The How, What and Why of Deep Learning

Seminar 7: **Deep Reinforcement Learning and Frontiers in Deep Learning**

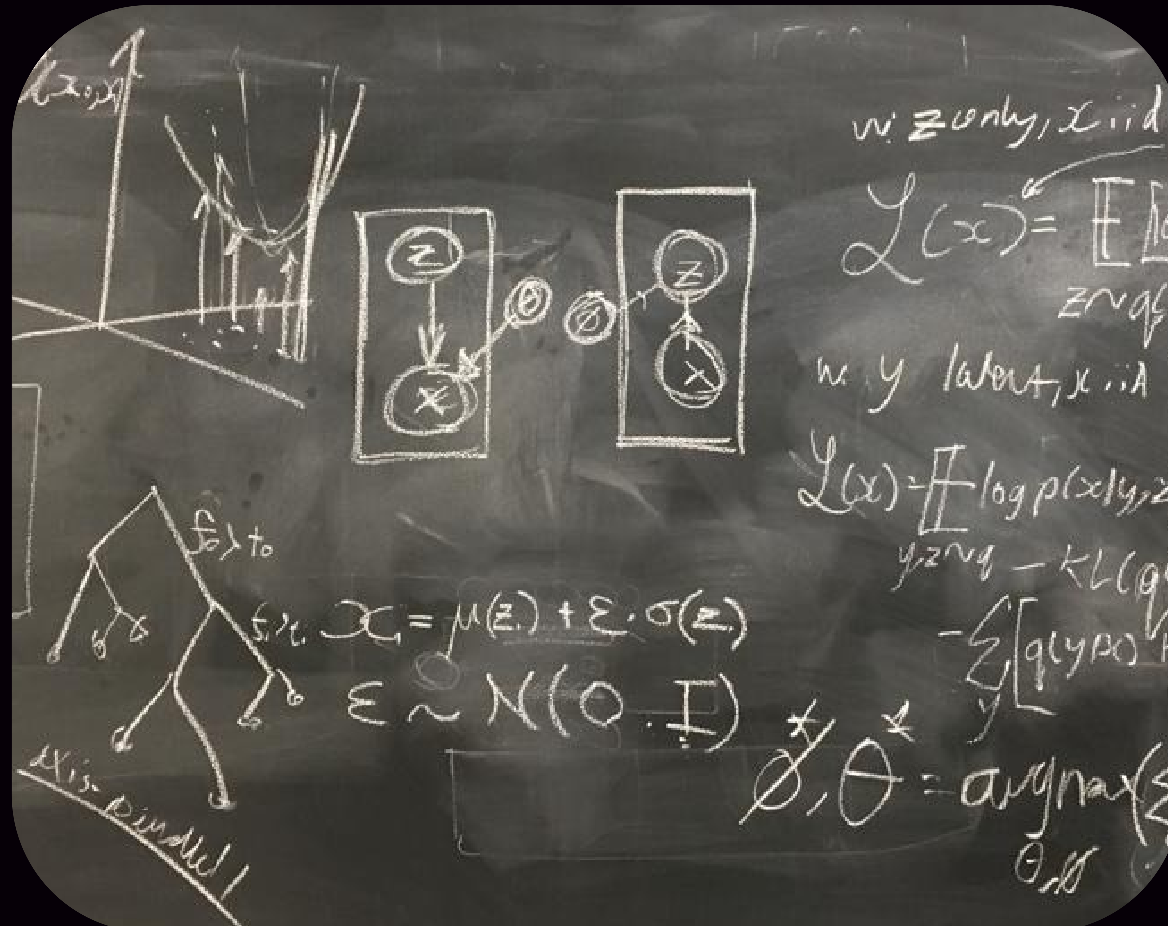
Big Data Institute, Oxford
⏰ June 14th, 2019

Alexander Camuto & Matthew Willetts

Course Timetable

- **22nd March**: Introduction: MLPs
- **5th April**: Computational Graphs: Implementations with Keras, Optimisation & Regularisation methods
- **19th April**: Convolutional NNs
- **3rd May**: RNNs and LSTMs
- **17th May:** Auto Encoders and NLP
- **31st May:** Deep Generative Models: VAEs & GANs, and their implementation with Tensorflow

