ON BAYESIAN RECURRENT NEURAL NETWORKS APPLICABILITY TO FINANCIAL DATA

Ramiro Mata, Manuel Montoya Catala, Peter Holler Langhorn

Technical University of Denmark
Department of Applied Mathematics and Computer Science

ABSTRACT

Deep Neural Networks' powerful expressivity can often make them vulnerable to overfitting. Methods such as drop-out, pruning, and L1 and L2 regularization were built to overcome this problem. Among these, drop-out is the most celebrated regularization technique in deep learning architectures, but its effectiveness on Recurrent Neural Networks is limited due to noise propagation through time [2]. In this paper, we implement the novel regularized training algorithm, 'Bayes By Backprop Through Time' (BBBTT), from Fortunato et al. [1]. We test the algorithm on the Penntree data bank on the same task as in [3] which is a next-word prediction task. Experiments show that this algorithm can have significantly better performance over drop-out in certain model complexity configurations. Moreover, in this paper we explore the capabilities of the algorithm using simulated data, showing its potential use in the prediction of time series.

1. INTRODUCTION

Deep Neural Networks' ability to form complex relationships between their input and output makes them very expressive models, but with limited training data, this can also result in overfitting. Several methods to avoid learning the complex noisy patterns in the training dataset exist, such as early stopping, L1 and L2 weight regularization, and drop out. While these have been successfully applied to feed-forward neural networks (FFN), applying these methods to Recurrent Neural Networks (RNN), drop out in particular, has come with its own challenges. Specifically, it has been claimed that drop out in RNNs hurts learning due to amplified noise through recurrent connections [2].

As a result, using RNNs in practice has oftentimes lead to adoption of small RNN architectures because larger, more complex RNNs overfit. Zaremba et al., however, successfully applied a variation of drop out to RNNs with Long Short Term Memmory (LSTM) units. This was accomplished by applying drop out only to non-recurrent connections [3].

Thanks to Carsten Stahlhut and Alipes Capital for their support and guidance.

Recently, Fortunato et al. proposed the architecture Bayesian Recurrent Neural Networks. In this model, the weights of the network, θ, are modeled as hidden random variables instead of point values. The algorithm requires a prior $p(\theta)$ and a posterior $q(\theta)$ which comprises an approximation of the posterior distribution over the model's parameters (weights). During training, the loss function to minimize is the Free Energy function, obtained as a lower bound of the incomplete log likelihood of the data using the Variational Inference Scheme. We approximate the integral over the parameters of this function by means of Monte Carlo sampling of the weights from their posterior distribution at each batch during training.

The main advantages of this method are its regularization properties as well as its explicit representation of uncertainty. In this paper we attempt to replicate the results of [1] on the small and medium model configurations as they are given in [1] for the next-word prediction task, and we compare its performance with [3]'s variation of drop out.

We also explore its prediction capabilities on continuous time series that were generated using a Gaussian Process with a known mean function from which we label the data and train the network to estimate the probability that the next sample will have a bigger or lower value than the current one.

Source code can be found in the repository which is located here.

2. BACKGROUND

2.1. Making an RNN Bayesian

As any other neural network, an RNN is a parametric model which contains weights as its parameters θ. Generally speaking, we want to find the values of the weights of the RNN that minimize some loss function $\mathcal{L}(\theta)$. Since this loss function generally is non-convex on the parameters, we typically use a local search algorithm, such as Backpropagation, in which at each iteration we modify all the parameters in the direction that minimizes the loss function, given by the gradient. In the general approach, the weights are deterministic, i.e. they are point values that are modified at each iteration of the training algorithm.

