Multi-layer GRU using character level information for SQUAD challenge

Jinseok Jake Yoon
Stanford University
jakeyoon@stanford.edu

Abstract

In this paper, we have implemented additional features on top of the baseline model for the SQUAD challenge. We applied a bi-directional GRU layer that encodes character level information of the word vector. Afterward, we have applied bi-directional attention mechanism on the combined information with the word-level embedding. We have tested our model with five different level of regularization(dropout). But, we were not able to achieve the better performance than the baseline model. We have examined 50 errors made by our best performing model. The most frequent type (40-50%) of error was that the unnecessarily long answers that contain true answer in it. In this case, the F1 score was in between 10-20%, and of course, EM score was False. This explains that our dev F1 and EM score were stuck a plateau($\approx$ 30-35 F1 score, 20-25 EM score). To improve, we need to train our model in a larger dataset, since our model got overfitted to the training dataset. (77% F1 score)

1 Introduction

The initial intention of this experiment was to reproduce the result of the paper Bi-directional attention flow for machine comprehension (Minjoon Seo et al. [1]) and make improvements on top of it. But, due to the time constraints, we could not implement all features of the Minjoon Seo et al. Instead, we started this experiment by implementing the bi-directional attention mechanism. In the first prototype, we have implemented the similarity score tensor and the additional Question to Context attention flow mechanism as directed in the default final project paper [2]. The result was not very successful, as it gave the similar performance as the baseline model. Next, we have tried an LSTM cell in the bi-directional RNN layer with the bi-directional attention flow mechanism. Again, the result was almost similar to the previous model and even worse, the training was slower than using the same GRU cell as the baseline model.

After a thorough investigation, we have found a major bug in the code. Having the bug fixed, we have implemented additional bi-directional GRU layer that encodes character level information of each word. That is, we averaged the character level encoding vector of a word into a single vector, then we concatenated it with the original word vector. The intention of building this mechanism was to introduce a way to process out-of-vocabulary words rather than just marking them <unk> in the word vector embedding.

2 Brief history of this experiment

2.1 GRU with bi-directional attention flow

Here we have implemented the bi-directional attention flow model as directed in the default final project paper. We have trained our model with the 0.03, and 0.15 dropout probability. Here is the Tensorboard plot for those models.

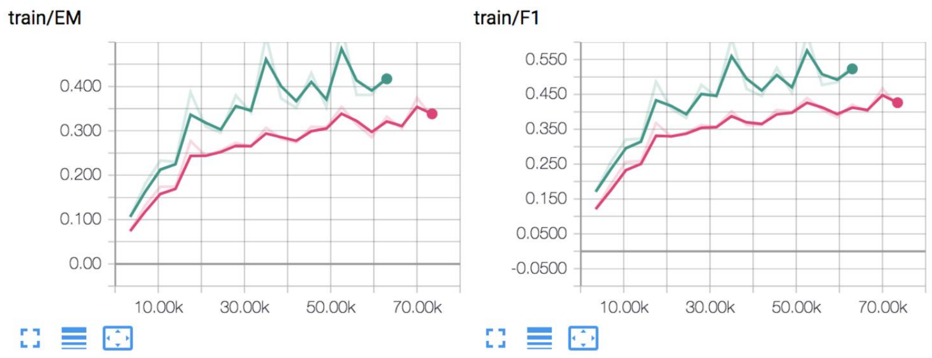

Figure 1: Traning result (Green : 0.03, Magenta : 0.15 dropout probability)

Due to the high-rank tensors in the similarity matrix, we had to reduce the batch size (≈ 15) significantly to fit our model to the given GPU memory constraint (8GB). Thus, the training took 4-5 times longer than the baseline model (≈ 4.1 sec on average). From the figure 1, it seems our model is still not complex enough to capture variance in the training set since our model wasn't able to reach 60% F1 score in the training set.

