IFT 6135 – Representation Learning

Assignment 2 – Programming Part

Recurrent Neural Networks, Optimization and Attention

|  |  |
| --- | --- |
| **Students:** | Stefan Wapnick (id 20143021)  Mohamed Amine Arfaoui (id 20150893)  Oussema Keskes (id 20145195)  Stephan Anh Vu Tran (id 20145195) |
| **Due Date:** | March 25, 2019 |
| **Github Link:** | <https://github.com/stefanwapnick/IFT6135PracticalAssignments> |

# I - Experimental Setup

The following environment was used for all experiments in this assignment:

* Google Cloud Deep Learning Virtual Machine
* P100 GPU, 4vCPUs
* Python 3.7.1
* Pytorch 1.0.1.post2
* CUDA 10.0.13

# - Implementing a Simple RNN

## Methodology

In this section a simple Recurrent Neural Network (RNN) was implemented using Pytorch to be tested against the Penn Treebank Dataset. The principal equations of an RNN as listed below:

|  |  |
| --- | --- |
|  | Eq. 1 |
|  | Eq. 2 |

Where is the hidden cell state at time of the network, is the ’th input token (typically a word embedding) and the predicted next token in the sequence given the context .

A typical representation of an RNN illustrative its recursive nature is shown in XX.

However, for analysis purposes, the network is typically unrolled through time as in XX.

## Source Code

Listing 1 contains the implementation of the simple RNN done using Pytorch.

The following design decisions were made:

* The conventional Eq. 2 for computing the hidden state was transformed to use a single weight matrix for efficiency:
* A component class was made for each RNN cell or layer. The RNN class itself is then composed of many RNNLayer classes.
* A RNNBase class was made to be re-used in the GRU section since these two architectures are essentially the same except for the cell type.
* Weights were initialized using a form like Xavier initialization (although in this case the fan in size is always taken to be the hidden\_size)

Listing – RNN Implementation

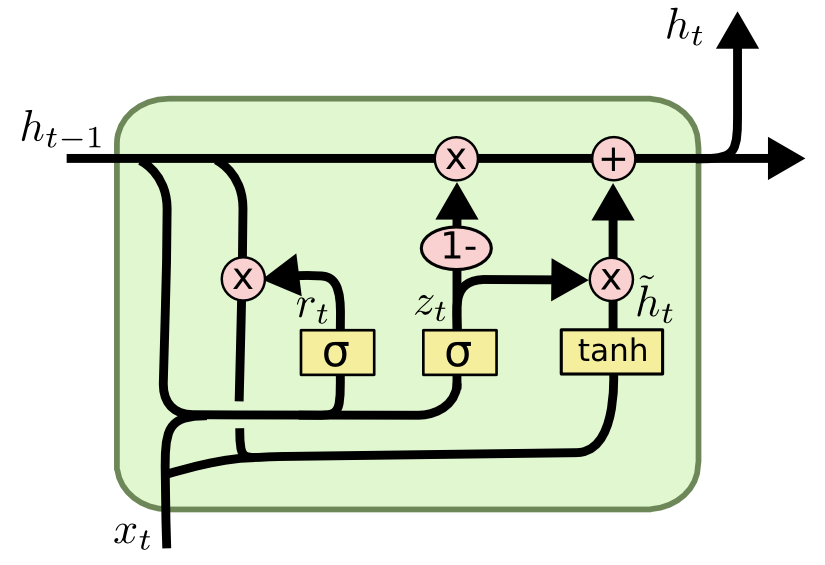
# Implement RNN with Gated Recurrent Units (GRU)

## Methodology

In this section the basic RNN of section 1 is augmented with gated recurrent units (GRU). The GRU adds in trainable weights for the reset and forget operations that allow the GRU to learn more long-term dependencies and alleviate the vanishing gradient problem. Although such an improvement comes with additional complexity and computational cost.

The principal equations of a GRU are:

is known as the reset gate and is known as the forget gate.



