A Batch-Normalized Recurrent Network for Sentiment Classification

Horia Margarit
horia@stanford.edu

Raghav Subramaniam
rsub@stanford.edu

Abstract

In this paper, we build a batch-normalized variant of the LSTM architecture in which each LSTM cell's input, hidden state, and cell state is normalized. The LSTM architecture allows for models with fewer vanishing gradients and therefore more memory than a vanilla recurrent network. Batch normalization lends a higher training speed to the model. We apply this architecture to a five-class sentence sentiment classification task with data from the Stanford Sentiment Treebank and show the effects of batch normalization.

1 Introduction

Our task is a five-class (very negative, negative, neutral, positive, very positive) sentence sentiment classification task with sentence data and labels from the Stanford Sentiment Treebank [6]. We use a Long Short-Term Memory (LSTM) [3] neural network architecture with batch normalization on the input, hidden states, and cell state of each LSTM cell, as in [2].

In order to adapt the batch-normalized LSTM (BN-LSTM) architecture to the sentiment classification task, we had to make a few changes. First, we converted one-hot word vectors into word vectors with lower dimensionality by using an embedding matrix. Next, the task described in [2] is classification on a sequential version of the MNIST dataset, so that the inputs to the LSTM have the same length. We had to employ padding and bucketing in order to allow for sentences of varying lengths to be inputs to the LSTM.

The Stanford Sentiment Treebank contains predefined train, dev, and test splits. We train our model on the sentences in the train split and a sampling of the labeled sub-sentence phrases in the training set, validate on the sentences in the dev split, and test on the sentences in the test split.

LSTMs are advantageous to vanilla recurrent networks because they can avoid the vanishing gradient problem, increasing their "memory". Batch normalization helps to increase the convergence speed of stochastic gradient descent (SGD) in training recurrent models [2].

We use the Tensorflow [1] library to build our model, numerically compute the backpropagation gradients, and run SGD.

2 Related Work

Hochreiter et al. introduced the LSTM architecture for sequential tasks in [3] and Ioeffe et al. introduced batch normalization for deep neural networks in [4]. Cooijmans et al. very recently introduced a variant of batch normalization for LSTM in [2]. The five-way sentence sentiment classification task and the Stanford Sentiment Treebank dataset are outlined in [6]. The state of the art for this task is achieved by Kumar et al. in [5].

29th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

3 Approach

3.1 LSTM

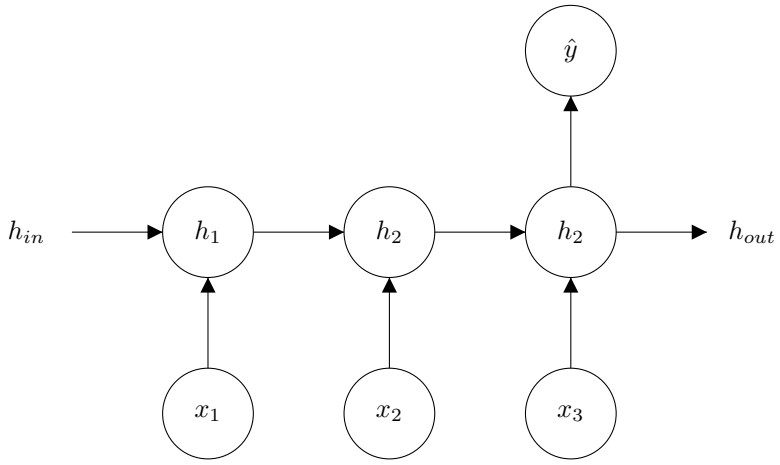

Figure 1: A many-to-one RNN architecture. We feed the hidden state of each RNN cell as an input into the next RNN cell. We use the output of the last RNN cell as in input into a softmax classifier.

