Assignment 3

Xiyan Liu (2815678)
Qiang Zhao (2814236)
Mehaa Prababakar (2817950)

December 2023

Question 1

Implement this padding and conversion. Show the function in your report.

```
import torch
from data_rnn import load_imdb

PAD, START, END, UNK = '.pad', '.start', '.end', '.unk'

def pad_and_convert(sequences, w2i, add_start_end_tokens=False):
    pad_token_idx = w2i[PAD]

    # index
    if add_start_end_tokens:
        start_token_idx = w2i[START]
        end_token_idx = w2i[END]
    else:
        start_token_idx = None
        end_token_idx = None

    # find max
    max_length = max(len(seq) for seq in sequences)

    # pad to max
    padded_sequences = [
        seq[:max_length] + [pad_token_idx] * max(0, max_length - len(seq)) for seq
        in sequences
    ]

    # convert
    padded_sequences_tensor = torch.tensor(padded_sequences, dtype=torch.long)
    if add_start_end_tokens:
        # start
        start_tokens = torch.full((len(padded_sequences), 1), start_token_idx,
        dtype=torch.long)
        padded_sequences_tensor = torch.cat([start_tokens, padded_sequences_tensor], dim=1)

        # end
        end_tokens = torch.full((len(padded_sequences), 1), end_token_idx,
        dtype=torch.long)
        padded_sequences_tensor = torch.cat([padded_sequences_tensor, end_tokens], dim=1)

    return padded_sequences_tensor

# load the data
(x_train, y_train), (x_val, y_val), (i2w, w2i), _ = load_imdb(final=False, val=5000,
seed=0, voc=None, char=False)

# train data
padded_x_train = pad_and_convert(x_train, w2i, add_start_end_tokens=True)

# validation data
padded_x_val = pad_and_convert(x_val, w2i, add_start_end_tokens=True)

# to check
print(len(x_train))
print(len(y_train))
print(padded_x_train.size())
print(torch.tensor(y_train, dtype=torch.long).size())
```

After padding and tensor converting, let's check the size of x_train and y_train to prepare for subsequent training:

```
2000 #the size of x_train
20000 #the size of y_train
torch.Size([20000, 2516]) #the shape of x_train after padding and tensor converting
torch.Size([20000]) #the shape of y_train after tensor converting
```

It can be concluded that our function is indeed effective and keeps the size of the two the same.

Question 2

Build a model.

```
class CustomModel(nn.Module):
```

```
def __init__(self, num_tokens, embedding_size=300, hidden_size=300, num_classes=2):
    super(CustomModel, self).__init__()

    # 1.Embedding Layer
    self.embedding = nn.Embedding(num_tokens, embedding_size)

    # 2.Linear Layer
    self.linear = nn.Linear(embedding_size, hidden_size)

    # 3.ReLU Layer
    self.relu = nn.ReLU()

    # 5.Output Layer
    self.output = nn.Linear(hidden_size, num_classes)

    # Initialize the optimizer and loss function
    self.optimizer = optim.Adam(self.parameters(), lr=0.001)
    self.criterion = nn.CrossEntropyLoss()

def forward(self, x):
    # 1.Embedding Layer
    embedded = self.embedding(x)

    # 2.Linear Layer
    linear_out = self.linear(embedded)

    # 3.ReLU Layer
    relu_out = self.relu(linear_out)

    # 4.Global max pooling
    pooled = torch.max(relu_out, dim=1).values

    # 5.Output Layer
    output = self.output(pooled)

    return output

def train_model(self, train_loader, criterion, optimizer, num_epochs=1):
    model.train()
    for epoch in range(num_epochs):
        for batch_inputs, batch_labels in train_loader:
            optimizer.zero_grad()
            outputs = model(batch_inputs)
            loss = criterion(outputs, batch_labels)
            loss.backward()
            optimizer.step()
        print(f'Epoch {epoch + 1}/{num_epochs}, Loss: {loss.item()}')

def evaluate(self, val_loader):
    self.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for val_inputs, val_labels in val_loader:
            val_outputs = self(val_inputs)
            _, predicted = torch.max(val_outputs.data, 1)
            total += val_labels.size(0)
            correct += (predicted == val_labels).sum().item()

    accuracy = correct / total
    print(f'Validation Accuracy: {accuracy * 100:.2f}%')
```

(1)How can you find this from the return values of the load_imdb function?

The number of tokens (vocabulary size) can be obtained by finding the length of the i2w or w2i dictionary.

```
print(len(i2w))
```

The result will be **99430**.

(2)The model is not memory-intensive, so you can easily go to large batch sizes to speed up training. Note, however, that the amount of padding also increases with the batch size. Can you explain why?

Increasing the batch size improves training speed because more examples can be processed at each model update step. However, the variable length nature of text sequences results in shorter sequences requiring padding to match the longest sequence length. As the batch size increases, the maximal length of the sequence within the batch increases, and so does the fill quantity.

