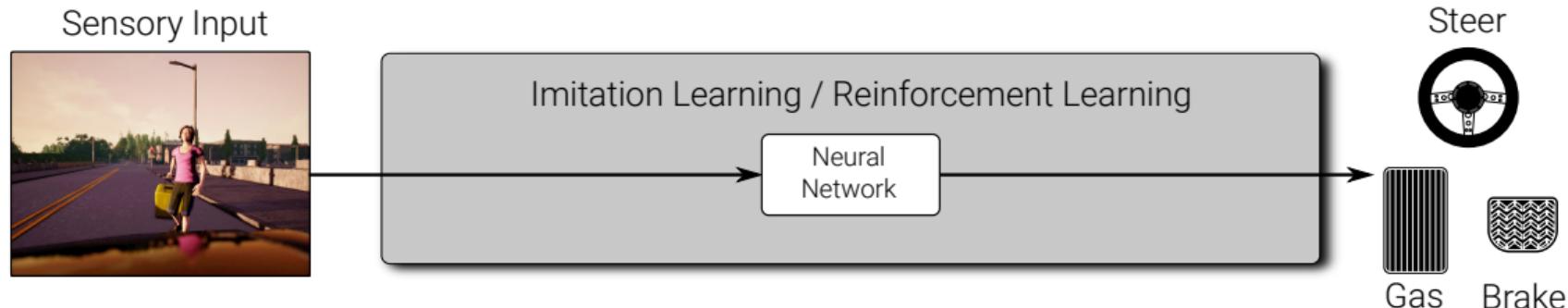


Examples:

- ▶ [Pomerleau, NIPS 1989]
- ▶ [Bojarski, Arxiv 2016]
- ▶ [Codevilla et al., ICRA 2018]



Autonomous Driving: End-to-End Learning



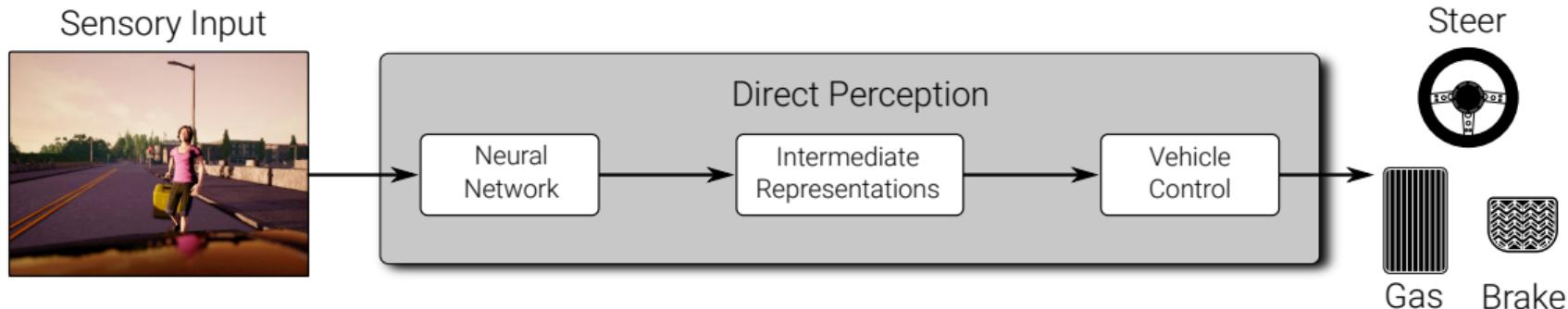
Pros:

- ▶ End-to-end training
- ▶ Cheap annotations

Cons:

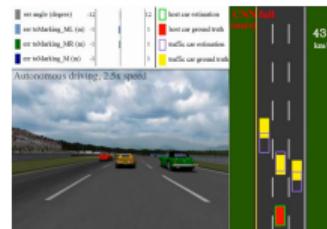
- ▶ Training / Generalization
- ▶ Interpretability

Autonomous Driving: Direct Perception

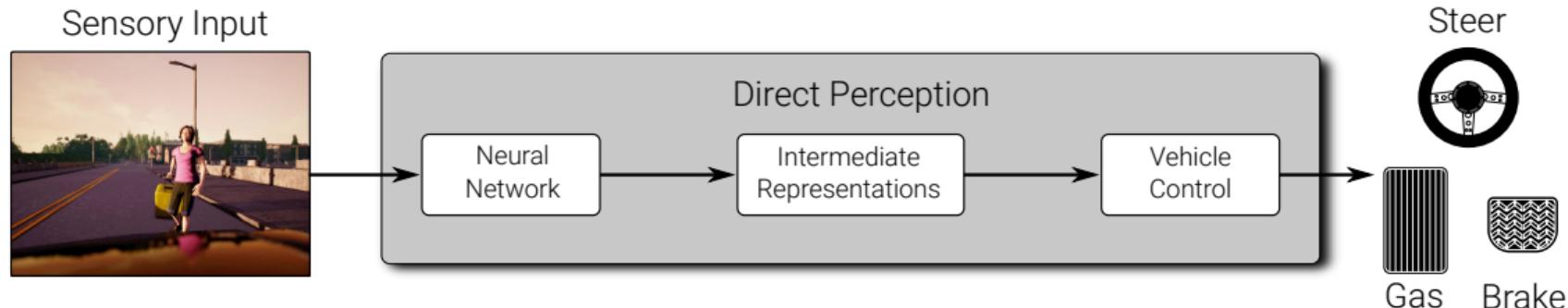


Examples:

- ▶ [Chen et al., ICCV 2015]
- ▶ [Sauer et al., CoRL 2018]
- ▶ [Behl et al., IROS 2020]



Autonomous Driving: Direct Perception



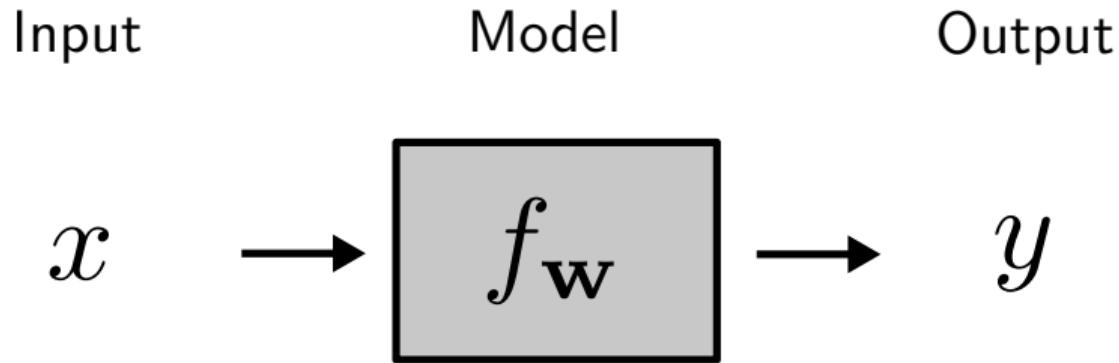
Pros:

- ▶ Compact Representation
- ▶ Interpretability

Cons:

- ▶ Control typically not learned jointly
- ▶ How to choose representations?

Supervised Learning



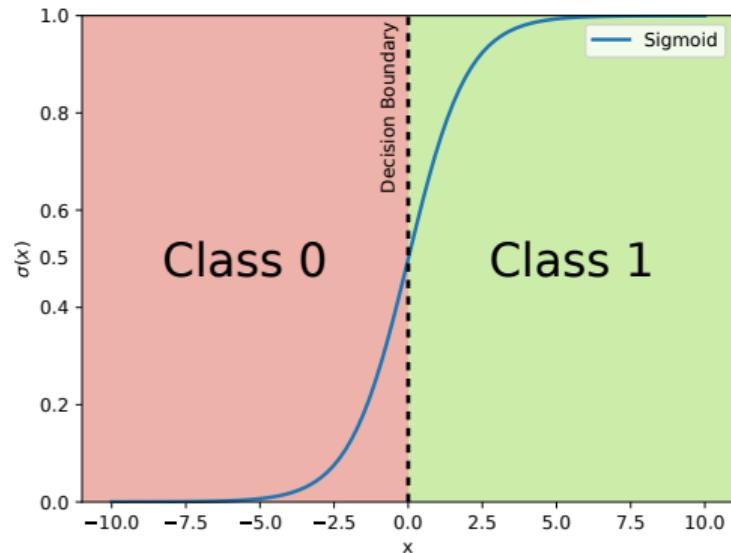
- ▶ **Learning:** Estimate parameters \mathbf{w} from training data $\{(x_i, y_i)\}_{i=1}^N$
- ▶ **Inference:** Make novel predictions: $y = f_{\mathbf{w}}(x)$

Linear Classification

Logistic Regression

$$\hat{y} = \sigma(\mathbf{w}^\top \mathbf{x} + w_0) \quad \sigma(x) = \frac{1}{1 + e^{-x}}$$

- ▶ Let $\mathbf{x} \in \mathbb{R}^2$
- ▶ Decision boundary: $\mathbf{w}^\top \mathbf{x} + w_0 = 0$
- ▶ Decide for class 1 $\Leftrightarrow \mathbf{w}^\top \mathbf{x} > -w_0$
- ▶ Decide for class 0 $\Leftrightarrow \mathbf{w}^\top \mathbf{x} < -w_0$
- ▶ Which problems can we solve?



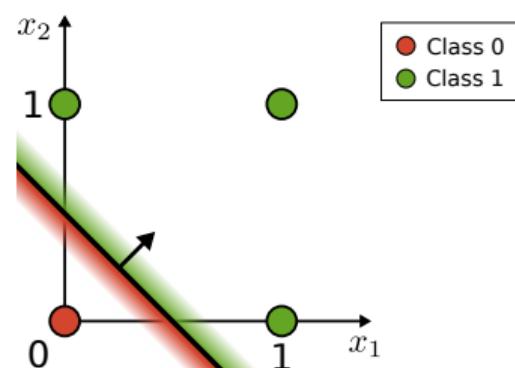
Linear Classification

Linear Classifier:

$$\text{Class 1} \Leftrightarrow \mathbf{w}^\top \mathbf{x} > -w_0$$

$$\underbrace{\begin{pmatrix} 1 & 1 \end{pmatrix}}_{\mathbf{w}^\top} \underbrace{\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}}_{\mathbf{x}} > \underbrace{-w_0}_{0.5}$$

x_1	x_2	$\text{OR}(x_1, x_2)$
0	0	0
0	1	1
1	0	1
1	1	1



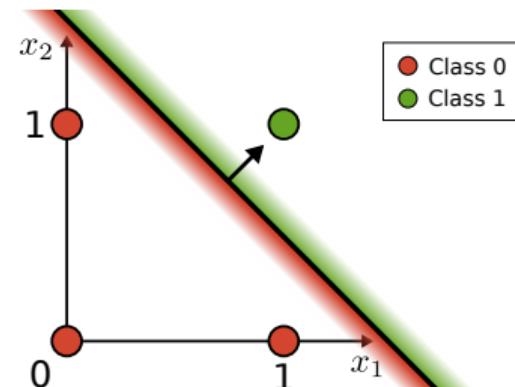
Linear Classification

Linear Classifier:

$$\text{Class 1} \Leftrightarrow \mathbf{w}^\top \mathbf{x} > -w_0$$

$$\underbrace{\begin{pmatrix} 1 & 1 \end{pmatrix}}_{\mathbf{w}^\top} \underbrace{\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}}_{\mathbf{x}} > \underbrace{-w_0}_{1.5}$$

x_1	x_2	$\text{AND}(x_1, x_2)$
0	0	0
0	1	0
1	0	0
1	1	1



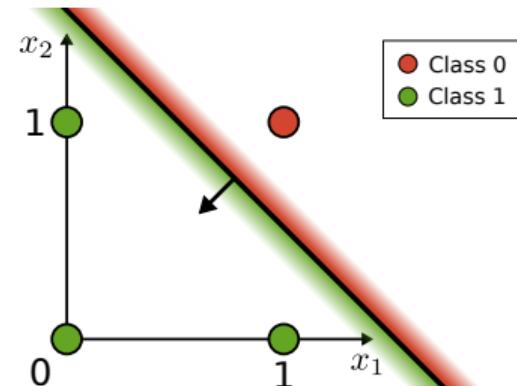
Linear Classification

Linear Classifier:

$$\text{Class 1} \Leftrightarrow \mathbf{w}^\top \mathbf{x} > -w_0$$

$$\underbrace{\begin{pmatrix} -1 & -1 \end{pmatrix}}_{\mathbf{w}^\top} \underbrace{\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}}_{\mathbf{x}} > \underbrace{-1.5}_{-w_0}$$

x_1	x_2	$\text{NAND}(x_1, x_2)$
0	0	1
0	1	1
1	0	1
1	1	0



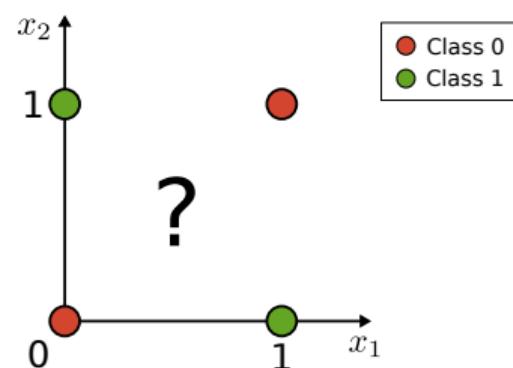
Linear Classification

Linear Classifier:

$$\text{Class 1} \Leftrightarrow \mathbf{w}^\top \mathbf{x} > -w_0$$

$$\underbrace{\begin{pmatrix} ? & ? \end{pmatrix}}_{\mathbf{w}^\top} \underbrace{\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}}_{\mathbf{x}} > \underbrace{-w_0}_{?}$$

x_1	x_2	$\text{XOR}(x_1, x_2)$
0	0	0
0	1	1
1	0	1
1	1	0

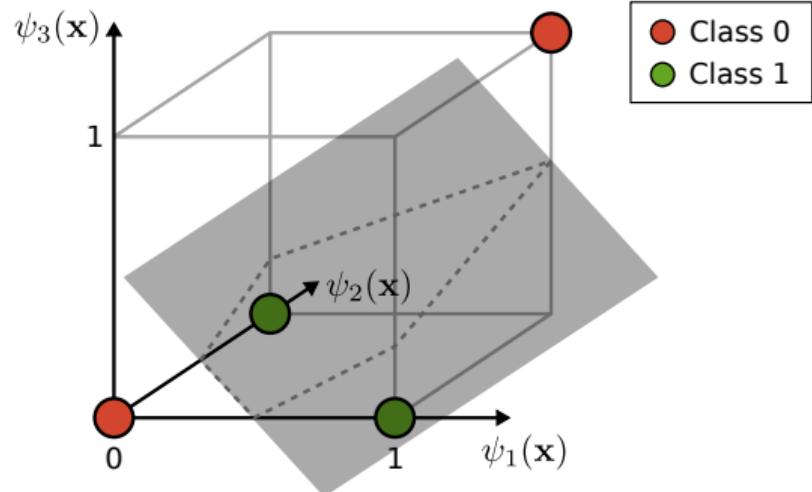


Linear Classification

**Linear classifier with
non-linear features ψ :**

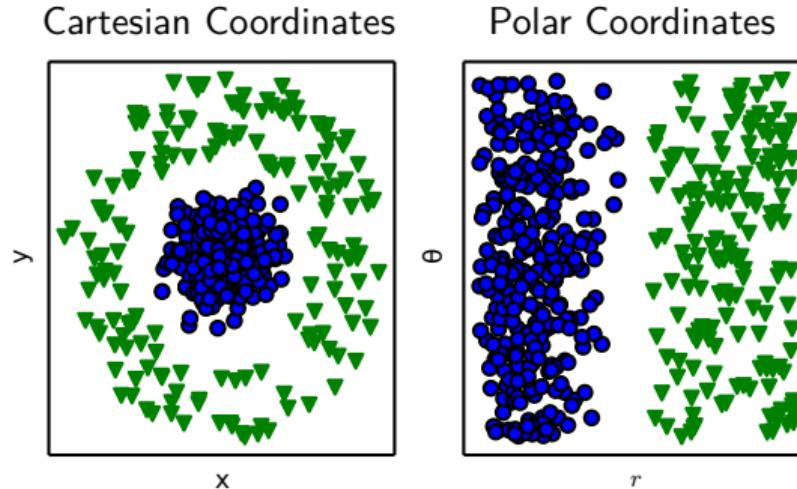
$$\mathbf{w}^\top \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_1x_2 \end{pmatrix}}_{\psi(\mathbf{x})} > -w_0$$

x_1	x_2	$\psi_1(\mathbf{x})$	$\psi_2(\mathbf{x})$	$\psi_3(\mathbf{x})$	XOR
0	0	0	0	0	0
0	1	0	1	0	1
1	0	1	0	0	1
1	1	1	1	1	0



- Non-linear features allow linear classifier to solve non-linear classification problems!