Other more elaborate approaches have a modified loss function based on a Bayesian approach, where they impose a prior $p(\theta)$ on the weights. This is the case for example with the L1 regularization when the prior is Laplacian or L2 regularization when it is Gaussian. These techniques although based on a Bayesian approach, where the loss function is the negative logarithm of the posterior probability of the weights:

$$\mathcal{L}(\theta) \propto -log(p(\theta|\mathcal{D})) = -log(p(\mathcal{D}|\theta)) - log(p(\theta))$$

still only compute point values of the parameters θ, without estimating their posterior probability, since the closed form solution of the integral needed to compute the posterior is intractable. Other approaches such as Laplace approximation, averaging over different initializations or Monte Carlo sampling could be used to compute an estimation of this posterior distribution.

Instead of this classic approach, Fortunato et al. proposes a Variational Inference approach where *all* weights in an RNN are hidden random variables with a prior $p(\theta)$ and an approximate posterior $q(\theta)$, called the variational posterior. The loss function in this case is a lower bound of the negative incomplete data log likelihood, obtained when we used the variational posterior instead of the real posterior.

$$-\log(p(\mathcal{D})) \leq \mathcal{L}(\theta) = -\mathbb{E}_{q(\theta)}[\log(p(\mathcal{D}|\theta))] + \text{KL}[q(\theta)||p(\theta)]$$

Thus, instead of having deterministic weights being sequentially tuned at each train iteration (batch) of the backpropagation algorithm; we now sequntially tune the parameters of the approximate posterior to minimize the loss function, which increases the lower bound of the incomplete data log likelihood $p(\mathcal{D})$.

In order to do this, at each iteration we obtain an approximation of the loss function by means of Monte Carlo sampling of the current posterior weights $q(\theta)$ and we update the parameters of the variational posterior through Bayes by Backprop (see section 2.2 for details on BBB) such that we minimize the variational free energy.

2.2. Bayes By Backprop

In order to tune the parameters of the variational posterior distribution on the weights, Fortunato et al. proposes a training algorithm based on Backpropagation using as loss function, an estimation by Monte Carlo simulation of the variational free energy. They call this algorithm ”Bayes By Backprop” (BBB) and the pseudo-code for this sequential procedure can be seen below (Algorithm 1).

In their implementation, they propose a variational posterior with a Guassian form for every weight θ, where each weight has its own independent mean μ and standard deviation σ. Furthermore they describe the algorithm in a mini-batch mode where we divide our dataset into B batches containing C chains of equal length each. We therefore have a total of $N = BC$ sequences of the same length which in a normal set up are considered independent.

In practice these equal-length chains are obtained by chopping original long sequences by the specified length, obtaining what are called ”truncated sequences” which are not necessarily independent, as it is the case of chopping sentences of a document. This violation of the assumption is partially solved by not resetting the state of the LSTM cells to 0, but injecting the final state of the previous chain, this way information from the previous chain is given to the next.

This is to address the problem with e.g. vanishing gradients when the sequences become too long. Furthermore, training on several sequences at once helps to reduce the variance in the gradients. So a mini-batch could e.g. consist of 5 sequences which are 10 time steps long.

Algorithm 1 Bayes by Backprop for RNNs

```
Sample $\varepsilon \sim \mathcal{N}(0, I), \varepsilon \in \mathbb{R}^d$
Set network parameters to $\theta = \mu + \sigma\varepsilon$
Sample a minibatch of truncated sequences $(x, y)$
Do forward propagation and backpropagation as normal on minibatch
Let $g$ be the gradient with respect to $\theta$ from backpropagation.
Let $g_\theta^{KL}, g_\mu^{KL}, g_\sigma^{KL}$ be the gradients of $\log \mathcal{N}(\theta|\mu, \sigma)$ - $\log p(\theta)$ w.r.t. $\theta, \mu$ and $\sigma$ respectively.
Update $\mu$ according to the gradient $\frac{g + \frac{1}{C} g_\theta^{KL}}{B} + \frac{g_\mu^{KL}}{BC}$
Update $\sigma$ according to the gradient $\left(\frac{g + \frac{1}{C} g_\theta^{KL}}{B}\right)\varepsilon + \frac{g_\sigma^{KL}}{BC}$
```