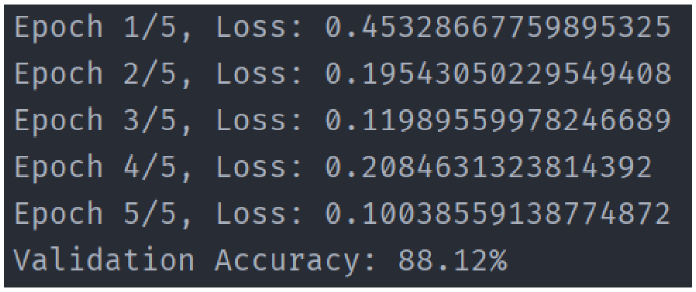
Question 3

Train for at least one epoch and compute the validation accuracy. It should be above 60% for most reasonable hyperparameters.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from data_rnn import load_imdb
```

```
class CustomModel(nn.Module):
    def __init__(self, num_tokens, embedding_size=300, hidden_size=300, num_classes=2):
        # ... (part of the previous part)

    def train_model(self, train_loader, criterion, optimizer, num_epochs=1):
        # ... (part of the previous part)

    def evaluate(self, val_loader):
        # ... (part of the previous part)


(x_train, y_train), (x_val, y_val), (i2w, w2i), _ = load_imdb(final=False,
val=5000, seed=0, voc=None, char=False)

PAD, START, END, UNK = '.pad', '.start', '.end', '.unk'

def pad_and_convert(sequences, w2i, add_start_end_tokens=False):
    # ... (part of the previous part)

# pad and convert training sequences
padded_x_train = pad_and_convert(x_train, w2i, add_start_end_tokens=True)

# pad and convert validation sequences
padded_x_val = pad_and_convert(x_val, w2i, add_start_end_tokens=True)

# model
model = CustomModel(num_tokens=len(i2w), embedding_size=300, hidden_size=300,
num_classes=2)

optimizer = model.optimizer
criterion = model.criterion

# create DataLoader
train_dataset = torch.utils.data.TensorDataset(padded_x_train,
torch.tensor(y_train, dtype=torch.long))
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64, shuffle=True)

val_dataset = torch.utils.data.TensorDataset(padded_x_val,
torch.tensor(y_val, dtype=torch.long))
val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=64, shuffle=False)

# trainning
model.train_model(train_loader, criterion, optimizer, num_epochs=1)

# validation
model.evaluate(val_loader)
```

Training results:
Epoch 1/1, Loss: 0.3668537139892578
Validation Accuracy: 86.78%



```
Epoch 1/5, Loss: 0.45328667759895325
Epoch 2/5, Loss: 0.19543050229549408
Epoch 3/5, Loss: 0.11989559978246689
Epoch 4/5, Loss: 0.2084631323814392
Epoch 5/5, Loss: 0.10038559138774872
Validation Accuracy: 88.12%
```

Figure 1: The loss within 5 epochs on validation data

Epoch 1/5, Loss: 0.45328667759895325

Epoch 2/5, Loss: 0.19543050229549408

Epoch 3/5, Loss: 0.11989559978246689

Epoch 4/5, Loss: 0.2084631323814392

Epoch 5/5, Loss: 0.10038559138774872

Validation Accuracy: 88.12

Question 4

Fill in the missing parts of this code. And build a second model, like the one in the previous part, but replacing the second layer with an Elman(300, 300, 300) layer.

Based on the network structure provided in the lecture materials, we designed an Elman network involving two sets of parameters, namely weights w and biases b, weights v and biases c. The primary operations include concatenating the previous layer's hidden layer with the current layer, the first linear transformation (involving weights w and biases b), sigmoid processing, and the second linear transformation (involving weights v and biases c). Although an output layer Y is produced for each current layer, we only consider the output layer of the last layer.

```
#Elman model
import torch.nn.functional as F
from torch.nn.parameter import Parameter
import torch.nn.init as init
class Elman(nn.Module):
    def __init__(self, insize, outsize, size):
        super(Elman,self).__init__()
        self.in size =insidee
        self.size = size
        self.outsize = outsize
        #Initialize the weight parameter and bias
        self.w = Parameter(torch.empty(insize+hsize,size))
        self.v = Parameter(torch.empty(size, outsize))
        self.b = Parameter(torch.zeros(size))
        self.c = Parameter(torch.zeros(outsize))
        init.xavier_uniform_(self.w)
        init.xavier_uniform_(self.v)

    def forward(self, x, hidden=None):
        b, t, e = x.size()
        if hidden is None:
            hidden = torch.zeros(b, e, dtype=torch.float)
        outs = []
        for i in range(t):
            inp = torch.cat([x[:, i, :], hidden], dim=1)
            y = F.linear(inp,self.w.T,self.b)
            hidden = F.sigmoid(y)
            out = F.linear(hidden,self.v,self.c)
            outs.append(out[:, None, :])
        return torch.cat(outs, dim=1), hidden
```

The **Elman** model was implemented using the following code:

```
def padding_tensor(input):
    max_len = max(len(i) for i in input)
    pad_input = []
    for data in input:
        for pad in range(max_len - len(data)):
            data.append(0)
        pad_input.append(data)

    tensor_input = torch.tensor(pad_input, dtype=torch.long)

    start_token, end_token = w2i['.start'], w2i['.end']
    batch_size = len(input)
    start_tokens = torch.full((batch_size, 1), start_token, dtype=torch.long)
    tensor_input_s = torch.cat([start_tokens, tensor_input], dim=1)
    end_tokens = torch.full((batch_size, 1), end_token, dtype=torch.long)
    batch_sequences_s_e = torch.cat([tensor_input_s, end_tokens], dim=1)

    return batch_sequences_s_e

class SequenceModel(nn.Module):
    def __init__(self, vocab_size, batch_size, embedding_dim, hidden_dim, num_classes):
        super(SequenceModel, self).__init__()
        # Embedding layer
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        # Elman layer (Replaced the previous linear layer)
        self.elman = Elman(batch_size,embedding_dim,hidden_dim)
        # Output layer
        self.output = nn.Linear(hidden_dim, num_classes)

    def forward(self, x):
        # Embedding layer
        x = self.embedding(x)  # (batch, time, emb)
        # Elman layer
        x, _ = self.elman(x)
        x = nnfunction.relu(x)  # ReLU activation
        # Global max pooling along the time dimension
        x, _ = torch.max(x, dim=1)  # (batch, hidden)
        # Output layer
        x = self.output(x)  # (batch, num_classes)
        return x

embedding_size = 300
hidden_size = 300
num_classes = 2
words_size = len(w2i)
batch_size = 300

model = SequenceModel(words_size,batch_size, embedding_size, hidden_size, num_classes)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
epochs = 1
batch_size = 300
rounds = int(len(x_train)/batch_size)

#Trainig dataset
for epoch in range(epochs):
    all_acc = 0
    all_loss = 0
```

```
for r in range(rounds):
    s,e = (r) * batch_size, (r + 1) * batch_size
    input_data = padding_tensor(x_train[s:e])
    output_label = torch.tensor(y_train[s:e])
    output_pred = model(input_data)
    loss = criterion(output_pred, output_label)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    pred_label = output_pred.argmax(1)
    correct = torch.eq(pred_label, output_label)
    accuracy = torch.mean(correct.float())
    all_acc += round(accuracy.item(), 3)
    all_loss += loss
print(f"Training data: epoch {epoch+1} | loss: {all_acc / rounds:.3f}
| accuracy:{all_loss / rounds:.3f}")
```

Training data: epoch 1 — loss: 0.691 — accuracy:0.594

Question 5

Tune the hyperparameters for these three models (MLP, Elman, LSTM). The following code present the relevant chages within the code for the class construction.

1. MLP.
(Already given in previous questions.)
2. Elman.

```
    # 2.RNN Layer (Elman network)
    self.rnn = nn.RNN(input_size=embedding_size, hidden_size=hidden_size,
    batch_first=True)
...
...
    # 2.RNN Layer (Elman network)
    rnn_out, _ = self.rnn(embedded)
```

3.LSTM.

```
    # 2.LSTM Layer
    self.lstm = nn.LSTM(input_size=embedding_size,
                        hidden_size=hidden_size,
                        num_layers=num_layers,
                        dropout=dropout,
                                 batch_first=True)
...
...
    # 2.LSTM Lyaer
    lstm_out, _ = self.lstm(embedded)
```

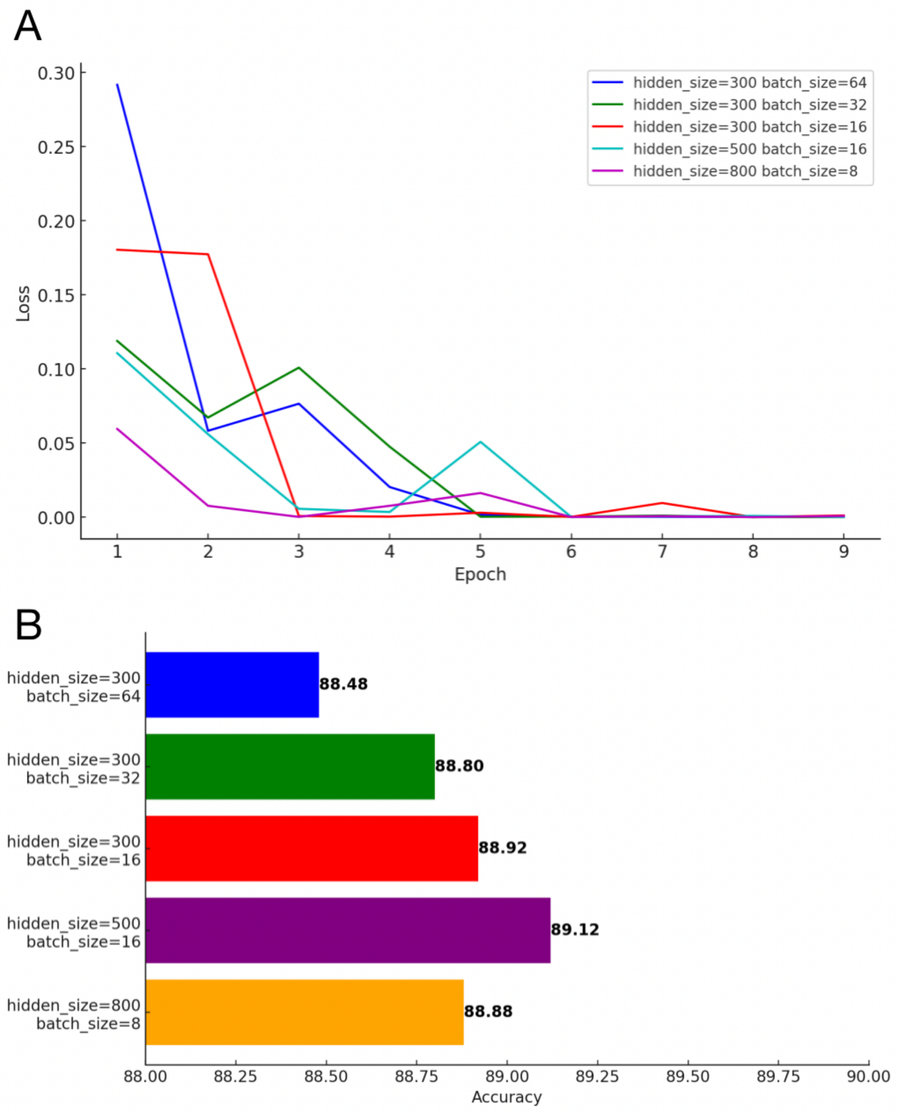


Figure 2: The loss and accuracy of different hyperparameters for LSTM model on validation data

Figure 2 displays the performance of the **LSTM** model in terms of training loss and accuracy under various conditions. Apart from the settings of hyperparameters, other parameters are as follows:

```
num_layers=2
dropout=0.2
batch_first=True
bias=True
```

With the model utilizing the **CUDA** device and trained for **ten** epochs. It is evident from the figure that the LSTM model achieves a higher level of accuracy of **0.8912** under the condition where the hidden size is 500, and the batch size is 16.

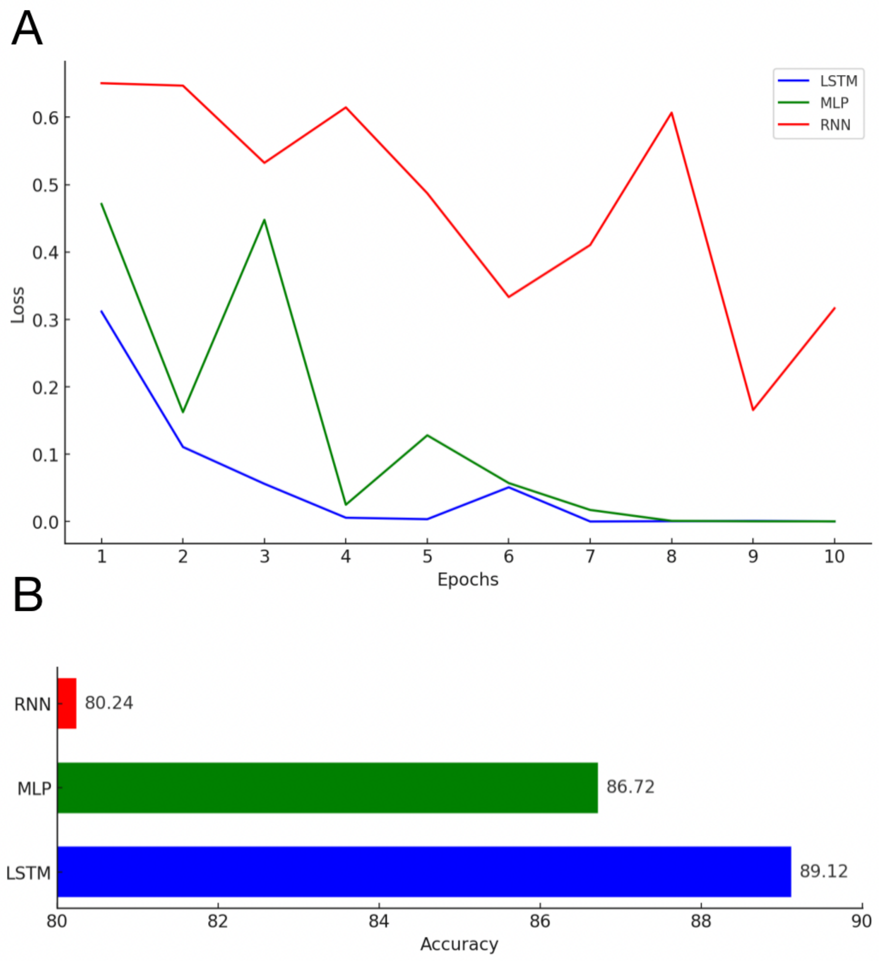


Figure 3: The loss and accuracy of different models on validation data

Figure 3 illustrates three models' LOSS and accuracy performance under identical parameter conditions. As anticipated, the **LSTM** model achieved optimal accuracy following LOSS convergence, surpassing the accuracy of the similarly converged **MLP** model. This accuracy is also significantly higher than that of the **RNN** model, which, despite nearing convergence, demonstrates a slower convergence rate.

Question 6

For this question, we set up a model following the structure provided in the task. The complete code for this can be found in the appendix. The model is structured with an embedding layer that processes integer-encoded tokens, a single-layer LSTM which is designed to capture the temporal dependencies inherent in the data, and a concluding linear layer. This final layer is responsible for transforming the hidden state into a vector that represents the probability distribution of possible next characters. First the model was applied on the NDFA dataset. The results from this are:

- epoch 1 average loss: 0.3040017635583776
- epoch 2: average loss: 0.20950191598957715
- epoch 3: average loss: 0.20873977178739506

The model was then applied to the brackets dataset and the results are:

- epoch 1: average loss: 0.5619228385490566
- epoch 2: average loss: 0.5266067649209204
- epoch 3: average loss: 0.526650496755342

These results suggest that the model is effectively learning from the NDFA and brackets dataset. The decreasing average loss values across epochs for both datasets indicate that the model is improving its predictions over time. The drop in average loss from the first to the second epoch in the NDFA dataset suggests a rapid adaptation to this dataset. On the other hand, the more gradual decrease in the brackets dataset implies a steady learning progression in a more challenging context. After this, sampling was done from the brackets dataset to inspect the model. Using the seed sequence from the assignment gave the following output: .start(()).end. This shows that the model has learned to generate a correct structure of the sequence which in this case is a balanced parenthesis sequence. The ".start" and ".end" tokens show the beginning and end of the sequence which are likely part of the training data's formatting to help the model recognize sequence boundaries. From the results and output of the model it can be said that the model is successfully able to learn from the dataset and generate the next sequence.

Question 7

Using the code template provided in the assignment, the model was applied on the the bracket dataset. The results of running the model with a **temperature=1.0** can be found in the appendix. From those results it can be seen that all the answers are incorrect and therefore a lower temperature was chosen. A temperature of **0.5** was selected and the code was run for **3** epochs of **10** samples each. The results of running the model with a temperature=1.0 can be found in the appendix. From the r

The result of this are given below:

- Epoch 1, sample: `.start(())`
- Epoch 1, sample: `.start(()(((())`
- Epoch 1, sample: `.start(())`
- Epoch 1, sample: `.start(())`
- Epoch 1, sample: `.start(()(()))`
- Epoch 1, sample: `.start(())`
- Epoch 1, sample: `.start(()())`
- Epoch 1, sample: `.start(())`
- Epoch 1, sample: `.start(())`
- Epoch 2, sample: `.start(())`
- Epoch 2, sample: `.start(())`
- Epoch 2, sample: `.start(())`
- Epoch 2, sample: `.start(()((()()`
- Epoch 2, sample: `.start(())`
- Epoch 2, sample: `.start(()())`
- Epoch 2, sample: `.start(())`
- Epoch 2, sample: `.start(())`
- Epoch 2, sample: `.start(())`
- Epoch 3, sample: `.start(()())`
- Epoch 3, sample: `.start(())`

- Epoch 3, sample: `.start(())`
- Epoch 3, sample: `.start(())`
- Epoch 3, sample: `.start(())`
- Epoch 3, sample: `.start(())`
- Epoch 3, sample: `.start(()()(((()`
- Epoch 3, sample: `.start(())`
- Epoch 3, sample: `.start(()(()()))`
- Epoch 3, sample: `.start(())`

Assessing the performance of the LSTM model in predicting the next token in a sequence, several things are observed First, the model seems to have a basic understanding of how sequences start, as it consistently adds the .start token at the beginning of each sequence. The result of predicting the next token is mixed across the epochs. It balances some sequences like .start(()()) well, but struggles with others, like .start(()(((()). The model tends to produce repetitive and simple patterns, such as .start(()), or incomplete ones with abrupt endings. This suggests it can generate basic structures but has difficulty with more complex or complete patterns. When comparing early and later training epochs, there's little progress in complexity or accuracy, implying the model isn't learning effectively over time or needs more data or training. There's also a noticeable lack of diversity in its outputs as there are many similar patterns repeating across epochs. This could mean the model is overfitting to simple patterns and needs to learn to generate more varied sequences.

Question 8

The autoregressive model was trained over **50** epochs. **torch.nn.utils.clip_grad_norm_** was used to ensure the gradients remained within a manageable range. Monitoring the training process was done using TensorBoard. The loss per token to control for variable sequence lengths and batch sizes using reduction='**sum**' within the cross-entropy loss function and normalizing by the total number of non-padding tokens. The graph of the loss curves can be seen below.

