Lesson Notes (02/05 - 16/05)

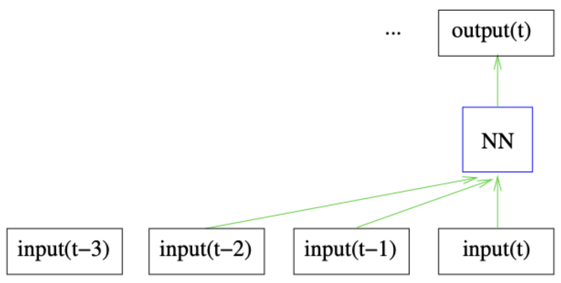
Lesson 17: Recurrent Neural Networks

@May 2, 2023

Typically associated with the task of processing sequences of data, especially temporal sequences. Some typical tasks could be:

- turn an input sequence into an output sequence in the same or different domains (natural language processing tasks like translation between languages, speech/sound recognition…)
- predict an item in the sequence (usually the next term in a sequence or a missing item) - Blurs the distinction between supervised and unsupervised learning
- predict a result from a temporal sequence of states (Reinforcement Learning and robotics)

Memory-less approach: typically you try to compute the output as a result of a fixed number of elements in the sequence → the output we want to predict depends on a small window of items in the input sequence. Limitation is that maybe it could be not enough to model the problem. Approach used (e.g.) in Q-Learning. NB you can't fix a priori the number of elements in the input sequence.



Think in terms of the temporal unfolding of the network. The temporal unfolding of a RNN is just a feed-forward neural network, you don't have cycles anymore as they've been solved to obtain one-directional flow again, which we know how to solve through back-prop. The only trick is that you need to remember the complete unfolding as one network, so weights must be shared. In Convolution you practice spatial averaging, here is a Temporal Averaging.

It is easy to modify the backpropagation algorithm to incorporate equality constraints between weights. We compute the gradients as usual, then we average them so that they induce a same update. If the initial starting weights satisfied the constraint, they will continue to do.

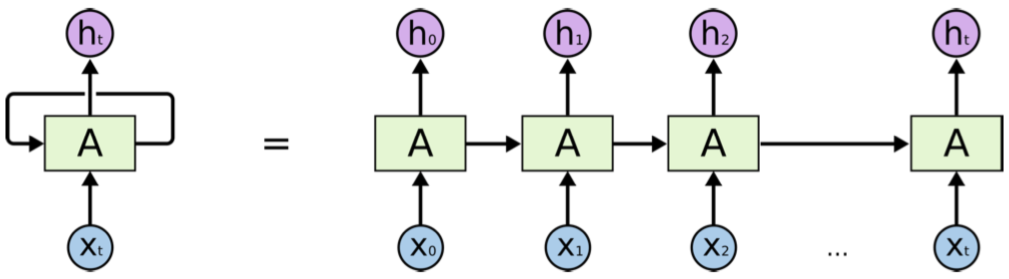
As we already said, we can think of the Recurrent Networks as some layered, feed-forward networks with shared weights and train this feed-forward networks with weight constraints. As we reason in the time domain:

- the forward pass builds up a stack of the activities of all the units at each time step
- the backward pass peels activities off the stack to compute the error derivatives at each time step
- we add together the derivatives at all the different times for each weight.

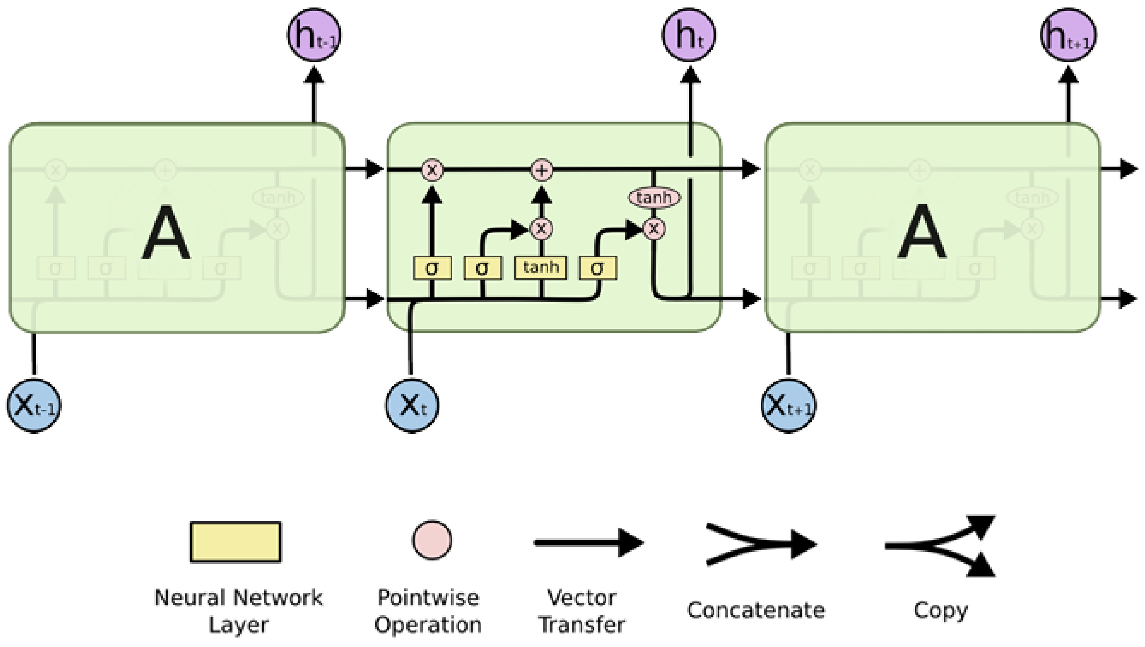
We need an initial state (both weights and values) for the hidden states → two possibilities: random initialization or learn during training an optimal initialization

Long-Short Term Memory (LSTM)

Any time we have a Recurrent Neural Network, we have a vector of inputs, a vector of outputs and a network inside; some links are direct, some are cycles. Think of the network as temporal unfolding (unrolled form = we see the network as connections into the future). A forward link, therefore, should be understood as a looping connection in the original network.



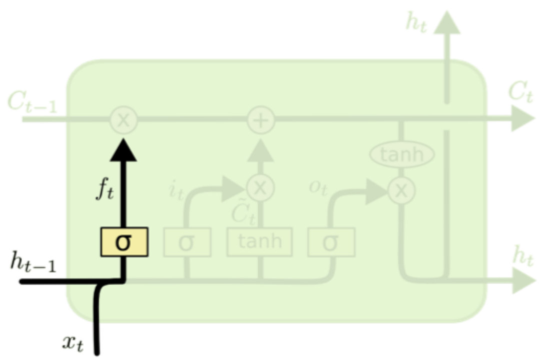
In a traditional situation, the internal value of a cell is the memory of the network C_t, which is then passed on. At time step t you receive a new input x_t. You process it through the network (you can have a smashing activation to keep the content of the memory controlled (e.g. $\tanh$) to avoid situations of exponential explosions). The produced content can be passed both as output and to an instance in the future. This is problematic, as as you try to process the information you risk losing the previous values (destroy the memory). We want to preserve the memory as much as possible.



Overall Structure of a LSTM. The LSTM has the ability to remove or add information to the cell state, in a way regulated by suitable gates. Gates are a way to optionally let information through: the product with a sigmoid neural net layer simulates a boolean mask (Introduction of the concept of Attention)

We have a straight line (C-line) from the previous value to the future. It is a sort of bus that allows in principle the memory to pass through with minor modifications (product and sum gates), which are controlled by some mechanisms called gates (ways to slightly modifying the content of the cells through controlled modifications). In this case these are 1-layer networks. In this case we have a distinction between the content of the cell and the output produced, as the output is obtained as a sort of post-processing from the memory.

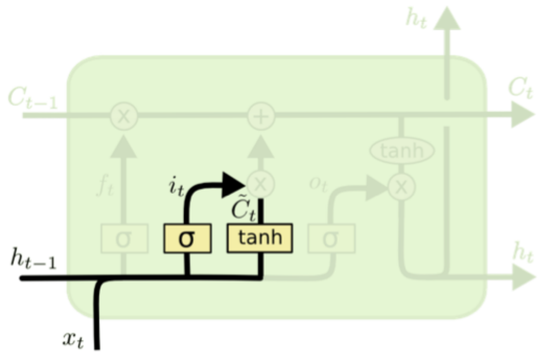
First thing we want to decide is if the memory I have is still interesting or I want to forget part of it. So, I decide what is still relevant by computing a boolean mask that marks the importance of the particular byte (probability that the byte should be preserved as it is important). It is computed according to the current input and the previous output of the network, you concatenate them, process them through a dense network to obtain the "forget map" that will be then multiplied to the content of the memory to obtain a result: form of Attention → you're focusing on some part



Forget Gate

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

of the memory according to the input and previous output).

