## Andrej Karpathy blog

Deep Reinforcement Learning: Pong from Pixels

May 31, 2016

This is a long overdue blog post on Reinforcement Learning (RL). RL is hot! You may have noticed that computers can now automatically learn to play ATARI games (from raw game pixels!), they are beating world champions at Go, simulated quadrupeds are learning to run and leap, and robots are learning how to perform complex manipulation tasks that defy explicit programming. It turns out that all of these advances fall under the umbrella of RL research. I also became interested in RL myself over the last ~year: I worked through Richard Sutton's book, read through David Silver's course, watched John Schulmann's lectures, wrote an RL library in Javascript, over the summer interned at DeepMind working in the DeepRL group, and most recently pitched in a little with the design/development of OpenAI Gym, a new RL benchmarking toolkit. So I've certainly been on this funwagon for at least a year but until now I haven't gotten around to writing up a short post on why RL is a big deal, what it's about, how it all developed and where it might be going.
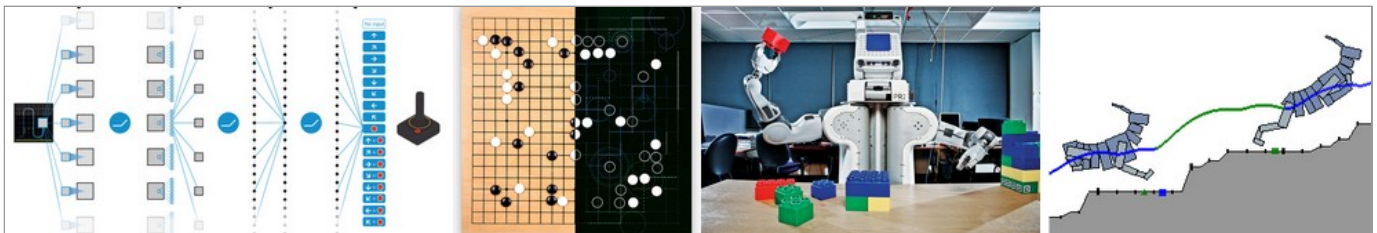
Examples of RL in the wild. **From left to right**: Deep Q Learning network playing ATARI, AlphaGo, Berkeley robot stacking Legos, physically-simulated quadruped leaping over terrain.

It's interesting to reflect on the nature of recent progress in RL. I broadly like to think about four separate factors that hold back 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.).

Similar to what happened in Computer Vision, the progress in RL is not driven as much as you might reasonably assume by new amazing ideas. In Computer Vision, the 2012 AlexNet was

mostly a scaled up (deeper and wider) version of 1990's ConvNets. Similarly, the ATARI Deep Q Learning paper from 2013 is an implementation of a standard algorithm (Q Learning with function approximation, which you can find in the standard RL book of Sutton 1998), where the function approximator happened to be a ConvNet. AlphaGo uses policy gradients with Monte Carlo Tree Search (MCTS) - these are also standard components. Of course, it takes a lot of skill and patience to get it to work, and multiple clever tweaks on top of old algorithms have been developed, but to a first-order approximation the main driver of recent progress is not the algorithms but (similar to Computer Vision) compute/data/infrastructure.

Now back to RL. Whenever there is a disconnect between how magical something seems and how simple it is under the hood I get all antsy and really want to write a blog post. In this case I've seen many people who can't believe that we can automatically learn to play most ATARI games at human level, with one algorithm, from pixels, and from scratch - and it is amazing, and I've been there myself! But at the core the approach we use is also really quite profoundly dumb (though I understand it's easy to make such claims in retrospect). Anyway, I'd like to walk you through Policy Gradients (PG), our favorite default choice for attacking RL problems at the moment. If you're from outside of RL you might be curious why I'm not presenting DQN instead, which is an alternative and better-known RL algorithm, widely popularized by the ATARI game playing paper. It turns out that Q-Learning is not a great algorithm (you could say that DQN is so 2013 (okay I'm 50% joking)). In fact most people prefer to use Policy Gradients, including the authors of the original DQN paper who have shown Policy Gradients to work better than Q Learning when tuned well. PG is preferred because it is end-to-end: there's an explicit policy and a principled approach that directly optimizes the expected reward. Anyway, as a running example we'll learn to play an ATARI game (Pong!) with PG, from scratch, from pixels, with a deep neural network, and the whole thing is 130 lines of Python only using numpy as a dependency (Gist link). Lets get to it.

Pong from pixels
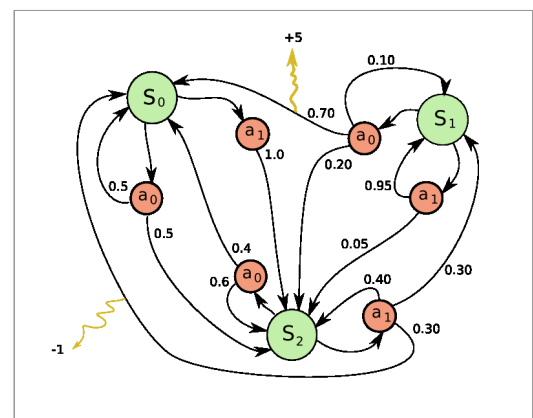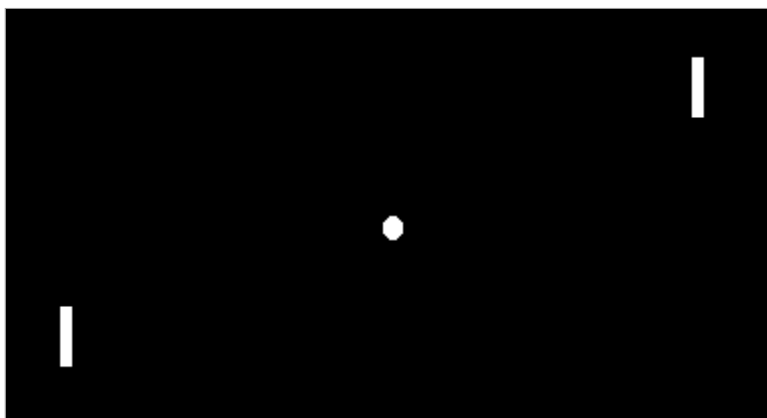
**Left:** The game of Pong. **Right:** Pong is a special case of a Markov Decision Process (MDP): A graph where each node is a particular game state and each edge is a possible (in general probabilistic) transition. Each

edge also gives a reward, and the goal is to compute the optimal way of acting in any state to maximize rewards.

The game of Pong is an excellent example of a simple RL task. In the ATARI 2600 version we'll use 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 (I don't really have to explain Pong, right?). On the low level the game works as follows: 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). 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. And of course, our goal is to move the paddle so that we get lots of reward.

As we go through the solution keep in mind that we'll try to make very few assumptions about Pong because we secretly don't really care about Pong; We care about complex, high-dimensional problems like robot manipulation, assembly and navigation. Pong is just a fun toy test case, something we play with while we figure out how to write very general AI systems that can one day do arbitrary useful tasks.

Policy network. First, we're going to define a *policy network* that implements our player (or "agent"). This network will take the state of the game and decide what we should do (move UP or DOWN). As our favorite simple block of compute we'll 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. Note that it is standard to use a *stochastic* policy, meaning that we only produce a *probability* of moving UP. Every iteration we will sample from this distribution (i.e. toss a biased coin) to get the actual move. The reason for this will become more clear once we talk about training.
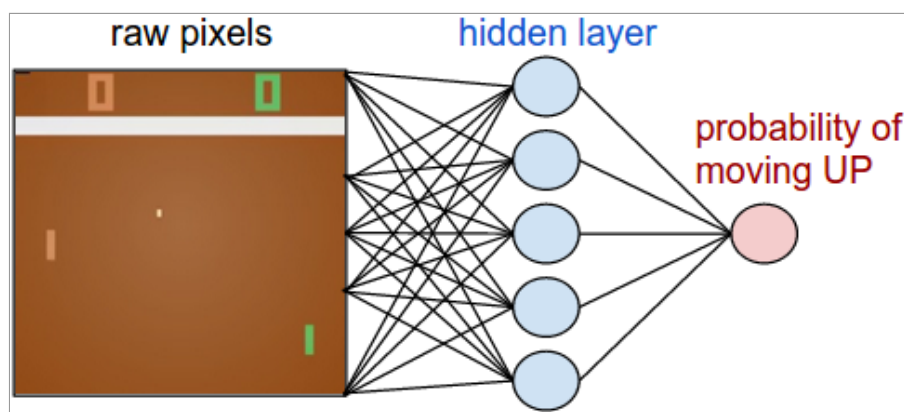
Our policy network is a 2-layer fully-connected net.

