Weekly report (July 9, 2018 - July 13, 2018)

Wyk
182507635559@163.com

Abstract

This week, I implemented DQN to solve a simple CartPole game. Finished 'Sequence models' first week courses on coursera, and completed the corresponding assignment.

1 Implement DQN

After learning the concepts, I did some further searching and finally finish a simplified DQN algorithm. Following are the ideas and details.

B.T.W: I won't mention much about the coding part here because of the inconvenience, the detailedly annotated code is at my github: `https://github.com/W-yk/Deep_learning/blob/master/DQN/Solving_CartPole_with_DQN.py`. Forgive me if the code seems clumsy for my coding experience is quite little, and it's probably better to adopt an object-oriented way instead of mine because we are coding an agent.

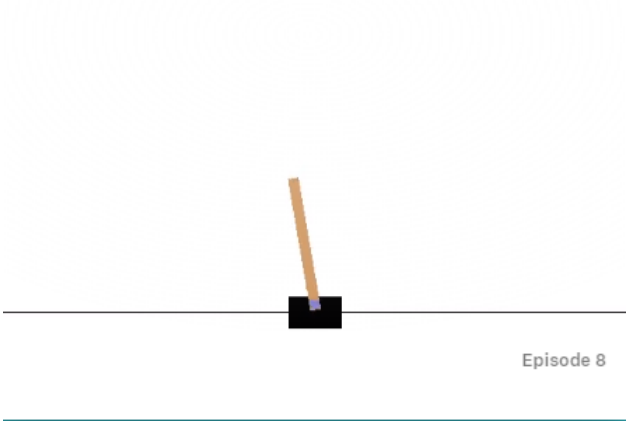
1.1 Experience Replay

Last week I covered some basic concepts in reinforcement learning. For a recap, the most important concepts are the 'MDP' which is our agent's environment, 'Q-funtion' used to measure benefit of an action and 'Bellman equation' for computation. For DQN, I mentioned the 'Value Function Approximation' method. And to implement there is another important part in the original paper which is called 'Experience Replay'.

In a supervised Deep learning network, the input dataset must have a fixed distribution and each data must be independent. So in the case of DQN, we can't just train the model with every new state after an action. To address this issue, 'Experience Replay' is adopted. The idea is to store the states and randomly sample the training batch from them. This way we break the correlation of data and solve the non-stationary distribution problem.

1.2 CartPole environment

This game is from the OpenAI gym library. It's a collection of test problems environments that can be used to work out your reinforcement learning algorithms.



For the game CartPole, a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

1.3 Implementation detail

The simplified algorithm is as follows:

1. Initialize the environment.
2. Build the deep network.
3. For each episode:
   Restart the game by reset the environment to the beginning.
   Do if a game haven't end:
   (a) Take an action randomly or according to the network output.
   (b) Update the state and add it to the memory.
   (c) Sample batch from the memory if enough and train the model.

Here are some details that worth mentioning:

- Use deque to store the memory. Deque is a double-end queue, so if the memory run out of space, it will delete the first added state at the beginning. Which gives our agent the ability to forget old memories, just like we humans.
- I built model with only dense layers because the input state is quite simple in this game, which turns out works fine.
- The network's output is for all the actions, but we can only get one action value at a time. So the label must be built base on the previous output and only change the value for the action we take. I actually made a mistake here at first by assigning the label based on an empty numpy array and it takes me quite some time to figure out this bug.
- The agent should be more dependent on our model after training for some time. Which means we should decreasing the probability of choosing random actions gradually.
- Use the number of actions taken in a game as the score. The more actions taken while staying balanced the higher the score is.

1.4 Result

A window should pop up after running the code, it will show the process of our agent playing the game. It takes some time to train, but it's quite fun watching the agent try to stay balance then fail and then start again. It's not possible to add a video in this report, or else I would be happy to show you the process.
After training for around 400 episodes, the agent can already reach the maximum score I set.

```
episode: 86/100, score: 499, epsilon: 0.01
episode: 87/100, score: 499, epsilon: 0.01
episode: 88/100, score: 499, epsilon: 0.01
episode: 89/100, score: 320, epsilon: 0.01
episode: 90/100, score: 499, epsilon: 0.01
episode: 91/100, score: 499, epsilon: 0.01
episode: 92/100, score: 499, epsilon: 0.01
episode: 93/100, score: 499, epsilon: 0.01
```

Which suggests the implementation succeed.

