Reinforcement Learning: Incorporating Human Preferences in the Fine-Tuning of Large Language Models

Avinash Kak

Purdue University

Lecture Notes on Deep Learning by Avi Kak and Charles Bouman

Tuesday 22nd April, 2025 09:36

Preamble

To TOC To HowTo

Reinforcement Learning as a research subject owes its origins to the study of behaviorism in psychology. The behaviorists believe that, generally speaking, our minds are shaped by the reward structures in a society. B. F. Skinner, a famous American psychologist of the last century, is considered to be the high priest of behaviorism.

Unfortunately for the behaviorists, their research was dealt a serious blow by Noam Chomsky. Through his investigations in languages, Chomsky came to the conclusion that the most fundamental structures in our minds are shaped by our biology regardless of where we are born (implying that regardless of the reward structures in a society).

In machine learning, reinforcement learning as a topic of investigation owes its origins to the work of Andrew Barto and Richard Sutton at the University of Massachusetts. Their 1998 book "Reinforcement Learning: An Introduction" (for which a much more recent second-edition is now available on the web) is still a go-to source document for this area.

The goal of this lecture is to give you an introduction to how to incorporate human preferences in LLMs with Reinforcement Learning (RL).

Preamble (contd.)

I suppose your first question is likely to be: What? Human preferences? Why?

Since LLM are trained on terabytes of textual data, it is highly likely that they would pick up inappropriate information along the way.

[**Just imagine an LLM that was trained unsupervised as all LLMs are. Along the lines of my Week 14 lecture, imagine that the unsupervised training of this LLM was based on its ingesting pairs of textual sequences, with the second being a continuation of the first. Obviously depending on the nature of the first sequence, it is entirely possible that the LLM would pick up multiple possibilities for its continuation — with some using profanities and other forms of foul and possibly violent language.**]

What that implies is that, in general, you would need to sanitize the learned model before its public release, or before having businesses build their applications on top of the model.

[**Sanitizing an LLM is made complex by the fact that what is appropriate in one cultural context many not be so in another.**]

Sanitizing an LLM falls under the general category of how to incorporate the more acceptable human preferences in a model.

While many aspects of how the AI-focused companies bring to bear normal human preferences on their LLMs are not entirely clear, broadly speaking though, it is generally believed that they use RL for the job.

Preamble (contd.)

In particular, the companies use an approach known as "Proximal Policy Optimization Algorithms (PPO)" described in the following paper by Schulman et al.:

https://arxiv.org/pdf/1707.06347.pdf

PPO is an algorithm for iterative updating an RL agent's policy (a policy is a mapping from the states of the environment — LLM in our case — to the actions by the agent vis-a-vis the environment) in a sufficiently conservative manner that ensures the stability of the model updating process.

There is another algorithm also out there of very recent vintage for incorporating human preferences in language models: the DPO algorithm (Direct Preference Optimization) presented in following paper by Rafailov et el. It claims to get about the same sort of results as the PPO approach but more simply and without using RL:

https://arxiv.org/pdf/2305.18290.pdf

In order to understand the nuances of the PPO approach and to also understand the DPO algorithm (since its authors justify DPO on the basis of how it performs vis-a-vis PPO), you need to get to know the fundamental vocabulary of RL.

Preamble (contd.)

In this lecture, in order to balance the competing requirements of a good presentation of the PPO algorithm and a good presentation of the fundamental definitions of RL, I have given a higher priority to the former.

For that reason, the lecture begins with PPO.

However, the degree to which you will be able to understand the PPO material depends on how comfortable you already are with the fundamental concepts of RL. If you are not, the intro material that starts with Section 4 should be your starting point before you start digging into PPO in the first three sections.

About the intro material that starts with Section 4, note that traditional reinforcement learning has dealt with discrete state spaces. Consider, for example, learning to play the game of tic-tac-toe. Each legal arrangement of X's and O's in a 3×3 grid constitutes a discrete state. One can show that there is a maximum of 765 states in this case. (See the Wikipedia page on "Game Complexity".)

What's interesting is that the neural-network based formulations of RL can be used to solve the learning problems even when the state spaces are continuous and when a forced discretization of the state space would result in unacceptable loss in learning efficiency.

Preamble (contd.)

The intro material that starts with Section 4 first takes up what is known as Q-Learning in RL. I'll illustrate Q-Learning with a couple of implementations and show how this type of learning can be carried out for discrete state spaces and how, through a neural network, for continuous state spaces.

Toward the end of this lecture, I'll review the concept of policy based methods for RL. In particular, I'll briefly go over an application of RL for improving the quality of an e-commerce search engine with human feedback.

For a good understanding of the PPO material at the beginning of this lecture, you need to have understood how the Policy Gradient methods work, as explained at the end of lecture.

Preamble – How to Learn from These Slides

It is possible that your initial reaction to the order in which I have presented the concepts in this lecture is going to be negative.

I have front-loaded the presentation with the concepts that dive immediately into how you can use RL for incorporating human feedback in an LLM. A more natural way would have been to push that material to the end of the lecture where it actually belongs.

My reason was that is that RL is now being taught in more than one ML class at Purdue and it is not uncommon for the students in my class to have had previous exposure to RL. Such students would have been extremely bored by a revisit to the basic definitions of RL before launching into what's needed for LLM fine-tuning.

Nevertheless, if you have had zero prior exposure to RL, I'd highly recommend that you first make yourself familiar with the basic definitions on Slides 34 through 52 before tackling the material on Slides 9 through 33.

Outline

Outline

RLHF Notation

- For explaining Reinforcement Learning for Human Feedback (RLHF), I am going to use the notation I have seen in the publication "Fine-Tuning Language Models from Human Preferences" by Ziegler et al.:

  https://arxiv.org/pdf/1909.08593.pdf

- I'll use Σ to denote the tokenizer vocabulary and Σ_n to denote the set of sequences of n or fewer tokens taken from the set Σ. Each token sequence in Σ_n may be displayed as $<c_0, c_1, \ldots, c_{n-1}>$ where $c_i \in \Sigma$ is a token — ignorning for a moment that the length of the sequence may be shorter than n.

- Any exercise in language modeling through, say, unsupervised learning will induce a probability distribution over the token sequences in Σ_n. We will express these probabilities by $\rho(c_0, c_1, \ldots, c_{n-1})$.

- Obviously, ρ, when expressed as a function of the n arguments as shown, is the joint probability of the individual tokens showing up in the order indicated.

RLHF Notation (contd.)

- For Autoregressive Modeling of the language, the joint distribution can be expressed as a product of the the marginals for each token in the sequence subject to all the tokens previously seen.

$$\rho(c_0, c_1, \ldots, c_{n-1}) \quad = \quad \prod_{k=0}^{n-1} \rho(c_k \mid c_0, c_1, \ldots c_{k-1}) \tag{1}$$

- We will now imagine a user interacting with the language model with a collection of prompts that come from the set $\mathbf{X} = \Sigma_m$ of prompts.

- A prompt will consist of a sequence of tokens with the proviso that the number of tokens in a prompt be no longer than m. The notation Σ_m merely says that the tokens are being drawn from the vocab Σ and that the sequences formed are limited in length to m.

- The set of all responses to the prompts will be denoted $\mathbf{Y} = \Sigma_n$, implying that the response to a prompt will be a sequence of n tokens.

RLHF Notation (contd.)

- As an example of the difference between the sets $\mathbf{X}$ and $\mathbf{Y}$, a prompt $\mathbf{x} \in \mathbf{X}$ could be an article of length up to 1000 tokens and the response $\mathbf{y} \in \mathbf{Y}$ could be a 100-token summary.

- **Let us now examine what the probability distribution ρ induced by unsupervised learning has to say about the probabilistic relationship between the prompts and their responses.**

- We could pose the following question to the learned language model: Given a prompt sequence $\mathbf{x}$ and assume that we want the model to return a meaningful token continuation of $\mathbf{x}$ in the form of a response sequence $\mathbf{y}$, what is the probability $\rho(\mathbf{xy})$ vis-a-vis the probability $\rho(\mathbf{x})$. The following conditional probability captures the ratio of the two probabilities:

$$\rho(\mathbf{y}|\mathbf{x}) \quad = \quad \frac{\rho(\mathbf{xy})}{\rho(\mathbf{x})} \tag{2}$$

RLHF Notation (contd.)

- In principle, if we have access to the probability distribution ρ as defined in Eq. (1) on Slide 11, we should be able to estimate both the $\rho(\mathbf{xy})$ and $\rho(\mathbf{x})$ probabilities for the numerator and the denominator in Eq. (2). So we should be able to assess the appropriateness of a given $\mathbf{y}$ as a continuation of the prompt $\mathbf{x}$.
- As you will see later, the main idea in RL is to learn a *policy*, which, in general, is a mapping from the different possible *states* of the *environment* to the *actions* that an *agent* can take in order to learn how to best get to a presumed *goal* vis-a-vis the *environment*.
- Any *action* chosen by the agent will change the *state* of the environment in a way that is beyond the understanding of the agent.
- The environment is our case is the LLM and agent is the computer algorithm whose job is to make small changes to the learnable weights in the LLM so that the LLM returns the best possible responses to the human-supplied prompts.

RLHF Notation (contd.)

- To continue the thought in the last bullet on the previous slide, the algorithm (meaning, the agent) needs to learn a *policy* that would tell the algorithm what changes to make to the LLM after gauging its state.
- In general, the optimum policy is discovered iteratively through an attempt to maximize an estimate of the *future expected rewards*.
- We can start this iterative discovery of the optimum policy by assuming that the probability distribution ρ is itself (implicitly) a policy for the start. So if we use π to denote the policy that is being learned, we can set $\pi = \rho$ initially.
- That begs the question: How can ρ, meant to be a probabilistic model of the language as learned during unsupervised pre-training, serve as a policy that is meant to be a mapping from the states of the LLM to the actions of the algorithm that is trying to improve the performance of the LLM.

RLHF Notation (contd.)

- Before I answer the question at the bottom of the previous slide, let's first get a notational fix on what exactly is meant by a reward.
- We can formulate the reward as a mapping from the cross-product space $\mathbf{X} \times \mathbf{Y}$ to the set $\mathcal{R}$ of real numbers. That is, $r : \mathbf{X} \times \mathbf{Y} \rightarrow \mathcal{R}$. We are obviously modeling a concatenation like $\mathbf{xy}$, where $\mathbf{x}$ is a prompt and $\mathbf{y}$ a continuation of the prompt, as *a state of the environment*. The entity $\mathbf{xy}$ is an element of the cross-product space $\mathbf{X} \times \mathbf{Y}$.

  [**More accurately stated, we can use a set of concatenations like xy as the current state of the LLM.**]
- We can now write the following formula for the Future Expected Reward

$$\mathrm{E}_{\pi}[r] \quad = \quad \mathrm{E}_{\mathbf{x} \in \mathbf{X},\ \mathbf{y} \sim \rho(\mathbf{y}|\mathbf{x})}[r(\mathbf{x}, \mathbf{y})] \tag{3}$$

- Although this formula for the Expected Reward is cool to look at, it does not directly help us incorporate human preferences in the LLM.
- **What we need next is a way to model the reward $r(\mathbf{x}, \mathbf{y})$ itself.**

RLHF Notation (contd.)

