hw1-2-neural-network-tokenization-to-student-2024

January 11, 2025

0.1 Word Tokenizer exercise

In this exercise, you are going to build a set of deep learning models on a (sort of) real world task using pyTorch. PyTorch is a deep learning framwork developed by facebook to provide an easier way to use standard layers and networks.

To complete this exercise, you will need to build deep learning models for word tokenization in Thai ( ) using NECTEC's BEST corpus. You will build one model for each of the following type: - Fully Connected (Feedforward) Neural Network - One-Dimentional Convolution Neural Network (1D-CNN) - Recurrent Neural Network with Gated Recurrent Unit (GRU)

and one more model of your choice to achieve the highest score possible.

We provide the code for data cleaning and some starter code for PyTorch in this notebook but feel free to modify those parts to suit your needs. Feel free to use additional libraries (e.g. scikit-learn) as long as you have a model for each type mentioned above.

Don't forget to change hardware accelerator to GPU in Google Colab.

[1]:
```
!pip install wandb torchinfo huggingface_hub lightning
```

```
Requirement already satisfied: wandb in /usr/local/lib/python3.10/dist-packages
(0.19.1)
Requirement already satisfied: torchinfo in /usr/local/lib/python3.10/dist-
packages (1.8.0)
Requirement already satisfied: huggingface_hub in
/usr/local/lib/python3.10/dist-packages (0.24.7)
Requirement already satisfied: lightning in /usr/local/lib/python3.10/dist-
packages (2.5.0.post0)
Requirement already satisfied: click!=8.0.0,>=7.1 in
/usr/local/lib/python3.10/dist-packages (from wandb) (8.1.7)
Requirement already satisfied: docker-pycreds>=0.4.0 in
/usr/local/lib/python3.10/dist-packages (from wandb) (0.4.0)
Requirement already satisfied: gitpython!=3.1.29,>=1.0.0 in
/usr/local/lib/python3.10/dist-packages (from wandb) (3.1.43)
Requirement already satisfied: platformdirs in /usr/local/lib/python3.10/dist-
packages (from wandb) (4.3.6)
Requirement already satisfied: protobuf!=4.21.0,!=5.28.0,<6,>=3.19.0 in
/usr/local/lib/python3.10/dist-packages (from wandb) (3.20.3)
Requirement already satisfied: psutil>=5.0.0 in /usr/local/lib/python3.10/dist-
packages (from wandb) (5.9.5)
```

```
Requirement already satisfied: pydantic<3,>=2.6 in
/usr/local/lib/python3.10/dist-packages (from wandb) (2.9.2)
Requirement already satisfied: pyyaml in /usr/local/lib/python3.10/dist-packages
(from wandb) (6.0.2)
Requirement already satisfied: requests<3,>=2.0.0 in
/usr/local/lib/python3.10/dist-packages (from wandb) (2.32.3)
Requirement already satisfied: sentry-sdk>=2.0.0 in
/usr/local/lib/python3.10/dist-packages (from wandb) (2.19.2)
Requirement already satisfied: setproctitle in /usr/local/lib/python3.10/dist-
packages (from wandb) (1.3.4)
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-
packages (from wandb) (71.0.4)
Requirement already satisfied: typing-extensions<5,>=4.4 in
/usr/local/lib/python3.10/dist-packages (from wandb) (4.12.2)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-
packages (from huggingface_hub) (3.16.1)
Requirement already satisfied: fsspec>=2023.5.0 in
/usr/local/lib/python3.10/dist-packages (from huggingface_hub) (2024.6.1)
Requirement already satisfied: packaging>=20.9 in
/usr/local/lib/python3.10/dist-packages (from huggingface_hub) (24.1)
Requirement already satisfied: tqdm>=4.42.1 in /usr/local/lib/python3.10/dist-
packages (from huggingface_hub) (4.66.5)
Requirement already satisfied: lightning-utilities<2.0,>=0.10.0 in
/usr/local/lib/python3.10/dist-packages (from lightning) (0.11.9)
Requirement already satisfied: torch<4.0,>=2.1.0 in
/usr/local/lib/python3.10/dist-packages (from lightning) (2.4.1+cu121)
Requirement already satisfied: torchmetrics<3.0,>=0.7.0 in
/usr/local/lib/python3.10/dist-packages (from lightning) (1.6.0)
Requirement already satisfied: pytorch-lightning in
/usr/local/lib/python3.10/dist-packages (from lightning) (2.4.0)
Requirement already satisfied: six>=1.4.0 in /usr/local/lib/python3.10/dist-
packages (from docker-pycreds>=0.4.0->wandb) (1.16.0)
Requirement already satisfied: aiohttp!=4.0.0a0,!=4.0.0a1 in
/usr/local/lib/python3.10/dist-packages (from
fsspec[http]<2026.0,>=2022.5.0->lightning) (3.10.5)
Requirement already satisfied: gitdb<5,>=4.0.1 in
/usr/local/lib/python3.10/dist-packages (from gitpython!=3.1.29,>=1.0.0->wandb)
(4.0.11)
Requirement already satisfied: annotated-types>=0.6.0 in
/usr/local/lib/python3.10/dist-packages (from pydantic<3,>=2.6->wandb) (0.7.0)
Requirement already satisfied: pydantic-core==2.23.4 in
/usr/local/lib/python3.10/dist-packages (from pydantic<3,>=2.6->wandb) (2.23.4)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests<3,>=2.0.0->wandb) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
packages (from requests<3,>=2.0.0->wandb) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests<3,>=2.0.0->wandb) (2.2.3)
```

```
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests<3,>=2.0.0->wandb)
(2024.8.30)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages
(from torch<4.0,>=2.1.0->lightning) (1.13.3)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-
packages (from torch<4.0,>=2.1.0->lightning) (3.3)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages
(from torch<4.0,>=2.1.0->lightning) (3.1.4)
Requirement already satisfied: numpy>1.20.0 in /usr/local/lib/python3.10/dist-
packages (from torchmetrics<3.0,>=0.7.0->lightning) (1.26.4)
Requirement already satisfied: aiohappyeyeballs>=2.3.0 in
/usr/local/lib/python3.10/dist-packages (from
aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]<2026.0,>=2022.5.0->lightning) (2.4.0)
Requirement already satisfied: aiosignal>=1.1.2 in
/usr/local/lib/python3.10/dist-packages (from
aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]<2026.0,>=2022.5.0->lightning) (1.3.1)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.10/dist-
packages (from
aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]<2026.0,>=2022.5.0->lightning) (24.2.0)
Requirement already satisfied: frozenlist>=1.1.1 in
/usr/local/lib/python3.10/dist-packages (from
aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]<2026.0,>=2022.5.0->lightning) (1.4.1)
Requirement already satisfied: multidict<7.0,>=4.5 in
/usr/local/lib/python3.10/dist-packages (from
aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]<2026.0,>=2022.5.0->lightning) (6.1.0)
Requirement already satisfied: yarl<2.0,>=1.0 in /usr/local/lib/python3.10/dist-
packages (from
aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]<2026.0,>=2022.5.0->lightning) (1.11.1)
Requirement already satisfied: async-timeout<5.0,>=4.0 in
/usr/local/lib/python3.10/dist-packages (from
aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]<2026.0,>=2022.5.0->lightning) (4.0.3)
Requirement already satisfied: smmap<6,>=3.0.1 in
/usr/local/lib/python3.10/dist-packages (from
gitdb<5,>=4.0.1->gitpython!=3.1.29,>=1.0.0->wandb) (5.0.1)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.10/dist-packages (from
jinja2->torch<4.0,>=2.1.0->lightning) (2.1.5)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in
/usr/local/lib/python3.10/dist-packages (from
sympy->torch<4.0,>=2.1.0->lightning) (1.3.0)
```

[2]:
```
# Run setup code
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import torch
from sklearn.metrics import accuracy_score
from huggingface_hub import hf_hub_download
from tqdm import tqdm

%matplotlib inline

# To guarantee reproducible results
torch.manual_seed(5420)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
np.random.seed(5420)
```

0.2 Wandb Setup

We also encourage you to use Wandb which will help you log and visualize your training process.

