Deep Learning 2020	**(Due: [2 December])**

Submission Assignment 4

Instructor: Jakub Tomczak *Name:* Richie Lee, *Netid:* [KLE680]

Introduction

One of the main factors that distinguishes humans from A.I. are our emotions. In this report we will challenge this consensus, by exploring Sentiment Classification. The approach for this emotion identification will be by applying various RNN's for movie-reviews. This task could be considered developing networks with EQ (Emotional Intelligence) and as a result potentially closing the gap between humans and A.I.

1 Problem statement

Data *Introducing the IMDb dataset*
IMDb (Internet Movie Database) is known as the world's most popular source for movie reviews. However, it is also a commonly-used sequence-learning database as it is a large, challenging classification dataset with plenty of examples. The utilised dataset consists of 50.000 preprocessed and tokenized reviews.

During model development and tuning we will work with a validation split of 20.000 training instances and 5.000 validation instances. For the final test, we will apply a canonical test/train split of 25.000 each.

Objective *Sentiment Classification*
The objective is to accurately identify the opinions in text and to assign corresponding labels. This binary sentiment classification will assign every review to either its positive or negative class.

2 Methodology

First we will apply *variable batch sizes*[1]. In other words, we will construct (approximately) equal-sized batches in terms of tokens, while allowing the amount of sequences per batch to differ. Implementing this improves our training time by significantly reducing the required padding and as a result the amount of computations. Furthermore we can now expect roughly the same amount of computations per batch.

This report will explore RNN's[2] (Recurrent Neural Networks). These networks are characterized by their ability to "remember" previous inputs, allowing temporal dynamic behaviour. This is beneficial in environments such as this classification where input dependency is expected. In addition, RNN's also provide better compatibility with arbitrary-length sequences. In our experimentation, we compare 2 RNN variations, alongside a MLP. Furthermore, we apply an Embedding layer and ADAM optimizer for all variations.

1. **MLP (Multilayer Perceptron)**, often referred to as "vanilla" neural networks, these basic feed-forward models can be considered the bread and butter of deep learning. Our model will contains a single hidden layer.
2. **Elman**, this model applies an Elman layer. In addition to regular hidden layer properties,it also contains context units which hold information of previous iterations. Moreover, we also implemented a residual connection[3] to counter exploding and vanishing gradients[4].
3. **LSTM (long-short term memory)**, this model is a RNN variation. In short, LSTM's main benefit is the "conveyer belt" consisting of only linear operations, avoiding gradient problems in the process. In addition, we also feed selected memory of the past to this belt through various "gates".

[1]Code snippets: Section 6.1, figure 4
[2]Architecture details provided in Appendix
[3]Code snippets: Section 6.2, figure 6
[4]Gradient problem details provided in Appendix

3 Results

Performance after first epoch *compares the three models in terms of average accuracy and loss*

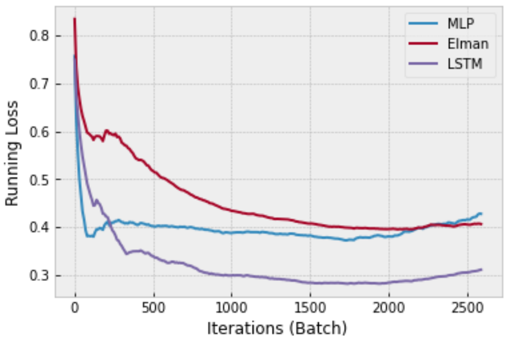
Table 1: Performance after first epoch

	Training		Test		
	Loss	Accuracy	Loss	Accuracy	Runtime
MLP ($lr = 0.001$)	0.4257	0.8250	0.4015	0.8455	0:34
Elman ($lr = 0.001$)	0.4050	0.7970	0.4056	0.8276	0:46
LSTM ($lr = 0.001$)	0.3102	0.8687	0.5182	0.7533	1:16

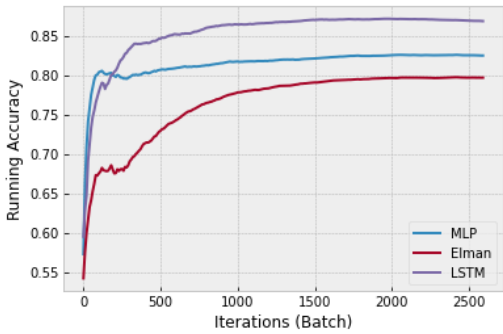
Generally we expect the LSTM to perform best, followed by the Elman and MLP respectively. However, from our models we observe the best performance using the MLP, followed by the Elman and LSTM. The reason for this unexpected result could be the difference in optimality in hyperparameters[5] (learning rate) among the models. Another potential driver could be insufficient training epochs, as we've decided to train single epochs (due to time constraints).