> ***14th June:*** *Deep RL & Frontiers in Deep Learning*

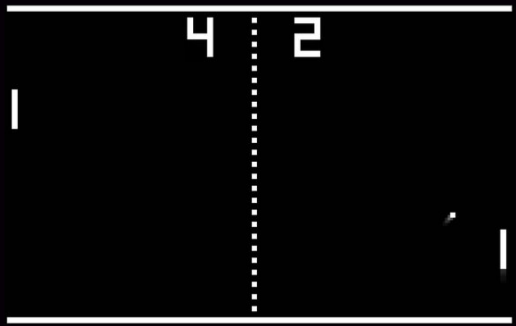


Today.

TODAY

- Policy Gradient.
- Q-learning
- Frontiers

Reinforcement Learning



- An agent interacts with an environment (eg. Pong).
- In each time step *t*, the agent receives observations(e.g. pixels) which give it information about the state (e.g. positions of the ball and paddle) the agent picks an action (e.g. keystrokes) which affects the state
  - observations(e.g. pixels) which give it information about the state (e.g. positions of the ball and paddle)
  - the agent picks an action (e.g. keystrokes) which affects the state
- The agent periodically receives a reward which depends on the state and action (e.g. points).
- The agent wants to learn a policy π: Distribution over actions depending on the current state and parameters θ.
  - Distribution over actions depending on the current state and parameters θ.

MDPs

- The environment is represented as a Markov decision process (MDP).

- Markov assumption: all relevant information is encapsulated in the current state.
- Components of an MDP:
  - initial state distribution $p(s_0)$
  - transition distribution $p(s_{t+1}|s_t, a_t)$
  - reward function $r(s_t, a_t)$
  - policy parametrised by a set of parameters θ $\pi_\theta(a_t|s_t)$
  - Assume a fully observable environment, i.e the state can be observed directly
- Rollout, or **trajectory** is a sequence of (state, action pairs): $\tau = (s_0, a_0, s_1, a_1, ..., s_T, a_T)$
- Probability of a trajectory modelled as an MDP is given by:

$$p(\tau) = p(\mathbf{s}_0)\,\pi_\theta(\mathbf{a}_0 \mid \mathbf{s}_0)\,p(\mathbf{s}_1 \mid \mathbf{s}_0, \mathbf{a}_0) \cdots p(\mathbf{s}_T \mid \mathbf{s}_{T-1}, \mathbf{a}_{T-1})\,\pi_\theta(\mathbf{a}_T \mid \mathbf{s}_T)$$

REINFORCE and Policy Gradient

- Return for a rollout: $r(\tau) = \sum_{t=0}^{T} r(\mathbf{s}_t, \mathbf{a}_t)$
- REINFORCE is an elegant algorithm for maximising the expected return $R = \mathbb{E}_{p(\tau)}\left[r(\tau)\right]$
- Intuition: Almost operates like trial and error
  - Sample a rollout τ. If you get a high reward, try to make it more likely.
  - If you get a low reward, try to make it less likely

REINFORCE and Policy Gradient

$$
\begin{aligned}
\frac{\partial}{\partial \boldsymbol{\theta}} \mathbb{E}_{p(\tau)}\left[r(\tau)\right] &= \frac{\partial}{\partial \boldsymbol{\theta}} \sum_{\tau} r(\tau) p(\tau) \\
&= \sum_{\tau} r(\tau) \frac{\partial}{\partial \boldsymbol{\theta}} p(\tau) \\
&= \sum_{\tau} r(\tau) p(\tau) \frac{\partial}{\partial \boldsymbol{\theta}} \log p(\tau) \\
&= \mathbb{E}_{p(\tau)}\left[r(\tau) \frac{\partial}{\partial \boldsymbol{\theta}} \log p(\tau)\right]
\end{aligned}
$$

