LSTM LANGUAGE MODELING
Experiments on training parameters

Wim Boes & Robbe Van Rompaey

November 14, 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 modeling. To this end, the TensorFlow tutorial model of a recurrent neural network for language modeling (publicly available) is used with the Penn Treebank 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 Treebank (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. Out-of-vocabulary words are mapped to a special <unk>-token that is also a part of the vocabulary.

The principles and architecture of the model will be explained in more detail in the final thesis.

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.

The parameters mentioned in this section will be explained in more detail in the final thesis.

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 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.

1.2.3 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 are present in the model.

1.2.4 num_steps

The gradients used during training (by the optimization algorithms) are calculated by the back-propagation 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.

1.2.5 hidden_size

This parameter indicates the number of neurons in each hidden layer.

1.2.6 embedded_size

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

1.2.7 learning_rate

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

1.2.8 lr_decay

As described in section 1.2.9, after **max_epoch** epochs exponential learning rate decay is applied with factor **lr_decay**.

1.2.9 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 1.2.10) 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 1.2.8).

1.2.10 max_max_epoch

This parameter indicates the maximum number of training epochs. Early stopping is applied in the following way: if the current validation perplexity is not lower than the highest validation perplexity of the previous three epochs, training is stopped.

1.2.11 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].

1.2.12 batch_size

batch_size is the number of training examples per batch. One training example contains **num_steps** words (see section 1.2.4). After each batch the parameters are updated, so that these are updated multiple times during one epoch.

1.2.13 optimizer

This parameter indicates which optimization 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.
- *"Adagrad"*: Adagrad algorithm.
- *"Momentum"*: Momentum algorithm, with a fixed momentum term of 0.33 (arbitrary value).
- *"Adam"*: Adam algorithm

For *"Adadelta"*, *"Adagrad"* and *"Adam"*, the default hyperparameters in the TensorFlow package are chosen.

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**:

- *"full_softmax"*: cross-entropy using full softmax:

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

  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].

2 Results and conclusions of experiments

This section contains the results and conclusions of the different experiments as described in section 2.1. 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:

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

The tables include the training speed in function of the values of the variable parameters. This speed is expressed in processed words per seconds. All experiments are performed on the same machine so the training rate of the different test runs can be compared.

2.1 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 fixed parameters are based on the medium configuration of the TensorFlow tutorial model. Each network is also initialized in the same way (by setting the seed of the random initializer). The next list indicates the variable parameters of each experiment:

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

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

2.2 learning_rate & lr_decay

Table 1: Last training, validation and test perplexities and the training speed with **learning_rate & lr_decay** as variable parameters.

Name	Train PPL	Valid PPL	Test PPL	Average speed
learning_rate = 0.1, lr_decay = 0.2	214.4731	209.3648	204.0305	5316.4381
learning_rate = 0.5, lr_decay = 0.2	90.1018	106.3778	103.5632	5311.6258
learning_rate = 1.0, lr_decay = 0.2	78.1643	96.0926	92.7933	5324.2165
learning_rate = 0.1, lr_decay = 0.8	160.0688	163.6912	159.6735	5324.8893
learning_rate = 0.5, lr_decay = 0.8	66.5606	96.0881	93.0609	5308.7686
learning_rate = 1.0, lr_decay = 0.8	56.8053	88.9418	85.3886	5316.8231
learning_rate = 0.1, lr_decay = 1.0	84.6244	118.6662	117.5150	5319.1841
learning_rate = 0.5, lr_decay = 1.0	60.9704	97.5769	95.6626	5317.9425
learning_rate = 1.0, lr_decay = 1.0	64.2170	93.3527	91.0134	5313.8851

