Program : **B.Tech**

Subject Name: **Machine Learning**

Subject Code: **CS-601**

Semester: **6th**

Department of Computer Science and Engineering
Subject Notes
CS 601- Machine Learning
Unit 4

Recurrent neural network

Recurrent Neural Network (RNN) are a type of Neural Network where the output from previous step are fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other, but in cases like when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus, RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is Hidden state, which remembers some information about a sequence.

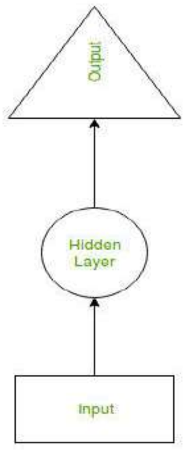


Figure : 4.1

RNN have a "memory" which remembers all information about what has been calculated. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output. This reduces the complexity of parameters, unlike other neural networks.
It's part of the network. RNNs can take one or more input vectors and produce one or more output vectors and the output(s) are influenced not just by weights applied on inputs like a regular NN, but also by a "hidden" state vector representing the context based on prior input(s)/output(s). So, the same input could produce a different output depending on previous inputs in the series.

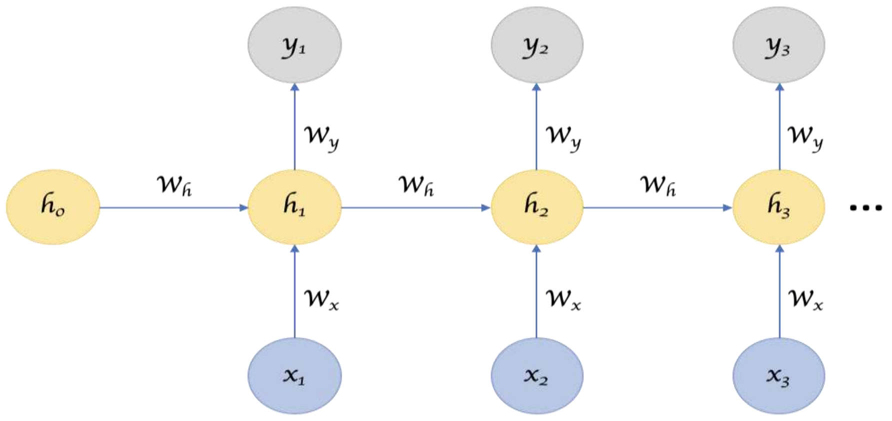


Figure :4.2 A Recurrent Neural Network, with a hidden state that is meant to carry pertinent information from one input item in the series to others.

The formula for the current state can be written as –

$$h_t = f(h_{t-1}, x_t)$$

Here, Ht is the new state, ht-1 is the previous state while xt is the current input. We now have a state of the previous input instead of the input itself, because the input neuron would have applied the transformations on our previous input. So each successive input is called as a time step.
Taking the simplest form of a recurrent neural network, let's say that the activation function is tanh, the weight at the recurrent neuron is Whh and the weight at the input neuron is Wxh, we can write the equation for the state at time t as –

$$h_t = \tanh\ (W_{hh}h_{t-1}+ W_{xh}x_t)$$

The Recurrent neuron in this case is just taking the immediate previous state into consideration. For longer sequences the equation can involve multiple such states. Once the final state is calculated we can go on to produce the output
Now, once the current state is calculated we can calculate the output state as-

$$y_t = W_{hy}h_t$$

Training through RNN

1. A single time step of the input is supplied to the network i.e. xt is supplied to the network
2. We then calculate its current state using a combination of the current input and the previous state i.e. we calculate ht
3. The current ht becomes ht-1 for the next time step
4. We can go as many time steps as the problem demands and combine the information from all the previous states
5. Once all the time steps are completed the final current state is used to calculate the output yt
6. The output is then compared to the actual output and the error is generated
7. The error is then backpropagated to the network to update the weights (we shall go into the details of backpropagation in further sections) and the network is trained

Advantages of Recurrent Neural Network

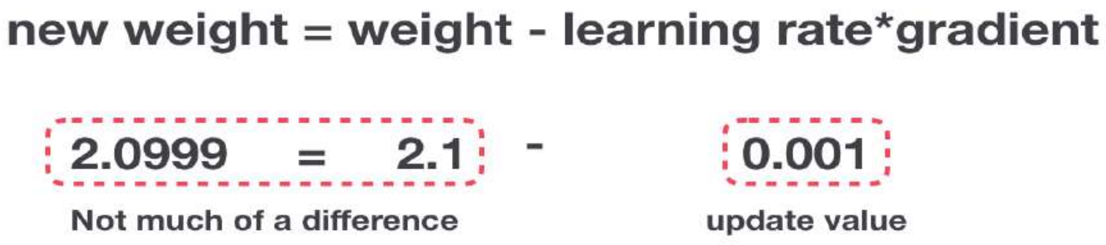
1. An RNN remembers each and every information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called Long Short-Term Memory.
2. Recurrent neural network is even used with convolutional layers to extend the effective pixel neighborhood.