In addition we observe significant difference in training time for every model variation. These differences are not surprising as RNN's rely on higher amounts of computations. Finally, we observe significant performance difference between the LSTM's test and training performance. We believe this to be a result of overfitting[6].

Performance development during first epoch *compares the three models in terms of average running accuracy and loss during first epoch*



(a) Loss Development



(b) Accuracy Development

Figure 1: Running performance during first (Training) epoch

Figure 1 shows how the loss and accuracy developed throughout the first epoch, by showing the average loss/accuracy up to its corresponding iteration on the x-axis. These plots give clear insight on the training speed, which in turn contributes to parameter tuning.

4 Conclusion

This report was intended to demonstrate the advantages and disadvantages of RNN's. However our findings contradicted the expected results to some degree. We believe this to be the result of sub-optimal hyperparameter tuning. Furthermore we also observed exploding and vanishing gradients for the Elman network, but we conclude this drawback to be manageable, as we have managed to resolve it using residual connections.

In terms of preferred models, we are unable to draw a final conclusion. This is because we are potentially overlooking RNN advantages due to the applied hyperparameters. From our findings, we can however conclude all three models to be reasonable selections with performance expectancy exceeding 75% accuracy.

[5]Hyperparameter tuning is discussed in Section 5, Discussion

[6]The occurrence of optimizing "too well", up to a point from which we start learning the training data, rather than improving the general task

5 Discussion

- **Hyperparameter Tuning (Learning Rates):** Due to long training times and time constraints, we were just able to test a small selection of learning rates for single epochs. Applying the models on the validation/training split yielded the following findings:

Table 2: Learning Rate Tuning (Accuracies)

	Training			Validation		
Learning Rate	0.0001	0.001	0.01	0.0001	0.001	0.01
MLP	0.7571	**0.8261**	0.7691	0.7728	**0.8080**	0.8068
Elman	0.5728	**0.7721**	0.4889	0.5948	**0.8113**	0.4952
LSTM	0.6980	**0.8113**	0.8072	0.7358	0.8028	**0.8238**

Table 2 displays significantly different performance among learning rates. Especially the RNN variations (the Elman in particular) showed sensitivity to hyperparameter tuning. For model optimization, more tuning will be required.
The MLP and Elman show clear preferences for Learning rates, however this does not hold for the LSTM. We decided to select 0.001, because of more stable training loss development (figure 2).

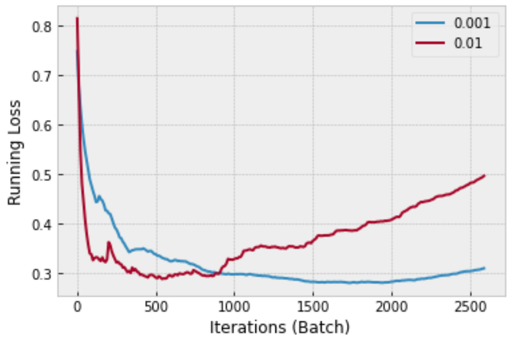
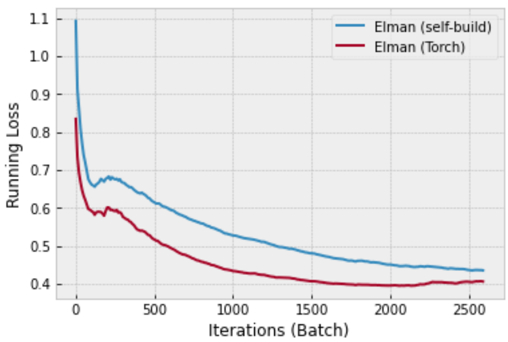
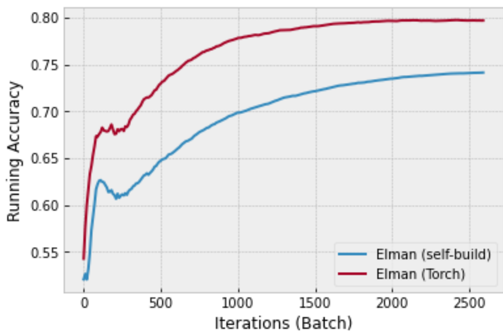


