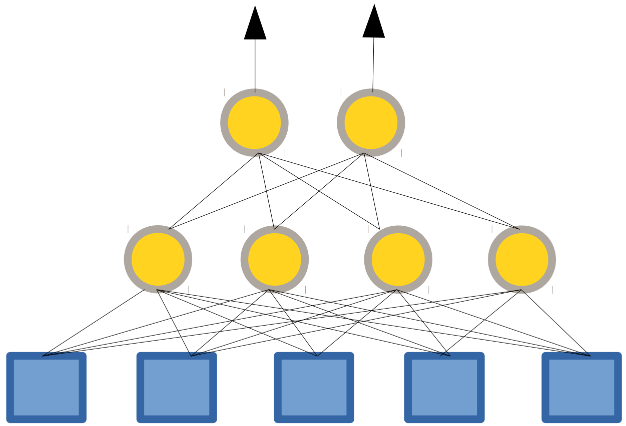
Recurrent Neural Networks

Interactive Robotics Lab

Feed-Forward Neural Networks

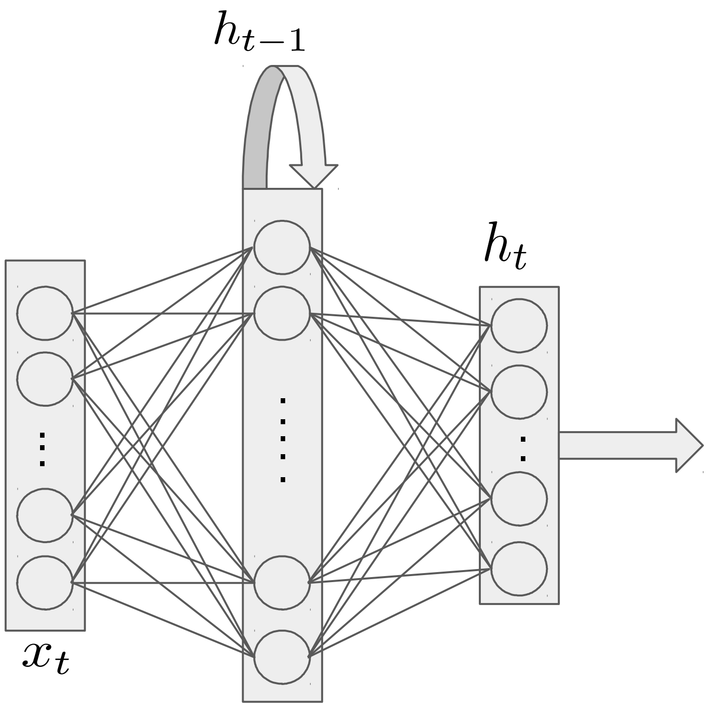
- Hierarchy of neurons
- Input layer, hidden layers, output layer
- Does not have a **memory**
- Independent of last decision



Recurrent Neural Networks

A Recurrent Neural Network (RNN) learns temporal correlations between arbitrarily distant events

$x_t \in \mathbb{R}^m$

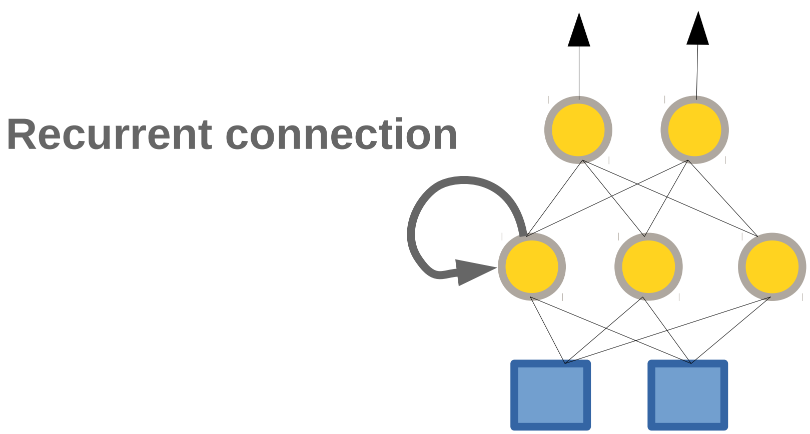


$h_t \in \mathbb{R}^h$

RNNs Regress, classify, predict and generate sequential data in almost all machine learning domains

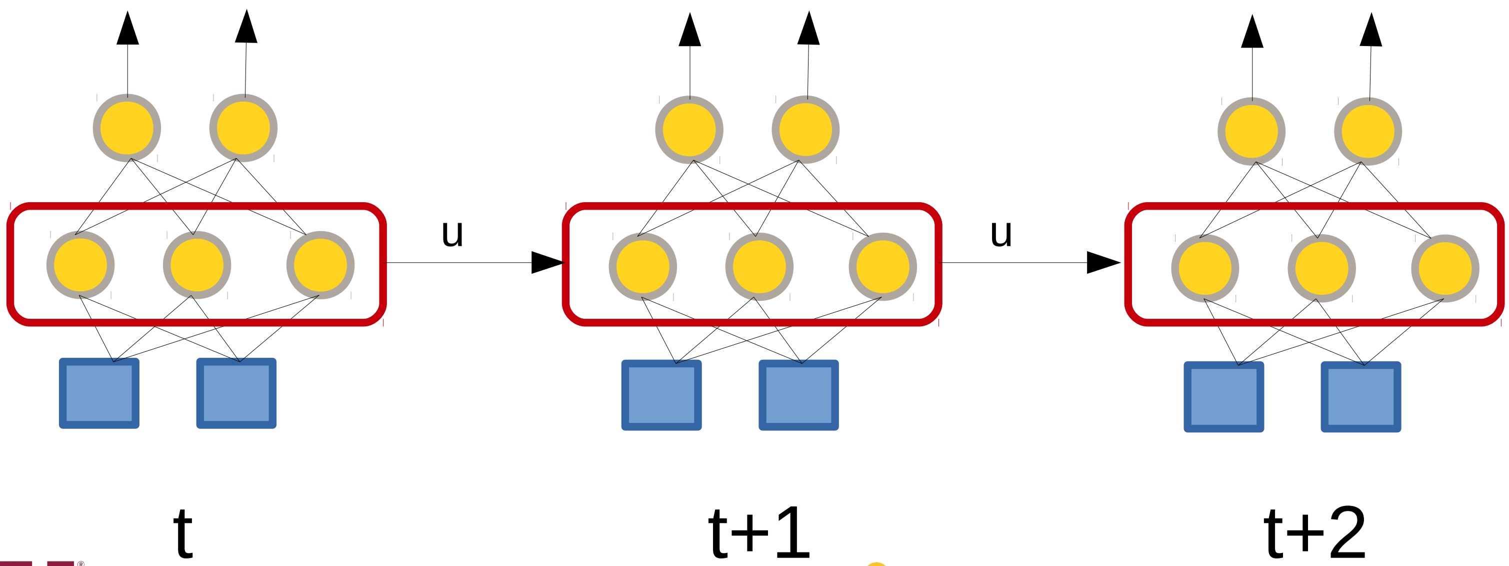
Recurrent Neural Networks

- Memory through **recurrent connections**
- Feedback information from last steps
- Loops in the network



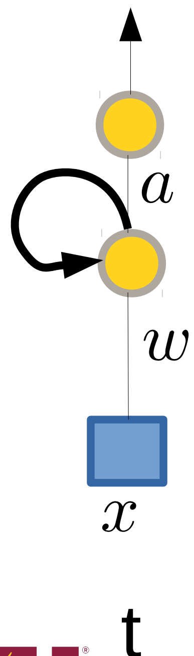
A Different Perspective

- RNN can be seen as multiple ANN communicating
- Message sent between them
- Ideal for sequence learning (text, music, video)
- Time-series prediction



