Neural Turing Machines

1 Citation

Graves, Alex, Greg Wayne, and Ivo Danihelka. "Neural turing machines." arXiv preprint arXiv:1410.5401 (2014).

`https://arxiv.org/pdf/1410.5401.pdf`

2 Abstract

We design an external memory (a matrix) and an attention mechanism, which we combine with other parts of a neural network to get a differentiable Turing machine that can learn simple algorithms like copying and sorting.

3 Introduction

A program requires elementary operations, logical operations, and external memory. We know Recurrent Neural Networks (RNN) are Turing-complete, but researchers don't yet know how to make a network learn an arbitrary program. Our Neural Turing Machine is a differentiable computer that learns simple programs. We give it an external memory (a large matrix) and it can learn how to use that memory with its attention mechanism.

4 Foundational Research

The human brain has working memory that helps it achieve tasks. In the 1980s, people thought that symbol processing systems would be the best way to model intelligence because they thought neural networks cannot learn how to store variables in memory or process variable length inputs. There is a debate whether recursive processing is uniquely human and specialized towards language.

RNNs are able to maintain hidden state, but they are hard to train because they have vanishing and exploding gradients. Using a Long Short-Term Memory (LSTM) and gradient clipping can mitigate both these problems. LSTMs have been applied to speech recognition, text generation, translation, and more.

5 Neural Turing Machines

An NTM consists of a neural memory controller and memory bank. The controller receives inputs from the outside world, manipulates memory with a read head and write head, and produces output. To identify which memory locations to write or read, we use an attention mechanism.

Let the memory at time t be $\mathbf{M}_t \in \mathbb{R}^{N \times M}$ where N is the number of memory cells and M is the vector size of each cell. Our attention mechanism produces a set of weights $\mathbf{w}_t$ where $\sum_i w_t(i) = 1$ and $0 \leq w_t(i) \leq 1$. The read operation is then just a weighted sum of the memory cells:

$$\mathbf{r}_t = \sum_i w_t(i)\mathbf{M}_t(i)$$

We decompose writing into an erase and add. That is, given an erase vector $\mathbf{e}_t$ (each element lies in $(0, 1)$) and add vector $\mathbf{a}_t$, our write operation is:

$$\mathbf{M}_t(i) = \mathbf{M}_{t-1}(i)[\mathbf{1} - w_t(i)\mathbf{e}_t] + w_t(i)\mathbf{a}_t$$

Now we need to discuss how the weights $\mathbf{w}_t$ are produced. We use two approaches - content based addressing and location based addressing. The former takes the controller input (i.e. input vector into the neural net) and figures out which memory cell it is most similar to. The latter uses primitives to rotate through memory cells.

Let's consider content based addressing first. We produce a vector $\mathbf{k}_t$ called the key vector. We then our content based weighting:

$$w_t^c(i) = \frac{\exp\left(\beta_t K[\mathbf{k}_t, \mathbf{M}_t(i)]\right)}{\sum_j \left(\exp\left(\beta_t K[\mathbf{k}_t, \mathbf{M}_t(j)]\right)\right)}$$

where the $K[\mathbf{u}, \mathbf{v}] = \frac{\mathbf{u} \cdot \mathbf{v}}{||\mathbf{u}|| \cdot ||\mathbf{v}||}$ (cosine similarity).

Location based addressing is designed to support iteration over memory cells and random jumps. First, we produce an interpolation gate $g_t \in (0, 1)$ that blends the content weighting with the previous weighting:

$$\mathbf{w}_t^g = g_t \mathbf{w}_t^c + (1 - g_t)\mathbf{w}_{t-1}$$

We produce a shift distribution $\mathbf{s}_t$, which is a probability distribution over the allowed shifts (i.e. -1, 0, or 1). You can also view it as a lower bound of a width-1 uniform distribution over the shift (e.g. 6.7 would have 0.7 mass on 7 and 0.3 mass on 6). To apply the shift, we do the following (arithmetic is modulo N):

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

if the shift distribution does not have a lot of probability mass on one shift, the resulting weighting can be blurry. To sharpen it, we produce a scalar γ_t and do:

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

Our memory controller can be an LSTM, which is nice because it has a hidden state (they act like registers) or a feedforward net, which is easier to interpret.

6 Experiments

After each input sequence, we reset the NTM's hidden state to a learned bias vector and erased the memory.

First, we trained the network to copy a sequence of input vectors. We feed it a sequence of one to twenty 8-bit binary vectors and a delimiter. Then we fed the sequence as the targets to the network to see if it produced the right outputs. The NTM with LSTM controller totally beats a plain old LSTM - it learns with much fewer examples and scales to longer input sequences.

Next, we trained the network to copy and input sequence k times, where the k is specified in a separate channel of the input vector at the end of the input sequence. NTM with LSTM controller learns faster and scales to longer sequences better than plain old LSTM. It's not perfect though, sometimes repeating the input more or less times than is necessary.

Next, we trained the network to read an input sequence and then a sequence of queries (i.e. index into input sequence) and return the sequence vectors corresponding to the queries. Again, NTM with LSTM controller dominates.

Next, we trained the network to create an N-gram table for an input sequence. NTM with LSTM controller is slightly better than LSTM, but neither solves the problem.

Next, we trained the network to read input sequences (each with a priority) and sort them according to priority. The NTM with LSTM is best again. We needed eight parallel read and write heads to get this performance.

During training, we used RMSprop and clipped gradients.

7 Conclusion

The Neural Turing Machine uses an attention mechanism and memory matrix to learn simple programs better than an LSTM can.