Deep Learning Study Lecture Note

Lecture 7 : Other networks, Other methods

Kim Tae Young

February 12, 2020

7.1 Introduction

In this chapter, we will treat other networks that we didn't treat before. These are not generally used as MLP, CNN, RNN, however performs better then these networks in specific topics. Also, we will treat other types of problem, instead of regression and classification.

7.2 Graph Convolutional Neural Network

A Graph Convolutional Neural Network is variation of CNN, where instead of n-dimensional tensor input, input is in form of graph, $G = (V, E)$.
More specific, the input is in form

- Feature description x_v for each vertex in V. This input is given as $N \times D$ matrix, where N is number of vertex, D is number of input feature.
- Representative description of graph structure : Usually as adjacency matrix A.

and output is output feature description, as $N \times F$ matrix. Here, F is number of output feature.
Then one single layer is

$$H^{(l+1)} = f(H^{(l)}, A)$$

and specification of network is given by f.
The simplest layer is given as

$$f(H^{(l)}, A) = \sigma(AH^{(l)}W^{(l)})$$

where $W^{(l)}$ is weight matrix need to learn. Since usual adjacency matrix's diagonal terms are all zero unless graph has self loop, so we add identity matrix to adjacency matrix, making we have connection from each node to itself.

$$f'(H^{(l)}, A) = \sigma((A + I)H^{(l)}W^{(l)})$$

Secondly, it is usual that adjacency matrix is not normalized, so multiplication of A will yield change scale of feature vector. So we normalize A by multiplying inverse of degree matrix,

$$\hat{A} = A + I$$

$$\hat{D}_{i,j} = \begin{cases} 0 & i \neq j \\ deg(i) + 1 & i = j \end{cases}$$

$$f(H^{(l)}, A) = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)})$$

The main problems treated by GCNN is first, society network. Since connection between individual can be viewed as graph, we can apply GCNN over it. Also other example is molecules. By encoding each atom to input feature and molecular structure to graph structure, we can apply GCNN.

7.3 Neural Turing Machine

Neural Turing Machine is modeling of real computer as neural network. Neural Turing Machine satisfies Von Neumann architecture consisting memory, input output device, and processor.
The structure of Neural Turing Machine is composed by two parts, controller and memory and input output head that connects controller and memory. The controller is where neural network placed, and memory is implemented by some large matrix, with size $M \times N$, meaning M locations of N sized vector.

$$\begin{tikzcd}
 & ReadHead \arrow[dl] & \\
Controller \arrow[dr, "erase", "add"'] & & \begin{pmatrix} M^{(t)}_{(1,1)} & \cdots & M^{(t)}_{(1,N)} \\ \vdots & & \vdots \\ M^{(t)}_{(M,1)} & \cdots & M^{(t)}_{(M,N)} \end{pmatrix} \arrow[ul, "wM^{(t)}"'] \\
 & WriteHead \arrow[ur, "erase", "add"'] &
\end{tikzcd}$$

Figure 7.1: Neural Turing Machine

Then the controller gets two input, internal input and external input. The external input is input to neural turing machine itself which is given as sequence, and internal input is read value from memory. However unlike classical computer where we read single input from memory, neural turing machine reads whole value, as a weighted sum.
So first, controller creates weight vector w, which has size M. Then the internal input is computed as weighted sum of memory matrix. Note that this weighted sum is matrix multiplication.
Creating weight is splitted to two parts, content addressing and location addressing.
In content addressing, first we compare output of controller at time t, key vector k_t and each column value of memory matrix, $M_t(i)$. Then the similarity of two vector is adjusted using key strength, β_t which is also output of controller. Then computed weight is in following form where $K[-,-]$ is cosine similarity.

$$w_t^c(i) = \frac{exp(\beta_t K[k_t, M_t(i)])}{\sum_j exp(\beta_t K[k_t, M_t(j)])}$$

Then we interpolate weight with previous weight, as following. Again interpolation weight g_t is output of controller.