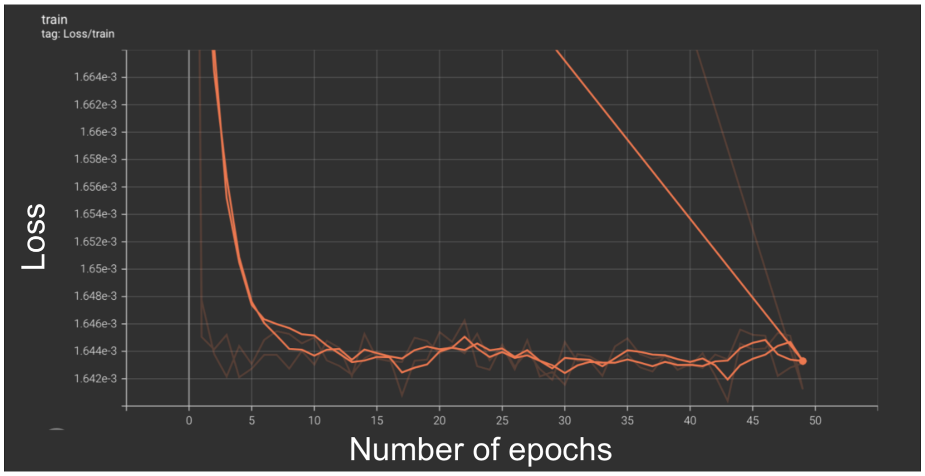


Figure 4: Graph showing average loss results over 50 epochs during the training of the model

When we look at the TensorBoard graph for our LSTM model's 50-epoch loss curve. There a clear downward trend at the beginning especially in the first 10 epochs. This shows that the model is learning as we expected. This also suggests rapid learning and optimization of the models parameters. The loss appears to plateau around epoch 10 following which there are instances of significant spikes in loss. These spikes could indicate periods where the model encountered sequences that diverged from the patterns it had learned, or they could be a result of overfitting to certain aspects of the training data. Toward the later epochs particularly after epoch 40 there is a dramatic increase in loss. This could because of an unstable learning rate or the model encountering outlier sequences that are not well represented in the training set.

A Appendix

Here are some relevant codes and additional results for different questions. Some of the code was pasted with line breaks done manually to account for page margins.

Appendix question 3 Full code:

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.nn.utils.rnn import pad_sequence
from data_rnn import load_imdb
```

```
import torch.nn.functional as F


# medel
class CustomModel(nn.Module):
    def __init__(self, num_tokens, embedding_size=300, hidden_size=300, num_classes=2):
        super(CustomModel, self).__init__()

        # 1.Embedding Layer
        self.embedding = nn.Embedding(num_tokens, embedding_size)

        # 2.Linear Layer
        self.linear = nn.Linear(embedding_size, hidden_size)

        # 3.ReLU Layer
        self.relu = nn.ReLU()

        # 5.Output Layer
        self.output = nn.Linear(hidden_size, num_classes)

        # Initialize the optimizer and loss function
        self.optimizer = optim.Adam(self.parameters(), lr=0.001)
        self.criterion = nn.CrossEntropyLoss()

    def forward(self, x):
        # 1.Embedding Layer
        embedded = self.embedding(x)

        # 2.Linear Layer
        linear_out = self.linear(embedded)

        # 3.ReLU Layer
        relu_out = self.relu(linear_out)

        # 4.Global max pooling
        pooled = torch.max(relu_out, dim=1).values

        # 5.Output Layer
        output = self.output(pooled)

        return output

    def train_model(self, train_loader, criterion, optimizer, num_epochs=1):
        model.train()
        for epoch in range(num_epochs):
            for batch_inputs, batch_labels in train_loader:
                optimizer.zero_grad()
                outputs = model(batch_inputs)
                loss = criterion(outputs, batch_labels)
                loss.backward()
                optimizer.step()

            print(f'Epoch {epoch + 1}/{num_epochs}, Loss: {loss.item()}')

    def evaluate(self, val_loader):
        self.eval()
        correct = 0
        total = 0

        with torch.no_grad():
            for val_inputs, val_labels in val_loader:
                val_outputs = self(val_inputs)
                _, predicted = torch.max(val_outputs.data, 1)
                total += val_labels.size(0)
                correct += (predicted == val_labels).sum().item()

        accuracy = correct / total
        print(f'Validation Accuracy: {accuracy * 100:.2f}%')


# load data
(x_train, y_train), (x_val, y_val), (i2w, w2i), _ = load_imdb(final=False, val=5000,
seed=0, voc=None, char=False)

# ped and convert
PAD, START, END, UNK = '.pad', '.start', '.end', '.unk'


def pad_and_convert(sequences, w2i, add_start_end_tokens=False):
    pad_token_idx = w2i[PAD]

    if add_start_end_tokens:
        start_token_idx = w2i[START]
        end_token_idx = w2i[END]
    else:
        start_token_idx = None
        end_token_idx = None

    # Find the maximum sequence length
    max_length = max(len(seq) for seq in sequences)

    # Pad sequences to the maximum length
    padded_sequences = [
        seq[:max_length] + [pad_token_idx] * max(0, max_length - len(seq))
        for seq in sequences
    ]

    # Convert to PyTorch tensor
    padded_sequences_tensor = torch.tensor(padded_sequences, dtype=torch.long)

    if add_start_end_tokens:
        # Add start token to the beginning of each sequence
        start_tokens = torch.full((len(padded_sequences), 1), start_token_idx,
        dtype=torch.long)
        padded_sequences_tensor = torch.cat([start_tokens, padded_sequences_tensor], dim=1)

        # Add end token to the end of each sequence
        end_tokens = torch.full((len(padded_sequences), 1), end_token_idx,
        dtype=torch.long)
        padded_sequences_tensor = torch.cat([padded_sequences_tensor, end_tokens], dim=1)

    return padded_sequences_tensor


# Pad and convert training sequences
padded_x_train = pad_and_convert(x_train, w2i, add_start_end_tokens=True)

# Pad and convert validation sequences
padded_x_val = pad_and_convert(x_val, w2i, add_start_end_tokens=True)


# model
model = CustomModel(num_tokens=len(i2w), embedding_size=300, hidden_size=300,
num_classes=2)

optimizer = model.optimizer
criterion = model.criterion
print(len(y_train))
print(padded_x_train.size())
print(torch.tensor(y_train, dtype=torch.long).size())


# Create DataLoader
train_dataset = torch.utils.data.TensorDataset(padded_x_train, torch.tensor(y_train,
dtype=torch.long))
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64, shuffle=True)

val_dataset = torch.utils.data.TensorDataset(padded_x_val, torch.tensor(y_val,
dtype=torch.long))
val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=64, shuffle=False)

# train
model.train_model(train_loader, criterion, optimizer, num_epochs=1)


# val
model.evaluate(val_loader)
```