and to make things concrete here is how you might implement this policy network in Python/numpy. 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 g
```

where in this snippet `W1` and `W2` are two matrices that we initialize randomly. We're not using biases because meh. Notice that 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 (which have their weights arranged along the rows of `W1`) 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. Now, the initial random `W1` and `W2` will of course cause the player to spasm on spot. So the only problem now is to find `W1` and `W2` that lead to expert play of Pong!

Fine print: preprocessing. 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 (I did these experiments on my Macbook) I'll 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).

It sounds kind of impossible. At this point I'd like you to appreciate just how difficult the RL problem is. We get 100,800 numbers (210*160*3) and forward our policy network (which easily involves on order of a million parameters in `W1` and `W2`). Suppose that we decide to go UP. The game might respond that we get 0 reward this time step and gives us another 100,800 numbers for the next frame. We could repeat this process for hundred timesteps before we get any non-zero reward! E.g. 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 call this the *credit assignment problem*. In the specific case of Pong we know that 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 and things are looking quite bleak.

Supervised Learning. Before we dive into the Policy Gradients solution I'd like to remind you briefly about supervised learning because, as we'll see, RL is very similar. Refer to the diagram below. In ordinary supervised learning we would feed an image to the network and get some probabilities, e.g. for two classes UP and DOWN. I'm showing log probabilities (-1.2, -0.36) for UP and DOWN instead of the raw probabilities (30% and 70% in this case) because we always optimize the log probability of the correct label (this makes math nicer, and is equivalent to optimizing the raw probability because log is monotonic). Now, in supervised learning we would

have access to a label. For example, we might be told that the correct thing to do right now is to go UP (label 0). In an implementation we would enter gradient of 1.0 on the log probability of UP and run backprop to compute the gradient vector $\nabla_W \log p(y = UP \mid x)$. This gradient would tell us how we should change every one of our million parameters to make the network slightly more likely to predict UP. For example, one of the million parameters in the network might have a gradient of -2.1, which means that if we were to increase that parameter by a small positive amount (e.g. `0.001`), the log probability of UP would decrease by `2.1 *` `0.001` (decrease due to the negative sign). If we then did a parameter update then, yay, our network would now be slightly more likely to predict UP when it sees a very similar image in the future.
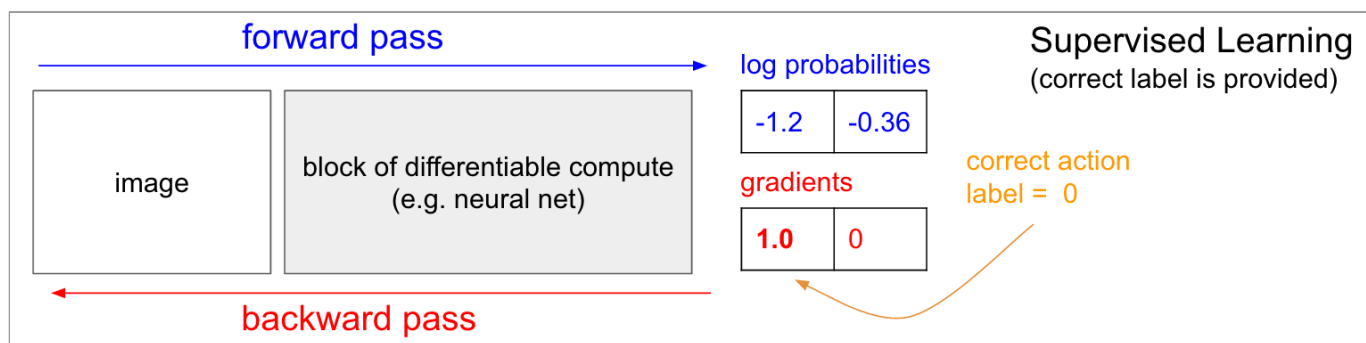
Policy Gradients. Okay, but what do we do if we do not have the correct label in the Reinforcement Learning setting? Here is the Policy Gradients solution (again refer to diagram below). Our policy network calculated probability of going UP as 30% (logprob -1.2) and DOWN as 70% (logprob -0.36). We will now sample an action from this distribution; E.g. suppose we sample DOWN, and we will execute it in the game. At this point notice one interesting fact: 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. So we can immediately evaluate this gradient and that's great, but the problem is that at least for now we do not yet know if going DOWN is good. But the critical point is that that's okay, because we can simply wait a bit and see! For example 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 (DOWN in this case). 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).
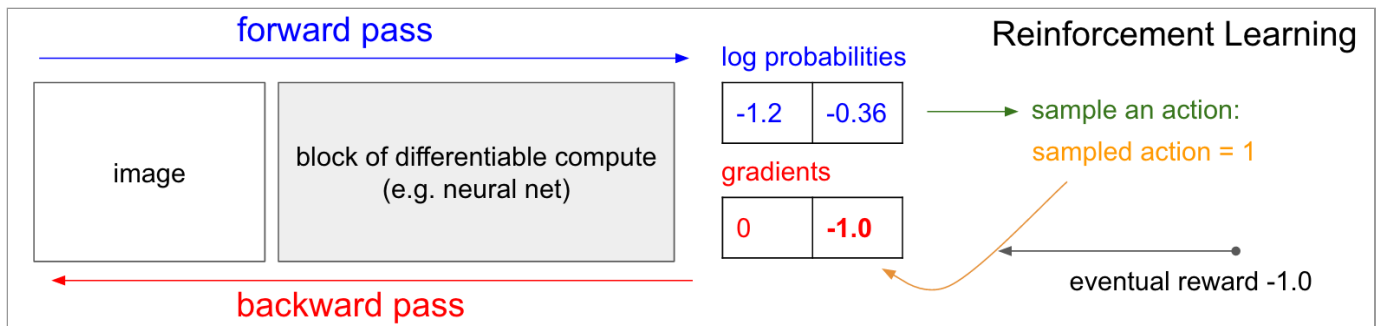
And that's it: we have a stochastic policy that samples actions and then actions that happen to eventually lead to good outcomes get encouraged in the future, and actions taken that lead to bad outcomes get discouraged. Also, the reward does not even need to be +1 or -1 if we win the game eventually. It can be an arbitrary measure of some kind of eventual quality. For example if things turn out really well it could be 10.0, which we would then enter as the gradient instead of -1 to start off backprop. That's the beauty of neural nets; Using them can feel like cheating: You're allowed to have 1 million parameters embedded in 1 teraflop of compute and you can make it do arbitrary things with SGD. It shouldn't work, but amusingly we live in a universe where it does.

Training protocol. So here is how the training will work in detail. 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 so 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. All that remains now is to label every decision we've made as good or bad. For example 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). And… that's it. 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 and rinse and repeat.

> *Policy Gradients: Run a policy for a while. See what actions led to high rewards. Increase their probability.*
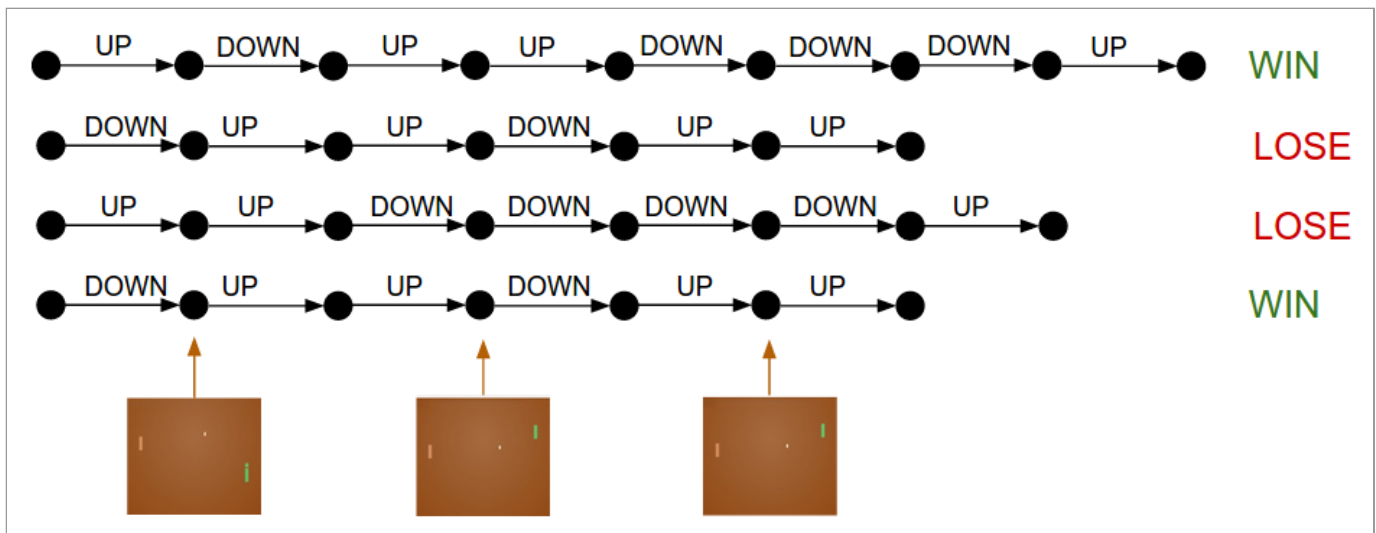
Cartoon diagram of 4 games. Each black circle is some game state (three example states are visualized on the bottom), and each arrow is a transition, annotated with the action that was sampled. In this case we won 2 games and lost 2 games. With Policy Gradients we would take the two games we won and slightly encourage every single action we made in that episode. Conversely, we would also take the two games we lost and slightly discourage every single action we made in that episode.