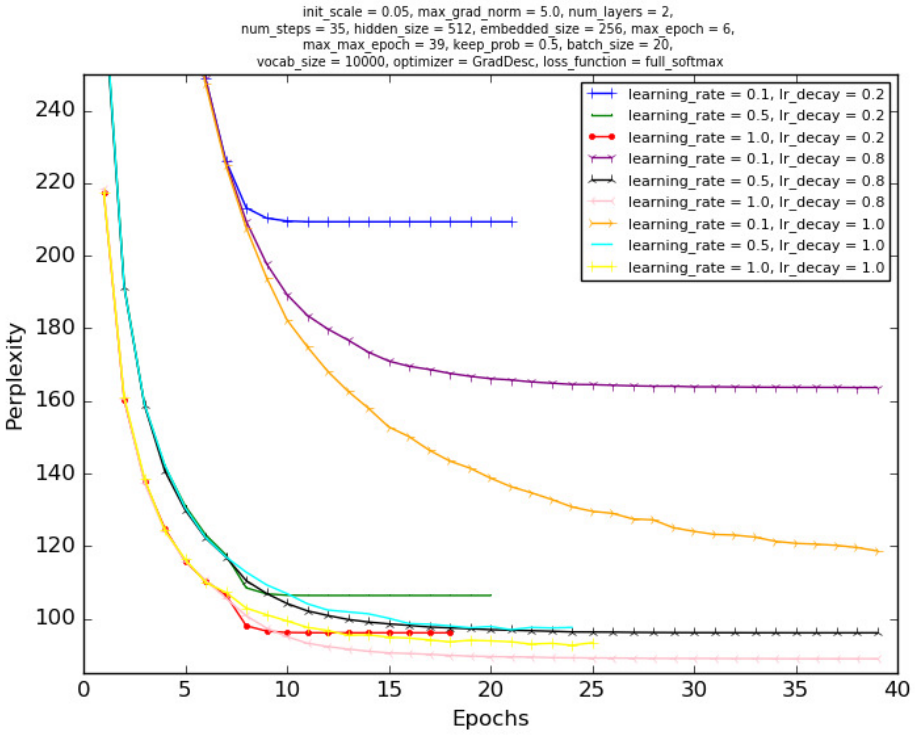

Figure 1: Convergence plot of the validation perplexity of the experiment with **learning_rate & lr_decay** as variable parameters.

The results of this experiment are displayed in table 1 and figure 1. The most apparent conclusion of this experiment is the fact that the learning rate should not be too small. The tests with **learning_rate** equal to 0.1 perform the worst. This is because the learning rate is too low and exponential learning rate decay is applied too early. This is clearly visible in the figure, especially when **lr_decay** is large. The performance of the other tests are much better, in this case there seems to be a local optimum for **learning_rate** equal to 1 and **lr_decay** equal to 0.8.

2.3 init_scale & hidden_size

Table 2: Last training, validation and test perplexities and the training speed with **init_scale & hidden_size** as variable parameters.

Name	Train PPL	Valid PPL	Test PPL	Average speed
init_scale = 0.005, hidden_size = 128	125.4301	122.0181	116.7535	11288.7535
init_scale = 0.005, hidden_size = 512	58.1604	90.2953	86.3675	5298.5033
init_scale = 0.05, hidden_size = 128	124.6025	120.4653	115.8137	11433.1415
init_scale = 0.05, hidden_size = 512	56.9165	89.3015	85.8495	5305.4898
init_scale = 0.5, hidden_size = 128	155.3053	139.7519	133.8476	11216.0070
init_scale = 0.5, hidden_size = 512	139.2114	130.7206	126.0680	5318.1586