Appendix question 5 Relevant code for training and validating models using GPU:

```
...
# check GPU availability
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
...

    def train_model(self, train_loader, criterion, optimizer, num_epochs=1):
        model.train()
        for epoch in range(num_epochs):
            for batch_inputs, batch_labels in train_loader:
                optimizer.zero_grad()
                # move data to GPU
                batch_inputs, batch_labels = batch_inputs.to(device),
                batch_labels.to(device)
                outputs = model(batch_inputs)
                loss = criterion(outputs, batch_labels)
                loss.backward()
                optimizer.step()

            print(f'Epoch {epoch + 1}/{num_epochs}, Loss: {loss.item()}')

    def evaluate(self, val_loader):
        self.eval()
        correct = 0
        total = 0

        with torch.no_grad():
            for val_inputs, val_labels in val_loader:
                # move data to GPU
                val_inputs, val_labels = val_inputs.to(device), val_labels.to(device)
                val_outputs = self(val_inputs)
                _, predicted = torch.max(val_outputs.data, 1)
                total += val_labels.size(0)
                correct += (predicted == val_labels).sum().item()

        accuracy = correct / total
        print(f'Validation Accuracy: {accuracy * 100:.2f}%')
...
# model
model = CustomModel(num_tokens=len(i2w), embedding_size=300, hidden_size=300, num_classes=2
# move model to GPU
model = model.to(device)
```

