Deep Learning Lab - Assignment 3

name: Fabian Gobet, email: gobetf@usi.ch

9 Dec 2023

1 Language models with LSTM [90/100]

1.1 Data (20 points)

1.

After having download the .json file containing the data, I used the function pd.read_json() to load it and proceeded to using the .info, .shape and .describe methods on the variable holding the data to explore the contents of it. Notice that the 'lines' flag is set to True because data comes in json format per line.

The data is initially composed of 209527 entries with 6 attribute columns, namely link, headline, category, short-description, authors and date.

```
df = pd.read_json('./News_Category_Dataset_v3.json', lines=True)
print(df.info(),"\n")
print(df.shape,"\n")
print(df.describe,"\n")
```

We are interested in filtering out all entries that do not have 'POLITICS' as category. To achieve this, I use the .loc() method to select all entries that have 'POLITICS' in the category column. Furthermore, after filtering the data I again use .info() and .shape to see properties of the data and print the first 3 entries.

```
df2 = df.loc[df['category'] == 'POLITICS']
print(df2.info(),"\n")
print(df.shape,"\n")
print(df2[0:3],"\n")
```

2.

To tokenize the words I didn't go with a standard space split because this would mean that sequences such has 'word', 'word,', 'another' and 'another,' would be considered 4 different tokens. Furthermore, the aggregation of punctuation to words in a token would limit the model's expressivity and in general increase the vocabulary size.

As such, in order to consider punctuation and white spaces I used the 'TreeBankWordTokenizer' from NLTK library, which provides the exact functionality that I am looking for.

After instantiating the tokenizer, I generate a list of lists, where each element is a list of the ordered tokens that compose a sentence with the

added '$< EOS >$' token, for all sentences in the headlines of our filtered dataset.

```
t = TreebankWordTokenizer()
tkns = [t.tokenize(hdline.lower())+["<EOS>"] for hdline in df2['headline']]
```

Opposed to the suggested expectation, the tokenizing process was rather quick. Since it is a stepstone of this exercise to save the tokenization of the headline sentences into a pickle format file, I proceeded defining a function that takes in as argument an object and a path to save the object to. Likewise, i also defined a function to load a file given the path.

```
def save_to_pickle(object_var, path):
  with open(path, 'wb') as handle:
    pickle.dump(object_var, handle, protocol=pickle.HIGHEST_PROTOCOL)
def load_from_pickle(path):
  with open(path, 'rb') as handle:
    return pickle.load(handle)
:::
save_to_pickle(tkns, './tokenized.pickle')
#tkns = load_from_pickle('./tokenized.pickle')
```

3.

To assemble a list of all words, without repetition, and to count the frequency of each word, we can run a nested for loop in the list of of sentences. The nested loop will iterate through the sentences and through the tokens of each sentence. In each word, we will save its frequency to a dictionary whose key will be the word itself. Furthermore, if the word doesn't yet exist in the dictionary, we initialize it and also add it to the 'all_words' list.

```
all_words = []
frequency = {}
for t in tkns:
  for w in t:
    if w in frequency:
      frequency[w] = frequency[w]+1
    else:
      frequency[w] = 1
      all_words.append(w)
```

We are now in position of finalizing the assembling of the 'all_words' list and printing out the top 5 most frequent words. To do this, I i sort the dictionary by its values (opposed to keys) and print the top 5. Furthermore, to finalize the 'all_words' list i remove the element '¡EOS¿' and then concatenate it at the beginning and also 'PAD' at the end.

```
for i,f in enumerate(sorted(frequency, key=frequency.get, reverse=True)[:5]):
  print(f"{i+1}: {f} -> {frequency[f]} repetitions")
all_words.remove("<EOS>")
all_words = ["<EOS>"] + all_words + ["PAD"]
```

We are now in conditions of building the 'word_to_int' and 'int_to_word' dictionaries. To achieve this I'll use the current order of the list to map the word to its index, and then i use the same mapping to define its reverse.

```
int_to_word = {}
for i in range(len(all_words)):
  int_to_word.update({i:all_words[i]})
word_to_int ={}
for k,v in int_to_word.items():
  word_to_int.update({v:k})
```

And to finalize, like in the prior exercise, I save both dictionaries and the list of all words to a pickle file

```
save_to_pickle(all_words, './all_words.pickle')
save_to_pickle(int_to_word, './int_to_word.pickle')
save_to_pickle(word_to_int, './word_to_int.pickle')
#all_words =  load_from_pickle('./all_words.pickle')
#int_to_word = load_from_pickle('./int_to_word.pickle')
#word_to_int = load_from_pickle('./word_to_int.pickle')
```

4.

For the Dataset class I used inheritance from the from the torch.utils.data.Dataset, like in class, with inputs the list of tokenized sequences and the word_to_int dictionary.

