NLP Final Project Part 1 to 3

December 13, 2024

1 50.040 Natural Language Processing (Fall 2024) Final Project (100 Points)

DUE DATE: 13 December 2024

Final project will be graded by Chen Huang

2 Group Information (Fill before you start)

Group Name:

ChatGPT Course

Name(STUDNET ID) (2-3 person):

Beckham Wee Yu Zheng(1006010)

Deshpande Sunny Nitin(1006336)

Patrick Mo(1006084)

Please also rename the final submitted pdf as `finalproject_[GROUP_NAME].pdf`

-1 points if info not filled or file name not adjusted before submission, -100 points if you copy other's answer. We encourage discussion, but please do not copy without thinking.

2.1 [!] Please read this if your computer does not have GPUs.

2.1.1 Free GPU Resources

We suggest that you run neural language models on machines with GPU(s). Google provides the free online platform Colaboratory, a research tool for machine learning education and research. It's a Jupyter notebook environment that requires no setup to use as common packages have been pre-installed. Google users can have access to a Tesla T4 GPU (approximately 15G memory). Note that when you connect to a GPU-based VM runtime, you are given a maximum of 12 hours at a time on the VM.

Colab is web-based, fast and convinient. You can simply upload this notebook and run it online. For the database needed in this task, you can download it and upload to colab OR you can save it in your google drive and link it with the colab.

It is convenient to upload local Jupyter Notebook files and data to Colab, please refer to the tutorial.

In addition, Microsoft also provides the online platform Azure Notebooks for research of data science and machine learning, there are free trials for new users with credits.

3 Instructions

Please read the following instructions carefully before you begin this project:

- This is a group project. You are allowed to form groups in any way you like, but each group must consist of either 2 or 3 people. Please submit your group information to eDimension as soon as possible if you have not yet done so (the deadline was 11th October 2024).
- Each group should submit code along with a report summarizing your work, and provide clear instructions on how to run your code. Additionally, please submit your system's outputs. The output should be in the same column format as the training set.
- You are given **8** weeks to work on the project. We understand this is a busy time during your final term, so Week **13** will be reserved as the "Final Project Week" for you to focus on the project (i.e., there will be no classes that week). Please plan and manage your time well.
- Please use Python to complete this project.

4 Project Summary

Welcome to the design project for our natural language processing (NLP) course offered at SUTD!

In this project, you will undertake an NLP task in *sentiment analysis*. We will begin by guiding you through data understanding, pre-processing, and instructing you to construct RNN and CNN models for the task. Afterward, you are encouraged to develop your own model to improve the results. The final test set will be released on **11 December 2024 at 5pm** (48 hours before the final project deadline). You will use your own system to generate the outputs. The system with the highest F1 score will be announced as the winner for each challenge. If no clear winner emerges from the test set results, further analysis or evaluations may be conducted.

5 Task Introduction and Data pre-processing (20 points)

5.1 Sentiment Analysis

With the proliferation of online social media and review platforms, vast amounts of opinionated data have been generated, offering immense potential to support decision-making processes. Sentiment analysis examines people's sentiments expressed in text, such as product reviews, blog comments, and forum discussions. It finds wide applications in diverse fields, including politics (e.g., analyzing public sentiment towards policies), finance (e.g., evaluating market sentiments), and marketing (e.g., product research and brand management).

Since sentiments can often be categorized into discrete polarities or scales (e.g., positive or negative), sentiment analysis can be framed as a text classification task. This task involves transforming text sequences of varying lengths into fixed-length categorical labels.

5.2 Data pre-processing

In this project, we will utilize Stanford's large movie review dataset for sentiment analysis. The dataset consists of a training set and a testing set, each containing 25,000 movie reviews sourced from IMDb. Both datasets have an equal number of "positive" and "negative" labels, representing different sentiment polarities. Please download and extract this IMDb review dataset in the path `../data/aclImdb`.

Hints: While the following instructions are based on a split of 25,000 for training and 25,000 for testing, you are free to choose your own dataset split, as we will provide a separate test set for the final evaluation. However, any changes you make to the default split must be clearly indicated in your report. Failure to explicitly mention such modifications may result in a penalty.

[26]:
```
print('sa oot sopo')
import os
import torch
from torch import nn
from d2l import torch as d2l # You can skip this if you have trouble with this␣
 ↪package, all d2l-related codes can be replaced by torch functions.
```

```
sa oot sopo
```

[27]:
```
import pandas as pd
import zipfile
```

[28]:
```
# Skip this if you have already downloaded the dataset
d2l.DATA_HUB['aclImdb'] = (d2l.DATA_URL + 'aclImdb_v1.tar.gz',
                           '01ada507287d82875905620988597833ad4e0903')

data_dir = d2l.download_extract('aclImdb', 'aclImdb')
```

[29]:
```
data_dir = 'data/aclImdb'
```

6 Questions

6.0.1 Question 1 [code] (5 points)

Complete the function `read_imdb`, which reads the IMDb review dataset text sequences and labels. Then, run the sanity check cell to check your implementation.

[30]:
```
#@save
def read_imdb(data_dir, is_train):
    """Read the IMDb review dataset text sequences and labels."""
    ### YOUR CODE HERE
    data, labels = [], []

    for label in ('pos', 'neg'): # label is either 'pos' or 'neg'
```

```
        folder_name = os.path.join(data_dir, 'train' if is_train else 'test',␣
↪label) # folder_name is either 'train/pos' or 'train/neg' or 'test/pos' or␣
↪'test/neg'

        for file in os.listdir(folder_name):
            with open(os.path.join(folder_name, file), 'rb') as f:
                review = f.read().decode('utf-8').replace('\n', '')
                data.append(review)
                labels.append(1 if label == 'pos' else 0)

    ### END OF YOUR CODE
    return data, labels
```

[31]:
```
# Sanity check
train_data = read_imdb(data_dir, is_train=True)
print('# trainings:', len(train_data[0]))
for x, y in zip(train_data[0][:3], train_data[1][:3]):
    print('label:', y, 'review:', x[:60])
```

```
# trainings: 25000
label: 1 review: Bromwell High is a cartoon comedy. It ran at the same time a
label: 1 review: Homelessness (or Houselessness as George Carlin stated) has
label: 1 review: Brilliant over-acting by Lesley Ann Warren. Best dramatic ho
```

6.0.2 Question 2 [code] (5 points)

Treating each word as a token and filtering out words that appear less than 5 times, we create a vocabulary out of the training dataset. After tokenization, please plot the histogram of review lengths in tokens. (Hint: you can use `matplotlib` package to draw the histogram.) Then, run the sanity check cell to check your implementation.

