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

Submission Assignment #3

Coordinator: Jakub Tomczak *Name:* Bartłomiej Boczek (2705555), Jan-Felix De Man (2750183), Teresa Liberatore (2732197), *Netid:*

1 RNN: BINARY CLASSIFICATION

1.1 Problem Statement

The IMDb data-set contains 50 000 reviews of movies, taken from the Internet Movie Database which are either highly positive or highly negative. The task is to predict which for a given review, through sentiment analysis. The goal is thus training a sequence-to-label model neural network such that the loss function is minimized and the accuracy maximized for the given data. The performance of MLP, Elman and LSTM trained neural networks with different hyper-parameters will be compared.

The input data contains tokenized words converted to integers indices in a fixed vocabulary, where each integer represents a word. The Target data contains the corresponding class labels: 0 for positive and 1 for negative.

1.2 Methodology

In order to achieve the task 5 neural networks have been implemented: MLP, RNN Elman implemented from scratch and MLP, Elman and LSTM networks already implemented in Torch. The structure of the model is the same for all the networks 1, with embedding and hidden size equal to 300. The embedding layer creates a number of embedding vectors equal to the number of tokens, which is equal to 99430 and can be found by taking the length of the dictionary that converts integers back to words, i2w.

The training input and target data are sliced into batched, padded to a fixed length and then converted to a torch tensor. It is to be remarked that since the model is not memory-intensive, it is possible to go to large batch size (fixed to 128) to speed up training: nevertheless the amount of padding increases with the batch size. This happens because if the batch size is increased, sequences of different lengths will be included in the same batch, and therefore the amount of padding will also increase.

	input/output	layer
1	tensor of `dtype=torch.long` with size (batch, time)	**`nn.Embedding(...)`** Convert the integer indices to embedding vectors
2	tensor of `dtype=torch.float` with size (batch, time, emb)	**`nn.Linear(emb, hidden)`** Map each token by a shared MLP
3	tensor of dtype=torch.float with size (batch, time, hidden)	`ReLU`
4	"	Global max pool along the time dimension Note the difference between a max pool as used in a CNN and a *global* maxpool. There is no special layer for this in pytorch, you can just implement it manually.
5	tensor of `dtype=torch.float` with size (batch, hidden)	**`nn.Linear(emb, numcls)`** Project down to the number of classes
	tensor of `dtype=torch.float` with size (batch, num_clases)	

Figure 1: Model for classification of IMDb data-set

1.2.1 Batch preparation and sequence preprocessing

In order to feed the training data into the several networks, some preprocessing needs to be done. First of all, the start and end values are added to every point in our data. Furthermore, sequences need to be padded to

the same size per batch using pad tokens. In this work, sequences were split into batches with a maximum size of 20 000 tokens. This was done in order of length, such that the longest sequences are in the same batch, and vice versa for the short sequences. Afterwards all the sequences were padded to the same length, the length of the longest sequence in that batch. By preparing batches this way we optimize GPU memory and models performance by reducing the number of padding to the minimum.