$$w_t^g = g_t w_t^c + (1 - g_t) w_{t-1}$$

Now we revise w_t^g, computed in content addressing based on its location. First we perform rotational shifting, with shift weighting s_t.

$$\tilde{w}_t(i) = \sum_{j=0}^{N-1} w_t^g(j) s_t(i-j)$$

Finally, we apply sharpening, resulting weights vector $w_t(i)$.

$$w_t(i) = \frac{\tilde{w}_t(i)\gamma_t}{\sum_j \tilde{w}_t(j)\gamma_t}$$

Then we compose internal input and external input, we produce 3 outputs, external output, erase output e, add output a.
Now we write to memory, with two output erase output and add output. First, erase output is size M vector. We take inner product of erase output and weight vector, and subtract that value from 1, and multiply subtracted value to memory matrix.

$$\hat{M}^{(t)} = M^{(t)}(1 - w^T e)$$

Second, add output is size N vector. We multiplicate add output and weight vector to create size $M \times N$ matrix, and add it to memory matrix.

$$M^{(t+1)} = \hat{M}^{(t)} + wa^T$$

The main use of NTM is sequence two sequence translation, like RNN, however it shows better results on problems like sequence copying, priority sorting.

7.4 Modular Neural Network

Modular Neural Network is one type of neural network. Unlike MLP, CNN, RNN, modular neural network contains multiple networks.

$$\begin{tikzcd}
 & NN_1 \arrow[dr] & & \\
Input \arrow[ur] \arrow[dr] & \vdots & NN_g \arrow[r] & Output \\
 & NN_n \arrow[ur] & &
\end{tikzcd}$$

Figure 7.2: Modular Neural Network

Here, each NN_i and NN_g could be any of MLP, CNN, RNN, or even other networks. Then after forwarding input to each of output, gate network NN_g finally computes output given outputs of NN_i.

7.5 Deep Reinforcement Learning

As a reminder, reinforcement learning is finding policy $P(Action|State)$ that maximizes rewards. In deep reinforcement learning, we use deep neural network as a model for reinforcement learning, getting help of its universal approximation.

$$\begin{tikzcd} Agent \arrow[d, bend left, "action"] \\ Enviroment \arrow[u, bend left, "state"] \arrow[u, bend left=80, "reward"] \end{tikzcd}$$

Figure 7.3: Agent-Enviroment Interaction

First, in reinforcement learning, we consider our Agent-Environment Interaction as markov decision process. I will not go deep into the markov chain here.

Definition 7.5.1. *Markov Decision Process is composed with 4 elements, discrete state space and action space, state transition probability, reward function.*

- *Discrete state space is finite set of states, embedding environment.*
- *Action space is set of actions that agent can perform.*
- *State transition probability is probability given as $P(S'|S, A)$ meaning that given current state and chosen action, describes probability of next state.*
- *Reward function is function on state space, explaining reward.*

The purpose of Markov Decision Process is to find an optimal policy $P(A|S)$, that maximizes expected sum of rewards. When computing sums, if our task is infinitely many, sum of total reward may diverge. So instead we compute discounted future reward, where γ is discount factor.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

Assuming that we know environment dynamics $P(S', r|S, A)$, we can do exhaustive search until end step, however it is highly time consuming. However by applying dynamic programming, as dividing total problem as subproblem for two steps each, we can reduce computational cost by $O(N^2T)$. This is possible solution, however it still requires to know environment dynamics.
So instead, we use Monte Carlo method, or Temporal Difference Learning. In deep reinforcement learning, we use Q-learning, called DQN(Deep Q-Learning).
First, Q-learning is algorithm that is finding optimal strategy from finite markov decision process. Here we find function $Q : S \times A \to \mathbb{R}$ that calculates quality

of state-action pair by iterated assign.

```
**Data:** Learning rate $\alpha$, reward $r_t$, discount factor $\gamma$, end time $T$
init $Q$;
for $t \leftarrow 1$ to $T$ do
    $Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \max_{a \in A} Q(s_{t+1}, a))$;
end
```

In DQN, we create loss from the fact that every Q-function satisfies Bellman equation where $\pi(s)$ is chosen action by function π and s' is next state from $\pi(s)$.

$$Q^{\pi}(s,a) = r + \gamma Q^{\pi}(s', \pi(s))$$

With Bellman equation, we get temporal difference loss

$$\delta = Q(s,a) - (r + \gamma \max_a Q(s',a))$$

$$L = \frac{1}{|B|} \sum_{(s,a,s',r) \in B} \delta^2$$

where B is batch storing replay from simulation.

