Reinforcement Learning Across Thousands of CPUs: Coach, Ray, and TensorFlow

Stephen Offer (stephen.offer@intel.com)

Introduction

The following topics will be covered:

- Deep Learning Review
- Distributed Training
- Reinforcement Learning
- The Machine Learning Ecosystem
- Ray
- Intel AI

The lecture will take one hour and I shall be available for questions afterwards.

Speaker Bio: Stephen Offer

I work at Intel helping university researchers and professors with:

- Researching distributed ML training
- Assisting research teams across the world with optimizing their work for large clusters
- Developing graduate courses in ML
- Creating software libraries for distributed computing and ML optimization

Deep Learning Review

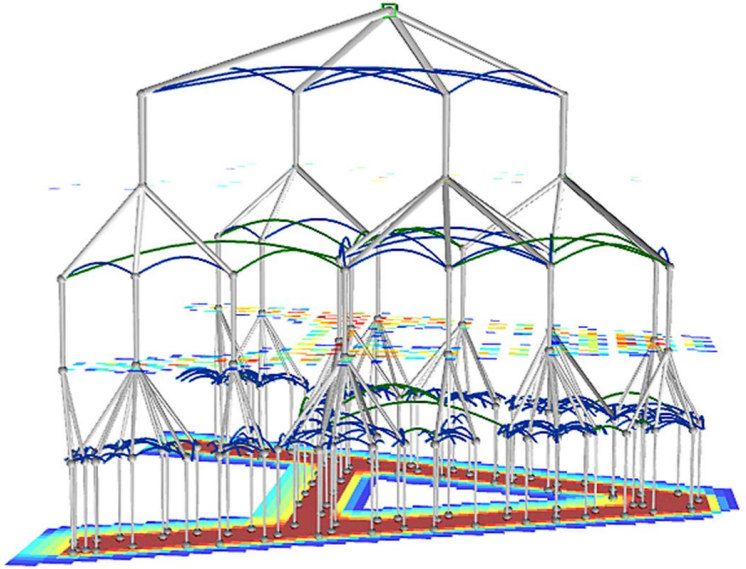
The Intuition behind Deep Learning

If somebody asked you why one example of the letter “A” looks like another example of the letter “A”, what would be your answer?

Maybe something like “there’s this slanted line here and another slanted line that meets it at the top and a horizontal line in the middle”?

What does your response tell you about how vision works? We can not only recognize simple shapes that form complex shapes but also understand them in relation to each other.

Your answer, however, would not be “There’s a line of approximately 0.5 cm wide at an angle of 135 degrees…” This is how traditional computer recognition systems might work, but deep learning instead tries to imitate the first example.



Imagine this shows how there is a hierarchy of shapes that you recognize when you see an “A”. The slanted lines on the edges are next to the other slanted lines which are next to the straight lines which form a corner...

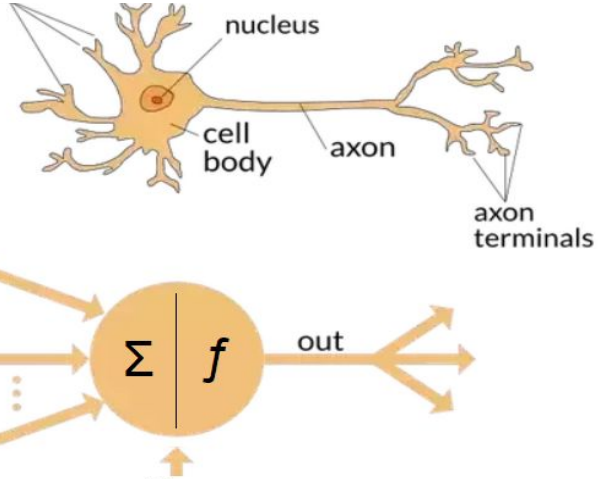
The Intuition Behind Deep Learning: Biology

So how do we create a system that is as flexible as the human vision system? By creating a very rough mathematical approximation of the process.

Although the brain is an extremely complicated process, we know that information is passed between neurons from one neuron's axon through a dendrite to another neuron's synapses.

Electrical signals are released when there are stimuli, such as changes of electric potential along the cell membrane or chemical transmitters.

To over-simplify the process, it can be reduced to a cell receiving multiple signals and then releasing multiple signals when certain conditions are met.



To imitate this, we have a model of a neuron where information comes into a node, weights change the value of the information, and an activation function gives the output of the node to the next node.

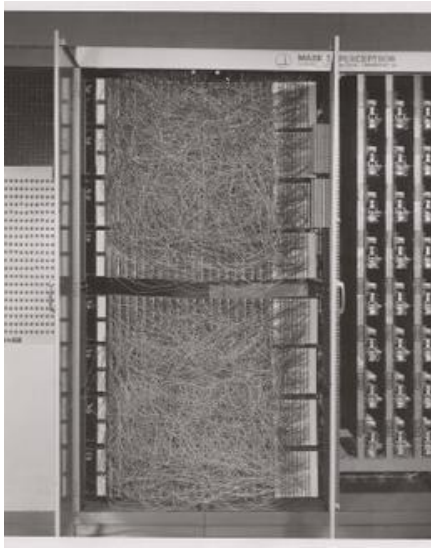
The Intuition Behind Deep Learning

A single neuron model is still a weighted sum, which means that it can only represent a linear relationship between the input and the output. So how can we represent non-linear relationships with these neurons?

This simple model was called the Perceptron, a single layer of these neuron models that learned to represent patterns.

However, it could only represent linearly-separable data, not non-linear data. It famously couldn't learn the simple XOR logic function, which lead to a decline in the study of neural networks until recently.

This problem was solved when it was discovered that if you had layers of neurons, the first layer would learn patterns from the input data but the second layer would learn patterns from the first layer and so forth. This solved the non-linear problem and created "deep neural networks", hence the term "deep learning".



The first implementation of the Perceptron built by its inventor Frank Rosenblatt in 1957 at the Cornell Aeronautical Laboratory.

Deep Neural Networks: How is information represented?

By stacking layers of neurons on top of each other, this solved the issue of non-linear relationships in data, but it didn't solve the issue of what the weights are inside each neuron.

The weights are how information is represented. Say that you have a node where the input is the following:

- A: Whether a person has a supercar [0 or 1]
- B: Whether the person lives on a farm [0 or 1]
- C: Whether the person has brown hair [0 or 1]

The output of the node will be whether that person is a billionaire. A positive number will be yes while a negative number will be no. The scale of the number will be your confidence in your answer.

So what your function look like?

Maybe something like this?

f(A,B,C) = 20*A - 10*B + 0*C

- A is high because if somebody has a supercar, they are rich regardless of where they live.
- B is low because farmers are not billionaires, but not so low as to misclassify the eccentric billionaire who has a supercar but lives on a farm.
- C is irrelevant, so it has no effect.

This relation between the data is represented in just these three numbers. Similarly, the connections in our neuron model represent the effect of the input data to the output.

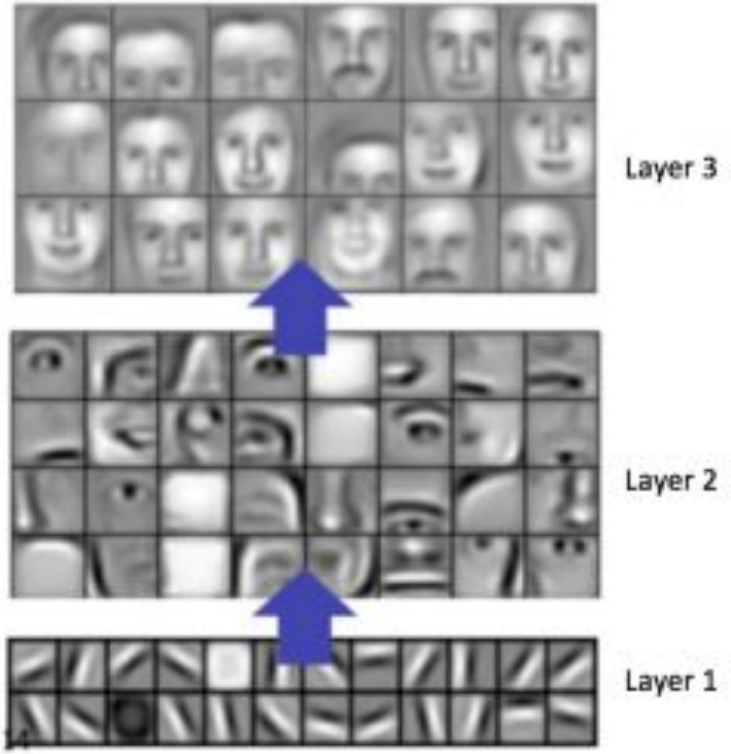
Neural Networks: Representing data

Deep neural networks answer the question of how to represent something in a similar way to the human visual cortex: a flexible hierarchical representation of spatial data.

To explain, when a human sees an “A”, they see the simple shapes and how these shapes spatially interact to create the letter. Our neuron model is able to create linear-relation in data to represent simple shapes such as curved lines, straight lines, or edges.

When we stack layers on top of the output of the first neuron layer, it learns more abstract information about the input, such as combinations of lines that form more complex shapes.

As we stack more and more layers, the information becomes more and more abstract to the point where a single neuron may only activate when there’s a face or a car in the image.



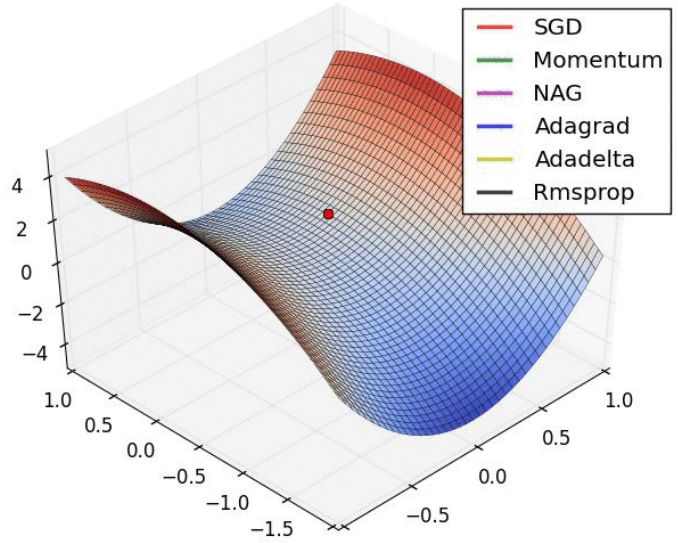
Neural Networks: How do we find the weights?

We have a model that can find patterns within data, but so far, no way of setting the weights to the right values to get the required output.

As soon as we add a cost function (how wrong the prediction of a model is compared to the actual label of the data), we can apply calculus to get a gradient of the weights with respect to the cost. This tells us which direction to move the values of the weights to lower the cost.

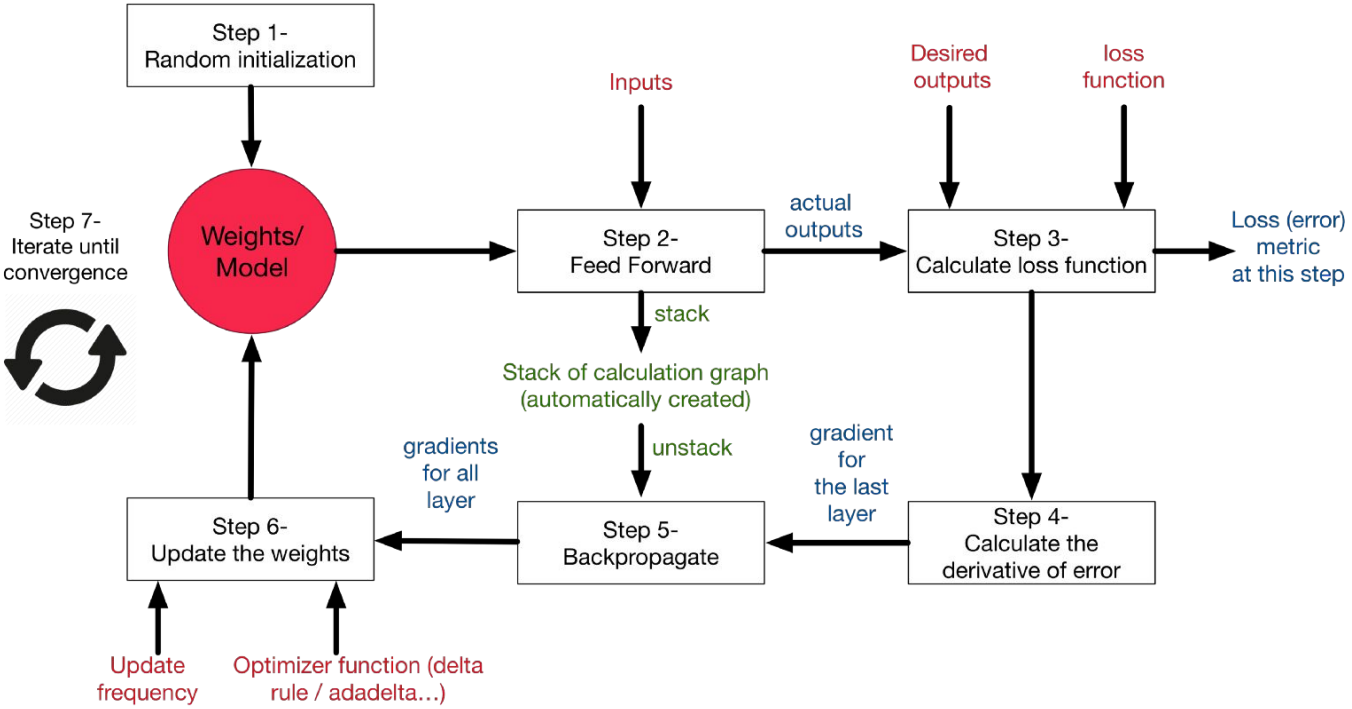
When we do this over and over again, the weights eventually converge to a point where they represent the data, enough to make predictions.

These changes to the weights are fed back into the neural network by an algorithm called Backpropagation.



A simple two variable example of different optimization algorithms. Note that in deep learning, since each variable creates another dimension, it is actually performing gradient descent in a high-dimensional space (millions of dimensions).