[32]:
```
train_tokens = d2l.tokenize(train_data[0], token='word')
# Tokenize the training reviews, return a list of token lists
# Each token list corresponds to a review

vocab = d2l.Vocab(train_tokens, min_freq=5, reserved_tokens=['<pad>'])
 # Create a vocabulary for the tokens, only include the tokens that appear at␣
↪least 5 times in the training reviews
 # Pad token '<pad>' is used to pad the sequence to a fixed length
 # Padding is done to make the sequence length the same for all reviews, it␣
↪helps with the batching of reviews
```

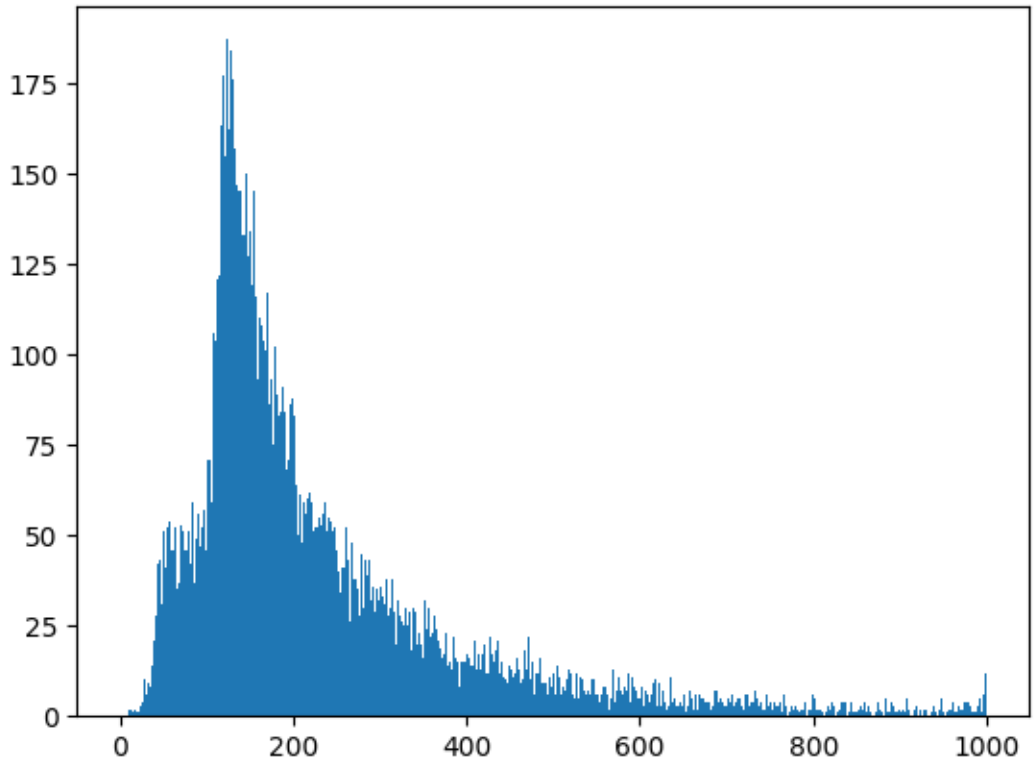
[33]:
```
# train_tokens : list of list of tokens
# vocab : Vocab object
import matplotlib.pyplot as plt
plt.hist([len(line) for line in train_tokens], bins=range(0, 1000, 1)) # Plot␣
↪the histogram of the number of tokens in the reviews
```

```
[33]: (array([  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   2.,
                1.,   2.,   0.,   1.,   1.,   0.,   2.,   2.,   1.,   1.,   1.,
                2.,   3.,   3.,   4.,   4.,   5.,  10.,   4.,   6.,   5.,   9.,
               11.,  17.,   8.,  19.,  14.,  18.,  21.,  22.,  28.,  42.,  42.,
               43.,  28.,  43.,  43.,  31.,  54.,  51.,  46.,  41.,  47.,  48.,
               52.,  49.,  54.,  49.,  46.,  62.,  46.,  37.,  52.,  43.,  34.,
               35.,  49.,  37.,  54.,  53.,  54.,  51.,  41.,  37.,  46.,  58.,
               46.,  49.,  51.,  39.,  42.,  56.,  59.,  46.,  42.,  37.,  47.,
               49.,  47.,  56.,  42.,  47.,  36.,  52.,  59.,  52.,  57.,  44.,
               46.,  56.,  71.,  64.,  71.,  68.,  65.,  59.,  93., 106.,  95.,
              104., 110., 121., 120., 122., 132., 116., 163., 148., 177., 155.,
              155., 173., 187., 181., 162., 170., 185., 184., 182., 176., 167.,
              157., 187., 147., 158., 155., 145., 172., 145., 163., 133., 153.,
              133., 152., 150., 152., 125., 127., 128., 134., 141., 119., 113.,
              145., 114., 116., 124., 115.,  93., 107., 110.,  99., 108., 107.,
              104., 110.,  95., 101., 129., 117., 109.,  86.,  98.,  93.,  66.,
               75., 107.,  95., 102.,  88.,  89.,  83.,  83.,  91.,  84.,  74.,
               98.,  91.,  71.,  84.,  72.,  68.,  71.,  71.,  65.,  86.,  73.,
               72.,  88.,  68.,  83.,  63.,  64.,  73.,  50.,  67.,  61.,  58.,
               63.,  48.,  64.,  59.,  69.,  56.,  71.,  60.,  62.,  61.,  62.,
               62.,  59.,  47.,  51.,  40.,  52.,  59.,  52.,  61.,  56.,  55.,
               48.,  53.,  55.,  56.,  53.,  59.,  43.,  51.,  51.,  36.,  55.,
               46.,  54.,  48.,  51.,  59.,  52.,  46.,  45.,  46.,  53.,  40.,
               52.,  34.,  45.,  41.,  35.,  41.,  39.,  36.,  52.,  43.,  43.,
               47.,  26.,  47.,  48.,  38.,  38.,  32.,  36.,  38.,  38.,  35.,
               34.,  28.,  31.,  45.,  27.,  38.,  30.,  30.,  43.,  27.,  39.,
               22.,  43.,  33.,  32.,  28.,  41.,  36.,  34.,  29.,  31.,  35.,
               31.,  32.,  26.,  36.,  34.,  25.,  33.,  23.,  31.,  42.,  38.,
               37.,  28.,  27.,  30.,  30.,  33.,  38.,  37.,  29.,  26.,  20.,
               15.,  32.,  29.,  23.,  28.,  28.,  26.,  26.,  25.,  31.,  30.,
               25.,  28.,  25.,  23.,  29.,  29.,  18.,  23.,  30.,  19.,  29.,
               31.,  32.,  20.,  27.,  23.,  24.,  20.,  19.,  16.,  19.,  32.,
               25.,  25.,  24.,  24.,  30.,  28.,  22.,  22.,  23.,  24.,  18.,
               28.,  21.,  24.,  31.,  21.,  16.,  19.,  27.,  16.,  25.,  23.,
               17.,  22.,  23.,  25.,  14.,  17.,  15.,  13.,  13.,  14.,  21.,
               22.,   9.,  16.,  19.,  15.,  19.,   8.,  19.,  22.,  15.,  18.,
               15.,  10.,  15.,  22.,  17.,  17.,  16.,  21.,  18.,  14.,  17.,
               14.,  16.,  21.,  25.,  13.,  21.,  17.,  16.,  11.,  13.,  11.,
               17.,  14.,  20.,  19.,  12.,  14.,  12.,  12.,  15.,  22.,  18.,
               17.,  13.,  15.,  13.,  18.,   8.,  17.,  21.,  16.,  12.,  18.,
               15.,  14.,  11.,  11.,   9.,  10.,  15.,   9.,  10.,  14.,  11.,
               13.,  11.,  11.,  14.,  15.,  12.,  14.,  16.,  14.,  13.,  18.,
                9.,   7.,  10.,  13.,  12.,  18.,  15.,  13.,  12.,  22.,  11.,
               10.,  12.,   9.,  15.,  12.,   6.,  11.,  12.,  14.,  12.,   7.,
               16.,  14.,  19.,   9.,   4.,   9.,  11.,   9.,   8.,   6.,   8.,
               11.,   9.,   9.,   8.,   7.,  12.,  12.,   6.,  10.,  14.,  10.,
               15.,  11.,   6.,   6.,  12.,   8.,   5.,   7.,   7.,  10.,   9.,
```

```
       10.,  13.,   6.,  12.,   6.,   6.,  10.,   5.,  15.,  12.,  12.,
        9.,   5.,   8.,  11.,   8.,  10.,  12.,   7.,   4.,   7.,   6.,
