Course: Deep Learning **(8-12-2021)**

Submission Assignment #3: RNN's

Name: Brent van Dodewaard, Xinyu Hu, Venkat Mohit Sornapudi Netid: bdd400, xhu270, vsi410

1 Introduction

In this assignment, we will implement various types of RNN's, and use them to perform both classification and auto-regression. We have structured the assignment as a report. Instead of explicitly answering the questions of the assignment, they will be discussed when it makes sense in the report structure. In this report we discuss two different problems: classification and auto-regression. we will first discuss the classification task, and only after that the auto-regression task. The two problem are addressed by explaining the problem statement, the implemented methodology, the experiments conducted, and the results.

2 Classification: IMDb

2.1 Problem Statement

The IMDb dataset contains samples of text reviews of movies, with each being labelled as either a highly positive or a highly negative review. Our goal is to perform sentiment analysis on these samples, and classify each sample as either a negative or positive review. To achieve this, we will implement various types of RNN's, and a baseline linear NN.

For each of the neural network architectures constructed, we perform hyperparameter tuning to find the optimal combination of hyperparameters. We will then compare the performance between these architectures, given that the optimal combination of hyperparameters is used. Each of the hyperparamter combinations are evaluated using the validation dataset and compared using the obtained accuracies (metric of comparison). Finally, we will evaluate the best performing model using the optimal hyperparameters on the test set.

2.2 Methodology

Batching and padding Before we can work with our data, batching and padding will have to be applied. Padding will only have to be applied until each sentence reaches the length of the longest sample in the batch. When performing batching, we want to be sure that similarly sized sentences are batched together, because this can drastically reduce the total amount of padding needed. We implemented the following function to perform both padding and batching:

```python
def batch_and_pad(x, y, batch_size, pad_index):
    """Function which both creates batches of similar sized sentences and applies padding"""
    sorted_x, sorted_y = zip(*[x for (x,y) in sorted(zip(x, y),
                              key = lambda pair: len(pair[0]))])
    m, x_batches, y_batches = len(x), [], []
    for batch_i in range(int(np.ceil(m/batch_size))):
        batch_begin, batch_end = batch_i * batch_size, (batch_i+1) * batch_size
        batch_end = -1 if batch_end > m else batch_end

        # Longest sample occuring in batch
        max_batch_size = len(max(sorted_x[batch_begin:batch_end], key = lambda x: len(x)))

        # Create padded batches
        x_batch = [sample + [pad_index]*(max_batch_size-len(sample))
            for sample in sorted_x[batch_begin:batch_end]]
        y_batch = sorted_y[batch_begin:batch_end]
        x_batches.append(torch.tensor(x_batch, dtype=torch.long))
        y_batches.append(torch.tensor(y_batch, dtype=torch.long))
    return x_batches, y_batches
```

The models We have implemented a total of 4 different NN's for this task: A baseline linear NN, a self implemented Elman RNN, a Pytorch Elman RNN, and a LSTM NN. The structure of the baseline network can be found in table 1. For the Elman RNN's, we replace layer 2 in the baseline NN with an Elman layer, whose structure can be found in table 2. Note that for the Pytorch implementation of the Elman layer, the second linear layer is not included, which we thus had to add ourselves. For the LSTM NN, we replaced layer 2 in the baseline NN with a single layered LSTM layer, with a hidden size of 300. Code for the different models and self-implemented Elman layer can be found in Appendix 4.1.1.

Table 1: Baseline NN: Network structure

Layer	**Output shape**
0. Input	Batch, Time
1. Embedding	B, T, Emb(300)
2. Linear1	B, T, Hidden(300)
3. ReLu	B, T, H(300)
4. Global Max	B, H(300)
5. Linear2	B, N_classes(2)

Table 2: Elman Layer: structure

Layer	**Output shape**
0. Input	Batch, Time, Emb(300)
1. Concat(input, $TanH_{T-1}$)	B, T, (E(300) + Hidden(300))
1. Linear1	B, T, H(300)
2. TanH	B, T, H(300)
3. Linear2	B, T, H(300)

During the Deep Learning lecture, it was described that the hidden layer in the Elman layer uses the sigmoid activation function. However, the sigmoid function is not centered, which means it is suboptimal to use in the middle of a neural network. Since Pytorch uses the TanH activation function by deafult, we chose to also use this as the activation function in our own Elman implementation. We have used the following code to implement the Elman Layer:

```
class ElmanLayer(nn.Module):
    """An Elman layer which contains two linear layers,
    the first of which we add the hidden layer of t-1 to"""
    def __init__(self, insize=300, outsize=300, hsize=300):
        super().__init__()
        self.lin1, self.lin2 = nn.Linear(2 * insize, hsize), 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).to(device)

        outs = []
        for i in range(t):
            inp = torch.cat([x[:, i, :], hidden], dim=1)
            hidden = F.tanh(self.lin1(inp))
            out = self.lin2(hidden)
            outs.append(out[:, None, :])
        return torch.cat(outs, dim=1), hidden
```

Hyperparameter Tuning Since we want to make a fair comparison between the different types of models we have implemented for this task, we will perform hyperparameter tuning on them. We have selected the amount of epochs and the learning rate as the hyperparameters we want to tune. Since these two could influence the effect of the other, we will perform grid-search to find the optimal combination of learning rate and the amount of epochs. Note that we will only be performing hyperparameter tuning on the Pytorch version of the Elman RNN. Implementing the Elman layer ourselves was only for educational purposes. We will only be testing our self-implemented Elman RNN for a short training run, to show that it is functioning correcltly.