The init subfunction of this class converts each token to its index in the vocabulary and saves all of this data into a 'data' variable. The 'len' subfunction returns the size of the dataset and the 'get_item', given an index, returns a tuple of tensors, where the first one comprises all tokens of that sentence except for the last one, and the second tuple all tokens of that sentence except for the first one.

```
class Dataset(torch.utils.data.Dataset):
    def __init__(self, tokenized_sequences: list, word_to_int_dict: dict):
      self.data = []
      for sequence in tokenized_sequences:
        tokenized = [word_to_int_dict.get(word, word_to_int_dict['PAD'])
for word in sequence]
        self.data.append(tokenized)

    def __len__(self):
      return len(self.data)

    def __getitem__(self, ix):
      d = self.data[ix]
      return torch.tensor(d[:-1]), torch.tensor(d[1:])
```

5.

The collate function I defined will be receiving two arguments, the batch and the padding index. The reason for this is because the function will be defined outside the scope of the dictionary. Because of this, when employing the use of the collate function we can define a lambda function that partially instantiates this collate, which is then used for the Dataloader with an input argument for the batch.

The collate function first starts by unrolling the arguments of 'batch' inside a zip and extracting the data and targets. for each of these it's then applied the nn.utils.rnn.pad_sequence function with a flag True on batch_first and the the padding index. Both padded target and data are then returned.

```
def collate_fn(batch,pad_index):
  data, targets = zip(*batch)
  padded_data = nn.utils.rnn.pad_sequence(data,batch_first=True,padding_value=pad_idx)
  padded_targets = nn.utils.rnn.pad_sequence(targets,batch_first=True,padding_value=pad_idx)
  return padded_data, padded_targets
```

To create the dataloader I just have to define a batch size and call the functions with the lambda partially defined collate function, as mentioned before

```
batch_size = 64
collate = lambda batch: collate_fn(batch, pad_index=word_to_int['PAD'])
dataloader = DataLoader(dataset, batch_size=batch_size, collate_fn=collate, shuffle=True)
```

1.2 Model definition (20 points)

The model in this exercised will be LSTM based, with inputs $embed\_size$, $hidden\_size$, num_layers. It will have an initial embedding layer which will map the inputs from [0,$vocab\_size$) into some 'embed_size' vector space, an LSTM with input size $embed\_size$, hidden layer size $hidden\_size$, num_layers layers with batch as first input dimension, and dropout with default probability of 0 and a fully connected layers with input $hidden\_size$ and output $vocab\_size$.

The forward process follows the same order as the prior explanation, returning the output of the fully connected layer, aswell as the hidden and cell state from the LSTM.

The init_state() initializes both hidden state and cell state with all zeros and returns these two variables.

```
class myNet(nn.Module):
  def __init__(self, vocab_size, embed_size, hidden_size, num_layers,dropout=0):
      super(myNet, self).__init__()
      self.hidden_size = hidden_size
      self.num_layers = num_layers
      self.embed = nn.Embedding(vocab_size, embed_size)
      self.lstm = nn.LSTM(embed_size, hidden_size, num_layers, batch_first=True)
      self.linear = nn.Linear(hidden_size, vocab_size)
      self.drop = nn.Dropout(dropout)
  def forward(self, x, h):
      x = self.embed(x)
      out, (h, c) = self.lstm(x, h)
      out = self.drop(out)
      out = self.linear(out)
      return out, (h, c)
  def init_state(self, b_size=1):
      h = c = torch.zeros(self.num_layers, b_size, self.hidden_size)
      return h, c
```

1.3 Evaluation - part 1 (10 points)

- **random_sample_next()**

  Given and model, an input and previous state x and $prev\_state$, and possibly 'topk' positive integer and uniform boolean, this function computes the topk most probable words and returns a randomly chosen one according to either a uniform distribution or the distribution with probability space equal to the topk words.

The input and previous state are ran through the model, generating an output and a new state (hidden and cell). From the output we then choose the last word from the first (and only!) batch all of the probabilities for that word and save it into the $last\_word$ variable. If topk is not defined then we define it as being the same size of the vocabulary, and proceed to use the torch.topk function which selects the topk elements from an array, returning both their values and indexes.

If uniform isn't set to true, then it means we want to compute a probability distribution among the chosen topk logits, setting this distribution to variable p. Having done all the prior steps, we can now use numpys random.choise method to retrieve an element from our indexes given out distribution.

Finally, we return the chosen index and the state.

```
def random_sample_next(model, x, prev_state, topk=5, uniform=True):
  out, state = model(x, prev_state)
  last_out = out[0, -1, :]
  topk = topk if topk else last_out.shape[0]
  top_logit, top_ix = torch.topk(last_out, k=topk, dim=-1)
  p = None if uniform else F.softmax(top_logit.cpu().detach(), dim=-1).numpy()
  sampled_ix = np.random.choice(top_ix.cpu(), p=p)
  return sampled_ix, state
```

- **sample_argmax()**

  In theory we could reuse random_sample_next() and set topk to 1, this would return the highest probable word. I chose not to do so because i was afraid that this method of exploitation would be seen as not covering this exercise.

