Carnegie Mellon
School of Computer Science

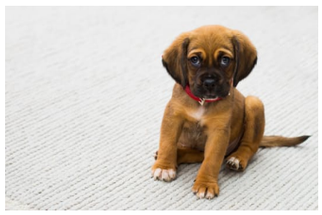
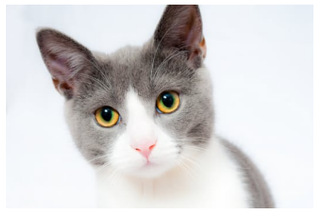
Deep Reinforcement Learning and Control

Learning to learn, Low shot learning

Katerina Fragkiadaki

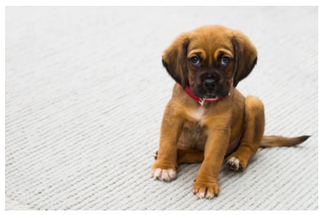
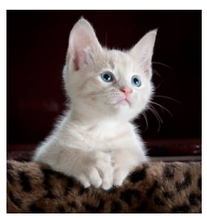
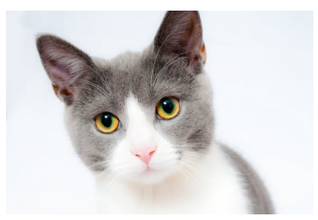
Supervised learning

training

X	Y
	1
	1
	1
	0
	0
	0

Supervised learning

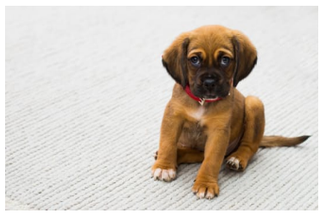
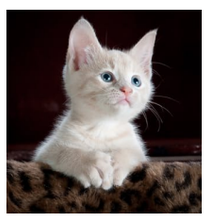
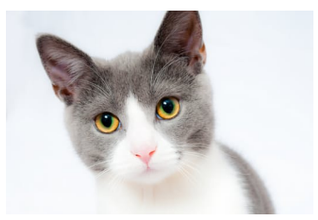
training

X	Y
	1
	1
	1
	0
	0
	0

$$y = f(x;\theta)$$

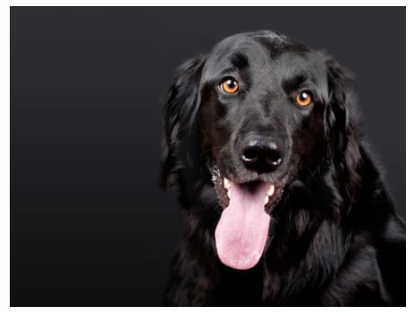
Supervised learning

training

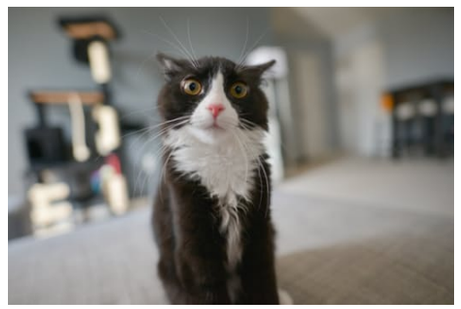
x	Y
	1
	1
	1
	0
	0
	0

$$y = f(x;\theta)$$

test



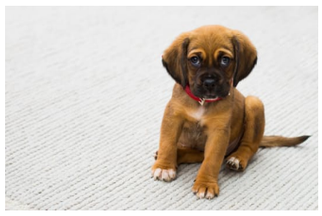
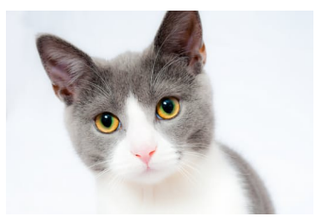
?



?

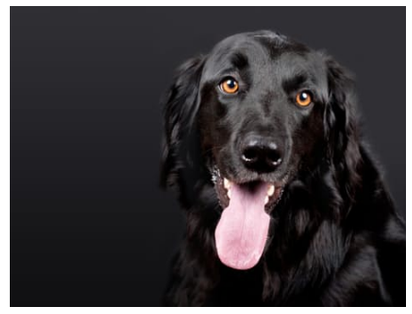
Supervised learning

training

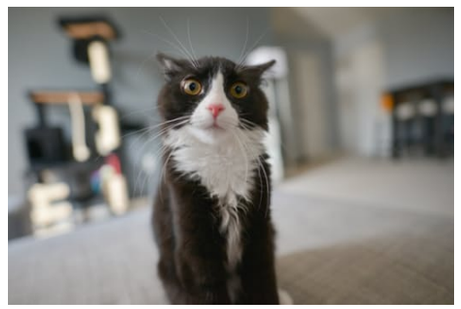
x	Y
	1
	1
	1
	0
	0
	0

$$y = f(x;\theta)$$

test



?

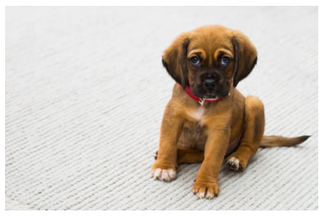
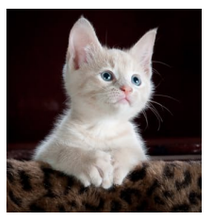
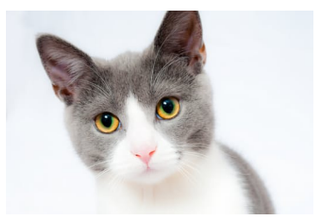


?

Generalization corresponds to the capacity to make predictions about the behavior of the target function at novel points.

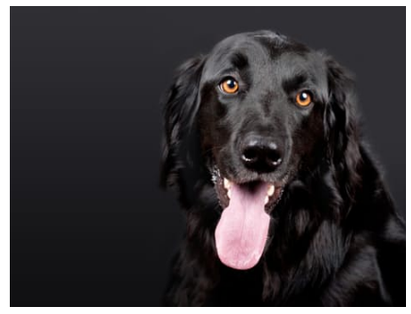
Supervised learning

training

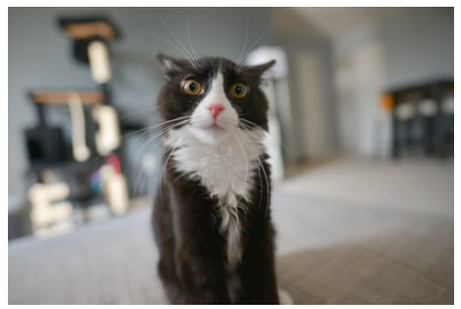
x	Y
	1
	1
	1
	0
	0
	0

$$y = f(x;\theta)$$

test



?



?

Human intelligence though is considered remarkable for its ability to generalize to new tasks, as opposed to new instances of the same task!

Learning to learn (meta-learning)

training		test	
x	Y		
Task-1	Solution-1	New Task	?
Task-2	Solution-2		
Task-3	Solution-3		
		Another New Task	?
Task-4	Solution-4		

Human intelligence though is considered remarkable for its ability to generalize to new tasks, as opposed to new instances of the same task!

Examples: recognizing your classmates, finding the classroom, the exit, the restroom in a new building.

Multi-task/transfer learning VS learning to learn

training		test	
x	Y		
Task-1	Solution-1	New Task	?
Task-2	Solution-2		
Task-3	Solution-3	Another New Task	?
Task-4	Solution-4		

Multi-task learning: how to rely on previous tasks so as to solve the new task faster, e.g., progressive nets (for effective feature sharing and augmenting), or hierarchical RL for re-using skills.

Learning to learn in addition adds learning to the hand designed parts of the above setups, that is how many iterations/examples/experience your need to master the new skill. Mastering the new skill becomes a learning problem itself.

Multi-task/transfer learning VS learning to learn

training		test	
x	Y		
Task-1	Solution-1	New Task	?
Task-2	Solution-2		
Task-3	Solution-3		
		Another New Task	?
Task-4	Solution-4		

Multi-task learning: learn to walk for various speeds towards various directions. I train a policy parameterized also by the goal, at test time I provide a new goal and i expect good performance.

Learning to learn: I train a policy network, that given a particular goal (walking speed and direction), learns to master the skill after K episodes of experience.

Learning to learn: This lecture

- Learning to optimize: learn parameter update rules
- One shot imitation learning
- Learning parameters so that a specific number of update steps under a loss of a new tasks yields good weights
- Learn a policy for learning a new task fast (within a specified window of experience)
- One shot learning using compositional neural network architectures

Learning to learn: This lecture

- Learning to optimize: learn parameter update rules
- One shot imitation learning
- Learning parameters so that a specific number of update steps under a loss of a new tasks yields good weights
- Learn a policy for learning a new task fast (within a specified window of experience)
- One shot learning using compositional neural network architectures

Learning by gradient descent

We have a function parametrized by \theta, and an objective function f(\theta), an initial starting point \theta_0, and want to take steps in our parameter space to minimize our loss function:

$$\theta_{t+1} = \theta_t - \alpha_t \nabla f(\theta_t)$$

Why we need learning the parameter update rule?

A zoo of parameter update rules

A variety of hand-designed update rules guided by human intuition:

gradient descent (GD):

$$\theta = \theta - \eta \cdot \nabla_\theta f(\theta)$$

A zoo of parameter update rules

A variety of hand-designed update rules guided by human intuition:

gradient descent (GD):

$$\theta = \theta - \eta \cdot \nabla_\theta f(\theta)$$

GD with momentum:

$$\nu_t = \gamma \nu_{t-1} + \eta \nabla_\theta f(\theta)$$

$$\theta = \theta - \nu_t$$

Image 2: SGD without momentum

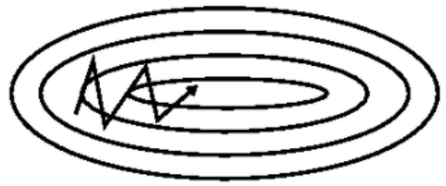


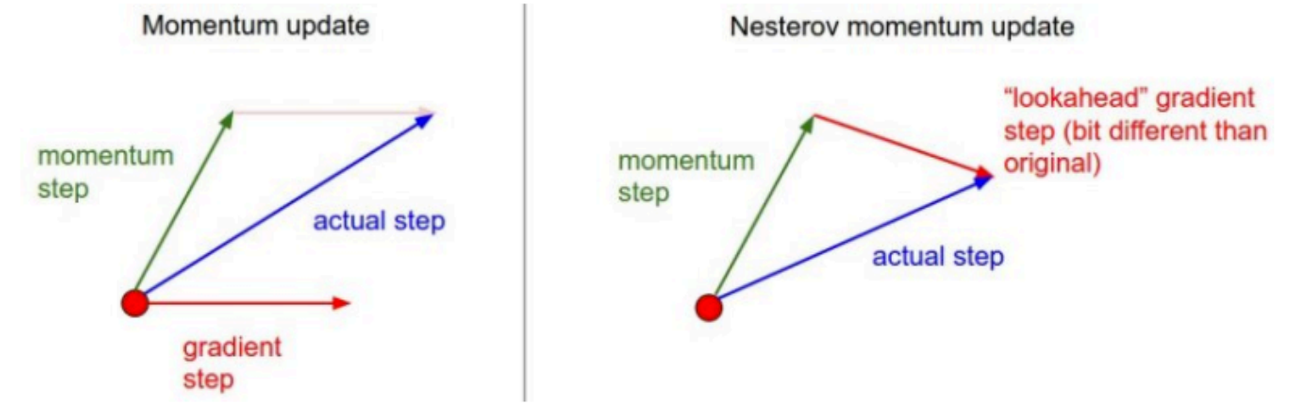
Image 3: SGD with momentum

A zoo of parameter update rules

A variety of hand-designed update rules guided by human intuition:

gradient descent (GD):

$$\theta = \theta - \eta \cdot \nabla_\theta f(\theta)$$



GD with momentum:

$$\nu_t = \gamma \nu_{t-1} + \eta \nabla_\theta f(\theta)$$

$$\theta = \theta - \nu_t$$

Nesterov accelerated gradient

$$\nu_t = \gamma \nu_{t-1} + \eta \nabla_\theta f(\theta - \gamma \nu_{t-1})$$

$$\theta = \theta - \nu_t$$

A zoo of parameter update rules

Adagrad:

It adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters.

A zoo of parameter update rules

Adagrad:

It adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

$G_t \in \mathbb{R}^{d \times d}$ here is a diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients w.r.t. $\theta_i$up to time step $t = 24$ while ϵ is a smoothing term that avoids division by zero (usually on the order of $1e - 8$).

You do not need to tune the l.r.! Problem: towards the end, l.r. becomes infinitely small.

A zoo of parameter update rules

Adagrad:

It adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

$G_t \in \mathbb{R}^{d \times d}$ here is a diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients w.r.t. $\theta_i$up to time step $t = 24$ while ϵ is a smoothing term that avoids division by zero (usually on the order of $1e - 8$).

You do not need to tune the l.r.! Problem: towards the end, l.r. becomes infinitely small.

Adadelta instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size. RMSProp is very similar.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma) g_t^2$$

A zoo of parameter update rules

Adam:

Adaptive Moment Estimation, it keep exponentially decaying average of both shares of gradients and their averages:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t.$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2.$$

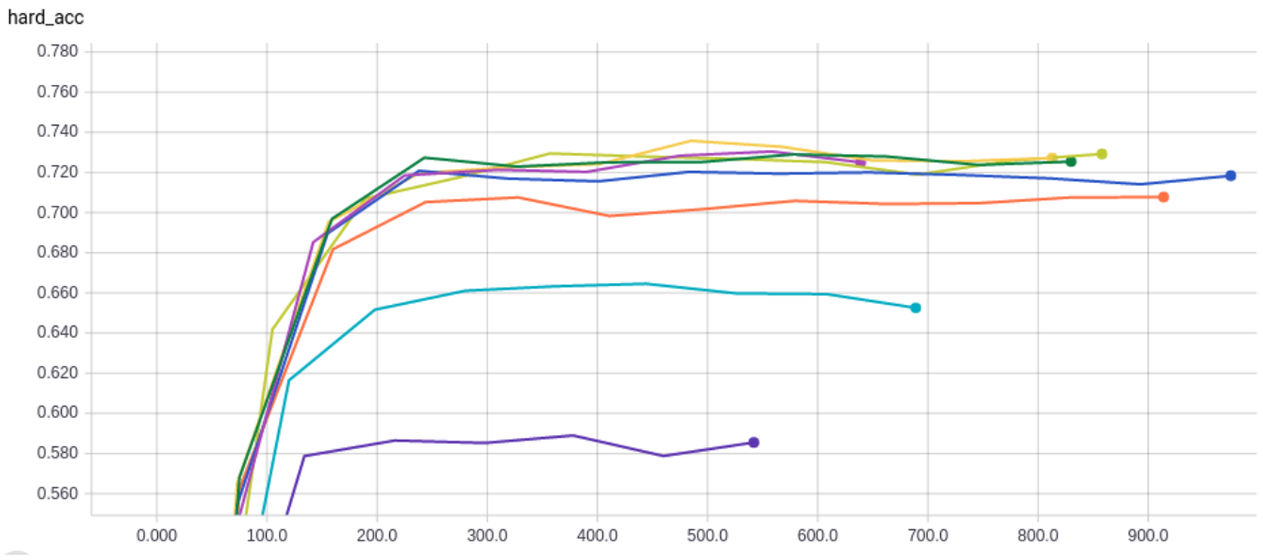
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}.$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}.$$

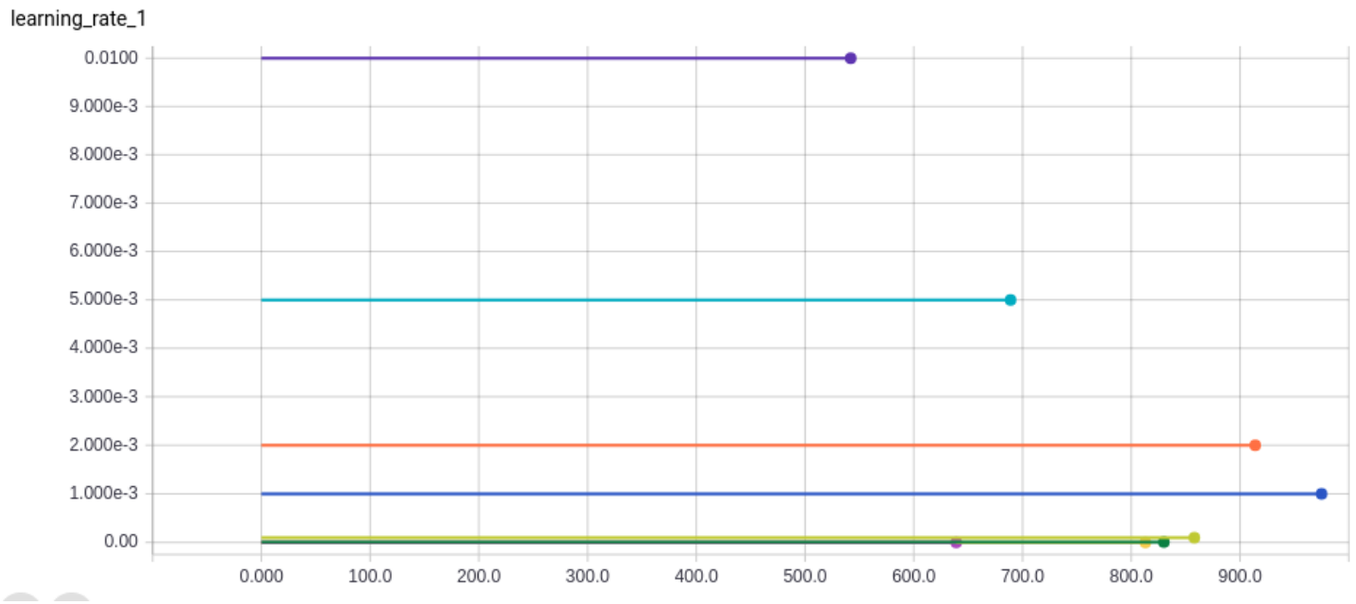
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t.$$

http://sebastianruder.com/optimizing-gradient-descent/index.html#gradientdescentvariants

Learning Rate tuning



Inception v2 in Imagenet classification task



Results by Yijie Wang

Learning to learn by gradient descent by gradient descent

Marcin Andrychowicz[1], Misha Denil[1], Sergio Gómez Colmenarejo[1], Matthew W. Hoffman[1], David Pfau[1], Tom Schaul[1], Brendan Shillingford[1,2], Nando de Freitas[1,2,3]

[1]Google DeepMind [2]University of Oxford [3]Canadian Institute for Advanced Research

marcin.andrychowicz@gmail.com
{mdenil,sergomez,mwhoffman,pfau,schaul}@google.com
brendan.shillingford@cs.ox.ac.uk, nandodefreitas@google.com

Idea: what if we learnt such an update rule so that we get the most out of our parameter updates?

We will parametrize the optimizer! ϕ is our learnable parameters!

$$\theta_{t+1} = \theta_t + g_t(\nabla f(\theta_t), \phi)$$

Learning [Learning by gradient descent]

Idea: what if we learnt such an update rule so that we get the most out of our parameter updates?

We will parametrize the optimizer! \Phi is our learnable parameters!

$$\theta_{t+1} = \theta_t + g_t(\nabla f(\theta_t), \phi)$$

Loss function for the optimizer: the resulting \theta* weights should do well:

$$\mathcal{L}(\phi) = \mathbb{E}_f \Big[f\big(\theta^*(f, \phi)\big) \Big]$$

Apart from the final value \theta*, let's consider a whole trajectory of thetas doing well:

$$\mathcal{L}(\phi) = \mathbb{E}_f \left[\sum_{t=1}^{T} w_t f(\theta_t) \right]$$

Learning [Learning by gradient descent]

Idea: what if we learnt such an update rule so that we get the most out of our parameter updates?

We will parametrize the optimizer! ϕ is our learnable parameters!