- Compute stochastic estimates of this expectation by sampling rollouts

REINFORCE and Policy Gradient

$$\frac{\partial}{\partial \boldsymbol{\theta}} \log p(\tau) = \frac{\partial}{\partial \boldsymbol{\theta}} \log \left[p(\mathbf{s}_0) \prod_{t=0}^{T} \pi_{\boldsymbol{\theta}}(\mathbf{a}_t \mid \mathbf{s}_t) \prod_{t=1}^{T} p(\mathbf{s}_t \mid \mathbf{s}_{t-1}, \mathbf{a}_{t-1}) \right]$$

$$= \frac{\partial}{\partial \boldsymbol{\theta}} \log \prod_{t=0}^{T} \pi_{\boldsymbol{\theta}}(\mathbf{a}_t \mid \mathbf{s}_t)$$

$$= \sum_{t=0}^{T} \frac{\partial}{\partial \boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\mathbf{a}_t \mid \mathbf{s}_t)$$

REINFORCE and Policy Gradient

- The REINFORCE algorithm is basically gradient ascent on the expected return
- Note that we want to reinforce actions only based on **future rewards** after the action is chosen.

Repeat forever:

Sample a rollout $\tau = (\mathbf{s}_0, \mathbf{a}_0, \mathbf{s}_1, \mathbf{a}_1, \ldots, \mathbf{s}_T, \mathbf{a}_T)$

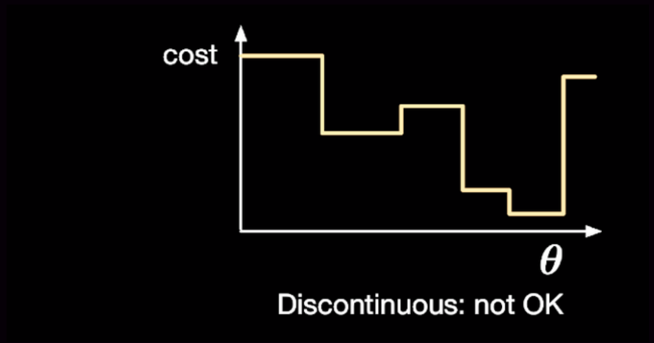
For $t = 0, \ldots, T$:

$r_t(\tau) \leftarrow \sum_{k=t}^{T} r(\mathbf{s}_k, \mathbf{a}_k)$

$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha r_t(\tau) \frac{\partial}{\partial \boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\mathbf{a}_k \mid \mathbf{s}_k)$

Optimising Discontinuous Objectives

- An RL formulation of a problem, can help optimise discontinuous cost functions that gradient descent could never solve !
- Gradient descent completely fails if the cost function is discontinuous:



Optimising Discontinuous Objectives

- RL formulation
  - one time step
  - state $\mathbf{x}$: an image
  - action $\mathbf{a}$: a digit class
  - reward $r(\mathbf{x}, \mathbf{a})$: 1 if correct, 0 if wrong
  - policy $\pi(\mathbf{a} \mid \mathbf{x})$: a distribution over categories
    - Compute using an MLP with softmax outputs – this is a policy network

Bringing things back to Backprop

- What's so great about backprop and gradient descent?
- Backprop does credit assignment – it tells you exactly which activations and parameters should be adjusted upwards or downwards to decrease the loss on some training example.
- REINFORCE doesn't do credit assignment. If a rollout happens to be good, all the actions get reinforced, even if some of them were bad. Reinforcing all the actions as a group leads to random walk behaviour.

Bringing things back to Backprop

- Assume an infinite horizon, i.e the number of steps per episode, *T*, is infinite.
- We can't sum infinitely many rewards, so we need to discount them: eg. $100 a year from now is worth less than $100 today
- Discounted return, with discount factor γ < 1:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$

- Value function of a state under a given policy: the expected discounted return if we start in **s** and follow **π:**

$$\begin{aligned} V^\pi(\mathbf{s}) &= \mathbb{E}[G_t \,|\, \mathbf{s}_t = \mathbf{s}] \\ &= \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} \,|\, \mathbf{s}_t = \mathbf{s}\right] \end{aligned}$$

