RNN LANGUAGE MODELLING
Experiments on training parameters

Wim Boes & Robbe Van Rompaey

October 22, 2016

Contents

1 Introduction

1.1 Setup of experiments

The purpose of these experiments is to investigate the impact of the different training parameters involved when using recurrent neural networks for language modelling. To this end, the Tensorflow tutorial model of a recurrent neural network for language modelling (publicly available) is used with the Penn Tree Bank dataset for training, validation and testing of the model [1].

The Tensorflow tutorial language model consists of different parts. The input words, represented by word IDs (integers) unique to each word, are first embedded into multidimensional representations. These are then sequentially fed into a recurrent neural network that consists of layers of LSTM cells [2]. The outputs of this neural network are then turned into a probability vector by a softmax classifier. Each element of this vector is the probability that the next word is the word whose word ID is the index of the element in this vector.

This language model uses the Penn Tree Bank (PTB) dataset, a popular benchmark for language models [3]. This dataset has a vocabulary of 10000 words and is split into train, validation and test sets. It is a relatively small dataset. Words that fall out of this 10000 words vocabulary are set to the ¡unk¿-word and get all by consequence the same word ID.

1.2 Explanation of training parameters

This section contains an overview of the different training parameters that will be referenced throughout the rest of this document. These training parameters can be changed in the Tensorflow tutorial language model (Python) files. They do not refer to the actual parameters of the model, such as the weights of the neurons in the neural networks.

1.2.1 init_scale

All weights of the language model (the weights of the neurons of the neural network, the coefficients of the logistic regression used in the softmax function, etc.) are initialized uniformly in the range [**-init_scale**, **init_scale**].

1.2.2 learning_rate

The parameter **learning_rate** indicates the starting learning rate of the optimizing algorithm (see also section 1.2.13).

1.2.3 max_grad_norm

The gradients calculated by the BackPropagation Through Time algorithm are clipped at a threshold, namely **max_grad_norm**, to prevent the exploding gradients problem [6, 7].

1.2.4 num_layers

As mentioned in section 1.1, the neural network consists of layers of LSTM cells. The parameter **num_layers** indicates how many layers present in the model.

1.2.5 num_steps

The gradients used during training (by the optimizing algorithms) are calculated by the BackPropagation Through Time algorithm, in which the the recurrent neural network is unfolded and regarded as a deep network [6]. The parameter **num_steps** indicates the number of unfolded layers used.

1.2.6 hidden_size

This parameter indicates the number of hidden neurons in the input gate layers, output gate layers and forget gate layers of the LSTM cells [2].

1.2.7 max_epoch

This is the number of training epochs without learning rate decay. For example, if **max_epoch** is 4 and **max_max_epoch** (see section 1.2.8) is 8, the first 4 epochs the optimizer will use the initial learning rate and the last 4 epochs learning rate decay will be applied (see section 1.2.10).

1.2.8 max_max_epoch

This parameter indicates the total number of training epochs. This explanation holds as long as early stopping is not applied, as is the case in most of the experiments. If early stopping was used, **max_max_epoch** would be the maximum number of training epochs.

1.2.9 keep_prob

The used neural network uses dropout to prevent overfitting to the training data [4]. **keep_prob** is the probability of keeping the outputs of each layer (applied to each element of the output separately, not to the output as a whole). The kept outputs are adjusted with a correction factor to conserve the energy in the connection.

1.2.10 lr_decay

As described in section 1.2.7, after **max_epoch** epochs learning rate decay is applied. This means that after each epoch, the learning rate is multiplied by the factor **lr_decay**.

1.2.11 batch_size

batch_size is the number of training examples per batch. One training example contains **num_steps** words (see section 1.2.5). After each batch the parameters are updated, so that these are updated mutiple times during one epoch.

1.2.12 embedded_size

As mentioned in section 1.1, the word IDs (integers) are transformed into multidimensional (real-valued) representations called the word embeddings. The number of dimensions used is indicated by the parameter **embedded_size**.

