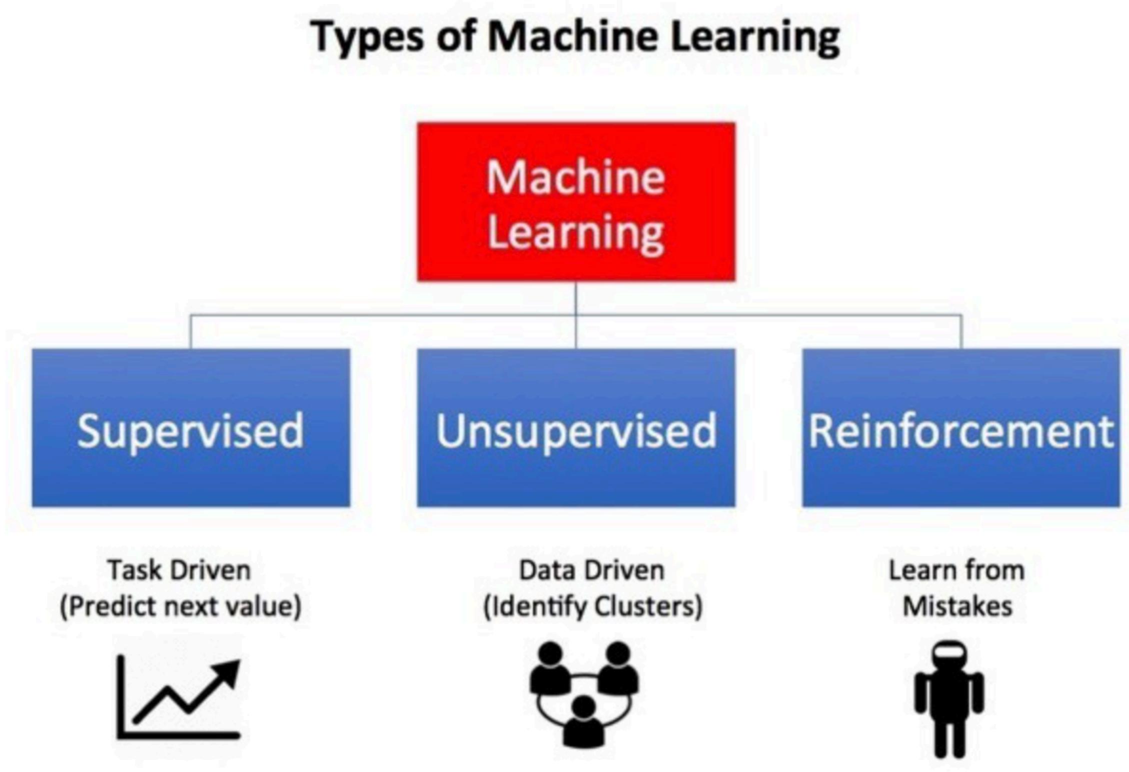
## Reinforcement Learning

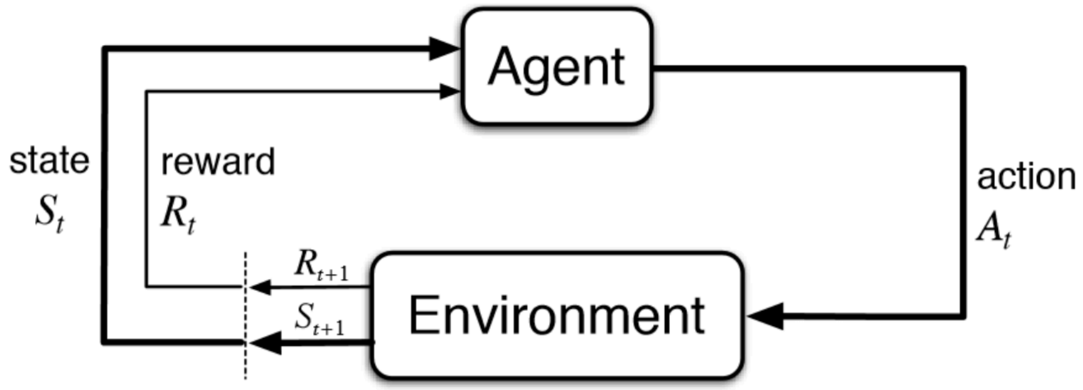
- Intro: ML- learn from data, RL- build your own data


Types of Machine Learning
Machine Learning
Supervised
Unsupervised
Reinforcement
Task Driven (Predict next value)
Data Driven (Identify Clusters)
Learn from Mistakes

**Reinforcement learning, a type of machine learning, in which agents take actions in an environment aimed at maximizing their cumulative rewards – NVIDIA**

Some key terms that describe the basic elements of an RL problem are

- **Agent.** The program you train, with the aim of doing a job you specify.
- **Environment.** The world, real or virtual, in which the agent performs actions.
- **Action.** A move made by the agent, which causes a status change in the environment.
- **Rewards.** The evaluation of an action, which can be positive or negative.