- In order to set us up for modeling the Future Expected Reward $\mathbb{E}_\pi[r]$, let me quickly review how the human input would be processed for improving the LLM.
- The human would query the LLM with a set of prompts and then, for each prompt, the LLM would sample its distribution ρ, which is a probabilistic model of the LLM, for a small set of responses. For example, if $\mathbf{x} \in \mathbf{X}$ is a prompt, the sampling of ρ might yield the following four different responses $\{\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \mathbf{y}_4\}$, with each $\mathbf{y}_i \in \mathbf{Y}$.
- Subsequently, the human would be asked to choose the preferred response amongst the four returned by the LLM. Following Ziegler et al., let's use the notation $\mathbf{b} \in \{1, 2, 3, 4\}$ to denote the preferred response $\mathbf{y}_b$.
- We can package this interaction between the human and the LLM in the form of sextuples like $(\mathbf{x}, \mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \mathbf{y}_4, \mathbf{b})$. The question now is: How do we use these sextuples for constructing a model for the reward?

Outline

Modeling the Reward for PPO

- Following the discussion on Slides 15 and 16, modeling the reward means estimating the mapping $r : \mathbf{X} \times \mathbf{Y} \rightarrow \mathcal{R}$ that reflects human preferences. Let's use the notation $r(\mathbf{x}, \mathbf{y})$ to represent this function.
- To see how that can be done, let's go back to the sextuples like $(\mathbf{x}, \mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \mathbf{y}_4, \mathbf{b})$ defined on Slide 16.
- As the reader will recall, the sextuple shown above means that for a prompt $\mathbf{x} \in \mathbf{X}$, we sampled the probabilistic model ρ of the LLM to receive the four different responses $\{\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \mathbf{y}_4\}$, with each $\mathbf{y}_i \in \mathbf{Y}$, and that a human evaluator chose the response indexed $\mathbf{b} \in \{1, 2, 3, 4\}$ as their preferred one.
- Given a set of sextuples like $(\mathbf{x}, \mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \mathbf{y}_4, \mathbf{b})$ for the purpose of modeling the reward, the issue we are now facing is how to set up a criterion for estimating the optimum reward function $r(\mathbf{x}, \mathbf{y})$ that would maximize the "weight" given to the human preferences as expressed by the choice of the index $\mathbf{b}$ in each sextuple?

Modeling the Reward for PPO (contd.)

- To see how we can set up a criterion along the lines mentioned at the bottom of the previous slide, you have to first appreciate the fact that for a given prompt $\mathbf{x}$, the following four ratios will add up to 1.0 regardless of the precise form of the reward function $r(\mathbf{x}, \mathbf{y})$:

$$\sum_{b_i \in \{1,2,3,4\}} \frac{e^{r(\mathbf{x}, \mathbf{y}_{b_i})}}{\sum_{k \in \{1,2,3,4\}} e^{r(\mathbf{x}, \mathbf{y}_k)}} \quad = \quad 1.0 \tag{4}$$

- What that implies that the following ratio for the specific preference $\mathbf{b}$ expressed by a human can be interpreted as the probability of that choice vis-a-vis the other possible choices.

$$\frac{e^{r(\mathbf{x}, \mathbf{y}_b)}}{\sum_k e^{r(\mathbf{x}, \mathbf{y}_k)}} \tag{5}$$

- For obvious reasons we would want the reward function $r(\mathbf{x}, \mathbf{y})$ to give as large a value to this preference as possible — taking into account the other preferences expressed by the same human or by others.

Modeling the Reward for PPO (contd.)

- On the strength of the claims made on the previous slide, we can devise a loss whose minimization should yield the reward function we are looking for.
- Let's assume that we have collected M number of sextuples $(\mathbf{x}, \mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \mathbf{y}_4, \mathbf{b})$ that capture the human preferences we are going to use to further fine-tune the LLM. Presumably, M is large.
- Using these M sextuples, our goal is to find the reward function $r(\mathbf{x}, \mathbf{y})$ that minimizes the reward modeling loss given by:

$$Loss \quad = \quad -\sum_{j=1}^{M} \log \frac{e^{r(\mathbf{x}, \mathbf{y}_b)}}{\sum_k e^{r(\mathbf{x}, \mathbf{y}_k)}} \tag{6}$$

- Just to give you an idea of what sort of parameters were used by Ziegler et el. to minimize this loss:

"... the reward model is trained using the Adam optimizer with the loss shown above. The batch size is 8 for style tasks and 32 for summarization, and the learning rate is 1.77×10^{-5} for both." Also, the authors trained for "a single epoch to avoid overfitting to the small amount of human data, and turned off dropout."

Modeling the Reward for PPO (contd.)

- Just to make it explicit, the reward function $r(\mathbf{x}, \mathbf{y})$ you get by the method described so far is a neural network. You feed the string $\mathbf{x}, \mathbf{y}$ into it, and it outputs the reward as a number.
- You may still ask as to why the $r(\mathbf{x}, \mathbf{y})$ function we get by the minimization of the loss shown on the previous slide constitutes a Future Expected Reward in the sense that phrase is used in RL.
- The reason as to why $r(\mathbf{x}, \mathbf{y})$ constitutes a reward is that, as the policy adapts itself increasingly to reflect the human preferences, the value returned by $r(\mathbf{x}, \mathbf{y})$ would get larger. That follows from the discussion on the previous two slides.

Outline

Proximal Policy Optimization (PPO)

- Now that we have a model for the Future Expected Reward, we need to get to solving the problem of updating the LLM with RL.
- As you can see in the depiction on the next slide, the model shown at left is the initial LLM model you obtained with unsupervised training. In the middle of the figure, you start with that model as your initial policy model. The policy model evolves iteratively during PPO based fine-tuning while the initial model remains unchanged.
- I'll continue to use ρ for a probabilistic representation of the initial model. And π_θ for a probabilistic representation of the policy model as a function of the learnable parameters θ of the LLM that will be updated by PPO. **At the end of the fine-tuning exercise, π_θ would become the representation of the fine-tuned version of the LLM.**
- In addition to the two networks, one for the initial LLM model and the other for the policy model, you also have the Reward Model network that works as described in the previous section.

PPO for RLHF (contd.)

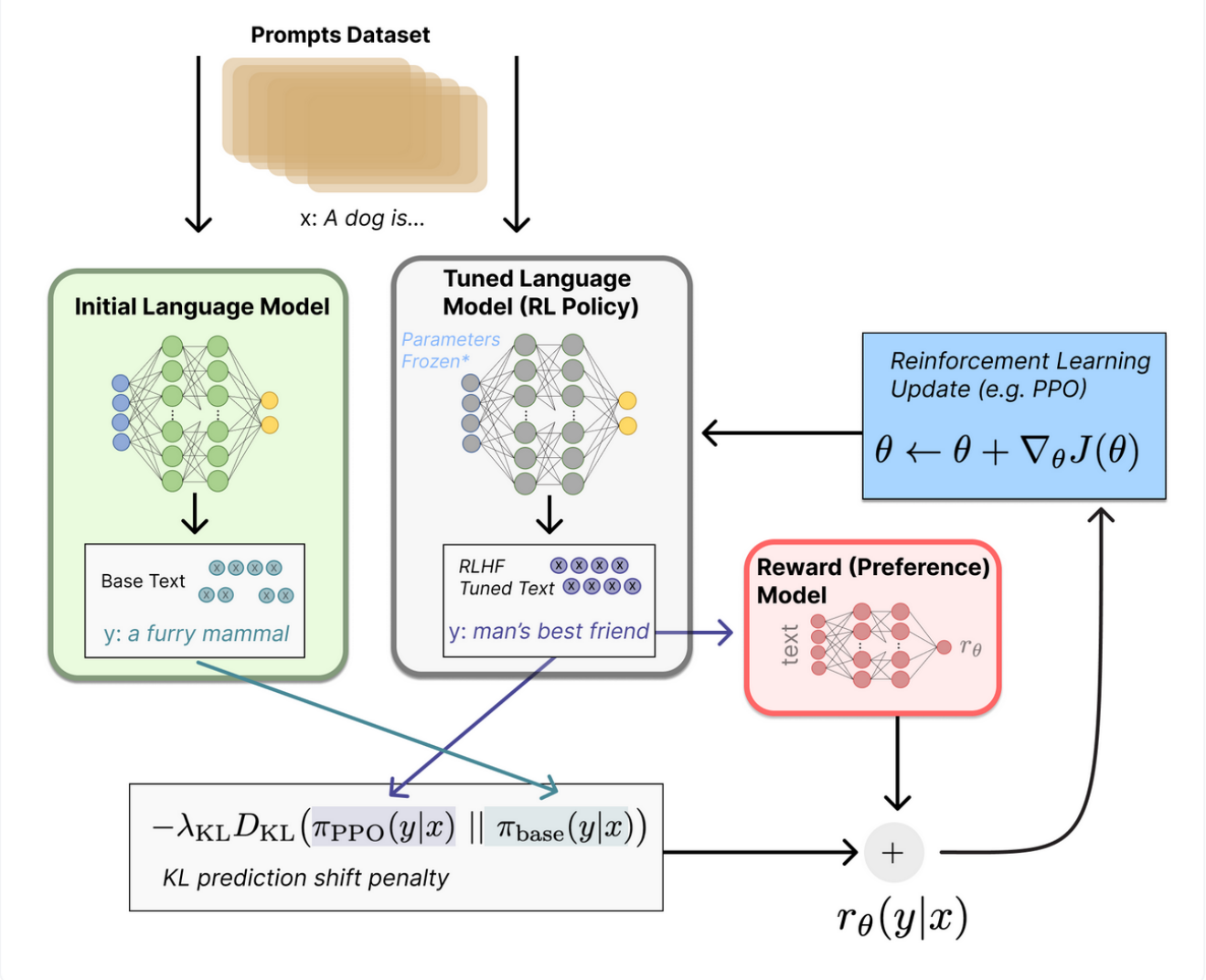

Figure: This figure is from the Hugging Face tutorial at `https://huggingface.co/blog/rlhf`.

PPO for RLHF (contd.)

- If the initial model is too large, you may subject only a part of the policy model to fine-tuning with PPO. The learnable weights in the rest would remain frozen during the fine-tuning exercise.
- The goal of PPO-based RLHF is to update the policy model and the reward model together in multi-epoch training.
- For stable fine-tuning of the initial model, we make sure that the policy model π_θ does not stray too far from the initial model ρ. To bring that about, we add a penalty term to the updating of the Reward Model as shown below. The notation $R(\mathbf{x}, \mathbf{y})$ represents the updated version of the initial reward model $r(\mathbf{x}, \mathbf{y})$ estimated as discussed in the previous section:

$$R(\mathbf{x}, \mathbf{y}) \quad = \quad r(\mathbf{x}, \mathbf{y}) \quad - \quad \beta \log \frac{\pi_\theta(\mathbf{y}|\mathbf{x})}{\rho(\mathbf{y}|\mathbf{x})} \tag{7}$$

- The coefficient β is set so that the KL-Divergence between the initial model ρ and the evolving policy model π_θ does not exceed a threshold. See the Ziegler et al. paper for how to set β.

PPO for RLHF (contd.)

- To understand how the PPO algorithm actually updates the policy π_θ, you need to understand what we mean by the Advantage Function that is closely related to the reward function.
- For a given policy π_θ, the Advantage Function gives us **a measure how much better any given action would be in relation to all other possible actions available to the agent.**
- If we measure the quality of a specific candidate action *a* by the value of the expected reward that is to be had through the state achieved by that action, and we somehow get the average of the same measure for all other possible actions in the same state, the Advantage Function would return the difference between the two values.
- The implementation of the Advantage Function is straightforward. It starts out as being the reward function directly that was described in the previous section $R(\mathbf{x}, \mathbf{y}) = r(\mathbf{x}, \mathbf{y})$.

PPO for RLHF (contd.)