1.2.13 optimizer

This parameter indicates which optimizing algorithm (available in the Tensorflow package) is used to train the model (by minimizing the loss function, see section 1.2.14) [5]. The possible values of **optimizer** are the following:

- *"GradDesc"*: gradient descent algorithm.
- *"Adadelta"*: Adadelta algorithm, does not use learning rate parameters **learning_rate**, **lr_decay** and **max_epoch**.
- *"Adagrad"*: Adagrad algorithm.
- *"Momentum"*: Momentum algorithm, with a fixed momentum term of 0.33 (arbitrary value).
- *"Adam"*: Adam algorithm, does not use learning rate parameters **learning_rate**, **lr_decay** and **max_epoch**.

1.2.14 loss_function

This parameter indicates which loss function is used by the optimizer (see section 1.2.13). There are two options for **loss_function**:

- *"sequence_loss_by_example"*: average negative log probability of the target words:

$$\text{loss} = -\frac{1}{N}\sum_{i=1}^{N} \ln p_{target_i}$$

  with p_{target_i} the probability indicated by the output of the language model (see section 1.1) on the position of the target word and N the length of the considered text piece (**num_steps**). This corresponds to the Shannon-McMillan-Breimann approximation of the cross-entropy of the language model [8].

- *"sampled_softmax"*: same loss function as above with some modifications. For the above loss function, for each target word the probability of every word in the vocabulary needs to be computed to obtain the properly normalized probability p_{target_i}. For vocabularies that are large, this is impractical. Hence, this loss function samples the vocabulary (32 samples in this case) to compute an approximation of the properly normalized p_{target_i}, this speeds up training. This loss function can only be used during training, not during inference (because the probability is not properly normalized) [9].

1.3 Performed experiments

In this section, the list of performed experiments is given. In each experiment, one or more parameters are variable while the rest of the parameters stay fixed. The next list indicates the variable parameters of each experiment:

- **hidden_size & embedded_size**
- **init_scale & learning_rate**
- **num_steps & batch_size**
- **loss_function**
- **optimizer**
- **num_layers**

The names of the subsections (one subsection for each experiment) in section 2 will refer to these variable parameters.

2 Results and conclusions of experiments

This section contains the results and conclusions of the different experiments as described in section 1.3. As mentioned there, the names of the subsections in this section refer to the variable parameters of the experiment. Each figure shows the different values of the variable parameters as well as the values of the fixed parameters.

Some of the figures display the test/validation perplexity as a function of the number of training epochs. The (average-per-word) test/validation perplexity is equal to the exponential value of the average negative log probability (cross-entropy) of the target (test/validation) words:

$$e^{-\frac{1}{N}\sum_{i=1}^{N} \ln p_{target_i}}$$

Some of the figures display a bar chart containing the training speed in function of the values of the variable parameters. This speed is expressed in words per seconds.

2.1 hidden_size & embedded_size

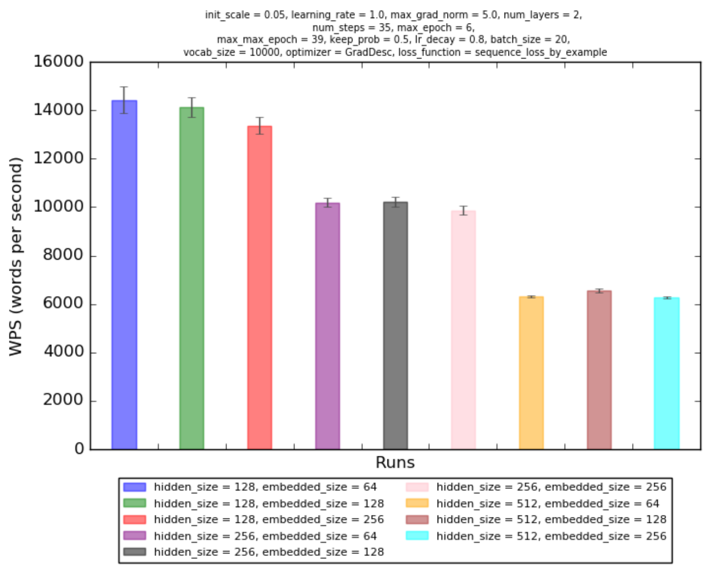

Figure 1: Results of speed experiment with **hidden_size & embedded_size** as variable parameters

