Using Fast Weights to Attend to the Recent Past

Wang CHEN
Using Fast Weights to Attend to the Recent Past

Jimmy Ba, Geoffrey Hinton, Volodymyr Mnih, Joel Leibo, Catalin Ionescu
University of Toronto
Google

Geoffrey Hinton
Vlad Minh
Joel Leibo
Catalin Ionescu
The Basic Idea

In recurrent neural networks (RNNs):

- The usual weights: They encode knowledge over the entire training dataset through gradient descent algorithm

- Slow varying, storing long term information about input and output mapping
The Basic Idea

In recurrent neural networks (RNNs):

- Hidden states or activity vectors: They act as a limited working memory storing the current sequence information
- Change at every timestep
The Basic Idea

How about short-term memory?

- Where do we store the temporary information?
- The hidden states have to remember the history of the current sequence but they also have to integrate appropriate memory content for the final classifier.
The Basic Idea

- **The slow weight**: slow varying, storing long term information
- **The fast weight**: rapid learning but also decaying rapidly, storing sequence specific temporary information
The Basic Idea

- **The slow weight**: slow varying, storing long term information

- **The fast weight**: rapid learning but also decaying rapidly, storing sequence specific temporary information
  - Hinton and Plaut, 1987, *Using fast weights to deblur old memories*
  - Schmidhuber, 1993, *Reducing the ratio between learning complexity and number of time varying variables in fully recurrent nets*
Ordinary RNN

\[ h_0(t + 1) = f(W h_0(t) + C x(t)) \]
Fast weights RNN

\[ h(t + 1) = f\left(\left[Wh(t) + Cx(t)\right] + A(t)h_0(t + 1)\right)\]
Fast weights RNN

\[ h_{s+1}(t + 1) = f([Wh(t) + Cx(t)] + A(t)h_s(t + 1)) \]
Hopfield Network and Associative Learning

- Fast weights update rule:

\[ A(t) = \lambda A(t - 1) + \eta h(t)h(t)^T \]

- Embedding a Hopfield network inside an RNN
Computation Efficiency

- Revisit the fast weights update rule:
  \[ A(t) = \lambda A(t - 1) + \eta h(t)h(t)^T \]
- Summation of rank-one matrices
  \[ A(0) = 0 \]
  \[ A(t) = \eta \sum_{\tau=1}^{\tau=t} \lambda^{t-\tau} h(\tau)h(\tau)^T \]
Computation Efficiency

- Fast weights computation is equivalent to attention to the past:
  \[ A(t)h_s(t + 1) = \eta \sum_{\tau=1}^{\tau=t} \lambda^{t-\tau} h(\tau)[h(\tau)^T h_s(t + 1)] \]

- Store hidden vectors instead of the fast weight matrix
Attention to the recent past

- Fast weights computation is equivalent to attention to the past:
  \[ A(t)h_s(t + 1) = \eta \sum_{\tau=1}^{\tau=t} \lambda^{t-\tau} h(\tau)[h(\tau)^T h_s(t + 1)] \]

- Fast weights is a biologically plausible implementation of the attention mechanism
Application: associative retrieval

c9k8j3f1 ?? c
Application: associative retrieval

c9k8j3f1 ?? c → 9
Application: associative retrieval

<table>
<thead>
<tr>
<th>Input string</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>c9k8j3f1??c</td>
<td>9</td>
</tr>
<tr>
<td>j0a5s5z2??a</td>
<td>5</td>
</tr>
</tbody>
</table>
Application: associative retrieval

<table>
<thead>
<tr>
<th>Model</th>
<th>R=20</th>
<th>R=50</th>
<th>R=100</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRNN</td>
<td>62.11%</td>
<td>60.23%</td>
<td>0.34%</td>
</tr>
<tr>
<td>LSTM</td>
<td>60.81%</td>
<td>1.85%</td>
<td>0%</td>
</tr>
<tr>
<td>A-LSTM</td>
<td>60.13%</td>
<td>1.62%</td>
<td>0%</td>
</tr>
<tr>
<td>Fast weights</td>
<td>1.81%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>
Application: recursive vision task

Classify facial expression from 48x48 images into six categories: {neutral, smile, surprise, squint, disgust and scream}
Application: recursive vision task

```python
def get_facial_expression(face):
    expression = None

    for facialFeature in face:
        featureExpression = get_facial_expression(facialFeature)
        expression = integrate_expression(expression, featureExpression)

    return expression
```
Application: recursive vision task

```python
def get_facial_expression(face):
    expression = None
    for facialFeature in face:
        featureExpression = get_facial_expression(facialFeature)
        expression = integrate_expression(expression, featureExpression)
    return expression
```
def get_facial_expression(face):
    expression = None
    for facialFeature in face:
        featureExpression = get_facial_expression(facialFeature)
        expression = integrate_expression(expression, featureExpression)
    return expression
Application: recursive vision task
Application: recursive vision task
Application: recursive vision task
### Application: recursive vision task

<table>
<thead>
<tr>
<th></th>
<th>IRNN</th>
<th>LSTM</th>
<th>ConvNet</th>
<th>Fast Weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test accuracy</td>
<td>81.11</td>
<td>81.32</td>
<td>88.23</td>
<td>86.34</td>
</tr>
</tbody>
</table>
Application: improving RL agents
Conclusion

- Fast weights provide a powerful *internal* storage mechanism for RNNs
  - Fast associative weights retrieves information by attracting new states of the hidden units towards similar recent hidden states
  - Layer normalization makes this kind of attention works much better
  - Hidden states are freed up to learn appropriate representation for the final classifier
  - Fast weights can be applied to solve recursive tasks without explicitly storing copies of hidden states