1.2.2 Elman implementation

In this work an own Elman layer was created. The implementation of the layer was the following:

```
class Elman(nn.Module):

  def __init__(self, insize = 300, outsize = 300, hsize = 300):
    super().__init__()

    self.lin1 =  nn.Linear(in_features= insize + hsize, out_features=hsize)
    self.lin2 =  nn.Linear(in_features= hsize, out_features= outsize)

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

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

    outs = []
    for i in range(t):
      inp = torch.cat([x[:, i, :], hidden], dim = 1)
      hidden = self.lin1(inp)
      hidden = F.relu(hidden)
      yi = self.lin2(hidden)
      out = yi

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

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

Since the lack of optimisation in our implementation and the computationally nature of the Elman layer, the equivalent RNN layer of the Pytorch library was used in the experiments.

1.2.3 Dropout layer

Training on RNN's has one *recurrent* problem: over-fitting. In order to reduce over-fitting done by our models with RNN layers, a dropout layer was implemented. A dropout layer randomly drops nodes out of the network, disconnecting them and any of its edges from the net for that batch. This should reduce the generalization that happens in a network, by forcing more responsibility' onto layers that are not dropped out.

1.2.4 Binary Cross Entropy loss with Logits

The loss function implemented was the binary cross entropy loss with logits. The loss functions applies a Sigmoid activation on the data. The function implies more numerical stability by combining both actions, using the log-sum-exp trick.

1.3 Experiments: tuning model hyper-parameters

Machine Learning gospel tells us that with the right tuning, the LSTM will perform best, followed by the Elman network, followed by the MLP.

In order to check which network performs the best, we train all of them tuning the hidden and embedding size while keeping learning rate, number of epochs and batch size constant to: 0.001, 10 and 128.

The hypothesis is that the LSTM network will perform the best, followed by the Elman and the MLP. For what concerns the hyper-parameters the hypothesis is that networks with larger hidden and embedding sizes will perform the best.

1.4 Results and discussion

The model with the MLP layer has the best validation accuracy, equal to 0.867229, when hidden size is fixed to 600 and embedding size to 150: here we report the plot of the mean loss and validation loss per epoch together with accuracy and f1 in Fig. 2. We can see from the loss plot that the model where this MLP network is implemented has a decreasing training loss over the epochs while the validation loss shows an increasing trend: this means that the model is over-fitting. MLP reaches best accuracy in epoch 4 and that is stats to decrease its performance due to overfitting. The model implementing the Elman layer that has the best validation accuracy, equal to 0.869905,is the one with embedding and hidden size equal to 300. The loss plot in Fig. 3 shows a decreasing training and validation loss across the epochs that gets to 0.1 and 0.4 respectively. The model with the LSTM layer shows the best validation accuracy, equal to 0.896665, is reached when hidden and embedding size is equal to 150. The loss plot of this model in Fig. 4 shows a decreasing training loss, while the validation loss shows an increasing trend: this is a sign that our model is over-fitting.

With best hyper-parameters for each architecture chosen on the validation set, we trained the models again on the entire training set (generated with $final = True$ flag). Models are evaluated on the test-set after training. Results are presented in Table 2. According to the intuition, MLP is the worst performing model with test accuracy 0.82, followed by Elamn with accuracy 0.86. Best results are achieved by our tuned LSTM network - accuracy is equal to 0.8793.

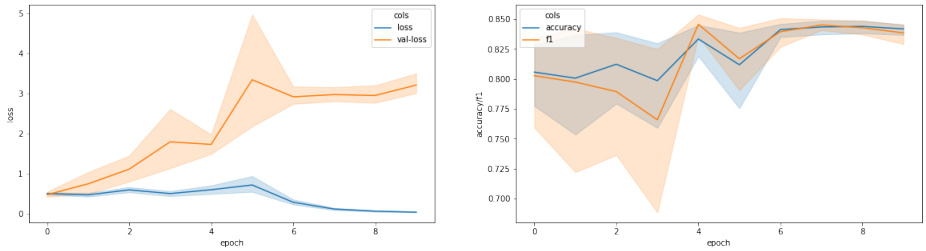

Figure 2: Loss and accuracy of MLP network with hidden size fixed to 600 and embedding size of 150. All metrics are mean per epoch. We evaluate 10 times during one epoch on **validation** set.

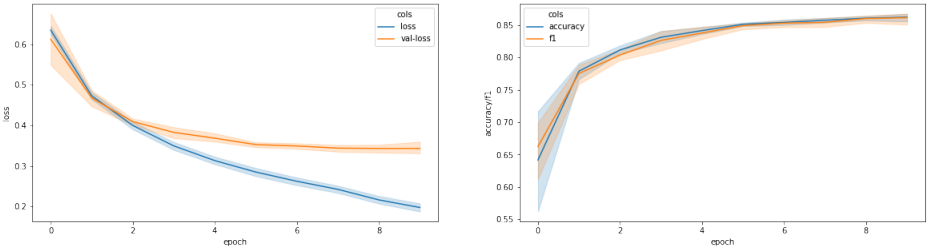

Figure 3: Loss and accuracy of RNN network with hidden and embedding size of 300. All metrics are mean per epoch. We evaluate 10 times during one epoch on **validation** set.

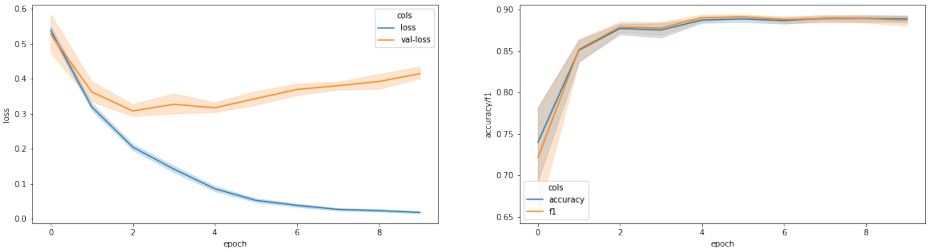

Figure 4: Loss and accuracy of LSTM network with hidden and embedding size of 600. All metrics are mean per epoch. We evaluate 10 times during one epoch on **validation** set.

Model	Hidden Size	Embedding Size	Learning Rate	Dropout	Max Validation Accuracy
MLP	300	300	0.01	-	0.862083
MLP	150	150	0.01	-	0.861260
MLP	600	600	0.01	-	0.860642
