Assignment 1: Sentiment with Deep Neural Networks

Welcome to the first assignment of course 3. In this assignment, you will explore sentiment analysis using deep neural networks.

Outline

In course 1, you implemented Logistic regression and Naive Bayes for sentiment analysis. However if you were to give your old models an example like:

This movie was almost good.

Your model would have predicted a positive sentiment for that review. However, that sentence has a negative sentiment and indicates that the movie was not good. To solve those kinds of misclassifications, you will write a program that uses deep neural networks to identify sentiment in text. By completing this assignment, you will:

- Understand how you can build/design a model using layers
- Train a model using a training loop
- Use a binary cross-entropy loss function
- Compute the accuracy of your model
- Predict using your own input

As you can tell, this model follows a similar structure to the one you previously implemented in the second course of this specialization.

- Indeed most of the deep nets you will be implementing will have a similar structure. The only thing that changes is the model architecture, the inputs, and the outputs. Before starting the assignment, we will introduce you to the Google library `trax` that we use for building and training models.

Now we will show you how to compute the gradient of a certain function `f` by just using `.grad(f)`.

- Trax source code can be found on Github: Trax
- The Trax code also uses the JAX library: JAX

Part 1: Import libraries and try out Trax

- Let's import libraries and look at an example of using the Trax library.

- Let's import libraries and look at an example of using the Trax library.

In [1]:

```
import os
import random as rnd

# import relevant libraries
import trax

# set random seeds to make this notebook easier to replicate
trax.supervised.trainer_lib.init_random_number_generators(31)

# import trax.fastmath.numpy
import trax.fastmath.numpy as np

# import trax.layers
from trax import layers as tl

# import Layer from the utils.py file
from utils import Layer, load_tweets, process_tweet
#from utils import
```

```
INFO:tensorflow:tokens_length=568 inputs_length=512 targets_length=114 noise_density=0.15
mean_noise_span_length=3.0
```

```
[nltk_data] Downloading package twitter_samples to
[nltk_data]     /home/jovyan/nltk_data...
[nltk_data]   Unzipping corpora/twitter_samples.zip.
[nltk_data] Downloading package stopwords to /home/jovyan/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

In [2]:

```
# Create an array using trax.fastmath.numpy
a = np.array(5.0)

# View the returned array
display(a)

print(type(a))
```

```
DeviceArray(5., dtype=float32)
```

```
<class 'jax.interpreters.xla.DeviceArray'>
```

Notice that trax.fastmath.numpy returns a DeviceArray from the jax library.

In [3]:

```
# Define a function that will use the trax.fastmath.numpy array
def f(x):

    # f = x^2
    return (x**2)
```

In [4]:

```
# Call the function
print(f"f(a) for a={a} is {f(a)}")
```

```
f(a) for a=5.0 is 25.0
```

The gradient (derivative) of function `f` with respect to its input `x` is the derivative of x^2.

- The derivative of x^2 is $2x$.
- When x is 5, then $2x=10$.

You can calculate the gradient of a function by using `trax.fastmath.grad(fun=)` and passing in the name of the function.

- In this case the function you want to take the gradient of is `f`.
- The object returned (saved in `grad_f` in this example) is a function that can calculate the gradient of f for a given trax.fastmath.numpy array.

In [5]:

```
# Directly use trax.fastmath.grad to calculate the gradient (derivative) of the function
grad_f = trax.fastmath.grad(fun=f)  # df / dx - Gradient of function f(x) with respect to x

# View the type of the retuned object (it's a function)
type(grad_f)
```

Out[5]:

```
function
```

In [6]:

```
# Call the newly created function and pass in a value for x (the DeviceArray stored in 'a')
grad_calculation = grad_f(a)

# View the result of calling the grad_f function
display(grad_calculation)
```

```
DeviceArray(10., dtype=float32)
```

The function returned by trax.fastmath.grad takes in x=5 and calculates the gradient of f, which is 2*x, which is 10. The value is also stored as a DeviceArray from the jax library.

Part 2: Importing the data

2.1 Loading in the data

Import the data set.

- You may recognize this from earlier assignments in the specialization.
- Details of process_tweet function are available in utils.py file

In [7]:

```
## DO NOT EDIT THIS CELL

# Import functions from the utils.py file

import numpy as np

# Load positive and negative tweets
all_positive_tweets, all_negative_tweets = load_tweets()

# View the total number of positive and negative tweets.
print(f"The number of positive tweets: {len(all_positive_tweets)}")
print(f"The number of negative tweets: {len(all_negative_tweets)}")

# Split positive set into validation and training
val_pos   = all_positive_tweets[4000:] # generating validation set for positive tweets
train_pos  = all_positive_tweets[:4000]# generating training set for positive tweets

# Split negative set into validation and training
val_neg   = all_negative_tweets[4000:] # generating validation set for negative tweets
train_neg  = all_negative_tweets[:4000] # generating training set for nagative tweets

# Combine training data into one set
train_x = train_pos + train_neg

# Combine validation data into one set
```

```
val_x  = val_pos + val_neg

# Set the labels for the training set (1 for positive, 0 for negative)
train_y = np.append(np.ones(len(train_pos)), np.zeros(len(train_neg)))

# Set the labels for the validation set (1 for positive, 0 for negative)
val_y  = np.append(np.ones(len(val_pos)), np.zeros(len(val_neg)))

print(f"length of train_x {len(train_x)}")
print(f"length of val_x {len(val_x)}")
```

```
The number of positive tweets: 5000
The number of negative tweets: 5000
length of train_x 8000
length of val_x 2000
```

Now import a function that processes tweets (we've provided this in the utils.py file).

- `process_tweets' removes unwanted characters e.g. hashtag, hyperlinks, stock tickers from tweet.
- It also returns a list of words (it tokenizes the original string).

In [8]:

```
# Import a function that processes the tweets
# from utils import process_tweet

# Try out function that processes tweets
print("original tweet at training position 0")
print(train_pos[0])

print("Tweet at training position 0 after processing:")
process_tweet(train_pos[0])
```

```
original tweet at training position 0
#FollowFriday @France_Inte @PKuchly57 @Milipol_Paris for being top engaged members in my community
this week :)
Tweet at training position 0 after processing:
```

Out[8]:

```
['followfriday', 'top', 'engag', 'member', 'commun', 'week', ':)']
```

Notice that the function `process_tweet` keeps key words, removes the hash # symbol, and ignores usernames (words that begin with '@'). It also returns a list of the words.

2.2 Building the vocabulary

Now build the vocabulary.

- Map each word in each tweet to an integer (an "index").
- The following code does this for you, but please read it and understand what it's doing.
- Note that you will build the vocabulary based on the training data.
- To do so, you will assign an index to everyword by iterating over your training set.

The vocabulary will also include some special tokens

- `__PAD__`: padding
- `</e>`: end of line
- `__UNK__`: a token representing any word that is not in the vocabulary.

In [9]:

```
# Build the vocabulary
# Unit Test Note - There is no test set here only train/val

# Include special tokens
# started with pad, end of line and unk tokens
```

```python
Vocab = {'__PAD__': 0, '__</e>__': 1, '__UNK__': 2}

# Note that we build vocab using training data
for tweet in train_x:
    processed_tweet = process_tweet(tweet)
    for word in processed_tweet:
        if word not in Vocab:
            Vocab[word] = len(Vocab)

print("Total words in vocab are",len(Vocab))
display(Vocab)
```

```
Total words in vocab are 9088
```

```
{'__PAD__': 0,
 '__</e>__': 1,
 '__UNK__': 2,
 'followfriday': 3,
 'top': 4,
 'engag': 5,
 'member': 6,
 'commun': 7,
 'week': 8,
 ':)': 9,
 'hey': 10,
 'jame': 11,
 'odd': 12,
 ':/': 13,
 'pleas': 14,
 'call': 15,
 'contact': 16,
 'centr': 17,
 '02392441234': 18,
 'abl': 19,
 'assist': 20,
 'mani': 21,
 'thank': 22,
 'listen': 23,
 'last': 24,
 'night': 25,
 'bleed': 26,
 'amaz': 27,
 'track': 28,
 'scotland': 29,
 'congrat': 30,
 'yeaaah': 31,
 'yipppi': 32,
 'accnt': 33,
 'verifi': 34,
 'rqst': 35,
 'succeed': 36,
 'got': 37,
 'blue': 38,
 'tick': 39,
 'mark': 40,
 'fb': 41,
 'profil': 42,
 '15': 43,
 'day': 44,
 'one': 45,
 'irresist': 46,
 'flipkartfashionfriday': 47,
 'like': 48,
 'keep': 49,
 'love': 50,
 'custom': 51,
 'wait': 52,
 'long': 53,
 'hope': 54,
 'enjoy': 55,
 'happi': 56,
 'friday': 57,
 'lwwf': 58,
 'second': 59,
 'thought': 60,
 '’': 61,
```

```
'enough': 62,
'time': 63,
'dd': 64,
'new': 65,
'short': 66,
'enter': 67,
'system': 68,
'sheep': 69,
'must': 70,
'buy': 71,
'jgh': 72,
'go': 73,
'bayan': 74,
':D': 75,
'bye': 76,
'act': 77,
'mischiev': 78,
'etl': 79,
'layer': 80,
'in-hous': 81,
'wareh': 82,
'app': 83,
'katamari': 84,
'well': 85,
'…': 86,
'name': 87,
'impli': 88,
':p': 89,
'influenc': 90,
'big': 91,
'...': 92,
'juici': 93,
'selfi': 94,
'follow': 95,
'perfect': 96,
'alreadi': 97,
'know': 98,
"what'": 99,
'great': 100,
'opportun': 101,
'junior': 102,
'triathlet': 103,
'age': 104,
'12': 105,
'13': 106,
'gatorad': 107,
'seri': 108,
'get': 109,
'entri': 110,
'lay': 111,
'greet': 112,
'card': 113,
'rang': 114,
'print': 115,
'today': 116,
'job': 117,
':-)': 118,
"friend'": 119,
'lunch': 120,
'yummm': 121,
'nostalgia': 122,
'tb': 123,
'ku': 124,
'id': 125,
'conflict': 126,
'help': 127,
"here'": 128,
'screenshot': 129,
'work': 130,
'hi': 131,
'liv': 132,
'hello': 133,
'need': 134,
'someth': 135,
'u': 136,
'fm': 137,
'twitter': 138,
```

'–': 139,
'sure': 140,
'thing': 141,
'dm': 142,
'x': 143,
"i'v": 144,
'heard': 145,
'four': 146,
'season': 147,
'pretti': 148,
'dope': 149,
'penthous': 150,
'obv': 151,
'gobigorgohom': 152,
'fun': 153,
"y'all": 154,
'yeah': 155,
'suppos': 156,
'lol': 157,
'chat': 158,
'bit': 159,
'youth': 160,
'□': 161,
'□': 162,
'□': 163,
'seen': 164,
'year': 165,
'rest': 166,
'goe': 167,
'quickli': 168,
'bed': 169,
'music': 170,
'fix': 171,
'dream': 172,
'spiritu': 173,
'ritual': 174,
'festiv': 175,
'népal': 176,
'begin': 177,
'line-up': 178,
'left': 179,
'see': 180,
'sarah': 181,
'send': 182,
'us': 183,
'email': 184,
'bitsy@bitdefender.com': 185,
"we'll": 186,
'asap': 187,
'kik': 188,
'hatessuc': 189,
'32429': 190,
'kikm': 191,
'lgbt': 192,
'tinder': 193,
'nsfw': 194,
'akua': 195,
'cumshot': 196,
'come': 197,
'hous': 198,
'nsn_supplement': 199,
'effect': 200,
'press': 201,
'releas': 202,
'distribut': 203,
'result': 204,
'link': 205,
'remov': 206,
'pressreleas': 207,
'newsdistribut': 208,
'bam': 209,
'bestfriend': 210,
'lot': 211,
'warsaw': 212,
'<3': 213,
'x46': 214,
'everyon': 215,

