ECE 046211 - Technion - Deep Learning

HW3 - Sequential Tasks and Training Methods

Keyboard Shortcuts

- Run current cell: **Ctrl + Enter**
- Run current cell and move to the next: **Shift + Enter**
- Show lines in a code cell: **Esc + L**
- View function documentation: **Shift + Tab** inside the parenthesis or `help(name_of_module)`
- New cell below: **Esc + B**
- Delete cell: **Esc + D, D** (two D's)

Students Information

- Fill in

Name	Campus Email	ID
Student 1	student_1@campus.technion.ac.il	123456789
Student 2	student_2@campus.technion.ac.il	987654321

Submission Guidelines

- Maximal garde: 100.
- Submission only in **pairs**.
  - Please make sure you have registered your group in Moodle (there is a group creation component on the Moodle where you need to create your group and assign members).
- **No handwritten submissions.** You can choose whether to answer in a Markdown cell in this notebook or attach a PDF with your answers.
- SAVE THE NOTEBOOKS WITH THE OUTPUT, CODE CELLS THAT WERE NOT RUN WILL NOT GET ANY POINTS!
- What you have to submit:
  - If you have answered the questions in the notebook, you should submit this file only, with the name: `ece046211_hw3_id1_id2.ipynb`.
  - If you answered the questionss in a different file you should submit a `.zip` file with the name `ece046211_hw3_id1_id2.zip` with content:
    - `ece046211_hw3_id1_id2.ipynb` - the code tasks
    - `ece046211_hw3_id1_id2.pdf` - answers to questions.

- No other file-types (`.py` , `.docx` ...) will be accepted.
* Submission on the course website (Moodle).
* **Latex in Colab** - in some cases, Latex equations may no be rendered. To avoid this, make sure to not use *bullets* in your answers ("* some text here with Latex equations" -> "some text here with Latex equations").

Working Online and Locally

* You can choose your working environment:
  1. `Jupyter Notebook` , **locally** with Anaconda or **online** on Google Colab
     - Colab also supports running code on GPU, so if you don't have one, Colab is the way to go. To enable GPU on Colab, in the menu: `Runtime` $\rightarrow$ `Change Runtime Type` $\rightarrow$ `GPU` .
  2. Python IDE such as PyCharm or Visual Studio Code.
     - Both allow editing and running Jupyter Notebooks.
* Please refer to `Setting Up the Working Environment.pdf` on the Moodle or our GitHub (https://github.com/taldatech/ee046211-deep-learning) to help you get everything installed.
* If you need any technical assistance, please go to our Piazza forum (`hw3` folder) and describe your problem (preferably with images).

Agenda

Part 1 - Theory

* You can choose whether to answser these straight in the notebook (Markdown + Latex) or use another editor (Word, LyX, Latex, Overleaf...) and submit an additional PDF file, **but no handwritten submissions**.
* You can attach additional figures (drawings, graphs,...) in a separate PDF file, just make sure to refer to them in your answers.

- $\LaTeX$ Cheat-Sheet (to write equations)
    - Another Cheat-Sheet

Question 1 - Transformer Encoding

In the following question we will assume we are given a Transformer capable of sentences of up to L tokens, where every token is represented by a d-dimensional vector.

1. Explain what a Positional Encoding is, why is it needed, and how it works.
2. One suggested encoding was to assign a number in range $[0, 1]$ to each word as follows: For a sentence of length $N \leq L$, add

$$\frac{t}{N-1}$$

   to the t-th word. This means we add 0 to the first word and 1 to the final word. What issue can arise from this encoding? Note that the length of each sentence N can differ between sentences.
3. Another suggested encoding was to add 1 to the first word, 2 to the second and so on. Would this be a good encoding? Explain your answer.

From here on out, we will use the following encoding - let $0 \leq t < N, 0 \leq k < d$, we define

$$P_{t,k} = \begin{bmatrix} \sin(\omega_k t) \\ \cos(\omega_k t) \end{bmatrix}$$

where $\omega_k = 10000^{-2k/d}$. The encoding of word t is a d-dimensional vector of pairs $P_{t,i}$:

$$P_t = \begin{bmatrix} \sin(\omega_1 t) \\ \cos(\omega_1 t) \\ \vdots \\ \sin(\omega_{d/2} t) \\ \cos(\omega_{d/2} t) \end{bmatrix}$$

4. Explain why this gives a unique encoding for each word in the sentence regardless of its length N.
5. Show that we can linearly transform $P_{t,k}$ via offset, meaning that for any offset τ there is a matrix