If you think through this process you'll start to find a few funny properties. 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? You're right - 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.

Update: December 9, 2016 - alternative view. In my explanation above I use the terms such as "fill in the gradient and backprop", which I realize is a special kind of thinking if you're used to writing your own backprop code, or using Torch where the gradients are explicit and open for tinkering. However, if you're used to Theano or TensorFlow you might be a little perplexed because the code is oranized around specifying a loss function and the backprop is fully automatic and hard to tinker with. In this case, the following alternative view might be more intuitive. In vanilla supervised learning the objective is to maximize $\sum_i \log p(y_i \mid x_i)$ where x_i, y_i are training examples (such as images and their labels). Policy gradients is exactly the same as supervised learning with two minor differences: 1) We don't have the correct labels y_i so as a "fake label" we substitute the action we happened to sample from the policy when it saw x_i, and 2) We modulate the loss for each example multiplicatively based on the eventual outcome, since we want to increase the log probability for actions that worked and decrease it for those that didn't. So in summary our loss now looks like $\sum_i A_i \log p(y_i \mid x_i)$, where y_i is the action we happened to sample and A_i is a number that we call an **advantage**. In the case of Pong, for example, A_i could be 1.0 if we eventually won in the episode that contained x_i and -1.0 if we lost. This will ensure that we maximize the log probability of actions that led to good

outcome and minimize the log probability of those that didn't. So reinforcement learning is exactly like supervised learning, but on a continuously changing dataset (the episodes), scaled by the advantage, and we only want to do one (or very few) updates based on each sampled dataset.

More general advantage functions. I also promised a bit more discussion of the returns. So far we have judged the *goodness* of every individual action based on whether or not we win the game. In a more general RL setting we would receive some reward r_t at every time step. One common choice is to use a discounted reward, so the "eventual reward" in the diagram above would become $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$, where γ is a number between 0 and 1 called a discount factor (e.g. 0.99). The expression states that the strength with which we encourage a sampled action is the weighted sum of all rewards afterwards, but later rewards are exponentially less important. In practice it can can also be important to normalize these. For example, suppose we compute R_t for all of the 20,000 actions in the batch of 100 Pong game rollouts above. One good idea is to "standardize" these returns (e.g. subtract mean, divide by standard deviation) before we plug them into backprop. This way we're always encouraging and discouraging roughly half of the performed actions. Mathematically you can also interpret these tricks as a way of controlling the variance of the policy gradient estimator. A more in-depth exploration can be found here.

Deriving Policy Gradients. I'd like to also give a sketch of where Policy Gradients come from mathematically. Policy Gradients are a special case of a more general *score function gradient estimator*. The general case is that when we have an expression of the form $E_{x \sim p(x|\theta)}[f(x)]$ - i.e. the expectation of some scalar valued score function $f(x)$ under some probability distribution $p(x; \theta)$ parameterized by some θ. Hint hint, $f(x)$ will become our reward function (or advantage function more generally) and $p(x)$ will be our policy network, which is really a model for $p(a \mid I)$, giving a distribution over actions for any image I. Then we are interested in finding how we should shift the distribution (through its parameters θ) to increase the scores of its samples, as judged by f (i.e. how do we change the network's parameters so that action samples get higher rewards). We have that:

$$\nabla_\theta E_x[f(x)] = \nabla_\theta \sum_x p(x) f(x) \qquad \text{definition of expectation}$$

$$= \sum_x \nabla_\theta p(x) f(x) \qquad \text{swap sum and gradient}$$

$$= \sum_x p(x) \frac{\nabla_\theta p(x)}{p(x)} f(x) \qquad \text{both multiply and divide by } p(x)$$

$$= \sum_x p(x) \nabla_\theta \log p(x) f(x) \qquad \text{use the fact that } \nabla_\theta \log(z) = \frac{1}{z} \nabla_\theta z$$

$$= E_x[f(x) \nabla_\theta \log p(x)] \qquad \text{definition of expectation}$$

To put this in English, we have some distribution $p(x; \theta)$ (I used shorthand $p(x)$ to reduce clutter) that we can sample from (e.g. this could be a gaussian). For each sample we can also evaluate the score function f which takes the sample and gives us some scalar-valued score. This equation is telling us how we should shift the distribution (through its parameters θ) if we wanted its samples to achieve higher scores, as judged by f. In particular, it says that look: draw some samples x, evaluate their scores $f(x)$, and for each x also evaluate the second term $\nabla_\theta \log p(x; \theta)$. What is this second term? It's a vector - the gradient that's giving us the direction in the parameter space that would lead to increase of the probability assigned to an x. In other words if we were to nudge θ in the direction of $\nabla_\theta \log p(x; \theta)$ we would see the new probability assigned to some x slightly increase. If you look back at the formula, it's telling us that we should take this direction and multiply onto it the scalar-valued score $f(x)$. This will make it so that samples that have a higher score will "tug" on the probability density stronger than the samples that have lower score, so if we were to do an update based on several samples from p the probability density would shift around in the direction of higher scores, making highly-scoring samples more likely.
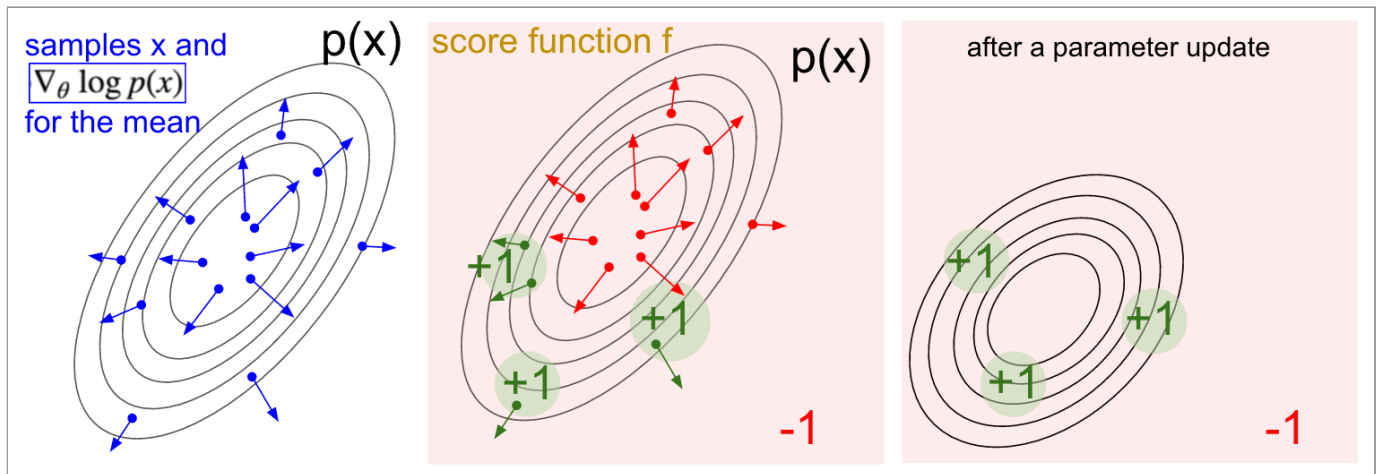
A visualization of the score function gradient estimator. **Left**: A gaussian distribution and a few samples from it (blue dots). On each blue dot we also plot the gradient of the log probability with respect to the gaussian's mean parameter. The arrow indicates the direction in which the mean of the distribution should be nudged to increase the probability of that sample. **Middle**: Overlay of some score function giving -1 everywhere except +1 in some small regions (note this can be an arbitrary and not necessarily differentiable scalar-valued function). The arrows are now color coded because due to the multiplication in the update we are going to average up all the green arrows, and the *negative* of the red arrows. **Right**: after parameter update, the green arrows and the reversed red arrows nudge us to left and towards the bottom. Samples from this distribution will now have a higher expected score, as desired.