2.3 Experiments

The hyperparameter tuning experiment will be performed on 3 models as previously described: a linear baseline model, an Elman model, and a LSTM model. We will be using a stochatic gradient descent (SGD) optimizer with momentum ($\beta = 0.9$). A batch size of 128 will be used. We are training on the IMDb dataset, which contains 20000 training samples, 5000 validation samples, and 25000 test samples. Cross entropy will be used as the loss function.

We will perform grid search using the following values for the learning rate: [0,3, 0.1, 0.03, 0.01, 0.003, 0.001, 0.003], and the following values for the amount of epochs: [1, 3, 7, 10, 20, 30, 40, 50, 60, 70, 80, 90]. This means we will have a total of 84 experiment results for each network. The networks will be evaluated using the validation accuracy. We will compare the best result for each network, and finally test our overall best performing network and optimal hyperparameters on the test set. We expect that the models will be in the following order of performance from low to high: Baseline Linear model, Elman model, and LSTM model.

We will also be testing our own implementation of the Elman Layer. We will be using a stochatic gradient descent (SGD) optimizer with momentum ($\beta = 0.9$), and a batch size of 128. The learning rate we will use is 0.01. We will train our model on the data for a total of 5 epochs. We will plot both the training and validation loss curves to see if the model is learning properly. We will also test the validation accuracy after each epoch. It should have a validation accuracy after the first epoch that is higher than 60%, which is indicated in the assignment.

2.4 Results and discussion

Validation accuracy results for the hyperparameter grid-search for the different models can be found in figure 1. The highest validation accuracy for the baseline model was 87.31%, 89.32% for the Elman model, and 88.83% for the LSTM model. Both of the models which take advantage of the fact that our data is a sequence, perform better than the baseline model, which we expected. However, the Elman model performed better than our more complex LSTM model, which we did not expect. If we look at figure 1 (c), we can see that most optimal validation accuracies are already reached early on in the maximum amount of epochs. As LSTM is the most complex model of the three, it might be the case that it is overfitting. Plotting the training and validation scores could help us diagnose if overfitting is occurring, and regularization methods might then help LSTM perform better than the Elman model. Another explanation might be that we have not finetuned the hyperparameters well enough. Tuning either the momentum β term, or the amount of LSTM layers may also help.

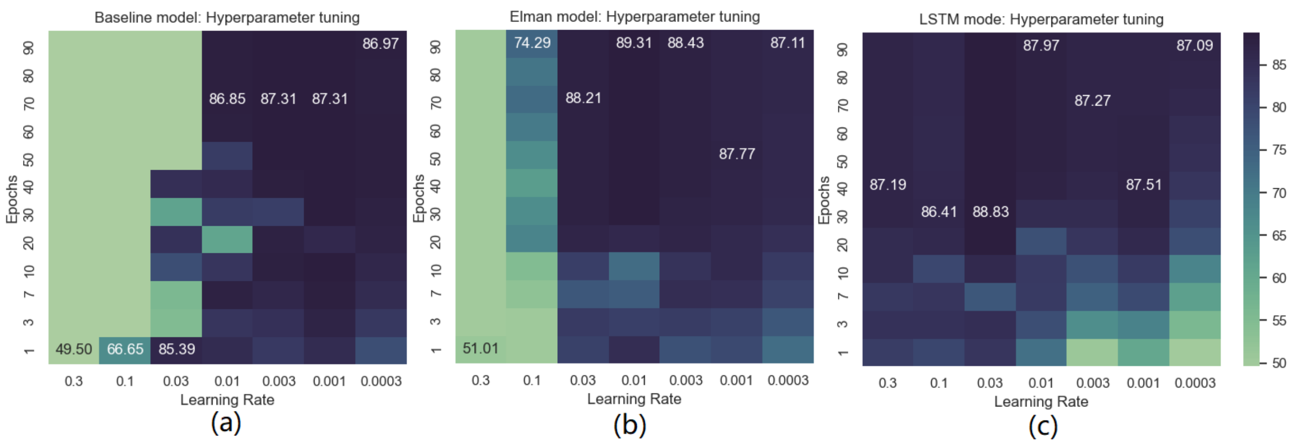

Figure 1: Validation accuracy results for the grid search of the hyperparameters learning rate and amount of epochs for the models: (a): Baseline Linear model, (b): Elman model, and (c): LSTM model. The best validation accuracy for each learning rate has its validation accuracy annotated by text.

Our best performing model is the Elman model, with a learning rate of 0.01, and using 70 epochs. We tested this model and hyperparameter combination on the test-set. This resulted in a final test-set accuracy of 89.92%.

