Language modeling using AWD-LSTM

Mátyás Vincze (229777)

University of Trento

matyas.vincze@studenti.unitn.it

1. Introduction (approx. 100 words)

This document is the report for the final project for the course Natural Language Understanding at the University of Trento. The main goal was to apply language modeling on the Penn Treebank dataset and experiment with different possible regularization techniques regarding recurrent neural networks. In our research, we implemented several regularization techniques from the ASGD Weight-Dropped LSTM (AWD-LSTM) model, which proved to be powerful in the field of language modeling before the appearance of the new state-of-the-art transformer model. The goal was to make a comparison on how much these methods contribute to the final result by making several runs, where each time we do not use one of the regularization methods.

2. Task Formalization (approx. 200 words)

Language modeling aims to assign probabilities to a sequence of words, hence trying to find the underlying probability distributions over all word sequences. These models exploit the statistical structure of a language to express the probability of a sequence: $\mathbb{P}(w_1, w_2, \ldots, w_N)$ and the probability of an upcoming word: $\mathbb{P}(w_k|w_1, w_2, \ldots, w_{k-1})$. To model the joint probability over a sequence of words w, we use the chain rule $\mathbb{P}(A \cap B) = \mathbb{P}(B|A) \cdot \mathbb{P}(A)$:

$$\mathbb{P}(w_1, w_2, w_3) = \mathbb{P}(w_1)\mathbb{P}(w_2|w_1)\mathbb{P}(w_3|w_1, w_2)$$

With the growth of the number of words, the conditional probability tables sooner or later are becoming intractable, this is where the Markov assumption comes into the picture suggesting that the probability of the next word only depends on the last or some small number of the previous words. N-gram models are based on N-order Markov models (assumptions). By far the most common choices are trigrams, bigrams and unigrams. We estimate the probabilities using co-occurence ratios. For example, in the bigram model:

$$\mathbb{P}(w_i|w_{i-1}) = \frac{\text{count}(w_{i-1}, w_i)}{\text{count}(w_{i-1})}$$

N-grams models have several issues such as the sparsity problem saying that if we did not see a given word of sentence before, we can not output any prediction. One solution would be to use a fixed-window neural language model which takes the context in a fixed size, represents the words as one-hot encoded (embedded) vectors, puts through a hidden layer and applies the softmax function at the end. This solves the sparsity problem, but the fixed-length input is a big limitation to the model.
To overcome this, recurrent neural networks (RNNs) are a natural choice for language modeling as they can take an input in any length, and they are ideal for sequential problems (when a point in a dataset is dependent on other points).
RNNs are capable of compressing all previous available information into a low dimensional space and compute probabilities for the possible next words using this latent representation of experience. Model size does not increase for longer input, same weights are applied on every time step. Their downside is that recurrent computation is slow and in practice it is difficult to access information from many steps back. To train these models, we use stochastic gradient descent, as the training data has to be big in order to achieve good results. The loss function used for training is cross-entropy, $-\sum_w y_w^{(t)} \log \hat{y}_w^{(t)}$ where $y^{(t)}$ is the true word (one-hot encoded) and $\hat{y}_w(t)$ is the predicted probability distribution over the corpus.

3. Data Description & Analysis (approx. 200-500 words)

For the experiments, the Penn Tree Bank dataset was used, featuring 2499 stories with over a million words of 1989 Wall Street Journal material. It contains train/valid/test sets with (929589, 73760, 82430) words respectively. Two special symbols in the dataset:
$<$ eos $>$ end of sequence
$<$ unk $>$ unknown word
The vocabulary length is ten thousand, with the most common words having over fifty thousand appearances each.

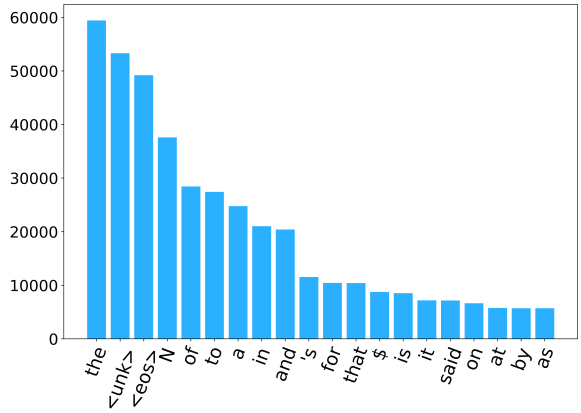

Figure 1: *20 most common words*

The dataset is originated from University of Pennsylvania, namely from Mitchell P. Marcus, Beatrice Santorini, Mary Ann Marcinkiewicz, Ann Taylor.