2 Learning recurrent neural network

I began studying the 'Sequence Model' taught by Andrew Ng on coursea. and following are my understandings.

2.1 what are sequence models and why use RNN ?

For the common FC neural network as well as convolutional neural network, we usually have the inputs and outputs as discrete data. But for some problem like speech recognition, music generation, machine translation and so on, there are a sequence of data either in input or output. And the model to solve this kind of problem is called 'Sequence Models'.

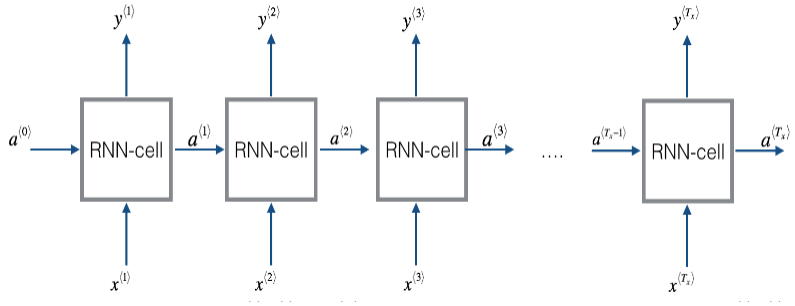
It's natural to try to use a standard network, and here are two main reason it won't work well:

1. Inputs, outputs can be different lengths in different examples. But for a standard network the input and out shape is fixed. You don't want your translator to be only able to translate sentences with a fixed size.
2. Doesn't share features learned across different positions of text. Which means to accomplish a task you need an enormous amount of parameters.

But a Recurrent neural network has neither of these problems because of its unique structure, which makes it excel at solving sequence problems.

2.2 Basic RNN structure

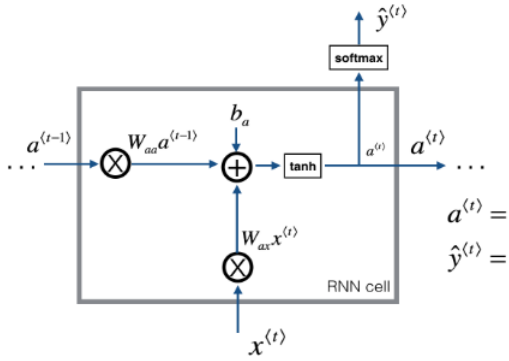
Recurrent has an overall structure like this:



At each time step, the network takes in data both from the input sequence and also the activation form the previous time step. Which allows the neuron to work with a sequence of data. If this is still

confusing, I think it can be intuitively understood as a standard deep network but instead of having only one input layer(output layer), we make each layer in the network into an input layer (output layer) by divide the input sequence into separate parts and feed them accordingly.

Add the computational detail is shown below:



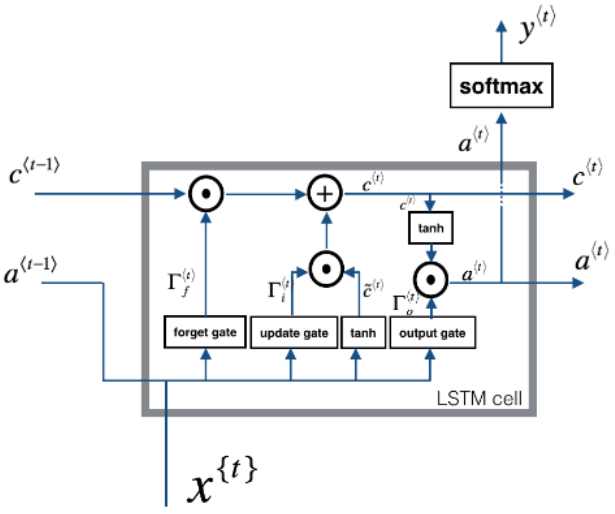
$$a^{\langle t \rangle} = \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b_a)$$
$$\hat{y}^{\langle t \rangle} = soft\max(W_{ya}a^{\langle t \rangle} + b_y)$$

2.3 Variant types of RNN

The basic RNN structure shown previously still have some drawbacks, and researchers have developed various versions to improve it. And there are two main directions for this:

2.3.1 Improve the cell

Vanishing gradients is a tricky problem for deep convolutional networks, and it is also the case for recurrent neural networks . A sequence problem can have a longterm dependence of data, for example the sentence "The cat, which already ate , was full. In English grammar, it's supposed to be 'was' because the 'cat' at the beginning is single. But it's hard for a standard RNN model to learn such a long span feature because of vanishing gradients. And the LSTM cell were design to solve this.