As mentioned we seek to minimize the variational free energy for an RNN on a sequence of length T as given in [1] as

$$\mathcal{L}(\theta) = -\mathbb{E}_{q(\theta)}[\log p(y_{1:T}|\theta, x_{1:T}] + \text{KL}[q(\theta)||p(\theta)]$$

However, since training is done on mini-batches, the variational free energy becomes

$$\mathcal{L}_{(b,c)}(\theta) = -\mathbb{E}_{q(\theta)}\left[\log p(y^{(b,c)})|\theta, x^{(b,c)}, s_{prev}^{(b,c)}\right] + w_{KL}^{(b,c)} \text{KL}[q(\theta)||p(\theta)]$$

Here (b, c) is the elements of the cth truncated sequence of the bth mini-batch, with B the number of mini-batches and C the number of truncated sequences as mentioned earlier. We note that the KL term now is penalized by the $w_{KL}^{(b,c)}$ term which is equally distributed over all mini-batches and truncated sequences, i.e. $w_{KL}^{(b,c)} = \frac{1}{CB}$. Furthermore, $s_{prev}^{(b,c)}$ is the initial state of the RNN for the mini-batch $x^{(b,c)}$. Since truncated sequences are picked in order, we have that $s_{prev}^{(b,c)}$ is set to the last state of the RNN for $x^{(b,c-1)}$.

As it can be seen in the description, the way the BBB algorithm works is by first drawing a sample from a normalized Gaussian distribution for each weight. Then, through the reparamaterization trick, we obtain the weights as given by $\theta = \mu + \sigma\varepsilon$. Here the reparameterization trick enables us to make the parameters of our distribution trainable (μ and σ). Then we can do forward and backpropagation on a mini-batch. We then get a gradient g from the negative log likelihood term and the gradients $g_\theta^{KL}, g_\mu^{KL}, g_\sigma^{KL}$ for the KL divergence term. We can then update the parameters μ and σ by the expressions given in the algorithm above, moving the parameters in the direction which minimizes the variational free energy.

3. PTB DATA

As mentioned earlier, we tested our model's performance on the Penn Tree Bank (PTB) dataset which is a dataset containing words (sentences) originating from a collection of Wall Street Journal stories. Specifically, the dataset contains 929.000 training words, 73.000 validation words and 82.000 test words and comprises a vocabulary of 10.000 words. Among the words, are certain "identifiers" or "tags" such as "`N`" replacing every number in the data set and the tag `<unk>` (meaning unknown) for words not in the vocabulary.

Before feeding words from the PTB data set to the model, the words are processed in a specific way. First of all, each word in the vocabulary is assigned an integer id. Thus the word ids range from 0 to 9999. Furthermore, each word is embedded such that the model is being fed a real-valued vector instead of just a single integer id (e.g. a 200 long real-valued vector). One can then simply look up the model's predicted word(s) in the vocabulary when done training/testing.

4. EXPERIMENTS

In this section we describe the set up for the experiments carried out with the PTB dataset and the simulated data. First of all we will talk about the prior and posterior forms chosen. Since we are not using the closed form solution of KL divergence between the prior and the variational posterior, but computing an approximation using Monte Carlo sampling, we are not constrained to only using Gaussian distributions for which we have their closed form equation. In order to replicate the results in [1] we used a Mixture of Gaussians for the prior with 0 mean and variances σ_1^2 and σ_2^2 to give more expressivity to the prior, i.e. allowing fatter tails. The distribution of the weights θ is then given by:

$$p(\theta) = \pi\mathcal{N}(\theta|0, \sigma_1^2) + (1-\pi)\mathcal{N}(\theta|0, \sigma_2^2)$$

