Recurrent Neural Nets II

Steven Spielberg Pon Kumar,
Tingke (Kevin) Shen
Machine Learning Reading Group, Fall 2016

9 November, 2016
Outline

1. Introduction
2. Problem Formulations with RNNs
3. LSTM for Optimization
4. Seq2Seq Learning
Feed-Forward Neural Networks (NN)
Rolling NN over time
Rolling NN over time
Rolling NN over time
Computation Flow in RNN
Computation Flow in RNN
Computation Flow in RNN
Computation Flow in RNN
Computation Flow in RNN
Computation Flow in RNN
Computation Flow in RNN
Computation Flow in RNN
Computation Flow in RNN
Computation Flow in RNN
Computation Flow in RNN
Computation Flow in RNN
RNN Representation
RNN Representation
RNN Representation
Given \( x_{t-3}, x_{t-2}, x_{t-1}, x_t \)

Find \( x_{t+1} \)
Time Series Prediction - Learning

- Error, \( e = \sum_{i=1}^{n} (\hat{x}_{t+1}^i - x_{t+1}^i)^2 \)
- Update Weights \( \theta \) with Error, \( e \) using Back Propagation through Time
- eg: Weather Forecasting, Stock Prediction etc.
Recurrent Neural Nets II
Problem Formulations with RNNs

Implementation

```python
import tensorflow as tf

# Inputs, Outputs and Weights
inputX = tf.placeholder("float", [None, MAX_STEP, INPUT_DIMENSION])
outputY = tf.placeholder("float", [None, N_CLASSES])
W = tf.variable()
B = tf.variable()

# Model Graph
lstm_cell = tf.nn.rnn_cell.LSTMCell(HIDDEN_UNITS=20)
lstm_layers = [lstm_cell]*N LAYERS
lstm_cell = tf.nn.rnn_cell.MultiRNNCell(lstm_layers)
lstm_outputs, _ = tf.nn.dynamic_rnn(lstm_cell, inputX)
prediction = tf.matmul(last_output(lstm_outputs), W) + B

# Loss and Optimization
loss = tf.pow(prediction - outputY, 2) / BatchSize
optimizer = tf.train.AdamOptimizer().minimize(loss)
for i in range(0, epoch):
sess.run(optimizer, feed_dict={inputX:X, outputY:Y})
```
Sentence Classification - RNN

- Error, \( e = -\sum_{i=1}^{n} y_i \times \log(\hat{y}_i) \)
- Update Weights \( \theta \) with Error, \( e \) using Back Propogation through Time
Sentence Classification - Bidirectional RNN

- Error, \( e = -\sum_{i=1}^{n} y_i \times \log(\hat{y}_i) \)
- Update Weights \( \theta \) with Error, \( e \) using Back Propogation through Time
Character Level RNN
Problem Formulations with RNNs

Character Level RNN

\[
\text{Loss} = - \sum_{i=1}^{n} \sum_{i=1}^{t} x_i \times \log \left( \hat{x}_i \right)
\]
Sampled Examples from Character level RNN

Lemma 0.1. Let $C$ be a set of the construction. Let $\mathcal{C}$ be a gerber covering. Let $\mathcal{F}$ be a quasi-coherent sheaves of $\mathcal{O}$-modules. We have to show that

$$\mathcal{O}_{\mathcal{O}_X} = \mathcal{O}_X(\mathcal{L})$$

Proof. This is an algebraic space with the composition of sheaves $\mathcal{F}$ on $X_{etale}$, we have

$$\mathcal{O}_X(\mathcal{F}) = \{\text{morphism } X \to Y\}$$

where $\mathcal{G}$ defines an isomorphism $\mathcal{F} \to \mathcal{F}$ of $\mathcal{O}$-modules.

Lemma 0.2. This is an integer $\mathbb{Z}$ is injective.

Proof. See Spaces, Lemma 7.

Lemma 0.3. Let $S$ be a scheme. Let $X$ be a scheme and $X$ is an affine open covering. Let $U \subseteq X$ be a canonical and locally of finite type. Let $X$ be a scheme. Let $X$ be a scheme which is equal to the formal complex. The following to the construction of the lemma follows.

Let $X$ be a scheme. Let $X$ be a scheme covering. Let $b : X \to Y \to Y \to Y \to Y \to Y \times_X Y$. be a morphism of algebraic spaces over $S$ and $Y$.

