Let the Machines Learn *A blog on data science, machine learning and artificial intelligence.*

What makes the AWD-LSTM great?

Posted on September 12, 2018November 30, 2018 by Yashu Seth

The **AWD-LSTM** has been dominating the state-of-the-art **language modeling** (https://en.wikipedia.org/wiki/Language_model). All the top research papers on word-level models incorporate AWD-LSTMs. And it has shown great results on character-level models as well (**Source (http://nlpprogress.com/language_modeling.html)**).

In this blog post, I go through the research paper – **Regularizing and Optimizing LSTM Language Models** (https://arxiv.org/abs/1708.02182) that introduced the AWD-LSTM and try to explain the various techniques discussed in it. The paper investigates a set of **regularization** and **optimization** strategies for word-based language modeling tasks that are not only highly effective but which can also be used with no modification to existing LSTM implementations.

The Chinese version of the post, translated by **Jakukyo Friel (@weakish) (http://twitter.com/weakish)** can be found here – **语言建模的王者：AWD-LSTM指南** (https://www.jqr.com/article/000622)

The **AWD-LSTM** stands for **ASGD Weight-Dropped LSTM**. It uses **DropConnect** and a variant of **Average-SGD (NT-ASGD)** along with several other well-known regularization strategies. We will go through all these techniques in detail. While all these methods have been proposed and theoretically explained before, the beauty of this paper lies in their successful application to the language modeling task achieving state-of-the-art results.

The code for reproducing the results is open sourced and is available at the **awd-lstm-lm GitHub repository** (https://github.com/salesforce/awd-lstm-lm).

If you want to refresh your memory with the internal working of an LSTM network you should definitely check out this famous article – **Understanding LSTM Networks** (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)by **Christopher Olah** (http://colah.github.io/about.html).

The mathematical formulation of an LSTM can be expressed as –

$$
\begin{aligned}
\mathbf{i_t} &= \sigma(\mathbf{W^i x_t} + \mathbf{U^i h_{t-1}}) \\
\mathbf{f_t} &= \sigma(\mathbf{W^f x_t} + \mathbf{U^f h_{t-1}}) \\
\mathbf{o_t} &= \sigma(\mathbf{W^o x_t} + \mathbf{U^o h_{t-1}}) \\
\mathbf{c'_t} &= \tanh(\mathbf{W^c x_t} + \mathbf{U^c h_{t-1}}) \\
\mathbf{c_t} &= \mathbf{i_t} \odot \mathbf{c'_t} + \mathbf{f_t} \odot \mathbf{c'_{t-1}} \\
\mathbf{h_t} &= \mathbf{o_t} \odot \tanh(\mathbf{c_t})
\end{aligned}
$$

where, $\mathbf{W^i}$, $\mathbf{W^f}$, $\mathbf{W^o}$, $\mathbf{W^c}$, $\mathbf{U^i}$, $\mathbf{U^f}$, $\mathbf{U^o}$, $\mathbf{U^c}$ are weight matrices, $\mathbf{x_t}$ is the vector input to timestep $\mathbf{t}$, $\mathbf{h_t}$ is the current exposed hidden state, $\mathbf{c_t}$ is the memory cell state, and $\odot$ is element-wise multiplication.

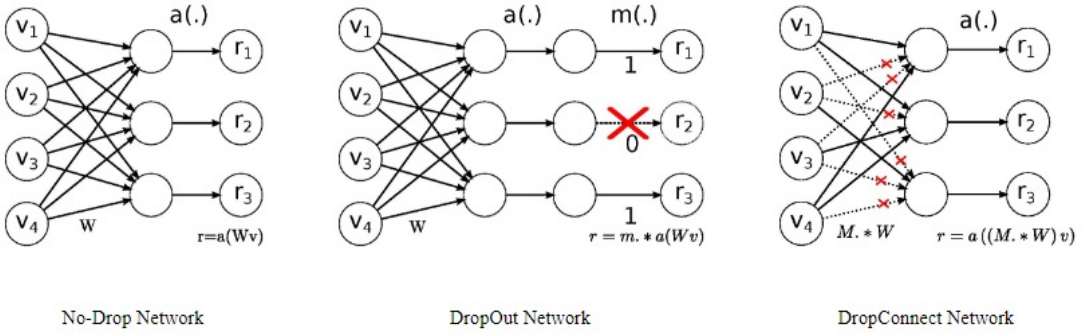
Let's go through each of the strategies proposed by the authors one by one.

Weight-dropped LSTM

The recurrent connections of an RNN have been prone to overfitting. Preventing this has been an area of great interest and extensive research. In this regard, Dropouts have been massively successful in feed-forward and convolutional neural networks. But applying dropouts similarly to an RNN's hidden state is ineffective as it disrupts the RNN's ability to retain long-term dependencies.