        6.,   7.,   7.,   6.,   8.,  10.,   9.,  10.,   4.,  10.,   6.,
        5.,   6.,  12.,   4.,  10.,   6.,   9.,   8.,   5.,  12.,   8.,
        9.,   6.,  10.,   2.,   8.,   5.,   6.,   5.,  13.,  13.,   4.,
        5.,   7.,   3.,  11.,   6.,   7.,   5.,   2.,   6.,   8.,   8.,
        6.,   7.,   6.,  12.,   8.,   9.,   4.,   7.,  11.,   5.,   8.,
        4.,   7.,   6.,   5.,   5.,   5.,   5.,   5.,   8.,   4.,   3.,
        5.,   7.,   8.,   6.,   4.,   4.,   4.,   7.,   6.,   3.,   9.,
        4.,  10.,   8.,   7.,   3.,   6.,   9.,   4.,   3.,   7.,   7.,
        3.,   4.,   5.,   3.,   2.,   7.,   3.,   7.,  11.,   6.,   4.,
        3.,   5.,   3.,   6.,   3.,   4.,   3.,   4.,   4.,   2.,   3.,
        6.,   5.,   6.,   6.,   1.,   3.,   3.,   3.,   7.,   4.,   5.,
        4.,   3.,   2.,   3.,   6.,   6.,   2.,   5.,   6.,   5.,   5.,
        6.,   4.,   5.,   8.,   4.,   2.,   4.,   1.,   4.,   5.,   4.,
        2.,   3.,   3.,   7.,   7.,   6.,   7.,   3.,   4.,   4.,   3.,
        5.,   1.,   4.,   4.,   3.,   1.,   3.,   6.,   5.,   5.,   5.,
        4.,   2.,   6.,   2.,   3.,   5.,   4.,   3.,   5.,   5.,   6.,
        5.,   4.,   3.,   0.,   2.,   2.,   6.,   3.,   2.,   6.,   1.,
        3.,   3.,   4.,   2.,   4.,   1.,   5.,   7.,   3.,   5.,   3.,
        3.,   1.,   4.,   2.,   2.,   0.,   2.,   2.,   3.,   6.,   3.,
        3.,   3.,   4.,   4.,   2.,   2.,   4.,   4.,   5.,   3.,   2.,
        3.,   2.,   4.,   2.,   2.,   1.,   6.,   6.,   2.,   3.,   1.,
        0.,   5.,   1.,   3.,   1.,   3.,   1.,   2.,   1.,   3.,   2.,
        2.,   1.,   1.,   2.,   6.,   2.,   4.,   2.,   4.,   4.,   4.,
        0.,   3.,   2.,   2.,   1.,   2.,   1.,   6.,   3.,   5.,   3.,
        2.,   4.,   3.,   2.,   3.,   2.,   1.,   1.,   5.,   0.,   7.,
        1.,   3.,   4.,   3.,   2.,   2.,   1.,   4.,   1.,   3.,   2.,
        7.,   0.,   3.,   1.,   1.,   2.,   1.,   4.,   2.,   4.,   1.,
        3.,   4.,   3.,   0.,   1.,   1.,   0.,   4.,   1.,   2.,   1.,
        0.,   1.,   2.,   1.,   1.,   2.,   2.,   3.,   2.,   3.,   2.,
        2.,   4.,   1.,   1.,   0.,   3.,   1.,   1.,   3.,   1.,   0.,
        2.,   0.,   2.,   1.,   2.,   4.,   0.,   1.,   2.,   4.,   1.,
        1.,   0.,   2.,   5.,   2.,   1.,   0.,   1.,   1.,   2.,   4.,
        2.,   3.,   1.,   1.,   1.,   1.,   0.,   3.,   2.,   3.,   1.,
        1.,   2.,   0.,   1.,   2.,   0.,   5.,   2.,   2.,   2.,   0.,
        2.,   0.,   0.,   2.,   5.,   2.,   3.,   1.,   0.,   0.,   2.,
        2.,   0.,   0.,   2.,   2.,   1.,   2.,   3.,   0.,   3.,   0.,
        0.,   1.,   2.,   1.,   3.,   1.,   1.,   1.,   0.,   2.,   2.,
        1.,   5.,   0.,   1.,   1.,   2.,   0.,   4.,   1.,   0.,   2.,
        2.,   0.,   2.,   2.,   4.,   3.,   1.,   4.,   1.,   3.,   3.,
        3.,   2.,   2.,   0.,   2.,   3.,   4.,   4.,   4.,   4.,   4.,
        2.,   1.,   3.,   2.,   1.,   3.,   1.,   2.,   1.,   3.,   6.,
        3.,   4.,   5.,   6.,   1.,   5.,   6.,   4.,  12.]),
array([  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.,
        11.,  12.,  13.,  14.,  15.,  16.,  17.,  18.,  19.,  20.,  21.,
        22.,  23.,  24.,  25.,  26.,  27.,  28.,  29.,  30.,  31.,  32.,
```

```
 33.,  34.,  35.,  36.,  37.,  38.,  39.,  40.,  41.,  42.,  43.,
 44.,  45.,  46.,  47.,  48.,  49.,  50.,  51.,  52.,  53.,  54.,
 55.,  56.,  57.,  58.,  59.,  60.,  61.,  62.,  63.,  64.,  65.,
 66.,  67.,  68.,  69.,  70.,  71.,  72.,  73.,  74.,  75.,  76.,
 77.,  78.,  79.,  80.,  81.,  82.,  83.,  84.,  85.,  86.,  87.,
 88.,  89.,  90.,  91.,  92.,  93.,  94.,  95.,  96.,  97.,  98.,
 99., 100., 101., 102., 103., 104., 105., 106., 107., 108., 109.,
110., 111., 112., 113., 114., 115., 116., 117., 118., 119., 120.,
121., 122., 123., 124., 125., 126., 127., 128., 129., 130., 131.,
132., 133., 134., 135., 136., 137., 138., 139., 140., 141., 142.,
143., 144., 145., 146., 147., 148., 149., 150., 151., 152., 153.,
154., 155., 156., 157., 158., 159., 160., 161., 162., 163., 164.,
165., 166., 167., 168., 169., 170., 171., 172., 173., 174., 175.,
176., 177., 178., 179., 180., 181., 182., 183., 184., 185., 186.,
187., 188., 189., 190., 191., 192., 193., 194., 195., 196., 197.,
198., 199., 200., 201., 202., 203., 204., 205., 206., 207., 208.,
209., 210., 211., 212., 213., 214., 215., 216., 217., 218., 219.,
220., 221., 222., 223., 224., 225., 226., 227., 228., 229., 230.,
231., 232., 233., 234., 235., 236., 237., 238., 239., 240., 241.,
242., 243., 244., 245., 246., 247., 248., 249., 250., 251., 252.,
253., 254., 255., 256., 257., 258., 259., 260., 261., 262., 263.,
264., 265., 266., 267., 268., 269., 270., 271., 272., 273., 274.,
