Report Practical assignment 2

Team members:
Fares Ben Slimane
Parviz Haggi
Mohammed Loukili
Jorge A. Gutierrez Ortega

March 24, 2019

Abstract

This report explains our approaches to solving the problems of the practical assignment 2, the experiments we performed, the results and conclusion of our work. The code we developed is uploaded to our Github repository [1].
In this assignment we implement and train sequential language models on the Penn Treebank dataset. Problem1-3 include the implementation of a simple vanilla RNN, an RNN with a gating mechanism(GRU) and a transformer network, respectively

1 Problem 1: Implementing a Simple RNN

The implementation of a Simple Recurrent Neural Network can be found in the models.py script. To train the model you need to execute the script ptb-ml.py.

2 Problem 2: Implementing an RNN with Gated Recurrent Units (GRU)

The implementation of an RNN with a gating mechanism (GRU) can be found in the models.py script. To train the model you need to execute the script ptb-ml.py.

3 Problem 3: Implementing the attention module of a transformer network

The implementation of the attention module of a transformer network can be found in the models.py script. To train the model you need to execute the script ptb-ml.py.

4 Training language models

4.1 Model Comparison

In this section we will present the result of the first experiment of the 3 models the vanilla RNN, GRU, and Transformer, with their correspondant hyperparameters shown in Table 1. All the models are run for 40 epochs.

Hyperparameters	Vanilla RNN	GRU	Transformer
Optimizer	ADAM	SGD_LR_SCHEDULE	SGD_LR_SCHEDULE
Learning rate	0.0001	10	20
Batch size	20	20	128
Sequence length	35	35	35
Hidden size	1500	1500	512
Number of layers	2	2	6
Dropout probability	0.35	0.35	0.9

Table 1: Model's settings

Table 2 shows the results of this first experiment. We notice that the Transformer is the best model in terms of time-processing and validation/training loss. The GRU is the most expensive model in term of time processing but gives a better result than the vanilla RNN. The latter is the worst in term of training and validation loss.

Result	Vanilla RNN	RNN with GRU	Transformer
Training PPL	120.97	65.85	???
Validation PPL	157.82	102.63	???
Time processing per epoch (s)	411.05	668.06	174.59

Table 2: First experiment results

The perplexity results are the same for RNN and GRU. For the for Transformer we got a result close to what was expected:

- RNN: train: 120 val: 157

- GRU: train: 65 val: 104

- TRANSFORMER: train(our): ?? val: ??

- TRANSFORMER: train(expected): 67 val: 146

This proves that our models are well implemented. The different values in the case of Transformer can be explained by the fact that the transformer is sensitive to initialization and implementation of the code.

Lastly, the Figures below show the learning curves for (train and validation) PPL per epoch and per wall-clock-time, for the architectures above:

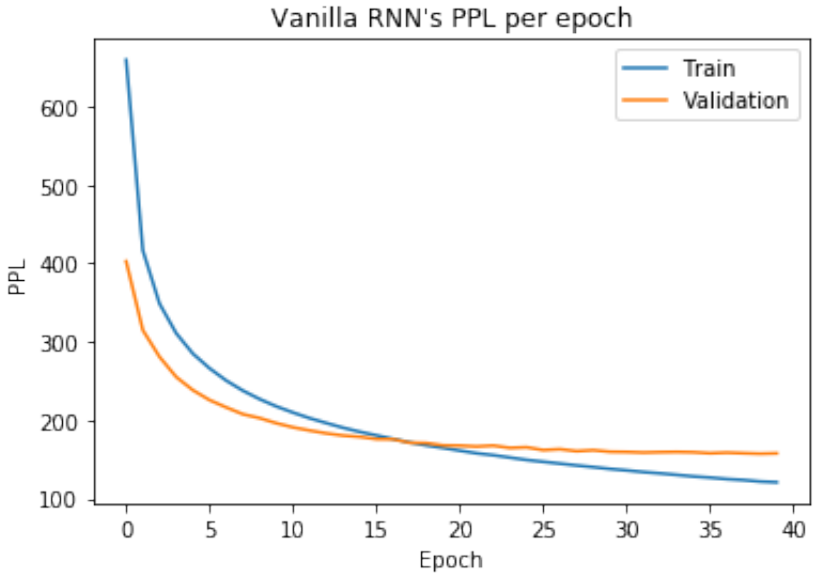

Figure 1: Vanilla RNN's PPL per epoch

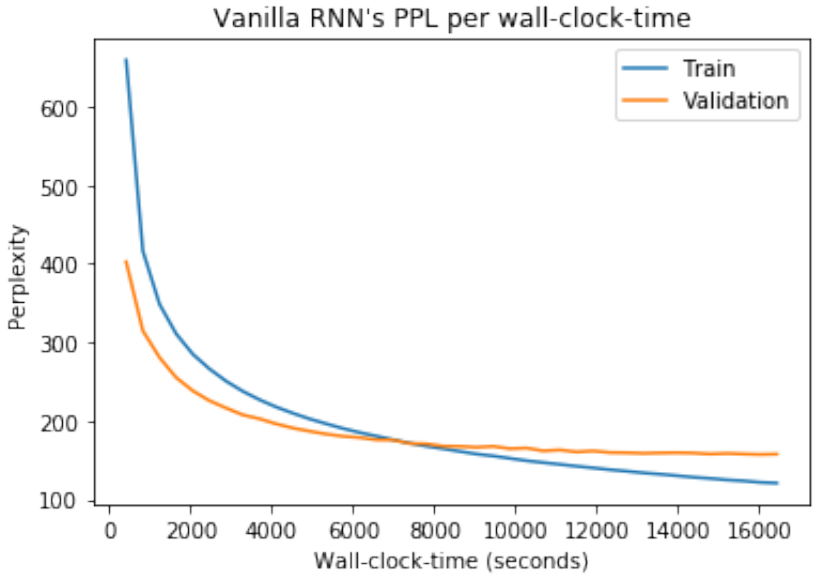

Figure 2: Vanilla RNN's PPL per wall-clock-time

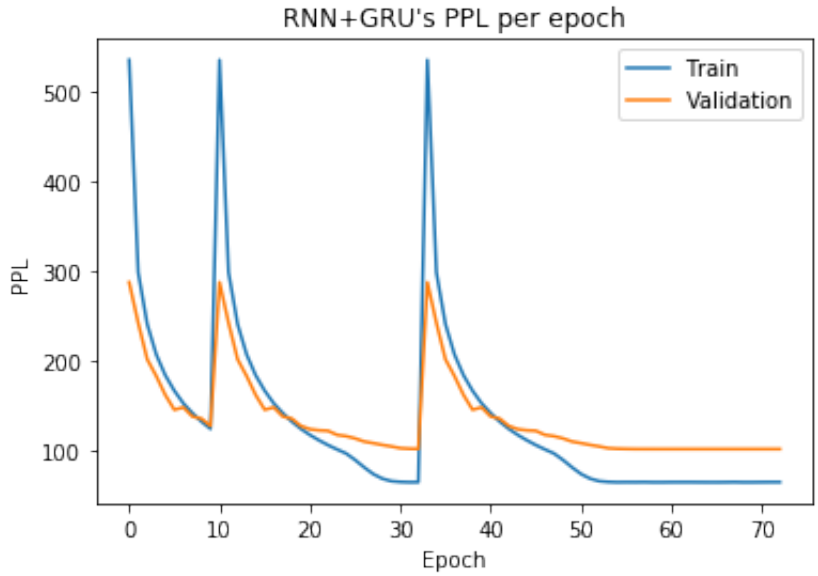

Figure 3: RNN with GRU PPL per epoch

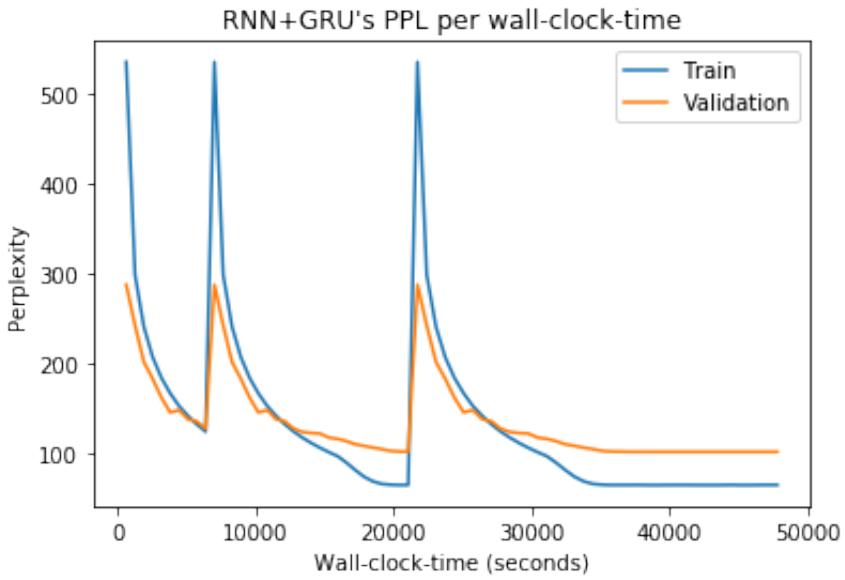

Figure 4: RNN with GRU PPL per wall-clock-time

4.2 Exploration of optimizers

In this section we will explore different optimizers for each of the previous models. Each model is run with two different optimizers with the hyperparameters as given in the assignment.