1. Register Wandb account (and confirm your email)
2. `wandb login` and copy paste the API key when prompt

[3]:
```
!wandb login
```

```
wandb: WARNING If you're specifying your api key in code,
ensure this code is not shared publicly.
wandb: WARNING Consider setting the WANDB_API_KEY
environment variable, or running `wandb login` from the command line.
wandb: Appending key for api.wandb.ai to your netrc file:
/root/.netrc
```

[4]:
```
import wandb
```

[5]:
```
#Check GPU is available
torch.cuda.device_count()
```

[5]:
```
2
```

[6]:
```
#Download dataset
hf_hub_download(repo_id="iristun/corpora", filename="corpora.tar.gz",␣
 ↪repo_type="dataset", local_dir=".")
```

[6]:
```
'corpora.tar.gz'
```

[7]:
```
!tar xvf corpora.tar.gz
```

```
corpora/
corpora/mnist_data/
corpora/mnist_data/t10k-images-idx3-ubyte.gz
corpora/mnist_data/train-images-idx3-ubyte.gz
corpora/mnist_data/.ipynb_checkpoints/
```

```
corpora/mnist_data/vis_utils.py
corpora/mnist_data/__init__.py
corpora/mnist_data/load_mnist.py
corpora/mnist_data/train-labels-idx1-ubyte.gz
corpora/mnist_data/t10k-labels-idx1-ubyte.gz
corpora/BEST/
corpora/BEST/test/
corpora/BEST/test/df_best_article_test.csv
corpora/BEST/test/df_best_encyclopedia_test.csv
corpora/BEST/test/df_best_novel_test.csv
corpora/BEST/test/df_best_news_test.csv
corpora/BEST/train/
corpora/BEST/train/df_best_encyclopedia_train.csv
corpora/BEST/train/df_best_article_train.csv
corpora/BEST/train/df_best_news_train.csv
corpora/BEST/train/df_best_novel_train.csv
corpora/BEST/val/
corpora/BEST/val/df_best_encyclopedia_val.csv
corpora/BEST/val/df_best_news_val.csv
corpora/BEST/val/df_best_article_val.csv
corpora/BEST/val/df_best_novel_val.csv
corpora/.ipynb_checkpoints/
corpora/.ipynb_checkpoints/Word_Tokenizer.new-checkpoint.ipynb
corpora/.ipynb_checkpoints/BackProp-checkpoint.ipynb
corpora/.ipynb_checkpoints/Word_Tokenizer_backup-checkpoint.ipynb
corpora/.ipynb_checkpoints/char2vec-checkpoint.ipynb
corpora/.ipynb_checkpoints/Word_Tokenizer-checkpoint.ipynb
corpora/cattern/
corpora/cattern/gradient_check.py
corpora/cattern/.ipynb_checkpoints/
corpora/cattern/__init__.py
corpora/cattern/data_utils.py
corpora/wiki/
corpora/wiki/thwiki_chk.txt
```

For simplicity, we are going to build a word tokenization model which is a binary classification model trying to predict whether a character is the begining of the word or not (if it is, then there is a space in front of it) and without using any knowledge about type of character (vowel, number, English character etc.).

For example,

‘ ’ -> ‘ ’

will have these true labels:

[(,1), (,0), (,0) (,1), (,0), (,1), (-,0), (,0), (,1), (-,0), (,0), (,1), (,0), (,0)]

In this task, we will use only main character you are trying to predict and the characters that surround it (the context) as features. However, you can imagine that a more complex model will try to include more knowledge about each character into the model. You can do that too if you feel like it.

[8]:
```
# Create a character map
CHARS = [
  '\n', ' ', '!', '"', '#', '$', '%', '&', "'", '(', ')', '*', '+',
  ',', '-', '.', '/', '0', '1', '2', '3', '4', '5', '6', '7', '8',
  '9', ':', ';', '<', '=', '>', '?', '@', 'A', 'B', 'C', 'D', 'E',
  'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R',
  'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '[', '\\', ']', '^', '_',
  'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
  'n', 'o', 'other', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y',
  'z', '}', '~', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
  ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
  ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
  ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '', ' ',
  ' ', '', '', '', '', '', '', '', ' ', ' ', ' ', ' ', ' ',
  ' ', ' ', '', '', '', '', '', '', '', ' ', ' ', ' ', ' ',
  ' ', ' ', ' ', ' ', ' ', ' ', '‘', '’', '\ufeff'
]
CHARS_MAP = {v: k for k, v in enumerate(CHARS)}
```

[9]:
```
def create_n_gram_df(df, n_pad):
  """
  Given an input dataframe, create a feature dataframe of shifted characters
  Input:
  df: timeseries of size (N)
  n_pad: the number of context. For a given character at position [idx],
    character at position [idx-n_pad/2 : idx+n_pad/2] will be used
    as features for that character.

  Output:
  dataframe of size (N * n_pad) which each row contains the character,
    n_pad_2 characters to the left, and n_pad_2 characters to the right
    of that character.
  """
  n_pad_2 = int((n_pad - 1)/2)
  for i in range(n_pad_2):
      df['char-{}'.format(i+1)] = df['char'].shift(i + 1)
      df['char{}'.format(i+1)] = df['char'].shift(-i - 1)
  return df[n_pad_2: -n_pad_2]
```

[10]:
```
def prepare_feature(best_processed_path, option='train'):
  """
  Transform the path to a directory containing processed files
  into a feature matrix and output array
  Input:
  best_processed_path: str, path to a processed version of the BEST dataset
```

```
    option: str, 'train' or 'test'
    """
    # we use padding equals 21 here to consider 10 characters to the left
    # and 10 characters to the right as features for the character in the middle
    n_pad = 21
    n_pad_2 = int((n_pad - 1)/2)
    pad = [{'char': ' ', 'target': True}]
    df_pad = pd.DataFrame(pad * n_pad_2)

    df = []
    # article types in BEST corpus
    article_types = ['article', 'encyclopedia', 'news', 'novel']
    for article_type in article_types:
        df.append(pd.read_csv(os.path.join(best_processed_path, option,␣
↪'df_best_{}_{}.csv'.format(article_type, option))))

    df = pd.concat(df)
    # pad with empty string feature
    df = pd.concat((df_pad, df, df_pad))

    # map characters to numbers, use 'other' if not in the predefined character␣
↪set.
    df['char'] = df['char'].map(lambda x: CHARS_MAP.get(x, 80))

    # Use nearby characters as features
    df_with_context = create_n_gram_df(df, n_pad=n_pad)

    char_row = ['char' + str(i + 1) for i in range(n_pad_2)] + \
               ['char-' + str(i + 1) for i in range(n_pad_2)] + ['char']

    # convert pandas dataframe to numpy array to feed to the model
    x_char = df_with_context[char_row].to_numpy()
    y = df_with_context['target'].astype(int).to_numpy()

    return x_char, y
```

Before running the following commands, we must inform you that our data is quite large and loading the whole dataset at once will **use a lot of memory (~6 GB after processing and up to ~12GB while processing)**. We expect you to be running this on Google Cloud or Google Colab so that you will not run into this problem. But, if, for any reason, you have to run this on your PC or machine with not enough memory, you might need to write a data generator to process a few entries at a time then feed it to the model while training.

[11]:
```
# Path to the preprocessed data
best_processed_path = 'corpora/BEST'
```

[12]:
```
# Load preprocessed BEST corpus
x_train_char, y_train = prepare_feature(best_processed_path, option='train')
x_val_char, y_val = prepare_feature(best_processed_path, option='val')
x_test_char, y_test = prepare_feature(best_processed_path, option='test')

# As a sanity check, we print out the size of the training, val, and test data.
print('Training data shape: ', x_train_char.shape)
print('Training data labels shape: ', y_train.shape)
print('Validation data shape: ', x_val_char.shape)
print('Validation data labels shape: ', y_val.shape)
print('Test data shape: ', x_test_char.shape)
print('Test data labels shape: ', y_test.shape)
```

```
Training data shape:  (16461637, 21)
Training data labels shape:  (16461637,)
Validation data shape:  (2035694, 21)
Validation data labels shape:  (2035694,)
Test data shape:  (2271932, 21)
Test data labels shape:  (2271932,)
```

[13]:
```
# Print some entry from the data to make sure it is the same as what you think.
print('First 3 features: ', x_train_char[:3])
print('First 30 class labels', y_train[:30])
```

```
First 3 features:  [[112. 140. 114. 148. 130. 142.  94. 142. 128. 128.   1.   1.
1.   1.
    1.   1.   1.   1.   1.   1.  97.]
 [140. 114. 148. 130. 142.  94. 142. 128. 128. 141.  97.   1.   1.   1.
    1.   1.   1.   1.   1.   1. 112.]
 [114. 148. 130. 142.  94. 142. 128. 128. 141. 109. 112.  97.   1.   1.
    1.   1.   1.   1.   1.   1. 140.]]
First 30 class labels [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 0
0]
```

[14]:
```
#print char of feature 1
char = np.array(CHARS)

#A function for displaying our features in text
def print_features(tfeature,label,index):
    feature = np.array(tfeature[index],dtype=int).reshape(21,1)
    #Convert to string
    char_list = char[feature]
    left = ''.join(reversed(char_list[10:20].reshape(10))).replace(" ", "")
    center = ''.join(char_list[20])
    right =  ''.join(char_list[0:10].reshape(10)).replace(" ", "")
    word = ''.join([left,' ',center,' ',right])
    print(center + ': ' + word + "\tpred = "+str(label[index]))
```

```python
for ind in range(0,30):
    print_features(x_train_char,y_train,ind)
```

```
:                pred = 1
:               pred = 0
:               pred = 0
:              pred = 0
:              pred = 0
:             pred = 0
:             pred = 0
:            pred = 0
:                   pred = 0
:                   pred = 0
:                  pred = 0
:                  pred = 0
:                  pred = 0
:                  pred = 0
:                  pred = 0
:                  pred = 0
:                  pred = 0
:                  pred = 0
:                   pred = 0
:                  pred = 0
:                  pred = 1
:                  pred = 0
:                  pred = 0
:                  pred = 1
:                  pred = 0
:                  pred = 0
:                  pred = 0
:                  pred = 1
:                 pred = 0
:                  pred = 0
```

Now, you are going to define the model to be used as your classifier. If you are using Pytorch, please follow the guideline we provide below. You can find more about PyTorch model structure here documentation.