  Both sample_argmax() and random_sample_next() will be called from the same scope with the same arguments. Therefore, I set topk and uniform as input on this function too, even though they are never used. This allows later to call a use functions as input arguments without having to worry about the input arguments.

  The sample_argmax() function follows the same idea as the random_sample_next(), but instead of considering the topk most probable words, we're solely interested in the most probable word. Like before, we ran the state and input throught the model, filter out the vocabulary logits for the last word and choose the most probable, returning its index and the state.

```
def sample_argmax(model, x, prev_state,topk=None, uniform=None):
  out, state = model(x, prev_state)
  last_out = out[0, -1, :]
  top_logit, top_ix = torch.topk(last_out, k=1, dim=-1)
  return top_ix, state
```

- **sample()**

  The exercise description for this function stated that there should be some minimal input arguments. This statement leaves room for some ambiguity of interpretation: one can see it as 'having at least those arguments" or as "having only those arguments". As such, my interpretation for it was that it should have 'at least those arguments'.

  The sample() function takes in as arguments the model, the seed words which represent the words that will start the sentence to be generated, a sampling function, a device, a topk argument, a uniform distribution

boolean, the max length allowed for sentence generation and a stop integer flag.

First, we check if seed words is not already an instance of a list or tensor, and if its not we convert it into a tensor. We then proceed to put the model in evaluation mode and set torch to not accumulate gradients.

We extract the vocabulary values for the seed words into a list and add an extra dimension at position 0 to our seeds words. We have to add this dimension because the model is set to use batch as first input dimension.

After this, we initialize the state of the model put it on the device and proceed to compute the words. In the following iteration we account for the words that were already in the seed so we don't surpass the sequence length. On each iteration we use the sampling function to generate the next word, save its index and set it to be the input for the next iteration along with the retrieved states. If in any of these iterations the index returns is equal to the stop argument (which should be used for the ¡EOS¿ tag index on the vocabulary), the iterations stop.

At the end, the model is set to training mnode again and the list with all generated indexes is returned.

```
def sample(model, seed_words, sample_function, device, topk=5,
    uniform=True, max_sntc_length=18, stop_on=None):
  seed_words = seed_words if (isinstance(seed_words, (list, tuple)
      ) or torch.is_tensor(seed_words)) else torch.tensor(
      seed_words)
  model.eval()
  with torch.no_grad():
    sampled_ix_list = seed_words.tolist()
    x = seed_words.unsqueeze(0).to(device)

    h,c = model.init_state(b_size=1)
    h, c = h.to(device), c.to(device)
    for t in range(max_sntc_length - len(seed_words)):
      sampled_ix, (h,c) = sample_function(model, x, (h,c), topk,
          uniform)
      h, c = h.to(device), c.to(device)
      sampled_ix_list.append(int(sampled_ix))
      x = torch.tensor([[sampled_ix]]).to(device)

      if sampled_ix==stop_on:
        break
  model.train()
  return sampled_ix_list
```

1.4 Training (35 points)

Both train_with_BBTT() and train_with_TBBTT() funtions are analogous, that is a lot of the code is repeated. Hence, all of the common explanation will be addressed in the BBTT (also referred and standard version) implementation.

Because the block of code is large, it will be segmented into three parts to make it more digestible.

Furthermore, some extra auxiliary functions were implemented, which will be addressed now, before moving into the standard and TBBTT implementations.

The function get_rand_partial_prompt() takes as input a tokenized sentence in list form, generates a random integer between 0 and then number of words

in the sentence divided by 2 (non included boundary), and returns all ordered words from that sentence up to the random index.

```
def get_rand_partial_prompt(sentence):
  idx = np.random.choice(len(sentence)//2,1)[0]
  if(idx==0):
    idx=1
  return sentence[:idx]
```

Next, we have a the evaluate() function, which is used to evaluate the sentences produced by the model in a current state. This function takes as input the model, a dataloader (for convenience the dataloader is used because of the randomness setting), the int to word dictionary, the device, the topk integer, the uniform boolean and the maximum permited length of the sentence. It returns a list containing the original selected sentence, the truncated sentence with get_rand_partial_prompt(), the argmax sampled sentence and the sample next sentence.

inside this function is a sub-function defined which gets as input the tokenized sentences and outputs the sentence in the vocabulary defined language.

The function first extracts a randomized batch from the dataloader and extracts the first element from it. It then produces a randomly truncated sentence and computes the models prediction for both argmax sample and sample next functions, returning all results.

The function also prints out all selected and computed values to the console.