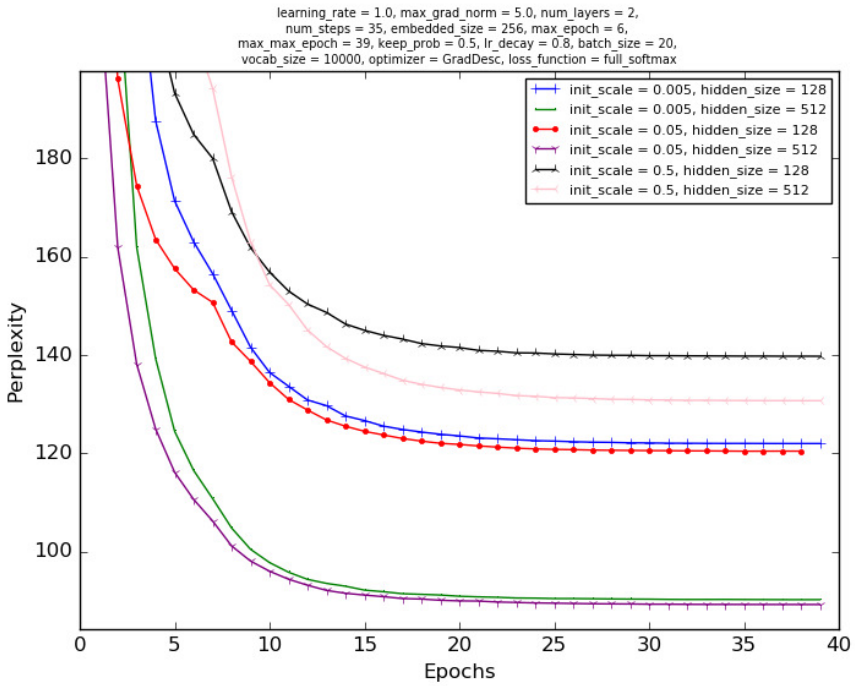

Figure 2: Convergence plot of the validation perplexity of the experiment with **init_scale & hidden_size** as variable parameters.

The results of this experiment are displayed in table 2 and figure 2. In this case, bigger values for **hidden_size** lead to better validation perplexities. 0.05 and 0.005 seem to be good values for **init_scale**, while 0.5 leads to a significantly worse performance. This seems to imply that the weights of the network should be initialized at rather small values.

The parameter **hidden_size** naturally influences the training speed. As the value for this parameter goes up, the training slows down. **init_scale** has no effect on the training speed.

2.4 hidden_size & embedded_size

Table 3: Last training, validation and test perplexities and the training speed with **hidden_size & embedded_size** as variable parameters.

Name	Train PPL	Valid PPL	Test PPL	Average speed
hidden_size = 128, embedded_size = 64	137.1526	129.8869	125.0642	11173.8044
hidden_size = 128, embedded_size = 128	130.6858	125.0439	120.6486	11027.2286
hidden_size = 128, embedded_size = 256	124.8371	120.8721	116.2822	11281.3936
hidden_size = 256, embedded_size = 64	96.5572	106.1885	102.8703	8874.8156
hidden_size = 256, embedded_size = 128	90.5044	101.9240	98.3112	8584.0692
hidden_size = 256, embedded_size = 256	85.3554	98.3592	94.8026	8698.9466
hidden_size = 512, embedded_size = 64	66.7117	94.9333	92.1504	5268.9036
hidden_size = 512, embedded_size = 128	60.8136	91.5243	87.9890	5461.2367
hidden_size = 512, embedded_size = 256	56.8551	89.6501	86.0614	5311.4149