Regarding the variational posterior $q(\theta)$, we chose to use a diagonal Gaussian for the weights. In the first iteration of the algorithm, we set the parameters of the posterior as a single Gaussian approximation of the prior, that is mean 0 and variance σ_{mix}^2, being:

$$\sigma_{mix} = \pi\sigma_1 + (1-\pi)\sigma_2$$

4.1. Next word prediction

Our primary experiment was the next word prediction task on the PTB dataset. We considered the "small" and "medium" model sizes as defined in [3]. Both configurations use a two-layer LSTM cell framework. The "small" model configuration consists of a batch size of 20 sequences having a sequence length of 20. So for each training batch we feed 20×20 words to the model. Furthermore, for the "small" model the number of hidden units per gate in an LSTM cell is 200, totalling 800 units per LSTM cell. The "small" model trains over 13 epochs.

The "medium" configuration has a batch size of 20 as well. However, the sequence length is extended to 35, such that a batch comprises 20×35 words. Furthermore, the number of hidden units per gate is increased to 650. Lastly, the "medium" model trains over 70 epochs.

As mentioned, the task is next word prediction, meaning for a time step t we feed a word through the model and get a probability distribution over all 10.000 possible words. The model should hopefully learn to choose the next word in the sequence as the most probable word.

4.2. Predictions on simulated data

In order to test the prediction capabilities of the algorithm for continuous signals, we modified the model to accept chains or real-valued vectors directly (instead of ids that are converted into random vectors). We also increased the possible configurations of the network so that it is able to have a different number of hidden neurons in the LSTM cells than the length of the chain.

Since the implemented algorithm performs classification, the task designed consists on predicting whether a continuous signal will increase or decrease in the next time instant. The data generated comes from a Gaussian Process with an underlying mean function $\mu(t)$ composed by two sinusoids and a linear trend. The label associated to each sample is 0 if the underlying mean signal will decrease and 1 if it will increase or stay the same.

$$X(t) = \mu(t) + \epsilon(t)$$

$$Y(t) = \begin{cases} 0 & \mu(t+1) < \mu(t) \\ 1 & \mu(t+1) \geq \mu(t) \end{cases}$$

The term $\epsilon(t)$ accounts for the noise of the Guassian process, we can vary the intensity and correlation between the noisy components to test the performance of the algorithm. Figure 1 shows an example of the underlying mean function $\mu(t)$ and its associated labelling.

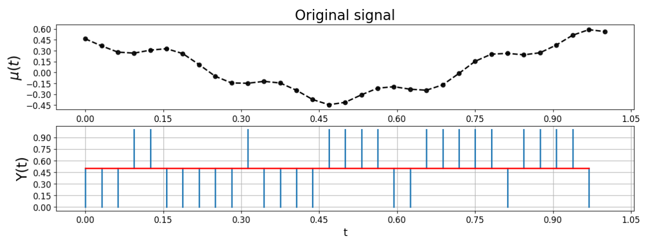


Fig. 1. Labelling of the generated data

The data generated is conformed by realizations of the Gaussian Process, each consisting of a chain of 20 samples with a random starting point. Figure 2 shows examples of the generated chains for an independent noisy kernel with the same variance for each sample $\sigma^2 = 0.03$.

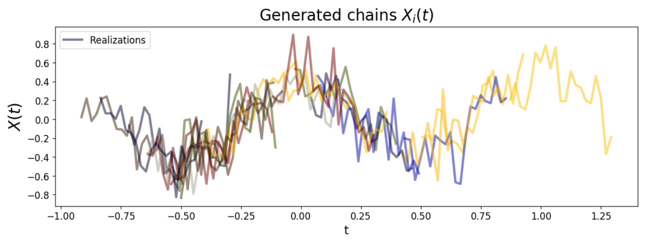


Fig. 2. Examples of the generated chains

5. RESULTS

In the two subsections below we state our results as they pertain to the next-word prediction task on the PTB data and on the classification task on simulated data. It's worth mentioning that the measure used in the following for e.g. test scores is the perplexity measure. Perplexity is the typical measure used when considering tasks such as next-word prediction. Perplexity is typically given as e^{loss}, where the loss is cross entropy, so the lower the perplexity the better.