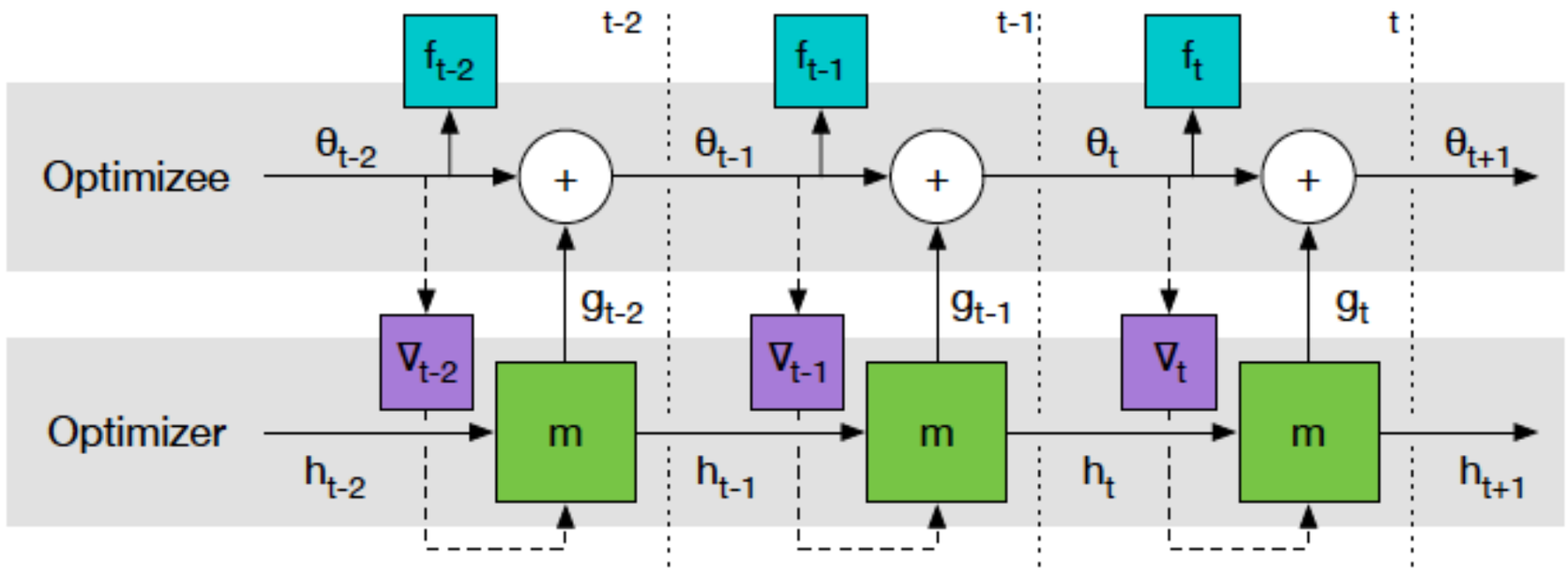
$$\theta_{t+1} = \theta_t + g_t(\nabla f(\theta_t), \phi)$$

Loss function for the optimizer: the resulting θ^* weights should do well:

$$\mathcal{L}(\phi) = \mathbb{E}_f \Big[f\big(\theta^*(f, \phi)\big) \Big]$$

Learning [Learning by gradient descent] by gradient descent

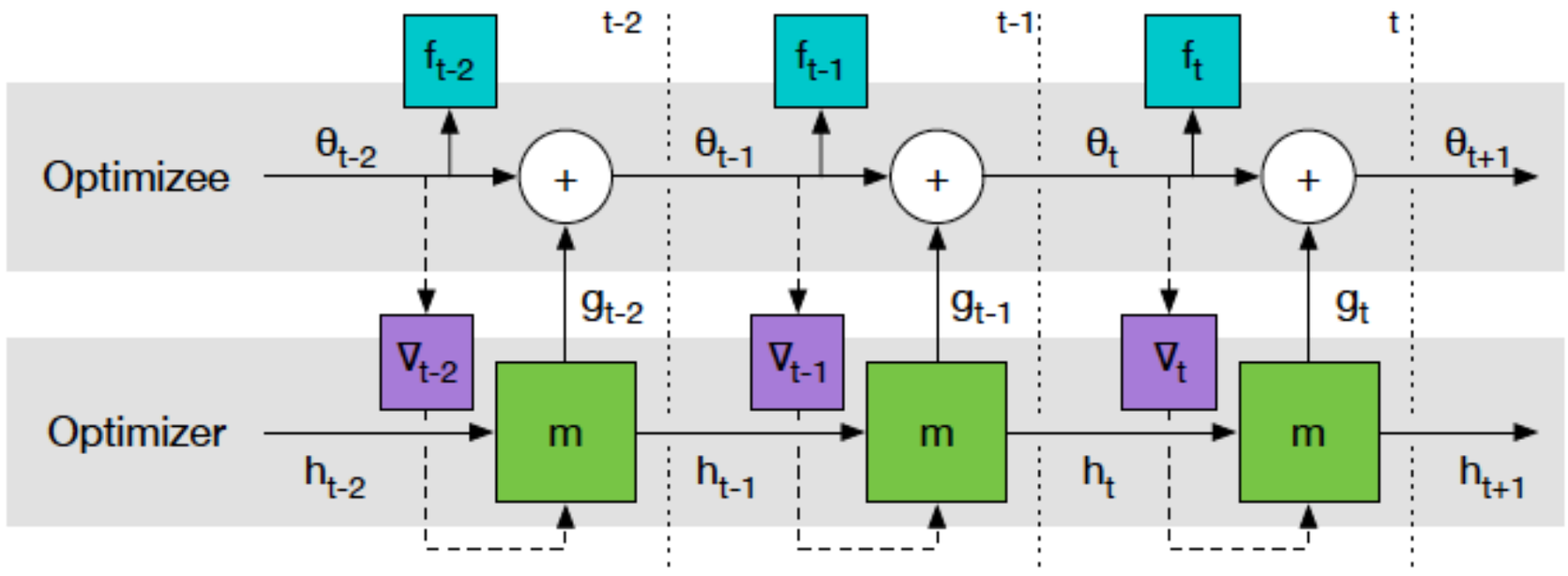
We will update \phi using gradient descent! Sample some objective f, and initial \theta_0 and propagate gradients through the following computational graph:



$$\begin{bmatrix} g_t \\ h_{t+1} \end{bmatrix} = m(\nabla_t, h_t, \phi)$$

Learning [Learning by gradient descent] by gradient descent

We will update \phi using gradient descent! Sample some objective f, and initial \theta_0 and propagate gradients through the following computational graph:



Only through the solid edges! The gradient of f do not depend of how well the optimization is going! We only update \phi

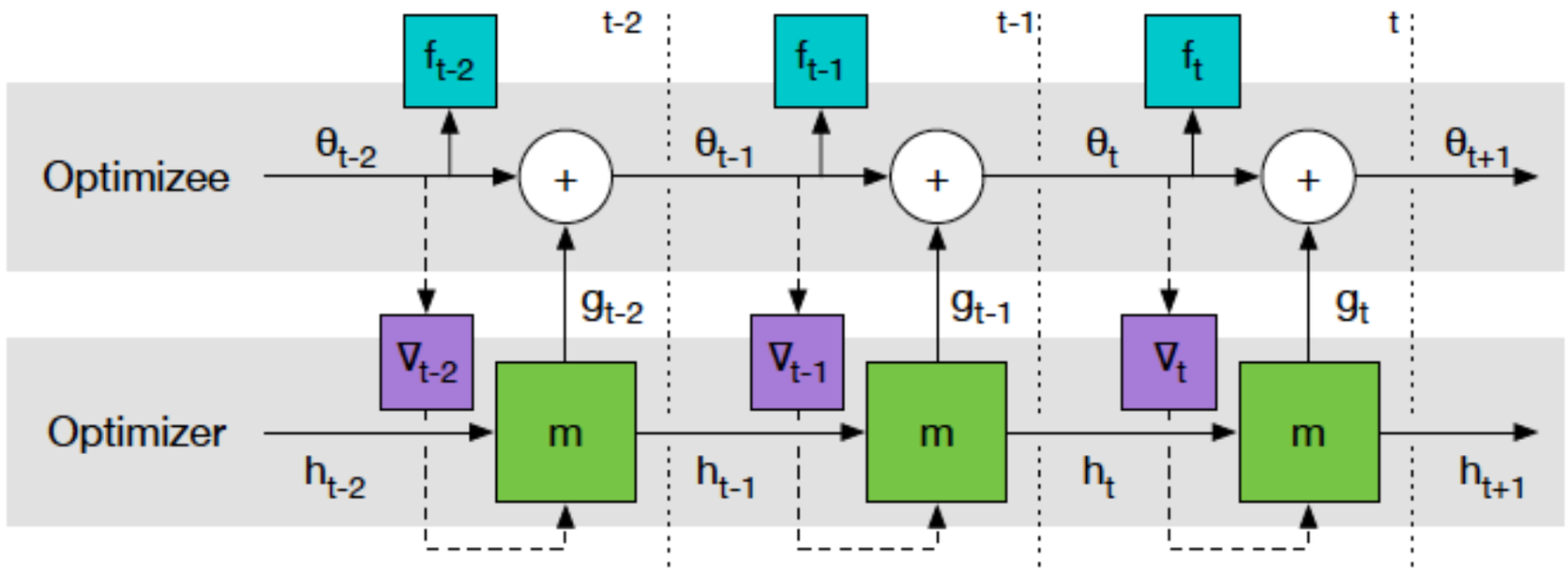
fast weights: \theta

slow weights: \phi

$$\begin{bmatrix} g_t \\ h_{t+1} \end{bmatrix} = m(\nabla_t, h_t, \phi)$$

Learning [Learning by gradient descent] by gradient descent

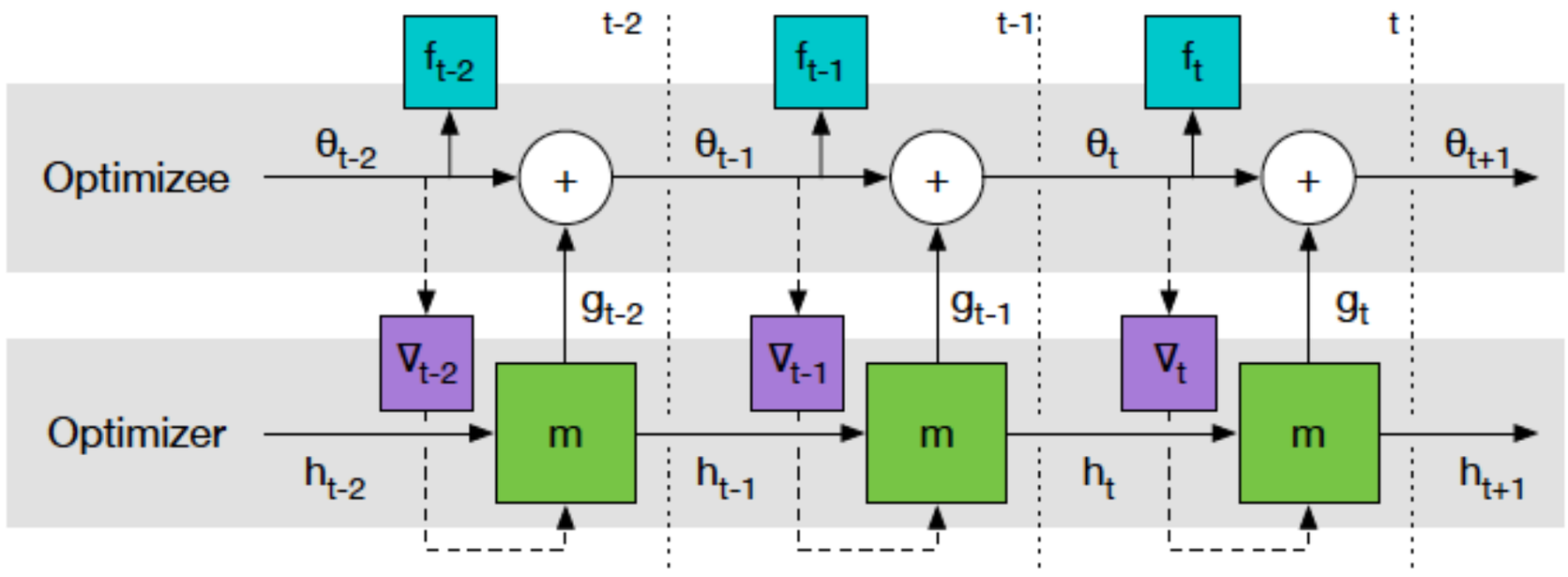
Our optimizer is actually a Recurrent Neural network: h is the hidden state which changes from every update step and remembers information regarding past gradients:



$$\begin{bmatrix} g_t \\ h_{t+1} \end{bmatrix} = m(\nabla_t, h_t, \phi)$$

Learning [Learning by gradient descent] by gradient descent

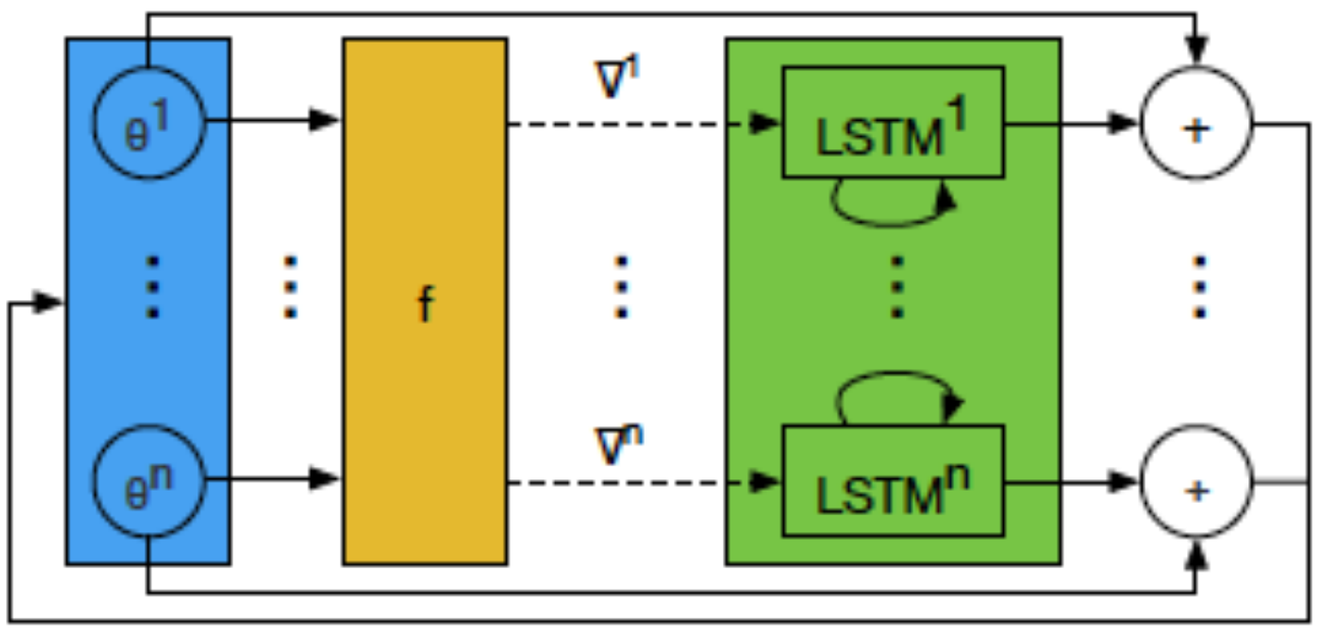
Our optimizer is actually a Recurrent Neural network: h is the hidden state which changes from every update step and remembers information regarding past gradients (of f w.r.t. θ)



$$\begin{bmatrix} g_t \\ h_{t+1} \end{bmatrix} = m(\nabla_t, h_t, \phi)$$

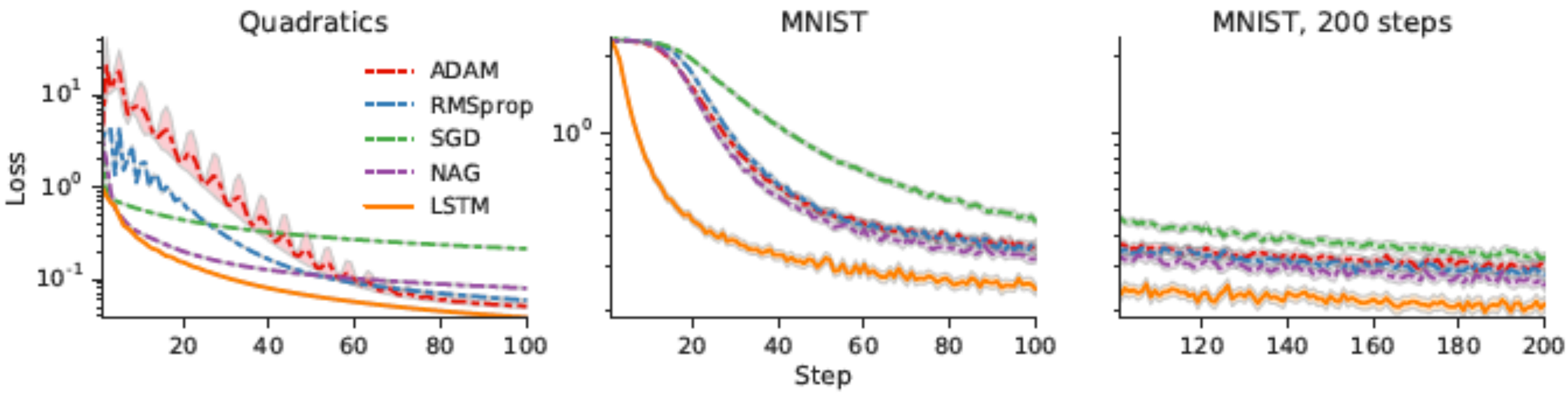
Learning [Learning by gradient descent] by gradient descent

Our optimizer is actually a Recurrent Neural network: h is the hidden state which changes from every update step and remembers information regarding past gradients (of f w.r.t. θ)

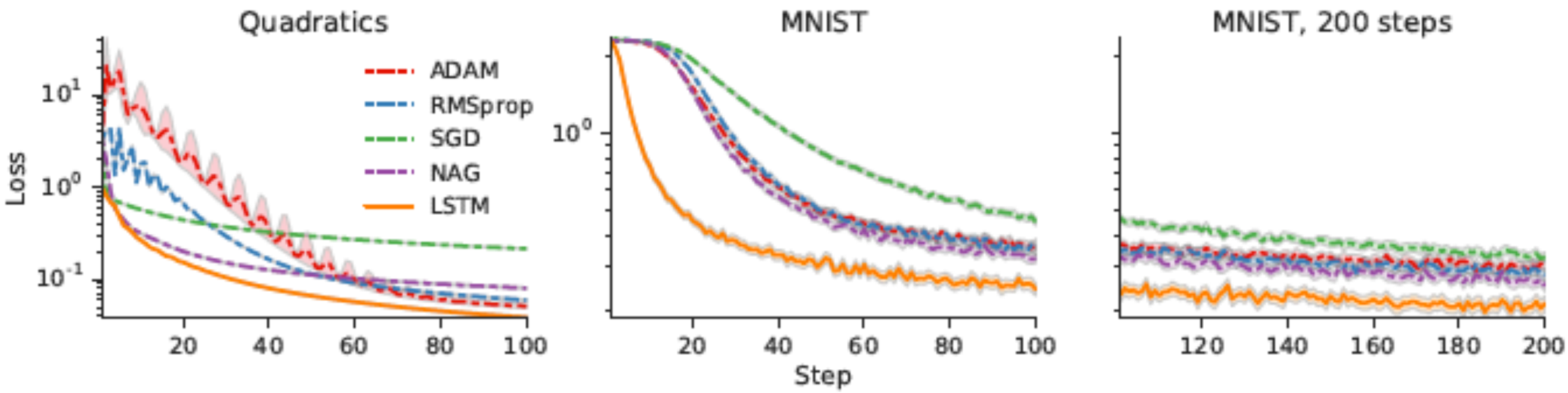


We apply the update rule in each parameter in isolation, in this way we can transfer the learnt update rule across problems of different type of networks. Each LSTM optimizer has the same ϕ but different hidden state!

Comparing learnt and hand crafted update rules



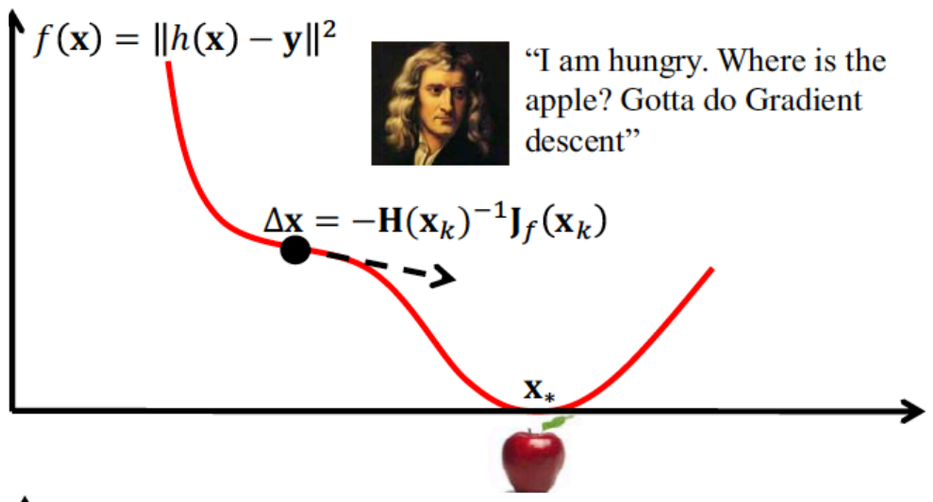
Comparing learnt and hand crafted update rules

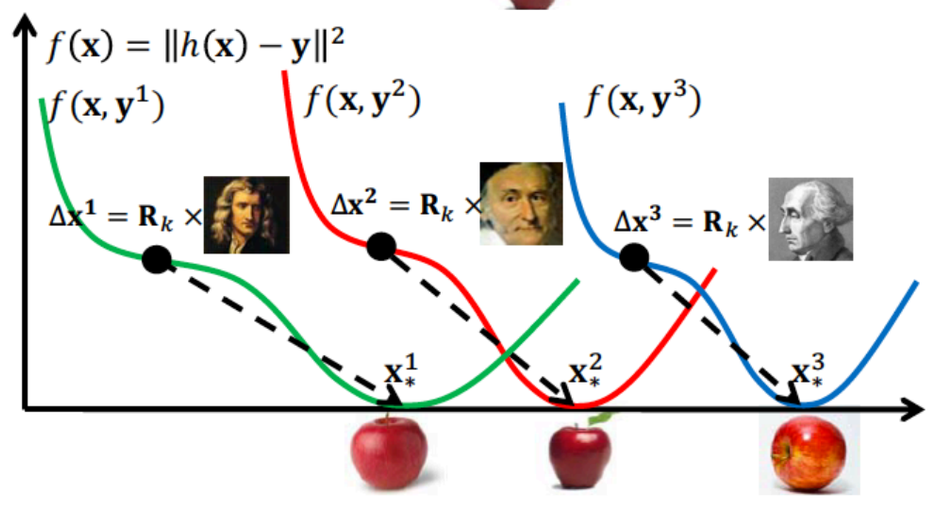


These may look like toy examples..

