Final Project

Simas Glinskis

March 2, 2017

Abstract

Batch normalization was first introduced to prevent covariant shift of the hidden layers in deep recurrent neural networks (RNN), preserving the gradients in backpropagation. Here we report the inclusion of batch normalization layers in a long short term memory (LSTM) architecture, training on the Penn Tree Bank data set. The results show that the normalization does not improve the performance or training time of the LSTM architecture.

Introduction

Batch normalization was first introduced for deep RNNs to reduce internal covariant shift through the many layers of the network [1]. The original algorithm is quite simple and is broken down into two segments. Directly from the paper, they are:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

for the batch normalization layer, and

Input: Network N with trainable parameters Θ;
subset of activations $\{x^{(k)}\}_{k=1}^{K}$
Output: Batch-normalized network for inference, $N_{\text{BN}}^{\text{inf}}$
1: $N_{\text{BN}}^{\text{tr}} \leftarrow N$ // Training BN network
2: **for** $k = 1 \ldots K$ **do**
3: Add transformation $y^{(k)} = \text{BN}_{\gamma^{(k)},\beta^{(k)}}(x^{(k)})$ to $N_{\text{BN}}^{\text{tr}}$ (Alg. 1)
4: Modify each layer in $N_{\text{BN}}^{\text{tr}}$ with input $x^{(k)}$ to take $y^{(k)}$ instead
5: **end for**
6: Train $N_{\text{BN}}^{\text{tr}}$ to optimize the parameters $\Theta \cup \{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^{K}$
7: $N_{\text{BN}}^{\text{inf}} \leftarrow N_{\text{BN}}^{\text{tr}}$ // Inference BN network with frozen // parameters
8: **for** $k = 1 \ldots K$ **do**
9: // For clarity, $x \equiv x^{(k)}, \gamma \equiv \gamma^{(k)}, \mu_{\mathcal{B}} \equiv \mu_{\mathcal{B}}^{(k)}$, etc.
10: Process multiple training mini-batches $\mathcal{B}$, each of size m, and average over them:

$$\text{E}[x] \leftarrow \text{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$$
$$\text{Var}[x] \leftarrow \frac{m}{m-1}\text{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$$

11: In $N_{\text{BN}}^{\text{inf}}$, replace the transform $y = \text{BN}_{\gamma,\beta}(x)$ with $y = \frac{\gamma}{\sqrt{\text{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\text{E}[x]}{\sqrt{\text{Var}[x]+\epsilon}}\right)$
12: **end for**

Algorithm 2: Training a Batch-Normalized Network

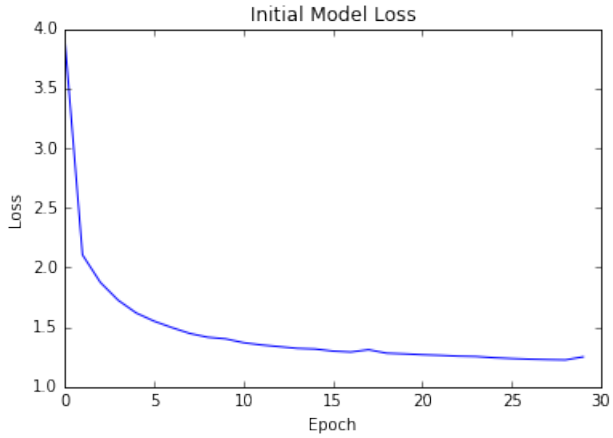
for implementing the layer during training and test time. It is important to note that during test time we use the stored variance and mean from training time, and weight due to sample statistics.

The goal of the experiment was to see if implementing a similar batch normalization layer would improve the accuracy and performance of an LSTM architecture trained on the Penn Tree Bank data set. The objective of the algorithm is to predict the next character within a sentence including the end of character, and output a sentence based on a initial kernel.

Initial Model

The LSTM is trained from a cold start to benchmark the performance and accuracy of batch normalization layers. In 30 epochs the algorithm is able to construct a sentence with appropriate end character, with a final log loss of 1.25168 and perplexity of 3.41982. The epochs average a training time of 17.2 minutes, and there seems to be an exponential decrease in the loss, shown in

this figure:



Normalizing the Carry

The initial idea, as given in the proposal, was to normalize only the carry.

$$C_{t+1} = BN(C_{t+1})$$

The naive theory here was that the inputs are assumed to be normalized which continuously feed into the network, so normalization of the hidden layers may not show any dramatic change. Unfortunately, after further thought and experimentation the theory is wrong

The carry is the one signal that cannot be normalized as it serves to preserve the gradients through the extended network. LSTM fundamentally requires that the carry remains unchanged outside the forget and diversion gates. This is also evident in practice, as backpropagation overflows while the gradients explode and the loss quickly converges to NaN before an epoch is complete. Various techniques such as gradient clipping and averaging did nothing to combat the overflow, and training continued to produce NaNs.

