Importing Dependencies

Trax framework is much more concise than TensorFlow and PyTorch. It runs on a TensorFlow backend but allows US to train models with one line commands. Trax also runs end to end, allowing you to get data, model and train all with a single terse statements. Trax is good for implementing new state of the art algorithms like Transformers, Reformers, BERT because it is actively maintained by Google Brain Team for advanced deep learning tasks. It runs smoothly on CPUs,GPUs and TPUs as well with comparatively lesser modifications in code.

```
!pip install -q trax
```

```
     |████████████████████████████████| 637 kB 5.3 MB/s
     |████████████████████████████████| 4.6 MB 50.3 MB/s
     |████████████████████████████████| 511.7 MB 5.6 kB/s
     |████████████████████████████████| 438 kB 57.0 MB/s
     |████████████████████████████████| 1.6 MB 50.0 MB/s
     |████████████████████████████████| 5.8 MB 41.1 MB/s
```

```
# import relevant libraries
import string
import re
import os
import nltk
import os
import shutil
import random as rnd
import trax
import trax.fastmath.numpy as np
from trax import layers as tl
from trax import fastmath

nltk.download('stopwords')
from nltk.corpus import stopwords # Stop words are messy and not that compelling;
stopwords_english = stopwords.words('english') # "very" and "not" are considered stop word

nltk.download('twitter_samples')
from nltk.corpus import twitter_samples

from nltk.stem import PorterStemmer # The porter stemmer lemmatizes "was" to "wa".  Seriou
stemmer = PorterStemmer() # Making an object

from nltk.tokenize import TweetTokenizer
tweet_tokenizer = TweetTokenizer(preserve_case=False, strip_handles=True, reduce_len=True)
```

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

```python
def load_tweets():
    all_positive_tweets = twitter_samples.strings('positive_tweets.json')
    all_negative_tweets = twitter_samples.strings('negative_tweets.json')
    return all_positive_tweets, all_negative_tweets

def train_val_split():
    # 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)))


    return train_pos, train_neg, train_x, train_y, val_pos, val_neg, val_x, val_y


train_pos, train_neg, train_x, train_y, val_pos, val_neg, val_x, val_y = train_val_split()
print(f"length of train_x:- {len(train_x)}")
print(f"length of val_x:-  {len(val_x)}")
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and
The number of positive tweets: 5000
The number of negative tweets: 5000
length of train_x:- 8000
length of val_x:-  2000
```

```python
def process_tweet(tweet):
    '''
    Input:
        tweet: a string containing a tweet
    Output:
        tweets_clean: a list of words containing the processed tweet
```

```
    '''
    # remove stock market tickers like $GE
    tweet = re.sub(r'\$\w*', '', tweet)
    # remove old style retweet text "RT"
    tweet = re.sub(r'^RT[\s]+', '', tweet)
    # remove hyperlinks
    tweet = re.sub(r'https?:\/\/.*[\r\n]*', '', tweet)
    # remove hashtags
    # only removing the hash # sign from the word
    tweet = re.sub(r'#', '', tweet)
    # tokenize tweets
    tokenizer = TweetTokenizer(preserve_case=False, strip_handles=True, reduce_len=True)
    tweet_tokens = tokenizer.tokenize(tweet) # ['hi', 'i', 'am', 'vaasu']

    tweets_clean = []
    for word in tweet_tokens:
        if (word not in stopwords_english and # remove stopwords
            word not in string.punctuation): # remove punctuation
            #tweets_clean.append(word)
            stem_word = stemmer.stem(word) # stemming word
            tweets_clean.append(stem_word)

    return tweets_clean

def get_vocab(train_x):

    # 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)

    return Vocab

Vocab = get_vocab(train_x)

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

```
 Total words in vocab are 9089
```

```
def tweet_to_tensor(tweet, vocab_dict, unk_token='__UNK__', verbose=False):
    '''
    Input:
        tweet - A string containing a tweet
        vocab_dict - The words dictionary
        unk_token - The special string for unknown tokens
```

```
        verbose - Print info durign runtime
    Output:
        tensor_l - A python list with

    '''

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

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

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

    # Get the unique integer ID of the __UNK__ token
    unk_ID = vocab_dict.get(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.get(word if word in vocab_dict else unk_token)
    ### END CODE HERE ###

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

    return tensor_l


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:
 [1064, 136, 478, 2351, 744, 8149, 1122, 744, 53, 2, 2671, 790, 2, 2, 348, 600, 2, 3
```

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, we were to train a model with one example at a time, it would take a very long time to train the model. we 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 we create the generator, we could include it in a for loop

for batch_inputs, batch_targets, batch_example_weights in data_generator: ... We 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 we could directly use in our model. It returns a triplet: 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, we will use it to mask input padding.