- Subsequently, for each possible extension **y** to **x**, all you have to do calculate the difference of the rewards for each choice **y** vis-a-vis all other choices. And you do the same with the updated rewards.
- Now that you understand the basic idea of the Advantage Function, we must next concern ourselves with actual fine-tuning of the language model — that is, with how to update the learnable weights in the policy network, which is the middle neural network on Slide 24.
- Updating π_θ requires great care because experience with RL has shown that a sudden change caused by the gradient ascent logic can quickly derail the fine-tuning proess.
- The care required in updating π_θ translates into using "clipped gradients" of the learnable parameters.
- The updating of π_θ is carried out through a stochastic gradient **ascent** algorithm as explained at the end of this lecture where I have introduced the Policy Gradient Approach to RL.

PPO for RLHF (contd.)

- With s_t representing the state at the iteration index t and a_t representing the action in that state, Stochastic Gradient Ascent (SGA) based update algorithm for the updating of π_θ is based on first calculating the ratio of the following probabilities:

$$ratio_t(\theta) \;=\; \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{t-1}}(a_t|s_t)} \tag{8}$$

- The gradient of the policy function π_θ would then be given by the following expression where A_t is the value returned by the Advantage Function for the action a_t.

$$\hat{g} \;=\; \mathrm{E}_t\Big[\nabla_\theta \min\Big[ratio_t(\theta)\cdot A_t,\quad clip[ratio_t(\theta),\;1-\epsilon,\;1+\epsilon]\cdot A_t\Big]\Big] \tag{9}$$

- The expectation shown above is carried out over the samples in the batch. The `min` function has to choose the smaller of the $ratio_t(\theta)\cdot A_t$ and a version of the same that is clipped to lie in the interval $[1-\epsilon, 1+\epsilon]$.

PPO for RLHF (contd.)

- If you want to automatically estimate the gradients shown on the previous slide with, say, the Adam optimizer, you are going to need to express the gradient shown there in the form of the following Objective Function:

$$\mathcal{L}^{clip}(\theta) \;=\; \mathrm{E}_t\Big[\min\Big[ratio_t(\theta)\cdot A_t,\quad clip[ratio_t(\theta),\;1-\epsilon,\;1+\epsilon]\cdot A_t\Big]\Big] \qquad (10)$$

- It is the updating of the learnable parameters of the language model through the objective function shown above that is the job of the neural network shown in the middle in Slide 24.

- Regarding the clipping action, first assume the case when the Advantage A_t is positive. In this case, we want to be disposed towards choosing the action a_t. However, the objective function for the estimation of the gradients by the Adams optimizer will receive a version of $ratio_t$ that is guaranteed to NOT be outside the clipping interval.

PPO for RLHF (contd.)

- The last bullet on the previous slide implies is that the calculated gradients will not be allowed to be too large for considering the action a_t. Therefore, there cannot be too large a change to our policy — something that was a big problem with the earlier Policy Gradient method described at the end of this lecture.
- For the case when the Advantage A_t is negative, now we want to disfavor the action a_t. But, again, the degree to which this action will get discounted through a reduced probability will be limited by the clipping action.
- For further information, visit the OpenAI documentation page on PPO at:

  `https://spinningup.openai.com/en/latest/algorithms/ppo.html`

  Click on the "Pseudocode" button at the above link to see the call to the Objective Function for the Adam optimizer as presented on the previous slide.

Implementing PPO for RLHF – Pseudocode

- Let's now talk about how one would go about implementing PPO for a Base Language Model as learned by, say, babyGPT presented in my Week 14 lecture.
- The first thing we need to do is construct "episodes" with each episode being a sequence of triplets (s_t, a_t, r_{t+1}) for $t < T$ where T is a user-defined value for what's known as the "Horizon". In general, the Horizon is the maximum number of steps in the interaction between the agent and the environment, which would be fine-tuning algorithm and the Language Model in our case. Each episode returns the sum of the discounted expected rewards accumulated during the episode.
- On Slide 15, I defined state s_i as the concatenation xy if x is the prompt and y the response. You could say that we figure out the current state of the language model by prompting it with a string x of our choice (using x like a probe you might say) and seeing the response y returned by the model.

PPO in Pseudocode (contd.)

- So if x_1 and y_1 are two elements in first such pair, we can declare the first state to be $s_1 = (x_1, y_1)$. Subsequently, our action a_1 would be to concatenate the two strings in s_1 to yield the prompt $x_2 = x_1 y_1$ for forming the next triplet in the sequence of triplets mentioned on the previous slide.

- For each <state, action> pair, we must also collect the reward r_{t+1} in the triplets thus constructed. We get the rewards from the Reward Network mentioned earlier and displayed by the rightmost network on Slide 24.

- That gives us all the ingredients for implementing the PPO framework according to the psuedocode shown on the next slide for the purpose of fine-tuning the base language model with human preferences.

PPO in Pseudocode (contd.)

- At the link below, you can see Yuki Minai's version of the OpenAI pseudocode mentioned in the previous slide:

https://medium.com/@ym1942/proximal-policy-optimization-tutorial-f722f23beb83

- An episode in Line 4 is a sequence of triplets (s_i, a_i, r_{i+1}). See previous slides for the definitions of the states s_i and actions a_i. The notation r_{i+1} is for the rewards expected from the environment.

```
Algorithm:  Episodic PPO

1.  Initialize baseline network \rho                                  ##  The leftmost network on Slide 24
2.  Initialize policy network \pi_theta                               ##  The middle network on Slide 24
3.  for iteration = 1,2, ...., num_episodes,  do
4.    Generate an episode s0,a0,r1, s1,a1,r2, s2,a2,r3, ... s_T-1, a_T-1, r_T  through policy \pi_theta
5.    for t = 0,1,2, ..., T-1, do
6.        G_t  =  \SUM_{k = t+1}^T  \gamma^{k-t-1} R_k              ## G_t is total future expected reward at t
                                                                    ## \gamma is the reward discounting factor
7.    end for
8.    Compute advantage estimates  A_t  =  ( G_t  -  \rho( s_t ) )
9.    for epoch = 1, 2, ... , num_epochs  do
10.     ## Compute the objective function:
11.     L^CLIP (\theta) =  E_\pi [ min[ r(theta) . A_t,   clip( r(theta), 1-\epsilon, 1+\epsilon ) . A_t ] ]
12.     L(w)            =  - 1/T  \SUM_{t=0}^{T-1}  ( G_t  -  V_w (S_t) )^2
13.     Update \pi_theta using Adam( \grad_theta L^CLIP (theta) )
14.     Update \rho using Adam( \grad_w L(w) )
15.   end for
16. end for
```

Outline

Reinforcement Learning — Some Basic Terminology

- Here is an alphabetized list of the more basic elements of the vocabulary of reinforcement learning (RI): *action*, *agent*, *environment*, *episode*, *goal*, *penalty*, *policy*, *reward*, *state*, and *state transition*.
- At any given time, the *environment* is in a particular *state*.
- The agent interacts with the environment through a *policy* that it is constantly refining on the basis of the rewards received from the environment. The *policy* tells the agent what *action* to invoke when the environment is in a particular *state*.
- You could say that the entire goal of an RL based solution to a problem is the learning of the best policy. As to what exactly is meant by "best policy" will soon be clear.

[**Here is an interesting asymmetry: While the agent can change the state of the environment through its policy-suggested action, as to what state the environment transitions into as a result of that action depends entirely on, say, the laws of physics or the laws of economics, etc. It is the environment that knows about those laws. The agent can only react to each state of the environment by executing an action, but the consequences of that action on the environment are beyond the agent's reckoning.**]

Basic Terminology (contd.)

- In general, when the *agent* takes an *action*, that can change the state of the *environment* and elicit a *reward* or a *penalty* from the environment. The reward may be positive or negative or zero. A zero or a negative reward constitutes a penalty.
- A sequence of state transitions caused by the agent's actions constitutes an episode if there does not exist a state transition from the terminal state in the sequence.
- *The overall goal of reinforcement learning is for the agent to learn the best possible policy whose action choices maximize the rewards received from the environment during each episode.*
- At all times, the *agent* is aware of the *state* of the *environment*. The agent also knows what *actions* it has at its disposal for changing that state. But what the agent does not know in advance is what action to deploy in each state in order to maximize its rewards from the environment during each episode. That it must learn on its own.

The Basic RL Notation

- Another way of saying the same thing as in the last bullet on the previous slide: While the agent can use its policy to decide what action to invoke in response to the current state of the environment, it cannot predict the consequences of that action. [Read again the note at the bottom of Slide 34.]
- Initially, the agent's policy chooses the actions randomly. Subsequently, based on the rewards received, the agent refines its policy in order to make the best choice for an action in each state of the environment.
- It is common to represent all possible states of the environment by the set $S = \{s_1, \ldots, s_n\}$ and the set of actions that agent can execute in order for the environment to transition from one state to another by $A = \{a_1, a_2, \ldots, a_m\}$.

The State, Action, Reward Sequence

- I'll use the symbol σ_t to denote the state of the environment at time t. Obviously, $\sigma_t \in S$. The action taken by the agent at time t would be denoted α_t. Obviously, again, $\alpha_t \in A$.
- The action α_t taken by the agent at time step t will cause the state of the environment to change to $\sigma_{t+1} \in S$ for the time step $t+1$.
- In response to this state transition, the environment will grant the agent a reward r_{t+1}. A reward is typically a small scalar value.
- After receiving the reward r_{t+1} while the environment is in state σ_{t+1}, the agent takes the action $\alpha_{t+1} \in A$. This action will cause the state of the environment to change to $\sigma_{t+2} \in S$. And that will be followed by the environment granting the agent a reward r_{t+2}, and so on.
- Assuming that the agent starts learning at time $t = 0$, we can think of the following sequence of states, actions, and rewards:

$$\sigma_0, \alpha_0, r_1; \quad \sigma_1, \alpha_1, r_2; \quad \sigma_2, \alpha_2, r_3; \quad .....$$

Visualizing the Learning Processing

- An agent learning in this manner in response to the rewards received from the environment may be visualized in the following manner:

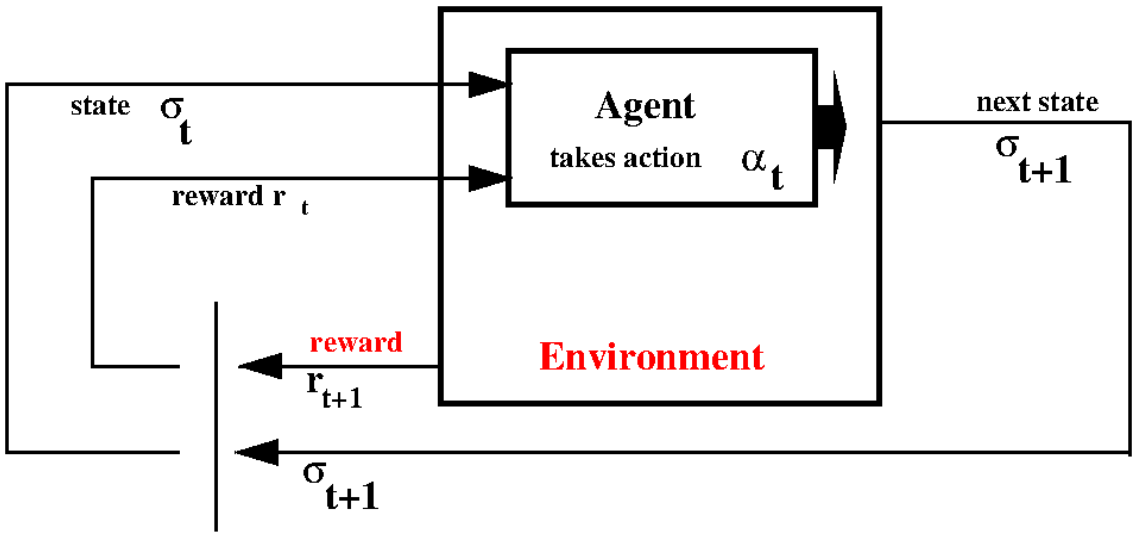

Figure: The Agent's action $\alpha_t \in A$ when the environment is in state $\sigma_t \in S$ results in the state transitioning to σ_{t+1} and the environment granting the agent a reward r_{t+1} (which can be negative).