275., 276., 277., 278., 279., 280., 281., 282., 283., 284., 285.,
286., 287., 288., 289., 290., 291., 292., 293., 294., 295., 296.,
297., 298., 299., 300., 301., 302., 303., 304., 305., 306., 307.,
308., 309., 310., 311., 312., 313., 314., 315., 316., 317., 318.,
319., 320., 321., 322., 323., 324., 325., 326., 327., 328., 329.,
330., 331., 332., 333., 334., 335., 336., 337., 338., 339., 340.,
341., 342., 343., 344., 345., 346., 347., 348., 349., 350., 351.,
352., 353., 354., 355., 356., 357., 358., 359., 360., 361., 362.,
363., 364., 365., 366., 367., 368., 369., 370., 371., 372., 373.,
374., 375., 376., 377., 378., 379., 380., 381., 382., 383., 384.,
385., 386., 387., 388., 389., 390., 391., 392., 393., 394., 395.,
396., 397., 398., 399., 400., 401., 402., 403., 404., 405., 406.,
407., 408., 409., 410., 411., 412., 413., 414., 415., 416., 417.,
418., 419., 420., 421., 422., 423., 424., 425., 426., 427., 428.,
429., 430., 431., 432., 433., 434., 435., 436., 437., 438., 439.,
440., 441., 442., 443., 444., 445., 446., 447., 448., 449., 450.,
451., 452., 453., 454., 455., 456., 457., 458., 459., 460., 461.,
462., 463., 464., 465., 466., 467., 468., 469., 470., 471., 472.,
473., 474., 475., 476., 477., 478., 479., 480., 481., 482., 483.,
484., 485., 486., 487., 488., 489., 490., 491., 492., 493., 494.,
495., 496., 497., 498., 499., 500., 501., 502., 503., 504., 505.,
506., 507., 508., 509., 510., 511., 512., 513., 514., 515., 516.,
517., 518., 519., 520., 521., 522., 523., 524., 525., 526., 527.,
528., 529., 530., 531., 532., 533., 534., 535., 536., 537., 538.,
539., 540., 541., 542., 543., 544., 545., 546., 547., 548., 549.,
```

```
       550., 551., 552., 553., 554., 555., 556., 557., 558., 559., 560.,
       561., 562., 563., 564., 565., 566., 567., 568., 569., 570., 571.,
       572., 573., 574., 575., 576., 577., 578., 579., 580., 581., 582.,
       583., 584., 585., 586., 587., 588., 589., 590., 591., 592., 593.,
       594., 595., 596., 597., 598., 599., 600., 601., 602., 603., 604.,
       605., 606., 607., 608., 609., 610., 611., 612., 613., 614., 615.,
       616., 617., 618., 619., 620., 621., 622., 623., 624., 625., 626.,
       627., 628., 629., 630., 631., 632., 633., 634., 635., 636., 637.,
       638., 639., 640., 641., 642., 643., 644., 645., 646., 647., 648.,
       649., 650., 651., 652., 653., 654., 655., 656., 657., 658., 659.,
       660., 661., 662., 663., 664., 665., 666., 667., 668., 669., 670.,
       671., 672., 673., 674., 675., 676., 677., 678., 679., 680., 681.,
       682., 683., 684., 685., 686., 687., 688., 689., 690., 691., 692.,
       693., 694., 695., 696., 697., 698., 699., 700., 701., 702., 703.,
       704., 705., 706., 707., 708., 709., 710., 711., 712., 713., 714.,
       715., 716., 717., 718., 719., 720., 721., 722., 723., 724., 725.,
       726., 727., 728., 729., 730., 731., 732., 733., 734., 735., 736.,
       737., 738., 739., 740., 741., 742., 743., 744., 745., 746., 747.,
       748., 749., 750., 751., 752., 753., 754., 755., 756., 757., 758.,
       759., 760., 761., 762., 763., 764., 765., 766., 767., 768., 769.,
       770., 771., 772., 773., 774., 775., 776., 777., 778., 779., 780.,
       781., 782., 783., 784., 785., 786., 787., 788., 789., 790., 791.,
       792., 793., 794., 795., 796., 797., 798., 799., 800., 801., 802.,
       803., 804., 805., 806., 807., 808., 809., 810., 811., 812., 813.,
       814., 815., 816., 817., 818., 819., 820., 821., 822., 823., 824.,
       825., 826., 827., 828., 829., 830., 831., 832., 833., 834., 835.,
       836., 837., 838., 839., 840., 841., 842., 843., 844., 845., 846.,
       847., 848., 849., 850., 851., 852., 853., 854., 855., 856., 857.,
       858., 859., 860., 861., 862., 863., 864., 865., 866., 867., 868.,
       869., 870., 871., 872., 873., 874., 875., 876., 877., 878., 879.,
       880., 881., 882., 883., 884., 885., 886., 887., 888., 889., 890.,
       891., 892., 893., 894., 895., 896., 897., 898., 899., 900., 901.,
       902., 903., 904., 905., 906., 907., 908., 909., 910., 911., 912.,
       913., 914., 915., 916., 917., 918., 919., 920., 921., 922., 923.,
       924., 925., 926., 927., 928., 929., 930., 931., 932., 933., 934.,
       935., 936., 937., 938., 939., 940., 941., 942., 943., 944., 945.,
       946., 947., 948., 949., 950., 951., 952., 953., 954., 955., 956.,
       957., 958., 959., 960., 961., 962., 963., 964., 965., 966., 967.,
       968., 969., 970., 971., 972., 973., 974., 975., 976., 977., 978.,
       979., 980., 981., 982., 983., 984., 985., 986., 987., 988., 989.,
       990., 991., 992., 993., 994., 995., 996., 997., 998., 999.]),
<BarContainer object of 999 artists>)
```