Proof. Let $X$ be a scheme of $X$. Let $X$ be an algebraic space. Let $\mathcal{F}$ be a quasi-coherent sheaf of $\mathcal{O}_X$-modules. The following are equivalent

1. $\mathcal{F}$ is an algebraic space over $S$.
2. If $X$ is an affine open covering.

Consider a common structure on $X$ and $X$ the functor $\mathcal{O}_X(U)$ which is locally of finite type.
Dynamic Systems

Loss = \sum_{i=1}^{n} \sum_{i=1}^{t} (\hat{x}_i^x - x_i^x)^2
Dynamic Systems Example

https://youtu.be/mEyQhkFwuPg
Optimization with LSTM

Gradient Descent

$$\theta_{t+1} = \theta_t + \alpha g(\nabla f(\theta))$$

where $g(\nabla f(\theta))$ is handcrafted update Rule

Learning Gradient Descent Update Rule

$$\theta_{t+1} = \theta_t + g(\nabla f(\theta), \phi)$$

where $\phi$ is parameters of LSTM
Learning to Learn Gradient Descent by Gradient Descent

Loss function, \[ f(\theta) = \sum (y - \theta_1 x + \theta_2)^2 \]
\[ \theta_{t-1} = \theta_{t-2} + g(\nabla f(\theta_{t-2}), \varphi) \]
Learning to Learn Gradient Descent by Gradient Descent

Loss function,

$$f(\theta) = \sum (y - \theta_1 x + \theta_2)^2$$

$$\theta_{t-1} = \theta_{t-2} + g(\nabla f(\theta_{t-2}), \varphi)$$
Learning to Learn Gradient Descent by Gradient Descent

Loss function, \( f(\theta) = \sum (y - \theta_1 x + \theta_2)^2 \)
\( \theta_{t-1} = \theta_{t-2} + g(\nabla f(\theta_{t-2}), \varphi) \)

\[ f(\theta_{t-2}) \]
\[ \theta_{t-2} \]
\[ \nabla f_{\theta_{t-2}} \]
\[ \varphi_x \]
\[ \varphi_y \]
\[ f(\theta_{t-1}) \]
\[ \theta_{t-1} \]
\[ g(\nabla f_{\theta_{t-2}}, \varphi) \]
Learning to Learn Gradient Descent by Gradient Descent

Loss function, \( f(\theta) = \sum (y - \theta_1 x + \theta_2)^2 \)

\( \theta_t = \theta_{t-1} + g(\nabla f(\theta_{t-1}), \varphi) \)
Learning to Learn Gradient Descent by Gradient Descent

Loss function,

\[ f(\theta) = \sum (y - \theta_1 x + \theta_2)^2 \]

\[ \theta_{t+1} = \theta_t + g(\nabla f(\theta_t), \varphi) \]
Learning to Learn Gradient Descent by Gradient Descent

Loss function,
\[ f(\theta) = \sum (y - \theta_1 x + \theta_2)^2 \]
\[ \theta_{t+1} = \theta_t + g(\nabla f(\theta_t), \varphi) \]

Loss function for RNN,
\[ f(\varphi) = \sum_{t=1}^{T} (f(\theta_t))^2 \]

\[ f(\theta_{t-2}) \]
\[ \theta_{t-2} \]
\[ \nabla f_{\theta_{t-2}} \]
\[ \varphi_y \]
\[ \varphi_x \]
\[ \varphi_{t-2} \]
\[ \varphi_2 \]
\[ \nabla f_{\theta_{t-1}} \]
\[ g(\nabla f_{\theta_{t-2}}, \varphi) \]
\[ \theta_{t-1} \]
\[ \nabla f_{\theta_{t-1}} \]
\[ \varphi_y \]
\[ \varphi_x \]
\[ \varphi_{t-1} \]
\[ \varphi_1 \]
\[ \nabla f_{\theta_t} \]
\[ g(\nabla f_{\theta_{t-1}}, \varphi) \]
\[ \theta_t \]
\[ \nabla f_{\theta_t} \]
\[ g(\nabla f_{\theta_t}, \varphi) \]
\[ \theta_{t+1} \]
\[ \varphi_2 \]
\[ \varphi_1 \]
Recurrent Neural Nets II
|-LSTM for Optimization

Learning to Learn Gradient Descent by Gradient Descent

Loss function,
\[ f(\theta) = \sum (y - \theta_1 x + \theta_2)^2 \]
\[ \theta_{t+1} = \theta_t + g(\nabla f(\theta_t), \varphi) \]