Disadvantages of Recurrent Neural Network

1. Gradient vanishing and exploding problems.
2. Training an RNN is a very difficult task.
3. It cannot process very long sequences if using tanh or ReLu as an activation function.

Long short-term memory

Recurrent Neural Networks suffer from short-term memory. If a sequence is long enough, they'll have a hard time carrying information from earlier time steps to later ones. So if you are trying to process a paragraph of text to do predictions, RNN's may leave out important information from the beginning.
During back propagation, recurrent neural networks suffer from the vanishing gradient problem. Gradients are values used to update a neural networks weight. The vanishing gradient problem is when the gradient shrinks as it back propagates through time. If a gradient value becomes extremely small, it doesn't contribute too much learning.



So in recurrent neural networks, layers that get a small gradient update stops learning. Those are usually the earlier layers. So because these layers don't learn, RNN's can forget what it seen in longer sequences, thus having a short-term memory.
Long Short Term Memory is a kind of recurrent neural network is the solution of the above problem. In RNN output from the last step is fed as input in the current step. LSTM was desgined by Hochreiter & Schmidhuber. It tackled the problem of long-term dependencies of RNN in which the RNN cannot predict the word stored in the long-term memory but can give more accurate predictions from the recent information. As the gap length increases RNN does not give efficent performance. LSTM can by default retain the information for long period of time. It is used for processing, predicting and classifying on the basis of time series data.

Structure Of LSTM:

LSTM has a chain structure that contains four neural networks and different memory blocks called **cells**.

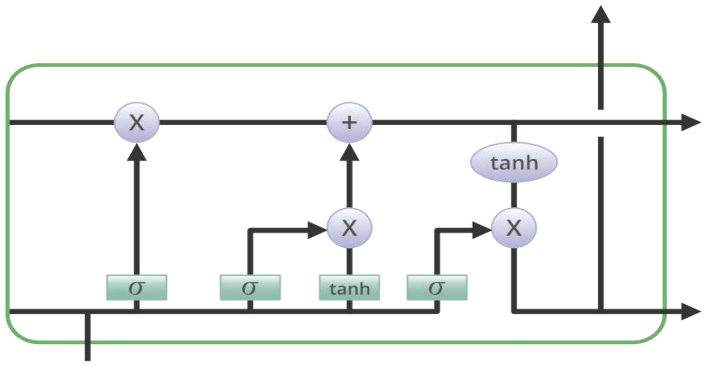


Figure: 4.3

Information is retained by the cells and the memory manipulations are done by the **gates.**

There are three gates –

1. **Forget Gate:** The information that no longer useful in the cell state is removed with the forget gate. Two inputs *x_t* (input at the particular time) and *h_t-1* (previous cell output) are fed to the gate and multiplied with weight matrices followed by the addition of bias. The resultant is passed through an activation function which gives a binary output. If for a particular cell state the output is 0, the piece of information is forgotten and for the output 1, the information is retained for the future use.

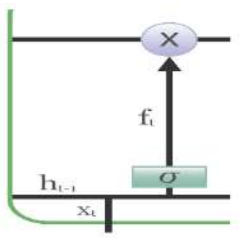


Figure: 4.4

2. **Input gate:** Addition of useful information to the cell state is done by input gate. First, the information is regulated using the sigmoid function and filter the values to be remembered similar to the forget gate using inputs *h_t-1* and *x_t*. Then, a vector is created using *tanh* function that gives output from -1 to +1, which contains all the possible values from h_t-1 and *x_t*. Atlast, the values of the vector and the regulated values are multiplied to obtain the useful information.

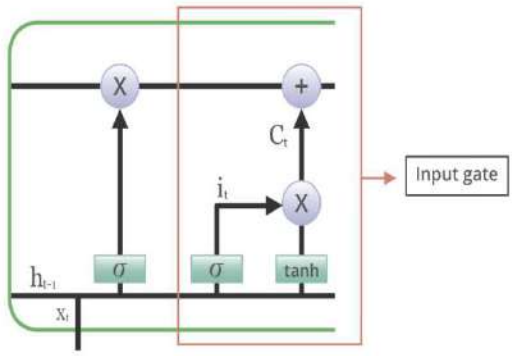


Figure: 4.5

3. **Output gate:** The task of extracting useful information from the current cell state to be presented as an output is done by output gate. First, a vector is generated by applying tanh function on the cell. Then, the information is regulated using the sigmoid function and filter the values to be remembered using inputs *h_t-1* and *x_t*. At last, the values of the vector and the regulated values are multiplied to be sent as an output and input to the next cell.