- Progress: What holds us back in AI
    1. Compute (the obvious one: Moore's Law, GPUs, ASICs),
    2. Data (in a nice form, not just out there somewhere on the internet - e.g. ImageNet),
    3. Algorithms (research and ideas, e.g. backprop, CNN, LSTM), and
    4. Infrastructure (software under you - Linux, TCP/IP, Git, ROS, PR2, AWS, AMT, TensorFlow, etc.).

- For RL no new algos since 1998 (Sutton). The 3 have improved.

## Supervised, Unsupervised, and Reinforcement Learning: What are the Differences?

## Difference #1: Static Vs.Dynamic

The goal of supervised and unsupervised learning is to search for and learn about patterns in training data, which is quite static.
RL, on the other hand, is about developing a policy that tells an agent which action to choose at each step — making it more dynamic.

## Difference #2: No Explicit Right Answer

In supervised learning, the right answer is given by the training data. In Reinforcement Learning, the right answer is not explicitly given: instead, the agent needs to learn by trial and error. The only reference is the reward it gets after taking an action, which tells the agent when it is making progress or when it has failed.

## Difference #3: RL Requires Exploration

A Reinforcement Learning agent needs to find the right balance between exploring the environment, looking for new ways to get rewards, and exploiting the reward sources it has already discovered. In contrast, supervised and unsupervised learning systems take the answer directly from training data without having to explore other answers.

## Difference #4: RL is a Multiple-Decision Process

Reinforcement Learning is a multiple-decision process: it forms a decision-making chain through the time required to finish a specific job. Conversely, supervised learning is a single-decision process: one instance, one prediction.

## OpenAI Gym

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents in everything from walking to playing games like Pong or Pinball.

OpenAI Gym gives us game environments in which our programs can take actions. Each environment has an initial status. After your agent takes an action, the status is updated.

When your agent observes the change, it uses the new status together with its policy to decide what move to make next. The policy is key: it is the essential element for your program to keep working on. The better the policy your agent learns, the better the performance you get out of it.

Think of policy as a mapping/ neural network. Input is state and output is an action.

### Policy gradients (PG) and DQN approaches to RL problems

Deep Q learning not so great. Paper from 2013. Even creators of DQN prefer PG.

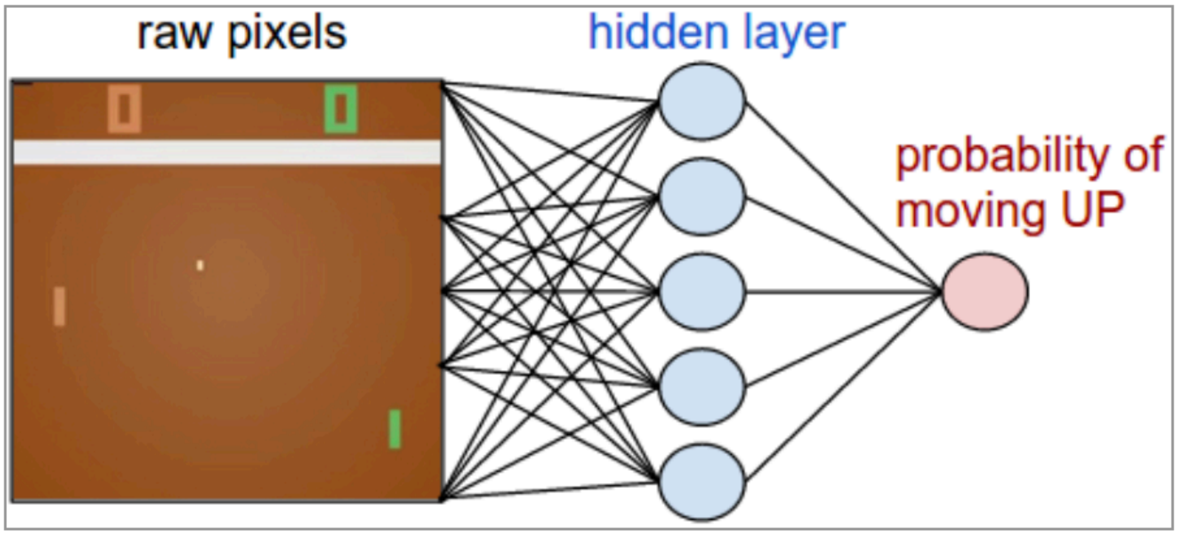
PG is preferred because it is end-to-end: there’s an explicit policy and a principled approach that directly optimizes the expected reward.