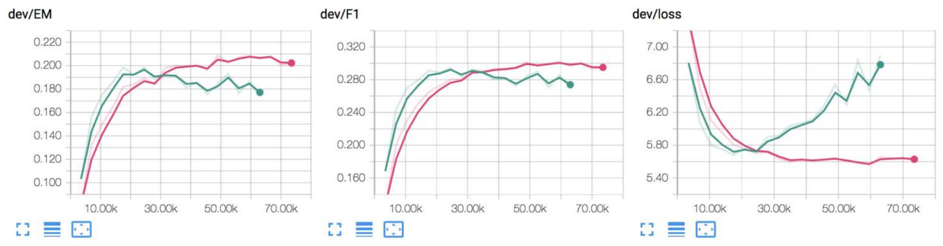

Figure 2: Traning result (Green : 0.03, Magenta : 0.15 dropout probability)

From the Figure 2, we can see that the low dropout model started to overfit the training set around 20-25K iterations, whereas the high dropout model reached a plateau after 40K iterations. The problem was that even with the bi-directional attention flow, this model performed worse than the baseline model (30% F1, 20% EM, compared to 39% F1, 28% EM).

After a thorough investigation, we have found a major bug in the code. The softmax function was applied to the wrong axis of the tensor (β in the 5.1.1 of the default final project paper) in the Question to Context attention flow, thereby eliminating all relevant information from this flow (setting all elements to 1). Having the bug fixed, we have implemented additional bi-directional GRU layer that encodes character level information of each word.

3 Model

Here is the introduction to our model's architecture.

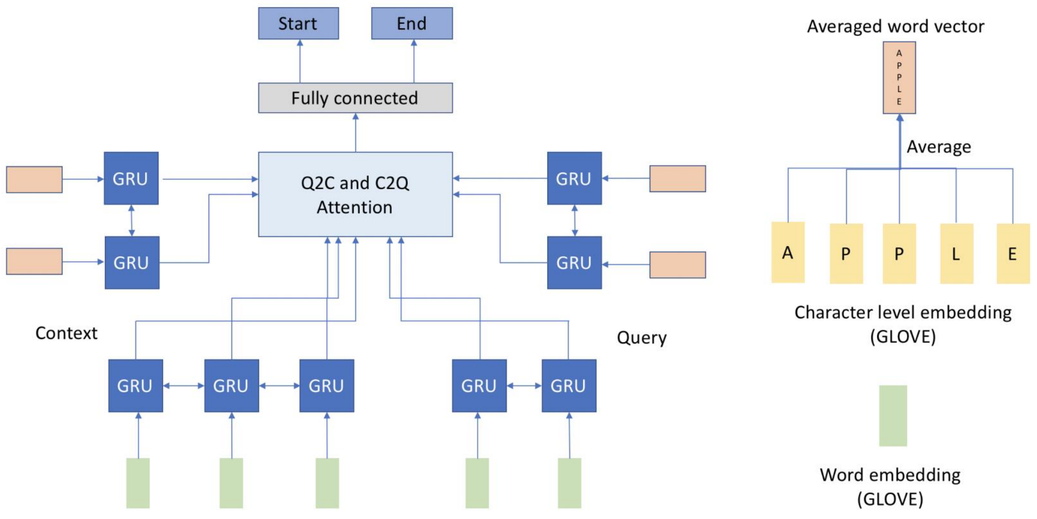

Figure 3: Model architecture

3.1 Word embedding layer

Word embedding layer is a mapping between words and a vector space, where every word gets vector representation. Here we have used the pretrained GLOVE word vectors (Pennington et al. [3]), and the dimension of each word vector was 100.

3.2 Character embedding layer

Similar to the word embedding layer, character embedding layer is a mapping between characters and vector space, where every alphanumeric character(including special characters, e.g. !@#$) gets vector representation. The intention of building character level embedding into this model was to introduce a way to generalize better on the out-of-vocabulary words in the given text. From the investigation of the previous model, we have found that unknown tokens are quite frequently appearing in the context, such as _budzyski_ and zbigniew _badowski_ (here underscore denotes that the word is unknown to the word embedding matrix).