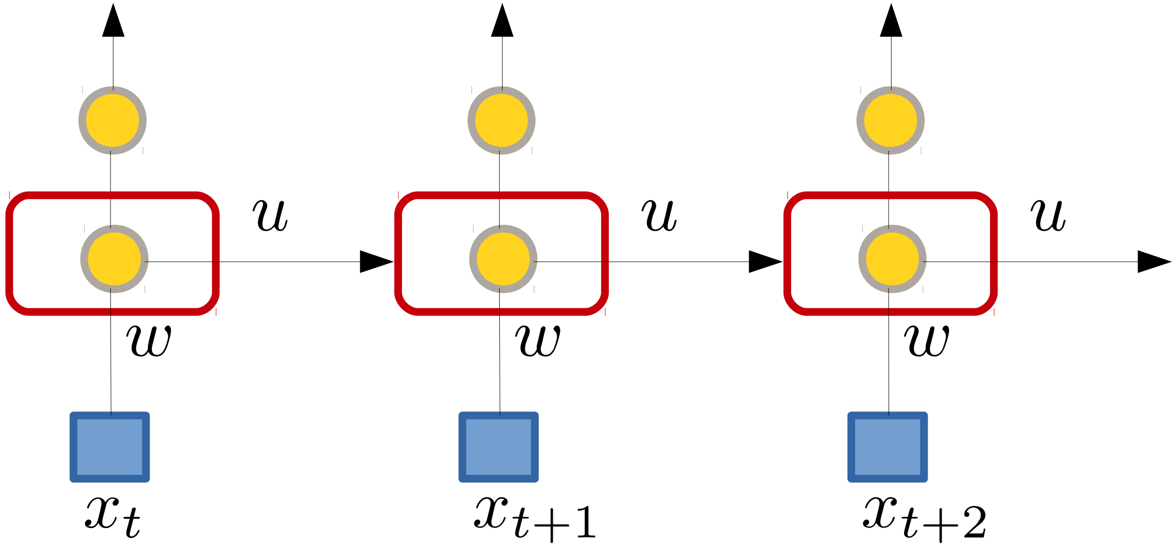
Backpropagation through Time

- The same as BP
- Use **unfolded** network
- Define maximum sequence length



Backpropagation through Time

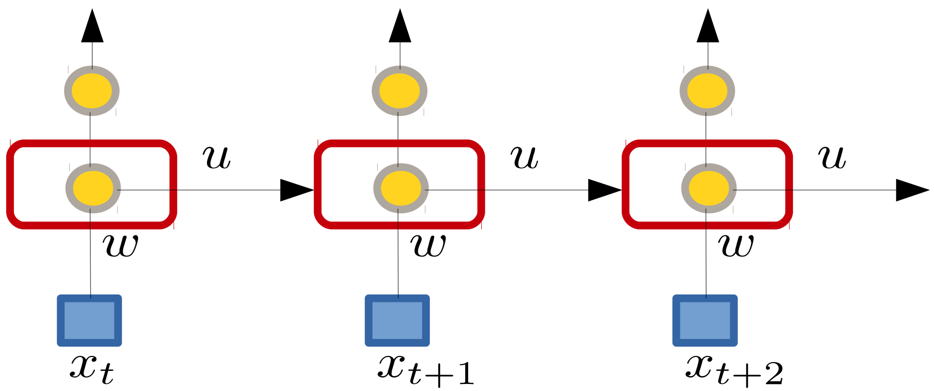
- The same as BP
- Use **unfolded** network
- Define maximum sequence length



Interactive Robotics Lab

Backpropagation through Time (BBTT)

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



Simple RNNs Revisited

$$h_t = \phi(\mathbf{W_x} x_t + \mathbf{U_h} h_{t-1} + b)$$

$\mathbf{W_x}$ – $Input\,Weight\,Matrix$

h_{t-1} – $Previous\,Hidden\,Output$

$\mathbf{U_h}$ – $Recurrent\,Weight\,Matrix$

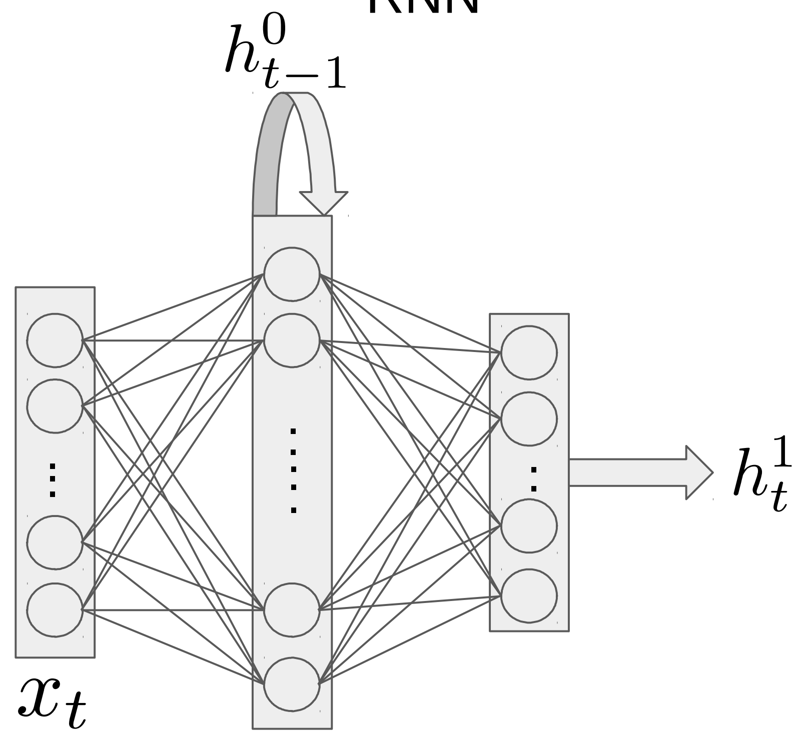
h_t – $Hidden\,Output$

b – $Bias\,Vector$

ϕ – $Activation\,Function$

x_t – $Input\,Vector$

Hidden layer to hidden layer connections allow temporal information to flow through the RNN



Interactive Robotics Lab

BPTT Chain Rule

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial \theta}$$

Partial derivative of loss with respect to output

$$\frac{\partial \mathcal{L}_t}{\partial \theta} = \sum_{k=1}^{t} \left(\frac{\partial \mathcal{L}_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial h_k} \cdot \frac{\partial h_k^+}{\partial \theta} \right)$$

Immediate partial derivative

$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k}^{t} \frac{\partial h_i}{\partial h_{i-1}}$$

The following term gives the relation of error through time where k < t

Vanishing and Exploding Gradients

Significant problem for learning long term dependencies

$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k}^{t} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=k}^{t} \mathbf{W}_{rec} diag(\phi'(x_{i-1}))$$

Recurrent Weight Matrix Contribution

If the Eigen values of the recurrent weight matrix deviate below one, the contribution of "distant" events quickly converges to zero

Vanishing and Exploding Gradients

This problem occurs when the norm of the gradients during training vanish or explode

$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k}^{t} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=k}^{t} \mathbf{W}_{rec} diag(\phi'(x_{i-1}))$$

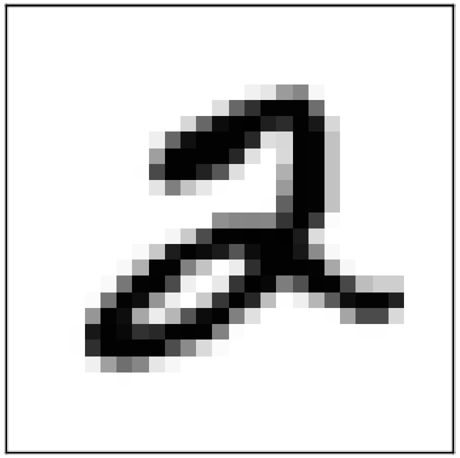
Activation Function Contribution

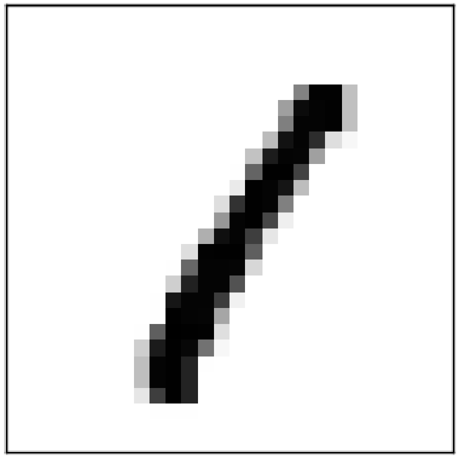
If the gradient of the activation function deviates considerably from one, the product above explodes or vanishes as k << t