Appendix question 6

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from data_rnn import load_brackets, load_ndfa

x_train, (i2w, w2i) = load_ndfa(n=150_000, seed=0)

class AutoregressiveLSTM(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, num_layers):
        super(AutoregressiveLSTM, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, num_layers=num_layers,
        batch_first=True)
        self.fc = nn.Linear(hidden_dim, vocab_size)

    def forward(self, x):
        embedded = self.embedding(x)
        lstm_out, _ = self.lstm(embedded)
        out = self.fc(lstm_out)
        return out

vocab_size = len(i2w)
embedding_dim = 32
hidden_dim = 16
num_layers = 1

model = AutoregressiveLSTM(vocab_size, embedding_dim, hidden_dim, num_layers)

criterion = nn.CrossEntropyLoss(ignore_index=w2i['.pad'])
optimizer = optim.Adam(model.parameters())
```

```
def prepare_data(sequences, w2i, batch_size=32):
    processed_sequences = [[w2i['.start']] + s + [w2i['.end']] for s in sequences]
    max_len = max(len(s) for s in processed_sequences)
    padded_sequences = [s + [w2i['.pad']] * (max_len - len(s)) for s in
    processed_sequences]
    input_sequences = torch.tensor(padded_sequences,
    dtype=torch.long)
    target_sequences = torch.tensor([s[1:] + [w2i['.pad']] for s in padded_sequences],
    dtype=torch.long)
    dataset = TensorDataset(input_sequences, target_sequences)
    return DataLoader(dataset, batch_size=batch_size, shuffle=True)

train_loader = prepare_data(x_train, w2i, batch_size=32)