The last input argument input_sentence is used to pass a sentence, rather then extracting it from the dataloader. This will be used in the last exercise

```
def evaluate(model,dataloader,int_to_word, device, topk=5, uniform=None
    , max_sntc_length=30, input_sentence=None):
  def translate(indxs_list):
    avoid = [0,len(int_to_word)-1]
    res = []
    for i in indxs_list:
      if(i not in avoid):
        res.append(int_to_word[i])
    return res
  if input_sentence:
    x = input_sentence
    seed_words = input_sentence
  else:
    x,_ = next(iter(dataloader))
    x=x[0]
    seed_words = get_rand_partial_prompt(x)
  argmax_indxs = sample(model, seed_words, sample_argmax, device, topk=
      topk, uniform=uniform, max_sntc_length=max_sntc_length, stop_on
      =0)
  topk_indxs = sample(model, seed_words, random_sample_next, device,
      topk=topk, uniform=uniform, max_sntc_length=max_sntc_length,
      stop_on=0)
  o_sntc = " ".join(translate(x.tolist()))
  seed_sntc = " ".join(translate(seed_words.tolist()))
  arg_sntc = " ".join(translate(argmax_indxs))
  topk_sntc = " ".join(translate(topk_indxs))
  print("Original sentence -> "+o_sntc)
  print("Seed -> "+seed_sntc)
  print("Argmax prompt -> "+arg_sntc)
  print("Topk prompt -> "+topk_sntc+"\n")
  return [o_sntc, seed_sntc, arg_sntc, topk_sntc]
```

In order to plot the training losses and perplexities I've used matplotlib and

generated plots with the losses and perplexities computed in the training. This exercise was developed on Google Colab, a notebook like platform that allows to execute single blocks of code and store variable values in memory. As such, the difference between perplexity and loss plots for the standard and truncated implementations was only a line of code. Therefore, for each of the cases one line will be commented out, which can be uncommented to fufill the plot of the other implementation.

```
fig, ax = plt.subplots(figsize=(10,5))
fig.suptitle("Loss values during training")
#plt.plot(np.arange(1,len(losses_standard)+1,1), losses_standard,
    marker='o', linestyle='dashed', color='green', label='Standard-BBTT
     loss')
plt.plot(np.arange(1,len(losses)+1,1), losses, marker='o', linestyle='
    dashed', color='blue', label='TBBTT loss')
plt.plot([1,len(losses)], [1.5,1.5], marker='o', linestyle='dashed',
    color='orange', label='1.5 threshold')
plt.xlabel("epochs")
plt.ylabel("losses")

yticks = [losses[0]]
min_interval = (losses[0]-losses[-1])/10
for tick in losses:
  if yticks[-1]-tick > min_interval:
    yticks.append(tick)
plt.yticks([1.5]+yticks)
plt.xticks(np.arange(1,len(losses)+1,1))
plt.grid()
plt.legend()
plt.savefig(PATH+'losses_TBBTT_plot')
plt.show()

fig, ax = plt.subplots(figsize=(10,5))
fig.suptitle("Perplexity values during training")
#plt.plot(np.arange(1,len(perplexities_standard)+1,1),
    perplexities_standard, marker='o', linestyle='dashed', color='
    orange', label='Standard-BBTT perpl.')
plt.plot(np.arange(1,len(perplexities)+1,1), perplexities, marker='o',
    linestyle='dashed', color='blue', label='TBBTT perpl.')
plt.xlabel("epochs")
plt.ylabel("perplexities")

yticks = [perplexities[0]]
min_interval = (perplexities[0]-perplexities[-1])/10
for tick in losses:
  if yticks[-1]-tick > min_interval:
    yticks.append(tick)
plt.yticks([1.5]+yticks+perplexities[1:6], fontsize=8)
plt.xticks(np.arange(1,len(losses)+1,1))
plt.grid()
plt.legend()
plt.savefig(PATH+'perplexities_TBBTT_plot')
plt.show()
```

All the plots are shown and also saved in the local file system. The use of this kind of plots was thoroughly used in all projects so far. The code does not use any special arrangements and is pretty straight forward.

1. **train_with_BBTT()**

   The train_with_BBTT() first starts by initializing the losses, perplexities, sentences and epoch variables. The losses and perplexities are to save the respective values at each epoch, the epoch variable is just a counter and the sentences variable saves the return of the evaluate() function at each epoch.

Once all variable are initialized, we start by iterating through the epochs. On each epoch we increment the variable for the number of epochs, initialize a running loss accumulator and the number of batches to 0 and put the model in training mode.

```
def train_with_BBTT(max_epochs, model, dataloader, criterion,
    optimizer, device, int_to_word, clip=None):
  losses = []
  perplexities = []
  sentences = []
  epoch = 0
  while epoch < max_epochs:
    epoch += 1
    running_loss = 0.0
    model.train()
    num_batches=0;
```

The next step is then to iterate through all the batches on the dataloader. In each iteration the number of batches variable is incremented and the optimizer is set to zero gradient.

We convert both the input and target into tensors of dtype torch.int64 and put them on the device. Next, we initialize the states using the model init_state() function and also move them onto the device.