Sequence MNIST Benchmark

- Goal:
  - Classify handwritten digits by reading one pixel at a time
  - Proposed as benchmark RNN dataset by Le, Jaitly and Hinton

- Input tensor shape
  - (batch, pixel)
- Output tensor shape
  - (batch, one_hot_sz)
- Sequence length
  - 784 – pixels in 1 image
- Cross entropy loss





[Source] http://yann.lecun.com/exdb/mnist/

Interactive Robotics Lab

PyTorch Dataset Class SeqMNIST

```
import torch
from torch.utils.data import Dataset

class SequentialMNIST(Dataset):

    def __init__(self, mode=MODE_TRAIN, pixel_wise=True, permute=False):
        #Initialize dataset here

    #Set the mode depending on train test or val
    def train(self):
        #self.mode = SequentialMNIST.MODE_TRAIN
    def val(self):
        #self.mode = SequentialMNIST.MODE_VAL
    def test(self):
        #self.mode = SequentialMNIST.MODE_TEST

    def __len__(self):
       #Return size of dataset for train, test, or val
    def __getitem__(self, i):
       #Depending on the mode – train, val test
        return batch_of_elements
```

Training Script Continued

```
def run_sequence(seq, target):
    predicted_list = []
    y_list = []

    #Initialize memory states
    model.reset(batch_size=seq.size(0), cuda=args.cuda)

    #Execute inference on the model sequentially
    for i, input_t in enumerate(seq.chunk(seq.size(1), dim=1)):
        input_t = input_t.squeeze(1)

        p = model(input_t)

        predicted_list.append(p)
        y_list.append(target)

    #Return predicted values as well as their corresponding targets
    return predicted_list, y_list
```

Training Script SeqMNIST

```
def train(epoch, model, dset):
    model.train()
    dset.train()
    #total_loss = 0.0, steps=0, n_correct=0, n_possible=0

    for batch_idx, (data, target) in enumerate(data_loader):
        if args.cuda:
            data, target = data.cuda().double(), target.cuda().double()
        data, target = Variable(data), Variable(target)

        predicted_list, y_list = run_sequence(data, target) #Defined on next slide

        pred = predicted_list[-1] #Take the final output from the RNN
        y_ = y_list[-1].long() #Take the final batch of targets

        prediction = pred.data.max(1, keepdim=True)[1].long()
        n_correct += prediction.eq(y_.data.view_as(prediction)).sum().cpu().numpy()
        n_possible += int(prediction.shape[0])

        loss = F.nll_loss(pred, y_) #Calculate batch loss

        loss.backward() #Calculate gradients
        optimizer.step() #Update NN weights
```

Training Script Continued

```
def run_sequence(seq, target):
    predicted_list = []
    y_list = []

    #Initialize memory states
    model.reset(batch_size=seq.size(0), cuda=args.cuda)

    #Execute inference on the model sequentially
    for i, input_t in enumerate(seq.chunk(seq.size(1), dim=1)):
        input_t = input_t.squeeze(1)

        p = model(input_t)

        predicted_list.append(p)
        y_list.append(target)

    #Return predicted values as well as their corresponding targets
    return predicted_list, y_list
```

Modern Solutions – Architecture

Modern RNN architectures have been proposed to address the vanishing and exploding gradient problem

Model	Description	Reference
LSTM	Most ubiquitous RNN architecture today. Adds gated computations and cell memory state for long term memory.	http://www.bioinf.jku.at/publications/older/2604.pdf
LSTM Forget Gates	Adds new gate to LSTM architecture that focuses on "forgetting" long-term dependencies that are no longer relevant.	https://pdfs.semanticscholar.org/1154/0131eae85b2e11d53df7f1360eeb6476e7f4.pdf
Peephole LSTM	Uses previous cell state for gate computations instead of hidden state; accesses constant error carousel.	ftp://ftp.idsia.ch/pub/juergen/TimeCount-IJCNN2000.pdf
GRU	Combines input and forget gates into single update gate and combines the cell and hidden memory states.	https://arxiv.org/pdf/1406.1078v3.pdf
IndRNN	Forces the recurrent weight matrix to be a vector that is multiplied element-wise by the previous hidden state.	https://arxiv.org/pdf/1803.04831.pdf
UGRNN RNN+	Modern architectures made to enhance trainability of deeply-stacked (RNN+) and shallow (UGRNN) models.	https://arxiv.org/pdf/1611.09913.pdf

Modern Solutions: Initialization

Eigenvalues of the recurrent weight matrix need to be equal to one in order to avoid the vanishing and exploding gradient problem

$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k}^{t} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=k}^{t} \mathbf{W}_{rec} diag(\phi'(x_{i-1}))$$

Both of these matrices have Eigenvalues equal to one

In practice, soft constraints imposed on these matrices after initialization improves trainability of RNNs

Identity Initialization

$$\mathbf{W}_{rec} = \begin{bmatrix} 1 & 0 & \dots \\ \vdots & \ddots & \\ 0 & & 1 \end{bmatrix}$$

Orthogonal Initialization

$$\mathbf{W}_{rec} = \begin{bmatrix} a_{11} & a_{12} & \dots \\ \vdots & \ddots & \\ a_{K1} & & a_{KK} \end{bmatrix}$$

Modern Solutions: Activations

The derivative of the activation function is part of the product that causes the temporal gradient to vanish or explode

$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k}^{t} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=k}^{t} \mathbf{W}_{rec} diag(\phi'(x_{i-1}))$$