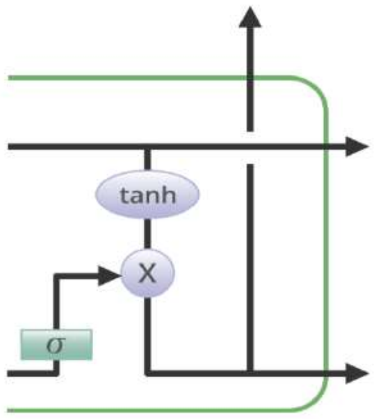


Figure: 4.6

Some of the famous applications of LSTM includes:

1. Language Modelling
2. Machine Translation

3. Image Captioning
4. Handwriting generation
5. Question Answering Chatbots

Gated recurrent unit

The GRU is the newer generation of Recurrent Neural networks and is pretty similar to an LSTM. GRU's got rid of the cell state and used the hidden state to transfer information. It also only has two gates, a reset gate and update gate.

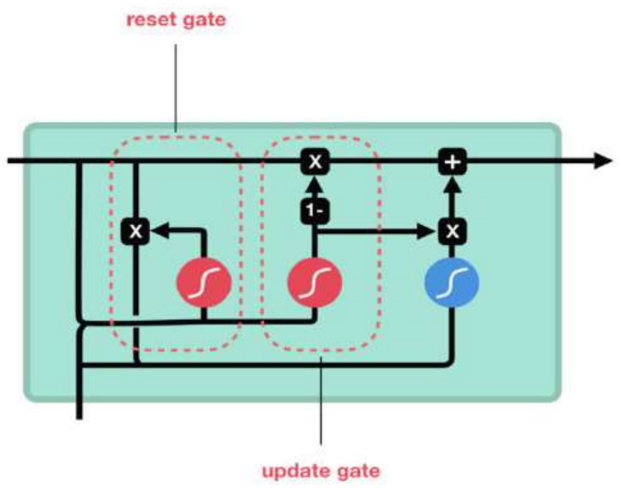


Figure: 4.7

Update Gate

The update gate acts similar to the forget and input gate of an LSTM. It decides what information to throw away and what new information to add.

Reset Gate

The reset gate is another gate is used to decide how much past information to forget.

Translation

One of the earliest goals for computers was the automatic translation of text from one language to another.

Automatic or machine translation is perhaps one of the most challenging artificial intelligence tasks given the fluidity of human language. Classically, rule-based systems were used for this task, which were replaced in the 1990s with statistical methods. More recently, deep neural network models achieve state-of-the-art results in a field that is aptly named neural machine translation.

Machine translation is the task of automatically converting source text in one language to text in another language.

In a machine translation task, the input already consists of a sequence of symbols in some language, and the computer program must convert this into a sequence of symbols in another language. Given a sequence of text in a source language, there is no one single best translation of that text to another language. This is because of the natural ambiguity and flexibility of human language. The fact is that accurate translation requires background knowledge in order to resolve ambiguity and establish the content of the sentence.

Classical machine translation methods often involve rules for converting text in the source language to the target language. The rules are often developed by linguists and may operate at the lexical, syntactic, or semantic level. This focus on rules gives the name to this area of study: Rule-based Machine Translation, or RBMT.

Statistical Machine Translation-

Statistical machine translation, or SMT for short, is the use of statistical models that learn to translate text from a source language to a target language.

Given a sentence T in the target language, we seek the sentence S from which the translator produced T. We know that our chance of error is minimized by choosing that sentence S that is most probable given T. Thus, we wish to choose S so as to maximize Pr(S|T).

The approach is data-driven, requiring only a corpus of examples with both source and target language text. This means linguists are no longer required to specify the rules of translation.

Neural Machine Translation-

Neural machine translation, or NMT for short, is the use of neural network models to learn a statistical model for machine translation.

The key benefit to the approach is that a single system can be trained directly on source and target text, no longer requiring the pipeline of specialized systems used in statistical machine learning.

Unlike the traditional phrase-based translation system which consists of many small sub-components that are tuned separately, neural machine translation attempts to build and train a single, large neural network that reads a sentence and outputs a correct translation.

Encoder-Decoder Model

Multilayer Perceptron neural network models can be used for machine translation, although the models are limited by a fixed-length input sequence where the output must be the same length.

These early models have been greatly improved upon recently through the use of recurrent neural networks organized into an encoder-decoder architecture that allow for variable length input and output sequences.