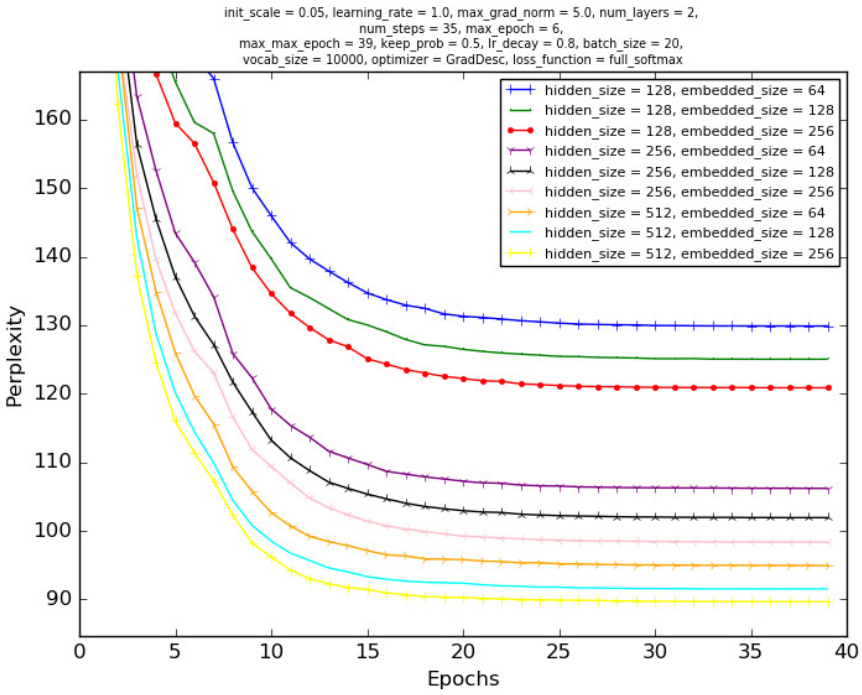

Figure 3: Convergence plot of the validation perplexity of the experiment with **hidden_size & embedded_size** as variable parameters.

The results of this experiment are displayed in table 3 and figure 3. 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).

2.5 max_grad_norm

Table 4: Last training, validation and test perplexities and the training speed with **max_grad_norm** as variable parameter.

Name	Train PPL	Valid PPL	Test PPL	Average speed
max_grad_norm = 1.0	103.9493	120.8001	118.1674	5322.4864
max_grad_norm = 5.0	56.8391	89.2325	85.9046	5309.6957
max_grad_norm = 10.0	61.9130	90.0071	86.5227	5328.3798

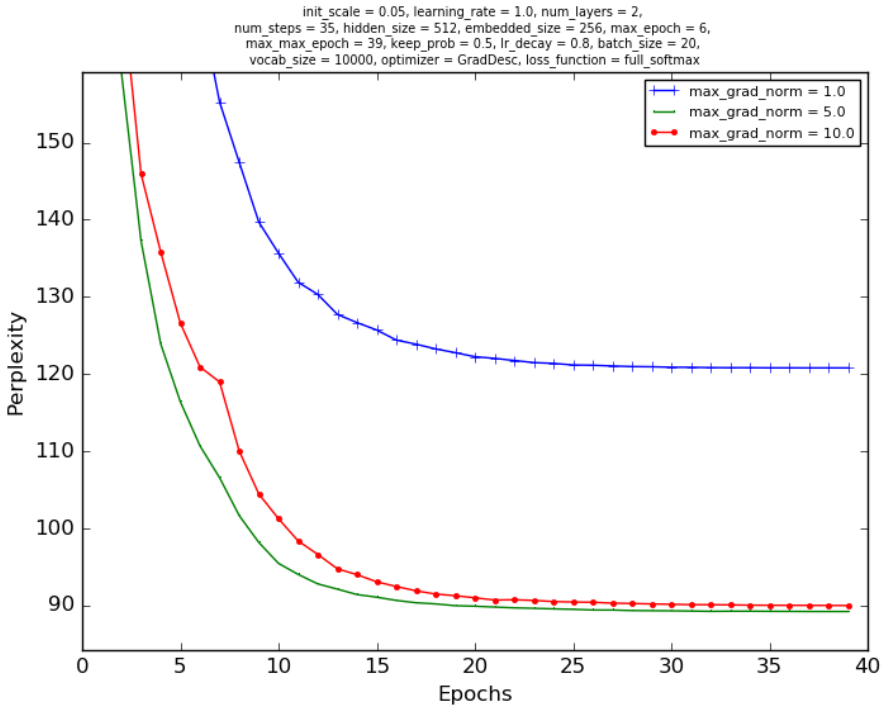

Figure 4: Convergence plot of the validation perplexity of the experiment with **max_grad_norm** as variable parameter.