Include the following tables: 1. For each experiment in 1-3, plot learning curves (train and validation) of PPL over both epochs and wall-clock-time.

1) **Results for Vanilla RNN:** The hyperparameters used for experiments 2 and 3 are given in the Table 3.

Hyperparameters	**Experiment 1**	**Experiment 2**	**Experiment 3**
Optimizer	ADAM	SGD	SGD_LR_SCHEDULE
Learning rate	0.0001	0.0001	1
Batch size	20	20	20
Sequence length	35	35	35
Hidden size	1500	1500	512
Number of layers	2	2	2
Dropout probability	0.35	0.35	0.35

Table 3: Vanilla RNN additionnal experiments' hyperparameters

The results of these experiments are shown in Table 4. We notice that SGD performed worst and could not converge within 40 epochs, whereas ADAM performs best for the same number of epochs and the same hyperparameters. Additionnaly, the SGD_LR_SCHEDULE works better than SGD for a bigger learning rate and lower model capacity. In terms of training time, experiment 1 was the slowest, experiment 3 was the fastest while experiment 2 showed an average performance. Given the above setting with the hyperparameters as shown in Table 3, we conclude that the first experiment (ADAM) is best in terms of performance on the validation set.

Result	**Experiment 1**	**Experiment 2**	**Experiment 3**
Training PPL	120.97	3008.63	229.56
Validation PPL	157.82	2220.49	195.67
Average time processing per epoch (s)	411	384	185

Table 4: Vanilla RNN results experiments

2) **Results for GRU:** In the experiments 2 and 3, for GRU, we used the parameters given in Table 5.

Hyperparameters	**Experiment 1**	**Experiment 2**	**Experiment 3**
Optimizer	SGD_LR_SCHEDULE	SGD	ADAM
Learning rate	10	10	0.0001
Batch size	20	20	20
Sequence length	35	35	35
Hidden size	1500	1500	1500
Number of layers	2	2	2
Dropout probability	0.35	0.35	0.35

Table 5: RNN with GRU additionnal experiments' hyperparameters

The results are shown in Table 6. We notice that SGD_LR_SCHEDULE performed best on the validation set, which indicates that it might also generalize well. With a larger learning rate, SGD performed better than on the Vanilla RNN. ADAM's performance on the training set was best, but its variance was greater than SGD_LR_SCHEDULE, which may indicate that the model starts to overfit.

Result	**Experiment 1**	**Experiment 2**	**Experiment 3**
Training PPL	65.85	50.33	59.98
Validation PPL	102.63	121.36	113.71
Average time processing per epoch (s)	668	648	675

Table 6: First experiment results

3) **Results for Transformer:** The following parameters were used:

Hyperparameters	**Experiment 1**	**Experiment 2**	**Experiment 3**
Optimizer	SGD_LR_SCHEDULE	SGD	ADAM
Learning rate	20	20	0.001
Batch size	128	128	128
Sequence length	35	35	35
Hidden size	512	512	512
Number of layers	6	6	2
Dropout probability	0.9	0.9	0.9

Table 7: The hyperparameters for additional Transformer experiments

The following are the results for the transformer:

Result	**Experiment 1**	**Experiment 2**	**Experiment 3**
Training PPL	???	???	???
Validation PPL	???	???	???
Average time processing per epoch (s)	???	???	???

Table 8: First experiment results

In conclusion, assuming that the comparison here was made with a set of hyperparameters that give a fair baseline for all the models, we can say that there is no universal best optimizer regardeless of the model architecture. In fact, each model architecture works better with a suited optimizer. Howerver, we notice that ADAM is a very powerfull optimizer that can make the model converge very quickly. However, if not well-tuned it could make the model overfit the training data.

4.3 Exploration of hyperparmeters

Figures and Tables:
Each table and

gure should have an explanatory caption. For tables, this goes above, for

figures it goes below. Tables should have appropriate column and/or row headers. Figures should have labelled axes and a legend.

Include the following tables: 1. For each experiment in 1-3, plot learning curves (train and validation) of PPL over both epochs and wall-clock-time.