Deep Learning Training Process:



Neural Networks: How to build networks

Neural networks used to be difficult to build, but now there are many frameworks to quickly build these models.


intel AI
Distributed Training

The challenges of deep learning training

Training is very resource expensive:

- Memory to store activations, weights, dataset.
- Processor speed and memory bandwidth

- Training is normally synchronous, since one layer feeds into the next. It can't be usually be split up asynchronously in the forward pass, and the backward pass can't be computed without the forward pass being complete.
- Being synchronous and computationally expensive, it is very time consuming, taking anywhere from a few minutes to many days.

model	input size	param mem	feat. mem	flops
alexnet	227 x 227	233 MB	3 MB	727 MFLOPs
caffenet	224 x 224	233 MB	3 MB	724 MFLOPs
squeezenet1-0	224 x 224	5 MB	30 MB	837 MFLOPs
squeezenet1-1	224 x 224	5 MB	17 MB	360 MFLOPs
vgg-f	224 x 224	232 MB	4 MB	727 MFLOPs
vgg-m	224 x 224	393 MB	12 MB	2 GFLOPs
vgg-s	224 x 224	393 MB	12 MB	3 GFLOPs
vgg-m-2048	224 x 224	353 MB	12 MB	2 GFLOPs
vgg-m-1024	224 x 224	333 MB	12 MB	2 GFLOPs
vgg-m-128	224 x 224	315 MB	12 MB	2 GFLOPs
vgg-vd-16-atrous	224 x 224	82 MB	58 MB	16 GFLOPs
vgg-vd-16	224 x 224	528 MB	58 MB	16 GFLOPs
vgg-vd-19	224 x 224	548 MB	63 MB	20 GFLOPs

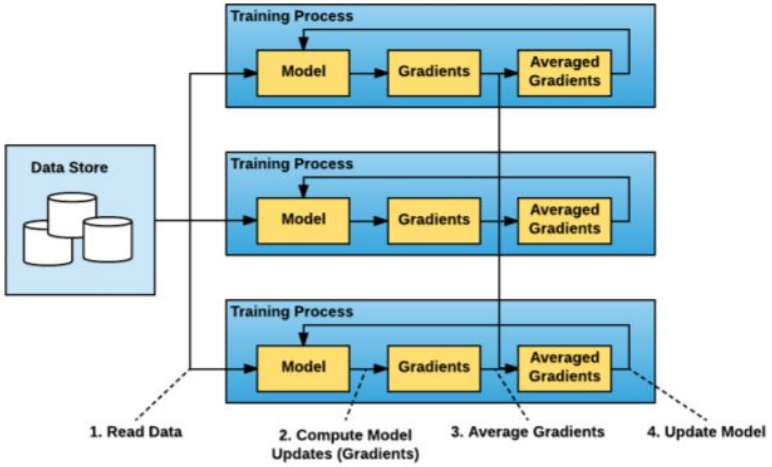
Distributed Training: Weight Averaging

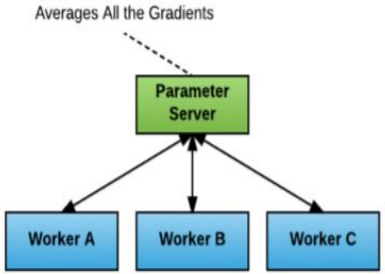
Weight averaging solves the sequential problem by creating copies of one model and training each on batches of the dataset. The gradients are averaged and returned to each model. This method imitates mini-batch learning and reduces the time required to go over the entire dataset.

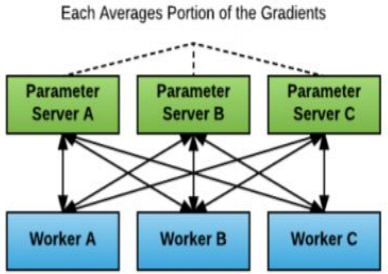
There are several problems with weight averaging:

- With many workers, the averaged weights become noise.
- As the models train, the weights diverge from each other to their own local minimums, so they have to be averaged and updated precisely at an optimal work vs. communication cost time.

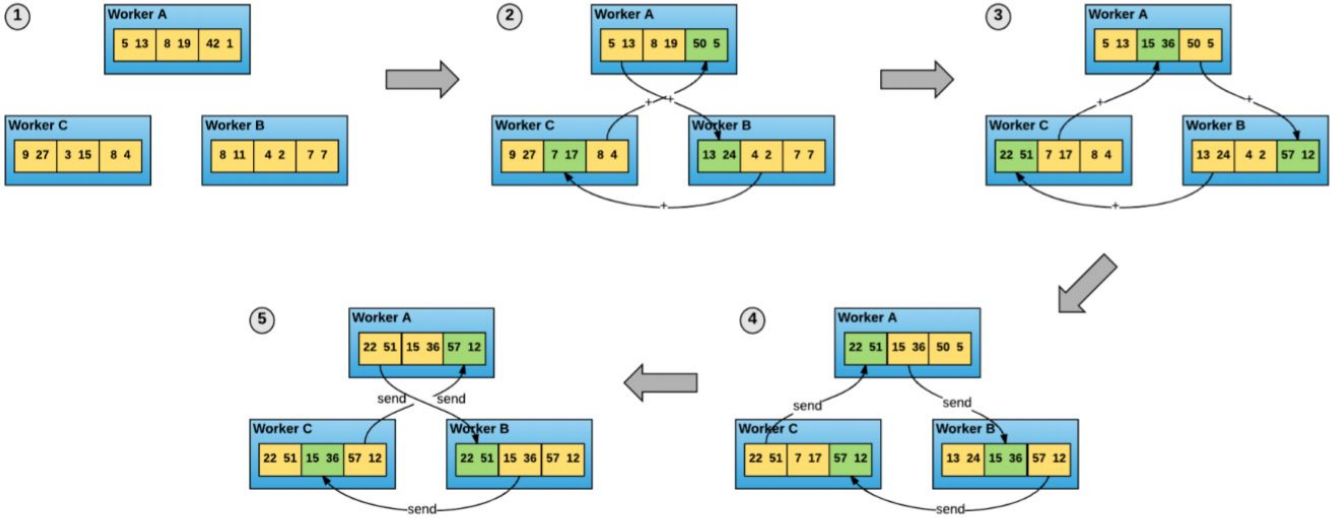
There are also variants including multiple parameter servers for multiple “central models.”







Distributed Training: Ring All-Reduce



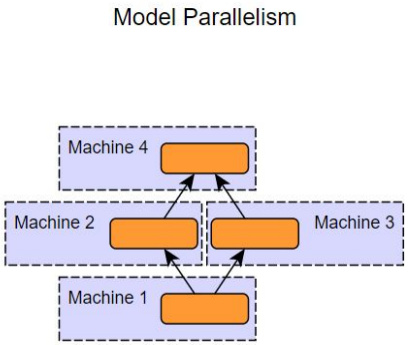
To solve the problem of model divergence and to make it so all workers are training an abstract “central model”, Ring All-Reduce utilizes the fact that each model copy has the same layer shape and exchanges layers. This eliminates the need for a central parameter server since the workers communicate directly with the other workers.

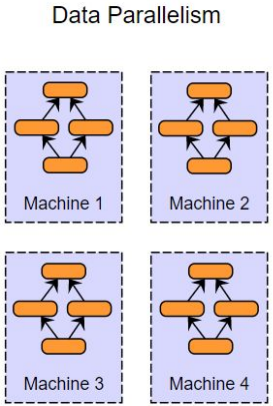
Distributed Training: Distributing the Model Itself

So far, the methods described have been examples of data parallelism, where the data itself is copied over to several nodes and there is some technique to extract a final model.

Another way is model parallelism, where the model itself is distributed across multiple nodes. The training algorithm is the same as regular single node training, so why bother with data parallelism? What issues might model parallelism create?

- It is more difficult to implement model parallelism.
- The I/O cost of transferring gradients across multiple nodes may not outweigh the advantages of single node.





Model parallelism does work well with complex or multi-network systems or supernetworks.

It also works well for networks where the size of the network is greater than a single node's memory.

Reinforcement learning

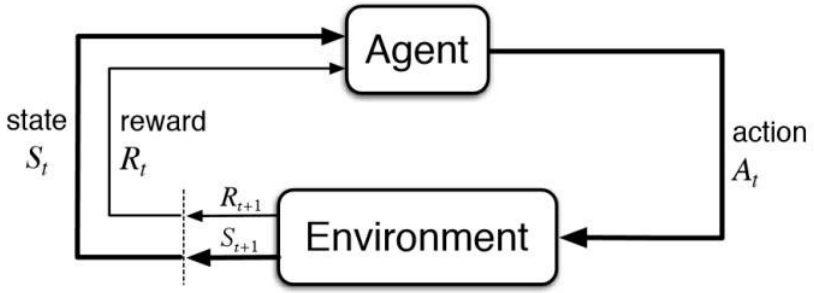
Introduction to Reinforcement Learning

A RL agent is a model that can learn from an environment about how to maximize a reward function.

It does this by looking at the current state and reward value and learning what action to take through a **policy** function. The policy is the behavior of an agent.

It determines how good a state is through a **value** function that learns how valuable the current and future states will be.

A RL **model** is anything that can represent an environment in terms of reward or transitions. It is a representation of the environment the agent is in.



For example, in a Chess RL system:

The **value** function tells how good the state of the board is.

The **policy** function tells to move a piece here and not there.

The **model** might be a CNN that can represent the state of the board in a way that can output the value or policy.

Biological Evidence for Reinforcement Learning

All animals clearly follow the basic principle of RL: increase a reward value through action.

For humans, our reward value is a very complex function comprising of physical, neurological, and social reward systems.

- Physical: food, drink, reproduction, comfort, avoidance of pain, etc.
- Neurological: dopamine, addictions, etc.
- Social: self perception, interaction, respect, lifestyle, goals, money, etc.

For each animal, the relations between these systems are different, but are still found in all animals. Evolutionary theory suggests that intelligence was promoted in the effort to maximize these rewards.

Reinforcement learning is based on the same questions that psychologists answer:

- How does an animal learn from delayed reward?
- How can we infer from previous tasks to learn new tasks?
- How to prioritize rewards in order to meet a final goal?

The Importance of Games

Since RL requires the agent to “experience” reward from actions, this means that the system needs to be in charge of either a robot or an agent in a simulation.

Value functions are difficult to properly define since in the biological sense, they are very complex. Games provide an easier way to give that reward function since they already give it in terms of whether the agent wins the game and/or how many points it earns.

Games are also beneficial since they can require very high-cognitive abilities like strategy, recognition, working with other agents to complete a task, and many other qualities that are very difficult for computers to master.

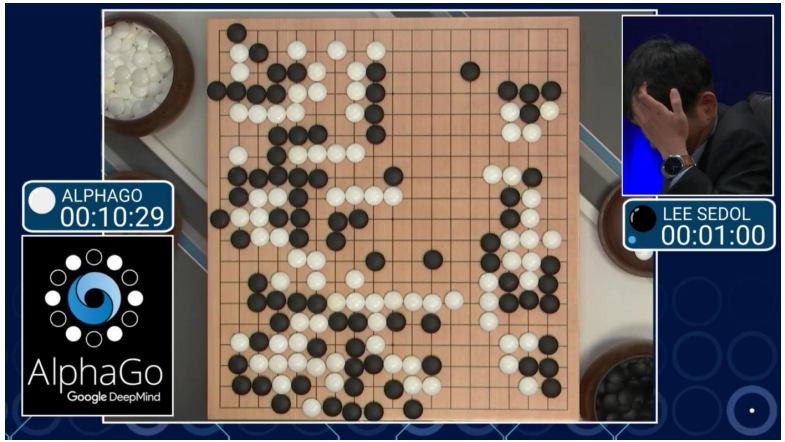
RL Agents have reached human-level or above at:

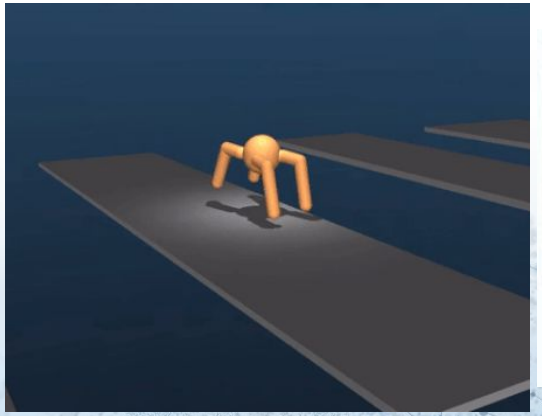
- Chess
- Go
- Starcraft II
- Atari
- And many more...

However, RL is not just applicable to games, which are simply a good way of testing algorithms. They are also used for many other types of problems. Some include:

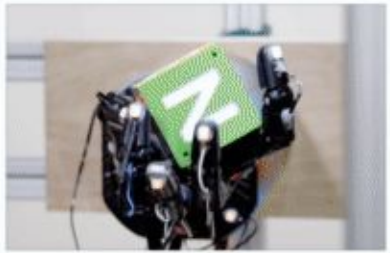
- Automated driving
- Robotic dynamics
- Aerospace dynamics
- Medical diagnosis

Research Uses of RL





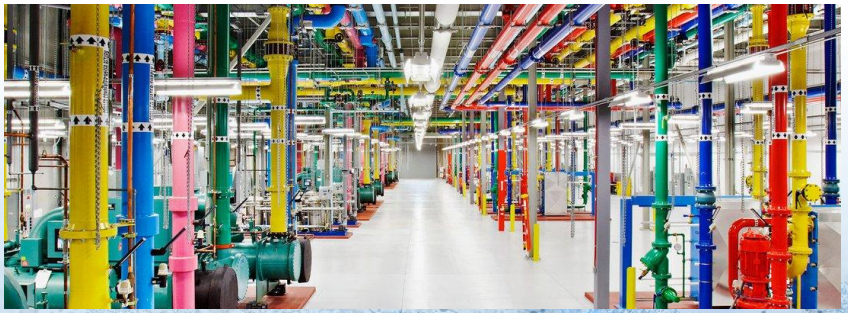
FINGER PIVOTING



SLIDING

FINGER GAITING

Industry Uses of RL



Markov Decision Processes (MDP)

A state is **Markov** iff

$$\mathbb{P}\left[S_{t+1}|S_t\right] = \mathbb{P}\left[S_{t+1}|S_1, ..., S_t\right]$$

A **state transition probability** from s to s' is:

$$P = \mathbb{P}\left[S_{t+1} = s'|S_t = s\right]$$

A **state transition matrix** for N states is NxN where (i,j) is the state transition probability for state i to state j.