Bringing things back to Backprop

- The benefit of the value function is credit assignment: we can see directly how an action affects future returns rather than wait for rollouts.
- Selecting an optimal action is more complex:
  - this requires taking the expectation with respect to the environment's dynamics, which we don't have direct access to:

$$\arg\max_{\mathbf{a}} r(\mathbf{s}_t, \mathbf{a}) + \gamma \mathbb{E}_{p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t)}[V^\pi(\mathbf{s}_{t+1})]$$

Q-function:

- expected returns if you take action and then follow your policy

$$Q^{\pi}(\mathbf{s}, \mathbf{a}) = \mathbb{E}[G_t \mid \mathbf{s}_t = \mathbf{s}, \mathbf{a}_t = \mathbf{a}]$$

- Relationship:

$$V^{\pi}(\mathbf{s}) = \sum_{\mathbf{a}} \pi(\mathbf{a} \mid \mathbf{s}) Q^{\pi}(\mathbf{s}, \mathbf{a})$$

- Optimal action:

$$\arg\max_{\mathbf{a}} Q^{\pi}(\mathbf{s}, \mathbf{a})$$

- The Bellman Equation is a recursive formula for the action-value function:

$$Q^{\pi}(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \gamma \mathbb{E}_{p(\mathbf{s}' \mid \mathbf{s}, \mathbf{a})\, \pi(\mathbf{a}' \mid \mathbf{s}')}[Q^{\pi}(\mathbf{s}', \mathbf{a}')]$$

Q-function

- Each time we sample consecutive states and actions $(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})$:

$$Q(\mathbf{s}_t, \mathbf{a}_t) \leftarrow Q(\mathbf{s}_t, \mathbf{a}_t) + \alpha \underbrace{\left[r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \max_{\mathbf{a}} Q(\mathbf{s}_{t+1}, \mathbf{a}) - Q(\mathbf{s}_t, \mathbf{a}_t) \right]}_{\text{Bellman error}}$$

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
    $S \leftarrow S'$;
  until S is terminal

Exploration - Exploitation

- Notice: Q-learning only learns about the states and actions it visits.
- Exploration-exploitation tradeoff: the agent should sometimes pick suboptimal actions in order to visit new states and actions.
- Simple solution: ϵ-greedy policy
  - With probability $1 - \epsilon$, choose the optimal action according to Q
  - With probability ϵ, choose a random action
- Believe it or not, ϵ-greedy is still used today!

Q-function

- Q-learning is an algorithm that repeatedly adjusts Q to solve the Bellman equation
- We've been assuming a tabular representation of Q: one entry for every state/action pair.
  - this is impractical to store for all but the simplest problems, and doesn't share structure between related states.
  - doesn't share structure between related states.
- Solution: approximate
  - linear function approximation

$$Q(\mathbf{s}, \mathbf{a}) = \mathbf{w}^\top \psi(\mathbf{s}, \mathbf{a})$$

  - compute Q with a neural net (Deep Q-Learning)

Deep Q-Learning

- For our training update rule, we'll use a fact that every Q function for some policy obeys the Bellman equation:

$$Q^{\pi}(s,a) = r + \gamma Q^{\pi}(s', \pi(s'))$$

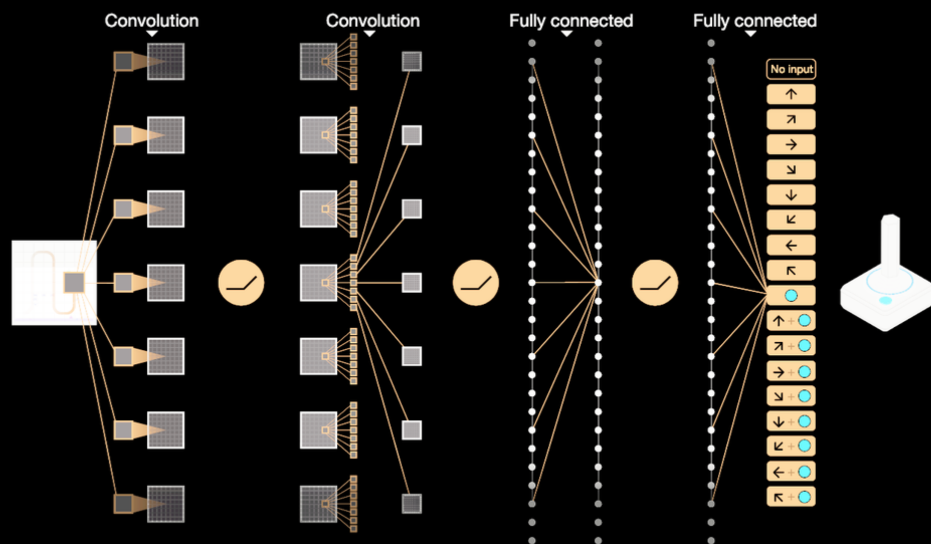
- The difference between the two sides of the equality is known as the temporal difference error, δ:

$$\delta = Q(s,a) - (r + \gamma \max_{a} Q(s',a))$$

- To minimise this error, we use the Huber Loss, which acts like the MSE when the error is small, but like the MAE when the error is large - making it more robust to outliers when the estimates of Q are very noisy. We sample a batch B from our replay memory:

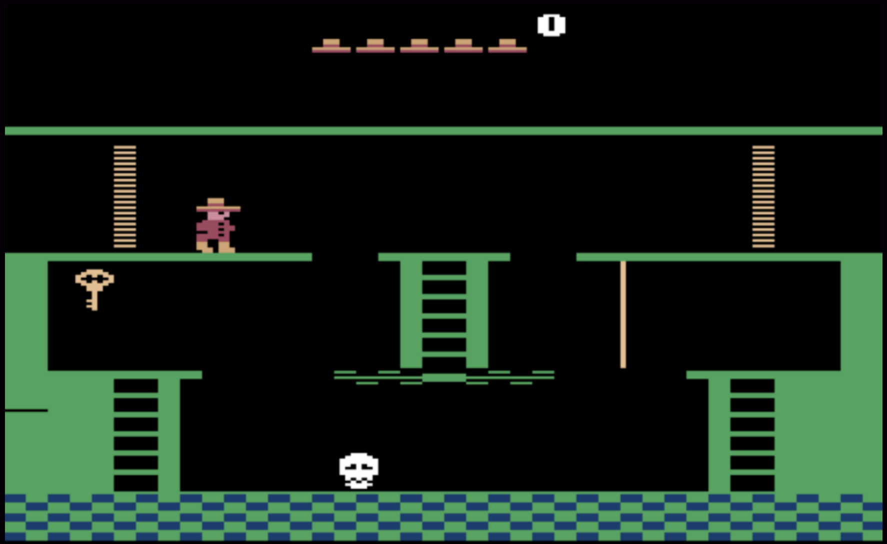
$$\mathcal{L} = \frac{1}{|B|} \sum_{(s,a,s',r) \ \in \ B} \mathcal{L}(\delta) \quad \text{where} \quad \mathcal{L}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \le 1, \\ |\delta| - \frac{1}{2} & \text{otherwise.} \end{cases}$$