[34]:
```
num_steps = 500  # sequence length
train_features = torch.tensor([d2l.truncate_pad(
    vocab[line], num_steps, vocab['<pad>']) for line in train_tokens])
print(train_features.shape)
```

```
torch.Size([25000, 500])
```

6.0.3 Question 3 [code] (5 points)

Create data iterator `train_iter`, at each iteration, a minibatch of examples are returned. Let's set the mini-batch size to 64.

[35]:
```
### YOUR CODE HERE

train_labels = torch.tensor(train_data[1]) # Convert the labels to a tensor



train_iter = d2l.load_array((train_features, train_labels), 64)

# Check the shapes of the batches
for X, y in train_iter:
    print('X:', X.shape, ', y:', y.shape)
```

```
    break

# Print the number of batches
print('# batches:', len(train_iter))
```

```
X: torch.Size([64, 500]) , y: torch.Size([64])
# batches: 391
```

6.0.4 Question 4 [code] (5 points)

Finally, wrap up the above steps into the `load_data_imdb` function. It returns training and test data iterators and the vocabulary of the IMDb review dataset.

[36]:
```
#@save
def load_data_imdb(batch_size, num_steps=500):


    # Read training and testing data
    train_data = read_imdb(data_dir, is_train=True)
    test_data = read_imdb(data_dir, is_train=False)

    # Tokenize the reviews
    train_tokens = d2l.tokenize(train_data[0], token='word')
    test_tokens = d2l.tokenize(test_data[0], token='word')

    # Build the vocabulary based on training tokens
    vocab = d2l.Vocab(train_tokens, min_freq=5, reserved_tokens=['<pad>'])

    # Convert reviews to indices and truncate/pad sequences according to␣
↪num_steps
    train_features = torch.tensor([
        d2l.truncate_pad(vocab[line], num_steps, vocab['<pad>']) for line in␣
↪train_tokens
    ])

    test_features = torch.tensor([
        d2l.truncate_pad(vocab[line], num_steps, vocab['<pad>']) for line in␣
↪test_tokens
    ])

    # Convert labels to tensors
    train_labels = torch.tensor(train_data[1])
    test_labels = torch.tensor(test_data[1])

    # Create DataLoaders for training and test sets
    # BECKHAM: Added is_train
    train_iter = d2l.load_array((train_features, train_labels), batch_size,␣
↪is_train=True)
```

```
    test_iter = d2l.load_array((test_features, test_labels), batch_size,␣
↪is_train=False)

    return train_iter, test_iter, vocab
```

7 Using RNN for sentiment analysis (30 points)

Similar to word similarity and analogy tasks, pre-trained word vectors can also be applied to sentiment analysis. Given that the IMDb review dataset is relatively small, using text representations pre-trained on large-scale corpora can help mitigate model overfitting. Each token can be represented using the pre-trained GloVe model, and these token embeddings can be fed into a multilayer bidirectional RNN to generate a sequence representation of the text, which will then be transformed into sentiment analysis outputs. Later, we will explore an alternative architectural approach for the same downstream task.

7.0.1 Question 5 [code] (10 points)

In text classification tasks, such as sentiment analysis, a varying-length text sequence is transformed into fixed-length categorical labels. Following the instructions, please complete `BiRNN` class, each token in a text sequence receives its individual pre-trained GloVe representation through the embedding layer (`self.embedding`). The entire sequence is then encoded by a bidirectional RNN (`self.encoder`). Specifically, the hidden states from the last layer of the bidirectional LSTM, at both the initial and final time steps, are concatenated to form the representation of the text sequence. This representation is subsequently passed through a fully connected layer (`self.decoder`) to produce the final output categories, which in this case are "positive" and "negative".

[14]:
```
batch_size = 64
train_iter, test_iter, vocab = load_data_imdb(batch_size)
```

[53]:
```
class BiRNN(nn.Module):
    def __init__(self, vocab_size, embed_size, num_hiddens,
                 num_layers, **kwargs):
        super(BiRNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # Set `bidirectional` to True to get a bidirectional RNN
        self.encoder = nn.LSTM(embed_size, num_hiddens, num_layers=num_layers,
                               bidirectional=True)
        self.decoder = nn.Linear(4 * num_hiddens, 2)

    def forward(self, inputs):
        # The shape of `inputs` is (batch size, no. of time steps). Because
        # LSTM requires its input's first dimension to be the temporal
        # dimension, the input is transposed before obtaining token
        # representations. The output shape is (no. of time steps, batch size,
        # word vector dimension)

        # Returns hidden states of the last hidden layer at different time
```

```
        # steps. The shape of `outputs` is (no. of time steps, batch size,
        # 2 * no. of hidden units)

        # Concatenate the hidden states at the initial and final time steps as
        # the input of the fully connected layer. Its shape is (batch size,
        # 4 * no. of hidden units)

        ### YOUR CODE HERE

        # 1. Use embedder to get word embeddings, assuming input is a tensor.␣
↪Transpose first.
        embeddings = self.embedding(inputs.T)

        # 2. Put into encoder
        encoder_output, (hidden, cell) = self.encoder(embeddings)

        # 3. Concatenate hidden states
        # TODO: Verify backward_hidden_initial and forward_hidden_initial
        forward_hidden_final = hidden[-2]  # Forward LSTM's final hidden state
        backward_hidden_final = hidden[-1]  # Backward LSTM's final hidden state
        forward_hidden_initial = encoder_output[0, :, :hidden.size(2)]  #␣
↪Initial hidden state
        backward_hidden_initial = encoder_output[-1, :, hidden.size(2):]  #␣
↪Initial hidden state

        concat_hidden = torch.cat(
            (forward_hidden_initial, backward_hidden_initial,
             forward_hidden_final, backward_hidden_final),
             dim=1
        )

        # 4. Put through decoder
        outs = self.decoder(concat_hidden)
        ### END OF YOUR CODE

        return outs
```

Let's construct a bidirectional RNN with two hidden layers to represent single text for sentiment analysis.

[54]:
```
embed_size, num_hiddens, num_layers, devices = 100, 100, 2, d2l.try_all_gpus()
net = BiRNN(len(vocab), embed_size, num_hiddens, num_layers)

def init_weights(module):
    if type(module) == nn.Linear:
        nn.init.xavier_uniform_(module.weight)
    if type(module) == nn.LSTM:
```

```
        for param in module._flat_weights_names:
            if "weight" in param:
                nn.init.xavier_uniform_(module._parameters[param])
net.apply(init_weights)
```

[54]:
```
BiRNN(
  (embedding): Embedding(49347, 100)
  (encoder): LSTM(100, 100, num_layers=2, bidirectional=True)
  (decoder): Linear(in_features=400, out_features=2, bias=True)
)
```

7.0.2 Loading Pretrained Word Vectors

[55]:
```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
embeds = glove_embedding[vocab.idx_to_token]
embeds.shape # Print the shape of the vectors for all the tokens in the␣
 ↪vocabulary.
```

[55]:
```
torch.Size([49347, 100])
```

We use these pretrained word vectors to represent tokens in the reviews and will not update these vectors during training

[56]:
```
net.embedding.weight.data.copy_(embeds)
net.embedding.weight.requires_grad = False
```