Therefore, we have built a layer that extracts character level embedding of each word using the pretrained GLOVE character embeddings (300 dimensions). Then, we have averaged each vector representation of characters in a word and used it as augmented word vector representation.

$$\frac{1}{n}\sum_{i=1}^{n} c_i = w_k$$

(where c_i are character embedding vectors and n is the length of the word w_k) The character-level averaged word vector was fed to bi-directional GRU layer, then its hidden state was concatenated with the hidden states from running bi-directional GRU on the word-level embedding vectors. The augmented hidden state was fed into the bi-directional attention flow layer as depicted in Figure 3.

$$h_i = biGRU(w_1, \dots, w_k)$$
$$\tilde{h}_i = biGRU(e_1, \dots, e_k)$$
$$\hat{h}_i = [h_i; \tilde{h}_i]$$

(here e_i denotes word-level GLOVE embedding, $\hat{h}_i$ was fed to bi-directional attention flow layer)

3.3 Bi-directional attention flow layer

The basic idea of bi-directional attention flow is the attention information should flow mutually not unidirectionally. Here we have followed the implementation details of the default final project paper. First, we have built the similarity matrix which contains the similarity between context and question hidden states.

$$S_{ij} = w_{sim}^T [c_i; q_i; c_i \odot q_i] \in \mathbb{R} \tag{1}$$

Then, the Context to Question attention was calculated from the weighted sum of the question hidden states where the weight is from the row-wise softmax value of the similarity matrix. The Question to Context attention was also calculated as a weighted sum of context hidden states, but in this case, the weight was the softmax value on top of the row-wise max value of the similarity matrix.

4 Experiments

4.1 Experiment conditions

Here are the hyperparameters that were used in our model. Batch size : 15, learning rate : 0.001, dropout rate : 0.03 - 0.25, hidden layer size : 150, word embedding size : 100, character embedding size : 300, maximum word length : 10 characters, maximum gradient norm : 5.0, question length : 30, context length : 600, optimizer : Adam optimizer.

4.2 Results

We have tested our model with following dropout rates [0.03, 0.07, 0.10, 0.15, 0.25]. Here is the result of 0.03, 0.07 dropout.

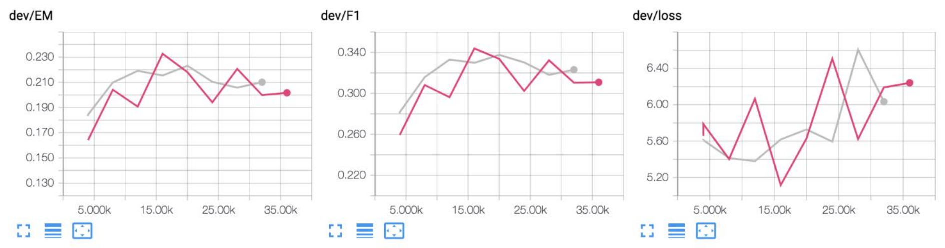

Figure 4: Dev set result (gray : 0.03, magenta : 0.07 dropout)

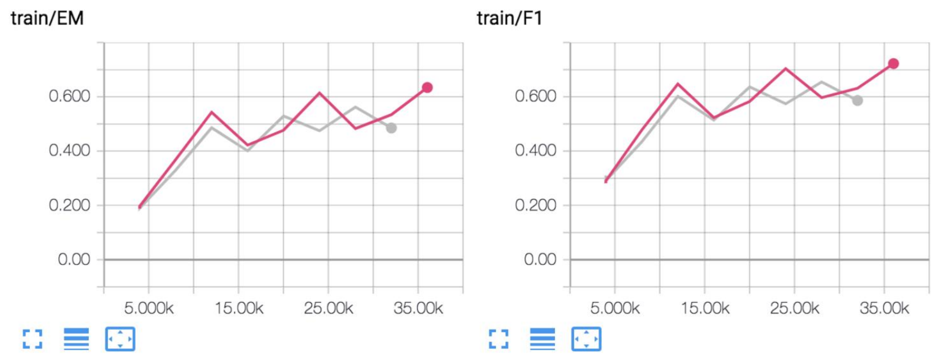

Figure 5: Training result (gray : 0.03, magenta : 0.07 dropout)

As we can see from the above figures, dropout rates lower than 10% make the model to be overfitted to the training dataset. In both cases, dev loss has started to rise around 15K iteration, which means the models have failed to generalize on the dev dataset. On the other hand, our model has reached 77% F1 score on the training dataset after 35K iteration; this means that the model is more complex than the baseline model and is powerful enough to capture variances of the training data.

4.2.1 gradient exploding problem

Interestingly, we have observed the gradient exploding problem with 0.10% dropout probability setting.