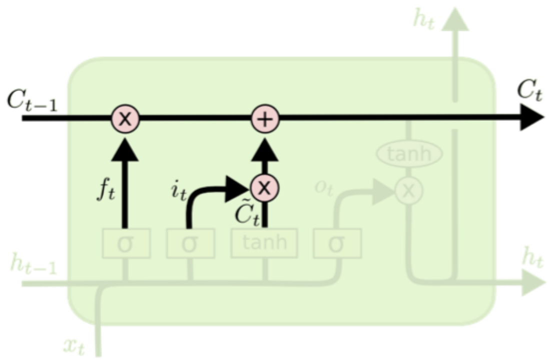


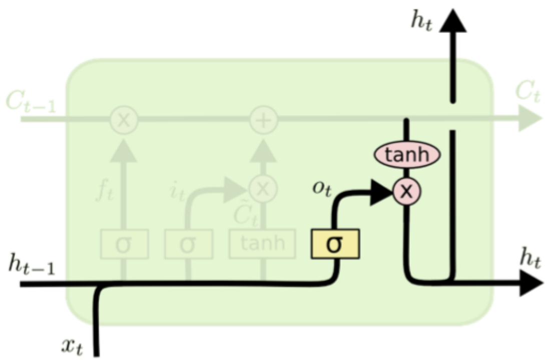
Then we need to know how to modify the memory. Two steps: synthesize the input and then process it using the tanh as an activation for this layer. We use an update gate to focus the attention on only a section of the input we want to really use. Focus on the expressiveness of the network. So we produce a potential update and we mask it.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Update the cell: we sum the memory obtained by the forget gate to the masked update to obtain the final results.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$





Then we need to compute the new output. We synthesize it, apply the tanh first of all to keep it in range [-1,1] (normalize) and then we still use attention mechanisms (synthesize a boolean map starting from the input and the previous outputs, multiplied to a sigmoid) to focus on only the relevant parts.

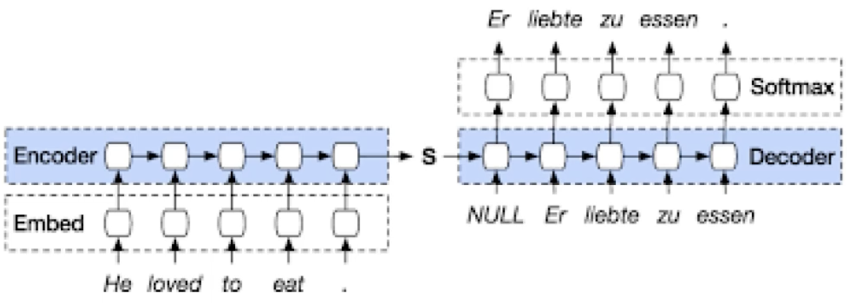
This is actually an old notion (end of last century), which has been used in NLP until 2018 (Transformers introduction → now state-of-the-art)

$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

In Keras, the LSTM layer is similar to a traditional layer and you can easily use it as a black box, after specifying the hyperparameters (`units` → dimension of your memory). The input is an array of dimension `[batch, timesteps, features]` and the output (unless you ask to return sequences) is `[batch, units]`.

A simple application is this: Translation between two languages (short sentences). You need to input sequences and process them by means of an encoder → obtain an interpretation of the content. What is the meaning that is expressed in the input sequence and summarize this meaning into some information, which can also be a sequence but also a different representation. Translation implies two problems:

- you have to generate a sentence that is correct (grammar of the target language). You could use (e.g.) a pre-trained model
- you are interested to translate the sentence → you need to stick to the meaning of the input sentence. Use conditional generation, conditioned by the content extracted from the input sequence.



You can use as S the internal state of the memory cell after processing the full input sequence, or another possibility could be to use S as the initial memory of my output.

Lesson 18: Transformers and Attention

@May 3, 2023

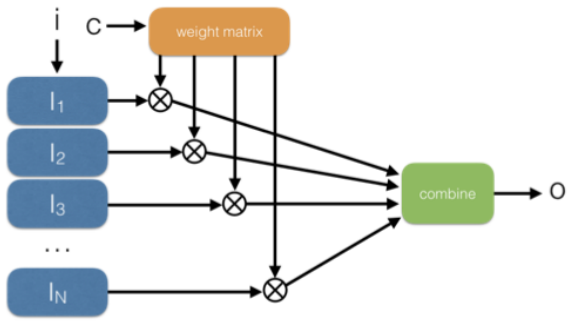
Attention

Attention (this is the general concept, Transformers are a derivation on this topic) is the ability to focus on different parts of the input according to the requirements of the problem being solved (lots of different kinds of attention you can consider). Important for any intelligent behavior, with potential application to a wide range of domains. You can't think of an intelligent behavior without a focusing mechanism. Conceptually, we have two different possibilities:

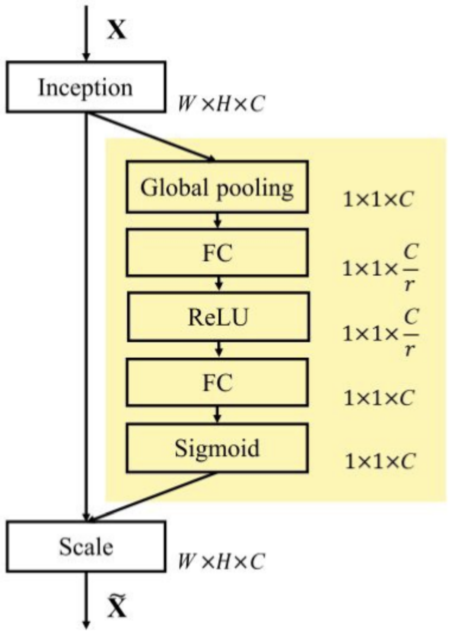
- We can try to “crop” the part that we want to focus on and process just it → Object Detection’s Region Proposal Methods is a form of attention (though it is not the canonical definition)
- The canonical definition of Attention in the field of Deep Learning is masking everything that is not relevant, you don’t modify the dimensions of the data/images. Mask = Zero-Out the sections you don’t want. If you do this operation, you train only on the interesting part of the image, removing what could be a nuisance.

From the point of view of NN, we expect the mechanism to be differentiable, so that we can use Standard Backpropagation techniques.

General idea of implementation are gating maps: I have an input and generate a weighting matrix that establishes the importance of the input, make the product and obtain the focus/attention output. Gating Maps are easy and allow to focus attention in a dynamical way



We’ve seen an example of this method in LSTMs.



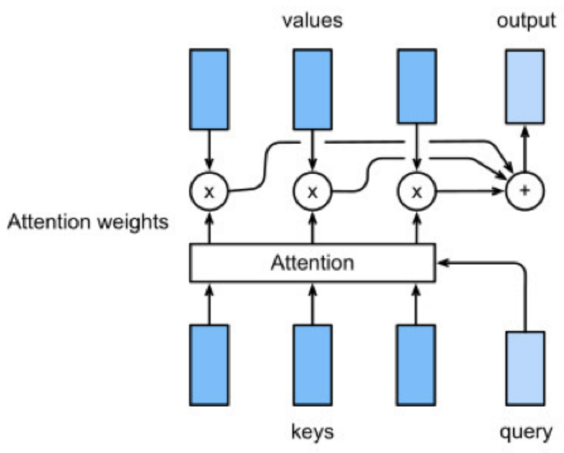
SE-Inception Module

Squeeze and Excitation Layer → new, modular layer. I have my input (image with three dimensions - width, height and channel dimension). The pipeline to extract the map is composed of a first processing through Global Pooling (spatial dimension modification), Fully Connected Layer to reduce the number of channels, then a ReLU, then another Fully Connected Layer to return to the initial dimension and a Sigmoid, from which you finally obtain the map of 0-1s (it’s not a probability map so NO Softmax). This makes the network dynamically understand which channels are relevant and which are not. Then I multiply the input with the map and get the output. Form of Self-Attention → the map itself is generated starting from the input and recalibrates the input itself.

Most famous attention layer is the Key-

Value attention approach, which you can find already implemented in most frameworks (e.g. Keras). It is based on associative memory: you have a dictionary with sets of key:value pairs → I use the keys to access the interesting values. We have an associative array we access by means of the keys, a query is the particular key you're interested in. You compare the query with each key to find the one you're interested and from that you retrieve the value associated to that particular key.

Operations we need: Comparison between vectors → a typical way to compare vectors is cosine similarity, or correlation: product between each key and the query, with the closer ones giving the maximum scores. So I need to compare the scores to obtain the measure of similarity. The result is a vector of scores, we transform into probabilities using a softmax function. Retrieval: an argmax strategy is not feasible because it is not differentiable. We multiply each one of the values by the corresponding probability and we sum all together, obtaining an output whose elements are weighted by the probability that it was the value we're interested in. Suppose we're in the ideal case: the softmax used would give 1 for the interesting case and 0 for the rest.

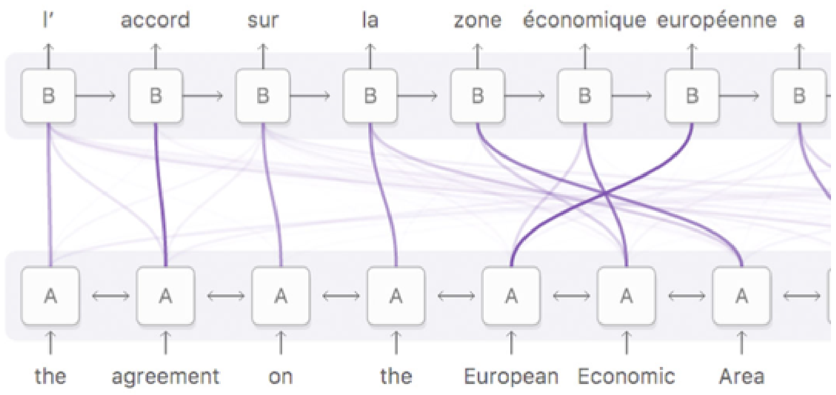


Scores:

- Dot product → more frequent, natural, clear meaning and interpretation. The query and the key must have the same dimension (the result of the dot product is scaled)
- MLP → score is computed by a single-layer neural network (I "learn" it)

Softmax is actually a flat function, so in most of the cases the output will be a noised version.