Supervised Descent Method and its Applications to Face Alignment

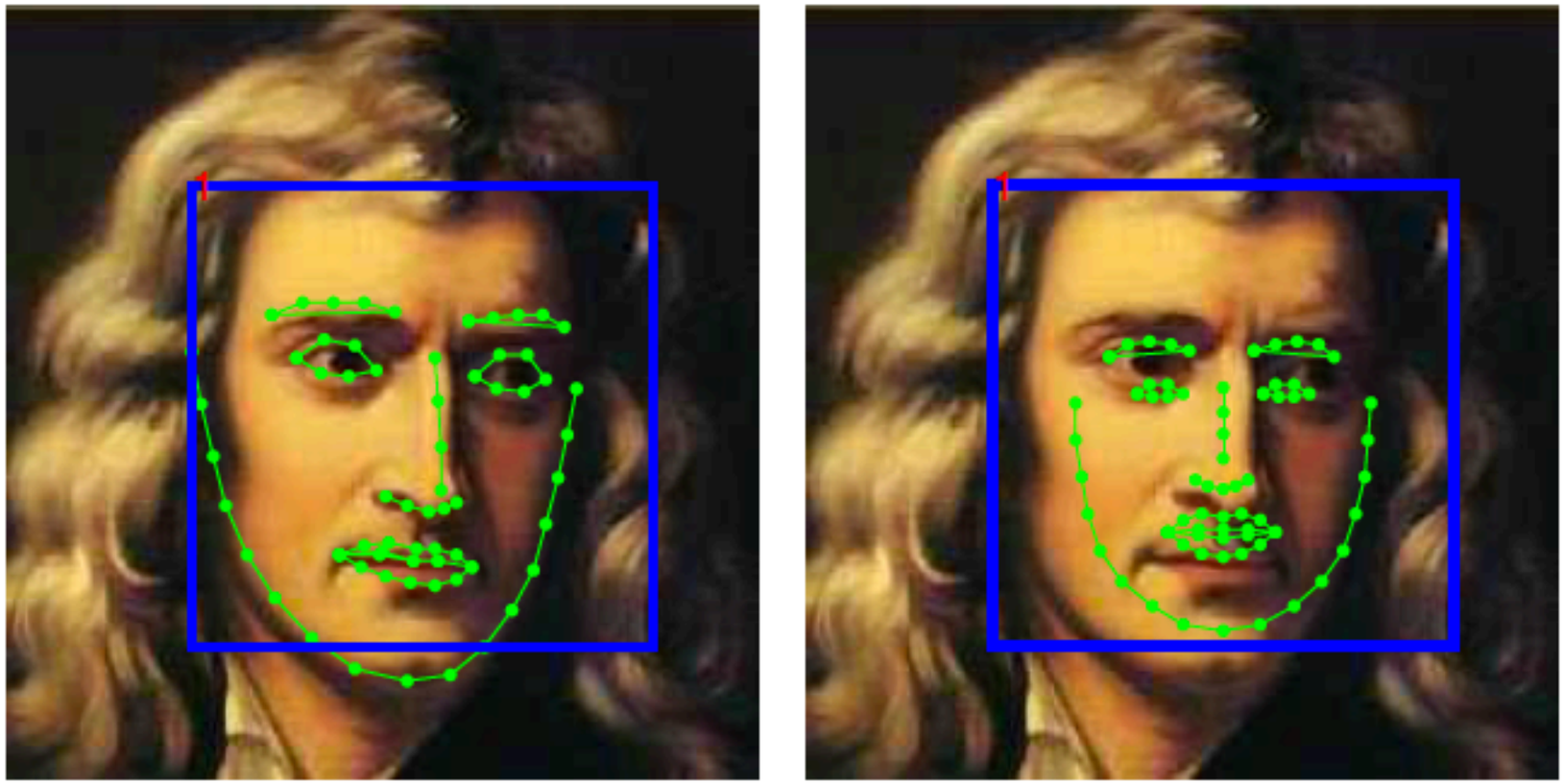
Xuehan Xiong **Fernando De la Torre**
The Robotics Institute, Carnegie Mellon University, Pittsburgh PA, 15213
xxiong@andrew.cmu.edu ftorre@cs.cmu.edu





Face alignment

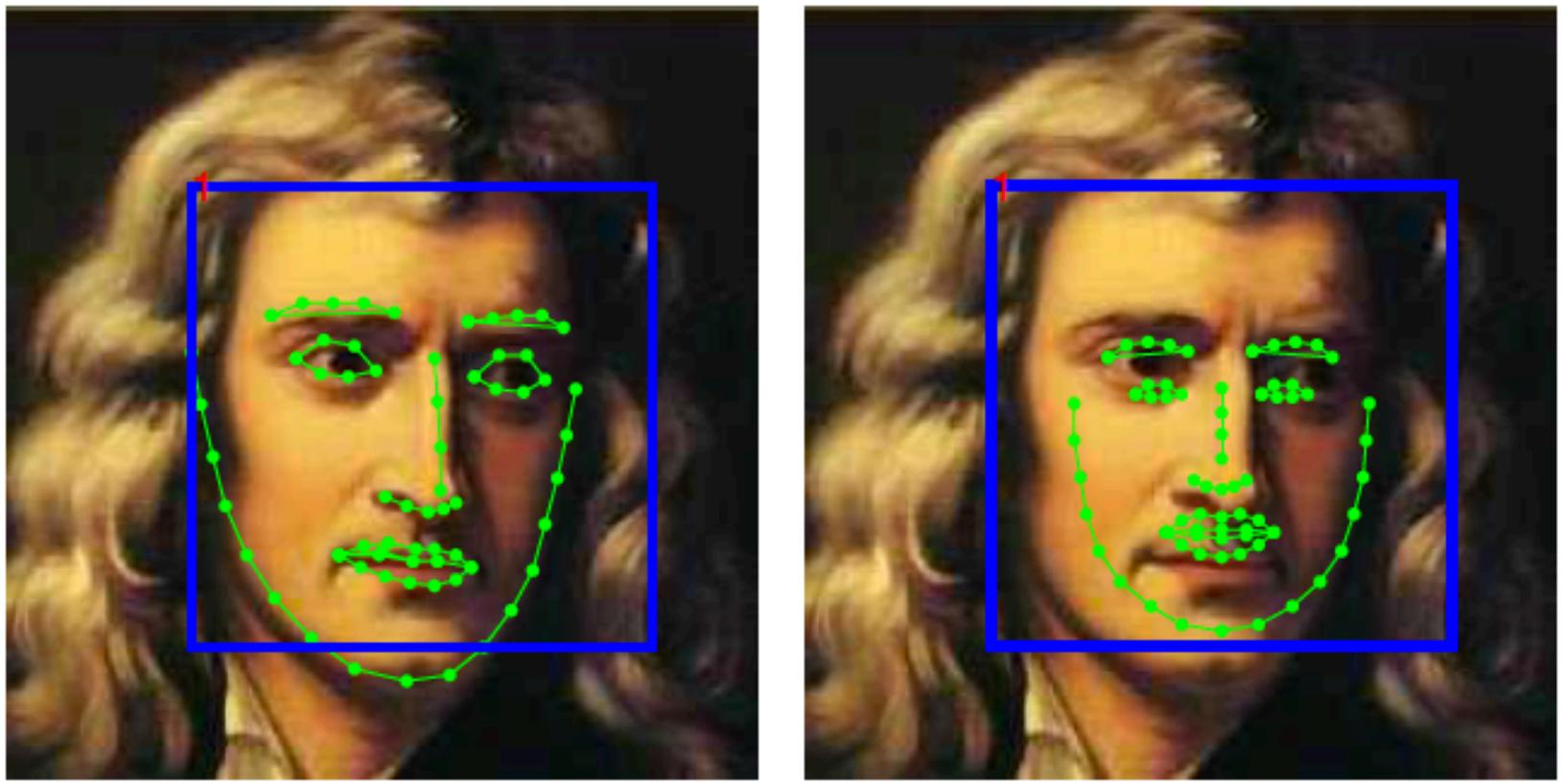
Task: starting from an initial landmark template configuration $\mathbf{x}_0$, get to desired alignment $\mathbf{x}_*$



(a) $\mathbf{x}_*$ (b) $\mathbf{x}_0$

Face alignment

Task: starting from an initial landmark template configuration $\mathbf{x}_0$, get to desired alignment $\mathbf{x}_*$



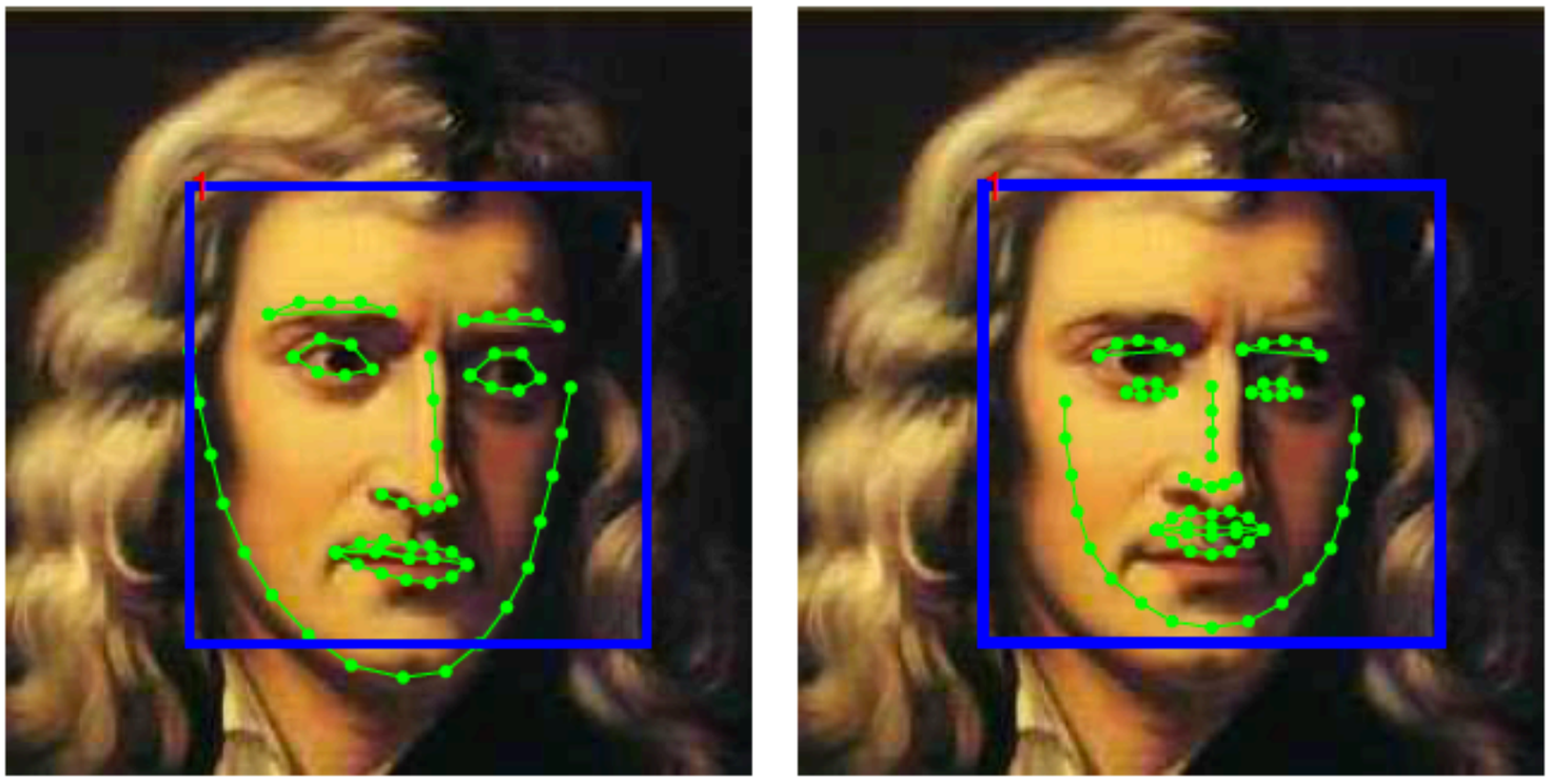
(a) $\mathbf{x}_*$ (b) $\mathbf{x}_0$

Loss function:

$$f(\mathbf{x}_0 + \Delta\mathbf{x}) = \|\mathbf{h}(\mathbf{d}(\mathbf{x}_0 + \Delta\mathbf{x})) - \boldsymbol{\phi}_*\|_2^2$$

Face alignment

Task: starting from an initial landmark template configuration $\mathbf{x}_0$, get to desired alignment $\mathbf{x}_*$



(a) $\mathbf{x}_*$ (b) $\mathbf{x}_0$

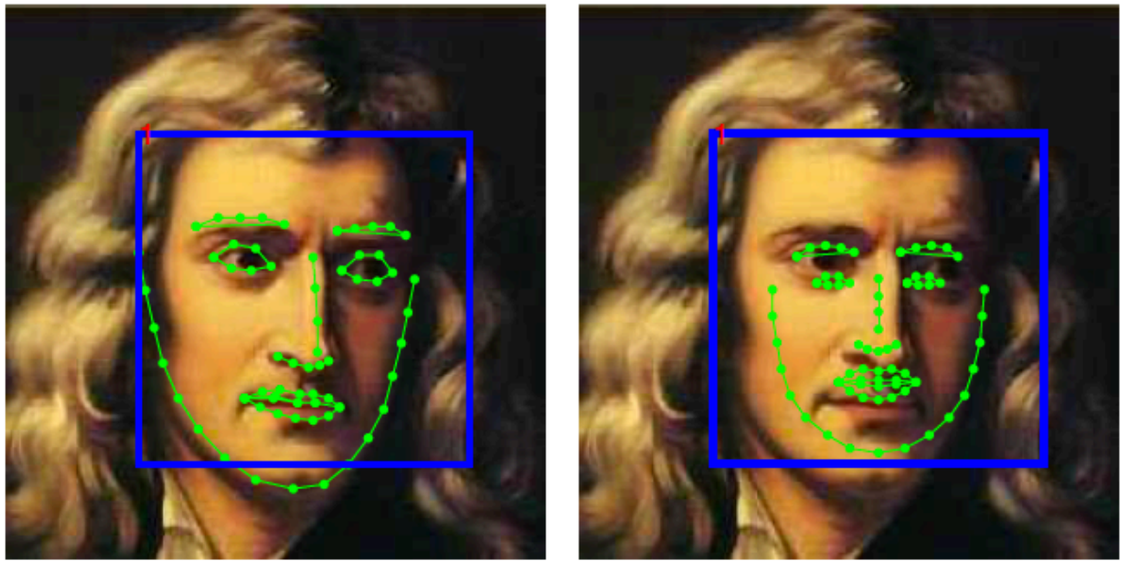
Loss function:

$$f(\mathbf{x}_0 + \Delta\mathbf{x}) = \|\mathbf{h}(\mathbf{d}(\mathbf{x}_0 + \Delta\mathbf{x})) - \boldsymbol{\phi}_*\|_2^2$$

$\mathbf{h}(\mathbf{d}(\mathbf{x}_0 + \Delta\mathbf{x}))$: SIFT features extracted around the landmarks

Newton’s method for face alignment

Task: starting from an initial landmark template configuration x_0, get to desired alignment x_*



(a) $\mathbf{x}_*$ (b) $\mathbf{x}_0$

$$f(\mathbf{x}_0 + \Delta\mathbf{x}) = \|\mathbf{h}(\mathbf{d}(\mathbf{x}_0 + \Delta\mathbf{x})) - \boldsymbol{\phi}_*\|_2^2$$

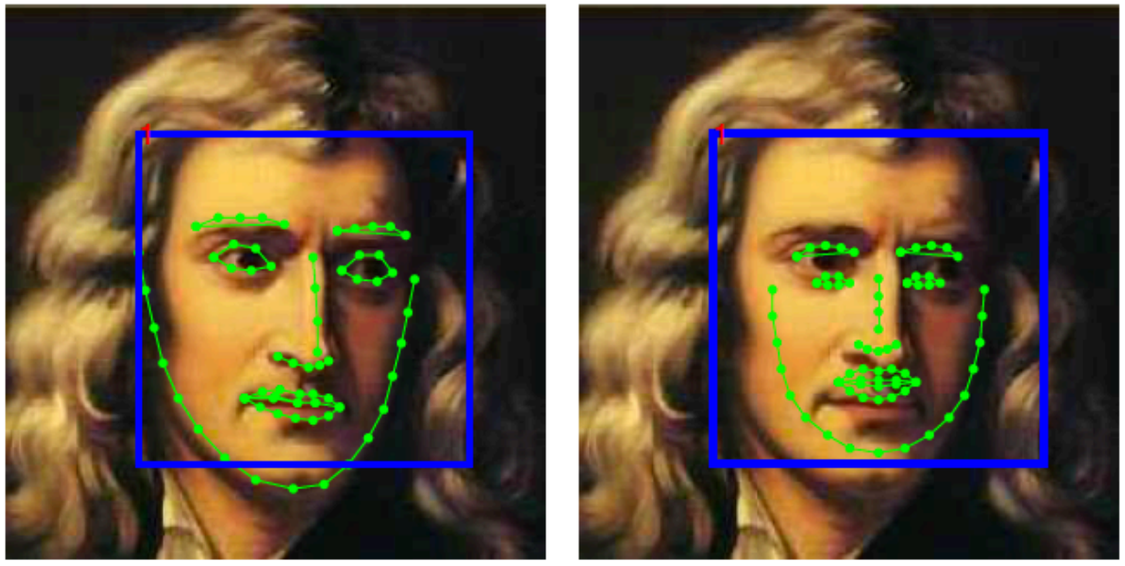
2nd order Taylor expansion:

$$f(\mathbf{x}_0 + \Delta\mathbf{x}) \approx f(\mathbf{x}_0) + \mathbf{J}_f(\mathbf{x}_0)^\top \Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^\top \mathbf{H}(\mathbf{x}_0)\Delta\mathbf{x}$$

$$\Delta\mathbf{x}_1 = -\mathbf{H}^{-1}\mathbf{J}_f = -2\mathbf{H}^{-1}\mathbf{J}_\mathbf{h}^\top(\boldsymbol{\phi}_0 - \boldsymbol{\phi}_*)$$

Newton’s method for face alignment

Task: starting from an initial landmark template configuration x_0, get to desired alignment x_*



(a) $\mathbf{x}_*$ (b) $\mathbf{x}_0$

$$f(\mathbf{x}_0 + \Delta\mathbf{x}) = \|\mathbf{h}(\mathbf{d}(\mathbf{x}_0 + \Delta\mathbf{x})) - \boldsymbol{\phi}_*\|_2^2$$

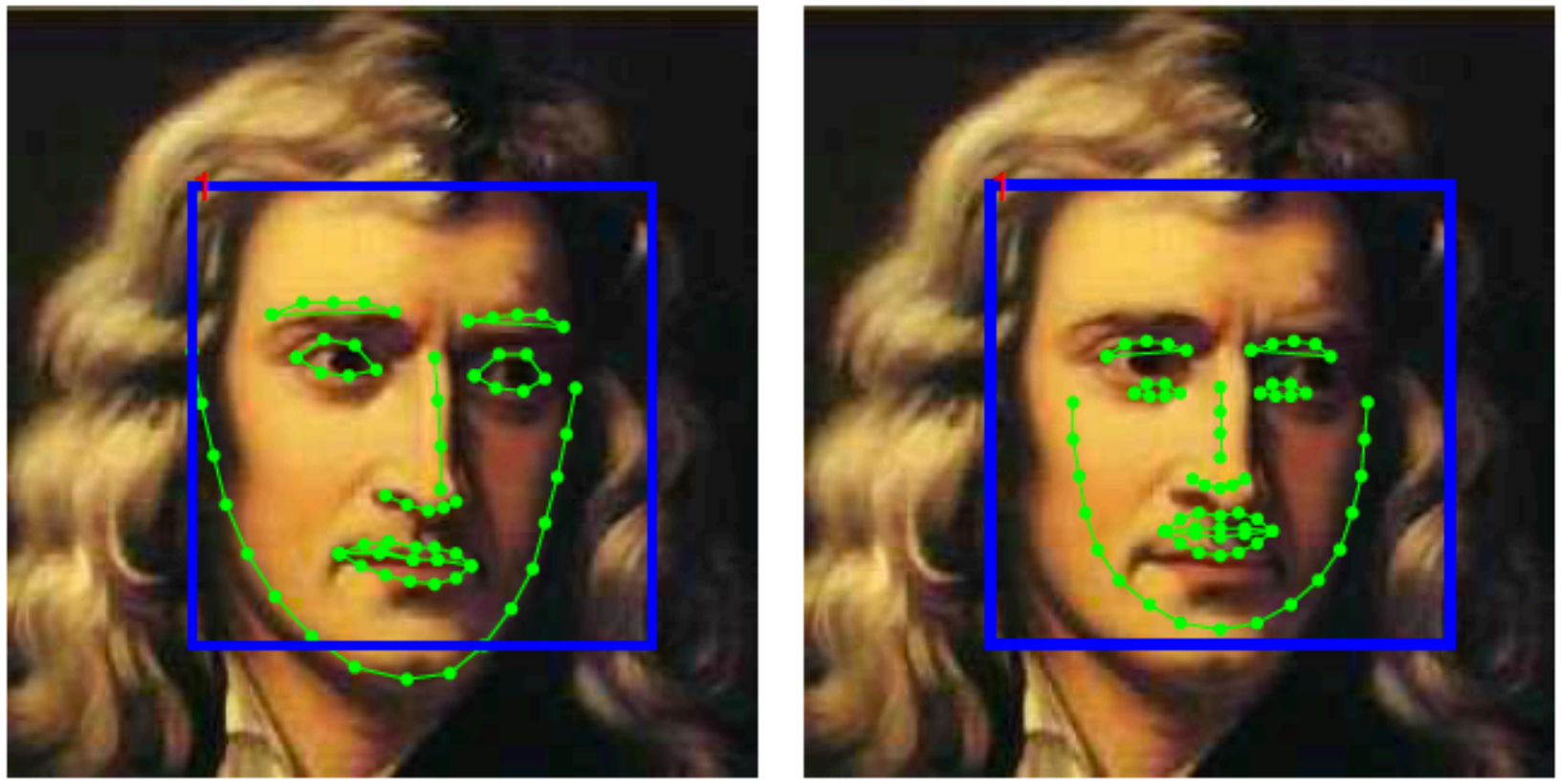
2nd order Taylor expansion:

$$f(\mathbf{x}_0 + \Delta\mathbf{x}) \approx f(\mathbf{x}_0) + \mathbf{J}_f(\mathbf{x}_0)^\top \Delta\mathbf{x} + \frac{1}{2}\Delta\mathbf{x}^\top \mathbf{H}(\mathbf{x}_0)\Delta\mathbf{x}$$

$$\Delta\mathbf{x}_1 = -\mathbf{H}^{-1}\mathbf{J}_f = -2\mathbf{H}^{-1}\mathbf{J}_\mathbf{h}^\top(\boldsymbol{\phi}_0 - \boldsymbol{\phi}_*)$$

$$\Delta\mathbf{x}_1 = \mathbf{R}_0\boldsymbol{\phi}_0 + \mathbf{b}_0 \qquad \mathbf{x}_k = \mathbf{x}_{k-1} + \mathbf{R}_{k-1}\boldsymbol{\phi}_{k-1} + \mathbf{b}_{k-1}$$

Supervised Newton's method



(a) $\mathbf{x}_*$ (b) $\mathbf{x}_0$

Idea: instead of using the Hessian and Jacobian to estimate R_k,b_k, let's estimate it from training data: pairs of images and desired landmark locations x*

$$\underset{\mathbf{R}_0,\mathbf{b}_0}{\arg\min} \sum_{\mathbf{d}^i} \sum_{\mathbf{x}_0^i} \|\Delta \mathbf{x}_*^i - \mathbf{R}_0 \boldsymbol{\phi}_0^i - \mathbf{b}_0\|^2$$