```
'watch': 216,
'documentari': 217,
'earthl': 218,
'youtub': 219,
'support': 220,
'buuut': 221,
'oh': 222,
'look': 223,
'forward': 224,
'visit': 225,
'next': 226,
'letsgetmessi': 227,
'jo': 228,
'make': 229,
'feel': 230,
'better': 231,
'never': 232,
'anyon': 233,
'kpop': 234,
'flesh': 235,
'good': 236,
'girl': 237,
'best': 238,
'wish': 239,
'reason': 240,
'epic': 241,
'soundtrack': 242,
'shout': 243,
'ad': 244,
'video': 245,
'playlist': 246,
'would': 247,
'dear': 248,
'jordan': 249,
'okay': 250,
'fake': 251,
'gameplay': 252,
';)': 253,
'haha': 254,
'im': 255,
'kid': 256,
'stuff': 257,
'exactli': 258,
'product': 259,
'line': 260,
'etsi': 261,
'shop': 262,
'check': 263,
'vacat': 264,
'recharg': 265,
'normal': 266,
'charger': 267,
'asleep': 268,
'talk': 269,
'sooo': 270,
'someon': 271,
'text': 272,
'ye': 273,
'bet': 274,
"he'll": 275,
'fit': 276,
'hear': 277,
'speech': 278,
'piti': 279,
'green': 280,
'garden': 281,
'midnight': 282,
'sun': 283,
'beauti': 284,
'canal': 285,
'dasvidaniya': 286,
'till': 287,
'scout': 288,
'sg': 289,
'futur': 290,
'wlan': 291,
'pro': 292,
```

'confer': 293,
'asia': 294,
'chang': 295,
'lollipop': 296,
'□': 297,
'nez': 298,
'agnezmo': 299,
'oley': 300,
'mama': 301,
'stand': 302,
'stronger': 303,
'god': 304,
'misti': 305,
'babi': 306,
'cute': 307,
'woohoo': 308,
"can't": 309,
'sign': 310,
'yet': 311,
'still': 312,
'think': 313,
'mka': 314,
'liam': 315,
'access': 316,
'welcom': 317,
'stat': 318,
'arriv': 319,
'1': 320,
'unfollow': 321,
'via': 322,
'surpris': 323,
'figur': 324,
'happybirthdayemilybett': 325,
'sweet': 326,
'talent': 327,
'2': 328,
'plan': 329,
'drain': 330,
'gotta': 331,
'timezon': 332,
'parent': 333,
'proud': 334,
'least': 335,
'mayb': 336,
'sometim': 337,
'grade': 338,
'al': 339,
'grand': 340,
'manila_bro': 341,
'chosen': 342,
'let': 343,
'around': 344,
'..': 345,
'side': 346,
'world': 347,
'eh': 348,
'take': 349,
'care': 350,
'final': 351,
'fuck': 352,
'weekend': 353,
'real': 354,
'x45': 355,
'join': 356,
'hushedcallwithfraydo': 357,
'gift': 358,
'yeahhh': 359,
'hushedpinwithsammi': 360,
'event': 361,
'might': 362,
'luv': 363,
'realli': 364,
'appreci': 365,
'share': 366,
'wow': 367,
'tom': 368,
'gym': 369,

'gym': 369,
'monday': 370,
'invit': 371,
'scope': 372,
'friend': 373,
'nude': 374,
'sleep': 375,
'birthday': 376,
'want': 377,
't-shirt': 378,
'cool': 379,
'haw': 380,
'phela': 381,
'mom': 382,
'obvious': 383,
'princ': 384,
'charm': 385,
'stage': 386,
'luck': 387,
'tyler': 388,
'hipster': 389,
'glass': 390,
'marti': 391,
'glad': 392,
'done': 393,
'afternoon': 394,
'read': 395,
'kahfi': 396,
'finish': 397,
'ohmyg': 398,
'yaya': 399,
'dub': 400,
'stalk': 401,
'ig': 402,
'gondooo': 403,
'moo': 404,
'tologooo': 405,
'becom': 406,
'detail': 407,
'zzz': 408,
'xx': 409,
'physiotherapi': 410,
'hashtag': 411,
'□': 412,
'monica': 413,
'miss': 414,
'sound': 415,
'morn': 416,
"that'": 417,
'x43': 418,
'definit': 419,
'tri': 420,
'tonight': 421,
'took': 422,
'advic': 423,
'treviso': 424,
'concert': 425,
'citi': 426,
'countri': 427,
"i'll": 428,
'start': 429,
'fine': 430,
'gorgeou': 431,
'xo': 432,
'oven': 433,
'roast': 434,
'garlic': 435,
'oliv': 436,
'oil': 437,
'dri': 438,
'tomato': 439,
'basil': 440,
'centuri': 441,
'tuna': 442,
'right': 443,
'back': 444,
'atchya': 445,
'even': 446,

```
'even': 446,
'almost': 447,
'chanc': 448,
'cheer': 449,
'po': 450,
'ice': 451,
'cream': 452,
'agre': 453,
'100': 454,
'heheheh': 455,
'that': 456,
'point': 457,
'stay': 458,
'home': 459,
'soon': 460,
'promis': 461,
'web': 462,
'whatsapp': 463,
'volta': 464,
'funcionar': 465,
'com': 466,
'iphon': 467,
'jailbroken': 468,
'later': 469,
'34': 470,
'min': 471,
'leia': 472,
'appear': 473,
'hologram': 474,
'r2d2': 475,
'w': 476,
'messag': 477,
'obi': 478,
'wan': 479,
'sit': 480,
'luke': 481,
'inter': 482,
'3': 483,
'ucl': 484,
'arsen': 485,
'small': 486,
'team': 487,
'pass': 488,
'□': 489,
'dewsburi': 490,
'railway': 491,
'station': 492,
'dew': 493,
'west': 494,
'yorkshir': 495,
'430': 496,
'smh': 497,
'9:25': 498,
'live': 499,
'strang': 500,
'imagin': 501,
'megan': 502,
'masaantoday': 503,
'a4': 504,
'shweta': 505,
'tripathi': 506,
'5': 507,
'20': 508,
'kurta': 509,
'half': 510,
'number': 511,
'wsalelov': 512,
'ah': 513,
'larri': 514,
'anyway': 515,
'kinda': 516,
'goood': 517,
'life': 518,
'enn': 519,
'could': 520,
'warmup': 521,
'15th': 522,
'bath': 523,
```

'dum': 524,
'andar': 525,
'ram': 526,
'sampath': 527,
'sona': 528,
'mohapatra': 529,
'samantha': 530,
'edward': 531,
'mein': 532,
'tulan': 533,
'razi': 534,
'wah': 535,
'josh': 536,
'alway': 537,
'smile': 538,
'pictur': 539,
'16.20': 540,
'giveitup': 541,
'given': 542,
'ga': 543,
'subsidi': 544,
'initi': 545,
'propos': 546,
'delight': 547,
'yesterday': 548,
'x42': 549,
'lmaoo': 550,
'song': 551,
'ever': 552,
'shall': 553,
'littl': 554,
'throwback': 555,
'outli': 556,
'island': 557,
'cheung': 558,
'chau': 559,
'mui': 560,
'wo': 561,
'total': 562,
'differ': 563,
'kfckitchentour': 564,
'kitchen': 565,
'clean': 566,
"i'm": 567,
'cusp': 568,
'test': 569,
'water': 570,
'reward': 571,
'arummzz': 572,
"let'": 573,
'drive': 574,
'travel': 575,
'yogyakarta': 576,
'jeep': 577,
'indonesia': 578,
'instamood': 579,
'wanna': 580,
'skype': 581,
'may': 582,
'nice': 583,
'friendli': 584,
'pretend': 585,
'film': 586,
'congratul': 587,
'winner': 588,
'cheesydelight': 589,
'contest': 590,
'address': 591,
'guy': 592,
'market': 593,
'24/7': 594,
'14': 595,
'hour': 596,
'leav': 597,
'without': 598,
'delay': 599,

'actual': 600,
'easi': 601,
'guess': 602,
'train': 603,
'wd': 604,
'shift': 605,
'engin': 606,
'etc': 607,
'sunburn': 608,
'peel': 609,
'blog': 610,
'huge': 611,
'warm': 612,
'☆': 613,
'complet': 614,
'triangl': 615,
'northern': 616,
'ireland': 617,
'sight': 618,
'smthng': 619,
'fr': 620,
'hug': 621,
'xoxo': 622,
'uu': 623,
'jaann': 624,
'topnewfollow': 625,
'connect': 626,
'wonder': 627,
'made': 628,
'fluffi': 629,
'insid': 630,
'pirouett': 631,
'moos': 632,
'trip': 633,
'philli': 634,
'decemb': 635,
"i'd": 636,
'dude': 637,
'x41': 638,
'question': 639,
'flaw': 640,
'pain': 641,
'negat': 642,
'strength': 643,
'went': 644,
'solo': 645,
'move': 646,
'fav': 647,
'nirvana': 648,
'smell': 649,
'teen': 650,
'spirit': 651,
'rip': 652,
'ami': 653,
'winehous': 654,
'coupl': 655,
'tomhiddleston': 656,
'elizabetholsen': 657,
'yaytheylookgreat': 658,
'goodnight': 659,
'vid': 660,
'wake': 661,
'gonna': 662,
'shoot': 663,
'itti': 664,
'bitti': 665,
'teeni': 666,
'bikini': 667,
'much': 668,
'4th': 669,
'togeth': 670,
'end': 671,
'xfile': 672,
'content': 673,
'rain': 674,
'fabul': 675,
'fantast': 676,
'♡': 677

'v': 677,
'jb': 678,
'forev': 679,
'belieb': 680,
'nighti': 681,
'bug': 682,
'bite': 683,
'bracelet': 684,
'idea': 685,
'foundri': 686,
'game': 687,
'sens': 688,
'pic': 689,
'ef': 690,
'phone': 691,
'woot': 692,
'derek': 693,
'use': 694,
'parkshar': 695,
'gloucestershir': 696,
'aaaahhh': 697,
'man': 698,
'traffic': 699,
'stress': 700,
'reliev': 701,
"how'r": 702,
'arbeloa': 703,
'turn': 704,
'17': 705,
'omg': 706,
'say': 707,
'europ': 708,
'rise': 709,
'find': 710,
'hard': 711,
'believ': 712,
'uncount': 713,
'coz': 714,
'unlimit': 715,
'cours': 716,
'teamposit': 717,
'aldub': 718,
'☕': 719,
'rita': 720,
'info': 721,
"we'd": 722,
'way': 723,
'boy': 724,
'x40': 725,
'true': 726,
'sethi': 727,
'high': 728,
'exe': 729,
'skeem': 730,
'saam': 731,
'peopl': 732,
'polit': 733,
'izzat': 734,
'wese': 735,
'trust': 736,
'khawateen': 737,
'k': 738,
'sath': 739,
'mana': 740,
'kar': 741,
'deya': 742,
'sort': 743,
'smart': 744,
'hair': 745,
'tbh': 746,
'jacob': 747,
'g': 748,
'upgrad': 749,
'tee': 750,
'famili': 751,
'person': 752,
'two': 753,

'convers': 754,
'onlin': 755,
'mclaren': 756,
'fridayfeel': 757,
'tgif': 758,
'squar': 759,
'enix': 760,
'bissmillah': 761,
'ya': 762,
'allah': 763,
"we'r": 764,
'socent': 765,
'startup': 766,
'drop': 767,
'your': 768,
'arnd': 769,
'town': 770,
'basic': 771,
'piss': 772,
'cup': 773,
'also': 774,
'terribl': 775,
'complic': 776,
'discuss': 777,
'snapchat': 778,
'lynettelow': 779,
'kikmenow': 780,
'snapm': 781,
'hot': 782,
'amazon': 783,
'kikmeguy': 784,
'defin': 785,
'grow': 786,
'sport': 787,
'rt': 788,
'rakyat': 789,
'write': 790,
'sinc': 791,
'mention': 792,
'fli': 793,
'fish': 794,
'promot': 795,
'post': 796,
'cyber': 797,
'ourdaughtersourprid': 798,
'mypapamyprid': 799,
'papa': 800,
'coach': 801,
'posit': 802,
'kha': 803,
'atleast': 804,
'x39': 805,
'mango': 806,
"lassi'": 807,
"monty'": 808,
'marvel': 809,
'though': 810,
'suspect': 811,
'meant': 812,
'24': 813,
'hr': 814,
'touch': 815,
'kepler': 816,
'452b': 817,
'chalna': 818,
'hai': 819,
'thankyou': 820,
'hazel': 821,
'food': 822,
'brooklyn': 823,
'pta': 824,
'awak': 825,
'okayi': 826,
'awww': 827,
'ha': 828,
'doc': 829,
'splendid': 830,

```
'spam': 831,
'folder': 832,
'amount': 833,
'nigeria': 834,
'claim': 835,
'rted': 836,
'leg': 837,
'hurt': 838,
'bad': 839,
'mine': 840,
'saturday': 841,
'thaaank': 842,
'puhon': 843,
'happinesss': 844,
'tnc': 845,
'prior': 846,
'notif': 847,
'fat': 848,
'co': 849,
'probabl': 850,
'ate': 851,
'yuna': 852,
'tamesid': 853,
'´': 854,
'googl': 855,
'account': 856,
'scouser': 857,
'everyth': 858,
'zoe': 859,
'mate': 860,
'liter': 861,
"they'r": 862,
'samee': 863,
'edgar': 864,
'updat': 865,
'log': 866,
'bring': 867,
'abe': 868,
'meet': 869,
'x38': 870,
'sigh': 871,
'dreamili': 872,
'pout': 873,
'eye': 874,
'quacketyquack': 875,
'funni': 876,
'happen': 877,
'phil': 878,
'em': 879,
'del': 880,
'rodder': 881,
'els': 882,
'play': 883,
'newest': 884,
'gamejam': 885,
'irish': 886,
'literatur': 887,
'inaccess': 888,
"kareena'": 889,
'fan': 890,
'brain': 891,
'dot': 892,
'braindot': 893,
'fair': 894,
'rush': 895,
'either': 896,
'brandi': 897,
'18': 898,
'carniv': 899,
'men': 900,
'put': 901,
'mask': 902,
'xavier': 903,
'forneret': 904,
'jennif': 905,
'site': 906,
'free': 907,
```

```
'50.000': 908,
'8': 909,
'ball': 910,
'pool': 911,
'coin': 912,
'edit': 913,
'trish': 914,
'♥': 915,
'grate': 916,
'three': 917,
'comment': 918,
'wakeup': 919,
'besid': 920,
'dirti': 921,
'sex': 922,
'lmaooo': 923,
'□': 924,
'loui': 925,
"he'": 926,
'throw': 927,
'caus': 928,
'inspir': 929,
'ff': 930,
'twoof': 931,
'gr8': 932,
'wkend': 933,
'kind': 934,
'exhaust': 935,
'word': 936,
'cheltenham': 937,
'area': 938,
'kale': 939,
'crisp': 940,
'ruin': 941,
'x37': 942,
'open': 943,
'worldwid': 944,
'outta': 945,
'sfvbeta': 946,
'vantast': 947,
'xcylin': 948,
'bundl': 949,
'show': 950,
'internet': 951,
'price': 952,
'realisticli': 953,
'pay': 954,
'net': 955,
'educ': 956,
'power': 957,
'weapon': 958,
'nelson': 959,
'mandela': 960,
'recent': 961,
'j': 962,
'chenab': 963,
'flow': 964,
'pakistan': 965,
'incredibleindia': 966,
'teenchoic': 967,
'choiceinternationalartist': 968,
'superjunior': 969,
'caught': 970,
'first': 971,
'salmon': 972,
'super-blend': 973,
'project': 974,
'youth@bipolaruk.org.uk': 975,
'awesom': 976,
'stream': 977,
'alma': 978,
'mater': 979,
'highschoolday': 980,
'clientvisit': 981,
'faith': 982,
'christian': 983,
'school': 984,
```

```
 'lizaminnelli': 985,
 'upcom': 986,
 'uk': 987,
 '😄': 988,
 'singl': 989,
 'hill': 990,
 'everi': 991,
 'beat': 992,
 'wrong': 993,
 'readi': 994,
 'natur': 995,
 'pefumeri': 996,
 'workshop': 997,
 'neal': 998,
 'yard': 999,
 ...}
```

The dictionary `Vocab` will look like this:

```
{'__PAD__': 0,
 '__</e>__': 1,
 '__UNK__': 2,
 'followfriday': 3,
 'top': 4,
 'engag': 5,
 ...
```

- Each unique word has a unique integer associated with it.
- The total number of words in Vocab: 9088

2.3 Converting a tweet to a tensor

Write a function that will convert each tweet to a tensor (a list of unique integer IDs representing the processed tweet).

- Note, the returned data type will be a **regular Python `list()`**
    - You won't use TensorFlow in this function
    - You also won't use a numpy array
    - You also won't use trax.fastmath.numpy array
- For words in the tweet that are not in the vocabulary, set them to the unique ID for the token `__UNK__`.

Example

Input a tweet:

```
'@happypuppy, is Maria happy?'
```

The tweet_to_tensor will first conver the tweet into a list of tokens (including only relevant words)

```
['maria', 'happi']
```

Then it will convert each word into its unique integer

```
[2, 56]
```

- Notice that the word "maria" is not in the vocabulary, so it is assigned the unique integer associated with the `__UNK__` token, because it is considered "unknown."

Exercise 01

Instructions: Write a program `tweet_to_tensor` that takes in a tweet and converts it to an array of numbers. You can use the `Vocab` dictionary you just found to help create the tensor.

- Use the vocab_dict parameter and not a global variable.
- Do not hard code the integer value for the `__UNK__` token.

▶ **Hints**

In [10]:

```python
# UNQ_C1 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: tweet_to_tensor
def tweet_to_tensor(tweet, vocab_dict, unk_token='__UNK__', verbose=False):
    '''
    Input:
        tweet - A string containing a tweet
        vocab_dict - The words dictionary
        unk_token - The special string for unknown tokens
        verbose - Print info durign runtime
    Output:
        tensor_l - A python list with

    '''

    ### START CODE HERE (Replace instances of 'None' with your code) ###
    # Process the tweet into a list of words
    # where only important words are kept (stop words removed)
    word_l = process_tweet(tweet)

    if verbose:
        print("List of words from the processed tweet:")
        print(word_l)

    # Initialize the list that will contain the unique integer IDs of each word
    tensor_l = []

    # Get the unique integer ID of the __UNK__ token
    unk_ID = vocab_dict[unk_token]

    if verbose:
        print(f"The unique integer ID for the unk_token is {unk_ID}")

    # for each word in the list:
    for word in word_l:

        # Get the unique integer ID.
        # If the word doesn't exist in the vocab dictionary,
        # use the unique ID for __UNK__ instead.
        word_ID = vocab_dict[word] if word in vocab_dict else unk_ID
    ### END CODE HERE ###

        # Append the unique integer ID to the tensor list.
        tensor_l.append(word_ID)

    return tensor_l
```

In [11]:

```python
print("Actual tweet is\n", val_pos[0])
print("\nTensor of tweet:\n", tweet_to_tensor(val_pos[0], vocab_dict=Vocab))
```

```
Actual tweet is
 Bro:U wan cut hair anot,ur hair long Liao bo
Me:since ord liao,take it easy lor treat as save $ leave it longer :)
Bro:LOL Sibei xialan

Tensor of tweet:
 [1065, 136, 479, 2351, 745, 8148, 1123, 745, 53, 2, 2672, 791, 2, 2, 349, 601, 2, 3489, 1017,
597, 4559, 9, 1065, 157, 2, 2]
```

Expected output

```
Actual tweet is
 Bro:U wan cut hair anot,ur hair long Liao bo
Me:since ord liao,take it easy lor treat as save $ leave it longer :)
Bro:LOL Sibei xialan

Tensor of tweet:
```

```
[1065, 136, 479, 2351, 745, 8148, 1123, 745, 53, 2, 2672, 791, 2, 2, 349, 601, 2, 3489, 1017
, 597, 4559, 9, 1065, 157, 2, 2]
```

In [12]:

```python
# test tweet_to_tensor

def test_tweet_to_tensor():
    test_cases = [

        {
            "name":"simple_test_check",
            "input": [val_pos[1], Vocab],
            "expected":[444, 2, 304, 567, 56, 9],
            "error":"The function gives bad output for val_pos[1]. Test failed"
        },
        {
            "name":"datatype_check",
            "input":[val_pos[1], Vocab],
            "expected":type([]),
            "error":"Datatype mismatch. Need only list not np.array"
        },
        {
            "name":"without_unk_check",
            "input":[val_pos[1], Vocab],
            "expected":6,
            "error":"Unk word check not done- Please check if you included mapping for unknown
word"
        }
    ]
    count = 0
    for test_case in test_cases:

        try:
            if test_case['name'] == "simple_test_check":
                assert test_case["expected"] == tweet_to_tensor(*test_case['input'])
                count += 1
            if test_case['name'] == "datatype_check":
                assert isinstance(tweet_to_tensor(*test_case['input']), test_case["expected"])
                count += 1
            if test_case['name'] == "without_unk_check":
                assert None not in tweet_to_tensor(*test_case['input'])
                count += 1



        except:
            print(test_case['error'])
    if count == 3:
        print("\033[92m All tests passed")
    else:
        print(count," Tests passed out of 3")
test_tweet_to_tensor()
```

```
All tests passed
```

2.4 Creating a batch generator

Most of the time in Natural Language Processing, and AI in general we use batches when training our data sets.

- If instead of training with batches of examples, you were to train a model with one example at a time, it would take a very long time to train the model.
- You will now build a data generator that takes in the positive/negative tweets and returns a batch of training examples. It returns the model inputs, the targets (positive or negative labels) and the weight for each target (ex: this allows us to can treat some examples as more important to get right than others, but commonly this will all be 1.0).

Once you create the generator, you could include it in a for loop

```
for batch_inputs, batch_targets, batch_example_weights in data_generator:
    ...
```

You can also get a single batch like this:

```
batch_inputs, batch_targets, batch_example_weights = next(data_generator)
```

The generator returns the next batch each time it's called.

- This generator returns the data in a format (tensors) that you could directly use in your model.
- It returns a triple: the inputs, targets, and loss weights: -- Inputs is a tensor that contains the batch of tweets we put into the model. -- Targets is the corresponding batch of labels that we train to generate. -- Loss weights here are just 1s with same shape as targets. Next week, you will use it to mask input padding.

Exercise 02

Implement `data_generator`.

In [13]:

```python
# UNQ_C2 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED: Data generator
def data_generator(data_pos, data_neg, batch_size, loop, vocab_dict, shuffle=False):
    '''
    Input:
        data_pos - Set of posstive examples
        data_neg - Set of negative examples
        batch_size - number of samples per batch. Must be even
        loop - True or False
        vocab_dict - The words dictionary
        shuffle - Shuffle the data order
    Yield:
        inputs - Subset of positive and negative examples
        targets - The corresponding labels for the subset
        example_weights - An array specifying the importance of each example

    '''
### START GIVEN CODE ###
    # make sure the batch size is an even number
    # to allow an equal number of positive and negative samples
    assert batch_size % 2 == 0

    # Number of positive examples in each batch is half of the batch size
    # same with number of negative examples in each batch
    n_to_take = batch_size // 2

    # Use pos_index to walk through the data_pos array
    # same with neg_index and data_neg
    pos_index = 0
    neg_index = 0

    len_data_pos = len(data_pos)
    len_data_neg = len(data_neg)

    # Get and array with the data indexes
    pos_index_lines = list(range(len_data_pos))
    neg_index_lines = list(range(len_data_neg))

    # shuffle lines if shuffle is set to True
    if shuffle:
        rnd.shuffle(pos_index_lines)
        rnd.shuffle(neg_index_lines)

    stop = False

    # Loop indefinitely
    while not stop:

        # create a batch with positive and negative examples
        batch = []

        # First part: Pack n_to_take positive examples

        # Start from pos_index and increment i up to n_to_take
        for i in range(n_to_take):
```

```
        for i in range(n_to_take):

            # If the positive index goes past the positive dataset lenght,
            if pos_index >= len_data_pos:

                # If loop is set to False, break once we reach the end of the dataset
                if not loop:
                    stop = True;
                    break;

                # If user wants to keep re-using the data, reset the index
                pos_index = 0

                if shuffle:
                    # Shuffle the index of the positive sample
                    rnd.shuffle(pos_index_lines)

            # get the tweet as pos_index
            tweet = data_pos[pos_index_lines[pos_index]]

            # convert the tweet into tensors of integers representing the processed words
            tensor = tweet_to_tensor(tweet, vocab_dict)

            # append the tensor to the batch list
            batch.append(tensor)

            # Increment pos_index by one
            pos_index = pos_index + 1

### END GIVEN CODE ###

### START CODE HERE (Replace instances of 'None' with your code) ###

        # Second part: Pack n_to_take negative examples

        # Using the same batch list, start from neg_index and increment i up to n_to_take
        for i in range(n_to_take):

            # If the negative index goes past the negative dataset length,
            if neg_index > len(data_neg):

                # If loop is set to False, break once we reach the end of the dataset
                if not loop:
                    stop = True;
                    break;

                # If user wants to keep re-using the data, reset the index
                neg_index = 0

                if shuffle:
                    # Shuffle the index of the negative sample
                    rnd.shuffle(neg_index_lines)
            # get the tweet as neg_index
            tweet = data_neg[neg_index_lines[neg_index]]

            # convert the tweet into tensors of integers representing the processed words
            tensor = tweet_to_tensor(tweet, vocab_dict)

            # append the tensor to the batch list
            batch.append(tensor)

            # Increment neg_index by one
            neg_index += 1

### END CODE HERE ###

### START GIVEN CODE ###
        if stop:
            break;

        # Update the start index for positive data
        # so that it's n_to_take positions after the current pos_index
        pos_index += n_to_take

        # Update the start index for negative data
        # so that it's n_to_take positions after the current neg_index
        neg_index += n_to_take
```

```
        # Get the max tweet length (the length of the longest tweet)
        # (you will pad all shorter tweets to have this length)
        max_len = max([len(t) for t in batch])


        # Initialize the input_l, which will
        # store the padded versions of the tensors
        tensor_pad_l = []
        # Pad shorter tweets with zeros
        for tensor in batch:
### END GIVEN CODE ###

### START CODE HERE (Replace instances of 'None' with your code) ###
            # Get the number of positions to pad for this tensor so that it will be max_len long
            n_pad = max_len - len(tensor)

            # Generate a list of zeros, with length n_pad
            pad_l = [0] * n_pad

            # concatenate the tensor and the list of padded zeros
            tensor_pad = tensor + pad_l

            # append the padded tensor to the list of padded tensors
            tensor_pad_l.append(tensor_pad)

        # convert the list of padded tensors to a numpy array
        # and store this as the model inputs
        inputs = np.array(tensor_pad_l)

        # Generate the list of targets for the positive examples (a list of ones)
        # The length is the number of positive examples in the batch
        target_pos = [1] * n_to_take

        # Generate the list of targets for the negative examples (a list of zeros)
        # The length is the number of negative examples in the batch
        target_neg = [0] * n_to_take

        # Concatenate the positve and negative targets
        target_l = target_pos + target_neg

        # Convert the target list into a numpy array
        targets = np.array(target_l)

        # Example weights: Treat all examples equally importantly.It should return an np.array. Hi
nt: Use np.ones_like()
        example_weights = np.ones_like(targets)


### END CODE HERE ###

### GIVEN CODE ###
        # note we use yield and not return
        yield inputs, targets, example_weights
```

Now you can use your data generator to create a data generator for the training data, and another data generator for the validation data.

We will create a third data generator that does not loop, for testing the final accuracy of the model.

In [14]:

```
# Set the random number generator for the shuffle procedure
rnd.seed(30)

# Create the training data generator
def train_generator(batch_size, shuffle = False):
    return data_generator(train_pos, train_neg, batch_size, True, Vocab, shuffle)

# Create the validation data generator
def val_generator(batch_size, shuffle = False):
    return data_generator(val_pos, val_neg, batch_size, True, Vocab, shuffle)

# Create the validation data generator
def test_generator(batch_size, shuffle = False):
    return data_generator(val_pos, val_neg, batch_size, False, Vocab, shuffle)
```

```
    return data_generator(val_pos, val_neg, batch_size, False, vocab, shuffle)

# Get a batch from the train_generator and inspect.
inputs, targets, example_weights = next(train_generator(4, shuffle=True))

# this will print a list of 4 tensors padded with zeros
print(f'Inputs: {inputs}')
print(f'Targets: {targets}')
print(f'Example Weights: {example_weights}')
```

```
Inputs: [[2005 4451 3201    9    0    0    0    0    0    0    0]
 [4954  567 2000 1454 5174 3499  141 3499  130  459    9]
 [3761  109  136  583 2930 3969    0    0    0    0    0]
 [ 250 3761    0    0    0    0    0    0    0    0    0]]
Targets: [1 1 0 0]
Example Weights: [1 1 1 1]
```

In [15]:

```
# Test the train_generator

# Create a data generator for training data,
# which produces batches of size 4 (for tensors and their respective targets)
tmp_data_gen = train_generator(batch_size = 4)

# Call the data generator to get one batch and its targets
tmp_inputs, tmp_targets, tmp_example_weights = next(tmp_data_gen)

print(f"The inputs shape is {tmp_inputs.shape}")
print(f"The targets shape is {tmp_targets.shape}")
print(f"The example weights shape is {tmp_example_weights.shape}")

for i,t in enumerate(tmp_inputs):
    print(f"input tensor: {t}; target {tmp_targets[i]}; example weights {tmp_example_weights[i]}")
```

```
The inputs shape is (4, 14)
The targets shape is (4,)
The example weights shape is (4,)
input tensor: [3 4 5 6 7 8 9 0 0 0 0 0 0 0]; target 1; example weights 1
input tensor: [10 11 12 13 14 15 16 17 18 19 20  9 21 22]; target 1; example weights 1
input tensor: [5738 2901 3761    0    0    0    0    0    0    0    0    0    0    0]; target 0;
example weights 1
input tensor: [ 858  256 3652 5739  307 4458  567 1230 2767  328 1202 3761    0    0]; target 0; e
xample weights 1
```

Expected output

```
The inputs shape is (4, 14)
The targets shape is (4,)
The example weights shape is (4,)
input tensor: [3 4 5 6 7 8 9 0 0 0 0 0 0 0]; target 1; example weights 1
input tensor: [10 11 12 13 14 15 16 17 18 19 20  9 21 22]; target 1; example weights 1
input tensor: [5738 2901 3761    0    0    0    0    0    0    0    0    0    0    0]; targe
0; example weights 1
input tensor: [ 858  256 3652 5739  307 4458  567 1230 2767  328 1202 3761    0    0]; targe
0; example weights 1
```

Now that you have your train/val generators, you can just call them and they will return tensors which correspond to your tweets in the first column and their corresponding labels in the second column. Now you can go ahead and start building your neural network.

Part 3: Defining classes

In this part, you will write your own library of layers. It will be very similar to the one used in Trax and also in Keras and PyTorch. Writing your own small framework will help you understand how they all work and use them effectively in the future.

Your framework will be based on the following `Layer` class from utils.py.