Sigmoid Activation

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

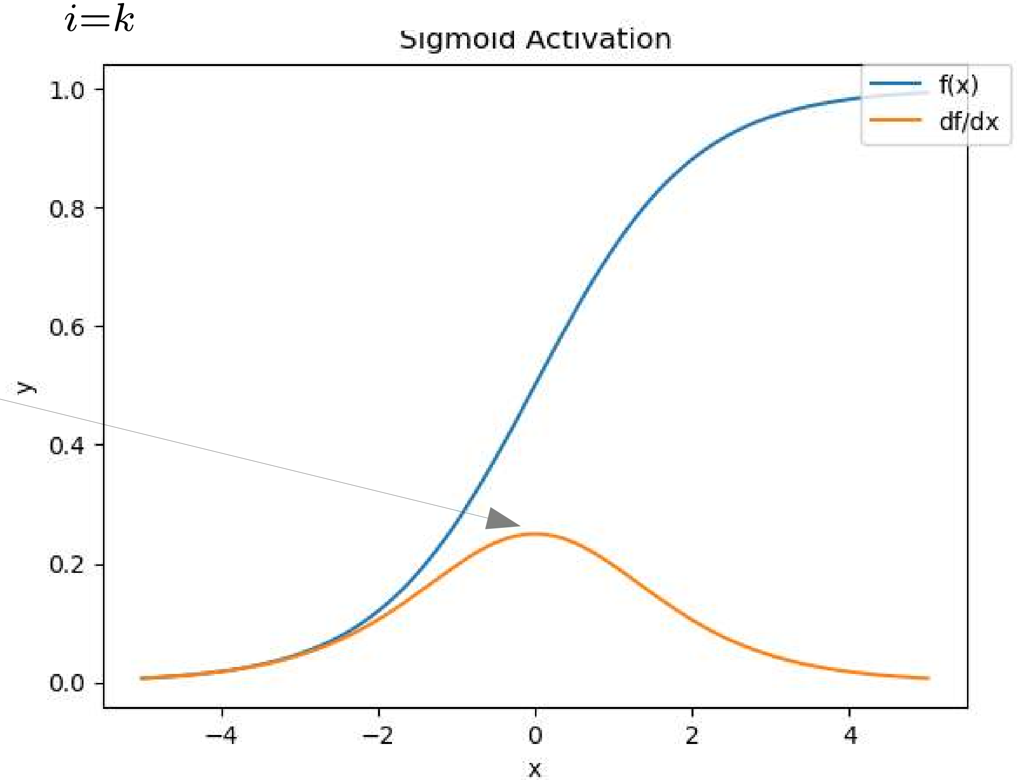
$$\sigma' = (1 - \sigma)\sigma$$

Max value of df/dx is .25

Temporal gradient vanishes quickly with this activation function

$$.25^2 = .0625$$

$$.25^5 = 0.00097$$



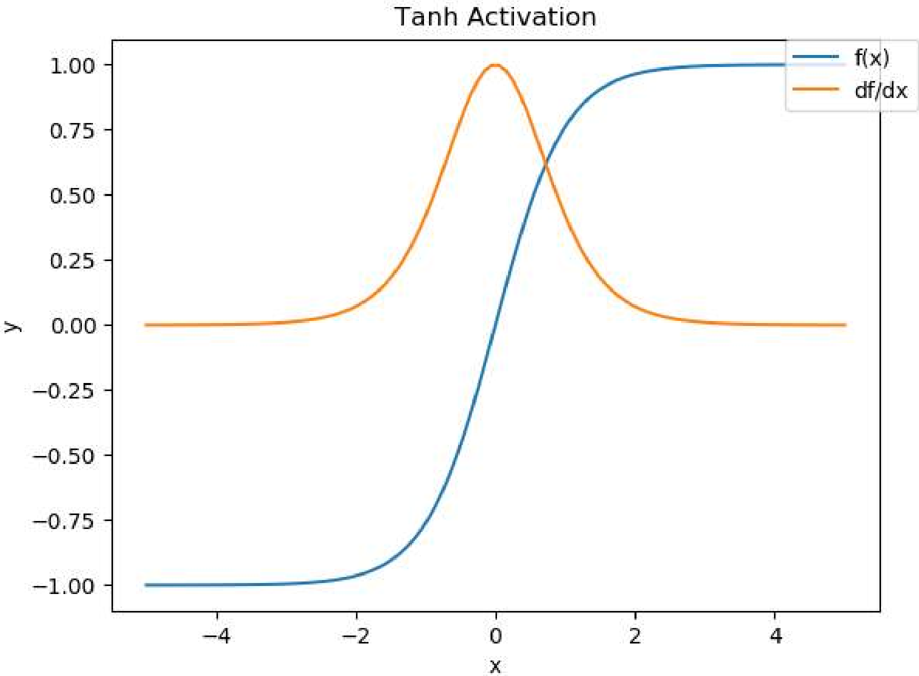
Modern Solutions: Activations

Tanh Activation

$$tanh = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad tanh' = 1 - tanh^2(x)$$

Heavily used in modern gated recurrent architectures

The gradient vanishes more quickly the further x deviates from 0

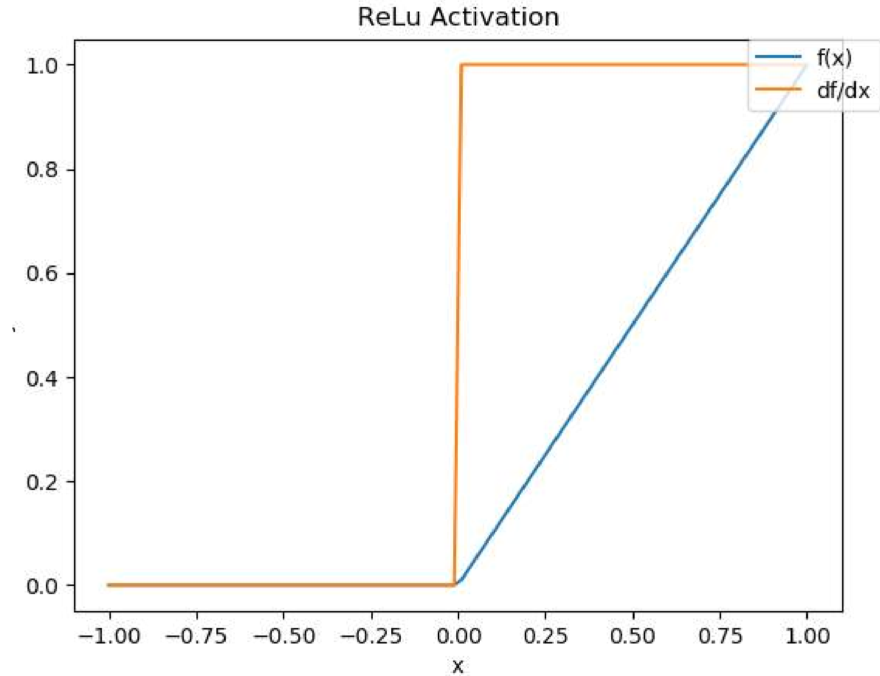


ReLu Activation

$$ReLu = max(0, x) \quad ReLu' = \begin{cases} x > 0, & 1 \\ x <= 0, & 0 \end{cases}$$

ReLu activation has desirable gradient behavior for values of x > 0

For x < 0 the temporal gradient does not exist



Custom RNN Cell Template PyTorch

```
class CustomRNNCell(nn.Module):
    def __init__(self, input_size,
                # Define custom variables of interest - dropout ect
                hidden_size):
        super(CustomRNNCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size

        #Initialize global variables
        #Initialize parameters W, U, b ect.

        self.hidden_state = None

    def reset(self, batch_size=1, cuda=True):
        #Initialize Memory States h_t, c_t ect.

    def forward(self, X_t):
        h_t_previous = self.hidden_state #extract memory states (h_t-1, c_t-1)
        #Do computation here
        #Set memory states
        self.hidden_state = y
        return y
```

IRNN in PyTorch

```
self.W_x = nn.Parameter(torch.zeros(input_size, hidden_size))
self.W_x = nn.init.xavier_normal_(self.W_x)

#Identity recurrent weight matrix initialization
self.U_h = torch.nn.Parameter(torch.eye(hidden_size))

self.b = nn.Parameter(torch.zeros(hidden_size))


def forward(self, X_t):
        h_t_previous = self.hidden_state

        out = F.relu( torch.mm(X_t, self.W_x) +
           torch.mm(h_t_previous, self.U_h) +
self.b)

        self.hidden_state = out
        return out
```

Identity matrix initialization

F.relu – Rectified linear unit activation function

Interactive Robotics Lab

LSTM & Peephole Connections

```
def forward(self, X_t):
        h_t_previous, c_t_previous = self.states

        f_t = F.sigmoid(
            torch.mm(X_t, self.W_f) + torch.mm(h_t_previous, self.U_f) + self.b_f)

        i_t = F.sigmoid(
            torch.mm(X_t, self.W_i) + torch.mm(h_t_previous, self.U_i) + self.b_i)

        o_t = F.sigmoid(
            torch.mm(X_t, self.W_o) + torch.mm(h_t_previous, self.U_o) + self.b_o)

        c_hat_t = F.tanh(
            torch.mm(X_t, self.W_c) + torch.mm(h_t_previous, self.U_c) + self.b_c)

        c_t = (f_t * c_t_previous) + (i_t * c_hat_t)

        h_t = o_t * F.tanh(c_t)

        self.states = (h_t, c_t)
        return h_t
```

Replace h_t_previous with c_t_previous for Peephole LSTM variant

GRU

```
def forward(self, X_t):

    h_t_previous = self.recurrent_state

    z_t = F.sigmoid(
        torch.mm(X_t, self.W_z) + torch.mm(h_t_previous, self.U_z) + self.b_z)

    r_t = F.sigmoid(
        torch.mm(X_t, self.W_r) + torch.mm(h_t_previous, self.U_r) + self.b_r)

    h_t = z_t * h_t_previous + ((z_t - 1) * -1) * F.tanh(
        torch.mm(X_t, self.W_h) + torch.mm((r_t * h_t_previous), self.U_h) + self.b_h)

    self.recurrent_state = h_t

    return h_t
```

UGRNN

```
def forward(self, X_t):
        h_t_previous=self.states

        g_t = F.sigmoid(
            torch.mm(X_t, self.W_g) + torch.mm(h_t_previous, self.U_g) + self.b_g)

        c_t = F.tanh(
            torch.mm(X_t, self.W_c) + torch.mm(h_t_previous, self.U_c) + self.b_c)

        h_t = g_t * h_t_previous + ((g_t - 1) * -1) * c_t

        self.states = h_t
        return h_t
```

Intersection RNN

```
def forward(self, X_t):

        h_t_previous = self.states

        y_in = F.tanh(
            torch.mm(X_t, self.W_yin) + torch.mm(h_t_previous, self.U_yin) + self.b_yin)

        h_in = F.tanh(
            torch.mm(X_t, self.W_hin) + torch.mm(h_t_previous, self.U_hin) + self.b_hin)

        g_y = F.sigmoid(
            torch.mm(X_t, self.W_gy) + torch.mm(h_t_previous, self.U_gy) + self.b_gy)

        g_h = F.sigmoid(
            torch.mm(X_t, self.W_gh) + torch.mm(h_t_previous, self.U_gh) + self.b_gh)

        y_t = g_y * X_t + ((g_y - 1) * -1) * y_in

        h_t = g_h * h_t_previous + ((g_h - 1) *-1) * h_in

        self.states = h_t
        return y_t
```

Exercise: RNN ZOO

Test novel RNN architectures on famous benchmark tasks Sequential MNIST and Permuted Sequential MNIST. Partial code is provided.

python train.py --hx=50 --layers=2 –model-type=lstm

run “python train.py --help” for description of hyperparameters

Students Task:

- Define weight matrix and recurrent weight matrix for vanilla RNN. See models/rnn.py
- Define LSTM forward method (LSTM secret sauce).