5.1. PTB Data

As shown in Table 1, our implementation of the small model configuration achieved a per-word test perplexity of 95.6. This implementation outperformed the regular LSTM and LSTM dropout from [3], which achieved a per-word perplexity of 114.5 and 115.9, respectively. In contrast, however, our implementation of the medium model configuration underperformed relative to LSTM dropout from [3]. We obtained a per-word test perplexity of 85, while LSTM dropout obtained a score of 82.1. Our results did not replicate what was reported in [1], which obtained a per-word perplexity of 75.5. These results are summarized in Table 2. We discuss some of the reasons for this discrepancy in the Discussion section.

Table 1. Results on small architecture configuration. Our implementation outperforms Zaremba et al's dropout.

Model (Small)	**Validation**	**Test**
LSTM	120.7	114.5
LSTM dropout*	121.4	115.9
Our Implementation	98.9	95.6

Table 2. Results for medium architecture configuration. Our implementation underperforms compared to both Zaremba's and Fortunato's RNN implementation.

Model (Medium)	**Validation**	**Test**
LSTM dropout	86.2	82.1
Bayes By Backprop	78.8	75.5
Our Implementation	89	85

5.2. Simulated Data

In order to test the learning capabilities of the network over the artificial generated data, we ran some experiments using a rather small setting for the network. We used 1000 artificially generated chains for 10 epochs and 15 hidden neurons as a starting point. Figure 3 shows the prediction for a test chain where we show the noisy signal, the real targets and the predicted ones. As we can observe, the networks seems to learn from the data, having very little uncertainty for the first samples and more certainty in the center of clusters of the original target labels.

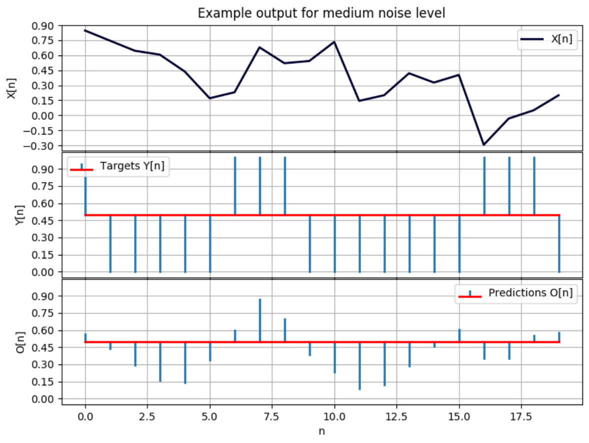


Fig. 3. Predictions for simulated data

The perplexity obtained for this configuration after tuning the hyperparameters reached 1.6, being 2.0 the maximum possible value and 1.0 the minimum. Increasing the noise level of the Gaussian process causes the perplexity to increase and the opposite happens when we decrease the noise, although no extensive study has been made in this regard.

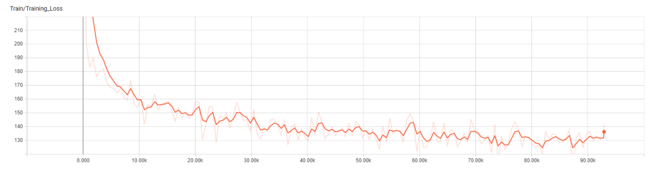
6. CONCLUSION & DISCUSSION

As summarized in Table 1 and Table 2, our implementation of Fortunato's Bayesian RNN (without Posterior Sharpening) using the small model configuration outperformed the regular as well as Zaremba's LSTM dropout in terms of per-word test perplexity. However, the same cannot be said of the medium model configuration. It is possible that we could not replicate Fortunato's medium model's per-word test perplexity due to a lack of fine-tuning of the model's hyperparameters. Due to the time constraints of the course project, we were not able to explore the model's hyperparameters fully.