```
class Layer(object):
    """ Base class for layers.
    """

    # Constructor
    def __init__(self):
        # set weights to None
        self.weights = None

    # The forward propagation should be implemented
    # by subclasses of this Layer class
    def forward(self, x):
        raise NotImplementedError

    # This function initializes the weights
    # based on the input signature and random key,
    # should be implemented by subclasses of this Layer class
    def init_weights_and_state(self, input_signature, random_key):
        pass

    # This initializes and returns the weights, do not override.
    def init(self, input_signature, random_key):
        self.init_weights_and_state(input_signature, random_key)
        return self.weights

    # __call__ allows an object of this class
    # to be called like it's a function.
    def __call__(self, x):
        # When this layer object is called,
        # it calls its forward propagation function
        return self.forward(x)
```

3.1 ReLU class

You will now implement the ReLU activation function in a class below. The ReLU function looks as follows:

$$ \mathrm{ReLU}(x) = \mathrm{max}(0,x) $$

Exercise 03

Instructions: Implement the ReLU activation function below. Your function should take in a matrix or vector and it should transform all the negative numbers into 0 while keeping all the positive numbers intact.

▶ **Hints**

In [16]:

```
# UNQ_C3 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: Relu
class Relu(Layer):
    """Relu activation function implementation"""
    def forward(self, x):
        '''
        Input:
            - x (a numpy array): the input
        Output:
```

```
        - activation (numpy array): all positive or 0 version of x
    '''
    ### START CODE HERE (Replace instances of 'None' with your code) ###

    activation = np.maximum(0, x)

    ### END CODE HERE ###

    return activation
```

In [17]:

```
# Test your relu function
x = np.array([[-2.0, -1.0, 0.0], [0.0, 1.0, 2.0]], dtype=float)
relu_layer = Relu()
print("Test data is:")
print(x)
print("Output of Relu is:")
print(relu_layer(x))
```

```
Test data is:
[[-2. -1.  0.]
 [ 0.  1.  2.]]
Output of Relu is:
[[0. 0. 0.]
 [0. 1. 2.]]
```

Expected Outout

```
Test data is:
[[-2. -1.  0.]
 [ 0.  1.  2.]]
Output of Relu is:
[[0. 0. 0.]
 [0. 1. 2.]]
```

3.2 Dense class

Exercise

Implement the forward function of the Dense class.

- The forward function multiplies the input to the layer (`x`) by the weight matrix (`W`)

$$\mathrm{forward}(\mathbf{x},\mathbf{W}) = \mathbf{xW} $$

- You can use `numpy.dot` to perform the matrix multiplication.

Note that for more efficient code execution, you will use the trax version of `math` , which includes a trax version of `numpy` and also `random` .

Implement the weight initializer `new_weights` function

- Weights are initialized with a random key.
- The second parameter is a tuple for the desired shape of the weights (num_rows, num_cols)
- The num of rows for weights should equal the number of columns in x, because for forward propagation, you will multiply x times weights.

Please use `trax.fastmath.random.normal(key, shape, dtype=tf.float32)` to generate random values for the weight matrix. The key difference between this function and the standard `numpy` randomness is the explicit use of random keys, which need to be passed. While it can look tedious at the first sight to pass the random key everywhere, you will learn in Course 4 why this is very helpful when implementing some advanced models.

- `key` can be generated by calling `random.get_prng(seed=)` and passing in a number for the `seed` .
- `shape` is a tuple with the desired shape of the weight matrix.
  - The number of rows in the weight matrix should equal the number of columns in the variable `x` . Since `x` may have 2

dimensions if it reprsents a single training example (row, col), or three dimensions (batch_size, row, col), get the last dimension from the tuple that holds the dimensions of x.
    - The number of columns in the weight matrix is the number of units chosen for that dense layer. Look at the `__init__` function to see which variable stores the number of units.
- `dtype` is the data type of the values in the generated matrix; keep the default of `tf.float32`. In this case, don't explicitly set the dtype (just let it use the default value).

Set the standard deviation of the random values to 0.1

- The values generated have a mean of 0 and standard deviation of 1.
- Set the default standard deviation `stdev` to be 0.1 by multiplying the standard deviation to each of the values in the weight matrix.

In [18]:

```
# use the fastmath module within trax
from trax import fastmath

# use the numpy module from trax
np = fastmath.numpy

# use the fastmath.random module from trax
random = fastmath.random
```

In [19]:

```
# See how the fastmath.trax.random.normal function works
tmp_key = random.get_prng(seed=1)
print("The random seed generated by random.get_prng")
display(tmp_key)

print("choose a matrix with 2 rows and 3 columns")
tmp_shape=(2,3)
display(tmp_shape)

# Generate a weight matrix
# Note that you'll get an error if you try to set dtype to tf.float32, where tf is tensorflow
# Just avoid setting the dtype and allow it to use the default data type
tmp_weight = trax.fastmath.random.normal(key=tmp_key, shape=tmp_shape)

print("Weight matrix generated with a normal distribution with mean 0 and stdev of 1")
display(tmp_weight)
```

```
The random seed generated by random.get_prng

DeviceArray([0, 1], dtype=uint32)

choose a matrix with 2 rows and 3 columns

(2, 3)

Weight matrix generated with a normal distribution with mean 0 and stdev of 1

DeviceArray([[ 0.95730704, -0.96992904,  1.0070664 ],
             [ 0.36619025,  0.17294823,  0.29092228]], dtype=float32)
```

Exercise 04

Implement the `Dense` class.

In [20]:

```
# UNQ_C4 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: Dense

class Dense(Layer):
    """
```

```
    A dense (fully-connected) layer.
    """

    # __init__ is implemented for you
    def __init__(self, n_units, init_stdev=0.1):

        # Set the number of units in this layer
        self._n_units = n_units
        self._init_stdev = init_stdev

    # Please implement 'forward()'
    def forward(self, x):

### START CODE HERE (Replace instances of 'None' with your code) ###

        # Matrix multiply x and the weight matrix
        dense = np.dot(x, self.weights)

### END CODE HERE ###
        return dense

    # init_weights
    def init_weights_and_state(self, input_signature, random_key):

### START CODE HERE (Replace instances of 'None' with your code) ###
        # The input_signature has a .shape attribute that gives the shape as a tuple
        input_shape = input_signature.shape

        # Generate the weight matrix from a normal distribution,
        # and standard deviation of 'stdev'
        w = self._init_stdev * random.normal(key = random_key, shape = (input_shape[-1], self._n_uni
ts))

### END CODE HERE ###
        self.weights = w
        return self.weights
```

In [21]:

```
# Testing your Dense layer
dense_layer = Dense(n_units=10)  #sets  number of units in dense layer
random_key = random.get_prng(seed=0)  # sets random seed
z = np.array([[2.0, 7.0, 25.0]]) # input array

dense_layer.init(z, random_key)
print("Weights are\n ",dense_layer.weights) #Returns randomly generated weights
print("Foward function output is ", dense_layer(z)) # Returns multiplied values of units and
weights
```

```
Weights are
  [[-0.02837108  0.09368162 -0.10050076  0.14165013  0.10543301  0.09108126
  -0.04265672  0.0986188  -0.05575325  0.00153249]
 [-0.20785688  0.0554837   0.09142365  0.05744595  0.07227863  0.01210617
  -0.03237354  0.16234995  0.02450038 -0.13809784]
 [-0.06111237  0.01403724  0.08410042 -0.1094358  -0.10775021 -0.11396459
  -0.05933381 -0.01557652 -0.03832145 -0.11144515]]
Foward function output is  [[-3.0395496   0.9266802   2.5414743  -2.050473   -1.9769388  -2.582209
  -1.7952735   0.94427425 -0.8980402  -3.7497487 ]]
```

Expected Outout

```
Weights are
  [[-0.02837108  0.09368162 -0.10050076  0.14165013  0.10543301  0.09108126
  -0.04265672  0.0986188  -0.05575325  0.00153249]
 [-0.20785688  0.0554837   0.09142365  0.05744595  0.07227863  0.01210617
  -0.03237354  0.16234995  0.02450038 -0.13809784]
 [-0.06111237  0.01403724  0.08410042 -0.1094358  -0.10775021 -0.11396459
  -0.05933381 -0.01557652 -0.03832145 -0.11144515]]
Foward function output is  [[-3.0395496   0.9266802   2.5414743  -2.050473   -1.9769388  -2.
582209
  -1.7952735   0.94427425 -0.8980402  -3.7497487 ]]
```

3.3 Model

Now you will implement a classifier using neural networks. Here is the model architecture you will be implementing.

For the model implementation, you will use the Trax layers library `tl`. Note that the second character of `tl` is the lowercase of letter `L`, not the number 1. Trax layers are very similar to the ones you implemented above, but in addition to trainable weights also have a non-trainable state. State is used in layers like batch normalization and for inference, you will learn more about it in course 4.

First, look at the code of the Trax Dense layer and compare to your implementation above.

- [tl.Dense](): Trax Dense layer implementation

One other important layer that you will use a lot is one that allows to execute one layer after another in sequence.

- [tl.Serial](): Combinator that applies layers serially.
  - You can pass in the layers as arguments to `Serial`, separated by commas.
  - For example: `tl.Serial(tl.Embeddings(...), tl.Mean(...), tl.Dense(...), tl.LogSoftmax(...))`

Please use the `help` function to view documentation for each layer.

In [22]:

```
# View documentation on tl.Dense
help(tl.Dense)
```

```
Help on class Dense in module trax.layers.core:

class Dense(trax.layers.base.Layer)
 |  Dense(n_units, kernel_initializer=<function ScaledInitializer.<locals>.Init at 
0x7f8590622620>, bias_initializer=<function RandomNormalInitializer.<locals>.<lambda> at 
0x7f85906226a8>, use_bias=True)
 |  
 |  A dense (a.k.a. fully-connected, affine) layer.
 |  
 |  Dense layers are the prototypical example of a trainable layer, i.e., a layer
 |  with trainable weights. Each node in a dense layer computes a weighted sum of
 |  all node values from the preceding layer and adds to that sum a node-specific
 |  bias term. The full layer computation is expressed compactly in linear
 |  algebra as an affine map `y = Wx + b`, where `W` is a matrix and `y`, `x`,
 |  and `b` are vectors. The layer is trained, or "learns", by updating the
 |  values in `W` and `b`.
 |  
 |  Less commonly, a dense layer can omit the bias term and be a pure linear map:
 |  `y = Wx`.
 |  
 |  Method resolution order:
 |      Dense
 |      trax.layers.base.Layer
 |      builtins.object
 |  
 |  Methods defined here:
 |  
 |  __init__(self, n_units, kernel_initializer=<function ScaledInitializer.<locals>.Init at 
0x7f8590622620>, bias_initializer=<function RandomNormalInitializer.<locals>.<lambda> at 
0x7f85906226a8>, use_bias=True)
 |      Returns a dense (fully connected) layer of width `n_units`.
```

```
 |      Returns a dense (fully connected) layer of width `n_units`.
 |
 |      A dense layer maps collections of `R^m` vectors to `R^n`, where `n`
 |      (`= n_units`) is fixed at layer creation time, and `m` is set at layer
 |      initialization time.
 |
 |      Args:
 |        n_units: Number of nodes in the layer, also known as the width of the
 |            layer.
 |        kernel_initializer: Function that creates a matrix of (random) initial
 |            connection weights `W` for the layer.
 |        bias_initializer: Function that creates a vector of (random) initial
 |            bias weights `b` for the layer.
 |        use_bias: If `True`, compute an affine map `y = Wx + b`; else compute
 |            a linear map `y = Wx`.
 |
 |  forward(self, x)
 |      Executes this layer as part of a forward pass through the model.
 |
 |      Args:
 |        x: Tensor of same shape and dtype as the input signature used to
 |            initialize this layer.
 |
 |      Returns:
 |        Tensor of same shape and dtype as the input, except the final dimension
 |        is the layer's `n_units` value.
 |
 |  init_weights_and_state(self, input_signature)
 |      Returns newly initialized weights for this layer.
 |
 |      Weights are a `(w, b)` tuple for layers created with `use_bias=True` (the
 |      default case), or a `w` tensor for layers created with `use_bias=False`.
 |
 |      Args:
 |        input_signature: `ShapeDtype` instance characterizing the input this layer
 |            should compute on.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from trax.layers.base.Layer:
 |
 |  __call__(self, x, weights=None, state=None, rng=None)
 |      Makes layers callable; for use in tests or interactive settings.
 |
 |      This convenience method helps library users play with, test, or otherwise
 |      probe the behavior of layers outside of a full training environment. It
 |      presents the layer as callable function from inputs to outputs, with the
 |      option of manually specifying weights and non-parameter state per individual
 |      call. For convenience, weights and non-parameter state are cached per layer
 |      instance, starting from default values of `EMPTY_WEIGHTS` and `EMPTY_STATE`,
 |      and acquiring non-empty values either by initialization or from values
 |      explicitly provided via the weights and state keyword arguments.
 |
 |      Args:
 |        x: Zero or more input tensors, packaged as described in the `Layer` class
 |            docstring.
 |        weights: Weights or `None`; if `None`, use self's cached weights value.
 |        state: State or `None`; if `None`, use self's cached state value.
 |        rng: Single-use random number generator (JAX PRNG key), or `None`;
 |            if `None`, use a default computed from an integer 0 seed.
 |
 |      Returns:
 |        Zero or more output tensors, packaged as described in the `Layer` class
 |        docstring.
 |
 |  __repr__(self)
 |      Return repr(self).
 |
 |  backward(self, inputs, output, grad, weights, state, new_state, rng)
 |      Custom backward pass to propagate gradients in a custom way.
 |
 |      Args:
 |        inputs: Input tensors; can be a (possibly nested) tuple.
 |        output: The result of running this layer on inputs.
 |        grad: Gradient signal computed based on subsequent layers; its structure
 |            and shape must match output.
 |        weights: This layer's weights.
 |        state: This layer's state prior to the current forward pass.
 |        new_state: This layer's state after the current forward pass.
```