In short, you need to inherit the class `torch.nn.Module` and override the constructor and the method `forward` as shown below:

```python
Class Model(torch.nn.Module):
  def __init__(self):
    super(Model, self).__init__()
    #init layer
  def forward(self, x):
    #forward pass the model
```

Also, beware that complex model requires more time to train and your dataset is already quite large. We tested it with a simple 1-hidden-layered feedforward nueral network and it used ~5 mins to train 1 epoch.

1 Three-Layer Feedforward Neural Networks

Below, we provide you the code for creating a 3-layer fully connected neural network in PyTorch. This will also serve as the baseline for your other models. Run the code below while making sure you understand what you are doing. Then, report the results.

[15]:
```python
import torch.nn.functional as F
from torchinfo import summary

class SimpleFeedforwardNN(torch.nn.Module):
  def __init__(self):
    super(SimpleFeedforwardNN, self).__init__()

    self.mlp1 = torch.nn.Linear(21, 100)
    self.mlp2 = torch.nn.Linear(100, 100)
    self.mlp3 = torch.nn.Linear(100, 100)
    self.cls_head = torch.nn.Linear(100, 1)

  def forward(self, x):
    x = F.relu(self.mlp1(x))
    x = F.relu(self.mlp2(x))
    x = F.relu(self.mlp3(x))
    x = self.cls_head(x)
    out = torch.sigmoid(x)
    return out

model = SimpleFeedforwardNN() #Initialize model
model.cuda() #specify the location that it is in the GPU
summary(model, input_size=(1, 21), device='cuda') #summarize the model
```

[15]:
```
================================================================================
==========
Layer (type:depth-idx)                   Output Shape              Param #
================================================================================
==========
SimpleFeedforwardNN                       [1, 1]                    --
 Linear: 1-1                             [1, 100]                  2,200
 Linear: 1-2                             [1, 100]                  10,100
 Linear: 1-3                             [1, 100]                  10,100
 Linear: 1-4                             [1, 1]                    101
================================================================================
==========
Total params: 22,501
Trainable params: 22,501
Non-trainable params: 0
```

```
Total mult-adds (M): 0.02
================================================================================
==========
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.09
Estimated Total Size (MB): 0.09
================================================================================
==========
```

1.0.1 Test whether the model is working as intended by passing dummy input.

[16]:
```
test_X = torch.tensor(np.zeros((64, 21)), dtype = torch.float).cuda()
print(model(test_X).shape)
```

```
torch.Size([64, 1])
```

A tensor is very similar to numpy, and many numpy functions has a tensor equivalent.

[17]:
```
example_tensor = torch.arange(6)
print(example_tensor.shape)

# addition and multiplication
print(example_tensor * 2 + 1)

# resize
example_tensor = example_tensor.view((2, 3))
print(example_tensor)

example_tensor1 = torch.
  ↪tensor([[[[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]]], dtype = torch.
  ↪float)
example_tensor2 = torch.ones_like(example_tensor1)
print(example_tensor1.shape, example_tensor2.shape)
print(example_tensor1)
print(example_tensor2)
print(example_tensor1.matmul(example_tensor2))
```

```
torch.Size([6])
tensor([ 1,  3,  5,  7,  9, 11])
tensor([[0, 1, 2],
        [3, 4, 5]])
torch.Size([1, 1, 4, 4]) torch.Size([1, 1, 4, 4])
tensor([[[[ 1.,  2.,  3.,  4.],
          [ 5.,  6.,  7.,  8.],
          [ 9., 10., 11., 12.],
          [13., 14., 15., 16.]]]])
tensor([[[[1., 1., 1., 1.],
          [1., 1., 1., 1.],
          [1., 1., 1., 1.],
          [1., 1., 1., 1.]]]])
tensor([[[[10., 10., 10., 10.],
          [26., 26., 26., 26.],
          [42., 42., 42., 42.],
          [58., 58., 58., 58.]]]])
```

To debug, you can always just try passing variables through individual layers by yourself.

```
[18]: mlp_test = torch.nn.Linear(21, 3).cuda() # a MLP that has 21 input nodes and 3␣
       ↪output nodes
      print(x_train_char[:4])
      print(x_train_char[:4].shape)
      test_input = torch.tensor(x_train_char[:4], dtype = torch.float).cuda()
      print(mlp_test(test_input).shape)
      print(mlp_test(test_input))
```

```
[[112. 140. 114. 148. 130. 142.  94. 142. 128. 128.   1.   1.   1.   1.
    1.   1.   1.   1.   1.   1.  97.]
 [140. 114. 148. 130. 142.  94. 142. 128. 128. 141.  97.   1.   1.   1.
    1.   1.   1.   1.   1.   1. 112.]
 [114. 148. 130. 142.  94. 142. 128. 128. 141. 109. 112.  97.   1.   1.
    1.   1.   1.   1.   1.   1. 140.]
 [148. 130. 142.  94. 142. 128. 128. 141. 109. 117. 140. 112.  97.   1.
    1.   1.   1.   1.   1.   1. 114.]]
(4, 21)
torch.Size([4, 3])
tensor([[-10.7609,  35.0017, -75.6997],
        [-27.3163,  26.4542, -69.2510],
        [ -9.0613,  42.3437, -91.1748],
        [-21.3748,  35.3703, -57.5476]], device='cuda:0',
       grad_fn=<AddmmBackward0>)
```

1.0.2 Typical PyTorch training loop

Before the training loop begins, a data loader respondsible for generating data in a trainable format has to be created first. In Pytorch, `torch.utils.data.Dataloader` is a readily available class that are commonly used for data preparation. The dataloader takes the object `torch.utils.data.Dataset` as an input. An example of a data loader for this task is shown below. You can read more about the class `Dataset` here https://pytorch.org/tutorials/beginner/basics/data_tutorial.html.

1.0.3 Converting the data into trainable format

In order to train the model using the PyTorch frame, the data has to be converted into `Tensor type`. In the cell below, we convert the data into `cuda.FloatTensor` type. You can read more about `Tensor` data type here : https://pytorch.org/docs/stable/tensors.html.

[19]:
```
class Dataset(torch.utils.data.Dataset):
  'Characterizes a dataset for PyTorch'
  def __init__(self, X, Y, dtype = 'float'):
        'Initialization'
        self.X = X
        self.Y = Y.reshape(-1, 1)
        if(dtype == 'float'):
          self.X = torch.tensor(self.X, dtype = torch.float).cuda()
        elif(dtype == 'long'):
          self.X = torch.tensor(self.X, dtype = torch.long).cuda()
        self.Y = torch.tensor(self.Y, dtype = torch.float).cuda()

  def __len__(self):
        'Denotes the total number of samples'
        return len(self.X)

  def __getitem__(self, index):
        'Generates one sample of data'
        # Select sample
        x = self.X[index]
        y = self.Y[index, :]
        return x, y
```

In the block below, we initialized the hyperparameters used for the training process. Normally, the optimizer, objective function, and training schedule is initialized here.

[20]:
```
from torch.utils.data import DataLoader
import torch.optim as optim

#hyperparameter initialization
NUM_EPOCHS = 3
criterion = torch.nn.BCELoss(reduction = 'none')
BATCHS_SIZE = 512
optimizer_class = optim.Adam
optimizer_params = {'lr': 5e-4}

config = {
    'architecture': 'simpleff',
    'epochs': NUM_EPOCHS,
    'batch_size': BATCHS_SIZE,
    'optimizer_params': optimizer_params,
}

train_loader = DataLoader( Dataset(x_train_char, y_train, dtype = 'float'),␣
 ↪batch_size = BATCHS_SIZE)
val_loader = DataLoader( Dataset(x_val_char, y_val, dtype = 'float'),␣
 ↪batch_size = BATCHS_SIZE)
```

```
test_loader = DataLoader( Dataset(x_test_char, y_test, dtype = 'float'),␣
 ↪batch_size = BATCHS_SIZE)
```

1.0.4 Pytorch Lightning Module

In most of our labs, we will use Pytorch Lightning. PyTorch Lightning is an open-source Python library that provides a high-level interface for PyTorch, making it easier/faster to use. It is considered an industry standard and is used widely on recent huggingface tutorials. Pytorch Lightning makes scaling training of deep learning models simple and hardware agnostic.

If you are not familiar with Lightning, you are encouraged to study from this simple tutorial.