I hope the connection to RL is clear. Our policy network gives us samples of actions, and some of them work better than others (as judged by the advantage function). This little piece of math is telling us that the way to change the policy's parameters is to do some rollouts, take the gradient of the sampled actions, multiply it by the score and add everything, which is what we've done above. For a more thorough derivation and discussion I recommend John Schulman's lecture.

Learning. Alright, we've developed the intuition for policy gradients and saw a sketch of their derivation. I implemented the whole approach in a 130-line Python script, which uses OpenAI Gym's ATARI 2600 Pong. I trained a 2-layer policy network with 200 hidden layer units using RMSProp on batches of 10 episodes (each episode is a few dozen games, because the games go up to score of 21 for either player). I did not tune the hyperparameters too much and ran the experiment on my (slow) Macbook, but after training for 3 nights I ended up with a policy that is slightly better than the AI player. The total number of episodes was approximately 8,000 so the algorithm played roughly 200,000 Pong games (quite a lot isn't it!) and made a total of ~800 updates. I'm told by friends that if you train on GPU with ConvNets for a few days you can beat the AI player more often, and if you also optimize hyperparameters carefully you can also consistently dominate the AI player (i.e. win every single game). However, I didn't spend too much time computing or tweaking, so instead we end up with a Pong AI that illustrates the main ideas and works quite well:
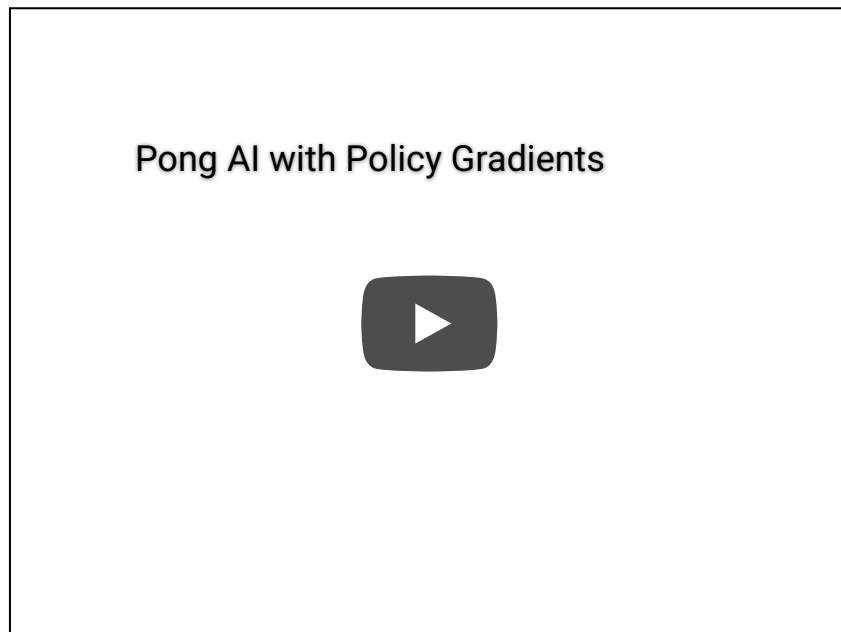
The learned agent (in green, right) facing off with the hard-coded AI opponent (left).

Learned weights. We can also take a look at the learned weights. Due to preprocessing every one of our inputs is an 80x80 difference image (current frame minus last frame). We can now take every row of `W1`, stretch them out to 80x80 and visualize. Below is a collection of 40 (out of 200) neurons in a grid. White pixels are positive weights and black pixels are negative weights. Notice that several neurons are tuned to particular traces of bouncing ball, encoded with alternating black and white along the line. The ball can only be at a single spot, so these neurons are multitasking and will "fire" for multiple locations of the ball along that line. The alternating black and white is interesting because as the ball travels along the trace, the neuron's activity will fluctuate as a sine wave and due to the ReLU it would "fire" at discrete, separated positions along the trace. There's a bit of noise in the images, which I assume would have been mitigated if I used L2 regularization.
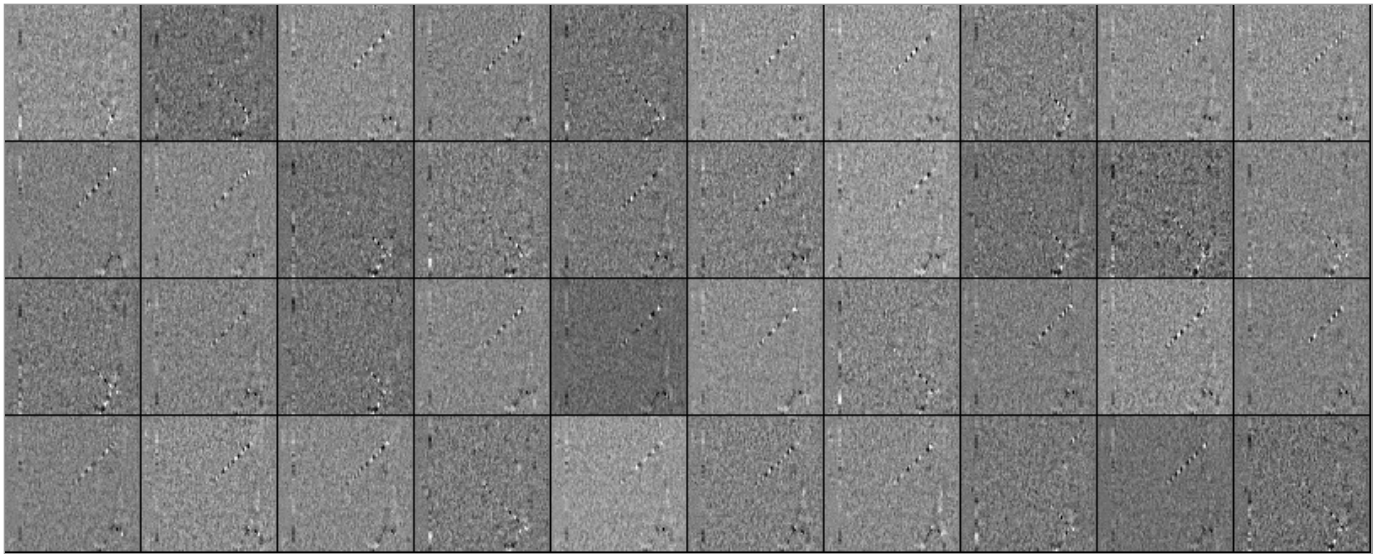
What isn't happening

So there you have it - we learned to play Pong from from raw pixels with Policy Gradients and it works quite well. The approach is a fancy form of guess-and-check, where the "guess" refers to sampling rollouts from our current policy, and the "check" refers to encouraging actions that lead to good outcomes. Modulo some details, this represents the state of the art in how we currently approach reinforcement learning problems. Its impressive that we can learn these behaviors, but if you understood the algorithm intuitively and you know how it works you should be at least a bit disappointed. In particular, how does it not work?

Compare that to how a human might learn to play Pong. You show them the game and say something along the lines of "You're in control of a paddle and you can move it up and down, and your task is to bounce the ball past the other player controlled by AI", and you're set and ready to go. Notice some of the differences:

- In practical settings we usually communicate the task in some manner (e.g. English above), but in a standard RL problem you assume an arbitrary reward function that you have to discover through environment interactions. It can be argued that if a human went into game of Pong but without knowing anything about the reward function (indeed, especially if the reward function was some static but random function), the human would have a lot of difficulty learning what to do but Policy Gradients would be indifferent, and likely work much better. Similarly, if we took the frames and permuted the pixels randomly then humans would likely fail, but our Policy Gradient solution could not even tell the difference (if it's using a fully connected network as done here).