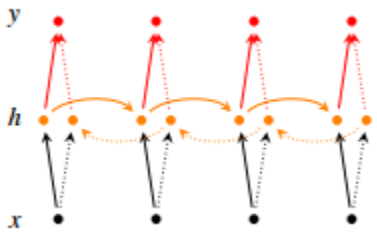


$$\Gamma_f^{\langle t \rangle} = \sigma(W_f[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_f)$$
$$\Gamma_u^{\langle t \rangle} = \sigma(W_u[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u)$$
$$\tilde{c}^{\{t\}} = \tanh(W_C[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_C)$$
$$c^{\langle t \rangle} = \Gamma_f^{\langle t \rangle} \circ c^{\langle t-1 \rangle} + \Gamma_u^{\langle t \rangle} \circ \tilde{c}^{\langle t \rangle}$$
$$\Gamma_o^{\langle t \rangle} = \sigma(W_o[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_o)$$
$$a^{\langle t \rangle} = \Gamma_o^{\langle t \rangle} \circ \tanh(c^{\langle t \rangle})$$

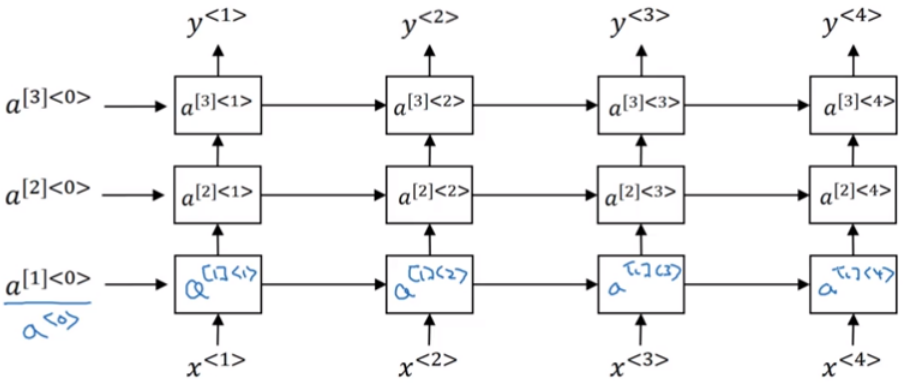
The most important idea of LSTM is adding something called cell state. It passed across cell at each timestep which is the upper line shown in the graph. With only a little amount of liner computation, data can remain the same after a long time. And to decide what kind of data to pass, there are control gates added to it. This way the network can learn to remember as well as forget data even form long distance.

2.3.2 Improve the overall structure

Aside from changing the RNN cell, the model can also be improved by changing how cells connent.



This is called 'Bi-directional RNN', which enables the RNN cell to look into the future data and then output. It is commonly used in solving nature language processing problems.
To improve the network's ability to learn even more complex features, there is also a deep version of RNN, It's built by stacking basic RNN vertically:



Because the recurrent network already have a big horizontal depth, it's hard to train one with also too much vertical depth. In practice, a three-layer-RNN is big enough .

3 Class assignments for sequence models

There are three assignments for the first week of 'sequence models' class:

3.1 Building a Recurrent Neural Network- Step by Step

The first assignment teaches how to use numpy to build a recurrent neural network from scratch. The assignment is pretty straight forward with detailed instructions, I didn't have much problems finishing it and it gives me a better understanding of the RNN structure during the process.

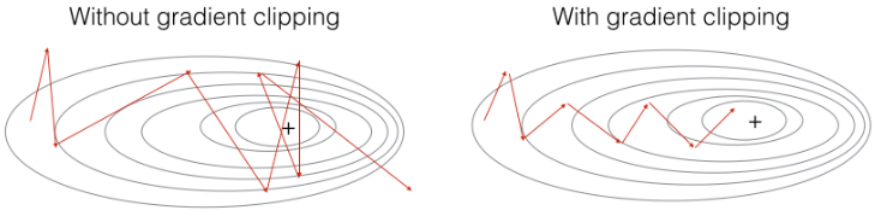
3.2 Character level language model - Dinosaurs land