MLP	600	150	0.01	-	**0.867229**
MLP	150	600	0.01	-	0.858584
RNN	300	300	0.001	0.3	**0.869905**
RNN	150	150	0.001	0.3	0.856114
RNN	600	600	0.001	0.3	0.780774
RNN	600	150	0.001	0.3	0.856114
RNN	150	600	0.001	0.3	0.862907
LSTM	300	300	0.001	0.3	0.896254
LSTM	150	150	0.001	0.3	0.879580
LSTM	600	600	0.001	0.3	**0.896665**
LSTM	600	150	0.001	0.3	0.885961
LSTM	150	600	0.001	0.3	0.886373

Table 1: Results of hyper-parameters tuning. All results are given for the validation set.

Model	Hidden Size	Embedding Size	Learning Rate	Dropout	Epochs	Testset Accuracy
MLP	600	150	0.01	-	4	**0.8201**
RNN	300	300	0.001	0.3	10	**0.8657**
LSTM	600	600	0.001	0.3	10	**0.8793**

Table 2: Results for best architectures chosen on hyper-parameters tuning stage. All models were trained on the entire training set - generated with $final = True$ flag- and tested on the test-set

2 AUTO-REGRESSIVE MODEL: SEQUENCE PREDICTION

2.1 Problem Statement

The task of this model is to predict the next token in a sequence given all the tokens that precede it, namely a prediction sequence-to-sequence model. Two data-sets are used for this task: the NDFA data-set contains sequences sampled from a non-deterministic finite automaton displayed in Fig. /refndfa. The second data-set is called 'brackets' and consists out of valid bracket combinations, eg. (), (()), (()(())). Both data-sets were generated using the provided scripts and had a size of 50 000 examples. ,

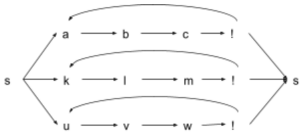

Figure 5: The finite automaton from which the NDFA data-set was sampled

Some remarks on NDFA As can be deducted from 5, the sequences generated by the NDFA should always start and end with an *s*. In between, either one of the 'words' *abc*!, *klm*! or *uvw*! should be present, for any amount of times (as long as it is the same word). The function used for generating sequences, however, allowed it to return the sequence '*ss*' as well, therefore these will also be considered correct in the remainder of this work.

Since the training is done autoregressively, the target is just the given sequence shifted one token to the left.

2.2 Methodology

The data is split into batches to make the training faster. A maximum number of tokens per batch is set in order to maximize memory utilization. The LSTM network is implemented to achieve this task and as it suffers from catastrophic forgetting the batches, which are shuffled before the start of each epoch, have variable sizes and contain similar length sequences. In that way the model will never go too long without seeing a long sequence. The model with three layers is implemented as shown in Fig. 6

The model is trained for 3 epochs with a single-layer network with embedding and hidden size equal to 32 and 16 respectively.

2.2.1 Batch preparation

Similar to the batch preparation described in Section 1.2.1, the data has been sorted by its size and put in to batches with sequences of the same size. Batch size is defined by maximum number of tokens. This time, however, no padding was implemented. Every sequence in every batch is of the same size. A lot of sequences had a length of, e.g. 2 (which is in turn the wrongful '*ss*'), thus we decided not to pad, only group together same length sequences.

2.2.2 Evaluation of the samples

The learning of this neural network is considered successful if in a sample of 10 sequences the majority is correct: therefore 10 samples are generated from a seed after each epoch. In order to get a random sample at each epoch,

	input/output	layer
1	tensor of `dtype=torch.long` with size (batch, time)	**`nn.Embedding(...)`** Convert the integer indices to embedding vectors
2	tensor of `dtype=torch.float` with size (batch, time, emb)	**`nn.LSTM(input_size=emb, hidden_size=h, numlayers=..., batch_first=True)`** Single layer LSTM
3	tensor of `dtype=torch.float` with size (batch, time, hidden)	**`nn.Linear(emb, vocab)`** Project down to the number of characters
	tensor of `dtype=torch.float` with size (batch, time, num_chars)	

Figure 6: Model for the sequence-to-sequence prediction

such that more variety is ensured, we cut the training data at a random point and use this as a seed. The seed are shuffled afterwards and used for sampling. In order to check if the sampled output is actually correct an evaluation function is written for each data-set. For the most complicated evaluation function, the finite automaton, first we check if the right letters are both at the start and end, both s. Afterwards we assert that no unknown elements are found in the string, nor any other other s's. Now we will check if any of the words is correctly found in the following places of the sequence, and if this word is only followed by itself. This is done by deleting all the elements that have been checked already and iterating over the 'words' that are left. It is implemented as follows:

```
def eval_ndfa(samples):
  words = (['abc!', 'uvw!', 'klm!'])
  abc = [w2i[x] for x in words[0]]
  uvw =  [w2i[x] for x in words[1]]
  klm =  [w2i[x] for x in words[2]]
  words = [abc, uvw, klm]

  correct = 0
  for sample in samples:

    sample =sample[0].tolist()

    # Delete .start and .end
    sample.pop(0)

    sample.pop(-1)

    if sample[0] != w2i['s'] or sample[-1] != w2i['s']:
      print('Not start / end with s')
      continue

    if w2i['.unk'] in sample or w2i['.start'] in sample:
      print('unk or start in middle')
      continue

    # First and last element MUST BE s at this point, delete them:

    if(len(sample) < 2): continue
    sample.pop(0)

    sample.pop(-1)

    if(len(sample) == 0):
      correct += 1
      continue

    if w2i['s'] in sample:
      print('rogue s spotted')
      continue

    if len(sample) % 4 != 0:
      print('words not % 4')
      continue

    if sample[0:4] == words[0]:
      while len(sample) >= 4:
        sample = delete_by_indices(sample, [0,1,2,3])

        if len(sample) == 0:
          correct += 1
```

```
          continue

        if sample[0:4] != words[0]:
          print('different word 0 or sth')
          continue

    if sample[0:4] == words[1]:
      while len(sample) >= 4:
        sample = delete_by_indices(sample, [0,1,2,3])
        if len(sample) == 0:
           correct += 1
           continue

        if sample[0:4] != words[1]:
          print('different word 1 or sth')
          continue

    if sample[0:4] == words[2]:
      while len(sample) >= 4:
        sample = delete_by_indices(sample, [0,1,2,3])

        if len(sample) == 0:
           correct += 1
           continue

        if sample[0:4] != words[2]:
          print('different word or sth')
          continue

  accuracy = correct / len(samples[0])

  return accuracy
```

The evaluation of the brackets is more straightforward. Its implementation is the following:

```
def isValid(s):
  d={'(':')'}
  stack=[]
  opening= list(d.keys()) # (
  closing=list(d.values()) # )
  for c in s:
    if c in opening:
      stack.append(c)
    elif c in closing:
      if stack==[] or d[stack.pop()]!=c:
        return False
  return(stack==[])
```

2.3 Experiments

We perform 2 experiments - one for *NDFA* data-set, one for *Brackets* data-set.

For NDFA we train the network with embedding size of 32, 1 LSTM layer with 16 hidden neurons, learning rate 0.01. We do the training with batch size of maximum 10k of characters for 3 epochs.

For Brackets we use the same hyper-parameters as for NDFA, but we train the model longer, for 50 epochs.

2.4 Results and discussion

2.4.1 NDFA

As can be seen in 7, there is a smooth decay in loss during the first three epochs. The code below was the output from sampling after training for three epochs. The seeds were generated using the method described in Section **??**. From both sets of ten samples, we had an accuracy of 90 percent. We can conclude that training was successful.

```
input | .start s k l
s k l m ! k l m ! s
s k l m ! s
s k l m ! s
s k l m ! s
s k l m ! s
s k l m ! s
s k l m ! k l m ! s
s k l m ! s
s k l m c ! s
error | words not % 4
s k l m ! s
9/10 correct

input | .start s a b c ! a
s a b c ! a b c ! s
s a b c ! a b c c ! s
error | words not % 4
s a b c ! a b c ! s
s a b c ! a b c ! s
s a b c ! a b c ! s
s a b c ! a b c ! s
s a b c ! a b c ! s
s a b c ! a b c ! s
s a b c ! a b c ! s
s a b c ! a b c ! s
9/10 correct
```

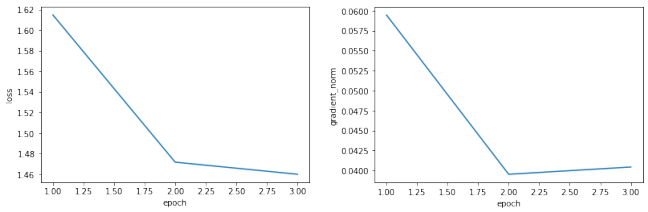

Figure 7: Training curves of seq2seq model for NDFA dataset. Loss on the left, normalized gradient on the right

2.4.2 Brackets

Fig. 9 portrays the loss curve of the training as well as its gradient norm. As can be expected, the gradient norms rise when the loss gets lower as well, as the gradient functions as an indicator for a change in loss. As expected because of the *gospel* mentioned in our assignment, the loss declined indeed after 30+ epochs of training.

To look at model performance we take 15 sequences of different lengths from the training set, cut in random place. This way we can see how out model performs, without calculating strict metrics.

Out inspection showed that although loss was still decreasing for 50 epochs, we noticed that the performance of the model is not good after 50 epochs and it frequently generates invalid sequences. Thus we decided to take model's checkpoint from 3rd epoch and evaluate it. It turned out, that results generated by the model are very good and great majority of the time, more than half of samples are correct - see printout below.

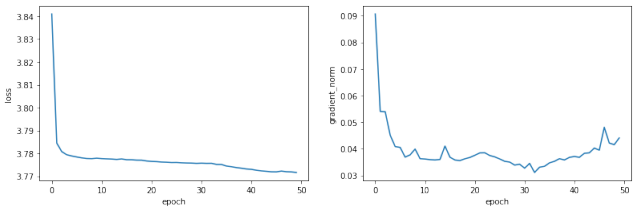

Figure 8: Training curves of seq2seq model for Brackets dataset. Loss on the left, normalized gradient on the right