The outputs and state are then computed by running the inputs and previous state through the model, and the outputs second and third dimensions are transposed so the Softmax+CrossEntropy criterion can for each word of a sequence compare the probabilities of for all the vocabulary for that word against the target index which corresponds to the correct word in the vocabulary.

Once the transposition in done, we can now compute the loss with the criterion, aswell as the running loss and backward propagate the gradients.

If the clip argument is set, then the norm of the gradients of the models parameters is clipped using the clip set value. In either case, after this step the step() function of the optimizer is executed to update the models parameters.

```
    for input, target in dataloader:
      num_batches = num_batches+1
      optimizer.zero_grad()
      input = input.to(device).to(torch.int64)
      target = target.to(device).to(torch.int64)
      h, c = model.init_state(b_size=input.shape[0])
      h = h.to(device)
      c = c.to(device)
      outputs, (h,c) = model(input, (h,c))
      outputs = outputs.transpose(1,2) # N VocabSize seq_len
      loss = criterion(outputs, target)
      running_loss = running_loss+loss.item()
      loss.backward()
      if clip:
        torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
      optimizer.step()
```

After all batches are processed, before turning intp the next epoch we still have some things to attend to.

Firs, we calculate the loss by taking the sum of all losses in the running loss variable, divide it by the number of batches and save it into the

losses list. Same goes for the perplexity, which results from exponentiation of calculated loss. It is noteworthy that perplexities are calculated through the use of a softmax + crossentropy criterion. Because this model uses this criterion as a loss function, we can easily compute it using our prior calculated results for the loss. Perplexities provide information on how good a probability model predicts a sample. It could also be seen, in this exercise, as the exponential scaling of the losses.

After having done this, the running loss is set to zero and we append to the sentences list the evaluation of the model using the evaluate() function previously described.

At the very end, the model, losses, perplexities and sentences are returned.

```
        loss_val = running_loss/num_batches
        perpl = np.exp(loss_val)
        running_loss = 0.0
        losses.append(loss_val)
        perplexities.append(perpl)
        print('Epoch [{}/{}], Loss: {:.4f}, Perplexity: {:5.2f}'.
            format(epoch, max_epochs, loss_val, perpl))
        sentences.append(evaluate(model,dataloader,int_to_word,device)
            )
    return model, losses, perplexities, sentences
```

We are now in conditions of initializing the model and all the hyperparameters, the criterion and start training.

```
vocab_size = len(all_words)
device = torch.device("cuda:0" if torch.cuda.is_available() else "
    cpu")

# hyperparameters
embed_size = 64
hidden_size = 1024
num_layers = 2
max_epochs = 15
chunk_size = 10
learning_rate = 0.002

criterion = nn.CrossEntropyLoss()
model_standard = myNet(vocab_size, embed_size, hidden_size,
    num_layers,dropout=0.3).to(device)
optimizer_standard = torch.optim.Adam(model_standard.parameters(),
     lr=learning_rate)

model_standard, losses_standard, perplexities_standard,
    sentences_standard = train_with_BBTT(max_epochs,
    model_standard, dataloader, criterion, optimizer_standard,
    device, int_to_word, clip=1)
```

Once the model has finished training for 15 epochs, a loss of 0.7 and a perplexity of 2.02 are achieved.