Outline

A Stochastic RL Agent

- The notation of Reinforcement Learning (RL) I presented in the previous section was sterile — in the sense that it might have created the impression that the relationships between the states, the actions, and the rewards were deterministic and designed to guarantee success.
- Such determinism cannot model the real-life scenarios in which one would want to use RL.
- You see, the environment has intractable aspects to it that can only be modeled by a probabilistic relationship between the current state and the consequences of any given action taken by the agent when the environment is in that state. In other words, there are uncertainties associated with our understanding of the environment.
- How to best model these uncertainties is an area of research unto itself. The models that have become the most popular are based on the usual Markovian assumption as shown on the next slide.

The Markovian Assumption

- With the Markovian assumption, you have a stochastic RL agent for which the rewards received from the environment and also the state transition achieved by the environment at time $t+1$ depend probabilistically on just on the state/action pair at time t.
- To be more precise, the state/action pair at time t completely dictates a probability distribution for the next-state/reward pair at time $t+1$. We denote this probability as follows:

$$prob(\sigma_{t+1}, r_{t+1} \mid \sigma_t, \alpha_t)$$

- Such an agent along with the environment is known as a Markov Decision Process (MDP).
- Such a probability distribution would allow us to estimate the conditional probabilities related to the state transitions:

$$prob(\sigma_{t+1} \mid \sigma_t, \alpha_t) = \sum_r prob(\sigma_{t+1}, r \mid \sigma_t, \alpha_t)$$

Future Expected Reward

- In a similar manner, we can also compute the expected reward at time $t+1$:

$$E(r_{t+1} \mid \sigma_t, \alpha_t) = \sum_r r \sum_{\sigma_{t+1}} prob(\sigma_{t+1}, r \mid \sigma_t, \alpha_t)$$

  where the outer summation on the right is with respect to all possible reward values when the environment is in state σ_t and the agent is invoking action α_t. The inner summation is with respect to all possible target states that the system can get to from the state σ_t.

- With respect to time t, the estimate $E(r_{t+1} \mid \sigma_t, \alpha_t)$ is a future expected reward for time $t+1$.

- In general, the agent's policy should be such that the action taken in each state maximizes the **future expected reward** from the environment.

The Quality Index Q

- Toward that end, we can associate with each (s_i, a_j) pair a quality index, denoted $Q(s_i, a_j)$, that is a measure of the maximum possible value for the future expected reward from the environment for that pair. That is, by definition,

$$Q(s_i, a_j) \quad = \quad \max E\Big(r_{t+1} \;\; | \;\; \sigma_t = s_i, \alpha_t = a_j\Big)$$

- The best policy for the agent, naturally, would be to choose that action $a_j \in A$ in state $s_i \in S$ which maximizes the quality index $Q(s_i, a_j)$.

- The conditioning shown on the right in the equation above means that assuming that σ_t, which is the state of the environment at time t, was s_i and assuming that α_t, the action chosen by the agent at the same time, was a_j, then if we *could* estimate the maximum possible value of the expected future reward at time $t+1$, that *would* be the value of the quality index $Q(s_i, a_j)$ at time t.

The Quality Index Q (contd.)

- The goal of RL should be to estimate the values for quality index $Q(s_i, a_j)$ for all possible state/action pairs (s_i, a_j).
- We can conceive of Q as a table with $|S|$ rows and $|A|$ columns, where $|.|$ denotes the cardinality of a set. We could place in each cell of the table the quality-index value we may conjure up for $Q(s_i, a_j)$.

Outline

What is Q-Learning?

- The probabilistic notions of the previous section serve a useful purpose in helping us articulate a principled approach to reinforcement learning. However, in practice, it is rather unlikely that we will have access to the conditional probability distribution mentioned on Slide 41 for a real life problem, not the least because of the intractable nature of the uncertainties in the interaction between the agent and the environment.

- Q-Learning is a method for implementing an RL solution for a learning agent that attempts to approximate the maximization of the future expected rewards in the absence of explicit probabilistic modeling.

- In Q-Learning, we attempt to place in the cells of the Q table the estimated values for the best expected cumulative rewards for the state-action pairs corresponding to those cells.

Maximizing the Sum of the Discounted Rewards

- Fundamentally, the entries in the cells of a Q table are the expected future rewards. For practical reasons, we must incorporate the future into these estimates on a discounted basis.
- You can think of discounting as reflecting our lack of knowledge of the conditional probability distributions mentioned in the last section. That is, we cannot be too sure about how the choice of an action in the current state will influence the future outcomes as the agent makes state-to-state transitions. With this uncertainty about the future, a safe strategy would be to deemphasize the expected rewards that are more distant into the future.
- Discounted or not, there is a troubling question here: Without any probabilistic modeling of the sort mentioned in the previous section, how can any agent look into the future and make any reasonable sense of what should go into the Q table? As to how practical implementations of Q-Learning get around this dilemma, we will get to that later.

Updating the Q Table (contd.)

- We will ignore for a moment the last bullet on the previous slide, and acknowledge the recursive nature of the definition of $Q(s_i, a_j)$ on Slide 43 when that definition is considered as a function of time. That definition is recursive because the future expected reward at t depends on its value at $t+1$ in the same manner as the future expected reward at $t+1$ depends on its value at $t+2$, and so on.
- By unfolding the recursion and incorporating the discounting mentioned on the previous slide, $Q(s_i, a_j)$ can be set to:

$$\begin{aligned} Q(s_i, a_j)\Big|_t &= \max\left(r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \ldots\right) \\ &= \max\left(\sum_{k=0}^{\infty} \gamma^k \cdot r_{t+1+k}\right) \end{aligned}$$

where γ is the discounting factor and where we intentionally do not start discounting the rewards until the time step $t+2$ (assuming that the current time is t) as a reflection of our desire to base the value of Q primarily on the next time step and of our wariness about the more distant future.

Updating the Q Table

- For the purpose of implementation, we express the update equation shown on the previous slide in the following form:

$$Q(s_i, a_j)\Big|_t \quad \Leftarrow \quad (1-\beta) * Q(s_i, a_j)\Big|_t \quad + \quad \beta * \Big[r_{t+1} + \gamma \cdot \max_{a_j \in A} \Big\{ Q(s_i, a_j)\big|_{t+1} \Big\}\Big]$$

  where the β is the learning rate. Note the assignment operator '$\Leftarrow$', which is what makes it an update formula.

- Calling on the RL Oracle to provide the agent with what would otherwise be unknowable, we can reexpress the above update formula as:

$$Q(s_i, a_j)\Big|_t \quad \Leftarrow \quad (1-\beta) * Q(s_i, a_j)\Big|_t \quad + \quad \beta * \Big[r_{t+1} + \{\text{Consult the RL Oracle for the best estimate of the future rewards}\}\Big]$$

- In the absence of consultation with an oracle, we could adopt the following strategy: The agent would start with random entries in the Q table. For obvious reasons, at the beginning the agent will very quickly arrive at the terminal states that indicate failure.

Updating the Q Table (contd.)

- Each failure will result in updating the Q table with negative rewards for the cells that cause transitions to such terminal states. This would eventually — at least one would so hope — result in a Q table that would reflect positive learning.

- To elaborate further, for any state transition to failure, the value of the reward r_{t+1} in an update formula of the type shown on the previous slide would be a large negative number. That would push down the entry in the $Q(s_i, a_j)$ table that led to such a transition, making it less likely that the transition would be taken the next time.

- In algorithm development, we talk about "time-memory tradeoff". What we have here is a "time-foresight tradeoff". [About what's meant by "time-memory" tradeoff: What that means is that you can solve a problem by having memorized or theoretically generated all of your solution paths and then, when confronted with a new instance of the problem, you simply look up the memorized strategies. That would be referred to as a purely memory based solution. An alternative approach would consist of only remembering the decision points in seeking out a solution to the problem. Then, when confronted with a new instance of the problem, you would step through the solution space and test which way you should turn at each decision point. In this case, you would be using lesser memory, but it would take longer to generate a solution.]

Updating the Q Table (contd.)

- I say "time-foresight tradeoff" because we do not have access to an oracle. We therefore choose to stumble into the future anyway through the state-transition table and learn from our mistakes as recorded in the evolving Q table. It may take time, but, hopefully, we'd still be able to solve the problem.
- For stumbling forward in the learning process, we rewrite the formula on Slide 49 as follows in which we estimate the future rewards by stepping through the state transition table and accumulating the future rewards by looking up the Q table:

$$Q(s_i, a_j)\Big|_t \Leftarrow (1-\beta)*Q(s_i, a_j)\Big|_t + \beta*\{\text{Estimate discounted future reward from Q-table and state-transition table}\}$$

- This update formula lends itself to an actual implementation in which you start with random entries in the Q table and, through repeated trials, you converge to the table that reflects positive learning. I have demonstrated this with the two implementations that follow in this lecture, one based on a discrete state space and the other on a continuous state space.

Outline

The Cart-Pole Example

- To explain Q-Learning with an example, consider what's known as the Cart Pole problem. The Cart-Pole problem is to RL what fruit-fly is to genetics research.
- In the Cart Pole problem, you have a cart that can move back and forth on a linear track. Mounted on the cart is a hinged pole. The hinge allows the pole to turn in the same plane that corresponds to the translational motions of the cart.
- The goal is to keep the pole standing upright and, should the pole lean one way or the other, the cart should move in such a way that counters the lean of the pole.
- Considering how unstable the pole would be standing upright, it would be difficult to construct a probabilistic framework for this problem along the lines described earlier in this lecture.

The Cart-Pole Example (contd.)

- The state-space for the Cart Pole problem is obviously continuous since the pole can lean to any degree.
- The question we pursue in this section is whether it is possible to discretize the state space and to then use the Q-Learning based RL as described in the previous section.
- Shown below for illustration is a 5-state discretization of the state space. Each entry represents the lean of the pole one way or the other, but within a range appropriate to the entry:

```
S  =  {  hard_lean_left,
         soft_lean_left,
         soft_lean_right,
         hard_lean_right,
         fallen
      }
```

The Cart-Pole Example (contd.)

- Here is a possible mapping between the next-state achieved and the ensuing reward as required by the update formula shown on Slide 49:

```
fallen              =>   -1
soft_lean_left      =>   +1
soft_lean_right     =>   +1
hard_lean_left      =>    0
hard_lean_right     =>    0
```

- And here are the two actions the agent could invoke on the cart in order to balance the pole:

```
A  = {  push_left,
        push_right
     }
```

- Now that you can appreciate what I mean by the discretization of the Cart-Pole's state space, it's time to actually look at some code that implements this type of Q-Learning.