for epoch in range(3):
    model.train()
    total_loss = 0
    for input_seqs, target_seqs in train_loader:
        optimizer.zero_grad()
        output = model(input_seqs)
        loss = criterion(output.transpose(1, 2), target_seqs)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    avg_loss = total_loss / len(train_loader)
    print(f'epoch {epoch+1}: average oss: {avg_loss}')
```

Appendix Question 7 The following result indicates the test result with a temperature of 1.0:

- `s:.start(().pad).pad.pad.pad))(.start.unk)`
- `s:.start(().start.pad)`
- `s:.start(()`
- `s:.start(().unk)`
- `s:.start(()`
- `s:.start(())).pad.pad).pad(`
- `s:.start(().unk()`
- `s:.start(().pad`
- `s:.start(()().unk).start`

- s:.start(().unk.start.start.unk.pad.start.unk(.pad.start.unk(.pad.pad.unk.start.unk
- s:.start(()).unk.unk
- s:.start(()((.unk)).pad)
- s:.start(()().start).pad(.unk).pad.unk.pad.unk.start.pad.start.pad(.unk.pad.unk)
- s:.start(().unk(().unk)
- s:.start(()().start
- s:.start(()(.unk
- s:.start(().unk).unk).pad.pad(.unk(.unk.unk.unk.start.start.pad.unk.pad.unk.unk.start.s
- s:.start(()
- s:.start(()).start
- s:.start(().start
- s:.start(().pad.pad(.start(
- s:.start(()
- s:.start(()
- s:.start(()(
- s:.start(().pad.start).unk
- s:.start(().unk.pad
- s:.start(().pad.start).unk(.unk.start
- s:.start(()
- s:.start(().pad
- s:.start(().start.unk(.pad).start(.pad.unk.start.start.start.pad.unk.unk.unk)

Appendix question 8

```
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter('runs/my_first_model')

for epoch in range(50):
    model.train()
    total_loss, total_tokens = 0, 0
    for input_seqs, target_seqs in train_loader:
        optimizer.zero_grad()
        output = model(input_seqs)
        output = output.transpose(1, 2)  # Adjusting the shape for the loss function

        loss = criterion(output, target_seqs)
        tokens = target_seqs.ne(w2i['.pad']).sum()  # Counting the number of actual words
        loss_per_token = loss / tokens

        loss_per_token.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1)
        optimizer.step()

        total_loss += loss.item()
        total_tokens += tokens.item()

    avg_loss = total_loss / total_tokens
    writer.add_scalar('Loss/train', avg_loss, epoch)
    print(f'epoch {epoch+1}: avg loss: {avg_loss}')
```

1. epoch 1: average loss: 0.0017845509120182114
2. epoch 2: average loss: 0.0016450618228415301
3. epoch 3: average loss: 0.0016441410848339329
4. epoch 4: average loss: 0.0016452082607128294
5. epoch 5: average loss: 0.0016421089885237756
6. epoch 6: average loss: 0.0016427350437660296
7. epoch 7: average loss: 0.0016437385357626186
8. epoch 8: average loss: 0.001643747541142293
9. epoch 9: average loss: 0.0016427384430281545
10. epoch 10: average loss: 0.001644030150277786
11. epoch 11: average loss: 0.0016430387688256983
12. epoch 12: average loss: 0.0016447746654778819
13. epoch 13: average loss: 0.0016442507542748076
14. epoch 14: average loss: 0.0016421917135820748
15. epoch 15: average loss: 0.001645275794489739
16. epoch 16: average loss: 0.0016434648055329612
17. epoch 17: average loss: 0.0016432887372671283

18. epoch 18: average loss: 0.001643225569059286
19. epoch 19: average loss: 0.0016449952680963069
20. epoch 20: average loss: 0.0016447494986815423
21. epoch 21: average loss: 0.0016438582711702477
22. epoch 22: average loss: 0.0016444210950371657
23. epoch 23: average loss: 0.0016438561254491343
24. epoch 24: average loss: 0.0016452828655992293
25. epoch 25: average loss: 0.00164334631692213
26. epoch 26: average loss: 0.0016443958317093115
27. epoch 27: average loss: 0.0016426887929813492
28. epoch 28: average loss: 0.0016448218548884756
29. epoch 29: average loss: 0.0016421661925062497
30. epoch 30: average loss: 0.0016424946923945202
31. epoch 31: average loss: 0.0016415661483862747
32. epoch 32: average loss: 0.0016437974468642156
33. epoch 33: average loss: 0.0016436235980000285
34. epoch 34: average loss: 0.0016431165295081158
35. epoch 35: average loss: 0.0016431430895498657
36. epoch 36: average loss: 0.001643766191342343
37. epoch 37: average loss: 0.0016428359084675814
38. epoch 38: average loss: 0.001642524667666312
39. epoch 39: average loss: 0.0016437092958481285
40. epoch 40: average loss: 0.0016426489638383993
41. epoch 41: average loss: 0.0016430540269739811
42. epoch 42: average loss: 0.00164278933932627
43. epoch 43: average loss: 0.0016437676026681964
44. epoch 44: average loss: 0.0016434177004746259
45. epoch 45: average loss: 0.0016455867298673986

46. epoch 46: average loss: 0.0016452100023266207
47. epoch 47: average loss: 0.0016451418870993284
48. epoch 48: average loss: 0.0016422093880828733
49. epoch 49: average loss: 0.0016428135580349615
50. epoch 50: average loss: 0.0016431392527866443