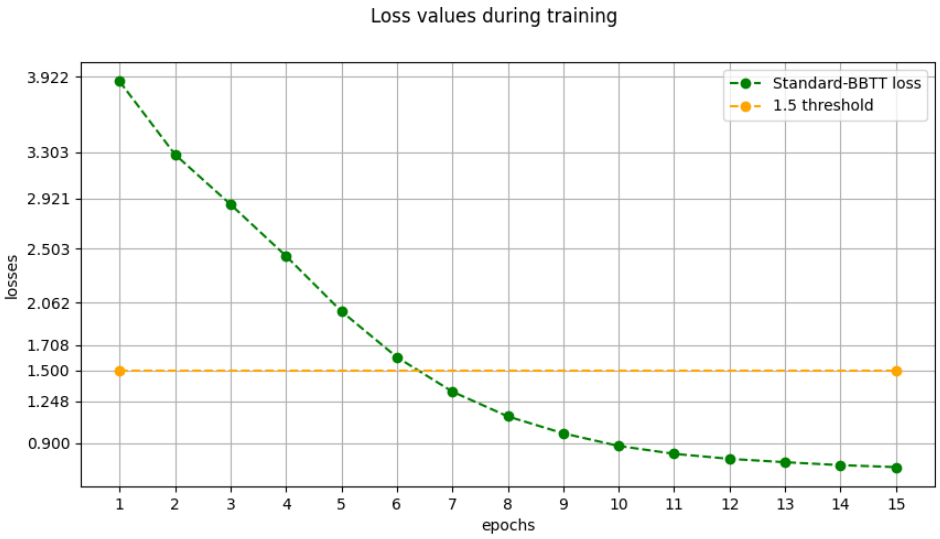

Figure 1: Standard-BBTT losses

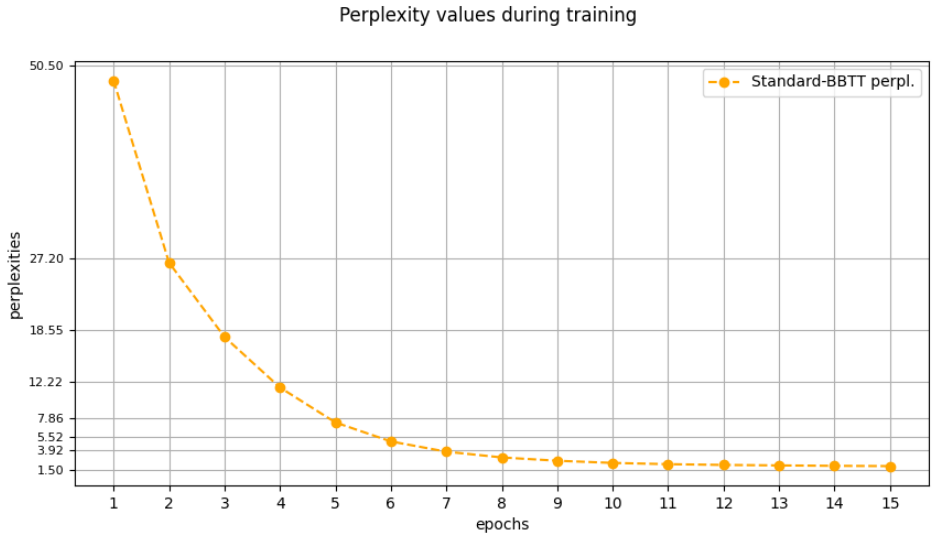

Figure 2: Standard-BBTT perplexities

We can see from the plots that losses go down in a linear fashion and then slowly converge to reach a asymptotic limit. The same can be said about the perplexities.

As for the evaluation outputs during training, it'll be shown the outputs from epoch 1, 8 and 14.

```
Epoch [1/15], Loss: 3.8854, Perplexity: 48.68
Original sentence -> if anyone is unifying the gop , it 's clinton
    , not trump
Seed -> if anyone is unifying
Argmax prompt -> if anyone is unifying the gop 's new hampshire
Topk prompt -> if anyone is unifying a major , trump and the
    country
--------------------------------------------------------------
Epoch [8/15], Loss: 1.1214, Perplexity: 3.07
Original sentence -> trevor noah calls fox news the 'phony victims
    unit '
Seed -> trevor noah calls
Argmax prompt -> trevor noah calls fox news a powerful threat
Topk prompt -> trevor noah calls on lawmakers to investigate
    mental health care because they got a chance '
--------------------------------------------------------------
Epoch [14/15], Loss: 0.7193, Perplexity: 2.05
Original sentence -> there is no trump pivot
Seed -> there
Argmax prompt -> there 's a major intensity gap on the gop 's new
    health bill
Topk prompt -> there is a major intensity on the house 's future
    is going to be the same
```

We can observe that after the first epoch it can already generate some sentences but they don't really hold proper semantic meaning. The structure of the language has been captured, but not the semantics.

On the 8th epoch we can see that both argmax and topk sampling already produce very good sentences that are both gramatically and semantically correct. Finally, in the 14th epoch we can still see that the generated sentences are well structured, yet the topk argument fails to produce a sentence that is semantically correct. This happens because the choice over the generating words is done sarcastically for the top 5 most probable words.

2. **train_with_TBBTT()**

The truncated back-propagation through time pretty much resembles the standard implementation. Only the case in which it differs will be addressed here, whereas the common partd is already adressed in the standard implementation.

The truncated version differs by further devide each batch into chunks along the sentences (and not along the training examples) and performing back prop on each chunck. This procedure can be very convenient if the sentences we are processing are long and memory is a resource to which some care must be accounted for. Of course, choosing to use truncation also comes with some cons. Namely, that some bias can be introduced (because the semantics of each sentence are truncated) and lower convergence rate.

The signature of the truncated version function takes in an extra argument, which is an hyperparameter, corresponding to the size of the sentence within a chunck.

```
def train_with_TBBTT(... optimizer, chunk_size, device, ...):
```

On each batch, we have to calculate the number of chuncks for that batch and then iterate through those chuncks, performing back prop in each of these. Furthermore, because context is very important, we only initialize the state for the model on the first chunck, whereas in the remaining we

always use the state computed by the last used chunck. Because we are performing back prop at each chunck, whenever we reuse the state from the rpevious step we also need to account for the gradient attached to that state. as such, for each chunck that is not the first one, we must detach the gradients at the very beggining and perform backprop on the state returned by the model.

