C3_W1_Assignment

October 31, 2020

1 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 - Section **??**

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.

In [ ]:
```
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
```

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

# View the returned array
display(a)

print(type(a))
```

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

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

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

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

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

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

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

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

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 [ ]:
```
## 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)}")
```

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 [ ]:
```
# 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])
```

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:

```
{'__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
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 [ ]: # 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):
            '''
```

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

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,
```

In [ ]:
```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
        }
    ]
    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["expe
                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()
```

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 [ ]: # 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):

        # 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 wo
            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_t
        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 pos_index
            tweet = data_neg[neg_index_lines[neg_index]]

            # convert the tweet into tensors of integers representing the processed wo
            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 ma
            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 ones)
            # 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.
            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 [ ]: # 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}')
```

```
In [ ]: # 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_we
```

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]; target
input tensor: [ 858  256 3652 5739  307 4458  567 1230 2767  328 1202 3761    0    0]; target
```

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:

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

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

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

```
In [ ]: # 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(x,0)

                ### END CODE HERE ###

                return activation
```

```
In [ ]: # 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))
```

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

1.0.1 Exercise

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

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

- 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 [ ]: # 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 [ ]: # 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 ten
        # 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)
```

Exercise 04
Implement the Dense class.

```
In [ ]: # 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]

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

```
In [ ]: # 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 uni
```

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  -2.58
  -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 [ ]: # View documentation on tl.Dense
        help(tl.Dense)
```

```
In [ ]: # 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).

```
In [ ]: # View documentation for tl.Embedding
        help(tl.Embedding)
```

```
In [ ]: tmp_embed = tl.Embedding(vocab_size=3, d_feature=2)
        display(tmp_embed)
```

- 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 [ ]: # view the documentation for tl.mean
        help(tl.Mean)
```

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

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

```
In [ ]: help(tl.LogSoftmax)
```

Online documentation

- tl.Dense
- tl.Serial
- tl.Embedding
- tl.Mean
- tl.LogSoftmax

Exercise 05 Implement the classifier function.

```
In [ ]: # 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 [ ]:
```
tmp_model = classifier()
```

In [ ]:
```
print(type(tmp_model))
display(tmp_model)
```

Expected Outout

```
<class 'trax.layers.combinators.Serial'>
Serial[
  Embedding_9088_256
  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 [ ]:
```
# View documentation for trax.supervised.training.TrainTask
help(trax.supervised.training.TrainTask)
```

In [ ]:
```
# View documentation for trax.supervised.training.EvalTask
help(trax.supervised.training.EvalTask)
```

In [ ]:
```
# View documentation for trax.supervised.training.Loop
help(trax.supervised.training.Loop)
```

In [ ]:
```
# View optimizers that you could choose from
help(trax.optimizers)
```

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

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 [ ]: # 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 [ ]: training_loop = train_model(model, train_task, eval_task, 100, output_dir_expand)
```

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 [ ]:
```
# 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
print(f"The shape of the tweet tensors is {tmp_inputs.shape} (num of examples, length o
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 s
```

In [ ]:
```
# 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
```

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 [ ]:
```
# 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 pos
```

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 [ ]:
```
# 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)
```

In [ ]:
```
tmp_pred.shape
```

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

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 [ ]:
```
# 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 int
    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 [ ]: # 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_

        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 to
```

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 6
```

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 [ ]:
```
# 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, targ

        # 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 [ ]:
```
# 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}', )
```

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
```

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

Notice that the model works well even for complex sentences.

1.0.2 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.