First applications of Attention Layers (in conjunction with LSTMs) that were published. Translation = passing from a language to another trying to preserve the meaning. Alignment = identify which part of the input are relevant to each word in the output. Alignment in the case of translation is the point of correctly understand which is the word of the input that correspond to a particular word on the output



E.g. in English and French the Adjectives come in different places with respect to the Nouns

You can have alignment also inside the same sentences, or between sentences. This is a form of attention, implemented:

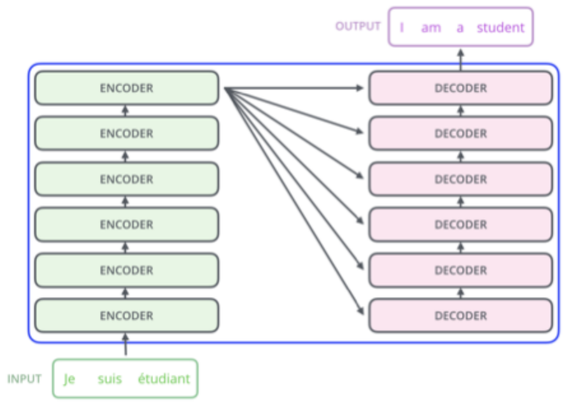
- I have a model who's processing the input sequence (e.g. a LSTM model). The attention is in parallel.
- I have an attending LSTM who's waiting the output. The attending network generates a query describing what it wants to focus on.
- I compare the query with all the values of the input sentence, generating the score. The score is passed to a softmax layer.
- We multiply each value of the input by the respective probability and the output is then obtained as this weighted sum that reflects the focus on the single parts.

The following step is to remove the processing of the input sequence by means of an LSTM model, sequentially. There's no evidence for the effectiveness of the method. This is why transformers are introduced.

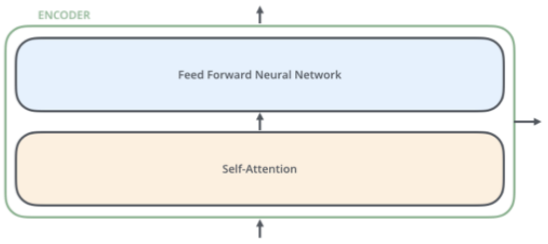
Transformers

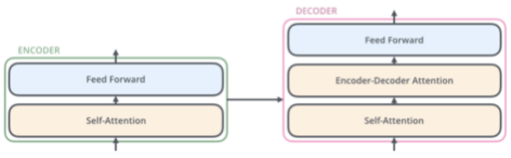
Introduced at the end of 2017-2018 (Attention is All You Need, BERT, GPT) and soon became the model of choice for all Natural Language Processing. Complex network but not so complex network at the same time.

Encoder-Decoder Structure - First stack works on creating an encoding of the input. Then you try to condition the generation according to the information extracted from the encoder (deduce semantics). The decoder stack is trained with complete partial input strings/sentences (it's given the first part of a sentence and has to try and guess which word comes after). Instead of LSTMs, you use two feed-forward networks with a heavy use of multi-head attention.



You usually condition the decoder using only information obtained from the last step of the encoder.





ENCODER: self-attention + feed-forward neural network → each one of the output is obtained as a weighted combination of the input, the output is passed again through a feed-forward Neural Network for processing. In case of the DECODER → In addition to the self-attention layer (query, key and value are all obtained by the same input), we have also an Encoder-Decoder Attention → now the encoder produces the value and we ensure that the encoder is focusing its attention selectively according to the key produced by the decoder. Then we process it through a FFNN.

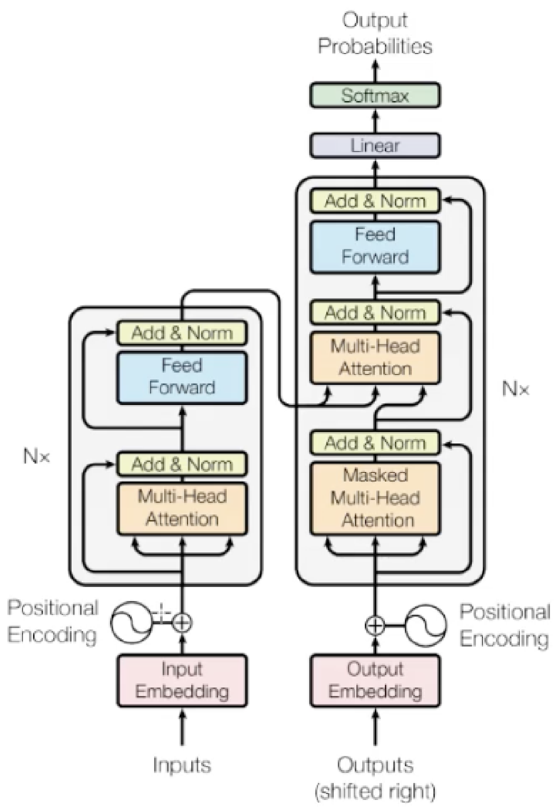
Additional complications: we do not have a single attention mechanism, but we have a multi-head attention mechanism → instead of having a single attention layer we have a stack we apply in parallel from which we obtain information and then concatenate.

When we deal with Transformers, we still use masking → interesting for the decoder part, to ensure that parts of the input which are not interesting for the output generation are ignored.

Each sub-layer in each encoder has a residual connection around it, and it is followed by a layer-normalization step.

Positional encoding is added to word embeddings to give the model some information about the relative position of the words in the sentence. The positional information is a vector of the same dimension d_{model} of the word embedding. The authors use $\sin$ and $\cos$ functions of different frequencies.

Try to let the network to easily focus its attention on parts of the sentence that are at a known relative distance from one another (we don't know how the sentence will be tokenized). Positional encoding is meant to give your position inside the sentence, but allow also a simple computation of relative distances. The position is encoded as a frequency. It is a transformation that can be easily learned as a linear function.



Lesson 19: Basic Reinforcement Learning

@May 9, 2023

Reinforcement Learning → we're interested in learning behaviors. The idea is that we have an agent interacting in an environment, which leads to state changes which drive further actions. The behavior is associated to choosing an action given the current state. We model the agent, the environment is a black box, described by an observation of a state.

The agent, which is in a state s_t, selects a new action a_t, the environment changes according to the action ($s_{t+1}$and gives a “so-called” local reward r_t in response to the action. At each time step t, the agent is in some state s_t. The agent chooses an action a_t; the action can be discrete or continuous (e.g. you decide speed, direction). You choose the action according to some policy → $\pi(a_t|s_t)$: the probability to choose action a_t while in the state s_t. The environment then answers to you with a local reward r_t and enters state s_{t+1}.

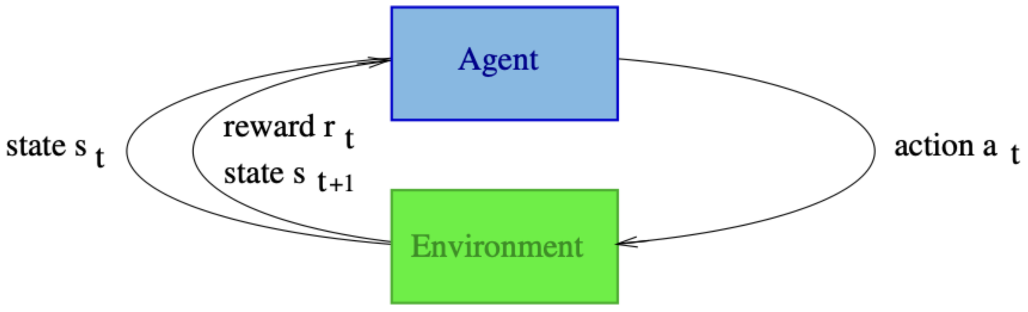
We want to learn the probability distribution of the actions given the state, aka the policy. We want to learn the best policy (according to a metric). The objective is to maximize the Future Cumulative Reward. Maximizing only local reward is a greedy method, giving an advantage only in the short run. Future Cumulative Reward is the sum of all the local rewards you will get in the future if you keep the same policy. You usually have a real goal, which is associated with a big reward, but you can have also smaller local rewards (e.g. in Games)

$$R = \sum_{1 \leq i} r_i$$

We can consider a slightly different quantity: Future Discounted Cumulative Reward. While you act according to a policy, you can be fairly sure of immediate rewards, but not the same for the faraway future. So you weight the reward according to their distance in the future with a discount rate $0 < \gamma \leq 1$, which exponentially decreases with time.

$$R = \sum_{1 \leq i} \gamma^i r_i$$

You're limiting the future you're able to see.



Markov Property → current state completely characterizes the state of the world: future actions only depend on the current state.

History might be important in some situations, so you could be forced to remember → in this case, the problem is not a Markov Decision Process. A Markov Decision Problem depends only on the current state of the world. Normally defined by a tuple of values:

- $\mathcal{S}$ → set of possible states
- $\mathcal{A}$ → set of possible actions
- $\mathcal{R}$ → reward probability given a (state, action) pair → conditional probabilities are generalization of a function
- $\mathcal{P}$ → transition probability to the next state given a (state,action) pair
- γ → discount factor

Now we can define the optimal policy. At time step $t = 0$, we start from the initial state s_0. From $t = 0$ until done, we choose an action a_t according to some policy $\pi(a_t|s_t)$. We sample an action, we communicate it to the environment, which samples the reward $r_t \sim R(r_t|s_t, a_t)$ and the new state $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$.