A **Markov Chain** is a tuple (S,P) for S states and P STM.

A **Markov Reward Process** is tuple (S,P,R,y) for S states, P STM, R reward function, and discount factor.

The **return** is not the reward of a state, it is the expected total reward for all future states.

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

The **value** of a state is: $v(s) = \mathbb{E}\left[G_t|S_t = s\right]$

The value can be rewritten as the **Bellman equation**:

$$v(s) = R_s + \gamma \sum_{s' \in S} P_{ss'} v(s)$$

For small MDPs, this can be solved linearly with O(n^3):

$$v = (I - \gamma P)^{-1} R$$

For large MDPs, there are other ways to solve MDPs:

- Dynamic Programming
- Monte-Carlo evaluation
- Many more…

MDPs are important because they not only introduce the information of a state but also provide context behind many types of RL algorithms.

RL Policies: Introducing choice into MDPs

Markov states describe the environment of an agent, but do not describe the behavior of an agent. The state changes depends on the behavior of the agent.

A **policy** is the behavior of an agent:

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$$

The **state-value function** of an MDP from a policy is:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

The **action-value function** is the return with a policy:

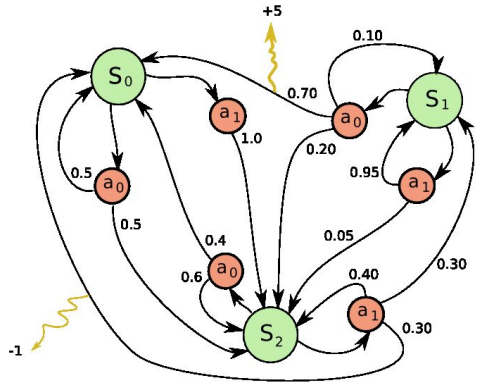
$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$$

An **optimal state-value function** is finding the best policy:

$$v_*(s) = \max_\pi v_\pi(s)$$

An **optimal action-value function** is also finding the best policy

$$q_*(s) = \max_\pi q_\pi(s, a)$$



A MDP is solved when the optimal value function is found.

For any MDP:

- There exists an optimal policy
- All optimal policies achieve the optimal value function
- All optimal policies also achieve the optimal action-value function

Actions are selected by the max of the policy function.

Problems with MDPs

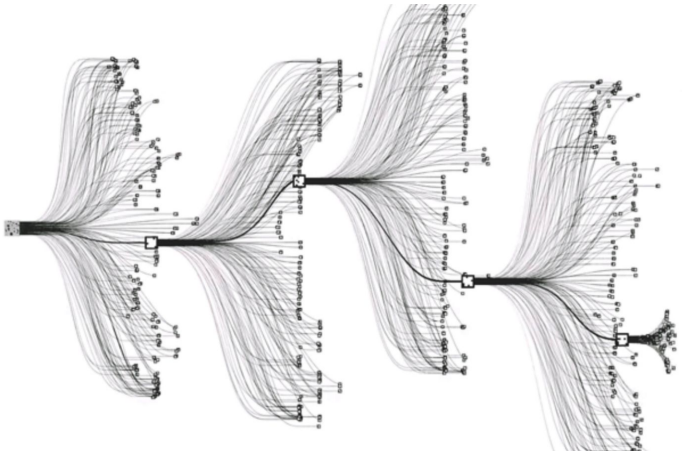
Of course, not all environments are Markov or fit into this optimal situation.

Partially Observable Markov Decision Process is when there are hidden states and actions. A POMDP consists of S states, A actions, O observations, P STPM, R reward, Z observation, and a gamma discount factor.

This introduces the incorporation of **history** sequences and **belief state**, probabilistic view of states based on previous states:

$$H_t = A_0, O_1, R_1, ..., A_{t-1}, O_t, R_t$$

$$b(h) = (\mathbb{P}\left[S_t = S^1 | H_t = h\right], ..., \mathbb{P}\left[S_t = S^n | H_t = h\right])$$



Another issue may be infinite or continuous MDPs.

For example, although Go may be seen as an MDP, for a game with 400 moves, there are 1e1023 possible states.

Continuous action spaces, such as angle movements of a robot, also show where our general model of an MDP has to be modified to accommodate different types of action spaces or state types.

Important Distinctions in RL

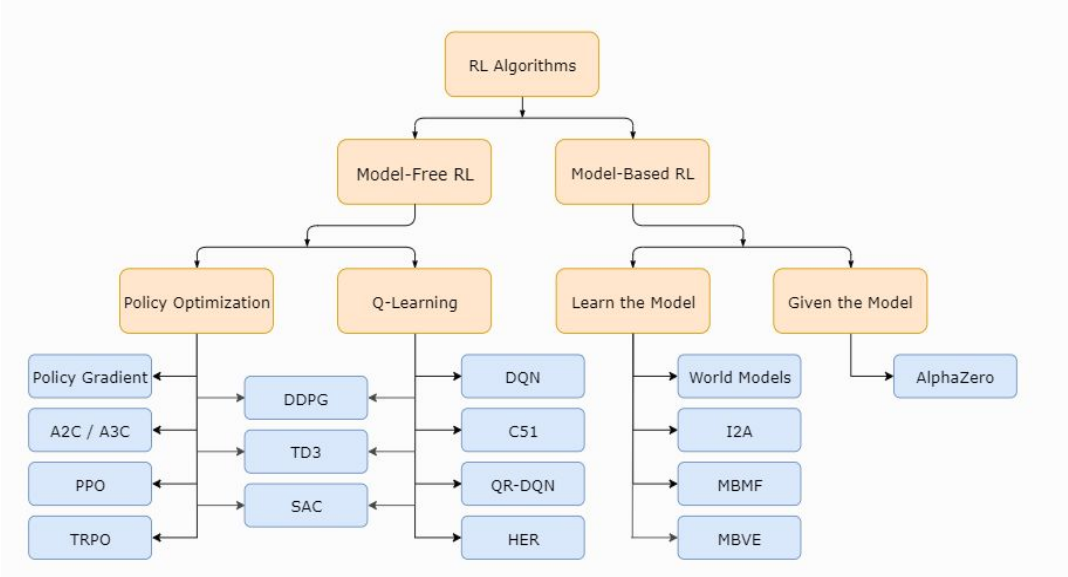
MDP showcases many important distinctions in RL:

An algorithm that has a **model** of the environment refers to whether it has a function that predicts state transitions and rewards, not whether it uses a neural network model. Algorithms are divided into two categories: model and **model-free**.

Environments can be **fully observable** or **partially observable**.

Action spaces can be **discrete** or **continuous**.

The value of states can be described in terms of the **state-value** or **action-value** or the **advantage-value**, which is the value of the how much better each action is over the other actions (the difference between the action-value and state-value).



Just because a state is fully observable does not make it Markov, nor does a partially observable state mean that it is not Markov.

Solving MDP with Dynamic Programming

Dynamic programming consists of subdividing a difficult task into subproblems that utilize recursive functions. MDPs can be solved with Dynamic Programming since the Bellman equation can be expressed in terms of itself.

Dynamic Programming requires two properties:

- The **principle of optimality** states that for any initial state, the remaining decisions are optimal with respect to the resulting states. This refers to optimal subproblems.
- **Overlapping subproblems.**

Ex: Iterative Policy Evaluation:

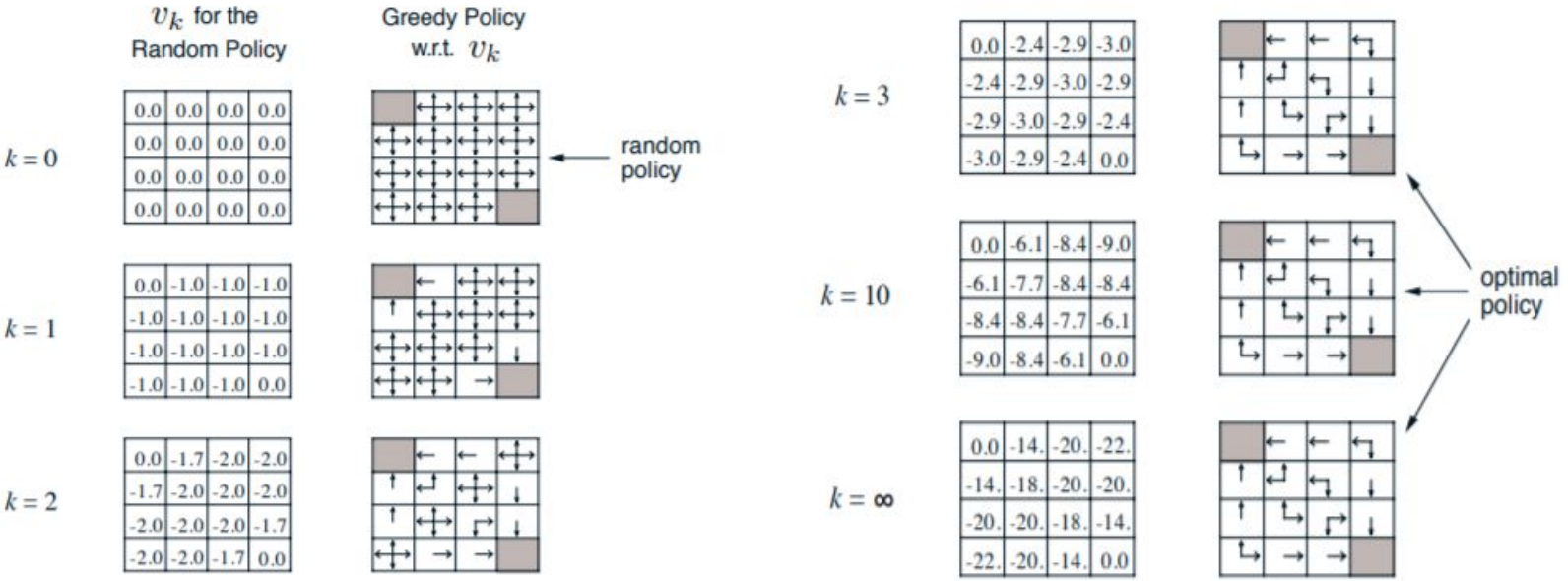
$$v_{k+1}(s) = \sum_{a \in A} \pi(a|s) \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s') \right)$$

For the example on the next slide, we're given a gridworld.

- The two grey corners are exit nodes.
- The policy must go to the exit nodes.
- The policy initially begins with an equal distribution across N, S, E, W directions, so 0.25 probability that the agent will move in that direction.
- The agent can begin anywhere on the grid randomly.
- The reward of each state not being an exit node is -1.
- The iterative policy update algorithm is a simple greedy algorithm which points to the neighboring state that has the max value.

The **principle of optimal policies** states that for any initial state, the optimal policy must be able to reach the optimal reward.

Solving MDP with Dynamic Programming



In this example, policy iteration converges to the optimal policy by first evaluating the policy and then improving it through a greedy algorithm (minimizing loss in each direction)

Model Free Prediction: Monte-Carlo

What if a state is Markov but the model of the environment is unknown?

Model-free algorithms estimate the value of state in the absence of an environment model.

Agents must learn from experience rather than from an MDP model.

Ex: Monte-Carlo Policy Evaluation:

Every time-step *t* that state *s* is visited in an episode:

State counter *N(s) += 1*

Total return *S(s) + G_t*

State value *V(s) = S(s)/N(s)*

As *N(s)* gets higher, it reaches the policy evaluation of the state.

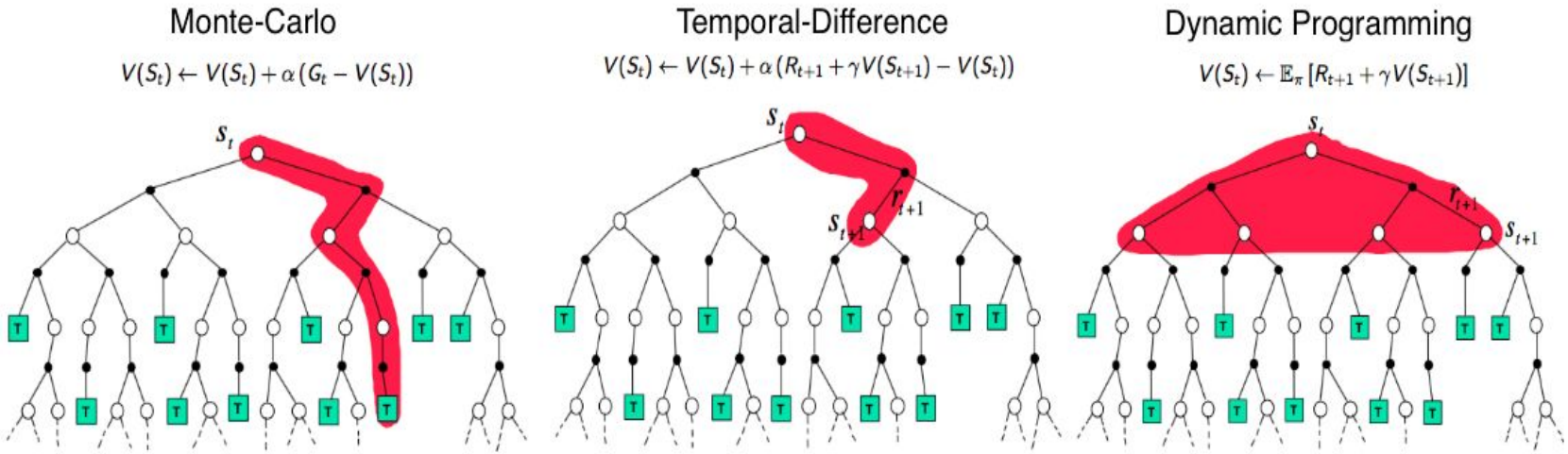
Simply put, a Monte-Carlo prediction finds the value of a state by looking at experience from the state to the terminal state in that episode.