There have been various strategies to overcome this problem but the majority of previous techniques act either on the hidden state vector $\mathbf{h_{t-1}}$ or on the update to the memory state $\mathbf{c_t}$. This prevents the use of inflexible but highly optimized black box RNN implementations such as the **NVIDIA's cuDNN LSTM** (https://devblogs.nvidia.com/optimizing-recurrent-neural-networks-cudnn-5/).

To counter this problem, the authors propose the use of **DropConnect** (https://cs.nyu.edu/~wanli/dropc/). We know that in Dropout, a randomly selected subset of activations is set to zero within each layer. In **DropConnect**, instead of activations, a randomly selected subset of weights within the network is set to zero. Each unit thus receives input from a random subset of units in the previous layer.



Source – Regularization of Neural Networks using DropConnect (https://cs.nyu.edu/~wanli/dropc/)

The **DropConnect** is applied on the hidden to hidden weight matrices $\mathbf{(U^i, U^f, U^o, U^c)}$ instead of the hidden or memory states. Since this dropout operation is performed once, before the forward and backward pass the impact on training speed is minimal and any standard optimized black box RNN implementation can be used. By performing dropout on the hidden-to-hidden weight matrices, overfitting can be prevented on the recurrent connections of the LSTM.

You can have a look at the **weight_drop.py** (https://github.com/salesforce/awd-lstm-lm/blob/master/weight_drop.py) module from the official **awd-lstm-lm** (https://github.com/salesforce/awd-lstm-lm) Github repository for a detailed implementation.

The authors suggest that **DropConnect** can also be used on non-recurrent weights of the LSTM. Though the focus was on preventing overfitting on the recurrent connections.

Non-monotonically Triggered Average SGD

It has been found that for the particular task of language modeling, traditional SGD without momentum outperforms other algorithms such as momentum SGD, Adam, Adagrad, and RMSProp. Therefore, the authors investigate a variant of the traditional SGD algorithm known as **ASGD (Average SGD).**

Average SGD

Let's first go through the **ASGD** algorithm. It takes gradient update step identical to the SGD algorithm but instead of returning the weight computed in the current iteration, it also takes into account the weights of the previous iterations and returns an average.

Traditional SGD update –

```
w_t = w_prev - lr_t * grad(w_prev)
```

ASGD update –

```
avg_fact = 1 / max(t - K, 1)

if avg_fact != 1:
  w_t = avg_fact * (sum(w_prevs) + (w_prev - lr_t * grad(w_prev)))
else:
  w_t = w_prev - lr_t * grad(w_prev)
```

where,

K is the minimum number of iterations run before weight averaging starts. So before **K** iterations, the ASGD will behave similarly to a traditional SGD. **t** is the current number of iterations done, **sum(w_prevs)** is the sum of weights from iteration **K** to **t** and **lr_t** is the learning rate at iteration **t** decided by a learning rate scheduler.

You can find the PyTorch implementation of **AGSD here** (https://github.com/pytorch/pytorch/blob/cd9b27231b51633e76e28b6a34002ab83b0660fc/torch/optim/asgd.py).

The authors highlight two drawbacks of this method –

- Unclear tuning guidelines to the learning rate scheduler.
- Unclear guidelines on the value of the parameter **K**. A value too small can adversely affect the efficacy of the method and a value too large may take additional iterations to converge.

They propose a non-monotonically triggered variant of ASGD **(NT-ASGD)** in which-

- Averaging is triggered only when the validation metric fails to improve for multiple cycles. This is ensured by the non-monotone interval hyperparameter **n**. So whenever the validation metric does

not improve for **n** cycles the algorithm switches to use **ASGD**. The authors found that setting n=5 works well.
- A constant learning rate is used throughout the experiment and hence no further tuning is required.

The authors use this setting of **NT-ASGD** and thus demonstrate that it achieves better results as compared to SGD.

Extended Regularization Techniques

In addition to the two techniques discussed above, the authors use additional regularization techniques that prevent overfitting and improve data efficiency.

Variable Length Backpropagation Sequences

The authors highlight the inefficiency of using fixed length **backpropagation through time (BPTT)**. Consider, having 100 elements to perform backpropagation with a fixed **BPTT** window of **10**. In this case, any element divisible by **10** will not have any element to backprop into. This prevents 1/10 of the data to improve itself in a recurrent fashion and 8/10 of the data only uses partial BPTT window.