The results of this experiment are displayed in table 4 and figure 4. Choosing **max_grad_norm** equal to 1 is clearly not a good idea: the clipping of the gradients is too strong and the changes to the weights of the network are too small. This leads to a bad validation perplexity. For the other two cases, there does not seem to be much difference in performance.

This parameter has no impact on the training speed.

2.6 num_layers

Table 5: Last training, validation and test perplexities and the training speed with **num_layers** as variable parameter.

Name	Train PPL	Valid PPL	Test PPL	Average speed
num_layers = 1	52.6574	87.7519	84.3744	8781.3653
num_layers = 2	56.4768	88.8830	85.7698	5318.5357
num_layers = 3	61.5084	93.3361	90.2382	3842.1562

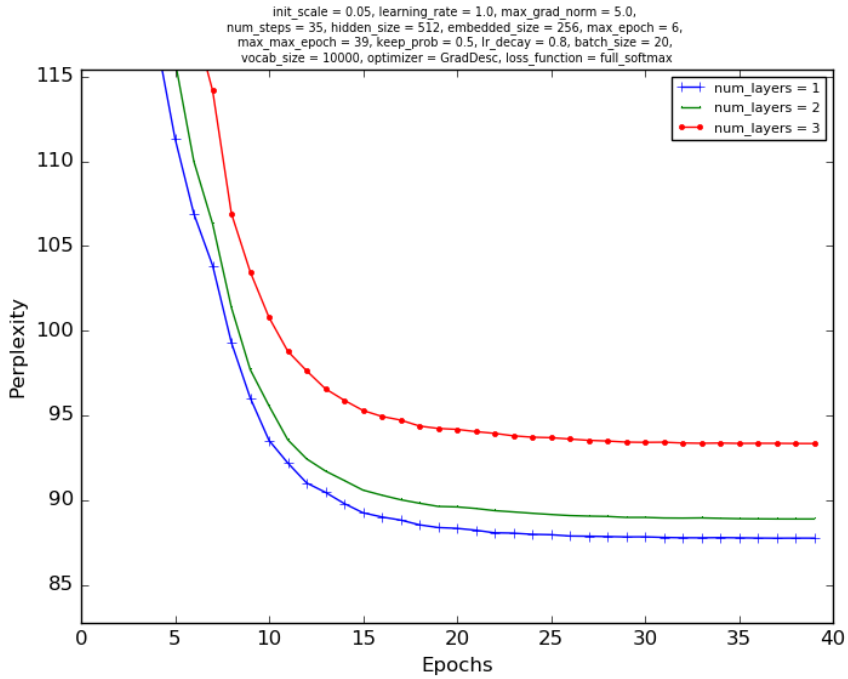

Figure 5: Convergence plot of the validation perplexity of the experiment with **num_layers** as variable parameter.

The results of this experiment are displayed in table 5 and figure 5. 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.

As expected, the training speed goes down significantly when the number of layers is increased.

2.7 num_steps & batch_size

Table 6: Last training, validation and test perplexities and the training speed with **num_steps & batch_size** as variable parameters.

Name	Train PPL	Valid PPL	Test PPL	Average speed
num_steps = 20, batch_size = 10	63.8183	91.3286	87.3871	2841.4514
num_steps = 35, batch_size = 10	57.4925	87.9364	84.7437	2967.6752
num_steps = 50, batch_size = 10	55.6267	87.5726	84.1104	3032.4830
num_steps = 20, batch_size = 20	57.7876	89.4243	85.4032	5283.1615
num_steps = 35, batch_size = 20	56.8357	89.0865	86.1965	5327.4472
num_steps = 50, batch_size = 20	57.4257	89.8376	86.9127	5340.5170
num_steps = 20, batch_size = 40	58.0128	91.2085	87.4406	7340.3399
num_steps = 35, batch_size = 40	59.9359	92.4088	89.0932	7222.8050
num_steps = 50, batch_size = 40	63.9817	95.3053	91.6211	7685.0879