A Recurrent Neural Network (RNN) is a deep neural network designed for sequential data. A vanilla RNN is a set of N RNN cells that map a set of inputs $x_1, \ldots, x_N$, where a subscript of i refers to the i^{th} input in the sequence or the i^{th} RNN cell, into a set of hidden states $h_1, \ldots, h_N$ via the update equations

$$\mathbf{h}_t = f(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b}) \tag{1}$$

where f is a nonlinearity (σ and $\tanh$ are common). From the hidden states, a softmax layer can be used to classify the inputs. A problem with vanilla RNNs is that of vanishing gradients, where the input at time t_1 has little effect on the hidden state at time t_2 with t_2 much greater than t_1, due to the effects of many iterated σs or $\tanh$s. The LSTM solves this problem. An LSTM is defined by the update equations

$$\begin{pmatrix} \tilde{\mathbf{f}}_t \\ \tilde{\mathbf{i}}_t \\ \tilde{\mathbf{o}}_t \\ \tilde{\mathbf{g}}_t \end{pmatrix} = \mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b} \tag{2}$$

$$\mathbf{c}_t = \sigma(\tilde{\mathbf{f}}_t) \odot \mathbf{c}_{t-1} + \sigma(\tilde{\mathbf{i}}_t) \odot \tanh(\tilde{\mathbf{h}}_t) \tag{3}$$

$$\mathbf{h}_t = \sigma(\tilde{\mathbf{o}}_t) \odot \tanh(\mathbf{c}_t) \tag{4}$$

The LSTM adds a set of cell states $c_1, \ldots, c_N$ to the model. Observe that, due to 3, the cell state for one LSTM cell, c_t, can be an affine function of the previous cells tate c_t. Since nonlinearities cause vanishing gradients, the LSTM can propagate cell states across a large number of LSTM cells. This makes an LSTM effective for tasks like sentence-level and paragraph-level sentiment detection, where a negation early on in a sentence can change the meaning of words later in the sentence.

We implemented a vanilla LSTM as a baseline and a point of comparison for the LSTM with batch normalization, described below.

3.2 LSTM with Batch Normalization

The phenomenon of covariate shift is known to reduce the training efficiency of deep neural networks [4]. Each batch of inputs to the model (and therefore other feed-forward activations) has different statistics that are not typically representative of the training data as a whole. The layers of the model must constantly adapt to these changing statistics, which leads to inefficient training. Batch normalization is a technique to reduce these effects and improve the training speed of the model.

$$BN(\mathbf{h}; \gamma, \beta) = \beta + \gamma \frac{\mathbf{h} - \widehat{\mathbb{E}}(\mathbf{h})}{\sqrt{\widehat{\mathrm{Var}}(\mathbf{h}) + \epsilon}} \tag{5}$$

We apply the batch normalization operation 5 to each input, hidden state, and cell state in the model. In the operation, β and γ are trainable parameters that represent the resulting population mean and variance of the outputs of the operation. A BN-LSTM is defined by the update equation.

$$\begin{pmatrix} \tilde{\mathbf{f}}_t \\ \tilde{\mathbf{i}}_t \\ \tilde{\mathbf{o}}_t \\ \tilde{\mathbf{g}}_t \end{pmatrix} = BN(\mathbf{W}_h \mathbf{h}_{t-1}; \gamma_h, \beta_h) + BN(\mathbf{W}_x \mathbf{x}_t; \gamma_x, \beta_x) + \mathbf{b} \tag{6}$$

$$\mathbf{c}_t = \sigma(\tilde{\mathbf{f}}_t) \odot \mathbf{c}_{t-1} + \sigma(\tilde{\mathbf{i}}_t) \odot \tanh(\tilde{\mathbf{h}}_t) \tag{7}$$

$$\mathbf{h}_t = \sigma(\tilde{\mathbf{o}}_t) \odot \tanh(BN(\mathbf{c}_t; \gamma_c, \beta_c)) \tag{8}$$

Unlike the LSTM weights and biases, the batch normalization statistics (e.g. $\widehat{\mathbb{E}}(\mathbf{h})$ and $\widehat{\mathrm{Var}}(\mathbf{h})$ are not shared across LSTM cells. Instead, we have separate statistics per cell. We fix β_h and β_x to 0 because these parameters are redundant with $\mathbf{b}$ [2].

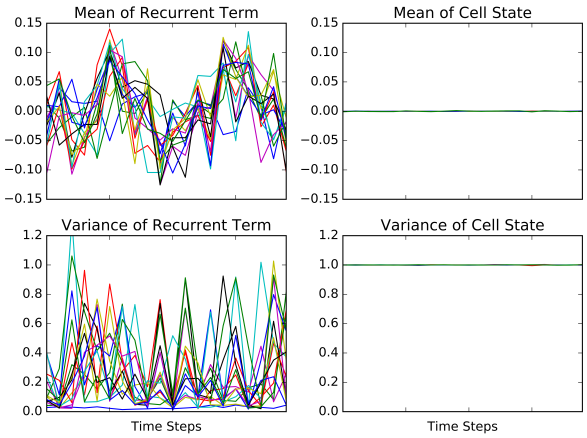

Figure 2: These plots show the effects of the batch normalization operation. The leftmost plots show the means and variance of some columns of c_1 while the rightmost plots show the means and variances of some columns of $BN(\mathbf{c}_1; \gamma_c, \beta_c)$ across inputs during the first few training iterations. γ was initialized to 1 and β was initialized to 0.