<http://sunlab.org/teaching/cse6250/fall2018/dl/dl-rnn.html#recurrent-neural-networks-2>

## Source Code

Listing 2 contains the implementation of the GRU done using Pytorch.

* Inputs and weight matrices for and are once again concatenated for efficiency as in section 1.
* A GRULayer component class is implemented that are composed inside the GRU call.
* The same RNNBase class from section 1 is re-used for the GRU class here since only the cell / layer type need be changed from section 1. All other logic for the recursive connections remains the same.

Listing – Implementation of GRU

# Attention Module of Transformer Network

## Methodology

The transformer is a newer architecture that uses the concept of attention (weighting of inputs based on perceived importance) for sequence modeling. Only a section of the transformer is implemented in this section, specifically the multi-head scaled dot-product attention defined below:

## Source Code

Listing – Implementation of Multi-Head Attention module

# Training Language Models

## Methodology

Using the implemented architectures of sections 1-3, experiments were ran against the Penn Treebank dataset with a variety of hyper-parameters.

## Results

### Learning Curves

### Exploration of Optimizers

### Exploration of Hyperparameters

### Summary of All Results

## Discussion

**Question 1. What did you expect to see in these experiments, and what actually happens? Why do you think that happens?**

**Question 2. Referring to the learning curves, qualitatively discuss the differences between the three optimizers in terms of training time, generalization performance, which architecture they're best for, relationship to other hyperparameters, etc.**

**Question 3. Which hyperparameters+optimizer would you use if you were most concerned with wallclock time? With generalization performance? In each case, what is the “cost” of the good performance (e.g. does better wall-clock time to a decent loss mean worse final loss? Does better generalization performance mean longer training time?)**

**Question 4. Which architecture is most \reliable" (decent generalization performance for most hyperparameter+optimizer settings), and which is more unstable across settings?**

**Question 5. Describe a question you are curious about and what experiment(s) (i.e. what architecture/optimizer/hyperparameters) you would run to investigate that question.**

# Detailed Evaluation of Trained Models

**Note:** For these experiments the architectures from Problem 4.1 (Model Comparison) are use.

Table – Models used for section 5

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Model | Optimizer | Initial Learning Rate | Batch Size | Sequence Length | Hidden Size | Layers | Dropout keep probability |
| RNN | ADAM | 0.0001 | 20 | 35 | 1500 | 2 | 0.35 |
| GRU | SGD\_LR\_Schedule | 10 | 20 | 35 | 1500 | 2 | 0.35 |
| Transformer | SGD\_LR\_Schedule | 20 | 128 | 35 | 512 | 6 | 0.9 |

## Average Loss per Time-Step

The average loss at each time-step is examined in this exercise. The losses were averaged over all mini-batches in the validation set.

**Results:**

Implementation can be found in 5\_1\_loss\_per\_timestep.py.

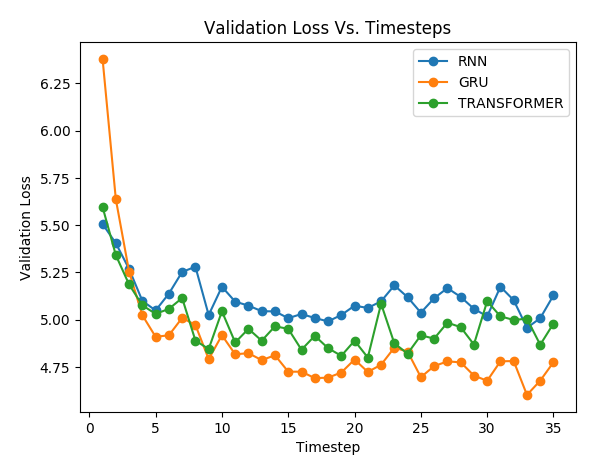


Figure – Validation loss over time-steps for each architecture of Problem 4.1

**Discussion:**

It was noted that the loss generally decreases over timesteps. This makes sense given that as more time passes more context is accumulated so the model hopefully makes better predictions (lower loss).