```
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

    '''
    # 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 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


    # Second part: Pack n_to_take negative examples
```

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

    # If the negative index goes past the negative dataset length,
    if neg_index >= len_data_neg:

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

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

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

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

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

    # Increment neg_index by one
    neg_index = neg_index + 1

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:

    # Get the number of positions to pad for this tensor so that it will be max_le
    n_pad = max_len - len(tensor)
```

```
            # Generate a list of zeros, with length n_pad
            pad_l = [0 for _ in range(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.asarray(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 _ in range(n_to_take)]

        # Generate the list of targets for the negative examples (a list of zeros)
        # The length is the number of negative examples in the batch
        target_neg = [0 for _ in range(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.asarray(target_l)

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


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

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

# Create the training data generator

def train_generator(batch_size, train_pos
                    , train_neg, vocab_dict, loop=True
                    , shuffle = False):
    return data_generator(train_pos, train_neg, batch_size, loop, vocab_dict, shuffle)

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

# Create the validation data generator
def test_generator(batch_size, val_pos
                    , val_neg, vocab_dict, loop=False
                    , shuffle = False):
```

```
    return data_generator(val_pos, val_neg, batch_size, loop, vocab_dict, shuffle)

# Get a batch from the train_generator and inspect.
inputs, targets, example_weights = next(train_generator(4, train_pos, train_neg, Vocab, sh

# 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 4450 3200    9    0    0    0    0    0    0    0]
 [4953  566 2000 1453 5173 3498  141 3498  130  458    9]
 [3760  109  136  582 2929 3968    0    0    0    0    0]
 [ 249 3760    0    0    0    0    0    0    0    0    0]]
Targets: [1 1 0 0]
Example Weights: [1 1 1 1]
```

```
# 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, train_pos=train_pos, train_neg=train_neg, v

# 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_weigh
```

```
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
input tensor: [5737 2900 3760    0    0    0    0    0    0    0    0    0    0    0
input tensor: [ 857  255 3651 5738  306 4457  566 1229 2766  327 1201 3760    0    0
```

Modelling

```
# Layers have weights and a foward function.
# They create weights when layer.initialize is called and use them.
# remove this or make it optional

class Layer(object):
    """Base class for layers."""
    def __init__(self):
        self.weights = None

    def forward(self, x):
        raise NotImplementedError

    def init_weights_and_state(self, input_signature, random_key):
        pass
```

```
    def init(self, input_signature, random_key):
        self.init_weights_and_state(input_signature, random_key)
        return self.weights

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


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
        '''
        activation = np.maximum(x,0)
        return activation


# 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.]]
```

```
# See how the trax.fastmath.random.normal function works
tmp_key = trax.fastmath.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 tensorf
# 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, -0.9699289 ,  1.0070665 ],
             [ 0.3661903 ,  0.1729483 ,  0.29092234]], dtype=float32)
```

```
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*trax.fastmath.random.normal(key=random_key, shape=(input_shap

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

For the model implementation, we will use the Trax layers module, imported as 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 we 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, we will learn more about it in course 4.

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

tl.Dense: Trax Dense layer implementation One other important layer that we will use a lot is one that allows to execute one layer after another in sequence.

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

```
def classifier(vocab_size=len(Vocab), embedding_dim=256, output_dim=2, mode='train'):

    # 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
    )

    # return the model of type
    return model
```

Training

[ ] ↳ *6 cells hidden*

Evaluation

Computing the accuracy on a batch

Writing a function that evaluates the 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

[ ] ↳ *4 cells hidden*

Predicting

[ ] ↳ *2 cells hidden*

Word Embeddings

[ ] ↳ *5 cells hidden*

The word embeddings for this task seem to distinguish negative and positive meanings very well. However, clusters don't necessarily have similar words since you only trained the model to analyze overall sentiment.

On Deep Nets

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

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