Loss function for RNN,
\[ f(\varphi) = \sum_{t=1}^{T} (f(\theta_t))^2 \]

\[ f(\theta_{t-2}) \quad \rightarrow \quad f(\theta_{t-1}) \quad \rightarrow \quad f(\theta_t) \quad \rightarrow \quad f(\theta_{t+1}) \]

\[ \nabla f_{\theta_{t-2}} \quad \rightarrow \quad g(\nabla f_{\theta_{t-2}}, \varphi) \quad \rightarrow \quad \theta_{t-1} \quad \rightarrow \quad \theta_t \quad \rightarrow \quad \theta_{t+1} \]

\[ \nabla f_{\theta_{t-1}} \quad \rightarrow \quad g(\nabla f_{\theta_{t-1}}, \varphi) \quad \rightarrow \quad \nabla f_{\theta_t} \quad \rightarrow \quad g(\nabla f_{\theta_t}, \varphi) \]

\[ \varphi_y \quad \rightarrow \quad \varphi_x \quad \rightarrow \quad \varphi_1 \quad \rightarrow \quad \varphi_2 \]
Learning to Learn Gradient Descent by Gradient Descent

Figure 4: Comparisons between learned and hand-crafted optimizers performance. Learned optimizers are shown with solid lines and hand-crafted optimizers are shown with dashed lines. Units for the y-axis in the MNIST plots are logits.

2.2 Information sharing between coordinates

In the previous section we considered a coordinatewise architecture, which corresponds by analogy to a learned version of RMSprop or ADAM. Although diagonal methods are quite effective in practice, we can also consider learning more sophisticated optimizers that take the correlations between coordinates into effect. To this end, we introduce a mechanism allowing different LSTMs to communicate with each other.

Global averaging cells

The simplest solution is to designate a subset of the cells in each LSTM layer for communication. These cells operate like normal LSTM cells, but their outgoing activations are averaged at each step across all coordinates. These global averaging cells (GACs) are sufficient to allow the networks to implement L2 gradient clipping [Bengio et al., 2013] assuming that each LSTM can compute the square of the gradient. This architecture is denoted as an LSTM+GAC optimizer.

NTM-BFGS optimizer

We also consider augmenting the LSTM+GAC architecture with an external memory that is shared between coordinates. Such a memory, if appropriately designed could allow the optimizer to learn algorithms similar to (low-memory) approximations to Newton's method, e.g. (L-)BFGS [see Nocedal and Wright, 2006]. The reason for this interpretation is that such methods can be seen as a set of independent processes working coordinatewise, but communicating through the inverse Hessian approximation stored in the memory. We designed a memory architecture that, in theory, allows the network to simulate (L-)BFGS, however we defer a detailed description of this architecture to Appendix B due to lack of space. We call this architecture an NTM-BFGS optimizer, because its use of external memory is similar to the Neural Turing Machine [Graves et al., 2014]. The pivotal differences between our construction and the NTM are (1) our memory allows only low-rank updates; (2) the controller (including read/write heads) operates coordinatewise.

3 Experiments

In all experiments the trained optimizers use two-layer LSTMs with 20 hidden units in each layer. Each optimizer is trained by minimizing Equation 3 using truncated BPTT as described in Section 2. The minimization is performed using ADAM with a learning rate chosen by random search. We use early stopping when training the optimizer in order to avoid overfitting the optimizer. After each epoch (some fixed number of learning steps) we freeze the optimizer parameters and evaluate its performance. We pick the best optimizer (according to the final validation loss) and report its average performance on a number of freshly sampled test problems.

We compare our trained optimizers with standard optimizers used in Deep Learning: SGD, RMSprop, ADAM, Adadelta, Adagrad, and Rprop. For each of these optimizer and each problem we try the following learning rates:

\[10^{-6}, 2 \cdot 10^{-6}, 2^2 \cdot 10^{-6}, \ldots, 2^{29} \cdot 10^{-6}\]. We report results with the learning rate that gives the best final error for each problem. When an optimizer has more parameters than just
Learning to Learn Gradient Descent by Gradient Descent

Figure 5: Comparisons between learned and hand-crafted optimizers performance. Units for the $y$-axis are logits.

Left: Generalization to the different number of hidden units (40 instead of 20).
Center: Generalization to the different number of hidden layers (2 instead of 1). This optimization problem is very hard, because the hidden layers are very narrow.
Right: Training curves for an MLP with 20 hidden units using ReLU activations. The LSTM optimizer was trained on an MLP with sigmoid activations.