  See models/lstm.py
- Define how to reset recurrent states for GRU.

  See models/gru.py

Sequence MNIST

- Goal

  Classify handwritten digits by reading one pixel at a time

  Proposed as benchmark RNN dataset by Le, Jaitly and Hinton

- Input tensor shape:

  (batch, pixel)

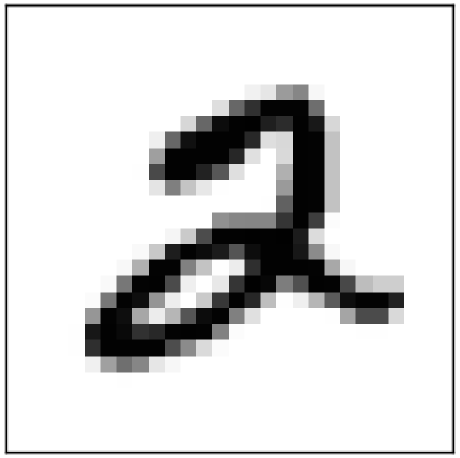
- Output tensor shape:

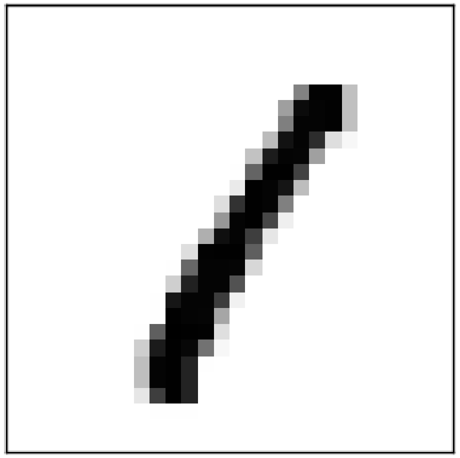
  (batch, one_hot_sz)

- Sequence length:

  784 – pixels in 1 image

- Cross entropy loss





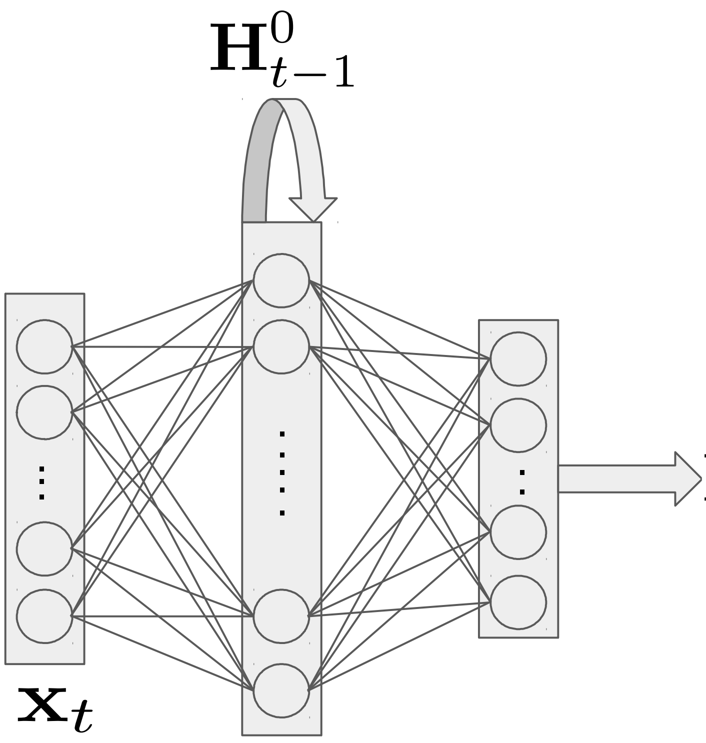
Convolutional RNNs

Learns spatio-temporal correlations

X is a set of activation maps

X is commonly an RGB or RGBD image

$$\mathbf{X}_t \in \mathbb{R}^{C \times L \times W}$$



H is a set of activation maps

H depends on multiple factors, including the number of filters in the hidden layer, stride, padding, ect.

$$\mathbf{H}_t^{1} \in \mathbb{R}^{C_h \times L_h \times W_h}$$

Convolutional RNNs

Feature extraction no longer occurs by fully connecting the input with its respective weight matrix; features are now extracted through convolutional layers

$$\mathbf{H}_t^n = \phi(conv(\mathbf{W}_x, \mathbf{X}_t^{n-1}) + conv(\mathbf{U}_h, \mathbf{H}_{t-1}^n) + \mathbf{B})$$

Be careful! Convolving **H** and **U** needs to produce the same shape tensor as convolving **W** and **X**

- For recurrent convolutional layer
  - Set stride equal to one
  - Make the number of filters in **U** equal to the number of filters in **W**
  - Set proper padding – assuming stride of one

Dodge Ball

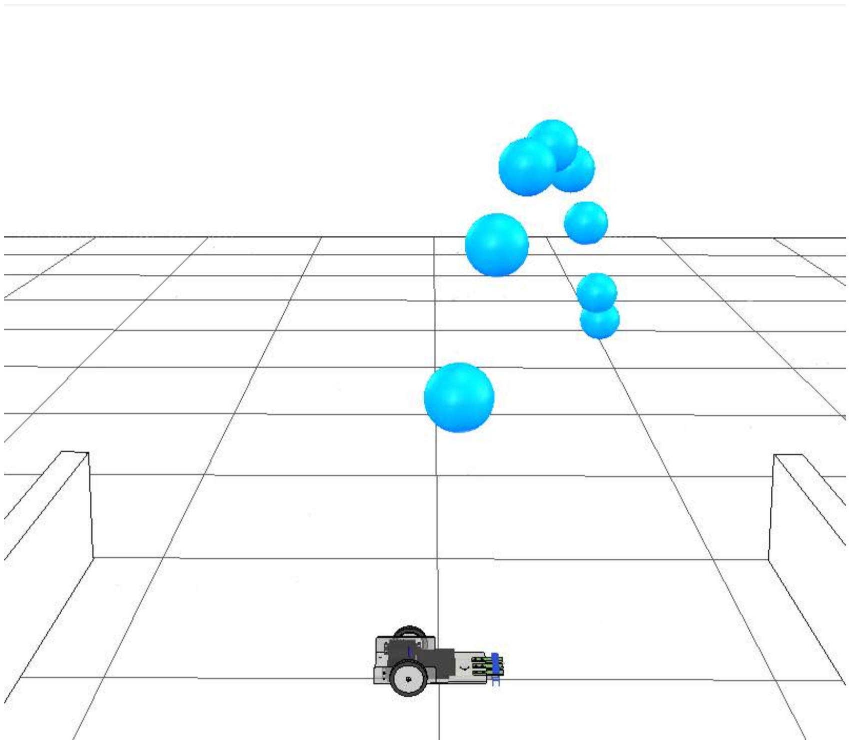
Can a robot dodge balls with a RGB video sensor?