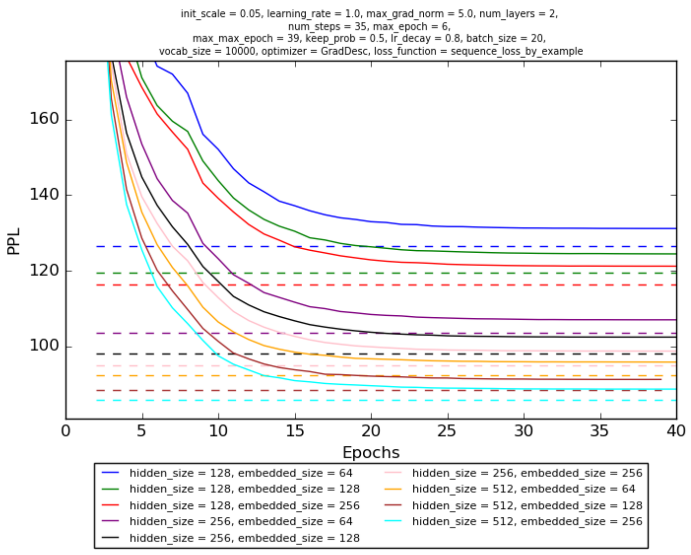

Figure 2: Results of performance experiment with **hidden_size & embedded_size** as variable parameters. The full line represents the validation perplexity evaluated after each training epoch, the dotted line represent the test perplexity evaluated at the vary last epoch of the training.

The results of this experiment are displayed in figures 1 and 2. The results are mostly as expected: higher values for **hidden_size** generally lead to better performance and slower training speed and higher values for **embedded_size** lead to better performance (though less impactful than **hidden_size**). However, the parameter **embedded_size** does not seem to have a big impact (if any at all) on the training speed. This is interesting as it might suggest that increasing **embedded_size** only has positive consequences (at least while the parameter is kept within a certain range).

2.2 init_scale & learning_rate

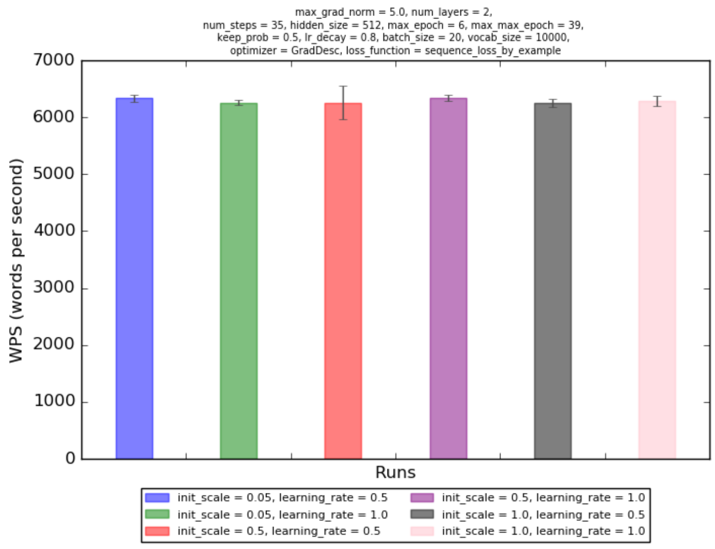

Figure 3: Results of speed experiment with **init_scale & learning_rate** as variable parameters