The policy defines so-called trajectories (or paths) → sequence of the states you traverse, plus the actions and the rewards along the way.

We want to find the optimal policy, that is

$$\pi^* = \arg\max_{\pi} \mathbb{E} \sum_{t \geq 0} \gamma^t r_t$$

The average is over all possible trajectories. We have an expected future cumulative reward and then you compare these policies and take the one that maximizes this quantity.

Model-Free and Model-Based Approach

The transition $s_t \to s_{t+1}$ is not always deterministic, but governed by some probability. In general, it is not known to the agent, it is something relative to the environment and maybe the environment is unknown. If, in order to do learning, we need to learn this probability (in which state we enter if we do an action), we call this Model-Based, as we are trying to model an environment to simulate the techniques. In Model-Free approaches, we don't know and we don't try to learn → we deduce what works from past experiences, as learning would be too complex.

Often, techniques are Model-Free. Planning, instead, is a Model-Based technique. Take for example the game of Chess. We get an agent and an opponent. It is a Markov Decision Problem, as the state is the current position, and it is enough to choose the next move. The past is not relevant at all. We know the environment and we know in which state we will end up if we do a particular move. Instead, in video games, you might never know what the possible evolutions are.

In principle, you should try to learn the model, but most of the techniques we consider are able to choose actions without being able to know the environment (e.g. driving the car).

Exploration vs Exploitation

Behaviors are compared along trajectories in the future. Plus, being Model-Free, we need to explore the environment, also making mistakes, so to learn which actions to do and which you should avoid. Exploration is a random activity. But only using randomicity is not the best solution, as when we're working with a Brownian movement, we can manage to explore only a relatively small area around the initial state. The strategy is to move around for a while, then reach a new state, ideally something you're not used to, and restart the random exploration. This is an approximation, as you can't freely explore.

This is called the Exploration/Exploitation Trade-Off:

- Exploration → finding more information about the environment, usually requiring randomicity
- Exploitation → take advantage of the available information to direct and possibly improve the exploration

In Model-Free approach, we have two main classes:

- Value-Based Techniques → try to give an estimation of the quality of the state you're in. The policy is implicit: we choose the action taking us to the next state with the best evaluation.
- Policy-Based Techniques → you don't evaluate the state. You have the current policy and you try to improve the local policy, where $\pi(s)$ is the probability to perform action a in the state s

Value-Based Approaches

We have two basic evaluation functions $V(s)$ (How good is a state) and $Q(s,a)$ (How good is an action a in a given state s). We suppose that the policy π is fixed and we know it. These expectations are all on all the trajectories defined by the given strategy.

$$V(s) = \mathbb{E}_{s_0=s} \sum_{t\geq 0} \gamma^t r_t$$
$$Q(s,a) = \mathbb{E}_{s_0=s, a_0=a} \sum_{t\geq 0} \gamma^t r_t$$

Can we relate $V(s)$ and $Q(s,a)$? We can compute V from Q - we sum every $Q(s,a)$ value weighted by the probability to take that action - but the other way round?

$$V(s) = \sum_a \pi(a|s) * Q(s,a)$$

The other way round is computed using a state s' (we want to know how good is the action a starting from state s) → but you need to know the probability to end up in s' given s and a. But we supposedly not know that if we are in Model-Free Approaches. In Model-Based Techniques you can use the value function for evaluation since you know in which state you'll end up (minmax game) [Here lies the great difference]

$$Q(s,a) = \sum_{s'} \mathcal{P}(s'|s,a) * V(s')$$

If we can obtain the Q value for the optimal strategy, we can derive which action is the best. The Optimal Q-value function is the maximum expected cumulative reward achievable from state s performing action a.

$$Q^*(s,a) = \max_\pi \mathbb{E}_{s_0=s,a_0=a} \sum_{t\geq 0} \gamma^t r_t$$

The optimal policy π^* then consists in taking the best action in every state as specified by Q^*. How do I compute it though?

We want to satisfy the fixed point equation → Bellman Equation. From a state s_t, when we perform an action a_t, we move to s_{t+1}. Suppose we know the value of the optimal strategy $Q^*(s_t, a_t)$, which is the expected future reward we obtain if we perform this action a_t in this state. And we know that we obtained a deterministic reward r_t, associated to the action a_t. Now I want to evaluate this state:

$$Q^*(s_t, a_t) = r_t + \max_a Q^*(s_{t+1}, a)$$

$\max_a Q^*(s_{t+1}, a)$ is the maximum reward we can expect in state $t+1$ if we perform the action a. We're evaluating how good is this move starting from this state. The two sides of the equation represent the same thing. This equation is the Bellman Equation, which is a keypoint for learning the optimal Q-function. At some point, we have an approximation, and we want to reach a fixed point as soon as possible. We want that the function satisfies for each s and each a this function.

The Bellman Equation expresses a relation between the solution for a given problem in terms of the solutions for subproblems - this is the non-deterministic version, which contains the expectation of falling in s'.

$$Q^*(s,a) = \mathbb{E}_{s'}[r_0 + \gamma \max_{a'} Q^*(s', a')]$$

$\max_{a'} Q^*(s,a) = V^*(s')$ is the optimal future cumulative reward from s' and the optimal future cumulative reward from s' when taking action a is $r_0 + \gamma R_{s'}$.

As we know that Q^* satisfies the Bellman equation, the idea is to use this to perform iterative update on progressive approximations Q^i of Q^*:

$$Q^{i+1}(s,a) \leftarrow Q^i(s,a) + \alpha(r_0 + \gamma \max_{a'} Q^i(s', a') - Q^i(s,a))$$

with α the learning rate.

The recursive update is the derivative of the quadratic distance between $Q^i(s,a)$ and $r_0 + \gamma \max_{a'} Q^i(s',a')$ that should be equal, according to the Bellman Equation

```
0 - initialize the QTable
1 - repeat until termination
  .1 - choose action a in current state s according to the current Qtable
  .2 - perform action a and observe reward r and new state s'
  .3 - update the table
```

In order to perform the update, all the information we need is contained in a transition tuple: (s, a, r, T, s'), which contains the current state, the action done, the reward obtained, a boolean stating if we terminated and the new state after doing the action. These transitions are collected by exploring the environment, with each tuple independent from the others. These tuples can be saved into an experience replay buffer that can be re-executed at leisure. It is better than learning from batches of consecutive samples because:

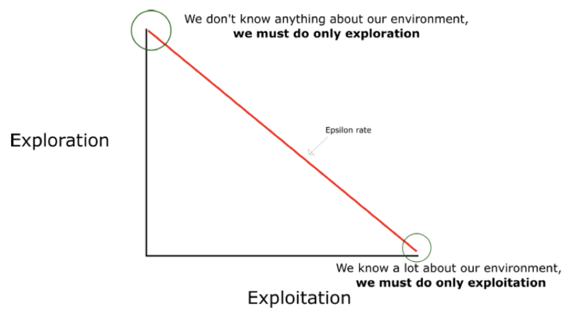
- consecutive samples are correlated, so this makes it an inefficient learning
- there's a great risk of introducing biases during the learning by exploiting an unbalanced set of transitions

Q-learning is an off-policy technique: it does not rely on any policy and only needs local transitions.

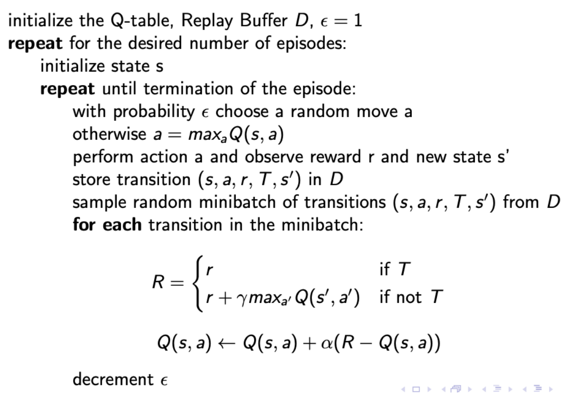
The Epsilon Greedy Strategy

At start, the Q-table is not really informative. Taking actions according to it could introduce biases and actually prevent exploration. In early stages, we want to privilege random exploration and start relying more on the table when more experience is acquired.

We specify an exploration rate ϵ, which is initially set equal to 1. This is the randomly done rate of steps. We generate a random number. If it is larger than ϵ, then we choose the action according to the information corrected in the Q-table (exploitation); otherwise we choose the action at random (exploration). We progressively reduce ϵ along training (we become more confident in our Q-table).


