Sentiment Classification DNNs

August 6, 2020

Sentiment with Deep Neural Networks

0.1 Outline

By completing this, 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

May you've 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
- 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

In [ ]:

```
# Part 1: Import libraries and try out Trax

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

In [184]:

```
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 #to make
compatible with JAX

# 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
```


1 Create an array using trax.fastmath.numpy

a = np.array(5.0)

2 View the returned array

display(a)

print(type(a)) Notice that trax.fastmath.numpy returns a DeviceArray from the jax library.

In [186]:
```python
# Define a function that will use the trax.fastmath.numpy array
def f(x):

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

In [187]:
```python
# 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 [188]:
```python
# 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[188]: function

In [189]:
```python
# 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

P**art 2: Importing the data ## 2.1 Loading in the data Import the** data - Details of process_tweet function are available in utils.py file

In [190]:
```
## 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 se
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 [191]:
```
# 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 mem
community this Tweet at training position 0 after processing:

Out[191]: ['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 [ ]: # 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 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)
```

The dictionary Vocab will look like this:

5

{'__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 represent- ing 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 Map each word in tweet to corresponding token in ‘Vocab’ Use Python’s Dictionary.get(key,value) so that the function returns a default value if the key is not found in the dictionary.

In [193]:
```
# 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 = 2
```

```
        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.get(word, unk_ID) ### END CODE HERE ###

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

        return tensor_l
```

In [194]:
```
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

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

In [195]:
```
# 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 d
    check if you included mapping for unknown } ]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":
```

8

```
            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 [196]: *# 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**):

import numpy as np *''' 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

                              1
                              0
    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


                                1
                                1
                neg_index = 0

                if shuffle:
                    rnd.shuffle(neg_index_lines) # Shuffle the index of the
                    negative sample

            # 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(q) for q 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 lo n_pad = max_l
len(tensor)
```

1
2

```
# Generate a list of zeros, with length n_pad pad_l = [0
for i in range(n_pad)] #print(pad_l)
```

```
            # 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 for i in
        range(n_to_take)] # Generate the list of targets for the negative examples (a list of zeros) # The
        length i the number of negative examples in the batch target_neg = [0 for i in range(n_to_take)]
        # Concatenate the positve and negative targets target_l =
        np.concatenate((target_pos,target_neg)) #print(target_l)

        # 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 example_w
        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 [197]:
```
# 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)

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 [198]: # 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}") 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) 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 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 input 858 256 3652 5739 307 4458 567 1230 2767 328 1202 3761 0 0]; target 0; example

Expected output

The inputs shape is (4, 14) 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

1
4
input tensor: [5738 2901 3761 0 0 0 0 0 0 0 0 0 0 0]; target 0; example input tensor: [858 256 3652 5739 307 4458 567 123 1202 3761 0 0]; target 0; example

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

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:

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

1

5

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 Please use numpy.maximum(A,k) to find the maximum between each element in A and a scalar k

In [199]: *# 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) ### #for j in range(len(x)):
 #for i in range(len(x[j])): # x[j][i]=np.max(x[j][i] , 0)
        x[x<=0]=0 activation = x

        ### END CODE HERE
        ###

        return activation
```

In [200]: *# 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

2.0.1 Exercise

Implement the forward function of the Dense class. - The forward function multiplies the input to the layer (x) by the weight matrix (W)

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

- 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 in- cludes 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 stan- dard 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 [201]:
```
# 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 [202]:
```
# 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 tenso
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 [203]:
```
# 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 shape = (input_shape[-1], self._n_units)

        # Generate the weight matrix from a normal distribution, # and standard
        deviation of 'stdev'  w = trax.fastmath.random.normal(random_key,
        shape) w=np.dot(w, self._init_stdev)


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

In [204]: # 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 we
print("Foward function output is ", dense_layer(z)) # Returns multiplied values of units and w
```

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 [205]:
```
# View documentation on tl.Dense
#help(tl.Dense)
```

In [206]:
```
# View documentation on tl.Serial
#help(tl.Serial)
```

- 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). – Recall from the
previous course 2, week 4, that the embedding is

In [207]: # View documentation for tl.Embedding

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

2

0

In [208]:
```
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 [209]:
```
# 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 [210]:
```
# 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
display(np.mean(tmp_embed,axis=1))
```

2
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 embedding

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

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

In [211]: 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 [212]:
```
# 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) ###

2
2
    # 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, keepdims=False)

    # 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(
```

tl.Embedding(vocab_size, embedding_dim), *# embedding layer* tl.Mean(axis=1), *# mean layer* tl.Dense(output_dim), *# dense output layer* tl.LogSoftmax() *# log softmax layer*) *### END CODE HERE ###*

 # return the model of type **return**
model

In [213]: tmp_model = classifier()

In [214]: 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[

2
3

Embedding_9088_256
Mean Dense_2
LogSoftmax]