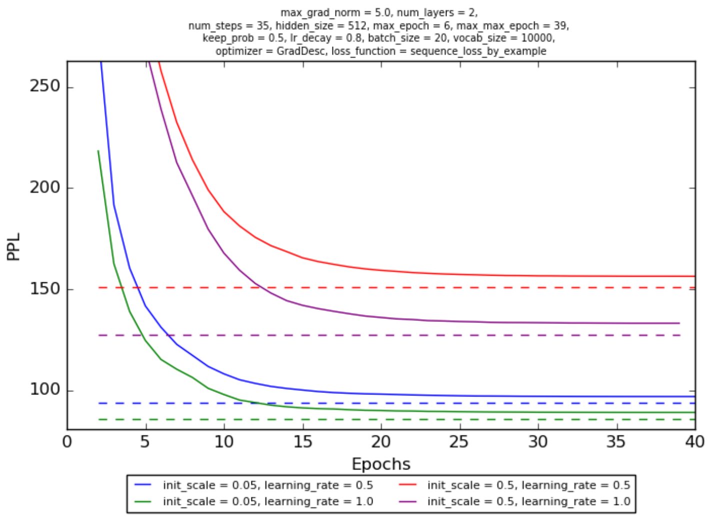

Figure 4: Results of performance experiment with **init_scale & learning_rate** as variable parameters. The full line represents the validation perplexity evaluated after each training epoch, the dotted line represent the test perplexity evaluated at the vary last epoch of the training.

The results of this experiment are displayed in figures 3 and 4. Experiments with **init_scale** equal to 1 were also performed but the performance results were so bad that these experiments are excluded from the figures. The results seem to suggest that it is beneficial to keep **init_scale** fairly small and **learning_rate** relatively large. The impact of the former also seems more significant than the impact of the latter.

2.3 num_steps & batch_size

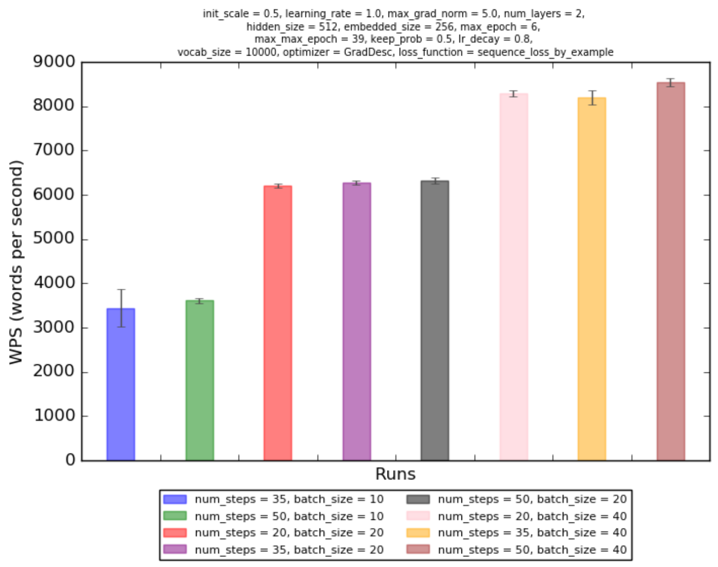

Figure 5: Results of speed experiment with **num_steps & batch_size** as variable parameters

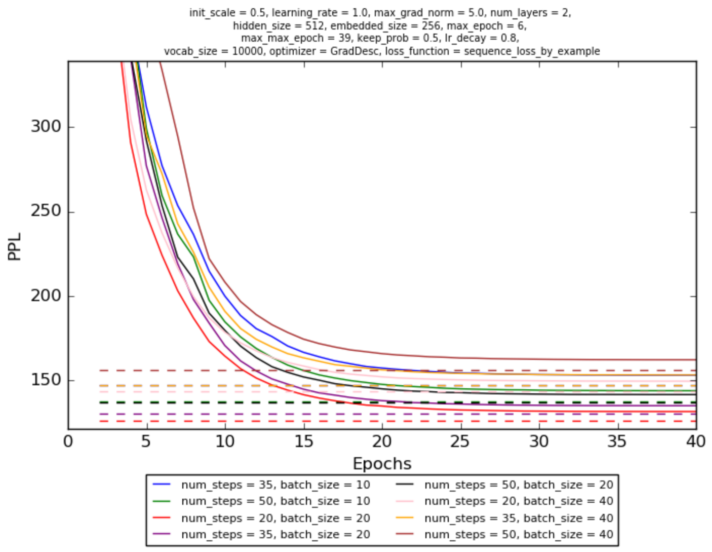

Figure 6: Results of performance experiment with **num_steps & batch_size** as variable parameters. The full line represents the validation perplexity evaluated after each training epoch, the dotted line represent the test perplexity evaluated at the vary last epoch of the training.