- A human brings in a huge amount of prior knowledge, such as intuitive physics (the ball bounces, it's unlikely to teleport, it's unlikely to suddenly stop, it maintains a constant velocity, etc.), and intuitive psychology (the AI opponent "wants" to win, is likely following

an obvious strategy of moving towards the ball, etc.). You also understand the concept of being "in control" of a paddle, and that it responds to your UP/DOWN key commands. In contrast, our algorithms start from scratch which is simultaneously impressive (because it works) and depressing (because we lack concrete ideas for how not to).

- Policy Gradients are a *brute force* solution, where the correct actions are eventually discovered and internalized into a policy. Humans build a rich, abstract model and plan within it. In Pong, I can reason that the opponent is quite slow so it might be a good strategy to bounce the ball with high vertical velocity, which would cause the opponent to not catch it in time. However, it also feels as though we also eventually "internalize" good solutions into what feels more like a reactive muscle memory policy. For example if you're learning a new motor task (e.g. driving a car with stick shift?) you often feel yourself thinking a lot in the beginning but eventually the task becomes automatic and mindless.

- Policy Gradients have to actually experience a positive reward, and experience it very often in order to eventually and slowly shift the policy parameters towards repeating moves that give high rewards. With our abstract model, humans can figure out what is likely to give rewards without ever actually experiencing the rewarding or unrewarding transition. I don't have to actually experience crashing my car into a wall a few hundred times before I slowly start avoiding to do so.
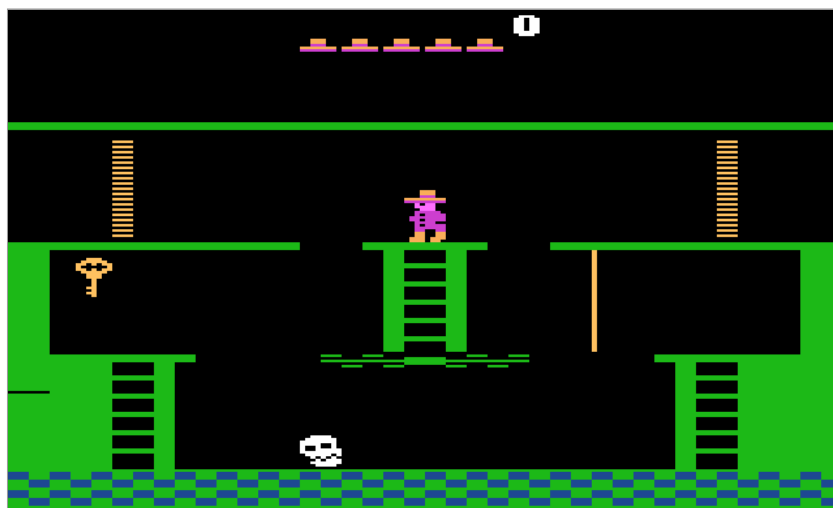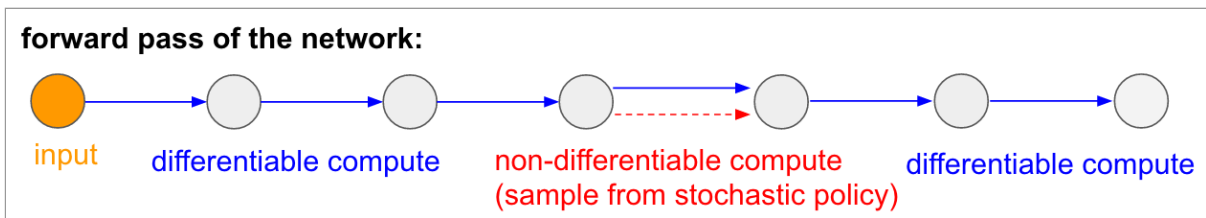
Left: Montezuma's Revenge: a difficult game for our RL algorithms. The player must jump down, climb up, get the key, and open the door. A human understands that acquiring a key is useful. The computer samples billions of random moves and 99% of the time falls to its death or gets killed by the monster. In other words it's hard to "stumble into" the rewarding situation. **Right:** Another difficult game called Frostbite, where a human understands that things move, some things are good to touch, some things are bad to touch, and the goal is to build the igloo brick by brick. A good analysis of this game and a discussion of differences between the human and computer approach can be found in Building Machines That Learn and Think Like People.

I'd like to also emphasize the point that, conversely, there are many games where Policy Gradients would quite easily defeat a human. In particular, anything with frequent reward signals that requires precise play, fast reflexes, and not too much long-term planning would be ideal, as these short-term correlations between rewards and actions can be easily "noticed" by the approach, and the execution meticulously perfected by the policy. You can see hints of this already happening in our Pong agent: it develops a strategy where it waits for the ball and then rapidly dashes to catch it just at the edge, which launches it quickly and with high vertical velocity. The agent scores several points in a row repeating this strategy. There are many ATARI games where Deep Q Learning destroys human baseline performance in this fashion - e.g. Pinball, Breakout, etc.

In conclusion, once you understand the "trick" by which these algorithms work you can reason through their strengths and weaknesses. In particular, we are nowhere near humans in building abstract, rich representations of games that we can plan within and use for rapid learning. One day a computer will look at an array of pixels and notice a key, a door, and think to itself that it is probably a good idea to pick up the key and reach the door. For now there is nothing anywhere close to this, and trying to get there is an active area of research.

Non-differentiable computation in Neural Networks

I'd like to mention one more interesting application of Policy Gradients unrelated to games: It allows us to design and train neural networks with components that perform (or interact with) non-differentiable computation. The idea was first introduced in Williams 1992 and more recently popularized by Recurrent Models of Visual Attention under the name "hard attention", in the context of a model that processed an image with a sequence of low-resolution foveal glances (inspired by our own human eyes). In particular, at every iteration an RNN would receive a small piece of the image and sample a location to look at next. For example the RNN might look at position (5,30), receive a small piece of the image, then decide to look at (24, 50), etc. The problem with this idea is that there a piece of network that produces a distribution of where to look next and then samples from it. Unfortunately, this operation is non-differentiable because, intuitively, we don't know what would have happened if we sampled a different location. More generally, consider a neural network from some inputs to outputs:

forward pass of the network:

input   differentiable compute   non-differentiable compute
(sample from stochastic policy)   differentiable compute

Notice that most arrows (in blue) are differentiable as normal, but some of the representation transformations could optionally also include a non-differentiable sampling operation (in red). We can backprop through the blue arrows just fine, but the red arrow represents a dependency that we cannot backprop through.

Policy gradients to the rescue! We'll think about the part of the network that does the sampling as a small stochastic policy embedded in the wider network. Therefore, during training we will produce several samples (indicated by the branches below), and then we'll encourage samples that eventually led to good outcomes (in this case for example measured by the loss at the end). In other words we will train the parameters involved in the blue arrows with backprop as usual, but the parameters involved with the red arrow will now be updated independently of the backward pass using policy gradients, encouraging samples that led to low loss. This idea was also recently formalized nicely in Gradient Estimation Using Stochastic Computation Graphs.

during training: sample & use PG:

input   embedded policy
(train red arrows with PG)

supervision

train blue arrows with
backprop as usual
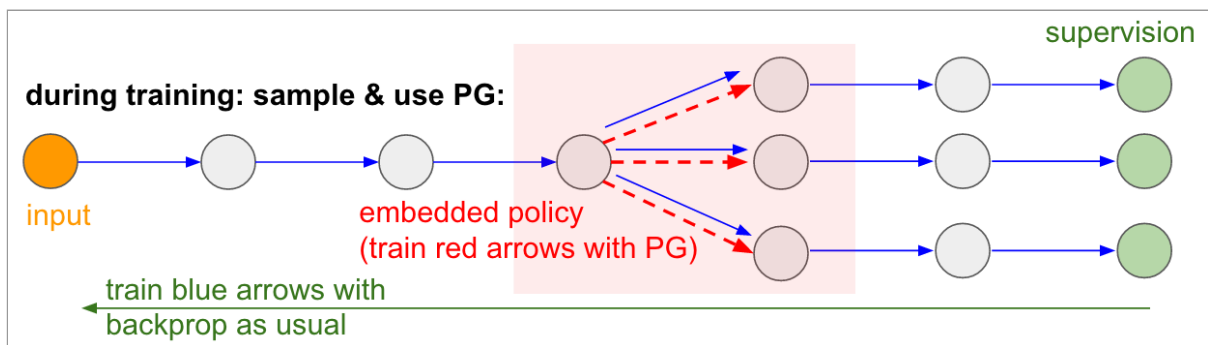
Trainable Memory I/O. You'll also find this idea in many other papers. For example, a Neural Turing Machine has a memory tape that they it read and write from. To do a write operation one would like to execute something like `m[i] = x`, where `i` and `x` are predicted by an RNN controller network. However, this operation is non-differentiable because there is no signal telling us what would have happened to the loss if we were to write to a different location `j != i`. Therefore, the NTM has to do *soft* read and write operations. It predicts an attention distribution `a` (with elements between 0 and 1 and summing to 1, and peaky around the index we'd like to write to), and then doing `for all i: m[i] = a[i]*x`. This is now differentiable, but we have to pay a heavy computational price because we have to touch every single memory cell just to write to one position. Imagine if every assignment in our computers had to touch the entire RAM!