Because the loss is now being calculated in each chunck, we must account that factor for the calculation of the running loss. Hence, each loss is divided by the number of chuncks and added to the running loss accumulator variable

```
...
for input, output in dataloader:
  n_chunks = input.shape[1] // chunk_size
  num_batches = num_batches+1
  for j in range(n_chunks):
      optimizer.zero_grad()
      if j < n_chunks - 1:
          input_chunk = input[:, j * chunk_size:(j + 1) *
              chunk_size].to(device).to(torch.int64)
          output_chunk = output[:, j * chunk_size:(j + 1) *
              chunk_size].to(device).to(torch.int64)
      else:
          input_chunk = input[:, j * chunk_size:].to(device).to(
              torch.int64)
          output_chunk = output[:, j * chunk_size:].to(device).to(
              torch.int64)
      if j == 0:
          h, c = model.init_state(b_size=input.shape[0])
      else:
          h, c = h.detach(), c.detach()
      h = h.to(device)
      c = c.to(device)
      outputs, (h,c) = model(input_chunk, (h,c))
      outputs = outputs.transpose(1,2) # N VocabSize seq_len
      loss = criterion(outputs, output_chunk)
      running_loss = running_loss+(loss.item()/n_chunks)
...
```

Everything else regarding this function is equal to the standard implementation.

We are now in position of training a new model with the truncated implementation. As such, like before, we define a mode, an optimizer and execute training. All other hyperparameters are to be the same.

```
model = myNet(vocab_size, embed_size, hidden_size, num_layers,
    dropout=0.3).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

model, losses, perplexities, sentences = train_with_TBBTT(
    max_epochs, model, dataloader, criterion, optimizer,
    chunk_size, device, int_to_word, clip=1)
```

Once the model has finished training for 15 epochs, a loss of 0.82 and a perplexity of 2.28 are achieved.

A small but yet noticeable difference can be seen regarding the loss and perplexity values when compared with the standard implementation. The models were trained using the same dataloader and further experiences were made using the same dataloader without shuffle. The results always showed that the truncated implementation tends to have a slower convergence that the standard implementation, where losses and perplexities are usually higher.

For the plots of the losses and perplexities of the truncated version we have the following images that are identical to the standard implementation, and which interpretation is already stated in the previous paragraph.