The state counter is included because the state between the first state and the terminal state may have been visited before. The value of the initial state should not incorporate the other episodes heavily. The value, therefore, is averaged by the state counter.

What are some possible problems this method?

- What if a future state has an unusually high value?
- What if a future state is independent of the initial state?
- What if the environment does not have a terminal state?

Model Free Prediction: Temporal-Difference



To fix the problems hinted at in the previous slide, Temporal-Difference is used to find the value of the state by looking at the value of just the current state and next state.

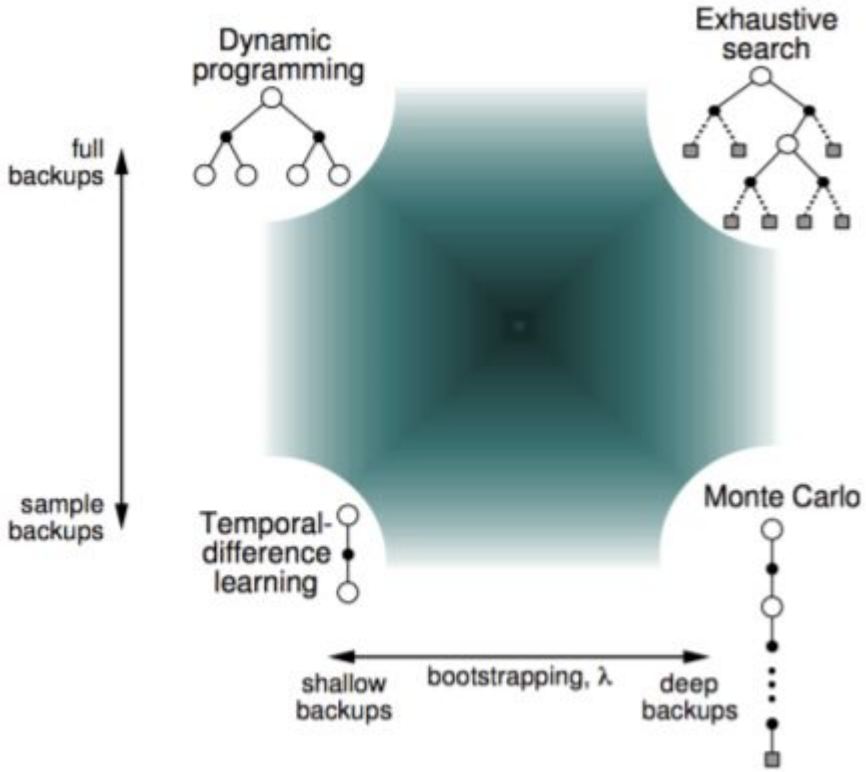
It updates the value of the state by an estimated return value instead of having to look at the entire episode. This allows it to work in continuous environments that have no terminal states, unlike Monte-Carlo predictions.

Model Free Prediction: Terminology and Taxonomy

Sampling is “sampling the actual experience data” instead of the expectation of the data.

Bootstrapping is when an estimate of a value is made using another estimate of the same kind of value (such as how much an estimate of value depends on another estimate of value). This is important in determining the uncertainty of the prediction.

Sampling and bootstrapping describe the relation between the prediction method and its reliance on actual data vs. predicted data.



Model Free Control, Exploration, and Exploitation

On-policy learning is when the policy is being optimized while playing.

Off-policy learning is when the policy learns from experience, “learns strategy from watching”.

We’ve covered a few examples on how to evaluate policies, but how are policies improved? One way is to introduce **policy exploration**- increasing random moves to explore different actions to see what happens.

Exploration is choosing more uncertain actions in the hopes that new actions may result in new rewards.

Exploitation is using previously acquired knowledge to gain more reward.

Increasing exploration may result in a broader set of valuable actions, while exploitation may result in a faster convergence to an optimal policy.

Regret refers to the difference between the effect of exploration and a normal greedy algorithm for selecting actions:

$$L_t = \mathbb{E}\left[\sum_{i=1}^{t} (max_{a \in A} Q(a)) - Q(a_i)\right]$$

Greedy policies have no exploration whereas **e-greedy policies** fall into the situation where the agent does not take advantage of the knowledge gained because the random actions cause non-optimal actions.Greedy and e-greedy policies both have linear regret functions.

One way of reducing policy regret is having a high amount of initial exploration and then reducing it to exploit knowledge by using a **decaying e-greedy policy**, which has logarithmic regret.

Example: Deep Q Network

Taking everything from this overview of RL, a DQN shows the implementation of these ideas.

This algorithm was invented in 2015 and has grown into many different variants of DQN.

It was the first that could master many Atari games without modification.

Can you identify the part of the algorithm that makes it easy to distribute, as compared to traditional deep learning?

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode $= 1, M$ **do**

Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

For $t = 1$,T **do**

With probability ε select a random action a_t

otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

Execute action a_t in emulator and observe reward r_t and image x_{t+1}

Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

Store transition $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in D

Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D

Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$

Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ

Every C steps reset $\hat{Q} = Q$

End For

End For

Example: Which parts of a DQN do you recognize?

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N — Episode History

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$ — Optimal Policy Function

For episode $= 1, M$ **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For $t = 1$,T **do**

    With probability ε select a random action a_t — e-Greedy Exploration/Exploitation

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D — Episode History Population

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$ — Exploration/Exploitation

    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q} = Q$ — Update Optimal Policy Function

  End For

End For

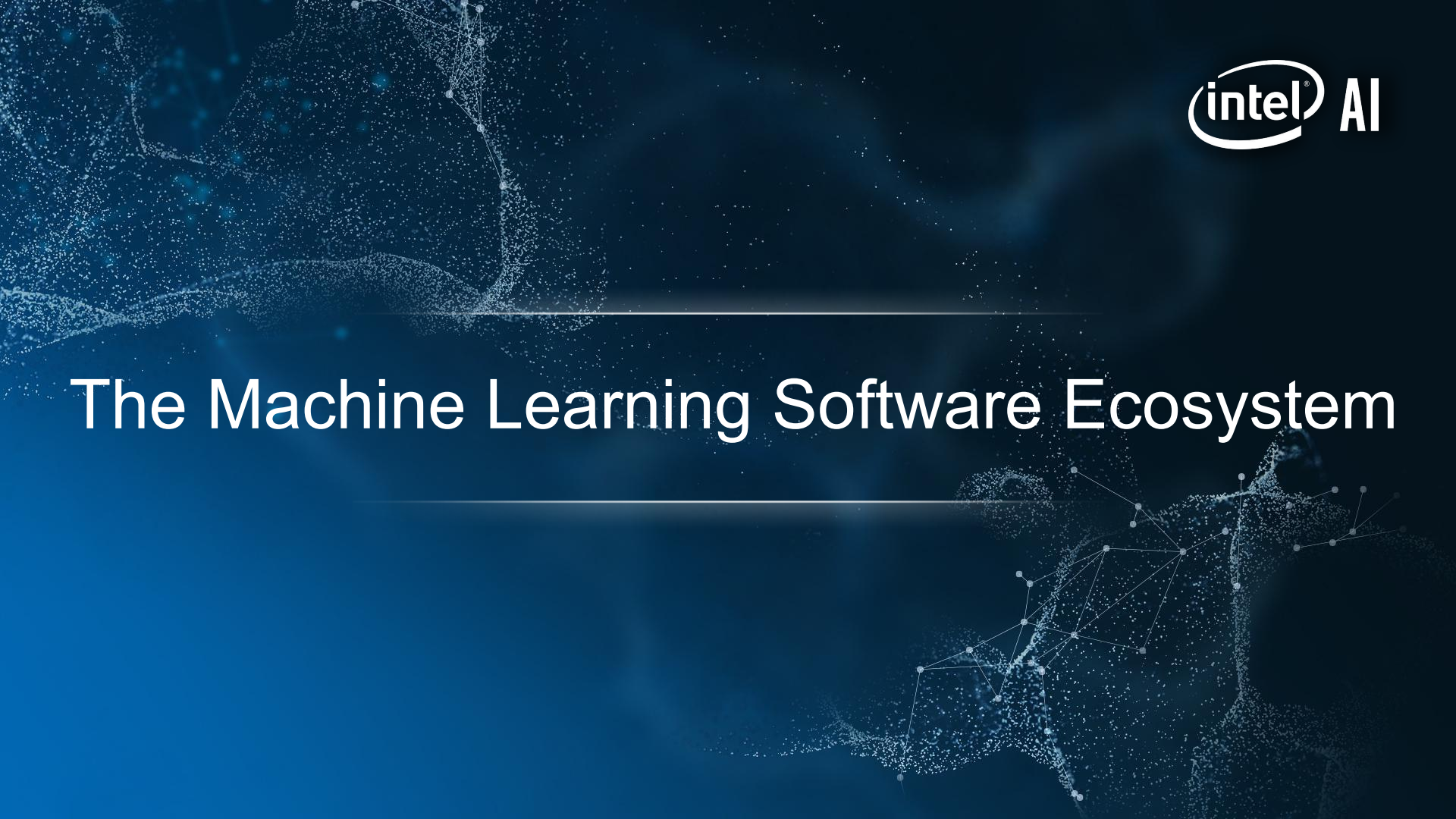
How is Reinforcement Learning Different?

Supervised Learning

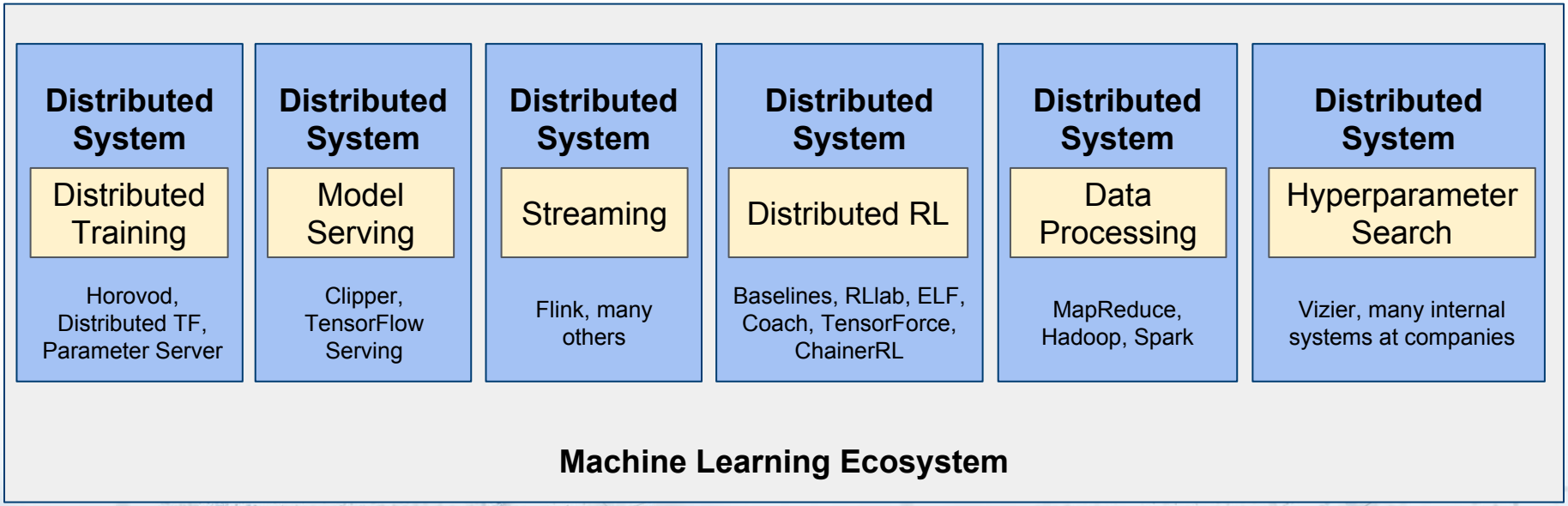
- Learning from a dataset
- Is not always sequentially based
- Training first, then inference
- Classification
- Cannot be easily distributed
- Feedback is instantaneous
- Inference does not affect data

Reinforcement Learning

- Learning from a reward signal
- Is almost always sequentially based
- Training and inference simultaneously
- Chooses an action
- Easily distributed
- Feedback is delayed
- Inference does affect data


intel AI
The Machine Learning Software Ecosystem

The Machine Learning Ecosystem



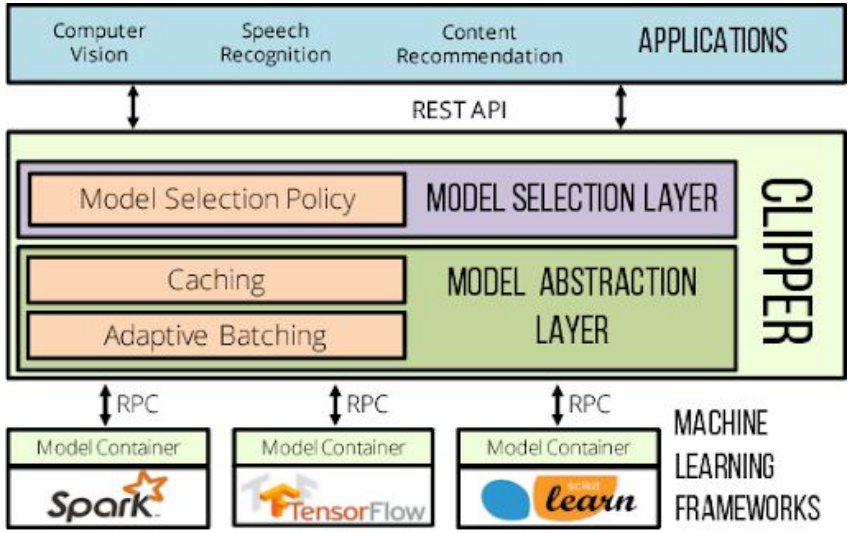
Machine Learning Ecosystem: ML Model Serving

Distributed systems are also required for the other half of machine learning-inference.

For an application that uses AI, any number of clients must be able to submit data to be run by the model.

Model serving frameworks (such as Clipper and TensorFlow Serving), bridge the gap between the application and the machine learning framework.