However, we can use policy gradients to circumvent this problem (in theory), as done in RL-NTM. We still predict an attention distribution `a`, but instead of doing the soft write we sample locations to write to: `i = sample(a); m[i] = x`. During training we would do this for a small batch of `i`, and in the end make whatever branch worked best more likely. The large computational advantage is that we now only have to read/write at a single location at test time. However, as pointed out in the paper this strategy is very difficult to get working because one must accidentally stumble by working algorithms through sampling. The current consensus is that PG works well only in settings where there are a few discrete choices so that one is not hopelessly sampling through huge search spaces.

However, with Policy Gradients and in cases where a lot of data/compute is available we can in principle dream big - for instance we can design neural networks that learn to interact with large, non-differentiable modules such as Latex compilers (e.g. if you'd like char-rnn to generate latex that compiles), or a SLAM system, or LQR solvers, or something. Or, for example, a superintelligence might want to learn to interact with the internet over TCP/IP (which is sadly non-differentiable) to access vital information needed to take over the world. That's a great example.

Conclusions

We saw that Policy Gradients are a powerful, general algorithm and as an example we trained an ATARI Pong agent from raw pixels, from scratch, in 130 lines of Python. More generally the same algorithm can be used to train agents for arbitrary games and one day hopefully on many valuable real-world control problems. I wanted to add a few more notes in closing:

On advancing AI. We saw that the algorithm works through a brute-force search where you jitter around randomly at first and must accidentally stumble into rewarding situations at least once, and ideally often and repeatedly before the policy distribution shifts its parameters to repeat the responsible actions. We also saw that humans approach these problems very differently, in what feels more like rapid abstract model building - something we have barely even scratched the surface of in research (although many people are trying). Since these abstract models are very difficult (if not impossible) to explicitly annotate, this is also why there is so much interest recently in (unsupervised) generative models and program induction.

On use in complex robotics settings. The algorithm does not scale naively to settings where huge amounts of exploration are difficult to obtain. For instance, in robotic settings one might have a single (or few) robots, interacting with the world in real time. This prohibits naive applications of the algorithm as I presented it in this post. One related line of work intended to mitigate this problem is deterministic policy gradients - instead of requiring samples from a stochastic policy and encouraging the ones that get higher scores, the approach uses a

deterministic policy and gets the gradient information directly from a second network (called a *critic*) that models the score function. This approach can in principle be much more efficient in settings with very high-dimensional actions where sampling actions provides poor coverage, but so far seems empirically slightly finicky to get working. Another related approach is to scale up robotics, as we're starting to see with Google's robot arm farm, or perhaps even Tesla's Model S + Autopilot.

There is also a line of work that tries to make the search process less hopeless by adding additional supervision. In many practical cases, for instance, one can obtain expert trajectories from a human. For example AlphaGo first uses supervised learning to predict human moves from expert Go games and the resulting human mimicking policy is later finetuned with policy gradients on the "real" objective of winning the game. In some cases one might have fewer expert trajectories (e.g. from robot teleoperation) and there are techniques for taking advantage of this data under the umbrella of apprenticeship learning. Finally, if no supervised data is provided by humans it can also be in some cases computed with expensive optimization techniques, e.g. by trajectory optimization in a known dynamics model (such as $F = ma$ in a physical simulator), or in cases where one learns an approximate local dynamics model (as seen in very promising framework of Guided Policy Search).

On using PG in practice. As a last note, I'd like to do something I wish I had done in my RNN blog post. I think I may have given the impression that RNNs are magic and automatically do arbitrary sequential problems. The truth is that getting these models to work can be tricky, requires care and expertise, and in many cases could also be an overkill, where simpler methods could get you 90%+ of the way there. The same goes for Policy Gradients. They are not automatic: You need a lot of samples, it trains forever, it is difficult to debug when it doesn't work. One should always try a BB gun before reaching for the Bazooka. In the case of Reinforcement Learning for example, one strong baseline that should always be tried first is the cross-entropy method (CEM), a simple stochastic hill-climbing "guess and check" approach inspired loosely by evolution. And if you insist on trying out Policy Gradients for your problem make sure you pay close attention to the *tricks* section in papers, start simple first, and use a variation of PG called TRPO, which almost always works better and more consistently than vanilla PG in practice. The core idea is to avoid parameter updates that change your policy too much, as enforced by a constraint on the KL divergence between the distributions predicted by the old and the new policy on a batch of data (instead of conjugate gradients the simplest instantiation of this idea could be implemented by doing a line search and checking the KL along the way).

And that's it! I hope I gave you a sense of where we are with Reinforcement Learning, what the challenges are, and if you're eager to help advance RL I invite you to do so within our OpenAI Gym :) Until next time!

108 Comments   Andrej's Blog   1 **Login**

♡ Recommend 154   🐦 Tweet   f Share   Sort by Best

Join the discussion…

LOG IN WITH   OR SIGN UP WITH DISQUS ?

                           Name

HeyUser4 • 3 years ago

Hey Karpathy, thanks for this. It is absolutely amazing. I wanted to say it for a long time, but this opportunity is as good as any. You are one of the best teachers ever. Thank you for your efforts.

40 ∧ | ∨ • Reply • Share ›

    Yongliang Qin ➔ HeyUser4 • 2 years ago

    Hinton set the baseline for neural networks
    LeCun set the baseline for convolutional neural networks
    ImageNet set the baseline for visual recognition competitions
    DeepMind set the baseline for deep RL algorithms
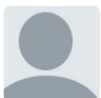   OpenAI set the baseline for deep RL environments

    Karpathy set the baseline for deep learning tutorials (at least for hackers).

    33 ∧ | ∨ • Reply • Share ›

    Jon Gomez ➔ HeyUser4 • 3 years ago

    I second that.

    2 ∧ | ∨ • Reply • Share ›

        Michael Campos ➔ Jon Gomez • 3 years ago

        Glad you found a job that let's you keep doing this sort of thing.

        ∧ | ∨ • Reply • Share ›

    Show more replies

Peter • 2 years ago

Hello! Just wanted to say thanks for all the info and inspiration. I recently read this blog and watched related videos and got the Atari python program working - I have never done any python before last weekend.
However, the game halts after 20 points are reached by the Atari Computer (initially it scores 20 usually without the AI learned player scoring a single point). Then it reliably halts as soon as 20 points are reached. Could this be something in the code, or could this be a lack of resources to continue the computation? I am running it on my only linux computer, which is a Raspberry pi :) therefore I likely think it hits a resource limit.

a Raspberry pi,) therefore I likely think it hits a resource limit.

I accept that it will take a very long time to learn, but I am in no rush. So could the problem of halting with no error message after 20 points be caused by the computer (e.g. the pi) or the program? Have you seen this before?

Any information would be much appreciated while I seek to obtain another linux environment.

Many thanks again for all your work and publications.

49 ∧ | ∨ · Reply · Share ›

Dmitry Rybakov · 7 months ago

Looks like overfitting does not matter for pong game. Neural Network may just commit to memory all combinations.

22 ∧ | ∨ · Reply · Share ›

hardmaru · 3 years ago

Very cool post. What you mention about multi-robot and actor-critic approach is also very interesting. What do you think will happen if you train the agent to play against itself, rather than against the hard-coded AI, and learn to play pong entirely via self-play? The second agent will use a flipped screen as the input, so the game states will be identical. I wonder if this will lead to ultimately a better pong-playing agent.

11 ∧ | ∨ · Reply · Share ›

Qian Hong · 3 years ago

Great post, thank you!

It seems like the algorithm of learning Pong from Pixels could also be used to learn automatically UI testing from Pixels: Assume we compile a GUI program like Firefox/Chrome/LibreOffice with test coverage benchmark tools like Gcov, so we can get a real time "score of game" using Gcov [1][2] output of "lines of covered code", then we train the policy network to generate a distribution on {move mouse right, move mouse left, left click, right click, type char a, type char b, ...}, sample from the distribution and interactive with the GUI program, try to maximum the lines of covered code, with enough training the RL algorithm might be able to learn to click on menus and sub menus and even type some random strings.