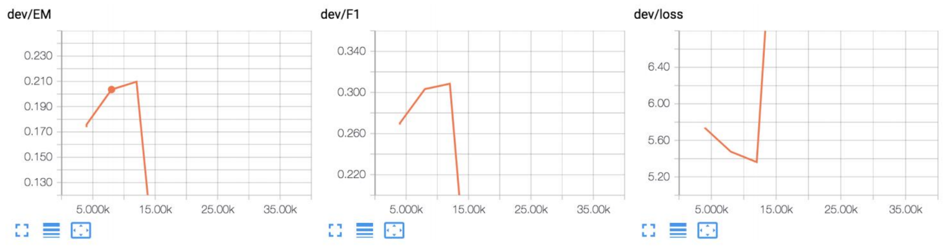

Figure 6: Dev set result (0.10 dropout)

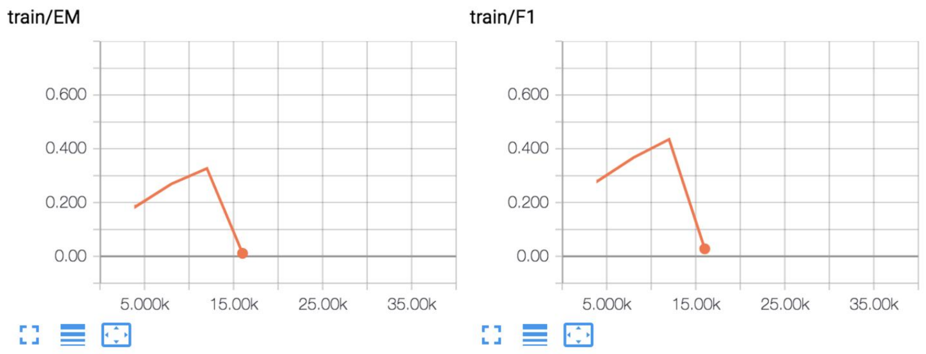

Figure 7: Training result (0.10 dropout)

As we can see from the above graph, the F1 and EM score plunged to zero on both training and dev dataset, and the dev loss skyrocketed. The interesting point was that the model was still trainable since the norm of the gradient was limited to 5.0, even though the parameters of the model were totally ruined. This result suggests that the maximum allowed norm of the gradient for this model is still too large enough to destroy the parameters.

```
epoch 2, iter 12336, loss 4.48446, smoothed loss 5.07060,
grad norm 742814646272.00000, param norm 398.28198, batch time 4.454

epoch 2, iter 12337, loss 6.36920, smoothed loss 5.08359,
grad norm 33718753280.00000, param norm 398.29202, batch time 4.197

epoch 2, iter 12341, loss 4.63616, smoothed loss 5.05918, grad norm inf,
param norm 398.32587, batch time 4.315
```

4.2.2 plateaus

The biggest problem of all was that even with the appropriate regularization, our model was stuck at a plateau with ≈ 32 F1 score and 22 EM score.

5 Discussion

5.1 Dealing with plateaus

Considering the fact that our model can reach $\approx 77\%$ F1 score on the training set, it is unlikely the case that our model is not complex enough to reach over 40% F1 score on the dev set. There are several possible explanations for our model being stuck at a plateau.

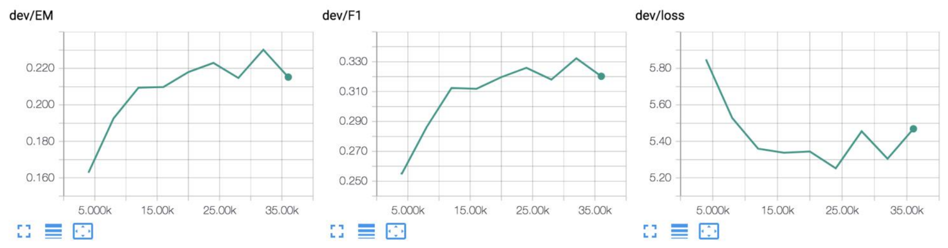

Figure 8: Dev set result (0.15 dropout)

5.1.1 stuck in the local optima & saddle point

Since optimizing the neural network being a non-convex optimization problem. It is possible that our hyperparameter setup was a poor initial condition, thereby our model was stuck at a plateau. Due to the time constraint, we were not able to do exhaustive hyperparameter search other than the dropout rate and gradient clipping. We used Adam optimizer during the whole experiment, and since the Adam optimizer keeps an exponentially decaying average of previous gradients, we could have tested different parameter values for the optimizer other than the default ones given by TensorFlow.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{\nu}_t} + \epsilon}\hat{m}_t$$
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

Diederik P. Kingma et al.[4] suggested 0.9 for β_1, but we think lowering β_1 might give a better result. Since momentum had already died out on top of a plateau, dropping the weight of the previous momentum might help the model to escape from the plateau.