Representation Matters



- ▶ But how to choose the transformation? Can be very hard in practice.
- ▶ Yet, this was the dominant approach until the 2000s (vision, speech, ..)
- ▶ In deep/representation learning we want to learn these transformations

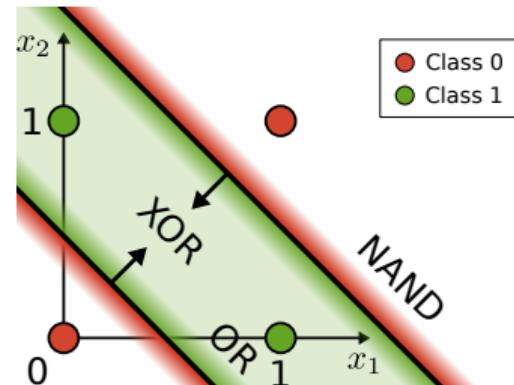
Non-Linear Classification

Linear Classifier:

$$\text{Class 1} \Leftrightarrow \mathbf{w}^\top \mathbf{x} > -w_0$$

x_1	x_2	$\text{XOR}(x_1, x_2)$
0	0	0
0	1	1
1	0	1
1	1	0

$$\begin{aligned}\text{XOR}(x_1, x_2) = \\ \text{AND}(\text{OR}(x_1, x_2), \text{NAND}(x_1, x_2))\end{aligned}$$



Non-Linear Classification

$$\text{XOR}(x_1, x_2) = \text{AND}(\text{OR}(x_1, x_2), \text{NAND}(x_1, x_2))$$

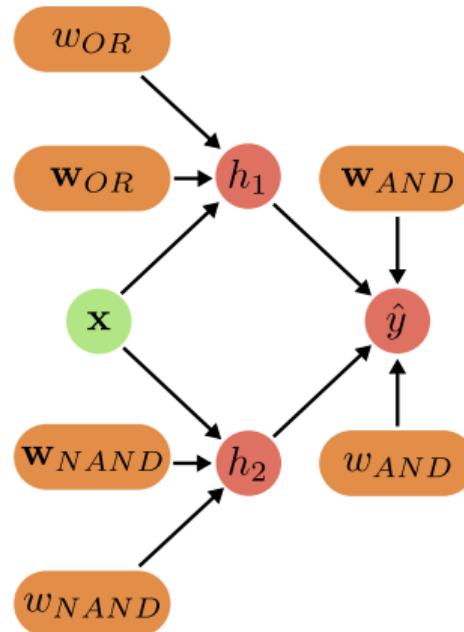
The above expression can be rewritten
as a **program of logistic regressors**:

$$h_1 = \sigma(\mathbf{w}_{OR}^\top \mathbf{x} + w_{OR})$$

$$h_2 = \sigma(\mathbf{w}_{NAND}^\top \mathbf{x} + w_{NAND})$$

$$\hat{y} = \sigma(\mathbf{w}_{AND}^\top \mathbf{h} + w_{AND})$$

Note that $\mathbf{h}(\mathbf{x})$ is a non-linear feature of \mathbf{x} .
We call $\mathbf{h}(\mathbf{x})$ a **hidden** layer.



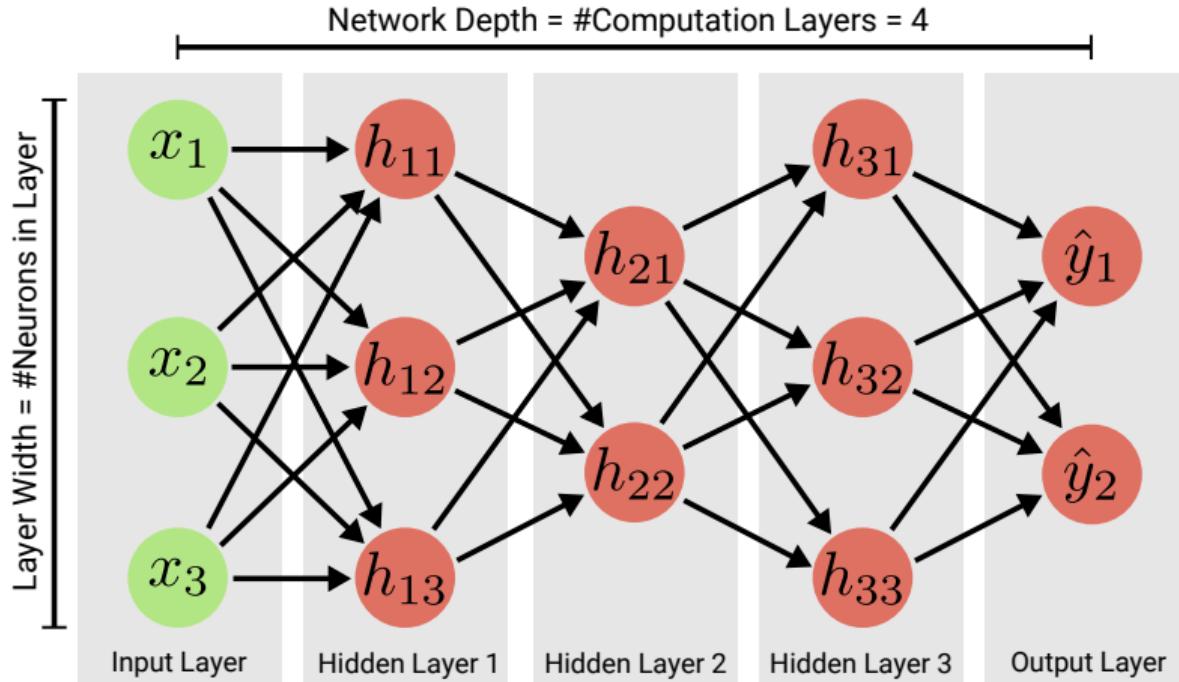
Multi-Layer Perceptrons

- ▶ MLPs are **feedforward** neural networks (no feedback connections)
- ▶ They **compose** several non-linear functions $\mathbf{f}(\mathbf{x}) = \hat{\mathbf{y}}(\mathbf{h}_3(\mathbf{h}_2(\mathbf{h}_1(\mathbf{x}))))$ where $\mathbf{h}_i(\cdot)$ are called **hidden layers** and $\hat{\mathbf{y}}(\cdot)$ is the **output layer**
- ▶ The data specifies only the behavior of the output layer (thus the name “hidden”)
- ▶ Each layer i comprises multiple **neurons** j which are implemented as **affine transformations** ($\mathbf{a}^\top \mathbf{x} + \mathbf{b}$) followed by non-linear **activation functions** (g):

$$h_{ij} = g(\mathbf{a}_{ij}^\top \mathbf{h}_{i-1} + \mathbf{b}_{ij})$$

- ▶ Each neuron in each layer is **fully connected** to all neurons of the previous layer
- ▶ The overall length of the chain is the **depth** of the model \Rightarrow “Deep Learning”

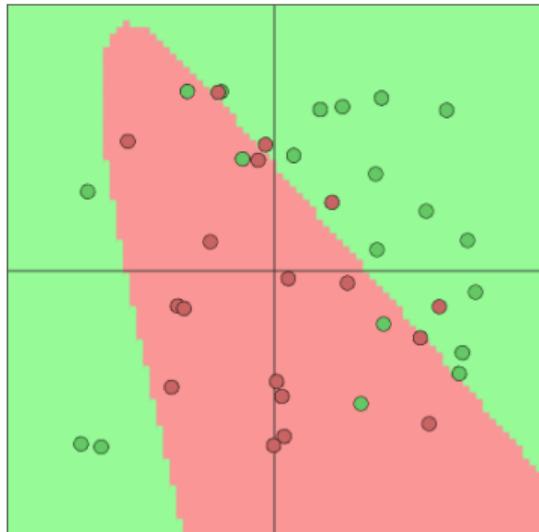
MLP Network Architecture



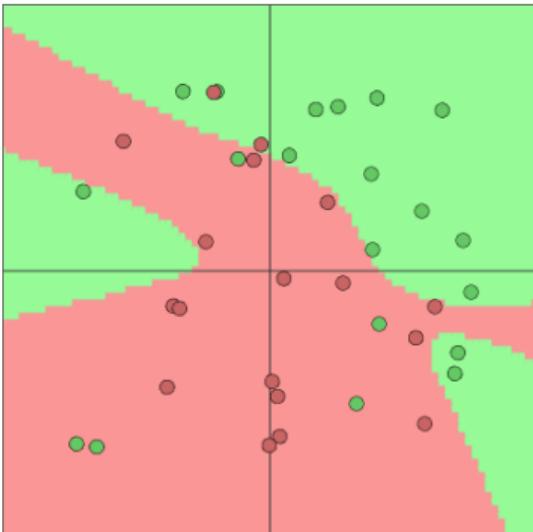
- ▶ Neurons are grouped into layers, each neuron **fully connected** to all prev. ones
- ▶ **Hidden layer** $\mathbf{h}_i = g(\mathbf{A}_i \mathbf{h}_{i-1} + \mathbf{b}_i)$ with **activation function** $g(\cdot)$ and weights $\mathbf{A}_i, \mathbf{b}_i$

Deeper Models allow for more Complex Decisions

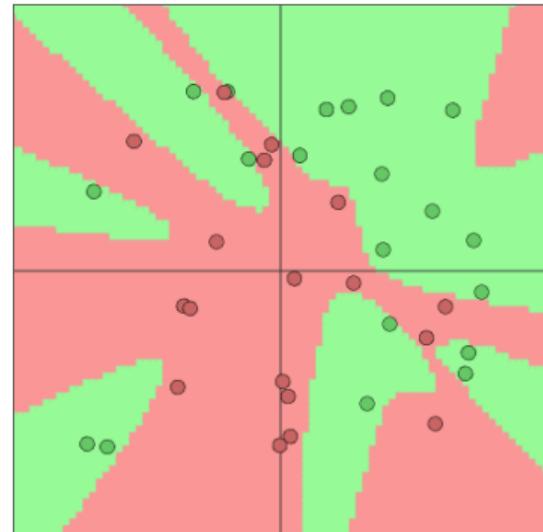
2 Hidden Neurons



5 Hidden Neurons

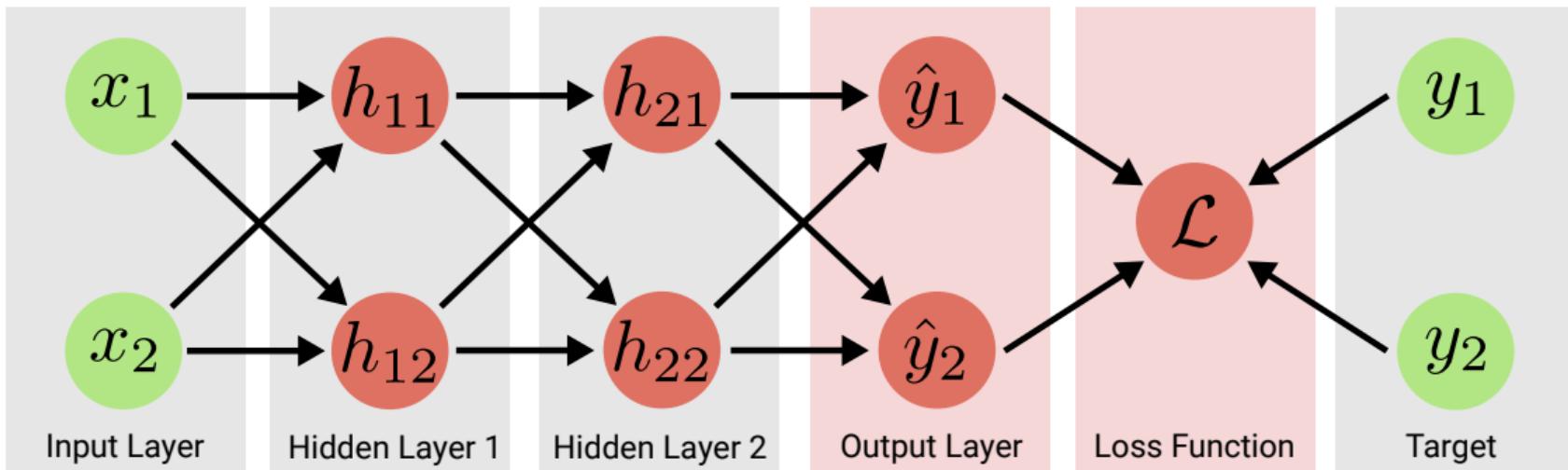


15 Hidden Neurons



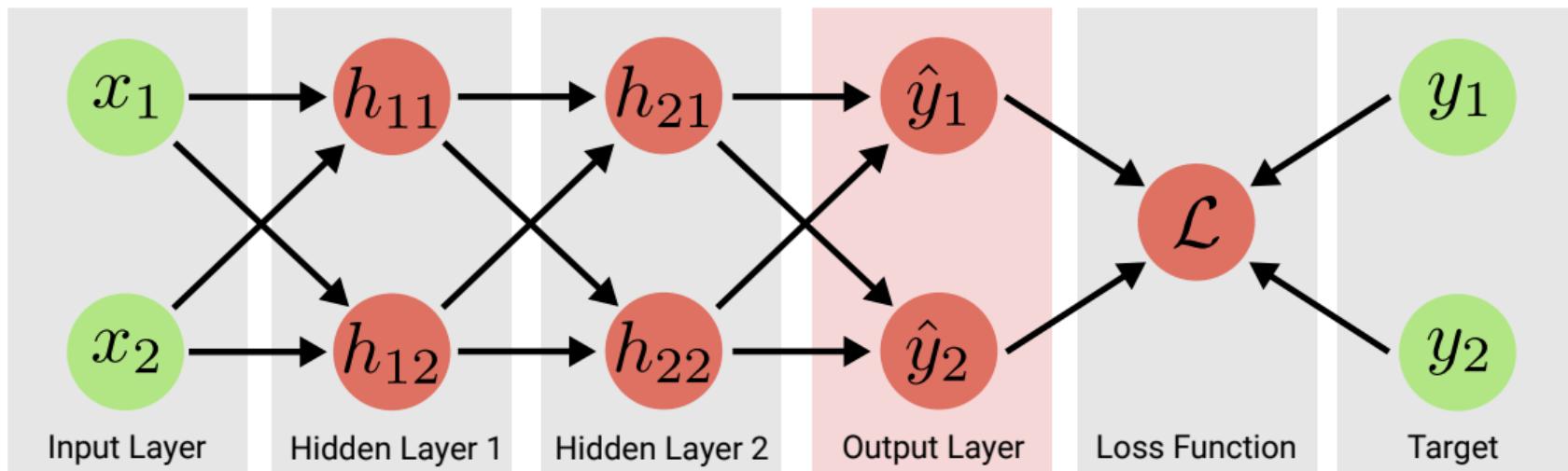
<https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html>