We don't know anything about our environment,
we must do only exploration
Exploration
Epsilon rate
We know a lot about our environment,
we must do only exploitation
Exploitation

There is no harm in exploring, so one does not go all the way in the exploitation direction



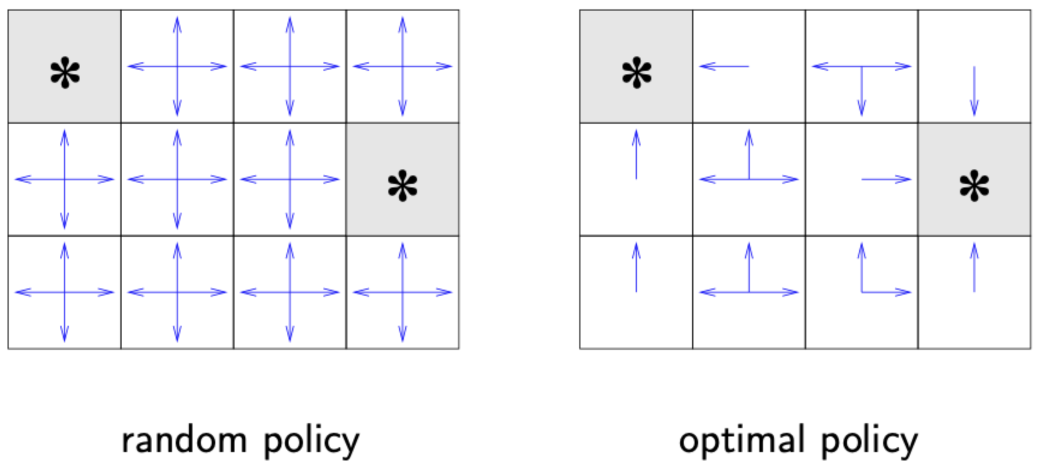
```
initialize the Q-table, Replay Buffer $D$, $\epsilon = 1$
repeat for the desired number of episodes:
    initialize state s
    repeat until termination of the episode:
        with probability $\epsilon$ choose a random move a
        otherwise $a = max_a Q(s,a)$
        perform action a and observe reward r and new state s'
        store transition $(s,a,r,T,s')$ in $D$
        sample random minibatch of transitions $(s,a,r,T,s')$ from $D$
        for each transition in the minibatch:
            $R = \begin{cases} r & \text{if } T \\ r + \gamma max_{a'} Q(s',a') & \text{if not } T \end{cases}$
            $Q(s,a) \leftarrow Q(s,a) + \alpha(R - Q(s,a))$
        decrement $\epsilon$
```

New version of the Q-learning pseudocode

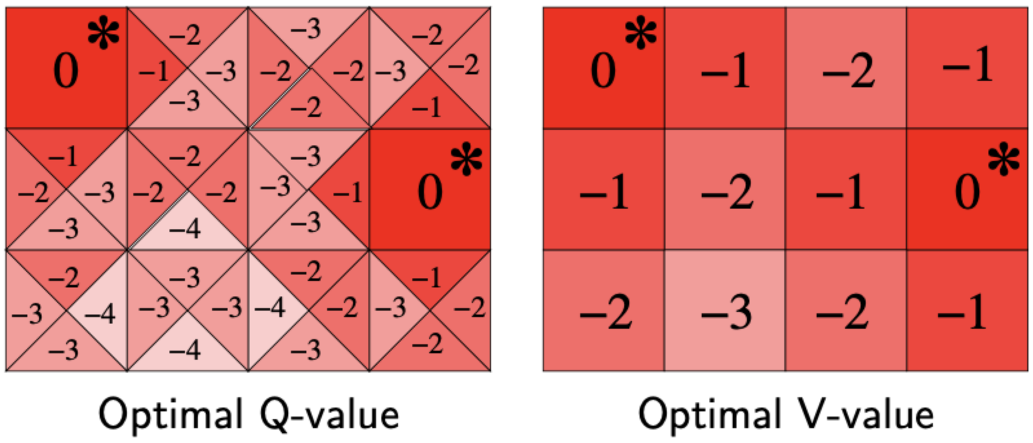
Example: the Grid World

@May 10, 2023

Abstraction in which each cell is a different state, in grey we have a final state. We have four actions possible and the objective is to find the grey state. Typical reward is a small negative reward for each transition, makes it possible to reach the exit in the minimum number of steps. The visibility of the agent is confined to its current cell. You only know where you are, but not the location of the final states; we work by trial and error. Random policy → same probability for each possible action. Optimal Policy → depending from where you are, you have a smaller set of optimal moves you can do and that have the same probabilities.



Design Q-value and V-value, keeping in mind that the Q-value table satisfies Bellman's Equation.



The algorithm looks at local updates, but local updates are relative to the state you end up where you move. So, if the estimations are not correct, you're not really sure of where you end up: in fact, when we start, we only know the right values for terminal states, so most of the actions produce meaningless updates, since the estimations of the Q-value function is erroneous. The relevant actions are those leading to states whose Q-value is accurate, which at the beginning, are only the terminal states.

Lesson 20: Deep Q-Learning (DQN)

We consider Q-learning the classical algorithm in table of (states,action). The problem is that in general, Bellman's Equation is not scalable. The table is too large. Too many states. When you have a videogame, you have a limited set of actions , the problem are the states. The solution would be use a Neural Network (approach goes by the name of Deep Q-Learning $\rightarrow$ instead of computing Q by means of a table, we replace it with a Neural Network parametrized by some θ and we try to learn the objective function.

$$Q(s, a, \theta) \approx Q^*(s, a)$$

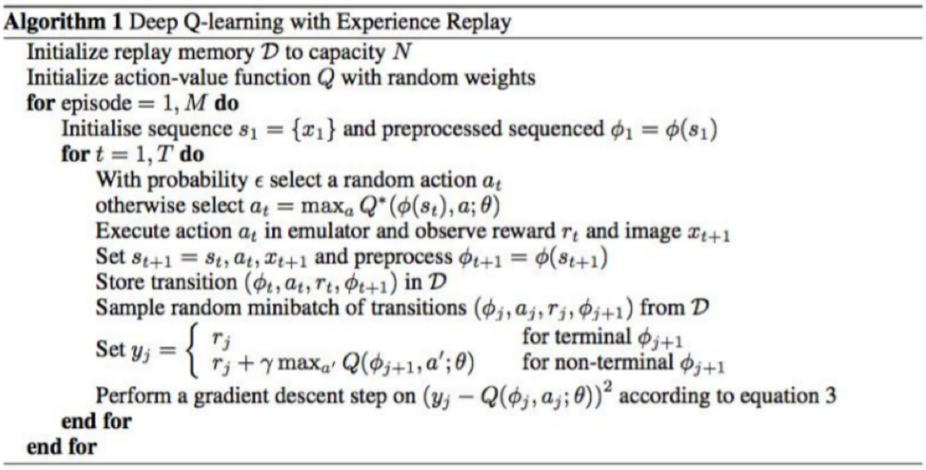
Slight modification $\rightarrow$ the Q-table takes in input state action and returns the Q-value. In DQN, we take in input the state, and for each action, you return a Q-value (the two functions are isomorphic).

How do we train: we want the Q-function and we want to satisfy the Bellman Equation (we put Bellman Equation as the objective function). The loss is then:

$$L(\theta) = (\mathbb{E}_{s'}[r_0 + \gamma \max_{a'} Q(s', a')] - Q(s, a, \theta))^2$$

To compute the loss function, we need transitions. We store these transitions in an experience memory and then replay them at leisure during training (experience replay). This is more beneficial than learning from batches of consecutive samples as:

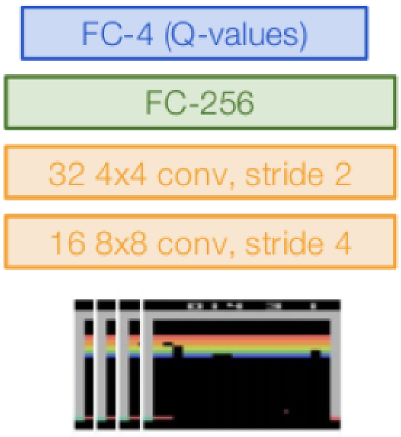
- samples tend to be correlated, and this leads to inefficient learning
- great risk of introducing biases during learning



Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

Structure of the Network used for the Atari Games → single network and training algorithm for all the games. Specific training for each game, with the same architecture. Inputs are frames. Does a single frame contain all the information we need? No, from a single frame we cannot derive information about movement (e.g. we're playing Pong). Two should be enough, but we consider a stack of 4. Also, a decision should be if take adjacent frames or skip some. They're not 4 consecutive frames, but they are distanced (+1). They transform it to grayscale and resize it. Processed with a couple convolutional layers (old structure, we don't convolve like this anymore). Not all values are significant



An alternative to stacking was processing the state with an LSTM at the end of the state processing, before computing QValues.

Human Level → Level of an average player (not an expert who's received a real training). We have a distinction between games that have a reactive nature (you see an action going on and you have to react, e.g. Pong) and Games that involve planning (e.g. Montezuma's Revenge or Pac-Man, for the position of the Ghosts, and the path decision), for which the approach does not really work.

The Reward System is the same for all games: +1 for positive results, -1 for negative and 0 for neutral. Rewards should not really matter, you risk biasing the agent → should just define a basic mechanism of scores which is coherent with the intended result.

Fixed Q-Targets

We use as gradient of the loss the difference between the approximated target and the value estimated by the network → the network appears twice in the equation, as it is also in the Bellman's Equation. We have what is called "Moving Target" → we try to approximate to a target that is moving itself.