If we train a network with multiple GUI programs including Firefox/Chrome/LibreOffice/etc, on different platforms including Linux/Android/Mac/iOS/Windows, with different resolutions including 1024*768/1440*900/etc, finally the network might become more and more smart because there would be a lot of shared knowledge among different GUI programs, different platforms and different screen resolutions, which might be able to be learned by the RL network. Once we archive this stage, we could try to use the training result to test closed

see more

5 ∧ | ∨ · Reply · Share ›

Shaohua Li ↱ Qian Hong · a year ago

this is a good idea that is definitely worth trying!

˄ | ˅ • **Reply** • **Share ›**

Neil • 3 years ago

I have a question that

logp = np.dot(W2, h) # compute log probability of going up

Why does this line compute the log probability? since the sigmoid function's codomain is (0, 1), the domain ought to be (-inf, inf), but the log probability's codomain with domain (0, 1) is (-inf, 0).

7 ˄ | ˅ • **Reply** • **Share ›**

Pawel Zaluski ➜ Neil • a year ago

Seems like it would make more sense to call it logit rather than logprob:

logit = np.dot(W2.h) # compute logit of going up
p = 1.0 / (1.0 + np.exp(-logit)) # sigmoid function (gives probability of going up)

3 ˄ | ˅ • **Reply** • **Share ›**

Daniel Seita ➜ Neil • 3 years ago

Good question ... it might be an artifact of naming conventions from other pieces of code?

1 ˄ | ˅ • **Reply** • **Share ›**

Show more replies

alesomenumber • 3 years ago

Brilliant post (as always). One tiny pedantic nitpick: DeepQ does not "destroy" humans in Breakout, or even match a non-novice player. Pretty much all papers from DeepMind get around 400 points in Breakout, which is almost clearing the first level. Their >100% comparison with human players is due to artificially handicapping of humans, in that Breakout wasn't designed to be played through a keyboard, but through a paddle. Clearing the first level with a paddle is something most people can do after playing for a couple of hours.

Deep Q gets stuck clearing the first level most of the time because it never learns to find isolated bricks (rewards are too sparse). It mainly learns to not let the ball through.

4 ˄ | ˅ • **Reply** • **Share ›**

Tejas Kulkarni • 3 years ago

nice post!

re sparse and delayed rewards (especially for games like Montezuma's Revenge), you might find this paper interesting --

http://arxiv.org/abs/1604.0...

here's the demo on montezuma's revenge:
https://drive.google.com/fo...

You rightly pointed out that model-building will be key to scale up RL.

IMHO, the deep generative modeling community (esp for vision data) is a bit lost or going in a different direction. I think building models that learn intermediate representations like objects, their intrinsic properties like relative pose/location, 3D shape etc will be crucial for RL and other tasks (mostly for solving goals that humans solve in physical environments). This will enable agents to explore hierarchically without wasting time on a senseless manifold. For this to work, we need to put in a lot more structure in our deep generative models -- spatio-temporal continuity, inductive biases for variable length items, and other inductive biases for perceptual binding. This will also take us away from building generative models that predict pixels. We should avoid doing this as much as possible anyways (except within a local context). Will simply putting the right inductive structure be enough to bootstrap learning such structured representations? I don't know. We will probably need semi-supervised approaches to further bootstrap this and therefore probably need new datasets.

5 ∧ | ∨ • **Reply** • **Share ›**

Shaohua Li ➜ Tejas Kulkarni • a year ago

Yeah I totally agree. Always think that it's crazy for deep generative models to try to reconstruct the whole image. Doing this drowns the model in the finest details instead of the big picture (rough structures). Anyway brains don't and couldn't do this.

10 ∧ | ∨ • **Reply** • **Share ›**

karpathy Mod ➜ Tejas Kulkarni • 3 years ago

Hey Tejas, thanks for dropping by :) I go back and forth about exactly how much structure we want to build into the networks. I think people have a bit of a tunnel vision about objects and forget that there are things like trees/bushes/grass, water, paper/stack of papers, sand, wires/fences/cables, clouds, messy piles of objects, etc. How are these represented in the brain? Surely it's not modeling every individual leaf as its own object, it's some weird "blob" thing with its own statistical movement patterns. Or when a piece of glass shatters into 5 - is there a discrete transition in the representation? Due to these reasons my convictions about encoding more structure waver on a sine curve.

5 ∧ | ∨ • **Reply** • **Share ›**

Tejas Kulkarni ➜ karpathy • 3 years ago

You raise some very interesting meta-physical questions -- the distinction between "things" and "stuff" and the recursiveness in them (stuff consists of things. but you decide to look closer at one of the things. this thing then essentially becomes stuff and the cycle continues). I agree that we should not put priors into our models so explicitly. Maybe we need the right kind of very high level inductive priors so these concepts emerge without too much computation -- spatio-temporal continuity, bias for variable binding, variable length, hierarchy of variables with inheritance and many more.

∧ | ∨ • **Reply** • **Share ›**

Show more replies

Zypex117 · 8 months ago

I've got a few questions:

1) Why are we using logp instead of p, the value from the sigmoid function?

2) How are the raw probabilities for UP and DOWN calculated? From my understanding higher the value of the logp the higher the raw probabilities are, but how are the exact values of 30% and 70% calculated? 0.36/1.2 = 0.3 is the only way I can see how the raw probabilities being calculated but that'd mean 0.6/1.2 = 0.5 so UP would have an equal probability of being picked as DOWN despite having twice the negative value as DOWN.

3) What do you mean by filling a gradient of 1 for DOWN? Does this mean dDOWN/dTheta = 1, where Theta in this situation is all the outputs of previous layer coming into that DOWN node?

2 ∧ ∨ · Reply · Share ›

> **Attila Kun** ➜ Zypex117 · 8 months ago
> Answer to question 2:
> log(p)=-1.2 => p=e^(-1.2)=0.301194212
> log(p)=-0.36 => p=e^(-0.36)=0.697676326
> 1 ∧ ∨ · Reply · Share ›

Nick Nobutaka Toyoda Kim · 3 months ago

https://stackoverflow.com/q...

I have no idea why a fake label is used. Don't we use the rewards of -1, 0, 1 as labels and then either encourage or discourage the actions based on outcome of the episode? Also what is y - aprob? Finally a log operation is not performed in the forward pass but it somehow is called logp. Please explain.

1 ∧ ∨ · Reply · Share ›

ThinKing Particle · a year ago

thanks a lot for the great post.
i reimplemented the project in keras.
https://github.com/thinking...

1 ∧ ∨ · Reply · Share ›

jupiter · a year ago

implemented it with tensorflow
https://github.com/gameofdi...

1 ∧ ∨ · Reply · Share ›

Juho Kupiainen · 2 years ago

A chat room for deep learning experts: taticti.com. The topic is general AI

A chat room for deep learning experts. tatioti.com. The topic is general AI.

1 ∧ | ∨ · Reply · Share ›

Shalabh Singh · 2 years ago

This post is so good !! This was the first blog post on deep reinforcement learning I read and it motivated me to learn more.

Here is the insight I have developed - I tried making a RL agent using policy gradients for Flappy Bird Game and the solutions weren't satisfactory. After, I had spent enough time on tuning the hyperparameters in policy gradient method, I learnt that policy gradient methods are in general not that good for getting good results. An actor-critic model (as Karpathy references in the paper above) using both policy gradient and Q-value is better and DQN is even better for good results.

However actor-critic method wins out as the favorite algorithm to use, because it's training time is less than a day on a CPU (for feed-forward network only). So using this approach, I made a Flappy Bird learner that trains in only 6 hours (even less than expected). I wrote a blog post to explain it. Here's the link - https://shalabhsingh.github...

1 ∧ | ∨ · Reply · Share ›

ibmua ➜ Shalabh Singh · 2 years ago

Thanks for sharing the code and experience! I'm pretty sure, though, if you were to implement the correct understanding of the game - rewards and other stuff - and tinker a lil bit, PG would work extremely well. Also, it might be that you have non-breaking bugs in your code, kind of like they talk about here https://blog.openai.com/ope... - I'd say that's highly likely. Also, seems like threading actually does nothing in your code. And you might want to modify wrapped_flappy_bird.py for larger FPS.