7.0.3 Question 6 [code] (10 points)

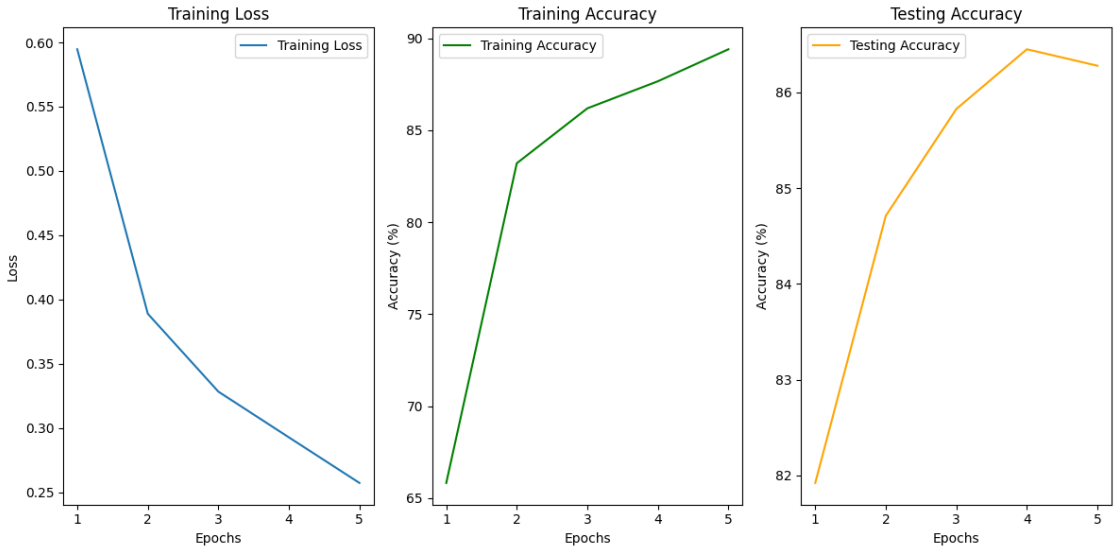
After loading the pretrained word vectors, we can now start to train the model. Please use `Adam` optimizer and `CrossEntropyLoss` for training and draw a graph about your training loss, training acc and testing acc.

[57]:
```
lr, num_epochs = 0.01, 5
### YOUR CODE HERE
# Loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=lr)

# device setup
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")␣
 ↪#***NOTE*** IM USING MPS CUZ IM ON APPLE SILICON
net = net.to(device)

# metrics tracking
training_loss = []
training_accuracy = []
testing_accuracy = []
```

```
# training loop
for epoch in range(num_epochs):
    net.train()  # set model to training mode
    epoch_loss = 0
    correct, total = 0, 0

    for features, labels in train_iter:
        features, labels = features.to(device), labels.to(device)

        # forward pass
        outputs = net(features)
        loss = criterion(outputs, labels)

        # backward pass and optimization
        optimizer.zero_grad() # gotta clear gradient from prev batches
        loss.backward() # backprop to train params
        optimizer.step() #

        # track loss
        epoch_loss += loss.item()

        # track accuracy
        _, predicted = torch.max(outputs, dim=1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    # save metrics
    avg_loss = epoch_loss / len(train_iter)
    training_loss.append(avg_loss)
    training_accuracy.append(100 * correct / total)

    # evaluate on test set
    net.eval()  # model evaluation mode
    correct, total = 0, 0
    with torch.no_grad():
        for features, labels in test_iter:
            features, labels = features.to(device), labels.to(device)
            outputs = net(features)
            _, predicted = torch.max(outputs, dim=1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    test_acc = 100 * correct / total
    testing_accuracy.append(test_acc)

    print(f"Epoch [{epoch + 1}/{num_epochs}], Loss: {avg_loss:.4f}, Train Acc:␣
↪{training_accuracy[-1]:.2f}%, Test Acc: {test_acc:.2f}%")
```

```
# plotting the results
plt.figure(figsize=(12, 6))

# training Loss
plt.subplot(1, 3, 1)
plt.plot(range(1, num_epochs + 1), training_loss, label='Training Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss')
plt.legend()

# training Accuracy
plt.subplot(1, 3, 2)
plt.plot(range(1, num_epochs + 1), training_accuracy, label='Training Accuracy',␣
↪color='green')
plt.xlabel('Epochs')
plt.ylabel('Accuracy (%)')
plt.title('Training Accuracy')
plt.legend()

# testing Accuracy
plt.subplot(1, 3, 3)
plt.plot(range(1, num_epochs + 1), testing_accuracy, label='Testing Accuracy',␣
↪color='orange')
plt.xlabel('Epochs')
plt.ylabel('Accuracy (%)')
plt.title('Testing Accuracy')
plt.legend()

plt.tight_layout()
plt.show()
### END OF YOUR CODE
```

```
Epoch [1/5], Loss: 0.5947, Train Acc: 65.83%, Test Acc: 81.92%
Epoch [2/5], Loss: 0.3889, Train Acc: 83.21%, Test Acc: 84.71%
Epoch [3/5], Loss: 0.3283, Train Acc: 86.19%, Test Acc: 85.83%
Epoch [4/5], Loss: 0.2927, Train Acc: 87.66%, Test Acc: 86.45%
Epoch [5/5], Loss: 0.2572, Train Acc: 89.40%, Test Acc: 86.28%
```



[ ]:

7.0.4 Question 7 [code] (10 points)

Once you have completed the training, it's time to evaluate your model's performance. Implement the function `predict_sentiment` to predict the sentiment of a text sequence using the trained model `net`. Next, define the function `cal_metrics` to assess your model on the test set by calculating both accuracy and the F1-score, including precision and recall. Finally, print out the evaluation results. Save the prediction results for each test sample and submit it in the zip file.

[ ]:

[43]:
```
#@save
def predict_sentiment(net, vocab, sequence):
    """Predict the sentiment of a text sequence."""
    # ensure model is on the correct device
    device = next(net.parameters()).device  # Get current device of the model

    # set model to evaluation mode
    net.eval()

    # sequence tokenize and convert to indices
    tokens = d2l.tokenize([sequence], token='word')[0]
    indices = torch.tensor(d2l.truncate_pad(
        vocab[tokens],
        500,
        vocab['<pad>']
    ))
```

```
    # add batch dimension and move to the same device as the model
    indices = indices.unsqueeze(0).to(device)

    # prediction
    with torch.no_grad():
        print("Starting forward pass...")
        output = net(indices)
        label = torch.argmax(output, dim=1).item()


    ### END OF YOUR CODE
    return 'positive' if label == 1 else 'negative'
```

[59]:
```
predict_sentiment(net, vocab, 'this movie is so great')
```

```
Starting forward pass...
```

[59]:
```
'positive'
```

[60]:
```
predict_sentiment(net, vocab, 'this movie is so bad')
```

```
Starting forward pass...
```