[21]:
```
import pytorch_lightning as pl
from pytorch_lightning.callbacks import ModelCheckpoint
from torchmetrics.functional import accuracy

class LightningModel(pl.LightningModule):
    def __init__(
        self,
        model=SimpleFeedforwardNN(),
        criterion=criterion,
        optimizer_class=optim.Adam,
        optimizer_params={'lr': 5e-4}
    ):
        super().__init__()
        self.model = model
        self.criterion = criterion
        self.optimizer_class = optimizer_class
        self.optimizer_params = optimizer_params

    def forward(self, x):
        return self.model(x)

    def training_step(self, batch, batch_idx):
        X_train, Y_train = batch
        Y_pred = self.model(X_train)
        loss = self.criterion(Y_pred, Y_train).mean()
        self.log('train_loss', loss, on_step=True, on_epoch=True)
        return loss

    def validation_step(self, batch, batch_idx):
        X_val, Y_val = batch
        Y_pred = self.model(X_val)
        loss = self.criterion(Y_pred, Y_val).mean()
        self.log('val_loss', loss, on_step=False, on_epoch=True)

        # Convert probalities to classes.
        val_pred = (Y_pred >= 0.5).float()
        # Calculate accuracy.
        val_acc = accuracy(val_pred, Y_val, task="binary")

        self.log('val_accuracy', val_acc, on_step=False, on_epoch=True)
        return {'val_loss': loss, 'val_accuracy': val_acc}

    def configure_optimizers(self):
        return self.optimizer_class(self.parameters(), **self.optimizer_params)
```

1.0.5 Initialize LightningModel and Trainer

[22]:
```
# Initialize LightningModel.
lightning_model = LightningModel(
  model,
  criterion,
  optimizer_class,
  optimizer_params,
)
# Define checkpoint.
feedforward_nn_checkpoint = ModelCheckpoint(
  monitor="val_accuracy",
  mode="max",
  save_top_k=1,
  dirpath="./checkpoints",
  filename='feedforward_nn'
)
# Initialize Trainer
trainer = pl.Trainer(
  max_epochs=NUM_EPOCHS,
  logger=pl.loggers.WandbLogger(),
  callbacks=[feedforward_nn_checkpoint],
  accelerator="gpu",
  devices=1,
)
```

1.0.6 Starting the training loop

[23]:
```
# # Initialize wandb to log the losses from each step.
# wandb.init(
#     project='simpleff',
#     config=config,
# )
# # Fit model.
# trainer.fit(lightning_model, train_loader, val_loader)

# print(f"Best model is saved at {feedforward_nn_checkpoint.best_model_path}")
```

1.0.7 Evaluate the model performance on the test set

[24]:
```python
from sklearn.metrics import f1_score,precision_score,recall_score

################################################################################
# A function to evaluate your model. This function must take test dataloader   #
# and the input model to return f-score, precision, and recall of the model.   #
################################################################################
def evaluate(test_loader, model):
  """
  Evaluate model on the splitted 10 percent testing set.
  """
  model.eval()
  with torch.no_grad():
    test_loss = []
    test_pred = []
    test_true = []
    for X_test, Y_test in tqdm(test_loader):
      Y_pred = model(X_test)
      loss = criterion(Y_pred, Y_test)
      test_loss.append(loss)
      test_pred.append(Y_pred)
      test_true.append(Y_test)

    avg_test_loss = torch.cat(test_loss, axis = 0).mean().item()
    test_pred = torch.cat(test_pred, axis = 0).cpu().detach().numpy()
    test_true = torch.cat(test_true, axis = 0).cpu().detach().numpy()

    prob_to_class = lambda p: 1 if p[0]>=0.5 else 0
    test_pred = np.apply_along_axis(prob_to_class,1,test_pred)

    acc = accuracy_score(test_true, test_pred)
    f1score = f1_score(test_true, test_pred)
    precision = precision_score(test_true, test_pred)
    recall = recall_score(test_true, test_pred)

  return {
    "accuracy": acc,
    "f1_score": f1score,
    "precision": precision,
    "recall": recall
  }
```

[25]:
```python
# Load best model and evaluate it.

# best_model_path = feedforward_nn_checkpoint.best_model_path
# # best_model_path = ... # Insert if you have already trained this model.
# best_model = LightningModel.load_from_checkpoint(best_model_path,␣
 ↪model=SimpleFeedforwardNN())

best_model = SimpleFeedforwardNN()
state_dict = torch.load('/kaggle/input/
 ↪hw1-2-neural-network-tokenization-to-student-2024/pytorch/default/3/
 ↪simpleff_model_weights.pth', map_location='cuda', weights_only=True)
new_state_dict = {key.replace("model.", ""): value for key, value in state_dict.
 ↪items()}
best_model.load_state_dict(new_state_dict)
best_model = best_model.cuda()

result = evaluate(test_loader, best_model)

wandb.finish()
print(result)
```

```
100%|     | 4438/4438 [00:26<00:00, 164.56it/s]

{'accuracy': 0.8914483356015938, 'f1_score': 0.8069875844451818, 'precision':
0.7996888509043643, 'recall': 0.8144207757742341}
```

SimpleFeedforwardNN
'accuracy': 0.8914483356015938,
'f1_score': 0.8069875844451818,
'precision': 0.7996888509043643,
'recall': 0.8144207757742341

2 Debugging

In order to understand what is going on in your model and where the error is, you should try looking at the inputs your model made wrong predictions.

In this task, write a function to print the characters on test data that got wrong prediction along with its context of size 10 (from [x-10] to [x+10]). Examine a fews of those and write your assumption on where the model got wrong prediction.

[26]:
```
# TODO#1
# Write code to show a few of the errors the models made.

def show_model_errors(test_loader, model, char_map):
    model.eval()
    with torch.no_grad():
        for X_test, Y_test in test_loader:
            Y_pred = model(X_test)
            Y_pred_classes = (Y_pred >= 0.5).float()

            errors = (Y_pred_classes.flatten() != Y_test.flatten())
            for i in range(len(errors)):
                if errors[i]:
                    print_features(X_test.cpu().numpy(), Y_test.cpu().numpy(),␣
↪i)
                    print(f"True Label: {int(Y_test[i].item())}, Predicted␣
↪Label: {int(Y_pred_classes[i].item())}")
                    print("-" * 50)

            break

show_model_errors(test_loader, best_model, CHARS)
```

```
:               pred = [0.]
True Label: 0, Predicted Label: 1
--------------------------------------------------
 :         :    pred = [0.]
True Label: 0, Predicted Label: 1
--------------------------------------------------
 :                  pred = [1.]
True Label: 1, Predicted Label: 0
--------------------------------------------------
 :          The      pred = [1.]
True Label: 1, Predicted Label: 0
--------------------------------------------------
 :         TheRe       pred = [0.]
True Label: 0, Predicted Label: 1
--------------------------------------------------
 :        TheRefo       pred = [1.]
True Label: 1, Predicted Label: 0
--------------------------------------------------
 :        TheRefor       pred = [0.]
True Label: 0, Predicted Label: 1
--------------------------------------------------
e: Perspectiv e            pred = [0.]
True Label: 0, Predicted Label: 1
--------------------------------------------------
 :                  pred = [0.]
True Label: 0, Predicted Label: 1
--------------------------------------------------
 :                  pred = [1.]
True Label: 1, Predicted Label: 0
--------------------------------------------------
 :                  pred = [1.]
True Label: 1, Predicted Label: 0
--------------------------------------------------
 : "         "       pred = [1.]
```

```
True Label: 1, Predicted Label: 0
--------------------------------------------------
 :        "          pred = [0.]
True Label: 0, Predicted Label: 1
--------------------------------------------------
 : "                 pred = [1.]
True Label: 1, Predicted Label: 0
--------------------------------------------------
 :                  pred = [1.]
True Label: 1, Predicted Label: 0
--------------------------------------------------
 :                    pred = [0.]
True Label: 0, Predicted Label: 1
--------------------------------------------------
 :                    pred = [0.]
True Label: 0, Predicted Label: 1
--------------------------------------------------
 :                     pred = [0.]
True Label: 0, Predicted Label: 1
--------------------------------------------------
 :                     pred = [0.]
True Label: 0, Predicted Label: 1
--------------------------------------------------
 :                    pred = [0.]
True Label: 0, Predicted Label: 1
--------------------------------------------------
 :                   pred = [0.]
True Label: 0, Predicted Label: 1
--------------------------------------------------
 :             )       pred = [1.]
True Label: 1, Predicted Label: 0
--------------------------------------------------
 :                   pred = [1.]
True Label: 1, Predicted Label: 0
--------------------------------------------------
 :                   pred = [0.]
True Label: 0, Predicted Label: 1
--------------------------------------------------
:                  pred = [0.]
True Label: 0, Predicted Label: 1
--------------------------------------------------
 :                  pred = [1.]
True Label: 1, Predicted Label: 0
--------------------------------------------------
 :                   pred = [1.]
True Label: 1, Predicted Label: 0
--------------------------------------------------
 :                  pred = [0.]
```

```
True Label: 0, Predicted Label: 1
--------------------------------------------------
 :                  pred = [1.]
True Label: 1, Predicted Label: 0
--------------------------------------------------
 :                 pred = [0.]
True Label: 0, Predicted Label: 1
--------------------------------------------------
 :                  pred = [0.]
True Label: 0, Predicted Label: 1
--------------------------------------------------
 :                   pred = [0.]
True Label: 0, Predicted Label: 1
--------------------------------------------------
 :                    pred = [1.]
True Label: 1, Predicted Label: 0
--------------------------------------------------
:                     pred = [0.]
True Label: 0, Predicted Label: 1
--------------------------------------------------
 :                    pred = [1.]
True Label: 1, Predicted Label: 0
--------------------------------------------------
```

3 Write your answer here

Your answer: TODO#2

When positive is begining of the word.