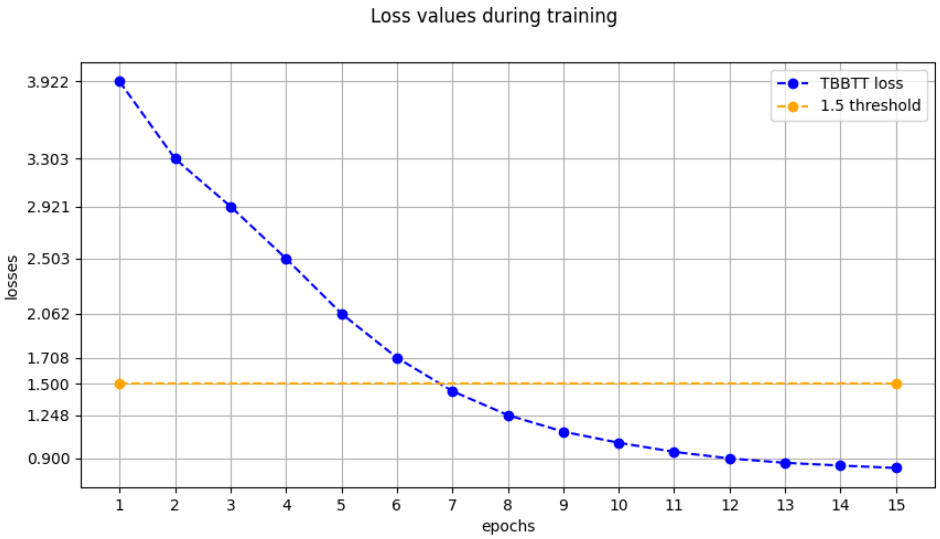

Figure 3: Truncated-BBTT losses

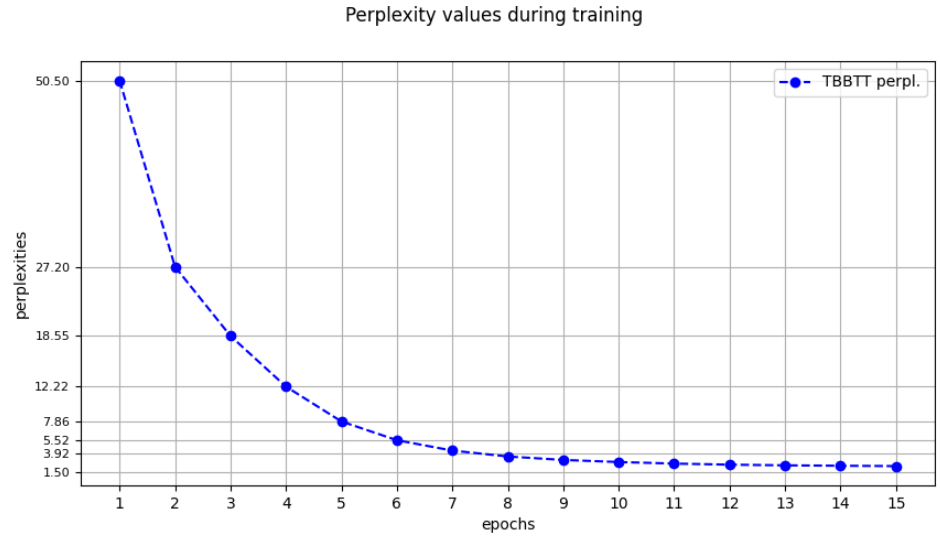

Figure 4: Truncated-BBTT perplexities

1.5 Evaluation - part 2 (5 points)

For this part of the exercise I implemented a small block of code that executes the evaluate() (already mentioned and explained) function to print out the sentences generated by the model from initial seed passed in as argument. The code prints out sentences for both the standard and truncated models.

I first start by defining a list of sentences, tokenize and convert them to integers relative to the index in the vocabulary, and proceed to use them as input for the evaluate function. Notice that now we don't need the dataloader but we do define the input_sentence argument.

```
sentences = ["americans are", "trump will", "a hero of"]
t = TreebankWordTokenizer()
input_sentences = []
for sntc in sentences:
  tokenized = t.tokenize(sntc.lower())+["<EOS>"]
  input_sentences.append([word_to_int[token] for token in tokenized])

for inpt in input_sentences:
  print("Standard-BBTT model")
  evaluate(model_standard,None,int_to_word,device, topk=5, uniform=None
      , max_sntc_length=30, input_sentence=inpt)
  print("TBBTT model")
  evaluate(model,None,int_to_word,device, topk=5, uniform=None,
      max_sntc_length=30, input_sentence=inpt)
```

- **Sampling strategy with topk=5**

```
Original sentence -> americans are
Seed -> americans are
Standard-BBTT model
Topk prompt -> americans are domestic terrorism isn't for donald
    trump
TBBTT model
Topk prompt -> americans are to control the media
------------------------------------------------------------------------

Original sentence -> trump will
Seed -> trump will
Standard-BBTT model
Topk prompt -> trump will u.s. a new climate deal , and still do n
    't do anything about the cause
TBBTT model
Topk prompt -> trump will campaign be at risk , ' says obama
    reports
------------------------------------------------------------------------

Original sentence -> a hero of
Seed -> a hero of
Standard-BBTT model
Topk prompt -> a hero of : a lesson to the fcc war
TBBTT model
Topk prompt -> a hero of statues
```

- **Greedy sampling**

```
Original sentence -> americans are
Seed -> americans are
Standard-BBTT model
Argmax prompt -> americans are divided on the supreme court
TBBTT model
Argmax prompt -> americans are on the most radical aspects of the
    election
------------------------------------------------------------------------
```

```
Original sentence -> trump will
Seed -> trump will
Standard-BBTT model
Argmax prompt -> trump will commencement trip to israel
TBBTT model
Argmax prompt -> trump will campaign be in new york
--------------------------------------------------------------------------------
Original sentence -> a hero of
Seed -> a hero of
Standard-BBTT model
Argmax prompt -> a hero of : a comprehensive nature and twitter
    has made nothing about trump
TBBTT model
Argmax prompt -> a hero of statues
```

We can see that the greedy strategy provides results that are more coherent in comparison to the sampling strategy. Titles in the greedy sample are credible and look like they were generated by a human being, showing a high relation to the topics present in the headlines.

1.6 Bonus question* (5 points)

No, the embedding used for this model could not generalize for the same level of semantic relationship as the word2vec embedding.

The justification for this lies within the size and context of our vocabulary, and the size of the vector space used for the embedding. The embedding used for this model was solely trained on a group of words and sentences that are related to a very small subset of all possible words and contexts. Because of this, semantic relations (even synonymity) are very bound to the context of the headlines.

Furthermore, the fact that word2vec uses an embedding of dimension 300 and we are using 64 already gives us a hint that the used embedding may be very restrictive and not produce the same semantic relation between words.

Another factor to take in consideration is the training objective of our model. The model was trained with a train set that doesn't fully explore the semantic relation of words. That is, its a set composed of politics headlines and thus, the embedding will map words into a vector space where the semantics have the context dependence of the topics that are present in the headlines.

2 Questions [5 points]

Character-level BPE is a subword tokenization that builds tokens by merging most frequent pairs of consecutive characters in an iterative way. Each character is treated as a token. During each iteration, most frequent pairs of consecutive tokens are merged, and this process continues until a predefined vocabulary size is reached. This approach is effective for handling rare words and creating a more adaptive vocabulary.

WordPiece is another subword tokenization that starts with a vocabulary consisting of individual characters and frequently occurring words. It iteratively merges the most likely pair of consecutive tokens based on a language model (or another metric) until a certain vocabulary size is reached. WordPiece is known for its flexibility in handling both frequent and rare words effectively.