An encoder neural network reads and encodes a source sentence into a fixed-length vector. A decoder then outputs a translation from the encoded vector. The whole encoder–decoder system, which consists of the encoder and the decoder for a language pair, is jointly trained to maximize the probability of a correct translation given a source sentence.

Encoder-Decoders with Attention

Although effective, the Encoder-Decoder architecture has problems with long sequences of text to be translated.

The problem stems from the fixed-length internal representation that must be used to decode each word in the output sequence.

The solution is the use of an attention mechanism that allows the model to learn where to place attention on the input sequence as each word of the output sequence is decoded.

The encoder-decoder recurrent neural network architecture with attention is currently the state-of-the-art on some benchmark problems for machine translation. And this architecture is used in the heart of the Google Neural Machine Translation system, or GNMT, used in their Google Translate service.

Beam search and width

A machine translation model is similar to a language model except it has an encoder network placed before. For this reason, it is sometimes referred as a conditional language model.

Seq2seq(sequence to sequence) architectures are considered to be an important medium to MT(Machine Translation).This sequence comes under many to many sequence architecture of variable input and output length. Generally the architecture for MT consist of a encoder and a decoder. The encoder takes the embedding of the words present in the vocabulary of one language, encodes it and provides it to a decoder as a starting activation. The decoder looks similar to language model which is used to generate random sequence, But the difference lies in here, language model is a one to many architectures which generate random sequences and the activation with which the model is initialized is 0. Whereas the decoder works on the theory of conditional probability and can be called as a Conditional Language model. It generates a sequence given an input i.e. the output of the encoder.

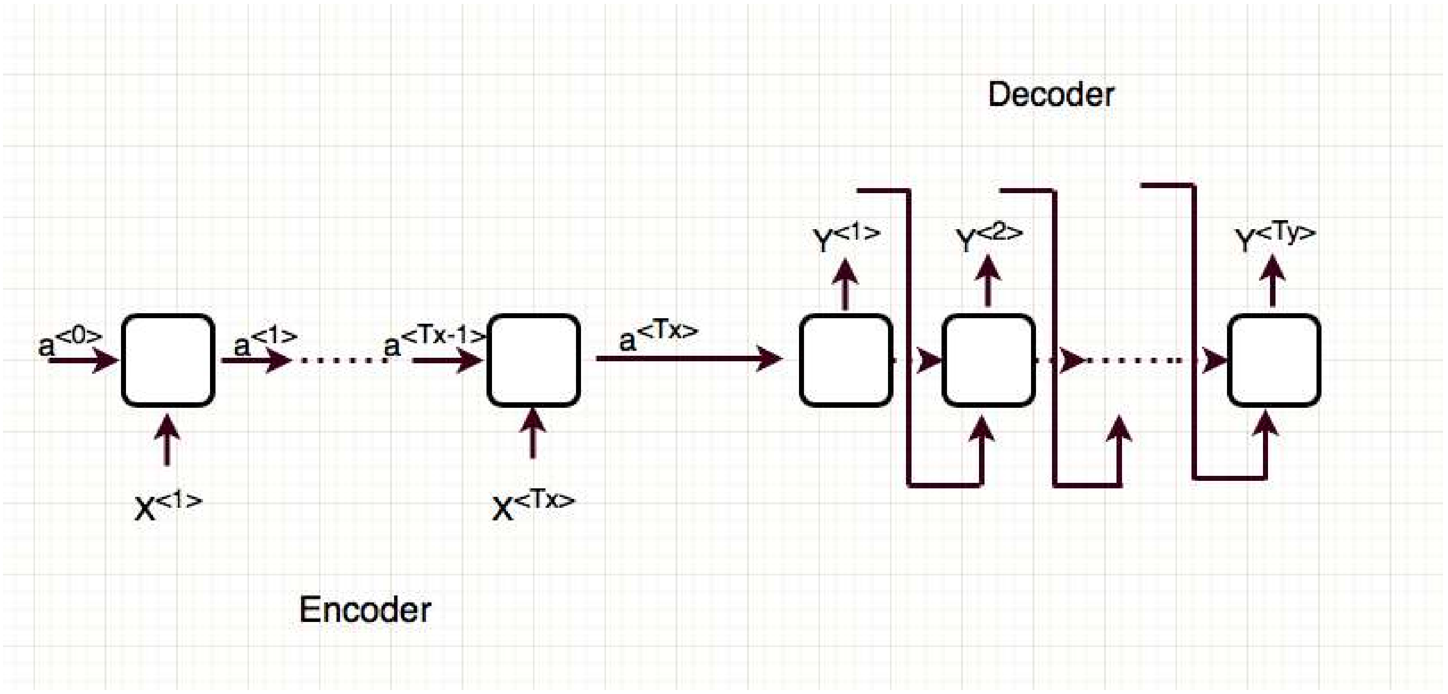


Figure: 4.8

Encoder and Decoder in Machine Translation
Given an input sequence x<1>, x<2>, x<3>,…., x<Tx> length Tx are in language-1 and the output generated by the decoder network y<1>, y<2>,…., y<Ty> be of length Ty in language-2. The output consist of a softmax layer. The outputs are given by the probability P(y<1>,y<2>,….., y<Ty>|a<Tx>). To pickup the most likely sentence this probability expression needs to be maximized i.e. :-
The goal is to find a sentence y such that:

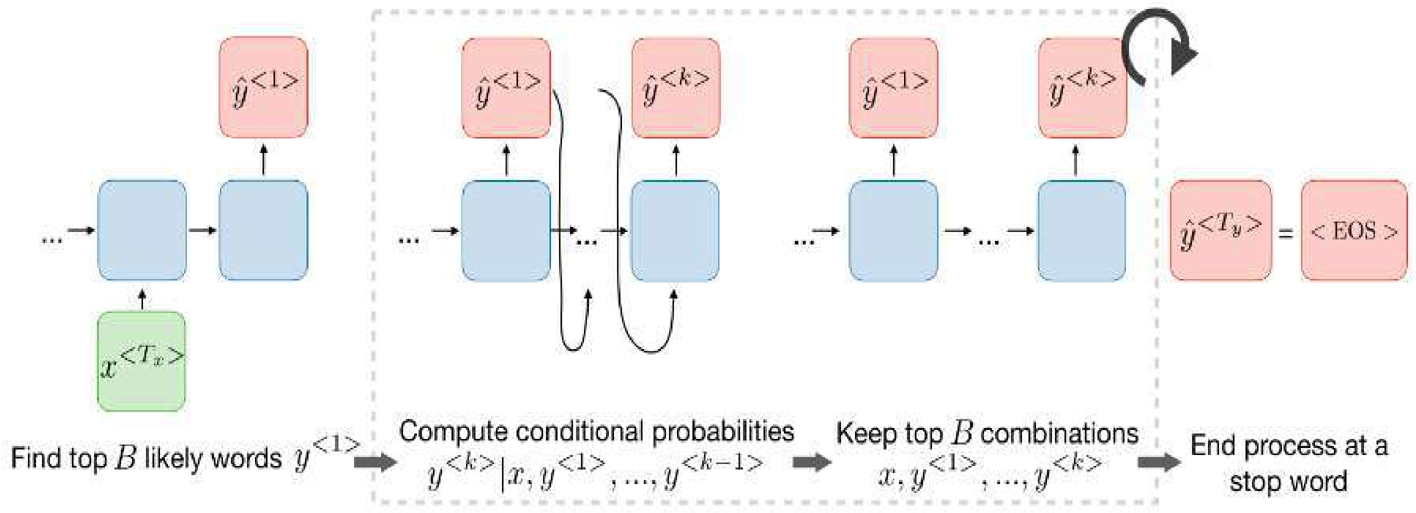
$$\boxed{y = \underset{y^{<1>},\ldots,y^{<T_y>}}{\arg\max}\ P(y^{<1>},\ldots,y^{<T_y>}|x)}$$

One popular heuristic method to execute the purpose is Beam Search. Another solution to the above is the use of Greedy Search. Greedy Search takes only one output into account that reduces the possibilities to get other sentences also which can be more likely output to the translation.

Beam search —

Beam search decoding iteratively creates text candidates (beams) and scores them.
It is a heuristic search algorithm used in machine translation and speech recognition to find the likeliest sentence y given an input x.

- Step 1: Find top B likely words $y^{<1>}$
- Step 2: Compute conditional probabilities $y^{<k>}|x, y^{<1>}, \dots, y^{<k-1>}$
- Step 3: Keep top B combinations $x, y^{<1>}, \dots, y^{<k>}$


$\hat{y}^{<1>}$
$x^{<T_x>}$
$\hat{y}^{<k>}$
$\hat{y}^{<T_y>}$ = < EOS >
Find top B likely words $y^{<1>}$
Compute conditional probabilities $y^{<k>}|x, y^{<1>}, \dots, y^{<k-1>}$
Keep top B combinations $x, y^{<1>}, \dots, y^{<k>}$
End process at a stop word

Remark: if the beam width is set to 1, then this is equivalent to a naive greedy search.

Figure: 4.9

Beam width

The value of beam width for production purpose is generally kept between 10–100 and for research purpose this value is usually taken in between 1000 to 3000. More the beam width, more is the possibility of finding a likely sentence but it makes the computational expenses and memory requirement significantly high.
The beam width B is a parameter for beam search. Large values of B yield to better result but with slower performance and increased memory. Small values of B lead to worse results but is less computationally intensive. A standard value for B is around 10.