Output and Loss Functions



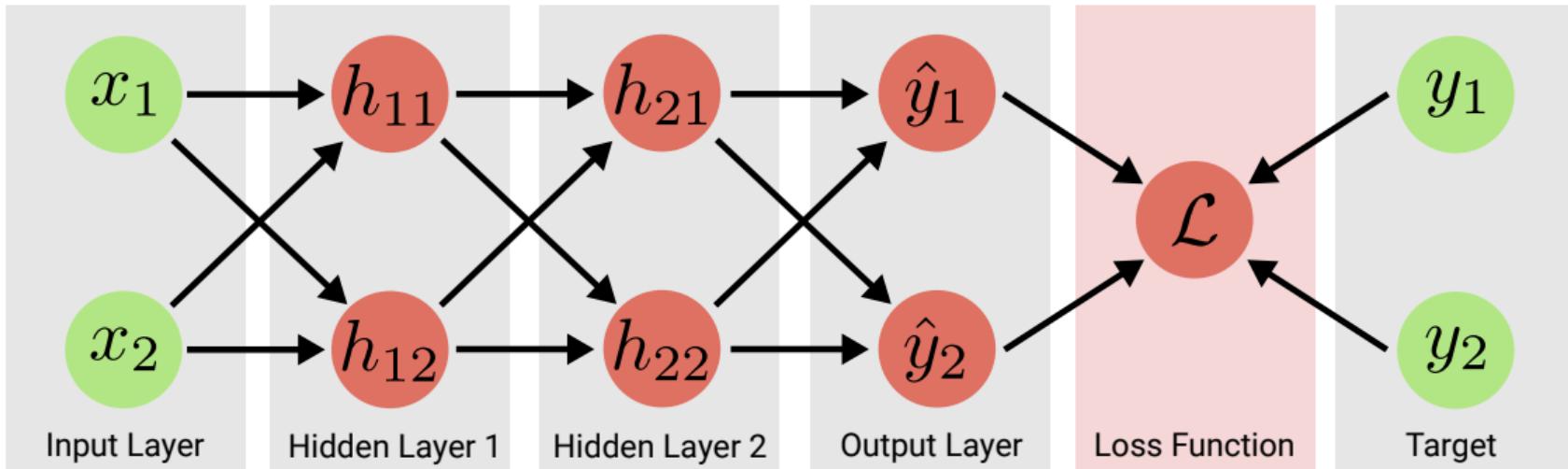
- The **output layer** is the last layer in a neural network which computes the output
- The **loss function** compares the result of the output layer to the target value(s)
- Choice of output layer and loss function depends on task (discrete, continuous, ..)

Output Layer



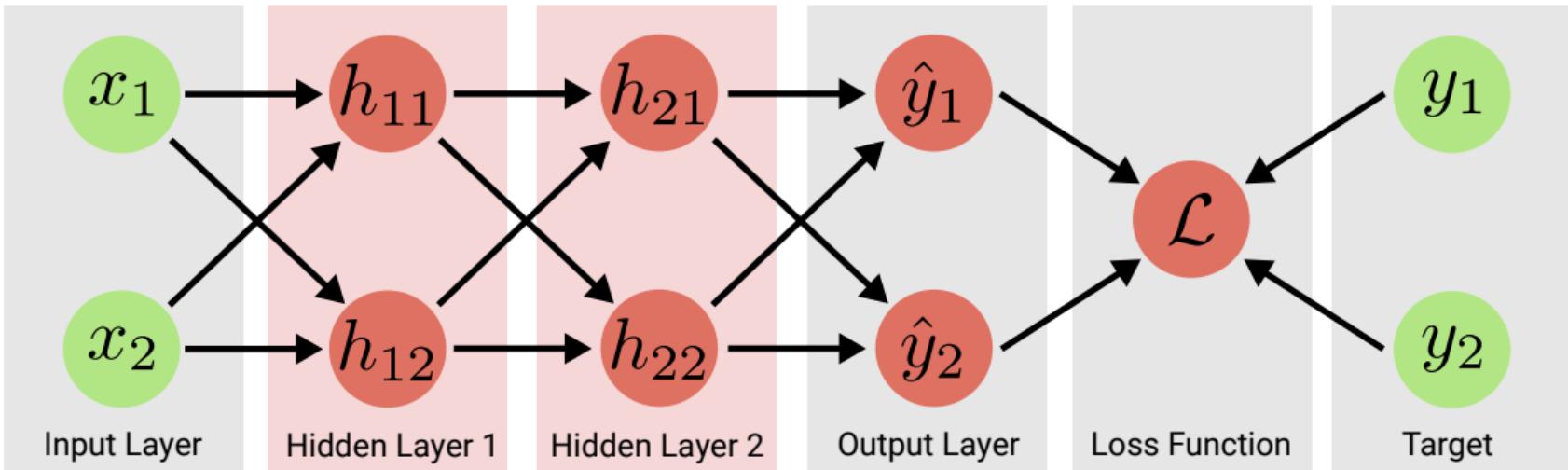
- ▶ For classification problems, we use a sigmoid or softmax non-linearity
- ▶ For regression problems, we can directly return the value after the last layer

Loss Function



- ▶ For classification problems, we use the (binary) cross-entropy loss
- ▶ For regression problems, we can use the ℓ_1 or ℓ_2 loss

Activation Functions

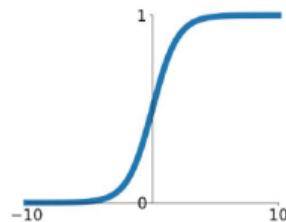


- ▶ Hidden layer $\mathbf{h}_i = g(\mathbf{A}_i \mathbf{h}_{i-1} + \mathbf{b}_i)$ with **activation function** $g(\cdot)$ and weights $\mathbf{A}_i, \mathbf{b}_i$
- ▶ The activation function is frequently applied **element-wise** to its input
- ▶ Activation functions must be **non-linear** to learn non-linear mappings

Activation Functions

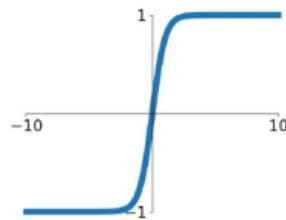
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



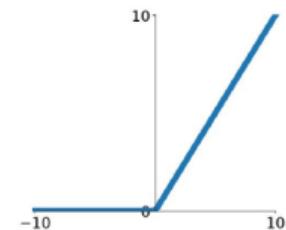
tanh

$$\tanh(x)$$



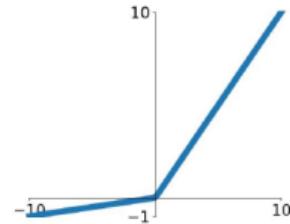
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

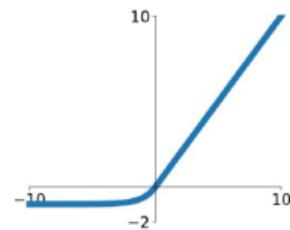


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

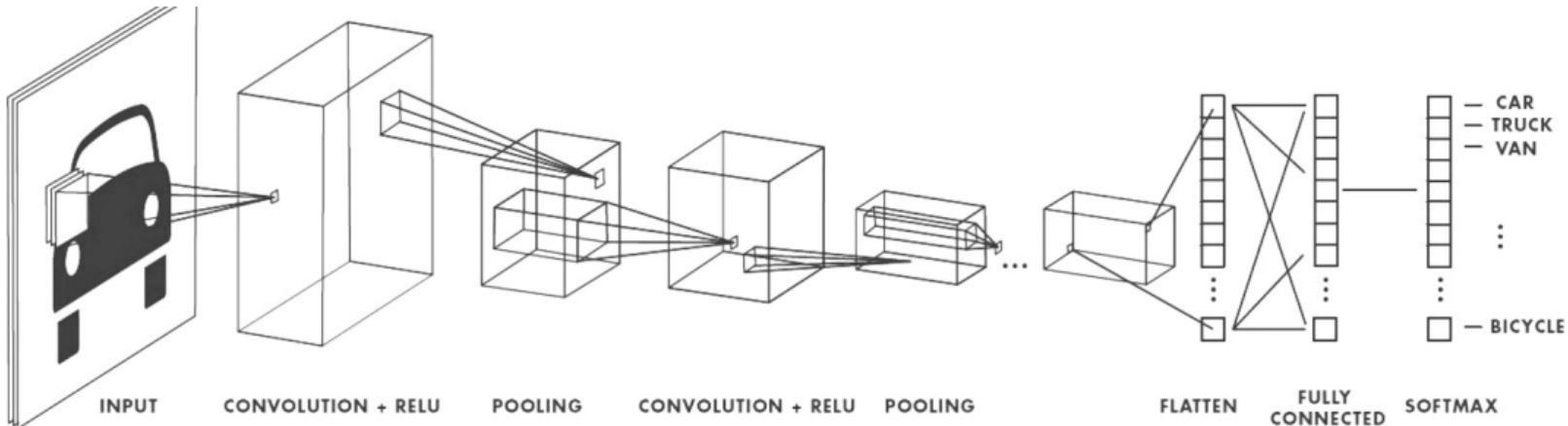
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



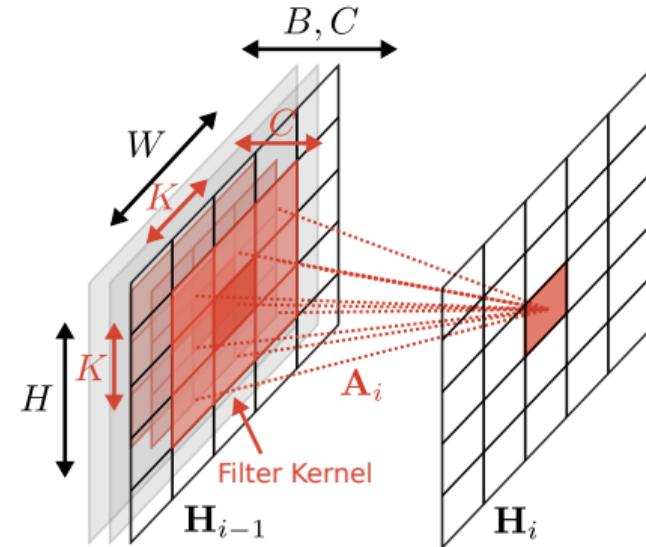
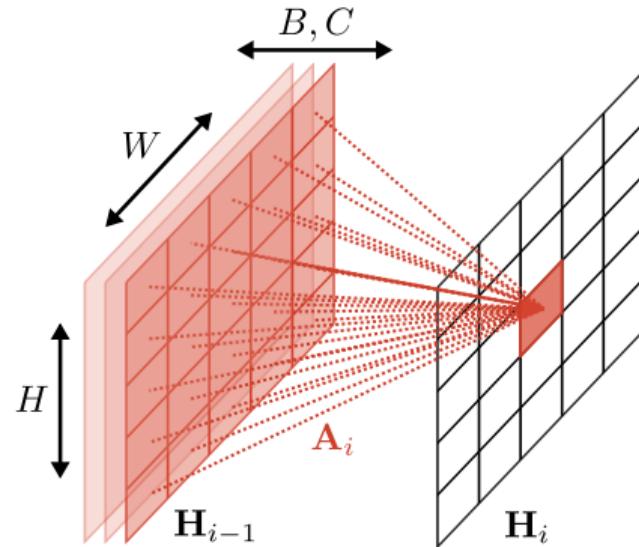
Convolutional Neural Networks

Convolutional Neural Networks



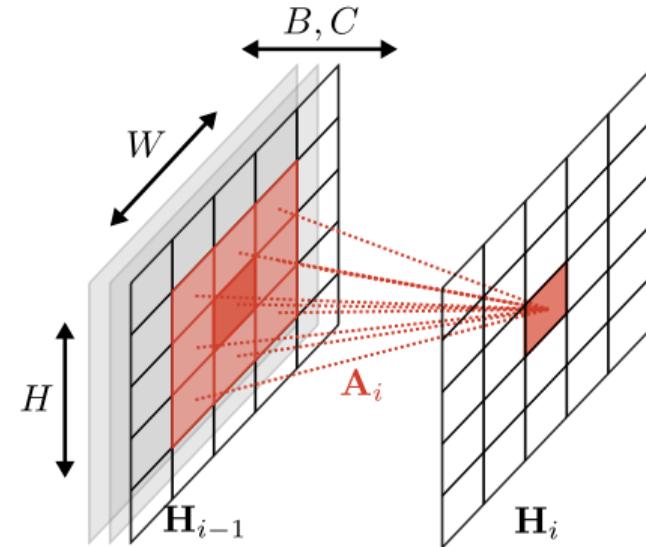
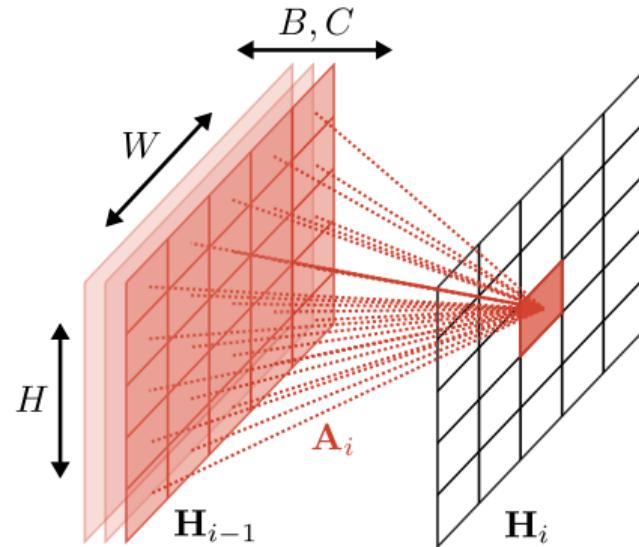
- ▶ Multi-layer perceptrons don't scale to high-dimensional inputs
- ▶ ConvNets represent data in 3 dimensions: width, height, depth (= feature maps)
- ▶ ConvNets interleave discrete convolutions, non-linearities and pooling
- ▶ Key ideas: sparse interactions, parameter sharing, equivariant representation