Part 4: Training

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 [215]:
```
# 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 (lab

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 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 outp :math:`\hat{y}=f(x)` to the target :math:`y`. | optimizer: Optimizer object that computes model weight updates from | loss-gradients. | lr_schedule: Learning rate schedule, a function step -> learning_rate. | n_steps_per_checkpoint: How many step

between checkpoints. || learning_rate(self, step) | Return the learning rate for the given step. || next_batch(self) | Returns on of labeled data: a tuple of input(s) plus label.

2
4

|| -- | 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 [216]:
```
# 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 [217]: *# 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 tha

for a given number of steps to train a supervised model. || The typical supervised training process randomly initializes a mo updates its weights via feedback (loss-derived gradients) from a training | task, by looping through batches of labeled data. loop can also | be configured to run periodic evals and save intermediate checkpoints. || For speed, the implementation take of JAX's composable function | transformations (specifically, `jit` and `grad`). It creates JIT-compiled | pure functions deri variants of the core model; schematically:

|| - training variant: jit(grad(pure_function(model+loss))) | - evals variant: jit(pure_function(model+evals)) || In training or

evals, these variants are called with explicit | arguments for all relevant input data, model weights/state, optimizer slots, | a

number seeds: || - batch: labeled data | - model weights/state: trainable weights and input-related state (e.g., as | used by ba
optimizer slots: weights in the optimizer that evolve during the training | process | - random number seeds: JAX PRNG ke

enable high-quality, distributed, | repeatable generation of pseudo-random numbers || Methods defined here: || __init__(sel

task, eval_model=None, eval_task=None, output_dir=None, checkpoint_at=None, ev | Configures a training `Loop`, inclu

random initialization. || Args: | model: Trax layer, representing the core model to be trained. Loss | functions and eval func
metrics) are considered to be | outside the core model, taking core model output and data labels as | their two inputs. | task:
instance, which defines the training data, loss function, | and optimizer to be used in this training loop. | eval_model: Optio
layer, representing model used for evaluation, | e.g., with dropout turned off. If None, the training model (model) | will be
eval_task: EvalTask instance or None. If None, don't do any evals. | output_dir: Path telling where to save outputs (evals a
checkpoints). | Can be None if both `eval_task` and `checkpoint_at` are None. | checkpoint_at: Function (integer --> boole
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 sh

evals. If None, run when checkpointing. || new_rng(self) | Returns a new single-use random number generator (JAX PRN

run(self, n_steps=1) | Runs this training loop for n steps. || Optionally runs evals and saves checkpoints at specified points.

2
7

| 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 [218]: *# 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

FUNCTIO
NS
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 [219]: **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.

2
9

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

In [220]: 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 [221]: # 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,
                                        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 [222]: training_loop = train_model(model, train_task, eval_task, 100, output_dir_expand)

Step 1: train CrossEntropyLoss | 1.12272775 Step 1: eval CrossEntropyLoss | 0.89718306 Step 1: eval Accuracy | 0.31250000 Step 10: train CrossEntropyLoss | 0.81427336 Step 10: eval CrossEntropyLoss | 0.50847912 Step 10: eval Accuracy | 0.75000000 Step 20: train CrossEntropyLoss | 0.52660501 Step 20: eval CrossEntropyLoss | 0.34810668

Step 20: eval Accuracy | 0.87500000 Step 30: train CrossEntropyLoss | 0.32775387 Step 30: eval CrossEntropyLoss | 0.30329353 Step 30: eval Accuracy | 0.87500000 Step 40: train CrossEntropyLoss | 0.29091102 Step 40: eval CrossEntropyLoss | 0.30724931 Step 40: eval Accuracy | 0.87500000 Step 50: train CrossEntropyLoss | 0.17655556 Step 50: eval CrossEntropyLoss | 0.13073292 Step 50: eval Accuracy | 1.00000000 Step 60: train CrossEntropyLoss | 0.16591090 Step 60: eval CrossEntropyLoss | 0.13529448 Step 60: eval Accuracy | 1.00000000 Step 70: train CrossEntropyLoss | 0.12349443 Step 70: eval CrossEntropyLoss | 0.07977644 Step 70: eval Accuracy | 1.00000000 Step 80: train CrossEntropyLoss | 0.11218882 Step 80: eval CrossEntropyLoss | 0.02898891 Step 80: eval Accuracy | 1.00000000 Step 90: train CrossEntropyLoss | 0.07728098 Step 90: eval CrossEntropyLoss | 0.02353959 Step 90: eval Accuracy | 1.00000000 Step 100: train CrossEntropyLoss | 0.06835373 Step 100: eval CrossEntropyLoss | 0.00595517 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 [223]: *# 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 print(f"The shap
tweet tensors is {tmp_inputs.shape} (num of examples, length of tweet 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