The results of this experiment are displayed in figures 5 and 6. As expected, as **batch_size** goes up, the training speed slows down while there seems to be very little impact of **num_steps** on the training speed. When it comes to the performance the results are much less clear: the best performing value for **batch_size** is 20, in between 10 and 40, there seems to be a local optimum for this parameter. Regarding the parameter **num_steps**, no definitive conclusion can be made based on this experiment.

2.4 loss_function

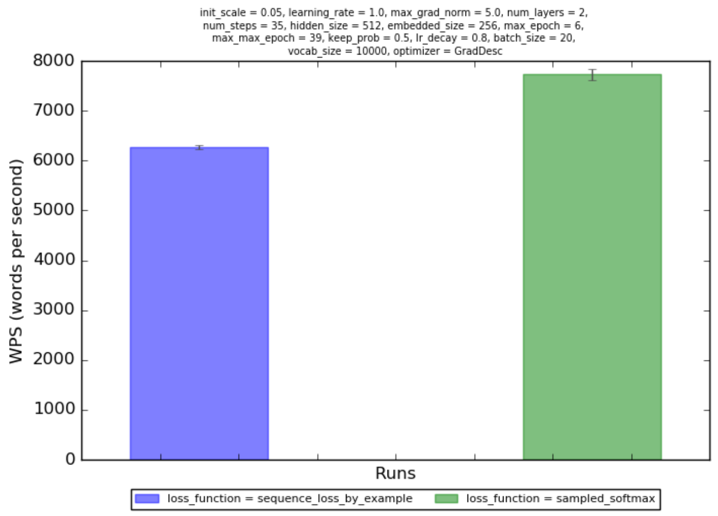

Figure 7: Results of speed experiment with **loss_function** as variable parameters

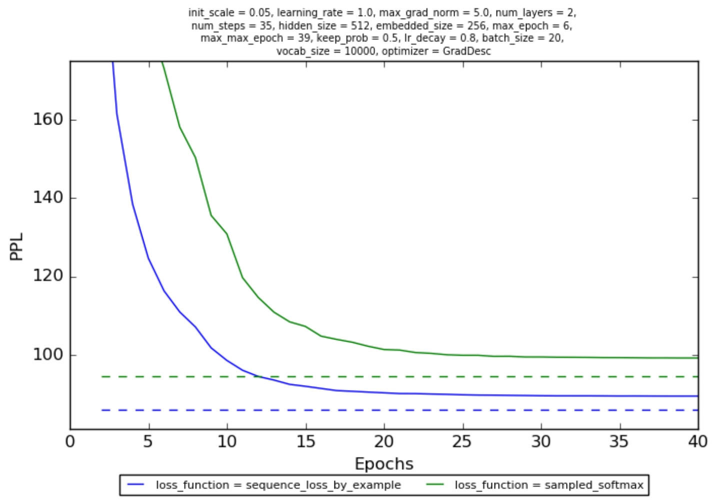

Figure 8: Results of performance experiment with **loss_function** as variable parameters. The full line represents the validation perplexity evaluated after each training epoch, the dotted line represent the test perplexity evaluated at the vary last epoch of the training.

The results of this experiment are displayed in figures 7 and 8. The results are as expected, the sampled softmax loss function trains quicker but performs worse than the full softmax function.