Instead of using a single network, you use two network: one you update, and one to compute the target value, so when you back-propagate you modify only one. In the long run you want the same network, so periodically you copy the weights.

$$r_0 + \gamma \max_{a'} \overline{Q(s', a')} - Q(s, a)$$

Implementation:

- You create two identical networks, which we call DQNetwork (the one you update) and TargetNetwork
- Define a function to transfer parameters from DQNetwork to TargetNetwork
- We use the target network to estimate the future reward, but we update only the DQNetwork. Every τ steps, we update the TargetNetwork with the DQNetwork (τ is a hyper-parameter)

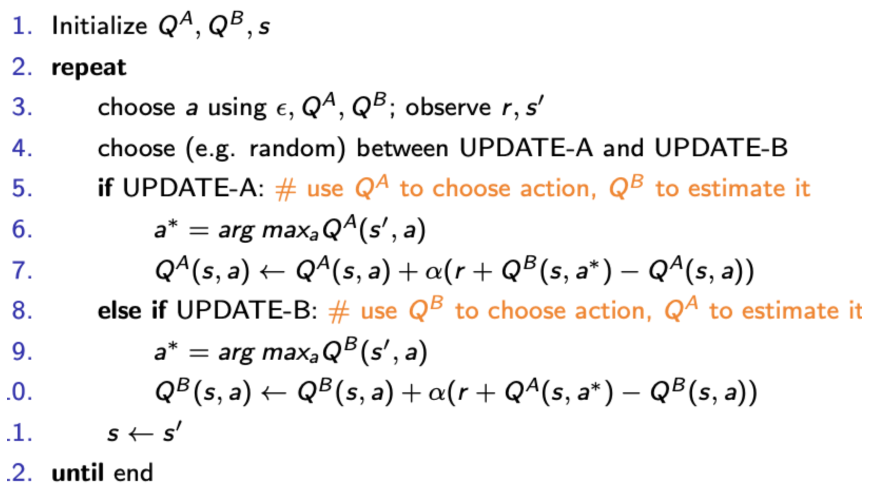
Empirically proven to be working → This decoupling is often used in implementation

Double Q-Learning

Also require two networks. The problem here is that we approximate the target action value computing a maximum over all the actions. The idea is that the Q-function is a noisy approximation of the real value, and we're systematically taking the maximum value → can be proved that we're systematically inducing an overestimation of the correct value. Decouple:

- One network that introduces the bias (takes the action that maximizes the value)

- One network that gives a second opinion (computes the future reward from the state s' if you choose the action found by the other network).



```
1.  Initialize $Q^A, Q^B, s$
2.  repeat
3.      choose $a$ using $\epsilon, Q^A, Q^B$; observe $r, s'$
4.      choose (e.g. random) between UPDATE-A and UPDATE-B
5.      if UPDATE-A: # use $Q^A$ to choose action, $Q^B$ to estimate it
6.          $a^* = arg\ max_a Q^A(s', a)$
7.          $Q^A(s, a) \leftarrow Q^A(s, a) + \alpha(r + Q^B(s, a^*) - Q^A(s, a))$
8.      else if UPDATE-B: # use $Q^B$ to choose action, $Q^A$ to estimate it
9.          $a^* = arg\ max_a Q^B(s', a)$
.0.         $Q^B(s, a) \leftarrow Q^B(s, a) + \alpha(r + Q^A(s, a^*) - Q^B(s, a))$
.1.     $s \leftarrow s'$
.2. until end
```

Essentially, you have a network that calculates the value and one that checks.

Prioritized Experience Replay

We know (and the agent knows), that the Replay Buffer memorizes past experiences and can use them to improve learning. We also know that some actions may be more important than others and are more worth to be remembered.

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

Choice adopted is to use the δ_t as a weight (close to 0 = minimal update). If you're inducing no error, it does not make sense to remember it.

$$p_t = |\delta_t|$$
$$P_t = \frac{p_t^\alpha}{\sum_t p_t^\alpha}$$

We want to associate a probability to these scores → normalize the δ_ts. We obtain the probability to pick up the transition during the next batch. If all the scores are equal, also the probabilities are equal. The higher the α, the higher the emphasis on the scores. Can use this parameter for tuning.

Problem: we introduce a bias toward high-probability samples → especially at the beginning of a sentence, we might overfit over a small portion of experiences we presume to be interesting.

We can weight the updates to correct this issue, compensating the harm introduced. You though never want to increase too much the δ

$$w_t = (N \cdot P)^{-\beta}$$

If β=1 we reduce the weights of high-probability transitions; β goes to 0 during training. The weights are folded into the Q-learning update by using $w_t \delta_t$ instead of δ_t.

For stability reasons, weights are normalized by $\frac{1}{\max_t w_t}$ so that they only scale the update δ_t downwards.

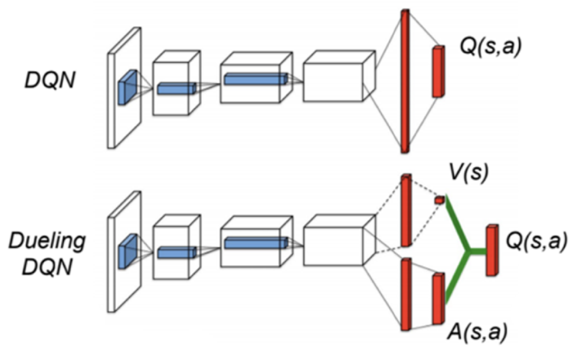
Dueling

Each Q-value $Q(s,a)$ estimates how good it is to take the action a in state s: it depends on both the values. We can decompose $Q(s,a)$ as the sum:

$$Q(s,a) = V(s) + A(s,a)$$

- $V(s)$ → the value of being at state s
- $A(s,a)$ → the advantage of taking the action a in state s, measuring how much better it is to take action a versus all other possible actions in that state.

The Dueling Network Architectures split the computation of $Q(s,a)$ into the computation of $V(s)$ and $A(s,a)$, two different streams.



Intuitively, the dueling architecture can learn which states are (not) valuable, without having to learn the effect of each action for each state. This is particularly useful states where actions do not affect the environment in any relevant way; conversely, in states in which the action is indeed relevant, it can focus on the advantage without caring for the current evaluation of the state.

The one stated before is the naïf aggregation, which has a problem: given $Q(s,a)$ it is impossible to recover $V(s)$ and $A(s,a)$ and hence it is difficult to distribute the error during backpropagation (identifiability)

The solution is to force A to have zero advantage for the best action a^*:

$$Q(s,a) = V(s) + A(s,a) - \max_a A(s,a)$$

which for $a^* = \arg\max_a A(s,a)$, makes $A(s,a^*) = 0$ and $Q(s,a^*) = 0$

Eventually, one would replace the $\max$ with a better performing mean, which improves stability during training.

Noisy Networks

Characterized by Noisy Dense Layers, combining a deterministic and a noisy stream:

$$y = b + Wx + (b_{noisy} \odot \epsilon^b + (W_{noisy} \odot \epsilon^W)x)$$

where:

- $W, b, W_{noisy}, b_{noisy}$ are learned parameters
- ϵ^W, ϵ^b are randomly generated

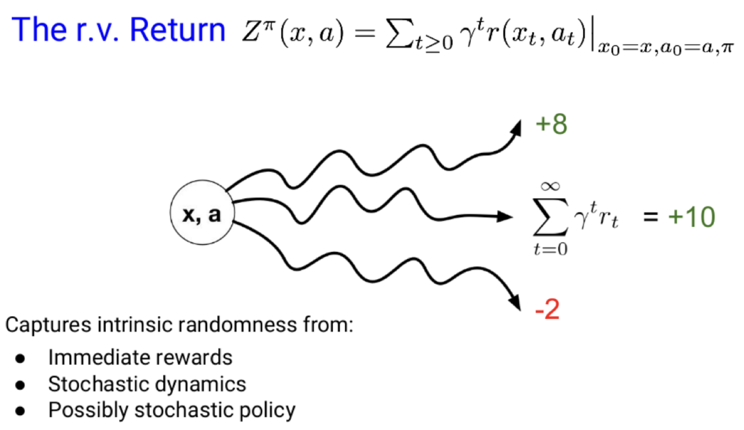
We use this layer in substitution of any standard dense layer, doubling the number of parameters.

The purpose of the noise is to augment the randomicity in the choice of actions. Since noisy-weights are learned and the resulting noise is state dependent, this allows the network to randomly explore the environment at different rates in different parts of the state space.

This architecture replace and improves the performance w.r.t. the ϵ-greedy strategy.

Distributional RL

Tries to learn the probability distribution of the future cumulative reward, instead of the traditional approach of modeling the expectation of this return. DRL addresses the random return Z whose expectation is the value Q


The r.v. Return $Z^{\pi}(x,a) = \sum_{t\geq 0} \gamma^t r(x_t, a_t)\big|_{x_0=x, a_0=a, \pi}$

Captures intrinsic randomness from:

- Immediate rewards
- Stochastic dynamics
- Possibly stochastic policy

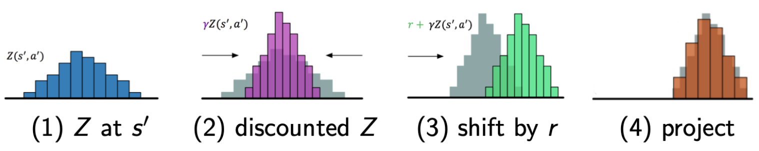
Let a^* be the best possible action in s'. Bellman is:

$$Q(x,a) = R(x,a) + \gamma(x', a^*)$$

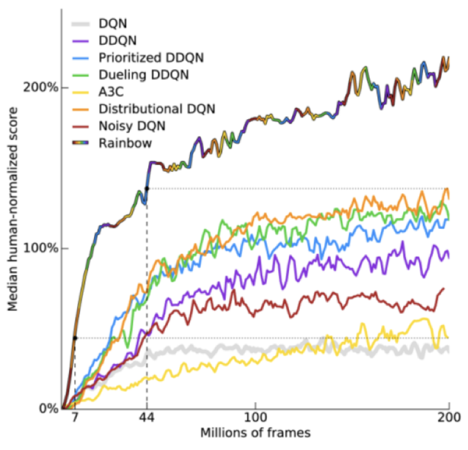
and similarly:

$$Z(x,a) = R(x,a) + \gamma Z(s', a^*)$$

The distribution is discretized over a support in a given range between V_{min} and V_{max} using a fixed number of bins. The loss between the two distributions can be computed using KL-divergence



Rainbow



Lesson 21: Policy Gradient Techniques

One of the problems of Q-learning is that it is a 1-step method: updates the action value $Q(s, a)$ towards the one-step return $r + \gamma \max_{a'} Q^i(s', a')$. This way, we only make updates on a specific state and a particular action, which are part of the transition considered. The values of other (state, action) pairs is affected only indirectly (the updates are local). The learning process is therefore slow due to the number of updates required to propagate the reward. Can we learn the policy directly? We try to learn to improve the policy.