The validation accuracy after each epoch for our self implemented Elman model can be found in figure 2 (a). A plot of the training and validation loss curve can be found in figure 2 (b). The validation accuracy after the first epoch is 80.61%, and the highest reached accuracy is 86.75%. The learning and validation curve seem to indicate functional learning, and the final validation accuracy is also relatively high. We conclude our Elman layer is properly implemented, and able to predict sequential data. When we compare the training and validation loss curves in figure 2 (b), we can however see that the difference betwee the two seems to grow as iterations progress. In the last epoch, validation accuracy and loss even go slightly down. This might be an indication overfitting is happening, which could potentially be remedied by applying some kind of regularization.

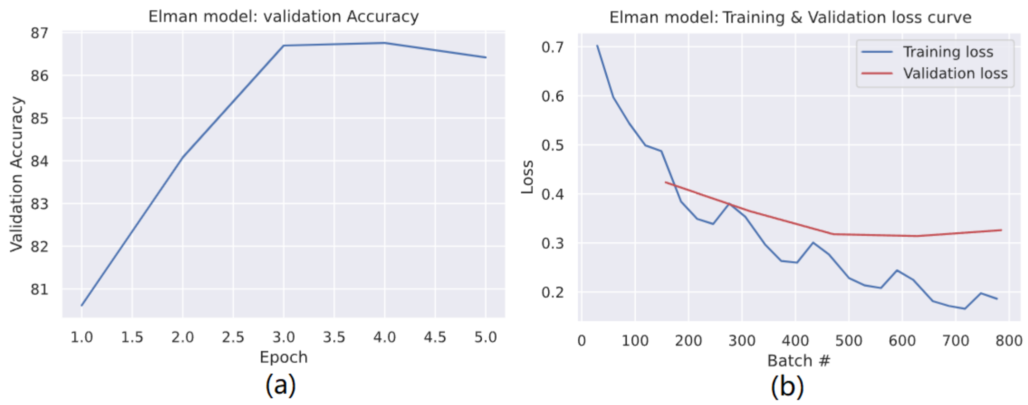

Figure 2: (a): Validation accuracy curve for the self-implemented Elman layer model. (b): Training and validation loss curves for the self-implemented Elman layer model.

3 Auto-regression: NDFA & Brackets

3.1 Problem Statement

An autoregressive model is a sequence to sequence model, that is able to predict the next token in a sequence given all the tokens before that. In this assignment, we will first implement such an autoregressive model by creating a LSTM Neural Network. We will then attempt to train this model, and use sampling from the model output distribution to create new samples from a seed. Given that the model is trained reasonably well, the created samples should be valid with regards to the type of dataset.

We will train our autoregressive model on two different datasets, creating a certain amount of model output samples after each epoch. These samples will be both automatically analyzed for correctness, and manually inspected by us. We will also be looking at the loss and gradient norm curves during training, to see if training is progressing correctly.

3.2 Methodology

Batching and padding Before we can work with our data, batching and padding will have to be applied. We also have to add the .start token and .end token to the beginning and ending of each sample. When performing batching, we want to be sure that similarly sized sentences are batched together, because this can drastically reduce the total amount of padding needed. To optimize the maximum batch size we can use, we also want to make sure that each batch contains the same amount of total tokens, and not the same amount of samples. After creating the batches, we also want to shuffle the batch order. This is done to avoid catastrophic forgetting, which is a common occuruce in LSTM's, where the model forgets what it has learned about long sequences after not seeing one for a while.

Creating the labels Our datasets do not contain labels by default, we have to create these ourselves. We achieve this by taking the training samples, shitfing them on token to the left, and adding a zero at the end of the sentence.

We implemented the following function to perform padding, batching, shuffling and label creation:

```python
def batching_padding_static_total_size(x, batch_size, pad_index, start_index, end_index):
    """Function which creates same sized batches, applies padding, and creates labels"""
    m, y = len(x), x.copy()
    sorted_x = [x for (x,y) in sorted(zip(x, y), key = lambda pair: len(pair[0]))]
    sorted_y = [y for (x,y) in sorted(zip(x, y), key = lambda pair: len(pair[0]))]

    x_batches, y_batches = [], []
    x_batch_current, y_batch_current = [], []
    current_batch_size = 0
    for x, y in zip(sorted_x, sorted_y):
        x_batch_current.append(x)
        y_batch_current.append(y)
        # +2 because we want to add .start and .end to the sentence too
        current_batch_size += len(x) + 2

        # If we have reached batch_size, create actual batch
        if current_batch_size > batch_size:
            # Longest sample occuring in batch
            max_batch_size = len(max(x_batch_current, key = lambda x: len(x)))

            # Create padded batchec
            x_batch = [sample + [pad_index]*(max_batch_size-len(sample))
                       for sample in x_batch_current]
            x_batch = [[start_index] + sample + [end_index] for sample in x_batch]

            y_batch = [sample + [pad_index]*(max_batch_size-len(sample))
                       for sample in y_batch_current]
```

```
                y_batch = [sample + [end_index] + [0] for sample in y_batch]
                x_batches.append(torch.tensor(x_batch, dtype=torch.long))
                y_batches.append(torch.tensor(y_batch, dtype=torch.long))

                # Reset current_batch
                x_batch_current, y_batch_current = [], []
                current_batch_size = 0

        # Shuffle the batches to counter catastrophic forgetting.
        x_and_y = list(zip(x_batches, y_batches))
        random.shuffle(x_and_y)
        x_batches, y_batches = zip(*x_and_y)
        return x_batches, y_batches
```

Padding masking We have also implemented masking of the padding tokens when calculating the loss. This is because a metric to represent how well a model is performing, should ideally not be influenced by how well or how bad it is predicting the padding tokens. This was achieved by giving each other token in the vocabulary a loss weight of 1, and the padding token loss weight of 0.