```
 |        new_state: This layer's state after the current forward pass.
 |        rng: Single-use random number generator (JAX PRNG key).
 |
 |      Returns:
 |        The custom gradient signal for the input. Note that we need to return
 |        a gradient for each argument of forward, so it will usually be a tuple
 |        of signals: the gradient for inputs and weights.
 |
 |  init(self, input_signature, rng=None, use_cache=False)
 |      Initializes weights/state of this layer and its sublayers recursively.
 |
 |      Initialization creates layer weights and state, for layers that use them.
 |      It derives the necessary array shapes and data types from the layer's input
 |      signature, which is itself just shape and data type information.
 |
 |      For layers without weights or state, this method safely does nothing.
 |
 |      This method is designed to create weights/state only once for each layer
 |      instance, even if the same layer instance occurs in multiple places in the
 |      network. This enables weight sharing to be implemented as layer sharing.
 |
 |      Args:
 |        input_signature: `ShapeDtype` instance (if this layer takes one input)
 |            or list/tuple of `ShapeDtype` instances.
 |        rng: Single-use random number generator (JAX PRNG key), or `None`;
 |            if `None`, use a default computed from an integer 0 seed.
 |        use_cache: If `True`, and if this layer instance has already been
 |            initialized elsewhere in the network, then return special marker
 |            values -- tuple `(GET_WEIGHTS_FROM_CACHE, GET_STATE_FROM_CACHE)`.
 |            Else return this layer's newly initialized weights and state.
 |
 |      Returns:
 |        A `(weights, state)` tuple.
 |
 |  init_from_file(self, file_name, weights_only=False, input_signature=None)
 |      Initializes this layer and its sublayers from a pickled checkpoint.
 |
 |      In the common case (`weights_only=False`), the file must be a gziped pickled
 |      dictionary containing items with keys `'flat_weights', `'flat_state'` and
 |      `'input_signature'`, which are used to initialize this layer.
 |      If `input_signature` is specified, it's used instead of the one in the file.
 |      If `weights_only` is `True`, the dictionary does not need to have the
 |      `'flat_state'` item and the state it not restored either.
 |
 |      Args:
 |        file_name: Name/path of the pickeled weights/state file.
 |        weights_only: If `True`, initialize only the layer's weights. Else
 |            initialize both weights and state.
 |        input_signature: Input signature to be used instead of the one from file.
 |
 |  output_signature(self, input_signature)
 |      Returns output signature this layer would give for `input_signature`.
 |
 |  pure_fn(self, x, weights, state, rng, use_cache=False)
 |      Applies this layer as a pure function with no optional args.
 |
 |      This method exposes the layer's computation as a pure function. This is
 |      especially useful for JIT compilation. Do not override, use `forward`
 |      instead.
 |
 |      Args:
 |        x: Zero or more input tensors, packaged as described in the `Layer` class
 |            docstring.
 |        weights: A tuple or list of trainable weights, with one element for this
 |            layer if this layer has no sublayers, or one for each sublayer if
 |            this layer has sublayers. If a layer (or sublayer) has no trainable
 |            weights, the corresponding weights element is an empty tuple.
 |        state: Layer-specific non-parameter state that can update between batches.
 |        rng: Single-use random number generator (JAX PRNG key).
 |        use_cache: if `True`, cache weights and state in the layer object; used
 |          to implement layer sharing in combinators.
 |
 |      Returns:
 |        A tuple of `(tensors, state)`. The tensors match the number (`n_out`)
 |        promised by this layer, and are packaged as described in the `Layer`
 |        class docstring.
 |
```

```
 |  weights_and_state_signature(self, input_signature)
 |      Return a pair containing the signatures of weights and state.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from trax.layers.base.Layer:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  has_backward
 |      Returns `True` if this layer provides its own custom backward pass code.
 |
 |      A layer subclass that provides custom backward pass code (for custom
 |      gradients) must override this method to return `True`.
 |
 |  n_in
 |      Returns how many tensors this layer expects as input.
 |
 |  n_out
 |      Returns how many tensors this layer promises as output.
 |
 |  name
 |      Returns the name of this layer.
 |
 |  rng
 |      Returns a single-use random number generator without advancing it.
 |
 |  state
 |      Returns a tuple containing this layer's state; may be empty.
 |
 |  sublayers
 |      Returns a tuple containing this layer's sublayers; may be empty.
 |
 |  weights
 |      Returns this layer's weights.
 |
 |      Depending on the layer, the weights can be in the form of:
 |
 |        - an empty tuple
 |        - a tensor (ndarray)
 |        - a nested structure of tuples and tensors
```

In [23]:

```
# View documentation on tl.Serial
help(tl.Serial)
```

```
Help on class Serial in module trax.layers.combinators:

class Serial(trax.layers.base.Layer)
 |  Serial(*sublayers, name=None, sublayers_to_print=None)
 |
 |  Combinator that applies layers serially (by function composition).
 |
 |  This combinator is commonly used to construct deep networks, e.g., like this::
 |
 |      mlp = tl.Serial(
 |        tl.Dense(128),
 |        tl.Relu(),
 |        tl.Dense(10),
 |        tl.LogSoftmax()
 |      )
 |
 |  A Serial combinator uses stack semantics to manage data for its sublayers.
 |  Each sublayer sees only the inputs it needs and returns only the outputs it
 |  has generated. The sublayers interact via the data stack. For instance, a
 |  sublayer k, following sublayer j, gets called with the data stack in the
 |  state left after layer j has applied. The Serial combinator then:
 |
 |    - takes n_in items off the top of the stack (n_in = k.n_in) and calls
 |      layer k, passing those items as arguments; and
 |
```

```
 |    - takes layer k's n_out return values (n_out = k.n_out) and pushes
 |      them onto the data stack.
 |
 |  A Serial instance with no sublayers acts as a special-case (but useful)
 |  1-input 1-output no-op.
 |
 |  Method resolution order:
 |      Serial
 |      trax.layers.base.Layer
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, *sublayers, name=None, sublayers_to_print=None)
 |      Creates a partially initialized, unconnected layer instance.
 |
 |      Args:
 |        n_in: Number of inputs expected by this layer.
 |        n_out: Number of outputs promised by this layer.
 |        name: Class-like name for this layer; for use when printing this layer.
 |        sublayers_to_print: Sublayers to display when printing out this layer;
 |          By default (when None) we display all sublayers.
 |
 |  forward(self, xs)
 |      Computes this layer's output as part of a forward pass through the model.
 |
 |      Authors of new layer subclasses should override this method to define the
 |      forward computation that their layer performs. Use `self.weights` to access
 |      trainable weights of this layer. If you need to use local non-trainable
 |      state or randomness, use `self.rng` for the random seed (no need to set it)
 |      and use `self.state` for non-trainable state (and set it to the new value).
 |
 |      Args:
 |        inputs: Zero or more input tensors, packaged as described in the `Layer`
 |            class docstring.
 |
 |      Returns:
 |        Zero or more output tensors, packaged as described in the `Layer` class
 |        docstring.
 |
 |  init_weights_and_state(self, input_signature)
 |      Initializes weights and state for inputs with the given signature.
 |
 |      Authors of new layer subclasses should override this method if their layer
 |      uses trainable weights or non-trainable state. To initialize trainable
 |      weights, set `self.weights` and to initialize non-trainable state,
 |      set `self.state` to the intended value.
 |
 |      Args:
 |        input_signature: A `ShapeDtype` instance (if this layer takes one input)
 |            or a list/tuple of `ShapeDtype` instances; signatures of inputs.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  state
 |      Returns a tuple containing this layer's state; may be empty.
 |
 |  weights
 |      Returns this layer's weights.
 |
 |      Depending on the layer, the weights can be in the form of:
 |
 |        - an empty tuple
 |        - a tensor (ndarray)
 |        - a nested structure of tuples and tensors
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from trax.layers.base.Layer:
 |
 |  __call__(self, x, weights=None, state=None, rng=None)
 |      Makes layers callable; for use in tests or interactive settings.
 |
 |      This convenience method helps library users play with, test, or otherwise
 |      probe the behavior of layers outside of a full training environment. It
 |      presents the layer as callable function from inputs to outputs, with the
 |      option of manually specifying weights and non-parameter state per individual
```

```
 |      option of manually specifying weights and non-parameter state per individual
 |      call. For convenience, weights and non-parameter state are cached per layer
 |      instance, starting from default values of `EMPTY_WEIGHTS` and `EMPTY_STATE`,
 |      and acquiring non-empty values either by initialization or from values
 |      explicitly provided via the weights and state keyword arguments.
 |
 |      Args:
 |        x: Zero or more input tensors, packaged as described in the `Layer` class
 |            docstring.
 |        weights: Weights or `None`; if `None`, use self's cached weights value.
 |        state: State or `None`; if `None`, use self's cached state value.
 |        rng: Single-use random number generator (JAX PRNG key), or `None`;
 |            if `None`, use a default computed from an integer 0 seed.
 |
 |      Returns:
 |        Zero or more output tensors, packaged as described in the `Layer` class
 |        docstring.
 |
 |  __repr__(self)
 |      Return repr(self).
 |
 |  backward(self, inputs, output, grad, weights, state, new_state, rng)
 |      Custom backward pass to propagate gradients in a custom way.
 |
 |      Args:
 |        inputs: Input tensors; can be a (possibly nested) tuple.
 |        output: The result of running this layer on inputs.
 |        grad: Gradient signal computed based on subsequent layers; its structure
 |            and shape must match output.
 |        weights: This layer's weights.
 |        state: This layer's state prior to the current forward pass.
 |        new_state: This layer's state after the current forward pass.
 |        rng: Single-use random number generator (JAX PRNG key).
 |
 |      Returns:
 |        The custom gradient signal for the input. Note that we need to return
 |        a gradient for each argument of forward, so it will usually be a tuple
 |        of signals: the gradient for inputs and weights.
 |
 |  init(self, input_signature, rng=None, use_cache=False)
 |      Initializes weights/state of this layer and its sublayers recursively.
 |
 |      Initialization creates layer weights and state, for layers that use them.
 |      It derives the necessary array shapes and data types from the layer's input
 |      signature, which is itself just shape and data type information.
 |
 |      For layers without weights or state, this method safely does nothing.
 |
 |      This method is designed to create weights/state only once for each layer
 |      instance, even if the same layer instance occurs in multiple places in the
 |      network. This enables weight sharing to be implemented as layer sharing.
 |
 |      Args:
 |        input_signature: `ShapeDtype` instance (if this layer takes one input)
 |            or list/tuple of `ShapeDtype` instances.
 |        rng: Single-use random number generator (JAX PRNG key), or `None`;
 |            if `None`, use a default computed from an integer 0 seed.
 |        use_cache: If `True`, and if this layer instance has already been
 |            initialized elsewhere in the network, then return special marker
 |            values -- tuple `(GET_WEIGHTS_FROM_CACHE, GET_STATE_FROM_CACHE)`.
 |            Else return this layer's newly initialized weights and state.
 |
 |      Returns:
 |        A `(weights, state)` tuple.
 |
 |  init_from_file(self, file_name, weights_only=False, input_signature=None)
 |      Initializes this layer and its sublayers from a pickled checkpoint.
 |
 |      In the common case (`weights_only=False`), the file must be a gziped pickled
 |      dictionary containing items with keys `'flat_weights', `'flat_state'` and
 |      `'input_signature'`, which are used to initialize this layer.
 |      If `input_signature` is specified, it's used instead of the one in the file.
 |      If `weights_only` is `True`, the dictionary does not need to have the
 |      `'flat_state'` item and the state it not restored either.
 |
 |      Args:
 |        file_name: Name/path of the pickeled weights/state file.
 |        weights_only: If `True`, initialize only the layer's weights. Else
```

```
 |          weights_only: If `True`, initialize only the layer's weights. Else
 |              initialize both weights and state.
 |          input_signature: Input signature to be used instead of the one from file.
 |  
 |  output_signature(self, input_signature)
 |      Returns output signature this layer would give for `input_signature`.
 |  
 |  pure_fn(self, x, weights, state, rng, use_cache=False)
 |      Applies this layer as a pure function with no optional args.
 |      
 |      This method exposes the layer's computation as a pure function. This is
 |      especially useful for JIT compilation. Do not override, use `forward`
 |      instead.
 |      
 |      Args:
 |        x: Zero or more input tensors, packaged as described in the `Layer` class
 |            docstring.
 |        weights: A tuple or list of trainable weights, with one element for this
 |            layer if this layer has no sublayers, or one for each sublayer if
 |            this layer has sublayers. If a layer (or sublayer) has no trainable
 |            weights, the corresponding weights element is an empty tuple.
 |        state: Layer-specific non-parameter state that can update between batches.
 |        rng: Single-use random number generator (JAX PRNG key).
 |        use_cache: if `True`, cache weights and state in the layer object; used
 |          to implement layer sharing in combinators.
 |      
 |      Returns:
 |        A tuple of `(tensors, state)`. The tensors match the number (`n_out`)
 |        promised by this layer, and are packaged as described in the `Layer`
 |        class docstring.
 |  
 |  weights_and_state_signature(self, input_signature)
 |      Return a pair containing the signatures of weights and state.
 |  
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from trax.layers.base.Layer:
 |  
 |  __dict__
 |      dictionary for instance variables (if defined)
 |  
 |  __weakref__
 |      list of weak references to the object (if defined)
 |  
 |  has_backward
 |      Returns `True` if this layer provides its own custom backward pass code.
 |      
 |      A layer subclass that provides custom backward pass code (for custom
 |      gradients) must override this method to return `True`.
 |  
 |  n_in
 |      Returns how many tensors this layer expects as input.
 |  
 |  n_out
 |      Returns how many tensors this layer promises as output.
 |  
 |  name
 |      Returns the name of this layer.
 |  
 |  rng
 |      Returns a single-use random number generator without advancing it.
 |  
 |  sublayers
 |      Returns a tuple containing this layer's sublayers; may be empty.
```

- [tl.Embedding](): Layer constructor function for an embedding layer.
  - `tl.Embedding(vocab_size, d_feature)`.
  - `vocab_size` is the number of unique words in the given vocabulary.
  - `d_feature` is the number of elements in the word embedding (some choices for a word embedding size range from 150 to 300, for example).

In [24]:

```
# View documentation for tl.Embedding
```

```
help(tl.Embedding)
```

```
Help on class Embedding in module trax.layers.core:

class Embedding(trax.layers.base.Layer)
 |  Embedding(vocab_size, d_feature, kernel_initializer=<function RandomNormalInitializer.
<locals>.<lambda> at 0x7f85906228c8>)
 |
 |  Trainable layer that maps discrete tokens/ids to vectors.
 |
 |  Method resolution order:
 |      Embedding
 |      trax.layers.base.Layer
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, vocab_size, d_feature, kernel_initializer=<function RandomNormalInitializer.<lo
cals>.<lambda> at 0x7f85906228c8>)
 |      Returns an embedding layer with given vocabulary size and vector size.
 |
 |      The layer clips input values (token ids) to the range `[0, vocab_size)`.
 |      That is, negative token ids all clip to `0` before being mapped to a
 |      vector, and token ids with value `vocab_size` or greater all clip to
 |      `vocab_size - 1` before being mapped to a vector.
 |
 |      Args:
 |        vocab_size: Size of the input vocabulary. The layer will assign a unique
 |            vector to each id in `range(vocab_size)`.
 |        d_feature: Dimensionality/depth of the output vectors.
 |        kernel_initializer: Function that creates (random) initial vectors for
 |            the embedding.
 |
 |  forward(self, x)
 |      Returns embedding vectors corresponding to input token id's.
 |
 |      Args:
 |        x: Tensor of token id's.
 |
 |      Returns:
 |        Tensor of embedding vectors.
 |
 |  init_weights_and_state(self, input_signature)
 |      Returns tensor of newly initialized embedding vectors.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from trax.layers.base.Layer:
 |
 |  __call__(self, x, weights=None, state=None, rng=None)
 |      Makes layers callable; for use in tests or interactive settings.
 |
 |      This convenience method helps library users play with, test, or otherwise
 |      probe the behavior of layers outside of a full training environment. It
 |      presents the layer as callable function from inputs to outputs, with the
 |      option of manually specifying weights and non-parameter state per individual
 |      call. For convenience, weights and non-parameter state are cached per layer
 |      instance, starting from default values of `EMPTY_WEIGHTS` and `EMPTY_STATE`,
 |      and acquiring non-empty values either by initialization or from values
 |      explicitly provided via the weights and state keyword arguments.
 |
 |      Args:
 |        x: Zero or more input tensors, packaged as described in the `Layer` class
 |            docstring.
 |        weights: Weights or `None`; if `None`, use self's cached weights value.
 |        state: State or `None`; if `None`, use self's cached state value.
 |        rng: Single-use random number generator (JAX PRNG key), or `None`;
 |            if `None`, use a default computed from an integer 0 seed.
 |
 |      Returns:
 |        Zero or more output tensors, packaged as described in the `Layer` class
 |        docstring.
 |
 |  __repr__(self)
 |      Return repr(self).
 |
 |  backward(self, inputs, output, grad, weights, state, new_state, rng)
```

```
 |      Custom backward pass to propagate gradients in a custom way.
 |
 |      Args:
 |        inputs: Input tensors; can be a (possibly nested) tuple.
 |        output: The result of running this layer on inputs.
 |        grad: Gradient signal computed based on subsequent layers; its structure
 |            and shape must match output.
 |        weights: This layer's weights.
 |        state: This layer's state prior to the current forward pass.
 |        new_state: This layer's state after the current forward pass.
 |        rng: Single-use random number generator (JAX PRNG key).
 |
 |      Returns:
 |        The custom gradient signal for the input. Note that we need to return
 |        a gradient for each argument of forward, so it will usually be a tuple
 |        of signals: the gradient for inputs and weights.
 |
 |  init(self, input_signature, rng=None, use_cache=False)
 |      Initializes weights/state of this layer and its sublayers recursively.
 |
 |      Initialization creates layer weights and state, for layers that use them.
 |      It derives the necessary array shapes and data types from the layer's input
 |      signature, which is itself just shape and data type information.
 |
 |      For layers without weights or state, this method safely does nothing.
 |
 |      This method is designed to create weights/state only once for each layer
 |      instance, even if the same layer instance occurs in multiple places in the
 |      network. This enables weight sharing to be implemented as layer sharing.
 |
 |      Args:
 |        input_signature: `ShapeDtype` instance (if this layer takes one input)
 |            or list/tuple of `ShapeDtype` instances.
 |        rng: Single-use random number generator (JAX PRNG key), or `None`;
 |            if `None`, use a default computed from an integer 0 seed.
 |        use_cache: If `True`, and if this layer instance has already been
 |            initialized elsewhere in the network, then return special marker
 |            values -- tuple `(GET_WEIGHTS_FROM_CACHE, GET_STATE_FROM_CACHE)`.
 |            Else return this layer's newly initialized weights and state.
 |
 |      Returns:
 |        A `(weights, state)` tuple.
 |
 |  init_from_file(self, file_name, weights_only=False, input_signature=None)
 |      Initializes this layer and its sublayers from a pickled checkpoint.
 |
 |      In the common case (`weights_only=False`), the file must be a gziped pickled
 |      dictionary containing items with keys `'flat_weights', `'flat_state'` and
 |      `'input_signature'`, which are used to initialize this layer.
 |      If `input_signature` is specified, it's used instead of the one in the file.
 |      If `weights_only` is `True`, the dictionary does not need to have the
 |      `'flat_state'` item and the state it not restored either.
 |
 |      Args:
 |        file_name: Name/path of the pickeled weights/state file.
 |        weights_only: If `True`, initialize only the layer's weights. Else
 |            initialize both weights and state.
 |        input_signature: Input signature to be used instead of the one from file.
 |
 |  output_signature(self, input_signature)
 |      Returns output signature this layer would give for `input_signature`.
 |
 |  pure_fn(self, x, weights, state, rng, use_cache=False)
 |      Applies this layer as a pure function with no optional args.
 |
 |      This method exposes the layer's computation as a pure function. This is
 |      especially useful for JIT compilation. Do not override, use `forward`
 |      instead.
 |
 |      Args:
 |        x: Zero or more input tensors, packaged as described in the `Layer` class
 |            docstring.
 |        weights: A tuple or list of trainable weights, with one element for this
 |            layer if this layer has no sublayers, or one for each sublayer if
 |            this layer has sublayers. If a layer (or sublayer) has no trainable
 |            weights, the corresponding weights element is an empty tuple.
 |        state: Layer-specific non-parameter state that can update between batches.
 |        rng: Single-use random number generator (JAX PRNG key).
```

```
 |          rng: Single-use random number generator (JAX PRNG key).
 |          use_cache: if `True`, cache weights and state in the layer object; used
 |            to implement layer sharing in combinators.
 |      
 |      Returns:
 |        A tuple of `(tensors, state)`. The tensors match the number (`n_out`)
 |        promised by this layer, and are packaged as described in the `Layer`
 |        class docstring.
 |  
 |  weights_and_state_signature(self, input_signature)
 |      Return a pair containing the signatures of weights and state.
 |  
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from trax.layers.base.Layer:
 |  
 |  __dict__
 |      dictionary for instance variables (if defined)
 |  
 |  __weakref__
 |      list of weak references to the object (if defined)
 |  
 |  has_backward
 |      Returns `True` if this layer provides its own custom backward pass code.
 |      
 |      A layer subclass that provides custom backward pass code (for custom
 |      gradients) must override this method to return `True`.
 |  
 |  n_in
 |      Returns how many tensors this layer expects as input.
 |  
 |  n_out
 |      Returns how many tensors this layer promises as output.
 |  
 |  name
 |      Returns the name of this layer.
 |  
 |  rng
 |      Returns a single-use random number generator without advancing it.
 |  
 |  state
 |      Returns a tuple containing this layer's state; may be empty.
 |  
 |  sublayers
 |      Returns a tuple containing this layer's sublayers; may be empty.
 |  
 |  weights
 |      Returns this layer's weights.
 |      
 |      Depending on the layer, the weights can be in the form of:
 |      
 |        - an empty tuple
 |        - a tensor (ndarray)
 |        - a nested structure of tuples and tensors
```

In [25]:

```
tmp_embed = tl.Embedding(vocab_size=3, d_feature=2)
display(tmp_embed)
```

```
Embedding_3_2
```

- tl.Mean: Calculates means across an axis. In this case, please choose axis = 1 to get an average embedding vector (an embedding vector that is an average of all words in the vocabulary).
- For example, if the embedding matrix is 300 elements and vocab size is 10,000 words, taking the mean of the embedding matrix along axis=1 will yield a vector of 300 elements.

In [26]:

```
# view the documentation for tl.mean
help(tl.Mean)
```

```
Help on function Mean in module trax.layers.core:

Mean(axis=-1, keepdims=False)
    Returns a layer that computes mean values using one tensor axis.

    `Mean` uses one tensor axis to form groups of values and replaces each group
    with the mean value of that group. The resulting values can either remain
    in their own size 1 axis (`keepdims=True`), or that axis can be removed from
    the overall tensor (default `keepdims=False`), lowering the rank of the
    tensor by one.

    Args:
      axis: Axis along which values are grouped for computing a mean.
      keepdims: If `True`, keep the resulting size 1 axis as a separate tensor
          axis; else, remove that axis.
```

In [27]:

```
# Pretend the embedding matrix uses
# 2 elements for embedding the meaning of a word
# and has a vocabulary size of 3
# So it has shape (2,3)
tmp_embed = np.array([[1,2,3,],
                      [4,5,6]
                     ])

# take the mean along axis 0
print("The mean along axis 0 creates a vector whose length equals the vocabulary size")
display(np.mean(tmp_embed,axis=0))

print("The mean along axis 1 creates a vector whose length equals the number of elements in a word embedding")
display(np.mean(tmp_embed,axis=1))
```

```
The mean along axis 0 creates a vector whose length equals the vocabulary size


DeviceArray([2.5, 3.5, 4.5], dtype=float32)


The mean along axis 1 creates a vector whose length equals the number of elements in a word embedd
ing


DeviceArray([2., 5.], dtype=float32)
```

- tl.LogSoftmax: Implements log softmax function
- Here, you don't need to set any parameters for `LogSoftMax()`.

In [28]:

```
help(tl.LogSoftmax)
```

```
Help on function LogSoftmax in module trax.layers.core:

LogSoftmax(axis=-1)
    Returns a layer that applies log softmax along one tensor axis.

    `LogSoftmax` acts on a group of values and normalizes them to look like a set
    of log probability values. (Probability values must be non-negative, and as
    a set must sum to 1. A group of log probability values can be seen as the
    natural logarithm function applied to a set of probability values.)

    Args:
      axis: Axis along which values are grouped for computing log softmax.
```

Online documentation

- tl.Dense
- tl.Serial

- tl.Embedding
- tl.Mean
- tl.LogSoftmax

Exercise 05

Implement the classifier function.

In [29]:

```
# UNQ_C5 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: classifier
def classifier(vocab_size=len(Vocab), embedding_dim=256, output_dim=2, mode='train'):

### START CODE HERE (Replace instances of 'None' with your code) ###
    # create embedding layer
    embed_layer = tl.Embedding(
        vocab_size=vocab_size, # Size of the vocabulary
        d_feature=embedding_dim)  # Embedding dimension

    # Create a mean layer, to create an "average" word embedding
    mean_layer = tl.Mean(axis=1)

    # Create a dense layer, one unit for each output
    dense_output_layer = tl.Dense(n_units = output_dim)


    # Create the log softmax layer (no parameters needed)
    log_softmax_layer = tl.LogSoftmax()

    # Use tl.Serial to combine all layers
    # and create the classifier
    # of type trax.layers.combinators.Serial
    model = tl.Serial(
      embed_layer, # embedding layer
      mean_layer, # mean layer
      dense_output_layer, # dense output layer
      log_softmax_layer # log softmax layer
    )
### END CODE HERE ###

    # return the model of type
    return model
```

In [30]:

```
tmp_model = classifier()
```

In [31]:

```
print(type(tmp_model))
display(tmp_model)
```

```
<class 'trax.layers.combinators.Serial'>


Serial[
  Embedding_9088_256
  Mean
  Dense_2
  LogSoftmax
]
```

Expected Outout

```
<class 'trax.layers.combinators.Serial'>
Serial[
  Embedding_9088_256
  Mean
```

```
    Mean
    Dense_2
    LogSoftmax
  ]
```

Part 4: Training

To train a model on a task, Trax defines an abstraction `trax.supervised.training.TrainTask` which packages the train data, loss and optimizer (among other things) together into an object.

Similarly to evaluate a model, Trax defines an abstraction `trax.supervised.training.EvalTask` which packages the eval data and metrics (among other things) into another object.

The final piece tying things together is the `trax.supervised.training.Loop` abstraction that is a very simple and flexible way to put everything together and train the model, all the while evaluating it and saving checkpoints. Using `Loop` will save you a lot of code compared to always writing the training loop by hand, like you did in courses 1 and 2. More importantly, you are less likely to have a bug in that code that would ruin your training.

In [32]:

```
# View documentation for trax.supervised.training.TrainTask
help(trax.supervised.training.TrainTask)
```