2.5 optimizer

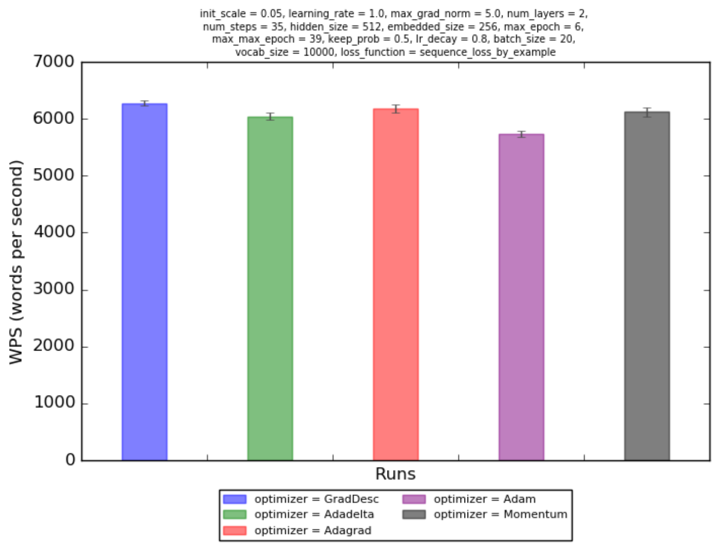

Figure 9: Results of speed experiment with **optimizer** as variable parameters

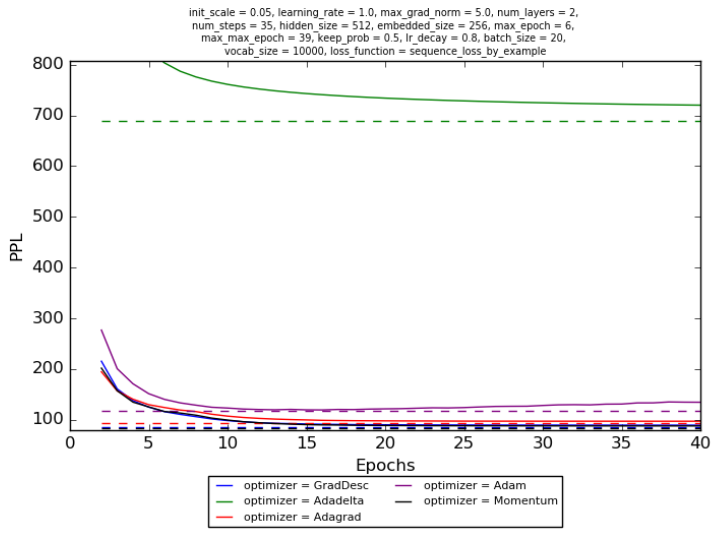

Figure 10: Results of performance experiment with **optimizer** as variable parameters. The full line represents the validation perplexity evaluated after each training epoch, the dotted line represent the test perplexity evaluated at the vary last epoch of the training.

The results of this experiment are displayed in figures 9 and 10. There seems be one clear loser in this experiment: the Adadelta optimizing algorithm. The rest of the optimizing algorithms are close in terms of speed and performance. One special thing to note is that the Adam algorithm seems to significantly suffer from the fact that early stopping is not applied.

2.6 num_layers

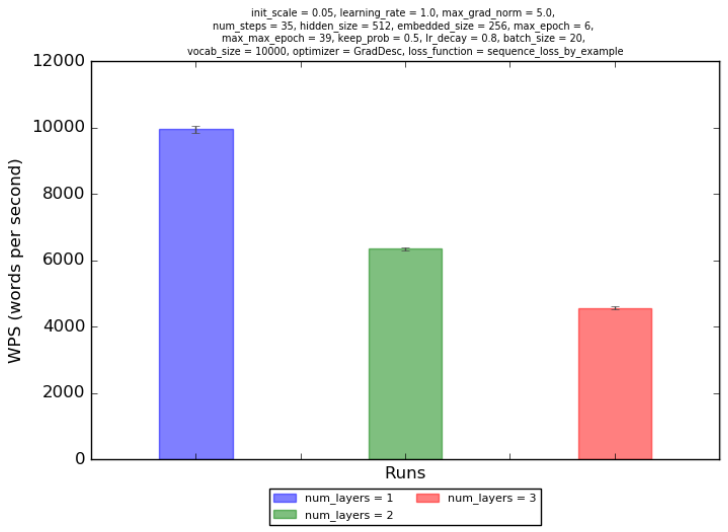

Figure 11: Results of speed experiment with **num_layers** as variable parameters