A Python Implementation of Q-Learning for the Cart-Pole Problem

- Starting with Slide 63, what you see is Rohan Sarkar's Python implementation of Q-Learning for solving a discretized version of the Cart-Pole problem. Rohan is working on his Ph.D. in RVL.
- In line (A), the code starts by initializing the various parameters and the environment variables.
- The function whose definition starts in line (B) creates a discretized set of states for the Cart Pole. The discretization is a much finer version of what I illustrated on the previous slide.
- The purpose of the function `optimal_action()` defined in line (C) is to return the best choice of action when the agent is in a given state s. This function implements what is known as the *epsilon-greedy* policy for choosing the best action in any given state. [This policy says that the earlier phases of training should make a choice between choosing an action randomly from all available actions in a given state, on the one hand, and the action offered by the Q table on the other. Furthermore, the policy says that as the training progresses we want to increasingly depend on just the Q table for choosing the best action. The parameter `eps_rate` in the header of the function plays a key role in implementing the epsilon-greedy policy. What is commonly referred to as 'training' in general machine learning is called 'exploration' in RL. And, what is commonly referred to as 'testing' in general machine learning is referred to as 'exploitation' in RL. During exploration, the value of `eps_rate` changes with each episode as follows: (1) we first multiply the current value of `eps_rate` by `EXPLORATION_DECAY`; and then (2) we assign to `eps_rate` the larger of `EXPLORATION_MIN` and the current value of `eps_rate`. When you see how `eps_rate` is used in the logic of `optimal_action()`, you will realize that that function is indeed implementing the epsilon-greedy policy. As `eps_rate` becomes ever smaller, it will become increasingly unlikely that the randomly generated value for p will be less than the current value for `eps_rate`.]

Python Implementation of Q-Learning (contd.)

- The function `updateQValues()` defined starting in line labeled (D) is called by the exploration code that is between the lines labeled (F) and (G). All of this code is for what's known as Double Q Learning (DQL) that was first proposed in the following famous paper:

  `https://arxiv.org/abs/1509.06461`

- DQL was created to counter the over-estimation bias in what's returned for the future expected reward by the Q table. [As you already know, each row of the Q table designates the state of the agent and the different columns of the table correspond to the different available actions in each state. The recommended action returned for each state is from that cell in the corresponding row which contains the highest future expected reward. This entails carrying out a maximization operation on the values stored in the different cells. In the presence of noise in the estimated rewards, this maximization is biased in the sense that it is an overestimate of the future expected reward for the chosen action.]

- DQL calls for using two Q tables, denoted $Q1$ and $Q2$ in the code between the lines labeled E and F, that are trained simultaneously with the idea that since the successive random training inputs are likely to span the permissible range, updating $Q1$ and $Q2$ alternately with the inputs will be such that the average of the rewards returned by the two tables is likely to possess reduced bias.

Python Implementation of Q-Learning (contd.)

- The testing part of the code (the exploitation part) comes after line G.
- I still have not said anything about how the agent interacts with the "environment." Actually, the role of the "environment" is a bit more complicated than what was depicted on Slide 38. That depiction only applies after an agent has been trained.
- In the code that follows, you will see the following calls that stand for the agent interacting with the environment:

```
env = gym.make('CartPole-v0')                          # at the beginning of the code file
action = env.action_space.sample()                     # in function optimal_action()
state = env.reset()                                    # in __main__
next_state, reward, done, info = env.step(action)      # in __main__
```

- In order to convey what is accomplished by the above statements, you need to understand how the Cart-Pole environment is actually implemented in code:

https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py

Python Implementation of Q-Learning (contd.)

- The code at the link at the bottom of the previous slide actually carries out a rudimentary physics based simulation of a Cart-Pole. It makes certain assumptions about the mass of the cart vis-a-vis that of the pole, about the dimensions of the cart and the pole, about the movement of the cart being frictionless, etc. The state of the cart-pole is a list of the following four elements:

```
[cart_position, cart_velocity, pole_angle, pole_angular_velocity]
```

- In terms of these elements and also in terms of the number of timesteps in any single episode during training or testing, the cart-pole is considered to have reached the terminal state (that is, when the pole is considered to have "fallen") when any of the following conditions is satisfied:
  - the pole angle is greater than 12 degrees
  - the cart position is greater than 2.4 units
  - the episode length is greater than 200 timesteps

Python Implementation of Q-Learning (contd.)

- An agent can invoke only two actions on the cart-pole: move left or move right. In each action, 10 units of force is applied to the cart in the desired direction of the movement.
- The environment also assumes that the time interval between successive interactions with it is 0.02 seconds regardless of the actual clock time involved.
- But what about the reward that is supposed to be issued by the environment? The environment issues a reward of 1 for all interactions with it. However, when the terminal condition is reached (meaning when we may think of the pole as having fallen), the environment returns that condition as "done". It is for the user program to translate "done" into a negative reward.
- In the sense indicated above, the environment itself does not judge the consequences of the actions called by the agent.

Python Implementation of Q-Learning (contd.)

- Regarding the rewards, it is for the RL logic being used by the agent to decide what rewards to associate with the actions, including the terminal action.
- Going back to the statements that are used in the code shown next in order to interact with the environment, the call `env.action_space.sample()` returns either 0 (which is the action for "move left") or 1 (which is the action for "move right") by choosing randomly between the two.
- The call `env.reset()`, which is typically how you start an episode, returns four random numbers for the four elements of the state, with numbers being uniformly distributed between -0.05 and +0.05.
- Finally, it is the call `next_state, reward, done, info = env.step(action)` that defines the interactions between the agent and the environment on an on-going basis.

Python Implementation of Q-Learning (contd.)

- With regard to the call `env.step(action)` mentioned at the bottom of the previous slide, at each time step during an episode, the agent calls on the environment with an action, which in our case is either 0 or 1, and the environment returns the 4-tuple `[next_state, reward, done, info]` where `done` would be set to the Boolean `True` if the cart-pole has reached the terminal condition.
- In the 4-tuple that is returned by `env.step(action)`, the `next_state` is again a 4-tuple of numbers standing for the 4 real numbers in the list `[cart_position, cart_velocity, pole_angle, pole_angular_velocity]`.

Python Implementation of Q-Learning (contd.)

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

##  Cartpole_DQL.v2.py

##  Balancing a cartpole using Double Q-learning (without a neural network) and using
##  a discretized state space. The algorithm is trained using epsilon-greedy approach
##  with Replay Memory
##
##  @author: Rohan Sarkar (sarkarr@purdue.edu)

import gym
import numpy as np
import time
import matplotlib.pyplot as plt
import sys
import random

seed = 0
random.seed(seed)
np.random.seed(seed)

## Initialize parameters and environment variables:
env = gym.make('CartPole-v0')                                                    ## (A)
GAMMA = 0.85
#MAX_ITER = 500
MAX_ITER = 1000
EXPLORATION_MAX = 1.0
EXPLORATION_MIN = 0.01
EXPLORATION_DECAY = 1+np.log(EXPLORATION_MIN)/MAX_ITER
REP_MEM_SIZE = 100000
MINIBATCH_SIZE = 64
N_states = 162
N_actions = 2
Q1 = np.zeros((N_states, N_actions))
Q2 = np.zeros((N_states, N_actions))
alpha = 0.001
eps_rate = EXPLORATION_MAX
cum_reward = 0
train_reward_history = []

## Map the four dimensional continuous state-space to discretized state-space:
## Credit: http://www.derongliu.org/adp/adp-cdrom/Barto1983.pdf.
def map_discrete_state(cs):                                                      ## (B)
    ds_vector = np.zeros(4);
    ds = -1
    # Discretize x (position)
    if abs(cs[0]) <= 0.8:
        ds_vector[0] = 1
    elif cs[0] < -0.8:
        ds_vector[0] = 0
    elif cs[0] > 0.8:
        ds_vector[0] = 2
    # Discretize x' (velocity)
    if abs(cs[1]) <= 0.5:
        ds_vector[1] = 1
    elif cs[1] < -0.5:
        ds_vector[1] = 0
    elif cs[1] > 0.5:
        ds_vector[1] = 2
```

(Continued on the next slide .....)

Python Implementation of Q-Learning (contd.)

(..... continued from the previous slide)

```
    # Discretize theta (angle)
    angle = 180/3.1428*cs[2]
    if -12 < angle <= -6:
        ds_vector[2] = 0
    elif -6 < angle <= -1:
        ds_vector[2] = 1
    elif -1 < angle <= 0:
        ds_vector[2] = 2
    elif 0 < angle <= 1:
        ds_vector[2] = 3
    elif 1 < angle <= 6:
        ds_vector[2] = 4
    elif 6 < angle <= 12:
        ds_vector[2] = 5
    # Discretize theta' (angular velocity)
    if abs(cs[3]) <= 50:
        ds_vector[3] = 1
    elif cs[3] < -50:
        ds_vector[3] = 0
    elif cs[3] > 50:
        ds_vector[3] = 2
    ds = int(ds_vector[0]*54+ds_vector[1]*18+ds_vector[2]*3+ds_vector[3])
    return ds

## Return the most optimal action and the corresponding Q value:
def optimal_action(Q, state, eps_rate, env):                              ## (C)
    p = np.random.random()
    # Choose random action if in 'exploration' mode
    if p < eps_rate:
        action = env.action_space.sample()       # choose randomly between 0 and 1
    else:
        # Choose optimal action based on learned weights if in 'exploitation' mode
        action = np.argmax(Q[state,:])
    return action

##  Update Qvalues based on the logic of the Double Q-learning:             ## (D)
def updateQValues(state, action, reward, next_state, alpha, eps_rate, env):
    p = np.random.random()
    if (p < .5):
    # Update Qvalues for Table Q1
        next_action = optimal_action(Q1, next_state, eps_rate, env)
        Q1[state][action] = Q1[state][action] + alpha * \
                (reward + GAMMA * Q2[next_state][next_action] - Q1[state][action])
    else:
    # Update Qvalues for Table Q2
        next_action = optimal_action(Q2, next_state, eps_rate, env)
        Q2[state][action] = Q2[state][action] + alpha * \
                  (reward + GAMMA * Q1[next_state][next_action] - Q2[state][action])
    return next_action

class ReplayMemory:                                                       ## (E)
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
    def push(self, s, a , r, ns):
        self.memory.append([s, a, r, ns])
        if len(self.memory) > self.capacity:
            del self.memory[0]
    def sample(self, MINIBATCH_SIZE):
        return random.sample(self.memory, MINIBATCH_SIZE)
    def __len__(self):
        return len(self.memory)
```

(Continued on the next slide .....)

Python Implementation of Q-Learning (contd.)

(..... continued from the previous slide)

```
if __name__ == "__main__":                                          ## (F)
    ## Learn the weights for Double Q learning:
    max_t = 0
    duration_history = []
    history = []
    epsilon_history = []
    repmemory = ReplayMemory(REP_MEM_SIZE)
    for i_episode in range(MAX_ITER):
        ## state is a 4-tuple: [cart_pos, cart_vel, pole_angle, pole_ang_vel]
        state = env.reset()         ###  state set to uniformly dist random bet -0.05 and 0.05
        done = False
        # Initial state and action selection when environment restarts
        state = map_discrete_state(state)
        action = optimal_action(0.5*(Q1+Q2), state, 0, env)
        t = 0
        # Decay the exploration parameter in epsilon-greedy approach.
        eps_rate *= EXPLORATION_DECAY
        eps_rate = max(EXPLORATION_MIN, eps_rate)
        while True:
            #env.render()
            next_state, reward, done, info = env.step(action)
            if done:
                reward = -10
            next_state = map_discrete_state(next_state)
            repmemory.push(state, action, reward, next_state)
            # Update Q table using Double Q learning and get the next optimal action.
            next_action = updateQValues(state, action, reward, next_state, alpha, eps_rate, env)
            # Update Q values by randomly sampling experiences in Replay Memory
            if len(repmemory)> MINIBATCH_SIZE:
                experiences = repmemory.sample(MINIBATCH_SIZE)
                for experience in experiences:
                    ts, ta, tr, tns = experience
                    updateQValues(ts, ta, tr, tns, alpha, eps_rate, env)
            state = next_state
            action = next_action
            t += 1
            if done:
                break
        history.append(t)
        if i_episode > 50:
            latest_duration = history[-50:]
        else:
            latest_duration = history
        #print(latest_duration)
        duration_run_avg = np.mean(latest_duration)
        #print(duration_run_avg)
        duration_history.append([t, duration_run_avg])
        epsilon_history.append(eps_rate)
        cum_reward += t
        if(t>max_t):
            max_t = t
        train_reward_history.append([cum_reward/(i_episode+1), max_t])
        print("\nEpisode: %d   Episode duration: %d timesteps | epsilon %f" % (i_episode+1, t+1, eps_rate))

    np.save('Q1.npy', Q1)
    np.save('Q2.npy', Q2)
    fig = plt.figure(1)
    fig.canvas.set_window_title("DQL Training Statistics")
    plt.clf()
    plt.subplot(1, 2, 1)
    plt.title("Training History")
    plt.plot(np.asarray(duration_history))
    plt.xlabel('Episodes')
    plt.ylabel('Episode Duration')
```

(Continued on the next slide .....)

Python Implementation of Q-Learning (contd.)

(..... continued from the previous slide)