Another possibility is that our model could have been stuck at a saddle point. A saddle point is a point on the surface of the graph of a function where the slopes (derivatives) of orthogonal function components defining the surface become zero (a stationary point) but are not a local extremum on both axes [5]. According to Yann N. Dauphin et al. [6], there are infinitely many saddle points in the non-convex optimization problem. Furthermore, Ge et al. 2015 [7], showed that noisy gradient descent can find a local minimum of strict saddle functions in polynomial time. Therefore, if this is the case, reducing the batch size might help our model escape from a plateau, since reducing the batch size makes the stochastic gradient descent noisier.

5.1.2 error analysis

We have examined 50 errors made by our best performing model. The most frequent type of error was that the unnecessarily long answers. Such as, for the question, ”which logo had the dw tardis insignia removed?”, our model's answer was [”tardis insignia placed to the right in 2012 , but the same font remained , albeit with a slight edit to the texture every episode , with the texture relating to some aspect of the story . **the logo for the twelfth doctor** had the ” dw ” tardis insignia”], where the true answer was just **the logo for the twelfth doctor**. Almost 40-50% of the errors were this type, which are unnecessarily long sentences that contain true answer in it. In this case, the F1 score was in between 10-20%, and of course, EM score was False (0). This explains that our dev F1 and EM score were stuck a plateau($\approx$ 30-35 F1 score, 20-25 EM score), and to be overfitted to the training set.

On the other hand, the errors that our model predicted completely unrelated answers were relatively rare[1]. We think this is the sign that at least our bi-directional attention mechanism is working correctly since our model is paying attention to words that are near the correct answer. In order to improve this model, we think we need to make our final prediction layer more sophisticated, which was just argmax of the softmax value of start and end token probability distribution. Also training our model on a larger training set will be very helpful since it seems that our model's final softmax layer wasn't trained well enough to generalize on the unseen dataset.

[1]Less than 20% of the 50 error cases

5.2 Dealing with gradient exploding problem

In a recurrent neural network, error gradients can accumulate during the backward propagation, and quickly become a very large value. We have found that increasing learning rate significantly increases the chance of gradient exploding problem. Although we could not explicitly measure the relationship between the learning rate and the probability of gradient exploding problem, in the cases that learning rate was greater than 0.005 [2], all of them had gradient exploding problem within 1,000 training iteration. This is mainly due to the architecture of our model. Since gradients with larger than 1.0 exponentially grow within a recurrent neural network, a large learning rate might have caused a large gradient, and consequently, a gradient exploding problem.

Furthermore, we also have found that clipping the maximum value of norm of the gradient is not very effective in protecting the parameters from the gradient exploding problem. We have tested [0.5 - 5.0][3]a range of max norm values, but none of them was able to protect the model, and the parameters were instantly devastated as soon as a large gradient appears.

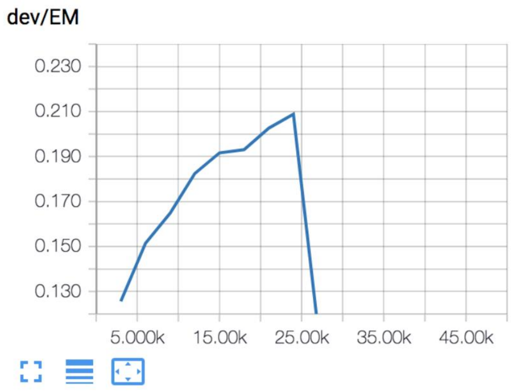

Figure 9: An instance of gradient exploding (learning rate was increased from 0.001 to 0.005 at 24K iteration)

According to Sekitoshi Kanai et al.[8], decomposing SVD of the weight matrix and reconstructing the weight matrix with singular values that are larger than the threshold might prevent a gradient exploding problem in a multi-layer GRU architecture.

$$W_{hh} = U\Sigma V$$

$$\hat{\Sigma} = diag(min(\sigma_1, 2-\delta), \ldots, min(\sigma_n, 2-\delta))$$

(where $2-\delta$ is the threshold)

$$\hat{W_{hh}} := U\hat{\Sigma}V$$

Since the computational complexity of SVD is $O(n^3)$, it might be computationally heavy to apply this method to our model directly. If that is the case, we can consider replacing the character-level GRU layer to other neural models that are more faster to compute, such as Convolutional neural network, or we could just simply reduce the level of GRU layer.