Deep Q-Learning



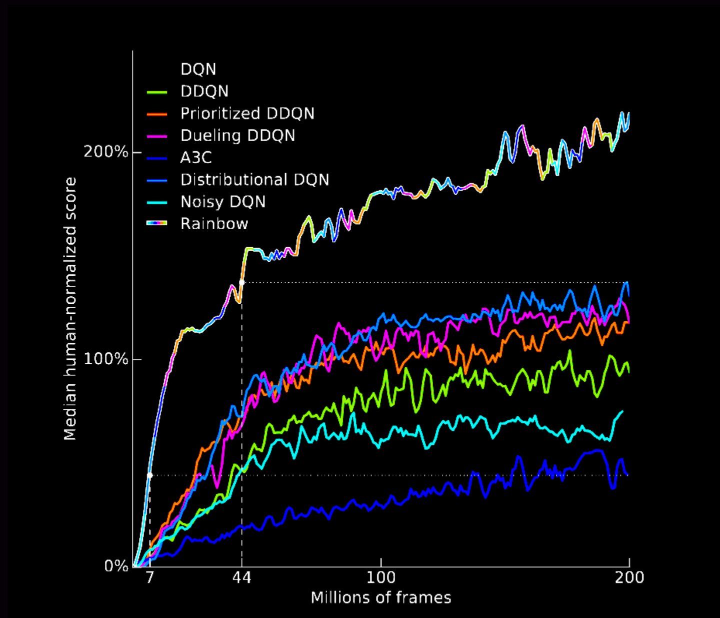
- **Human-level control through deep reinforcement learning; Mnih et al; 2015**
- Main technical innovation: store experience into a replay buffer, and perform Q-learning using stored experience
  - Gains sample efficiency by separating environment interaction from optimization — don't need new experience for every SGD update!

Deep Q-Learning



- **Human-level control through deep reinforcement learning; Mnih et al; 2015**
- Did very well on reactive games, poorly on ones that require long-term planning
    - (e.g. Montezuma’s Revenge)

Deep Q-Learning



- **Rainbow: Combining Improvements in Deep Reinforcement Learning; Mnih et al; 2017**

Deep Q-Learning

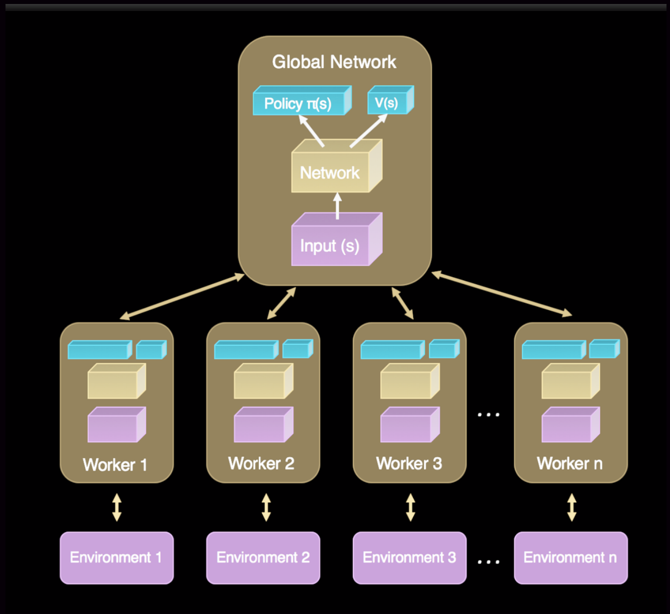
- Policy gradient and Q-learning use two very different choices ofrepresentation: policies and value functions

- Q-Learning
  - lower variance updates
  - does credit assignment
  - ✕ hard to handle large action-space (since you need to take the max)
  - ✕ biased updates since Q function is approximate (drinks its own Kool-Aid)

- Policy Gradient:
  - ☑ Unbiased estimate of gradient
  - ☑ can handle a large space of actions (since you only need to sample one)
  - ✕ high variance updates (implies poor sample efficiency)
  - ✕ no credit assignment

Actor Critic Methods

- Actor-critic methods combine the best of both worlds
  - Fit both a policy network (the “actor”) and a value network (“critic”)
  - Repeatedly update the value network to estimate the value function
  - Unroll for only a few steps, then compute the REINFORCE policy
    - update using the expected returns estimated by the value network
  - The two networks adapt to each other, much like GAN training