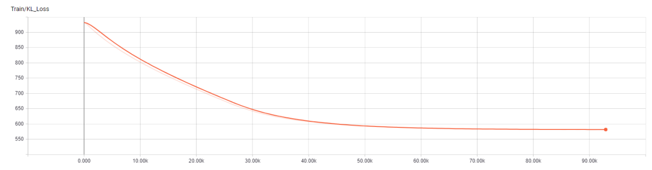
Another possible explanation, is one of slow convergence. When initializing the variance for the weights, this has to be done with care, making sure initial values are randomly selected within a certain range around the selected hyperparameter value. This aspect requires experimentation, and it's possible that given the timeframe of the project we didn't find an optimal range for the initial values.

It is interesting to note the magnitude of the two components of the total loss in figure 4. The total loss comprises the sum of the Likelihood loss and the KL Divergence loss. In the initial epochs during training of the model, it can be seen in figure 4 that the KL Divergence loss starts above 900 while the Likelihood loss starts above 200. It is evident that divergence between the variational posterior and the gaussian mixture prior imposed incurs a greater cost relative to the cost stemming from the likelihood loss. Further, figure 4 also shows how both losses stabilize around 35k steps. We can see this also in the weights' histograms in figures 5 and 6. We find that the temporal evolution of the Σ parameters is more dynamic compared to that of the μ parameters. Interestingly, we also find that the temporal evolution of the weights' μ (Fig. 5.a) for the LSTM cells is almost static. However, this is not the same case for the biases' μ.

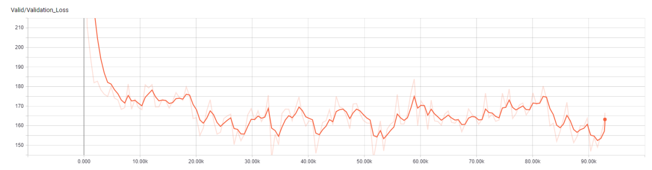
In closing, we feel confident that given more time we could've reproduced the results from [1] for the medium configuration as well through e.g. cross validation and further experimentation.



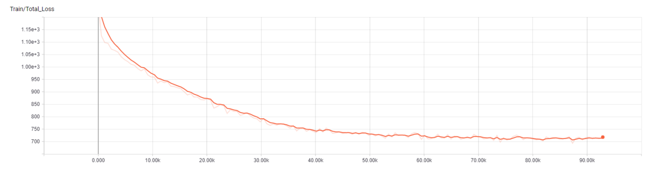
(a) Likelihood Loss



(b) KL Divergence Loss



(c) Validation Loss



(d) Total Loss

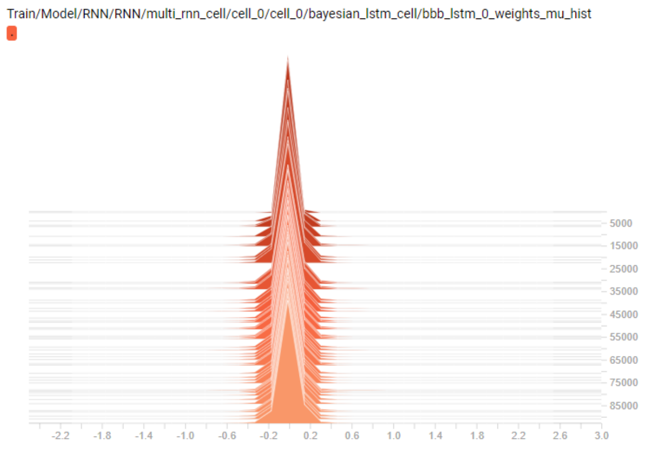
Fig. 4. The plots above show the relevant loss during training and testing time. Note the noisy Validation loss (c) due to a single Monte Carlo sampling after each minibatch.