Goal:

- Successfully predict future collisions given a randomly initialized projectile

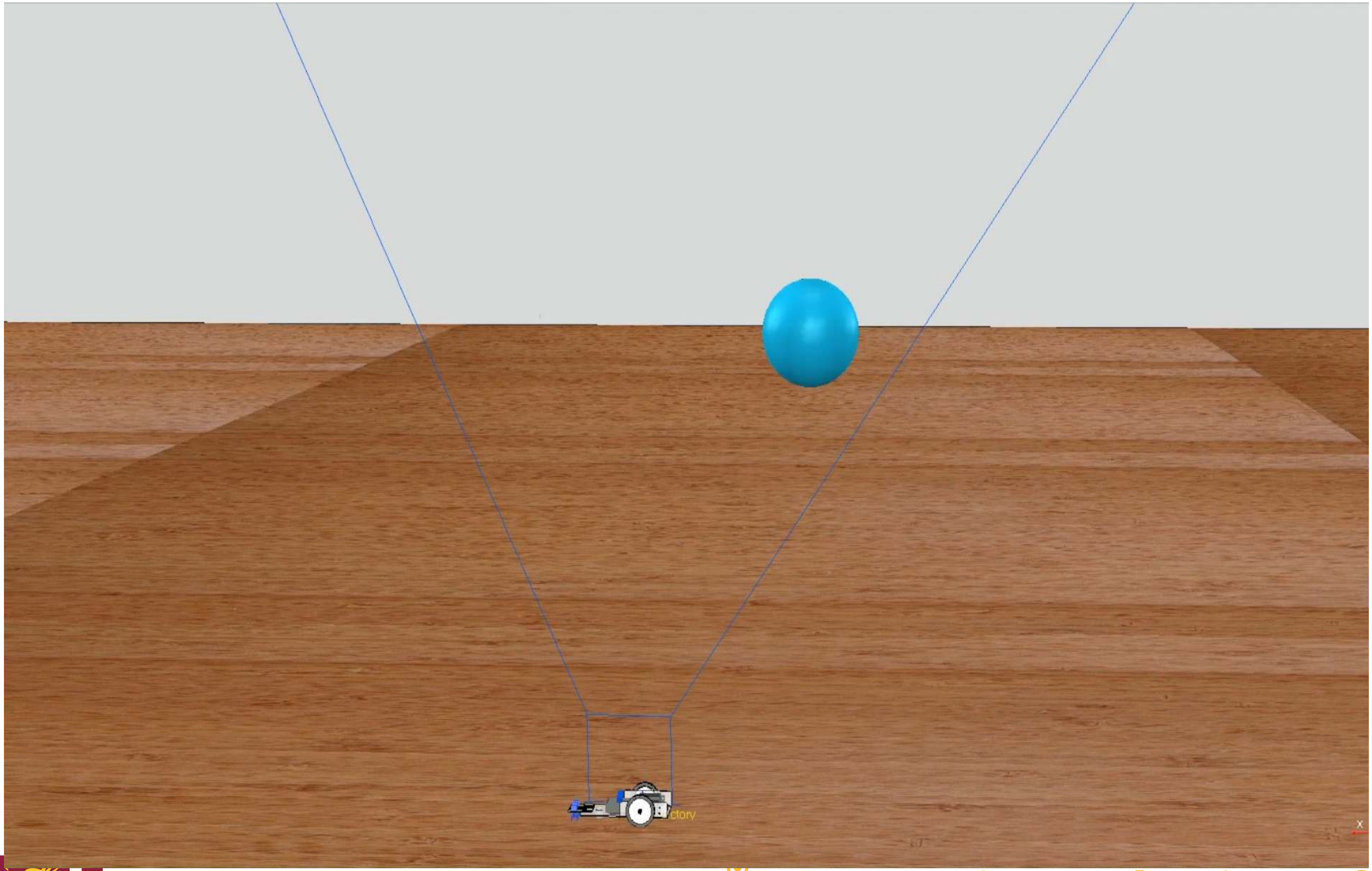
Solution:

- Convolutional RNN that learns the mapping between video input and probability of collision



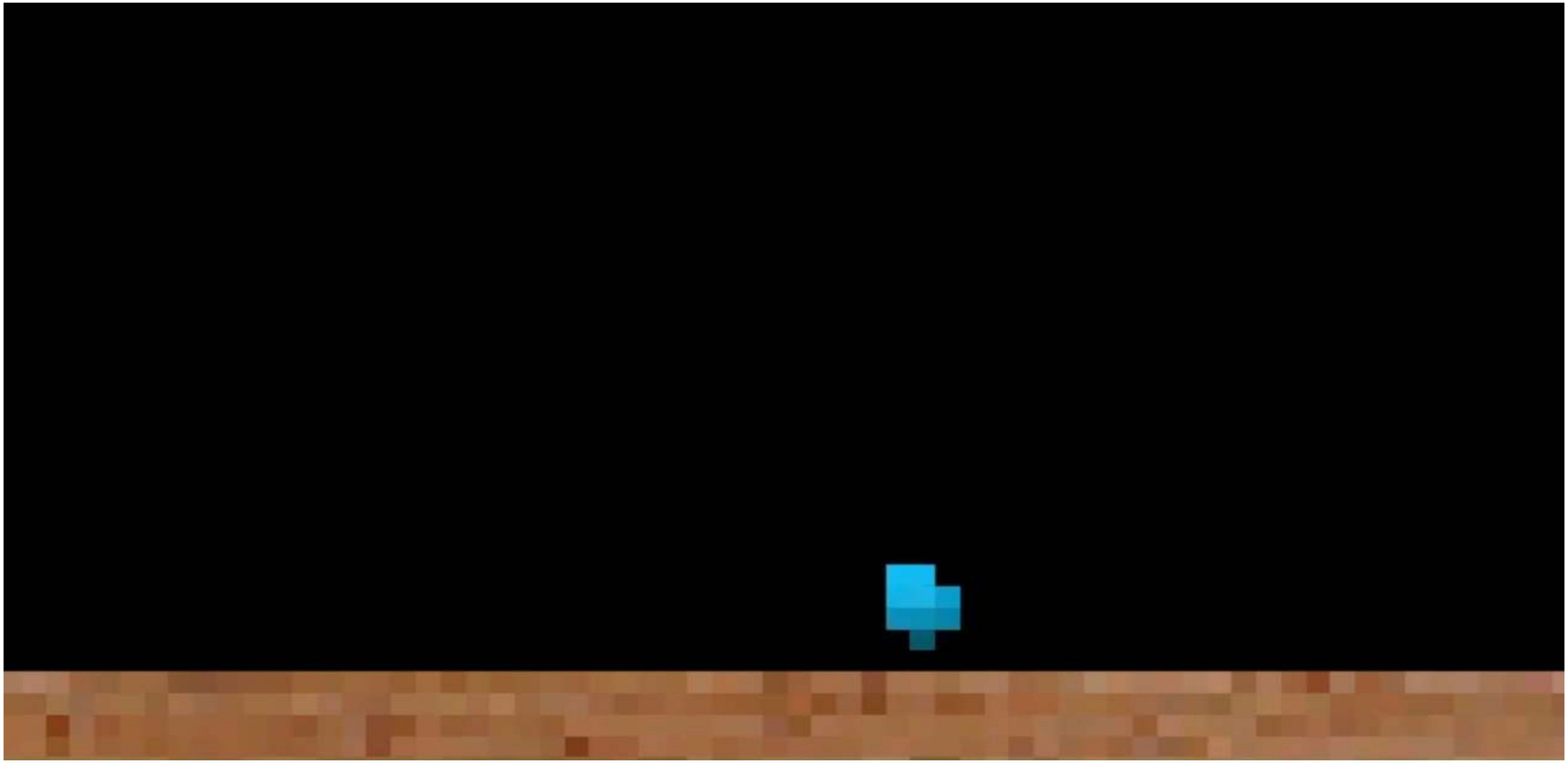
$$Input\ -\ \mathbf{X} \in \mathbb{R}^{T \times C \times L \times W}$$
$$Output - \hat{y} \in \mathbb{R}^1$$