```
    plt.subplot(1, 2, 2)
    plt.title("Epsilon for Episode")
    plt.plot(np.asarray(epsilon_history))
    plt.xlabel('Episodes')
    plt.ylabel('Epsilon Value')
    plt.savefig('Cartpole_DoubleQ_Learning.png')
    plt.show()                                                    ## (G)

    ##  Finished exploration. Start testing now.
    import pymsgbox
    response = pymsgbox.confirm("Finished learning the weights for the Double-Q Algorithm.  Start testing?")
    if response == "OK":
        pass
    else:
        sys.exit("Exiting")

    print("\n\n\nThe Testing Phase:\n\n")
    ##  Control the cartpole using the learned weights using Double Q learning
    play_reward =0
    for i_episode in range(100):
        observation = env.reset()
        done = False
        # Initial state and action selection when environment restarts
        state = map_discrete_state(observation)
        action = optimal_action(0.5*(Q1+Q2), state, 0, env)
        t = 0
        eps_rate = 0
#        time.sleep(0.25)
        while not done:
            env.render()
            time.sleep(0.1)
            next_state, reward, done, info = env.step(action)
            next_state = map_discrete_state( next_state )
            next_action = optimal_action(Q1, state, 0, env)
            state = next_state
            action = next_action
            t += 1
        play_reward += t
        print("Episode ",i_episode," Episode duration: {} timesteps".format(t+1))
#        print("Episode ",i_episode," : Exploration rate = ", eps_rate, " Cumulative Average Reward = ", play_reward/(i_episode+1))

    env.close()
```

Visualizing the Training Phase for DoubleQ Learning

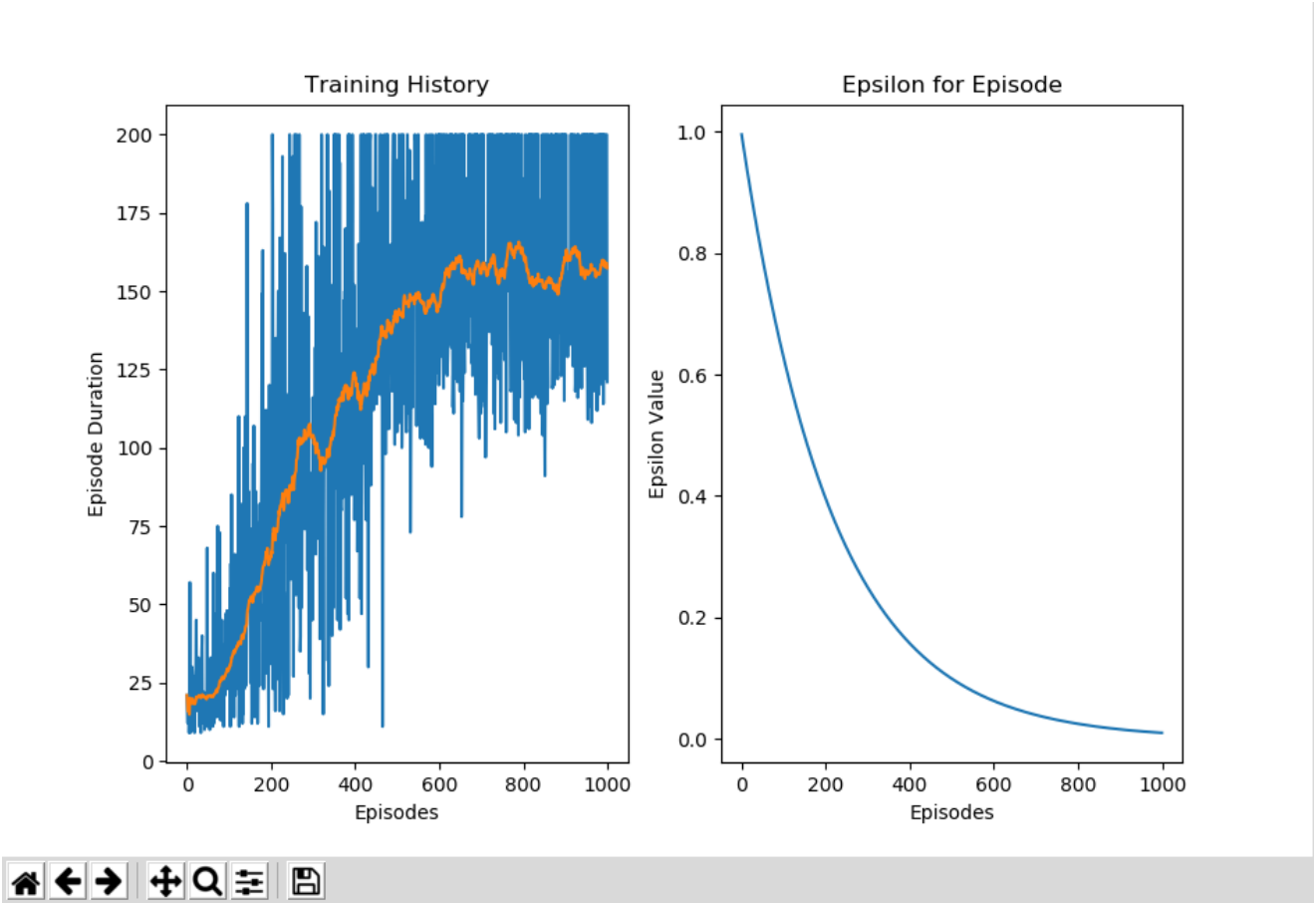

Figure: The orange plot at left shows how the average duration of an episode (in terms of the timesteps until the pole falls) increases as training proceeds. The value of epsilon as shown in the right plot controls the extent to which the agent chooses an action randomly vis-a-vis choosing an action based on Double Q learning. Initially, the value of epsilon is 1, implying that the actions will be chosen completely randomly. In the code, epsilon is represented by the variable `eps_rate`.

Outline

Why a Neural-Network Implementation of Q-Learning

- The Q-Learning based solution shown earlier explicitly constructs a Q table in which each row stands for a state and each column for an action. When in reality the state space is continuous — as it is for the cart-pole problem — the discretization will often introduce an approximation that degrades the quality of learning.
- There is an excellent way to evaluate the quality of learning for the Cart-Pole problem: the duration of each episode, which is the number of time-steps from the start of the episode ($t = 0$) until the time the state `fallen` is reached during each episode.
- If you execute the code shown in the previous section, the average duration you will get with that implementation of Q-Learning is likely to be around 100 time steps.
- To improve the performance beyond this, you'd need to go to a still finer discretization of the state space, or, better yet, to ditch the discretization altogether by using a neural-network based solution.

Q-Learning with a Neural Network

- What is shown next is a neural-network based implementation of Q-Learning that works with a continuous state space. This implementation is also by Rohan Sarkar.
- The neural-network is trained to take the continuous state value s at its input and to output the values of $Q(s, a_i)$ for each action a_i available to the agent in the input state. For its next action, the agent would obviously choose the action that has the largest $Q(s, a_i)$ value.
- Should the action chosen as described above result in getting further away from the desired goal, you would backpropagate a negative reward to reflect the undesirability of what was predicted by the neural network.

A Python Implementation of Q-Learning with a Neural Network

- The code shown next starts by initializing the hyperparameters in the lines that start with the one labeled A on Slide 77.
- By initializing `EPISODES=1000`, we are declaring that we only want to run the code for 1000 episodes. And the setting `EXPLORE_EPI_END = int(0.1*EPISODES)`, we are telling the system to use 100 of the episodes in the exploration phase. What does that mean?
- As it turns out, in its common usage, the phase "Exploration Phase" in neural Q-learning has a different meaning in relation to how I used it in the previous section. The "Exploration Phase" here means for the agent to figure out the next state and the reward just by interacting with the environment.

Q-Learning with a Neural Network (contd.)

- As to what we mean by the agent interacting with the environment, let's examine the following lines of code from the method `run_episode()` of `QNetAgent`. That method is defined at line G on Slide 78.

```
        while True:
            environment.render()
            # Select action based on epsilon-greedy approach
            action = self.select_action(FloatTensor([state]))                    ## (a)
            c_action = action.data.cpu().numpy()[0,0]                            ## (b)
            # Get next state and reward from environment based on current action
            next_state, reward, done, _ = environment.step(c_action)             ## (c)
            # negative reward (punishment) if agent is in a terminal state
            if done:                                                             ## (d)
                reward = -10 # negative reward for failing                       ## (e)
            # push experience into replay memory
            self.memory.push(state,action, reward, next_state)                   ## (f)
            # if initial exploration is finished train the agent
            if EXPLORE_EPI_END <= e <= TEST_EPI_START:                           ## (g)
                self.learn()                                                     ## (h)
            state = next_state                                                   ## (i)
            steps += 1                                                           ## (j)
```

- The call in line (c) above is what's meant by interacting with the environment. The variable `environment` is set in the first line under `__main__` on Slide 79 to `gym.make('CartPole-v0')` where `gym` is the very popular OpenAI platform for research in RL.

Q-Learning with a Neural Network (contd.)

- For the Cart-Pole problem, the `gym` library provides a *very rudimentary* physics-based modeling of the Cart-Pole. The call in line (c) on the previous slide asks this model as to what the next state and the next reward should be if a given action is invoked in the current state. Suppose the currently randomly chosen state of the pole is `soft_lean_right` and the current randomly chosen action is `move left` (which is represented by the integer 1 in the code), the environment is likely to return `hard_lean_right` for the next state and a reward of 0 or -1.

- It is such interactions that are carried out in the exploration phase. In each episode, the agent makes state-to-state transitions with the help of the environment until it reaches the terminal state `fallen`.

- Going back to the beginning of this section, the second statement in the hyperparameter initializations that start in line A on Slide 77 means that we want to use the first 100 episodes for exploration prior to starting neural learning.

Q-Learning with a Neural Network (contd.)

- The main purpose of the "Exploration Phase" described above is to fill the `ReplayMemory`, defined in line labeled D, with the state transitions generated during that phase. What is actually stored in the ReplayMemory are 4-tuples like `[state, action, next_state, reward]` that were generated during the exploration. These are stored in the instance variable `self.memory` defined for the `ReplayMemory` class.

- After the storage in `ReplayMemory` is initialized in the manner indicated above, the agent starts up the `Training Phase` for training the neural network that is defined in the section of the code that begins in the line labeled C.

- The function whose job is to train the neural network is called `learn()` and it is defined starting with line H. That function creates batches of randomly selected training samples from what is stored in `ReplayMemory` and feeds those into the neural network.

Q-Learning with a Neural Network (contd.)

- To continue with the description of the `learn()` function, from each 4-tuple `[state, action, next_state, reward]` chosen from the replay memory, the first two, meaning `state` and `action`, are used as input to the network and the target at the output consists of `next_state` and `reward`.

- The variable `e` in the code stands for the episode number. In the following statements taken from the beginning portion of the `run_episode()` function defined starting at line G, note how the episode number `e` is used to decide what phase the agent is in:

```
def run_episode(self, e, environment):
    state = environment.reset() # reset the environment at the beginning
    done = False
    steps = 0
    # Set the epsilon value for the episode
    if e < EXPLORE_EPI_END:
        self.epsilon = EPS_START
        self.mode = "Exploring"
    elif EXPLORE_EPI_END <= e <= TEST_EPI_START:
        self.epsilon = self.epsilon*EPS_DECAY
        self.mode = "Training"
    elif e > TEST_EPI_START:
        self.epsilon = 0.0
        self.mode = "Testing"
```

Q-Learning with a Neural Network (contd.)

- For the parameter initializations shown in the code, the value of `EXPLORE_EPI_END` in the code fragment shown on the previous slide will be set to 100. Therefore, for episode number less than 100, the "if" block shown above would set `self.epsilon` to `EPS_START`, which is equal to 1. And `self.mode` would be set to "Exploring" at the same time.
- Setting `self.epsilon` to 1 causes the `select_action()` method defined at line F to return a purely randomly chosen action in whatever mode the agent happens to be in at the moment. This is in keeping with the expected behavior of the agent in the Exploration phase according to the epsilon-greedy policy for agents.

Q-Learning with a Neural Network (contd.)

```
##  Solving the Cart-Pole problem with PyTorch
##  Code adapted from https://gist.github.com/Pocuston/13f1a7786648e1e2ff95bfad02a51521
##  Modified by Rohan Sarkar (sarkarr@purdue.edu)

##  QNet_pytorch_v5.py

import gym                                                                    ## (A)
import random
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import matplotlib.pyplot as plt
import numpy as np
##  Set the hyperparameters for training:
EPISODES = 1000           # total number of episodes
EXPLORE_EPI_END = int(0.1*EPISODES)     # initial exploration when agent will explore and no training
TEST_EPI_START = int(0.7*EPISODES )    # agent will be tested from this episode
EPS_START = 1.0           # e-greedy threshold start value
EPS_END = 0.05            # e-greedy threshold end value
EPS_DECAY = 1+np.log(EPS_END)/(0.6*EPISODES)         # e-greedy threshold decay
GAMMA = 0.8               # Q-learning discount factor
LR = 0.001                # NN optimizer learning rate
MINIBATCH_SIZE = 64       # Q-learning batch size
ITERATIONS = 40           # Number of iterations for training
REP_MEM_SIZE = 10000      # Replay Memory size

use_cuda = torch.cuda.is_available()                                          ## (B)
FloatTensor = torch.cuda.FloatTensor if use_cuda else torch.FloatTensor
LongTensor = torch.cuda.LongTensor if use_cuda else torch.LongTensor
ByteTensor = torch.cuda.ByteTensor if use_cuda else torch.ByteTensor
Tensor = FloatTensor

class QNet(nn.Module):                                                        ## (C)
    """
    Input to the network is a 4-dimensional state vector and the output a
    2-dimensional vector of two possible actions: move-left or move-right
    """
    def __init__(self, state_space_dim, action_space_dim):
        nn.Module.__init__(self)
        self.l1 = nn.Linear(state_space_dim, 24)
        self.l2 = nn.Linear(24, 24)
        self.l3 = nn.Linear(24, action_space_dim)
    def forward(self, x):
        x = F.relu(self.l1(x))
        x = F.relu(self.l2(x))
        x = self.l3(x)
        return x

class ReplayMemory:                                                           ## (D)
    """
    This class is used to store a large number, possibly say 10000, of the
    4-tuples '[State, Action, Next State, Reward]' at the output, meaning even
    before the nueral-network based learning kicks in.  Subsequently, batches
    are constructed from this storage for training. The dynamically updated
    as each new 4-tuple becomes available.
    """
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
    def push(self, s, a , r, ns):
        self.memory.append((FloatTensor([s]),
                    a,  # action is already a tensor
                    FloatTensor([ns]),
                    FloatTensor([r])))
        if len(self.memory) > self.capacity:
            del self.memory[0]
    def sample(self, MINIBATCH_SIZE):
        return random.sample(self.memory, MINIBATCH_SIZE)
    def __len__(self):
        return len(self.memory)
```

(Continued on the next slide .....)

Q-Learning with a Neural Network (contd.)

(..... continued from the previous slide)

```
class QNetAgent:                                                                    ## (E)
    def __init__(self, stateDim, actionDim):
        self.sDim = stateDim
        self.aDim = actionDim
        self.model = QNet(self.sDim, self.aDim)    # Instantiate the NN model, loss and optimizer for training the agent
        if use_cuda:
            self.model.cuda()
        self.optimizer = optim.Adam(self.model.parameters(), LR)
        self.lossCriterion = torch.nn.MSELoss()
        self.memory = ReplayMemory(REP_MEM_SIZE)  # Instantiate the Replay Memory for storing agent's experiences
        # Initialize internal variables
        self.steps_done = 0
        self.episode_durations = []
        self.avg_episode_duration = []
        self.epsilon = EPS_START
        self.epsilon_history = []
        self.mode = ""

    def select_action(self, state):                                                 ## (F)
        """ Select action based on epsilon-greedy approach """
        p = random.random() # generate a random number between 0 and 1
        self.steps_done += 1
        if p > self.epsilon:
            # if the agent is in 'exploitation mode' select optimal action
            # based on the highest Q value returned by the trained NN
            with torch.no_grad():
                return self.model(FloatTensor(state)).data.max(1)[1].view(1, 1)
        else:
            # if the agent is in the 'exploration mode' select a random action
            return LongTensor([[random.randrange(2)]])

    def run_episode(self, e, environment):                                          ## (G)
        state = environment.reset() # reset the environment at the beginning
        done = False
        steps = 0
        # Set the epsilon value for the episode
        if e < EXPLORE_EPI_END:
            self.epsilon = EPS_START
            self.mode = "Exploring"
        elif EXPLORE_EPI_END <= e <= TEST_EPI_START:
            self.epsilon = self.epsilon*EPS_DECAY
            self.mode = "Training"
        elif e > TEST_EPI_START:
            self.epsilon = 0.0
            self.mode = "Testing"
        self.epsilon_history.append(self.epsilon)
        while True:                                 # Iterate until episode ends (i.e. a terminal state is reached)
            environment.render()
            action = self.select_action(FloatTensor([state]))    # Select action based on epsilon-greedy approach
            c_action = action.data.cpu().numpy()[0,0]
            # Get next state and reward from environment based on current action
            next_state, reward, done, _ = environment.step(c_action)
            if done:                                # negative reward (punishment) if agent is in a terminal state
                reward = -10 # negative reward for failing
            # push experience into replay memory
            self.memory.push(state,action, reward, next_state)
            # if initial exploration is finished train the agent
            if EXPLORE_EPI_END <= e <= TEST_EPI_START:
                self.learn()
            state = next_state
            steps += 1
            if done:                                        # Print information after every episode
                print("{2} Mode: {4} | Episode {0} Duration {1} steps | epsilon {3}"
                      .format(e, steps, '\033[92m' if steps >= 195 else '\033[99m', self.epsilon, self.mode))
                self.episode_durations.append(steps)
                self. plot_durations(e)
                break
```


(Continued on the next slide .....)

Q-Learning with a Neural Network (contd.)

(..... continued from the previous slide)

```
    def learn(self):                                                                    ## (H)
        """
        Train the neural newtwork using the randomly selected 4-tuples
        '[State, Action, Next State, Reward]' from the ReplayStore storage.
        """
        if len(self.memory) < MINIBATCH_SIZE:
            return
        for i in range(ITERATIONS):
            # minibatch is generated by random sampling from experience replay memory
            experiences = self.memory.sample(MINIBATCH_SIZE)
            batch_state, batch_action, batch_next_state, batch_reward = zip(*experiences)
            # extract experience information for the entire minibatch
            batch_state = torch.cat(batch_state)
            batch_action = torch.cat(batch_action)
            batch_reward = torch.cat(batch_reward)
            batch_next_state = torch.cat(batch_next_state)
            # current Q values are estimated by NN for all actions
            current_q_values = self.model(batch_state).gather(1, batch_action)
            # expected Q values are estimated from actions which gives maximum Q value
            max_next_q_values = self.model(batch_next_state).detach().max(1)[0]
            expected_q_values = batch_reward + (GAMMA * max_next_q_values)
            # loss is measured from error between current and newly expected Q values
            loss = self.lossCriterion(current_q_values, expected_q_values.unsqueeze(1))
            # backpropagation of loss for NN training
            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()

    def plot_durations(self,epi):        #  Update the plot at the end of each episode   ## (I)
        fig = plt.figure(1)
        fig.canvas.set_window_title("DQN Training Statistics")
        plt.clf()
        durations_t = torch.FloatTensor(self.episode_durations)
        plt.subplot(1,2,1)
        if epi<EXPLORE_EPI_END:
            plt.title('Agent Exploring Environment')
        elif EXPLORE_EPI_END <= e <= TEST_EPI_START:
            plt.title('Training Agent')
        else:
            plt.title('Testing Agent')
        plt.xlabel('Episode')
        plt.ylabel('Duration')
        plt.plot(self.episode_durations)
        # Plot cumulative mean
        if len(durations_t) >= EXPLORE_EPI_END:
            means = durations_t.unfold(0, EXPLORE_EPI_END, 1).mean(1).view(-1)
            means = torch.cat((torch.zeros(EXPLORE_EPI_END-1), means))
            plt.plot(means.numpy())
        plt.subplot(1,2,2)
        plt.title("Epsilon per Episode")
        plt.xlabel('Episode')
        plt.ylabel('Epsilon')
        plt.plot(self.epsilon_history)
        plt.show(block=False)
        plt.draw()
        plt.pause(0.0001)