Fully Connected vs. Convolutional Layers



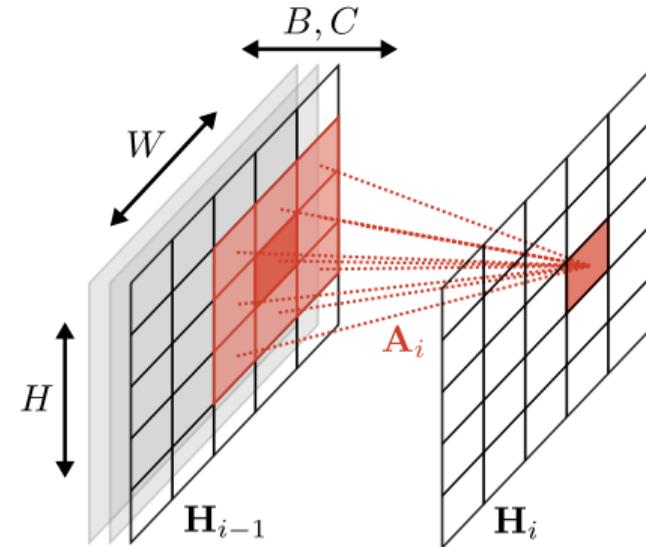
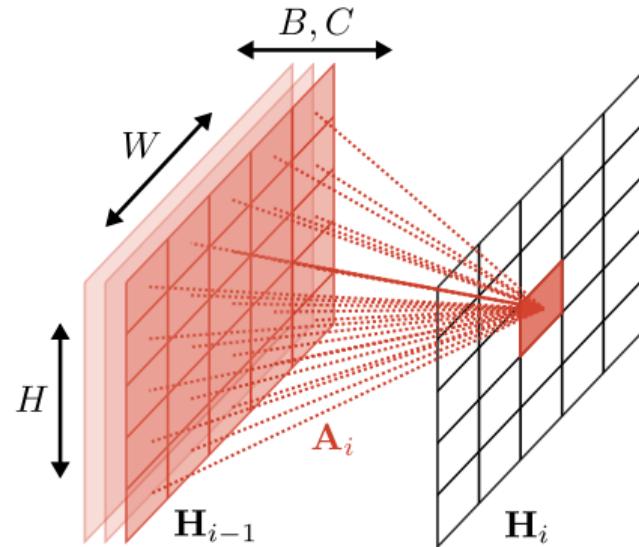
- **Fully connected layer:** #Weights = $W \times H \times C_{out} \times (W \times H \times C_{in} + 1)$
- **Convolutional layer:** #Weights = $C_{out} \times (K \times K \times C_{in} + 1)$ ("weight sharing")
- With C_{in} input and C_{out} output channels, layer size $W \times H$ and kernel size $K \times K$
- Convolutions are followed by non-linear activation functions (e.g., ReLU)

Fully Connected vs. Convolutional Layers



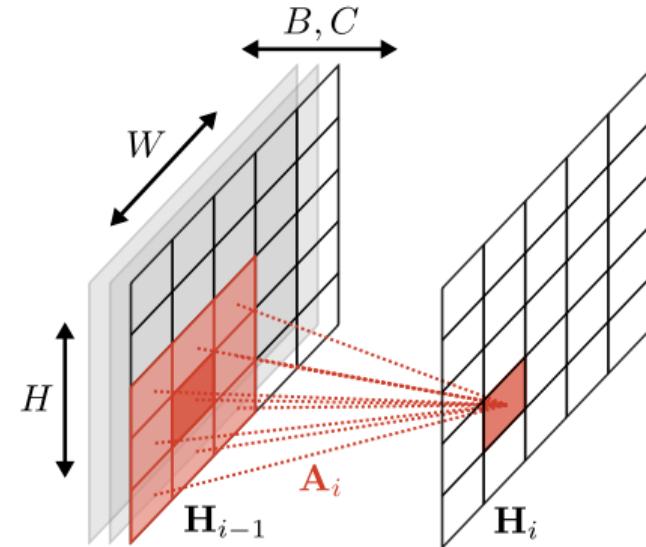
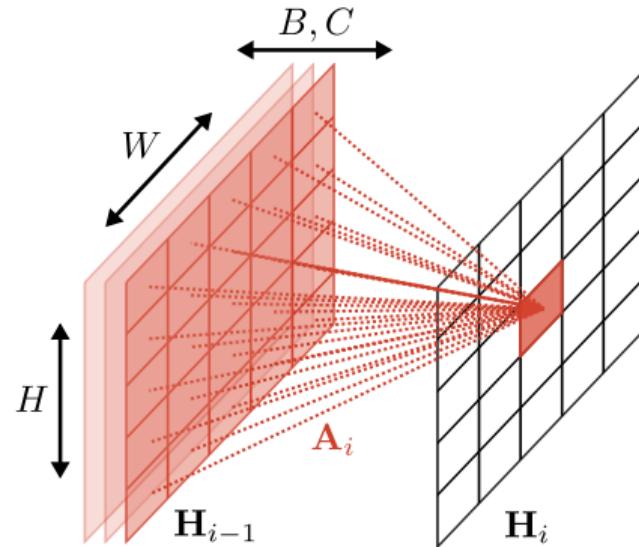
- **Fully connected layer:** #Weights = $W \times H \times C_{out} \times (W \times H \times C_{in} + 1)$
- **Convolutional layer:** #Weights = $C_{out} \times (K \times K \times C_{in} + 1)$ ("weight sharing")
- With C_{in} input and C_{out} output channels, layer size $W \times H$ and kernel size $K \times K$
- Convolutions are followed by non-linear activation functions (e.g., ReLU)

Fully Connected vs. Convolutional Layers



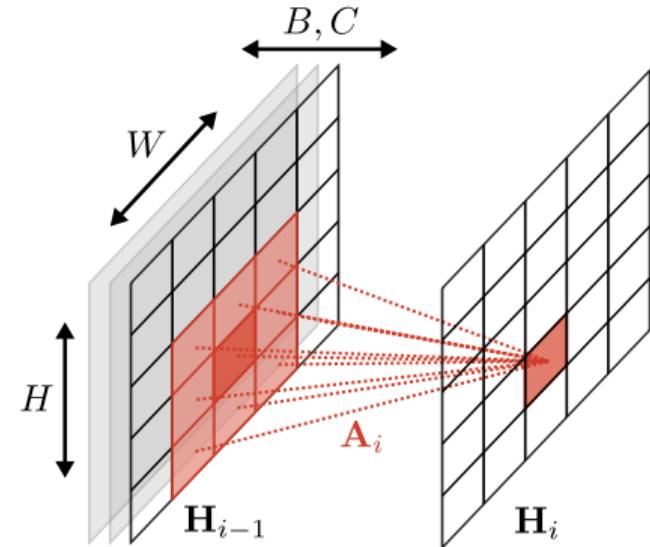
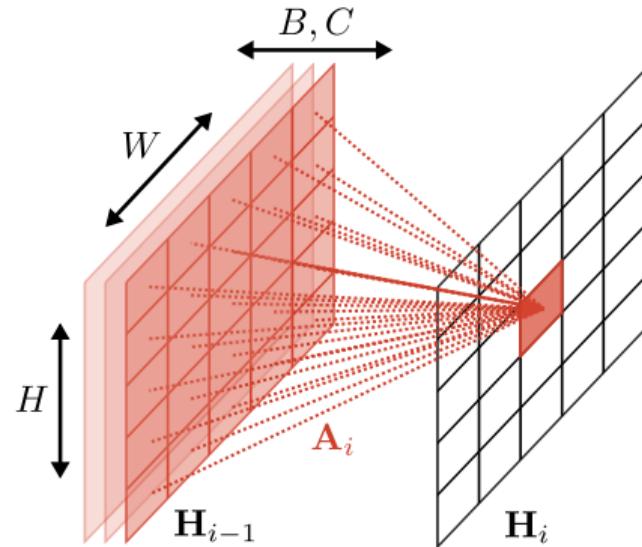
- **Fully connected layer:** #Weights = $W \times H \times C_{out} \times (W \times H \times C_{in} + 1)$
- **Convolutional layer:** #Weights = $C_{out} \times (K \times K \times C_{in} + 1)$ ("weight sharing")
- With C_{in} input and C_{out} output channels, layer size $W \times H$ and kernel size $K \times K$
- Convolutions are followed by non-linear activation functions (e.g., ReLU)

Fully Connected vs. Convolutional Layers



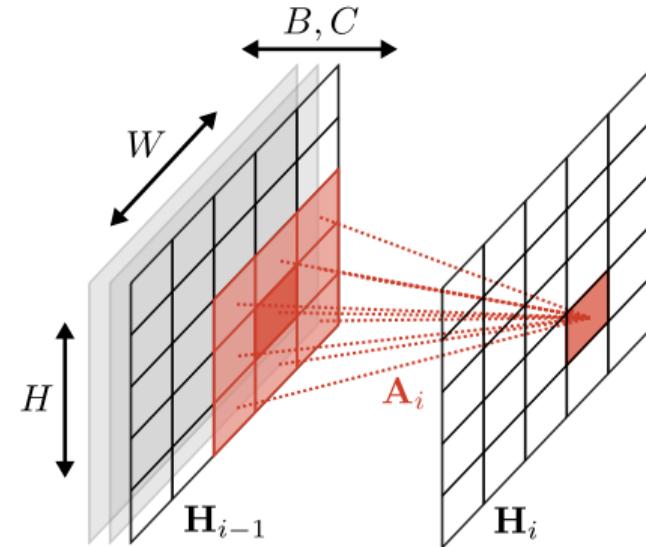
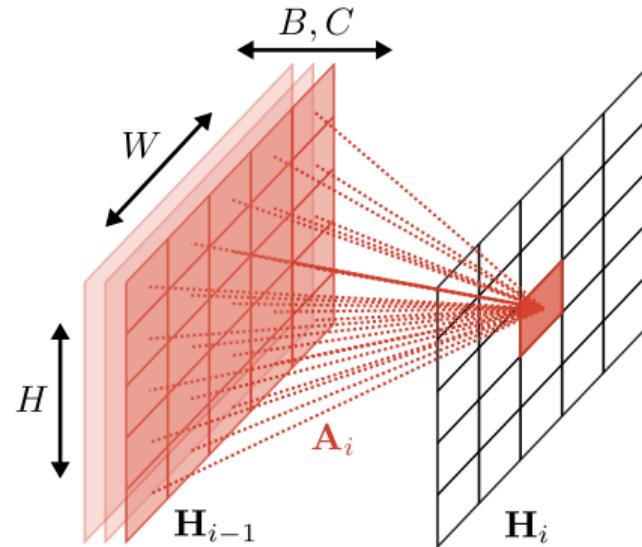
- ▶ **Fully connected layer:** #Weights = $W \times H \times C_{out} \times (W \times H \times C_{in} + 1)$
- ▶ **Convolutional layer:** #Weights = $C_{out} \times (K \times K \times C_{in} + 1)$ ("weight sharing")
- ▶ With C_{in} input and C_{out} output channels, layer size $W \times H$ and kernel size $K \times K$
- ▶ Convolutions are followed by non-linear activation functions (e.g., ReLU)

Fully Connected vs. Convolutional Layers



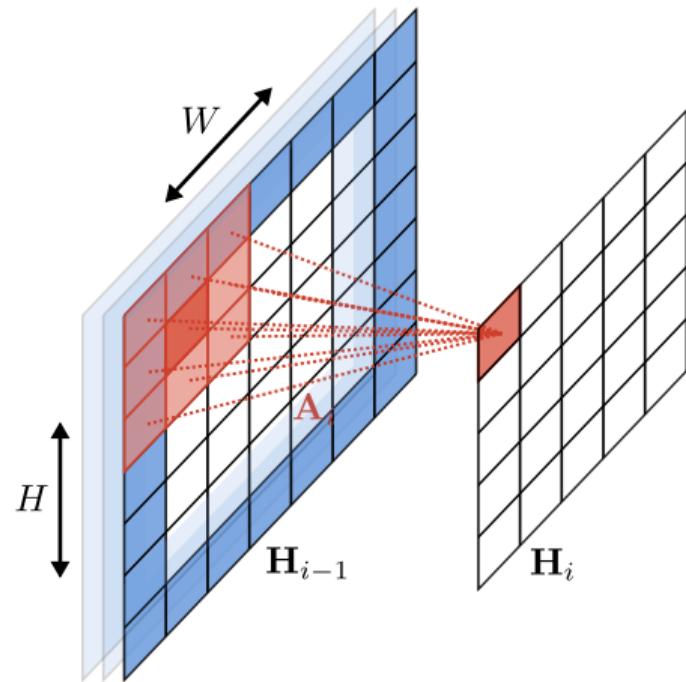
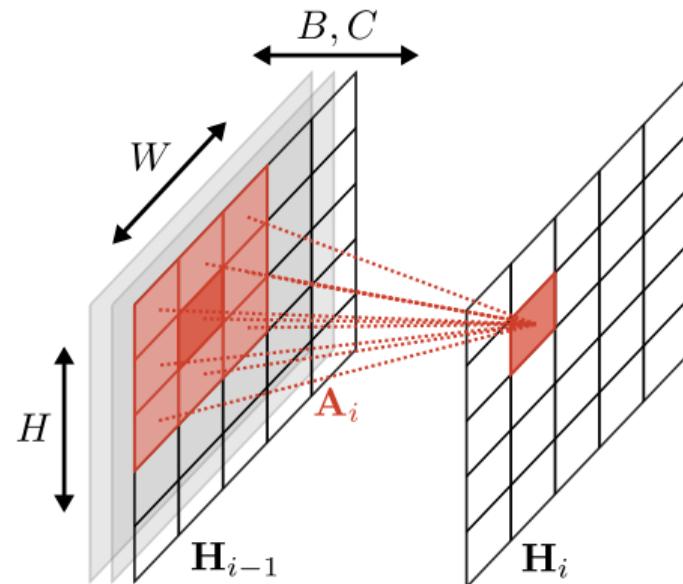
- **Fully connected layer:** #Weights = $W \times H \times C_{out} \times (W \times H \times C_{in} + 1)$
- **Convolutional layer:** #Weights = $C_{out} \times (K \times K \times C_{in} + 1)$ ("weight sharing")
- With C_{in} input and C_{out} output channels, layer size $W \times H$ and kernel size $K \times K$
- Convolutions are followed by non-linear activation functions (e.g., ReLU)

Fully Connected vs. Convolutional Layers



- **Fully connected layer:** #Weights = $W \times H \times C_{out} \times (W \times H \times C_{in} + 1)$
- **Convolutional layer:** #Weights = $C_{out} \times (K \times K \times C_{in} + 1)$ ("weight sharing")
- With C_{in} input and C_{out} output channels, layer size $W \times H$ and kernel size $K \times K$
- Convolutions are followed by non-linear activation functions (e.g., ReLU)

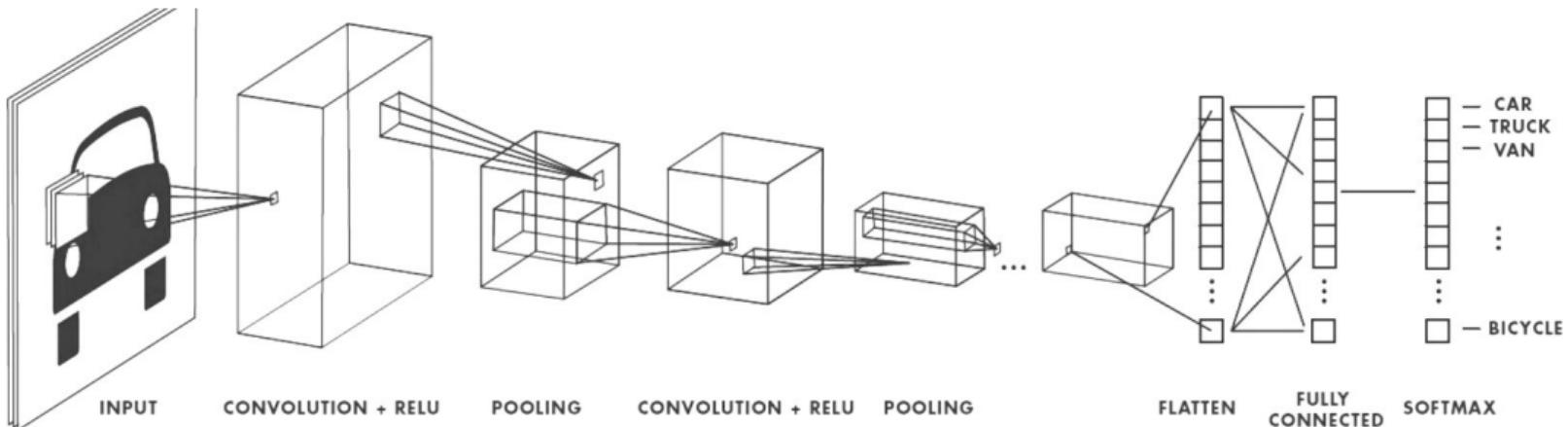
Padding



Idea of Padding:

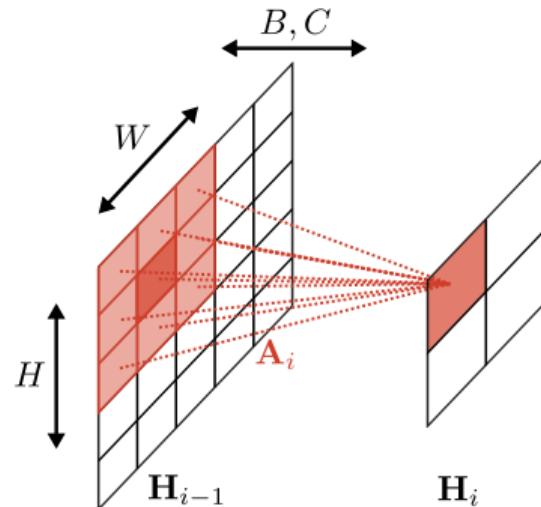
- ▶ Add boundary of appropriate size with zeros (blue) around input tensor

Downsampling



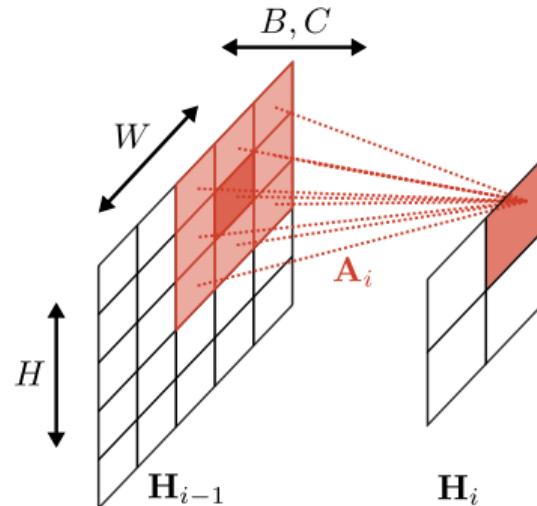
- Downsampling **reduces the spatial resolution** (e.g., for image level predictions)
- Downsampling **increases the receptive field** (which pixels influence a neuron)

Pooling



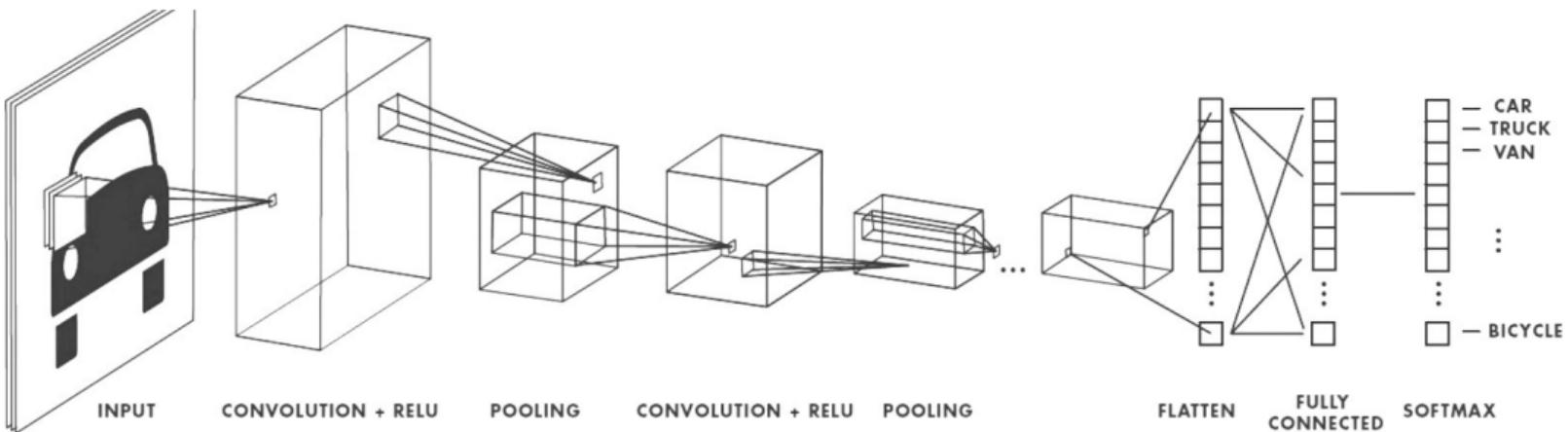
- ▶ Typically, stride $s = 2$ and kernel size $2 \times 2 \Rightarrow$ **reduces spatial dimensions by 2**
- ▶ Pooling has **no parameters** (typical pooling operations: max, min, mean)

Pooling



- ▶ Typically, stride $s = 2$ and kernel size $2 \times 2 \Rightarrow$ **reduces spatial dimensions by 2**
- ▶ Pooling has **no parameters** (typical pooling operations: max, min, mean)

Fully Connected Layers



- ▶ Often, convolutional networks comprise fully connected layers at the end

Optimization

Optimization

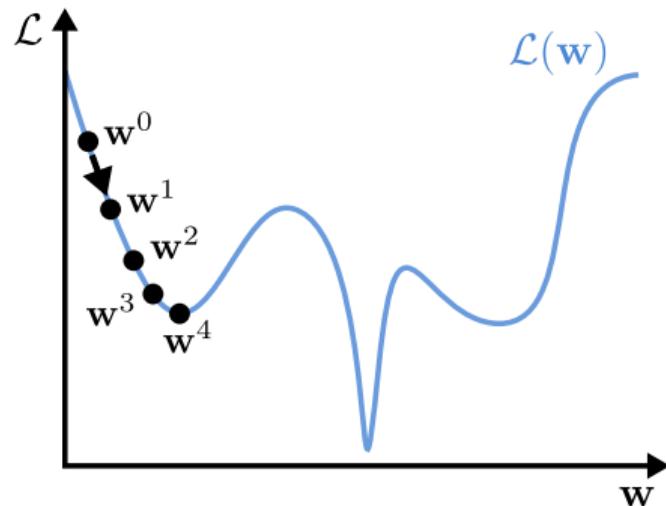
Optimization Problem: (dataset \mathcal{X})

$$\mathbf{w}^* = \operatorname{argmin}_{\mathbf{w}} \mathcal{L}(\mathcal{X}, \mathbf{w})$$

Gradient Descent:

$$\mathbf{w}^0 = \mathbf{w}^{\text{init}}$$

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla_{\mathbf{w}} \mathcal{L}(\mathcal{X}, \mathbf{w}^t)$$

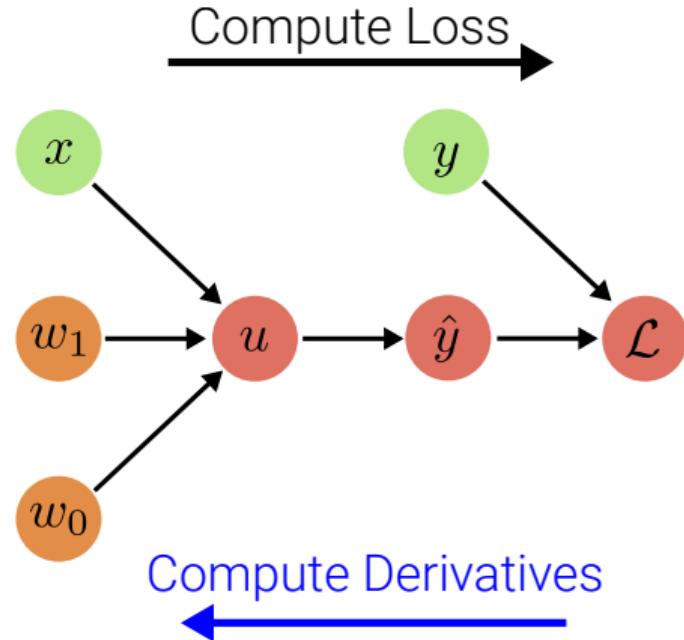


- Neural network loss $\mathcal{L}(\mathcal{X}, \mathbf{w})$ is **not convex**, we have to use gradient descent
- There exist **multiple local minima**, but we will find only one through optimization
- Good news: it is known that **many local minima** in deep networks **are good ones**

Backpropagation

- ▶ Values are efficiently computed forward, gradients backward
- ▶ Modularity: Each node must only “know” how to compute gradients wrt. its own arguments
- ▶ One fw/bw pass per data point:

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathcal{X}, \mathbf{w}) = \sum_{i=1}^N \underbrace{\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{y}_i, \mathbf{x}_i, \mathbf{w})}_{\text{Backpropagation}}$$



Gradient Descent

Algorithm:

1. Initialize weights \mathbf{w}^0 and pick learning rate η
2. For all data points $i \in \{1, \dots, N\}$ do:
 - 2.1 Forward propagate \mathbf{x}_i through network to calculate prediction $\hat{\mathbf{y}}_i$
 - 2.2 Backpropagate to obtain gradient $\nabla_{\mathbf{w}} \mathcal{L}_i(\mathbf{w}^t) \equiv \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}_i, \mathbf{y}_i, \mathbf{w}^t)$
3. Update gradients: $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{1}{N} \sum_i \nabla_{\mathbf{w}} \mathcal{L}_i(\mathbf{w}^t)$
4. If validation error decreases, go to step 2, otherwise stop

Challenges:

- ▶ Typically, millions of parameters $\Rightarrow \dim(\mathbf{w}) = 1$ million or more
- ▶ Typically, millions of training points $\Rightarrow N = 1$ million or more
- ▶ Becomes extremely expensive to compute and does not fit into memory

Stochastic Gradient Descent

Solution:

- The total loss over the entire training set can be expressed as the expectation:

$$\frac{1}{N} \sum_i \mathcal{L}_i(\mathbf{w}^t) = \mathbb{E}_{i \sim \mathcal{U}\{1, N\}} [\mathcal{L}_i(\mathbf{w}^t)]$$

- This expectation can be approximated by a smaller subset $B \ll N$ of the data:

$$\mathbb{E}_{i \sim \mathcal{U}\{1, N\}} [\mathcal{L}_i(\mathbf{w}^t)] \approx \frac{1}{B} \sum_b \mathcal{L}_b(\mathbf{w}^t)$$

- Thus, the gradient can also be approximated by this subset (=minibatch):

$$\frac{1}{N} \sum_i \nabla_{\mathbf{w}} \mathcal{L}_i(\mathbf{w}^t) \approx \frac{1}{B} \sum_b \nabla_{\mathbf{w}} \mathcal{L}_b(\mathbf{w}^t)$$

Stochastic Gradient Descent

Algorithm:

1. Initialize weights \mathbf{w}^0 , pick learning rate η and minibatch size $|\mathcal{X}_{\text{batch}}|$
2. Draw random minibatch $\{(\mathbf{x}_1, \mathbf{y}_1), \dots, (\mathbf{x}_B, \mathbf{y}_B)\} \subseteq \mathcal{X}$ (with $B \ll N$)
3. For all minibatch elements $b \in \{1, \dots, B\}$ do:
 - 3.1 Forward propagate \mathbf{x}_b through network to calculate prediction $\hat{\mathbf{y}}_b$
 - 3.2 Backpropagate to obtain batch element gradient $\nabla_{\mathbf{w}} \mathcal{L}_b(\mathbf{w}^t) \equiv \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}_b, \mathbf{y}_b, \mathbf{w}^t)$
4. Update gradients: $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{1}{B} \sum_b \nabla_{\mathbf{w}} \mathcal{L}_b(\mathbf{w}^t)$
5. If validation error decreases, go to step 2, otherwise stop

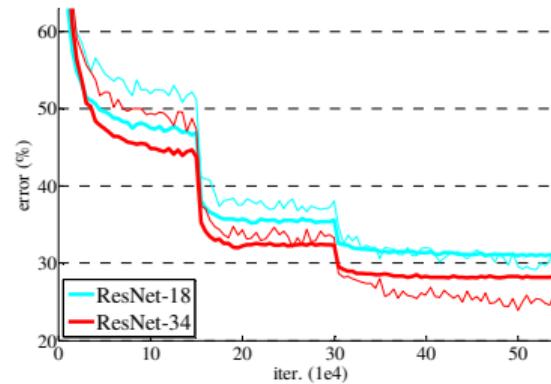
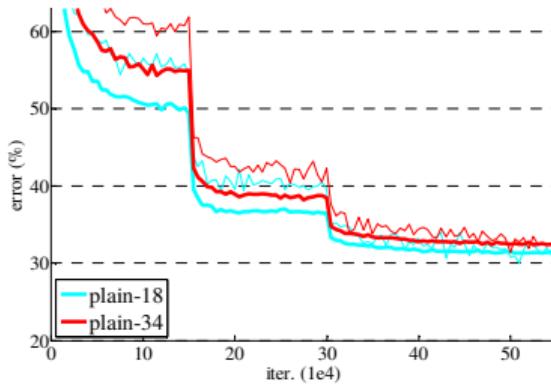
First-order Methods

There exist many variants:

- ▶ SGD
- ▶ SGD with Momentum
- ▶ SGD with Nesterov Momentum
- ▶ RMSprop
- ▶ Adagrad
- ▶ Adadelta
- ▶ Adam
- ▶ AdaMax
- ▶ NAdam
- ▶ AMSGrad

Adam is often the method of choice due to its robustness.