```
Help on class TrainTask in module trax.supervised.training:

class TrainTask(builtins.object)
 |  TrainTask(labeled_data, loss_layer, optimizer, lr_schedule=None, n_steps_per_checkpoint=100)
 |
 |  A supervised task (labeled data + feedback mechanism) for training.
 |
 |  Methods defined here:
 |
 |  __init__(self, labeled_data, loss_layer, optimizer, lr_schedule=None,
n_steps_per_checkpoint=100)
 |      Configures a training task.
 |
 |      Args:
 |        labeled_data: Iterator of batches of labeled data tuples. Each tuple has
 |            1+ data (input value) tensors followed by 1 label (target value)
 |            tensor.  All tensors are NumPy ndarrays or their JAX counterparts.
 |        loss_layer: Layer that computes a scalar value (the "loss") by comparing
 |            model output :math:`\hat{y}=f(x)` to the target :math:`y`.
 |        optimizer: Optimizer object that computes model weight updates from
 |            loss-function gradients.
 |        lr_schedule: Learning rate schedule, a function step -> learning_rate.
 |        n_steps_per_checkpoint: How many steps to run between checkpoints.
 |
 |  learning_rate(self, step)
 |      Return the learning rate for the given step.
 |
 |  next_batch(self)
 |      Returns one batch of labeled data: a tuple of input(s) plus label.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  labeled_data
 |
 |  loss_layer
 |
 |  n_steps_per_checkpoint
 |
 |  optimizer
 |
 |  sample_batch
```

In [33]:

```
# View documentation for trax.supervised.training.EvalTask
help(trax.supervised.training.EvalTask)
```

```
Help on class EvalTask in module trax.supervised.training:

class EvalTask(builtins.object)
 |  EvalTask(labeled_data, metrics, metric_names=None, n_eval_batches=1)
 |  
 |  Labeled data plus scalar functions for (periodically) measuring a model.
 |  
 |  An eval task specifies how (`labeled_data` + `metrics`) and with what
 |  precision (`n_eval_batches`) to measure a model as it is training.
 |  The variance of each scalar output is reduced by measuring over multiple
 |  (`n_eval_batches`) batches and reporting the average from those measurements.
 |  
 |  Methods defined here:
 |  
 |  __init__(self, labeled_data, metrics, metric_names=None, n_eval_batches=1)
 |      Configures an eval task: named metrics run with a given data source.
 |      
 |      Args:
 |        labeled_data: Iterator of batches of labeled data tuples. Each tuple has
 |            1+ data tensors (NumPy ndarrays) followed by 1 label (target value)
 |            tensor.
 |        metrics: List of layers; each computes a scalar value per batch by
 |            comparing model output :math:`\hat{y}=f(x)` to the target :math:`y`.
 |        metric_names: List of names, one for each item in `metrics`, in matching
 |             order, to be used when recording/reporting eval output. If None,
 |             generate default names using layer names from metrics.
 |        n_eval_batches: Integer N that specifies how many eval batches to run;
 |            the output is then the average of the outputs from the N batches.
 |  
 |  next_batch(self)
 |      Returns one batch of labeled data: a tuple of input(s) plus label.
 |  
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |  
 |  __dict__
 |      dictionary for instance variables (if defined)
 |  
 |  __weakref__
 |      list of weak references to the object (if defined)
 |  
 |  labeled_data
 |  
 |  metric_names
 |  
 |  metrics
 |  
 |  n_eval_batches
 |  
 |  sample_batch
```

In [34]:

```
# View documentation for trax.supervised.training.Loop
help(trax.supervised.training.Loop)
```

```
Help on class Loop in module trax.supervised.training:

class Loop(builtins.object)
 |  Loop(model, task, eval_model=None, eval_task=None, output_dir=None, checkpoint_at=None,
eval_at=None)
 |  
 |  Loop that can run for a given number of steps to train a supervised model.
 |  
 |  The typical supervised training process randomly initializes a model and
 |  updates its weights via feedback (loss-derived gradients) from a training
```

```
 |  task, by looping through batches of labeled data. A training loop can also
 |  be configured to run periodic evals and save intermediate checkpoints.
 |
 |  For speed, the implementation takes advantage of JAX's composable function
 |  transformations (specifically, `jit` and `grad`). It creates JIT-compiled
 |  pure functions derived from variants of the core model; schematically:
 |
 |    - training variant: jit(grad(pure_function(model+loss)))
 |    - evals variant: jit(pure_function(model+evals))
 |
 |  In training or during evals, these variants are called with explicit
 |  arguments for all relevant input data, model weights/state, optimizer slots,
 |  and random number seeds:
 |
 |    - batch: labeled data
 |    - model weights/state: trainable weights and input-related state (e.g., as
 |      used by batch norm)
 |    - optimizer slots: weights in the optimizer that evolve during the training
 |      process
 |    - random number seeds: JAX PRNG keys that enable high-quality, distributed,
 |      repeatable generation of pseudo-random numbers
 |
 |  Methods defined here:
 |
 |  __init__(self, model, task, eval_model=None, eval_task=None, output_dir=None,
checkpoint_at=None, eval_at=None)
 |      Configures a training `Loop`, including a random initialization.
 |
 |      Args:
 |        model: Trax layer, representing the core model to be trained. Loss
 |            functions and eval functions (a.k.a. metrics) are considered to be
 |            outside the core model, taking core model output and data labels as
 |            their two inputs.
 |        task: TrainTask instance, which defines the training data, loss function,
 |            and optimizer to be used in this training loop.
 |        eval_model: Optional Trax layer, representing model used for evaluation,
 |          e.g., with dropout turned off. If None, the training model (model)
 |          will be used.
 |        eval_task: EvalTask instance or None. If None, don't do any evals.
 |        output_dir: Path telling where to save outputs (evals and checkpoints).
 |            Can be None if both `eval_task` and `checkpoint_at` are None.
 |        checkpoint_at: Function (integer --> boolean) telling, for step n, whether
 |            that step should have its checkpoint saved. If None, the default is
 |            periodic checkpointing at `task.n_steps_per_checkpoint`.
 |        eval_at: Function (integer --> boolean) that says, for training step n,
 |            whether that step should run evals. If None, run when checkpointing.
 |
 |  new_rng(self)
 |      Returns a new single-use random number generator (JAX PRNG key).
 |
 |  run(self, n_steps=1)
 |      Runs this training loop for n steps.
 |
 |      Optionally runs evals and saves checkpoints at specified points.
 |
 |      Args:
 |        n_steps: Stop training after completing n steps.
 |
 |  run_evals(self, weights=None, state=None)
 |      Runs and records evals for this training session.
 |
 |      Args:
 |        weights: Current weights from model in training.
 |        state: Current state from model in training.
 |
 |  save_checkpoint(self, weights=None, state=None, slots=None)
 |      Saves checkpoint to disk for the current training step.
 |
 |      Args:
 |        weights: Weights from model being trained.
 |        state: State (non-weight parameters) from model being trained.
 |        slots: Updatable weights for the optimizer in this training loop.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |    dict
```

```
 |      dictionary for instance variables (if defined)
 |  
 |  __weakref__
 |      list of weak references to the object (if defined)
 |  
 |  current_step
 |      Returns current step number in this training session.
 |  
 |  eval_model
 |      Returns the model used for evaluation.
 |  
 |  model
 |      Returns the model that is training.
```

In [35]:

```
# View optimizers that you could choose from
help(trax.optimizers)
```

```
Help on package trax.optimizers in trax:

NAME
    trax.optimizers - Optimizers for use with Trax layers.

PACKAGE CONTENTS
    adafactor
    adam
    base
    momentum
    optimizers_test
    rms_prop
    sm3

FUNCTIONS
    opt_configure(*args, **kwargs)

FILE
    /opt/conda/lib/python3.7/site-packages/trax/optimizers/__init__.py
```

Notice some available optimizers include:

```
adafactor
    adam
    momentum
    rms_prop
    sm3
```

4.1 Training the model

Now you are going to train your model.

Let's define the `TrainTask`, `EvalTask` and `Loop` in preparation to train the model.

In [36]:

```
from trax.supervised import training

batch_size = 16
rnd.seed(271)

train_task = training.TrainTask(
    labeled_data=train_generator(batch_size=batch_size, shuffle=True),
    loss_layer=tl.CrossEntropyLoss(),
    optimizer=trax.optimizers.Adam(0.01),
    n_steps_per_checkpoint=10,
)
```

```
eval_task = training.EvalTask(
    labeled_data=val_generator(batch_size=batch_size, shuffle=True),
    metrics=[tl.CrossEntropyLoss(), tl.Accuracy()],
)

model = classifier()
```

This defines a model trained using `tl.CrossEntropyLoss` optimized with the `trax.optimizers.Adam` optimizer, all the while tracking the accuracy using `tl.Accuracy` metric. We also track `tl.CrossEntropyLoss` on the validation set.

Now let's make an output directory and train the model.

In [37]:

```
output_dir = '~/model/'
output_dir_expand = os.path.expanduser(output_dir)
print(output_dir_expand)
```

```
/home/jovyan/model/
```

Exercise 06

Instructions: Implement `train_model` to train the model (`classifier` that you wrote earlier) for the given number of training steps (`n_steps`) using `TrainTask` , `EvalTask` and `Loop` .

In [38]:

```
# UNQ_C6 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: train_model
def train_model(classifier, train_task, eval_task, n_steps, output_dir):
    '''
    Input:
        classifier - the model you are building
        train_task - Training task
        eval_task - Evaluation task
        n_steps - the evaluation steps
        output_dir - folder to save your files
    Output:
        trainer -  trax trainer
    '''
### START CODE HERE (Replace instances of 'None' with your code) ###
    training_loop = training.Loop(
                                classifier, # The learning model
                                train_task, # The training task
                                eval_task = eval_task, # The evaluation task
                                output_dir = output_dir) # The output directory

    training_loop.run(n_steps = n_steps)
### END CODE HERE ###

    # Return the training_loop, since it has the model.
    return training_loop
```

In [39]:

```
training_loop = train_model(model, train_task, eval_task, 100, output_dir_expand)
```

```
Step      1: train CrossEntropyLoss |  0.88939196
Step      1: eval  CrossEntropyLoss |  0.68833977
Step      1: eval          Accuracy |  0.50000000
Step     10: train CrossEntropyLoss |  0.61036736
Step     10: eval  CrossEntropyLoss |  0.52182281
Step     10: eval          Accuracy |  0.68750000
Step     20: train CrossEntropyLoss |  0.34137666
Step     20: eval  CrossEntropyLoss |  0.20654774
Step     20: eval          Accuracy |  1.00000000
Step     30: train CrossEntropyLoss |  0.20208922
Step     30: eval  CrossEntropyLoss |  0.21594886
```

```
Step     30: eval  CrossEntropyLoss |  0.21594886
Step     30: eval          Accuracy |  0.93750000
Step     40: train CrossEntropyLoss |  0.19611198
Step     40: eval  CrossEntropyLoss |  0.17582777
Step     40: eval          Accuracy |  1.00000000
Step     50: train CrossEntropyLoss |  0.11203773
Step     50: eval  CrossEntropyLoss |  0.07589275
Step     50: eval          Accuracy |  1.00000000
Step     60: train CrossEntropyLoss |  0.09375446
Step     60: eval  CrossEntropyLoss |  0.09290724
Step     60: eval          Accuracy |  1.00000000
Step     70: train CrossEntropyLoss |  0.08785903
Step     70: eval  CrossEntropyLoss |  0.09610598
Step     70: eval          Accuracy |  1.00000000
Step     80: train CrossEntropyLoss |  0.08858261
Step     80: eval  CrossEntropyLoss |  0.02319432
Step     80: eval          Accuracy |  1.00000000
Step     90: train CrossEntropyLoss |  0.05699894
Step     90: eval  CrossEntropyLoss |  0.01778970
Step     90: eval          Accuracy |  1.00000000
Step    100: train CrossEntropyLoss |  0.03663783
Step    100: eval  CrossEntropyLoss |  0.00210550
Step    100: eval          Accuracy |  1.00000000
```

Expected output (Approximately)

```
Step      1: train CrossEntropyLoss |  0.88939196
Step      1: eval  CrossEntropyLoss |  0.68833977
Step      1: eval          Accuracy |  0.50000000
Step     10: train CrossEntropyLoss |  0.61036736
Step     10: eval  CrossEntropyLoss |  0.52182281
Step     10: eval          Accuracy |  0.68750000
Step     20: train CrossEntropyLoss |  0.34137666
Step     20: eval  CrossEntropyLoss |  0.20654774
Step     20: eval          Accuracy |  1.00000000
Step     30: train CrossEntropyLoss |  0.20208922
Step     30: eval  CrossEntropyLoss |  0.21594886
Step     30: eval          Accuracy |  0.93750000
Step     40: train CrossEntropyLoss |  0.19611198
Step     40: eval  CrossEntropyLoss |  0.17582777
Step     40: eval          Accuracy |  1.00000000
Step     50: train CrossEntropyLoss |  0.11203773
Step     50: eval  CrossEntropyLoss |  0.07589275
Step     50: eval          Accuracy |  1.00000000
Step     60: train CrossEntropyLoss |  0.09375446
Step     60: eval  CrossEntropyLoss |  0.09290724
Step     60: eval          Accuracy |  1.00000000
Step     70: train CrossEntropyLoss |  0.08785903
Step     70: eval  CrossEntropyLoss |  0.09610598
Step     70: eval          Accuracy |  1.00000000
Step     80: train CrossEntropyLoss |  0.08858261
Step     80: eval  CrossEntropyLoss |  0.02319432
Step     80: eval          Accuracy |  1.00000000
Step     90: train CrossEntropyLoss |  0.05699894
Step     90: eval  CrossEntropyLoss |  0.01778970
Step     90: eval          Accuracy |  1.00000000
Step    100: train CrossEntropyLoss |  0.03663783
Step    100: eval  CrossEntropyLoss |  0.00210550
Step    100: eval          Accuracy |  1.00000000
```

4.2 Practice Making a prediction

Now that you have trained a model, you can access it as `training_loop.model` object. We will actually use `training_loop.eval_model` and in the next weeks you will learn why we sometimes use a different model for evaluation, e.g., one without dropout. For now, make predictions with your model.

Use the training data just to see how the prediction process works.

- Later, you will use validation data to evaluate your model's performance.

In [40]:

```
# Create a generator object
tmp_train_generator = train_generator(16)

# get one batch
tmp_batch = next(tmp_train_generator)

# Position 0 has the model inputs (tweets as tensors)
# position 1 has the targets (the actual labels)
tmp_inputs, tmp_targets, tmp_example_weights = tmp_batch

print(f"The batch is a tuple of length {len(tmp_batch)} because position 0 contains the tweets, an
d position 1 contains the targets.")
print(f"The shape of the tweet tensors is {tmp_inputs.shape} (num of examples, length of tweet ten
sors)")
print(f"The shape of the labels is {tmp_targets.shape}, which is the batch size.")
print(f"The shape of the example_weights is {tmp_example_weights.shape}, which is the same as inpu
ts/targets size.")
```