- Q-learning is an off-policy technique $\rightarrow$ it does not rely on any policy and only needs local transition, you can acquire knowledge using an arbitrary policy; in particular, we can take advantage of experience replay.
- On policy techniques try to improve the current policy $\rightarrow$ requires sampling long trajectories according to the current strategy and needs many diversified trajectories (e.g. parallel agents)

SARSA

Difference with Q-learning. Consider the update in Q-learning.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

In the case of SARSA, we just consider an extra action instead of the best, under the current policy.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

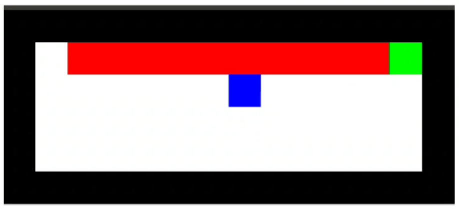
Instead of choosing the maximum (greedy) possible - we don't consider all the possible actions, but just one extra action - we consider the actual action.

Q-learning is based on single-step transitions, while SARSA is based on mini-trajectories of two steps (= two actions): $(s_i, a_i, r_i, s_{t+1}, a_{t+1})$ - you can see why it is called SARSA. This is why SARSA is considered an on-policy strategy.

Consider we're acting with a completely randomized policy: in SARSA the action is taken randomly and in a sense when we consider the value of the state we are considering a sort of min value. If little by little we specialize our behavior, in the end the action that we will choose will coincide with the best behavior, minimizing the difference with Q-learning. Can be proven that supposing that eventually the policy will approximate the optimal policy, these two techniques essentially coincide.

Example: Mouse and Cliff Scenario

A mouse (blue) is trying to get to a piece of cheese (green).
Additionally, there is a cliff in the map (red) that must be avoided,
or the mouse falls and dies.



If you try to use Q-learning, you will eventually learn the optimal strategy, which is running along the edge, occasionally jumping off and plummeting. This is though the most dangerous path. One would maybe like to stay safer. With SARSA, the mouse learns through small trajectories, so he knows it could die by staying in the border, so the path is different, safer. If in the end you remove any randomicity you know you can always act optimally, you fall into the Q-learning path.

Policy Gradients

Formalize the problem: how can we improve a strategy? Suppose we have a class of parametrized policies (=depending on some parameters θ). $\Pi = \{\pi_\theta\}$. For each policy π_θ we can define the expected cumulative reward $J(\theta)$

$$J(\theta) = \mathbb{E} \sum_{t \geq 0} \gamma^t r_t$$

We want to find the optimal strategy with the objective to optimize θ^*

$$\theta^* = \arg\max_\theta J(\theta)$$

We address the problem by gradient ascent on policy parameters.

The main idea underlying policy gradients is reinforcing good actions: to push up the probabilities of actions that lead to a higher return and push down the probabilities of actions that lead to a lower return, until you arrive at the optimal policy. The policy gradient method will iteratively amend the policy network weights (with smooth updates) to make state-action pairs that resulted in positive return more likely, and make state-action pairs that resulted in a negative return less likely.

REINFORCE Approach

It is an old algorithm (one of the many approaches). The idea is: we sample along trajectory, for each trajectory we measure the future cumulative reward we collect and use it as an estimate of how good is the action in the state and then update this quantity to the parameters of the network with the gradient.

$$\nabla_\theta \log \pi(a_t|s_t, \theta)R_t$$

If R_t is high we expect a strong reward then we improve the probability, we push it up. Else we push it down.

One of the problems with this technique is that we are evaluating a single action, so we’re not interested in the whole future cumulative reward (the raw value of a trajectory is not so meaningful). Introduce a baseline to evaluate the actions not in an absolute sense.

$$\nabla_\theta \log_\pi(a_t|s_t, \theta)(R_t - b(s_t))$$

Instead of just taking the reward, we take the difference between the reward and a base value for the state.

A possible and excellent choice for the baseline is the value function of the state

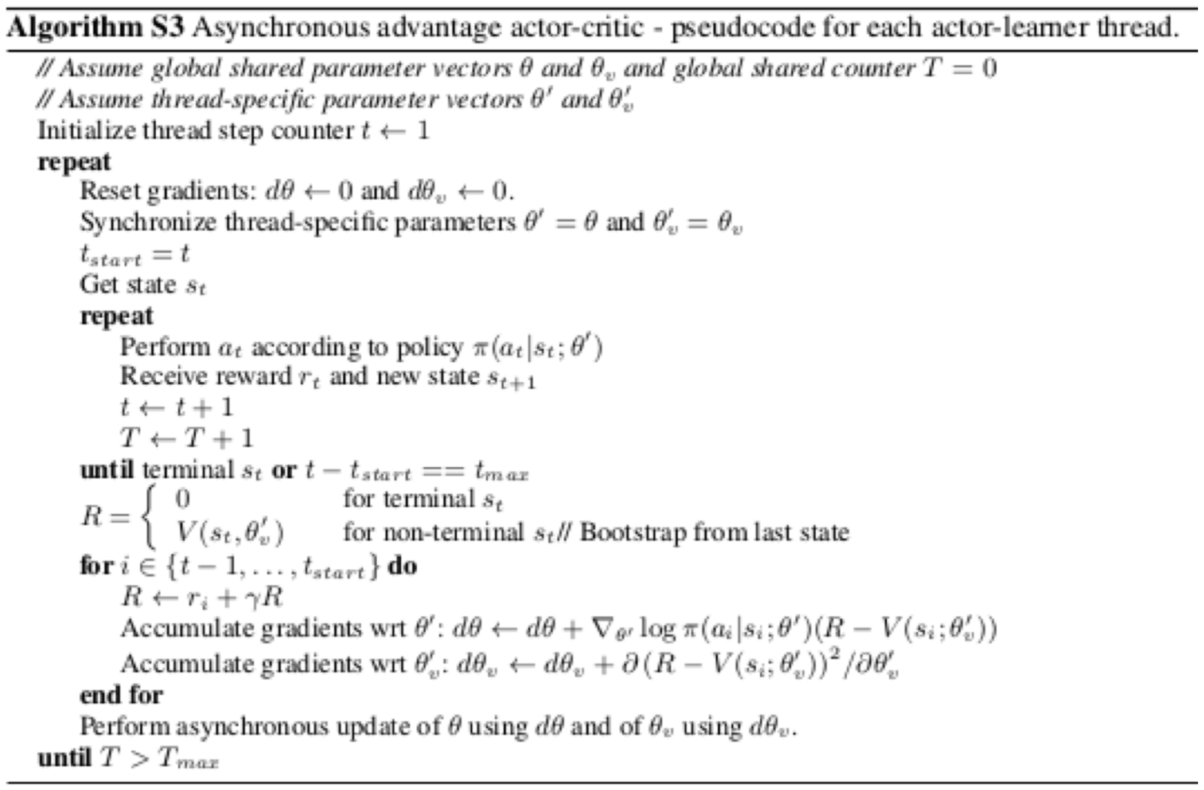
$$b(s_t) = V^\pi(s_t)$$

This is also called actor-critic architecture where the policy π is the actor and the value function (baseline) is the critic. We have an actor that acts according to a given policy and a critic that evaluates, assessing the quality of the policy.

A3C Pseudo-Code

(Asynchronous Advantage Actor-Critic)

Introduced after DQN and proved to behave better for Atari Games. Advantage comes from the fact that the quantity $R_t - b(s_t)$ is called Advantage = how this action is advantageous in this state. We jointly try and learn both the Actor and the Critic. And Asynchronous because we need to explore the universe according to many different trajectories, so it uses asynchronous threads (this has been criticized later, so we have also a synchronous version: A2C)



Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

```
// Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared counter $T = 0$
// Assume thread-specific parameter vectors $\theta'$ and $\theta'_v$
Initialize thread step counter $t \leftarrow 1$
repeat
    Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
    Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
    $t_{start} = t$
    Get state $s_t$
    repeat
        Perform $a_t$ according to policy $\pi(a_t|s_t;\theta')$
        Receive reward $r_t$ and new state $s_{t+1}$
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    until terminal $s_t$ or $t - t_{start} == t_{max}$
    $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
    for $i \in \{t-1, \ldots, t_{start}\}$ do
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients wrt $\theta'$: $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
        Accumulate gradients wrt $\theta'_v$: $d\theta_v \leftarrow d\theta_v + \partial(R - V(s_i;\theta'_v))^2 / \partial\theta'_v$
    end for
    Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$.
until $T > T_{max}$
```

We need a network that given the state returns the value (critic) and another network that given the state returns the probability distribution over all the actions (actor). They could be two distinct networks, but often they share components (have an initial structure in common - the extraction part - and then have two distinct dense layers at the end)

A2C is a single-worker variant of A3C, which produces more efficient and comparable performances.