The model For this assignment, we have implemented a single LSTM NN. The structure of the model can be found in table 3. We have chosen to use an embedding size of 32, a hidden layer size of 16, and a single layer for the LSTM. The output of the model will be the size of the vocabulary. When softmaxed, this output represents the predicted probability that each token will occur next. Code for the implementation of this model can be found in appendix 4.1.2.

Table 3: Autoregressive LSTM NN:
Network structure

Layer	Output shape
0. Input	Batch, Time
1. Embedding	B, T, Emb(32)
2. LSTM(numlayers=1)	B, T, Hidden(16)
3. Linear	B, T, Vocab_size

Model Sampling Instead of using a test/train split, we will sample instances from our model, inspect them, and determine their correctness. We will do this after an epoch, to determine how well the model is predicting the next tokens. Sampling is performed by starting with a starting seed sequence, which contains a few seed tokens start from. We will feed this sequence to our model, and sample a new token from the probabilities our model predicts. We append this new token to our seed sequence, and repeat the process until either an end token is sampled, or we reach a certain max length. We perform sampling by using the function as is defined in the assignment. We have decided to use a temperature of 1 for this function. This means we will be sampling directly using the probabilities as predicted by our model. We realise we are making the task of predicting correct sentences more difficult by doing this, but we think this will more fairly show the quality of the predictions of our model than using a lower temperature.

Determining sample correctness While we will also inspect the samples by hand, we also want a function which can automatically determine the correctness of samples. We have implemented such functions for both the Bracket dataset, and the NDFA datset. We will go over how these are implemented.

The NDFA dataset is sampled from a non-deterministic finite automaton. This means we can simply determine the possible next tokens for each token, and see if a sample is valid with regards to the legal transition. Something important to note, is that we found that the schematic figure of the automaton in the assignment is actually not fully correct. By inspecting the code of the dataset generation, we determined that there is a non-zero chance of an instant transition from the first s to the last s, something that is not represented in the given figure. We used the following code to determine if a sample is correct for the NDFA dataset:

```
def sample_is_correct_NDFA(sample):
    legal_transitions = {
     "s": ["s", ".end", "a", "k", "u"], ".start": ["s"], "a" : ["b"], "b": ["c"],
     "c": ["!"], "!": ["s", "a", "k", "u"], "k": ["l"],
     "l": ["m"], "m": ["!"], "u": ["v"], "v": ["w"], "w": ["!"]}

    for i in range(len(sample)-1):
        current_char, next_char = i2w[sample[i]], i2w[sample[i+1]]
        if next_char not in legal_transitions[current_char]:
            return False
    return True
```

The Brackets dataset contains samples from sequences containing "(" and ")", which are correctly matched together. This means we can simply determine the correctness of a sample by determining if the same amount of "(" and ")" occur in a sample. We used the following code to determine if a sample is correct for the Brackets dataset:

```
def sample_is_correct_brackets(sample):
    return (sample.count(w2i['(']) - sample.count(w2i[')'])) == 0
```

Training diagnostics During the training of our model, we will be plotting both the loss curve and gradient norm curve to determine if our training is progressing well. The gradient norm curve will be the sum of all norms (l2) of gradients in the network, and can determine if an exploding gradient is occurring. We calculate it using the following functions:

```
total_norm = 0
for p in net.parameters():
    param_norm = p.grad.detach().data.norm(2)
    total_norm += param_norm.item() ** 2
total_norm = total_norm ** 0.5
```

3.3 Experiments

The experiment will be perfomed on the LSTM model as previously described, using 1 LSTM layer, an embedding size of 32, and a hidden layer size of 16. We will be using a stochatic gradient descent (SGD) optimizer with momentum ($\beta = 0.9$), and a (token) batch size of 5000 will be used. We are training on both the Bracket and NDFA dataset, and we generate 150.000 training samples for each dataset. We chose to use a learning rate of 0.03. This was found as the optimal learning rate for the LSTM model in the previous classification problem. While we realise that this does not necessarily mean that it will also be the optimal learning rate for this problem, it should be a decent starting point. We will use per token cross entropy as the loss function.

We will train on both the Bracket and NDFA dataset for 50 epochs, and generate 100 model samples after each epoch. We will plot the loss and gradient norm curves, and the amount of valid samples for each epoch. As for the manual inspection of the model samples, we will look at 10 samples during the first 5 epochs, and then finally also inspect 10 samples after the 50 epoch training has finished. The seed for the Bracket dataset will be ['.start', '(', '(', ')'], and the seed for the NDFA dataset will be ['.start','s']. The maximum length we will use for the samples, after which we stop sampling without ever encountering an end token, is 100.