False Positive (FP):
 :   TheRe pred = [0.]
True Label: 0, Predicted Label: 1
Model   ‘ ’   ‘ ’   ‘   ’

False Negative (FN):
 :   pred = [1.]
True Label: 1, Predicted Label: 0
Model   ‘   ’   data   ‘   ’

4 Dropout

You might notice that the 3-layered feedforward does not use dropout at all. Now, try adding dropout to the model, run, and report the result again.

```
[27]: ################################################################################
      # TODO#3:                                                                      #
      # Write a model class that return feedforward model with dropout.              #
      ################################################################################
      import torch.nn.functional as F

      class SimpleFeedforwardNNWDropout(torch.nn.Module):
          def __init__(self, dropout_prob=0.5):
              super(SimpleFeedforwardNNWDropout, self).__init__()

              self.mlp1 = torch.nn.Linear(21, 100)
              self.dropout1 = torch.nn.Dropout(p=dropout_prob)
              self.mlp2 = torch.nn.Linear(100,100)
              self.dropout2 = torch.nn.Dropout(p=dropout_prob)
              self.mlp3 = torch.nn.Linear(100,100)
              self.dropout3 = torch.nn.Dropout(p=dropout_prob)
              self.cls_head = torch.nn.Linear(100, 1)

          def forward(self, x):
              x = F.relu(self.mlp1(x))
              x = self.dropout1(x)
              x = F.relu(self.mlp2(x))
              x = self.dropout2(x)
              x = F.relu(self.mlp3(x))
              x = self.dropout3(x)
              x = self.cls_head(x)
              out = torch.sigmoid(x)
              return out

      model = SimpleFeedforwardNNWDropout()
      model.cuda()
      summary(model, input_size=(1, 21), device='cuda')
```

[27]:
```
================================================================================
==========
Layer (type:depth-idx)                   Output Shape              Param #
================================================================================
==========
SimpleFeedforwardNNWDropout               [1, 1]                    --
 Linear: 1-1                             [1, 100]                  2,200
 Dropout: 1-2                            [1, 100]                  --
 Linear: 1-3                             [1, 100]                  10,100
 Dropout: 1-4                            [1, 100]                  --
 Linear: 1-5                             [1, 100]                  10,100
 Dropout: 1-6                            [1, 100]                  --
 Linear: 1-7                             [1, 1]                    101
================================================================================
==========
Total params: 22,501
```

```
Trainable params: 22,501
Non-trainable params: 0
Total mult-adds (M): 0.02
================================================================================
==========
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.09
Estimated Total Size (MB): 0.09
================================================================================
==========
```

[28]:
```
################################################################################
# TODO#4:                                                                      #
# Write code that performs a training process. Select your batch size carefully#
# as it will affect your model's ability to converge and                       #
# time needed for one epoch.                                                   #
################################################################################
# Complete the code to train your model with dropout
model_nn_with_dropout = SimpleFeedforwardNNWDropout().cuda()
summary(model_nn_with_dropout,  input_size=(64, 21), device='cuda') #summarize␣
 ↪the model

################################################################################
#                             WRITE YOUR CODE BELOW                            #
################################################################################


# Hyperparameter initialization
import torch.optim as optim
NUM_EPOCHS = 3
criterion = torch.nn.BCELoss(reduction = 'none')
BATCHS_SIZE = 2048
optimizer_class = optim.Adam
optimizer_params = {'lr': 5e-4}

config = {
    'architecture': 'simpleffdropout',
    'epochs': NUM_EPOCHS,
    'batch_size': BATCHS_SIZE,
    'optimizer_params': optimizer_params,
}

# DataLoader
from torch.utils.data import DataLoader
train_loader = DataLoader( Dataset(x_train_char, y_train, dtype = 'float'),␣
 ↪batch_size = BATCHS_SIZE)
```

```
val_loader = DataLoader( Dataset(x_val_char, y_val, dtype = 'float'),␣
 ↪batch_size = BATCHS_SIZE)
test_loader = DataLoader( Dataset(x_test_char, y_test, dtype = 'float'),␣
 ↪batch_size = BATCHS_SIZE)

# Initialize LightningModel.
lightning_model = LightningModel(
  model_nn_with_dropout,
  criterion,
  optimizer_class,
  optimizer_params,
)
# Define checkpoint.
feedforward_nn_dropout_checkpoint = ModelCheckpoint(
  monitor="val_accuracy",
  mode="max",
  save_top_k=1,
  dirpath="./checkpoints",
  filename='feedforward_nn_dropout'
)
# Initialize Trainer
trainer = pl.Trainer(
  max_epochs=NUM_EPOCHS,
  # logger=pl.loggers.WandbLogger(),
  callbacks=[feedforward_nn_dropout_checkpoint],
  accelerator="gpu",
  devices=1,
)

# Initialize wandb to log the losses from each step.
# wandb.init(
#     project='simpleffdropout',
#     config=config,
# )
# Fit model.
# trainer.fit(lightning_model, train_loader, val_loader)

# print(f"Best model is saved at {feedforward_nn_dropout_checkpoint.
 ↪best_model_path}")


# # Load best model and evaluate it.
# best_model_path = feedforward_nn_dropout_checkpoint.best_model_path
# # best_model_path = ... # Insert if you have already trained this model.
# best_model = LightningModel.load_from_checkpoint(best_model_path,␣
 ↪model=SimpleFeedforwardNNWDropout())
```

```
# # save model to local
# torch.save(best_model, 'simpleffdropout_model.pth')

# wandb.finish()
```

[30]:
```
# model_nn_with_dropout = torch.load('/kaggle/input/
↪hw1-2-neural-network-tokenization-to-student-2024/pytorch/default/2/
↪simpleffdropout_model.pth', weights_only=False)
# result = evaluate(test_loader, model_nn_with_dropout.cuda())
# print(result)
```

SimpleFeedforwardNNWDropout
'accuracy': 0.8243543380699775,
'f1_score': 0.6132824499977226,
'precision': 0.7933249427114412,
'recall': 0.499844402846559 07

5 Convolution Neural Networks

Now, you are going to implement you own 1d-convolution neural networks with the following structure: input -> embedding layer (size 32) -> 1D-convolution layer (100 filters of size 5, strides of 1) -> Dense size 5 (applied across time dimension) -> fully-connected layer (size 100) -> output.

These parameters are simple guidelines to save your time. You can play with them in the final section.

The results should be better than the feedforward model.

Embedding layers turn the input from a one-hot vector into better representations via some feature transform (a simple matrix multiply in this case).

Note you need to flatten the tensor before the final fully connected layer because of dimension mis-match. The tensor could be reshaped using the `view` method.

Do consult PyTorch documentation on how to use embedding layers and 1D-cnn.

Hint: to apply dense5 across the time dimension you should read about how the multiplication in the dense layer is applied. The output of the 1D-cnn should be [batch x nfilter x sequence length]. We want to apply the dense5 (a weight matrix of size 100 x 5) by multiplying the same set of numbers over the nfilter dimension repeated over the sequence length (this can be possible via broadcasting) which should give an output of [batch x 5 x sequence length]. You might want to use the function transpose somehow.