Normalizing Hidden Layers

The obvious next step was attempting to normalize the hidden layers of the network,

$$f = \sigma(BN(W_f[h, x]) + b_f)$$
$$i = \sigma(BN(W_i[h, x]) + b_i)$$
$$o = \sigma(BN(W_o[h, x]) + b_o)$$
$$\tilde{c} = \sigma(BN(W_c[h, x]) + b_c)$$

After training for 30 epochs, the loss exponentially decreased and then saturated to 1.86 while the perplexity decreased and then oscillated around values between 6.7 and 6.4. Inclusion of batch normalization for the hidden layers increased computation time per epoch to an average of 24 minutes, adding an additional 3.5 hours to the entire computation. Finally, the prediction did not have a form of a sentence and was unable to predict the end character.

It seemed like this was not the approach to take, as at test time variance and mean were averaged from all the time steps instead of uniquely for each step. The next iteration of the algorithm included a data structure to hold the average variance and mean during computation at each step to call at test time.

As the change only impacted the algorithm at test time, the training loss and perplexity exactly mirrored the original hidden layer normalization. However, computation time increased dramatically to 35 minutes per epoch, doubling the benchmark computation time. The inclusion of unique variance and mean per time step did not change the prediction at test time, and the final result was not a recognizable sentence and did not include the end character.

One final attempt with hidden layer normalization was to keep the sentences of roughly equal length. The data was folded by combining the first and last element of the data structure, as the data was stored in an ordered list of increasing length. The intuition was to have a more stable location for the end sentence character, as well as more uniform batches for normalization. However, again there was no noticeable change in the loss or perplexity, and the computation time did not change with the new data structure.

Momentum Averaging

An idea was brought up by a classmate to use momentum when computing the average mean and variance instead of a simple expectation value. With a $\beta = 0.9$,

$$\mu_{t+1} = \beta\mu_t + (1-\beta)\mu_B$$
$$\sigma^2_{t+1} = \beta\sigma^2_t + (1-\beta)\sigma^2_B$$

First, the network was trained maintaining the same hidden layer architecture described in the previous section, without any data structure to hold the variance and mean at each time step nor folding the data. In this configuration the loss decreased pseudo-linearly to a saturation of about 1.73 with a perplexity of 5.5. Each epoch took roughly 28 minutes, again much longer than without any normalization. The final prediction was not a recognizable sentence and without an end character.

Similar to the section above, both a time dependent mean and variance, and folded data structures were attempted for the momentum based hidden layer normalization.

Time dependent mean and variance seems to be the strongest candidate for batch normalization in an LSTM. After the first epoch, it was the only algorithm to maintain the characteristic prediction of the benchmark, where words

or phrases are repeated. For example, at epoch 0 the benchmark prediction printed:

the agreements bring the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the the

and the time dependent momentum averaging batch normalization printed:

the agreements bring the the the the the the the the the the the the the the the the the ther and the ther and the ther and the ther and the ther and the ther the the the the the the the the the the the the the the the the ther the the the ther and the ther the the ther ther ther ther ther ther ther ther ther ther ther ther ther ther ther ther therare thar the ther ther fore thare thar ther fore fofint

Unfortunately, after the first epoch there seemed to be an overflow in the gradient and the algorithm began producing NaNs. This architecture would be rigorously tested if there was more time for investigation, however with the increased training times of all the experiments it was not possible. The loss did decrease exponentially as in the benchmark case, with training at a rate of roughly 38 minutes an epoch.

Finally, removing the time dependent mean and variance and folding the data, the algorithm was able to train for a full 30 epochs without any overflows or NaNs down to a loss of 1.25297 and a perplexity of 3.48973, comparable to the benchmark. However, it did take an average of 50 minutes per epoch, almost a factor of 4 longer, and the final output was not a sentence nor was the end character predicted.

Conclusion

We attempted to include batch normalization into an LSTM architecture on the Penn Tree Bank data set, however no implementation surpassed the benchmark. Batch normalization prevents covariant shift in RNNs which maintains the gradients in deep networks, but there is no reason to believe that there is any measurable covariant shift in an LSTM. As the weights remain unchanged and the input data is normalized, the gradients remain in the active region during training. Any covariant shift is required for accurate prediction or the LSTM averages the inputs and the time dependent nature of the LSTM is lost.

Maintaining time dependent means and variances and or folding the data to keep the input sentences reasonably similar did not improve the performance of the algorithm, and only increased computation time. There was one promising case, but the gradients overflowed during training after the first epoch. This case requires further investigation, but as stated previously batch normalization is not required nor should produce any apparent improvements in accuracy or performance.

References

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