Course: Deep Learning **(2021-12-08)**

Submission Assignment #3

Coordinator: Jakub Tomczak *Name:* Tavin Cole, *Netid:* tce350

1 Introduction: RNN sequence learning

This assignment explored sequence classification and autoregression with recurrent neural networks (RNN).

The first learning task was sequence to label prediction. Using a public dataset of (natural language) IMDB reviews [IMDB (2018)] labeled as positive or negative, the goal was to predict this label. Three basic architectures were compared. These will go by the names MLP, Elman, and LSTM, respectively. All of them began with an identical embedding layer (converting words to feature vectors in $\mathbb{R}^{300}$). The next layer was either a plain linear layer (the MLP), an Elman layer, or an LSTM layer. In the latter two cases the activated outputs of the recurrent layer were then fed into another linear layer. The remainder of each network was again identical: a ReLU activation, a global maxpool along the time dimension, and a linear projection to logits for K-class prediction. Note that although the learning task involved 2 classes, a multiclass architecture was used. The loss function was cross-entropy.

The performance of these 3 architectures was compared, in terms of cumulative loss per epoch and validation accuracy at the end of each epoch, over 10 epochs at different learning rates. No special optimizers were investigated; ordinary mini-batch SGD was used with fixed learning rates. All 3 architectures were able to achieve good accuracies, with LSTM > Elman > MLP only by slight margins. Finally these results were confirmed on a testing set.

The second learning task was sequence autoregression. A simple LSTM-based model was trained to learn the probability of the next token given a sequence of tokens. It consisted of an embeding in $\mathbb{R}^{32}$, a single layer LSTM, and a linear projection to logits according to the number of tokens in the grammar. As before, cross-entropy loss was optimized with plain mini-batch SGD. Two datasets were used, both stochastically generated from simple grammars:

`brackets`	Dyck language of correctly nested parentheses
`ndfa`	Trivial NDFA obeying the regular expression `s((abc!)*|(uvw!)*|(klm!)*)s`

Contrary to expectations, this model was able to learn both grammars quite easily – within just 1 or 2 epochs at high learning rates – and with accuracies approaching 100%. Perhaps this was due to the batching method used, and/or the use of loss masking for the padding token. Here, accuracy was a measure of how many grammatically correct sequences were obtained by sampling from the model 100 times.

2 Sequence to label prediction: MLP vs. Elman vs. LSTM

These models were constructed as specified in the assignment, with the embedding dimension and all hidden dimensions set to 300. Elman and LSTM layers were not stacked, i.e. they were single layer units.

Experimentation was limited to measuring the performance of these models as optimized by mini-batch SGD at different learning rates, namely

$$0.001, 0.003, 0.01, 0.03, 0.1, 0.3.$$

Due to the long training times, especially for the LSTM network, only 2 runs of 10 epochs were possible; per-epoch losses and accuracies will thus be reported as 2-sample averages. Again due to the long training times, depth and size variations of the network layers weren't explored, and neither were different optimizers.

As an additional exercise, a naive Elman implementation was written, mimicking `pytorch.nn.RNN` without actually using that module. This was trained for a little while but was indeed "horribly slow" to quote the assignment. No quantitative results for this model will be discussed.

Please refer to the following python files for the implementations:

`model.py`	Base classes and training/validation logic
`model_elman.py`	Naive Elman implementation (not pytorch.nn.RNN)
`model_lstm.py`	LSTM implementation
`model_mlp.py`	MLP implementation
`model_rnn.py`	Elman implementation (pytorch.nn.RNN)
`mydata.py`	Helpers for batching the datasets in pytorch style
`mytorch.py`	GlobalMaxPool implementation

Note that in the naive Elman implementation, the linear output layer was NOT included in the Elman module. It was simpler to place this linear layer in the outer `nn.Sequential` stack. This matches the way pytorch does it in the standard RNN and LSTM layers, though it's a slight deviation from what was suggested for the assignment.

Without further ado, the accuracies obtained from training are presented in figure 1. The plot comes from the jupyter notebook `Assignment_3.ipynb`. Please refer to the notebook for a similar plot of the loss curves and the full numerical results highlighted in table 1.