Figure 6: Examples of images styled using the LSTM optimizer. Each triple consists of the content image (left), style (right) and image generated by the LSTM optimizer (center).

Left: The result of applying the training style at the training resolution to a test image.
Right: The result of applying a new style to a test image at double the resolution on which the optimizer was trained.

For LSTM+GAC and NTM-BFGS models we designate 5 of the 20 units in each layer as global averaging cells. NTM-BFGS uses one read head and 3 write heads.

Learning curves for different optimizers, averaged over many functions, are shown in the left plot of Figure 4. Each curve corresponds to the average performance of one optimization algorithm on many test functions; solid curves show learned optimizer performance and dashed curves show the performance of the standard hand-crafted baselines. It is clear the learned optimizers substantially outperform their generic counterparts in this setting, and also that the LSTM+GAC and NTM-BFGS variants, which incorporate global information at each step, are able to outperform the purely coordinatewise LSTM optimizer.

3.2 Training a small neural network on MNIST

In this experiment we test whether trainable optimizers can learn to optimize a small neural network on MNIST, and also explore how the trained optimizers generalize to functions beyond those they were trained on. To this end, we train the optimizer to optimize a base network and explore a series of modifications to the network architecture and training procedure at test time.
Sequence to Sequence Learning

- We covered how to use LSTM with fixed length inputs and fixed length outputs
- Sequence to Sequence learning uses two RNNs to solve general sequence to sequence problem of different length
- Eg: Machine Translation, Chatbots, Image Captioning etc.
Machine Translation
Machine Translation
Machine Translation
Machine Translation
Machine Translation
Machine Translation

- **Error**, $e = - \sum_{i=1}^{n} \sum_{t=1}^{T} y_{t}^{i} \times \log(\hat{y}_{t}^{i})$

- **Update Weights** $\theta$ with **Error**, $e$ using **Back Propogation through Time**
GPU Implementation

- 80k softmax by 1000 dims
- This is very big!
- Split softmax into 4 GPUs
- 160k vocab in input language
- 1000 LSTM cells
- 2000 dims per timestep
- 2000 x 4 = 8k dims per sentence

GPU1 → GPU2 → GPU3 → GPU4 → GPU5 → GPU6
GPU Implementation

- 80k softmax by 1000 dims
- This is very big!
- Split softmax into 4 GPUs
- 160k vocab in input language
- 1000 LSTM cells
- 2000 dims per timestep
- 2000 x 4 = 8k dims per sentence
- GPU1
- GPU2
- GPU3
- GPU4
GPU Implementation

This slide illustrates the GPU implementation of a Recurrent Neural Network (RNN) for Seq2Seq learning. The diagram shows the network's architecture divided across multiple GPUs to manage the large computational load. Specifically:

- The input vocabulary is 160k words.
- The LSTM cells have 1000 dimensions per timestep.
- Each sentence is represented in the network with 2000 x 4 = 8k dimensions.

The softmax layer, which is very large with 80k softmax by 1000 dimensions, is split into 4 GPUs to distribute the computation across these devices.
GPU Implementation

A  B  C  D

This is very big!
Split softmax into 4 GPUs
160k vocab in input language
1000 LSTM cells
2000 dims per timestep
2000 x 4 = 8k dims per sentence

GPU1
GPU2
GPU3
GPU4
GPU5
GPU6
80k softmax by 1000 dims
This is very big!
Split softmax into 4 GPUs
160k vocab in input language
1000 LSTM cells
2000 dims per timestep
2000 x 4 = 8k dims per sentence
GPU Implementation

- 80k softmax by 1000 dims
- This is very big!
- Split softmax into 4 GPUs
- 160k vocab in input language
- 1000 LSTM cells
- 2000 dims per timestep
- 2000 x 4 = 8k dims per sentence
- GPU1
- GPU2
- GPU3
- GPU4
- GPU5
- GPU6
80k softmax by 1000 dims
This is very big!
Split softmax into 4 GPUs
160k vocab in input language
1000 LSTM cells
2000 dims per timestep
2000 x 4 = 8k dims per sentence

GPU Implementation
Projection of the Encoder LSTM

**Table 1:** The performance of the LSTM on WMT'14 English to French test set (ntst14). Note that an ensemble of 5 LSTMs with a beam of size 2 is cheaper than a single LSTM with a beam of size 12.