Supervised Newton's method

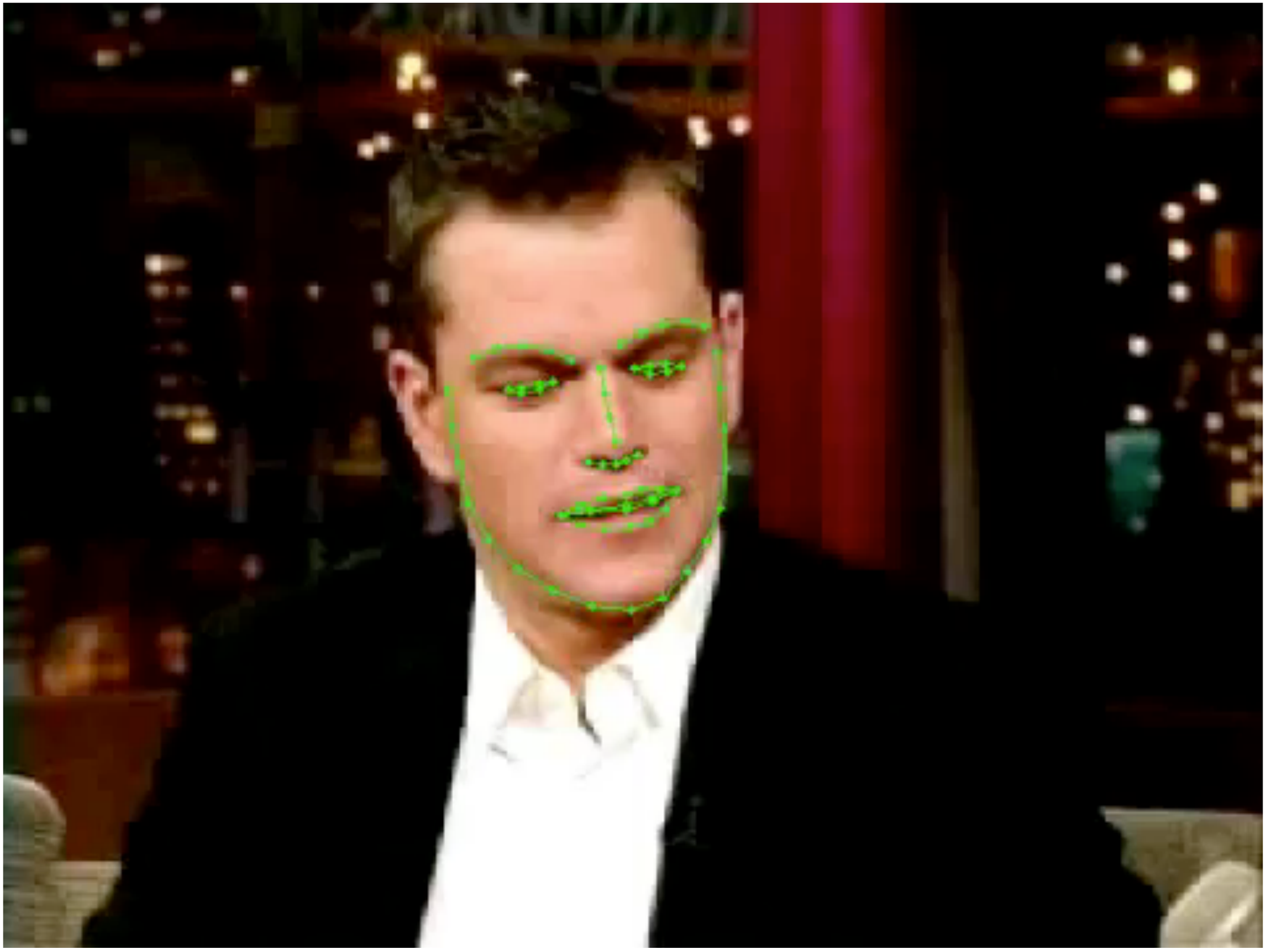
- At each iteration, move the landmarks according to predicted $\mathbf{R}_{k-1}, \mathbf{b}_{k-1}$

$$\mathbf{x}_k = \mathbf{x}_{k-1} + \mathbf{R}_{k-1}\boldsymbol{\phi}_{k-1} + \mathbf{b}_{k-1}$$

- And minimize for the remaining residual:

$$\underset{\mathbf{R}_k, \mathbf{b}_k}{\arg\min} \sum_{\mathbf{d}^i} \sum_{\mathbf{x}_k^i} \| \Delta \mathbf{x}_*^{ki} - \mathbf{R}_k \boldsymbol{\phi}_k^i - \mathbf{b}_k \|^2$$

Supervised Newton's method



Learning to learn: This lecture

- Learning to optimize: learn parameter update rules
- One shot imitation learning
- Learning parameters so that a specific number of update steps under a loss of a new tasks yields good weights
- Learn a policy for learning a new task fast (within a specified window of experience)
- One shot learning using compositional neural network architectures

One-Shot Imitation Learning

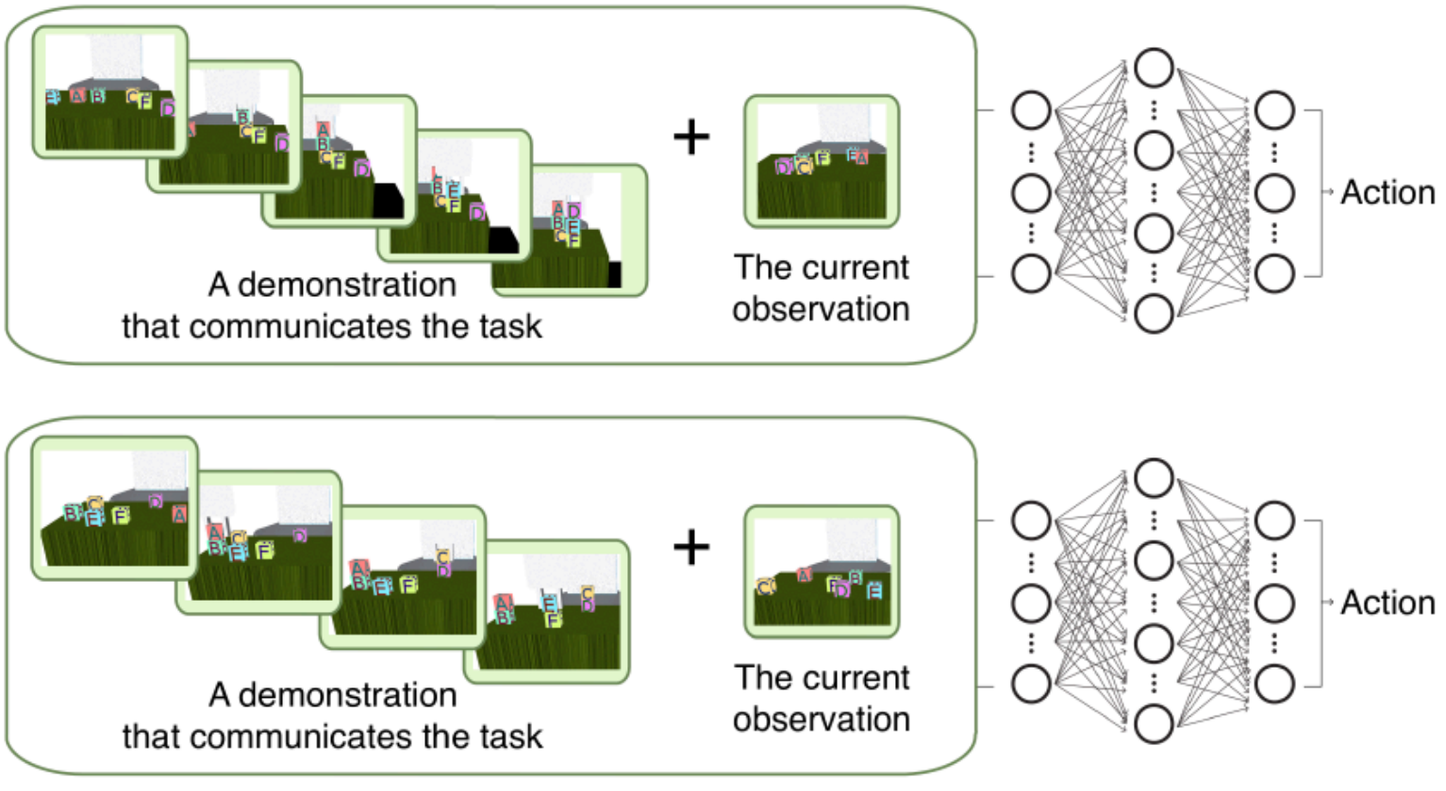
Yan Duan [1 2] **Marcin Andrychowicz** [1] **Bradly C. Stadie** [1 2] **Jonathan Ho** [1 2] **Jonas Schneider** [1] **Ilya Sutskever** [1]
Pieter Abbeel [1 2] **Wojciech Zaremba** [1]

Idea: Learn a policy network that, given a description of a new (related) task in the form of a single demonstration, and an initial observation, performs the task (outputs the right action sequence).

One-Shot Imitation Learning

Yan Duan [1 2] **Marcin Andrychowicz** [1] **Bradly C. Stadie** [1 2] **Jonathan Ho** [1 2] **Jonas Schneider** [1] **Ilya Sutskever** [1]
Pieter Abbeel [1 2] **Wojciech Zaremba** [1]

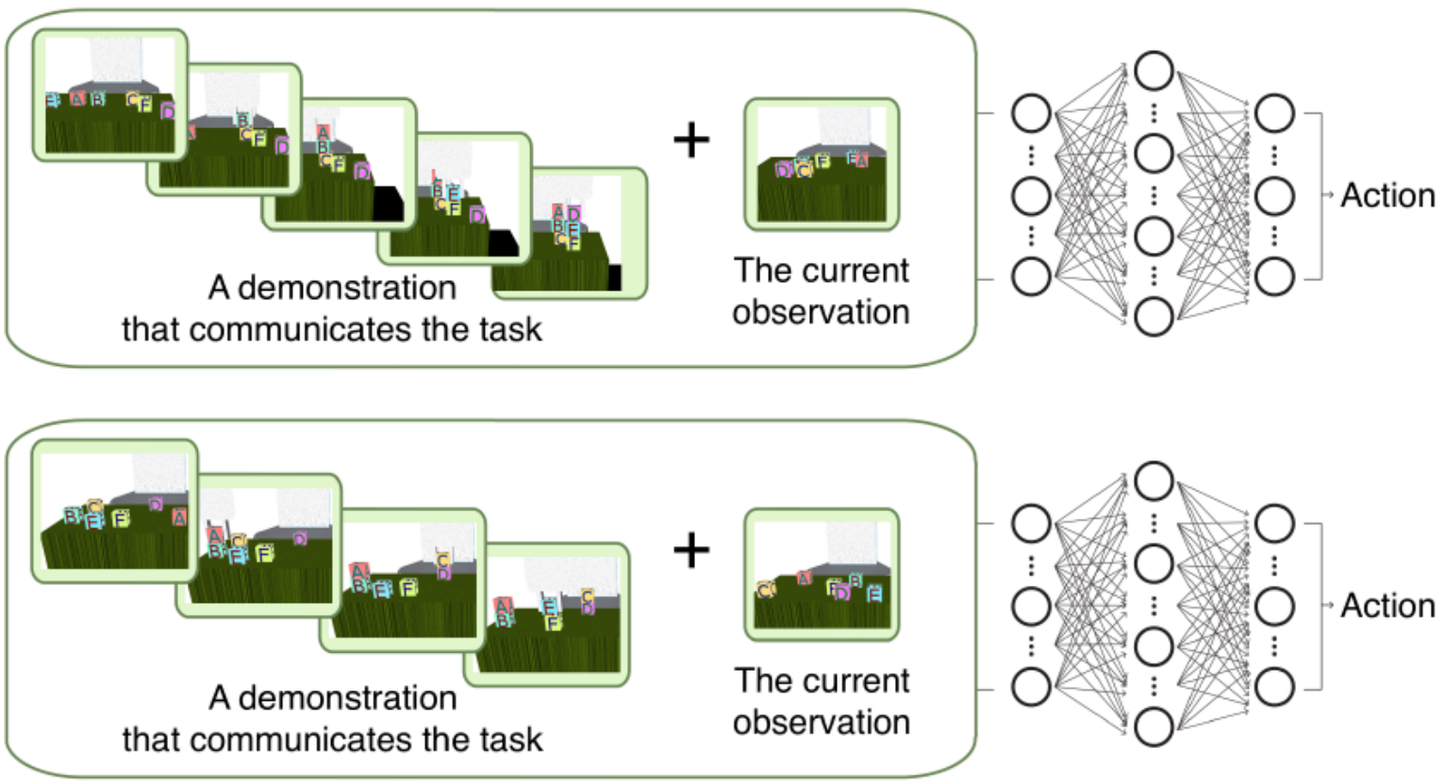
At training time, we are given pairs of demonstrations of related but not identical tasks. The policy network, conditioned on the first demonstration and the first observation of the second demonstration, learns to mimic the second demonstration. The policy net is trained using behavior cloning/ DAGGER.



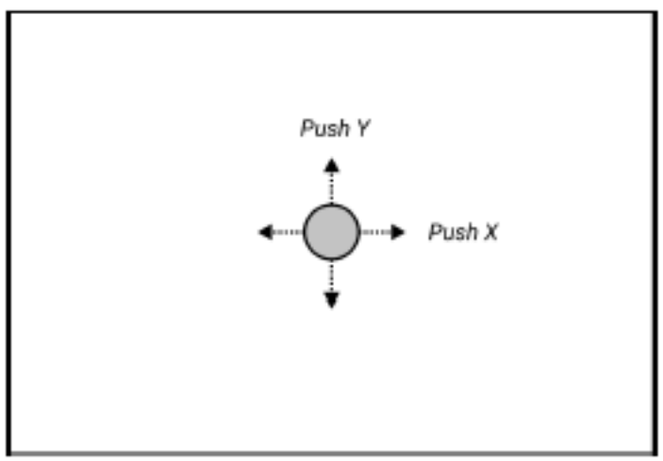
One-Shot Imitation Learning

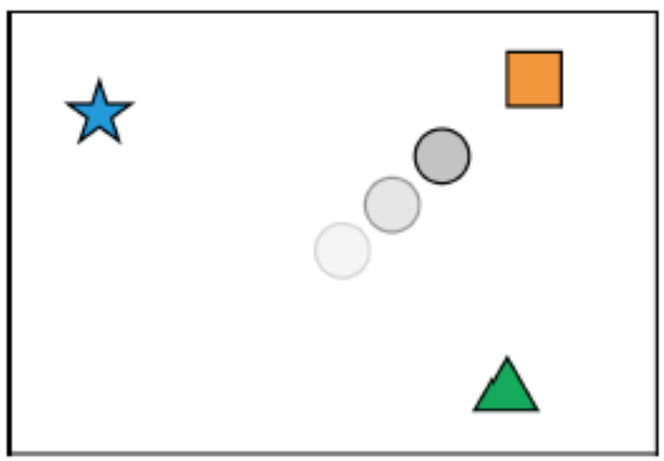
Yan Duan [1 2] **Marcin Andrychowicz** [1] **Bradly C. Stadie** [1 2] **Jonathan Ho** [1 2] **Jonas Schneider** [1] **Ilya Sutskever** [1]
Pieter Abbeel [1 2] **Wojciech Zaremba** [1]

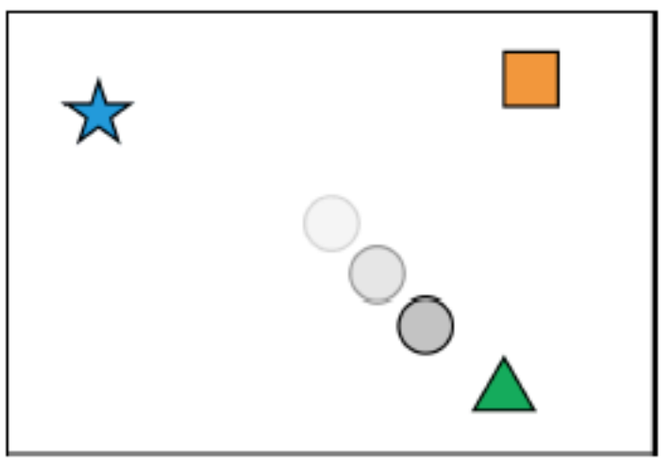
Alternatively, instead of providing a second demonstration we could have provided a reward function and train the policy network conditioned on the first demonstration and a new observation to carry out the desired task using RL (trail and error). But imitation is faster.



Reaching Task



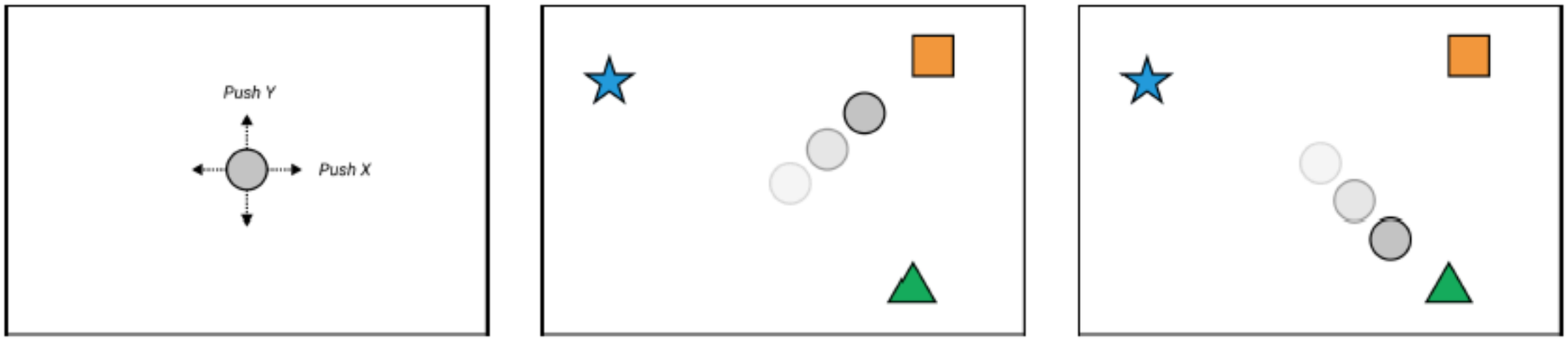




- The robot is a point mass controlled with 2-dimensional force.
- The family of tasks is to reach a target landmark.
- The identity of the landmark differs from task to task, and the model has to figure out which target to pursue based on the demonstration.

Appealing! We learn what is the essence of the demonstration using input output pairs: e.g., if i'm demonstrated how to hit the ball, I know that i should focus on the bouncing as opposed to the breathing of the expert, because this is gonna help me carry out the mimicking of the second demonstration.

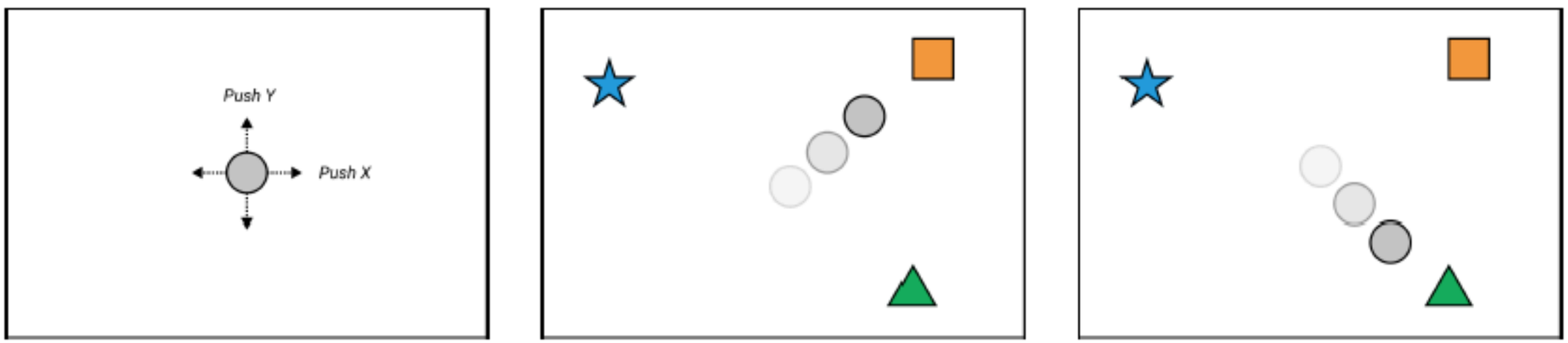
Reaching Task



- Observations: the agent's 2D location and the 2D locations of the landmarks
- Actions: 2D force applied to the agent
- The family of tasks is to reach a target landmark.
- The identity of the landmark differs from task to task, and the model has to figure out which target to pursue based on the demonstration.

Appealing! We learn what is the essence of the demonstration using input output pairs: e.g., if i'm demonstrated how to hit the ball, I know that i should focus on the bouncing as opposed to the breathing of the expert, because this is gonna help me carry out the mimicking of the second demonstration.

Reaching Task

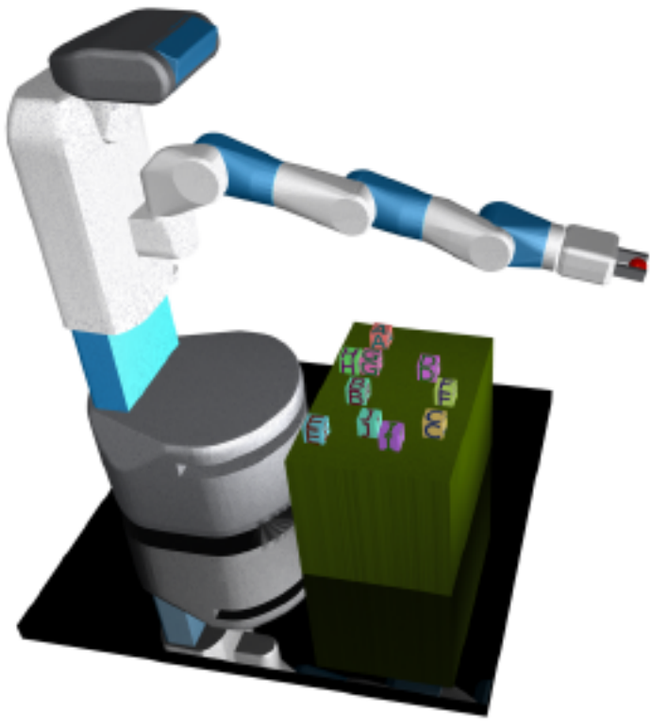


- The robot is a point mass controlled with 2-dimensional force.
- The family of tasks is to reach a target landmark.
- The identity of the landmark differs from task to task, and the model has to figure out which target to pursue based on the demonstration.