Model	LR	Accuracy	Loss
`LSTM`	0.30	0.8868	1.13
`Elman`	0.10	0.8846	3.24
`MLP`	0.03	0.8771	13.48

Table 1: Best results for each model within 10 epochs

So as expected, LSTM > Elman > MLP, but the difference is tiny in terms of real accuracy.

To complete this experiment, the winning models – two instances each of MLP[LR=0.03], Elman[LR=0.1], and LSTM[LR=0.3], saved after training – were trained for an additional 5 epochs on the canonical train/test split. The resulting accuracy plots are presented in figure 2. There was no appreciable change in the performance of the models and we can say the results obtained using the train/validation split are confirmed. That said, perhaps it's interesting to observe how the MLP and Elman models lost a very small fraction of their accuracy when confronted with new data, but recovered after a few epochs of additional training. LSTM didn't show this behavior. It may suggest that LSTM generalizes better in a sequential setting.

3 Sequence autoregression: LSTM

The model used for this task was so simple it can be quoted here:

```
model = nn.Sequential(
    nn.Embedding(num_embeddings=num_chars, padding_idx=padding_idx, embedding_dim=32),
    LSTM(32, 16),
    nn.Linear(16, num_chars),
)
```

The embedding dimension was 32 and the hidden dimension was 16, as was suggested. Please see the python file `arlstm.py` for details.

After training the model on time-shifted sequences from the stochastic datasets, and then sampling from the model to generate new sequences, it was easy to verify whether these were grammatically correct. The `ndfa` grammar obeys a simple regular expression and the `brackets` grammar just requires counting the opening and closing of parentheses. The validators used can be found in `mydata.py`.

Two slightly different validation techniques were used, which were named Synthesis and Completion. For Synthesis, the model was fed a start token and then sampled until hitting an end token or a length limit of 100 tokens. This was repeated 100 times (per epoch). For Completion, 100 sequences were randomly selected from the dataset and cut in half. The first halves were fed to the model and then sampling was done in the same way until hitting an end token or the length limit. In both cases, a temperature parameter was used to tune sampling randomness as outlined in the assignment. In this way a percent-correct validation score was obtained for Synthesis and Completion over the 5 temperatures 0, 0.25, 0.5, 0.75, and 1.0.

The batching strategy is of some interest. In order to simultaneously regulate memory usage and avoid catastrophic LSTM forgetting, the datasets were simply shuffled for every epoch, using pytorch's DataLoader `shuffle=True`, instead of working from short to long sequences or employing any workaround suggested in the assignment.