Length normalization — In order to improve numerical stability, beam search is usually applied on the following normalized objective, often called the normalized log-likelihood objective, defined as:

$$\boxed{\text{Objective} = \frac{1}{T_y^{\alpha}} \sum_{t=1}^{T_y} \log \left[p(y^{<t>}|x, y^{<1>}, \dots, y^{<t-1>}) \right]}$$

Remark: the parameter αα can be seen as a softener, and its value is usually between 0.5 and 1.

Bleu score — The bilingual evaluation understudy (bleu) score quantifies how good a machine translation is by computing a similarity score based on n-gram precision. It is defined as follows:

$$\boxed{\text{bleu score} = \exp\left(\frac{1}{n}\sum_{k=1}^{n} p_k\right)}$$

where p_n is the bleu score on n-gram only defined as follows:

$$p_n = \frac{\displaystyle\sum_{\text{n-gram}\in\hat{y}} \text{count}_{\text{clip}}(\text{n-gram})}{\displaystyle\sum_{\text{n-gram}\in\hat{y}} \text{count}(\text{n-gram})}$$

Attention model

Attention was presented by Dzmitry Bahdanau, et al. in their paper "Neural Machine Translation by Jointly Learning to Align and Translate" that reads as a natural extension of their previous work on the Encoder-Decoder model.

Attention is proposed as a solution to the limitation of the Encoder-Decoder model encoding the input sequence to one fixed length vector from which to decode each output time step. This issue is believed to be more of a problem when decoding long sequences.

This model allows an RNN to pay attention to specific parts of the input that is considered as being important, which improves the performance of the resulting model in practice. By noting $\alpha^{<t,t'>}$ the amount of attention that the output $y^{<t>}$ should pay to the activation $a^{<t'>}$ and $c^{<t>}$ the context at time t, we have:

$$\boxed{c^{<t>} = \sum_{t'} \alpha^{<t,t'>} a^{<t'>}} \quad \text{with} \quad \sum_{t'} \alpha^{<t,t'>} = 1$$

Remark: the attention scores are commonly used in image captioning and machine translation.

Attention weight The amount of attention that the output $y^{<t>}$ should pay to the activation $a^{<t'>}$ is given by $\alpha^{<t,t'>}$ computed as follows:

$$\boxed{\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\displaystyle\sum_{t''=1}^{T_x} \exp(e^{<t,t''>})}}$$

Remark: computation complexity is quadratic with respect to Tx.

Reinforcement Learning

Reinforcement learning is a branch of machine learning that is concerned to take a sequence of actions in order to maximize some reward.

Basically an RL does not know anything about the environment, it learns what to do by exploring the environment. It uses actions, and receive states and rewards. The agent can only change your environment through actions.

One of the big difficulties of RL is that some actions take time to create a reward, and learning this dynamics can be challenging. Also the reward received by the environment is not related to the last action, but some action on the past.

Some concepts:

- Agents take actions in an environment and receive states and rewards
- Goal is to find a policy that maximize it's utility function
- Inspired by research on psychology and animal learning

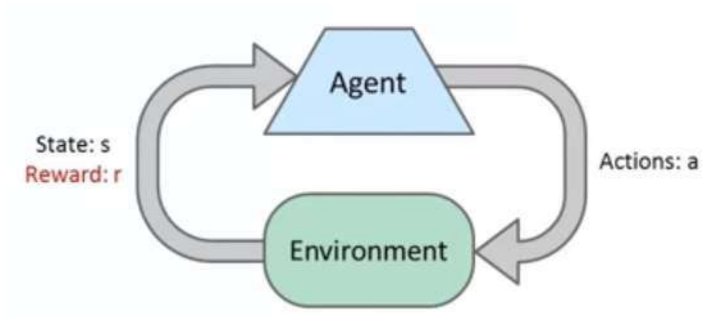


Figure: 4.10

Here we don't know which actions will produce rewards, also we don't know when an action will produce rewards, some times you do an action that will take time to produce rewards.

Basically all is learned with interactions with the environment.

Reinforcement learning components:

- Agent: Our robot
- Environment: The game, or where the agent lives.
- A set of states
- Policy: Map between state to actions
- Reward Function : Gives immediate reward for each state
- Value Function: Gives the total amount of reward the agent can expect from a particular state to all possible states from that state. With the value function you can find a policy.
- Model (Optional): Used to do planning, instead of simple trial-and-error approach common to Reinforcement learning. Here means the possible state after we do an action on the state

There is a variant of Reinforcement learning called Deep Reinforcement Learning where you use Neural Networks as function approximators for the following:

- Policy (Select next action when you are on some particular state)
- Value-Functions (Measure how good a state or state-action pair is right now)
- The whole Model/World dynamics, so you can predict next states and rewards.