Since the datasets are very simple, we predict that our model will be able to predict the sequences relatively well. This means we predict that most sequences sampled from our model should be valid. In the assignment, it is stated that the learning is considered successful if in a sample of 10 sequences, the majority is correct. However, we thus expect the model to perform considerably better than just having a sample validity accuracy of $> 50\%$.

3.4 Results and discussion

NDFA The training loss curve for the NDFA experiment can be found in figure 3a, the gradient norm curve in figure 3b, and the sample validity accuracy of the samples can be found in 3c. The model samples which were generated during each epoch can be found in appendix 4.2.1.

The training seems to have progressed well. The Training loss curve seems to converge fairly early in the training phase. The gradient norms are within acceptable values during the entire training run, so no exploding gradient seems to occur. The validity percentage of our samples seems to grow very quickly, the majority of samples are already valid after the first epoch. At the end of the training phase, almost all generated samples seem to be correct.

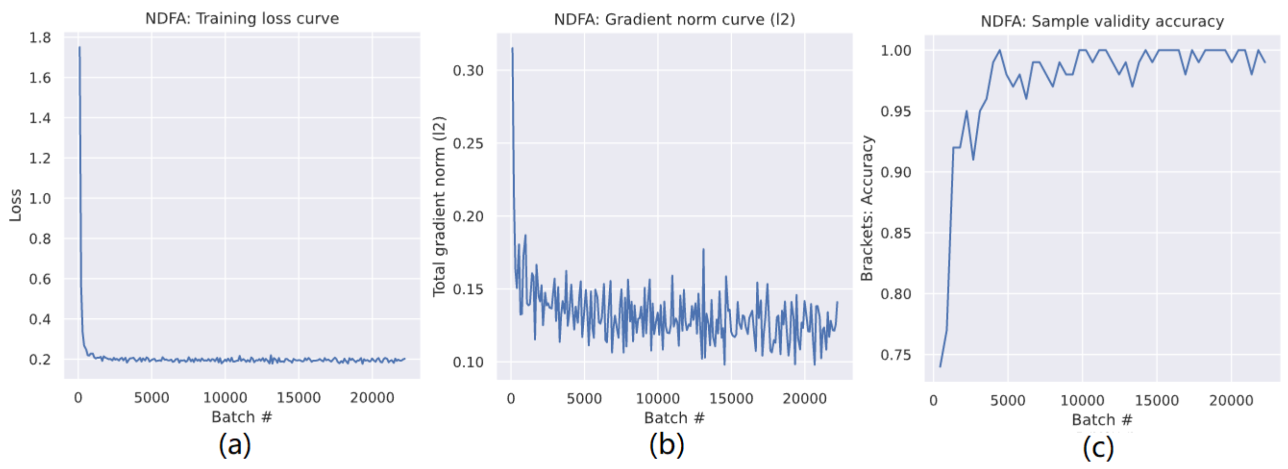

Figure 3: (a): The training loss curve for the NDFA experiment. (b): The gradient norm curve for the NDFA experiment. (c): The sample validity accuracy for the NDFA experiment during the training phase.

When manually inspecting the model samples found in appendix 4.2.1, we can see that there seems to be both a high diversity in sentence length and tokens used. This increases as the epochs progress. We find the quality of the samples to be overall high. Interestingly, we noted in the Methodology section that the s token could occur after the s token, which was not indicated in the assignment. Our own model also seems to predict this pattern fairly frequently, especially early on in the training phase.

Brackets The training loss curve for the Brackets experiment can be found in figure 4a, the gradient norm curve in figure 4b, and the sample validity accuracy of the samples can be found in 4c. The model samples which were generated during each epoch can be found in appendix 4.2.2.
Again, the training seems to have progressed well. The Training loss curve seems to converge fairly early in the training phase, just like in the previous experiment. The gradient norms are within acceptable values during the entire training run, so no exploding gradient seems to occur. The validity percentage of our samples seems to grow very quickly, the majority of samples are already valid after the first epoch. While at the end of the training phase, almost all generated samples seem to be correct, there is quite a lot of variance in the proportion of samples it got correct for each epoch. This is even true for the last few epochs.