If you have only a single demonstration of the task, how do you know that the essence is to approach the green triangle as opposed to go towards the lower right corner of the board? or to avoid the blue asterisk?

You don’t know! But here you have multiple demo pairs to learn from!

Stacking tower task

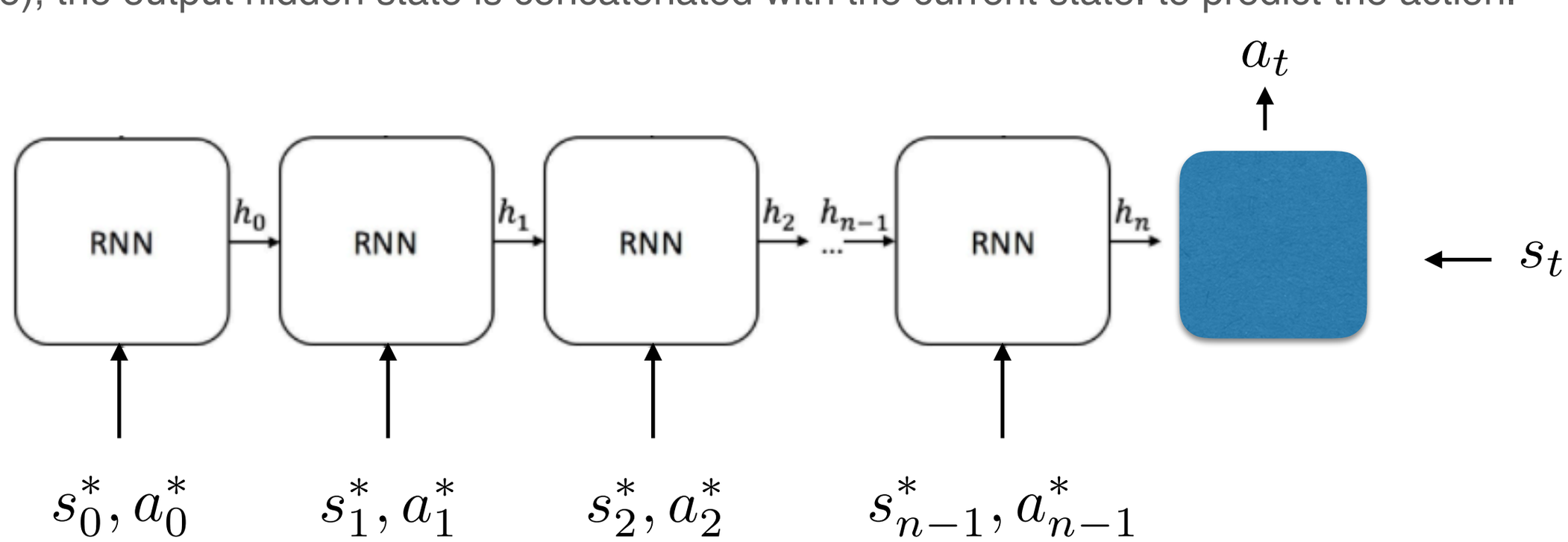


- Observations: (x,y,z) positions of the objects relative to the gripper, whether gripper is pen or closed
- Actions: 7 DOF fetch arm torques
- The family of tasks is to stack variable number of cuboids into tower configurations
- The number of cuboids differs from task to task! We need to generalize across the number of cubes!

Architecture of the policy network

- Input: a demonstration and an initial state x
- Output: corresponding action

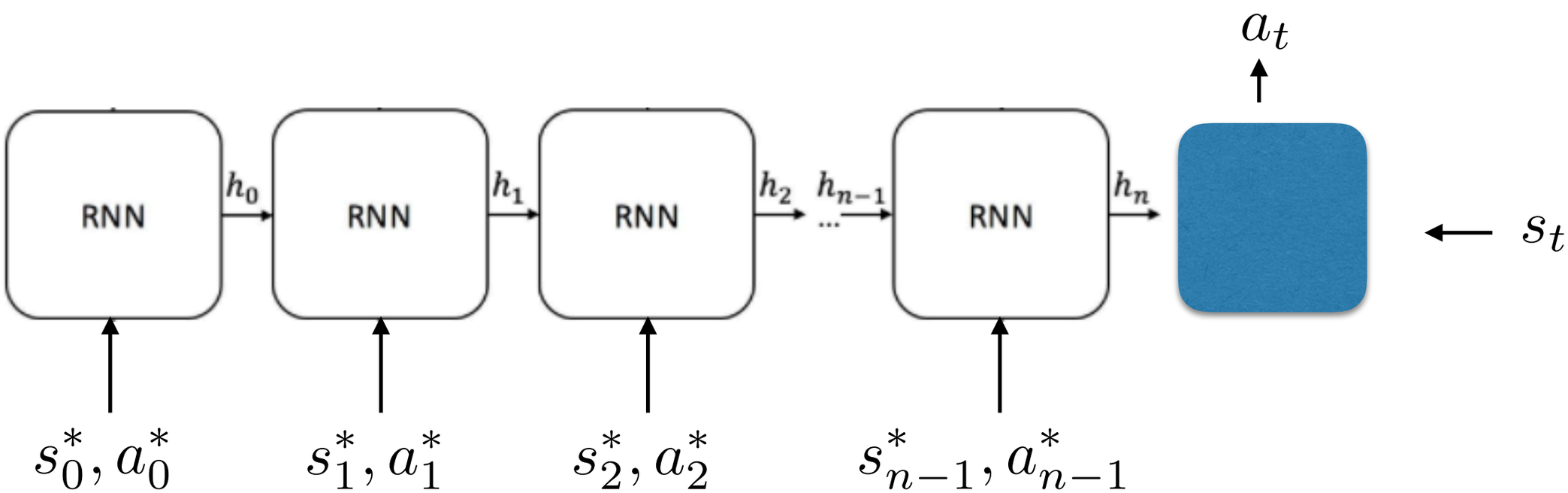
- Plain LSTM: it parses the demonstration trajectory of state/actions (one tilmestep at a time), the output hidden state is concatenated with the current state. to predict the action.



Architecture of the policy network

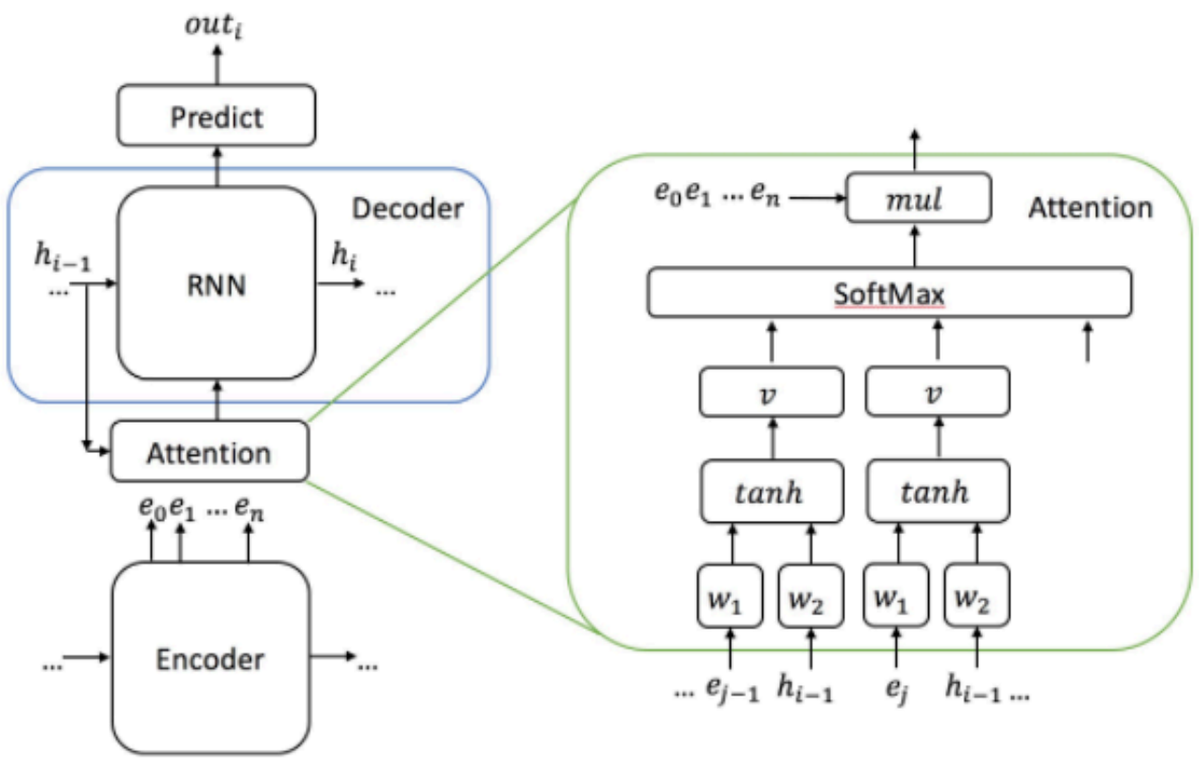
- Input: a demonstration and an initial state x
- Output: corresponding action

- Plain LSTM: it parses the demonstration trajectory of state/actions (one tilmestep at a time), the output hidden state is concatenated with the current state. to predict the action.



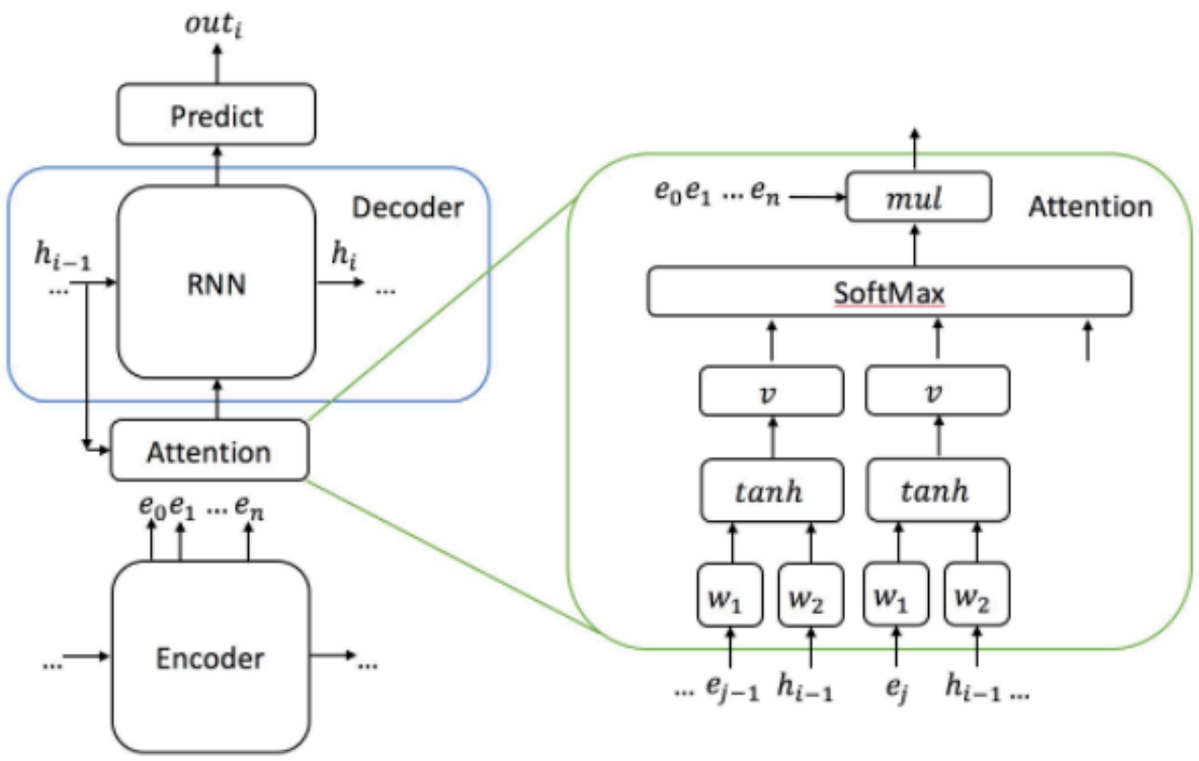
Architecture of the policy network

- Input: a demonstration and an initial state x
- Output: corresponding action
- LSTM with attention: at each time step, based on the hidden state, we can ``attend" in different parts of the demo trajectory (our memory). The attention forms a weight distribution over memory locations.



LSTM with Attention

- LSTM with attention: at each time step, based on the hidden state, we can ``attend" in different parts of the demo trajectory (our memory). The attention forms a weight distribution over memory locations.

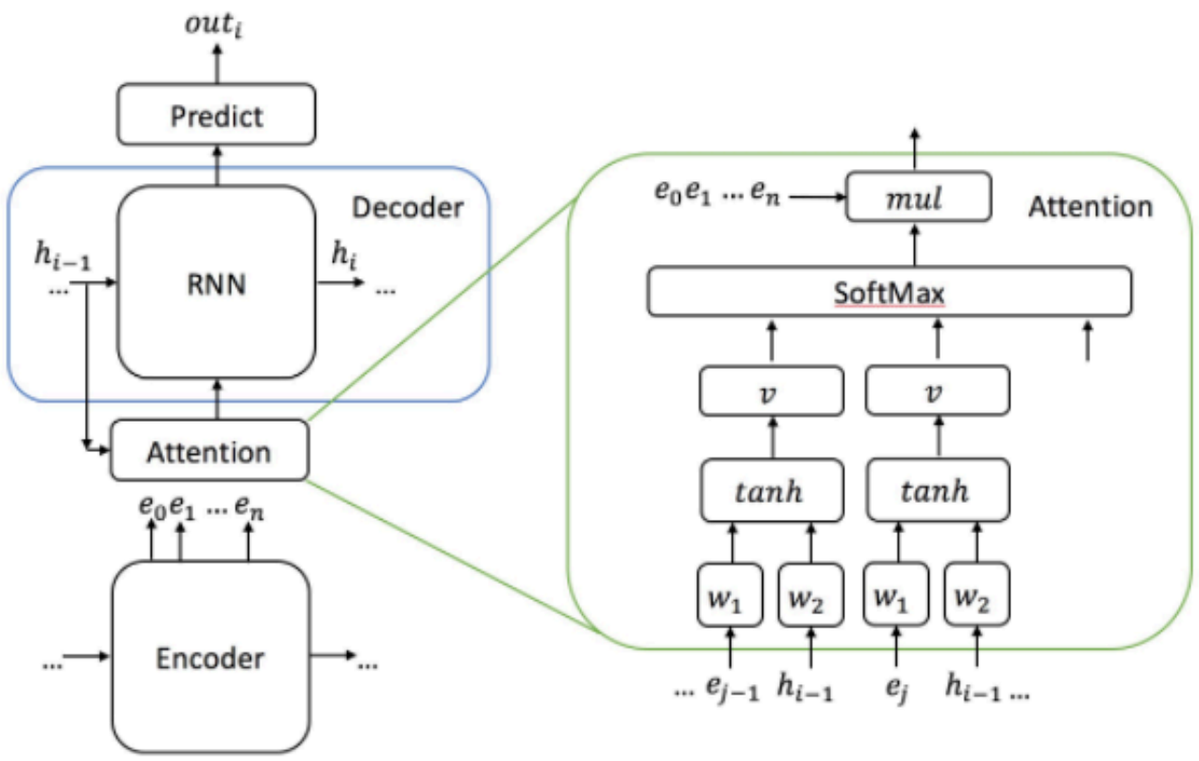


$$w_i \leftarrow v^T \tanh(q + c_i)$$

$$\text{output} \leftarrow \sum_i m_i \frac{\exp(w_i)}{\sum_j \exp(w_j)}$$

Neighborhood Attention-one per object

- LSTM with attention: at each time step, based on the hidden state, we compute a set of embeddings, one for each block present in the state. this operation allows each block to query other blocks in relation to itself (e.g. find the closest block), and extract the queried information



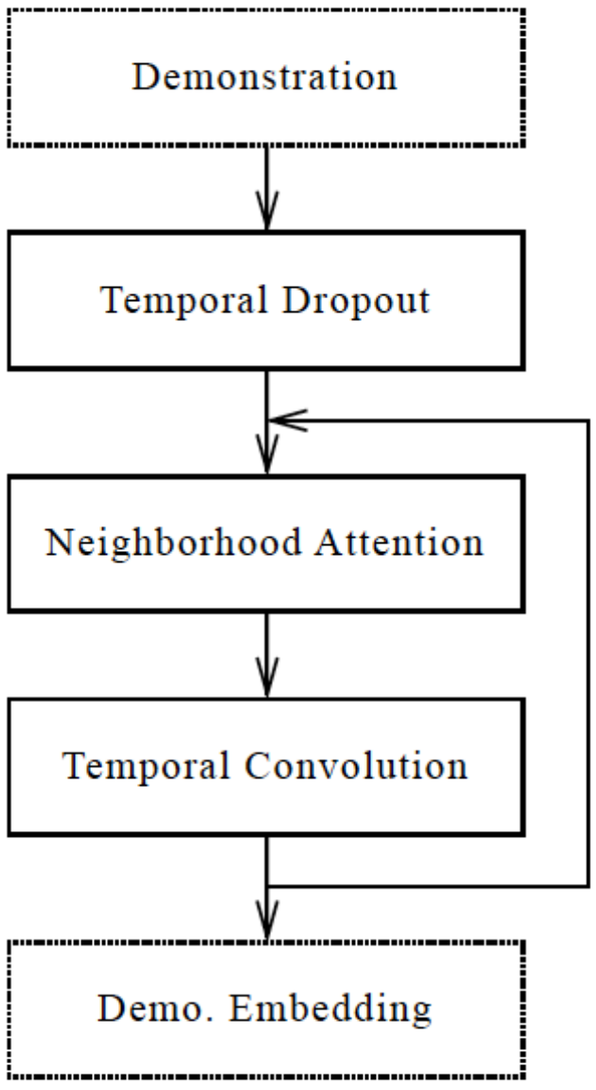
$$w_i \leftarrow v^T \tanh(q + c_i)$$

$$\text{output} \leftarrow \sum_i m_i \frac{\exp(w_i)}{\sum_j \exp(w_j)}$$

$$\begin{aligned}\text{result}_i \leftarrow \text{SoftAttention}(&\\ &\text{query} : q_i,\\ &\text{context} : \{c_j\}_{j=1}^{B},\\ &\text{memory} : \{\text{concat}((x_j, y_j, z_j), h_j^{in}))\}_{j=1}^{B}\\ )&\end{aligned}$$

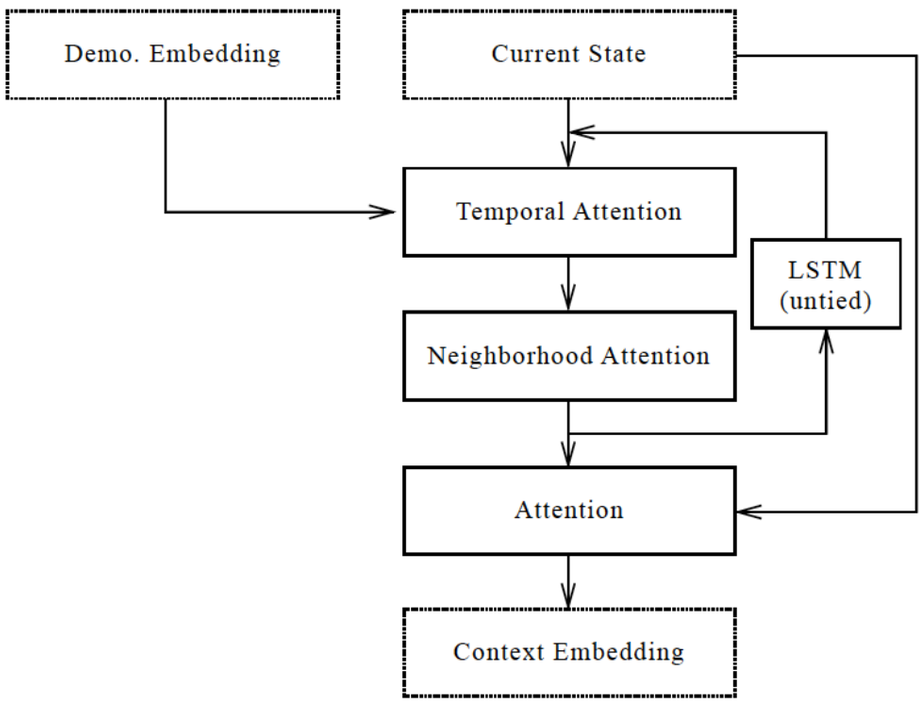
$$\text{output}_i \leftarrow \text{Linear}(\text{concat}(h_i^{in}, \text{result}_i, (x_i, y_i, z_i), s_{\text{robot}}))$$