4. Model (approx. 200-500 words)

We used three layers of Long short-term memory (LSTM) modules as a base for our experiments. Quick mathematical formulation of an LSTM module:

$$
\begin{aligned}
i_t &= \sigma(h_{t-1}W^i + x_tU^i) \\
f_t &= \sigma(h_{t-1}W^f + x_tU^f) \\
o_t &= \sigma(h_{t-1}W^o + x_tU^o) \\
\hat{c}_t &= \tanh(h_{t-1}W^c + x_tU^c) \\
c_t &= i_t \odot \hat{c}_t + f_t \odot \hat{c}_{t-1} \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

where W is the recurrent connection between the previous hidden layer and the current one. U is the one that connects the input to the hidden layer. There are three gates controlling the memory of the cell, deciding how much past information to use, how much to remember and how much information should we pass to the next state: input, forget, output.

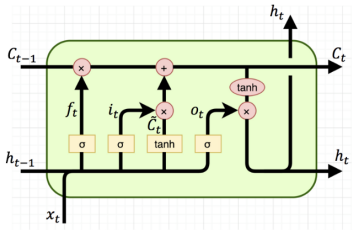

Figure 2: *LSTM module*

As neural networks can have millions and millions of parameters, dealing with overfitting has always been an important topic in the field. In the following, we will discuss several regularization features mainly from the AWD-LSTM model.

4.1. Weight dropout [1]

This regularization method is really similar to the normal dropout, but it differs in the fact that it is randomly dropping the weights rather than the activations with probability $1-p$. It does introduce dynamic sparsity within the model just as normal dropout, but the sparsity is in the weights W, instead of the output vectors of a layer. For a weight dropout layer, the output is given as:

$$\text{output} = \text{activation}((M * W) \cdot \text{input})$$

where W are weight parameters, and M is a binary matrix with $M_{ij} \sim \text{Bernoulli}(p)$. Each element of M is drawn independently for each input sample during training. Additionally, biases can be masked as well.

4.2. Embedding dropout [2]

When working with datasets with continuous inputs, it is common to apply standard dropout to the input layer. With discrete input datasets, this is rarely the case. When we are working with language models, we use word embedding, which maps each one-hot encoded word vector to the embedded space using the embedding matrix. Embedding dropout aims to regularize this typically huge layer, which is equivalent to randomly dropping input words. The remaining non-dropped-out word embedding are scaled by $\frac{1}{1-p_e}$, where p_e is the probability of the embedding dropout.

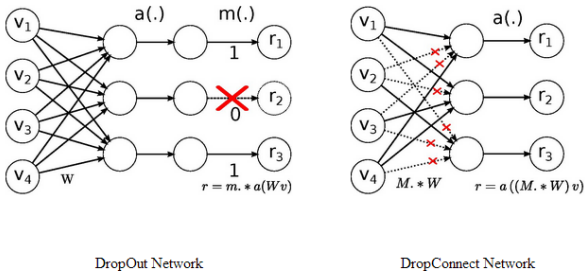

Figure 3: *Weight dropout*

4.3. Weight tying [3]

Improve performance and reduce the number of parameters by sharing weights of the embedding and softmax layer. In the embedding layer, we learn word representations, such that similar words (in meaning) are represented by vectors that are near each other. It has been shown in [3] that the softmax matrix, in which every word also has a vector representation, also exhibits this property.

4.4. Non-monotonically Triggered ASGD [4]

As the first steps of neural network training procedures strongly depend on the weight initialization method and the stochasticity in the model, [4] suggests using a combination of normal stochastic gradient (SGD) update and averaged stochastic gradient update (ASGD). First we use the standard SGD, until certain time has not passed (controlled by hyperparameter log_interval, n) and validation result did not improve. If both criteria are met, then we change SGD to ASGD and use it for the rest of the training.

Algorithm 1 Non-monotonically Triggered ASGD (NT-ASGD)

Inputs: Initial point w_0, learning rate γ, logging interval L, non-monotone interval n.

Initialize $k \leftarrow 0$, $t \leftarrow 0$, $T \leftarrow 0$, `logs` $\leftarrow$ `[]`
while stopping criterion not met **do**
  Compute stochastic gradient $\hat{\nabla} f(w_k)$ and take SGD step (1).
  if $\texttt{mod}(k, L) = 0$ and $T = 0$ **then**
    Compute validation perplexity v.
    if $t > n$ and $v > \min_{l \in \{t-n, \cdots, t\}}$ `logs[l]` **then**
      Set $T \leftarrow k$
    end if
    Append v to `logs`
    $t \leftarrow t + 1$
  end if
end while
return $\frac{\sum_{i=T}^{k} w_i}{(k-T+1)}$

Figure 4: *Non-monotonically Triggered ASGD*

4.5. Variational (lock) dropout [2]

We repeat the same dropout mask at each time step for both inputs, outputs, and recurrent layers (drop the same network

units at each time step). Whereas in normal/standard dropout, we use a different dropout mask for each time step for inputs and output alone.

4.6. Gradient clipping

Cliping computed gradients into a certain range $[-\text{clip\_gradient}, \text{clip\_gradient}]$ to control the maximum effect of a single gradient update and tackle the common exploding gradients problem of neural networks and RNNs.