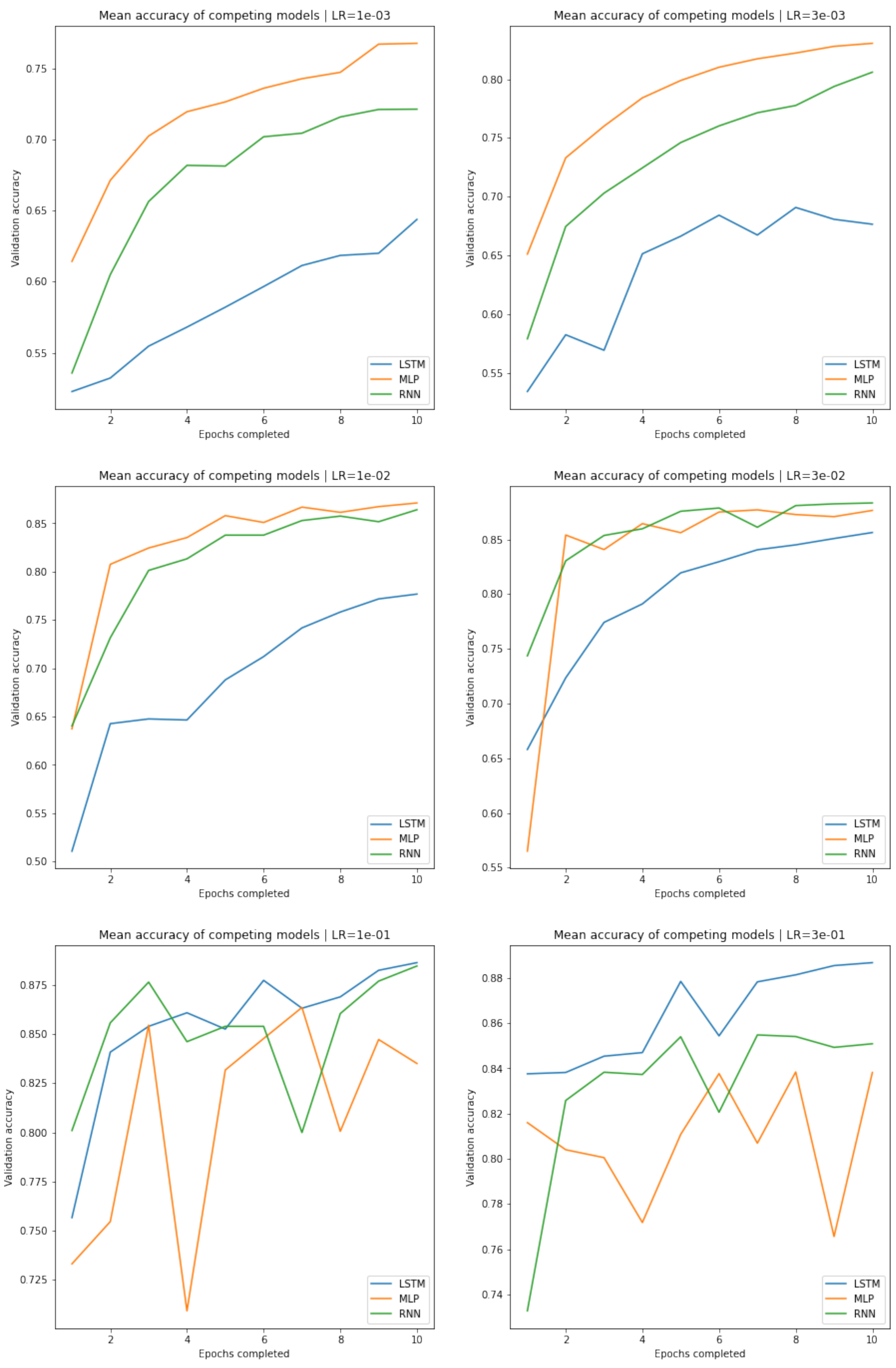

Figure 1: Mean validation accuracies of MLP, Elman, LSTM

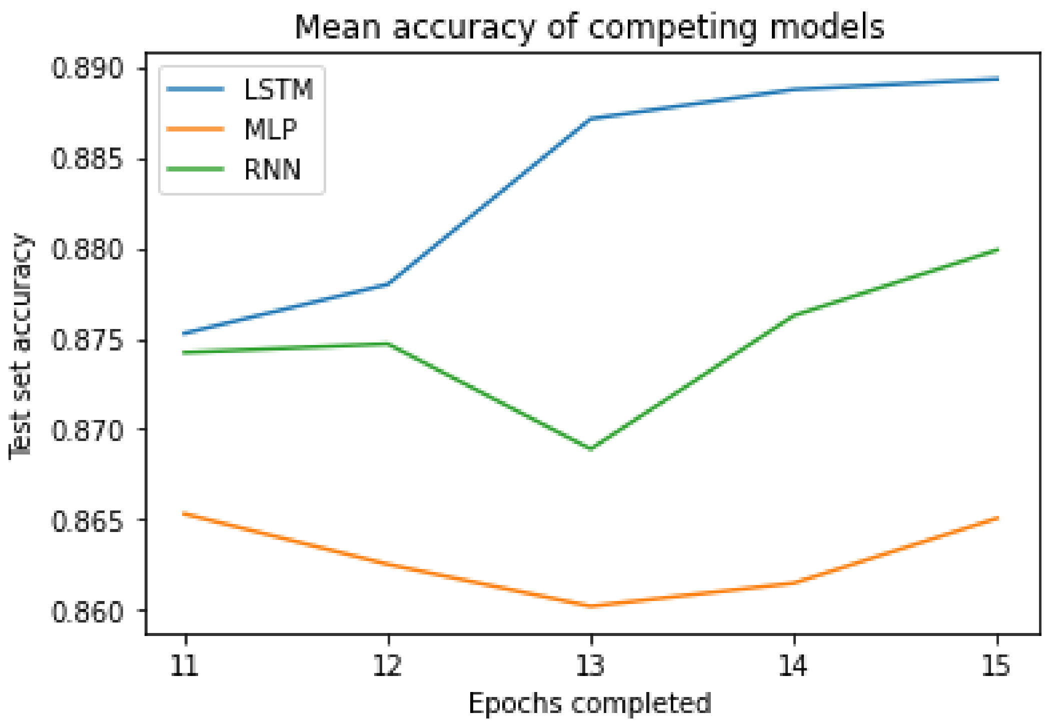

Figure 2: Mean test accuracies of MLP, Elman, LSTM

Another important hyperparameter of sorts was the use of loss masking for the padding token, i.e. passing `ignore_index=padding_idx` to the `cross_entropy` function. It should be noted that without enabling this option, the model had to train considerably longer to learn the grammars. A quantitative measure of this fact will not be presented; we'll focus instead on the excellent results obtained with this option enabled.

Following the suggestion in the assignment, the format of the experiment became training for 10 epochs while logging the cumulative loss and the validation scores to TensorBoard after each epoch. These fixed learning rates were evaluated:

$$0.01, 0.03, 0.1, 0.3, 1.0, 3.0.$$

Though the model did well at the slowest rates, setting the rate as high as 1.0 was found to improve training speed with no ill effects. Learning finally became volatile at 3.0. The results presented below were collected at a rate of 1.0.