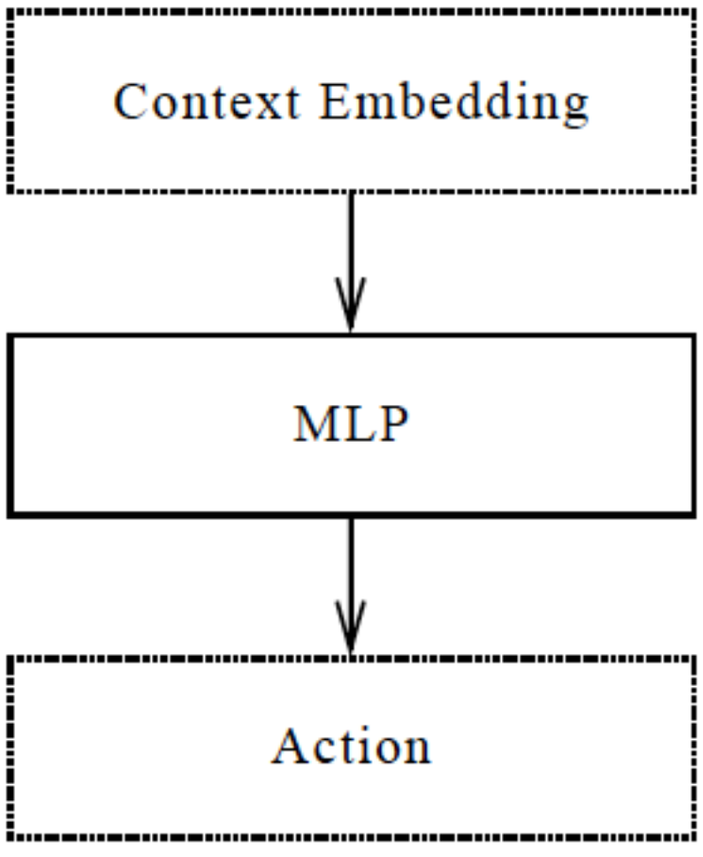
Demonstration Memory



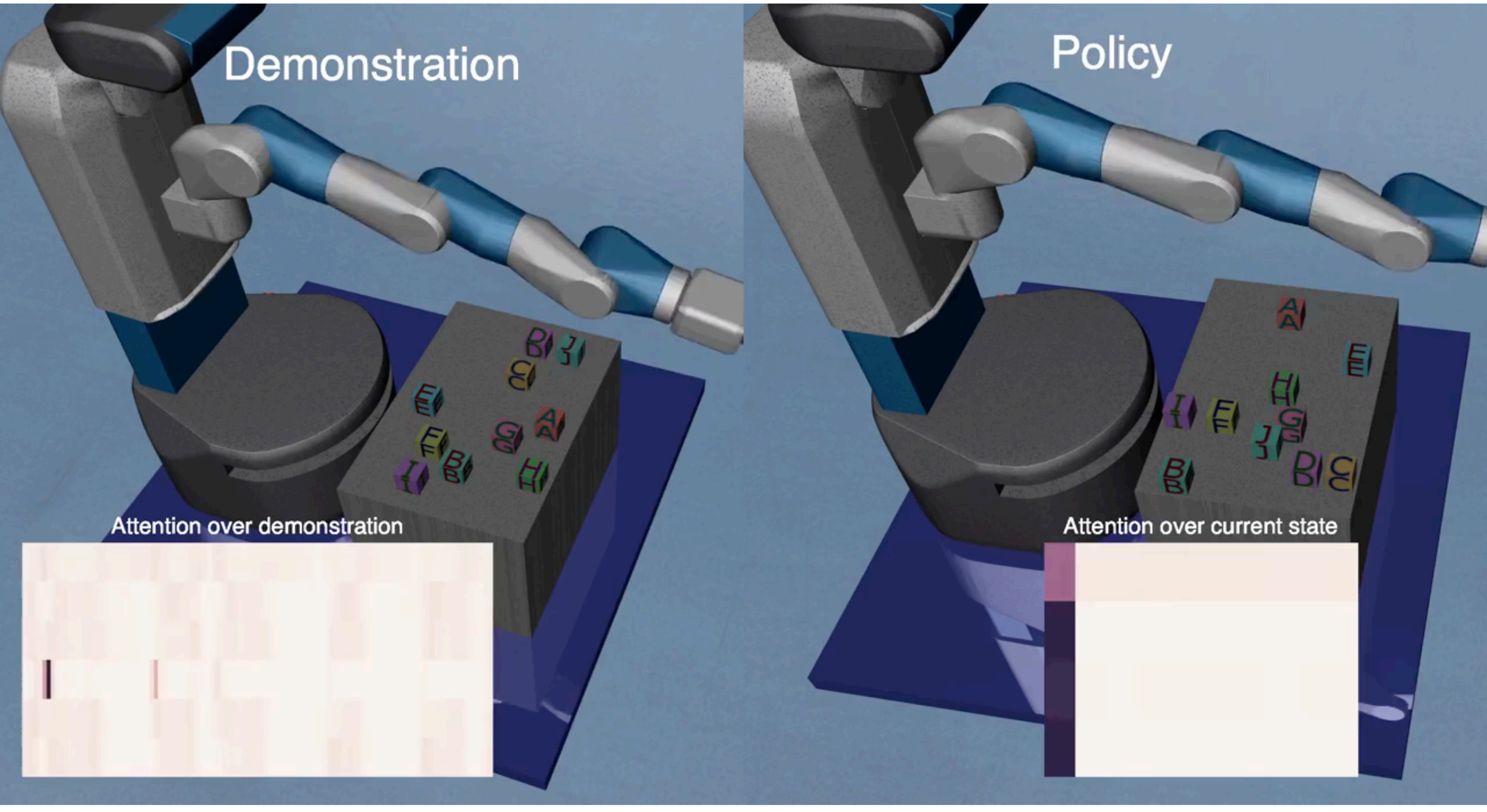
The embedding grows linearly with the demo length and number of blocks.

Context Embedding and policy net





One shot imitation learning



Learning to learn: This lecture

- Learning to optimize: learn parameter update rules
- One shot imitation learning
- Learn a policy for learning a new task fast (within a specified window of experience)
- Learning parameters so that a specific number of update steps under a loss of a new tasks yields good weights
- One shot learning using compositional neural network architectures

RL^2: Fast Reinforcement Learning via Slow Reinforcement Learning

Yan Duan[†‡], **John Schulman**[†‡], **Xi Chen**[†‡], **Peter L. Bartlett**[†], **Ilya Sutskever**[‡], **Pieter Abbeel**[†‡]
[†] UC Berkeley, Department of Electrical Engineering and Computer Science
[‡] OpenAI
{rocky,joschu,peter}@openai.com, peter@berkeley.edu, {ilyasu,pieter}@openai.com

RL^2: Fast Reinforcement Learning via Slow Reinforcement Learning

Yan Duan[†‡], **John Schulman**[†‡], **Xi Chen**[†‡], **Peter L. Bartlett**[†], **Ilya Sutskever**[‡], **Pieter Abbeel**[†‡]
[†] UC Berkeley, Department of Electrical Engineering and Computer Science
[‡] OpenAI
{rocky,joschu,peter}@openai.com, peter@berkeley.edu, {ilyasu,pieter}@openai.com

Reinforcement learning: master a task by interaction with the environment.

Big question: exploration-exploitation.

The most principled way would be to maintain a distribution over actions and sample actions according to that (in the beginning we are uncertain, the distribution is close to uniform and we explore a lot, and as time goes by we explore more and we converge).

Often, we pick exploration policies arbitrarily, e.g., ϵ-greedy. This results in large number of samples.

RL^2: Fast Reinforcement Learning via Slow Reinforcement Learning

Yan Duan[†‡], **John Schulman**[†‡], **Xi Chen**[†‡], **Peter L. Bartlett**[†], **Ilya Sutskever**[‡], **Pieter Abbeel**[†‡]
[†] UC Berkeley, Department of Electrical Engineering and Computer Science
[‡] OpenAI
{rocky,joschu,peter}@openai.com, peter@berkeley.edu, {ilyasu,pieter}@openai.com

Reinforcement learning: master a task by interaction with the environment.

Big question: exploration-exploitation.

Idea: What if we learn the exploration strategy itself? What if our learning objective is not doing well in a specific task but doing fast RL of any task: mastering a task with few interactions with the environment.

RL VS Learning RL

RL

Each time an episode is unrolled, reward is observed, policy is updated

Objective: maximize the expected total discounted reward accumulated during a single episode

No memory across episodes other than the policy's updated parameters

RL VS Learning RL

RL

Each time an episode is unrolled, reward is observed, policy is updated

Objective: maximize the expected total discounted reward accumulated during a single episode

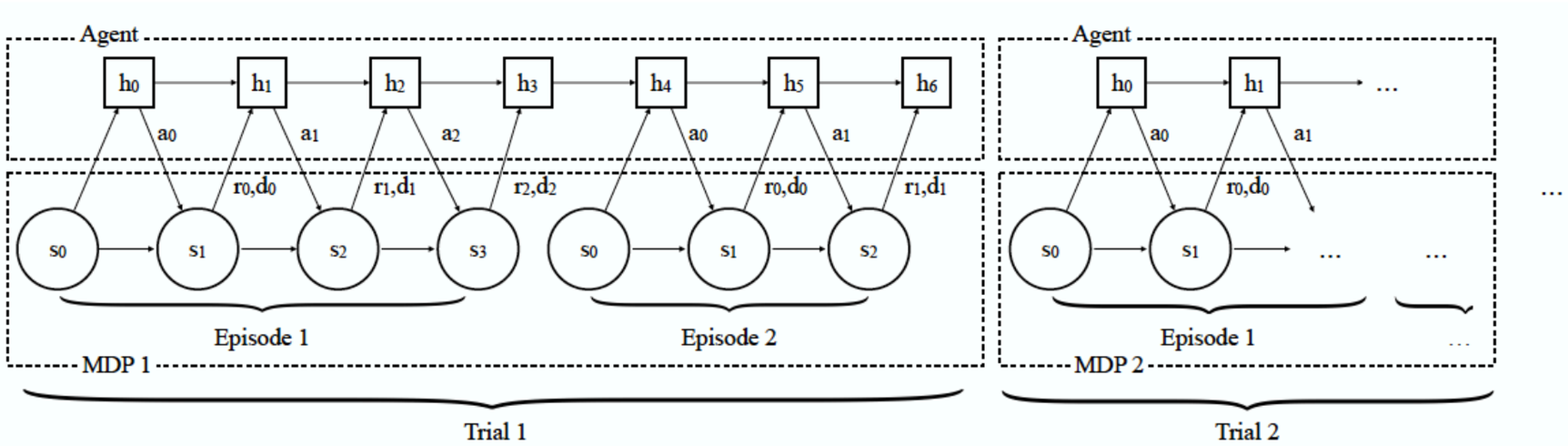
No memory across episodes other than the policy's updated parameters

Learning RL

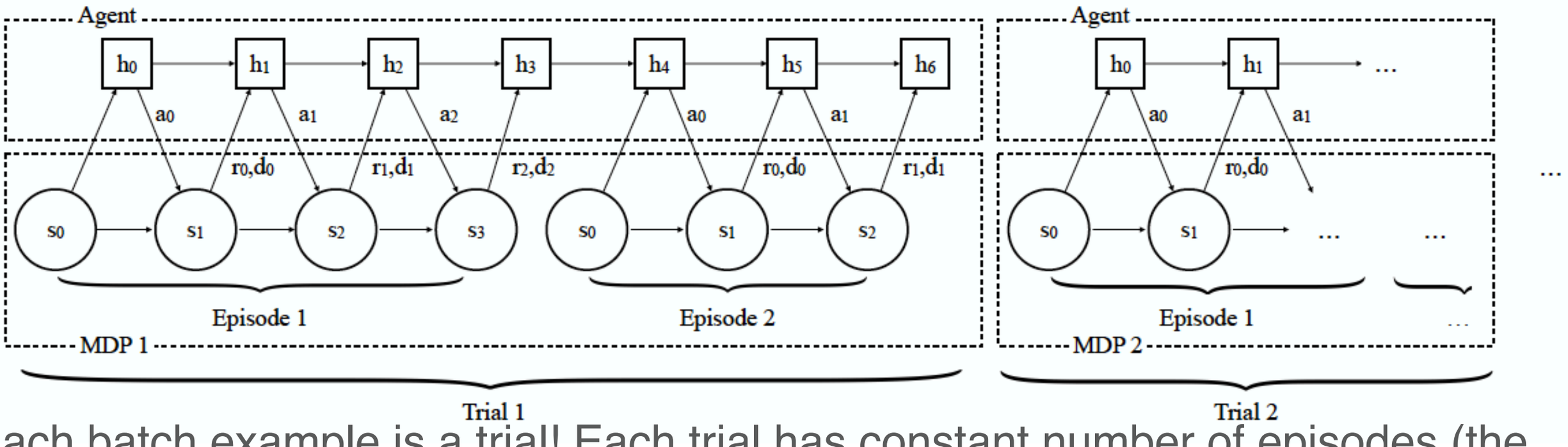
Each time, a series of episodes is experienced, reward is observed, policy is updated. Policy is updated at the end of the interaction budget (a specific number of episodes) given to me to master the task!

Objective: maximize the expected total discounted reward accumulated during a single trial (as opposed to episode)

Memory is preserved across episodes, e.g., I 'd better know what actions i tried earlier, not to try the same things



RL VS Learning RL



Each batch example is a trial! Each trial has constant number of episodes (the experience budget).

For each trial, a separate MDP is drawn.

Policy: an RNN that takes (state,action,reward,termination flag) as input at every time step.

TRPO as the meta-policy optimizer

Multi-armed bandits

Multi-armed bandit problems are a subset of MDPs where the agent's environment is stateless.

There are k arms (actions), and at every time step, the agent pulls one of the arms, and receives a reward drawn from an unknown distribution, e.g., a Bernoulli distribution with parameter p_i.

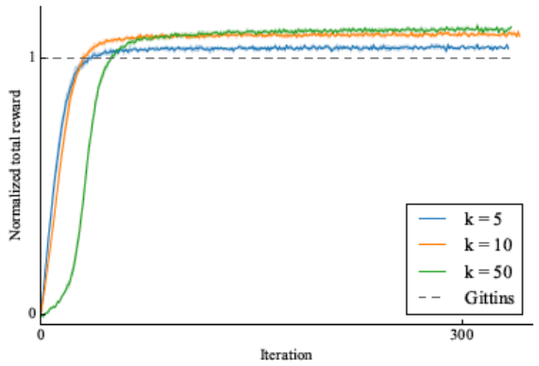
The goal is to maximize the total reward obtained over a fixed number of time steps.

The key challenge is balancing exploration and exploitation— "exploring" each arm enough times to estimate its distribution (p_i), but eventually switching over to "exploitation" of the best arm.

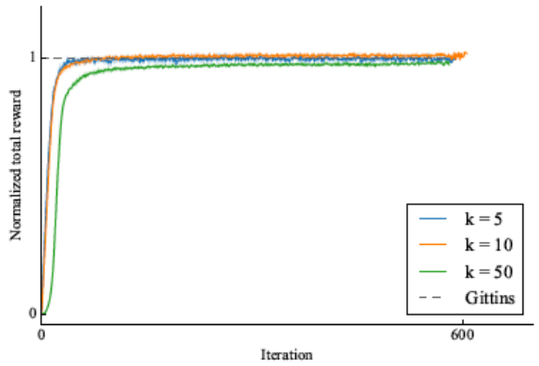
RL^2 aims at learning to solve bandit problems (learning such exploration) as opposed to doing well in a particular bandit setup!

Multi-armed bandits: learning to explore VS theoretically optimal exploration algorithms

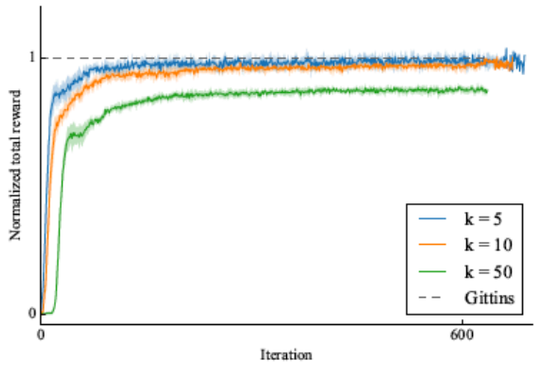
Setup	Random	Gittins	TS	OTS	UCB1	ϵ-Greedy	Greedy	RL^2
$n = 10, k = 5$	5.0	**6.6**	5.7	6.5	**6.7**	**6.6**	**6.6**	**6.7**
$n = 10, k = 10$	5.0	**6.6**	5.5	6.2	**6.7**	**6.6**	**6.6**	**6.7**
$n = 10, k = 50$	5.1	6.5	5.2	5.5	**6.6**	6.5	6.5	**6.8**
$n = 100, k = 5$	49.9	**78.3**	74.7	**77.9**	**78.0**	75.4	74.8	**78.7**
$n = 100, k = 10$	49.9	**82.8**	76.7	81.4	82.4	77.4	77.1	**83.5**
$n = 100, k = 50$	49.8	**85.2**	64.5	67.7	84.3	78.3	78.0	**84.9**
$n = 500, k = 5$	249.8	**405.8**	**402.0**	**406.7**	**405.8**	388.2	380.6	**401.6**
$n = 500, k = 10$	249.0	**437.8**	429.5	**438.9**	**437.1**	408.0	395.0	432.5
$n = 500, k = 50$	249.6	**463.7**	427.2	437.6	457.6	413.6	402.8	438.9



(a) $n = 10$



(b) $n = 100$



(c) $n = 500$

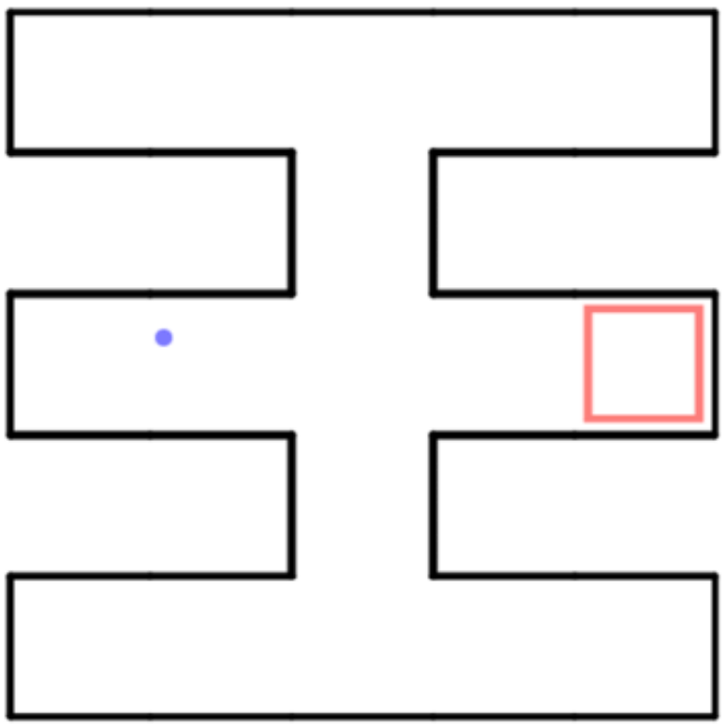
k: number of bandits

n: number of episodes

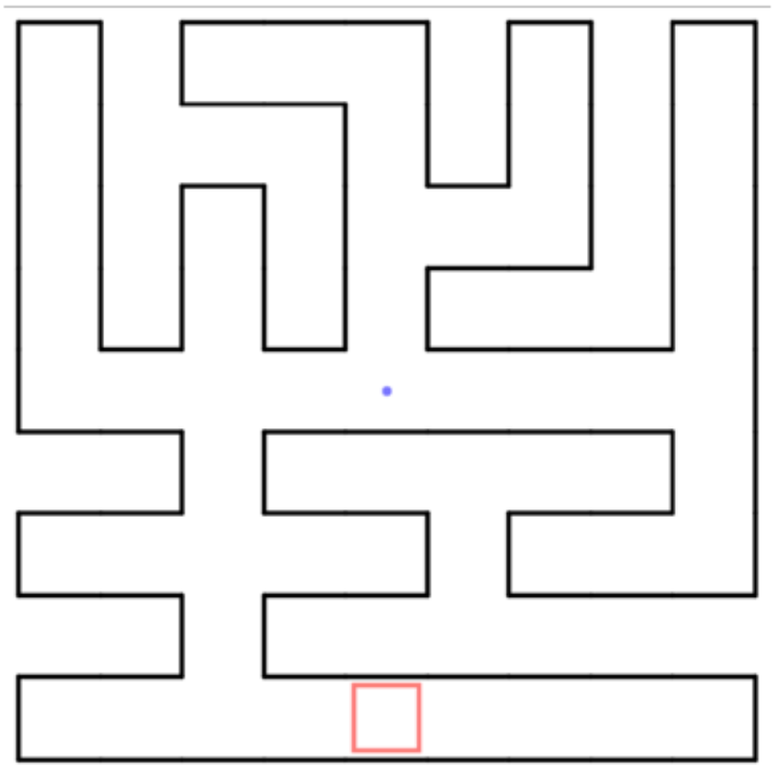
RL^2 for Visual navigation

Learning to explore

(a) Sample observation

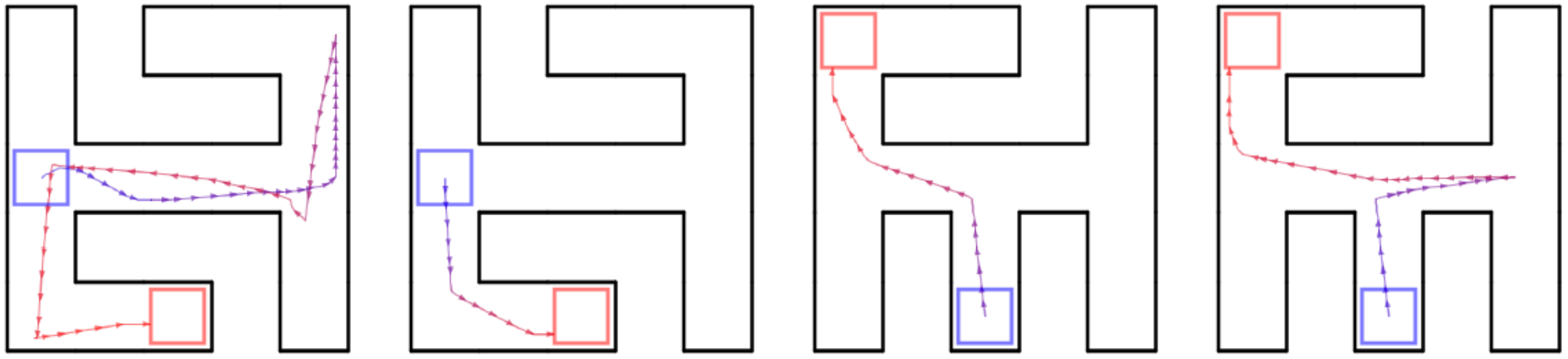


(b) Layout of the 5×5 maze in (a)



(c) Layout of a 9×9 maze

RL^2 for Visual navigation



(a) Good behavior, 1st episode

(b) Good behavior, 2nd episode

(c) Bad behavior, 1st episode

(d) Bad behavior, 2nd episode

Learning to learn: This lecture

- Learning to optimize: learn parameter update rules
- Learning parameters so that a specific number of update steps under a loss of a new tasks yields good weights
- Learn a policy for learning a new task fast (within a specified window of experience)

Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks

Chelsea Finn [1] **Pieter Abbeel** [1 2] **Sergey Levine** [1]

How can I compute parameters theta, so that, after a single gradient parameter update using a small set of K labelled examples from a new task, I will be able to master it?

Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks

Chelsea Finn [1] **Pieter Abbeel** [1 2] **Sergey Levine** [1]

How can I compute parameters theta, so that, after a single gradient parameter update using a small set of K labelled examples from a new task, I will be able to master it?

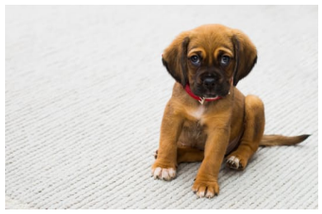
E.g., the new task could be, build a great ostrich detector after seeing few examples (e.g. 3) of ostrich images.

Few shot classification example

training

x Y

1

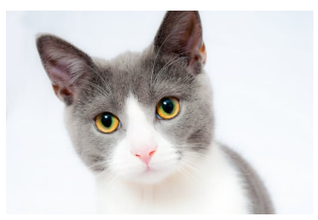


1

1

0

0

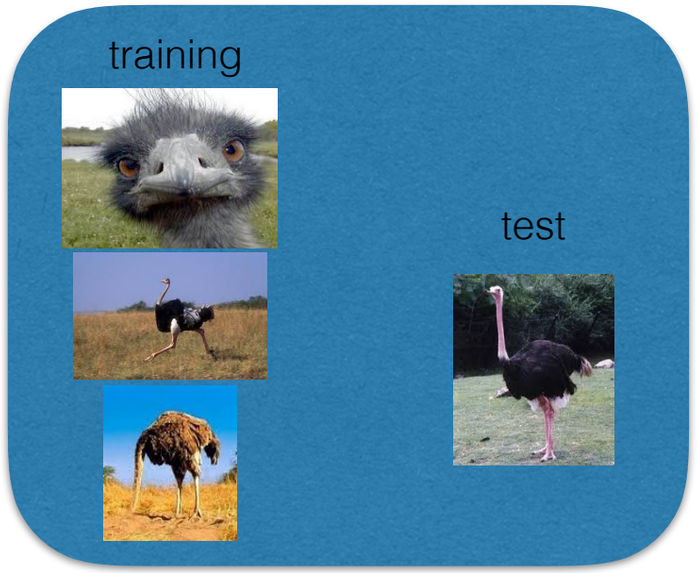


0

test

learn an ostrich detector using few training examples

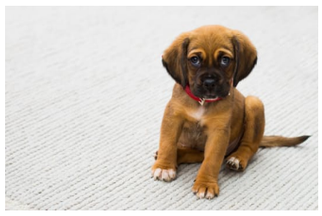
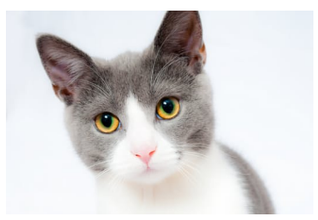
Model



Current solution is: train a general image classifier using Imagenet annotations, and fine-tune that classifier using the few image at hand. Problems?

Few shot classification example

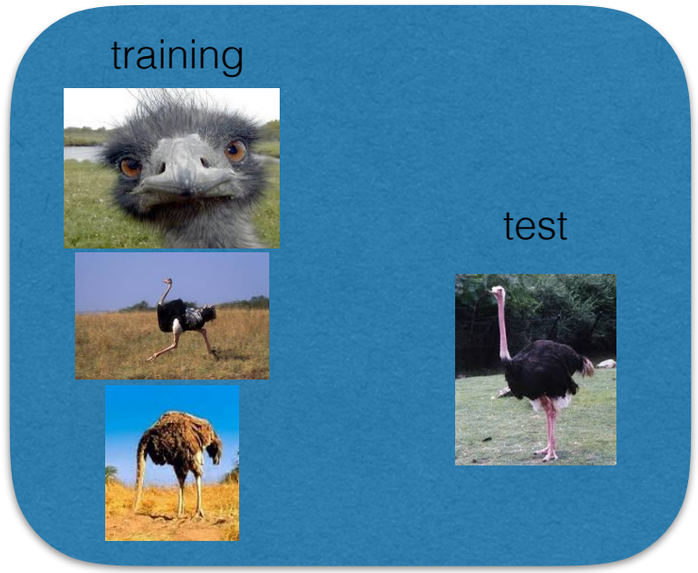
training

x	Y
	1
	1
	1
	0
	0
	0

test

learn an ostrich detector using few training examples

Model



Current solution is: train a general image classifier using Imagenet annotations, and fine-tune that classifier using the few image at hand. Problems?

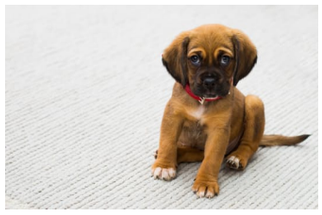
How much to fine-tune (# of iterations), with what l.r., how many layers etc., to guarantee generalization?

Few shot classification example

training

x Y

1

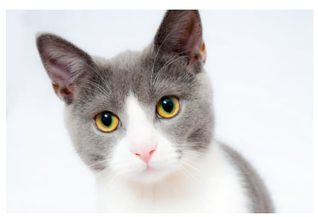


1

1

0

0

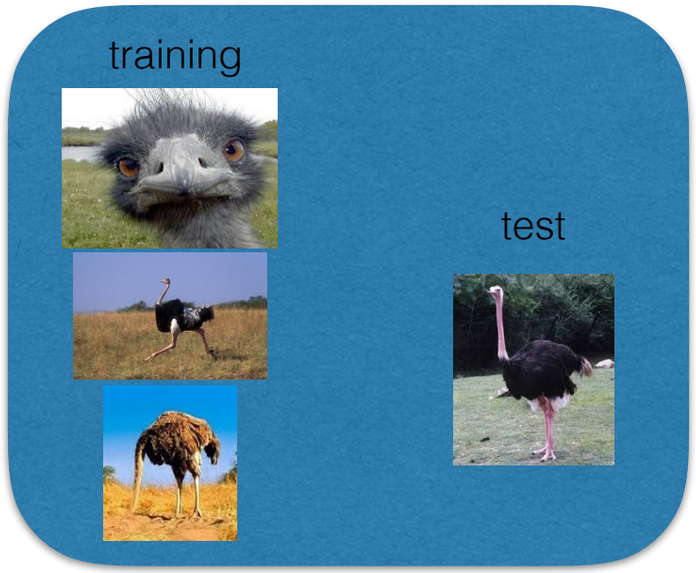


0

test

learn an ostrich detector using few training examples

Model



How much to fine-tune (# of iterations), with what l.r., how many layers etc., to guarantee generalization?

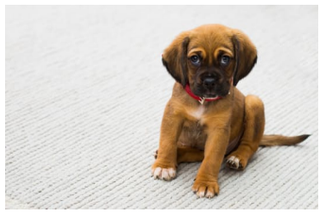
Problem is i do not train on what i will test on: capturing well new, unseen images of ostrich, as opposed to classifying correctly the few ones in my training set..

Few shot classification example

training

x Y

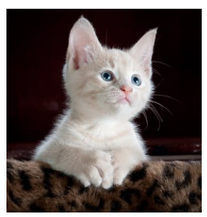
1



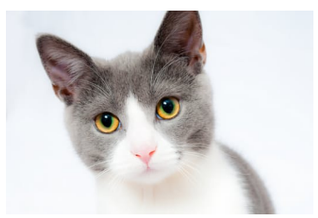
1

1

0



0

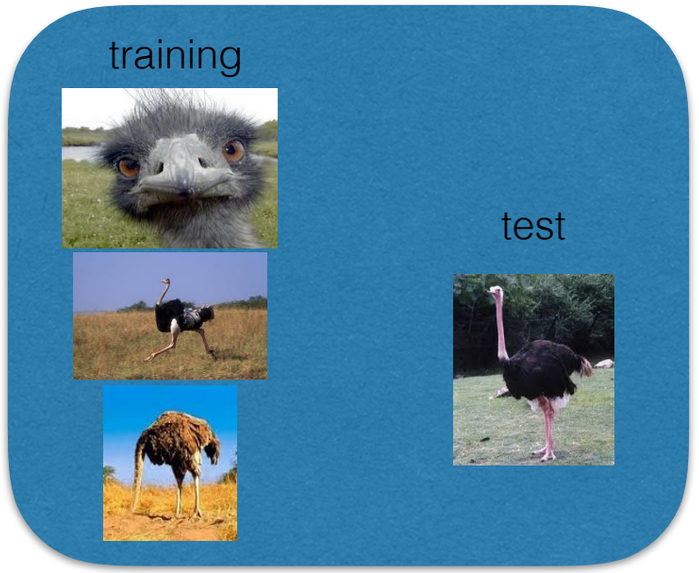


0

learn an ostrich detector using few training examples

test

Model

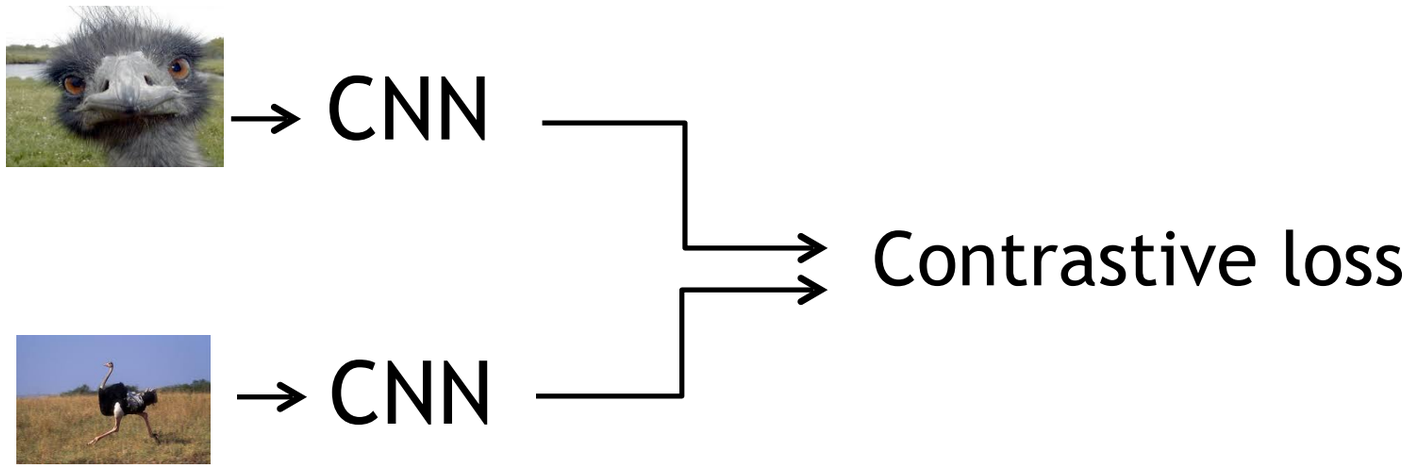


Other solutions to this problem: metric learning, matching networks Vinyals et al. 2016, learning to remember rare events Kaiser et. al 2017

Metric learning

Images are embedded so that same label images are closely and different label images are far apart.

You need to fine-tune the CNN to accommodate for the newly arrived ostrich images



$$min. \quad max(margin - \|\phi_W\,(\,\,) - \phi_W\,(\,\,)\|, 0)$$

$$min. \quad \|\phi_W\,(\,\,) - \phi_W\,(\,\,)\|$$

Metric learning

Images are embedded so that same label images are closely and different label images are far apart.

You need to fine-tune the CNN to accommodate for the newly arrived ostrich images

Again: how many iterations to do with th new ostrich labels? The train and test conditions (metric learning and few sho classification) are not the same!

$$min. \quad max(margin - \|\phi_W (\quad) - \phi_W (\quad) \|, 0)$$

$$min. \quad \|\phi_W (\quad) - \phi_W (\quad) \|$$

Metric learning

Images are embedded so that same label images are closely and different label images are far apart.

You need to fine-tune the CNN to accommodate for the newly arrived ostrich images

Again: how many iterations to do with th new ostrich labels? The train and test conditions (metric learning and few sho classification) are not the same!

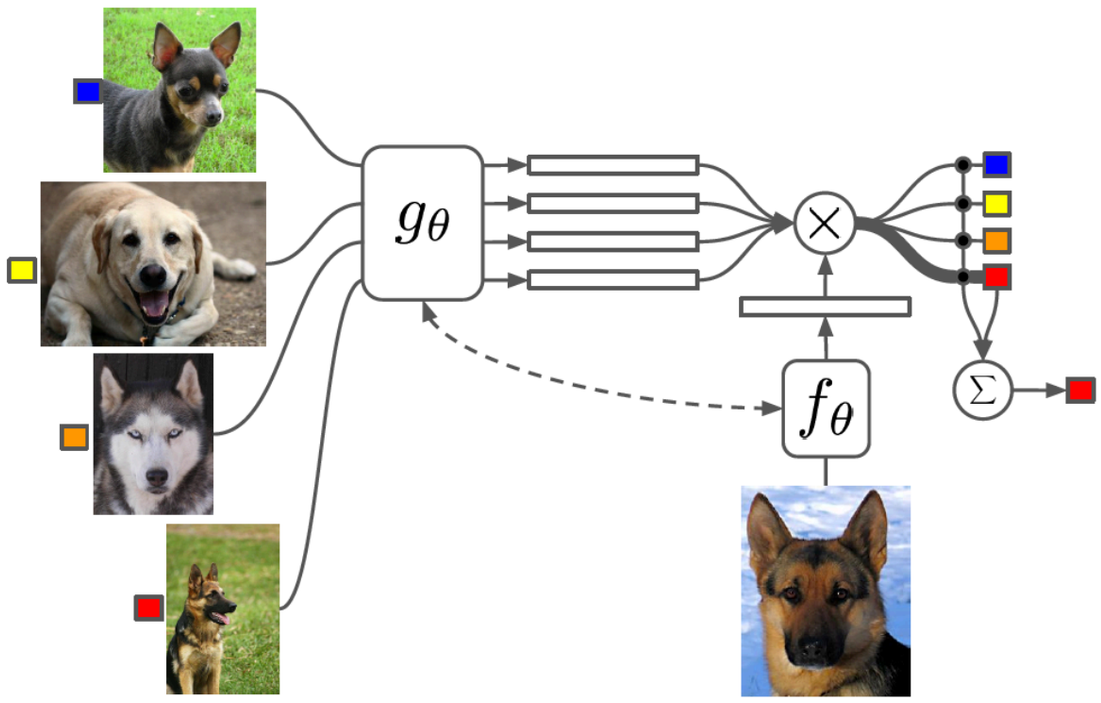
$$min. \quad max(margin - \|\phi_W(\ ) - \phi_W(\ )\|, 0)$$

$$min. \quad \|\phi_W(\ ) - \phi_W(\ )\|$$

Q: when do i know what i trained on what i'm testing on?

Matching Networks for one shot learning

Let's fix this!



$$\hat{y} = \sum_{i=1}^{k} a(\hat{x}, x_i) y_i$$

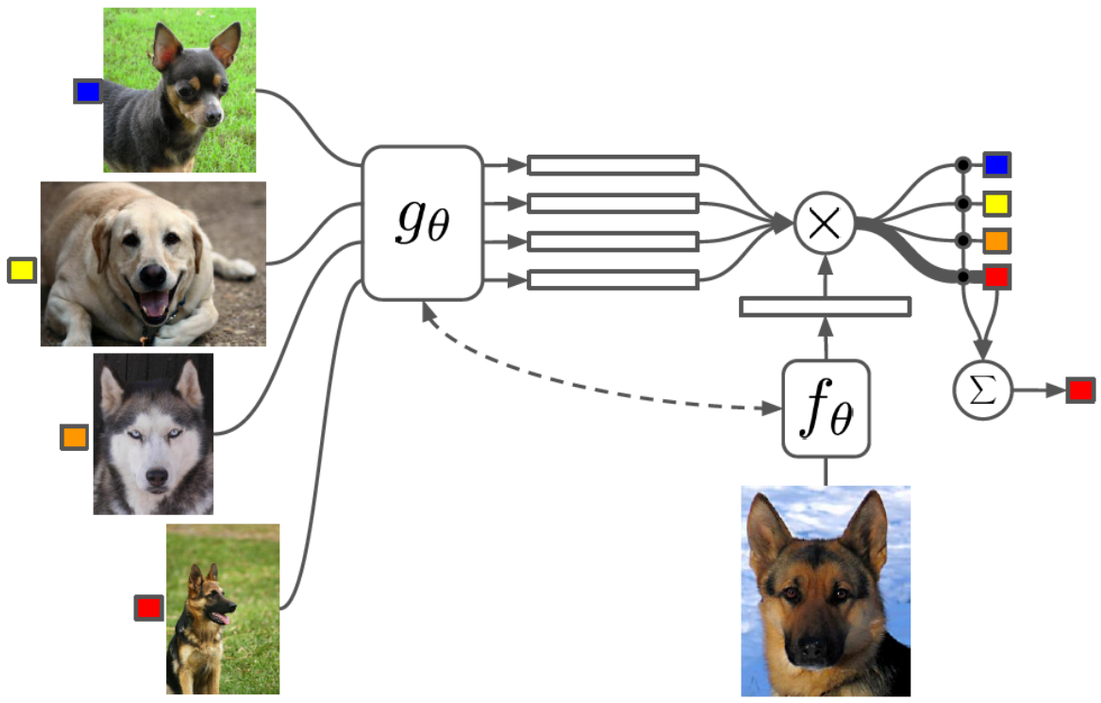
$$a(\hat{x}, x_i) = e^{c(f(\hat{x}), g(x_i))} / \sum_{j=1}^{k} e^{c(f(\hat{x}), g(x_j))}$$

If i also want the embedding of the query image to depend on my image set:

$$f(\hat{x}, S) = \text{attLSTM}(f'(\hat{x}), g(S), K)$$

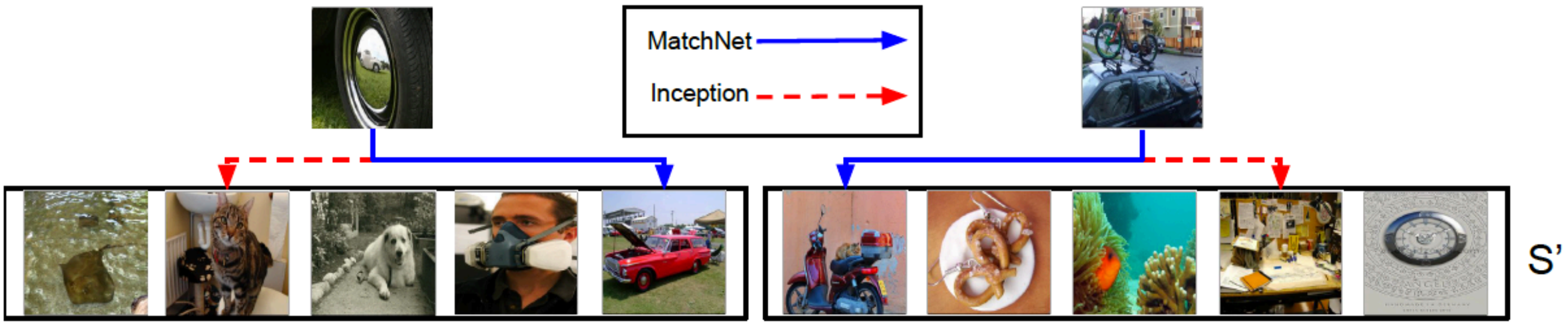
Matching Networks for one shot learning

Let's fix this!



$$\hat{y} = \sum_{i=1}^{k} a(\hat{x}, x_i) y_i$$

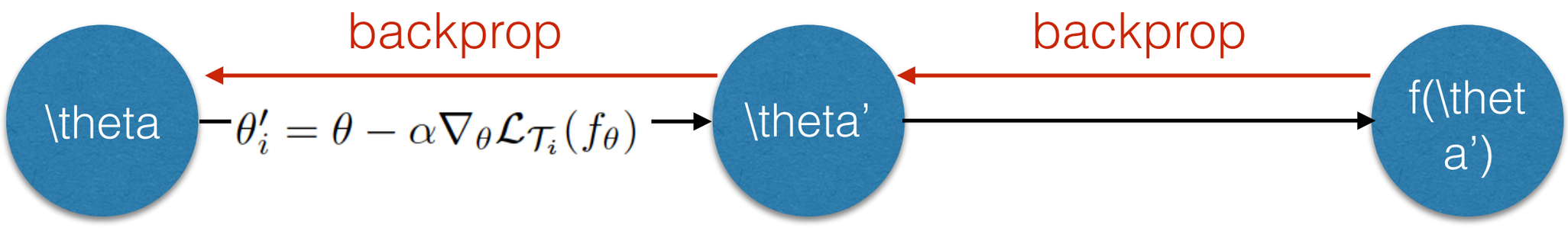
$$a(\hat{x}, x_i) = e^{c(f(\hat{x}), g(x_i))} / \sum_{j=1}^{k} e^{c(f(\hat{x}), g(x_j))}$$



Model agnostic meta-learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ with respect to K examples
6: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
7: **end for**
8: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$
9: **end while**

$$\min_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'}) = \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)})$$



Model agnostic meta-learning

Algorithm 2 MAML for Few-Shot Supervised Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Sample K datapoints $\mathcal{D} = \{\mathbf{x}^{(j)}, \mathbf{y}^{(j)}\}$ from $\mathcal{T}_i$
6: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ using $\mathcal{D}$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation (1) or (2)
7: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
8: Sample datapoints $\mathcal{D}_i' = \{\mathbf{x}^{(j)}, \mathbf{y}^{(j)}\}$ from $\mathcal{T}_i$ for the meta-update
9: **end for**
10: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$ using each $\mathcal{D}_i'$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 1 or 2
11: **end while**

Algorithm 3 MAML for Reinforcement Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Sample K trajectories $\mathcal{D} = \{(\mathbf{x}_1, \mathbf{a}_1, ...\mathbf{x}_H)\}$ using f_θ in $\mathcal{T}_i$
6: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ using $\mathcal{D}$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 3
7: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
8: Sample trajectories $\mathcal{D}_i' = \{(\mathbf{x}_1, \mathbf{a}_1, ...\mathbf{x}_H)\}$ using $f_{\theta_i'}$ in $\mathcal{T}_i$
9: **end for**
10: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$ using each $\mathcal{D}_i'$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 3
11: **end while**