5.3 Training the model

We used Azure's NV6 standard(E5-2690v3, 6 cores, 56GB memory, Tesla K80 GPU) and NV24(E5-2690v3, 24 cores, 224GB memory, 4$\times$Tesla K80 GPU). It took about two days to train our model to reach the best performance using a single Tesla K80 GPU. We trained multiple hyperparameter settings in parallel using the four Tesla K80 GPUs on the NV24 machine. Regarding batch training time, the performance of a single Tesla K80 GPU was similar to training our model using 24 core

[2]The default learning rate given by the baseline model was 0.001

[3]The gradient was clipped to 5.0 in the baseline model

CPU. The only problem with the GPU was that the memory of GPU was limited to 8GB. Since our model uses a large tensor object during the computation of bi-directional attention, it was a major obstacle to our model. Regarding memory efficiency, it is highly advised that we should avoid using tiling operation as much as possible since it was the major cause of the out of memory error. Also, concerning computational complexity, we need to avoid broadcasting a large tensor object. The single most computationally heavy computation was broadcasting of similarity vector (w_{sim}^T of equation (1)), and the batch process time almost doubled by this operation (on average, from 2.0 sec to 4.0 sec).

6 Conclusion

In this paper, we have implemented additional features on top of the baseline model for the SQUAD challenge. We applied a bi-directional GRU layer that encodes character level information of the word vector. Afterward, we have applied bi-directional attention mechanism on the combined information with the word-level embedding. We have tested our model with five different level of regularization(dropout). But, we were not able to achieve the better performance than the baseline model.

Because our model can reach $\approx 77\%$ F1 score on the training set, it is unlikely the case that our model is not complex enough to reach over 40% F1 score on the dev set. We have examined 50 errors made by our best performing model. The most frequent type (40-50%) of error was that the unnecessarily long answers that contain true answer in it. On the other hand, the errors that our model predicted completely unrelated answers were relatively rare. We think this is the sign that at least our bi-directional attention mechanism is working correctly since our model is paying attention to words that are near the correct answer. To improve this model, we think we need to make our final prediction layer more sophisticated, and also training our model on a larger training set, since it seems that our model's final softmax layer wasn't trained well enough to generalize on the unseen dataset.

Finally, the gradient exploding problem quite frequently appeared in the learning rates that are larger than the default one. Furthermore, we also have found that clipping the maximum value of norm of the gradient is not very effective in protecting the parameters from the gradient exploding problem. Sekitoshi Kanai et al. proposed that decomposing SVD of the weight matrix and reconstructing the weight matrix with singular values that are larger than the threshold might prevent a gradient exploding problem in a multi-layer GRU architecture.

References

[1] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension. arXiv preprint arXiv:1611.01603, 2016.

[2] Default final project paper for CS224N. http://web.stanford.edu/class/cs224n/default_project/default_project_v2.pdf

[3] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In EMNLP, 2014.

[4] Diederik P. Kingma, Jimmy Lei Ba. ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION. arXiv:1412.6980v9 [cs.LG] 30 Jan 2017

[5] Saddle Point, Wikipedia. https://en.wikipedia.org/wiki/Saddle_point

[6] Yann N. Dauphin, et al. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. arXiv:1406.2572v1 [cs.LG] 10 Jun 2014

[7] Rong Ge, Furong Huang, Chi Jin, Yang Yuan. Escaping From Saddle Points — Online Stochastic Gradient for Tensor Decomposition. arXiv:1503.02101 [cs.LG]

[8] Sekitoshi Kanai, Yasuhiro Fujiwara, Sotetsu Iwamura. Preventing Gradient Explosions in Gated Recurrent Units. NIPS 2017, 324.