Atari game (pong) with PG in python.
Game: you play as one of the paddles (the other is controlled by a decent AI) and you have to bounce the ball past the other player.

Low level meaning:

1. We receive an image frame (a 210x160x3 byte array (integers from 0 to 255 giving pixel values)) and we get to decide if we want to move the paddle UP or DOWN (i.e. a binary choice).
2. After every single choice the game simulator executes the action and gives us a reward: Either a +1 reward if the ball went past the opponent, a -1 reward if we missed the ball, or 0 otherwise.
3. And of course, our goal is to move the paddle so that we get lots of reward.

Policy Network
*A policy network* implements our player (or "agent").
This network will take the state of the game and decide what we should do (move UP or DOWN)
Use a 2-layer neural network that takes the raw image pixels (100,800 numbers total (210*160*3)), and produces a single number indicating the probability of going UP
*Stochastic* policy, meaning that we only produce a *probability* of moving UP



Our policy network is a 2-layer fully-connected net.

Suppose we're given a vector x that holds the (preprocessed*) pixel information. We would compute:

```
h = np.dot(W1, x)      # compute hidden layer neuron activations
h[h<0] = 0       # ReLU nonlinearity: threshold at zero
logp = np.dot(W2, h)       # compute log probability of going up
p = 1.0 / (1.0 + np.exp(-logp))        # sigmoid function (gives probability of going up)
```

W1 and W2 are two matrices that we initialize randomly, so the agent will play very

badly at the start.
We use the *sigmoid* non-linearity at the end, which squashes the output probability to the range [0,1]
Intuitively, the neurons in the hidden layer can detect various game scenarios (e.g. the ball is in the top, and our paddle is in the middle), and the weights in W2 can then decide if in each case we should be going UP or DOWN
We have to find the W1 and W2 to make the agent an expert player.

*Ideally you'd want to feed at least 2 frames to the policy network so that it can detect motion. To make things a bit simpler do a tiny bit of preprocessing, e.g. we'll actually feed *difference frames* to the network (i.e. subtraction of current and last frame).

How this works

- Get 100,800 numbers (210*160*3) and give it to this PN which will have millions of parameters.

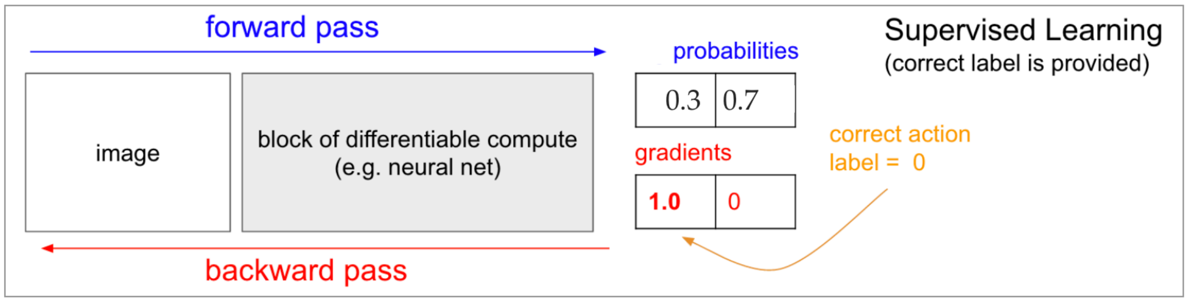
*Credit assignment problem*
Suppose we finally get a +1. That's great, but how can we tell what made that happen?
Was it something we did just now? Or maybe 76 frames ago? Or maybe it had something to do with frame 10 and then frame 90?
And how do we figure out which of the million knobs to change and how, in order to do better in the future?