For the `brackets` dataset, low temperates produced the best validation scores (naturally, choosing the most probable token avoids errors). It was the reverse for the `ndfa` dataset (choosing only the most probable token leads to a never-ending sequence that can't satisfy validation). Hence temperatures ≥ 0.5 for `brackets` have been filtered out, and temperatures ≤ 0.5 for `ndfa`. Filtered in this manner, figures 3 and 4 show the TensorBoard results for 10 trials of each dataset.

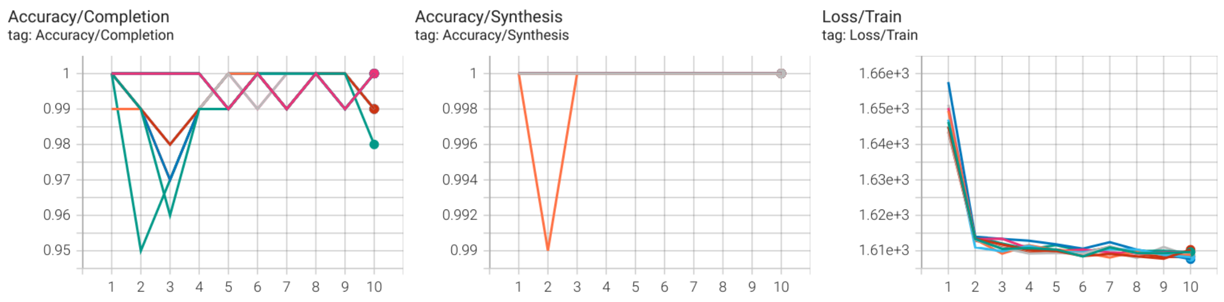

Figure 3: BRACKETS — lr=1.0

Without any special efforts, loss curves decayed smoothly[1] and the number of epochs needed was well under

[1]Well, at high learning rates, they dropped off a cliff straight to the minimum. At slow rates (not shown), they decayed smoothly.

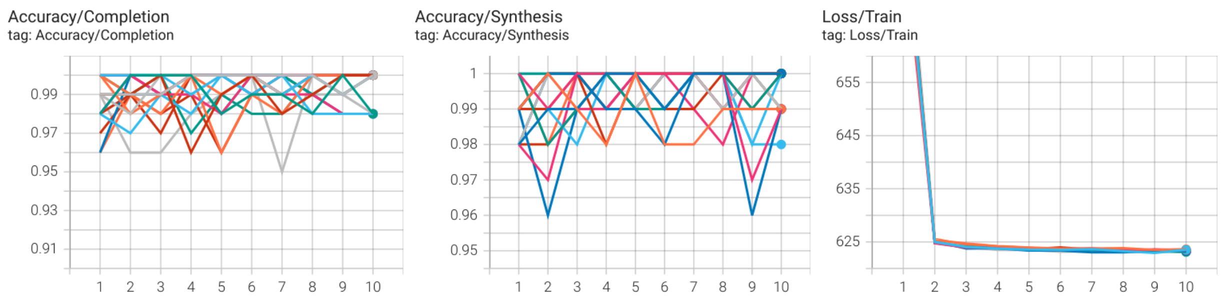

Figure 4: NDFA — lr=1.0

10, not close to 50 as the assignment warned. It's clear that for both datasets, the model essentially finished learning after 2 epochs, while it became nearly perfect at sequence generation. What's unclear is how to extract summary statistics from TensorBoard (e.g. means and variances or medians and percentiles). Hence these visualizations will stand as the final answer. Additional information from these TensorBoard runs is available upon request.