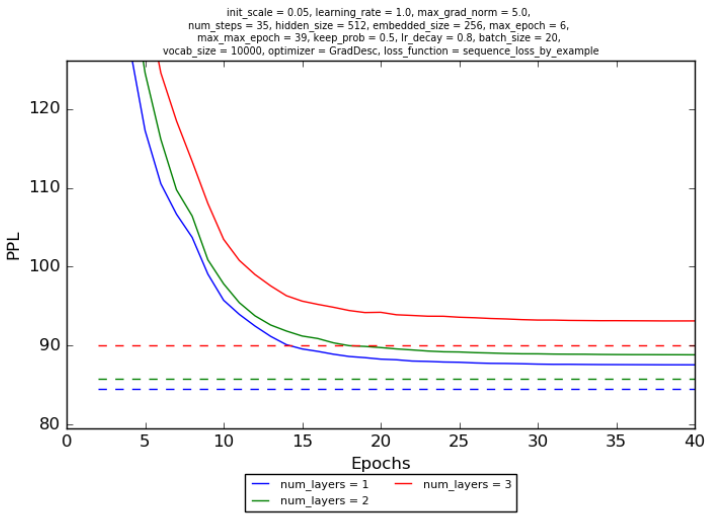

Figure 12: Results of performance experiment with **num_layers** as variable parameters. The full line represents the validation perplexity evaluated after each training epoch, the dotted line represent the test perplexity evaluated at the vary last epoch of the training.

The results of this experiment are displayed in figures 11 and 12. The results seem to indicate that in this case it is better to stay away from multiple layers of LSTM cells. This could be because the number of model parameters (weights) increase heavily as **num_layers** is increased, and there might not be enough training data available to cope with huge amounts of parameters.

3 Conclusion

These experiments investigated the impact of the different training parameters involved when using recurrent neural networks for language modelling. To this end, the Tensorflow tutorial model of a recurrent neural network for language modelling (publicly available) was used with the Penn Tree Bank dataset for training, validation and testing of the model [1].

The most interesting conclusions that could be drawn from these experiments are summarized in this list (see section 1.2 for an explanation of the parameters):

- Increasing **hidden_size** improves performance but lowers training speed.
- Increasing **embedded_size** improves performance slightly but does not have an impact or very little impact on training speed.
- **init_scale** should be kept at a relatively low value.
- **learning_rate** should be kept at a relatively high value.
- The parameter **batch_size** seems to have a local optimum regarding performance.
- No real conclusions could be drawn from these experiments regarding **num_steps**.
- The sampled softmax loss function has worse performance but better training speed than the full softmax loss function.
- Most optimizing algorithms are relatively close in performance besides Adadelta, that performs much worse than the other options.
- The optimal value for **num_layers** in this case is 1. However, this could be because there is not enough training data to properly train the model when multiple LSTM layers are employed.

References

[1] *https://www.tensorflow.org/versions/r0.11/tutorials/recurrent/index.html*

[2] *http://colah.github.io/posts/2015-08-Understanding-LSTMs/*

[3] *http://www.cis.upenn.edu/ treebank/home.html*

[4] *https://www.cs.toronto.edu/ hinton/absps/JMLRdropout.pdf*

[5] *https://www.tensorflow.org/versions/r0.11/api_docs/python/train.html*

[6] *http://ir.hit.edu.cn/ jguo/docs/notes/bptt.pdf*

[7] *http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/*

[8] *http://homes.esat.kuleuven.be/ jpeleman/wim_en_robbe/jurafsky04.pdf*

[9] *https://arxiv.org/pdf/1412.2007v2.pdf*