MDP

MDP is a framework that can solve most Reinforcement Learning problems with discrete actions. With the Markov Decision Process, an agent can arrive at an optimal policy for maximum rewards over time.

The Markov decision process, better known as MDP, is an approach in reinforcement learning to take decisions in a grid world environment. A grid world environment consists of states in the form of grids.

The MDP tries to capture a world in the form of a grid by dividing it into states, actions, models/transition models, and rewards. The solution to an MDP is called a policy and the objective is to find the optimal policy for that MDP task.

Thus, any reinforcement learning task composed of a set of states, actions, and rewards that follows the Markov property would be considered an MDP.

The aim of MDP is to train an agent to find a policy that will return the maximum cumulative rewards from taking a series of actions in one or more states.

Here are the most important parts:

- States: A set of possible states
- Model: Probability to go to state when you do the action while you were on state , is also called transition model.

- Action: , things that you can do on a particular state
- Reward: , scalar value that you get for been on a state.
- Policy: , our goal, is a map that tells the optimal action for every state
- Optimal policy: , is a policy that maximize your expected reward

Bellman equations

The agent tries to get the most expected sum of rewards from every state it lands in. In order to achieve that we must try to get the optimal value function, i.e. the maximum sum of cumulative rewards. Bellman equation will help us to do so.

Using Bellman equation, the value function will be decomposed into two part; an immediate reward, ***R****t+1*, and discounted value of the successor state ***γV***(***S****t+1*),

$$v(s) = \mathbb{E}[G_t \mid S_t = s]$$

We unroll the return ***G****t,*

$$= \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \mid S_t = s\right]$$
$$= \mathbb{E}\left[R_{t+1} + \gamma\left(R_{t+2} + \gamma R_{t+3} + \ldots\right) \mid S_t = s\right]$$

then substitute the return ***G****t+1*, starting from time step *t+1,*

$$= \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s]$$

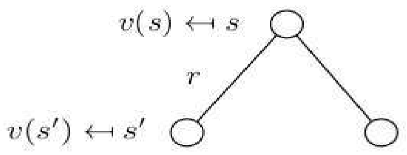
finally, since the expected value function is a linear function, meaning that $\mathbb{E}$(a*X*+*bY*)= a$\mathbb{E}$(*X*) +b$\mathbb{E}$(*Y*). The expected value of the return ***G****t+1* is the value of the state ***S****t+1,*

$$= \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s]$$

That gives us the Bellman equation for MRPs,

$$v(s) = \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s]$$

So, for each state in the state space, the Bellman equation gives us the value of that state,



$$v(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s')$$

The value of the state ***S*** is the reward we get upon leaving that state, plus a discounted average over next possible successor states, where the value of each possible successor state is multiplied by the probability that we land in it.

Value Iteration and Policy Iteration

The value-iteration and policy-iteration algorithms are two fundamental methods for solving MDPs. Both value-iteration and policy-iteration assume that the agent knows the MDP model of the world (i.e. the agent knows the state-transition and reward probability functions). Therefore, they can be used by the agent to (*offline*) plan its actions given knowledge about the environment before interacting with it.

Value iteration computes the optimal state value function by iteratively improving the estimate of **V(s)**. The algorithm initialize **V(s)** to arbitrary random values. It repeatedly updates the **Q(s, a)** and **V(s)** values until they converges. Value iteration is guaranteed to converge to the optimal values.

While value-iteration algorithm keeps improving the value function at each iteration until the value-function converges. Since the agent only cares about the finding the optimal policy, sometimes the optimal policy will converge before the value function. Therefore, another algorithm called policy-iteration instead of repeated improving the value-function estimate, it will re-define the policy at each step and compute the value according to this new policy until the policy converges. Policy iteration is also guaranteed to converge to the optimal policy and it often takes less iterations to converge than the value-iteration algorithm.

Value-Iteration vs Policy-Iteration

Both value-iteration and policy-iteration algorithms can be used for *offline planning* where the agent is assumed to have prior knowledge about the effects of its actions on the environment (they assume the MDP model is known). Comparing to each other, policy-iteration is computationally efficient as it often takes considerably fewer number of iterations to converge although each iteration is more computationally expensive.

Actor-critic model

1. The “Critic” estimates the value function. This could be the action-value (the *Q value*) or state-value (the *V value*).
2. The “Actor” updates the policy distribution in the direction suggested by the Critic (such as with policy gradients).

And both the Critic and Actor functions are parameterized with neural networks.