[60]:
```
'negative'
```

[ ]:
```

```

[47]:
```
def save_results(all_sequences, all_labels, all_preds, vocab,␣
↪output_dir="results", zip_filename="submission.zip"):
    """Save predictions and true labels to a CSV and zip the results."""
    # Create results DataFrame
    results_df = pd.DataFrame({
        'sequence': [' '.join([vocab.to_tokens(idx) for idx in seq if idx !=␣
↪vocab['<pad>']]) for seq in all_sequences],
        'true_label': all_labels,
        'predicted_label': all_preds
    })

    # create output directory if it doesn't exist
    os.makedirs(output_dir, exist_ok=True)

    # save csv
    csv_path = os.path.join(output_dir, "test_predictions.csv")
    results_df.to_csv(csv_path, index=False)
    print(f"Results saved to {csv_path}")

    # create a zip file
    zip_path = os.path.join(output_dir, zip_filename)
```

```
    with zipfile.ZipFile(zip_path, 'w') as zipf:
        zipf.write(csv_path, arcname="test_predictions.csv")
    print(f"Results zipped to {zip_path}")

    return csv_path, zip_path


def cal_metrics(net, test_iter, vocab, output_dir="results",␣
↪zip_filename="submission.zip"):
    """Calculate metrics, save predictions to a CSV, and zip the results."""
    device = next(net.parameters()).device
    net = net.to(device)
    net.eval()  # Set model to evaluation mode

    # init lists for predictions, labels, and sequences
    all_preds = []
    all_labels = []
    all_sequences = []

    # collect predictions and true labels
    with torch.no_grad():
        for features, labels in test_iter:
            features, labels = features.to(device), labels.to(device)
            outputs = net(features)
            preds = torch.argmax(outputs, dim=1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
            all_sequences.extend(features.cpu().numpy())  # Save raw input␣
↪sequences

    # calc metrics
    tp = sum(1 for pred, label in zip(all_preds, all_labels) if pred == label ==␣
↪1)
    fp = sum(1 for pred, label in zip(all_preds, all_labels) if pred == 1 and␣
↪label == 0)
    fn = sum(1 for pred, label in zip(all_preds, all_labels) if pred == 0 and␣
↪label == 1)
    tn = sum(1 for pred, label in zip(all_preds, all_labels) if pred == label ==␣
↪0)

    accuracy = sum(1 for pred, label in zip(all_preds, all_labels) if pred ==␣
↪label) / len(all_labels)
    precision = tp / (tp + fp) if (tp + fp) > 0 else 0
    recall = tp / (tp + fn) if (tp + fn) > 0 else 0
    f1 = 2 * (precision * recall) / (precision + recall) if (precision + recall)␣
↪> 0 else 0
```

```
    print(f"Test Set Metrics:\nAccuracy: {accuracy:.4f}\nPrecision: {precision:.
↪4f}\nRecall: {recall:.4f}\nF1-Score: {f1:.4f}")

    # save results using the helper function
    csv_path, zip_path = save_results(all_sequences, all_labels, all_preds,␣
↪vocab, output_dir, zip_filename)

    return f1, precision, recall, accuracy, zip_path
```

[62]:
```
cal_metrics(net, test_iter, vocab)
```

```
Test Set Metrics:
Accuracy: 0.8628
Precision: 0.8434
Recall: 0.8910
F1-Score: 0.8666
Results saved to results\test_predictions.csv
Results zipped to results\submission.zip
```

[62]:
```
(0.86656811639306,
 0.843404513100106,
 0.89104,
 0.8628,
 'results\\submission.zip')
```

8 Using CNN for sentiment analysis (20 points)

Although CNNs were originally designed for computer vision, they have been widely adopted in natural language processing as well. Conceptually, a text sequence can be viewed as a one-dimensional image, allowing one-dimensional CNNs to capture local features, such as n-grams, within the text. We will use the textCNN model to demonstrate how to design a CNN architecture for representing a single text.

Using one-dimensional convolution and max-over-time pooling, the textCNN model takes individual pre-trained token representations as input, then extracts and transforms these sequence representations for downstream tasks.

For a single text sequence with n tokens represented by d-dimensional vectors, the width, height, and number of channels of the input tensor are n, 1, and d, respectively. The textCNN model processes the input as follows:

1. Define multiple one-dimensional convolutional kernels and apply convolution operations on the inputs. Convolution kernels with varying widths capture local features across different numbers of adjacent tokens.
2. Apply max-over-time pooling to all output channels, then concatenate the resulting scalar outputs into a vector.

3. Pass the concatenated vector through a fully connected layer to generate the output categories. Dropout can be applied to reduce overfitting.

8.0.1 Question 8 [code] (10 points)

Implement the `textCNN` model class. Compared with the bidirectional RNN model in Section 2, besides replacing recurrent layers with convolutional layers, we also use two embedding layers: one with trainable weights and the other with fixed weights.

[37]:
```
batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

[38]:
```
class TextCNN(nn.Module):
    def __init__(self, vocab_size, embed_size, kernel_sizes, num_channels,
                 **kwargs):
        super(TextCNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # The embedding layer not to be trained
        self.constant_embedding = nn.Embedding(vocab_size, embed_size)
        self.dropout = nn.Dropout(0.5)
        #self.decoder = nn.Linear(sum(num_channels), 2)


        self.hidden_layer = nn.Linear(sum(num_channels), 128)  # New dense layer␣
↪with 128 units

        # Final dense layer for classification
        self.decoder = nn.Linear(128, 2)  # Connect hidden layer to output


        # The max-over-time pooling layer has no parameters, so this instance
        # can be shared
        self.pool = nn.AdaptiveAvgPool1d(1)
        self.relu = nn.ReLU()
        # Create multiple one-dimensional convolutional layers
        self.convs = nn.ModuleList()
        for c, k in zip(num_channels, kernel_sizes):
            self.convs.append(nn.Conv1d(2 * embed_size, c, k))

    def forward(self, inputs):
        # Concatenate two embedding layer outputs with shape (batch size, no.
        # of tokens, token vector dimension) along vectors

        # Per the input format of one-dimensional convolutional layers,
        # rearrange the tensor so that the second dimension stores channels

        # For each one-dimensional convolutional layer, after max-over-time
        # pooling, a tensor of shape (batch size, no. of channels, 1) is
```

```
        # obtained. Remove the last dimension and concatenate along channels

        ### YOUR CODE HERE
        conv_outputs = []

        # Concatenate embeddings
        embeddings = torch.cat((self.embedding(inputs), self.
↪constant_embedding(inputs)), dim=2)

        # Rearrange dimensions for Conv1
        embeddings = embeddings.permute(0, 2, 1)

        # Apply convolution, ReLU, and pooling for each Conv1d layer
        for conv in self.convs:
            conv_result = conv(embeddings)
            activated_result = self.relu(conv_result)
            pooled_result = self.pool(activated_result)
            squeezed_result = pooled_result.squeeze(-1)
            conv_outputs.append(squeezed_result)

        # Concatenate outputs from all convolutional layers
        concatenated = torch.cat(conv_outputs, dim=1)

        # Apply dropouts
        # To experiment with diff dropout layers
        dropped = self.dropout(concatenated)
        hidden_output = self.relu(self.hidden_layer(dropped))
        dropped_hidden = self.dropout(hidden_output)

        outputs = self.decoder(dropped_hidden)

        ### END OF YOUR CODE

        return outputs
```

[39]:
```
embed_size, kernel_sizes, nums_channels = 100, [3, 4, 5], [100, 100, 100]
devices = d2l.try_all_gpus()
net = TextCNN(len(vocab), embed_size, kernel_sizes, nums_channels)

def init_weights(module):
    if type(module) in (nn.Linear, nn.Conv1d):
        nn.init.xavier_uniform_(module.weight)

net.apply(init_weights)
```

[39]:
```
TextCNN(
  (embedding): Embedding(49346, 100)
```