Even more hints: https://stackoverflow.com/questions/58587057/multi-dimensional-inputs-in-pytorch-linear-method

[31]:
```
################################################################################
# TODO#5:                                                                      #
# Write a function that returns convolution nueral network model.            #
# You can choose any normalization methods, activation function, as well as   #
# any hyperparameter the way you want. Your goal is to predict a score        #
# between [0,1] for each input whether it is the beginning of the word or not. #
#                                                                             #
# Hint: You should read PyTorch documentation to see the list of available    #
# layers and options you can use.                                             #
###############################################################################
import torch
import torch.nn as nn

class SimpleCNN(nn.Module):
    def __init__(self, vocab_size=len(CHARS), seq_length=21):
        super(SimpleCNN, self).__init__()

        # Embedding layer
        self.embedding = nn.Embedding(num_embeddings=vocab_size,␣
 ↪embedding_dim=32)

        # 1D convolution
        self.conv = nn.Conv1d(in_channels=32, out_channels=100, kernel_size=5,␣
 ↪stride=1)

        # Dense layer
        self.dense = nn.Linear(in_features=100, out_features=5)

        # Fully connected layer
        conv_output_length = seq_length - 5 + 1 # kernel_size=5, stride=1, no␣
 ↪padding
        self.fc = nn.Linear(in_features=5 * conv_output_length,␣
 ↪out_features=100)

        # Final output layer
        self.out = nn.Linear(in_features=100, out_features=1)

        # Activation function
        self.relu = nn.ReLU()

    def forward(self, x):
        # Embedding: [batch_size, seq_length] -> [batch_size, seq_length, 32]
        x = self.embedding(x.long())
        x = x.transpose(1, 2)

        # 1D Convolution: [batch_size, 32, seq_length] -> [batch_size, 100,␣
 ↪new_seq_length]
        x = self.conv(x)
        x = self.relu(x)
        x = x.transpose(1, 2)
        x = self.dense(x)
        x = self.relu(x)
        x = x.transpose(1, 2)
        x = x.contiguous().view(x.size(0), -1)
        x = self.fc(x)
        x = self.relu(x)
        x = self.out(x)
        x = torch.sigmoid(x)

        return x
```

[35]:
```
# Debug
# Dummy input: Batch size 64, sequence length 21
model = SimpleCNN().cuda()
dummy_input = torch.randint(0, len(CHARS), (64, 21)).cuda()
output = model(dummy_input)
print(output.shape)  # Expected: (64, 1)
```

```
torch.Size([64, 1])
```

[36]:
```
################################################################################
# TODO#6:                                                                      #
# Write code that performs a training process. Select your batch size carefully#
# as it will affect your model's ability to converge and                       #
# time needed for one epoch.                                                   #
################################################################################
model_conv1d_nn = SimpleCNN().cuda()
from torchinfo import summary
summary(model_conv1d_nn,  input_size=(64, 21), device='cuda') #summarize the␣
 ↪model
################################################################################
#                            WRITE YOUR CODE BELOW                             #
################################################################################


# Hyperparameter initialization
import torch.optim as optim
NUM_EPOCHS = 3
criterion = torch.nn.BCELoss(reduction = 'none')
BATCHS_SIZE = 512
optimizer_class = optim.Adam
optimizer_params = {'lr': 5e-4}

config = {
    'architecture': 'simplecnn',
    'epochs': NUM_EPOCHS,
    'batch_size': BATCHS_SIZE,
    'optimizer_params': optimizer_params,
}

# DataLoader
from torch.utils.data import DataLoader
train_loader = DataLoader( Dataset(x_train_char, y_train, dtype = 'float'),␣
 ↪batch_size = BATCHS_SIZE)
val_loader = DataLoader( Dataset(x_val_char, y_val, dtype = 'float'),␣
 ↪batch_size = BATCHS_SIZE)
test_loader = DataLoader( Dataset(x_test_char, y_test, dtype = 'float'),␣
 ↪batch_size = BATCHS_SIZE)

# Initialize LightningModel.
lightning_model = LightningModel(
  model_conv1d_nn,
  criterion,
  optimizer_class,
  optimizer_params,
)
# Define checkpoint.
simplecnn_checkpoint = ModelCheckpoint(
  monitor="val_accuracy",
  mode="max",
  save_top_k=1,
  dirpath="./checkpoints",
  filename='simplecnn_checkpoint'
)
# Initialize Trainer
trainer = pl.Trainer(
  max_epochs=NUM_EPOCHS,
  logger=pl.loggers.WandbLogger(),
  callbacks=[simplecnn_checkpoint],
  accelerator="gpu",
  devices=1,
)

# # Initialize wandb to log the losses from each step.
# with wandb.init(project='simplecnn',config=config):
#     # Fit model.
#     trainer.fit(lightning_model, train_loader, val_loader)

#     print(f"Best model is saved at {simplecnn_checkpoint.best_model_path}")


#     # Load best model and evaluate it.
#     best_model_path = simplecnn_checkpoint.best_model_path
#     # best_model_path = ... # Insert if you have already trained this model.
#     best_model = LightningModel.load_from_checkpoint(best_model_path,␣
↪model=SimpleCNN())

#     # save model to local
#     torch.save(best_model, 'simplecnn_model.pth')

#     # wandb.finish()
```

[37]:
```
# model_conv1d_nn = torch.load('/kaggle/input/
↪hw1-2-neural-network-tokenization-to-student-2024/pytorch/default/3/
↪simplecnn_batch512_model.pth', weights_only=False)
# result = evaluate(test_loader, model_conv1d_nn.cuda())
# print(result)
```

SimpleCNN:
{'accuracy': 0.9687618291392525, 'f1_score': 0.9448753090380073, 'precision': 0.9294457059555208, 'recall': 0.9608258496631361}

6 Final Section

7 PyTorch playground

Now, train the best model you can do for this task. You can use any model structure and function available. Remember that training time increases with the complexity of the model. You might find wandb helpful in tuning of complicated models.

Your model should be better than your CNN or GRU model in the previous sections.

Some ideas to try 1. Tune the parameters 2. Recurrent models 3. CNN-GRU model 4. Improve the learning rate scheduling

[38]:
```
# class LightningModel(pl.LightningModule):
#     def __init__(
#         self,
#         model=SimpleFeedforwardNN(),
#         criterion=criterion,
#         optimizer_class=optim.Adam,
#         optimizer_params={'lr': 5e-4}
#     ):
#         super().__init__()
#         self.model = model
#         self.criterion = criterion
#         self.optimizer_class = optimizer_class
#         self.optimizer_params = optimizer_params

#         # Metrics for monitoring
#         self.train_accuracy = Accuracy(task="binary")
#         self.val_accuracy = Accuracy(task="binary")
#         self.val_f1 = F1Score(task="binary")

#     def forward(self, x):
#         return self.model(x)

#     def training_step(self, batch, batch_idx):
#         x, y = batch
#         y = y.float().view(-1, 1)  # Flatten target tensor to match model␣
 ↪output
#         y_hat = self(x)
#         loss = self.criterion(y_hat, y)

#         # Log training metrics
#         preds = (y_hat > 0.5).float()
#         self.train_accuracy.update(preds, y)
#         self.log("train_loss", loss, prog_bar=True, on_epoch=True)
#         self.log("train_acc", self.train_accuracy, prog_bar=True,␣
 ↪on_epoch=True)

#         return loss

#     def validation_step(self, batch, batch_idx):
#         x, y = batch
#         y = y.float().view(-1, 1)  # Flatten target tensor to match model␣
 ↪output
#         y_hat = self(x)
#         loss = self.criterion(y_hat, y)

#         # Compute validation metrics
#         preds = (y_hat > 0.5).float()
#         self.val_accuracy.update(preds, y)
#         self.val_f1.update(preds, y)

#         self.log("val_loss", loss, prog_bar=True, on_epoch=True)
#         return loss

#     def on_validation_epoch_end(self):
#         # Log validation metrics at the end of each epoch
#         self.log("val_acc", self.val_accuracy.compute(), prog_bar=True)
#         self.log("val_f1", self.val_f1.compute(), prog_bar=True)

#         # Reset metrics
#         self.val_accuracy.reset()
#         self.val_f1.reset()
#     def configure_optimizers(self):
#         optimizer = self.optimizer_class(self.model.parameters(), **self.
 ↪optimizer_params)
#         scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,␣
 ↪mode="max", factor=0.5, patience=2, verbose=True)

#         return {
#             "optimizer": optimizer,
#             "lr_scheduler": {
#                 "scheduler": scheduler,
#                 "monitor": "val_f1",
#             },
#         }
```