3.3 Padding and Bucketing

Inputs for sentiment analysis include phrases and words of varying lengths. We use padding and bucketing techniques to handle such inputs.

We first pick a bucket size ℓ_b. The bucket number $n_b(x)$ for a given input x of length $\ell(x)$ is given by $\ell(x)$ divided by ℓ_b (discarding the remainder). We next build an LSTM of length $N = \max_x \ell(x) \cdot \ell_b$.

We collect input batches $x_1, \ldots, x_M$ with $n_b(x_1) = \ldots = n_b(x_M)$, then pad them with `NAW` ("Not A Word") tokens so that the results each have length $n_b(x_1) \cdot \ell_b$. We provide these as inputs into the first $n_b(x_1) \cdot \ell_b$ LSTM cells, feeding zero inputs into the remaining LSTM cells. We only consider the LSTM hidden states for the $n_b(x_1) \cdot \ell_b^{\text{th}}$ cell in computing the softmax cross-entropy loss. This is an example of a many-to-one architecture.

For our training set, we had $N = 55$, so we used a bucket size of 5.

3.4 Other Considerations

We use a word embedding matrix to convert one-hot word vectors into word vectors with lower dimensionality. Sequences of these word vectors corresponding to input phrases and sentences are inputs to the BN-LSTM The output of the LSTM is fed into an affine layer followed by a softmax classifier with a cross-entropy loss function.

We observed exploding gradients while training the BN-LSTM so we clip our gradients to $[-1, 1]$. We use ℓ_2 regularization and 50% dropout (on the hidden states) to reduce overfitting.

4 Experiment

We used the approximately 9000 sentences in the training split of the Stanford Sentiment Treebank dataset and a 20% sampling of the labeled sub-sentence phrases with length greater than 5 words as the training data for our model. We validated (and tuned hyperparameters) in the dev split, and we tested on the sentences in the test split, we first trained a vanilla LSTM with the following hyperparameters:

Hyperparameter	**Value**
Bucket size	5
Batch size	200
Embed size	32
Hidden size	20
Learning rate	0.01
ℓ_2 regularization	0.03
Dropout	0.5

Figure 3: Hyperparameters for the vanilla LSTM.

This network resulted in a test accuracy of 0.3543. We next trained a BN-LSTM with the following hyperparameters:

Hyperparameter	**Value**
Bucket size	5
Batch size	200
Embed size	32
Hidden size	20
Learning rate	0.01
ℓ_2 regularization	0.04
Dropout	0.5

Figure 4: Hyperparameters for the BN-LSTM.

This network resulted in a test accuracy of 0.3170. The following plots show the training and validation loss, the training and validation accuracy, and the testing precision and recall:

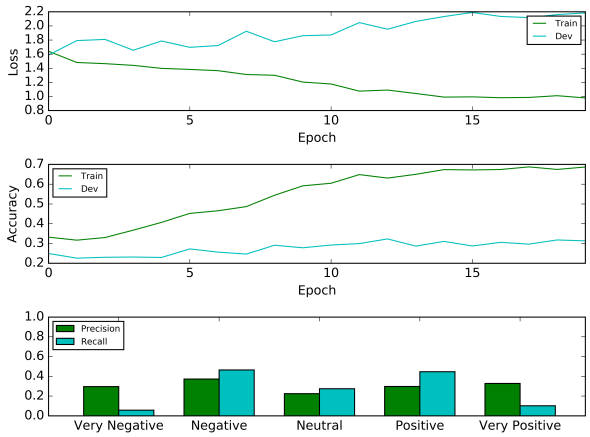

Figure 5: Results for the vanilla LSTM.