Distributed systems can greatly increase inference speed, even on single node.



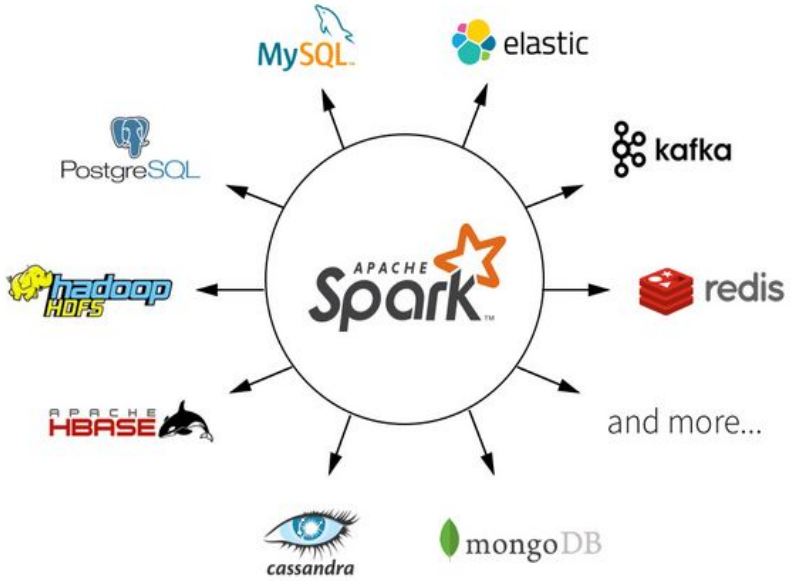
Machine Learning Ecosystem: Data Processing

The most obvious advantage to using distributed systems with ML is the benefits of networked computation:

- Data can be retrieved from any number of remote nodes, which allows for very large datasets.
- Multiple compute nodes allow for distributed training and inference engines.

Existing frameworks include famous examples, such as:

- Hadoop
- Spark
- MapReduce



Machine Learning Ecosystem: Hyperparameter Search

One of the main bottlenecks in creating ML systems is the amount of time required to fine tune an algorithm.

As an example, most computer vision deep learning systems have at least the following layers and their associated hyperparameters:

- 3 Conv. layers (stride, filter size, dropout)
- 3 Pooling layers (filter size)
- 2 Dense layers (width, dropout)
- Output layer

Even with the simplest optimization technique (SGD), there is still one hyperparameter- the learning rate.

So even for the simplest CNN, there can still be around 17 basic hyperparameters. This does not count the fine tuning that can be done with more technical modifications of each layer.

Another area in which hyperparameters must be chosen involve the system itself to achieve better performance:

- ML framework version
- OMPNUMTHREADS, the number of OpenMP threads
- Intraop threads, the number of CPU threads
- Interop threads, the number of parallel threads by the operator.
- *The interop and intraop variables should not be set to maximum and the best configuration has to be found for a particular workload to maximize performance.

These large numbers of hyperparameters alongside the large amount of time it takes to run a single test means that these tests are expensive. To speed this up, hyperparameter search distributively spreads this across many nodes to run multiple tests simultaneously.

Machine Learning Ecosystem: Federated Learning

One of the largest obstacles to machine learning applications is privacy concerns about personal data.

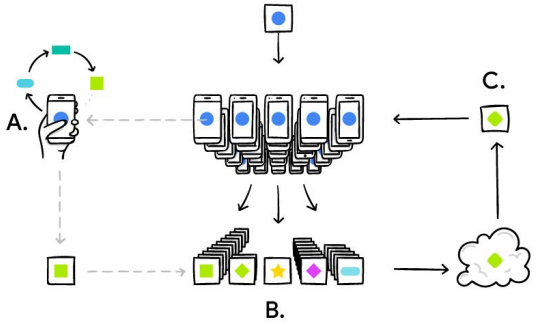
This may include information such as:

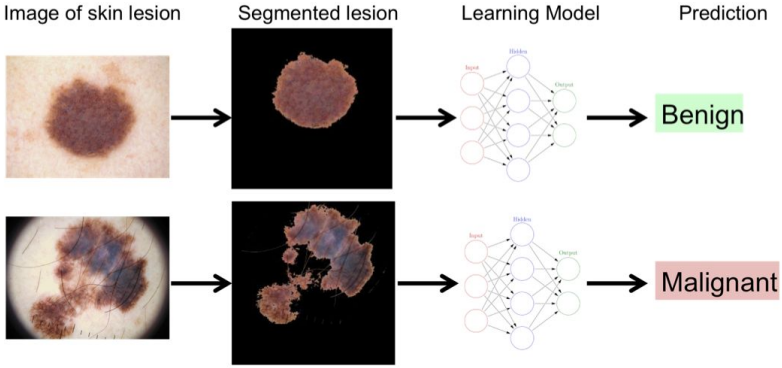
- Social media data
- Medical records
- Phone data (Geolocation, call history, etc.)

The way to get around some of these security laws is to not let the training data leave the server that it is stored on, but rather to export the model to those servers.

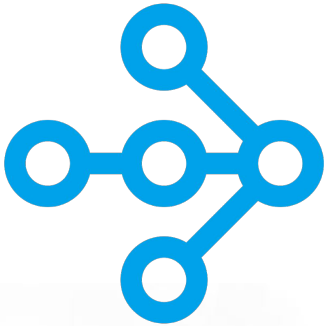
This is not an issue when the entire dataset is on the remote server but it becomes a distributed learning task when the complete dataset is spread across many nodes.

This also reduces cloud storage costs by keeping the data local as opposed to storing it all in a central location.



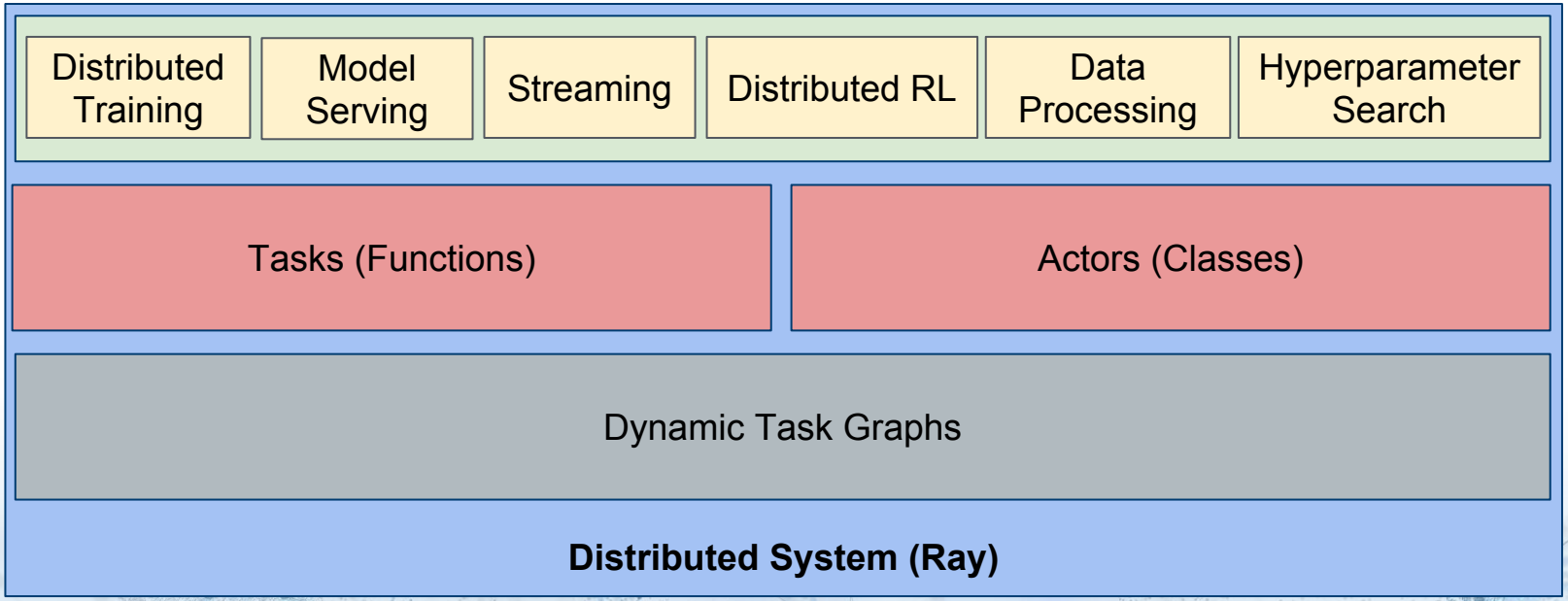


riselab
UC Berkeley



RAY

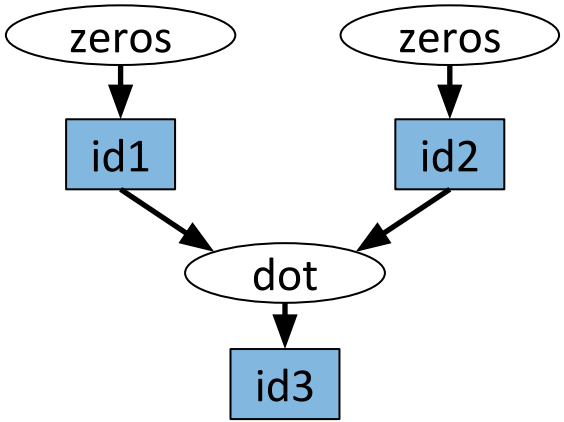
What is Ray? A unified machine learning ecosystem.



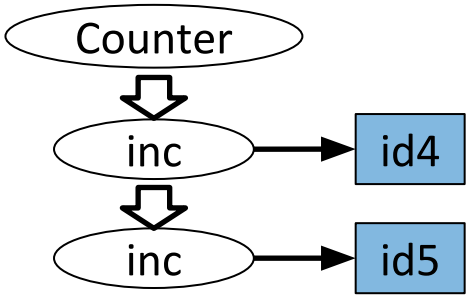
Ray API

Tasks

```
@ray.remote
def zeros(shape):
    return np.zeros(shape)

@ray.remote
def dot(a, b):
  return np.dot(a, b)

id1 = zeros.remote([5, 5])
id2 = zeros.remote([5, 5])
id3 = dot.remote(id1, id2)
ray.get(id3)
```



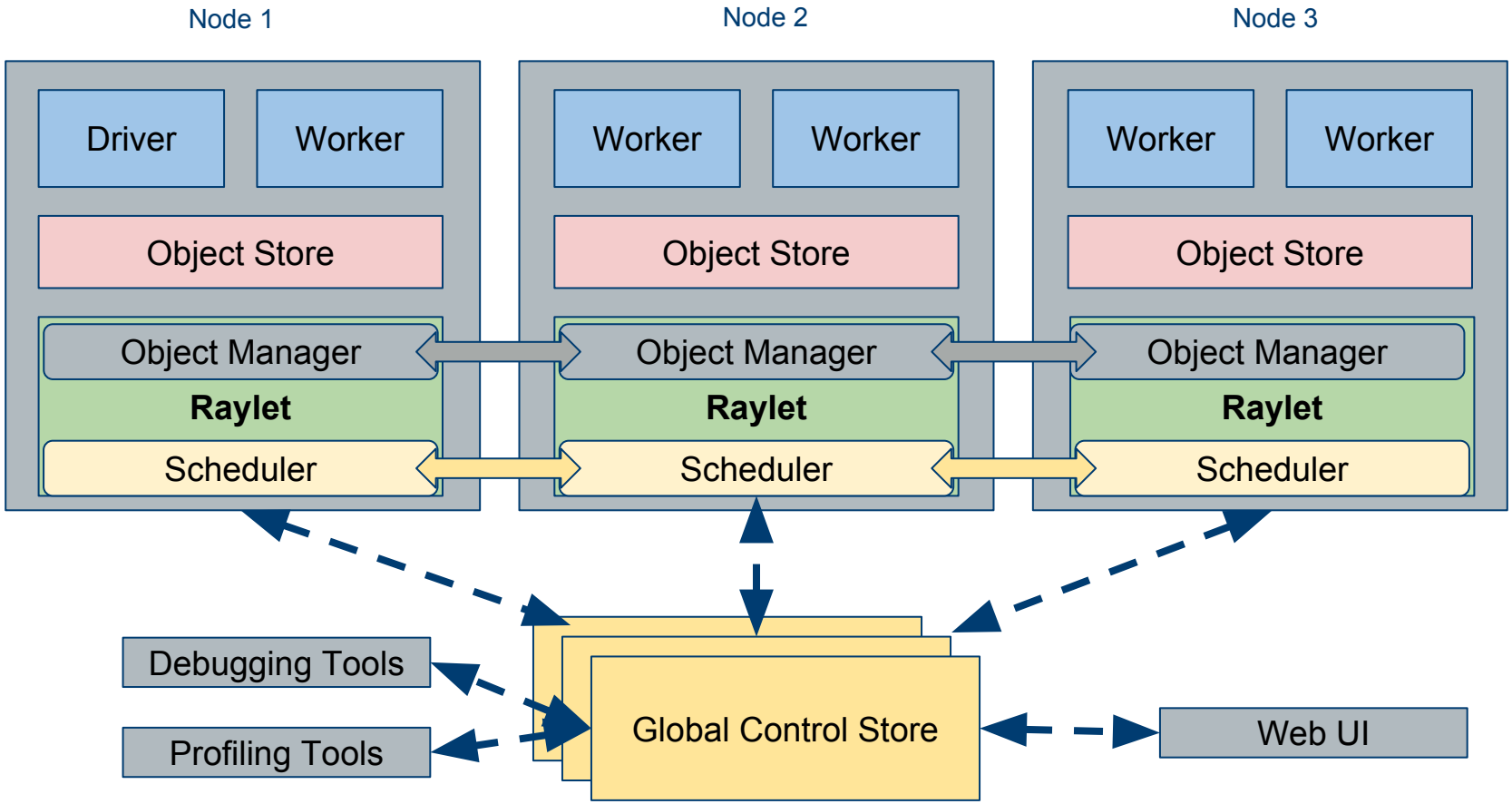
Ray API



Actors

```
@ray.remote(num_cpus=1)
class Counter(object):
    def __init__(self):
        self.value = 0
    def inc(self):
        self.value += 1
        return self.value

c = Counter.remote()
id4 = c.inc.remote()
id5 = c.inc.remote()
ray.get([id4, id5])
```

Ray Architecture



Ray: Local vs. Cluster

Local

```
@ray.remote
def task(args):
    # Task goes here

ray.init()
id1 = dot.remote(args)
ray.get(id1)

>>> python ray_script.py
```

Cluster

```
@ray.remote
def task(args):
    # Task goes here

ray.init(redis_address=<ADDRESS>)
id1 = dot.remote(args)
ray.get(id1)

>>> ray start --head
(worker device) >>> ray start
   --redis-address <ADDRESS>
>>> python ray_script.py
```

Advantages of Ray

Most distributed Python frameworks use the common threading format:

```
t = threading.Thread(target=f,args=())
t.start()
```

While this does utilize common multiprocessing techniques such as Semaphores or nested parallelism, this is not friendly for transitioning between distributed and serial.