(We get a +1 if the ball makes it past the opponent)

The *true* cause is that we happened to bounce the ball on a good trajectory, but in fact we did so many frames ago - e.g. maybe about 20 in case of Pong, and every single action we did afterwards had zero effect on whether or not we end up getting the reward.
In other words we're faced with a very difficult problem.

**How to solve this with Supervised Learning**
Feed an image to the network and get some probabilities. We would have access to a label- we might be told that the correct thing to do right now is to go UP (label 0).
Let's say the network currently predict 30% UP and 70% DOWN.
We know that the correct output is UP. So the output will be 1.

When we compute the gradients and back propagate to update the million parameters so that the network will be more likely to predict UP.



**How to solve this with Policy Gradients-** We do not have the correct label in the Reinforcement Learning.
Our policy network calculated probability of going UP as 30% and DOWN as 70%

We decide DOWN, and we will execute it in the game.

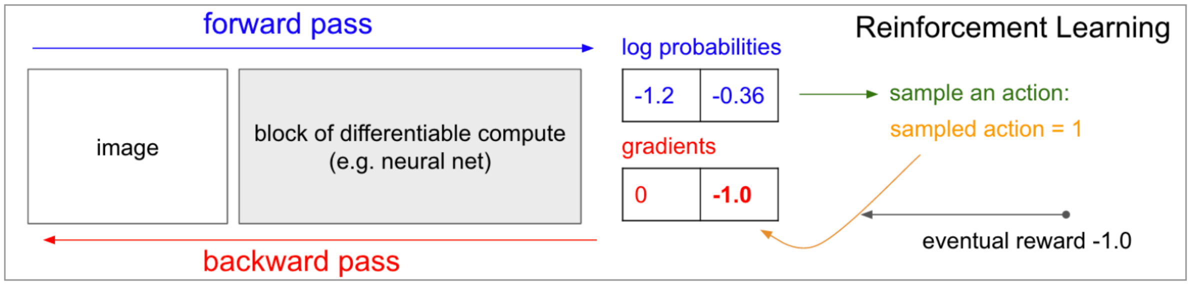
We could immediately fill in a gradient of 1.0 for DOWN as we did in supervised learning, and find the gradient vector that would encourage the network to be slightly more likely to do the DOWN action in the future.

But the problem is that at least for now we do not yet know if going DOWN is good.

But the point is that that's okay, because we can simply wait a bit and see.

In Pong we could wait until the end of the game, then take the reward we get (either +1 if we won or -1 if we lost), and enter that scalar as the gradient for the action we have taken.

In the example below, going DOWN ended up to us losing the game (-1 reward). So if we fill in -1 for log probability of DOWN and do backprop we will find a gradient that *discourages* the network to take the DOWN action for that input in the future (and rightly so, since taking that action led to us losing the game).



What this does:

1. Samples actions i.e. decide on UP/ DOWN from the output probabilities
2. Actions taken that lead to good outcomes get encouraged in the future
3. And actions taken that lead to bad outcomes get discouraged

Training Protocol:
-We will initialize the policy network with some W1, W2 and play 100 games of Pong (we call these policy “rollouts”).
-Lets assume that each game is made up of 200 frames
-In total we’ve made 20,000 decisions for going UP or DOWN and for each one of these we know the parameter gradient, which tells us how we should change the parameters if we wanted to encourage that decision in that state in the future.
-Now we have to label every decision we’ve made as good or bad.

Suppose we won 12 games and lost 88.

We’ll take all 200*12 = 2400 decisions we made in the winning games and do a positive update (filling in a +1.0 in the gradient for the sampled action, doing backprop, and parameter update encouraging the actions we picked in all those states).

And we’ll take the other 200*88 = 17600 decisions we made in the losing games and do a negative update (discouraging whatever we did).

The network will now become slightly more likely to repeat actions that worked, and slightly less likely to repeat actions that didn’t work.

Now we play another 100 games with our new, slightly improved policy network.

Weird property:
For example what if we made a good action in frame 50 (bouncing the ball back correctly), but then missed the ball in frame 150?
If every single action is now labeled as bad (because we lost), wouldn’t that discourage the correct bounce on frame 50?
It would.
However, when you consider the process over thousands/millions of games, then doing the first bounce correctly makes you slightly more likely to win down the road, so on average you’ll see more positive than negative updates for the correct bounce and your policy will end up doing the right thing.