<table>
<thead>
<tr>
<th>Method</th>
<th>Test BLEU Score (ntst14)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline System [29]</td>
<td>33.30</td>
</tr>
<tr>
<td>Bahdanau et al. [2]</td>
<td>28.45</td>
</tr>
<tr>
<td>Single forward LSTM, beam size 12</td>
<td>26.17</td>
</tr>
<tr>
<td>Single reversed LSTM, beam size 12</td>
<td>30.59</td>
</tr>
<tr>
<td>Ensemble of 5 reversed LSTMs, beam size 1</td>
<td>33.00</td>
</tr>
<tr>
<td>Ensemble of 2 reversed LSTMs, beam size 12</td>
<td>33.27</td>
</tr>
<tr>
<td>Ensemble of 5 reversed LSTMs, beam size 2</td>
<td>34.50</td>
</tr>
<tr>
<td>Ensemble of 5 reversed LSTMs, beam size 12</td>
<td>34.81</td>
</tr>
</tbody>
</table>

**Table 2:** Methods that use neural networks together with an SMT system on the WMT'14 English to French test set (ntst14).

*3.8 Model Analysis*

- **Figure 2:** The figure shows a 2-dimensional PCA projection of the LSTM hidden states that are obtained after processing the phrases in the Figure 2. The phrases are clustered by meaning, which in these examples is primarily a function of word order, which would be difficult to capture with a bag-of-words model. Notice that both clusters have similar internal structure.

- One of the attractive features of our model is its ability to turn a sequence of words into a vector of fixed dimensionality. Figure 2 visualizes some of the learned representations. The figure clearly shows that the representations are sensitive to the order of words, while being fairly insensitive to the translation.
Seq2Seq Application: Video to Text

Input:
https://www.youtube.com/watch?v=IiDyuE5qMOA

Output:
"A monkey is pulling a dog's tail and is chased by the dog."

Supervised problem: Given video and sentence pairs
Why Describe Videos?

- Robotics applications: human to robot interaction
- Describing videos for the blind
- Video indexing
- Just because
Alternative Models

Holistic video representation:

- Train classifiers to suggest subject, object, actions
- Combine objects/actions with language model using graphical model and "real world knowledge"
- Pick most probable subject, action, object triplet
- Insert triplet into sentence template

Fix video sequence length

- Encode video with vanilla NN
- Output sentence using LSTM

Recall: why Sequence to Sequence?

- Model ‘sees’ the entire input sequence before starting to output
- Output sequence length is not fixed to be equal to input sequence length
Recurrent Neural Nets II

Seq2Seq Learning

Model

S2VT Overview

Now decode it to a sentence!

Encoding stage

Decoding stage

Sequence to Sequence - Video to Text (S2VT)
Predictions

We want to pick the most probable sequence of words

- generate sentences greedily, picking highest softmax probability at each time step
- use beam search: e.g. try top 3 most probable words each time step and only keep top 3 most probable sequences so far
Examples

Demo:
https://www.youtube.com/watch?v=pER0mjzSYaM

Paper, code, examples:
https://www.cs.utexas.edu/~vsub/s2vt.html
Seq2Seq Models and ‘Attention’

Problem: Basic seq2seq RNN models cannot handle very long sequences

NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE
Authors: Dzmitry Bahdanau, KyungHyun Cho, Yoshua Bengio
Basic Seq2Seq Model

\[
p(y_i | \{y_1 \ldots y_{i-1}\}, x) = g(y_{i-1}, s_i, c)
\]
Attention Seq2Seq Model

\[ p(y_i \mid \{y_1 \ldots y_{i-1}\}, x) = g(y_{i-1}, s_i, c_i) \]

\[ c_i = \sum_{j=1}^{T} \alpha_{ij} h_j \]
Encoder: Bidirectional RNN

- 2 independent RNNs, 1 each direction
- Overall hiddens is concatenation of 2 independent hiddens
- Hiddens at each time contain information from entire input
- $h_j$ is influenced by inputs around $x_j$ the most
Decoder: Attention

\[ c_i = \sum_{j=1}^{T} \alpha_{ij} h_j \]

\[ \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T} \exp(e_{ik})} \]

\[ e_{ij} = a(s_{i-1}, h_j) \]
Improvements
Attention Visualization

- Picture shows matrix of $\alpha_{ij}$
- $\alpha_{ij} \sim$ relevance of input word $j$ to output word $i$
- White is 1, black is 0
Thank you!