Dodge Ball



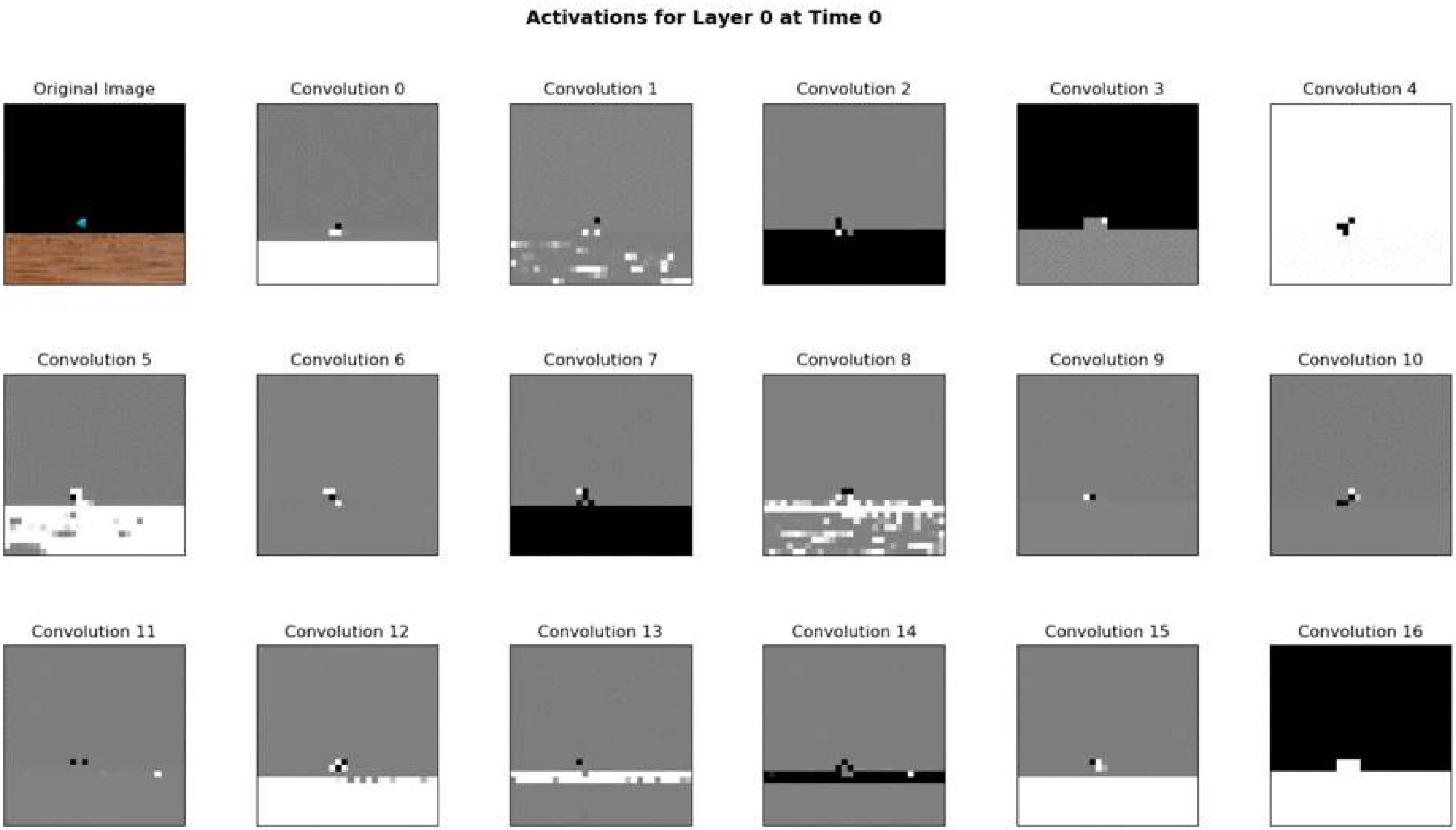
Interactive Robotics Lab

Dodge Ball from Robot Perspective

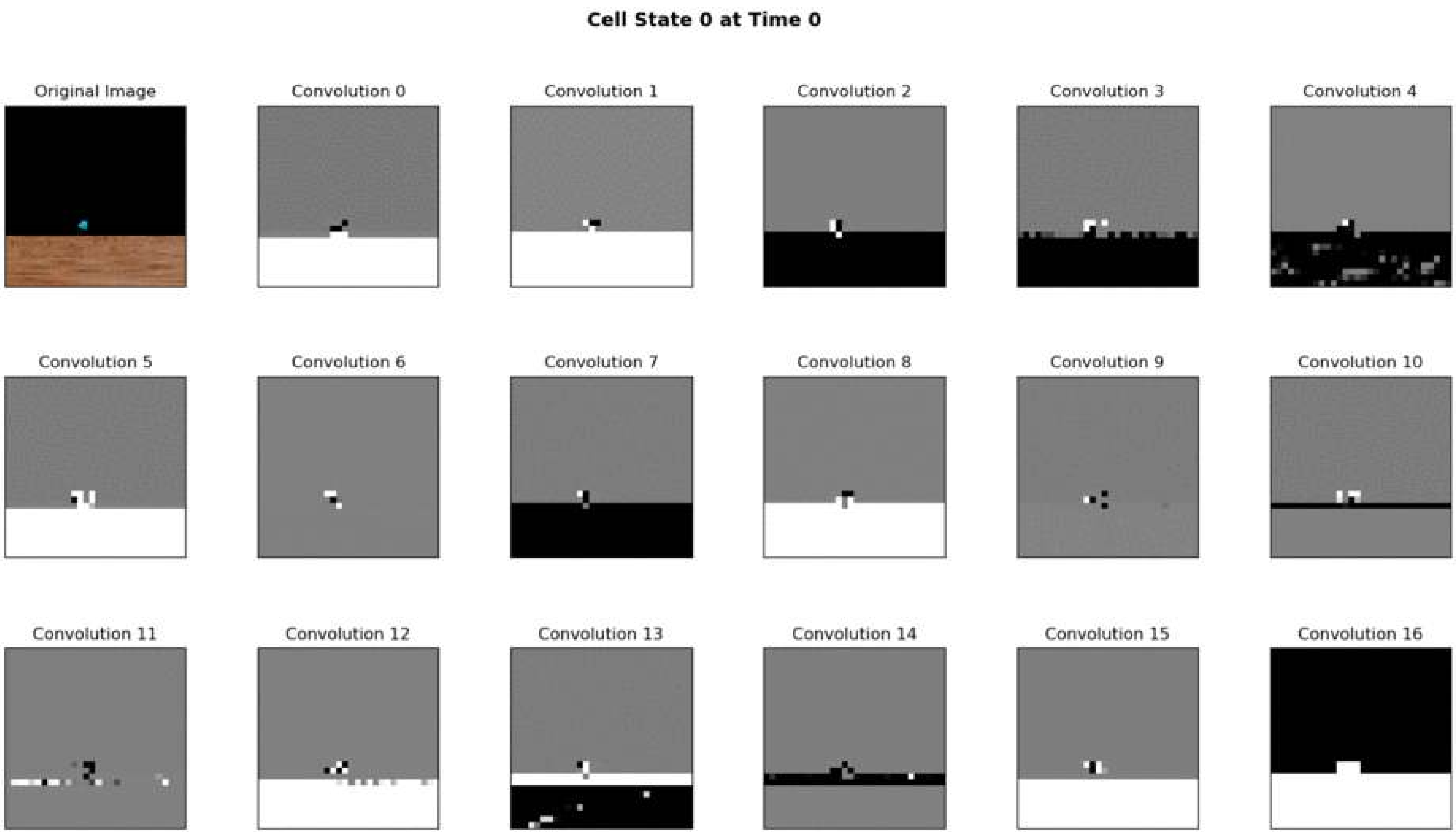


Interactive Robotics Lab

Visualize Hidden Activation Maps in ConvLSTM



Visualize Cell State Activation Maps in ConvLSTM



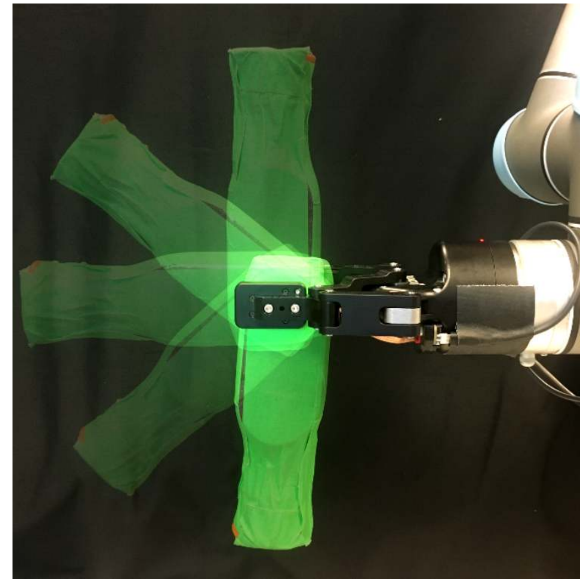
RNNs and Robotics

Goal

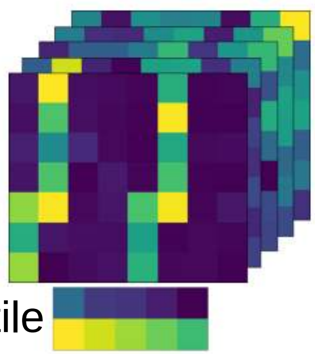
- Utilize slip for dexterous in-hand manipulation of grasped objects