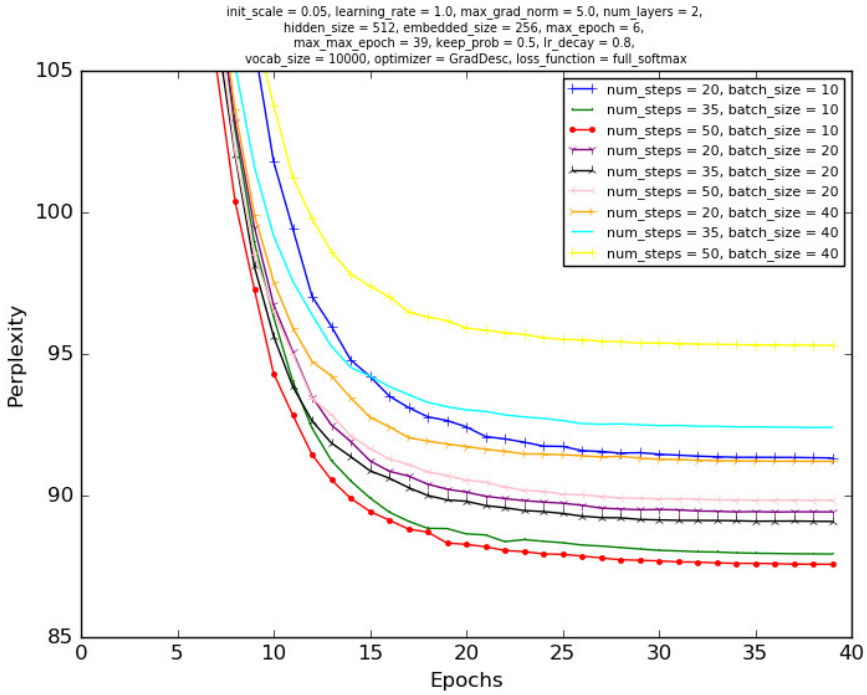

Figure 6: Convergence plot of the validation perplexity of the experiment with **num_steps & batch_size** as variable parameters.

The results of this experiment are displayed in table 6 and figure 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 not very clear: there seem to be local optima for these parameters. No definitive conclusion can be made based on this experiment.

2.8 num_steps

Table 7: Last training, validation and test perplexities and the training speed with **num_steps** as variable parameter.

Name	Train PPL	Valid PPL	Test PPL	Average speed
num_steps = 15	59.9904	89.7558	86.1790	4796.6922
num_steps = 35	56.3721	88.4145	85.7701	5317.4104
num_steps = 55	58.1670	91.1254	87.9059	5522.2976
num_steps = 75	60.4014	92.5607	89.4328	5606.5938