$M_k^\tau \in \mathbb{R}^{2\times 2}$ such that

$$P_{t+\tau,k} = M_k^\tau P_{t,k}$$

Hint: remember that

$$\sin(\alpha+\beta) = \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta)$$

$$\cos(\alpha+\beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)$$

7. Extend this to P_t, show that for $M^\tau \in \mathbb{R}^{d\times d}$

$$P_{t+\tau} = M^\tau P_t$$

Question 2 - Preventing Variance Explosion

This question relates to lectures 8-9 (from slide 7):

Find an initializtion scheme such that

$$\forall l, i, : (1)\ \mathbb{E}\left[F_l(u_l)|u_l\right] = 0,\ (2)\ Var(u_l[i]) = 1,$$

assuming skip connections: $u_{l+1} = u_l + F_l(u_l)$ with a single skip $F_l(u_l) = W_l\phi(u_l) + b_l$ and the activation is ReLU: $\phi(x) = \mathrm{ReLU}(x) = \max(0, x)$.

Question 3 - Recurrent Neural Networks

You are given a recurrent/feedback neural network with LReLU activations $\phi(u) = \max[pu, u]$, with input x_t and a representation $v_t \in \mathbb{R}^d$ that is updated as follows:

$$\forall \tau = 1, 2, \dots t\ : v_\tau = \phi(u_\tau)\ , u_\tau = Wv_{\tau-1} + Bx_\tau,$$

from initialization v_0, and outputs $\hat{y}_t = Cv_t$. The network is trained with GD on a single long series $\{x_\tau, y_\tau\}_{\tau=1}^{t}$ with a cost function $\ell(y_t, \hat{y}_t)$ over the last term in the series.

1. Calculate the exact gradient $\frac{\partial \ell}{\partial W[i,j]}$ using Backpropagation through time (BPTT).
2. Recall that calculating the gradient using the method in the previous section there are two issues for $t \to \infty$: (1) the required computational resources grow indefinitely, and (2) the gradients explode or vanish. For each problem: explain it, provide an example for a method to alleviate it and describe any limitations of this method.

Part 2 - Code Assignments

- You must write your code in this notebook and save it with the output of all of the code cells.
- Additional text can be added in Markdown cells.
- You can use any other IDE you like (PyCharm, VSCode...) to write/debug your code, but for the submission you must copy it to this notebook, run the code and save the notebook with the output.

In [ ]:

```
# this part uses the Wikitext-2 dataset. To access torchtext datasets, please install `torc
# `pip install torchdata` ir `conda install -c pytorch torchdata` in activated environment
# or `!pip install torchdata` on colab.
!pip install torchdata
# notes:
# torch=2.0.0 <-> torchtext 0.15.1
# torch=1.13.0 <-> torchtext 0.14.0
# torch=1.12.1 <-> torchtext 0.13.1
# downgrading torchtext example: !pip install torchtext==0.13.1 --no-deps
# torchtext requires the `portalocker` package to download datasets:
!pip install portalocker
```

In [ ]:

```
# imports for the practice (you can add more if you need)
import numpy as np
import matplotlib.pyplot as plt
import time
import os
import math
```

```python
from typing import Tuple

# pytorch
import torch
from torch import nn, Tensor
import torch.nn.functional as F
from torch.nn import TransformerEncoder, TransformerEncoderLayer
from torch.utils.data import dataset

# torchtext
import torchtext
from torchtext.datasets import WikiText2
from torchtext.data.utils import get_tokenizer
from torchtext.vocab import build_vocab_from_iterator

seed = 211
np.random.seed(seed)
torch.manual_seed(seed)
```

In [ ]:

```python
print(f'pytorch: {torch.__version__}, torchtext: {torchtext.__version__}')
```

Sequence-to-Sequence with Transformers

- In this exercise, you are going to build a language model using PyTroch's Transformer module.
- We will work with the **Wikitext-2** dataset: the WikiText language modeling dataset is a collection of over 100 million tokens extracted from the set of verified Good and Featured articles on Wikipedia.
- After training, you will be able to generate senetences!

Task 1 - Loading and Observing the Data

---

1. Run the following cells that define the functions `batchify` and `data_process` and initialize the tokenizer, vocabulary and the WikiText2 train dataset.
2. Create the train, valid and test data using the provided `batchify` function.
3. Print the shape of `train_data`, write in a comment the meaning of each dimension (e.g. `# [meaning of dim1, meaning of dim2]`).
4. Print the first 20 words of one training sample from `train_data`. Use the vocabulary you built to transfer between tokens to words: `itos = vocab.vocab.get_itos()` will give a "int to string" list.