Ray easily provides this transition with ray.get and object ids. Typically, the only required changes is to add ray.get and f.remote() instead of having to create any complicated feedback process.

Less issues with GIL compatibility with optimized frameworks such as Numpy.

Tricks to Use While Using Ray.

When a function is wrapped with the ray.remote decorator, the dependencies are automatically found with the cloudpickle library since the function is serialized to be executed remotely.

This means that users must be very careful about transferring arguments that are large datasets since these must be stored and shared globally, decreasing performance and increasing resource consumption.

TensorFlow sessions cannot be serialized directly. Often the ways to implement distributed session involve many sesion working simultaneously. As a result, it is easier to work with an API for TensorFlow such as Keras instead.

Simple example:

Parameter server in 10 lines of code:

```
@ray.remote
class ParameterServer(object):
    def __init__(self, keys, values):
        values = [value.copy() for value in values]
        self.weights = dict(zip(keys, values))

    def push(self, keys, values):
        for key, value in zip(keys, values):
            self.weights[key] += value

    def pull(self, keys):
        return [self.weights[key] for key in keys]
```

Personal Use Cases of Ray

Personal use cases of Ray framework:

TensorQTL-Fast: Genomics mapping from MIT and Harvard's Broad Institute was found to be 2.5x the speed of the default version by distributing.

Coach-Ray: Using Ray to distribute RL Coach instead of distributed TensorFlow or Kubernetes.

Simple inference engine that can speed up 8x on single node just by utilizing low CPU utilization per process to use more of a CPU through multiprocessing.


intel AI
Reinforcement Learning Frameworks

Reinforcement Learning Frameworks: Common Elements

Most reinforcement learning frameworks can be organized into the following groups:

Algorithm based:

- Provides a library of RL agents
- Terminal and/or Python API based.
- Integrates with an environment framework instead of building environments.
- Allows for customization of agent configurations.
- Has scaffolding systems for new agents.

Environment based:

- Generally focused on a single or subset of games or tasks.
- Could also be a plugin into a larger simulation system.
- Bare minimum requires control input, sensor output, and reward system, but many come with many sophisticated simulation data.
- Most popular is OpenAI Gym.

Reinforcement Learning Frameworks: Intel RL Coach

Comes with a very large selection of agents and envs:

- Gym, ViZDoom, Roboschool, Carla, PySC2, DeepMind Control Suite, Gym Extensions
- 32 learning algorithms

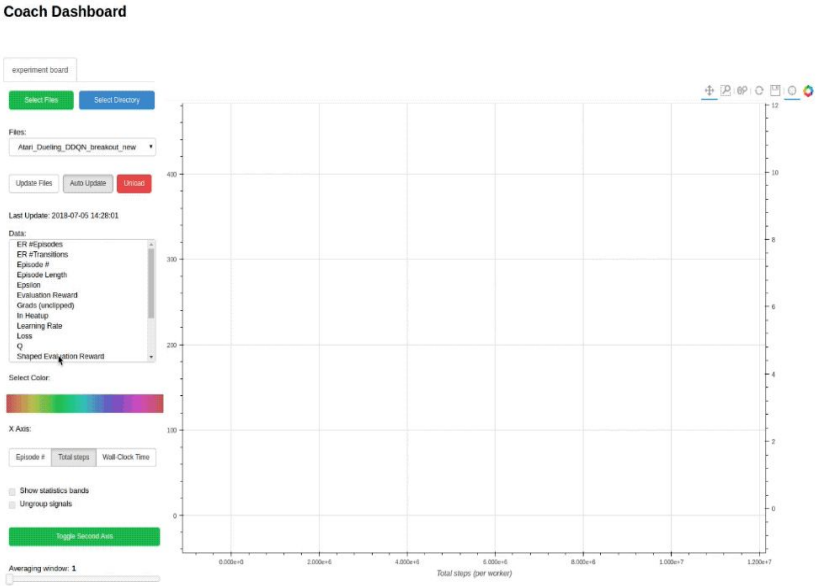
Terminal based.

Supports imitation learning.

Single-node, distributed, and multi-node using Kubernetes, Redis (memory backend), S3 and NFS (data storage), and Docker containers.

Switch between deep learning frameworks.

It has a Dashboard visualization application and can output results of training through gif files.



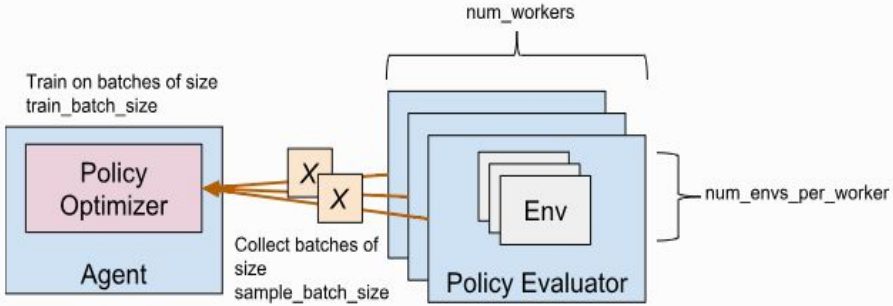
Reinforcement Learning Frameworks: Ray RLlib

Python API and terminal commands with config settings from argv or yaml files.

Easily customizable for new agents or environments.

Comes with 16 agents and an interface to Gym.

Outputs training visualization to TensorBoard.




```python
import ray
import ray.rllib.agents.ppo as ppo
from ray.tune.logger import pretty_print

ray.init()
config = ppo.DEFAULT_CONFIG.copy()
config["num_gpus"] = 0
config["num_workers"] = 1
agent = ppo.PPOAgent(config=config, env="CartPole-v0")

# Can optionally call agent.restore(path) to load a checkpoint.

for i in range(1000):
   # Perform one iteration of training the policy with PPO
   result = agent.train()
   print(pretty_print(result))

   if i % 100 == 0:
       checkpoint = agent.save()
       print("checkpoint saved at", checkpoint)
```

Reinforcement Learning Frameworks: Ray RLlib and Tune

RLlib also supports integration with Tune, a hyperparameter search framework.

This allows training tests to be run under different configuration settings using the Python API.

Supports many types of deep learning frameworks such as PyTorch, TensorFlow, and Keras.

There are many options for model search techniques and optimizations approaches.

Results of Tune can be visualized with TensorBoard, Plot.ly, and VisKit.

Can be run on the laptop to the cluster without changing any of the code.

```
import ray
import ray.tune as tune

ray.init()
tune.run_experiments({
    "my_experiment": {
        "run": "PPO",
        "env": "CartPole-v0",
        "stop": {"episode_reward_mean": 200},
        "config": {
            "num_gpus": 0,
            "num_workers": 1,
            "lr": tune.grid_search([0.01, 0.001, 0.0001]),
        },
    },
})
```

Reinforcement Learning Frameworks: Google Dopamine

Made for high flexibility for research implementations, not a collection of industry ready agents and examples.

Uses the Google gin framework to configure agents.

Includes professional implementations of DQN and Rainbow agents.

Training results can be visualized with TensorBoard.

Relatively small framework in terms of number of examples and number of contributors. The framework is also somewhat new, opened in April 2018. The high star count is most likely due to it being from Google.

```
class MyRandomDQNAgent(dqn_agent.DQNAgent):
  def __init__(self, sess, num_actions):
    """This maintains all the DQN default argument values."""
    super(MyRandomDQNAgent, self).__init__(sess, num_actions)

  def step(self, reward, observation):
    """Calls the step function of the parent class, but returns a random action.
    """
    _ = super(MyRandomDQNAgent, self).step(reward, observation)
    return np.random.randint(self.num_actions)

def create_random_dqn_agent(sess, environment, summary_writer=None):
  """The Runner class will expect a function of this type to create an agent."""
  return MyRandomDQNAgent(sess, num_actions=environment.action_space.n)

random_dqn_config = """
import dopamine.discrete_domains.atari_lib
import dopamine.discrete_domains.run_experiment
atari_lib.create_atari_environment.game_name = '{}'
atari_lib.create_atari_environment.sticky_actions = True
run_experiment.Runner.num_iterations = 200
run_experiment.Runner.training_steps = 10
run_experiment.Runner.max_steps_per_episode = 100
""".format(GAME)
gin.parse_config(random_dqn_config, skip_unknown=False)

# Create the runner class with this agent. We use very small numbers of steps
# to terminate quickly, as this is mostly meant for demonstrating how one can
# use the framework.
random_dqn_runner = run_experiment.TrainRunner(LOG_PATH, create_random_dqn_agent)
```

Watch 449 | Unstar 7,541 | Fork 927

Reinforcement Learning Frameworks: Deepmind TRFL

TRFL (“Truffle”) is a collection of building blocks to create RL systems out of TensorFlow.

It is not a collection of RL agents but rather an extension to TensorFlow designed to accommodate the operations required by RL.

This is not to be confused with Sonnet, which is DeepMind’s deep learning framework which is also built on top of TensorFlow, meant for accommodating the construction of more complex super-networks.

```
import tensorflow as tf
import trfl

# Q-values for the previous and next timesteps, shape [batch_size, num_actions].
q_tm1 = tf.get_variable(
    "q_tm1", initializer=[[1., 1., 0.], [1., 2., 0.]], dtype=tf.float32)
q_t = tf.get_variable(
    "q_t", initializer=[[0., 1., 0.], [1., 2., 0.]], dtype=tf.float32)

# Action indices, discounts and rewards, shape [batch_size].
a_tm1 = tf.constant([0, 1], dtype=tf.int32)
r_t = tf.constant([1, 1], dtype=tf.float32)
pcont_t = tf.constant([0, 1], dtype=tf.float32)  # the discount factor

# Q-learning loss, and auxiliary data.
loss, q_learning = trfl.qlearning(q_tm1, a_tm1, r_t, pcont_t, q_t)
```

Watch 194 | Star 2,396 | Fork 279

Reinforcement Learning Frameworks: Tensorforce

A modular RL framework also built on top of TensorFlow for fast prototyping.

The configuration settings for each agent are modified in Python, instead of some of the other configuration techniques used by major frameworks.

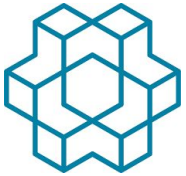
Has seven pre-built agents and the ability to create custom agents.

Can be run in Gym, Universe, Lab, and Unreal Engine 4 environments.

Includes several preprocessing functions to experiment with how to manipulate input data.

```python
from tensorforce.agents import PPOAgent

# Instantiate a Tensorforce agent
agent = PPOAgent(
    states=dict(type='float', shape=(10,)),
    actions=dict(type='int', num_values=5),
    network=[
        dict(type='dense', size=64),
        dict(type='dense', size=64)
    ],
    step_optimizer=dict(type='adam', learning_rate=1e-4)
)

# Initialize the agent
agent.initialize()

# Retrieve the latest (observable) environment state
state = get_current_state()  # (float array of shape [10])

# Query the agent for its action decision
action = agent.act(states=state)  # (scalar between 0 and 4)

# Execute the decision and retrieve the current performance score
reward = execute_decision(action)  # (any scalar float)

# Pass feedback about performance (and termination) to the agent
agent.observe(reward=reward, terminal=False)
```

Reinforcement Learning Frameworks: Environments: Gym

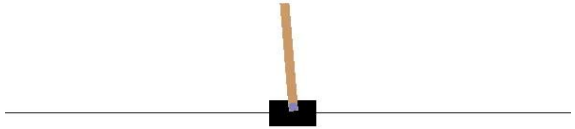


Gym is the quintessential library to learn for RL and is used by almost every RL agent framework.

Created by OpenAI, it includes a very large selection of environments that have a simple interface with a Python API.

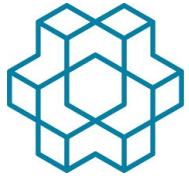
Almost all can be installed with a simple pip install with the exception of MuJoCo, which requires registering with the physics engine itself before being integrated with Gym.

OpenAI also has Gym Retro (previously Universe) which makes video games into Gym environments as well as Roboschool for robotic simulations.



```
import gym
env = gym.make('CartPole-v0')
env.reset()
for _ in range(1000):
    env.render()
    # take a random action
    env.step(env.action_space.sample())
```

Reinforcement Learning Frameworks: Environments: Gym



Classic control

Control theory problems from the classic RL literature.

Acrobot-v1
Swing up a two-link robot.

CartPole-v1
Balance a pole on a cart.

MountainCar-v0
Drive up a big hill.

MountainCarContinuous-v0
Drive up a big hill with continuous control.

Pendulum-v0
Swing up a pendulum.

Box2D

Continuous control tasks in the Box2D simulator.

BipedalWalker-v2
Train a bipedal robot to walk.

BipedalWalkerHardcore-v2
Train a bipedal robot to walk over rough terrain.