2. Make a table of results summarizing the train and validation performace for each experiment, indicating the architecture and optimizer. Sort by architecture, then optimizer, and number the experiments to refer to them easily later. Bold the best result for each architecture.

3. List all of the hyperparameters for each experiment in your report (e.g. specify the command you run in the terminal to launch the job, including the command line arguments).

4. Make 2 plots for each optimizer; one which has all of the validation curves for that optimizer over epochs and one over wall-clock-time. 5. Make 2 plots for each arcitecture; one which has all of the validation curves for that architecture over epochs and one over wall-clock-time.

4.4 Discussion

5 Problem 5: Detailed evaluation of trained models

1. We compute the average loss for all the validation sequences at each time step. The main script has been modified to load a pre-trained model and process the validation model while computing the loss at each time step individually(This is indicated in the script ptb-lm-P5-1.py under "Loss computation", Line 405). The resulting curves for the three models are shown in Figure 5.
   The loss is clearly larger when the training starts as the model does not have much information about the sequence at hand and cannot predict accurately. As we proceed, all the models should start making better predictions and thus the reduction of the loss. Although the loss fluctuates, it seems to be stabilizing around certain values, of which the RNN loss shows a higher loss while the GRU and Transformer perform similarly to each other but better than RNN.

2. Similarly to point 1 above, we modified the main script (see ptb-lm-P5-2.py under LOSS COMPUTATION on line 405). Here instead of processing the whole sequence at each step, we only run one mini-batch, processing each time step separately without touching the hidden states. This allows us to compute the gradient at time T with respect to the hidden state at each time step t.
   We compute the norm of the concatenated gradient vectors for the different layers and plot them with respect to the time steps as illustrated in Figure 6. Note that the values for each curve are rescaled to be in the range $[0, 1]$. This way we can compare the behavior of the gradients of the two different models (RNN and GRU). The graph shows how the gradients with respect to earlier time-steps are smaller than the gradients with respect to later times. This indicates that long-term dependencies contribute less to the gradient at a given point. The drop in the dependency is particularly steep on the RNN model. On the other hand, the mechanics of GRU help better mainatining the long-term dependencies.

3. We generate samples from both the simple RNN and GRU. We use the models from section 4.1. The sampling code can be found in the script generate.py. We produce 20 samples from RNN and GRU, 10 samples of the same length as the training sequnences (35) and 10 samples that are twice the length of the training sequeneces (70). The samples can be found in the Appendix of the report. As requested, we choose the three best, worst and interesting samples. The results are declared in the Appendix and can also be found in

Figure 5: text

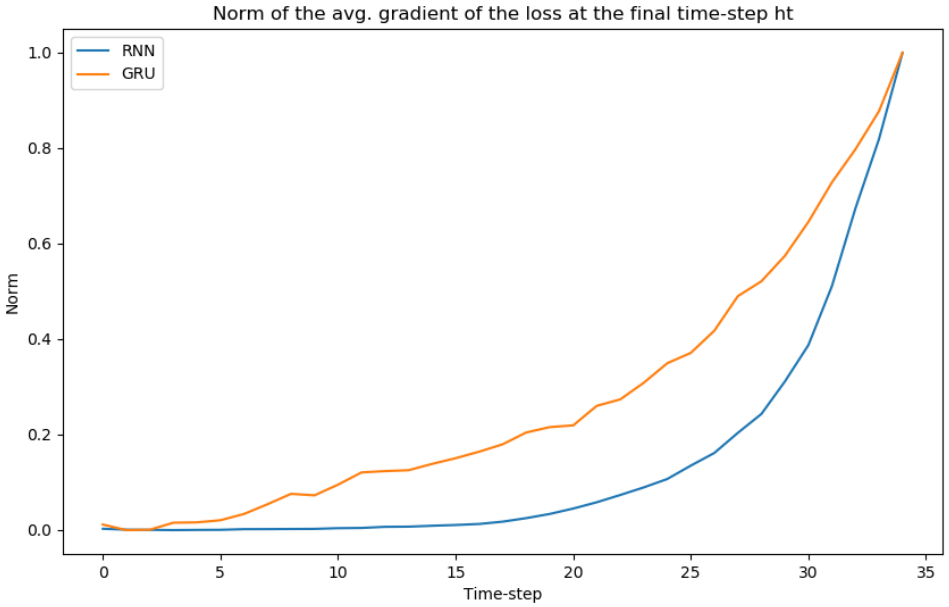

Figure 6: text

the provided github repository under folder samples. The subfolders are rnn-35, rnn-70, gru-35, gru-70.

References

[1] The machinists' github repository
`https://github.com/faresbs/Representation-Learning`