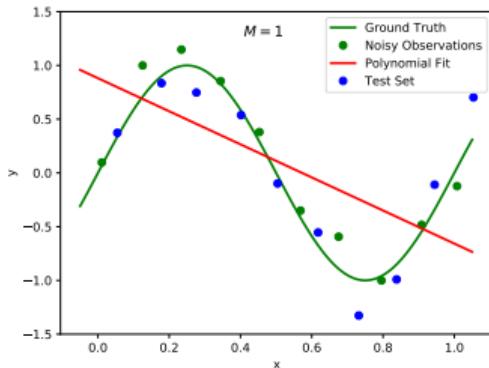
Learning Rate Schedules



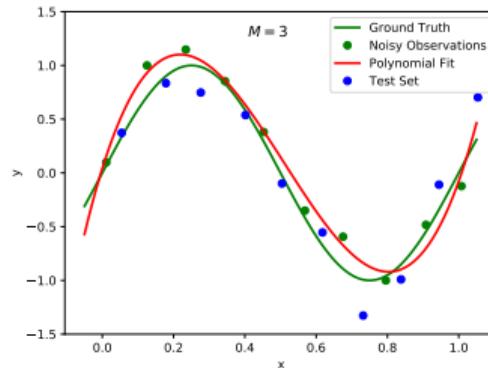
- ▶ A fixed learning rate is too slow in the beginning and too fast in the end
- ▶ Exponential decay: $\eta_t = \eta\alpha^t$
- ▶ Step decay: $\eta \leftarrow 0.5\eta$ (every K iterations)

Regularization

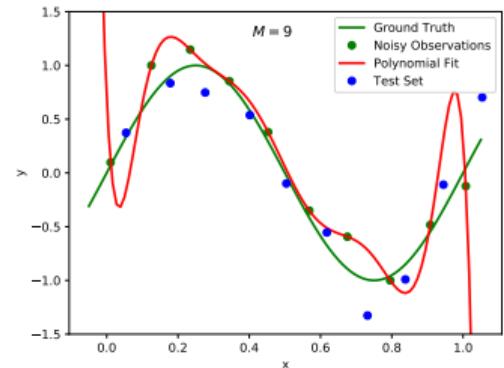
Capacity, Overfitting and Underfitting



Capacity too low



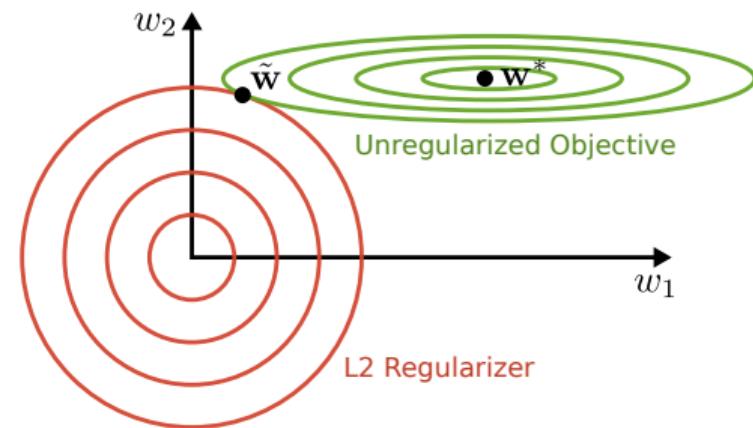
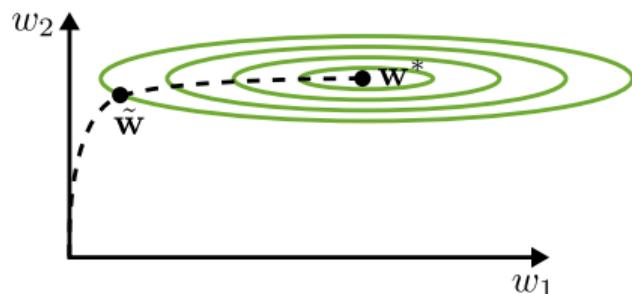
Capacity about right



Capacity too high

- ▶ **Underfitting:** Model too simple, does not achieve low error on training set
- ▶ **Overfitting:** Training error small, but test error (= generalization error) large
- ▶ **Regularization:** Take model from third regime (right) to second regime (middle)

Early Stopping and Parameter Penalties



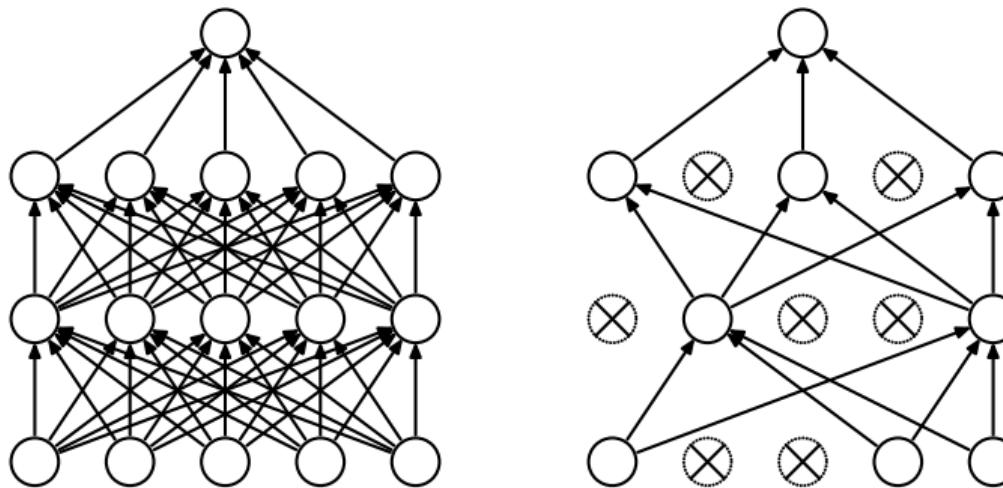
Early stopping:

- Dashed: Trajectory taken by SGD
- Trajectory stops at $\tilde{\mathbf{w}}$ before reaching minimum training error \mathbf{w}^*

L2 Regularization:

- Regularize objective with L_2 penalty
- Penalty forces minimum of regularized loss $\tilde{\mathbf{w}}$ closer to origin

Dropout

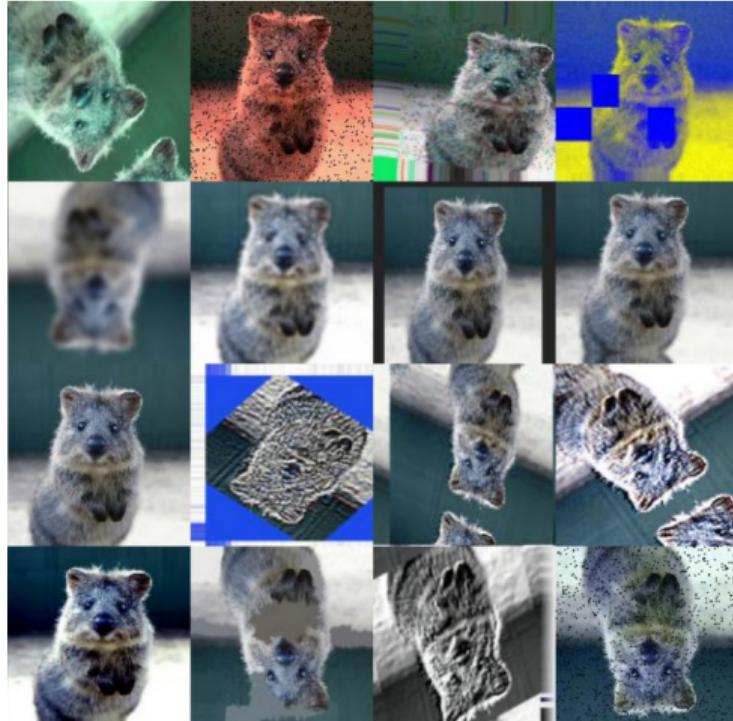


Idea:

- ▶ During training, **set neurons to zero** with probability μ (typically $\mu = 0.5$)
- ▶ Each binary mask is one model, changes randomly with every training iteration
- ▶ Creates **ensemble “on the fly”** from a single network with shared parameters

Data Augmentation

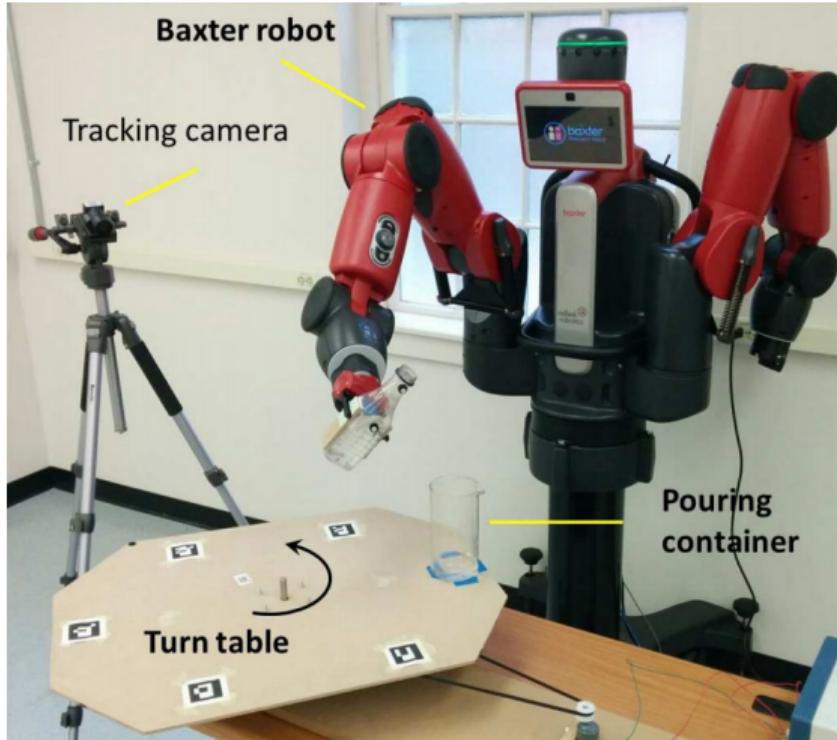
- ▶ Best way towards better generalization is to **train on more data**
- ▶ However, data in practice often limited
- ▶ Goal of data augmentation: create **“fake” data** from the existing data (on the fly) and add it to the training set
- ▶ New data must **preserve semantics**
- ▶ Even **simple operations** like translation or adding per-pixel noise often already greatly improve generalization
- ▶ <https://github.com/aleju/imgaug>



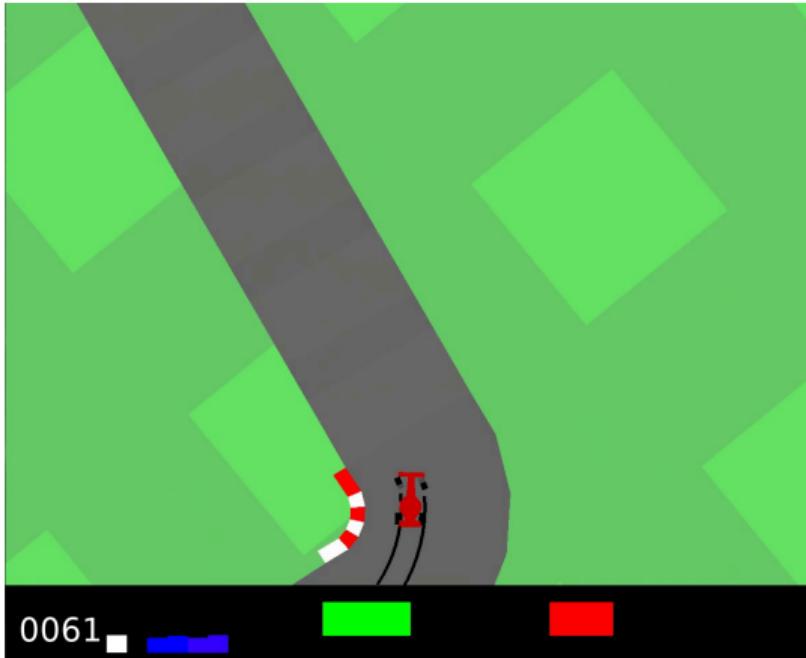
2.3

Imitation Learning

Imitation Learning: Manipulation



Imitation Learning: Car Racing

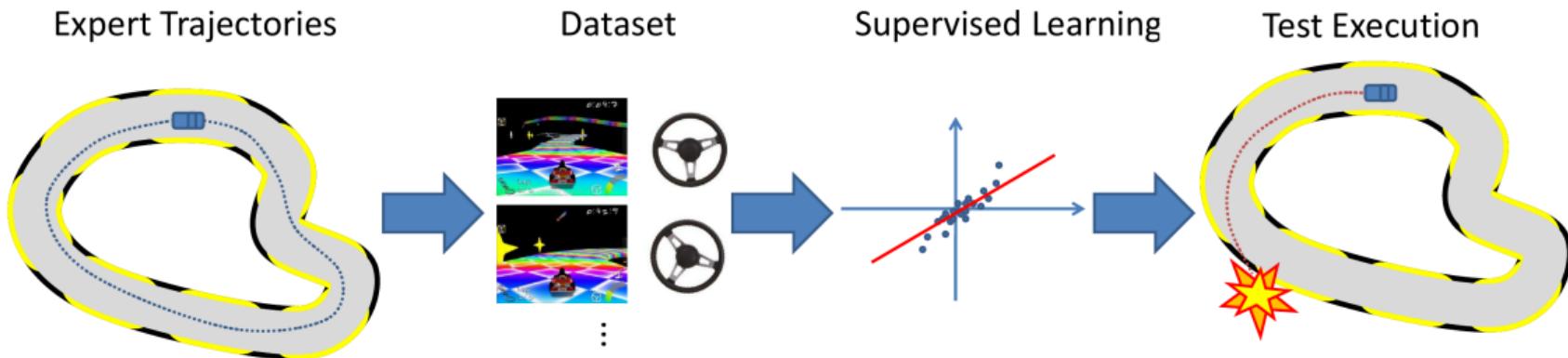


Trainer
(Human Driver)



Trainee
(Neural Network)

Imitation Learning in a Nutshell



Hard coding policies is often difficult \Rightarrow Rather use a data-driven approach!

- ▶ **Given:** demonstrations or demonstrator
- ▶ **Goal:** train a policy to mimic decision
- ▶ **Variants:** behavior cloning (this lecture), inverse optimal control, ...

Formal Definition of Imitation Learning

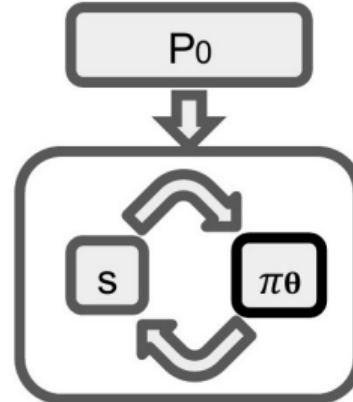
- ▶ State: $s \in \mathcal{S}$ may be partially observed (e.g., game screen)
- ▶ Action: $a \in \mathcal{A}$ may be discrete or continuous (e.g., turn angle, speed)
- ▶ Policy: $\pi_\theta : \mathcal{S} \rightarrow \mathcal{A}$ we want to learn the policy parameters θ
- ▶ Optimal action: $a^* \in \mathcal{A}$ provided by expert demonstrator
- ▶ Optimal policy: $\pi^* : \mathcal{S} \rightarrow \mathcal{A}$ provided by expert demonstrator
- ▶ State dynamics: $P(s_{i+1}|s_i, a_i)$ simulator, typically not known to policy
Often deterministic: $s_{i+1} = T(s_i, a_i)$ deterministic mapping
- ▶ Rollout: Given s_0 , sequentially execute $a_i = \pi_\theta(s_i)$ & sample $s_{i+1} \sim P(s_{i+1}|s_i, a_i)$
yields trajectory $\tau = (s_0, a_0, s_1, a_1, \dots)$
- ▶ Loss function: $\mathcal{L}(a^*, a)$ loss of action a given optimal action a^*

Formal Definition of Imitation Learning

General Imitation Learning:

$$\operatorname{argmin}_{\theta} \mathbb{E}_{s \sim P(s|\pi_{\theta})} [\mathcal{L}(\pi^*(s), \pi_{\theta}(s))]$$

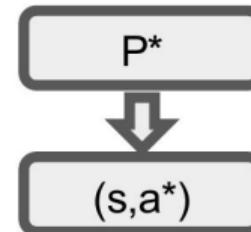
- ▶ State distribution $P(s|\pi_{\theta})$ depends on rollout determined by current policy π_{θ}



Behavior Cloning:

$$\operatorname{argmin}_{\theta} \underbrace{\mathbb{E}_{(s^*, a^*) \sim P^*} [\mathcal{L}(a^*, \pi_{\theta}(s^*))]}_{= \sum_{i=1}^N \mathcal{L}(a_i^*, \pi_{\theta}(s_i^*))}$$

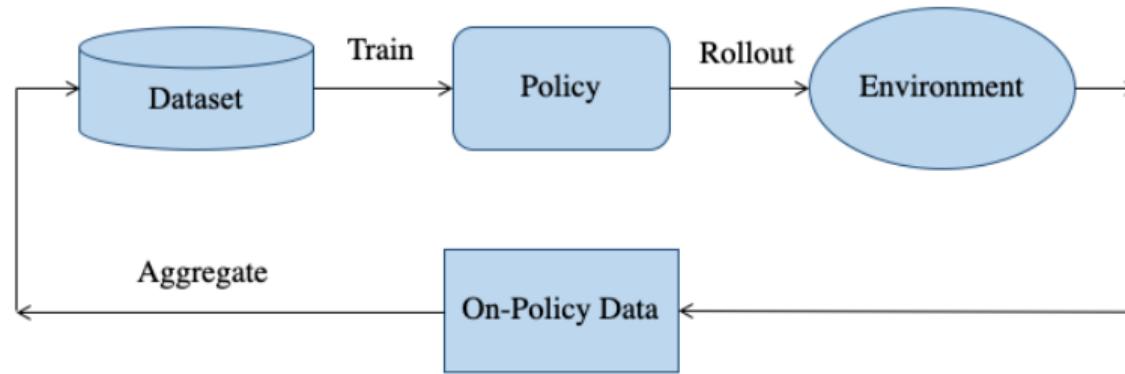
- ▶ State distribution P^* provided by expert
- ▶ Reduces to supervised learning problem



Challenges of Behavior Cloning

- ▶ Behavior cloning makes IID assumption
 - ▶ Next state is sampled from states observed during expert demonstration
 - ▶ Thus, next state is sampled independently from action predicted by current policy
- ▶ What if π_θ makes a mistake?
 - ▶ Enters new states that haven't been observed before
 - ▶ New states not sampled from same (expert) distribution anymore
 - ▶ Cannot recover, catastrophic failure in the worst case
- ▶ What can we do to overcome this train/test distribution mismatch?