Actor-Critics aim to take advantage of all the good stuff from both value-based and policy-based while eliminating all their drawbacks. And how do they do this?

The principal idea is to split the model in two: one for computing an action based on a state and another one to produce the Q values of the action.

The actor takes as input the state and outputs the best action. It essentially controls how the agent behaves by learning the optimal policy (policy-based). The critic, on the other hand, evaluates the action by computing the value function (value based). Those two models participate in a game where they both get better in their own role as the time passes. The result is that the overall architecture will learn to play the game more efficiently than the two methods separately.

How Actor Critic works

Imagine you play a video game with a friend that provides you some feedback. You’re the Actor and your friend is the Critic.

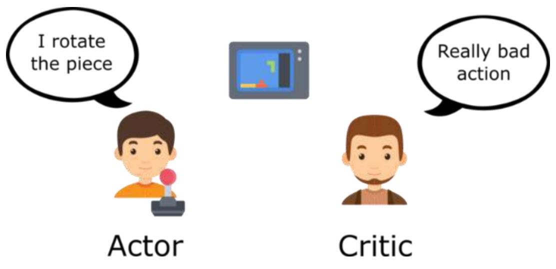


Figure: 4.11

At the beginning, you don’t know how to play, so you try some action randomly. The Critic observes your action and provides feedback.

Learning from this feedback, you'll update your policy and be better at playing that game.
On the other hand, your friend (Critic) will also update their own way to provide feedback so it can be better next time.
The idea of Actor Critic is to have two neural networks. We estimate both:

$\pi(s, a, \theta)$ ACTOR : A policy function, controls how our agent acts.	$\hat{q}(s, a, w)$ CRITIC : A value function, measures how good these actions are.

Both run in parallel. Because we have two models (Actor and Critic) that must be trained, it means that we have two set of weights that must be optimized separately.

Q-learning

In the case where the agent does not know apriori what are the effects of its actions on the environment (state transition and reward models are not known). The agent only knows what are the set of possible states and actions, and can observe the environment current state. In this case, the agent has to actively learn through the experience of interactions with the environment. There are two categories of learning algorithms:

model-based learning: In model-based learning, the agent will interact to the environment and from the history of its interactions, the agent will try to approximate the environment state transition and reward models. Afterwards, given the models it learnt, the agent can use value-iteration or policy-iteration to find an optimal policy.

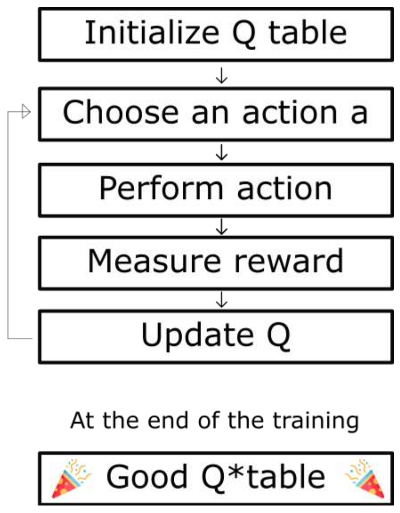
model-free learning: in model-free learning, the agent will not try to learn explicit models of the environment state transition and reward functions. However, it directly derives an optimal policy from the interactions with the environment.

Q-Learning is an example of model-free learning algorithm. It does not assume that agent knows anything about the state-transition and reward models. However, the agent will discover what are the good and bad actions by trial and error.

The basic idea of Q-Learning is to approximate the state-action pairs Q-function from the samples of Q(s, a) that we observe during interaction with the environment. This approach is known as Time-Difference Learning.

Q-learning is an off policy reinforcement learning algorithm that seeks to find the best action to take given the current state. It's considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn't needed. More specifically, q-learning seeks to learn a policy that maximizes the total reward. The 'q' in q-learning stands for quality. Quality in this case represents how useful a given action is in gaining some future reward.

The Q-learning algorithm Process



SARSA

The SARSA stands for **State Action Reward State Action** which symbolizes the tuple (s, a, r, s', a') is an On-Policy algorithm for TD-Learning. The major difference between it and Q-Learning, is that the maximum reward for the next state is not necessarily used for updating the Q-values. Instead, a new action, and therefore reward, is selected using the same policy that determined the original action. The name Sarsa actually comes from the fact that the updates are done using the quintuple **Q(s, a, r, s', a').** Where: **s, a** are the original state and action, **r** is the reward observed in the following state and **s', a'** are the new state-action pair.

SARSA vs Q-learning

The difference between these two algorithms is that **SARSA** chooses an action following the same current policy and updates its Q-values whereas **Q-learning** chooses the greedy action, that is, the action that gives the maximum Q-value for the state, that is, it follows an optimal policy.