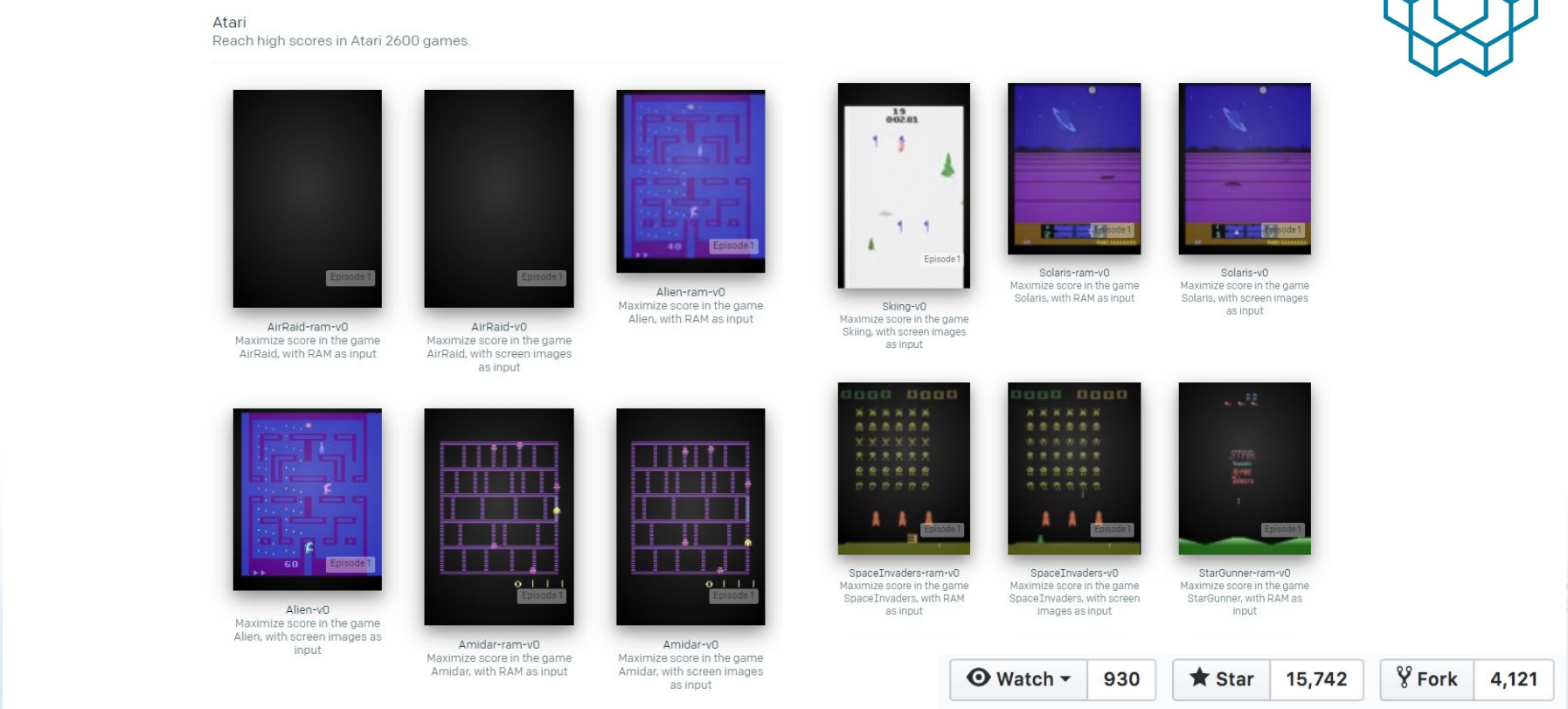
CarRacing-v0
Race a car around a track.

LunarLander-v2
Navigate a lander to its landing pad.

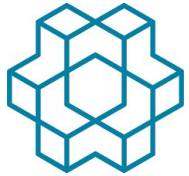
LunarLanderContinuous-v2
Navigate a lander to its landing pad.

Watch 930 | Star 15,742 | Fork 4,121

Reinforcement Learning Frameworks: Environments: Gym

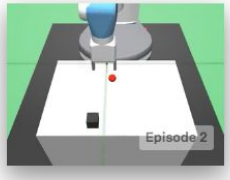


Reinforcement Learning Frameworks: Environments: Gym

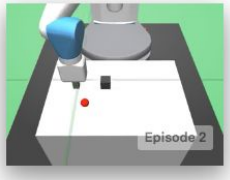


Robotics

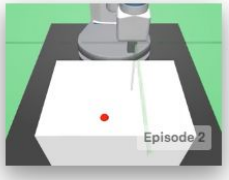
Simulated goal-based tasks for the Fetch and ShadowHand robots.



FetchPickAndPlace-v0
Lift a block into the air.

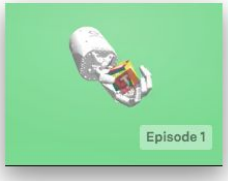


FetchPush-v0
Push a block to a goal position.

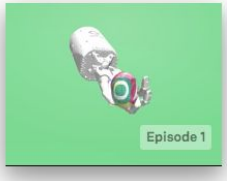


FetchReach-v0
Move Fetch to a goal position.

FetchSlide-v0
Slide a puck to a goal position.



HandManipulateBlock-v0
Orient a block using a robot hand.

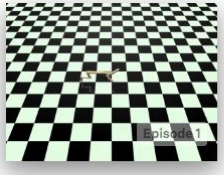


HandManipulateEgg-v0
Orient an egg using a robot hand.

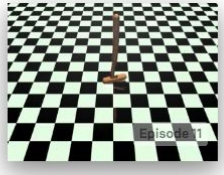
MuJoCo

Continuous control tasks, running in a fast physics simulator.

Ant-v2
Make a 3D four-legged robot walk.



HalfCheetah-v2
Make a 2D cheetah robot run.



Hopper-v2
Make a 2D robot hop.

Humanoid-v2
Make a 3D two-legged robot walk.

HumanoidStandup-v2
Make a 3D two-legged robot standup.

InvertedDoublePendulum-v2
Balance a pole on a pole on a cart.

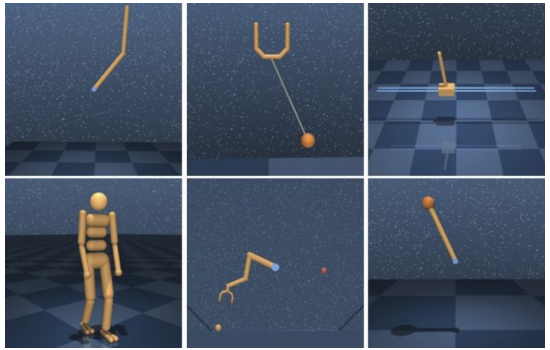
Watch 930 | Star 15,742 | Fork 4,121

Reinforcement Learning Frameworks: Environments: DeepMind Lab / Control Suite / PySC2

Lab

3D navigation and puzzle solving tasks with a simple Python API.

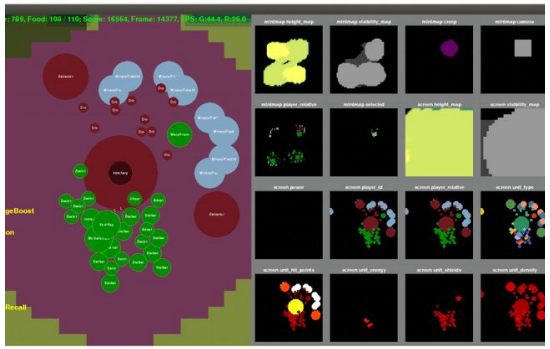
Watch 516 | Star 5,671 | Fork 1,122



Control Suite

Python interface to the powerful MuJoCo physics engine for RL dynamics.

Watch 97 | Star 1,390 | Fork 196



PySC2

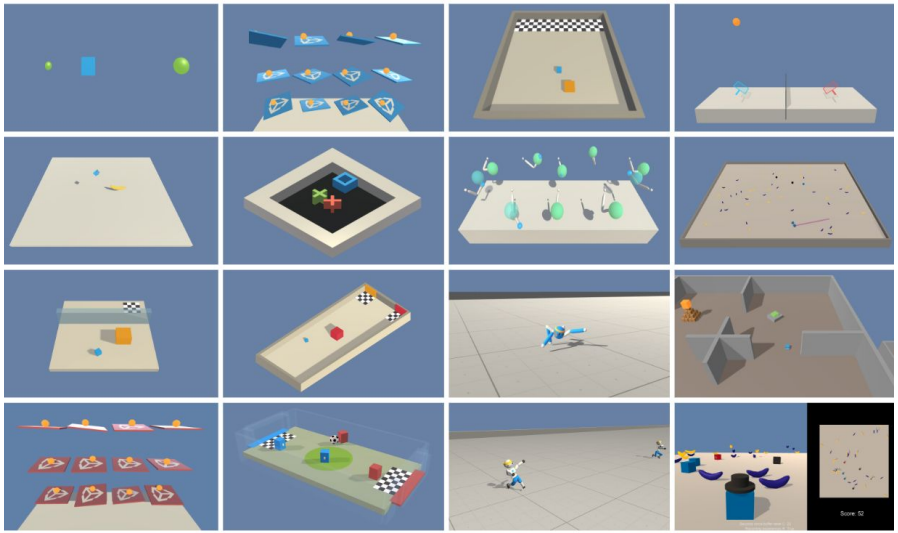
Collaboration with Blizzard to create a Python API for StarCraft II.

Watch 377 | Star 5,804 | Fork 837

Reinforcement Learning Frameworks: Environments: Unity ML-Agents Toolkit

Unity is one of most popular real-time game engine (creates half of the world's games).

- Python API
- Supports multiple environment configs
- Run RL agents in custom games
- Comes with sample environments
- Integrates with TensorFlow
- Can be used for:
  - Reinforcement Learning
  - Imitation Learning
  - Neuroevolution
- Can be set up with Docker



Watch 465 Star 5,270 Fork 1,306

Reinforcement Learning Frameworks: Environments: ViZDoom

One of the major first-person game environments for RL.

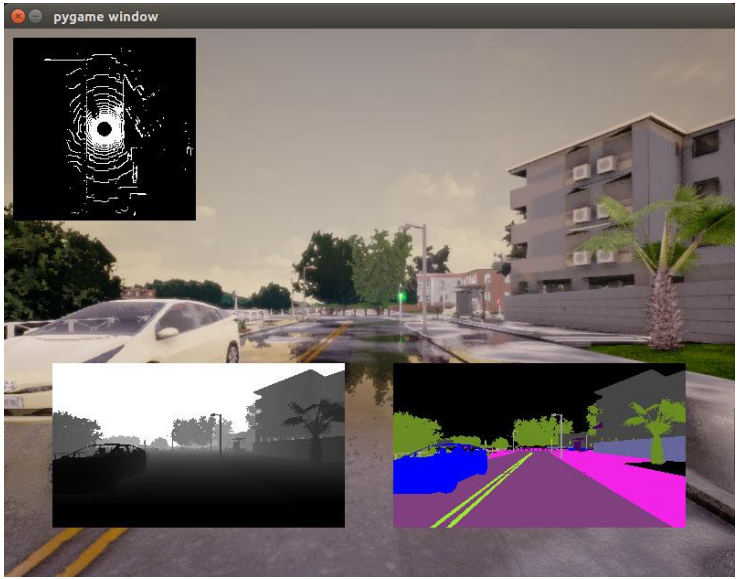
- Multi-platform
- API for C++, Lua, Python, and Julia
- Supports custom scenarios
- Async and sync player
- Allows multi-player mode
- 7000 fps (very fast for RL environment)
- Comes with 3D depth vision and regional classifier

Watch 50 Star 840 Fork 219

Reinforcement Learning Frameworks: Environments: CARLA

High quality automated driving simulator:

- Supports ROS
- User editable map
- High quality graphics for real world use
- Users can control other vehicles and pedestrians
- Rendering can be turned off for simulations
- Comes with simulated RADAR, depth mapping, and classifier segmentation sensor data.
- Has RL and imitation learning examples available.



Watch 169 Star 2,290 Fork 568

Reinforcement Learning Frameworks: Frameworks by the Numbers

Algorithm based:

	Watch		Unstar / Star		Fork	
Dopamine:	Watch	449	Unstar	7,541	Fork	927
RLlib/Tune:	Watch	283	Unstar	6,114	Fork	833
TRFL:	Watch	194	Star	2,396	Fork	279
TForce:	Watch	145	Star	2,213	Fork	388
Coach:	Watch	103	Star	1,158	Fork	209

Environment based:

	Watch		Star		Fork	
Gym:	Watch	930	Star	15,742	Fork	4,121
PySC2:	Watch	377	Star	5,804	Fork	837
Lab:	Watch	516	Star	5,671	Fork	1,122
Unity ML:	Watch	465	Star	5,270	Fork	1,306
CARLA:	Watch	169	Star	2,290	Fork	568
DeepMind CS:	Watch	97	Star	1,390	Fork	196
ViZDoom:	Watch	50	Star	840	Fork	219

So which framework to use?

The frameworks cannot be judged simply from the GitHub stats:

- Some are from large companies that already have many followers and press releases (Google).
- Some are from smaller accounts or are completely community based.
- Each are for different levels of developers and offer a different set of tools.

Which frameworks do researchers need to know?

While OpenAI's Gym is clearly an industry favorite, the success of the other environment frameworks is because they offer different functionality. Gym however, is found in almost all RL algorithm frameworks.

The RL algorithm frameworks do not however, have a clear favorite. It is useful to be familiar with at least one or two of the frameworks to get familiar with the advantages of different types of configuration techniques and hardware accelerator options.

Intel in AI

Why use Intel for AI?

According to public sales announcements, Intel is one of the world's biggest AI companies, selling more than $1B in AI hardware in just the past year, almost twice than the $606M reported by NVIDIA®.