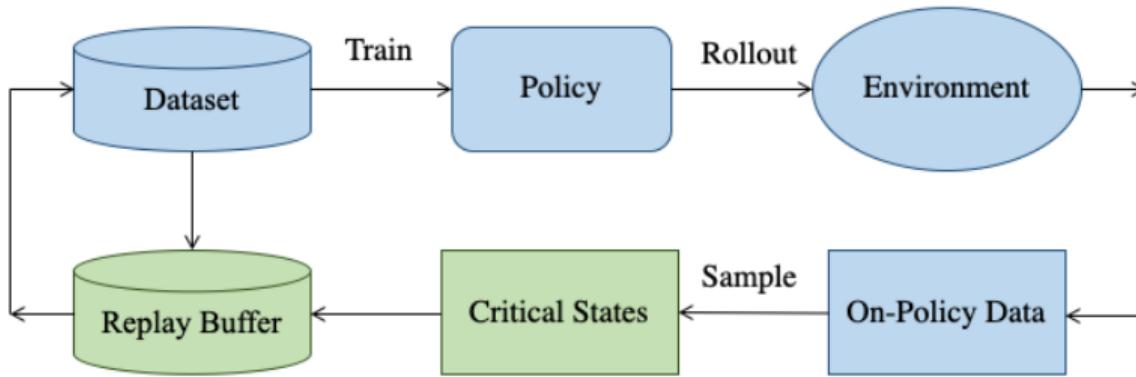
DAgger



Data Aggregation (DAgger):

- ▶ Iteratively build a set of inputs that the final policy is likely to encounter based on previous experience. Query expert for aggregate dataset
- ▶ But can easily overfit to main mode of demonstrations
- ▶ High training variance (random initialization, order of data)

Dagger with Critical States and Replay Buffer



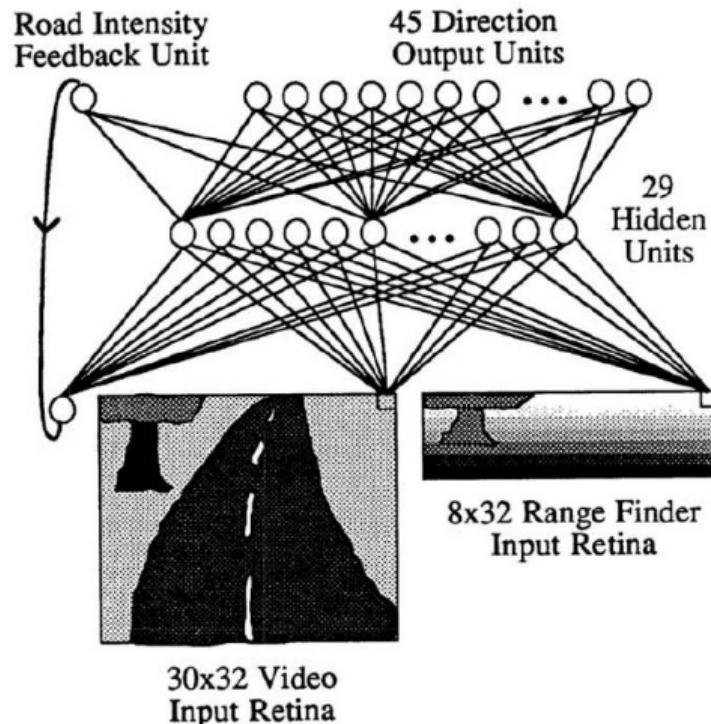
Key Ideas:

1. Sample **critical states** from the collected on-policy data based on the utility they provide to the learned policy in terms of driving behavior
2. Incorporate a **replay buffer** which progressively focuses on the high uncertainty regions of the policy's state distribution

ALVINN: An Autonomous Land Vehicle in a Neural Network

ALVINN: An Autonomous Land Vehicle in a Neural Network

- ▶ Fully connected 3 layer neural net
- ▶ 36k parameters
- ▶ Maps road images to turn radius
- ▶ Directions discretized (45 bins)
- ▶ Trained on simulated road images!
- ▶ Tested on unlined paths, lined city streets and interstate highways
- ▶ 90 consecutive miles at up to 70 mph

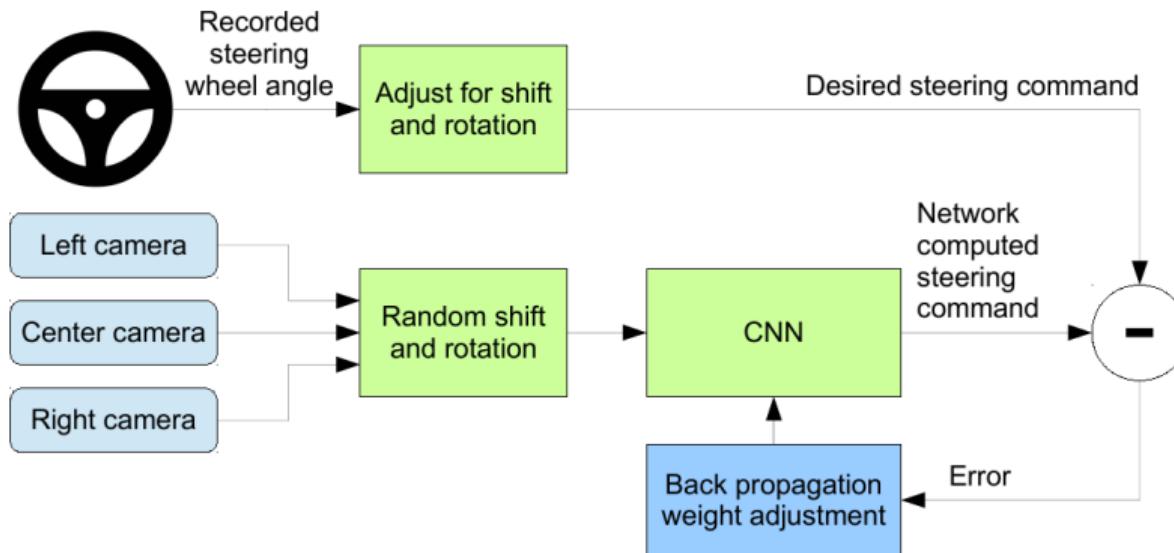


ALVINN: An Autonomous Land Vehicle in a Neural Network



PilotNet: End-to-End Learning for Self-Driving Cars

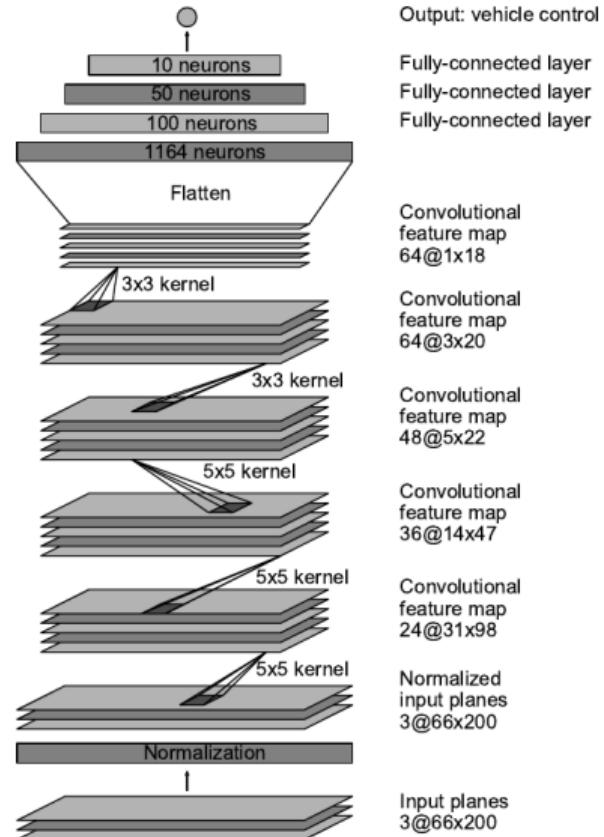
PilotNet: System Overview



- ▶ Data augmentation by 3 cameras and virtually shifted / rotated images assuming the world is flat (homography), adjusting the steering angle appropriately

PilotNet: Architecture

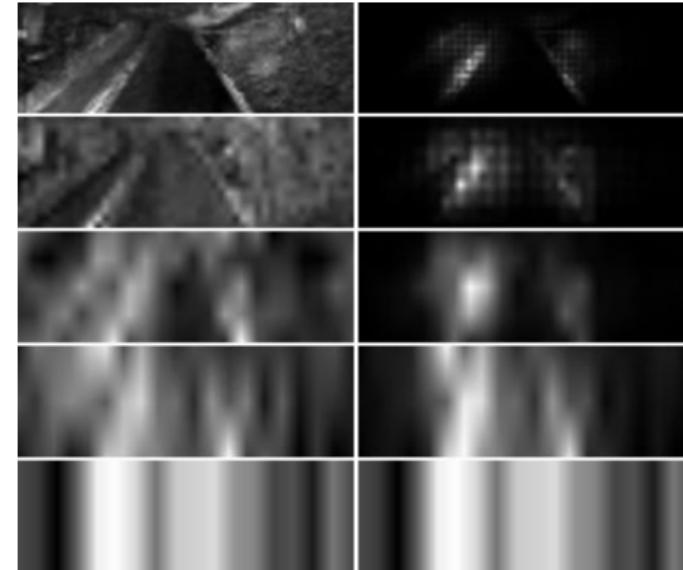
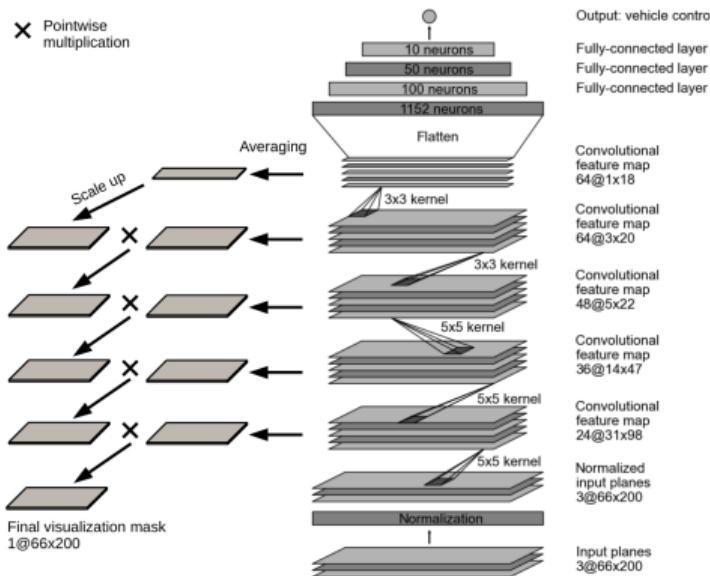
- ▶ Convolutional network (250k param)
- ▶ Input: YUV image representation
- ▶ 1 Normalization layer
 - ▶ Not learned
- ▶ 5 Convolutional Layers
 - ▶ 3 strided 5x5
 - ▶ 2 non-strided 3x3
- ▶ 3 Fully connected Layers
- ▶ Output: turning radius
- ▶ Trained on 72h of driving



PilotNet: Video



VisualBackProp



- Central idea: find **salient image regions** that lead to high activations
- Forward pass, then iteratively scale-up activations

VisualBackProp



VisualBackProp

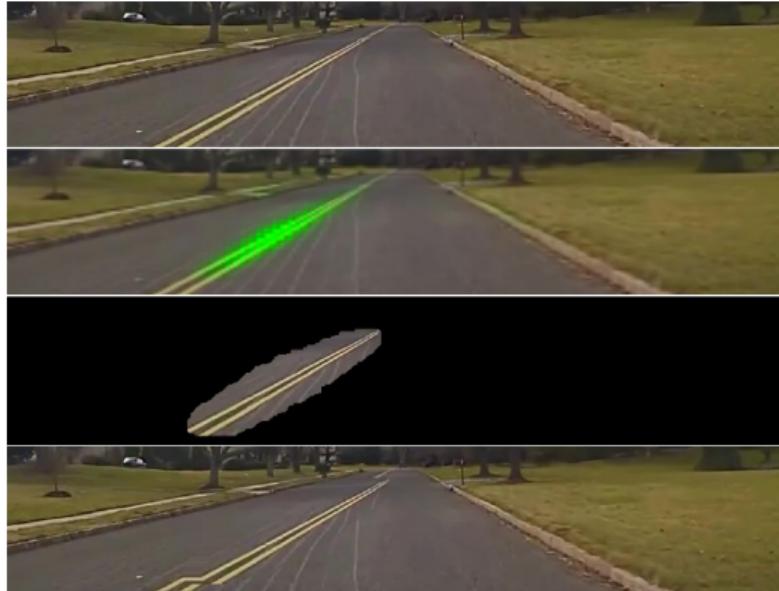
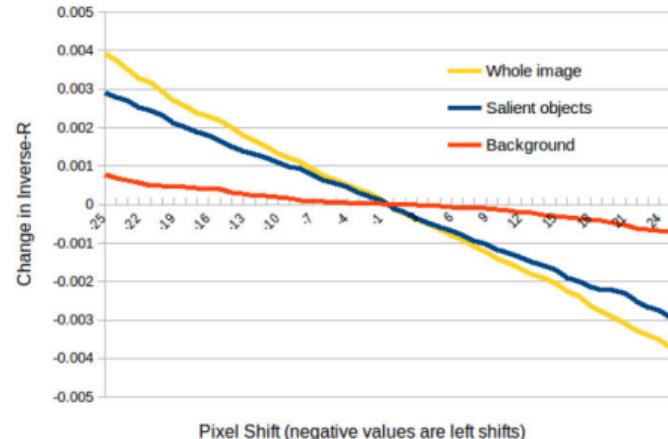


Figure 7: Images used in experiments to show the effect of image-shifts on steer angle.