The idea of this assignment is to generate dinosaurs names with RNN model. And I think the most important thing to remember are shown below:

3.2.1 Gradient clipping

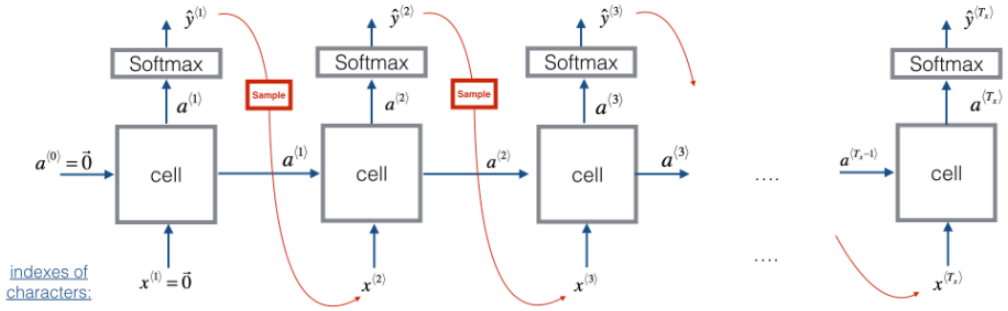
In deep-learning, the model can sometimes output a bunch of NaNs, and this is the result of exploding gradients(gradients with overly large values). For RNN models, this happens much more often. Because of the unique structure, the process of backward propagation have a long time dependence, which in this case is a continued product of weight matrices, and this could easily lead to infinite numbers.
To prevent this from happening we can use a method called 'Gradient Clipping'. Namely it means clip every element of the gradient vector to lie between some range [-N, N], if any component of

the gradient vector is greater than N, it would be set to N. if any component of the gradient vector is less than -N, it would be set to -N. If it is between -N and N, it is left alone. This way the model can be more robust during the training process.



3.2.2 Sampling a sequence from a trained RNN

After training a model with some sequence data, if we want it to output something similar but new, what we gonna do is called 'Sampling'.



It has the following steps:

1. **Step1**: Feed the network's first layer with zero vectors or some chosen vectors.
2. **Step2**: Sample form the output. Because the network have a softmax output layer, what we get is a probability vector. And sample means randomly choose one according to this distribution. Adding it to the output sequence.
3. **Step3**: Take step2's corresponding one hot vector and feed it into the next layer.
4. **Step4**: Loop over Step3 and Step4 until getting a EOS output.

And using this we can generate not only dinosaurs names, but also poems as well as musics.

3.2.3 model performance

The dinosaurs name from the input dataset is like this:

```
Anasazisaurus
Anatosaurus
Anatotitan
Anchiceratops
Anchiornis
Anchisaurus
Andesaurus
Andhrasaurus
```

Following the instructions, the RNN model output below names:

```
Iteration: 34000, Loss: 22.418502

Mixtrolomophus
Inedalosaurus
Ixrythoinokus
Macalosaurus
Yussandosaurus
Edakrophantitan
Utmarisaurus
```

As you can see, the endings of these words are just like those the input dataset, which shows the RNN's ability to learn sequence features.

3.3 Improvise a Jazz Solo with an LSTM Network

This assignment is about implementing a model that uses LSTM to generate music. It uses Keras this time instead of the basic numpy version. But because the build in function to get the model's output in keras require a complete input data, unlike what we do in sampling. So to fix that, in the assignment two models were built, one is for the RNN model we train, and the other is for sampling to get the output. I finish the task with the detailed instructions it provides, and generate some pretty decent musics which I think are awesome. But the detail of building a RNN model with Keras still confuse me, more practice needed, I guess.

4 Summary

Another enriching week . It took me some efforts to implement the DQN model, but it's a real pleasure to finally watch the agent you code playing the game. And I also learnt to generate names as well as music pieces with RNN. But I think my understanding of RNN still stays more on the concepts, and I hope I can find some time implement some code to get a better sense.

Questions

1. In the simple RNN model and LSTM model, the tanh nonlinearity is used, I wonder why not use the popular RELU function? Is this just because of history issues or the RELU just doesn't help too much in RNN case ?

Plans for next week

1. Learn more variant of DQN models.
2. Learn the sequence model class on coursera and finish the scened week's assignment.
3. Check out the deeplearning model pytorch.