In [ ]:

```python
def batchify(data, bsz):
    """Divides the data into bsz separate sequences, removing extra elements
    that wouldn't cleanly fit.

    Args:
        data: Tensor, shape [N]
        bsz: int, batch size

    Returns:
        Tensor of shape [N // bsz, bsz]
    """
    seq_len = data.size(0) // bsz
    data = data[:seq_len * bsz]
```

```python
    data = data.view(bsz, seq_len).t().contiguous()
    return data.to(device)
```

In [ ]:

```python
def data_process(raw_text_iter: dataset.IterableDataset) -> Tensor:
    """Converts raw text into a flat Tensor."""
    data = [torch.tensor(vocab(tokenizer(item)), dtype=torch.long) for item in raw_text_ite
    return torch.cat(tuple(filter(lambda t: t.numel() > 0, data)))
```

In [ ]:

```python
train_iter = WikiText2(root="./data", split='train')
tokenizer = get_tokenizer('basic_english')
vocab = build_vocab_from_iterator(map(tokenizer, train_iter), specials=['<unk>'])
vocab.set_default_index(vocab['<unk>'])
```

In [ ]:

```python
# train_iter was "consumed" by the process of building the vocab,
# so we have to create it again
train_iter, val_iter, test_iter = WikiText2(root="./data")
train_data = data_process(train_iter)
val_data = data_process(val_iter)
test_data = data_process(test_iter)

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
```

In [ ]:

```python
batch_size = 20
eval_batch_size = 10
```

In [ ]:

```python
"""
Your Code Here
"""
train_data = # complete
val_data = # complete
test_data = # complete
```

Task 2 - Preparing the Data - Separating to Inputs and Targets

- For a language modeling task, the model needs the following words as `Target`.
    - For example, for the senetence "I have a nice dog", the model will be given "I have a nice" as input, and "have a nice dog" as the target.
- Implement (complete) the function `get_batch(source, i, bptt)` : it generates the input and target sequence for the transformer model. It subdivides the source data into chunks of length `bptt`.
    - For example, for `bptt=2` and at `i=0`, the output of `data, target = get_batch(train_data, i=0, bptt=2)` : `data` will be of shape (2, 20), where the batch size is 20 and `target` will be of length 40 (the target for each element is two words, but we flatten `target`).
    - Example: for `bptt=2`, and the ABCDEFG... characters as input, our batches will be in the form of: `data=[a, b], target=[b, c]`. For `bptt=3` : `data=[a, b, c], target=[b, c, d]` and so on. This one example is a batch.
    - Print a sample from `data` and `target`.

In [ ]:

```python
"""
Your Code Here
"""
def get_batch(source, i, bptt):
    """
    Args:
        source: Tensor, shape [full_seq_len, batch_size]
        i: int
```

```
        bptt: int
    Returns:
        tuple (data, target), where data has shape [seq_len, batch_size] and
        target has shape [seq_len * batch_size]
    """
    seq_len = min(bptt, len(source) - 1 - i)
    data = source[i:i + seq_len]
    target = # compelte
    return data, target
```

Task 3 - Define Hyperparameters and Initialize the Model

- Define the following hyperparameters (`[a, b]` means in the range between `a` and `b`):
  - Embedding size: choose from `[200, 250]`
  - Number of hidden units: choose from `[200, 250]`
  - Number of layers: choose from `[2, 4]`
  - Number of attention heads: choose from `[2, 4]`
  - Dropout: choose from `[0.0, 0.3]`
  - Loss criterion: `nn.CrossEntropyLoss()`
  - Optimizer: choose from `[SGD, Adam, RAdam]`
  - Learning rate: choose from `[5e-3, 5.0]`
  - Learning Scheduler: `torch.optim.lr_scheduler.StepLR(optimizer, 1.0, gamma=0.95)` or any scheduler of your choosing.
  - Transformer LayerNormalization: `post` (`norm_first=False`) or `pre` (`norm_first=True`).
- Intialize an instance of `TransformerModel` (given) and send it to `device`. Note that you need to give it the number of tokens to define the output of the decoder. You should use the number of tokens in the vocabulary. Print the number of tokens, print **all** the chosen hyper-parameters and print the model (`print(model`).

In [ ]:
```
class PositionalEncoding(nn.Module):

    def __init__(self, d_model, dropout=0.1, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0).transpose(0, 1)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + self.pe[:x.size(0), :]
        return self.dropout(x)

class TransformerModel(nn.Module):

    def __init__(self, ntoken, ninp, nhead, nhid, nlayers, dropout=0.5, norm_first=False):
        super(TransformerModel, self).__init__()
        self.pos_encoder = PositionalEncoding(ninp, dropout)
        encoder_layers = TransformerEncoderLayer(ninp, nhead, nhid, dropout, norm_first=nor
        self.transformer_encoder = TransformerEncoder(encoder_layers, nlayers)
        self.encoder = nn.Embedding(ntoken, ninp)
        self.ninp = ninp
```

```
        self.decoder = nn.Linear(ninp, ntoken)

        self.init_weights()

    def generate_square_subsequent_mask(self, sz):
        mask = (torch.triu(torch.ones(sz, sz)) == 1).transpose(0, 1)
        mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask == 1, fl
        return mask

    def init_weights(self):
        initrange = 0.1
        self.encoder.weight.data.uniform_(-initrange, initrange)
        self.decoder.bias.data.zero_()
        self.decoder.weight.data.uniform_(-initrange, initrange)

    def forward(self, src, src_mask):
        src = self.encoder(src) * math.sqrt(self.ninp)
        src = self.pos_encoder(src)
        output = self.transformer_encoder(src, src_mask)
        output = self.decoder(output)
        return output
```