Applying Displacement to Salient Objects, Background, and Whole Image
And Measuring the Median Change in Predicted Inverse-R
Across a Sample of 200 Images



- ▶ Test if shift in salient objects affects predicted turn radius more strongly

2.4

Conditional Imitation Learning

Conditional Imitation Learning

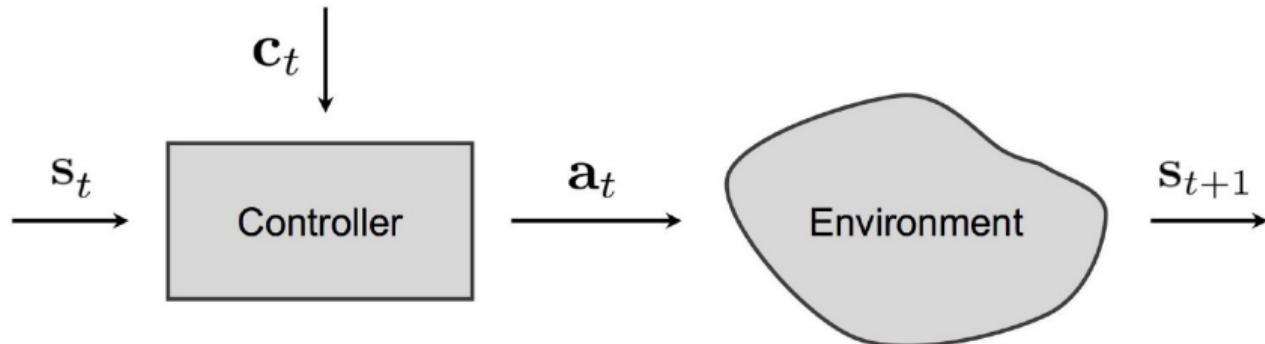


(a) Aerial view of test environment

(b) Vision-based driving, view from onboard camera

(c) Side view of vehicle

Conditional Imitation Learning



Idea:

- ▶ Condition controller on **navigation command** $c \in \{\text{left}, \text{right}, \text{straight}\}$
- ▶ High-level navigation command can be provided by consumer GPS, i.e., telling the vehicle to **turn left/right** or go **straight** at the next intersection
- ▶ This removes the task ambiguity induced by the environment
- ▶ State s_t : current image Action a_t : steering angle & acceleration

Comparison to Behavior Cloning

Behavior Cloning:

- ▶ Training Set:

$$\mathcal{D} = \{(a_i^*, s_i^*)_{i=1}^N\}$$

- ▶ Objective:

$$\operatorname{argmin}_{\theta} \sum_{i=1}^N \mathcal{L}(a_i^*, \pi_{\theta}(s_i^*))$$

- ▶ Assumption:

$$\exists f(\cdot) : a_i = f(s_i)$$

Often violated in practice!

Conditional Imitation Learning:

- ▶ Training Set:

$$\mathcal{D} = \{(a_i^*, s_i^*, c_i^*)_{i=1}^N\}$$

- ▶ Objective:

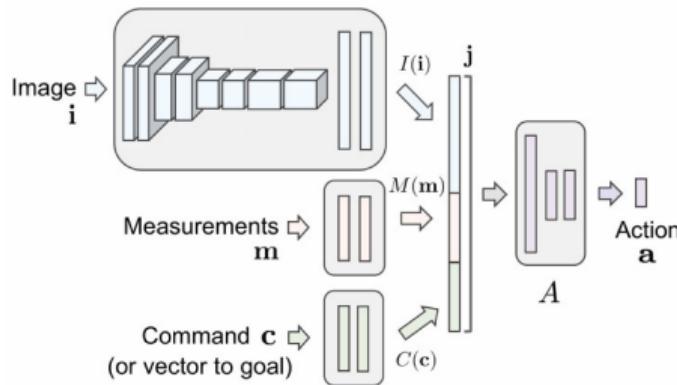
$$\operatorname{argmin}_{\theta} \sum_{i=1}^N \mathcal{L}(a_i^*, \pi_{\theta}(s_i^*, c_i^*))$$

- ▶ Assumption:

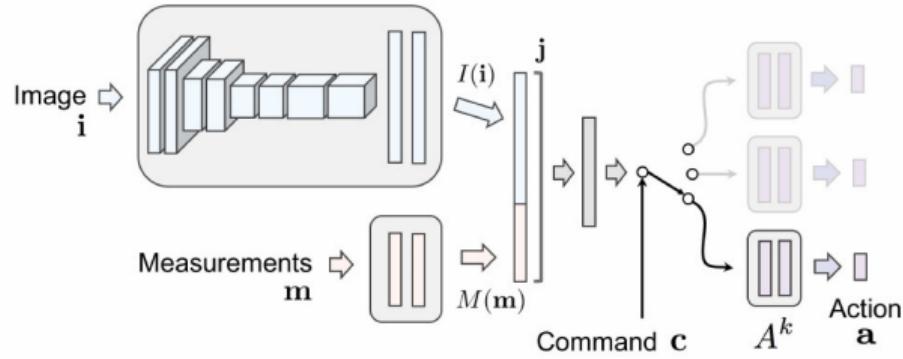
$$\exists f(\cdot, \cdot) : a_i = f(s_i, c_i)$$

Better assumption!

Conditional Imitation Learning: Network Architecture



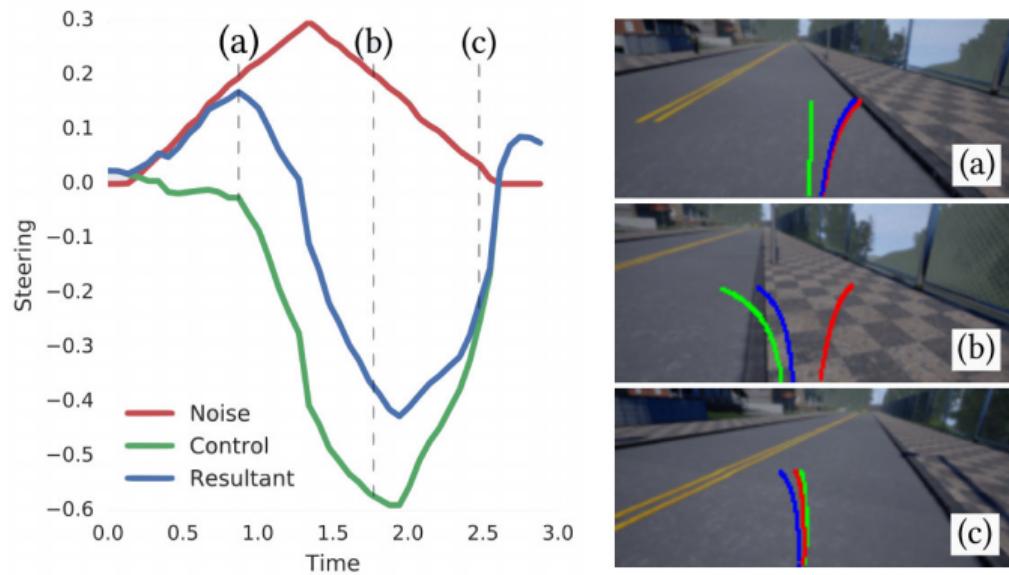
(a)



(b)

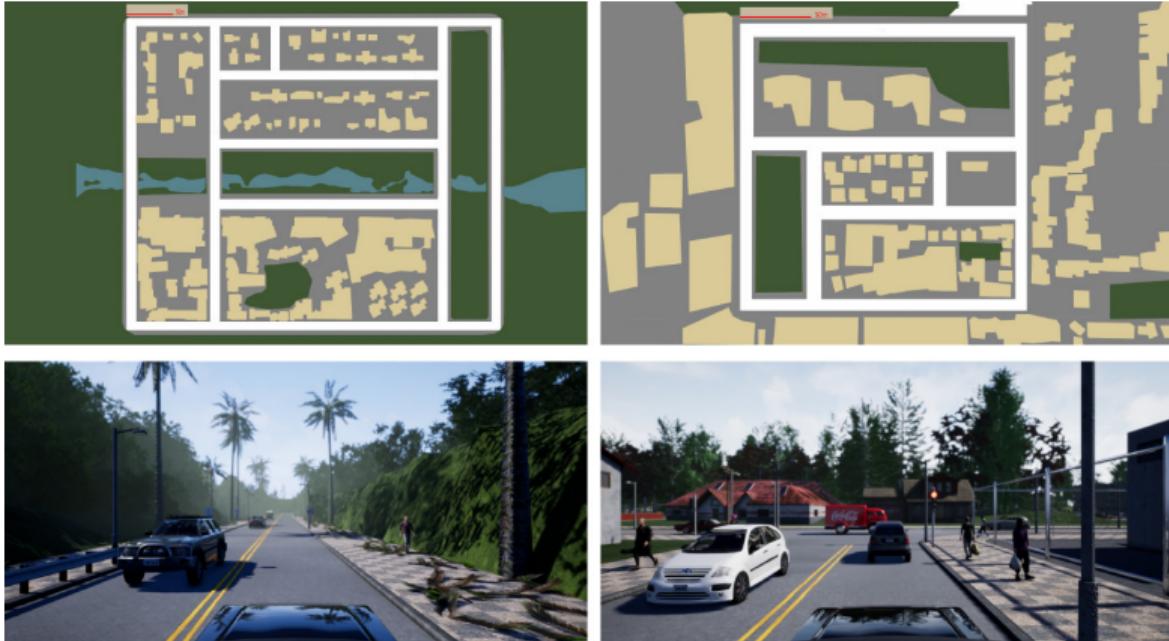
- ▶ This paper proposes two network architectures:
 - ▶ (a) Extract features $C(\mathbf{c})$ and concatenate with image features $I(\mathbf{i})$
 - ▶ (b) Command \mathbf{c} acts as switch between specialized submodules
- ▶ Measurements \mathbf{m} capture additional information (here: speed of vehicle)

Conditional Imitation Learning: Noise Injection



- ▶ Temporally correlated noise injected into trajectories \Rightarrow drift (only 12 minutes)
- ▶ Record driver's (=expert's) corrective response \Rightarrow recover from drift

CARLA Simulator

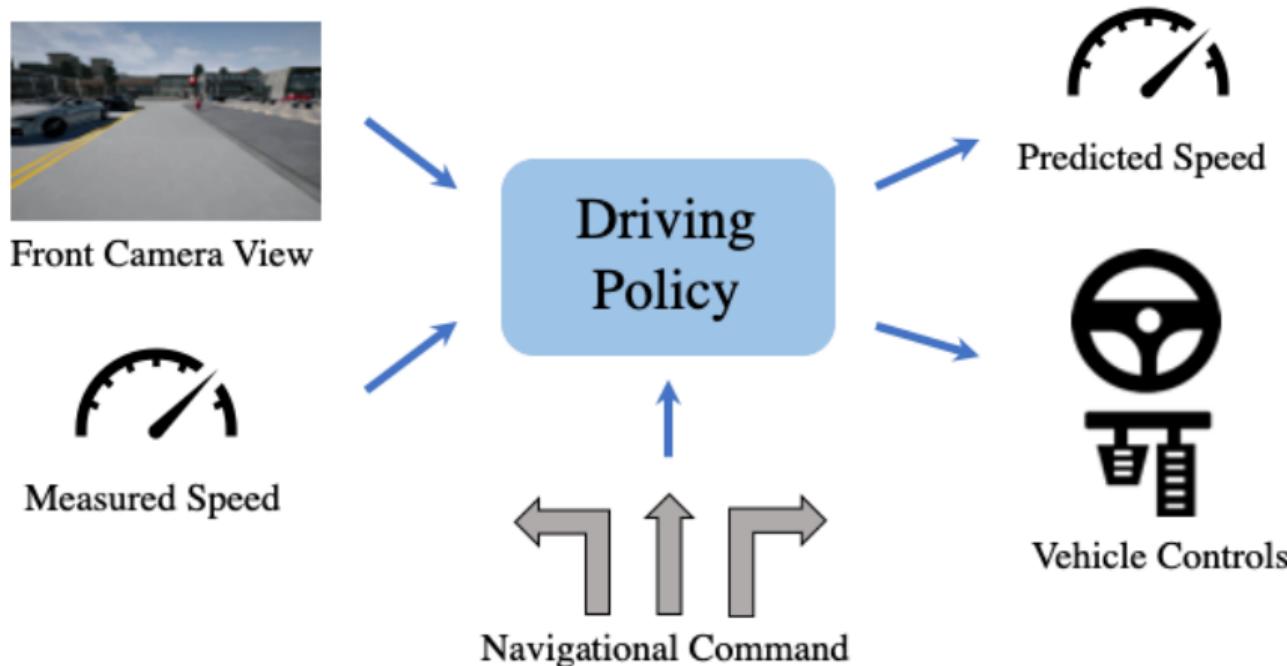


Town 1 (training)

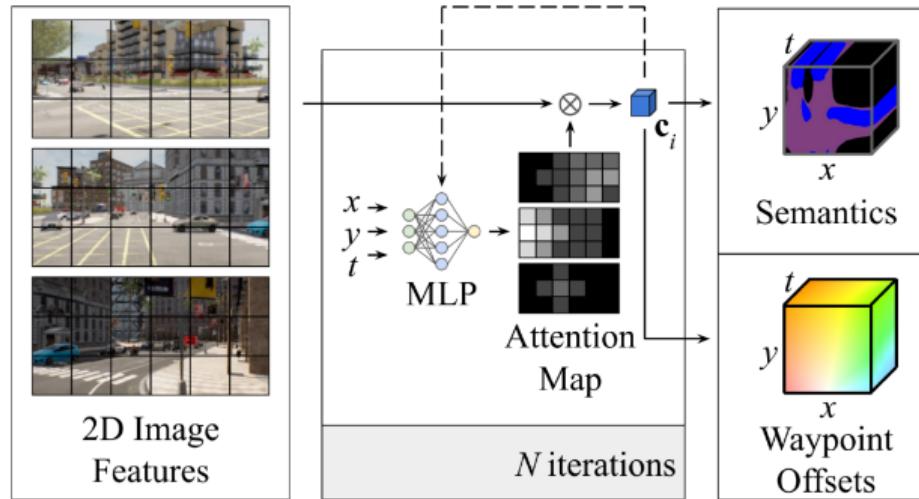
Town 2 (testing)

<http://www.carla.org>

Conditional Imitation Learning



Neural Attention Fields



- ▶ An MLP iteratively **compresses** the high-dimensional input into a compact representation \mathbf{c}_i ($\mathbf{c} \neq$ nav. command) based on a **BEV query location** as input
- ▶ The model predicts **waypoints** and auxiliary **semantics** which aids learning

Summary

Advantages of Imitation Learning:

- ▶ Easy to implement
- ▶ Cheap annotations (just driving while recording images and actions)
- ▶ Entire model trained end-to-end
- ▶ Conditioning removes ambiguity at intersections

Challenges for Imitation Learning?

- ▶ Behavior cloning uses IID assumption which is violated in practice
- ▶ Direct mapping from images to control ⇒ No long term planning
- ▶ No memory (can't remember speed signs, etc.)
- ▶ Mapping is difficult to interpret ("black box"), despite visualization techniques