Solution

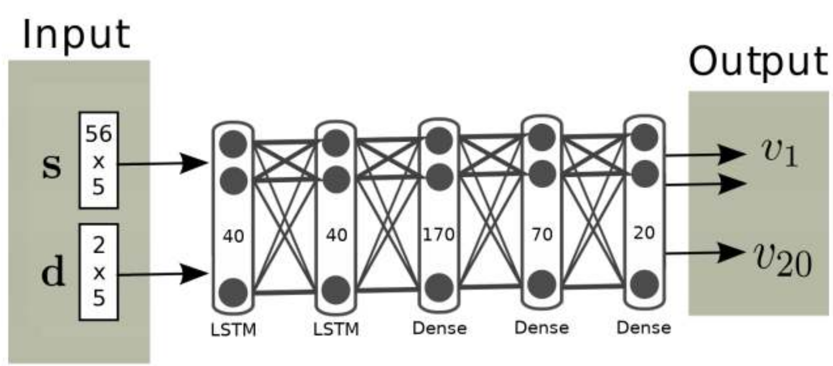
- Predictive RNN model that estimates future poses of grasped object from **past experiences**



Static tactile data (s)



Dynamic tactile data (d)

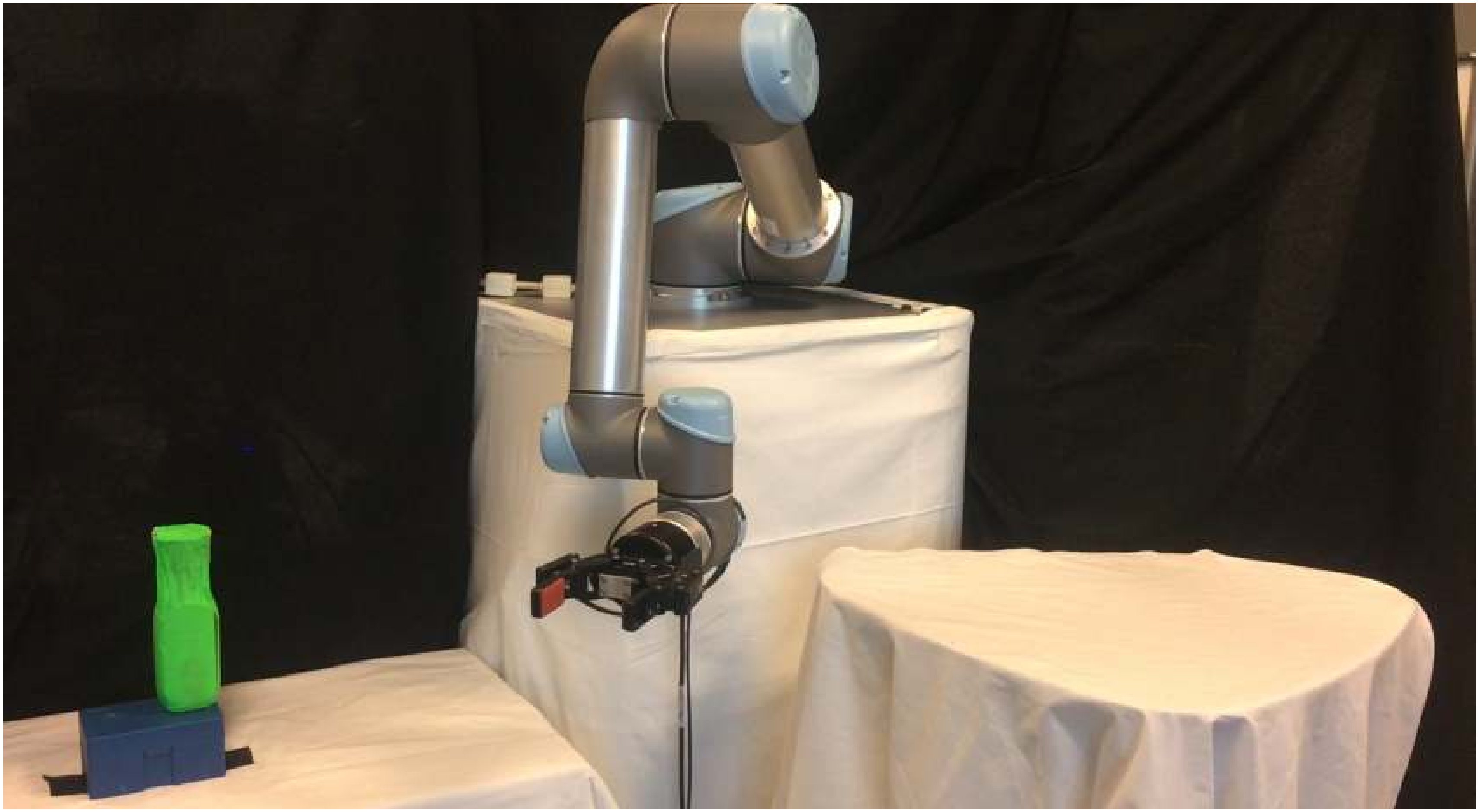


Predicted future poses for the next 20 time-steps

Ground truth captured by accelerometer

[Source] https://simonstepputtis.com/static/paper/icra2018.pdf

RNNs and Robotics



Play Video

ASU

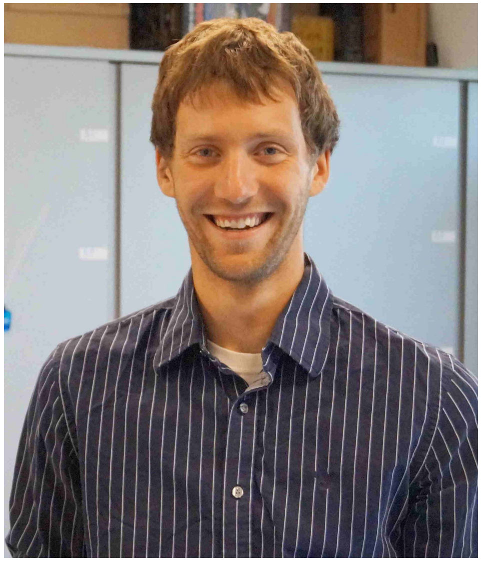
Interactive Robotics Lab

Summary

- We introduced recurrent networks
- Most widely used are LSTMs and GRUs
- Critical for tasks that require **memory**
- Robot may take past states into account during decision-making
- ConvolutionalLSTMs can be used to extract visual features and track them over time

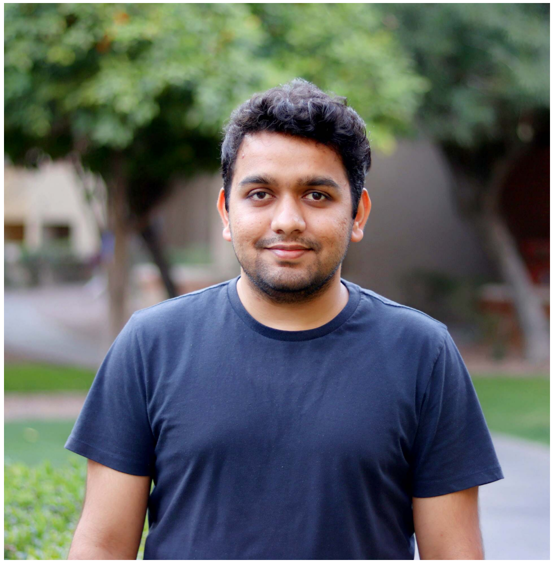
Development Team

- The “Robot Learning” material was developed by



Trevor Barron

Trevor Richardson



Nambi Srivatsav

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