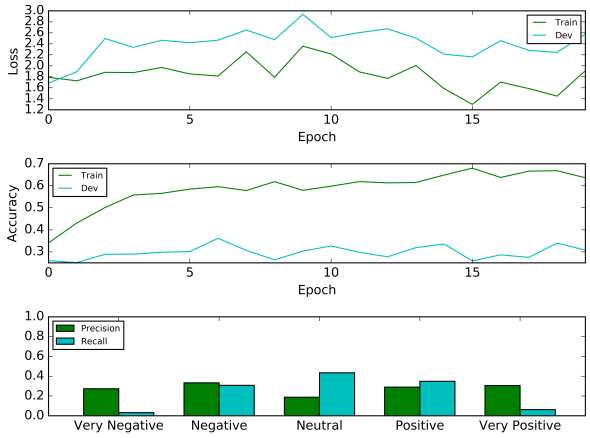

Figure 6: Results for the BN-LSTM.

4.1 Discussion

We were able to coarsely tune the hyperparameters for the vanilla LSTM. We can see from the loss plot that overfitting is still an issue. The hyperparameters for the BN-LSTM were set to be similar to those for the LSTM so that we can make comparisons between the two models. Again, overfitting is an issue. In the vanilla LSTM, SGD converges at around the 15^{th} epoch, after which the training and validation losses and accuracies stay fairly constant. We increased the regularization for the BN-LSTM, since it has a few more model parameters than the vanilla LSTM, and an increased number of model parameters can induce overfitting. In the BN-LSTM, it looks like SGD converges at around the 3^{rd} epoch, after which the training and validation accuracies stay fairly constant. The training and validation losses bounce around even after the accuracies converge. We're not sure why this is occurring. Next, because we only tuned hyperparameters for the vanilla LSTM, it has better testing performance than the BN-LSTM. In theory, batch normalization should not have too much of an effect on final accuracy.

From the precision and recall plots, we see that both architectures have low recall for the very negative and very positive classes. This implies that both classifiers are reluctant to classify inputs as either of the extreme classes. This could be in response to the unbalanced nature of the dataset; it has more neutral examples than examples belonging to the extreme classes.

4.2 Future Direction: Bidirectional LSTM

We additionally implemented a bidirectional LSTM with ℓ_2 regularization and an embedding matrix of word vectors that are shared by the forward and backward LSTMs. This implementation was motivated by the hypothesis that the sentiment classification could be improved by context words before and after the current input. However, this model failed to meet classify well; it consistently classified test examples as positive. We suspect the likely causes to be omitting the bucketing step and considering only the final hidden states of each of the LSTMs for softmax classification. Future work therefore includes rectifying these two points and re-evaluating the bidirectional LSTM with and without batch normalization.

5 Conclusion

We implemented and tuned a vanilla LSTM and implemented a batch-normalized LSTM (BN-LSTM), with the batch normalization variant from [2], for a 5-class sentence-level sentiment analysis task on data from the Stanford Sentiment Treebank. We were able to get a test accuracy of 35.4% with the vanilla LSTM, and we were able to show that batch normalization does indeed improve the training efficiency of the LSTM model. For comparison, the table below shows the performance of recent deep neural models on the sentiment analysis task, including the state-of-the-art DMN [5] architecture, which attains 52.1% test accuracy.

Model	Test Accuracy
MV-RNN	44.4
RNTN	45.7
DCNN	48.5
PVec	48.7
CNN-MC	47.4
DRNN	49.8
CT-LSTM	51.0
DMN	52.1
LSTM	**35.4**
BN-LSTM	**31.7**

Figure 7: Test accuracies for the 5-class sentiment classification task on the Stanford Sentiment Dataset [5]. Our results are shown in bold.

Looking at the state-of-the-art performance, it's evident that this is a difficult classification problem. We are not convinced that BN-LSTM will outperform vanilla LSTM to any noticeable degree. However, combined with other LSTM optimizations such as CT-LSTM [7], a tree-structured LSTM, batch normalization can lead to a near-state-of-the-art deep neural model for sentiment classification with a significantly higher training speed.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Tim Cooijmans, Nicolas Ballas, César Laurent, and Aaron Courville. Recurrent batch normalization. *CoRR*, abs/1603.09025, 2016.

[3] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[4] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[5] Ankit Kumar, Ozan Irsoy, Jonathan Su, James Bradbury, Robert English, Brian Pierce, Peter Ondruska, Ishaan Gulrajani, and Richard Socher. Ask me anything: Dynamic memory networks for natural language processing. *CoRR*, abs/1506.07285, 2015.

[6] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher Manning, Andrew Ng, and Christopher Potts. Parsing With Compositional Vector Grammars. In *EMNLP*. 2013.

[7] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *CoRR*, abs/1503.00075, 2015.