Intel® joined the AI race and has invested in:

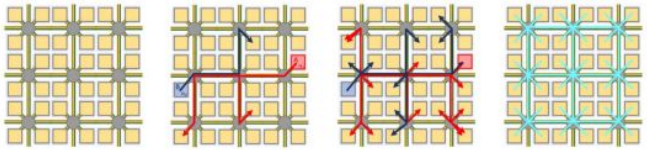
- Automated Driving (Mobileye)
- Robotics
- Computer Vision (VPUs, OpenVINO)

Intel holds the current records in performance for:

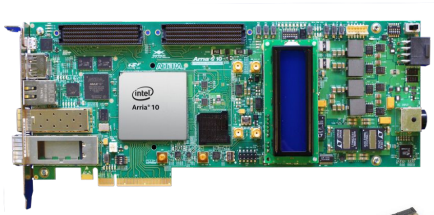
- 3 of the top 5 ImageNet Inference records (Stanford DAWN Bench)
- Reinforcement Learning (MLPerf), 13x the speed of a P100 GPU

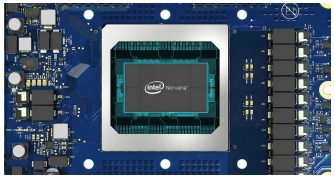
Intel is the only company that uses AI on such a large variety of experimental hardware:

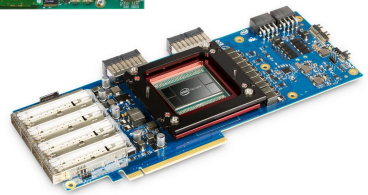
- FPGAs
- Movidius VPUs
- Neural Network Processors (NNP)
- Quantum Computers
- Neuromorphic Computers

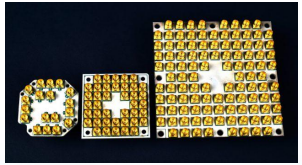


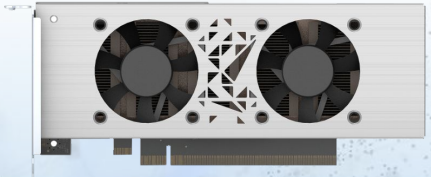
How much AI hardware does Intel make? 120+ types of CPUs, FPGA, VPUs, etc.

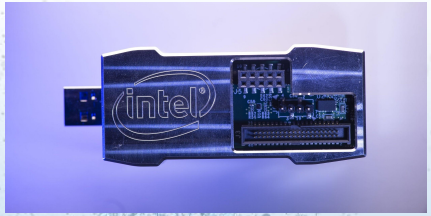




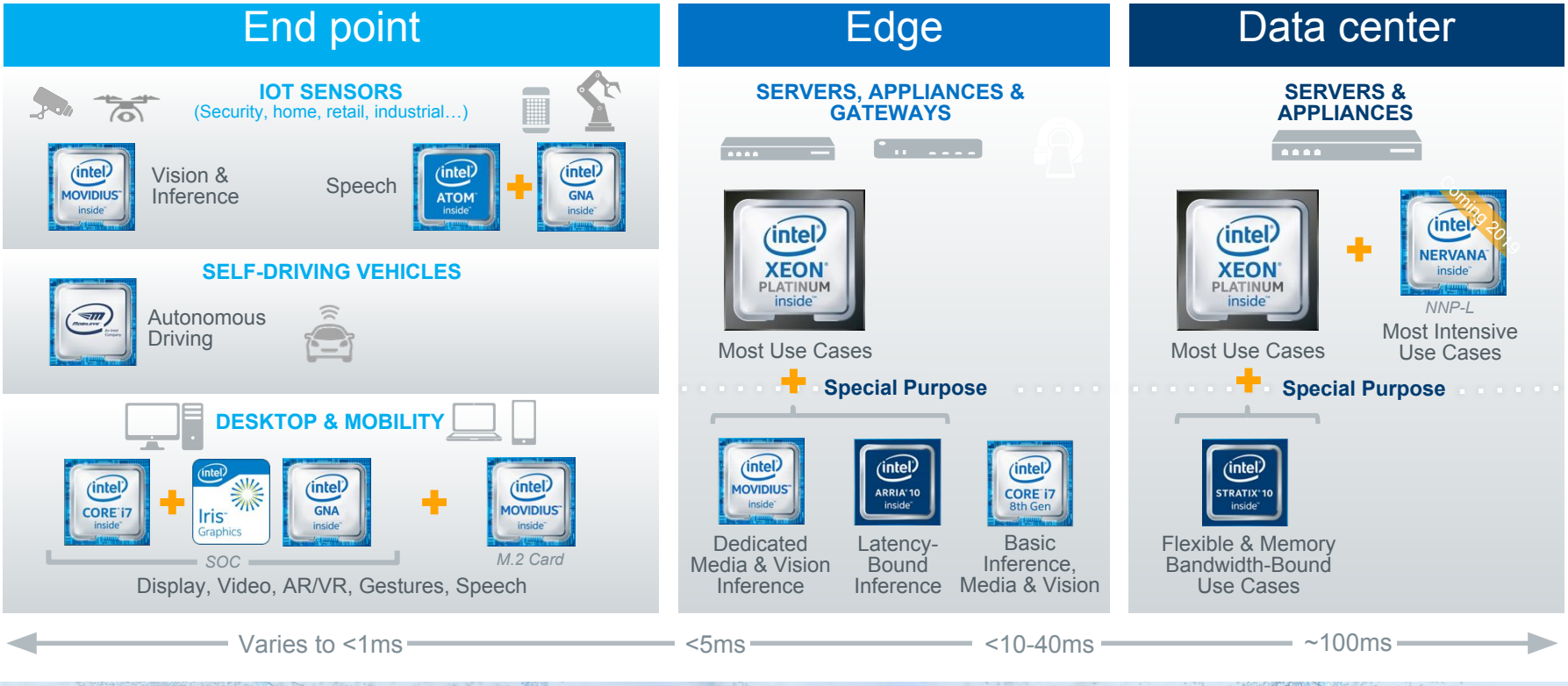








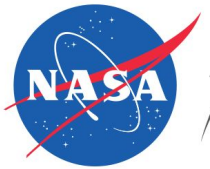
What hardware do I use for which part of an AI application?



AI Partners:

Tencent YouTu Lab

Google

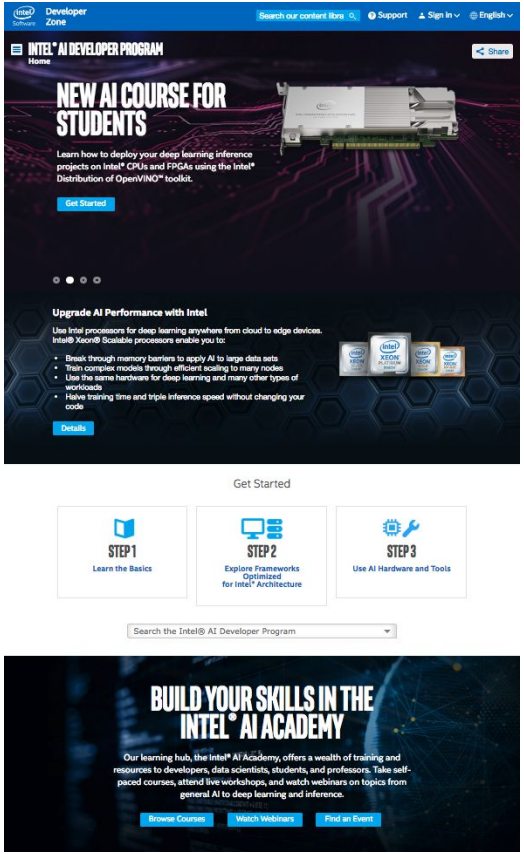


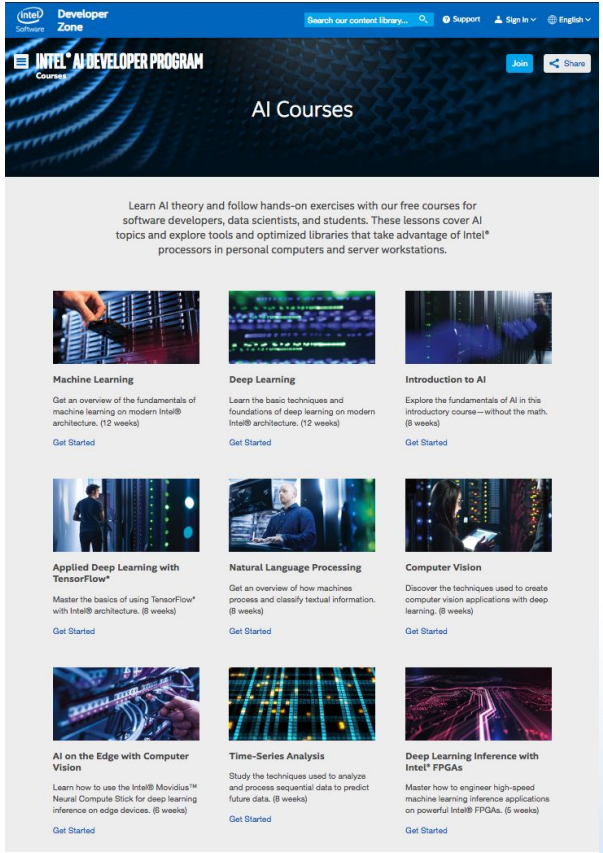
Lenovo

How Intel can help students:

Student Resources

- Free AI courses
  - Including courses on Ray and RL
- Free access to the AI DevCloud:
  - 24 core Xeon Scalable CPUs
  - 96 GB of RAM and 200 GB of storage
  - Each user can use 5 nodes at a time
- Student Ambassador Program
- Research Grants
- Internships and AI Fellowships
  - Intel Labs, AIPG, Quantum, Neuromorphic, etc..





Where can I learn more about AI or RL theory?

Including the free courses available through Intel, other resources that I recommend include:

- UCL's RL course by David Silver (summary of RL was taken from this material)
- Stanford's online lecture notes for CS231n on Convolutional Neural Networks
- *Deep Learning* by Ian Goodfellow, Yoshua Bengio, and Aaron Courville

Where can I get started to develop these types of research or applications?

The Intel Developer Program supports research grants as well as free cloud compute resources.

Learn more about Ray through the course *Distributed AI using the Ray Framework* offered through Intel.

As always, the best way to learn is to do. So start with hackathons, competitions, group projects, get involved with research groups even if you're an undergrad, and of course, paper implementations.

To find a copy of these lecture slides, search for RL_Ray_Lecture on GitHub.

Questions

Legal Notices and Disclaimers

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit intel.com/performance.

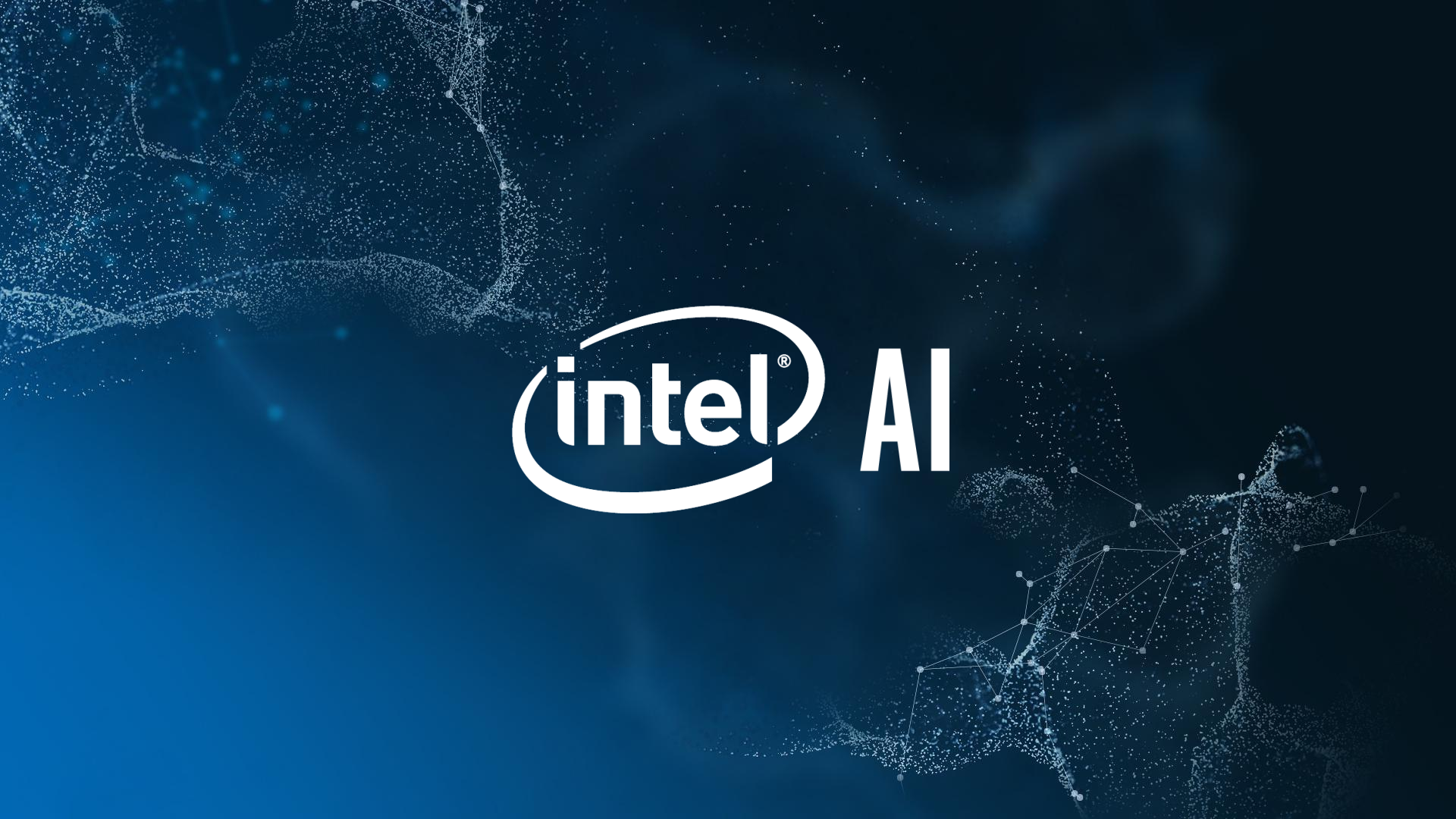
Intel does not control or audit the design or implementation of third-party benchmark data or websites referenced in this document. Intel encourages all of its customers to visit the referenced websites or others where similar performance benchmark data are reported and confirm whether the referenced benchmark data are accurate and reflect performance of systems available for purchase.

Optimization notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software, or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at intel.com/benchmarks.

Intel, the Intel logo, Intel Inside, the Intel Inside logo, Intel Atom, Intel Core, Iris, Movidius, Myriad, Intel Nervana, OpenVINO, Intel Optane, Stratix, and Xeon are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.


intel
AI