```
The batch is a tuple of length 3 because position 0 contains the tweets, and position 1 contains t
he targets.
The shape of the tweet tensors is (16, 15) (num of examples, length of tweet tensors)
The shape of the labels is (16,), which is the batch size.
The shape of the example_weights is (16,), which is the same as inputs/targets size.
```

In [41]:

```
# feed the tweet tensors into the model to get a prediction
tmp_pred = training_loop.eval_model(tmp_inputs)
print(f"The prediction shape is {tmp_pred.shape}, num of tensor_tweets as rows")
print("Column 0 is the probability of a negative sentiment (class 0)")
print("Column 1 is the probability of a positive sentiment (class 1)")
print()
print("View the prediction array")
tmp_pred
```

```
The prediction shape is (16, 2), num of tensor_tweets as rows
Column 0 is the probability of a negative sentiment (class 0)
Column 1 is the probability of a positive sentiment (class 1)

View the prediction array
```

Out[41]:

```
DeviceArray([[-4.9417334e+00, -7.1678162e-03],
             [-6.5846415e+00, -1.3823509e-03],
             [-5.4463043e+00, -4.3215752e-03],
             [-4.3487482e+00, -1.3007164e-02],
             [-4.9131694e+00, -7.3764324e-03],
             [-4.7097692e+00, -9.0477467e-03],
             [-5.2801600e+00, -5.1045418e-03],
             [-4.1103225e+00, -1.6538620e-02],
             [-1.8327236e-03, -6.3028107e+00],
             [-4.7376156e-03, -5.3545618e+00],
             [-3.4697056e-03, -5.6654320e+00],
             [-1.1444092e-05, -1.1379558e+01],
             [-1.0051131e-02, -4.6050973e+00],
             [-1.0130405e-03, -6.8951964e+00],
             [-6.1047077e-03, -5.1017356e+00],
             [-7.4422359e-03, -4.9043016e+00]], dtype=float32)
```

To turn these probabilities into categories (negative or positive sentiment prediction), for each row:

- Compare the probabilities in each column.
- If column 1 has a value greater than column 0, classify that as a positive tweet.
- Otherwise if column 1 is less than or equal to column 0, classify that example as a negative tweet.

In [42]:

```
# turn probabilites into category predictions
tmp_is_positive = tmp_pred[:,1] > tmp_pred[:,0]
for i, p in enumerate(tmp_is_positive):
    print(f"Neg log prob {tmp_pred[i,0]:.4f}\tPos log prob {tmp_pred[i,1]:.4f}\t is positive? {p}\
t actual {tmp_targets[i]}")
```

```
Neg log prob -4.9417 Pos log prob -0.0072  is positive? True  actual 1
Neg log prob -6.5846 Pos log prob -0.0014  is positive? True  actual 1
Neg log prob -5.4463 Pos log prob -0.0043  is positive? True  actual 1
Neg log prob -4.3487 Pos log prob -0.0130  is positive? True  actual 1
Neg log prob -4.9132 Pos log prob -0.0074  is positive? True  actual 1
Neg log prob -4.7098 Pos log prob -0.0090  is positive? True  actual 1
Neg log prob -5.2802 Pos log prob -0.0051  is positive? True  actual 1
Neg log prob -4.1103 Pos log prob -0.0165  is positive? True  actual 1
Neg log prob -0.0018 Pos log prob -6.3028  is positive? False  actual 0
Neg log prob -0.0047 Pos log prob -5.3546  is positive? False  actual 0
Neg log prob -0.0035 Pos log prob -5.6654  is positive? False  actual 0
Neg log prob -0.0000 Pos log prob -11.3796  is positive? False  actual 0
Neg log prob -0.0101 Pos log prob -4.6051  is positive? False  actual 0
Neg log prob -0.0010 Pos log prob -6.8952  is positive? False  actual 0
Neg log prob -0.0061 Pos log prob -5.1017  is positive? False  actual 0
Neg log prob -0.0074 Pos log prob -4.9043  is positive? False  actual 0
```

Notice that since you are making a prediction using a training batch, it's more likely that the model's predictions match the actual targets (labels).

- Every prediction that the tweet is positive is also matching the actual target of 1 (positive sentiment).
- Similarly, all predictions that the sentiment is not positive matches the actual target of 0 (negative sentiment)

One more useful thing to know is how to compare if the prediction is matching the actual target (label).

- The result of calculation `is_positive` is a boolean.
- The target is a type trax.fastmath.numpy.int32
- If you expect to be doing division, you may prefer to work with decimal numbers with the data type type trax.fastmath.numpy.int32

In [43]:

```
# View the array of booleans
print("Array of booleans")
display(tmp_is_positive)

# convert boolean to type int32
# True is converted to 1
# False is converted to 0
tmp_is_positive_int = tmp_is_positive.astype(np.int32)


# View the array of integers
print("Array of integers")
display(tmp_is_positive_int)

# convert boolean to type float32
tmp_is_positive_float = tmp_is_positive.astype(np.float32)

# View the array of floats
print("Array of floats")
display(tmp_is_positive_float)
```

```
Array of booleans


DeviceArray([ True,  True,  True,  True,  True,  True,  True,  True,
             False, False, False, False, False, False, False, False],             dtype=bool)


Array of integers


DeviceArray([1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int32)
```

```
DeviceArray([1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int32)
```

```
Array of floats
```

```
DeviceArray([1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0.,
             0.], dtype=float32)
```

In [44]:

```
tmp_pred.shape
```

Out[44]:

```
(16, 2)
```

Note that Python usually does type conversion for you when you compare a boolean to an integer

- True compared to 1 is True, otherwise any other integer is False.
- False compared to 0 is True, otherwise any ohter integer is False.

In [45]:

```
print(f"True == 1: {True == 1}")
print(f"True == 2: {True == 2}")
print(f"False == 0: {False == 0}")
print(f"False == 2: {False == 2}")
```

```
True == 1: True
True == 2: False
False == 0: True
False == 2: False
```

However, we recommend that you keep track of the data type of your variables to avoid unexpected outcomes. So it helps to convert the booleans into integers

- Compare 1 to 1 rather than comparing True to 1.

Hopefully you are now familiar with what kinds of inputs and outputs the model uses when making a prediction.

- This will help you implement a function that estimates the accuracy of the model's predictions.

Part 5: Evaluation

5.1 Computing the accuracy on a batch

You will now write a function that evaluates your model on the validation set and returns the accuracy.

- `preds` contains the predictions.
  - Its dimensions are `(batch_size, output_dim)`. `output_dim` is two in this case. Column 0 contains the probability that the tweet belongs to class 0 (negative sentiment). Column 1 contains probability that it belongs to class 1 (positive sentiment).
  - If the probability in column 1 is greater than the probability in column 0, then interpret this as the model's prediction that the example has label 1 (positive sentiment).
  - Otherwise, if the probabilities are equal or the probability in column 0 is higher, the model's prediction is 0 (negative sentiment).
- `y` contains the actual labels.
- `y_weights` contains the weights to give to predictions.

Exercise 07

Implement `compute_accuracy`.

In [46]:

```
# UNQ_C7 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: compute_accuracy
def compute_accuracy(preds, y, y_weights):
    """
    Input:
        preds: a tensor of shape (dim_batch, output_dim)
        y: a tensor of shape (dim_batch, output_dim) with the true labels
        y_weights: a n.ndarray with the a weight for each example
    Output:
        accuracy: a float between 0-1
        weighted_num_correct (np.float32): Sum of the weighted correct predictions
        sum_weights (np.float32): Sum of the weights
    """
    ### START CODE HERE (Replace instances of 'None' with your code) ###
    # Create an array of booleans,
    # True if the probability of positive sentiment is greater than
    # the probability of negative sentiment
    # else False
    is_pos =  preds[:, 1] > preds[:, 0]

    # convert the array of booleans into an array of np.int32
    is_pos_int = is_pos.astype(np.int32)

    # compare the array of predictions (as int32) with the target (labels) of type int32
    correct = is_pos_int == y

    # Count the sum of the weights.
    sum_weights = np.sum(y_weights)

    # convert the array of correct predictions (boolean) into an arrayof np.float32
    correct_float = correct.astype(np.float32)

    # Multiply each prediction with its corresponding weight.
    weighted_correct_float = correct_float * y_weights

    # Sum up the weighted correct predictions (of type np.float32), to go in the
    # denominator.
    weighted_num_correct = np.sum(weighted_correct_float)

    # Divide the number of weighted correct predictions by the sum of the
    # weights.
    accuracy = weighted_num_correct / sum_weights

    ### END CODE HERE ###
    return accuracy, weighted_num_correct, sum_weights
```

In [47]:

```
# test your function
tmp_val_generator = val_generator(64)

# get one batch
tmp_batch = next(tmp_val_generator)

# Position 0 has the model inputs (tweets as tensors)
# position 1 has the targets (the actual labels)
tmp_inputs, tmp_targets, tmp_example_weights = tmp_batch

# feed the tweet tensors into the model to get a prediction
tmp_pred = training_loop.eval_model(tmp_inputs)

tmp_acc, tmp_num_correct, tmp_num_predictions = compute_accuracy(preds=tmp_pred, y=tmp_targets, y_w
eights=tmp_example_weights)

print(f"Model's prediction accuracy on a single training batch is: {100 * tmp_acc}%")
print(f"Weighted number of correct predictions {tmp_num_correct}; weighted number of total observa
tions predicted {tmp_num_predictions}")
```

```
Model's prediction accuracy on a single training batch is: 100.0%
Weighted number of correct predictions 64.0; weighted number of total observations predicted 64
```

Expected output (Approximately)

```
Model's prediction accuracy on a single training batch is: 100.0%
Weighted number of correct predictions 64.0; weighted number of total observations
predicted 64
```

5.2 Testing your model on Validation Data

Now you will write test your model's prediction accuracy on validation data.

This program will take in a data generator and your model.

- The generator allows you to get batches of data. You can use it with a `for` loop:

```
for batch in iterator:
   # do something with that batch
```

`batch` has dimensions `(X, Y, weights)`.

- Column 0 corresponds to the tweet as a tensor (input).
- Column 1 corresponds to its target (actual label, positive or negative sentiment).
- Column 2 corresponds to the weights associated (example weights)
- You can feed the tweet into model and it will return the predictions for the batch.

Exercise 08

Instructions:

- Compute the accuracy over all the batches in the validation iterator.
- Make use of `compute_accuracy`, which you recently implemented, and return the overall accuracy.

In [48]:

```
# UNQ_C8 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: test_model
def test_model(generator, model):
    '''
    Input:
        generator: an iterator instance that provides batches of inputs and targets
        model: a model instance
    Output:
        accuracy: float corresponding to the accuracy
    '''

    accuracy = 0.
    total_num_correct = 0
    total_num_pred = 0

    ### START CODE HERE (Replace instances of 'None' with your code) ###
    for batch in generator:

        # Retrieve the inputs from the batch
        inputs = batch[0]

        # Retrieve the targets (actual labels) from the batch
        targets = batch[1]

        # Retrieve the example weight.
        example_weight = batch[2]

        # Make predictions using the inputs
        pred = model(inputs)

        # Calculate accuracy for the batch by comparing its predictions and targets
        batch_accuracy, batch_num_correct, batch_num_pred = compute_accuracy(pred, targets, example
```

```
_weight)

        # Update the total number of correct predictions
        # by adding the number of correct predictions from this batch
        total_num_correct += batch_num_correct

        # Update the total number of predictions
        # by adding the number of predictions made for the batch
        total_num_pred += batch_num_pred

    # Calculate accuracy over all examples
    accuracy = total_num_correct / total_num_pred

    ### END CODE HERE ###
    return accuracy
```

In [49]:

```
# DO NOT EDIT THIS CELL
# testing the accuracy of your model: this takes around 20 seconds
model = training_loop.eval_model
accuracy = test_model(test_generator(16), model)

print(f'The accuracy of your model on the validation set is {accuracy:.4f}', )
```

```
The accuracy of your model on the validation set is 0.9931
```

Expected Output (Approximately)

```
The accuracy of your model on the validation set is 0.9931
```

Part 6: Testing with your own input

Finally you will test with your own input. You will see that deepnets are more powerful than the older methods you have used before. Although you go close to 100% accuracy on the first two assignments, the task was way easier.

In [50]:

```
# this is used to predict on your own sentnece
def predict(sentence):
    inputs = np.array(tweet_to_tensor(sentence, vocab_dict=Vocab))

    # Batch size 1, add dimension for batch, to work with the model
    inputs = inputs[None, :]

    # predict with the model
    preds_probs = model(inputs)

    # Turn probabilities into categories
    preds = int(preds_probs[0, 1] > preds_probs[0, 0])

    sentiment = "negative"
    if preds == 1:
        sentiment = 'positive'

    return preds, sentiment
```

In [51]:

```
# try a positive sentence
sentence = "It's such a nice day, think i'll be taking Sid to Ramsgate fish and chips for lunch at
Peter's fish factory and then the beach maybe"
tmp_pred, tmp_sentiment = predict(sentence)
print(f"The sentiment of the sentence \n***\n\"{sentence}\"\n***\nis {tmp_sentiment}.")

print()
# try a negative sentence
sentence = "I hated my day, it was the worst, I'm so sad."
```

```
tmp_pred, tmp_sentiment = predict(sentence)
print(f"The sentiment of the sentence \n***\n\"{sentence}\"\n***\nis {tmp_sentiment}.")
```

```
The sentiment of the sentence
***
"It's such a nice day, think i'll be taking Sid to Ramsgate fish and chips for lunch at Peter's fi
sh factory and then the beach maybe"
***
is positive.

The sentiment of the sentence
***
"I hated my day, it was the worst, I'm so sad."
***
is negative.
```

Notice that the model works well even for complex sentences.

On Deep Nets

Deep nets allow you to understand and capture dependencies that you would have not been able to capture with a simple linear regression, or logistic regression.

- It also allows you to better use pre-trained embeddings for classification and tends to generalize better.