Flappy seems like one of the easiest targets for RL, so there shouldn't be any problem, really.

∧ | ∨ · Reply · Share ›

Shalabh Singh ➜ ibmua · 2 years ago

Thanks for your insight. I am by no means an experienced practitioner in this area so it might be the case that I would have some bugs in the code. But there are a few things I am confident about- threading definitely helps in converging to a better solution and without threading (or experience replay as in DQN), it is very difficult to get a satisfactory solution at all or else people would have mastered atari games before DeepMind did in 2013 using experience replay. In my code, the threads do not run in parallel as they should have but they run independently one after the other and weight updates proposed by them are then included in the model simultaneously after a few runs which is actually what is done in A3C models.

I kept the FPS to a value of 30 so that the game doesn't become too fast to watch when the trained model actually plays the game. The value can be changed as the user wants to in the actual run.

1 ∧ | ∨ · **Reply** · **Share ›**

Show more replies

Niu Tong · 2 years ago

Hi Andrej, thank you for your amazingly clear post. I am now trying to understand the policy gradient you illustrated in terms of sequence level training (https://arxiv.org/pdf/1511..... In eq.9, the paper gives the loss function and states that "in practice, we approximate this expectation with a single sample
from the distribution of actions implemented by the RNN". I wonder if I have a scalar reward value for the whole generated sequence, how should I back-propagate this loss if I am using Tensorflow? Thank you!

1 ∧ | ∨ · **Reply** · **Share ›**

Abhishek Mishra · 3 years ago

Thanks for the post!

I was looking at your code of the Pong's game. Please correct me if my understanding is wrong.

For training the policy network, you are assuming that the actions in the Pong games are coming from a Gaussian distribution with variance 1 and the neural network is predicting the mean of this Gaussian distribution.
If this is correct, then for deciding the action you are using the Bernoulli distribution with the mean predicted by the policy network. This mismatch in the distribution between training and actually playing the game is working. But do you have any mathematical justification about why is it working?

1 ∧ | ∨ · **Reply** · **Share ›**

> **karpathy** Mod ↱ Abhishek Mishra · 3 years ago
>
> there are no gaussians, it's just a single Bernoulli at both train and test time.
>
> ∧ | ∨ · **Reply** · **Share ›**

> > **Abhishek Mishra** ↱ karpathy · 3 years ago
> > I have one confusion in this regard.
> >
> > In your code, during the back propagation, you passed the quantity [(fake_labels - probability) * discounted_return] to the network. "Fake Labels" are the labels that you have assigned to the actions as it was described in the code.
> >
> > Using the policy gradient theorem, if the actions are continuous and policy network is predicting the mean of the Gaussian distribution, only then I was able to prove that passing the above quantity is mathematically justifiable.
> >
> > In case we are using the Bernouli distribution, then my calculation says that we should pass ((fake_labels - probability)/(probability*(1-probability))).

My calculation could be wrong but I double checked it. Please let me know where am I wrong.

Thank you

∧ | ∨ • Reply • Share ›

Show more replies

esscee • 3 years ago

Enjoyed this post and mentioned it on This Week in Machine Learning & AI - 6/3/16 (towards the end). Thanks!

https://soundcloud.com/twim...

1 ∧ | ∨ • Reply • Share ›

장영연 • 14 days ago

first of all, thank you so much! i have a question.

i am analyzing your python code. i am very curious about hidden state meaning.

∧ | ∨ • Reply • Share ›

Kurt Driessens • 16 days ago

Hi Andrej,

I love your blogposts and have referenced them multiple times while teaching, but I think your statement on normalisation or standardisation of the discounted rewards is confusing. You say that if you normalise these, you encourage and discourage 50% of the actions. While this might work in a game like pong, what if you actually need a sequence of actions to solve a game? The variance reduction that you point to is actually something quite different, since it puts a constraint on the baseline function that you can subtract from the gradient that says it should be action independent (i.e. it can be only state dependent). The mean of an episode will most definitely not be action independent. As said, this might work for a game like pong where the last actions of an episode are probably the ones that won you the point and thus the ones you want to encourage, but you should not generalise this. Kind regards,
Kurt Driessens

∧ | ∨ • Reply • Share ›

paws • 21 days ago

So in the video i saw that the policy you train continuously keeps moving and then hits the ball. What you do to train the policy is give a game a positive reward if it wins and a negative reward if it loses. So for thought experiments sakes, what if i tried to penalize the amount of movement? like i would count the number of times the pong paddle moves rapidly in a game and penalize it above a threshold ? would that kind of penalty enforce the bot to move smoothly ?

In general would such multi reward agents learn to do justice / optimize both?

∧ | ∨ • Reply • Share ›

Mehmet Çağrı • 3 months ago

I have one question. Did i use any game for this? I mean can i use any race game or advanture game? or just i need to use spesific invoriments?

∧ | ∨ • Reply • Share ›

Yossarian42 • 3 months ago

Thanks a lot!!! Wonderful explanation and introduction to policy gradient

∧ | ∨ • Reply • Share ›

Micheldeweert • 5 months ago

Hello Karpathy! I want to compliment you on the wonderful post you have made! I do have a question on this topic, namely, is it possible to adapt the implementation to another game, for example breakout? Does the reward function then needs to be adapted to scale from -1 to 1? And if we use a larger action space than 2, would the loss still be well defined?

∧ | ∨ • Reply • Share ›

Rayed Wahed • a year ago

I'm sorry since this may sound stupid to a lot of you but is there a trained network available that i can deploy in a gym environment and play with?

∧ | ∨ • Reply • Share ›

Kirill Makukhin • a year ago

Andrej, when you discussing the difference of policy grad to humans you say "I don't have to actually experience crashing my car into a wall a few hundred times before I slowly start avoiding to do so". That's actually is not quite true. You have learned the physics model and consequences of bumping into a wall when you was 1 year old (yes, a few hundred times :)). Now you are able to scaffold the car driving with that knowledge. Perhaps, this is indeed the most amaizing property of the brain that is hard hard to model.

∧ | ∨ • Reply • Share ›

Kaushal Maurya • a year ago

Hi Karpathy, Thanks lot for such wonderful blog. It's really awesome. From last 6-7 month I am also learning and implementing deep RL stuff(mostly on Actor-Critic model). However, I have a doubt(forgive me if it is naive) i.e. How to identify the convergence of algorithm(mathematically, not visually)? How many episodes are needed to get optimal policy? Any suggestion in this direction will be much appreciable. Thank you.

∧ | ∨ • Reply • Share ›

vznvzn • a year ago

brilliant! thx for all the details! somewhat inspired by game AI research, suspect there are some deep AGI principles involved. in particular could these algorithms find game solutions with some "not very radical" modifications, *without using game scores*? was esp interested in the montezuma revenge case study. isnt it true the AI attempts fail in "nearly the same way(s)"? what about an algorithm that "tries" to fail in *novel* ways? some of that is at the heart of this new semicomprehensive theory for AGI...

https://vzn1.wordpress.com/...

∧ | ∨ • Reply • Share ›

Olle Welin • a year ago

I was try to do a similar with a C++ program and OpenCV
But there I was test with a Ball or a Square symbol. The agent was told to learn to catch the
Ball but Avoid the rectangle.
https://www.youtube.com/wat...
https://github.com/olleweli...

∧ | ∨ • Reply • Share ›

mayduavongts • a year ago

Thanks for sharing, nice post! Post really provice useful information!
http://anthaison.vn/sp/may-...

∧ | ∨ • Reply • Share ›

Abhishek Dubey • 2 years ago

You rock dude, as always.
I have converted you code to TF and its works like butter :)
https://github.com/Abhishek...

∧ | ∨ • Reply • Share ›

Olle Welin • 2 years ago

Thanks very much.
I was test the approach and wrote an own test program in C++ OpenCV on my Raspberry Pi
3
Code Basic version here:
https://gist.github.com/oll...

Or a extended version here (here I correct so the dice also are involved when take action):
https://gist.github.com/oll...

Video here:
https://www.youtube.com/wat...
I was supriced when it start to work.
After night run about 11000 episode it win almost every time. And the visualization of the
weights get stronger.

Andrej Karpathy blog   ⊙ karpathy   Musings of a Computer Scientist.
                                   🐦 karpathy