In [ ]:

```
"""
Your Code Here
"""
```

Task 4 - Train and Evaluate the Language Model

- Fill in the missing line in the training code and train the model.
- Use `bptt=35` .
- Use the provided function to evaluate it on the validatation set (after each epoch) and on test test (after training is done). **Print and plot** the results (loss and perplexity).
- If you see that the performance does not improve, go back to Task 3 and re-think you hyper-parameters.

In [ ]:

```
def evaluate(model, eval_data):
    model.eval()  # turn on evaluation mode
    total_loss = 0.
    src_mask = model.generate_square_subsequent_mask(bptt).to(device)
    with torch.no_grad():
        for i in range(0, eval_data.size(0) - 1, bptt):
            data, targets = get_batch(eval_data, i, bptt)
            seq_len = data.size(0)
            if seq_len != bptt:
                src_mask = src_mask[:seq_len, :seq_len]
            output = model(data, src_mask)
            output_flat = output.view(-1, ntokens)
            total_loss += seq_len * criterion(output_flat, targets).item()
    return total_loss / (len(eval_data) - 1)
```

In [ ]:

```
"""
Your Code Here
"""
def train(model, bptt):
    model.train()  # turn on train mode
    total_loss = 0.
    log_interval = 200
    start_time = time.time()
    src_mask = model.generate_square_subsequent_mask(bptt).to(device)
```

```
    num_batches = len(train_data) // bptt
    for batch, i in enumerate(range(0, train_data.size(0) - 1, bptt)):
        data, targets = get_batch(train_data, i, bptt)
        seq_len = data.size(0)
        if seq_len != bptt:  # only on last batch
            src_mask = src_mask[:seq_len, :seq_len]
        output = # complete
        loss = # complete

        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 0.5)
        optimizer.step()

        total_loss += loss.item()
        if batch % log_interval == 0 and batch > 0:
            lr = scheduler.get_last_lr()[0]
            ms_per_batch = (time.time() - start_time) * 1000 / log_interval
            cur_loss = total_loss / log_interval
            ppl = math.exp(cur_loss)
            print(f'| epoch {epoch:3d} | {batch:5d}/{num_batches:5d} batches | '
                  f'lr {lr:02.2f} | ms/batch {ms_per_batch:5.2f} | '
                  f'loss {cur_loss:5.2f} | ppl {ppl:8.2f}')
            total_loss = 0
            start_time = time.time()
```

In [ ]:

```
"""
Your Code Here
"""
best_val_loss = float("inf")
epochs = # complete the number of epochs to run
best_model = None
bptt = 35

for epoch in range(1, epochs + 1):
    epoch_start_time = time.time()
    # complete: call train() here with appropriate paramteters
    val_loss = evaluate(model, val_data)
    print('-' * 89)
    print('| end of epoch {:3d} | time: {:5.2f}s | valid loss {:5.2f} | '
          'valid ppl {:8.2f}'.format(epoch, (time.time() - epoch_start_time),
                                     val_loss, math.exp(val_loss)))
    print('-' * 89)

    if val_loss < best_val_loss:
        best_val_loss = val_loss
        best_model = model

    scheduler.step()
```

Task 5 - Generate Sentences

Use the following function to generate 3 sentences of length 20, and print them. Do they make sense? (you can compare generated sentences over epochs, to see if some logic is gained during training).

In [ ]:

```
def generate(model, vocab, nwords=100, temp=1.0):
    model.eval()
    ntokens = len(vocab)
    itos = vocab.vocab.get_itos()
    model_input = torch.randint(ntokens, (1, 1), dtype=torch.long).to(device)
```

```
    words = []
    with torch.no_grad():
        for i in range(nwords):
            output = model(model_input, None)
            word_weights = output[-1].squeeze().div(temp).exp().cpu()
            word_idx = torch.multinomial(word_weights, 1)[0]
            word_tensor = torch.Tensor([[word_idx]]).long().to(device)
            model_input = torch.cat([model_input, word_tensor], 0)
            word = itos[word_idx]
            words.append(word)
    return words
```

In [ ]:

```
"""
Your code Here
"""
```

Credits

- Icons made by Becris from www.flaticon.com
- Icons from Icons8.com - https://icons8.com
- Datasets from Kaggle - https://www.kaggle.com/