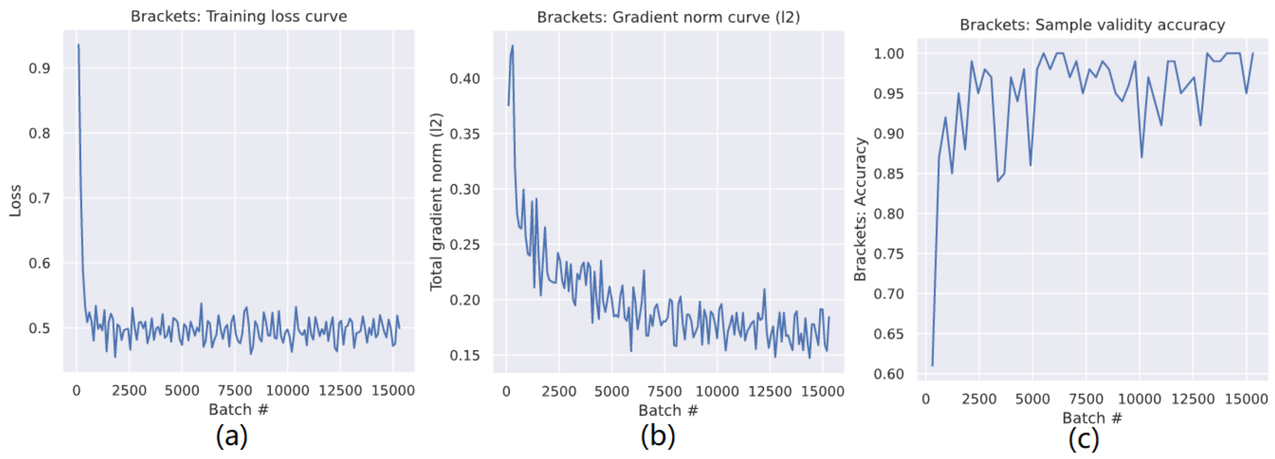

Figure 4: (a): The training loss curve for the Brackets experiment. (b): The gradient norm curve for the Brackets experiment. (c): The sample validity accuracy for the Brackets experiment during the training phase.

When manually inspecting the model samples found in appendix 4.2.2, we can see that while longer samples with diverse patterns sometimes get sampled, we also get a lot of samples which are the same: (”(( ))”). We don't think this is a problem with our model, but rather just a intrinsic property of the dataset in combination with our seed. As a seed we chose (”(( )”). This means if the character ”)” gets predicted next, the sentence is already valid, and our model will predict that it should be closed. This means a large fraction of our samples will already be closed after the first sample. A potential solution to this would be to either use a seed with a bigger gap between the occurrence of ”(and ”)”, or to use a random sequence as the seed, randomly generated each time.

3.4.1 Conclusion

In conclusion, the results of our experiment were excellent. The loss and gradient norm curves during training indicate that the training progressed well. This is confirmed by the great model sample validity percentage we achieve for both datasets. The generated samples by our model also show a relatively high amount of variability between samples, which is desirable. There are three specific things we think we could have potentially improved. The training seems to converge after just a few epochs for both datasets. Choosing a lower learning rate, might have slowed down training a bit, but could have potentially led to a improved final model. We also concluded that some of the samples which were found to be invalid, were only invalid because the maximum size of the samples we used made them invalid. This leads to a improper representation of the sample validity. We expect that a higher maximum sample size might solve this. An idea for this, is to use the maximum sample size as found in the dataset as the maximum sample size you will generate. The last potential improvement we would make, is to use randomly generated seeds to generate new model samples. This would promote diversity in both tokens and sample length.

4 Appendix

4.1 Code

4.1.1 Pytorch models: Classification

Code for the Pytorch models as used for the Classification problem

```
class BaselineNet(nn.Module):
    """The baseline model, which does not use the sequence order at all"""
    def __init__(self):
        super().__init__()

        self.name = "BaselineNet"
        self.embedding = nn.Embedding(len(i2w), 300)
        self.linear1 = nn.Linear(300, 300)
        self.linear2 = nn.Linear(300, numcls)

    def forward(self, x):
        x = self.embedding(x)
        x = F.relu(self.linear1(x))
        x, inds = torch.max(x,dim=1)
        x = self.linear2(x)
        return x
```

```
class TorchElmanNet(nn.Module):
    """ElmanNet implemented using the Elman layer of pytorch"""
    def __init__(self):
        super().__init__()

        self.name = "TorcElmanNet"
        self.embedding = nn.Embedding(len(i2w), 300)

        self.rnn= nn.RNN(300,300,num_layers=1, batch_first=True)
        self.linear1 = nn.Linear(300, 300)
        self.linear2 = nn.Linear(300, numcls)

    def forward(self, x):
        x = self.embedding(x)
        x, hidden = self.rnn(x)
        x = F.relu(self.linear1(x))
        x, inds = torch.max(x,dim=1)
        x = self.linear2(x)
        return x
```

```
class OwnElmanNet(nn.Module):
    """The ElmanNet we implemented ourselved"""
    def __init__(self):
        super().__init__()
        self.embedding = nn.Embedding(len(i2w), 300)
        self.linear2 = nn.Linear(300, numcls)
        self.elman= ElmanLayer()

    def forward(self, x):
        x = self.embedding(x)
        x, hidden = self.elman(x)
        x = F.relu(x)
        x, inds = torch.max(x,dim=1)
        x = self.linear2(x)
        return x

class ElmanLayer(nn.Module):
    """A single Elam layer, consists of two linear layers, the first of which -
    we concantenate the hidden layer of t-1 to"""
    def __init__(self, insize=300, outsize=300, hsize=300):
        super().__init__()
        self.lin1 = nn.Linear(2 * insize, 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).to(device)

        outs = []
        for i in range(t):
            inp = torch.cat([x[:, i, :], hidden], dim=1)
            hidden = F.tanh(self.lin1(inp))
            out = self.lin2(hidden)
            outs.append(out[:, None, :])
        return torch.cat(outs, dim=1), hidden
```

```
class TorchLSTM(nn.Module):
    """ElmanNet implemented using the LSTM layer of pytorch"""
    def __init__(self):
        super().__init__()

        self.name = "TorchLSTM"
        self.embedding = nn.Embedding(len(i2w), 300)

        self.rnn= nn.LSTM(300,300,num_layers=1, batch_first=True)
        self.linear1 = nn.Linear(300, 300)
        self.linear2 = nn.Linear(300, numcls)

    def forward(self, x):
        x = self.embedding(x)
        x, hidden = self.rnn(x)
        x = F.relu(self.linear1(x))
        x, inds = torch.max(x,dim=1)
        x = self.linear2(x)
        return x
```