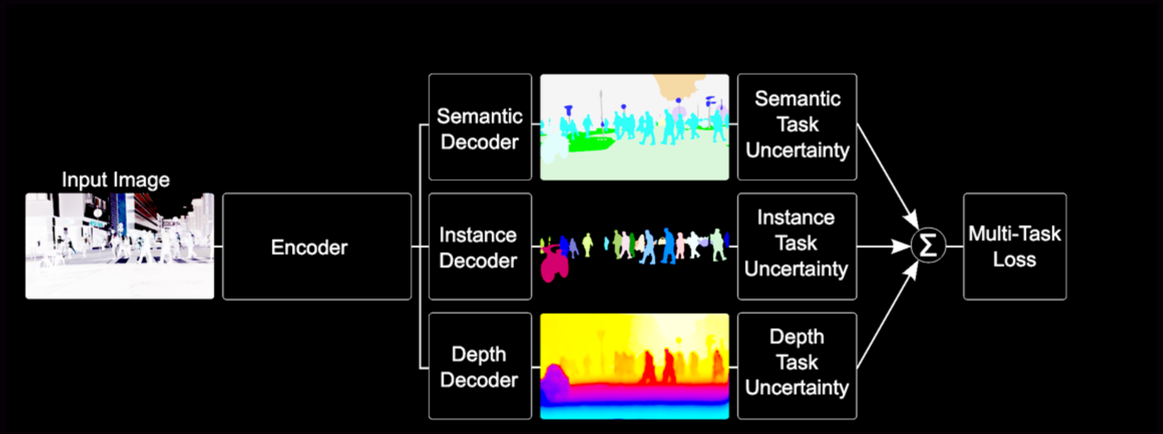


- **Asynchronous Methods for Deep Reinforcement Learning; Mnih et al.; 2016**

Inverse Reinforcement Learning

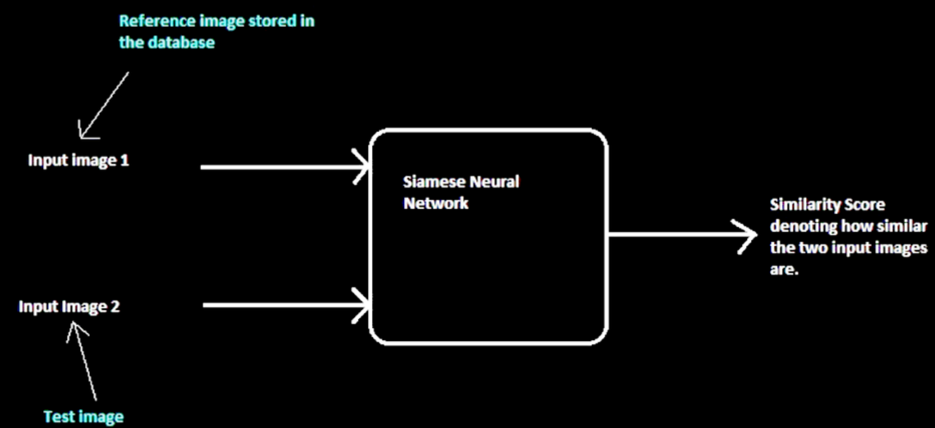
- Inverse reinforcement learning is the problem of inferring the reward function of an observed agent, given its policy or behavior.
  - This can be done asynchronously across many workers / agents !
  - Recent efforts have taken inspiration from GANs !
    - *Ermon ’16. Generative Adversarial Imitation Learning.*

Frontiers: Multi-Task Learning



- **Multi-Task Learning Using Uncertainty to Weigh Losses for Scene Geometry and Semantics; Kendall et al; 2018**

Frontiers: One-Shot Learning



- In one-shot learning we constrain Deep Learning systems to have only one training example for each class.
- Applications:
  - Facial recognition technology
  - Drug discovery where data is very scarce
  - Signature authentication
    - **Dey et Al., 2017. SigNet: Convolutional Siamese Network for Writer Independent Offline Signature Verification**

Frontiers: Zero-shot learning

- Zero-shot learning is a variant of multi-class classification problems where no training data is available for some of the classes. In most cases it consists in learning how to recognise new concepts by just having a description of them.
- Eg. Zebra = Horse + Stripes
- The most common approach is to re-use the learnt features in a Neural Network and rearrange them to perform some other task without additional training.
- **Paredes & Torr., 2015. An embarrassingly simple approach to zero-shot learning**

Acknowledgments

- Grant Sanderson for figures
- Gabriel Goh for the momentum simulations
- Chris Holmes and Gil McVean for arranging this seminar series