```
MODEL TRAINED FOR 50 EPOCHS
input | [len: 14] .start ( ( ( ( ( ) ( ( ) ) ) )
10/10 failed
input | [len: 5] .start ( ( ) (
0/10 failed
input | [len: 42] .start ( ( ) ( ( ( ) ( ) ) ( ) ( ( ( ( ( ) ) ( ( ) ( ( ) ( ( ) ) ) ( ( ( ( ) ( ( ) )
1/10 failed
input | [len: 9] .start ( ( ( ( ) ( ) )
10/10 failed
input | [len: 5] .start ( ( ( )
9/10 failed
input | [len: 5] .start ( ( ) (
2/10 failed
input | [len: 7] .start ( ( ) ( ( )
0/10 failed
input | [len: 11] .start ( ( ) ( ( ) ( ) ( )
0/10 failed
input | [len: 25] .start ( ( ) ( ) ( ( ) ( ) ) ( ) ( ) ( ( ( ) ( ( ) ) )
0/10 failed
input | [len: 31] .start ( ( ( ( ( ) ( ) ( ) ( ) ( ) ) ( ( ) ) ( ( ( ) ( ) ) ) (
10/10 failed

MODEL TRAINED FOR 3 EPOCHS

input | [len: 7] .start ( ( ( ) ) (
3/10 failed
input | [len: 2] .start (
1/10 failed
input | [len: 2] .start (
0/10 failed
input | [len: 2] .start (
1/10 failed
input | [len: 2] .start (
0/10 failed
input | [len: 2] .start (
1/10 failed
input | [len: 14] .start ( ( ) ( ( ) ( ( ) ( ( ) (
4/10 failed
input | [len: 15] .start ( ( ( ( ) ( ) ) ( ( ) ) ) (
4/10 failed
input | [len: 13] .start ( ( ( ( ) ) ) ( ( ( ( )
2/10 failed
input | [len: 7] .start ( ( ( ) ( )
2/10 failed
input | [len: 214] .start ( ( ) ( ( ( ) ) ( ( ( ) ) ( ) ( ) ) ( ( ( ( ( ) ) ( ) ( ( ( ) ( ( ( ) ( ) ( )
2/10 failed
input | [len: 193] .start ( ( ) ( ( ( ( ) ) ) ) ( ( ( ( ( ( ) ( ( ) ) ) ) ( ( ) ( ) ) ) ( ) ( ( ( ) ) (
7/10 failed
```

```
input | [len: 335] .start ( ( ( ( ( ( ( ) ( ( ) ) ( ( ( ( ( ) ) ) ( ( ) ) ( ( ) ) ) ) ( ) ( ) ) ( ( ( )
3/10 failed
input | [len: 245] .start ( ( ( ) ( ) ( ( ( ) ( ( ) ) ( ( ( ) ) ( ( ( ( ( ) ) ) ) ( ( ( ) ( ( ( ) ) ) ) )
4/10 failed
input | [len: 391] .start ( ( ) ( ( ( ( ( ) ( ( ( ) ( ( ) ( ) ( ) ( ( ( ( ( ) ( ( ( ) ) ( ) ) ) ) ) ) ) ) ( ) (
2/10 failed
```

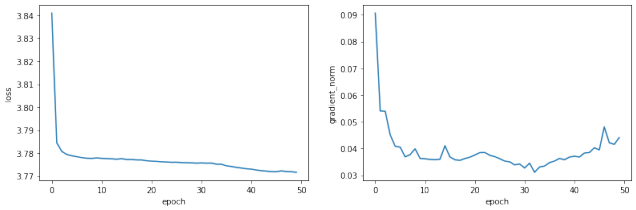

Figure 9: Training curves of seq2seq model for Brackets dataset. Loss on the left, normalized gradient on the right