Figure 2: Loss Development LSTM

- **Comparing the self-build and Torch built-in Elman implementations**:Perhaps the most interesting finding, the performance difference among our Elman variations. (Though not closely related to the main objective of this report)



(a) Loss Development



(b) Accuracy Development

Figure 3: Comparing Training development of the "Self-build Elman" and the "Torch Built-in Elman"

Table 3: Performance after first epoch

	Training		Test		
Elman	Loss	Accuracy	Loss	Accuracy	Runtime
Torch ($lr = 0.001$)	0.4050	**0.7970**	0.4056	0.8276	0:46
Self-Build ($lr = 0.001$)	0.4343	0.7414	0.3323	**0.8562**	4:54

Interestingly enough, we find seemingly contradicting results in figure 3 and table 3. When evaluating the training performance, we observe better results for the Torch implementation. However, the Test-run tells us the opposite, with significantly better results for the Self-Build version.
The only difference between the models is the difference in non-linearity for the Elman layer. The self-build utilizes a *Sigmoid* activation in its Elman layer and the Torch a *Tanh*. This difference could affect performance through data compatibility.
Another aspect to consider are the different initialisations. These differences would fade away on the long term, but our testing range of 1 epoch would not catch it. Therefore we expect some degree of volatility as well.

6 Code Snippets

6.1 Variable Batch Size Implementation

```python
# Variable Batch Size
"""
Slices data into batches with approximately equal amounts of tokens
Pads accordingly
"""

# Padding function
def pad(l, size, padding):
    return l + [padding] * abs((len(l)-size))

# Variable batch slicing
def batch_slicing(x, y):

    # Initialize variables (e.g. x_B = x_batched)
    batch, x_B, y_B = [], [], []
    current_batch, current_size = 0, 0
    i = 1

    # Maximum # Tokens in batch = # Tokens in largest sequence (x_train = sorted)
    max_tokens = len(x[-1])

    # Iterates over dataset and slices into appropiate sizes (token-based)
    while i != len(x) + 1:
        batch.append(x[i - 1])
        current_size += len(x[i - 1])

        # Final batch/element (longest sequence)
        if i == len(x):
            x_B.append(batch)
            y_B.append(y[-1:])

        # Slice batch if max_tokens condition violated
        elif current_size + len(x[i]) >= max_tokens:

            # Padding for batch
            for j in range(len(batch)):
                batch[j] = pad(batch[j], size = len(batch[-1]), padding = 0)

            # store current batch
            x_B.append(batch)
            y_B.append(y[i - len(batch): i])

            # reset variables
            current_size = 0
            batch = []
            current_batch += 1

        i += 1

    # convert to Torch Tensors
    for i in range(len(x_B)):
        x_B[i] = torch.tensor(x_B[i], dtype = torch.long)
        y_B[i] = torch.tensor(y_B[i], dtype = torch.long)

    return x_B, y_B

# Obtained batched training data
x_train_B, y_train_B = batch_slicing(x_train, y_train)
x_val_B, y_val_B = batch_slicing(x_val, y_val)
```

Figure 4: Variable Batch Size and Padding functions

6.2 Model Implementations

```python
# MLP
"""
MLP network Class
utilizes PyTorch built-in functions
"""

class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.emb = nn.Embedding(num_embeddings = len(i2w), embedding_dim = 300)
        self.fc1 = nn.Linear(300, 300)
        self.fc2 = nn.Linear(300, 2)

    def forward(self, x):
        x = self.emb(x)
        x = self.fc1(x)
        x = F.relu(x)
        x, y = torch.max(x, dim = 1) # global max pool
        x = self.fc2(x)

        return x

mlp_net = MLP()
```