The batch is a tuple of length 3 because position 0 contains the tweets, and position 1 contains the tar The shape of the twee
(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 [224]: # 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)")

3
2

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[224]: DeviceArray([[-4.3611403e+00, -1.2845993e-02], [-4.1929121e+00, -1.5217304e-02], [-4.8559332e+00, -7.8125000e-03], [-3.6934922e+00, -2.5199771e-02], [-3.5967889e+00, -2.7794361e-02], [-3.5383053e+00, -2.9493213e-02], [-5.2808285e+00, -5.1012039e-03], [-5.4767599e+00, -4.1916370e-03], [-1.2308240e-02, -4.4036212e+00], [-5.5770874e-03, -5.1918826e+00], [-8.0459118e-03, -4.8266072e+00], [-7.2956085e-05, -9.5227098e+00], [-1.6406775e-02, -4.1182537e+00], [-2.8779507e-03, -5.8520947e+00],

[-1.1285543e-02, -4.4898720e+00], [-4.8916340e-03, -5.3226776e+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 [225]:
```
# 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?
```

Neg log prob -4.3611 Pos log prob -0.0128 is positive? True actual 1 Neg log prob -4.1929 Pos log prob -0.0152 is positive? True actual 1 Neg log prob -4.8559 Pos log prob -0.0078 is positive? True actual 1 Neg log prob -3.6935 Pos log prob -0.0252 is positive? True actual 1 Neg log prob -3.5968 Pos log prob -0.0278 is positive? True actual 1 Neg log prob -3.5383 Pos log prob -0.0295 is positive? True actual 1 Neg log prob -5.2808 Pos log prob -0.0051 is positive? True actual 1 Neg log prob -5.4768 Pos log prob -0.0042 is positive? True actual 1 Neg log prob -0.0123 Pos log prob -4.4036 is positive? False actual 0 Neg log prob -0.0056 Pos log prob -5.1919 is positive? False actual 0 Neg log prob -0.0080 Pos log prob -4.8266 is positive? False actual 0

Neg log prob -0.0001 Pos log prob -9.5227 is positive? False actual 0 Neg log prob -0.0164 Pos log prob -4.1183 is positive? False actual 0 Neg log prob -0.0029 Pos log prob -5.8521 is positive? False actual 0 Neg log prob -0.0113 Pos log prob -4.4899 is positive? False actual 0 Neg log prob -0.0049 Pos log prob -5.3227 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 senti- ment). - 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 [226]:
```
# 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

3
4

```
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 [227]: tmp_pred.shape

Out[227]: (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 [228]: 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 un- expected 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 pre- diction is 0 (negative sentiment). - y contains the actual labels. - y_weights contains the weights to give to predictions.

Exercise 07 Implement compute_accuracy.


```
In [229]: # 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 =
```

```
np.array(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 =
np.array(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 =
[np.multiply(correct_float[i],y_weights[i]) for i in range(len(co

# 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 [230]:
```
# test your function
tmp_val_generator = val_generator(64)
```

3
6

```
# 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

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

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 (batch size, 2). - Column 0 corresponds to the tweet as a tensor. - Column 1 corresponds to its target (actual label, positive or negative sentiment). - 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 [231]:
```
# 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

                        3
                        7
    ''
    '

    accuracy = 0.
```

```python
    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 =
        None

        # Retrieve the targets (actual labels) from the batch targets = None

        # Retrieve the example weight.
        example_weight = None

        # Make predictions using the inputs pred =
        None

        # Calculate accuracy for the batch by comparing its predictions and targets batch_accuracy,
        batch_num_correct, batch_num_pred = None

        # Update the total number of correct predictions # by adding the number of
        correct predictions from this batch total_num_correct += None

        # Update the total number of predictions # by adding the number of
        predictions made for the batch total_num_pred += None

    # Calculate accuracy over all examples accuracy
    = None

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

In [232]:
```python
# 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}', )
```

3
8
TypeError Traceback (most recent call last)

<ipython-input-232-5a6afde371c2> in <module>

2 # testing the accuracy of your model: this takes around 20 seconds 3 model = training_loop.eval_model ----> 4 accuracy = test_model(test_generator(16), model)

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

<ipython-input-231-cd65dad7d79d> in test_model(generator, model)

3031 # Calculate accuracy for the batch by comparing its predictions and targets ---> 32 batch_accuracy, batch_num_correct, batch_num_pred = None

3334 # Update the total number of correct predictions

TypeError: cannot unpack non-iterable NoneType object

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 [ ]:
```
# 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
```

3
9

```
In [ ]: # try a positive sentence
    sentence = "It's such a nice day, think i'll be taking Sid to Ramsgate fish and chips for lunch tmp_pred, tmp_senti
    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}.")
```

Notice that the model works well even for complex sentences.

2.0.2 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.