Proximal Policy Optimization (PPO)

We have some problems with Policy Gradient Methods:

- Sample Inefficiency → Samples (frequently) are only used once. When policy is updated, the new policy is used to sample another trajectory. As sampling is often expensive - we need to interact with the environment, this can be prohibitive. Difficult to apply experience replay techniques. However, after a large policy update, the old samples are no longer representative, as they are based on no longer significant policies.
- Inconsistent policy updates → Policy updates tend to overshoot and frequently miss the reward peak, or stall prematurely. Vanishing and exploding gradients are typical and severe problems. The algorithm may not recover from a poor update.
- High reward variance → Policy Gradient is a Monte Carlo learning approach, taking into account the full reward trajectory. Such trajectories often suffer from high variance,

hampering convergence (which I can partially address by adding a critic - intervene on the advantage)

When passing from a given policy π_k (e.g. a randomized version of the current policy) to a new policy π, even small modifications can easily result in large fluctuations in behaviors and performances. How can we take the biggest possible improvement step on a policy, still remaining inside a trusted region (i.e. without stepping so far that we accidentally cause performance collapse)?

Let π_θ be a policy with parameters θ. In theory, the TRPO (Trusted Region Policy Optimization) - very complex - update is:

$$\theta_{k+1} = \arg\max_\theta \mathcal{L}(\theta_k, \theta) \text{ s.t. } \overline{KL}(\theta \| \theta_k) \leq \delta$$

where $\mathcal{L}(\theta_k, \theta)$ is the surrogate advantage, a measure of how the policy π_θ performs relative to the old policy π_{θ_k} using data from the old policy ($A^{\pi_{\theta_k}}(s, a)$ is the advantage of a in s). We want to keep the "distance" - the KL divergence - below a certain threshold.

$$\mathcal{L}(\theta_k, \theta) = \mathbb{E}_{s,a\sim\pi_{\theta_k}} \frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a)$$

and $\overline{KL}$ is an average Kullback-Leibler Divergence between policies across states visited by the old policy:

$$\overline{KL}(\theta \| \theta_k) = \mathbb{E}_{s\sim\pi_{\theta_k}} KL(\pi_\theta(\cdot|s) \| \pi_{\theta_k}(\cdot|s))$$

The theoretical update is hard to implement. TRPO makes approximations based on Taylor Expansions to improve efficiency (also in order to be able to integrate and make sure that the constraints are satisfied).

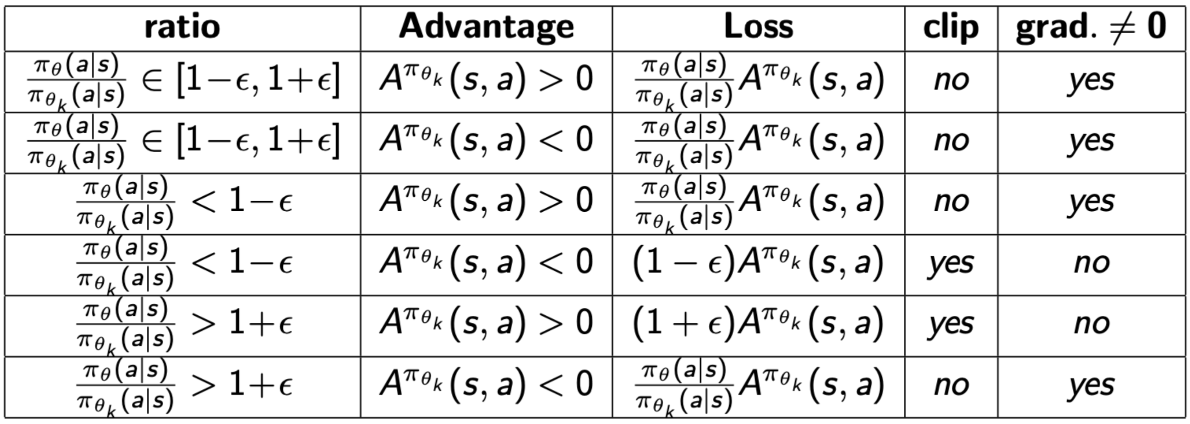
PPO - simpler version, considered state-of-the-art - achieves a similar objective by updating policies via:

$$\theta_{k+1} = \arg\max_\theta \mathbb{E}_{s,a\sim\pi_{\theta_k}} [L(s, a, \theta_k, \theta)]$$

where

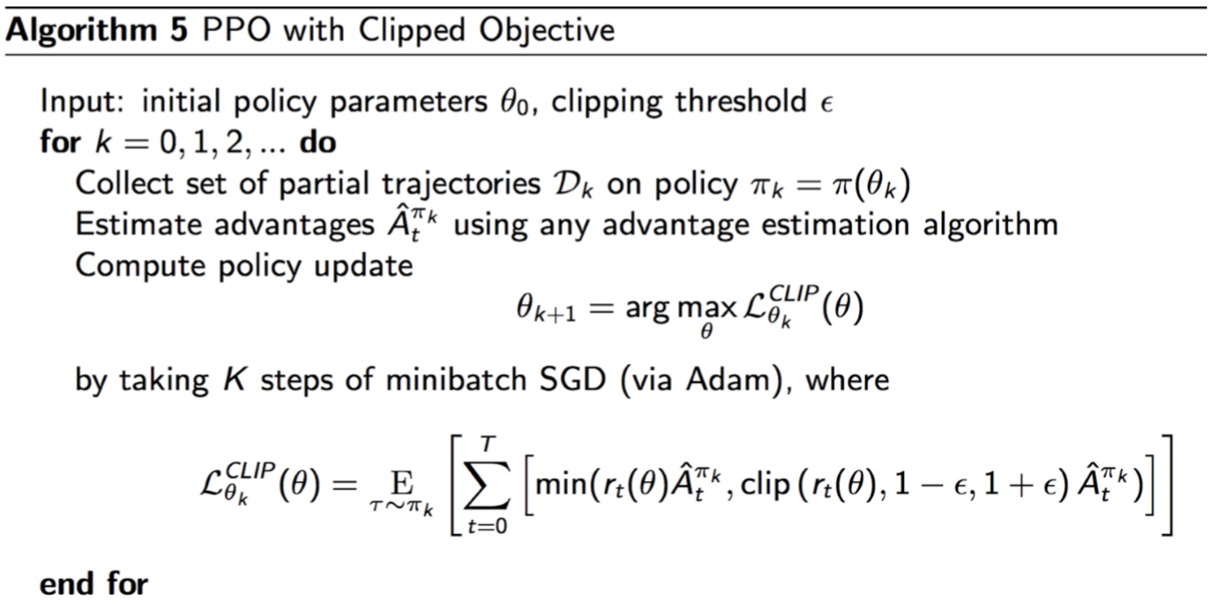
$$L(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), clip\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(s, a)\right)$$

The probability ratio is clipped in a range $[1-\epsilon, 1+\epsilon]$, but only if the resulting loss is larger than the unclipped value; in this way, the final objective is a lower (i.e. pessimistic) bound on the unclipped objective. We essentially don't want the probability to have a high value modification.



ratio	Advantage	Loss	clip	grad. $\neq 0$
$\frac{\pi_\theta(a\|s)}{\pi_{\theta_k}(a\|s)} \in [1-\epsilon, 1+\epsilon]$	$A^{\pi_{\theta_k}}(s,a) > 0$	$\frac{\pi_\theta(a\|s)}{\pi_{\theta_k}(a\|s)} A^{\pi_{\theta_k}}(s,a)$	*no*	*yes*
$\frac{\pi_\theta(a\|s)}{\pi_{\theta_k}(a\|s)} \in [1-\epsilon, 1+\epsilon]$	$A^{\pi_{\theta_k}}(s,a) < 0$	$\frac{\pi_\theta(a\|s)}{\pi_{\theta_k}(a\|s)} A^{\pi_{\theta_k}}(s,a)$	*no*	*yes*
$\frac{\pi_\theta(a\|s)}{\pi_{\theta_k}(a\|s)} < 1-\epsilon$	$A^{\pi_{\theta_k}}(s,a) > 0$	$\frac{\pi_\theta(a\|s)}{\pi_{\theta_k}(a\|s)} A^{\pi_{\theta_k}}(s,a)$	*no*	*yes*
$\frac{\pi_\theta(a\|s)}{\pi_{\theta_k}(a\|s)} < 1-\epsilon$	$A^{\pi_{\theta_k}}(s,a) < 0$	$(1-\epsilon) A^{\pi_{\theta_k}}(s,a)$	*yes*	*no*
$\frac{\pi_\theta(a\|s)}{\pi_{\theta_k}(a\|s)} > 1+\epsilon$	$A^{\pi_{\theta_k}}(s,a) > 0$	$(1+\epsilon) A^{\pi_{\theta_k}}(s,a)$	*yes*	*no*
$\frac{\pi_\theta(a\|s)}{\pi_{\theta_k}(a\|s)} > 1+\epsilon$	$A^{\pi_{\theta_k}}(s,a) < 0$	$\frac{\pi_\theta(a\|s)}{\pi_{\theta_k}(a\|s)} A^{\pi_{\theta_k}}(s,a)$	*no*	*yes*

Possible situations in which I can find myself



Algorithm 5 PPO with Clipped Objective

```
Input: initial policy parameters $\theta_0$, clipping threshold $\epsilon$
for $k = 0, 1, 2, \ldots$ do
    Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
    Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
    Compute policy update
        $\theta_{k+1} = \arg\max_\theta \mathcal{L}_{\theta_k}^{CLIP}(\theta)$
    by taking $K$ steps of minibatch SGD (via Adam), where
        $\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathop{\mathrm{E}}_{\tau \sim \pi_k}\left[\sum_{t=0}^{T}\left[\min(r_t(\theta)\hat{A}_t^{\pi_k}, \mathrm{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t^{\pi_k})\right]\right]$
end for
```