[39]:
```
# class LightningModel(pl.LightningModule):
#     def __init__(
#         self,
#         model=SimpleFeedforwardNN(),
#         criterion=criterion,
#         optimizer_class=optim.Adam,
#         optimizer_params={'lr': 5e-4}
#     ):
#         super().__init__()
#         self.model = model
#         self.criterion = criterion
#         self.optimizer_class = optimizer_class
#         self.optimizer_params = optimizer_params

#         # Metrics for monitoring
#         self.train_accuracy = Accuracy(task="binary")
#         self.train_f1 = F1Score(task="binary")
#         self.val_accuracy = Accuracy(task="binary")
#         self.val_f1 = F1Score(task="binary")

#     def forward(self, x):
#         return self.model(x)

#     def training_step(self, batch, batch_idx):
#         x, y = batch
#         y = y.float().view(-1, 1)  # Flatten target tensor to match model␣
 ↪output
#         y_hat = self(x)
#         loss = self.criterion(y_hat, y)

#         # Predictions and metrics
#         preds = (y_hat > 0.5).float()
#         self.train_accuracy.update(preds, y)
#         self.train_f1.update(preds, y)

#         # Log training metrics
#         self.log("train_loss", loss, prog_bar=True, on_epoch=True)
#         self.log("train_acc", self.train_accuracy, prog_bar=True,␣
 ↪on_epoch=True)
#         self.log("train_f1", self.train_f1, prog_bar=True, on_epoch=True)

#         return loss

#     def validation_step(self, batch, batch_idx):
#         x, y = batch
#         y = y.float().view(-1, 1)  # Flatten target tensor to match model␣
 ↪output
#         y_hat = self(x)
#         loss = self.criterion(y_hat, y)

#         # Predictions and metrics
#         preds = (y_hat > 0.5).float()
#         self.val_accuracy.update(preds, y)
#         self.val_f1.update(preds, y)

#         # Log validation loss
#         self.log("val_loss", loss, prog_bar=True, on_epoch=True)

#         return loss

#     def on_validation_epoch_end(self):
#         # Log validation metrics at the end of each epoch
#         val_acc = self.val_accuracy.compute()
#         val_f1 = self.val_f1.compute()
#         self.log("val_acc", val_acc, prog_bar=True)
#         self.log("val_f1", val_f1, prog_bar=True)

#         # Reset metrics
#         self.val_accuracy.reset()
#         self.val_f1.reset()

#     def test_step(self, batch, batch_idx):
#         x, y = batch
#         y = y.float().view(-1, 1)  # Flatten target tensor to match model␣
 ↪output
#         y_hat = self(x)
#         loss = self.criterion(y_hat, y)

#         # Predictions and metrics
#         preds = (y_hat > 0.5).float()
#         self.log("test_loss", loss, prog_bar=True)
#         self.log("test_acc", Accuracy(task="binary")(preds, y), prog_bar=True)
#         self.log("test_f1", F1Score(task="binary")(preds, y), prog_bar=True)

#         return loss

#     def configure_optimizers(self):
#         optimizer = self.optimizer_class(self.model.parameters(), **self.
 ↪optimizer_params)
#         scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,␣
 ↪mode="max", factor=0.5, patience=2, verbose=True)

#         return {
#             "optimizer": optimizer,
#             "lr_scheduler": {
#                 "scheduler": scheduler,
#                 "monitor": "val_f1",
#             },
#         }
```

[45]:
```
import torch
from torch.optim.lr_scheduler import CosineAnnealingLR
from torchmetrics.classification import Accuracy, F1Score


class LightningModel(pl.LightningModule):
    def __init__(
        self,
        model=SimpleFeedforwardNN(),
        criterion=criterion,
        optimizer_class=optim.Adam,
        optimizer_params={'lr': 5e-4},
        t_max=10
    ):
        super().__init__()
        self.model = model
        self.criterion = criterion
        self.optimizer_class = optimizer_class
        self.optimizer_params = optimizer_params
        self.t_max = t_max

        self.train_accuracy = Accuracy(task="binary")
        self.train_f1 = F1Score(task="binary")
        self.val_accuracy = Accuracy(task="binary")
        self.val_f1 = F1Score(task="binary")

    def forward(self, x):
        return self.model(x)

    def training_step(self, batch, batch_idx):
        x, y = batch
        y = y.float().view(-1, 1)
        y_hat = self(x)
        loss = self.criterion(y_hat, y)

        preds = (y_hat > 0.5).float()
        self.train_accuracy.update(preds, y)
        self.train_f1.update(preds, y)

        self.log("train_loss", loss, prog_bar=True, on_epoch=True)
        self.log("train_acc", self.train_accuracy, prog_bar=True, on_epoch=True)
        self.log("train_f1", self.train_f1, prog_bar=True, on_epoch=True)

        return loss

    def validation_step(self, batch, batch_idx):
        x, y = batch
        y = y.float().view(-1, 1)
        y_hat = self(x)
        loss = self.criterion(y_hat, y)

        preds = (y_hat > 0.5).float()
        self.val_accuracy.update(preds, y)
        self.val_f1.update(preds, y)

        self.log("val_loss", loss, prog_bar=True, on_epoch=True)

        return loss

    def on_validation_epoch_end(self):
        val_acc = self.val_accuracy.compute()
        val_f1 = self.val_f1.compute()
        self.log("val_acc", val_acc, prog_bar=True)
        self.log("val_f1", val_f1, prog_bar=True)

        self.val_accuracy.reset()
        self.val_f1.reset()

    def test_step(self, batch, batch_idx):
        x, y = batch
        y = y.float().view(-1, 1)
        y_hat = self(x)
        loss = self.criterion(y_hat, y)
        preds = (y_hat > 0.5).float()
        self.log("test_loss", loss, prog_bar=True)
        self.log("test_acc", Accuracy(task="binary")(preds, y), prog_bar=True)
        self.log("test_f1", F1Score(task="binary")(preds, y), prog_bar=True)

        return loss

    def configure_optimizers(self):
        optimizer = self.optimizer_class(self.model.parameters(), **self.
 ↪optimizer_params)
        scheduler = CosineAnnealingLR(optimizer, T_max=self.t_max)

        return {
            "optimizer": optimizer,
            "lr_scheduler": {
                "scheduler": scheduler,
                "interval": "epoch",
                "frequency": 1,
            },
        }
```

[46]:
```
#␣
 ↪################################################################################
# # TODO#7                                                                    ␣
 ↪ #
# # Write a class that returns your best model. You can use anything          ␣
 ↪ #
# # you want. The goal here is to create the best model you can think of.     ␣
 ↪ #
# # Your model should get f-score more than 97% from calling evaluate().      ␣
 ↪ #
# #                                                                           ␣
 ↪ #
# # Hint: You should read PyTorch documentation to see the list of available  ␣
 ↪ #
# # layers and options you can use.                                           ␣
 ↪ #
#␣
 ↪################################################################################

# import torch
# import torch.nn as nn

# class BestModel(nn.Module):
#     def __init__(self, vocab_size=len(CHARS), embed_dim=64, hidden_size=256,␣
 ↪num_layers=2, dropout_rate=0.3):
```

```
#         """
#         Args:
#           vocab_size (int): Size of the vocabulary.
#           embed_dim (int): Dimension of embedding vectors.
#           hidden_size (int): Number of features in the LSTM hidden state.
#           num_layers (int): Number of recurrent layers.
#           dropout_rate (float): Dropout probability for regularization.
#         """
#         super(BestModel, self).__init__()

#         # Embedding layer to convert token indices into dense vectors
#         self.embedding = nn.Embedding(num_embeddings=vocab_size,␣
↪embedding_dim=embed_dim)

#         # Bidirectional LSTM: uses two directions to capture context from␣
↪past and future
#         self.lstm = nn.LSTM(
#             input_size=embed_dim,
#             hidden_size=hidden_size,
#             num_layers=num_layers,
#             batch_first=True,
#             bidirectional=True,
#             dropout=dropout_rate
#         )

#         # Dropout layer for regularization
#         self.dropout = nn.Dropout(dropout_rate)

#         # Fully connected layer to map concatenated hidden states to a single␣
↪output score
#         # Multiplying hidden_size by 2 due to bidirectionality
#         self.fc = nn.Linear(in_features=hidden_size * 2, out_features=1)

#     def forward(self, x):
#         """
#         Forward pass for the model.

#         Args:
#           x (Tensor): Input tensor of shape [batch_size, sequence_length]␣
↪containing token indices.

#         Returns:
#           Tensor: Output scores of shape [batch_size, 1] between 0 and 1.
#         """
#         # Convert token indices to embeddings: [batch_size, sequence_length]␣
↪-> [batch_size, sequence_length, embed_dim]
#         embedded = self.embedding(x.long())
#         # Pass embeddings through the LSTM
#         # lstm_out: [batch_size, sequence_length, hidden_size*2] (for␣
↪bidirectional)
#         # h_n: [num_layers*2, batch_size, hidden_size] contains the final␣
↪hidden states for each layer and direction
#         lstm_out, (h_n, c_n) = self.lstm(embedded)

#         # Extract the last layer's hidden states for both directions
#         # Since it's bidirectional, h_n has 2*num_layers layers stacked.
#         # The last two layers correspond to the forward and backward␣
↪directions of the final LSTM layer.
#         h_forward = h_n[-2, :, :]  # Last layer forward hidden state:␣
↪[batch_size, hidden_size]
#         h_backward = h_n[-1, :, :] # Last layer backward hidden state:␣
↪[batch_size, hidden_size]

#         # Concatenate forward and backward final hidden states: [batch_size,␣
↪hidden_size*2]
#         h_combined = torch.cat((h_forward, h_backward), dim=1)

#         # Apply dropout for regularization
#         h_dropped = self.dropout(h_combined)

#         # Fully connected layer maps the combined hidden state to a single␣
↪output
#         logits = self.fc(h_dropped)

#         # Use sigmoid activation to produce an output score between 0 and 1
#         output = torch.sigmoid(logits)

#         return output
```