7. REFERENCES

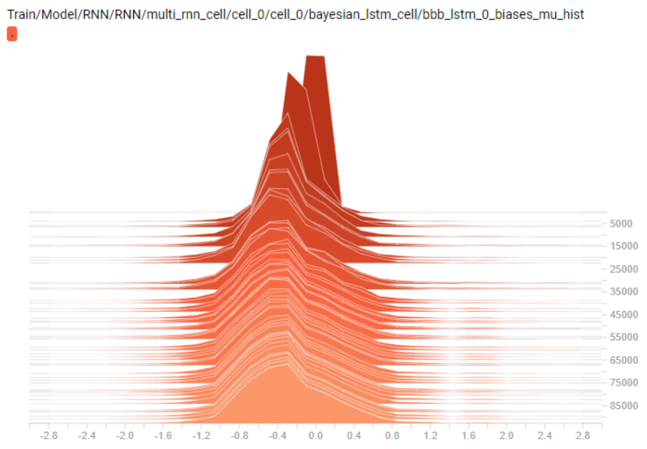
[1] Fortunato et al. (2017), *Bayesian Recurrent Neural Networks*

[2] Bayer et al. (2014), *On Fast Dropout and its Applicability to Recurrent Networks*

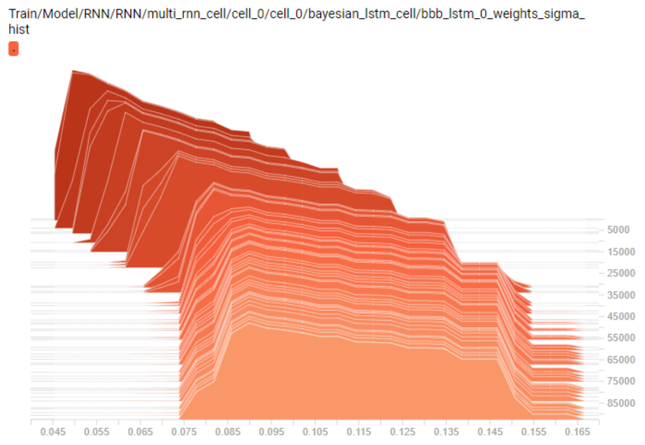
[3] Zaremba et al. (2015), *Recurrent Neural Network Regularization*



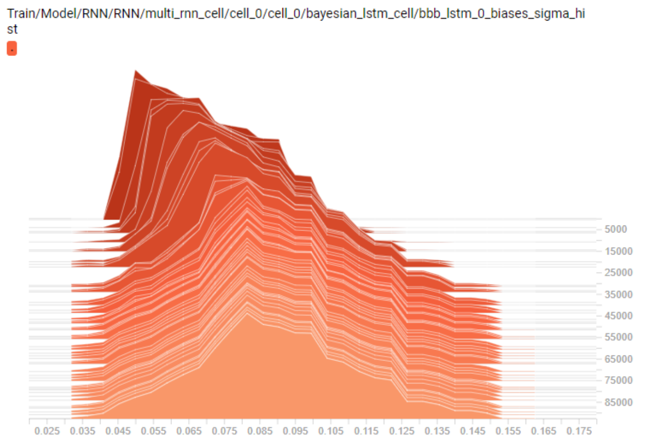
(a) LSTM cell W's (weights) μ histogram



(a) LSTM cell b's (biases) μ histogram



(b) LSTM cell W's (weights) Σ histogram



(b) LSTM cell b's (biases) Σ histogram

Fig. 5. The plots above show the histograms of how the weights of the LSTM cells evolved during training. It is of particular interest that the μ of the variational posterior over the W parameters in the LSTM cell remain almost static.

Fig. 6. The plots above show the histograms of how the biases of the LSTM cells evolved during training. Unlike the W parameters, the temporal evolution of μ in the biases is not as static.