7.6 Deep Metric Learning

Metric Learning is type of problem that focus on measuring similarity of two data, like what cosine similarity function does. Since data like image, song, voice, similarity is nontrivial. Therefore adding neural network creates semantical similarity.
Our problem is to find two function f, D where E is embedding space, f is embedding from data space to embedding space, D is metric over embedding space. In general, we simply use L^2 norm for D, and only train for f.
We have two ways of metric learning, pair learning and triplet learning.
First, pair learning is also called Siamese Network. Our purpose is to find embedding f, satisfying

$$L^2(f(x_i), f(x_j)) > m$$

for data from different class x_i, x_j and margin m. Assuming f is some neural network, loss is defined as following.

$$L(y_{ij}, x_i, x_j) = \frac{y_{ij}}{2} L^2(f(x_i), f(x_j)) + \frac{1-y_{ij}}{2} \max(0, m - L^2(f(x_i), f(x_j)))$$

Here, y_ij is label denoting whether x_i and x_j are in same class.
Second, in triplet learning instead, we use three data, x_i, x_p, x_n where x_i is anchor and x_p is positive sample, having same class with x_i, x_n is negative sample having different class with x_i. Here we will find f such that

$$L^2(f(x_i), f(x_p)) + \delta < L^2(f(x_i), f(x_n))$$

Then we have following loss,

$$L(x_i, x_p, x_n) = \max(0, L^2(f(x_i) - f(x_p)) - L^2(f(x_i) - f(x)n)) + \delta)$$

7.7 Generative Adversarial Network

GAN(Generative Adversarial Network) is generative model, where the models we learned are discriminative models. In MLP or CNN, we are given high dimensional data and predicted their classification, however in GAN we are given low dimensional explanation of data and generate sample from learned distribution.
First, as ‘adversarial’ states, we have two sub models that compete each other. One model is called generator model, which generates new examples, that will be resulting model and discriminator model, which discriminate fake example from generator model and real data.
Now in generator model we assume fixed length latent random variable. From latent variable, generator gets random vector and generates a sample. You can think as this random vector as low dimensional data, and generator model is lifting this to high dimensional data, like image.
Then discriminator model is given two classes of input, real data and generated data form generator model, and predict binary class label whether data is real or generated. Here, discriminator model is classification model we used.
When training the model, generator and discriminator are trained together. Our purpose is letting generator minimizes discriminator's accuracy and discriminator maximizes its accuracy. Then discriminator's optimizer should minimize following loss

$$V(D) = \mathbb{E}_{x\sim p_{data}(x)}[\log(D(x))] + \mathbb{E}_{z\sim p_z(z)}[1 - \log D(G(z))]$$

while generator's optimizer should maximize following loss.

$$V(G) = \mathbb{E}_{z\sim p_z(z)}[\log D(G(z))]$$

Bibliography

[1] GRAPH CONVOLUTIONAL NETWORKS, THOMAS KIPF. https://tkipf.github.io/graph-convolutional-networks/

[2] Hands On Memory-Augmented Neural Networks Build, Xavier L. https://towardsdatascience.com/hands-on-memory-augmented-neural-networks-implementation-part-one-a6a4a88beba3

[3] Spiking Neural Network, Shikhargupta@github https://github.com/Shikhargupta/Spiking-Neural-Network

[4] Introduction of Deep Reinforcement Learning https://www.slideshare.net/NaverEngineering/introduction-of-deep-reinforcement-learning

[5] Metric Learning https://www.wewinserv.tistory.com/91