Deep Learning 2020 **(Due: [06-12-2020])**

Submission Assignment #4

Instructor: Jakub Tomczak *Name:* Jens van Holland, *Netid:* jhd660

This report is about taks A. Also, task B was tried first but did not work out, the code for this task is also submitted to show effort.

1 Problem statement

This report investigates the IMDb data set. This data set contains movie reviews which are positive or negative. The train and test data contain 25,000 reviews each. The task at hand is to classify if a review is positive or negative, using several deep learning models.

2 Methodology

Firstly, the train data is split into a new train data set and validation set with respectively 20,000 and 5,000 sequences. The data is sequential, to work with this data one needs to make each sequence of the same length. The maximum length of a sequence in the train data set is 2514 tokens, this means that each sequence should have this length. This requires more memory but does not contain more information (since most of the tokens are just '.pad', '.start' and '.end' tokens). For this problem, a variable batch size is used. To make sure that the last batch does not contain 1 sequence, each batch contains 15,000 tokens (or less) instead of the maximum length of a sequence (2514). In total, there are 325 batches and the length of the batches can differ. Also, the batches are shuffled during the training process. Furthermore, each token in a sequence is represented as a vector of length 300. Thus, a sequence is represented as a maxtrix.
Next, five different models are used, namely:

- Linear network
- Elman network (self implemented), see appendix B
- Elman network (RNN in PyTorch)
- LSTM
- Bi-LSTM

The Adam optimzer and an exponential learning rate schedule are used. Meaning that the learning rate is multiplied by $\gamma < 1$ before each epoch, except for the first epoch.
Using figures, one model (and its hyperparameters) is chosen for the final run. To evaluate the models, the final training and test scores are showed.

3 Experiments

As stated in the previous section, five different models are used. The results[1] are shown in figure 1. The hyperparameter specifics are stated in appendix A.

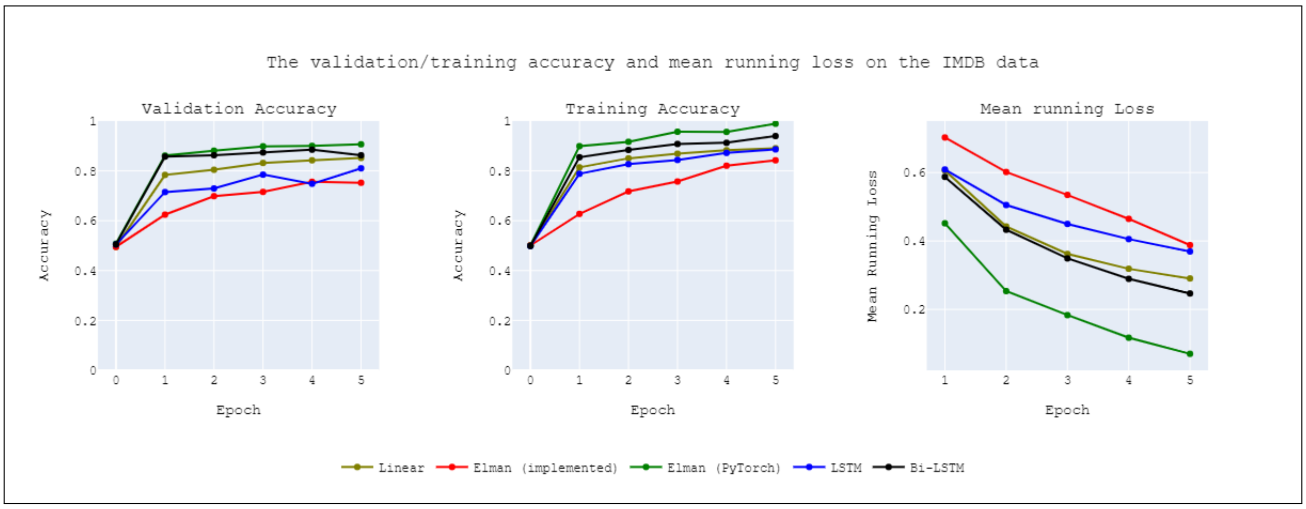

Figure 1: The validation accuracies (left), training accuracies (center) and the mean running loss (right) of the five models on the IMBD data set. Note that the training duration is 5 epochs, each evaluation is done before the new epoch starts, except for the running loss.