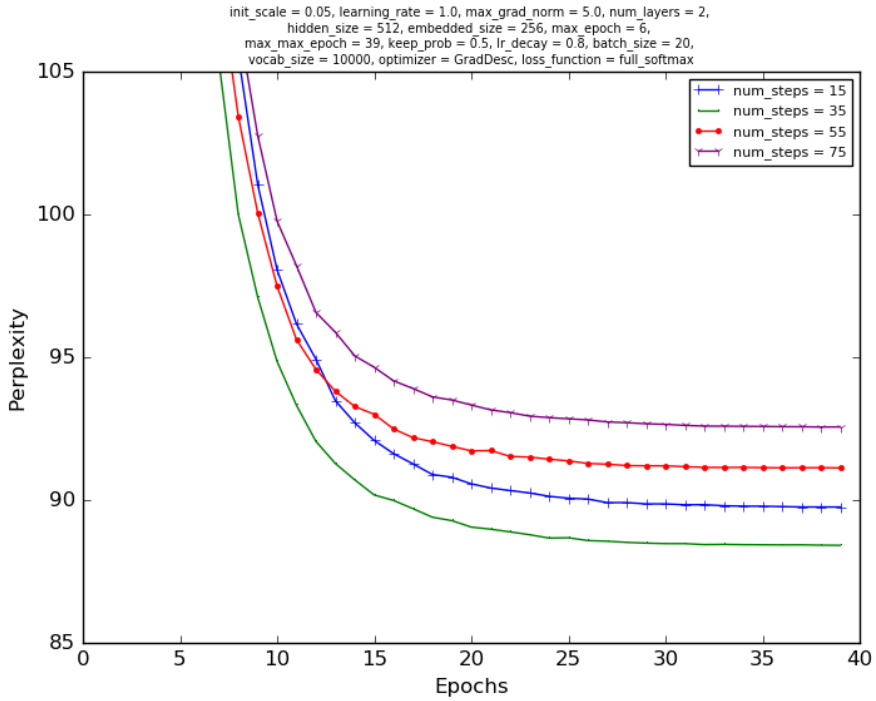

Figure 7: Convergence plot of the validation perplexity of the experiment with **num_steps** as variable parameter.

The results of this experiment are displayed in table 7 and figure 7. The figure shows a local optimum for **num_steps** equal to 35. This is somewhat strange as one would expect the performance to go up when more time steps are taken into account for training.

The training speed seems to go down as **num_steps** is reduced, which is logical as each element of the training batch has length **num_steps**. This leads to less batches in one epoch, hence there are less weight updates per epoch and the speed goes up.

2.9 optimizer

Table 8: Last training, validation and test perplexities and the training speed with **optimizer** as variable parameter.

Name	Train PPL	Valid PPL	Test PPL	Average speed
optimizer = GradDesc	56.5491	89.0828	85.6273	5324.8581
optimizer = Adadelta	764.3933	720.4745	689.8613	5188.6086
optimizer = Adagrad	67.8530	97.5119	93.7309	5261.3310
optimizer = Adam	66.7381	118.1673	108.4468	4806.0537
optimizer = Momentum	58.1349	88.0675	84.2937	5264.8214

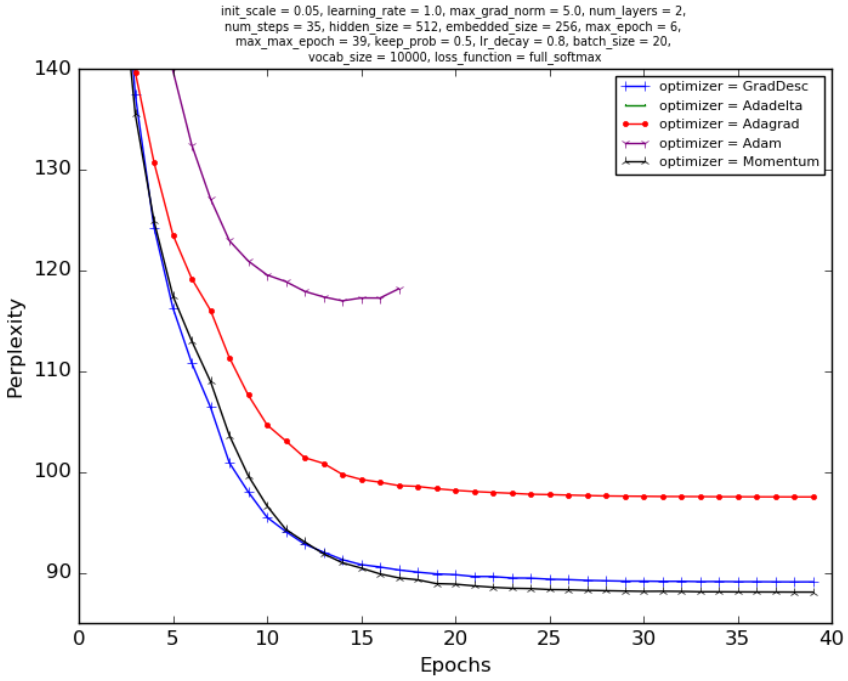

Figure 8: Convergence plot of the validation perplexity of the experiment with **optimizer** as variable parameter. The Adagrad optimizer is not shown because his last perplexity at epoch 39 was still far above the others.