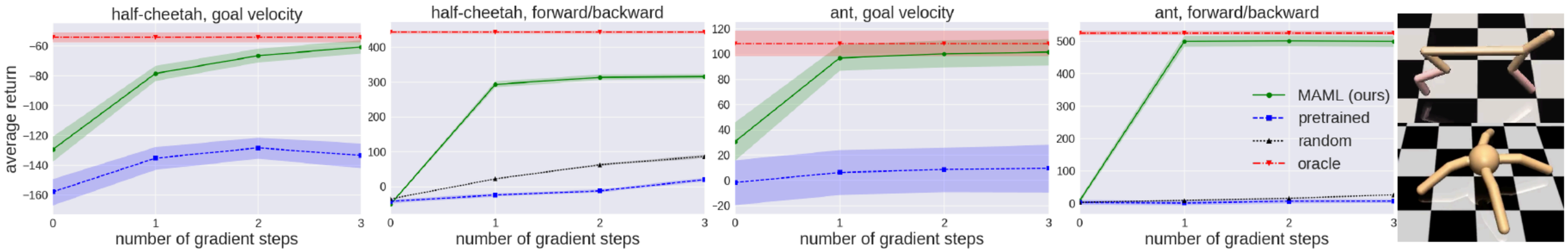
Few shot classification

Omniglot dataset: 20 instances from 1623 characters from 50 different alphabets

N-way classification: provide K different instances of N unseen character classes, evaluate the model's ability to classify new instances within those classes

Siamese nets (Koch, 2015)	97.3%	98.4%	88.2%	97.0%
matching nets (Vinyals et al., 2016)	98.1%	98.9%	93.8%	98.5%
neural statistician (Edwards & Storkey, 2017)	98.1%	99.5%	93.2%	98.1%
memory mod. (Kaiser et al., 2017)	98.4%	99.6%	**95.0%**	98.6%
MAML (ours)	$\mathbf{98.7 \pm 0.8\%}$	$\mathbf{99.9 \pm 0.3\%}$	$93.1 \pm 1.5\%$	$\mathbf{98.8 \pm 0.6\%}$

2D navigation



The policy was trained with MAML to maximize performance after 1 policy gradient update using 20 trajectories.

https://sites.google.com/view/maml

Summary so far

- Casting acquiring a new skill as a learning problem itself.
- Formulations for learning the update step, learning a policy for exploration, learning easily modifiable neural net weights
- Yet, learning to learn is about compositionality: composing old skills to form new ones

Learning to learn: This lecture

- Learning to optimize: learn parameter update rules
- One shot imitation learning
- Learn a policy for learning a new task fast (within a specified window of experience)
- Learning parameters so that a specific number of update steps under a loss of a new tasks yields good weights
- One shot learning using compositional neural network architectures

Compositional detectors

Instead of training a red bottle detector, train a bottle detector, a sunglass detector, a red detector and white detector, and then by composing those detectors you can also detect a red bottle, all compositions of objects and their colors

Bottle detector

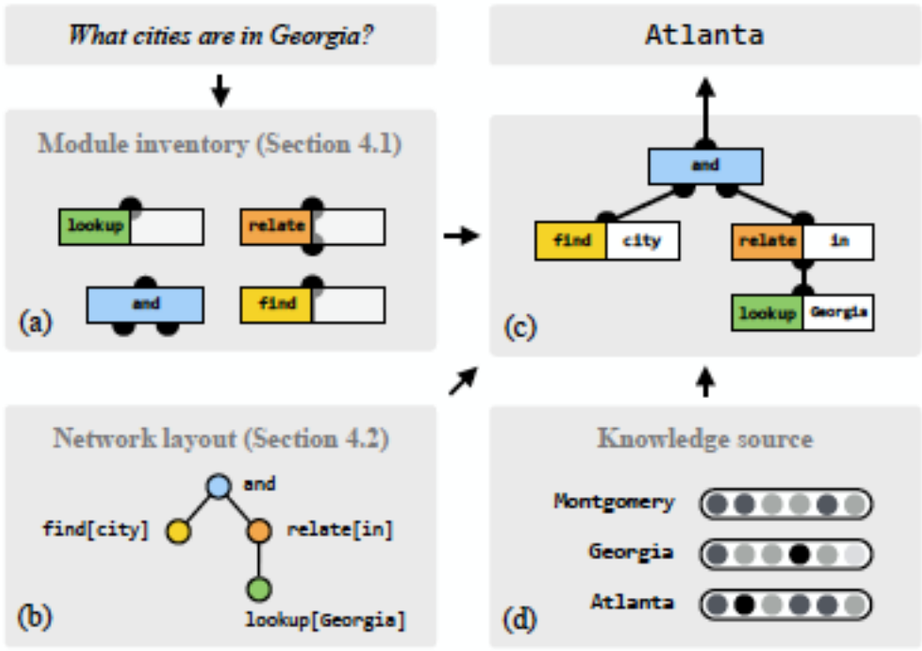
Color detector

(applied inside the bo

detected red bottle

Compositional policies

Identify subgoals and issue specific skills: hierarchical RL

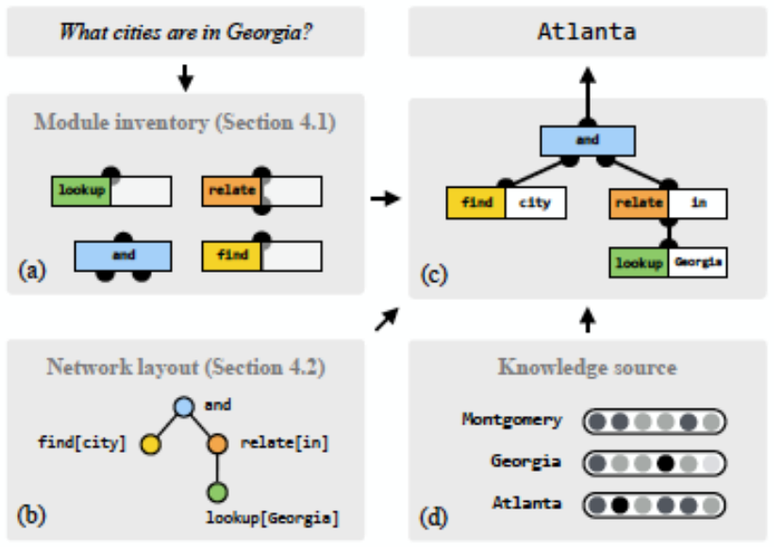


- Given a query, obtain the dependency parsing and issue neural models whose composition answers the query successfully
- A small initial vocabulary of neural modules
- Learning: which module to issue in which order. Learning using REINFORCE.

Learning to Compose Neural Networks for Question Answering, Jacob et al. 2016

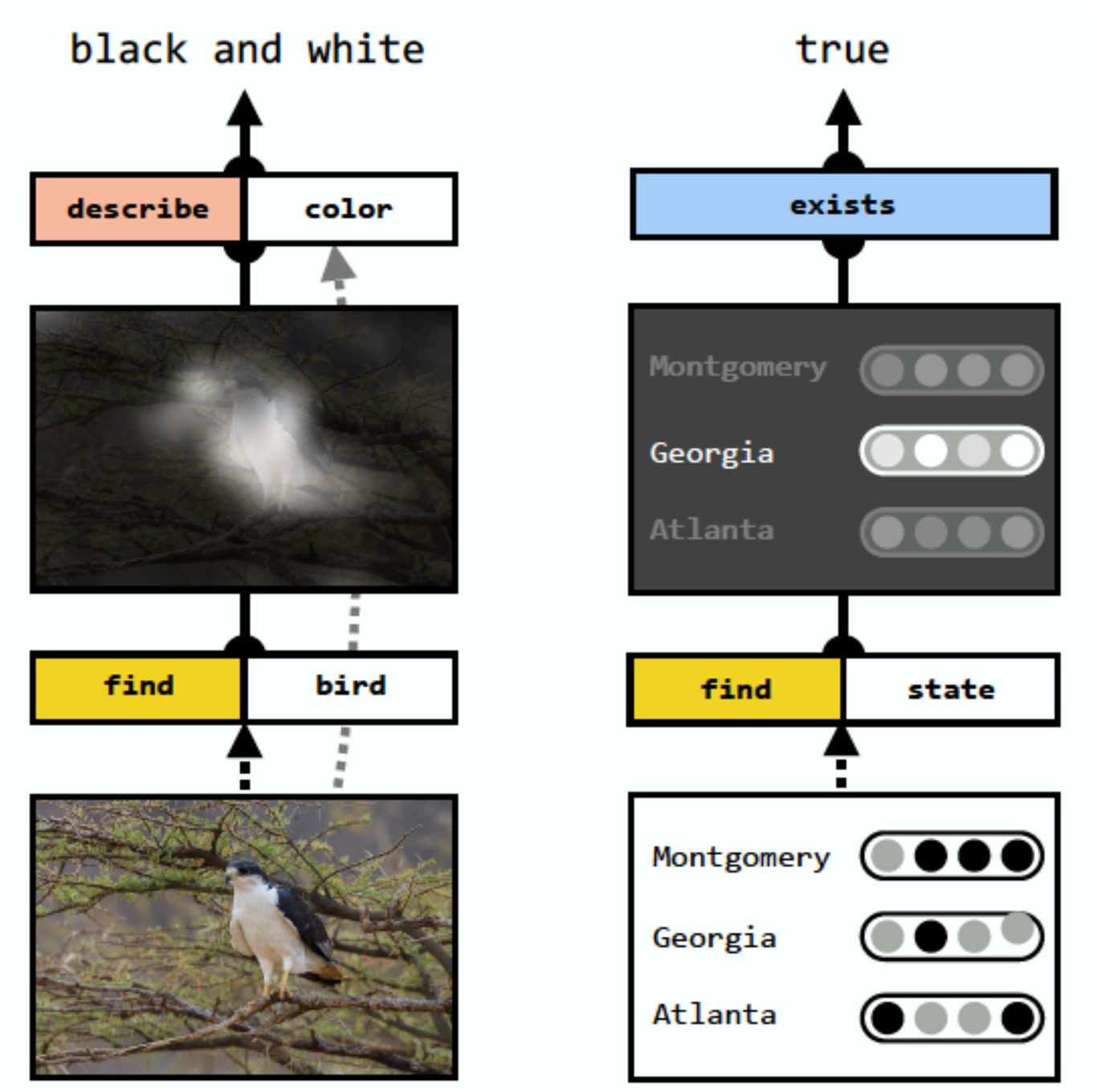
Learning to Compose Neural Networks for Question Answering

Jacob Andreas and **Marcus Rohrbach** and **Trevor Darrell** and **Dan Klein**
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
{jda,rohrbach,trevor,klein}@eecs.berkeley.edu



- Deep nets as reconfigurable programs: a set of (neural) modules chained together to produce an output, here, the answer to the question.
- A small initial vocabulary of neural modules
- Given a query, obtain the dependency parsing and learn to compose neural models whose composition answers the query successfully

Neural nets as functional programs



A small initial vocabulary of neural modules

Lookup (→ Attention)

`lookup[i]` produces an attention focused entirely at the index $f(i)$, where the relationship f between words and positions in the input map is known ahead of time (e.g. string matches on database fields).

$$[\![\texttt{lookup[}i\texttt{]}]\!] = e_{f(i)} \quad (2)$$

where e_i is the basis vector that is 1 in the ith position and 0 elsewhere.

Find (→ Attention)

`find[i]` computes a distribution over indices by concatenating the parameter argument with each position of the input feature map, and passing the concatenated vector through a MLP:

$$[\![\texttt{find[}i\texttt{]}]\!] = \text{softmax}(a \odot \sigma(Bv^i \oplus CW \oplus d)) \quad (3)$$

Relate (Attention → Attention)

`relate` directs focus from one region of the input to another. It behaves much like the `find` module, but also conditions its behavior on the current region of attention h. Let $\bar{w}(h) = \sum_k h_k w^k$, where h_k is the k^{th} element of h. Then,

$$[\![\texttt{relate[}i\texttt{]}(h)]\!] = \text{softmax}(a \odot \sigma(Bv^i \oplus CW \oplus D\bar{w}(h) \oplus e)) \quad (4)$$

And (Attention* → Attention)

`and` performs an operation analogous to set intersection for attentions. The analogy to probabilistic logic suggests multiplying probabilities:

$$[\![\texttt{and}(h^1, h^2, \ldots)]\!] = h^1 \odot h^2 \odot \cdots \quad (5)$$

Describe (Attention → Labels)

`describe[i]` computes a weighted average of w under the input attention. This average is then used to predict an answer representation. With $\bar{w}$ as above,

$$[\![\texttt{describe[}i\texttt{]}(h)]\!] = \text{softmax}(A\sigma(B\bar{w}(h) + v^i)) \quad (6)$$

Exists (Attention → Labels)

`exists` is the existential quantifier, and inspects the incoming attention directly to produce a label, rather than an intermediate feature vector like `describe`:

$$[\![\texttt{exists}](h)]\!] = \text{softmax}\left(\left(\max_k h_k\right)a + b\right) \quad (7)$$

A small initial vocabulary of neural modules

Lookup (→ Attention)

`lookup[i]` produces an attention focused entirely at the index $f(i)$, where the relationship f between words and positions in the input map is known ahead of time (e.g. string matches on database fields).

$$[\![\texttt{lookup[}i\texttt{]}]\!] = e_{f(i)} \quad (2)$$

where e_i is the basis vector that is 1 in the ith position and 0 elsewhere.

Find (→ Attention)

`find[i]` computes a distribution over indices by con-

Output is a distribution over indices (a set) over the domain of interest (pixels or entries of a database)

Relate (Attention → Attention)

`relate` directs focus from one region of the input to another. It behaves much like the `find` module, but also conditions its behavior on the current region of attention h. Let $\bar{w}(h) = \sum_k h_k w^k$, where h_k is the k^{th} element of h. Then,

$$[\![\texttt{relate[}i\texttt{]}(h)]\!] = \text{softmax}(a \odot \sigma(Bv^i \oplus CW \oplus D\bar{w}(h) \oplus e)) \quad (4)$$

And (Attention* → Attention)

`and` performs an operation analogous to set intersection for attentions. The analogy to probabilistic logic suggests multiplying probabilities:

$$[\![\texttt{and}(h^1, h^2, \ldots)]\!] = h^1 \odot h^2 \odot \cdots \quad (5)$$

Describe (Attention → Labels)

`describe[i]` computes a weighted average of w under the input attention. This average is then used to predict an answer representation. With $\bar{w}$ as above,

$$[\![\texttt{describe[}i\texttt{]}(h)]\!] = \text{softmax}(A\sigma(B\bar{w}(h) + v^i)) \quad (6)$$

Exists (Attention → Labels)

`exists` is the existential quantifier, and inspects the incoming attention directly to produce a label, rather than an intermediate feature vector like `describe`:

$$[\![\texttt{exists}](h)]\!] = \text{softmax}\left(\left(\max_k h_k\right)a + b\right) \quad (7)$$

A small initial vocabulary of neural modules

Lookup (→ Attention)

`lookup[i]` produces an attention focused entirely at the index $f(i)$, where the relationship f between words and positions in the input map is known ahead of time (e.g. string matches on database fields).

$$[\![\texttt{lookup[}i\texttt{]}]\!] = e_{f(i)} \quad (2)$$

where e_i is the basis vector that is 1 in the ith position and 0 elsewhere.

Find (→ Attention)

`find[i]` computes a distribution over indices by concatenating the parameter argument with each position of the input feature map, and passing the concatenated vector through a MLP:

$$[\![\texttt{find[}i\texttt{]}]\!] = \text{softmax}(a \odot \sigma(Bv^i \oplus CW \oplus d)) \quad (3)$$

Relate (Attention → Attention)

`relate` directs focus from one region of the input to another. It behaves much like the `find` module, but also conditions its behavior on the current region of attention h. Let $\bar{w}(h) = \sum_k h_k w^k$, where h_k is the k^{th} element of h. Then,

$$[\![\texttt{relate[}i\texttt{]}(h)]\!] = \text{softmax}(a \odot \sigma(Bv^i \oplus CW \oplus D\bar{w}(h) \oplus e)) \quad (4)$$

And (Attention* → Attention)

`and` performs an operation analogous to set intersection for attentions. The analogy to probabilistic logic suggests multiplying probabilities:

$$[\![\texttt{and}(h^1, h^2, \ldots)]\!] = h^1 \odot h^2 \odot \cdots \quad (5)$$

Describe (Attention → Labels)

`describe[i]` computes a weighted average of w under the input attention. This average is then used to predict an answer representation. With $\bar{w}$ as above,

$$[\![\texttt{describe[}i\texttt{]}(h)]\!] = \text{softmax}(A\sigma(B\bar{w}(h) + v^i)) \quad (6)$$

Exists (Attention → Labels)

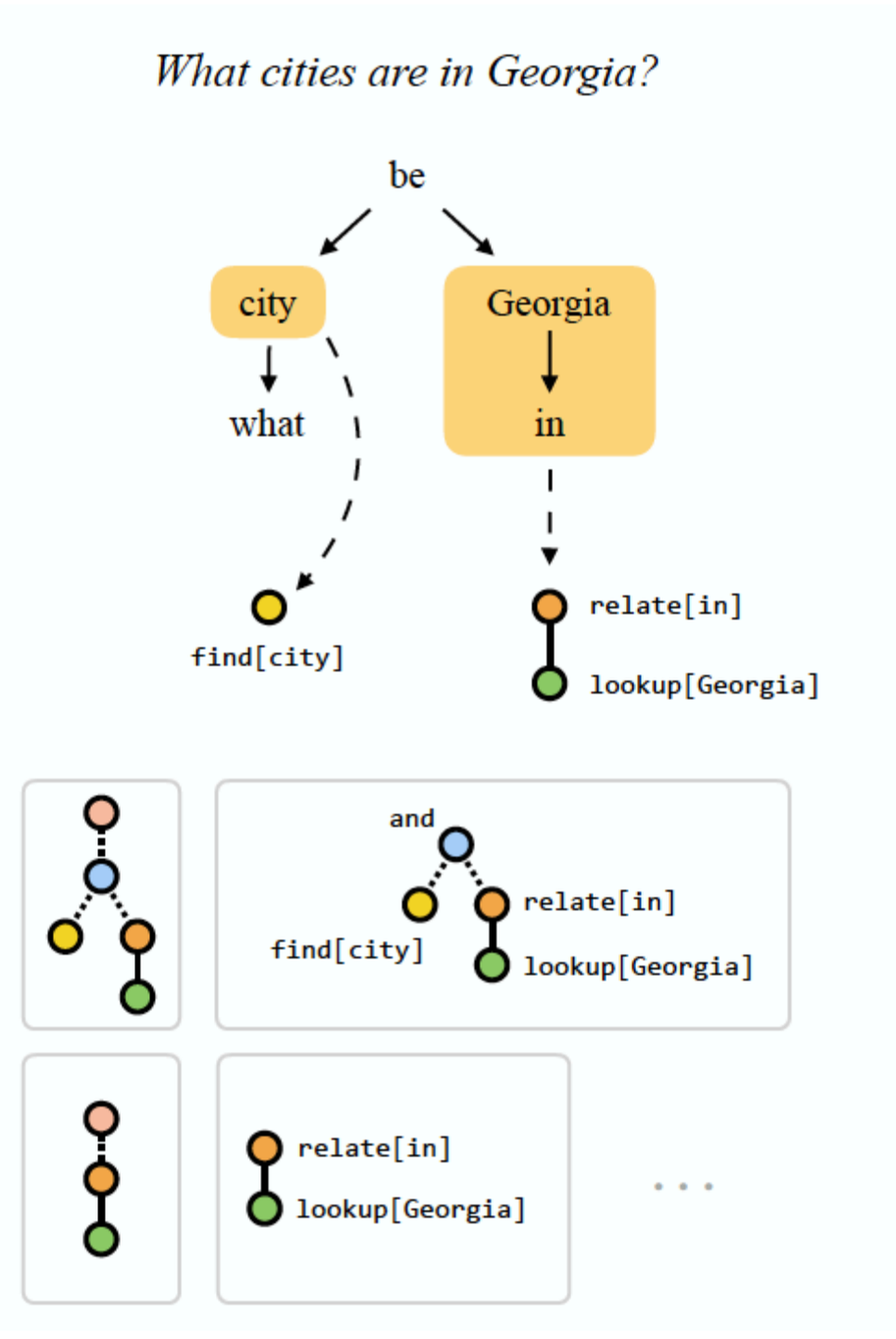
`exists` is the existential quantifier, and inspects the incoming attention directly to produce a label, rather than an intermediate feature vector like `describe`:

$$[\![\texttt{exists}]\!](h) = \text{softmax}\left(\left(\max_k h_k\right)a + b\right) \quad (7)$$

Different arguments in the brackets result in different weights for each module.

Neural module composition

Compositions are restricted by type constraints!



The input sentence is represented as a dependency parse.

Fragments of this dependency parse are then associated with appropriate modules, and these fragments are assembled into full layouts.

Given fixed layout, the parameters of the modules can be updated with standard SGD. Parameters are tied across module instantiations

Recipe for mapping parses into neural compositions

1. Represent the input sentence as a dependency tree.

2. Collect all nouns, verbs, and prepositional phrases and associate each of these with a layout fragment: Ordinary nouns and verbs are mapped to a single find module. Proper nouns to a single lookup module. Prepositional phrases are mapped to a depth-2 fragment, with a relate module for the preposition above a find module for the enclosed head noun.

4. Form subsets of this set of layout fragments. For each subset, construct a layout candidate by joining all fragments with an and module, and inserting either a measure or describe module at the top (each subset thus results in two parse candidates.)

Layout selection module

Trained using REINFORCE.

x: and LSTM encoding of the query

$$s(z_i|x) = a^\top \sigma(Bh_q(x) + Cf(z_i) + d)$$

$$p(z_i|x;\theta_\ell) = e^{s(z_i|x)} \Big/ \sum_{j=1}^{n} e^{s(z_j|x)}$$

Question Answering

What is in the sheep's ear?

```
(describe[what]
    (and find[sheep]
         find[ear]))
```

tag

What color is she wearing?

```
(describe[color]
    find[wear])
```

white

What is the man dragging?

```
(describe[what]
    find[man])
```

boat (board)

Is Key Largo an island?

```
(exists (and lookup[key-largo] find[island]))
```

yes: correct

What national parks are in Florida?

```
(and find[park] (relate[in] lookup[florida]))
```

everglades: correct

What are some beaches in Florida?

```
(exists (and lookup[beach]
             (relate[in] lookup[florida])))
```

yes (daytona-beach): wrong parse

What beach city is there in Florida?

```
(and lookup[beach] lookup[city]
     (relate[in] lookup[florida]))
```

[none] (daytona-beach): wrong module behavior