All models are performing quite well on the training and validation set. From the left and center graph, there are indications for overfitting. It seems that the Elman (RNN from PyTorch) works best, except that it overfits after the second epoch. Since there are two Elman models, the self implemented Elman model is omitted from further analysis.
For the final runs[2], the models will be trained for 2 epochs, using the full training and test set. The results are shown in table 2. The hyperparameters stated in appendix A are used.

Dataset	Linear	Elman (PyTorch)	LSTM	Bi-LSTM
Training	0.8526	0.9378	0.8270	0.8737
Test	0.8436	0.9090	0.8211	0.8685

Table 1: The final accuracies of the last epoch (2) of the final runs on the IMBD data set.

4 Results and discussion

Firstly, the models are performing quite good, except that some models tend to overfit. From the validation accuracies in figure 1, one could say that the Elman network performs better than the other networks. It is somewhat surprising that the linear model performs better than LSTM. The Bi-LSTM performs better than the LSTM, this was to be expected since the Bi-LSTM uses more information. The assignment stated that the (Bi-)LSTM would perform best, next the Elman network and at last the linear network. This is somewhat different than observed, possibly due to the chosen hyperparameters.

Based on these validation results, one would expect that the Elman model performs best on the final test data. After that the Bi-LSTM model and at last the LSTM and Linear model. This is verified by the final results. It seems that the Linear, Bi-LSTM and LSTM model don't overfit as much as the Elman model. Possibly, with some extra hyperparameter tuning, the Bi-LSTM and LSTM will perform better than the Elman model.

[1]Models are trained using a GPU from Google Colab
[2]Normally, I would choose a model and test it on the test set, but for the assignment's sake I evaluated all the models.

A Model hyperparameters

Model	learning rate	epochs	γ
Linear	0.0001	5	0.95
Elman (implemented)	0.005	5	0.95
Elman (PyTorch)	0.0005	5	0.999
LSTM	0.0001	5	0.95
Bi-LSTM	0.0001	5	0.95

Table 2: The hyperparameters of the models.

B Implementation (and usage) Elman network

```
class Elman(nn.Module):
  def __init__ ( self , insize= 300 , outsize= 300 , hsize= 300 ):
    super().__init__ ()

    #layer initializations
    self.lin1 = nn.Linear(insize + hsize, hsize)
    self.lin2 = nn.Linear(hsize, outsize)

  def forward( self , x, hidden= None):
    b, t, e = x.size()

    if hidden is None :
      hidden = torch.zeros(b, e, dtype =torch.float)

    outs = []
    # push to the GPU
    x = x.to(device)
    hidden = hidden.to(device)

    # iterate over time dim
    for i in range (t):

      inp = torch.cat([x[:, i, :], hidden], dim = 1 )
      hidden = torch.sigmoid(self.lin1(inp))
      out = self.lin2(hidden)

      outs.append(out[:, None , :])

    return torch.cat(outs, dim = 1 ), hidden

class ElmanRNN(nn.Module):
    #vocab size is assigned during initialization of Network
    def __init__(self, emb_dim = 300, vocab_size = None,
                 hidden_size = 300, classes = 2 ):
        super(ElmanRNN, self).__init__()
        # assign values to object attributes
        self.emb_dim =  emb_dim
        self.vocab_size =  vocab_size
        self.hidden_size = hidden_size
        self.classes = classes

        # layer initializations (Elman by default Elman(300, 300, 300))
        self.emb = nn.Embedding(self.vocab_size, self.emb_dim)
        self.elman = Elman()
        self.fc2 = nn.Linear(self.hidden_size, self.classes)

    def forward(self, x, hidden = None):
        x = self.emb(x)
        x = self.elman(x, hidden)[0]
        x = F.relu(x)
        x = torch.max(x, 1)[0]
        x = self.fc2(x)
        return x
```