The results of this experiment are displayed in table 8 and figure 8. There seems be one big loser in this experiment: the Adadelta optimizing algorithm. Adam also doesn't seem to perform well and is also the slowest. Adagrad is slightly better, but the gradient descent and momentum algorithms are the clear winners of this experiment.

2.10 loss_function

Table 9: Last training, validation and test perplexities and the training speed with **loss_function** as variable parameter.

Name	Train PPL	Valid PPL	Test PPL	Average speed
loss_function = full softmax	56.7543	88.6126	85.5887	5318.9263
loss_function = sampled softmax	4.7374	99.3000	94.9911	6250.8133

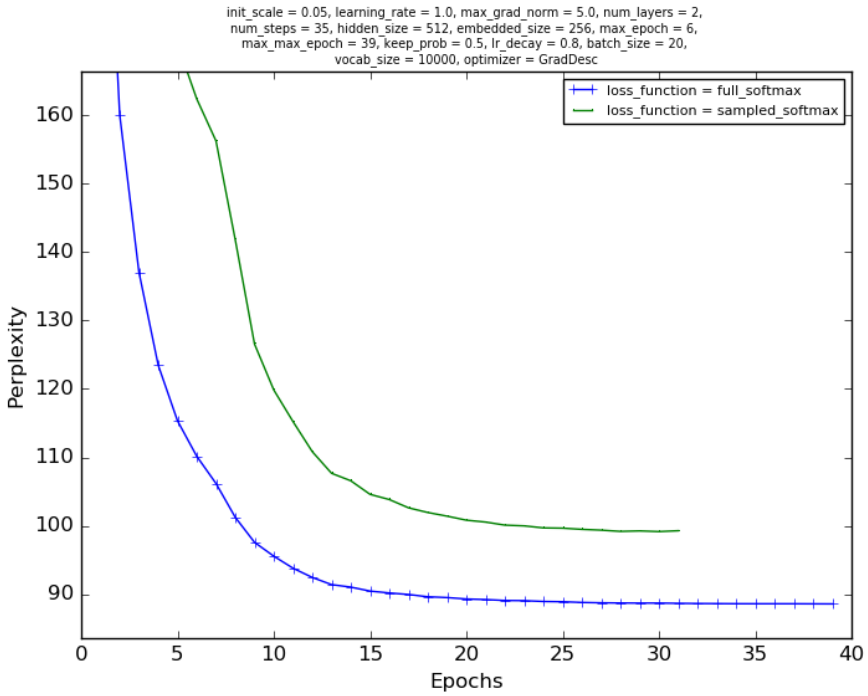

Figure 9: Convergence plot of the validation perplexity of the experiment with **loss_function** as variable parameter.

The results of this experiment are displayed in table 9 and figure 9. The results are as expected, the sampled softmax loss function trains quicker but performs worse than the full softmax function. Extra experiments will have to be done to see if this loss in performance is worth the gain in speed.

Please note that the training perplexity when using the sampled softmax loss function is not a real perplexity value. This is because the training perplexity is calculated based on the output of the sampled softmax probabilities. These probabilities are not properly normalized, as described in section 1.2.14, and hence the training performance is strongly overestimated.

3 Conclusion

These experiments investigated the impact of the different training parameters involved when using recurrent neural networks for language modeling. To this end, the TensorFlow tutorial model of a recurrent neural network for language modeling (publicly available) was used with the Penn Treebank 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 and **lr_decay** should not be too high.
- **max_grad_norm** should not be too low as this hinders the learning process of the network.
- Increasing **num_steps** makes training faster, regarding performance this parameter seems to have a local optimum.
- The sampled softmax loss function has worse performance but better training speed than the full softmax loss function.
- The gradient descent and momentum optimization algorithms do better than the other algorithms in terms of performance and speed.
- 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*