```
  (constant_embedding): Embedding(49346, 100)
  (dropout): Dropout(p=0.5, inplace=False)
  (hidden_layer): Linear(in_features=300, out_features=128, bias=True)
  (decoder): Linear(in_features=128, out_features=2, bias=True)
  (pool): AdaptiveAvgPool1d(output_size=1)
  (relu): ReLU()
  (convs): ModuleList(
    (0): Conv1d(200, 100, kernel_size=(3,), stride=(1,))
    (1): Conv1d(200, 100, kernel_size=(4,), stride=(1,))
    (2): Conv1d(200, 100, kernel_size=(5,), stride=(1,))
  )
)
```

[40]:
```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
embeds = glove_embedding[vocab.idx_to_token]
net.embedding.weight.data.copy_(embeds)
net.constant_embedding.weight.data.copy_(embeds)
net.constant_embedding.weight.requires_grad = False
```

8.0.2 Question 9 [code] (10 points)

Similar to what we have done in Section 2 with RNN, train the CNN model with the same optimizer and loss function. Draw a graph about your training loss, training acc and testing acc. Use the prediction function you defined in Question 7 to evaluate your model performance.

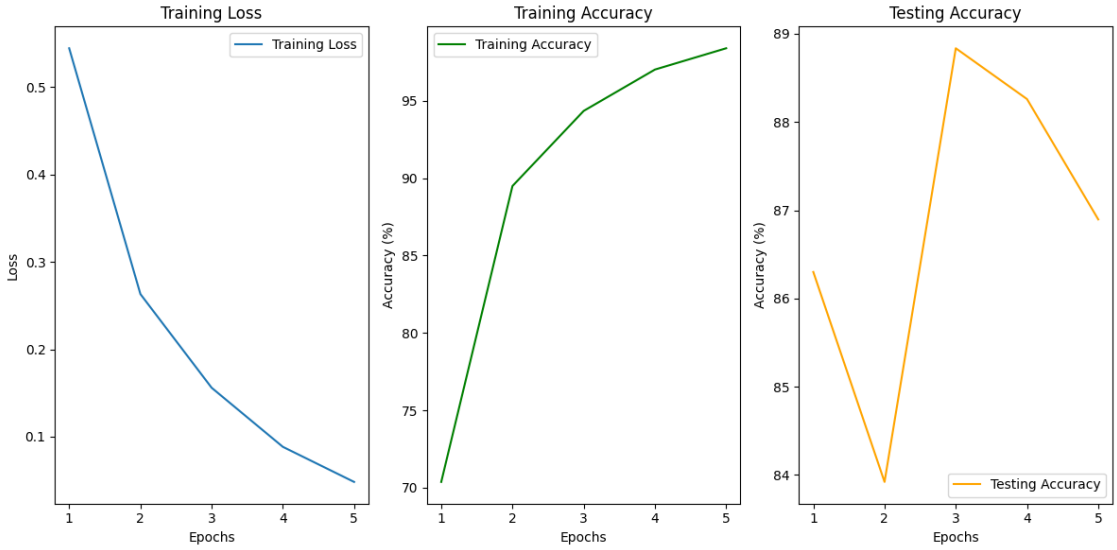
[41]:
```
lr, num_epochs = 0.001, 5

### YOUR CODE HERE

# Loss Func. and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=lr)

# Initialize GPU/CPU  usage (try to use GPU else CPU)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")␣
 ↪#***NOTE*** IM USING MPS CUZ IM ON APPLE SILICON
net = net.to(device)

# Training metrics
training_loss = []
training_accuracy = []
testing_accuracy = []
incorrect_predictions = []

# Train model
for epoch in range(num_epochs):
    net.train()
```

```
    epoch_loss = 0
    total = 0
    correct = 0

    for features, labels in train_iter:
        features = features.to(device)
        labels = labels.to(device)

        # Forwards pass to get outputs
        outputs = net(features)

        # Calculate loss
        loss = criterion(outputs, labels)
        epoch_loss += loss.item()

        # Backwards pass to compute grads
        optimizer.zero_grad() # clear old grads
        loss.backward() # compute grads

        # Weight update
        optimizer.step()

        # Track accuracy
        _, predicted = torch.max(outputs, dim=1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()


        # Find incorrect predictions
        incorrect_indices = (predicted != labels).nonzero(as_tuple=True)[0]

        # Add each incorrect sample as a tuple to the list
        for idx in incorrect_indices:
            feat_list = []
            for j in features[idx].tolist():
                feat_list.append(vocab[j])
            incorrect_predictions.append((
                feat_list,        # Convert feature tensor to list
                predicted[idx].item(),      # Predicted label
                labels[idx].item()          # True label
            ))



    # Save metrics
    avg_loss = epoch_loss/len(train_iter)
    training_loss.append(avg_loss)
```

```
    training_accuracy.append(100 * correct/total)

    # Evaluate test set
    net.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for features, labels in test_iter:
            features = features.to(device)
            labels = labels.to(device)
            outputs = net(features)
            _, predicted = torch.max(outputs, dim=1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    test_acc = 100*correct/total
    testing_accuracy.append(test_acc)

    print(f"Epoch [{epoch + 1}/{num_epochs}], Loss: {avg_loss:.4f}, Train Acc:␣
↪{training_accuracy[-1]:.2f}%, Test Acc: {test_acc:.2f}%")

# Plotting the results
plt.figure(figsize=(12, 6))

# Training Loss
plt.subplot(1, 3, 1)
plt.plot(range(1, num_epochs + 1), training_loss, label='Training Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss')
plt.legend()

# Training Accuracy
plt.subplot(1, 3, 2)
plt.plot(range(1, num_epochs + 1), training_accuracy, label='Training Accuracy',␣
↪color='green')
plt.xlabel('Epochs')
plt.ylabel('Accuracy (%)')
plt.title('Training Accuracy')
plt.legend()

# Testing Accuracy
plt.subplot(1, 3, 3)
plt.plot(range(1, num_epochs + 1), testing_accuracy, label='Testing Accuracy',␣
↪color='orange')
plt.xlabel('Epochs')
plt.ylabel('Accuracy (%)')
```

```
plt.title('Testing Accuracy')
plt.legend()

plt.tight_layout()
plt.show()

### END OF YOUR CODE
```

```
Epoch [1/5], Loss: 0.5446, Train Acc: 70.37%, Test Acc: 86.30%
Epoch [2/5], Loss: 0.2632, Train Acc: 89.50%, Test Acc: 83.92%
Epoch [3/5], Loss: 0.1560, Train Acc: 94.35%, Test Acc: 88.84%
Epoch [4/5], Loss: 0.0884, Train Acc: 97.02%, Test Acc: 88.26%
Epoch [5/5], Loss: 0.0483, Train Acc: 98.40%, Test Acc: 86.90%
```



[44]:
```
predict_sentiment(net, vocab, 'this movie is so great')
```

```
Starting forward pass...
```

[44]: 'positive'

[45]:
```
predict_sentiment(net, vocab, 'this movie is so bad')
```

```
Starting forward pass...
```

[45]: 'negative'

[48]:
```
cal_metrics(net, test_iter, vocab)
```

```
Test Set Metrics:
Accuracy: 0.8690
Precision: 0.8296
```

```
Recall: 0.9286
F1-Score: 0.8763
Results saved to results\test_predictions.csv
Results zipped to results\submission.zip
```

```
[48]: (0.876340027178016,
 0.8296169239565466,
 0.92864,
 0.86896,
 'results\\submission.zip')
```

```
Footnote:
- RNN results are saved in:
results\test_prediction_rnn.csv
results\test_prediction_rnn.zip

- CNN results are saved in:
results\test_prediction_cnn.csv
results\test_prediction_cnn.zip
```