To counter this, the authors propose the use of variable length backpropagation sequences. They first select the base sequence length to be **bptt** with probability **p** and **bptt/2** with probability **1-p**. In the PyTorch implementation, the authors use p = 0.95.

```
base_bptt = bptt if np.random.random() < 0.95 else bptt / 2
```

Then, this **base_bptt** is used to get the **seq_len** using **N(base_bptt, s)** where **s** is the standard deviation and **N** is a normal distribution. This is suggested in the paper. Although, the PyTorch implementation has the following code for this step –

```
seq_len = max(5, int(np.random.normal(base_bptt, 5)))
```

The learning rate is also rescaled according to the **seq_length** being used. The rescaling step is necessary as sampling arbitrary sequence lengths with a fixed learning rate favors short sequences over longer ones.

```
lr2 = lr * seq_len / bptt
```

Variational Dropout

In a standard dropout, a new dropout mask is sampled each time the dropout connection is called. While in **Variational Dropout**, the dropout mask is sampled only once upon the first call and then that locked dropout mask is repeatedly used for all connections within the forward and backward pass.

While **DropConnect** is used rather than **Variational Dropout** to regularize the hidden-to-hidden transition within an RNN, **Variational Dropout** is used for all other dropout operations, specifically using the same dropout mask for all inputs and outputs of the LSTM within a given forward and backward pass. Each example within the mini-batch uses a unique dropout mask, rather than a single dropout mask being used over all examples, ensuring diversity in the elements dropped out.

The variational dropout implementation from the official **awd-lstm-lm** GitHub repository can be found **here (https://github.com/salesforce/awd-lstm-lm/blob/master/locked_dropout.py)**. For more details, please refer to the **original paper** (https://arxiv.org/abs/1512.05287).

Embedding Dropout

The authors employ Embedding Dropout which was first introduced in the paper – **A Theoretically Grounded Application of Dropout in Recurrent Neural Networks (https://arxiv.org/abs/1512.05287)**. It performs dropout on the embedding matrix at a word level and is used for a full forward and backward pass. **This results in the disappearance of all occurrences of a specific word within that pass.**

Reduction in Embedding Size

The easiest reduction in total parameters for a language model can be achieved by reducing the word vector size. Even though it does not help in reducing overfitting, the size of the embedding dimensions is reduced. The first and last LSTM layers are modified such that their input and output dimensions respectively are equal to the reduced embedding size.

Activation Regularization

L_2-regularization is often used on the weights to reduce overfitting. L_2 decay can also be used on the individual unit activations. This is termed activation regularization. AR penalizes activations that are significantly larger than 0 as a means of regularizing the network.

```
loss = loss + alpha * dropped_rnn_h.pow(2).mean()
```

Temporal Activation Regularization

In this, L_2 decay is applied to the difference in outputs of an RNN at different time steps. It penalizes the model from producing large changes in the hidden state.

```
loss = loss + beta * (rnn_h[1:] - rnn_h[:-1]).pow(2).mean()
```

Here, alpha and beta are scaling coefficients. **AR and TAR loss are only applied to the output of the final RNN layer.**

Model Analysis

The authors also experimented with different variants of the best performing model. They removed each of the strategies one at a time to see their impact independently.

Model	PTB Validation	PTB Test	WT2 Validation	WT2 Test
AWD-LSTM (tied)	60.0	57.3	68.6	65.8
– fine-tuning	60.7	58.8	69.1	66.0
– NT-ASGD	66.3	63.7	73.3	69.7
– variable sequence lengths	61.3	58.9	69.3	66.2
– embedding dropout	65.1	62.7	71.1	68.1
– weight decay	63.7	61.0	71.9	68.7
– AR/TAR	62.7	60.3	73.2	70.1
– full sized embedding	68.0	65.6	73.7	70.7
– weight-dropping	71.1	68.9	78.4	74.9

Each row in the above figure represents the effect on the perplexity score when that particular strategy is removed. This lets us compare the impact of the various strategies employed independently. For example, **the most extreme perplexity jump was in removing the hidden-to-hidden LSTM regularization provided by the weight-dropped LSTM (11 points)**.

Important Experiment Details

Data – The evaluation of the impact of these approaches was done on two datasets – **Penn Tree Bank (PTB) Dataset (https://corochann.com/penn-tree-bank-ptb-dataset-introduction-1456.html)** and **WikiText 2 Dataset (https://einstein.ai/research/the-wikitext-long-term-dependency-language-modeling-dataset)**.

Network Architecture – All experiments use a **three-layer LSTM** model with **1150** units in the hidden layer and an embedding of size **400**.

Batch Size – Batch size was 80 for WT2 and 40 for PTB dataset. **It was observed empirically that larger batch sizes (e.g., 40-80) outperformed smaller sizes (e.g., 10-20).**

Pointer Models – The authors also demonstrate the use of pointer based attention models along with these techniques. The **neural cache model** (https://arxiv.org/abs/1612.04426) further improves the perplexity score.

The details of the other hyperparameters can be referred from the paper. The hyperparameters were chosen on trial and error and further improvements may be possible if a fine-grained hyperparameter search is employed.

Conclusion