4.1.2 Pytorch models: autoregression

Code for the Pytorch models as used for the autoregression problem:

```
class AutoregLSTM(nn.Module):
    """ElmanNet implemented using the LSTM layer of pytorch"""
    def __init__(self, num_LSTM_layers = 1 ):
        super().__init__()

        self.name = "AutoregLSTM"
        self.num_LSTM_layers = num_LSTM_layers

        self.embedding = nn.Embedding(len(i2w), 32)
        self.rnn= nn.LSTM(32, 16, num_layers=self.num_LSTM_layers, batch_first=True)
        # The last layer always need to have output size of vocab
        self.linear_final = nn.Linear(16, len(i2w))

    def forward(self, x):
        x = self.embedding(x)
        x, hidden = self.rnn(x)
        x = self.linear_final(x)
        return x
```

4.2 Autoregressive samples

4.2.1 Samples: NDFA

The following samples were generated using our model by training on the NDFA dataset:

```
Epoch 1, Samples correct: 74
.start s .end
.start s k l m ! s .end
.start s u v w ! u v w ! u v w ! u v w ! s .end
.start s s a b c ! s .end
.start s .end
.start s s a b ! s .end
.start s s .end
.start s s .end
.start s .end
.start s k l m ! s .end

Epoch 2, Samples correct: 77
.start s u v w ! u v w ! u v w ! s .end
.start s k l m ! k l m ! k l m ! k l v w ! s .end
.start s s .end
.start s s .end
.start s s .end
.start s a b c ! a b c ! a b c ! a b c ! a b c ! k l m ! s .end
.start s s .end
.start s s .end
.start s u v w ! k l m ! k l m ! k l m ! k l m ! s .end
.start s s .end

Epoch 3, Samples correct: 92
.start s s .end
.start s k l m ! k l m ! k l m ! s .end
.start s a b c ! s a b c ! a b c ! a b c ! a b c ! s .end
.start s k l m ! k l m ! k l m ! s .end
.start s u v w ! u v w ! u v w ! u v w ! s .end
.start s k l m ! k l m ! s .end
.start s a b c ! a b c ! a b c ! a b c ! a a b c ! a b c ! a a b c ! a b c ! a b c ! a b c
    ! s .end
.start s u v w ! u v w ! s .end
.start s a b c ! a b c ! s .end
.start s k l m ! s .end

Epoch 4, Samples correct: 92
.start s k l m ! s .end
.start s k l m ! k l m ! k l m ! k l m ! k l m ! s .end
.start s a b c ! s .end
.start s k l m ! k l m ! s .end
.start 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 ! s .end
```

```
.start s u v w ! s .end
.start s a b c ! a b c ! a b c ! a b c ! s .end
.start s s .end
.start s u v w ! u v w ! s .end
.start s k l m ! k l m ! k l m ! k l m ! a b c ! a b c ! a b c ! s .end

Epoch 5, Samples correct: 95
.start s k l m ! k l m ! k l m ! s .end
.start s s .end
.start s s .end
.start s k l m ! s .end
.start s u v w ! s .end
.start s u v w ! s .end
.start s a b c ! a b c ! s .end
.start s s .end
.start s k l m ! s .end
.start s s .end
Samples correct: 95

Epoch 50, Samples correct: 99
.start s k l m ! k l m ! s .end
.start s s .end
.start s k l m ! s .end
.start s u v w ! u v w ! u v w ! u v w ! u v w ! u v w ! u v w ! s .end
.start s s .end
.start s a b c ! a b c ! a b c ! a b c ! a b c ! a b c ! a b c ! s .end
.start s s .end
.start s a b c ! a b c ! a b c ! a b c ! a b c ! a b c ! a b c ! s .end
.start s k l m ! k l m ! k l m ! k l m ! s .end
.start s k l m ! s .end
```