Figure 5: MLP

```python
# Elman (without Torch built-in function)
"""
Elman Network Class, and Elman (selfbuild) layer class
doesn't utilize PyTorch built-in RNN function, uses "Elman_selfbuild" class instead
"""

class Elman(nn.Module):
    def __init__(self, insize=300, outsize=300, hsize=300):
        super().__init__()
        self.fc1 = nn.Linear(insize + hsize, hsize)
        self.fc2 = nn.Linear(hsize, outsize)

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

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

        outputs_list = []
        for i in range(t):
            inputs = torch.cat([x[:, i, :], hidden], dim = 1)
            hidden = torch.sigmoid(self.fc1(inputs))
            outputs = self.fc2(hidden)
            outputs_list.append(outputs[:, None, :])

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

Figure 6: Elman (Selfbuild)

```python
# Elman (with Torch)
"""
Elman Network Class
utilizes PyTorch built-in function
"""

class Elman_Torch(nn.Module):
    def __init__(self):
        super(Elman_Torch, self).__init__()
        self.emb = nn.Embedding(num_embeddings = len(i2w), embedding_dim = 300)
        self.elman = torch.nn.RNN(input_size=300, hidden_size=300) # tanh non-linear
        self.fc2 = nn.Linear(300, 2)

    def forward(self, x):
        x = self.emb(x)
        x_residual, hidden = self.elman(x) # residual connection
        x = x + x_residual
        x, y = torch.max(x, dim = 1) # global max pool
        x = self.fc2(x)

        return x

elman_torch_net = Elman_Torch()
```

Figure 7: Elman (Torch)

```python
# LSTM (with Torch)
"""
LSTM Network Class
utilizes PyTorch built-in function
"""
class LSTM_Torch(nn.Module):
    def __init__(self):
        super(LSTM_Torch, self).__init__()
        self.emb = nn.Embedding(num_embeddings = len(i2w), embedding_dim = 300)
        self.lstm = torch.nn.LSTM(input_size=300, hidden_size=300)
        self.fc2 = nn.Linear(300, 2)

    def forward(self, x):
        x = self.emb(x)
        x, hidden = self.lstm(x)
        x, y = torch.max(x, dim = 1) # global max pool
        x = self.fc2(x)

        return x

lstm_torch_net = LSTM_Torch()
```

Figure 8: LSTM (Torch)

Appendix

- **Model Architecture specifications:**

  Input → Embedding Layer (300) → *[Model Dependent]* → Global Maxpool → Fully Connected layer (300,2) → Softmax → Cross Entropy Loss

$$[ModelDependent] = \begin{cases} MLP: & ReLu \\ Elman: & Elman(300, 300, 300), \text{ non-linearity} = \tanh \\ LSTM: & LSTM(300, 300) \end{cases}$$

- **Vanishing/Exploding gradients problem:** In our first version of the Elman (self-build) implementation, we kept running into vanishing gradient problems after roughly 1000 batches (Originating from the Sigmoid activation). For this reason we decided to implement residual connections (Code snippets, figure 6, 7) The original undesired output[7] looked like this:

```
[1,  1090] loss: 0.375
[1,  1100] loss: 0.411
[1,  1110] loss: 268345.881
[1,  1120] loss: nan
[1,  1130] loss: nan
[1,  1140] loss: nan
```

Figure 9: Output Gradient problems

- **Residual Connection implementation problem:** At our first attempt of implementing residual connection we ran into an interesting issue, we managed to resolve. However we still don't understand how it works. It was related to in place variable modification, disturbing the auto-gradient computations.

```python
203 class Elman_Torch(nn.Module):
204     def __init__(self):
205         super(Elman_Torch, self).__init__()
206         self.emb = nn.Embedding(num_embeddings = len(i2w), embedding_dim = 300
207         self.elman = torch.nn.RNN(input_size=300, hidden_size=300) # tanh non-
208         self.fc2 = nn.Linear(300, 2)
209 
210     def forward(self, x):
211         x = self.emb(x)
212         x_residual, hidden = self.elman(x)
213 
214         # Fails
215         x += x_residual
216 
217         # Works
218         x = x + x_residual
219 
220         x, y = torch.max(x, dim = 1) # global max pool
221         x = self.fc2(x)
222 
223         return x
224 
225 elman_torch_net = Elman_Torch()
```

Figure 10: Residual Connection Coding Issue (Line 214-218)

[7]Numbers between brackets can be interpreted as: [epoch, iteration/batch]