[47]:
```
import torch
import torch.nn as nn

class Attention(nn.Module):
    def __init__(self, hidden_size):
        super(Attention, self).__init__()
        self.attention = nn.Linear(hidden_size * 2, hidden_size * 2)
        self.context_vector = nn.Linear(hidden_size * 2, 1, bias=False)

    def forward(self, lstm_output):
        scores = torch.tanh(self.attention(lstm_output))  # [batch_size,␣
↪seq_length, hidden_size*2]
```

```
        scores = self.context_vector(scores).squeeze(-1)  # [batch_size,␣
↪seq_length]
        attention_weights = torch.softmax(scores, dim=1).unsqueeze(-1)  #␣
↪[batch_size, seq_length, 1]
        context = torch.sum(lstm_output * attention_weights, dim=1)  #␣
↪[batch_size, hidden_size*2]
        return context

class BestModel(nn.Module):
    def __init__(self, vocab_size=len(CHARS), embed_dim=128, hidden_size=256,␣
↪num_layers=2, dropout_rate=0.3):
        super(BestModel, self).__init__()

        self.embedding = nn.Embedding(num_embeddings=vocab_size,␣
↪embedding_dim=embed_dim)

        # Bidirectional LSTM
        self.lstm = nn.LSTM(
            input_size=embed_dim,
            hidden_size=hidden_size,
            num_layers=num_layers,
            batch_first=True,
            bidirectional=True,
            dropout=dropout_rate
        )

        self.attention = Attention(hidden_size)
        self.layer_norm = nn.LayerNorm(hidden_size * 2)
        self.dropout = nn.Dropout(dropout_rate)

        self.fc1 = nn.Linear(in_features=hidden_size * 2, out_features=128)
        self.activation = nn.GELU()
        self.fc2 = nn.Linear(in_features=128, out_features=1)

    def forward(self, x):
        embedded = self.embedding(x.long())  # [batch_size, seq_length,␣
↪embed_dim]

        lstm_out, _ = self.lstm(embedded)  # [batch_size, seq_length,␣
↪hidden_size*2]

        context = self.attention(lstm_out)  # [batch_size, hidden_size*2]
        context = self.layer_norm(context)
        context = self.dropout(context)

        hidden = self.fc1(context)  # [batch_size, 128]
        hidden = self.activation(hidden)
        hidden = self.dropout(hidden)

        logits = self.fc2(hidden)  # [batch_size, 1]
        output = torch.sigmoid(logits)

        return output
```

[48]:
```
################################################################################
# TODO#8                                                                      #
# Write code that perform a trainin loop on this dataset. Select your          #
# batch size carefully as it will affect your model's ability to converge and  #
# time needed for one epoch.                                                   #
#                                                                              #
################################################################################
print('start training')
my_best_model = BestModel()
################################################################################
#                             WRITE YOUR CODE BELOW                            #
################################################################################




# Hyperparameter initialization
import torch.optim as optim
NUM_EPOCHS = 3
# criterion = torch.nn.BCELoss(reduction = 'none')
criterion = torch.nn.BCELoss()
BATCHS_SIZE = 512
optimizer_class = optim.Adam
optimizer_params = {'lr': 5e-4}

config = {
    'architecture': 'bestmodel',
    'epochs': NUM_EPOCHS,
    'batch_size': BATCHS_SIZE,
    'optimizer_params': optimizer_params,
}

# DataLoader
from torch.utils.data import DataLoader
train_loader = DataLoader( Dataset(x_train_char, y_train, dtype = 'float'),␣
  ↪batch_size = BATCHS_SIZE)
val_loader = DataLoader( Dataset(x_val_char, y_val, dtype = 'float'),␣
  ↪batch_size = BATCHS_SIZE)
test_loader = DataLoader( Dataset(x_test_char, y_test, dtype = 'float'),␣
  ↪batch_size = BATCHS_SIZE)
```

```python
# Initialize LightningModel.
lightning_model = LightningModel(
  my_best_model,
  criterion,
  optimizer_class,
  optimizer_params,
)
# Define checkpoint.
bestmodel_checkpoint = ModelCheckpoint(
  monitor="val_f1",
  mode="max",
  save_top_k=1,
  dirpath="./checkpoints",
  filename='bestmodel_checkpoint'
)
# Initialize Trainer
trainer = pl.Trainer(
  max_epochs=NUM_EPOCHS,
  logger=pl.loggers.WandbLogger(),
  callbacks=[bestmodel_checkpoint],
  accelerator="gpu",
  devices=1,
)

# Initialize wandb to log the losses from each step.
wandb.finish()
with wandb.init(project='bestmodel',config=config):
    # Fit model.
    trainer.fit(lightning_model, train_loader, val_loader)

    print(f"Best model is saved at {bestmodel_checkpoint.best_model_path}")

    # save model to local
    torch.save(my_best_model, 'bestmodel_model.pth')
```

```
start training

wandb: Using wandb-core as the SDK backend.  Please refer to
https://wandb.me/wandb-core for more information.
wandb: Currently logged in as: pupipat-sk
(pupipatsk). Use `wandb login --relogin` to force relogin

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>

/usr/local/lib/python3.10/dist-packages/pytorch_lightning/loggers/wandb.py:396:
There is a wandb run already in progress and newly created instances of
`WandbLogger` will reuse this run. If this is not desired, call `wandb.finish()`
before instantiating `WandbLogger`.

Sanity Checking: |          | 0/? [00:00<?, ?it/s]

/usr/local/lib/python3.10/dist-
packages/pytorch_lightning/trainer/connectors/data_connector.py:424: The
'val_dataloader' does not have many workers which may be a bottleneck. Consider
increasing the value of the `num_workers` argument` to `num_workers=3` in the
`DataLoader` to improve performance.
/usr/local/lib/python3.10/dist-
packages/pytorch_lightning/trainer/connectors/data_connector.py:424: The
'train_dataloader' does not have many workers which may be a bottleneck.
Consider increasing the value of the `num_workers` argument` to `num_workers=3`
in the `DataLoader` to improve performance.

Training: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Best model is saved at /kaggle/working/checkpoints/bestmodel_checkpoint.ckpt

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

[49]:
```python
def evaluate(test_loader, model):
    model.eval()
    test_loss = []
    test_pred = []
    test_true = []

    with torch.no_grad():
        for batch in test_loader:
            X_test, Y_test = batch
            X_test, Y_test = X_test.cuda(), Y_test.cuda()

            # Forward pass
            Y_hat = model(X_test)
            loss = criterion(Y_hat, Y_test.view(-1, 1))
            # Collect test loss
            test_loss.append(loss.item())  # Store scalar loss

            # Collect predictions and true labels
            preds = (Y_hat > 0.5).float()
            test_pred.append(preds)
            test_true.append(Y_test)

    # Compute average loss
    avg_test_loss = sum(test_loss) / len(test_loss)

    # Concatenate predictions and true labels for metric calculation
    test_pred = torch.cat(test_pred, axis=0).cpu().detach().numpy()
    test_true = torch.cat(test_true, axis=0).cpu().detach().numpy()

    # Compute metrics (e.g., accuracy, F1 score)
    test_accuracy = Accuracy(task="binary")(torch.tensor(test_pred), torch.
 ↪tensor(test_true))
    test_f1 = F1Score(task="binary")(torch.tensor(test_pred), torch.
 ↪tensor(test_true))

    print(f"Test Loss: {avg_test_loss:.4f}")
    print(f"Test Accuracy: {test_accuracy:.4f}")
    print(f"Test F1 Score: {test_f1:.4f}")

    return avg_test_loss, test_accuracy, test_f1
```

[51]:
```
summary(my_best_model)
```

[51]:
```
=================================================================
Layer (type:depth-idx)                   Param #
=================================================================
BestModel                                --
 Embedding: 1-1                         22,784
 LSTM: 1-2                              2,367,488
 Attention: 1-3                         --
     Linear: 2-1                        262,656
     Linear: 2-2                        512
 LayerNorm: 1-4                         1,024
 Dropout: 1-5                           --
 Linear: 1-6                            65,664
 GELU: 1-7                              --
 Linear: 1-8                            129
=================================================================
Total params: 2,720,257
Trainable params: 2,720,257
Non-trainable params: 0
```

```
=================================================================
```

```
[50]: # Load best model and evaluate it.
      best_model_path = bestmodel_checkpoint.best_model_path
      # best_model_path = ... # Insert if you have already trained this model.
      best_model = LightningModel.load_from_checkpoint(best_model_path,␣
       ↪model=BestModel())

      evaluate(test_loader, my_best_model.cuda())
```

```
Test Loss: 0.0467
Test Accuracy: 0.9857
Test F1 Score: 0.9745
```

```
[50]: (0.046656914323302 04, tensor(0.9857), tensor(0.9745))
```

Simple-LSTM-maxacc:
{'accuracy': 0.9784161673852915, 'f1_score': 0.961787554168131, 'precision': 0.9490793998412864, 'recall': 0.9748406511385447}

LSTM-sch-maxf1:
'val_accuracy': 0.97921,
'val_f1_score': 0.96257,

LSTM-ATT-maxf1-1epoch:
'accuracy': 0.9813,
'f1_score': 0.9669,

LSTM-ATT-Cos-maxf1-3epoch(1.5hr):
'accuracy': 0.9857,
'f1_score': 0.9745,

```
[ ]:
```