4.2.2 Samples: Brackets

The following samples were generated using our model by training on the Brackets dataset:

```
Epoch 1, Samples correct: 49/100
.start ( ( ) ( ) ) .end
.start ( ( ) ) .end
.start ( ( ) ( ) ( ( ( ( ( ( ) ( ( ) ) ( ( ( ) ( ( ) ) ( ) ( ( ) ) ) ) ( ( ( ( ) ( ( ( ) (
    ( ( ( ) ) ) ( ( ) ( ( ( ) ) ( ( ( ) ( ( ( ( ( ( ( ) ) ( ) ( ) ( ) ( ) ( ) ) ) ) ( ( (
    ( ) ( ) ) ) ( ) ( ( ) ( ) (
.start ( ( ) ( ( ( ( ( ( ) ) ) ) ) ( ( ) ( ) ) ( ) ) ( ) ) ) ) .end
.start ( ( ) ) ) .end
.start ( ( ) ( ) ( ( ) ( ) ( ( ) ( ) ) ( ) ) ) ( ) ( ) ) ) ( ( ) ( ( ( ) ) ) ) ( ( ( ) (
    ) .end
.start ( ( ) ( ( ) ) ) ) .end
.start ( ( ) ) .end
.start ( ( ) ) ( ) ) .end
.start ( ( ) ) .end

Epoch 2, Samples correct: 85/100
.start ( ( ) ( ) ) ( ) .end
.start ( ( ) ( ( ) ) ( ( ) ) ) .end
.start ( ( ) ( ) ) .end
.start ( ( ) ) .end
.start ( ( ) ( ) ( ) ( ( ( ) ( ( ) ( ) ) ( ) ( ) ) ( ( ( ( ( ) ) ) ( ( ( ) ( ) ) ) ( ) )
    ) ( ) ) ( ( ) ) ( ) ) .end
.start ( ( ) ( ( ( ) ( ( ( ) ( ( ( ( ( ) ( ( ( ( ) ) ) ( ( ) ) ) ) ( ) ( ( ) ( ( ( ) ) )
    ( ( ) ( ) ) ( ) ) ( ) ) ( ) ( ) ) ( ( ( ) ( ) ) ) ) ( ( ) ( ( ) ) ) ) ) ) ( ( ) ( ) )
    ) ) ) .end
.start ( ( ) ( ) ( ) ) .end
.start ( ( ) ) .end
.start ( ( ) ( ( ( ) ) ( ) ( ( ) ) ( ( ( ( ) ) ( ( ) ) ) .end
.start ( ( ) ( ) ( ) ) .end

Epoch 3, Samples correct: 90/100
.start ( ( ) ) .end
.start ( ( ) ( ) ) .end
.start ( ( ) ( ( ( ) ( ( ) ) ( ) ( ) ) ) ( ( ) ( ( ) ( ) ( ) ) ) ( ) ( ) ) ( ( ) ( ) ( )
    ( ) ) ( ) ( ( ) ) ( ) ( ) ) .end
.start ( ( ) ( ( ( ( ( ( ( ) ) ) ( ) ) ( ) ) ( ( ( ) ( ) ) ) ( ) ( ( ) ) ( ( ) ) ) ) ) .end
.start ( ( ) ) .end
.start ( ( ) ( ( ( ( ( ) ( ) ( ( ( ) ) ( ( ) ) ( ( ) ) ) ( ) ( ) ) ) ( ) ) ( ) ( ) ( ( ) (
    ) ) ( ) ( ) ( ( ) ) ( ( ( ( ( ) ( ) ( ( ) ) ) ) ( ) ) ( ( ( ) ( ( ) ( ( ( ( ) ) ( ) ) )
    ( ) ) ) ( ) ( ( ) ( ( ) (
.start ( ( ) ( ) ) .end
.start ( ( ) ) .end
.start ( ( ) ) .end
.start ( ( ) ) .end

Epoch 4, Samples correct: 91/100
.start ( ( ) ( ) ) .end
.start ( ( ) ( ) ) .end
.start ( ( ) ) .end
.start ( ( ) ) .end
.start ( ( ) ) .end
.start ( ( ) ( ( ) ( ( ) ) ) ) .end
.start ( ( ) ( ( ) ( ) ) ) .end
.start ( ( ) ) .end
.start ( ( ) ) .end
.start ( ( ) ( ) ) .end

Epoch 5, Samples correct: 94/100
.start ( ( ) ) .end
.start ( ( ) ) .end
.start ( ( ) ( ) ( ) ) .end
.start ( ( ) ) .end
.start ( ( ) ( ) ) .end
.start ( ( ) ) .end
.start ( ( ) ) .end
.start ( ( ) ( ) ) .end
.start ( ( ) ) .end
.start ( ( ) ) .end

Epoch 50, Samples correct: 94/100
.start ( ( ) ( ( ( ) ( ( ( ) ( ) ) ( ) ) ( ) ( ) ) ) ( ) ) ( ( ) ) ( ( ) ( ( ( ) ) ( ) (
    ( ( ( ) ) ( ) ) ) ( ) ) ) ) .end
.start ( ( ) ( ( ) ) ( ( ( ( ( ( ) ) ) ) ( ( ( ( ( ) ( ) ( ) ( ) ( ) ( ( ) ( ( ) ( ( ) ( )
    ( ( ) ) ( ( ) ) ) ( ( ) ) ) ( ( ) ( ) ) ( ) ( ( ( ( ( ) ( ( ( ( ( ) ( ( ( ) ) ) ) )
    ( ( ( ( ) ) ( ) ) ) ( ( ( )
.start ( ( ) ( ) ) .end
.start ( ( ) ) .end
.start ( ( ) ( ) ) .end
.start ( ( ) ( ( ) ) ( ( ) ) ) .end
.start ( ( ) ) .end
.start ( ( ) ( ) ( ) ) .end
```

```
.start ( ( ) ) .end
.start ( ( ) ) .end
```