Excerpts of the training script's console output can be found in the appendix. These show a few good and bad generated sequences, as used to score validation accuracy.

References

IMDB (2018). Imdb dataset of 50k movie reviews. https://www.kaggle.com/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews.

Appendix

For each validation task shown below, only the first good and first bad sequence were printed to the console. Bad sequences were printed in red and good ones in green.

Console output snippet from `brackets` training

```
# Epoch 10
Cumulative loss: 1.6e+03
( ) .end
Synthesis accuracy: 1.000; temperature: 0.0
( ( ) ) .end
Synthesis accuracy: 1.000; temperature: 0.25
( ) .end
Synthesis accuracy: 1.000; temperature: 0.5
( ) .end
Synthesis accuracy: 1.000; temperature: 0.75
( ( ) ( ) ) .end
Synthesis accuracy: 1.000; temperature: 1.0
( ( ) ) .end
Completion accuracy: 1.000; temperature: 0.0
( ( ) ( ) ) .end
( ( ( ) ( ) ( ( ) ) ( ( ( ) ( ( ( ( ) ) ( ( ( ( ) ( ) ( ( ) ) ( ) ( ) ( ) ( ) ( ) ( ( ( ( ( ) ) ( ) ( )
Completion accuracy: 0.990; temperature: 0.25
( ( ( ) ) ) .end
Completion accuracy: 1.000; temperature: 0.5
( ( ) ( ) ( ( ( ) ) ) ) .end
( ( ) ( ( ( ( ( ( ) ( ) ( ) ( ( ) ( ) ( ) ) ) ) ) ( ( ( ) ( ( ) ( ) ) ) ) ) ( ( ) ( ) ( ) ( ( ( ( ( (
Completion accuracy: 0.990; temperature: 0.75
( ( ) ( ( ( ) ) ( ) ) ( ) ) .end
( ( ( ( ( ( ) ( ) ( ) ) ( ) ) ) ( ( ( ) ( ( ) ( ( ( ( ) ( ) ) ) ( ( ( ) ( ( ) ( ( ( ) ) ( ( ( ( ) ( ( )
Completion accuracy: 0.980; temperature: 1.0
```

```
Elapsed time: 41.7s
```

Console output snippet from `ndfa` training

```
# Epoch 10
Cumulative loss: 6.2e+02
s u v w ! u v w ! u v w ! u v w ! u v w ! u v w ! u v w ! u v w ! u v w ! u v w ! u v w ! u v w ! u v w
Synthesis accuracy: 0.000; temperature: 0.0
s a b c ! a b c ! a b c ! a b c ! a b c ! a b c ! a b c ! a b c ! a b c ! a b c ! a b c ! a b c ! a b c
s s .end
Synthesis accuracy: 0.320; temperature: 0.25
s u v w ! u v w ! u v w ! u v w ! u v w ! u v w ! u v w ! u v w ! u v w ! u v w ! u v w ! u v w ! u v w
s k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m
Synthesis accuracy: 0.880; temperature: 0.5
s u v w ! u v w ! u v w ! s .end
Synthesis accuracy: 1.000; temperature: 0.75
s k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! s .end
s a b c ! a b c ! a b c ! a b c ! a b c ! a b c ! a b c ! a b c ! a b c ! a b c ! a b c ! a b c ! a b c
Synthesis accuracy: 0.990; temperature: 1.0
s k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m
Completion accuracy: 0.000; temperature: 0.0
s k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m
s a b c ! a b c ! s .end
Completion accuracy: 0.270; temperature: 0.25
s u v w ! u v w ! u v w ! u v w ! u v w ! u v w ! u v w ! u v w ! u v w ! s .end
s k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m
Completion accuracy: 0.820; temperature: 0.5
s k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m
s k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m ! k l m
Completion accuracy: 0.980; temperature: 0.75
s a b c ! a b c ! s .end
Completion accuracy: 1.000; temperature: 1.0
Elapsed time: 48.8s
```