- *your network/ algorithm (do not spend too much text in explaining already existing models, focus on your solution),*
- *the pipeline if used any, including tokenizer, featurizer, extractor, etc.*
- *Your baseline and the experiments you have tried*

5. Evaluation (approx. 400-800 words)

During the training, we used the previously mentioned cross-entropy loss. The main metric to measure how well our model is doing was perplexity, which is the normalized inverse probability of the dataset we are evaluating on.

$$\text{Perplexity}(\mathbf{w}) = \sqrt[N]{\frac{1}{\mathbb{P}(w_1, w_2, \ldots, w_N)}}$$

Intuitively, we want our model not to be surprised by the sentences seeing in the test set, therefore having a good understanding of how the language works. The reason why perplexity uses normalization is that longer text would bring more uncertainty, which is not what we want, and normalization helps us create a per-word measure. Since in the equation inverse probability is used, lower values correspond to better models. We can alternatively define perplexity using cross-entropy

$$\text{Perplexity}(\mathbf{w}) = 2^{-\frac{1}{N}\log_2 \mathbb{P}(w_1, w_2, \ldots, w_N)}$$

We measured perplexity during training after each epoch on the validation set and saved our model each time we achieved a new personal best result.

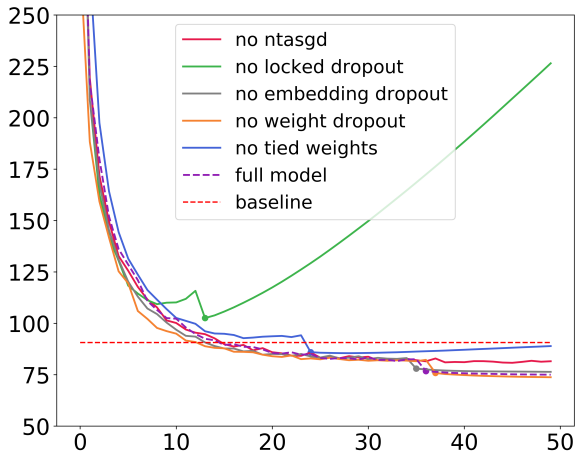

Figure 5: *Validation perplexity*

For each setting, we trained for 50 epochs and did not change any of the other hyperparameters except one of the regularization ones (setting it to zero), which is differentiating one setting from another. Training batch size was set to 20, validation and test to 10 each. The backpropagation through time window size (bptt) was 70, embedding size 400 and hidden size 1150. Number of LSTM layers was 3, learning rate 30 and the ntasgd parameter n to 5 and log_interval to 100. Gradient clipping was 0.25, lock dropout for input, hidden, output (0.4,0.25,0.4) respectively. Embedding dropout probability was 0.1, while for the weight dropout it was 0.5.
At the end of the 50 epoch training, we measured again on the test set. We repeated this procedure on the full model using all regularization techniques described above, and repeat with some regularization techniques being left out to measure how much influence they have on the final result.

model version	test perplexity
no weight dropout	71.22
no tied weights	77.85
no embedding dropout	73.40
no locked dropout	110.72
no ntasgd	75.57
all regularization	71.56
all regularization 100 epochs	70.66
baseline	90.70

Table 1: *Test perplexity for all models*

The results on the validation set aims to show the strength of each regularization technique. Gradient clipping has been applied each time as without it no convergence happens during training because of the exploding gradients. In all cases, except when leaving locked dropout out, we still achieve better perplexity on the test set than the baseline. One interesting observation would be that assigning zero probability to the weight dropout actually resulted in a better result than using all regularization techniques at once. This is possibly due to stochasticity in the model, and only shows that weight drop does not play a significant role in our model when all the other regularization techniques are being used.
These results clearly show the need of regularization, and the different power of each of them. Weight dropout seems to be really important to achieve results with good generalization capabilities, and all the other methods help to improve the model performance even further. Moreover, techniques such as weight tying significantly decreases the number of parameters, resulting in a more efficient and easier training procedure.

6. Conclusion

In this project we successfully achieved better result than the baseline with a big margin and showed how much each regularization technique has to do with that.
Next step would be to implement the rest of the regularization methods from the AWD-LSTM model, namely activation regularization and temporal activation regularization. The first one adds a term to the loss function using L_2 norm on the activations (instead of the weights) $\alpha L_2(m \circ h_t)$ while the second one penalizes differences between states that have been explored in the past $\beta L_2(h_t - h_{t-1})$.

7. References

[1] [Online]. Available: https://cds.nyu.edu/projects/regularization-neural-networks-using-dropconnect/

[2] Y. Gal and Z. Ghahramani, "A theoretically grounded application of dropout in recurrent neural networks," 2015. [Online]. Available: https://arxiv.org/abs/1512.05287

[3] O. Press and L. Wolf, "Using the output embedding to improve language models," 2016. [Online]. Available: https://arxiv.org/abs/1608.05859

[4] S. Merity, N. S. Keskar, and R. Socher, "Regularizing and optimizing lstm language models," 2017. [Online]. Available: https://arxiv.org/abs/1708.02182