if __name__ == "__main__":                                                              ## (J)
    environment = gym.make('CartPole-v0')               # creating the OpenAI Gym Cartpole environment
    state_size = environment.observation_space.shape[0]
    action_size = environment.action_space.n
    # Instantiate the RL Agent
    agent = QNetAgent(state_size, action_size)
    for e in range(EPISODES):                           # Train the agent
        agent.run_episode(e, environment)
    print('Complete')
    test_epi_duration = agent.episode_durations[TEST_EPI_START-EPISODES+1:]
    print("Average Test Episode Duration",np.mean(test_epi_duration))
    environment.close()
    plt.ioff()
    plt.show()
```

Visualizing the Training Phase

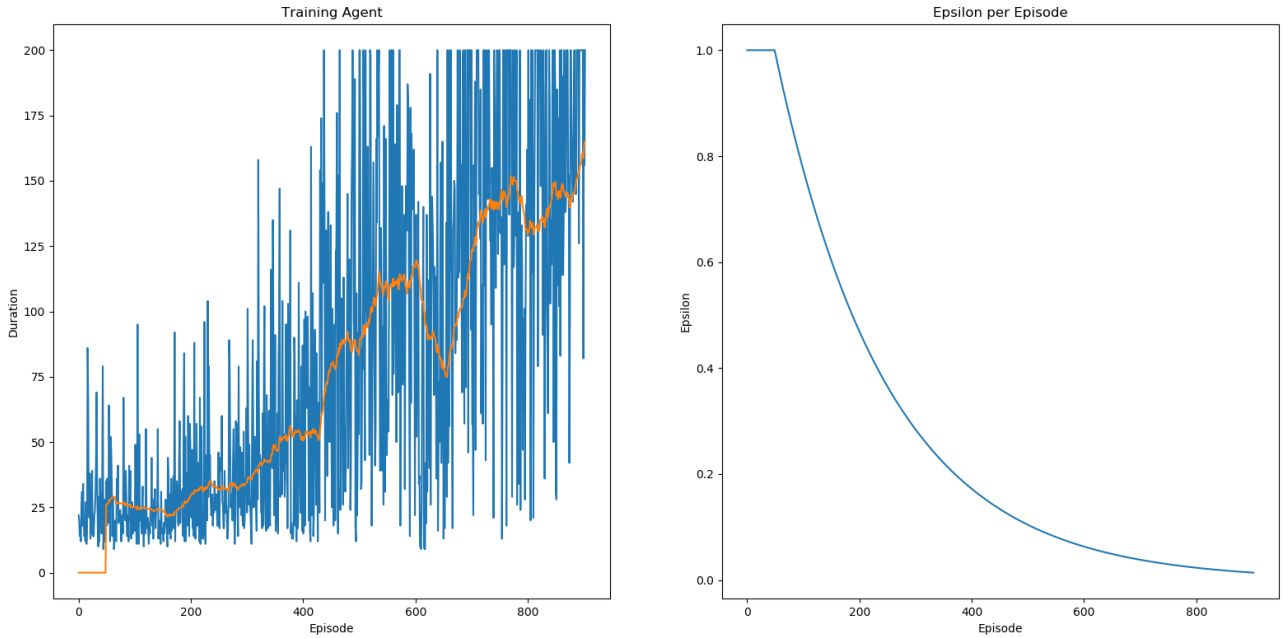

Figure: See how the average duration of an episode, shown in orange in the plot at left, increases as training proceeds. The value of epsilon as shown in the right plot controls the extent to which the agent chooses an action randomly vis-a-vis choosing an action based on neural learning. Initially, the value of epsilon is 1, implying that the actions will be chosen completely randomly. In the code, epsilon is represented by the variable 'self.epsilon'. The small flat part of the curve at the beginning of the plot at right is for the episodes when the replay memory is initialized by interacting with the environment prior to the start of neural learning.

Outline

RL with Policy Based Methods

- So far I have talked about the Q-Learning based approaches to RL. A most important characteristic of Q-Learning was that it estimated explicitly the expected future reward for each possible action in every state of the environment.
- The Q-Learning based methods for RL are also called value-function based methods, the term value-function referring to the function that returns an approximation to the future expected rewards for any given action that the agent may consider applying to the environment in any given state that the environment finds itself in.
- Modern alternatives to Q-Learning methods for RL consist of **policy** based methods **that bypass the need for an explicit estimation of the future expected rewards.**
- Policy based methods *directly* estimate a probability distribution over all available actions in any given state of the environment to help the Agent choose the action in that state.

Policy Based Methods

- Going back to the Pole-Cart problem in which the agent has a choice of exactly two possible actions to choose from — move left or move right — a policy based method would directly estimate a probability distribution over these two actions for the agent to invoke in every state of the pole-cart.
- Let θ represent the parameters in the RL framework for learning to balance the pole-cart. Since *policy* means what action the framework chooses when the pole-cart is in a given state, we can say θ itself represents the policy at the current moment — if the goal of the framework is to directly predict the best choice for the next action.
- That raises the question of how to train such a framework so that it continually improves its policy. [**Remember, the goal of Policy Based methods is the same as it is for the value-function based methods: Maximization of the future expected rewards.**]
- RL frameworks that implement policy-based methods are trained with what's known as the hill-climbing approach explained next.

Hill Climbing for Policy Based Methods

- In the hill climbing approach, the learning framework starts with a random choice for the individual parameters in θ and executes an episode while collecting the rewards with the execution of the recommended actions in each state of the environment until the end of the episode.
- For a simple explanation of hill climbing, let's say that the values of the parameters in θ are perturbed randomly and another episode attempted in the same manner as before. We again collect the rewards for the episode. If the rewards this time exceed the rewards in the previous attempt, we retain the new values for θ as the new best policy. Should the opposite be true, we go back to the previous value of θ as the current best policy.
- In this manner, with each episode, we seek to improve the policy as represented by the parameter vector θ. We are engaged in hill-climbing because at the core of the algorithm we constantly seek to maximize the rewards reaped in each episode.

Outline

RL with Policy Gradients

- The most successful of the current applications of RL employ policy-based methods through **Policy Gradients**.
- At the heart of a policy-gradient based approach to RL is a formula that *directly* expresses the future expected rewards as a function of the parameters θ. In such formulas, the future expected reward is **typically represented by the notation** $J(\theta)$. Since the goal of all RL frameworks is to maximize the future expected rewards when deciding which action to invoke in the current state, as you can imagine, the gradient $\nabla_\theta J(\theta)$ would directly tell us how to update the parameter vector θ with the choice made for the action in the current state.
- For a deeper explanation of the policy-gradient approach, I'll consider the application of RL as reported in "*Reinforcement Learning to Rank in E-Commerce Search Engine: Formalization, Analysis, and Application*", by Hu et al. at

https://arxiv.org/pdf/1803.00710.pdf

A Case Study in Policy Gradients for RL

- The publication mentioned on the previous slide presents an e-commerce search engine that is trained to rank the items returned for a query by keeping track of the interaction between the consumer and the search engine and using that interaction in an RL algorithm to improve the search engine.
- As you can imagine, the reward function in such a search engine is as simple as it can be: The reward is 1 if the consumer ends up ordering one of the items shown by the search engine, and 0 if the consumer leaves the search engine.
- Let θ represent the parameters of a learning framework that maps the current **state** of the environment to a probability distribution over the actions. We use the symbol π to denote the policy that maps the states to actions. That is, we have $\pi : \mathcal{S} \rightarrow \mathcal{A}$, where $\mathcal{S}$ is the set of all states and $\mathcal{A}$ the set of all actions. Since the policy at any given moment will depend on θ, we can also use the notation π_θ to denote the current policy.

A Case Study in Policy Gradients for RL (contd.)

- For a more general understanding of what I mentioned in the last bullet of the previous slide, consider RL in a stochastic context. For stochastic RL, we can define a policy by $\pi_\theta(s,a) = Pr(a_t = a|s_t = s,\theta)$ for $\forall s \in \mathcal{S}$ and $\forall a \in \mathcal{A}$ where $\mathcal{S}$ is the set of all states and $\mathcal{A}$ the set of all actions. For a specific current state s, the expression $Pr(a_t = a|s_t = s,\theta)$ returns a probability distribution over **all** the actions for the agent. As to which specific action the agent chooses to execute while the environment is in state s is a different issue.
- On the other hand, in a deterministic context, it is simpler to express the policy by $\pi_\theta(s) = a$ for $s \in \mathcal{S}$.

  [**To remind again, we refer to π as policy because it's about a deliberate decision to invoke a specific action a when the environment is in a given state s. Initially, the choice of the action will be random. The goal of RL would be to learn θ so that "smart" choices for the actions will help achieve the overall goal to which RL is being applied.**]
- Our goal is to learn the optimum policy, that is, the policy that maximizes the future expected rewards.

RL with Policy Gradients (contd.)

- We consider a T-step interaction between the consumer and the search engine for driving the RL algorithm. The values of the parameters at iteration t in this T-step interaction will be denoted θ_t.
- When we go from iteration t to the next iteration $t+1$, we want the parameters θ_t to change to θ_{t+1} in such a way that we maximize the total expected rewards at the end of each T-step interaction — which would be the goal of any optimum policy.
- We now define $J(\theta)$ as the expectation of the T-step reward over all possible trajectories $\tau : s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_{T-1}, a_{T-1}, r_{T-1}, s_T$:

$$J(\theta) \quad = \quad E_{\tau \sim \rho_\theta}\{R(\tau)\} \quad = \quad E_{\tau \sim \rho_\theta}\Big\{ \sum_{t=0}^{T-1} r_t \Big\}$$

- In the above equation, $R(\tau)$ is the expected reward on the trajectory τ and the expectation is over all possible trajectories.

RL with Policy Gradients (contd.)

- In the expression for $J(\theta)$ on the previous slide, if the terminal state of a trajectory is reached in under T decision steps, the summation over t is truncated to that state. Also note that ρ_θ is the distribution of trajectories like τ under the policy parameters θ.
- The gradient of $J(\theta)$ with respect to the policy parameters θ is given by:

$$\nabla_\theta J(\theta) \quad = \quad E_{\tau \sim \rho_\theta} \Big\{ \sum_{t=0}^{T-1} \nabla_\theta \pi_\theta(s_t) \cdot R_t^T(\tau) \Big\}$$

  where $R_t^T = \sum_{t'=t}^{T-1} r_{t'}$ involves the future expected rewards only from the step t until the end of the T-step decision process.
- Assuming that the partial $J(\theta)$ exists, when we go from iteration index t to the iteration index $t+1$, we can update the parameter vector θ by

$$\theta_{t+1} \quad = \quad \theta_t \; + \; \alpha \cdot \nabla J(\theta)$$

  where α is a non-negative number that controls the step size for updating the parameters.

RL with Policy Gradients (contd.)

- In the parameter update equation on the previous slide, note that, unlike how gradients of loss are used for training a typical deep learning neural network, the step in the parameter hyperplane would be in the direction of maximizing the future expected reward. In other words, our meandering in the parameter hyperplane could be construed as hill climbing.
- The expression shown for $\nabla J(\theta)$ on the previous slide involves a summation at all the time steps t of the gradient of the policy π_θ with respect to the parameters θ — recall that the policy π_θ is a mapping from the states to the actions. This is multiplied by the expected rewards from that time step until the end of the T time steps.
- The e-commerce paper cited at the beginning of this section only considered deterministic policies in their application of the policy gradient method for RL. This leads to the simplification shown on the next slide.

RL with Policy Gradients (contd.)

- For the case of deterministic policies π_θ (meaning that the policy returns a specific action for each state) and when the Q values can be estimated for the the (s, a) pairs, the above formula for the gradient can be expressed as:

$$\nabla_\theta J(\theta) \quad = \quad E_{\tau \sim \rho_\theta} \Big\{ \sum_{t=0}^{T-1} \nabla_\theta \pi_\theta(s_t) \cdot \nabla_a Q^{\pi_\theta}(s_t, a) \Big|_{a=\pi_\theta(s_t)} \Big\}$$

- In general, though, we are more interested in stochastic policies in which π_θ returns a probability distribution over the action space.

- Nonetheless, it is good to know that a deterministic policy on which the above formula is based is a limiting case of a stochastic policy as the variance of the latter goes to zero. Recall the definition given earlier for the policy $\pi_\theta(s, a)$ for the stochastic case.

RL with Policy Gradients (contd.)

- The challenge in the deterministic implementation based on the gradient formula shown on the previous slide is in estimating the value function $Q^{\pi_\theta}(s_t, a)$, the difficulties being caused by (according to the authors) high variance and unbalanced distribution of the immediate rewards in each state.
- The authors say that above mentioned high variance is a result of the fact that for most action pairs (s, a) the rewards tend to be zero since a non-zero reward, when it is given, occurs only at the last step in the T-step interaction between the consumer and the search engine.
- See the paper at the link provided at the beginning of this section for further information.