The GRU model appears to achieve best results in this regard (lowest loss at the end of the sequence length). This makes sense given that the GRU with its additional gates it better able to learn long-term dependencies. This can be contrasted with the RNN which can sometimes struggle with learning longer term dependencies. The transformer appears to be a middle ground between these two architectures.

It was also noted that the GRU appears to the start with the highest loss despite achieving final best performance. This could be coincidental or perhaps due to the larger number of parameters that must be tuned in comparison to the vanilla RNN so there is a larger possibility for error initially.

## Gradient per Time-Step

In this exercise the gradient of each hidden state to the final loss at the last timestep () is examined for a single mini-batch. Gradients in a batch are averaged together. The normal of these gradient vectors is computed and normalized to a range of [0, 1]. In the case of multiple layers, the gradient vectors are concatenated together (such that there is a single gradient vector per timestep.

**Results:**

Implementation can be found in 5\_2\_grad\_per\_timestep.py.

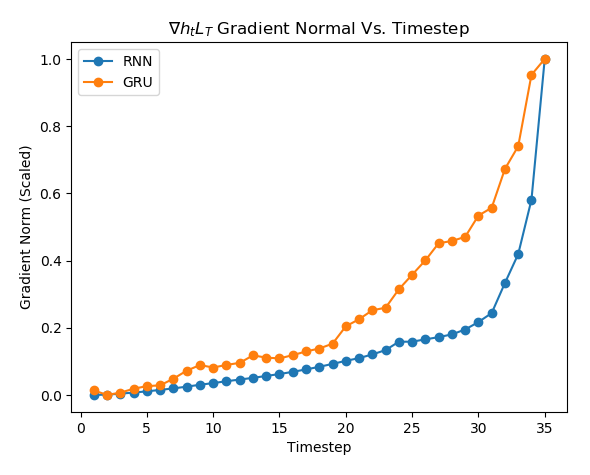


Figure - normal over timesteps

**Discussion:**

Gradients with respect to the final loss (at ) were found to be largest closer to . This makes sense since gradients of each typically decay (given the sequence of matrix multiplications during back-propagation) giving rise to the vanishing gradient problem in certain instances.

Gradients decay less rapidly in the GRU model, given its gated architecture alleviates the vanishing gradient problem and lets it learn long term dependencies.

## Generation of Samples

Using the trained RNN and GRU models, novel sentences were generated. This was done by sampling from the predicted distribution of output symbols and feeding back in this prediction as the next timestep input of the network. Adapting Eq. 1, the sampling operation can be written as:

**Results:**

Implementation can be found in 5\_3\_generate\_sentences.py.

Only 9 samples are shown here for analysis. All generated sequences are in the appendix.

**Discussion:**

The generated sentences are in general coherent however there is a noticeable different to human speech. In general, the models seem to be generating short sequences of coherent words, however the topic appears to change abruptly after several tokens.

It should also be noted that since outputs (or inputs for the next timestep) are sampled it is possible to receive an unlucky sample that may change the direction of the sentence.

In general, one would expect the GRU to generate more logical phrases (given its lower perplexity during testing and the nature of its gated architecture for learning longer term dependencies). This seems to be the case in many instances. However, it is also hard to provide an objective analysis of a sentence’s quality since it is somewhat opiniated and no concrete metric is being used here.

# References

|  |  |
| --- | --- |
| [2] | N.A. “Taxi-V2”. OpenAI. Available: <https://gym.openai.com/envs/Taxi-v2/> [Accessed: 2019-03-14] |
| [1] | R. Sutton and A. Barto, *Reinforcement Learning: An Introduction (2e).* MIT Press, 2018. Section 2.3 p. 28 |
| [3] | N.A. “Pendulum-V0”. OpenAI. Available: <https://gym.openai.com/envs/Pendulum-v0/> [Accessed: 2019-03-14] |
| [4] | D. Precup, “Comp-767: Reinforcement Learning – Assignment 2”. 2019, McGill University. Available: https://www.cs.mcgill.ca/~dprecup/courses/RL/Lectures/rl-hw2-2019.pdf [Accessed: 2019-03-14] |